Semi-Supervised Recursive Autoencoder

Si Chen and Yufei Wang
Department of Electrical and Computer Engineering
University of California San Diego
{sic046, yuw176}@ucsd.edu

Abstract

In this project, we implement the semi-supervised Recursive Autoencoders (RAE), and achieve the result comparable with result in [1] on the Movie Review Polarity dataset[1]. We achieve 76.08% accuracy, which is slightly lower than [1] 's result 76.8%, with less vector length. Experiments show that the model can learn sentiment and build reasonable structure from sentence.We find longer word vector and adjustment of words' meaning vector is beneficial, while normalization of transfer function brings some improvement. We also find normalization of the input word vector may be beneficial for training.

1 Introduction

Semi-supervised recursive autoencoders seek to use hierarchical structure to understand sentiment, which is different from other models such as bag-of-words representation. In this project, we implement the RAE model, and try to reproduce the result in [1] on Movie Review Polarity dataset. We try to investigate the role of different factors in [1]: length of word vector, normalization of transfer function and input word vector, and adjustment of word vector. We also want to investigate sentiment the model learns by observing interesting words and phrases.

The report is organized as follows. Section 2 gives a brief overview of RAE. Section 3 gives implementation details of the model, with detailed derivation of backpropogation results. Experiment design and results are shown in Section 4. Section 5 concludes our gains and findings.

2 Theoretical Overview

In this section, we briefly overview the semi-supervised recursive autoencoders model. This section references [1] and [2].

2.1 Traditional Recursive Autoencoders

The goal of traditional recursive autoencoders is to learn a reduced dimensional vector representation of sentences. Suppose the binary tree structure of a sentence is given, and the list of word vectors $x = (x_1, ..., x_n)$ is also known. Each word $x_i \in \mathbb{R}^d$ is a leaf node of the tree. The tree branches triplets of parents with children: $(p \to c_1 c_2)$. Each child can be either a word vector or a nonterminal node in the tree. Given this structure, we can compute the parent representation:

$$p = \frac{\mathrm{f}(W^{(1)}[c_1; c_2] + b^{(1)})}{\left\|\mathrm{f}(W^{(1)}[c_1; c_2] + b^{(1)})\right\|} \tag{1}$$

[1] https://www.cs.cornell.edu/people/pabo/movie-review-data/

where $[c_1; c_2]$ means c_1 concatenated vertically with c_2, and $W^{(1)} \in \mathbb{R}^{d \times 2d}$ and $b^{(1)} \in \mathbb{R}^d$ are parameters to be learned. f$(\cdot)$ is a pointwise activation function such as `tanh`. We normalize the parent vector to have unit length, to prevent zero hidden units.

To assess how well this $d-$dimensional vector represents its children, we try to reconstruct the children:

$$[y_1; y_2] = W^{(2)} p + b^{(2)} \tag{2}$$

where $W^{(2)} \in \mathbb{R}^{2d \times d}$ and $b^{(2)} \in \mathbb{R}^{2d}$ are parameters to be learned.

The goal is to minimize the reconstruction error for each input pair:

$$E_{rec}([c_1; c_2]) = \|c_1 - y_1\|^2 + \|c_2 - y_2\|^2 \tag{3}$$

To give more importance to reconstructing the meaning of longer phrases than shorter phrases or words, we can adjust the reconstructing error to be:

$$E_{rec}([c_1; c_2]) = \frac{n_1}{n_1 + n_2} \|c_1 - y_1\|^2 + \frac{n_2}{n_1 + n_2} \|c_2 - y_2\|^2 \tag{4}$$

where n_1, n_2 are the number of words underneath a current potential child.

The process repeats until the full tree is constructed and we have a reconstruction error at each nonterminal node.

2.2 Unsupervised Recursive Autoencoder

Now we assume that there is no tree structure given, and our goal is to minimize the reconstruction error of all vector pairs of children in a tree. Let $A(x)$ be the set of all possible trees for an input sentence x. $T(y)$ is the set of triplets of indexed by s of all the nonterminal nodes in a tree. Using the error of the whole tree, we compute

$$RAE_\theta(x) = \arg\min_{y \in A(x)} \sum_{s \in T(y)} E_{rec}([c_1; c_2]_s) \tag{5}$$

We use a greedy approximation that quickly constructs a tree that is good but not necessarily optimal.

We consider the $n - 1$ pairs of consecutive words and select the pair with smallest error, and its parent will replace both children in the sentence word list. Then the new sequence has $n-2$ possible pairs, select the pair with smallest error, and replace the pair with its parent. Continue until there is only one node.

2.3 Semi-Supervised Recursive Autoencoder

We extend RAEs to predict a target distribution for each sentence or phrase. In our application there are 2 possible labels: positive and negative. We can simply add on top of each parent node a `sigmoid` layer to predict label distributions

$$sm(p; \theta) = \text{sigmoid}(W_{cat} \cdot p + b_{cat}) \tag{6}$$

where $\theta = (W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}, W_{cat}, b_{cat})$ is the set of parameters, matrix $W_{cat} \in \mathbb{R}^{1 \times d}$, b_{cat} is a scalar. $sm(p; \theta)$ can xtbe viewed as predicted probability of the label 1(positive).

We use squared error of the predictions as objective function:

$$E_{sE}(p, l; \theta) = (l - sm)^2 \tag{7}$$

where l is the target label.

