# Monster Carlo 2: Integrating Learning and Tree Search for Machine Playtesting

Oleksandra Keehl
*University of California, Santa Cruz*
Santa Cruz, CA, USA
okeehl@ucsc.edu

Adam M. Smith
*University of California, Santa Cruz*
Santa Cruz, CA, USA
amsmith@ucsc.edu

***Abstract*—We describe a machine playtesting system that combines two paradigms of artificial intelligence—learning and tree search—and intends to place them in the hands of independent game developers. This integration approach has shown great success in Go-playing systems like AlphaGo and AlphaZero, but until now has not been available to those outside of artificial intelligence labs. Our system expands the Monster Carlo machine playtesting framework for Unity games by integrating its tree search capabilities with the behavior cloning features of Unity's Machine Learning Agents Toolkit. Because experience gained in one playthrough may now usefully transfer to other playthroughs via imitation learning, the new system overcomes a serious limitation of the older one with respect to stochastic games (when memorizing a single optimal solution is ineffective). Additionally, learning allows search-based automated play to be bootstrapped from examples of human play styles or even from the best of its own past experiences. In this paper we demonstrate that our framework discovers higher-scoring and more-representative play with minimal need for machine learning or search expertise.**

***Index Terms*—behavior cloning, machine playtesting, Monte Carlo Tree Search, machine learning**

## I. Introduction

Playtesting is a necessary but costly part of game design process. While questions like "Is this game fun?" or judgments on aesthetic properties of a game are usually reserved for human playtesters, many other functions of playtesting can be performed in an automated fashion. Benefits of machine playtesting can include more thorough coverage of game-space [1], shorter turn-around times (on account of automated agents being able to play games magnitudes of times faster than humans), and the ability to collect more data faster. The drawback is that developing an agent which can competently play any given game is far from a trivial task.

Artificial intelligence (AI) for playing games has been making significant advances (such as to play Go or StarCraft at master levels). At the same time, the videogame industry has begun to move towards the use of automated playtesting [2]. Unfortunately, the large scale computational resources and AI teams which allow these companies to test their games with a high degree of precision are usually not available to independent developers or those in hobby or educational contexts. Now that free game development software such as Unity[1] and Unreal[2] has lowered the barrier of entry for aspiring game makers, we conjecture quality assurance test automation is a key point of imbalance between small and big studios. We contribute the Monster Carlo 2 (MC2) framework in a step towards bridging this gap.

DeepMind's successes with AlphaGo [3] are the result of combining two branches of AI: search (or planning) and learning. By integrating Monster Carlo (MC1), which is a search-based machine playtesting framework for Unity games; with Unity's open-source Machine Learning Agents Toolkit, which provides the capability to learn from examples; it is now possible to assemble a system with an architecture that mimics DeepMind's AlphaGo while directly integrating with in-development Unity games. Our research makes this cutting-edge AI approach available to game developers in a form that does not require advanced machine learning or search knowledge to apply.

In this paper, we describe the components of MC2 and the results of integrating them with the *Tetris*-like game *It's Alive!*.[3] The stochastic nature of this game (the player does not know which randomly selected piece will drop next) makes it challenging for test automation without applying a strategy of determinization (fixing the game's random seed), which can heavily bias results. In this example integration, we demonstrate the system's ability to bootstrap from its own experience or from sample human gameplay. MC2 can learn a state-dependent action policy. With that, the quality of play that does not exploit knowledge of the future (an artifact of search-based agents) can be inspected. The MC2 framework is now freely available on GitHub.[4]

This paper makes the following contributions:

- An automated playtesting framework integrating search and learning in the context of Unity games.
- Experimental validation of the benefits of combining learning and search without the need for advanced ML expertise.

[1] https://unity.com/

[2] https://www.unrealengine.com

[3] https://www.kongregate.com/games/saya1984/its-alive

[4] https://github.com/saya1984/MonsterCarlo2

978-1-7281-1884-0/19/$31.00 ©2019 IEEE

## II. Background

### A. Machine Playtesting

Machine playtesting [4] is the practice of using computational tools to simulate gameplay in a way that helps answer questions about a game design or the gameplay impact of design changes. Typically, this has been approached as an AI problem to be addressed with search and optimization or, separately, with machine learning approaches.

Holmgard et al. used Monte Carlo Tree Search (MCTS) and distinct utility functions in order to imitate playthroughs of levels in the *MiniDungeons 2* game by players with different priorities—*procedural personas*. For example, the Runner is a persona who tries to finish the level as fast as possible, while the Monster Killer is a persona who tries to kill as many monsters as possible before completing the level. This approach required hand-crafted utility functions specific to the game. When the design questions to be asked are not specific to the hand-crafted personas, evolved utility functions could be used instead [5]. In MC2, MCTS is also used as the core search algorithm. Developers integrating the framework may expose any notion of score or utility to the framework which they would like it to optimize; the framework is compatible with *personas*-style investigations.

