Neural Networks for Named Entity Recognition*

Programming Assignment #4

Due: Friday, December 4th 2015 at 11:59pm

IMPORTANT NOTE #1: You may complete this assignment individually or in pairs. *We strongly encourage collaboration*. Your submission must include a statement describing the contributions of each collaborator. See the collaboration policy on the course website: `http://cs224n.stanford.edu/grading.html`.

IMPORTANT NOTE #2: Please read this assignment soon and make sure that you are able to access the relevant files and compile the code. You would have to understand the task, the data, and neural networks. Start working early so that you will have ample time to discover stumbling blocks and ask questions.

1 Overview

In this exercise, you will implement a neural network for named entity recognition (NER). Our task is a simplified version of the CoNLL 2003 shared task where many systems were benchmarked. [1] You should review the lecture notes on both NER and neural networks before you start this assignment.

2 Setup

We've put everything for this assignment in the directory `/afs/ir/class/cs224n/cs224n-pa/pa4/`. The starter code can be found in `/afs/ir/class/cs224n/cs224n-pa/pa4/`. Copy over the starter code to your local directory and make sure you can compile it without errors:

```
cd
mkdir -p cs224n/pa4
cd cs224n/pa4
cp -r /afs/ir/class/cs224n/cs224n-pa/pa4 .
cd java
ant
```

2.1 Code

The (very limited) starter code includes the following java files ($\star$ indicates files you will need to complete):

- `Datum.java` - A class to store the words and their labels. No need to change anything in here.
- [$\star$] `FeatureFactory.java` - A class that reads in the train and test data. You want to modify at least one function in here, that reads in the vectors and vocabulary (the set of words we will use).
- `GradientCheck.java` - A helper class to check that your gradients are correct. No modification needed.
- [$\star$] `NER.java` - Main function to start with and to run.
- `ObjectiveFunction.java` An interface for the neural network's objective function. An implementation of this class should be used to do the gradient check.
- [$\star$] `WindowModel.java` - This is where you might want to implement training and testing the neural network model.

You are free to use another setup if that helps you.

*Revision: November 16, 2015

[1] `http://www.cnts.ua.ac.be/conll2003/ner/` simplified to not use BIO but just the labels

2.2 Data

The data files are found at `/afs/ir/class/cs224n/data/pa4`. You should inspect it with any text editor to gain an understanding of the task:

```
-DOCSTART- O

EU ORG
rejects O
German MISC
call O
to O
boycott O
British MISC
lamb O
. O

Peter PER
Blackburn PER
...
```

The data consists of sentences with one token per line where an empty line marks the sentence boundaries. `-DOCSTART-` marks the boundary between different documents. Each word is associated with 5 possible labels: {`O, LOC, MISC, ORG, PER`} representing "not any named entity", location, miscellaneous, organization, person respectively. [2] There are 3 main files in the data folder, all in the same format:

- `train` - The labeled training data that you use to fit your model
- `dev` - The labeled dev set you will try to push performance on by tuning some hyperparameters and trying different things

2.3 Evaluation

A Perl script to evaluate the performance of your predictions is provided: `/afs/ir/class/cs224n/cs224n-pa/pa4/conlleval`. You program should provide some very simple outputs in the format shown in `example.out` where the columns are the token, the gold label, and your predicted label from left to right. Inspect `example.out` and run

```
./conlleval -r -d '\t' < example.out
```

TODO: Implement the very simple baseline of exact string match with unambiguous named entity in the training data. This should give about 60% F1.

2.4 Word vectors

Also in the data directory `/afs/ir/class/cs224n/data/pa4`, we provide an initial set of word vectors that have been trained with an unsupervised method.[1] You will train these vectors further as a part of your task. The vectors are $n = 50$ dimensional.

- `vocab.txt` - List of words in the vocabulary, one per line, uncased, in the same order as the word vectors file.
- `wordVectors.txt:` - Word vectors corresponding to the words in vocab.txt. Again, one vector per row. Each row should contain 50 numbers since our word vectors are 50-dimensional.

