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., $\mathbf{xW} + \mathbf{b}$ instead of $\mathbf{Wx} + \mathbf{b}$).

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 ($\mathbf{x}^{(t)}$) with the named entities tagged above each token ($\mathbf{y}^{(t)}$) as well as hypothetical predictions produced by a system ($\hat{\mathbf{y}}^{(t)}$):

$\mathbf{y}^{(t)}$	**ORG**	**ORG**	**O**	**O**	**O**	**ORG**	**ORG**	...	**O**	**PER**	**PER**	**O**
$\hat{\mathbf{y}}^{(t)}$	MISC	O	O	O	O	ORG	O	...	O	PER	PER	O
$\mathbf{x}^{(t)}$	American	Airlines,	a	unit	of	AMR	Corp.,	...	spokesman	Tim	Wagner	said.

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 $p = \frac{3}{4}$).
- 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 $r = \frac{3}{6}$).
- F_1 is the harmonic mean of the two: $F_1 = \frac{2pr}{p+r}$. (in the above example, it would be $F_1 = \frac{6}{10}$).

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 $\frac{1}{3}$.
- 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 $\frac{1}{3}$.
- Finally, the F_1 score is still the harmonic mean of the two, and would be $\frac{1}{3}$ 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.

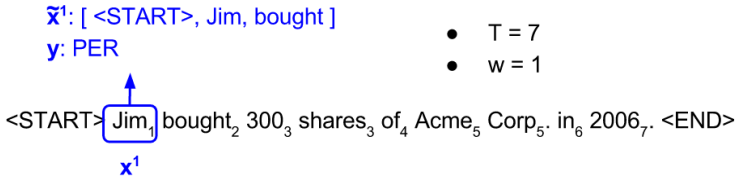


Figure 1: A sample input sequence

Figure 1 shows an example of an input sequence and the first window from this sequence. Let $\mathbf{x} \stackrel{\text{def}}{=} \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(T)}$ be an input sequence of length T and $\mathbf{y} \stackrel{\text{def}}{=} \mathbf{y}^{(1)}, \mathbf{y}^{(2)}, \ldots, \mathbf{y}^{(T)}$ be an output sequence,

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

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 left and right of $\mathbf{x}^{(t)}$: $\tilde{\mathbf{x}}^{(t)} \stackrel{\text{def}}{=} [\mathbf{x}^{(t-w)}, \ldots, \mathbf{x}^{(t)}, \ldots, \mathbf{x}^{(t+w)}]$; 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:

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

where $L \in \mathbb{R}^{V \times D}$ 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 (here 5).

(a) (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).

> **Solution:** Here are a couple of examples:
>
> - "Spokesperson for Levis, Bill Murray, said . . . ", where it is ambiguous whether Levis is a person or an organization.
> - "Heartbreak is a new virus," where "Heartbreak" could either be a MISC named entity (it's actually the name of a virus), or simply a noun.
>
> Basically, we are looking for any situation wherein the sentence contains a word.

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

> **Solution:** Often named entities can be rare words, e.g. peoples names or "heartbreak". Using features like casing helps the system generalize.

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.

Solution: Casing of words and parts of speech.

(b) (5 points) (written)

i. (2 points) What are the dimensions of $\mathbf{e}^{(t)}$, W and U if we use a window of size w?

Solution: $\mathbf{e}^{(t)}$ is $1 \times (2w+1)D$. W is $(2w+1)D \times H$. U is $H \times C$.

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

Solution: For a single window, it requires $O((2w+1)D)$ operations to compute $\mathbf{e}^{(t)}$, $O((2w+1)DH + H)$ to compute $\mathbf{h}^{(t)}$ and $O(HC + C)$ operations to compute $\mathbf{y}^{(t)}$.
In total, it requires $O((2w+1)DHT + HC)$ operations to predict labels for the entire sentence. Assuming that $D, H \gg C$, $O((2w+1)DHT)$ is also an acceptable answer.

(c) (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.

