CS224N PA4 - Report

Will Meyer - Autumn 2013

1 – Introduction

In this report, we examine my implementation of a feedforward neural network utilizing stochastic gradient descent-based training on backpropagation-computed gradients to learn named entity classification for distinguishing words belonging to a single NER class, "PERSON", as opposed to words belonging to a catch-all "O", or "not any named entity", class. With appropriate tuning of the hyperparameters of the neural network, the algorithm was able to achieve an F1 score of .750 testing on a given "dev" data set and .714 testing on a "test" data set.

2 – Implementation

The basic design of the feedforward neural network is as follows: it takes as input a series of word vectors, known as a "window", centered around a given word x_i, transforms them into a "hidden" vector of dimensionality H, then uses this "hidden" vector to output a classification for x_i. The feedforward neural network's classification is based on an output probability that a given word x_i belongs to the classification of "PERSON"; the probability results in a binary determination of classification for x_i, where x_i is classified as "PERSON" in case of a probability output greater than or equal to 0.5, and "O" otherwise. The probability output for x_i, h, is determined by the following set of equations:

$$z = W\begin{bmatrix} x_{i-1} \\ x_i \\ x_{i+1} \end{bmatrix} + b^{(1)} \qquad a = f(z) \qquad h = g(U^T a + b^{(2)})$$

We define the components of the equations as follows:

$\begin{bmatrix} x_{i-1} \\ x_i \\ x_{i+1} \end{bmatrix}$: The window for a given x_i. Here, each x_i represents the word vector corresponding to the word i, where x_i is an n-dimensional vector. In this case, the window size C_w is 3, as the window vector contains 3 words.

W : The weight matrix, in $R^{H \times (C_w \times n)}$ – where H refers to the size of the hidden layer of the network, used for a linear transformation from the input layer to the hidden layer of the network.

$b^{(1)}$: The bias vector corresponding to the intercept of the linear transformation achieved by W.

$f(z)$: The nonlinear transformation function applied to the linear transformation achieved by z. In this model, f corresponds to the hyperbolic tanh function.

U : The weight matrix used to transform the hidden layer vector into a scalar.

$b^{(2)}$: The bias scalar corresponding to the intercept of the transformation achieved by *U*.

g : The function that returns the final probability that x_i is classified as "PERSON". In this model, *g* corresponds to the sigmoid function: sigmoid(z) = 1/(1+e^(-z)).

In this model, the values for the parameters *W*, *U*, $b^{(1)}$ and $b^{(2)}$ are initialized uniformly at random in the range $[-\varepsilon_{\text{init}}, \varepsilon_{\text{init}}]$, where $\varepsilon_{\text{init}}$ = sqrt(6)/sqrt(fanIn+fanOut), fanIn = *n**C_w and fanOut = *H*.

The stochastic gradient descent training is designed to minimize a regularized cost function, corresponding to the log-likelihood of *m* given data samples. This regularized cost function is given by

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

where C_{reg} corresponds to a Gaussian prior regularization term.

3 – Gradients

The backpropagation algorithm used for stochastic gradient descent relies on updating each parameter *U*, *W*, $b^{(1)}$, $b^{(2)}$, and *L* (the n x |V|, where V is the size of the input vocabulary, matrix containing word vectors) in a series of single steps; each step subtracts, from each parameter, a quantity equal to the learning rate α times the cost function *J* derived with respect to the given parameter. The derived gradients for each parameter are presented as follows:

$$\frac{\partial J(\theta)}{\partial U} = \frac{1}{m}\sum_{i=1}^{m}\left[\delta^{(2)}a^{(i)}\right] + \frac{C_{reg}}{m}U, \text{ where } \delta^{(2)} = h_\theta(x^{(i)}) - y^{(i)}$$

$$\frac{\partial J(\theta)}{\partial W} = \frac{1}{m}\sum_{i=1}^{m}\left[\delta^{(1)}(x^{(i)})^T\right] + \frac{C_{reg}}{m}W, \text{ where } \delta^{(1)} = \delta^{(2)}U^T(1-\tanh^2(z^{(i)}))$$

$$\frac{\partial J(\theta)}{\partial b^{(1)}} = \frac{1}{m}\sum_{i=1}^{m}\left[\delta^{(1)}\right]$$

