

# Assignment #2

**Due date: 15/05 11:59 PM**

This handout consists of several homework problems, as well as instructions on the “deliverables” associated with the coding portions of this assignment.

These questions require thought, but do not require long answers. Please be as concise as possible.

Not all questions will be checked.

**Note on Logistics:** In response to feedback from the first assignment, Assignment 2 relies much less on notebook grading. Instead, I’ve provided a list of “deliverables” for the programming sections. Some of these are short-answer questions; put these in your written portion. For the rest, I give you code snippets that create the correct output; just run the corresponding cells, and include the files they create in your submission .zip file.

## Setup

***Note:** Please be sure you have Python 2.7.x installed on your system, and follow the setup instructions below.*

**Get the starter code from the course website**

**Python package requirements:** We'll be using IPython (Jupyter) notebooks in this assignment just like the last one. If you are able to open those, then the notebooks for this assignment should work fine. You'll also need the following packages, which may already be installed:

- numpy 1.9.2
- matplotlib 1.4.3
- scikit-learn (sklearn) 0.16.0
- pandas 0.15.2

If you had trouble setting up IPython or other dependencies for the last assignment, or if you are using Windows, I recommend installing the [Anaconda](#) Python distribution (use the Python 2.7 version). It includes a package manager `conda` that greatly simplifies installing and linking most common Python packages.

Additionally, if you have an Intel CPU you can speed up your models by using Intel MKL - this will automatically parallelize many matrix and vector operations in NumPy. The easiest way to get this is to install a compatible distributions:

[Anaconda](#) and [Enthought Canopy](#) both include this in their full versions, which are free for academic use.

**If you use Anaconda:** just run and you should be good to go:

```
conda update conda
conda update anaconda
conda update scikit-learn
```

## Submitting your work

This assignment requires you to submit a number of files, which are listed in the assignment handout under "**deliverables**" for each section. Please be sure to use the provided code snippets in the notebook to generate these files, and follow the instructions below to submit a .zip file and your writeup PDF.

## Submission Logistics

There are four steps to submit your assignment, so *please read the following carefully*:

1. Copy Part 1.1 code to [part1lprobing.py](#)
2. Run the sanity checker [collectSubmission.py](#)
3. Upload [<your-sunet-id>.zip](#) and your writeup PDF

**Step 1:** In order to evaluate Part 1.1 (Probing Neuron Responses), I ask that you copy-and-paste your code - minus the print statements - for 1.1(a), 1.1(b), and 1.1(c) from the [part1-NER.ipynb](#) notebook into the respective functions [part\\_a\(\)](#), [part\\_b\(\)](#), and [part\\_c\(\)](#) in the template file [part1lprobing.py](#). Put this file in your assignment directory.

Unless you made significant modifications to the starter code (*not recommended*), this should only take a couple minutes. You can test your code by running, in the notebook:

```
from part1lprobing import part_a, part_b, part_c
# clf is your trained WindowMLP model
part_a(clf, num_to_word)
part_b(clf, num_to_word, num_to_tag)
part_c(clf, num_to_word, num_to_tag)
```

or alternatively by running `python part1probing.py` from the command line to test your code on a nonsense model.

**Step 2:** I've written a sanity checker script that will run a few *very basic* tests to make sure your output and code are in the correct format. Please note that this script *does not* verify correctness or guarantee that all output is in the correct format. *It only checks for a few cases of malformed output and does not run gradient checks!* The script will also check for all the files you need to submit, and will prepare a .zip file for submission. This script should work on all platforms (Mac, Linux, Windows).

Run:

```
python collectSubmission.py
```

The script will warn you if files are missing or tests fail; you may opt to continue and it will still prepare the .zip, but you do so at your own risk.

**Step 3:** Upload `<your-student-id>.zip`, as output by the sanity checker, to webcourse.

## Assignment Overview (Tasks)

There are three parts to this assignment, each of which has both written and programming components. As with Assignment #1, the programming components are centered around IPython notebooks. To reduce clutter and scrolling, I've given you three separate notebooks, `part0-XOR`, `part1-NER`, and `part2-RNNLM`, corresponding to each part of the assignment. You'll also be writing code in standalone python files `misc.py`, `nerwindow.py`, and `rnnlm.py` to implement your models.

Be sure to follow the instructions carefully, in the handout, notebook, and comments that define the functions you'll be implementing. As with Assignment #1, I strongly recommend that you complete the written component for each section before you beginning the accompanying programming section.

