Individual Programming Project Report

An analysis of project implementation

PREPARED BY: ADITYA KUMAR

Application Details

Approaches that I used:

1 - Natural Language Processing (NLP)

2 - Bidirectional Long-Short Term Memory

3 - Recurrent Neural Network

4 - Reinforcement Learning (Deployed Strategy is more than 60% accurate.)

[Note: All the code of the above mentioned approaches are attached here.]

Idea that I have not used due to Shortage of time:

1 – Gated Recurrent Unit (GRU layers)

Approaches that I used: Description of the strategy I deployed

1. Reinforcement Learning:

I tried my hand at building a reinforcement learning-enabled model to play Hangman. The goal is a win percentage greater than 50%. There are multiple ways to solve this problem, I'd have to imagine, but the problem statement seemed to lend itself well towards RL.

As per your recommendations I've attached the alternative solution architectures also, for your kind perusal !

- My initial solution design was to employ Q-Learning and statistical in the sense that first of all, I take maximum count of a letter in a word as one because ultimately, if a letter exists more than once it is filled at each location. I cleaned the word, removed spaces from input and searched for similar lettered word in dictionary and guessed the maximum occurring letter which is not guessed at current iteration.
- Next, I took out the ratio of vowels count to length of the world and found out that if 55% of the length of word is vowel, it is unlikely (from the distribution) that there will be more vowels.
- If we run out of similar words in given dictionary as the semi guessed word. So, like the thought process of a human, I try to break the word in smaller fragments (suffix/prefix) to try to fit in the most appropriate letter. This is what my algorithm tries to do by looking into another dictionary of the substring of words given in the original dictionary.
- At last if the letter is not found in the substrings' dictionary, the also guesses the most common letter in whole words which is not yet guessed.
- I got to a point where I saw actual training happening. At some point, I dug into Deep-Q Networks but ultimately went away from that because I wasn't quite following what was going on, and if I can't explain it to myself, I can't explain it to anyone else.

2. Natural Language Processing (NLP):

Now Drawing inspiration from other Hangman solvers, As I integrated several strategies to approach solving the Hangman problem. It one of them using NLP.

Before any guessing, I update a dictionary of possible words from the training set that match the guessed word-in-progress in terms of length and characters. I calculate the vowel/length ratio of the word-in-progress, and using this information later on, if more than 50% of the letters in the guessed word-in-progress are vowels, then we should not guess a vowel. I also leveraged the nltk library to get the frequencies of n-grams with n from 2 to 5 in the training set.

Each time a letter is guessed, I update both the dictionary of valid words and frequencies, frequency of valid n-grams that don't have incorrect letters that were guessed previously, and the vowel/length ratio.

I use four "safety nets" calculate the highest weighted, or most likely candidate letter. In the event that one fails, I move onto the next safety net. At each safety net I make the usual precaution regarding vowel frequency:

1) Across all training words that match the word-in-progress, calculate the most frequent letter and return it.

2) For all substrings in length from 3 to half of the length of the word being guessed, from the matching words in the training set, get the frequencies of each letter, and choose the most frequent out of all.
3) Find the largest valid n-grams in length from 2-5 surrounding each '_' index in the word-in-progress, and the most likely letter by frequency, giving priority to longer n-grams. Across all indices, choose the one with the highest frequency.

4) Return to the original training set, calculate frequencies, and return the most likely candidate letter.

3. Bidirectional Long-Short Term Memory

Hangman challenge by trexquant is a challenge to predict words, letter by letter.

##Vowel Prior Probability
Vowels: [a, e, i, o, u]
Created dictionary vowel_prior such that
vowel_prior = {}
keys: length of words
values: the probability of vowels given the length of words

Data Encoding
Input Data
Permutation:
From ~220k words dictionary, we have created around 10 million words by masking different letters in the word, i.e., by replacing letters with underscore.

The maximum length of a word in the given dictionary is 29. Testing will happen on mutually exclusive datasets. Thus max word length is assumed at 35.
Each input word is encoded to a 35-dimensional vector, such that alphabets {a-z} will be replaced by numbers {1-26} and underscore will be replaced by 27. The vector will be pre-padded.
Thus, the masked word "aa__" of the word "aaa" will be encoded as
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,27]

Target Data
For each of the masked input words, the output will be the original unmasked word. This word has been encoded into a 26-dimensional vector with each position representing letters of the alphabet from a to z.
Thus the output encoding for the word "aaa" will be:
[1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]

As the word contains only one letter "a", output encoding will have 1 at the first position.

#Modelling
Encoding + bi-LSTM has been built to train on this data.

#Prediction Strategy
It is required to predict the word within 6 incorrect tries.

1. Vowel Prediction: Leveraging Vowel_priors, we will guess the top vowel if tries_remains > 4 and len(guessed_letters) <= max_vowel_guess_limit

2. The remaining tries will be utilized by the bi-lstm model

3. The prediction will happen letter-by-letter

4. Recurrent Neural Network

Creating an entire Hangman game with an RNN involves multiple components, including data processing, model building, and game logic. Here's an outline of the steps you might take in Python using TensorFlow and Keras:

Data Preparation:
1. *Dataset Collection:* Obtain a dataset of words.
2. *Preprocessing:* Convert words into sequences of letters and create input-output pairs.

Model Building:
3. *Vectorization:* Convert letters to numerical representations.
4. *RNN Model:* Design an RNN architecture using LSTM or GRU layers.

Game Logic:
5. *Training:* Train the RNN model on the dataset.
6. *Hangman Game Loop:* Implement a loop to simulate the Hangman game.
   - Initialize the hidden word.

- Allow user input for guessed letters.
- Use the RNN to predict the next letter.
- Update the hidden word based on correct guesses.
- Display Hangman progress and guessed letters.

Idea that I have not used due to Shortage of time:

1. GRU LAYERS

Unfortunately due to shortage of time I can't perform the experiment GRU LAYERS.

I can guide you on how to create a flowchart depicting the Hangman game with a GRU (Gated Recurrent Unit) layer :

1. *Start*: Begin with a starting point indicating the initiation of the game.

2. *Initialization*: Initialize the Hangman game by setting up the word to be guessed and the number of attempts allowed.

3. *Input*: Prompt the player to input a character guess.

4. *Check Input*: Verify if the input is a valid character (alphabet) and whether it hasn't been guessed before.

5. *GRU Layer*: Represent the GRU layer where the neural network processes the current game state, including the known letters and their positions in the word.

6. *Prediction*: The GRU layer predicts the most probable next letter or a probability distribution over the alphabet.

7. *Update Game State*: Update the game state based on the prediction. If the predicted letter is correct, reveal its positions in the word; otherwise, decrease the number of attempts remaining.

8. *Check Win/Loss Condition*: Check if the word has been guessed completely or if the attempts have run out.

9. *End*: End the flowchart, indicating either a win or loss.

THANK YOU!!