Learning hypernymy in distributed word vectors via a stacked LSTM network

Irving Rodriguez irodriguez@stanford.edu

June 4, 2016

Abstract

We aim to learn hypernymy present in distributed word representations using a deep LSTM neural network. We hypothesize that the semantic information of hypernymy is distributed differently across the components of the hyponym and hypernym vectors for varying examples of hypernymy. We use an LSTM cell with a replacement gate to adjust the state of the network as different examples of hypernymy are presented. We find that a seven layer LSTM model with dropout achieves a test accuracy of 81.4% on the Linked Hypernyms Dataset, though further comparison with other models in the literature is necessary to verify the robustness of these results.

1 Introduction

In the last several years, distributed word vectors have shown the ability to learn semantic and syntactic information without receiving explicit inputs outlining these properties. Particularly, both the word2vec[6] and GloVe[13] models have shown that distributed word vectors cluster together in space based on linguistic similarities and that these relationships can be quantitatively captured with simple vector algebra and analogy tasks.

How these properties manifest themselves within the word vectors, however, is not well understood. This is especially true of paradigmatic linguistic relationships like hypernymy and synonymy. There is significant linguistic benefit in understanding these complex properties - a robust hypernymy model would greatly facilitate automatic taxonomy construction, for example, and Wolter and Gylstad have noted that paradigmatic relationships tend to be similar across languages and could be used to improve machine translation models.[15]

Previous models have failed to capture the full linguistic complexity of hypernymy. These models often rely on supervised methods that are not linguistically consistent with the properties of hypernymy, namely asymmetry, transitivity, and hierarchy. As such, we attempt to remedy these shortcomings through a recurrent neural net that maps a hyponym vector to one of its hypernym vectors. Specifically, we set out to learn the mapping from a hyponym to a hypernym vector based strictly on the components of the distributed vectors with a stacked LSTM network.

2 Methods

We set out to build a deep recurrent neural network to learn a function $H(h_{w_o}$ that produces the hypernym vector given the input hyponym vector of word w_o, h_{w_o}.

2.1 Stacked LSTM Model

A long short-term memory (LSTM) models seem well-suited for our task. We hypothesize that hypernymy (and other linguistic relationships which distributed vector models seem to automatically learn) are baked directly into the individual components of the vectors. The LSTM architecture contains a cell state C_t at each given time step t. In our case, this represents our activation matrix which maps a hyponym vector to its hypernym vector. The hidden layer then consists of four calculations dubbed the forget, input, activation, and output gates. At each time step, we input the hyponym vector h_{t_o} into the first cell,

$$i_t = \sigma(W_i h_{(t-1)_e} + W_i h_{t_o} + b_i)$$

$$f_t = \sigma(W_f h_{(t-1)_e} + W_f h_{t_o} + b_f)$$

$$o_t = \sigma(W_o h_{(t-1)_e} + W_o h_{t_o} + b_o)$$

$$c_t = tanh(W_c h_{(t-1)_e} + W_i h_{t_o} + b_c)$$

where i, f, o, c denote the forget, input, and output, and activation gates with their respective bias terms, and $h_{(t-1)_e}$ denotes the predicted hypernym in the previous time step.

In essence, each of these gates calculates which components of the vectors within the cell carry meaningful information and updates the cell state accordingly. The forget gate scales down the components of the input state C_{t-1}, the input gate scales the components of w_t to be added to the cell state, the activation gate determines the new component values to be added to the cell state, and the output gate yields a vector to scale the predicted hypernym.

Our LSTM cell combines the input and forget gates into a replacement gate which only adds the output of the input gate to the cell state in components which were forgotten. We then update the cell state and calculate the predicted hypernym,

$$C_t = f_t \times C_{t-1} + (1 - f_t) \times c_t \tag{1}$$

$$h_{t_e} = o_t \times tanh(C_t) \tag{2}$$

where $\times$ denotes element-wise multiplication.

We hypothesize that the input, forget, and output layers should be able to identify the components of the hyponym vectors which best predict hypernymy and update the weights accordingly. Since hypernymy is hierarchal and a single word can have hypernyms on varying levels of generality, these components could change from example to example but should be handled appropriately at each step t in the replacement gate.

We minimize the quadratic loss function over our training set using mini-batch SGD

$$J = \frac{1}{2}\Sigma_{i=1}^{m}||h_{i_e} - h_{w_e}||_2^2 \tag{3}$$

and optimize in the number of layers in the model and the number of hyponym-hypernym pairs processed at each iteration of mini-batch SGD.

In order to prevent overfitting and improve our model's robustness outside of the validation set, we introduce dropout in between the penultimate and final layers for our best-performing model.

We implement our stacked LSTM model using TensorFlow.