3 The Neural Network

In class, we described a feedforward neural network and how to use and train it for named entity recognition with multiple classes. You will implement the word embedding layer, the feedforward neural network and the

[2]More details: `http://www.cnts.ua.ac.be/conll2003/ner/annotation.txt`

corresponding backpropagation training algorithm. Furthermore, you will learn how to make sure your neural network code is bug-free by checking your gradients. As with most machine learning method, we will analyze our errors and tune various model parameters.

3.1 Model overview

An illustration of the model is shown in Figure 1.

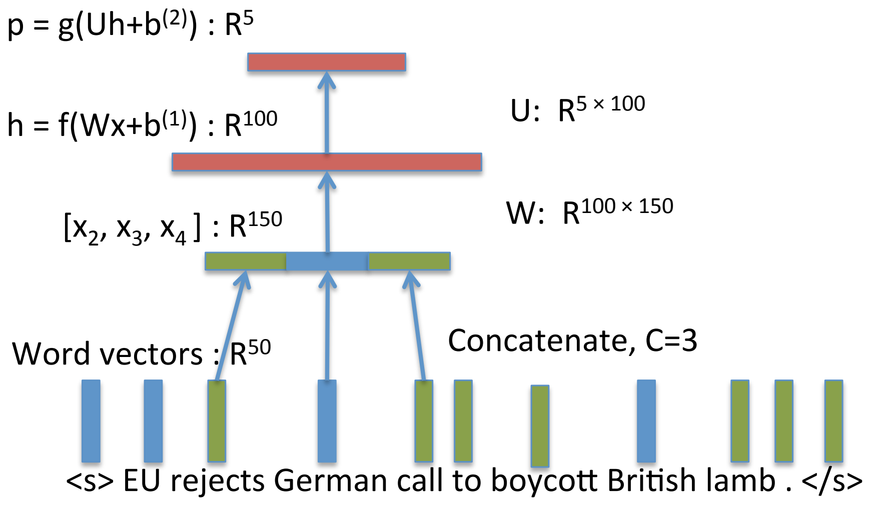

Figure 1: Figure showing forward propagation on the neural network for one training window. Assume a context window that takes one word from each side

This neural network has one single hidden layer. We will go into the structure of the network in more detail in the following sections.

3.2 Input Layer: Word vectors and context windows

The idea of the model is that in order to classify each word, you take as input that word's vector representation and the vector representations of the words in its context.

Word Vectors

Assume you have the following sequence of words in your training set:
`George is happy about the election outcome.` Each word is associated with an index into a vocabulary, e.g., `George` might have index 303.

All word vectors are n-dimensional and saved in a large matrix $L \in \mathbb{R}^{n \times V}$, where V is the size of the vocabulary. This means that each word is mapped to a point in an n-dimensional space.

TODO: You will have to load these vectors into your Java program and save them into a matrix along with their indices and the words they represent. Later, you will compare these word vectors with randomly initialized word vectors.

Context Windows

Assume that at the first location of the sentence we have vector x_1, which was retrieved via the index of the word `George`. Similarly, we can represent the whole sentence as a list of vectors $(x_1, x_2, \ldots, x_8)$. When we want to classify the first and last word and include their context, we will need special beginning and end tokens, which we define as `<s>` and `</s>` respectively. If we use a context size of C, we will have to pad each input of the training accordingly. For example, if we have $C = 3$ and we define the vector corresponding to the begin padding as x_s and for the end padding as x_e. Hence, we will get the following sequence of vectors:

$$(x_s, x_1, x_2, \ldots, x_8, x_e).$$

For this training corpus, we have the following windows that we will give as input to the neural network:

$$\{[x_s, x_1, x_2], [x_1, x_2, x_3], [x_2, x_3, x_4], \ldots, [x_6, x_7, x_8], [x_7, x_8, x_e]\}$$

Each window is associated with the label of the center word. So for this sequence, we get the labels $\{y_1, y_1, \ldots, y_8\} \in \{\texttt{O, LOC, MISC, ORG, PER}\}^8$. Note that the model has no knowledge of the underlying sentence and just sees each window as a separate training instance.