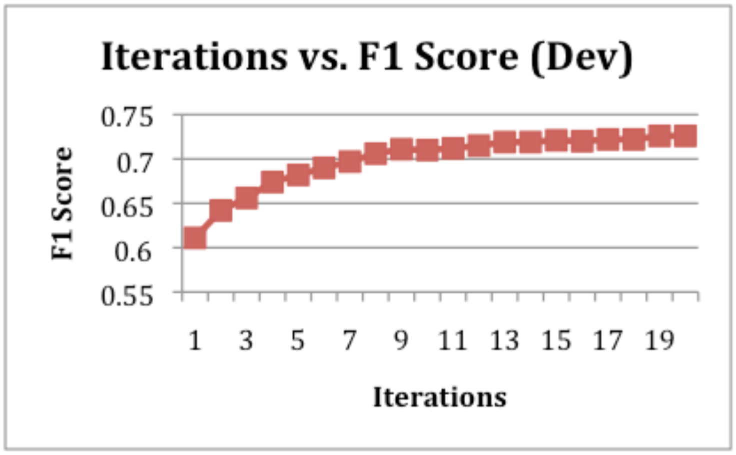
$$\frac{\partial J(\theta)}{\partial b^{(2)}} = \delta^{(2)}$$

$$\frac{\partial J(\theta)}{\partial L} = \frac{1}{m}\sum_{i=1}^{m}\left[W^T\delta^{(1)}\right]$$

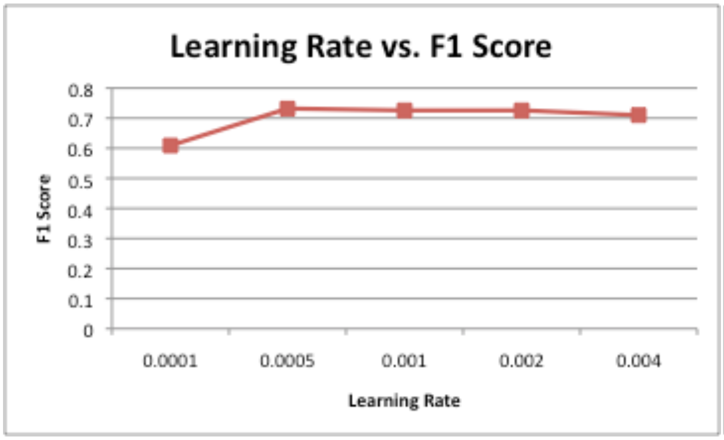
My neural network contains code to numerically verify derived gradient values by checking the approximation that $f_i(\theta) \approx \frac{J(\theta^{(i+)}) - J(\theta^{(i-)})}{2\varepsilon}$, where ε=10^-4, is less than 10^-7. This code is located at line 165 of WindowModel.java under the function "gradientCheck"; it is toggled on/off by a boolean variable "checkingGradient" in the WindowModel constructor function. A sample training run shows gradient checks to produce differences of roughly 10^-20 for each modified parameter, confirming the derivations to be correct.

