Multi-Modal Reinforcement Learning with Videogame Audio to Learn Sonic Features

by

Faraaz Nadeem

B.S. Computer Science and Engineering
Massachusetts Institute of Technology, 2019

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2020

Author . . . .
Department of Electrical Engineering and Computer Science
August 18, 2020

Certified by . . . .
Eran Egozy
Professor of the Practice in Music Technology
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Multi-Modal Reinforcement Learning with Videogame Audio to Learn Sonic Features

by

Faraaz Nadeem

Submitted to the Department of Electrical Engineering and Computer Science
on August 18, 2020, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Most videogame reinforcement learning (RL) research only deals with the video component of games, even though humans typically play games while experiencing both audio and video. Additionally, most machine learning audio research deals with music or speech data, rather than environmental sound. We aim to bridge both of these gaps by learning from in-game audio in addition to video, and providing an accessible introduction to videogame audio related topics, in the hopes of further motivating such multi-modal videogame research.

We present three main contributions. First, we provide an overview of sound design in video games, supplemented with introductions to diegesis theory and Western classical music theory. Second, we provide methods for extracting, processing, visualizing, and hearing gameplay audio alongside video, building off of Open AI's Gym Retro framework. Third, we train RL agents to play on different levels of *Sonic The Hedgehog* for the SEGA Genesis, to understand 1) what kinds of audio features are useful when playing videogames, 2) how learned audio features transfer to unseen levels, and 3) if/how audio+video agents outperform video-only agents.

We show that in general, agents provided with both audio and video outperform agents with access to only video. Specifically, an agent with the current frame of video and past 1 second of audio outperforms an agent with access to the current and previous frames of video, no audio, and 55% larger model size, by 6.6% on a joint training task, and 20.4% on a zero-shot transfer task. We conclude that game audio informs useful decision making, and that audio features are more easily transferable to unseen test levels than video features.

Thesis Supervisor: Eran Egozy
Title: Professor of the Practice in Music Technology

Acknowledgments

First and foremost, I would like to thank my parents and my sister Sanya for giving me so much love and support throughout my life. I owe so many of my accomplishments to my parents, and they continue to care for me no matter how many times I forget to do my chores around the house. Sanya is the best sister in the whole wide world <3 (wow sir).

I would like to give a huge thank you to my thesis supervisor and friend Eran Egozy, for being such a helpful, fun, and kind mentor over the past 2 years. Your experience with music and audio (and life in general) has been invaluable to this individual with limited formal training. I will miss our weekly hangouts!

Thank you Tom7 for your hilarious and educational YouTube series on Learnfun & Playfun [75]. It has been my dream to work on a videogame playing AI ever since I watched those videos in high school several years ago.

Shout out to Mirchi, the MIT dance community, and to all of the amazing friends I have made during my time here. You are the most inspiring people I have ever met, and I am so lucky to have had the opportunity to interact with you all on a daily basis for the past 5 years. Looking forward to see where the next chapter takes us!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The next decade promises to show a rise in automation stemming from current research being done in machine learning (ML). Existing contributions to ML research allow us to train models that can 1. learn low dimensional representations of high dimensional data, such as video, text, and audio, and 2. learn how to make classifications or decisions based on these representations [47].

Some studies forecast that 8 million vehicles will ship with SAE Level 3, 4, and 5 (level 5 being fully autonomous) self driving cars by 2025, due to our ability to extract and learn from features in LIDAR and camera data [10]. Chat bots are leveraging chat histories to be able to answer large numbers of simple requests in the customer service industry, and images of cancerous cells are being used to train models for cancer diagnosis in the medical industry.

Although classification and decision making are well studied areas of machine learning, most research and practical application has involved the use of text or video based data, rather than audio. The audio research that exists can largely be subdivided into use of music or speech. Music data is commonly used for genre classification and recommendation tasks, and speech data is most often used for captioning, as a means of transforming the medium into text data. Only a small portion of research has approached other forms of audio data, such as environmental or feedback-related sounds [62].

This gap in research needs to be addressed since environmental audio is critical to achieving human performance on a number of tasks, especially when visual or textual clues are not sufficient. Self driving cars cannot achieve full automation without being able to react and respond to emergency vehicle sirens in the distance, or nearby car horns. The tone and pitch of a customer's voice can help identify the sentiment of a sentence that was spoken [29], and background noise can help identify their environment [39]. Doctors rely on audio cues from a heartbeat monitor while navigating the visually dense and procedurally complicated task of performing surgery. Emergency responders listen for calls for help or natural dangers such as burning, to locate the emergency when arriving at a scene.

Environmental audio cues can also serve to reinforce existing visual ideas. A car failing to start can be corroborated by the sound of a sputtering engine, and a sizzle on the pan verifies that food is cooking. The clicking feedback produced by a mouse, keyboard, or button, confirms that the hardware or software has received the user input. Realistic environmental sound design helps to fully submerge an individual into the world of a movie or TV show [34].

In this work, we use video games as an environment to understand how reinforcement learning (RL) models can incorporate both audio and video observations, to make decisions with immediate consequence and long term goal.

We present three main contributions. First, we provide an overview of sound design in video games, accessible to a non-musician audience, supplemented with introductions to diegesis theory and Western classical music theory. Second, we provide methods for extracting, processing, visualizing, and hearing gameplay audio alongside video, building off of Open AI's Gym Retro framework. Third, we train RL models to play on different levels of Sonic The Hedgehog for the SEGA Genesis to understand 1) what kinds of audio features are useful when playing videogames, 2) how learned audio features transfer to unseen levels, and 3) if/how audio+video agents outperform video-only agents.

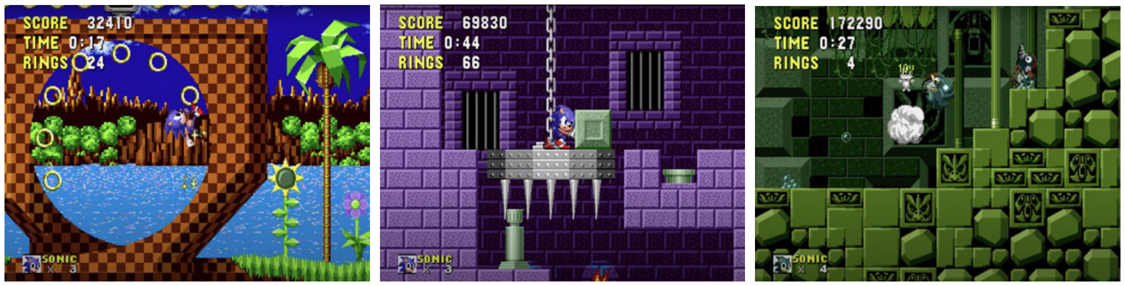


Figure 1-1: Screenshots from *Sonic The Hedgehog*. From left to right: 1) In Green Hill Zone, Sonic uses his momentum to run through a vertical loop, while collecting rings. 2) In Marble Zone, Sonic pushes a block onto a switch that will lift the spiked chain blocking his path. 3) In Labyrinth Zone, Sonic defeats a Badnik during an underwater section of the level.

1.1 Sonic Video Game Franchise

In this section, we go over some of the history behind the Sonic videogames, and provide an overview of gameplay. Both are important for understanding how the Sonic games on the SEGA Genesis compare to other videogames being used in RL research, and what challenges we can expect our model to learn from.

SEGA, the Japanese multinational video game developing and publishing company, commissioned a team in 1990 to create a mascot that could be the face of a franchise rivaling Nintendo's iconic characters like Mario and Link [15]. In 1991, this goal was achieved through the realization of Sonic, the main character of *Sonic The Hedgehog* (Sonic 1) for the SEGA Genesis. The game features the adventures of a blue anthropomorphic hedgehog in his quest to defeat the evil Dr. Robotnik, and retrieve the stolen chaos emeralds (figure 1-1). Sequels to this game, *Sonic The Hedgehog 2* (Sonic 2) and *Sonic 3 & Knuckles* (Sonic 3&K), were released in 1992 and 1994 respectively. The commercial success of the games cemented Sonic as SEGA's core videogame franchise.

1.1.1 Core Innovations

The Sonic games boast a fast-paced style of play designed to leverage the Genesis' processing speed, which was the console's main advantage over its competitors, such as the SNES. Programmer Yuji Naka capitalized on the speed running habits of NES players, who would memorize and speed through the starting levels of Mario and other games because save files were uncommon at the time [38]. Sonic level designs feature multiple paths to completing each level, with higher elevation trajectories typically rewarding players with more opportunities for high speed platforming, in exchange for increased difficulty (figure 1-2).

The Genesis was the first major 16-bit home console to reach the market, which is why Sonic was one of the most visually detailed, bright, and colorful games when it was released. Each level visually has foreground, middle ground, and background layers, and Zones are themed with vegetation, buildings, machinery, mountains, bodies of water, caves, sand dunes, and other natural or man-made landscapes.

The SEGA Genesis also features a sound system that was advanced and unique for its time. The NES (1983) for example, ran on the Ricoh RP2A03 audio chip which only offered 5 audio channels (2 pulse waves, 1 triangle wave, 1 noise, 1 sample playback) [16]. Sound design for the NES was limited to manipulating pitch or frequency of these channels. The SNES (1990) had a more developed sound system with 8 channels, all centered around sample playback. Due to audio RAM constraints, large waveforms on the SNES were often put together in creative ways by looping single cycle waveforms [20]. The SEGA Genesis (1988) sound system, on the other hand, included the Yamaha YM2612 synthesizer and Texas Instruments SN76489 audio chip for a total of 10 audio channels (6 operator fm synthesis or sample player, 3 square wave, 1 noise) [19]. Having more channels along with the ability to synthesize audio allowed for a kind of hybrid system, which was more capable of translating complex musical compositions than the NES, and did so more naturally than the SNES.

Masato Nakamura of the JPop band "Dreams Come True" was commissioned to compose the music for Sonic 1 and 2. He took the unorthodox approach of treating the game as a film, and was challenged to create themes that matched the feel of gameplay, while only having access to early development sketches and visuals [37].

Nakamura and co. used this advanced sound system to create a number of game

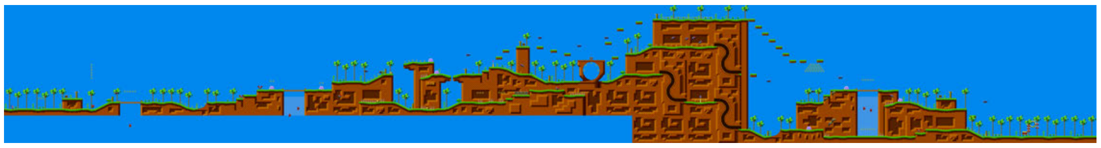


Figure 1-2: The full map of Green Hill Zone Act 1. Ability to navigate towards upper elevation in the level is rewarded with fewer obstacles and faster paced platforming.

scores that remain iconic to this day. The Sonic Zone themes in particular are critically acclaimed for their quality and diversity, and how they pushed the boundaries of videogame music.

1.1.2 Gameplay Overview

Sonic The Hedgehog is a 2D side scrolling platforming game. The main idea of the game is to navigate Sonic through vertical loops, over bottomless pits, off of springs, while collecting rings and defeating robot enemies. A central game mechanic is that Sonic can build great speed if his movement is not interrupted by stopping or bumping into obstacles, which can then be used to launch him into the air off of ramps, or plow through an array of enemies. The first game consists of 7 zones, and each zone has 3 acts. Each act can be considered to be a level. The last level of each zone ends with a boss fight against Dr. Robotnik.

Sonic encounters a number of different types of enemies called "Badniks", who are animals trapped in Dr. Robotnik's evil robotic skeletons. Some Badniks can fire at Sonic, some traverse a set path, and others can fly in the air. When Sonic jumps, he turns into a ball with his spikes protruding from the center. In this ball state, Sonic can defeat enemies that he touches.

There are also different hazards that Sonic must avoid, such as pits, spikes, lava, rotating spiked balls on chains, crushing platforms, and drowning underwater.

Rings are placed in groups of 3 or more along the levels. They give points, and act like a shield. If Sonic is hit by an enemy or hazard without any rings, he loses a life. But if Sonic has at least 1 ring, then hitting an enemy or hazard knocks Sonic backwards and he forfeits up to 20 of his rings, without losing a life. The rings are

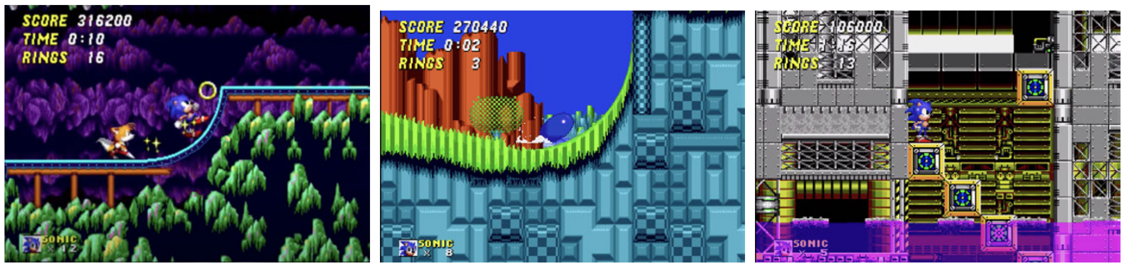


Figure 1-3: Screenshots from *Sonic The Hedgehog 2*. From left to right: 1) In Mystic Cave Zone, Sonic is followed by Tails up a small ramp. 2) In Hill Top Zone, Sonic revs up a Spin Dash in order to gain the momentum needed to climb the steep slope. 3) In Chemical Plant Zone, Sonic platforms up a narrow column to escape the rising toxic liquid. Note: Tails is missing from the last two screenshots because he was left behind, or lost a life. He returns by default after a short time.

available for a short period of time to recollect. If Sonic falls into a bottomless pit, drowns, gets crushed by a platform, or takes more than 10 minutes to complete a level, then he loses a life regardless of how many rings he has. For every 100 rings collected, Sonic gains an extra life.

Sonic starts the game with 3 lives, and upon losing all 3 lives, the player may "continue" and return to the beginning of the level. Progression is made easier with checkpoints. Upon passing a checkpoint, which is visually indicated by a lamp post whose spherical top spins while being passed, Sonic is able to return to this point rather than the beginning of the level when he loses lives.

Many levels contain sections which are underwater. Sonic can survive approximately 30 seconds underwater until he needs to find an air bubble or jump out of the water to avoid drowning.

Video monitors are placed throughout the levels, which provide Sonic with powerups when destroyed. Sonic can receive an extra 10 rings, an extra life, a shield, a temporary speed boost, or temporary invincibility from these monitors.

Sonic can also collect chaos emeralds in hidden special stages. Obtaining all 7 chaos emeralds allows Sonic to turn into Super Sonic, which is a mode that consumes rings in exchange for invincibility and super speed.

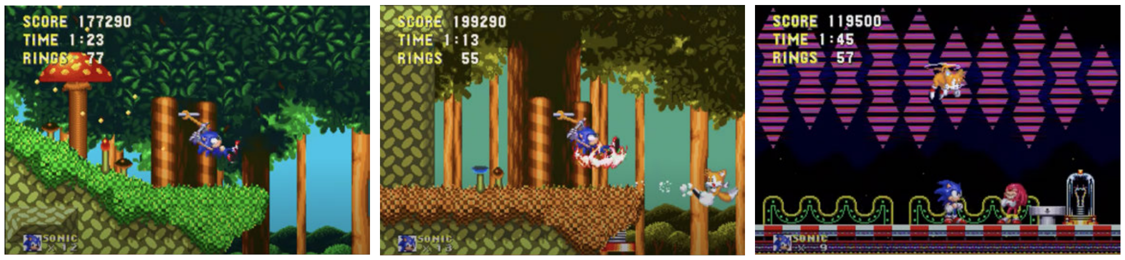


Figure 1-4: Screenshots from *Sonic 3 & Knuckles*. From left to right: 1) In Mushroom Hill Zone Act 1, Sonic rotates around a device that will launch him into the air, if his release is timed correctly. 2) In Mushroom Hill Zone Act 2, a nearly identical section as the previous screenshot is textured differently to reflect fall colors. 3) In Carnival Night Zone, Sonic and Tails encounter Knuckles in a cutscene.

1.1.3 Changes in Sonic 2 and Sonic 3&K

There are a number of key add-ons to Sonic 1 in Sonic 2 (figure 1-3) and Sonic 3&K (figure 1-4). These add-ons provide additional challenges for our models, and more tools to complete them.

Sonic 2 introduces Tails, a shy orange fox, as a sidekick to Sonic. In one player mode, Tails follows along as Sonic progresses through each level, and autonomously interacts with his nearby environment. Each level is on average longer than Sonic 1, and there is more visual variety between levels. The Spin Dash Attack is introduced as a new move, where Sonic can charge up a spin from a stationary position before releasing and dashing off with a full speed rolling attack.

Sonic 3&K retains the additions from Sonic 2, with a few more changes. Mid-level cutscenes add a cinematic element to the game, and feature a new character called Knuckles, a red dreadlocked animal with long quills. Levels are designed to be even longer and more complicated. Levels within zones now have slight differences in visual textures and music, according to the plot progression. For example, a cutscene after Angel Island Zone Act 1 shows Dr. Robotnik burning down the island; the following visuals in Act 2 are ashy, with embers strewn about, and the backing musical theme has rougher instrumentation than Act 1.

Chapter 2

Background

In this section we will cover some background on reinforcement learning (RL), frameworks for interacting with videogames in an RL setup, and related works.

2.1 Reinforcement Learning

Reinforcement learning [73] [26] is an increasingly popular field of machine learning research. It is a semi supervised approach for learning complex sequential decision making with potentially sparsely labeled data or delayed reward.

Many see high potential in RL research to be able to solve complex real life tasks, which can be modeled as sequential interactions with an environment. Completely supervised methods commonly suffer from issues where large amounts of labeled data must be provided in order for the model to be able to generalize.

For example, we can consider the task of playing the game Go, which is a board game with simple rules but complex strategy. In a supervised setting, we would need to curate a dataset of millions of board positions, and label which move to take at each position. Curating such a dataset with optimal moves is impossible since the full move tree of Go has not yet been enumerated; in fact, Go has a higher branching factor than Chess, which has not been fully enumerated either [36]. We might resort to using a database of human Go games, where the labels for board positions are the moves taken by the winner of the game. Even if we were able to accumulate a

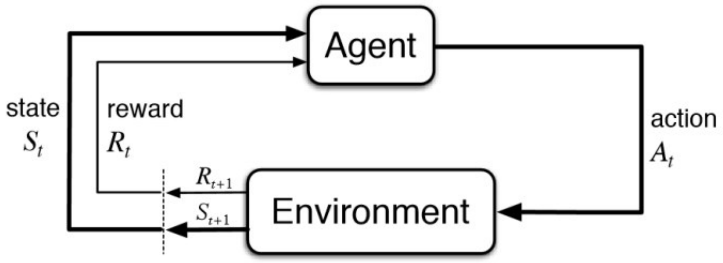


Figure 2-1: Environment loop of an RL setup.

sufficient number of board states in this fashion, we would still be training a model that at best achieves human level performance, rather than super-human.

In the RL setting, we instead learn from interactions between an agent and environment (figure 2-1). Interactions are generated as follows. At a given time step, the agent observes the environment state and decides to take an action on the environment. The environment responds by updating its state according to its state-action transition rules, and provides a reward to the agent. The agent's goal is to maximize the total reward over one sequence of decisions, called an episode.

The Go board consists of 18 horizontal and vertical lines, and players can place their pieces at each intersection. Therefore, the environment state can be represented by 324 values indicating whose piece, if any, is on the corresponding board intersection. The agent takes actions by placing one of their pieces on an empty intersection, and the board transitions to a new state according to Go's piece capturing rules. A reward of 1 is given to the player who controls more territory at the end of the game, when all pieces have been placed. Although this is a simple problem setup without any reliance on heuristics, RL agents have been trained to consistently beat world champion caliber Go players [69] [68] [71].

2.2 RL for Videogames

Board games remain popular settings for RL research, because the agent and environment are readily defined, and it is possible to learn increasingly complex policies from simple rules or transitions. It becomes much harder, however, to use RL for physical or environmentally more complicated tasks. Real world setups may suffer from mechanical issues, generate agent-environment interactions very slowly, and rely on sensors or motors that are too expensive or unreliable for defining clear actions and rewards.

Videogames provide a natural bridge between simple games and complicated real world setups. Many videogames are already created with the goal of emulating real world environments or decision making, so we expect that performance in these virtual environments would transfer to physical ones too. Indeed, the videogame-like environment called Roboschool [48] has an option to train on a simulation of the physical Atlas robot [11].

At the same time, videogames necessarily have well defined agents and environments because they are programmed to run on computers. This comes with the added benefit of potentially being able to run identical environments in parallel, generating interactions faster than real time, and having fine grained control over all aspects of the training setup.

We will now proceed with discussing popular frameworks for videogame RL research, along with some significant results.

2.2.1 Arcade Learning Environment

The Arcade Learning Environment (ALE) [6] is an interface to hundreds of Atari 2600 videogame environments that was introduced as a challenge problem and platform to facilitate RL research on videogames. It is a rigorous testbed for research on model learning, transfer learning, intrinsic motivation, and a number of other related topics.

The Deep Q Network (DQN) [45] model baseline in 2015 is considered to be one of the first major baselines for the ALE. The DQN uses a CNN architecture to process the videogame screen into a hidden state representation, and then uses Q Learning as the policy gradient algorithm to optimize for reward.

In 2020, Agent57 [3] became the first model to achieve super-human performance on all 57 Atari 2600 benchmark games. It builds off of existing improvements to the original DQN model, along with advancements in short-term memory [28], episodic memory [4], exploration [63], and meta-controlling [81]. Agent57's performance on the ALE benchmark is impressive, particularly because different games have completely different environments or objectives.

The ALE is a useful benchmark, but it is limited to a preset number of game environments, and does not support game audio. Indeed, the benchmark consists of a total of 57 games, and lack of sound generation is labeled as a feature for enabling fast emulation with minimal library dependencies. Even if audio features were accessible, we do not expect to learn much from the simplistic Atari 2600 sounds. The Atari 2600 only provided 2 channels of 1-bit monaural sound with 4-bit volume control [40], as compared to the 10 channels of 16 bit audio on the SEGA Genesis.

2.2.2 Open AI Gym Retro

Open AI Gym Retro [18] [49] is a more recently developed platform for reinforcement learning research on video games. It is built on the Libretro API [59], which gives it natural cross platform and emulator compatibility, and access to features such as game audio. Currently, there are thousands of games available to use with Gym Retro. Anyone can use the integration tool, which helps to locate memory addresses of game state variables such as score, to incorporate new ROMs into the library.

Gotta Learn Fast [47] is a transfer learning benchmark with Gym Retro on the Sonic games for the SEGA Genesis, and is one of the primary inspirations for this work. It shows that pretraining on 47 train levels before fine tuning on 11 test levels achieves the best test result when compared to several other models. They do not perform any audio-related experiments, although the Sonic game audio is accessible through the Gym Retro framework.

2.3 Policy Gradient Algorithm

In this section, we build towards and define the policy gradient algorithm used to train our RL models.

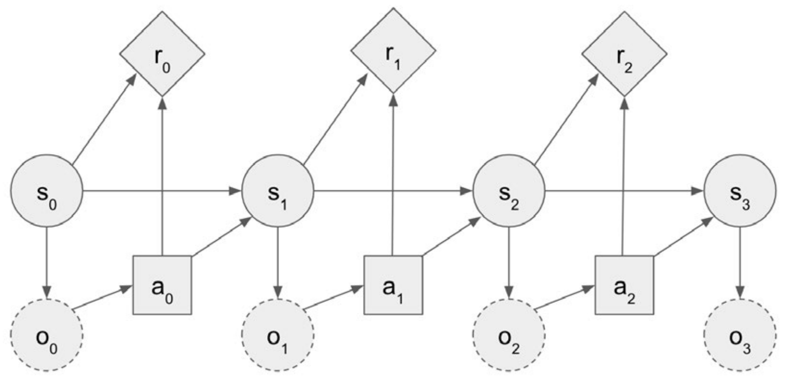


Figure 2-2: A graph showing the dependencies of a Partially Observable Markov Decision Process (POMDP) unraveled for the first 3 time steps. Arrows indicate flow from dependencies to values sampled from distributions conditioned on these dependencies.

2.3.1 POMDPs

Go is a fully observable and deterministic game, which means that the board is a true representation of the game state, and there is a single transition for each action-state pair. On the other hand, Sonic and many other videogames are only partially observable and stochastic. In other words, video and audio are not enough to understand the true underlying game state, and performing an action in a given game state does not lead to a guaranteed outcome because of randomness in the system.

To account for this, we define our agent-environment interactions more concretely as partially-observable Markov decision processes (POMDPs) [2] [25]. Under this formulation, we maintain our core spaces of actions A, states S, and rewards R, and additionally define an environment observation space O, where observations are surface-level representations of a true state. We also define our transition and observation functions $\mathcal{T}, \mathcal{O}$ to now follow stochastic distributions, and leave our reward function $\mathcal{R}$ as a deterministic mapping. A visual representation of a POMDP is given by figure 2-2.

In the Sonic games, we can imagine a scenario where Sonic is being approached by a Badnik. The true underlying game state tells us the magnitude and direction of the Badnik's velocity. An observation of this state might be a single frame of video which tells us the Badnik is close to Sonic, but not its relative velocity. When the agent takes an action to control Sonic, performing a spin attack may lead to a transition to a new state where the Badnik is defeated, depending on if the Badnik decided to move closer to Sonic. This would result in a new visual observation of the defeated Badnik, and a reward of 100 points.

More concretely, we can say our agent operates under a policy π with parameters θ, that produces actions given an observation, $\pi_\theta : Dist(O) \to A$. At time t, the agent takes an action $a \in A$ while in state $s \in S$, transitions to state $s' \sim \mathcal{T}(s, a, s')$, and receives observation $o \sim \mathcal{O}(o, s')$ and reward $r = \mathcal{R}(s, a) \in R$.

2.3.2 Objective Function

We wish to maximize the expected return of our policy π with parameters θ such that

$$\max_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{H} \gamma^t r_t\right]$$

for episode step t, horizon H, and discount factor γ. In other words, we want to find a policy that maximizes the expected reward over an episode. In practice, we set $\gamma < 1$ to encourage faster reward accumulation.

