CS224n Program Assignment 4 Report

Enhao Gong, Yuming Kuang

December 7, 2014

1. Implementation Details and Design Choices

1.1 Baseline

We implemented the simple baseline in *BaselineModel.java,* which takes the named entity at the time of appearance of each word in the training data and use exact string match for the test data. The result is shown in Table 1.

1.2 Window Model (1-layer NN)

We implemented the 1-layer neural netword window model in *WindowModel.java.* We'd like to note the following details in the implementation.

- Weights and partial result: We treat weights L, W, $b^{(1)}$, U, $b^{(2)}$ and partial result $h = \tanh(WL_{cur} + b^{(1)})$ and $p = softmax(Uh + b^{(2)})$ (where L_{cur} is the word vector correponding to current window, $softmax$ refers to the softmax function) as element variables of the class, thus we don't have to pass them between functions and manage them easily.
- Vocabulary: When finding the corresponding L vector for a window, if the word is not in the vocabulary, we check whether it can be interpreted as a number. If so, we give the L corresponding to NNNUMMM to it, otherwise we give the L corresponding to UUUKNNN.
- Feed forward: This is implemented in function *feedForward*, computing h and p for a sample given the weights. To compute tanh efficiently, we use the function *FastMath.tanh.*
- Cost function: Implemented in functions *singleCost* (computing cost for one sample), *costFunction* (computing cost for a given batch of windows plus regularization term), *allCostFunction* (computing cost for the whole data set plus regularization term).
- Gradient: We first give the gradient expression (including regularization term, take $\lambda = 0$ to exclude it). Given the prediction probability vector $p_\theta(x^{(i)}) \in R^{K\times 1}$ and true label vector $y^{(i)} \in R^{K\times 1}$ of sample i, define the error propagated to each layer

$$\begin{aligned} \delta_2^{(i)} &= p_\theta(x^{(i)}) - y^{(i)} \\ \delta_1^{(i)} &= U^T\delta_2^{(i)}. \times (1 - h_{(i)}^2) \end{aligned}$$

Then given a batch of m samples, we can express the gradients as

$$\begin{aligned} \frac{\partial J_R}{\partial U} &= \frac{1}{m}\sum_{i=1}^{m}\delta_2^{(i)}h_{(i)}^T + \frac{\lambda}{m}U \\ \frac{\partial J_R}{\partial b^{(2)}} &= \frac{1}{m}\sum_{i=1}^{m}\delta_2^{(i)} \\ \frac{\partial J_R}{\partial W} &= \frac{1}{m}\sum_{i=1}^{m}\delta_1^{(i)}L_{(i)}^T + \frac{\lambda}{m}W \\ \frac{\partial J_R}{\partial b^{(1)}} &= \frac{1}{m}\sum_{i=1}^{m}\delta_1^{(i)} \\ \frac{\partial J_R}{\partial L} &= \frac{1}{m}\sum_{i=1}^{m}W^T\delta_1^{(i)} \end{aligned}$$

We implement the gradient part in function *gradient*, which takes a batch of samples and computes the gradient. In order to be efficient, we store $\frac{\partial J}{\partial L}$ in dL as a map that only includes the indice of word vectors which need to be updated and the corresponding gradient. The gradient update is implemented in *updateGradient*, in which we update L according the map dL.

- Gradient check: Implemented in function *checkGradient*, which takes a batch of samples, computes the numerical gradient for each weight element and compare with the derived gradient. The criterion we use is l_∞ norm, i.e the max absolute value of difference of gradients is less than 1e-7. At the beginning of training, we random select 10 samples to do gradient check. Our implemention successfully passed the gradient check.

- SGD training: Implemented in function *train*. At each iteration of the data set, it iteratively goes through each sample, computes the gradient and do the weight update. After each iteration, we compute the cost function of all the samples given current weights. The stop criterion is that the cost function change is less than $tol = 1e-4$ or reach max iteration number *maxIter*. So far because the training takes a long time to converge, usually it's stopped by *maxIter*.

