NLP Project – Implicit Discourse

Auther: Yiyi Zhang 张億一

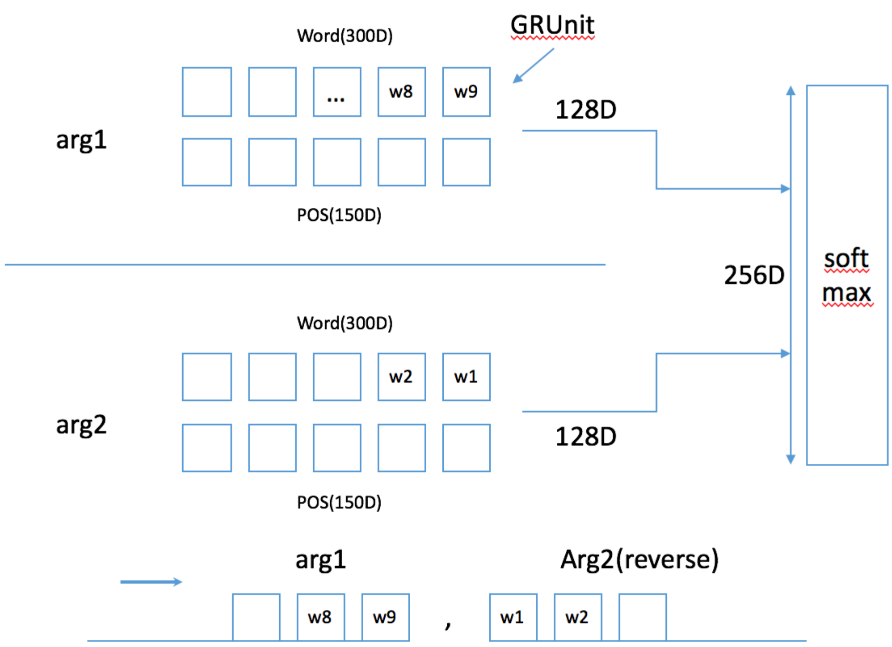
Student no: 5132409031

Email: yi95yi@163.com

ACM Honored Class 2013, ZhiYuan College

Overview

To solve the implicit discourse classification problem, I use GRU model shown as below. For each sentence, get the word and POS from it, the word will be 300D corresponding to the GoogleNews-vector dictionary and the POS information will be saved in a hot vector of 50D, each of them will be served as the unit of GRU. One arg will output a 128D hidden layer, as there are two arg, the input of softmax will be 256D, the output of softmax is the 12 labels, also known as the 12 kinds of sense.



pic 1. GRU model

There are 3 files in this project:

- Preparedata.py
  Process the data and reformat them into ideal form
    - Vocal.pkl: store the distinct words from the dataset
    - Pos_dict.pkl: store the POS info
    - Google_emb.npy: GoogleNews-vector dictionary
    - Discourse.pkl: store the vector of 10000 most freq word according to GoogleNews-vector dictionary
- GRU_pos_slice.py: training step, build the model
  Model parameter: w.npy, w2.npy, b.npy, br2.npy, mlp_w.npy, mlp_b.npy, output_w.npy, output_b.npy
- Classifier_pos.py: test the model

When applying on dev_pdtb.json, the model has the accuracy of 44.0777%

```
zhangyiyideMacBook-Pro:implicitDiscourseorg zhangyiyi$ python newscorer.py dev_pdtb.json dev_result.json

================================================
Acc: 0.440777
```

Optimization:

Argument 2 is in the reverse order for the content close to the comma is usually more important

I. Prepare Data

In preparing data, there are four phases: build vocabulary, build pos dict, Google word2vec and build sentence vector. From the dataset, we select the word, pos and label.

For words, we choose 10000 most frequent words in the dataset and store them in vocal.pkl, using Google word2vec as pretrained dictionary, if the word is in the dictionary, return the corresponding word vector, else return “UNKNOWN”, and store them in the discourse.pkl.

