Task 1

Model Architecture and Design:

CNN models are developed to predict the coordinates (x, y, and z) for this task. The first model was designed to predict only the x coordinate, and the second model was created to predict all three coordinates (x, y, z).

The first model has a simple structure with two convolutional layers and a fully connected layer that predicts the x coordinate. The second model uses a multi-branch CNN architecture, where each branch is used for predicting one of the coordinates (x, y, and z). Both models were trained for 20 epochs, and validation data was used to observe how the model performance during training.

Approach and Methodology:

For both the models Mean Squared Error (MSE) was used as the loss function because the task was to predict continuous values (coordinates). In the first model, to optimize the prediction, MSE was used of the x coordinate, whereas for the second model, MSE was applied to each output (x, y, z). The model combined the individual losses into a total loss to optimize for all three outputs at the same time.

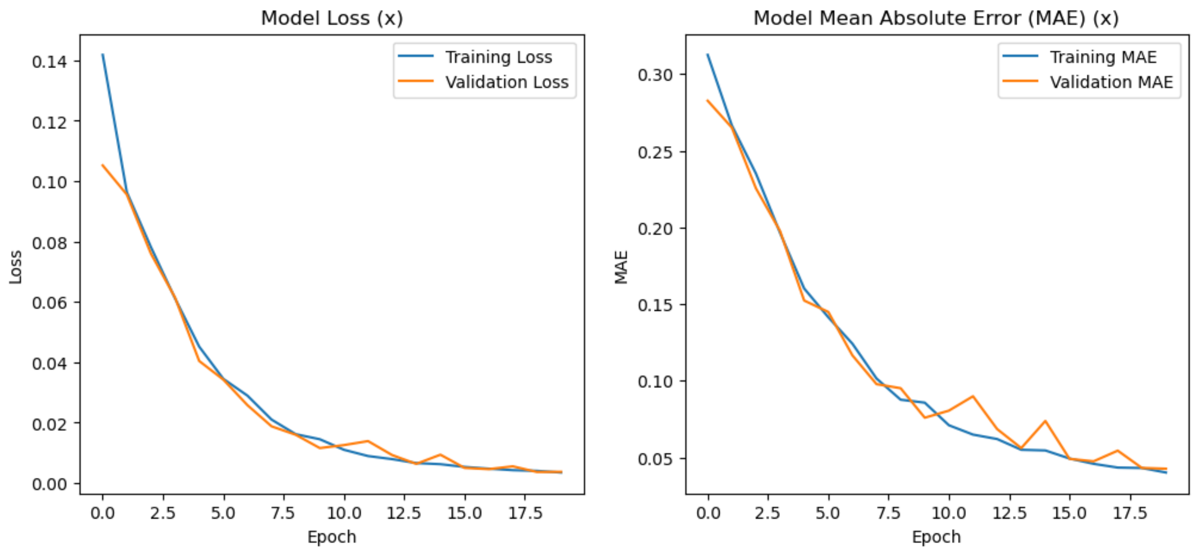


Figure *1*: First Model

In the first model (Figure 1), the loss dropped from 0.14 to nearly 0.00 over 20 epochs, while the MAE reduced from 0.30 to approximately 0.05.

