Probabilistic Modeling of Human Actions for RTS Games with Limited Reinforcement Learning Feasibility: A *Clash Royale*™ Case Study

Cooper Dean [1] **Christian Reynolds** [1] **Diego Peña-Stein** [1] **Omkar Tikar** [1] **Rowan Faulkner** [1]

Abstract

In this paper, we explore using human replays as a means to train a deep learning model to play the popular real-time strategy game *Clash Royale*™. This work is inspired by previous reinforcement learning approaches, which we believe are not scalable for *Clash Royale*™, due to its extremely large card selection, number of possible deck combinations, lack of high-speed API, and difficult-to-identify rewards. We find that our imitation approach is functional but limited in performance when trained on a small dataset, but shows promise if it could be scaled up. We present our findings and takeaways from the entire process, from data collection and label extraction to training and deploying a model to play against real players.

1. Introduction

Clash Royale™ is a one-on-one real-time strategy game where each player places units and spells on a battlefield to destroy the opposing player's towers. It involves quick decision making, precise placements, and careful timing, making it an interesting game to try and tackle with deep learning. In this paper, we demonstrate an end-to-end pipeline for training and deploying an AI agent to play *Clash Royale*™.

We propose using a more efficient approach to train a deep learning model. Instead of learning to play from scratch, we show that human replays can be used to train a model to imitate human behavior in a much smaller amount of time. In theory, this pre-trained DL model could be used in conjunction with reinforcement learning techniques, enabling transfer learning.

[1]Worcester Polytechnic Institute.

CS/DS 541 Deep Learning Final Project, Worcester Polytechnic Institute.

1.1. What is *Clash Royale*™?

In a game of *Clash Royale*™, each player has three towers which they have to defend, while simultaneously attacking their opponent's towers using cards that they place on the arena. Each player holds four cards in their hand at any given time, which they can play by paying an "elixir cost". Each card has an "elixir cost" which is displayed at the bottom of the card. Players accumulate elixir over time, up to a maximum of 10, which they spend to play cards according to their cost.

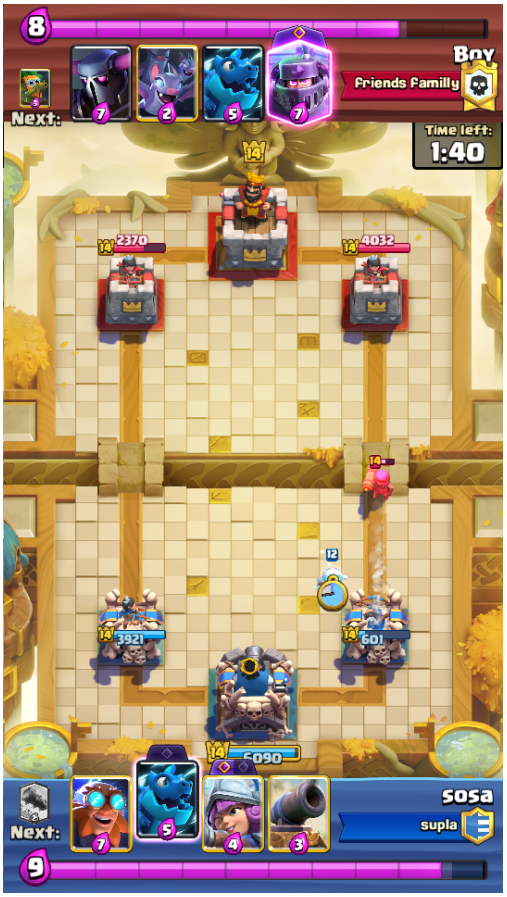

Figure 1. A screenshot from a *Clash Royale*™ replay that shows the towers, cards, and elixir bars

Each player brings a deck of eight unique cards to the battle. Each card serves different defensive and/or offensive purposes and can be used in combination with each other to create powerful synergies. Cards can be divided into three distinct categories: units, buildings, and spells. Units advance toward the enemy towers. Buildings stay in one place and draw units to them. Spells can be played anywhere on the arena, defensively or aggressively, and even deal damage directly to towers. Generally, it is suggested to include a combination of units, buildings, and spells in most decks.