2. Experients on Tuning

We did parameter tuning for the Window Model on *window size (C)*, *hidden layer size (H)*, *learning rate (α)*, *epoch (T=maxIter)*. The FB1 score plot is shown in Figure 1 for both *train* and *dev* set.

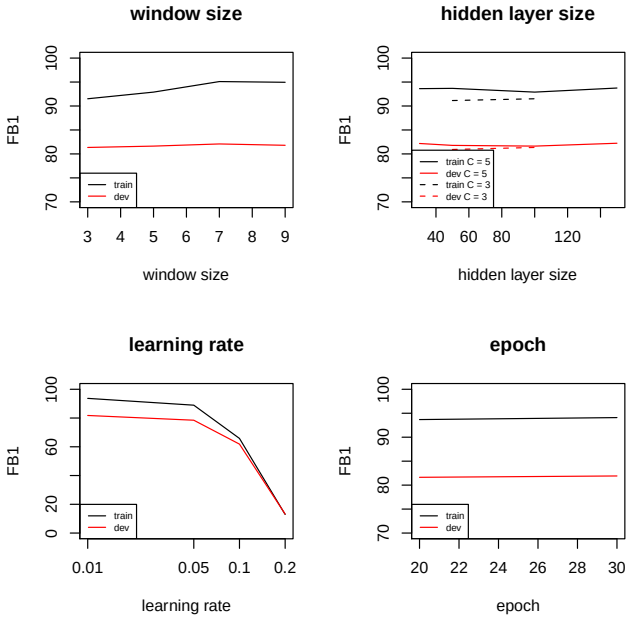

Figure 1: Parameter tuning results

- Window size (C): Fix $H = 100, T = 20, \alpha = 0.01, \lambda = 1e-4$, we ran the model for $C = 3, 5, 7, 9$. The result shows that the FB1 score increases in training set when C gets large, because the model becomes more complex. There is also increase of FB1 score in dev set, but only a little.

- Hidden layer size (H): Fix $T = 20, \alpha = 0.01, \lambda = 1e-4$, for $C = 5$, we ran with $H = 30, 50, 100, 150$. For $C = 3$, we ran with $H = 50, 100$. The result shows that FB1 score keeps quite flat in both training and dev set. This might because the fixed epoch limits the amount of 'information' we can learn from the training data. For more complicated model we need to train for longer time.

- Learning rate (α): Fix $C = 5, H = 50, T = 20, \lambda = 1e-4$, we ran with $\alpha = 0.01, 0.05, 0.1, 0.2$. The result shows that for our setting, larger α cause the performance in both training and dev set to drop dramatically. This is because when α gets large, the step of each gradient descent becomes large, and we are not falling into the minimal point of cost function but jumping and vacillate around. Here on other hand, if we choose α too small, we are learning too slow and takes longer iterations to converge. The minimal α we tried in our experiment doesn't fall into that region.

- Epoch (T): Fix $C = 5, H = 50, \alpha = 0.01, \lambda = 1e-4$, we ran with $T = 20, 30$. The result shows better performance in both training and dev set for larget T, but not very significant. Certainly we gain by training for longer time, but the amount quickly becomes marginal. This is because the neural network model are complicated and usually takes a long time to converge. Also although we can't train that many times due to time limit, training too many times for NN model can lead to overfitting as well.

With the experiments above, and also considering the runtime limit, we finalize our hyperparameter to be $C = 5, H = 150, T = 20, \alpha = 0.01, \lambda = 1e-4$. This setting leads to the best FB1 score on dev set among all the settings listed above.

