Game Engine Learning from Video

Matthew Guzdial, Boyang Li, Mark O. Riedl
School of Interactive Computing, Georgia Institute of Technology
mguzdial3@gatech.edu, boyangli@gatech.edu, riedl@cc.gatech.edu

Abstract

Intelligent agents need to be able to make predictions about their environment. In this work we present a novel approach to learn a forward simulation model via simple search over pixel input. We make use of a video game, Super Mario Bros., as an initial test of our approach as it represents a physics system that is significantly less complex than reality. We demonstrate the significant improvement of our approach in predicting future states compared with a baseline CNN and apply the learned model to train a game playing agent. Thus we evaluate the algorithm in terms of the accuracy and value of its output model.

1 Introduction

Automated Game Understanding represents the field of work devoted to applying artificial intelligence to derive knowledge about video game systems for the purposes of game play, design, or critique. We define *Automatic Game Understanding* as the problem of developing formalized, complete models of the underlying processes in games. To this point its greatest success has been in the field of automated game playing. From retro Atari games with Deep Mind [Mnih *et al.*, 2013] to board games such as Go [Churchland and Sejnowski, 2016] there is an understanding that playing games represents fundamental AI research. However we contend that games can offer a testbed to fundamental research beyond game playing.

Outside of the mass of work on automated game playing and some work on automated game critique [Zook *et al.*, 2015; Canossa and Smith, 2015; Guzdial *et al.*, 2016] there has been little effort to automatically understand the systems that power games. However, this is an important area for research as it represents the problem of learning a model of a game's simplified physics, which could then scale up to more realistic domains to parallel work in automated game playing.

In this paper, we describe a Game Engine Search algorithm capable of learning a game engine from gameplay data. We define a *game mechanic* as discrete rule that maps a cause to an effect (e.g. if the player is falling and hits the ground then the player stops falling). We define a *game engine* as the backend set of a game's mechanics. The algorithm functions by scanning through the output of the game engine, represented as video, and iteratively improving a hypothesized engine to minimize errors through greedy search. To our knowledge this represents the first approach capable of deriving a game engine (a simulator of a specific game) only from output from another game engine (gameplay video). We test our technique with gameplay video from Super Mario Bros., a classic platformer, and present evidence that our technique outputs a learned engine very similar to the true game engine. We anticipate this technique to aide in applications for automated game playing, explainable AI, gameplay transfer, and game design tasks such as automated game design.

The remainder of this paper is organized as follows. We start by providing some background on automated game playing and understanding. Section 3 presents the proposed algorithm Game Engine Search and Section 4 presents our experimental evaluations and their results.

2 Background

2.1 Automated Game Playing

Automated game playing stands as a goal for artificial intelligence from its earliest days [Brooks, 1999], and has had great success in classic games like chess and go. Most closely related to this work are approaches to utilize pixel input to learn to play retro video games. For example, Mnih et al. [Mnih *et al.*, 2013] used deep convolutional networks to learn how to the play Atari games from pixel input. Later work has brought automated game playing to Doom [Hafner, 2016], and even for more modern games such as Minecraft [Oh *et al.*, 2016]. Although these systems and our own process pixels to learn models of a game, our system focuses on deriving game rules instead of learning to play the game. The end models differ: Mnih et al.'s extracted model will recognize the action for the player to activate in each state, and will ignore any elements that do not impact reward (score) and those not currently visible. On the other hand, our system includes decorative elements and can reason about the off-screen effects of actions (for example, killing an enemy offscreen by recognizing when an enemy does not appear when expected).

2.2 Automated Understanding

Automated understanding, also known as common sense learning [Singh *et al.*, 2002] or hypothesis generation [Gettys and Fisher, 1979], is the problem of taking a sequence of events, building some model that explains the events, and using the model to predict future events. A common modern approach to this problem relies on convolutional neural nets, learning from sequences of images [Ranzato *et al.*, 2014; Vukotić *et al.*, 2017]. These techniques take in a frame and predict the next frame in a sequence, without modeling the underlying effects. Perhaps closest to our own work Selvaraju et al. [Selvaraju *et al.*, 2016] derive an explanation for a convolutional neural net's prediction, without learning a general model of the physics at play in the frames.

