Final Report - SS4850

Emiliano Penaloza
Anthony Rinaldi

April 19, 2022

Table of Contents

1 Introduction

The way in which most people/agents negotiate and come to a decision based on differing preferences and goals is through the use of natural language. This is a complex problem because one must determine their intent, find a way to communicate this to another agent, and understand the intent of the other agent. What makes this scenario even more complex is the use of deceit, which is difficult to ascertain, but also very common in human negotiations [6]. Negotiation dialogues can take on many different forms depending on the agents involved (e.g., cooperative or aggressive) and each agent must come up with *utterances* (sequences of words) to help them achieve their goal in the negotiation.

We take a similar approach to Lewis et al. [5] using a very large one-on-one negotiation dataset to train neural negotiation agents. These agents learn to negotiate by maximizing the likelihood of copying human actions and are further refined by using reinforcement learning (RL) when negotiating against one another. Reinforcement learning forces the agent to maximize their reward, rather than simply copying human actions; possibly giving the agents skills that never would be exhibited by human participants. We supplement these models with the use of the *Monte Carlo tree search* (MCTS) algorithm to estimate the expected reward of different utterances. MCTS has been widely successful in strategic games which is why we expect it to perform well in strategic negotiations [2].

We study these negotiations by using a dataset of semi-cooperative dialogues between participants. Participants are shown a set of objects (books, balls, and hats), each with a specific value and are tasked with maximizing their value by determining how to divide the objects between themselves and another participant. Participants are unaware of the value function of the other agent.

The remainder of the paper is organized as follows: §2 gives an overview of the negotiation dataset used in the paper. §3 introduces both the exploratory and main methodology of the paper. §4 describes the main results of the paper. §5 concludes the paper with discussion about the implications of the results and §6 proposes future works.

2 Dataset

The dataset consists of 5,808 dialogues collected using Amazon's Mechanical Turk, a crowdsourcing service that gathers freelance workers to complete on-demand tasks that computers are unable to do. Each dialogue consists of two human agents engaging in a negotiation aiming to maximize their total reward. The two agents negotiate over 5-7 total items each of which belongs to one of three categories: hats, balls and books. The negotiation begins by providing each agent with a different randomly generated value function where the total value of all items is 10 for each agent and no item has a zero reward for both users. The agents then engage in conversation until one of them declares that an agreement has been made which marks the end of the negotiation. Thereafter, each agent declares the agreed-upon number of items assigned to each agent, if both agents agree on the decision, the appropriate rewards are assigned, if they disagree, they are both assigned a total reward of zero. This methodology of gathering dialogues yielded 2,236 different scenarios (unique value functions and number of items). An example scenario and the corresponding dialogue can be seen in Table 1 and Figure 1, respectively. As previously mentioned, the data is publicly available in the following repository: https://github.com/facebookresearch/end-to-end-negotiator.

	Hat	Ball	Book
Amount of items in pool	1	2	3
Agent 1 value function	4	0	2
Agent 2 value function	1	3	1

Table 1: Example Scenario

Dialogue:

Agent 1: I want the books and the hats, you get the ball
Agent 2: Give me a book too and we have a deal
Agent 1: Ok, deal
Agent 2: <choose>

Figure 1: Example Dialogue

3 Methods

3.1 Exploratory Data Analysis

The data used in this report is quite unique and traditional data analysis methods will not be very useful. Most of the data are English sentences, which we cannot easily visualize. One approach we will use is plotting the frequency of words used, however, this is not used in modelling and will only be used to understand the language in the dialogues. The other visualizations we will consider involve the score of each agent and their value functions. We will plot the distribution of the final score for the player negotiating first and the player negotiating second. We will view the number of Pareto optimal outcomes.[1] We will view the distribution of number of turns and number of words per turn across all negotiations. Finally, we will consider the relationship between some of these variables, such as the score of a game versus the number of turns in a game. Not all of the exploratory data analysis will be useful in this report, thus we will only include significant findings in the final report.

3.2 Existing Methods

Lewis et al. [5] proposed four model architectures to build a conversational negotiation agent. They first propose a model that is trained by optimizing the log-likelihood of the predicted token plus the log-likelihood of the output choice. Thereafter, they propose a model that is, first trained as the previous log-likelihood model, but then fine-tuned by selecting the utterance with the maximum expected reward. Finally, they proposed utterance rollouts for action selection, which can be combined with either of the stated models. We propose to improve these models by using *Monte Carlo Tree Search* (MCTS) and *Transformers*. We propose eight new methods.

3.3 Transformers vs GRUs

In the original work, Lewis et al. [5] propose a GRU (Gated Recurrent Unit) model architecture. We propose using Transformers in preference to GRUs. Given the large amount of empirical evidence showing that Transformers outperform GRU-based architectures [4], it is an intuitive suggestion. Transformers, initially proposed by Vaswani et al. [7] are neural network architectures that aim to replace traditional convolutional and recurrent networks. They have been shown to outperform said networks in language and vision tasks by exploiting multi-head self-attention. Unlike traditional recurrent or convolutional layers, through the use of multi-head attention, transformers are able to predict the output sequence by attending to the most relevant previous information.

[1] A solution is Pareto optimal if neither agent's score can be improved without lowering the other's score.

Lewis et al. [5] GRU Architecture	**Proposed Transformer Architecture**
$h^g = GRU_g(g)$	$h^g = GRU_g(g)$
$h_t = GRU_w(h_{t-1}, [Ex_{t-1}, h^g])$	$h_t = GRU_w(h_{t-1}, [Ex_{t-1}, h^g])$
$h_t^{o\leftrightarrows} = GRU_{o\leftrightarrows}(h_{t\pm1}^{o\leftrightarrows}, [Ex_t, h_t])$	$h^o = Trans_o([Ex, h_t])$
$h_t^a = W[tanh(W' h_t^{o\leftrightarrows})]$	$h^a = W[tanh(W' h^o)]$
$\alpha_t = \frac{exp(w \cdot h_t^a)}{\sum_t exp(w \cdot h_t^a)}$	$\alpha_t = \frac{exp(w \cdot h^a[t])}{exp(w \cdot h^a)}$
$h^s = tanh(W^s[h^g, \sum_t \alpha_t h_t])$	$h^s = tanh(W^s[h^g, \sum_t \alpha_t h_t)$

Table 2: Model Architectures

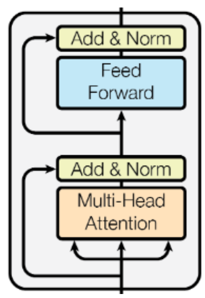


Figure 2: Transformer Encoder Architecture

This method not only, on average, yields improved performance but allows for parallelization when training the network. This is because each *head* in the layer is summative rather than a product, allowing for faster gradient calculations. For a comparison between our proposed model architecture and the one proposed by Lewis et al. [5] please refer to Table 2. We expect that by introducing a Transformer, we will not only achieve a higher average score but produce more human-like utterances.

We will not be using a complete Transformer architecture, rather will be using just a single encoder layer. The network layout for the encoder can be seen in Figure 2. This architecture includes the powerful multi-head attention, along with a feed-forward layer, making this attention global. This means that after attention scores are calculated, a fully connected layer is used so that every point in the sequence can be connected to one another (i.e., global). We do not use the decoder part of the model because this is typically used for sentence-to-sentence modelling. Since our approach takes in a complete dialogue and generates the agent's item choices, there is no sequence-to-sequence modelling so the decoder is not necessary. The code used to create the transformer model and make predictions with it can be found in Code 11.

3.4 Monte Carlo Tree Search

This algorithm is similar to the rollouts algorithm that Lewis et al. [5] propose in the report. The key difference is that MCTS builds a game tree and simulates many rollouts to determine the optimal action given the current state/node. Once the actions are proposed, only those that are the most successful given the current state are expanded upon and simulated (rolled out) further. Since the algorithm uses an *Upper Confidence Bound* (UCB) to weight which node to expand, it is able to explore a much larger state space than a traditional rollout algorithm. Additionally, MCTS explores a larger state space in a more computationally efficient matter. Since MCTS does not require expansion of every node but uses UCB to weight its options, it only explores the most promising nodes, marginally reducing the computational cost. Yet even with the reduced expansion, MCTS still achieves higher performance by searching into a deeper state space. Due to this, we expect MCTS to out-perform the traditional rollout method, and produce more relevant and fluent utterances. The MCTS description can be found in Algorithm 1.

The UCB formula is shown below in equation (1). The UCB gives us a number that represents the trade-off between exploring (trying new actions) and exploiting (reusing actions that we know have a good result). In the equation, s_i represents the total score of node i, n_i represents the number of times node i has been visited, C is a constant that we set to its typical value of 2, and N_i represents the number of times the parent of node i has been visited.

$$UCB_i = \frac{s_i}{n_i} + C \cdot \sqrt{\frac{\log N_i}{n_i}} \tag{1}$$

The class we implement for MCTS can be found in Code 12. This class has the ability to understand its opponent's turn, and thereafter perform MCTS to determine its best response. After the opponent gives their response the class will generate five unique possible responses. From there, it will iterate through 50 simulations of the MCTS algorithm, determining which of the five responses will result in the highest score. It then chooses the response that results in the highest score as its next action.

