Large-Scale Learning of Embeddings with Reconstruction Sampling

Yann N. Dauphin[(1)] DAUPHIYA@IRO.MONTREAL.CA
Xavier Glorot[(1)] GLOROTXA@IRO.MONTREAL.CA
Yoshua Bengio[(1)] BENGIOY@IRO.MONTREAL.CA
[(1)] Dept. IRO, Université de Montréal. Montréal (QC), H3C 3J7, Canada

Abstract

In this paper, we present a novel method to speed up the learning of embeddings for large-scale learning tasks involving very sparse data, as is typically the case for Natural Language Processing tasks. Our speed-up method has been developed in the context of Denoising Auto-encoders, which are trained in a purely unsupervised way to capture the input distribution, and learn embeddings for words and text similar to earlier neural language models. The main contribution is a new method to approximate reconstruction error by a sampling procedure. We show how this approximation can be made to obtain an unbiased estimator of the training criterion, and we show how it can be leveraged to make learning much more computationally efficient. We demonstrate the effectiveness of this method on the Amazon and RCV1 NLP datasets. Instead of reducing vocabulary size to make learning practical, our method allows us to train using very large vocabularies. In particular, reconstruction sampling requires 22x less training time on the full Amazon dataset.

1. Introduction

In recent years, there has been a surge of interest for unsupervised representation learning algorithms, often for the purpose of building deep hierarchies of features [1]. See (Bengio, 2009) for a recent review of Deep Learning algorithms, which are based on unsupervised learning of representations, one layer at a time, in order to build more abstract higher-level representations by the composition of lower-level ones. These representations are often used as input for classifiers, and measuring classification error is a good way, also chosen here, for evaluating the usefulness of these representations. One problem with these unsupervised feature learning approaches is that they often require computing a mapping from the learned representation back into the input space, e.g., either to reconstruct the input, denoise it, or stochastically generate it. Consider learning tasks where the input space is huge and sparse, as in many Natural Language Processing (NLP) tasks. In that case, computing the representation of the input vector is very cheap because one only needs to visit the non-zero entries of the input vector, i.e., multiply a very large dense matrix by a very sparse vector. However, **reconstructing a huge sparse vector** involves computing values for all the elements of that vector, and this can be much more expensive. For example with a bag-of-words representation of a 100-word paragraph and a vocabulary size of 100,000 words, computing the reconstruction from the representation is 1000 times more expensive than computing the representation itself.

The **main contribution** of this work starts from a very simple idea: train to **reconstruct only the non-zeros and a random subset of the zeros.** This introduces a bias in the reconstruction error (giving more weight to non-zeros than to zeros), which can be potentially beneficial or detrimental, but that can be corrected by a reweighting of error terms. The idea has also been refined in the context of the Denoising Auto-encoder, used for unsupervised learning of the embeddings in our experiments. Instead of focusing only on the non-zeros of the uncorrupted input, we include also the non-zeros of the corrupted input, in order to sample the inputs on which the error is most likely to be large (since this minimizes the variance of our sampling-based estimator).

[1] see NIPS'2010 Workshop on Deep Learning and Unsupervised Feature Learning, http://deeplearningworkshopnips2010.wordpress.com/

Appearing in *Proceedings of the 28th International Conference on Machine Learning*, Bellevue, WA, USA, 2011. Copyright 2011 by the author(s)/owner(s).

2. Related Work

There has been much previous work on learning embeddings for NLP. See (Bengio, 2008) for a review in the context of neural-network based models, which are related to the approach described here. A core computational limitation of these models is that the neural network prediction (e.g., of the next word given previous words) consists of a probability for each word in the vocabulary, which makes computation scale with vocabulary size. In early work, this was addressed by limiting the vocabulary of the predicted words (and possibly using a cheaper predictor such as n-grams for the other ones).

In order to address this computational limitation and scale to larger vocabularies and larger datasets, two kinds of approaches were introduced in the past: using a tree structure for the predictions, or using sampling to visit only a few of the possible words. The approach introduced here is of the second kind. Tree-structured predictors are based on learning a class hierarchy and require only visiting the path from the root to the leaf corresponding to the observed word (Morin and Bengio, 2005; Mnih and Hinton, 2009). Sampling-based algorithms rely on stochastic approximations of the gradient which only require to compute the prediction on a small subset of the words (Bengio and Sénécal, 2003; 2008; Collobert and Weston, 2008). Whereas the early attempts (Bengio and Sénécal, 2003; 2008) are focused on correctly estimating conditional probabilities (for the next word), Collobert and Weston (2008) only try to *rank* the words, with a criterion that can be written as a sum over words (comparing the score of the observed word with the score of any other word). This sum can be estimated by a Monte-Carlo sample. This works even with a single sample in the context of stochastic gradient descent, where we do a very large number of stochastic updates but each of them is small, hence averaging out much of the sampling noise. Whereas all these focused on predicting the next word, we focus here on reconstructing an input bag-of-words, or more generally a very sparse high-dimensional vector, since this kind of reconstruction is a basic requirement for many Deep Learning algorithms. We are not trying to predict word probabilities, only to learn good embeddings (which are used as part of a classifier) so we do not really need the reconstruction outputs to be calibrated probabilities.

