Winning Isn't Everything: Training Agents to Playtest Modern Games

Igor Borovikov[*†], **Yunqi Zhao**[*†], **Ahmad Beirami**[*†]

Jesse Harder[†], **John Kolen**[†], **James Pestrak**[†], **Jervis Pinto**[†], **Reza Pourabolghasem**[†]

Harold Chaput[†], **Mohsen Sardari**[†], **Long Lin**[†], **Navid Aghdaie**[†], **Kazi Zaman**[†]

Abstract

Recently, there have been several high-profile achievements of agents learning to play games against humans and beat them. We propose an approach that instead addresses how the player experiences the game, which we consider to be a more challenging problem. In this paper, we present an alternative context for developing AI that plays games. Specifically, we study the problem of creating intelligent game agents in service of the development processes of the game developers that design, build, and operate modern games. We highlight some of the ways in which we think intelligent agents can assist game developers to understand their games, and even to build them. Our main contribution is to propose a learning and planning framework that is uniquely tuned to the environment and needs of modern game engines, developers and players. Our game agent framework takes a few steps towards addressing the unique challenges that game developers face. We discuss some early results from an initial implementation of our framework.

Artificial intelligence; artificial game agent; reinforcement learning; imitation learning; deep learning; model-based learning; game design; game playtesting; non-player character (NPC).

Introduction

The history of artificial intelligence (AI) can be mapped by its achievements playing and winning various games. From the early days of Chess-playing machines to the most recent accomplishments of Deep Blue and AlphaGo, AI has advanced from competent, to competitive, to champion in even the most complex games.

Games have been instrumental in advancing AI, and most notably in recent times, reinforcement learning (RL). IBM Deep Blue was the first AI agent who beat the chess world champion, Gary Kasparov (Deep Blue 1997). A decade later, Monte Carlo Tree Search (MCTS) (Coulom 2006; Kocsis and Szepesvári 2006) was a big leap in solving games. MCTS agents for playing Settlers of Catan were reported in (Szita, Chaslot, and Spronck 2009; Chaslot et al. 2008) and shown to beat previous heuristics. Other work compares multiple approaches of agents to one another in the game Carcassonne on the two-player variant of the game and discusses variations of MCTS and Minimax search for playing the game (Heyden 2009). MCTS has also been applied to the game of 7 Wonders (Robilliard, Fonlupt, and Teytaud 2014) and Ticket to Ride (Huchler 2015).

Recently, DeepMind researchers demonstrated that deep neural networks (DNNs) combined with MCTS could lead to AI agents that play Go at a super-human level (Silver et al. 2016), and solely via self-play (Silver et al. 2017; Silver et al. 2018). Subsequently, OpenAI researchers showed that AI agents could learn to cooperate at a human level in Dota 2 (OpenAI Five 2018).

The impressive recent progress on RL to solve games is partly due to the advancements in processing power and AI computing technology.[1] Further, deep Q networks (DQNs) have emerged as a general representation learning framework from the pixels in a frame buffer combined with Q function approximation without need for task-specific feature engineering (Mnih et al. 2015). The design of a DQN and setting the hyperparameters is still a daunting task. In addition, it takes hundreds of thousands of state-action pairs for the agent to reach human-level performance. Applying the same techniques to modern games would require obtaining and processing even more state-action pairs, which is infeasible in most cases because speeding up the game engine may not be possible and the game state may be difficult to infer from the frame buffer. The costs associated with such an approach may be too high for many applications, not justifying the benefits.

On modern strategy games, DeepMind and Blizzard showed that existing techniques fall short even on learning the rules of StarCraft II (Vinyals et al. 2017). While breaking the state space and action space into a hierarchy of simpler learning problems has shown to be

[*]These authors contributed equally to this work. Contact: {iborovikov, yuzhao, abeirami}@ea.com.

[†]EA Digital Platform – Data & AI, Electronic Arts, 209 Redwood Shores Pkwy, Redwood City, CA 94065, USA.

[1]The amount of AI compute has been doubling every 3-4 months in the past few years (AI & Compute 2018).

promising (Vinyals et al. 2017; Pang et al. 2018), additional complications arise when building agents for a modern game that is still under development. Some of these challenges are:

1. The game state space is huge, with continuous attributes, and is only partially observable to the agent.
2. The set of available actions is huge, parametric, partly continuous, and potentially unknown to the agent rendering a direct application of MCTS infeasible.
3. The game itself is dynamic in the design and development stage, and multiple parameters and attributes (particularly related to graphics) may change between different builds.
4. The games are designed to last tens of millions of timesteps[2] leading to potentially long episodes, and the manner the player engages with the game environment impacts the gameplay strategy.[3]
5. Multiple players may interact in conflict or cooperation leading to an exploding state space and non-convergence issues due to invalidation of the Markov assumption in a multi-agent learning environment.
6. Winning isn't everything, and the goal of the agent could be to exhibit human-like behavior/style to better engage human players, making it non-trivial to design a proper rewarding mechanism.