Figure 2. Example deck including one building, three spells, and four unit cards

(RoyaleAPI)

1.2. Research contributions

Many attempts to train machine learning models on real-time strategy games use reinforcement learning in order to learn some optimal policy given the game and a reward function. Our research shows that recording and approximating player input as probability distributions given the game state can be a much better alternative in situations without well defined reward functions or when training in game is expensive.

2. Related Work

Previous attempts to create *Clash Royale*™ bots have used reinforcement learning (Brody, Dai). The issue with this approach is *Clash Royale*™ does not provide any means by which to speed up gameplay for training purposes, so this is an extremely time consuming process. Existing models play with little interaction with the other player or apparent strategy. This is likely because any reward function (towers taken/lost) using available on-screen information is too far removed from actual gameplay to teach the model anything useful. Additionally, the reinforcement learning approach limits the model to only train on a given deck of 8 cards, crippling any generalization to the 150+ existing cards.

3. Proposed Method

In order for a machine learning model to play *Clash Royale*™, it has to make multiple choices in concert. It must decide when to act, which card to play, and where to play it. Our model learns the probability distribution of each of these three choices using a three-headed deep neural network which has an ”action” head, ”card” head, and ”placement” head.

Formally, we want to estimate $P(A|G)$, where A represents the action of playing a card and G represents the state of the game (428x683x3 tensor representing screen capture, scalar representing current elixir, and binary encoding of current hand), $P(C_h|G, A)$, where C_h represents the action of choosing some card h, and $P(R_{x,y}|G, C_h, A)$, where $R_{x,y}$ represents the action of placing a card on a particular tile (x, y).

Each of these probabilities depends on the output of the previous decision. In practice, each head produces output independently of the others, which we combine after the forward pass to form the final decision. For the card head, this is trivial, since its output is not considered when ”no action” is selected by the action head. For the placement head, however, since $R_{x,y}$ depends on the selected card h, the placement head outputs a unique heatmap for all cards in the game. Then, we consider only the relevant output once we have selected a card to play.

3.1. Data Collection

Training an imitation learning model for *Clash Royale*™ requires a large and diverse set of high quality replays from human games. Collecting data like this from YouTube would require too much manual work, and *Clash Royale*™ does not have any publicly available APIs for this, so we needed another solution for collecting data (Github). The in-game TV Royale feature continuously showcases games between two random players, with a new replay added to each of the 31 arenas every hour, allowing the theoretical collection of up to 744 replays every day.

To automate the collection of these replays, we created a framework around MuMuPlayer, an Android emulator capable of running *Clash Royale*™. Our framework takes advantage of the Android Debug Bridge (ADB) to provide fast raw framebuffer streaming, multi-instance support, and per-instance code execution.

We then utilized this framework in our replay collection pipeline, where emulators are assigned to record different ranges of arenas. Using simple template matching to identify arenas, we detect the current TV Royale page we are on, navigate through them, check if replays are already watched, and record replays. We discard the first and last 40 frames to remove unwanted artifacts from our data, and

store the frames in compressed Parquet files. An uploader thread handles uploading completed replays to HuggingFace.

Using this pipeline, we collected 2500 replays over 4 days, comprising over 4 million unique frames (Huggingface).

3.2. Data Labeling

After collecting many replays, any actions taken by human players must be extracted to create a labeled data set. This is done in two stages: Marking which card is played (C_h), and where on the screen it is played ($R_{x,y}$). Once the time t of each action is recorded, it is bundled together with the screen capture (G) and saved to a data set.

