ChatGPT

Brainstorming Improvements for the Clash Royale Bot

Project Goals and Current State

This project aims to create a **fully autonomous Clash Royale bot** that can **learn and adapt** to consistently win games and climb the trophy ladder (targeting ~15k trophies). Achieving this means the bot should start and play battles on its own, improve its strategy over time through machine learning, and handle all in-game events without human intervention.

Current State: The existing bot is a Python-based reinforcement learning agent (using DQN) that plays via BlueStacks with computer vision (Roboflow) to detect game elements [1] [2]. However, there are known issues and limitations: - *Autonomy gaps:* Right now, the bot requires a hacky "keyburst" trigger to enter battle and requeue. It spams a key to click *Battle* and *Play Again*, which is a temporary workaround because the victory screen handling was not implemented [2]. This is not truly autonomous and can be unreliable. - *Learning performance:* The bot uses a basic Deep Q-Network for decision-making. DQN can play the game, but winning consistently at high trophies is very challenging due to the game's complexity (large state/action space, partial information, etc.). The current ML model may need improvements or even different algorithms to reach top-tier performance. - *Input and environment:* It relies on PyAutoGUI to simulate clicks and keystrokes on the BlueStacks emulator. This requires the emulator window to be in focus and at a specific position/size, which is fragile. Additionally, using the visual game output as state input (pixels or detected objects) makes learning slower compared to using higher-level game data.

Goal Breakdown:
1. **Full Autonomy** – From launch to gameplay to post-match, the bot should control the entire loop. No manual keyburst triggers; it should automatically press "Battle", play the match, recognize victory/defeat, and press "Play Again" to continue, indefinitely.
2. **Smart Learning and Adaptation** – The bot should improve over time, learning from each game. Eventually it should play well enough to climb into the highest trophy ranges. This likely involves a combination of reinforcement learning, better state representation, and possibly leveraging expert data or advanced algorithms to handle the game's complexity.
3. **Utilize/Improve Existing ML Models** – Preferably build on the current DQN-based framework and the pre-trained Roboflow detection models. However, if other approaches (different algorithms or model architectures) would dramatically improve performance, those should be considered as well. Essentially, we want the best of both: leverage what's already there *and* integrate new ideas for better results.

With these goals in mind, we will brainstorm solutions in two key areas: **(A)** making the bot **fully autonomous and robust** in handling the game interface, and **(B)** enhancing the bot's **learning strategy and AI model** for stronger gameplay.

Ensuring Full Autonomy in Gameplay

For the bot to be truly “hands-off,” it must reliably handle all the menu navigation and game flow on its own. This involves entering battles, playing them, and then dealing with post-game screens or any interruptions. Here are strategies to achieve full autonomy:

1. Automatic Battle Queuing and Replay Handling

Detecting and Clicking “Battle”: When the bot starts, it should detect the main battle screen (the screen with the “Battle” button). Instead of the current key-spam approach, implement an image recognition step to find the *Battle* button on screen and click it. This can be done via template matching (using a stored image of the button) or by leveraging the existing computer vision setup (for example, training a simple classifier or using the Roboflow API to detect UI elements). Once detected, the bot can programmatically move the mouse and click (or tap via ADB, discussed below).

In-Game Status Monitoring: While the battle is ongoing, the bot should monitor for the **end-of-game condition**. It can detect the victory/defeat screen by looking for distinctive visuals – e.g., the **“Victory”** or **“Defeat”** banner that appears, or the presence of the *“Play Again”* button. The README of the project notes that the bot currently doesn’t handle the *play-again* prompt correctly [2], leading to it getting stuck without the keyburst hack. Fixing this means adding logic such as: - If a “Victory” or “Defeat” text is detected on screen (or if the battle timer is over and crowns are tallied), then the match has ended. - At that point, locate the “Play Again” button. This could be through image search (since the button has a unique icon/text) or by known screen coordinates (if the game window is a fixed size and the button always appears at the same place). - Click the “Play Again” button to queue the next match. Possibly include a short delay after clicking, to allow the matchmaking to initiate.

By implementing the above, the bot will seamlessly loop into the next battle without human help. This removes the reliance on the temporary keyburst solution and ensures the bot truly runs autonomously match after match.

Handling Other Screens: Beyond victory/defeat, the bot might occasionally face other pop-ups or menus (for example, the “Chest is full” warning, level-up dialogs, or connection issues). To be safe: - You could add detection for common pop-up elements (maybe the “Close” X button) and have the bot dismiss them if they appear, so the bot doesn’t get stuck. - If the bot finds itself on the main menu (outside of battle) without intending to be (say after a rare disconnection or manual interruption), it should recognize that and navigate back to the battle screen. Keeping track of the game state (in-battle vs out-of-battle) via vision cues or timestamps can help recover from unexpected situations.

Summary: The aim is that as soon as you launch the bot, it **automatically presses Battle**, plays the game, then **presses Play Again** indefinitely – achieving a full battle loop on its own. This addresses point (2) in your list by removing the need for manual keyburst triggers and fixing the victory screen handling properly [2].

2. Improved Input Control (Switching from PyAutoGUI to Direct Control)

Right now, input is handled by PyAutoGUI, which moves the mouse cursor and clicks on the BlueStacks window. This works, but has drawbacks: the emulator must stay in a specific position and remain the active window, and any misalignment can throw off the clicks. A more robust method is to use a **direct**

Android input bridge. In fact, an open issue suggests using a *Pure Python Android Bridge* (likely via ADB) instead of PyAutoGUI [3]. Here's why this is beneficial:

- **Headless Operation:** By using Android Debug Bridge (ADB) or a similar direct input method, you can send touch commands to the emulator without needing to render it on screen or have it in focus. The issue discussion notes this would enable "headless" manipulation and eliminate problems with window placement [3]. In practice, this means you could run the bot in the background (or even multiple bots in parallel) without needing a monitor for each instance.
- **Precise Coordinate Taps:** ADB allows tapping at specific screen coordinates (`adb shell input tap x y`). You can map the game's coordinate system once (for a given resolution) and then send taps directly, which is often more precise than moving a PC mouse. This ensures the bot clicks exactly on the intended buttons or deploys cards in the correct locations.
- **Faster and More Reliable:** Without relying on the OS-level GUI, input injection is generally faster (no need to move a cursor visually) and cannot be thrown off by other windows or accidental mouse movements. It also means the bot won't accidentally lose control if, say, a notification pops up or if the BlueStacks window slightly moves.
- **Multiple Instances & Parallel Training:** Crucially, going headless with direct input opens up the possibility of running **multiple emulator instances simultaneously** [3]. Each instance can be controlled via a separate ADB connection (BlueStacks Multi-Instance Manager allows running multiple Android VMs). This means the bot could play multiple games in parallel, vastly increasing the rate of data collection for learning. Training with multiple emulators at once is mentioned as a benefit of moving away from PyAutoGUI [3].
- **Screen Reading via ADB:** Along with sending input, ADB can retrieve screenshots (`adb shell screencap`) from the emulator. The bot could use this to get the game's image data directly, rather than capturing the screen via PyAutoGUI. This ensures the image is always from the correct emulator and not affected by anything else on the host desktop. It also might be faster in some cases and can be done in parallel for multiple instances (each emulator can produce its own screen capture).

Implementation: Incorporating ADB might require some initial setup (e.g., enabling Android debugging on BlueStacks, connecting to the emulator via `adb connect` on the correct port). Once set up, you can use Python libraries (or subprocess calls) to send tap commands and capture screens. Libraries like **pure-python-adb** or **ADBKit** could help integrate this into the Python code. The end result is the bot will not rely on the GUI – it will **directly control the emulator**, making it much more resilient and flexible.

3. Resilient Timing and Flow Control

Autonomy isn't just clicking the right things – it's also **when** to click and ensuring the bot's actions are in sync with the game's state. A few things to consider:

- **Synchronization:** After clicking "Battle", the bot should wait until a match actually starts (e.g., look for the elixir bar or the arena view to appear) before trying to deploy cards. If it tries to deploy while the match is still loading, those inputs might be dropped. So adding a check for "match start" (perhaps the elixir bar turning from gray to active, or the timer appearing) is useful.
- **End-of-Match Delay:** Similarly, when a game ends, the bot should halt any further card deployment and focus on the end screen. Even if the RL agent wants to "take an action" right after the game ends, that action is meaningless – so the environment should signal to the agent that the episode is over. In implementation, once victory/defeat is detected, you would end the

current game loop, record the outcome for learning, and not issue any more clicks until the next battle is queued.

- **Handling Unusual Events:** If the network lags or the opponent disconnects (Clash Royale might show a "Opponent left, you win" or just slow down), the bot should ideally detect if the battle is effectively over or if it needs to keep playing. While these are edge cases, a truly robust bot would account for them – for now, a simple approach is to treat any scenario that leads back to the main screen as an end-of-episode and just queue again.
- **Avoiding Bans/Detection:** Running a bot violates the game's Terms of Service, so minimizing obvious bot-like behavior is wise. Rapid-fire clicking the exact same pixel at perfectly regular intervals, for example, could be a giveaway. Using ADB taps with slight random jitter or short random delays can humanize the interaction. Also, ensuring the bot doesn't run 24/7 without breaks (or occasionally uses the menu like a human) might reduce suspicion. This is more of a practical note if the account's safety is a concern.

By covering the above points, the bot will start matches on its own, play them, finish them, and restart – *completely autonomously*. This fulfills the requirement that **"as soon as I start it, it should be fully autonomous"** and addresses the temporary fixes previously used. With the environment loop solid and scalable (especially via direct control and possibly parallel instances), we can now focus on the core challenge: **making the bot smart enough to win consistently**.

Enhancing the Bot's Learning and Adaptation

Getting the bot to play *at a basic level* is one thing – teaching it to **consistently win** against skilled opponents is another. Clash Royale is a complex game: it has a huge state space, partially hidden information, and diverse strategies. A 2019 research paper on a Clash Royale AI noted the combination of **"huge state space, imperfect information, sparse rewards, and various strategies"** makes it very difficult for a single naive learning approach to excel [4]. In other words, we likely need to **augment the bot's intelligence in multiple ways**. Below we explore improvements in state representation, action selection, learning algorithms, and training methods to help the bot learn and adapt effectively.

1. Richer State Representation

How the game state is represented to the AI is fundamental. Currently, the bot likely uses the Roboflow computer vision models to identify troops on the field and cards in hand, then feeds some representation of that into the DQN. We should ensure the state information is both **comprehensive** and **structured** in a way that the model can learn from efficiently:

- **Include Essential Game Info:** At minimum, the state should include:
- Positions and types of all friendly and enemy units on the arena.
- Health of towers (friendly and enemy remaining tower health or whether they are up/down).
- Current elixir of the player (and maybe an estimate of opponent's elixir).
- Which cards are in our hand (and their identity, e.g., using one-hot encoding for each of the 4 cards).
- Time or phase of the game (e.g., normal time vs double elixir vs overtime). This can influence strategy significantly.

If any of these are not currently fed to the model, adding them will help. For example, if the model isn't aware of elixir, it might make moves that are impossible or suboptimal (like trying to play a card when

elixir is low). If it doesn't know the game timer, it won't realize it can play more aggressively in the final minute.

- **Structured Representation:** Instead of using raw pixels (which is high-dimensional and hard to learn from), we can construct a more abstract representation from the detected objects:
- One approach is to divide the arena into a grid and mark which units are present where. For example, break the field into, say, a 20x10 grid and encode each cell with who is present (friendly troop, enemy troop, or empty), possibly with type encoded by channels. This is similar to what some projects have done; in one PPO-based CR bot, they downscaled the field into small images and other features [5].
- Another approach is to use a **feature vector**: e.g., distances of each enemy troop to your towers, their HP, etc. However, designing such a feature vector by hand can be complex given the many unit types and possibilities.

- Using an **autoencoder** is an interesting option: one could feed a top-down view of the game state into an autoencoder network that compresses it into a vector, as was done in a research project [6]. That compressed vector can then be input to the decision network. This lets the AI learn its own compact representation of the battlefield.