## 0 Warmup: Boolean Logic

A famous result from the early days of AI (see [https://en.wikipedia.org/wiki/Perceptrons\\_\(book\)](https://en.wikipedia.org/wiki/Perceptrons_(book))) proved that a linear classifier, such as a single layer of a neural network, cannot compute even simple functions such as binary XOR. We will show by example that a two-layer neural network, however, is perfectly capable of learning this pattern.

- (a) Let  $x \in \{0, 1\}$  and  $y \in \{0, 1\}$ . Express the XOR ( $\oplus$ ) function as the composition of other boolean logic operations (NOT, AND, OR, implication, etc.). Argue graphically (i.e. plot or sketch  $(x, y, \text{class} = f(x, y))$  for  $x \in \{0, 1\}$  and  $y \in \{0, 1\}$ ) that the operations you use in your composition are possible to compute with an ordinary (single-layer) linear classifier.
- (b) Assume that each neuron has a step-function activation, i.e  $h_i(x, y) = \theta(w_{i1}x + w_{i2}y + b_i)$ , where:

$$\theta(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

Find weights that allow  $h_i(x, y)$  to compute the single-layer functions from (a). Your classifier can have real-valued weights; only the data need be binary. (Hint: which operation is given by  $\theta(x + y - 0.5)$ ?)

- (c) Open the IPython notebook `part0-XOR.ipynb` and follow the instructions, using what you derived in (a) and (b) to implement a very simple 2-layer network that can compute (i.e. classify) binary XOR.

## 1 Deep Networks for Named Entity Recognition

In this section, we'll get to practice backpropagation and training deep networks to attack the task of Named Entity Recognition: predicting whether a given word, in context, represents one of four categories:

- Person (PER)
- Organization (ORG)
- Location (LOC)
- Miscellaneous (MISC)

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).

The model is a 2-layer neural network, with an additional representation layer similar to what you saw with word2vec. Rather than averaging or sampling, here we explicitly represent context as a “window” consisting of a word concatenated with its immediate neighbors:

$$x^{(t)} = [Lx_{t-1}, Lx_t, Lx_{t+1}] \in \mathbb{R}^{3d} \quad (1)$$

where the input  $x_{t-1}, x_t, x_{t+1}$  are one-hot vectors (really, just indices) into a word-representation matrix  $L \in \mathbb{R}^{d \times |V| - 1}$ , with each column  $L_i$  as the vector for a particular word  $i = x_t$ . We then compute our prediction as:

$$h = \tanh(Wx^{(t)} + b_1) \quad (2)$$

$$\hat{y} = \text{softmax}(Uh + b_2) \quad (3)$$

And evaluate by cross-entropy loss

$$J(\theta) = - \sum_{k=1}^5 y_k \log \hat{y}_k \quad (4)$$

where  $y \in \mathbb{R}^5$  is a one-hot label vector. To compute the loss for the training set, we sum (or average) this  $J(\theta)$  as computed with respect to each training example.

For this problem, we let  $d = 50$  be the length of our word vectors, which are concatenated into a window of width  $3 \times 50 = 150$ . The hidden layer has a dimension of 100, and the output layer  $\hat{y}$  has a dimension of 5.

a) Compute the gradients of  $J(\theta)$  with respect to all the model parameters:

$$\frac{\partial J}{\partial U} \quad \frac{\partial J}{\partial b_2} \quad \frac{\partial J}{\partial W} \quad \frac{\partial J}{\partial b_1} \quad \frac{\partial J}{\partial L_i}$$

$$U \in \mathbb{R}^{5 \times 100} \quad b_2 \in \mathbb{R}^5 \quad W \in \mathbb{R}^{100 \times 150} \quad b_1 \in \mathbb{R}^{100} \quad L_i \in \mathbb{R}^{50}$$

---

<sup>1</sup> In the code, we'll implement this with word vectors as rows for efficiency, and so you can access  $L_i$  as  $L[i]$ . See `nerwindow.py` for more details

In the spirit of backpropagation, you should express the derivative of activation functions (tanh, softmax) in terms of their function values (as with sigmoid in Assignment 1).

This identity may be helpful:

$$\tanh(z) = 2 \operatorname{sigmoid}(2z) - 1$$

Furthermore, you should express the gradients by using an “error vector” propagated back to each layer; this just amounts to putting parentheses around factors in the chain rule, and will greatly simplify your analysis. All resulting gradients should have simple, closed-form expressions in terms of matrix operations. (Hint: you’ve already done most of the work here as part of Assignment 1.)

