StepMania Agents: Using Machine Learning to Play A Simple Rhythm Game

Ivan Ho
COGS-4420 Game AI at Rensselaer Polytechnic Institute
New York, United States hoi3@rpi.edu

***Abstract*—Rhythm games are a popular genre of music-based video games that require players to match the rhythm of a song by pressing buttons or keys in time with the music. In recent years, researchers have explored the use of machine-learning techniques to train agents for these games.**

Machine learning algorithms can be used to train agents to play rhythm games by learning from gameplay data and optimizing their performance. These agents can be trained using different types of data, such as human gameplay recordings or generated gameplay data.

Keywords—Unity, machine learning, rhythm game

I. Introduction

Rhythm games are a genre of video games where players make a sequence of inputs based on elements that appear on the screen, while simultaneously timing their inputs to match the rhythm of a song played in the background. The appeal for these types of games stems from the interest in music and extremely high skill ceilings.

Creating automated behaviors for these types of games are often quite easy since the mechanics for rhythm games are usually very simple. However, most of these behaviors are implemented using artificial intelligence, thus there is quite a scarce amount of work done on rhythm games which utilize machine learning. Additionally, much of the works done on rhythm games utilizing machine learning focused on generating new levels (a.k.a. charts) for various kinds of songs. This project aims to further research in this field by training machine learning agents to analyze and observe common game elements within the rhythm game genre and use those to influence their actions.

StepMania is a popular rhythm game that challenges players to hit arrows on a dance pad or keyboard in time with music [3]. While many players enjoy the game, mastering it can take years of practice. To accelerate the learning process and push the limits of human performance, researchers and game developers have begun exploring the use of machine learning techniques to train agents to play StepMania.

One popular framework for training machine learning agents in games is Unity ML Agents, a toolkit developed by Unity Technologies that allows developers to create and train intelligent agents within the Unity game engine. Using Unity ML Agents, researchers and game developers can train agents to play StepMania by providing them with raw input data, such as the positions of arrows on the screen, and a reward signal that encourages the agent to hit the arrows accurately and in time with the music.

By training agents to play StepMania, researchers and game developers can explore new gameplay strategies and create more challenging and engaging experiences for players. Additionally, the use of machine learning techniques in StepMania has the potential to improve the accessibility of the game, as agents can be trained to play the game using alternative input devices or to adapt to the needs of individual players. Overall, the application of Unity ML Agents to StepMania holds promise for advancing the state of the art in both machine learning and game development.

II. Environment

StepMania is a cross-platform rhythm game that was initially released in 2001. Since then the game has evolved immensely to support a variety of rhythm-based game modes, control schemes, and provides a wide variety of quality-of-life features and analytical tools for their players. The software is also open-source and is available under the MIT License.

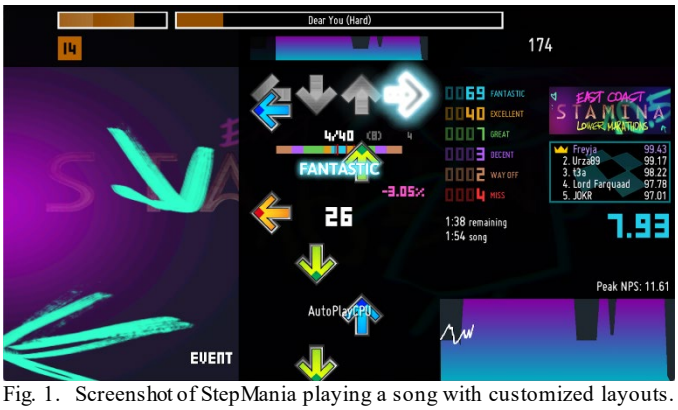

Fig. 1. Screenshot of StepMania playing a song with customized layouts.

A. Concept

Despite being around for over two decades, StepMania's game mechanics remain to be very straightforward to this day. Arrows scroll up from the bottom of the screen to the receptors (i.e., gray arrows in Fig. 1) where the player must press the corresponding arrow on their keyboard. The arrows that appear follow a predetermined sequence and match the rhythm of the song that plays along with the sequence. Each sequence is the equivalent of a single level in a video game, or what is commonly referred to as "charts" in the rhythm-game space.

B. Notes

A chart consists of a sequence of arrows, also known as "notes." There are multiple types of notes:

- **Tap:** A simple note that is hit when the corresponding key is hit at the right time.
- **Hold Head:** A tap note representing the start of a hold note.
- **Hold Tail:** The plain colored body that follows a hold head to indicate how long the corresponding key needs to be held down.

