DeepPong: A Deep Learning Approach to Real-time Frame Generation for Pong

Eric Brown
California State University, Sacramento
Sacramento, California, USA
ebrown5@csus.edu

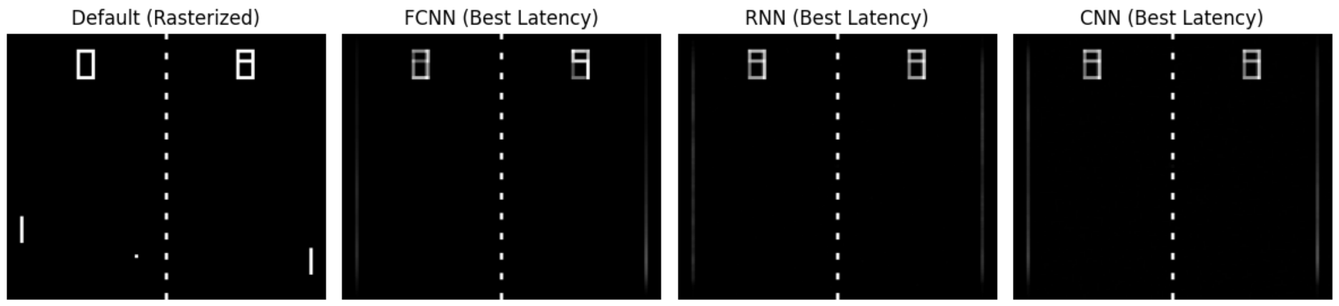

Figure 1: Comparison between the default rasterized Pong frame and frames generated by the lowest-latency neural networks of different architectures. Low-latency models in particular struggle to accurately render dynamic objects like the ball and paddles.

Abstract

Real-time rendering in computer graphics traditionally relies on rasterization techniques, which use approximations and heuristics to simulate realistic visuals. As deep learning continues to revolutionize various domains, exploring its potential for graphics rendering could open up new possibilities for visual fidelity and efficiency. This project investigates the feasibility of rendering entire computer game frames in real-time using deep learning, eliminating the need for conventional rasterization pipelines. Specifically, I implemented a version of Pong in Python, and trained a variety of neural networks to predict and generate game frames based on the game state in real-time. Initial results demonstrate that the deep learning-based renderer achieves highly accurate outputs at maintainable frame rates, suggesting a promising alternative to traditional rendering methods in simple game environments. However, the models lacked the ability to predict important pixels associated with the ball, making them not quite usable in their current state for an actual game of Pong.

Keywords

Deep learning, Pong, Real-time graphics, Frame generation, Neural network

1 Introduction

The modern graphics pipeline relies primarily on rasterization to simulate a detailed approximation of a particular scene. While effective, these methods can become increasingly complex and computationally expensive when rendering hyper-realistic visuals, especially in real-time. Exploring deep learning's potential in computer graphics applications could lead the way to alternative rendering techniques that emphasize visual fidelity and simplicity.

This project explores the feasibility of using deep learning to render computer game frames in real-time, bypassing traditional rasterization techniques. To achieve this, I implemented a version of Pong in Python and used it to generate a dataset to ultimately train multiple neural networks in predicting frames from a corresponding game state. The following is a short list of my contributions to this particular area of interest:

- Implemented a deep learning-based renderer for Pong in Python.
- Trained multiple neural network architectures to generate game frames from the game state.
- Demonstrated near-perfect predictions at reasonable frame rates.
- Identified challenges in maintaining adequate FPS and accurately rendering dynamic elements like the ball.
- Proposed potential solutions to refine neural network architectures and training processes for this particular problem.

The remainder of this paper is organized as follows. Section 2 formally defines the problem of using deep learning to generate Pong frames in real-time. Section 3 describes the design and architecture of the models evaluated, including FCNN, RNN, and CNN configurations. Section 4 presents the methodology, experimental setup, evaluation metrics, results, and potential improvements. Section 5 reviews relevant prior research in real-time neural network rendering. Section 6 summarizes the findings of this project. Section 7 outlines the division of work for the project. Section 8 highlights the learning experience and experience gained from the project. Finally, the appendix provides supplementary figures, including the confusion matrices and ROC curves for the best models.