- **Opponent Card Tracking:** Since Clash Royale is partly an imperfect-information game (you don't initially know the opponent's cards), the bot can benefit from tracking seen cards. After the opponent has revealed all 8 cards in their deck, the game becomes perfect-information in terms of card possibilities. The state representation could include flags or an encoded list of known opponent cards and even which cards have been played recently (to infer rotations). For example, a simple addition: an 8-length binary vector representing which opponent cards have been seen. Once a card is seen, that input stays "on" for the rest of the match. This helps the bot adapt its strategy if it knows, say, the opponent has a certain win condition or a counter spell.

- We could go further and include an estimate of opponent hand. However, predicting their exact hand order is complex (though theoretically possible with a tracking algorithm). A simpler approximation is the **Opponent Card Estimator** idea from a theoretical CRAI blog [7] [8] – essentially, incorporate knowledge like "if opponent just played a Golem, they won't have it available for a while (until it cycles and they have enough elixir again)". We can approximate opponent elixir and cooldown on cards if needed.

- **Use of Memory:** The game state is sequential by nature (what happened 10 seconds ago can matter now). Incorporating a temporal component can improve adaptation. This could mean:

- Feeding the network the last N states (frame stacking, like in Atari RL, though here the "frames" are game states).
- Or using a **recurrent neural network (LSTM)** that can maintain an internal state. An LSTM could, for instance, help the bot remember that the opponent has a certain card still unplayed, or that it used its big spell already. Many advanced game agents use recurrence to handle partial observability.
- Another form of memory is simply logging certain past events in the state. For example, a counter of how long since the opponent played their win-condition card, which could hint when it might come again.

Overall, **the goal is to provide the learning algorithm a digestible, information-rich picture of the game at each decision point**. By extracting features (like card identities, positions, healths, elixir counts) from raw pixels via the object detection, we make the learning problem much easier than if it

had to figure everything out from vision alone. This structured state will give any ML model a better chance to understand the game dynamics and make informed decisions.

2. Expanding and Discretizing the Action Space

Clash Royale's action space is quite complex: the bot must choose one of potentially 4 (or 8) cards, and then choose a tile on the arena to deploy that card. If treated naively, that's a huge number of possible actions (there are dozens of possible drop locations for each card). A key improvement is to **constrain and structure the action space** so the learning algorithm can handle it:

- **Card + Position as Action:** Instead of a monolithic "play card X at position (x,y)" action, break it into two steps or two decisions:
- Decide **which card** to play (or a decision to not play any card right now).
- Decide **where** to play that card (given the choice above).

This approach is sometimes called *action factorization* or *multi-discrete actions*. It simplifies learning because the network can learn about card choices somewhat independently of placement choices. In one multi-agent setup, they literally had separate networks for card selection and tile selection [9].

- **Discrete Tile Sampling:** Represent the possible placement positions in a discrete way. For example, you might define a set of allowable drop points rather than every pixel. One project divided the arena into **"origin" and "shell" tiles** – essentially a coarser grid of candidate drop locations [10] [11]. By doing so, they reduced hundreds of pixel positions to a smaller set of ~57 possible locations. We can adopt a similar grid approach:
- Perhaps define a grid of, say, 20x10 across the arena, but then filter out positions that are essentially equivalent or not useful (like you don't need two adjacent drop points when one would do). Ensuring some spacing as that project did (no two adjacent origin tiles) covers the field with minimal redundancy [12].

- Also account for card type: some cards (like spells) can be placed anywhere on the enemy side, others (troops) only on your side or river. We might need separate sets of positions depending on the card's allowed placement. To keep it simple, we can always restrict to valid placements: e.g., if the chosen card is a troop, the bot should only consider grid positions on its side of the arena (or on the bridge if allowed), etc. This can be coded as a mask on the action choices.

- **No-Op Action:** It's important to allow the bot to sometimes *do nothing* (wait) when that is the best course. In Clash Royale, a common strategy is to wait for full elixir or wait for the opponent's move. If our agent is forced to play a card at each time-step, it will never learn patience. Therefore, include a "no action" or "idle" decision in the action space (especially in a turn-based framing or if we step the environment every second or so). Implementation-wise, you can treat "no-op" as a special action in addition to playing one of the cards. The agent then has the choice each timestep to either play something or hold. The challenge then is to decide when the bot should make decisions – possibly on a fixed time interval (like every half-second it can choose to act or not). Alternatively, you could design the environment to only prompt a decision when there is at least 1 elixir available (so that a card *could* be played). This reduces needless decision points.

- **Action Filtering by Availability:** Obviously, the bot cannot play a card that it doesn't have elixir for or that isn't currently in hand. The environment or agent should handle this by masking out those actions. For example, if fireball costs 4 elixir but you have 3 elixir, the action "play fireball" should either be disallowed or result in a no-op if chosen. Ideally, during training, you mask

invalid actions so the agent doesn't get rewarded for learning to pick them. Many RL algorithms (like policy gradient methods) can incorporate action masks to avoid selecting illegal moves.

- **Macro-Actions (Sequences):** Another consideration is whether to allow the bot to play sequences or combos as a single decision (probably not necessary initially). But sometimes a strategy involves playing two cards in quick succession (e.g., minor + poison). Our agent right now would handle that by making two decisions in two time steps. That's fine, but it requires it to understand the synergy. This will likely come from learning if our reward mechanism and state allow it to see the benefit of combos. We probably don't need to hardcode anything for combos; just mentioning that the agent's policy should eventually capture those if it's truly learning optimal play.

