

# Recursive Neural Network Part-of-Speech Tagger

**Hao Liu**  
NYU CIMS  
New York, NY  
haoliu@nyu.edu

**Jiali Huang**  
NYU CIMS  
New York, NY  
jiali.huang@nyu.edu

**Robert Dionne**  
NYU Game Center  
New York, NY  
robertsdionne@nyu.edu

## Abstract

We use a recursive neural network (RNN) on a part-of-speech tagging task (POS). We propose and implement two models: a basic RNN and a Compositional Vector Tagger (CVT) RNN. The performance of the basic RNN is close to the HMM model and will likely beat it if capitalization information is captured. The intractable tagging time of the CVT model means we cannot analyze it. We tried some useful neural network training tricks and carefully analyze the performance of the basic RNN model, by verifying the contribution of sequential information and error rates of different parts of sentences.

## 1 Introduction

For our final project, we implemented a recursive neural network part-of-speech tagger based on similar prior work by Socher, Manning and Ng (2010) for building sentence parse trees and work by Collobert and Weston (2011) for their distributed word representations. We wanted to explore natural language processing using neural networks, so we chose a small problem (part-of-speech tagging) which we already had experience with during the third homework assignment and applied these modern techniques in neural network research to the problem.

### 1.1 Recursive Neural Networks

Recursive neural networks are neural network models that allow for cyclic connections between neurons. Generally, recursive networks can have very complex structures, but for our application to part-of-speech tagging, we consider a restricted structure in which the network observes its own output on prior sentence context while consuming as input the current word in order to produce a

classification label representing the current word’s part of speech tag. When training the network or using the network to classify an example sentence, we temporarily unroll the recurrent network into an equivalent, feedforward multilayer perceptron with the weight matrix shared between all layers.

Our network structure is a simplification of the more general binary tree structure trained by Socher et al. (2010) which iteratively constructs a parse tree by evaluating, at the base case, the best combinations of pairs of words, and subsequently the best combinations of pairs of previous combinations. Instead, we construct a linear, degenerate tree by always pairing the current word (on the right) to the previous combination (on the left) of prior sentence context and the prior word to obtain a phrase representation. We hope that the resulting phrase representation captures both the possible parts-of-speech for the current word and distinguishing context that allows an accurate choice of the correct part-of-speech.

### 1.2 Distributed Word Representations

Underlying the entire model are distributed word representations, which we use directly from the research of Collobert and Weston (2011), downloadable from their website ([ronan.collobert.com/senna](http://ronan.collobert.com/senna)). Distributed word representations are dense vectors embedded within an  $N$ -dimensional vector space that differ from sparse, “one-hot” vectors embedded within a  $|\mathcal{V}|$ -dimensional vector space, where  $\mathcal{V}$  is the vocabulary of the language domain.

The distributed word representations are induced by training a neural network in an unsupervised manner to distinguish a word within its concrete contexts from randomly chosen replacement words within those same contexts, and then back-propagating the errors into an  $N \times |\mathcal{V}|$ -dimensional lookup table with randomly initialized values. The lookup table converts between the original “one-

hot” vector representation, which encodes zero information about the word aside from identity, and the superior distributed representation, which encodes a word’s contextual information. Since we believe a word’s context encodes information about that word’s part-of-speech, we hypothesize that distributed word representations will be vital to our part-of-speech tagging model.

## 2 Our Models

We implemented a recursive neural network with the programming language Lua and the neural network library Torch7 (Collobert, 2011). Our code is available on Github at [github.com/robertsdionne/nlp](https://github.com/robertsdionne/nlp).

The first model we considered was a basic recursive neural network that combines a phrase representation  $p_{i-1}$  encompassing words  $w_1, w_2, \dots, w_{i-1}$  and the word representation  $w_i$  into an output phrase representation  $p_i$  passed to a multilayer perceptron classifier which outputs the guessed tag.

The second model we considered was a compositional vector tagger, based upon the compositional vector grammar model by Socher, et al. (2013), in which different weight matrices are associated with different part-of-speech tags in order to generate tag-dependent phrase representations, which are then scored instead of classified. This model would perform sentence tagging by maximizing the overall score by choosing the correct tag-associated weight matrices for each word using a beam or greedy search.