2 Problem Formulation

The primary problem addressed in this project is the real-time rendering of computer game frames using deep learning techniques. Traditional rasterization methods, while effective, can be computationally intensive in certain instances and fail to capture the desired level of detail in certain environments. While other works have attempted to predict frames based on user input and previous frames, I was unable to locate any past examples of applying supervised learning *directly* to the internal game state.

The focus of this project is limited to the game Pong. Pong is a classic two-dimensional video game that was originally released by Atari in 1972 for the arcade that simulates table tennis [6]. It features two vertical paddles (vertical bars), each controlled by a player, and a ball that moves across the screen. The objective is to use the paddles to prevent the ball from passing one's side of the screen while attempting to score points by getting the ball past the opponent's paddle. Pong is characterized by its simple visuals, black and white color scheme, and basic physics, making it an ideal target for studying frame generation and real-time rendering tasks.

In this project, I attempted to use deep learning to generate graphics for Pong in real-time. I trained a multitude of networks based on the current state of the Pong game that includes the positions of the paddles, position and velocity of the ball, and the score of each player. These values were normalized, and then sent through a neural network that predicts a single value for each pixel in the frame. This problem was treated as a binary classification problem because each resulting probability between 0 and 1 could be easly converted into a pixel color: 0 being a completely black pixel and 1 being a completely white pixel. Each network was responsible for predicting 30,720 unique values for a single frame which was based on Pong's resolution for the Atati: 160 x 192 [5]. These predictions were then drawn on the screen and shown to the user. This process was then repeated several times a second in order to achieve a stable framerate.

3 System Design

To evaluate the ability of neural networks to accurately represent the internal state of a game, three types of networks were tested: fully-connected neural networks (FCNNs), recurrent neural networks (RNNs), and convolutional neural networks (CNNs). For each network architecture, a set of hyperparameters was defined, including the number of layers, the number of neurons per layer, the choice of optimizer, and so on. Every combination of these parameters was then used to train and evaluate a set of distinct models. These parameters were chosen to produce models deep enough to meaningfully learn the target output whilst being shallow enough to train in reasonable time and maintain a low latency per prediction.

3.1 FCNN Architecture

To test the effectiveness of FCNNs, several model configurations were defined and exhaustively tested by evaluating all combinations of the following options:

- **Number of hidden layers and neurons per layer:**
  - No hidden layers
  - 1 layer: 128 neurons
  - 2 layers: 128 and 256 neurons
  - 3 layers: 128, 256, and 512 neurons
  - 4 layers: 128, 256, 512, and 1024 neurons
- **Activation function for each hidden layer:**
  - None
  - ReLU
  - Tanh
- **Optimizer:**
  - Adam
  - AdamW
  - SGD

Each FCNN concluded with an output layer containing 30,720 neurons and the sigmoid activation function. Taking the Cartesian product of all the configuration options resulted in 45 distinct FCNN architectures, each of which was trained and evaluated.

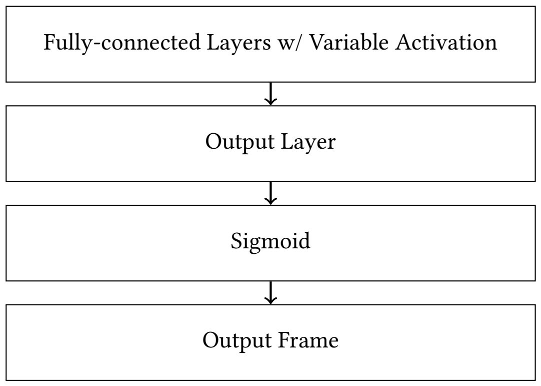

Figure 2: Architecture of the fully-connected neural networks (FCNN) used for frame generation.

