CSCI 1430 Final Project Report:
AI Rhythm Game Player: DJMax with Computer Vision

Team name: Lixing Wang, Jonie Nishimura, Tiffay Gao. TA name: Lisa Baek. Brown University

Abstract

This project focuses on making a lightweight neural network system capable of playing the rhythm game DJMax Respect V in real time using grayscale screen captures. Our model combines spatial feature extraction via a modified ResNet-18 and temporal modeling with a Gated Recurrent Unit (GRU) to predict key presses frame-by-frame. We also designed a dynamic loss function based on Binary Cross-Entropy that emphasizes temporal precision and label transitions, addressing the sparsity and imbalance of keypress label data. Evaluations showed a precision per key greater than 80% (up to 99. 8%), and real game tests demonstrated performance that exceeded human scores on difficult songs on which the model was not trained. This project highlights the challenges and feasibility of using computer vision and sequence modeling for real-time game control in visually complex environments.

Please view the accompanying video demos of our model in action at: *Google Drive Link*

1. Introduction

Modern rhythm games are one of the difficult problems for traditional computer vision methods to excel at because they often involve extensive visual effects and require real-time predictions. In this project, we will develop a light-weight mixed-architecture neural network to play a visual-heavy rhythm game *DJMax Respect V* in real time.

2. Related Work

We chose our primary framework as `PyTorch` [5] for its cross-platform compatibility and the freedom to experiment with different model designs. We also used ResNet-18 [6], which allowed us to extract spatial features from video frames with low computation latency. We used a Gated Recurrent Unit (GRU) [1] to capture temporal information. We developed our own dynamic loss function based on Binary Cross-Entropy (BCE) loss tailord to this task. To sync the game with our model, we used `mss` to capture the screen and `pynput` to record and perform key presses and releases.

Prior research on rhythm games and note recognition informed our model design and evaluation strategies. Kondo's work on rhythm game design offered insight into how timing, feedback, and input modalities affect gameplay [3]. This guided our decisions around temporal labeling and model responsiveness. Additionally, Tanaka and Nakagawa explored real-time note recognition using convolutional recurrent networks [4], which inspired our hybrid approach of combining convolutional and recurrent layers.

3. Method

Our goal was to create a model to play the game by predicting keys to press based on visual input in real-time. This section will discuss approaches to data collection, data processing, model architecture, training and real-time inference.

3.1. Data Collection

The data collection was done by a group member, who recorded 31 videos of real game play at 30 FPS, alongside with keystrokes, with `mss` and `pynput`. The code used for capturing is in `capture.py`. The code captures a predefined region on the screen, and saves the video and keystrokes to `.mp4` and `.txt` respectively. Video files are downsampled by a factor of 2. Each line of a `.txt` key log file is structured as `[timestamp] [framestamp] [action] [key]`. Timestamp is the exact time the action was performed, and framestamp is the exact frame in the corresponding video when the action was performed. Key takes 6 values as the game uses 6 keys, which are left shift, a, x, ., ', and right shift. The final dataset consisted of 31 videos and key logs, each has duration of around 2 minutes.

3.2. Data Processing

The videos and key logs underwent different data processing in `preprocess.py`. We first validated the key logs to make sure each press is paired with a release, and fixed data by manual inspection if needed. The key strings were then mapped to integers from 0 to 5, converted to `NumPy` arrays of shape `(T, 6)`, and saved to `.npy` files for easy access. The videos where trimmed and cropped to include only active gameplay and interested screen region. After preprocessing, videos have resolution 224x128. When training

and inferencing, this resolution was futher reduced to 112x64 for performance consideration. We used the `.mp4` format and decided to decode the video in real-time during training, because our system was bounded by IO performance.

3.3. Model Architecture

The most challenging part is the design of model architecture. We used first few layers of ResNet18 to extract CNN features for each frame, obtaining 128 feature maps for each. We disabled every pooling layers and reduced vertical strides of convolution layers to keep vertical resolution. Each feature map obtained was then divided horizontally into 4 equal regions each dedicated to a track. The 1st and 2nd regions were grouped and copied as the 5th region, and the 3rd and 4th regions were copied as the 6th region, because each shift key (called "side key" in the game) spans across two tracks. The following snippet defines the regions:

```
self.slices = [
    (0, w//2),              # Key 0
    (0, w//4),              # Key 1
    (w//4, w//2),           # Key 2
    (w//2, w//2 + w//4),    # Key 3
    (w//2 + w//4, w),       # Key 4
    (w//2, w)               # Key 5
]
```

where `w` is width of the feature maps.