The inputs to both models consist of:

$\mathcal{V} = \{w_1, w_2, \dots, w_{ \mathcal{V} }\}$	the vocabulary
$\mathcal{T} = \{t_1, t_2, \dots, t_{ \mathcal{T} }\}$	the part-of-speech tags
$N = 50, P = 50$	word/phrase dimension
$V \in M_{N \times  \mathcal{V} }(\mathbb{R})$	the word lookup table
$ w_i\rangle \in \mathcal{V}$	the “one-hot” vectors
$V w_i\rangle \in \mathbb{R}^N$	the word vectors
$\begin{bmatrix} \bullet \\ \bullet \end{bmatrix}$	vector concatenation

### 2.1 Basic Recursive Neural Network Model

The basic recursive neural network model also has the following parameters:

$W_{combine} \in M_{P \times (P+N)}(\mathbb{R})$	matrix for combining
$b_{combine} \in \mathbb{R}^P$	bias for combining
$W_{classify} \in M_{ \mathcal{T}  \times P}(\mathbb{R})$	matrix for tagging
$b_{classify} \in \mathbb{R}^{ \mathcal{T} }$	bias for tagging

We began by training our model on the Penn Treebank sentences from assignment three, available at [www.cs.nyu.edu/petrov/restricted/data3.zip](http://www.cs.nyu.edu/petrov/restricted/data3.zip).

To train the model, we first initialize  $W_{combine}$ ,  $b_{combine}$ ,  $W_{classify}$  and  $b_{classify}$  with small, uniformly distributed random values. We will discuss how to choose the range for these values in Section 3.2.1 below. Then we perform a classification step in which we convert each word into its distributed word representation, pass each of these representations through the unrolled recursive neural network’s inputs, and feed the values forward to the classification layers corresponding to each tag output.

For instance, given the sentence  $(w_{i_1}, w_{i_2}, \dots, w_{i_K})$  of length  $K$ , we first calculate the word representations  $V|w_{i_1}\rangle, \dots, V|w_{i_K}\rangle$ , then we calculate each phrase representation  $p_j$  from the previous phrase representation  $p_{j-1}$  and the current word representation  $V|w_{i_j}\rangle$  by concatenating the two representation vectors, multiplying by  $W_{combine}$  and adding  $b_{combine}$  before applying the  $\tanh(\cdot)$  function:

$$p_j = \tanh(W_{combine} [p_{j-1}; V|w_{i_j}\rangle] + b_{combine})$$

Subsequently, each  $p_j$  is passed along to the classifier, which applies the operation:

$$t_{i_j} = \underset{t}{\operatorname{argmax}} \operatorname{softmax}(W_{classify} p_j + b_{classify})$$

to obtain the predicted tag  $t_{i_j}$  for  $w_{i_j}$ .

Finally, we perform backpropagation through time, described in section 3.1 below, to update the weight matrices and bias vectors. Performing a backpropagation after classifying each sentence in the dataset performs stochastic gradient descent in minibatches (where we consider one sentence to be a minibatch of tags and words) to update the weight matrices to more accurately predict the golden tags.

### 2.2 Compositional Vector Tagger Model

The second model we considered uses weight matrices that depend upon the golden tags of the training sentences. The parameters are as follows:

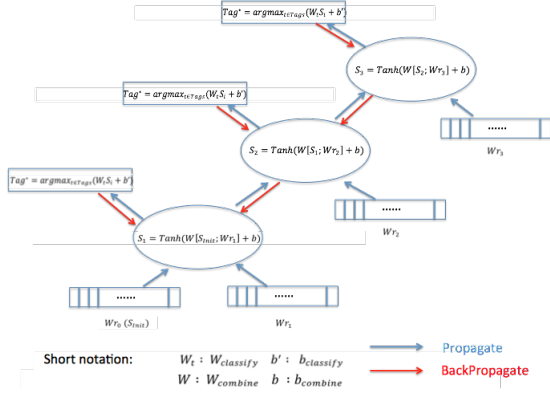


Figure 1: Recursive Neural Network Model

$$\begin{aligned}
 W_{combine,t} &\in M_{P \times (P+N)}(\mathbb{R}) && \text{matrix for tag } t \\
 b_{combine,t} &\in \mathbb{R}^P && \text{bias for tag } t \\
 W_{score} &\in M_{1 \times P}(\mathbb{R}) && \text{matrix for scoring} \\
 b_{score} &\in \mathbb{R} && \text{bias for scoring}
 \end{aligned}$$

