Recognize named entities on Twitter with LSTMs

In this assignment, you will use a recurrent neural network to solve Named Entity Recognition (NER) problem. NER is a common task in natural language processing systems. It serves for extraction such entities from the text as persons, organizations, locations, etc. In this task you will experiment to recognize named entities from Twitter.

For example, we want to extract persons' and organizations' names from the text. Than for the input text:

```
Ian Goodfellow works for Google Brain
```

a NER model needs to provide the following sequence of tags:

```
B-PER I-PER    O     O   B-ORG  I-ORG
```

Where *B-* and *I-* prefixes stand for the beginning and inside of the entity, while *O* stands for out of tag or no tag. Markup with the prefix scheme is called *BIO markup*. This markup is introduced for distinguishing of consequent entities with similar types.

A solution of the task will be based on neural networks, particularly, on Bi-Directional Long Short-Term Memory Networks (Bi-LSTMs).

Libraries

For this task you will need the following libraries:

- Tensorflow (https://www.tensorflow.org) — an open-source software library for Machine Intelligence.
- Numpy (http://www.numpy.org) — a package for scientific computing.

If you have never worked with Tensorflow, you would probably need to read some tutorials during your work on this assignment, e.g. this one (https://www.tensorflow.org/tutorials/recurrent) could be a good starting point.

Data

The following cell will download all data required for this assignment into the folder `week2/data`.

In [33]:
```
import sys
sys.path.append("..")
from common.download_utils import download_week2_resources

download_week2_resources()
```

```
File data/train.txt is already downloaded.
File data/validation.txt is already downloaded.
File data/test.txt is already downloaded.
```

Load the Twitter Named Entity Recognition corpus

We will work with a corpus, which contains twits with NE tags. Every line of a file contains a pair of a token (word/punctuation symbol) and a tag, separated by a whitespace. Different tweets are separated by an empty line.

The function *read_data* reads a corpus from the *file_path* and returns two lists: one with tokens and one with the corresponding tags. You need to complete this function by adding a code, which will replace a user's nickname to `<USR>` token and any URL to `<URL>` token. You could think that a URL and a nickname are just strings which start with *http://* or *https://* in case of URLs and a @ symbol for nicknames.

In [34]:
```
def read_data(file_path):
    tokens = []
    tags = []

    tweet_tokens = []
    tweet_tags = []
    for line in open(file_path, encoding='utf-8'):
        line = line.strip()
        if not line:
            if tweet_tokens:
                tokens.append(tweet_tokens)
                tags.append(tweet_tags)
            tweet_tokens = []
            tweet_tags = []
        else:
            token, tag = line.split()
            # Replace all urls with <URL> token
            # Replace all users with <USR> token

            ######################################
            ######### YOUR CODE HERE #############
            ######################################
            if (token.startswith('@')):
                token = '<USR>'
            elif token.startswith('http:') or token.startswith('https:'):
                token = '<URL>'

            tweet_tokens.append(token)
            tweet_tags.append(tag)

    return tokens, tags
```

And now we can load three separate parts of the dataset:

- *train* data for training the model;
- *validation* data for evaluation and hyperparameters tuning;
- *test* data for final evaluation of the model.

In [35]:
```
train_tokens, train_tags = read_data('data/train.txt')
validation_tokens, validation_tags = read_data('data/validation.txt')
test_tokens, test_tags = read_data('data/test.txt')
```

You should always understand what kind of data you deal with. For this purpose, you can print the data running the following cell:

In [36]:
```
for i in range(3):
    for token, tag in zip(train_tokens[i], train_tags[i]):
        print('%s\t%s' % (token, tag))
    print()
```

```
RT      O
<USR>   O
:       O
Online  O
ticket  O
sales   O
for     O
Ghostland       B-musicartist
Observatory     I-musicartist
extended        O
until   O
6       O
PM      O
EST     O
due     O
to      O
high    O
demand  O
.       O
Get     O
them    O
before  O
they    O
sell    O
out     O
...     O

Apple   B-product
MacBook I-product
Pro     I-product
A1278   I-product
13.3    I-product
"       I-product
Laptop  I-product
-       I-product
MD101LL/A       I-product
(       O
June    O
,       O
2012    O
)       O
-       O
Full    O
read    O
by      O
eBay    B-company
<URL>   O
<URL>   O

Happy   O
Birthday        O
<USR>   O
!       O
May     O
Allah   B-person
s.w.t   O
bless   O
```

```
you     O
with    O
goodness        O
and     O
happiness       O
.       O
```

Prepare dictionaries

To train a neural network, we will use two mappings:

- {token}$\rightarrow${token id}: address the row in embeddings matrix for the current token;
- {tag}$\rightarrow${tag id}: one-hot ground truth probability distribution vectors for computing the loss at the output of the network.