3.5 Proposed Models

We propose eight additional models expanding upon Lewis et al.'s [5] initial four. These models will be evaluated on the same dataset as the original four. In order to be computationally efficient, all hyperparameters will be selected using a validation dataset similar to Lewis et al. [5].

1. Supervised model with probabilistic sampling with addition of transformers
2. Supervised model, rollout dialogue (likelihood evaluation) with addition of transformers
3. RL, likelihood dialogue (one-step expected reward) with addition of transformers
4. RL, rollout dialogue (expected reward) with addition of transformers
5. Supervised model, MCTS dialogue (likelihood evaluation) with addition of transformers
6. Supervised-model, MCTS dialogue (likelihood evaluation) base GRUs
7. RL, MCTS dialogue (expected reward) base GRUs
8. RL, MCTS dialogue (expected reward) with addition of transformers

All hyper-parameters; number of layers, layer depth, discount factor, output importance α, learning rate, and batch size were determined using cross-validation on a smaller validation dataset.

Algorithm 1: MCTS For Negotiation Dialogue

Result: Get the next best word based on Monte Carlo simulation
Input : Dialogue History (i.e., $x_{1,\ldots,k}$), items, value function, number of iterations
Output: Utterance to finish turn (i.e., $x_{k+1,\ldots,k+n}$)

```
Initialize the root node of the game tree
while current iteration < number of iterations do
    Traverse the tree downwards until reaching a leaf node, selecting subsequent nodes that
      have the largest UCF score
    if node has been visited before then
        Expand the node to have children (each child is a dialogue for a full turn)
        Select the first child as the current node
    else
        Select the leaf node as the current node
    end
    Rollout to the end of the dialogue on the current node by sampling the likelihood function
      of tokens
    Make selections for the items based on the dialogue that occurs during the rollout
    Calculate the score of the game given the value function of the player
    Backpropogate the results:
        Add the score of the game recursively to every parent node until reaching the root node
        Increment the times visited variable of every parent recursively until reaching the root
          node
    Increment current iteration
end
Return the utterance with the highest average score
```

3.6 Training Setting

Model training will consist of supervised learning based on a training set of recorded dialogues and RL against another fitted model. This will give us two models: (1) supervised and (2) supervised + RL. With these models, we can apply rollouts and MCTS when competing against another player to see how the models perform. Supervised training takes the typical steps of deep learning: pass a batch through the network to calculate the output, calculate a loss function, backpropagate the error, calculate the new parameter estimates, and repeat. We found poor model performance using the same hyperparameters suggested by Lewis et al. [5], so implemented a Bayesian parameter tuning approach. This uses the `optuna` package in `Python` to determine the best parameters given priors and the observation likelihood. The code for the hyperparameter tuning can be found in Code 13. Thereafter we fit model (1) using the entire training dataset and the optimal hyperparameters. We then perform RL on model (1) to generate model (2). This is done by allowing model (1) to play a version of itself and work on optimizing the reward it receives from the negotiation.[2] For parameter stability, only one version of the model is trained with RL at a time, while the other is held constant. Further, we intertwine supervised training into the process as well so that the model does not deviate from the English language. We do so after every four iterations of RL, with a small learning rate to not take away from the RL process. The code for

[2]Code 14 contains the code that generates a dialogue between two models, which is used in this process.

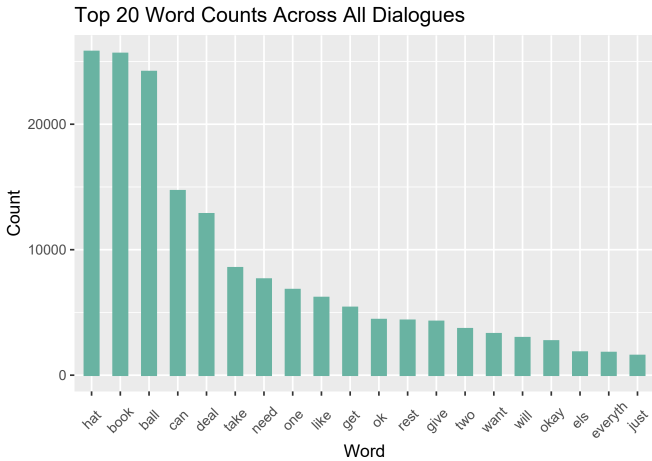


Figure 3: Top 20 Words and their Count

The figure shows the most common words used across both players throughout all negotiation dialogues.

the RL training process can be found in Code 15. The code for the agent/player that trains itself by playing in the mentioned RL training process can be found in Code 16.

3.7 Testing Setting

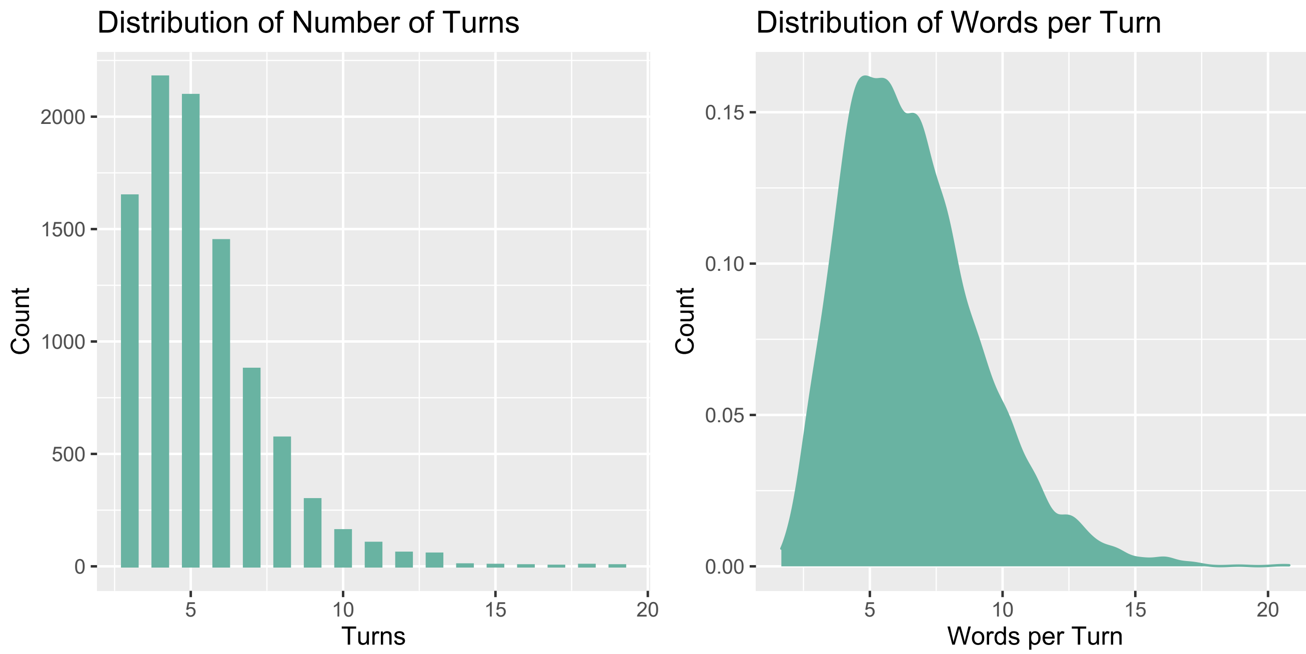
Testing the models is very different than a typical supervised learning approach (e.g., classifier accuracy). If we were to evaluate our models by how well they perform on the dialogues we have, this would simply give a measure of similarity between our model's predictions and a human's sentences. However, we do not wish for our model to simply imitate humans, we want it to understand how to negotiate and maximize its reward. This gives it more of an RL evaluation than a supervised learning evaluation. To evaluate each model's performance we put it head-to-head against the most basic model we have: the supervised Transformer model. These models are given 8,172 different scenarios (i.e., items and value functions) and engage in dialogue. The scores are recorded, along with if there was agreement, and if the outcome is Pareto optimal. These scores are then aggregated across all scenarios to see how the proposed model performs relative to a baseline model. Here we can also choose to apply rollouts, MCTS, or neither to supplement the model that is undergoing evaluation. This is the same method that Lewis et al. [5] used to evaluate their models.

4 Results

4.1 Exploratory Data Analysis

We first look at the most common words used in the dialogues. This helps us determine the level of sophistication of the negotiations. Figure 3 shows the top 20 words used in the dialogues along with their respective word count (see Code 7 for the `R` code). Quite logically we see the top three words are the items that the players are trying to divide. The remaining words all seem typical of a negotiation dialogue with words like *need* and *deal*. The level of sophistication of these dialogues is quite low since we see no complex words; it is all very simple English. This means the model should be able to learn how to negotiate quite easily since the complexity of the language is low.

