FINAL PROJECT REPORT

ACTIVE CONTROL OF INVERTED PENDULUM USING Q LEARNING AND DEEP-Q LEARNING

ENPM 808F- ROBOT LEARNING

Submitted to:

Prof. Donald Sofge

By

ANIRUDH TOPIWALA

UID:115192386

TABLE OF CONTENT

1. Abstract

The inverted pendulum is a classical control problem, which involves developing a system to balance a pendulum. For visualization purposes, this is similar to trying to balance a broomstick on a finger. Here, I have considered the 2D case, where the pendulum can fall either on the left or the right side, which can be balanced by a counter force applied by the cart moving on a rail. The classic control problem of the inverted pendulum is interesting because it can be solved using a wide variety of ways. Here we see how **Q learning and Deep Q learning** can be used to make the system learn to balance the pendulum.
After implementing both the techniques, we compare them and discuss the limitations of each method over the other. From the results obtained, I can infer that the number of episodes taken in Deep Q learning is far more than Q learning. This is because training the network to fit a function takes more time and data as compared to just storing the values in tabulated form. The advantage of Deep Q learning, is that once trained it can maintain the pendulum in the upright position for a large amount of time.

2. Introduction

The task of balancing an inverted pendulum (also known as the pole-balancing problem) was originally used to demonstrate a series of conventional control techniques. Roberge was one of the first to show that we can develop effective controls despite the presence of nonlinearities and the inherently unstable nature of the system (Roberge, 1960) [1] . The problem has since been used to demonstrate the effectiveness of reinforcement learning techniques ranging from genetic algorithms to neural networks, and there has been an increased focus on finding effective algorithms for this task with little or no prior systems information. The problem becomes significantly more difficult under these conditions because the algorithm now needs to develop a policy while learning about the underlying dynamics of the system.

The goal of the inverted pendulum task is to balance a pendulum on a cart while ensuring that the cart remains within certain bounds on a track. The Cart-Pole world consists of a cart that moves along the horizontal axis and a pole that is anchored on the cart. At every time step, you can observe its position (*x*), velocity (*x_dot*), angle (*theta*), and angular velocity (*theta_dot*). These are the observable states of this world. At any state, the cart only has two possible actions: *move to the left* or *move to the right*. In other words, the state-space of the Cart-Pole has four dimensions of continuous values and the action-space has one dimension of two discrete values. An illustration of the system is given in Figure 1.

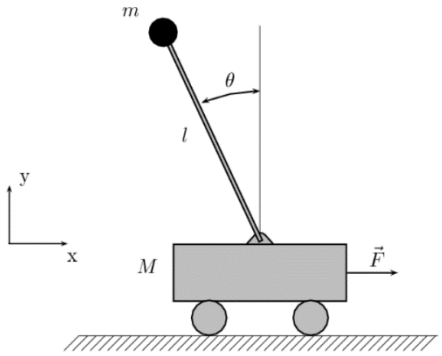


Fig 1: Inverted pendulum problem

For a practical case, the states (x, x`, θ, θ`) can be obtained by the control equations. These control equations can be formulated by calculating the lagrangian of the system. For the inverted pendulum the control equations are as follows:

$$\ddot{x}(M + m\sin(\theta)^2) = F + mg\sin(\theta)\cos(\theta) - ml\dot{\theta}^2\sin(\theta)$$

$$l\ddot{\theta} = \frac{\cos\theta}{M + m\sin(\theta)^2}(F - ml\dot{\theta}^2\sin(\theta) + mg\sin(\theta)\cos(\theta)) + g\sin(\theta)$$

I have used OpenAI Gym to simulate my environment. This helps in continuously getting feedback information of the states. It also makes simulating easier as the environment is already setup and we only need to work on generating the optimal policy, which is the crux of the problem. In the following sections we will see how Q learning and DQN is used to solve the problem.