The idea of using AI techniques to augment game development and playtesting is not new. Algorithmic approaches have been proposed to address the issue of game balance, in board games (De Mesentier Silva et al. 2017; Hom and Marks 2007) and card games (Krucher 2015; Mahlmann, Togelius, and Yannakakis 2012). More recently, (Holmgard et al. 2018) builds a variant of MCTS to create a player model for AI Agent based playtesting. These techniques are relevant to creating rewarding mechanisms for mimicking player behavior.

Another line of research focuses on investigating approaches where AI and machine learning can play the role of a co-designer, making suggestions during development (Yannakakis, Liapis, and Alexopoulos 2014). Tools for creating game maps (Liapis, Yannakakis, and Togelius 2013) and level design (Smith, Whitehead, and Mateas 2010; Shaker, Shaker, and Togelius 2013) are also proposed. Other approaches have explored AI for designing new games. (Browne and Maire 2010) generates entirely new abstract games by means of evolutionary algorithms. (Salge and Mahlmann 2010) relates the game design to the concept of relevant information. (Smith, Nelson, and Mateas 2010) addresses the behavior emanating from the design by having an engine capable of recording play traces. (Treanor et al. 2015) proposes an ideation technique to embed design patterns in AI based game design. (Zhu, Wang, and Zyda 2018) uses a measure for similarity between game events to transfer different levels across games. See (Togelius et al. 2011; Summerville et al. 2018) for a survey of these techniques in game design.

In the next section, we make the case for using AI as more of a tool to help designers tune their game, rather than build an agent with super-human performance or in order to create a game (or a part thereof).

Playtesting Game Agents

While achieving optimal super-human gameplay using modern RL techniques is impressive, our goal is to train agents that can help game designers ensure their game provides players with optimal experiences. As it is not obvious to define a reward function that abstracts an optimal experience, this problem does not necessarily lend itself to a traditional RL formulation. For instance, we considered the early development of The Sims Mobile, whose gameplay is about "emulating life": players create avatars, called Sims, and conduct them through a variety of everyday activities. In this game, there is no single predetermined goal to achieve. Instead, players craft their own experiences, and the designer's objective is to evaluate different aspects of that experience.

To validate their design, game designers conduct playtesting sessions. Playtesting consists of having a group of players interact with the game in the development cycle to not only gauge the engagement of players, but also to discover elements and states that result in undesirable outcomes. As a game goes through the various stages of development, it is essential to continuously iterate and improve the relevant aspects of the gameplay and its balance. Relying exclusively on playtesting conducted by humans can be costly and inefficient. Artificial agents could perform much faster play sessions, allowing the exploration of much more of the game space in much shorter time. This becomes even more valuable as game worlds grow large enough to hold tens of thousands of simultaneously interacting players. Games at this scale render traditional human playtesting infeasible.

Recent advances in the field of RL, when applied to playing computer games (e.g., (OpenAI Five 2018; Mnih et al. 2015; Vinyals et al. 2017; Harmer et al. 2018)), assume that the purpose of a trained artificial agent ("agent" for short) is to achieve the best possible performance with respect to clearly defined rewards while the game itself remains fixed for the foreseen future. In contrast, during game development the objectives and the settings are quite different. The agents can play a variety of roles with the rewards that are not obvious to define formally, e.g., an objective of an agent exploring a game level is different from foraging, defeating all adversaries, or solving a puzzle. Also,

[2] A timestep is the unit time where the agent takes an action, its observation is updated, and a reward signal is sensed.

[3] In a game that is designed to last many months or years, the player is not expected to be taking actions at all timesteps. Rather, the player's best strategy depends on the level of their engagement with the game.

the game environment changes frequently between the game builds. In such settings, it is desirable to quickly train agents that help with automated testing, data generation for the game balance evaluation and wider coverage of the gameplay features. It is also desirable that the agent be mostly re-usable as the game build is updated with new appearance and gameplay features. Following the direct path of throwing computational resources combined with substantial engineering efforts at training agents in such conditions is far from practical and calls for a different approach.

In this paper, we propose a framework that supports game designers with automated playtesting. We begin by describing a training pipeline that is needed to be able to universally apply this framework to a variety of games. We then provide some of the solution techniques that we have found useful in solving the problem. Finally, we lay down our future roadmap.

Training Pipeline

The training pipeline consists of two key components:

- *Gameplay environment* refers to the simulated game world that executes the game logic with actions submitted by the agent every timestep and produces the next state.

- *Agent environment* refers to the medium where the agent interacts with the game world. The agent observes the game state and produces an action. This is where training occurs.

