# Natural Language Processing with Hierarchical Neural Network Models

by

**William R. Foland, Jr.**

B.S.E.E., University of Colorado, 1980

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Department of Computer Science

2014

This thesis entitled:
Natural Language Processing with Hierarchical Neural Network Models
written by William R. Foland, Jr.
has been approved for the Department of Computer Science

_____

Prof. James Martin

_____

Prof. Martha Palmer

_____

Prof. Wayne Ward

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Foland, Jr., William R. (MS, Computer Science)

Natural Language Processing with Hierarchical Neural Network Models

Thesis directed by Prof. James Martin

Unsupervised training has recently been successfully used to enhance the performance of neural networks. To understand the advantage provided by the structure of unsupervised pre trained models, a network theory based analysis of word representation similarities was performed, revealing the structure discovered by unsupervised models trained on a large english language corpus. A Part of Speech Tagger and two versions of Semantic Role Labelers were defined and tested to explore architectural configurations and training strategies. In order to thoroughly test various Neural Network Natural Language Models, a highly configurable software implementation was developed.

## Dedication

This thesis is dedicated to the memory of my loving mother, Dorothy, who watched with me in amazement as HAL refused to open the pod bay doors.

## Acknowledgements

Special thanks to Prof. Martin for encouraging me to investigate this fascinating architecture, and for taking the time to meet with me periodically to discuss it.

# Contents

**Chapter**

# Tables

**Table**

# Figures

**Figure**

# Chapter 1

# Introduction

"Once I knew only darkness and stillness...
my life was without past or future...
but a little word from the fingers of another
fell into my hand that clutched at emptiness,
and my heart leaped to the rapture of living."

**- Helen Keller**

Converting natural language text, such as written English, into programmer friendly data structures is a fundamental Artificial Intelligence goal. As with many complex problems, the task can be broken down into pipelined stages of analysis. Traditionally, linear statistical models are applied to ad-hoc features chosen manually for each pipelined algorithm, where rules are incrementally added to improve performance. This leads to processes which are sensitive to the chosen features, and to sets of heuristic, human work intensive enhancements which can be error prone and difficult to maintain. The training sets rely heavily on having domain experts (linguists, doctors, etc.) manually annotate large bodies of text, which is tedious and expensive.

Excellent results have been demonstrated by Bengio et al. [1], using an unsupervised learning approach to create what is sometimes called a Hierarchical Neural Network Language Model (HNN-LM). An improved model was later published by Collobert and Weston [7], and refined still further in Collobert et al. [8], which was used extensively as a reference for this thesis.

The word representations used in the model are created by running large amounts of text through a neural network structure, which learns a simple vector representation for each word. The context of the word within each sentence is the only input to the pre-training algorithm, which could be run on a corpus in any language, or even mixed languages. Domain specific models can be created by choosing specific input corpora, for example medical or legal texts, to enhance performance in those areas.

This thesis is presented in chapters.

- Chapter 2 describes the general Neural Network algorithms and equations used to implement them.

- Chapter 3 describes the Viterbi algorithm and considerations for creating and training Hierarchical Systems which include Neural Networks.

- Chapter 4 discusses the Word Representation Model, and analyzes the structures embedded in this model, created by an unsupervised learning algorithm.

- Chapter 5 describes the architecture for the Part of Speech Tagger, and presents the results of various experiments with training and testing the model.

- Chapter 6 describes the more extensive system designed for Semantic Role Labeling, and the results obtained with it.

- Chapter 7 discusses the Dependency Parser driven Semantic Role Labeler and results obtained with it.

- Finally, Conclusions are drawn in Chapter 8.

# Chapter 2

## Neural Networks

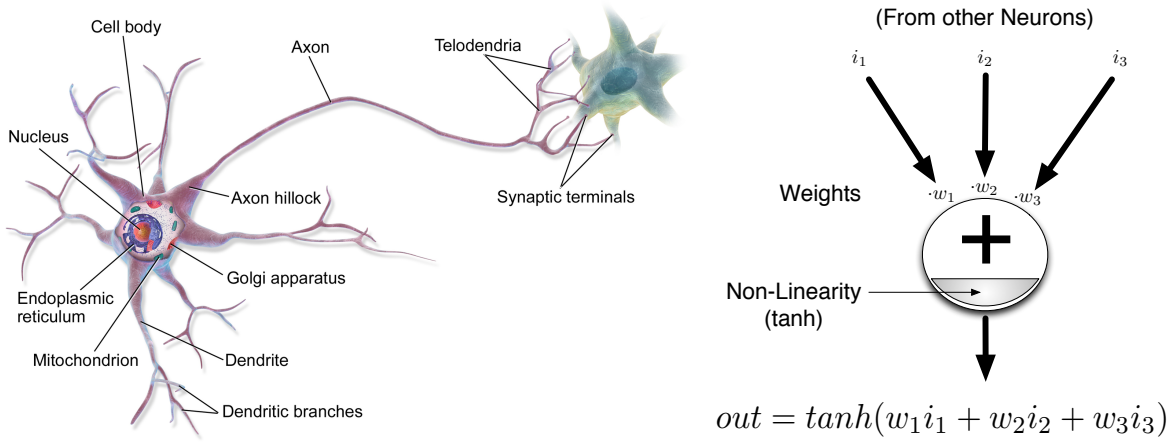
The hierarchical neural network systems which are used to implement the natural language processing tasks described in chapters 3, 5, 6 and 7 include simple artificial neural networks. These are algorithms which can be run on a computer which loosely mimic the way we believe that brains, including human brains, work. The theory is over fifty years old (Rosenblatt [18]), but new applications using advanced architectures and much cheaper and powerful computers are making them very interesting once again. Practical advanced applications including the Jeopardy competing Watson from IBM and Apple's Siri use this technology successfully today.

In a biological neural network (figure 2.1(a)), axon terminals connect via synapses to dendrites on other neurons. The electrical signals from one neuron to another can have different synaptic strengths. If the sum of the input signals into one neuron surpasses a certain threshold, the neuron sends an action potential (AP) at the axon hillock and transmits this electrical signal along the axon to the next group of neurons.

In an artificial neural network (figure 2.1(b)), signals from a simulated neuron are multiplied by a **weight** before being transmitted to other neurons. This weight is similar to synaptic strength in a biological network.

Biological axon firing based on a threshold is a form of **non-linearity**. The artificial equivalent is a quick but smooth mathematical transition, such as a hyperbolic tangent. An example of a non-linear function which is applied to the output of artificial neurons is shown in figure 2.2. Two

of the most common ones are the sigmoid and the tanh function. Nonlinearities that are symmetric around the origin are preferred because they tend to produce zero-mean inputs to the next layer (which leads to better convergence properties).




(a) Biological (b) Artificial

Figure 2.1: Biological [1] and Artificial Neurons

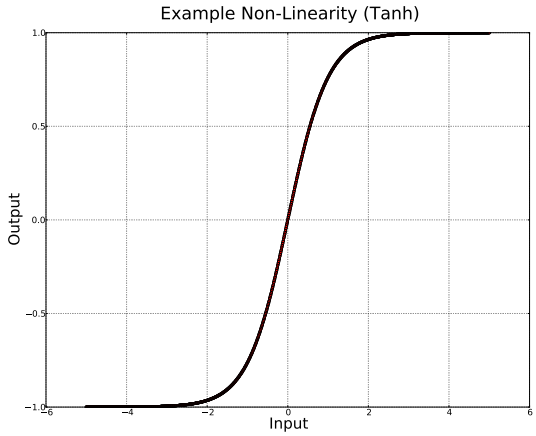



Figure 2.2: Example of a Non-Linearity
The artificial equivalent of neuron firing is a quick but smooth mathematical transition, such as the hyperbolic tangent shown. As the input increases from a negative value to a positive one, the output switches form -1 to +1. The threshold can be manipulated with a bias term.

[1] Source: "Blausen 0657 MultipolarNeuron" by BruceBlaus - Own work. Licensed under Creative Commons Attribution 3.0 via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:Blausen_0657_MultipolarNeuron.png#mediaviewer/File:Blausen_0657_MultipolarNeuron.png.

In brains, the concept of learning is believed to be the process of forming and breaking connections between neurons over time. In addition, the strength of these connections is a part of the brain's primitive structure (synaptic strength) . The brain consists of trillions of these connections, but has many hierarchical levels of organization, for example the cortex, neocortex, visual cortex, etc. which specialize in various functions.

Like many natural phenomena, rather than the incredibly complex structure of trillions of independent connections, the brain is believed to be organized in a more organized, fundamental way which can be thought of as successive application of a common pattern of neurons, which some, such as Kurzweil [13], call a pattern recognizer. The brain is remarkably adaptable. For example, it is known that when a specialized part of the brain is damaged, other parts can learn to compensate, suggesting that seemingly unlike functions such as sight or speech are composed of similar structures.

Artificial Neural Networks are algorithms which are constructed to be similar to the way a brain pattern recognizer works (see figure 5.5). A general structure is created with default connections and connection strengths, and an algorithm is applied to "learn" the connections and weights of this general structure. The algorithm is called "back-propagation" because it runs in the opposite direction from which the network normally runs, as will be explained. The process of running this learning algorithm is called "training".

When the parameters of a network have been trained, the network is ready to perform its task, which is referred to as running the network, or model, forward.

## 2.1 Forward

Neural networks are composed of layers. The parameters for each layer are referred to as $\Theta$, which includes a matrix of weights, $W$, and a vector of bias terms $b$. Each layer's output, prior to the activation function, can be calculated from the previous layer's activation output and parameters.

$$f_\Theta^l = W^{l-1} f_\Theta^{l-1} + b^{l-1} \tag{2.1}$$

## 2.2 Word Level Likelihood

An objective function is attached to the outputs of the network in order to provide a framework for what the network produces. A function which produces higher outputs, or scores, for good results, and low outputs for bad results is desired, and can be used to train the network to achieve that objective. One objective function is for example a mean squared error. A more commonly used, probabilistic objective is log-likelihood. To maximize a log-likelihood objective, the predictions of the network are converted to properly normalized log-probabilities using a softmax function (Bridle [4]), which turns a linear regression into a logistic regression. This is sometimes referred as stacking a softmax function on top. This will coerce the network into producing normalized probabilities, which are perfect for classification problems where we are trying to figure out the most probable class to assign to the input.

Using the notation from [8], the score of the system for $tag_i$, given a $\Theta$ and training example x, is $[f(\Theta, x)]_i$. The training example x is composed of the words of a sentence and some limited features extracted from the words, which are specific to the model. For Word Level Likelihood, a conditional tag probability given the training sample and the system parameters, $p(i|x, \Theta)$ can be computed as:

$$p(i|x, \Theta) = \frac{e^{|f(\Theta,x)|_i}}{\sum_k e^{|f(\Theta,x)|_k}} \tag{2.2}$$

Defining the log add operator as

$$\underset{i}{logadd}(z_i) = log(\sum_i e^{z_i}) \tag{2.3}$$

To simplify some of the following descriptions the reference to x will be omitted, so that the output of the network for a given tag i, $|f(\Theta, x)|_i$, will be shortened to $f[\Theta]_i$.

It's mathematically convenient to maximize the log of the probability, called log likelihood. The log-likelihood of one training example (x,y) can then be expressed as:

$$logp(y|x, \Theta) = f[\Theta]_y - \underset{j}{logadd}(f[\Theta]_j) \tag{2.4}$$

The softmax training criterion is also referred to as cross-entropy. It doesn't consider the often important relationships between words in the sentence, so a sequence detector (such as a Viterbi detector) is commonly added to the system to enforce dependencies between predicted tags.

## 2.3 Back Propagation

Back propagation is an important algorithm used to train the weights of the system. The objective is to choose parameters which will maximize the likelihood that the system produces the desired output. Back propagation does this by first calculating a cost function (which is the log of the inverse of the probability discussed in section 2.2), then finding the partial derivatives of each parameter with respect to this cost function. By subtracting a fraction of this derivative, or gradient, from the parameters $\Theta$ while training, $\Theta$ are coaxed into a set of values which cause $f[\Theta]$ to produce values with minimum cost (maximum likelihood). Note that there are many such configurations, we are only looking for one during a single training session. This process is known as stochastic gradient descent, and is used to train the models described here.

## 2.4 Word Level Likelihood Gradients

Backpropagation can be based on the Word Level Log-Likelihood gradients such as described in section 2.2, or it can be based on Sentence Level Log-Likelihood gradients, which are described in more detail in chapter 3.

Backpropagation works by first computing the partial derivatives of the inputs of the neurons (after the sum is calculated, but before the activation is applied). Once these so called $\delta$ terms are computed, the gradients for parameters can be calculated from them. Calculation of the Cost function with respect to the output gradients of the network, will now be described.

If y is the true tag for a given word, maximizing 2.4 corresponds to minimizing:

$$C(f_\Theta) = \underset{i}{logadd}[f_\Theta]_j - [f_\Theta]_y \tag{2.5}$$

So the gradient w.r.t. $f_\Theta$ is

$$\frac{\partial C}{\partial [f_\Theta]_i} = \frac{e^{[f_\Theta]_i}}{\sum_k e^{[f_\Theta]_k}} - 1_{i=y} \ \forall i \tag{2.6}$$

Backpropagation then proceeds backwards, from output to input of the network, to calculate the rest of the necessary partial derivatives of the cost function with respect to inputs:

$$\frac{\partial C}{\partial f_\Theta^{l-1}} = \left[W^{l-1}\right]^T \frac{\partial C}{\partial f_\Theta^l} \tag{2.7}$$

Finally, the gradients of the $\Theta$ parameters, $W$ and $b$, can be calculated.

$$\frac{\partial C}{\partial W^{l-1}} = \left[\frac{\partial C}{\partial f_\Theta^l}\right] \left[f_\Theta^{l-1}\right]^T \tag{2.8}$$

$$\frac{\partial C}{\partial b^{l-1}} = \left[\frac{\partial C}{\partial f_\Theta^l}\right] \tag{2.9}$$

# Chapter 3

# Hierarchical Models

## 3.1 Daisy

The Java software, called Daisy[1], was written to train and test programmable configurations of hierarchical architectures, such as those described in [8] and implemented in [6]. Model definition parameters, such as feature definition, layer sizes, number of layers, activation functions, etc. were defined in a flexible way to support the experiments described later.

## 3.2 SENNA

The C source code for a tagging system based on Collobert et al. [8], called SENNA, was published by the authors (Collobert [6]). This software implements the forward algorithms for the components described in later chapters, and was used to test the Daisy Java architecture forward algorithms, and to supply parameters to verify the Daisy architecture.

## 3.3 Weight Initialization

It's important to initialize the parameters to be trained to something besides all zeros, since initialization to the same value will cause all artificial neurons in a layer to behave identically, thus rendering them redundant. Therefore, a common technique is to initialize the parameters to random values.

[1] Named after the song sung by artificially intelligent HAL in the movie *2001 A Space Odyssey* while his brain was being dismantled.

All parameters were initialized to a random value with a variance equal to the inverse of the square-root of the fan-in. This means for example that lookups have a variance of 1.0, but variance of the neural network is $\frac{1}{\sqrt{300}}$ for a default layer size of 300.

## 3.4 Learning Rates

When Parameters are trained in the model, a learning rate $\alpha$ is used to specify the fraction of the gradient $\Delta$ which is subtracted from the parameter to adjust it.

$$\Theta' = \Theta - \alpha \cdot \Delta \tag{3.1}$$

The sensitivity to learning rate $\alpha$ is notorious for being difficult to optimize. If it is too high, parameters may oscillate, the system might converge on a non-optimum solution, or not at all. If it is too low, the system can take very long to train. In addition, hierarchical systems have many sections, and the sections interact with each other, so learning rate sensitivities are even more pronounced than with a simple neural network.

Sometimes, the learning rate is decreased over time. Here, it was kept constant, as described in [8], and the fastest convergence occurred when the AdaGrad algorithm was used.

### 3.4.1 AdaGrad

Adagrad (Duchi et al. [9]) is a method which automatically decreases the learning rate of each individual parameter over time. It is based on the magnitude of previous corrections, so it balances out the amount of training on a per-parameter basis. It requires that a history $S$ of the sum of the gradients squared is maintained as a separate value for every parameter, which is initialized to 1 prior to training.

$$S' = S + \Delta^2 \tag{3.2}$$

And training is scaled as follows:

$$\Theta' = \Theta - \alpha \cdot \Delta * \frac{1}{\sqrt{S}} \tag{3.3}$$

In experiments described in Chapters 5 6 and 7, it was much easier to tune the systems using Adagrad than simply applying equation 3.1.

## 3.5 Viterbi Forward

The Viterbi algorithm input is a matrix formed by joining column vectors created by the neural network. Each column vector consists of scores for all possible tags, where a score represents the unnormalized log probability that a tag corresponds to the word. The tags are considered to be hidden, or latent, states. The choice of log-likelihood cost functions for training the neural network produces coerces the network into producing unnormalized log probabilities which can be converted to normalized probabilities by using the softmax equation (equation 2.4).

The neural network output matrix which is passed on to the Viterbi detector is called $f_\Theta$, and it contains elements $[f_\Theta]_{i,t}$ for every tag i and word t.

The Viterbi algorithm is initialized with a learned set of weights per tag (the I matrix), and computes the log-likelihood of transitioning from each state to the next by applying a learned set of weights from a square transition matrix $A$, with $N^2$ elements, where N is the number of tags.

By considering all possible state transitions for all words, the Viterbi algorithm evaluates all possible combinations of states for the sentence (a very large number, $N^T$, where T is the number of words in the sentence). It finds the most likely path by selecting the most likely path at each state along the way, and computes the most likely path in linear time.

Let $[j]_1^T$ be the set of all $N^T$ possible paths which can describe a tag sequence. $[x]_1^T$ is the input sentence, composed of T words. The viterbi parameters for the transition matrix and the initialization matrix are grouped together with the other system parameters, $\Theta$, and the entire group of parameters is called $\widetilde{\Theta}$.

