

Controlling Agents with Reinforcement Learning

Team Information

Name	SID	Part	Contribution
Denny-Thomas-Varghese	55506653	CartPole	33%
BATYRBEKOV Ilias	55591554	Mountain Car	33%
AGALTSOV Daniyar	55340231	Pendulum	33%

Background

Reinforcement Learning is one of the most important types of Machine Learning. The basic concept of Reinforcement learning is an Agent which learns how to behave in an environment by performing actions and receiving awards for its actions. OpenAI Gym provides various problems (environments) for reinforcement learning. The “Classic Control” problems are the combination of mechanics and reinforcement learning. Three “**Classic Control**” environments were chosen to demonstrate how reinforcement learning and Q Learning and Deep Q networks can be applied.

Deep Q Network For CartPole Problem

Goal: To move the cart from side-to-side and balance the pole on top of the cart.

Success evaluation: Balance the pole on the cart for 500 frames.

Failure evaluation: At any point in time (any frame), if the cart moves from the center by over 2.4 distance units or if the pole is off by an angle of 15 or higher compared to fully vertical.

Target goal: Set to obtain 500 continuous successful frames.

Values given by `env.step(action)` are `next_state`, `reward`, `done` and `info`.

`done` is a Boolean value that specifies whether the game (or rather the episode) has ended or no.,

`next_state` is the possible resulting state values from performing that action, `reward` is the obtained reward for performing that action, and `info` includes 4 values: the cart position, cart velocity, pole angle and pole velocity.

Actions (`action`) that can be performed by the agent are 0 and 1 for moving the cart left and right respectively.

The Deep Q Network or DQN

Hyperparameters: Number of Episodes, Exploration rate Epsilon and its decay and minimum value, Discount Rate Gamma, and Batch size.

Number of episodes: `env.reset()` starts the episode at state 0.

[Maximum] Number of time-steps: maximum number of time-steps that can be taken in one episode by the agent.

The neural network model is created using Keras. An empty neural network is created, with the parameters (that define the characteristics of the neural network), `relu` activation and `he_uniform` `kernel_initializer` to initialize the model. This is a three-layer neural network with node-counts 512, 256, and 64. This is a simple task, so these node-counts might seem overkill, but they seem to provide the best results. `.fit()` feeds the model the inputs and outputs to the model and this will train the

model. The model will approximate the output based on the input. MSE is used as the loss criterion, and RMSProp is used as the optimizer with hyperparameters lr, rho and epsilon tuned to obtain the best results observed. Keras is able to apply the idea of the loss function (the squared loss), and even apply a learning rate that may or may not be defined by the programmer (if absent, the model will define it by itself). Takes state and target. Furthermore, it is provided with a batch_size parameter value to decrease the gap between the predicted value by the learning rate; i.e the approximation of the q value converges to the actual q value as we repeat the process of updating the values. After training the model, .predict() is used so that the model will predict the output from an unseen input (current state) based on the training data. The loss function can be thought of in this way: First, the agent carries out an action a, obtains the reward r and state s. The immediate reward is then added to a value to obtain the target value. It calculates this value by finding the maximum target q for resulting state s and discounts it (using gamma) so that a future reward is worth less than the immediate reward. To account for punishing a larger loss more and to allow the treatment of positive and negative loss values as the same, the current prediction value is subtracted from the target and then squared. This is the resulting formula: $loss = (r + \gamma \max Q'(s, a') - Q(s, a))^2$