3.2 RNN Architecture

To test the effectiveness of RNNs, several model configurations were defined and exhaustively tested by evaluating all combinations of the following options:

- **Bottom recurrent layers (LSTM):**
  - 1 layer: 64 units
  - 2 layers: 64 and 256 units
  - 3 layers: 128, 256, and 512 units
- **Top fully-connected layers:**
  - None
  - 2 layers: 128 and 512 neurons
  - 1 layer: 1024 neurons
  - 1 layer: 2048 neurons
- **LSTM dropout rate:**
  - 0.0
  - 0.1
- **Optimizer:**
  - Adam
  - AdamW
  - SGD

Each RNN model concluded with an output layer containing 30,720 neurons and a sigmoid activation function. Taking the Cartesian product of all the configuration options resulted in 72 distinct RNN architectures, each of which was trained and evaluated.

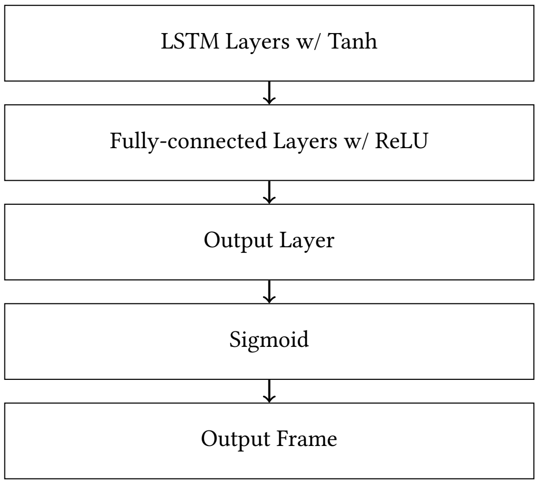

Figure 3: Architecture of the recurrent neural networks (RNN) used for frame generation.

3.3 CNN Architecture

To test the effectiveness of CNNs, several model configurations were defined and exhaustively tested by evaluating all combinations of the following options:

- **Bottom convolutional layers:**
  - 1 layer: 32 filters
  - 2 layers: 16 and 32 filters
  - 2 layers: 32 and 64 filters
- **Top fully-connected layers:**
  - No top layers
  - 2 layers: 128 and 256 neurons
  - 3 layers: 64, 128, and 256 neurons
  - 1 layer: 512 neurons
- **Pooling size:**
  - 3×3
  - 5×5
- **Kernel size:**
  - 2×2
  - 3×3

Each CNN model concluded with an output layer containing 30,720 neurons and the sigmoid activation function. All CNN models were optimized using AdamW. Taking the Cartesian product of all the configuration options resulted in 48 distinct CNN architectures, each of which was trained and evaluated.

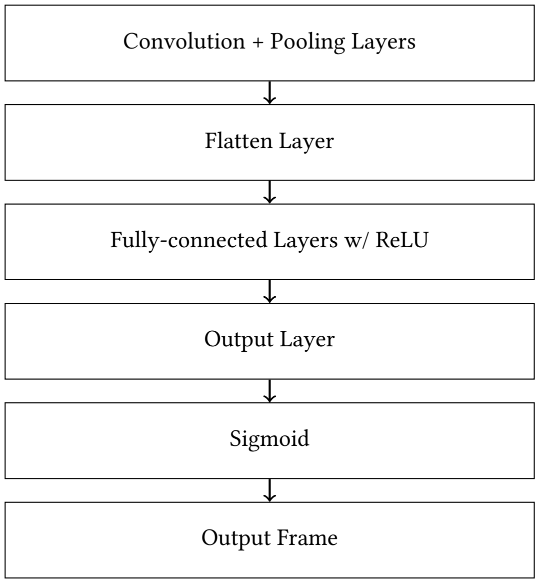

Figure 4: Architecture of the convolutional neural networks (CNN) used for frame generation.

4 Experimental Evaluation

This section presents the experimental setup used to train and evaluate the proposed models. It describes the dataset generation process, the experimental settings, the evaluation metrics applied, the comparison of different neural network architectures, and outlines some potential improvements.