We can recursively compute the value of taking an action a_t given the observation o_t, using the Bellman Equation [7]

$$Q(o_t, a_t) = \sum_s \mathcal{O}(o_t|s) \left(\mathcal{R}(s, a_t) + \gamma \sum_{\substack{o_{t+1}\\ a_{t+1}\\ s_{t+1}}} \mathcal{O}(o_{t+1}|s)\, \mathcal{T}(s_{t+1}|s, a_t)\, Q(o_{t+1}, a_{t+1}) \right)$$

The optimal policy π^* is therefore

$$\pi^*(o) = \max_a Q(o, a)$$

Unfortunately, defining such a policy is impossible in practice. In many videogames, it is computationally infeasible to enumerate all possible spaces, let alone distributions $\mathcal{T}, \mathcal{O}, \mathcal{R}$, or table of observation-action values $Q(o, a)$, which itself has exponential branching complexity. This motivates us to use a model-free (i.e. without knowledge of $\mathcal{T}, \mathcal{O}, \mathcal{R}$) algorithm, which we also expect to generalize better.

We can train a neural network based policy with parameters θ to learn a value or reward estimate for observation-action pairs, $Q^{\pi_\theta}(o, a)$. Training data is gathered by generating a set of agent experiences with the environment, and sampled randomly with a replay buffer. To normalize for observations with large magnitude values independent of action, we use the advantage function

$$A^{\pi_\theta}(o, a) = Q^{\pi_\theta}(o, a) - V^{\pi_\theta}(o)$$

where $V^{\pi_\theta}(o)$ is a learned estimate for observation value. Overall, this gives us a new objective

$$\max_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{H} \gamma^t \, \log \pi_\theta(a_t | o_t) \, A^{\pi_\theta}(o_t, a_t) \right]$$

2.3.3 Policy Gradient

We can optimize [80] the expected reward by taking stochastic gradient descent on policy parameters θ, giving us the policy gradient

$$g = \Delta_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{H} \gamma^t \, \Delta_\theta \log \pi_\theta(a_t | o_t) \, A^{\pi_\theta}(o_t, a_t) \right]$$

There are a few limitations of policy gradients that are defined in this way. First, sampling efficiency is poor because data that is generated under an old policy is not representative of a new policy, and therefore suffers from bias if used for future training. In order to train with no bias, we would need to throw out our entire batch of generated data and generate a new set after each gradient update.

Second, distance in parameter space is not the same as distance in policy space. While optimizing our parameters, it is hard to tune step size in a way that prevents potentially destructive changes in policy, while also taking an acceptable time to train.

2.3.4 Proximal Policy Optimization

Proximal Policy Optimization (PPO) [65] is a policy gradient algorithm that achieves state of the art results on a number of RL benchmarks. It builds on former advancements in policy gradient research [32] [44] [70], while also aiming to be fast and easy to implement.

At a high level, PPO approximately enforces the KL constraint, also known as the trust region [64]. The trust region is the area within which we can perform a policy update, such that the new policy π_θ is not too far removed from the old policy π_{θ_k}. The clipped objective variant of PPO also allows us to de-bias data generated under an old policy.

Under the clipped objective, we compute this difference in old and new policy with the importance sampling (IS) weight $w_t(\theta) = \pi_\theta(a_t|o_t)/\pi_{\theta_k}(a_t|o_t)$. We can use this IS weight to de-bias our advantage. This allows us to perform multiple gradient updates with a single batch of data, without being biased to the old policy under which it was generated, and therefore increase our data efficiency. However, this de-biasing step is unstable, and increases our chances of updating to a new policy outside the trust region. Both of these issues are mitigated by taking the lower bound of this IS re-weighted advantage with a clipped term $\texttt{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)$ re-weighted advantage.

Our objective function now becomes

$$\mathcal{L}_{\theta_k}^{CLIP}(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\Big[\min(w_t(\theta)\,\hat{A}_t^{\pi_k}, \texttt{clip}(w_t(\theta), 1-\epsilon, 1+\epsilon)\,\hat{A}_t^{\pi_k})\Big]\right]$$

where the IS weight removes bias from data generated under the old policy, and the clipping term helps to both stabilize the re-weighting and keep the new policy within the trust region. In other words, both our problems of data efficiency and disconnect between parameter and policy space are mitigated with this PPO approach. See algorithm 1 for a structured outline of the algorithm.

Algorithm 1: PPO with Clipped Objective

```
Input: initial policy parameters $\theta_0$, clipping threshold $\epsilon$, ppo epochs $n$
for $k = 0, 1, 2, ...$ do
    Use policy $\pi_k = \pi(\theta_k)$ to collect a set of partial trajectories $\mathcal{D}_k$
    Estimate advantages $\hat{A}_t^{\pi_k}$ with any advantage estimation method
    Compute policy update $\theta_{k+1} = \arg\max_\theta \mathcal{L}_{\theta_k}^{CLIP}(\theta)$
        with K iterations of minibatch SGD
    for $0, 1, 2, ..., n-1$ do
        Compute policy update $\theta_{k+1} = \arg\max_\theta \mathcal{L}_{\theta_{k+1}}^{CLIP}(\theta)$
            with K iterations of minibatch SGD
    end
end
```

2.4 Related Work

What follows is a compilation of related work on videogame reinforcement learning [67], multi-modal deep learning, and transfer methods.

Kim et al. [31] modify the ALE to support audio queries, and demonstrate that an attention mechanism can identify useful latent audio/video features that increase performance on H.E.R.O and Amidar on the Atari 2600, and transfer knowledge to accelerate learning in a door puzzle game. This is an encouraging result, and it is the only work on incorporating audio into videogame RL that we are aware of. It is important to note however that the environment, controls, and soundscape of these three games are quite simplistic as compared to the Sonic games on the SEGA Genesis. All of the sounds are short and sparsely played, and the environments are lower resolution, more deterministic, and less interactive. Amidar and the door puzzle in particular have single screen levels, and contain sounds only characterized by a few short bleeps.

Separate works on multi modal deep learning show that learning from additional modalities improves performance. Kaplan et al. [27] show that the additional modality of natural language suggestions can be used to improve performance on the Atari 2600 games. Ngiam et al. [46] present a series of tasks for multimodal learning and demonstrate cross modality feature learning, where better features for one modality (e.g., video) can be learned if multiple modalities (e.g., audio and video) are present at feature learning time. Poria et al. [57] show that a multimodal system fusing audio, visual, and textual clues outperforms previous state of the art systems by 20% on a YouTube sentiment dataset.

A number of works show positive transfer learning results with video observations in videogames. Parisotto et al. [54] show that pretraining on some Atari games leads to increased learning speed on others. Higgins et al. [22] propose a zero-shot transfer method that works better than base RL algorithms in Deepmind Lab [5] and MUJOCO [74]. Mittel et al. [43] show that competitive RL can be used to transfer between Pong and Breakout. Rusu et al. [60] propose a progressive neural network architecture that transfers between Atari games, and show that it improves training speed on previously unseen games.

OpenAI Five [9] used massive amounts of parallel training to extend PPO application to Dota 2, one of the most popular and complex esports games. It became the first AI system to defeat the world champions of an esports game.

Chapter 3

Videogame Audio Theory

In this section, we we analyze videogame sound design through the frameworks of diegesis theory and Western classical music theory.

3.1 Videogame Diegesis Theory

Sound design plays a key role in perceived realism and many central gameplay mechanics, and there can be several layers or types of sound in a game. To help facilitate discussion, we will review videogame audio diegesis theory and categorize sounds along different axes of interactivity.

3.1.1 IEZA Framework

The IEZA Framework [23] is a method of categorizing videogame sound, inspired by previous research in film audio theory. Its primary axis separates diegetic from non-diegetic sounds, i.e. sounds that emanate from sources in the game world, and those that do not. It further categorizes sounds on a secondary axis, where they either provide information about in game activity, or in game setting.

To get a better understanding of this, we will proceed with placing sounds from the Sonic games into each of the categories. These are our own subjective placements, so we give our reasoning for them as well.

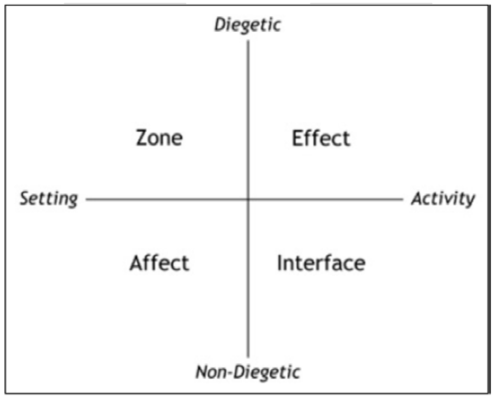


Figure 3-1: Visual representation of the IEZA Framework.

- **Effect:** Sounds that emanate from sources in the game (diegetic), and provide information about in game activity. They typically provide immediate feedback on player actions, or events triggered in the environment, and mimic behavior of sound in the real world. Example: the sound of footsteps or gunshots in a first person shooter.

  The jump, spin attack, and screech halt sound effects are produced as a direct result of performing one of the moves in Sonic's moveset (figure A.1). The source of all three sounds can be clearly traced to being made by Sonic. Note that not all of these sounds, in particular the jump sound, are entirely realistic. But for simplicity, we count all of them in this category.

  Collecting rings, losing rings, breaking rocks, collapsing bridges, passing a checkpoint, defeating enemies, losing a life, and losing a life by drowning, are all interactions between Sonic and his environment. The source for the corresponding sounds are either Sonic or the environment object, and the sounds themselves provide feedback on the specific environment interaction.

  In Sonic 2 and Sonic 3&K, Tails follows Sonic around, and can interact with the environment in all of the same ways to produce the same sounds. Although Tails cannot be directly controlled, his movements are still influenced by Sonic, who is controllable.

- **Zone:** Sounds that emanate from sources in the game (diegetic), and provide information about in game setting. The main difference between Effect and Zone is that this category consists of one layer of sound, rather than separate sound sources. This layer is often ambient, environmental, or background sound, and provides little information about in game activity. Examples: wind and rain in an open world game, or crowd noise in a sports game.

The environmental sounds of Knuckles and Dr. Robotnik pressing buttons, destroying bridges, etc. during cutscenes in Sonic 3&K fall into this category. They emanate directly from sources in the cutscene environment, and provide no information on player actions since the player cannot control the cutscenes.

- **Interface:** Sounds that do not emanate from sources in the game (non-diegetic), and provide information about in game activity. These typically do not have an equivalent sound source in real life, and often provide feedback on meta game information. Examples: menu navigation sounds, and low health warning beeps.

  When Sonic is underwater, an ominous drowning theme begins to play when he is running out of air. As it picks up in speed, it informs that there is increasingly limited time left for the player to take action. It takes about 10 seconds for this theme to play out and for Sonic to actually drown. Note that there is a separate diegetic sound that plays when Sonic loses a life by drowning.

  Before Sonic gets to this drowning stage, there are also 4 pings that play in 5 second second intervals while Sonic is underwater, indicating that the drowning theme is approaching. Overall, it takes about 30 seconds for Sonic to drown after he has entered the water.

  The level completion jingle provides audio reinforcement indicating that the player took the correct steps to finish the level. It does not have a direct source in the game world, and provides feedback on player actions, just like the drowning theme and underwater pings.

- **Affect:** Sounds that do not emanate from sources in the game (non-diegetic), and provide information about in game setting. These can be a way to include cultural references or set emotional expectation. Examples: orchestral music in an adventure game, creepy music in a horror game, or the official World Cup theme in a soccer game.

The Sonic start screen, boss fight, and Zone themes provide musical background with no in game source, and provide no information about in game activity. Note the distinction between Zone themes and the IZEA Zone category. Zone themes play in the background of each Act, on a loop, and there are no in-game actions that can change their musical behavior (besides interrupting with the drowning, Super Sonic, or extra life theme).

Given this framework of categorization, we would expect sounds relaying information about in game activity to be the most important for completing levels in Sonic. Both Effect and Interface provide information about in game activity, and Effect directly relates sounds to sources on screen. Although the sounds typically do not provide novel information beyond the visual component, they are still useful for reinforcing feedback of in game actions.

Setting related sounds in Sonic mostly serve aesthetic purposes, but we assert that the Zone themes can still convey indirect action-related information, because the theme reflects the general Zone design. The Sandopolis Zone plays a mysterious, Middle East inspired tune, the Sky Chase and Wing Fortress Zones play light and airy music, and the Chemical Plant Zone plays an upbeat industrial vibe. The ominous and percussive Death Egg Zone theme mirrors the dark and spooky level design. Although we cannot trace music to a source in each level or direct actionable feedback, the Zone environment does reflect a connection to the type of music that is played.

Levels, especially within Zones, share similar design characteristics. Both Aquatic Ruin Acts require extensive traversal underwater, and all three Labyrinth Zone Acts require significant backtracking to the left in order to make net progress towards the right. Hearing similar Zone themes might also indicate similar techniques needed to complete Acts with those themes. Indeed, Green Hill Zone and Emerald Hill Zone share comparable musical instrumentation, and strikingly similar visual aesthetics and level design.

3.1.2 Sound as a Critical Gameplay Component

Here, we will briefly cover games where diegetic sounds provide gameplay information beyond the visual component.

Sounds in videogames can be critical to success as non visual cues for proximity. In *Minecraft*, a crackling hiss warns that a Creeper is about to explode next to you, and a bubbling noise heard while caving tells you there is lava behind a stone wall. In *Dead by Daylight*, an increasingly loud and fast heart rate as a survivor means the killer is stalking nearby, and the noise created by a failed mechanical repair exposes which generator a survivor is working on to escape the killer. In competitive, team-based tactical first person shooters like *Valorant*, the sound of footsteps might reveal the location of an attacking team around the corner, or a ploy devised by one attacking member to draw defenders away from the point that is meant to be attacked (figure 3-2). In *Fruit Ninja*, the number of fruit launching sounds informs how many fruit will appear on screen, and how long you should wait to ensure a bonus of slicing all of them with one stroke.

Diegetic sounds in the Sonic games are only supplementary to the visual information presented on screen, except for a few edge cases (figure 3-3). But we can imagine that small, easy to miss visuals in Sonic might benefit from sounds at a similar level as in the games above.

3.2 Brief Introduction to Western Music Theory

3.2.1 Notes and Intervals

In Western music, a note is a symbol denoting a musical sound, typically with a pitch/frequency and duration. A440, also known as the Stuttgart pitch, serves as the tuning standard and anchor point for calculating the frequency of other notes. It is

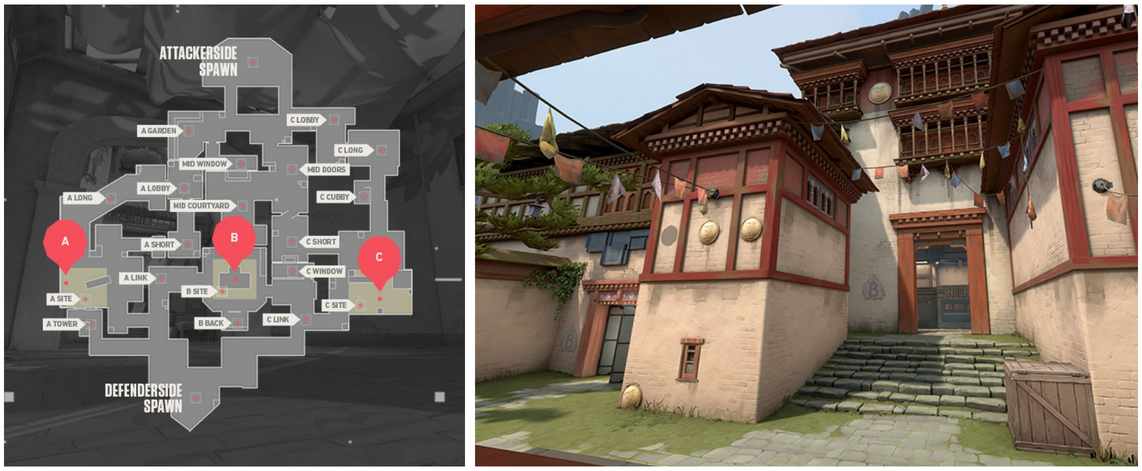


Figure 3-2: *Valorant* Haven map. Left: minimap showing the three capture sites, marked in red. The limited field of view created by the narrow and winding pathways, along with the ability to shoot through some walls, make sound critical for finding enemy players. As a counter strategy, if one attacker can draw defender attention via noise to a site (ex. A, far left), it will take a long time for the defenders to rotate and defend the intended attacker site (ex. C, far right). Right: In game example of a U-shaped path.

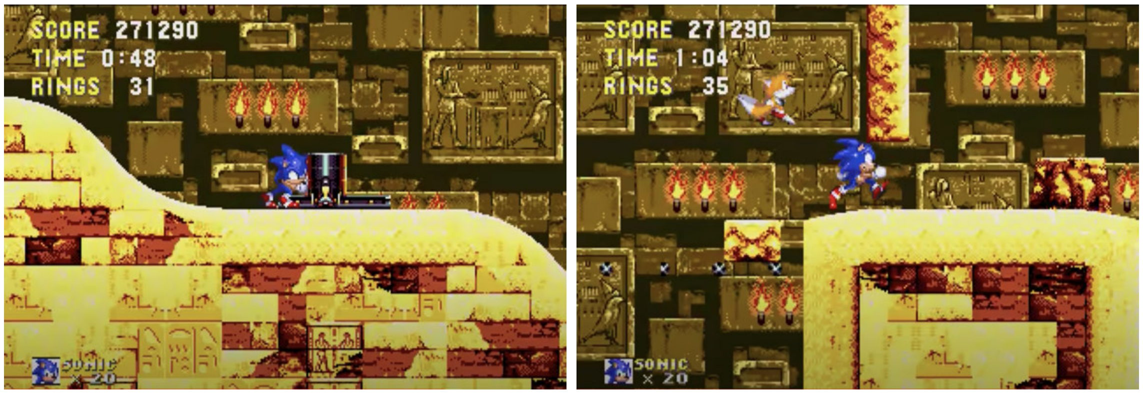


Figure 3-3: Door puzzle in Sandopolis Zone 2 with audio feedback. Left: Sonic pushes a black device to the right, which lifts a door. Over time, the device shifts back to the left. A steady clicking noise indicates that the door is slowly closing, even if both the device and door are off-screen. Right: Sonic makes it past the closing door, just in time. The same cannot be said for Tails.

the fifth A (called A4) on the piano, and has a fundamental frequency of 440 Hz [1].

The difference in pitch between two notes is called an interval, and an octave is the interval that specifies the difference between one musical pitch and another with double its frequency. Western music defines 12 notes (A, A#, B, C, C#, D, D#, E, F, F#, G, G#) along an octave, also called a chromatic scale. Therefore, the sixth A (called A5) is 880 Hz, and in general, frequencies of notes form a geometric sequence [58].

Since notes follow a geometric sequence, doubling in frequency every 12 notes, the frequency of each note can be calculated by successively multiplying by $\sqrt[12]{2}$. A4 is the 49th key on a piano, so the frequency of the nth key is given by $440 \cdot \left(\sqrt[12]{2}\right)^{n-49}$. Indeed, using this formula to calculate the frequency of A5 gives us $440 \cdot \left(\sqrt[12]{2}\right)^{12} = 440 \cdot 2 = 880$ Hz.

3.2.2 Interval Ratios

Western musicians and scientists have spent hundreds of years studying the harmonics of interval frequency ratios [13]. We can look at these ratios to understand how notes with different frequencies interact when played at the same time. The ratio between pitch $i + n$ and i is calculated as

$$\frac{440 \cdot \left(\sqrt[12]{2}\right)^{i+n-49}}{440 \cdot \left(\sqrt[12]{2}\right)^{i-49}} = \left(\sqrt[12]{2}\right)^{n}$$

Given our 12 note scheme between octaves, there are 13 intervals we can define within an octave, where $0 \leq n \leq 12$.

Interval ratios that can be approximated by low integer numbers are perceived as consonant, or more stable and blending well. Those that are approximated by large integer numbers are perceived as dissonant, or more unstable and clashing. For example, the most consonant non-unison interval (where unison is two of the same note played at the same time, with ratio $\frac{1}{1}$) is an octave, which can be written as the ratio $\frac{2}{1}$. The next most consonant interval is a perfect fifth, which can be written as $\frac{3}{2}$. The most dissonant interval is the augmented fourth, which is best approximated

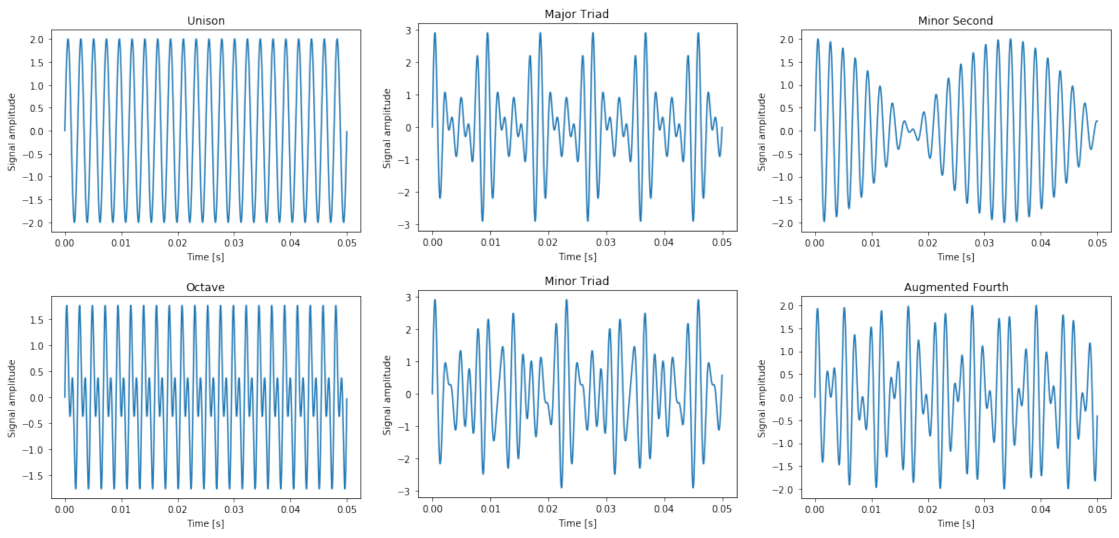


Figure 3-4: Waveforms of Interval Ratios. Lower integer interval ratios resolve to consonance in part by repeating more often within the same time frame (# repeats). Top row, left to right: Unison (21), Major Triad (5), Minor Second (1.5). Bottom row, left to right: Octave (21), Minor Triad (2), Augmented Fourth (<1).

by the ratio $\frac{45}{32}$.

In ascending order of n, the names and ratios of each interval in an octave are: 0. perfect unison ($1.0 = \frac{1}{1}$) 1. minor second ($1.0595 \approx \frac{16}{15}$) 2. major second ($1.1225 \approx \frac{9}{8}$) 3. minor third ($1.1892 \approx \frac{6}{5}$) 4. major third ($1.2599 \approx \frac{5}{4}$) 5. perfect fourth ($1.3348 \approx \frac{4}{3}$) 6. augmented fourth ($1.4142 \approx \frac{45}{32}$) 7. perfect fifth ($1.4983 \approx \frac{3}{2}$) 8. minor sixth ($1.5874 \approx \frac{8}{5}$) 9. major sixth ($1.6818 \approx \frac{5}{3}$) 10. minor seventh ($1.7818 \approx \frac{16}{9}$) 11. major seventh ($1.8877 \approx \frac{15}{8}$) 12. perfect octave ($2.0 = \frac{2}{1}$).

Qualitatively, we can understand that low integer interval ratios sound more consonant because they form simple frequency patterns that repeat much more often in a given time frame than high integer interval ratios. A visual example of this phenomenon is given by figure 3-4.

Note that there are a number of different tuning systems, and we are using the Equal Temperament scale when we say that intervals are based on the octave and geometrically separated in frequency by $\sqrt[12]{2}$. Another option, for example, is the Pythagorean tuning system, where frequency ratios of all intervals are made from pure perfects fifths and octaves [42].

3.2.3 Chords

A chord is a combination of two or more notes. In tonal Western music, the fundamental building blocks of chords have three notes, also known as triads.

With chords, we talk about intervals relative to the root note, or lowest pitch note in the chord. A major triad consists of a root, major third, and perfect fifth, and a minor triad consists of a root, minor third, and perfect fifth. Visual differences in waveform can be seen in figure 3-4.

Western music often utilizes chord progressions, which are sequences of chords, as a rhythmic bass under a higher pitched melody. Progressions introduce the idea of resolution at a broader level than simple intervals.

Chords are named relative to the key, or group of notes / scale that forms the basis of a musical composition. The so-called tonic note of this group serves as a kind of implied root note for the piece. For example, a composition in C major features a scale C D E F G A B C, with a tonic note C.

We name chords relative to this tonic note, with a Roman numeral indicating the root note's distance from the tonic, and capital / lowercase representing major / minor. For example, a very common progression in Western pop music is a repetition of the four chords I V vi IV. In a C major key, these numerals represent the triads C major, G major, A minor, and F major respectively.

Returning to the tonic chord, I, establishes a sense of resolution for the listener. We can theorize that I V vi IV is a popular progression because the listener is hooked into a cycle of 1) listening to the end of a 4 measure section, and 2) continuing to listen for the IV to I resolution to start the next 4 measure sequence.

3.2.4 Overtones

When we play a note on a piano or string based instrument, secondary vibrations on the string that occur at higher frequencies than the original note (figure 3-5) emit additional vibration energy called overtones. Wind, brass, and other instruments vibrate in similar ways to produce overtones. This natural variation in vibration

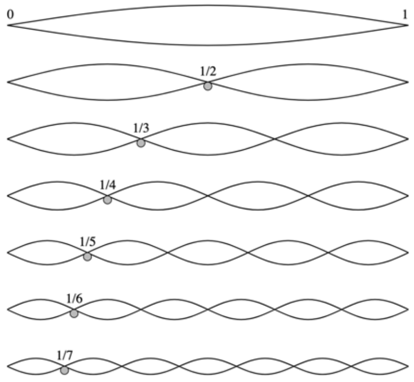


Figure 3-5: String partials.