The objective function over (sentence, label) pairs (x, l) in a corpus is

$$J = \frac{1}{N} \sum_{(x,l)} E(x, l; \theta) + \frac{\lambda}{2} \|\theta\|^2 \tag{8}$$

where λ controls strength of regularization, and $E(x, l; \theta)$ is the sum of error at all the nodes

$$E(s, l; \theta) = \sum_{s \in T(RAE_\theta(x))} \alpha E_{rec}([c_1; c_2]_s; \theta) + (1 - \alpha) E_{sE}(p_s, l; \theta) \tag{9}$$

The hyperparameter α weighs reconstruction and squared error.

2.4 Backpropogation

In order to compute the gradient

$$\frac{\partial J}{\partial \theta} = \frac{1}{N} \sum_{(x,l)} \frac{\partial E(x,t;\theta)}{\partial \theta} + \lambda \theta \tag{10}$$

we use backpropogation. For every non-leaf node j, let a_j be the total activation coming it to node j, before nonlinearity. Define

$$\delta_j = \frac{\partial J}{\partial a_j} \tag{11}$$

Then the parameters from other nodes to an output node j is easy to compute from δ_j using the chain rule. For example, w_{ij}, the weight from node i to node j can be calculated

$$\frac{\partial J}{\partial w_{ij}} = \delta_j \frac{\partial a_j}{\partial w_{ij}} = \delta_j c_j \tag{12}$$

where c_j is the scalar computed at node j.

For a node j that is not an output node, that feeds into nodes indexed by k, we can also write the delta function

$$\delta_j = \frac{\partial J}{\partial a_j} = \sum_k \frac{\partial J}{\partial a_k} \frac{\partial a_k}{\partial a_j} = \sum_k \delta_k \frac{\partial a_k}{\partial a_j} \tag{13}$$

The needed partial derivative is

$$\frac{\partial a_k}{\partial a_j} = w_{jk} \mathrm{f}'(a_j) \tag{14}$$

This way, we can compute delta function in each non-leaf node and compute the partial derivative with respect to each parameter.

3 Implementation Details

In this section, we are going to describe in detail how to implement the semi-supervised recursive autoencoder. The details are based on the code released by [1]. In order to make it more clear, we use $[W1, W2]$ and $[W3, W4]$ to represent the encoding and decoding weights respectively instead of $W^{(1)}$ and $W^{(2)}$. We use $b1$ instead of $b^{(1)}$ for the encoding bias and $(b2, b3)$ instead of $b^{(2)}$ to represent the decoding bias for the left and right child respectively. The representations are clearly shown in Fig. 1. The algorithm is implemented in Python.

3.1 Random Word Initialization

We use random initialization for word encoding. Suppose that we have totally N different words in both the test and training set and each word is encoded by a vector $x_i \in R^d$, then the whole code book W_e can be represented using a $d \times N$ matrix. The elements in W_e are randomly sampled from a uniform distribution in [-0.05,0.05] . While training, we can tune the word vectors by back propagation.

3.2 Greedy Tree Construction

Firstly, we randomly initialize $W_1, W_2, W_3, W_4, W_{cat}$ by sampling from a uniform distribution in $[r, -r]$, where $r = 10^{-3}\sqrt{\frac{6}{2\times d+1}}$. The biases b_1, b_2, b_3 are all set to zero vectors. We can build a sub-optimal tree for each sentence based on $(W_1, W_2, W_3, W_4, b_1, b_2, b_3, W_e)$ using the greedy algorithm described in Section 2.2.

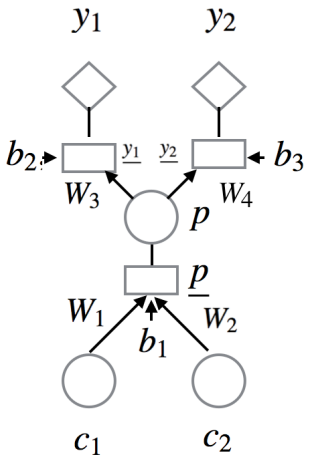

Figure 1: This is a classical autoencoder. The underlined symbols represent the corresponding values before the nonlinearity. For example, $p = \text{func}(\underline{p})$, where `func` is the nonlinear function. Also $\underline{p} = W_1 \cdot c_1 + W_2 \cdot c_2 + b_1$, $\underline{y_1} = W_3 \cdot p + b_2$, $\underline{y_2} = W_4 \cdot p + b_3$

3.3 Backpropagation

The goal of the training procedure is to find the best set of parameters that minimize a certain cost function. As a result, we should first make clear the cost function. There are two kinds of cost functions in our implementation. For reconstruction cost, we follow the implementation in [1] to use non-weighted error in Equation 3. The cost functions are:

$$\begin{aligned}\mathbf{cost_{RAE}} =& \frac{1}{N_t} \times 0.5 \times \alpha \times \sum_{i \in allNodes} (\left\|c_1^{(i)} - y_1^{(i)}\right\|_2^2 + \left\|c_2^{(i)} - y_2^{(i)}\right\|_2^2) \\ &+ \frac{\lambda_w}{2}(\|W_1\|_2^2 + \|W_2\|_2^2 + \|W_3\|_2^2 + \|W_4\|_2^2) + \frac{\lambda_L}{2}\|W_e\|_2^2 ,\end{aligned} \tag{15}$$

$$\begin{aligned}\mathbf{cost_{SUP}} =& \frac{1}{N_s} \times 0.5 \times (1-\alpha) \times (\sum_{i \in leaf} (l^{(i)} - sm^{(i)})^2 + \beta \sum_{i \in nonleaf} (l^{(i)} - sm^{(i)})^2) \\ &+ \frac{\lambda_w}{2}(\|W_1\|_2^2 + \|W_2\|_2^2 + \|W_3\|_2^2 + \|W_4\|_2^2) + \frac{\lambda_L}{2}\|W_e\|_2^2 + \frac{\lambda_C}{2}\|W_{cat}\|_2^2 ,\end{aligned} \tag{16}$$