For each region, we used a Conv2D layer to compress the width and channel dimensions so each region has shape `(T, 1)` at the end. Three different kernels were used to better capture nuances in features. Key 0 and 5 (side keys) used a 1-by-(w/2) convolution, key 1 and 4 (white keys) used a 1-by-(w/4) convolution, and key 2 and 3 (orange keys) use a separate 1-by-(w/4) convolution.

```
self.skey_extract = nn.Conv2d(C, 1, (1, W//2))
self.wkey_extract = nn.Conv2d(C, 1, (1, W//4))
self.okey_extract = nn.Conv2d(C, 1, (1, W//4))
```

We then concatenated 6 sequences and obtained a tensor of shape `(6, T, self.feature_H)` where `self.feature_H` is the height of the feature maps. The tensor was then fed to a 3-layer GRU, which returns a tensor of shape `(6, T, self.hidden_size)`. A `nn.Linear` layer was applied to compress the the last dimension from `self.hidden_size` to 1, which gave us an output of shape `(T, 6)` after permutation. After `Sigmoid`, each vector of length 6 was the probabilities of whether a key is pressed for each key each frame.

3.4. Training and Loss Function

We designed a special dynamic loss function tailored to this task, a distance-weighted BCE loss that emphasizes temporal changes in labels. The model was 1) punished less for not responding on the exact framestamp while 2) punished more when it does not respond to sudden changes in the labels. The former was done by convolving the ground truth label sequence with a 1D "triangle"-like kernel, a weighted moving average for each key track. (See `moving_avg()` in `train.py`.) The latter was done by calculating the discrete derivative of the label sequence and using it as weights to increase loss responses. (See `dynamic_loss()`.)

The rhythm game involves "tap" keys much more than "hold" keys, so keys are not pressed for most of the time. To mediate this uneven distribution of labels, we set `pos_weight` argument to `nn.BCEWithLogitsLoss`, increasing loss for missing a press by 25 times than for missing a release. This way, we avoided the model to learn to predict releases (outputting zeros) all the time.

For reference, BCE loss is defined as:

$$\mathcal{L}_{\text{BCE}} = -\left[y \cdot \log\left(\sigma(x)\right) + (1-y) \cdot \log\left(1-\sigma(x)\right)\right] \quad (1)$$

and Sigmoid is defined as:

$$\sigma(x) = \frac{1}{1+e^{-x}} \quad (2)$$

where x is the raw model output, $y \in \{0, 1\}$ is the ground truth binary label, and $\sigma(x)$ is the sigmoid activation function.

In each training iteration, we fed 100 frames using sliding window to reduce memory consumption. The training was done on Google Colab and, primarily, a Windows desktop with Docker environment.

3.5. Real-time Inference

Real-time inference is the key to the task. The interested region of the screen was captured similarly as in `capture.py`, then downscaled to 112x64 and converted to grayscale. Although the training dataset was colored videos, a mismatch in color profiles of `.mp4` and live screen capture had led the model unable to recognize the correct orange color. Grayscaling the input solved this problem to a satisfying degree. We implemented `self.inference_forward()` method dedicated to inference. (See `model.py`.) Each time, only 1 new frame was fed. To keep track of temporal information, `self.last_hidden` records the last hidden state of GRU, and was fed in with the new frame. This allowed for ∼45 FPS inference rate on a M3Max chip while running the game. `self.last_hidden` was reset every 100 iterations to mimic the training process. For the probability vector output, we threshold it on 0.6 to determine if a key should be

Evaluation Metrics	
Key 0 Accuracy:	0.9986
Key 1 Accuracy:	0.8385
Key 2 Accuracy:	0.8270
Key 3 Accuracy:	0.8690
Key 4 Accuracy:	0.8531
Key 5 Accuracy:	0.9975
Micro F1:	0.6665
Micro Precision:	0.5951
Micro Recall:	0.7572

Table 1. Evaluation of the Model Predictions vs Human Labels

pressed. 0.6 was a hyperparameter chosen to reduce false positives. We used 0.5 for fair testing.

4. Results

The final model was a light-weight model with 3M parameters and trained for 14 epochs. The testing involves offline tests (feeding recordings 100 frames at a time and comparing with labels) and online tests (testing on real game and recording the scores).