4.1 Methodology

4.1.1 Dataset Generation. To generate a dataset for training and evaluation, the game state was randomized by assigning random values to the paddle positions, ball position, and ball velocity. The game was then advanced for several time steps, capturing the rasterized frame at each step using the traditional graphics pipeline. For each time step, user input was simulated: if the ball was higher than the center of a paddle, the paddle moved upward; if it was lower, the paddle moved downward. These steps produced data that closely resembled realistic gameplay.

The randomization process was repeated across all possible score combinations. Since the game ends when a player reaches a score of 9, there are a total of 99 unique score combinations. Therefore, the total number of samples generated is given by the following:

$$\text{Samples} = 99 \times \frac{\text{Random States}}{\text{Score Combination}} \times \frac{\text{Time Steps}}{\text{State}}$$

In this study, 50 random states were generated for each score combination, each over 5 time steps, resulting in a total of 24,750 samples. These were split into a training set (19,800 samples) and a testing set (4,950 samples).

4.1.2 Experimental Setting. Training was performed with early stopping and model checkpointing applied to avoid overfitting. Model evaluations were performed on the testing set that comprised of 20% of the total data selected randomly. All training was conducted using Google Colab, utilizing an NVIDIA A100 GPU to accelerate model training.

4.1.3 Evaluation Metrics. The following metrics were used to evaluate and compare the performance of the different models:

- **Weighted F1-score**: Most accurate across all pixels.
- **Latency**: Average time taken to predict a single frame, measured across the first 10 samples of the testing set.

In addition to these primary metrics, confusion matrices, receiver operating characteristic (ROC) curves, and classification reports were generated for the best-performing models.

4.1.4 Comparison Methods. For each architecture, two models were selected for detailed evaluation: the model achieving the highest weighted F1-score (best performing or most accurate) and the model achieving the lowest latency (fastest). The best models for each category were then evaluated through a combination of confustion matrix, classification report, and receiver operating characteristic (ROC) curve.

4.2 Results

The following results summarize the performance of the best models selected based on highest weighted F1-score and lowest latency. Each architecture—FCNN, RNN, and CNN—was evaluated on its ability to accurately predict frame pixels while maintaining real-time inference speeds. The tables below present a comparison of these models across both criteria.

Model Type	F1-score (Weighted)	Latency (ms)
FCNN	0.99326	97.69
RNN	0.90933	70.66
CNN	0.90909	99.27

Table 1: Best models based on highest weighted F1-score.

Model Type	F1-score (Weighted)	Latency (ms)
FCNN	0.91336	71.79
RNN	0.01193	66.43
CNN	0.90139	79.12

Table 2: Best models based on lowest latency over 10 predictions.

These results demonstrate the potential of using neural networks to generate entire computer game frames in real time. Unfortunately, not all pixels are created equal, and none of the models successfully rendered the ball consistently. Some frames exhibited flickering white pixels in random locations, suggesting that the networks attempted to capture the ball's position but failed to do so reliably. One explanation for these results is that the ball is not captured by the model because–being only 4 pixels in size–the ball has a very small impact on the overall loss. Coupled with the fact that the ball moves quickly around the screen makes it difficult for the model to learn how to accurately represent the ball. This ultimately renders each model unsuitable for practical usage, as the ball is arguably the most critical component to render accurately.

Despite this limitation, the models achieved notable successes. In particular, The FCNN models consistently rendered the score without artifacts and displayed the paddles with high fidelity. In contrast, both the RNN and CNN architectures struggled to achieve comparable results. One possible explanation is that previous state data offers little or no benefit to the prediction process because traditional games rasterize the entire frame based solely on the *current* game state: no previous state is needed. Additionally, reshaping the dataset to fit the input requirements of recurrent and convolutional networks reduced the number of training samples, which may have further impacted their performance.