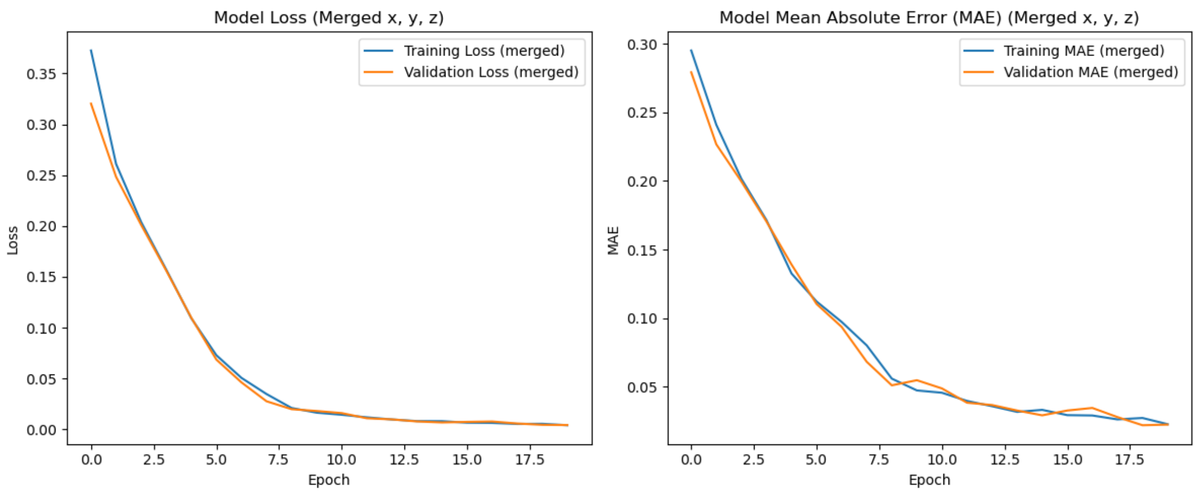


Figure 2: Second Model

The second model (Figure 2) showed a similar trend, with the merged loss dropping from 0.35 to near 0.00, and the MAE for the combined coordinates decreasing from 0.30 to around 0.05.

The graphs show that the models converged well, since the training and validation curves follow each other very close without overfitting. This indicates that the models generalized good on unseen data.

Training Duration and Performance:

The training of both models were done for 20 epochs. The first model ran for 4.06 seconds, whereas the second model ran for 8.08 seconds.

The performance on the test set was assessed using a tolerance-based accuracy metric. The first model achieved 70.41% accuracy, and the second model's accuracy for the x, y, and z coordinates were 84.02%, 86.39%, and 97.63%. The overall accuracy for the multi-output model was 89.35%.

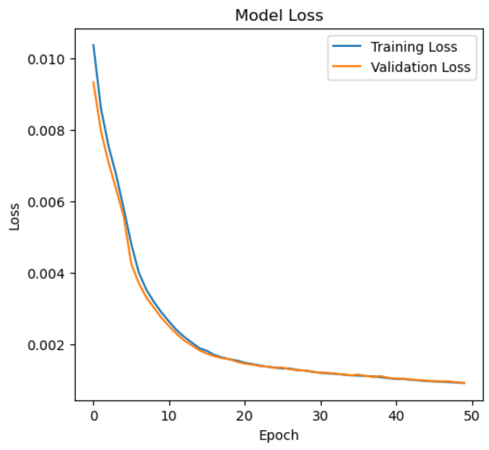
Task 2

Here a convolutional autoencoder is implemented to reconstruct images from the MiniPong dataset using the allpix.csv dataset. The images were normalized to the range [0, 1] and reshaped into (15, 15, 1) for processing.

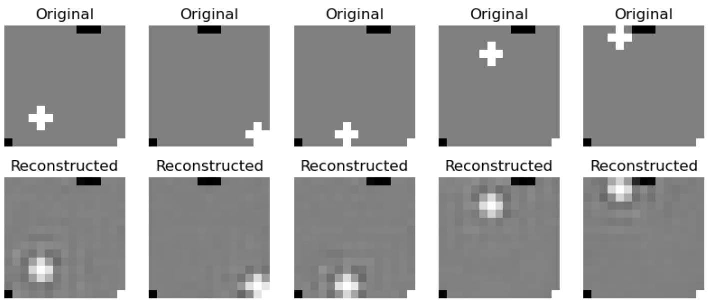
The autoencoder architecture consists of an encoder that compresses the input image into a smaller latent representation using convolutional and pooling layers, and a decoder that restores the original image using upsampling and transposed convolutions. To ensure the output matched the input size a Cropping2D layer was used.

