# Hangman with ML

Spencer Gibson

## What’s hangman?

You try to guess a hidden word by guessing letters in the word.

If you guess a letter correctly, it’s occurrences in the hidden word are revealed

Otherwise, you lose one life. You typically get six lives.

Best way to get it is to play a few games

## What strategy do we use?

An effective strategy with “common” words is to guess vowels then fill in the consonants.

Guessing “E” first is probably the best choice for words with a few letters.

Playing hangman is pretty hard with less common words!

You need to rely more on linguistic patterns than typical word recognition.

## Statistical strategy

Common statistical strategy is the n-gram strategy.

N-grams: Words can be broken into pieces. “Hello” -> “He”, “el”, “lo” 2-grams or “Hel”, “ell”, “llo” 3-grams, etc.

In a partially revealed state, “he__o” we can split into pieces 2-grams around the missing letters to get “e_”, “_o”, match this with 2-grams in our training set and calculate which valid letter most likely fills in the gap.

Difficulty: doesn’t take much of a global view. Knowing WHERE a hidden letter lies in a word is really important. Letters at the end of the word, for example, tend to be consonants and are most likely to be E,S,T,D,N.

Not super clear how to use incorrect letters to aid guessing

Variants of this strategy typically get around 11 - 30% depending on how sophisticated they are and how large an “n” they use for their n-grams.

## First step - reinforcement learning

Hangman is a game where the agent is you or something else. Agent selects some letter, the action. After some steps you get feedback - you win or lose.

RL works by defining a reward function and giving the agent a reward through its play. Hopefully, the reward and the play give enough signal so that the agent can learn to play well.

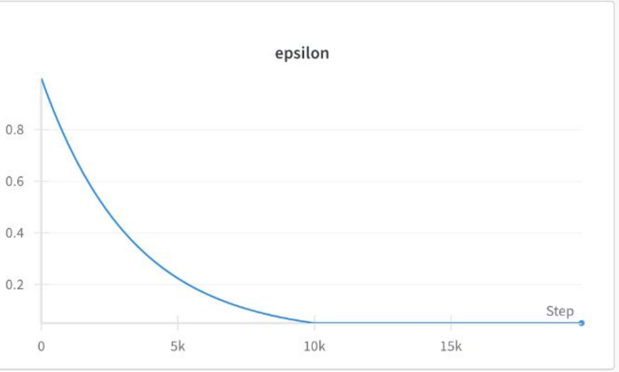
Sounds easy, can be super hard to make work.

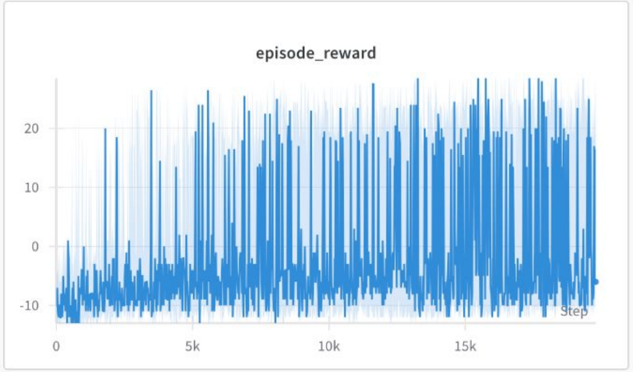
Why RL for this?

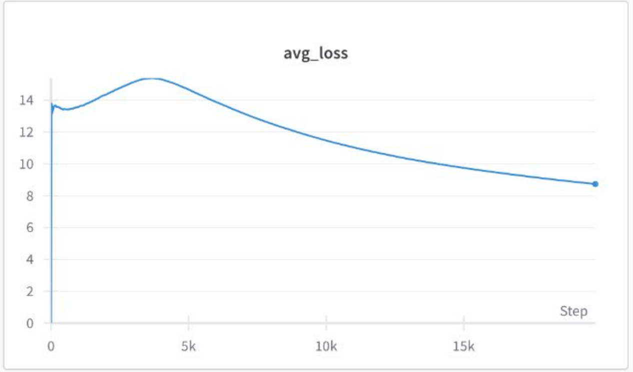
Hangman is a game. At each point in time, you might expect the best strategy to depend on the current state, incorrect guesses, and the amount of lives left. If you have more lives left, you might take riskier guesses, and reverting to safer ones later.