In practice, the game architecture can be complex and it might be too costly for the game to directly communicate the complete state space information to the agent at every timestep. To train artificial agents, we create a universal interface between the gameplay environment and the learning environment.[4] The interface extends OpenAI Gym (OpenAI Gym 2016) and supports actions that take arguments, which is necessary to encode action functions and is consistent with PySC2 (Vinyals et al. 2017; PySC2 2017). In addition, our training pipeline enables creating new players on the game server, logging in/out an existing player, and gathering data from expert demonstrations. We also adapt Dopamine (Bellemare et al.) to this pipeline to make DQN (Mnih et al. 2015) and Rainbow (Hessel et al. 2017) agents available for training in the game. Additionally, we add support for more complex preprocessing other than the usual frame buffer stacking, which we explicitly exclude following the motivation presented in the next section.

[4]These environments are usually physically separated, and hence, we prefer a thin (i.e., headless) client that supports fast cloud execution, and is not tied to frame rendering.

Solution Techniques

State Abstraction

The use of frame buffer as an observation of the game state has proved advantageous in eliminating the need for manual feature-engineering in Atari games (Mnih et al. 2015). However, to achieve the objectives of RL in a fast-paced game development process, the drawbacks of using frame buffer outweigh its advantages. The main considerations which we take into account when deciding in favor of a lower-dimensional engineered representation of game state are:

(a) During almost all stages of the game development, the game parameters are evolving on a daily basis. In particular, the art may change at any moment and the look of already learned environments can change overnight. Hence, it is desirable to train agents using features that are more stable to minimize the need for retraining agents.

(b) The gameplay environment and the learning environment usually reside in physically separate nodes. Naturally, closing the RL state-action-reward loop in such environments requires a lot of network communication. Presence of frame buffers as the representative of game state would significantly increase this communication cost whereas derived game state features enable more compact encodings.

(c) Obtaining an artificial agent in a reasonable time (a few hours at most) usually requires that the game be clocked at a rate much higher than the usual gameplay speed. As rendering each frame takes a significant portion of every frame's time, overclocking with rendering enabled is not practical. Additionally, moving large amount of data from GPU to main memory drastically slows down the game execution and can potentially introduce simulation artifacts, by interfering with the target timestep rate.

(d) Last but not least, we can leverage the advantage of having privileged access to the game code to let the game engine distill a compact state representation that could be inferred by a human player from the game and pass it to the agent environment. By doing so we also have a better hope of learning in environments where the pixel frames only contain partial information about the the state space.

The compact state representation could include the inventory, resources, buildings, the state of neighboring players, and the distance to target. In an open-world first-person shooter game the features may include the distance to the adversary, angle at which the agent approaches the adversary, presence of line of sight to the adversary, direction to the nearest waypoint generated by the game navigation system. The feature selection may require some engineering efforts but it is logically straightforward after the initial familiarization with the gameplay mechanics, and often similar to that of CPU AI sensing, which will be informed by the game designer. We remind the reader that our goal is not to

train agents that win but to simulate human-like behavior, so we train on information that would be accessible to a human player.

Open-Loop Control

Before tackling the challenging problem of closed-loop RL, we consider a more simplified case that applies when the game is fully observable and the game dynamics are fully known. This simplified case allows for the extraction of a lightweight model of the game. While this requires some additional development effort, we can achieve a dramatic speedup in training agents by avoiding RL and using open-loop planning techniques instead. We then translate the learning back into the actual game.

As the first case study, we consider a mobile game with known dynamics, where each player can pursue different careers, and as a result will have a different experience. In the use-case that we consider in this section, the designer's goal is to measure the impact of high-level decisions on the progression path of the player. We refer the interested reader to (Silva et al. 2018) for a more complete study of this problem.

The abstract model of the deterministic state transition of the game provides full knowledge of the game state. The lightweight model also allows storage in memory (or on a disk), and loading it back within the same game session, all with negligible overhead. Such simplicity of the model manipulation enables search-based algorithms with the ability to look ahead and operate on game states using graph-based algorithms. In particular, we use the A* algorithm as it enables computation of an optimal strategy by exploring the state transition graph instead of the more expensive iterative processes, such as dynamic programming,[5] and even more expensive Monte Carlo search based algorithms. The customizable heuristics and the target states corresponding to different gameplay objectives, offered by A*, provide sufficient control to conduct various experiments and explore multiple aspects of the game.

We validate our approach against (approximately) solving a full optimization over the entire game strategy space using evolution strategies, where the agent optimizes for a utility function that selects between available actions.[6]

We compare the number of actions that each approach needs to achieve the goal for each career in Figure 1. We emphasize that our goal is to show that a simple planning method, such as A*, can sufficiently satisfy the designer's goal in this case. We can see that

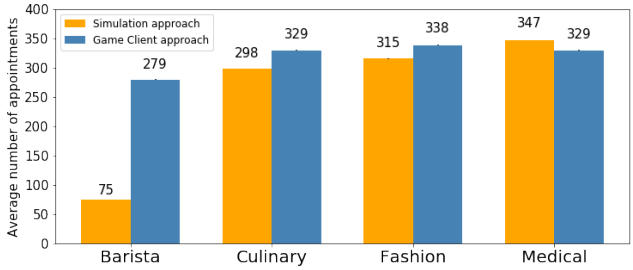

Figure 1: Comparison of the average amount of career actions (appointments) to reach the goal using A* search and evolution strategy adapted from (Silva et al. 2018).

the more expensive optimization based evolution strategy reaches a style of gameplay that is similar to the much simpler A* search.