For pos, we use a 50D hot vector to store them in pos_dict.pkl, shown as

50

NNP	**1**	**0**	**0**	**0**	**...**
VB	**0**	**1**	**0**	**0**	**...**
JJ	**0**	**0**	**1**	**0**	**...**
...					

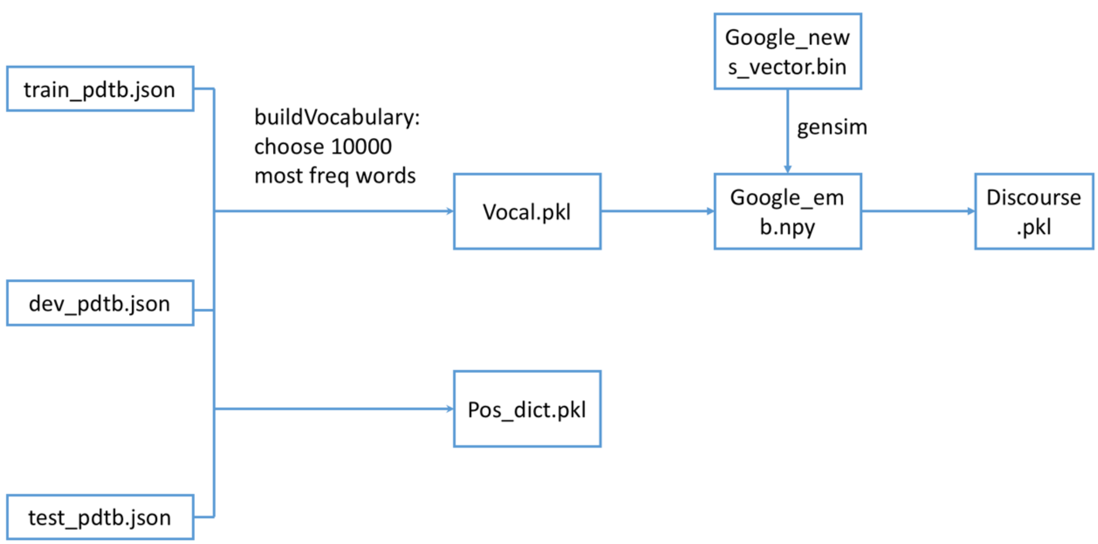
pic 1. the way to store pos

The workflow is shown below:

pic2. Overview of prepare data

A. Build vocabulary

First, we load the things we need: arg1, arg2, pos1, pos2 and label from the datasets into sentence.



```python
def loadFile(filename):
    sentences_arg1 = []
    sentences_arg2 = []
    sentences_pos1 = []
    sentences_pos2 = []
    label = []
    with open(filename) as file:
        lines = file.readlines()
        for line in lines:
            data = MyDecoder().decode(line)
            sense = data.get('Sense')[0]
            if sense != 'EntRel':
                arg1 = [replace(t) for t in data.get('Arg1').get('Word')]
                arg2 = [replace(t) for t in data.get('Arg2').get('Word')]
                pos1 = data.get('Arg1').get('POS')
                pos2 = data.get('Arg2').get('POS')
                sentences_arg1.append(arg1)
                sentences_arg2.append(arg2)
                sentences_pos1.append(pos1)
                sentences_pos2.append(pos2)
                label.append(sense)
        file.close()
    return sentences_arg1, sentences_arg2, label, sentences_pos1, sentences_pos2
```

Then, we scan all the words in the dataset, sort them according to their frequency. We find the most 10000 frequent word and print out the 9999th frequent word and its occur time, write them to vocal.pkl

```python
def BuildVocabulary(token_sentences, size):
    word_freq = {}
    for sentence in token_sentences:
        for word in sentence:
            if word in word_freq:
                word_freq[word] += 1
            else:
                word_freq[word] = 1
    vocab = sorted(word_freq.items(), key=lambda x: (x[1]), reverse=True)
    if size > len(vocab):
        size = len(vocab)
    print 'Saw %d distinct words, the most %dth freq word is %s, occur %d' % (
        len(vocab), size - 1, vocab[size - 2][0], vocab[size - 2][1])
    vocab = vocab[:size - 1]
    ind2word = ['<UNKNOW>'] + [pair[0] for pair in vocab]
    word2ind = dict([(w, i) for i, w in enumerate(ind2word)])
    return ind2word, word2ind
```