3. Denoising Auto-Encoders

3.1. Introduction

In this paper we have applied the proposed idea of sampling reconstructions in the context of the Denoising Auto-Encoder (DAE) as the building block for training deep architectures, because our preliminary experiments found that a particular form of DAE surpassed the state-of-the-art in a text categorization task of sentiment analysis (Glorot *et al.*, 2011). The DAE is a learning algorithm for unsupervised feature extraction (Vincent *et al.*, 2008): it is provided with a stochastically corrupted input and trained to reconstruct the original clean input. Its training criterion can be shown to relate to several training criteria for density models of the input, either via bounds (Vincent *et al.*, 2008) or through Score Matching (Hyvärinen, 2005; Vincent, 2010). Intuitively, the difference vector between the reconstruction and the input is the model's guess as to the direction of greatest increase in the likelihood, whereas the difference vector between the noisy corrupted input and the clean original is nature's hint of a direction of greatest increase in likelihood (since a noisy version of a training example is very likely to have a much lower probability under the data generating distribution than the original). It can also be shown that the DAE is extracting a representation that tries to preserve as much as possible of the information in the input (Vincent *et al.*, 2008).

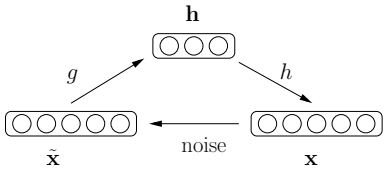

Figure 1. Schematic of the Denoising Auto-Encoder

The Denoising Auto-Encoder reconstruction $f(\mathbf{x}) = h(g(\mathbf{x}))$ is composed of an encoder function $g(\cdot)$ and a decoder function $h(\cdot)$ (see Figure 1). During training, the input vector $\mathbf{x} \in [0,1]^{d_x}$ is partially and randomly corrupted into the vector $\tilde{\mathbf{x}}$. The encoder takes $\tilde{\mathbf{x}}$ and maps it into a hidden representation $\mathbf{h} \in [0,1]^{d_h}$. The decoder takes the representation $\mathbf{h}$ and maps it back to a vector $\mathbf{z}$ in the input space ($[0,1]^{d_x}$ in our case). The DAE is trained to map a corrupted input $\tilde{\mathbf{x}}$ into the original input $\mathbf{x}$ such that $g(h(\tilde{\mathbf{x}})) \approx \mathbf{x}$. This forces the code $\mathbf{h}$ to capture important and robust features of $\mathbf{x}$. Many corruption processes are possible, but they should have the property of generally producing *less plausible* examples. Typically, inputs are corrupted by randomly setting elements of $\mathbf{x}$ to 0 or 1, or adding Gaussian noise. The encoder function used in Vincent *et al.* (2008) is

$$\begin{aligned} \mathbf{a}_1 &= \mathbf{W}^{(1)}\tilde{\mathbf{x}} + \mathbf{b}^{(1)} \\ \mathbf{h} &= s_1(\mathbf{a}_1) \end{aligned} \quad (1)$$

where s_a is a non-linear function like the sigmoid $s_a(u) = 1/(1+\exp(-u))$, $\mathbf{W}^{(1)}$ is a $d_h \times d_x$ weight

matrix and $\mathbf{b}^{(1)}$ is a $d_h \times 1$ vector. In Vincent *et al.* (2008) the function computed by the decoder is

$$\begin{aligned} \mathbf{a}_2 &= \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)} \\ \mathbf{z} &= s_2(\mathbf{a}_2) \end{aligned} \tag{2}$$

Where $\mathbf{W}^{(2)}$ is a $d_x \times d_h$ weight matrix and $\mathbf{b}^{(2)}$ is a $d_x \times 1$ vector.

3.2. Training

Given a dataset $D_n = (\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(n)})$, the parameters ($\mathbf{W}^{(1)}$, $\mathbf{b}^{(1)}$, $\mathbf{W}^{(2)}$, $\mathbf{b}^{(2)}$) are trained by stochastic gradient descent, following Vincent *et al.* (2008), to minimize the cross-entropy

$$\begin{aligned} \hat{R}(f, D_n) &= \frac{1}{n}\sum_{i}^{n} L(\mathbf{x}^{(i)}, f(\tilde{\mathbf{x}}^{(i)})) \\ L(\mathbf{x}, \mathbf{z}) &= \sum_{k}^{d} H(\mathbf{x}_k, \mathbf{z}_k) \\ &= \sum_{k}^{d} -[\mathbf{x}_k \log \mathbf{z}_k + (1 - \mathbf{x}_k)\log(1 - \mathbf{z}_k)] \end{aligned}$$

where H is the cross-entropy, $\mathbf{x}$ and $\mathbf{z}$ are considered as *vectors of binomial probabilities.*

3.3. Motivation

The Denoising Auto-Encoder can learn a representation from unlabeled data, but it can be later fine-tuned using labeled data. The ability to exploit large quantities of unlabeled data is very important because labeled data are usually scarce. Obtaining labeled data usually requires paying for the manual labeling of unlabeled samples. Furthermore, in the context of Natural Language Processing, the World Wide Web is a gold mine of unlabeled data. In contrast, using a purely supervised training approach (i.e. SVMs, CRFs) can only exploit the scarce labeled data.

