Learning Named Entity Recognition with Deep Neural Networks

Awni Hannun
awni@stanford.edu

Ibrahim Cotran
icotran@stanford.edu

December 6, 2012

1 Introduction

Named entity recognition (NER) has strong applications in natural language processing including Machine Translation and semantic representation of language. Here we explore the application of single and multi-layer Neural Networks (NN) to the task of NER. We attempt to learn a simplified NER task of the binary classification of a word as a person (1) or not (0).

We learn both a single hidden layer NN architecture and a double hidden layer NN architecture. After optimizing the hyperparameters and training schedule of the model, we are able to achieve an F1 score of 93%. These results are particularly impressive given that we encode no specific information about natural language rules or features to the model.

1.1 Supplemental Material

In the Appendix A of this paper we present a full derivation of the backpropagation algorithm for finding the gradients of the parameters of a single layer NN.

We also prove in Appendix B that a softmax classifier with only 2 classes is equivalent to logistic regression with a single weight vector.

Lastly in Appendix C we include visualizations of the word vector representations both before and after training our model with the optimal architecture and parameters. We use the t-SNE algorithm for dimensionality reduction [1].

2 Model

We learn and tune both a single and double hidden layer Nueral Network. The cost function for both NNs is given by

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

The Feedforward function for the single hidden layer NN is

$$h_\theta(x^{(i)}) = g\left(U^T f\left(Wx^{(i)} + b^{(1)}\right) + b^{(2)}\right) \tag{2}$$

The Feedforward function for the double hidden layer NN is

$$h_\theta(x^{(i)}) = g\left(U^T f\left(W^{(2)} f\left(W^{(1)}x^{(i)} + b^{(1)}\right) + b^{(2)}\right) + b^{(3)}\right) \tag{3}$$

In both (2) and (3), f and g are given by the *sigmoid* and *hyperbolic tangent* functions respectively.

We use Stochastic Gradient Descent (SGD), $\theta^{(t)} := \theta^{(t-1)} - \alpha \frac{\partial}{\partial \theta^{(t-1)}} J_i(\theta)$, to find the values of the parameters $W^{(i)}, U, b^{(i)}$ and L which minimize (1) and thereby maximize the log-likelihood of our data. Here L is the word vector matrix containing n-dimensional pretrained word vectors for each word in the vocabulary (in this paper $n = 50$).

In order to implement SGD we must first derive the gradient of (1) with respect to the above parameters. Appendix A steps through the backpropagation algorithm in order to derive these parameters.

We also implement the numerical gradient calculation method in order to verify the correctness of our gradient derivations and implementations in both the single and double hidden layer NNs. We find that the norm of the difference between the numerically computed and the analytically computed gradients is significantly less than 10^{-7}, on the order of 10^{-9}, thus we are confident in the correctness of our gradient derivation and implementation.

In 3 we explore the results found tuning the model hyperparameters for both NN architectures. These include the hidden layer size, the window size, the learning rate and the regularization constant.

The input vector $x^{(i)}$ to our model represents an $n * C$ dimensional column vector where C is the contextual window size for inputs to the model, i.e. the $C-1$ words that surround the word we atttempt to classify in the training corpus. We explore various tunings of this window size in 3 as well.

We implement and observe the difference between using the full document as the context or each sentence as the context. In practice this is done using start and end symbols which are placed in windows that overlap the beginning and end of the document or the beginning and end of each sentence.

Our final model appends a capitalization vector [2] to each word vector corresponding to one of four capitalization scenarios. These scenarios are: all lowercase, first word capitalized, all uppercase and at least one other than the first letter is uppercase. We use a capitalization matrix $LC \in \mathbb{R}^{5 \times 4}$ in order to store and update these parameters.

3 Netwok Analysis

We now discuss the tuning of the Neural Network architecture and hyper-parameters in order to optimize F1 in the NER classification task.[1]

Implementing the capitalization feature vectors as described in the section 2 and [2] drastically improved both the train and dev F1 scores as seen in 1. Clearly this implies that NER for person classification relies heavily on word capitalization information. Previously the model was unable to distinguish between a capitalized and non-capitalized word which actually contains a lot of information about the classification of the word since person words tend to be uppercased in the first letter. Thus we add this feature vector to each word vector in tuning the rest of our model hyper parameters.