where N_s is the number of training sentences, $\|\cdot\|_2$ is the second order norm. $l^{(i)}$ is the true label of the i-th node. Here we use square loss for both cost functions. $\mathbf{cost_{RAE}}$ is the cost of recursive autoencoder and $\mathbf{cost_{SUP}}$ is the cost of supervised learning. β is used to change the relative importance of the leaf and non leaf nodes. Suppose N_l is the number of leaf nodes, then $N_t = N_l - N_s$. N_t and N_s are used to eliminate the influence of the number of nodes and sentences to the cost functions. In our experiments, α and β are set to 0.2 and 0.5 respectively. Since

$$\mathbf{cost_{total}} = \mathbf{cost_{RAE}} + \mathbf{cost_{SUP}},$$

we can calculate the gradient of the two kinds of cost functions separately. Also, we won't discuss the gradient of the regularization term, which is very simple.There are two main differences between recursive neural network and traditional neural network. One is that the former use shared weights and the other is that the recursive neural network has loss functions for every node while the traditional one only has loss functions on the root node. The second difference makes it difficult to calculate the delta functions δ_j.

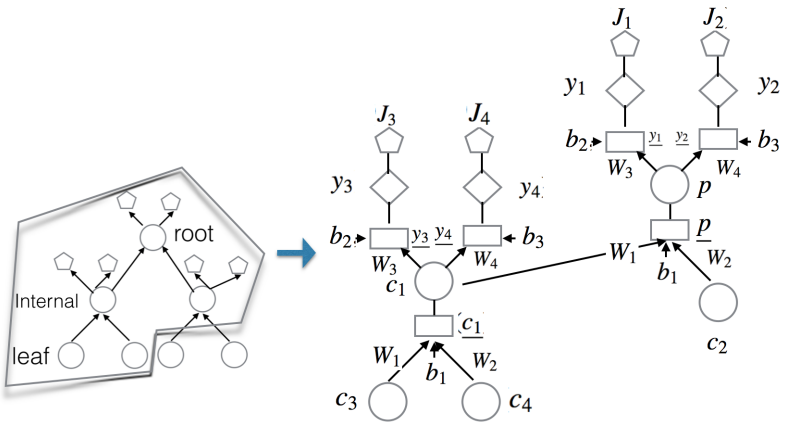

Figure 2: The computation graph for $\mathbf{cost_{RAE}}$. Left: the tree constructed from a sentence. Right: Corresponding graph with detailed calculating information. p is the root node. c_1 and c_2 are internal nodes. c_3 and c_4 are leaf nodes.

3.3.1 Backpropagation for $\mathbf{cost_{RAE}}$

Regardless of the regularization terms, the cost function is:

$$\begin{aligned}\mathbf{cost^{(1)}_{RAE}} &= \frac{1}{N_t} \times 0.5 \times \alpha \times \sum_{i \in allNodes} (\left\|c_1^{(i)} - y_1^{(i)}\right\|_2^2 + \left\|c_2^{(i)} - y_2^{(i)}\right\|_2^2) \\ &= \frac{1}{N_t} \times 0.5 \times \alpha \times \sum_{s \in N_s} \sum_{i \in allNodes^{(s)}} (\left\|c_1^{(i)} - y_1^{(i)}\right\|_2^2 + \left\|c_2^{(i)} - y_2^{(i)}\right\|_2^2),\end{aligned} \tag{17}$$

where $allNodes^{(s)}$ is the set of all nodes of the tree constructed from sentence s. So the problem is equivalent to calculating the gradient of $\sum_{i \in allNodes^{(s)}} (\left\|c_1^{(i)} - y_1^{(i)}\right\|_2^2 + \left\|c_2^{(i)} - y_2^{(i)}\right\|_2^2)$ w.r.t. $(W_1, W_2, W_3, W_4, b_1, b_2, b_3, W_e)$ for the tree constructed by each sentence and then add them together. Below, we will first discuss how to use backpropagation to calculate the delta functions of three different kinds of nodes given a tree(see Fig. 2).

- Root Node
  For the root node, there are only 2 loss functions related, which is the reconstruction error of the left(J_1) and right child(J_2). We define $J = J_1 + J_2$. By chain rule, we have:

$$\frac{\partial J_1}{\partial \underline{y_1}} = \mathrm{f}'(\underline{y_1})(-2(c_1 - y_1)), \tag{18}$$

$$\frac{\partial J_2}{\partial \underline{y_2}} = \mathrm{f}'(\underline{y_2})(-2(c_2 - y_2)), \tag{19}$$

$$\frac{\partial J}{\partial \underline{p}} = \mathrm{f}'(\underline{p})(W_3^T \frac{\partial J_1}{\partial \underline{y_1}} + W_4^T \frac{\partial J_2}{\partial \underline{y_2}}), \tag{20}$$

  where $\mathrm{f}'(\cdot)$ is the gradient of the nonlinear function. Since both input and output of $\mathrm{f}(\cdot)$ are d-dimension vectors, the output of $\mathrm{f}'(\cdot)$ is a $d \times d$ matrix. The three equations above show how to calculate the delta functions w.r.t. $\underline{y_1}, \underline{y_2}, \underline{p}$.