Figure 1 shows one such window for word vector x_i being fed into the neural network.

3.3 Hidden and Output Layers

The neural network transforms the input word vectors into a "hidden" vector, of dimensionality H. The final output layer is a **logistic regression (a.k.a. MaxEnt)** classifier which uses the "hidden" vector to predict whether or not the input belongs to a certain class. The intuition behind the hidden layer is that each node learns some representation of the input that helps the classifier come to a decision. There can be multiple hidden layers too, with each layer's output feeding on to the next one. The dimensionality of the hidden layer as well as the number of hidden layers is a parameter of the model and you will get to play around with these.

So how does the neural network compute the "hidden" vector and the output p_i? Read on.

3.4 Definition of feedforward network function

The feedforward operation of the neural network follows the following set of equations:

$$\begin{aligned} z &= W \begin{bmatrix} x_{i-1} \\ x_i \\ x_{i+1} \end{bmatrix} + b^{(1)} && (1) \\ h &= f(z) && (2) \\ p &= g(Uh + b^{(2)}). && (3) \end{aligned}$$

The final prediction $p \in \mathbb{R}^K$ is x_i's probability of being in each of the K classes ($K = 5$ in this assignment). Let us define all the involved notation: the model parameters W, U and the model functions f, g.

Weight Matrix $W \in \mathbb{R}^{H \times Cn}$, $b^{(1)} \in \mathbb{R}^H$: The matrix W and bias vector $b^{(1)}$ represent the linear transformation from the input layer to the hidden layer. In our example with a window size of $C = 3$, word vector dimension $n = 50$ and if we have a hidden layer size such as $H = 100$, we get $W \in \mathbb{R}^{100 \times 150}$. The bias of the hidden layer is $b^{(1)} \in \mathbb{R}^{H \times 1}$. The bias terms corresponds to the intercept in the linear transformation. The vector z is the result of applying this linear transformation

Nonlinear Function f: Each hidden layer neuron applies a non-linear transformation to its input z. The function f can be either the $\mathrm{sigmoid}$ function, which has range [0,1] or the hyperbolic $\tanh$ function, which has range [-1,1]. This function can be thought of as an activation function that determines whether or not a neuron is "firing". For example, a sigmoid output that is close to 1 can be interpreted as the neuron being active. For your default choice, please use $\tanh$. It might be helpful to express the derivative of $\tanh$ terms of the function value itself:

$$\frac{d}{dx} \tanh x = 1 - \tanh^2 x \tag{4}$$

Weight Matrix $U \in \mathbb{R}^{K \times H}$, Bias $b^{(2)} \in \mathbb{R}^K$, Function g: These parameters perform similar functions as above, but between the hidden layer and the output layer. $U \in \mathbb{R}^{K \times H}$ and the bias of the logistic regression $b^{(2)}$ is a vector of dimension K, the number of classes. Note that you could also add a single 1 to a h and use $U \in \mathbb{R}^{K \times (H+1)}$, in other words including the bias term $b^{(2)}$ inside U. The function $g : \mathbb{R}^K \to \mathbb{R}^K$ needs to return a probability over K classes so we will use the softmax fuction

$$g_i(v) = \mathrm{softmax}(v_i, v) = \frac{\exp(v_i)}{\sum_{k=1}^{K} \exp(v_k)} \tag{5}$$

Now, let us summarize this whole procedure, the network and its inputs by the following notation. The training inputs consist of a set of (window,label) pairs $(x^{(i)}, y^{(i)})$, with i ranging from 1 to the length of your training corpus. Each $x^{(i)} = [x_{i-1}, x_i, x_{i+1}]$ (note the difference in notation between superscript and subscript) and $y^{(i)} \in \{0, 1\}^K$.

