Final Project EECS298A

Jamil Haidar-Ahmad

December 5, 2022

1 Introduction

This course has been an very interesting for me as I have always wanted to learn signal processing and computer vision techniques. Therefore, for this final project, I decided to create my own dataset and showcase multiple techniques and models along with the challenges faced and what was learned.

I was inspired by the ConvLSTM and Actor-Critic lectures to create a dataset consisting of a timeseries of images, yet I wanted to be able to control those images. I chose to do this since if I use a public dataset, I would just be reusing someone's data and replicating their results. Instead, I wanted to create my own dataset. Therefore, I settled on creating a game from scratch. The game I chose to create is Pong. Two paddles, left and right, reflect a ball that bounces off the top and bottom walls. Whenever a paddle is unable to reach the ball at its side of the image, it loses. I decided to go all out with the implementation of the game, including non-linearities as well as adding noise.

1.1 Important note

While creating the report, I ran the scripts to get new outputs (after cleaning and documenting the Colab), but it unfortunately crashed multiple times during training. Instead, I resorted to running it on my local laptop, which is extremely slow (Colab is 15x faster). Therefore, it took more time to process all images to submit the project; instead of taking around 20 hours, it took 3 consecutive complete days.

2 Game Implementation

2.1 Ball Movement

2.1.1 Velocity update

The ball moves in a linear motion with constant speed when in the middle of the box, reflects speed when it hits one of the wall edges, and reflects non-linearly when it hits a paddle (according to where it hits the paddle and its previous direction). Here are the equations for the ball's motion:

$$(V_x, V_y) = \left\{ \begin{array}{ll} (V_x, -V_y) & \iff \text{ball bounced on wall} \\ (V_x, -V_y) & \iff \text{free motion} \\ (|V|\cos\theta + w_x, |V|\sin\theta + w_y) & \iff \text{ball hit paddle} \end{array} \right\}$$

where θ is dependent on where the ball hits the paddle. The paddle is mapped to range $[-1, 1]$, where the center of the paddle is 0. This is then mapped to $\theta = [-75, 75]$, which is the reflected angle. This means hitting the paddle dead center will eliminate any vertical speed. Finally, w_x and w_y are white noise instances normally distributed as $w_i \sim sign(V_i) \cdot N(0, 0.1)$.

2.1.2 Position Updates

The ball position dynamics are:

$$\begin{bmatrix} P_x(t+1) \\ P_y(t+1) \end{bmatrix} = \begin{bmatrix} P_x(t) \\ P_y(t) \end{bmatrix} + \begin{bmatrix} V_x(t) \\ V_y(t) \end{bmatrix}$$

2.2 Paddle AI

I coded two different AI for the left and right paddles:

- The left paddle attempts to keep the ball within its length as long as the ball is within its side of the box (half the box). If the ball is on the other side, the left player moves the paddle to the middle, making it easier to reach all sides of the board when the ball comes back to their side of the box.
- The right paddle attempts to keep the ball very close to the center of its length, there are some edge cases at the edges (where the paddle stops since it can't go outside the box). The right paddle follows the ball the entire time.

2.3 Image creation

The states are projected onto a (128,192) matrix. The ball size is (3,3) starting from its location at the bottom left. The paddle size is (4,20), where its location is the paddle's bottom left. The matrix is converted to a numpy matrix, which in turn is turned to a greyscale image through OpenCV, where the values 0 mean black and 255 mean white. It is important to note that OpenCV flips the x and y coordinates when outputing an image, making left to top and right to bottom.

2.4 Recording games

A game is played until one player scores or 1000 frames have passed. I created a game recorder object to record each game. Games are played and recorded indefinitely until 50 games are won (one player has scored). This is random, each

game's ball has a random initial velocity, random initial position, and nonlinear paddle reflections. However, I noticed that on average, I need 100-150 total games at most for 50 games to be won. I have attached a video of one of the played games title 'Normal_Game.mp4'.

2.5 Present Non-Linearity

The ball has float values for its speed, which keep changing whenever a paddle hits it (due to non-linear reflection) but the measurements (the image seen) is a 128x192 image, therefore, the ball position is converted to an integer before outputting the frame. Finally, whenever a ball is reflected by a paddle, not only is there a non-linear function affecting its reflected speed, but also additive noise.

Therefore, in this system, we have linearities and nonlinearities in the form of cos,sin,additive white noise, flooring, and rounding.

3 Ball Position Model

The only input any of our models will use is based on the board image. To start with, we will try to predict the ball's position. For this, a simple model architecture is presented.

