Reinforcement Learning

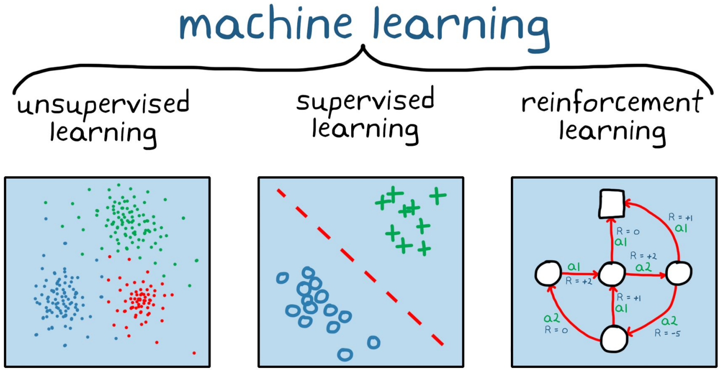
Deep Learning Project

Background on Reinforcement Learning (RL)

RL as Self Supervised Learning

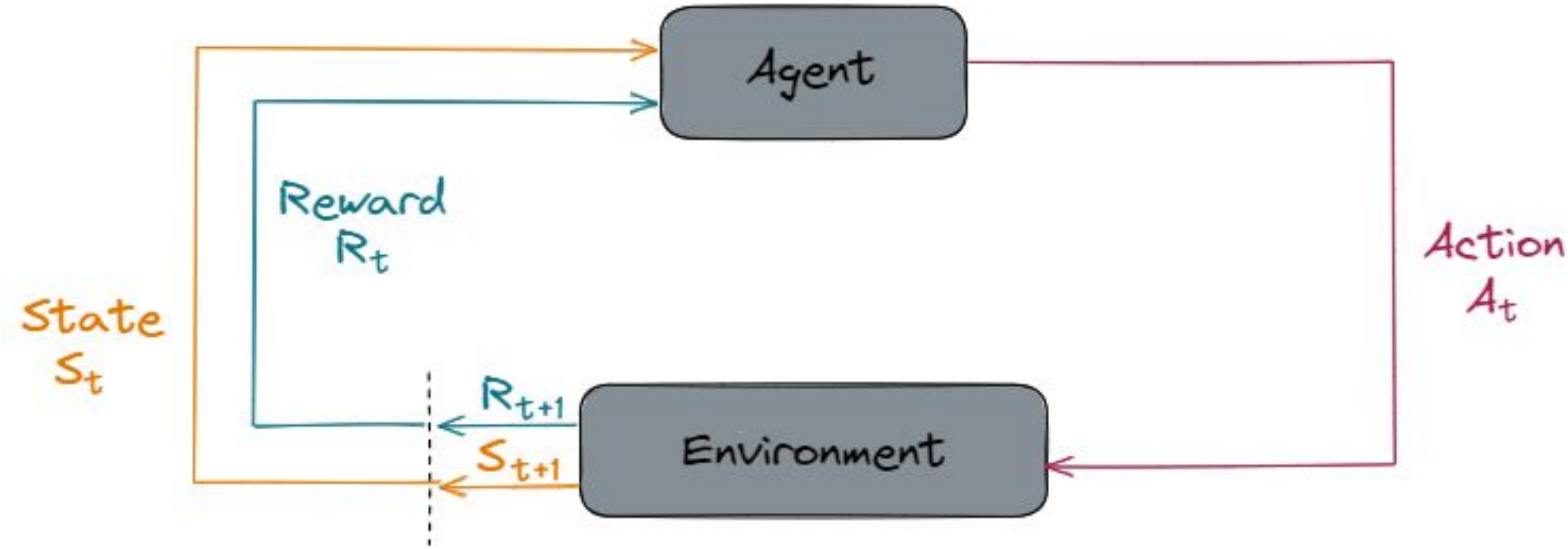
Def: Reinforcement learning (RL) is an area of machine learning that focuses on how an agent might act in an environment in order to maximize some given reward.

1. Supervised learning → Training data with labels.
2. Unsupervised learning → Only training data (no labels).
3. Self Supervised Learning (aka RL) → Generates its own data.



Markov Decision Processes (MDP)

Markov Property: The transition from one state to the next state is independent of all previous states (this is also known as the memory-less property).



GOAL: Maximize cumulative reward

Maximizing Expected Return

We can maximize cumulative rewards by taking an action to maximizing something called the expected return at each timestep.