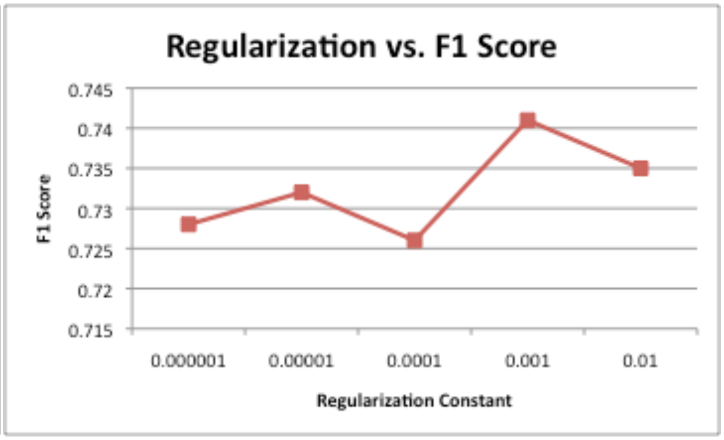
4 – Network Analysis

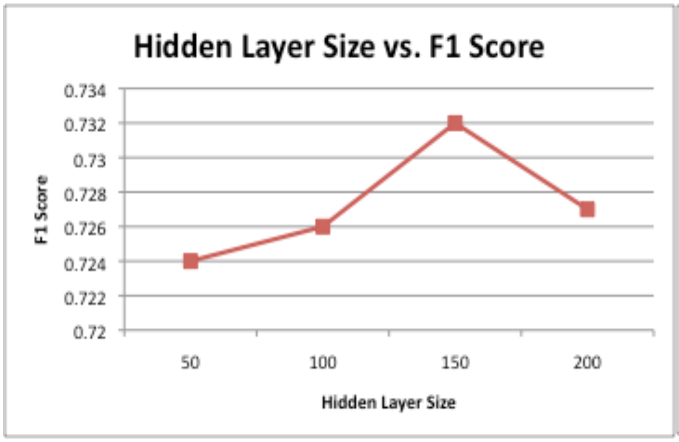
The neural network was run with the following initial parameters: window size C_w = 5, hidden layer size H = 100, learning rate α = 0.001, regularization constant C_{reg} = 0.0001, number of iterations through the data set K = 2. Tuning the number of iterations so as to extend data training, proved to have an extremely positive effect on F1 scores on a held-out development data set:

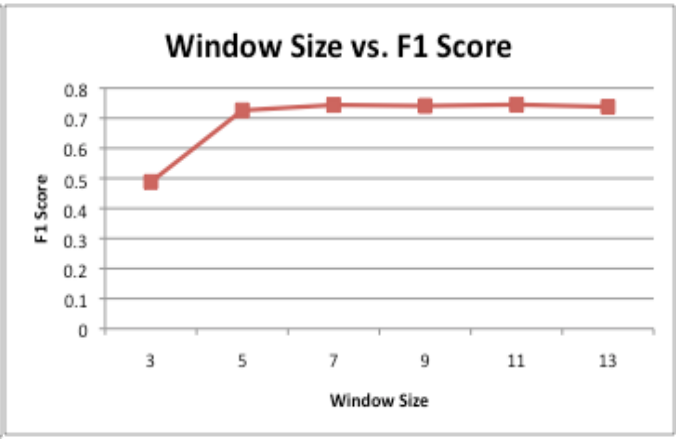


The remaining hyperparameters were then tested in isolation – in other words, the number of iterations was kept constant (in this case, K=20, since at 20 iterations, the model had reached a point where recall was showing limited improvement – roughly .001 per iteration – and precision was actually beginning to decrease, suggesting the beginnings of overfitting to the training data). The results of tuning each hyperparameter are shown below, and on the next page:







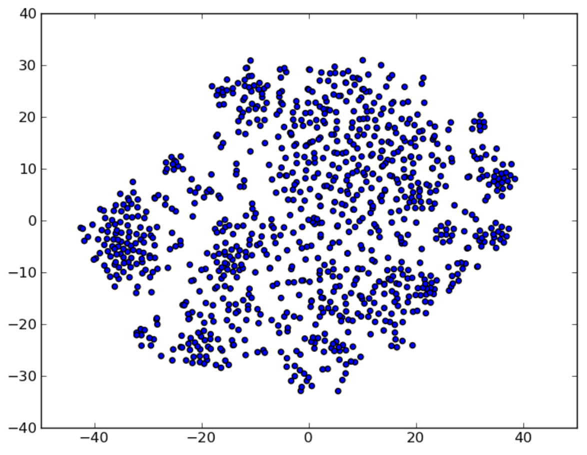


Each hyperparameter's pattern of data fitting stands out as having a nature characteristic of its associated parameter. Shortening the size of a given window (to 3, for example), or setting the learning rate too low (to .0001), cause extreme underfitting and inaccurate performance due to, respectively, stripping the model of contextual evidence and lowering the rate at which it learns (essentially making it approximate untrained data). By contrast, increasing window size and learning rate made the model become accurate faster, but led to overfitting after a lower number of iterations due to both parameters' tendency to approximate training data; for example, using a window of 9 produced an F1 score of .75 after 11 iterations, but subsequent iterations lowered this score, resulting in an F1 score of .741 on the 20th iteration. Adjusting the regularization constant tended to have relatively minute effects on performance, which is expected behavior, as regularization contributes relatively little to each gradient update; still, increasing or decreasing the regularization constant past certain points still decreased performance, as too high of a constant pushed weights towards 0 and limited the effectiveness of training (underfitting the data) while too low of a constant produced overfitting. Finally, increasing and decreasing the size of the hidden layer also produced notable effects; too small of a hidden layer compressed input too much to produce useful weights, decreasing F1 scores by underfitting, while too large of a hidden layer caused the data to approximate training data (and increased training time, causing reduced performance given a set number of iterations).

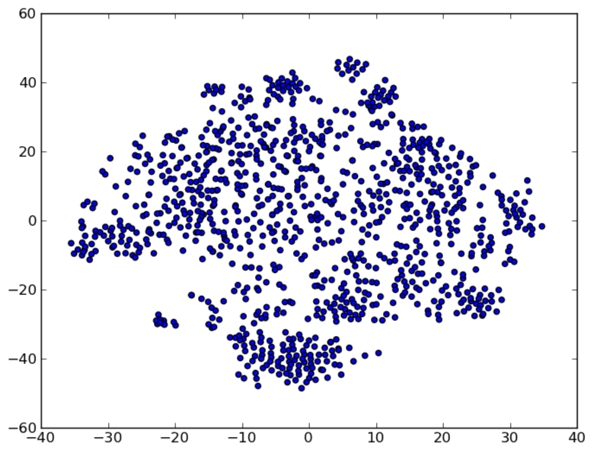
To achieve roughly optimal scores - .750 testing on the dev set and .714 testing on the test set – the hyperparameters used were $C_w = 11$, $H = 150$, $\alpha = 0.0005$, $C_{reg} = 0.001$, $K = 20$.

5 – Data Visualization

A visualization of the data provides some interesting insights. Training vectors tends to produce clusters of words that have similar contextual usage; for example, words for months ("January", "December") or words for places ("Egypt", "Chicago") are grouped together, as are modifiers with roughly similar usage ("along", "against") or just related words in general ("north", "east"; "science", "technology"). Some of these words are already grouped together, or relatively close, in untrained vectors, but training solidifies the clusters and tends to bring in outliers. Below are visualizations of samples of 1000 word vectors before and after training, without word labels (as labels are too small to show properly in this document):