Solution: The best development F_1 we got on the corpus is 83% and has the following confusion matrix:

gold/guess	PER	ORG	LOC	MISC	O
PER	2973	59	41	14	62
ORG	152	1648	94	62	136
LOC	57	104	1868	25	40
MISC	47	58	45	1012	106
O	46	49	12	33	42619

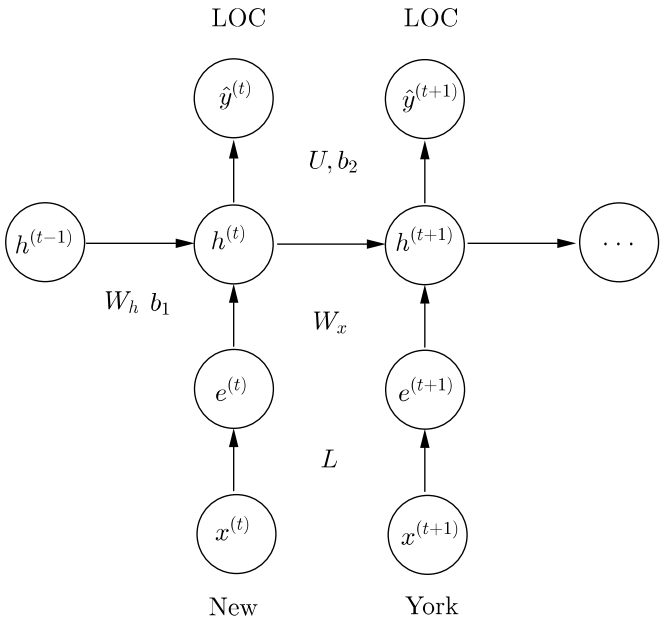
The confusion matrix shows that the biggest source of confusion for the model is the organization label, where many organizations are confused for people or missed out on. On the other hand, it seems like people are pretty well recognized.

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.

Solution: The window-based model can not use information from neighboring predictions to disambiguate labeling decisions, leading to non-contiguous entity predictions. Here, we concisely display examples with a word/gold/guess notation. For example the sentence, "Quarter-final results in the Hong/ORG/LOC Kong/ORG/ORG Open/ORG/ORG on Friday". The window around Kong can easily identify that it is part of an organization, while the window around Hong doesn't know whether it is part of a location or an organization. The window-based model also can't use information from other parts of the sentence: take this example: *" I'm an emotional player", said the 104th-ranked Tarango/PER/LOC.*. Here the sentence clearly describes Tarango as being a person (player), but this information can't be used by the window based model.

2. 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:

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

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 $\mathbf{b}_1 \in \mathbb{R}^H$ 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:

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

(a) (4 points) (written)

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

> **Solution:** The RNN model differs from the window based model in that (a) W_x has only $D \times H$ parameters instead of $(2w+1)D \times H$ for W in the window model and (b) it has $H \times H$ additional parameters for W_h.

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

> **Solution:** It takes $O(D)$ operations to compute $\mathbf{e}^{(t)}$, $O(H^2 + DH + H)$ operations to compute each $\mathbf{h}^{(t)}$ and $O(HC + C)$ operations to compute each $\hat{\mathbf{y}}^{(t)}$. In total, it takes about $O((D+H)HT)$ operations to predict labels for the whole sentence.

(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.

> **Solution:** Consider the scenario with a sentence with words/gold labels "The James/MISC scandal/MISC". If you predicted "The James/MISC scandal/O" versus "The James/O scandal/O", your cross entropy loss would decrease because you've predicted one more token's label correctly. On the other hand, your entity-level F_1 score would also decrease because you have now predicted another entity incorrectly: your precision goes down while your recall remains the same.

ii. (1 point) Why it is difficult to directly optimize for F_1?

> **Solution:** For starters, F_1 is not differentiable. Secondly, it is hard to directly optimze for F_1 because it requires predictions from the entire corpus to compute, making it very difficult to batch and parallelize.

(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 $\mathbf{x}$ and $\mathbf{y}$ to make them M tokens long. These "0-vectors" are still one-hot vectors, representing a new `NULL` token.
2. Create a *masking vector*, $(m^{(t)})_{t=1}^{M}$ 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.[3]
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:

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

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

> **Solution:** The loss includes the accuracy of predicting extra 0-labels. The gradients from the padding input would flow through the hidden state and affect the learning of the parameters. By masking the loss, we zero-out the loss due to these extra 0-labels (and hence their gradients), solving the 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 W_x and W_h 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>/`

[3] In our code, we will actually use the Boolean values of `True` and `False` instead of 1 and 0 for computational efficiency.