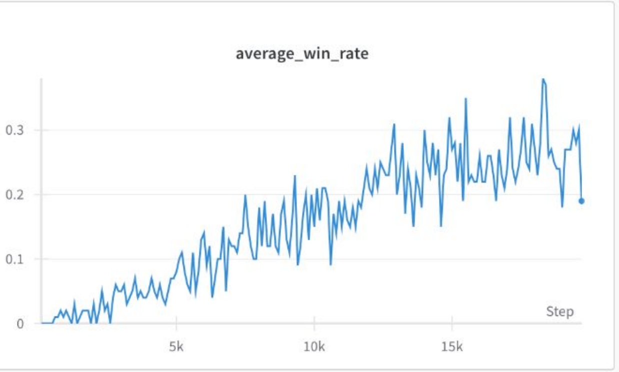
Not obvious what an amazing strategy is and collecting data is not really obvious, so let an agent figure it out!

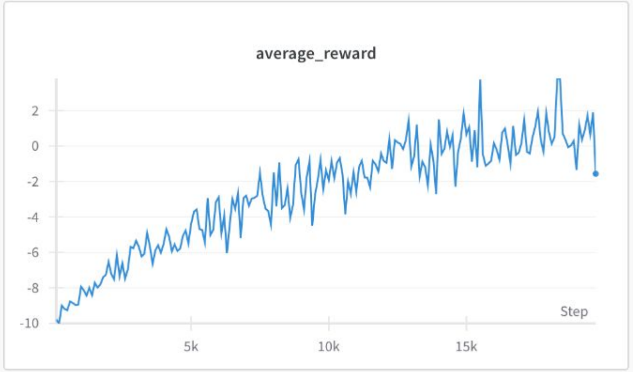
# First choice - DQN



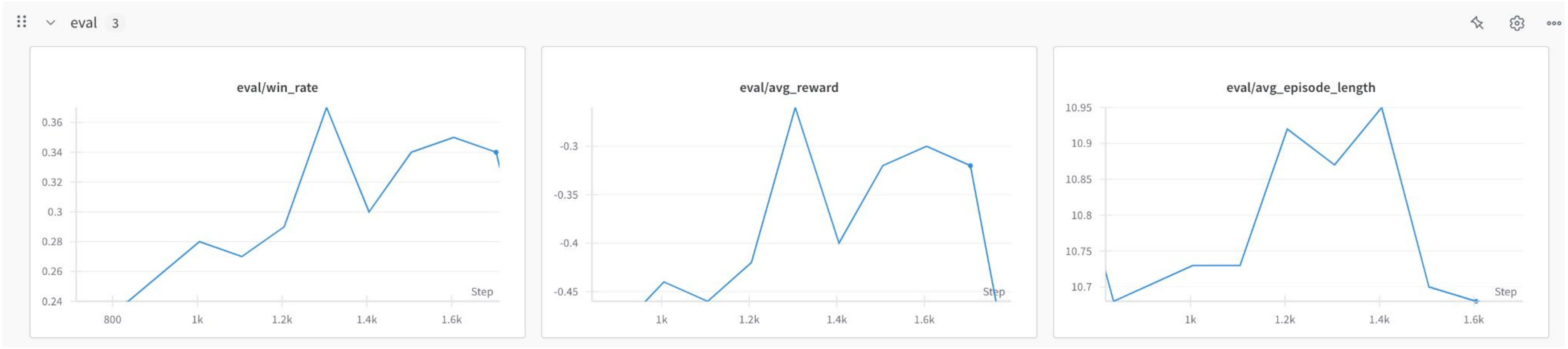








## Second choice - PPO with transformer



## Key issue with RL for Hangman (will come back to this)

Starting out, I really wanted an RL agent that could learn to play Hangman super well completely from scratch.