[1]

2.2 Piecewise Projection Model

We compare our performance to the supervised model outlined by Fu et. al.[4] To replicate results, we use K-Means clustering in order to cluster the vector difference $h_{w_o} - h_{w_e}$ into k clusters for each word pair (w_o, w_e) in our dataset. For each cluster, we then learn a linear projection $\Psi_j(h_{w_o})$ that maps a hyponym to its hypernym and classify positive pairs when $|\Psi_j(h_{w_{o_i}}) - h_{w_{e_i}}| < \delta$ for some hypersphere radius δ in our difference vector space. We minimize the quadratic loss over the dataset to learn the projections.

2.3 Distributed Word Vectors

We first train highly-dimensional word vectors using the publicly available word2vec module.[9] Due to time constraints, we use a single optimized CBOW model to obtain our pre-trained vectors for use as inputs into our LSTM network.

Our model uses Mikolov et. al's method for consolidating common phrases into a single token [**?**]. We thus learn an individual vector for these phrases (consisting of up to 3 words, like "new_york_times") and treat them as a single hyponym or hypernym.

2.4 Data

The Linked Hypernyms Dataset (LHD) provided by Kliegr as part of the DB-Pedia project contains 3.7M hyponym-hypernym pairs. [10] These pairs are generated by parsing the first sentences of Wikipedia articles for syntactic patterns (i.e, "X is a Y") for hypernyms matching the article title.

The LHD set contains tokens for phrases longer than 3 words (like "history_of_the_united_states") and also contains many words which are not in the vocabulary of our vector model. As such, we prune these pairs from the set. This leaves roughly 1.5 million hyponym-hypernym pairs. We then sample 20% of the remaining pairs to and evenly split them for use as our validation and test sets.

We also evaluate the performance of our model on the BLESS dataset used in previous hypernymy classifiers. [4] [14] The dataset contains 2.7k example pairs parsed from the WackyPedia corpus using similar syntactic patterns. We attempt to perform the same 10-fold cross-validation in order to compare our accuracy with these classifiers.

The most common tokens for both datasets are shown in Appendix A. Both of these datasets are publicly available. [3] [10]

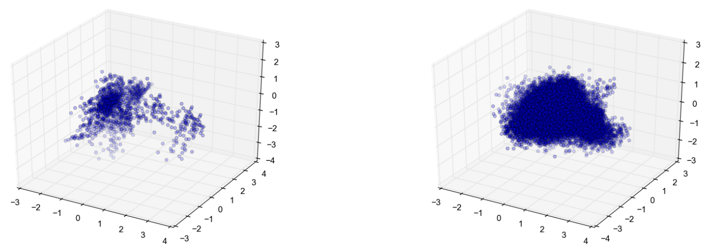

Figure 1: Plots of the 3 principal components for each data set. Left: BLESS. Right: Linked Hypernyms

3 Results

3.1 Word Vector Model

We trained a skip-gram model[5], choosing the set of vector dimensions, window size, and negative sampling size (denoted d, w, s_n, respectively) based on the performance of the given model on the analogy task provided in the Google word2vec implementation [9]. We use the July 2015 English Wikipedia dump as the training set for our word vectors.

The parameters which achieve the highest accuracy are $d^* = 300, c^* = 9, s_n^* = 9$. (Fig. 2)

3.2 Piecewise-Projection Model

We trained our piecewise-projection model using our pre-trained word2vec vectors. We reserve 20% of the data for our validation and test sets and training on the remaining pairs. We choose the optimal k, δ based on the F1

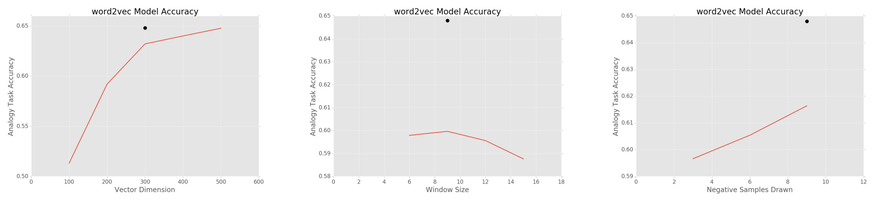

Figure 2: Analogy task accuracy of word vector models. The accuracy of the model with optimal parameters is shown as the black dot. Parameters are fixed at $(d = 200, c = 5, s_n = 3)$.

score of each model with 10-fold cross-validation on the training set and find $(k^*_{BLESS} = 30, \delta^*_{BLESS} = 3.5)$ and $(k^*_{LHD} = 15, \delta^*_{LHD} = 4.0)$ (Fig. 3)

We then use the optimal learned parameters to evaluate the model on the respective test set. We report an F1 score of 87.0% on the BLESS set and 61.3% on the LHD set.