3.2.1. Identifying C_h

For a given replay screenshot, each available card in hand is shown at a fixed location. In order to detect what cards are in hand, we use a template matching algorithm to compare the input image against known cards. After some preprocessing to denoise and apply effects to the input image, we find the best candidate.

Once each screenshot has each card identified, the next step is to find out when a card is played. This is quite simple given a known hand by the previous method, as the only way a card can leave a hand is by being played.

3.2.2. Identifying $R_{x,y}$

To find out where a card is played, we perform multi-template matching using arena specific clock templates. Given the timestamp at which the card C_h is played, we scan the subsequent frames and apply template matching to the arena to pinpoint the exact arena pixel where the card is played. This gives us the initial position of the deployed unit.

We then convert that screen position into the arena's discrete tile coordinates, which we record as R(x,y). If multiple possible matches show up, we use the frame immediately after the card is played, since that's when the arena clock overlay appears and alignment is most likely reliable. We remove invalid detections and anything outside legal tiles to ensure accurate placement labeling. This yields the final deployment coordinates R(x,y)

3.2.3. Identifying A

In addition to noting C_h and $R_{x,y}$, we must also note if an action is taken A. While by definition,

$$A = \begin{cases} 1 & \text{if } C_h \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

storing each no action taken inflates the dataset far too much (a given replay has roughly 40 actions $A = 1$ and 1800 inactions $A = 0$). In order to save space and training time, we compute the timing of inactions as the midpoint between two actions.

$$t_n^{A=0} = \frac{t_n^{A=1} + t_{n+1}^{A=1}}{2}$$

For example, if one action occurred at $t = 6$ and the next at $t = 10$, an inaction is saved to the dataset at $t = 8$.

3.3. Model Architecture

We tested two different model architectures, one which uses a CNN, and one which uses a ConvLSTM. The first model uses a MobileNetV2 backbone to represent the current frame. The second model uses a convolutional LSTM (still using MobileNetV2) with temporal attention to accommodate input sequences of multiple frames. Both architectures use the same three identical output heads (action, card, and placement) mentioned earlier.

Due to time limitations and very slow upload times, we did not have the chance to test the ConvLSTM model using multiple frames, so in our results you will see a very similar performance between the two models.

3.4. Training and Hyperparameter Tuning

Unfortunately, we did not have the opportunity to spend time tuning the hyperparameters of the models. Most of the time was spent fixing issues with the data processing steps and converting labeled data to high quality model input.

3.5. Inference

At runtime, we use a tick-based decision system which queries the model multiple times per second to get its analysis of the current game state. Then we use a temperature value T to control the eagerness of the model in terms of how often it takes actions. Tuning this value is critical to achieve a good result.

4. Experiment

Describe the computational experiments you conducted to validate the approach you took.

4.1. Evaluating the Model

To evaluate the models, we used a number of metrics. Each head produces its own loss value, which are summed up to calculate the total loss. But loss alone doesn't tell a clear enough story about how strong a model is. So, in addition to loss, we also calculated precision, recall, and F1 scores for both the action and card heads of the model.

For the placement head, we calculated the Expected Euclidean Distance (EED) and Argmax Euclidean Distance (AED). EED is computed by looking at the entire heatmap and calculating the expected distance from the ground truth using the softmax probabilities. AED is computed by just directly taking the distance between the highest probability output and the ground truth. EED provides useful information about the tightness of the clustering, while AED provides a more simplistic measurement of "how far off is the model", without punishing spread-out distributions.

Using these metrics, we evaluated both models on a held out test set, shown in Section 5.

4.2. Live Testing

After evaluating the models against a held out test set, we further tested against heuristic models and live players. Testing against live players was performed on Trophy Road, where the model can gain and lose "trophies" depending on if it wins or loses individual games. Since the number of trophies can vary, this metric can be best expressed by reaching certain milestones called "arenas".