To understand the level of sophistication further, we view the distribution of the number of turns and the number of words used per turn. Figure 4 shows that most negotiations terminate in less than the maximum 20 turns allowed (see Code 8 for the R code). The majority of negotiations



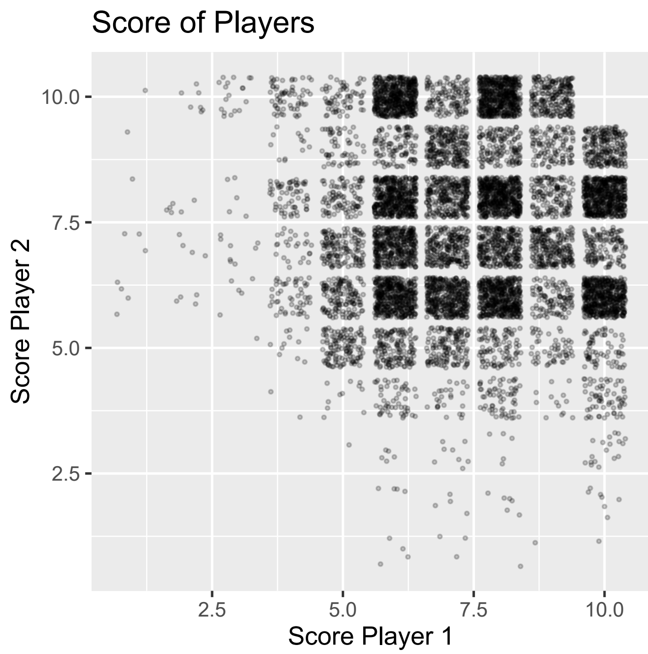
Left: number of turns taken to complete a negotiation. **Right:** average number of words used per turn in a dialogue.

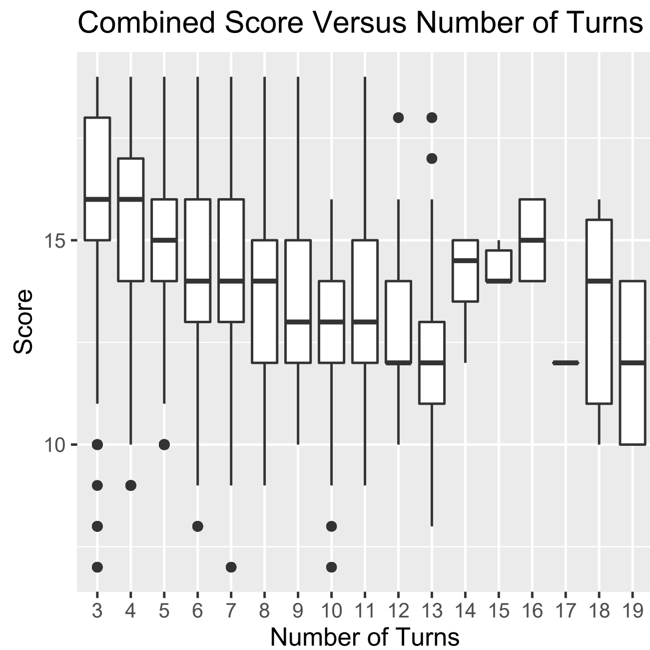
Figure 4: Distribution of Turns and Words per Turn

finish in less than seven turns. This supports the previous conclusions that the negotiations cannot be extremely complex if they only take a few turns to arrive at an agreement. Further, Figure 4 also shows that there are very few words used per turn. With less than 10 words used per turn in most cases, it is unlikely that these negotiations are highly sophisticated. This supports the previous conclusion and indicates that the model should be able to learn the dialogues relatively easily.

Next, we try and understand the score of the agents better. First, we compare the score of each player in Figure 5 where we added *jitter* to the points so we can see the density at each score combination (see Code 9 for the R code). Players generally do not need to sacrifice their score for the other player to achieve a higher score. The plot shows that it is possible for both players to achieve a high score simultaneously (although not once did both players score a perfect 10). The most common scores are when both players score between six and eight. We also try and understand if the score of the game will increase as there are more turns. We expect this relationship since the players can better communicate their needs with more turns and get higher scores. The right side of Figure 5 shows the combined scores of both players versus the length of the dialogue. We actually see the inverse relationship than we expected; as there are more turns, the average combined score is lower. This could possibly be explained by irrational human behaviour such as the *endowment effect* where people think that the things they perceive to own are worth more and are less willing to give them up [3]. This result is counterintuitive and has implications for our model. We may seek to develop a model that attempts to end the negotiation as quickly as possible since that seems to produce better results.

Finally, we compare the score of a player to their positions in the dialogue (i.e., first or second). We expect that the player who goes first will have a higher score since they can set a reference point in the dialogue that all discussions will be based around (known as an anchoring point in





Left: comparison of scores of each player. **Right:** how scores vary as the length of dialogue varies.

Figure 5: Score Distributions

behavioural economics [1]). Figure 6 shows that the first player performs better in most cases, having a higher chance of achieving a higher score (see Code 10 for the R code). Further, the first player tends to achieve a perfect score at twice the rate of the second player. This finding agrees with our prior understanding of one-on-one negotiations.

4.2 Main Data Analysis

We carry out the training and testing steps mentioned in §3. This results in four models: (1) Transformer, (2) Transformer + RL, (3) Base GRU, and (4) Base GRU + RL. The baseline model we will use as our *default* agent to negotiate against other models is model (1). The default agent is compared to both model (1) and model (2) while using rollouts, MCTS, or no modifications on the latter two models. This gives us six groups of dialogues to analyze. Thereafter we compare model (3) and model (4), both with MCTS, to the default agent. This gives an additional two groups of dialogues to analyze, giving eight groups in total. These eight groups correspond to the eight new models we proposed, each compared to the default agent.

These groups of dialogues are individually analyzed to get the average score of each agent, the average score of each agent conditioned that the agents agreed, the agreement rate, and the proportion of Pareto optimal outcomes. These results are presented Table 3.

The model that performed the best in almost all metrics is the Transformer model that uses rollouts when competing against other agents. This is an unexpected result since we thought the best model would utilize MCTS given the algorithm's success in other domains.

We note that the RL is not successful at all and seems to be working in favour of the opponent instead of itself. Comparing any model without RL to its equivalent with RL shows that there is a significant score decrease. This is quite strange and we propose some reasons for this phenomenon

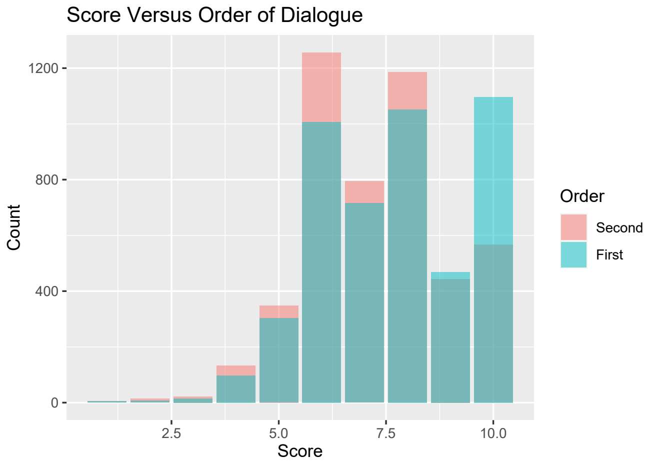


Figure 6: Distribution of Score versus Starting Player

This chart compares the number of times a player received a certain score based on if they were the first player to initiate the dialogue or the second player.

shortly. Also note that for all the Transformer models, the addition of RL increases the agreement rate significantly. It seems that the model is choosing to agree easier, at the expense of its score. This is a unique result but not the intention of using RL.

A possible flaw in the RL version of the model is the lack of hyperparameter tuning. Hyperparameters are just as important as in the supervised case but training takes much longer, making it difficult to tune. Each supervised model takes around 10-20 minutes to train, whereas an RL model takes around three hours. It is not feasible to train many RL models with different hyperparameters and assess which is the best. With more powerful devices, we may have been able to tune the parameters better for RL and get a better performing RL model, but we, unfortunately, did not have access to powerful computing resources.

We also note that all the agreement rates are much higher than those of the models proposed by Lewis et al. [5]. Since the part of the model responsible for generating sentences (encoder and decoder) is the same in our architecture and theirs, this difference can be attributed to the part of the model that makes the item selection. We can think of this as every dialogue being the exact same between our model and their model, but the item selections are the only difference. The difference in agreement rates is only attributable to the model being able to understand the dialogue and come up with the right item selection numbers. The Transformer is better able to predict the item selections given the dialogue compared to the GRU since it has a significantly higher agreement rate.

5 Conclusion and Discussion

We find that our model works well in dialogues and produces real English sentences. Along with generating coherent sentences, the model also negotiates quite well and can achieve a high score while not sacrificing the opponent's score. There is quite a high ratio of Pareto optimal outcomes, indicating the negotiations are equally beneficial for both parties, which is a preferred outcome. The model is also very agreeable, improving the rate greatly compared to previous approaches.

