Neural Networks for Named Entity Recognition

Benjamin Au (bau227) & Chentai Kao (chentai)

Abstract

We implement a deep feed-forward neural network to predict the label of words. We use a trigram and 5-gram language model to train our neural network from labeled sentences from our corpus. Our neural network uses a tanh hidden layer and a softmax output layer with an output node size of 5. We implement neural network practice practice design such as small random initialization of weights and adding a regularized cost function. We run manual gradient checking to ensure that our backpropagation is correct. We then perform 3 phases of error analysis to find the optimal set of parameters. Our optimized neural net has one hidden layer of 50 nodes, performs learning on input parameters, has lambda of 0.001 and alpha of 0.0005 and word window size of 5. By running stochastic gradient descent across 100 epochs, our neural network achieves an F1 score of 81.43. We then completely randomize our inputs, instead of using the provided 50-d vocabulary inputs. This further improves our optimized neural network to performance by about 6 points. The best F1 we achieved is 84.52.

Data Pre-Processing

Baseline

In the baseline model, we use a map to store all pairs of <word, label> in the training data. Prediction can be made by a quick lookup from the map. The word must be a case-sensitive exact match. Even with this simple model we obtained F1 as high as 74.52%.

We preprocess the training data to a list of sentences. Boundary of sentence is determined by empty lines in the data, where we pad with a <s> and </s> to mark the beginning and end of the sentence. Since the each datum is a tuple of <text, label>, what we mean by sentence is a sequence of Datum objects.

How we pad the sentence with <s> (and </s>) is related to how we interpret the window size. Given window size 5 and a sentence [<s>, w1, w2, w3, w4, …, </s>], there are two interpretation regarding the center word and its context.

- Start from w1, and pad the beginning sentence with excessive <s>. In this case, the first word vector becomes [<s>, <s>, **w1**, w2, w3].
- Start from w2 without padding any <s>. The first word vector becomes [<s>, w1, **w2**, w3, w4]

Both choices have their own strength. The first choice avoid sacrificing boundary samples, while the second choices ensures each word has better context that is composed of words instead of paddings. We tried both, with the first approach giving F1 = 54.34, while the second way gives F1 = 52.18. Since there is no significant difference, we choose the first way, ensuring that more data is processed.

Our code is adaptive to any window size, as long as the window size is an odd number. In our experiments, we tried on window size 3 and 5. We didn't use larger numbers since the the way we design it would sacrifice some boundary words. If the window size is too large, then we would lose a large portion of samples, which is a waste of data.

Since the labels is a fixed set of [O, LOC, MISC, ORG, PER], we use a map to look up its corresponding y value. For example, the entry "O" gives the y vector [1, 0, 0, 0, 0]. Having a lookup is fast and justified because we just need a lookup from label to y value, therefore avoiding the need to calculate y for every sample.

Before each iteration in the training session, we shuffle all sentences to further randomize the training sequence.

Mapping Words to Word Vectors

We use a SimpleMatrix object to store all word vectors. Another possibility would be using a HashMap that maps each word to its feature vector. If SimpleMatrix is implemented as an array, then both choices have equal access time O(1). Therefore, we pertain to its original design, using SimpleMatrix.

Since we chose SimpleMatrix to lookup word feature, we also need a mapping from word to its matrix index. We use a HashMap for that. Here the entry is case-insensitive, where all words are converted to lowercase. There are some special cases listed as follows.

- If the word is not found in the vocabulary, use the special word "UUUNKKK". While this generalizes the model, it inevitably reduces the difference between all unknown words, since now they're all mapped to the same vector. One possible solution is expanding the vocabulary dynamically whenever a new unknown word is encountered.
- If the word is a number, consisting purely of digits and period, then each digit is replaced with a "DG". Although this may lose some variance among numbers, this catches common formats such as year "DGDGDGDG" or decimals "DG.DGDG". This concept can be further extended to enclose any combination of digits and punctuation. As a result, date format like mm/dd/yyyy can be captured.