Ludocore [6] is a logical game engine for modeling videogames. In this system, gameplay trace inference (accomplished using constraint-solving techniques which utilize search) can be used to ask for examples of gameplay under constraints that characterize assumptions about player or non-player character behavior. Although Ludocore could offer optimality and non-existence guarantees, it could only do this for games specified in a limited logical modeling language. Because developers integrating MC2 with their game control which actions are made available to the AI system, constraints on player behavior can be easily modeled with our framework. Likewise, constraints in playtraces can be implemented with the use of flags (a playtrace that does not meet a certain condition can assigned be a zero score). The exact optimality and non-existence guarantees are unfortunately not possible for games beyond those with very small state spaces.

Recently, researchers at game development firm King[5] (the creators of *Candy Crush*) worked through several different approaches to playtesting their continuously released new levels.[6] They started with human playtesters, which was too slow; moved on to machine playtesting via hard-coded heuristics (procedural personas without search), which were limited in representing strategies; then tried MCTS, which was slow and ultimately resulted in unrealistically super-human play; and, finally, to non-search player models trained via supervised learning on masses of player data in the wild. King used high-capacity deep neural networks and trained them on the data from 1% of global players selected at random during a 2 week period. This resulted in approximately 12 million samples. Since MC2's target audience is small-scale Unity game development teams (or individuals) who are unlikely to have access to that scale of player data or neural network expertise, we intend to offer a more accessible and generic way for using machine playtesting. Lower-capacity networks will mostly be trained on the results of inexpensive simulations.

Ultimately, MC2 aims to combine the usefulness of search (for discovering new play styles) and learning (for summarizing knowledge into an executable form) in a package that is tied to a popular platform rather than an individual game.

### B. Monster Carlo

The original Monster Carlo framework [7] (denoted as MC1 in this paper) used the game-agnostic MCTS algorithm [8] to realize a machine playtesting tool for games built in the Unity game engine. It was best suited to games where player behavior could be well-modeled as making occasional high-level decisions between a relatively small number of meaningful alternatives. Further, it requires games to be deterministic (or at least determinizable by fixing random seeds). MC1 could be set to answer design questions within Jaffe's *restricted play* paradigm [9] where the value of a design element (such as giving players a new kind of action to consider) is judged by the impact of restricting the use of that element on the score of (approximately) optimal play.

While MC1 succeeded in demonstrating a proof-of-concept for search-based machine playtesting for Unity games, inherent limitations of the MCTS algorithm became limitations on the system's ability to usefully answer design questions. MCTS doesn't learn from experience across games (including different determinizations of the same game) or benefit from the existence of samples of human play which are always available for in-development games. As a result, in many cases MC1 was not able to discover gameplay that achieved scores comparable to those of the game's designer, even after exploring the game tree for tens of thousands of rollouts. Further, because of the cost of searching with many rollouts, it was impractical to average the results of machine playtesting experiments over different determinizations (many different random seeds). When MCTS does discover high-scoring play, it has often unrealistically used knowledge of the future (e.g. knowing when a randomized element of the game will drop a favorable outcome by searching ahead) to do it. Answers to design questions asked via restricted play are only as good as the approximations to optimal play on which they are based.

The second iteration of Monster Carlo, MC2, is intended to use a new AI architecture that is capable of learning from experience (of humans and itself) and transferring knowledge across game states and different determinizations. Supporting a mode where decisions are made without access to an oracle for stochastic outcomes, the system also plays the game in a way that better structurally models the human player's experience.

### C. Unity and Unity Machine Learning Agents Toolkit

Unity is a real-time 3D development platform. While it now offers services to a variety of industries, from architecture to

[5] https://king.com/

[6] https://medium.com/techking/human-like-playtesting-with-deep-learning-92adafffe921

cinematics, for many years it has been primarily known as a game development platform and a popular tool for many independent developers as well as hobbyist and educators.

The Unity Machine Learning Agents Toolkit (UMLAT) [10] is described on their GitHub page:[7] "[UMLAT] is an open-source Unity plugin that enables games and simulations to serve as environments for training intelligent agents. Agents can be trained using reinforcement learning, imitation learning, neuroevolution, or other machine learning methods through a simple-to-use Python API." For our purposes, the imitation learning capability of UMLAT is the most relevant, as well as its built-in support for runtime inference, as it offers a convenient way to both train, and later use a trained model in game. In MC2, imitation learning is used to summarize a collection of gameplay samples into a reactive decision policy. Imitation learning allows us to answer the question: "If I were to play in the general style of the samples given, what would be the most likely move for me to take in the current game state?"

### D. AlphaGo and AlphaZero

AlphaGo [3] introduced a revolutionary approach to computer Go, using two deep neural networks: a value network to evaluate board positions (prediction of the chance of winning) and a policy network for move selection (deciding which move should be selected). These networks were initially trained by supervised learning using human expert playtraces, and then further trained on data extracted from self-play using tree search advised by those same networks.

Later, AlphaZero [11] showed that the expert human data used in AlphaGo could be ignored. Bootstrapping from initially random choices using self-play alone could take the system all the way to and beyond the strength of play seen in AlphaGo. While impressive, it is important to consider the depth of machine learning expertise as well as raw computational power needed to achieve these feats. To offer even a fraction of this level of success for in-development Unity games, it is important that our approach doesn't require custom neural network design, careful adjustment of hyperparameters, or specialized hardware not already available to practicing game developers.

