Advanced Methods in NLP | Lecturer: Jonathan Berant

Home Assignment 5

Due Date: *June 30, 2019*

In this home assignment we will implement models for NER tagging, get familiar with TensorFlow and learn how to use TensorBoard to analyze our models training process and performance. To evaluate the quality of our models, we will look at precision, recall and the F_1 measure, as well as confusion matrices. In case you are not familiar with these terms, please see the references.[1] This assignment was adapted from the Stanford CS224n course.

Download the data and supporting code from the following link: https://drive.google.com/file/d/1n9ocMc5ICh4PIwLRgRZ1FmHFBgf2nb5G/view?usp=sharing. Notice that you will only need to edit the files `window.py`, `rnn.py` and `rnn_cell.py`. For your convenience, coding sections are marked with ♣ and additional deliverables (to the coding and written solutions) are marked with ■.

Your code should run properly on Python **2.7** on Linux. All models implemented in this assignment can be trained on a CPU in less than a few hours (therefore, you will not need a GPU to complete it).

Submit your solution through Moodle, create a zip file named `<id1>_<id2>_<id3>.zip` (where id1 refers to the ID of the first student). The submission zip should include: (1) the code necessary for running the tests provided out-of-the-box, (2) your written solution, (3) additional deliverables (3 prediction files and 3 plots, as will be specified). Only one student needs to submit.

Your code will be tested on the School of Computer Science operating system, installed on `nova` and other similar machines. Please make sure your code runs there. If your code does not run on `nova`, you will not get points.

1 A window into NER

The first model we are going to implement is a simple baseline that predicts a label for each token separately using features from a window around it.

Figure 1 shows an example of an input sequence and the first window from this sequence. Let $\mathbf{x} \stackrel{\text{def}}{=} \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(T)}$ be an input sequence of length T and $\mathbf{y} \stackrel{\text{def}}{=} \mathbf{y}^{(1)}, \mathbf{y}^{(2)}, ..., \mathbf{y}^{(T)}$ be an output sequence, also of length T. Here, each element $\mathbf{x}^{(t)}$ and $\mathbf{y}^{(t)}$ are one-hot vectors representing the word at the t-th index of the sentence. In a window based classifier, every input sequence is split into T new data points, each representing a window and its label. A new input is constructed from a window around $\mathbf{x}^{(t)}$ by concatenating w tokens to the

[1] https://en.wikipedia.org/wiki/Precision_and_recall,
https://en.wikipedia.org/wiki/Confusion_matrix

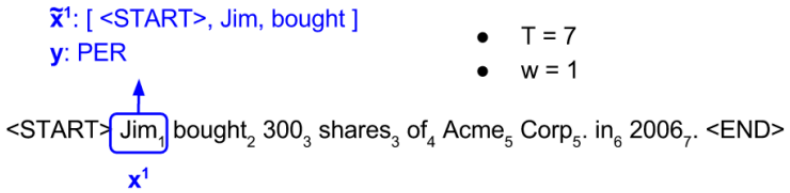

Figure 1: A sample input sequence for NER tagging

left and right of $\mathbf{x}^{(t)}$: $\tilde{\mathbf{x}}^{(t)} \stackrel{\text{def}}{=} [\mathbf{x}^{(t-w)}, ..., \mathbf{x}^{(t)}, ..., \mathbf{x}^{(t+w)}]$; we continue to use $\mathbf{y}^{(t)}$ as its label. For windows centered around tokens at the very beginning of a sentence, we add special start tokens (`<START>`) to the beginning of the window and for windows centered around tokens at the very end of a sentence, we add special end tokens (`<END>`) to the end of the window. For example, consider constructing a window around Jim in the sentence above. If window size were 1, we would add a single start token to the window (resulting in a window of [`<START>, Jim, bought`]). If window size were 2, we would add two start tokens to the window (resulting in a window of [`<START>, <START>, Jim, bought, 300`]). With these, each input and output is of a uniform length (w and 1 respectively) and we can use a simple feedforward neural net to predict $\mathbf{y}^{(t)}$ from $\tilde{\mathbf{x}}^{(t)}$:

As a simple but effective model to predict labels from each window, we will use a single hidden layer with a ReLU activation, combined with a softmax output layer and the cross-entropy loss:

$$\begin{aligned}
\mathbf{e}^{(t)} &= [\mathbf{x}^{(t-w)}E, ..., \mathbf{x}^{(t)}E, ..., \mathbf{x}^{(t+w)}E] \\
\mathbf{h}^{(t)} &= \text{ReLU}(\mathbf{e}^{(t)}W + \mathbf{b}_1) \\
\hat{\mathbf{y}}^{(t)} &= \text{softmax}(\mathbf{h}^{(t)}U + \mathbf{b}_2) \\
J &= \text{CE}(\mathbf{y}^{(t)}, \hat{\mathbf{y}}^{(t)}) \\
\text{CE}(\mathbf{y}^{(t)}, \hat{\mathbf{y}}^{(t)}) &= -\sum_i y_i^{(t)} \log \hat{y}_i^{(t)}
\end{aligned}$$

where $E \in \mathbb{R}^{V \times D}$ are word embeddings, $\mathbf{h}^{(t)}$ is dimension H and $\hat{\mathbf{y}}^{(t)}$ is of dimension C, where V is the size of the vocabulary, D is the size of the word embedding, H is the size of the hidden layer and C are the number of classes being predicted.

(a) i Provide 2 examples of sentences containing a named entity with an ambiguous type (e.g. the entity could either be a person or an organization, or it could either be an organization or not an entity).

ii Why might it be important to use features apart from the word itself to predict named entity labels?

iii Describe at least two features (apart from the word) that would help in predicting whether a word is part of a named entity or not.

(b) i What are the dimensions of $\mathbf{e}^{(t)}$, W and U if we use a window of size w?

ii What is the computational complexity of predicting labels for a sentence of length T?

(c) Implement a window-based classifier model in `window.py` using this approach. To do so, you will have to:

i ♣ Transform a batch of input sequences into a batch of windowed input-output pairs in the make windowed data function. You can test your implementation by running `python window.py test1`.

ii ♣ Implement the feed-forward model described above by appropriately completing functions and the variable `n_window_features` in the `WindowModel` class. You can test your implementation by running `python window.py test2`.

iii Train your model using the command python `window.py` train. The code should take only about 2-3 minutes to run and you should get a development score of at least 81% F_1.

The model and its output will be saved to `results/window/<timestamp>/`, where `<timestamp>` is the date and time at which the program was run. The file `results.txt` contains formatted output of the models predictions on the development set, and the file log contains the printed output, i.e. confusion matrices and F_1 scores computed during the training. Finally, you can interact with your model using:
`python window.py shell -m results/window/<timestamp>/`

■ Deliverable: the `window_predictions.conll` file from the appropriate results folder.

(d) Analyze the predictions of your model using the files generated above.

i Report your best development entity-level F_1 score and the corresponding token-level confusion matrix. Briefly describe what the confusion matrix tells you about the errors your model is making.

ii Describe at least 2 modeling limitations of the window-based model and support these conclusions using examples from your models output (i.e. identify errors that your model made due to its limitations). You can also support your conclusions using predictions made by your model on examples manually entered through the shell.

2 RNNs for NER

We will now tackle the task of NER by using a recurrent neural network (RNN). Recall that each RNN cell combines the previous hidden state with the current input using a sigmoid. We then use the hidden state to predict the output at each timestep:

$$\begin{aligned}
\mathbf{e}^{(t)} &= \mathbf{x}^{(t)} E \\
\mathbf{h}^{(t)} &= \sigma(\mathbf{h}^{(t-1)} W_h + \mathbf{e}^{(t)} W_e + \mathbf{b}_1) \\
\hat{\mathbf{y}}^{(t)} &= \text{softmax}(\mathbf{h}^{(t)} U + \mathbf{b}_2)
\end{aligned}$$