- Internal Node
  For the internal node, the gradients are more complicated. All the loss functions from that node to the root node are related. We take the left child of the root node for an example. $J_{left} = J + J_3 + J_4$. By chain rule, we have:

$$\frac{\partial J_3}{\partial \underline{y_3}} = \mathrm{f}'(\underline{y_3})(-2(c_3 - y_3)), \tag{21}$$

$$\frac{\partial J_4}{\partial \underline{y_4}} = \mathrm{f}'(\underline{y_4})(-2(c_4 - y_4)), \tag{22}$$

$$\frac{\partial J_1}{\partial c_1} = 2(c_1 - y_1), \tag{23}$$

$$\frac{\partial J_{left}}{\partial \underline{c_1}} = \mathrm{f}'(\underline{c_1})(W_3^T \frac{\partial J_3}{\partial \underline{y_3}} + W_4^T \frac{\partial J_4}{\partial \underline{y_4}} + W_1^T \frac{\partial J}{\partial \underline{p}} + \frac{\partial J_1}{\partial c_1}), \tag{24}$$

The four equations above can be used to calculate the delta function of any internal nodes. Obviously, $\frac{\partial J_3}{\partial \underline{y_3}}$ and $\frac{\partial J_4}{\partial \underline{y_4}}$ are calculated similarly to $\frac{\partial J_1}{\partial \underline{y_1}}$.

- Leaf Node
  Leaf nodes are special, for they don't have reconstruction loss. For c_3 in Fig. 2, $J_{leaf} = J_{left}$. By chain rule, we have:

$$\frac{\partial J_{leaf}}{\partial c_3} = W_1^T \frac{\partial J_{left}}{\partial \underline{c_1}} + \frac{\partial J_3}{\partial c_3}, \tag{25}$$

  where

$$\frac{\partial J_3}{\partial c_3} = 2(c_3 - y_3). \tag{26}$$

After we get all the delta functions, we can easily calculate $\mathbf{cost_{RAE}}$ w.r.t. the parameters for each node. Take the root node of Fig. 2 for an example:

$$\frac{\partial J}{\partial b_1} = \frac{\partial J}{\partial \underline{p}} \tag{27}$$

$$\frac{\partial J}{\partial b_2} = \frac{\partial J_1}{\partial \underline{y_1}} \tag{28}$$

$$\frac{\partial J}{\partial b_3} = \frac{\partial J_2}{\partial \underline{y_2}} \tag{29}$$

$$\frac{\partial J}{\partial W_1} = \frac{\partial J}{\partial \underline{p}} c_1^T \tag{30}$$

$$\frac{\partial J}{\partial W_2} = \frac{\partial J}{\partial \underline{p}} c_2^T \tag{31}$$

$$\frac{\partial J}{\partial W_3} = \frac{\partial J_1}{\partial \underline{y_1}} p^T \tag{32}$$

$$\frac{\partial J}{\partial W_4} = \frac{\partial J_2}{\partial \underline{y_2}} p^T \tag{33}$$

As is mentioned before, the recursive neural networks use shared weights, i.e. all the $W_1, W_2, W_3, W_4, b_1, b_2, b_3$ are shared by all the nodes. We could simply add all the gradients corresponding to every node together. For the gradients w.r.t. W_e, they are just the same as the delta functions for the leaf nodes.

3.3.2 Backpropagation for $\mathbf{cost_{SUP}}$

Regardless of the regularization terms, the cost function is:

$$\begin{aligned} \mathbf{cost_{SUP}} =& \frac{1}{N_s} \times 0.5 \times (1-\alpha) \times (\sum_{i \in leaf} (l^{(i)} - sm^{(i)})^2 + \beta \sum_{i \in nonleaf} (l^{(i)} - sm^{(i)})^2) \\ =& \frac{1}{N_s} \times 0.5 \times (1-\alpha) \times \sum_{s \in N_s} (\sum_{i \in leaf^{(s)}} (l^{(i)} - sm^{(i)})^2 + \beta \sum_{i \in nonleaf^{(s)}} (l^{(i)} - sm^{(i)})^2) \end{aligned} \tag{34}$$

Similar to Section 3.3.1, we can calculate the gradient of $\mathbf{cost_{SUP}}$ for each tree first (i.e. $\sum_{i \in leaf^{(s)}} (l^{(i)} - sm^{(i)})^2 + \beta \sum_{i \in nonleaf^{(s)}} (l^{(i)} - sm^{(i)})^2$) and the add them up. We use the same tree structure to illustrate how to calculate the delta functions.

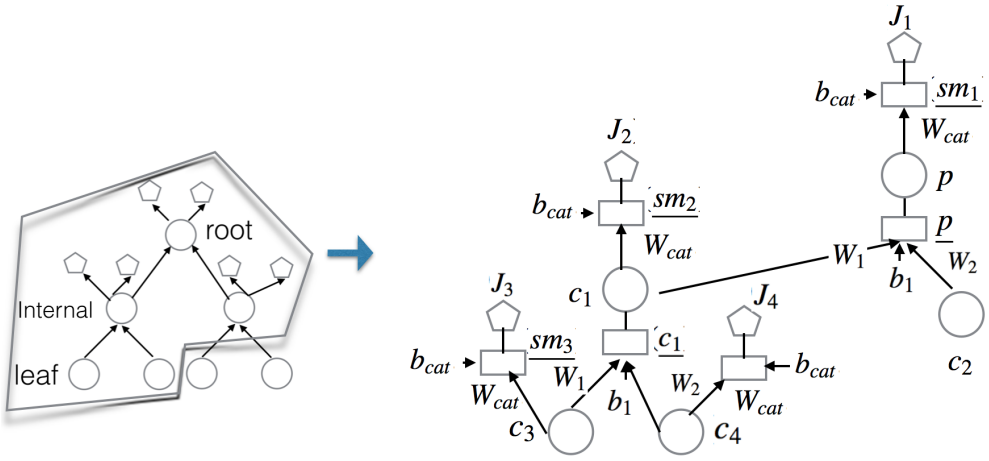

Figure 3: The computation graph for $\mathbf{cost_{SUP}}$. Left: the tree constructed from a sentence. Right: Corresponding graph with detailed calculating information. p is the root node. c_1 and c_2 are internal nodes. c_3 and c_4 are leaf nodes.

