Assignment Cover Sheet

School of Computer, Data, and Mathematical Sciences

Student Name	Pramod K C
Student Number	22085342
Subject	INFO7001: Advanced Machine Learning
Title of Assignment	Project
Due Date	27 Oct 2024
Date Submitted	27 Oct 2024

DECLARATION

I hold a copy of this assignment that I can produce if the original is lost or damaged.

I hereby certify that no part of this assignment/product has been copied from any other student's work or from any other source except where due acknowledgement is made in the assignment. No part of this assignment/product has been written/produced for me by another person except where such collaboration has been authorised by the subject lecturer/tutor concerned.

Signature: ......Pramod......................................

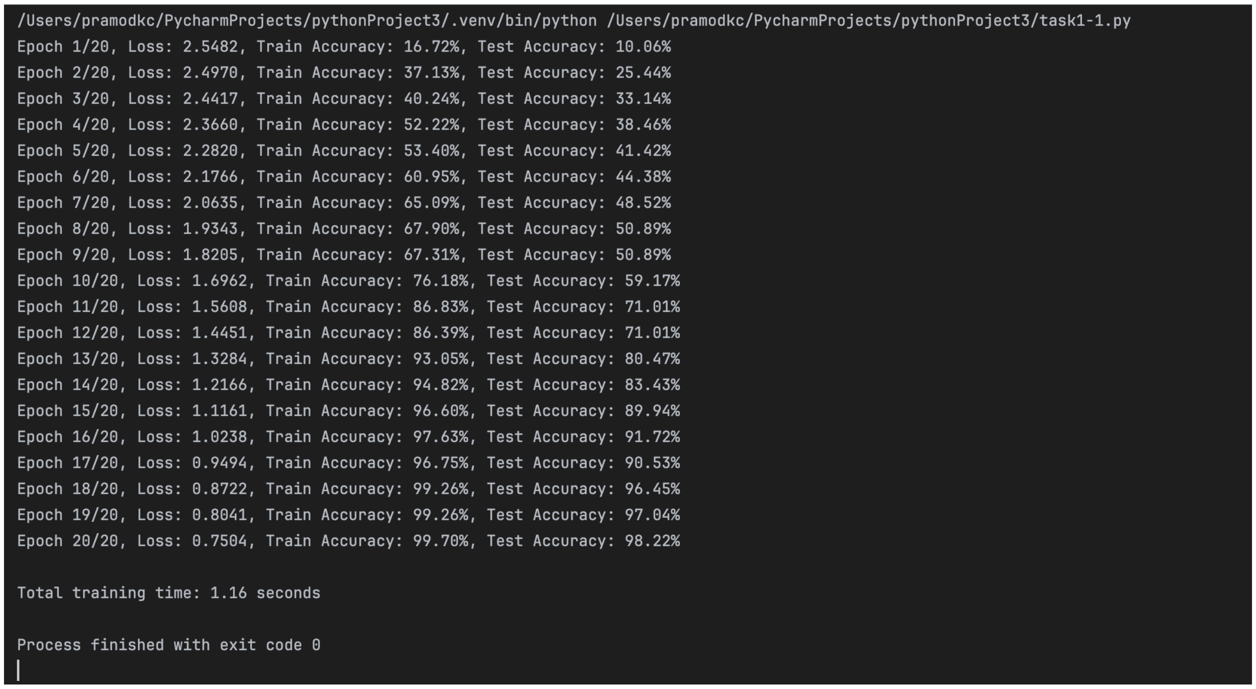
(Note: An examiner or lecturer/tutor has the right not to mark this assignment if the above declaration has not been signed)

Task 1: Train a CNN to predict object positions

The model CNN_singlepred is build to predict only the x-coordinate from 15x15 grayscale images. The output can be classified into one of 13 classes, with 0 to 12 being the probable outcomes. Its architecture comprises one convolutional layer of 32 filters, followed by the max-pooling layer, which will shrink the spatial dimensions of feature maps. Next comes a fully connected layer, the result of which flattens the feature maps into a 128-dimensional vector that feeds the final fully connected layer making up the x-coordinate prediction.

CrossEntropyLoss is used for the loss function, which in general is a good choice when dealing with classification problems and also works well in this problem, where we predict categorical values for the possible 13 x-coordinates. The optimization of the model is done using the Adam optimizer with a learning rate set to 0.0001, which is efficient enough in updating the weights of the model during training.

The model was build for 20 epochs. The total training time is 1.16 seconds. After each epochs each loss, training and test accuracy is calculated and we can see that we get 98.22% test accuracy.

