Celeste--AI

Mark Ponomarenko[1], Timothy Chang, Ricardo Parada, Kelly Chang.

Part 1: Abstract

From *Super Mario Bros* [4] and *Atari* [7] to *Go* [10] and even *Starcraft* [9], various forms of machine learning have been used to create game-playing algorithms. A common technique used for this task is reinforcement learning, especially deep Q-Learning. In this paper, we present a novel attempt to use these reinforcement-learning techniques to solve the first stage of *Celeste Classic* [3].

Part 2: Background

Our work is heavily based off the research done by Minh et. al in *Human-Level Control through Deep Reinforcement Learning* [6]. The algorithm we developed to solve *Celeste Classic* uses a deep Q-learning algorithm supported by replay memory, with a modified reward system and explore-exploit probability. This is very similar to the architecture presented by Minh et al.

The greatest difference between our approach and the approach of *Human-Level Control* is the input space and neural network type. Minh et. al use a convolutional neural network, which takes the game screen as input. This requires a significant amount of training epochs and computation time, and was thus an unreasonable approach for us. We instead used a plain linear neural network with two inputs: player x and player y.

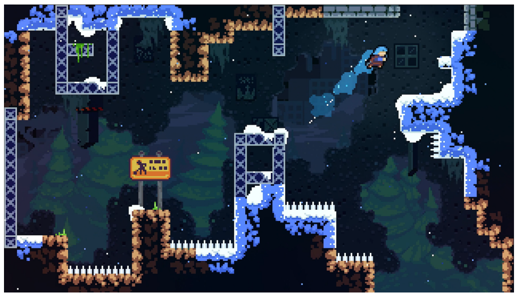
Another project similar to ours is AiSpawn's *AI Learns to Speedrun Celeste* [1]. Here, AiSpawn completes the same task we do---solving *Celeste Classic*---but he uses a completely different, evolution-based approach.

[1]Wrote this paper.

Part 3: Introduction

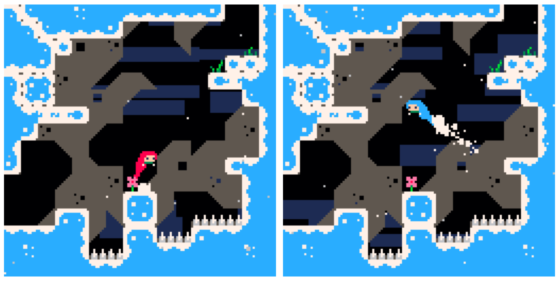
Celeste [2] is a fairly successful 2018 platformer, known for high-quality level design, a vibrant speedrunning[2] community, and brutally difficult progression. It is based on *Celeste Classic*, a 4-day game jam project by the same authors. There are a few reasons we chose to create an agent for *Celeste Classic*:

1: *Celeste Classic* is designed for humans, unlike the environments from, for example, the `gymnasium` [5] library.
2: It runs on the PICO-8 [8], which allows us to modify its code. This grants us a reliable way to interface with the game interface. This is not true of *Celeste* (2018) --- writing a wrapper for *Celeste* would take a significant amout of time.
3: The action space of *Celeste Classic* is small, especially when ineffective actions are pruned.


The first stage of *Celeste* (2018), showing the player dashing to the upper-right.

When we started this project, our goal was to develop an agent that would learn to finish the first stage of this game. It starts in the bottom-left corner of the stage, and needs to reach the top right. If the agent touches the spikes at the bottom of the stage, the game is reset and the agent must try again.
To achieve this end, our agent selects one of nine actions (listed below) at every time step. It does this using a Q-learning algorithm, which is described in detail later in this paper.

Possible actions:
- `left`: move left
- `right`: move right
- `jump-l`: jump left
- `jump-r`: jump right
- `dash-l`: dash left
- `dash-r`: dash right
- `dash-u`: dash up
- `dash-ru`: dash right-up
- `dash-lu`: dash left-up


The first stage of *Celeste Classic*. Two possible actions our agent can take are shown: `jump-r` followed by `dash-lu`.

