Natural Language Understanding (2017–2018)

School of Informatics, University of Edinburgh
Clara Vania and Adam Lopez
Credits: much of this material was developed by Mirella Lapata, Frank Keller, and Richard Socher

NLU Coursework: Recurrent Neural Networks

The assignment is due 16th March 2018, 16:00.

Pair work policy. You are encouraged to work with a partner on this assignment. When you work with another person, you learn more, because you need to explain things to each other as you pool your collective expertise to solve problems, and explaining something to another person helps you debug your own thinking. For this reason, I encourage you to seek partners with complementary skills to your own. You may not work in teams of three or more.

If you work with a partner, only one of you should submit your completed work. But I need to know that the submission represents the work of two people. So **you must do the following to ensure that both of you receive credit:** Email ⟨alopez@inf.ed.ac.uk⟩ with the subject *1111824013*. The body of the email should include:

1. Your name.
2. Your UUN.
3. Your partner's name, and
4. Your partner's UUN.

You **must** cc: your partner on the email, and you will both receive a confirmation from me within two working days.

You and your partner will receive identical marks, and you may not change partners after you have emailed me, so you should this commitment seriously. I will ignore emails received after the deadline. **I advise you to choose your partner now and get to work.** You are welcome to use piazza to search for a partner.

Submission Information You are encouraged to work in pairs on this assignment. When you work with another person, you will learn more, because you will need to explain things to each other as you pool your collective expertise to solve problems.

Your solution should be delivered in two parts and uploaded to Blackboard Learn. **Please do not include your name or your partner's name in either the code or the writeup.** The coursework will be marked anonymously since this has been empirically shown to reduce bias. (I anonymize the filenames before marking.)

For your writeup:

- Write up your answers in a file titled `<UUN>.pdf`. For example, if your UUN is `S123456`, your corresponding PDF should be named `S123456.pdf`.
- The answers should be clearly numbered and can contain text, diagrams, graphs, formulas, as appropriate. Do not repeat the question text. If you are not comfortable with writing math on Latex/Word you are allowed to include scanned handwritten answers in your submitted pdf. You will lose marks if your handwritten answers are illegible.
- On Blackboard Learn, select the Turnitin Assignment "NLU Coursework ANSWERS". Upload your `<UUN>.pdf` to this assignment, and use the submission title `<UUN>`. So, for above example, you should enter the submission title `B123456`.
- Please make sure you have submitted the right file. We cannot make concessions for students who turn in incomplete or incorrect files by accident.

For your code and parameter files:

- Compress your code for `rnn.py` as well as your saved parameters `rnn.U.npy`, `rnn.V.npy`, and `rnn.W.npy` into a ZIP file named `<UUN>.zip`. For example, if your UUN is `S123456`, your corresponding ZIP should be named `S123456.zip`.
- On Blackboard Learn, select the Turnitin Assignment "NLU Coursework CODE". Upload your `<UUN>.zip` to this assignment, and use the submission title `<UUN>`. So, for above example, you should enter the submission title `S123456`.

Good Scholarly Practice Please remember the University requirement as regards all assessed work for credit. Details and advice about this can be found at:

`http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct`

and links from there. Note that you are required to take reasonable measures to protect your assessed work from unauthorised access. For example, if you put any such work on a public repository then you must set access permissions appropriately (generally

permitting access only to yourself, or your group in the case of group practicals). For your writeup, and particularly on the final question, you should pay close attention to the guidance on plagiarism. Your instructor is **very good** at detecting plagiarism that even Turnitin can't spot, so it does not suffice to simply pass the Turnitin check. In short, if you have borrowed or lightly edited someone else's words, you have plagiarised. Write your text in your own words.

Recommended Milestones The entire assignment is due on 16 March. However, I recommend that you observe some intermediate milestones to pace yourself. You should be able to complete Question 1 and 2 by 5th March, and Question 3 by 9th March, leaving you a week to work on Question 4. This timeline is consistent with past versions of the coursework, which many students completed successfully. (Note that this version is not identical to past versions.)

