Colors in Context: An Implementation

Alec Brickner
Stanford University
Palo Alto, CA 94305
abrickne@stanford.edu

Abstract

Colors tend to be defined by the contexts in which they occur. A yellowish-green may be interpreted as a "yellow" when there are darker greens nearby, but that same color might be interpreted as a "green" when there are sharper yellows around it. We implement a model defined by Monroe, et al. [1] to choose a color in a context given a description, which utilizes the Rational Speech Acts framework with an LSTM at the base level.

1 Introduction

In their paper C̈olors in Context: A Pragmatic Neural Model for Grounded Language Understanding,̈ Monroe et al. [1] build a model that learns to choose a color out of a set of 3, given a description of the correct color in context of the other two colors. The motivation for this task is that sometimes, colors do not occur in isolation. It may be useful to have a model that can output a color based on a description, but relative to other colors, people might describe colors differently. What might be "blue" in isolation could become "light blue" when a darker blue is placed next to it. Further, it could become "the middle shade of blue" when both a lighter and darker shade of blue are placed beside it. I have implemented the model from the paper myself, and will discuss the details of implementation in this paper.

2 Task Description

The basic goal of the task was mentioned previously; I will go into it in further depth here. Data was collected by Monroe et al. [1] using a Mechanical Turk HIT that paired 2 people together. Each person was assigned a role, which was either "speaker" or "listener". Both the speaker and listener were presented with three colors, and one of these three colors was specified to the speaker. The speaker's goal was to describe this color to the listener in such a way that would allow the listener to choose that color correctly. The result of this data collection, publicly available on the Computation and Cognition Lab's website, is the data to be used in our implementation.

3 Implementation

Because of the setup, in which we have a speaker talking to a listener who must choose between options, the approach used to solve this task is the Rational Speech Act framework [2] [3]. The intuition behind this framework works as follows. Suppose we have a speaker, S, and a listener, L. S makes an utterance, and L must try to understand what it means. L, assuming that S is speaking rationally, knows that S is saying something which is meant to get L to understand the true meaning of the utterance. She believes that S is considering a potential set of utterances, and imagining how the listener would respond to each. In this way, the listener is thinking about the speaker, who is in turn reasoning about the listener. This could potentially be recursive, but most implementations

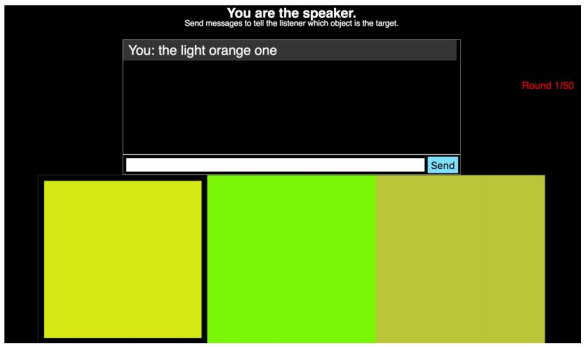

Figure 1: The HIT as seen by the speaker. Image from the website of the Computation and Cognition lab.

(including this one) stop here. Monroe et al. choose to call the top-level listener (the *pragmatic* listener) L_2, the speaker S_1, and the base-level listener L_0.

Usually, L_0 is implemented with a fixed vocabulary, and might use a tool such as basic, rule-based semantic parsing in order to understand the sentence. In practice, however, this does not scale - implementing RSA either requires a high-level semantic parser, or another tool that allows for a wide variety of sentence types. Monroe et al. use a neural net at the base layer, which lets any sentence work as a listener to the input.

Additionally, note that the speaker, reasoning about the listener, must consider the listener's predicted response over a set of sentences. In the standard RSA algorithm, this is usually a distribution over results from a fixed set of potential utterances. However, in practice, we don't have a fixed set of sentences - we have a speaker, who is thinking about all potential utterances they might produce. To that end, Monroe et al. use a base *speaker* neural net, S_0, which produces text based on a color. S_1 takes samples from this network by using the color it wants the listener to guess, and then runs those utterances on L_0 to see what it will think in regards to those utterances.