The score of a path $[i]_1^T$ is the sum of the transition scores and the network scores, (adding

logs instead of multiplying probabilities).

$$s([x]_1^T, [i]_1^T, \widetilde{\Theta}) = \sum_{t=1}^{T} ([A]_{[i]_{t-1},[i]_t} + [f_\Theta]_{[i]_t,t}) \tag{3.4}$$

The Viterbi algorithm finds the best sequence score $s$ by computing the best path leading up to each state in the sequence and discarding paths which are suboptimal. By backtracking the states of the best path through the "trellis", the best path states (which result in the best score) can be computed:

$$\underset{\forall [j]_1^T}{argmax} \;\; s([x]_1^T, [j]_1^T, \widetilde{\Theta}) \tag{3.5}$$

## 3.6 Viterbi Cost Function (SLL)

The Viterbi cost function is based on Sentence Level Likelihood and is similar to equation 2.4, except the reference path score must be normalized by using the sum of the exponential of all path scores (the sum of unnormalized probabilities for **all** possible paths, instead of for all possible tags).

$$logp([y]_1^T | [x]_1^T, \widetilde{\Theta}) = s([x]_1^T, [y]_1^T, \widetilde{\Theta}) - \underset{\forall [j]_1^T}{logadd}(s([x]_1^T, [j]_1^T, \widetilde{\Theta})) \tag{3.6}$$

A recursion, described in Rabiner [17] and specified in [8], provides a method of computing the second term in equation 3.6. An intermediate vector, $\delta$, is calculated, which will contain the unnormalized log probability that any path through the trellis will pass through a particular state k for the particular word t. The *delta* vectors have a dimension of N, the number of tags, and they will be used for training viterbi and optionally network parameters as will be described later.

Initialize $\delta_0$ for all states k, using the Viterbi initial State Log Likelihood matrix, I:

$$\delta_t(k) = f_\Theta[x]_{k,0} + [I]_k) \;\forall k \tag{3.7}$$

Recursively compute the $\delta$s for words 1 through T:

$$\delta_t(k) = f_\Theta[x]_{k,t} + \underset{i}{logadd}(\delta_{t-1}(i) + [A]_{i,k}) \ \forall k \tag{3.8}$$

Followed by the termination:

$$\underset{\forall [j]_1^T}{logadd}(s([x]_1^T, [j]_1^T)) = \underset{i}{logadd}(\delta_T(i)) \tag{3.9}$$

Which allows us to solve 3.6 for the log-likelihood in linear time.

## 3.7 Sentence Level Likelihood Gradients And Viterbi Training

If $[y]_1^T$ is the expected tag path for a sentence (from training data),

$$C(f_{\widetilde{\Theta}}) = \underset{\forall [j]_1^T}{logadd}(s([x]_1^T, [j]_1^T, \widetilde{\Theta})) - s([x]_1^T, [y]_1^T, \widetilde{\Theta}) \tag{3.10}$$

the second half of equation 3.10 is the Viterbi score of the expected path. From equation 3.4,

$$s([x]_1^T, [y]_1^T, \widetilde{\Theta}) = \sum_{t=1}^{T} ([A]_{[y]_{t-1},[y]_t} + [f_\Theta]_{[y]_t,t}) \tag{3.11}$$

(The first half of equation 3.10 is log of the sum of all possible tag paths.)

We want to calculate the gradients for the Viterbi transition matrix $\frac{\partial C}{\partial [A]_{i,j}}$ and the gradients of the inputs $\frac{\partial C}{\partial [f_\Theta]_{i,t}}$, which can be optionally used for Sentence Level Log-Likelihood calculations. From [8], these can be calculated with a recursive procedure:

Initialize the gradients to zero.

$$\frac{\partial C}{\partial [A]_{i,j}} = 0, \forall i, j \text{ and } \frac{\partial C}{\partial [f_\Theta]_{i,t}} = 0, \forall i, t. \tag{3.12}$$

accumulate gradients over the second part of equation 3.10, $s([x]_1^T, [y]_1^T, \widetilde{\Theta})$, by traversing the expected path $[y]_1^T$,

$$\frac{\partial C}{\partial [A]_{[y]_{t-1},[y]_t}} += 1, \forall t \text{ and } \frac{\partial C}{\partial [f_\Theta]_{[y]_t,t}} += 1, \forall t. \tag{3.13}$$

Defining the first part of equation 3.10 as $C_{logadd}$,

Use recursion to compute terms $\frac{\partial C_{logadd}}{\partial \delta_t(i)}$. First, initialize $\frac{\partial C_{logadd}}{\partial \delta_T(i)}$, using softmax:

$$\frac{\partial C_{logadd}}{\partial \delta_T(i)} = \frac{e^{\delta_T(i)}}{\sum_k e^{\delta_T(k)}} \forall i \tag{3.14}$$

Then traverse the trellis from T-1 to 1 (backwards) to iteratively compute the remaining $\frac{\partial C_{logadd}}{\partial \delta_t(i)}$:

$$\frac{\partial C_{logadd}}{\partial \delta_{t-1}(i)} = \sum_j \frac{\partial C_{logadd}}{\partial \delta_t(j)} \frac{e^{\delta_{t-1}(i)+[A]_{i,j}}}{\sum_k e^{\delta_{t-1}(k)+[A]_{k,j}}} \tag{3.15}$$

Using both the forward $\delta_t$ and the backward $\frac{\partial C_{logadd}}{\partial \delta_t(i)}$, at each step we can iteratively update both the $\frac{\partial C}{\partial [f_\Theta]_{[y]_t,t}}$ (which can optionally be used to back propagate the neural network):

$$\frac{\partial C}{\partial [f_\Theta]_{i,t}} += \frac{\partial C_{logadd}}{\partial \delta_t(i)} \tag{3.16}$$

and the gradient terms of the transition score matrix A:

$$\frac{\partial C}{\partial [A]_{i,j}} += \frac{\partial C_{logadd}}{\partial \delta_t(j)} \frac{e^{\delta_{t-1}(i)+[A]_{i,j}}}{\sum_k e^{\delta_{t-1}(k)+[A]_{k,j}}} \tag{3.17}$$

This is described in [17] as calculating the $\xi_t$, and the sum of these quantities can be interpreted to be the score for transitioning from state i to state j.

The gradient for the initial score matrix I can be computed with one last step of the algorithm (or by considering it to be the first vector of an N by N+1 sized gradient matrix).

### 3.8 WLL/SLL Network Training Option

Sentence Level Log Likelihood is used to train the parameters of the Viterbi detector, but during this training, the gradient of the states for each word are calculated. These can be saved and applied directly to the neural network outputs to train the rest of the system, instead of using the softmax Word-level Log Likelihood function. This was found to improve performance significantly for Semantic Role Labeling, but didn't make much of a difference for Part of Speech labeling.

# Chapter 4

# Pre-trained Word Models

The word representation vectors, stored in a word lookup table, are an inherent part of the hierarchical models described in chapters 3, 5, 6 and 7. Like most other weights in the network, these vectors can be initialized to small random values to start with, and then trained using a supervised training algorithm. A limitation of this approach is that the training data sets for supervised learning are limited in size, partly due to the expensive human labor required to annotate them. This limited amount of training information leads to many words being encountered very few times (or never) during supervised training, which means that very little is learned about uncommon words.

There is an almost unlimited amount of free, untagged information available on the web. A method of training just the word representations from untagged databases has been very successfully applied to create a starting set of vectors that can be used to initialize a network, which is then fine-tuned with supervised training to execute a specific task. By "pre-training" these word representations using large amounts of untagged text, very informative word relationships can be inexpensively extracted, and later used as the starting point for task specific application learning, see for example Hinton et al. [12], Bengio et al. [2] and Weston et al. [21].

## 4.1 Reference Word Representations

The word representations generated by Collobert et al. [8] were created using a pairwise ranking approach ((Schapire and Singer [19]). The goal of the neural network is to compute a

higher score when given a correct phrase than when given an incorrect phrase. The network is first given the actual windowed words from the training corpus, then the same words with the center word replaced by a nonsensical word. This generates a score for the correct and incorrect phrases, which can be used as the ranking criteria:

$$\Theta \mapsto \sum_{x \in X} \sum_{w \in D} max\left\{0, 1 - f_\Theta(x) + f_\Theta(x^{(w)})\right\} \tag{4.1}$$

where $X$ is the set of all possible text windows with dwin words coming from the training corpus, $D$ is the dictionary of words, and $x^{(w)}$ denotes the text window obtained by replacing the central word in the text window by the word $w$.

The collection of 130K word representations from [8] was pre-computed and published by the authors. Each representation is a vector of fifty real numbers. This model was created by running an unsupervised neural network on the entire English Wikipedia, which took seven weeks. The text was tokenized using the Penn Treebank tokenizer script, which resulted in a dataset containing about 631 million words. The most common words from a Wall Street Journal corpus were selected from the model, and another 30,000 of the most common words from a Reuters corpus were also selected.

## 4.2 Word Model Network Similarities

Unsupervised pre-trained word representations have been noted by Mikolov et al. [15] and others to have very interesting linguistic structure. This structure was analyzed using an approach based on Network Theory. Each word can be considered to be a vertex in a 50 dimensional hyperspace, and the Euclidean distance between words provides a similarity measure which is assigned to each edge in a network. This network can be *grown* by applying edges to the word vertices, starting from the closest words, and adding edges in order of increasing distance. Figure 4.1 shows how the distance between words varies in the sorted list, and the three stages during network growth which were selected for close examination. The networks of 997 and 25,668 words were analyzed to determine the groups of words which were connected together in clusters. The

list of words in each cluster were then examined for semantic and syntactic similarity.

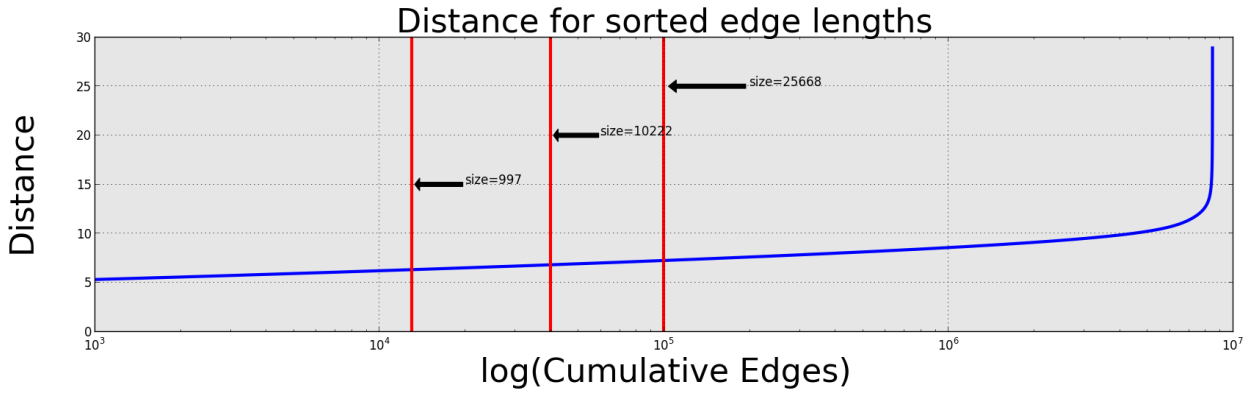



Figure 4.1: Word Representation euclidean distance for sorted edges

Word Representation euclidean distance for sorted edges, showing the size of the large components and number of edges added for the grown networks chosen for detailed investigation.

An agglomerative clustering algorithm, using the raw euclidean distance[1], was run on the raw word representation data for the formed word clusters.

The vectors which define the word representations (vertices) within these components are plotted together in Figure 4.2. Distinct bands of similar strength show the common traits between vectors which are responsible for the similarities which cause them to be grouped together within the network.

For each stage of network growth there are very small components of two or three as well as the large component. The components smaller than the large component can easily be seen to show correlation between words. A sampling of the words within distinct components prior to large component assimilation is shown in Table 4.2.

The large components for each stage examined in detail are shown in Figure 4.3. The large component which is formed after 13,000 edges have been added is shown in Figure 4.3(a). The previously distinct components which have been merged into the large component are still easily seen within the structure. A larger pattern is visible where General Biology connects to Anatomy, Medical Conditions, Drugs, and finally Chemicals.

[1]Cosine similarity was also used, with similar results.

| Summary | Size | Edges | Words |
|---|---|---|---|
| Amino Acids | 22 | 12200 | adenine, aspartate, choline, cysteine, cytosine, dmso, glycerol, guanine, heme, histidine, lipid, lysine, pyridine, pyrimidine, ribose ... |
| Anatomy | 62 | 9000 | abdomen, anus, aorta, cecum, cerebellum, cerebrum, cervix, clavicle, clitoris, cochlea, conjunctiva, cornea, cytoplasm, cytoskeleton, dermis ... |
| Animals | 91 | 7800 | alligators, amphibians, aphids, arthropods, bivalves, boars, centipedes, cephalopods, cetaceans, clams, cockatoos, cockroaches, copepods, corals, cormorants ... |
| Basic Chemicals | 26 | 8000 | arsenide, azide, bromide, carbonate, chlorate, chloride, chlorine, deuterium, fluoride, fluorine, helium, hydrate, hydroxide, hypochlorite, iodide ... |
| Biological Parts | 38 | 9000 | barbels, blotches, bracts, branchlets, carpels, catkins, eyebrows, forelegs, forewing, forewings, fronds, hairs, hindwings, incisors, inflorescences ... |
| Biological | 19 | 8300 | astrocytes, chloroplasts, cytokines, erythrocytes, fibroblasts, hepatocytes, histones, leukocytes, lysosomes, macrophages, mitochondria, monocytes, nematodes, neurotransmitters, neutrophils ... |
| Chemical Compounds | 13 | 8000 | alkaloids, allergens, antibiotics, medications, pathogens, starches, sugars, tannins, toxins, vaccinations, vaccines, vitamins, yeasts ... |
| Chemicals | 36 | 8000 | abrasives, acids, alcohols, aldehydes, alkenes, amines, anions, antioxidants, carbons, cations, esters, ethers, foams, glycoproteins, halides ... |
| Chemicals | 78 | 8600 | acetone, allyl, ammonium, arsenide, azide, barium, benzyl, beryllium, bilirubin, bismuth, boron, bromide, butyl, caesium, carbonate ... |
| Drugs | 56 | 8000 | acetaldehyde, acetaminophen, acetylcholine, aldosterone, amphetamines, analgesics, anticonvulsants, antidepressants, antihistamines, antipsychotics, barbiturates, benzodiazepines, bupropion, cannabinoids, corticosteroids ... |
| Foods | 112 | 7000 | almonds, apples, apricots, avocados, bananas, beans, beetroot, berries, blackberries, blueberries, breadfruit, breads, buckwheat, burgers, burritos ... |
| Medical Conditions | 126 | 11000 | abnormalities, adenocarcinoma, adenoma, anaemia, anemia, arrhythmias, arthritis, atherosclerosis, atresia, bloating, bradycardia, bronchitis, cancers, carcinoma, cholera ... |
| Sports Teams | 74 | 11000 | 0ers, aeros, alouettes, astros, badgers, beavers, bengals, bisons, bobcats, braves, broncos, bruins, buccaneers, buckeyes, bulldogs ... |

Table 4.1: Selected, representative components during network growth. The size and number of edges added are shown along with the first fifteen words from each component (in alphabetical order). These components were selected because they are eventually joined and become part of the large component examined after 13,000 edges have been added.

| Summary | Size | Words |
|---|---|---|
| Eastern Places | 616 | achaea, adilabad, afghanistan, agra, ahmedabad, ajmer ... |
| UK Places | 344 | aberdare, aberdeenshire, abergavenny, abingdon, airdrie, aldershot ... |
| First Names | 213 | abreu, adolfo, aguilar, aguirre, agustin, alberto ... |
| US Places | 194 | albany, allentown, altoona, amarillo, anaheim, arizona ... |
| English First Names | 191 | aaron, abby, abigail, adam, alan, alex ... |
| Peoples | 138 | abbasids, alamanni, albanians, almoravids, arabs, armenians ... |
| Occupations | 134 | anatomist, anthropologist, apologist, archaeologist, archeologist, architect ... |
| Medical | 109 | abdominal, adrenal, amniotic, articular, atherosclerotic, atrial ... |
| Computing Terms | 101 | .net, adsl, adsl0+, amd0, amigaos, asp.net ... |
| Arabic Names | 70 | 'ali, 'amr, 'ali, abdullah, abu'l, ahmad ... |
| TV Stations | 67 | cfcf, kabc, kcbs, kcop, kcra, kdka ... |
| Kings | 60 | aethelfrith, alexios, amalric, andronikos, antigonus, antiochus ... |
| Specialties | 58 | anesthesiology, anthropology, astronomy, astrophysics, audiology, bacteriology ... |
| Transformers | 56 | airazor, anubis, apokolips, astrotrain, blackarachnia, blitzwing ... |

Table 4.2: The largest 14 of more than 2000 smaller components after adding 13,000 edges. Visual inspection shows remarkable correlation within components. The Summary term was determined manually.

