Lawrence McAfee
CS 224n
6/4/08

Document Classification using Deep Belief Nets

Overview

This paper covers the implementation and testing of a Deep Belief Network (DBN) for the purposes of document classification. Document classification is important because of the increasing need for document organization, particularly journal articles and online information. Many popular uses of document classification are for email spam filtering, topic-specific search engine (where we only want to search documents that fall under a certain topic), or sentiment detection (to determine whether a document has more of a positive or negative tone) [1].

DBNs using a mostly unsupervised clustering algorithm for training are implemented. The clustering algorithm groups documents together that fall under the same category. Once the documents are grouped, a supervised learning algorithm (SVM in this case) is used to "assign" labels to the groups. More information and a background on DBNs is given in the following section.

Testing of the DBN is primarily performed along three axes: 1) the number of hidden neurons in each layer of the DBN (which determines a "shape" for the overall network), 2) the number of layers in the DBN, and 3) the number of iterations that are used to train each layer of the DBN. Additionally, classification results of the DBN are compared with results obtained from using support vector machine (SVM) and Naïve Bayes (NB) classifiers. Some final experiments investigate how vocabulary size and word preprocessing affects performance.

Preliminary results indicate that the DBN, given the implementations used in this paper, is not a viable alternative to other forms of document classifiers. The DBN has a significantly longer training time, and no matter how many training iterations are used, the DBN appears to have worse accuracy than either the SVM or NB classifiers. A number of suggestions for improvement are given in the final section, at least one of which should greatly increase the performance of the DBN.

Background on Deep Belief Networks

A Deep Belief Network is a generative model consisting of multiple, stacked levels of neural networks that each can efficiently represent non-linearities in training data. Typically, these building block networks for the DBN are Restricted Boltzmann Machines (more on these later). The idea of a DBN is that since each Restricted Boltzmann Machine (RBM) is non-linear, then when composing several RBMs together, one would expect to be able to represent highly non-linear/highly varying patterns in the training data.

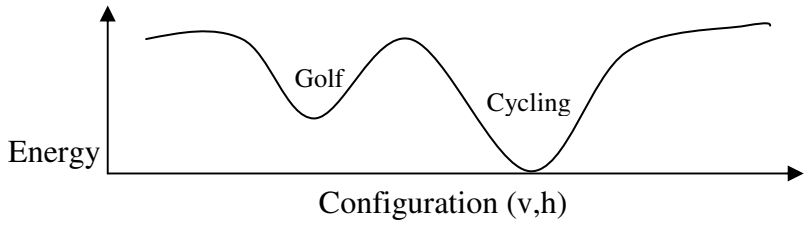
A Boltzmann Machine (the unrestricted version of an RBM) is a network that consists of a layer of visual neurons and a layer of hidden neurons. Frequently, and as

used in this paper, the neurons are binary stochastic neurons, meaning that they can only be in one of two states, "on" or "off". The probability of a neuron turning on is a sigmoid function of its bias, weights on connections it has with all other neurons in the network, and the states of all other neurons in the network. A Boltzmann machine is defined in terms of energies of configurations of the visible and hidden neurons, where a configuration simply defines the state of each neuron. To simplify for training time complexity, DBNs use the Restricted Boltzmann Machine, which only allows connections between a hidden neuron and a visible neuron, and no connections are between two visible neurons or between two hidden neurons. In an RBM, the energy is defined as:

$$Energy(v,h) = h'Wv + b'v + c'h$$

where (v,h) is the configuration, W is the weight matrix, and b and c are the bias vectors on the visible and hidden neurons, respectively. In training a Boltzmann machine, we present inputs to the network through the visible neurons, and the goal is to update the weights and biases so as to minimize the energy of the configuration of the network when the training data is used as input.

In text classification, when presenting the training data (binary bag-of-word vectors indicating the occurrence of a word) to an RBM, most documents within a certain category are expected to have similar training vectors and are therefore expected generate similar configurations on the RBM. For example, assuming we have several documents falling under the two categories "Golf" and "Cycling", after training the RBM (to be explained next), we will have an energy graph that should look something as follows:



where documents pertaining to "Golf" and "Cycling" have lower energy for their configurations because the training data is made up documents from these two categories. The probability of a configuration is determined by the energy of a configuration, where lower energy configurations have higher probability of occurring:

$$P(v,h) = \frac{1}{Z} e^{-energy}$$

where Z is a normalization constant computed by summing over energies for all possible configurations of the visible and hidden units.