where $E \in \mathbb{R}^{V \times D}$ are word embeddings, $W_h \in \mathbb{R}^{H \times H}$, $W_e \in \mathbb{R}^{D \times H}$ and $\mathbf{b}_1 \in \mathbb{R}^H$ are parameters for the RNN cell, and $U \in \mathbb{R}^{H \times C}$ and $\mathbf{b}_2 \in \mathbb{R}^C$ are parameters for the softmax. As before, V is the size of the vocabulary, D is the size of the word embedding, H is the

size of the hidden layer and C are the number of classes being predicted (here 5).

In order to train the model, we use a cross-entropy loss for the every predicted token:

$$J = \sum_{t=1}^{T} \mathrm{CE}(\mathbf{y}^{(t)}, \hat{\mathbf{y}}^{(t)})$$
$$\mathrm{CE}(\mathbf{y}^{(t)}, \hat{\mathbf{y}}^{(t)}) = -\sum_{i} y_i^{(t)} \log \hat{y}_i^{(t)}$$

(a) i How many more parameters does the RNN model in comparison to the window-based model?

ii What is the computational complexity of predicting labels for a sentence of length T (for the RNN model)?

(b) Recall that the actual score we want to optimize is entity-level F_1.

i Name at least one scenario in which decreasing the cross-entropy cost would lead to an decrease in entity-level F_1 scores.

ii Why it is difficult to directly optimize for F_1?

(c) ♣ Implement an RNN cell using the equations described above in the rnn cell function of `rnn_cell.py`. You can test your implementation by running `python rnn_cell.py test`. Hint: We recommend you initialize the bias terms in the RNN/GRU cell, with 0s. You can do this by using `tf.constant_initializer`.

(d) Implementing an RNN requires us to unroll the computation over the whole sentence. Unfortunately, each sentence can be of arbitrary length and this would cause the RNN to be unrolled a different number of times for different sentences, making it impossible to batch process the data.
The most common way to address this problem is pad our input with zeros. Suppose the largest sentence in our input is M tokens long, then, for an input of length T we will need to:

- Add 0-vectors to $\mathbf{x}$ and $\mathbf{y}$ to make them M tokens long. These 0-vectors are still one-hot vectors, representing a new `NULL` token.
- Create a masking vector, $(m^{(t)})_{t=1}^{M}$ which is 1 for all $t \leq T$ and 0 for all $t > T$. This masking vector will allow us to ignore the predictions that the network makes on the padded input.
- Of course, by extending the input and output by $M - T$ tokens, we might change our loss and hence gradient updates. In order to tackle this problem, we modify our loss using the masking vector:

$$J = \sum_{t=1}^{M} m^{(t)} \mathrm{CE}(\mathbf{y}^{(t)}, \hat{\mathbf{y}}^{(t)})$$

i How would the loss and gradient updates change if we did not use masking? How does masking solve this problem?

ii ♣ Implement pad sequences in your code. You can test your implementation by running python `rnn.py test1`.

(e) ♣ Implement the rest of the RNN model assuming only fixed length input by appropriately completing functions in the `RNNModel` class. This will involve:

- Implementing the `add_placeholders`, `create_feed_dict`, `add_embedding`, `add_training_op` functions.
- Implementing the `add_prediction_op_rnn` operation that unrolls the RNN loop `self.max_length` times. Remember to reuse variables in your variable scope from the 2nd timestep onwards to share the RNN cell weights W_h and W_e across timesteps.
- Implementing `add_loss_op` to handle the mask vector returned in the previous part.

You can test your implementation by running `python rnn.py test2`.

(f) Train your model using the command `python rnn.py train`. Training should take about 1-2 hours on your CPU. You should get a development F_1 score of at least 85%.