There exist several open-source reimplementations of the AlphaGo system such as OpenGo [12]. OpenGo is built on the ELF [13] platform for game AI research. ELF and other frameworks can integrate with new games, however they are primarily targeted at machine learning researchers and are not designed to ease integration with any platform in particular (e.g. Unity).

## III. System Design

The components of our system come from three sources (diagrammed in Fig. 1): UMLAT, which is responsible for the machine learning and run-time inference; the game developer, who provides their own game, including the required modifications to integrate it with UMLAT and MC2; and MC2, which provides the MCTS implementation, experiment setup framework, playtrace collection, result visualization, and facilitates behavior cloning through UMLAT. In this section we will examine each of these components in detail.

### A. Unity Machine Learning Agents Toolkit

Among many other capabilities, UMLAT allows developers to train a decision making model (also called a policy network) from play examples using an imitation learning technique known as behavior cloning (recognizable as a form of supervised learning). This model can be used to play the game automatically. MC2 uses this capability to learn from human expert playtraces, or those discovered via automated search (with MCTS), to train the decision making model. Where in MC1 decisions during MCTS rollouts were selected at random or guided by a developer-provided heuristic, in MC2 they can now be made using the trained decision making model, which leads to more informative search and thus to better high-scores (as confirmed by experimental results covered in a later section).

Because UMLAT's imitation learning mode is primarily setup to take human player input directly during training, training from pre-recorded playtraces (such as those saved from a run of MCTS) was not supported. We made a minor adjustment to UMLAT's open source code to achieve the required functionality. We made it possible for the MC2's C# plugin to dictate the moves taken by UMLAT's *Teacher* agent, which in turn broadcast them through UMLAT's behavior cloning pipeline.

### B. Game Modifications and Integration

MCTS requires a *forward model* to play games. In MC2 (just as in MC1), the Unity game itself is used to provide this. Rather than requiring that game developers be able to save and restore the precise state of their game (an unreasonable task for systems not originally designed to support this), we only require that the game be determinizable. Game states, as they are manipulated by MCTS, are identified by the sequence of player choices needed to reach them. Because identical game states can be reached via different sequences of choices, it is relevant that MC2 is able to learn from experience to transfer knowledge between perceptually equivalent (or even just similar) game states.

As in MC1, the game needs to be able to identify legal actions at each step, request an index of the action to take, and apply that action. There are two additional requirements needed to enable the machine learning aspects of MC2. The first is collection of gamestate at each decision-making point. This should include the information we imagine human players are using to make their own decisions. UMLAT provides best practices for this.[8] The second requirement is that the game needs to be able to wait for the trained model to make the decision, as it is delivered asynchronously.

[7] https://github.com/Unity-Technologies/ml-agents

[8] https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Design-Agents.md

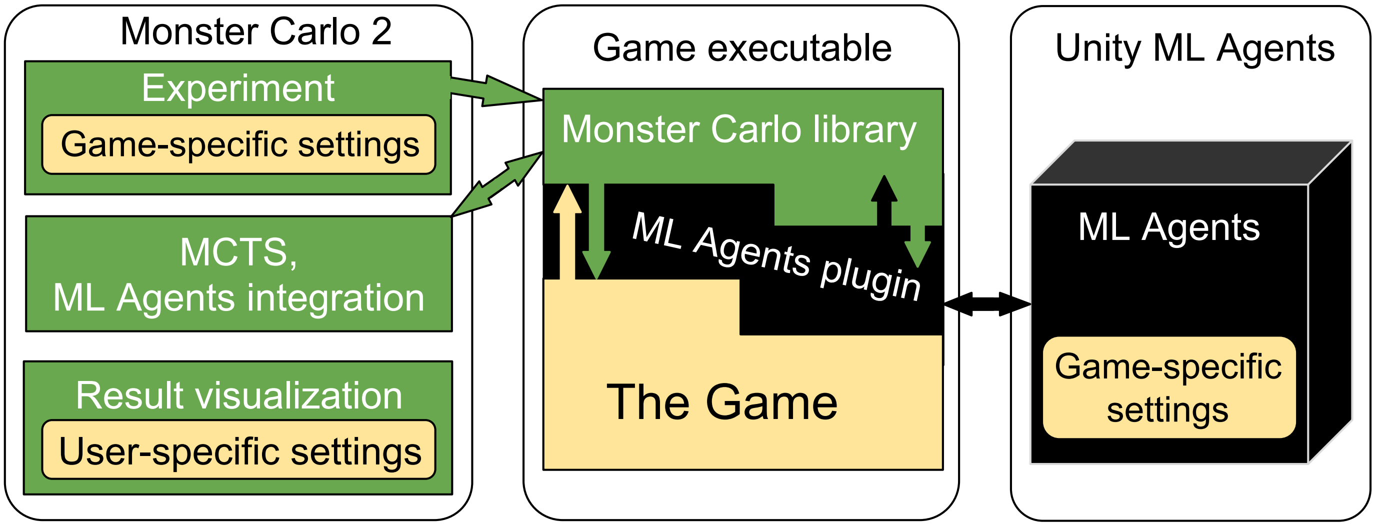


Fig. 1. Software architecture: Green components are provided by the Monster Carlo framework, black components are provided by the Unity Machine Learning Agents Toolkit, and yellow components are project-specific details created by the game developer.

