**Neural Networks for Named Entity Recognition**

**Stanford CS224n Natural Language Processing – 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 and back propagation calculations 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 Details**

<DAOYING>

**4. Model Results**

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.

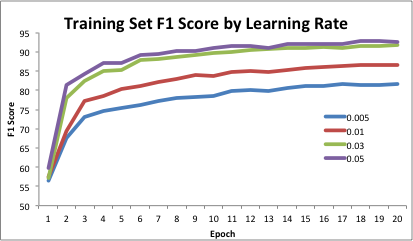
We finally settled on a model that gave an overall training F1 value of **91.37%** and a holdout set overall F1 of **82.73%**. This model used a window size of 5, a single hidden layer of 100 dimensions, an initial learning rate of 0.03 (that decayed across epochs), a lambda value of 0.001, including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

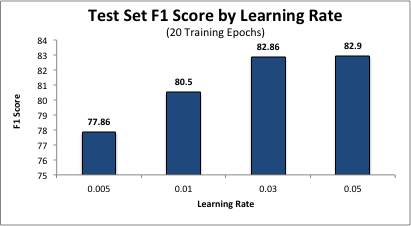
After this, we ran a number of a number of controlled tests to understand the impact of each parameter on the network’s performance:

**a. Learning Rates and Iteration Count**

The first test we conducted was experimenting with different learning rates and their performance across different training epochs. The results indicate that our 0.03 learning rate is near the optimal point.

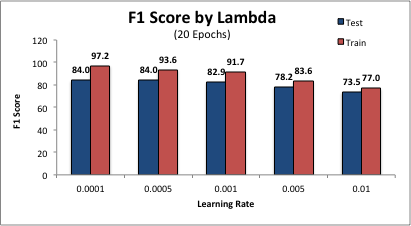
Lower learning rates not only had worse early performance but also saw their performance near flat line at 20 epochs, which may be due to the learning rate now allowing the model to surpass the regularization. Larger learning rates learned quicker but eventually began jumping around in later epochs. 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**

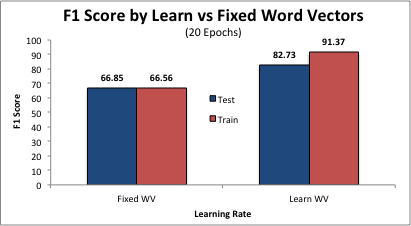
<DAOYING>

**d. Window Size**

<DAOYING>

**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**

<DAOYING>

**5. Error Analysis**

a. Location

b. Organization

c. Person

d. Miscellaneous

<BOTH OF US – CAN WE CREATE SOME SECTIONS?>

**6. Extra Credit**

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

**a. Alternative Word Vectors**

<DAOYING>

**b. Deeper Networks**

<DAOYING>

**a. Visualization of Word Vector Training**

<PATRICK>