# Rationalizing Sentiment Analysis in Tensorflow

**Alyson Kane**
Stanford University
alykane@stanford.edu

**Henry Neeb**
Stanford University
hneeb@stanford.edu

**Kevin Shaw**
Stanford University
keshaw@stanford.edu

## Abstract

Sentiment analysis using deep learning models is a leading subject of interest in Natural Language Processing that is as powerful as it is opaque. Current state-of-the-art models can produce accurate predictions, but they provide little insight as to why the model predicted this sentiment. Businesses relying on these models might be less likely to act on insight given the lack of evidence for predictions. These people would be more likely to trust such predictions if a brief explanation of the outcome is provided. Recent work by Lei et al [4]. has set forth a framework for a multi-aspect sentiment analysis concurrently providing text rationalization with each prediction. This framework sets forth a two-part approach, which summarizes a review and predicts a sentiment. In this paper, we explore the performance of this framework, seeking to recreate and improve upon it in TensorFlow.

## 1 Introduction

Many businesses and organizations can find a use for sentiment analysis. Customer reviews can provide insights on which products are popular and which need to be redesigned. Communications via email can be scanned to find clients that are dissatisfied with services so that they can be accommodated. Current state-of-the-art sentiment analysis models utilize deep learning architectures and achieve fairly high accuracy when predicting positive or negative sentiment - on the order of 80% to 90% [3].

Unfortunately, with many applications knowing the sentiment of a communication or a review may not be enough. For a sentiment prediction to be useful, a user may need to know why the model predicted a specific sentiment. Further, deep learning models are notoriously uninterpretable, lending to some users not trusting model results, and in some industries and nations, leading to the complete prohibition of their use [1].

In *Rationalizing Neural Predictions* [4], Lei et al propose a framework to address this problem. The first task (generator) provides a rationalization of each rating, selecting a subset of words or phrases from the original text review. A rationale should be a short but comprehensible summary of a review. The authors frame this as a semi-unsupervised learning problem, creating a two-layer bidirectional recurrent neural net (RCNN) which passes over each word in the original text. The resultant vector is a probability that each word from the original text is selected in the summary text.

These probabilities are used to sample summary text, which is fed into the second task (encoder). This is a supervised learning task where the sampled summary text is used to predict the sentiment. The authors used a two-layer RCNN for this task. By forcing ratings to be predicted using only the rationale of the review and not the review itself, we can jointly train tasks by having the first task place higher probabilities on words that are relevant to the sentiment. Further, to force rationale coherency and brevity by adding in a regularization factor that penalizes for selecting too large of a rationale and for not selecting a small span of words.

“a: it **poured a clear orange to copper body with** a modest **white head**. **there** was **decent** layered **lacing that** lingered until the beer was gone. s: it was dominated by sweet **moderately** toasted malts …”

Figure 1: Sample Rationale

Lei et al implemented this framework in Theano. We use this framework as an inspiration for our own implementation in TensorFlow and try to match the author’s sentiment prediction and rationale results as a baseline. We then experiment with different encoder and generator architectures to see if we can improve upon our baseline model. We try a LSTM and GRU implementation of both the encoder and generator. Further, we experiment with forcing a fixed-width coherence and remove the sparsity and coherency regularization parameters.

## 2 Related Work

### 2.1 Framework

Our experiment relied heavily on the work by Lei et al in *Rationalizing Neural Predictions* [4]. The paper prescribes a two-part model which predicts a multi-sentiment analysis (called encoder) and extracts summary phrases (called generator).

The encoder (enc) is a supervised learning problem which predicts a rating given a text review. Training samples are (x, y) pairs, where $x = \{x_t\}_{t=1}^{T}$ is an input text sequence of length T and $y \in [0, 1]^m$ is an output vector where *m* denotes the number of aspects in a review. Loss for the encoder is calculated using squared error loss:

$$L(x, y) \ = \parallel \hat{y} - y \parallel_2^2 = \parallel enc(x) - y \parallel_2^2 \tag{1}$$

The generator (gen) is a text summarization task which selects a subset of words of the text review as a rationale describing rating. There is no target rationale, but rather both the encoder and generator are jointly trained. The output of the generator are probabilities of each word in the original review being selected as part of the rationale. These probabilities are used to sample a $z - layer$, where each $z_t \in \{0, 1\}$ is an indicator variable for each word in a text sequences indicating if a given word was chosen as rationale. There is no target rationale, but rather both the encoder and generator are jointly trained. That is, our final predictions are trained on the rationale output from the generator, not the full text review. Thus, our final prediction in enc(gen(z,x)).