The hypothesis that has been confirmed earlier for specific datasets (Vincent *et al.*, 2008) is that the representation $\mathbf{h}$ learned by the DAE makes the statistical structure of the input clearer, in the sense that it can be advantageous for initializing a supervised classifier. It has been shown that Auto-Encoders (especially deep ones) go beyond Principal Component Analysis (PCA) by capturing multi-modal interactions in the input distribution (Japkowicz *et al.*, 2000; Hinton and Salakhutdinov, 2006). In other words, the encoder learns to project $\mathbf{x}$ into a space $\mathbf{h}$ where the original factors of variation of the data tend to be better separated. Experimental results show that using $\mathbf{h}$ instead of $\mathbf{x}$ as input to a classifier can significantly help achieve better generalization (Erhan *et al.*, 2010; Larochelle *et al.*, 2007).

4. Scaling the Denoising Auto-Encoder

4.1. Challenges

The dot products involved in the training of the DAE are expensive. The computations involved in g, h, the gradient ∇g through g, and the gradient ∇h through h are all in $O(d_x \times d_h)$, where d_x is the size of the sparse input vector and d_h is the size of the representation code.

This is problematic in the context of Natural Language Processing because the desired input size d_x may be in the millions.

4.2. Scaling the Encoder: Sparse Dot Product

We can take advantage of sparsity in any dot product $\mathbf{u}\cdot\mathbf{v}$ because the null elements $\mathbf{u}_i$ or $\mathbf{v}_i$ do not influence the result. This is also true for the matrix-vector product. Therefore, we can reduce the cost of the dot product by computing only the operations linked to non-zero elements, i.e., $g \in O(d_{NNZ} \times d_h)$, where d_{NNZ} is the number of non-zero elements in $\mathbf{x}$. The theoretical speed-up would be ${}^{d_x}/_{d_{NNZ}}$. This also applies to the gradient with respect to $\mathbf{W}^{(1)}$ (${}^{\partial \hat{R}}/_{\partial \mathbf{W}^{(1)}} = {}^{\partial \hat{R}}/_{\partial \mathbf{a}_1}\mathbf{x}'$). In practice, the speed-up is smaller because working with dense vector and matrix multiplications can be done more efficiently on modern computers, i.e., there is an overhead for handling sparse vectors. In our experiments we have found the overhead to be on the order of 50%. On the other hand, the biggest loss in comparison to a dense implementation comes from losing the use of BLAS' optimized matrix-matrix product (GEMM), when training the model by showing one minibatch at a time (e.g. 10 in our experiments). The speedup from the dense matrix-matrix multiplication is on the order of 3 in our experiments, hence for the sparse computation to be advantageous, the sparsity level must be high enough to compensate for these two disadvantages).

4.3. Scaling the Decoder: Reconstruction Sampling

We introduce *reconstruction sampling* to make the decoder scalable. The idea is to calculate the reconstruction cost L based only on a sub-sample of the input units:

$$\hat{L}(\mathbf{x}, \mathbf{z}) = \sum_{k}^{d} \frac{\hat{\mathbf{p}}_k}{\mathbf{q}_k} H(\mathbf{x}_k, \mathbf{z}_k)$$

where we introduce $\hat{\mathbf{p}} \in \{0, 1\}^{d_x}$ with $\hat{\mathbf{p}} \sim P(\hat{\mathbf{p}}|\mathbf{x})$, and scalar weights $1/\mathbf{q}$. The sampling pattern $\hat{\mathbf{p}}$ controls whether a given input unit will participate in the learning objective for this presentation of the example $\mathbf{x}$. If training iterates through examples in the training set, the next time $\mathbf{x}$ is seen again, a different pattern

$\hat{\mathbf{p}}$ may be sampled. In this paper, we have found that an effective sampling procedure is to choose $P(\hat{\mathbf{p}}|\mathbf{x})$ to reconstruct all non-zero inputs and a set of randomly chosen zero inputs. The average number of sampled units is defined as d_{SMP}.

The scalar weights $1/\mathbf{q}_k$ allow us to compensate for non-uniform choices of the sampling probabilities for the corresponding binary random variables $\hat{\mathbf{p}}_k$. If $\mathbf{q}_k = E[\hat{\mathbf{p}}_k|k, \mathbf{x}, \tilde{\mathbf{x}}]$, then this is an **importance sampling** scheme, i.e., the expected cost is guaranteed to be unchanged by the sampling procedure since $E[\frac{\hat{\mathbf{p}}_k}{\mathbf{q}_k}|k, \mathbf{x}, \tilde{\mathbf{x}}] = 1$. This is related to but different from (Bengio and Sénécal, 2003), in which $\mathbf{x}$ is a one-hot vector indicating what is the next word, and there is less information about which bits it matters most to sample.

