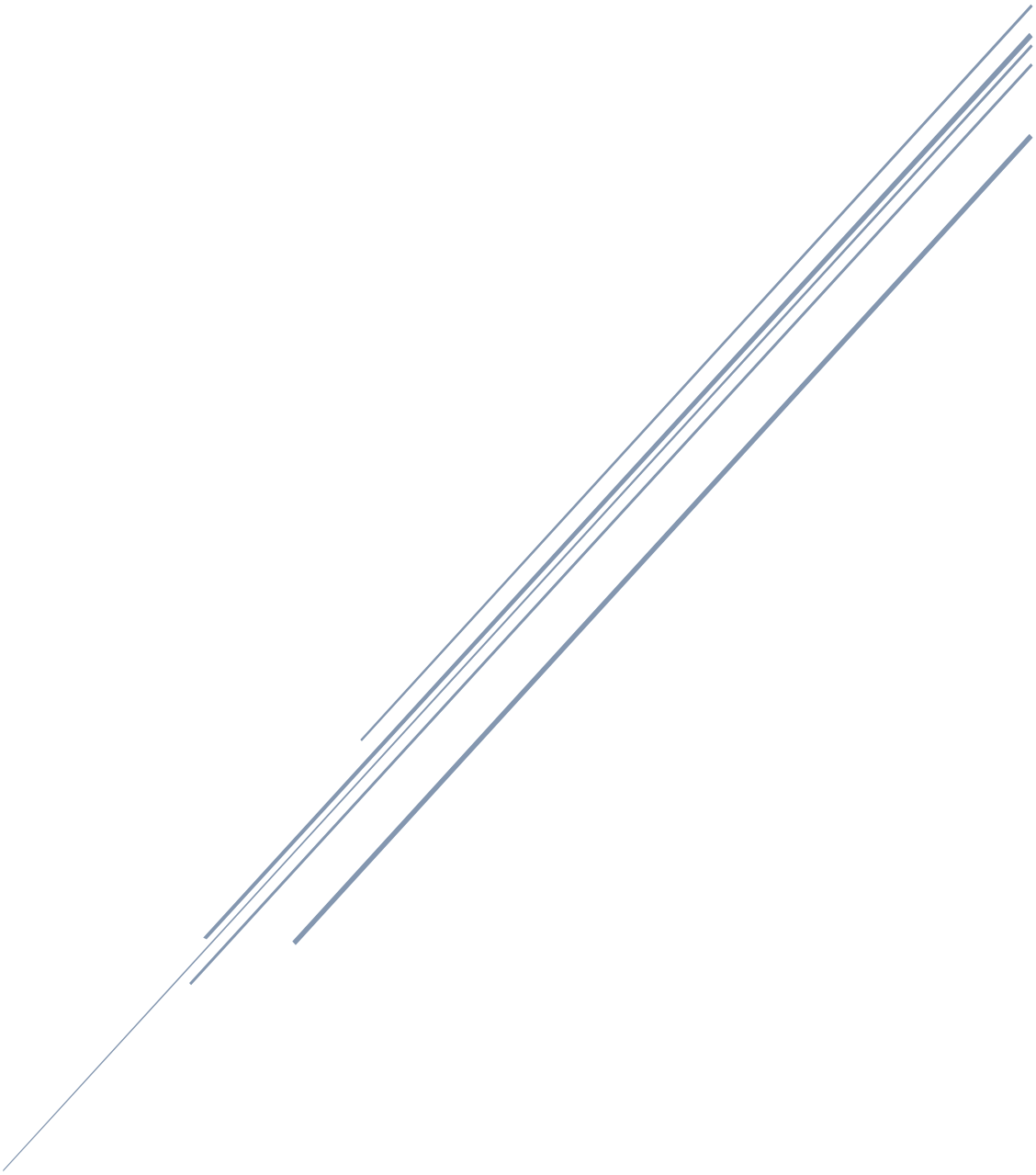
INVERSE PENDULUM REINFORCEMENT LEARNING

Final Project- Deep Learning 097200



Shir Kozlovsky 204321228
Daniel Novikov 204362420

1. abstract

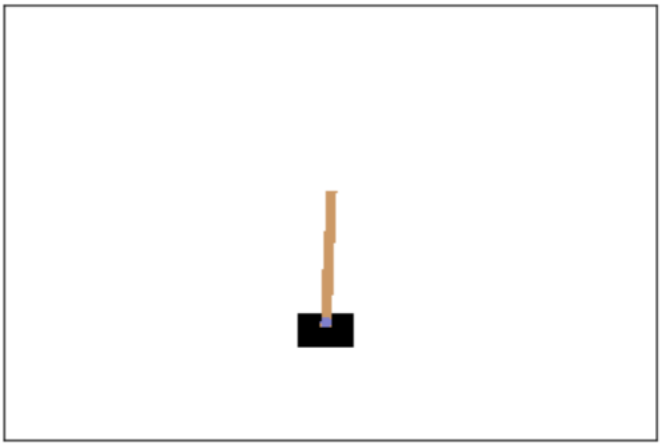
in this project, we will compare different methods of deep reinforcement learning in the CartPole environment. At first, we will go through the basic equations and background. Then we will explain the method we used – a random one as a baseline for this project, deep Q-Learning, deep Q-Learning with memory replay and double Deep Q-Learning. We will compare each method and we will see which one gives the best performance.

Contents

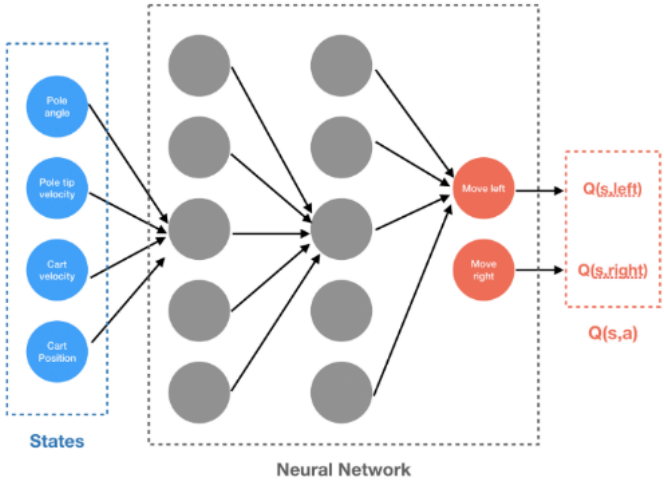
2. introduction and background

2.1 Environment

CartPole, also known as an inverted pendulum, is a game in which you try to balance the pole as long as possible. It is assumed that at the tip of the pole, there is an object which makes it unstable and very likely to fall over. The goal of this task is to move the cart left and right so that the pole can stand (within a certain angle) if possible.



The state space is represented by four values: cart position, cart velocity, pole angle, and the velocity of the tip of the pole. The action space consists of two actions: moving left or moving right. A reward of $+1$ is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.



2.2 Q-Learning

Reinforcement learning is a machine learning method that is based on rewards received from the environment rather than examples. The agent and the environment interact continuously, as shown in figure 2. The agent decides which actions to perform given the state of the environment and its policy. The actions of the agent change the state of the environment and result in an appropriate reward. The goal of the agent is to learn a policy that maximizes the reward it receives over a long period.

Q-value, $Q(s,a)$, is the expected reward for each state-action pair.

the potential future reward of that state-action pair describes as (Bellman equation):