We can get the result when the input is train_pdtb, dev_pdtb, test_pdtb:

```
Saw 29633 distinct words, the most 9999th freq word is commanders, occur 4
```

B. Build pos_dict

We use a 50D vector to store POS info of the arg in pos_dict.pkl, also use function BuildVocabulary, as follows:

```
ind2pos, pos2ind = BuildVocabulary(pos, 50)
data = ind2pos, pos2ind
pickle.dump(data, open('pos_dict.pkl', 'wb'))
```

C. Google word2vec

GoogleNews-vectors-300.bin contains 300-dimensional vectors for 3 million words and phrases. We view it as the dictionary and by using genism word2vec, we can get the corresponding 300D vector of the word in vocal.pkl

1. Install and using genism to get the word vector from GoogleNews-vectors-300.bin

```
# install genism
easy_install numpy
easy_install scipy
```

```
easy_install –upgrade genism
```

Below is code using genism to get the wordvec and store them in GVector.txt:

2. Build Google_emb.npy

Like the way to store the pos dict, we first initialize the numpy of Google_emb to be all zero. As the vector in GoogleNews has 300D, the numpy also has 300D. Scan all the word in vocal.pkl, and look them up in GoogleNews dictionary, if they are in, return the vector and replace the numpy in Google_emb.npy, else, still remain zero.

```python
import numpy
from gensim.models import *
from gensim import matutils

model = Word2Vec.load_word2vec_format('Desktop/NLP/GoogleNews-vectors-negative300.bin', binary=True)

f = open('Desktop/NLP/implicitDiscourse/vocal.txt','r')
fw = open('Desktop/NLP/implicitDiscourse/GoogleWordVector/GVector.txt','w')
while 1:
    words = f.readline().split(' ')
    if words:
        for word in words:
            try:
                ary = model[word]
                #ary = word
                fw.write(str(ary))
                fw.write(' ')
                fw.write('\r')
            except:
                continue
    else:
        break
f.close()
fw.close()
```

```python
def BuildGoogleEmb(word2ind, gvector_filename):
    vectors = [numpy.zeros(300).astype(dtype=numpy.float32)] * len(word2ind)
    with open(gvector_filename) as file:
        lines = file.readlines()
        for line in lines[1:]:
            splits = line.split(' ')
            if splits[0] in word2ind:
                vectors[word2ind[splits[0]]] = [float(x) for x in splits[1:301]]
        file.close()
    for word in word2ind.keys():
        if len(vectors[word2ind[word]]) != 300:
            print word
    return numpy.asarray(vectors, dtype=numpy.float32)
```

Then, save them to google_emb.npy

```python
numpy.save(open('google_emb.npy', 'w'), BuildGoogleEmb(word2ind, 'GoogleWordVector/GVector.txt'))
```

D. Build sentence vector

Reformat to [[arg1,…,…],[arg2,…,…],label], code as follows:

```python
def reformat(arg1, arg2, labels, word2ind, label_char):
    test = []
    for a1, a2, l in zip(arg1, arg2, labels):
        t1 = [vectorize(w, word2ind) for w in a1]
        t2 = [vectorize(w, word2ind) for w in a2]
        for ind, la in enumerate(label_char):
            if l == la:
                test.append([t1, t2, ind])
    return test
```