We propose $\hat{L}$ instead of L as a (stochastic) training objective because it can be computed more efficiently, and we empirically find that it yields similar solutions for the same number of training updates. If $\mathbf{p}_k = 0$, then $\mathbf{z}_k$ need not be computed since it does not influence the cost $\hat{L}$. Calculating each $\mathbf{z}_k = s_2(\mathbf{W}_k^{(2)}\mathbf{h} + \mathbf{b}_k^{(2)})$ is on the order of $O(d_h)$. Therefore, computing only the units $\mathbf{z}_k$ that are sampled yields

$$h \in O(d_{SMP} \times d_h)$$

with the expected speed-up of ${}^{d_x}/_{d_{SMP}}$.

The gradients for $\hat{L}$ can also be calculated more efficiently. The gradient for the elements $\mathbf{z}_k$ where $\mathbf{p}_k = 0$ is null, so $\frac{\partial \hat{R}}{\partial \mathbf{z}}$ contains only d_{SMP} non-zero values. We calculate the gradients using the sparse dot product presented in section 4.2:

$$\begin{aligned} \frac{\partial \hat{R}}{\partial \mathbf{W}^{(2)}} &= \frac{\partial \hat{R}}{\partial \mathbf{a}_2}\mathbf{h}' \\ \frac{\partial \hat{R}}{\partial \mathbf{h}} &= \mathbf{W}^{(2)\prime}\frac{\partial \hat{R}}{\partial \mathbf{a}_2} \end{aligned}$$

The speed-up for both operations is on the order of ${}^{d_x}/_{d_{SMP}}$.

Sampling probabilities The optimal sampling probabilities $P(\hat{\mathbf{p}}|\mathbf{x})$ are those that yield the minimum variance of the estimator (under the assumption that we choose the weights $1/\mathbf{q}$ in order to get an unbiased estimator), since the total error of the sampling-based estimator is variance plus bias squared (and we are setting the bias to 0). Like for importance sampling, the minimum variance is achieved when $P(\hat{\mathbf{p}}|\mathbf{x})$ is proportional to the absolute value of the original distribution (uniform here) times the integrand, which here is just the reconstruction loss. Hence we should ideally pick those bits k on which the model is most likely to make a large error, but of course we do not know that before we sample which ones to reconstruct. The heuristic we propose is to always pick those bits k on which either $\mathbf{x}_k = 1$ or $\tilde{\mathbf{x}}_k = 1$, and to pick the same number of bits randomly from the remainder. Let $\mathcal{C}(\mathbf{x}, \tilde{\mathbf{x}}) = \{k : \mathbf{x}_k = 1 \text{ or } \tilde{\mathbf{x}}_k = 1\}$. Then we choose to reconstruct unit k with probability

$$P(\hat{\mathbf{p}}_k = 1|\mathbf{x}_k) = \begin{cases} 1 & \text{if } k \in \mathcal{C}(\mathbf{x}, \tilde{\mathbf{x}}) \\ |\mathcal{C}(\mathbf{x}, \tilde{\mathbf{x}})|/d_x & \textit{otherwise} \end{cases} \quad (3)$$

The motivation for this heuristic is that because of the input sparsity, the 1's tend to come more as a surprise than 0's, and hence yield a larger reconstruction error. Regarding the cases where the auto-encoder input $\mathbf{x}_k = 1$ when $\mathbf{x}_k \neq 1$, these also tend to yield large errors, because the auto-encoder has to uncover the fact that those bits were flipped due to the corruption process, and cannot just copy them from the input. A smaller-variance estimator could probably be obtained by numerically estimating the average error associated to different bits depending on whether or not it is a 1 or a 0 in $\mathbf{x}_k$ and $\tilde{\mathbf{x}}_k$, but we have found that with this simple scheme we achieve the same accuracy curve (as a function of number of updates) as with the dense (not sampled) training scheme, hence there is not much room left for improvement. In fact, it is questionable whether perfectly unbiasing the sampling scheme (i.e. choosing corrections $\mathbf{q}_k = P(\hat{\mathbf{p}}_k = 1)$) is what most helps produce the most useful embeddings, e.g., as measured by classification error from the learned intermediate features, on a predictive task of interest. For example, it could be argued that in the case of sparse input vectors, the non-zero inputs provide more important information and that error on them should be penalized more, which would argue in favor of choosing weights $1/\mathbf{q}_k$ constant (e.g. 1). We therefore experiment with both the unbiased scheme (eq. 3) and a biased scheme ($\mathbf{q}_k = 1$).

5. Implementation

5.1. Encoder

We implement the encoder as:

$$\mathbf{h} = s_1(\texttt{SparseDot}_{CSR}(\tilde{\mathbf{x}}, \mathbf{W}^{(1)}) + \mathbf{b}^{(1)})$$

$\texttt{SparseDot}_{CSR}(\mathbf{A}, \mathbf{B}) = \mathbf{AB}$. Note the operation is transposed compared to equation 1. In this setting, $\mathbf{W}^{(1)}$ is a $d_x \times d_h$, $\mathbf{b}^{(1)}$ is $1 \times d_h$. $\texttt{SparseDot}_{CSR}$ is more efficient when the sparse operand appears first. The input $\mathbf{x}$ and $\tilde{\mathbf{x}}$ are stored in Compressed Sparse Row (CSR) format.

The gradient is given by:

$$\frac{\partial \hat{R}}{\partial \mathbf{W}^{(1)}} = \texttt{SparseDot}_{CSC}(\mathbf{x}', \frac{\partial \hat{R}}{\partial \mathbf{a}_1})$$

Where $\mathbf{x}^T$ is in Compressed Sparse Column (CSC) format.