Training an RBM consists of presenting a training vector to the visible units, then using the Contrastive Divergence algorithm [2] by alternatively sampling the hidden units, p(h|v), and sampling the visible units, p(v|h). Fortunately, when using an RBM, we never have to calculate the joint probability P(v,h) above, and sampling is easy:

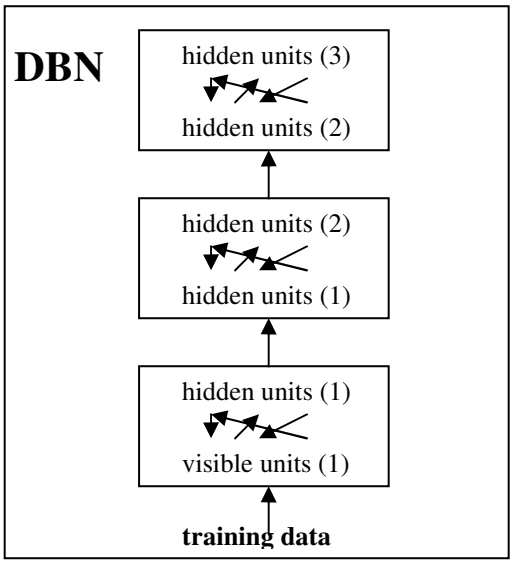
$$P(v_k = 1 \mid h) = sigm\left(-b_k - \sum_j W_{kj} h_j\right)$$
$$P(h_j = 1 \mid v) = sigm\left(-c_j - \sum_k W_{kj} v_k\right)$$

After doing just a single iteration of Gibbs sampling (according to Contrastive Divergence), we can update the weights and biases for the RBM as follows:

$$W_{kj} = W_{kj} - \alpha\left(\langle v_{k0} h_{j0} \rangle - \langle v_{k1} h_{j1} \rangle\right)$$
$$b_k = b_k - \alpha\left(\langle v_{k0} \rangle - \langle v_{k1} \rangle\right)$$
$$c_j = c_j - \alpha\left(\langle h_{j0} \rangle - \langle h_{j1} \rangle\right)$$

where α is the learning rate, v_0 is sampled from the training data, h_0 is sampled from $P(h|_{v0})$, v_1 is sampled from $P(v|h_0)$, and h_1 is sampled from $P(h|v_1)$. We repeat this update for several samples of the training data.

To train a stack of RBMs, we iteratively train each layer by first training the lowest layer from the training data, then training each next higher layer by treating the activities of the hidden units of the previous layer as the input data/visible units for the next higher layer. This is best illustrated as a picture:



The idea of the layer-by-layer training is that each layer "learns" features to represent data in the previous layer. Therefore, one would expect the highest layer to be able to capture very abstract, high-level features of the training data. The top layer is also an associative memory, where "dips" in its energy graph represent the categories of the documents in our training set.

This is a very quick introduction to DBNs and RBMs, and much more information can be found in the References section.

Methodology

Corpus and Preprocessing

The corpus used for all experiments is the Wikipedia XML Corpus [7]. The reason for using this corpus is because of my interest in learning to automatically classifying webpages, which would be very useful for document-specific searching. Other papers on classification frequently use the Reuters newswire corpus. However, my interest is primarily in discovering if DBNs are an algorithm for text classification that should be explored further. To gather a comparison against other classifiers, I run SVM and Naïve Bayes classifiers on the same corpus and compare the results.

In [7], there are multiple corpuses available. The one used for the following experiments is the "Single-Label Corpus", where each document only falls under a single category.

The Wikipedia documents come in XML format, so preprocessing had to be done to transform the raw documents into usable feature vectors. The method for preprocessing was as follows:

1) Choose a subset of categories and their corresponding articles to use as the corpus for the remaining steps.
2) Strip all tag information from each document.
3) Strip all punctuation from each document.
4) Stem each word in the document, using the Porter Stemmer [9].
5) Record list of words that occur in each document, and the number of times that word occurred in the document.
6) Once (3) is completed for each document, a master list of words and their total number of occurrences across the corpus can be formed.
7) Top k occurring words over the entire corpus are used as the feature vector. For most experiments, k=2000, except when otherwise noted.

The categories in this corpus have varying numbers of documents, and for most experiments, I chose the five categories that contained the largest number of documents. The exception is the experiment when I compare the DBN results to the SVM and Naïve Bayes classifiers, where I used 30 categories.

Also, once the above steps are complete, I divide the data into 90% training data and 10% testing data. In the case of using the top 5 categories, 300 documents are taken from each category. In the case of using the top 30 categories, 155 documents are taken from each category, since the lower categories have fewer available documents.