$$Q(s,a) = r + \gamma \max_{a'} Q(s',a') \quad (1)$$

Where $\gamma \in [0,1]$ is a discount factor. $Q(s,a)$ is the expected reward, if we are in the state (s) and making (a) action. After getting a reward (r) by making an action (a), we will reach another state (s'). Then, we just look up in our Q table and find the best action to take at state (s'). So, the idea here is we don't need to consider the entire future actions, but only the one at the next time-step.

The basic update rule for $Q(s,a)$ is given by:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right] \quad (2)$$

Where α is the learning rate. This rule base on the difference between what we think we know about $Q(s,a)$ and what the current episode tells us about $Q(s,a)$.

Is our case, the policy that maximizes our rewards when want to take any action in a given state:

$$\pi^*(S) = argmax_a Q^*(s,a) \quad (3)$$

Because we don't have access to Q^*, we will create one and train it to resemble Q^*.
Every Q function for some policy obeys the Bellman equation, so the training update rule:

$$Q^{\pi}(s,a) = r + \gamma Q^{\pi}\left(s', \pi(s')\right) \quad (4)$$

The difference between the two sides of the equality is known as the temporal difference error:

$$\delta = Q(s,a) - (r + \gamma \max_{a} Q(s^{'},a)) \quad (5)$$

To minimize this error, we will use the Huber loss. The Huber loss acts like the mean squared error when the error is small, but like the mean absolute error when the error is large - this makes it more robust to outliers when the estimates of Q are very noisy. We calculate this over a batch of transitions, B, sampled from the replay memory:

$$\mathcal{L} = \frac{1}{|B|} \sum_{s,a,s',r \in B} \mathcal{L}(\delta) \tag{6}$$

Where :

$$\mathcal{L}(\delta) = \begin{cases} \frac{1}{2}\delta^2 & for\ |\delta| \leq 1 \\ |\delta| - \frac{1}{2} & o.w \end{cases} \tag{7}$$

2.1 Deep Q-Learning (DQN)

Our environment is deterministic, so all equations presented here are also formulated deterministically for the sake of simplicity.