3 Result and Analysis

3.1 Result table

We ran 4 models on test data set: Baseline, 1-layer NN with parameter setting tuned in Section 2, 1-layer NN with word vector randomly initialized and the same parameter setting, 2-layer NN (extra credit). The result summary is in Table 1. We can see that 1-layer NN and 2-layer NN model with provided word vector have similar performance and both outperform the 1-layer NN with random initialized word vector. All the 3 NN models outperform the baseline.

	train/dev/test (%)		
	precision	recall	FB1
baseline	88.69 / 85.23 / 79.81	88.91 / 67.19 / 56.32	88.80 / 75.14 / 66.04
1-layer NN	**95.54** / **86.58** / **81.43**	**92.02** / 78.30 / 72.76	**93.75** / **82.23** / 76.85
1-layer NN (randomly initialized word vector)	91.76 / 82.92 / 77.77	86.06 / 73.29 / 67.25	88.82 / 77.81 / 72.13
2-layer NN (extra credit)	94.55 / 86.01 / 80.54	91.18 / **78.33** / **73.71**	92.84 / 81.99 / **76.97**

Table 1: Result summary

3.2 Comparison for randomly initialized word vector

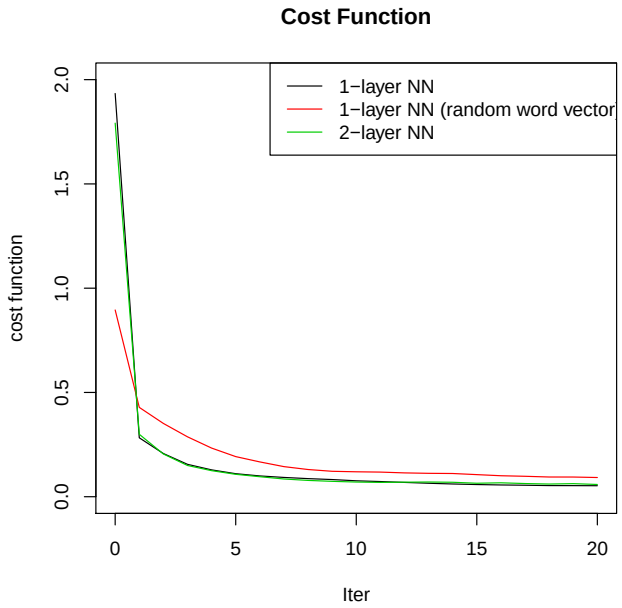

Figure 2: Cost function for 1-layer NN, 1-layer NN (randomly inialized word vector) and 2-layer NN

- From Table 1, we see that with randomly initialized word vector, the performance of 1-layer NN drop dramatically. The reason is that the cost function of neural network is non-convex, so optimizing NN usually leads to a local minimum. Different initialization of the weights will give different local minimum after optimization. In our case, the local minimum of random word vector initialization is worse than the local minimum of provided word vector initialization, thus gives a worse model.

- To give an evidence, Figure 2 plot the cost function v.s iteration time, we see that the cost function of randomly initialized word vector stays higher than the cost function using provided word vector after 3 iterations. This illustrates that the neural network learns less from training using the randomly initialized word vector.
- The provided word vector is trained with other unsupervised data, thus generally gives a better presentation of the words than random initialization, and leads to a better neural network as well.

3.3 Error Analysis of 1-layer NN Model

First we give the per label type result of 1-layer NN model in Table 2 and make the following comments:

- The model performs well for LOC and PER.
- For MISC, the training score is quite high, but performance drops dramatically in dev and test set. This implies that the model is overfitting MISC type, i.e some MISC pattern fitted in training set might not appear in dev and test set. Because the training set has least number of MISC entities, which is not enough to make the model generalized. And note that in test set, the recall is higher than the precision, meaning the model is assigning many O to be MISC. This is because many MISC entities consist of normal words, which can be also used as O. So the model becomes uncertain on these words. Here are two examples in the test set.

```
Uzbek       MISC       O
advancing   O          MISC
```

- For ORG, performance is comparable to the other three type in all train, dev and test set. The reason is that ORG entities can behave similarly to both PER and LOC entities, so the model cannot figure out the ambiguity. Here are some examples.

```
Naoki   PER     ORG
Soma    PER     ORG
FIFA    ORG     LOC
```

	train/dev/test (%)		
	precision	recall	FB1
Overall	95.54 / 86.58 / 81.43	92.02 / 78.30 / 72.76	93.75 / 82.23 / 76.85
LOC	97.39 / 90.26 / 87.93	95.66 / 85.00 / 86.75	96.52 / 87.56 / 84.09
MISC	92.02 / 80.79 / 67.18	94.10 / 74.29 / 70.70	93.05 / 77.40 / 68.90
ORG	94.21 / 78.75 / 76.51	82.94 / 64.29 / 60.82	88.22 / 70.79 / 67.77
PER	96.72 / 90.97 / 86.19	96.64 / 84.76 / 78.76	96.68 / 84.75 / 82.31

