Evaluate DQN with Different Network Strategies

Table of contens

ABSTRACT

In this assessment, our team is going to build a Deep Q network for the Atari game Asterix with three different input types (RAM, pixel image, and both) to evaluate its performance.

Reinforcement learning from pixel image framework

1.1 Create an environment

Asterix is an arcade-style action game. Players pick up various treasures between 8 panels (this game is mainly a cauldron) while avoiding the lyre. We will use the environment of the terminal Asterix-v0 and import it through make() in the gym library. The environment is loaded as follows:

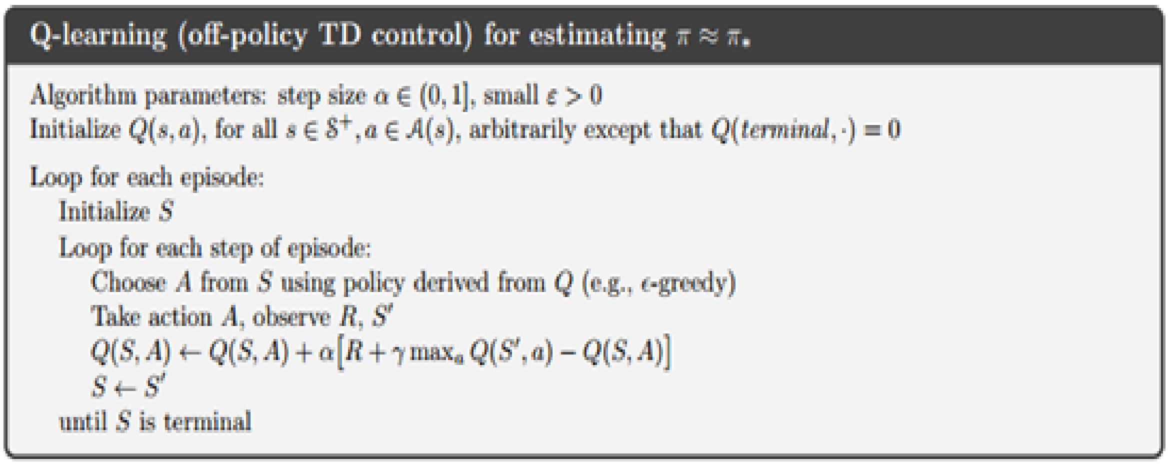
env_1 = gym.make('Asterix-v0')

env_2=gym.make('Asterix-ram-v0')

Finally, two observation results can be output, namely pixels and RAM. The first observation is an array of (210,160, 3). The second type is represented by a byte with 128, and its value is between 0 and 255. This is the RAM state of Asterix in a given game state. The agent can perform 9 actions, namely: 'NOOP', 'UP', 'RIGHT', 'LEFT', 'DOWN', 'UPRIGHT', 'UPLEFT', 'DOWNRIGHT', 'DOWNLEFT'. The rewards for performing the action training environment are obtained from the Atari2600VCS simulator. The following are the reward rules: the agent will receive 50 points of positive reward every time he picks up a cauldron and avoid the deadly lyre of Assurancetourix. When the player touches the lyre, he will consume a life of -1 and receive a negative reward. The original health value is 3. The game restarts when the three health points are consumed（Several lives left for agent will be saved in the info dictionary information）. This game is an endless puzzle game, and the purpose is to allow players to get more points when they are not dead. The difficulty is that the lyre can accelerate as the player's score increases. This is the biggest challenge of this game.

1.2 Q-learning and Deep Q-learning（DQN）

Q-Learning is a value-based algorithm in the reinforcement learning algorithm. Q is Q(s, a). In the s state (s ∈ S) at a particular moment, the action a (a ∈ A) can get benefits.[3] The expectation of the environment will be the corresponding reward (r) according to the agent’s action feedback, so the main idea of the algorithm is to construct a Q-table of State and Action to store the Q value, and then select according to the Q value to obtain the maximum benefit Actions. The purpose of the Q-learning algorithm is clearly to find a strategy that can obtain the greatest reward. The algorithm pseudo-code is described as follows:



```
Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize $S$
  Loop for each step of episode:
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
    Take action $A$, observe $R$, $S'$
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until $S$ is terminal
```

Among them, α is the learning rate, and γ is the discount factor. Solving the Markov decision process from the Bellman equation, summed up the updated state formula of the previous step as:

$$Q\left(S_t, A_t\right) \leftarrow Q\left(S_t, A_t\right) + \alpha\left[R_{t+1} + \gamma \max_a Q\left(S_{t+1}, a\right) - Q\left(S_t, A_t\right)\right]$$

maxaQ(S’,a) refers to the Q value of the action A with the maximum Q value in the next state S ′, and finally continues to iterate [1]. In practical applications, most of the problems are that there is a huge state space or action space. For example, if you want to create a Q table in this game task, memory is absolutely not allowed, and the amount of data and time overhead are also problems. At this time, you need to use the deep q learning network (DQN).

The basic idea of the Deep Q-Learning algorithm comes from Q-Learning. However, the difference from Q-Learning is that the state values and actions do not directly calculate its Q value. The above Q value is stored in a table, but if there are many state actions, the space consumption is too large. So replace this Q table with a network, that is, a Q network. Recalling Q-Learning, we update the Q table using the reward of each step and the current Q table to iterate. [4] Compared with Q-learning, DQN learning is not linear, and learning like Q-learning is linear. Then we can use this calculated Q value as the "label" for supervised learning to design the Loss Function. We use the following form, which is the mean square error between the approximate value and the actual value [2,3]:

$$J(w) = \mathbb{E}_{\pi}\left[(q_{\pi}(s,a) - \hat{q}(s,a,w))^2\right]$$

The input of DQN is the state vector ϕ(s) corresponding to our states, and the output is the action value function Q of all actions in this state. The Q network can be DNN, CNN, or RNN, and there is no specific network structure requirement. In this task, we use CNN. The main idea in DQN is the memory bank (sample set). Each sample contains the current state, the current action, the reward for the action, and the next state. After each action is completed, CNN does not directly learn the current data but stores it in the memory bank and randomly selects one from the memory bank to learn. This can cut off the correlation between the data.

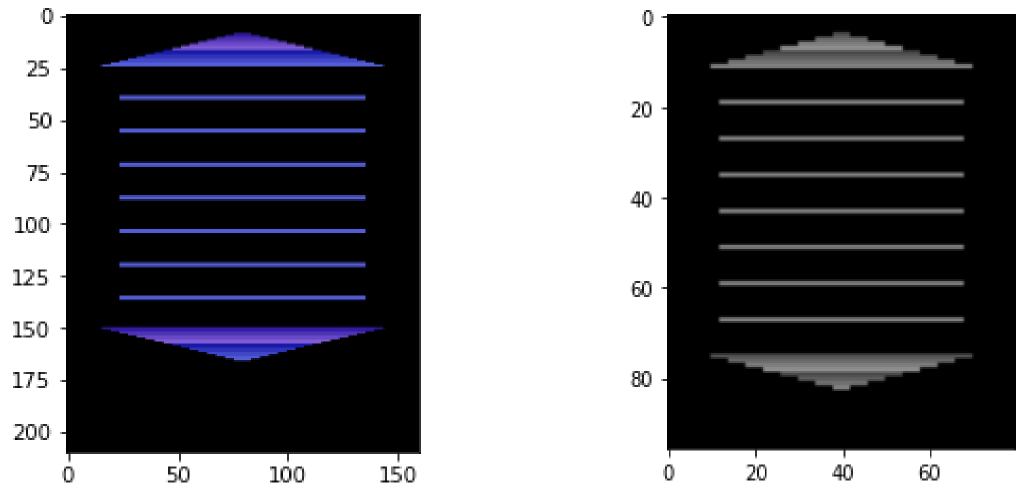
1.3 Pretreatment

In the pre-processing in the screen frame, to reduce the amount of original image data and facilitate the subsequent processing with less calculation, we have carried out the gray-scale processing of the color image. At the same time, these values are converted to 8-bit unsigned integers (uint8) to reduce the memory further. The area we need to use in the original image is not that large, so to further process the image, we cropped the original image, cropping (210,160, 3) to (96,80,3). Improve our training speed and reduce the complexity of the network. The tailoring method we adopted is sub-sampling because the CNN contains a fixed-size feature detector. Once a feature in an image is detected, information about the feature's location in the image can be ignored. The code is show as below:

```
def preprocess_observation(observation):

    img = observation[1:192:2, ::2] #This becomes 96, 80,3

    img = img.mean(axis=2) #to grayscale (values between 0 and 255)
    img = (img - 128).astype(np.int8) # normalise from -128 to 127
return img.reshape(96, 80, 1)
```



(a)before (b)after

Figure1: Preprocess images

1.4Agents for deep convolutional networks

1.4.1Build the CNN

For the first task, the Q network we use is CNN. Before we start, we need to divide by 255 to reduce the intensity of the gray-scale pixel to [0,1] because the pixel value is between 0 and 255. Next, we use the variance scaling initializer to define the network's weights. Then our CNN model is composed of three convolutional models, as shown in Appendix [1.4.1] below. The first convolution layer consists of 32 filters, with an 8X8 kernel and a 4 strides. The second layer comprises 64 filters with a 4X4 kernel and 2 strides. The last layer comprises 64 filters with a 3X3 kernel and 1 stride. Every layer has a Relu activation function for the better performance of CNN. The specific code is shown in Appendix [1.4.2].