Now you need to implement the function *build_dict* which will return {token or tag}$\rightarrow${index} and vice versa.

In [37]:
```
from collections import defaultdict
```

In [38]:
```
def build_dict(tokens_or_tags, special_tokens):
    """
        tokens_or_tags: a list of lists of tokens or tags
        special_tokens: some special tokens
    """
    # Create a dictionary with default value 0
    tok2idx = defaultdict(lambda: 0)
    idx2tok = []

    # Create mappings from tokens (or tags) to indices and vice versa.
    # At first, add special tokens (or tags) to the dictionaries.
    # The first special token must have index 0.

    # Mapping tok2idx should contain each token or tag only once.
    # To do so, you should:
    # 1. extract unique tokens/tags from the tokens_or_tags variable, which
 is not
    #    occure in special_tokens (because they could have non-empty inters
ection)
    # 2. index them (for example, you can add them into the list idx2tok
    # 3. for each token/tag save the index into tok2idx).

    ########################################
    ######### YOUR CODE HERE #############
    ########################################

    idx = 0
    for token in special_tokens:
        idx2tok.append(token)
        tok2idx[token] = idx
        idx += 1

    for token_list in tokens_or_tags:
        for token in token_list:
            if token not in tok2idx:
                idx2tok.append(token)
                tok2idx[token] = idx
                idx += 1

    return tok2idx, idx2tok
```

After implementing the function *build_dict* you can make dictionaries for tokens and tags. Special tokens in our case will be:

- `<UNK>` token for out of vocabulary tokens;
- `<PAD>` token for padding sentence to the same length when we create batches of sentences.

In [39]:
```
special_tokens = ['<UNK>', '<PAD>']
special_tags = ['O']

# Create dictionaries
token2idx, idx2token = build_dict(train_tokens + validation_tokens, special_to
kens)
tag2idx, idx2tag = build_dict(train_tags, special_tags)
```

The next additional functions will help you to create the mapping between tokens and ids for a sentence.

In [40]:
```
def words2idxs(tokens_list):
    return [token2idx[word] for word in tokens_list]

def tags2idxs(tags_list):
    return [tag2idx[tag] for tag in tags_list]

def idxs2words(idxs):
    return [idx2token[idx] for idx in idxs]

def idxs2tags(idxs):
    return [idx2tag[idx] for idx in idxs]
```

Generate batches

Neural Networks are usually trained with batches. It means that weight updates of the network are based on several sequences at every single time. The tricky part is that all sequences within a batch need to have the same length. So we will pad them with a special `<PAD>` token. It is also a good practice to provide RNN with sequence lengths, so it can skip computations for padding parts. We provide the batching function *batches_generator* readily available for you to save time.

In [41]:
```
def batches_generator(batch_size, tokens, tags,
                      shuffle=True, allow_smaller_last_batch=True):
    """Generates padded batches of tokens and tags."""

    n_samples = len(tokens)
    if shuffle:
        order = np.random.permutation(n_samples)
    else:
        order = np.arange(n_samples)

    n_batches = n_samples // batch_size
    if allow_smaller_last_batch and n_samples % batch_size:
        n_batches += 1

    for k in range(n_batches):
        batch_start = k * batch_size
        batch_end = min((k + 1) * batch_size, n_samples)
        current_batch_size = batch_end - batch_start
        x_list = []
        y_list = []
        max_len_token = 0
        for idx in order[batch_start: batch_end]:
            x_list.append(words2idxs(tokens[idx]))
            y_list.append(tags2idxs(tags[idx]))
            max_len_token = max(max_len_token, len(tags[idx]))

        # Fill in the data into numpy nd-arrays filled with padding indices.
        x = np.ones([current_batch_size, max_len_token], dtype=np.int32) * tok
en2idx['<PAD>']
        y = np.ones([current_batch_size, max_len_token], dtype=np.int32) * tag
2idx['O']
        lengths = np.zeros(current_batch_size, dtype=np.int32)
        for n in range(current_batch_size):
            utt_len = len(x_list[n])
            x[n, :utt_len] = x_list[n]
            lengths[n] = utt_len
            y[n, :utt_len] = y_list[n]
        yield x, y, lengths
```