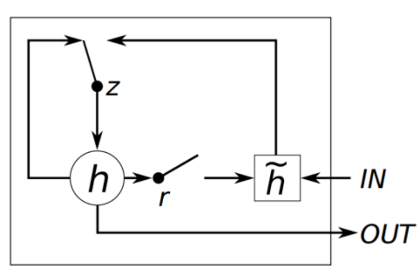
Then, in the main, reformat the train_data, test_data, and write them back to discourse.pkl

```python
    train_data = reformat(train_1, train_2, train_l, word2ind, label_char)
    test_data = reformat(test_1, test_2, test_l, word2ind, label_char)
    pickle.dump([train_data, test_data], open('discourse.pkl', 'wb'))
```

II. GRU Train

A. GRU model

A gated recurrent unit (GRU) was proposed by Cho et al. to make each recurrent unit to adaptively capture dependencies of different time scales.



Similarly to the LSTM unit, GRU has gating units that modulate the flow of information inside the unit, however, without having a separate memory cells.

The update equation:

$$
\begin{aligned}
r_t &= \sigma(x_t W_{xr} + h_{t-1} W_{hr} + b_r) \\
z_t &= \sigma(x_t W_{xz} + h_{t-1} W_{hz} + b_z) \\
\widetilde{h_t} &= g(x_t W_{xh} + (r_t \odot h_{t-1}) W_{hh} + b_h) \\
h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \widetilde{h_t}
\end{aligned}
$$

$g(.)$ is the activation function for the layer, $\sigma(.)$ is the logistic sigmoid, which ensures that the two gates in the layer are limited to the open interval (0, 1). $\odot$ indicates elementwise multiplication.

Parameters:

- bh — vector of bias values for each hidden unit
- br — vector of reset biases
- bz — vector of rate biases
- xh — matrix connecting inputs to hidden units
- xr — matrix connecting inputs to reset gates
- xz — matrix connecting inputs to rate gates
- hh — matrix connecting hiddens to hiddens
- hr — matrix connecting hiddens to reset gates
- hz — matrix connecting hiddens to rate gates

Outputs

- out — the post-activation state of the layer
- pre — the pre-activation state of the layer
- hid — the pre-rate-mixing hidden state
- rate — the rate values

The code is as follows:

```
def encoder(word, h_tm1, w, b):
            x_t = self.iemb[word[0]]
            p_t = self.posv[word[1]]
            wr = slice(w[0])
            wz = slice(w[1])
            wh = slice(w[2])
            r_t = T.nnet.sigmoid(T.dot(x_t, wr[0]) + T.dot(p_t, wr[1]) + T.dot(h_tm1, wr[2]) + b[0])
            z_t = T.nnet.sigmoid(T.dot(x_t, wz[0]) + T.dot(p_t, wz[1]) + T.dot(h_tm1, wz[2]) + b[1])
            uh_tm1 = h_tm1 * r_t
            uh_t = T.tanh(T.dot(x_t, wh[0]) + T.dot(p_t, wh[1]) + T.dot(uh_tm1, wh[2]) + b[2])
            h_t = h_tm1 * z_t + uh_t * (1 - z_t)
            return h_t

encode_process, _ = theano.scan(fn=encoder, sequences=arg1, outputs_info=dict(initial=T.zeros(n_hidden)),
                                        non_sequences=[self.w, self.b])
        arg1_code = encode_process[-1]

        encode_process, _ = theano.scan(fn=encoder, sequences=arg2, outputs_info=dict(initial=T.zeros(n_hidden)),
                                        non_sequences=[self.w2, self.b2], go_backwards=True)
        arg2_code = encode_process[-1]

        con = T.concatenate([arg1_code, arg2_code])
        internal = T.tanh(T.dot(con, self.mlp_w) + self.mlp_b)
        pred = T.nnet.softmax(T.dot(internal, self.output_w) + self.output_b)[0]

        nll = -T.log(pred[label])

        # SGD
        grad = T.grad(nll, self.params)
        grad_updates = [(param, param - lr * grad - l2_regular * param) for param, grad, l2_regular in
                        zip(self.params, grad, self.l2mask)]

        self.sentence_train = theano.function(inputs=[arg1, arg2, label, lr], outputs=nll,
                                              updates=grad_updates)
        self.sentence_error = theano.function(inputs=[arg1, arg2, label], outputs=nll)
        self.predict = theano.function(inputs=[arg1, arg2], outputs=T.argmax(pred))
```