target_next is the model.predict() of the next state. There are two other important features of the DQN - the remember() and replay() functions. The remember() function is responsible for maintaining a “memory” of observations. This memory is a list of observations that is later used to re-train the model with old observations. The memory variable array element structure is the same as the structure returned by the environment when an action is performed. At this point, the epsilon value might need to be updated in order to account for the calculation of the cumulative discounted rewards for q value updation and maximize the immediate reward. The replay() function will sample some elements from

```
For Episode: 0 out of a maximum of 1000, the score is: 500
For Episode: 1 out of a maximum of 1000, the score is: 398
For Episode: 2 out of a maximum of 1000, the score is: 410
For Episode: 3 out of a maximum of 1000, the score is: 299
For Episode: 4 out of a maximum of 1000, the score is: 357
For Episode: 5 out of a maximum of 1000, the score is: 314
For Episode: 6 out of a maximum of 1000, the score is: 500
For Episode: 7 out of a maximum of 1000, the score is: 500
For Episode: 8 out of a maximum of 1000, the score is: 306
For Episode: 9 out of a maximum of 1000, the score is: 274
For Episode: 10 out of a maximum of 1000, the score is: 252
For Episode: 11 out of a maximum of 1000, the score is: 356
For Episode: 12 out of a maximum of 1000, the score is: 466
For Episode: 13 out of a maximum of 1000, the score is: 500
For Episode: 14 out of a maximum of 1000, the score is: 289
For Episode: 15 out of a maximum of 1000, the score is: 231
For Episode: 16 out of a maximum of 1000, the score is: 500
For Episode: 17 out of a maximum of 1000, the score is: 242
For Episode: 18 out of a maximum of 1000, the score is: 260
For Episode: 19 out of a maximum of 1000, the score is: 228
```

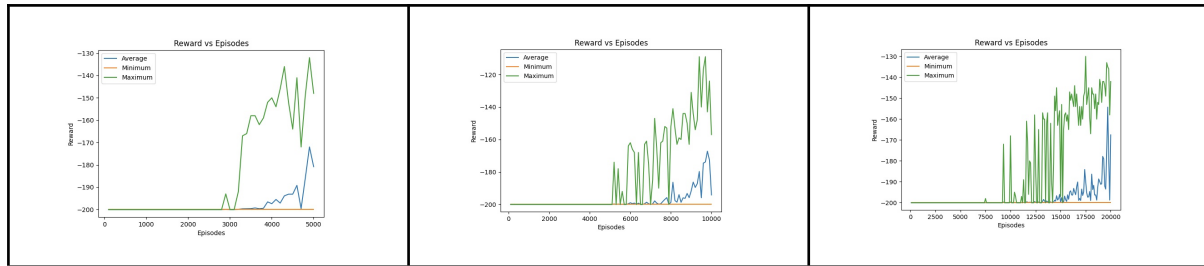
memory created in remember() based on batch_size, and re-train the model with those old observations in memory. The model is saved as an .h5 file to use for testing purposes. A function testModel() uses the previously saved .h5 file to test the performance (and observe scores) of the model, after the model is trained with the trainModel(). The results of 20 episodes are illustrated in the output screenshot (max score = 500). The model reached a score of 500 and completed training after 100 episodes.

Q learning for Mountain Car problem

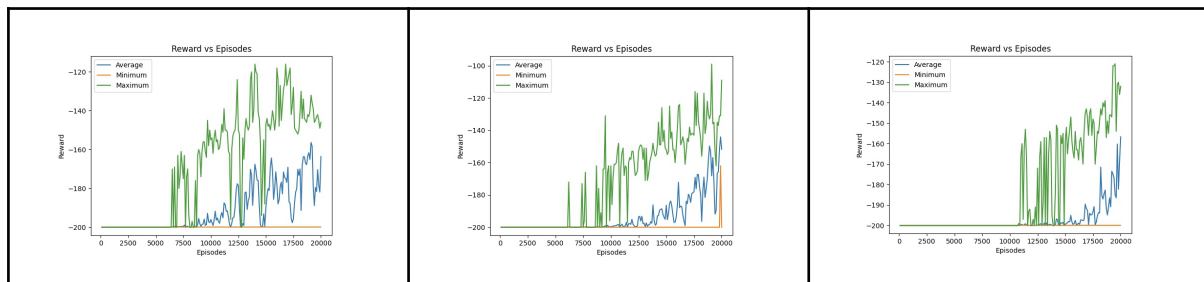
An agent in this environment has to reach the flag. Actions are chosen based on values in the Q table. Q table is initialized with random values (no prior knowledge). The shape of the table is (19, 15, 3) where 15 and 19 represent states between [-0.07,0.07] and [-1.2,0.6] respectively, and 3 represents actions: [move left, move right, don't move). After an episode Q-values are updated using this