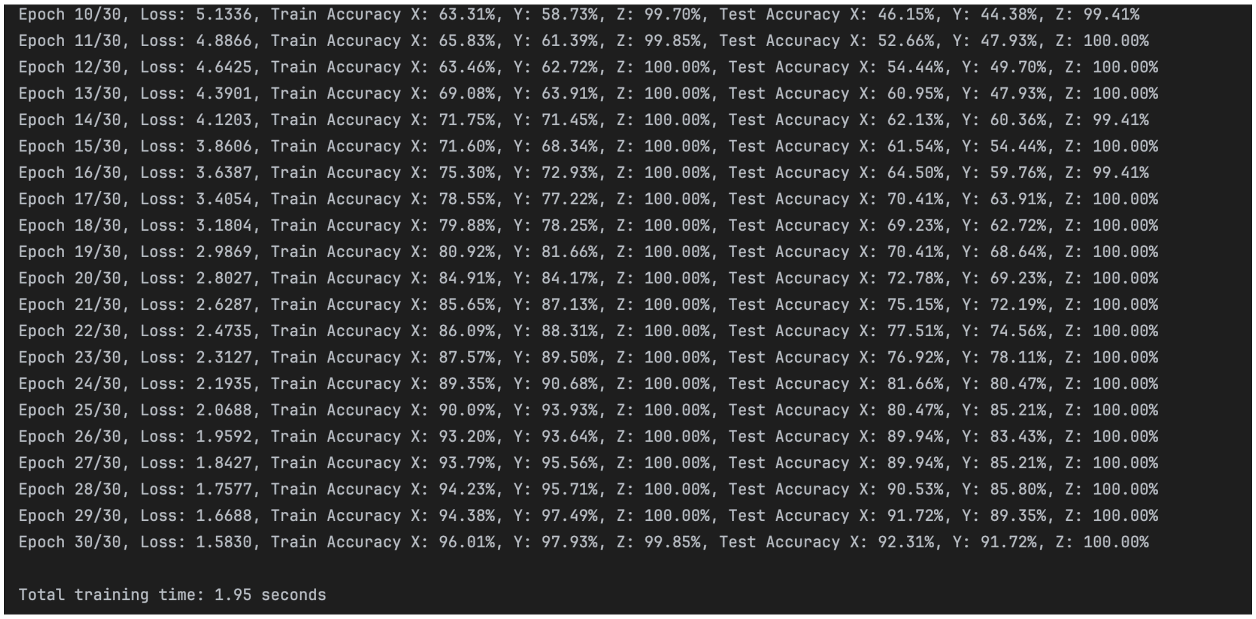


```
/Users/pramodkc/PycharmProjects/pythonProject3/.venv/bin/python /Users/pramodkc/PycharmProjects/pythonProject3/task1-1.py
Epoch 1/20, Loss: 2.5482, Train Accuracy: 16.72%, Test Accuracy: 10.06%
Epoch 2/20, Loss: 2.4970, Train Accuracy: 37.13%, Test Accuracy: 25.44%
Epoch 3/20, Loss: 2.4417, Train Accuracy: 40.24%, Test Accuracy: 33.14%
Epoch 4/20, Loss: 2.3660, Train Accuracy: 52.22%, Test Accuracy: 38.46%
Epoch 5/20, Loss: 2.2820, Train Accuracy: 53.40%, Test Accuracy: 41.42%
Epoch 6/20, Loss: 2.1766, Train Accuracy: 60.95%, Test Accuracy: 44.38%
Epoch 7/20, Loss: 2.0635, Train Accuracy: 65.09%, Test Accuracy: 48.52%
Epoch 8/20, Loss: 1.9343, Train Accuracy: 67.90%, Test Accuracy: 50.89%
Epoch 9/20, Loss: 1.8205, Train Accuracy: 67.31%, Test Accuracy: 50.89%
Epoch 10/20, Loss: 1.6962, Train Accuracy: 76.18%, Test Accuracy: 59.17%
Epoch 11/20, Loss: 1.5608, Train Accuracy: 86.83%, Test Accuracy: 71.01%
Epoch 12/20, Loss: 1.4451, Train Accuracy: 86.39%, Test Accuracy: 71.01%
Epoch 13/20, Loss: 1.3284, Train Accuracy: 93.05%, Test Accuracy: 80.47%
Epoch 14/20, Loss: 1.2166, Train Accuracy: 94.82%, Test Accuracy: 83.43%
Epoch 15/20, Loss: 1.1161, Train Accuracy: 96.60%, Test Accuracy: 89.94%
Epoch 16/20, Loss: 1.0238, Train Accuracy: 97.63%, Test Accuracy: 91.72%
Epoch 17/20, Loss: 0.9494, Train Accuracy: 96.75%, Test Accuracy: 90.53%
Epoch 18/20, Loss: 0.8722, Train Accuracy: 99.26%, Test Accuracy: 96.45%
Epoch 19/20, Loss: 0.8041, Train Accuracy: 99.26%, Test Accuracy: 97.04%
Epoch 20/20, Loss: 0.7504, Train Accuracy: 99.70%, Test Accuracy: 98.22%

Total training time: 1.16 seconds

Process finished with exit code 0
```

The model CNN_multipredict will predict x, y, and z coordinates on 15x15 grayscale images. This model will predict three different things independently: the x-coordinate as one of 13 classes ranging from 0 to 12, the y-coordinate as one of 13 classes ranging from 0 to 12, and the z-coordinate as one of 5 classes ranging from 0 to 4, showing the paddle position. It is similar in architecture to the first model described herein, starting with a convolutional layer followed by max pooling. A pooling layer is followed by flattening the feature maps into a shared fully connected layer, which outputs a 128-dimensional vector. Thereafter, from this shared layer, the model branches out to three different fully connected layers-also known as "output heads"-each of which is responsible for predicting one of the x, y, or z coordinates.

Since all these three-outputs, x, y, and z are classification tasks, CrossEntropyLoss has been used for each. The total loss will be the sum of losses for x, y, and z individually. This makes the training process cumbersome as compared to the single output CNN_singlepred model. As in Model 1, the optimizer is the same, which is the Adam optimizer with a learning rate of 0.0001.

