CS 224N: Assignment 3

RYAN MCMAHON FRIDAY 28TH APRIL, 2017

Contents

Problem 1: A Window Into NER (30 pts)

See "~/03-HW3/assignment3.pdf" for the full introduction to the question.

... 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 $\boldsymbol{y}^{(t)}$ from $\tilde{\boldsymbol{x}}^{(t)}$: 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:

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

where $L \in \mathbb{R}^{V \times D}$ are word embeddings, $\boldsymbol{h}^{(t)}$ is dimension H and $\hat{\boldsymbol{y}}^{(t)}$ 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 is the number of classes being predicted (here 5).

1.1 (a) Conceptual (5 pts)

1.1.1 i) (2 pts)

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).

Answer:

1. What have you heard about Louis Vuitton?
2. We had dinner at that new restaurant, Frank's, last night.

1.1.2 ii) (1 pt)

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

Answer:
The word feature matrix is an extremely sparse representation format, wherein it is going to be difficult to recognize entities that don't appear very often.

1.1.3 iii) (2 pts)

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.

Answer:
The most obvious additional predictor would be capitalization. Another would be part-of-speech tags (e.g., if the previous word, $w^{(t-1)}$, is a determiner, the current word is more likely to be an entity).

1.2 (b) Network Components (5 pts)

1.2.1 i) (2 pts)

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

Answer:

1. $\boldsymbol{e}^{(t)}$ is going to be a row vector of length $(2w + 1) \times D$
2. W is going to be a matrix of dimensionality $|\boldsymbol{e}^{(t)}| \times H$
3. U is going to be a matrix of dimensionality $H \times C$

1.2.2 ii) (3 pts)

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

Answer:
Since we know the dimensionality of the network's elements (see above), and we know that complexity scales linearly with the operations on those elements, we can generalize from the dimensionality to complexity. Then, for a sentence of length T, the complexity of this model is $O\left((|\boldsymbol{e}^{(t)}| \times H \times T) + (H \times C)\right)$, where $|\boldsymbol{e}^{(t)}| = (2w + 1) \times D$.

1.3 (c) Implementing a Window Based Model (15 pts)

Implement a window-based classifier model in `q1_window.py` *using this approach. To do so, you will have to:*

1.3.1 i) (5 pts)

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 q1_window.py test1`.

Answer:

See code: `~/code/q1_window.py`.

1.3.2 ii) (8 pts)

Implement the feed-forward model described above by appropriately completing functions in the `WindowModel` *class. You can test your implementation by running* `python q1_window.py test2`.

Answer:

See code: `~/code/q1_window.py`.

1.3.3 iii) (2 pts)

Train your model using the command `python q1_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.

Answer:

See code, `~/code/q1_window.py`, and log file, `~/code/results/window/20170419_165429/log`. Loading the data, compiling the model, and fitting (for 15 epochs instead of the default 10) took ~44 seconds. The best entity level F_1 score on the development set was 87%.

1.4 (d) Analysis (5 pts)

1.4.1 i) (1 pt)

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.*

Answer:

The best entity level F_1 score on the development set was 87% (see results for epoch 12 in the log file). The token-level confusion matrix can be seen below in Table 1.1.

Table 1.1: Development Set Confusion Matrix

	Predicted Label				
True Label	**PER**	**ORG**	**LOC**	**MISC**	**O**
PER	2,973	41	57	12	66
ORG	106	1,727	88	57	114
LOC	36	72	1,931	18	37
MISC	33	66	34	1,026	109
O	34	39	24	29	42,633

We can see, from Table 1.1, that ORG and MISC are the most difficult tags for the model to predict. The model tends to mis-classify tokens of either tag as not being a named entity (i.e., the O tag) far more frequently than for tokens labeled as a person or location. Additionally, we can see that the model mis-classifies organizations as people more frequently than the reverse.

1.4.2 ii) (4 pts)

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.

Answer:

The first, and most obvious, limitation of the window-based model is that the contextual semantic vectors associated with a word are defined entirely by the window size. Thus, attributes of a word (e.g., a clarifying clause) that are seen more than a few hops away are omitted from the model. An example error that may be attributed to this is the misclassification of the *Titanic* as an organization:

> "A (O/O) 20-ton (O/O) piece (O/O) of (O/O) the (O/O) Titanic (MISC/ORG) 's (O/O) steel (O/O) hull (O/O) . . . ".

The model producing this prediction had a window size of two. So when classifying 'Titanic', the context was "of the Titanic 's steel". Given this window, it's relatively easy to see how such an error could be made.

A second limitation is the omission of surrounding tags as features. This is especially problematic for identifying geopolitical entities. The following prediction serves a useful example:

> "The (O/O) Federal (LOC/ORG) Republic (LOC/LOC) of (O/ORG) Yugoslavia (LOC/LOC) is (O/O) the (O/O) only (O/O) country (O/O) . . . ".

Predictions for 'Federal' and 'of' should both benefit from knowing the tags associated with nearby words.

Problem 2: Recurrent Neural Nets for NER (40 pts)

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

$$\begin{aligned}
\boldsymbol{e}^{(t)} &= \boldsymbol{x}^{(t)} L \\
\boldsymbol{h}^{(t)} &= \sigma\left(\boldsymbol{h}^{(t-1)} W_h + \boldsymbol{e}^{(t)} W_x + \boldsymbol{b}_1\right) \\
\hat{\boldsymbol{y}}^{(t)} &= \text{softmax}\left(\boldsymbol{h}^{(t)} U + \boldsymbol{b}_2\right) \ ,
\end{aligned}$$

where $L \in \mathbb{R}^{V \times D}$ are word embeddings, $W_h \in \mathbb{R}^{H \times H}$, $W_x \in \mathbb{R}^{D \times H}$, and $\boldsymbol{b}_1 \in \mathbb{R}^H$ are the parameters for the RNN cell, and $U \in \mathbb{R}^{H \times C}$ and $\boldsymbol{b}_2 \in \mathbb{R}^C$ are the 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 is the number of classes being predicted (e.g., 5 here).