Fig. 2. StepMania screenshot with holds and taps.

Each note is color-coded based on its timing relative to the beat of the song. For instance, Fig. 2 shows red notes which represent quarter notes (4ths) and appear in line with the beat of the song. Blue notes represent 8ths, green represents 16ths, etc.

C. Scoring

Players are scored based on how accurately they can hit the arrows in time to the rhythm or beat of the song. Accuracy for each note is determined by Judgements: varying timing windows of increasing difficulty. A timing window for a single Judgement dictates the largest absolute value of the difference between when a note is supposed to be hit versus when the player actually hits the note. Smaller timing windows require more precisely timed inputs and as such reward better accuracy. The number of Judgements and timing windows can change based on player preference, but the standard is as follows:

- Marvelous
- Perfect
- Good
- Bad
- Miss

Marvelous has the tightest timing windows and rewards perfect accuracy (100%) whereas Miss has the widest timing windows and rewards no accuracy (0%).

A player's overall accuracy is calculated at the end of the song, where their efforts are given a letter grade and score based on their performance (like typical letter grading in academic structures).

III. Agent Attributes

This project uses Unity ML-Agents, a machine learning package, to program and train agents to play through various environments in Unity using machine learning. Unfortunately, StepMania was not built using Unity, nor does it have support for the game engine. As a result, the environment used in this study is UnityMania, a StepMania clone recreated in Unity by Kaj Rumpff [2]. The difference between these two environments is purely aesthetic and actual gameplay remains practically identical (i.e. UnityMania uses different Judgement titles from StepMania, but the timing windows remain the same).

A. Training

Multiple charts were used for training the agent, each at varying levels of difficulty (easy, medium, hard). A single episode will last from the beginning of the chart all the way to the end of the chart, where the environment is reset and a new episode begins. In each episode, the agent attempts to play through the chart as if a human player would, and their performance will influence the rewards and punishments the agent receives.

B. Actions

At each step, the agent's available actions are split between four discrete branches, one for each direction (left, down, up, right). In each branch, the agent has two available discrete actions, which represent pressing or releasing the corresponding key. The agent's actions along with its actions from the previous step will determine the state of each receptor, whether it is…

- Pressed
- Held
- Released

C. Observations

At any given step, the agent will "look ahead" and observe the first four incoming notes (notes that have already passed receptors and largest timing window are disregarded) in the sequence. Each note has three attributes that the agent observes:

- **Direction:** The direction/receptor the note is under (left, down, up, right).
- **Distance:** The distance between the note and the respective receptor (measured in seconds).
- **Note Type:** The type of note (tap, hold head, hold tail).

Altogether the agent makes up to twelve observations per step, with some exceptions being towards the end of a chart when there are less than four notes in the sequence to observe.

The following pseudocode is an example of how observations are made within a single step:

```
public override void
CollectObservations(VectorSensor
    sensor)
{
    foreach (NoteData data in LookAheadArrows)
    {
        sensor.AddObservation(data.distance);
        sensor.AddObservation((int)data.direction);
        sensor.AddObservation((int)data.type);
    }
```

```
}

protected NoteData[] LookAheadArrows
{
    get
    {
        int activeArrowsFound = 0;
        List<Note> noteList = // all notes in a
chart
        NoteData[] result
        for (int i in noteList.Count)
        {
            Note note = noteList[i];
            if (!note.IsActive)
                continue;

            float dist = note.DistanceInSec;
            if (dist < 0)
                continue;

            result[activeArrowsFound] = new
NoteData
            {
                distance = dist,
                direction = note.Direction,
                type = note.Type
            };
            activeArrowsFound++;
        }

        return result;
    }
}

protected struct NoteData
{
    /// <summary>
    /// Distance from the respective receptor.
    /// </summary>
    public float distance;

    /// <summary>
    /// The direction of the note's arrow.
    /// </summary>
    public Directions direction;

    /// <summary>
    /// The type of note.
    /// </summary>
    public NoteTypes type;
}
```

D. *Rewards and Punishments*

The agent is rewarded whenever an action results in hitting a note. The reward scales according to the quality of the Judgement received for the hit. For instance, rewards for the first few iterations of training maintain the following ratio:

- **Marvelous:** 10
- **Perfect:** 9
- **Great:** 7
- **Good:** 3
- **Bad:** 1

The maximum cumulative reward an agent can receive may vary across multiple charts, but the maximum cumulative reward an agent can receive remains consistent for every episode in the same environment/chart.