- Root Node
  For the root node, there are only 1 loss function related, which is the classification error. By chain rule, we have:

$$\frac{\partial J_1}{\partial \underline{sm_1}} = \mathrm{g}'(\underline{sm_1})(2\beta(l_1 - sm_1)), \tag{35}$$

$$\frac{\partial J_1}{\partial \underline{p}} = \mathrm{f}'(\underline{p})(W_{cat}^T \frac{\partial J_1}{\partial \underline{sm_1}}), \tag{36}$$

  where $\mathrm{f}'(\cdot)$ is the gradient of the nonlinear function and $\mathrm{g}'(\cdot)$ is the gradient of sigmoid function. Note that $\frac{\partial J_1}{\partial \underline{sm_1}}$ and $\mathrm{g}'(\cdot)$ are scalars.

- Internal Node
  We take the left child of the root node for an example. $J_{left} = J_1 + J_2$. By chain rule, we have:

$$\frac{\partial J_2}{\partial \underline{sm_2}} = \mathrm{g}'(\underline{sm_2})(2\beta(l_2 - sm_2)), \tag{37}$$

$$\frac{\partial J_{left}}{\partial \underline{c_1}} = \mathrm{f}'(\underline{c_1})(W_1^T \cdot \frac{\partial J_1}{\partial \underline{p}} + W_{cat}^T \frac{\partial J_2}{\partial \underline{sm_2}}), \tag{38}$$

  The two equations above can be used to calculate the delta function of any internal nodes.

- Leaf Node
  For $\mathbf{cost_{SUP}}$, leaf nodes also have their own classification loss. For c_3 in Fig. 3, $J_{leaf} = J_{left} + J_3$. By chain rule, we have:

$$\frac{\partial J_3}{\partial \underline{sm_3}} = \mathrm{g}'(\underline{sm_3})(2(l_3 - sm_3)), \tag{39}$$

$$\frac{\partial J_{leaf}}{\partial c_3} = W_1^T \cdot \frac{\partial J_{left}}{\partial \underline{c_1}} + W_{cat}^T \frac{\partial J_3}{\partial \underline{sm_3}}. \tag{40}$$

The parameters that are related to $\mathbf{cost_{SUP}}$ are $W_1, W_2, W_{cat}, b_1, b_{cat}$. Also take the root node for example:

$$\frac{\partial J_1}{\partial b_1} = \frac{\partial J_1}{\partial \underline{p}} \tag{41}$$

$$\frac{\partial J}{\partial W_1} = \frac{\partial J_1}{\partial \underline{p}} c_1^T \tag{42}$$

$$\frac{\partial J}{\partial W_2} = \frac{\partial J_1}{\partial \underline{p}} c_2^T \tag{43}$$

$$\frac{\partial J}{\partial W_{cat}} = \frac{\partial J_1}{\partial \underline{sm_1}} p^T \tag{44}$$

$$\frac{\partial J}{\partial b_{cat}} = \frac{\partial J_1}{\partial \underline{sm_1}} \tag{45}$$

For W_e, the gradients are just the same as the delta functions for the leaf nodes.

3.3.3 Calculation of nonlinear function $\mathrm{f}'(\cdot)$

Now let's look at the calculation of $\mathrm{f}'(\cdot)$. In our implementation, we use `tanh` as nonlinear function. There are two options for nonlinear function.

- First, we can use non-normalized `tanh`. This way, the transfer function is a pointwise $\tanh(a)$. For simplicity, we use f_i for short of $\mathrm{f}(a_i)$.

$$\mathrm{f}'(a) = \begin{pmatrix} \frac{\partial \mathrm{f}_1}{\partial a_1} & \cdots & \frac{\partial \mathrm{f}_d}{\partial a_1} \\ & \cdots & \\ \frac{\partial \mathrm{f}_1}{\partial a_d} & \cdots & \frac{\partial \mathrm{f}_d}{\partial a_d} \end{pmatrix} = \begin{pmatrix} 1-\tanh^2(a_1) & 0 & \cdots & 0 \\ 0 & 1-\tanh^2(a_2) & \cdots & 0 \\ & & \ddots & \\ 0 & \cdots & 0 & 1-\tanh^2(a_d) \end{pmatrix} \tag{46}$$

  where $\mathrm{f} = (\mathrm{f}_1, \mathrm{f}_2, ..., \mathrm{f}_d)^T$, $a = (a_1, a_2, ..., a_d)^T$. Note that $\mathrm{f}'(a)$ is the transpose of Jacobian matrix.

- The second option is to use the normalized transfer function $\tanh(a)/\left\|\tanh(a)\right\|$.
  $\mathrm{f}'(a)$ for normalized function is more complicated. Let $\mathrm{g}(a) = \tanh(a)$. Then, we have

$$\mathrm{f}(a) = \frac{\mathrm{g}(a)}{\sqrt{\sum_{i=1}^{d} \mathrm{g}(a_i)^2}} \tag{47}$$