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T,$$

Sum of future rewards

Expected return

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}.$$

Weighted sum of future rewards

Discount rate

Expected return depends only on the immediate reward, and the expected return at the next time-step (weighted by the discount factor)

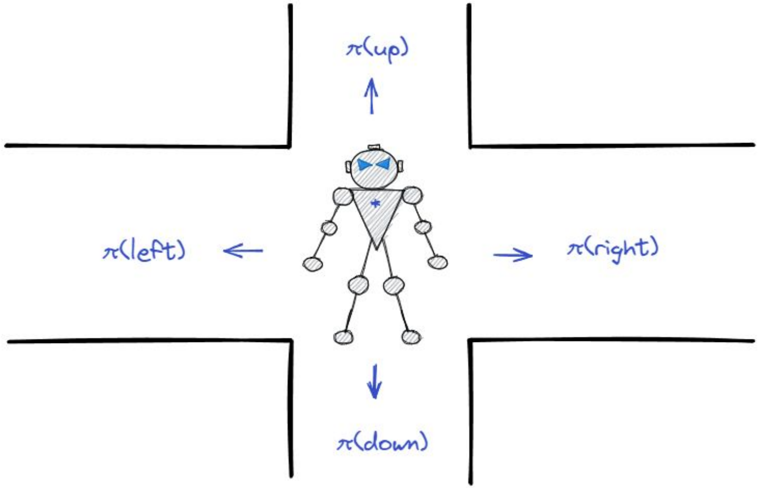
$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \\ &= R_{t+1} + \gamma \left(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots \right) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned}$$

GOAL: Maximize expected return

Policies

Def: A policy is a function that maps a given state to probabilities of selecting each possible action from that state.

$$\pi(a \mid s)$$ ← Probability of taking action *a* in state *s*



GOAL: Find the optimal policy that maximizes expected return

Q-Functions

Def: A Q-function is a function that tells us how “good” it is to take a given action from a given state while following a certain policy. The “goodness” can be measured by expected return.

Expected return

$$
\begin{aligned}
q_\pi(s, a) &= E_\pi\left[G_t \mid S_t = s, A_t = a\right] \\
&= E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right]
\end{aligned}
$$

Q-value of taking action *s* in state *a* while following policy π

We must take expectation because the environment may not be deterministic

GOAL: Find the optimal Q-function corresponding to the optimal policy

Bellman Optimality Criterion

The optimal Q-Function must be equal to the expected return at each state-action pair. Hence, it can be shown that the optimal Q-function must obey the following equation (known as the bellman equation).

$$q_*(s,a) = E\left[R_{t+1} + \gamma \max_{a'} q_*(s',a')\right]$$

Optimal q-function

Immediate reward

Expected return at the next time-step (assuming that the Q-function is optimal already)

Maximize cumulative reward

Maximize expected return

Find the optimal policy that maximizes expected return

Find the optimal Q-function corresponding to the optimal policy

Q-Learning

Q-Table

The main idea behind Q-learning is to represent the Q-function as a Q-table that contains a box for all possible state action pairs. Each entry in the Q-table will then contain its corresponding Q-value i.e the expected return of taking that action in that state.

	Action 1	**Action 2**	**Action 3**	**Action 4**
State 1	$Q(s_1,a_1)$	$Q(s_1,a_2)$	$Q(s_1,a_3)$	$Q(s_1,a_4)$
State 2	$Q(s_2,a_1)$	$Q(s_2,a_2)$	$Q(s_2,a_3)$	$Q(s_2,a_4)$
State 3	$Q(s_3,a_1)$	$Q(s_3,a_2)$	$Q(s_3,a_3)$	$Q(s_3,a_4)$
State 4	$Q(s_4,a_1)$	$Q(s_4,a_2)$	$Q(s_4,a_3)$	$Q(s_4,a_4)$
State 5	$Q(s_5,a_1)$	$Q(s_5,a_2)$	$Q(s_5,a_3)$	$Q(s_5,a_4)$
State 6	$Q(s_6,a_1)$	$Q(s_6,a_2)$	$Q(s_6,a_3)$	$Q(s_6,a_4)$

Bellman Update Equation

We want the optimal Q-table, and we know that an optimal Q-table will obey the bellman equation for all the Q-values.

$$q_*(s,a) = E\left[R_{t+1} + \gamma \max_{a'} q_*(s',a')\right]$$

However, we run into a problem here because we need an optimal Q-table to compute the optimal Q-table, so where do we start?

The solution to this is to start with a Q-table full of zeros, and iteratively update the Q-values in a way that it will eventually converge to the optimal Q-values.

