Abstract

In this project, we implemented the recursive autoencoder (RAE) as described in Socher’s paper to discover the sentiment of sentences. We train and test our RAEs with a dataset of over 10000 sentences from movie reviews, and achieve 75.4% accuracy.

1 Introduction

The paper of Socher2011 describes a framework that uses recursive autoencoders to analyse the meanings of texts. They represent each word in the vocabulary with a fixed-length vector that encodes the meaning of that word. By constructing a recursive autoencoder, the meaning vectors of words are propagated forward to a top node that encapsulates the meaning of the entire text segment. Many applications can be built using these meaning vectors, one of which is to detect the sentiment underlying a piece of text. In this project we implement Socher’s approach to analyse the sentiment of sentences from movie reviews.

The data is a set of 10662 sentences from the website Rotten Tomatos, each of 5-50 words long and labelled with an either positive or negative sentiment. We aim to learn an RAE that predicts the sentiment for these sentences.

2 Backpropagation in RAE

Recursive autoencoder is an instance of a feedforward neural network. We aim to learn the weights on the edges that minimizes some measure of error for examples in the training set. The error gradients with respect to these parameters can be computed via backpropagation.

2.1 Structure of RAE

In our RAE, a phrase node is a node that is fed from two other phrase/word nodes. Each phrase node has two reconstruction nodes that tries to recover the meaning vectors of that phrase node’s two children. Each phrase node, as well as each word node, has a label node, that contains the predicted label for the meaning vector of that phrase/word node.

Suppose the dimension of the meaning vectors is $d$. The parameters $\theta$ of an RAE are the weights on the edges, including the extended combination-weight matrix $W = [W_l, W_r, b_w] \in \mathbb{R}^{d \times (2d+1)}$, the extended reconstruction-weight matrix $U = [U_l, b_l; U_r, b_r] \in \mathbb{R}^{2d \times (d+1)}$ and the extended label matrix $V = [V_0, b_v] \in \mathbb{R}^{p \times (d+1)}$, where $p$ is the dimension of labels. We also learn the meaning vectors of each word in the vocabulary, $L \in \mathbb{R}^{d \times C}$, where $C$ is the size of the vocabulary.

Suppose an internal node $s$ receives activation $a$, and its value $r = \tanh(a)$. The two children of $s$ are $[c_l]_s$ and $[c_r]_s$, with values $c_l$ and $c_r$. The parent node of $s$ is $p_s$. The two reconstruction nodes are $[y_l]_s$ and $[y_r]_s$, with values $y_l$ and $y_r$. The label node is $v_s$ with value $v$. The feedforward formula of the values in the RAE are,
\[ r = \tanh(W[c_l; c_r; 1]) \]  
\[ [y_l; y_r] = \tanh(U[r; 1]) \]  
\[ v = V[r; 1] \]  

2.2 Error and error gradient

For each non-terminal node \( s \), the reconstruction error is defined as a square loss
\[ E_{rec,s} = \frac{1}{2} \left( \| [y_l]_s - [c_l]_s \|^2 + \| [y_r]_s - [c_r]_s \|^2 \right) \]  

For each non-output node \( s \), the label error can be defined using the cross-entropy as in the course notes:
\[ E_{label,s} = -t_s^T \log \text{softmax}(V[r_s; 1]) \]  
or a square loss as in Socher’s code
\[ E_{label,s} = \frac{1}{2} \| t_s - v_s \|^2 \]

where \( t_s \) is the target label for \( s \). For the sake of computational simplicity, we use squared loss in our implementation.

The error \( E^{(m)} \) for the \( m \)’th training example is defined as the sum of these two kinds of errors over all associated nodes in the RAE of that example. For the \( m \)’th example, denote the set of non-terminal nodes and the set of non-output nodes (including leaves) in its RAE as \( NT^{(m)} \) and \( NO^{(m)} \) respectively.
\[ E^{(m)} = \alpha \sum_{s \in NT^{(m)}} E_{rec,s} + (1 - \alpha) \sum_{s \in NO^{(m)}} E_{label,s} \]  

The total error for all training examples is
\[ J = \frac{1}{N} \sum_m E^{(m)} + \frac{\lambda}{2} \| \theta \|^2 \]

where \( N \) is the total number of non-terminal nodes. It follows that the error gradient with respect to the parameters \( \theta \) is
\[ \frac{\partial J}{\partial \theta} = \frac{1}{N} \sum_m \frac{\partial E^{(m)}}{\partial \theta} + \lambda \theta \]

2.3 Compute \( \delta \) for each node

For each node \( s \) in the RAE of one example, we define a vector \( \delta_s = \partial E^{(m)}/\partial a_s \), i.e. the error gradient with respect to the input vector \( a \) of that node. We distinguish between output nodes and non-output nodes.

The output nodes in an RAE are the two reconstruction nodes and one label node for each non-terminal node. For all reconstruction nodes \([y_l]_s \) and \([y_r]_s \),
\[ \delta_{[y_l]}_s = \alpha (y_l - c_l) \]  
\[ \delta_{[y_r]}_s = \alpha (y_r - c_r) \]

For all label nodes \( v_s \), with squared loss
\[ \delta_{v_s} = (1 - \alpha) (r_s - t_s) \sigma' (a_{v_s}) \]
The non-output nodes in an RAE are the phrase nodes and leaf nodes. For each such node \( s \), we compute its \( \delta \) vector by backpropagation from the \( \delta \) vectors of connected output nodes.