3.1 Model 1

The first model is presented in Figure 1:

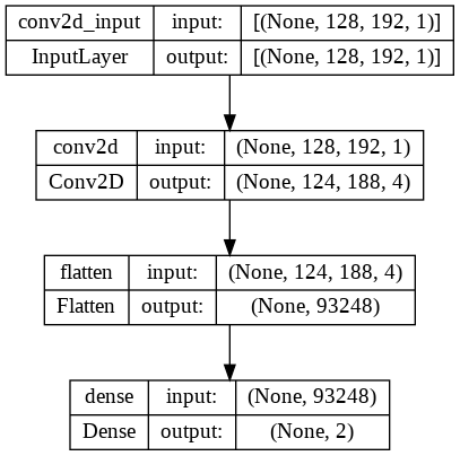


Figure 1: Ball position first model.

The model is trained on a sample of the images across all possible games in Figure 2:

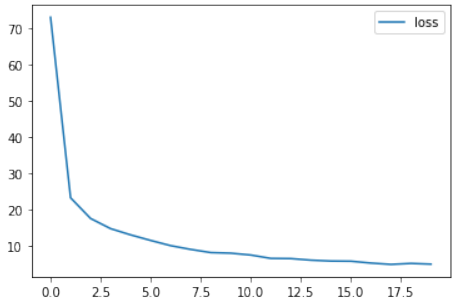


Figure 2: Ball position first model loss.

The convolution filter is of size (5,5), with a ReLU activation function and valid padding. The Dense layer has a linear activation function, with MSE being the loss function.

This model is satisfactory, but is not accurate enough, and takes too long to train. Therefore, a smaller model is proposed.

3.2 Model 2

The following model is proposed in Figure 3:

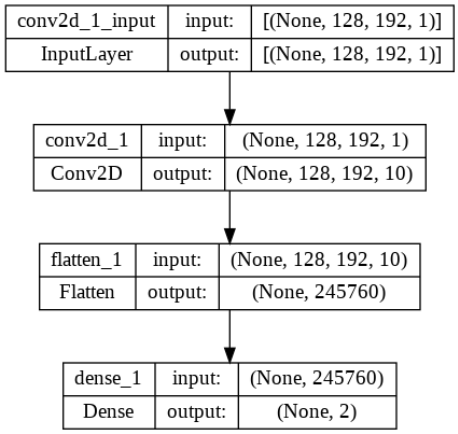


Figure 3: Ball position second model.

The convolution filter is of size (3,3), with a ReLU activation function and 'same' padding. The Dense layer has a linear activation function, with MSE being the loss function.

The reduction in filter size allows for a more precise filter, and the padding makes a difference for edge cases where the ball is close to the edge with a paddle next to it.

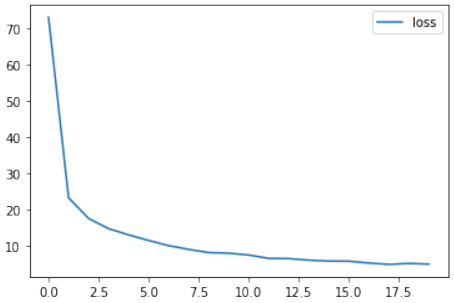


Figure 4: Ball position second model loss.

3.3 Model Testing

The model is tested on a completely new game, and the model's position is predicted from each frame. A video is generated where a green ball is drawn where the model predicts its position to be. The video is uploaded as 'predicted_ball_video.mp4'

4 Paddle Position Prediction

The paddles do not move in the x direction. Therefore, their states are encoded in the y direction only. Furthermore, the paddles move 2 pixels each frame. Therefore, the position space is reduced. The position of the left paddled can be modeled as any value in the set $P = \{2, 4, \ldots, 172\}$. This is because the position value is the bottom position of the paddle, and since it can't go outside the image, its maximum position is $192 - 20 = 172$. This sounds like a one-hot encoding problem. A model is created for each paddle. The problems are similar, so I will showcase the left paddle models explored.

4.1 Data preparation

For each frame, the left position is extracted, divided by 2, then subtracted by 1. This transforms the range in the following fashion:
$[0, 191]$(y axis) $\rightarrow [2, 4, \ldots, 172]$(possible positions for left paddle) $\rightarrow [1, 2, \ldots, 86] \rightarrow [0, 1, \ldots, 85]$.
Now that we have the one-hot encoded values, we group them with their corresponding frame.