B. Program parameter

```
    prog_para = {
        'n_hidden': 128,
        'n_input_char': 10000,
        'n_emb': 300,
        'n_pos': 50,
        'model_name': 'w2v_pos_slice',
        'pre_trained_w2v': 'google_emb.npy',

        'print_freq': 1000,
        'num_epoch': 1000,
        'num_case_in_epoch': 100000
    }
```

C. Build model

```
    model = GRU(n_hidden=prog_para['n_hidden'], n_input_char=prog_para['n_input_char'],
                n_pos=prog_para['n_pos'], n_emb=prog_para['n_emb'])
    model.loadmodel(prog_para['model_name'])
    model.loadWordVector(prog_para['pre_trained_w2v'])
```

D. Trainining

```
bestacc = 0.
    for epoch in range(prog_para['num_epoch']):
        epoch_tic = timeit.default_timer()
        numpy.random.shuffle(train_data)
        pos_split = len(train_data) / 10
        t_data, v_data = train_data[pos_split:], train_data[:pos_split]

        cnt = 0
        train_err = 0.
        for [arg1, arg2, y] in t_data:
            train_err += model.sentence_train(arg1, arg2, y, 0.01)
            cnt += 1
            if cnt % prog_para['print_freq'] == 0:
                print train_err / cnt
            if cnt == prog_para['num_case_in_epoch']:
                break
```

E. Validate and Test

```
        # validate
        v_err = 0.
        for [arg1, arg2, y] in v_data:
            v_err += model.sentence_error(arg1, arg2, y)

        # test
        t_err = 0.
        acc = 0.
        for [arg1, arg2, y] in test_data:
            t_err += model.sentence_error(arg1, arg2, y)
            pred = model.predict(arg1, arg2)
            if pred == y:
                acc += 1
```

F. Save model

```python
if acc / len(test_data) > bestacc:
            model.savemodel(prog_para['model_name'])
            bestacc = acc / len(test_data)
        print '[trainning] epoch complete in %.2f, train_err=%.6f, validate_err=%.6f, test_err=%.6f, test_acc=%.6f,epoch=%d ' \
              % (timeit.default_timer() - epoch_tic, train_err / cnt, v_err / len(v_data), t_err / len(test_data),
                 acc / len(test_data), epoch)
```

III. Classify

Test the data and export to dev_result.json. Code as follows:

```python
acc = 0.
    cnt = 0.

    fp = open('dev_pdtb.json','r')
    fw = open('dev_result.json','w')
    fp_word = fp.readlines()
    for fp_simple in fp_word:
        dict = json.loads(fp_simple)
        if dict['Type'] == 'Implicit' :
            cnt += 1
            arg1 = dict.get('Arg1').get('Word')
            arg2 = dict.get('Arg2').get('Word')
            sense = dict['Sense']
            pos1 = dict.get('Arg1').get('POS')
            pos2 = dict.get('Arg2').get('POS')

            t1 = [[preparedata.vectorize(w, word2ind), pos2ind[p]] for w, p in zip(arg1, pos1)]
            t2 = [[preparedata.vectorize(w, word2ind), pos2ind[p]] for w, p in zip(arg2, pos2)]

            pred = model.predict(t1, t2)

            if [label_char[pred]] == sense:
                print 'predict: %s, object: %s -------------' % (label_char[pred], sense)
                acc += 1
            else:
                print 'predict: %s, object: %s' % (label_char[pred], sense)

            dict['Sense'] = [label_char[pred]]
            print dict['Sense']
            fw.write(json.dumps(dict)+"\n")
```