This model is trained for 30 epochs. Since this model is predicting three outputs at once, training compared to Model 1 takes longer. The training time in seconds for the above code is 1.95 seconds. Finally, track the performance of this model by calculating the accuracy of x, y, and z predictions separately on both the training and test sets. This multioutput structure gives a more complete assessment of the model performance because it tracks accuracy across all three coordinate predictions. We can observe that after each epoch, loss is decreasing, and accuracy is increasing. At the end, we get test accuracy for x, y and z are 92.31%, 91.72%, 100.00%, respectively.



```
Epoch 10/30, Loss: 5.1336, Train Accuracy X: 63.31%, Y: 58.73%, Z: 99.70%, Test Accuracy X: 46.15%, Y: 44.38%, Z: 99.41%
Epoch 11/30, Loss: 4.8866, Train Accuracy X: 65.83%, Y: 61.39%, Z: 99.85%, Test Accuracy X: 52.66%, Y: 47.93%, Z: 100.00%
Epoch 12/30, Loss: 4.6425, Train Accuracy X: 63.46%, Y: 62.72%, Z: 100.00%, Test Accuracy X: 54.44%, Y: 49.70%, Z: 100.00%
Epoch 13/30, Loss: 4.3901, Train Accuracy X: 69.08%, Y: 63.91%, Z: 100.00%, Test Accuracy X: 60.95%, Y: 47.93%, Z: 100.00%
Epoch 14/30, Loss: 4.1203, Train Accuracy X: 71.75%, Y: 71.45%, Z: 100.00%, Test Accuracy X: 62.13%, Y: 60.36%, Z: 99.41%
Epoch 15/30, Loss: 3.8606, Train Accuracy X: 71.60%, Y: 68.34%, Z: 100.00%, Test Accuracy X: 61.54%, Y: 54.44%, Z: 100.00%
Epoch 16/30, Loss: 3.6387, Train Accuracy X: 75.30%, Y: 72.93%, Z: 100.00%, Test Accuracy X: 64.50%, Y: 59.76%, Z: 99.41%
Epoch 17/30, Loss: 3.4054, Train Accuracy X: 78.55%, Y: 77.22%, Z: 100.00%, Test Accuracy X: 70.41%, Y: 63.91%, Z: 100.00%
Epoch 18/30, Loss: 3.1804, Train Accuracy X: 79.88%, Y: 78.25%, Z: 100.00%, Test Accuracy X: 69.23%, Y: 62.72%, Z: 100.00%
Epoch 19/30, Loss: 2.9869, Train Accuracy X: 80.92%, Y: 81.66%, Z: 100.00%, Test Accuracy X: 70.41%, Y: 68.64%, Z: 100.00%
Epoch 20/30, Loss: 2.8027, Train Accuracy X: 84.91%, Y: 84.17%, Z: 100.00%, Test Accuracy X: 72.78%, Y: 69.23%, Z: 100.00%
Epoch 21/30, Loss: 2.6287, Train Accuracy X: 85.65%, Y: 87.13%, Z: 100.00%, Test Accuracy X: 75.15%, Y: 72.19%, Z: 100.00%
Epoch 22/30, Loss: 2.4735, Train Accuracy X: 86.09%, Y: 88.31%, Z: 100.00%, Test Accuracy X: 77.51%, Y: 74.56%, Z: 100.00%
Epoch 23/30, Loss: 2.3127, Train Accuracy X: 87.57%, Y: 89.50%, Z: 100.00%, Test Accuracy X: 76.92%, Y: 78.11%, Z: 100.00%
Epoch 24/30, Loss: 2.1935, Train Accuracy X: 89.35%, Y: 90.68%, Z: 100.00%, Test Accuracy X: 81.66%, Y: 80.47%, Z: 100.00%
Epoch 25/30, Loss: 2.0688, Train Accuracy X: 90.09%, Y: 93.93%, Z: 100.00%, Test Accuracy X: 80.47%, Y: 85.21%, Z: 100.00%
Epoch 26/30, Loss: 1.9592, Train Accuracy X: 93.20%, Y: 93.64%, Z: 100.00%, Test Accuracy X: 89.94%, Y: 83.43%, Z: 100.00%
Epoch 27/30, Loss: 1.8427, Train Accuracy X: 93.79%, Y: 95.56%, Z: 100.00%, Test Accuracy X: 89.94%, Y: 85.21%, Z: 100.00%
Epoch 28/30, Loss: 1.7577, Train Accuracy X: 94.23%, Y: 95.71%, Z: 100.00%, Test Accuracy X: 90.53%, Y: 85.80%, Z: 100.00%
Epoch 29/30, Loss: 1.6688, Train Accuracy X: 94.38%, Y: 97.49%, Z: 100.00%, Test Accuracy X: 91.72%, Y: 89.35%, Z: 100.00%
Epoch 30/30, Loss: 1.5830, Train Accuracy X: 96.01%, Y: 97.93%, Z: 99.85%, Test Accuracy X: 92.31%, Y: 91.72%, Z: 100.00%

Total training time: 1.95 seconds
```

The difference between Model 2 from the first one is that it has been converted from a single output-that which predicted the x-coordinate-to a multi-output model, now predicting the coordinates at x, y, and z all at once. The amendment went on to increase the complexity of the model greatly. While the first model computes the loss only for the x-coordinate, Model 2 combines losses from all three outputs-the x, y, and z values. With that, Model 2 becomes more computationally heavy and takes longer to train compared to Model 1, because the multioutput structure requires one step computation and optimization of multiple predictions.

Task 2: Train a convolutional autoencoder

Instead of predicting positions, create a convolutional autoencoder that compresses the MiniPong screenshots to a small number of bytes (the encoder), and then transforms them back to the original form (in the decoder part).

Our dataset consists of 15x15 grayscale images, read from a CSV file, trainingpix.csv. The reshaped images take this shape: shape=(batch_size, 1, 15, 15), which is the shape required as input to the convolutional layers in PyTorch. Then a DataLoader loads the data in batches for training. Since we will work with an autoencoder that tries to reconstruct an image that is as close as possible to the input image, images here play a dual role: both as input and target in our dataset.

It is a variant of the convolutional autoencoder model, which normally consists of two parts: the encoder and decoder. This model compresses the input image into its small representation through the encoder. It starts with a convolutional layer using 32 filters of kernel size 3x3, followed by another layer downsampling the feature map to 16 filters using a stride of 2. The final layer in the encoder downsamples the feature map even further down to 8 filters, reducing the spatial size to 8x4x4. It further up-samples the feature maps using the transposed convolution layer until it is returned to its normal size, and inverting the compressed representation back to the original 15x15 image.