This task has no direct practical applications. However, by developing an agent that completes this task, we will explore possible techniques and modifications to the traditional DQN algorithm, and we will learn how a simple machine learning model can be adjusted for a rather complicated task.

[2]*speedrunning:* a competition where participants try to complete a game as quickly as possible, often abusing bugs and design mistakes.

Part 4: Methods

Our solution to *Celeste Classic* consists of two major parts: the *interface* and the *agent*. The first provides a high-level interface for the game, and the second uses deep Q-learning techniques to control the player.

4.1 Interface

The interface component does not have any machine-learning logic. Its primary job is to send input and receive game state from *Celeste Classic*. We send input by emulating keypresses with the standard X11 utility `xtodool`. A minor consequence of this is the fact that our agent may only be run in a linux environment, but this can be remedied with a bit of extra code.

We receive game state by abusing the PICO-8's debugging features. Since PICO-8 games are plain text files, we were able to modify the code of *Celeste Classic* with a few well-placed debug-print statements. The interface captures this text, parses it, and feeds it to our model.

The final component of the interface is timing. First, we modified *Celeste Classic* to only run frames when a key is pressed. This allows the agent to run in in-game time, which wouldn't be possible otherwise: *Celeste* usually runs at 30 fps, and the hardware we used to train our model cannot compute gradients that quickly.
Second, we implemented a ``frame skip'' mechanism to the interface, which tells the game to run a certain number of frames---many more than one---after the agent selects an action. The benefit of this is twofold: first, it prevents our model from training on redundant information. The game's state does not see significant change over consecutive frames. Second, frame skipping allows transitions to more directly reflect the consequences of an action.
For example, say the agent chooses to dash upwards. Due to the way *Celeste* is designed, the player cannot take any other action until that dash is complete. Our frame-skip mechanism will run the game until the dash is complete, returning a new state only when a new action can be taken.

4.2 Agent

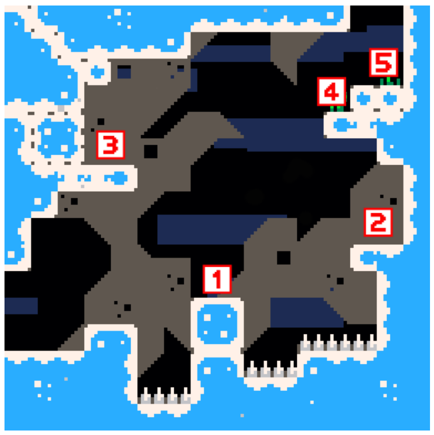
The agent we trained to solve *Celeste Classic* is a plain deep Q-learning agent. A neural network estimates the reward of taking each possible action at a given state, and the agent selects the action with the highest predicted reward. This network is a four-layer fully-connected linear net with 128 nodes in each layer and a ReLU activation function on each hidden node. It has two input nodes that track the player's X and Y-position, and nine output nodes which each correspond to an action the agent can take.

4.2.1 Reward

During training, the agent receives 10 reward whenever it reaches a checkpoint (at right) or completes the stage. If the agent skips a checkpoint, it gets extra reward for each checkpoint it skipped. For example, jumping from point 1 to point 3 would give the agent 20 reward.

These checkpoints are distributed close enough to keep the agent progressing, but far enough away to give it a challenge. Points 4 and 5 are particularly interesting in this respect. When training an agent without point 4, it would often reach the ledge and fall off, getting no reward.

Despite many thousand epochs, this training process was unable to finish the stage. Though the ledge under point 4 is fairly easy to reach from either point 2 or 3, it is highly unlikely that an untrained agent would make it from point 2 to point 5 without the extra reward at point 4.



Locations of non-final checkpoints

4.2.2 Exploration Probability

At every step, we use the Q network to predict the expected reward for taking each of the nine actions. Naturally, the best action to take is the one with the highest predicted reward. In order to encourage exploration, we also take a random action with a probability given by

$$P(c) = \epsilon_1 + (\epsilon_0 - \epsilon_1)e^{-c/d}$$

Where ϵ_0 is the initial random probability, ϵ_1 is the end random probability, and d is the rate at which $P(c)$ decays to ϵ_1. c is a rather unusual ``time" parameter: it counts the number of times the agent has reached the next point.

