CS224N Neural Networks for Name Entity Recognition

Jiayuan Ma
jiayuanm@stanford.edu

Xincheng Zhang
xinchen2@stanford.edu

December 7, 2013

1 Implementation Details

1.1 Matlab Style Code

Because we are comfortable with Matlab style of operating with matrices and vectors and EJML does not support all Matlab-like APIs, we choose to implement some Matlab style matrix helper functions such as `repmat`, `horzcat` in `MatlabAPI`.

We also implement several helper methods such as `sigmoid`, `tanh`, `tanhDerivative` to make our neural network code more modular, e.g. feedforward to be more readable.

1.2 Multiple Layer Support

Since we decide to target Extra Credit at the very beginning, our design and implementation automatically support multiple (≥ 1) hidden layer neural networks (or deep architectures). In addition to the default constructor of `WindowModel` which takes in a single number as the size of one hidden layer, the sizes of multiple hidden layers can be passed in as an array of integers to another "deep" constructor of `WindowModel`. All our implementations of feed forward propagation, back propagation and gradient checking support multiple layer architectures.

1.3 Building Blocks

Feed Forward: The implementation of feed forward function is relatively straightforward: doing iterative matrix multiplications and feeding the result to nonlinear activation functions (tanh or sigmoid depends on which layer we are processing). We also implemented a regularized likelihood cost function to analyze the changes of cost function during training. We implemented these two functions in batch mode, which can take in a batch of inputs, not a single input.

```
Line 235: public SimpleMatrix batchFeedforward(SimpleMatrix windows)
Line 251: public double costFunction(SimpleMatrix X, SimpleMatrix L)
```

Initialization: We initialize all W randomly and set b to zero as instructed. In addition, since we support multiple layer, the $fanIn$ value of each deeper layer is the hidden size of its previous layer.

```
Line 280: public void initWeights()
```

Back Propagation: There are several smaller components for building back propagation algorithm. We first do a feed forward propagation to get $z^{(l)}$ and $a^{(l)}$ (defined in (3)), and then back propagate errors to calculate the gradient. We refer interested readers to Section 2 for more details. The following implementation also supports gradient in batch mode so that we can also experiment with mini-batch gradient descent.

```
Line 313: protected SimpleMatrix[] backpropGrad(SimpleMatrix batch, SimpleMatrix label)
```

Gradient Check: We implemented the numerical gradient calculation in

```
Line 418: protected SimpleMatrix[] numericalGrad(SimpleMatrix batch, SimpleMatrix label)
```

and we use `checkGradient` function to ensure our gradient is calculated correctly (the L_∞ norm between two gradient vectors is below 10^{-8}).

All of these components are implemented to support multiple layers calculation as well as batch mode. We setup an initial run with window size (C) 5, one hidden layer of size (H) 100, iterations through the dataset (K) 20. The F1 score is 0.734.

2 Gradients by Backpropagation (with Extra Credit)

First of all, we "absorb" all bias terms into their corresponding weight matrices. We did this by appending bias terms as the first column of the new weight matrices and padding. Doing so simplifies the problem formulation and derivation of gradients.
For example, the procedure works as follows.

$$Wx + b \qquad \text{is rewritten into} \qquad W'x' = \begin{bmatrix} b & W \end{bmatrix} \begin{bmatrix} 1 \\ x \end{bmatrix} \tag{1}$$

Unless otherwise indicated, the report assumes the weight matrix $W^{(l)}$ of layer l is the absorbed version of the original weight matrix and its corresponding bias term, i.e.

$$\begin{aligned} W^{(l)} &= \begin{bmatrix} b^{(i)} & W^{(l)}_{\text{ori}} \end{bmatrix} \\ U &= \begin{bmatrix} b & U_{\text{ori}} \end{bmatrix} \end{aligned} \tag{2}$$

We derive the gradients of our neural network regardless of how deep the network is. Therefore, the gradients derived here apply to networks of one or multiple hidden layers (extra credit).

For a network that has L hidden layers, we iteratively define the activation vectors $a^{(l)}$ and each layer's nonlinear input vectors $z^{(l)}$ as follows.

$$\begin{aligned} z^{(0)} &= x & a^{(0)} &= \begin{bmatrix} 1 \\ z^{(0)} \end{bmatrix} & &\text{where } x \text{ is the input vector} \\ z^{(l)} &= W^{(l)} a^{(l-1)} & a^{(l)} &= \begin{bmatrix} 1 \\ f(z^{(l)}) \end{bmatrix} & &\text{for } l = 1, \ldots, L \\ z^{(L+1)} &= U a^{(L)} & a^{(L+1)} &= g(z^{(L+1)}) \end{aligned} \tag{3}$$

where f is the nonlinear tanh function and g is the sigmoid function. Notice that the definition of a and z follow closelt the procedure of forward propagation. The final layer of activation $h_\theta(x) = a^{(L+1)}$ is the output of the entire neural network.

