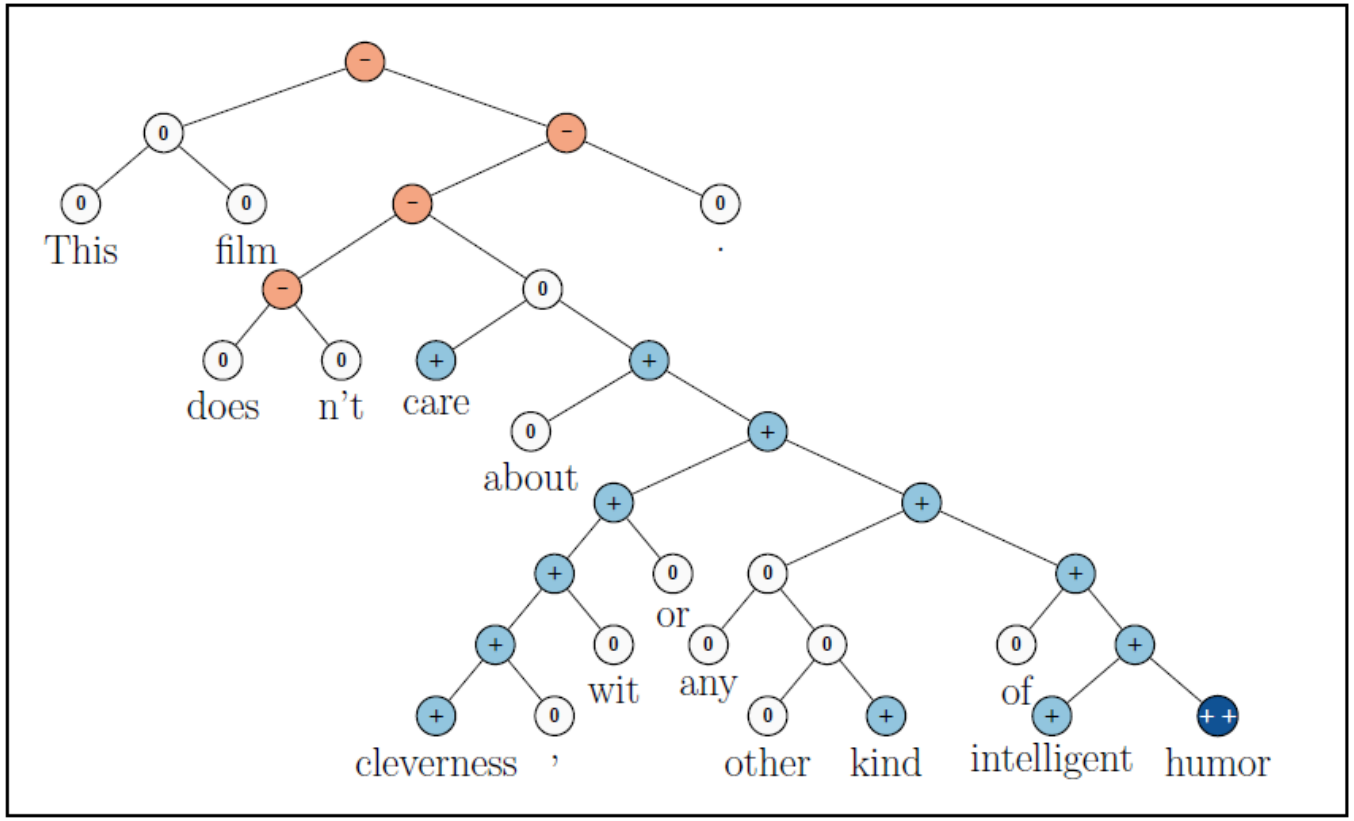
Trees & Recursive Models

Socher et al. (2013). Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank. In EMNLP 2013.