Representation of Training Data

All experiments in this paper were performed using binary training vectors, where features are simply a bag-of-words, and a "1" indicates that a word occurred at least once in a document, and a "0" indicates otherwise. One reason for using binary feature vectors is because the majority of available information on DBNs assumes binary features, and so attempting to use some form of continuous features would have been highly

experimental. Also, the information that is available on continuous-valued data and neurons indicates that training is much slower than with binary inputs. Given that training on binary inputs itself is a fairly slow process, training on continuous inputs would have been infeasible.

SVM on Top Layer to Form Associative Memory

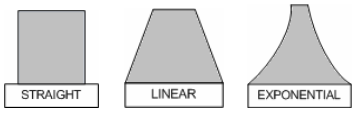
As mentioned, RBMs automatically learn features on their inputs. The top level RBM in a DBN ideally learns very abstract features that model the empirical data very well. However, these abstract features still need to be classified and associated with a specific label. To do this I used the activities of the top layer of the RBM as input data to an SVM to do supervised learning. To generate a sample of the activities from the top RBM (or from any level RBM), one inputs a training datum to the first-level RBM and propagates the activities up from one level to the next using the weights and biases on the hidden units.

"Shapes" Experiment

The first experiment I did was on trying different "shapes" for the network. The shape of the network is determined by the number of hidden neurons present in each layer. For example, if the number of hidden units is the same in each layer, I call this the "straight" shape. The number of hidden units per layer in the straight shape is not necessarily equal to the vocabulary size.

For the "linear" shape, I build the network such that on one end there is the vocabulary size, on the other end is the number of hidden units I want present in the top layer, and the number of hidden units in each intermediate layer scales such that there is a linear change in the number of hidden units from one layer to the next.

For the "exponential" shape, I start off similar to with the linear shape, except here the number of hidden units in each intermediate layer scales such that there is a certain rate of change in the number of hidden units in one layer to the next, instead of an absolute change.



Depth Experiment

For this experiment, I tested how the accuracy of the DBN changes with varying numbers of layers. I performed a depth experiment for straight-shaped networks and a depth experiment for linear-shaped networks.

Training Iterations Experiment

When training an RBM, rather than running directly through all training data, one typically samples from the training data. We can choose how many times we would like to sample from the training data, but of course should typically chose a number on the order of the number of training vectors, and generally we choose a number much larger than this[1]. In this experiment, I train the DBN for varying numbers of training iterations for each RBM layer. We should expect the DBN to perform better with a larger number of training iterations, assuming overfitting does not come into play.

Comparison to SVM, Naïve Bayes

In this experiment, I compare the results from the DBN to the results obtained when applying a support vector machine [8] and Naïve Bayes[2] directly on the training data. Two experiments were performed, one using five categories, and the other using thirty categories.

Vocabulary Size Experiment

Here, I test the DBN with varying sizes of vocabulary.

Lowercasing experiment

As mentioned, each word is stemmed in the preprocessing stage. In this last experiment, I test the performance of the DBN when I additionally lowercase all words before taking word counts. Lowercasing all words increases the variety of word types that can occur in the vocabulary of the feature vectors.

Experiment Results/Discussion

Shapes (robustness graphs for each shape)

In this experiment, the vocabulary size was 2000 words. The results for the various DBN setups (see Figure 1) are referenced by the number of hidden neurons in the top level of the network. For a straight-shaped network, this number is equal to the number of hidden neurons in all levels. For example, for 500 hidden units and 4 layers, this would constitute a DBN of (starting with the input vector) 2000-500-500-500-500. For a linear-shaped network with 1000 hidden units in the top level, this means a 2000-1750-1500-1250-1000 DBN. All tests were done with a 4-level network, and 1000 training iterations for each level.

[1] However, in the case of all other experiments, sampling on the order of the number of training vectors should be okay since I am comparing relative results, not absolute results.
[2] Code written by Lawrence McAfee