The second major finding is that MCTS does not provide a significant benefit over rollouts. Comparing the TRANSFORMER + ROLLOUTS to TRANSFORMER + MCTS we see that the rollouts perform better in most cases. We hypothesize that this is the case because the length of the game is quite short, averaging around seven turns per dialogue. MCTS was proposed to sample a large and deep action space as a way to approximate the value of a node in a dense game tree. Since these

Model	vs. LIKELIHOOD Score (all)	Score (agreed)	% Agreed	% Pareto Optimal
TRANSFORMER LIKELIHOOD	4.9 vs. 4.8	5.2 vs. 5.2	93.9	30.2
TRANSFORMER + ROLLOUTS	**6.5 vs. 5.0**	**6.8 vs. 5.2**	95.9	**42.0**
TRANSFORMER RL	3.7 vs. 6.6	3.7 vs. 6.5	98.9	36.6
TRANSFORMER + RL + ROLLOUTS	4.0 vs. 6.5	4.1 vs. 6.6	**99.1**	39.4
TRANSFORMER + MCTS	6.0 vs. 4.8	6.4 vs. 5.1	94.2	37.1
BASE RNN + MCTS	6.0 vs. 4.9	6.3 vs. 5.2	95.1	39.3
BASE RNN + RL + MCTS	6.0 vs. 5.0	6.3 vs. 5.2	94.9	40.8
TRANSFORMER + RL + MCTS	4.0 vs. 6.5	4.0 vs. 6.6	98.6	39.1

Table 3: Model Performance

dialogues are short the game tree is not too dense, meaning there are not many actions to sample from. Compared to rollouts there isn't much of a difference because rolling out a few candidates will cover a majority of the possible actions, making it more equivalent to MCTS.

MCTS is also meant to be used in scenarios where there are many potential (equally likely) moves. In our scenario, the dialogue for a given agent is essentially unimodal given the previous turns in the negotiation. This means that there are not many possible actions, rather there is one very likely action, with all other actions being very unlikely. This makes the dialogue somewhat deterministic. If we use rollouts, we will be sampling the most likely actions and this will inform us more about the likely outcome, compared to using MCTS and forcing the model to consider different actions when they are not all likely. MCTS considers many unlikely responses, meaning it is collecting unnecessary information about the action space and potentially misinforming itself about the most rewarding action.

Since we choose to sample the distribution of actions/responses to determine possible next moves, we are not actually considering all possible next turns. A possible turn would include any combination of any amount of the words in the game dictionary, meaning the possible action space would be massive. Even if combinations of words are not actual English, we should still consider them by MCTS theory, but we do not do this. We limit our game tree to have only the most probable combination of words, given the probability distribution. As mentioned previously, since the model is unimodal and almost deterministic there are not many possible actions to consider from the probability distribution. We found it almost impossible for the model to create five possible responses, given the previous turns, by randomly sampling the action space. Generating possible responses at random typically resulted in the same one or two responses being repeated over and over. This weakens MCTS since there are not many possible actions and makes it more similar to performing a rollout on a few different candidate responses (i.e., what Lewis et al. [5] implemented).

In Table 3 we also see that the TRANSFORMER + MCTS slightly outperforms the BASE RNN + MCTS in the agreed score. This shows that the Transformer has only slightly better performance than the original GRU network, but not as much of an improvement as we were hoping for. The two also have similar agree rates, and the BASE RNN + MCTS has slightly higher Pareto optimality likely because it is willing to sacrifice its score to benefit its partner's score. This may be preferred in some cases, but here we hope to gain as much reward as possible, making the Transformer model preferred.

6 Future Works

The main adjustment we hope to make in this area in the future is using the Transformer model for understanding the dialogue better. Currently, we use the Transformer to generate the output selection, given the dialogue output from the GRU. We expected an improvement given the inherent sequence-to-sequence nature of the problem. Given that the dialogue generation is also a sequence-to-sequence problem, we believe the transition from a GRU to a Transformer would be beneficial. Further, one can also use Long Short-Term Memory (LSTM) cells instead of GRU cells. LSTMs have one more internal *gate* than a GRU unit, and thus tend to work better with longer sequences. The use of LSTMs or a Transformer might result in longer and more sophisticated dialogue than with GRUs.

We also suggest a different use for the MCTS algorithm be attempted in the future. Since the number of possible full-turn dialogues is limited and almost deterministic given the previous turns, we suggest performing MCTS with individual words, rather than with entire turns. This will generate many more possible turns for the model to consider, giving it a better evaluation of the action space. This will also increase the computational complexity though since the number of possible words to use at any point is quite large (above 400). We suggest limiting the scope of words to the 20 most likely words, rather than using the entire dictionary.

A final task we hope to take on in the future is comparing the model to humans. This is a real test of performance and will show if our model can negotiate better than humans can. We will create a web application for this paper, allowing people to compete against the different models we proposed. We will alternate the model that people play against and aggregate the scores for each model to see how well they perform against real people.

7 Appendix

The `Python` code used in this report is extensive so cannot be entirely included here. We include the most important `R` and `Python` code here and refer the reader here for the entirety of the code.

List of Code

Code 1: Import Packages

```r
library(knitr)
library(kableExtra)
library(ggplot2)
library(tm)
library(dplyr)
library(tidytext)
library(grid)
library(gridExtra)
```

Code 2: Load Data

```r
raw_data = scan("data.txt", what = character(), sep = "\n")
```

Code 3: Data Scraping Functions

```r
# function to get the score of a given dialogue
get_score = function(values, selections, agree){
  if (! agree) {return(0)}
  return(values %*% selections)
}

# function to get the following information from a dialogue:
#   number of items
#   value function for each player
#   Boolean variable indicating the players came to an agreement
#   the division of items between players
#   the score of each player
```

```
#    the words in the dialogue
#    Boolean variable indicating the player had the first turn
#
# the function also checks for disconnected sessions and sessions where
# the players agreed to disagree
get_info = function(row){
  split_vec = strsplit(row, " ")[[1]]
  # get the number of items
  item0 = split_vec[1]
  item1 = split_vec[3]
  item2 = split_vec[5]
  # get the value function
  value0 = split_vec[2]
  value1 = split_vec[4]
  value2 = split_vec[6]

  n = length(split_vec)
  # get the other player's value function
  othervalue0 = split_vec[n-4]
  othervalue1 = split_vec[n-2]
  othervalue2 = split_vec[n]

  # check if the players agreed
  agree = split_vec[n-6] == "agree"

  # check if the players agreed to disagree
  no_agree = regmatches(row, regexpr("(?<=<selection>)(.*)(?=<eos>)", row, perl
    =T)) == " no agreement "
  # check if the players disconnected
  disconnect = regmatches(row, regexpr("(?<=<selection>)(.*)(?=<eos>)", row,
    perl=T)) == " disconnect "

  if (no_agree | disconnect){
    # give 0 score and no selection for disagreement and disconnection
    selection = NA
    score = 0
    other_score = 0
    agree = FALSE
  } else {
    # get the selection of player 1
    selection0 = strsplit(regmatches(row, regexpr("item0=[0-9]+", row)), "=")
    [[1]][2]
    selection1 = strsplit(regmatches(row, regexpr("item1=[0-9]+", row)), "=")
    [[1]][2]
    selection2 = strsplit(regmatches(row, regexpr("item2=[0-9]+", row)), "=")
    [[1]][2]

    selection = as.numeric(c(selection0, selection1, selection2))

    # get the score of player 1
    score = get_score(
      as.numeric(c(value0,value1,value2)),
      as.numeric(c(selection0, selection1, selection2)),
      agree
    )
    # get the score of player 2
```

```
    other_score = get_score(
      as.numeric(c(othervalue0, othervalue1, othervalue2)),
      as.numeric(c(item0, item1, item2)) - as.numeric(c(selection0, selection1,
    selection2)),
      agree
    )
  }
  # get all the words in the dialogue
  dialogue = regmatches(row, regexpr("(THEM|YOU)(.+)(<selection>)", row))
  # check if player 1 went first or second
  went_first = split_vec[7] == "YOU:"

  return(list(
    item=as.numeric(c(item0, item1, item2)),
    value=as.numeric(c(value0,value1,value2)),
    othervalue=as.numeric(c(othervalue0, othervalue1, othervalue2)),
    agree = agree,
    selection = selection,
    dialogue = dialogue,
    score = score,
    other_score = other_score,
    went_first = went_first
  ))
}
```

Code 4: Dataframe Creation

```
# use the get_info function to turn the dialogues into a dataframe
df = as.data.frame(t(sapply(raw_data, get_info, USE.NAMES = F)))
# format columns to be vectors instead of lists
df$agree = as.logical(df$agree)
df$dialogue = as.character(df$dialogue)
df$score = as.numeric(df$score)
df$other_score = as.numeric(df$other_score)
df$went_first = as.logical(df$went_first)
```