Implementing Multi-step Actions: If staying with DQN, one way is to integrate the multi-step decision in the environment logic: - You could have the agent select a card (action 1 out of, say, 5 including no-op) in one step, then in the next immediate step, have it select a position (action 2 from, say, 50 possible tiles). Then combine these into playing the card. This effectively doubles the number of steps for one real move, which complicates learning because it has to coordinate two decisions for one outcome. - A more elegant way is to encode the action as a single composite token. For example, enumerate all card+tile combinations as distinct actions. This could be thousands of combinations, which is too many to learn reliably. But by constraining the tile choices and cards, the number can shrink. Suppose 4 cards in hand and ~50 possible tile positions -> 200 actions, plus 1 no-op = 201 actions. That's fairly large but maybe manageable for a neural network if many are masked out at any given time (e.g., if you have only 4 cards, only those card's related actions are valid). - The multi-agent approach (with separate networks or separate output heads for card and position) might be easier to train using policy gradient (as was done with PPO [9]). With DQN, it's tricky to have two-step outputs. However, you can cheat by having two Q-networks: one that picks cards (with knowledge of state) and another that picks position (with knowledge of state and maybe which card was chosen). But that second depends on the first. Alternatively, a single Q-network could output a matrix of Q-values (cards x positions). In theory, one could design a neural net that given state, outputs something like 4 (cards) × 50 (positions) Q-values. But maximizing over that in DQN would involve picking the highest Q for any card-position pair that's valid. This is doable, but implementing it is more complex than a standard flat output.

Given the complexity, a simpler starting point: **focus on card selection as the primary decision**, and use a heuristic for placement initially: - For instance, the bot could learn which card to play, and for placement you apply a rule like "place it at the safest spot on your side" or "if it's a building, place centrally; if it's a troop, maybe at the back to build push unless under attack". This reduces action space during initial learning. Once it can pick decent cards, you can open up the placement choices for finer optimization. - However, ultimately to reach very high trophies, the bot will need to master placement as well (e.g., precise kite placements, spell targeting, etc.), so we will have to implement that either via discretization or a learned model as discussed.

By taming the action space like this, we prevent the learning algorithm from flailing in an astronomical set of possibilities. The agent will more quickly learn *what type of actions are effective* (e.g., which card to play in a given situation) and then *where roughly to play them*. This structured approach was used in prior research and is likely essential for the bot to handle the game's complexity [10] [11].

3. Upgrading the Learning Algorithm

The current bot uses a Deep Q-Network (DQN). While DQN is a fundamental RL algorithm, more advanced methods might learn faster or achieve higher performance in a complex environment like this. We should consider improvements to the learning algorithm itself:

- **Enhanced DQN Techniques:** If we stick with DQN, integrate the known improvements from RL literature:
- *Double DQN*: This addresses the overestimation bias of Q-learning by using a separate network for action selection vs Q evaluation. It leads to more stable learning.
- *Dueling Network Architecture*: This splits the network's output into two parts – one estimates the *value* of the state, and the other estimates the *advantage* of each action. It can help the network learn which states are generally good or bad, independent of action, which speeds up learning in large state spaces.
- *Prioritized Experience Replay*: Instead of sampling past experiences uniformly, bias the sampling towards experiences that had high prediction error (surprising or informative experiences). In a game as sparse as CR (where a win or loss is rare feedback), prioritizing the moments that led to big tower damage swings or wins might improve learning efficiency.
- *Multi-step returns*: Using n-step Q-learning can sometimes improve credit assignment over long delays. For example, a 3-step return might help propagate rewards a bit faster in this long-duration game.

- Ensure a **large replay buffer** and possibly **slow target network updates** – since games are long, you want to retain a lot of history and update targets slowly to keep learning stable.

- **Policy Gradient Methods (PPO/A3C):** An alternative is to use **Proximal Policy Optimization (PPO)** or **Advantage Actor-Critic (A3C/A2C)**. These are policy-gradient algorithms that have shown great success in complex games (e.g., OpenAI Five for Dota, AlphaStar for StarCraft, etc.). In fact, one open-source project implemented a PPO-based agent for Clash Royale with some success [13]. Advantages of these methods:

- They can naturally handle continuous or large discrete action spaces (since they don't require maxing over Q-values, they just sample actions from a learned distribution).
- They often converge to better policies in complicated environments because they directly optimize for win probability (instead of Q-values).
- They can be combined with **LSTMs** to handle long time dependencies – PPO with an LSTM network could allow the agent to maintain memory of past states within a match, helping with things like predicting the opponent's next move.
- Multi-agent or multi-headed policies are quite feasible. For example, with PPO you could have one policy network that outputs two distributions: one for card, one for position. The training would encourage the joint decisions to maximize reward without enumerating every combination explicitly.

The downside is these methods are a bit more complex to implement and tune. They also typically require a lot of training data – though if we enable multiple instances and self-play, that's manageable.

- **Multi-Agent / Factorized Learners:** As touched on, we could treat the problem as multiple agents that cooperate:
- Agent A chooses the card, Agent B chooses the tile (and perhaps Agent C could be conceived to choose a second coordinate if needed, like the distinction some make between "origin" and

"shell" tiles [11]). In the PPO project, they had three agents working jointly (card picker, and two that together pick the position) [9].
- They share the same state input and could even share some layers of the network (to save computation and encourage a unified understanding). But each has its own output and policy.
- During training, you'd reward them jointly based on the outcome of the game, so they learn to coordinate. This is advanced, but it effectively breaks a huge decision into smaller ones that are easier to learn.

- If sticking to DQN, multi-agent is harder to implement, but one could use a *hierarchical DQN* approach: first level DQN picks a card (or high-level action), then given that choice, a second DQN picks exact placement. The second DQN's state input can include "what card was chosen by first DQN" as part of the state. The training would have to be done carefully (maybe train the second-level on data when the first-level is reasonably good).

- **Model-Based Planning:** Another possibility (though quite difficult here) is to incorporate a predictive model of the game – essentially simulate the game forward to evaluate actions (like MCTS or lookahead planning). The blog on CRAI mentions looking at **AlphaGo-like lookahead** [14] [15]. To do this, one would need a *Game Engine simulator* function that, given a state and an action, predicts the next state. Clash Royale's rules are complex but theoretically deterministic. However, writing a full simulator for CR is almost as complex as the game itself. Unless one could extract the game's logic, this might be infeasible. Alternatively, a learned forward model (a neural network that predicts outcomes of actions) could be attempted, but that's a whole project on its own and may not be reliable enough.

- If a forward model existed, one could use Monte Carlo Tree Search (MCTS) at decision time to try out various sequences of moves and pick the best – much like how AlphaZero works. But without a simulator, the next best thing is to rely on the policy's training to approximate that planning implicitly.