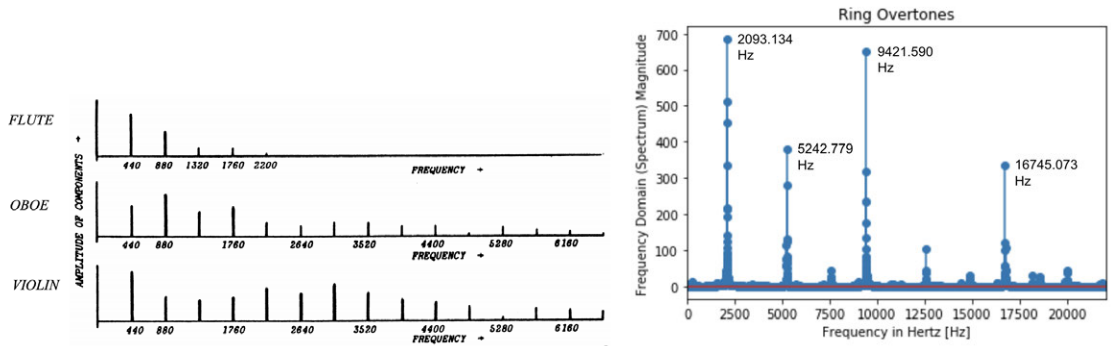


Figure 3-6: Overtones. Left, top to bottom: Flute, Oboe, Violin. Right: Overtones produced by the acquiring ring sound in Sonic. The root note is 2093 Hz, or C7 on a piano. The top 3 remaining frequency magnitudes from left to right are 5242 $(2.5 \cdot 2093)$ Hz, 9421 $(4.5 \cdot 2093)$ Hz , and 16745 $(8 \cdot 2093)$ Hz.

gives instruments a fuller sound that reaches more levels of the frequency spectrum, and part of their unique, identifiable character or timbre.

Interval ratios become all the more important for maintaining consonance when playing instruments with overtones. There can be 20-30 audible frequencies being played between just two notes on a violin (figure 3-6). Luckily, overtones operate on small interval ratios, by nature of how they are produced. Therefore, maintaining consonant intervals between root notes is generally sufficient for retaining consonance as a whole.

3.2.5 Western Associations in Music

In Western music theory and culture, there are clear associations of consonance, dissonance, and rhythmic patterns with certain emotions. The major mode is slightly more consonant, and therefore associated with "happy, merry, graceful, and playful", while the minor mode is slightly more dissonant, and associated with "sad, dreamy, and sentimental". Firm rhythms are perceived as "vigorous and dignified", while flowing rhythms are "happy, graceful, dreamy, and tender". Combining these ideas, complex dissonant harmonies are "exciting, agitating, vigorous, and inclined towards sadness", and simple consonant harmonies are "happy, graceful, serene, and lyrical" [21].

Studies have shown that these associations can largely be attributed to cultural conditioning, rather than universal human behavior. Residents of a village in the Amazon rainforest with no exposure to Western music showed no preference between consonant and dissonant sounds [61]. We can trace back Western preference for consonance at least to the early 18th century, when the name "diabolus in musica", or "the Devil in music" was attributed to the augmented fourth, the most dissonant interval [1].

3.2.6 Application to Videogame Sound Effects

Videogame music and sound effects are designed to leverage our implicit biases as a means of meeting expectation and lowering the barrier to entry for learning a new game. We can better understand this by applying it to diegetic analysis of the Sonic games.

It is good for Sonic to collect rings, since they give some protection before losing a life, award more points, and collecting enough of them result in an extra life. Recall from our IEZA framework (section 3.1.1) that the ring sound falls in the Effect category, i.e. a diegetic sound that conveys information about in game actions. In line with our understanding of Western preference for simple consonance, the ring sound is a major triad, implying a sense of positive value for the act of acquiring

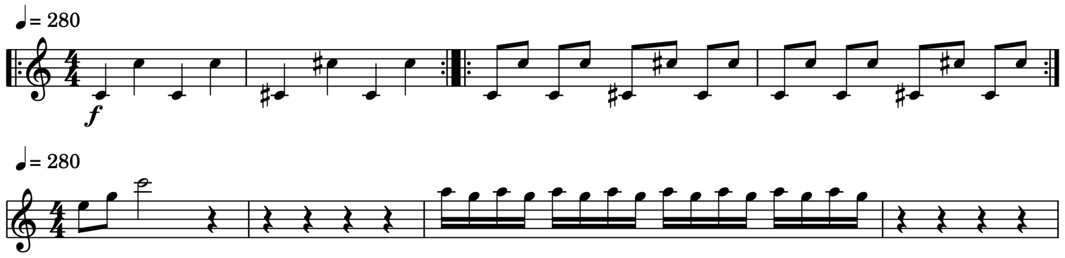


Figure 3-7: Drowning Theme and Ring Sounds. Top: The drowning theme starts out slower, and increases in speed, with dissonant jumps between octave intervals. Bottom left: three consonant notes of the ring acquiring sound. Bottom right: alternating dissonant notes of the losing rings sound.

a ring. It is comprised of the notes E5, G5, and C6, which form specifically a first inversion C major triad (figure 3-7).

When Sonic loses rings after being hit by an enemy or hazard, this is bad because he is prone to losing a life with an additional hit. Recall that the losing rings sound also falls in the Effect category. In line with our understanding of Western music theory, the corresponding sound is aggressively dissonant, implying a sense of negative value for the act of losing rings. This sound is created by rapidly alternating notes A6 and G6, which form a major second interval (figure 3-7).

When Sonic has spent too much time underwater without air, and is close to drowning, the drowning theme begins to play. It alternates between octave intervals on C, and a minor second interval jump to octave intervals on C# (figure 3-7). The jumps lie on a dissonant interval, and the music gets increasingly loud and fast as Sonic gets closer to drowning, giving a strong sense of impending doom. This is an example of an Interface sound, i.e. non-diegetic and relating to in game actions, relaying negative value for the act of drowning.

In general, consonance bias is an effective tool for conveying positive or negative value of player actions or setting emotional expectation with audio, without needing to spoon-feed explicit instructions or visual cues.

3.2.7 Application to Videogame Music

Nakamura designed the Zone themes, which fall in the Affect category, to set emotional expectation using a wide array of musical concepts.

The bass section of the Death Egg Zone theme begins with alternating the root note A with G (minor seventh), F (minor sixth), and C (minor third), which are all considered to be dissonant intervals. This gives it an ominous and dangerous emotion, which fits with the level design and setting inside of Dr. Robotnik's evil Death Egg. A slow, chromatically rising 5th on a minor chord, which is commonly associated with the suspenseful James Bond theme, is used to bridge sections of melody.

In comparison, the Green Hill Zone theme begins with a lively repeating C octave interval on the bass, and iconic alternating E minor and D minor chords in the melody. This invokes a positive feeling of excitement and adventure, which is fitting for its placement at the beginning of the game.

The melody that comes after this section follows a IV^7 iii^7 ii^7 I^7 progression. The resolution from ii to I is soft and almost relaxing, especially in comparison to a stronger and more common V to I resolution. Coloring the progression with diatonic 7ths helps to create a dream-like, floating feeling. Overall, this second section provides a soothing and nostalgic contrast to the energetic beginning of Green Hill Zone. This change in tone helps to keep the music feeling fresh, while staying within the overall intended feel of the Zone.

Note that this style of analysis is meant to be simple enough to get a point across; composing music is an art form, and not just a set of rules that are easy to apply and get an intended emotional effect.

There are countless other musical techniques that Nakamura uses to craft a genre and emotion around each Zone theme, such as background/foreground melody switching, broken and full chords, harmonies and counter-harmonies, and a diverse set of instrumentation. Some other themes we encounter are a funky jazz feel in Spring Yard, ascending and airy in Star Light, percussive and industrial in Labyrinth, bluegrass in Hill Top, Middle Eastern in Oil Ocean, electro rock in Metropolis, and medieval march in Wing Fortress. Nakamura's approach was picked up handily in Sonic 3&K by a number of artists including Michael Jackson, resulting in a soundtrack taking influences from calypso, funk, carnival, new wave, prog rock, hip hop, R&B, and more [35]. Overall, the sheer variety and complexity of music on each of the Sonic soundtracks set it apart from any other franchise of its time.

Chapter 4

Working with Audio

The goal of this chapter is to introduce methods for obtaining, listening to, visualizing, and processing audio features made available by the Gym Retro interface.

4.1 Extracting Audio from Gym Retro

The Open AI Gym Retro package does not readily expose an API for interacting with the retro emulator's audio. Upon further investigation, we find that the Emulator class does indeed have a function called "get_audio()". We can use this function to get the audio samples that correspond to the current frame.

```
env = retro.make("SonicTheHedgehog-Genesis", "GreenHillZone.Act1")
obs = env.reset()  # initial video observation
samples = env.em.get_audio()  # initial audio observation
```

The default sample rate and fps are 44100 samples/sec and 60 frames/sec, so we expect to receive $\frac{44100}{60} = 735$ samples per frame. After running a simple loop for generating observations, we find that this is not quite the case. Most of the time we receive 735 samples, but we also sometimes receive 736. We assume there is potentially some rounding error happening in the emulator, and for simplicity we truncate the number of received samples to 735. Even if we are missing the occasional sample, the difference is not very consequential.

With this understanding of how to extract audio from the emulator, we can move forward with incorporating audio into our visualization and feature processing pipelines.

4.2 Offline Playback with Audio

The Gym Retro package offers built in methods for viewing and saving episode videos, but does not have an option for doing the same with episode audio. To be able to listen to the audio generated by our retro environment alongside the video, we first open temporary video and audio files with the opencv-python [50] [51] and wave packages. At each environment step, we write the corresponding video frame and audio samples to these files. Once the environment step loop is finished, we make a subprocess call to FFmpeg [17] with the Python subprocess library. We join the video and audio with the following FFmpeg command:

```
ffmpeg -y -i /tmp/temp.webm -i /tmp/temp.wav -c:v copy -c:a aac -strict
experimental vid_with_sound.webm
```

Note: We use jupyter notebook as a convenient interface for viewing videos on a remote server in this fashion without needing to download them locally. Saving videos under the webm file format is required for playback embedded in the Chrome browser.

4.3 Real Time Playback with Audio

There are a number of ways to implement real time playback, and each come with different tradeoffs. We decided to prioritize a simple playback system without additional multi-threading or buffering, at the minor expense of performance and accuracy.

We display video and play audio using two separate components. For video, we send generated frames to Gym Retro's SimpleImageViewer class, with a call to the imshow function. This is the same class used under the hood for Gym Retro's built-in environment render function.

For audio, we send samples to the speaker system's audio buffer using the PyAudio [56] Python package. PyAudio provides Python bindings for PortAudio [8], which is a cross-platform audio I/O library written for C. We use PyAudio to initialize a stream with a buffer size, and we can simply write samples to this stream to play audio from our speaker system.

4.3.1 Buffer Underflow and Overflow Issues

If we decide to select a random action at each step instead of doing something more computationally expensive, the environment loop that generates observation data typically runs much faster than 60 frames per second. When visualizing frames of video, this is not a big deal as it simply results in visually "fast forwarded" playback.

If we are generating and playing audio samples, however, this means that we are putting samples into the buffer at a faster rate than they are being played. Therefore, we end up overflowing the buffer, and losing the audio frames which were previously queued in the buffer. Overflowing the buffer is itself computationally expensive, so we also end up with pockets of silence in the audio during the time that it takes to perform a buffer reset. These pockets of silence create discontinuities between samples, resulting in loud and jarring clicking noises when played back. Overall, these issues make for an unpleasant listening experience.

We can check how much buffer writing capacity is available, so a solution to prevent buffer overflow would be to truncate our samples to the available capacity before we send them to the buffer.

Truncation still suffers from the problem of losing audio data, so a secondary solution for this might be to add an artificial delay to our environment loop, to make it run closer to 60 frames per second. However, finding the correct delay is empirically difficult, especially because of variance caused by external factors on the machine running the code. Figure 4-1 shows how small error in this artificial delay that prevents us from reaching the exactly desired fps can add up over time into a buffer underflow.

Buffer underflow occurs when our environment loop runs slower than 60 frames

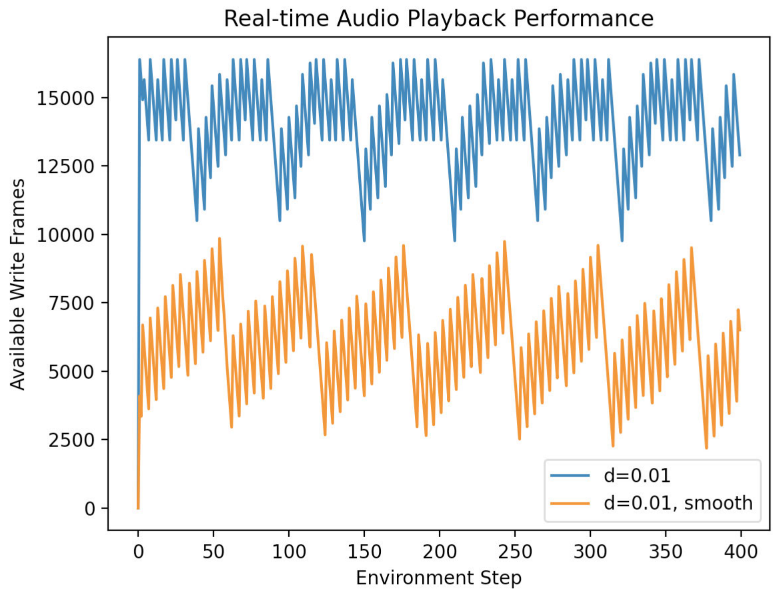


Figure 4-1: We estimate 0.007 seconds per step in the environment loop and therefore apply an artificial delay d of 0.01 seconds to reach 60 fps. We see that without smoothing (blue), small error in our estimation adds to repeated buffer underflow, or full 16k frame write capacity. After applying smoothing method with dynamic resampling (orange), we avoid sample truncation and also prevent buffer underflow.

per second. Since we are now putting in audio samples at a slower rate than they are being played, the buffer eventually runs out of samples to play. Underflow creates pockets of silence, and the audio discontinuities that bookend these pockets of silence produce jarring clicking noises, as mentioned above. Note that clicking noises should not be confused with the term audio clipping, which is when audio is played too loudly and becomes distorted, as in the Sanic Hegehog meme [78] [53].

4.3.2 Smoothing with Dynamic Resampling

To handle the issues of buffer underflow and overflow, we introduce dynamic audio resampling into our system. Each time an observation is generated by the environment, we calculate the time since the last observation was generated, and resample the audio under the assumption that this time will be representative of time taken to generate future observations as well.

The Sonic Retro environment runs at 60 frames per second (fps), and the audio sample rate is 44100 samples per second. This means we expect to receive 735 samples of audio per frame, and the expected time difference between each frame is $1/60 = 0.0167$ seconds. If the actual time difference 0.015 seconds, this means we need to account for this change by sending $\frac{0.015}{1/60} \cdot 735 = 441$ frames to the buffer instead. Instead of simply truncating from 735 to 441, we can perform one-dimensional interpolation to preserve the acoustic features in the samples.

In general, we can say that the number of samples s needed to send at environment step i is given by

$$s = \frac{t_i - t_{i-1}}{1/60} \cdot 735$$

where t_i represents the wall clock time at which environment step i was generated. Using one-dimensional interpolation allows us to scale our 735 samples to s for any $s > 0$.

4.3.3 Tradeoffs in Practice

Although this system prevents both overflow and underflow even in extreme cases (figure 4-2), there are a few tradeoffs.

First of all, simple resampling with interpolation alters the audio pitch. If the environment loop runs faster than 60 fps, the audio becomes higher pitched, and if the loop runs slower, the audio is lower pitched. We suggest adding artificial delay to very fast loops and reverting to offline video playback for very slow loops.

Second of all, even when this system runs on a 60 fps loop, we typically experience a slight audio lag behind the video. The first few Gym Retro environment observations usually go through a "warm-up" period where they take longer to generate, which means those audio samples are sampled longer. This "warm-up" phase is followed by a relative speed up to normal generation speed, but the previously resampled audio samples already in the buffer take a disproportionately longer time to play, causing a delay.

The lag may also be experienced without a "warm-up" phase, because we start

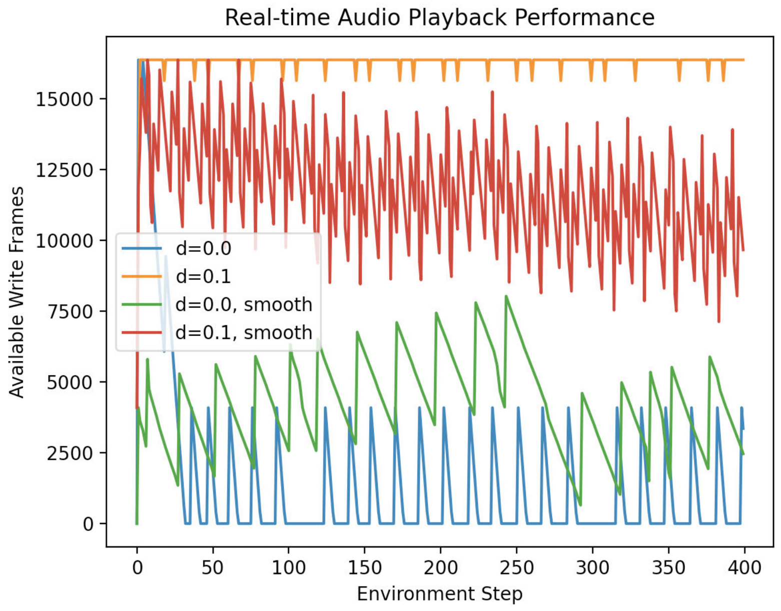


Figure 4-2: We present two extreme cases: repeated overflow with no artificial delay (blue), and repeated underflow with large artificial delay (orange). Our smoothing method with dynamic resampling prevents both buffer overflow and underflow in both cases (green, red).

by assuming the environment loop is running at 60 fps. We cannot resample the first set of audio samples, because there is no 0th set of samples upon which to compute a time difference. If the environment loop runs faster than 60 fps, then the first set of samples will take disproportionately longer to play from the buffer.

We could conceivably devise a more complicated audio scheme that accounts for the warm up phase or 0th sample set issue by introducing an additional buffer, but we feel this makes the system unnecessarily complicated. A large enough secondary buffer effectively leads to the offline playback system described in section 4.2, so we leave offline playback as a means for generating video with audio aligned perfectly.

4.4 Visualizing Audio with Spectrograms

Now that we can hear the gameplay audio, we can look into visualization techniques to understand how to see it.

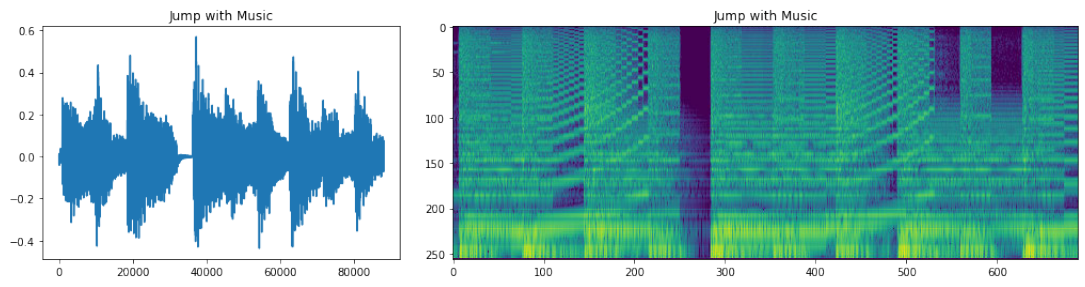


Figure 4-3: An audio clip of Sonic jumping twice, with the Green Hill Zone theme playing in the background. On the left, shown as a 1D waveform. On the right, shown as a 2D spectrogram. It is much easier to see the jump sounds in the spectrogram (the lines curving upward, approximately 110 to 210 and 425 to 525 on the x-axis).

As a reminder, the audio samples that we are given are one-dimensional: a sequence of float values. We can plot these values to see what is called a wave form, as shown in figure 4-3. Looking at the amplitude, or y-axis, we can get an idea of where the audio is louder or quieter, but the plot as a whole is unable to tell us about other important information such as pitch, frequency, or harmonic content. In other words, we can get an idea of where sounds are being made, but it is difficult to discern what sounds they are.

4.4.1 Short Time Fourier Transform

We can use the Short Time Fourier Transform (STFT) [12] [66], along with a few other techniques, to decompose the one-dimensional samples into a two-dimensional spectrogram. The x-axis of this spectrogram represents time, the y-axis represents frequency, and values represent magnitude. The high level procedure of this process is as follows.

A sliding window is used to divide the audio samples into chunks of equal length, with partial overlap. We compute the Fourier transform on each of these chunks, revealing the Fourier spectrum at the center of each chunk. Each computed spectrum is one-dimensional, with complex values. It is common practice to take the magnitude of this complex array, to obtain a real-valued array where the axis represents the range of frequencies, and values represent the amplitude of the corresponding frequency (we used this technique to uncover overtones in figure 3-6).

We could similarly recover real-valued phase information from the FFT by taking the angle of the complex values. However, our ears generally cannot distinguish phase offsets, and therefore this data is typically thrown out. Phase information is important for reconstructing an audio signal from a spectrogram, but not often used for analysis.

By compiling the magnitudes of the Fourier spectrums and arranging them in the same order as the chunks used to compute them, we can see how the loudness of each frequency changes over time. This is a spectrogram.

4.4.2 Mel Scaling

Stevens et al. [72] found that human perception of frequency scales logarithmically. This aligns with our understanding from section 3.2.1 that pitch frequencies follow a geometric sequence. On a piano, this is demonstrated by the fact that the two lowest semitones (A0 and A0#) are separated by 1.6 Hz, while the two highest (B7 and C8) are separated by 234.9 Hz.

To account for this phenomenon in our spectrogram, we apply a non-linear transformation called a Mel Scale. This transforms the y-axis so that distance scales linearly relative to human perception of frequency. In other words, it is logarithmically scaled so lower frequencies take up more space, and higher frequencies are packed closer together.

To convert from a frequency f Hz to m mels, we can use the following formula

$$m = 2595 \log_{10}\left(1 + \frac{f}{700}\right)$$

Any time we mention or display a figure of a spectrogram, we will be referring to a Mel-scaled spectrogram.

4.4.3 Hyperparameter selection

There are a few hyperparameters needed to specify the sliding window and frequency resolution used to produce our spectrogram. These values should be empirically determined based on the type of audio being transformed. We will go over what we found worked well for videogame sound, along with general best practices. Examples of spectrograms constructed with these hyperparameter values can be found in figures 4-3, 4-4, 4-5, and 4-6.

- Hop Length H. This hyperparameter determines how far we move the sliding window after each Fourier spectrum calculation. Using a larger hop size means lower time resolution and faster computation, and vice versa for smaller hop size. The length of our resulting spectrogram relative to our number of samples S and hop length H is simply given by $\frac{S}{H}$.

  When we account for our sample rate S_r, we can calculate the length of each hop in seconds with the expression $T_{hop} = \frac{H}{S_r}$. Typically, this value of T_{hop} is around 93 ms for music processing and 23 ms for speech processing.

  Since we are working with artificially generated videogame audio, we can afford for this T_{hop} value to be much lower, without adding a lot of noise. We found that setting $H = 512$, resulting in $T_{hop} = \frac{512}{44100} = 11.6$ ms, gave us good temporal resolution for our sound effects which are on average around 0.5 to 1 seconds long. We also experiment with a variant, $H = 128$.

- Number of frequency bins N_{fft}. This hyperparameter defines the frequency resolution, or number of frequency bins, of our non-Mel spectrogram. (This spectrogram will eventually be Mel scaled to a new frequency resolution N_{mels}.)

  N_{fft} is closely related to our sample rate, because the highest possible frequency that can be recorded is half of the sample rate. The Sonic games operate on a sample rate of 44100 Hz, which means we could theoretically set N_{fft} to a value of $\frac{44100}{2} = 22050$ in order to separate frequencies into bins 1 Hz apart.

In practice, this level of frequency resolution is excessive for differentiating between sounds, and larger values of N_{fft} result in both increased computational complexity and reduced temporal resolution. For a sample rate of 44100, typical values of N_{fft} include 2048 and 4096. We decided to set $N_{fft} = 2048$.

- Window Length W. This hyperparameter determines the size of our window at each Fourier spectrum calculation. In the large majority of cases, it is standard to set $W = N_{fft}$. This maximizes the spectrum information available in an N_{fft} size window of audio samples.

  However, we experiment with setting $W < N_{fft}$ as a method for adding redundancy to our audio observation. As mentioned previously, videogame sounds are artificially generated, so we expect that it may be easy to overfit a model to audio that has high resolution and zero natural variation. When we set $W < N_{fft}$, we must add $N_{fft} - W$ data points of either redundancy or noise to get a total of N_{fft} samples for computing the Fourier spectrum. We set these additional points to zero so that our artificial inflation of frequency resolution is achieved with redundancy. We could have introduced similar methods for inflating frequency resolution with choice of N_{fft} or N_{mels}, but we choose to localize it to our choice of W.

  Window length is also closely related to hop length. It is good practice to set $W = cH$, where $c > 1$, so that the sliding window "revisits" samples $c-1$ times. This idea of overlap with previously visited points usually makes for a smoother spectrogram. We empirically choose to set $W = 2H$.

- Number of Mel frequency bins N_{mels}. This hyperparameter defines the frequency resolution, or number of frequency bins, of our spectrogram after applying Mel Scaling. This is the length of the y-axis for our final spectrogram.