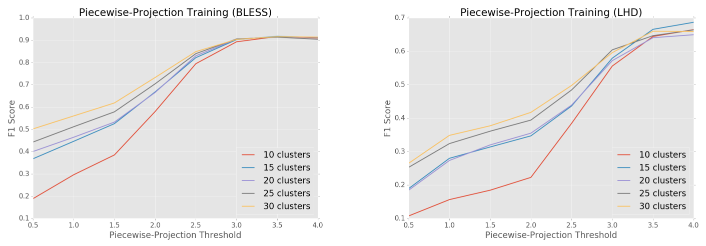

Figure 3: F1 score of training phase for k clusters and a threshold δ. Left: 2.7k pairs, BLESS. Right: 1.5M pairs, LHD.

3.3 Stacked LSTM Model

For the stacked LSTM model, we vary the number of layers from $n_l = 1$ to 8. Additionally, we tune the number of examples fed into the mini-batch SGD at any iteration. Due to time constraints, we only add dropout to the two models with the highest validation accuracy. We then test these models on the BLESS set instead of tuning them once more on the validation set of BLESS.

For a given output vector, we count correct predictions as hypernym tokens that one of the three vectors in our word vector space with the highest cosine similarity to the predicted vector.

We find that that the models with $b^*_s = 50$ and $n_l = 4, 7$ perform best on the LHD set. We report the test accuracies of these models and their dropout counterparts in Figure 4.

Model (LHD)	**Training Accuracy (%)**	**Validation Accuracy (%)**	**Test Accuracy (%)**
PW-PROJ	66.1	69.1	61.3
4-LSTM	76.7	76.8	75.9
7-LSTM	73.4	77.1	75.0
4-LSTM + D	84.0	82.4	79.8
7-LSTM + D	85.2	84.1	**81.3**

Model (BLESS)	**Training Accuracy (%)**	**Validation Accuracy (%)**	**Test Accuracy (%)**
PW-PROJ	91.3	93.5	**87.0**
4-LSTM	98.2	96.2	78.7
7-LSTM	97.8	96.3	71.1
4-LSTM + D	73.6	71.4	72.4
7-LSTM + D	76.7	74.6	76.2

Figure 4: Test accuracies of models with highest validation performance on both the BLESS and LHD sets (F1 score for piecewise-projection classifier.

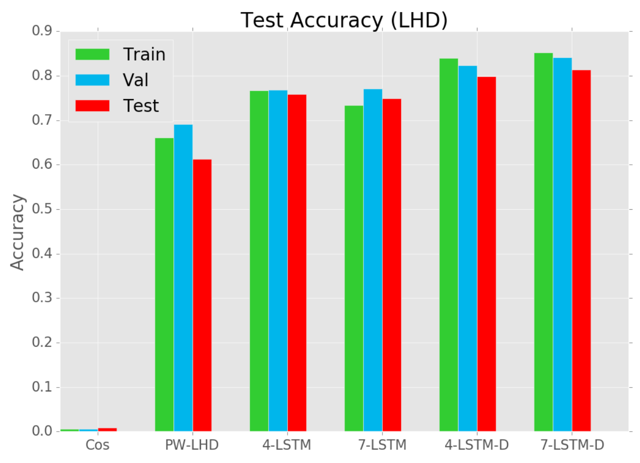

Figure 5: Accuracies for the piecewise-projection and stacked LSTM models over the LHD set.

4 Discussion

Our stacked LSTM model has achieved moderate results on both BLESS and LHD, though they are difficult to compare due to the widely varied results of the piecewise-projection model on the two datasets.

The small size of the BLESS set, the abundance of many general words (object, artifact) and the repetition of several hyponyms (castle, glove, cottage) are likely biasing both models. It is clear that the LSTM model without dropout is overfitting on the BLESS set because of its small size relative to the number of total cells in the network. Though the model still performs well on the test set, this is likely due to the test set containing words which also appear in the training and validation sets (Table 1) whose hypernyms the model has memorized.

On the other hand, it seems that the piecewise-projection is also simply learning features of the words in the BLESS set. For a low threshold, increasing the number of clusters greatly increases the model's F1 score on the validation set (Figure 3). We would expect only a modest increase as our number of clusters approaches some "true" value that describes the different levels of hypernymy that exist in English. Nevertheless, all values of k converge to the same F1 score on both the BLESS and LHD sets, and we find two very different optimal values of k for both sets. This suggests the model is not learning linguistic information about hypernymy but is rather optimizing for the structure of the BLESS set.