- b) To avoid parameters from exploding or becoming highly correlated, it is helpful to augment our cost function with a Gaussian prior: this tends to push parameter weights closer to zero, without constraining their direction, and often leads to classifiers with better generalization ability. If we maximize log-likelihood (as with the cross-entropy loss, above), then the Gaussian prior becomes a quadratic term <sup>2</sup> (L2 regularization):

$$J_{reg}(\theta) = \frac{\lambda}{2} \left[ \sum_{i,j} W_{ij}^2 + \sum_{i',j'} U_{i'j'}^2 \right] \quad (5)$$

and we optimize the combined loss function

$$J_{full}(\theta) = J(\theta) + J_{reg}(\theta) \quad (6)$$

Update your gradients from part (a) to include the additional term in this loss function (i.e. compute  $\frac{dJ_{full}}{dW}$  etc.)

- c) In order to avoid neurons becoming too correlated and ending up in poor local minima, it is often helpful to randomly initialize parameters. Empirically, the following has been found to work well:

---

<sup>2</sup> **Optional (not graded):** The interested reader should prove that this is indeed the maximum-likelihood objective when we let  $W_{ij} \sim N(0, 1/\lambda)$  for all  $i, j$ .

Given a matrix of  $A$  of dimension  $m \times n$ , select values  $A_{ij}$  uniformly from  $[-\epsilon, \epsilon]$ , where

$$\epsilon = \frac{\sqrt{6}}{\sqrt{m+n}} \quad (7)$$

Implement the function `random_weight_matrix(m,n)` in `misc.py` to perform this initialization. A cell is provided to test this code in `part1-NER.ipynb`.

- d) Open the notebook `part1-NER.ipynb` and follow the instructions to implement the NER window model, using the gradients you derived in (a) and (b). You'll also want to take a look at the example classifier in `softmax_example.py` for a guide on how to implement your model using our starter code.

**Deliverables:**

- a. Working implementation of the NER window model, in `nerwindow.py`. (We'll look at, and possibly run this code for grading.)
  - b. In your writeup (i.e. where you're writing the answers to the written problems), briefly state the optimal hyperparameters you found for your model: regularization, dimensions, learning rate (including time-varying, such as annealing), SGD batch size, etc.
- e) In the notebook, follow the instructions to plot learning curves for your best model, and for a comparison of learning rates.

**Deliverables:**

- a. Plot of the learning curve for your best model, in `ner.learningcurve.best.png`.
  - b. Plot comparing  $\alpha = 0.01$  to  $\alpha = 0.1$  in `ner.learningcurve.comparison.png`.
- f) In the notebook, follow the instructions to evaluate your model's performance on the dev set, and compute predictions on the test data. Note that the test set has only dummy labels; I'll compare your predictions against the ground truth after you submit.

Note that you should compute F1 scores by a weighted average across all classes except "O", since this null class is not of interest for practical applications. The function `eval_performance()` in `nerwindow.py` will do this for you.

**Deliverables:**

- a. Report, in your writeup, the performance of your model on the dev set (as output by `eval_performance()`).
- b. List of predicted labels for the test set, one per line, in the file `test.predicted`.

## 1.1 Deep Networks: Probing Neuron Responses

Still in the part1-NER.ipynb notebook, follow the instructions to “probe” the responses of the hidden and output neurons in your network. You should report the following in your writeup:

- a) Top-10 word lists for the center word, on 5 hidden layer neurons of your choice.
- b) Top-10 word lists for the center word, on model output for PER, ORG, LOC, and MISC.
- c) Top-10 word lists for the first word (preceding the center word), on model output for PER, ORG, LOC, and MISC.

For each, give a brief (no more than 2 sentence) comment on what the model appears to learn.

## 2 Recurrent Neural Networks: Language Modeling

In this section, you’ll implement your first recurrent neural network (RNN<sup>3</sup>) 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  $x_1, \dots, x_t$ , a language model predicts the following word  $x_{t+1}$  by modeling:  $P(x_{t+1}=v_j | x_1, \dots, x_t)$  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”  $x_t, x_{t-1}, \dots, x_1$ . Formally, the model<sup>4</sup> is, for  $t=1, \dots, n-1$ :

---

<sup>3</sup> We’ll start talking about recursive neural networks soon and also call these RNNs - but it turns out that recurrent nets are just a special case of recursive nets, so there’s actually nothing ambiguous!