The largest discrepancy arises for the Barista career, which might be explained by the fact that this career has an action that does not reward experience by itself, but rather enables another action that does it. This action can be repeated often and can explain the high numbers despite having half the number of levels. Also, we observe that in the case of the medical career, the 2,000 node A* cutoff has led to a suboptimal solution.

When running the two approaches, another point of comparison can be made: how many sample runs are required to obtain statistically significant results? We ran 2,000 runs for the evolution strategy while it is notable that the A* agent learns a deterministic playstyle, which has no variance. On the other hand, the agent trained using an evolution strategy has a high variance and requires a sufficiently high number of runs of the simulation to approach a final reasonable strategy (Silva et al. 2018).

In this experiment, we were able to create a simulation model for the game mechanics, and we found that its benefits outbalance the time needed to run the actual simulations to answer different questions raised by the game designer. However, it is worth reiterating that the availability of a game model as a separate application is not universally expected due to the huge state space, complex game dynamics, and a weakly structured heterogeneous action space of high dimensionality. The next section discusses solution techniques to solve these problems using the state-of-the-art approaches.

Model-Free Control

When the game dynamics are unknown, the go-to method has been RL (and particularly DQN). In this section, we show how such model-free control techniques fit into the paradigm of training agents to play modern games.

In our second case study, we consider a mobile game designed to engage many players for months exhibiting all of the challenges discussed in the introduction. The

[5]Unfortunately, in the dynamic programming every node will participate in the computation while it is often true that most of the nodes are not relevant to the shortest path problem in the sense that they are unlikely candidates for inclusion in a shortest path (Bertsekas 2005).

[6]Coincidentally, OpenAI has recently advocated for evolution strategies as an alternative for reinforcement learning in training agents to play games (Salimans et al. 2017).

designer's primary concern is to test how quickly an expert player can possibly progress in the game.

The state consists of ∼50 continuous and ∼100 discrete state variables. The set of possible actions α is a subset of a space A, which consists of ∼25 action classes, some of which are from a continuous range of possible action values, and some are from a discrete set of action choices. The agent has the ability to generate actions $a \in A$ but not all of them are valid at every game state since $\alpha = \alpha(s, t)$, i.e., α depends on the timestep and the game state. Moreover, the subset $\alpha(s, t)$ of valid actions is only partially known to the agent.

We rely on the game server to validate whether a submitted action is available because it is impractical to encode and pass the set α of available actions to the agent at every timestep. While the problem of a huge state space (Hoey and Poupart 2005; Spaan and Vlassis 2005; Porta et al. 2006), a continuous action space (Lillicrap et al. 2016), and a parametric actions space (Hausknecht and Stone 2015) could be dealt with, these techniques are not directly applicable to our problem. Finally, the game is designed to last tens of millions of timesteps, taking the problem of training a functional agent in such environment outside of the domain of previously explored problems.

We study game progression while taking only valid actions. As we already mentioned, the set of valid actions α is not fully determined by the current observation, and hence, we deal with a partially observable Markov decision process (POMDP). Given the practical constraints outlined above, it is infeasible to apply deep reinforcement learning to train agents in the game in its entirety. In this game, we show progress toward training an artificial agent that takes valid actions and progresses fast in the game like expert human players. We connect this game to our training pipeline with DQN and Rainbow agents, where we use a network with two fully connected hidden layers and ReLU activation.

We create an episode by setting an early goal state in the game that takes an expert human player ∼5 minutes to reach. We let the agent submit actions to the game server every second. We reward the agent with '+1' when they reach the goal state, '-1' when they submit an invalid action, '0' when they take a valid action, and '-0.1' when they choose the "do nothing" action. The game is such that at times the agent has no other valid action to choose, and hence they should choose "do nothing", but such periods do not last more than a few seconds.

We consider two different versions of the observation space, both extracted from the game engine. The first is what we call the "complete" state space. The complete state space contains information that is not straightforward to infer from the real observation in the game and is only used as a baseline for the agent. The polar opposite of this state space could be called the "naive" state space, which only contains straightforward information. The second state space we consider is what we call the "augmented" observation space, which contains