  We first work out the Jacobian matrix:

$$\begin{aligned} \frac{\partial(\mathrm{f}_1, \mathrm{f}_2, ..., \mathrm{f}_d)}{\partial(a_1, a_2, ..., a_d)} &= \frac{\partial(\mathrm{f}_1, \mathrm{f}_2, ..., \mathrm{f}_d)}{\partial(\mathrm{g}_1, \mathrm{g}_2, ..., \mathrm{g}_d)} \cdot \frac{\partial(\mathrm{g}_1, \mathrm{g}_2, ..., \mathrm{g}_d)}{\partial(a_1, a_2, ..., a_d)} \\ &= \begin{pmatrix} \frac{\left\|\mathrm{g}\right\|_2 - \frac{\mathrm{g}_1^2}{\left\|\mathrm{g}\right\|_2}}{\left\|\mathrm{g}\right\|_2^2} & -\frac{\mathrm{g}_1\mathrm{g}_2}{\left\|\mathrm{g}\right\|_2^3} & \cdots & -\frac{\mathrm{g}_1\mathrm{g}_d}{\left\|\mathrm{g}\right\|_2^3} \\ -\frac{\mathrm{g}_2\mathrm{g}_1}{\left\|\mathrm{g}\right\|_2^3} & \frac{\left\|\mathrm{g}\right\|_2 - \frac{\mathrm{g}_2^2}{\left\|\mathrm{g}\right\|_2}}{\left\|\mathrm{g}\right\|_2^2} & & -\frac{\mathrm{g}_2\mathrm{g}_d}{\left\|\mathrm{g}\right\|_2^3} \\ \vdots & & \ddots & \\ -\frac{\mathrm{g}_d\mathrm{g}_1}{\left\|\mathrm{g}\right\|_2^3} & -\frac{\mathrm{g}_d\mathrm{g}_2}{\left\|\mathrm{g}\right\|_2^3} & \cdots & \frac{\left\|\mathrm{g}\right\|_2 - \frac{\mathrm{g}_d^2}{\left\|\mathrm{g}\right\|_2}}{\left\|\mathrm{g}\right\|_2^2} \end{pmatrix} \cdot \begin{pmatrix} 1-\mathrm{g}_1^2 & 0 & \cdots & 0 \\ 0 & 1-\mathrm{g}_2^2 & \cdots & 0 \\ & & \ddots & \\ 0 & \cdots & 0 & 1-\mathrm{g}_d^2 \end{pmatrix} \\ &= (\frac{1}{\left\|\mathrm{g}\right\|_2}\mathrm{I} - \frac{1}{\left\|\mathrm{g}\right\|_2^3}\mathrm{g}\cdot\mathrm{g}^T) \cdot \begin{pmatrix} 1-\mathrm{g}_1^2 & 0 & \cdots & 0 \\ 0 & 1-\mathrm{g}_2^2 & \cdots & 0 \\ & & \ddots & \\ 0 & \cdots & 0 & 1-\mathrm{g}_d^2 \end{pmatrix} \end{aligned} \tag{48}$$

  where $\mathrm{g} = (\mathrm{g}_1, \mathrm{g}_2, ..., \mathrm{g}_d)^T$.

Then $\mathrm{f}'(a)$ is the transpose of the Jacobian:

$$\mathrm{f}'(a) = \begin{pmatrix} 1-\mathrm{g}_1^2 & 0 & ... & 0 \\ 0 & 1-\mathrm{g}_2^2 & ... & 0 \\ & & \ddots & \\ 0 & ... & 0 & 1-\mathrm{g}_d^2 \end{pmatrix} \cdot \left(\frac{1}{\|\mathrm{g}\|_2}\mathrm{I} - \frac{1}{\|\mathrm{g}\|_2^3}\mathrm{g}\cdot\mathrm{g}^T\right) \tag{49}$$

3.4 Optimization

In the last section, we describe in detail how to use backpropagation to calculate the gradients of $\mathbf{cost_{total}}$ for all parameters $(W_1, W_2, W_3, W_4, W_{cat}, b_1, b_2, b_3, b_{cat}, W_e)$. We use L-BFGS algorithm to optimize $\mathbf{cost_{total}}$ using the gradients, since it requires less hyper-parameters than gradient descent.

3.5 Test the model

After we train the model, for each training and test sentence, we build a tree using the same greedy algorithm and store the encoding for each node. The feature for each sentence is a $2 \times d$ vector, the first d entries of which are the encoding of the root node. The last d entries of the feature are the average of the encodings of all nodes for that sentence. We then train a linear classifier (logistic regression) using the training features and labels. The test features are classified using the same classifier. The ratio of the size of the training set and test set are 10:1. We use the accuracy rate of the test set as the criteria.

4 Experiment

In the experiment, we want to duplicate [1] 's result, observe some interesting features of the method, and study several different variations of the method. Hyperparameters are set to $(\alpha = 0.2, \beta = 0.5)$. The maximum number of epochs is set to be 70. Our python code is released online. [2]

4.1 Verifying the derivatives

To verify that derivatives we compute are correct, we compare them with numerical derivatives:

$$\frac{\partial J}{\partial w_{ij}} = \frac{J(w_{ij}+\epsilon) - J(w_{ij}-\epsilon)}{2\epsilon} + O(\epsilon^2) \tag{50}$$

To compute the numerical derivatives quickly, we decrease the vector length to 2, and use only 2 sentences for verification. The difference of the two derivatives is computed as the L2-norm:

$$\Delta d = \left\| \frac{\partial J}{\partial \theta}_{\text{(numerical)}} - \frac{\partial J}{\partial \theta}_{\text{(backprob)}} \right\|_2 \tag{51}$$

where θ is the vector of all parameters.

The difference as a function of ϵ is shown in Fig. 4. As shown, the differences tend to zero as ϵ tends to zero. When $\epsilon < 10^{-4}$, the differences drop v.s. ϵ is nearly a line in the log-log scale. From the plot we can observe that difference tends to zero at the rate of ϵ^2. This can be proved by Equation 50, the second term $O(\epsilon^2)$ is the difference we calculate.

