CS 224n: Assignment #3

**Due date: 2/25 11:59 PM PST** (You are allowed to use 4 late days maximum for this assignment)*Because this assignment was released late, we are providing a free extra late day for the rest of the course that can be used for either this assignment or for assignment 4 / the final project. However, note that assignment 4 or the final project will still be due on 3/17 and will allow a maximum of 3 late days (i.e. the latest they can be submitted is 3/20) so that we can complete grading on time.* These questions require thought, but do not require long answers. Please be as concise as possible. **This assignment can be completed in groups of up to 3 people.** We encourage students to discuss in groups for assignments. **However, each student group must finish the problem set and programming assignment individually** (i.e. by only the group members). We ask that you abide the university Honor Code and that of the Computer Science department, and make sure that all of your submitted work are done by yourself and your team members only. Please review any additional instructions posted on the assignment page at http://cs224n.stanford.edu/assignments.html. When you are ready to submit, please follow the instructions on the course website.

**Last updated: February 17, 2017, 4:40PM PST  
Note: In this assignment, the inputs to neural network layers will be row vectors because this  
is standard practice for TensorFlow (some built-in TensorFlow functions assume the inputs  
are row vectors). This means the weight matrix of a hidden layer will right-multiply instead  
of left-multiply its input (i.e., xW** + **b instead of Wx** + **b).**

**A primer on named entity recognition**

In this assignment, we will build several different models for named entity recognition (NER). NER is a subtask of information extraction that seeks to locate and classify named entities in text into pre-defined categories such as the names of persons, organizations, locations, expressions of times, quantities, monetary values, percentages, etc. In the assignment, for a given a word in a context, we want to predict whether it represents one of four categories:

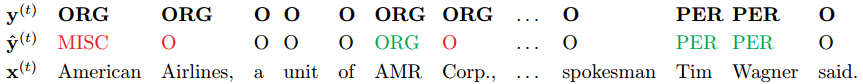
*•* **Person (PER)**: e.g. “Martha Stewart”, “Obama”, “Tim Wagner”, etc. Pronouns like “he” or “she” are *not* considered named entities

*•* **Organization (ORG)**: e.g. “American Airlines”, “Goldman Sachs”, “Department of Defense”.

*•* **Location (LOC)**: e.g. “Germany”, “Panama Strait”, “Brussels”, but not unnamed locations like “the bar” or “the farm”.

*•* **Miscellaneous (MISC)**: e.g. “Japanese”, “USD”, “1,000”, “Englishmen”.

We formulate this as a 5-class classification problem, using the four above classes and a null-class (O) for words that do not represent a named entity (most words fall into this category). For an entity that spans multiple words (“Department of Defense”), each word is separately tagged, and every contiguous sequence of non-null tags is considered to be an entity.  
Here is a sample sentence () with the named entities tagged above each token () as well as hypothetical predictions produced by a system ():



In the above example, the system mistakenly predicted “American” to be of the MISC class and ignores “Airlines” and “Corp.”. All together, it predicts 3 entities, “American”, “AMR” and “Tim Wagner”. To evaluate the quality of a NER system’s output, we look at precision, recall and the *F*1 measure.1 In particular, we will report precision, recall and *F*1 at both the token-level and the name-entity level. In the former case:

* Precision is calculated as the ratio of correct non-null labels predicted to the total number of non-null labels predicted (in the above example, it would be).
* Recall is calculated as the ratio of correct non-null labels predicted to the total number of *correct* non-null labels (in the above example, it would be).
* *F*1 is the harmonic mean of the two:. (in the above example, it would be).

For entity-level *F*1:

* Precision is the fraction of predicted entity name spans that line up exactly with spans in the gold standard evaluation data. In our example, “AMR” would be marked incorrectly because it does not cover the whole entity, i.e. “AMR Corp.”, as would “American”, and we would get a precision score of.
* Recall is similarly the number of names in the gold standard that appear at exactly the same location  
  in the predictions. Here, we would get a recall score of.
* Finally, the *F*1 score is still the harmonic mean of the two, and would be  in the example.

Our model also outputs a token-level *confusion matrix* 2. A confusion matrix is a specific table layout that allows visualization of the classification performance. Each column of the matrix represents the instances in a predicted class while each row represents the instances in an actual class. The name stems from the fact that it makes it easy to see if the system is confusing two classes (i.e. commonly mislabelling one as another).

1. **A window into NER (30 points)**

Let’s look at a simple baseline model that predicts a label for each token separately using features from a window around it.