To compare against other models, we used a tutorial mode called "Training Camp". These games are played against Clash Royale's very simple bot which is designed to lose against novice players. We compare our model with a simple heuristic model that plays random cards in random locations on screen every few seconds. We collect statistics such as the health of each tower at the end of the game and compare the models based on these statistics.

5. Results

Table 1 shows the evaluation metrics (described above) that were achieved on a held out test set by the CNN and ConvLSTM models. Interestingly, despite only being given a single frame of gameplay, the ConvLSTM model performed slightly better on average than the CNN model.

Table 1. Evaluation Metrics on Test Set

Metric	CNN	ConvLSTM
Loss	5.0875	**4.8664**
Action Loss	0.4581	**0.4397**
Card Loss	1.1116	**0.9282**
Placement Loss	3.5178	**3.4984**
Action Precision	**0.7639**	0.7412
Action Recall	0.8067	**0.8271**
Action F1	**0.7847**	0.7818
Card Precision (weighted)	0.5079	**0.5184**
Card Recall (weighted)	0.4832	**0.5105**
Card F1 (weighted)	0.4737	**0.4837**
Expected Euclidean Distance	5.3693	**5.2140**
Argmax Euclidean Distance	4.8990	**4.4371**

Table 2 shows the results of the live testing in the "Training Camp" after the initial evaluation. We played five games with each bot and collected the number of wins, crowns, and health of the towers after the game ended. Interestingly, our model showed a significant increase in the health of the friendly towers compared to the heuristic model. This can be attributed to our model effectively defending its towers.

Table 2. Live testing results after 5 games

Metric	DL Clash Bot	Random Bot
Wins	5	4
Crowns Won	10	11
Crowns Lost	2	7
Model Health Left	5894	4632
Enemy Health Left	17597	8486

When testing the model on Trophy Road, we noticed that it only reached Arena 3. The trophy count fluctuated between 600 and 700, but never reached any higher. This shows that our model was able to beat most new players in the game, but struggled against players with only a few days of experience.

6. Discussion and Limitations

6.1. Dataset size and processing time

The main bottleneck for creating a dataset is not processing the dataset, but moving the physical information comprising the dataset. Every replay is already quite large, approaching 2TB combined over 2500 replays. Saving and reading this dataset from the cloud and to client machines can take hours, if not days. Each part of the processing pipeline takes on the scale of minutes, but with the size of the dataset necessitating downloading chunks, processing a subset of 300 replays can take up to a day. This could be massively sped up by recording and processing gameplay footage on the same device, although this was not enabled by our workflow.

For these reasons, our dataset was relatively limited in size, which becomes a big issue when there are more than 150 card classes that have to be learned.

Also, as mentioned earlier, we were only able to feed single-frame inputs to the ConvLSTM model, even though it is built to handle multi-frame input.

Finally, we expect that there was a side effect of the skewed distribution of the replays selected for training. Higher arena games were chosen because of increased human player skill, but using a model trained with high level gameplay may struggle with low level cards and different arena backgrounds.

6.2. Inaction Distribution

While each inaction was taken as the midpoint between two actions, this approach is not representative of real gameplay and was used in order to save space. Distributions for sampling inactions such as uniform, random, normal, or some other may be better for teaching the model when not to take actions in game.

6.3. Spells

Spells are a particularly difficult game element to capture for two reasons: Their effect appears after an unknown number of frames after their card is used, and each spell has a unique area of effect that is difficult to discern where a player has clicked. While the location and timing of a spell being played are deterministic from the game's standpoint, these are not trivial to determine from a replay.

Figure 3. Example placement of the *Arrows* spell card

7. Conclusion and Future Work

In this paper, we demonstrate an end-to-end pipeline for deep learning in *Clash Royale*™, including automated data collection, data labeling, model training, and deployment against human players in real-time.

Additionally, by leveraging TV Royale replays and automated data extraction, we created a powerful dataset for training future imitation models on supervised data, containing over 2,500 replays and nearly 2 TB of data.

