Neural Networks for Named Entity Recognition – CS224n Final Project

By Patrick Manion and Daoying Lin

1. Introduction

For the final project, we explored using a neural network to classify which of five named entities a word represented: a location (*LOC*), an organization (*ORG*), a person (*PER*), a miscellaneous entity (*MISC*), or not any named entity (*O*).

The neural network input was a representation of all the words in a window around the target word. If we had a sentence *the boy ran* and wanted to predict *boy* with a size 3 window, the neural network would receive a vector representation of *the*, *boy*, and *ran* combined together into a single input vector. Windows that expanded beyond the start of a sentence were padded with a <s> tag, and windows beyond the end of a sentence were padded with </s> tag. Every word was represented by a 50-dimensional word vector (pre-trained in an unsupervised manner), and a special word vector UUUNKKK was included for any unknown or new words.

For training, we used a cross-entropy error function with a penalty for the size of the weight vectors. All hidden layers used a hyperbolic tangent activation function while the final output layer used a softmax activation function. The network included 5 separate output nodes, and the node with the highest output activation was used as the predicted named entity class.

2. System Design

In designing our system, we wanted to enable fast experimentation and decouple the complexity of the project as much as possible.

In order to decouple the complexity, we built numerous classes that could be separately implemented. *WindowModel* was the primary model class that wrapped up all of our tools into a train and test method. This class also prepared all of the inputs to send to the *NeuralNetwork* class for training. This allowed our neural network to only be considered with the numeric scoring, back propagation calculations, random initialization, and gradient updates, and not with any domain specific information for named entity recognition.

To support these primary classes, we also created a *Document* and *DocumentSet* classes that split the large training set of *Datum* into the right granularity for our window model, which we ended up keeping at the sentence level. Then, each sentence was passed into a *WordWindow* class that handled adding the relevant start and end tags and then efficiently rolling through each window in the sentence. This class also heavily leveraged a *WordMap* class that handled the conversion between words, word ID's, and the word vectors associated with each word.

For fast experimentation, we first knew we would need a central place to set and store all settings. We created a central *Configuration* class that houses every setting that we were interested in testing from the path to the vocabulary file to the number and size of the hidden layers. This class can take a string of key-value pairs so we could directly pass command line arguments into *Configuration*.

We also recognized running tests one-by-one can be burdensome, so we created a *TestConfigReader* that allowed us to write YAML-like files that contained multiple different test configurations. We also built a *CoNLLEval* class to automatically run the CoNLL evaluation tool and output results while the model was training.

In order to leverage these classes, we also created two main methods to run the program. *Launcher* is used to run a single instance of the network and can take command line arguments to be passed to *Configuration*. We also created a *TestConfigLauncher* that can take in a test configuration file and output directory and then run every test while storing the relevant outputs.

Finally, we also implemented unit tests for the majority of our classes and methods. This allowed us to be confident that the functionality of our helper classes were all correct in isolation, which also made us confident the final network with all the components was also running correctly.

3. Gradient and Gradient Derivation

It can be shown that the expression for $\frac{\partial J(\theta)}{\partial L}$ is:

$$\frac{\partial J(\theta)}{\partial L} = W^T U^T (p_\theta - y) \odot tanh'(Wx + b^{(1)})$$

.

We also generalized the gradient expression for multiple layers of neural network, which is summarized next.

Let w^l_{jk} denote the weight for connecting the k^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron in the l^{th} layer; b^l_j denote the bias for the j^{th} neuron in the l^{th} layer; a^l_j denote the activation of the j^{th} neuron in the l^{th} layer; z^l_j denote the weighted input to the j^{th} neuron in the l^{th} layer; $h_l(.)$ denote the activation function for the weighted input $\mathbf{z}_l$. Note that $z^l_j = \sum_i w^l_{ji} a^{l-1}_i + b^l_j$ and $a^l_j = h_l(z^l_j)$. Let's define $\delta^l_j = \frac{\partial J}{\partial z^l_j}$, the error of neuron j in layer l. Then it can be easily derived that the following four equations are true for any backpropagation system with any number of hidden layers:

$$\delta^L = \frac{\partial J}{\partial a^L} \odot h'_L(z^L) \tag{1a}$$

$$\delta^l = (W^{l+1})^T \delta^{l+1} \odot h'_l(z^l) \text{ for } l = 1, ..., L-1 \tag{1b}$$

$$\frac{\partial J}{\partial b^l_j} = \delta^l_j \text{ for } l = 1, ..., L \tag{1c}$$

$$\frac{\partial J}{\partial w^l_{jk}} = \delta^l_j (a^{l-1}_k)^T \text{ for } l = 1, ..., L \tag{1d}$$

where $h'_L(z^L) = p_\theta * (1 - p_\theta)$ and $tanh'(x) = 1 - tanh^2(x)$.