From the Bellman Equation

$$q^{new}(s,a) = (1-\alpha)\underbrace{q(s,a)}_{\text{old value}} + \alpha\overbrace{\left(R_{t+1} + \gamma \max_{a'} q(s',a')\right)}^{\text{learned value}}$$

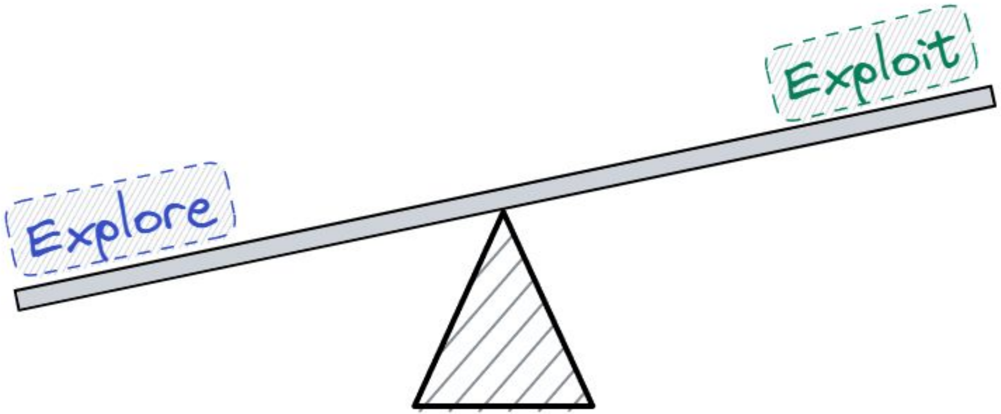
Learning rate (between 0 and 1)

From the current Q-table

Exploration vs Exploitation

Problem: How does the agent take the first action if the Q-table is full of zeros? Also how does the agent discover new strategies that can potentially be even more rewarding?

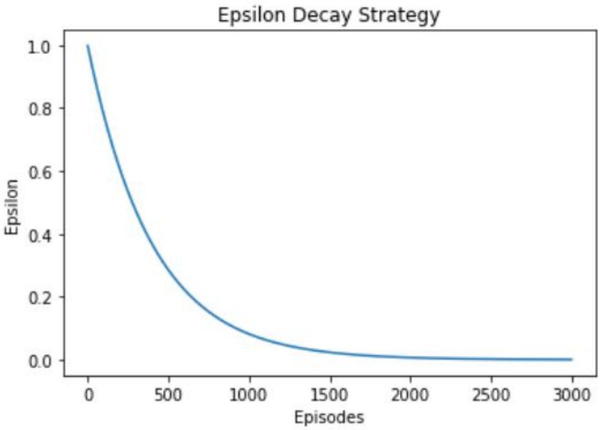
Solution: We have the agent occasionally "explore" the environment by taking a random action instead of the action that will result in the best expected return. This is known as an *epsilon greedy strategy*, where epsilon is the exploration rate (probability that the agent will explore).



Decaying Exploration and Learning Rates

As the agent progresses, we will want it to “explore” less and “exploit” more, so that it will converge to the optimal Q-table faster. We will also want it to make smaller adjustments to the Q-values so that it doesn’t overshoot the optimal Q-values.

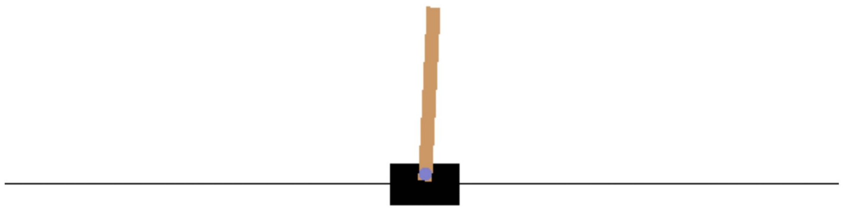
This can be implemented using decaying exploration and learning rates, where the exploration and learning rates start close to 1 and decay close to zero by the end of the training session.



The Cart-Pole Problem

Cart-Pole Problem from Open AI Gym

- The problem I will try to solve is a classic "hello world" style problem for RL.
- The goal is to balance a pole on a cart moving on a frictionless track.
- The cart can take two actions (move left or right), and it is given a reward of +1 for each timestep it stays upright.
- The game is over when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.
- The problem is considered solved if the agent can balance the pole for an average of 200 timesteps over 100 episodes.