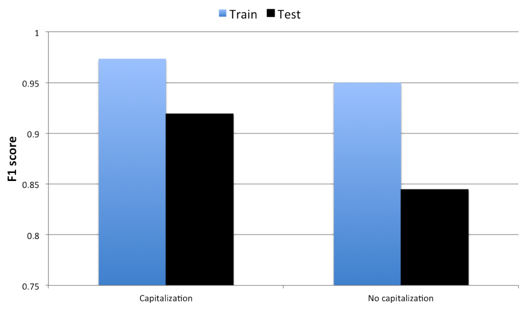

Figure 1: F1 scores of model with the presence and absence of capitalization matrix (C = 0.0 W = 5 H = 100 A = 0.01)

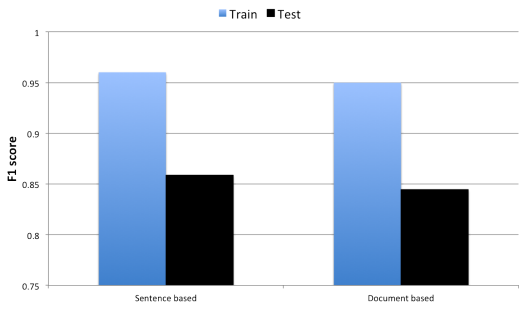

Figure 2: F1 scores of model with document and sentence based start and end tokens (C = 0.0 W = 5 H = 100 A = 0.01)

[1] We use the following short form to represent the hyperparameters used in the neural network run. (C = [regularization constant] W = [window size] H = [hidden layer size] A = [learning rate]).

Varying where we place our start and end tokens did not have much of an effect on our F1 score. We see from Figure 2 that although the sentence model performs slightly better than the document based model, due to the randomness of initial values and the random permutation optimization implemented for SGD (described in Section **??**) this can be attributed to the inherent noise in our system. In the mean, varying this parameter does not show significant changes to our system. Thus this feature is not added to our system before tuning other hyper parameters.

The next hyperparameter we tuned was the learning rate for the SGD update step. Figures 3 and 4 displays the learning curves for three different values of the learning rate. The smallest of value, .0001, is not able to break out of bad local minima when optimizing and thus converges to a much lower F1 value than the two higher learning rates. The learning rate of .001 as in 3 and 4 is the most stable yet takes much longer to converge than .01 and thus we chose to use .01 as our final learning rate in tuning the other parameters. Notice that with a learning rate of .01 our model is much more erratic as it oscillates about the minimum yet does not converge precisely. We use this learning rate in order to get baseline results for tuning parameters, however, in tuning the final model we anneal our learning rate in order to precisely converge to the highest optimum.

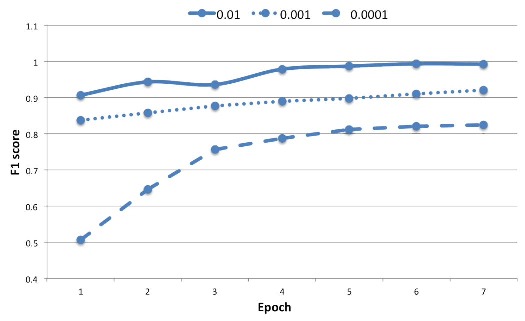

Figure 3: F1 scores of the training set varying the learning rate for stochastic gradient (C = 0.0001 W = 7 H = 100)

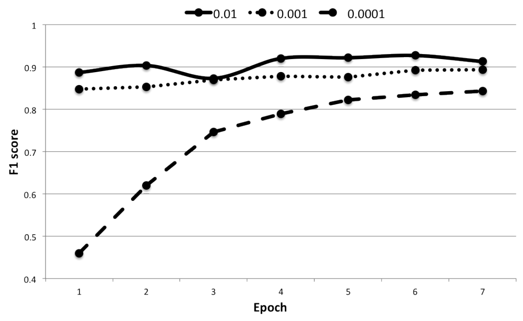

Figure 4: F1 scores of the test set varying the learning rate for stochastic gradient (C = 0.0001 W = 7 H = 100)