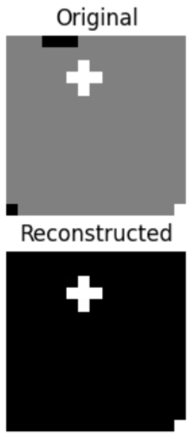
Mean Squared Error MSE is used for the loss function. This, in essence, calculates the difference between the original image and the reconstructed image. Therefore, MSE would be a good choice because image reconstruction tasks require pixel-by-pixel accuracy. This model utilizes an Adam optimizer, where the learning rate was set to 0.001, meaning it adapts the learning rate during training. It generally converges faster than conventional optimizers, such as SGD.

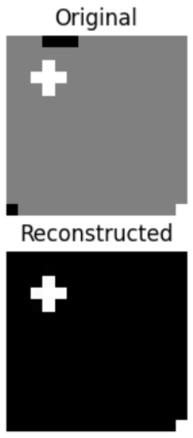
It trains this autoencoder for 100 epochs; at each epoch, a minimum loss occurs depending on how the original image looks like from its reconstruction. At each epoch, it prints out the loss to monitor the progress of the model in learning to reconstruct images. After training, the original and the reconstructed images are visualized, side by side, as to assess how well the autoencoder is doing.

This results in an encoded image of architecture size 8x4x4, which is considerably smaller than the original image of 15x15. Theoretically, the smallest size of the hidden layer representation which will allow perfect reconstruction is equal to that of the original image; here, this amounts to 225 pixels. However, using a convolutional autoencoder, one can compress the image into a much smaller representation while still yielding recognizable reconstructions. The minimum size that would allow for perfect reconstruction should be at least 225 features to hold all the pixel information without loss.

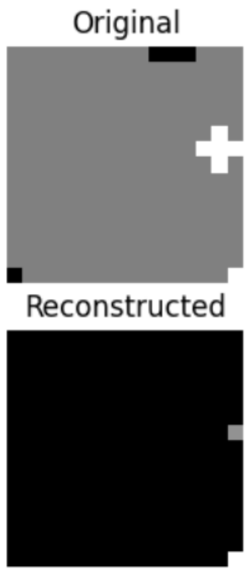
In a 15×15 image, there are a total of 225 pixels; thus, theoretically, it would require at least 225 units in the hidden layer to be able to perfectly reconstruct the image, because each unit in the hidden layer will have to represent one pixel's information from the original image. However, in practice, convolutional autoencoders can often learn a much more compact representation summarizing either the most salient features or patterns in an image to allow recognizable reconstructions with fewer units than the total number of pixels. In any case, for a perfect reconstruction without loss, you would want to keep all the pixel information, which means the hidden layer would have to store at least 225 features.

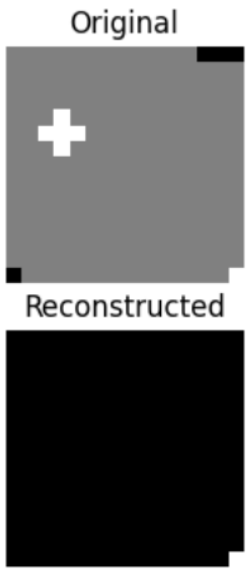
Good Image:





Bad Image:





Task 3: Create an RL agent for MiniPong
The code in mini pong.py provides an environment similar in style to other gymnasium environments, with different levels and sizes. Below is a description of the environment dynamics:

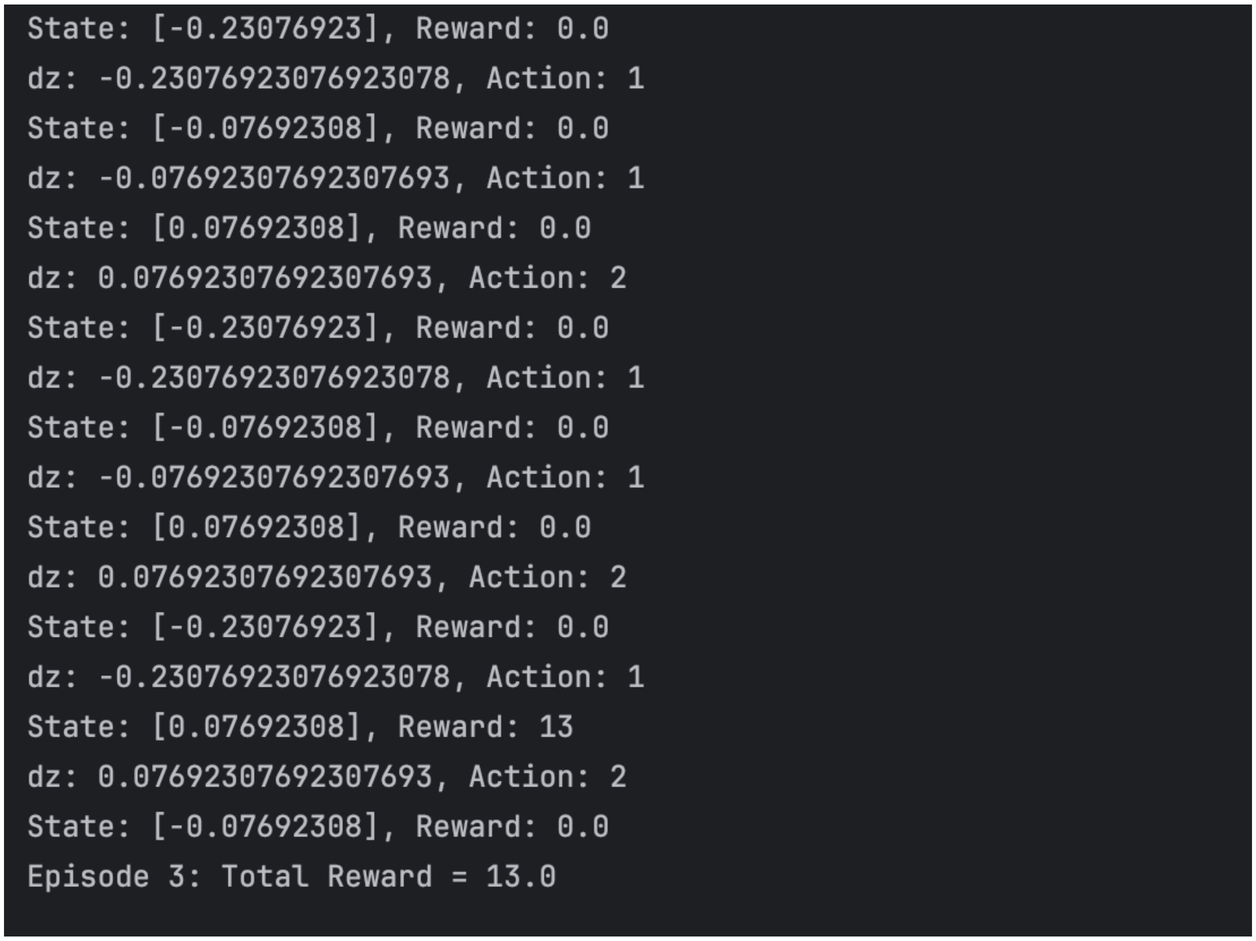
- **The ball+ marker moves diagonally at each step. When it hits the paddle or a wall (top,left, or right), it reflects.**
- **The agent controls the paddle−, moving it one 3-pixel slot per step. The agent has three actions available: do nothing, move left, or move right. The paddle cannot move outside the boundaries.**
- **The agent receives a positive reward when the + marker reflects off the paddle. In this case, the + may also move randomly by 1 or 2 pixels left or right.**
- **An episode ends when the + reaches the bottom row without reflecting off the paddle. In the level 1 version of the environment, the observed state (the information available to the agent after each step) consists of a single value, dz: the relative position of the + marker to the centre of the paddle−. This value is negative if the + marker is on one side, and positive if on**
the other.