| Size | Words |
|---|---|
| 137 | 0pac, aaliyah, ac/dc, aerosmith, akon, anastacia ... |
| 93 | aikido, angling, aquatics, archery, backpacking, badminton ... |
| 69 | abnormally, alarmingly, appreciably, astonishingly, badly, comically ... |
| 54 | aldiss, benchley, bogdanovich, borgnine, braff, branagh ... |
| 49 | aharon, akiva, aryeh, avraham, avrohom, baruch ... |
| 47 | a.b., b.a, b.a., b.s, b.s., b.sc ... |
| 36 | baserunners, batters, boxers, completions, defencemen, defensemen ... |
| 34 | aquabats, ataris, banshees, beatles, bosstones, byrds ... |
| ... | ... |
| 3 | accomplish, achieve, attain |
| 3 | achieving, attaining, gaining |
| 3 | actresses, artistes, heroines |
| 3 | addicts, traffickers, trafficking |
| 3 | adhere, conform, revert |
| 3 | adirondacks, catskills, ozarks |
| 3 | adorn, adorning, decorate |
| 3 | adorns, encloses, occupies |
| 3 | adrift, aground, ashore |
| 3 | aeronautical, aerospace, automotive |
| 3 | african, asian, korean |
| 3 | africans, asians, koreans |
| 3 | afterward, afterwards, thereafter |

Table 4.3: A sampling of large (but not largest) components and small components after 100,000 edges are added

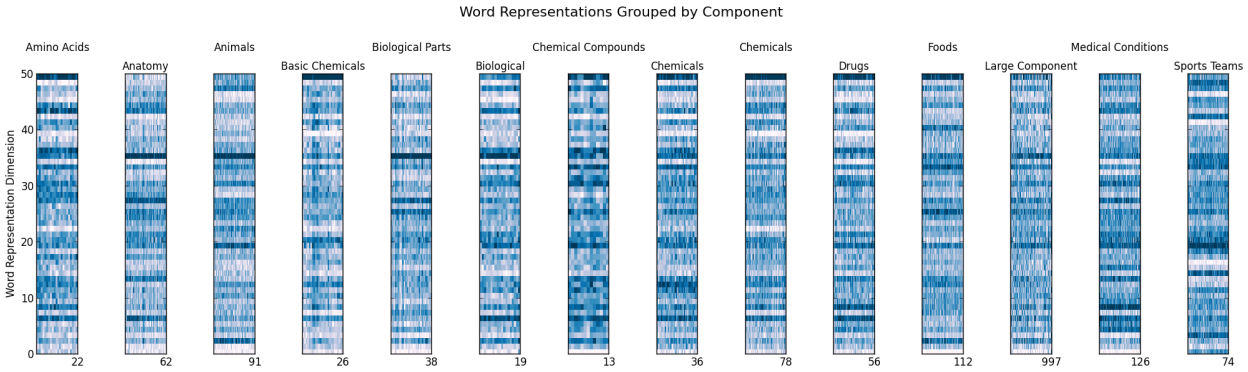
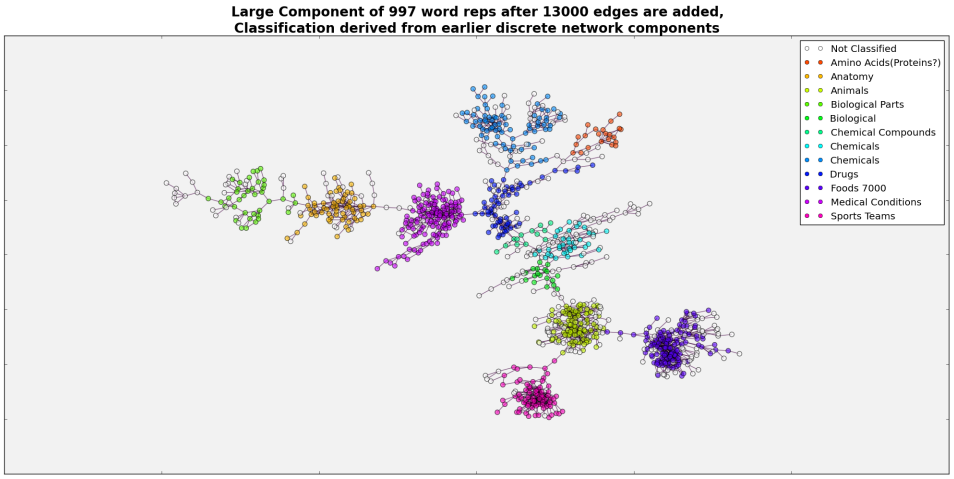



Figure 4.2: Word Representations grouped by Component

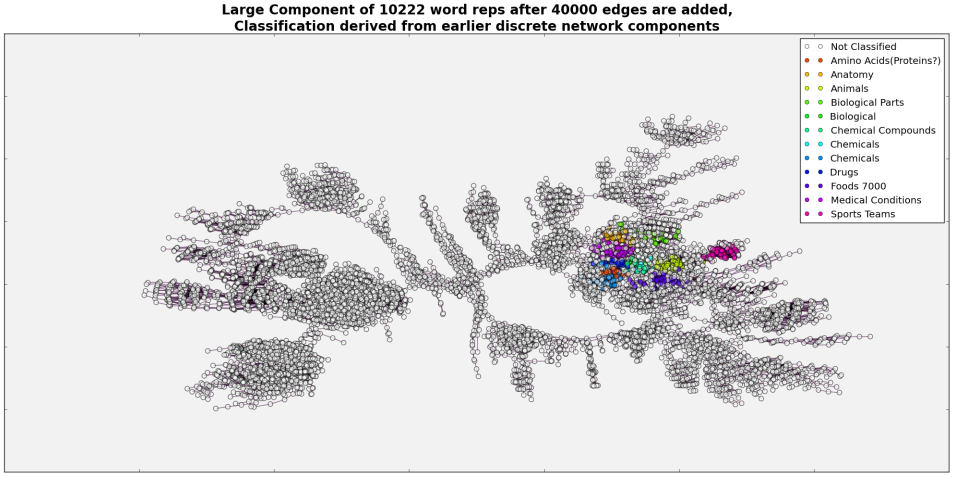
Word Representations grouped by Component for the selected, representative early components during network growth. Each representation is a 50 element real vector. Small euclidean distances, which represent the close similarities responsible for component formation, can be seen as light or dark bands across each group.

The hierarchical community structure of the large component after adding 13,000 edges is clearly visible in in Figure 4.4(a). The path to each word from the root of the dendrogram is used to sort the word order, and the words are colored using the same color scheme as in Figure 4.3(a). The illegible word labels under the dendrogram are colored based on the scheme from Figure 4.3(a). The gray colored words were added after the smaller components were identified, but were confirmed to be highly correlated to the nearby word groups. For each pair of adjacent words in the dendrogram, the logarithm of the probability of the highest common node between the words gives a good function to discern boundaries between clusters.

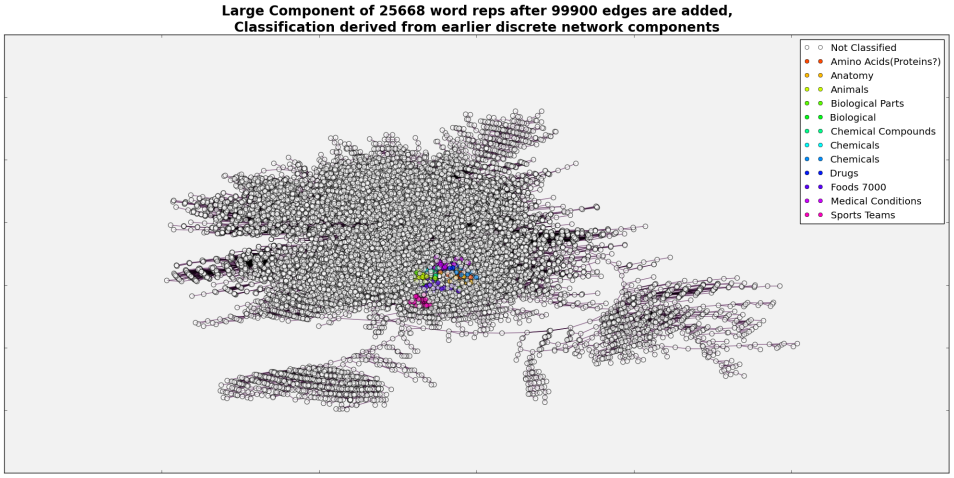
Representative dendrograms for the large component after adding 100,000 edges is shown in Figure 4.4(b). Note that the reference component structure remains visible, shown by the colors of the words. The hierarachy within the large component contains considerable correlation in the words as before.




(a)

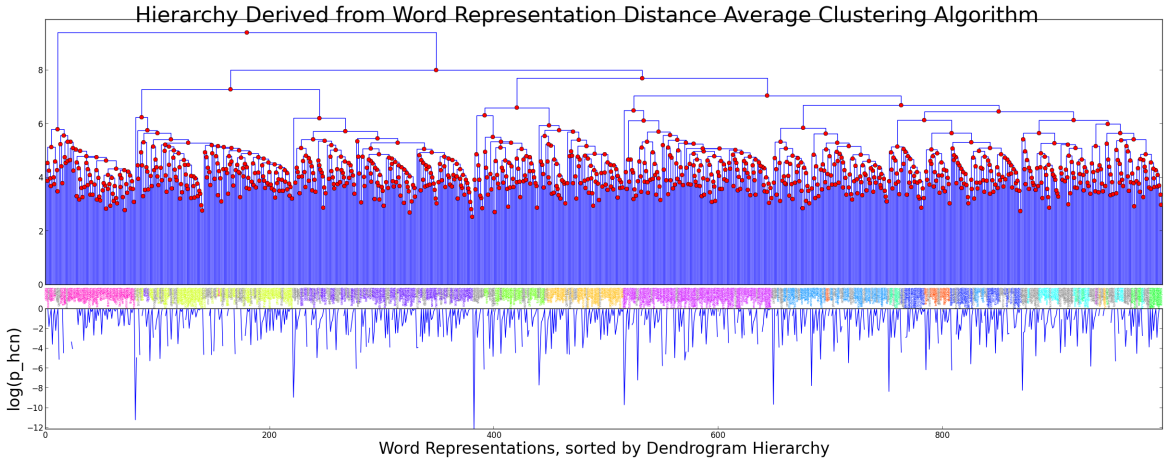



(b)

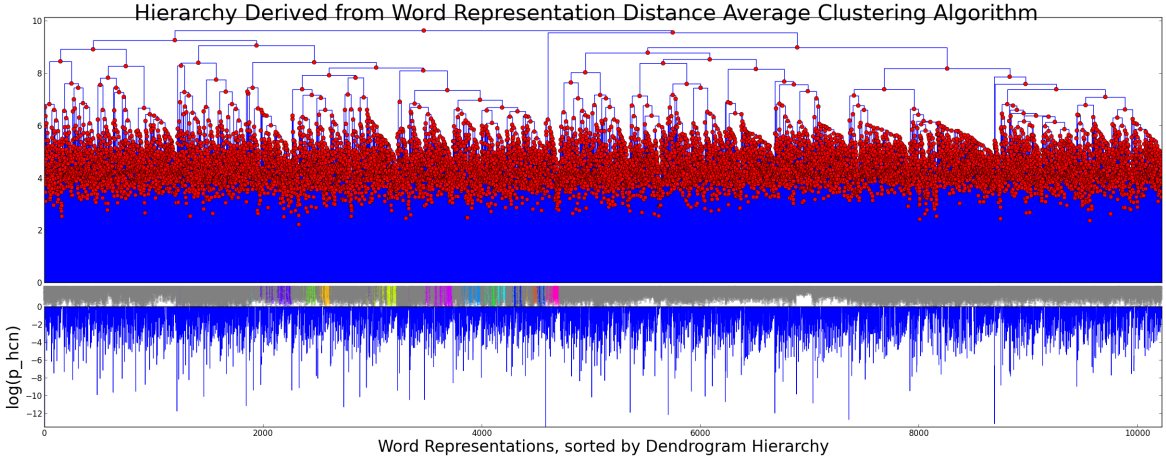



(c)

Figure 4.3: The large component as the network is grown. The structure of the colored groups, corresponding to the selected early distinct components, is especially visible in the top diagram due to the networkx Spring-Layout Visualization algorithm.




(a)




(b)

Figure 4.4: Dendrograms, Colored Word Labels, and $Log(p_{highest\ common\ node})$ for the large components after 13,000 and 100,000 edges have been added, using an average hierarchical clustering algorithm. Note the reference colored clusters.

| Summary | Size | Words |
|---|---|---|
| French-First-Names | 36 | ambroise, laurent, etienne, gaspard, guillaume, philippe ... |
| Performers | 127 | goulet, soderberg, timberlake, kossoff, jansch, metheny ... |
| Actor/Comedian | 802 | mamet, frankenheimer, leguizamo, belushi, urich, cleese ... |
| Italian | 350 | pieve, fuori, storia, venta, comunidad, provincia ... |
| Asian | 299 | khao, khlong, petaling, putra, tanjung, jebel ... |
| Ancient Scripts | 420 | luwian, runic, gurmukhi, prakrit, cuneiform, hieratic ... |
| Nordic | 1406 | gunnar, eero, torsten, ulrika, olof, vaclav ... |
| Places | 97 | ouro, ribeirao, cagayan, camarines, ilocos, azul ... |
| Europeans | 51 | neurath, adenauer, kautsky, darlan, donitz, kluge ... |
| Italian-First-Names | 904 | massimiliano, antonello, luchino, cesare, sigismondo, lodovico ... |
| Military | 58 | jg, cortdiv, tf, subron, tg, desdiv ... |
| Rivers | 160 | berbice, demerara, essequibo, araguaia, luapula, orinoco ... |
| European Places | 107 | guingamp, triestina, brugge, heerenveen, bochum, wolfsburg ... |
| French-Geography | 2494 | bourg, riviere, puy, baie, pont, chalons ... |
| Info Processing | 84 | indexing, querying, archiving, cataloging, authoring, editing ... |
| Reasoning | 575 | extrapolated, deduced, inferred, positing, speculated, surmised ... |
| Governing | 852 | organises, organizes, governs, administers, oversees, assesses ... |
| Sports | 51 | cricketer, sportsperson, striker, footballer, goalkeeper, athlete ... |
| Relationships | 439 | chauffeur, landlady, classmate, colleague, godson, grandnephew ... |
| Ancient Scripts | 453 | demotic, indic, masoretic, peshitta, devanagari, ge'ez ... |
| Politics | 1226 | oligarchic, absolutist, orleanist, authoritarian, imperialist, monarchical ... |
| Politics | 53 | nationalistic, bureaucratic, repressive, classless, pluralistic, ethical ... |
| Musical Styles | 548 | klezmer, fado, zouk, soca, norteno, rumba ... |
| TV/Movies | 377 | lwt, utv, adv, mgm, miramax, pixar ... |
| Arts | 570 | philology, pedagogy, filmmaking, musicology, ethnomusicology ... |
| Sci Fi Chars | 84 | cylon, shi'ar, goa'uld, sgc, tau'ri, sodan ... |
| Politics | 112 | nda, anc, oas, sandinistas, fsln, mnr ... |
| Education | 97 | lecturers, professorships, seminaries, yeshivot, students, teachers ... |
| Villians | 751 | savages, fiends, trolls, wraiths, cultists, elementals ... |
| Fun Places | 302 | cabarets, discotheques, buskers, nightclubs, carnivals, circuses ... |
| Lawyers | 230 | remuneration, liability, reimbursement, depreciation, refinancing, liquidity ... |
| Dominance | 62 | invasions, conquests, incursions, overlordship, suzerainty, vassalage ... |
| Groups | 1192 | armies, legions, expatriates, immigrants, emigrants, migrants ... |
| Weapons? | 464 | asroc, exocet, ka0, me0, be0, ac0 ... |
| Tech Acronyms | 1155 | ejb, jsr, iso, fips, iso/iec, rsa ... |
| Mathematicians | 498 | baire, heyting, kronecker, pontryagin, stefansson, alfven ... |
| Currencies | 119 | reais, rmb, usd, usd0, gbp, dkk ... |
| Direction | 60 | lateralward, medialward, downwards, inwards, outwards, longitudinally ... |
| Families | 86 | fagaceae, acanthaceae, myrtaceae, solanaceae, pieridae, rutaceae ... |
| Rugby/Leagues | 219 | nswrl, vfa, nrl, sanfl, wafl, ajhl ... |
| Art | 57 | carved, painted, sculpted, embellished, frescoed, incised ... |
| Physics | 31 | ionospheric, mesoscale, geomagnetic, gravitational, neutrino, dielectric ... |
| Bird Traits | 224 | beaked, crested, horned, pygmy, endangered, migratory ... |
| Birds/Plants | 388 | macaw, pipit, pinyon, sedge, wattle, sundew ... |
| Plants | 1087 | cupressus, rhus, cyathea, litoria, brassica, opuntia ... |
| Mechanical | 26 | machined, riveted, welded, absorbent, lubricated, crimped ... |
| Cellular | 2551 | monoclonal, recombinant, diploid, eukaryotic, mammalian, prokaryotic ... |

Table 4.4: Communities within the 25,668 word large component, after 100,000 edges have been added, found from the betweenness linkage. The threshold for group boundaries is log(p) = 10.0. Very intuitive linguistic groupings appear in the hierarchy.

# Chapter 5

# Part of Speech Tagger

Part of speech tags are labels which are meant to categorize words in a sentence based on their role in that sentence. Noun, verb, and adjective are familiar examples of this type of tag. A more complete set of tags is defined in Marcus et al. [14], which was used to tag more than a million words from the Wall Street Journal. This corpus is called the Penn Treebank, first annotated by linguists and released in 1992, and it continues to be enhanced and used for NLP research and development.

A Part of Speech (POS) Tagger is a program that tries to label each word in a sentence with a POS tag. Sentences, and the POS tags for words in the sentence, are used to train and test this program. A benchmark used by Toutanova et al. [20] describes dividing the Penn Treebank into sections, 0-18 are used for training, while 19-21 are for validation and 22-24 for testing. This benchmark was used to evaluate the performance of the POS tagger described in in Collobert et al. [8], and was replicated to test the performance of the Daisy POS Tagger.

## 5.1 Data Preparation

The Penn Treebank files describe a parse tree for each annotated sentence, which includes the part of speech tags for each word. A perl script was used to flatten this hierarchy and then create a database with just words and POS tags, modeled after the OpenNLP standard.