After optimizing the input features and learning rate of the model we then tune the three hyper parameters: window size, hidden layer size and regularization. We hold all other parameters to the model fixed and vary only the hyper parameter we attempt to optimize for. In Figure 5 we see that the model does not vary significantly with window size changes above size 5 with an optimal value at 7. Thus we choose window size of 7 for our final model. In Figure 6 we see again that as we vary the hidden layer size the model performance does not change significantly. However, we do note that the training time increases substantially as we increase hidden layer size. Thus given the time to run and the fact that a hidden layer size of 100 performs as well or better than the others, we choose a hidden layer size of 100 for our optimal model. In Figure 7 we see that the model performance increases slightly with a small regularization weight (.0001) however drops off significantly if we increase the regularization constant above .001. Thus we use regularization constant of 0.0001 for our optimal model.

After tuning some of our hyperparameters, we tested our NN model by adding a second hidden layer and running that model against our first hidden layer. The results can be seen in Figure 8. We find that the second hidden layer is more erratic on the training set (it has a higher standard deviation) but in general the results are comparable using both layers. It is worth noting that for our training set, even with one hidden we achieve near perfect score. Since our F1 score is already very high and our model learns the most from our L matrix, the incremental value that we have from adding a second hidden layer is not noticeable. However, after running our model over a number of different tests, we found that adding our second layer did slightly improve our score by a small amount.

The best result achieved was an F1 score of 93.2%. Note that this is using a token level F1 evaluation metric. On an entity level, this score is 88.0%. For this score we used the following hyperparameters: C = 0.0001, W = 7, H = 200, A = 0.01 (with annealing), K = 10.

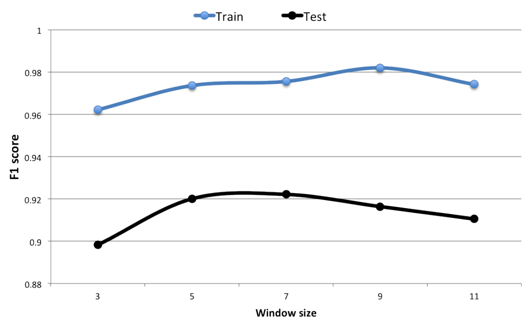

Figure 5: F1 scores of train and dev sets with a 1 hidden layer model varying the window size. (C = 0.0 H = 100 A = 0.01)

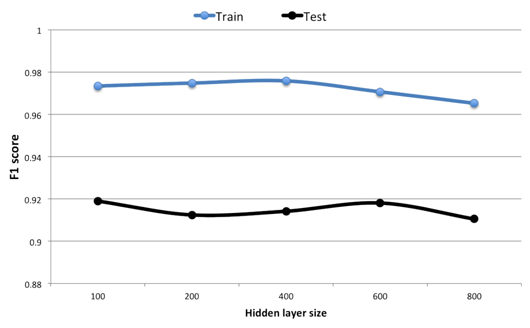

Figure 6: F1 scores of train and dev sets with a 1 hidden layer model varying the hidden layer size. (C = 0.0 W = 5 A = 0.01)

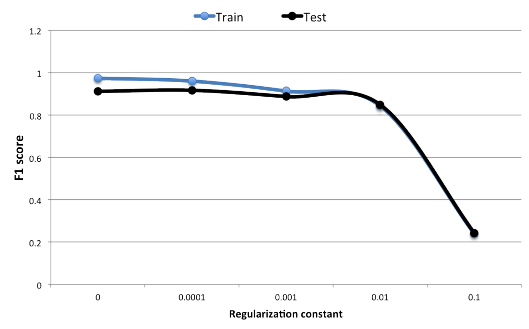

Figure 7: F1 scores of train and dev sets with a 1 hidden layer model varying the regularization constant. (W = 5 H = 100 A = 0.01)

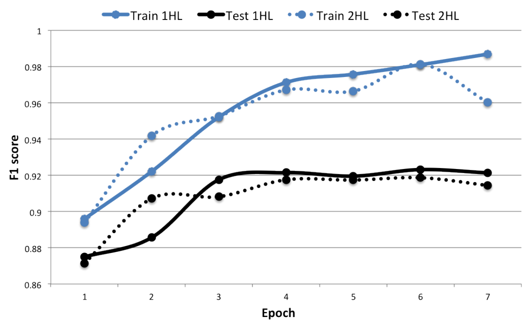

Figure 8: F1 scores of train and dev sets with both the single and double hidden layer NN architectures. (C=0.001 W = 7 H = 100 A = 0.01)

4 Further Optimizations

This section outlines and describes several further optimzation techniques that we implemented in the model or chose not to implement and why.