We define θ to hold all network parameters: $\theta = (L, W, b^{(1)}, U, b^{(2)})$. Using this notation, we can now define the entire neural network in one function:

$$p_\theta(x^{(i)}) = g\left(Uf\left(W\begin{bmatrix} x_{i-1} \\ x_i \\ x_{i+1} \end{bmatrix} + b^{(1)}\right) + b^{(2)}\right) \tag{6}$$

TODO: Implement this feedforward hypothesis function p_θ.

3.5 Random initialization

When you implement the feedforward function you will notice that you need the parameters of the model. In the beginning you initialize these parameters randomly. One effective strategy for random initialization is to randomly select values for W *uniformly* in the range $[-\epsilon_{init}, \epsilon_{init}]$. One effective strategy for choosing ϵ_{init} is to base it on the number of units feeding into the layer and the number of units of this current layer. A good choice of ϵ_{init} is

$$\epsilon_{init} = \frac{\sqrt{6}}{\sqrt{fanIn + fanOut}}, \tag{7}$$

where $fanIn = nC$ and $fanOut = H$ in our case. This range of values ensures that the parameters are kept small and makes the learning more efficient. You can initialize the biases to zero.

TODO: You need to randomly initialize all the parameters of your model. **Hint:** Use the function `SimpleMatrix.random` from the SimpleMatrix library that we provide with the code.

3.6 Cost function

In logistic regression we want to maximize the log-likelihood of our parameters given all our (say m) data samples:[3]

$$\begin{aligned} \ell(\theta) &= \log \prod_{i=1}^{m} p(y^{(i)}|x^{(i)};\theta) & (8) \\ &= \sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log p_{\theta,k}(x^{(i)}). & (9) \end{aligned}$$

By convention, we will minimize the negation of log-likelihood instead. Therefore, our cost function for the neural network is

$$J(\theta) = -\frac{1}{m}\ell(\theta), \tag{10}$$

This objective is equivalent to minimizing the cross-entropy error, and we will proceed to describe the neural network setup according to the cross-entropy error interpretation.

3.7 Regularized cost function

As discussed in the lecture on MaxEnt classifiers, we should put a Gaussian prior, parameterized by λ, on our parameters to prevent overfitting. The cost function for neural network with regularization is given by

$$J_R(\theta) = J(\theta) + \frac{\lambda}{2m}\left(||W||^2 + ||U||^2\right) \tag{11}$$

Larger values of λ push the weights closer to zero.

Note that you should not be regularizing the terms that correspond to the bias. Notice that you can first compute the unregularized cost function J and then later add the cost for the regularization terms.

[3]refer to the lecture slides on the MaxEnt model or the section on logistic regression from these class notes: `cs229.stanford.edu/notes/cs229-notes1.pdf` for the full model you need to have to use.

4 Backpropagation Training

In this part, you will implement the backpropagation algorithm to compute the gradient for the neural network cost function. Once you have computed the gradient, you will be able to train the neural network by minimizing the cost function $J(\theta)$ using a simple optimization technique called stochastic gradient descent (SGD).

You will first implement the backpropagation algorithm to compute the gradients for the parameters for the (unregularized) neural network. After you have verified that your gradient computation for the unregularized case is correct, you will implement the gradient for the regularized neural network.

Recall that the intuition behind the backpropagation algorithm is as follows. Given a training example $(x^{(i)}, y^{(i)})$, we will first run a "forward pass" to compute all the activations throughout the network, including the output value of the hypothesis $p_\theta(x)$. Once we have the prediction, we will compute the binary cross-entropy error. Then, for each node j in the hidden layer, we would like to compute an "error term" δ_j that measures how much that node was "responsible" for the cross-entropy error.

You need to follow similar steps as in the lecture where we described the derivatives of single elements of the unsupervised word vector learning model and then generalized it to the gradients of the full matrices. The *only difference* is that we now have a logistic regression classifier instead of a linear score. The rest of the model is the same to the one derived in the lecture.