An alternate approach to automated understanding common to reinforcement learning techniques is *forward model learning* [Kober *et al.*, 2013]. In forward model learning a transition function is learned that allows an agent to make predictions in a simplified state space from sequences of state changes. This approach has been applied to navigation control [Ng *et al.*, 2006], video games such as Starcraft [Uriarte and Ontanón, 2015] and arcade-like games [Braylan and Miikkulainen, 2016]. Closest to our work, Ersen and Sariel [2015] derive a model of the environment for a puzzle game composed of formal logic rules based on hand-authored sequences of events. Our work differs from typical forward model techniques as it learns from and makes predictions in a pixel-level state space.

2.3 Automated Game Understanding

The field of automated game understanding is much more recent than automated game playing. Martens et al. [Martens *et al.*, 2016] made use of case based reasoning to derive possible meanings from specially constructed representations of games. Most closely related to this work Sumerville et al. [2017a] demonstrated the ability to automatically derive object characteristics (e.g. "hurts player", "is killed by player) from gameplay logs. We differ from this work by learning an entire game engine, capable of forward simulation and by deriving these models from video rather than gameplay logs. Despite this work being the first to attempt to learn game engines in this manner both Gow and Corneli [2015] and Summerville et al. [2017b] propose this problem and suggest possible solutions to it.

There is a significant body of prior work at generating game rules, with the majority of prior work relying on a human-authored game engine or space of possible game rules upon which an optimization process runs [Nelson *et al.*, 2016]. Togelius [Togelius, 2011] presents an iterative development paradigm for a human player to create a game engine during play, as an intelligent system attempts to learn the "causes" behind user-specified effects to create rulesets. Both Cook et al. [2013], and Zook and Riedl [2014] make use of approaches that generate player-character game rules within human-authored game engines, and verify these rules with forward simulation.

3 System Overview

The goal of our work is to develop a computational system capable of learning a game engine, the backend set of rules that runs a game, from input video. First, our system scans each input video frame to determine the set of objects present per

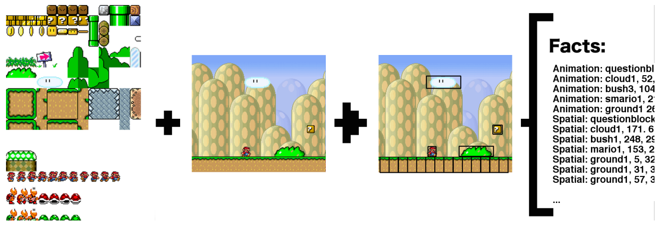

Figure 1: Visualization of the frame parsing process in the Infinite Mario engine . A frame is parsed to locate spritesheet elements in a frame, which then is translated into a list of facts.

frame. Second, we run a greedy matching algorithm across pairs of adjacent frames to determine how objects change between frames. Lastly we parse each frame and run an engine search algorithm when the second frame differs from the predicted next frame by more than some set threshold.

3.1 Parsing Frames

We begin by supplying our system with two things: a set of videos to learn from and a sprite palette as seen on the left of Figure 1. By sprite palette we indicate the set of $sprites$ or individual images used to build levels of a 2D game. For this proof-of-concept we make use of a fan-authored sprite sheet for the game Super Mario Bros. and longplay video that represents a single player playing through the entirety of the game. With these elements the system makes use of OpenCV [Pulli *et al.*, 2012], an open-source machine vision toolkit, to determine the number and placement of sprites in each frame of the video. All together we transform the initial pixel input into a set of sprites and their spatial positions in each frame.

Given a sequence of frames, where each frame is defined as the set of sprites and their locations, we can run a simple greedy matching algorithm to match each sprite to its closest neighbor (both visually and spatially) in each adjacent frame. In the case where there are an unequal number of sprites we create "empty" sprites to match the remainders. This will occur in the cases where a sprite is created or destroyed (e.g. when Mario shoots a fireball a sprite is created and when Mario jumps on an enemy a sprite is destroyed).

The final step of parsing each frame alters the representation from sprites to a set of facts or percepts. Each "type" of fact requires a hand-authored function to derive it from an input frame. We list the set of fact types and the means by which they are derived below.