For reference, the implementation of $\texttt{SparseDot}_{CSR}$ is given in Algorithm 1. The implementation of $\texttt{SparseDot}_{CSC}$ is similar. $\texttt{NON-ZERO-INDICES}(\mathbf{u})$ returns the set of non-zero indices in the row vector $\mathbf{u}$. $\texttt{AXPY}(\alpha, \mathbf{A}, \mathbf{B}) = \alpha\mathbf{A}+\mathbf{B}$ is part of the BLAS programming interface (Lawson *et al.*, 1979).

In our experiments, AXPY is provided by the highly optimized Goto BLAS (Goto and Geijn, 2008). Note that this is an important optimization. Typical implementations of Auto-Encoders rely on BLAS for their dot products, our operations must also leverage BLAS to be competitive.

Algorithm 1 $\texttt{SparseDot}_{CSR}(\mathbf{A}, \mathbf{B})$

Input: $\mathbf{A} = [A_{ij}]_{M\times K}$, $\mathbf{B} = [B_{ij}]_{K\times N}$
Output: $\mathbf{C} = [C_{ij}]_{M\times N}$

for $m = 1$ **to** M **do**
  for all $k \in \texttt{NON-ZERO-INDICES}(\mathbf{x}_m)$ **do**
    $\mathbf{C}_m \leftarrow \texttt{AXPY}(\mathbf{A}_{mk}, \mathbf{B}_k, \mathbf{C}_m)$
  end for
end for

5.2. Decoder

The decoder is implemented as:

$$\mathbf{z} = s_2(\texttt{SamplingDot}(\mathbf{h}, \mathbf{W}^{(2)}, \hat{\mathbf{p}}) + \mathbf{b}^{(2)})$$

$\texttt{SamplingDot}(\mathbf{A}, \mathbf{B}, \mathbf{C})$ outputs $\mathbf{C} \circ (\mathbf{AB})$, where $\circ$ is element-wise multiplication. In comparison with equation 2, the operations are transposed and $\mathbf{W}^{(2)}$ is supposed to be $d_h \times d_x$ while $\mathbf{b}^{(2)}$ is $1 \times d_x$.

The key differences between SamplingDot and $\mathbf{C} \circ (\mathbf{AB})$ are:

1. The values in $\mathbf{AB}$ set to 0 by the element-wise product with $\mathbf{C}$ are not calculated.
2. The $\mathbf{B}$ matrix is expected to be $d_x \times d_h$ instead of $d_h \times d_x$. In other words, SamplingDot assumes $\mathbf{B}$ is transposed. This allows cache-friendly traversal of that matrix. This is especially important because in our setting $\mathbf{B}$ is a huge matrix.

$\mathbf{W}^{(2)}$ is stored as $d_x \times d_h$ instead of $d_h \times d_x$.

The gradients are calculated as:

$$\frac{\partial \hat{R}}{\partial \mathbf{W}^{(2)}} = \texttt{SparseDot}_{Dense}(\frac{\partial \hat{R}}{\partial \mathbf{a}_2}', \mathbf{h})$$
$$\frac{\partial \hat{R}}{\partial \mathbf{h}} = \texttt{SparseDot}_{Dense}(\frac{\partial \hat{R}}{\partial \mathbf{a}_2}, \mathbf{W}^{(2)})$$

$\texttt{SparseDot}_{Dense}$ calculates the dot product between a sparse matrix represented in dense format and a dense matrix. As explained in section 4.3, $\frac{\partial \hat{R}}{\partial \mathbf{a}_2}$ contains very few non-zero elements. It isn't converted into a sparse representation because the conversion is expensive and would have to be performed for each training update.

The implementation for SamplingDot is given in Algorithm 2. The implementation of $\texttt{SparseDot}_{Dense}$ is similar to Algorithm 1. $\texttt{DOT}(\mathbf{u}, \mathbf{v}) = \mathbf{u} \cdot \mathbf{v}$ is the vector dot product. It is part of the BLAS programming interface and is implemented by Goto BLAS in our experiments.

Algorithm 2 $\texttt{SamplingDot}(\mathbf{A}, \mathbf{B}, \mathbf{C})$

Input: $\mathbf{A} = [A_{ij}]_{M\times K}$, $\mathbf{B} = [B_{ij}]_{N\times K}$, $\mathbf{C} = [C_{ij}]_{M\times N}$
Output: $\mathbf{D} = [D_{ij}]_{M\times N}$

for $m = 1$ **to** M **do**
  for $n = 1$ **to** N **do**
    if $\mathbf{C}_{mn} \neq 0$ **then**
      $\mathbf{D}_m \leftarrow \texttt{DOT}(\mathbf{A}_m, \mathbf{B}_n)$
    end if
  end for
end for

6. Experiments

We perform two sets of experiments on two popular NLP datasets. First, we show the properties and effectiveness of our approach in a setting where we can compare with the non-sampled version of the training algorithm for DAEs. Second, we train large-scale models on the Amazon dataset.