It is also possible that the simplicity of Pong inherently favored fully-connected architectures; FCNNs may excel when tasked with rendering relatively simple scenes. In contrast, more complex or 3D games might benefit from RNN or CNN architectures, where multiple game states could potentially enhance camera effects like motion blur. Further investigation is needed to determine whether incorporating previous game states could meaningfully enhance frame prediction tasks in more complex environments.

In terms of latency, most aritectures achieved similar results. Surprisingly, the lowest latency came from an RNN even though RNNs typically process more data per frame. However, this came at a significant cost to accuracy, as this model only achieved an F1-score of 0.01193. This result appears to be an outlier of this particular run because the next fastest RNN (not shown) achieved and F1-score of 0.90046 with a latency of 66.70 ms–a difference of only 0.27 ms. This suggests that the low-latency, low-accuracy model was likely due to random chance rather than a meaningful trade-off.

Each model averaged approximately 15 frames per second, which makes them almost usable in a real-world application. However, this performance falls short of the 30 frames per second commonly considered the minimum threshold for smooth gameplay by modern standards. Potential solutions to address this bottleneck are discussed in Section 4.3.2.

4.3 Potential Improvements

Several improvements can be further explored to enhance model accuracy and latency:

4.3.1 Accuracy Improvements. While the current models demonstrated promising results overall, several methods may improve the previously-noted issue with rendering the ball correctly:

- **Larger Dataset:** Expanding the dataset by generating more random game states could further reduce overfitting. Because the dataset is already generated deterministically and algorithmically, there is theoretically no limit to the size of the dataset that can be generated.
- **Custom Loss Function:** Developing a custom loss function that assigns greater weight to important visual elements—such as the paddles, the ball, and whites pixel in general—could encourage the models to prioritize the most important areas of the frame. Unfortunately, there was insufficient time to fully implement this approach.
- **Alternative Architectures:** Exploring different or more specialized architectures might yield improved performance.

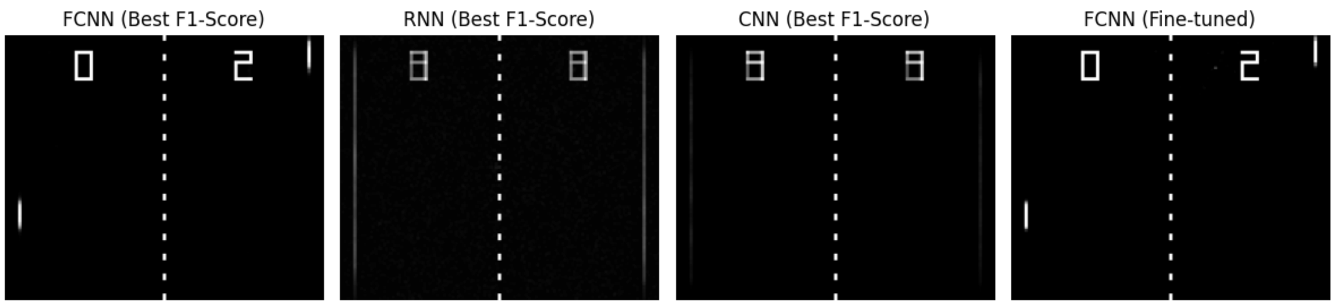

Figure 5: Visual comparison of frames generated by the best F1-score models across FCNN, RNN, and CNN architectures.