In summary, to reach top performance, **consider moving beyond vanilla DQN**. The current model can be improved incrementally with DQN tweaks, but exploring algorithms like PPO or a custom multi-agent RL might yield a more powerful AI. Notably, the research community had to use such advanced techniques to tackle similar RTS problems. For example, a multi-agent PPO with state abstraction was used to train a CR agent in another project [16], and academic research introduced a specialized model (SEAT: selection + attention) for CR [17] [4]. These indicate that a one-size-fits-all approach might fall short – instead, combining techniques is key (the IJCAI paper notes that **"heuristic methods, reinforcement learning methods and deep learning methods cannot work well alone"** for such games [18]).

Given your preference, a good path might be: **start by enhancing the DQN** (since it's already in place) for quicker wins, and in parallel prototype a PPO or actor-critic approach to see if it learns faster/better. We can also leverage imitation or supervised learning to boost whichever model we use, which we discuss next.

4. Training Strategies: Imitation and Self-Play

Reinforcement learning from scratch in a live PvP game is slow and potentially ineffective by itself – the bot might lose thousands of games while learning, which is not only time-consuming but also might not

converge to a high skill if the environment is too challenging initially. To address this, we can employ **imitation learning, self-play, or curriculum learning**:

- **Imitation Learning (Supervised Pre-training):** One way to jump-start the bot's skill is to teach it from human examples. If we have data of how good players play in various situations, we can train the bot's model to imitate those decisions. For Clash Royale, you could use replays from *TV Royale* or other sources:
- The game provides replays of top players (TV Royale) which show all actions both players take. If you can capture those (perhaps by recording the screen and running the same detection on them, or by using an API if available to get a list of moves), you could create a dataset: state -> optimal action.
- The Reddit discussion you provided also suggests this: *"You could use TV Royale videos + imitation learning to train an agent without self-play"* [19] . That indicates others have considered using the abundant replays of expert play as training data.
- Even a smaller scale, you personally (or some decent player) could play a few matches with the same deck and record the data to give the bot some "ground truth" to learn from. This could teach basics like "if enemy plays a hog rider, play Tombstone (building) to counter".

- After supervised training, the model would at least not be doing random things – it would roughly emulate a decent player's responses. Then you can switch to reinforcement learning to allow it to further improve beyond the human strategy or adjust to different opponents.

- **Self-Play Training:** In games like Chess, Go, and Dota, self-play has been a cornerstone of reaching superhuman performance. For Clash Royale, self-play means having the bot (or multiple instances of it) spar against itself:

- You could set up two bots on two instances of the game and have them battle each other. Since you control both sides, you can generate unlimited games without worrying about ladder rating or bans (you'd use friendly battles).
- Self-play allows the bot to learn even when it's not yet strong – it's essentially playing against a copy of itself, so as it improves, the opponent it faces also improves, maintaining a competitive learning environment. This was vital in systems like AlphaGo and OpenAI's Dota agents.
- A practical implementation: if using direct control (ADB) and multiple emulators, you can connect two emulators in a private match (Clash Royale allows friendly battles via clan or friend links). The bot code would need to handle two game instances in parallel and assign one as "maximizer" and one as "minimizer" from a learning perspective, or simpler, just treat one as the learning agent and use a fixed strategy for the other as a sparring partner initially.
- If not full self-play, even having a **rule-based bot opponent** for training can help. For example, script a simple opponent that plays random cards or follows a basic strategy (like always deploy the cheapest troop when available). Our learning agent can play many games against this scripted bot to learn core mechanics (like tower targeting, trading positive elixir, etc.). As it gets better and starts beating the simple bot easily, we can increase the opponent difficulty (this is a form of curriculum).
- One challenge: getting two instances to battle might require them to be friends or in the same clan. This can be managed by creating two accounts and ensuring they can always match (maybe use the clan training feature or custom tournament room).

- Self-play is computationally intensive (you're running potentially two game instances for each self-play match), but with headless mode and possibly lowering graphics, you might run several pairs at once. Each pair of bots could generate training data simultaneously – greatly boosting experience collection. As pointed out, going headless could allow multiple concurrent training games [3] .

- **Curriculum Learning:** Start the bot off in easier scenarios and gradually ramp up difficulty:

- For example, first train against the trainer or a scripted easy opponent until it learns to win consistently.
- Then let it play on ladder from low trophies; as it improves it will naturally face stronger opponents. (Ladder itself can serve as a curriculum since early arenas are easier and later arenas are harder with more cards unlocked.)
- If the bot gets stuck at a certain trophy range, you might introduce a new training phase, like self-play or go back to imitation on higher-level play to give it new ideas.

- Ensure the bot's deck/cards are competitive (reaching 15k trophies likely requires maxed level cards and a meta deck). If the bot's account has low-level cards, it might lose due to card level disadvantage rather than strategy – that confounds learning. So either provide a level playing field (tournament standard matches or a maxed account for the bot if possible).

- **Continuous Online Learning:** Once the bot is decent, you can have it continue to learn as it plays on ladder. For DQN, this means always adding new experiences to its replay buffer and periodically training on them (possibly in a background thread or between matches). For policy gradients, you could run mini-updates after each game or batch of games. This way the bot adapts to the current meta. If a new card is introduced or a new dominant strategy arises, the bot should gradually experience it and adjust its policy. This addresses the *adaptation* part – the bot isn't fixed; it keeps learning from each loss. Over many games, it might discover strategies to beat things that initially confounded it.

- However, be cautious with continuous learning: if the bot starts to "forget" earlier knowledge (called catastrophic forgetting), its performance might oscillate. One way to mitigate this is to keep a portion of older experience in the training mix (for DQN, the replay buffer naturally does this; for policy gradients, you might occasionally train against snapshots of its old self or keep some imitation loss to not stray too far).
- Monitoring the bot's performance over time is key – you can track its win rate or trophy count trend. If it plateaus, that might indicate it needs new ideas (maybe policy architecture change or more training data or different exploration).

