Neural Networks for Named Entity Recognition

Programming Assignment 4
CS 224N / Ling 284

Due Date: Dec. 5th, 2012

This assignment may be done individually or in groups of two. We strongly encourage collaboration; however your submission must include a statement describing the contributions of each collaborator. See the collaboration policy on the website.[1]

Please read this assignment soon and make sure that you are able to access the relevant files and compile the code. Especially if your multivariate calculus experience is limited, start working early so that you will have ample time to discover stumbling blocks and ask questions. This assignment requires you to derive new equations using the techniques from class and then implement them. Depending on your background this might be significantly harder than previous assignments. The assignment is scored out of 100 and will be normalized afterwards to fit 30% of your final grade. The required items you need to implement are marked with **TODO**.

1 Setup

Copy over the new code to your local directory and make sure you can compile the code without errors.

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

[1] http://class.stanford.edu/cs224n/Fall2012/pages/assignments

```
cd java
ant
```

You can open the train and dev data files in any text editor. The included README file also explains how to set up the project in Eclipse, which we highly recommend for debugging. This assignment is more complex than previous ones and you will need to be able to investigate your variables in a good debugger.

2 Introduction

In this exercise, you will implement a neural network for named entity recognition (NER). Before starting on the programming exercise, we strongly recommend watching the lectures. The starter code is very rudimentary and includes the following files:

Java files

`Datum.java` - A class to store the words and their labels. No need to change anything in here.
[⋆] `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).
[⋆] `NER.java` - Main function to start with and to run.
[⋆] `WindowModel.java` - This is where you might want to implement training and testing the neural network model.
⋆ indicates files you will need to complete

Data files

`train` - The labeled training data that is being read in by the FeatureFactory
`dev` - The labeled dev set you will try to push performance on
`vocab.txt` - Text file with list of words, one per line, in the same order as the wordVectors file
`wordVectors.txt` - Word vectors, one per row. You will need to read in this file

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. In this exercise, you will implement such a network for learning a single named entity class PERSON. You will derive and implement the word embedding layer, the feedforward neural network and the 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 any machine learning method we will analyze our errors, visualize model variants and tune various model parameters.

3.1 Model overview

The neural network has 3 layers – an input layer, a hidden layer and an output layer. Recall that our inputs are the word vectors which are also model parameters that we will optimize. Our top layer will be a logistic regression classifier. The final cost function (also often called loss function) will be the binary cross-entropy error, which is similar to the general cross entropy but for only two classes.

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. These vectors constitute your "features". These features are the input to a neural network which is a function that will first transform them into a "hidden" vector and then use that vector to predict a probability of how likely the window's center word is of a certain class.

3.2 Word vectors and context windows

Assume you have the following sequence of words in your training set

```
George is happy about the election outcome .
```

Each word is associated with a label y defining that word as either `PERSON` or not any named entity `O` . We define `PERSON` to be class 1 ($y = 1$) and `O` to be class 0. Each word is also associated with an index into a vocabulary, so the word `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. We will provide you with an initial set of word vectors that have been trained with an unsupervised method [1].

The vectors are $n = 50$ dimensional.

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.

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 corpus with C many of these special tokens. For simplicity, let us use $C = 1$. 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 $(1, 0, 0, 0, 0, 0, 0, 0)$. Note that the model just sees each window as a separate training instance.

3.3 Definition of feedforward network function

Each window's vectors are given as input to a single neural network layer (called a "hidden layer") which has dimensionality H. This layer is used as features for a **logistic regression** classifier which will return the probability that the center word belongs to either of the two classes. For the first window, the feed-forward equations are as follows:

$$\begin{aligned} z &= W \begin{bmatrix} x_s \\ x_1 \\ x_2 \end{bmatrix} + b^{(1)} && (1) \\ a &= f(z) && (2) \\ h &= g(U^T a + b^{(2)}). && (3) \end{aligned}$$

The final prediction h is x_1's probability of being a `PERSON`. Let us define all the involved notation: the model parameters W, U and the model functions f, g. We have the following neural network parameters: $W \in \mathbb{R}^{H \times Cn}$. For our case with a window size of $C = 3$ and if we have a reasonable hidden layer size such as $H = 100$, we get $W \in \mathbb{R}^{100\times 150}$. The bias of the hidden layer is $b^{(1)} \in \mathbb{R}^{H\times 1}$ and the bias of the logistic regression $b^{(2)}$ is a scalar. The parameters for the logistic regression weights then have to be $U \in \mathbb{R}^{H\times 1}$. Note that you could also add a single 1 to a a and use $U \in \mathbb{R}^{(H+1)\times 1}$, in other words including the bias term $b^{(2)}$ inside U.