3.1 Base Listener

The base listener is an LSTM, which takes in an utterance as input and outputs a probability distribution over the three possible colors. See Figure 2 for a visual representation. Each input word is sent through an embedding layer (initialized using a normal distribution N(0, 0.01)), which is of size 100. After this layer is the recurrent LSTM layer, which is bidirectional. The LSTM cell is implemented as in [4], which includes the memory cell in the calculations of each gate, and has a trainable weight vector for the memory cell input of each gate. Each weight matrix is initialized using the normal distribution N(0, 0.1), while bias vectors are all initialized to 0. The size of this layer is 100. Note that while padding is implemented, with the total length of the recurrent layer being set to the length of the longest input in the training set, masking is *not* implemented. Even if the input is just a single word followed by a large amount of padding, the network goes all the way until the end of the input, counting every pad input. This is likely to be important because the length of the input sentence provides valuable information - for example, longer sentences might mean that the correct color is more difficult to guess (shown to be true by Monroe et al.)

The final hidden states of each LSTM direction are concatenated together, creating a 200-cell vector. We then multiply this vector by two weight matrices to get a size-54 vector, μ, and size 54x54 matrix, Σ, parameters of a quadratic form. The weight matrices are initialized using Xavier initialization; a bias, initialized to 0, is added to μ, while a bias that is initialized with the identity matrix is added to Σ. μ and Σ parameterize a Gaussian distribution, where μ is a vector of expected values of the distribution, and Σ is a covariance matrix. (Monroe et al. note that this is not guaranteed to be a Gaussian distribution, but it is in about 95% of cases.)

The labels for the L_0 network are three vectors of size 54, each encoding a color. To get these vectors, the initial RGB values for each color are Fourier-transformed, described in [5]. (The code to do this was taken from the codebase of [1].) Each of these vectors is combined with μ and σ, and

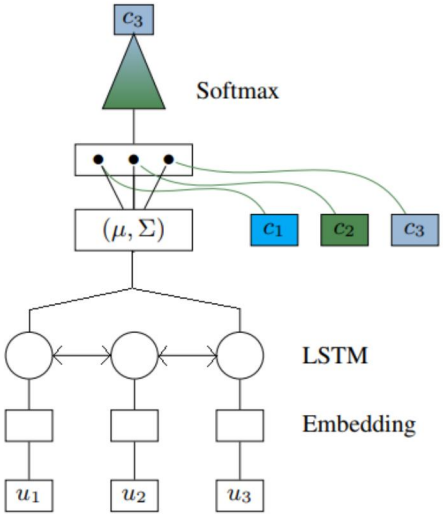

Figure 2: The base listener model. It takes in a sequence of words, passes them through a bidirectional LSTM, and computes the parameters of a quadratic form. Each color is scored with this quadratic form; the softmax of these scores is the output. Original figure from [1], with modifications added to show bidirectionality.

the quadratic form is calculated to produce a score as follows (taken from [1]):

$$\text{score}(f) = (\mu - f)\Sigma(\mu - f)^T \tag{1}$$

The results of this equation for each of the three colors are passed into the softmax function, and from there, the cross-entropy loss is taken with the label of the correct color.

3.2 Base Speaker

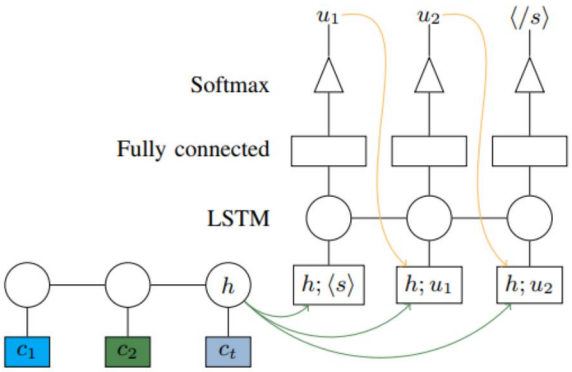

Figure 3: The base speaker model. An encoder LSTM takes in three colors, and the decoder LSTM outputs (or scores the likelihood of) a sentence. Figure from [1].