Since the generator is not trained on labels, we introduce regularization to force the number of words that are predicted in the $z - layer$ to be small. Further, rationales need to be coherent, so we need to enforce that words selected are within a range. The authors suggest that we add in a sparsity and coherency penalization. The final cost is defined as:

$$Cost(z, x, y) = L(x, y) + \lambda_1 \parallel z \parallel + \lambda_2 \sum_{t=1}^{T} \mid z_t - z_{t-1} \mid \tag{2}$$

where $\lambda_1$ penalizes number of words and $\lambda_2$ penalizes distance between chosen words.

### 2.2 Expected Cost

The $z - layer$ sampling method is necessary to tractably compute the cost. If we did not sample from the generator’s probability distribution, we would need to compute the full expected cost. This would require us realizing all possible sequences of rationales, computing the cost, and then weighting it by the probability of that specific rational being sampled. For each review of size $k$, we would need to realize a total of $2^k$ possible rationales, which is computationally infeasible.

Sampling the $z - layer$ in TensorFlow kills the gradient, which prevents training the generator and any parameters below the $z - layer$. To allow training, we calculate binary cross-entropy calculation between the true probabilities and the predicted probabilities:

$$H(p, q) \; = \; -\sum_i p_i \log q_i \; = \; -y \log \hat{y} - (1 - y) \log(1 - \hat{y}) \tag{3}$$

The final cost is *Cost(z, x, y)* weighted by the cross-entropy term *H(p, q)*.

## 3 Approach

### 3.1 TensorFlow Framework

The original model as outlined in *Rationalizing Neural Predictions* was created in Theano. A main goal of our project was to recreate this model in TensorFlow. We spent a large portion of time working to translate Theano code to TensorFlow code line by line. This was a huge learning experience, as we found that these deep learning platforms have some fundamental differences.

Overall, Theano seems to provide a lower level framework that can be much more effectively and efficiently wrapped in hand-coded Python numpy operations. Given similar ”compile” and ”run” phases of working with a computational graph, both Theano and TensorFlow allow certain comparable hacks to print variable values at run time. Also, Theano and Tensorflow have a series of roughly equivalent functions. However, Tensorflow requires more explicit type specification (so we used tf.cast() extensively in the in the initial attempt at a Theano to Tensorflow baseline translation) as well as variable specification.

The authors make extensive use of the theano.scan() function. We studied and replicated theano.scan() using tf.scan() which was added relatively recently to tensorflow. We found performance of tf.scan() was an order of magnitude worse than running theano.scan() on the GPU. At this point, we decided to start writing our Tensorflow code from scratch.

### 3.2 Data

#### 3.2.1 Datasets

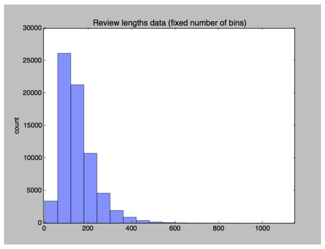
Consistent with the Lei et al. paper, we use a dataset consisting of 1.5 million reviews from BeerAdvocate. BeerAdvocate is a user review website of beer, such that reviews are multi-aspect. That is, reviews are asked to speak to each of five categories describing a beer: look, smell, taste, feel, and overall. Each rating is on a scale of 0 - 5, inclusive.

In addition to this dataset, there is a small subset of almost 1,000 annotated reviews. Annotated reviews have a tag for each sentence in the review indicating which aspect the sentence is speaking to. The model will not train on this data, but instead is used as test data and can be used to calculate precision of final model.

#### 3.2.2 Preprocessing

In the Lei et al. paper, models are trained separately for each aspect. As a proof of concept, we choose to model on only one aspect, namely appearance, and drop all other ratings.