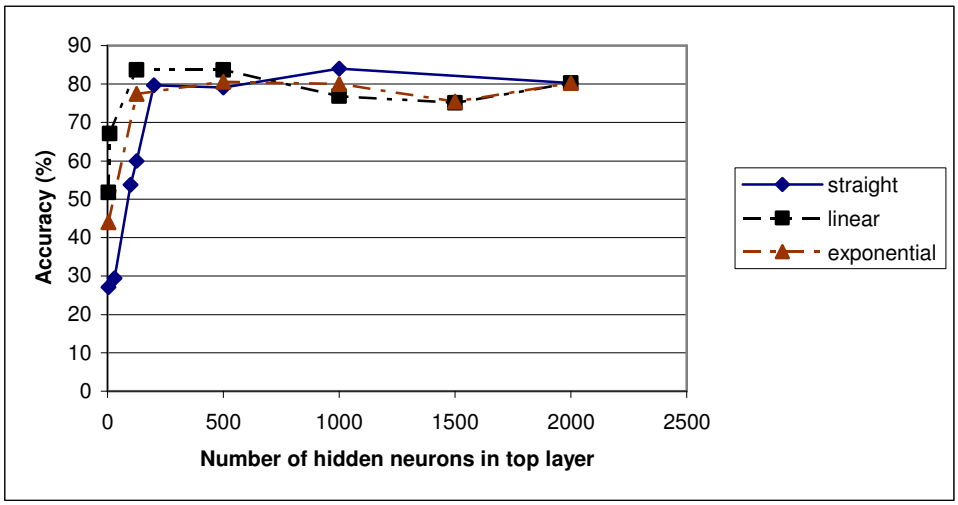


Figure 1: Performance results for various shapes of the DBN

What we see here is that the performance of the DBN is less a function of the shape of the DBN, and more a function of how many hidden units are in the top level. Once the number of hidden units in the top level cross a threshold of around 100-125 units, the performance essentially flattens at around 80% accuracy, and no longer becomes dependent on the number of hidden units in the top level.

These results indicate there is a certain minimum number of neurons needed in a layer to fully encode the features of a document. It is difficult to infer exactly what this means in the case of document classification, since the algorithm is automatically learning features of the training data, and hence we don't know what these features are. However, in the case of visual input, the features learned pertain to learning which pixels are on at the same time. So what one can assume is that the DBN is learning word relationship features, extracting properties such as which words tend to occur together for a certain category.

Depth (learning curve)

For this experiment, two different tests were run, one with a straight-shaped network, and the other with a linear-shaped network. For the straight-shaped network, 1000 hidden units were used for each level. A sweep was done from a network consisting of a single level (one RBM) to a network that consisted of seven levels. With the linear-shaped network, 500 units were chosen for the top level, and again a sweep was done from one level (i.e., 2000-500) to seven levels (i.e., 2000_1786_1572_1358_1144_930_715_500).

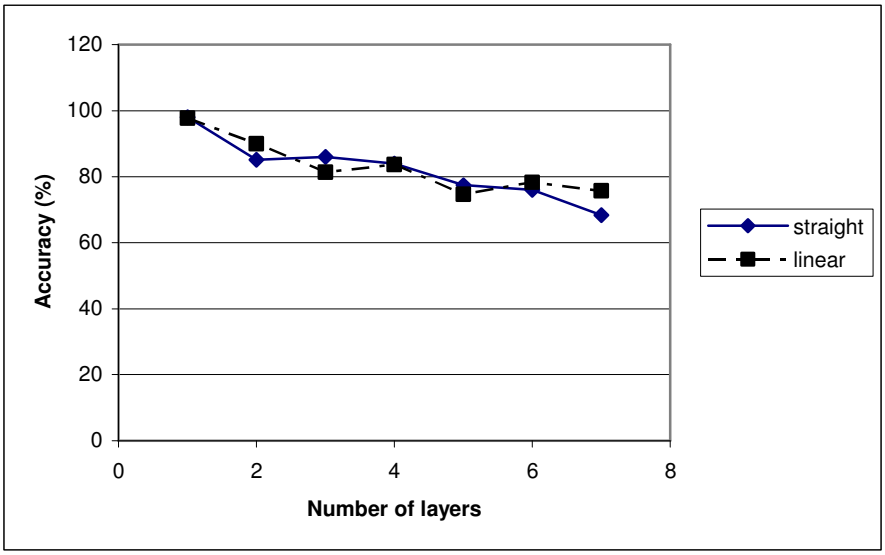


Figure 2: Depth testing for straight- and linear-shaped networks

As can be seen in Figure 2, performance tends to decrease as the number of layers increases. There are a couple of reasons for why this could be happening. The first possible cause is that using 1000 iterations to train the RBM of each level is not enough training iterations, and so the parameters for each RBM (the weights and biases) are not fully converged. There are a few reasons for why this is not a very likely explanation. First of all, from actual observations I did of printing out the percentage change of the weights and biases from one iteration to the next, the weights and biases were changing by a small fraction of a percent after only about 100-200 iterations. Second, we are only using five categories and 1000 training examples for the input data. This means that there are 200 training documents per category. In a similar experiment done with visual data on handwritten digits, there were ten digits and again 1000 training examples (100 examples per digit), and 100 training iterations for the RBMs was enough for the RBMs to converge.