This didn’t really work, though.

The agent kept get stuck in shallow patterns - guessing letters by frequency without much linguistic sense.

Key problem - exploration. With 6 lives, most games were lost. The state space is pretty large (words could have up to length 30, possibly incorrect guesses is a bit vector of length 27, lives up to 6) and wasn’t explored very well.

Of course, most of the state space is nonsense, but that relies on linguistic knowledge.

Reward shaping: giving reward when the game is over makes it hard for the agent to learn which actions were good or bad. So you can give some reward at every step - this is called reward shaping.

Reward shaping often introduces undesired bias!

## Interesting ideas that DID NOT work

Warm starting the agent: letting it play with more lives (15 instead of 6) and gradually reducing the number of lives.

I thought maybe that would let it overcome the previous problem of not getting any positive reward, which may have caused the attribution problem, turns out that isn’t the case.

Revealing common letters in words and then playing. Don’t remember exact numbers but I don’t remember this doing very well either.

More lives is kind of like starting off with revealed common letters anyway.

Starting with shorter words and then moving to longer words. Shorter words were easier to learn, probably because the model can overfit to them.

## What’s hard here?

If we don’t know the word we’re guessing, or if the model doesn’t know where the word comes from, we need to extrapolate patterns from words we do know.

Let’s solve an easier problem.

Given that the word belongs to a corpus you have access to, how would you play?

## Contextual hangman

No longer need machine learning or statistical techniques. Regular old pattern matching (regex) works fine. Here’s one way:

Given a partially revealed word “he__o”, and incorrect guesses “j, k” do:

Filter from your list L all words with the incorrect letters “j” and “k”

Keep all remaining words that exactly match the revealed letters in the hidden word AND that do NOT contain extra instances of those letters, i.e. “app” shouldn’t be viable for “ap_”

Select the most frequently appearing letter among these words

## Does this strategy extend?

Well, not exactly…

It’s basically a hangman-based search for a word in a list. If the word’s not in the list, it’s not going to work!

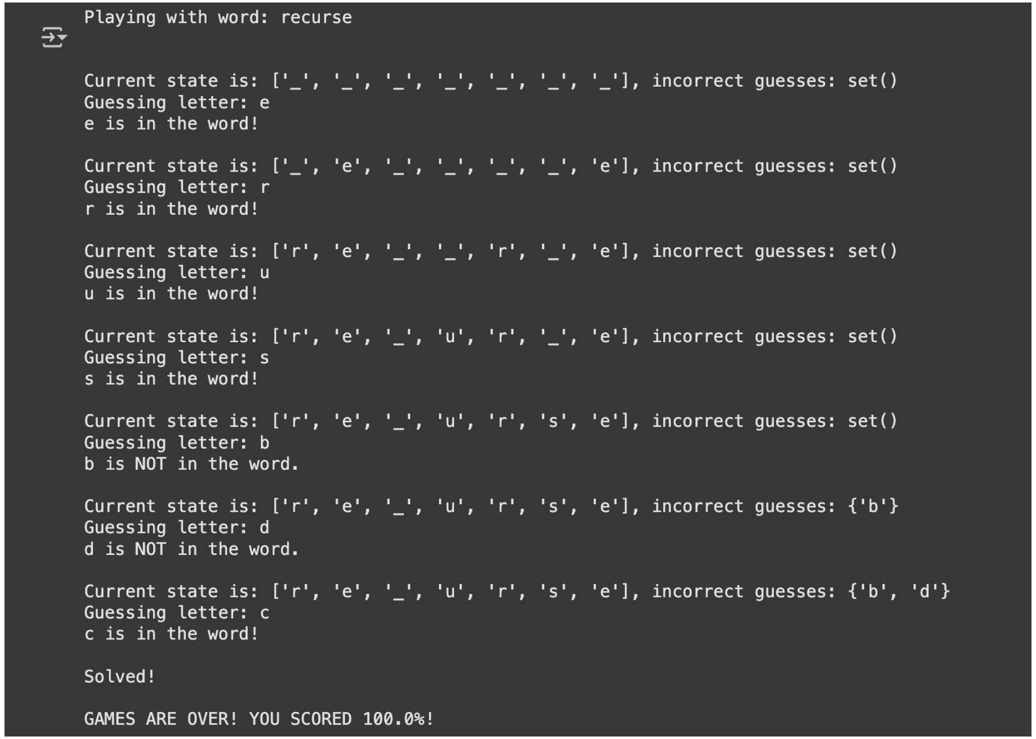
BUT, this search encodes a lot of linguistic knowledge because it uses incorrect guesses and revealed letter states to get the next most likely letter distribution!