Amazon Multi-Domain Sentiment Dataset. Sentiment analysis aims to determine the judgment of a writer given a textual comment. We investigate our reconstruction sampling method on the Amazon sentiment analysis data set, introduced by Blitzer *et al.* (2007). It proposes a collection of more than 340,000 product reviews on 25 different domains. For tractability, a smaller and more controlled dataset has been released, containing four different domains, with 1000 positives and 1000 negatives examples for each domain and a few thousands of unlabeled data. Our experiments will be conducted on both versions, we will refer to this last version as "small Amazon" and to the complete set as "full Amazon."

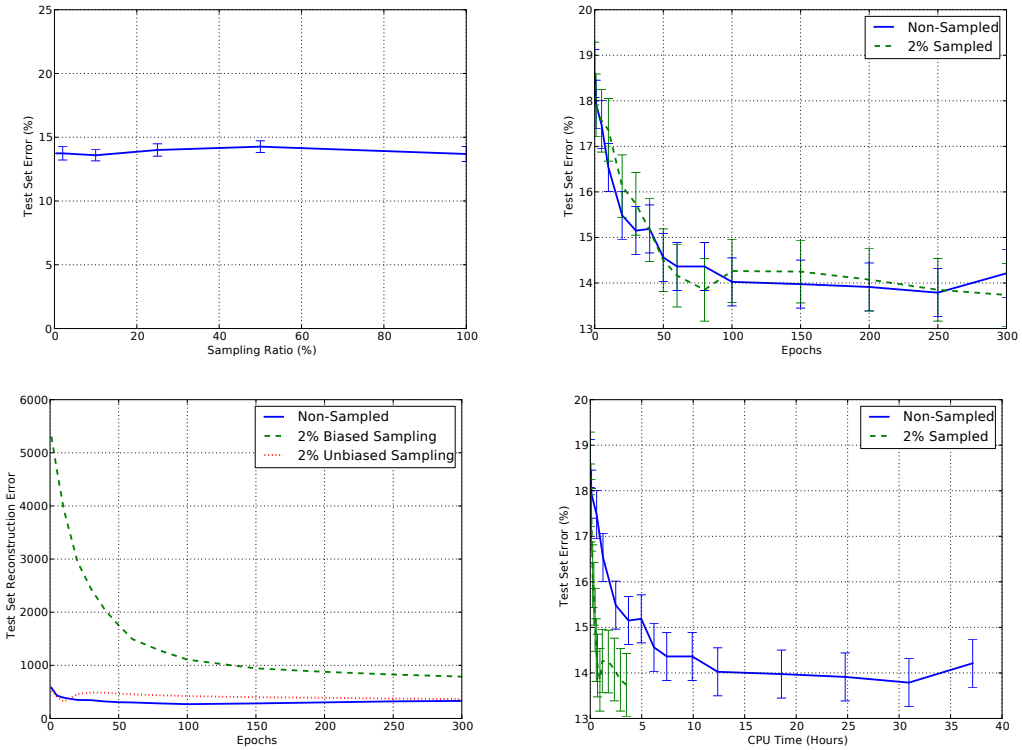

Figure 2. Experimental Results on Amazon (small set). Increasing the sampling approximation does not hurt classification error, but yields a 10.5x speedup. The biased estimator gets a worse reconstruction error, but not the unbiased one, and both convergence curves (in terms of training epochs) are similar for all models.

Reuters Corpus Volume I (RCV1) is a popular benchmark for document classification (Lewis *et al.*, 2004). It consists of over 800,000 real-world news wire stories represented in bag-of-words vectors with 47,236 dimensions. The dataset is split into a training set with 23,149 documents and a test set with 781,265 documents. The are three categories of labels to predict: Topics, Industries and Regions. There are 103 non-mutually exclusive topics. We focus our experiment on predicting the topics of documents. As proposed by (Lewis *et al.*, 2004) the performance measure is the $F_{1.0}$ over the test set.

Training Methodology. In our experiments, we perform unsupervised feature extraction using DAEs and we use these features as input to classifier. On the Amazon dataset, we train linear SVMs (Fan *et al.*, 2008). On the RCV1 dataset, a logistic regression is used.

We train a set of baseline DAEs that perform no sampling as well as a set of DAEs that have multiple levels of sampling. On the small Amazon and RCV1 dataset, we reduce the vocabulary to the 5000 most frequent input tokens in order to make the training of the baseline practical. On the full Amazon dataset, we kept 25,000 dimensions.

DAEs are trained with a minibatch size of 10. We reserve 10% of the training set of each dataset as a validation set. All hyper-parameters, for the DAEs and the classifiers, are chosen based on the performance on the validation set. We monitor validation and test error at different training epochs.

On the Amazon datasets we train linear SVMs for sentiment classification on different domains (4 on the small Amazon and 7 on the large scale Amazon), The reported value is the averaged test error and its standard deviation across domains.

The experiments are run on a cluster of computers with a double quad-core Intel(R) Xeon(R) CPU E5345@2.33GHz with 8Gb of RAM.

Results. The kinds of embeddings learned on the Amazon data is shown in Figure 5, where the learned representations are non-linearly mapped to 2 dimensions by t-SNE (van der Maaten and Hinton, 2008).