- *Animation:* The animation fact is simply a list of each sprite image according to its original filename, width, and height (e.g. if an image "mario1.png" of size [26px, 26px] was found at position 0,0 that would create the Animation fact $\langle mario1, 26, 26 \rangle$).
- *Spatial:* The spatial fact is the sprite's location in the frame, listed as the sprite's filename and it's x and y position in the frame.
- *RelationshipX:* The relationship in the x-dimension of each pair of sprites in the initial frame, of the form (sprite1's filename, sprite1's closest edge to sprite2, distance in pixels, sprite 2's filename, sprite 2's closest edge

to sprite1). This condition allows the system to learn collision rules. For example, Mario's x-velocity changes from positive to zero when Mario's right side hits some other sprite's left side.

- *RelationshipY:* The exact same as the above condition but for the y-dimension. This allows the system to learn collision rules in the y-dimension. Such as Mario stops falling when his feet (bottom of his sprite) hit the top of some other sprite.
- *VelocityX:* This fact captures information about a sprite's velocity in the x-dimension and is derived according to the greedy matching to the next frame's sprite. For example if Mario is at position [0,0] in frame 1 and [10,0] in frame 2, then frame 1 will include the fact *VelocityX*: $\langle mario, 10\rangle$.
- *VelocityY:* This fact type captures the same information as the prior type but for the y-dimension.
- *CameraX:* This is a unique fact type that simply stores a value representing how far along the camera is in the level. We included this as original attempts at this technique had the issue that when the camera moved one direction, all stationary objects would appear (according to the other facts) to move in the opposite direction. We derive this fact by looking at the average shift in pixels of each sprite from frame to frame, therefore avoiding the issue of sliding in the opposite direction.

Notably each fact can be linked back to the characteristics of a sprite that it arose from. In other words if the spatial fact $\langle mario1, 0, 0\rangle$ changes to $\langle mario1, 100, 100\rangle$ the system can move the sprite mario1 to position [100, 100]. This concept is the primary method by which rules (represented in our system by changes in facts) act upon a frame.

3.2 Engine Learning

Our engine learning approach seeks to derive a game engine that can predict the changes observed in the set of parsed frames derived in the prior section. A game engine, as defined in this approach is a set of rules where each rule is a single IF-THEN production with the If represented by a set of conditional facts and the Then representing a change as a pair of facts. For example, a rule might change a *VelocityX* fact from $\langle mario1, 0\rangle$ to $\langle mario1, 5\rangle$ for a given frame (THEN) given the set of facts that make up its conditions are present in that frame (IF).

At a high level, the game engine learning approach scans through the sequence of parsed frames and begins a search for a set of rules that explains any sufficient difference between the predicted and actual frame. If a game engine is found that reduces the difference to some threshold, then the scan begins again to ensure that this new engine has not lost the ability to accurately predict prior frames. We express the algorithm to scan through the frames in Algorithm 1 and the engine search algorithm in Algorithm 2.

Algorithm 1 gives the simple algorithm that scans through the sequence of frames to identify opportunities to learn for the engine search algorithm, Algorithm 2. The distance function on line five represents a pixel-by-pixel distance function

Algorithm 1: frame scan

```
input : A sequence of parsed frames of size $f$, and
        threshold $\theta$
output: A game engine
1  e ← new Engine();
2  cF ←frames [0];
3  while $i \leftarrow 1$ to $f$ do
4      Check if this engine predicts within the threshold;
5      frameDist ←Distance(e, cF, $i+1$);
6      if frameDist $< \theta$ then
7          cF ←Predict (e, cF, $i+1$);
8          continue;
9      Update engine and start parse over;
10     e ← EngineSearch(e, cF, $i+1$);
11     $i \leftarrow 1$;
12     cF ←frames [0];
```

between a ground truth frame $(i + 1)$ and a predicted frame (derived from running the current frame cF through the current engine e) and counts the number of pixels that do not match (0 if a perfect match, 1 otherwise). We choose to use pixel distance rather than a difference in terms of each frame's set of facts as we ultimately care about the final engine being able to produce the same set of frames, not a list of facts. The $Predict$ function on line seven returns the closest frame to the ground truth frame $(i+1)$ given the current frame cF and engine e. Multiple frames can be produced by a given engine for a given frame due to the possibility of "input" rules (e.g. the player choosing to press left, right, etc), which we discuss in more detail below. We make use of a predicted frame rather than setting the current frame to the previous ground truth frame in order to build a more general game engine, rather than one that only explains frame-to-frame changes.

