Natural Language Understanding (2016–2017)

School of Informatics, University of Edinburgh
Mirella Lapata and Frank Keller

Assignment 2: Recurrent Neural Networks

The assignment is due 28th March 2017, 16:00.

Submission Information Your solution should be delivered in two parts and uploaded to Blackboard Learn.

For your writeup:

- Write up your answers in a file titled `<EXAM NO>.pdf`. For example, if your exam number is `B123456`, your corresponding PDF should be named `B123456.pdf`.
- The answers should be clearly numbered and can contain text, diagrams, graphs, formulas, as appropriate. Do not repeat the question text. This assignment involves performing some mathematical calculations by hand. If you are not comfortable with writing math on Latex/Word you are allowed to include scanned handwritten answers in your submitted pdf.
- On Blackboard Learn, select the Turnitin Assignment "Assignment 2a: Recurrent Neural Networks ANSWERS". Upload your `<EXAM NO>.pdf` to this assignment, and use the submission title `<EXAM NO>`. 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 `<EXAM NO>.zip`. For example, if your exam number is `B123456`, your corresponding ZIP should be named `B123456.zip`.
- On Blackboard Learn, select the Turnitin Assignment "Assignment 2b: Recurrent Neural Networks CODE". Upload your `<EXAM NO>.zip` to this assignment, and use the submission title `<EXAM NO>`. So, for above example, you should enter the submission title `B123456`.

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, in particular, 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).

Assignment Data The necessary files for this assignment are available on the NLU course page and on NLU's project space, which is accessible from DICE machines at `/afs/inf.ed.ac.uk/group/project/nlu/assignment2`. If you are working on your private machine, make sure you are able to access the data.

Python Virtual Environment – *Ignore if you have one ready from Assignment 1*
For both assignments 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. You can read here if you want to learn more about virtual environments: `https://virtualenv.pypa.io/en/stable/`.

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.

1. Change directory to home and create a virtual environment.
   ```
   $> cd
   $> virtualenv --distribute --python=/usr/bin/python2.7 virtualenvs/nlu
   ```
2. Navigate to and activate the virtual environment.
   ```
   $> cd virtualenvs/nlu
   $> source ./bin/activate
   ```
3. Install the python packages we need. Once the correct virtual environment is activated, `pip install` will install packages to that virtual environment only. These commands will produce long outputs.
   ```
   $> pip install -U pip
   $> pip install -U setuptools
   $> pip install numpy
   $> pip install gensim
   $> pip install pandas
   ```

You should now have all the required packages installed. You only need to create the virtual environment (step 1) and perform the package installations (step 3) **once**. However, make sure you activate your virtual environment (step 2) **every time** you open a new terminal to work on your NLU assignments. Remember to use the `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* andstrongly *vector* mathematical operations. As matrix and vector notation in the literature is somewhat inconsistent, for this assignment we will adhere to 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*:

$$\mathbf{v} \circ \mathbf{w} = [\mathrm{v}_0\mathrm{w}_0, \mathrm{v}_1\mathrm{w}_1, \ldots, \mathrm{v}_n\mathrm{w}_n]$$

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 deal with the notion of *sequence* (or *time*), where each component is in a different state depending on the time step. We use the notation $\mathbf{M}^{(t)}$, $\mathbf{v}^{(t)}$ to express the state of matrices and vectors at time step t.

Provided code and use of NumPy We provide the template file `rnn.py` which you have to 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 own. If you need help with NumPy, please refer to its documentation[3] and look for answers on Google before asking on Piazza.

[1]https://en.wikipedia.org/wiki/Matrix_multiplication
[2]https://en.wikipedia.org/wiki/Outer_product
[3]http://docs.scipy.org/doc/numpy/reference/index.html

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 can be found at the heart of speech recognition, machine translation, and many other systems. Given a word sequence $w_1, w_2, \dots, w_t$, a language model predicts the next word w_{t+1} by modeling:

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

In Question 1, you will be introduced to the main elements of a simple RNN for LM, based on the model proposed by Mikolov *et al.* (2010), and implement its core word prediction functionality. In Question 2, you will focus on the RNN's training by implementing a loss function and the model's gradient accumulation through back-propagation. Finally, in Question 3, you will train and fine-tune your language model on actual data and use it to generate sentences.

Question 1: Recurrent Neural Networks [31 points]

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

$$\begin{aligned}
\mathbf{s}^{(t)} &= f\left(\mathbf{net}_{in}^{(t)}\right) && (1)\\
\mathbf{net}_{in}^{(t)} &= \mathbf{V}\mathbf{x}^{(t)} + \mathbf{U}\mathbf{s}^{(t-1)} && (2)\\
\hat{\mathbf{y}}^{(t)} &= g\left(\mathbf{net}_{out}^{(t)}\right) && (3)\\
\mathbf{net}_{out}^{(t)} &= \mathbf{W}\mathbf{s}^{(t)} && (4)
\end{aligned}$$

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

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 $\hat{\mathbf{y}}^{(t)}$:

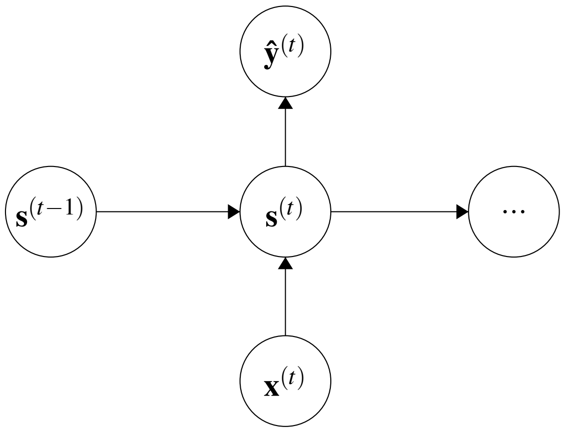
$$P(w_{t+1} = j \mid w_t, \ldots, w_1) = \hat{y}_j^{(t)} \tag{5}$$

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.