Attempt 1: Q-Table

Discretizing State Space

The state is made up of four numbers:

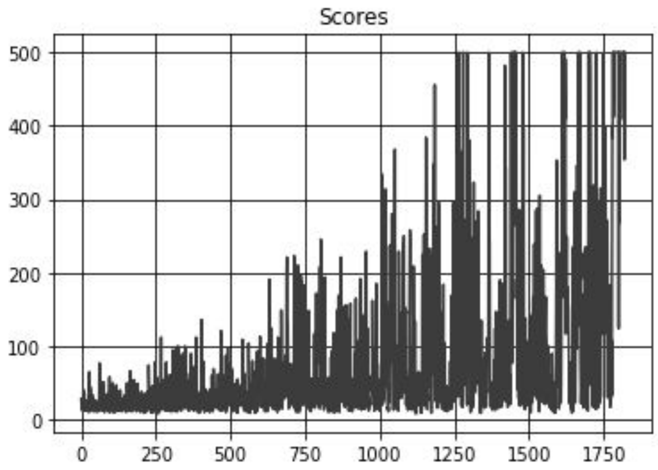
1. Position → Horizontal position of the cart
2. Velocity → Horizontal velocity of the cart
3. Angle → Tilt angle of the pole
4. Angular velocity → Tilt angular velocity of the pole

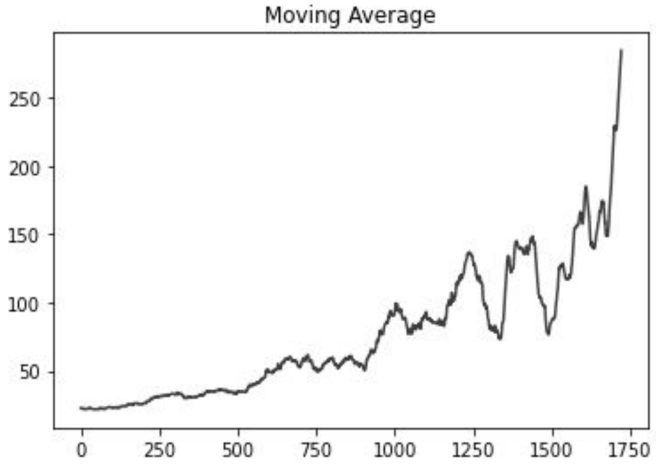
Since a Q-table needs a discrete number of states, it is necessary to “discretize” the states into bins.

I created 6 bins for the angle and 12 bins for the angular velocity. This gives us a Q-table of dimension (6,12,2). Note that this could just as easily have been a two dimensional table with dimension (72,2).

Results

- The algorithm converges to the optimal Q-table in around 1800 iterations.
- There is a high amount of variability during the training (partially due to randomness of exploration).
- During test time, the algorithm is able to consistently balance the pole for 500 timesteps (the maximum possible score).
- The average score of 100 iterations is 500 so it is considered solved!



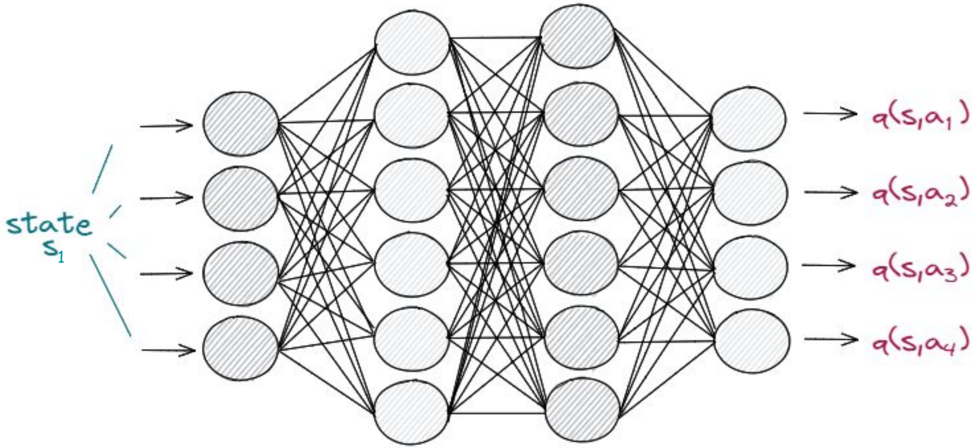


Deep Q-Networks (DQN)

Neural Networks as Functional Approximators

The Q-table is simply a function that maps a discrete number of state-action pairs to Q-values. What happens when we have a continuous state space?

We can use a neural network to act as a functional approximator, and hope that it learns the Q-values of every possible state-action pair through its parameters. This is called deep Q-learning using a deep Q-network (DQN).