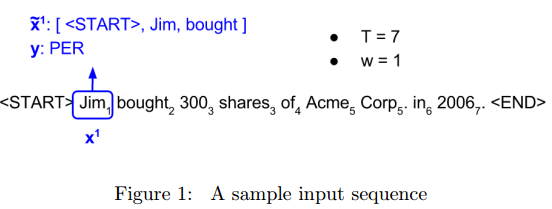
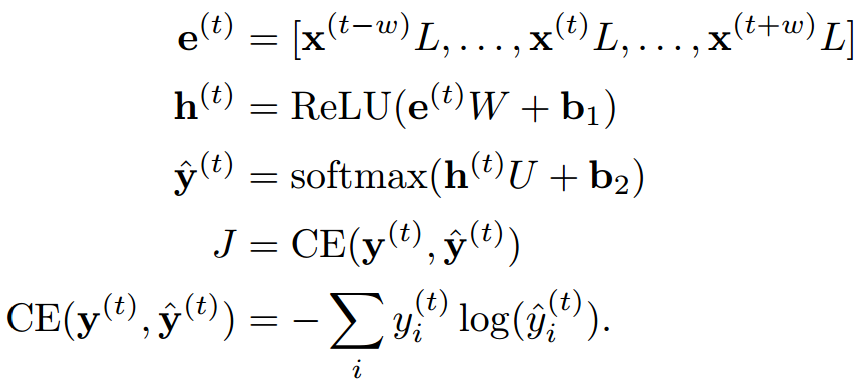
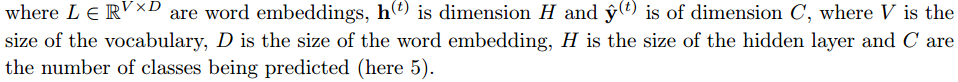


Figure 1 shows an example of an input sequence and the first window from this sequence. Let be an input sequence of length *T* and be an output sequence, also of length *T*. Here, each elementand are one-hot vectors representing the word at the t-th index of the sentence. In a window based classifier, 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 by concatenating *w* tokens to the left and right of ; we continue to use 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 from:

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:





1. (5 points) (written)

i. (2 points) 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: Jordan: (1) a player name (2) a sports shoe name.

ii. (1 point) Why might it be important to use features apart from the word itself to predict named entity labels?

命名实体通常有一些特殊的标志可以帮助预测。比如说，英文中命名实体通常首字母大写。

iii. (2 points) 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.

1. 单词首字母大写。②英文名之间有连字符· ③ 组织机构通常以Corp或Department开始或结束。
2. (5 points) (written)

i. (2 points) What are the dimensions of, *W* and *U* if we use a window of size *w*?

: 1\*(2w+1)D

W: (2w+1)D\*H

U: H\*C

ii. (3 points) What is the computational complexity of predicting labels for a sentence of length *T* ?

|  |  |
| --- | --- |
| 计算项 | 计算复杂度 |
|  | (2w+1)\*D |
|  | (2w+1)DH + H |
|  | H\*C + C |
| 单步总计 |  |

通常,所以单步总计。

长度为T的序列，总计算复杂度：

1. (15 points) (code) Implement a window-based classifier model in *q1\_window.py* using this approach. To do so, you will have to:

i. (5 points) 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**.

ii. (8 points) 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**.

iii. (2 points) 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.

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 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 q1\_window.py shell -m results/window/<timestamp>/**

**Deliverable:** After your model has trained, copy the window predictions.conll file from the appropriate results folder into the root folder of your code directory, so that it can be included in your submission.

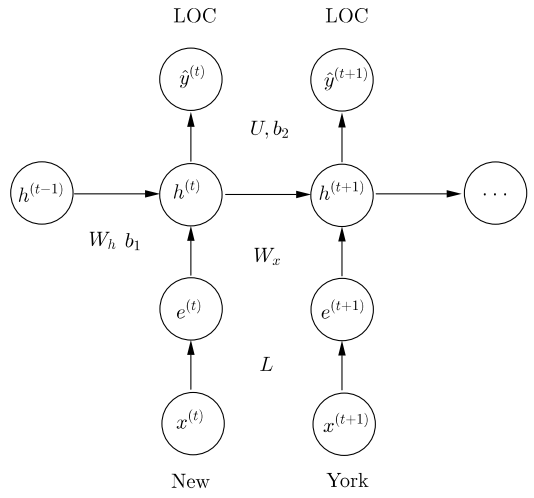
(d) (5 points) (written) Analyze the predictions of your model using the files generated above.

i. (1 point) 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.

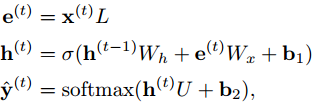
ii. (4 points) Describe at least 2 modeling limitations of the window-based model and support these conclusions using examples from your model’s output (i.e. identify errors that your model made due to its limitations). You can also support your conclusions using predictions made by your model on examples manually entered through the shell.