For example, a sentence from the file wsj_2450.mrg in the Penn Treebank that looks like this:

```
( (S
```

```
(NP-SBJ-1 (DT The) (NNS results) )
(VP (VBD were)
  (VP (VBN announced)
    (NP (-NONE- *-1) )
    (SBAR-TMP (IN after)
      (S
        (NP-SBJ (DT the) (NN stock) (NN market) )
        (VP (VBD closed) )))))
(. .) ))
```

was stripped of hierarchical information to extract pairs of word/tags,

```
Words:      The results were announced after the stock market closed .
POSTags:    DT  NNS     VBD  VBN       IN    DT  NN    NN     VBD    .
```

and then the pairs are joined with an underscore to create an OpenNLP format:

```
The_DT results_NNS were_VBD announced_VBN after_IN the_DT
stock_NN market_NN closed_VBD ._.
```

Using a standard format for the input to the program means that other databases in this format can be used directly for training and testing.

## 5.2 POS Forward System diagram and description

The architecture for the full POS Forward System, with default options, is shown in figure 5.1.

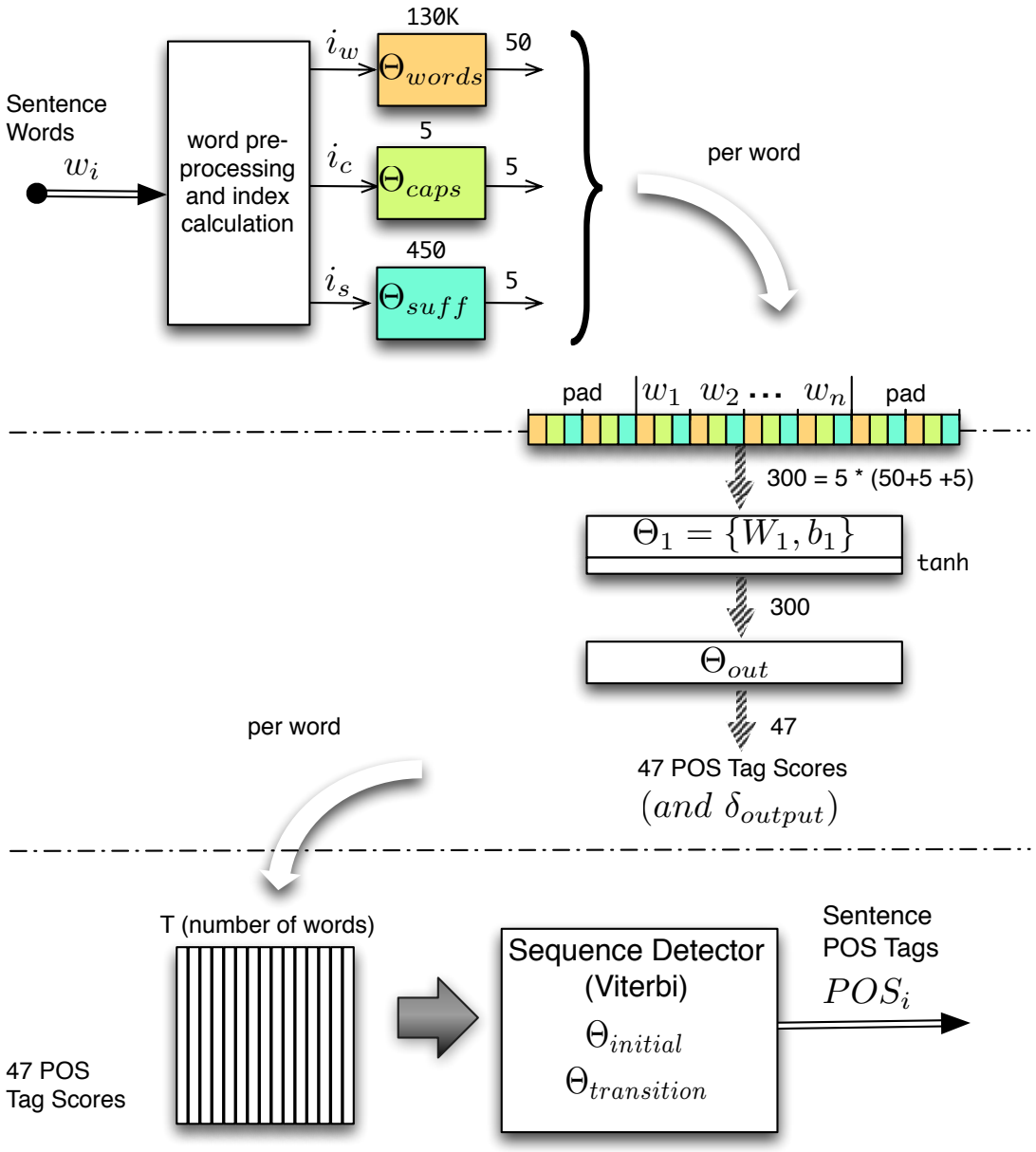



Figure 5.1: POS Tagger Architecture

The POS Tagger is composed of three main sections: Lookup, Neural Network, and Viterbi. The diagram describes a system with a window size of 5, a single Neural Network Layer, an output layer size of 300, and the lookup table sizes from Collobert [6], all of which are parameterized in the Daisy software.

The input to the tagger is a sentence, which is a list of words $w_i$ from $w_1$ to $w_T$. The output

is the list of predicted POS tags for each word, $POS_i$. The upper third of the diagram shows the Feature Lookup, which runs once for the whole sentence, and which creates the Feature Vector, depicted as the striped color bar. The middle section of the tagger is the Neural network, which contains the neural network and output layers, and which is run once per word. The results of this middle section are accumulated in a large matrix which is the input to the Viterbi sequence detector (Viterbi). The Viterbi is run once for each sentence, and determines the most likely sequence of tokens, $POS_i$.

```
getPOSTags(sentence)
1  sentenceFeatureVector = GetSentenceFeatureVector(sentence)
2  viterbiInMatrix = RunNNForward(sentenceFeatureVector)
3  POS = RunViterbi(viterbiInMatrix)
4  return POS
```

Figure 5.2: POS Tagger Forward Pseudocode

```
RunNNForward(sentenceFeatureVector)
1  Layers = system.Layers //
2  for each wordIndex in wordCount
3        Layers[0].theta = extractVector(FeatureVector, wordIndex)
4        WordPOSScores = singleCycleRunForward(Layers)
5        insertColumn(outputMatrix, wordIndex, WordPOSScores)
6  return outputMatrix
```

Figure 5.3: RunNNForward Pseudocode

```
singleCycleRunForward(Layers)
1  lastLayer = length(Layers)-1
2  for i in 0 to lastLayer-1
3        z = Layers[i].theta * Layers[i].activation // matrix multiplication
4        if (Layers[i] has BiasTerm)
5              z = z + Layer[i].bias);
6        Layers[i+1].activation = elementWiseActivation(z) // such as tanh
7  return Layers[lastLayer].activation;
```

Figure 5.4: singleCycleRunForward Pseudocode

### 5.2.1 Lookup Section

The upper portion of figure 5.1 shows the process of extracting features from the words. At this level, textual information is converted to numeric **features** suitable for further algorithmic processing. The numeric information from the features for each word is concatenated together to form one long Feature Vector, shown in the diagram as a multicolored set of rectangles.

Daisy allows new features to be defined, and the number of features is also programmable. The four types of features tested for the POS tagger were:

- word representation
- capitalization
- suffix(2)
- suffix (2,3,4)

#### 5.2.1.1 Word Representations

The input data provided by CoNLL has already gone through some initial tokenizing. This prevents tokenization differences of different systems from influencing the results, which are meant to allow comparison of the POS tagging architecture itself. The Daisy pre-processor does not split hyphenated input words, so each input word will result in a single pre-processed word. Numeric values are collapsed to the single common **0** token, and words are lower-cased to create a word representation lookup word. A vector of 50 floating point values is produced by this lookup using the default architecture. If the word is in the word representation table, the associated vector is output, otherwise the vector corresponding to the special token **UNKNOWN** is output.

#### 5.2.1.2 Capitalization

The caps feature is used to preserve information about each word implied by its upper case letters. Prior to lower casing, each word is checked for all capitals, initial capital, any capital, or no capitals, and this criteria is used to lookup a vector (default length 5) from the caps table.

#### 5.2.1.3 Suffix

Suffix (2) uses the last two letters of a word as a lookup, and suffix (2,3,4) uses up to the last four letters of each word as a lookup. These tables are initially populated by analyzing the training data and creating entries for the most common suffixes, 450 for the suffix(2) and 1000 for the suffix(234).

Each feature lookup table also contains an entry for **PADDING**. In order to allow the window to extend beyond boundaries of the sentence for early and late words the Feature Vector is padded with the **PADDING** value from each lookup table. The **PADDING** values are also trained.

### 5.2.2 Neural Network Section

The neural network consists of one non-linear layer with a default size of 300, and the output layer, also with a default size of 300.

Equation 2.1 specifies a consistent method for iterating forward through the layers, namely to multiply the previous layers weight matrix by the previous layers activation function in order to calculate the current layer's pre-activation values. Then the current layer's activation function is applied. To enable this method to be applied consistently, and for similar reasons during back propagation, two simple layers are added to the system at the beginning and end of the neural network pipeline, making a total of four, as shown in figure figure 5.5.

The windowed Feature Vector for each word is the input to the network, stored in the weights section of the bookkeeping layer 0. When multiplied by an activation of Unity, this results in an activation for layer 1, whose activation function is also Unity, of the same input (the windowed feature vector).

The output layer provides a score for each possible tag. After running all words through the system, a matrix of tags X words is created, which will be used as the input to the Viterbi sequence detector.

### 5.2.3 Sequence Detection (Viterbi)

The Viterbi algorithm from line 3 of figure 5.2 is represented by the bottom third of figure 5.1. Its input is a matrix which consists of a vector of POS scores for each word. The algorithm is initialized with a learned set of weights per tag, and computes the log-likelihood of transitioning from each state to the next by applying a learned set of weights from the transition matrix.

## 5.3 Training and Forward Model Creation

The model is created based on architectural parameters, and trained based on training parameters. The model is eventually stored to disk along with all trained parameters, and can be

read back in for further training or for POS tagging.

```
trainPOSTagger(sentence, referenceTags, useWLL)
 1  // multiple Arrays represent Gradient Sum
 2  Set GradientSum arrays to zero
 3  tags = getPOSTags(sentence)
 4  // computing the cost requires computation of delta
 5  Vdelta = computeCostAndVDelta(tags, referenceTags)
 6
 7  VGradients = computeVGradients(VDelta)
 8  updateVWeights(VGradients)
 9
10  for wordIX in 1 to wordCount
11        singleCycleRunForward
12        if useWLL
13              Deltas = computeDeltasWLL(referenceTags[wordIX])
14        else
15              Deltas = computeDeltasSLL(Vdelta[wordIX])
16        Gradients = computeGradients(Deltas)
17        GradientSum = AccumulateGradients(Gradients, GradientSum)
18
19  updateWeightsFromGradientSum(GradientSum)
20
21  updateFeatureLookups(GradientSum)
```

Figure 5.6: POS Tagger Training Pseudocode

### 5.3.1 Training Algorithm

Figure 5.7 shows the five stages involved in training the $\Theta$ weights of the system. Figure 5.6 is a pseudocode description of the training process.

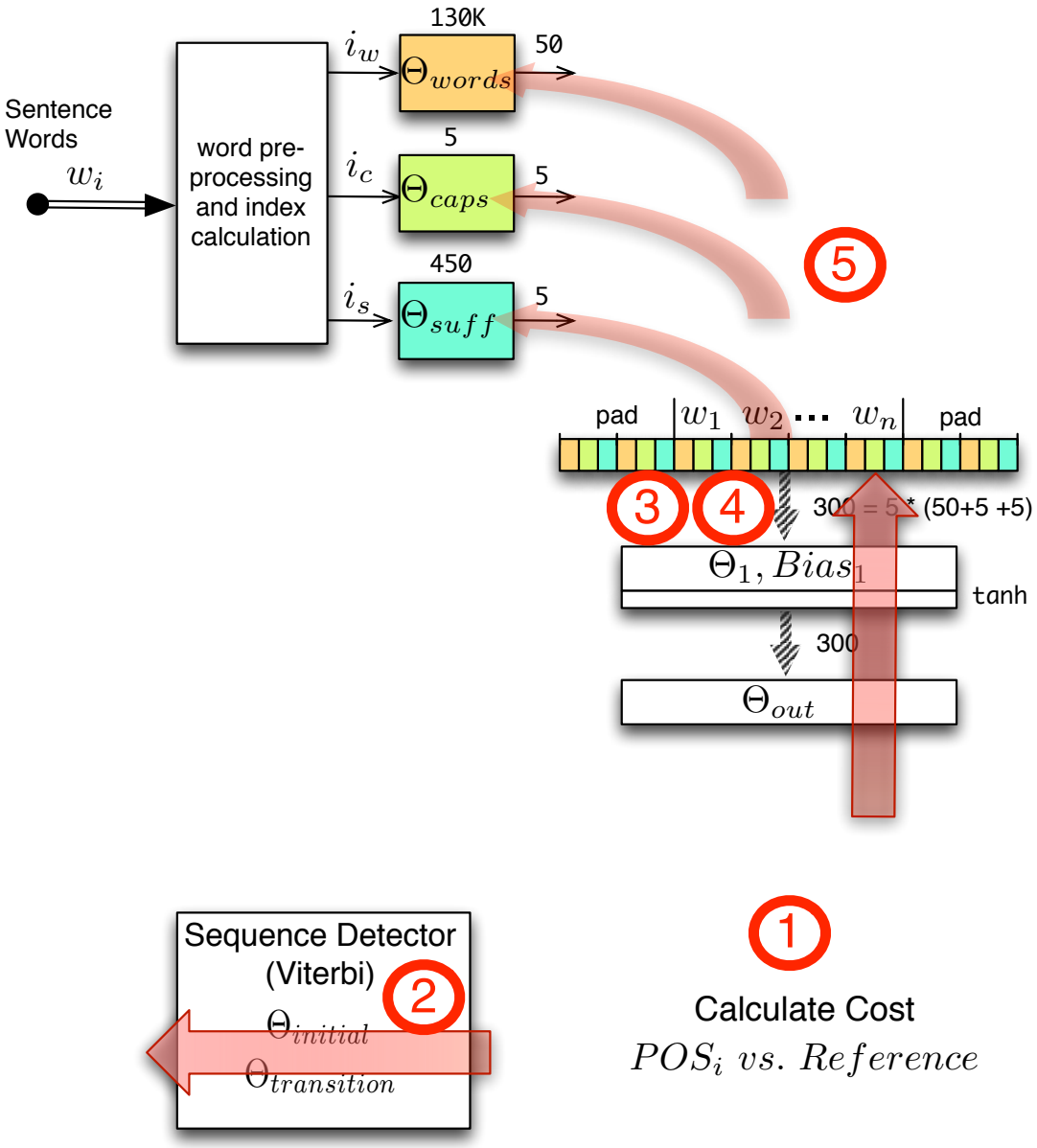



Figure 5.7: POS Training

After a forward pass to compute the predicted tags, Training the POS Tagger is done in five steps:

1: Calculate the SLL cost using equation 3.6.

2: Backpropagate to calculate gradients and update Viterbi weights.

3: for each word, backpropagate to calculate gradients and accumulate a sum of updates for the Output Layer, Neural Network, and Feature Vector.

4: After all words are processed, update Output and Neural Network weights using the update sum from step 3.

5: Update Lookup table weights.

The five steps shown in Figure 5.7 involved in training the POS model are now described.

#### 5.3.1.1 Step 1: Cost Calculation

The Viterbi parameters for initial score and transition parameters use the Sentence Level Log-Likelihood Cost, which is calculated using equation 3.6 and shown in lines 1-5 of Figure 5.6. This calculation requires computation of intermediate terms referred to as the $\delta$s, which can be re-used for the gradient calculation in step 2. The iterative algorithm used to calculate the SLL cost is based on the forward algorithm and is described in section 3.6.

#### 5.3.1.2 Step 2: Viterbi Backpropagation

The Viterbi gradients are calculated using equation 3.17, and is shown in lines 7-8 of Figure 5.6. This calculation reuses the saved $\delta$s from the SLL Cost calculation. The iterative algorithm used to calculate the gradients is described in section 3.7.

#### 5.3.1.3 Step 3: Neural Network Backpropagation

The Neural Network gradients are calculated using the classic back propagation algorithm. They are shown in lines 10-17 of Figure 5.6. First, based on maximizing the word level log-likelihood cost function (equation 2.4), the gradients of the inputs of the output neurons are calculated from equation 2.6.

Equation 2.7 is then applied iteratively, from the end of the network back to the Feature Vector, to compute the partial derivatives of the cost function with respect to the inputs. Based on the input gradients, the actual parameter gradients can be calculated using equations 2.8 and 2.9.

The gradients are not applied word for word, since that would alter the behavior of the forward network. Instead, they are accumulated as each word is processed, an later the sum of the gradients is used for updating parameter weights.

#### 5.3.1.4 Step 4: Weight Updates

Line 19 of Figure 5.6 refers to updating the neural network parameter weights. The sum of the gradients is used for this update, and it is applied to all parameters in the neural network (excluding feature tables). Equations 3.1 3.3 are used based on whether adaGrad is specified or not.

#### 5.3.1.5 Step 5: Lookup Weight Updates

Line 21 of Figure 5.6 refers to updating the feature lookup table parameters. The features for each word are retrieved from the table, modified, and stored back, which allows words which occur more than once in a sentence to be trained based on all occurrences. Equations 3.1 3.3 are used based on whether adaGrad is used or not.

