Student No. 756011
Zeinab R. Yousefi

Assignment

ELEC-E8125
November 4, 2019

1 Task 1 : Implement Q-learning using function approximation in Cartpole environment

The part of code to implement Q-learning using function approximation is shown in the algorithm 1 and in order to update Q-values I use the 2 line of code in the main.py file.

Algorithm 1: =Q-learning using function approximation

```
def single_update(self, state, action, next_state, reward, done):
        # Calculate feature representations of the
        # Task 1: TODO: Set the feature state and feature next state
        featurized_state = self.featurize(state)
        featurized_next_state = self.featurize(next_state)

        # Task 1:  TODO Get Q(s', a) for the next state
        next_qs = [q.predict(featurized_next_state)[0] for q in self.q_functions]

        # Calculate the updated target Q- values
        # Task 1: TODO: Calculate target based on rewards and next_qs
        if(done == True):
            target = [reward]
        else:
            target = [reward + self.gamma * np.max(next_qs)]

        # Update Q-value estimation
        self.q_functions[action].partial_fit(featurized_state, target)
```

Algorithm 2: Update the Q-values

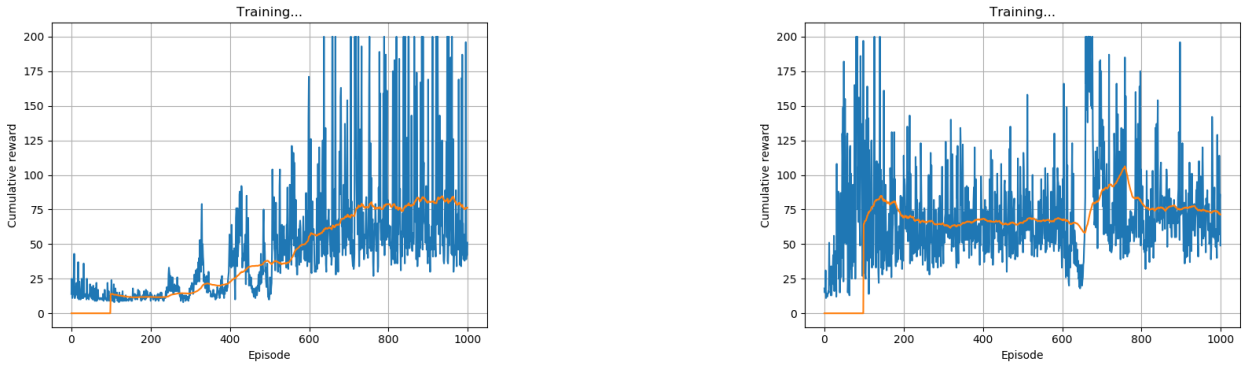
```
agent.single_update(state, action, next_state, reward, done)
```

- **a) handcrafted feature vector $\Phi(s) = (s, |s|)^T$**

  The featurize function for handcrafted feature vector is shown in algorithm 3:

Algorithm 3: Handcrafted feature vector

```
def featurize(self, state):
        if len(state.shape) == 1:
            state = state.reshape(1, -1)
        # Task 1a: TODO: Use (s, abs(s)) as features
        return np.concatenate([state, np.abs(state)], axis = 1)
```



(a) incremental update

(b) batch update

Figure 1: Training process using handcrafted featurizer

Figures 1a and 1b illustrate training process of Cartpole environment using handcrafted features, in incremental update (left) and batch update (right).

- **b) radial basis function representations (use the featurizer inside the Agent class).**

  The featurize function for RBF features is as algorithm 4:

Algorithm 4: RBF feature vector

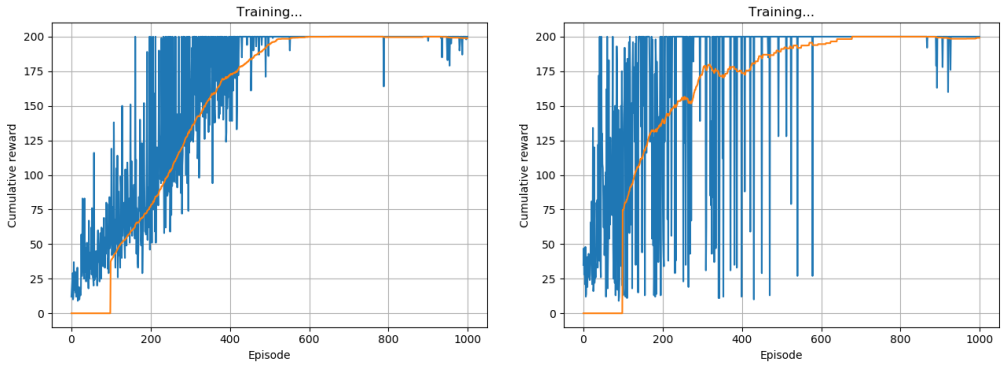
```
def featurize(self, state):
    if len(state.shape) == 1:
        state = state.reshape(1, -1)
    # Task 1b: RBF features
    return self.featurizer.transform(self.scaler.transform(state))
```