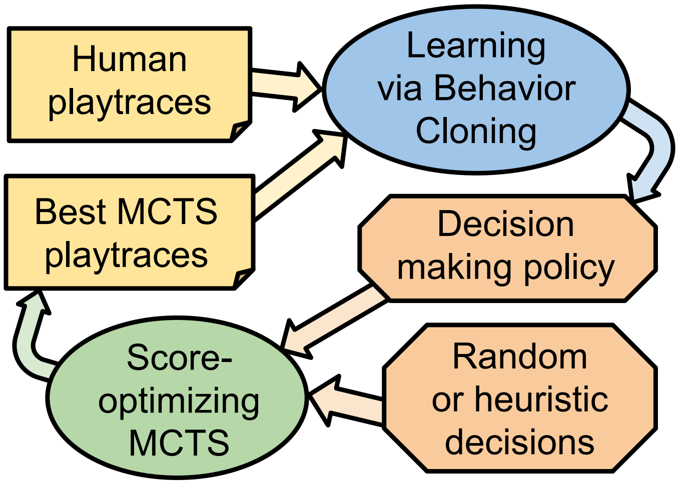


Fig. 2. MC2 Process diagram. The two distinct procedures in MC2 are MCTS rollouts and behavior cloning. Each relies on the output of the other. The behavior cloning needs the behavior to clone (in the form of playtraces). These can either come from human examples or from the games played with MCTS. MCTS needs a way to make decisions during rollouts. This can be done either with a model produced by behavior cloning, by making choices at random or using a user-provided heuristic.

The game developer also needs to specify experiment parameters inside the MC2 experiment setup (a Python program for which we provide a template file), and select features to visualize from the results in the visualization notebook. We also recommend some minor changes to the UMLAT neural network configuration file. Since we do not expect our users to have expert machine learning knowledge, we offer a simple suggestion for setting the network parameters in UMLAT: use a single hidden layer with the number of nodes equal to the average of the number of inputs and outputs [14] (for example if the gamestate can be described with 20 values, and there are four possible actions in the game, the hidden layer could have $(20 + 4)/2 = 12$ nodes. We were able to achieve favorable results using this simple setup.

### C. Monster Carlo 2

MC2 has all the software components present in MC1, albeit modified, as described further: experiment setup, results visualization, MCTS implementation, and the C# library to be included with the game, responsible for decision making and communication with the python modules of MC1. In addition to those, MC2 includes a method to facilitate behavior cloning from playtraces through UMLAT.

Like MC1, MC2 provides the implementation of MCTS, which offers the user several settings, such as the number of rollouts to perform per game, the number of games to play, the UCB value to use and optional terminal branch treatment. The new system has several differences. MC1 would play a limited number of differently-determinized games (in the dozens) exploring each game tree for a relatively long time (10k rollouts) in order to achieve higher scores. This limited number of games made the results vulnerable to the random seeds selected to make the stochastic elements of the game deterministic (a necessity for MCTS). The new system can play thousands of games with significantly fewer rollouts, trying out thousands of random seeds, and thus removing that vulnerability. The user decides how many rollouts each game can perform, and how many rollouts are allowed at each step before the system commits to a move and no longer explores any options that came before it. The obtained scores may be lower than those obtained from prolonged searches of MC1; however, because MC2 is capable of learning from its experiences and iteratively improving its performance, this limitation is ultimately negated and the results are more representative of the game's stochastic nature.

Both MC1 and MC2 allow the option to run the search remotely and in parallel, potentially exploiting many-core and cluster-computing hardware configurations. In the interest of serving small teams and individuals, all experiments in this paper were run on a single laptop computer.

MC2's C# library, which integrates with the game, communicates with the MCTS implementation and (new in MC2) can request or direct decision making of UMLAT-trained agents. It supports four modes of operation: collection of human playtraces; MCTS rollouts with no model; MCTS rollouts utilizing a trained model; and coordinating behavior cloning by using the playtraces provided by a Python training module to feed state-action pairs through the UMLAT imitation learning pipeline.

Finally, like MC1, MC2 includes a Jupyter Notebook with templates for visualizing the results. All the result figures in this paper were obtained this way.

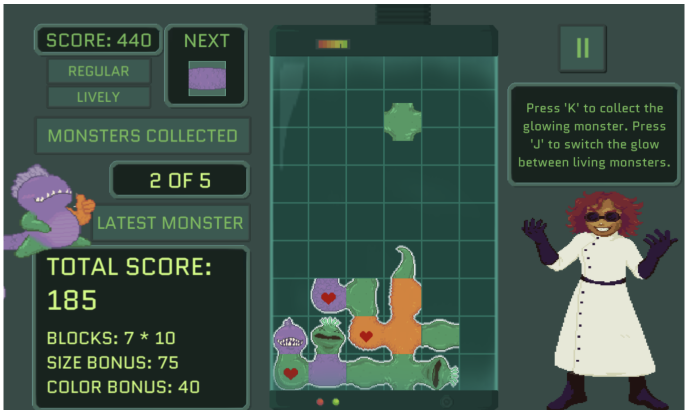
Notably, in both AlphaGo and the experiments by King, researchers avoided training on consecutive sequences of moves from the same game, using random sampling from their considerable available data. The claimed reason for this is that the states of the consecutive moves are usually very close to each other and an over-fitting to those states can become a problem when a general policy is needed. While that may be true, with the limited amount of training data available in MC2's context and the system's inability to jump to an arbitrary game state without running through all the moves leading up to it, we chose to use full consecutive playtraces in our training. We found that even with this simplistic approach we were able to achieve quantitative improvement over MC1's results. There is no doubt that better performance could be obtained with more attention to the machine learning component of MC2. However, it is important for our target audience that this level of expertise is not required.