  After creating a non-Mel spectrogram with frequency resolution N_{fft}, Mel Scaling applies a logarithmic transformation on the frequency spectrum of this spectrogram. It is not possible to reconstruct a higher resolution of the lower frequencies, so instead we bucket together higher frequency fft bins in a way that

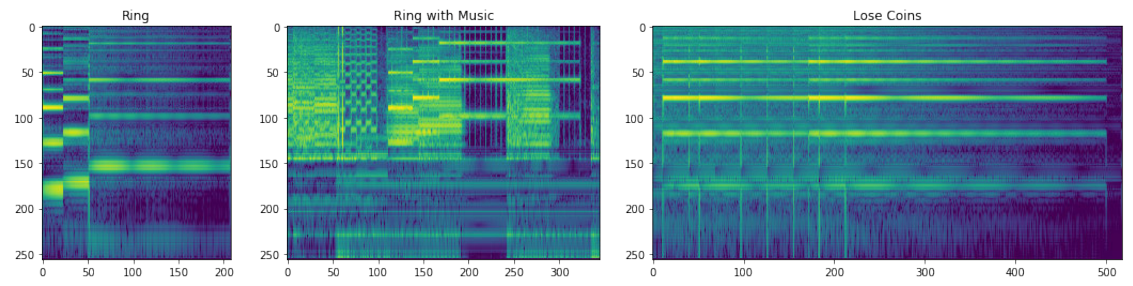


Figure 4-4: Three ring-related spectrograms, with widths scaled proportional to the audio length. From left to right: 1) Isolated audio of Sonic collecting a ring. 2) Sonic collecting a ring to the backdrop of the Hill Top Zone theme. Note how the ring sound effect (approximately 125 to 350 on the x-axis) is still clearly visible in the higher frequencies. 3) Isolated audio of Sonic losing his rings.

ends up proportional to the log scale. This operation reduces the overall frequency resolution of our spectrogram, so N_{mels} is typically set to be 4 to 8 times smaller than N_{fft}.

If we were to set N_{mels} to be too large, for example $N_{mels} = N_{fft}$, we would end up creating redundant and/or disruptive artifacts in the lower mel bins. Many mel scale implementations use a 1 to 1 mapping from fft to mel bin, so mel bins without an fft mapping would simply contain zeros. Visually, we might see this as black horizontal lines interrupting the lower frequencies of our spectrogram. On the other hand, if N_{mels} is too small, the spectrogram does not tell us detailed frequency information. We decided to set $N_{mels} = 256$.

4.4.4 Visibility of Sound Effects

Now that we can visualize Sonic game audio, we can predict which sound effects might be easier for our agent (defined in section 5.2) to see in a given spectrogram.

In figures 4-3 and 4-4, we can see how the backing Zone theme consistently dominates the lower frequencies. This is typical of songs in general: the lower frequencies consist of repetitive rhythmic or droning elements, while the higher frequencies are left for more sparsely populated, melodic tunes. Grabbing and losing rings, along with the upper half of the jump sound, are still recognizable in this musical context. We will look to see if lower frequency sound effects that relay important information,

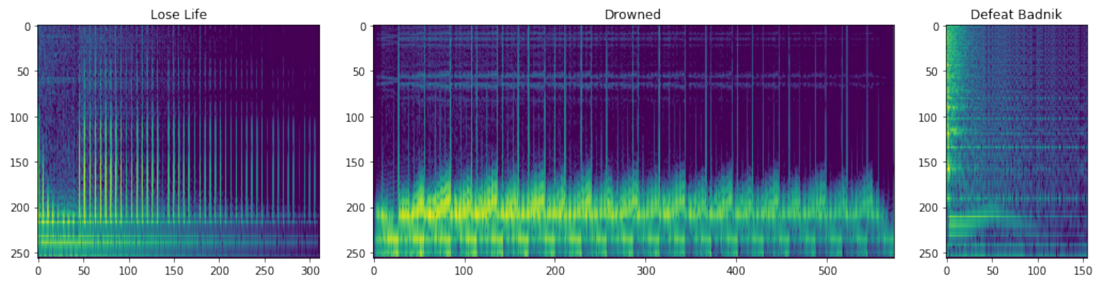
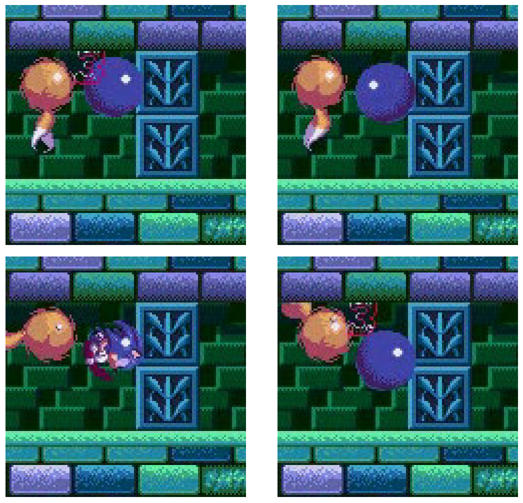


Figure 4-5: Three sound effects that are low pitched. These are harder to identify in a spectrogram with the Zone theme playing in the background, since the lower frequencies (around mel filter 150 to 256 on the y-axis) are more "polluted" with the rhythmic part of the backing track. From left to right: 1) Losing a life. 2) Losing a life from drowning. 3) Defeating an enemy.



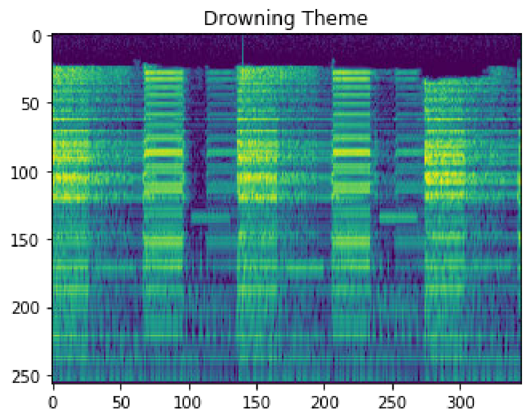


Figure 4-6: Video and audio approaches to conveying that Sonic is drowning. Left: A small see-through bubble in the shape of a 3 flashes on and off between timesteps. Once this bubble counts down to 0, Sonic will drown. This is very hard to notice, even when watching continuous video playback. Right: The loud and ominous drowning theme interrupts the Zone theme and any other sound effect being played at the time. It very clearly takes up the entire range of of the audio spectrogram.

such as losing a life, or defeating an enemy, can be recognized by the agent as well (figure 4-5).

The drowning theme is an exception, because it interrupts the Zone theme. We can see in figure 4-6 that this theme is much easier to see in a spectrogram than the corresponding visual effect, especially at a frame by frame level.

Chapter 5

Experimental Setup

In this chapter, we outline the experimental setup that we use to test the hypothesis that videogame audio features contain important and transferable decision making information.

5.1 Environment Setup

5.1.1 Save States

We use 58 save states, each representing the start of a level in one of the 3 Sonic games. Each one begins with a standard set of 3 lives and no Chaos Emeralds. Save states allow us to choose the specific level we want to train on, rather than having to complete all of the levels that precede it. They also allow us to skip the starting screen, character selection, boss fights, special stages, some cut scenes, and other parts of the game which are not representative of 2D platforming (figure 5-1).

By adding level end conditions as described in section 5.1.2, we keep each level self-contained. This denies the possibility of an agent retrieving all Chaos Emeralds and transforming into Super Sonic, or completing a level and proceeding to a boss fight.

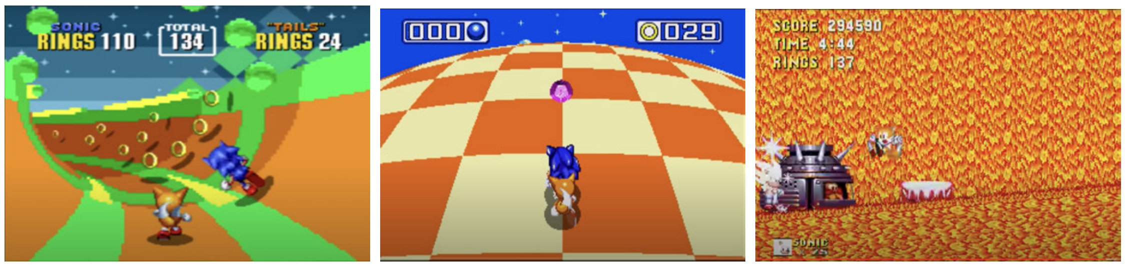


Figure 5-1: Examples of special stages and a boss level. From left to right: 1) Special stage in Sonic 2 where Sonic collects rings in a twisting halfpipe to retrieve a Chaos Emerald. 2) Special stage in Sonic 3&K where Sonic navigates a sphere with a top-down view to retrieve a Chaos Emerald. 3) Boss fight against Dr. Robotnik at the end of Lava Reef Zone, where invincible Super Sonic wins the battle with ease.

5.1.2 Episode Boundaries

To quickly review, Sonic begins the game with a default of 3 lives, and a time limit of 10 minutes to complete each act. If Sonic runs out of lives, runs out of time, or completes a level, this marks the boundary, or end, of an episode.

We make a few modifications to these default boundaries, so that episodes end on the following conditions:

1. Sonic loses a life. This is different from Sonic having 1 life. Sonic can gain extra lives with enough rings, but extra lives do not prolong this boundary any further.

2. Sonic runs out of time, where the time limit is 5 minutes instead of 10.

3. Sonic completes a level, which we define as reaching a pre-defined horizontal offset for that level. This allows us to avoid dealing with boss fights, which can be found beyond this horizontal offset in the final levels of some zones.

5.1.3 Stochastic Frame Skip

The retro environment for the Sonic games runs at 60 frames per second (fps). In other words, there are 60 button press opportunities in one second. We found that the average human, on the other hand, can make 8 to 10 button presses per second. To make the model input rate more realistic, we use a frame skip of 4 to bring the

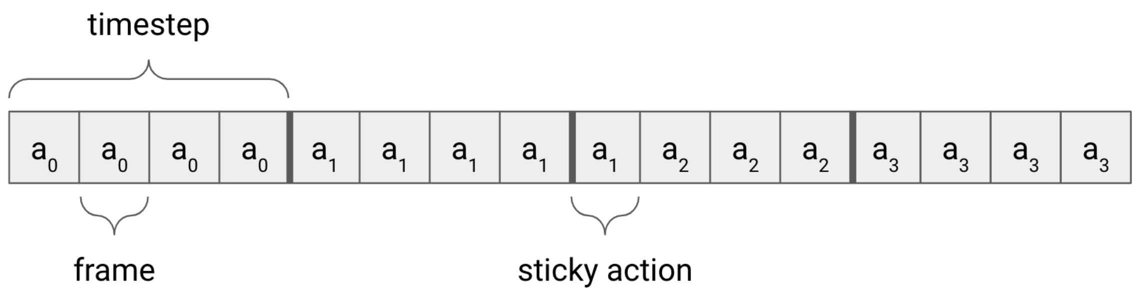


Figure 5-2: Visual demonstration of frame skip.

actionable fps down to 15. The idea of a frame skip is to repeat a given input n times, where in this case $n = 4$. Using a frame skip of 4 is common practice for ALE environments, and going forward we will refer to the actionable frames occurring at 15 fps as time steps.

When we construct our observations, we can now choose to either keep all of the n frames, for example by concatenating them to form a sequence of images, or throw out $n - 1$ frames so we are only left with the most recent one. When working with video with a frame skip of 4, we choose to throw out the 3 unactionable frames and avoid adding unnecessary complexity. We assume that the screen does not change much over the course of $3/60 = 0.05$ seconds.

When working with audio, throwing out intermittent frames creates to discontinuities, which in turn produce loud and disruptive clicking sounds. This can happen even for small sizes of skipped audio frames. As a result, we decide to keep all 4 frames of audio in our observation when using our frame skip of 4.

To help prevent the model from memorizing deterministic solutions to a level, we also incorporate some stochasticity to make a "sticky" frame skip. This introduces randomness into actions taken by the agent by repeating an action $n + 1$ instead of n times, with some probability p. If the action is repeated $n + 1$ times, then the next action is only repeated $n - 1$ or n times, depending on if it is also a "sticky" action. Figure 5-2 visualizes this idea of a sticky frame skip. We choose to set $p = 0.25$, following previous works [47].

5.1.4 Memory Buffer

We create a time step memory buffer of size m, that optionally augments each observation with frames from $m-1$ previous time steps. For $m > 1$, key game state variables, such as Sonic's relative velocity to on screen enemies or hazards, can theoretically be calculated by an agent.

In our experiments, we set the video memory buffer size m_{video} to 1 or 2. Note that a value of 1 represents only the current time step. The video data is quite dense, and we found that larger values of m_{video} made it difficult to interpret or extract meta-information.

For our audio observations, having a longer memory size is important because very little audio information can be conveyed in just 1 time step. The sound of Sonic acquiring a ring takes about 0.6 seconds to play, and the sound of Sonic losing his rings takes about 1.3 seconds. In comparison, 1 time step of audio is $4/60 = 0.066$ seconds. We set $m_{audio} = 16$, which is 1.066 seconds.

If we wish to learn from the game music or sounds longer than simple sound effects, we might need to consider a time scale on the order of several seconds. However, we found that most themes and longer sound effects are still recognizable in 1 second chunks.

5.1.5 Video Processing

In our Sonic Retro environment, video observations are 24-bit RGB images, 320 pixels wide and 224 pixels tall. These images represent the screen that the game is played on.

We optionally modify these images by converting from RGB to grayscale. We perform this conversion from $\{r, g, b\}$ to y using the weighted formula [24]:

$$y = .2126 * r + .7152 * g + .0722 * b$$

Then we normalize these grayscale values to the range [0,1]. In general, colors are not vital to the completion of the levels, and so this processing method helps to reduce the complexity of our observations. Our image depth is thus reduced from 3 to 1.

If our memory buffer size m_{video} is 2, we concatenate the additional buffer image along the depth axis. Now our new image depth m_{video} represents time rather than values of RGB, where each image along this time axis is in grayscale. Although this is not a standard processing step, it allows for a more modular experiment setup and some potential performance benefits described in section 5.2.2.

5.1.6 Audio Processing

The Sonic retro environment produces 735 samples for 1 frame of audio. Since we keep all audio samples that are generated between skipped frames, this comes out to $735 * 4 = 2940$ samples per individual time step. When we add the memory buffer audio from the previous 15 time steps, we end up with a total of $2940 \cdot 16 = 47,040$ samples per observation. Recall that our sample rate is 44100 Hz, which is why each audio observation ends up being approximately 1 second long.

We process the 1D sequence of samples into a 2D mel spectrogram with 256 mels, and either use window size 256 and hop length 128, or window size 1024 and hop length 512. The former produces a spectrogram similar in size as a frame of video (367 by 256 pixels), while the latter contains the same information but 4 times smaller (92 by 256 pixels).

To normalize the spectrogram values, we first convert from the power to decibel scale, setting the max decibel range to 80, and then divide by 80 to put the range of values between 0 and 1.

5.1.7 Action Space Reduction

At each time step, the agent can press any combination of buttons that are available. For SEGA Genesis games, the buttons on the controller are `A, B, MODE, START, UP, DOWN, LEFT, RIGHT, C, Y, X, Z`. Since there are 12 buttons, the number of possible button presses at a single time step are on the order of 12!, or about 480,000.

Given the finite amount of Sonic moves, described in figure A.1, and the restricted setting of 2D platforming, we reduce all possible combinations of button presses to the following 8 essential ones: `{}`, `{LEFT}`, `{RIGHT}`, `{DOWN}`, `{B}`, `{LEFT, DOWN}`, `{RIGHT, DOWN}`, `{DOWN, B}`.

`UP` is used to make Sonic look up and slightly shift the screen upwards, which is only useful on certain occasions. The remaining moves in Sonic's moveset can be constructed with the button presses above. `B` is the button used to jump.

5.1.8 Reward Function

For each action that is taken by the agent, a reward is received as a result of that action on the environment. In general, this can be negative, positive, or zero. We define our reward function in two parts: a level progression reward, and a level completion bonus.

As the agent progresses through the level, we keep track of its horizontal offset relative to the start of the level, and relative to the maximum horizontal offset it has reached in this episode. Each time the agent achieves a new max horizontal offset, it receives a positive reward proportional to both the size of improvement, and horizontal length of the level. The net reward gained for reaching the horizontal offset required for completing the level is 9000. In other words, the level progression reward can be written as

$$r_{progress} = \frac{\max(0, x_{cur} - x_{max})}{d} \cdot 9000$$

and the update step after each time step can be written as

$$x_{max} = \max(x_{cur}, x_{max})$$

By scaling horizontal offset to a net reward of 9000, we can compare levels of varying length. Assuming all levels are of similar difficulty, this scaling also controls for potentially easier progress made in longer levels. We do not penalize the agent for moving backwards, because many levels require moving to the left for extended

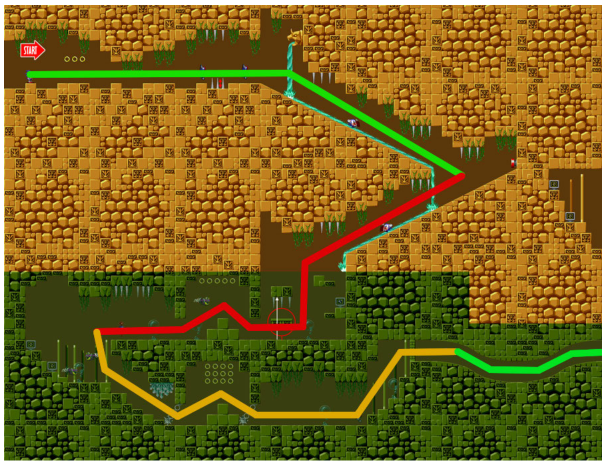


Figure 5-3: In Labyrinth Zone Act 2, significant backtracking is required to progress through the first part of the level. We found that section in red and orange takes a human around 20-30 seconds to complete.

periods of time to complete the level (figure 5-3).

To incentivize the agent to move quickly, we also include a completion bonus proportional to the amount of time taken to complete the level. The completion bonus starts at 1000, and scales linearly to 0 at 4500 time steps. We can write this bonus as

$$r_{complete} = \frac{4500 - t}{4500} \cdot 1000$$

This reward is never negative because one of our episode boundaries is a time-out at $t = 4500$ time steps. Additionally, this reward can only be received once because reaching this horizontal offset represents level completion, triggering another one of our episode boundaries.

5.1.9 Random Start

To further reduce the chance of learning a deterministic policy for completing a level, we optionally include a random start to the beginning of each episode. We pick a number of time steps t uniformly distributed between 16 and 45 (about 1 and 3 seconds respectively), and perform t random actions to start the episode. Since the beginning of each level is the most commonly observed part of the level, we hypothesize adding this randomness discourages memorizing an "opening sequence" of actions by naturally creating new starting points.

5.2 Agent Definition

There are 3 main parts to our agent definition:

1. Encoder to process an environment observation and produce a corresponding action.
2. A rollout object to keep track of and serve a batches of agent-environment experiences.
3. PPO implementation to construct loss from a batch of experiences and update agent parameters.

5.2.1 Observation Encoder

After performing the processing in sections 5.1.5 and 5.1.6, our video observation is a 320 by 224 pixel grayscale image with depth m_{video}, and our audio observation is either a 368 by 256 or 92 by 256 pixel spectrogram with depth 1.

We use a convolutional neural network (CNN) [30] with layers outlined in figure 5-4 and 5-5 to process both our video and audio observations. If we are observing both video and audio at the same time, we create separate but structurally identical CNNs for each observation.

The final output shape of 256 represents the encoded hidden state of the observation. If we are observing both video and audio, we concatenate both hidden states to create a final hidden state of size 512. This is a standard CNN architecture used in other reinforcement learning and ALE tasks that we have referenced previously.

This CNN architecture works particularly well for the Sonic games because the receptive field for the third layer is a 36 by 36 pixel square, which can handily fit the 30 pixels wide by 35 pixels tall Sonic.

Layer	**Hyper-params**	**Params**	**Out Shape**	**R. Field**
1. Conv2d	`in_cls=1, out_cls=32, kernel_size=8, stride=4`	256	[32, 55, 79]	8x8
2. ReLU	`none`	0	[32, 55, 79]	
3. Conv2d	`in_cls=32, out_cls=64, kernel_size=4, stride=2`	8192	[64, 26, 38]	20x20
4. ReLU	`none`	0	[64, 26, 38]	
5. Conv2d	`in_cls=64, out_cls=64, kernel_size=3, stride=1`	12,288	[64, 24, 36]	36x36
6. ReLU	`none`	0	[64, 24, 36]	
7. Flatten	`none`	0	[55296]	
8. Linear	`in_shp=55296, out_shp=256`	14,155,776	[256]	

Figure 5-4: CNN architecture used to process a grayscale frame of video into a hidden state of size 256.

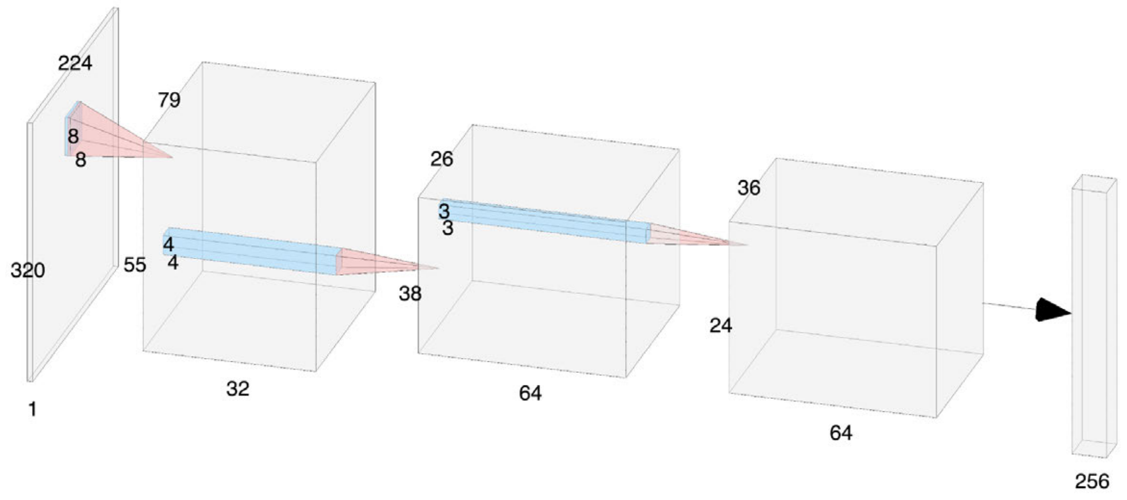


Figure 5-5: CNN architecture for the observation encoder. The input observation in this example is a grayscale frame of video.

To output a distribution over the 8 essential action combinations described in section 5.1.7, we first add a linear layer that takes in the concatenated hidden state and outputs 8 logits. Then we wrap these logits in a Categorical distribution so we can sample and update gradients accordingly. To output a value prediction, we add a second linear layer that takes in the concatenated hidden state and outputs the predicted value of this state.

5.2.2 Extension to Video Sequences

As mentioned in section 5.1.5, the additional $m_{video} - 1$ memory buffer of video data is simply appended to the depth dimension of the original grayscale image. This means we do not need to change our CNN architecture to support different values of m_{video} beyond setting `in_cls=` m_{video} for the first Conv2d layer. However, this method is unconventional relative to other techniques that use recurrent neural networks [41] to recognize temporal changes, so there is a tradeoff.

Temporal features that we wish to detect on screen must be contained within the CNN's receptive field. To demonstrate this idea, let us use a scenario where Sonic is jumping in the air, and we want to figure out if he is rising or falling. If the difference in Sonic's height between the last frame (time step $t-1$) and the current frame (time step t) is positive, this indicates an upward trajectory. For our CNN to be able to capture this idea, it might use the second Conv2d layer recognize Sonic's head in channel 1 (time step $t-1$) and his shoes in channel 0 (time step t). If both items are found within the third Conv2d layer's receptive field, a 36 by 36 pixel square, then the CNN can conclude an upward trajectory. If Sonic's shoes are 40 pixels above where his head was in the previous time step, the CNN will not be able to understand the direction of movement.

But given that the there are only 0.066 seconds between time steps, we find that most relevant on screen temporal changes do indeed occur within this receptive field. In fact, we believe that CNNs are better suited for this $m_{video} = 2$ task than a CNN and RNN combination, where a CNN encodes frames individually before feeding them in chronological order to an RNN. It would be very difficult for this combination to detect small changes between two images without encoding literal positional information with the CNN.

5.2.3 Extension to Audio Spectrograms

Audio data is sequential in nature, and so one may assume that we would use a recurrent model to process our audio observations. However, we are only using 1 second of audio at a time, and we found that the average Sonic sound effect lasted about 1 second or less. As a result, we can treat this more similar to an image detection problem. Indeed, Salamon et al. [62] use a CNN architecture to classify environmental sounds, and argue that reasonably sized receptive fields would allow for spectro-temporal patterns to naturally arise from the different sound classes.

To this end, we vary our spectrogram hop length, as described in section 4.3.3, as a method for testing the same receptive field size against different resolution spectrograms. This effectively gives the same result as maintaining hop length, and varying the receptive field size itself.

5.2.4 Experience Generation

Agents use individual observations to inform which action to take, but we need to record entire experiences in order to train with the PPO algorithm. An experience consists of an observation, the predicted value, the selected action with its underlying action log probabilities, and the reward given after the action was taken.

Since the Gym Retro environment only supports one instance of a game per process, we spawn $48 \cdot 3 = 144$ processes, each assigned to one of the 48 training levels. Each of these processes runs for 512 steps to generate 512 experiences. The total of $48 \cdot 3 \cdot 512 = 73,728$ experiences are aggregated with a batch sampler and subset random sampler, with a mini-batch size of 72. Therefore, our total batch size for each update is 73,728, and gradients are accumulated using the PPO algorithm on our GPU, in 72 mini-batch updates of size 1024 each. The gradient update is only applied for each PPO epoch after all mini-batch gradients have been accumulated.

Experience Gen. Hyper-params	**Value**
Workers Per Level	3
Horizon	512
Batch Size	73,728
Minibatch Size	1024
GPU	Tesla V-100
Pytorch Version	1.4.0
Gym Retro Version	0.8.0

Figure 5-6: Experience Generation Hyper-parameters.

To avoid memory overflow on the GPU, we keep all experiences on the CPU except for the current mini-batch (figure 5-6).