Model

Word Model

- Each word is represented by 50-dimension float input vector
- If we use a trigram model (window size = 3), x vector becomes 150-D, concatenated by 3 word vectors pulled from the vocabulary matrix. We can also use any n-gram model. Those experiment are listed in the section describing our grid search for the best result.

NN Model

- Our deep neural net has one hidden layer. The hidden layer has anywhere between 30 and 200 nodes. The output consists of five nodes.
- Activation functions: linear for the input layer, tanh for hidden layer, and softmax for output layer. The derivation of feed forward and back propagation is shown in other sections.
- Following the suggestion on the assignment, we use equation (7) on it to initialize the weight matrix W and U. This considers the fan-in and fan-out of each layer to give random value.

Implementation

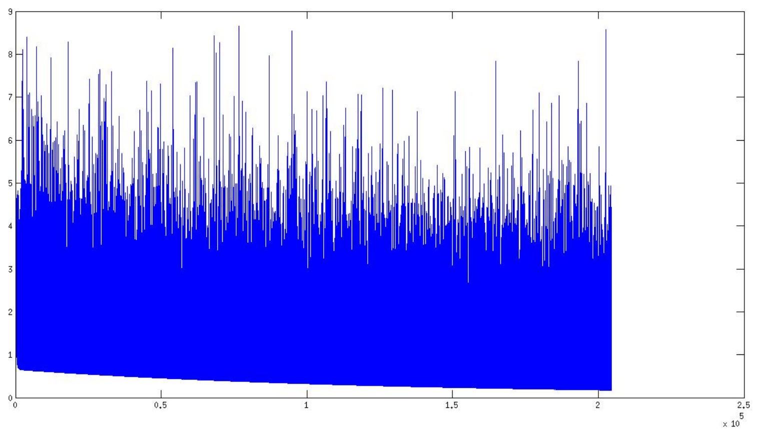
Design Decisions

There are several intermediate variables we need to calculate in feed forward. For example, p, q, h, z, b1, b2, delta1, and delta2. Each iteration of feed forward will update all of them, which are stored as private variables within WindowModel. This cache is necessary because these variables will be reused multiple times in back propagation.

To evaluate the effect of parameters, we parameterized the main function to take in values of parameters (lambda, alpha, hidden layer size, window size, epochs) from the command line. This allows for parallel processing by running multiple instances with different configuration at once, increasing the speed of grid search for the best combination of parameters.

Our program generates output files corresponding to different combination of parameters, preventing them from polluting others' output.

To visualize how the error decreases along with SGD, we output the cost value after every run of SGD, and then visualize it with Matlab. From the result below, we can see that, though the cost change is jagged, there is a steady trend of decreasing error. This is consistent with our expectation of SGD. See Figure 1.



]

Figure 1. Training Error during Stochastic Gradient Descent

Gradient Check

Our gradient check is executed in function gradientCheck() of WindowModel.java.

This confirms the correctness of our error propagation for each paramater in the model. Fore the sake of brevity, we do not include a derivation for each error propagation parameter derivation. We show the final closed form for dJ/dL in Figure 2.

$$\frac{\delta}{\delta L}J(\theta) = [\frac{1}{m}W^T(U^T\delta^{(2)}x(1-tanh^2(wx^{(i)}+b^{(1)})))$$

where $\delta^{(2)} = p_{\theta,j}(x^{(i)}) - y^{(i)}$

Figure 2. Error propagation derivation for dJ/dL

We run gradient check on 10 input examples. This is done by having a counter that is incremented by each gradient check.
Below is the output of one of the samples. We can see that the difference between the norm of our gradient function and the approximation is nearly zero. The check result verified our correct implementation of gradient function. See Figure 3.

```
gradient check U: -9.540634948734805E-11
gradient check W: 6.59282939352579E-9
gradient check b1: 3.3746700989212286E-9
gradient check b2: 1.1385559162135905E-11
gradient check L: 6.374856198476664E-11
```

Figure 3. Sample output of gradient check

Error Analysis

For performance, we chose to measure F1 score.