## IV. Experimental Validation

In this section we describe the game that we used for our experiments; the game-specific neural network architecture and reasoning behind its design; and finally, the goals and results of our experiments.

### A. The game: *It's Alive!*

To compare our results with the previous Monster Carlo paper we decided to use the same game. *It's Alive!* is a *Tetris*-like game, in which monster pieces fall from the top of the screen and the player gets to choose the position and orientation of each piece before it lands (see Fig. 3). The player loses if the pieces pile up to the top of the screen. A monster comes to life if it has at least one head and at least one heart. At this point, the player can collect the monster to free up space, or continue building it up to achieve a higher score. For the purpose of our experiments we set the game field to grid size of four by five and the game ended when two monsters were collected (the limit is normally five). Having a smaller field frequently led to some of the randomly generated sequences resulting in an immediate loss, as there was no heart or no head before the screen filled up. Having a larger field led to longer game rollouts, and therefore lengthier experiments. This may be desirable when testing specific features of the game related to the shape of the field, but in our case, a 4-by-5 field was sufficient. We limited the number of monsters required for completion for the same reason. Developers integrating MC2 are expected to make decisions that balance the complexity and fidelity of playtesting experiments for themselves.



Press 'K' to collect the glowing monster. Press 'J' to switch the glow between living monsters.

Fig. 3. Screenshot of *It's Alive!*, the *Tetris*-like game used in our experiments. During play, the player assembles monsters by rotating falling blocks. A monster comes alive when it has at least one head and one heart. Once alive it can optionally be collected to score points and free-up space. The player may choose to continue building up the monster to earn more points. The goal is to get the highest score after collecting five monsters. The game is lost if the pieces pile up to the top.

### B. Monte Carlo Tree Search

For our experiments we used the following MCTS settings: we limited the search to performing 80 rollouts before committing to a move, which is approximately four times the number of possible moves at each step. The scores achieved by the search increased with the size of the decision limit, but so did the time it took to conduct the experiments. We chose 80 as it provided a reasonable speed–score balance.

The paper about MC1 suggested to use the expected human score as the Upper Confidence Bound (UCB) constant. However, MC1 did not have a notion of a decision limit. In the present experiments, we found that setting the UCB that high diversified the search too much, leading to results not much better than random play. By trial and error, we settled on a value of 200 (which is approximately $\frac{1}{8}$ of the typical human expert score).

### C. The Neural Network Architecture

*1) Input Representation:* Our game state representation (input to a neural network) was inspired by King's experiments used for testing *Candy Crush* [2], namely, having a binary (0/1) channel for each relevant feature of a cell in the game's grid. However, while *Candy Crush* used a binary channel for each candy type, we decided that in our case the more pertinent information was whether any cell on the grid was occupied, and if so, whether things could be connected to it from each of the four directions (up, down, left and right), and whether the monster part occupying the cell or *any*

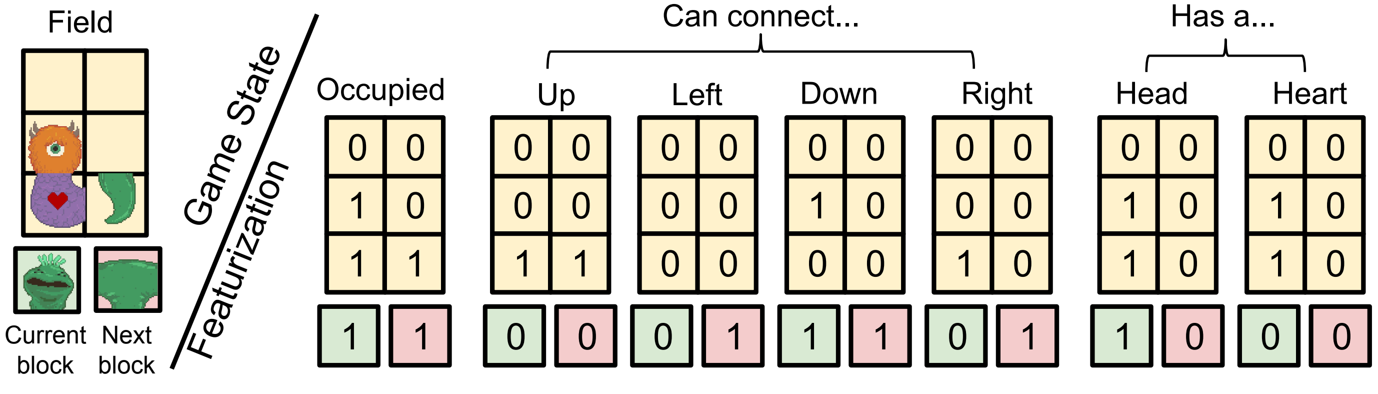


Fig. 4. Game state representation of *It's Alive!*. Here we see what the state representation would look like for the pictured 2x3 field on the left. Seven binary channels with the dimensions matching those of the playfield represent the game state by binary encoding of each of the features we deemed relevant to make the decision on where to place the current block: whether a grid cell is occupied, whether the part in the cell has connections in one of four directions, and whether the part or the monster it belongs to has a head or a heart. The relevant information for the current and upcoming blocks is also captured (the green- and red-tinged numbers respectively in this illustration).