The base-level speaker consists of two LSTMs - an encoder and a decoder. The encoder takes as input the three context colors. The decoder is slightly more complex, as its input depends on the purpose for which S_0 is being used; we will get to it in time. The colors, which are expressed in HSV (in contrast to the listener's RGB), then Fourier transformed to a 54-dimensional representation, are input into the encoder LSTM, one at a time. As in [1], the correct color is input last. The LSTM cell

is the same as it is in L_0. After all three colors are input, we take the last hidden state for use in the decoder. Call this state h.

We will first discuss the overall layout of the decoder. At each time step, the decoder takes in a single word, concatenated with the encoder's final hidden state, h. The embedding matrix for the word is initialized the same as it is in the base listener. The word/state pair is the input to the decoder LSTM (which is unidirectional). The output is then sent through a weight matrix (size 100 x vocab_size, initialized with Xavier initialization), and a bias of size vocab_size is added to this. We add a softmax nonlinearity to this layer to give us a distribution over possible words. When training, we perform cross-entropy loss on the softmax distribution and the one-hot encoding of the actual word for that timestamp. Additionally, we perform masking here, so only words up to the end token contribute to the loss (and, thus, gradient descent).

If we are using S_0 to *score* an input, then at each timestep t, index $t-1$ of the input utterance is the input word. (If $t = 0$, the input is the start token $\langle s \rangle$.) The output of this neural net is the softmax distribution we got before. To score the input sentence, we perform the following: at each timestep, find the probability corresponding the the word occurring at that timestamp. Stop when we see the end token $\langle \backslash s \rangle$. Multiply all of these probabilities together to get the score. This is used in L_1 in order to determine which color of the 3 is the most likely to have produced the given utterance.

If we are using S_0 to *generate* text, then after finishing a timestamp t, take a word from from the distribution generated at the output layer. This word is the input word at timestamp $t+1$. Output the sequence of words generated. This process is used in S_1 to generate possible utterances to produce for each context color.

3.3 Pragmatic Listener

As discussed previously, this model implements the RSA algorithm. In order to implement this, we have an L_2 pragmatic listener and S_1 pragmatic speaker. These are not models, but they make use of S_0 and L_0. S_1 generates 8 utterances for each context color using S_0, giving us 24 utterances in all. These utterances, along with the actual utterance u, are all sent through L_0, which tells us the probability of each possible color for each utterance. We can then calculate the probability distribution of S_1, given the outputs from L_0:

$$S_1(u|t) = \frac{L_0(t|u)^\alpha}{\sum_{u'} L_0(t|u')^\alpha} \tag{2}$$

α allows us to soften the resulting distribution, in a way. It makes options of high probability lower, and options of low probability higher. While this does not affect which output of L_2 is the largest, it is useful when the distribution itself is used, e.g. in ensemble learning. In [1], this value is set to 0.544.

We can calculate S_1 for every color t, telling us the likelihood that the speaker would have uttered u, provided that the actual color was t. This gives us the distribution L_2:

$$L_2(t|u) = \frac{S_1(u|t)}{\sum_{t'} S_1(u|t')} \tag{3}$$

3.4 Ensemble Models

Because L_0, L_1, and L_2 learns colors differently, they might be able to detect certain intricacies in utterances the other listeners cannot. As such, we can blend these models together in order to potentially get higher accuracies. This is done as follows:

$$L_a(t|u) \propto L_0(t|u)^{\beta_a} \cdot L_1(t|u)^{1-\beta_a} \tag{4}$$

$$L_b(t|u) \propto L_1(t|u)^{\beta_b} \cdot L_1(t|u)^{1-\beta_b} \tag{5}$$

$$L_e(t|u) \propto L_a(t|u)^{\gamma} \cdot L_b(t|u)^{1-\gamma} \tag{6}$$

In the original paper, these parameters are determined through a grid search. They find that the best results are $\beta_a = 0.492$, $\beta_b = -0.15$, and $\gamma = 0.491$. Our model results, however, are a little different, and we find that different parameters work better for us here. Through direct variable manipulation, we found that the parameters that work best are $\beta_a = 0.465$, $\beta_b = 0.2$, and $\gamma = 1$.