Assignment Data The necessary files for this assignment are available on the NLU course page.

Python Virtual Environment For this assignment you will be using Python along with a few open-source packages. These packages cannot be installed directly, so you will have to create a virtual environment. We are using virtual environments to make the installation of packages and retention of correct versions as simple as possible. For this assignment we are going to use Miniconda + Python 3.5.

Open a terminal on a DICE machine and follow these instructions. We are expecting you to enter these commands in one-by-one. Waiting for each command to complete will help catch any unexpected warnings and errors.

Note: You can skip step (1) and (2) if you already have Miniconda installed in your machine.

1. Open a terminal on DICE machine and type the following command:
   ```
   $> wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh
   $> bash Miniconda3-latest-Linux-x86_64.sh
   ```
2. Restart your terminal.
3. Create conda virtual environment:
   ```
   $> conda create -n nlu python=3.5
   ```
4. Activate conda environment:
   ```
   $> source activate nlu
   ```

5. Install packages that we need:
   `$> conda install numpy gensim pandas`

You should now have all the required packages installed. You only need to create the virtual environment and perform the package installations (step 1-5) **once**. However, make sure you activate your virtual environment (step 4) **every time** you open a new terminal to work on your NLU assignment. Remember to use the `source deactivate` command to disable the virtual environment when you don't need it. If you encounter any unexpected issues, please e-mail the course TA's as soon as possible.

Terminology/definitions Most neural network components, as well as their architecture and functionality, can be described using *matrix* and *vector* mathematical operations. Matrix and vector notation in the literature is somewhat inconsistent, so for this assignment we will use the following conventions:

(1) *Matrices* are assigned bold capital letters, e.g., $\mathbf{U}$, $\mathbf{V}$, $\mathbf{W}$.

(2) *Vectors* are written in bold lower-cased letters, e.g., $\mathbf{x}$, $\mathbf{s}$, $\mathbf{net}_{in}$, $\mathbf{net}_{out}$.

(3) For a matrix $\mathbf{M}$ and a vector $\mathbf{v}$, $\mathbf{Mv}$ represents their *matrix-vector (dot) product*.[1]

(4) For vectors $\mathbf{v}$ and $\mathbf{w}$ of equal length n, $\mathbf{v} \circ \mathbf{w}$ represents their *element-wise product*:

$$\mathbf{v} \circ \mathbf{w} = [v_0 w_0, v_1 w_1, \ldots, v_n w_n]$$

Similarly, $\mathbf{v} + \mathbf{w}$ and $\mathbf{v} - \mathbf{w}$ express element-wise addition and subtraction.

(5) For vectors $\mathbf{v}$ and $\mathbf{w}$, $\mathbf{v} \otimes \mathbf{w}$ represents their *outer product*.[2]

(6) In recurrent neural networks, we process a *sequence* (or *time*), where each component is in a different state depending on the position in the sequence (or time step). We use the notation $\mathbf{M}^{(t)}$, $\mathbf{v}^{(t)}$ to refer to matrices and vectors at time step t.

Provided code and use of NumPy We provide the template file `rnn.py` which you must use to write your code. We also provide an additional module, `rnnmath.py`, which consists of helper functions you can use. Please familiarize yourself with the provided code and make sure you **don't** change the provided function signatures.

Throughout this assignment, the use of NumPy methods for all matrix/vector operations is *strongly* advised. Please **do not** try to implement matrix/vector functionality on your

[1] `https://en.wikipedia.org/wiki/Matrix_multiplication`
[2] `https://en.wikipedia.org/wiki/Outer_product`

own. If you need help with NumPy, please refer to its documentation[3] and look for answers on Google before asking on Piazza.

Introduction

For this assignment, you are asked to implement some basic functionality of a Recurrent Neural Network (RNN) for Language Modeling (LM).