The nonlinearity function f can be either the sigmoid function of the hyperbolic tanh function. For your default choice, please use tanh. One useful property of tanh is that its derivative can be expressed in terms of the function value itself (instead of original input):

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

The function g needs to return a probability so we will always use the sigmoid function:

$$g(z) = \text{sigmoid}(z) = \frac{1}{1+e^{-z}} \tag{5}$$

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 $y^{(i)} \in \{0, 1\}$. 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:

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

TODO: Implement this feedforward function h.

3.4 Random initialization

When you implement the feedforward function you 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, except L.
Hint: Use the function `SimpleMatrix.random` from the SimpleMatrix library that we provide with the code. You can read more about this library at http://code.google.com/p/efficient-java-matrix-library/.

3.5 Cost function

In logistic regression we want to maximize the log-likelihood of our parameters given all our (say m) data samples:[2]

$$\begin{aligned}
\ell(\theta) &= \log \prod_{i=1}^{m} p(y^{(i)}|x^{(i)};\theta) && (8)\\
&= \log \prod_{i=1}^{m} \left(h_\theta(x^{(i)})\right)^{y^{(i)}} \left(1-h_\theta(x^{(i)})\right)^{1-y^{(i)}} && (9)\\
&= \sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1-y^{(i)}) \log(1-h_\theta(x^{(i)})) && (10)
\end{aligned}$$

By convention of many optimization techniques, we will minimize the negation of log-likelihood instead. Therefore, our (almost) final cost function for the neural network is

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left[-y^{(i)}\log(h_\theta(x^{(i)})) - (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right],$$

3.6 Regularized cost function

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

[2]If this is confusing, please refer to the lecture slides on the MaxEnt model or these class notes: cs229.stanford.edu/notes/cs229-notes1.pdf.

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left[-y^{(i)}\log(h_\theta(x^{(i)})) - (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right] +$$
$$\frac{C}{2m}\left[\sum_{j=1}^{nC}\sum_{k=1}^{H} W_{k,j}^2 + \sum_{k=1}^{H} U_k^2\right]$$

The larger C is the more the regularization pushes the weights of all our parameters 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.

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 $h_\theta(x)$. 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.

Deriving the full algorithm with all the derivatives is the challenge for this assignment. You need to follow similar steps as in the lecture where we described the derivatives of the unsupervised word vector learning model. The main 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 all the following gradients (and include them either in your pdf or hardcopy):

$$\frac{\partial J(\theta)}{\partial U}, \frac{\partial J(\theta)}{\partial W}, \frac{\partial J(\theta)}{\partial b}, \frac{\partial J(\theta)}{\partial L} \tag{11}$$

Note that the derivative of L is a short form for taking the derivative of each word vector that appears in each window.

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

4.1 Gradient checking

Deriving these gradients and implementing them correctly is no easy task. Fortunately, these kinds of models have the awesome property that 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 $\frac{\partial}{\partial \theta_i} J(\theta)$; you'd like to check if f_i is outputting correct derivative values.

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

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:

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

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: Implement this gradient check, apply it to a small set of 10 windows and make sure that your gradient is correct by comparing the norm of the differences between your gradient and the numerically computed one. The difference 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 $K = 2$ iterations of the following procedure:

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

$$\begin{aligned}
U^{(t)} &= U^{(t-1)} - \alpha \frac{\partial}{\partial U} J_i(U) && (12)\\
{b^{(2)}}^{(t)} &= {b^{(2)}}^{(t-1)} - \alpha \frac{\partial}{\partial b} J_i(b^{(2)}) && (13)\\
W^{(t)} &= W^{(t-1)} - \alpha \frac{\partial}{\partial W} J_i(W) && (14)\\
{b^{(1)}}^{(t)} &= {b^{(1)}}^{(t-1)} - \alpha \frac{\partial}{\partial b} J_i(b^{(1)}) && (15)\\
L^{(t)} &= L^{(t-1)} - \alpha \frac{\partial}{\partial L} J_i(L), && (16)
\end{aligned}$$

where α is the learning rate (a good one to try is $\alpha = 0.001$) and we define J_i to be the cost of only the ith sample.