Build a recurrent neural network

This is the most important part of the assignment. Here we will specify the network architecture based on TensorFlow building blocks. It's fun and easy as a lego constructor! We will create an LSTM network which will produce probability distribution over tags for each token in a sentence. To take into account both right and left contexts of the token, we will use Bi-Directional LSTM (Bi-LSTM). Dense layer will be used on top to perform tag classification.

In [42]:
```
import tensorflow as tf
import numpy as np
```

In [43]:
```
class BiLSTMModel():
    pass
```

First, we need to create placeholders (https://www.tensorflow.org/versions/master/api_docs/python/tf/placeholder) to specify what data we are going to feed into the network during the execution time. For this task we will need the following placeholders:

- *input_batch* — sequences of words (the shape equals to [batch_size, sequence_len]);
- *ground_truth_tags* — sequences of tags (the shape equals to [batch_size, sequence_len]);
- *lengths* — lengths of not padded sequences (the shape equals to [batch_size]);
- *dropout_ph* — dropout keep probability; this placeholder has a predefined value 1;
- *learning_rate_ph* — learning rate; we need this placeholder because we want to change the value during training.

It could be noticed that we use *None* in the shapes in the declaration, which means that data of any size can be feeded.

You need to complete the function *declare_placeholders*.

In [44]:
```
def declare_placeholders(self):
    """Specifies placeholders for the model."""

    # Placeholders for input and ground truth output.
    self.input_batch = tf.placeholder(dtype=tf.int32, shape=[None, None], name
='input_batch')
    self.ground_truth_tags = tf.placeholder(dtype=tf.int32, shape=[None, None
], name='ground_truth_tags')

    # Placeholder for lengths of the sequences.
    self.lengths = tf.placeholder(dtype=tf.int32, shape=[None], name='lengths'
)

    # Placeholder for a dropout keep probability. If we don't feed
    # a value for this placeholder, it will be equal to 1.0.
    self.dropout_ph = tf.placeholder_with_default(tf.cast(1.0, tf.float32), sh
ape=[])

    # Placeholder for a learning rate (tf.float32).
    self.learning_rate_ph = tf.placeholder(dtype=tf.float32, name='learning_ph
_rate')
```

In [45]:
```
BiLSTMModel.__declare_placeholders = classmethod(declare_placeholders)
```

Now, let us specify the layers of the neural network. First, we need to perform some preparatory steps:

- Create embeddings matrix with tf.Variable (https://www.tensorflow.org/api_docs/python/tf/Variable). Specify its name (*embeddings_matrix*), type (*tf.float32*), and initialize with random values.
- Create forward and backward LSTM cells. TensorFlow provides a number of RNN cells (https://www.tensorflow.org/api_guides/python/contrib.rnn#Core_RNN_Cells_for_use_with_TensorFlow_s_cor ready for you. We suggest that you use *BasicLSTMCell*, but you can also experiment with other types, e.g. GRU cells. This (http://colah.github.io/posts/2015-08-Understanding-LSTMs/) blogpost could be interesting if you want to learn more about the differences.
- Wrap your cells with DropoutWrapper (https://www.tensorflow.org/api_docs/python/tf/contrib/rnn/DropoutWrapper). Dropout is an important regularization technique for neural networks. Specify all keep probabilities using the dropout placeholder that we created before.