Code 5: Supplemental Functions

```
# get number of turns
get_turns = function(dialogue){
  people = c("YOU:", "THEM:")
  split_vec = strsplit(dialogue, " ")[[1]]
  turns = 0
  for (i in 1:length(split_vec)){
    if (split_vec[i] %in% people) {
      turns = turns + 1
    }
  }
  return(turns)
}

# get the total number of words
get_words = function(dialogue){
  ignore = c("YOU:", "THEM:", ".", "<eos>")
  split_vec = strsplit(dialogue, " ")[[1]]
  words = 0
  for (i in 1:length(split_vec)){
```

```r
    if (! split_vec[i] %in% ignore) {
      words = words + 1
    }
  }
  return(words)
}

# get possible item divisions
get_choices = function(items){
  ind = 0
  res = list()
  for (i in 0:items[1]){
    for (j in 0:items[2]){
      for(k in 0:items[3]){
        ind = ind + 1
        res[[ind]] = c(i, j, k)
      }
    }
  }
  return(res)
}

# get pareto optimal
is_pareto = function(row){
  agree = row$agree
  items = row$item
  score = row$score
  other_score = row$other_score
  values = row$value
  other_values = row$othervalue

  if (! agree) {return(FALSE)}

  choices = get_choices(items)
  # iterate through every possible item division in the game
  for (i in 1:length(choices)){
    # calculate the scores for the current division
    curr_choice = choices[[i]]
    curr_opp_choice = items - curr_choice
    curr_score = get_score(values, curr_choice, T)
    curr_opp_score = get_score(other_values, curr_opp_choice, T)
    # check if both players could be better off
    if ((curr_score > score & curr_opp_score >= other_score) |
        (curr_score >= score & curr_opp_score > other_score)){
      return(FALSE)
    }
  }
  return(TRUE)
}
```

Code 6: Supplement Dataframe

```r
# use new functions to add new columns to the data frame
df$n_turns = sapply(df$dialogue, get_turns, USE.NAMES = F)
df$n_words = sapply(df$dialogue, get_words, USE.NAMES = F)
df$avg_words = df$n_words / df$n_turns
df$pareto = apply(df, 1, is_pareto)
```

Code 7: Word Analysis

```
# use corpus to manipulate the words
corpus = VCorpus(VectorSource(df$dialogue))
corpus = tm_map(corpus, removeNumbers)
corpus = tm_map(corpus, content_transformer(tolower))
# remove stopwords and end of sentence and selection marker
corpus = tm_map(corpus, removeWords, c(stopwords('english'), "eos", "selection"
    ))
corpus = tm_map(corpus, removePunctuation)
corpus = tm_map(corpus, stripWhitespace)
corpus = tm_map(corpus, stemDocument)

# turn the cleaned corpus into a dataframe
dialogue_df = data.frame(text = sapply(corpus, as.character), stringsAsFactors=
    F)

# get the count of each word in every dialogue
word_df = dialogue_df %>% unnest_tokens(word, text) %>% count(word, sort = TRUE
    )
word_df$word = factor(word_df$word, levels = word_df$word[order(word_df$n,
    decreasing = T)])

# plot the count of the top 20 words in all dialogues
ggplot(word_df[1:20, ], aes(y=n, x=word)) +
  geom_bar(colour="#69b3a2", fill="#69b3a2", stat = "identity", width=0.5) +
  ggtitle("Top 20 Word Counts Across All Dialogues") +
  xlab("Word") +
  ylab("Count") +
  theme(axis.text.x = element_text(angle = 45, vjust = 0.5))
```

Code 8: Plot of Turns and Words per Turn

```
# create a separate dataframe for plotting
plot_data = df
# convert the went_first column to a factor
plot_data$went_first = factor(plot_data$went_first, labels = c("Second", "First
    "))

# plot the distribution of number of turns
p1 = plot_data[plot_data$agree, ] %>%
  ggplot(aes(x=n_turns)) +
  geom_bar(colour="#69b3a2", fill="#69b3a2", width = 0.5) +
  ggtitle("Distribution of Number of Turns") +
  xlab("Turns") +
  ylab("Count")

# plot the distribution of words per turn
p2 = plot_data[plot_data$agree, ] %>%
  ggplot(aes(x=avg_words)) +
  geom_density(colour="#69b3a2", fill="#69b3a2") +
  ggtitle("Distribution of Words per Turn") +
  xlab("Words per Turn") +
  ylab("Count")

grid.arrange(p1,p2, ncol=2)
```

Code 9: Plot of Score per Player and Score versus Turns

```
# plot the score of each player
# adding scatter so we can see the overlap
p1 = plot_data[plot_data$agree, ] %>%
  ggplot(aes(x=score, y=other_score)) +
  geom_jitter(alpha = 0.2, cex = 0.5) +
  ggtitle("Score of Players") +
  xlab("Score Player 1") +
  ylab("Score Player 2")

# plot the distribution of score versus number of turns
p2 = plot_data[plot_data$agree, ] %>%
  ggplot(aes(x=as.factor(n_turns), y = score+other_score)) +
  geom_boxplot() +
  ggtitle("Combined Score Versus Number of Turns") +
  xlab("Number of Turns") +
  ylab("Score")

grid.arrange(p1,p2, ncol=2)
```

Code 10: Plot of Score versus Dialogue Order

```
# plot the score versus the dialogue order
plot_data[plot_data$agree, ] %>%
  ggplot(aes(x = score, fill = went_first))+
  geom_bar(alpha=0.5, position = "identity")+
  ggtitle("Score Versus Order of Dialogue")+
  ylab("Count") +
  xlab("Score")+
  scale_fill_discrete(name="Order")
```

Code 11: Transformer Model Class

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.nn.init
from torch.autograd import Variable
import torch.nn.functional as F

import data
from engines.rnn_engine import RnnEngine
from domain import get_domain
from models.utils import *
from models.ctx_encoder import MlpContextEncoder

class RnnModel(nn.Module):
    corpus_ty = data.WordCorpus
    engine_ty = RnnEngine

    def __init__(self, word_dict, item_dict, context_dict, count_dict, args):
        super(RnnModel, self).__init__()
        domain = get_domain(args.domain)
        self.word_dict = word_dict
        self.item_dict = item_dict
        self.context_dict = context_dict
        self.count_dict = count_dict
```

```
        self.args = args

        self.word_encoder = nn.Embedding(len(self.word_dict), args.nembed_word)
        self.word_encoder_dropout = nn.Dropout(args.dropout)

        ctx_encoder_ty = MlpContextEncoder
        self.ctx_encoder = nn.Sequential(
            ctx_encoder_ty(
                len(self.context_dict),
                domain.input_length(),
                args.nembed_ctx,
                args.nhid_ctx,
                args.dropout,
                args.init_range,
            ),
            nn.Dropout(args.dropout),
        )

        self.reader = nn.GRU(
            args.nhid_ctx + args.nembed_word, args.nhid_lang, bias=True
        )  # h_t
        self.reader_dropout = nn.Dropout(args.dropout)

        self.decoder = nn.Sequential(
            nn.Linear(args.nhid_lang, args.nembed_word), nn.Dropout(args.dropout)
        )

        self.writer = nn.GRUCell(
            input_size=args.nhid_ctx + args.nembed_word,
            hidden_size=args.nhid_lang,
            bias=True,
        )

        # Tie the weights of reader and writer
        self.writer.weight_ih = self.reader.weight_ih_l0
        self.writer.weight_hh = self.reader.weight_hh_l0
        self.writer.bias_ih = self.reader.bias_ih_l0
        self.writer.bias_hh = self.reader.bias_hh_l0

        self.sel_rnn = nn.TransformerEncoderLayer(
            d_model=args.nhid_lang + args.nembed_word, nhead=4, dropout=args.dropout
        )

        self.sel_dropout = nn.Dropout(args.dropout)

        # Mask for disabling special tokens when generating sentences
        self.special_token_mask = torch.FloatTensor(len(self.word_dict))

        self.sel_encoder = nn.Sequential(
            torch.nn.Linear(
                args.nhid_lang + args.nembed_word + args.nhid_ctx, args.nhid_sel
            ),
            nn.Tanh(),
```

```python
            nn.Dropout(args.dropout),
        )

        self.attn = nn.Sequential(
            torch.nn.Linear(args.nhid_lang + args.nembed_word, args.nhid_attn),
            nn.Tanh(),
            torch.nn.Linear(args.nhid_attn, 1),
        )

        self.sel_decoders = nn.ModuleList()
        for i in range(domain.selection_length()):
            self.sel_decoders.append(nn.Linear(args.nhid_sel, len(self.item_dict)))

        self.init_weights()

        self.special_token_mask = make_mask(
            len(word_dict),
            [word_dict.get_idx(w) for w in ["<unk>", "YOU:", "THEM:", "<pad>"]],
        )

    def flatten_parameters(self):
        self.reader.flatten_parameters()
        self.sel_rnn.flatten_parameters()

    def zero_h(self, bsz, nhid=None, copies=None):
        nhid = self.args.nhid_lang if nhid is None else nhid
        copies = 1 if copies is None else copies
        h = torch.Tensor(copies, bsz, nhid).fill_(0)
        return Variable(h)

    def word2var(self, word):
        x = torch.Tensor(1).fill_(self.word_dict.get_idx(word)).long()
        return Variable(x)

    def init_weights(self):
        init_rnn(self.reader, self.args.init_range)
        init_cont(self.decoder, self.args.init_range)
        self.word_encoder.weight.data.uniform_(
            -self.args.init_range, self.args.init_range
        )

        init_cont(self.attn, self.args.init_range)
        init_cont(self.sel_encoder, self.args.init_range)
        init_cont(self.sel_decoders, self.args.init_range)

    def forward_context(self, ctx):
        ctx_h = self.ctx_encoder(ctx).unsqueeze(0)
        return ctx_h

    def forward_lm(self, inpt_emb, lang_h, ctx_h):
        # append the context embedding to every input word embedding
        ctx_h_rep = ctx_h.narrow(0, ctx_h.size(0) - 1, 1).expand(
            inpt_emb.size(0), ctx_h.size(1), ctx_h.size(2)
        )
```