For this task, you should use level=1 and size=5. You can choose to normalise the state information to values between -1 and 1 by setting normalise=True, or use the original (integer)

values in the range−13...13 by setting normalise=False.
env = M i n i P o n g E n v (level =1 , size =5 , normalise = True)

In this project, I implemented two key components to play and learn MiniPong:

Part 1: Rule-Based Agent, No Reinforcement Learning
I was required to provide a naive rule-based agent play MiniPong depending on the value of dz. Therefore, the agent only required the simple heuristic that if dz < 0, then move left; if dz > 0, then move right; and if dz = 0, do nothing. That let the agent follow basic game mechanics without learning through trial and error.

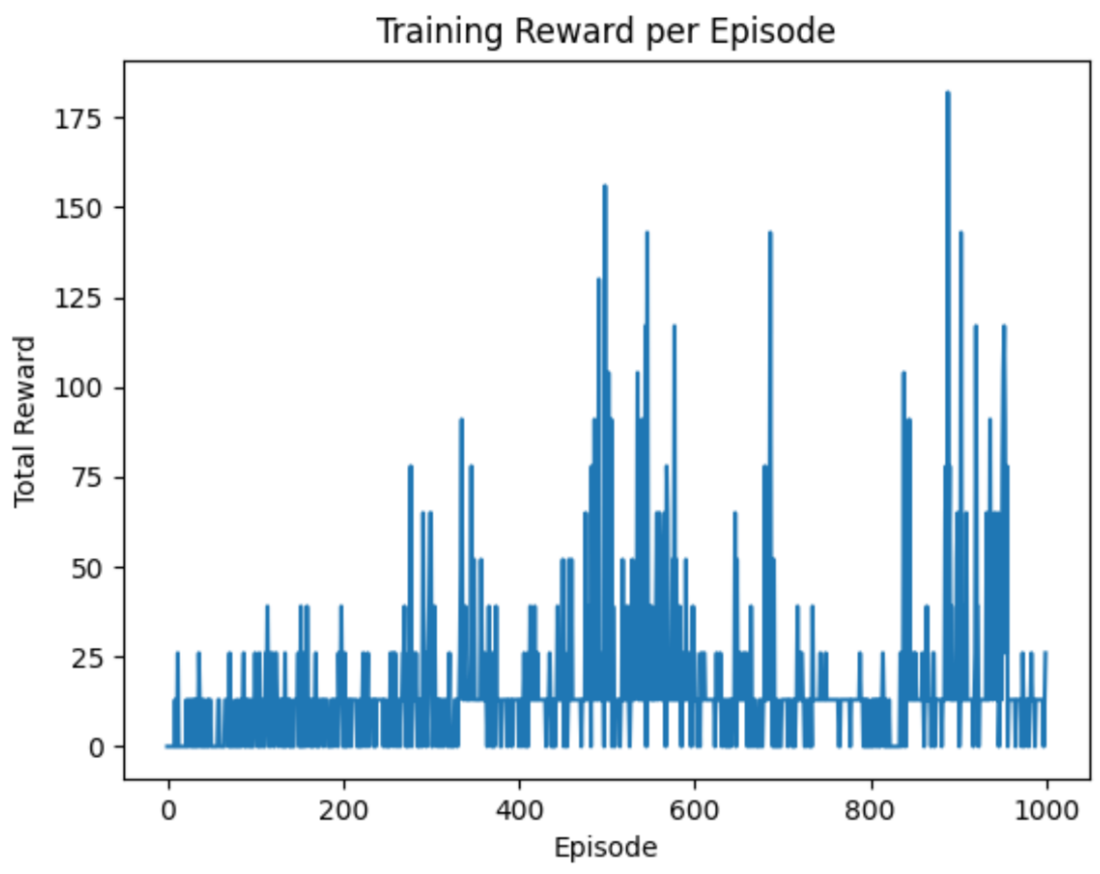


```
State: [-0.23076923], Reward: 0.0
dz: -0.23076923076923078, Action: 1
State: [-0.07692308], Reward: 0.0
dz: -0.07692307692307693, Action: 1
State: [0.07692308], Reward: 0.0
dz: 0.07692307692307693, Action: 2
State: [-0.23076923], Reward: 0.0
dz: -0.23076923076923078, Action: 1
State: [-0.07692308], Reward: 0.0
dz: -0.07692307692307693, Action: 1
State: [0.07692308], Reward: 0.0
dz: 0.07692307692307693, Action: 2
State: [-0.23076923], Reward: 0.0
dz: -0.23076923076923078, Action: 1
State: [0.07692308], Reward: 13
dz: 0.07692307692307693, Action: 2
State: [-0.07692308], Reward: 0.0
Episode 3: Total Reward = 13.0
```