After that, you can build the computation graph that transforms an input_batch:

- Look up (https://www.tensorflow.org/api_docs/python/tf/nn/embedding_lookup) embeddings for an *input_batch* in the prepared *embedding_matrix*.
- Pass the embeddings through Bidirectional Dynamic RNN (https://www.tensorflow.org/api_docs/python/tf/nn/bidirectional_dynamic_rnn) with the specified forward and backward cells. Use the lengths placeholder here to avoid computations for padding tokens inside the RNN.
- Create a dense layer on top. Its output will be used directly in loss function.

Fill in the code below. In case you need to debug something, the easiest way is to check that tensor shapes of each step match the expected ones.

In [46]:
```
def build_layers(self, vocabulary_size, embedding_dim, n_hidden_rnn, n_tags
):
    """Specifies bi-LSTM architecture and computes logits for inputs."""

    # Create embedding variable (tf.Variable) with dtype tf.float32
    initial_embedding_matrix = np.random.randn(vocabulary_size, embedding_d
im) / np.sqrt(embedding_dim)

    embedding_matrix_variable = tf.Variable(dtype=tf.float32,
                                            initial_value=initial_embedding
_matrix,
                                            name="embeddings_matrix")

    # Create RNN cells (for example, tf.nn.rnn_cell.BasicLSTMCell) with n_h
idden_rnn number of units
    # and dropout (tf.nn.rnn_cell.DropoutWrapper), initializing all *_keep_
prob with dropout placeholder.
    forward_cell =  tf.nn.rnn_cell.DropoutWrapper(tf.nn.rnn_cell.BasicLSTMC
ell(n_hidden_rnn),
                                                  input_keep_prob=self.drop
out_ph,
                                                  output_keep_prob=self.dro
pout_ph,
                                                  state_keep_prob=self.drop
out_ph)

    ######### YOUR CODE HERE #############
    backward_cell = tf.nn.rnn_cell.DropoutWrapper(tf.nn.rnn_cell.BasicLSTMC
ell(n_hidden_rnn),
                                                  input_keep_prob=self.drop
out_ph,
                                                  output_keep_prob=self.dro
pout_ph,
                                                  state_keep_prob=self.drop
out_ph)

    # Look up embeddings for self.input_batch (tf.nn.embedding_lookup).
    # Shape: [batch_size, sequence_len, embedding_dim].
    embeddings = tf.nn.embedding_lookup(embedding_matrix_variable, self.inp
ut_batch) ######### YOUR CODE HERE #############

    # Pass them through Bidirectional Dynamic RNN (tf.nn.bidirectional_dyna
mic_rnn).
    # Shape: [batch_size, sequence_len, 2 * n_hidden_rnn].
    # Also don't forget to initialize sequence_length as self.lengths and d
type as tf.float32.
    (rnn_output_fw, rnn_output_bw), _ =  tf.nn.bidirectional_dynamic_rnn(ce
ll_fw=forward_cell,
                                                                         ce
ll_bw=backward_cell,
                                                                         in
puts=embeddings,
                                                                         se
quence_length=self.lengths,
                                                                         dt
ype=tf.float32)
```

```
    rnn_output = tf.concat([rnn_output_fw, rnn_output_bw], axis=2)

    # Dense layer on top.
    # Shape: [batch_size, sequence_len, n_tags].
    self.logits = tf.layers.dense(rnn_output, n_tags, activation=None)
```

In [47]:
```
BiLSTMModel.__build_layers = classmethod(build_layers)
```

To compute the actual predictions of the neural network, you need to apply softmax (https://www.tensorflow.org/api_docs/python/tf/nn/softmax) to the last layer and find the most probable tags with argmax (https://www.tensorflow.org/api_docs/python/tf/argmax).

In [48]:
```
def compute_predictions(self):
    """Transforms logits to probabilities and finds the most probable tags."""

    # Create softmax (tf.nn.softmax) function
    softmax_output = tf.nn.softmax(self.logits)


    # Use argmax (tf.argmax) to get the most probable tags
    # Don't forget to set axis=-1
    # otherwise argmax will be calculated in a wrong way
    self.predictions = tf.argmax(softmax_output, axis=-1)
```

In [49]:
```
BiLSTMModel.__compute_predictions = classmethod(compute_predictions)
```

During training we do not need predictions of the network, but we need a loss function. We will use cross-entropy loss (http://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html#cross-entropy), efficiently implemented in TF as cross entropy with logits (https://www.tensorflow.org/api_docs/python/tf/nn/softmax_cross_entropy_with_logits). Note that it should be applied to logits of the model (not to softmax probabilities!). Also note, that we do not want to take into account loss terms coming from `<PAD>` tokens. So we need to mask them out, before computing mean (https://www.tensorflow.org/api_docs/python/tf/reduce_mean).