5. Hybrid of Rule-Based and Learning-Based (Getting "Both")

You mentioned wanting both – presumably both smart ML and some form of stable performance. Indeed, a pure RL bot might do silly things early on (or even later, it might find weird strategies that occasionally fail spectacularly). Combining rule-based logic with ML could yield better results, especially in the interim: - **Safety Constraints:** We can embed basic rules that the bot must follow, to prevent obvious blunders. For example, a rule could be "Don't play a high-cost card like Golem when the opponent has just rushed the other lane – defend first." These could be hardcoded triggers that override the ML decision if certain conditions are met. Essentially, the ML proposes an action, but the rule-based system can veto or modify it if it violates common-sense Clash Royale principles. This ensures the bot doesn't make beginner mistakes even while learning. Over time, as the AI gets better, these safety rules might be relaxed. - **Heuristic Guiding Rewards:** Instead of outright rules, you can encode knowledge in the reward function. If you know certain behaviors are desirable, give a small reward for them. For instance, *if the bot successfully counters an enemy card with an appropriate response (like using Minions to kill a Hog Rider without the Hog doing much damage)*, give an extra reward. This requires being able to detect that event, which might be complex to program. But simpler: give a negative reward for wasted elixir (sitting at 10 elixir without playing) to encourage not overflowing, or a small positive reward whenever the bot scores damage on the enemy tower (which encourages

offense). - **Hardcoded Opening or Sequences:** Many strategies in CR have an "opening move" or standard sequence (like cycle a cheap card at back, or wait for opponent). The bot could use a scripted opening strategy (perhaps chosen at random from a few good ones) to avoid the awkward initial phase where neither side plays – or worse, the bot plays something foolish. Once the battle progresses or certain conditions occur (enemy plays a card), switch control to the ML policy. This is analogous to how some chess engines use an opening book. - **Deck Selection Adaptation:** Outside of a single match, an advanced consideration is adapting the *deck* or card upgrades. Reaching 15k might require not just good play, but also using a top-tier deck. The bot could potentially learn to suggest deck changes (this goes beyond your current scope, but a "God mode" CRAI might even consider optimal deck building [20]). For now, ensure the bot is using a well-rounded, strong deck so that learning isn't happening on an uphill battle due to a suboptimal deck.

One must be careful not to hardcode too much, as it can limit the ceiling of the AI. The ideal scenario is the rules provide a scaffold that prevents disastrous play while the AI is learning; as the AI becomes competent, it can override or ignore those rules because it has learned a better policy. For instance, you might start with a rule "always respond to an opponent's win-condition play with a defensive card" – later the AI might learn sometimes a base-race (counter pushing other lane) is more effective. The rule would have to eventually be removed or made advisory.

A successful hybrid might involve an **ensemble or arbitration**: have a rule-based agent and an ML agent both "propose" an action. If the ML agent's choice conflicts with basic heuristics (like it wants to spend 7 elixir on offense when the opponent dropped a big push on the other side), the rule-based agent might modify that. There are research ideas like **reward shaping via advice**, where the rules gently nudge the ML by shaping rewards or adjusting Q-values, rather than a hard override.

Ultimately, the **long-term goal** is the ML becomes so good that it inherently follows optimal principles (thus it implicitly does what rules would say, but even better adapted to context). In the short term, combining both can yield "the best of both" – the immediate competence from rules plus the long-term adaptability of learning.

6. Adaptation and Meta-Strategy

To climb very high in trophies (towards 15k), the bot must handle a wide variety of opponents and strategies, and even adapt **within a match** to what the opponent is doing. A few ideas on enabling adaptation:

- **Opponent Modeling:** As briefly mentioned, track the opponent's deck and tendencies. The bot could maintain counts like "opponent played card X, how often do they play it at the bridge vs the back?" etc., but that might be too detailed. Simpler: once the opponent's deck is known, the bot's strategy might shift:
- If the opponent has a very expensive deck, maybe apply constant pressure to exploit their slow cycle.
- If the opponent has all air troops, the bot should ensure it saves some anti-air card at all times.
- These strategic adjustments can be learned (if the state includes the opponent's known cards, the neural net can implicitly figure some of these out). But we can expedite it by occasionally simulating how certain opponent archetypes should be handled and maybe giving the bot some "lessons" (imitation again, e.g., show it replays of how to beat a specific deck).

- Another approach is to have multiple sub-policies in the bot: like a defensive mode and an offensive mode. The bot could sense "am I ahead or behind?" or "what's the opponent's style?"

and switch modes (this could either be a learned switch or a simple rule: e.g., if ahead by 1 tower, play more defensively).

- **Meta-Learning:** This is advanced, but meta-learning algorithms would have the bot *learn how to learn* from each game. For example, after each loss, it might adjust some parameters quickly to do better next time against that strategy. This typically isn't done in game bots (because training is separate from execution), but conceptually an online learning rate can allow the model to update after each game on the fly. If using something like an LSTM with a long-term memory, it might even adjust its play style across a best-of-n series (though ladder is always one-off games, except global tournaments).

- **Evaluation and Continuous Improvement:** Always test the bot in various scenarios to find weaknesses. If the bot struggles against, say, siege decks (like X-Bow), then focus training or hard examples on those: have it practice specifically against a strong X-Bow bot or feed it replays of how to counter siege. This targeted training will improve its adaptability.

- **Reward for Winning Trophies:** A final note on the objective: the bot's fundamental reward should correlate with winning. In ladder, winning gives trophies (except at the cap), but also each match is win/lose. Our RL reward can simply be +1 for win, -1 for loss (or +some large value for win). We might even tie the reward to trophy change (like +30 * (trophies gained) for a win, which is basically +30 at most, and -30 for a loss, etc.), but since trophy gain is directly tied to win/loss, it's similar. The key is that ultimately the bot should care about *winning the game*, not just doing well. All intermediate rewards should be shaped in service of that end goal. The reason to emphasize this is to ensure the bot doesn't learn to, for example, farm damage for reward while neglecting actually closing out games. Keeping a large final reward for victory ensures that if it finds a way to truly win (even by making short-term sacrifices), it will get the highest reward. The intermediate shaping is just to guide it along the way.