The following pseudocode is an example of how rewards and punishments are distributed in the latter iterations of training:

```
public void AddNoNoteReward()
{
    AddReward(noNoteReward);
}

public void AddJudgement(float
noteScore, Judgements
    judgement)
{
    switch (judgement)
    {
        case Judgements.marvelous:
            AddReward(noteScore);
            break;
        case Judgements.perfect:
            AddReward(noteScore /
2);
            break;
        case Judgements.miss:
            AddReward(missReward);
            break;
        default:
            AddReward(missReward /
noteScore);
            break;
    }
}
```

IV. Measuring Results

Training follows an iterative process, where the agent's training session is typically initialized from a generated model from a previous training session, generating multiple models as a result. For instance, the training session that would generate Model 2 may have been initialized by Model 1, Model 3 would be initialized from Model 2, and so on and so forth. This iterative process helps record the progress of training results while still being able to expand upon an existing neural network. Rewards, punishments, and environments may be adjusted and fine-tuned between models/training sessions. These adjustments help guide the agent's behavior toward the desired result.

The performance of a model is determined by the overall accuracy said model can achieve. A model with more consistent or higher accuracy results indicates an overall improvement in performance.

Results for each model have been compiled into three separate sets. Graph data for each set is available at the end of the report. For more details, visit the TensorBoard experiment for this project.

A. *Set 1: Models 1 – 5*

Total Steps	Accuracy Range
~3,000,000	**Easy:** 34 – 47%

Adjustments and Other Notes

- Models in this set were trained on Easy difficulty charts.
- **Models 4 – 5:** Judgement rewards were squared (reverted in future models).

B. Set 2: Models 6 –10

Total Steps	Accuracy Range
~10,000,000	**Easy:** 40 – 49% **Medium:** 43 – 48%

Adjustments and Other Notes

- Models in this set were trained on Medium difficulty charts.
- **Models 6 – 8:** Misses give harsher punishments than before.
- **Models 9 – 10:** Bad, Good, and Great Judgements now punish the agent instead of rewarding it.

C. Set 3: Models 11 – 14

Total Steps	Accuracy Range
~50,000,000	**Easy:** 43 – 47% **Medium:** 42 – 46% **Hard:** 44 – 50%

Adjustments and Other Notes

- **Models 11 – 14.5:** Trained on Medium difficulty charts.
- **Rest of Model 14:** Trained on Hard difficulty charts.
- Perfect rewards were reduced by 50% to put more emphasis on the Marvelous reward.

D. Conclusion

Overall, the training agent could not achieve greater than 50% accuracy on any chart across all difficulties, however, training the agent on more difficult charts greatly improved the agent's performance for easier difficulty charts.

V. Future Work

StepMania Agents has shown some promising results in the agents being able to consistently achieve over 40% accuracy across multiple charts after just one week of training. Nevertheless, this project is just the beginning, as there is still much more work to be done before an agent is able to achieve a perfect (100%) score on any chart. Some potential avenues for future work can be focused on fine-tuning and experimenting with different reward distributions (i.e. bonus rewards for consecutive hits, separate reward weights for different note types, etc.), more supervised training, and configuring agents to play StepMania with their feet instead of simple keyboard inputs.

References

[1] Unity Technologies. "Unity-Technologies/ML-Agents." GitHub, 15 Apr. 2019, github.com/Unity-Technologies/ml-agents. Accessed 26 Apr. 2023.

[2] Rumpff, Kaj. "Rumpff/Unitymania." GitHub, 31 Dec. 2022, github.com/rumpff/unitymania. Accessed 26 Apr. 2023.

[3] StepMania Team. "StepMania." GitHub, 2015, github.com/stepmania/stepmania. Accessed 26 Apr. 2023.K. Elissa, "Title of paper if known," unpublished.