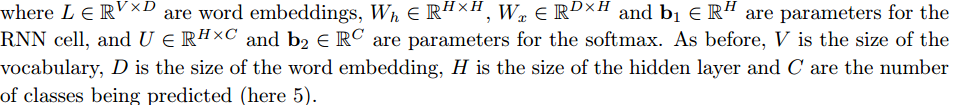
1. **Recurrent neural nets for NER (40 points)**

We will now tackle the task of NER by using a recurrent neural network (RNN).

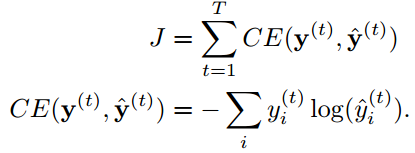


Recall that each RNN cell combines the hidden state vector with the input using a sigmoid. We then use the hidden state to predict the output at each timestep:





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



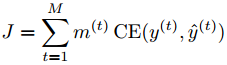
1. (4 points) (written)  
   i. (1 point) How many more parameters does the RNN model in comparison to the window-based model?

ii. (3 points) What is the computational complexity of predicting labels for a sentence of length *T* (for the RNN model)?

(b) (2 points) (written) Recall that the actual score we want to optimize is entity-level *F*1.  
i. (1 point) Name at least one scenario in which decreasing the cross-entropy cost would lead to  
an *decrease* in entity-level *F*1 scores.  
ii. (1 point) Why it is difficult to directly optimize for *F*1?

(c) (5 points) (code) Implement an RNN cell using the equations described above in the rnn cell  
function of q2 rnn cell.py. You can test your implementation by running python q2 rnn cell.py  
test.

(d) (8 points) (code/written) 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:  
1. Add 0-vectors to **x** and **y** to make them *M* tokens long.  
2. Create a *masking vector*, which is 1 for all *t ≤ 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:



1. (3 points) (written) How would the loss and gradient updates change if we did not use masking?  
   How does masking solve this problem?

ii. (5 points) (code) Implement pad sequences in your code. You can test your implementation  
by running python q2 rnn.py test1.

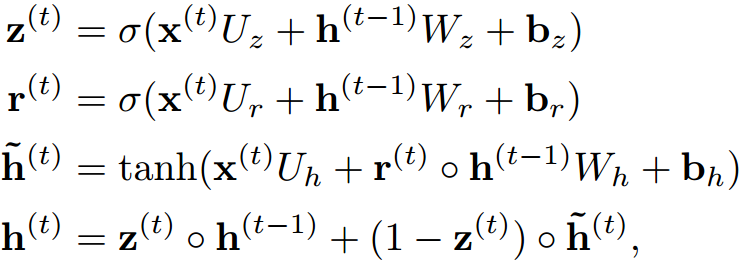
(e) (12 points) (code) Implement the rest of the RNN model assuming only fixed length input by  
appropriately completing functions in the RNNModel class. This will involve:  
1. Implementing the add placeholders, add embedding, add training op functions.  
2. Implementing the add prediction op operation that unrolls the RNN loop self.max length  
times. Remember to *reuse* variables in your variable scope from the 2nd timestep onwards to  
share the RNN cell weights *Wx* and *Wh* across timesteps.  
3. Implementing add loss op to handle the mask vector returned in the previous part.  
You can test your implementation by running python q2 rnn.py test2.

(f) (3 points) (code) Train your model using the command python q2 rnn.py train. Training  
should take about 2 hours on your CPU and 10{20 minutes if you use the GPUs provided by  
Microsoft Azure. 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 q2 rnn.py shell -m results/rnn/<timestamp>/  
**Deliverable:** After your model has trained, copy the rnn predictions.conll file from the  
appropriate results folder into your code directory so that it can be included in your submission.

(g) (6 points) (written)  
i. (3 points) Describe at least 2 modeling limitations of this RNN model and support these conclusions using examples from your model’s output.  
ii. (3 points) For each limitation, suggest some way you could extend the model to overcome the limitation.

1. **Grooving with GRUs (30 points)**

In class, we learned that a gated recurrent unit (GRU) is an improved RNN cell that greatly reduces  
the problem of vanishing gradients. Recall that a GRU is described by the following equations:

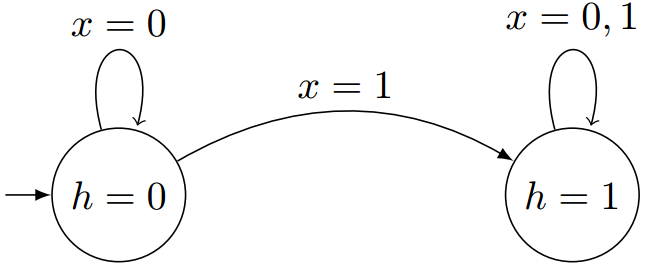


where **z**(*t*) is considered to be an *update gate* and **r**(*t*) is considered to be a *reset gate*.4  
Also, to keep the notation consistent with the GRU, for this problem, let the basic RNN cell be described  
by the equations:



To gain some intuition, let’s explore the behavior of the basic RNN cell and the GRU on some generated  
1-D sequences.