The Problem:

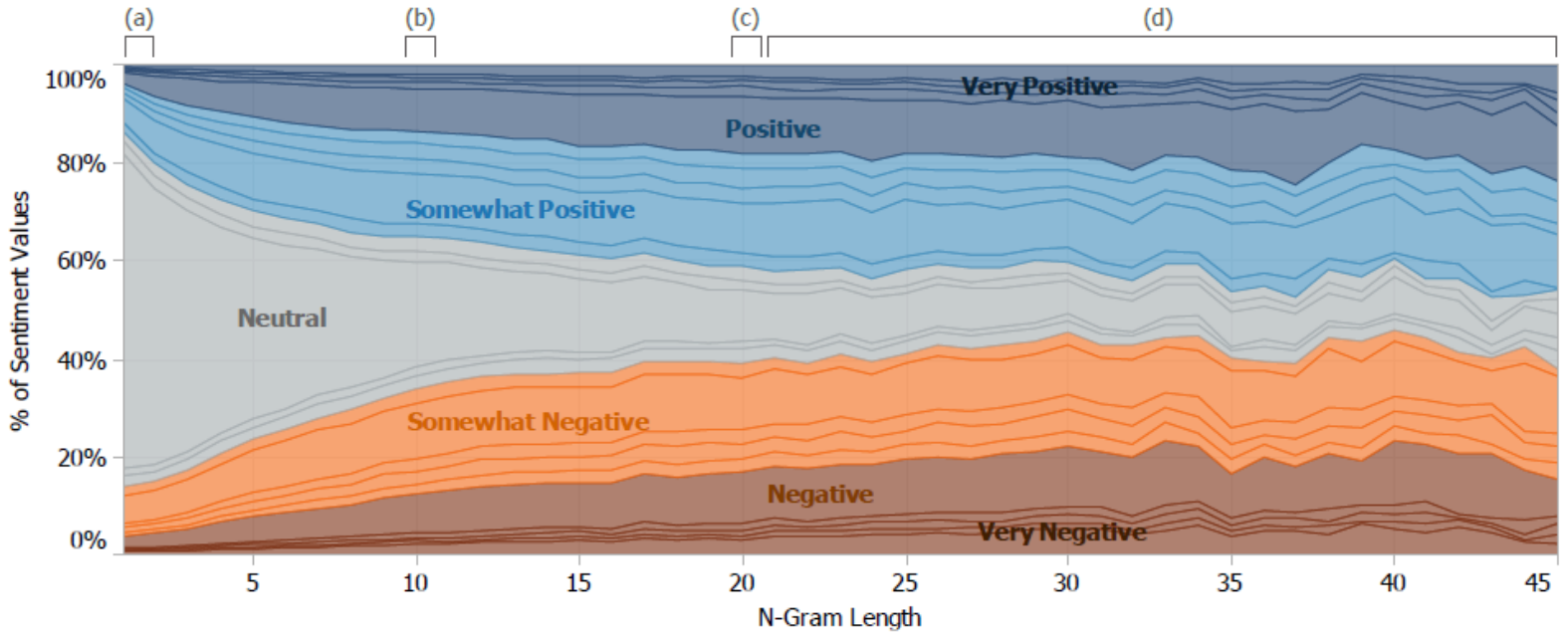
- Semantic vector spaces are useful as features, but they can't capture the meaning of larger phrases properly
- Compositionality is the logical next step, but there weren't really good large and labelled resources and models
- Two contributions:
  - Stanford Sentiment Treebank
  - Recursive Neural Models: introducing Recursive Neural Tensor Network (RNTN)

Stanford Sentiment Treebank

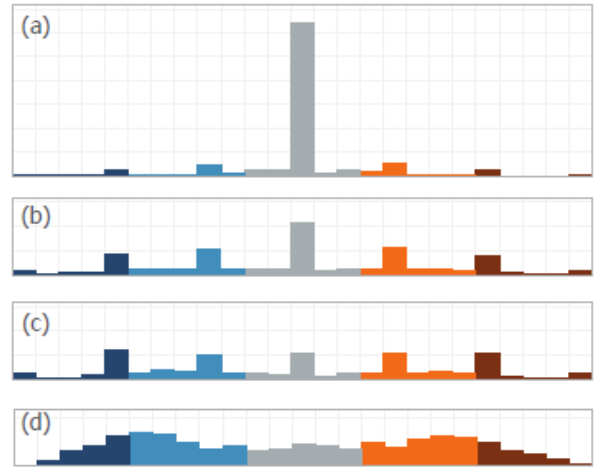
- Researchers took an existing corpus of Rotten Tomatoes reviews
  - 10,662 sentences
  - Half positive, half negative
  - Classified based on the overall sentiment of the review
- Stanford parser was used to parse all sentences,
- Split sentences up to produce a corpus of 215,154 unique phrases
- Amazon Mechanical Turk used to label the resulting phrases on a 7-point scale

How did humans annotate sentiment?

