Cartpole control environment

- It consists of a cart that can move linearly, and a rotating bar or a pole attached to it via a bearing. Another name for this system is the inverted pendulum.

- **Objective:** The control objective is to keep the pole in the vertical position by applying horizontal actions (forces) to the cart.
- **State space:** Cart position, cart velocity, pole angular position, pole angular velocity
- **Action space:** Move left or move right
- **Reward:** +1 for each time step the pole remains upright
- Episode ends if pole angle > 12 degrees or cart position > 2.4 units from the center

Environment Specifications

The **action space consists** of two actions:

- Push the cart left – denoted by 0
- Push the cart right – denoted by 1

Observation Space:

Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-∞	∞
2	Pole Angle	~ -0.418 rad (-24°)	~ 0.418 rad (24°)
3	Pole Angle Velocity	-∞	∞

All observations are assigned a uniformly random value in (-0.05, 0.05)

Q-Learning

An episode terminates under the following conditions:

The pole angle becomes greater than |12| degrees (absolute value of) or |0.2095| radians (absolute value of).

Cart position is greater than |2.4| (absolute value of).

If the number of steps in an episode is greater than 500 for version v1 of CartPole.

The reward of +1 is obtained every time a step is taken within an episode.

This is because the control objective is to keep the pole in the upright position. That is, the higher sum of rewards is obtained for longer episodes (the angle or rotation does not exceed |12| degrees), and when the cart is for a longer time period close the vertical position.

Q-Learning

- Q-values are updated iteratively based on the observed rewards and the estimated maximum future rewards.

The **formula used to update the weights of the networks** is:

- *Qn*(*s*,*a*) is the updated Q-value for state *s* and action *a* after the *n*-th iteration.
- is the Q-value for state *s* and action *a* from the previous iteration).
- *α* is the learning rate, and it is adjusted based on the number of times the state-action pair (*s*,*a*) has been visited up to the (*n*−1)-th iteration.
- *r* is the immediate reward obtained after taking action *a* in state *s*.
- *γ* is the discount factor.
- *s*′ is the next state.
- *a*′ is the action that maximizes the Q-value in the next state.

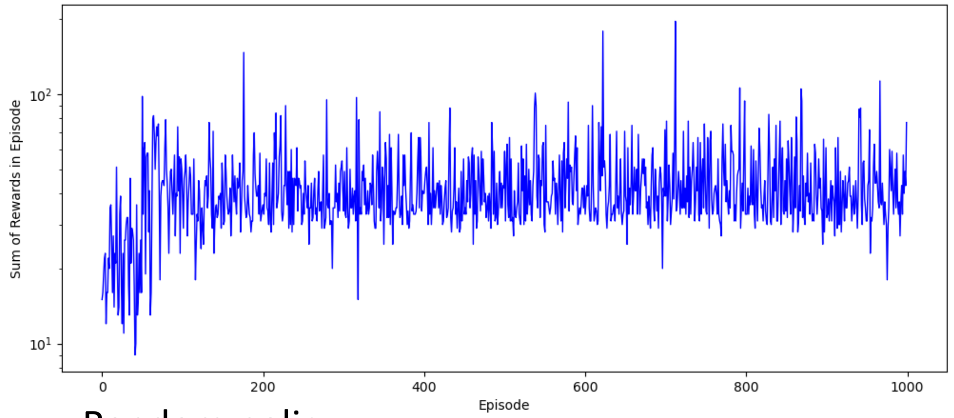
$$\hat{Q}_n(s,a) \leftarrow (1-\alpha)\hat{Q}_{n-1}(s,a) + \alpha[r + \gamma \max_{a'} \hat{Q}_{n-1}(s',a')]$$

$$\alpha = \frac{1}{1 + visits_{n-1}(s,a)}$$

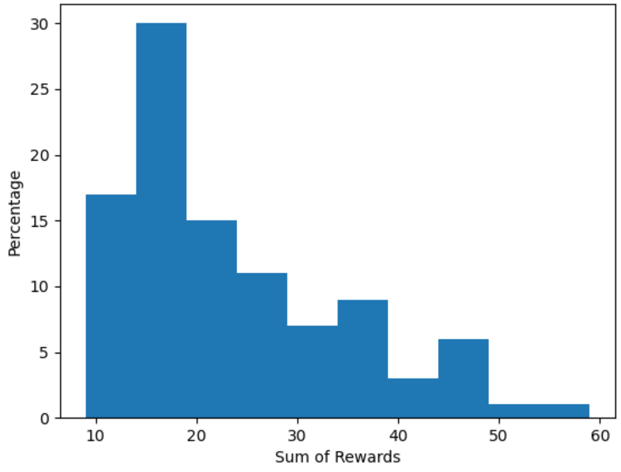
If the next state S' is the terminal state then Q(S',Q')=0 by definition, and consequently, update the action value function.