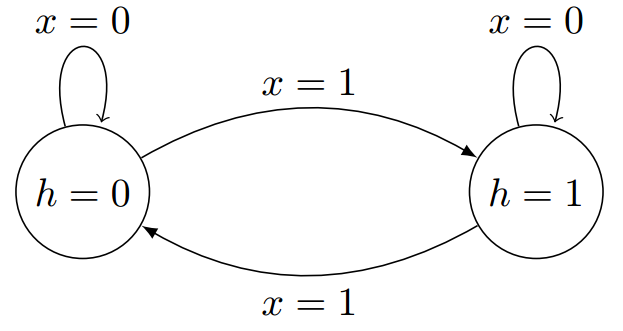
(a) (4 points) (written) **Modeling latching behavior.** Let’s say we are given input sequences starting  
with a 1 or 0, followed by *n* 0s, e.g. 0, 1, 00, 10, 000, 100, etc. We would like our state *h* to continue  
to remember what the first character was, irrespective of how many 0s follow. This scenario can  
also be described as wanting the neural network to learn the following simple automaton:



In other words, when the network sees a 1, it should change its state to also be a 1 and stay there.  
In the following questions, assume that the state is initialized at 0 (i.e. *h*(0) = 0), and that all  
the parameters are scalars. Further, assume that all sigmoid activations and tanh activations are  
replaced by the indicator function:  
**

i. (1 point) Identify values of *wh*, *uh* and *bh* for an RNN cell that would allow it to replicate the behavior described by the automaton above.  
ii. (3 points) Let *wr* = *ur* = *br* = *bz* = *bh* = 0. Identify values of *wz*, *uz*, *wh* and *uh* for a GRU cell that would allow it to replicate the behavior described by the automaton above.

1. (6 points) (written) **Modeling toggling behavior.** Now, let us try modeling a more interesting  
   behavior. We are now given an arbitrary input sequence, and must produce an output sequence  
   that switches from 0 to 1 and vice versa whenever it sees a 1 in the input. For example, the input  
   sequence 00100100 should produce 00111000. This behavior could be described by the following  
   automaton:



Once again, assume that the state is initialized at 0 (i.e. *h*(0) = 0), that all the parameters are scalars, that all sigmoid activations and tanh activations are replaced by the indicator function.

i. (3 points) Show that a 1D RNN can not replicate the behavior described by the automaton above.

ii. (3 points) Let *wr* = *ur* = *bz* = *bh* = 0. Identify values of *br*, *wz*, *uz*, *wh* and *uh* for a GRU cell that would allow it to replicate the behavior described by the automaton above.

(c) (6 points) (code) Implement the GRU cell described above in q3 gru cell.py. You can test your  
implementation by running python q3 gru cell.py test.

(d) (6 points) (code) We will now use an RNN model to try and learn the latching behavior described  
in part (a) using TensorFlow’s RNN implementation: tf.nn.dynamic rnn.  
i. In q3 gru.py, implement add prediction op by applying TensorFlow’s dynamic RNN  
model on the sequence input provided. Also apply a sigmoid function on the final state to  
normalize the state values between 0 and 1.  
ii. Next, write code to calculate the gradient norm and implement gradient clipping in add training op.

iii. Run the program: python q3 gru.py predict -c [rnn|gru] [-g]

to generate a learning curve for this task for the RNN and GRU models. The -g flag activates gradient clipping.

These commands produce a plot of the learning dynamics in q3-noclip-<model>.png and q3-clip-<model>.png respectively.  
**Deliverable:** Attach the plots of learning dynamics generated for a GRU and RNN, with and  
without gradient clipping (in total 4 plots) to your write up.

(e) (5 points) (written) Analyze the graphs obtained above and describe the learning dynamics you  
see. Make sure you address the following questions:  
i. Does either model experience vanishing or exploding gradients? If so, does gradient clipping  
help?  
ii. Which model does better? Can you explain why?

(f) (3 points) (code) Run the NER model from question 2 using the GRU cell using the command:  
python q2 rnn.py train -c gru  
Training should take about 3{4 hours on your CPU and about 30 minutes if you use the GPUs  
provided by Microsoft Azure. 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. 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 q2 rnn.py shell -m results/gru/<timestamp>/  
**Deliverable:** After your model has trained, copy the gru predictions.conll file from the  
appropriate results folder into your code directory so that it can be included in your submission.