Natural Language Understanding (2015–2016)

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

Assignment 2: Recurrent Neural Networks

The assignment is due 25th March 2016, 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`.
- 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`.

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. Details about this can be found at:
`http://www.ed.ac.uk/schools-departments/academic-services/students/undergraduate/discipline/academic-misconduct`
and at:
`http://www.inf.ed.ac.uk/admin/ITO/DivisionalGuidelinesPlagiarism.html`

Assignment Data The files necessary to complete this assignment are available on the course homepage, as well as on the NLU project space. The project space is accessible (from lab machines) at `/afs/inf.ed.ac.uk/group/project/nlu/`. If you are working on your private machine, make sure you are able to access the data.

Pandas The assignment makes use of the Pandas[1] Python package. Pandas version 0.15.2 is installed on the lab machines and is ready to use. If you are working on a private machine, make sure to have Pandas installed. As we only use it to read in data, other versions than 0.15.2 might work however, we only guarantee correct functionality when working on a lab machine.

Numpy In this assignment, you will implement a Recurrent Neural Network (RNN). As RNNs naturally involve matrices and vectors, we strongly recommend you make use of Numpy[2] for operations on these data structures. Numpy version 1.9.2 is installed on the lab machines and is ready to use. All method arguments and return values that are lists/arrays/matrices should be assumed to be Numpy data structures.

Terminology/definitions In Recurrent Neural Networks, we deal with *matrices* and *vectors* for both forward prediction, and back propagation. As notations for these structures and operations on them can be confusing, we use the following conventions:

(1) *Matrices* are assigned *capital letters*, e.g., U, V, W.

(2) *Vectors* are written in *lower cased* letters, e.g., $s^{(t)}$, $\delta^{(t)}$.

(3) For a matrix M and a vector v, Mv designates the *Matrix-vector product*.

(4) For two vectors v and w of length n, vw designates their *entrywise product*:

$$vw = [v_0 * w_0,\ v_1 * w_1,\ \cdots\ v_n * w_n]$$

Similarly, $v - w$ designates their entrywise substraction, and $v + w$ entrywise addition.

(5) For two vectors v and w, $M = v \otimes w$ designates the *outer product* of v and w.

Question 1: Recurrent Neural Networks [31 points]

In this section, you will implement your first recurrent neural network (RNN) and use it to build a language model.
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 words $x_1, \ldots, x_t$, a language model predicts the following word x_{t+1} by modeling:

$$P(x_{t+1} = v_j \mid x_t, \ldots, x_1)$$

[1] http://pandas.pydata.org/
[2] http://www.numpy.org/

where v_j is a word in the vocabulary.

Your job is to implement a recurrent neural network language model, which uses feedback information in the hidden layer to model the "history" $x_t, x_{t-1}, \ldots, x_1$. Formally, the model[3] has to calculate, for $t = 1, \ldots, n-1$:

$$
\begin{aligned}
s^{(t)} &= \text{sigmoid}\left(net_{in}^{(t)}\right) && (1)\\
net_{in}^{(t)} &= Vx^{(t)} + Us^{(t-1)} && (2)\\
\hat{y}^{(t)} &= \text{softmax}\left(net_{out}^{(t)}\right) && (3)\\
net_{out}^{(t)} &= Ws^{(t)} && (4)
\end{aligned}
$$

where $x^{(t)}$ is the one-hot vector representing the index of the current word, $s^{(t)}$ and $\hat{y}^{(t)}$ are the corresponding hidden and generated output layers, and $net_{in}^{(t)}$, $net_{out}^{(t)}$ are the activations for the hidden and output layers.
For a given input $[x^{(1)}, \ldots, x^{(t)}]$, the probability of the next word at time index $t+1$ can then be read from the output vector $\hat{y}^{(t)}$:

$$
\bar{P}(x^{(t+1)} = j \mid x^{(t)}, \ldots, x^{(1)}) = \hat{y}_j^{(t)} \quad (5)
$$

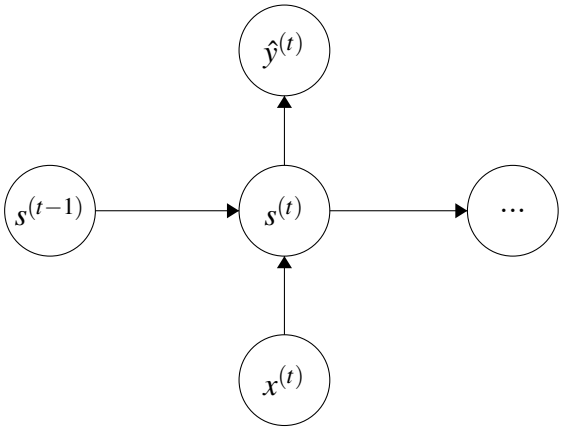
The functions *sigmoid* and $softmax$ are the *activation functions* for the hidden layer and the output layer of the RNN respectively, and the parameters to be learned are:

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

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

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

[3] This model is adapted from a paper by Tomas Mikolov, et al. from 2010: `http://www.fit.vutbr.cz/research/groups/speech/publi/2010/mikolov_interspeech2010_IS100722.pdf`



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

(c) Given the matrices U, V and W below, calculate the hidden layer $s^{(1)}$, for the given 1-hot input vector $x^{(1)} = [0\ 1\ 0]$. Assume that the hidden layer at time $t = 0$ is the vector $s^{(0)} = [0.3\ 0.6]$. 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 $s^{(1)}$ calculated above, calculate the output vector $\hat{y}^{(1)}$. 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 list of word indices $[w_1, \cdots, w_n]$, where w_t stands in for the 1-hot vector $[x_1^{(t)}, \cdots, x_i^{(t)}]$ with the element at index $j = w_t$ set to 1. The return values are the corresponding matrices of hidden layers s and output layers $\hat{y}$. [8 points]

Question 2: Training Recurrent Neural Networks [37 points]

When training RNNs, we need to propagate the errors observed at the output layer $\hat{y}$ back through the network, and adjust the weight matrices U, V and W to minimize the observed loss w.r.t. a desired output d. 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 $d^{(t)}$ is the one-hot vector corresponding to the desired output word at time t. This is a point-wise loss, and we sum (or average) the cross-entropy loss across all examples in a sequence, across all sequences in the dataset in order to evaluate model performance.

(a) Using your vector $\hat{y}^{(1)}$ from question 1(d), calculate the loss assuming the desired vector $d^{(1)} = [0\ 0\ 1]$, and the subsequent pairs ($\hat{y}^{(2)} = [0.168\ 0.229\ 0.603]$, $d^{(2)} = [0\ 0\ 1]$) and ($\hat{y}^{(3)} = [0.475\ 0.317\ 0.208]$, $d^{(3)} = [0\ 1\ 0]$) [6 points]

(b) In `rnn.py`, implement the methods `compute_loss` and `compute_mean_loss`. The first method should, for a given pair x and d of input/desired output words, return the loss of the predicted words $\hat{y}$ when applying the RNN. The `compute_mean_loss` should compute the average loss over a corpus of inputs. [6 points]

Optimizing the loss in our back propagation means we have to calculate the update values Δ w.r.t. the gradients of our loss function for the the observed errors. For the output layer weights, we update matrix W with:

$$\Delta W = \eta \sum_{p}^{n} \delta_{out,p} \otimes s_p \tag{8}$$

$$\delta_{out,p} = (d_p - \hat{y}_p) g'(net_{out,p}) \tag{9}$$

We then further propagate the error observed at the output back to V with:

$$\Delta V = \eta \sum_{p}^{n} \delta_{in,p} \otimes x_p \tag{10}$$

$$\delta_{in,p} = W^T \delta_{out,p} f'(net_{in,p}) \tag{11}$$

In case of cross-entropy loss using *sigmoid* and $softmax$, the derivatives are given as[4]:

$$g'(net_{out,p}) = \vec{1} \tag{12}$$

$$f'(net_{in,p}) = s_p(\vec{1} - s_p) \tag{13}$$

Finally, when only going back one step in time, we can update the recurrent weights with:

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

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

(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. [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 ΔU and ΔV. 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, the updates ΔW can be derived as before. For ΔU and ΔV, we additionally recursively update at times $(t-1)$, $(t-2)$ $\cdots(t-\tau)$:

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

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

$$\delta_{in,p}^{(t-\tau)} = U^{T} \delta_{in,p}^{(t)} f'(net_{in,p}^{(t-\tau)}) \tag{17}$$

(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) In `rnn.py`, 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 Modelling

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: 10, 50, 100), the look-back in back propagation (at least: 0, 3, 10), 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[5].Report your findings during parameter estimation. [12 points]

(b) Using your best parameter settings found in a), train an RNN on the full training set, again tuning on the first 1000 development sentences. When your model is trained, evaluate it on the full dev set and report the mean loss, as well as both the adjusted and unadjusted perplexity your model achieves[6]. 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 $x^{(t+1)}$ from the distribution $\hat{y}^{(t)}$ 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

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

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 cross-entropy loss. Include 2 or 3 generated sentences in your report, along with their loss and perplexities[7]. [10 points]

[5]Note that training models might take some time. For example, running one iteration over 1000 sentences and calculating loss on 1000 sentences takes my network about 90 seconds when using a vocabulary of 2000 words, 100 hidden units, and 3 steps of back propagation. Training for 9 to 10 iterations may take 15 minutes.

[6]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

[7]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"[8].

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