Technical Strategy Write-Up

1. Problem Framing

- **Objective**: Build an agent that plays Hangman via a remote API, leveraging both reinforcement learning (RL) with deep Q-learning and periodic masked-language-model (MLM) fine-tuning to maximize win rate.
- **Constraints**:
  - Only the provided 250 K-word dictionary may be used.
  - External API enforces a limit of 20 new games/minute and distinguishes "practice" (unrecorded) vs. "recorded" runs.
  - Must slot into the existing notebook by overriding a single guess(self, word) hook.

2. Environment Design (

HangmanEnv

)

1. **State Representation**
   - A fixed-length vector encoding the current word pattern, e.g. _ A _ _ E → [_, A, _, _, E], padded to the longest word.
2. **Action Space & Masking**
   - 27 discrete actions (letters 'a'–'z' plus underscore).
   - Maintain a binary "legal mask" that zeroes out already-tried letters so the policy never re-guesses.
3. **Reward Structure**

    - **Correct guess**: +10 base reward + “bonus” computed from:
        - **Global frequency** of that letter in the *entire* dictionary.
        - **Relative frequency** of that letter among candidate words consistent with the current pattern.
        - Dynamically weight these two based on how many blanks remain (early guesses favor broad frequency; later guesses favor pattern-specific frequency).
    - **Incorrect guess**: –5 penalty and decrement a life.
    - **Repeated guess**: –2 penalty and decrement a life.
    - **Win bonus**: +50 when all letters are revealed.

3. Policy Architecture

- **Backbone**: bert-tiny pretrained transformer repurposed as a feature extractor on each masked word.
- **Head**: Two small fully-connected layers projecting BERT’s 128-dim “pooler output” to 27 Q-values.
- **Action Masking**: After producing raw Q’s, set illegal actions to a large negative constant so argmax ignores them.

4. Deep Q-Learning Agent

1. **Experience Replay**
    - Store transitions (state, mask, action, reward, next_state, next_mask, done) in a large ring buffer (~50 K entries).
    - Sample mini-batches of size 128 to break temporal correlation and stabilize learning.
2. **Double-DQN Updates**

    - **Online network** selects next action; **target network** evaluates its Q-value.
    - Periodically (every 500 episodes) copy online → target weights.

3. **ε-Greedy Exploration**
    - Start with ε=1.0 and exponentially decay to ε=0.01 over the course of training (decay span ≈ total episodes).

5. Hybrid MLM Warm-Up

- **Motivation**: Prevent catastrophic forgetting of letter-level language priors during RL fine-tuning.
- **Mechanism**: Every 1 000 RL episodes, perform a small number (≈20) of masked LM gradient steps on the training word list (character-masking at p=0.15).
- **Weight Sync**: Copy the MLM's updated transformer weights back into the RL policy's BERT backbone.

6. Training Loop

1. **Optional Head Pretraining**
    - (Separate CLI mode) Fine-tune BERT's MLM head on single words for ≈20 000 steps. Saves bert_output.pth.
2. **RL+MLM Cycle**

```
Initialize agent, target networks, replay buffer
For episode = 1..N:
  Reset HangmanEnv on training set
  Perform two "rule-based" guesses ('e' then 'a') to seed learning
  Loop until terminal:
    Select a via ε-greedy Q-policy
    Execute step, receive r, next_state, done
    Store transition; optimize Q-network on sampled batch
  If episode % 500 == 0: sync target ← online
  If episode % 1000 == 0: run 20 MLM batches → sync into online.bert
  Periodically evaluate on held-out words and checkpoint best Q-head
```

2.

3. **Hyperparameters**
   - Episodes: 20 000–50 000
   - Replay buffer: 50 000; batch size 128
   - γ=0.99; lr=1e-3; ε decay ≈ total episodes
   - MLM interval = 1 000 episodes; MLM steps = 20

7. Notebook Integration & API Flow

1. **Upload Artifacts**: best_qhead.pth, your pretrained_rl_hangman.py, and words_250000_train.txt into Colab.
2. **Load Policy**:
   - Instantiate prl.Agent with zero episodes for CLI args.
   - Load best_qhead.pth into agent.policy and set policy.eval().
3. **Patch the API**: Override HangmanAPI.guess(self, word) to:
   - Spin up a local HangmanEnv with the same dictionary.
   - Manually set env.word_state and env.tried_letters from the API's mask & guessed letters.
   - Call agent.select(state, mask) to produce your guess.
4. **Practice vs. Recorded**:
   - **Practice** (practice=1) uses unrecorded runs (up to 100 000).
   - **Recorded** (practice=0) logs your 1 000 final games for judge scoring.
5. **End-to-End Call Sequence**:
   - Notebook calls api.start_game(practice=…, verbose=…).

- API returns initial mask (e.g. _ p _ _ e).
- SDK invokes your guess hook → your RL policy picks the next letter.
- SDK sends /guess_letter → server responds with updated mask & remaining tries.
- Loop continues until win/lose.
- Stats fetched via api.my_status().

Outcome: By combining a frequency-aware reward shaping, rule-based “vowel seeds,” Double-DQN on a BERT backbone, and periodic MLM regularization, this agent substantially outperforms the ~18 % baseline and slots seamlessly into TrexQuant’s existing API notebook without modifying any of their networking code.