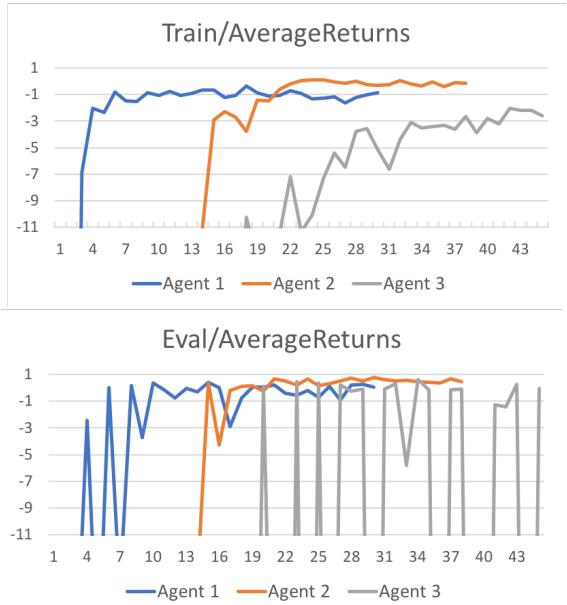

Figure 2: Average cumulative reward (return) in training and evaluation for the agents as a function of the number of iterations. Each iteration is worth ∼60 minutes of gameplay. The trained agents are: (1) a DQN agent with complete state space, (2) a Rainbow agent with complete state space, (3) a DQN agent with augmented observation space, and (4) a Rainbow agent with augmented observation space.

information from the "naive" state space and information the agent would reasonably infer and retain from current and previous game observations. Note that current RL techniques would have had difficulty in inferring this information from the frame buffer pixels which only constitute a partially observable state space.

We trained four types of agents as shown in Figure 2. As expected, the Rainbow agent converges to a better performance level while with more training episodes compared to the DQN agent. We believe that this is due to the fact that we did not optimize the extra hyperparameters in Rainbow for the best performance, which we intend to do in the future. We also see that the augmented observation space makes the training slower and also results in a worse performance on the final strategy. We intend to experiment with shaping the reward function for achieving different play styles. We also intend to investigate augmenting the replay buffer with expert demonstrations for faster training.

While we achieved a certain level of success using the outlined approach (streamlined access to the game state, and direct communication of actions to the game, followed by training using a deep neural network), we observe that the training within the current paradigm of RL remains costly. Specifically, even using the complete state space, it takes several hours to train a model that achieves a level of performance expected of an ex-

pert human player on this relatively short episode. This calls for the exploration of complementary approaches to augment the training process.

Model-Based Control & Imitation Learning

We have shown the value of simulated agents in a fully modeled game, and the potential of training agents in a complex game to model player progression. We can take these techniques a step further and make use of agent learning to help build the game itself. Instead of applying RL to capture player behaviors, we consider an approach to gameplay design where the player agents learn behavior policies from the game designers.

To bridge the gap between the agent and the designer, we introduce imitation learning (IL) to our system. In the present application, IL allows us to translate the intentions of the game designer into a primer and a target for our agent learning system. Learning from expert demonstrations has traditionally proved very helpful in training agents. In particular, the original Alpha Go (Silver et al. 2016) used expert demonstrations in training a deep Q network. There are also other cases where the preferred solution for training agents would utilize a few relatively short demonstration episodes played by the software developers or designers at the end of the current development cycle.

As our last case study, we consider training artificial agents in an open-world video game, where the game designer is interested in training non-player characters in the game that follow certain behavioral styles. We need to efficiently train an agent using demonstrations capturing only a few key features. The training process has to be computationally inexpensive and the agent has to imitate the behavior of the teacher(s) by mimicking their relevant style (in a statistical sense) for implicit representation of the teacher's objectives.

Casting this problem directly into the RL framework is complicated by two issues. First, it is not straightforward how to design a rewarding mechanism for imitating the style of the expert.[7] Second, the RL training loop often requires thousands of episodes to learn useful policies, directly translating to a long training time.

We propose a three-component solution to the stated problem.

- The first component is an ensemble of multi-resolution Markov models capturing the style of the teacher(s) with respect to key game features.
- The second one is a DNN trained as a supervised model on samples bootstrapped from an agent playing the game following the Markov ensemble.
- Lastly, we enable an interactive medium where the game designer can take the controller back at any time to provide more demonstration samples.

[7]While inverse RL aims at solving this problem, its applicability is not obvious given the reduced representation of the huge state-action space that we deal with and the ill-posed nature of the inverse RL problem.

Multi-resolution Markov agent. We start with capturing the demonstration data which consists of a few engineered features reported by the game at every timestep. Total dimensionality of an individual frame data is ∼20 variables with, some of them reported only once every few timesteps. We also record actions that are controller inputs from a human player. We intend to provide a more complete description of this setup by publishing a preprint of our internal report (Borovikov and Harder 2018). The key point is that the data from the demonstrations has low dimensionality and is sparse, but sufficient to capture the main characteristics of the core gameplay loop of a first person shooter in an open world.

To build our Markov agent we use a direct approach to style reproduction inspired by natural language processing literature (see review (Zhai 2008)). The demonstrations are converted to symbolic sequences using a hierarchy of multi-resolution quantization schemes with different levels of details for both the continuous data and the discrete channels. The most detailed quantization and higher order Markov models are able to reproduce sequences of human actions in similar situations with high accuracy, thus capturing the gameplay style. The coarsest level corresponds to a Markov agent blindly sampling actions from the demonstrations. The multi-resolution ensemble of Markov models provides an initial way of generalizing the demonstration data. The ensemble is straightforward to build and the inference is essentially a look-up process. We observe that even such a basic approach provided considerable mileage towards solving the stated problem.

