Trexquant H A N G M A N challenge

By Kunaal Gautam | +91 95999 13428 | kunaalg21@iitk.ac.in

The documentation below outlines my strategies, approaches, heuristics and efforts in playing the hangman challenge for 7 days. I have proposed two strategies - one is deterministic, the other is Machine Learning based. My final submission strategy is the first one which yields a 59.6% accuracy.

Definitions

N-gram: Any ordered sequence of consecutive characters of length N

e.g. “ing” is a 3-gram of “tripping”

Masked Word: The incomplete word provided by the server to the agent, wherein the revealed letters are shown and the unknown letters are represented by _

e.g. “tr_pp_n_” is a masked word

Unmasked Word: The final answer *or* the complete word that is revealed when all the letters are guessed by the agent

e.g. “tripping” is an unmasked word and is the final answer of a possible game whose states are being used as examples in these definitions

Entropy : A quantity indicating the ‘reduction in disorder’ of the letter Ω. It tells us how much easier it would be to guess the word in the given tries given we guess Ω in the current try.

Information Gain : A quantity that indicates how much information the agent procures on guessing a letter Ω; ‘*information*’ here primarily affects how much our prior beliefs (probabilities) for different guesses have been updated.

Morphemes : The smallest grammatical units in a language that carry meaning. They can be words themselves (*bound*) or parts of words (*free*).

e.g. “ing” is a bound morpheme that is used in addition to the root word while “apple” is a free morpheme

1. ***Information Gain & Ngrams Strategy***

The Ngram approach is based on a mix of information theory and N-grams (length N substrings of words), which are widely used in natural language processing.

The idea behind the approach is to use the frequency of N-grams instead of just single letters. We first iterate through the whole dictionary, extracting substrings of length 1, 2, 3 to N and storing them separately. Now, for each N-gram, we gather their total frequency

across all words. We also do the same for N-letter prefixes (starting at the start of the word) and N-letter suffixes (ending at the end of the word).

These frequency tables are given specific weights depending on the length of the masked word (the length of the word to be guessed) and used to make guesses. We use different weights for different lengths, indicating that matches of greater lengths are more indicative than matches of smaller lengths.

We do place a restraint on some patterns, not considering those which have more than 2 not-yet-guessed letters, to ensure that we don't partake in "excess and vague" matches.

We use N = 6 because larger values become computationally expensive with a very marginal effect on the accuracy of the approach.

While the Ngram approach is highly deterministic and accurate in nature, it requires a certain threshold of information to work its magic. Our aim is to have a methodology for reaching this 'threshold knowledge' before utilizing the Ngram approach while minimizing the number of lives lost and maximizing the information we have about the masked word.

Here, we utilize the idea of Entropy and Information Gain. Implementing these two entities is analogous to their usage in Decision Trees, but they are used layer by layer in our strategy and are defined and constructed differently. The definitions have been provided above and here is their mathematical construction.

Formula for calculating entropy (summed over i = 1 to i = n) for say a letter β when the word length is α

$$Entropy = -\sum\left(\left(\left(\frac{x_i}{x}\right)\left(\frac{i}{\alpha}\right)^2\right) * log_2\left(\left(\frac{x_i}{x}\right)\left(\frac{i}{\alpha}\right)^2\right)\right)$$

i = number of times the letter β occured in the word
$n = max(i)$ {maximum times β appeared in a word}
α = length of the word
x_i = number of words where β appeared i times
x = total words eligible for search

Initially, x would be the set of words with length α, which would be given at the start
Post that, x would keep updating with each guess

Formula for Information Gain of choosing a letter β

$$I.G. = \left(\frac{x_\beta}{x}\right)(E(\beta)) + (\gamma)\left(\frac{x - x_\beta}{x}\right)\left(-\left(\frac{x - x_\beta}{x}\right)log_2\left(\frac{x - x_\beta}{x}\right)\right)$$

x_β = number of words where β is present
x = total words eligible for search
$E(\beta)$ = entropy of β
γ = penalising factor (we can't afford to be wrong)
$x - x_\beta$ = number of words where β is not present
We'll guess the letter with maximum I.G.