The model was trained for 50 epochs with a batch size of 32, using Mean Squared Error (MSE) as the loss function and the Adam optimizer. Both the training and validation losses consistently decreased, showing good learning progress. The final test loss was 0.0009, showing that the autoencoder was able to reconstruct the images with minimal error.

Below is a loss curve showing the model's performance during training:



Here are five examples comparing original images (top) with their reconstructed versions (bottom). While the model performs well on some images, such as the first, it struggles with finer details in others, like the fourth image:



In theory, a convolutional autoencoder's hidden layer could be as small as 1×1 by compressing all image information into a single value. However in practice this size loses too much detail for complex images. The ideal latent space size maintains a balance between compression and reconstruction quality, thus a slightly bigger hidden layer often exceeds expectations particularly with pictures containing special characteristics.

Task 3

To create an RL agent for MiniPong, a tabular Q-learning approach was implemented with a TD-learning update rule. The environment's observed state (level 1) is based on dz, the relative position of the ball to the paddle. This state is discretized into 21 bins, allowing us to create a finite Q-table that represents the state-action pairs.

The agent follows an epsilon-greedy policy for action selection, beginning with $\epsilon = 1$ for exploration and decaying gradually to a minimum of $\epsilon = 0.1$. The agent was trained over 500 episodes and at each step it updates its Q-values based on rewards received, learning to maximize its cumulative reward by effectively positioning the paddle.

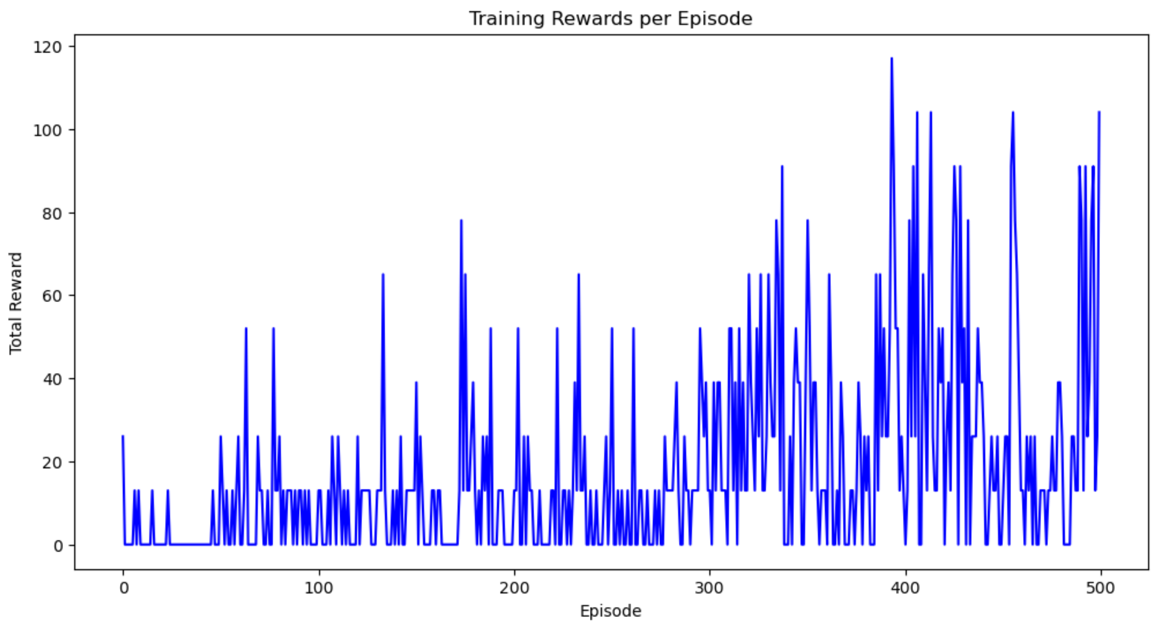
The agent was trained for 500 episodes, with each episode capped at 200 steps to be efficient. The Training Reward plot below shows the total rewards per episode, showing how the agent's performance improved over time. This training process took approximately 5.66 seconds.