Ratings are a continuous value between 0 and 5, thus we are solving a regression problem. We normalize output to [0, 1] values, allowing us to use a final layer which predicts values between 0 and 1, such as sigmoid or tanh.



As is common in deep learning, we use word embeddings to represent each word in our text. We begin using a 200-dimensional word embedding trained on Wikipedia 2014 data using the GloVe algorithm.

The maximum review length in our dataset was 1,145 words, however the average review length is around 200 words. Adding padding of this length greatly slowed down the model. We chose to clip reviews to 300 words to speed up processing time.

### 3.3 Model

In this section, we outline the various components used in our model.

#### 3.3.1 RNNs

A recurrent neural network (RNN) is a model which conditions on all previous words, making it very useful when working with sequential data. At each step, the next word in the text is fed into a hidden layer, along with the output from the previous hidden layer. Final output is then computed from the hidden state.

In the encoding task of our model, we use a two-layer stacked RNN. The model is as follows:

$$h_1^t = f_1(W_1 x^t + U_1 h_1^{t-1} + b_1) \tag{4}$$
$$h_2^t = f_2(W_2 h_1^t + U_2 h_2^{t-1} + b_2) \tag{5}$$
$$\hat{y} = f_3(R[h_1^T : h_2^T] + b_3) \tag{6}$$

where $x^t$ denotes input word at time t, $h_i^t$ denotes a hidden state at time t, and $\hat{y}$ denotes output. Note $\hat{y}$ is only calculated at time T, where T is the length of each rating.

As a baseline, we implement the above model using tanh for all activation functions. Results for experimentation are described below.

#### 3.3.2 Bidirectional RNNs

Bidirectional RNNs are used when both previous and future words are useful for determining output. At each layer, the input is fed into two hidden states, one which is fed forward through time steps and the other which is fed backward.

$$\overrightarrow{h}^t = f_1(\overrightarrow{W} x^t + \overrightarrow{U}\overrightarrow{h}^{t-1} + \overrightarrow{b}) \tag{7}$$
$$\overleftarrow{h^t} == f_2(\overleftarrow{W} x^t + \overleftarrow{U}\overleftarrow{h}^{t+1} + \overleftarrow{b}) \tag{8}$$
$$\hat{y} = f_3(R[\overrightarrow{h}^T : \overrightarrow{h}^T] + c) \tag{9}$$

where variables with $\rightarrow$ denote inputs moving left to right and $\leftarrow$ denote inputs moving right to left.

Similar to the stacked RNN, we use tanh as an activation function and experiment with various other activations.

#### 3.3.3 Vanishing and Exploding Gradients

Vanishing and exploding gradients are common occurrences in deep learning. As gradients are backpropagated through time steps, we are continuously multiplying by the gradients of the weights and biases of each layer. When these values are small, the gradient of a layer many time steps back will approach zero and thus never update. Alternatively, if these values are large, we will see an exploding gradient.

#### GRUs

Vanishing Gradients can often be solved using Gated Recurrent Units (GRU). GRUs are an update to RNN activation units which capture long-term dependencies.

$$z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1}) \tag{10}$$
$$r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1}) \tag{11}$$
$$\widetilde{h_t} = tanh(r_t \circ Uh_{t-1} + Wx_t) \tag{12}$$
$$h_t = (1 - z_t) \circ \widetilde{h_t} + z_t \circ h_{t-1} \tag{13}$$

The biggest change to this cell is $r_t$, the reset gate. The reset gate determines how much of past memory ($h_{t-1}$) tp pass along to the next state.

**LSTMs**

A modification of the GRU is a Long-Short-Term-Memories (LSTMs). Within a hidden unit, the LSTM has an input gate [eq (14)], which controls which words can be passed into the unit and an output gate [eq (15)], which controls how much of the memory can affect the next hidden state.

$$i_t = \sigma(W^{(i)}x_t + U^{(i)}h_{t-1}) \tag{14}$$
$$f_t = \sigma(W^{(f)}x_t + U^{(f)}h_{t-1}) \tag{15}$$
$$o_t = \sigma(W^{(o)}x_t + U^{(o)}h_{t-1}) \tag{16}$$
$$\widetilde{c_t} = tanh(W^{(c)}x_t + U^{(c)}h_{t-1}) \tag{17}$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \widetilde{c_t} \tag{18}$$
$$h_t = o_t \circ tanh(c_t) \tag{19}$$

**Gradient Clipping**

Exploding gradients can be solved using a technique called gradient clipping. Each time the gradient surpasses a set threshold, we reset the gradient to a given upper or lower bound. Specifically, we used bounds if 1, -1 (upper, lower).