Usually, such ϵ policies depend on the number of training steps competed. For many applications, this makes sense: if a model is trained on many iterations, it begins to perform better, and thus has less of a need to explore. In our case, that doesn't work: we need to explore until we find a way to reach a checkpoint, and rely on the model's preditions once we've found one. Therefore, instead of computing P with respect to a simple iteration counter, we instead compute it with respect to c.

4.2.3 Target Network, Replay Memory

To prevent an unstable training process, we use a *target network* as described in *Human-Level Control* [6]. However, instead of periodically hard-resetting the target network to the Q network, we use a soft update defined by the following equation, where W_Q and W_T are weights of the Q network and target network, respectively.

$$W_T = 0.05W_Q + 0.95W_T$$

We also use *replay memory* from the same paper, with a batch size of 100 and a total size of 50,000. Our model is optimized using Adam with a learning rate of 0.001.

4.2.4 Bellman Equation

Our goal is to train our model to approximate the value function $Q(s, a)$, which tells us the value of taking action a at state s. This approximation can then be used to choose the best action at each state. We define Q using the Bellman equation:

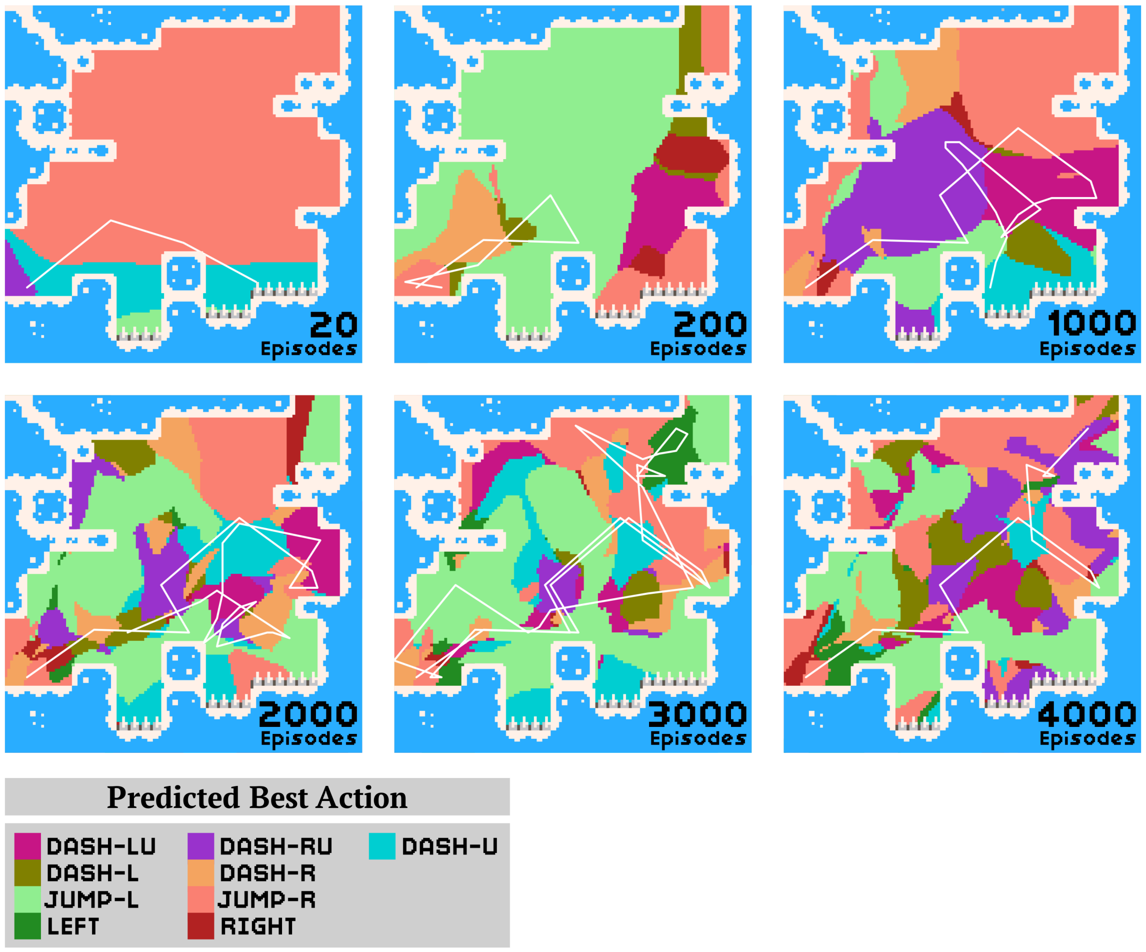
$$Q(s, a) = r(s) + \gamma Q(s_a)$$

Where $r(s)$ is the reward at state s, $Q(s_a)$ is the value of the state we get to when we perform action a at state s, and γ is a discount factor that makes present reward more valuable than future reward. In our model, we set γ to 0.9.

Part 5: Results