Note that our original implementation did not have processes dedicated to single levels. After a process's level reached a boundary condition, we re-initialized the process with a training level selected uniformly at random, to ensure episodes were split evenly across levels. However, we found that this created a negative feedback loop, where good performance on a level meant that the agent would be less likely to die on the level, spend more time, accumulate more reward, and continue to overfit its policy to that level. Conversely, poor performance on a level meant that the agent would be more likely to die on the level, spend less time, accumulate less reward, and underfit its policy for that level.

By assigning a single process to a single level, we ensure that the number of experiences are split evenly across all levels, rather than the number of resets or episodes.

5.2.5 PPO Implementation

As described in section 2.3.4, PPO is a policy gradient algorithm that has performed well on many reinforcement learning tasks, including the ALE. Since CNNs are constructed with gradient information flowing all the way from generated experiences to predicted actions and values, optimizing the PPO objective necessarily updates our CNN parameters. We use Pytorch for our PPO implementation [55] [33], and set the hyper-parameters as described in figure 5-7.

PPO Hyper-params	**Value**
Epochs	4
Discount Factor γ	0.99
GAE Parameter λ	0.95
Clipping Parameter ϵ	0.2
Entropy Coefficient	0.001
Reward Scale	0.005

Figure 5-7: PPO Hyper-parameters.

The reward scale is a small constant factor applied to the reward, and used to bring advantages into a range better suited for neural networks.

5.3 Evaluation Methods

We use both quantitative and qualitative metrics to evaluate agent performance. Quantitatively, we use a variant of mean scoring, graph the performance over training time, and monitor gradients and action entropy. Qualitatively, we watch generated videos of trained agent play, and augment these videos with a CNN weight visualization technique called Score CAM [79] [52].

5.3.1 Mean Scoring

We use a variant of a previously established method of mean scoring [47] to evaluate the performance of our trained agents on the Sonic retro environment. The procedure for calculating this score is as follows:

1. Train the agent for any number of steps.

2. Run the agent in evaluation mode for each of the test levels, for 15,000 time steps each.

3. Average the total reward and compute the standard deviation (stdev) over all episodes for each test level separately, giving a per-level mean score and error margin.

4. Average the per-level mean score and compute the stdev over all test levels, giving a final performance metric and error margin.

We consider a few other factors to get a wholistic idea of model performance. A steady decrease in action distribution entropy indicates that the agent is converging. A steady increase in mean score indicates that the agent is progressively learning techniques that build on each other.

5.3.2 Augmented Video Playback

The offline playback method described in section 4.2 allows us to view and hear episode runs generated by the agent, but it lacks other information needed for critical evaluation. At each frame, we augment the generated video with the following information: level name, step number, current horizontal offset, max horizontal offset, percent of level completed, accumulated reward, predicted value of current state, selected action, and log scaled action probabilities.

To understand where in the video the agent is attending to inform its current action decision, we implement Score CAM. At a high level, Score CAM is an algorithm that takes in the current observation and CNN layer weights, and outputs a heat map over the observation, showing where the CNN is choosing to "look". More implementation details of this process can be found in the original Score CAM paper [79].

5.4 Task Outlines

To test the effect of audio on videogame RL, we set up 3 different tasks.

1. Individual PPO Training. We train a separate agent on each level individually, for 1 million time steps. Then we calculate a per-level mean score for each trained agent. In this setting, we expect that it would be easier for an agent to memorize the correct way to progress through a level.

Game	**Zone**	**Act**
Sonic The Hedgehog	Green Hill Zone	2
Sonic The Hedgehog	Scrap Brain Zone	1
Sonic The Hedgehog	Spring Yard Zone	1
Sonic The Hedgehog	Star Light Zone	3
Sonic The Hedgehog 2	Casino Night Zone	2
Sonic The Hedgehog 2	Hill Top Zone	2
Sonic The Hedgehog 2	Metropolis Zone	3
Sonic 3 & Knuckles	Angel Island Zone	2
Sonic 3 & Knuckles	Flying Battery Zone	2
Sonic 3 & Knuckles	Hydrocity Zone	1
Sonic 3 & Knuckles	Lava Reef Zone	1

Figure 5-8: Test Set Levels. These were selected by first randomly choosing Zones with more than 1 level, and then randomly picking an act from each selected zone [47]. This ensures that an agent trained jointly is already familiar with the textures and music in each test level.

2. Joint PPO Training. We train a single agent on 47 of the 58 levels, which we call the training levels. The remaining 11 testing levels are listed in figure 5-8. This agent trains for 30 million time steps, and we self-evaluate the agent on the training levels every 5 million steps. It is more difficult to memorize trajectories for all 48 levels, so in this setting we hope that the model learns general principles for level progression.

3. Zero-shot Transfer. We take the jointly trained agent from the previous experiment and evaluate it on the 11 test set levels, without any fine tuning. If the agent learned general techniques for level progression in experiment 2, it should be able to perform better than a random agent, or one following a simple policy. Ideally, it would perform similar to agents trained in experiment 1.

We train 3 of each agent variant (defined in the next section) on each of these three tasks and use the mean scoring method above to consolidate performance.

5.5 Agent Outlines

We define and train 5 different agents on the tasks above and compare relative performance to understand the effect of audio features in playing Sonic. We also compare performance to simple policy and human baselines.

5.5.1 "Hold Right" Baseline

By definition of the reward function we have defined in section 5.1.8, Sonic accumulates reward as he makes net progress towards the right. We can therefore define a simple yet effective policy that requires no machine learning: always make Sonic move right. We evaluate this policy and set it as a lower bound for the performance we expect our trained agent to achieve.

5.5.2 Human Baseline

There is an existing human baseline for the Sonic Genesis games. In this baseline, 4 test subjects practiced on the 47 Sonic training levels for 2 hours, before playing each of the 11 test levels over the course of an hour.

5.5.3 Agent Variants

The 5 agent variants are described in figure 5.1. Our reasoning for defining these specific variants is as follows.

Agent 1 is the same as the one used in the Gotta Learn Fast [47] benchmark, albeit with fewer trainable parameters and shorter training time due to computational constraints.

We introduce random start and grayscaling with Agent 2, which we expect will result in improvements to generalization.

Agent 3 features larger model size and $m_{video} = 2$, giving it an additional video frame from the previous time step. We hypothesize that Agent 3 may be able to learn temporal features, and therefore achieve better results than Agent 2.

Agent	**Video Frames**	**Random Start**	**Grayscale**	**Hidden Size**	**Audio (W, H)**	**Num Params**
1	1	no	no	256	none	14m
2	1	yes	yes	256	none	14m
3	2	yes	yes	512	none	28m
4	1	yes	yes	$2 \cdot 256$	(256, 128)	31m
5	1	yes	yes	$2 \cdot 256$	(1024, 512)	18m

Table 5.1: Agent Variants.

We train Agent 4 and Agent 5 with audio and compare against Agent 2 and Agent 3, to quantify the effect of learned audio features across varying model size and video information access.

Chapter 6

Results

In this chapter, we present the results from our experiments across 3 different tasks, and 5 different agent variants. We then qualitatively and quantitatively analyze the effect of using audio to inform sequential decision making in videogames.

6.1 Overview

Figure 6.1 shows us the most important result of this thesis, which is that audio+video agents outperform video-only agents on the joint training task, and achieve higher scores on the zero-shot transfer task. Agent 5, which is provided with the current frame of video and past 1 second of audio, outperforms Agent 3, which is provided with the current and previous frames of video, no audio, and 55% larger model size, by 6.6% on the joint task, and 20.4% on the zero-shot task. This result supports our hypothesis that Sonic game audio informs sequential decision making, and extracted audio features are more easily transferable to unseen test levels than video features.

Sonic game audio is mostly supplementary to the visual information on screen, which explains the lesser performance of audio agents on the solo train task. The goal for agents in this task is to memorize the level as fast as possible, and audio data hinders agents' ability to overfit to levels quickly, especially with the added random start to each level.

The "Final Avg" table contains the primary scores used to evaluate overall agent

Agent #) Desc.	**Joint Final Avg**	**Zero-Shot Final Avg**	**Solo Avg**
1) 1V 256 n.p.	1344.6 ± 206.4	435.6 ± 130.0	$\mathbf{2477.5 \pm 2482.6}$
2) 1V 256	1479.5 ± 911.3	578.5 ± 409.7	1389.6 ± 1040.0
3) 2V 512	1905.0 ± 286.9	678.5 ± 238.3	2020.1 ± 1361.9
4) 1VA $2 \cdot 256$ 128H	1893.7 ± 225.3	817.3 ± 320.2	1876.5 ± 979.4
5) 1VA $2 \cdot 256$ 512H	$\mathbf{2031.1 \pm 501.9}$	$\mathbf{936.6 \pm 220.8}$	1617.9 ± 2051.7

Agent #) Desc.	**Joint Best Ckpts**	**Zero-Shot Best Ckpts**	**Solo Best Ckpts**
1) 1V 256 n.p.	1780.2 ± 1708.5	755.1 ± 1028.1	3207.6 ± 2752.4
2) 1V 256	2998.6 ± 2010.63	1526.8 ± 1257.4	2601.8 ± 2302.9
3) 2V 512	3013.0 ± 2186.2	1686.9 ± 1155.1	$\mathbf{3304.6 \pm 2443.9}$
4) 1VA $2 \cdot 256$ 128H	$\mathbf{3041.4 \pm 2227.9}$	$\mathbf{1779.2 \pm 1403.1}$	2870.5 ± 1805.8
5) 1VA $2 \cdot 256$ 512H	2901.5 ± 2011.1	1756.9 ± 1298.3	2300.7 ± 2669.9

Baseline	**Joint Final Avg**	**Zero-Shot Final Avg**	**Solo Avg**
Hold Right	1099.1 ± 1092.8	321.9 ± 277.5	321.9 ± 277.5
Human [47]	–	7438.2 ± 1126.0	–
Gotta Learn Fast [47]	5083.6 ± 91.8	~ 1000	1755.1 ± 65.2

Table 6.1: Performance summary of all 5 agent variants and 3 baselines over all 3 tasks.

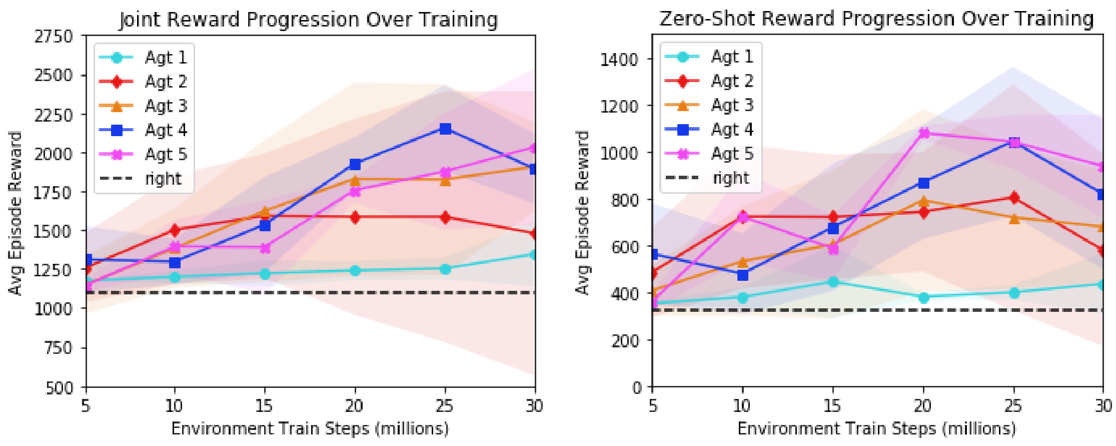


Figure 6-1: Progression over train time, evaluated every 5 million steps. Left: Progression of joint score. Right: Progression of zero-shot transfer score.

performance. Final refers to the fact that these scores were computed by evaluating the final saved checkpoints of each agent, rather than averaging over the entire training run. A visual progression of joint and transfer scores over train time is given by figure 6-1.

We add a "Best Ckpts" table, which aggregates the top per-level mean scores across all corresponding agent checkpoints. We trained each agent 3 times for 30 million time steps, saving and evaluating checkpoints every 5 million time steps, which gives us a total of 18 checkpoints per agent. The goal of adding this table is to demonstrate each agent's best possible effort for each level. A smaller difference between values in the "Best Ckpts" table and "Final Avg" table might indicate greater robustness and/or less tendency for that agent to overfit.

Scores for each agent are presented as the mean ± the first standard deviation taken along the individual level averages. See the appendix for the full details of agent performance on each level.

In the following gameplay figures with Score CAM heat maps, hot pink represents important areas that inform the agent's decision making. This is followed in descending order of importance by blue, green, yellow, orange, and finally red.

We will now proceed to evaluate each baseline and agent variant.

6.2 Baselines

6.2.1 "Hold Right" Baseline

Although our test set levels were selected in a random fashion, we find that these levels are significantly harder than the train levels for our "hold right" policy agent to progress through. Indeed, this agent is unable to exceed a score of 1000 on any of the test levels, but achieves an average score of 1084.5 on the training set.

There are 3 levels where our "hold right" policy agent can achieve a score over 3000, or one-third progress through the level, and 6 levels where this agent cannot muster even a score of 90, or 1% progress through the level (see appendix).

6.2.2 Human Baseline

As expected, humans achieve very high scores on the test set. They were able to complete at least 50% of each level, with the average performance resting around 80%. It is important to note however that even humans are only able to finish 3 of the 11 test levels with consistency (see appendix).

6.2.3 Gotta Learn Fast

This baseline is not directly comparable to our results, because they perform joint training with 4 threads per level instead of 3, horizon size 8192 per thread instead of 512, total train steps 400 million instead of 30 million, and hidden layer size 512 with RGB and no random start. However, it is still interesting to note the high ceiling for (likely memorizing) agent joint scores, and the relatively low zero-shot and solo averages in comparison.

6.3 Video-Only Agents

Although the focus of this work is on analyzing learned audio features, our video-only agents learn a number of interesting visual features which are important for putting

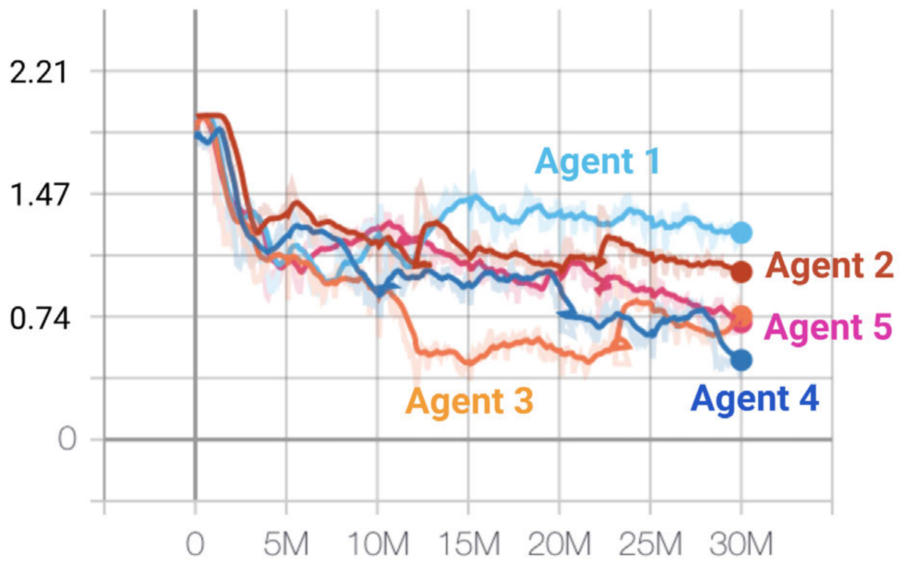


Figure 6-2: Action entropy over train time.

our learned audio features in context.

6.3.1 Agent 1

Agent 1 uses 1 frame of video and hidden layer size 256, without no grayscaling or random start. It is structurally the same as the agent described in the Gotta Learn Fast benchmark. Due to computational constraints, however, we use half the hidden size and train the joint model for 7.5% the number of timesteps, with 4.8% the batch size.

As a result, Agent 1 heavily underfits during joint training and performs poorly on the zero-shot evaluation. In figure 6-2, we see that this is quantitatively supported by the fact that Agent 1 maintains the highest action distribution entropy, and is only able to outperform the "hold right" policy by a small margin. Qualitatively, we see in figure 6-3 that Agent 1 is unable to progress beyond simple obstacles.

On the other hand, Agent 1 achieves the highest score on the solo train task, beating the Gotta Learn Fast PPO baseline by a substantial amount. The 41% increase in solo train task performance over this baseline suggests that when only training on 1 level, reduced model size allows our agent to memorize and converge faster. The

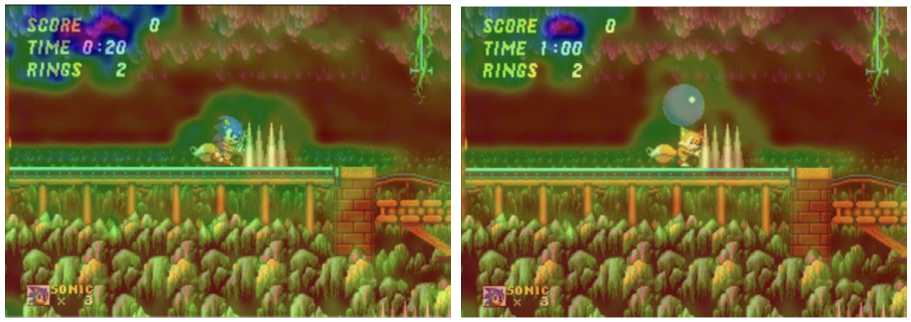


Figure 6-3: In the joint train task, Agent 1 manages to both underfit and clock watch. Left: Sonic waits in front of a spike, with an high entropy action distribution. Right: Sonic jumps over, 60 seconds in, because the agent has learned to associate "1:00" on the clock with jumping.

action distributions for the solo task converge almost an order of magnitude lower than the joint models, which shows that the agents are memorizing predetermined paths (figure B-2).

Remember that Agent 1 does not have access to temporal data, so it must be finding alternate ways of memorizing sequences of moves. We call one such technique "clock watching". In the top left corner of the screen, a timer indicates how long the agent has been playing on the current level. An agent can begin to memorize a sequence by learning to recognize time and assigning a move to each second.

Eventually, the agent will learn other level-specific tricks that supplement clock watching and lead to both higher efficiency and more quirky behavior. In Angel Island Zone Act 2, Agent 1 uses a combination of clock watching and the falling bricks of a bridge to perform a frame perfect execution of the spin dash attack, which requires 3 successive button presses (figure 6-4). In Green Hill Zone Act 2, Agent 1 learns a technique for jumping through vertical loops, which is faster than running through them.

6.3.2 Agent 2

Agent 2 uses grayscaling instead of RGB, and adds a random start between 1 and 3 seconds to the Agent 1 setup. Adding the random start makes it much more difficult for Agent 2 to memorize a high reward path through the each level, so this results in

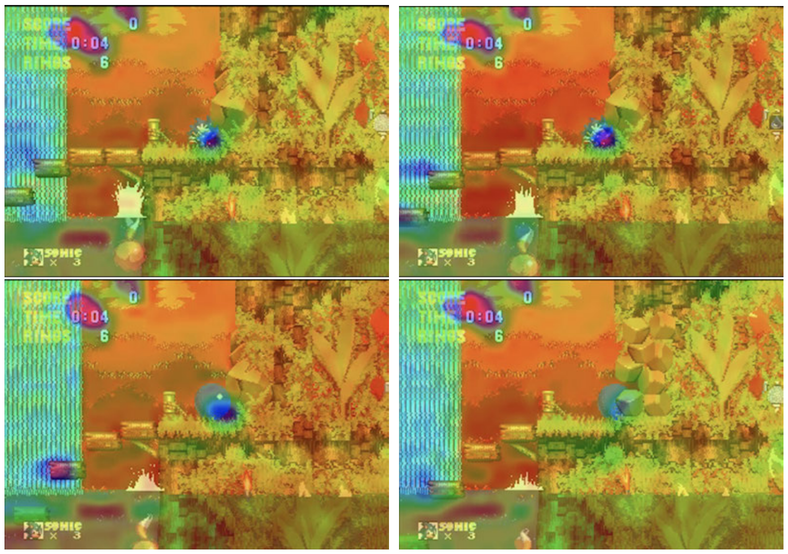


Figure 6-4: In the solo train task, Agent 1 uses falling bricks to input a frame perfect spin dash and break the wall blocking its path. Top Left: Sonic is hunched over and the agent waits for the leftmost brick to fall. Agent keeps pressing down on the D pad. Top Right: Leftmost brick falls far enough. Agent presses B to begin charging spin. Bottom Left: Sonic is in a spinning ball, and the brick second from the left is falling. Agent presses right on the D pad to start releasing the spin dash. Bottom right: Sonic begins to spin dash through the wall.

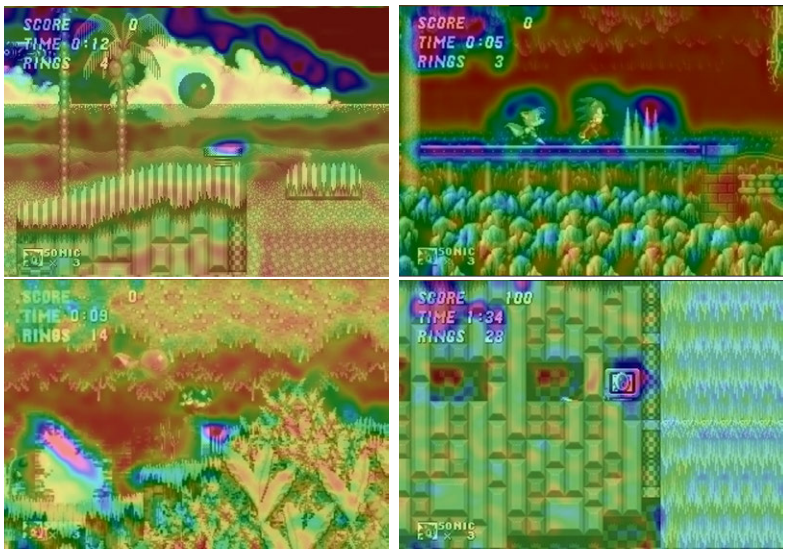


Figure 6-5: Agent 2 detects various objects during training. Top Left: the agent jumps on a spring to launch Sonic over the pit of spikes. Top Right: the agent recognizes that Sonic needs to jump over the spikes. Bottom Left: an example of a sideways spring that hampers progress by pushing Sonic back to the left. Bottom Right: a powerup TV that gives an extra life.

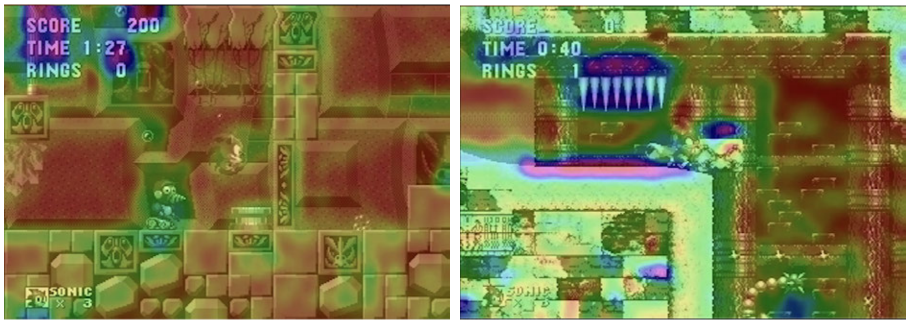


Figure 6-6: Agent 2 performs multi-step actions. Left: Sonic presses a button which opens the door. Right: Sonic pushes a box out of the way while avoiding jumping into the spikes above.

a 44% decrease in solo train score from Agent 1.

At the same time, the random start substantially improves the joint and zero-shot tasks. This is because increased starting state diversity forces Agent 2 to learn more general features for level progression. If the agent learns that jumping over spikes is good in one level, the the lower layers of the CNN that were used to recognize the spikes will help to transfer that knowledge to other levels. This is in contrast to clock watching, where different levels will compete for seconds on the clock to result in actions that achieve greater reward for them.

Some examples of commonly recognized objects are power ups, springs, spikes, and Badniks (figure 6-5). Fundamental improvements are made in a number of different levels relative to Agent 1, and qualitatively we can see Agent 2 is able to perform non-trivial multi-step movements such as pressing buttons to open doors, or pushing blocks out of the way (figure 6-6).

Agent 2 converges to a lower entropy action distribution than Agent 1 and raises the joint Best Ckpts score by 68%, and the zero-shot Best Ckpts by 102%. The relatively large difference between Agent 2's Final and Best Ckpt scores is explained by the fact that there was large variance in performance between the three runs. This might indicate that Agent 2 is not a particularly robust agent for training.

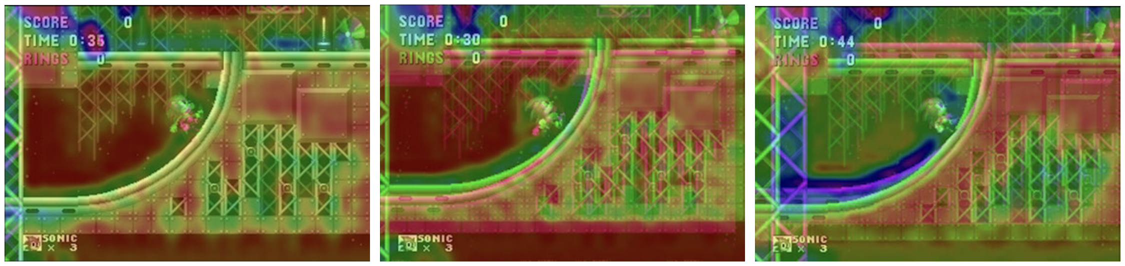


Figure 6-7: Agent 3 uses temporal information to judge its progression up the ramp. The two screenshots on the left are moments when the agent decides it does not have enough momentum, and backtracks to the left. The pink and dark blue ScoreCAM weighting in the screenshot on the right shows the agent correctly understands that the screen shifting upwards means that Sonic has enough momentum to make it up the ramp.

6.3.3 Agent 3

Agent 3 builds upon Agent 2 by doubling the hidden layer size to 512 and increasing the number of video frame observations from 1 to 2. Making everything bigger makes it easier to overfit, which is shown by the 45% increase in solo train score over Agent 2, and record for solo Best Ckpts score.