By implementing the above enhancements, the bot should gradually evolve into a much stronger player. Initially, improvements in state representation and action space will let it **make more sensible decisions**. Upgrading the learning algorithm and using more data (imitation or self-play) will allow it to **learn faster and smarter**, picking up on deeper strategies. And incorporating some domain knowledge and adaptive logic will help it **handle different opponents and scenarios**, which is crucial for climbing to top ladder where one might face all kinds of decks.

Using Current Models vs Exploring New Ones

You expressed a preference to use the current ML models (likely the DQN and the detection CNNs). We can absolutely start with those and incrementally improve, but it's worth comparing with other approaches to ensure we're not missing out on something significantly better:

- **Current Model (DQN with CNN detection):**
- *Pros:* Already implemented, we can build on it. Off-policy learning (DQN) means it can reuse past game data and continue learning from it, even if the policy changes. DQN is simpler to debug in some ways.
- *Cons:* DQN struggles with very large action spaces and long delays before reward. A Clash Royale match might last 3-5 minutes, and a critical mistake in the first minute might only be felt at the end (sparse, delayed reward). Standard DQN might have trouble assigning credit properly. Also, DQN's epsilon-greedy exploration can be inefficient in such a vast decision space – it might rarely stumble on a winning strategy by chance if the space is too big.

- *Improvements:* We should definitely integrate Double DQN, etc., as mentioned. Also consider using *Rainbow DQN* (which combines many improvements like double, dueling, prioritized, multi-step, distributional RL). Rainbow is a state-of-the-art extension of DQN that could be suitable if we go the value-learning route.

- **Convolutional Neural Nets (for detection):** The current object detectors for troops and cards are presumably CNN models (maybe YOLO or similar) provided via Roboflow. If they work well, great. If not, improving them is another avenue:

- Ensure the detection models are accurate and fast. Any mis-detections will feed wrong info to the bot. You might consider retraining or fine-tuning them with more images (especially if certain troop types aren't recognized well in the heat of battle).

- Alternatively, you could replace them with a single end-to-end model that directly outputs a structured game state from the raw image (this is complex and basically re-does what Roboflow is doing, so probably unnecessary given you already have those models).

- **Considering New Algorithms (PPO or Others):**

- The PPO approach (with possibly multi-agent) as used by Jaso1024's project [16] has the appeal of being designed for exactly this kind of problem (complex real-time decisions). If DQN plateaus, trying PPO might break through to better performance. It's known that policy gradient methods can sometimes handle the stochastic, continuous nature of such games better than Q-learning.
- There's also **Deep Deterministic Policy Gradient (DDPG)** or **SAC** if we treated placement as continuous coordinates. For instance, one could frame card placement as continuous (x,y) and use an actor-critic that outputs continuous values. But given the discrete nature of cards and the complexity, this might not be any simpler than discrete approaches.

- **AlphaZero-like Self-Play with MCTS:** This is the gold standard for games like chess and was attempted for some RTS as well. If we had unlimited computational resources and possibly a way to speed up game simulations, one could train an AlphaZero style agent: self-play games, use MCTS to guide policy improvement. However, without an internal model of the game, doing MCTS on a live game is not possible (you can't forward simulate the actual game without just playing it out in real time). So we likely stick to model-free RL.

- **Evaluate "Much Better" Approaches:** It's worth noting that even the best known research approaches for Clash Royale (or similar RTS) have not produced an AI that can beat top human players in the *real game environment*. The SEAT model in the paper was tested against some rule-based AI, not necessarily against top human players [21]. This means we are in relatively uncharted territory aiming for 15k trophies (which is extremely high, likely top 0.1% of players). We'll need **all the tricks**: a powerful learning algorithm, lots of training, and possibly some advantages like being able to react faster than a human (though CR has a limit on how fast you can play due to elixir).

- **Resource Considerations:** Using advanced models and running many training battles will require a good amount of compute. If that's a constraint, we might lean on current models more initially and optimize them. If compute is available (multiple GPU for training, etc.), then going for a more complex model could pay off in the long run.

Recommendation: Start by implementing the easier improvements (autonomy fixes, better state encoding, reward shaping) with the current DQN approach to see a baseline improvement.

Simultaneously, prototype a PPO or actor-critic model on the same environment and maybe use self-play in a controlled setting (even a simplified version of the game) to see how fast it learns. We might find PPO learns certain tactics much faster than DQN. If so, we can migrate the bot to that framework for ladder play. The current detection models and environment interface would be reused in any case, so that effort isn't wasted.

The key is to remain flexible: **if a different ML model promises significantly better performance, be ready to adopt it**. For instance, if you find that the bot's winrate stops improving with DQN even after many games, that might be the time to try a policy gradient approach. Or, if you gather a large dataset of expert games, you might decide to train a supervised neural network that plays at a high level without explicitly doing RL at all (behavior cloning), then fine-tune it with RL.

Ultimately, to reach the very high trophy counts, we expect needing a combination of approaches – leveraging human knowledge (imitation), self-play, and strong RL algorithms. This is aligned with how cutting-edge game AIs are built (they are rarely purely one technique). The **good news** is that each of these techniques can be integrated incrementally: you can keep improving the bot piece by piece.

Conclusion and Next Steps

We've explored a wide range of improvements and possibilities for the Clash Royale bot, addressing both **autonomy** and **intelligence**:

- **Full Autonomy:** By using image recognition or direct emulator integration, the bot will handle starting battles and replaying automatically. Fixing the *"Play Again"* issue is a priority [2]. Adopting a direct input approach (ADB bridge) will make the bot more robust and even allow scaling up (multiple instances, faster data collection) [3]. These changes ensure the bot can run continuously without intervention, an essential foundation for training and climbing trophies.

- **Smart Learning & Adaptation:** We will refine what the bot "sees" (state representation) and how it decides actions (action space), so it has the right tools to learn effectively. Then we'll improve the learning algorithm itself – starting with enhancements to DQN and potentially transitioning to more powerful methods like PPO with multi-agent outputs. The training regime will combine reinforcement learning with possible supervised boosts from human replays [19] and self-play to accelerate skill acquisition. We'll also incorporate domain knowledge through reward shaping and safety rules to guide the learning process. All these measures align with known strategies in AI research for complex games, acknowledging that **complex games like CR require a hybrid of techniques** for the best results [4].