4.2 Model 1: One-Hot/Softmax/Categorical Cross-Entropy

4.2.1 Architecture

The first model is shown in Figure 5:

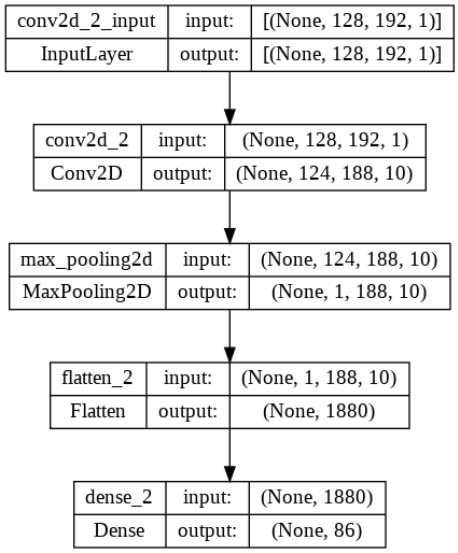


Figure 5: Left Paddle model architecture

The Conv2D layer has 10 filters, each of size (5,5), an ReLU activation function, and 'valid' padding. The Dense layer has Softmax as its activation function, and the loss function is Categorical Crossentropy.

It is important to note that the padding remains as 'valid'. This allows the model to create filters specialized for one side of the image.

4.2.2 Training

The following figure 6 shows the training loss:

4.2.3 Layer Visualization

In order to understand what is going on, I decided to dive deeper into the weights of the model. I get a sample image and forward propagate while recording each layer's output.

First, we start with convolution layer as shown in Figure 7: Next, an important addition to the network, a MaxPooling2D layer. The model was taking too long to train. If I want to flatten 10 filtered images, each of size (124,188), each to an output for one-hot encoding, we get $10 \times 124 \times 188 \times 86 = \approx 2 \times 10^7$ parameters, not counting the filter parameters and biases. Since I am training

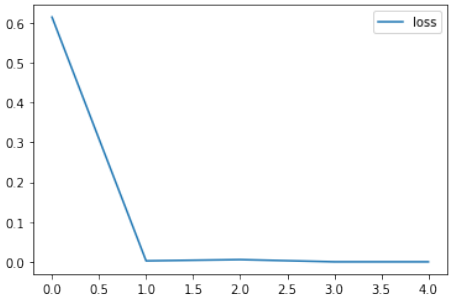


Figure 6: Paddle first model loss

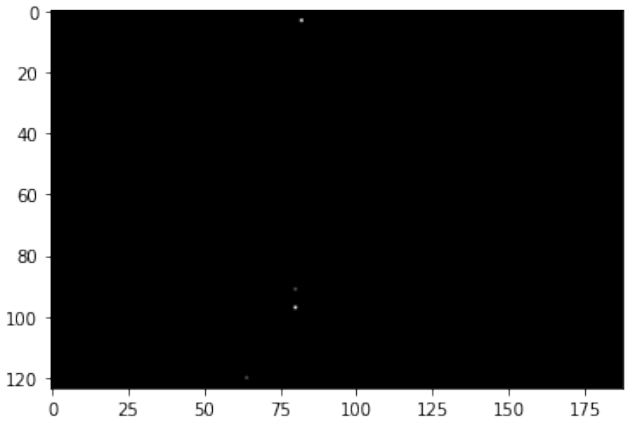


Figure 7: Convolution layer output.

on my laptop, this will not be possible in a feasible amount of time. The MaxPooling2D layer collapses the image in the x direction, giving the maximum value over the y direction. AveragePooling2D could perform better as it keeps more information of what is happening across the image instead of taking the maximum on each y coordinate.

The MaxPooling2D layer's output is shown in Figure 8

Finally, we get the one-hot encoding output using softmax as shown in Figure 9:

This corresponds to the correct value.

4.3 Discussion

Since we forced the model to learn only one convolution filter, it has to create a filter that matches really well with the input image to extract a paddle shape. We can sort of see this in Figure 10 below, where the image attempts to find a stripe corresponding to the paddle.