By including the video frame from the previous time step, Agent 3 now has the ability to learn temporal meta-variables, such as relative velocity of objects on screen. Agent 3 learns to use this information to determine, for example, when Sonic does not have enough momentum to go up a steep ramp, and must backtrack in order to build the momentum needed to try again (figure 6-7).

Agent 3 improves over the Agent 2 joint train and zero-shot scores by 29% and 17%, and also provides lower variance results. In figure B-1, we see it reaches higher scores at a faster rate than other video-only agents on the levels Star Light 2, Emerald Hill 1, Chemical Plant 1, and Mushroom Hill 2.

However, a qualitative look shows that Agent 3 is also learning to overfit to critical frames of action in these levels. In Star Light 2, Agent 3 is exposed in a run where a miscalculation of a jump throws the agent off of its memorized path, causing it to get stuck at a simple obstacle (figure 6-8). Over the training steps, its action entropy reaches lower levels much faster than other agents, and its zero-shot improvement is relatively low given the magnitude of joint train improvement.

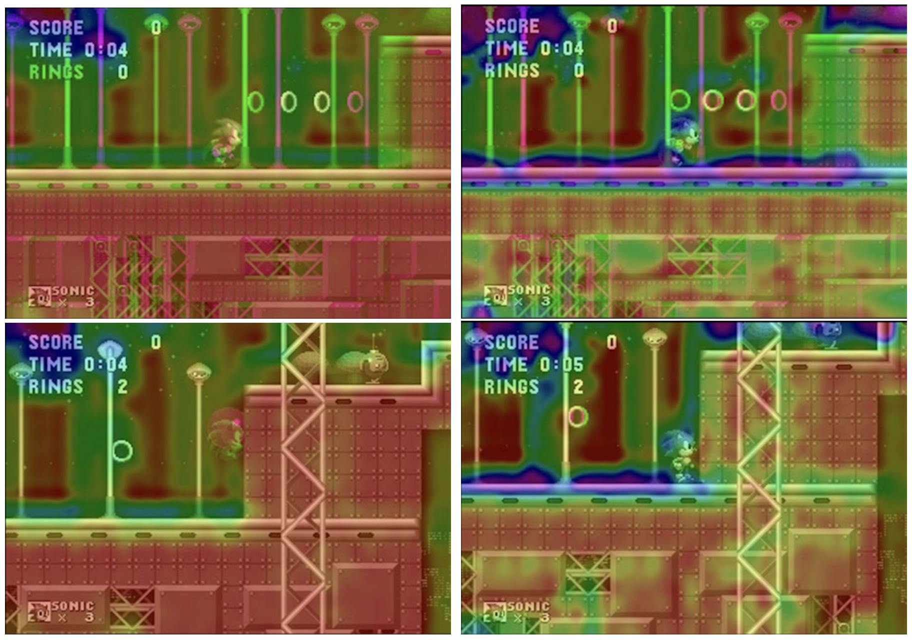


Figure 6-8: Agent 3 learns to memorize parts of Star Light 2, which backfires when it misses a jump. Top Left: Agent runs to the right. Top Right: CNN activations indicate it is time to jump. Bottom Left: Sonic does not reach the top of the platform. Bottom Right: CNN activations indicate it is time to jump again, but Sonic is already holding down B from the previous jump. He is stuck.

6.4 Audio+Video Agents

Instead of continuing to build off of Agent 3, Agents 4 and 5 replace the second video frame with a spectrogram representing the last 1 second of in-game audio. Separate CNNs are used to process the video and audio observations, each with hidden size 256, and concatenated together into a final hidden state size 512.

6.4.1 Relative Receptive Field Comparison

Agent 4 constructs a spectrogram with window size 256 and hop length 128, while Agent 5 uses window size 1024 and hop length 512. This leads to spectrogram sizes 368 by 256 pixels and 92 by 256 pixels, respectively. Both convey the same amount of information, but result in audio CNNs with different numbers of trainable parameters (Agent 4 has about 14 million more than Agent 5) and different relative receptive field sizes (Agent 4 has a 4 times smaller relative receptive field). In other words, Agent 4

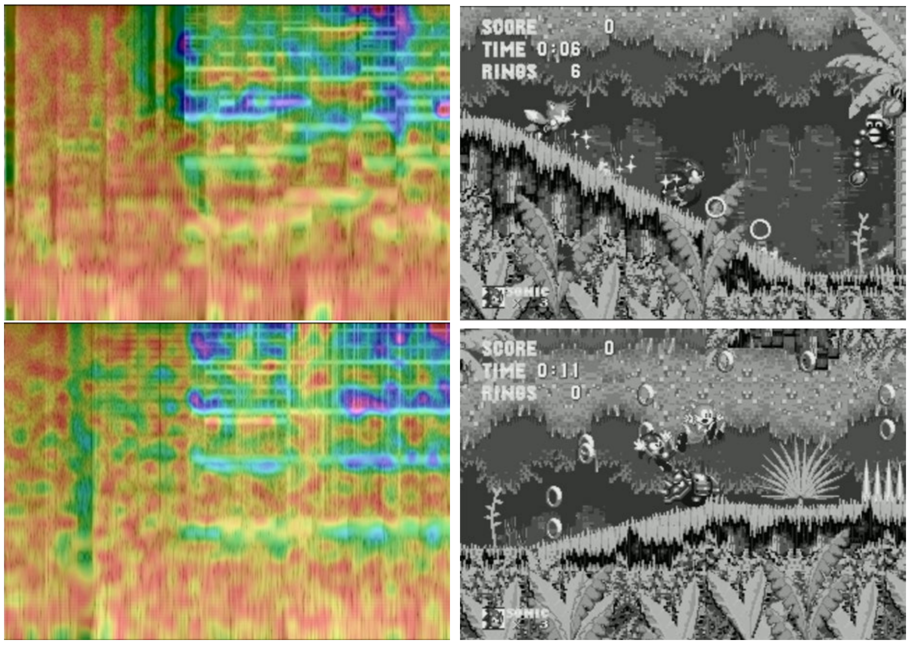


Figure 6-9: Agent 4 listens to ring sounds and gives increased probability to moving right. The smaller relative receptive field of Agent 4 makes it so that the CNN activations are not as coherent. It picks up on subsections of the losing rings sound, and is therefore more susceptible to misidentifying a different sound effect with similar components.

learns a larger number of smaller audio details, and Agent 5 learns a smaller number of larger ones.

In general, we found that Agent 5 was able to learn higher level audio features than Agent 4, which was more prone to noise and overfitting. Quantitatively, this is shown by the fact that Agent 4 reaches a lower action entropy than Agent 5, scores lower in both the joint and zero-shot final scores, and uses similar tricks as Agent 3 to achieve higher scores on certain levels. Qualitatively, see see that the Score CAM weights over a spectrogram with ring sounds are patchy and disconnected for Agent 4 (figure 6-9).

6.4.2 Learned Audio Features

In general, we find that audio features learned by these agents fall into three of the four IEZA framework categories, and that there is a similar agent response to audio features in the same category.

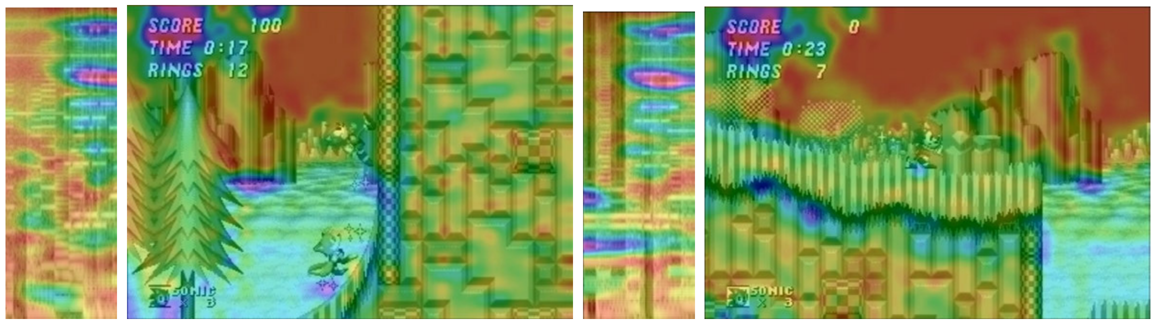


Figure 6-10: In Hill Top 2, during a zero-shot run, Agent 5 looks at ring sound as motivation to move right. Left: The agent acquires rings while running up a steep cliff. Right: The agent continues to run into the rock blocking its path until the ring sound is freed from its memory.

- Most learned audio features fall into the Effect category. These are diegetic sounds that inform in game activity, like the sound effects of acquiring or losing rings. Rings are small, keep rotating, and turn into even smaller stars when grabbed by Sonic, so the act of acquiring them is not recognized by the agent's visual CNN component. The corresponding sound is easy to see in a spectrogram, so it is readily learned by the audio CNN component. The same is true for when Sonic loses his rings.

  An agent will respond to the ring acquiring sound by increasing its likelihood of taking the "move right" action, since rings are generally located along paths that lead to forward progress in the level. Figure 6-10 shows an example of Agent 5 doing this on a zero-shot run of Hill Top 2. Ironically, losing rings also produces the effect of increased "move right" likelihood, despite having a dissonant sound, negative connotation for humans, and perceived negative impact on gameplay. This is because Sonic is granted temporary invincibility after losing his rings, so the agent uses this as an opportunity to get past the danger that was originally in the way.

- The drowning theme and underwater pings are audio features that fall into the Interface category. Recall that Interface sounds are non-diegetic sounds that inform in game activity. As shown in figure 4-6, visual indicators that Sonic is about to drown are small see through bubbles with numbers that do not appear

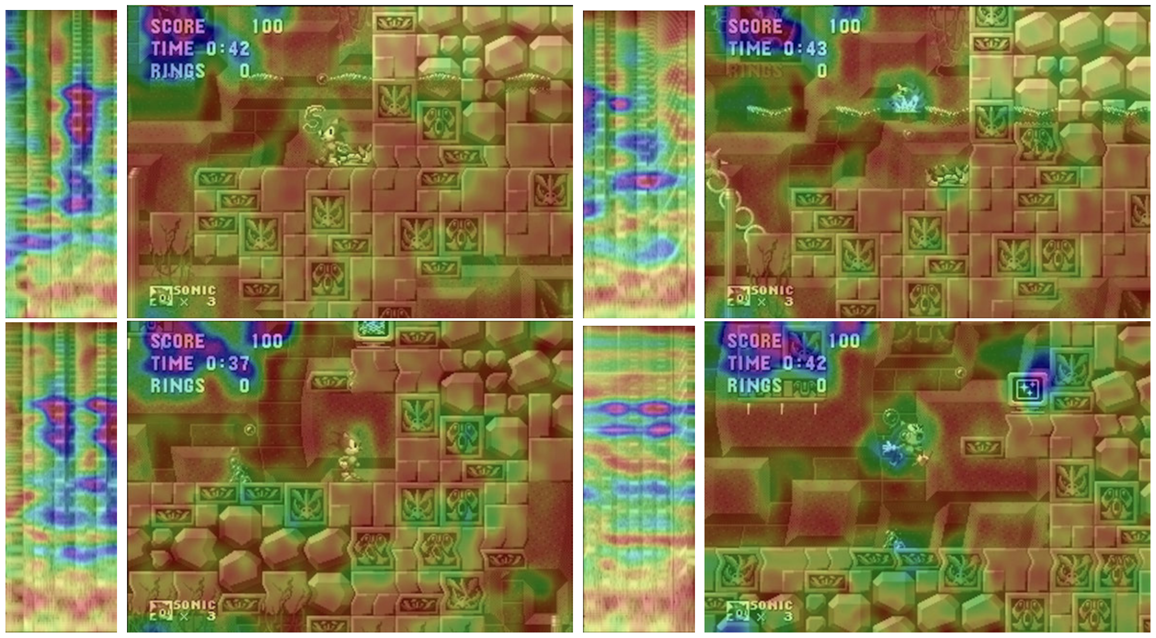


Figure 6-11: Agent 5 handles two cases of drowning. Top: problem is solved by simply jumping up and getting air from above the water. The glow around Sonic shows that the model understands he has broken up to the surface to breathe the air, and is free to return below. Bottom: agent navigates Sonic to a bubble stream on the left, and waits for a bubble to appear. The model highlights Sonic as he breathes in the air bubble.

every frame, and are easily missed by the visual CNN. Upon hearing this theme via the audio CNN, our agents learn to quickly locate an oxygen-restoring air bubble or exit the water entirely to avoid losing a life (figure 6-11). This call to action of the drowning theme is in sharp contrast to the laid back and suggestive nature of the Effect sounds.

Before Sonic gets to the drowning stage, there are pings that play every 5 seconds Sonic is underwater. In figure 6-12 we see that Agent 5 remembers to exit the water after hearing the first one of these pings, in a zero-shot run of Angel Island 2.

- Surprisingly, a significant number of sounds highlighted by the Score CAM algorithm come from the backing Zone theme, which falls into the Affect category. Recall that Affect sounds are non-diegetic sounds that inform in game setting. It is not immediately clear what use the Zone theme might have for completing

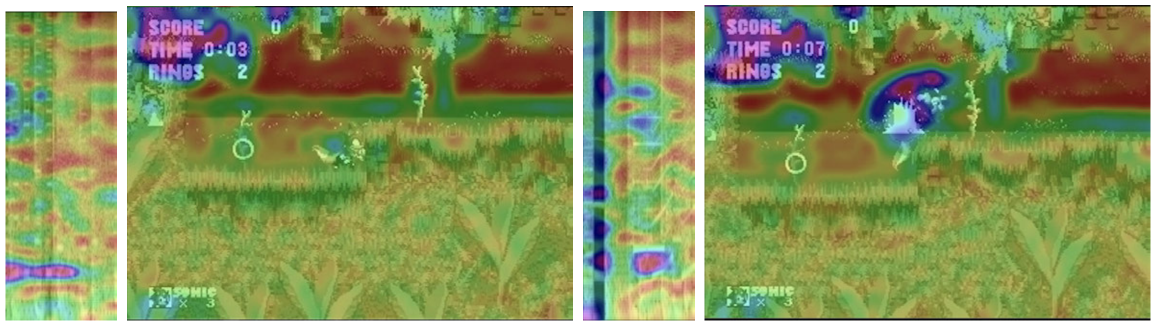


Figure 6-12: Agent 5 gets out of the water after hearing a ping. Left: Sonic is holding down right on the D pad. Right: Sonic presses B to jump, and the spectrogram shows that this action is informed by the ping sound, highlighted in pink near the bottom middle.

levels.

We hypothesize that these learned features are mainly an artifact of the audio CNN incorrectly attributing accumulated reward to the part of the theme that happened to be playing at the time. Typically these features do not seem to have a significant effect on agent actions, but sometimes they are coincidentally reinforced. For example, in figure 6-13, we see that a note in the Emerald Hill Zone theme sometimes prompts Sonic to jump. At the next obstacle, the agent chooses to wait for this note to loop back and be played again, rather than understand that it needs to jump over the spike blocking its path. This reinforces the positive reward associated with that note, and creates a unique style of periodic overfitting.

These features are occasionally also useful when Sonic gets stuck in a visually "quiet" part of the level, with minimal on-screen movement. In this case, the main variance in agent hidden state comes from the audio, and so a unique note or instrument can come along to "inspire" the agent to take a new action. This too can end up becoming a self fulfilling prophecy of periodic overfitting.

6.4.3 Not Learned Audio Features

There are a few sound effects that we expected to be more important for our training tasks. We thought that the jump sound effect would be an important temporal

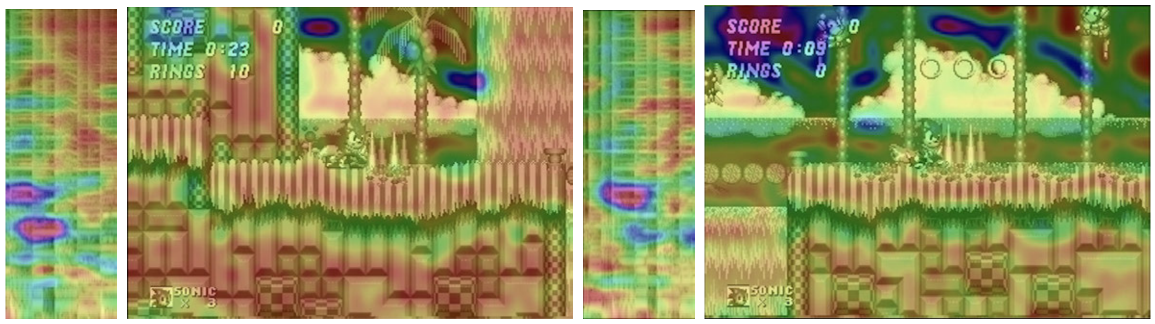


Figure 6-13: Agent 5 attends to a specific part of the music when deciding to jump, instead of the spike directly blocking its path. We see that this is a kind of periodic activation that loops with the music, as the agent waits for the music to repeat when it reaches the second spike. It is likely an artifact of attributing reward gained from jumping over the spike to the music instead of the spike itself.

indicator of Sonic's jump trajectory, i.e. determining if he is rising or falling. We also expected the sound effect produced by defeating a Badnik to be important for learning to defeat enemies before getting hit and losing a life. Figure 6-14 shows how these sounds are often paid no attention by the audio CNN.

We conclude that these features were not learned because in Sonic 2 and Sonic 3&K, Tails follows Sonic around and produces many of the same sound effects while autonomously interacting with his local environment. This means that the jump sound or defeating a Badnik sound is often not attributed to Sonic, and therefore loses its predictive power.

Our agents also did not learn to differentiate between consonant and dissonant sounds, or along any other axis of Western classical music theory explained in section 3.2. This was expected, since the sounds in the Sonic games are not plentiful or diverse enough to be able to learn concepts beyond high level sound effect identification. These theories should play a greater role if/when a large unsupervised audio model is trained on massive amounts of Western audio data and fine tuned on videogame RL tasks, similar to how BERT [14] and GPT-3 [76] have transformed the field of NLP.

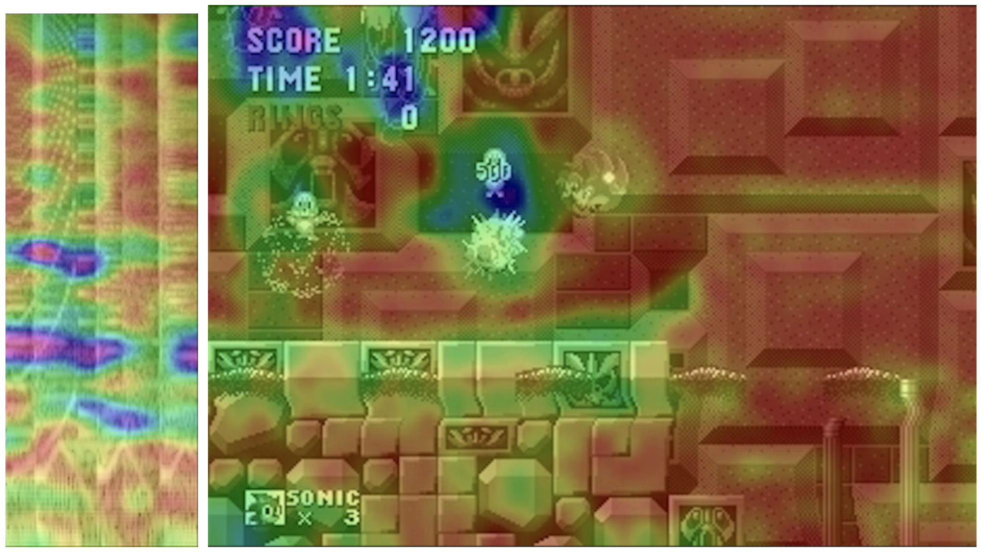


Figure 6-14: Agent 5 does not value the defeating Badnik or jump sound effects. Left: the three white peaks near the bottom of the spectrogram come from defeating 3 Badniks in a row. The leftmost peak temporally overlaps with the jump sound effect, which are the upward curving lines above it. Neither of these are considered to be very important for the audio CNN. Right: Sonic has just finished attacking the third Badnik.

6.4.4 Learned Video Features

None of this detailed audio analysis is to say that our audio agents have weaker visual components. In fact, in figure 6-15 we see that Agent 5 is capable of navigating multiple multi-step visual challenges with what appears to be limited overfitting. This suggests that the audio component can be purely supplemental to the video component, although as mentioned previously, the agent may confuse itself in compounding ways when it chooses to assign reward responsibility to the wrong component.

6.5 Further Takeaways

Since Sonic sound effects are visually the same in all of the levels, and therefore pass through the same frequency range on a spectrogram, learned audio features are naturally transferable. This is supported by the fact that agents with audio features achieve the best results on the joint and transfer tasks.

Visually learned obstacles or hazards were harder to transfer, often because they

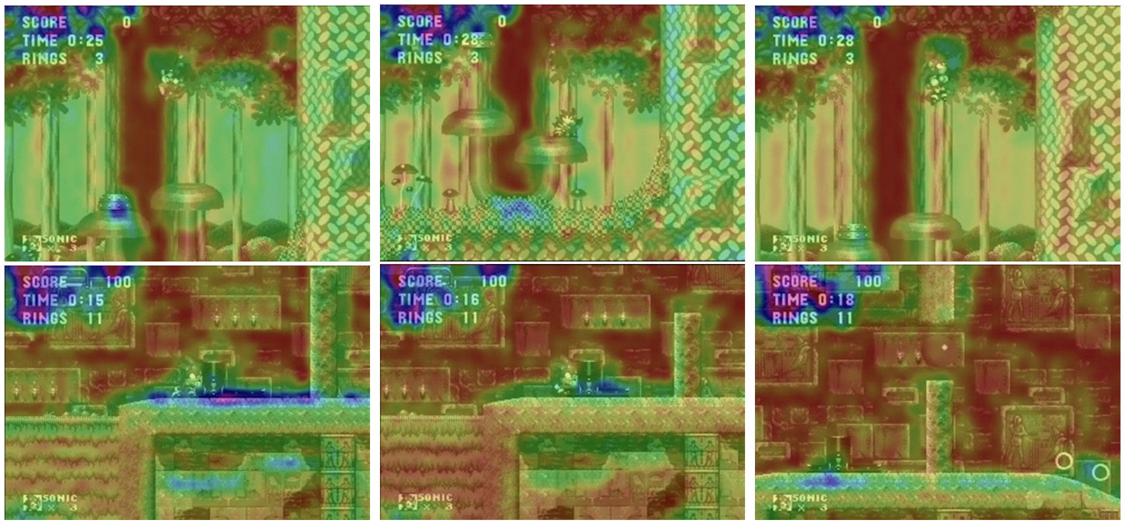


Figure 6-15: Agent 5 learns to perform visually complicated multi-step procedures. Top Row: from left to right, the agent 1) identifies the location of a weight on a seesaw-like pair of mushrooms 2) lands on the mushroom on the right, sending the weight flying in the air 3) is launched into the air once the weight returns to the ground. Bottom Row: the agent 1) begins pushing the black device to the right 2) sees that this has lowered the wall on the right 3) leaves the stone and jumps through the opening.

would be learned as objects located at specific positions on screen at certain points in time, rather than generally relative to Sonic. Results from Gotta Learn Fast [47] suggest that a few shot learning objective is needed to fully capture the usefulness of the lower-level visual CNN layers that learn to recognize objects.

Chapter 7

Conclusion

In this thesis, we provide three main contributions to videogame RL research.

First, we analyze videogame audio from a theoretical lens. We use the IEZA framework to categorize Sonic sounds as diegetic or non-diegetic, and either relating to in game activity or in game setting. This helps to facilitate discussion on what sounds we expect and find to be important for completing levels in *Sonic the Hedgehog*. We also provide a brief overview of Western classical music theory, introducing the core ideas behind consonance, dissonance, chords, instrumentation, genre, and more. This allows us to understand the reasoning behind various Sonic sound design decisions, and how sounds are crafted to enhance and ease the videogame playing experience for humans.

Second, we provide methods for extraction, playback, and processing of videogame audio, and explain the core concepts behind these methods. We use FFmpeg to combine video and audio data into a single file for offline playback, and PyAudio for online playback with smoothing via dynamic resampling. To process the audio, we use Mel Spectrograms, and explain the procedure of identifying reasonable values for each corresponding hyper-parameter.

Third, we construct an experimental setup for testing the effect of videogame audio on sequential decision making, and analyze the results. For our environment setup, we explain our use of save states, episode boundaries, frame skips, memory buffers, observation processing, random elements, and reward function. We define agents with a CNN architecture for processing observations, describe the experience generation process, and specify our policy gradient algorithm hyper-parameters. We outline our three training tasks, five agent variants, and evaluation methods. Finally, we analyze the differences in performance between audio+video and video-only agents, both qualitatively and quantitatively.

7.0.1 Closing Remarks

We show that audio+video agents outperform video-only agents on both the joint training task, zero-shot transfer tasks. We conclude that Sonic game audio informs useful decision making, and that audio features are more easily transferable to unseen test levels than video features.

The diverse nature of our learned audio features supports some of our original hypotheses, namely that environmental audio can reinforce existing visual ideas (i.e. hearing the acquiring ring sound and increasing the likelihood of moving right, while already moving towards the right), and call attention to cues that are simply non visual (i.e. hearing the drowning theme and searching for air), or missed by the visual component (i.e. Zone theme artifacts coming to aid when obstacles are not recognized). We can expect that the supplementary role of our learned audio features, combined with their natural transfer ability, would extend beyond our work on *Sonic The Hedgehog*.

As video games become increasingly realistic, and videogame RL research continues to improve, we expect that agents with the ability to process both audio and video will be the ones to achieve state of the art results on these benchmarks. We also expect that agents with the ability to understand environmental sounds will be important for other machine learning related fields, such as self driving car research. For future work, it would be interesting to see 1) how audio features transfer across different games (say, Sonic and Mario), 2) how state of the art ALE models such as Agent 57 might perform when given access to audio and games with richer audio/video components, and 3) how audio might speed up the few-shot transfer learning process, because we gotta go fast(er) [77].

Overall, we hope that this work provides a useful introduction to audio theory, processing, and application to multi-modal videogame reinforcement learning, and motivates further such research in this field.