Additionally, we find that a model blending L_1 and L_2 may prove useful. We define the following two models:

$$L_c(t|u) \propto L_1(t|u)^{\delta} \cdot L_2(t|u)^{1-\delta} \tag{7}$$

$$L_{e2}(t|u) \propto L_a(t|u)^{\epsilon} \cdot L_c(t|u)^{1-\epsilon} \tag{8}$$

$$\tag{9}$$

We find that the best δ is 0.3, and the best ϵ is 0.78.

4 Training

The division of training data is exactly the same as in [1]. Data processing works in the same way, too: for each round in the dataset, all the listener utterances are removed, speaker utterances are concatenated together using the ' ' character, and all words are set to lowercase. Next, the words are tokenized in the following manner: punctuation is split off, though words with hyphens or apostrophes are kept together. Sequences of dots or asterisks are also kept as single tokens. Further, numbers (which may have a '.' or '/', as well as a +/- sign) are single tokens. (Note that this specific tokenizer was taken from the codebase of [1].) When preprocessing data for the listener, we additionally split off the following word endings: "er", "ish", and "est". These tokens are prepended with a "+" character. This process is not performed when preprocessing data for the speaker, since the speaker must be able to produce full words - it should not be able to randomly say "+ish". When preprocessing the training set, we also create a dictionary, mapping words to integers. This allows us to encode our input as a sequence of numbers. However, for any word that appears only once in the dataset, we replace it with "⟨unk⟩". When using this dictionary to map words to integers in other datasets, we replace any word that is not in the dictionary with "⟨unk⟩".

In order to actually train the model, we perform the following. For the base listener, we take the cross-entropy loss, and apply the ADADELTA gradient optimizer [6] with a learning rate of 0.2. In the base speaker, we use the Adam optimizer [7] with a learning rate of 0.004. Using a batch size of 128, we train L_0, keeping track of the accuracy (defined as the number of times for which $L_0(t|u)$ is the highest for the correct color t) on the development dataset. Whenever we reach a new highest accuracy, we save the current weights. We stop training when it is clear that the model is overfitting, and we can see that the accuracy on the dev set will not be able to achieve the same results it did earlier.

Training the base speaker is a little different. Because it is difficult to determine accuracy of a text generator, we define a listener L_1 as follows:

$$L_1(t|u) = \frac{S_0(u|t)}{\sum_{t'} S_0(u|t')} \tag{10}$$

Like with L_0, the color we choose is whichever color gets the highest score here - in other words, the color which is most likely to have produced the utterance u. Accuracy is calculated in the same manner, and training continues until the model starts to overfit and accuracy begins to slip downwards. (We have experimented on minimizing loss on the dev set, and we find that the maximum accuracy coincides with the mimimum loss.) While the paper uses a batch size of 128 for training, we actually find that a batch size of 32 produces far better results, and so this is the batch size that we apply.

5 Results and Analysis

The highest accuracy that we were able to achieve on the development set was 84.42%. This was from utilizing L_{e2}, our model blending L_a and L_c. On the test set, we were able to get 86.03%. This increase would simply appear to be a natural difference in the level of difficulty in the datasets, seeing as humans perform better on the test set. See Figure 4 for more information.