The more likely explanation for why the performance decreases as we increase the number of layers is due to the representation of the training data as binary vectors. Binary vectors are not very expressive of all the information that is contained in a document. As mentioned, these feature vectors simply indicate whether a word occurred at least once in a document, and no information is given on word count or word frequency. What is most likely the cause of the poor performance is that these binary vectors do not reveal enough information about the documents of each category for the clustering algorithm to successfully group similar documents and separate dissimilar documents. The downward-spiraling effect is that the input vector is a poor representation of the document, and so poor features are learned by the first RBM in the input data, which in turn causes poor features to be learned for the second RBM, and so on. At each level, we are learning abstract features that model the input data less and less, and so at the top level RBM, the features do not model the input data very well, and as a result the performance suffers.

This could potentially be remedied if some form of information about word frequencies could be portrayed in the input data. For example, there should be a

difference between the word "protein" occurring 500 times in a category on medicine than "protein" occurring once in a category on real estate.

Number of Training Iterations

To test how the number of training iterations used to train each RBM affects the overall DBN performance, I swept the number of training iterations from 10 to 10,000. The DBN used here is a straight-shape, 4-layer, 2000-500-500-500-500 DBN. (The reason 500 hidden units is used in this experiment is because, as noted in the "shape" experiment, using any number of hidden units above about 100 units resulted in similar performance.)

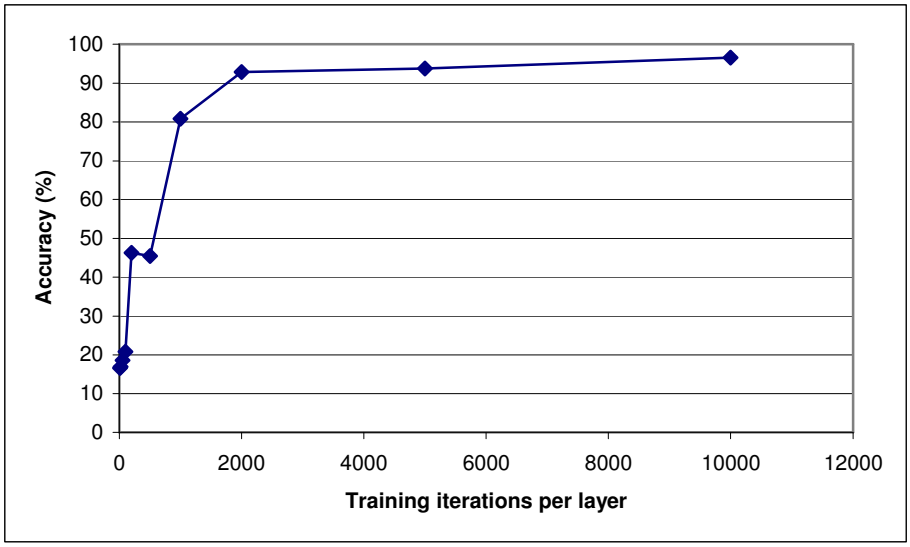
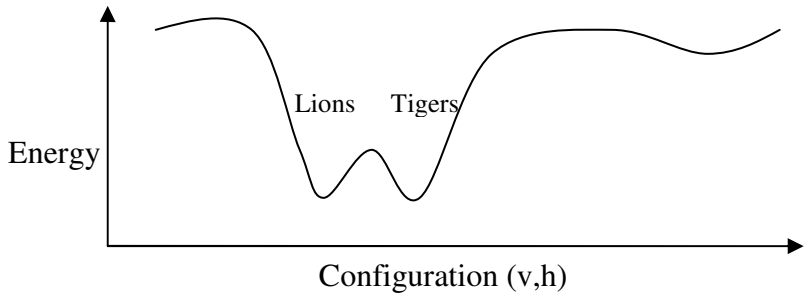
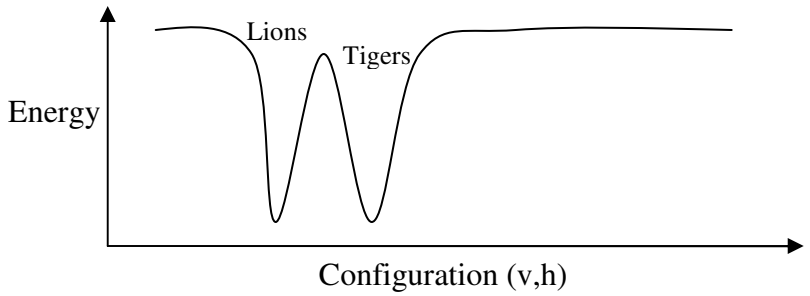