Our aim will be to train a policy that tries to maximize the discounted, cumulative reward:

$$R_{t_0} = \sum_{t=t_0}^{\infty} \gamma^{t-t_0} r_t \tag{8}$$

Where $\gamma \in [0,1]$ is a discount factor.

To cope with this problem, we need something to approximate a function that takes in a state-action pair (s, a) and returns the expected reward for that pair. That is when deep learning comes in. It is renowned for approximating a function just from the training data. In deep Q-learning, we use a neural network to approximate the Q-value function. The state is given as the input and the Q-value of all possible actions is generated as the output. the steps involved in reinforcement learning using deep Q-learning networks (DQNs) are:

- All the experience is stored by the user in memory
- The next action is determined by the maximum output of the Q-network
- The loss function here is the mean squared error of the predicted Q-value and the target Q-value – Q*. This is basically a regression problem. However, we do not know the target or actual value here as we are dealing with a reinforcement learning problem.[1]

2.2 Replay Memory

One of the main problems with DQN is stability. When we are feeding observations into our network, our network generalizes over the past experiences.

Replay memory helps us to know what actions to take given an observation. That is why the replay buffer helps to stabilize the training.[2]

2.3 Double Deep Q Learning

The solution involves using two separate Q-value estimators (Q, Q'), each of which is used to update the other. Using these independent estimators, we can unbiased Q-value estimates of the actions selected using the opposite estimator. We can thus avoid maximization bias by disentangling our updates from biased estimates.

The update procedure of basic Q-learning is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(R_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \quad (9)$$

Where α is the learning rate.

For double Q-Learning:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(R_{t+1} + \gamma \max_a \boldsymbol{Q'}(\boldsymbol{s_{t+1}}, \boldsymbol{a'}) - Q(s_t, a_t) \quad (10)$$

$$a' = \max_a Q(s_{t+1}, a)$$

$$q_{estimated} = Q'(s_{t+1}, a')$$

Q function is for selecting the best action *a* with maximum Q-value of next state, and Q' function is for calculating expected Q-value by using the action a selected above.[3]

3. Code & Parameters

As we said before the number of states is 4 (cart position, cart velocity, pole angle and the velocity of the tip of the pole), and the number of actions is 2 (left, right).

We choose 100 episodes, 50 hidden nodes in DQN and learning rate of 0.001

3.1. basic search

this part we are the baseline for this project.

3.2. Q-Learning

This is the basic class for all the methods. It contains the environment of the CartPole, the number of games that we want to play (episodes).

This function gets:

- env - the Open Ai Gym environment of CartPole
- model – the model that we want to run
- gamma - $\gamma = 0.9$
- epsilon - represents the proportion of random actions relative to actions that are informed by existing "knowledge" that the agent accumulates during the episode. Before playing the game, the agent doesn't have any experience, so it is common to set epsilon to higher values and then gradually decrease its value.
- eps_decay - the speed at which the epsilon decreases as the agent learns.
- replay – if we want to use replay memory or not.
- replay_size – the memory size that we want.
- double – if we want to use double DQN or not.
- title – the title of the graph
- n_update - parameter specifies the interval, after which the target network should be updated.

3.3. Deep Q-Learning (DQN)

The DQN class implementation a neural network implemented in PyTorch that has two main methods, predict and update. The network takes the agent's state as an input and returns the Q values for each of the actions. The maximum Q value is selected by the agent to perform the next action.

This function gets:

- state_dim – the state dimension
- action_dim - the action dimension
- hidden_dim – the dimension of the hidden layers
- alpha – the learning rate (0.001)

the architecture of this NN is a linear layer (state_dim, hidden_dim), then LeakyReLU, then another linear layer (hidden_dim, $2 \cdot$ hidden_dim), LEakyRelu, linear ($2 \cdot$ hidden_dim, action_dim)

we use Adam of optimization.