```
        inpt_emb = torch.cat([inpt_emb, ctx_h_rep], 2)
        # n_words x (ctx_h size {64} + inpt size {256})
        # combine word embeddings and context (g) embeddings

        lang_hs, _ = self.reader(inpt_emb, lang_h)
        lang_hs = self.reader_dropout(lang_hs)  # n_words x 128 (nhid_lang)

        decoded = self.decoder(lang_hs.view(-1, lang_hs.size(2)))
        # n_words x 256 (nembed_word)

        out = F.linear(decoded, self.word_encoder.weight)
        # n_words x 463 (encoder size)

        return out, lang_hs

    def forward_selection(self, inpt_emb, lang_h, ctx_h):
        # run a birnn over the concatenation of the input embeddings and
    language model hidden states
        h = torch.cat([lang_h, inpt_emb], 2)
        # n_words x (256 + 128)
        # combine word imbeddings and output of second GRU

        attn_h = self.zero_h(h.size(1), self.args.nhid_attn, copies=2)
        # 2 x 64 (nhid_attn)
        # initial hidden state of the third GRU
        # n_words x 128 (nhid_attn x 2) b/c bidirectional

        h = self.sel_rnn(h)

        # h = self.sel_dropout(h)

        h = h.transpose(0, 1).contiguous()
        # batch_size x n_words x 128

        logit = self.attn(h.view(-1, self.args.nhid_lang + self.args.
    nembed_word)).view(
            h.size(0), h.size(1)
        )
        # first removes the batch dimension
        # then calculates attn (Linear, Tanh, Linear)
        # batch_size x n_words
        # this is h_t^a

        prob = F.softmax(logit, dim=1).unsqueeze(2).expand_as(h)
        # batch_size x n_words x 128
        # these are all the same along the last dimension
        # these are the alpha_t

        attn = (
            torch.sum(torch.mul(h, prob), 1, keepdim=True).transpose(0, 1).
    contiguous()
        )
        # 1 x 1 x 128

        h = torch.cat([attn, ctx_h], 2).squeeze(0)
        # concatenate the context vector (g) and the attention scores
```

```
        # 1 x 1 x 192

        h = self.sel_encoder.forward(h)
        # this is h_s
        # batch_size x 128

        # there are 6 decoders in self.sel_decoders
        # each is a linear with output size of 18

        outs = [decoder.forward(h) for decoder in self.sel_decoders]
        out = torch.cat(outs, 0)
        # this is equation (9) in the paper
        # 6 x 18
        return out

    def forward(self, inpt, ctx):
        # ctx is 6 x 1
        # inpt is n_words x 1
        ctx_h = self.forward_context(ctx)  # 1 x 1 x 64

        lang_h = self.zero_h(
            ctx_h.size(1), self.args.nhid_lang
        )  # 1 x 1 x 128 all zeros
        # initial hidden state of the second GRU

        inpt_emb = self.word_encoder(inpt)  # n_words x 256
        inpt_emb = self.word_encoder_dropout(inpt_emb)

        out, lang_hs = self.forward_lm(inpt_emb, lang_h, ctx_h)
        sel_out = self.forward_selection(inpt_emb, lang_hs, ctx_h)
        return out, sel_out
```

Code 12: Monte Carlo Tree Search Agent Class

```
class RnnMCTSAgent(RnnAgent):
    def __init__(self, model,args, name="Alice", train=False, diverse=False):
        super(RnnMCTSAgent, self).__init__(model, args, name)
        # add variables to store the number of MCTS
        self.nsim = 50
        self.rollout_len = 100
        self.n_tries = 5

    def pickNode(self, curr_node: Node):
        '''
        picks max ucb node
        '''
        max_ucb = -math.inf
        selection = None
        for child in curr_node.children:
            curr_ucb = child.get_ucb()
            if curr_ucb > max_ucb:
                max_ucb = curr_ucb
                selection = child
        return selection

    def expansion(self, curr_node: Node):
        """
```

```
        goes through all nodes until it reaches a leaf
        """
        if not curr_node.children:
            return curr_node
        selection = self.pickNode(curr_node)
        return self.expansion(selection)


    def backprop(self, curr_node: Node, score: float):
        """
        backpropagate scores throughout the tree
        """
        while True:
            curr_node.score += score
            curr_node.n += 1
            if not curr_node.parent:
                break
            curr_node = curr_node.parent
        return curr_node

    def get_all_children_states(self, curr_node: Node, n_tries: int):
        """
        get a few possible dialogues given the current state
        """
        children = []
        prev_moves = []
        for _ in range(n_tries):
            _, move, move_lang_h, move_lang_hs = self.model.write(
                curr_node.lang_h, self.ctx_h, self.rollout_len, self.args.temperature)
            is_selection = len(move) == 1 and \
                self.model.word_dict.get_word(move.data[0][0]) == '<selection>'
            if not any([torch.equal(move,x) for x in prev_moves]):
                children.append(Node(parent = curr_node, lang_h=move_lang_h, lang_hs = move_lang_hs, move = move, sel=is_selection))
                prev_moves.append(move)
        return children

    def prepare_words(self, combined_words):
        """
        get words in the right format to be passed to _choose1
        """
        res = []
        start = 0
        end = 0
        for i in range(combined_words.size(0)):
            if combined_words[i, :] == 0:
                end = i + 1
                res.append(combined_words[start:end, :])
                start = end
        return res

    def _choose1(self, sents, sample=False):
        lens, rev_idxs, hid_idxs = self._make_idxs(sents)
        sel_out = self.sel_model.forward(sents, lens, rev_idxs, hid_idxs, Variable(self.ctx))
```

```python

        choices = self.domain.generate_choices(self.context, with_disagreement=True)

        choices_logits = []
        for i in range(self.domain.selection_length()):
            idxs = [self.sel_model.item_dict.get_idx(c[i]) for c in choices]
            idxs = Variable(torch.Tensor(idxs).long())
            choices_logits.append(torch.gather(sel_out[i], 0, idxs).unsqueeze(1))

        choice_logit = torch.sum(torch.cat(choices_logits, 1), 1, keepdim=True).squeeze(1)
        choice_logit = choice_logit.sub(choice_logit.max(0)[0].item())
        prob = F.softmax(choice_logit, dim=0)

        if sample:
            idx = prob.multinomial(1).detach()
            logprob = F.log_softmax(choice_logit, dim=0).gather(0, idx)
        else:
            _, idx = prob.max(0, keepdim=True)
            logprob = None

        p_agree = prob[idx.item()]

        # Pick only your choice
        return choices[idx.item()][:self.domain.selection_length()], logprob, p_agree.item()

    def write(self, max_words):
        """
        new write method for generating turn
        """
        root = Node(lang_h = self.lang_h)
        root.children = self.get_all_children_states(root, self.n_tries)

        for _ in range(self.nsim):
            score = 0
            '''
            get max node for traversal
            '''
            candidate = self.expansion(root)

            if candidate.n != 0:
                candidate.children = self.get_all_children_states(candidate, self.n_tries)
                candidate = candidate.children[0]


            combined_words = self.words + [self.model.word2var('YOU:'), candidate.outs]
            # print(combined_words)
            if not candidate.is_selection:
                """"
                do full rollout and get the score if its not a terminal node
                """
```

```
                _, rollout, _, rollout_lang_hs = self.model.write(
                        candidate.lang_h, self.ctx_h, self.rollout_len, self.args.temperature,
                        stop_tokens=['<selection>'], resume=True)
                combined_words += [rollout]

            combined_words = [(lambda x: x.unsqueeze(1) if len(x.size()) == 1 else x)(x) for x in combined_words]
            combined_words = torch.cat(combined_words, dim = 0)
            combined_words = self.prepare_words(combined_words)
            rollout_choice, _, p_agree = self._choose1(sents=combined_words, sample=False)
            rollout_score = self.domain.score(self.context, rollout_choice)
            # score += p_agree * rollout_score
            score += rollout_score

            self.backprop(candidate, score)

        max_score = -math.inf
        for child in root.children:
            if child.score / child.n > max_score:
                bestChild = child
                max_score = child.score / child.n
        self.lang_h = bestChild.lang_h
        self.lang_hs.append(bestChild.lang_hs)
        self.words.append(self.model.word2var('YOU:'))
        self.words.append(bestChild.outs)
        self.sents.append(torch.cat([self.model.word2var('YOU:').unsqueeze(1), bestChild.outs], 0))
        return self._decode(bestChild.outs, self.model.word_dict)
```

Code 13: Hyperparameter Tuning