3. Q Learning

Q-Learning is model-free Reinforcement Learning technique. It can be used to find an optimal action-selection policy for any given Markov Decision Process (MDP), a process comprising of information states which contains all useful information from the history. The output of Q-Learning technique is a Q-value table which represents the expected long term reward of the algorithm assuming that it takes a perfect sequence of actions from a specific state. But, what are actions and states? [4]

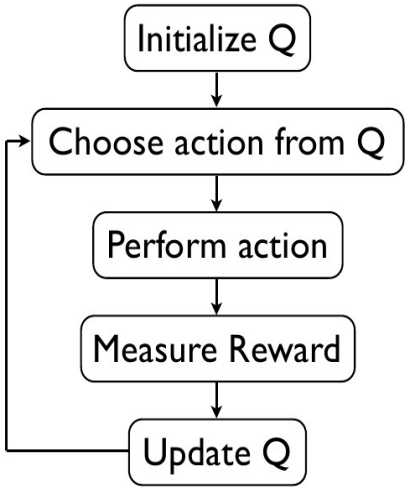
States are the various configurations that the robot takes. It can be the position, velocity or acceleration of the agent as a whole or of some component of the agent. Here the states are the position and velocity of cart and the angular displacement and angular velocity of pendulum. The other term of interest is actions. It represent all the possible movements that the agent can make to achieve a desired final state. An agent can move the cart by performing a series of actions of 0 or 1 to the cart, pushing it left or right. Let's see some of the terminology now:

- Reward Function defines the reinforcement that has to be provided to the agent for a particular state.
- Discount factor is the factor that diminishes/enhances the effect of future rewards on the Q-value of the current state-action pair.
- Learning rate, as the name suggests, is the rate of learning the various Q-values.

3.1 General Algorithm

Let's see a brief step by step procedure for Q learning:

1. Initialize the Q-table containing the state-action pairs to an initial value of zero.
2. Define the learning rate, discount factor and a reward function.
3. Select a random initial state and chose any of the action to get the next state.
4. Get the max Q value for the state and update the Q table as well as, set the state equal to next state.
5. Run multiple such episodes sets the Q-values such that when the Q-table is used for testing, the maximum Q-value for each state corresponds to the most optimum action possible at that state.



The important question here is how to decide what actions should be taken? This is very crucial as it is a measure between the exploitation verses the exploration. We need We must explore all possible solutions to get the most optimum result after training, but we do not want to explore too much. We have a specific goal to achieve, making exploitation also necessary. The equation shown below is the main equation used to calculate the Q table. [5]

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

3.2 Software Used

I have implemented the simulation using OpenAi Gym. The cart pole v1 was used to simulate the cart pole problem. Gym makes interacting with the game environment simple. With predefined functions as:
next_state, reward, done, info = env.step(action), we can easily perform the simulation.

As discussed above, action can be either 0 or 1. Env() will return the Boolean value, done, indicating whether the game is complete or not. The state information paired with action and next_state and reward we have all the information needed for training the agent.

3.3 Algorithm for Inverted Pendulum problem

Lets now go through the algorithm for the inverted pendulum:

Initialize:

1. Importing different libraries like numpy, math, gym, etc.
2. Initialize cart pole environment provided by OpenAI.
3. Set Q table to zero.
4. Set discount factor = 0.99 (this is because the world is not changing)
5. The learning rate and the exploration rate are decreased logarithmically.
   ```
   return max(MIN_EXPLORE_RATE, min(1, 1.0 - math.log10((t+1)/25)))
   return max(MIN_LEARNING_RATE, min(0.5, 1.0 - math.log10((t+1)/25)))
   ```
   where MIN_EXLORE_RATE=0.01 AND MIN_LEARNING_RATE=0.1
   A more detailed variation of how the change in learning rate is shown in coming sections.
6. The max number of episodes are 1200.
7. Initialize the number of state and actions for the environment.
8. The problem is solved if the pendulum can stay vertical for 150 time steps a total of 100 times. That is when the streak is 100, the problem solves.

Function simulate:

1. Get the logarithmic learning and explore rate based upon the time step ‘t’.
2. For random state get action.
3. Get max Q and update table
4. Set state equal to next state.
5. Repeat from 1.

Function Select action:

1. First gives out a random action
2. For the next iterations gives the action for state defined in qtable.

Function get_explore_rate

1. Returns the logarithmic decaying exploration rate. Starts from 1 and decays till 0.01. That is, it explores entirely in the beginning and exploits entirely towards the end.

Function get_learning_rate

1. Returns the logarithmic decaying learning rate. Starts from 0.5 and decays till 0.01.

Function State_to_bucket

1. This is used to divide the continuous states into discrete sates.
2. More the number of states better is the Q table.
3. The states are divided into buckets depending upon the bounds set on the buckets.