- **Climbing to 15k Trophies:** This is an ambitious target. It will likely require a **massive amount of training** and perhaps the emergence of non-intuitive strategies (the kind of thing a well-trained AI can sometimes discover). By continuously training and adapting, especially using self-play and ongoing learning, the bot can keep improving its win rate. With each improvement (better state info, better model, more data), we should see it climb higher. Reaching the top tier may also require that the bot effectively reacts faster or more precisely than human opponents – our autonomous input (with no slip-ups) and potentially faster reaction times could be an edge. If it learns optimal plays, it might eventually outpace human reflexes in card deployment.

- **Potential Challenges:** We should be realistic that 15k trophy range is extremely high; human top players have deep game knowledge and mind-games that are hard to replicate. There may be strategies involving predicting opponent behavior that are tricky for an AI. However, with

enough self-play, the AI might develop its own formidable tactics. We should monitor its performance and identify failure modes – e.g., does it handle split lane pushes well? Does it know how to play out a draw if needed? Each time we spot a weakness, we can adjust the training (add a scenario or tweak the reward or input) to address it. Over time, these targeted fixes will iron out the flaws.

Next Steps: 1. **Implement Autonomy Fixes:** Remove the keyburst and integrate a proper loop for battle start/end using image detection or ADB. Test the bot playing multiple games in a row hands-free. 2. **Integrate ADB Control:** Set up the Pure Python Android Bridge approach to send taps and get screenshots. Verify that we can run two instances with independent control. 3. **State & Action Refactor:** Modify the `env.py` (environment) to provide a richer state (incorporating elixir, tower health, etc.) and to handle a discretized action space (with possibly two-stage actions or a combined action ID). This likely includes writing code to translate an action (like action ID 37 might correspond to "play card 2 at tile 5") into the actual game command (choosing the card slot and tapping the coordinate). 4. **Reward Shaping:** Define the reward function with intermediate rewards for tower damage, tower destruction, etc., on top of the final win/loss reward. This might be adjusted and experimented with to find what leads to best learning. 5. **Train and Tune with DQN:** Run training with the improved DQN (double, dueling, etc.) in a controlled setting (maybe start with some easy opponent or even self-play). Check if the bot's performance improves over time (e.g., win rate goes up against a baseline opponent). 6. **Explore PPO/Actor-Critic:** In parallel, possibly set up a PPO training loop using the same environment. Even a simplified version (1v1 against a fixed strategy) to see if it learns faster. This will give insight into whether switching algorithms is worth it. Keep an eye on the community or any updates in the GitHub issues for others working on similar enhancements. 7. **Imitation Learning Data:** If feasible, start collecting replay data. For example, use the bot's detection on a few recorded top-player games to get state-action pairs, and see if training on that helps the bot's initial policy. This could be a separate script to avoid complicating the main training loop. 8. **Testing on Ladder:** Once the bot shows competency in training scenarios, unleash it on ladder (perhaps on a secondary account to be safe). Observe its trophy progression. This will be the true test – see how it deals with the variety of decks and situations. Expect it to struggle at first; it might oscillate or slowly climb. With continuous training enabled, it should gradually improve. Logging its matches (states/actions/outcomes) will help analyze where it fails. 9. **Iterate:** Use the insights from ladder tests to drive further improvements. For example, if the bot often loses because it overcommits in x2 time, adjust the reward or input to make it aware of that. If it misplays certain high-level strategies, maybe add those scenarios to self-play or training.

By iterating through these steps, we'll be steadily pushing the bot's capabilities. Each component we brainstormed feeds into the ultimate goal: an autonomous, intelligent agent that **learns from experience and consistently competes at a top-tier level**. It's a challenging road (combining robotics, vision, and advanced AI in a live game environment), but following this comprehensive approach gives the best shot at achieving the 15k trophy bot dream. Good luck, and enjoy the process of seeing your bot evolve into a champion!

Sources:

- Krazyness (Andre Nijman), *Clash Royale Bot* – Project README and issues (discussing current limitations like replay handling and input control) [2] [3].
- Jaso1024, *Real-Time-Strategy-RL-Clash-Royale* – Description of a PPO multi-agent approach to Clash Royale (for action space factoring and reward shaping) [9] [22].
- Reddit r/MachineLearning discussion – Ideas on difficulty of pure RL in CR and suggestion of imitation learning from expert replays [23] [19].
- Arun Patro's *Thoughts on CRAI* – Theoretical considerations for a Clash Royale AI, emphasizing state modeling, lookahead, and opponent modeling [24] [7].

- Liu et al., *IJCAI 2019 – Playing Card-Based RTS Games with Deep RL* – Highlights the challenges of Clash Royale for AI (huge state space, hidden info, sparse reward) and introduces the SEAT model (card selection + placement attention) [4] [17] .

---

[1] [2] GitHub - krazyness/CRBot-public
https://github.com/krazyness/CRBot-public

[3] use Pure Python Android Bridge instead of PyAutoGUI to send input to the emulator · Issue #6 · krazyness/CRBot-public · GitHub
https://github.com/krazyness/CRBot-public/issues/6

[4] [17] [18] [21] Playing Card-Based RTS Games with Deep Reinforcement Learning
https://www.ijcai.org/proceedings/2019/0631.pdf

[5] [6] [9] [10] [11] [12] [13] [16] [22] GitHub - Jaso1024/Real-Time-Strategy-RL-Clash-Royale: A Reinforcement Learning agent for Clash Royale created using Proximal Policy Optimization
https://github.com/Jaso1024/Real-Time-Strategy-RL-Clash-Royale

[7] [8] [14] [15] [20] [24] Thoughts on building a theoretical Clash Royale AI | Arun Patro
https://arunpatro.github.io/blog/crai/

[19] [23] [P] Clash Royale Build A Bot : r/MachineLearning
https://www.reddit.com/r/MachineLearning/comments/soop8v/p_clash_royale_build_a_bot/