UE23CS352A: MACHINE LEARNING

Hackman

Names : CHARAN M REDDY

CHARAN K

GAUTHAMDEV R HOLLA

DILEEP

CHANDRASHEKAR AWWANNA TELI

SRN : PES2UG23CS146

PES2UG23CS145

PES2UG23CS197

PES2UG23CS177

PES2UG23CS144

Section : C

Date : 3/11/2025

1. Key Observations

Most challenging parts:

The hardest thing was definitely getting the HMM to actually work. We tried using the hmmlearn library first and it just kept failing , couldn't train on any word length. Spent time debugging before We realized the library expects complete sequences, but We were trying to feed it masked words. We had to scrap that and build a simpler position-based model from scratch.

Another pain point was the state size explosion. Initially tried one-hot encoding for every letter position and the DQN couldn't learn anything useful. Training was slow and the win rate stayed below 30% even after 10k episodes.

The dimension mismatch error taught us to always print state sizes before building the network. Small mistake but killed 2 hours of training.

Key insights:

Pattern matching beats fancy ML for short words. When there's only 10-20 candidate words matching the pattern, just picking the most common letter works better than any neural network. The HMM and RL only help for longer/rarer words.

The reward function matters way more than I thought. Our first version gave +10 for correct guesses and -10 for wrong ones. Agent learned to just spam vowels. Changed it to +15 per letter revealed and -30 for wrong guesses, and suddenly win rate jumped from 40% to 65%.

Epsilon decay is tricky. Started at 0.3 and it dropped too fast - agent stopped exploring after 5k episodes and got stuck in local optima. We had to restart with epsilon=0.9 and slower decay (0.9997 instead of 0.995).

2. Strategies

HMM Design:
Gave up on traditional HMMs pretty quick after the library failures. Instead built something simpler - just count how often each letter appears at each position across all words of the same length.
For example, in 5-letter words:

- Position 0: A appears 12%, S appears 8%, etc.
- Position 1: A appears 15%, E appears 18%, etc.

Also tracked letter transitions (bigrams) - if the previous letter is 'Q', next letter is probably 'U'. This helped when we know some letters in the word.
Why separate models per length? Because letter patterns are completely different in 3-letter vs 12-letter words. Short words use common letters, long words have more variety.
RL State Design:
Went with a 95-dimensional vector:

- 15 numbers for the word (1-26 for known letters, 0 for blanks, -1 for padding)
- 26 binary flags for guessed letters
- 2 normalized values (lives remaining, word length)
- 26 HMM probabilities
- 26 pattern matching probabilities

Need to tell the agent:

1. What the word looks like right now
2. What letters are already guessed (don't guess them again)
3. How much danger we're in (lives left)
4. What letters are likely based on language patterns (HMM)
5. What letters fit the specific pattern (pattern matcher)

Tried simpler states first (just the masked word as a string) but the agent couldn't learn anything useful.
Reward Design:
Final rewards:

- +150 for winning
- -150 for losing (ran out of lives)
- +15 per letter revealed on correct guess
- -30 for wrong guess
- -20 for repeated guess

Why these values? Wanted the agent to:

- Prioritize winning over everything
- Avoid wrong guesses (they cost lives)
- Really avoid repeated guesses (pure waste)
- Value guesses that reveal multiple letters (like guessing 'E' in "E__E_E")

The multiplier on revealed letters was important. Without it, agent treated all correct guesses equally and ignored high-value letters.

3. Exploration

Used epsilon-greedy with decay:

- Start: epsilon = 0.9 (90% exploration)
- End: epsilon = 0.05 (5% exploration)
- Decay: 0.9997 per episode

But exploration isn't random - it's guided by HMM and pattern matching probabilities. When exploring, agent picks letters based on:

- 70% pattern matching probability
- 30% HMM probability

This is way better than pure random because we're exploring "smart guesses" instead of just random letters.

Also delayed training for first 500 episodes - just collected experiences and let the replay buffer fill up. This gave the agent diverse experiences before it started learning.

Target network updates every 300 episodes prevented the agent from chasing moving targets. Without this, the Q-values kept oscillating and training was unstable.

The combination worked well - agent learned basic strategies early (guess vowels first) then refined to context-specific choices as epsilon dropped.

4. Future Improvements

1. Word embeddings Instead of position-based encoding, use something like Word2Vec or character-level embeddings. Would capture semantic relationships better - "APPLE" and "APPLY" should have similar representations.

2. Curriculum learning Train on easy words first (high frequency, short length) then gradually introduce harder words. Right now it's learning from random samples which is inefficient.

3. Better pattern matching Current system just does exact pattern matching. Could add:

- Fuzzy matching for similar patterns
- Word family clustering (all -ING words, all -TION words)
- N-gram models beyond bigrams (trigrams would help)

4. Prioritized experience replay Some games teach more than others. Losses and close wins should be replayed more often than easy wins. Would speed up learning.

5. Ensemble approach Train 3-5 different agents with different hyperparameters and vote. Would smooth out individual agent weaknesses.

6. Adaptive strategy Switch strategies based on game state:

- First 2 guesses: always use frequency (E, A, R, I, O)
- Middle game: use pattern matching if >50 candidates, else use RL
- Last 1-2 lives: be super conservative, pick highest probability letter

7. Better state representation Current 95D vector is kinda redundant. Could use:

- Attention mechanism to focus on blank positions
- Separate encoding for word structure vs. letter probabilities
- Add a "difficulty score" based on remaining candidates

8. Longer training Only ran 25k episodes due to time. With 100k+ episodes and proper checkpointing, agent would probably hit 85-90% win rate.