This paper shows that imitation learning from human replays is a potential viable alternative to reinforcement learning for games without high-speed APIs like *Clash Royale*™. However, the model's performance against real players also highlights the challenges of scaling imitation learning in a game as complex and dynamic as *Clash Royale*™.

7.1. Memory and Temporal Features

The model's input can be expanded by adding multiple frames for each prediction to identify timing-related features, including additional information at the start of the game such as the cards in the deck that the model controls, including information in short term memory such as the cards that the opponent has in their hand and the elixir count of the opponent, and including the health of the towers as input using OCR to read the on screen text.

7.2. Game State Representation

A helpful pre-training step may be to train an autoencoder to compress the game image. This can save time by allowing replays to be compressed into a vector before being stored in the cloud, in addition to having part of the model understand how the game works before training the decision heads with additional context of player input.

7.3. Gameplay Masking

Another potentially useful method to apply is gameplay masking. Much of the game is spent waiting, and most pixels on the screen are not specifically needed for humans to make a decision. Giving higher weight to pixels important for decision making could improve the model's application with noisy backgrounds, and would likely improve performance. The figure shows example gameplay with a mask to show only pixels different than their base value. This could be used in conjunction with an autoencoder to help the latent representation to only capture on relevant parts of the game image.

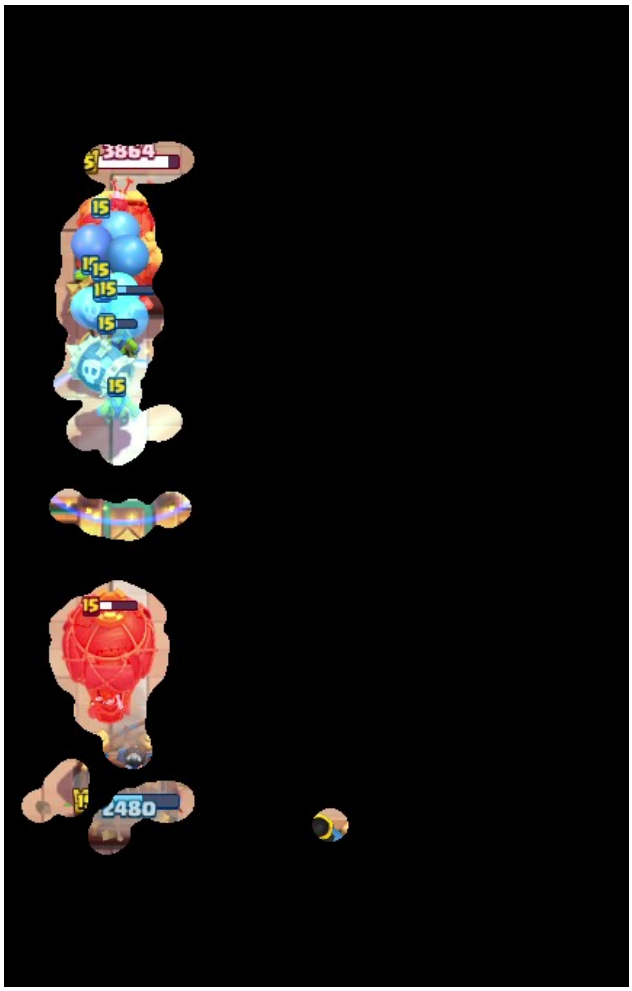

Figure 4. Gameplay footage masked to show only pixels and neighbors not equal to their initial value.

References

Brody, Dai. I made a clash royale bot with machine learning. https://www.youtube.com/watch?v=6Gm-pnNieMU.

Github. Code repository. https://github.com/chrisrca/CS541-Deep-Learning-Clash-Royale-Project.

Huggingface. Dataset. https://huggingface.co/datasets/chrisrca/clash-royale-tv-replays.

RoyaleAPI. Popular decks. https://royaleapi.com/decks/popular?lang=en.