The training and classification procedures differ from the basic model. First, in unrolling the recurrent neural network, each recurrent layer adopts the weight matrix associated with the golden tag of the word in that position. For instance, if  $w_{i_3}$  had tag  $t_{i_3} = \text{NN}$  then the third recurrent layer would adopt the weight matrix  $W_{combine, \text{NN}}$ . Each recurrent layer adopts the appropriate weight matrix before generating the phrase representations  $p_j$ . We then apply backpropagation to maximize the score for the representations provided by the golden tag weight matrices by updating those matrices and the score matrix.

Raw classification on unlabeled examples now requires a search procedure. The brute-force procedure to tag a sentence of length  $K$  would be to search through all  $K^{|\mathcal{T}|}$  sequences of tag weight matrices and return the sequence that attains the highest overall score based on the score matrix. As our tag set has size  $|\mathcal{T}| = 48$ , this search is intractable for realistic sentences, so a greedy or beam search would need to be used instead.

As the complexity of the compositional vector tagger model is significantly higher than the basic recurrent neural network model, we were unable to properly train the model and implement a satisfactory search procedure within the remaining time of the assignment, so we have omitted analysis of the results of this model.

## 2.3 Future Models

**Manually Extended Embeddings** From the analysis of Section 3.3.3 and also the results in Section 3.3.2, we see that because of the erasure of capitalization information of words, the RNN model is bad at distinguishing between NN and NNP. So we think we can manually extend the embeddings of words by some human made features like capitalized or not, or perhaps even suffix information. This, in our view, will likely improve the performance of the model a lot.

### Bi-directional Recursive Neural Network

From Section 3.3.3 Figure 6, we find it is more likely to make mistakes on the end of sentences. It makes sense, because our basic RNN model uses only forward sequential information. So we think a bi-directional model will help, at least compared to the basic RNN. As for CVT model, which is simulating the HMM model considering the tags of whole sentence together, a bi-directional model may also help to kind of simulate CRF. There are two bi-direction models we are thinking:

- **Independently training and combining the probability:** Train two RNN models, and tag the sentence by combining the predicted probability from the two models' classifiers (after the softmax).
- **Jointly training:** Train two RNN models jointly, and concatenate two phrase representations and pass it to a shared classifier, which will in turn backward propagate the gradients to the two models so that the classifier part will get the forward and backward sequential information from two phrase representations. Figure 2 shows the idea of this jointly training model.

### Complex combination and classification model

We are using one layer linear neural network on both combination and classification parts, because it is easy to train. In the beginning, we did try a two layer neural network, but it was too slow to train (about five times slower) and added another two hyper parameters to choose. However, because of the higher ability to capture more complex patterns, complex combination and classification may have better performance. Especially on classification part, we can use any machine learning classification model that can backward propagate gradients.

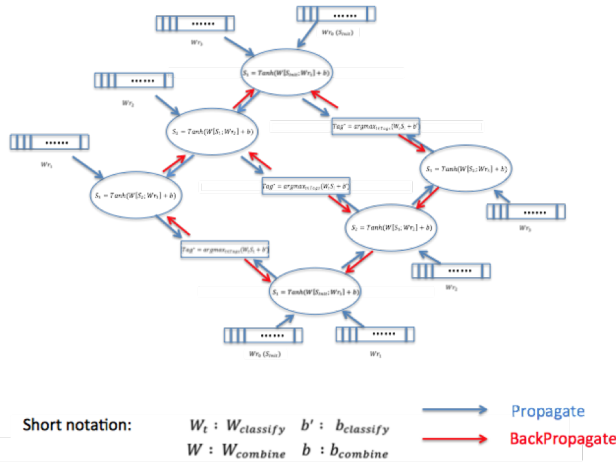


Figure 2: Idea chart for jointly training bi-direction RNN model

**Update the lookup table** In the beginning, we tried to update the lookup table together with other parameters. But the result was much worse than not updating it. But because deep learning researchers do update the feature learning layers (in our model, just the lookup table), it is reasonable to try this. And instead of updating at the beginning of training, we can begin the updating after other parameters are well trained.