3.4 Different Optimizations

- **Variation of learning rate**: the learning rate was kept constant in the beginning at 0.1. But the problem did not converge. That is, it was not able to get 100 streaks or not able to keep the pendulum vertical for 150 timesteps 100 times. Then the exploration rate was varied while keeping the learning rate constant even that had no results. Then I changed both and I was able to get a best conversion in 258-time steps. Moving forward I varied the the minimum learning rate. The following table shows the parameter values with the change in learning rates:

Episodes to converge	Best Q	Learning Rate
305	99.368716	0.5
259	99.487325	0.3
234	99.529058	0.1
267	97.024772	0.01
287	97.528066	0.001

The learning rate was always decreased logarithmically from 0.5 to the min_learning rate. We can infer from the table above that the minimum learning rate of 0.1 is giving the best results. This is because with a higher learning rate, we are allowing the new results to overpower the old results very suddenly and increase the variance. Whereas if the learning rate is too low, then the new results are virtually discarded, and the function takes time learning.

- Another optimization needed was the **reduction of state space**.
    - This was needed as, when the state space included all the four variables (x, x_dot, theta, theta_dot) with the bucket sizes as (3,3,6,3) respectively, it took the program a long time to converge. With a min learning rate of 0.1, it took 454 episodes with Best Q= 98.656561 and with a min learning rate of 0.01, the program did not converge even after 1200 episodes.
    - If I set the state space parameters as (1,1,6,3), the program takes much less time as the first two state spaces x and x_dot are ignored. This is a valid solution because I noticed that the cart didn't really move out of bound that often. Also, I only needed to balance the pole for 150-time steps. It typically didn't drift that far while balancing the pole. With this improvement the converging episode went down to 258 episodes. The effect of learning rate on such configuration is already discussed above.

- **Variation in min_explore_rate**: after experimenting with state space and learning rate, I tried playing with the exploration rate. This is crucial because too high an exploration rate and the program will never converge and too low the exploration rate and the program will never learn. Therefore, I varied the min exploration rate and the resultant data is shown below. The exploration rate always starts from 0.9 and is logarithmically reduced to the min_exploration rate.

Episodes	Best Q	Min Exploration Rate
366	99.930757	0.1
234	99.256767	0.01
289	99.675968	0.001

We can infer from the above reading that the min exploration rate of 0.01 is giving the best convergence episode by setting a right balance between exploration and exploitation. Therefore, the min exploration rate of 0.01 was fixed for further experimentations.

- **A final simulation with learning rate equal to 0.1 and exploration rate equal to 0.01, with states taken as (1,1,3,6), we get convergence in 234 episodes. The simulation can be found here.** (https://youtu.be/x-snt2aYUHU)

3.5 Why the need to implement Deep Q Learning?

The major limitation of the tabulated Q learning is the reduction in number of states. In my code I had to reduce the states from infinity (due to the observation's continuous nature) to 1*1*6*3 = 18 discrete states. Carrying out such generalization did make my code converge faster, however I wanted to know, if there was a way to handle the continuous data I was getting from the observations of the OpenAI environment.

Using neural networks with Q learning was the key here, that is deep Q learning. Neural networks are inherently efficient when handling very high dimensional problems. That's why they are doing so well with image, video and audio data. Additionally, they can easily handle continuous inputs, whereas with our classical approach we needed the Q-table to be a finite matrix Accordingly, with DQN we don't need discrete buckets anymore, but are able to directly use the raw observations. Therefore, a DQN network was formed and the results were compared with the simple Q learning.

4. Deep Q Learning

Deep Q learning is the use of neural networks with Q learning. Q learning is known to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the action-value (Q function). This instability has several causes: the correlations present in the sequence of observations, the fact that small updates to Q may significantly change the policy and therefore change the data distribution, and the correlations between the action-values (Q) and the target values $r + \gamma \max_{a'} Q(s', a')$. To address these issues a novel approach was introduced in the paper "Human-level control through deep reinforcement learning". [8] First, they used a biologically inspired mechanism termed experience replay that randomizes over the data, thereby removing correlations in the observation sequence and smoothing over changes in the data distribution. Secondly, they used an iterative update that adjusts the action-values (Q) towards target values that are only periodically updated, thereby reducing correlations with the target. Lets now understand this in detail.

4.1 Methodology

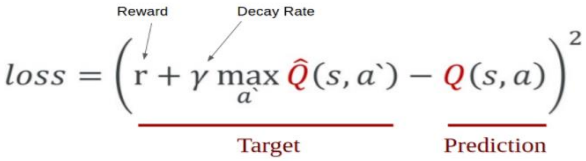
I have used OpenAi with built in modules like keras to implement the neural network. The DQN code was developed and modified with reference to a blogpost by keon. [7]. There is one input layer that receives the four state space inputs. There are three hidden layers with 24 nodes and the output layer has only two nodes for the two actions possible (0 or 1 i.e.; left or right).