Unfortunately, we were not able to achieve the same results as the original paper. For one, we were not able to match their accuracy on the base listener, L_0 - they scored 83.3%, while we got 82.34%. It is likely that this difference affected many other results, as the pragmatic listener relies heavily on the base listener, as do the ensemble learners and their precise parameters. We are currently

Model	Dev Accuracy (%)	Perplexity
L_0	82.34	1.57
L_1	82.06	1.53
L_2	82.36	1.56
L_a	84.39	1.46
L_b	82.40	1.54
L_c	84.40	1.49
L_e	84.39	1.46
L_{e2}	84.42	1.46
Human	90.40	

Model	Test Accuracy (%)	Perplexity
L_0	84.31	1.50
L_1	83.61	1.48
L_2	84.47	1.50
L_a	86.01	1.41
L_b	84.58	1.48
L_c	85.99	1.44
L_e	86.01	1.41
L_{e2}	86.03	1.41
Human	91.08	

Figure 4: The results for both the development dataset and test dataset.

Model	Dev Accuracy (%)	Perplexity
L_0	83.30	1.73
L_1	80.51	1.59
L_2	83.95	1.51
L_a	84.72	1.47
L_b	83.98	1.50
L_e	84.84	1.45
Human	90.40	

Model	Test Accuracy (%)	Perplexity
L_0	85.08	1.62
L_e	86.98	1.39
Human	91.08	

Figure 5: The results obtained by Monroe et al. [1].

unsure what differences in our models create this difference (if we did, we would have fixed them by now). We were able to make multiple improvements to our score by searching through the code of the original model, ensuring that every detail, such as weight initializations and LSTM cells, were implemented in precisely the same manner. (For example, L_0 is bidirectional - a fact not mentioned in the original paper.) Even so, there remains a single percentage point of difference that we were unable to eliminate. One notable difference in our results, however, is in the difference of perplexities. Our L_0 has a perplexity of 1.57, while perplexity of their model is 1.73. This indicates that while their model tends to be more accurate, the probability distributions that they output tend to lean less strongly towards the correct color. Perhaps if we were to find a way to relax our output distributions, our model might be more accurate.

One major improvement on our end is in L_1, the listener for the base speaker. Monroe et al. get 80.51% accuracy, while we are able to attain an accuracy of 82.06%. As stated previously, we made sure that every detail in the logic of our models matched, and as such, we are unsure as to what led to this difference. In their paper, Monroe et al. describe the base speaker as simply being a generator, which always uses the word generated as an input to the next timestep. However, in the context of L_1, we use S_0 to score our utterances - in other words, we want to find the probability that the correct word was output at each timestep; in the next timestep, we always input the correct word. This process is not described in the paper, but it appears to be done in their codebase. It was also the approach which I initially found to be more intuitive - if we start outputting incorrect words, then the probability of the correct sentence will be even lower, since later words depend on earlier words. This actually may have contributed to the S_0 generator leaning towards putting color words at the beginning of utterances - since the beginning tends to direct the rest of the utterance, examples that started with colors likely tended to score higher. (This may just be a result of a large number of 1 or 2 word utterances, however.) One major departure from the original paper in training S_0 is the batch size. Monroe et al. use a batch size of 128, but we use a batch size of 32. This ends up increasing our accuracy by a full percentage point (from around 81%), though even without the decreased batch size, our accuracy is still significantly higher than theirs. It is the difference in L_1 accuracies, I believe, that allows us to achieve 84.39% on the development set using L_a. The difference in increase between L_0 and L_a for Monroe et al. is 1.42%, while for us, it is 2.05%.

One issue on our end was that we were not able to get the increase from L_0 to L_2 that we expected, since it was present in [1]. This was even more surprising, since our L_1 score meant that our S_0 modeled speech better than theirs did. However, this difference may be what caused our L_2 model to perform poorly. S_0 might be good at modeling, but perhaps its speech generation skills may not