1.4.2Build the Agent

For building the agent, we created two networks. One of them is the online network, which calculates the best auction result through the target network. The other is the target network, through which the best auction results are stored, and the online network is iterated as the training progresses. In the construction of the proxy model, we also calculated the error to define the loss function, that is, through the current Q value and our target Q, and then square it. This will make our online Q value closer to the target Q value. Then built the Adam optimizer global_step to reflect the variables of our progress in order to track when to update our target network to reduce the loss of training. Then the training progress is stored by self. Saver to prevent the process from being interrupted, and if the saved file exists, it is restored. If there are no new variables to initialize, create an online network and a target network. Next, create a get_action function to calculate the Q value in a specific state. At the same time, choose to use the optimal Q value or choose according to the value of epsilon. The epsilon value will change at any time, so you need to use max to limit its value to not less than 0.1.

We also define a class called PrioritizedReplyBuffer that store the data return by *env.set(action)* as the experience ordered by the value of error. When the agent is going to make action, the agent will also take the experience into account.

We define a *train* function for the training process of our agent. The function will take a batch of priorized experience from experience replay buffer as the input. Then the function will use the value of the variable *next_state* to calculate the next q-values, and select the corresponding action based on the calculated q-values. Then, the function will calculate the target values obtained by taking the corresponding reward from experience, next q-values, and discount rates. Finally, the function will update the weights and biases of the neural network based on the results obtained, and calculate a new loss value. Finally, the function saves the experience gained in this training into the replay buffer, and assigns a priority to it according to the value of loss.

1.4.3 Adjust hyperparameters

For better implementation, we will use the method of controlling variables.[5] In our models, unless otherwise specified,we will use the following consistent hyperparameters shown in Table-1.

Hyperparameter	Value	Description
batch_size	32	Number of training cases over which each network update computed.
maxlen	1000	The velocity of prioritized experience replay buffer is

		1000 experience maximum
Activation function	ReLU	Name of the algorithm optimizing the layers in the neural network.
Learning rate	0.0001	The learning rate for Adam optimizer
Discount	0.97	Discount factor γ used in the Q-learning update. Measures how much loss do we value the expectation of the value of the state in comparison to observed
max_epsilon	1.0	The probability of choosing a random action, as well as the ε at the beginning.
min_epsilon	0.1	The minimum value of ε can be decayed to
Episodes	500	How many episodes the agent would play the game
train_step	5	The agent trains the model every 5 episodes

Table-1. Hyper-Parameters Table

1.5Training of agents and assessments

This training was conducted on Google ColaboratoryPro. At the beginning of each training set, a training environment is initialized, and a total of 500 training sets are completed. After training, we showed the first episode, showing the total number of training steps and the reward for that episode (for the convenience of the analysis later, we will get a reward set to 1 point). Each episode loops one training step during the training until the Agent encounters a lyre three times. In the episode, all steps are tracked and acquired from the model optimizer, the target network is updated, and checkpoints are saved. When the episode ends, set the next state to the current state of the next episode.

In order to observe the specific results and analyze them, We have a train_step = 5, and we train every 5 times. This way we can watch the training results while training. It can be seen from the line chart Appendix [1.5.1] that the results obtained by the agent show an overall upward trend. However, there was a significant decline in 18-20, 33-42, 68-70, and 87-95. In the 87 groups, the highest score reached 8 points. In general, the scores of agent training are getting higher and higher, and it is also a good place. There is no 0 point in the agent training process. It can be seen from the scatter chart Appendix [1.5.2] that the scores of the agents are mainly concentrated in 4.5-5 points. The next pixel, RAM, and mixed graph will compare the scatter plot and the line graph to see the overall score.

RL BASED ON THE RAM INPUT

2.1 RAM description

In this session, we will focus on the training by using RAM. Unlike the game using pixel, the game's input using RAM is not the game's screen but the game's ram state. For the Asterix game, its ram state is 128bytes of ram. Since the input is smaller than the input using the image, it means that our training will be faster and easier. However, because the ram state of the game is stored in the memory, it makes it difficult for us to obtain game information.

In the figure below, we have drawn a heatmap to view the important ram state in observation when inputting ram. The colors on the picture are from light to dark, representing the numbers from 0 to 255. In the initial state, the value under some bytes is large, which represents the state of several bytes that are important to the initial state.As the game progresses, this map will change, and the values of the opposing heatmap will also be different.

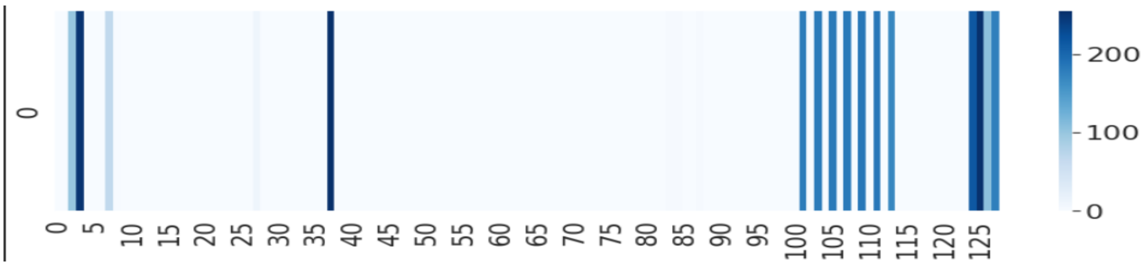


Figure2: Heatmap of inputting RAM

2.2 Develop an agent equipped with a dense neural network

In this part, we will discuss how to build an agent and neural network for an Asterix game that only uses RAM, so that the agent can learn from it and is able to choose the best action. There are two manifestations to represent the game frame to our agent; one is the image, image is an array with a shape of (210x160x3), the other one is RAM series of the frame, the series is (128,). When the ram game is used as the neural network input, we will have a one-dimensional input: (128,), which is different from using image frames as input in the previous part. Obviously, when the input data is one-dimensional, it is more reasonable to use a fully connected network for analysis.

When building the `ram_model` function, we created a neural network with five dense layers: the first dense layer receives scaled X_state_ram as input, uses a unit size of 128, and the activation function uses Relu. The next three dense layers sequentially receive the previous layer's output as input. The unit uses 128, and the activation function uses Relu. The last layer is the output layer, and the dense layer is still used. The output of the previous layer is the input, and the final output is the action space, which is a one-dimensional sequence of (9,). The neural network structure is shown in the Appendix [2.1] below.

After building the model, we need to define the RAM frame agent. Our agent still uses the agent defined in the previous part, and the detailed parameters are set to the same parameters as the previous part. The only difference is that the input shape of the model is changed. By using the same agent, we can compare the performance of Pixel and Ram.

2.3 Train the agent and evaluation

After establishing the neural network and agent, we start the model's training. Same as the previous part, we still use the same parameters and train 500 episodes in this part. And still, use frame skipping, let the agent learn once every 5 times of training. After 500 episodes of learning, scatter plots and line graphs were drawn based on the episode and the reward of each episode. Then, we drew the corresponding scatter plots and line graphs based on the average of every 5 episodes in the training process. We have plotted the violin diagram and box diagram to evaluate the reward.