Table 2: Per label type result of 1-layer NN Model

Also we want to note the following error cases:

- Some words might appear in multiple type of named entities. The rare type for these words is hard to predict. The following two words are usually used as LOC, but in this rare case they are PER and MISC.

```
CHINA   PER     LOC
Soviet  MISC    LOC
```

- Our model do prediction for each word each time. But for words in an expression, they usually share the same named entity type, and it might be better to deal with them at the same time. The following is an example of LOC expression.

```
United  LOC     ORG
Arab    LOC     LOC
```

- For weird numbers and symbols, our model just treats as UNKNOWN words which also contains a lot of PER. It's better to find a way to determine weird numbers and symbols and deal with them seperately. Here is a quick example of misspecifying weird symbol to be PER.

```
8-1-60-0     O     PER
```

3.4 Further improvement

- Consider the capitalization as a feature. It might be helpful to distinguish between MISC and O.
- Make use of previous/future predicted labels as features, so that we can better deal with expression case.

4 Extra Credit

4.1 Deeper Neural Network

- Implementation: We implemented the 2-layer NN in *twoLayerModel.java.* It's basically similar to the *WindowModel.java,* the only thing different is that we now have an extra layer. So we made the corresponding change to include the extra layer. The partial result computed in feed forward is $h_1 = \tanh(W^{(1)}L_{cur} + b^{(1)})$, $h_2 = \tanh(W^{(2)}h^{(1)} + b^{(2)})$ and $p = softmax(Uh^{(2)} + b^{(3)})$. For the gradient, we can define $\delta_3^{(i)} = p_\theta(x^{(i)}) - y^{(i)}$, $\delta_2^{(i)} = U^T\delta_3^{(i)}. \times (1 - (h_2^{(i)})^2)$ and $\delta_1^{(i)} = W^{(2)}\delta_2^{(i)}. \times (1 - (h_1^{(i)})^2)$ as the propagated error to each layer, then we can express the gradient as (omitted $b^{(j)}$'s due to similarity):

$$\begin{aligned}
\frac{\partial J_R}{\partial U} &= \frac{1}{m}\sum_{i=1}^{m}\delta_3^{(i)}(h_2^{(i)})^T + \frac{\lambda}{m}U \\
\frac{\partial J_R}{\partial W^{(2)}} &= \frac{1}{m}\sum_{i=1}^{m}\delta_2^{(i)}(h_1^{(i)})^T + \frac{\lambda}{m}W^{(2)} \\
\frac{\partial J_R}{\partial W^{(1)}} &= \frac{1}{m}\sum_{i=1}^{m}\delta_1^{(i)}(L_{(i)})^T + \frac{\lambda}{m}W^{(1)} \\
\frac{\partial J_R}{\partial L} &= \frac{1}{m}\sum_{i=1}^{m}(W^{(1)})^T\delta_1^{(i)}
\end{aligned}$$

- Tuning: We tried the following 3 settings on hidden layer sizes. The result is shown in Table 3. We can see that generally with more complex model setting, we get higher perfermance. This is because larger layer sizes give the model more powerful in representation thus in learning. By tuning, we finalize our setting to be $C = 5, H_1 = 150, H_2 = 100, T = 20, \alpha = 0.01, \lambda = 1e-4$.

	train/dev (%)		
	precision	recall	FB1
$H_1 = 100, H_2 = 50$	**95.37** / **87.62**	89.26 / 76.26	92.22 / 81.55
$H_1 = 100, H_2 = 100$	94.17 / 85.43	91.07 / 78.03	92.59 / 81.56
$H_1 = 150, H_2 = 100$	94.55 / 86.01	**91.18** / **78.33**	**92.84** / **81.99**