## 5.4 Architectural Definition Parameters

Architectural options (defined at training time), with default values, are shown in 5.1.

| Architectural Option | Default |
|---|---|
| window size | 5 |
| features (words, caps, suffix2 or suffix234) | words, caps, suffix2 |
| word feature size | 50 |
| caps feature size | 5 |
| suffix feature size | 5 |
| neural network layers | 1 |
| neural network layer size | 300 |
| output layer size | 300 |

Table 5.1: POS Tagger Architectural Options

### 5.4.1 Training Parameters

Training options include:

- words: pretrained/random
- Training Method WLL/SLL
- AdaGrad usage (per feature or layer)
- Learning rate (per feature or layer)

The Viterbi detector is trained in a way that is consistent with the overall network. A cost function based on comparing the reference sequence to the predicted sequence is used. Then the partial derivatives of the input of each viterbi state are calculated (the $\delta$'s). Finally, the partial derivatives of each viterbi transition and initialization parameter are calculated and applied in the same fashion as the gradients for a neural network.

Training of the Neural Network and the Lookup parameters can be done using Word Level Likelihood (WLL), or Sentence Level Likelihood (SLL). In the case of WLL, back annotation of the parameters in the Neural Network and the Lookup section can be done for each word independently. In the case of SLL, the gradients calculated during Viterbi training are used during back-propagation as the $\delta$ layer for the Neural Network output layer.

Feature table options:

- Initialize to pre-trained values, or start with random values.
- Train, or not
- use AdaGrad, or fixed.
- learning rate

This level of control makes it easy, for example, to test learning rates for one section at a time, making the search process much faster because of earlier model convergence.

## 5.5 Experimental Results

First, a sort of crude grid search was used to find a decent set of parameters, ie. one that converged quickly to a solution which tested reasonably well. The trained parameters were saved in a model file for later use.

Starting with the retrieved trained parameters, an investigation was performed to find the best parameters for each section. This was done by keeping the system weights fixed for all but the section being tested.

In order to make this process go more quickly, development testing was periodically done on the model during training. This helped to determine how much training was necessary to be able to reliably judge the effect of different strategies and hyper parameters on the resulting performance of the model. A typical set of "learning curves" are shown in Figure /refPOSLearningCurves

The best AdaGrad learning rates are shown in Table 5.2.

| Trained Section | AdaGrad Learning Rate | AdaGrad Ratio | Suggested Ratio in Collobert et al. [8] |
|---|---|---|---|
| words | 1.00 | 1 | 1 |
| caps | 0.6 | 6/10 | 1 |
| suffix | 0.6 | 6/10 | 1 |
| nn layer | 0.025 | 1/40 | 1/300 |
| output layer | 0.025 | 1/40 | 1/300 |
| viterbi | 0.05 | 1/20 | 1/50 |

Table 5.2: POS Tagger AdaGrad Learning Rates

Word feature learning rates can make a big difference on the performance of the system, and they were varied in many experiments to arrive at the best set of learning rates shown in table 5.2. table 5.3 shows the variation in performance when the word features learning rates were scaled.

| | 1.0 | 2.0 | 0.5 |
|---|---|---|---|
| Daisy SLL, suffix(234) | 97.24% | 97.13% | 97.19% |

Table 5.3: POS Test Accuracy for various word feature learning rates

Daisy performance with Random word representations is slightly better (0.52%) than the reported SENNA [6] numbers, but the best performance with pre-trained words is slightly worse (0.05%). Also, it seems that more suffix feature information improves Random performance significantly, making the pre-trained difference fairly small (0.11%) (table 5.4).

| System Description | Random | Pre-trained | Difference |
|---|---|---|---|
| SENNA [6] SLL, nosuffix | 96.37% | 97.20% | +0.83% |
| Daisy SLL, nosuffix | 96.66% | 97.11% | +0.45% |
| Daisy SLL, suffix(2) | 96.99% | 97.21% | +0.22% |
| Daisy SLL, suffix(234) | 97.13% | 97.24% | +0.11% |

Table 5.4: POS Test Accuracy Improvement with Pre-Trained Words

The suffix feature lookup table stores a small (5) vector for each 2 letter suffix found by gathering statistics of the words in the POS training set. A more complex suffix feature, called suffix234, maintains a set of vectors for common 4, 3 and 2 letter suffixes. If the 4 letter suffix is not found, the three letter suffix is used, followed by 2 letters. An experiment to determine the effect of various suffix size from 2 to the 2,3,4 scheme was run, and results are shown in table 5.5. The suffix2 feature helps per word accuracy by about .1%, but the suffix234 improves over suffix2 just slightly, about 0.03%.

| System Description | Test Accuracy |
|---|---|
| Benchmark | 97.24% |
| SENNA [6] suffix(2) | 97.29% |
| Daisy SLL WR, nosuffix | 97.11% |
| Daisy SLL WR, suffix(2) | 97.21% |
| Daisy SLL WR, suffix(234) | 97.24% |

Table 5.5: POS Test Accuracy

Word Level Likelihood for POS Neural Network training seems to perform only very slightly better than Sentence Level Likelihood. It makes much more of a difference for other tasks (table 5.6).

| System Description | SLL | WLL | Difference |
|---|---|---|---|
| Daisy pre-trained, nosuffix | 97.11% | 97.06% | -0.05% |
| Daisy pre-trained, suffix(2) | 97.21% | 97.22% | +0.01% |
| Daisy pre-trained, suffix(234) | 97.24% | 97.15% | -0.09% |
| Daisy random, nosuffix | 96.66% | 96.65% | -0.01% |
| Daisy random, suffix(2) | 96.99% | 96.96% | -0.03% |
| Daisy random, suffix(234) | 97.13% | 97.02% | -0.11% |

Test Accuracy is generally slightly worse for Word Level Likelihood as compared to Sentence Level Likelihood driven training.

## 5.6 Part of Speech Tagger Development

The Part of Speech Tagger allowed fairly quick development and experimentation of a basic system and to understand how gradient checking could be used to confirm back propagation. It was also used to explore the use of training sections within the component separately as a way of

solving the bigger problem, by comparing fixed vs. adaGrad weight training methods to accelerate convergence and minimize learning rate sensitivities. The Viterbi training algorithm was developed and tested using this relatively simple environment, so that it could be later just dropped into the much more complex semantic role labelling systems.

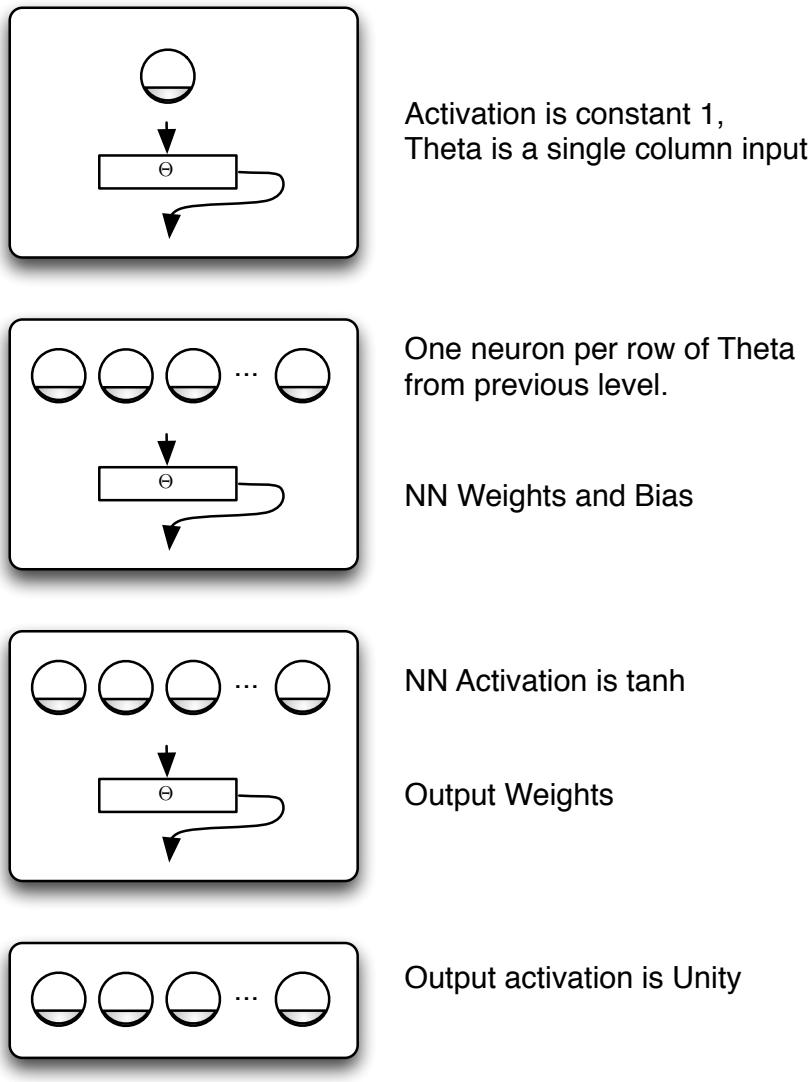



Figure 5.5: Neural Network Detail (Forward)

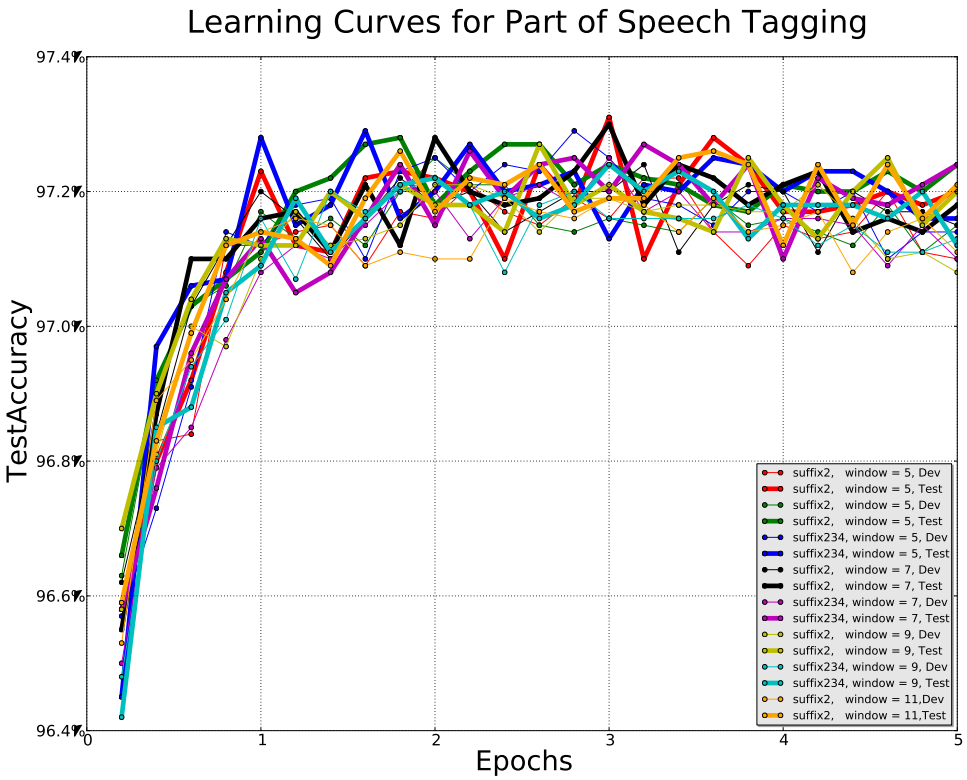



Figure 5.8: ExamplePOS Learning Curves

# Chapter 6

# Semantic Role Labeler (CoNLL 2005)

## 6.1 Semantic Roles

A primary goal of Natural Language Processing is to enable computers to make sense of human language. Semantic Role Labels are tags which are meant to categorize portions of a sentence which are semantic arguments for a given predicate and which determine the appropriate role for each identified portion. Typical semantic arguments include Agent, Patient and Instrument. Recognizing the semantic arguments of sentences is an important step towards giving a computer the unambiguous, structured input that it needs.

A Semantic Role Labeler (or SRL Tagger) is a program that tries to define the portions of a sentence which should be tagged, given the verb that is part of one of the predicates in the sentence. Sentences, parse tree information, a list of verbs, and the expected semantic role labels are used to train and test this program. The CoNLL 2005 shared task Carreras and Màrquez [5] provides standard data sets and evaluation methods for consistent comparison of Semantic Role Labeling systems. This benchmark was used to evaluate the performance of the SRL tagger described in in [8], and was replicated to test the performance of the Daisy SRL Tagger.

## 6.2 SRL Training Dataset

The English Proposition Bank described by Palmer et al. [16] is an important resource of sentences from the Penn Treebank WSJ database annotated with semantic role labels, and was used as the basis for defining the CoNLL 2005 shared task Carreras and Màrquez [5] benchmark.

They provide standard data sets and evaluation methods for consistent comparison of Semantic Role Labeling systems, shown in table 6.1.

The CoNLL distribution data does not include words for the training and development data sets, so those need to come from the Penn Treebank. A short summary of what is available in each database is shown in table 6.2.

| | |
|---|---|
| props | Target verbs and correct propositional arguments. |
| synt.upc | PoS tags, and partial parses by the UPC processors |
| synt.col2 | PoS tags, and full parses of Collins', with WSJ-style Non-Terminals |
| synt.col2h | PoS tags, and full parses of Collins', with Collins-style Non-Terminals |
| synt.cha | PoS tags and full parses of Charniak |
| ne.tar | Named Entities of (Chieu and Ng 03) |

Table 6.1: Dataset files for Semantic Role Labelling

| Item | PennTB2 | ConLL |
|---|---|---|
| words | not Brown | Test only |
| POS | not Brown | Multiple extractions |
| Charniak | yes | no |
| ConLL Charniak | no | yes |
| Collins WSJ-style | no | yes |

Table 6.2: Item Distribution between CoNLL and Penn Treebank 2

## 6.3 Database Preparation

Parse tree information is important for semantic role labeling (Gildea and Palmer [10]), and is provided in many forms for CoNLL 2005. An example sentence from the text database is shown in figure 6.1, which contains rich relationships of entities that form a tree, as diagrammed in figure 6.2.

[8] describes how parse trees can be decomposed into levels which can be fed into the tagging system on a word by word basis, which is consistent with the other features used in the system. These levels can be extracted from words by training a classifier, but the benchmark data input includes parse tree information directly. Any number of levels can be extracted from any of the given parse trees, for example, table 6.3 shows ten levels extracted from the Charniak parse tree.

For training and development, words from PennTB2 were matched with POS from the ConLL Charniak parse file. For test, ConLL given words were matched with POS from the ConLL Charniak

parse file.

The bottom five levels of the Charniak Tree were added to the database files, referred to as PT0-PT4, along with propositions (verbs) to be tagged. The list of actors for each verb were added and were used as the reference tags during training and testing.

Each line the training, development, and test database file consists of: **Word POS PT0..4 Verb (A0..An)**

as shown in figure 6.4, with blank lines to separate sentences.

```
(S1(S(NP( DT The )( NNS results ))(VP( AUX were )(VP( VBN announced )
(SBAR( IN after )(S(NP( DT the )( NN stock )( NN market ))
(VP( VBD closed ))))))( . . )))
```

Figure 6.1: Raw Development Sentence 1225, CoNLL Charniak Tree

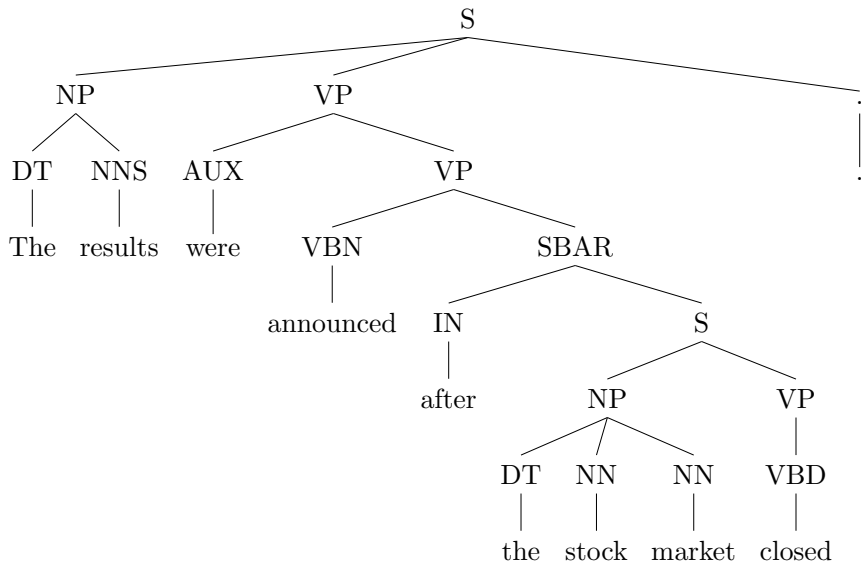



Figure 6.2: Tree for Charniak Parse Tree

| Word | POS | PT0 | PT1 | PT2 | PT3 | PT4 | PT5 | PT6 | PT7 | PT8 | PT9 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| The | DT | B-NP | O | O | O | O | O | B-S1 | B-S1 | B-S1 | B-S1 |
| results | NNS | E-NP | O | O | O | O | O | I-S1 | I-S1 | I-S1 | I-S1 |
| were | AUX | S-VP | S-VP | S-VP | S-VP | B-VP | O | I-S1 | I-S1 | I-S1 | I-S1 |
| announced | VBN | S-VP | S-VP | S-VP | B-VP | I-VP | O | I-S1 | I-S1 | I-S1 | I-S1 |
| after | IN | S-SBAR | S-SBAR | B-SBAR | I-VP | I-VP | O | I-S1 | I-S1 | I-S1 | I-S1 |
| the | DT | B-NP | B-S | I-SBAR | I-VP | I-VP | O | I-S1 | I-S1 | I-S1 | I-S1 |
| stock | NN | I-NP | I-S | I-SBAR | I-VP | I-VP | O | I-S1 | I-S1 | I-S1 | I-S1 |
| market | NN | E-NP | I-S | I-SBAR | I-VP | I-VP | O | I-S1 | I-S1 | I-S1 | I-S1 |
| closed | VBD | S-VP | E-S | E-SBAR | E-VP | E-VP | O | I-S1 | I-S1 | I-S1 | I-S1 |
| . | . | O | O | O | O | O | O | E-S1 | E-S1 | E-S1 | E-S1 |

Table 6.3: SRL Flattened Charniak Parse Tree

| Word | POS | PT0 | PT1 | PT2 | PT3 | PT4 | Verb | A[announce] | A[close] |
|---|---|---|---|---|---|---|---|---|---|
| The | DT | B-NP | O | O | O | O | - | B-A1 | O |
| results | NNS | E-NP | O | O | O | O | - | E-A1 | O |
| were | AUX | S-VP | S-VP | S-VP | S-VP | B-VP | - | O | O |
| announced | VBN | S-VP | S-VP | S-VP | B-VP | I-VP | announce | S-V | O |
| after | IN | S-SBAR | S-SBAR | B-SBAR | I-VP | I-VP | - | B-AM-TMP | O |
| the | DT | B-NP | B-S | I-SBAR | I-VP | I-VP | - | I-AM-TMP | B-A1 |
| stock | NN | I-NP | I-S | I-SBAR | I-VP | I-VP | - | I-AM-TMP | I-A1 |
| market | NN | E-NP | I-S | I-SBAR | I-VP | I-VP | - | I-AM-TMP | E-A1 |
| closed | VBD | S-VP | E-S | E-SBAR | E-VP | E-VP | close | E-AM-TMP | S-V |
| . | . | O | O | O | O | O | - | O | O |

Table 6.4: SRL Test Sentence Example

## 6.4 SRL Forward System diagrams and description

The architecture for the full SRL Forward System, with default options, will now be described. There are four major sections:

- Word Derived Feature Convolution (Figure 6.4).
- Verb Position Feature Convolution (Figure 6.5).
- Word Position Feature Convolution (Figure 6.6).
- Neural Network and Viterbi (Figure 6.7).