In the scatter plot of Appendix [2.3.1], we can see that after the model is trained three hundred episodes, points with the reward of 0 are no longer generated, and the range of points is roughly around reward=5. A small number of points are distributed in the range of 10-15. The highest value is between 100 and 150 episodes, and the maximum value is close to 25.

From Appendix [2.3.2], we can observe that there is no obvious trend in the change of reward with the increase of episodes. The reason may be due to too few training episodes. In this experiment, we only trained 500 episodes, which may make the agent not accumulate enough experience to learn.

To further refine the model, we can increase the training episode to see if the reward can increase accordingly. Then, we can adjust parameter settings, such as learning rate. High learning rate can make parameter adjustment faster, but it will be more noisy. We can use a lower learning rate in the future. In addition, we can adjust the structure of the model or try to add some other layers to try to get better model performance and further optimize the model.

2.4 Comparison of RAM and Pixel

By observing the mean value of reward obtained from all episodes of the two models, we found that the two models have little difference in the mean reward performance. The pixel-based average reward is 5.052, and the ram-based average reward is 5.040. These two values are very close. It may indicate that there seems to be no obvious difference between the advantages and disadvantages of using RAM and Pixel. In order to compare the distribution of reward of ram and pixel models, we plot a violin graph (Appendix [2.4]) to compare the two experiments. It can be seen that the maximum value of ram is higher and can reach close to 25, while the maximum value of the pixel-based model is only about 15. Moreover, the rewards of these two models are mostly distributed around 5, but the model using the pixel frame has more sample distributions between 4.5 and 5 than the model using the ram.

In summary, the two models based on ram and pixel have little difference in performance when episode=500. Perhaps when episodes have a large number of training sessions, their differences will become more apparent.

RL BASED ON THE MIX INPUT

3.1. Network and Agent Design

We have implemented two neural network models for our evaluation in the previous sections. In this section, we wonder whether there is any difference we can train an agent by mixing the inputs of image and RAM environments.

The idea is to re-build the neural network model by concatenating the output of the last hidden layer of CNN with FC. The game frame has two different representations. One is a frame image, and the other one is RAM series. Thus, we know that when the agent executes an action to make the frame from *frame 1* to *frame 2*, each state has two different manifestations. Therefore, we will keep these two manifestations as the input variables of our mixed model. We will keep the architecture of the CNN model in the last section and cut three dense layers we used in RAM to boost the training speed. Then we will concatenate the outputs of CNN and RAM dense layers. Finally, we will use a dense layer again to figure out q-values. For controlling the variables, we will adopt the hyperparameters according to *table-1* are the same as the last two sections as well. As the active agent will only make actions to the game frame, the variables of actions, rewards, info is the same. Thus, we do not have to use two manifestations to represent them in the neural network model. The Appendix [3.1.1] shows the architecture of our mix model. The code snippet is attached in Appendix [3.1.2].

3.2 Training and Evaluation

Because we decided to adopt the variable-controlling method to conduct our experiment, we will play the agent for 500 episodes and set up the *frame_skip_rate* as 5. Thus, We will train the agent every 5 episodes of playing the game. When the agent finishes all the episodes, we will plot the line and scatter of *episodes rewards, mean rewards of every 5 episodes* of the training, and calculate the overall average rewards of 500 episodes game. We will make the same state of the environment with two manifestations as two inputs for our mixed network. In the *train* function, we will revise the input of the replay buffer as well. In the before, the train function only took the current observation and the next observation from the replay buffer, we will amend the function to take two current observations and the next observations.

After the training step is completed, we can calculate the average reward of 500 episodes is 5.202. Because the line plot of reward in per episode is too narrow mussy to observe so we made the scatter plots instead, we still made a line plot based on the average reward of each 5 episodes. From the scatter Appendix [3.2.1], and line Appendix [3.2.2], we can observe that most of the rewards values are concentrated in the interval 4.5 to 5.5. And from the overall observation, we found that as the number of game episode increases, the rewards value does not have a significant upward trend. Therefore, we may need to compare the training scores of other models to get our conclusions.

3.3 Comparison img_mode vs. ram_mode vs. mix_mode

We can make observations from the scatter and violin plots are shown in Appendix [2.4] and Appendix [3.5] that the main rewards of each episode are dropped in the interval from 4.5 to 5.5. From the scatter and line plots shown in the appendix[3.2.2] and appendix[3.2.1], we can observe that the training does not have a significant influence on the performance of the agent as the curve does not have a clear upward trend. Although none of the plots show a significant upward trend in Appendix[3.5] and Appendix[3.6], we can still tell the different influences when we adopt different training modes from Appendix [2.4]. Comparing mixed-mode to image mode, there are more values staying between 4.5-5 in image mode, and the maximum value is only 15. In mixed mode, more values tend to be distributed in a higher range, for example, $rewards > 7.5$. The maximum value of mixed-mode is higher than that of the image model and has more distribution in the interval of $rewards > 15$. Comparing to ram mode, in the mixed-mode, although the highest score in the mixed-mode game is not as high as in the ram mode, we can observe that there are more values starting to distribute towards the higher part, for example, $rewards > 10$, the mixed mode has more sample distributions in the interval where $rewards > 10$.

Overall, the performance of mixed-mode is better than that of image mode and ram mode because after adopting the training strategy of mixed mode, the overall reward values start to move towards a higher interval. In the calculation of the maximum value, the value of image mode is greater than that of mixed mode, but from a statistical point of view, some values of sporadic distribution can be treated as outliers, and are not considered in the overall distribution characteristics of the sample. This also proves that the rewards distribution obtained from the mixed-mode training strategy is more stable and generalized.

CONCLUSION

From the simple scatter plot, line graph of rewards, and the table we summarize in Appendix, we cannot observe that the trained agent has a significant improvement in the performance of the game. But by comparison, we can still observe the impact of using different training models on game performance. After 500 rounds of training, the agent's performance in the game is roughly the same. We can see this from the average score. By comparing RAM and image mode, we can observe that the distribution of agent performance scores has begun to show an upward trend, although there are not many scores in the high score range. And the highest score of the agent has refreshed the record, although in statistics, such a value can be ignored as an outlier. Furthermore, by observing the training results of the Mix-model, we can find that the scores of the agent are distributed in the rounds between the high partitions more than the previous two models. But the mixed-mode cannot easily refresh the scoring record. Therefore, we recommend using the mixed-mode to train the agent, so that the model can make up for the slow improvement of the image modes, and it can also make up for the shortcomings of the ram mode that are prone to outliers. Mixed-mode is a training strategy that can find a balance between the deficiencies of image mode and ram mode.

FUTURE WORK

In this experiment, the agent's performance after training did not improve much because the number of model training was too small. In the entire 500 rounds, the model was trained 100 times in total. So if we can address this point in future experiments, we can make the following improvements:

1. Optimize the structure of the model and the code of the agent to reduce the cost of time spent in a single training and a single game. This way we can train more times per unit time.

2. Increase the number of training rounds. We can increase the number of game episodes to 5000, or even 100,000 so that the model can calculate more accurate weights and paranoia values according to the environment.

In our experiment, we adopted a method of training while testing, that is, training once every 5 episodes to observe the agent's performance after training. In the case of a small number of training sessions, it is difficult to clearly distinguish the score after each training is completed from the score before the training. Therefore, in the future experiment, we can redesign the structure of the agent. We can integrate the neural network model into the agent so that the agent can be trained separately according to the observation parameters of the game environment. After the training is completed, we will put the agent into the real game environment, let the agent directly start the game, and then observe its score and performance. After training for N times, if we are not satisfied with the agent's performance in the game environment, then we can continue to train the agent for more epochs on the original basis. This method is similar to the traditional machine learning training method. The model is trained for a certain number of epochs and verified on the verification set. The train set and the validation set are two different sets, which are determined by the observation, action, info, etc. of the game environment data. Finally, the model will be tested on the test set. The test set is a real game environment. The agent controls the game itself. Everything that happens in the game is unknown.

REFERENCE

[1] Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction - Chapter 9: On-policy Prediction with Approximation. MIT press.