Nonetheless, the test accuracy that the seven-layer LSTM achieves on the LHD set shows promise for learning a hypernymy mapping. The neural net should improve (or stabilize) as the dataset grows, especially if dropout prevents it from memorizing as it trains on more data. The significant positive offset in validation accuracy during training as the batch size is increased suggests that the model may benefit from seeing more "levels" of hypernymy in a single iteration. Though the information that the network state receives at a given time step from previous time steps is not particularly clear, the broad diversity of tokens present in the LHD set suggests that hypernymy is indeed complex but not so broad that it manifests itself differently in word vectors in different domain spaces and across pairs of varying generality. The model was able to pinpoint the location of the input's hypernym with some success in spite of these complexities without making any assumptions about hypernymy's behavior, like the piecewise-projection model does.

We note that the model still has a major shortcoming in that it only predicts a single hypernym vector for a given hyponym. Hypernymy, however, is a many-to-one mapping, and several papers [4] have shown that the hypernyms of a single noun do not cluster together in vector space. As such, this stacked LSTM model, if successful, is still unable to disambiguate between the many hypernyms that a word may have. For example, there is little recourse for choosing between "fruit" and "food" for the word "apple" and also for a polysemous hypernym like "company". Even so, it would be possible train a model for different domains. More generally, the model could use the predicted vector to generate a probabilistic distribution of possible hypernyms and condition on other context words (say, those that appear with the hyponym in a given sentence) to select the most likely hypernym from this group.

5 Conclusion

We note that future studies are necessary in order to verify and improve the validity of our stacked LSTM model. If the model indeed succeeded in learning a hypernymy mapping, it should perform well at predicting the hypernyms of a noun hierarchy. As such, a well-documented hypernymy tree like that of WordNet would serve as a reliable benchmark for testing this model.

Additionally, further tests can use different word vector models to see if other learned representations, such as those in GloVe, are "better" at learning semantic relationships which are more easily parsed from the vector components. If the LSTM cell is indeed capable of identifying these relationships, similar models could be built for other semantic relationships like synonymy.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] E. A. et. al. A study on similarity and relatedness using distributional and wordnet-based approaches.

[3] M. B. et. al. How we blessed distributional semantic evaluation.

[4] R. F. et. al. Learning semantic hierarchies via word embeddings.

[5] T. M. et. al. Distributed representations of words and phrases and their compositionality.

[6] T. M. et. al. Efficient estimation of word representations in vector space.

[7] Y. B. et. al. "a neural probabilistic language model".

[8] Y. Goldberg and O. Levy. word2vec explained: Deriving mikolov et al.s negative-sampling word-embedding method.

[9] Google. word2vec, tool for computing continuous distributed representations of words.

[10] T. Kliegr. Linked hypernymys: Enriching dbpedia with targeted hypernym discovery.

[11] N. Nayak. Learning hypernymy over word embeddings.

[12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[13] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[14] J. Weeds. Learning to distinguish hypernyms and co-hyponyms.

[15] B. Wolter and H. Gyllstad. *Collocational links in the L2 mental lexicon and the influence of L1 intralexical knowledge.* 2011.

Appendices

APPENDIX I

Here, we present a list of some properties of the BLESS and Weeds datasets.

BLESS Hypernym	**Weeds Hypernym**
artifact - 91	confederate - 2
object - 71	anthropology - 2
animal - 60	knoll - 2
creature - 60	artillery - 2
chordate - 52	feature - 2
vertebrate - 52	sociability - 2
food - 43	caterpillar - 2
device - 32	therapy - 2
produce - 31	delight - 2
beast - 26	fleck - 2
implement - 25	truthfulness - 2
good - 23	murder - 2
mammal - 23	letter - 2
commodity - 23	voting - 2
vehicle - 19	significance - 2
BLESS Hyponym	**Weeds Hyponym**
castle - 15	stock - 1
hat - 14	gown - 1
glove - 12	beachhead - 1
bomber - 11	throbbing - 1
cloak -11	matron - 1
robe - 11	codification - 1
blouse - 11	anglican - 1
sweater - 11	underbrush - 1
vest - 11	scratch- 1
dress - 11	skit - 1
shirt - 11	livelihood - 1
fighter - 11	adjective - 1
sweater - 11	manager - 1
cottage - 11	nursery - 1

Table 1: **Dataset Comparison**.The top 15 most common hyponyms and hypernyms in the BLESS and Weeds data. The BLESS data contains 1337 pairings while the Weeds data contains 2564.

Hyponym	**Hypernym**
saw	artifact
banana	food
rat	rodent
beetle	arthropod
jacket	clothing
radio	system
dress	good
cabbage	food
rat	rodent
cat	beast
frigate	ship
car	vehicle
saw	artefact
saw	object
bear	vertebrate
bear	chordate
coat	clothing
salmon	food
hotel	building
pistol	device
jacket	commodity
sword	implement
bear	animal
bear	creature
cabbage	veggie
mackerel	chordate
pine	tree
flute	artifact
saw	utensil
cat	mammal

Table 2: **Indicative Pairs**. All indicative pairs for $k = 30$ clusters.