Bootstrapped DNN agent. We treat the existing demonstrations as a training set for a supervised learning problem where we need to predict the next action from a sequence of observed state-action pairs. This approach has proved to be useful in pre-training of self-driving cars (Montemerlo et al. 2006). Since our database of demonstrations is relatively small, it is desirable to generate more data by bootstrapping the dataset for which we use our base Markov agent interacting with the game.

Such a bootstrap process is easy to parallelize since we can have multiple simulations running without the need to cross-interact as in some learning algorithms like A3C (Mnih et al. 2016). The generated augmented data set is used to train a DNN that predicts the next action from the already observed state-action pairs. Due to partial observability, the low dimensionality of the feature space results in fast training in a wide range of model architectures, allowing a quick experimentation loop. We converged on a simple model with a single "wide" hidden layer for motion control channels and a DNN model for discrete channels responsible for turning on/off actions like sprint, firing, climbing. The approach shows promise even with many yet unexplored opportunities to improve its efficiency.

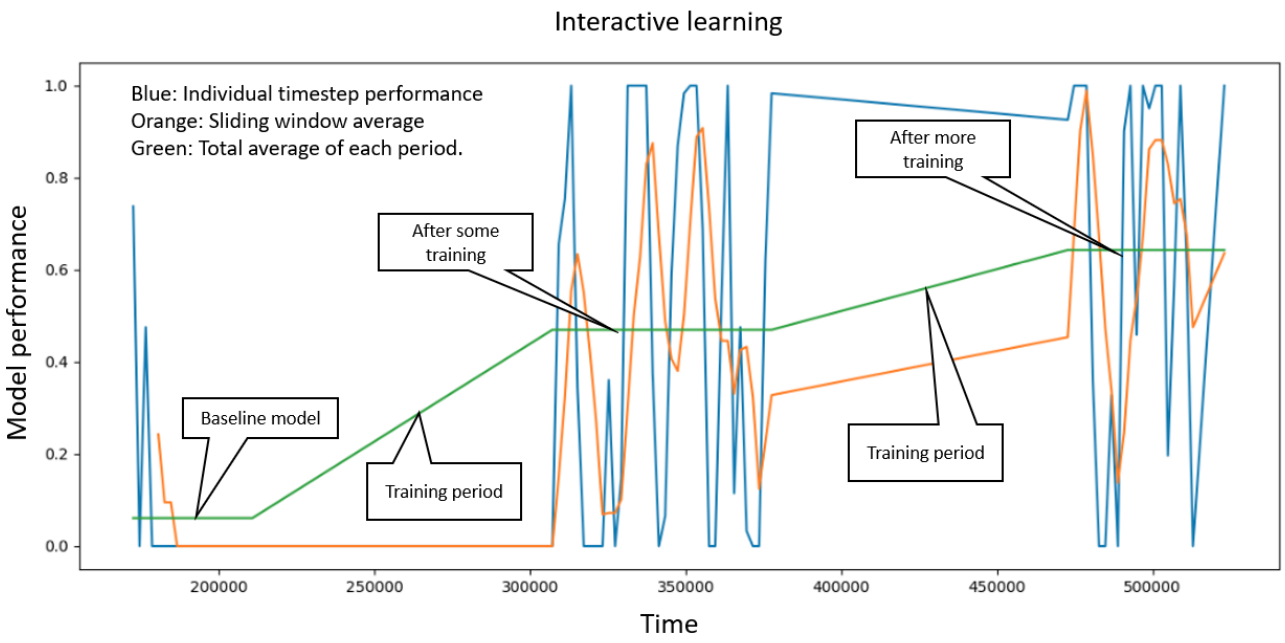

Figure 3: Model performance (as the ability to respond to the current state with an inferred action) during interactive training from demonstrations in a proprietary open-world game.

Interactive learning. While bootstrapping can help training a better model, the quality of the final trained model is still limited by the amount of information contained in the relatively small set of demonstrations. Hence, it is highly desirable to obtain further information from the game designer, particularly in unexplored and underexplored parts of the state space where the trained model has little hope of generalizing.

We find that such a direct approach provides an opportunity to make the entire process of providing the demonstrations interactive. The interactivity entirely comes from the compound hierarchical nature of the initial ensemble of models, making it easy to add new ones to the set of already existing sub-models. In practical terms, it enables adding new demonstrations to directly override or augment already recorded ones. This allows a high level of interactivity for supporting newly added features or updating the existing model otherwise. The designer can directly interact with the game, select a particular moment where a new demonstration is required, adjust the initial location of the character object, and run a short demonstration without reloading the game. The interactivity eliminates most of the complexity of the agent design process and brings down the cost of gathering data from under-explored parts of the state space.