Once we confirm the correctness of our feedforward and backpropagation implementation, we perform phase one of parameter tuning by running grid search on lambda E {0.01, 0.005, 0.001}, alpha E {0.0005, 0.0001, 0.0005} and hiddenLayerSize E {50, 100, 150, 200}. For grid search we kept constant the following parameters: learnedInput = yes; numEpochs = 10; windowSize = 5. We tune these parameters after the basic grid search.

The search for the optimal lambda parameter showed significant disparity in results. This performance disparity is shown in Figure 4. It is clear that Lambda = 0.001 is better-performing value of lambda.

Lambda = 0.001; F1 = [54.39 - 69.21]
Lambda = 0.01; F1 = [49.36 - 54.86]
Lambda = 0.05; F1 = [26.97 - 33.20]

Figure 4. Range of F1 scores across lambda values in phase 1 error analysis

Given this optimal value of Lambda = 0.001, we show the 3 x 4 grid search for alpha X hiddenLayerSize in Figure 5.

	alpha		
hiddenLayerSize	*0.00005*	*0.00001*	*0.0005*
50	54.38	59.62	**69.21**
100	55.02	59.69	**68.44**
150	56.14	59.72	68.1
200	55.77	59.84	68.24

Figure 5. Results from grid search on alpha X hiddenLayerSize. Lambda = 0.0001; inputParameterLearning = Yes; windowSize = 5; numEpochs = 10

From this grid search, we find top-performers in: alpha = 0.0005, with hiddenLayerSize = 50 and 100.

From pre-grid search parameter exploration, we had found that in general, windowSize = 5 performed better than windowSize = 3. And learning inputs was better than not learning inputs. Once we tuned alpha, and lambda, we confirmed this in phase 2 of error analysis. The results are shown in Figure 6.

	windowSize; learnInputs		
hiddenSize	*5; yes*	*5; no*	*3; no*
50	**69.21**	63.54	65.63
100	68.44	**68.6**	65.94
150	68.1	68.27	65.22
200	68.24	67.84	65.73

Figure 6. Error analysis of windowSize E {3,5]; learnInputs = {no, yes}

As expected windowSize of 5 performed substantially better than windowSize of 3, by about 4-5%. (Implementation note: to analyze windowSize of three, we threw out the first and last word of each sentence, so that each word would only have 1 special sentence tag on each end.) However, not learning inputs seemed to perform quite well for hidden size. Given the similar performance of input learning and non-input learning, we took the best parameter set from each and boosted the number of epochs to 100 to find optimal performance for each. The final results are in Figure 7.

	numHidden; inputLearning	
numEpochs	*50; yes*	100; no
10	69.21	68.6
70	80.01	67.63
100	**81.43**	80.75

Figure 7. Final performance of high-performers from Error Analysis Phase 2. Commons Parameters: alpha = 0.0005; lamba = 0.001; windowSize = 5; usedGivenInputs

As shown in Figure 7, our final neural network obtains an F1 score of 81.43. The tuned parameters are: *alpha = 0.0005; lamba = 0.001; windowSize = 5; usedGivenInputs=yes; learnInputs = yes; numHiddenNodes = 50.*

Error analysis demonstrated that parameter tuning provided massive boosts in performance. Poorly tuned neural networks suffered from F1 performance as low as 26.97. Our final optimally tuned neural network had performance of 81.43, almost 55 points higher.

Comparison to Random Initialization of Word Vectors

Additionally, we perform error analysis on comparing our given 50-d word vector inputs relative to a random 50-d initialization of inputs. As expected, we find our given 50-d vector inputs substantially improved relative to the random baseline. See Figure 7.

	Initialization Strategy	
numEpochs	*Use Given Inputs*	*Randomize Inputs*
10	*69.21*	*75.16*
100	*81.43*	*84.52*

Figure 7. Error analysis of given inputs to random initialization.
Parameter set: lambda = 0.001; alpha = 0.0005; hiddenSize = 50; windowSize = 5; numEpochs = 5

Future Work

With more time, we would have explored neural networks with 2 hidden layers. Also, to avoid being caught in local maximum, several runs with random initialization could be used to select the best result.