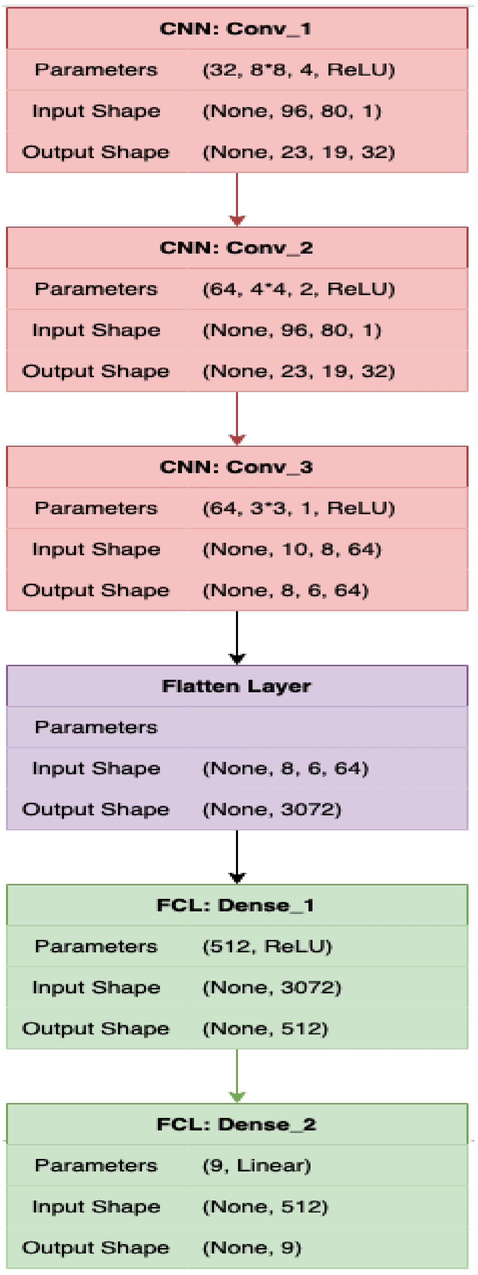
[2] Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction - Chapter 10: On-policy Control with Approximation. MIT press.

[3] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Hassabis, D. (2015). Human-level control through deep reinforcement learning. nature, 518(7540), 529-533.

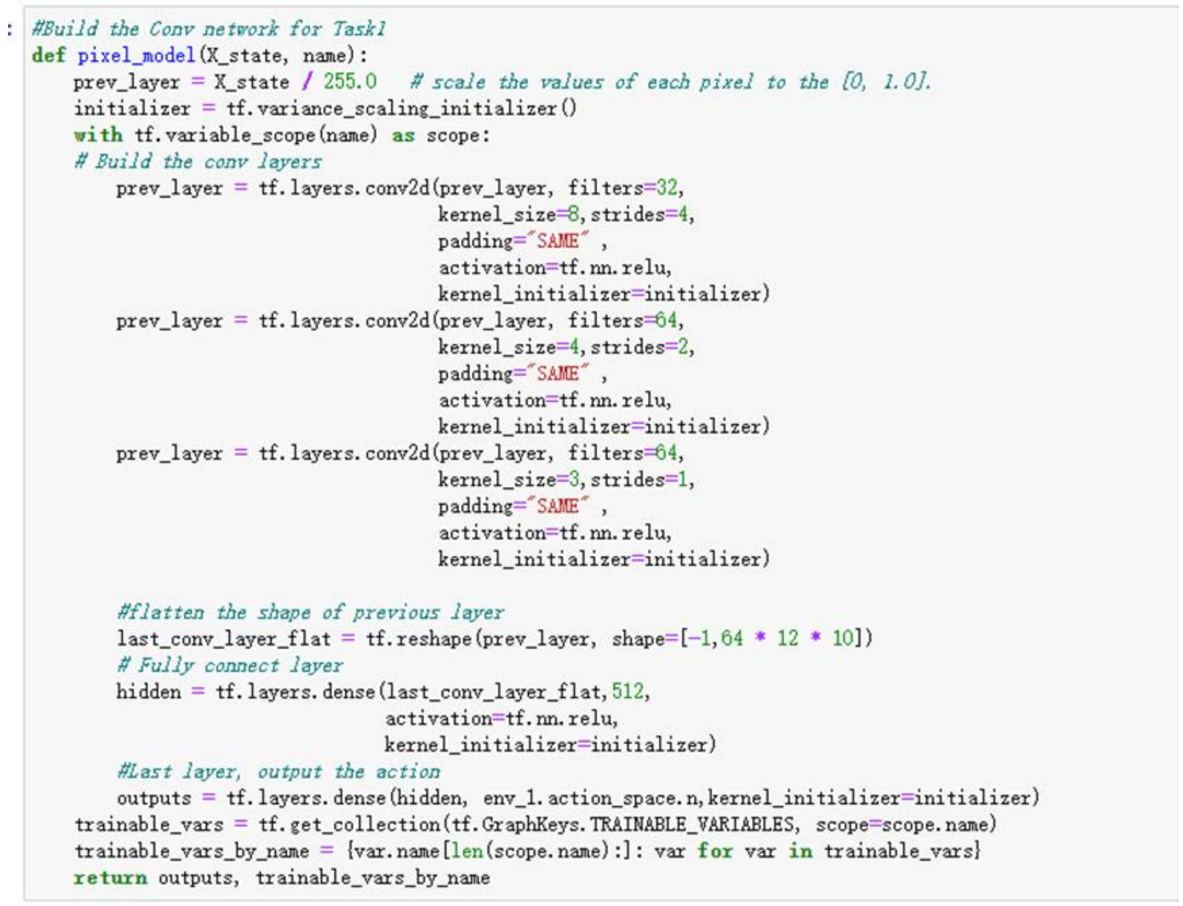
[4] Van Hasselt, H., Guez, A., & Silver, D. (2016, March). Deep reinforcement learning with double q-learning. In Proceedings of the AAAI conference on artificial intelligence (Vol. 30, No. 1).

[5] Hansen, S. (2016). Using deep q-learning to control optimization hyperparameters. arXiv preprint arXiv:1602.04062.

APPENDIX


CNN: Conv_1
Parameters (32, 8*8, 4, ReLU)
Input Shape (None, 96, 80, 1)
Output Shape (None, 23, 19, 32)
CNN: Conv_2
Parameters (64, 4*4, 2, ReLU)
Input Shape (None, 96, 80, 1)
Output Shape (None, 23, 19, 32)
CNN: Conv_3
Parameters (64, 3*3, 1, ReLU)
Input Shape (None, 10, 8, 64)
Output Shape (None, 8, 6, 64)
Flatten Layer
Parameters
Input Shape (None, 8, 6, 64)
Output Shape (None, 3072)
FCL: Dense_1
Parameters (512, ReLU)
Input Shape (None, 3072)
Output Shape (None, 512)
FCL: Dense_2
Parameters (9, Linear)
Input Shape (None, 512)
Output Shape (None, 9)

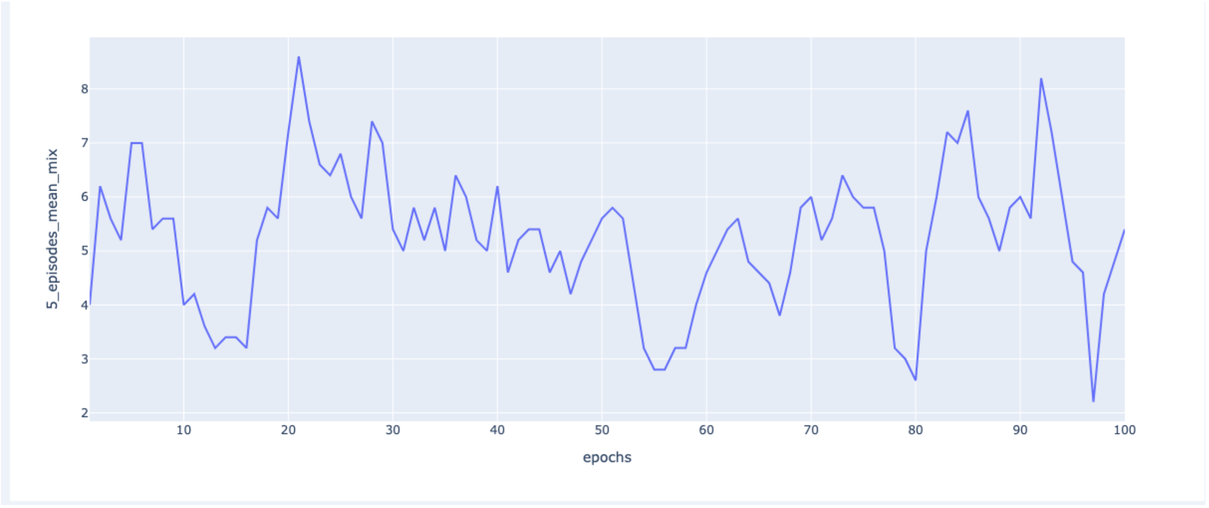
Appendix [1.4.1]: Training model network structure for pixel input