**Rerank with CVT** When we implemented and tested the CVT model, the forever tagging time for each sentence stopped us from working with it. So in the future, we can use the rerank idea from Socher’s CVG paper. Basically, we get top N, say 100, tagging candidates from other models, and use CVT model to get score for each sentence and rerank the candidates by these scores.

### 3 Experiments

Our experiments consisted of training the recursive network on training sentences drawn from the Penn Treebank data of assignment three, as we described above. We frequently trained on 100, 1000, 10,000 or 39,815 (all) of the training sentences in exploring the performance of the recursive neural network for part-of-speech tagging. Training on the entire data set for 25 iterations, or on 10,000 training sentences for 100 iterations, each took about 4 hours on a 2012 Macbook Pro Retina with a 2.6 GHz Intel Core i7 CPU and 8 GB of RAM.

To evaluate the network, we used a basic metric which counts the ratio of the number of correctly tagged words out of the total number of words

within the validation or test set (1700 in-domain validation sentences, 1016 out-of-domain validation sentences and 1015 out-of-domain test sentences). We also count the ratio of correctly tagged unknown words, where a word is defined to be unknown if it was not encountered during the training set; however, note that it may be present within  $\mathcal{V}$ , the vocabulary associated with the distributed representation lookup table. If the word is unknown to the distributed representation lookup table, the model falls back upon the distributed representation for the special token “UNKNOWN.”

### 3.1 Backpropagation Through Time

Backpropagation through time is a technique to run backpropagation against recursive neural networks, which include cyclic connections. We perform backpropagation through time automatically by running standard backpropagation against our unrolled recursive neural network with shared weight matrices. Each gradient update to the weight matrix is independently accumulated, resulting in the correct overall update. We perform the gradient update in minibatches, where a single sentence is a minibatch of tagged words and we simultaneously accumulate the errors for each of the words as we backpropagate. See Figure 3 (Mikolov, 2012).

### 3.2 Training Tricks

Training neural networks can be tricky since the backpropagation algorithm relies upon a learning rate parameter as well as initial weight and bias values. The learning rate should not be too large, otherwise the gradient descent will skip over local or global minima. However, it should also not be too small, otherwise training may take too long. It perhaps should vary over the course of training and decrease in order to better fine tune the fit to the minima. However, it is not obvious exactly which set of choices will be best.

We researched existing techniques to choose initial weights and biases, and to better update the weights based on adaptive learning rates, as described in the next subsections.

#### 3.2.1 Weight Initialization

Early on, we randomly initialized our weight matrices and biases using Torch7’s uniform tensor function,

```
torch.rand(P, P+N)
torch.rand(P)
```

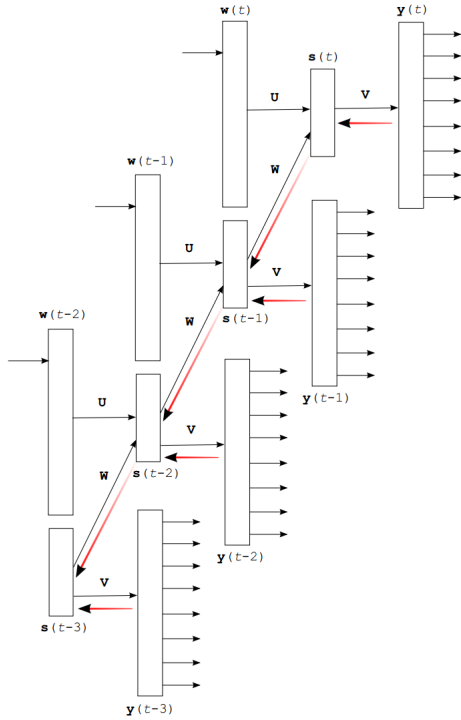


Figure 3: Back Propagation Through Time (Mikolov, 2012)

which draws from  $\mathcal{U}[0, 1)$ . However, we discovered guidelines for weight initialization from [deeplearning.net/tutorial/mlp.html#mlp](https://deeplearning.net/tutorial/mlp.html#mlp), borrowed from Glorot and Bengio (2010), and decided to explore how they impacted our training convergence.