The neural network can be setup using predefined functions as activation, loss and the optimizer. By defining each of the above parameters the neural network behaves differently. The loss function used here is mse or 'mean squared error'. The activation I set as 'relu'. For a neural net to understand and predict based on the environment data, we have to feed the information needed. Fit (), method feeds input and output pairs to the model. Then the model will train on those data to approximate the output based on the input. This training process makes the neural net to predict the reward value from a certain state. After training, the model now can predict the output from unseen input. When we call predict () function on the model, the model will predict the reward of current state based on the data we trained.

4.2 Implementing DQN

Normally in games, the reward directly relates to the score of the game. Imagine a situation where the pole from CartPole game is tilted to the right. The expected future reward of pushing right button will then be higher than that of pushing the left button since it could yield higher score of the game as the pole survives longer.

In order to logically represent this intuition and train it, we need to express this as a formula that we can optimize on. The loss is just a value that indicates how far our prediction is from the actual target. For example, the prediction of the model could indicate that it sees more value in pushing the right button when in fact it can gain more reward by pushing the left button. We want to decrease this gap between the prediction and the target (loss). We will define our loss function as follows.



We first carry out an action a, and observe the reward r and resulting new state s`. Based on the result, we calculate the maximum target Q and then discount it so that the future reward is worth less than immediate reward. Lastly, we add the current reward to the discounted future reward to get the target value. Subtracting our current prediction from the target gives the loss. Squaring this value allows us to punish the large loss value more and treat the negative values same as the positive values.Keras takes care of the most of the difficult tasks for us. We just need to define our target. We can express the target in a magical one-liner in python.
target = reward + gamma * np.amax(model.predict(next_state))

Keras does all the work of subtracting the target from the neural network output and squaring it. It also applies the learning rate we defined while creating the neural network model. This all happens inside the fit() function. This function decreases the gap between our prediction to target by the learning rate. The approximation of the Q-value converges to the true Q-value as we repeat the updating process. The loss will decrease and score will grow higher.

The most notable feature of the DQN algorithm was the use of experience replay. One of the challenges for DQN is that neural network used in the algorithm tends to forget the previous experiences as it overwrites them with new experiences. So we need a list of previous experiences and observations to re-train the model with the previous experiences. We will call this array of experiences **memory** and use **remember** () function to append state, action, reward, and next state to the memory. For example, the code will look like:

memory = [(state, action, reward, next_state, done)...]

And remember function will simply store states, actions and resulting rewards to the memory like below:
def remember (self, state, action, reward, next_state, done):
self. memory.append((state, action, reward, next_state, done))

Done is just a boolean that indicates if the state is the final state.
Replay: a method that trains the neural net with experiences in the memory is called replay(). First, we sample some experiences from the memory and call them minibatch. They can be defined by:

minibatch = random.sample(self.memory, batch_size)

The above code will make minibatch, which is just a randomly sampled element of the memories of size batch_size. We set the batch size as 64 for our case. To make the agent perform well in long-term, we need to consider not only the immediate rewards but also the future rewards we are going to get. In order to do this, we are going to have a 'discount rate' or 'gamma'. This way the agent will learn to maximize the discounted future reward based on the given state. The way the agent will act is that, it will first select action by a certain percentage called the 'exploration rate'. This is because it is better for the agent to explore all kind of things before it starts recognizing a pattern. When it is not deciding the action randomly, the agent will predict the reward value based on the current state and pick the action that will give the highest reward. **np.argmax()** is the function that picks the highest value between two elements in the **act_values[0].** Further explanation on this is given in the code itself.

The hyper parameters for DQN are:

- Model: Sequential; Input layer= 4 state space; Optimizer: Adam; Loss function: mse (mean squared error)
- Hidden layers= 2 layers of 24 nodes each(dense). Activation: relu
- Output layer = 2 as we only have two actions, Activation: Linear
- Learning rate= 0.01; minibatch size =64; Replay Memory Size= 100,000