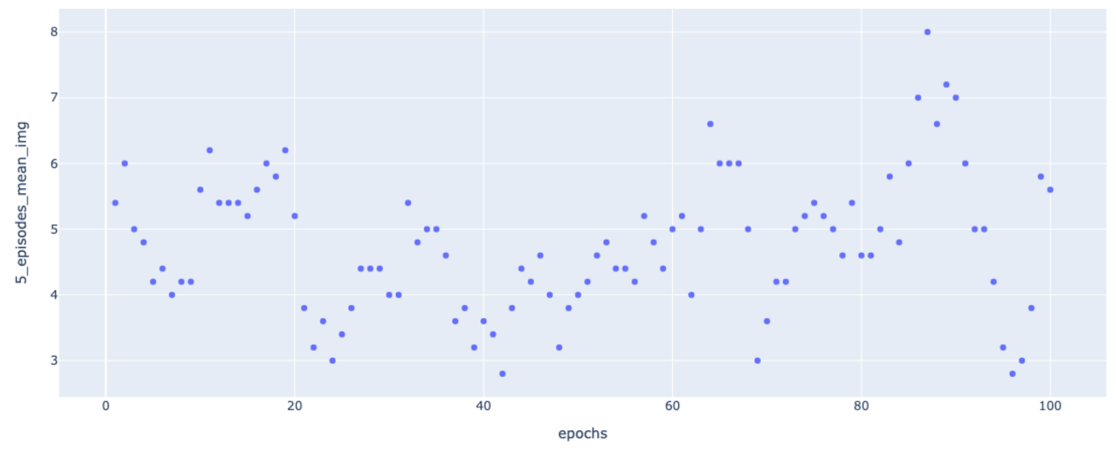
```
: #Build the Conv network for Task1
def pixel_model(X_state, name):
    prev_layer = X_state / 255.0   # scale the values of each pixel to the [0, 1.0].
    initializer = tf.variance_scaling_initializer()
    with tf.variable_scope(name) as scope:
    # Build the conv layers
        prev_layer = tf.layers.conv2d(prev_layer, filters=32,
                                      kernel_size=8,strides=4,
                                      padding="SAME" ,
                                      activation=tf.nn.relu,
                                      kernel_initializer=initializer)
        prev_layer = tf.layers.conv2d(prev_layer, filters=64,
                                      kernel_size=4,strides=2,
                                      padding="SAME" ,
                                      activation=tf.nn.relu,
                                      kernel_initializer=initializer)
        prev_layer = tf.layers.conv2d(prev_layer, filters=64,
                                      kernel_size=3,strides=1,
                                      padding="SAME" ,
                                      activation=tf.nn.relu,
                                      kernel_initializer=initializer)

        #flatten the shape of previous layer
        last_conv_layer_flat = tf.reshape(prev_layer, shape=[-1,64 * 12 * 10])
        # Fully connect layer
        hidden = tf.layers.dense(last_conv_layer_flat,512,
                                 activation=tf.nn.relu,
                                 kernel_initializer=initializer)
        #Last layer, output the action
        outputs = tf.layers.dense(hidden, env_1.action_space.n,kernel_initializer=initializer)
    trainable_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope=scope.name)
    trainable_vars_by_name = {var.name[len(scope.name):]: var for var in trainable_vars}
    return outputs, trainable_vars_by_name
```

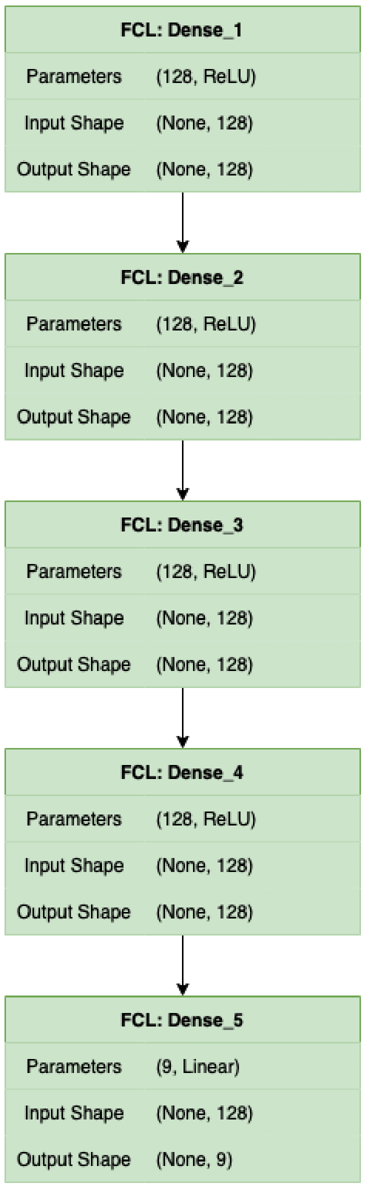
Appendix [1.4.2]: Training model for the pixel input



Appendix [1.5.1]: Pixel reward line-type plot



Appendix [1.5.2]: Pixel reward scatter plot



FCL: Dense_1	
Parameters	(128, ReLU)
Input Shape	(None, 128)
Output Shape	(None, 128)

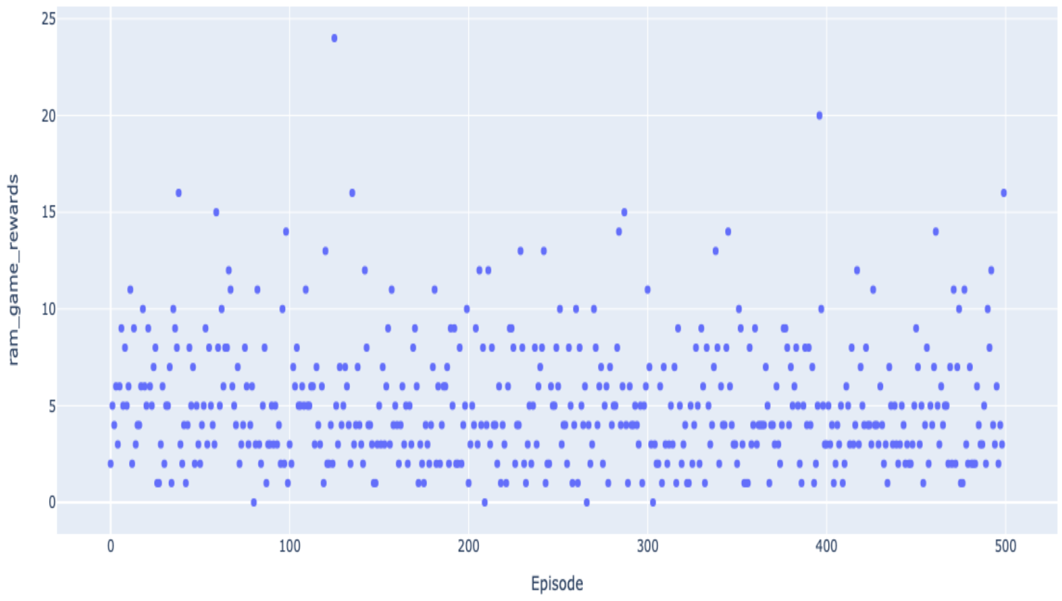
FCL: Dense_2	
Parameters	(128, ReLU)
Input Shape	(None, 128)
Output Shape	(None, 128)

FCL: Dense_3	
Parameters	(128, ReLU)
Input Shape	(None, 128)
Output Shape	(None, 128)

