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 is relative straight forward by doing the matrix multiplication iteratively, and feed the result to tanh or sigmoid helper function depends on where we progress. We also implemented a cost function adding the regularization to result difference between feed forward and label.

  ```
  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 component for building back propagation algorithm. We first ran a feed forward algorithm to get $z^{(i)}$ and $a^{(i)}$

  Refer the implementation at:

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

- **Gradient Check**: We implemented the numerical gradient calculation in WindowModel.java:

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

All of these components are implemented to support multiple layers calculation. User can pass an array of integers instead of only one integer to specify the hidden layer structure. After we make sure the gradient computing implementation is correct by the check test. We setup an initial run with option window size 5, hidden layer size 100, iteration 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.

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

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.

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

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

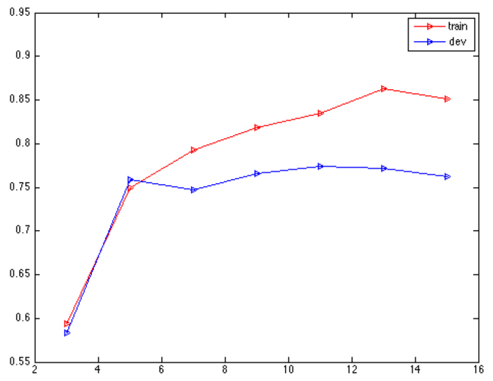
3 Network Analysis

3.1 Parameter Exploration

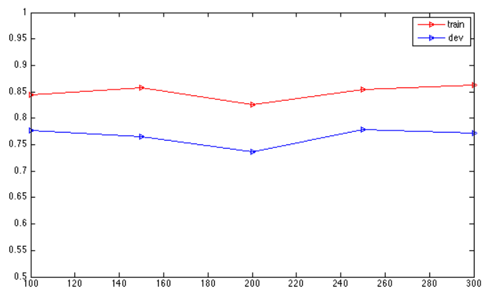
We setup series of parameter setting to explore the network.

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

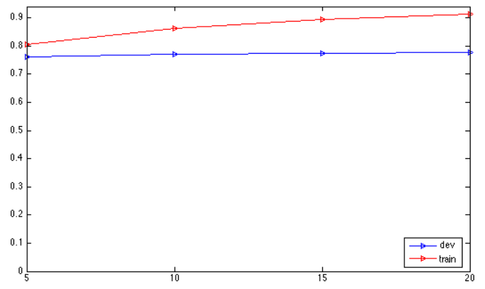
- Figure 1b: Fix learning rate as 0.001, regularization constant as 0.0001, window size 13 and run 10 iterations. Varies the hidden layer size from 100 to 300 with step size 50.
- Figure 1c: Fix learning rate as 0.001, regularization constant as 0.0001, hidden layer size 300 and window size 13. Varies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



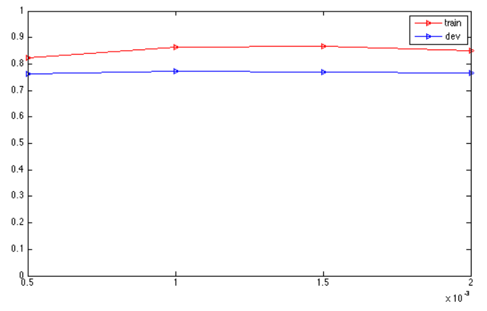
(a) Window Size v.s. F1



(b) Hidden Layer Size v.s. F1



(c) Epoch v.s. F1



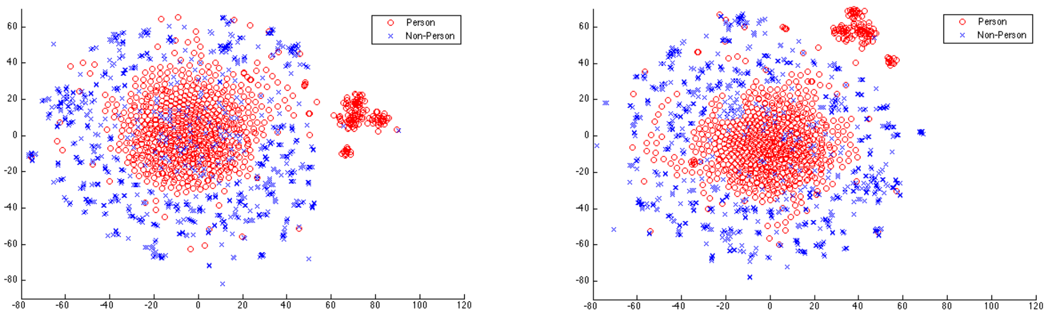
(d) Learning Rate v.s. F1

Figure 1: parameter exploration

The window size seems to be the most influential result according to our observation. We also realized that we never actually overfit the neural network throughout our exploration. The F1 stay at 0.75 0.77, it didn't drop down even we increase our hidden layer size or the iterations.