For current system, we've three layers: input layer, hidden layer and output layer. The cost function is $J = -ylna^L$. Using the above general system, we can obtain the following:

$$\delta^3 = p_\theta - y \tag{2a}$$

$$\delta^2 = (W^3)^T \delta^3 \odot tanh'(z^2) = U^T (p_\theta - y) \odot tanh'(Wx + b^{(1)}) \tag{2b}$$

$$\delta^1 = (W^2)^T \delta^2 \odot I'(x) = W^T \delta^2 = W^T U^T \delta^3 \odot tanh'(Wx + b^{(1)}) \tag{2c}$$

And

$$\frac{\partial J}{\partial U} = \delta^3 (a^2)^T = tanh(Wx + b^{(1)})(p_\theta - y) \tag{3a}$$

$$\frac{\partial J}{\partial W} = \delta^2 (a^1)^T = LU^T (p_\theta - y) \odot tanh'(Wx + b^{(1)}) \tag{3b}$$

$$\frac{\partial J}{\partial L} = \delta^1 = W^T U^T (p_\theta - y) \odot tanh'(Wx + b^{(1)}) \tag{3c}$$

$$\frac{\partial J}{\partial b^{(2)}} = \delta^3 = p_\theta - y \tag{3d}$$

$$\frac{\partial J}{\partial b^{(1)}} = \delta^2 = U^T (p_\theta - y) \odot tanh'(Wx + b^{(1)}) \tag{3e}$$

Our model correctly passes the supplied gradient checking function.

4. Model Results

We started by creating a baseline model that did an exact string match between the word-entity pairs seen in the training data with the test. This provided 68.49% F1 on the test if we ignore case.

After building our model, we wanted to find a set of baseline parameters that gave solid performance as a starting point for our tests. We first tested a wide set of possible parameter values to understand the broad impacts of different parameters, and we then iteratively tried increasingly smaller permutations of the best performing networks.

During this process, we found that a flat small learning rate provided low performance and poor training speed, but a flat large learning rate learned quickly at first but then experienced large jumps in performance in later epochs. To balance this, we tested several different learning rate decay approaches until settling on dividing the learning rate by the current epoch number. The first epoch would use the provided learning rate, the second would be half the provided learning rate, the third would be one-third the provided learning rate, etc.

We finally settled on a model that gave an overall training F1 value of **91.37%** and a holdout set overall F1 of **82.73%**. This model used a window size of 5, a single hidden layer of 100 dimensions, an initial learning rate of 0.03 (that decayed across epochs), a lambda value of 0.001, including updating of the word vectors, and ran for 20 iterations.

After this, we ran a number of a number of controlled tests to understand the impact of each parameter on the network's performance (**graphs are provided at the end**):

a. Learning Rates and Iteration Count

Our first test was experimenting with different learning rates across epochs. The results indicated 0.03 is near the optimal point. Lower learning rates not only had worse early performance but also saw their performance near flat line at 20 epochs, which may be due to the learning rate now allowing the model to surpass the regularization. Larger learning rates learned quicker but eventually began jumping around in later epochs. The results do suggest that training beyond 20 epochs may provide further improvement in performance.

b. Regularization Values

Testing different regularization values showed a modest impact in performance. Larger values than our baseline 0.001 began to have large impacts on performance. Smaller values actually increased the performance, and the 0.0001 gave 84.03% overall F1 on the test set. However, these lower values also showed an increasing disconnect between the training and test scores, which indicates the model was increasingly overfitting to the training data.

c. Hidden Layer Size

We trained a series of single hidden layer neural networks with dimensions from 100 to 300 with a step size of 50. We observed that the performance is very flat for both training and test set. For single layer neural network, adding layers above 100 provides little incremental value.

d. Window Size

To study the effect of window size, we varied the value of window from 3 to 15 with step size 2. We observed there is a substantial performance improvement from 3 to 5. After that, F1 score increases as the window size increases on training. However, the test F1 score remains relatively flat, which suggests overfitting for larger window sizes. This may be because larger window sizes increase the model but also data sparsity,

e. Fixed Word Vectors vs. Updating Word Vectors

Moving from fixing the word vectors to updating the word vectors provided one of the biggest boosts of all the options. This is intuitive as the word vectors were created in an unsupervised manner based on word co-occurrence, which will place words that appear in similar contexts near each other in the space. However, this does not mean the words represent similar named entities, so the model benefits greatly from being able to adjust the space.

f. Randomly Initialized vs. Pre-Trained Word Vectors

We also compared the performance of using randomly initialized word vectors and pre-trained word vectors. Pre-training improved the performance dramatically on both training and test data. Does pre-training always help? Intuitively, if words were pertained on relevant corpus this should be the case. But what if words were pre- trained on Wall Street Journal and the problem is to do NER on corpus related to Football or some other not relevant field? Without more experiments we can't make a general conclusion.