Part 2: Q-Learning Agent (Reinforcement Learning)
The second part was to develop a Q-learning agent that learns to play MiniPong in a tabular Q-learning approach with an epsilon-greedy action selection. First, the Q-learning agent discretizes the continuous state space and then learns a Q-value table, mapping states to actions. The agent first explores the environment with ϵ = 1-full exploration-and decays thereafter to 0.1 as the agent learns. In each step of a training episode, an agent updates its Q-table based on the TD update rule.

Training Process and Results:
Training Episodes: The agent trained for 1000 episodes. In each episode, the environment was first reset to its default setting, then at every step, the agent interacted with the environment with the best action from the epsilon-greedy policy being chosen. Moving on with the reward accumulated in each episode - it was tracked and stored.

Training Reward Plot: We plotted the cumulated rewards per episode, which allowed us to visualize how the agent improves over time. The Training Reward plot typically shows a positive trend while starting to learn and may be noisy due to balancing exploration and exploitation.

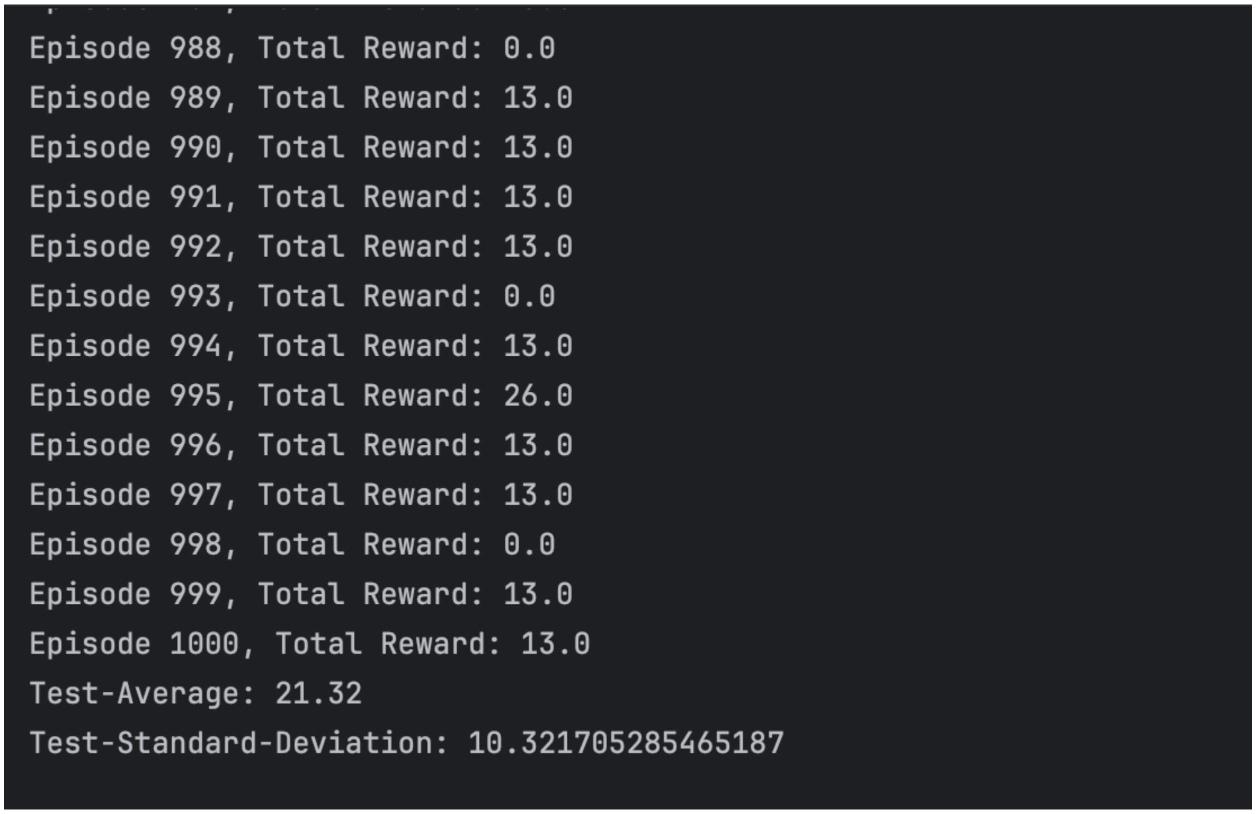


The training graph plots the total reward the agent received during each episode throughout training. The horizontal axis represents the number of episodes, ranging from 0 to 1000, while the vertical axis represents the total reward earned in each episode. Early on in training, within the first 200 episodes, the rewards are low since the agent is yet to explore the environment and learn how to act in such a way as to accumulate rewards. Moving further along in training, rewards keep fluctuating upwards, with instances as high as 80 to 100, especially after episode 500. This behavior is very normal for reinforcement learning, due to exploration-exploitation trade-offs. Sometimes it explores new actions to receive lesser rewards, and other times it uses its learned input to receive higher rewards. The upward trend within the rewards shows an agent that's learning and is putting the learned strategy into practice; the variability suggests it's still exploring, but the higher rewards point to progress in the learning process.