We iterate through the set of words in our dictionary (*referred to as x above*) and calculate the Information Gain for each possible guess; we guess the one with the highest Information Gain. We do this until we reach specific condition(s), after which we transition to the Ngrams approach.

The algorithm is as follows:

```
Input : a masked string indicating the current state of the word

Output : a char (the letter to be guessed in the game)

Process :
```

- iterate over all the words in the dictionary provided in the following manner:
  - create three dictionaries ngrams_freq, prefix_freq & suffix_freq
  - for n = 1 to n = 6:
    - create a window of size n, and iterate:
      - increase the frequency of the respective substring for the window n in ngram_freq
      - increase the frequency of the respective beginning substring for the window n in prefix_freq
      - increase the frequency of the respective ending substring for the window n in suffix_freq
- determine all the possible words in the dictionary which have a match with the provided input masked string AND do not contain letters which have already been guessed and are not part of the unmasked word.
- store these words in new_dictionary, which gets created and updated each time the guess function is called
- determine blank_frac (i.e. fraction of string covered by blanks)
  - if blank_frac >= 0.85 AND new_dictionary is not empty:
    - guess the letter which bears the highest information gain, based on the new_dictionary
  - else:
    - specify the weights for each length of ngram, prefix and suffix, in the form of a list. These weights vary depending on the length of the masked string
    - for each length:

- search for the most frequently occurring n-gram, prefix and suffix which can be a part of the unmasked word
- increase the guessing probability of the letters which have not been guessed yet in the most frequently occurring pattern, using the frequency of the pattern and the weight assigned to it - depending on the length of the masked string

parameters involved:

blank_fraction threshold,

weights assigned to each length

Error Analysis & subsequent modifications

1. Intuitively, one of the first things we can question is why the transition point (blank_frac) is such an arbitrary number. One would expect that the threshold knowledge required for the ngrams strategy would vary with the word length.
    a. Very large words (len_word > 9) are generally made up of the root words plus smaller (bound) morphemes. Here, the role of the ngrams strategy becomes vital and performs better even when there is no information gained (at the start) than the alternative strategy (*Info. Gain*)
    b. Very small words (len_word < 5) have a limited number of possible ngrams, which lowers the "vagueness" of the guesses and performs competitively to the alternate strategy (*Info. Gain*).

To gather data to test the effects of the modification and analyze its effect, we updated the 'start_game' function and the hangman class to compute the number of games won and number of games lost per length of the word, from which we defined a function to calculate the success rate for each length N in the possible word length ratios.

We observed a significant jump in the efficacy of predicting large words (>75%), while there was a minor jump in the case of small words. What was worrying was that for words of length 3,4,5, the success rate was as low as 2% in some cases. Clearly, there was a need for parameter tuning in the case of guessing small words in the game.

2. Another interesting thing to note is that if we look at it very superficially, guessing a vowel has a higher rate of being correct simply because of their high presence throughout the set - but there must be an approximate ratio, depending on word length, between the vowel count and the consonant count in a word.
    a. We iterated through the test training set and, for each word length n, found out the median vowel-to-consonant ratio. The logic employed here was that if the word crosses its median vowel-to-consonant ratio, the chance of a

vowel being a correct guess falls. We used the median instead of the common mean to account for the outliers.

b. The assumption here is that both the training set and test set, while disjoint, are large enough to have similar variance in terms of word length and the type of words within a subset of word length n.
c. For large words, this strategy yielded mixed results, likely due to overfitting while for small words, there was a marked increase in the accuracy.

It becomes obvious from spending a couple of nights doing parameter tuning, that there's a supernum to the accuracy of any deterministic algorithm when being used to solve hangman - there's a limit to how much feature extraction we humans can do manually and these class of algorithms would always need to follow a predetermined set of rules & heuristics rather than them learning from data.