Glorot and Bengio (2010) analyzed how initial values for weights influenced the early learning of neural networks, specifically those that use  $\tanh(\cdot)$  or  $\text{sigmoid}(\cdot)$  and discovered optimal early learning rates with weights drawn from

- $\mathcal{U}\left[-\sqrt{\frac{6}{fan_{in}+fan_{out}}}, \sqrt{\frac{6}{fan_{in}+fan_{out}}}\right)$  for  $\tanh$  layers
- $\mathcal{U}\left[-4\sqrt{\frac{6}{fan_{in}+fan_{out}}}, 4\sqrt{\frac{6}{fan_{in}+fan_{out}}}\right)$  for  $\text{sigmoid}$  layers

where  $fan_{in}$  is the number of inputs to the layer and  $fan_{out}$  is the number of outputs.

After adopting these new ranges, and initializing our biases to start at zero, our attainable accuracies dramatically increased from the values we presented in class in our slides, from about 60% to about 80%.

### 3.2.2 AdaGrad

We used the AdaGrad online learning method to train our model (Socher et al., 2013). Let  $\theta$  be a vector of all parameters needed to train an RNN model and  $\theta_t$  means the parameter vector before training on the  $t$ -th sentence. And let  $g_{t,i}$  to be the gradient value of  $i$ -th parameter on the  $t$ -th sentence. Then to perform AdaGrad is just to update the parameter vector by

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\alpha}{\sqrt{\sum_{s=1}^t \theta_{s,i}^2}} g_{t,i}$$

, where the  $\alpha$  is a constant to set the scale of the learning rate.

We compared the AdaGrad with fixed learning rate online training. The Figure 4 is the accuracy curve on training iteration. From Figure 4, it is quite clear that the AdaGrad converges faster and better than fixed learning rate. The discrepancy between the initial positions is due to plotting after the first iteration over the training set, at which point we've already performed one AdaGrad update for each sentence.

Additionally, we found that performing an AdaGrad update after each word evaluation performed worse than updating once after each sentence evaluation.

### 3.3 Analysis

#### 3.3.1 Training Progress Diagram

As we all know, there will be overfitting if we train too "hard" on training set. And if overfitting happens, the test accuracy (in our experiments, the out-of-domain accuracy) will drop. Figure 4 shows this drop on both total accuracy and unknown word accuracy.

#### 3.3.2 Comparison to Other Models

Below, we tally the results against the in-domain validation dataset and the out-of-domain validation dataset on all significantly performing recursive neural network models compared to the hidden-markov-model+TNT, plain hidden-markov-model and maximum likelihood tag model.

The AdaGrad variants started with a learning rate parameter at 0.1. The non-AdaGrad variants started with a learning rate of 0.0005. The RNN models trained on the entire training set (39815 sentences) were trained for 25 iterations.

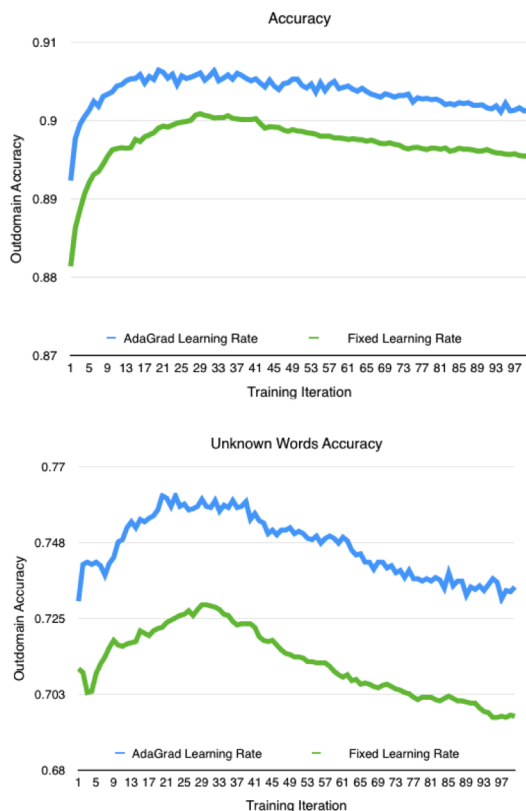


Figure 4: Out-of-domain accuracy comparison of adaGrad learning rate and fixed learning rate

