Problem Representation:

1.

I think the state as the hardware ram (array) would be easier for the agent to learn from, since q-learning seeks to learn a policy that maximizes the total **numerical** rewards signals. Also, compared to numerical arrays, decision making for image processing is a cumbersome process as an image's data may be large and highly redundant, it can be subject to multiple interpretations. Thus, I think it would be easier for the agent to learn from hardware ram.

2.

The neural network in Q-Learning is used to approximate the value and the policy function here, the neural nets learn to map states to values or state-action pairs to Q-values here. For the neural network class **`QLearner(nn.Module)`**, the inputs are: **`env`**, it is the environment in Reinforcement Learning, which is the agent's world in which it lives and interacts in Q-Learning; **`num_frames`**, which is the number of frames the game takes to train the agent with both random and learned policies; **`batch_size`**, which is the size of the representative of the whole data batch to calculate the gradient of loss on the subset of the whole data; **`gamma`**, which represents the discount factor that quantifies how much importance given for the future rewards and approximate their noises, and the agent here will consider its rewards tends based on the values of gamma value; **`replay_buffer`**, which is used in Q-Learning to store trajectories of experience when executing policies in the environment during the training; **`input_shape`** is a variable that starts tensor sent to the first hidden layer and also to define the input layer; and **`num_actions`** is the numbers of the actions of the agent in the training experience. The outputs: **`features`**, which represents the features are therefore explanatory variables relating to the state or state and action that needs to be sufficient to explain the value or optimal action of that state; and **`action`**, which outputs the agent's methods in the **`act`** function to interact and change its environment thereby transferring between states.

3.

These lines of codes, in this if-else statement, epsilon gets greater with the number of learning frames. Greater epsilon indicates less probability for the if condition to be true. Random **exploration** at the start, then **exploitation** at the end. **`if random.random() > epsilon,`** we choose the best action to be performed, hence maximum reward for a particular action, and then exploitation. **`else: action = random.randrange(self.env.action_space.n)`** Else, we randomly select a valid action, then exploitation at the end.

4.

The programming part in `act` function is done in **dqn.py.**

Making the Q-Learner Learn:

1.

One-state lookahead below $Loss_i(Theta_i) = (y_i-Q(s,a;Theta_i))^2$ in loss function implementation:

We get the Q(s,a;Theta) as:

`actual_q = model(state).gather(1,action.unsqueeze(-1)).squeeze(-1)`

Then we start computing y_i, and get the $max(Q(s',a';Theta))_i$:

`expected_y = target_model(next_state).detach().max(1)[0]`

Then we need to multiply each $max(Q(s',a';Theta))_i$ by gamma discount factor as:

`expected_y *= gamma` we need to check if the gamma value changed from previous

For `zero = torch.zeros(32).cuda()` here is just a tensor of zeroes on cuda.

And for condition each $max(Q(s',a';Theta))_i$*gamma,

If done is 1, which means terminated, then we place a zero at that place:

$(max(Q(s',a';Theta))_i$*gamma won't contribute.

If done is 0, which means not the terminal state, then we keep the old value. So we have:

`expected_y = torch.where(done != 0., zero, expected_y)`

Then add $reward_i$ to y_i: `expected_y = expected_y.add(reward)`

Then assign this important flag as: `expected_y.requires_grad = True`

Last, we need to use MSELoss to sum up all the $(expected\_y_i - actual\_q_i)^2$:

```
loss_fn = nn.MSELoss(reduction='sum')
loss = loss_fn(expected_y, actual_q)
```

2.

The implementation of the loss function is done in **dqn.py**

Extend the Deep Q-Learner:

1.

The implementation of the sample function is done in **dqn.py**

Learning to Play Pong:

1.

The adjustment in **run_dqn_pong.py** is done.

2.

The **run_dqn_pong.py** is modified and the memories of losses and rewards are recorded.