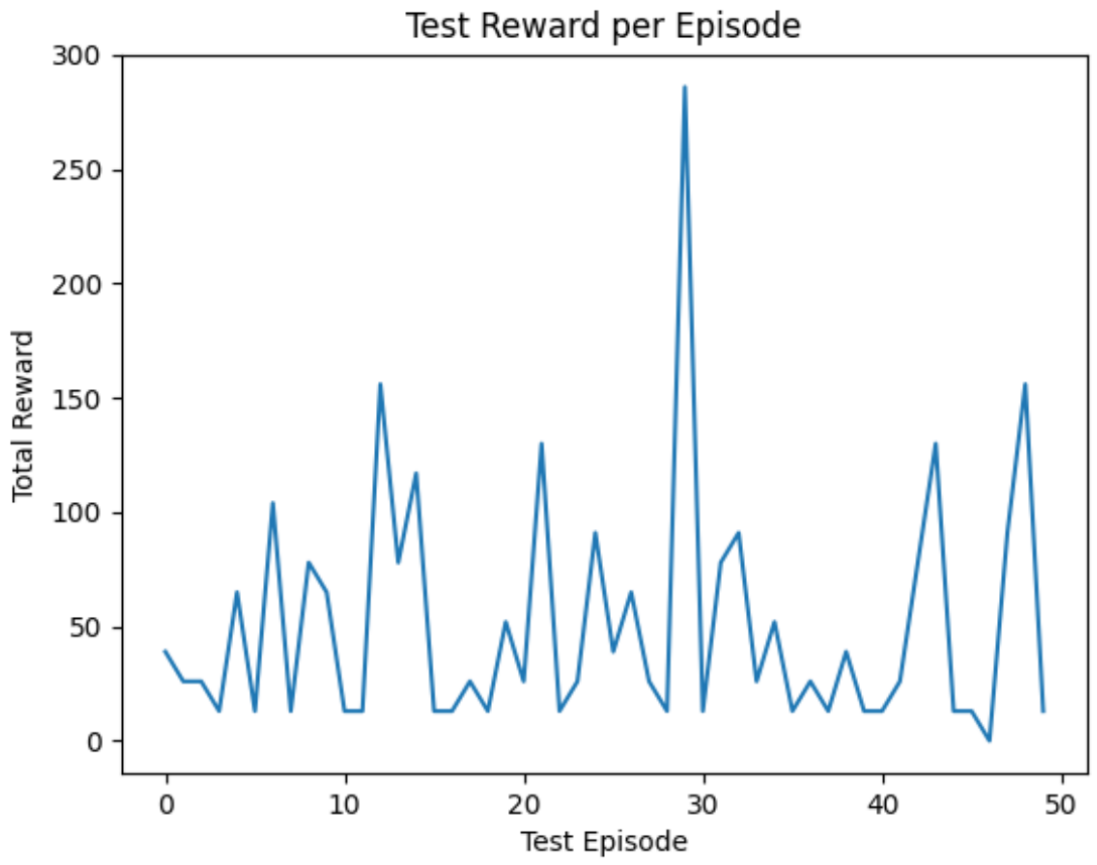
Test Performance:
Then, I ran 50 test episodes with ϵ = 0 (pure exploitation, no exploration) to test the agent's performance using its learned policy. At the beginning of every episode, the environment was reset, and cumulative rewards were collected.

Test Average:
The average sum of rewards over 50 test episodes was 21.32.

Test-Standard-Deviation:
The standard deviation of the sum of rewards over the 50 test episodes was 10.322.



```
Episode 988, Total Reward: 0.0
Episode 989, Total Reward: 13.0
Episode 990, Total Reward: 13.0
Episode 991, Total Reward: 13.0
Episode 992, Total Reward: 13.0
Episode 993, Total Reward: 0.0
Episode 994, Total Reward: 13.0
Episode 995, Total Reward: 26.0
Episode 996, Total Reward: 13.0
Episode 997, Total Reward: 13.0
Episode 998, Total Reward: 0.0
Episode 999, Total Reward: 13.0
Episode 1000, Total Reward: 13.0
Test-Average: 21.32
Test-Standard-Deviation: 10.321705285465187
```



The test graph plots the total reward per episode for testing; the horizontal axis corresponds to the 50 episodes of testing while the vertical axis shows the total reward for each episode. During testing, the agent follows its learned policy without exploration, $\epsilon = 0$. The rewards are very diverse-some episodes have very low rewards, almost 0, others as high as nearly 300. It's inconsistent, though-the agent in some episodes performs very well and peaks around episode 30, while in other episodes the reward is much lower. This variability suggests that the agent has learned an OK policy; however, it is not stable in performance for each episode. Spikes indicate that the agent could do well in this or that case, if everything goes just so. The overall inconsistency may point out at least areas where the policy perhaps doesn't generalize well or adapt to different situations occurring within the environment.

Task 4: Create an RL agent for MiniPong (level 3)
In the level 3 version of the game, the observed state (the information available to the agent after each step) consists of three values: y, dx, and dz. These represent the following:

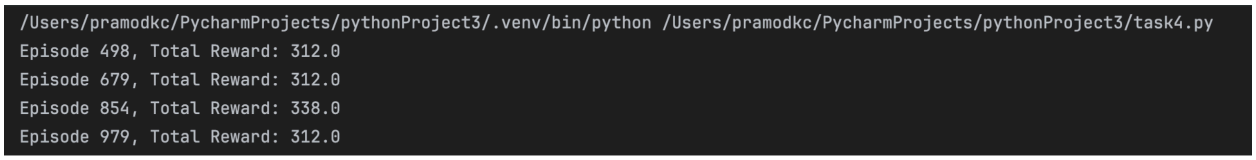
- **y: the ball's y-coordinate.**
- **dx: the change in the ball's x-coordinate from the previous step to the current step.**

- dz: the relative position of the ball to the centre of the paddle (same as in previous levels).
Similar to previous tasks, you can initialise MiniPong in two ways for this task:
10 points
env = M i n i P o n g E n v (level =3 , size =5 , normalise = True)
or
env = M i n i P o n g E n v (level =3 , size =5 , normalise = False)
In the first version, step() returns normalised values for y and dz (between−1 and 1), while in the second version these values are unnormalised. The dx values are always unnormalised (but should be -1 or 1 in most cases, except after the paddle has been hit).