The model and its output will be saved to `results/rnn/<timestamp>/`, where `<timestamp>` is the date and time at which the program was run. The file `results.txt` contains formatted output of the model's predictions on the development set, and the file log contains the printed output, i.e. confusion matrices and F_1 scores computed during the training. Finally, you can interact with your model using:
`python rnn.py shell -m results/rnn/<timestamp>/`

■ Deliverable: the `rnn_predictions.conll` file from the appropriate results folder.

(g) i Describe at least 2 modeling limitations of this RNN model and support these conclusions using examples from your model's output.

ii For each limitation, suggest some way you could extend the model to overcome the limitation.

3 GRUs and TensorBoard

In this part, we will change our model to use TensorFlow's implementation of a dynamic RNN with GRU cell, and analyze the training process with TensorBoard. All coding parts should be implemented in the same file as in question 2, e.g. `rnn.py`.

(a) ♣ Implement the `add_prediction_op_gru` operation, that apply a dynamic RNN model with GRU cell on the sequence input provided, followed by a single layer feed-forward neural network (FFNN). Here, we shall use TensorFlow API functions, rather

than implementing the ops by ourselves. The prediction op is described by the following equations:

$$\begin{aligned} \mathbf{e}^{(t)} &= \mathbf{x}^{(t)}E \\ \mathbf{o}^{(1)}, ..., \mathbf{o}^{(T)} &= \text{RNN}(\mathbf{e}^{(1)}, ..., \mathbf{e}^{(T)}) \\ \hat{\mathbf{y}}^{(t)} &= \text{softmax}(\text{FFNN}(\mathbf{o}^{(t)})) \end{aligned}$$

where $\mathbf{o}^{(t)}$ is the RNN output for timestep t, and all other notations are the same as in question 2. Note that the softmax op is applied as part of `add_loss_op`, as in question 2, `add_prediction_op_gru` should return the logits that the FFNN outputs.

(b) ♣ Implement the `add_summary_op` operation to summarize: (1) the average loss, (2) the FFNN output logits (as a histogram), (3) the average entropy of the predictions. To calculate the entropy, use the TensorFlow function `tf.log` that implements the natural logarithm, to have:

$$\text{Ent}(\hat{\mathbf{y}}^{(t)}) = -\sum_i \hat{y}_i^{(t)} \ln \hat{y}_i^{(t)}$$

Use the function `tf.clip_by_value` to clip the prediction values before applying the logarithm function. This will help avoiding invalid input values, that could be resulted from numerical rounding.

You can test your implementation by running `python rnn.py test2 -c gru`.

(c) Train the model with the 'summarize' flag activated, by running `python rnn.py train -c gru -s`. Training should take about 3-4 hours, and you should get a development F_1 score of at least 85%.

The model and its output will be saved to `results/gru/<timestamp>/`, where `<timestamp>` is the date and time at which the program was run. To view the summarized metrics, run `tensorboard --logdir=<path_to_model_dir>` and in your browser open the url `http://localhost:6006`.[2]

i What is the maximum entropy value possible, for a single timestep prediction?

ii Analyze the 3 graphs obtained from the training. What does the entropy tells us about the model predictions? How is this reflected by the logits histogram?

■ Deliverable: the `gru_predictions.conll` file from the appropriate results folder.
■ Deliverable: 3 plots from TensorBoard (simply by taking a screen capture of the graph) of the average loss, logits histogram, and average entropy. For the scalar summaries, make sure to (1) set "smoothing" to zero on the left menu, (2) check out the "Ignore outliers in chart scaling" option on the left menu, and (3) enlarge the plot by clicking the small button below it.

[2]This is TensorBoard's default port, it can be configured from the command line

(d) Examine your model's predictions while considering the prediction values. This can be done by activating the 'verbose' flag, either for evaluation on the development set or for interaction though the shell:

```
python rnn.py evaluate -s -c gru -m results/gru/<timestamp>/
python rnn.py shell -s -c gru -m results/gru/<timestamp>/
```

What labels does the model predict well? Where does it struggle? Is it reflected by the prediction values? Support your claims using examples from the model's output, or using predictions made by it on examples manually entered through the shell.