5. Error Analysis

Using our benchmark model above, we also explored the reason for errors in each entity types:

a. Location

There were several common errors with locations. First, the model had trouble with organization or person names that are also locations like "Dynamo Moscow" and "Washington" as a last name. These are difficult due to their frequency as locations, and the model would need more semantic understanding of the sentence to get them right. The model also made a number of errors where case may have helped like excluding *city* from "Panama City" or considering "Brown Deer Park Golf Course" to be a combination of a person "Brown Deer", a location "Park", and a miscellaneous entity "Golf". Finally, the model made mistakes like assuming "93.94" was a location in the Wall Street speak "down 11 bps at 93.94", which some rule based overlays could resolve.

b. Organization

Similar to Location, many of the Organization misclassifications were due to person, location, or animal names in the organization (e.g. Moody's). The model also made mistakes on organizations with mostly common words in their name like "Test and Country Credit Board", and taking case into account may help the model identify these as special. One interesting mistake that caused a huge number of errors was where the model appeared to overfit to one sports rating pattern, where the organization came after the country in parenthesis, when a different rating pattern placed a race time after the country in parenthesis. This is a classic case of "know your data" and could be resolved by more carefully preprocessing or cleaning the data.

c. Person

For false positives, almost 50% are where O is misclassified as PER. Most of these are numbers (e.g. 1988) and compound adjectives (e.g. over-allotment, soft-spoken). These could be partially avoided if we could enforce some rules like numbers are never (or rarely) a person. An additional 26% are ORG misclassified as PER, and many of these are organizations that are named after their founders, which is very difficult for the model to distinguish. Another 15% of the errors are MISC misclassified as PER. One interesting example is "Michael Collins", which is a very common human name, but actually relates to a movie. This type of error is hard to avoid without the model having much broader semantic understanding.

The false negatives are dominated by misclassifying PER as O. Most of these errors relate to uncommon English names like Inzamam-ul-Haq, Djorkaeff, and Sihanouk. This type of error can be reduced by training on translated English text or other more global texts.

d. Miscellaneous

The false positives are dominated by misclassifying O as MISC. The majority of these are numbers (e.g. 13, 40) and compound adjectives (e.g. little-known, army-backed).

The false negatives are dominated by misclassifying MISC as O. Many of them are related to sports events like "English County Championship" and "U.S. Open Tennis Championship". The model often correctly classifies parts of the phrase as MISC, like U.S. Open, but then fails to classify Championship correctly. This is challenging to fix as most of the time Championship is actually an O except in compound phrases like this. Similarly, MISC is often misclassified as LOC in some compound phrases like "the *Hong Kong* Open" and "the *Chicago* PMI". These errors may be reduced by sequence modeling or treating those phrases as a single "word".

6. Extra Credit

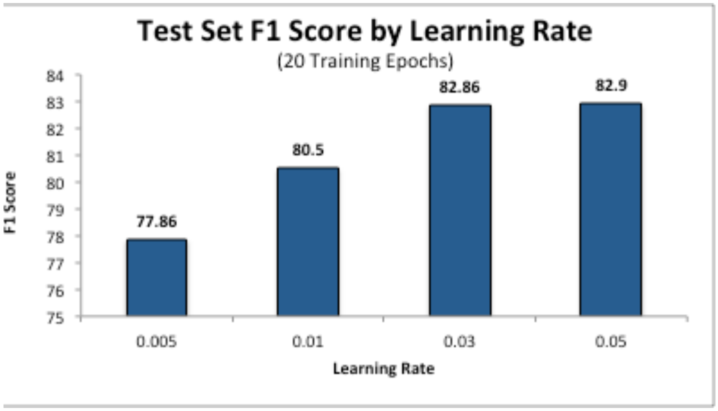
We also explored a couple different extra credit opportunities to further test the performance of our network and better understand how the network was learning.

a. Deeper Networks

Our system design was built to allow for any hidden layer architecture, and so to study the effect of deeper networks, we ran experiments with two hidden layers with sizes to be the Cartesian combination of {50, 100} and {50, 150, 300} for the first and second hidden layer, respectively. This showed deeper networks do improve performance but only slightly. This held true even when we moved to 3 and 4 hidden layer networks. This suggests that most of the "power" available with the window model is already captured by the smaller models.