Sampling has no effect on the quality of the representation learned. In Figure 2 (top-left) and 3 (left), we plot the test set accuracy for different levels of sampling. We observe the DAEs trained by reconstructing only 2% of the input units give results as good as the DAEs trained without sampling. However, we have

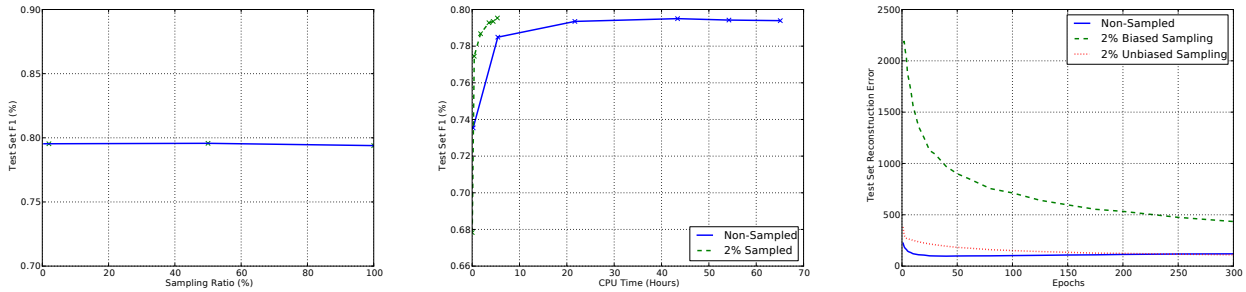

Figure 3. Experimental Results on RCV1. Increasing the sampling approximation does not hurt test F1, but yields a 12x speedup. The biased estimator gets a worse reconstruction error, but not the unbiased one.

found that the representation never reaches the same quality when the sampling probability doesn't depend on $\mathbf{x}$ (i.e. $P(\hat{\mathbf{p}}_k = 1) = d_{SMP}/d_x$).

One epoch is a training pass through the training set. Figure 2 (top-right) shows that the DAEs trained using sampling converge as fast in terms of epochs (i.e. in term of training updates) as the DAE trained without. The sampling DAE is trained to reconstruct only 2% of the input units. While initially the baseline DAE converges faster, after a few epochs both DAEs exhibit similar convergence. In order to assess the quality of the obtained results, we compared the averaged test error over the 4 domains with published results by Blitzer *et al.* (2007), we obtained an averaged test error of 13.7%, whereas they reported 16.7%.

Figure 2 (bottom-right) and 3 (middle) show that the training of DAEs using sampling is much faster. In particular, we compare the learning curve of a DAE that reconstructs only 2% of the input units and the baseline DAE. The sampling DAE converges 10.5x faster on the Amazon dataset and 12x faster on the RCV1 dataset.

Figure 2 (bottom-left) and 3 (right) show the effect of the term $\mathbf{q}$ on the convergence of the reconstruction cost. In the biased DAE we set $\mathbf{q}_k = 1$ and in the unbiased DAE we use $\mathbf{q}_k = E[\hat{\mathbf{p}}_k | k, \mathbf{x}, \tilde{\mathbf{x}}]$. This experiment shows that the DAE trained with the unbiased objective converges to the same reconstruction cost as the baseline while the unbiased version does not (since it minimizes a different cost). However, the networks using the biased and unbiased objectives converge similarly in terms of the quality of the representation.

Figure 4 shows the speed-up and training curves obtained on the full Amazon dataset, where the sampled reconstruction model converges 22 times faster, and reconstructing about 0.5% of the inputs. Keep in mind that the baseline dense training has been optimized already for speed (e.g., choosing the minibatch size and optimized code).

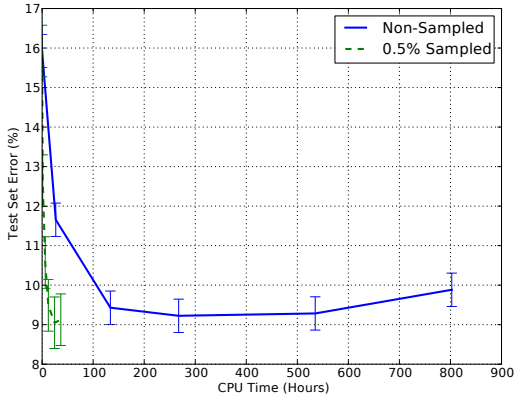

Figure 4. Experimental Results on Full Amazon set: test error vs CPU time. The speed-up is about 22x.

7. Conclusion

We have introduced a very simple optimization to speed-up training of unsupervised learning algorithms such as auto-encoders when the input vectors are very large and very sparse. The basic idea is to reconstruct only the non-zero's and a random subsample of the zero's of the input vector. A weighting scheme similar to importance sampling yields an unbiased estimator. On a dataset with a large input size we have found speed-up's of up to 22x, even comparing to optimized dense computation (using minibatches and BLAS' optimized matrix-matrix multiplications). We expect much larger speed-ups will be obtained in applications involving very large sparse input vectors, where the degree of sparsity is even larger than those tested here (2% and .5%).

References

Bengio, Y. (2008). Neural net language models. *Scholarpedia*, **3**(1), 3881.