- People really don't like using the extremes, so a 5-point scale is sufficient to capture the main variability of these labels
- The shorter the n-gram, the more likely it is to be rated as neutral by annotators, but for longer n-grams the phrases are well-distributed

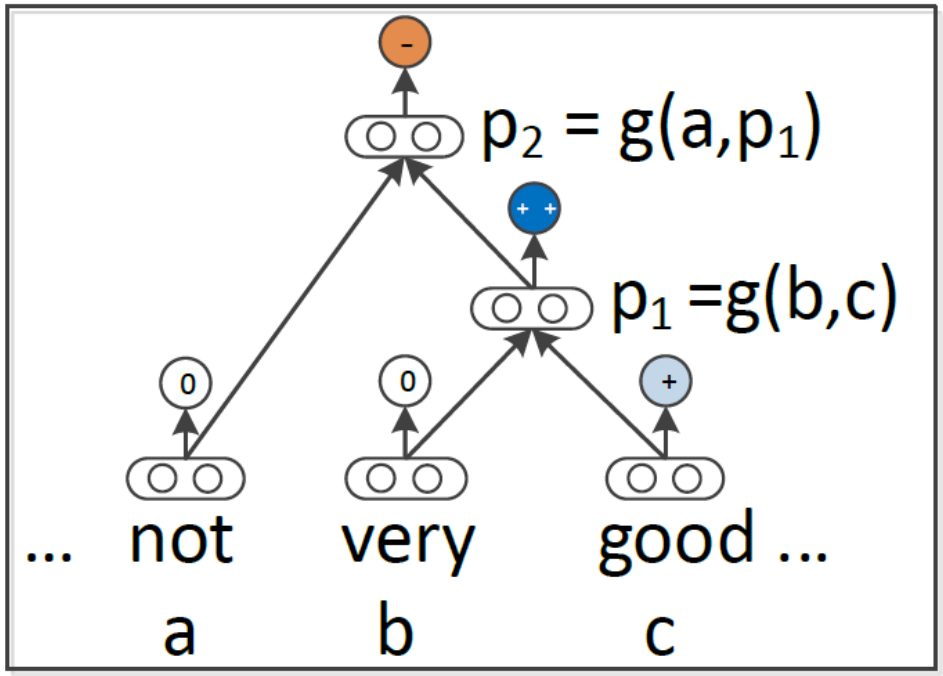


Distributions of sentiment values for (a) unigrams, (b) 10-grams, (c) 20-grams, and (d) full sentences.



Recursive Neural Models

- When a model is given an n-gram, it parses it into a binary tree and each leaf is represented as a vector
  - These word vectors can be used as parameters to optimize and inputs to softmax classifier with the sentiment classification matrix W_s
- A composition function *g* is then used to compute parent vectors and "climb" the tree from the bottom up until the whole n-gram can be classified
- The main difference between models is how they compute p_1
- They compare three different recursive models:
  - RNN
  - MV-RNN
  - RNTN