Back propagation algorithm works by propagating errors $\delta^{(l)}$ progressively from the output layer to the input layer. Following back propagation, we give the iterative definition of $\delta^{(l)}$ as follows.

$$\begin{aligned}
\delta^{(L+1)} &= h_\theta(x) - y = a^{(L+1)} - y \\
\delta^{(L)} &= U^T \delta^{(L+1)}(2:\text{end}). * \tanh'(z^{(L)}) \\
\delta^{(l)} &= (W^{(l+1)})^T \delta^{(l+1)}(2:\text{end}). * \tanh'(z^{(l)}) \qquad \text{for } l = 1, \ldots, L-1 \\
\delta^{(0)} &= (W^{(1)})^T \delta^{(1)}(2:\text{end})
\end{aligned} \tag{4}$$

where $.*$ is the Matlab notation of element-wise multiplication, $(2:\text{end})$ is the Matlab notation of dropping the first row and $\tanh'$ is the derivative of tanh function, namely $1 - \tanh^2(x)$.

Finally, we can define the gradient of our neural network as follows.

$$\begin{aligned}
\frac{\partial J(\theta)}{\partial U} &= \delta^{(L+1)}(a^{(L)})^T \\
\frac{\partial J(\theta)}{\partial W^{(l)}} &= \delta^{(l)}(a^{(l-1)})^T \qquad \text{for } l = 1, \ldots L \\
\frac{\partial J(\theta)}{\partial L} &= \delta^{(0)}
\end{aligned} \tag{5}$$

Notice that $\frac{\partial J(\theta)}{\partial W^{(l)}}$ is applied to the absorbed weight matrix $W^{(l)}$ which includes the bias terms $b^{(l)}$, so that we don't have a separate $\frac{\partial J(\theta)}{\partial b^{(l)}}$ gradient.

If we add regularization terms to our cost function, the gradient of $\frac{\partial J(\theta)}{\partial U}$ and $\frac{\partial J(\theta)}{\partial W^{(l)}}$ should be updated as follows.

$$\begin{aligned}
\frac{\partial J(\theta)}{\partial U} &= \delta^{(L+1)}(a^{(L)})^T + C \begin{bmatrix} 0 & U_{12} & \ldots & U_{1m} \\ 0 & U_{22} & \ldots & U_{2m} \\ \vdots & \vdots & \ldots & \vdots \\ 0 & U_{n2} & \ldots & U_{nm} \end{bmatrix} \\
\frac{\partial J(\theta)}{\partial W^{(l)}} &= \delta^{(l)}(a^{(l-1)})^T + C \begin{bmatrix} 0 & W_{12} & \ldots & W_{1m} \\ 0 & W_{22} & \ldots & W_{2m} \\ \vdots & \vdots & \ldots & \vdots \\ 0 & W_{n2} & \ldots & W_{nm} \end{bmatrix} \qquad \text{for } l = 1, \ldots L
\end{aligned} \tag{6}$$

Because we don't want to penalize bias terms, we zero out the first column (absorbed bias term) of each weight matrix.

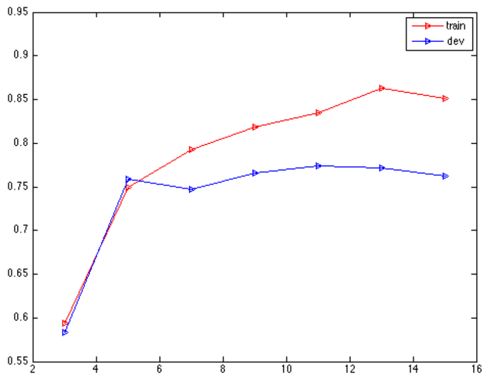
3 Network Analysis

3.1 Parameter Exploration

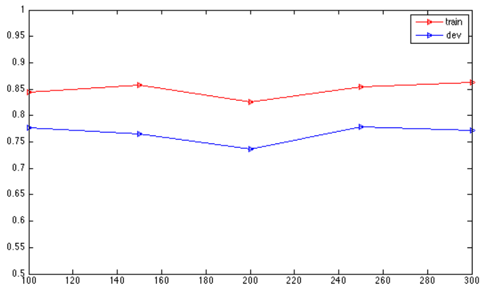
We setup series of parameter setting to explore the parameters of the network.