  Figures 2a and 2b show training process of Cartpole environment using RBF features, in incremental update (left) and batch update (right), respectively.

Question 1 - Would it be possible to learn Q-values for the Cartpole problem using linear features (by passing the state directly to a linear regressor)? Why/why not?

A linear function is not effective at learning Q(s,a) values for high dimensional problems, because the problems are inherently non-linear.

In the low dimension, many algorithms can achieve "good" performance, but in the high-dimensional big-data regime, the complexity of the model matters. Neural networks can learn and represent far more complicated functions.



(a) incremental update (b) batch update

Figure 2: Training process using RBF featurizer

2 Task 2 : Modify your Task 1 implementation to perform minibatch updates and use experience replay (while keeping the original code for Task 1 submission).

The implementation of minibatch update is shown in the algorithm 5.

Algorithm 5: minibatch update estimator

```
def update_estimator(self):
    if len(self.memory) < self.batch_size:
        # Use the whole memory
        samples = self.memory.memory
    else:
        # Sample some data
        samples = self.memory.sample(self.batch_size)

    # Task 2: TODO: Reformat data in the minibatch
    states = np.array([s[0] for s in samples])
    action = np.array([s[1] for s in samples])
    next_states = np.array([s[2] for s in samples])
    rewards = np.array([s[3] for s in samples])
    dones = [s[4] for s in samples]

    # Task 2: TODO: Calculate Q(s', a)
    featurized_next_states = self.featurize(next_states)
    next_qs = np.max([q.predict(featurized_next_states)
                      for q in self.q_functions], axis=0)

    # Calculate the updated target values
    # Task 2: TODO: Calculate target based on rewards and next_qs
```

```
targets = rewards + self.gamma * next_qs
targets[dones == True] = rewards[dones == True]

# Calculate featurized states
featurized_states = self.featurize(states)

# Get new weights for each action separately
for a in range(self.num_actions):
    # Find states where a was taken
    idx = action == a

    # If a not present in the batch, skip and move to the next action
    if np.any(idx):
        act_states = featurized_states[idx]
        act_targets = targets[idx]

        # Perform a single SGD step on the Q-function params
        self.q_functions[a].partial_fit(act_states, act_targets)
```

Question 2.1 - Which method is the most sample efficient, and why? Although the gradient descent is simple and appealing, it is not sample efficient (does not take all profit from samples). we have an experience, we see it, and the next we throw that experience away and get a new experience. That is not data efficient, we did not use the experience in the most efficient manner. Using minibatch updates is more sample efficient and is able to learn using only a few interactions with the environment. Between handcrafted features and RBF features, also the later one is more sample efficient.

Question 2.2 - How could the efficiency of handcrafted features be improved? Handcrafted features are always very task-specific and in most cases there are no possibilities of generalization in order to solve other tasks similarly. In low dimensional state space, the handcrafted features could be improved by acquiring the main knowledge about the rules of the environment. For example having features that are the linear function over states and actions in some problems could help. Designing task-specific features by hand is often very challenging and not very cost-efficient since a set of features that provides a good representation of the data for a task is usually worthless for all other tasks. Therefore, automatic feature learning is desirable.

Question 2.3 - Do grid based methods look sample-efficient compared to any of the function approximation methods? Why/why not? In the low dimensional problems, using grid methods by handcrafted features may be more sample efficient and accelerates the convergence. In this problems the rules of the environment are known and can be conveyed well through feature engineering, therefore sample efficiency can improve. However, by increasing the dimension of the states and actions the function approximation methods.

Question 2.4 - Which hyperparameters and design choices impact the sample efficiency of function approximation methods

Hyperparameters such as gamma (discount factor) and epsilon, and design choices such as number of hidden layers, optimizaer in NN affect the sample efficiency of function approximation methods.