After the training completes you will proceed to report the training and testing accuracy of your classifier by computing the F1 score of examples it got correct. If your implementation is correct, and with hidden layer size $H = 100$, window size $C = 5$ and learning rate $\alpha = 0.001$, you should see a testing F1 score of at very least 61.0%.

TODO: Implement SGD training and report your initial training and testing F1 scores. Your code should not take about 15 minutes to train and test on a standard laptop.

5 Network Analysis

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

1. the regularization constant C,
2. the learning rate α,
3. the window size C,
4. the hidden layer size H,
5. the number of iterations through the dataset K,
6. training the word vectors or keeping them fixed or
7. other ideas you have

TODO: Vary at least 3 of these hyperparameters (by going up and down from the initial values we gave you above) and show plots of how training and testing F1 scores change. Can you make the model overfit? Is it underfitting?

The above hyperparameters should almost always be tuned when you work with neural networks. Below, we have some more difficult analyses.

TODO: Submit one of the following three, implement or derive it and include it in your submission.

1. Visualizing the word vectors before and after training with the t-sne algorithm http://homepage.tudelft.nl/19j49/t-SNE.html and describe differences.

2. 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:

$$h_\theta(x^{(i)}) = g\left(U^T f\left(W^{(2)} f\left(W^{(1)} \begin{bmatrix} x_{i-1} \\ x_i \\ x_{i+1} \end{bmatrix} + b^{(1)}\right) + b^{(2)}\right) + b^{(3)}\right) \tag{17}$$

   You will use the same cross-entropy error as above. If you can derive the necessary equations and implement this function you will have mastered the full backpropagation algorithm in all its glory.

3. Prove that a softmax classifier for 2 classes is equivalent to a logistic regression classifier with only a single weight vector.

Extra Credit TODO: For every additional one you do (on top of the one everybody has to do), you will get 10 extra credit points.

Extra Credit TODO: The group that gets the highest F1 score will get an additional 10 extra credits.

6 Tips and Tricks

This model was introduced by Collobert et al. You can read about the model details and variants in their paper [2].

The first token in the vocabulary file is the unknown word token that you will also use for words that are in your training or dev sets but which are not in the original vocabulary file.

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`

More tutorial materials on neural networks, the backpropagation algorithm and gradient checks can be found in the first three links of this tutorial website: `http://ufldl.stanford.edu/wiki/index.php/UFLDL_Tutorial`.

You can find a very detailed derivation of the full backpropagation algorithm (for a deep neural network with an arbitrary number of layers) in section 4 of this document: http://nlp.stanford.edu/~socherr/deepLearningDerivations.pdf

7 Grading

Part	**Points**
Feedforward and Cost Function	20 points
Gradients for cost function of neural network	20 points
Gradient for regularized cost function	10 points
Gradient check	10 points
SGD training	10 points
Network analysis experiments, plots and report	30 points
Total points	100 points
Maximum possible extra credits	30 points

8 Submitting the assignment

8.1 The program: electronic submission

You will submit your program code using a Unix script that we have prepared. To submit your program, first put all the files (but not the 4 files in the data directory) to be submitted in one directory on a Leland machine (or any machine from which you can access the Leland AFS filesystem). This should include all source code files, but should not include compiled class files or large data files. Normally, your submisssion directory will have a subdirectory named src which contains all your source code. When you're ready to submit, please cd to your submission directory, and then type:

```
/afs/ir/class/cs224n/bin/submit-pa4
```

This will (recursively) copy everything in your submission directory into the official submission directory for the class. If you need to resubmit it type

```
/afs/ir/class/cs224n/bin/submit-pa4 -replace
```

We will compile and run your program on the Leland systems, using ant and our standard build.xml to compile, and using java to run. So, please make sure your program compiles and runs without difficulty on the Leland machines. If there's anything special we need to know about compiling or running your program, please include a README file with your submission.

Your code doesn't have to be beautiful but we should be able to scan it and figure out what you did without too much pain.

8.2 The report: turn in a hard copy

You should turn in a write-up of the work you have done, as well as the code. The write-up should specify what you built and what choices you made, and should include the accuracies, etc., of your systems. If you are an on-campus student, your write-up must be submitted as a hard copy. Hard copies may be submitted in class, or in the box outside Professor Manning's office. SCPD students may submit the write-up electronically on CourseWare.

There is no set length for write-ups, but a ballpark length might be 8 pages, including your evaluation results, a graph or two, error analysis, and some interesting examples. Your write-up should not exceed 15 pages in length.

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] 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.