After training, the agent was evaluated over 50 test episodes with $\epsilon = 0$ to test its performance based on the learned Q-values. The results were as follows:

Test Average Reward: 161.2

Test Standard Deviation: 146.4333295394187

The following plot shows the cumulative reward per episode during the training phase. The upward trend indicates that the agent learned to keep the ball in play more effectively as training progressed.



In level 2 of MiniPong, the observed state includes the ball's y-coordinate along with dz. This information would help the agent learn more effectively. The y-coordinate allows the agent to understand when the ball will reach the paddle which provides better spatial context. This additional detail can improve decision-making by allowing the agent to make more accurate paddle movements.

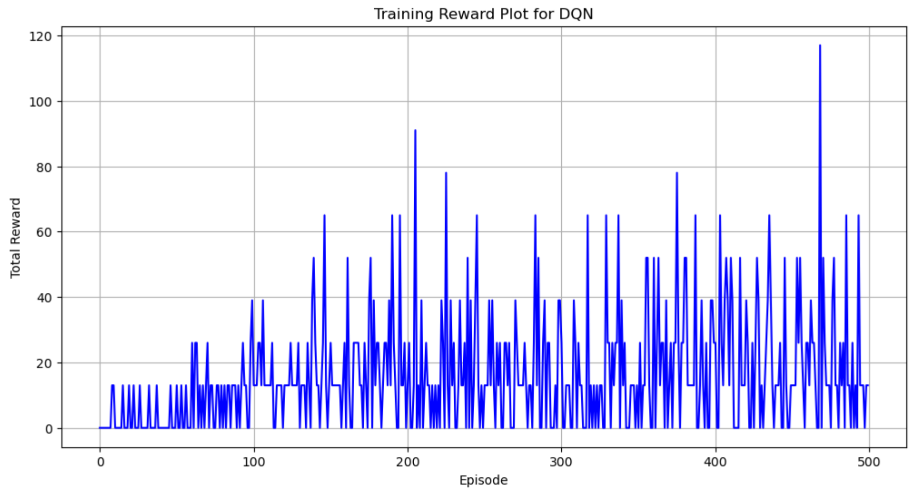
Task 4

To tackle the MiniPong environment at Level 3, a Deep Q-Network (DQN) agent was implemented. The DQN architecture has two hidden layers with 32 neurons each using ReLU activations to ensure non-linearity in decision-making. The network takes a 3-dimensional input (y, dx, and dz) which represents the state and outputs Q-values for each of the three actions (move left, move right, do nothing).

The agent was trained over 500 episodes with an epsilon-greedy action selection strategy, where ϵ started at 1 (for maximum exploration) and decayed gradually to a minimum of 0.1 to encourage exploitation. The training loop involved storing experiences in a replay memory and updating Q-values using a batch size of 64.

To decrease the computational load, each episode was capped at 200 steps. The learning rate for the Adam optimizer was set at 0.001, and gamma (the discount factor) was set to 0.99 to prioritize long-term rewards.

The Training Reward Plot shows the total rewards achieved by the agent per episode. Although there were improvements over time, the rewards did not stabilize at a high level, with the highest peaks around 100. This suggests the agent was learning but did not achieve consistently high performance.



After training, the agent was evaluated over 50 test episodes using a greedy policy (ϵ = 0). The performance results were as follows:

Test Average Reward: 30.42

Test Standard Deviation: 28.32

These results show that while the agent learned to obtain some positive reward, its performance remained variable and inconsistent due to limitations in learning stability and the complexity of the task.

Despite tuning the learning rate and epsilon decay rate, the DQN did not consistently achieve high rewards > 300. The lack of stability could be due to the limited training episodes or architectural simplicity. Further optimization, such as increasing the number of neurons or adding more episodes, might improve performance.