3.2 Word Cluster

We picked 1000 most frequent words from both person and non-person set for running tSNE to reduce the data dimension to 2 for visualization.



(a) before training (b) after training

Figure 2: word vectors distribution

The unsupervised algorithm give the initial word vectors actually performs well. Those personal words are divided into two clusters with the larger one surrounding by those non-person words. However, we can observe several non-person words are mixed with those person words.
After training, the larger person points cluster tends to be more compact than before.

3.3 Error Analysis

- False Positive: We observe that the non-person words be recognized as a person name can be categorized into:
  - Proper Noun:

    `Place: Russia, Moscow, England, Texas, Chicago, China ...`

    `Date: Wednesday, Tuesday, September, Thursday...`

    `Others: Honda, Mercedes-Benz, Reuters, Eurodollars, T-bonds, KDP, KeyCorp ...`

    `Numbers: 6-0-40-1, 9.5-0-44-1, 42 ...`
  - Pronoun, Determinant or Preposition appear with Name

    `I, he, they, One, Some, A, An, The, St, At ...`
  - Other

    `Cardinal, Federal, Court, Commerce ...`

  For those special nouns such as brand names or place names, they serve a very similar purpose as a personal name in a sentence. We also notice some of these words are having a very high response score. 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 person name always appears with some determinant or proposition words, those words are being misclassified as a person word even in some situations they are not appeared with a name.

Generally speaking, we observe that the words being misclassified as person are not random. They share both similar context and word category as a person name.

- False Negative: similar to false positive, we categorize these personal words not being recognized into:
  - Symbol or Proposition

    `", (, ), of`
  - Names

    `Clinton, Mark, Van, Albert, Dole, ...`

  Due to those symbols are more often appear with some non person name, they are not recognized as person names.

  There are various reason for these names are not being recognized. 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.

4 Extra Credit and More Experiments

4.1 Mini-batch Gradient Descent

We extend the SGD to a mini-batch gradient update scheme to see if we can converge to a better result in less epoch. We extend WindowModel class and override the train function to select a batch of instances.

4.1.1 Partition

Since there are a lot more non-person words than person words, we try to pick up a batch of instances contains both person words and non-person words together for gradient descent. We partition the training data by FIXME.

4.1.2 Result

The result FIXME

4.2 More Parameters Tuning

4.2.1 Multiple Hidden Layer

We did some experiements to make the network deeper by specifying different hidden layer stucture while fixing the window size 13, iteration 20, learning rate 0.001 and regularization cost 0.0001. Following table 1 display results:

Hidden Layer Size	PERSON (training/dev)			NON-PERSON (training/dev)		
	Precision	Recall	F1	Precision	Recall	F1
100, 100	78.07/78.76	74.41/69.91	76.19/74.07	66.06/73.99	67.37/66.51	66.81/70.05
150, 150	79.99/80.80	75.15/69.60	77.50/74.78	65.90/75.83	68.93/66.68	67.38/70.96
200, 200	85.37/84.05	72.39/65.29	78.35/73.49	80.72/85.20	64.69/62.62	71.82/72.19

Table 1: Scores for experiment different multiple hidden layers structure

4.2.2 Cut Off Point

Instead of using fixed cutoff 0.5 for decided if a response indicated whether the window is person or not, we tried to treat this cutoff as a hyper parameter and tune it on the dev set. We vary the

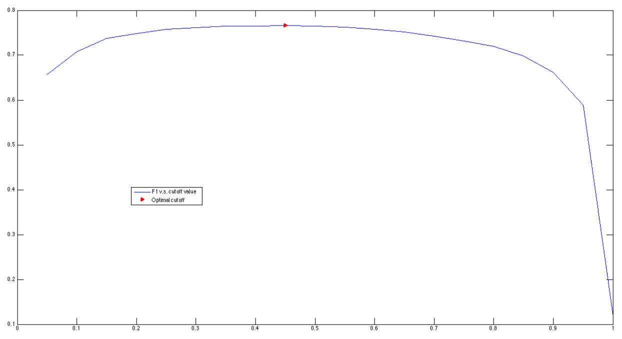


Figure 3: F1 v.s. cutoff value

cutoff value 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.