TODO: Derive the following gradients and include **only the final expression** of $\frac{\partial J(\theta)}{\partial L}$ in your report:

$$\frac{\partial J(\theta)}{\partial U}, \frac{\partial J(\theta)}{\partial W}, \frac{\partial J(\theta)}{\partial b}, \frac{\partial J(\theta)}{\partial L} \tag{12}$$

Note that the gradient of L is a short form for taking the derivative of each word vector that appears in each window. So, you find the index of each word in a window (let's call it index v) and then you only update the *column* at that index. In Matlab notation, you can retrieve that index simply by $L(:, v)$.

TODO: Implement these gradients such that you can compute them for any window in your training data.

Practical Tip: First compute $\frac{\partial J(\theta)}{\partial U}$, then proceed backwards through the network for the rest of the derivatives. You might find it easier to pick one particular element u_i from the vector U, calculate $\frac{\partial J(\theta)}{\partial u_i}$, and then generalize the result you obtain into a matrix equation. Using the activation vector a and input vector z as defined in the equations 3 would definitely help. In addition, expressing your result in terms of error vectors δ for each layer would lead you to a nice re-usable expression for gradients at each layer.
As an example, define the error for the output layer as:

$$\delta^{(2)} = p_\theta(x_i) - y_i \tag{13}$$

Now, once you calculate $\frac{\partial J(\theta)}{\partial U}$, try and express the result in terms of $\delta^{(2)}$ - the error term for the output layer and a - the activation vector from the hidden layer. Going backwards in the network, if you carefully define a similar error vector for the hidden layer $\delta^{(1)}$, the expression for $\frac{\partial J(\theta)}{\partial W}$ would have the same form as $\frac{\partial J(\theta)}{\partial U}$

4.1 Gradient checking

You can check whether your implementation is bug-free using gradient checks.

In your neural network, you are minimizing the cost function $J(\theta)$. To perform gradient checking on your parameters, you can imagine "unrolling" the parameters θ into a long vector. Then you can use the following gradient checking procedure.

Suppose you have a function $f_i(\theta)$ that purportedly computes $\frac{\partial}{\partial \theta_i} J(\theta)$ and you would like to check if f_i is outputting correct derivative values.

$$\text{Let} \quad \theta^{(i+)} = \theta + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \epsilon \\ \vdots \\ 0 \end{bmatrix} \qquad \text{and} \quad \theta^{(i-)} = \theta - \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \epsilon \\ \vdots \\ 0 \end{bmatrix} \tag{14}$$

So, $\theta^{(i+)}$ is the same as θ, except its i-th element has been incremented by ϵ. Similarly, $\theta^{(i-)}$ is the corresponding vector with the i-th element decreased by ϵ. You can now numerically verify $f_i(\theta)$'s correctness by checking, for each i, that:

$$f_i(\theta) \approx \frac{J(\theta^{(i+)}) - J(\theta^{(i-)})}{2\epsilon} \tag{15}$$

The degree to which these two values should approximate each other will depend on the details of J. But assuming $\epsilon = 10^{-4}$, you'll usually find that the left- and right-hand sides of the above will agree to at least 4 significant digits (and often many more).

TODO: We provide you with an implementation of this gradient check (`GradientCheck.java`). Use this function to make sure that the gradients are correct. The difference between your gradient and the numerically computed one should be less than 10^{-7}.

Practical Tip: When performing gradient checking, it is much more efficient to use a small neural network with a relatively small number of input units and hidden units, thus having a relatively small number of parameters. Each dimension of θ requires two evaluations of the cost function and this can be expensive. So you can set $H = 2$ for instance. Furthermore, after you are confident that your gradient computations are correct, you should turn off gradient checking before running your learning algorithm.

4.2 Learning parameters with SGD

After you have successfully implemented the neural network cost function and gradient computation, the next step is use SGD to learn a good set parameters.