Table 3: Tuning 2-layer NN result

- Compare with 1-layer NN model: In Table 1 and Figure 2, we give the performance of 2-layer NN model in train/dev/test set, as well as the decreasing process of cost function during training. Compared with the result of 1-layer NN model, we can see that they basically give similar performance, because two models share the same iteration number thus learn similar amount of 'information'. This can be verified by that 1-layer and 2-layer NN models have almost the same cost function decreasing curve. To fully utilize the representation power of deeper neural network, we may need more data and longer time to train.
- Error Analysis: As it has similar performance with 1-layer NN model, we also find that it also has similar problems as summarized in Section 3.3 by looking at the result of test set. We want to note an improvement that with 2-layer, the model gains more power to detect expressions, although it still sometimes predicts the wrong label for the whole expression. Here are two examples.

```
Yasuto  PER     PER
Honda   PER     PER
United  LOC     ORG
Arab    LOC     ORG
```

	train/dev (%)		
	precision	recall	FB1
provided word vector	93.61 / 86.40	88.80 / 76.11	91.14 / 80.93
word2vec	87.98 / 79.07	86.56 / 74.65	87.26 / 76.80

Table 4: Results of Comparing Word Vectors

4.2 Comparing Word Vectors

- Here we implement the word2vec (https://code.google.com/p/word2vec/). Using python, we constructed the word vector file for the words in vocab.txt into a new wordvector file based on the word2vec model trained using its provided corpus text8. Here compares the results of the same methods and parameters ($n = 50, C = 3, T = 20, H = 50, \alpha = 0.01, \lambda = 1e-4$) using different sets of word vectors. The result is summarized in Table 4.

- From the results, the provided word vectors result in better prediction for both training and dev dataset. Word2vec is a State-of-Art word representation especially for comparing word distances. It can help the recall since it quantifies the similarity between words of the same named entity. Such as it results in higher recall in PER (97.69 / 85.33 v.s 95.73 / 83.90 (provided)) due to that names have high similarities.

```
model.most_similar(positive=['peter'])
Out[25]:  [(u'paul', 0.8712412714958191),  (u'nicholas', 0.7892143726348877),
```

- However, it is not designed for NER and there are some limitations. Thus it might not provide a good representation of word for NER task. One example of error is it cannot predict a 'Road' to be Location.

4.3 Word Vector Visualization

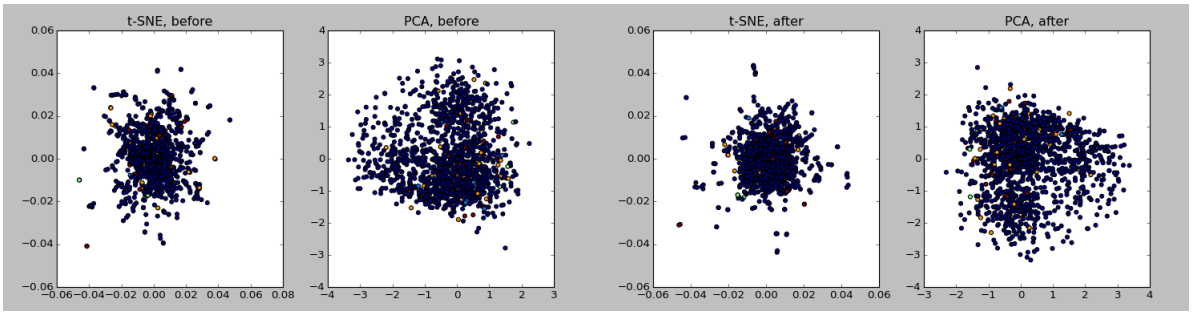

Figure 3: t-SNE and PCA word vector visualization

- Here we used t-SNE algorithm to map the 50-dimensional word vectors onto a 2D space. In addition, a PCA plot is also provided. Figure 3 shows the projected word vectors before and after training process.
- Since there are so many O words, it's hard to see the result of other label words. But clearly we see that the O words are more concentrated after training. Besides we can see a little trend that words with different labels are seperated. The trend is not very clear because we just did 20 iterations and it's far before convergence.