The first optimization we added to the model was to randomly shuffle the training sample set before each epoch of SGD (an epoch is a full iteration through the training set). The purpose of implementing this was to allow for smoother learning throughout the epoch rather than biasing the gradient with several related word updates at a single step. We do not show the results here due to space constraints, but we found that the optimal score achieved using this method of learning was slightly higher and the learning throughout the epochs was much smoother and more predictably converged to an optimum.

Given the training F1 scores achieved in Figure 8, we see that the model fits the training set almost perfectly achieving an F1 of approximately 99%. We also note that the models do not noticeably overfit the test that despite perfect fitting of the training set. From this we conclude that better optimization techiniques such as LBFGS or conjugate gradient methods would not allow the model to better fit the training data and likely would run slower since they calculate the gradient over the full training set over each iteration. Thus we chose not to explore other optimization techiniques and conclude that SGD performance is not the bottle neck in achieving higher results.

In order to achieve the optimal results of F1=93.2% token and F1=88% entity as mentioned in 3 we employ a technique known as annealing in order to ensure that our model converges perfectly to the optimal value for parameters. We anneal the learning rate by an order of magnitude after the first 7 epochs and another order of magnitude after the first 9 epochs and converge to the optimal results. SInce we start the model with a high learning rate of 0.01 we find that the SGD oscillates around an optimal value and never quite converges thus dropping the learning rate allows us to converge more precisely to the optimal value.

5 Error Analysis

-quick analysis of before/after tsne visualizations.

References

[1] L.J.P. van der Maaten and G.E. Hinton. Visualizing High-Dimensional Data Using t-SNE. *Journal of Machine Learning Research* 9(Nov):2579-2605, 2008.

[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*. 12:24932537, 2011.

Appendices

A Backpropagation

We now derive the Backpropagation algorithm for finding the gradients of the cost function of the neural network. For our parameters $W \in \mathbb{R}^{H \times nC}$, $b^{(1)} \in \mathbb{R}^H$, $U \in \mathbb{R}^H$,$b^{(2)} \in \mathbb{R}$ and $L \in \mathbb{R}^{nC \times V}$ the cost function for our neural network is defined as follows.

$$\begin{aligned} J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left[-y^{(i)} log(h_\theta(x^{(i)})) - (1 - y^{(i)}) log(1 - h_\theta(x^{(i)})) \right] + \\ \frac{C}{2m} \left[\sum_{j=1}^{nC} \sum_{k=1}^{H} W_{k,j}^2 + \sum_{k=1}^{H} U_k^2 \right] \end{aligned} \tag{4}$$

$$h_\theta(x^{(i)}) = g\left(U^T f\left(W x^{(i)} + b^{(1)} \right) + b^{(2)} \right) \tag{5}$$

$$f(z) = tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{6}$$

$$g(z) = sigmoid(z) = \frac{1}{1 + e^{-z}} \tag{7}$$

The derivatives of the *tanh* and *sigmoid* functions g and f are given by

$$\frac{d}{dz} f(z) = 1 - \left(\frac{e^z - e^{-z}}{e^z + e^{-z}} \right)^2 = 1 - f^2(z) \tag{8}$$

$$\frac{d}{dz} g(z) = \frac{1}{1 + e^{-z}} \left(1 - \frac{1}{1 + e^{-z}} \right) = g(z)(1 - g(z)) \tag{9}$$

Notice that the derivatives of both the hyperbolic tangent and logistic functions are defined completely in terms of the functions themselves. This will simplify deriving the gradients for each parameter. Since we will implement stochastic gradient descent in order to optimize our cost function with respect to the parameters $U, b^{(2)}, W, b^{(1)}$ and L we only need to observe the gradient with respect to these parameters at a single training example at a time. Thus we find the gradient for the simplified cost function below where x and y represent a single training example.