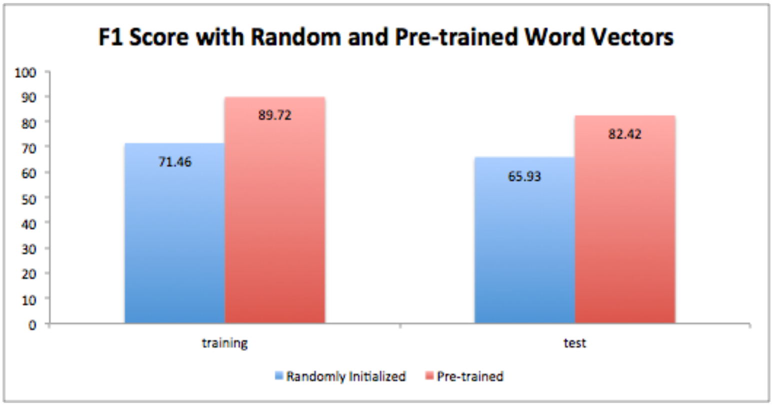
b. Visualization of Word Vector Training

<PATRICK>

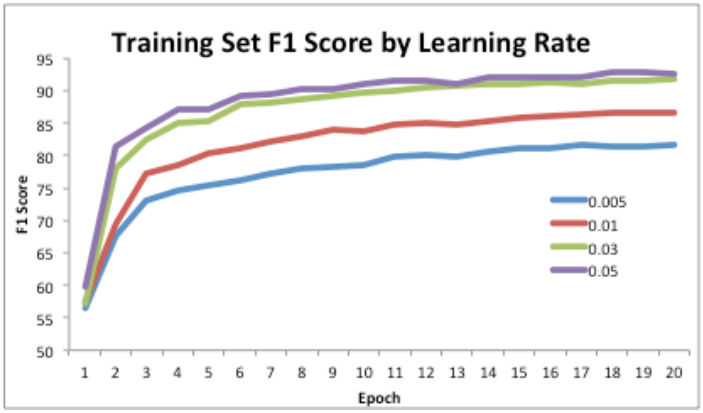


Test Set F1 Score by Learning Rate

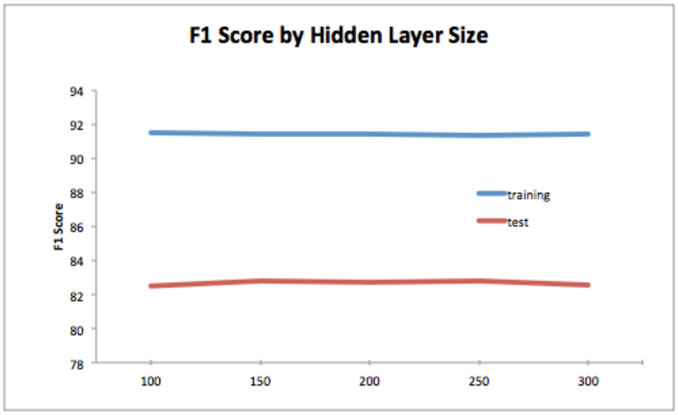
(20 Training Epochs)



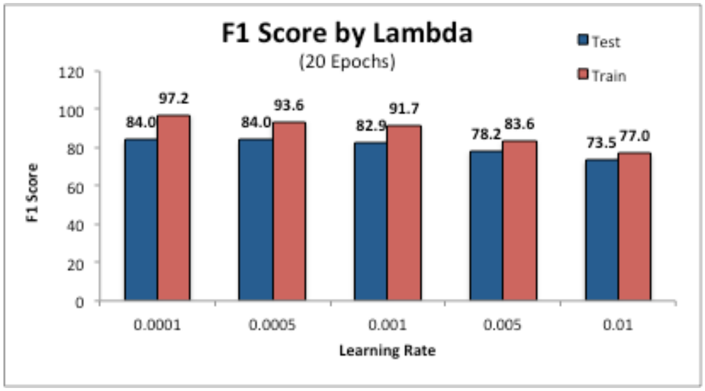
F1 Score with Random and Pre-trained Word Vectors



Training Set F1 Score by Learning Rate

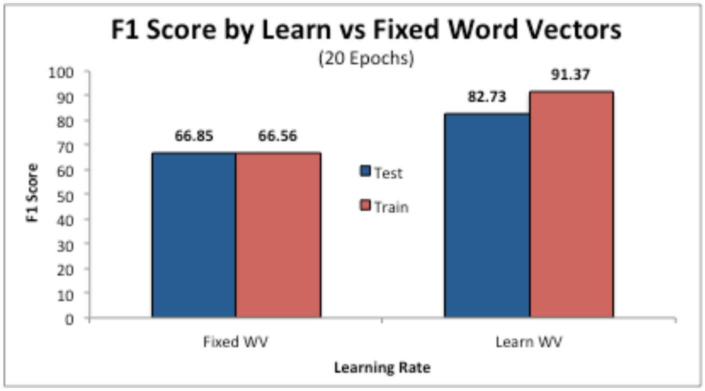


F1 Score by Hidden Layer Size



F1 Score by Lambda

(20 Epochs)



F1 Score by Learn vs Fixed Word Vectors

(20 Epochs)