- Figure 1a: Fix learning rate as 0.001, regularization constant as 0.0001, one hidden layer of size 300 and run 10 iterations. Vary the window size from 3 to 15 with step size 2.

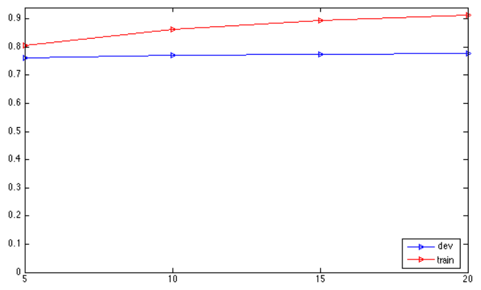
- Figure 1b: Fix learning rate as 0.001, regularization constant as 0.0001, window size 13 and run 10 iterations. Vary the single hidden layer size from 100 to 300 with step size 50.
- Figure 1c: Fix learning rate as 0.001, regularization constant as 0.0001, one hidden layer of size 300 and window size 13. Vary the iteration from 5 to 20 with step size 5.
- Figure 1d: Fix window size as 13, regularization constant as 0.0001, one hidden layer of size 300 and run 10 iterations. Vary the learning rate from 0.0005 to 0.002 with step size 0.0005.



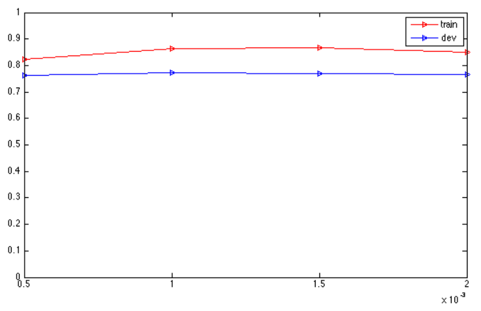
(a) F1 score v.s. Window size C



(b) F1 score v.s. Single hidden layer size H



(c) F1 score v.s. Epochs K



(d) F1 score v.s. Learning Rate α

Figure 1: Network parameter exploration

Based on our experiments, the window size (C) seems to be the most influential parameter. We also observed that we seldom overfit our model throughout our exploration. The F1 score on the dev set stays between 0.75 and 0.77, it does not drop down even we increase our hidden layer size or the iterations. We also plot learning curves in Figure 2 by evaluating the cost functions after doing every 3000 stochastic gradient descent. From this plot, we can convince ourselves that there is no need to go beyond 10 epochs because the cost function has already converged. We summarize some of our best results from these runs in Table 1. We use the optimal parameter configurations to run our model only once on the **test set**, we obtain a F1 score of 0.692 (Precision: 80.44%, Recall: 60.77%).

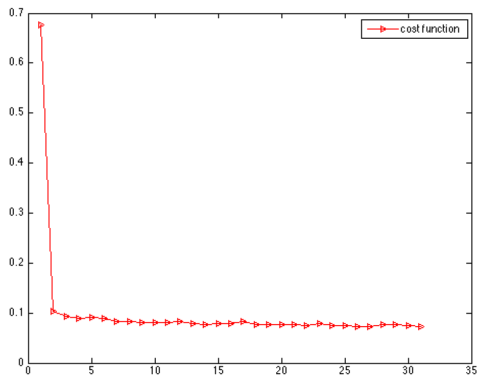


Figure 2: Learning curve obtained by evaluating cost functions every 3000 SGD iterations

Network Parameters	PERSON (training/dev)		
	Precision(%)	Recall (%)	F1 (%)
C=11, H=300, K=10, $\alpha = 0.001$	81.78/77.20	85.14/77.64	83.43/77.42
C=13, H=300, K=5, $\alpha = 0.001$	85.13/83.35	76.60/70.09	80.64/76.14
C=13, H=250, K=10, $\alpha = 0.001$	87.66/81.21	83.43/74.79	**85.49/77.86**