$$\begin{aligned} J(\theta) = -y log(h_\theta(x)) - (1 - y) log(1 - h_\theta(x)) + \\ \frac{C}{2} \left[\sum_{j=1}^{nC} \sum_{k=1}^{H} W_{k,j}^2 + \sum_{k=1}^{H} U_k^2 \right] \end{aligned} \tag{10}$$

We now derive the gradients $\frac{\partial J}{\partial U}$,$\frac{\partial J}{\partial b^{(2)}}$, $\frac{\partial J}{\partial W}$, $\frac{\partial J}{\partial b^{(1)}}$ and $\frac{\partial J}{\partial L}$. In order to simplify the gradient for each parameter notice that the regularization term (the second term in the cost function) does not depend on $b^{(1)}$, $b^{(2)}$ or L thus vanishes for those terms. Also since the gradient distributes over addition we can take the derivative of this term with respect to U and W separately and add it in later. The gradient of the regularization term for the two parameters is

$$\frac{\partial}{\partial U} \frac{C}{2} \left[\sum_{j=1}^{nC} \sum_{k=1}^{H} W_{k,j}^2 + \sum_{k=1}^{H} U_k^2 \right] = CU \tag{11}$$

$$\frac{\partial}{\partial W} \frac{C}{2} \left[\sum_{j=1}^{nC} \sum_{k=1}^{H} W_{k,j}^2 + \sum_{k=1}^{H} U_k^2 \right] = CW \tag{12}$$

To simplify notation we define $z^{(1)}, z^{(2)}, a^{(1)}, a^{(2)}$ as follows.

$$\begin{aligned} z^{(1)} =& Wx + b^{(1)}, \\ a^{(1)} =& f(z^{(1)}), \\ z^{(2)} =& U^T a^{(1)} + b^{(2)}, \\ a^{(2)} =& g(z^{(2)}) \end{aligned} \tag{13}$$

Finally we let the even further simplified cost function written in terms of the above defininitions without regularization (as we will add that in later) be defined as

$$J(U, b^{(2)}, W, b^{(1)}, L) = -ylog(a^{(2)}) - (1-y)log(1-a^{(2)}) \tag{14}$$

First we derive $\frac{\partial J}{\partial U}$ with respect to (14). Using the chain rule and remembering the derivative of the sigmoid function from (9) we find

$$\begin{aligned} \frac{\partial J}{\partial U} &= \left(\frac{-y}{a^{(2)}} + \frac{(1-y)}{1-a^{(2)}} \right) \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial U}, \\ \frac{\partial J}{\partial U} &= \left(\frac{-y}{a^{(2)}} + \frac{(1-y)}{1-a^{(2)}} \right) (a^{(2)})(1-a^{(2)}) \frac{\partial z^{(2)}}{\partial U}, \\ \frac{\partial J}{\partial U} &= \left(-y(1-a^{(2)}) + (1-y)a^{(2)} \right) \frac{\partial z^{(2)}}{\partial U}, \end{aligned}$$

$$\frac{\partial J}{\partial U} = \left(a^{(2)} - y \right) \frac{\partial z^{(2)}}{\partial U} \tag{15}$$

Now we only need to find the partial $\frac{\partial z^{(2)}}{\partial U}$ in (15) in order to derive the gradient.

$$\frac{\partial z^{(2)}}{\partial U} = \frac{\partial}{\partial U} \left(U^T a^{(1)} + b^{(2)} \right) = (a^{(1)})^T \tag{16}$$

Putting together (15) and (16) we arrive at

$$\frac{\partial J}{\partial U} = \left(a^{(2)} - y \right) (a^{(1)})^T \tag{17}$$

Also using the exact same steps to arrive at (15) we can solve for $\frac{\partial J}{\partial b^{(2)}}$

$$\frac{\partial J}{\partial b^{(2)}} = \left(a^{(2)} - y \right) \frac{\partial z^{(2)}}{\partial b^{(2)}} = \left(a^{(2)} - y \right) \tag{18}$$

We find $\frac{\partial J}{\partial W}$ using the same derivation as in (15) and applying the chain rule again.