Algorithm 2 gives the engine search algorithm, representing the bulk of this approach. It can be understood as a greedy search to find a set of rules that creates a predicted frame within some threshold of the actual ground truth frame. The primary means of accomplishing this is in the generation of neighbors for a given engine (as seen in line ten). Neighbors are generated via (1) adding rules, (2) modifying a rule's set of condition facts, (3) modifying a rule to cover an additional sprite, and (4) modifying a rule into a control rule.

Adding a rule to a game engine requires picking out a pair of facts of the same type, one from the current frame and one from the goal frame, which differ in some way (e.g. one VelocityX fact with values $\langle mario1, 0\rangle$ and one with values $\langle mario1, 5\rangle$). This pair of facts represents the change that the rule handles. All other facts from the current frame make up the initial condition set for this rule, which ensures it'll be activated on this current frame. This is initially a very specific set of facts, and will include facts that are not truly necessary to activate the rule. For example, Mario can move right as long as he is not blocked to the right but an initial rule that moves Mario right might include the condition that he has a cloud above him, a bush to his left, etc.

The set of conditions in an initial rule can later be mini-

Algorithm 2: Engine Search

```
input : An initial engine $engine$, the current frame
        $cF$, and goal frame $g$
output: A new engine $finalEngine$
closed ←[];
open ←PriorityQueue();
open.push(1,engine);
while open is not empty do
    node ←open.pop();
    if node [0] $< \theta$ then
        return node [1];
    engine ←node;
    closed.add(engine);
    for Neighbor n of engine do
        if n in closed then
            continue;
        d ←Distance(engine, cF, g);
        open.push(d +engine.rules.length,n);
```

mized by the second type of neighbor, modifying a rule's set of condition facts. In this case, the intersection of an existing rule's condition facts and the current set of condition facts is taken, and set as the set of a condition facts for a modified rule in a neighboring engine. If this neighbor decreases the pixel distance of the predicted frame from the goal frame, it will be more likely to be chosen above a neighbor that simply adds a new rule as engines are placed into the PriorityQueue (*open*) according to their pixel distance and the number of rules in the engine. This preferences smaller, more general engines. The third type of neighbor works in much the same way, except that it expands the set of changes it can make. For example, many types of enemies "disappear" when Mario jumps on them (in the language of the engine, go from an Animation fact with values to one without), and therefore a single rule can handle all of these cases.

The final type of neighbor that the system handles, changes a rule from being handled normally to being considered a "control" rule. This handles the case of rules that the player makes decisions upon. For example, the input that controls the player character (moving Mario left, right, and jumping). When $Predict$ is called (either in the algorithms presented or within $Distance$), this leads to a branch of two possible frames for each control rule (the player chooses to move right or doesn't). This can lead to a large number of final predicted frames as an engine can contain many control rules that each can be either active or not. The current goal frame is therefore used in $Predict$ to select from this final set the frame that is closest to the ground truth. Notably, these rules still contain a set of condition facts that must be present for the rule to fire at all, meaning not all condition facts can be active every frame (e.g. Mario cannot jump once already in the air).

The frame scan, breaking into occasional engine search tasks, continues until it reaches the end of the sequence of parsed frames. At this point we can guarantee that we have an engine that can predict the entire sequence of frames from the initial frame. Notably this means that the engine can only reflect changes that actually occur in the input sequence of parsed frames. In addition, the choice of threshold θ has significant impact on the final engine. A θ that is too large will lead to an engine that does not represent smaller changes, since the engine search algorithm gives preference to engines that decrease the pixel distance by large amounts. Therefore, a final engine might miss the instances that cause a sprite to begin to move or alter its position. But the smaller the value of θ the longer the algorithms will run for, which we explain further in the next section.

3.3 Limitations

The algorithm presented above has a number of clear drawbacks. First, it is comparatively slow depending on the threshold θ provided. While this is an offline process, meant to only run once, the running time can still be prohibitive with lower values of θ (running for up to two weeks for a single game level on a 2013 iMac). We have begun to explore the ability to parallelize this process and "merge" the learned engines according to the merge criteria discussed above and have found success. We hope to explore this in future work.

This algorithm requires that each instance of a sequence of input data be represented as a series of facts. This requires authoring to determine the space of possible facts as we discussed in Section 3.1. This makes the algorithm less applicable to more complex domains (e.g. real video). However, we anticipate that with a sufficient space of possible facts our technique could model any environment (whether video game or otherwise) where the majority of action occurs on screen. We identify the ability to automatically derive the space of possible facts as an important avenue for future work.