The RNN models trained on just 10,000 sentences were trained for 100 iterations.

Because we stripped all capitalization from our training and test data before converting words to their distributed vector representations, we noticed a high error rate due to conflating proper nouns with regular nouns, so to be fair to our model we evaluated the best model with no distinction between the two tags NNP and NN to see how it would fare had it properly considered capitalization as a feature.

Also, because we are using fully-trained models, we are certainly overfitting, especially compared to the peak results depicted in Figure 4.

Model	In	(In Unk)	Out	(Out Unk)
HMM+TNT	95.88	(87.13)	93.67	(84.61)
AdaGrad RNN <sub>10,000</sub> NN=NNP	94.40	(78.48)	91.86	(70.37)
RNN <sub>all</sub> NN=NNP	94.52	(88.39)	91.70	(83.21)
HMM	95.52	(63.95)	91.23	(53.00)
AdaGrad RNN <sub>all</sub>	93.58	(76.90)	90.41	(69.82)
RNN <sub>all</sub>	93.47	(77.06)	90.28	(70.43)
AdaGrad RNN <sub>10,000</sub>	93.02	(79.17)	90.12	(73.43)
RNN <sub>10,000</sub>	92.72	(77.51)	89.54	(69.60)
Max Likelihood Tag	91.31	(42.74)	87.01	(53.31)

### 3.3.3 Error Analysis

**Confusion matrices** We plot the confusion matrix (Figure 5) for the basic RNN model and find that:

1. The **most serious confusion is between NNP and NN**. This makes sense because of the fact that the embeddings we are using make no distinction between capitalized and non-capitalized words;
2. Another strange confusion is from quotes, which was also met in common HMM models. This may come from the overwhelming sequential information, and it makes a big difference on accuracy (about 1% on out-of-domain accuracy), to keep everything trained automatically;
3. Other confusions look normal, and also occur in HMM models, like JJ and NN.

**Position of wrong tags** We plot the histogram (Figure 6) of relative position (absolute position in sentence divided by sentence length in (0, 1]).

- **In-domain** (left figure): The relative position is almost uniformly distributed.
- **Out-of-domain** (right figure): The number of wrong tags is generally increasing with relative position. And from the different scale