RNN: Recursive Neural Network

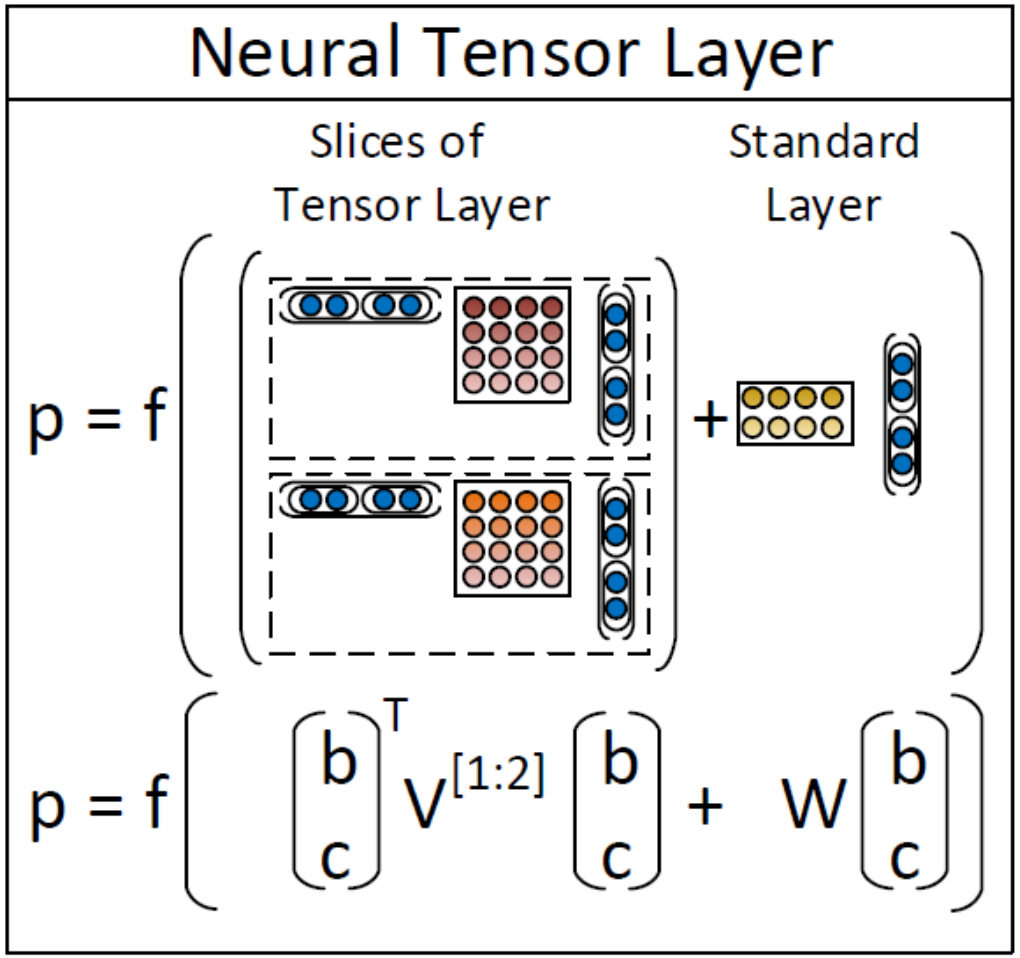
$$p_1 = f\left(W\begin{bmatrix} b \\ c \end{bmatrix}\right), p_2 = f\left(W\begin{bmatrix} a \\ p_1 \end{bmatrix}\right)$$

- By far the simplest model of the ones we're looking at
- First it determines which parent already has all its children computed
- Then it multiplies the vector representations of the children by the matrix *W*
  - *W* is the main parameter to learn
  - Nonlinearity function *f* (here tanh is used) is then applied
- The parent vectors must be of the same dimensionality to be compatible
- Each parent vector is fed to the same softmax classifier as was used for the leaves

MV-RNN: Matrix-Vector RNN

$$p_1 = f\left(W\begin{bmatrix} Cb \\ Bc \end{bmatrix}\right), P_1 = f\left(W_M\begin{bmatrix} B \\ C \end{bmatrix}\right)$$

- Each word is represented as both a vector *a* and a matrix *A*
- The parent node's vector is computed by
  - multiplying the left node's matrix by the right node's vector and vice-versa
  - applying the weighting matrix followed by the nonlinearity function
- The parent node's matrix is computed by
  - concatenating the children's matrices
  - applying a different weighting matrix W_M followed by the nonlinearity function
- The vector for each node is used for classifying and passed to the same softmax used for the leaves
- Since each matrix here is a parameter, the number of parameters becomes very large and depends on the size of the vocabulary, which is less than ideal



RNTN: Recursive Neural Tensor Network

- Uses the same, tensor-based composition function for all nodes
- Introducing $V^{[1:d]}$, a tensor with as many layers as there are dimensions in the word vectors
- For each layer:
  - the word vectors and the transpose thereof are multiplied by that layer of the tensor
  - the word vectors are also multiplied by the traditional weight matrix
  - those products are added together, and then the nonlinearity function is applied
- The same weights are used for each node
- The benefit over an ordinary RNN is that when *V* is set to 0, the tensor directly relates the input vectors in a way RNNs can't.