We note that the current technique only takes as training a single sequence of video, but that it can be extended to additional videos by treating the start of each new video as a new point to start over at when the algorithm has reached the end of one video. Further, we anticipate that this would only improve the end model as a single sequence of video cannot be expected to contain all gameplay events.

4 Evaluation

In this section we present results from two distinct evaluations meant to demonstrate the utility of our system. Given the novelty of this system we had no clear baseline to compare against. Instead we make use of two baselines to demonstrate two important characteristics of the learned engine: (1) its ability to accurately forward simulate and predict upcoming states and (2) the quality of those predictions to a game-playing task. For the first evaluation we compare the frames predicted by our learned engine against a naive baseline and a Convolutional Neural Net (CNN) constructed for frame-to-frame prediction [Ranzato *et al.*, 2014]. For the second evaluation we evaluate the application of the learned engine's knowledge to training a game playing agent, compared to agents with access to no game engine or the true game engine.

For both evaluations we make use of a game engine trained on gameplay video of a single level, Level 1-1 of Super Mario

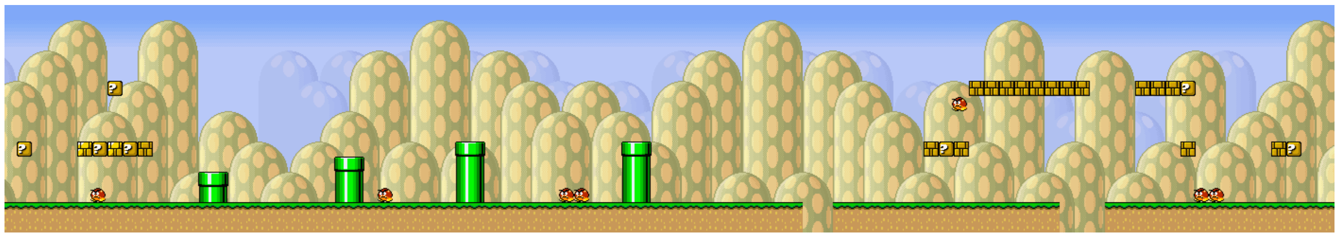

Figure 2: A section of Level 1-1 of Super Mario Bros. represented in the Infinite Mario engine.

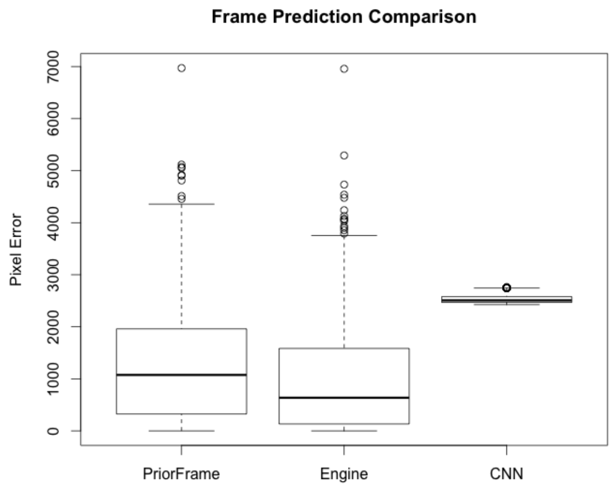

Figure 3: Comparison of the pixel error between the previous frame, engine's prediction, and the CNN

Bros.. We chose to do this in order to demonstrate the approach's ability to learn from a single example. In addition we set θ to zero, to ensure the best possible output engine.

4.1 Frame Accuracy Evaluation

In our first evaluation we made use of a common train and test procedure. Notably we selected two examples of gameplay footage of two distinct players playing Level 1-1 of Super Mario Bros. We made use of gameplay footage that had been evaluated in prior work by Summerville et al. [2016]. In particular we drew on the two most different players and their respective gameplay videos which Summerville et al. describe as the "speedrunner" and "explorer". In this way we could be sure that the two gameplay videos represented significantly different approaches to the same level. In particular, we trained on the "speedrunner" video, video of the player who took the least time in Level 1-1, thus making the training problem as difficult as possible.