*pieces of the monster it belonged to* had a head or a heart (see Fig. 4). Applicable information for the currently falling block and the upcoming piece was also collected as part of the state. The illustration shows a hypothetical state for a 2x3 game-field, which resulted in 56 values. Since the game used in our experiments had a 4x5 field, our state had 154 observations (4x5=20 of occupied observations, 4x5x4=80 connectivity in four directions observations, 4x5x2=40 head and heart observations, plus 7 observations each for the current and upcoming blocks). The version of UMLAT we used allowed the input of state by successively adding single values via *AddVectorObs(float observation)* method, so while it is conceptually helpful to imagine the game state in terms of tensors, it was actually treated as a long one-dimensional array with all the values. Notably, our input layer had one important flaw: our gamestate didn't capture the information about the color of blocks, which is relevant for decision making, as placing same-colored blocks next to each other lead to color bonus. It is expected that the featurization of the game state doesn't describe every possible detail of the game, just capturing most of the useful details.

*2) Output Representation:* Neural networks have a fixed input and output size. Since our outputs represented the space of possible actions to take, which differed from state to state, we set its size to what we estimated to be the biggest number of possible actions that could be available at any one time. In our case, we had 16 landing actions (width of the field = 4 times possible orientations = 4), plus three monster-collecting actions for at most three living monsters on the field of that size. We used the UMLAT action-masking feature to prevent the model from selecting an illegal action, such as trying to collect a monster when no live monsters were present. In the rare cases when this feature failed (approximately 0.5% of the time), MC2 fell back to making a random choice among legal actions.

*3) Network Architecture:* By default, UMLAT uses a multi-layer neural network architecture with three hidden layers, each with 8 nodes. We found that a network with that structure did not perform very well in our case. One of the rules of thumb for choosing the hidden layer structure is a simple formula [14], claimed to work acceptably for most problems. Since we do not expect our users to have advanced machine learning expertise, and nor are we experts ourselves, we decided to go with this formula, as it produced favorable results (described later). In our case, we had 154 input features and 19 possible outputs, so we decided to use 90 hidden nodes in a single hidden layer.

### *D. Experiments*

Our work was inspired by Google's DeepMind's successes with AlphaGo and AlphaZero. We set out to see if we could recreate some of their impressive results, albeit on a vastly smaller scale and with a more practical application for game developers. With that in mind, we identified several sets of data we wanted to examine and compare. The first set was a collection of expert playtraces (to stand in for grandmaster players). These were recorded from the game developer's own play. We obtained 124 game's worth of state-move pairs (about 3300), each game with a different random seed determining piece drops. We trained a move-prediction model on this data for three epochs (experimentally, we found that adding more epochs led to overfitting). We call this policy Human Model 1 (H1). H1 correctly predicted approximately 50% of expert moves. This is significant considering that, in our experiments there were 19 possible moves at every step, so the accuracy of uniform random guessing would be just over 5%.

We also collected 2000 playtraces from games played by using the MCTS algorithm with uniform-random rollouts. We used 124 (the same number of expert samples) best scoring games from this set to train another policy, which we call Search Model 1 (S1).

We collected 2000 playtraces from MCTS with S1 used to execute rollouts and conducted additional training on S1 using the best 124 traces from the set, thus creating the Search Model 2 (S2)