In [50]:
```
def compute_loss(self, n_tags, PAD_index):
    """Computes masked cross-entopy loss with logits."""

    # Create cross entropy function function (tf.nn.softmax_cross_entropy_w
ith_logits)
    ground_truth_tags_one_hot = tf.one_hot(self.ground_truth_tags, n_tags)
    loss_tensor =  tf.nn.softmax_cross_entropy_with_logits(labels=ground_tr
uth_tags_one_hot, logits=self.logits)
    ######### YOUR CODE HERE #############

    mask = tf.cast(tf.not_equal(self.input_batch, PAD_index), tf.float32)
    # Create loss function which doesn't operate with <PAD> tokens (tf.redu
ce_mean)
    # Be careful that the argument of tf.reduce_mean should be
    # multiplication of mask and loss_tensor.
    self.loss = tf.reduce_mean(loss_tensor * mask)
    #tf.reduce_mean(tf.reduce_sum(tf.multiply(loss_tensor, mask), axis=-1)
 / tf.reduce_sum(mask, axis=-1))
#    tf.reduce_mean(loss_tensor * mask)  ######### YOUR CODE HERE #########
####
```

In [51]:
```
BiLSTMModel.__compute_loss = classmethod(compute_loss)
```

The last thing to specify is how we want to optimize the loss. We suggest that you use Adam (https://www.tensorflow.org/api_docs/python/tf/train/AdamOptimizer) optimizer with a learning rate from the corresponding placeholder. You will also need to apply clipping (https://www.tensorflow.org/versions/r0.12/api_docs/python/train/gradient_clipping) to eliminate exploding gradients. It can be easily done with clip_by_norm (https://www.tensorflow.org/api_docs/python/tf/clip_by_norm) function.

In [64]:
```
def perform_optimization(self):
    """Specifies the optimizer and train_op for the model."""

    # Create an optimizer (tf.train.AdamOptimizer)
    self.optimizer =tf.train.AdamOptimizer(learning_rate=self.learning_rate_ph
)
    ######### YOUR CODE HERE #############
    self.grads_and_vars = self.optimizer.compute_gradients(self.loss)

    # Gradient clipping (tf.clip_by_norm) for self.grads_and_vars
    # Pay attention that you need to apply this operation only for gradients
    # because self.grads_and_vars contains also variables.
    # list comprehension might be useful in this case.
    clip_norm = tf.cast(1.0, tf.float32)
    self.grads_and_vars =  [(tf.clip_by_norm(grad, clip_norm), var) for grad,
var in self.grads_and_vars]
    ######### YOUR CODE HERE #############

    self.train_op = self.optimizer.apply_gradients(self.grads_and_vars)
```

In [65]:
```
BiLSTMModel.__perform_optimization = classmethod(perform_optimization)
```

Congratulations! You have specified all the parts of your network. You may have noticed, that we didn't deal with any real data yet, so what you have written is just recipes on how the network should function. Now we will put them to the constructor of our Bi-LSTM class to use it in the next section.

In [55]:
```
def init_model(self, vocabulary_size, n_tags, embedding_dim, n_hidden_rnn, PAD
_index):
    self.__declare_placeholders()
    self.__build_layers(vocabulary_size, embedding_dim, n_hidden_rnn, n_tags)
    self.__compute_predictions()
    self.__compute_loss(n_tags, PAD_index)
    self.__perform_optimization()
```

In [56]:
```
BiLSTMModel.__init__ = classmethod(init_model)
```

Train the network and predict tags

Session.run (https://www.tensorflow.org/api_docs/python/tf/Session#run) is a point which initiates computations in the graph that we have defined. To train the network, we need to compute *self.train_op*, which was declared in *perform_optimization*. To predict tags, we just need to compute *self.predictions*. Anyway, we need to feed actual data through the placeholders that we defined before.