Figure 3: Accuracy results for RBM training iteration testing

As we expect, the performance increases as we train each RBM for an increasing number of iterations. This does not, however, contradict the findings in the previous experiment (i.e., that the RBMs are learning poor features for the input data) because, even though the RBMs may not be learning optimal features, the goal of the optimization algorithm that the RBMs are performing is to better and better group similar documents, and to separate dissimilar documents. In terms of the energy graph that each RBM is learning, this means that as we train for more iterations, we are creating better-defined "dips" in the energy graph. As an example, let's say we have two similar categories, "lions" and "tigers". Given the binary input vectors, the RBMs are not learning optimal features to describe these two categories. So when we train on too few iterations, we get something as follows:



However, as we train for more iterations, we eventually learn to discern between the "lions" and "tigers" categories. The features being learned at each RBM level are not as good as they should be, but given enough iterations, we will be able to learn enough to separate categories better, and get something like this:



Essentially, learning worse features means that we must train for more iterations to get similar performance. It may also be the case that even if we train for infinite iterations, we will still not yield the same performance as a network would achieve should the input data be more expressive and the neurons are setup for continuous data.

Another interesting thing to take note of from Figure 3 is the zero increase in performance from 200 iterations to 500 iterations. One would expect the performance to always increase for more training iterations. My best reasoning for why this is the case is that the distribution of the samples from the training data is not necessarily representative of the distribution of the training data. For example in one case when checking how many documents of each category had been sampled during training, I found that 166 samples had come from one category, whereas 341 had come from another category. This must have been one of the more rare cases, but inequalities in the numbers of documents chosen from each category is expected due to the stochasticity in the algorithm. Hence, highly skewed samples such as the one described above could cause learning of inadequate features.

Comparison to basic SVM, NB

As mentioned previously, this is the one experiment where 30 categories were used instead of five. The reason for this is because when using only five categories, the SVM and Naïve Bayes classifiers achieve nearly perfect performance on the test vectors (98.9% and 96.6% accuracy, respectively). The goal of comparing the DBN to the SVM and NB classifiers is to find a situation where an SVM and NB classifier perform poorly, and see if the DBN can perform better. Less than ideal performance for SVM and NB is

the case when training on 30 categories, achieving only 45.8% and 40.8% accuracy, respectively.

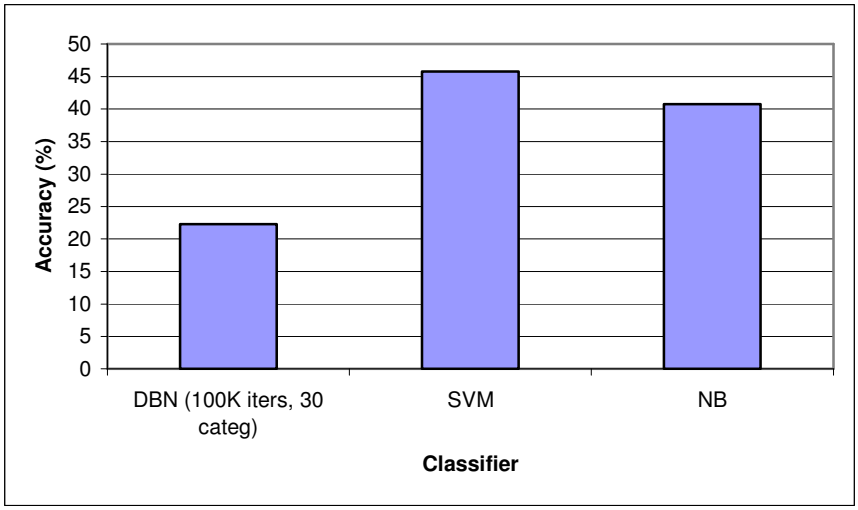


Figure 4: Comparison of DBN to support vector machine and Naïve Bayes (30 categories)

Again, a 4-level 2000-500-500-500-500 DBN was used. This time, the DBN was trained for 100,000 iterations per RBM to ensure convergence.

We see in Figure 4 that even with 100,000 training iterations, the DBN performs worse (22.3% accuracy) than both the SVM and NB classifiers. The reason is that the DBN simply is not capturing the high-level, abstract features that it should be. The reason for this again comes from the fact that using binary input data – which is not expressive of the word frequency properties of each document – does not provide a solid starting point for learning useful features. The features are useful to some extent, since a 22.3% accuracy is better than random guessing (which would be about 3%).

This can only be remedied by using either word frequency input data, or data that represents integer-valued numbers of occurrences of words, for which neurons that could take in discrete values as inputs would need to be designed. However, as it stands, the binary inputs are causing poor features to be learned, and hence the performance suffers.

In addition to the poor performance of the DBN, the training time required to complete 100,000 training iterations for the four RBMs is enormous compared to the training time for the SVM and NB classifiers. The DBN took about 2 hours 13 minutes to complete training, compared to the SVM's 4 seconds, and the NB's 3 seconds.