4.2 Observing the learned semantics

We show the ten most positive and negative words learned by RAE. As shown in Table 1, most words are reasonable and show strong positive/negative meaning. However, there exists some mistakenly picked neutral words, such as "cinema", or "feels". This is probably because the linear classifier we use isn't powerful enough.

Table 2 shows ten most positive/negative sentences predicted in test set. Most of the sentence is with strong positive/negative sentiment. For example, "amazingly lame" is successfully predicted

[2] http://acsweb.ucsd.edu/ sic046/

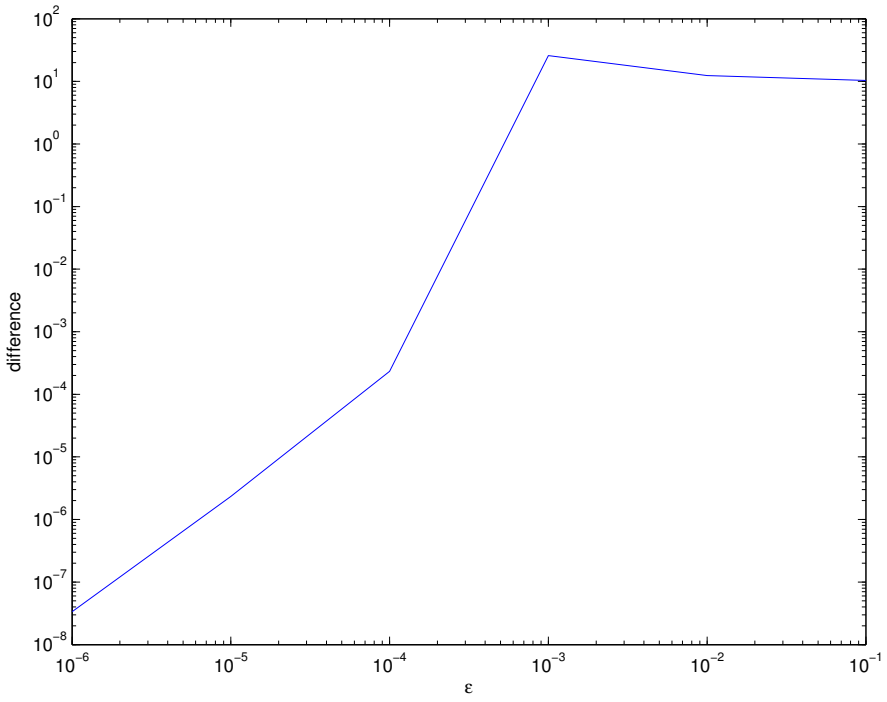

Figure 4: Difference of two derivatives as a function of ϵ.

Table 1: Ten most positive/negative words

Ten most positive words	**Ten most negative words**
best	too
performances	bad
cinema	dull
heart	?
still	no
entertaining	worst
human	feels
our	was
moving	boring
enjoyable	lack

as negative. There are mistakenly predicted sentences, though. For example "... not a bad way to spend an hour or two" should be a neutral to a little negative sentence, but is predicted as strongly negative sentence.

We pick a sentence and show the tree structure built by greedy algorithm in Fig. 5. The sentence is "a truly wonderful tale combined with stunning animation.". The tree construction is basically right, but it mistakenly binds "tale" with "combined".

4.3 Factors that influence the performance

We explore several factors that may influence the system's performance, and do experiments to observe their effects over performance. Results are shown in Table 3.

Length of vector With normalization on the transfer function, we change the length of vector to observe whether decrease of length impairs the performance greatly. The first three lines in Table 3

Table 2: Ten most positive/negative sentences

Most positive sentences on test set
a poignant and compelling story about relationships , food of love takes us on a bumpy but satisfying journey of the heart .
twist open the ouzo ! it 's time to let your hair down greek style . a vibrant whirlwind of love , family and all that goes with it , my big fat greek wedding is a non-stop funny feast of warmth , color and cringe .
a simple tale of an unlikely friendship , but thanks to the gorgeous locales and exceptional lead performances , it has considerable charm.
even if you do n't know the band or the album 's songs by heart , you will enjoy seeing how both evolve , and you will also learn a good deal about the state of the music business in the 21st century.
the film is filled with humorous observations about the general absurdity of modern life as seen through the eyes outsiders , but deftly manages to avoid many of the condescending stereotypes that so often plague films dealing with the mentally ill .
still rapturous after all these years , cinema *UNKNOWN* stands as one of the great films about movie love .
all right , so it 's not a brilliant piece of filmmaking , but it is a funny *UNKNOWN* sometimes hilarious *UNKNOWN* comedy with a deft sense of humor about itself , a playful spirit and a game cast
... is funny in the way that makes you ache with sadness *UNKNOWN* the way UNKNOWN* is funny *UNKNOWN* , profound without ever being *UNKNOWN* , warm without ever succumbing to sentimentality .
though few will argue that it ranks with the best of *UNKNOWN* 's works , invincible shows he 's back in form , with an astoundingly rich film .
maelstrom is strange and compelling , engrossing and different , a moral tale with a twisted sense of humor .
Most negative sentences on test set
" sorority boys " was *UNKNOWN* , and that movie was pretty bad .
amazingly lame .
it 's like every bad idea that 's ever gone into an after-school special compiled in one place , minus those daytime programs ' *UNKNOWN* and sophistication *UNKNOWN* and who knew they even had any ? *UNKNOWN* .
it 's not that waiting for happiness is a bad film , because it is n't . it 's just incredibly dull .
it 's not too fast and not too slow . it 's not too racy and it 's not too offensive . it 's not too much of anything .
poor editing , bad *UNKNOWN* , and *UNKNOWN* dialogue highlight the radical action .
at times a bit melodramatic and even a little dated *UNKNOWN* depending upon where you live *UNKNOWN* , ignorant *UNKNOWN* is still quite good-natured and not a bad way to spend an hour or two .
the movie makes absolutely no sense . its underlying mythology is a hodgepodge of inconsistencies that pose the question : since when did dumb entertainment have to be this dumb ?
would *UNKNOWN* 's italian *UNKNOWN* have been any easier to sit through than this hastily dubbed disaster ?
the following things are not at all entertaining : the bad sound , the lack of climax and , worst of all , watching *UNKNOWN* *UNKNOWN* who is also one of the film 's producers *UNKNOWN* do everything he can to look like a good guy .