In [57]:
```
def train_on_batch(self, session, x_batch, y_batch, lengths, learning_rate, dr
opout_keep_probability):
    feed_dict = {self.input_batch: x_batch,
                 self.ground_truth_tags: y_batch,
                 self.learning_rate_ph: learning_rate,
                 self.dropout_ph: dropout_keep_probability,
                 self.lengths: lengths}

    session.run(self.train_op, feed_dict=feed_dict)
```

In [58]:
```
BiLSTMModel.train_on_batch = classmethod(train_on_batch)
```

Implement the function *predict_for_batch* by initializing *feed_dict* with input *x_batch* and *lengths* and running the *session* for *self.predictions*.

In [59]:
```
def predict_for_batch(self, session, x_batch, lengths):
    ######################################
    ######### YOUR CODE HERE #############
    ######################################
    feed_dict = {
        self.input_batch: x_batch,
        self.lengths: lengths
    }
    predictions = session.run(self.predictions, feed_dict=feed_dict)

    return predictions
```

In [60]:
```
BiLSTMModel.predict_for_batch = classmethod(predict_for_batch)
```

We finished with necessary methods of our BiLSTMModel model and almost ready to start experimenting.

Evaluation

To simplify the evaluation process we provide two functions for you:

- *predict_tags*: uses a model to get predictions and transforms indices to tokens and tags;
- *eval_conll*: calculates precision, recall and F1 for the results.

In [61]:
```
from evaluation import precision_recall_f1
```

In [62]:
```
def predict_tags(model, session, token_idxs_batch, lengths):
    """Performs predictions and transforms indices to tokens and tags."""

    tag_idxs_batch = model.predict_for_batch(session, token_idxs_batch, length
s)

    tags_batch, tokens_batch = [], []
    for tag_idxs, token_idxs in zip(tag_idxs_batch, token_idxs_batch):
        tags, tokens = [], []
        for tag_idx, token_idx in zip(tag_idxs, token_idxs):
            tags.append(idx2tag[tag_idx])
            tokens.append(idx2token[token_idx])
        tags_batch.append(tags)
        tokens_batch.append(tokens)
    return tags_batch, tokens_batch


def eval_conll(model, session, tokens, tags, short_report=True):
    """Computes NER quality measures using CONLL shared task script."""

    y_true, y_pred = [], []
    for x_batch, y_batch, lengths in batches_generator(1, tokens, tags):
        tags_batch, tokens_batch = predict_tags(model, session, x_batch, lengt
hs)
        if len(x_batch[0]) != len(tags_batch[0]):
            raise Exception("Incorrect length of prediction for the input, "
                            "expected length: %i, got: %i" % (len(x_batch[0]),
 len(tags_batch[0])))
        predicted_tags = []
        ground_truth_tags = []
        for gt_tag_idx, pred_tag, token in zip(y_batch[0], tags_batch[0], toke
ns_batch[0]):
            if token != '<PAD>':
                ground_truth_tags.append(idx2tag[gt_tag_idx])
                predicted_tags.append(pred_tag)

        # We extend every prediction and ground truth sequence with 'O' tag
        # to indicate a possible end of entity.
        y_true.extend(ground_truth_tags + ['O'])
        y_pred.extend(predicted_tags + ['O'])

    results = precision_recall_f1(y_true, y_pred, print_results=True, short_re
port=short_report)
    return results
```

Run your experiment

Create *BiLSTMModel* model with the following parameters:

- *vocabulary_size* — number of tokens;
- *n_tags* — number of tags;
- *embedding_dim* — dimension of embeddings, recommended value: 200;
- *n_hidden_rnn* — size of hidden layers for RNN, recommended value: 200;
- *PAD_index* — an index of the padding token (`<PAD>`).

Set hyperparameters. You might want to start with the following recommended values:

- *batch_size*: 32;
- 4 epochs;
- starting value of *learning_rate*: 0.005
- *learning_rate_decay*: a square root of 2;
- *dropout_keep_probability*: try several values: 0.1, 0.5, 0.9.

However, feel free to conduct more experiments to tune hyperparameters and earn extra points for the assignment.