The idea of SGD is really simple: instead of minimizing the full cost function $J(\theta)$ over the entire dataset, you simply take derivatives over only a single data sample (a single window). Then you update all your parameters by taking one single step in the right direction. The algorithm runs through say 2 iterations of the following procedure:

For all windows $i = 1, \ldots, m$, update parameters by:

$$U^{(t)} = U^{(t-1)} - \alpha \frac{\partial}{\partial U} J_i(U) \tag{16}$$

$$b^{(2)^{(t)}} = b^{(2)^{(t-1)}} - \alpha \frac{\partial}{\partial b} J_i(b^{(2)}) \tag{17}$$

$$W^{(t)} = W^{(t-1)} - \alpha \frac{\partial}{\partial W} J_i(W) \tag{18}$$

$$b^{(1)^{(t)}} = b^{(1)^{(t-1)}} - \alpha \frac{\partial}{\partial b} J_i(b^{(1)}) \tag{19}$$

$$L^{(t)} = L^{(t-1)} - \alpha \frac{\partial}{\partial L} J_i(L), \tag{20}$$

where α is the learning rate, and we define J_i to be the cost of only the ith sample.

Note again, that in each window of size C you only have updates for C word vectors (i.e. columns in L). So, if the C words have indices into the vocabulary $(v_1, v_2, \ldots, v_C)$, then your L update will merely update the columns at these indices!

$$L(:,v_1)^{(t)} = L(:,v_1)^{(t-1)} - \alpha \frac{\partial}{\partial L(:,v_1)} J_i(L(:,v_1)) \tag{21}$$

$$\vdots$$

$$L(:,v_C)^{(t)} = L(:,v_C)^{(t-1)} - \alpha \frac{\partial}{\partial L(:,v_C)} J_i(L(:,v_C)) \tag{22}$$

After the training completes you will proceed to report the training, development and testing accuracy of your classifier by computing the F1 score using the provided evaluation script.

TODO: Implement SGD training and report your initial training, development, and testing F1 scores. Compare randomly initialized L vs. L initialized with the provided vectors. Your code should take about 15 minutes to train and test on a standard laptop.

5 Tuning and Visualization

It is possible to get higher training accuracies by trying out some different values of:

- the regularization constant λ,
- the learning rate α,
- the hidden layer size H,
- the number of iterations through the dataset (epochs)

The above hyperparameters should almost always be tuned when you work with neural networks. There is no need to try all combination exhaustively, just investigate a few and report your findings. In addition to these generic things, you might find the following problem specific issues worth investigating:

- the window size C,
- fixed work vectors vs. learning them
- how you deal with capitalization, especially if the source of your word vector is uncased.

TODO: Report on 2 effects from the above lists.

6 Extra Credit

We will give up to 20% Extra Credit for any of the following experiments. You are also free to explore a topic not listed below for extra credit (check with a TA).

Compare Word Vectors

There are many ways of training word vectors, and most of these are not designed for NER. Some examples are:

- Turian vectors (cased): `http://metaoptimize.com/projects/wordreprs/`
- word2vec (cased): `https://code.google.com/p/word2vec/`
- glove (uncased) [2]: `http://www-nlp.stanford.edu/data/`

For this part, compare with at least one other type of word vectors and report on your findings. Some aspects you could consider are the dimension of the word vector and if they are cased.

Deeper Networks

Add an extra layer to the neural network and report changes in training and testing F1 scores. The overall network would look something like this:

$$p_\theta(x^{(i)}) = g\left(Uf\left(W^{(2)}f\left(W^{(1)}\begin{bmatrix} x_{i-1} \\ x_i \\ x_{i+1} \end{bmatrix} + b^{(1)}\right) + b^{(2)}\right) + b^{(3)}\right) \tag{23}$$

You will use the same cross-entropy error as above.

Sequence Modelling

The previous label is useful for predicting the current label, you can take advantage of this by using previous/future predicted labels as input. Describe how you do this and report on the differences in network performance.

