Abstract:

The Turing test is a game in which a human interrogator has to evaluate two responses to a question, one question from a human, and the other from a machine. If the interrogator cannot reliably tell the machine from the human, the machine passes the test, wins the game. My project flips the Turing test on its head and replaces the human interrogator with a machine interrogator that tries to reliably tell the human response from the machine response.

Conclusions:

Word2Vec total binary cross entropy loss normalized by number of batches : 0.4132
TextGenerate cross entropy loss normalized by number of batches: 5.74
MachineInterrogator accuracy:

Epoch 1: 97%
Epoch 2: 100%

I include epoch 1 just to show that at the start of training for some of the responses, the machine thought that we were machines…It quickly learned what machine vs human responses were.

Data:

To construct a language basis for this machine interrogator I built a web crawler that scraped text off of wikipedia. For humor, I began the page crawl from the “Machine Learning” wikipedia page. To clean the data, all text is lowered and alphanumeric, I included stop words for completeness of statements. I need to add a stop word weight so that their frequency doesn’t dominate the language model. Once I’ve gathered all the text and I encapsulate it in a Data class called Corpus.

Models:

- Word2Vec (Constructing a language for both machines)
- TextGenerate (Responding Machine)
- MachineInterrogator

Word2Vec:

My Word2Vec model uses a skip-gram model rather than a continuous bag of words model. Which means that my model tries to predict surrounding words from a target word, rather than predict the target word based on surrounding words. To understand how I created it you can take a look at the comments I put in my code.

You can change the start link of the wikipedia scrapper, and the number of links.
To view a new t-SNE plot of the word embeddings you can run the plot_word_embeddings with the word_embeds.vec file that outputs from the Word2Vec Model.

TextGenerate:

This is a model that acts as the machine that responds to a question. This model uses an LSTM (Long short term memory) which is a type of recurrent network, however more complex by adding internal state that travels through LSTM inputs. RNN (recurrent neural network) have phenomena called vanishing/exploding gradients. To avoid these problems, we use LSTMs and GRUs (Gated recurrent units) are used. In the case of LSTM we use these internal states. Since this model needs to work with lots of text information in sequence it needs to remember very long ago seen text to correctly predict the next word to use. Especially once responses get longer and longer. In order for the model to choose the next word, it generates a probability for every word in its vocabulary. You can imagine that if the machine only knew 5 words, the output for this model would be 5 probabilities that add up to 1, and I concatenate the highest probabilities words over a number of iterations based on how long the response needs to be.

MachineInterrogator:

This model acts as the machine interrogator. This model outputs a binary response, whether or not it's a human or a machine (0/1). This model also uses an LSTM because it needs to read responses and understand each word in the sentence and how they relate together from beginning to end of sentence.

```
MachineInterrogator(
  (lstm): LSTM(40, 128, num_layers=2, batch_first=True, dropout=0.15)
  (lim): Linear(in_features=128, out_features=1, bias=True)
)
```

References:

https://pytorch.org/docs/stable/index.html
Lectures from https://www.cs.rpi.edu/~zaki/courses/mlib/