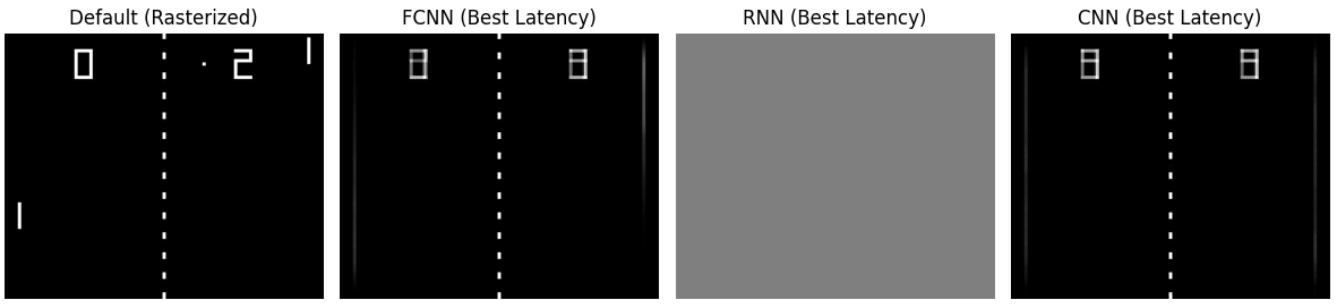

Figure 6: Visual comparison of frames generated by the best latency models across FCNN, RNN, and CNN architectures.

- **Deeper Fully-connected Networks:** Given that FCNNs showed the most promise, increasing the network depth could potentially enhance performance further, especially if combined with a larger training dataset.

4.3.2 Latency Improvements. Latency acted as a bottleneck throughout the entirety of this project. Several possible strategies that could help to reduce it include the following:

- **Avoiding Retracing:** In some cases, TensorFlow exhibited warnings that the computational graph was being retraced during ad-hoc predictions. This additional overhead, if eliminated, could help reduce per-frame prediction times.
- **Utilizing Compute Shaders:** Moving the model directly to a shader could reduce the time needed to move data between the CPU and GPU and therefore potentially lead to faster frame generation.
- **Alternative Frameworks:** Exploring alternative deep learning libraries such as PyTorch, might offer more optimized performance.

5 Related Work

The body of literature addressing real-time rendering with neural networks remains relatively limited. While image generation through deep learning is well-established, applying these techniques to real-time game rendering has received far less attention.

Guo et al. [2] attempted to predict the next frame of Atari games based on the previous frame and current user input. Similarly, Moody [4] also attempts a similar goal but for Pong using a separate neural network to predict the internal state of the game. Kim et al. [3] also explored frame prediction, relying on previous frames and user inputs to anticipate the next visual output. More recently, Decart [1] introduced a model capable of emulating Minecraft, a complex 3D game.

In contrast to these *all* of these approaches, this work feeds the complete internal game state directly to the neural network which eliminates the need for external inputs as they are essentially encoded by the game state. By providing a full snapshot of the current game state, this method theoretically supplies the model with all the details necessary to predict the current scene without error.

6 Conclusion

Overall, I believe the project achieved promising results, even though there were some notable challenges. While the models were able to generate frames that captured the general structure of the game, they struggled significantly with rendering the ball which ultimately limits the playability of the game while relying solely on model output for rendering.

Before conducting the experiments, I anticipated that the models would produce blurry but recognizable frames, while achieving faster frame rates and smaller model sizes. In reality, the models delivered relatively high precision in capturing larger elements like the paddles and score but failed to produce consistently playable outputs. Frame rates were slower than desired, and the resulting model files were quite large with a combined size of approximately 1 gigabyte.

Despite these challenges, I believe the core idea has significant potential. Extending this approach to more complex scenarios–such as 3D games with full, RGB color–could open interesting research opportunities in real-time frame generation using neural networks. Additional techniques such as custom loss functions and larger datasets should be explored in the future to further assess deep learning's potential in this particular domain.

7 Work Division

I was the only person involved in this project and was therefore resposible for all the work associated with DeepPong.

8 Learning Experience

This project provided hands-on experience in regards to the intersection between machine learning and software development. After training a variety of neural networks to achieve a specific goal, I gained a deeper understanding of model design and hyperparameter tuning. I also learned how to properly save and load trained models embedded within a real-world application.

I also experimented with designing a custom loss function to the specific goal of this project. Unfortunately, I did not have sufficient time to integrate my own loss function to improve my results as I was unfamiliar with operations available to me that preserve the computation graph. Although time constraints prevented fully realizing this idea, I still became more familiar with the process. Also, building this dataset from scratch forced me to think about what features would be useful in the context of this problem, which leaves me better equipped to handle a machine learning problem of this caliber in the future.