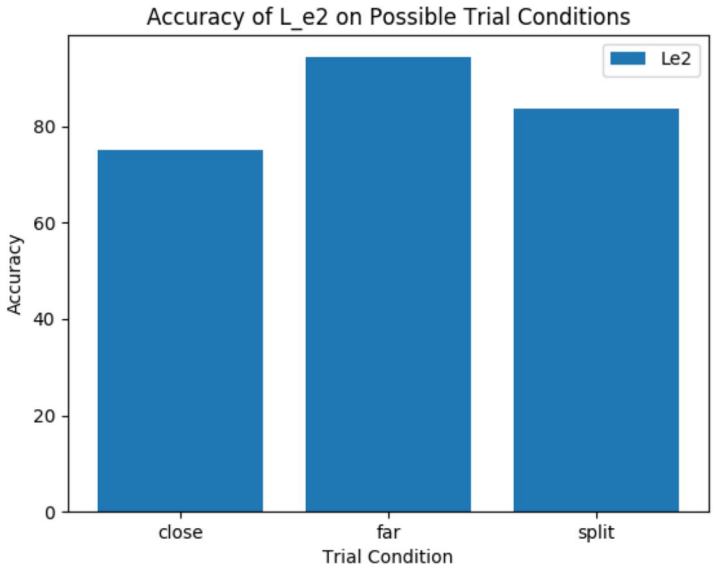

Figure 6: Accuracy of L_{e2} on the close, split, and far divisions of the development dataset.

be as good as those of S_0 from [1]. This also may be due to L_0 not being able to recognize speech generated from a neural net in some cases. However, since neither of our base listeners were trained on this type of data, this may not be the key difference. Rather, the difference in how our S_0 models act seems like the most likely suspect at the moment. This issue, I believe, led L_2 giving us no new information, meaning that L_b, the blend of L_0 and L_2, did not help us that much. As a result, the maximum value we were able to achieve with L_e was equivalent to that of L_a, since including information from L_b only lowered our accuracy. The introduction of L_c and L_{e2}, however, was able to help us, if only marginally. This might be because our L_2 was mostly equivalent to L_0, so we gain very little new information from using it. If we had achieved a significant increase in the score of L_2, then L_c may have performed far better than it did here. Even a marginal increase in accuracy, however, is nice to see.

One interesting area that we might look at is differences in close, split, and far trials. *Far* trials are cases in which all colors are different from each other, *split* trials have the target color as similar to one of the incorrect colors but different from the other, and *close* trials have all three colors similar to each other. As one would expect, the accuracy of the model applied to far trials far exceeds that of split trials, which exceeds that of close trials. But it might be interesting to compare the results from our listeners to those of Monroe et al. On the development set, our L_{e2} model gets 94.34% accuracy on the far set, 83.79% on the split set, and 74.96% on the close set. This can be seen in Figure 6. Monroe et al., on the other hand, using their L_e model, get about 94% on the far set, 84% on the split set, and 77% on the close set. While our far sets and split sets have similar accuracies, the most glaring difference is in our close sets. It would appear that their ability to handle close cases is their strength, and gives them the better overall accuracy. This might be based in the base learner, however. Their base learner gets a score of about 75% accuracy on the close set, while our base learner's accuracy on that set is 72.75%. If we are to improve our overall score, then, we may need to focus on learning to solve close sets with the base learner.

6 Conclusion

While we were not able to exactly reproduce the results of [1], we were able to get reasonably close, and made some interesting improvements on the side. The pragmatic listener, L_2, did not end up being very useful to us (against our expectations), but the L_1 listener ended up being very helpful (again, against our expectations). It would seem that by learning from data in a different manner, we can learn intricacies that might not be present when learning data in the normal matter (e.g. text

to color). Extending this concept to other machine learning problems may be an interesting concept for future study.

References

[1] Will Monroe, Robert XD Hawkins, Noah D Goodman, and Christopher Potts. Colors in context: A pragmatic neural model for grounded language understanding. *arXiv preprint arXiv:1703.10186*, 2017.

[2] Michael C Frank and Noah D Goodman. Predicting pragmatic reasoning in language games. *Science*, 336(6084):998–998, 2012.

[3] Noah D Goodman and Andreas Stuhlmüller. Knowledge and implicature: Modeling language understanding as social cognition. *Topics in cognitive science*, 5(1):173–184, 2013.

[4] Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[5] Will Monroe, Noah D Goodman, and Christopher Potts. Learning to generate compositional color descriptions. *arXiv preprint arXiv:1606.03821*, 2016.

[6] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

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