(a) Below you see a visual representation of an RNN at time step t. Expand the schematic to time steps $t+1$, $t+2$. [6 points]



(b) In this assignment, we use the *sigmoid* and *softmax* transfer functions to produce vectors at the hidden and output layers respectively. While we are free to choose other transfer functions, explain why we want to avoid using the *sigmoid* function at the output layer, and instead opt for a *softmax*. [5 points]

(c) Given the matrices $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{W}$ below, calculate the hidden layer $\mathbf{s}^{(1)}$, for the given 1-hot input vector $\mathbf{x}^{(1)} = [0\ 1\ 0]^T$. Assume that the hidden layer at time $t=0$ is the vector $s^{(0)} = [0.3\ 0.6]^T$. Show your work and report your result rounded to 3 significant figures. [6 points]

$$U = \begin{bmatrix} 0.5 & 0.3 \\ 0.4 & 0.2 \end{bmatrix} \quad V = \begin{bmatrix} 0.2 & 0.5 & 0.1 \\ 0.6 & 0.1 & 0.8 \end{bmatrix} \quad W = \begin{bmatrix} 0.4 & 0.2 \\ 0.3 & 0.1 \\ 0.1 & 0.7 \end{bmatrix}$$

(d) Using your $\mathbf{s}^{(1)}$ calculated above, calculate the output vector $\hat{\mathbf{y}}^{(1)}$. Show your work and report your result rounded to 3 significant figures. [6 points]

(e) 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] [8 points]

Question 2: Training Recurrent Neural Networks [37 points]

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) Using your vector $\hat{\mathbf{y}}^{(1)}$ from question 1(d), calculate the loss assuming the desired vector $\mathbf{d}^{(1)} = [0\ 0\ 1]$. Do the same for pairs: ($\hat{\mathbf{y}}^{(2)} = [0.168\ \ 0.229\ \ 0.603]$, $\mathbf{d}^{(2)} = [0\ 0\ 1]$) and ($\hat{\mathbf{y}}^{(3)} = [0.475\ \ 0.317\ \ 0.208]$, $\mathbf{d}^{(3)} = [0\ 1\ 0]$). Show your work and report your result rounded to 3 significant figures. [6 points]

(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. [6 points]

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

Optimizing the loss using back propagation 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 **U**, **V** and **W** for a simple back propagation through the RNN, where we only look back one step in time as described above. [12 points]

So far, we have only looked at and implement simple back propagation (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 back propagation through time (BPTT), which takes into account the previous τ time steps during back propagation. At time t,

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

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) Intuitively, what does this recursive update in BPTT mean? What is the difference to applying simple back propagation? What can be advantages of BPTT, and what can be drawbacks? [5 points]

(e) Implement the method `acc_deltas_bptt` that accumulates the weight updates for **U**, **V** and **W** using back propagation through time for τ time steps. [8 points]

Question 3: Language Modeling [32 points]

By now you should have everything in place to train a full Recurrent Neural Network using back propagation through time. In the following questions, we will use the training and development data provided in `ptb-train.txt` and `ptb-dev.txt`. The training data consists of the first 20 sections of the WSJ corpus of the Penn Treebank, 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 PTB data, and the `__main__` method of the `rnn.py` module provides some starter code for training your models.

(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 back propagation (at least: 0, 2, 5), and learning rate (at least: 0.5, 0.1, 0.05). The method `train` 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. [12 points]

(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 points]

One exciting property of RNNs is that they can be used as *sentence generators* to statistically generate new (unseen) sentences. Since the model effectively has learned a probability distribution over words w_{n+1} for a given sequence $[w_1, \cdots w_n]$, we can generate new sequences by starting with an "empty" sentence beginning with a *start symbol* <s>. We can then apply the RNN forward and sample a new word w_{t+1} from the distribution $\hat{\mathbf{y}}^{(t)}$ at each time step. Then we feed this word in as input at the next step, and repeat until the model emits an end token </s>. At each step, we can also compute the point cross-entropy loss for the currently generated word i as:

$$loss(\hat{\mathbf{y}}^{(t)}) = -log(\hat{y}_i^{(t)})$$

Summing over the losses of generated words, we can calculate the perplexity of the newly generated sequence.

(c) In `rnn.py`, implement the method `generate_sequence` which starting from a start symbol, generates a new sentence by applying forward predictions in the RNN. The method should return the generated sequence, as well as its perplexity. Include 2 or 3 generated sentences in your report, along with their loss and perplexities[9]. [10 points]

[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

[9]Your generated sentences will probably contain a lot of UNKNOWN tokens. This is fine, but if you have time and fun with it, you may try to find a way to filter or replace those.

References

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

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

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"[10].

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