Pseudocode for the SRL Tagger is shown in figure 6.3.

```
getSRLTags(sentence, pt0, verbLocs, System)
 1  // WDFMap is the Word Derived Feature Layer Map
 2  // VPMap is the Verb Position Layer Map
 3  // WPMap is the Word Position Layer Map
 4  // NNMap is the Neural Network Layer Map
 5  sentenceFeatureVector = GetSentenceFeatureVector(sentence, pt0)
 6  ConvolveMap(System.WDFMap, sentenceFeatureVector)
 7  for each verbIndex in verbLocs
 8        verbPosFeatureVector = GetVerbPosFeatureVector(sentence, verbIndex)
 9        ConvolveMap(System.VPMap, verbPosFeatureVector)
10        for each wordIndex in (1..sentence.length())
11              wordPosFeatureVector = GetWordPosFeatureVector(sentence, wordIndex)
12              ConvolveMap(System.WPMap, wordPosFeatureVector)
13              sumConv = computeSumOfFeatureMapOutputs(System)
14              maxIX = computeMaxIndices(sumConv)
15              nnInput = sumConv[maxIX] // maxIX is an array, multiple elements are extracted
16              nnOutput = singleCycleRunForward(System.NNMap, nnInput)
17              viterbiInMatrix[wordIndex] = nnOutput // tags are rows, words are columns...
18        SRL[verbIndex] = RunViterbi(viterbiInMatrix)
19  return SRL // Returns a 2D Array, one list of tags per verb
```

Figure 6.3: SRL Tagger Forward Pseudocode

The input to the SRL tagger is a sentence, which is a list of words $w_i$ from $w_1$ to $w_n$, a list of verb positions, and parse tree information for the sentence. The output is the list of predicted SRL IOBES tags for each word, $SRL_i$.

### 6.4.1 Word Derived Feature Convolution Section

The upper portion of figure 6.4 shows the process of extracting features from the words and parse tree information, described in pseudocode lines 5 and 6 of figure 6.3. The numeric information from the features for each word is concatenated together to form one long Feature Vector, shown in the diagram as a multicolored set of rectangles.

Each feature lookup table also contains an entry for **PADDING**. In order to allow the window to extend beyond boundaries of the sentence for early and late words the Feature Vector is padded with the **PADDING** value from each lookup table. The **PADDING** values are also trained.

The Word Derived Feature Vector, shown as a multicolor interleaved block in figure 6.4, is windowed by three words worth of feature information, which is multiplied by the the weights and bias of $\Theta_4$ and stored in the Convolved Word Derived Feature Vector for each word in the sentence. For the default convolution width of 300, this results in a long vector of $300 \cdot n$, where $n$ is the number of words in the sentence.

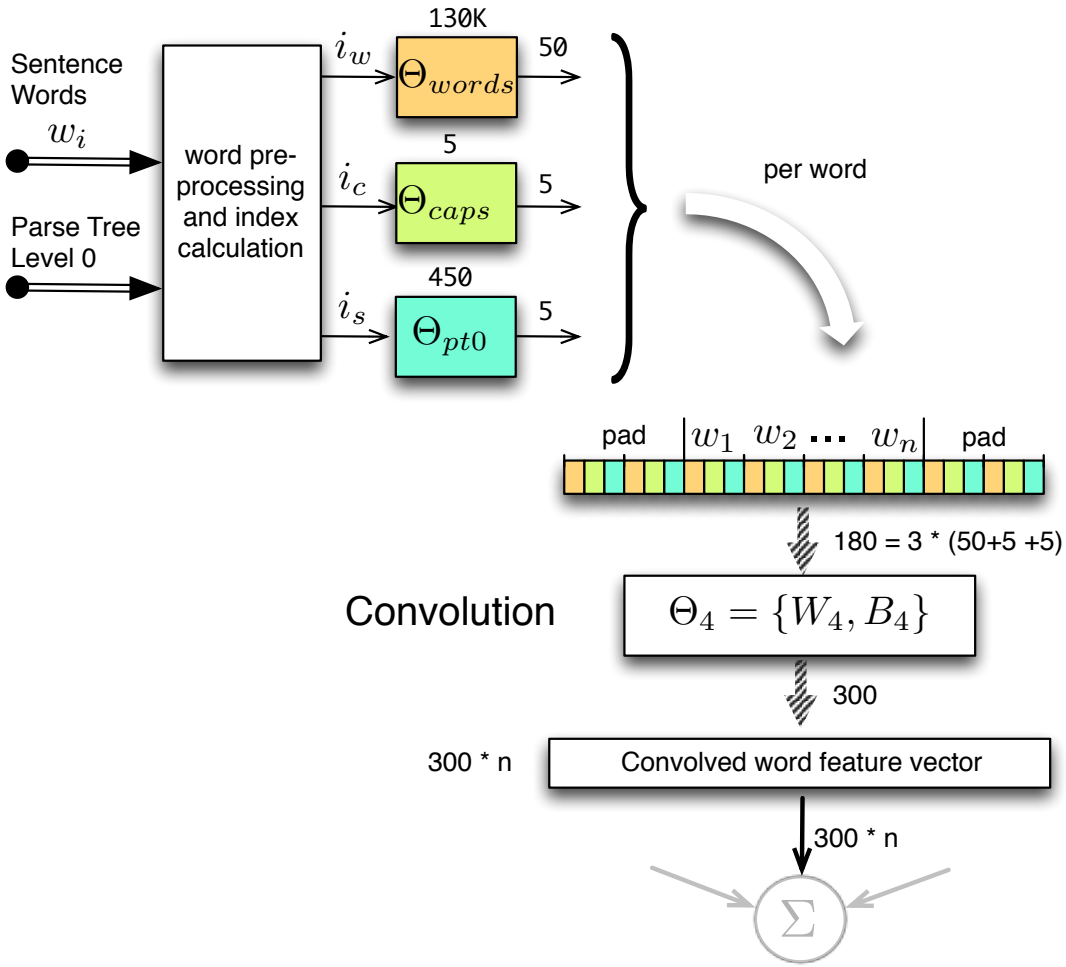



Figure 6.4: SRL Word Derived Feature Convolution

Word derived features are extracted from the sentence and the parse tree information, and the features are interleaved into the Word Derived Feature Vector. This vector is convolved in the "WDFMap" Layer structure.

The three types of features tested for the SRL tagger were:

- Word Representations
- Capitalization
- PT0

#### 6.4.1.1 Word Representations

The input data provided by CoNLL has already gone through some initial tokenizing. This prevents tokenization differences of different systems from influencing the results, which are meant to allow comparison of the POS tagging architecture itself. The Daisy pre-processor does not split hyphenated input words, so each input word will result in a single pre-processed word. Numeric values are collapsed to the single common **0** token, and words are lower-cased to create a word representation lookup word. A vector of 50 floating point values is produced by this lookup using the default architecture. If the word is in the word representation table, the associated vector is output, otherwise the vector corresponding to the special token **UNKNOWN** is output.

#### 6.4.1.2 Capitalization

The caps feature is used to preserve information about each word implied by its upper case letters. Prior to lower casing, each word is checked for all capitals, initial capital, any capital, or no capitals, and this criteria is used to lookup a vector (default length 5) from the caps table.

#### 6.4.1.3 PT0

The bottom level of the flattened Charniak parse tree, discussed earlier.

### 6.4.2 Verb Position Feature Convolution Section

The upper portion of figure 6.5 shows the process of extracting the verb position feature, described in pseudocode lines 8 and 9 of figure 6.3. This is done once for each verb in the sentence. The feature is limited to describing distances of $\pm 12$, and distances out this range are saturated. Using default system parameters, the result of convolving the Feature Vector with $\Theta_3$ results in a sentence length dependent vector size of $300 \cdot n$.

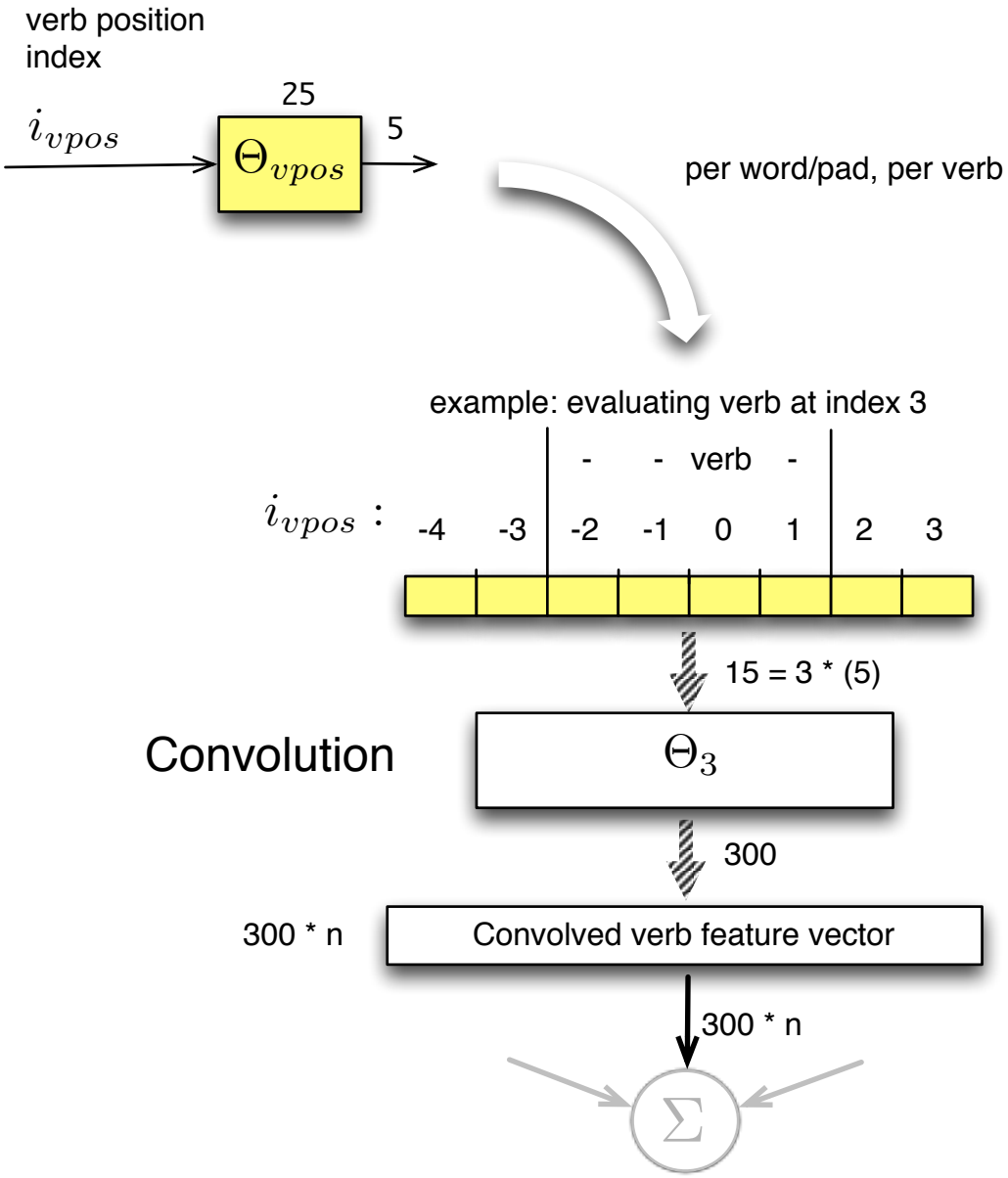



Figure 6.5: SRL Verb Position Feature Convolution

Verb Position Features are based on the position of the verb in the sentence, in this example, the verb is at position 3 of a four word sentence. The Verb Position Feature Vector (yellow) is convolved in the "VPMap" Layer structure.

### 6.4.3 Word Position Feature Convolution Section

The upper portion of figure 6.6 shows the process of extracting the word position feature, described in pseudocode lines 11 and 12 of figure 6.3. This is done once for each word in the sentence, while analyzing each particular verb. The feature is limited to describing distances of $\pm 12$, and distances out this range are saturated. Using default system parameters, the result of convolving the Feature Vector with $\Theta_2$ results in a sentence length dependent vector size of $300 \cdot n$.

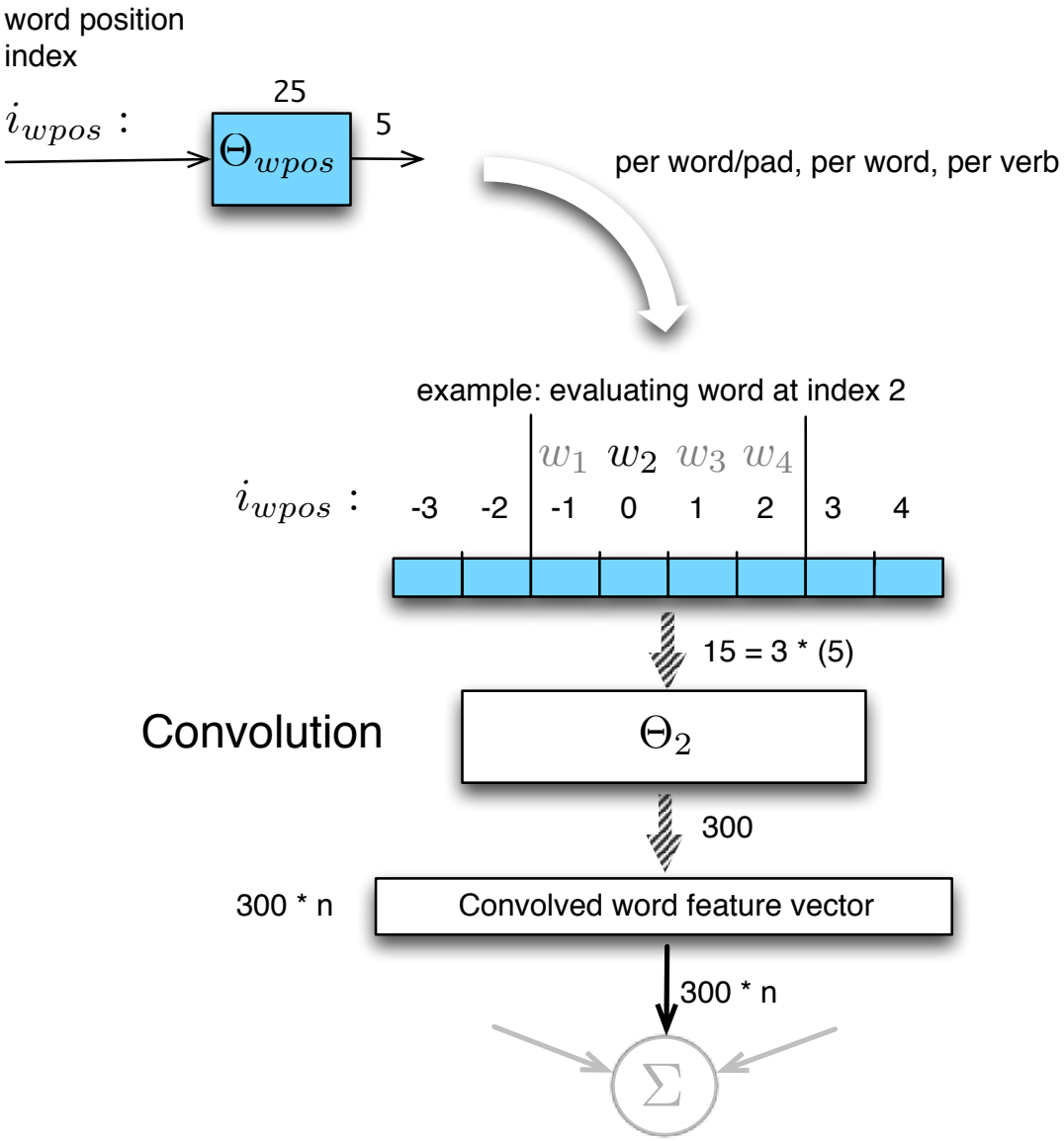



Figure 6.6: SRL Word Position Feature Convolution

Word Position Features are based on the position of the currently evaluated word in the sentence, in this example, the word is the second word in a four word sentence. The Word Position Feature Vector (blue) is convolved in the "WPMap" Layer structure.