3.4. Deep Q-Learning with Replay memory

Experience replay stores the agent's experiences in memory. 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

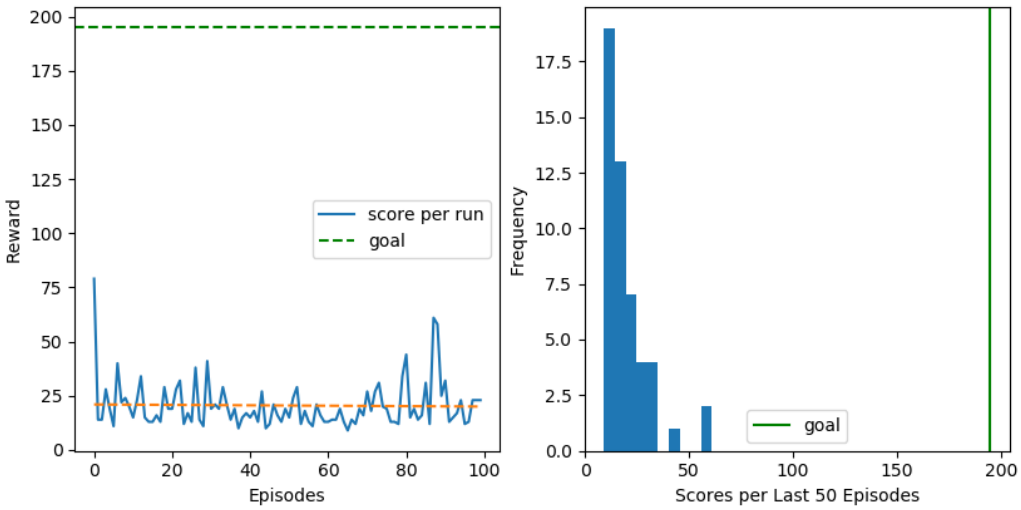
agent's state, reward, and action, as well as whether the action finished the game and the next state.

3.5. Double Q-Learning

The Q values will be taken from this new network, which is meant to reflect the state of the main DQN. However, it doesn't have identical weights because it's only updated after a certain number of episodes.