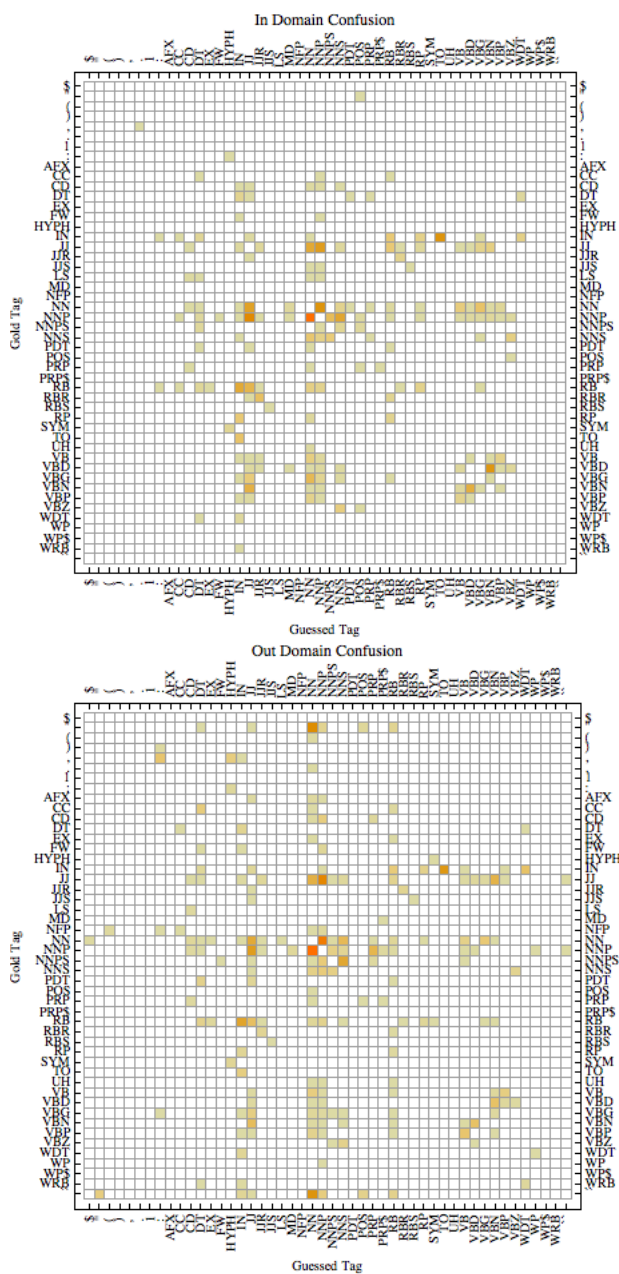


Figure 5: Confusion matrices for basic RNN

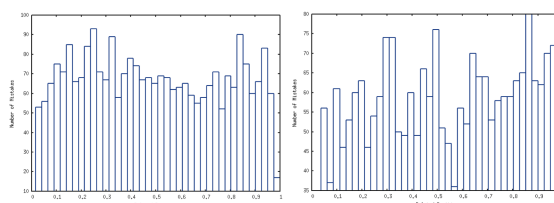


Figure 6: Histogram of related position of wrong tags, left is in domain, right is out of domain

of this histogram, we find there is a big drop around 0.5 (middle of sentence). This may come from the sequential similarity between in-domain and out-of-domain. This may also come from the limitation of sequential information passing. So a bi-directional RNN may help.

**Not just a maximum likelihood model** In the beginning, we worried about whether this RNN model can pass and use the sequential information correctly. Perhaps it just learns a maximum likelihood (most frequent) predictor for each word or even each group of words. So we did some experiments to verify the contribution of sequential information to the RNN results.

- **Same word test:** We chose some words with more than one common tag and saw whether our model predicted those tags correctly. We tested on 'to' (possible tags are TO and IN) and 'work' (common possible tags are NN, VB and VBP). Our RNN predicts 'to' with 0.80 in-domain and 0.77 out-of-domain accuracy, covering TO and IN. The RNN predicts 'work' with 0.88 in-domain and 0.83 out-of-domain accuracy, covering NN, VB and VBP.
- **Disconnected core module test:** We disconnected the sequential information passing by ignoring the input embeddings from lower layers, and backward gradient from upper layers. So although it is still an RNN, there is actually no information passed between layers. We trained this non-sequential RNN and the original RNN with 100 sentences and 100 iterations, same random seed and training parameters. The Figure 7 shows the out-of-domain accuracy of two models by the number of iterations, which can be also seen from Figure 8. The difference is not small. So we think the sequential information was passed and used at least partially correctly.

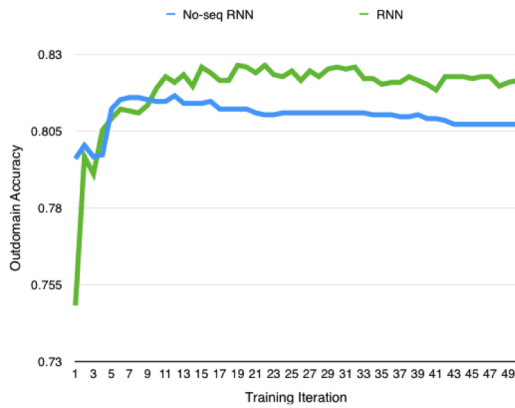


Figure 7: Comparison of non-sequential RNN and original RNN for disconnected core module test