Bengio, Y. (2009). Learning deep architectures for AI.

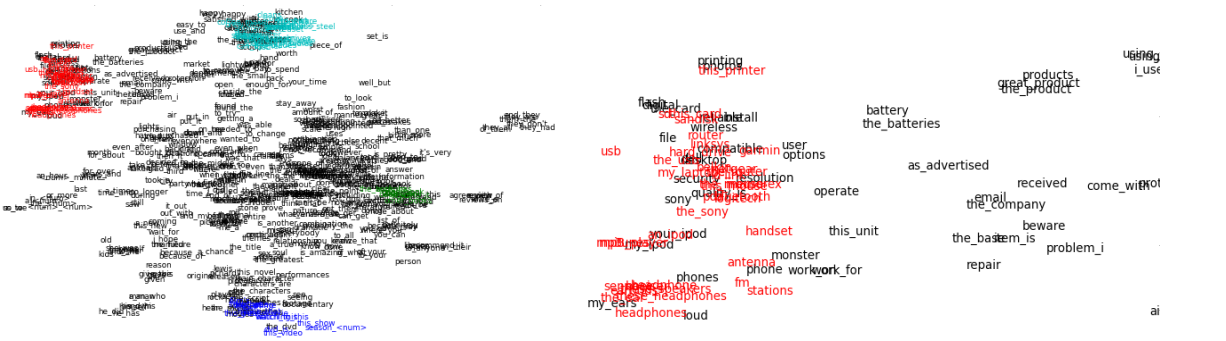

Figure 5. Embeddings learned on the Amazon sentiment data for a randomly selected set of word stems. Colors indicate the Amazon domain, showing that the embedding large scale (left) naturally discovers these categories. Right: zoom showing semantically similar words grouped near each other, on the topic of electronics.

Foundations and Trends in Machine Learning, **2**(1), 1–127. Also published as a book. Now Publishers, 2009.

Bengio, Y. and Sénécal, J.-S. (2003). Quick training of probabilistic neural nets by importance sampling. In *Proceedings of the conference on Artificial Intelligence and Statistics (AISTATS)*.

Bengio, Y. and Sénécal, J.-S. (2008). Adaptive importance sampling to accelerate training of a neural probabilistic language model. *IEEE Transactions on Neural Networks*, **19**(4), 713–722.

Blitzer, J., Dredze, M., and Pereira, F. (2007). Biographies, bollywood, boom-boxes and blenders: Domain adaptation for sentiment classification. In *Proceedings of the Association for Computational Linguistics (ACL'07)*, pages 440–447.

Collobert, R. and Weston, J. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning. In W. W. Cohen, A. McCallum, and S. T. Roweis, editors, *Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML'08)*, pages 160–167. ACM.

Erhan, D., Courville, A., Bengio, Y., and Vincent, P. (2010). Why does unsupervised pre-training help deep learning? In *JMLR W&CP: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2010)*, volume 9, pages 201–208.

Fan, R.-E., Chang, K.-W., Hsieh, C.-J., Wang, X.-R., and Lin, C.-J. (2008). LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, **9**, 1871–1874.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of The Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS'11)*.

Goto, K. and Geijn, R. A. v. d. (2008). Anatomy of high-performance matrix multiplication. *ACM Transactions Mathematical Software*, **34**, 12:1–12:25.

Hinton, G. E. and Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science*, **313**(5786), 504–507.

Hyvärinen, A. (2005). Estimation of non-normalized statistical models using score matching. *Journal of Machine Learning Research*, **6**, 695–709.

Japkowicz, N., Hanson, S. J., and Gluck, M. A. (2000). Nonlinear autoassociation is not equivalent to PCA. *Neural Computation*, **12**(3), 531–545.

Larochelle, H., Erhan, D., Courville, A., Bergstra, J., and Bengio, Y. (2007). An empirical evaluation of deep architectures on problems with many factors of variation. In Z. Ghahramani, editor, *Proceedings of the 24th International Conference on Machine Learning (ICML'07)*, pages 473–480. ACM.

Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. (1979). Basic linear algebra subprograms for fortran usage. *ACM Transactions Mathematical Software*, **5**, 308–323.

Lewis, D. D., Yang, Y., Rose, T. G., and Li, F. (2004). Rcv1: A new benchmark collection for text categorization research. *Journal of Machine Learning Research*, **5**, 361–397.

Mnih, A. and Hinton, G. E. (2009). A scalable hierarchical distributed language model. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems 21 (NIPS'08)*, pages 1081–1088.

Morin, F. and Bengio, Y. (2005). Hierarchical probabilistic neural network language model. In R. G. Cowell and Z. Ghahramani, editors, *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics (AISTATS'05)*, pages 246–252.

van der Maaten, L. and Hinton, G. E. (2008). Visualizing data using t-sne. *Journal of Machine Learning Research*, **9**, 2579–2605.

Vincent, P. (2010). A connection between score matching and denoising autoencoders. Technical Report 1358, Université de Montréal, DIRO.

Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P.-A. (2008). Extracting and composing robust features with denoising autoencoders. In W. W. Cohen, A. McCallum, and S. T. Roweis, editors, *Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML'08)*, pages 1096–1103. ACM.