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

$$\begin{aligned}
J &= \sum_{t=1}^{T} \text{CE}\left(\boldsymbol{y}^{(t)} \ , \hat{\boldsymbol{y}}^{(t)}\right) \\
\text{CE}\left(\boldsymbol{y}^{(t)} \ , \hat{\boldsymbol{y}}^{(t)}\right) &= -\sum_i y_i^{(t)} \log\left(\hat{y}_i^{(t)}\right) \ .
\end{aligned}$$

2.1 (a) Size and Complexity (4 pts)

2.1.1 i) (1 pt)

How many more parameters does the RNN model in comparison to the window-based model?

Answer:
The RNN model has $(H \times H) - (2w \times D \times H)$ more parameters than a window-based model with a window size of w. This is because we have the new $W_h \in \mathbb{R}^{H \times H}$ weight matrix, but are using a smaller $W_x \in \mathbb{R}^{D \times H}$ matrix instead of the $W \in \mathbb{R}^{(2w+1) \times DH}$ weight matrix from the window model. So, depending on the dimensionality of the embeddings and hidden layer, there may not be much of a difference between the two wrt the number of parameters. For example, if $D = 50$, $H = 200$, and $w = 2$ they have the same number of parameters.

2.1.2 ii) (3 pts)

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

Answer:
Working forward in the network (for one time-step), we first calculate $\boldsymbol{e}^{(t)}$ in $O(D)$ time. Next comes $\boldsymbol{h}^{(t)}$, which takes $O\left((H \times H) + DH + H\right)$ operations. Finally, we have to make the predictions, $\hat{\boldsymbol{y}}^{(t)}$, and that takes $O(HC + C)$ operations.

So, for one time-period, we can factor the first and second steps to $O\left((H + D)(H + 1)\right)$ and the prediction step to $O\left(C(H + 1)\right)$. To get the complexity for the entire sentence of length T, we combine those two pieces and multiply by T: $O\left(((H + D)(H + 1) + C(H + 1)) \times T\right)$.

2.2 (b) Optimizing the Loss Function (2 pts)

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

2.2.1 i) (1 pt)

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

Answer:
This happens pretty much whenever the model moves from predicting an $n \geqslant 2$ token length entity as being all 'O's to predicting some $c < n$ component tokens correct. Token level accuracy will increase, thus lowering the cross-entropy cost. However, at the entity level, the model will now have a lower precision (i.e., TP / (TP + FP)) and recall will remain the same (i.e., TP / (TP + FN)).

2.2.2 ii) (1 pt)

Why is it difficult to directly optimize for F_1?

Answer:
One problem with optimizing for F_1 is that it is not convex (Ye et al. 2012). Additionally, is "non-decomposable" (Kar, Narasimhan and Jain 2014): meaning that the metric can't be broken down by individual data points.

2.3 (c) RNN Cell (5 pts)

Implement an RNN cell using the equations described above in the `rnn_cell` *function of* `q2_rnn_cell.py`. *You can text your implementation by running* `python q2_rnn_cell.py test`.

Answer:

See code, `~/code/q2_rnn_cell.py`

2.4 (d) Padding Sentences (8 pts)

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 to 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:*

1. *Add "0-vectors" to $\boldsymbol{x}$ and $\boldsymbol{y}$ to make them M tokens long. These "0-vectors" are still one-hot vectors, representing a new* `NULL` *token.*
2. *Create a* masking vector, $\left(m^{(t)}\right)_{t=1}^{M}$ *which is 1 for all $t \leqslant T$ and 0 for all $t > T$. This masking vector will allow us to ignore the predictions that the network makes on the padded input.*
3. *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:*

   $$J = \sum_{t=1}^{M} m^{(t)} \text{CE}\left(y^{(t)}, \hat{y}^{(t)}\right)$$

2.4.1 i) (3 pts)

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

Answer:

The loss would be impacted based on our predictions of the pads (i.e., predicting 0s). This then would change the gradient as errors are propagated back and impact the learning of the weights. With the masking vector in place, the loss associated with the pads is always 0, so they don't impact learning.

2.4.2 ii) (5 pts)

Implement `pad_sequences` *in your code. You can test your implementation by running* `python q2_rnn.py test1`.

Answer:

See code, `~/code/q2_rnn.py`. I did essentially the same thing earlier when making the windows in the `make_windowed_data` function of `~/code/q1_window.py`.

2.5 (e) Implementing an RNN in TensorFlow (12 pts)

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

1. *Implementing the* `add_placeholders`, `add_embedding`, `add_training_op` *functions.*
2. *Implementing the* `add_prediction_op` *operation that unrolls the RNN loop* `self.max_length` *times. Remember to* reuse *variables in your variable scope from the* 2^{nd} *timestep onwards to share the RNN cell weights* W_x *and* W_h *across timesteps.*
3. *Implementing the* `add_loss_op` *to handle the mask vector returned in the previous part*

You can test your implementations by running `python q2_rnn.py test2`.

Answer:

See code, `~/code/q2_rnn.py`.