Language modeling is a central task in NLP, and language models are at the heart of speech recognition, machine translation, and many other systems. Given a word sequence $w_1, w_2, \ldots, w_t$, a language model predicts the next word w_{t+1} by modeling:

$$P(w_{t+1} \mid w_1, \ldots, w_t).$$

Below, you will be introduced to the main elements of a simple RNN for LM, based on the model proposed by Mikolov *et al.* (2010). In Question 2, you will implement its core word prediction functionality and its training by implementing a loss function and the model's gradient accumulation through backpropagation. In Question 3, you will train and fine-tune your language model on actual data and use it to generate sentences. In Question 4, you will adapt your language model to a new task.

Recurrent Neural Networks

A recurrent neural network for language modeling uses feedback information in the hidden layer to model the "history" $w_1, w_2, \ldots, w_t$ in order to predict w_{t+1}. Formally, at each time step, the model needs to compute:

$$\mathbf{s}^{(t)} = f\left(\mathbf{net}_{in}^{(t)}\right) \quad (1)$$

$$\mathbf{net}_{in}^{(t)} = \mathbf{V}\mathbf{x}^{(t)} + \mathbf{U}\mathbf{s}^{(t-1)} \quad (2)$$

$$\hat{\mathbf{y}}^{(t)} = g\left(\mathbf{net}_{out}^{(t)}\right) \quad (3)$$

$$\mathbf{net}_{out}^{(t)} = \mathbf{W}\mathbf{s}^{(t)} \quad (4)$$

where $f()$ and $g()$ are the *sigmoid* and *softmax* activation functions respectively, $\mathbf{x}^{(t)}$ is the one-hot vector representing the vocabulary index of the word w_t, $\mathbf{net}_{in}^{(t)}$ and $\mathbf{net}_{out}^{(t)}$

[3] `http://docs.scipy.org/doc/numpy/reference/index.html`

are the activations for the hidden and output layers, and $\mathbf{s}^{(t)}$ and $\hat{\mathbf{y}}^{(t)}$ are the corresponding hidden and output vectors produced after applying the *sigmoid* and *softmax* non-linearities.

For a given input $[w_1, w_2, \ldots, w_t]$, the probability of the next word at time step $t+1$ can be read from the output vector $\hat{\mathbf{y}}^{(t)}$:

$$P(w_{t+1} = j \mid w_t, \ldots, w_1) = \hat{y}_j^{(t)} \tag{5}$$

The parameters to be learned are:

$$\mathbf{U} \in \mathbb{R}^{D_h \times D_h} \quad \mathbf{V} \in \mathbb{R}^{D_h \times |V|} \quad \mathbf{W} \in \mathbb{R}^{|V| \times D_h} \tag{6}$$

where $\mathbf{U}$ is the matrix for the recurrent hidden layer, $\mathbf{V}$ is the input word representation matrix, $\mathbf{W}$ is the output word representation matrix, and D_h is the dimensionality of the hidden layer.

Question 1: Training RNNs [30 marks]

When training RNNs, we need to propagate the errors observed at the output layer $\hat{\mathbf{y}}$ back through the network, and adjust the weight matrices $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{W}$ to minimize the observed loss w.r.t. a desired output. There are several loss functions suitable for use in RNNs. In RNN language models, an effective loss function is the (un-regularized) cross-entropy loss:

$$J^{(t)}(\theta) = -\sum_{j=1}^{|V|} d_j^{(t)} \log \hat{y}_j^{(t)} \tag{7}$$

where $\mathbf{d}^{(t)} = [d_1^{(t)}, d_2^{(t)}, \ldots, d_{|V|}^{(t)}]$ is the one-hot vector representing the vocabulary index of desired output word at time t. This is a point-wise loss (i.e. for a single word prediction). In order to evaluate the model's performance, we average the cross-entropy loss across all steps in a sentence and across all sentences in the dataset.