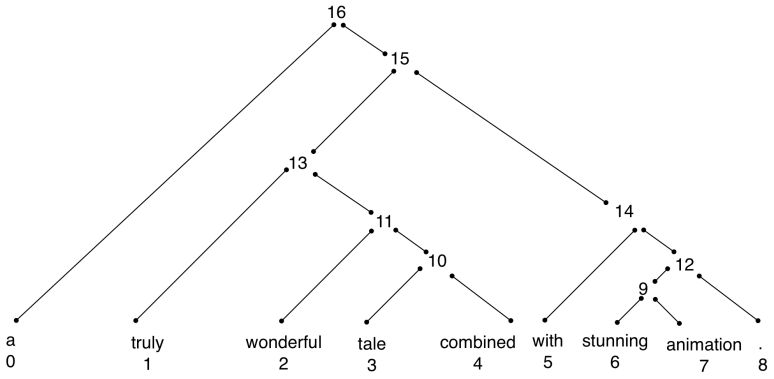

Figure 5: Tree structure of sentence "a truly wonderful tale combined with stunning animation.".

Table 3: Accuracy on training and test set with different methods and different word vector length. Full method means method with normalization of transfer function and adjustment of meaning vector.

Method	**Training accuracy(%)**	**Test accuracy(%)**
Full method (d=20)	63.38	63.22
Full method (d=40)	81.91	75.05
Full method (d=50)	84.09	76.08
Without normalization of transfer function(d=50)	82.60	74.30
Without adjustment of meaning vector of words(d=50)	58.17	58.91
With normalization of word vector (d=20)	86.04	74.77
With normalization of word vector (d=50)	95.71	74.95

show the result. Accuracy on training set and test set increases with length of word vector, and the improvement gets slower when length is larger. With length 50, there is only 1% increase compared to length 40 on the test set. Also, with the increase of vector length, overfitting becomes obvious. With length 20, there is no noticeable overfitting; however, when length is 40, overfitting occurs, and becomes more serious when length increases to 50.

Normalization on transfer function Fixing word vector length at 50, we compare the result with and without normalizing transfer functions. As shown in third and forth rows of Table 3, there is around 1.5% increase in accuracy on both training and test set when using normalization. This implies normalization on transfer function brings some benefits. The reason why normalization can bring benefits is that normalization of the transfer function force the value of each non-leaf node less than 1. This prevents the nonlinear function from saturating. When nonlinear function saturates, the gradient is very small, thus the converging rate will be slow. Therefore, with normalization of the transfer function, saturation is prevented, thus bringing improvement to performance.

Adjustment of word's meaning vector Fixing word vector length at 50 and with normalizing the transfer function, we compare the method with the one not adjusting meaning vector for each word. As shown in fifth row of Table 3, without adjustment of the meaning vector of words, the accuracy drops greatly, and there is no overfitting. This is because without adjustment for W_e, there is a great drop in number of parameters, thus the capability of the model decreases largely. Therefore, adjusting the meaning vector for each word is necessary.

Normalization of word vector As mentioned in Section 3.1, we randomly initialize the word vector elements in range [-0.05,0.05], which means each word vector has norm much less than 1. With normalization of the transfer function, each transfer function has norm 1. This inevitably causes a large reconstruction error at the early stage of training, and may cause the result fall into a local minimum. To avoid this and force training process to start from a good point, we first normalize each word vector to have norm 1. From Table 3, we show that when vector length is 20, there is a big

improvement after normalizing the word vector in both training and test set. When vector length is 50, however, training set can achieve the high accuracy of more than 95%, while accuracy of test set decreases. The results for both length 20 and 50 suggest large overfitting. This is probably because the normalization of word vector makes the algorithm find a better minimum on training set, with the cost of large overfitting. We also observe that with normalization of word vector, the algorithm needs much more epochs to converge: For length 20, the algorithm doesn't converge after 143 epochs; and for length 50, the algorithm doesn't converge within 300 epochs(Without normalized word vector, algorithm converges within 70 epochs). Overfitting can be reduced by applying a validation set, and we believe by doing so, the performance of RAE should be better. However, due to limited time, we haven't done this experiment yet.

5 Conclusion

In this project, we implement the semi-supervised RAE with random word initialization. We test our result on Movie Reviews Polarity dataset, and our model achieves 76.08% accuracy on test set. The observation with representative words and sentences shows the model learns sentiment to a certain degree, and can build reasonable sentence structure. We also found: 1)longer word vector will achieve better result, 2) normalization of transfer function has some benefits, 3) adjustment of words' meaning vector is necessary, and 4) normalization of the input word vector may be beneficial.

References

[1] Socher, R., Pennington, J., Huang, E.H., Ng, A.Y., Manning, C.D.: Semi-supervised recursive autoencoders for predicting sentiment distributions. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing. EMNLP '11, Stroudsburg, PA, USA, Association for Computational Linguistics (2011) 151–161

[2] Elkan, C.: Learning meanings for sentences. (February 25, 2014)