### 6.4.4 Neural Network and Viterbi

Figure 6.7 shows the process of combining the Convolved Feature Vectors, processing with a neural network, and finding the most likely sequence with a Viterbi detector.

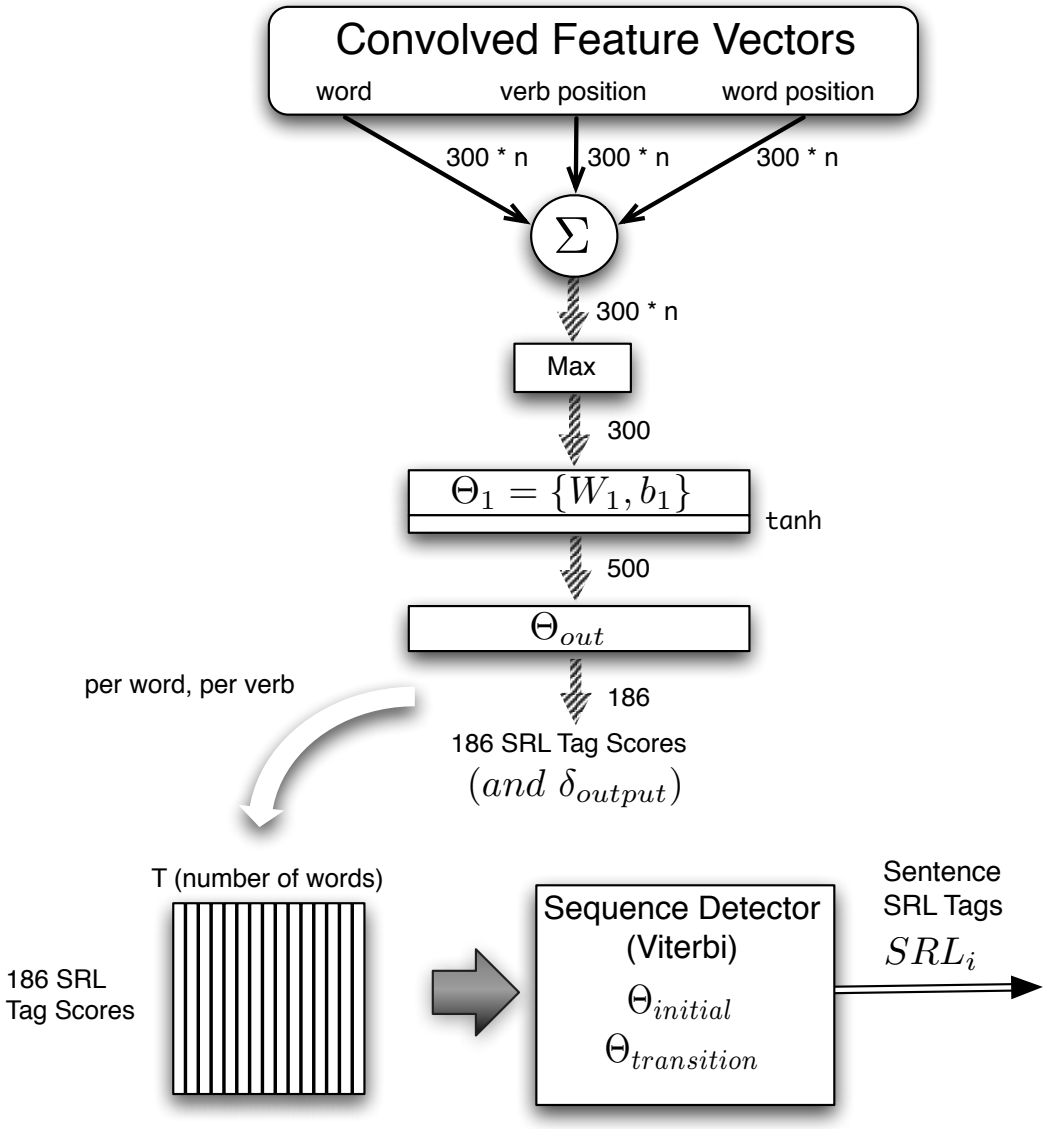



Figure 6.7: SRL Neural Network and Viterbi

The three Convolved Feature Vectors which are shown as the input to this section are diagrammed separately.

The three Convolved Feature Vectors (diagrammed separately) are first added together, then the maximum for each index within each group of 300 is determined. This value is extracted and the result is a 300 element vector which will be the input to the Neural Network. This level of the network is referred to as the "NNMap" Layer structure, and it operates based on the principles described for Figure 5.5.

The output layer provides a score for each possible tag. After running all words through the system for a single verb, a matrix of $tags \times words$ is created, which will be used as the input to the Viterbi sequence detector.

### 6.4.5 Sequence Detection (Viterbi)

The Viterbi algorithm described in pseudocode line 18 of figure 6.3 has the same architecture as the POS Viterbi, its input is a matrix which consists of a vector of SRL scores for each word. Because the tag set for SRL has 186 tags vs. 43 for POS, the Viterbi algorithm runtime is more than 4 times longer for SRL.

## 6.5 Training and Forward Model Creation

Figure 6.8 shows the six stages involved in training the $\Theta$ weights of the system. Figure 6.9 is a pseudocode description of the training process.

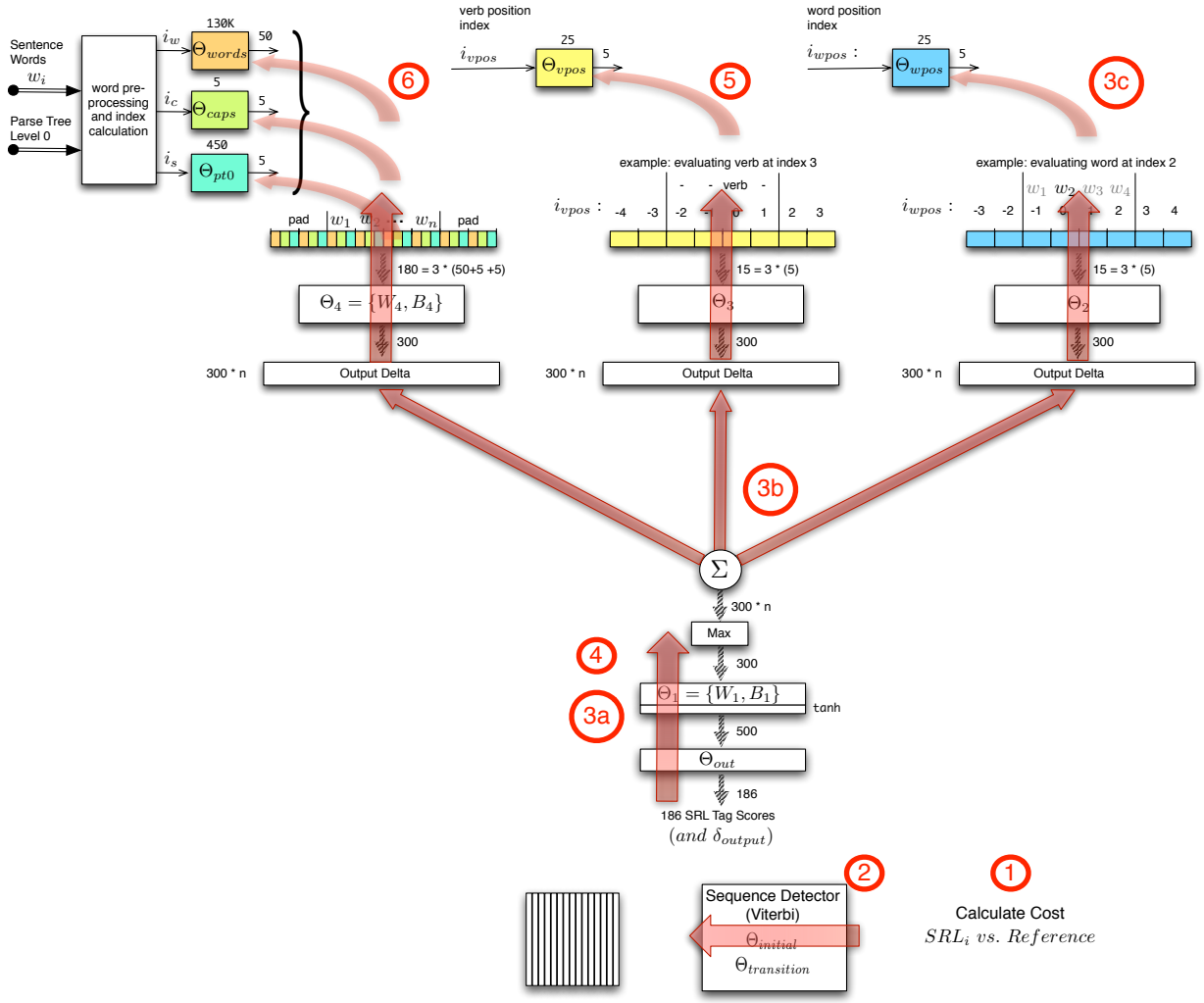



Figure 6.8: SRL Training
After a forward pass to compute the predicted tags, Training the SRL Tagger is done in six stages:
1: Calculate the SLL cost using equation 3.6.
2: Backpropagate to calculate gradients and update Viterbi weights.
3: for each word
a) Backpropagate to calculate gradients and accumulate a sum of updates for the Output Layer, Neural Network, and Feature Vector.
b) Update all Feature Layer Deltas.
c) calculate and update all weights for the word position feature layers.
4: After all words are processed, update Output and Neural Network weights using the update sum from step 3.
5: After all words for a verb are processed, calculate gradients and update all weights for the verb position feature layers.
6: After all verbs are processed, calculate gradients and update all weights for the word-derived feature layers.

A forward pass of the SRL algorithm is inherent in the training procedure, but during the forward pass, intermediate values are saved (lines 13 and 14 of figure 6.9). The activation and maxIndices could change when the word position feature parameters are changed, so they are saved and reused during gradient calculation.

The six stages involved in training the SRL model are now described.

### 6.5.1 Step 1: Cost Calculation

The Viterbi parameters for initial score and transition parameters use the Sentence Level Log-Likelihood Cost, which is calculated using equation 3.6 (line 15 of figure 6.9). This calculation requires computation of intermediate terms referred to as the $\delta$s, which can be re-used for the gradient calculation in step 2. The iterative algorithm used to calculate the SLL cost is based on the forward algorithm and is described in section 3.6.

### 6.5.2 Step 2: Viterbi Backpropagation

The Viterbi gradients are calculated using equation 3.17 (line 17 of figure 6.9). This calculation reuses the saved $\delta$s from the SLL Cost calculation. The iterative algorithm used to calculate the gradients is described in section 3.7.

### 6.5.3 Step 3: Neural Network Gradients and Word Position Updates

This step has three parts, 3a, 3b, and 3c. It is performed per word, per verb, inside of two loops of the training algorithm.

#### 6.5.3.1 Step 3a: Neural Network Backpropagation

The Neural Network gradients are calculated using the classic back propagation algorithm. Gradients are calculated based on maximizing the word level log-likelihood cost function (equation 2.4), or sentence level log-likelihood (equation 3.6), based on a training parameter. Equation 2.7 is then applied iteratively, from the end of the network back to the Feature Vector, to compute

the partial derivatives of the cost function with respect to the inputs. Based on the calculated gradients, the actual parameter gradients can be calculated using equations 2.8 and 2.9 (line 18 of Figure 6.9). The gradients are saved to a stack, and Neural Network weights are not modified during this step since that would alter the behavior of the forward network.

#### 6.5.3.2 Step 3b: Propagation through the Max

Based on the saved maxIX, the relevant delta terms in the output layers of word derived, verb pos, and word pos are added in (line 19 of Figure 6.9).

#### 6.5.3.3 Step 3c: Word Position Layer Updates

Based on the saved activation states, a full back propagation of the Word Position Layers only is done. Equation 2.7 is then applied iteratively, from the end of the network back to the Feature Vector, to compute the partial derivatives of the cost function with respect to the inputs. Based on the calculated gradients, the actual parameter gradients can be calculated using equations 2.8 and 2.9. The $\Theta_2$ parameters and word position lookup table weights $\Theta_{wpos}$ are updated (line 19 of Figure 6.9).

### 6.5.4 Step 4: Neural Network Weight Updates

After all words are processed for a verb by repeatedly running step3, step 4 of Figure 6.8 and line 20 of Figure 6.9 uses the sum of the gradients calculated during step 3 to update $\Theta_{out}$ and $\Theta_1$. Equations 3.1 or 3.3 are used based on whether adaGrad is specified in the training parameters.

### 6.5.5 Step 5: Verb Position Layer Updates

After all words for a verb have been processed, a full back propagation of the Verb Position Layers is done by deconvolving through the $\Theta_3$ parameters. Equation 2.7 is then applied iteratively, from the end of the network back to the Feature Vector, to compute the partial derivatives of the cost function with respect to the inputs. Based on the calculated gradients, the actual parameter

gradients can be calculated using equations 2.8 and 2.9. The $\Theta_3$ parameters and word position lookup table weights $\Theta_{vpos}$ are then updated (line 20 of Figure 6.9).

### 6.5.6 Step 6: Word Derived Feature Layer Updates

After all verbs have been processed, a full back propagation of the Word Derived Feature Layers is done by deconvolving through the $\Theta_4$ parameters. Equation 2.7 is then applied iteratively, from the end of the network back to the Feature Vector, to compute the partial derivatives of the cost function with respect to the inputs. Based on the calculated gradients, the actual parameter gradients can be calculated using equations 2.8 and 2.9. The $\Theta_4$ parameters and word derived feature lookup table weights $\Theta_{words}$, $\Theta_{caps}$, and $\Theta_{pt0}$ are then updated (line 21 of Figure 6.9).

```
trainSRLTagger(sentence, pt0, verbLocs, referenceTags, System)
    // System.WDFMap is the Word Derived Feature Layer Map
    // System.VPMap is the Verb Position Layer Map
    // System.WPMap is the Word Position Layer Map
    // System.NNMap is the Neural Network Layer Map
 1  sentenceFeatureVector = GetSentenceFeatureVector(sentence, pt0)
 2  ConvolveMap(System.WDFMap, sentenceFeatureVector)
 3  for each verbIndex in verbLocs
 4      verbPosFeatureVector = GetVerbPosFeatureVector(sentence, verbIndex)
 5      ConvolveMap(System.VPMap, verbPosFeatureVector)
 6      for each wordIndex in (1..sentence.length())
 7          wordPosFeatureVector = GetWordPosFeatureVector(sentence, wordIndex)
 8          ConvolveMap(System.WPMap, wordPosFeatureVector)
 9          sumConv = computeSumOfFeatureMapOutputs(System)
10          maxIX = computeMaxIndices(sumConv)
11          nnInput = sumConv[maxIX] // maxIX is an array, multiple elements are extracted
12          nnOutput = singleCycleRunForward(System.NNMap, nnInput)
13          savedState.activationList[wordIndex] = activation
14          savedState.maxIndicesList[wordIndex] = maxIndices
15          viterbiInMatrix[wordIndex] = nnOutput // tags are rows, words are columns...
16      tagsForVerb = RunViterbi(viterbiInMatrix)
17      ViterbiGradient = trainViterbiForVerb(tagsForVerb, referenceTags[verbIndex]);
18      NNGradients = computeGradients(tagsForVerb, referenceTags[verbIndex],
                ViterbiGradient, savedState.activationList, savedState.maxIndicesList)
19      updateGradientsAndWordPositionWeights(System, NNGradients)
20      deconvolveMap(srlVPosMap, parms.windowSize, srlVerbPositionFeatureFactory, f);
21  deconvolveMap(srlWDFMap, parms.windowSize, wordDerivedFeatureFactory, f);
```

Figure 6.9: SRL Tagger Training Pseudocode

## 6.6 Results

A comparison of the F1 results of various systems are shown in figure 6.5. The CoNLL 2005 test script was used to double check results with Daisy, which were not as good as the best CoNLL 2005 results but better than the reported [8] results. The best CoNLL results were achieved by

using inference on the output of six systems, each trained with a different parse tree. The published difference between the best single system and the best inferred results (on the development data only) was 2.5%.

| System Description | Test F1 |
|---|---|
| Benchmark CoNLL 2005 | 77.92% |
| Benchmark CoNLL 2005 best single system (est.) | 75.42% |
| Scratch Best | 75.49% |
| Daisy Best | 76.30% |

Table 6.5: SRL Test F1

The best AdaGrad learning rates are shown in Table 6.6.

| Trained Section | AdaGrad Learning Rate |
|---|---|
| $\Theta_{words}$ | 0.420 |
| $\Theta_{caps}$ | 0.600 |
| $\Theta_{pt0}$ | 0.600 |
| $\Theta_4$ | 0.025 |
| $\Theta_{vpos}$ | 0.600 |
| $\Theta_3$ | 0.025 |
| $\Theta_{wpos}$ | 0.600 |
| $\Theta_2$ | 0.025 |
| $\Theta_1$ | 0.025 |
| $\Theta_{out}$ | 0.025 |
| $\Theta_{Vinit}$ | 0.05 |
| $\Theta_{Vtrans}$ | 0.05 |

Table 6.6: SRL Tagger AdaGrad Learning Rates

Word feature learning rates can make a big difference on the performance of the system, and they were varied in many experiments to arrive at the best set of learning rates shown in table 6.6. table 6.7 shows the variation in performance when the word features learning rates were scaled.

| | 0.25 | 0.5 | 1.0 | 2.0 | 4.0 |
|---|---|---|---|---|---|
| Daisy SLL | 76.19% | 76.11% | 76.30% | 75.71% | 75.34% |

Table 6.7: Daisy Test F1 for various word feature learning rates

Table 6.8 compares the performance of SENNA [6] and Daisy systems when using randomized word features vs. the pre-trained word features described in Chapter 4. The SENNA [6] performance is based on an earlier system, without the PT0 parse information feature, and shows a larger increase when pre trained words are used, possibly because some of the information from the parse tree is made available by the pre trained words. The Daisy performance for the optimum learning rate (scaled by 1.0) is about 0.8% better with SLL.

| System Description | ParseInfo | Random | Pre-trained | Difference |
|---|---|---|---|---|
| SENNA [6] SLL | none | 70.90% | 74.15% | +3.25% |
| Daisy SRL SLL 1.0x | none | 74.84% | 75.55% | +0.71% |
| Daisy SRL SLL 1.0x | pt0 | 75.11% | 76.30% | +1.19% |

Table 6.8: SRL Test F1 Improvement with Pre-Trained Words

On Daisy systems with pre trained word features, Word Level Likelihood for SRL Neural Network training performs about 1.68% better for a tuned system than Sentence Level Likelihood. (Table 6.9) The WLL trained systems seemed to prefer a lower word feature learning rate, but this was not explored in depth.

| System Description | WLL | SLL | Difference |
|---|---|---|---|
| Daisy 0.25 word LR | 75.17% | 76.19% | +1.02% |
| Daisy 0.5 word LR | 75.08% | 76.11% | +1.03% |
| Daisy 1.0 word LR | 74.62% | 76.30% | +1.68% |
| Daisy 2.0 word LR | 74.03% | 75.71% | +1.68% |
| Daisy 4.0 word LR | 73.10% | 75.34% | +2.24% |

Table 6.9: Daisy SRL Test F1 WLL-driven vs. SLL-driven Training (using pre trained word features, various learning rates.)