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.

Solution: One limitation is that the model does not get to see into the future while making predictions; e.g. "New York State University". The first New is likely to be tagged "LOC" (for New York). This problem can be solved with a biRNN.
Another limitation is that the model doesn't enforce that adjacent tokens have the same tag (illustrated by the same example above). Introducing a pair-wise agreements (i.e. using a CRF loss) in the loss would solve this problem.

3. 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:

$$\begin{aligned}
\mathbf{z}^{(t)} &= \sigma(\mathbf{x}^{(t)}U_z + \mathbf{h}^{(t-1)}W_z + \mathbf{b}_z) \\
\mathbf{r}^{(t)} &= \sigma(\mathbf{x}^{(t)}U_r + \mathbf{h}^{(t-1)}W_r + \mathbf{b}_r) \\
\tilde{\mathbf{h}}^{(t)} &= \tanh(\mathbf{x}^{(t)}U_h + \mathbf{r}^{(t)} \circ \mathbf{h}^{(t-1)}W_h + \mathbf{b}_h) \\
\mathbf{h}^{(t)} &= \mathbf{z}^{(t)} \circ \mathbf{h}^{(t-1)} + (1 - \mathbf{z}^{(t)}) \circ \tilde{\mathbf{h}}^{(t)},
\end{aligned}$$

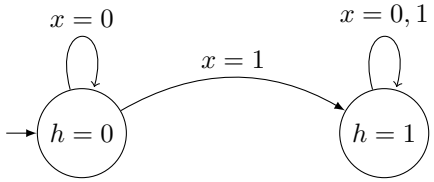
where $\mathbf{z}^{(t)}$ is considered to be an *update gate* and $\mathbf{r}^{(t)}$ is considered to be a *reset gate*.[4]

Also, to keep the notation consistent with the GRU, for this problem, let the basic RNN cell be described by the equations:

$$\mathbf{h}^{(t)} = \sigma(\mathbf{x}^{(t)}U_h + \mathbf{h}^{(t-1)}W_h + \mathbf{b}_h).$$

To gain some inutition, 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

[4]Section 4.2.2 of `https://arxiv.org/pdf/1511.07916.pdf` provides a good introduction to GRUs. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/` provides a more colorful picture of LSTMs and to an extent GRUs.

replaced by the indicator function:

$$\sigma(x) \to \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \qquad \tanh(x) \to \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}.$$

i. (1 point) Identify values of w_h, u_h and b_h for an RNN cell that would allow it to replicate the behavior described by the automaton above.

Solution: For the RNN cell

$$\mathbf{h}^{(t)} = \sigma(\mathbf{x}^{(t)}U_h + \mathbf{h}^{(t-1)}W_h + \mathbf{b}_h)$$

we consider the following four cases:

- $\mathbf{h}^{(t-1)} = 0, \mathbf{x}^{(t)} = 0$. Since we want $\mathbf{h}^{(t)} = 0$, we have

$$\begin{aligned} \sigma(\mathbf{b}_h) &= 0 \\ \mathbf{b}_h &\leq 0 \end{aligned}$$

- $\mathbf{h}^{(t-1)} = 0, \mathbf{x}^{(t)} = 1$. Since we want $\mathbf{h}^{(t)} = 1$, we have

$$\begin{aligned} \sigma(U_h + \mathbf{b}_h) &= 1 \\ U_h + \mathbf{b}_h &> 0 \end{aligned}$$

- $\mathbf{h}^{(t-1)} = 1, \mathbf{x}^{(t)} = 0$. Since we want $\mathbf{h}^{(t)} = 1$, we have

$$\begin{aligned} \sigma(W_h + \mathbf{b}_h) &= 1 \\ W_h + \mathbf{b}_h &> 0 \end{aligned}$$

- $\mathbf{h}^{(t-1)} = 1, \mathbf{x}^{(t)} = 1$. Since we want $\mathbf{h}^{(t)} = 1$, we have

$$\begin{aligned} \sigma(W_h + U_h + \mathbf{b}_h) &= 1 \\ W_h + U_h + \mathbf{b}_h &> 0 \end{aligned}$$

Therefore, the inequalities $W_h, U_h, \mathbf{b}_h$ should satisfy are:

$$\begin{aligned} \mathbf{b}_H &\leq 0 \\ U_h + \mathbf{b}_h &> 0 \\ W_h + \mathbf{b}_h &> 0 \end{aligned}$$

ii. (3 points) Let $w_r = u_r = b_r = b_z = b_h = 0$. Identify values of w_z, u_z, w_h and u_h for a GRU cell that would allow it to replicate the behavior described by the automaton above.