Finally, writing this report using LaTeX improved my technical documentation skills and familiarized me with formal academic writing practices which are essential for effective research dissemination.

References

[1] Decart AI. 2024. OASIS: An Interactive AI Video Game Model. https://www.decart.ai/articles/oasis-interactive-ai-video-game-model. Technical article.

[2] Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard L. Lewis, and Xiaoyu Wang. 2015. Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning. *Advances in Neural Information Processing Systems (NeurIPS)* 28 (2015). https://ar5iv.labs.arxiv.org/html/1507.08750 arXiv preprint.

[3] Wonjoon Kim, Jaesik Yoon, and Yoshua Bengio. 2020. GameGAN: Learning to Generate Games from Gameplay. https://research.nvidia.com/labs/toronto-ai/gameGAN/. NVIDIA Research.

[4] Josh Moody. 2020. Emulating Pong with a Neural Network. https://joshmoody.org/blog/emulating-pong-neural-network/. Blog post.

[5] Wikipedia contributors. 2024. Atari 2600 Hardware. https://en.wikipedia.org/wiki/Atari_2600_hardware. Accessed: 2024-04-27.

[6] Wikipedia contributors. 2024. Pong. https://en.wikipedia.org/wiki/Pong. Accessed: 2024-04-27.

A Additional Results

This appendix provides supplementary figures for the previously-discussed results. It includes confusion matrices and receiver operating characteristic (ROC) curves for the best-performing and lowest-latency models across the FCNN, RNN, and CNN architectures enumerated in Tables 1 and 2.

A.1 FCNN Results

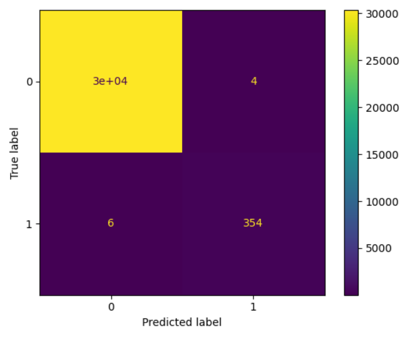

Figure 7: Confusion matrix for FCNN model with the best weighted F1-score.

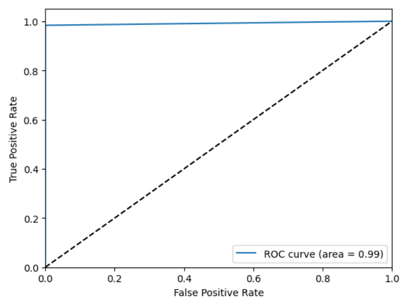

Figure 8: ROC curve for FCNN model with the best weighted F1-score.

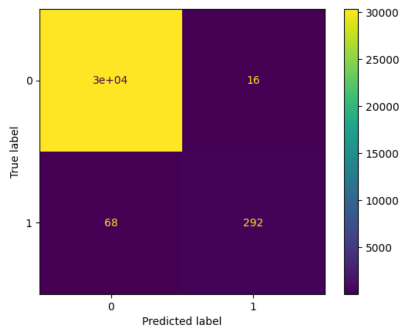

Figure 9: Confusion matrix for FCNN model with the best latency over 10 predictions.

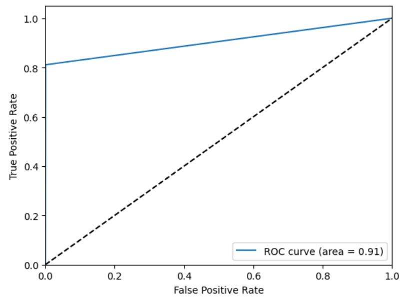

Figure 10: ROC curve for FCNN model with the best latency over 10 predictions.