We first define \([W_{lr}]_s\),

\[
[W_{lr}]_s = \begin{cases} 
W_l & \text{if } s \text{ is a left child of } p_s \\
W_r & \text{if } s \text{ is a right child of } p_s
\end{cases}
\]

If \( s \) is not a leaf node,

\[
\delta_s = \tanh'(a_s) \circ \left( \delta_{[y_l]s}^T U_l + \delta_{[y_r]s}^T U_r + \delta_{p_s}[W_{lr}]_s - \alpha([y_r]_{ps} - r_s) + \delta_{v_s}^T V \right) \quad (13)
\]

If \( s \) is a leaf node,

\[
\delta_s = [W_{lr}]_s^T \delta_{p_s} - \alpha([y_l]_{ps} - r_s) + V_0^T \delta_{v_s} \quad (14)
\]

With these \( \delta \) vectors, we can easily compute the error gradient with respect to the parameters.

### 2.4 Compute the gradient for each node

The total error gradient is the cumulation of the error gradient obtained at each node.

The change of \( U \) propagates to the total error through reconstruction nodes, so for each reconstruction node \([y_l]_s\) and \([y_r]_s\) we compute the gradient with respect to \( U \):

\[
\frac{\partial E_s}{\partial U} = [\delta_{[y_l]_s}; \delta_{[y_r]_s}][r; 1]^T 
\]

(15)

The change of \( W \) propagates to the total error through phrase nodes, so for each phrase node \( s \), we compute the gradient with respect to \( W \).

\[
\frac{\partial E_s}{\partial W} = \delta_s[a_l; a_r; 1]^T 
\]

(16)

The change of \( V \) propagates to the total error through each label node \( v_s \), so for each label node \( v \), we compute the gradient with respect to \( V \).

\[
\frac{\partial E_s}{\partial V} = \delta_v[r_s; 1]^T 
\]

(17)

The change of \( L \) propagates to the total error through each leaf node \( s \). Because there is no nonlinearity or summation over inputs at a leaf node, the gradient is simply

\[
\frac{\partial E_s}{\partial L} = \delta_s 
\]

(18)

### 3 Implementation

#### 3.1 Parameters

In our experiments, the input vector dimension \( d = 50 \), the label dimension \( p = 1 \). We set \( \alpha = 0.2 \).

#### 3.2 Compute gradient via backpropagation

We first do a forward propagation starting from the leaves, during which we compute the values of all nodes, as well as the \( \delta \) vectors for all output nodes.

Then we do a backpropagation to compute the \( \delta \) vectors for all non-output nodes. Starting from the root node, we perform a depth-first traverse of all non-output nodes, and compute the \( \delta \) vector for each node visited. In this way, we guarantee that when we compute the \( \delta \) for a node, the \( \delta \) vector of its parent non-output node is already available.
When the $\delta$ vector of a node is obtained, computing the contribution to the total gradient through that node is straightforward by equation (12)-(15). We then add up all the contributions from each node, and obtain the total gradient.

The computed gradients are compared to the results of numerical differentiation of the loss function\[1\]. We test the gradient starting from a minimal RAE with only two inputs and each input is simply a two-dimensional vector. When the gradients match, we then increase the number and dimension of inputs, and add nonlinearity and normalization to the values. We also try to determine whether the problem comes from the reconstruction part or the label part, by testing with $\alpha = 0$ and $\alpha = 1$. After step-by-step debuggings, we manage to make the ratio between computed gradient for the complete RAE and the results of numerical differentiation extremely close to unity, which confirm the correctness of our gradient evaluation.

### 3.3 Optimization

For optimization, we used L-BFGS\[2\]. We originally intended to compare its performance against SGD, but due to time constraints, we did not manage to do the experiment. We guess that as a batch optimization technique, L-BFGS will be slower than SGD, which makes an update to the weights after processing each example.

### 3.4 Prediction

After we obtain the parameters, we build an RAE for each test example, and then compute label value for the top node using the trained $V$ matrix. In Socher’s code, they take the top node’s vector and the average of all vectors in the tree as a concatenated feature vector, and train a separate classifier using logistic regression. After three iterations, our method gives an accuracy of 68.9%, while Socher’s method gives an accuracy of 75.4%. This shows that Socher’s methods is empirically better because it considers the average meanings in the phrase level and is more robust to noise at the top node.

### 4 Discussion

#### 4.1 Is normalization necessary?

Theoretically, normalization is necessary for a node’s value to be a proper meaning vector which lies on an embedded $d$-dimensional sphere. However, we did not normalize the node values in our implementation, as it makes the gradient harder to calculate. We estimate that normalization does not play an important role in the prediction accuracy since ultimately it is the proportion of elements in the meaning vector that determine the prediction, not the magnitude of the meaning vector. In addition, the value of each node will not be too large due to lack of normalization as it is restricted by a sigmoid function.

#### 4.2 Is full backup necessary?

Dividing a RAE into separate triplet training examples makes the gradient computation easier, but the obtained gradient is just an approximation to the true gradient. An advantage of this approximation may be the possibility of more parallel computation.

### 5 Conclusion

In this project we implemented the recursive autoencoder (RAE) as described in Socher’s paper to detect the sentiment of movie reviews. We detailed the algorithms for using backpropagation to compute error gradient with respect to the parameters. We train and

---

\[1\] http://people.csail.mit.edu/jrennie/matlab/checkgrad2.m
\[2\] http://www.di.ens.fr/~mschmidt/Software/minFunc.html
test our RAEs with sentences from movie reviews, and achieve 75.4% accuracy. We also
discussed some design choices and algorithmic issues in our implementation.