We drew on prior work in predicting frame transitions with convolutional neural networks (CNN) for a baseline, specifically work by Ranzato et al. [2014]. We also implemented the non-time-dependent network from Vukotić et al. [2014] but found it performed strictly worse, and so do not include it in our results. We made use of the same convolutional neural network architecture described in the Ranzato et al. paper, a network with four convolutional layers with 128 filters each, each making use of relu activation. For further details on the network please see the original paper. To accommodate our input to this convolutional neural net, we had to shrink all our frames and map all pixels to grayscale values. Thus we went from 416x364 pixel RGB images to 104x91 pixel grayscale images, which brought the images more into line with the input of the original work. We trained this CNN on the same training data as the engine learning approach to better demonstrate their relative performance, meaning that the CNN was trained on only 1569 frames. We trained until the CNN's performance on the test set converged to avoid overfitting.

Given that we could not directly compare the convolutional neural network error with the pixel distance function described above we instead converted all predicted frames from the learned engine to the same dimensions as the CNN frames and converted them to grayscale. We then defined a simple matrix subtraction function to compare two grayscale frames of size 104x91 pixels (varying from 0 to 9,464). We can then compare the output of the maximally likely CNN output with the output of the learned engine.

We present the results of this evaluation in Figure 3. Notably we also include the naive approach of predicting the previous frame, as an "empty" engine would simply return the prior frame. While the CNN is much more consistent than either of the other two predictions, we find that our learned engine predicts frames significantly more similar to the true frame (Wilxocon paired test, $p < 2.2e^{-16}$). Further our engines predicted frames were significantly more similar to the true frame than the "naive" assumption of predicting the prior frame (Wilxocon paired test, $p < 1.247e^{-6}$)). This provides strong evidence that our engine derives an accurate, general model of Level 1-1 of Super Mario Bros. from a single example of parsed input frames.

4.2 Gameplay Learning Evaluation

Our second evaluation seeks to address the extent to which the learned game engine represents procedural knowledge valuable to other tasks besides predicting frames. While we anticipate that the learned game engine could be utilized for a variety of tasks, we identify game playing as an obvious application. However, there is no direct means of measuring whether two game engines can support the same gameplay. Directly comparing the rules of two distinct game engines does not answer this question as the same gameplay "effect" could be represented by several different rules (e.g. changing the velocity of a sprite versus removing a sprite and causing it to reappear somewhere else). Therefore, we instead settled on indirectly measuring the learned engine's ability to afford accurate gameplay by comparing its performance in training

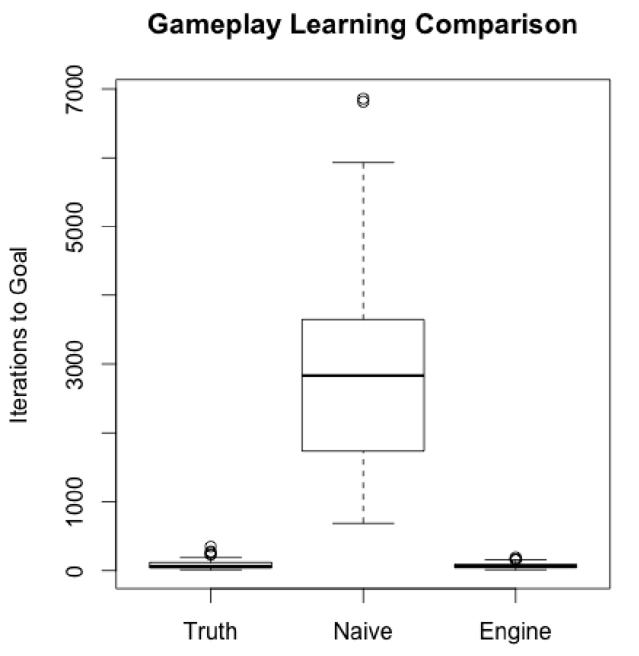

Figure 4: Comparison of the iterations required to reach the goal across the three agents.

a game playing agent.