Appendix A

Tables

Move	**Description**
Walk	Push left or right on the D-pad to initiate Sonic's movement in either direction. As the player holds the button down, Sonic gains speed.
Run	Begin walking and hold down the button to make Sonic gain speed. After a few seconds, he'll break into a run.
Screech Halt	While running, quickly press and hold the opposite direction on the D-pad to make Sonic screech and skid a short distance to a halt.
Look up	While standing still, press up on the D-pad to make Sonic gaze to the sky. As the player holds up, the camera pans upward.
Crouch	While standing still, press down on the D-pad to make Sonic duck down. As the player holds down, the camera pans downward.
Spin Attack	Pressing down on the D-pad while moving will make Sonic curl into a rolling attack. Sonic remains in a ball until player jumps or slows down.
Spin Jump	Pressing an action button will make Sonic leap into the air with a rolling attack. Height of the jump is proportional to the length of the button press.
Push	Continue holding the D-pad against a block to have Sonic push it along the ground. This sprite will also be shown if the player attempts this with an object that can't be pushed.

Table A.1: Sonic Moveset.

Sonic The Hedgehog	***Sonic The Hedgehog 2***	***Sonic 3 & Knuckles***
Green Hill (GH)	Emerald Hill (EH)	Angel Island (AI)
Marble (MB)	Chemical Plant (CP)	Hydrocity (HC)
Spring Yard (SY)	Aquatic Ruin (AR)	Marble Garden (MG)
Labyrinth (LB)	Casino Night (CS)	Carnival Night (CR)
Star Light (SL)	Hill Top (HT)	Icecap (IC)
Scrap Brain (SB)	Mystic Cave (MC)	Launch Base (LB)
	Oil Ocean (OO)	Mushroom Hill (MH)
	Metropolis (MP)	Flying Battery (FB)
	Wing Fortress (WF)	Sandopolis (SP)
		Lava Reef (LR)
		Hidden Palace (HP)
		Death Egg (DE)

Table A.2: Game to Zone List. Zone abbreviations noted in parentheses.

Level	**Right Avg**	**Level**	**Right Avg**
GH1	640.4 ± 15.9	*OO2*	943.3 ± 120.8
GH2	139.4 ± 19.8	*MP1*	397.6 ± 81.4
MB1	1168.6 ± 1.5	*MP2*	583.4 ± 48.1
MB2	891.0 ± 723.6	*WF*	544.3 ± 347.0
MB3	1274.3 ± 318.5	*AI1*	1309.2 ± 17.1
SY2	503.6 ± 207.4	*HC2*	58.9 ± 142.0
SY3	1888.3 ± 314.2	*MG1*	454.8 ± 165.3
LB1	597.0 ± 45.2	*MG2*	199.6 ± 2.4
LB2	140.6 ± 2.1	*CR1*	2083.5 ± 77.1
LB3	304.4 ± 429.2	*CR2*	2846.9 ± 60.9
SL1	463.5 ± 14.8	*IC1*	4750.9 ± 6.1
SL2	3957.6 ± 21.5	*IC2*	47.7 ± 4.4
SB2	403.8 ± 0.6	*LB1*	1364.6 ± 269.9
EH1	3950.5 ± 87.7	*LB2*	2201.2 ± 89.9
EH2	868.4 ± 2.5	*MH1*	580.3 ± 15.6
CP1	2971.6 ± 19.0	*MH2*	1034.9 ± 0.3
CP2	1544.7 ± 1125.0	*FB1*	1290.7 ± 6.2
AR1	546.9 ± 29.5	*SP1*	133.8 ± 133.8
AR2	2013.0 ± 113.4	*SP2*	39.0 ± 0.2
CS1	1365.2 ± 238.8	*LR2*	229.8 ± 116.7
HT1	639.0 ± 233.3	*HP*	1440.0 ± 53.2
MC1	670.1 ± 7.8	*DE1*	85.7 ± 2.6
MC2	546.6 ± 0.0	*DE2*	318.2 ± 15.4
OO1	1231.2 ± 58.4	*Total*	1099.1 ± 1092.8

Table A.3: Hold Right Train Baseline.

Level	**Right Avg**	**Human Avg**
GH2	133.0 ± 0.6	8166.1 ± 614.0
SY1	101.6 ± 22.3	6744.0 ± 1172.0
SL3	355.6 ± 32.1	8597.2 ± 729.5
SB1	714.0 ± 266.1	6413.8 ± 922.2
MP3	313.3 ± 110.5	6004.8 ± 440.4
HT2	395.7 ± 241.4	8600 ± 772.1
CS2	521.4 ± 135.2	8662.3 ± 1402.6
LR1	95.3 ± 79.5	8758.3 ± 477.9
FB2	840.9 ± 18.5	6021.6 ± 1006.7
HC1	48.8 ± 41.2	7146.0 ± 1555.1
AI2	21.3 ± 4.1	6705.6 ± 742.4
Total	321.9 ± 277.5	7438.2 ± 1126.0

Table A.4: Hold Right and Human Test Baselines.

Level	**Joint Avg**	**Best Ckpts**	**Level**	**Joint Avg**	**Best Ckpts**
GH1	649.8 ± 0.0	649.8 ± 0.0	*OO2*	990.6 ± 0.0	990.6 ± 0.0
GH2	214.3 ± 45.4	254.6 ± 0.0	*MP1*	443.7 ± 0.0	443.7 ± 0.0
MB1	1170.1 ± 0.0	1170.1 ± 0.0	*MP2*	1077.6 ± 447.0	1335.7 ± 0.0
MB2	1353.2 ± 0.0	1408.1 ± 0.0	*WF*	1145.3 ± 0.0	1145.3 ± 0.0
MB3	1923.5 ± 889.3	2950.4 ± 0.0	*AI1*	1326.3 ± 0.0	1761.5 ± 0.0
SY2	821.6 ± 1.3	823.7 ± 1.3	*HC2*	21.0 ± 0.0	21.0 ± 0.0
SY3	2067.9 ± 210.4	2237.2 ± 0.0	*MG1*	1460.1 ± 434.5	1989.9 ± 4.4
LB1	635.7 ± 0.0	635.7 ± 0.0	*MG2*	241.3 ± 66.0	317.6 ± 0.0
LB2	1085.7 ± 1633.3	42971.7 ± 0.0	*CR1*	2206.3 ± 0.0	3541.8 ± 35.0
LB3	242.2 ± 0.0	242.2 ± 0.0	*CR2*	2861.4 ± 20.8	2873.4 ± 0.0
SL1	1522.1 ± 906.8	3866.4 ± 2302.6	*IC1*	4895.8 ± 0.0	4895.8 ± 0.0
SL2	3694.2 ± 354.7	3980.2 ± 0.0	*IC2*	52.1 ± 0.0	600.0 ± 9.2
SB2	832.4 ± 421.6	1252.2 ± 0.0	*LB1*	1324.3 ± 536.0	1634.5 ± 0.0
EH1	4038.2 ± 0.0	4038.2 ± 0.0	*LB2*	2255.6 ± 0.0	2255.6 ± 0.0
EH2	3832.1 ± 5123.1	9747.7 ± 238.9	*MH1*	725.2 ± 186.5	1399.6 ± 742.7
CP1	3014.0 ± 96.0	3115.4 ± 0.0	*MH2*	1101.2 ± 179.6	1814.1 ± 778.9
CP2	948.4 ± 0.0	948.4 ± 0.0	*FB1*	939.5 ± 628.3	1302.3 ± 0.0
AR1	1581.9 ± 1373.2	3248.3 ± 0.0	*SP1*	0.0 ± 0.0	0.0 ± 0.0
AR2	2080.7 ± 0.0	2080.7 ± 0.0	*SP2*	220.7 ± 278.0	540.8 ± 14.4
CS1	1711.3 ± 0.0	1711.3 ± 0.0	*LR2*	160.0 ± 0.0	160.0 ± 0.0
HT1	766.7 ± 0.0	766.7 ± 0.0	*HP*	1960.0 ± 0.0	1960.0 ± 0.0
MC1	1060.5 ± 403.1	1481.4 ± 649.3	*DE1*	466.7 ± 207.8	958.7 ± 462.2
MC2	546.6 ± 0.0	546.6 ± 0.0	*DE2*	333.6 ± 0.0	403.1 ± 0.0
OO1	1195.5 ± 0.0	1195.5 ± 0.0	*Total*	1344.6 ± 1124.4	1780.2 ± 1708.5

Table A.5: Agent 1 (1V 256 n.p.) Joint Scores.

Timesteps	**Joint Avg**	**Bot 5 Avg**	**Transfer Avg**
5m	1175.7 ± 0.0	84.2 ± 0.0	346.6 ± 0.0
10m	1200.0 ± 42.8	51.3 ± 0.0	373.5 ± 46.6
15m	1222.9 ± 82.6	59.3 ± 13.7	439.2 ± 160.4
20m	1240.5 ± 61.5	53.9 ± 4.5	375.1 ± 25.5
25m	1253.7 ± 71.2	62.3 ± 16.1	393.6 ± 40.9
30m	1344.6 ± 206.4	65.9 ± 18.9	430.3 ± 134.7

Table A.6: Agent 1 (1V 256 n.p.) Score Progression.

Level	Solo Avg	Solo Best Ckpts	Transfer Avg 30m	Transfer Best Ckpts
GH2	9627.4 ± 397.0	10064.0 ± 344.2	139.2 ± 0.0	139.2 ± 0.0
SY1	1889.7 ± 31.7	1923.3 ± 0.0	116.9 ± 0.0	116.9 ± 0.0
SL3	2881.4 ± 206.9	3105.4 ± 0.0	404.5 ± 0.0	404.5 ± 0.0
SB1	1893.0 ± 516.4	2461.2 ± 405.5	612.1 ± 3.2	613.9 ± 0.0
MP3	2025.7 ± 517.7	2623.3 ± 484.2	327.7 ± 0.0	327.7 ± 0.0
HT2	783.6 ± 117.0	851.2 ± 0.0	453.5 ± 459.8	984.5 ± 475.1
CS2	2626.5 ± 3452.2	6612.5 ± 0.0	1189.2 ± 894.4	3607.4 ± 818.5
LR1	659.7 ± 965.2	1774.3 ± 1378.7	259.0 ± 123.1	338.8 ± 273.4
FB2	858.7 ± 19.0	869.6 ± 0.0	868.5 ± 64.1	1409.5 ± 14.0
HC1	2311.2 ± 956.6	3006.9 ± 184.3	253.7 ± 0.0	253.7 ± 0.0
AI2	1695.6 ± 267.3	1991.4 ± 584.7	109.6 ± 0.0	109.6 ± 0.0
Total	2477.5 ± 2482.6	3207.6 ± 2752.4	430.3 ± 340.5	755.1 ± 1028.1

Table A.7: Agent 1 (1V 256 n.p.) Test Scores.

Level	Joint Avg 30m	Best Ckpts	Level	Joint Avg 30m	Best Ckpts
GH1	2114.0 ± 2261.3	5202.9 ± 4.7	*OO2*	1446.6 ± 948.1	5676.2 ± 2109.7
GH2	663.1 ± 576.8	2679.0 ± 944.9	*MP1*	668.2 ± 831.5	1683.4 ± 16.2
MB1	3121.4 ± 2007.3	5405.0 ± 391.2	*MP2*	1320.0 ± 1213.7	3176.3 ± 272.7
MB2	1444.8 ± 767.9	2332.8 ± 0.0	*WF*	789.9 ± 362.7	1093.5 ± 10.2
MB3	1808.5 ± 964.0	2949.7 ± 98.9	*AI1*	1487.1 ± 1326.3	3884.2 ± 842.3
SY2	671.3 ± 136.2	1357.5 ± 143.8	*HC2*	229.2 ± 390.0	679.5 ± 331.2
SY3	1695.9 ± 290.5	2030.6 ± 306.1	*MG1*	1522.1 ± 433.5	2132.7 ± 368.9
LB1	1612.9 ± 970.6	2729.1 ± 1033.2	*MG2*	312.6 ± 280.5	601.2 ± 131.1
LB2	2692.2 ± 401.0	2962.8 ± 14.2	*CR1*	2974.5 ± 2568.7	5513.9 ± 25.7
LB3	1429.3 ± 1174.3	2612.0 ± 1.2	*CR2*	1644.6 ± 1354.5	2849.6 ± 34.5
SL1	1463.8 ± 1619.7	3528.8 ± 219.4	*IC1*	4798.2 ± 67.3	5029.4 ± 199.5
SL2	1436.3 ± 769.9	4380.0 ± 718.7	*IC2*	950.8 ± 650.2	2726.6 ± 8.0
SB2	1192.2 ± 82.1	1252.2 ± 0.0	*LB1*	1110.5 ± 1452.6	2754.4 ± 20.9
EH1	3479.0 ± 2412.7	6070.4 ± 996.1	*LB2*	1386.0 ± 536.1	2635.9 ± 78.9
EH2	3207.6 ± 2537.6	8606.8 ± 2704.9	*MH1*	674.0 ± 637.9	1428.4 ± 5.7
CP1	1646.5 ± 1485.9	4320.1 ± 1421.2	*MH2*	560.0 ± 461.6	1648.7 ± 652.8
CP2	1340.6 ± 1364.1	3516.3 ± 133.7	*FB1*	621.4 ± 612.8	1302.3 ± 0.0
AR1	2658.7 ± 2018.0	4557.0 ± 258.2	*SP1*	142.2 ± 122.1	893.3 ± 604.8
AR2	2166.2 ± 1971.9	3881.7 ± 939.4	*SP2*	212.8 ± 335.0	600.8 ± 5.5
CS1	1300.5 ± 918.1	2307.0 ± 421.9	*LR2*	145.8 ± 19.8	1296.6 ± 4.1
HT1	425.3 ± 497.1	975.5 ± 375.9	*HP*	5435.2 ± 3834.2	9811.6 ± 354.2
MC1	857.3 ± 800.7	1589.7 ± 775.8	*DE1*	403.2 ± 311.1	2942.8 ± 549.2
MC2	340.3 ± 296.4	546.6 ± 0.0	*DE2*	604.7 ± 792.6	2433.1 ± 464.2
OO1	1329.7 ± 1037.5	2345.5 ± 226.6	*Total*	1479.5 ± 1148.3	2998.6 ± 2010.6

Table A.8: Agent 2 (1V 256) Joint Scores.

Timesteps	**Joint Avg**	**Bot 5 Avg**	**Transfer Avg**
5m	1255.9 ± 220.1	137.8 ± 114.1	484.6 ± 192.9
10m	1501.4 ± 347.9	202.1 ± 40.3	721.9 ± 305.7
15m	1591.1 ± 399.6	164.8 ± 22.2	720.0 ± 265.9
20m	1584.9 ± 628.8	189.7 ± 106.6	742.0 ± 254.2
25m	1585.3 ± 803.4	172.5 ± 159.9	803.5 ± 482.7
30m	1479.5 ± 911.3	174.1 ± 198.4	578.5 ± 409.7

Table A.9: Agent 2 (1V 256) Score Progression.

Level	**Solo Avg**	**Solo Best Ckpts**	**Transfer Avg 30m**	**Transfer Best Ckpts**
GH2	3674.0 ± 4710.3	9021.3 ± 2105.0	1629.6 ± 1956.9	3843.9 ± 1662.9
SY1	1105.0 ± 1021.0	1297.2 ± 715.2	377.5 ± 77.3	636.6 ± 116.7
SL3	2790.5 ± 333.9	3095.2 ± 10.2	352.8 ± 140.0	1302.5 ± 776.0
SB1	1912.1 ± 535.4	2526.6 ± 409.0	661.1 ± 249.3	1293.1 ± 1216.7
MP3	993.5 ± 532.1	1701.3 ± 14.5	221.1 ± 179.4	358.7 ± 288.1
HT2	869.7 ± 1102.9	1649.6 ± 498.8	530.5 ± 588.3	2441.2 ± 5.3
CS2	396.0 ± 343.3	3617.4 ± 2131.5	809.8 ± 701.6	3666.2 ± 2132.9
LR1	1038.9 ± 529.8	752.7 ± 848.8	204.8 ± 313.2	566.0 ± 155.7
FB2	134.0 ± 232.1	836.7 ± 32.9	852.1 ± 291.1	1399.5 ± 120.9
HC1	979.8 ± 954.4	2024.3 ± 1444.1	148.8 ± 123.5	279.4 ± 520.3
AI2	1391.8 ± 1166.3	2097.3 ± 476.7	575.1 ± 387.5	1007.4 ± 607.2
Total	1389.6 ± 1040.0	2601.8 ± 2302.9	578.5 ± 422.6	1526.8 ± 1257.4

Table A.10: Agent 2 (1V 256) Test Scores.

Level	**Joint Avg 30m**	**Best Ckpts**	**Level**	**Joint Avg 30m**	**Best Ckpts**
GH1	3044.0 ± 1471.6	4437.6 ± 313.8	*OO2*	2282.9 ± 1152.5	2982.6 ± 603.9
GH2	475.8 ± 410.0	949.2 ± 782.3	*MP1*	917.5 ± 460.0	1644.9 ± 98.3
MB1	1831.9 ± 645.0	3179.1 ± 1275.6	*MP2*	1230.2 ± 113.4	2380.2 ± 738.6
MB2	2076.8 ± 345.5	2332.8 ± 0.0	*WF*	1020.1 ± 500.8	1448.6 ± 1115.3
MB3	2254.4 ± 840.1	2947.4 ± 95.0	*AI1*	2048.8 ± 771.7	4091.7 ± 1.6
SY2	805.8 ± 168.7	1158.3 ± 342.3	*HC2*	461.4 ± 323.8	752.5 ± 126.6
SY3	1993.8 ± 22.2	2210.6 ± 342.7	*MG1*	1643.8 ± 744.6	2348.9 ± 185.7
LB1	1627.9 ± 795.2	2301.8 ± 1129.6	*MG2*	514.7 ± 196.2	719.1 ± 43.5
LB2	2919.9 ± 25.7	2964.0 ± 11.8	*CR1*	2439.5 ± 1471.3	5540.0 ± 2.5
LB3	1099.8 ± 821.9	2587.9 ± 26.6	*CR2*	2473.0 ± 357.8	2854.3 ± 30.6
SL1	1690.4 ± 278.3	4858.0 ± 1362.7	*IC1*	4876.0 ± 204.5	5393.8 ± 154.6
SL2	3044.7 ± 1108.1	7166.1 ± 996.1	*IC2*	1091.8 ± 1436.2	3678.0 ± 112.2
SB2	1178.5 ± 91.4	1251.6 ± 0.6	*LB1*	1063.4 ± 550.5	2413.6 ± 333.9
EH1	6025.4 ± 3386.6	9798.2 ± 1471.1	*LB2*	2295.4 ± 357.7	2699.8 ± 21.9
EH2	8589.6 ± 2267.9	10210.7 ± 487.5	*MH1*	857.5 ± 497.8	1432.2 ± 9.4
CP1	2129.2 ± 1645.7	4857.1 ± 114.6	*MH2*	1193.4 ± 626.0	2168.9 ± 619.4
CP2	2069.7 ± 984.2	3639.3 ± 10.7	*FB1*	1248.6 ± 48.2	1302.3 ± 0.0
AR1	3875.7 ± 583.6	4748.1 ± 1338.5	*SP1*	275.9 ± 103.0	964.4 ± 431.2
AR2	2998.5 ± 1027.5	4030.3 ± 1187.5	*SP2*	341.3 ± 235.9	552.2 ± 33.756
CS1	1039.6 ± 925.2	2494.2 ± 110.2	*LR2*	542.4 ± 230.3	689.0 ± 580.1
HT1	750.3 ± 3.3	754.3 ± 55.6	*HP*	4158.7 ± 875.6	6430.8 ± 2069.7
MC1	998.2 ± 109.0	1286.0 ± 384.3	*DE1*	411.8 ± 376.7	1676.0 ± 80.6
MC2	1444.0 ± 1013.8	2122.6 ± 1899.5	*DE2*	642.6 ± 697.9	2908.7 ± 5.4
OO1	1538.6 ± 312.9	2253.2 ± 124.5	*Total*	1905.0 ± 1572.3	3013.0 ± 2186.2

Table A.11: Agent 3 (2V 512) Joint Scores.

Timesteps	**Joint Avg**	**Bot 5 Avg**	**Transfer Avg**
5m	1146.5 ± 185.0	108.8 ± 64.4	406.8 ± 108.9
10m	1383.5 ± 253.1	140.7 ± 65.3	531.6 ± 234.2
15m	1622.7 ± 455.5	175.0 ± 122.3	602.3 ± 315.2
20m	1827.2 ± 618.9	226.4 ± 161.0	791.6 ± 389.9
25m	1824.0 ± 611.4	207.4 ± 94.4	718.6 ± 304.5
30m	1905.0 ± 286.9	217.7 ± 68.4	678.5 ± 238.3

Table A.12: Agent 3 (2V 512) Score Progression.

Level	Solo Avg	Solo Best Ckpts	Transfer Avg 30m	Transfer Best Ckpts
GH2	5379.8 ± 3127.6	8750.4 ± 2238.0	1532.1 ± 1631.6	3383.3 ± 1566.8
SY1	1413.5 ± 383.5	1814.9 ± 54.5	219.0 ± 205.1	1006.9 ± 763.2
SL3	2746.5 ± 320.0	3014.0 ± 44.4	607.7 ± 367.4	1736.0 ± 21.5
SB1	1621.8 ± 135.7	1725.4 ± 899.3	810.7 ± 53.7	1279.5 ± 561.7
MP3	1298.5 ± 459.3	1677.4 ± 20.4	792.4 ± 820.8	1740.0 ± 893.5
HT2	811.5 ± 1395.0	2422.3 ± 6.9	1141.9 ± 762.8	2096.4 ± 1132.0
CS2	3124.2 ± 3509.3	7144.7 ± 610.9	800.2 ± 97.3	4139.8 ± 1348.9
LR1	2248.5 ± 1080.9	3413.5 ± 199.6	321.8 ± 274.0	652.5 ± 63.0
FB2	423.4 ± 383.4	793.6 ± 8.2	740.2 ± 43.6	1331.0 ± 101.8
HC1	1650.0 ± 1356.0	3165.3 ± 19.7	129.4 ± 82.0	410.5 ± 762.3
AI2	1503.0 ± 1243.9	2429.7 ± 509.2	368.2 ± 252.6	780.7 ± 372.0
Total	2020.1 ± 1361.9	3304.6 ± 2443.9	678.5 ± 416.4	1686.9 ± 1155.1

Table A.13: Agent 3 (2V 512) Test Scores.

Level	Joint Avg 30m	Best Ckpts	Level	Joint Avg 30m	Best Ckpts
GH1	1884.6 ± 1369.6	3948.2 ± 268.5	*OO2*	1845.8 ± 850.4	2619.4 ± 953.1
GH2	576.4 ± 568.4	1233.2 ± 705.3	*MP1*	872.0 ± 176.5	1703.7 ± 7.1
MB1	1992.5 ± 329.5	3191.5 ± 1008.1	*MP2*	1158.3 ± 201.7	2422.2 ± 903.5
MB2	1708.1 ± 502.0	2332.8 ± 0.0	*WF*	1147.8 ± 572.8	1750.9 ± 1315.3
MB3	2380.3 ± 927.1	3043.2 ± 5.4	*AI1*	2545.8 ± 1116.5	5305.0 ± 155.0
SY2	878.3 ± 389.8	2351.8 ± 1036.9	*HC2*	270.4 ± 415.1	748.4 ± 139.3
SY3	1831.7 ± 298.9	2043.7 ± 322.8	*MG1*	1529.7 ± 61.0	1875.6 ± 451.9
LB1	1598.1 ± 395.2	2147.8 ± 256.4	*MG2*	265.1 ± 44.2	317.3 ± 0.3
LB2	2114.1 ± 1399.2	2970.7 ± 7.2	*CR1*	3175.6 ± 2490.2	5569.6 ± 7.9
LB3	1442.5 ± 1132.4	2568.8 ± 45.7	*CR2*	2534.7 ± 280.4	2859.0 ± 35.8
SL1	1475.6 ± 1145.0	4726.9 ± 81.0	*IC1*	4771.5 ± 48.8	4824.5 ± 2.0
SL2	2240.0 ± 1705.8	6429.9 ± 1291.9	*IC2*	1945.9 ± 1276.1	4037.0 ± 67.5
SB2	1196.0 ± 61.8	1251.6 ± 0.6	*LB1*	1274.4 ± 1304.6	2769.5 ± 5.8
EH1	7782.6 ± 2803.2	9834.3 ± 256.6	*LB2*	2286.0 ± 380.7	2710.0 ± 3.4
EH2	6239.6 ± 2489.0	9837.3 ± 320.9	*MH1*	1766.0 ± 1386.6	3331.5 ± 20.5
CP1	2918.3 ± 31.0	4982.3 ± 0.9	*MH2*	1353.5 ± 390.3	2417.6 ± 2.3
CP2	2107.4 ± 675.3	2945.8 ± 210.7	*FB1*	1195.9 ± 104.9	1302.3 ± 0.0
AR1	2531.4 ± 502.9	4127.9 ± 1397.5	*SP1*	415.7 ± 493.0	1269.0 ± 940.1
AR2	2382.1 ± 741.1	4458.2 ± 562.9	*SP2*	372.9 ± 312.3	625.7 ± 118.2
CS1	1619.7 ± 518.1	2116.6 ± 434.5	*LR2*	185.0 ± 40.1	778.0 ± 522.7
HT1	742.6 ± 12.8	755.0 ± 76.9	*HP*	6208.8 ± 2501.5	8654.3 ± 1229.3
MC1	905.2 ± 234.0	1087.1 ± 247.3	*DE1*	108.4 ± 105.0	1286.4 ± 20.0
MC2	552.1 ± 228.7	1587.1 ± 1086.9	*DE2*	970.3 ± 950.0	1905.8 ± 14.1
OO1	1704.2 ± 145.1	1892.4 ± 374.3	*Total*	1893.7 ± 1570.3	3041.4 ± 2227.9

Table A.14: Agent 4 (1VA $2 \cdot 256$ 128H) Joint Scores.