For offline tests, we generally found that the accuracy for each key stayed within the 80-100% range, with most being above 85%. The highest accuracy was seen on keys 0 and 5 where the accuracy was above 99%. Our results also showed that our micro F1 score was around 70%, the micro precision was around 60%, and the micro recall score was around 75% (Table 1). We were able to generate visualizations that plotted our model's predictions vs ground truth (Figure 1). Our figure contains the predictions vs ground truth for each of the 6 keys, and each of the graphs mostly showed overlapping lines with a few exceptions, which shows that our model's predictions were mostly accurate and aligned well with the labels.

For online tests, the model was able to achieve at least 95% game score on most of the untrained songs, and could reach up to 97.5% on difficult ones. Occasionally, it beat the highest record of the member who generated the training data by around 1%. Note that the online test songs are generally harder than songs in the training dataset. Here is a link to demo videos of the model playing in real time on difficult, untrained songs. Check how it performs on the song *L*, one of the most difficult songs in the game with fast-paced, overwhelmingly many tap notes.

4.1. Technical Discussion

Our method raises some interesting questions such as the purpose of dividing the feature maps into separate regions/tracks. This creates a specific area for each track, which allows the model to create special convolutions for each key type. While this is a suitable method for DJMax and allows for higher accuracy and performance, but this makes our model less applicable for other games, which do not have the same layout as DJMax. We chose to do this separation on feature maps, not the video input, to reduce computation costs. However, splitting the screen before passing to the feature extractor could potentially further reduce interference between tracks, which were observed to some degree in model's responses in offline testing. Our custom dynamic loss function improves the model performance, but it is more computation intensive and prolonged the training. More importantly, training on colored videos while inferencing on grayscale inputs is only a temporary fix for not recognizing the correct orange color. The model did struggle to tell blue side keys apart from normal hold keys, which share many common features only except for the color. Inferencing on colored inputs would definitely improve online testing results. Lastly, even with grayscale inputs, the real-game performance is actually better than the results of offline testing, because different metrics were used. In real game, there is tolerance for small out-of-sync of hitting a key, while in offline testing, the model prediction is compared with the labels frame-to-frame in a more strict manner, and the labels are not perfect gameplay either. With all these intricacies, noting that we tested the model on songs much harder than those provided for training, our model did work really well in real gameplay and was able to achieve good scores.

To resolve imperfection in training data, one could utilize traditional computer vision techniques to obtain perfect gameplay data based on video inputs alone, as the falling notes *are* key triggers required. Althought this wasn't done due to short amount of time available, extracting perfect key triggers can drastically improve the quality of training data, and allow for training on much more difficult songs. Indeed traditional methods might be able to respond correctly to notes and play the game, but we argue that using machine learning models has a performance advantage for real-time inference, without compromising precision, because we can make use of GPU.

5. SRC Critique Response

We appreciate the thoughtful critique of our project and the careful attention given to core SRC principles. Below, we address each of the key points raised and describe how we have incorporated this feedback into our thinking and development process.

5.1. Project Overview and Potential Impact

Our project explores the use of machine learning to map rhythm game footage to game play input logs. While this is pretty niche and doesn't seem to have an immediate societal impact, this project does intersect with broader societal concerns around automation, accessibility, and fair use in

digital spaces. Any machine learning systems that imitate or automate human behavior can have powerful applications and unintended consequences.

5.2. Accessibility and Inclusive Design

We agree with the concern about accessibility that was raised. Rhythm games should be enjoyable and accessible to all players, regardless of ability. Rhythm games (and video games in general) are often seen as meritocratic environments where everyone starts on equal footing. Introducing AI generated input could be seen as disrupting that balance. If misused, such technology could give some players an unfair advantage, effectively excluding others from competitive spaces.

However, we argue that the same technology can be designed to include rather than exclude. For players with motor impairments or limited mobility, rhythm games can be difficult or even impossible to access. By leveraging AI to interpret visual cues and generate input, we can enable alternative control schemes making these games more accessible to a broader range of players.

As developers, we must acknowledge the dual-use nature of machine learning technologies and prioritize use cases that promote accessibility, creativity, and inclusion over those that undermine fairness.

5.3. Diversity in Training Data and Player Representation

Another critique is aimed on limited training data diversity. While our current model is focused on video to key log/label prediction rather than player identity, we recognize that expanding the variety of game play styles in the training data could strengthen generalization. At present, our training model is based on one of our team member's (Lixing's) playing data, but our model does not encode player identity or physical characteristics. It operates purely on visual and temporal patterns from the game play screen. Nonetheless, we agree that ensuring equitable performance across different play styles is essential to fairness and will be an important step in maturing this project if it goes beyond the scope of research and academia.