Visualization of Word Vectors

Visualizing the word vectors before and after the training process is an interesting exercise to get a qualitative understanding of what the neural network is learning. Since the word vectors are in a 50 dimensional space, the t-SNE algorithm is a neat tool to project these onto a 2D space for visualization. You can read more about the algorithm here: `http://homepage.tudelft.nl/19j49/t-SNE.html`

TODO: Visualize the word vectors before and after training with the t-sne algorithm and describe differences.

7 Grading Criteria

Your code will be run with the following command with Java 7 on Farmshare:

```
java -Xmx2g -cp "classes:extlib/*" cs224n.deep.NER ../data/train ../data/dev -print
```

Your code should finish training in about an hour or so for networks with hidden layer size $H = 100$ or less. Larger networks will understandably take longer to train, but try to keep your submission runtime below one hour.

If you did not complete the assignment on Farmshare, then please verify your code compiles and runs on it before submission. If there's anything special we need to know about your code (such as modes or flags to toggle) please place it in the `README.txt` in the `java/` subdirectory.

Part	**Points**
Baseline	5 points
Feedforward and cost function	10 points
Gradients for cost function of neural network	10 points
Gradient for regularized cost function	5 points
Gradient check	5 points
SGD training	10 points
Compare provided word vectors with random word vectors	10 points
Experiments, plots, error analysis and report	45 points
Total points	100 points
Extra Credit	20 points

The goal of this assignment is to show the state-of-the-art performance of neural networks for NER. **For full credit, we expect a dev set F1 score of at least 80%.** We will also run your network on a hidden test set.

You should turn in a write-up of the work you have done. As a rough guideline, your write-up should be around 5-6 pages and include:

1. System implementation details and design choices made
2. Results of the gradient check
3. Analysis with plots
4. Error analysis
5. Extra credit implementations
6. Anything else you tried

7.1 Submission Instructions

Please note:

- **We only accept electronic submissions**. *Do not submit a paper copy of the report*.
- If you are working in a group, then **only one member of the group needs to submit the assignment**.

You will submit your program code using a Unix script that we've prepared. To submit your program, first put your files in one directory on Farmshare. This should include all source code files, but should not include compiled class files. Please make sure your submission directory has the following format:

- **Your report PDF in the root of your submission directory**.
- **Your** `build.xml` **in the root of your submission directory, and a 'src' directory so that we can run** `ant` **to compile your code from your submission directory.**

Then submit:

```
cd /path/to/my/submission/directory
/afs/ir/class/cs224n/bin/submit
```

This script will (recursively) copy everything in *your* submission directory into the official submission directory for the class. If you need to re-submit your assignment, you can run:

```
/afs/ir/class/cs224n/bin/submit -replace
```

8 Tips and Tricks

- This model was introduced by Collobert et al. You can read about the model details and variants in their paper [3].
- We include in the starter code, the efficient Java matrix library (ejml) package. If you save all matrices of your model ($L, W, b^{(1)}, U, b^{(2)}$) in that format and create a few helper functions, you can write almost pseudo code for the major forward and backprop equations.
- If you're familiar with matlab, this will help:
`http://code.google.com/p/efficient-java-matrix-library/wiki/MatlabFunctions`
- More information on logistic regression can be found in the class notes of CS229: `cs229.stanford.edu/notes/cs229-notes1.pdf`
- An in-depth tutorial on deep learning can be found here (Highly recommended, both for understanding the basics and extended reading): `http://nlp.stanford.edu/courses/NAACL2013/`

References

[1] E. H. Huang, R. Socher, C. D. Manning, and A. Y. Ng. Improving Word Representations via Global Context and Multiple Word Prototypes. In *ACL*, 2012.

[2] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Proceedings of EMNLP*, 2014.

[3] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural Language Processing (Almost) from Scratch. *Journal of Machine Learning Research*, 12:2493–2537, 2011.