Training the RNTN Model: Tensor Backprop through Structure

- Maximize the probability of the correct prediction – aka, minimize the cross-entropy error between the predicted distribution y^i and target distribution t^i at node i.
- Each node backpropagates its error through to the recursively used weights V, W
- The softmax error vector is calculated for each node, but it's only the final error value for the topmost node
- It's used to generate a vector that is then passed down to its children and combined with their softmax error vector to produce their final error value (and likewise for their children

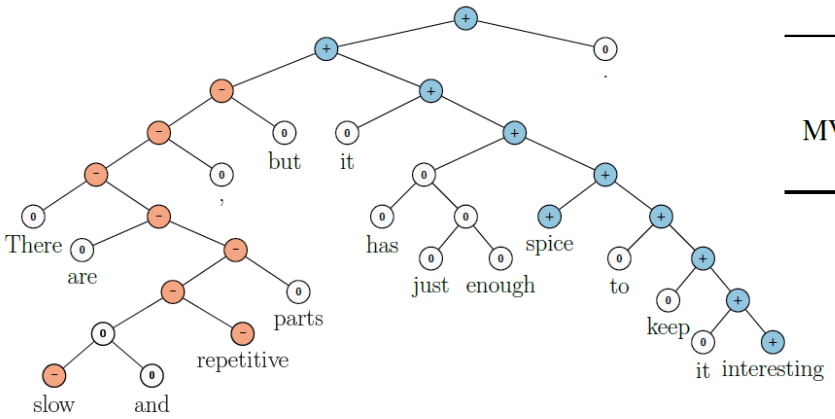
$$\delta^{i,s} = \left(W_s^T(y^i - t^i)\right) \otimes f'(x^i)$$

$$\frac{\partial E^{p_2}}{\partial V^{[k]}} = \delta_k^{p_2,com} \begin{bmatrix} a \\ p_1 \end{bmatrix} \begin{bmatrix} a \\ p_1 \end{bmatrix}^T$$

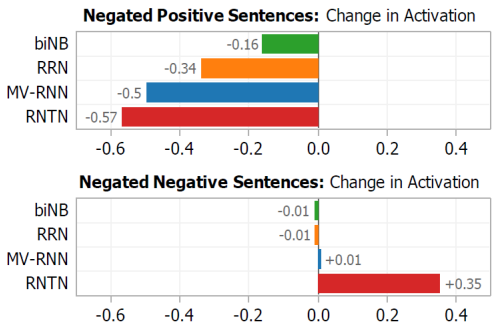
$$\delta^{p_2,down} = \left(W^T \delta^{p_2,com} + S\right) \otimes f'\left(\begin{bmatrix} a \\ p_1 \end{bmatrix}\right)$$

$$S = \sum_{k=1}^{d} \delta_k^{p_2,com} \left(V^{[k]} + \left(V^{[k]}\right)^T\right) \begin{bmatrix} a \\ p_1 \end{bmatrix}$$

Model	Fine-grained		Positive/Negative	
	All	Root	All	Root
NB	67.2	41.0	82.6	81.8
SVM	64.3	40.7	84.6	79.4
BiNB	71.0	41.9	82.7	83.1
VecAvg	73.3	32.7	85.1	80.1
RNN	79.0	43.2	86.1	82.4
MV-RNN	78.7	44.4	86.8	82.9
RNTN	**80.7**	**45.7**	**87.6**	**85.4**



Model	Accuracy	
	Negated Positive	Negated Negative
biNB	19.0	27.3
RNN	33.3	45.5
MV-RNN	52.4	54.6
RNTN	**71.4**	**81.8**



Experiments

- All of the models performed better with the new Stanford Sentiment Treebank to work from
- RNTN outperformed all the other models, beating the state-of-the art at the time
- Performed particularly well on a few specific tasks:
  - Contrastive conjunction (41% correct vs. 37% for MV-RNN)
  - High-level negation
  - Selection of the most strongly positive and negative features for activation