Computing the Loss

The loss for one time-step can be computed by taking the difference between the current Q-value and the target Q-value for a given state-action pair.

$$q_*(s, a) - q(s, a) = loss$$

e.g MSE, MAE, Huber Loss, etc.

$$\underbrace{E\left[R_{t+1} + \gamma \max_{a'} q_*(s', a')\right]}_{\text{Target Q-value}} - \underbrace{E\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\right]}_{\text{Current Q-value}} = loss$$

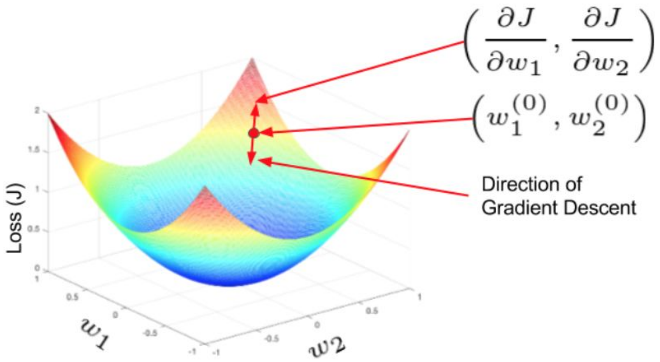
To compute the target Q-value, we use the current Q-network to compute the maximum Q-value of the next state. By performing gradient descent with this loss repeatedly, it can be shown that we will eventually converge to an optimal Q-network (assuming that the network is complex enough to begin with).

$$E\left[\underbrace{R_{t+1}}_{\text{Immediate reward (from playing the game)}} + \gamma \max_{a'} \underbrace{q_*(s', a')}_{\text{Future expected return (computed using the current Q-network)}}\right]$$

Training the DQN

Training a DQN is just like training any other neural network:

1. Compute the target value
2. Compute the loss from the difference between the target Q-value and the actual Q-value.
3. Find the gradients with backpropagation (just like with any other neural network).
4. Update the parameters according to some optimization algorithm (e.g Adam, RMSProp, etc.)



Replay Memory

Instead of computing the loss for one time-step and performing stochastic gradient descent, we can speed things up by using replay memory.

1. Repeatedly play the game and store the information from each time-step in an experience tuple containing the state, action, reward, and next state.

2. Store the experience tuples in a collection called the replay memory (aka replay buffer).

3. Randomly sample mini-batches from the replay memory and perform mini-batch gradient descent.

This method also has the added advantage of breaking the correlation between samples which further speeds up training.

Improvement: Prioritized Experience Replay

Fixed Q-Targets

Problem:

We use the Q-network to compute the target Q-values and then we optimize the Q-network based on these targets which changes the target Q-values for the next optimization step. This results in the losses becoming bigger as the algorithm “chases its own tail”.

Solution:

- Create a different network to compute the target Q-values (target network) and a different network that we want to optimize (policy network a.k.a. online/local network).
- Don’t allow gradient descent to effect the target net, but instead copy the policy net weights into the target net every tau episodes/iterations (tau is a hyperparameter).
- This fixes the target Q-values for some time allowing the policy network to be optimized to to fixed Q-values before the target Q-values change again to more accurately describe the problem.

Improvement: Target network t-Soft Update

Double Deep Q-Networks (DDQN)

Problem: Q-learning uses the maximum action value as an approximation for the target Q-value. This results in a systematic positive bias causing Q-learning to often overestimate the target Q-values. This is especially true at the start when the weights of the Q-network are essentially random.

Solution: (This is the official updated approach by Hasselt et al. in 2015.)

- Use two networks, one to select the action and another to compute the maximum expected Q-value for that given action. This has the result of sometimes underestimating and sometimes overestimating the target Q-value, removing the systematic bias.

- Since we already have two Q-networks (the target network and the policy network), we can simply use the policy network to select the action, and then use the target network to evaluate the Q-value at that action.

Target Q-value according to the target net when taking action according to the policy net

$$Y_t^{\text{DoubleDQN}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q(S_{t+1}, a; \boldsymbol{\theta}_t), \boldsymbol{\theta}_t^-)$$