(a) In the file `rnn.py`, implement the method `predict` of the `RNN` class. The method is used for *forward prediction* in your RNN and takes as input a sentence as a list of word indices $[w_1, \cdots, w_n]$. The return values are the matrices produced by concatenating hidden vectors $\mathbf{s}^{(t)}$ and output vectors $\hat{\mathbf{y}}^{(t)}$ for $t = 1, 2, \ldots, n$.[4] **[5 marks]**

[4] See the provided documentation on `rnn.py` for more details on the functions you need to implement.

(b) In `rnn.py`, implement the methods `compute_loss` and `compute_mean_loss`. Given a sequence of input words $w = [w_1, \ldots, w_n]$ and a sequence of desired output words $d = [d_1, \ldots, d_n]$, `compute_loss` should return the total loss produced by the model's predictions for the sentence. The `compute_mean_loss` should compute the average loss over a corpus of input sentences. Here, we average across all words in all sentences of the given corpus. **[5 marks]**

Optimizing the loss using backpropagation means we have to calculate the update values Δ w.r.t. the gradients of our loss function for the observed errors. For the output layer weights, at time step t we accumulate the matrix $\mathbf{W}$ updates using:

$$\Delta \mathbf{W} = \eta \sum_{p=1}^{n} \delta_{out,p}^{(t)} \otimes \mathbf{s}_p^{(t)} \tag{8}$$

$$\delta_{out,p}^{(t)} = (\mathbf{d}_p^{(t)} - \hat{\mathbf{y}}_p^{(t)}) \circ g'(\mathbf{net}_{out,p}^{(t)}) \tag{9}$$

where η is the learning rate and p indicates the index of the current training pattern (sentence). We then further propagate the error observed at the output back to $\mathbf{V}$ with:

$$\Delta \mathbf{V} = \eta \sum_{p=1}^{n} \delta_{in,p}^{(t)} \otimes \mathbf{x}_p^{(t)} \tag{10}$$

$$\delta_{in,p}^{(t)} = \mathbf{W}^T \delta_{out,p}^{(t)} \circ f'(\mathbf{net}_{in,p}^{(t)}) \tag{11}$$

The derivatives of the softmax and sigmoid functions are respectively given as[5]:

$$g'(\mathbf{net}_{out,p}^{(t)}) = \vec{1} \tag{12}$$

$$f'(\mathbf{net}_{in,p}^{(t)}) = \mathbf{s}_p^{(t)} \circ (\vec{1} - \mathbf{s}_p^{(t)}) \tag{13}$$

Finally, in order to update the recurrent weights $\mathbf{U}$, we need to look back one step in time:

$$\Delta \mathbf{U} = \eta \sum_{p=1}^{n} \delta_{in,p}^{(t)} \otimes \mathbf{s}_p^{(t-1)} \tag{14}$$

(c) In `rnn.py`, implement the method `acc_deltas` that accumulates the weight updates for $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{W}$ for a simple backpropagation through the RNN, where we only look back one step in time as described above. **[10 marks]**

[5] We use $\vec{1}$ as shorthand for the all-ones vector of appropriate length.

Now we have implemented simple backpropagation (BP) for recurrent networks—that is, RNNs that just look at the previous hidden layer when accumulating $\Delta\mathbf{U}$ and $\Delta\mathbf{V}$. An extension to standard BP is backpropagation through time (BPTT), which takes into account the previous τ time steps during backpropagation. At time t, the updates $\Delta\mathbf{W}$ can be derived as before. For $\Delta\mathbf{U}$ and $\Delta\mathbf{V}$, we additionally recursively update at times $(t-1)$, $(t-2)$ $\cdots(t-\tau)$:

$$\Delta\mathbf{V} = \eta \sum_{p=1}^{n} \delta_{in,p}^{(t-\tau)} \otimes \mathbf{x}_p^{(t-\tau)} \tag{15}$$

$$\Delta\mathbf{U} = \eta \sum_{p=1}^{n} \delta_{in,p}^{(t-\tau)} \otimes \mathbf{s}_p^{(t-\tau-1)} \tag{16}$$

$$\delta_{in,p}^{(t-\tau)} = \mathbf{U}^T \delta_{in,p}^{(t-\tau+1)} \circ f'(\mathbf{net}_{in,p}^{(t-\tau)}) \tag{17}$$