Table 1: F1 scores for experiments of single hidden layer architecture

3.2 Word Vector Visualization

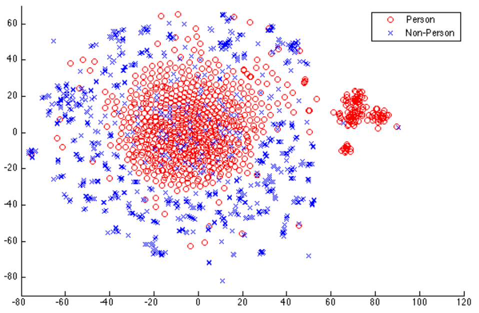
We picked 1000 most frequent words from both PERSON and NON-PERSON set for running tSNE to embed vectors from 50 dimensional space to 2 dimensional space for visualization. The embedding algorithm shows that the initial word vectors actually perform well. Those PERSON words are approximately divided into two clusters with the larger one surrounding by those NON-PERSON words. However, we can still observe that several NON-PERSON words are mixed with those PERSON words. After training, the larger PERSON cluster tends to be more compact than before and there are less NON-PERSON words that are mixed with PERSON words.

3.3 Error Analysis

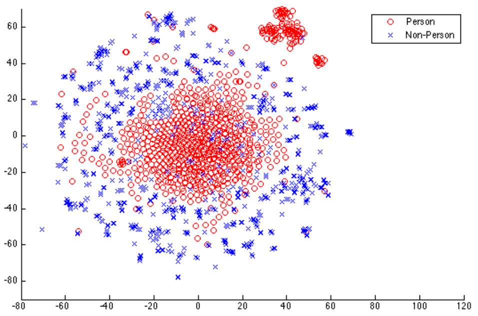
False Positive We observe that the situations where a NON-PERSON word is recognized as a person name can be categorized as follows. The number in the parenthesis is the response score of trained neural network.

- Proper Noun

```
Place: Russia(0.9), Moscow(0.894), England(0.947), Texas(0.944),
Chicago(0.941), China(0.967) ...
```



(a) Before training



(b) After training

Figure 3: Word vector visualization

`Date: Wednesday(0.953), Tuesday(0.996), September(0.897), Saturday(0.835)...`

`Others: Honda(0.982), Mercedes-Benz(0.929), Reuters(0.841), Eurodollars(0.788), Al(0.981), KeyCorp(0.769) ...`

`Numbers: 6-0-40-1(0.941), 9.5-0-44-1(0.97), 42(0.92) ...`

- Pronoun, Determinant or Preposition appear with Name

  `I(0.939), he(0.813), they(0.818), One(0.781), Some(0.643), A(0.726), The(0.818), St(0.709), At(0.973) ...`

- Other

  `Cardinal(0.879), Federal(0.767), Court(0.558), Commerce(0.674) ...`

For those special nouns such as brands or places, they serve a very similar purpose as a PERSON word in a sentence. We also notice some of these words are having a very high response score. It illustrates that unless with adequate domain knowledge of them, it is even hard for a real human to tell if they are a person name or not. As a PERSON name always appears with some determinant or proposition phrases, those phrases have a higher change to be misclassified as PERSON even in some NON-PERSON context.

Generally speaking, we observe that the words that have been misclassified as person names are not random. They share both similar context and word category as person names.

False Negative Similar to false positive, we categorize these personal words not being recognized into:

- Symbol or Proposition

  `'"'(0), '('(0), ')'(0), of(0)`

- Names

```
Clinton(0.001), Mark(0.297), Van(0.015), Albert(0.004), Dole(0.009), ...
```

Due to the fact that those symbols appear more often together with NON-PERSON words, they are not recognized as person names. There are various reasons that can explain why these PERSON names are not being recognized correctly. Some names do not appear in the training set. Some names may serve other purposes in a sentence, e.g. "Mark" may be a verb in many cases, and "Van" may be a valid noun under many cases.

4 Extra Credit and More Experiments

4.1 Multiple Hidden Layers

We did some experiments to make the network deeper by specifying different hidden layer stuctures while fixing the window size (C) to be 13, epoch of iterations (K) 5, learning rate (α) 0.001 and regularization constant (C) 0.0001. Table 2 summarizes results from these experiments. Training

Hidden Layer Size	PERSON (training/dev)		
	Precision(%)	Recall (%)	F1 (%)
100, 100	82.09/80.43	76.47/70.50	79.18/75.14
150, 100	80.83/78.10	79.48/72.82	80.15/75.37
100, 150	78.55/76.41	82.49/76.34	**80.47/76.38**

Table 2: F1 scores for experiments of multiple hidden layers architecture

deep architectures takes longer than training shallow architectures. However, training deep architectures for less number of iterations can give comparable results to shallow architectures trained with more number of iterations.