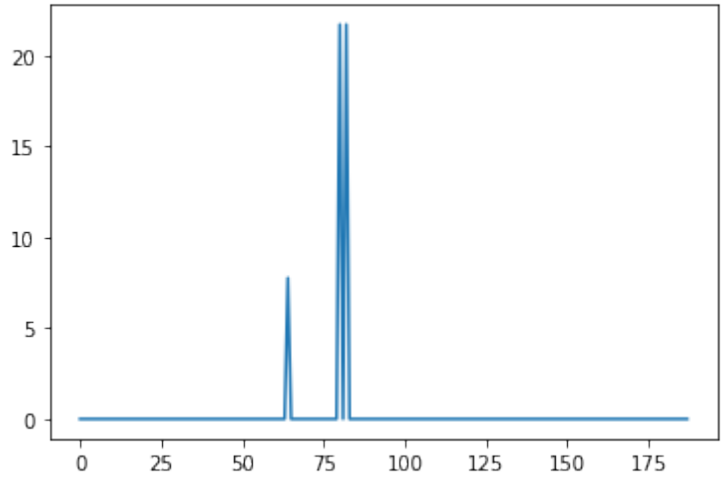


Figure 8: MaxPooling layer output

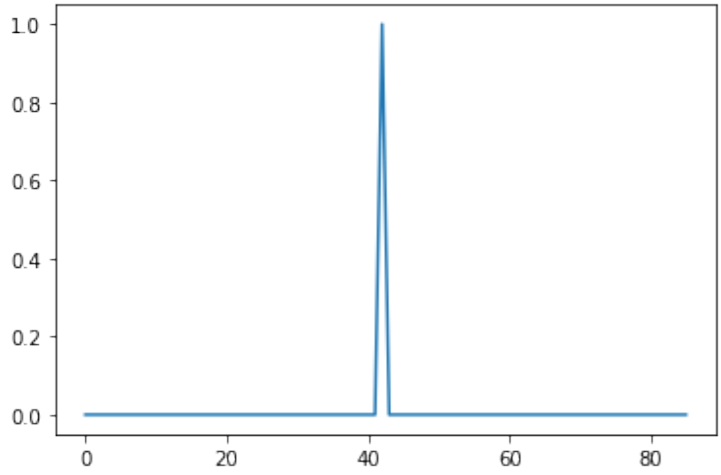


Figure 9: Softmax Output

Since the right and left paddles are the same shape, what differentiates them (if doing convolution only), is their proximity to the edges. This is why I made the padding 'valid', so that no additional zeros are added, allowing for a filter that works for the left paddle, but not right one.

I believe that the bottom and right pixels are almost the same weight, (which probably is removing that part of the image), making the filter asymmetric and creating a filter for the left paddle only instead of the right paddle. This could also be analyzed as trying to filter out the ball (3x3) shape.

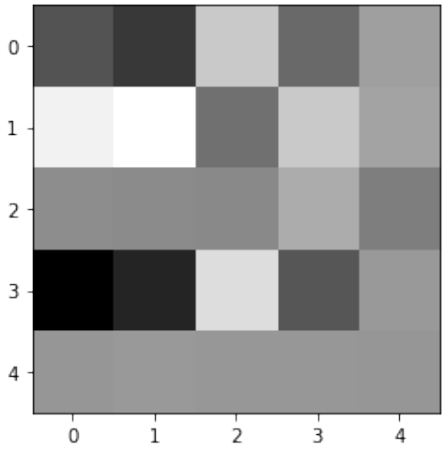


Figure 10: First model filter

The relu activation function lets the neural network either accept th convolution sum or reject it (by sliding the relu using the bias).

I added in a MaxPooling2D layer to collapse the image onto the dimension we're looking for, giving the maximum value across it. For example, if the filter was a (4,2) where each row is [1,1,0,0], then we'll get a maximum value across the entire paddle length. This will show up as a rect after max pooling. Then, all we need is to get the start (rising edge), and that is the left paddle coordinate.

However, since during the game, the paddle y positions can overlap, this would lead to windows where the left paddle coordinate is lost in the middle. This is why a more unique filter should be applied to detect only the left paddle, where the left paddle coordinate shows as a spike.

4.4 Model 2: Linear-MSE

4.4.1 Architecture

We compare the previous model's performance with a model with linear output. The architecture is shown in Figure 11:
Here, I take a different approach. I force the model to learn only 1 filter of size (5,5), with padding being valid. Then, I flatten the output and pass it through a single unit with linear activation function. The constraint of having only one filter forces the model to learn a filter that is specialized for the left paddle. The filter must be able to extract features corresponding to the left paddle's corner position, or else the next neuron will not be able to differentiate between the ball and the paddle if they are on the same x level for example. Furthermore, making the Dense neuron have a linear activation function puts most of the focus on the convolutional filter to get it right.

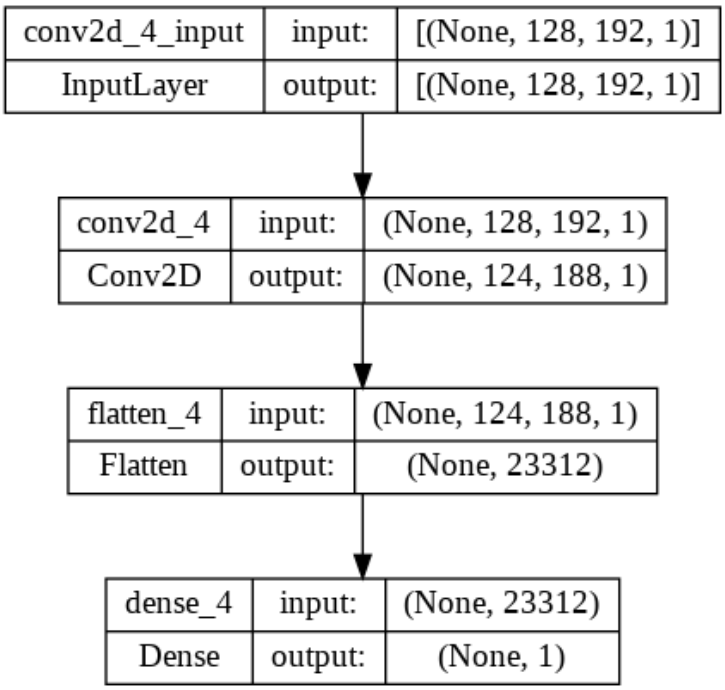


Figure 11: Left Paddle second model

4.4.2 Training

The training loss is shown below in Figure 12:

4.4.3 Visualizing the outputs

The following images in Figure 13 show the outputs right out of the convolutional layer. We can see that the right paddle is completely filtered out. This is due to the convolution having a relu activation function and that the padding is 'valid'. The filter is specializing to keep the left paddle's position, which means that after the filter is multiplied by the section of the image, the sum will be calculated, and the bias which shifts the Relu function, decides if this sum is enough to pass, or is considered 0. Therefore, the better the image section fits the filter, it passes, else, it could end up being erased completely.

4.4.4 Visualizing the filter

Since we narrowed the convolution down to only one filter, let's take a look at the filter. It looks very similar to the first discovered filter. The right and bottom values are low and as we go to the upper left, a different output is shown. This filter could have been manually created to get perfect results.

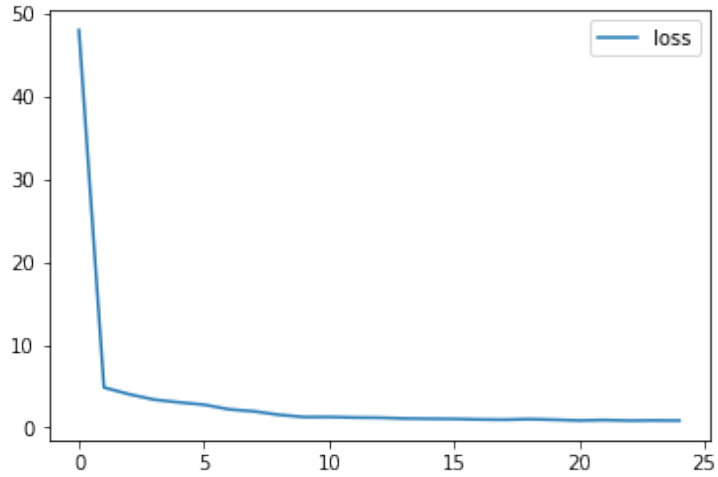


Figure 12: Paddle second model loss

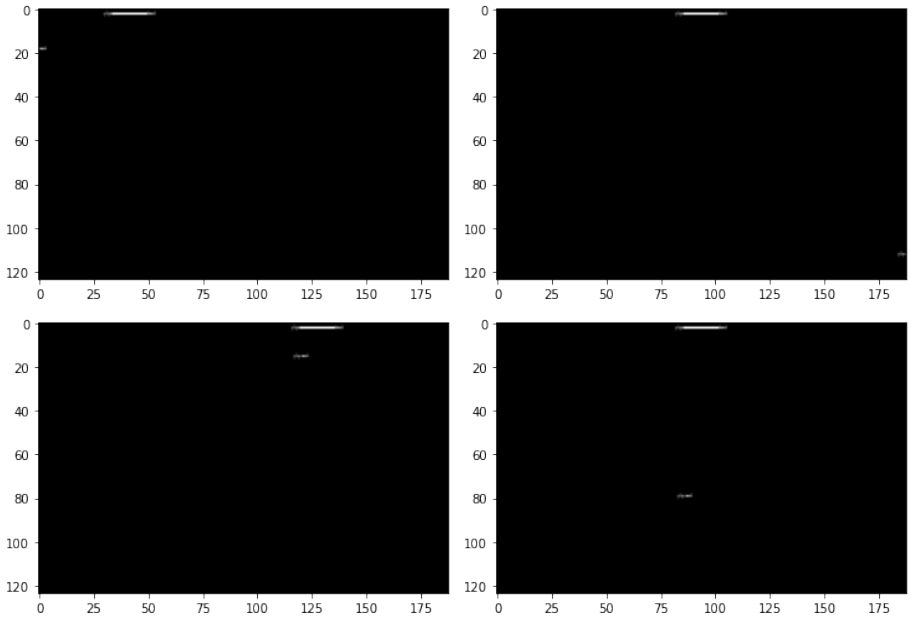


Figure 13: Convolution of different instances

4.5 Right Paddle Model

The exact procedure is done for the right paddle. Figure 15 shows the output of the convolution layer, showing how the left paddle is now removed.