5.4. Cheating and Integrity

Another critique is the risk of misuse especially in competitive or online contexts where automated game play may be used to cheat or undermine fairness. In multiplayer or leader board driven communities, such tools could devalue human skill, and disrupt the rules that uphold fair competition. We therefore emphasize that our work is strictly intended for academic and exploratory use, not deployment in competitive scenarios. We hope that through our research, game developers can better understand how AI sees and reacts with their visual designs. In this way, we hope to promote future anti-bot measures.

We also acknowledge that as models improve, it may become more difficult to distinguish between human and machine generated input. Developers must take a proactive role in mitigating these risks through platform level policies, such as watermarking AI generated inputs or tracking non-human behavior patterns. Our project could help inform the design of such detection tools by modeling how AI interacts with game interfaces.

5.5. Cultural Sensitivity and Game Context

We appreciate the concern regarding cultural representation and potential appropriation in rhythm games. However, we would like to clarify that our project does not involve creating original rhythm game content, such as music, visuals, or themes. Instead, our work focuses on building an AI model that can interpret and replicate player inputs based on existing game play footage.

The game we used for our dataset, DJMax, is a rhythm game series developed primarily by Neowiz MUCA, a South Korean studio. As such, the cultural content in the game is created and curated by developers native to that region. We are not contributing to the artistic or musical direction; rather, we are working within the constraints of preexisting media.

Our choice of DJMax was not based on aesthetics or cultural themes but on technical feasibility. DJMax supports key-based inputs that are straightforward to simulate and log on a computer, which is essential for our project. Other popular rhythm games, such as Project:sekai and Osu!, often require drag, flick, or touchscreen-based interactions, which are significantly more complex to simulate and automate using keyboard based AI systems. Our goal is to explore interaction patterns between visual signals and game play mechanics, not to encode or reproduce any cultural content.

6. Conclusion

We designed and created a light-weight mixed-architecture neural network that excels at the rhythm game DJMax Respect V. Overcoming the challenges, we successfully trained the model to consistently score around 94-98% on untrained songs of difficulties in real time. This project is a good basis for exploring temporal precision, multi-label classification, and real-time inference. All of these are important and useful in computer vision and our model shows that it is possible to design and train a model to watch fast visual inputs and make highly precise predictions. Looking forward, this project can be generalized and applied to other rhythm games that make use of fast and precise output such as Piano Tiles or even games in other genres. Our project also has potential to be adjusted to aid those with physical disabilities play rhythm games. Lastly, our project could also

inspire other tasks or projects that utilize temporal precision and real-time inference such as autonomous driving.

References

[1] GeeksforGeeks Authors. Gated recurrent unit (gru) networks. https://www.geeksforgeeks.org/gated-recurrent-unit-networks/, 2023. Accessed: 2025-05-14. 1

[2] GNU Project. The gnu general public license (gpl), n.d. Free Software Foundation.

[3] Kosuke Kondo. Rhythm game design: A study of timing, feedback, and input. *Journal of Game Design & Development Education*, 2(1):1–10, 2019. 1

[4] Ryo Tanaka and Masataka Nakagawa. Real-time note recognition in music games using convolutional recurrent neural networks. *Proceedings of the International Conference on Multimedia & Expo Workshops (ICMEW)*, pages 1–4, 2017. 1

[5] PyTorch Team. Pytorch: An open source machine learning framework. https://pytorch.org/, 2024. Accessed: 2025-05-14. 1

[6] PyTorch Team. Resnet-18 — torchvision main documentation. https://docs.pytorch.org/vision/main/models/generated/torchvision.models.resnet18.html, 2024. Accessed: 2025-05-14. 1

Appendix

Team contributions

Lixing Wang I was responsible for designing model architecture and loss function, writing `capture.py` and collecting data, implementing half of `preprocess.py`, and testing the model in real gameplay. I also implemented `dataset.py` based on what Tiffany had done in her `VideoDataset()` class. I was involved in bug fixes and writing this report, especially Section 3.

Tiffany Gao I trained and tested the original model, and contributed to its design and evaluation writing `model.py`, `train.py`, and `test_model.py`. I wrote the `VideoDataset()` class to load game play videos and corresponding key logs which will further be implemented by Lixing and also wrote the code to generate graphs and evaluation tables for visualization. I contributed to some bug fixes and also writing this report more specifically the abstract and the SRC Critique Response.