$$\frac{\partial J}{\partial W} = \left(a^{(2)} - y \right) \frac{\partial z^{(2)}}{\partial W} = \left(a^{(2)} - y \right) \frac{\partial z^{(2)}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial W} \tag{19}$$

To simplify the derivation we take all of the partial derivatives in (19) with respect to a single element, W_{ij}. First notice that only the i^{th} element of $a^{(1)}$ and $z^{(1)}$ depend on the ij^{th} element of W, thus we have

$$\frac{\partial J}{\partial W_{ij}} = \left(a^{(2)} - y \right) \frac{\partial z^{(2)}}{\partial a_i^{(1)}} \frac{\partial a_i^{(1)}}{\partial z_i^{(1)}} \frac{\partial z_i^{(1)}}{\partial W_{ij}} \tag{20}$$

Now we can find each partial separately and combine them.

$$\frac{\partial z^{(2)}}{\partial a_i^{(1)}} = \frac{\partial}{\partial a_i^{(1)}} \left(U^T a^{(1)} + b^{(2)} \right) = U_i^T \tag{21}$$

From the derivative of the hyperbolic tangent function found in (8) we have

$$\frac{\partial a_i^{(1)}}{\partial z_i^{(1)}} = \frac{\partial}{\partial z_i^{(1)}} f(z_i^{(1)}) = 1 - f(z_i^{(1)})^2 = 1 - (a_i^{(1)})^2 \tag{22}$$

$$\frac{\partial z_i^{(1)}}{\partial W_{ij}} = \frac{\partial}{\partial W_{ij}} \left(Wx + b^{(1)}\right) = x_j \tag{23}$$

Combining (20), (21), (22) and (23) we end up with

$$\frac{\partial J}{\partial W_{ij}} = \left(a^{(2)} - y\right) U_i^T \left(1 - (a_i^{(1)})^2\right) x_j \tag{24}$$

In order to find the gradient in matrix form we need to introduce new notation. Let $\diamond$ be defined as an element-wise multiplication of two matrices or vectors of the same size. Now we have the full gradient as

$$\frac{\partial J}{\partial W} = \left(a^{(2)} - y\right) U \diamond \left(1 - (a^{(1)})^2\right) x^T \tag{25}$$

To find $\frac{\partial J}{\partial b^{(1)}}$ we apply the same steps we used to arrive at (20)

$$\frac{\partial J}{\partial b_i^{(1)}} = \left(a^{(2)} - y\right) \frac{\partial z^{(2)}}{\partial a_i^{(1)}} \frac{\partial a_i^{(1)}}{\partial z_i^{(1)}} \frac{\partial z_i^{(1)}}{\partial b_i^{(1)}} \tag{26}$$

$$\frac{\partial z_i^{(1)}}{\partial b_i^{(1)}} = \frac{\partial}{\partial b_i^{(1)}} \left(Wx + b^{(1)}\right) = 1 \tag{27}$$

We combine (20), (21), (22) and (27) and find

$$\frac{\partial J}{\partial b^{(1)}} = \left(a^{(2)} - y\right) U \diamond \left(1 - (a^{(1)})^2\right) \tag{28}$$

In order to find the last parameter gradient $\frac{\partial J}{\partial L}$, we find $\frac{\partial J}{\partial x}$ where x is the training example and then map this gradient to its corresponding piece of the $\frac{\partial J}{\partial L}$ gradient. We use the same steps as (20) but this time every element of $a^{(1)}$ and $z^{(1)}$ depend on the i^{th} element of x.

$$\frac{\partial J}{\partial x_i} = \left(a^{(2)} - y\right) \frac{\partial z^{(2)}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial x_i} \tag{29}$$

Letting $W_{\bullet i}$ designate the i^{th} column of W,

$$\frac{\partial z^{(1)}}{\partial x_i} = \frac{\partial}{\partial x_i} \left(Wx + b^{(1)}\right) = W_{\bullet i} \tag{30}$$

Using the same results for the partial derivatives from (21), (22) and the above two equations we have

$$\frac{\partial J}{\partial x_i} = \left(a^{(2)} - y\right) \left[U \diamond \left(1 - (a^{(1)})^2\right)\right]^T W_{\bullet i} \tag{31}$$

To find $\frac{\partial J}{\partial x_i}$ in vector form notice that (31) gives a row vector if we multiply by the full matrix W thus we simply need the transpose of this row vector to arrive at the column vector gradient for x

$$\frac{\partial J}{\partial x} = \left(a^{(2)} - y\right) W^T \left[U \diamond \left(1 - (a^{(1)})^2\right)\right] \tag{32}$$

To see how to map (32) back to $\frac{\partial J}{\partial L}$ notice that x is an nC column vector where n is the vector size of each word and C is the window size or number of context words in each sample. This means we have

C columns of L to update for each training sample. Thus we can write x as a C long vector with each element corresponding to a word vector.