formula: $Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t))$

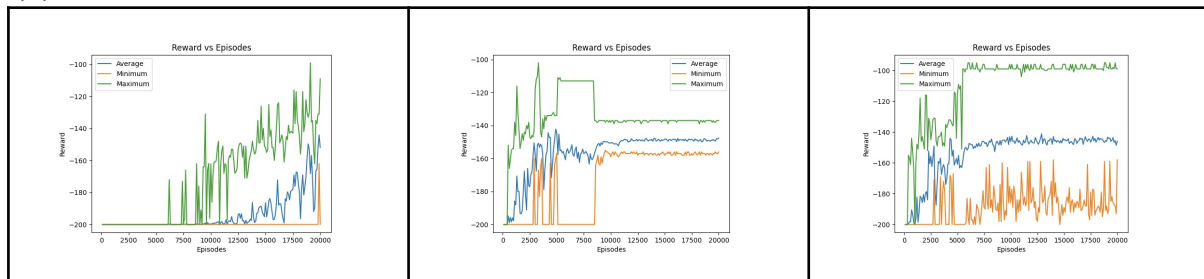
The more episodes an agent plays the better (at least in theory) its understanding of the environment. The below table shows rewards when the agent plays 5 000, 10 000, and 20 000 episodes respectively.



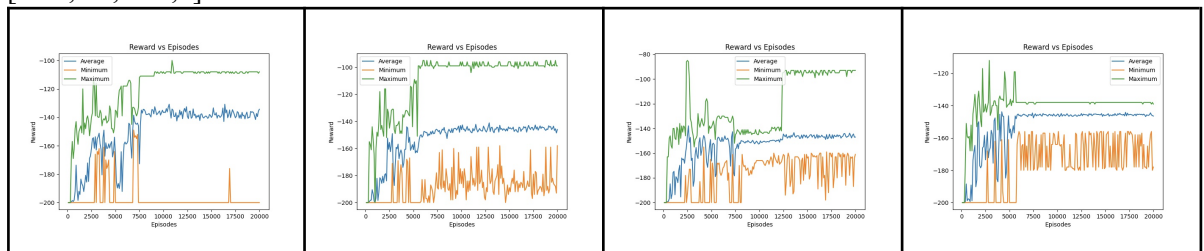
After approx. 15000 episodes there is almost no improvement, so we choose 20000 as the default number of episodes for agents to learn. Learning rate determines to what extent new information updates old Q values (Sutton & Barto, 1998). We use small alpha to keep previous knowledge, but large enough to update Q values overtime. Graphs with alpha values of 0.01, 0.1, and 0.2 respectively are shown below



We choose $\alpha=0.1$. Agents need to balance between exploration and exploitation. Exploitation: choosing an action based on Q-table (Matiisen, 2015), exploration: choosing a random action with probability of *epsilon* (Melo) to avoid local maxima. Epsilon decay is performed, so agents choose a single solution. Decay methods: linear in (a), exponential in (b) and (c) epsilon is divided by 2 and 1.5 respectively. Previously mentioned methods are shown in the table below in the following sequence: a,b,c.



With exponential decay epsilon reaches ~ 0 quickly, and an agent might converge at a local maxima. However, the best result was with method c, so we continue with method c. Gamma (discount factor) represents the weight of the rewards in the future. Since our environment isn't chaotic, we shall use large gamma because then the agent focuses on the long-term goal of reaching the flag. Gamma=1 can be used because of reachable terminal state (Russell & Norvig, 2010). Graphs with gamma values [0.85,0.9,0.95,1] is below:

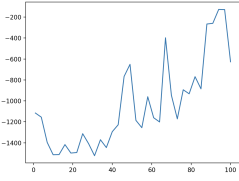
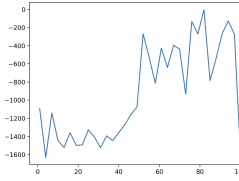
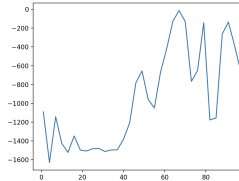