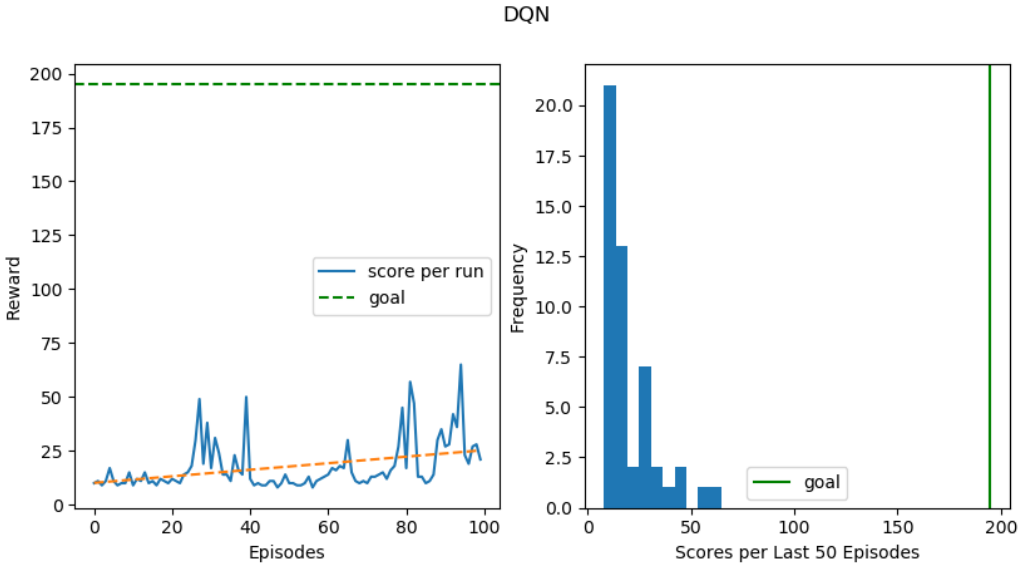
4. Results

4.1. basic search



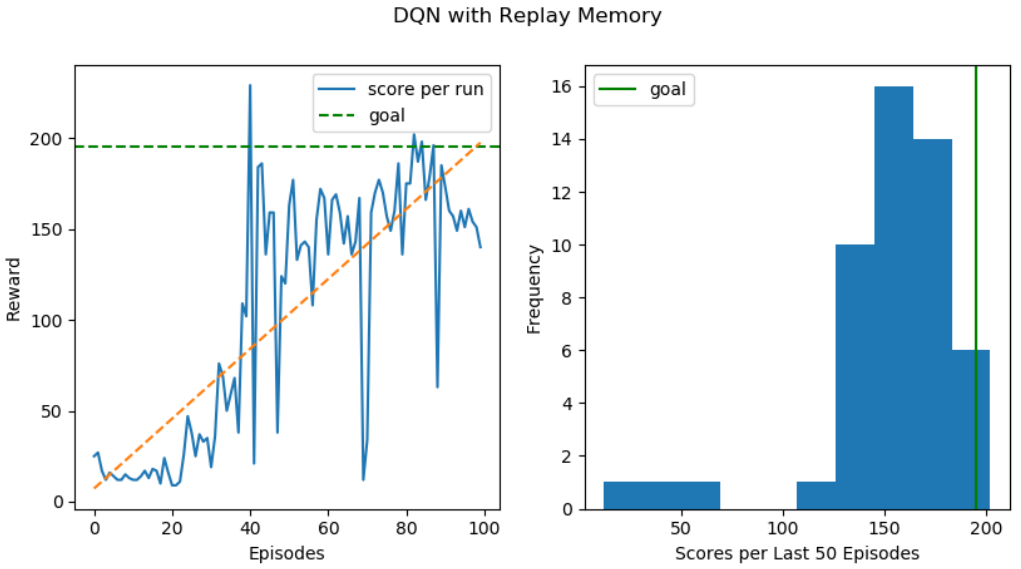
As we can see, because this strategy is random, its impossible to solve this problem. The agent isn't learning from his experience. The average performance is low then 10 steps.

4.2. Deep Q-Learning



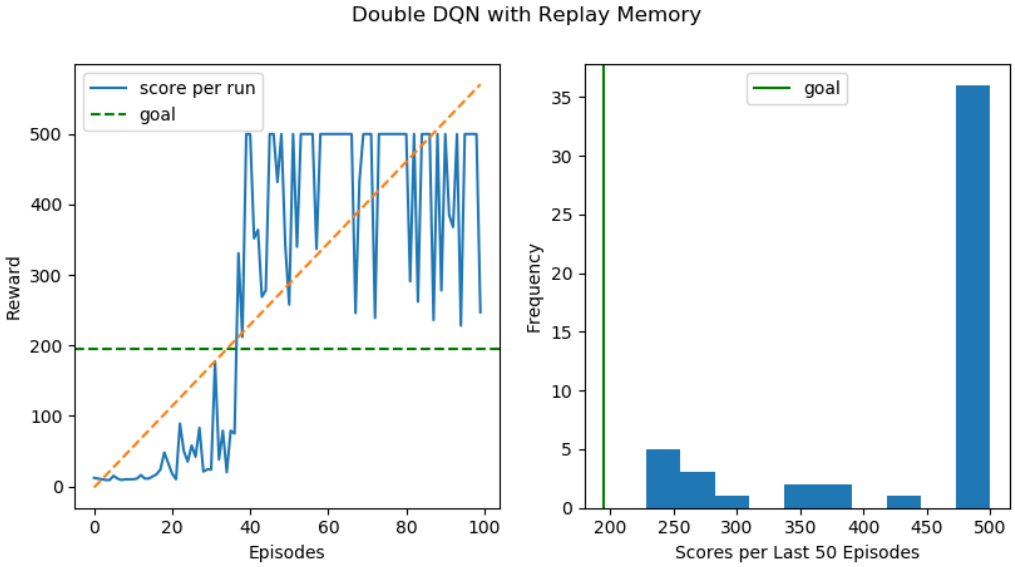
As we can see, the agent's performance improved. We get almost 100 steps. we can see that the performance increases over time and the trend line is positive. But still, the agent isn't capable to get to the goal.

4.3. Deep Q-Learning with Replay memory



The neural network with the replay is more robust compared to its counterpart that only remembers the last action. We can see that the agent's performance significantly improves. the performance increases over time and the trend line is positive. The agent reaches the goal, but it looks unstable.

4.4. Double Q Learning with Replay Memory



Here we can see that the agent achieves the goal only after 40 episodes, it is stable and robust. The agent gets a very high reword of 500.

5. Conclusion

As we improve our methods, we saw better results. From the methods that we implement, the method that gave the most robust, stable and with the highest reword is the double Q-Learning with the replay memory. And the most inefficient method we check in the random one. Of course, we can improve our result if we add more function or try another method (for example soft update).

6. reference

1. Wang Z, Schaul T, Hessel M, Van Hasselt H, Lanctot M, De Frcitas N. Dueling Network Architectures for Deep Reinforcement Learning. *33rd Int Conf Mach Learn ICML 2016*. 2016;4(9):2939-2947.

2. Liu R, Zou J. The Effects of Memory Replay in Reinforcement Learning. *2018 56th Annu Allert Conf Commun Control Comput Allert 2018*. 2019:478-485. doi:10.1109/ALLERTON.2018.8636075

3. Hasselt H Van, Guez A, Silver D. Double DQN.pdf. *Proc 30th AAAI Conf Artif Intell*. 2016:2094-2100.