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_C \end{bmatrix}$$

Each of these C n-dimensional vectors, x_i, correspond to a column of L. Thus we can update the columns of L using this breakdown of word vectors of x and the gradient found in (32). In order to do this we must keep track what column each of the word vectors of x correspond to.

Our final derivations for $\frac{\partial J}{\partial b^{(1)}}$, $\frac{\partial J}{\partial b^{(2)}}$ and $\frac{\partial J}{\partial L}$ are given in (28), (18) and (32) respectively since these gradients don't depend on our regularization term. To find the final form of $\frac{\partial J}{\partial W}$ we combine (12) and (25). Similarly to find the final form of $\frac{\partial J}{\partial U}$ we combine (11) and (17).

$$\frac{\partial J}{\partial W} = \left(a^{(2)} - y\right) U \diamond \left(1 - (a^{(1)})^2\right) x^T + CW \tag{33}$$

$$\frac{\partial J}{\partial U} = \left(a^{(2)} - y\right) (a^{(1)})^T + CU \tag{34}$$

This completes the derivation of the parameters to the cost function (10) for our neural network model.

B Softmax and Logistic Regression

We now show that the Softmax classification model with only two classes is equivalent to logistic regression model with a single weight vector. We begin with the general Sofmax model and setting the number of classes to 2 we then derive logistic regression with a single weight vector.

Recall that Sofmax classification is defined as

$$p(y = i|x;\theta) = \phi_i = \frac{e^{\theta_i^T x}}{\sum_{j=1}^{k} \theta_j^T x} \tag{35}$$

Thus the hypothesis function $h_\theta(x)$ should output a k-dimensional vector of probabilities that x belongs to each class as defined in (35) where k represents the number of classes. However, since the probability over all classes must sum to one, we have that $p(y = k|x;\theta) = 1 - \sum_{i=1}^{k-1} p(y = i|x;\theta) = 1 - \sum_{i=1}^{k-1} \phi_i$. Since $p(y = k|x;\theta)$ can be expressed as a function of all other $k-1$ probabilities, our hypothesis function only needs to output the first $k-1$ probabilities.

$$h_\theta(x) = \begin{bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_{k-1} \end{bmatrix} \tag{36}$$

Now we set $k = 2$ since we are considering Softmax with only two classes and we obtain

$$\begin{aligned}
h_\theta(x) &= \phi_1 = \frac{e^{\theta_1^T x}}{\sum_{j=1}^{2} \theta_j^T x} \\
h_\theta(x) &= \frac{e^{\theta_1^T x}}{e^{\theta_1^T x} + e^{\theta_2^T x}} \\
h_\theta(x) &= \frac{e^{\theta_1^T x}}{e^{\theta_1^T x} + e^{\theta_2^T x}} * \frac{\frac{1}{e^{\theta_1^T x}}}{\frac{1}{e^{\theta_1^T x}}} \\
h_\theta(x) &= \frac{1}{1 + \frac{e^{\theta_2^T x}}{e^{\theta_1^T x}}} \\
h_\theta(x) &= \frac{1}{1 + e^{(\theta_2 - \theta_1)^T x}} \\
h_\theta(x) &= \frac{1}{1 + e^{-(\theta_1 - \theta_2)^T x}}
\end{aligned} \tag{37}$$

We see that the final result derived in (37) is exactly equivalent to the hypothesis function for logistic regression with a single weight matrix, namely $\theta = \theta_1 - \theta_2$.

C Word Vector Visualization Using t-SNE

D Collaboration

We both worked on the code and the report together. We pair programmed the entire code base as a team, and wrote the report as a team as well.