4.2 Cut Off Point

Observing that our data is not well-balanced, we think it makes sense to play around with the cutoff value of final response of logistic regression. In general, the recall of PERSON category is not so good given that NON-PERSON training instances outnumbers PERSON training instances. Intuitively, we can lower the cutoff value so that the classifier will recall more PERSON instances. Instead of using the fixed 0.5 to determine whether a word is a person name or not, we tried to treat this cutoff as a hyperparameter and tune it on the dev set. In Figure 4, we vary the cutoff point from 0 to 1 with step size 0.05 and find out the best cut off value is just around 0.45 to 0.5.

4.3 Mini-batch Gradient Descent

As mentioned before, our implementation support batch evaluation. We extend stochastic gradient descent to a mini-batch gradient update scheme to see if we can converge to a better result in less number of epochs. In `MinibatchModel`, we extend `WindowModel` class and override the train function to run mini-batch gradient descent.

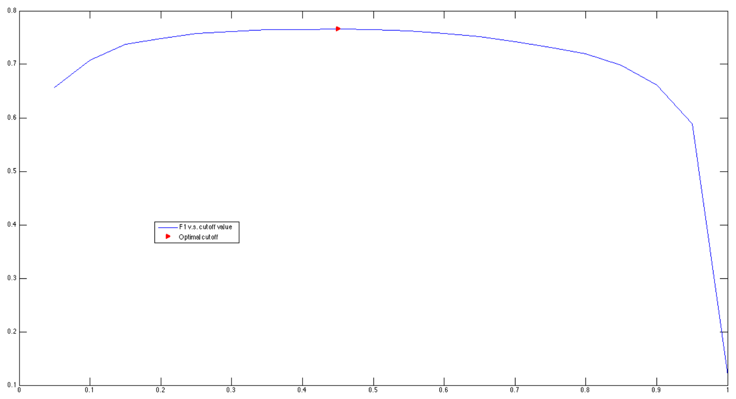


Figure 4: F1 score v.s. cutoff value

4.3.1 Partition

Since there are a lot more NON-PERSON words than PERSON words, we try to partition the data into batches of instances that contain both PERSON words and NON-PERSON words for mini-batch gradient descent.

4.3.2 Batch Gradient

The gradient under batch mode is not so different from the gradient in stochastic gradient descent. The batch gradient should be as follows.

$$\begin{aligned}
\frac{\partial J(\theta)}{\partial U} &= \frac{1}{m}\sum_{i=1}^{m}\frac{\partial J_i(\theta)}{\partial U} + \frac{C}{m}\begin{bmatrix} 0 & U_{12} & \dots & U_{1m} \\ 0 & U_{22} & \dots & U_{2m} \\ \vdots & \vdots & \dots & \vdots \\ 0 & U_{n2} & \dots & U_{nm} \end{bmatrix} \\
\frac{\partial J(\theta)}{\partial W^{(l)}} &= \frac{1}{m}\sum_{i=1}^{m}\frac{\partial J_i(\theta)}{\partial W^{(l)}} + \frac{C}{m}\begin{bmatrix} 0 & W_{12} & \dots & W_{1m} \\ 0 & W_{22} & \dots & W_{2m} \\ \vdots & \vdots & \dots & \vdots \\ 0 & W_{n2} & \dots & W_{nm} \end{bmatrix} \qquad \text{for } l = 1, \dots L \\
\frac{\partial J(\theta)}{\partial L} &= \frac{1}{m}\sum_{i=1}^{m}\frac{\partial J_i(\theta)}{\partial L}
\end{aligned} \tag{7}$$

where m is the batch size and $\frac{\partial J_i(\theta)}{\partial \cdot}$ is the corresponding gradient (defined in (5)) for each instance in the batch.

4.3.3 Result

As discussed with one of TAs during office hour, we conclude that the result of mini-batch gradient descent is not so good. There are two reasons to explain why:

- The imbalance between PERSON and NON-PERSON catergories. Inside each mini-batch, there are more NON-PERSON instances than PERSON ones, which makes batch gradient inaccurate.
- No good way to update L parameter. In batch gradient descent, we input a batch of windows into the network, and all these windows contain different word vectors. However, in (7) we have only one gradient over L. If we use that gradient to update across all different word vectors in the batch, the discrimination power of our model between words is greatly undermined.