Size of vocabulary

The vocabulary used for training a network can have a large impact on the network's performance, since certain words are more indicative for discerning between two categories of documents. In this experiment, I swept the vocabulary size from 10 to 2000 words and recorded performance results. The network used for each vocabulary size changed with each vocabulary size. A 4-level straight-shaped network was used for each test, where the number of hidden layers in each level was equal to the vocabulary size.

(Since using 2000 hidden units for a vocabulary of size 2000 worked well in previous experiments, it is expected that this setup – of using hidden units in each layer equal to the vocabulary size – for each test should be near optimal for each vocabulary size.) Here, again, we use five categories.

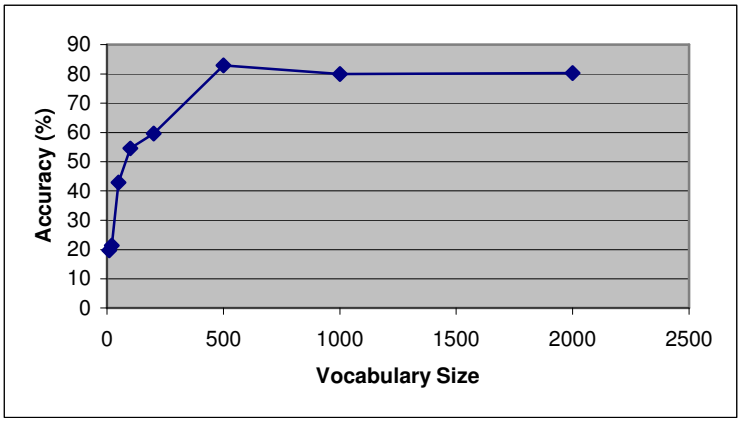


Figure 5: Vocabulary size testing

What we see is that the performance for the network continually improves up to a vocabulary size of about 500, at which point the performance levels off at around 80% again. These results tell us that, as a very rough estimate, about the first 500 words are the most indicative words in discerning between categories.

Indeed, from looking at the list of words that are included in a vocabulary size of 500, we see several words that are indicative of the different categories. For example, in the 500-word vocabulary, we see words (and their location on the list) such as football (word 202) and tackle (word 377) for the category "national football league players", words such as transmission (word 409) and Jaguar (word 357) for the category "formula one race reports", and indicative words for the other categories that included in a 500-word vocabulary, but not in a 200-word vocabulary. More indicative words occur in vocabularies larger than 500 words, but these results show us that enough of these indicative words occur in the 500-word vocabulary that the algorithm can discern reasonably well among the categories.

Lowercasing

In the final experiment, I compared the performance obtained when all the words are made lowercase before taking word counts. In the preprocessing steps listed under the methodology, this means inserting the lowercase step after the stemming step, but before taking the word counts (between steps 4 and 5). In this experiment, a vocabulary size of 2000 words is used, and the DBN is a 4-level 2000-1000-1000-1000-1000 network.

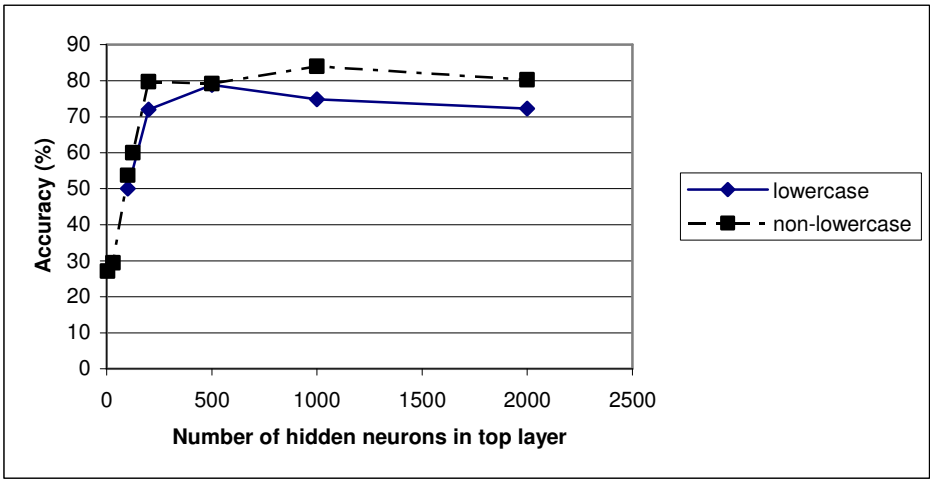


Figure 6: Lowercase testing results

Surprisingly, the results show a decrease in performance when we lowercase the words. One might expect an increase in performance, since lowercasing allows larger set of words to be including in the vocabulary, since words such as "record" and "Record" are not double counted.