Simulation Results of Learned Policy and Random Policy

Optimal learned policy



Random policy



- We can observe that gradually the sum of rewards increases during episodes.
- Every time, obtained a different value of the sum of rewards. This is because in every episode simulation, the system is initialized from a different initial state.
- We can observe that the optimal learning policy obtains a much higher sum of rewards compared to the random learning strategy. This is clear evidence that the Q-Learning algorithm works in practice.
- A high learning rate can make the algorithm converge quickly, but it may also lead to instability or overshooting optimal values. I chose dynamic learning rate alpha by considering visit numbers of agent.

Deep Q-Learning

Instead of using q-table we used NN to approximate Q-value. We have used the TensorFlow's Keras API for defining and compiling the neural network model.

The architecture is follows:

- Input layer: 4 nodes (corresponding to the state dimension)
- Hidden layer: 128 nodes with ReLU activation
- Hidden layer: 56 nodes with ReLU activation
- Output layer: 2 nodes (corresponding to the action dimension) with linear activation

The Q-value update in the Deep Q Learning algorithm can be represented as follows

- *Q*(*s*,*a*) is the Q-value for state *s* and action *a*.
- *α* is the learning rate, and in your code, it is dynamically adjusted: *α*=1/(1+index)
- *r* is the immediate reward obtained after taking action *a* in state *s*.
- *γ* is the discount factor.
- *s*′ is the next state.
- *a*′ is the action that maximizes the Q-value in the next state.

$$\hat{Q}_n(s,a) \leftarrow (1-\alpha)\hat{Q}_{n-1}(s,a) + \alpha[r + \gamma \max_{a'} \hat{Q}_{n-1}(s',a')] \qquad \alpha = \frac{1}{1 + visits_{n-1}(s,a)}$$

Cost Function

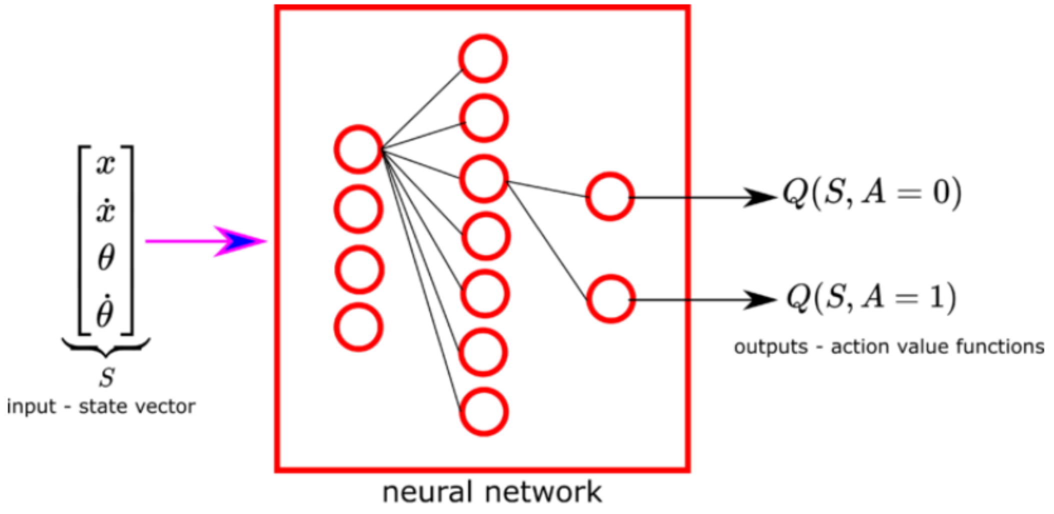
Using the Mean Squared Error (MSE) loss function as the cost function during the training of the neural network, The choice of the loss function is crucial for training the neural network to approximate the Q-function accurately.

$$\frac{1}{N}\sum_{i=1}^{N}\left(y_i - Q(S_i, A_i, \theta)\right)^2$$

- The neural network used backpropagation to adjust its weights in a way that minimizes the chosen loss function.
- The loss function is typically defined to minimize the temporal difference error (TD error), which is the difference between the predicted Q-value and the target Q-value. This is done using gradient descent.