```
# define function to maximize
def objective(trial: optuna.trial.Trial):
    # add arguments to be read from the command line
    parser = argparse.ArgumentParser(description='training script')
    parser.add_argument('--data', type=str, default='data/negotiate',
        help='location of the data corpus')
    parser.add_argument('--nembed_word', type=int, default=256,
        help='size of word embeddings')
    parser.add_argument('--nembed_ctx', type=int, default=64,
        help='size of context embeddings')
    parser.add_argument('--nhid_lang', type=int, default=256,
        help='size of the hidden state for the language module')
    parser.add_argument('--nhid_cluster', type=int, default=256,
        help='size of the hidden state for the language module')
    parser.add_argument('--nhid_ctx', type=int, default=64,
        help='size of the hidden state for the context module')
    parser.add_argument('--nhid_strat', type=int, default=64,
        help='size of the hidden state for the strategy module')
    parser.add_argument('--nhid_attn', type=int, default=64,
        help='size of the hidden state for the attention module')
    parser.add_argument('--nhid_sel', type=int, default=64,
        help='size of the hidden state for the selection module')
    parser.add_argument('--lr', type=float, default=20.0,
        help='initial learning rate')
```

```
    parser.add_argument('--min_lr', type=float, default=1e-5,
        help='min threshold for learning rate annealing')
    parser.add_argument('--decay_rate', type=float,  default=9.0,
        help='decrease learning rate by this factor')
    parser.add_argument('--decay_every', type=int,  default=1,
        help='decrease learning rate after decay_every epochs')
    parser.add_argument('--momentum', type=float, default=0.0,
        help='momentum for sgd')
    parser.add_argument('--clip', type=float, default=0.2,
        help='gradient clipping')
    parser.add_argument('--dropout', type=float, default=0.5,
        help='dropout rate in embedding layer')
    parser.add_argument('--init_range', type=float, default=0.1,
        help='initialization range')
    parser.add_argument('--max_epoch', type=int, default=30,
        help='max number of epochs')
    parser.add_argument('--num_clusters', type=int, default=50,
        help='number of clusters')
    parser.add_argument('--bsz', type=int, default=25,
        help='batch size')
    parser.add_argument('--unk_threshold', type=int, default=20,
        help='minimum word frequency to be in dictionary')
    parser.add_argument('--temperature', type=float, default=0.1,
        help='temperature')
    parser.add_argument('--partner_ctx_weight', type=float, default=0.0,
        help='selection weight')
    parser.add_argument('--sel_weight', type=float, default=0.6,
        help='selection weight')
    parser.add_argument('--seed', type=int, default=1,
        help='random seed')
    parser.add_argument('--cuda', action='store_true', default=False,
        help='use CUDA')
    parser.add_argument('--model_file', type=str,  default='',
        help='path to save the final model')
    parser.add_argument('--prediction_model_file', type=str,  default='',
        help='path to save the prediction model')
    parser.add_argument('--selection_model_file', type=str,  default='',
        help='path to save the selection model')
    parser.add_argument('--cluster_model_file', type=str,  default='',
        help='path to save the cluster model')
    parser.add_argument('--lang_model_file', type=str,  default='',
        help='path to save the language model')
    parser.add_argument('--visual', action='store_true', default=False,
        help='plot graphs')
    parser.add_argument('--skip_values', action='store_true', default=False,
        help='skip values in ctx encoder')
    parser.add_argument('--model_type', type=str, default='rnn_model',
        help='model type', choices=models.get_model_names())
    parser.add_argument('--domain', type=str, default='object_division',
        help='domain for the dialogue')
    parser.add_argument('--clustering', action='store_true', default=False,
        help='use clustering')
    parser.add_argument('--sep_sel', action='store_true', default=False,
        help='use separate classifiers for selection')

    # grab the arguments from the command line
```

```
    args = parser.parse_args()

    # use GPU and set seed
    utils.use_cuda(args.cuda)
    utils.set_seed(args.seed)

    # set the possible range for variables to tune
    args.clip = trial.suggest_float("clip",0.25,0.75)
    args.decay_every = trial.suggest_int("decay_every", 1, 5)
    args.decay_rate = trial.suggest_int("decay_rate", 2, 10)
    args.dropout = trial.suggest_float("dropout", 0.1, 0.9)
    args.init_range = trial.suggest_float("init_range", 0.1, 0.5)
    args.lr = trial.suggest_float("initial_learning_rate", 1e-5, 1e-2)
    args.min_lr = trial.suggest_float("min_learning_rate", 1e-9,1e-6)
    args.momentum = trial.suggest_float("momentum", 0, 1)
    args.nembed_ctx = trial.suggest_categorical("ctx_embeding", [64,128,256])
    args.nembed_word = trial.suggest_categorical("word_embeding", [64,128,256])
    args.nhid_attn = trial.suggest_categorical("hidden_attn_size", [64,128,256])
    args.nhid_ctx = trial.suggest_categorical("hidden_ctx_size", [64,128,256])
    args.nhid_lang = trial.suggest_categorical("hidden_lang_size", [64,128,256])
    args.nhid_sel = trial.suggest_categorical("hidden_selection_size", [64,128,256])
    args.sel_weight = trial.suggest_float("selection_weight", 0.2, 0.8)

    # set the domain of the game (i.e., dividing objects between players)
    domain = get_domain(args.domain)
    # get the model we use for training
    model_ty = models.get_model_type(args.model_type)
    # get the dialogues we train on
    corpus = model_ty.corpus_ty(domain, args.data, freq_cutoff=args.unk_threshold,
        verbose=True, sep_sel=args.sep_sel)
    # initialize the model
    model = model_ty(corpus.word_dict, corpus.item_dict_old,
        corpus.context_dict, corpus.count_dict, args)
    if args.cuda:
        model.cuda()
    # initialize the engine (the object that actually trains the model)
    engine = model_ty.engine_ty(model, args, verbose=True)
    # train the model
    train_loss, valid_loss, select_loss, extra = engine.train(corpus)
    # save the model
    utils.save_model(engine.get_model(), args.model_file)

    return valid_loss


# define how we will optimize our objective function
study = optuna.create_study(direction = 'minimize', sampler = optuna.samplers.TPESampler(seed=4850))
# find the best hyperparameters
study.optimize(objective, n_trials=1000)
# print the best parameters
print(study.best_trial)
```

Code 14: Class to Create Dialogues Between Two Players

```
class Dialog(object):
    def __init__(self, agents, args):
        # For now we only suppport dialog of 2 agents
        assert len(agents) == 2
        self.agents = agents
        self.args = args
        self.domain = domain.get_domain(args.domain)
        self.metrics = MetricsContainer()
        self._register_metrics()

    def _register_metrics(self):
        self.metrics.register_average('dialog_len')
        self.metrics.register_average('sent_len')
        self.metrics.register_percentage('agree')
        self.metrics.register_moving_percentage('moving_agree')
        self.metrics.register_average('advantage')
        self.metrics.register_moving_average('moving_advantage')
        self.metrics.register_time('time')
        self.metrics.register_average('comb_rew')
        self.metrics.register_average('agree_comb_rew')
        for agent in self.agents:
            self.metrics.register_average('%s_rew' % agent.name)
            self.metrics.register_moving_average('%s_moving_rew' % agent.name)
            self.metrics.register_average('agree_%s_rew' % agent.name)
            self.metrics.register_percentage('%s_sel' % agent.name)
            self.metrics.register_uniqueness('%s_unique' % agent.name)
        # text metrics
        if self.args.ref_text:
            ref_text = ' '.join(data.read_lines(self.args.ref_text))
            self.metrics.register_ngram('full_match', text=ref_text)

    def _is_selection(self, out):
        return len(out) == 1 and (out[0] in ['<selection>', '<no_agreement>'])

    def show_metrics(self):
        return ' '.join(['%s=%s' % (k, v) for k, v in self.metrics.dict().items
    ()])

    def run(self, ctxs, logger, max_words=5000):
        self.agents[0].model.train()
        assert len(self.agents) == len(ctxs)
        for agent, ctx, partner_ctx in zip(self.agents, ctxs, reversed(ctxs)):
            agent.feed_context(ctx)
            agent.feed_partner_context(partner_ctx)
            logger.dump_ctx(agent.name, ctx)
        logger.dump('-' * 80)