Timesteps	**Joint Avg**	**Bot 5 Avg**	**Transfer Avg**
5m	1311.1 ± 214.8	121.3 ± 60.1	563.4 ± 215.2
10m	1297.4 ± 146.5	94.0 ± 25.7	478.0 ± 171.3
15m	1534.5 ± 306.9	134.7 ± 64.9	676.0 ± 271.6
20m	1924.9 ± 170.4	189.1 ± 44.0	868.5 ± 243.5
25m	2154.6 ± 275.4	204.1 ± 25.3	1042.8 ± 318.2
30m	1893.7 ± 225.3	128.7 ± 76.6	817.3 ± 320.2

Table A.15: Agent 4 (1VA $2 \cdot 256$ 128H) Score Progression.

Level	**Solo Avg**	**Solo Best Ckpts**	**Transfer Avg 30m**	**Transfer Best Ckpts**
GH2	3716.4 ± 3646.0	7781.7 ± 2605.4	661.6 ± 531.5	2550.8 ± 1460.7
SY1	1199.9 ± 630.6	1917.3 ± 6.0	387.3 ± 236.0	765.9 ± 772.3
SL3	1694.0 ± 1153.4	2636.0 ± 657.4	1548.9 ± 1039.3	2599.3 ± 728.6
SB1	1857.1 ± 381.3	2286.0 ± 1746.0	657.8 ± 383.0	1360.7 ± 728.7
MP3	1275.8 ± 446.2	1730.6 ± 544.1	369.9 ± 46.6	444.4 ± 411.8
HT2	3244.8 ± 345.9	3644.2 ± 1053.9	1689.4 ± 675.2	2671.8 ± 715.0
CS2	2668.5 ± 1002.0	3630.9 ± 23.5	1754.4 ± 1152.4	5181.6 ± 1420.5
LR1	1780.0 ± 1511.9	2694.5 ± 1367.3	389.7 ± 553.4	1165.3 ± 972.6
FB2	366.9 ± 605.8	1066.2 ± 225.1	957.2 ± 289.9	1401.2 ± 20.8
HC1	1146.2 ± 601.0	1765.9 ± 969.3	191.0 ± 136.3	341.7 ± 350.7
AI2	1692.1 ± 1030.8	2422.7 ± 803.2	382.8 ± 248.8	1088.4 ± 583.3
Total	1876.5 ± 979.4	2870.5 ± 1805.8	817.3 ± 582.4	1779.2 ± 1403.1

Table A.16: Agent 4 (1VA $2 \cdot 256$ 128H) Test Scores.

Level	**Joint Avg 30m**	**Best Ckpts**	**Level**	**Joint Avg 30m**	**Best Ckpts**
GH1	1792.5 ± 1173.7	3961.8 ± 265.3	*OO2*	1030.5 ± 97.9	1414.8 ± 755.5
GH2	968.4 ± 805.2	1934.6 ± 2119.8	*MP1*	904.6 ± 638.5	1630.6 ± 80.2
MB1	3054.8 ± 118.1	3464.7 ± 1153.5	*MP2*	1629.6 ± 430.9	2125.3 ± 811.3
MB2	1415.8 ± 524.2	2332.8 ± 0.0	*WF*	939.9 ± 555.3	1576.9 ± 1343.7
MB3	1639.8 ± 495.6	2211.0 ± 970.26	*AI1*	2725.5 ± 1450.6	4376.1 ± 356.6
SY2	949.0 ± 358.8	2093.4 ± 1270.5	*HC2*	397.9 ± 342.3	744.9 ± 241.2
SY3	1873.2 ± 168.3	2224.2 ± 9.7	*MG1*	1721.5 ± 420.9	2207.1 ± 386.0
LB1	2270.2 ± 223.4	2575.8 ± 876.1	*MG2*	328.8 ± 234.7	676.8 ± 9.5
LB2	2233.7 ± 662.5	2954.0 ± 40.6	*CR1*	4946.9 ± 799.3	5555.7 ± 16.8
LB3	1160.3 ± 1253.2	2596.6 ± 19.2	*CR2*	2425.1 ± 491.6	2838.8 ± 79.9
SL1	2821.1 ± 1180.8	3848.9 ± 541.7	*IC1*	4954.2 ± 316.4	5319.3 ± 3.0
SL2	2585.1 ± 1778.4	4635.7 ± 831.7	*IC2*	2065.8 ± 1040.7	3114.1 ± 448.2
SB2	1221.5 ± 10.9	1248.7 ± 3.4	*LB1*	1106.1 ± 375.1	2774.1 ± 0.6
EH1	5294.5 ± 1939.0	7333.1 ± 25.6	*LB2*	2301.0 ± 704.7	2710.7 ± 4.1
EH2	6874.2 ± 5206.4	10006.3 ± 346.0	*MH1*	1116.8 ± 359.1	3148.9 ± 172.9
CP1	4102.7 ± 340.7	4493.4 ± 44.5	*MH2*	1724.1 ± 1033.9	2591.5 ± 205.6
CP2	2006.7 ± 630.4	3486.2 ± 8.6	*FB1*	1142.2 ± 204.3	1291.1 ± 5.0
AR1	3092.6 ± 728.0	3933.1 ± 686.1	*SP1*	676.4 ± 647.9	1423.6 ± 261.1
AR2	3102.9 ± 333.0	3397.9 ± 1110.0	*SP2*	350.5 ± 271.9	754.0 ± 268.8
CS1	1760.6 ± 101.4	2224.9 ± 15.4	*LR2*	286.1 ± 305.1	1238.0 ± 75.5
HT1	727.7 ± 7.1	777.0 ± 297.6	*HP*	6659.9 ± 2316.8	9122.7 ± 166.5
MC1	1089.9 ± 174.2	1290.6 ± 336.5	*DE1*	738.4 ± 532.7	1755.5 ± 487.7
MC2	507.3 ± 16.8	789.3 ± 267.0	*DE2*	962.5 ± 794.2	2008.2 ± 1644.8
OO1	1782.8 ± 324.9	2155.3 ± 239.6	*Total*	2031.1 ± 1572.0	2901.5 ± 2011.1

Table A.17: Agent 5 (1VA $2 \cdot 256$ 512H) Joint Scores.

Timesteps	**Joint Avg**	**Bot 5 Avg**	**Transfer Avg**
5m	1143.0 ± 42.3	84.2 ± 17.7	360.2 ± 34.1
10m	1394.7 ± 170.5	153.3 ± 74.8	721.8 ± 193.0
15m	1390.5 ± 300.6	187.5 ± 124.4	587.0 ± 140.6
20m	1754.4 ± 81.9	217.5 ± 25.5	1078.4 ± 30.9
25m	1874.8 ± 373.0	261.7 ± 100.7	1041.0 ± 114.6
30m	2031.1 ± 501.9	257.9 ± 121.9	936.6 ± 220.8

Table A.18: Agent 5 (1VA $2 \cdot 256$ 512H) Score Progression.

Level	Solo Avg	Solo Best Ckpts	Transfer Avg 30m	Transfer Best Ckpts
GH2	6687.1 ± 4554.9	9464.5 ± 2557.6	1285.0 ± 711.6	2616.3 ± 21.3
SY1	316.7 ± 256.4	599.5 ± 130.6	237.8 ± 51.8	753.7 ± 519.3
SL3	2348.2 ± 322.9	2720.2 ± 252.6	1729.6 ± 1282.0	2939.9 ± 114.6
SB1	1291.5 ± 396.2	1747.2 ± 25.5	855.4 ± 86.6	1539.4 ± 559.1
MP3	1018.0 ± 234.0	1288.2 ± 287.5	327.6 ± 6.3	439.7 ± 403.5
HT2	619.3 ± 902.0	1660.2 ± 1180.2	1800.9 ± 765.1	2915.8 ± 904.42
CS2	4033.7 ± 220.5	4169.9 ± 12.7	1973.1 ± 946.2	4446.0 ± 465.3
LR1	122.8 ± 160.9	307.0 ± 306.1	201.5 ± 139.3	447.5 ± 247.5
FB2	54.0 ± 35.5	78.8 ± 70.8	908.6 ± 216.5	1401.4 ± 11.2
HC1	246.8 ± 366.8	670.2 ± 336.4	305.7 ± 51.3	699.7 ± 1047.0
AI2	1058.7 ± 1351.3	2601.5 ± 375.1	677.1 ± 203.2	1126.9 ± 636.8
Total	1617.9 ± 2051.7	2300.7 ± 2669.9	936.6 ± 666.6	1756.9 ± 1298.3

Table A.19: Agent 5 (1VA $2 \cdot 256$ 512H) Test Scores.

Appendix B

Figures

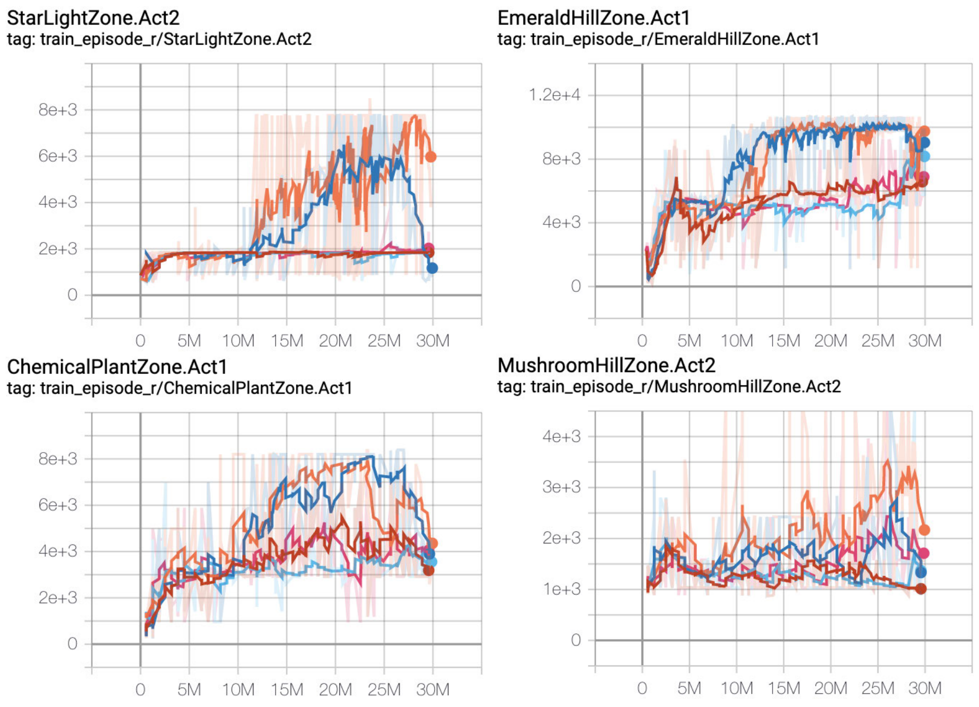


Figure B-1: Graph of training episode rewards over time. Agent 3 and Agent 4 achieve significantly higher scores on these four levels than the other agent variants. This appears to be the result of memorizing certain sections.

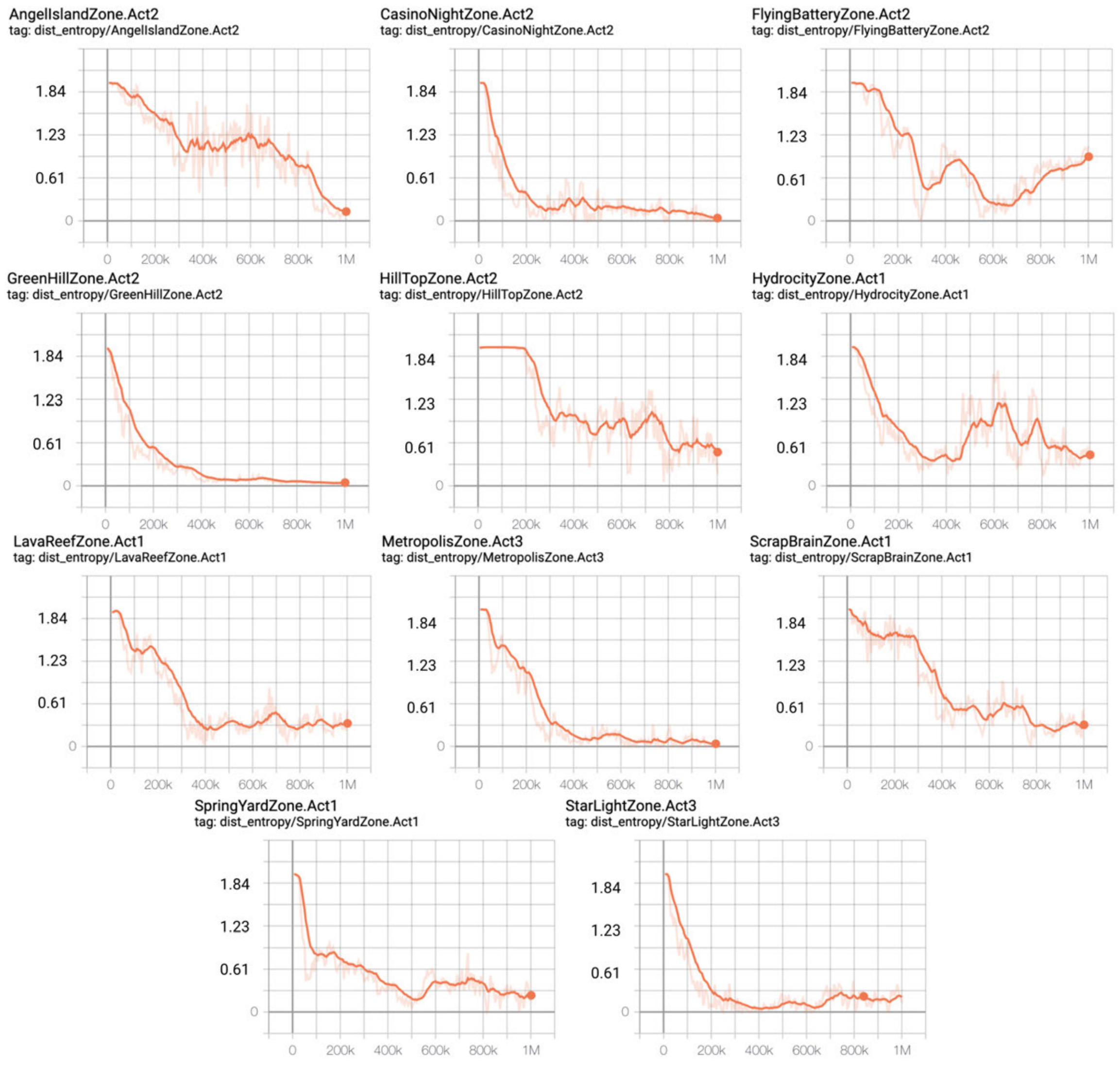


Figure B-2: Agent 1 action distribution entropy over time for the solo train task.

Bibliography

[1] Willi Apel. *The Harvard dictionary of music.* Harvard University Press, 2003.

[2] Karl J Astrom. Optimal control of markov processes with incomplete state information. *Journal of mathematical analysis and applications*, 10(1):174–205, 1965.

[3] Adrià Puigdomènech Badia, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Daniel Guo, and Charles Blundell. Agent57: Outperforming the atari human benchmark. *arXiv preprint arXiv:2003.13350*, 2020.

[4] Adrià Puigdomènech Badia, Pablo Sprechmann, Alex Vitvitskyi, Daniel Guo, Bilal Piot, Steven Kapturowski, Olivier Tieleman, Martín Arjovsky, Alexander Pritzel, Andew Bolt, et al. Never give up: Learning directed exploration strategies. *arXiv preprint arXiv:2002.06038*, 2020.

[5] Charles Beattie, Joel Z Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, et al. Deepmind lab. *arXiv preprint arXiv:1612.03801*, 2016.

[6] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[7] Richard Bellman. On the theory of dynamic programming. *Proceedings of the National Academy of Sciences of the United States of America*, 38(8):716, 1952.

[8] Ross Bencina and Phil Burk. Portaudio-an open source cross platform audio api. In *ICMC*, 2001.

[9] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

[10] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[11] Boston Dynamics. Atlas. `"https://www.bostondynamics.com/atlas"`, 2020. [Online; accessed 14-August-2020].

[12] Ronald Newbold Bracewell and Ronald N Bracewell. *The Fourier transform and its applications*, volume 31999. McGraw-Hill New York, 1986.

[13] J Peter Burkholder, Donald Jay Grout, and Claude V Palisca. *A History of Western Music: Tenth International Student Edition*. WW Norton & Company, 2019.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[15] D.S. Cohen. History of sonic the hedgehog by sega genesis. `"https://www.lifewire.com/history-of-sonic-the-hedgehog-729671"`, 2019. [Online; accessed 14-August-2020].

[16] explod2A03. Nes audio: Brief explanation of sound channels. `"https://www.youtube.com/watch?v=la3coK5pq5w"`, 2012. [Online; accessed 14-August-2020].

[17] FFmpeg. Ffmpeg. `"https://ffmpeg.org/"`, 2020. [Online; accessed 14-August-2020].

[18] L. Pettersson J. Schneider J. Schulman J. Tang G. Brockman, V. Cheung and W. Zaremba. Open ai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[19] GST Channel. The sound capabilities of the sega genesis. `"https://www.youtube.com/watch?v=IGy7HBG3I1c"`, 2017. [Online; accessed 14-August-2020].

[20] GST Channel. The sound capabilities of the snes. `"https://www.youtube.com/watch?v=dtK0t8k6akg"`, 2019. [Online; accessed 14-August-2020].

[21] Kate Hevner. Experimental studies of the elements of expression in music. *The American Journal of Psychology*, 48(2):246–268, 1936.

[22] Irina Higgins, Arka Pal, Andrei A Rusu, Loic Matthey, Christopher P Burgess, Alexander Pritzel, Matthew Botvinick, Charles Blundell, and Alexander Lerchner. Darla: Improving zero-shot transfer in reinforcement learning. *arXiv preprint arXiv:1707.08475*, 2017.

[23] Sander Huiberts and Richard Van Tol. Ieza: A framework for game audio. *En ligne. Gamasutra,< http://www. gamasutra. com/view/feature/3509/ieza_ a_framework_for_ game_ audio. php*, 2008.

[24] ITURBT ITU. Parameter values for the hdtv standards for production and international programme exchange. *Recommendation ITU-R BT*, pages 709–5, 2002.

[25] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134, 1998.

[26] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[27] Russell Kaplan, Christopher Sauer, and Alexander Sosa. Beating atari with natural language guided reinforcement learning. *arXiv preprint arXiv:1704.05539*, 2017.

[28] Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. Recurrent experience replay in distributed reinforcement learning. In *International conference on learning representations*, 2018.

[29] Lakshmish Kaushik, Abhijeet Sangwan, and John HL Hansen. Sentiment extraction from natural audio streams. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8485–8489. IEEE, 2013.

[30] Asifullah Khan, Anabia Sohail, Umme Zahoora, and Aqsa Saeed Qureshi. A survey of the recent architectures of deep convolutional neural networks. *Artificial Intelligence Review*, pages 1–62, 2020.

[31] Dong-Ki Kim, Shayegan Omidshafiei, Jason Pazis, and Jonathan P How. Cross-modal attentive skill learner: learning in atari and beyond with audio–video inputs. *Autonomous Agents and Multi-Agent Systems*, 34(1):16, 2020.

[32] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.

[33] Ilya Kostrikov. Pytorch implementations of reinforcement learning algorithms. `https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail`, 2018.

[34] Birger Langkjær. Making fictions sound real-on film sound, perceptual realism and genre. *MedieKultur: Journal of media and communication research*, 26(48):13–p, 2009.

[35] Megan Lavengood. Timbre, genre, and polystylism in sonic the hedgehog 3. In *On Popular Music and Its Unruly Entanglements*, pages 209–234. Springer, 2019.

[36] Alan Levinovitz. The mystery of go, the ancient game that computers still can't win. *Wired Magazine*, 2014.

[37] Liam Triforce. Understanding the music of sonic the hedgehog. `"https://www.youtube.com/watch?v=XDooMjw9uhU"`, 2020. [Online; accessed 14-August-2020].

[38] Logan Plant. Sonic designer yuji naka provides insight on early days of sonic the hedgehog. `"https://nintendowire.com/news/2018/05/15/sonic-designer-yuji-naka-provides-insight-early-days-sonic-hedgehog/"`, 2018. [Online; accessed 14-August-2020].

[39] Ling Ma, Dan J Smith, and Ben P Milner. Context awareness using environmental noise classification. In *Eighth European Conference on Speech Communication and Technology*, 2003.

[40] Martin Korth. Atari 2600 specifications. `"https://problemkaputt.de/2k6specs.htm#audio"`, 2004. [Online; accessed 14-August-2020].

[41] Tomáš Mikolov, Stefan Kombrink, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Extensions of recurrent neural network language model. In *2011 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 5528–5531. IEEE, 2011.

[42] Andrew Milne, William Sethares, and James Plamondon. Isomorphic controllers and dynamic tuning: Invariant fingering over a tuning continuum. *Computer Music Journal*, 31(4):15–32, 2007.

[43] Akshita Mittel and Purna Sowmya Munukutla. Visual transfer between atari games using competitive reinforcement learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 0–0, 2019.

[44] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[45] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[46] Jiquan Ngiam, Aditya Khosla, Mingyu Kim, Juhan Nam, Honglak Lee, and Andrew Y Ng. Multimodal deep learning. In *ICML*, 2011.

[47] Alex Nichol, Vicki Pfau, Christopher Hesse, Oleg Klimov, and John Schulman. Gotta learn fast: A new benchmark for generalization in rl. *arXiv preprint arXiv:1804.03720*, 2018.

[48] Open AI. Roboschool. `"https://github.com/openai/roboschool"`, 2017. [Online; accessed 14-August-2020].

[49] Open AI. Gym retro. `"https://github.com/openai/retro"`, 2020. [Online; accessed 14-August-2020].

[50] OpenCV. Opencv. `"https://github.com/opencv/opencv"`, 2020. [Online; accessed 14-August-2020].

[51] OpenCV. Opencv. `"https://opencv.org/"`, 2020. [Online; accessed 14-August-2020].

[52] Utku Ozbulak. Pytorch cnn visualizations. `https://github.com/utkuozbulak/pytorch-cnn-visualizations`, 2019.

[53] Pangina Pangina. theme of sanic hegehog. `"https://www.youtube.com/watch?v=PX7zPlQjAr8"`, 2012. [Online; accessed 14-August-2020].

[54] Emilio Parisotto, Jimmy Lei Ba, and Ruslan Salakhutdinov. Actor-mimic: Deep multitask and transfer reinforcement learning. *arXiv preprint arXiv:1511.06342*, 2015.

[55] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[56] Hubert Pham. Pyaudio: Portaudio v19 python bindings. *URL: https://people. csail. mit. edu/hubert/pyaudio*, 2006.

[57] Soujanya Poria, Erik Cambria, Newton Howard, Guang-Bin Huang, and Amir Hussain. Fusing audio, visual and textual clues for sentiment analysis from multimodal content. *Neurocomputing*, 174:50–59, 2016.

[58] Ebenezer Prout. *Harmony: its theory and practice.* Cambridge University Press, 2011.

[59] RetroArch. Atlas. `"https://docs.libretro.com/"`, 2010. [Online; accessed 14-August-2020].

[60] Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.

[61] Sabbi Lall. Universal musical harmony. `"http://news.mit.edu/2020/universal-musical-harmony-0701"`, 2020. [Online; accessed 14-August-2020].

[62] Justin Salamon and Juan Pablo Bello. Deep convolutional neural networks and data augmentation for environmental sound classification. *IEEE Signal Processing Letters*, 24(3):279–283, 2017.

[63] Nikolay Savinov, Anton Raichuk, Raphaël Marinier, Damien Vincent, Marc Pollefeys, Timothy Lillicrap, and Sylvain Gelly. Episodic curiosity through reachability. *arXiv preprint arXiv:1810.02274*, 2018.

[64] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897, 2015.

[65] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[66] Ervin Sejdić, Igor Djurović, and Jin Jiang. Time–frequency feature representation using energy concentration: An overview of recent advances. *Digital signal processing*, 19(1):153–183, 2009.

[67] Kun Shao, Zhentao Tang, Yuanheng Zhu, Nannan Li, and Dongbin Zhao. A survey of deep reinforcement learning in video games. *arXiv preprint arXiv:1912.10944*, 2019.

[68] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[69] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[70] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. 2014.

[71] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[72] Stanley Smith Stevens, John Volkmann, and Edwin B Newman. A scale for the measurement of the psychological magnitude pitch. *The Journal of the Acoustical Society of America*, 8(3):185–190, 1937.

[73] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[74] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[75] Murphy Vii Tom. The first level of super mario bros. is easy with lexicographic orderings and time travel... after that it gets a little tricky. 2013.

[76] Nick Ryder Melanie Subbiah Jared Kaplan Prafulla Dhariwal Arvind Neelakantan Pranav Shyam Girish Sastry Amanda Askell Sandhini Agarwal Ariel Herbert-Voss Gretchen Krueger Tom Henighan Rewon Child Aditya Ramesh Daniel M. Ziegler Jeffrey Wu Clemens Winter Christopher Hesse Mark Chen Eric Sigler Mateusz Litwin Scott Gray Benjamin Chess Jack Clark Christopher Berner Sam McCandlish Alec Radford Ilya Sutskever Dario Amodei Tom B. Brown, Benjamin Mann. Language models are few-shot learners. *https://arxiv.org/abs/2005.14165*, 2020.

[77] Trick Lobo. Gotta go fast. `"https://knowyourmeme.com/memes/gotta-go-fast"`, 2011. [Online; accessed 14-August-2020].

[78] VHS Tape. Sanic hegehog. `"https://knowyourmeme.com/memes/sanic-hegehog"`, 2012. [Online; accessed 14-August-2020].

[79] Haofan Wang, Zifan Wang, Mengnan Du, Fan Yang, Zijian Zhang, Sirui Ding, Piotr Mardziel, and Xia Hu. Score-cam: Score-weighted visual explanations for convolutional neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 24–25, 2020.

[80] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[81] Zhongwen Xu, Hado P van Hasselt, and David Silver. Meta-gradient reinforcement learning. In *Advances in neural information processing systems*, pages 2396–2407, 2018.