Before Training



After Training

6 – Error Analysis

There are two types of errors made by the neural network's classification system: "O" words misclassified as "PERSON" (false positives), and "PERSON" words misclassified as "O" (false negative). Examining the data leads to some theorizing regarding the classifier's remaining errors. For example, unusual names corresponding to people ("Dutroux", "Buyoya", etc.) are often misclassified as "other"; this is most likely due to a combination of their contextual usage being roughly the same as that of a non-person object (i.e. only the last name is used) and their status as a previously unseen word (meaning that they are represented by a generic unknown token, rather than a word, and contextual evidence is the only classification information the network has). By a similar error, place names and other proper nouns ("Chicago", "Bangladesh", "Warner Bros") are classified as people, most likely due to contextual usage that is very similar to that of a name – they are capitalized, used as the subjects of sentences, etc.. Capitalized pronouns ("I", "He", "My") or capitalized words used as the subjects of sentences ("Analysts", "Organizers") also are classified as people; again, this is most likely due to contextual information being similar to names in that these words begin with capital letters and are used in sentences where objects referring to people are generally used. Numbers ("185", "5", "1996-08-31") are also routinely classified as "PERSON" objects – the reason for this is less clear, but it may also have something to do with relatively low or nonexistent frequencies in training data combined with contextual usage information (for example, the "5" is used in reference to "Interstate 5", both of which are wrongly classified as people; for some reason, the system appears to view numbers as roughly equivalent to capital letters, meaning that this most likely represents the same information pattern as a two-word string where both words start with a capital letter – the most common representation of a name). Conversely, individuals' names that are also English words ("Dick", "Bill", "Wang", "Dole") are misclassified as objects, most likely due to their relatively frequent appearance in their object form relative to their usage as names.

7 – Extra Credit

I implemented the extra credit – adding an extra layer to the neural network – producing a network whose characteristic feedforward equation approximates the following:

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

The gradients of the new network were given by the following:

$$z = W^{(1)}\begin{bmatrix} x_{i-1} \\ x_i \\ x_{i+1} \end{bmatrix} + b^{(1)} \qquad a = f(z) \qquad z^{(2)} = W^{(2)}\left[a\right] + b^{(2)} \qquad a^{(2)} = f(z^{(2)})$$

$$\frac{\partial J(\theta)}{\partial U} = \frac{1}{m}\sum_{i=1}^{m}\left[\delta^{(3)} a^{(2)(i)}\right] + \frac{C_{reg}}{m}U, \text{ where } \delta^{(3)} = h_\theta(x^{(i)}) - y^{(i)}$$

$$\frac{\partial J(\theta)}{\partial W^{(2)}} = \frac{1}{m}\sum_{i=1}^{m}\left[\delta^{(2)}(a^{(i)})^T\right] + \frac{C_{reg}}{m}W^{(2)}, \text{ where } \delta^{(2)} = \delta^{(3)}U^T(1 - \tanh^2(z^{(2)(i)}))$$

$$\frac{\partial J(\theta)}{\partial W^{(1)}} = \frac{1}{m}\sum_{i=1}^{m}\left[\delta^{(1)}(x^{(i)})^T\right] + \frac{C_{reg}}{m}W^{(1)}, \text{ where } \delta^{(1)} = W^{(2)^T}\delta^{(2)} \bullet (1 - \tanh^2(z^{(i)}))$$

$$\frac{\partial J(\theta)}{\partial b^{(1)}} = \frac{1}{m}\sum_{i=1}^{m}\left[\delta^{(1)}\right]$$

$$\frac{\partial J(\theta)}{\partial b^{(2)}} = \frac{1}{m}\sum_{i=1}^{m}\left[\delta^{(2)}\right]$$

$$\frac{\partial J(\theta)}{\partial b^{(3)}} = \delta^{(3)}$$

$$\frac{\partial J(\theta)}{\partial L} = \frac{1}{m}\sum_{i=1}^{m}\left[W^T\delta^{(1)}\right]$$

My extra layer simply inserted a layer of hidden size *H/2* for the second layer. When run using parameters identical to the baseline – $C_w = 5$, $H = 100$, $\alpha = 0.001$, $C_{reg} = 0.0001$, and $K = 2$ – the network actually performed slightly worse, achieving an F1 score of .650 (as opposed to .656) on the development data set. However, when iterations increased to $K = 20$, the extra credit network outperformed the basic network from the assignment, achieving an F1 score of .733 (as opposed to .726). Of particular note is that recall was still showing an increase of > .002 on the final two iterations, which suggests that the model's F1 score could probably be improved slightly with several more iterations, which in turn fits with the general concept of hidden networks – a larger number of layers will take more iterations to train.

The extra credit can be found in the file WindowModelEC.java; it can be run by commenting out the line in NER.java running the basic WindowModel and uncommenting the line running the EC version.