Target Q-values

Immediate Reward

Best action according to the policy net

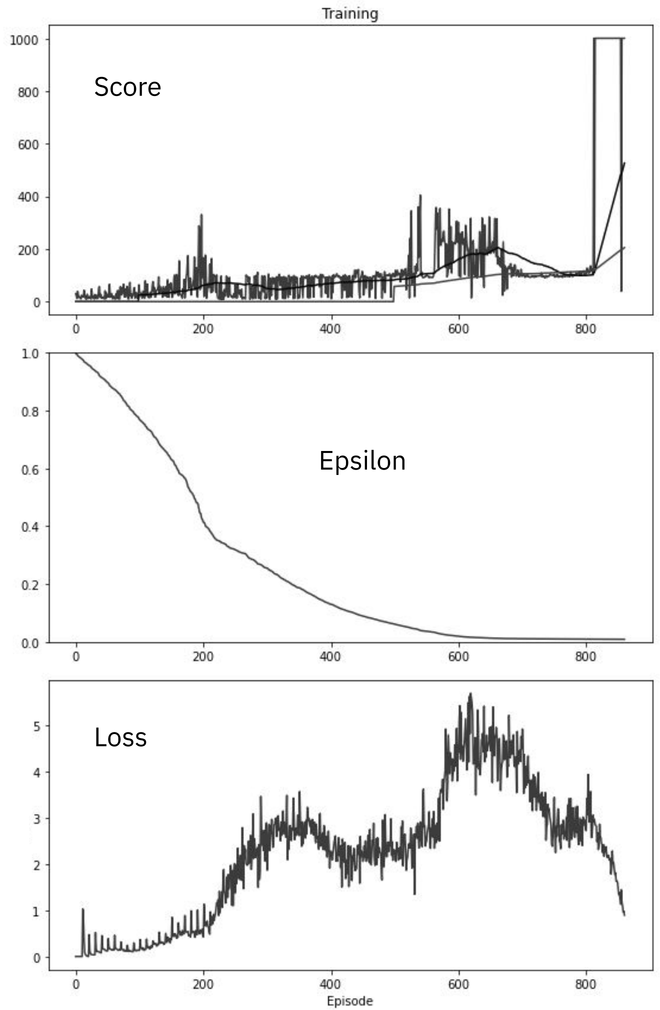
Attempt 2: DQN and DDQN

States and Hyperparameters

```
State = tensor([-0.0021, -0.5809,  0.0293,  0.8910], device='cuda:0') # Example of state

batch_size = 256

gamma = 0.999 # Discount rate in bellman equation

eps_start, eps_end, eps_decay = 1, 0.01, 0.0001 # Parameters for epsilon-greedy strategy

target_update = 10 # Rate at which we update the target net

memory_size = 100000 # Size of replay memory

lr = 0.001 # Learning rate for the optimizer

num_episodes = 2000 # Maximum number of training episodes
```