## 6.7 Extensions of the Semantic Role Labeling Architecture

Extensions of this work could include adding more parse information as input features, for example more levels of the Charniak tree and other parse tree information. The effect of increasing the number of layers, size of layers, and window sizes would also be interesting. A big reason for the resurgence of using these techniques is that compute power continues to become cheaper and faster. Even so, the Daisy Semantic Role Labeling systems take 3-4 days to fully train. In order to explore more architectural options, the Java architecture could probably be sped up by a factor of ten if the compute architecture is based around tight C, using more multithreading, mini batch instead of pure stochastic gradient descent, CUDA for GPU acceleration, and a more careful application of linear algebra libraries.

# Chapter 7

# Semantic Role Labeler (CoNLL 2009)

## 7.1 SRL (from Dependency Parser) Training Dataset

The CoNLL 2009 shared task objective is to perform and evaluate Semantic Role Labeling (SRL) using a dependency-based representation for semantic dependencies (Hajič et al. [11]).

The dataset contains labels for propositions centered around verbal predicates as in the 2005 dataset, but also includes propositions centered around nouns and other major part-of-speech categories. It includes data for seven different languages, but here only English results are presented. Also, the syntactic dependencies to be modeled are more complex than the ones used in the previous CoNLL 2005. Table 7.1 shows an example sentence and its representation in the dataset. The PDEPREL feature is the predicted dependency relation tag, output from a dependency parser, and the PHEAD feature is the predicted head word from a parser, both of which are used as features for training the neural network system.

| ID | FORM | LEMMA | PLEMMA | POS | PPOS | FEAT | PFEAT | HEAD | PHEAD | DEPREL | PDEPREL | FILLPRED | PRED | A[announce] | A[close] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | The | the | the | DT | DT | _ | _ | 2 | 2 | NMOD | NMOD | _ | _ | _ | _ |
| 2 | results | result | result | NNS | NNS | _ | _ | 3 | 3 | SBJ | SBJ | _ | _ | A1 | _ |
| 3 | were | be | be | VBD | VBD | _ | _ | 0 | 0 | ROOT | ROOT | _ | _ | _ | _ |
| 4 | announced | announce | announce | VBN | VBN | _ | _ | 3 | 3 | VC | VC | Y | announce.01 | _ | _ |
| 5 | after | after | after | IN | IN | _ | _ | 4 | 4 | TMP | TMP | _ | _ | AM-TMP | _ |
| 6 | the | the | the | DT | DT | _ | _ | 8 | 8 | NMOD | NMOD | _ | _ | _ | _ |
| 7 | stock | stock | stock | NN | NN | _ | _ | 8 | 8 | NMOD | NMOD | _ | _ | _ | _ |
| 8 | market | market | market | NN | NN | _ | _ | 9 | 9 | SBJ | SBJ | _ | _ | _ | A1 |
| 9 | closed | close | close | VBD | VBD | _ | _ | 5 | 5 | SUB | SUB | Y | close.02 | _ | _ |
| 10 | . | . | . | . | . | _ | _ | 3 | 3 | P | P | _ | _ | _ | _ |

Table 7.1: SRL Dependency Parse Input Test Sentence Example

## 7.2 SRL Dependency Parse Input Forward System

The architecture for the SRL System described in chapter 6 is essentially the same for the dependency parse input, except for the Word Derived Feature Convolution, shown in figure 7.1.

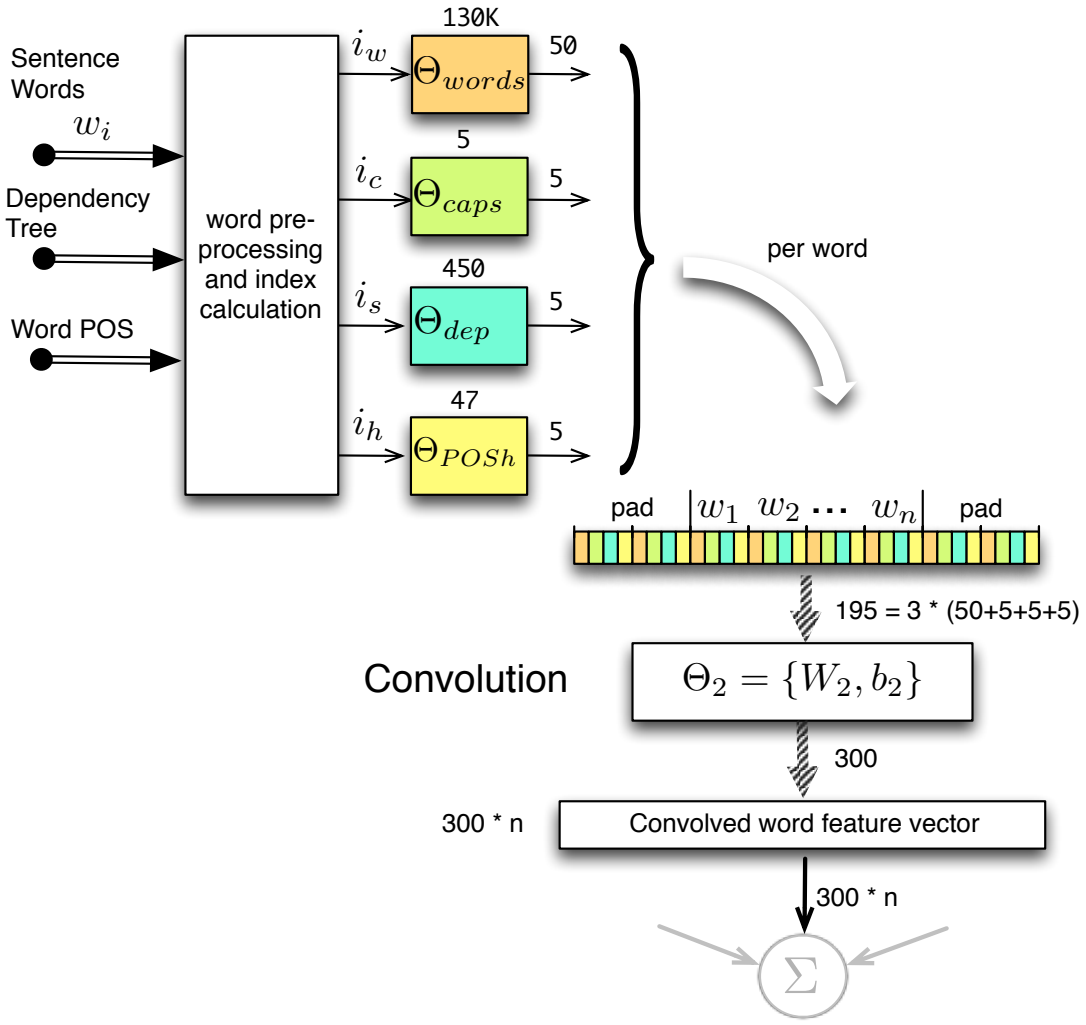



Figure 7.1: SRL with Dependency Parser Input Front-end Flow.

The four types of features tested for the SRL Dependency Parse tagger were:

- Word Representations
- Capitalization

- Dependency Relation
- POS tag of head

### 7.2.1 Word Representations

See section 6.4.1.1.

### 7.2.2 Capitalization

See section 6.4.1.2.

### 7.2.3 Dependency Relation

The PDEPREL column of table 7.1.

### 7.2.4 POS tag of head

The PPOS column of the word indexed by PHEAD (table 7.1).

## 7.3 Results

A comparison of the F1 results of various systems are shown in figure 7.2. The CoNLL 2009 test script was used to double check results with Daisy. According to Björkelund et al. [3], 20 features were used for argument identification, including the Dependency Relation Path, and Part of Speech of Dependency Relation Path. A reranker was run on the output of multiple system outputs, and the results prior to the use of the reranker were a little worse then the Daisy results.

| System Description | Test F1 |
|---|---|
| Benchmark CoNLL 2009 (Nugues) | 85.63% |
| Benchmark CoNLL 2009 Without Reranking (Nugues) | 84.44% |
| Daisy Best | 84.63% |

Table 7.2: SRL Dependency Parse Test F1

The learning rate was increased for the word features only, and the results are shown in table 7.3. This was done using Word Level Log Likelihood (WLL) and Sentence Level Log Likelihood (SLL). SLL driven training clearly improves performance for this system.

| System Description | WLL | SLL |
|---|---|---|
| HPOS words 1.0x | 84.27% | 84.63% |
| HPOS words 2.0x | 84.09% | 84.27% |
| HPOS words 4.0x | 84.02% | 84.16% |
| HPOS words 8.0x | 83.71% | 83.95% |

Table 7.3: SRL Dependency Parse Word Gain Sweep Test F1 Results

The advantage of using pre trained word representations is shown in figure 7.4. There is about a 0.43% improvement shown in these experiments from using the pre trained representations.

| System Description | No HPOS | With HPOS |
|---|---|---|
| SLL Pretrained Words | 84.21% | 84.63% |
| SLL Random Words | 83.66% | 84.08% |

Table 7.4: SRL Dependency Parse Random and HPOS Test F1 Results

# Chapter 8

# Conclusion

Word representations trained with unsupervised learning techniques on large corpora were found to contain a remarkable amount of linguistic grouping, which was advantageously applied to practical natural language processing tasks. The use of unsupervised training should result in much less costly development of NLP algorithms since supervised training data is very expensive to create. The neural network based systems can used relatively simple feature extraction, which results in high speed and low memory requirements, and the performance of all three of the systems compared closely to the benchmarks.

# Appendix A

## EC2 Infrastructure

The Amazon EC2 infrastructure used to run the tests is shown in figure A.1.

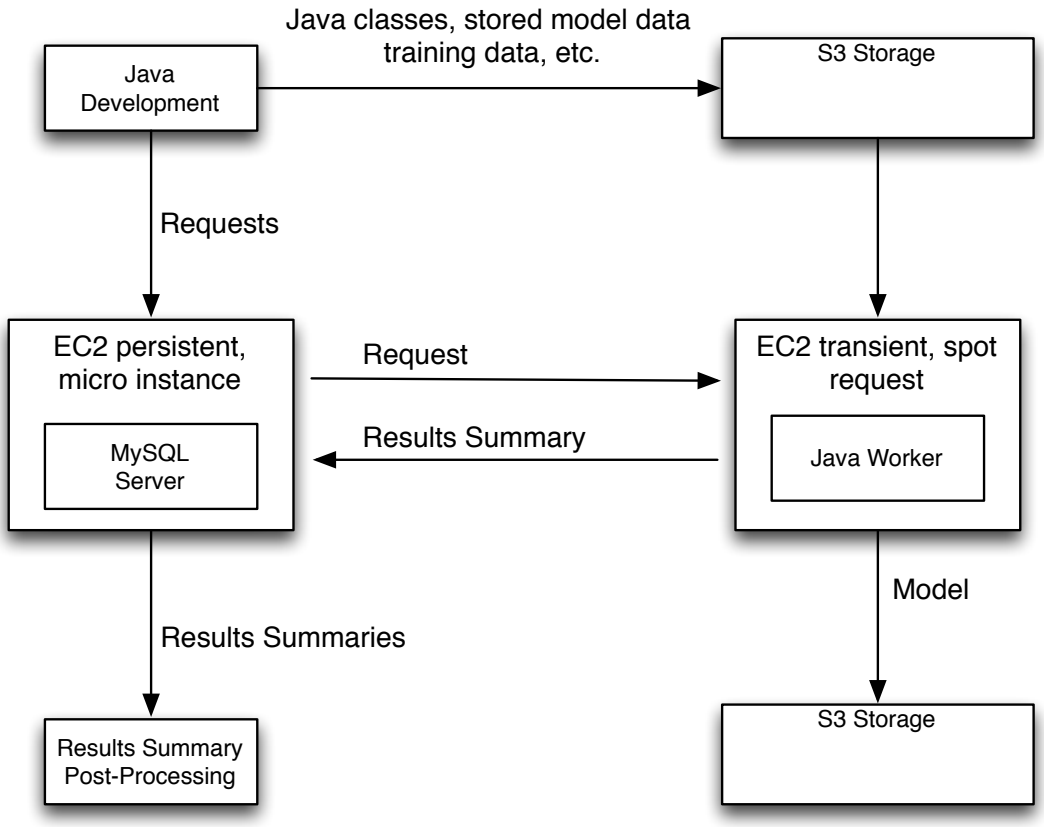



Figure A.1: Amazon S3 and EC2 System Flow

An important design consideration was to encapsulate the description of the system, the training procedure, and the results. This enabled storing and retrieving experiments using a standard database.

## Bibliography


[1] Yoshua Bengio, Holger Schwenk, Jean-Sébastien Senécal, Fréderic Morin, and Jean-Luc Gauvain. Neural probabilistic language models. In Innovations in Machine Learning, pages 137–186. Springer, 2006.

[2] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. Advances in neural information processing systems, 19:153, 2007.

[3] Anders Björkelund, Love Hafdell, and Pierre Nugues. Multilingual semantic role labeling. In Proceedings of the Thirteenth Conference on Computational Natural Language Learning: Shared Task, pages 43–48. Association for Computational Linguistics, 2009.

[4] John S Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In Neurocomputing, pages 227–236. Springer, 1990.

[5] Xavier Carreras and Lluís Màrquez. Introduction to the conll-2005 shared task: Semantic role labeling. In Proceedings of the Ninth Conference on Computational Natural Language Learning, pages 152–164. Association for Computational Linguistics, 2005.

[6] Ronan Collobert. Senna, August 2011. URL `http://ml.nec-labs.com/senna/`.

[7] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In Proceedings of the 25th international conference on Machine learning, pages 160–167. ACM, 2008.

[8] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. The Journal of Machine Learning Research, 12:2493–2537, 2011.

[9] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. The Journal of Machine Learning Research, 12: 2121–2159, 2011.

[10] Daniel Gildea and Martha Palmer. The necessity of parsing for predicate argument recognition. In Proceedings of the 40th Annual Meeting on Association for Computational Linguistics, pages 239–246. Association for Computational Linguistics, 2002.

[11] Jan Hajič, Massimiliano Ciaramita, Richard Johansson, Daisuke Kawahara, Maria Antònia Martí, Lluís Màrquez, Adam Meyers, Joakim Nivre, Sebastian Padó, Jan Štěpánek, et al. The conll-2009 shared task: Syntactic and semantic dependencies in multiple languages. In Proceedings of the Thirteenth Conference on Computational Natural Language Learning: Shared Task, pages 1–18. Association for Computational Linguistics, 2009.

[12] Geoffrey Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. Neural computation, 18(7):1527–1554, 2006.

[13] Ray Kurzweil. How to create a mind: The secret of human thought revealed. Penguin, 2012.

[14] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. COMPUTATIONAL LINGUISTICS, 19(2): 313–330, 1993.

[15] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In Proceedings of NAACL-HLT, pages 746–751, 2013.

[16] Martha Palmer, Daniel Gildea, and Paul Kingsbury. The proposition bank: An annotated corpus of semantic roles. Computational linguistics, 31(1):71–106, 2005.

[17] Lawrence Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. Proceedings of the IEEE, 77(2):257–286, 1989.

[18] Frank Rosenblatt. Principles of neurodynamics. Spartan Book, 1962.

[19] William W Cohen Robert E Schapire and Yoram Singer. Learning to order things. In Advances in Neural Information Processing Systems 10: Proceedings of the 1997 Conference, volume 10, page 451. MIT Press, 1998.

[20] Kristina Toutanova, Dan Klein, Christopher D Manning, and Yoram Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1, pages 173–180. Association for Computational Linguistics, 2003.

[21] Jason Weston, Frédéric Ratle, Hossein Mobahi, and Ronan Collobert. Deep learning via semi-supervised embedding. In Neural Networks: Tricks of the Trade, pages 639–655. Springer, 2012.