<sup>4</sup> This model is adapted from a paper by Toma Mikolov, et al. from 2010:  
[http://www.fit.vutbr.cz/research/groups/speech/publi/2010/mikolov\\_interspeech2010\\_IS100722.pdf](http://www.fit.vutbr.cz/research/groups/speech/publi/2010/mikolov_interspeech2010_IS100722.pdf)



$$h^{(t)} = \text{sigmoid} \left( Hh^{(t-1)} + Lx^{(t)} \right) \quad (8)$$

$$\hat{y}^{(t)} = \text{softmax} \left( Uh^{(t)} \right) \quad (9)$$

$$\bar{P}(x_{t+1} = v_j \mid x_t, \dots, x_1) = \hat{y}_j^{(t)} \quad (10)$$

where  $h^{(0)} = h_0 \in \mathbb{R}^{D_h}$  is some initialization vector for the hidden layer and  $Lx^{(t)}$  is the product of  $L$  with the one-hot vector  $x^{(t)}$  representing index of the current word<sup>5</sup>. The parameters are:

$$H \in \mathbb{R}^{D_h \times D_h} \quad L \in \mathbb{R}^{D_h \times |V|} \quad U \in \mathbb{R}^{|V| \times D_h} \quad (11)$$

where  $L$  is the input word representation matrix and  $U$  is the output word representation matrix, and  $D_h$  is the dimension of the hidden layer.

The output vector  $\hat{y}^{(t)} \in \mathbb{R}^{|V|}$  is a probability distribution over the vocabulary, and we optimize the (un-regularized) cross-entropy loss:

$$J^{(t)}(\theta) = - \sum_{j=1}^{|V|} y_j^{(t)} \log \hat{y}_j^{(t)} \quad (12)$$

where  $y^{(t)}$  is the one-hot vector corresponding to the target word (which here is equal to  $x_{t+1}$ ). As in Part 1, this is a point-wise loss, and we sum (or average) the cross-entropy loss across all examples in a sequence, across all sequences<sup>6</sup> in the dataset in order to evaluate model performance.

- (a) Conventionally, when reporting performance of a language model, we evaluate on perplexity, which is defined as:

$$\text{PP}^{(t)}(\hat{y}^{(t)}, y^{(t)}) = \frac{1}{\bar{P}(x_{t+1}^{\text{pred}} = x_{t+1} \mid x_t, \dots, x_1)} = \frac{1}{\sum_{j=1}^{|V|} y_j^{(t)} \cdot \hat{y}_j^{(t)}} \quad (13)$$

i.e. the inverse probability of the correct word, according to the model

---

<sup>5</sup> As in Part 1, in the code it is more convenient to represent  $L$  as a "tall" matrix and access rows as  $L[x[t]]$ .

<sup>6</sup> We implement this for you in `compute_mean_loss` in `rnnlm.py`.

distribution  $\bar{P}$ . Show how you can derive perplexity from the cross-entropy loss (Hint: remember that  $y^{(t)}$  is one-hot!), and thus argue that minimizing the (arithmetic) mean cross-entropy loss will also minimize the (geometric) mean perplexity across the training set. **This should be a very short problem - not too perplexing!**

For a vocabulary of  $|V|$  words, what would you expect perplexity to be if your model predictions were completely random?  
 Compute the corresponding cross-entropy loss for  $|V| = 2000$  and  $|V| = 10000$ , and keep this in mind as a baseline.

- (b) As you did in part 1, compute the gradients with for all the model parameters at a single point in time  $t$ :

$$\left. \frac{\partial J^{(t)}}{\partial U} \quad \frac{\partial J^{(t)}}{\partial L_{x^{(t)}}} \quad \frac{\partial J^{(t)}}{\partial H} \right|_{(t)}$$

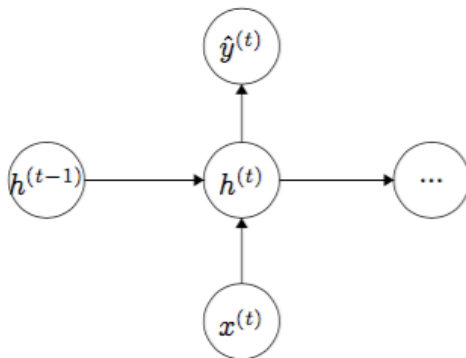
where  $L_{x^{(t)}}$  is the column of  $L$  corresponding to the current word  $x^{(t)}$ , and  $|_{(t)}$  denotes the gradient for the appearance of that parameter at time  $t$ .

(Equivalently,  $h^{(t-1)}$  is taken to be fixed, and you need not backpropagate to earlier timesteps just yet - you'll do that in part (c)).