In [ ]:

In [66]:
```
tf.reset_default_graph()

model = BiLSTMModel(vocabulary_size=len(idx2token),
                    n_tags=len(idx2tag),
                    embedding_dim=200,
                    n_hidden_rnn=200,
                    PAD_index=token2idx['<PAD>'])

batch_size = 32
n_epochs = 4
learning_rate = 0.005
learning_rate_decay = np.sqrt(2.)
dropout_keep_probability = 0.5
```

If you got an error *"Tensor conversion requested dtype float64 for Tensor with dtype float32"* in this point, check if there are variables without dtype initialised. Set the value of dtype equals to *tf.float32* for such variables.

Finally, we are ready to run the training!

In [67]:
```
sess = tf.Session()
sess.run(tf.global_variables_initializer())

print('Start training... \n')
for epoch in range(n_epochs):
    # For each epoch evaluate the model on train and validation data
    print('-' * 20 + ' Epoch {} '.format(epoch+1) + 'of {} '.format(n_epochs)
+ '-' * 20)
    print('Train data evaluation:')
    eval_conll(model, sess, train_tokens, train_tags, short_report=True)
    print('Validation data evaluation:')
    eval_conll(model, sess, validation_tokens, validation_tags, short_report=T
rue)

    # Train the model
    for x_batch, y_batch, lengths in batches_generator(batch_size, train_token
s, train_tags):
        model.train_on_batch(sess, x_batch, y_batch, lengths, learning_rate, d
ropout_keep_probability)

    # Decaying the learning rate
    learning_rate = learning_rate / learning_rate_decay

print('...training finished.')
```

```
Start training...

-------------------- Epoch 1 of 4 --------------------
Train data evaluation:
processed 105778 tokens with 4489 phrases; found: 78274 phrases; correct: 19
3.

precision:  0.25%; recall:  4.30%; F1:  0.47

Validation data evaluation:
processed 12836 tokens with 537 phrases; found: 9492 phrases; correct: 26.

precision:  0.27%; recall:  4.84%; F1:  0.52

-------------------- Epoch 2 of 4 --------------------
Train data evaluation:
processed 105778 tokens with 4489 phrases; found: 2182 phrases; correct: 488.

precision:  22.36%; recall:  10.87%; F1:  14.63

Validation data evaluation:
processed 12836 tokens with 537 phrases; found: 193 phrases; correct: 32.

precision:  16.58%; recall:  5.96%; F1:  8.77

-------------------- Epoch 3 of 4 --------------------
Train data evaluation:
processed 105778 tokens with 4489 phrases; found: 4452 phrases; correct: 163
3.

precision:  36.68%; recall:  36.38%; F1:  36.53

Validation data evaluation:
processed 12836 tokens with 537 phrases; found: 302 phrases; correct: 120.

precision:  39.74%; recall:  22.35%; F1:  28.61

-------------------- Epoch 4 of 4 --------------------
Train data evaluation:
processed 105778 tokens with 4489 phrases; found: 4505 phrases; correct: 277
9.

precision:  61.69%; recall:  61.91%; F1:  61.80

Validation data evaluation:
processed 12836 tokens with 537 phrases; found: 364 phrases; correct: 162.

precision:  44.51%; recall:  30.17%; F1:  35.96

...training finished.
```

Now let us see full quality reports for the final model on train, validation, and test sets. To give you a hint whether you have implemented everything correctly, you might expect F-score about 40% on the validation set.

The output of the cell below (as well as the output of all the other cells) should be present in the notebook for peer2peer review!