#### 3.3.4 Adam Optimization

We chose to use Adam optimization [2] in our model, which is a more complex update method as compared to Stochastic Gradient Descent. Over each batch of data, we update parameters using algorithm:

$$m \leftarrow \beta_1 m + (1 - \beta_1)\nabla_\theta J_\theta \tag{20}$$
$$v \leftarrow \beta_2 v + (1 - \beta_2)\nabla_\theta J_\theta^2 \tag{21}$$
$$\theta \leftarrow \theta - \alpha \circ m/\sqrt{v} \tag{22}$$

We keep a rolling average of the first and second moments. The first moment, $m$, will prevent the gradient from varying too much. The second moment, $v$, helps update parameters with small gradients to speed learning.

## 4 Experiments

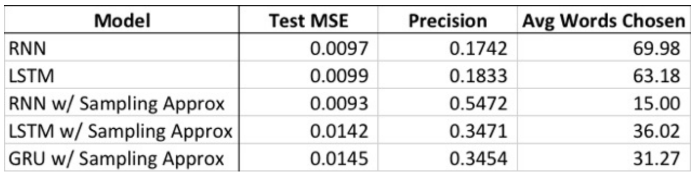
### 4.1 Initial Approach

As a baseline, we sought to re-create the model proposed by Lei, et al. in TensorFlow. We implemented a model with a two-layer stacked bi-directional RNN generator and a two-layer stacked RNN encoder. Training mean squared error (MSE) decreased gradually; however, after 10 or so epochs our MSE spiked. This behavior indicates an exploding gradient, solved using gradient clipping with bounds [-1, 1].

After this fix, the model produced MSE results consistent with the paper but with poor precision. Calculating the norm of each gradient, we were able to detect a vanishing gradient issue. When sampling words from the generator output, we are creating a break in our model graph and can no longer back propagate the gradient past this layer. Interestingly, the method used by the paper's authors to fix this issue (weighting cost by cross-entry), had no effect on our model.

In figure 1, we can see all models achieve a low test MSE, even when the generator is not training. This is likely because all aspects are highly correlated. If we choose a random subset of words which do not pertain to the aspect we are focusing on, we will still reduce MSE but score poorly on precision.

We note that if we were to randomly select words for rationales across our test data, our expected precision would be approximately 18%. The RNN and LSTM with out sampleing approximation have precision around this random selection, which is a sign that our generator is not training and not selecting good rationales. However, when we do sample, we see that we beat a random sample approach, with our best model performing at 54% precision.



| Model | Test MSE | Precision | Avg Words Chosen |
|---|---|---|---|
| RNN | 0.0097 | 0.1742 | 69.98 |
| LSTM | 0.0099 | 0.1833 | 63.18 |
| RNN w/ Sampling Approx | 0.0093 | 0.5472 | 15.00 |
| LSTM w/ Sampling Approx | 0.0142 | 0.3471 | 36.02 |
| GRU w/ Sampling Approx | 0.0145 | 0.3454 | 31.27 |

Figure 2: Model Results

## 4.2 Sigmoid Sampling Approximation

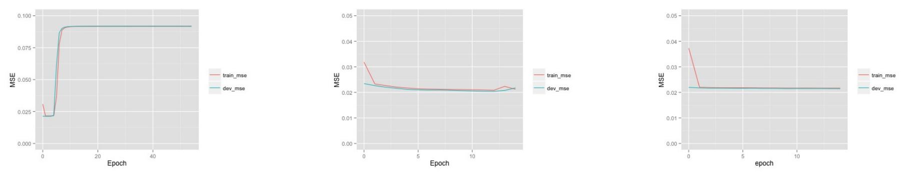
We were ultimately able to solve this problem using a sigmoid function to approximate sampling. By adding a large $\alpha$ parameter to the function, we create a steep sigmoid curve which pushes values toward 0 and 1. Data is centered so probabilities above 0.5 are chosen and those below are not.

$$z = \sigma\{\alpha(p(z \mid x) - 0.5)\} \tag{23}$$

Choosing an $\alpha$ = 60 was still a differentiable function and weights in the generator were able to train. However, we continue to be open to other better Tensorflow-based approaches to achieving this same result.