Additionally, compute the derivative with respect to the previous hidden layer value:

$$\frac{\partial J^{(t)}}{\partial h^{(t-1)}}$$

- (c) Below is a sketch of the network at a single timestep:



Draw the “unrolled” network for 3 timesteps, and compute the backpropagation-through-time gradients:

$$\frac{\partial J^{(t)}}{\partial L_{x^{(t-1)}}} \quad \frac{\partial J^{(t)}}{\partial H} \Big|_{(t-1)}$$

where  $\Big|_{(t-1)}$  denotes the gradient for the appearance of that parameter at time  $(t - 1)$ . Because parameters are used multiple times in feed-forward computation, we need to compute the gradient for each time they appear.

You should use the backpropagation rules from Lecture 5 to express these derivatives in terms of an error term  $\delta^{(t)}$ , such that you can re-use expressions for  $t - 2$ ,  $t - 3$ , and so on.

Note that the true gradient with respect to a training example requires us to run backpropagation all the way back to  $t = 0$ . In practice, however, we generally truncate this and only backpropagate for a fixed number  $\tau \approx 3 - 5$  timesteps.

- (d) Given  $h^{(t-1)}$ , how many operations are required to perform one step of forward propagation to compute  $J^{(t)}(\theta)$ ? How about backpropagation for a single step in time? For  $\tau$  steps in time? Express your answer in big-O notation in terms of the dimensions  $D_h$  and  $|V|$  of the matrices  $L$ ,  $H$ , and  $U$  (Equation 11). What is the slow step?

**Bonus:** Given your knowledge of similar models (i.e. word2vec), suggest a way to speed up this part of the computation. Your approach can be an approximation, but you should argue why it's a good one. The paper “Extensions of recurrent neural network language model” (Mikolov, et al. 2013) may be of interest here.

- (e) Implement the above model in `rnnlm.py`. You'll need to implement just three functions, for now:

- `init ()` (not much to do here)
- `acc_grads()`
- `compute seq loss()`

Data loaders and other starter code is provided in the `part2-RNNLM.ipynb` notebook, and you should use this to verify your implementation.

*Be sure to read the instructions carefully in the starter code!* They describe the data format and how to run your model over a sequence. Particularly,

you should sum the pointwise costs  $J^{(t)}(\theta)$  over a sequence. When accumulating gradients, you should also add up all the gradients you compute for  $J^{(t)}(\theta)$  for each target word in the sequence. (This is basically minibatch SGD.)

- (f) Train a model on the ptb-train data, consisting of the first 20 sections of the WSJ corpus of the Penn Treebank. For speed, we recommend using a small vocabulary of 2000-5000 words.

As in Part1, you should tune your model to maximize generalization performance (minimize cross-entropy loss) on the dev set. I'll evaluate your model on an unseen, but similar set of sentences.

**Deliverables:**

- a. In your writeup, include the best hyperparameters you used (training schedule, number of iterations, learning rate, backprop timesteps), and your perplexity score on the ptb-dev set.
  - b. Model parameters saved as rnnlm.U.npy, rnnlm.H.npy, and rnnlm.L.npy; I'll use these to test your model.
- (g) The networks that you've seen in Assignment 1 and in Part1 of this assignment are discriminative models: they take data, and make a prediction. The RNNLM model you've just implemented is a generative model, in that it actually models the distribution of the data sequence  $x_1, \dots, x_n$ . This means that not only can we use it to evaluate the likelihood of a sentence, but we can actually use it to generate one!

In rnnlm.py, implement the generate sequence() function. This should run the RNN forward in time, beginning with the index for the start token <s>, and sampling a new word  $x_{t+1}$  from the distribution  $\hat{y}^{(t)}$  at each timestep. Then feed this word in as input at the next step, and repeat until the model emits an end token (index of </s>).

***Note:** this should not require a lot of coding - my solution is less than 15 lines, and most of this is copied from compute\_seq\_loss().*

**Deliverables:**

- Include 2-3 generated sentences in your writeup. See if you can generate something humorous!
- Your code, in rnnlm.py; I'll run this to test using the parameters from (f).

- Bonus: Implement the unigram-filling described in the notebook.
- Bonus2: If you want to experiment further with language models, you're welcome to load up your own texts and train on them - sometimes the results can be quite entertaining! (See <http://kingjamesprogramming.tumblr.com/> for a great one<sup>7</sup> trained on a mix of the King James Bible and the Structure and Interpretation of Computer Programs.)

---

<sup>7</sup> This one just uses a simple n-gram Markov model, but there's no reason an RNNLM can't compete!