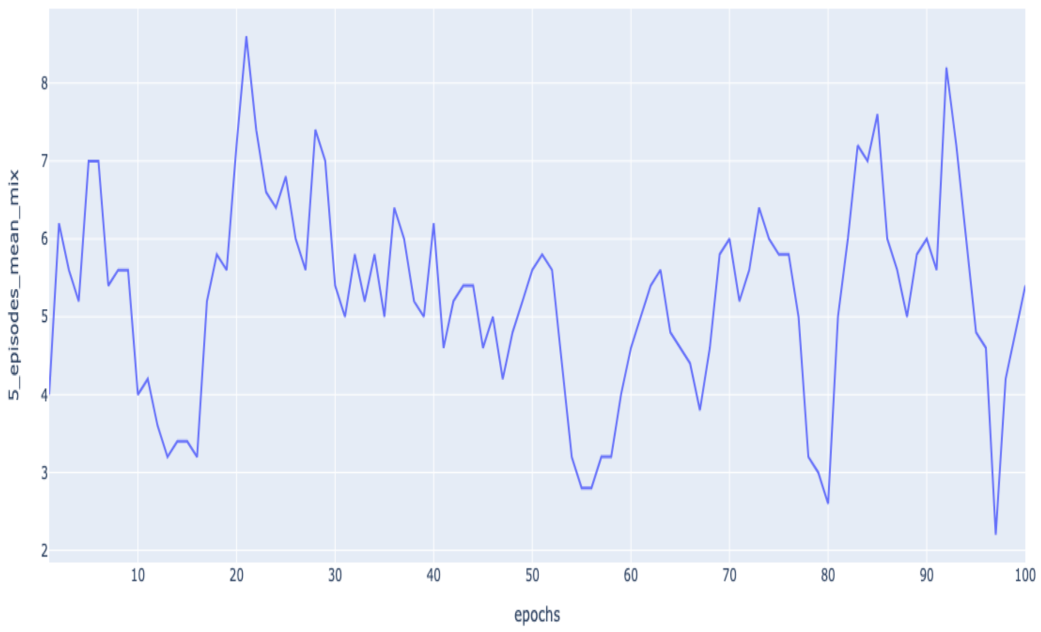
FCL: Dense_4	
Parameters	(128, ReLU)
Input Shape	(None, 128)
Output Shape	(None, 128)

FCL: Dense_5	
Parameters	(9, Linear)
Input Shape	(None, 128)
Output Shape	(None, 9)

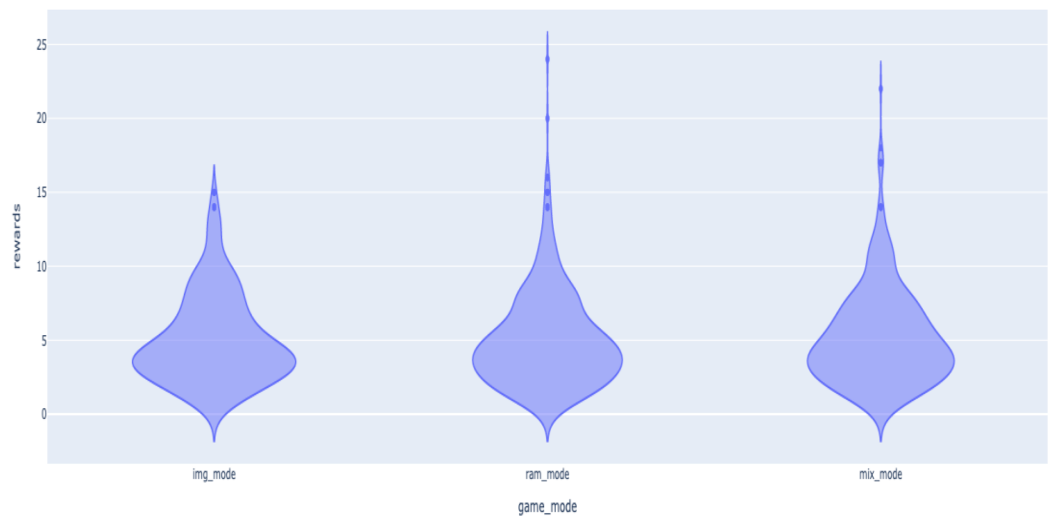
Appendix [2.1]: Training model network structure for RAM input



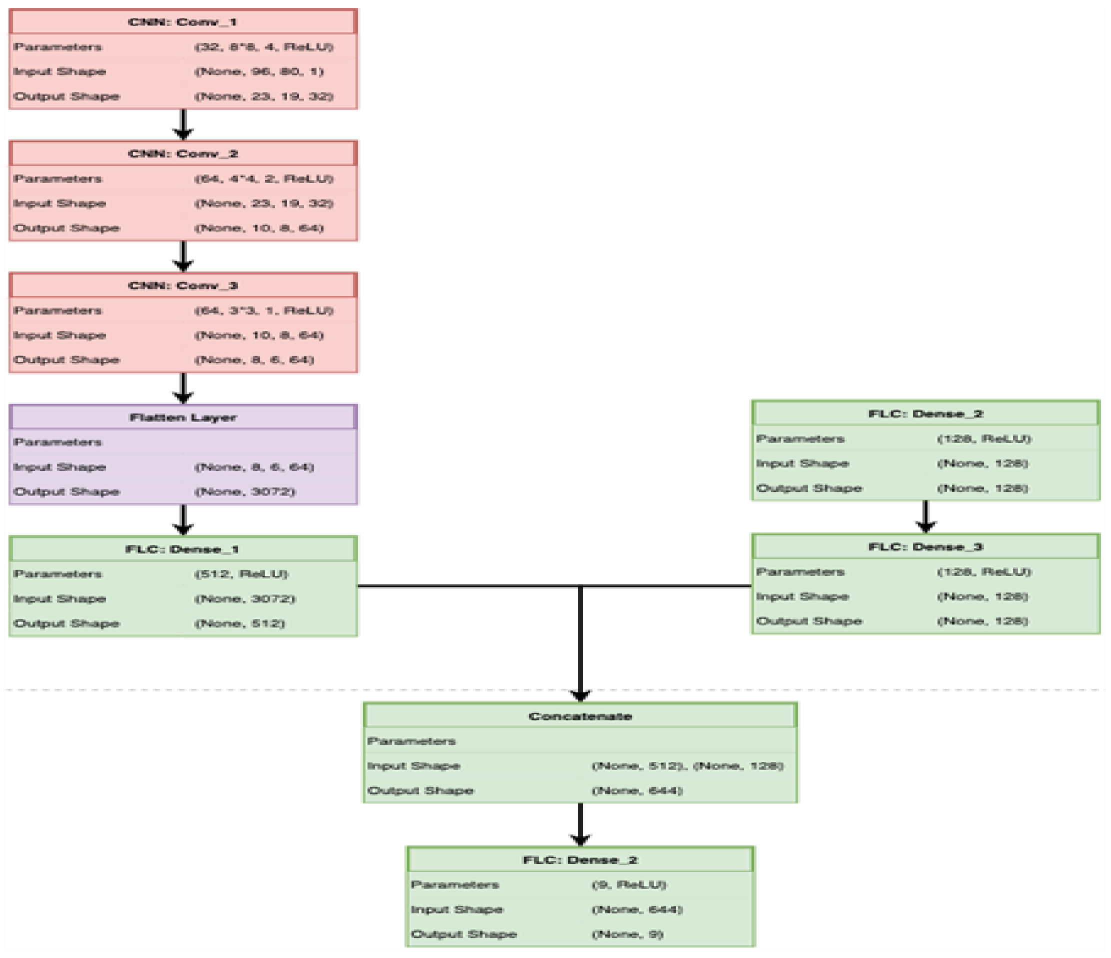
Appendix [2.3.1]: RAM reward scatter plot