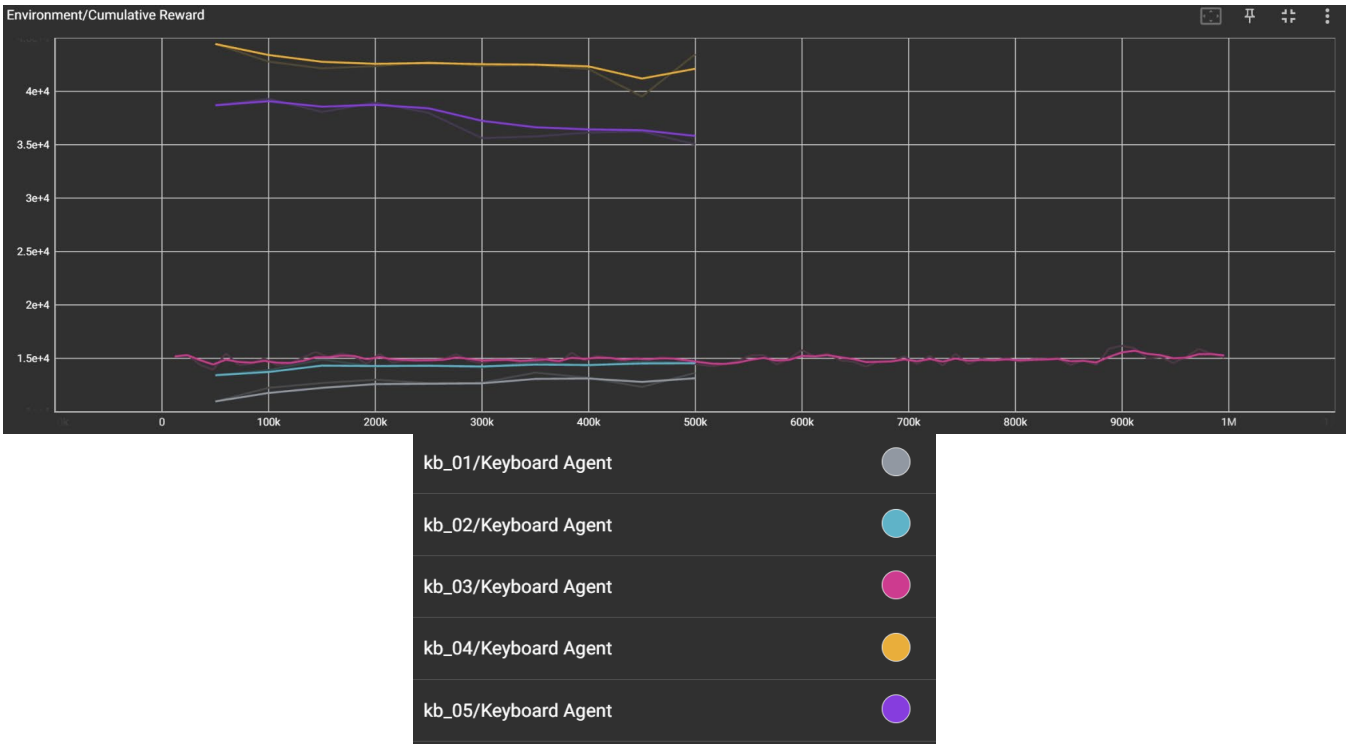

Fig. 3. Models 1 – 5 Cumulative Reward Results

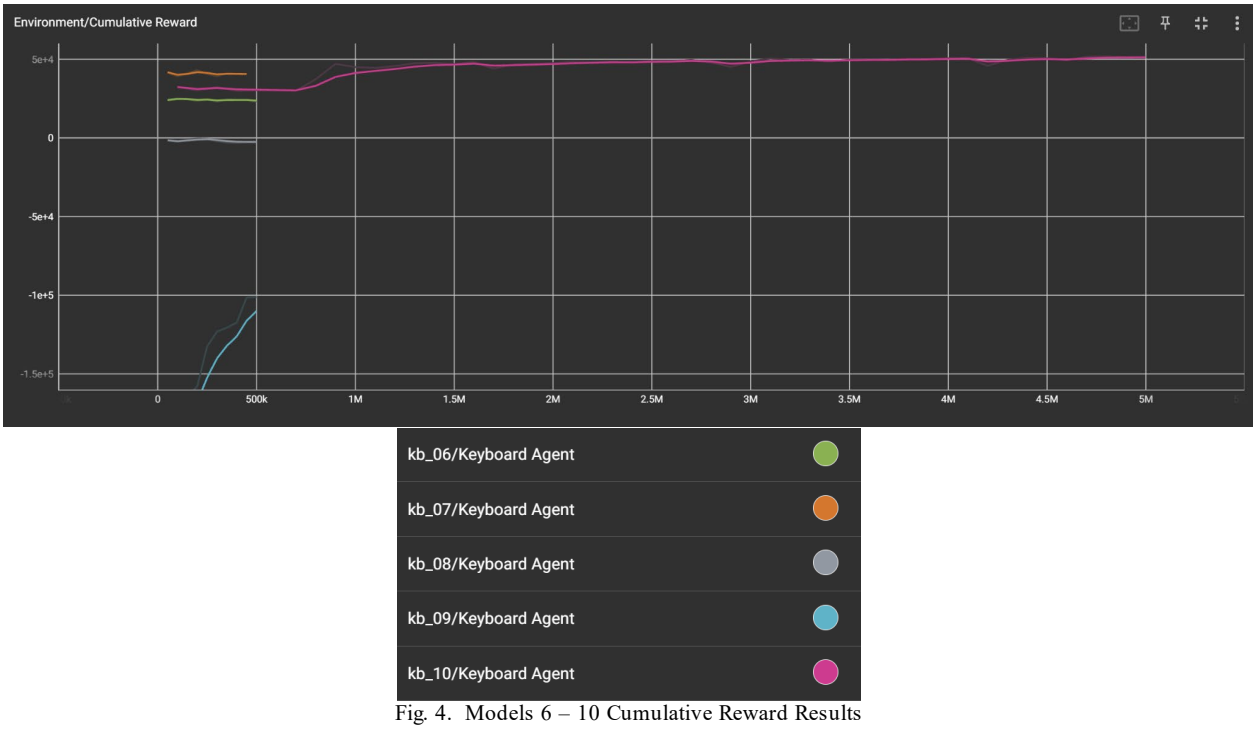