Deep Q-Learning

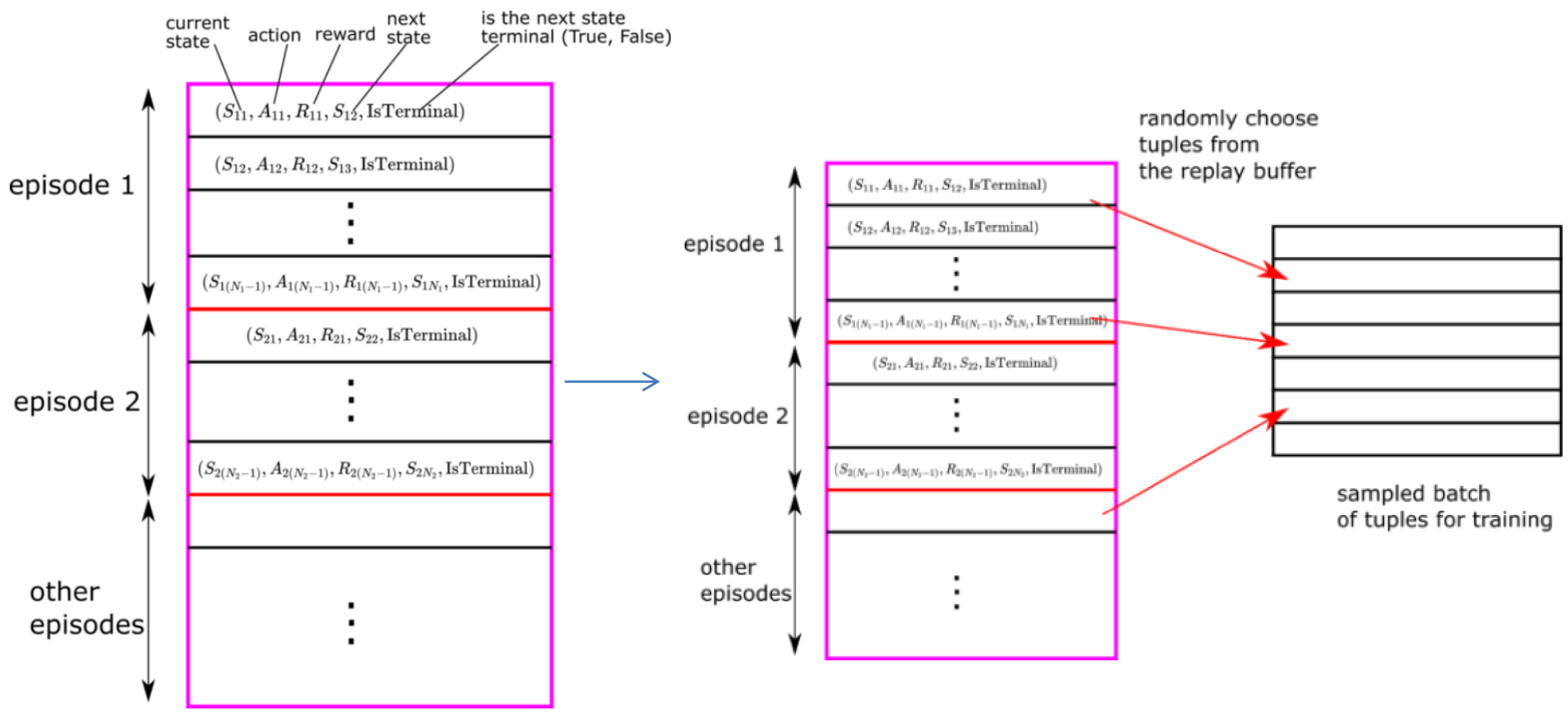
The main idea of the deep Q networks is to use a neural network as the mapping, Our goal will be to train this network. Once we train this network, we will use its prediction and the greedy approach to select the optimal control actions.



Neural network for approximating the action value functions.

Replay Memory of Deep Q-Learning

- To train the network, we need to introduce the concept of **the replay buffer.**


current state
action
reward
next state
is the next state terminal (True, False)
$(S_{11}, A_{11}, R_{11}, S_{12}, \text{IsTerminal})$
$(S_{12}, A_{12}, R_{12}, S_{13}, \text{IsTerminal})$
$(S_{1(N_1-1)}, A_{1(N_1-1)}, R_{1(N_1-1)}, S_{1N_1}, \text{IsTerminal})$
$(S_{21}, A_{21}, R_{21}, S_{22}, \text{IsTerminal})$
$(S_{2(N_2-1)}, A_{2(N_2-1)}, R_{2(N_2-1)}, S_{2N_2}, \text{IsTerminal})$
episode 1
episode 2
other episodes
randomly choose tuples from the replay buffer
sampled batch of tuples for training

Replay Memory of Deep Q-Learning

- The approximation of Q using one sample at a time is not very effective. The network managed to achieve a much better performance compared to a random agent.
- Experience replay stores the agent's experiences in memory.
- The replay buffer is essential for stabilizing and improving training. Ensure that the replay buffer size (replayBufferSize) and the batch size (batchReplayBufferSize) are reasonable for the task.
- Batches of experiences are randomly sampled from memory and are used to train the neural network. Such learning consists of two phases--gaining experience and updating the model. The size of the replay controls the number of experiences that are used for the network update. Memory is an array that stores the agent's state, reward, and action, as well as whether the action finished and the next state.