(d) Implement the method `acc_deltas_bptt` that accumulates the weight updates for **U**, **V** and **W** using backpropagation through time for τ time steps. **[10 marks]**

Question 2: Language Modeling [20 marks]

By now you should have everything in place to train a full Recurrent Neural Network using backpropagation through time. In the following questions, we will use the training and development data provided in `wiki-train.txt` and `wiki-dev.txt`. The training data consists of sentences from the parsed English Wikipedia corpus from Linzen *et al.* (2016), and each input/output pair x, d is of the form $[w_1, \cdots w_n]$ / $[w_2, \cdots w_{n+1}]$ that is, the desired output is always the next word of the current input:

time index	t=1	t=2	t=3	t=4
input:	Banks	struggled	with	the
output:	struggled	with	the	crisis

The `utils.py` module provides functions to read the Wikipedia data, and the `__main__` method of the `rnn.py` module provides some starter code for training your models. Use mode `train-lm` to train your language model.

(a) Perform parameter tuning using a subset of the training and development sets. Use a fixed vocabulary of size 2000, and vary the number of hidden units (at least: 25, 50), the look-back in backpropagation (at least: 0, 2, 5), and learning rate (at least: 0.5, 0.1, 0.05). The mode `train-lm` in `rnn.py` allows for more parameters, which you are free to explore. You should tune your model to maximize generalization

performance (minimize cross-entropy loss) on the dev set. For these experiments, use the first 1000 sentences of both the training and development sets and train for 10 epochs.[6] Report your findings. **[10 marks]**

(b) Using your best parameter settings found in (a), train an RNN on a much larger training set. Use a fixed vocabulary size of 2000, 25000 training sentences and, as before, use the first 1000 development sentences to evaluate the model's performance during training. When your model is trained[7], evaluate it on the full dev set and report the mean loss, as well as both the adjusted and unadjusted perplexity your model achieves[8]. Save your final learned matrices **U**, **V** and **W** as files `rnn.U.npy`, `rnn.V.npy` and `rnn.W.npy`, respectively. **[10 marks]**

Question 3: Predicting Subject-Verb Agreement [15 marks]

The form of an English third-person present tense verb depends on whether the **head** of the syntactic subject is plural or singular. For example, native English speakers strongly prefer sentences (i) and (iv) below, and regard (ii) and (iii) as ungrammatical:

i) The **key** is on the table.

ii) *The **key** are on the table.

iii) *The **keys** is on the table.

iv) The **keys** are on the table.

This agreement tends to persist even when the head of the subject is separated from the verb by intervening words:

v) The **keys** to the cabinet are on the table.

Agreement rules like this are widespread, and are often more complex than in English. Our goal for this question will be to test (in a limited way) whether an RNN can also learn them. For our first test, we will train a model predict agreement using direct supervision. That is, we will give our model the sequence of words preceding the verb, and we will ask it to predict whether the verb is singular (VBZ), or plural (VBP). Our training and test data will be in this form:

[6] Note that training models might take some time. For example, a sweep of the parameters settings described above should take roughly 2 hours on a student lab DICE machine. Please avoid using `student.compute` to train your models as run times will become very slow on a busy server.

[7] This should also take roughly 2 hours. Again, avoid using `student.compute`.