SO, we can use this strategy on the training set, and for each step of our game encode the state (PARTIAL WORD, WRONG LETTERS, LIVES) and our manual strategy’s output (GUESSED_LETTER).

Use this to get a training set {(X_i, y_i)} for all training words!

## Training set

To give some examples of what the training set looks like, for the word “recurse” it would be:



```
Playing with word: recurse


Current state is: ['_', '_', '_', '_', '_', '_', '_'], incorrect guesses: set()
Guessing letter: e
e is in the word!

Current state is: ['_', 'e', '_', '_', '_', '_', 'e'], incorrect guesses: set()
Guessing letter: r
r is in the word!

Current state is: ['r', 'e', '_', '_', 'r', '_', 'e'], incorrect guesses: set()
Guessing letter: u
u is in the word!

Current state is: ['r', 'e', '_', 'u', 'r', '_', 'e'], incorrect guesses: set()
Guessing letter: s
s is in the word!

Current state is: ['r', 'e', '_', 'u', 'r', 's', 'e'], incorrect guesses: set()
Guessing letter: b
b is NOT in the word.

Current state is: ['r', 'e', '_', 'u', 'r', 's', 'e'], incorrect guesses: {'b'}
Guessing letter: d
d is NOT in the word.

Current state is: ['r', 'e', '_', 'u', 'r', 's', 'e'], incorrect guesses: {'b', 'd'}
Guessing letter: c
c is in the word!

Solved!

GAMES ARE OVER! YOU SCORED 100.0%!
```

## ML, finally…

ML is actually used :P

I trained a small, wide transformer model from scratch on this data using a v4 TPU provided through Google's TPU Research Cloud program (it's free, apply if you need TPUs!)

The model learns the training strategy really well! On a validation subset of 5000 words belonging to a corpus of about 479K English words (https://github.com/dwyl/english-words/tree/master) the model scores about 77% accuracy.

I revealed dashes ("-") in the hidden word, which maybe I should change because it does give a hint sometimes. Probably minor, though.

## Lessons

Please take with a grain of salt

Supervised ML (training a model with correct answers) can be better than RL even if the answers are only approximately correct. In my case, internalizing the contextual strategy in a model though supervised training way outdid all RL techniques I tried.

RL: exploration vs. exploitation trade-off can be difficult to make when you don’t have lots of resources. Many environments, including mine, are inherently a bit sequential so the best you can do is run a few in parallel. Bottlenecks between CPU and GPU or CPU and TPU communication are plenty.

## Lessons pt. 2

Test hypotheses in as controlled a way as possible!

It’s easy, and deceptively fun, to come up with plausible reasons for a phenomenon. Without solid, ideally isolated, evidence, it’s likely wrong!

Use Occam’s Razor as a guide. The simpler an approach is, usually the less assumptions it makes and the easier it is to both understand and test.

Complex explanations are hard because (1) they’re complex and (2) their assumptions are usually also complex! So they’re doubly complex.

Take with a grain of salt again because many hard problems require outside the box approaches.

Guess “E” first!

## Thanks!