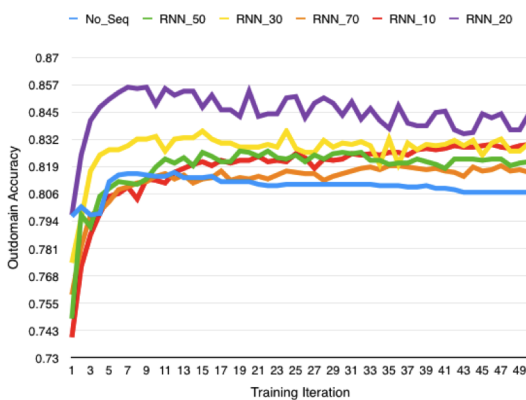


Figure 8: Out-of-domain accuracy comparison of different state embedding size

**Different embedding size of state** The size of state embeddings (left side of each RNN layer) is another hyper-parameter to set up. And it is important because it may determine the importance of previous sequential information compared to the current word embedding. Figure 8 shows the the out-of-domain accuracy of two models by the number of iterations. And it looks like 20 is around the optimal size for state embeddings given a training set size of 100. However, we trained on the full training set and observed better performance at the original state embedding size of 50.

## 4 Conclusion

We learned that because we eliminated capitalization information from our training and test data in order to use the distributed word embeddings from Collobert and Weston (2011), we severely handicapped the capability of our model to dis-

tinguish a large portion of part-of-speech tags, for instance by confusing proper nouns with regular nouns. However, we propose to solve this problem, in order to outperform the hidden-markov-model, by manually extending the word embeddings to encode capitalization, and perhaps plurality.

We also discovered that quotes cause large problems with our model because they frequently occur directly next to nouns, and are thus often confused with nouns. We would like to explore manual constraints to resolve this confusion in our model.

We propose to fix the error rate correlated with word index within a sentence by using a bidirectional model to gain context information flowing from both the front and the back of the sentence.

We are looking forward to all the future models proposed and the performance improvements they might bring.

## 5 Future Works

Because the limited time of final projects, there are still a lot of things we want to do in the future. We mentioned some new models to try in Section 2.3, and here are some most wanted works to do:

- Training and tagging the CVT model: we have mentioned the hardness of training and tagging with CVT model. In the future, we want to train a CVT model no matter how long it may take, say several days or weeks. And implement the rerank trick mentioned in Section 2.3.
- Move to GPU: We are working with torch (<http://www.torch.ch>), which is made by NYU people and supports GPU. So in the future, we can use GPU to speed up both training and tagging. And Theano (<http://deeplearning.net/software/theano/>) can be another choice, which has a Python interface and also GPU support and has very good neural network support.

## 6 Academic Honesty Pledge

Honor Pledge

We pledge our honor that all the work described in this report is solely ours and that we have given credit to all third party resources that we have used.



## References

- [Collobert and Weston2011] Ronan Collobert, Jason Weston, Lon Bottou, Michael Karlen, Koray Kavukcuoglu and Pavel Kuksa. 2011. Natural Language Processing (Almost) from Scratch. *Journal of Machine Learning Research 12*, (2011) 2461-2505. The MIT Press, Cambridge, MA.
- [Collobert, Kavukcuoglu and Farabet2011] Ronan Collobert, Koray Kavukcuoglu and Clment Farabet. 2011. Torch7: A matlab-like environment for machine learning. *BigLearn, NIPS Workshop*.
- [Glorot and Bengio2010] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. *International Conference on Artificial Intelligence and Statistics*.
- [Mikolov2012] Tom Mikolov. 2012. Statistical Language Models Based on Neural Networks. *Diss. Ph.D. thesis, Brno University of Technology*.
- [Socher, Bauer, Manning and Ng2013] Richard Socher, John Bauer, Christopher D. Manning and Andrew Ng. 2013. Parsing with Compositional Vector Grammars. *In Proceedings of the ACL conference*.
- [Socher, Manning and Ng2010] Richard Socher, Christopher D. Manning and Andrew Ng. 2010. Learning Continuous Phrase Representations and Syntactic Parsing with Recursive Neural Networks. *Proceedings of the NIPS-2010 Deep Learning and Unsupervised Feature Learning Workshop*.