        # Choose who goes first by random
        if np.random.rand() < 0.5:
            writer, reader = self.agents
        else:
            reader, writer = self.agents

        conv = []
        self.metrics.reset()
```

```

        #words_left = np.random.randint(50, 200)
        words_left = max_words
        length = 0
        expired = False
        while True:
            out = writer.write(max_words=words_left)
            words_left -= len(out)
            length += len(out)

            self.metrics.record('sent_len', len(out))
            if 'full_match' in self.metrics.metrics:
                self.metrics.record('full_match', out)
            self.metrics.record('%s_unique' % writer.name, out)

            conv.append(out)
            reader.read(out)
            if not writer.human:
                logger.dump_sent(writer.name, out)

            if self._is_selection(out):
                self.metrics.record('%s_sel' % writer.name, 1)
                self.metrics.record('%s_sel' % reader.name, 0)
                break

            if words_left <= 1:
                break

            writer, reader = reader, writer


        choices = []
        for agent in self.agents:
            choice = agent.choose()
            choices.append(choice)
            logger.dump_choice(agent.name, choice[: self.domain.selection_length() // 2])

        agree, rewards = self.domain.score_choices(choices, ctxs)
        if expired:
            agree = False
        logger.dump('-' * 80)
        logger.dump_agreement(agree)
        for i, (agent, reward) in enumerate(zip(self.agents, rewards)):
            logger.dump_reward(agent.name, agree, reward)
            j = 1 if i == 0 else 0
            agent.update(agree, reward, choice=choices[i],
                partner_choice=choices[j], partner_input=ctxs[j], partner_reward=rewards[j])

        if agree:
            self.metrics.record('advantage', rewards[0] - rewards[1])
            self.metrics.record('moving_advantage', rewards[0] - rewards[1])
            self.metrics.record('agree_comb_rew', np.sum(rewards))
            for agent, reward in zip(self.agents, rewards):
                self.metrics.record('agree_%s_rew' % agent.name, reward)
```

```

        self.metrics.record('time')
        self.metrics.record('dialog_len', len(conv))
        self.metrics.record('agree', int(agree))
        self.metrics.record('moving_agree', int(agree))
        self.metrics.record('comb_rew', np.sum(rewards) if agree else 0)
        for agent, reward in zip(self.agents, rewards):
            self.metrics.record('%s_rew' % agent.name, reward if agree else 0)
            self.metrics.record('%s_moving_rew' % agent.name, reward if agree
    else 0)

        logger.dump('-' * 80)
        logger.dump(self.show_metrics())
        logger.dump('-' * 80)
        for ctx, choice in zip(ctxs, choices):
            logger.dump('debug: %s %s' % (' '.join(ctx), ' '.join(choice)))

        return conv, agree, rewards
```

Code 15: Class for Reinforcement Learning

```
class Reinforce(object):
    def __init__(self, dialog, ctx_gen, args, engine, corpus, logger=None):
        self.dialog = dialog
        self.ctx_gen = ctx_gen
        self.args = args
        self.engine = engine
        self.corpus = corpus
        self.logger = logger if logger else DialogLogger()

    def run(self):
        validset, validset_stats = self.corpus.valid_dataset(self.args.bsz)
        trainset, trainset_stats = self.corpus.train_dataset(self.args.bsz)

        n = 0
        for ctxs in self.ctx_gen.iter(self.args.nepoch):
            n += 1
            if self.args.sv_train_freq > 0 and n % self.args.sv_train_freq ==
    0:
                batch = random.choice(trainset)
                self.engine.model.train()
                self.engine.train_batch(batch)
                self.engine.model.eval()
            self.logger.dump('=' * 80)
            self.engine.model.train()
            self.dialog.run(ctxs, self.logger)
            self.logger.dump('=' * 80)
            self.logger.dump('')
            if n % 100 == 0:
                self.logger.dump('%d: %s' % (n, self.dialog.show_metrics()),
    forced=True)

        def dump_stats(dataset, stats, name):
            loss, select_loss = self.engine.valid_pass(N, dataset, stats)
            self.logger.dump('final: %s_loss %.3f %s_ppl %.3f' % (
                name, float(loss), name, np.exp(float(loss))),
                forced=True)
```

```
            self.logger.dump('final: %s_select_loss %.3f %s_select_ppl %.3f' % (
                name, float(select_loss), name, np.exp(float(select_loss))),
                forced=True)

        dump_stats(trainset, trainset_stats, 'train')
        dump_stats(validset, validset_stats, 'valid')

        self.logger.dump('final: %s' % self.dialog.show_metrics(), forced=True)
```

Code 16: Agent Class for Reinforcement Learning

```
class RlAgent(RnnAgent):
    def __init__(self, model, args, name='Alice', train=False):
        self.train = train
        super(RlAgent, self).__init__(model, args, name=name)
        self.model.train()
        self.sel_model.train()
        self.opt = optim.RMSprop(
            self.model.parameters(),
            lr=args.rl_lr,
            momentum=self.args.momentum)

        self.all_rewards = []

        if self.args.visual:
            self.model_plot = vis.ModulePlot(self.model, plot_weight=False, plot_grad=True)
            self.agree_plot = vis.Plot(['agree',], 'agree', 'agree')
            self.reward_plot = vis.Plot(
                ['reward', 'partner_reward'], 'reward', 'reward')
            self.loss_plot = vis.Plot(['loss',], 'loss', 'loss')
            self.agree_reward_plot = vis.Plot(
                ['reward', 'partner_reward'], 'agree_reward', 'agree_reward')
        self.t = 0

    def feed_context(self, ctx):
        super(RlAgent, self).feed_context(ctx)
        self.logprobs = []

    def write(self, max_words):
        logprobs, outs, self.lang_h, lang_hs = self.model.write(self.lang_h, self.ctx_h,
            100, self.args.temperature)
        self.logprobs.extend(logprobs)
        self.lang_hs.append(lang_hs)
        self.words.append(self.model.word2var('YOU:').unsqueeze(0))
        self.words.append(outs)
        assert (torch.cat(self.words).size()[0] == torch.cat(self.lang_hs).size()[0])
        return self._decode(outs, self.model.word_dict)

    def choose(self):
        if self.args.eps < np.random.rand():
            choice, _, _ = self.choose(sample=False)
        else:
            choice, logprob, _ = self._choose(sample=True)
```

```
            self.logprobs.append(logprob)
        return choice

    def update(self, agree, reward, choice=None, partner_choice=None, partner_input=None, partner_reward=None):
        if not self.train:
            return

        self.t += 1
        if len(self.logprobs) == 0:
            return
        reward_agree = reward
        partner_reward_agree = partner_reward

        reward = reward if agree else 0
        partner_reward = partner_reward if agree else 0

        diff = reward - partner_reward
        self.all_rewards.append(diff)
        #self.all_rewards.append(reward)
        r = (diff - np.mean(self.all_rewards)) / max(1e-4, np.std(self.all_rewards))
        g = Variable(torch.zeros(1, 1).fill_(r))
        rewards = []
        for _ in self.logprobs:
            rewards.insert(0, g)
            g = g * self.args.gamma

        loss = 0
        for lp, r in zip(self.logprobs, rewards):
            loss -= lp * r

        self.opt.zero_grad()
        loss.backward()
        nn.utils.clip_grad_norm(self.model.parameters(), self.args.rl_clip)
        if self.args.visual and self.t % 10 == 0:
            self.model_plot.update(self.t)
            self.agree_plot.update('agree', self.t, int(agree))
            self.reward_plot.update('reward', self.t, reward)
            self.reward_plot.update('partner_reward', self.t, partner_reward)
            self.agree_reward_plot.update('reward', self.t, reward_agree)
            self.agree_reward_plot.update('partner_reward', self.t, partner_reward_agree)
            self.loss_plot.update('loss', self.t, loss.data[0][0])

        self.opt.step()
```

8 References

[1] Adrian Furnham and Hua Chu Boo. A literature review of the anchoring effect. *The Journal of Socio-Economics*, 40(1):35–42, 2011.

[2] Sylvain Gelly, Levente Kocsis, Marc Schoenauer, Michèle Sebag, David Silver, Csaba Szepesvári, and Olivier Teytaud. The grand challenge of computer go. *Communications of the ACM*, 55(3):106–113, 2012.

[3] Daniel Kahneman, Jack L Knetsch, and Richard H Thaler. Anomalies: The endowment effect, loss aversion, and status quo bias. *Journal of Economic Perspectives*, 5(1):193–206, 1991.

[4] Shigeki Karita, Nanxin Chen, Tomoki Hayashi, Takaaki Hori, Hirofumi Inaguma, Ziyan Jiang, Masao Someki, Nelson Enrique Yalta Soplin, Ryuichi Yamamoto, Xiaofei Wang, Shinji Watanabe, Takenori Yoshimura, and Wangyou Zhang. A comparative study on transformer vs RNN in speech applications. *CoRR*, abs/1909.06317, 2019.

[5] Mike Lewis, Denis Yarats, Yann N. Dauphin, Devi Parikh, and Dhruv Batra. Deal or no deal? end-to-end learning for negotiation dialogues. *CoRR*, abs/1706.05125, 2017.

[6] Alan Strudler. On the ethics of deception in negotiation. *Business Ethics Quarterly*, 5(4):805–822, Oct 1995.

[7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Conference on Neural Information Processing Systems*, 31, 2017.