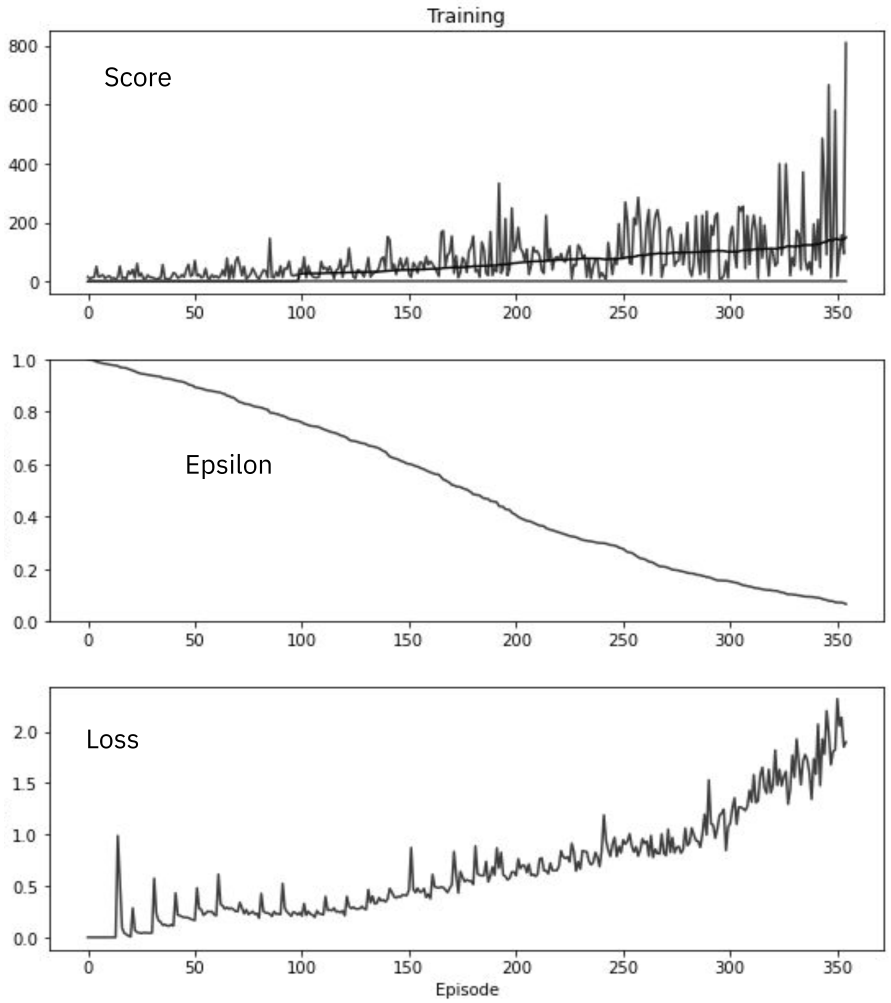
DQN Results

- The algorithm converges to the optimal Q-network in only 860 episodes.
- Once the the network figures it out and the learning rate decays enough, the score shoots up!
- During test time, the algorithm is able to consistently balance the pole for 1000 timesteps (the maximum possible score).
- The average score of 100 iterations is 981.22 so it is considered solved!



DDQN Results

- The algorithm converges to the optimal Q-network in only 350 episodes!
- During test time, the algorithm is able to consistently balance the pole for over 200 timesteps (what it was trained to do).
- The average score of 100 iterations is 202.88 so it is considered solved!

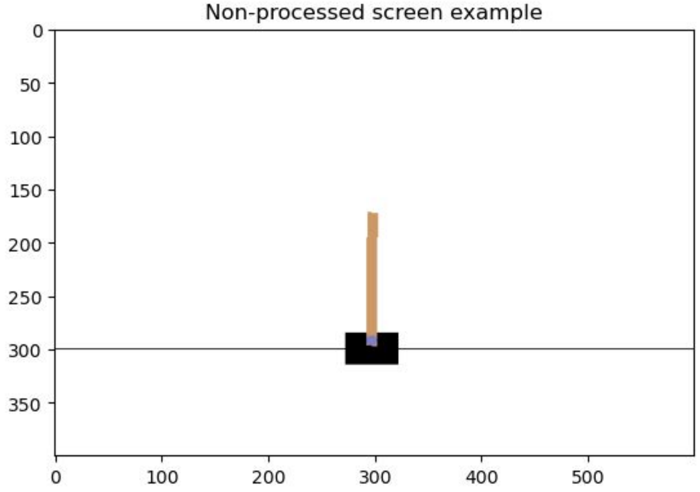


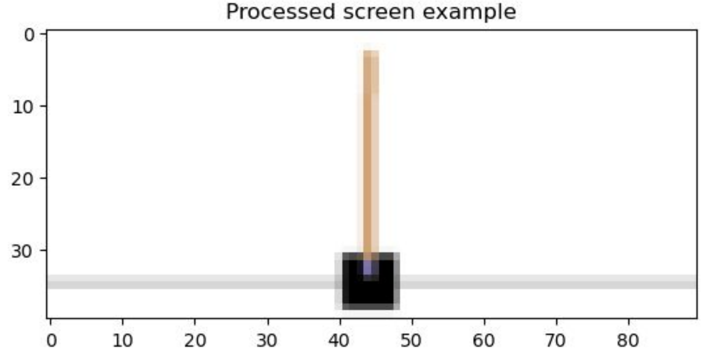
Cart-Pole with Pixel Values

Exponentially Harder!

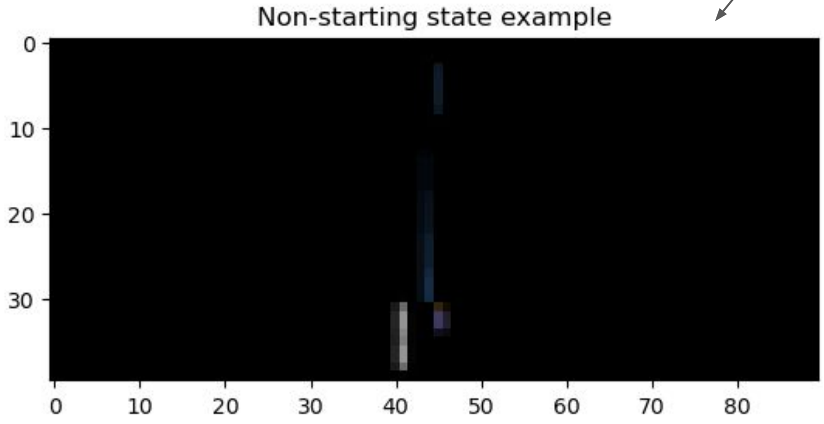
- Instead of representing the state as a vector of four numbers, we can represent the state as an image of the cartpole of dimension (40,90).

- To capture the velocity of the cart-pole, we can represent the state as the difference between the previous state and the current state.