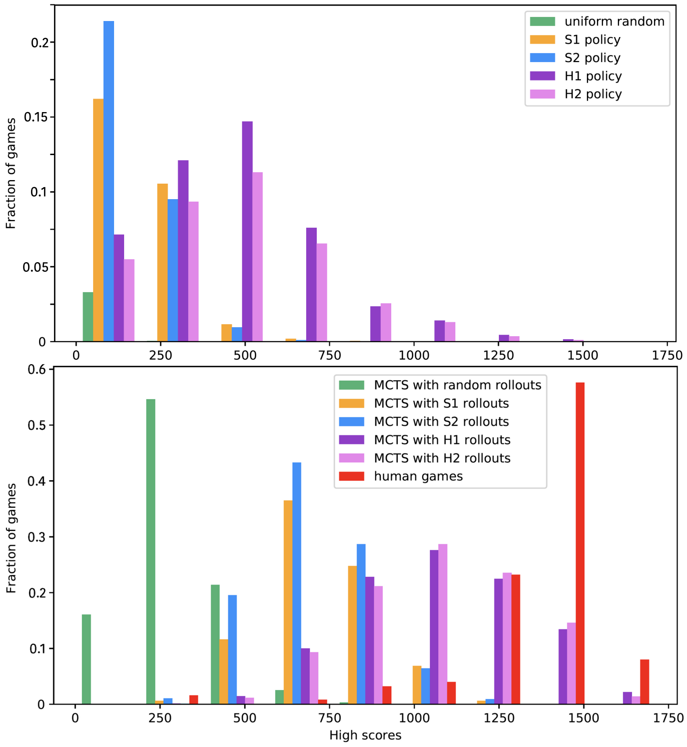


Fig. 5. Top: Distribution of scores for play simulated without using search, just using a decision making policy based on our feature representation of game state. Bottom: Distribution of scores for play using search (making use of different rollout policies). The distribution of human expert scores (from 124 games) are included here. The two charts use the same horizontal axis scaling for game scores, but the vertical axes use different scales.

| Dataset | Mean | SD | Median | Max |
|---|---|---|---|---|
| Uniform Random | 1.08 | 12.03 | 0 | 195 |
| S1 Policy | 61.14 | 109.82 | 0 | 770 |
| S2 Policy | 50.05 | 95.01 | 0 | 680 |
| H1 Policy | 202.31 | 277.90 | 0 | 1430 |
| H2 Policy | 169.72 | 271.12 | 0 | 1430 |
| MCTS with Random | 295.72 | 141.13 | 295 | 685 |
| MCTS with S1 | 734.13 | 159.44 | 725 | 1310 |
| MCTS with S2 | 709.14 | 165.25 | 700 | 1290 |
| MCTS with H1 | 1063.02 | 244.52 | 1055 | 1710 |
| MCTS with H2 | 1070.87 | 237.23 | 1070 | 1680 |
| Human Play | 1326.90 | 245.93 | 1400 | 1600 |

Fig. 6. Summary statistics for each dataset of scores. With exception of MCTS with H1 and MCTS with H2, each pair of the results was statistically distinct with $p < 0.009$ according to a two-sided Mann-Whitney U test.

Finally, we collected 2000 playtraces from games played by using the MCTS algorithm where rollouts were executed using the H1 policy. Again, we used the top 124 scoring games to further train H1 (that is apply additional training to the already trained model), thus creating H2. Whether H2 can outperform H1 depends on the capacity of the neural network used to represent it and the hyperparameters used in its training. In our case, our networks inability to learn about monster colors likely contributed to the model's lack of improvement from H1 to H2. Similarly, the hyperparameters may be to blame for the fact that S2 performed worse than S1.

The results of our machine playtesting experiments are presented in Fig. 5 and Fig. 6. The top chart of Fig. 5 shows results obtained by playing the game with five different ways of making decisions, 2000 games each. The five decision models are: uniform random; S1, S2, H1 and H2 (described earlier). Notably, the games with zero score are not included in the graphic, as the 1924 zeroes resulting from random playthroughs dwarfed the rest of the results. The H1 policy only got 980 zeroes.

The bottom chart of Fig. 5 shows the distribution of scores obtained by playing the game using MCTS with rollout decisions based on the five policies in the figure above. The expert human game score distribution is included for comparison. As expected, scores obtained with search are generally higher than scores obtained without search. Notably, researchers at King decided to abandon search-based playtesting partially because the determinized search's look-ahead quality lead to superhuman performance, as search would select moves based on the sequence of pieces unseen by the player. However, due to computational limitations, the users of MC2 are unlikely to perform the number of MCTS rollouts comparable to King for this to become a problem. Also, not all games' results are as drastically affected by stochastic elements as *Candy Crush*, so this may not always be an issue at all. Finally, because our system allows to train models which can be used to playtest the games by making decisions based on current state without look-ahead to the future (though, admittedly, with much lower scores in our experiments), it is conceivable that the look-ahead problem can be avoided entirely.

Fig. 6 goes over the quantitative measurements of our results. Notably, MC1's experiments failed to reach human play results even with tens of thousands of MCTS rollouts, whereis MC2's combination of MCTS and H1 and H2 models were able to occasionally surpass human performance.

Our results show the utility of combining learning and search to discover high-scoring play. Learning can be applied to human expert play where available and to self-play simulations when not. Although we were hoping to see stronger benefits from iterating the cyclic process illustrated in Fig. 2, we hypothesize that the weak lift seen in our experience mostly reflects the low capacity of our neural networks and lack of fine-tuned hyperparameters. For the individual or small-team game developer, we have demonstrated a clear benefit for combining learning and search. Those willing to learn more about how neural network architectures are specified in UMLAT can turn up the capacity of the model to match the scale of their data as needed.