Solution:

Since $w_r = u_r = b_r = b_z = b_h = 0$, the GRU cell can be simplified as:

$$\begin{aligned}
\mathbf{z}^{(t)} &= \sigma(\mathbf{x}^{(t)}U_z + \mathbf{h}^{(t-1)}W_z) \\
\mathbf{r}^{(t)} &= 0 \\
\tilde{\mathbf{h}}^{(t)} &= \tanh(\mathbf{x}^{(t)}U_h) \\
\mathbf{h}^{(t)} &= \mathbf{z}^{(t)} \circ \mathbf{h}^{(t-1)} + (1 - \mathbf{z}^{(t)}) \circ \tilde{\mathbf{h}}^{(t)}
\end{aligned}$$

We consider the following four cases:

- $\mathbf{h}^{(t-1)} = 0, \mathbf{x}^{(t)} = 0$. Since we want $\mathbf{h}^{(t)} = 0$, we have

$$\begin{aligned}
\mathbf{z}^{(t)} &= \sigma(0) = 0 \\
\tilde{\mathbf{h}}^{(t)} &= \tanh(0) = 0 \\
\mathbf{h}^{(t)} &= 0
\end{aligned}$$

- $\mathbf{h}^{(t-1)} = 0, \mathbf{x}^{(t)} = 1$. Since we want $\mathbf{h}^{(t)} = 1$, we have

$$\begin{aligned}
\mathbf{z}^{(t)} &= \sigma(U_z) \\
\tilde{\mathbf{h}}^{(t)} &= \tanh(U_h) = 0 \\
\mathbf{h}^{(t)} &= (1 - \sigma(U_z)) \circ \tanh(U_h) = 1 \\
\rightarrow U_z &\leq 0 \\
U_h &> 0
\end{aligned}$$

- $\mathbf{h}^{(t-1)} = 1, \mathbf{x}^{(t)} = 0$. Since we want $\mathbf{h}^{(t)} = 1$, we have

$$\begin{aligned}
\mathbf{z}^{(t)} &= \sigma(W_z) \\
\tilde{\mathbf{h}}^{(t)} &= \tanh(0) = 0 \\
\mathbf{h}^{(t)} &= \mathbf{z}^{(t)} \circ \mathbf{h}^{(t-1)} = \sigma(W_z) = 1 \\
\rightarrow W_z &> 0
\end{aligned}$$

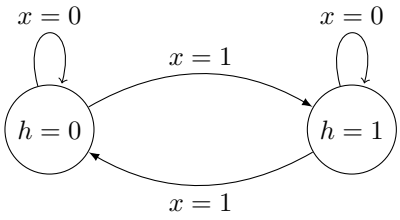
- $\mathbf{h}^{(t-1)} = 1, \mathbf{x}^{(t)} = 1$. Since we want $\mathbf{h}^{(t)} = 1$, we have

$$\begin{aligned}
\mathbf{z}^{(t)} &= \sigma(U_z + W_z) \\
\tilde{\mathbf{h}}^{(t)} &= \tanh(U_h) = 0 \\
\mathbf{h}^{(t)} &= \mathbf{z}^{(t)} + (1 - \sigma(U_z)) \circ \tanh(U_h) = 1 \\
\rightarrow U_z + W_h &> 0 \quad \textit{or} \quad U_z + W_h \leq 0 \\
\rightarrow W_h &\in R
\end{aligned}$$

Therefore, the inequalities W_z, U_z, W_h, U_h should satisfy are:

$$\begin{aligned}
W_z &> 0 \\
U_z &\leq 0 \\
W_h &\in R \\
U_h &> 0
\end{aligned}$$

(b) (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.