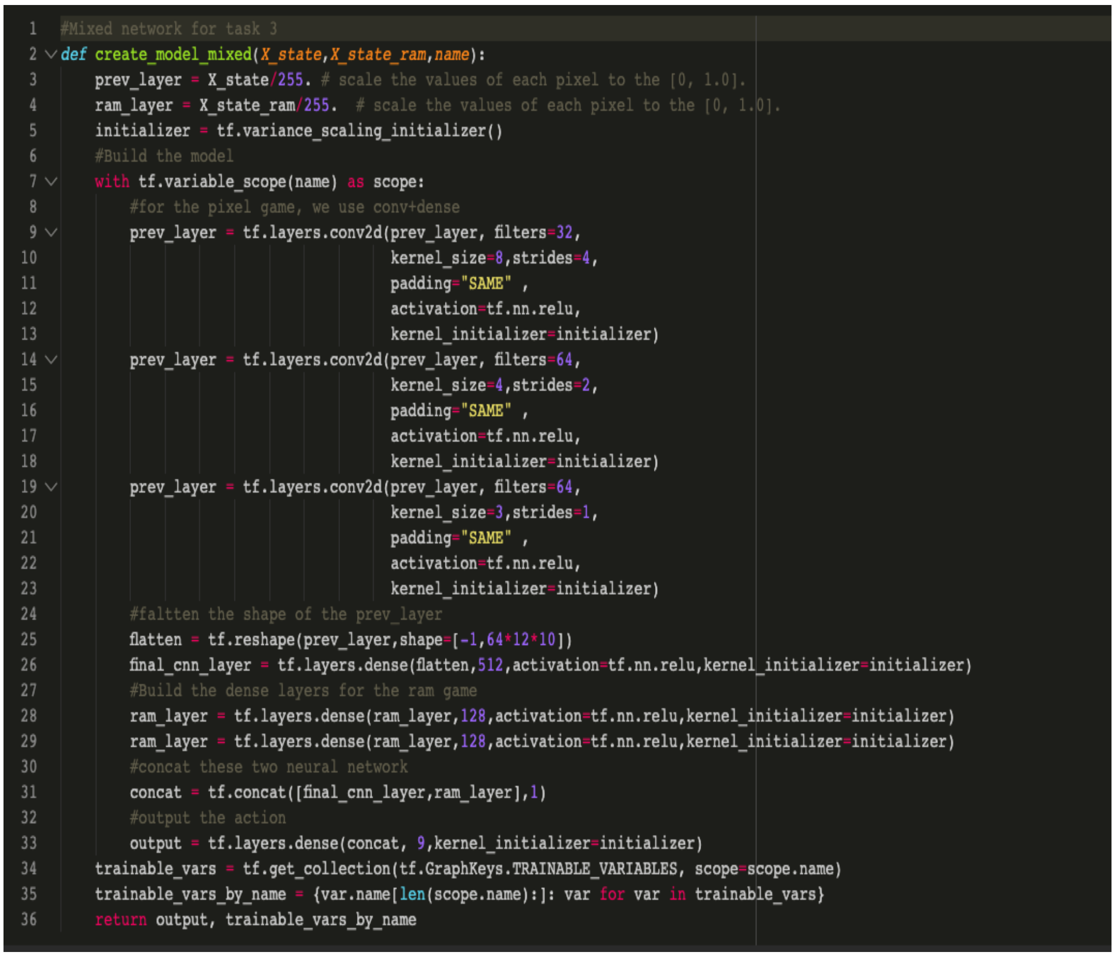
Appendix [2.3.2]: RAM reward line-type plot



Appendix [2.4]: Pixel， RAM and Mix violin graph

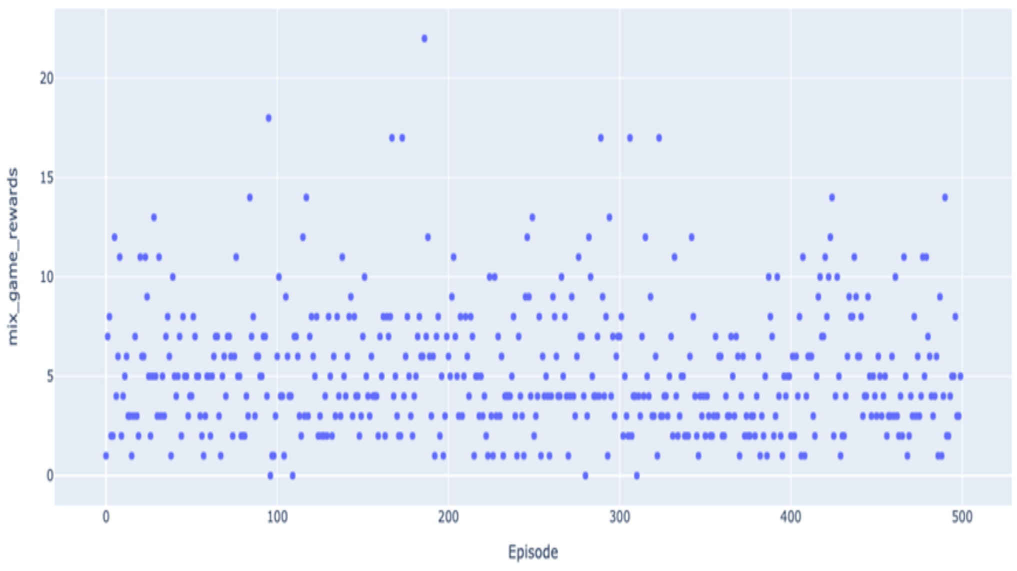


Appendix [3,1.1]: The architecture of Mixed Network

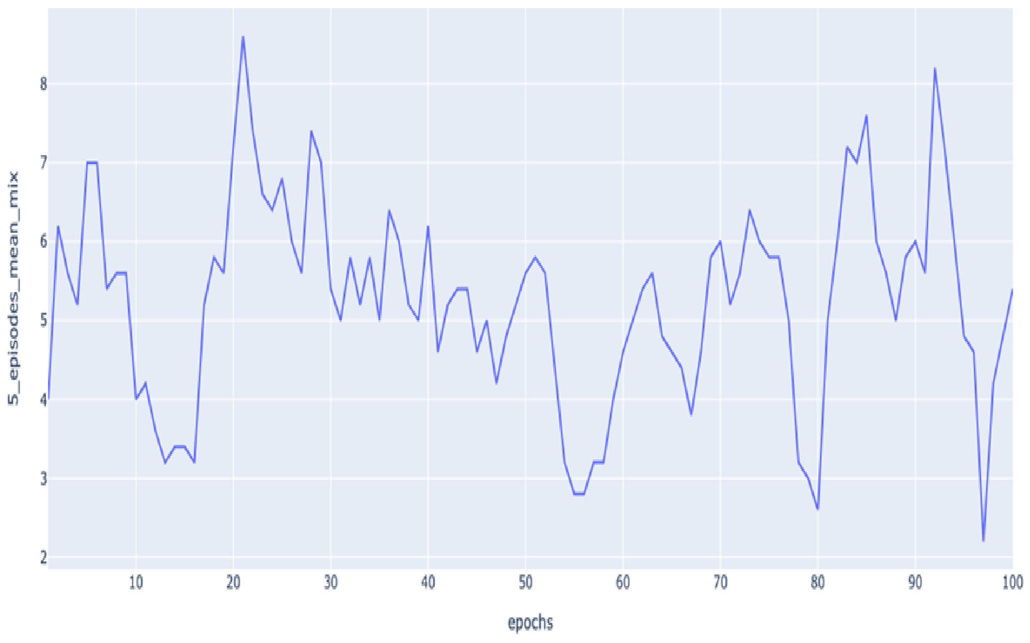


```
#Mixed network for task 3
def create_model_mixed(X_state,X_state_ram,name):
    prev_layer = X_state/255. # scale the values of each pixel to the [0, 1.0].
    ram_layer = X_state_ram/255.  # scale the values of each pixel to the [0, 1.0].
    initializer = tf.variance_scaling_initializer()
    #Build the model
    with tf.variable_scope(name) as scope:
        #for the pixel game, we use conv+dense
        prev_layer = tf.layers.conv2d(prev_layer, filters=32,
                                      kernel_size=8,strides=4,
                                      padding="SAME" ,
                                      activation=tf.nn.relu,
                                      kernel_initializer=initializer)
        prev_layer = tf.layers.conv2d(prev_layer, filters=64,
                                      kernel_size=4,strides=2,
                                      padding="SAME" ,
                                      activation=tf.nn.relu,
                                      kernel_initializer=initializer)
        prev_layer = tf.layers.conv2d(prev_layer, filters=64,
                                      kernel_size=3,strides=1,
                                      padding="SAME" ,
                                      activation=tf.nn.relu,
                                      kernel_initializer=initializer)
        #faltten the shape of the prev_layer
        flatten = tf.reshape(prev_layer,shape=[-1,64*12*10])
        final_cnn_layer = tf.layers.dense(flatten,512,activation=tf.nn.relu,kernel_initializer=initializer)
        #Build the dense layers for the ram game
        ram_layer = tf.layers.dense(ram_layer,128,activation=tf.nn.relu,kernel_initializer=initializer)
        ram_layer = tf.layers.dense(ram_layer,128,activation=tf.nn.relu,kernel_initializer=initializer)
        #concat these two neural network
        concat = tf.concat([final_cnn_layer,ram_layer],1)
        #output the action
        output = tf.layers.dense(concat, 9,kernel_initializer=initializer)
    trainable_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope=scope.name)
    trainable_vars_by_name = {var.name[len(scope.name):]: var for var in trainable_vars}
    return output, trainable_vars_by_name
```