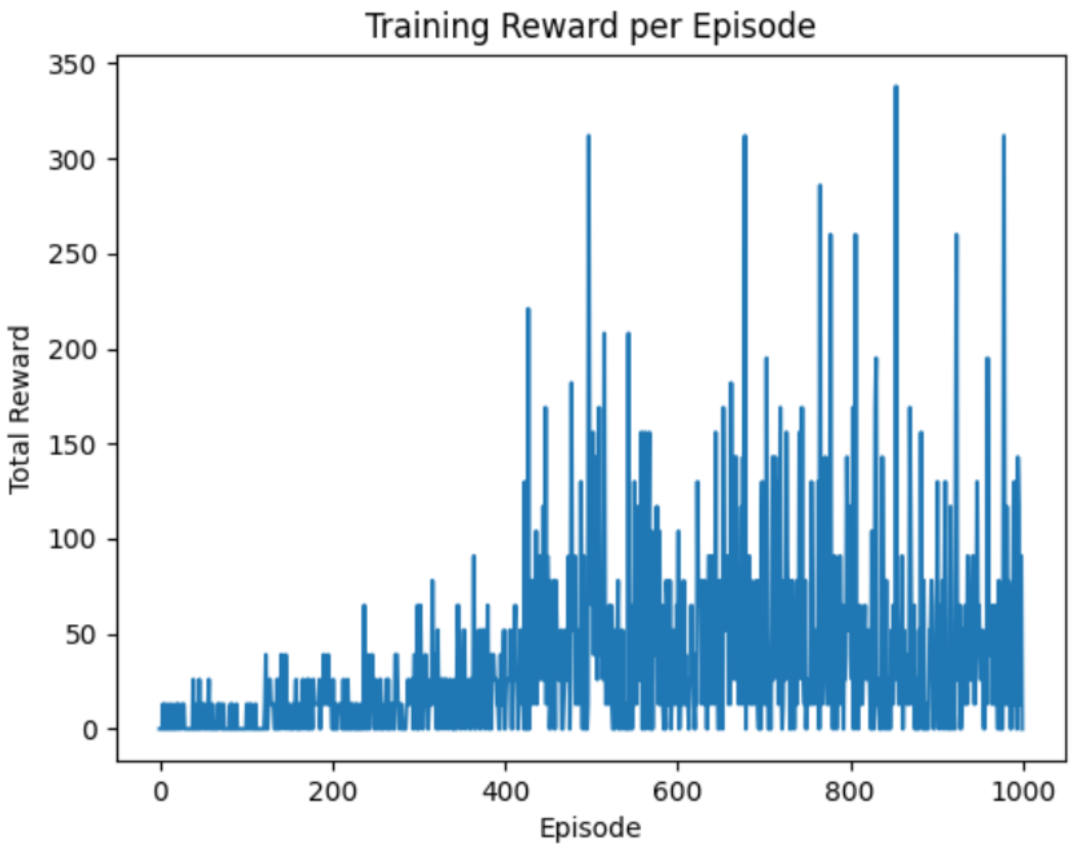
The state for the MiniPong level 3 DQN agent was full state information, including ball y-coordinate, dx of the ball, and relative position of the ball to the paddle. A neural network-based reinforcement learning agent was to be designed that learns an optimal policy for maximizing rewards over time. For the agent, the neural network has been designed with one single hidden layer of 64 units in an attempt to approximate the Q-values for each state-action pair. Wherein, the state vector of input y, dx, and dz feeds into the network, and the Q-value for each possible action is taken as output. This modest architecture has been selected in view of trying to minimize the time spent in training while still allowing the learning of an effective policy by the agent.

The agent utilized the DQN algorithm with experience replay, storing experiences in a replay buffer of form (state, action, reward, next state, and done), sampled in batches of 32 experiences to update the Q-network. An epsilon-greedy strategy that starts with a high epsilon ϵ = 1, for exploration, undergoes an exponential decay until it reaches a minimum of 0.1 to allow the agent to exploit its learned policy in later stages of training. The Adam optimizer updated the model's parameters with a learning rate of 0.001, while the discount factor was set to 0.99 to make sure that the agent would consider long-term rewards.

The agent was trained for more than 1000 episodes, and the total reward earned in each episode was recorded. The Training Reward plot shows that indeed, the performance of the agent had a trend towards improvement over time, starting with low rewards in the early episodes while it was exploring the environment. Indeed, the performance improved markedly at around episode 300, while the rewards became more consistent from about episode 600 onward, even though some fluctuations persisted.



```
/Users/pramodkc/PycharmProjects/pythonProject3/.venv/bin/python /Users/pramodkc/PycharmProjects/pythonProject3/task4.py
Episode 498, Total Reward: 312.0
Episode 679, Total Reward: 312.0
Episode 854, Total Reward: 338.0
Episode 979, Total Reward: 312.0
```



After training, the agent was evaluated over 50 test episodes, during which epsilon was set to 0 (pure exploitation). The Test-Average and Test-Standard-Deviation were calculated based on the total rewards from these test episodes. The average sum of rewards over 50 test episodes was 56.16. The standard deviation of the sum of rewards over the 50 test episodes was 10.322. The standard deviation of the sum of rewards over the 50 test episodes was 52.93.