Hyperparameters of DQL algorithm with replay Single Network

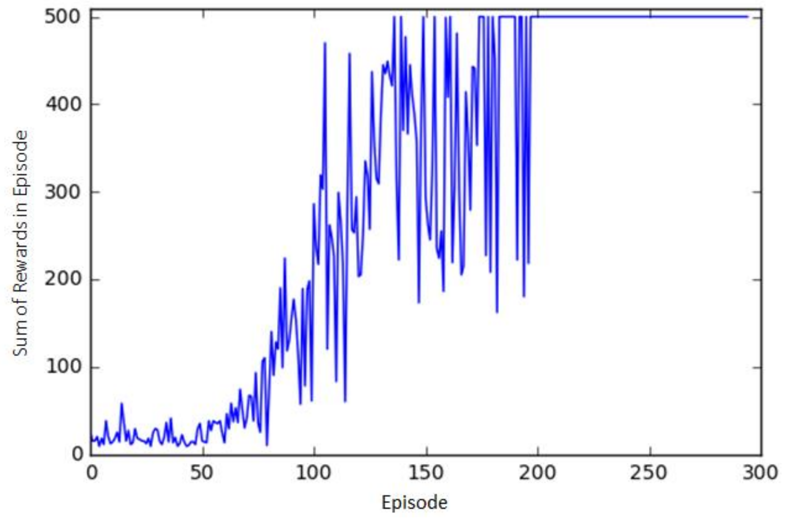
Hyperparameter	**Value**
gamma	1
epsilon	0.1 (initial)
numberEpisodes	2000
stateDimension	4
actionDimension	2
replayBufferSize	300
batchReplayBufferSize	50
learning rate	alpha = 1 / (1 + self.visits[state_index][action_index])

Hyperparameters of DQL algorithm with replay Separate Networks as Target and Online Network

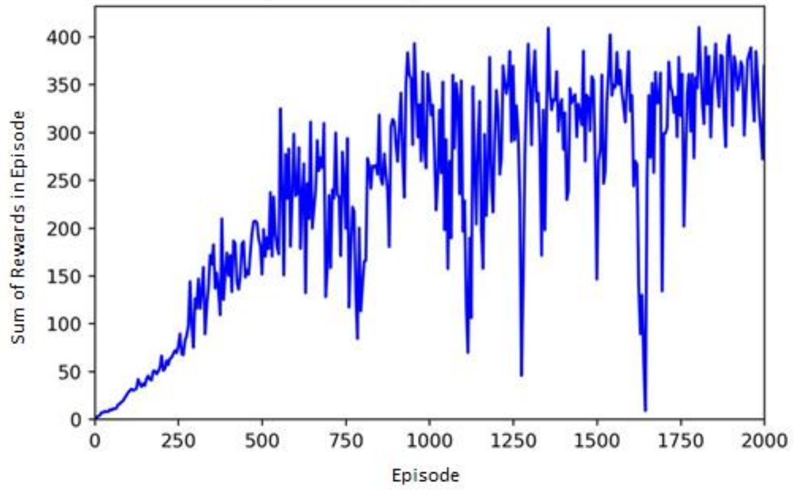
Hyperparameter	**Value**
gamma	1
epsilon	0.1 (initial)
numberEpisodes	300
stateDimension	4
actionDimension	2
replayBufferSize	300
batchReplayBufferSize	50
updateTargetNetworkPeriod	10
learning rate	alpha = 1 / (1 + self.visits[state_index][action_index])

Performance analysis of DQL

DQL with 2 Separate Target and Online Networks



DQL with Single Network



The incorporation of a replay buffer proved effective in stabilizing the learning process by providing a diverse set of experiences for the agent.

Using a separate target network helps stabilize the training process. The target network parameters are updated less frequently, providing a more stable target for the Q-learning update.

The separate target network introduces a delay in the learning process because it updates less frequently(once per 10 timestep). This delay can slow down the learning rate, but it's a trade-off for increased stability.

Maintaining two separate networks adds complexity to the implementation and increases computational requirements