Jonie Nishimura I implemented some of the preprocessing functions including `validation()` and `trim_video()` to validate key presses/releases and crop gameplay videos. I also worked on the inference pipeline and wrote the `play.py` file. I then converted `play.py` from using TensorFlow to PyTorch for consistency. I designed around half of the poster, focusing on the motivation, goals, and methodology. Lastly, I contributed to the final report by writing the Introduction, Related Work, Method, Results, Technical Discussion, and Conclusion sections.

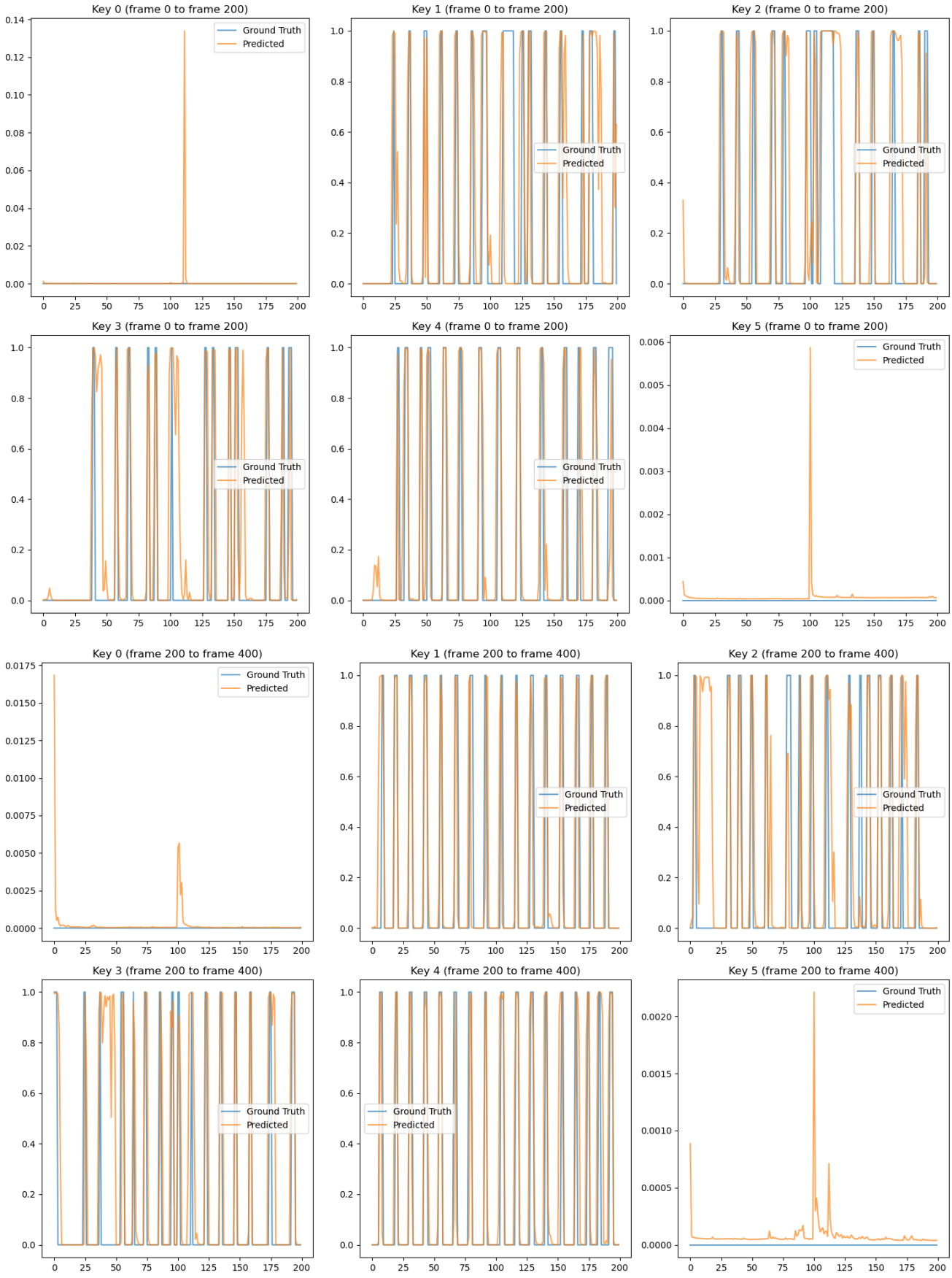

Figure 1. Model Predictions vs Ground Truth on a Sample Test Video (0 to 400 frames).