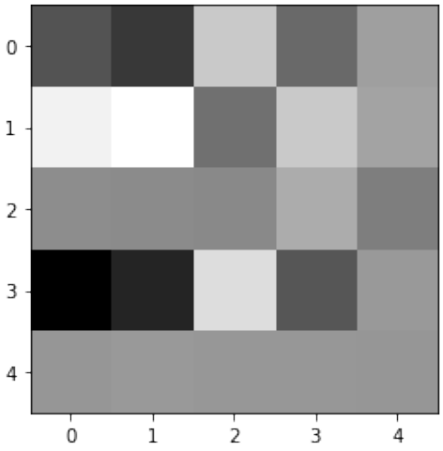


Figure 14: Second model filter

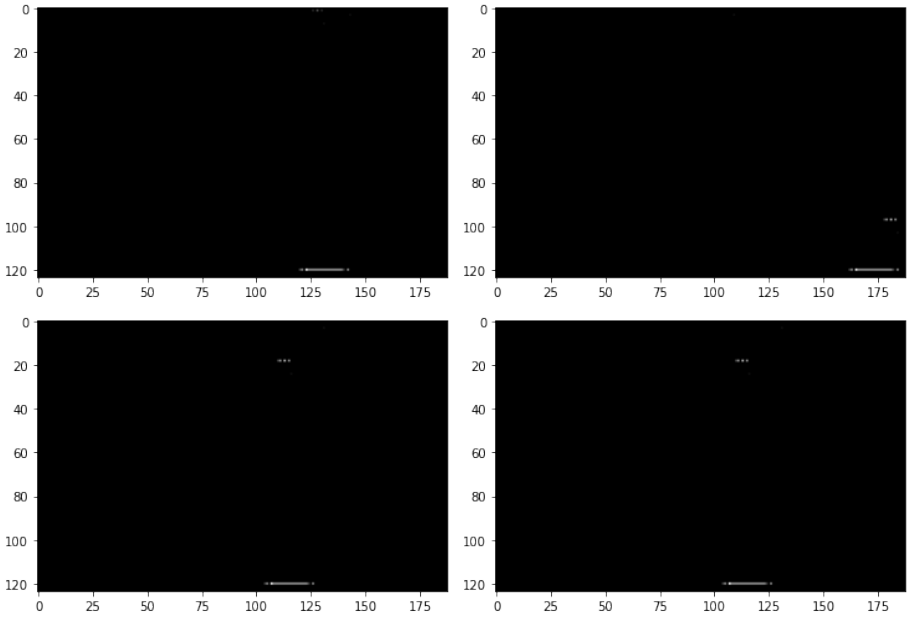


Figure 15: Convolution of different game instances

5 Ball Velocity estimation

5.1 Free movement Velocity

This is the simplest model. For each frame, we predict its velocity by giving it the previous 3 frames. Figure 16 shows the model architecture. The output

activation function is linear. The model is only trained for 2 epochs. The output is shown in Figure 17

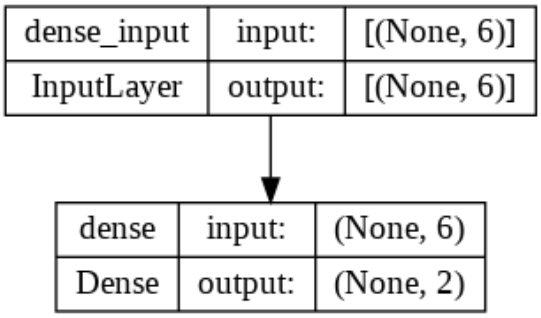


Figure 16: Free Velocity Model

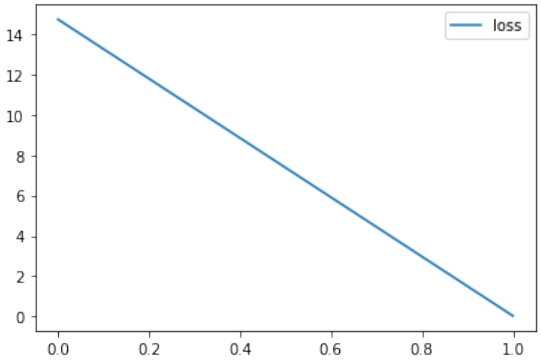


Figure 17: Free Velocity Training

5.2 Bounce Velocity

Next, we want to predict the velocity for when the ball hits one of the walls. The approach is exactly the same as the free velocity model. Training is done as shown in Figure 18.