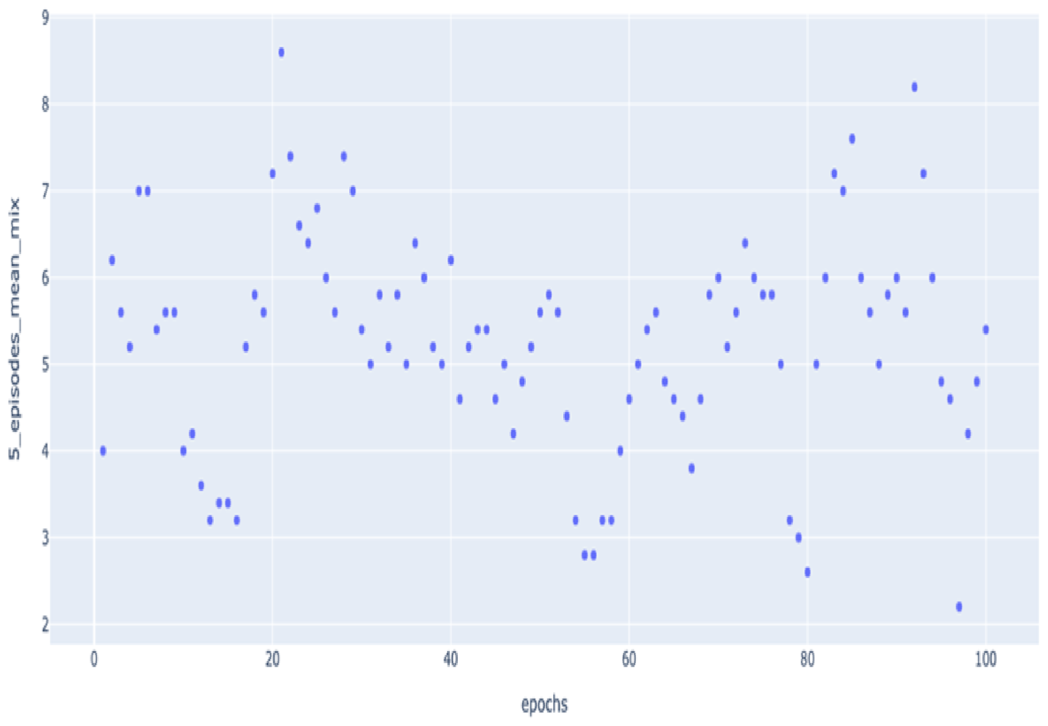
Appendix [3.1.2]: Training model for the MIX input.



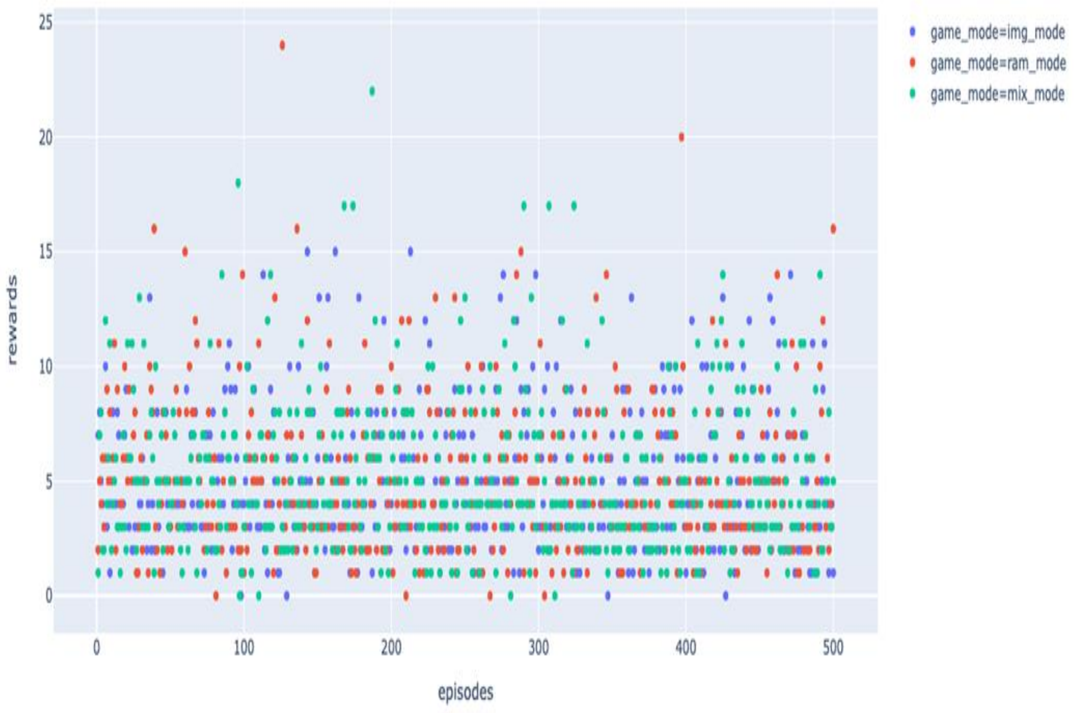
Appendix [3.2.1]: Scatter plot, rewards obtained from per episode.



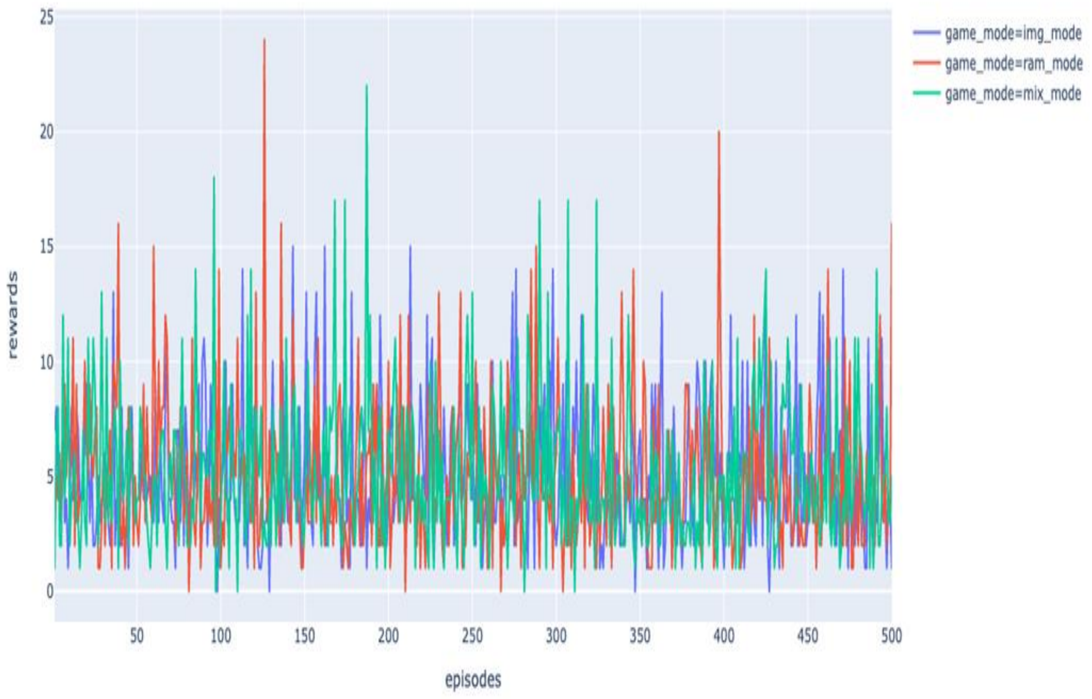
Appendix [3.2.2]: Line plot, 5 episodes average rewards obtained from per training



Appendix [3.3]: Scatter, 5 episodes average rewards obtained from per training



Appendix [3.5]: Scatter plot, img_mode vs. ram_mode vs. mix_mode



Appendix [3.6]: Line plot, img_mode vs. ram_mode vs. mix_mode