However, upon closer analysis, one can see some drawbacks to having a richer vocabulary. One of those drawbacks is overfitting of the training data, since we may not necessarily need the extra words that we get when we lowercase. As we saw in the previous experiment on vocabulary size, although most of the experiments use a vocabulary size of 2000, we only need a vocabulary size of about 500 words. Since a vocabulary size of 500 words is sufficient, any larger vocabulary size is liable to overfitting, since we are introducing more words than we really need. Hence, this is most likely the reason for the reduced performance when we lowercase.

Conclusion & Suggestions for Improvement

Deep belief networks show promise for certain applications in the future, given the theory behind them. Much information has been published on the expressive power of DBNs in [5] and [6]. However, the results of my experiments show that DBNs are perhaps more sensitive to having an appropriate implementation for a given application than other learning algorithms/classifiers. Even though the same binary training data was presented to both the DBN and the SVM and NB classifiers, the SVM and NB classifiers outperformed the DBN for the document classification application. As mentioned previously, the most likely reason for this is due to the fact that DBNs iteratively learn "features-of-features" in each level's RBM. If the network has an appropriate implementation for the task at hand, this will potentially lead to a very high accuracy classifier (in [10] Hinton describes how a DBN can be built to outperform other types of classifiers on the task of digit recognition). However, if the network and data do not work perfectly well together, this feature-of-feature learning can potentially lead to recursively learning features that do not appropriately model the training data. This is what I believe to be the case in the experiments described in this paper, since the DBN

performed far better than random guessing, but was outperformed by the relatively stock SVM and NB classifiers.

However, there are a number of improvements that can be made that may improve the performance of the DBN for the document classification task:

1) **Use an appropriate continuous-valued neuron**. This may include either linear neurons or Gaussian neurons. Bengio describes these two types of neurons and how to sample/update parameters in his paper [3]. However, continuous-valued neurons are slower to train, and not much or any work has been done to apply these types of neurons to document classification.
2) **Implement backpropagation to fine-tune weights and biases**. Hinton [10] suggests that backpropagation be used to propagate error derivatives from the top-level back to the inputs to fine-tune the RBM parameters. This must only be performed after unsupervised layer-wise training has completed. The unsupervised layer-wise training of the parameters is considered to be a very good initial setting of all the parameters, and the backpropagation is just used to slightly modify the parameters. Backpropagation cannot be used as an end-all solution because without a good initial setting of the parameters, backpropagation is liable to get stuck in local optima.
3) **Spelling correction**. This is expected to have the least effect, particularly for the Wikipedia corpus. Most words are already assumed to be spelled correctly, and so the benefits of spell correction should be minimal.

Of these improvements, the one that is expected to have the greatest effect is to change the type of neuron used. Doing this, along with using training vectors consisting of word frequencies, will make it possible to better represent documents, and should improve performance.

References

[1] Manning, C. D., Raghavan, P., Schutze, H. *Introduction to Information Retrieval*. Cambridge University Press. Chp. 13. 2008.

[2] Hinton, G. E., Osindero, S. and Teh, Y. *A fast learning algorithm for deep belief nets*. 2006.

[3] Bengio, Y., Lamblin, P., Popovici, P., Larochelle, H. *Greedy Layer-Wise Training of Deep Networks.* NIPS 2006.

[4] Hinton, G. E. and Salakhutdinov, R. R. *Reducing the dimensionality of data with neural networks.* Science, Vol. 313. no. 5786, pp. 504 - 507, 28 July 2006.

[5] Bengio, Y. *Learning Deep Architectures for AI*. 2007

[6] Bengio, Y., LeCunn, Y. *Scaling Learning Algorithms towards AI*. 2007.

[7] Wikipedia XML Corpus. http://www-connex.lip6.fr/~denoyer/wikipediaXML/.

[8] LibSVM Library for Matlab. http://www.csie.ntu.edu.tw/~cjlin/libsvm/.

[9] Java Implementation of Porter Stemmer.
http://www.ils.unc.edu/~keyeg/java/porter/index.html.

Deep Belief Networks

[1],[3] above

[10] Geoffrey Hinton's Tutorial on DBNs. Deep Learning Workshop, NIPS07.
URL: http://nips.cc/Conferences/2007/Program/event.php?ID=572

Restricted Boltzmann Machines

[11] Geoffrey Hinton's CSC321 Course, Lecture 19. University of Toronto
URL: http://www.cs.toronto.edu/~hinton/csc321/lectures.html

[13] Geoffrey Hinton's Scholarpedia article on Boltzmann Machines.
URL: http://www.scholarpedia.org/article/Boltzmann_machine