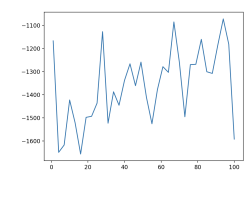
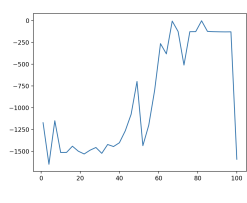
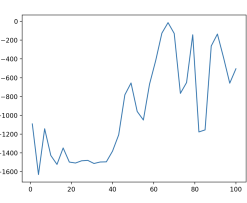
Gamma of 0.95 gave the best result. Because, agents have to climb a left-hill to gain momentum. Now, agents can reach the flag in 116 steps. Number of steps in last 4 episodes: [144, 116, 138, 141]

DDPG for pendulum problem

The main task of the pendulum problem is to apply force and torque to keep the pendulum in an inverted position for a maximum amount of time. This problem is quite harder than the mountain car problem due to continuous action space. In order to solve this problem, the Deep Determining Policy Gradients(DDPG) need to be used. DDPG is an algorithm which concurrently learns a Q-function and a policy. Since the action space is continuous, the Q function is differentiable with respect to the action argument. As a result, it allows us to make an efficient, gradient-based learning rule for a policy $\mu(s)$. Instead of expensive computing $\max_a Q(s,a)$, we can approximate $\max_a Q(s,a) \approx Q(s, \mu(s))$. The final solution can be divided into three parts, main training loop, Actor network and Critic Network. The DDPG relies on Policy-Gradient(PG) algorithms and Actor-Critic algorithms. PG algorithm computes noisy estimates of the gradient of the expected reward of the policy and then updates the policy in a gradient direction. Actor-Critic algorithm represents the policy function independently of the value function. The actor function outputs action given the current state. Critic Function outputs the Q value, which is the estimation of how good a state-action pair is. It inputs state and the output of the actor function. The learning process is based on the output of the critic function. Actor-critic architecture is shown on the figure. Apart from that, DDPG uses two techniques that are not present in DQN. Firstly, it uses two target networks to add stability in training. DDPG learns from estimated targets and target networks are updated slowly. Therefore, it keeps our estimated targets stable. Secondly, it uses experience replay, which is represented by BufferReplay class. BufferReplay stores a list of tuples(state,action, reward, next_state) and instead of learning only from the most recent experience, it learns from all experiences. In order to make Actor Network's exploration better, Ornstein-Uhlenbeck process is used to generate noise. It samples noise from a correlated normal distribution. This function was implemented in the OrnsteinUhlenbeckNoise class. In the final code, two Classes were created for two networks, Network_Actor and Network_Critic. Tflern was used in constructing the neural networks for both of these networks. Pendulum problems have a continuous action space, which represents a real line with boundaries. That's why we use tanh layer for the output of the actor network.

Critic Network inputs both state and actions as inputs. For the Actor and Critic Network, target networks are created and later target network parameters are updated by creating Tensor Flow Operation. After that, we need to compute gradients and optimize Tensorflow Operations in Actor Network. In order to do that, we use ADAM as an optimization method. For the Critic Network, we define loss as the Mean Squared Error and use ADAM as an optimization method as well. After that, we take action-value gradients and pass them to the policy network for gradient computation and actor network training. Finally, our solution was able to reach the goal after around 60 episodes.

Actor Learning rate = 0.0005	Rate = 0.001	Rate = 0.0001 - Best
		

Critic Learning rate = 0.0001	Rate = 0.01	Rate = 0.001 - Best
		

References:

Lillicrap, T., Hunt, J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., . . . Wierstra, D. (2019, July 05). Continuous control with deep reinforcement learning. Retrieved May 14, 2021, from <https://arxiv.org/abs/1509.02971>

Matiisen, T. (2015, December 19). Computational Neuroscience Lab.
<https://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning/>.

Melo, F. S. Convergence of Q-learning: a simple proof, Retrieved from
<http://users.isr.ist.utl.pt/~mtjspaam/readingGroup/ProofQlearning.pdf>

Russell, S. J., & Norvig, P. (2010). *Artificial intelligence: a modern approach*. Prentice-Hall.

Silver, D. (n.d.). Deterministic Policy Gradient Algorithms. Retrieved from
<http://proceedings.mlr.press/v32/silver14.pdf>

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning an introduction*. A Bradford Book.