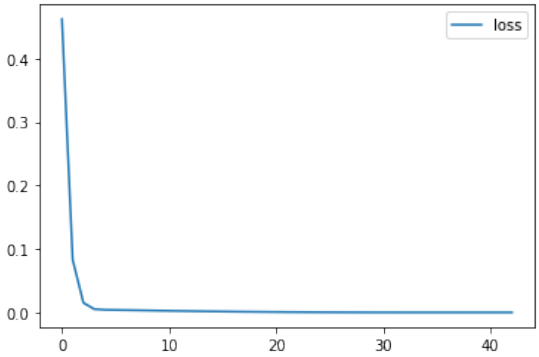


Figure 18: Free Velocity Training

Figure 19 shows an example of a predicted output and the expected output:

```
[101] print(ball_velocity_model.predict(np.array([x[123]]))*2)
      print(y[123]*2)

      1/1 [==============================] - 0s 26ms/step
      [[1.4199747 1.519763 ]]
      [1.42 1.52]
```

Figure 19: Free Velocity Training

5.3 Collision Estimation

The most difficult part is modeling the ball velocity after colliding with a paddle. This includes multiple non-linearities and has to take into account noise.

5.3.1 Architecture

The model is 3 layered. The first two layers are 10 unit sigmoids, and the last layer is a linear output layer. The inputs are the ball's position and the paddle's position when collision occurs (since the function depends solely on this). Figure 20 show the model's architecture.

5.3.2 Training

The model trains well as its loss decreases to acceptable levels. It is allowed 50 epochs as the training set is not too large, but it converges in the early stages of training. Figure 21 depicts the training loss.

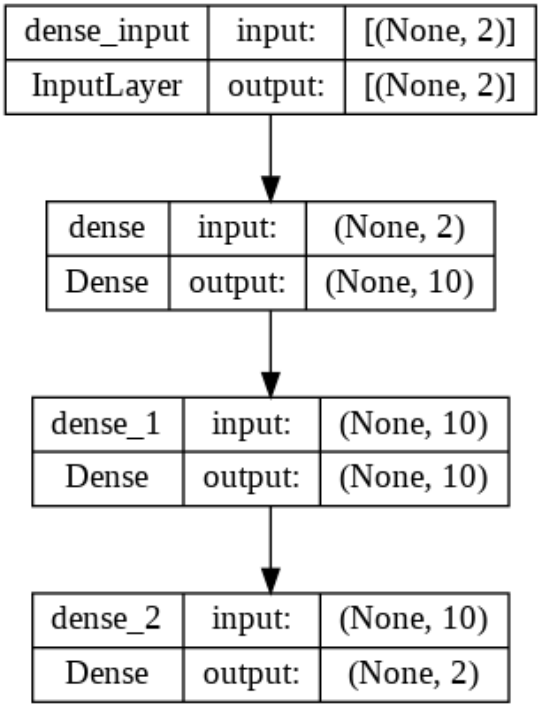


Figure 20: Collision Velocity Model

5.3.3 Outputs

The time indices for each game are collected when a ball collides with the left paddle. The input to the model is created and the output is tested on this instance. Figure 22 shows the example.
The model is tested on the a separate test dataset and the outputs for the x and y velocities are plotted in Figure 23.

6 State Classification

To complete the prediction of the ball's states, we need to know when the ball is moving freely, when it is bouncing, and when it is colliding with a paddle. We use a CNN for this task with sparse categorical crossentropy. The states are encoded as 0 if the ball is bounced, 1 for colliding with a paddle, and 2 for free movement.

Since the data is extremely skewed, some preprocessing must be done. The free moving indices are about 60k while the bouncing indices are just about 600, and the collision indices are only about 800. The two other indices are very sparse. Therefore the solution was to artificially increase their size by padding.

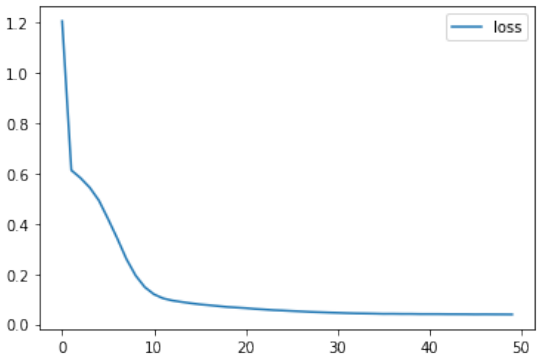
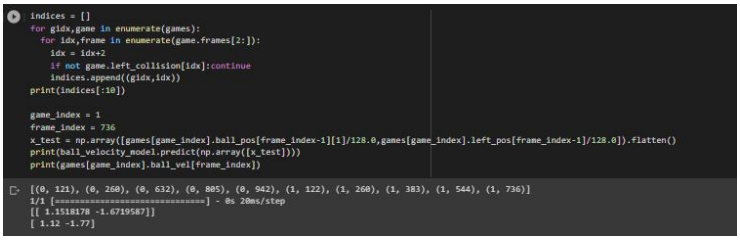