A.2 RNN Results

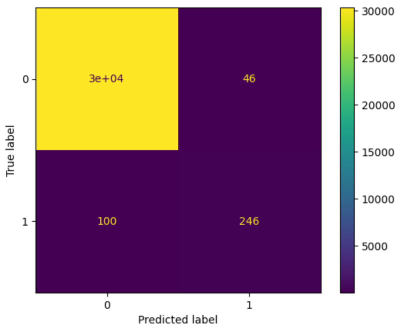

Figure 11: Confusion matrix for RNN model with the best weighted F1-score.

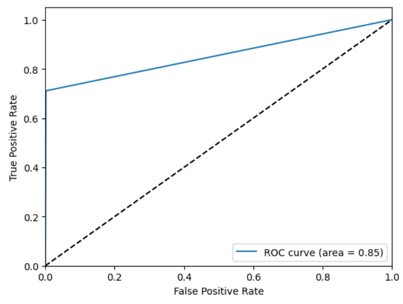

Figure 12: ROC curve for RNN model with the best weighted F1-score.

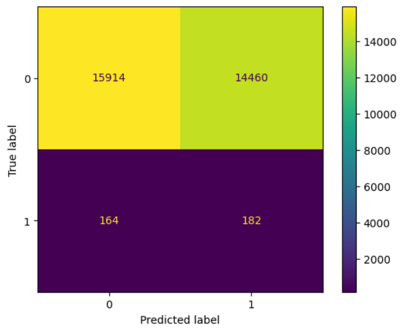

Figure 13: Confusion matrix for RNN model with the best latency over 10 predictions.

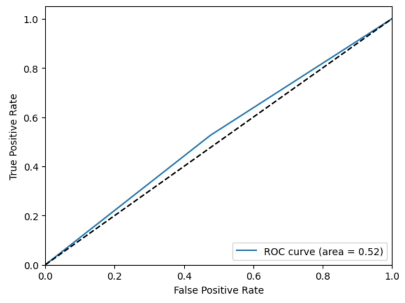

Figure 14: ROC curve for RNN model with the best latency over 10 predictions.

A.3 CNN Results

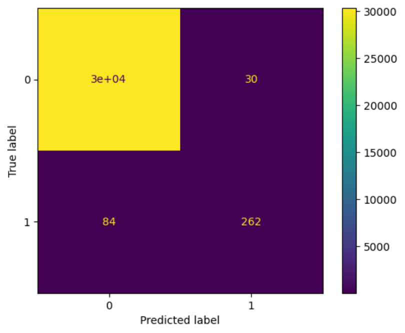

Figure 15: Confusion matrix for CNN model with the best weighted F1-score.

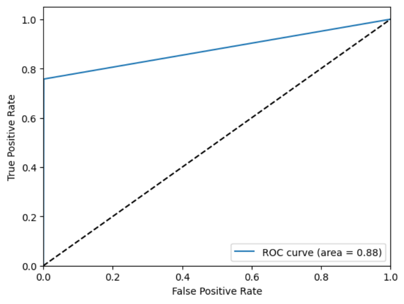

Figure 16: ROC curve for CNN model with the best weighted F1-score.

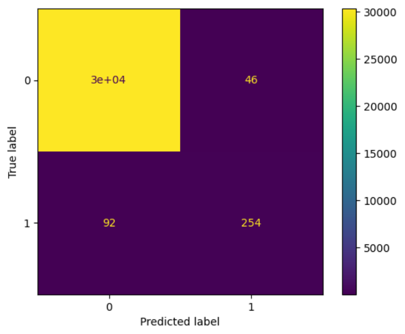

Figure 17: Confusion matrix for CNN model with the best latency over 10 predictions.

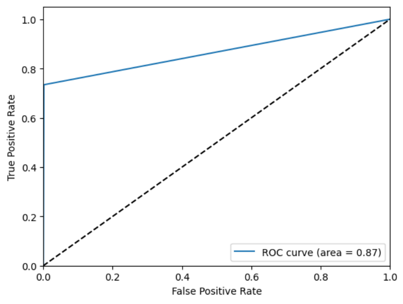

Figure 18: ROC curve for CNN model with the best latency over 10 predictions.