A designer can start with the most basic gameplay, e.g., approach the target, then add more elements, e.g., attack the target, followed by more sophisticated gameplay. To provide additional feedback to the designer, we provide a (near) real-time chart showing how well the current model generalizes to the current conditions in the game environment.

An example of such interactive training chart is reported in Figure 3. The prolonged period of training may increase the size of the model with many older demonstrations already irrelevant, not used for inference, but still contributing to the model size. Instead of using rule-based compression of the resulting model ensemble, we consider a DNN from the ensemble of Markov models via a novel bootstrap using the game itself as the way to compress the model representation and strip off obsolete demonstration data. Using the proposed approach, we train an agent that satisfies the design needs in only a few hours of interactive training.

Table 1 illustrates the computational resources that solving this problem with the outlined 3-step process required as compared to training 1v1 agents in Dota 2 (OpenAI Five 2018). While we acknowledge that the goal of our agent is not to play optimally against the opponent and win the game, we observe that using model-based training augmented with expert demonstrations to solve the Markov decision process, in a game even more complex than Dota 2, results in huge computational savings compared to an optimal reinforcement learning approach.

Concluding Remarks

In this paper we have described our approach to game-playing agents that considers the player experience over the agent's ability to win. We feel this is a more challenging but more beneficial route in understanding how players interact with games, and how to modify the games to change and improve player interaction. In each of our three case studies, we give a simple example of how this approach to game-playing agents can yield valuable insight, and even drive the construction of the game itself. We believe that this is just the beginning of a long and fruitful exploration into experience-oriented game-playing agents that will not only deliver radical improvements to the games we play, but will be another major milestone on the roadmap of AI.

Table 1: Comparison between OpenAI 1v1 Dota 2 Bot (OpenAI Five 2018) training metrics and training an agent via bootstrap from human demonstrations in a proprietary open-world game. While the objectives of the training are different, the environments are comparable.[9] The metrics below highlight the details of practical training of agents during the game development cycle.

	OpenAI 1v1 Bot	Bootstrapped Agent
Experience	∼300 years (per day)	∼5 min human demonstrations
Bootstrap using game client	N/A	×5-20
CPU	60,000 CPU cores on Azure	1 local CPU
GPU	256 K80 GPUs on Azure	N/A
Size of observation	∼3.3kB	∼0.5kB
Observations per second of gameplay	10	33

Acknowledgement

The authors are thankful to the anonymous reviewer who brought to their attention the related line of work of (Holmgard et al. 2018).

References

AI & Compute. 2018. [Online, May 2018] https://blog.openai.com/ai-and-compute.

Bellemare, M. G.; Castro, P. S.; Gelada, C.; and Kumar, S. [Online, 2018] https://github.com/google/dopamine.

Bertsekas, D. P. 2005. *Dynamic programming and optimal control*, volume 1. Athena scientific Belmont, MA.

Borovikov, I., and Harder, J. 2018. Learning models to imitate personal behavior style with applications in video gaming. Technical report, Electronic Arts Digital Platforms, Data & AI. In preparation.

Browne, C., and Maire, F. 2010. Evolutionary game design. *IEEE Transactions on Computational Intelligence and AI in Games* 2(1):1–16.

Chaslot, G.; Bakkes, S.; Szita, I.; and Spronck, P. 2008. Monte-carlo tree search: A new framework for game ai. In *AIIDE*.

Coulom, R. 2006. Efficient selectivity and backup operators in Monte-Carlo tree search. In *International conference on computers and games*, 72–83. Springer.

De Mesentier Silva, F.; Lee, S.; Togelius, J.; and Nealen, A. 2017. AI-based playtesting of contemporary board games. In *Foundations of Digital Games 2017*. ACM.

Deep Blue. 1997. [Online] http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue.

Harmer, J.; Gisslen, L.; del Val, J.; Holst, H.; Bergdahl, J.; Olsson, T.; Sjoo, K.; and Nordin, M. 2018. Imitation learning with concurrent actions in 3d games.

Hausknecht, M., and Stone, P. 2015. Deep reinforcement learning in parameterized action space. *arXiv preprint arXiv:1511.04143*.

Hessel, M.; Modayil, J.; Van Hasselt, H.; Schaul, T.; Ostrovski, G.; Dabney, W.; Horgan, D.; Piot, B.; Azar, M.; and Silver, D. 2017. Rainbow: Combining improvements in deep reinforcement learning. *arXiv preprint arXiv:1710.02298*.

Heyden, C. 2009. *Implementing a computer player for Carcassonne*. Ph.D. Dissertation, Maastricht University.

Hoey, J., and Poupart, P. 2005. Solving POMDPs with continuous or large discrete observation spaces. In *IJCAI*, 1332–1338.

Holmgard, C.; Green, M. C.; Liapis, A.; and Togelius, J. 2018. Automated playtesting with procedural personas with evolved heuristics. *IEEE Transactions on Games*.

Hom, V., and Marks, J. 2007. Automatic design of balanced board games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 25–30.