[8] Use your method `compute_mean_loss` to calculate loss on the development set, and the provided method `adjust_loss` to get adjusted/unadjusted perplexities for your models

time index	t=1	t=2	t=3	t=4	t=5
input:	The	keys	to	the	cabinet
output:					VBP

Since the head of the subject may be arbitrarily far from the verb, this problem is a natural application of RNNs, which can encode the input sentence. But since the task is now binary classification, we must make some changes to the RNN. Instead of making predictions at **every** time step, we only make a prediction at the **final** time step.

(a) Implement new functions for weight updates (`acc_deltas_bptt_np`), loss fuction (`compute_loss_np`), and prediction accuracy (`compute_acc_np`) to reflect the structure of the number prediction problem. **[10 marks]**

Check your implementation by running the code:

```
$> python rnn.py train-np data_dir hdim lookback learning_rate
```

(b) Train your new model, explaining the parameters you used and how you chose them, and include the results in your report. **[5 marks]**

Question 4: Number Prediction with an RRNLM [35 marks]

Let's consider the problem from question 3 from a slightly different perspective, inspired by the Linzen *et al.* (2016) paper that provided our data. Human learners of language generally learn morphosyntactic features of language, like number agreement, with little to no direct supervision. Can a computational model like an RNN also do this? That is, can it learn agreement simply from the language data itself?

Let's return to the RNNLM that you implemented in Question 1 and trained in Question 2. Suppose once again that your input looks like this:

time index	t=1	t=2	t=3	t=4	t=5
Input $x =$	The	keys	to	the	cabinet

To use an RNN for the task in question 3, we simply say:

$$\text{Output } = \begin{cases} \text{VBZ if } P(\text{is} \mid x) > P(\text{are} \mid x) \\ \text{VBP otherwise} \end{cases}$$

(a) Implement method `compare_num_pred` and evaluate your prediction accuracy:
`$> python rnn.py predict-lm rnn.py data_dir rnn_dir`
Include your result in your final report. **[4 marks]**

(b) The final part of this question is open-ended. I want you to ask your **own** question about the number prediction task and RNNs, and then attempt to answer it—put another way, I want you to develop a simple scientific hypothesis and test it. Your answer should be supported by **empirical evidence**, so you'll almost certainly need to analyze the data and count something. You are welcome (but not required) to implement another model, if doing so helps you answer a question. But this question is not a test your implementation skills: a simple, well-posed analysis or experiment, clearly explained, will often be better than a implementation of the fanciest RNN variant that you found while browsing the latest papers on arxiv.org. This question is also not a test of the **amount** of work that you do: I realize that you have limited time for this part of the coursework, and I don't expect you to produce a novel result or even a novel question. I do expect you to ask a small question that you find interesting, even if it's a question you've seen elsewhere, or just something small that intrigued you about the data. It is perfectly ok if you present an interesting question and run a well-posed analysis whose results are inconclusive—that is often how science works.[9] In short, the brief for this question is for you to be curious and engage with the topic of the course. Your answer should tell me something interesting that you learned (or attempted to learn) and it will be marked according to the descriptors for the common marking scheme.[10] (Note that these criteria require "elements of personal insight / creativity / originality" for a mark above 70—that's why this question is worth 31 marks.) Your answer should not be more than two pages, including figures. **[31 marks]**

References

Jiang Guo. Backpropagation Through Time. *Unpubl. ms., Harbin Institute of Technology*, 2013.

Tal Linzen, Emmanuel Dupoux, and Yoav Goldberg. Assessing the ability of LSTMs to learn syntax-sensitive dependencies. *Transactions of the Association for Computational Linguistics*, 4:521–535, 2016.

Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *INTERSPEECH*, volume 2, page 3, 2010.

[9] "Research is the process of going up alleys to see if they are blind."—Marston Bates

[10] `https://info.maths.ed.ac.uk/assets/files/Projects/college-criteria.pdf`

Acknowledgements

This assignment is based in parts on code and text kindly provided by Richard Socher, from his course on "Deep Learning for Natural Language Processing"[11].

[11] http://cs224d.stanford.edu/