For a baseline we drew on the "Infinite Mario" engine used for the Mario level playing AI competition [Togelius *et al.*, 2010] and constructed a simple reinforcement learning agent for the engine utilizing the value iteration algorithm. We further constructed two variations on this agent. The first made use of the native forward simulation available in the "Infinite Mario" engine to construct an initial policy rather than relying on the typical approach of a uniformly random initial policy. In particular we set the initial value for each state s to $V[s] = max_a \sum_{s'} R(s, a, s')$ according to a reward function that gave infinite reward for reaching the end of the level and negative infinite reward for deaths. This can be understood as changing the value iteration problem into a simple greedy agent, and represents a theoretical "lower bound" for predicting the quality of the next state. In comparison, we constructed an agent that instead made use of forward simulation according to the learned game engine (with numerical values and names converted to the appropriate equivalent in the "Infinite Mario" engine). For each agent we defined the current state as the size of a screen and Mario's current x and y velocity. Since each of these agents in essence made use of slightly different reward functions we compare the number of iterations required for each agent to reach the goal state.

We ran 100 agents of each type over a Infinite Mario version of Super Mario Bros Level 1-1. (Figure 2) and compare the iteration where each first reached the goal state in Figure 4. As anticipated, both the true engine and learned engine out-performed the basic RL agent. Notably, in comparing the true and learned engines in terms of their impact on their respective agents, we were unable to reject the null hypothesis that the values came from the same distribution (Wilcoxon paired test $p = 0.5783$). This represents evidence that the true and learned engines represent similar enough gameplay knowledge that they are indistinguishable, at least for this task. However, given that the learned engine is only trained on Level 1-1 of Super Mario Bros. we would not anticipate the same similarity in performance in a level that drew on mechanics not present in Level 1-1.

5 Future Work and Conclusions

Our technique has drawbacks, notably we do not represent player death or level transitions, which makes these key types of mechanics impossible to learn. We hope to address these shortcomings in future work. In addition, we are interested in possible applications of this approach to procedural content generation and full game generation.

In this paper we present a novel technique to derive a game engine from input gameplay video. We ran two evaluations, which provide strong evidence that the learned engine represents an accurate, general model of the video game engine responsible for producing the input gameplay video. Our technique relies on a relatively simple search algorithm that searches through possible sets of rules that can best predict a set of frame transitions. To our knowledge this represents the first technique to learn a game engine, a set of rules capable of simulating a game world. This represents a major step forward in the field of automatic game understanding.

Acknowledgments

We gratefully acknowledge the NSF for supporting this research under NSF award 1525967. Special thanks to Brent Harrison for advice and draft comments. This paper contains images generated by the Infinite Mario Bros. game.

References

[Braylan and Miikkulainen, 2016] Alexander Eric Braylan and Risto Miikkulainen. Object-model transfer in the general video game domain. In *The 12th Artificial Intelligence and Interactive Digital Entertainment Conference*, 2016.

[Brooks, 1999] Rodney Allen Brooks. *Cambrian intelligence: The early history of the new AI*, volume 44. Mit Press Cambridge, MA, 1999.

[Canossa and Smith, 2015] Alessandro Canossa and Gillian Smith. Towards a procedural evaluation technique: Metrics for level design. *The 10th International Conference on the Foundations of Digital Games*, page 8, 2015.

[Churchland and Sejnowski, 2016] Patricia S Churchland and Terrence J Sejnowski. *The computational brain*. MIT press, 2016.

[Cook *et al.*, 2013] Michael Cook, Simon Colton, Azalea Raad, and Jeremy Gow. Mechanic miner: Reflection-driven game mechanic discovery and level design. In *European Conference on the Applications of Evolutionary Computation*, pages 284–293. Springer, 2013.

[Ersen and Sariel, 2015] Mustafa Ersen and Sanem Sariel. Learning behaviors of and interactions among objects through spatio–temporal reasoning. *IEEE Transactions on*

Computational Intelligence and AI in Games, 7(1):75–87, 2015.

[Gettys and Fisher, 1979] Charles F Gettys and Stanley D Fisher. Hypothesis plausibility and hypothesis generation. *Organizational behavior and human performance*, 24(1):93–110, 1979.

[Gow and Corneli, 2015] Jeremy Gow and Joseph Corneli. Towards generating novel games using conceptual blending. In *11th Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.

[Guzdial *et al.*, 2016] Matthew Guzdial, Nathan Sturtevant, and Boyang Li. Deep static and dynamic level analysis: A study on infinite mario. In *3rd Experimental AI in Games Workshop*, 2016.

[Hafner, 2016] Danijar Hafner. Deep reinforcement learning from raw pixels in doom. *arXiv preprint arXiv:1610.02164*, 2016.