## V. Future Work

Compared to AlphaGo and AlphaZero, MC2 is notably missing the ability to evaluate a state (whereas the state evaluation network in AlphaZero was trained at the same

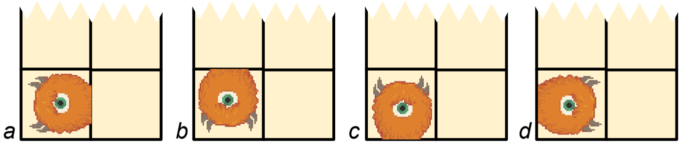


Fig. 7. Game states resulting from four possible piece-landing actions. States *a* and *b* are much more desirable than *c* and *d* because nothing can be attached to the monster head in the latter two cases, making it permanently take up space in the field.

time as the policy network used for rollouts). We believe that the ability to evaluate intermediate states (rather than only learn from long and questionably accurate rollouts) could improve the playing strength of our system without increasing its overall complexity as experienced by its users. Fig. 7 illustrates how a quick judgment of game states might be used to guide the search away from less desirable states; many samples of policy-guided play would be required to discover what an evaluation network could make in one step. Training of an evaluation function and use of it during search would be no more complicated for the game programmer than that of the policy network for the current system. When using the restricted play paradigm to evaluate the importance of individual player choices, we believe a future system's ability to transfer the knowledge in the state evaluation function would dramatically boost the system's effectiveness when the value of most states has not changed by much. Additionally, the output of the evaluation function could be used to color-code samples of play to aid in visual analysis of machine playtesting results.

## VI. Conclusion

In this paper we described Monster Carlo 2, a system for playtesting Unity games which expanded the original Monster Carlo framework by integrating it with Unity's Machine Learning Agents Toolkit. MC2 combined two paradigms of AI—machine learning through neural networks and tree search through MCTS—an approach that showed unprecedented success with AlphaGo. Our system aims to bring a scaled down version of this technology to independent game developers, hobbyists and educators alike. It does this by eliminating the need for neural network expertise and heavy computational resources featured in recent breakthrough gameplay AI advances and by tying it to a popular game development platform, the Unity game engine.

MC2 allows the user to record human playtraces or collect playtraces with MCTS rollouts; train decision making models from playtraces; and use those models during MCTS rollouts.

We described our experience using the system on a published game, including the reasoning behind our network design and MCTS parameters. We demonstrated the benefits of the combination of search with learning over either of the methods used individually through a series of experiments. Notably, this approach was able to reach human player scores, something MC1 was not capable of.

For the games like *Candy Crush*, where the stochastic nature of the game can lead to wildly unpredictable consequences, such as a chain reaction of candy "explosions," which are exploited by MCTS and can lead to unnatural super-human play, MC2 offers reactive playtesting with models trained on human play or from MCTS rollouts. Admittedly, MC2's purposefully simplistic neural net designs don't produce very competent models, so developers of *Candy Crush*-like games may benefit from investing time to develop their network setups a little more thoroughly. This is not a problem for games in which stochastic elements have less drastic consequences, such as *Tetris* and *It's Alive!*

Finally, our framework is freely available for public use.

## References

[1] K. Chang, B. Aytemiz, and A. M. Smith, "Reveal-more: Amplifying human effort in quality assurance testing using automated exploration," in *IEEE Conference on Games (COG)*, 2019.

[2] S. Gudmundsson, P. Eisen, E. Poromaa, A. Nodet, S. Purmonen, B. Kozakowski, R. Meurling, and L. Cao, "Human-like playtesting with deep learning," in *IEEE Conference on Computational Intelligence and Games (CIG)*, 2018.

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, jan 2016.

[4] A. M. Smith, M. J. Nelson, and M. Mateas, "Computational support for play testing game sketches.," in *Proceedings of the 5th Artificial Intelligence for Interactive Digital Entertainment Conference (AIIDE)*, 2009.

[5] C. Holmgård, M. C. Green, A. Liapis, and J. Togelius, "Automated playtesting with procedural personas through MCTS with evolved heuristics," *CoRR*, vol. abs/1802.06881, 2018.

[6] A. M. Smith, M. J. Nelson, and M. Mateas, "Ludocore: A logical game engine for modeling videogames," in *IEEE Conference on Computational Intelligence and Games (CIG)*, 2010.

[7] O. Keehl and A. M. Smith, "Monster carlo: An MCTS-based framework for machine playtesting unity games," in *IEEE Conference on Computational Intelligence and Games (CIG)*, 2018.

[8] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of Monte Carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, pp. 1–43, March 2012.

[9] A. Jaffe, A. Miller, E. Andersen, Y.-E. Liu, A. Karlin, and Z. Popović, "Evaluating competitive game balance with restricted play," in *Proceedings of the Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'12, pp. 26–31, AAAI Press, 2012.

[10] A. Juliani, V. Berges, E. Vckay, Y. Gao, H. Henry, M. Mattar, and D. Lange, "Unity: A general platform for intelligent agents," *CoRR*, vol. abs/1809.02627, 2018.

[11] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[12] Yuandong Tian, Jerry Ma*, Qucheng Gong*, Shubho Sengupta*, Zhuoyuan Chen, James Pinkerton, and C. Lawrence Zitnick, "Elf opengo: An analysis and open reimplementation of alphazero," *CoRR*, vol. abs/1902.04522, 2019.

[13] Y. Tian, Q. Gong, W. Shang, Y. Wu, and C. L. Zitnick, "Elf: An extensive, lightweight and flexible research platform for real-time strategy games," in *Advances in Neural Information Processing Systems*, pp. 2656–2666, 2017.

[14] J. Heaton, *Introduction to Neural Networks with Java*. Heaton Research, 2008.