4.3 Playing With the code and carrying out Optimizations:

1. It takes 8 minutes to train the network, which is more than time compared to normal Q learning,
2. The maximum value of score obtained was 500 after 2177 episodes.
3. Once trained the pendulum was able to remain vertical for a larger period of time as compared to the normal Q learning.
4. Change in learning rate is not significantly affecting the training time or the convergence episode.
5. Changing the activation of the hidden layers from relu to tanh. This was done as, relu is a linear function that maps the input to itself, but thresholded at zero. Therefore, for the actions that are negative, it will lead to the loss of neurons. The result was not distinguishable, because of the high covariance in the deep q network.
6. Increased the replay Memory size to 100,000 from 2000. To increase the variety in training samples. I think this helps in getting better training data. The result was that it took a longer time for the pendulum to remain vertical.
7. Tried using epsilon decay in place of logarithmic decay for exploration rate, just for the sake of trying. Using this I got better results than logarithmic decay. My training time reduced from 13 minutes to 8 minutes in just 2177 episodes.
8. Finally, I tried increasing the number of nodes in the hidden layers from 24 to 48 and increasing the minibatch size to 64. This also increased the training time, but I was able to get a solution or a score of 500 in 8 minutes.

4.4 Analysis of DQN

After making, may modifications to the code as cited above, I was finally able to get the network trained in 8 minutes. Using Experience replay was one of the key factors in decreasing the training time as well as for getting a convergence. Another important observation was that the DQN outputs are very random. That is, even for successive episodes the output or the score of the episode can vary greatly. Many times, it dint even converge after training it for 7000 episodes. This may be because at the beginning of each episode randomized data is selected.

All in all, once the network was trained, the pendulum was able to remain vertical for 500 timesteps. At this point I stopped the episodes, but it can continue to remain vertical.

The simulation of the DQN can be found here: https://youtu.be/ryP4QAFefhc

5. Comparing Q learning and DQN

	Q learning	Deep Q learning
Training Time	Convergence in 234 episodes. It took 7 minutes.	Trained for 2177 episodes in 8 minutes.
Performance	Can keep the pendulum vertical for 100 streaks	Can keep the pendulum vertical for indefinite time once trained.
State variables	Discretised into buckets	Divided into random minibatches.
Variance in results	Low variance	Very high variance.

6. Conclusion

In this project I simulated a cart pole problem in OpenAI gym. Initially I used a simple Q table to train the network. This was proved to be efficient as I was able to keep the pendulum upright for 100 streaks (100*150-time steps) in just 234 episodes. I observed that by only using the theta and theta_dot state spaces I can get faster convergence. Also, a min learning rate of 0.1 and an min exploration rate of 0.01 with logarithmic decay function proved to be the best parameters for my simulation.
As I wanted to use the continuous states available from the environment I tried implementing DQN. In this I was able to train the network in approximately 8 minutes and in 2177 episodes, after carrying out a lot of optimizations as discussed in the report.

To answer the question, should we use Q learning or DQN? I would say it depends upon the use. If the goal is to keep the pendulum upright for a very large period of time, then using DQN is the best choice. But if we want to train the network quickly and need to just keep the pendulum vertical for a brief time then Q learning should be used.

Another important observation is that DQN will be much more resistant to disturbances then q learning and therefore, if we want to implement this in real life where there are a lot of variables and noise, then DQN should be used.

The link to the codes can be found here.
https://drive.google.com/drive/folders/1VJiIQc60Jp_bAWOmdoX7uYc3a5XGMJJ6?usp=sharing

7. Future Work:

The first thing to do would be to improve the DQN code. With the current code I was able to get a solution of 500 streaks in 2177 episodes. This can be improved by playing with the hyper parameters.

The other thing is to try different learning techniques like imitation learning and genetic algorithm for the same problem and compare how the performance is changing.

Finally, once we have the best method to train the problem, we should apply it to a real system. That is train the network and then retrain it on a physical model. Observe the challenges of crossing the reality gap. Also, one can try implementing the swing up problem for cart pole or try and solve the acrobat problem given in OpenAI.

8. Bibliography

1. Roberge, J. K. (1960). The mechanical seal. PhD thesis, M.I.T., Cambridge, Massachusetts.
2. https://medium.com/@tuzzer/cart-pole-balancing-with-q-learning-b54c6068d947
3. https://gist.github.com/n1try/2a6722407117e4d668921fce53845432
4. https://en.wikipedia.org/wiki/Q-learning#Learning_rate
5. **Lecture Notes: Lecture8 (Q-Learning and Actor-Critic Methods)**
6. https://ferdinand-muetsch.de/cartpole-with-a-deep-q-network.html
7. https://keon.io/deep-q-learning/
8. **Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. & Hassabis, D. (2015). Human-level control through deep reinforcement learning. Nature, 518, 529--533.**