After sufficient training, our model consistently completed the first stage of *Celeste*. 4000 training episodes were required to achieve this result.

The figure below summarizes our model's performance during training. The color of each pixel in the plot is determined by the action with the highest predicted value, and the path the agent takes through the stage is shown in white. The agent completes the stage in the ``4000 Episodes'' plot, and fails to complete it within the allocated time limit in all the rest. Training the model on more than 4000 episodes did not have a significant effect on the agent's behavior.



A few things are interesting about these results. First, we see that the best-action patterns in the above graphs to not resemble the shape of the stage. At every point that the agent doesn't visit, the predicted best action does not resemble the action an intelligent human player would take. This is because the model is not trained on these points. The predictions there are a side-effect of the training steps applied to the points in the agent's path.

Second, the plots above clearly depict the effect of our modified explore/exploit policy. We can see the first few segments of the agent's path are the same in each graph. In addition, the more the agent trains, the longer this repeated path is. This is a direct result of our explore/exploit policy: our agent stops exploring sections of the stage it can reliably complete, and therefore repeats paths that work.

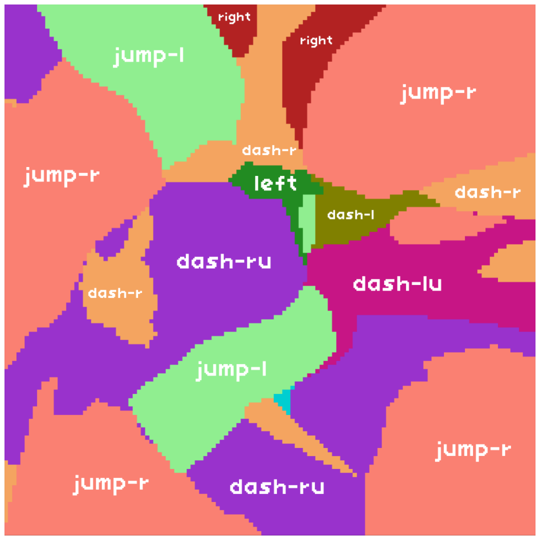
Part 6: Conclusion

Using the methods described above, we were able to successfully train a Q-learning agent to play *Celeste Classic*.