Fig. 4. Models 6 – 10 Cumulative Reward Results

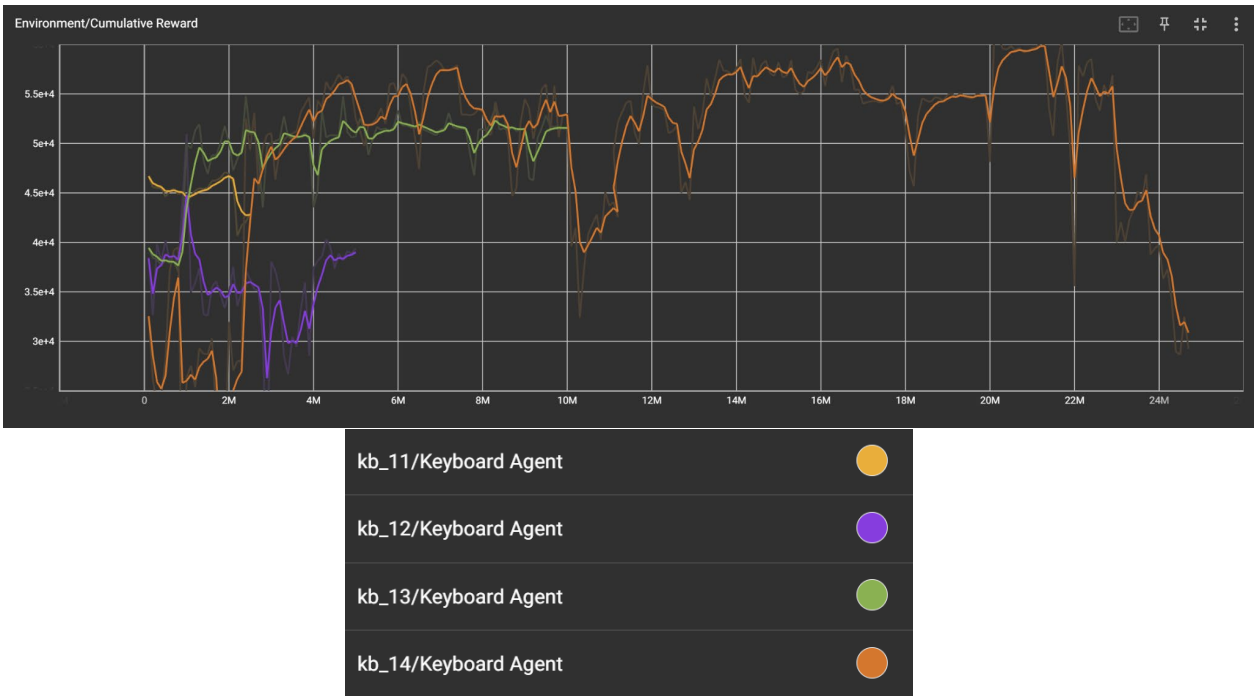

Fig. 5. Models 11 – 14 Cumulative Reward Results