[Kober *et al.*, 2013] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[Martens *et al.*, 2016] Chris Martens, Adam Summerville, Michael Mateas, Joseph Osborn, Sarah Harmon, Noah Wardrip-Fruin, and Arnav Jhala. Proceduralist readings, procedurally. In *12th Artificial Intelligence and Interactive Digital Entertainment Conference*, 2016.

[Mnih *et al.*, 2013] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and Martin Riedmiller. Playing Atari with deep reinforcement learning. Technical report, Deepmind Technologies, 2013. arXiv:1312.5602.

[Nelson *et al.*, 2016] Mark J Nelson, Julian Togelius, Cameron B Browne, and Michael Cook. Rules and mechanics. In *Procedural Content Generation in Games (Computational Synthesis and Creative Systems)*, pages 99–121. Springer, 2016.

[Ng *et al.*, 2006] Andrew Ng, Adam Coates, Mark Diel, Varun Ganapathi, Jamie Schulte, Ben Tse, Eric Berger, and Eric Liang. Autonomous inverted helicopter flight via reinforcement learning. *Experimental Robotics IX*, pages 363–372, 2006.

[Oh *et al.*, 2016] Junhyuk Oh, Valliappa Chockalingam, Satinder Singh, and Honglak Lee. Control of memory, active perception, and action in minecraft. *arXiv preprint arXiv:1605.09128*, 2016.

[Pulli *et al.*, 2012] Kari Pulli, Anatoly Baksheev, Kirill Kornyakov, and Victor Eruhimov. Real-time computer vision with OpenCV. *Commun. ACM*, 55(6):61–69, June 2012.

[Ranzato *et al.*, 2014] MarcAurelio Ranzato, Arthur Szlam, Joan Bruna, Michael Mathieu, Ronan Collobert, and Sumit Chopra. Video (language) modeling: a baseline for generative models of natural videos. *arXiv preprint arXiv:1412.6604*, 2014.

[Selvaraju *et al.*, 2016] Ramprasaath R Selvaraju, Abhishek Das, Ramakrishna Vedantam, Michael Cogswell, Devi Parikh, and Dhruv Batra. Grad-cam: Why did you say that? visual explanations from deep networks via gradient-based localization. *arXiv preprint arXiv:1610.02391*, 2016.

[Singh *et al.*, 2002] Push Singh, Thomas Lin, Erik T Mueller, Grace Lim, Travell Perkins, and Wan Li Zhu. Open mind common sense: Knowledge acquisition from the general public. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 1223–1237. Springer, 2002.

[Summerville *et al.*, 2016] Adam Summerville, Matthew Guzdial, Michael Mateas, and Mark Riedl. Learning player tailored content from observation: Platformer level generation from video traces using lstms. In *The 12th Artificial Intelligence and Interactive Digital Entertainment Conference*, 2016.

[Summerville *et al.*, 2017a] Adam Summerville, Morteza Behrooz, Michael Mateas, and Arnav Jhala. What does that ?-block do? learning latent causal affordances from mario play traces. In *1st AAAI Workshop on what's next for AI in games*, 2017.

[Summerville *et al.*, 2017b] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. Procedural content generation via machine learning (pcgml). *arXiv preprint arXiv:1702.00539*, 2017.

[Togelius *et al.*, 2010] Julian Togelius, Sergey Karakovskiy, and Robin Baumgarten. The 2009 mario ai competition. In *IEEE Congress on Evolutionary Computation*, pages 1–8, 2010.

[Togelius, 2011] Julian Togelius. A procedural critique of deontological reasoning. In *Proceedings of DiGRA*, 2011.

[Uriarte and Ontanón, 2015] Alberto Uriarte and Santiago Ontanón. Automatic learning of combat models for rts games. In *11th Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.

[Vukotić *et al.*, 2017] Vedran Vukotić, Silvia-Laura Pintea, Christian Raymond, Guillaume Gravier, and Jan Van Gemert. One-step time-dependent future video frame prediction with a convolutional encoder-decoder neural network. *arXiv preprint arXiv:1702.04125*, 2017.

[Zook and Riedl, 2014] Alexander Zook and Mark O Riedl. Automatic game design via mechanic generation. In *AAAI*, pages 530–537, 2014.

[Zook *et al.*, 2015] Alexander Zook, Brent Harrison, and Mark O Riedl. Monte-carlo tree search for simulation-based strategy analysis. In *Proceedings of the 10th Conference on the Foundations of Digital Games*, 2015.