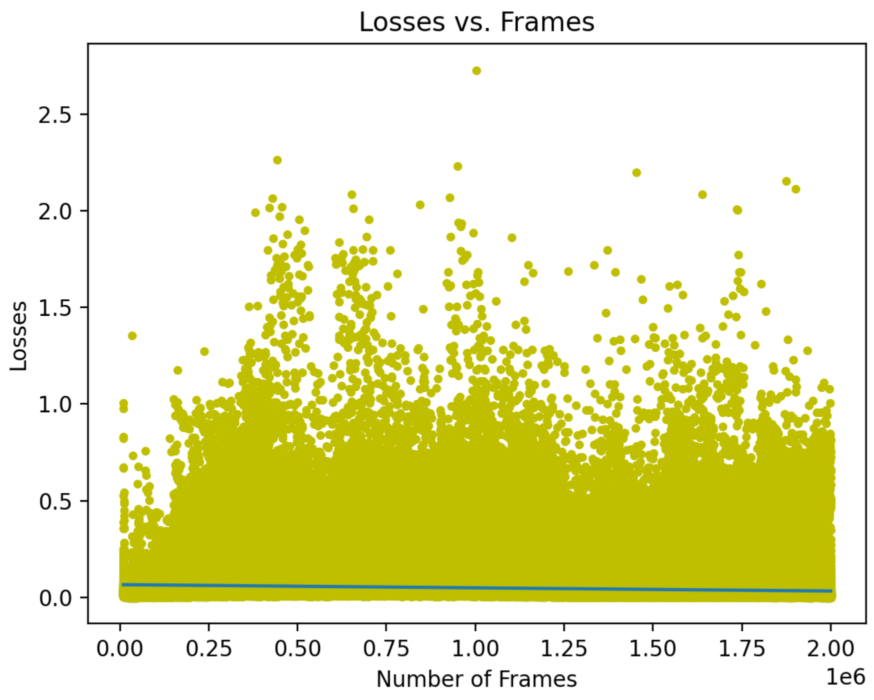
3.

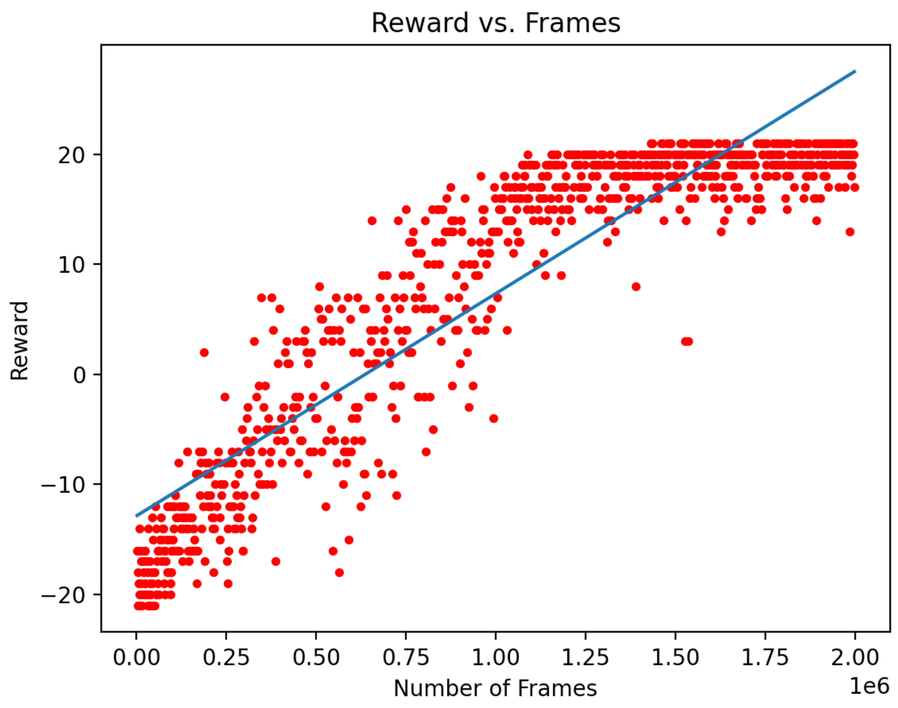
Teach the Atari pong game to play against an opponent.

It took me more than 10 hours to train the model.pth on the Google Cloud virtual machine.

4.

The plot of how loss and reward change during the training process:


Losses vs. Frames
Losses
1e6
Number of Frames


Reward vs. Frames
Reward
1e6
Number of Frames