Solution: First of all, we need that if input is 0, the state maintains its value:

$$0 \times w_h + 0 \times u_h + b_h \leq 0$$
$$1 \times w_h + 0 \times u_h + b_h > 0.$$

This implies that we *must* have $w_h > 0$.
Next, we need to be able to model the property that when the state needs to flip from 0 to 1 when seeing an input of 1 and flip back from 1 to 0 when seeing a 1 again. In equations, this requires that:

$$0 \times w_h + 1 \times u_h + b_h > 0$$
$$1 \times w_h + 1 \times u_h + b_h \leq 0$$

However, this implies that we *must* have $w_h < 0$, which contradicts the earlier condition!

ii. (3 points) Let $w_r = u_r = b_z = b_h = 0$. Identify values of b_r, w_z, u_z, w_h and u_h for a GRU cell that would allow it to replicate the behavior described by the automaton above.

Solution: First, let's set $b_r = 1$ to "deactivate" the reset gate. We'll exploit the update gate that the GRU has, setting it to be 1 only when $x = 1$ (and 0 otherwise). We'll also set the pre-output gate, $\tilde{h}$ to be less than 0 when $h = 1$ and greater than 0 otherwise. We can achieve the first property by setting $u_z = 1$, $b_z = w_z = 0$, and the second property by setting $u_h = 0$, $b_h = 1$, $w_h = -2$.

(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.

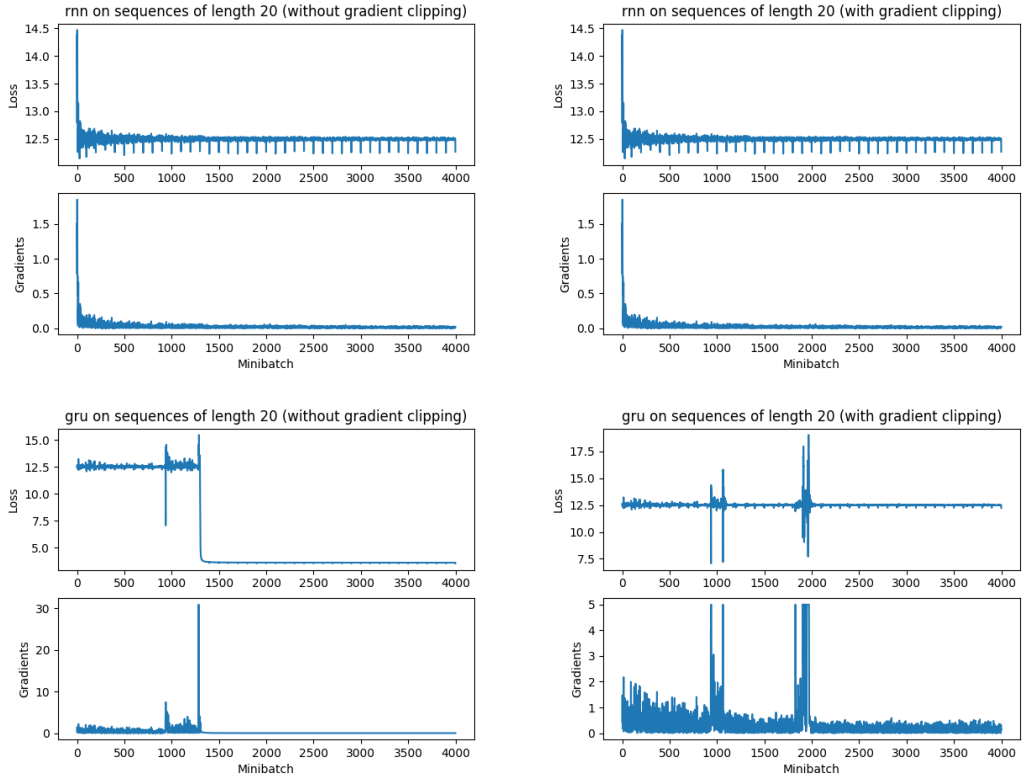
Deliverable: Attach the plots of learning dynamics generated for a GRU and RNN, with and without gradient clipping (in total 4 plots) to your write up.

(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?

Solution:



The plots for this question vary a bit based on initialization, and we are evaluating their descriptions of what they see. Some things that we are looking for: (a) gradients should eventually decrease (or something is definitely fishy).

The RNN's gradients vanish quickly, so clipping doesn't really help. For the GRU, gradients are maintained and it can improve on the loss. That said, clipping can prevent these gradients from improving the loss.

(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.