## 4.3 Experimenting with Cells

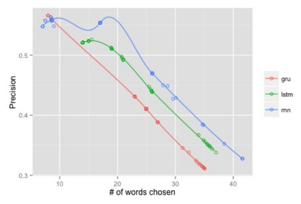
We performed experiments using RNN, LSTM, and GRU cells. We believed that LSTM or GRU models would outperform the RNN model because of their ability to understand long-term dependences. This hypothesis was incorrect; the RNN achieved the best results in both mse and precision (see figure 2). This result makes sense, because perhaps we do not need as long-term of a memory as we had originally assumed. The average length of each review is 300 words, but the average number of words in a review which pertain to a given aspect is only 20 words.



(a) LSTM (b) RNN (c) GRU

Figure 3: Learning Curves for Models with Sampling Approximation

The charts in figure 3 show learning rate for each cell in our experiment. All models stop learning after few epochs. This is likely because parameters are not tuned, as we ran out of time at the end of the project.



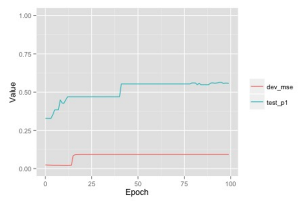
On the right, precision plotted by average number of words chosen as rationale for each review is shown. We see that precision increases as the number of words chosen decreases. We see the best precision when about 15 words are chosen as rationale.

## 4.4 Rationales of Best Model

Visual inspection of rationales on our best model show that we are generally extracting words and phrases that pertain to the desired sentiment. However, we notice that we sometimes select sporadic words which may pertain to the aspect, but are not coherent (see figure 1 - Sample Rationale). This could be signs that we have not tuned our coherency and sparsity regularization parameters well. In the future, we would like to experiment more with these parameters.

## 4.5 Precision per Epoch

To get a better understanding of how our model was learning to predict rationales, we trained our RNN model and evaluated extracted rationale precision per epoch. We noticed while training that precision remained stagnant for a number of epochs and appeared to "jump" to higher values as training proceeded. These jumps also come at the price of higher MSE in our development predictions.



There are a couple of explanations for this behavior. Our approximate rounding function that allows parameter training and sparsity and coherency regularization is very sensitive to $z - layer$ probabilities differing from $0.5$. Probabilities that are slightly above $0.5$ are rapidly pushed towards $1$ while those less than $0.5$ are pushed towards $0$. Given we randomly initialize our generator parameters such that most our initial $z - probabilities$, and thus $z - layer$ predictions are random in the first forward propagation, we train the encoder parameters on a random selection of words. It is unclear how much of the error in the back propagation affects the generator parameters, but assuming moderate changes, we believe that it is unlikely that the $z - layer$ predictions are changing that frequently. This means we train our encoder on roughly the same subset of words for the first few epochs. Given the cliff between being selected in a rationale or not, we believe that the jump in precision may come from after sufficient training causes a switch in the number of generator parameters, causing our generator to fundamentally select different words that are passed into the encoder. This would have the ability to provide better precision as we may be selecting better words for our rationale. However, our encoder was training on a different subset of words this whole time, so it is not necessarily optimized for this new subset of words, which may cause our development MSE to increase and then subsequently decrease as our encoder trains better to the new subset.

Given random parameter initialization and the fact that our sampling is very inhospitable to predictions even slightly below $0.5$, we recognize that it may be difficult for our generator to get out of local minima. To this extent, it may be better to decrease our sampling threshold (say take all rationales that are above probability $0.3$) so we can sample more words. Also, we may want to try an ensemble approach where we train a couple models with random initialization and explore various model minima.