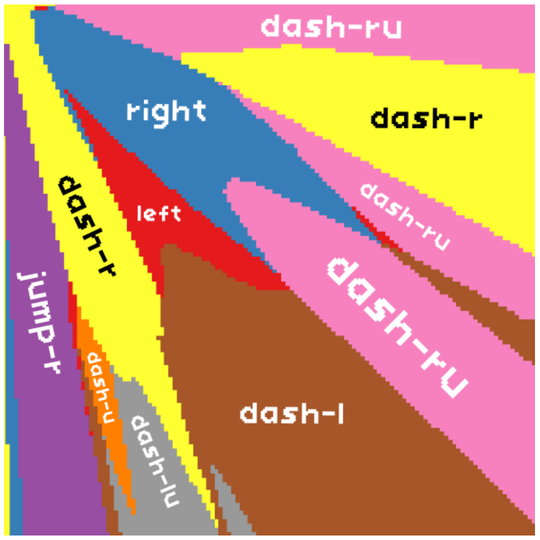
The greatest limitation of our model is its slow training speed. It took the model 4000 episodes to complete the first stage, which translates to about 8 hours of training time. A simple evolutionary algorithm, such as the one presented in *AI Learns to Speedrun Celeste* [1] would likely have better performance than our Q-learning agent. Such an algorithm is much better for incremental tasks (such as this one) than a Q-learning algorithm.

We could further develop this model by making it more autonomous---specifically, by training it on raw pixel data rather than curated `(player_x, player_y)` tuples. This modification would *significantly* slow down training, and is therefore best left out of a project with a ten-week time limit.

While developing our model, we encountered a few questions that we could not resolve. The first of these is the effect of position scaling, which is visible in the graphs below. Note that colors are inconsistent between the graphs---since we refactored our graphing tools after the right graph was generated.



``Best-action'' plot after 500 training episodes with position rescaled to the range $[0, 1]$.



``Best-action'' plot after 500 training episodes with position in the original range $[0, 128]$.

In these graphs, we see that, without changing the model, the scaling of input values has a *significant* effect on the model's performance. Large inputs cause a ``zoomed-out linear fanning'' effect in the rightmost graph, while the left graph, with rescaled values, has a much more reasonable ``blob'' pattern.

In addition to this, we found that re-centering the game's coordinate system so that `(0, 0)` is in the center rather than the top-left also has a significant effect on the model's performance. Without centering, the model performs perfectly well. With centering, our loss grows uncontrollably and the model fails to converge.

In both of these cases, the results are surprising. In theory, re-scaled or re-centered data should not affect the performance of the model. This should be accounted for while training, with the weights of the neural network being adjusted to account for different input ranges. We do not have an explanation for this behavior, and would be glad to find one.

Part 7: Contribution Statement

Ricardo:

code | hypothesis | model design | literature review | research | report

Mark:

code | model design | report | literature review | plots

Timothy:

code | hypothesis | model design | research | code debugging | report

Kelly:

code | hypothesis | model design | organization | report | presentation

References: Sites

[1] AiSpawn. *AI Learns to Speedrun Celeste*. Youtube. url: https://www.youtube.com/watch?v=y8g1AcTYovg. Accessed 2023-02-22.

[2] *Celeste*. 2018. url: https://www.celestegame.com. Accessed 2023-02-25.

[3] *Celeste Classic*. 2015. url: https://www.lexaloffle.com/bbs/?pid=11722. Accessed 2023-02-25.

[4] Yuansong Feng et al. *Train a Mario-playing RL Agent*. url: https://pytorch.org/tutorials/intermediate/mario_rl_tutorial.html. Accessed 2023-02-25.

[5] *Gymnasium*. url: https://github.com/Farama-Foundation/Gymnasium. Accessed 2023-02-25.

[8] *PICO-8*. url: https://www.lexaloffle.com/pico-8.php. Accessed 2023-02-25.

[9] *SC2 AI Arena*. url: https://sc2ai.net. Accessed 2023-02-25.

References: Articles

[6] Volodymyr Mnih et al. ``Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. issn: 00280836. url: http://dx.doi.org/10.1038/nature14236.

[7] Volodymyr Mnih et al. ``Playing Atari with Deep Reinforcement Learning". In: (2013). url: http://arxiv.org/abs/1312.5602.

[10] David Silver et al. ``Mastering the game of Go without human knowledge". In: *Nature* 550 (2017), pp. 354–. url: http://dx.doi.org/10.1038/nature24270.

Part 8: Appendix

Our code is available at `https://git.betalupi.com/Mark/celeste-ai`