Huchler, C. 2015. An mcts agent for ticket to ride. Master's thesis, Maastricht University.

Kocsis, L., and Szepesvári, C. 2006. Bandit based Monte-Carlo planning. In *European conference on machine learning*, 282–293. Springer.

Krucher, J. 2015. Algorithmically balancing a collectible card game. Bachelor's thesis, ETH Zurich.

Liapis, A.; Yannakakis, G. N.; and Togelius, J. 2013. Sentient sketchbook: Computer-aided game level authoring. In *FDG*, 213–220.

Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2016. Continuous control with deep reinforcement learning.

Mahlmann, T.; Togelius, J.; and Yannakakis, G. N. 2012. Evolving card sets towards balancing dominion. In *Evolutionary Computation (CEC), 2012 IEEE Congress on*, 1–8. IEEE.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529.

Mnih et al., V. 2016. Asynchronous Methods for Deep Reinforcement Learning. *arXiv:1602.01783v2*.

Montemerlo, M.; Thrun, S.; Dahlkamp, H.; and Stavens, D. 2006. Winning the darpa grand challenge with an ai robot. In *In Proceedings of the AAAI National Conference on Artificial Intelligence*, 17–20.

OpenAI Five. 2018. [Online, June 2018] https://openai.com/five.

OpenAI Gym. 2016. [Online] https://gym.openai.com.

Pang, Z.-J.; Liu, R.-Z.; Meng, Z.-Y.; Zhang, Y.; Yu, Y.; and Lu, T. 2018. On reinforcement learning for full-length game of starcraft. *arXiv preprint arXiv:1809.09095*.

Porta, J. M.; Vlassis, N.; Spaan, M. T.; and Poupart, P. 2006. Point-based value iteration for continuous POMDPs. *Journal of Machine Learning Research* 7(Nov):2329–2367.

PySC2. 2017. [Online] https://github.com/deepmind/pysc2.

Robilliard, D.; Fonlupt, C.; and Teytaud, F. 2014. Monte-carlo tree search for the game of "7 wonders". In *Computer Games*. Springer. 64–77.

Salge, C., and Mahlmann, T. 2010. Relevant information as a formalised approach to evaluate game mechanics. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, 281–288. IEEE.

Salimans, T.; Ho, J.; Chen, X.; Sidor, S.; and Sutskever, I. 2017. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*.

Shaker, N.; Shaker, M.; and Togelius, J. 2013. Ropossum: An authoring tool for designing, optimizing and solving cut the rope levels. In *AIIDE*.

Silva, F. D. M.; Borovikov, I.; Kolen, J.; Aghdaie, N.; and Zaman, K. 2018. Exploring gameplay with AI agents. In *AIIDE*.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529(7587):484–489.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; et al. 2017. Mastering the game of Go without human knowledge. *Nature* 550(7676):354.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science* 362(6419):1140–1144.

Smith, A. M.; Nelson, M. J.; and Mateas, M. 2010. Ludocore: A logical game engine for modeling videogames. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, 91–98. IEEE.

Smith, G.; Whitehead, J.; and Mateas, M. 2010. Tanagra: A mixed-initiative level design tool. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, 209–216. ACM.

Spaan, M. T., and Vlassis, N. 2005. Perseus: Randomized point-based value iteration for POMDPs. *Journal of artificial intelligence research* 24:195–220.

Summerville, A.; Snodgrass, S.; Guzdial, M.; Holmgård, C.; Hoover, A. K.; Isaksen, A.; Nealen, A.; and Togelius, J. 2018. Procedural content generation via machine learning (PCGML). *IEEE Transactions on Games* 10(3):257–270.

Szita, I.; Chaslot, G.; and Spronck, P. 2009. Monte-carlo tree search in settlers of catan. In *Advances in Computer Games*, 21–32. Springer.

Togelius, J.; Yannakakis, G. N.; Stanley, K. O.; and Browne, C. 2011. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games* 3(3):172–186.

Treanor, M.; Zook, A.; Eladhari, M. P.; Togelius, J.; Smith, G.; Cook, M.; Thompson, T.; Magerko, B.; Levine, J.; and Smith, A. 2015. AI-based game design patterns.

Vinyals, O.; Ewalds, T.; Bartunov, S.; Georgiev, P.; Vezhnevets, A. S.; Yeo, M.; Makhzani, A.; Küttler, H.; Agapiou, J.; Schrittwieser, J.; et al. 2017. StarCraft II: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*.

Yannakakis, G. N.; Liapis, A.; and Alexopoulos, C. 2014. Mixed-initiative co-creativity. In *Proceedings of the 9th Conference on the Foundations of Digital Games*.

Zhai, C. 2008. Statistical language models for information retrieval a critical review. *Foundations and Trends in Information Retrieval* 2(3):137–213.

Zhu, T.; Wang, B.; and Zyda, M. 2018. Exploring the similarity between game events for game level analysis and generation. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, 8. ACM.