In [69]:
```
print('-' * 20 + ' Train set quality: ' + '-' * 20)
train_results = eval_conll(model, sess, train_tokens, train_tags, short_report
=False)

print('-' * 20 + ' Validation set quality: ' + '-' * 20)
validation_results = eval_conll(model, sess, validation_tokens, validation_tag
s, short_report=False)

print('-' * 20 + ' Test set quality: ' + '-' * 20)
test_results = eval_conll(model, sess, test_tokens, test_tags, short_report=Fa
lse)
```

```
-------------------- Train set quality: --------------------
processed 105778 tokens with 4489 phrases; found: 4813 phrases; correct: 349
5.

precision:  72.62%; recall:  77.86%; F1:  75.15

             company: precision:   75.20%; recall:   89.11%; F1:   81.57; pre
dicted:   762

            facility: precision:   62.93%; recall:   82.17%; F1:   71.27; pre
dicted:   410

             geo-loc: precision:   87.02%; recall:   91.57%; F1:   89.24; pre
dicted:  1048

               movie: precision:    0.00%; recall:    0.00%; F1:    0.00; pre
dicted:     0

         musicartist: precision:   43.05%; recall:   28.02%; F1:   33.94; pre
dicted:   151

               other: precision:   69.24%; recall:   81.77%; F1:   74.98; pre
dicted:   894

              person: precision:   80.65%; recall:   94.58%; F1:   87.06; pre
dicted:  1039

             product: precision:   43.24%; recall:   66.35%; F1:   52.36; pre
dicted:   488

          sportsteam: precision:   90.48%; recall:    8.76%; F1:   15.97; pre
dicted:    21

               tvshow: precision:    0.00%; recall:    0.00%; F1:    0.00; pre
dicted:     0

-------------------- Validation set quality: --------------------
processed 12836 tokens with 537 phrases; found: 466 phrases; correct: 189.

precision:  40.56%; recall:  35.20%; F1:  37.69

             company: precision:   55.32%; recall:   50.00%; F1:   52.53; pre
dicted:    94

            facility: precision:   35.14%; recall:   38.24%; F1:   36.62; pre
dicted:    37

             geo-loc: precision:   66.67%; recall:   49.56%; F1:   56.85; pre
dicted:    84

               movie: precision:    0.00%; recall:    0.00%; F1:    0.00; pre
dicted:     0

         musicartist: precision:   16.67%; recall:    7.14%; F1:   10.00; pre
dicted:    12

               other: precision:   28.71%; recall:   35.80%; F1:   31.87; pre
```

```
dicted:   101

              person: precision:   47.76%; recall:   28.57%; F1:   35.75; pre
dicted:    67

             product: precision:    7.04%; recall:   14.71%; F1:    9.52; pre
dicted:    71

          sportsteam: precision:    0.00%; recall:    0.00%; F1:    0.00; pre
dicted:     0

               tvshow: precision:    0.00%; recall:    0.00%; F1:    0.00; pre
dicted:     0

-------------------- Test set quality: --------------------
processed 13258 tokens with 604 phrases; found: 565 phrases; correct: 227.

precision:  40.18%; recall:  37.58%; F1:  38.84

             company: precision:   57.14%; recall:   42.86%; F1:   48.98; pre
dicted:    63

            facility: precision:   34.62%; recall:   38.30%; F1:   36.36; pre
dicted:    52

             geo-loc: precision:   74.77%; recall:   50.30%; F1:   60.14; pre
dicted:   111

               movie: precision:    0.00%; recall:    0.00%; F1:    0.00; pre
dicted:     0

         musicartist: precision:   14.29%; recall:    3.70%; F1:    5.88; pre
dicted:     7

               other: precision:   24.29%; recall:   33.01%; F1:   27.98; pre
dicted:   140

              person: precision:   39.39%; recall:   50.00%; F1:   44.07; pre
dicted:   132

             product: precision:    3.39%; recall:    7.14%; F1:    4.60; pre
dicted:    59

          sportsteam: precision:  100.00%; recall:    3.23%; F1:    6.25; pre
dicted:     1

               tvshow: precision:    0.00%; recall:    0.00%; F1:    0.00; pre
dicted:     0
```

Conclusions

Could we say that our model is state of the art and the results are acceptable for the task? Definately, we can say so. Nowadays, Bi-LSTM is one of the state of the art approaches for solving NER problem and it outperforms other classical methods. Despite the fact that we used small training corpora (in comparison with usual sizes of corpora in Deep Learning), our results are quite good. In addition, in this task there are many possible named entities and for some of them we have only several dozens of trainig examples, which is definately small. However, the implemented model outperforms classical CRFs for this task. Even better results could be obtained by some combinations of several types of methods, e.g. see this (https://arxiv.org/abs/1603.01354) paper if you are interested.