Figure 21: Collision Velocity Model loss



```
indices = []
for gidx,game in enumerate(games):
  for idx,frame in enumerate(game.frames[2:]):
    idx = idx+2
    if not game.left_collision[idx]:continue
    indices.append((gidx,idx))
print(indices[:10])

game_index = 1
frame_index = 736
x_test = np.array([games[game_index].ball_pos[frame_index-1][1]/128.0,games[game_index].left_pos[frame_index-1]/128.0]).flatten()
print(ball_velocity_model.predict(np.array([x_test])))
print(games[game_index].ball_vel[frame_index])
```

```
[(0, 121), (0, 260), (0, 632), (0, 805), (0, 942), (1, 122), (1, 260), (1, 383), (1, 544), (1, 736)]
1/1 [==============================] - 0s 20ms/step
[[ 1.1518178 -1.6719587]]
[ 1.12 -1.77]
```

Figure 22: Single example instance for collision velocity prediction.

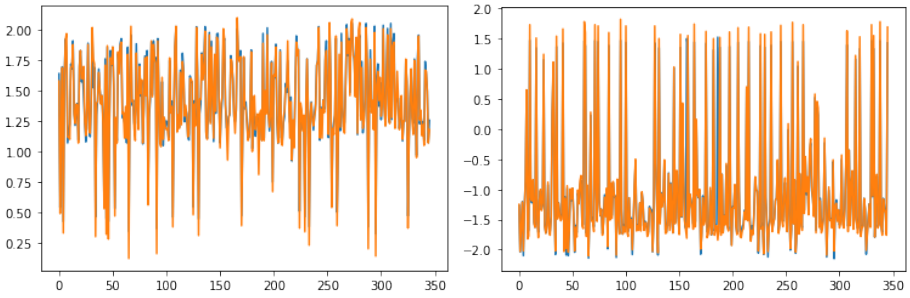


Figure 23: Comparison of collision velocity model on test set.

A total of 4000 images for each case is chosen randomly. The inputs to the model are the ball position and paddle position at each frame.

The model is trained with **shuffle=True** to shuffle around the categories (and apply persistence of excitation) and the loss is shown in Figure 24. Finally,

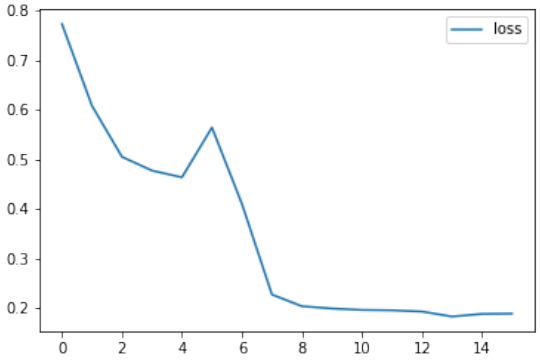


Figure 24: Categorization loss

the model is tested on the entire dataset of all games and we get the confusion matrix shown in Figure 25:

7 Conclusion

An approach similar to tackling a real life problem is presented. Given a video of a game, we attempt to extract state information from the images, and then utilize those information to predict future states as well as classify modes of operation. The most challenging part in this project was the extremely slow training time and multiple crashes faced (causing the entire kernel to shutdown and losing all variables), making it necessary that I get the model right from the first few tries. I learned to save weights at every epoch and load them when necessary. I also learned how CNN filters are created and how settings such as padding affect the filters as well as how the activation function affects its behavior and outputs. I learned that even simple problems could have complicated solutions, and some complicated problems could have simple solutions. Finally, I learned that it is important to divide problems into sub-problems, as neural networks could find it difficult to learn huge tasks, even if they seem simple to us. For example, I had attempted to find the positions of the ball and both paddles at the same time, but it took too long to converge. Furthermore, when the model did converge, its accuracy was not "good enough". I needed a precision strictly less than 2 pixel for the position of the ball in order to be able to predict its velocity. A precision of 3 pixels completely throws the ball off trajectory. For sensitive and precise problems, it is better to create a network for each sub-problem. Finally, I have attached a video of all models working at

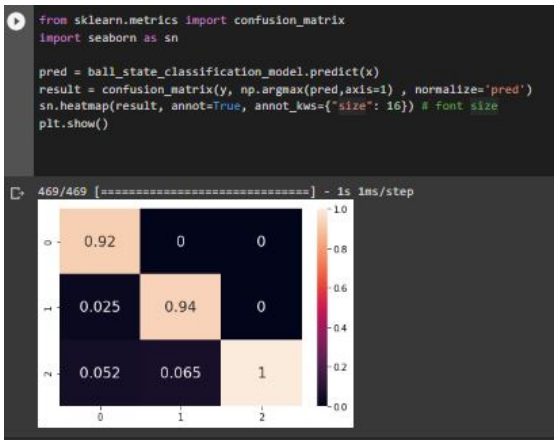


Figure 25: Confusion matrix for sparse category classification.

the same time to predict the board's states and estimate the future position of the ball.