In the current strategy, there are certain issues which would be extremely difficult to sort by setting manual strategies (ie without the use of any *machine learning*)

1. The strategy doesn't give great importance to the position of these blank characters, which results in guesses which seem counterintuitive to users of the language.
2. The strategy does not learn from its mistakes - wherein it would make the same erroneous guess in the same situation multiple times. That results in a significant loss of accuracy
3. The core intention behind using n-grams and morphemes are their semantic and morphological validity - "ie" exists but "kzl" does not. In the current implementation of the strategy, these nonsensical ngrams are not considered different and hence lead to a loss in accuracy

To try to resolve these issues, instead of focusing on parameter tuning which was giving very low jumps (~1-3%) in accuracy, we tried to implement a machine learning model in the last 48 hours of the challenge.

2. ***LSTM (Long short-term memory networks) strategy***

The idea behind this strategy is for the model to be sufficiently trained to identify patterns and sequences and then guess the letter, following similar patterns as a human agent would do to solve the game.

Neural networks are the building block of the strategy. They consist of layers of interconnected nodes, or artificial neurons, that process information and are excellent at recognizing patterns in data by continuously shifting their weights and 'learning' from data.

Recurrent Neural Networks (RNNs) bring memory to the table, wherein they are used for sequential data analysis and in cases where the order of data matters, such as our game. We use Long short-term memory networks (LSTMs), a very specific and advanced version of RNNs.

LSTMs introduce specialized units called "cells" and use gates to control the flow of information within the cell. These gates regulate what gets stored and what gets discarded, making LSTMs incredibly skilled at capturing long-term dependencies in sequences.

One of the key advantages of LSTMs in Hangman is their capability to capture long-range dependencies within the sequence. In the game, letters guessed early on can significantly impact later guesses. LSTMs address this challenge by permitting information to propagate through their memory cells over extended periods, enabling the model to remember and leverage insights from early guesses to enhance later ones.

The algorithm is simple in nature, though a little complicated in implementation.

- We leverage PyTorch's libraries and functions to create our RNN model and neural layers. All our major calculations occur in the form of PyTorch tensors, which we convert from and to Python lists near I/O.
- *Word2Batch* is defined as a class that creates instances for the agent to play the game.
    - The game state is encoded, and we use certain entities that convert this state into numerical representations that the model can work with.
        - *obscure_word()*: encodes the current state of the word
        - *prev_guess()*: the agent's guessed letters are placed in a numerical vector
        - *correct_response()*: creates a representation, signifying which letters are still hidden and need to be guessed. The probability of each letter is normalized based on the remaining letters.
    - *game_mimic()* implements the game logic wherein it runs until the agent guesses the word correctly or runs out of lives
- *StatefulLSTM* class is used to implement custom neural network layers. We define hidden and cell states that are used to retain information by the LSTM cell

    - *LockedDropout* is utilized to regularize and prevent overfitting (randomly drops some neurons during training to improve generalization)
    - Our aim is to make sure the model does not rely too heavily on specific neurons
- Finally, we define the *RNN_model* class, which would be called onto our main guess function, albeit already pre-trained on words_250000_train.txt
    - Along with the LSTM layer, a *batch normalization* layer (*nn.BatchNorm1d*)is applied to stabilize training, after which dropout is also implemented.
    - *forward* method is defined for the data flow through the network, taking in *obscure_word & prev_guess*
    - The output is the agent's guess letter for the game
- The last part is similar to our previous approach in creating n-grams, though, in the base level of implementation, only 2 grams are created to prevent over computation.

The model is then trained on words_250000_train.txt. The dataset was broken into 20 smaller sets with random divides and the model was sequentially trained on all of them.

In our main guess function, the pre-trained model is loaded along with *model.pth* which has all the trained weights and biases. The main two parameters (*obscure_word and prev_guess)* are determined, a block is added to remove already guessed letters from a possible set of guessing and the guess function is implemented.

Error Analysis

This strategy is definitely on the right path and can be optimized to achieve high accuracy. We have used only 2-grams since there was a lack of time for training the model but using more n-grams would be useful in updating the weights for the guess function.

We would have preferred to optimize the strategy more and fine-tune the parameters but couldn't do it as the 7-day deadline was up. The strategy guessed the word nearly half the time based on training for 2 epochs for 20 subsets of the training set.

We can also experiment with better loss functions than Binary Cross-Entropy (BCE) loss - Categorical Cross-Entropy (CE) loss would be a more apt function in this case