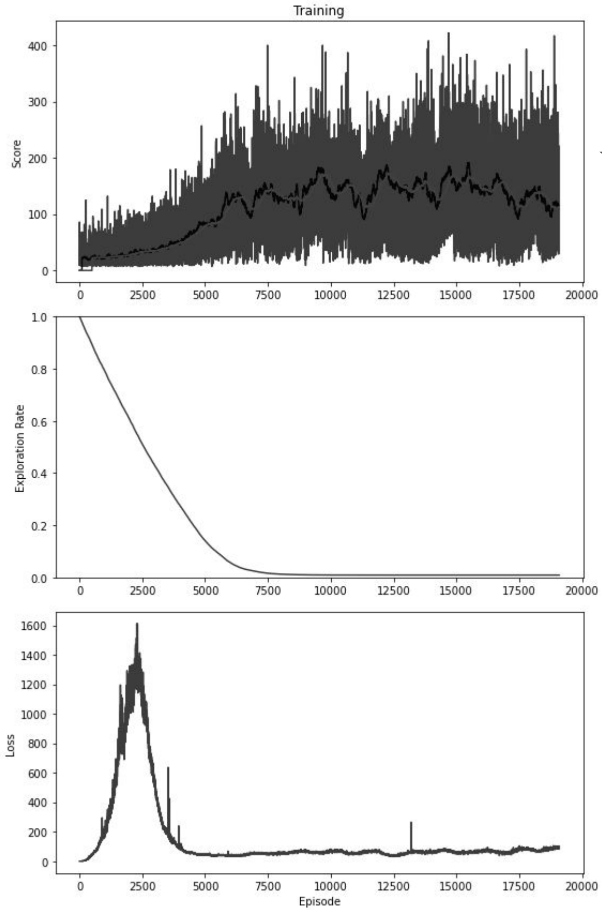


Mostly black because it's the difference between the previous screen and the current screen



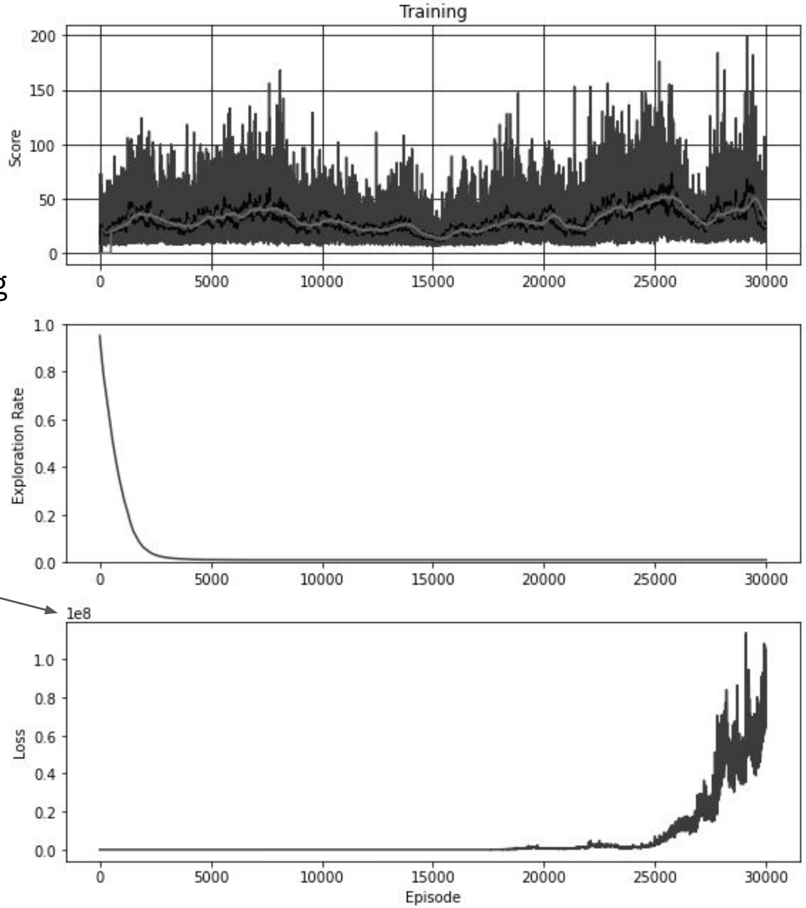
Attempt 3: Cart-Pole with Pixel Values

Attempts



Using simple MLP and training for a long time (it seems to stop learning after a certain point)

Using more complicated ConvNet model (the loss seems to explode because *tau* is too small)



Going Forward

- Increase *tau* and try training for even longer
- Try prioritized experience replay
- Change the architecture of the neural network
- Try a different loss function (softmax?)
- Try a whole different RL algorithm (policy networks?)