Another possibility for this phenomena is that over time our sparsity and coherence regularization becomes a dominant factor in our cost. The regularization that wants to keep the number of predictions in our $z - layer$ small can dominate, forcing our generator to pick fewer and fewer words. This would cause our precision to jump because we would be making fewer erroneous predictions. However, the fewer overall predictions could make our encoder perform worse as it has less information to make a prediction, which would also explain the increase in MSE.

### 4.6 TensorFlow Sampling

As discussed briefly above, the most significant challenge we faced was an attempt in TensorFlow to pass a gradient through a sample. The framework for producing rationales (excerpts from reviews) is based on selecting subsets of words (a sample) that best predicts sentiment for a specific aspect while also attempting to minimize weights related to sparsity (fewer words selected) and coherence (words close together). The authors do not seem to have gradient passing challenge in Theano. He simply uses:

```
theano.disconnected_gradient()
```

However, in TensorFlow any casting to boolean or integer or even rounding a float kills the propagation of the gradient. Further, we found that weighting cost by cross-entropy term had no effect on saving the gradient. Digging into this issue required us to build a separate minimal back propagation example and try many different approaches. The approach which was ultimately sucessful was the sigmoid sampling approximation.

#### 4.6.1 Word Span Rationale Selection

We first attempted to remove the need for sampling words by instead selecting the best fixed-length word span over the entire review. First, we defined a fixed span length of 10, however this would better be served as a tunable parameter. Next, we iterate over each element in the review as the starting point, capturing all words in the window. We choose the span which minimized mse.

This method would require computing an expected cost, but would require us to realize far fewer rationales than if our previous method of allowing each word according to a probability. For a review of length $k$, this would require $O(k)$ rationale and cost computations (as compared to the $O(2^k)$ computations with each word being a binary choice). Despite these savings, preliminary experiments took a prohibitively long time. We estimate that it would take approximately $10$ hours to complete a single epoch. Due to the prohibitive time cost, we abondoned this methodology.

## 5 Conclusion and Next Steps

We were very quickly able to achieve a low MSE with any model we experimented with, suggesting that such a complex model isn't needed to predict rating. However, achieving a high precision was a much harder task. Choosing a random subset of words across a review provides an expected precision of $18\%$. Our best model achieves precision of 3 times this random value, $54\%$.

Using TensorFlow for deep learning has a very steep learning curve. We spent much of our time building out the basic components of our model. In the future, we would like to continually build upon the model that we have created. If we had more time, we would like to focus on parameter tuning and experimenting with other deep learning architectures, such as convolution neural nets.

Sample rationales are promising in that they appear to be selecting words from reviews that refer to appearance qualities. However, qualitative review of rationales exposes that selections are not as coherent as desired. As mentioned in the *Precision per Epoch* section, we believe that this may be caused by our stringent sampling procedure causing our generator to get stuck in local minimum, or it could be due to our sparsity regularization parameter dominating our coherency regularization parameter. An imbalance between sparsity and coherency means we could prefer sparse predictions over coherent ones. For next steps, we would really like to tune our sampling cutoff and our sparsity and coherency regularization. Further, we believe that creating an ensemble model may help with prediction as we can explore various local minimums.

We believe that this framework could be incredibly useful in the medical field. Doctors may be unwilling to trust a machine prediction of an x-ray. However, if we could extract a rationale for classification diagnoses of image from existing medical notes or records, such technology may be more easy adopted. We hope to continue to explore such applications in further projects next quarter.

## References

[1] Seth Flaxman Bryce Goodman. European union regulations on algorithmic decision-making and a right to explanation. *arXiv preprint*, 1606.08813, 2016.

[2] Jimmy Ba Diederik P. Kingma. Adam: A method for stochastic optimization. *arXiv preprint*, 1412.6980, 2014.

[3] Jean Y. Wu Jason Chuang Christopher D. Manning Andrew Y. Ng Richard Socher, Alex Perelygin and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Empirical Methods in Natural Language Processing (EMNLP)*, volume 1632, 2013.

[4] Regina Barzilay Tao Lei and Tommi Jaakkola. Rationalizing neural predictions. *EMNLP*, 2016.