3 Task 3 : Create a 2D plot of policy (best action in terms of state) learned with RBF with experience replay in terms of x and θ for $\dot{x} = 0$ and $\dot{\theta} = 0$.

The implemented algorithm to plot the policy heatmap is shown in 6.

Algorithm 6: policy heatmap

```
# Task 3 - plot the policy
discr = 128
x_min, x_max = -2.4, 2.4
th_min, th_max = -0.3, 0.3
x_grid = list(np.linspace(x_min, x_max, discr))
th_grid = list(np.linspace(th_min, th_max, discr))
policy = np.zeros([discr, discr])

policy = np.zeros((discr, discr))
for x in x_grid:
    for theta in th_grid:
        state = agent.featurize(np.array([x, 0, theta, 0]))
        qs = np.array([q.predict(state)[0] for q in agent.q_functions])
        a = np.argmax(qs, axis=0)
        policy[x_grid.index(x), th_grid.index(theta)] = a
plt.imshow(policy, cmap='hot', interpolation='nearest')
plt.show()
```

The 2D plot of policy learned with RBF with experience replay is shown in Figure 3. As it can be seen, the environment has been divided approximately to two main parts according to the best action.

4 Task 4 : Replace the RBFs in your Task 2 implementation with a neural network (while keeping the original code for Task 2 submission). Evaluate the method's performance in CartPole and LunarLander environments.

The part of code to computing the expected Q-values is shown in the algorithm 7

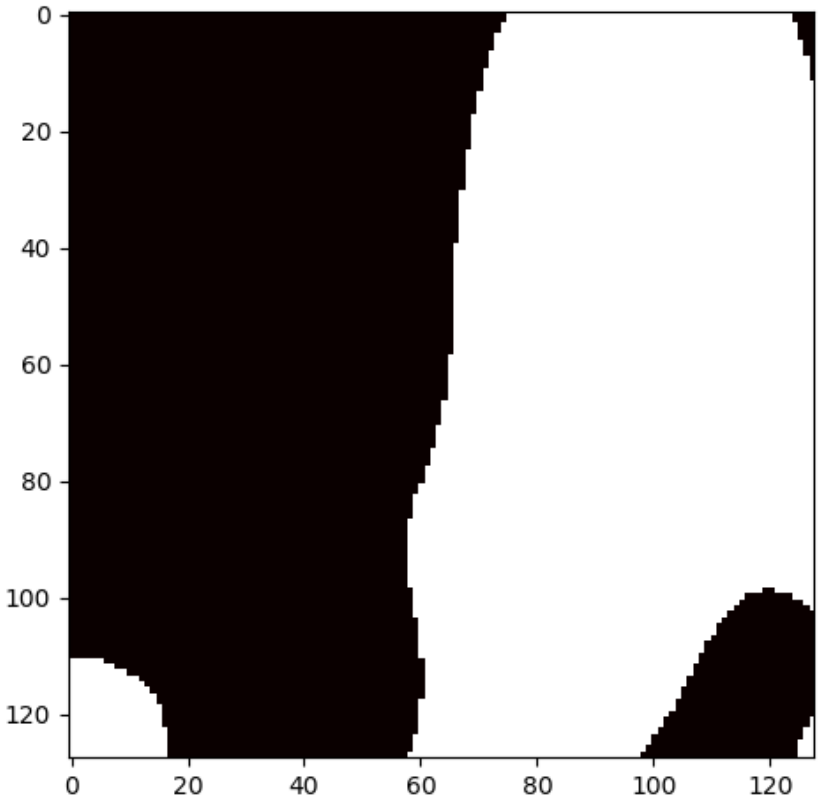


Figure 3: policy learned with RBF with experience replay

Algorithm 7: Compute the expected Q values

```
# Task 4: TODO: Compute the expected Q values
expected_state_action_values = reward_batch + (self.gamma * next_state_values)
```

And in order to update DQN the following algorithm 8 is implemented.

Algorithm 8: Update DQN

```
# Task 4: Update the DQN
agent.store_transition(state, action, next_state, reward, done)
agent.update_network(updates=1)
```

The training process for Cartpole and Landlunar environments are illustrated in figures 4 and 5 respectively.

Question 3.1 - Can Q-learning be used directly in environments with continuous action spaces?

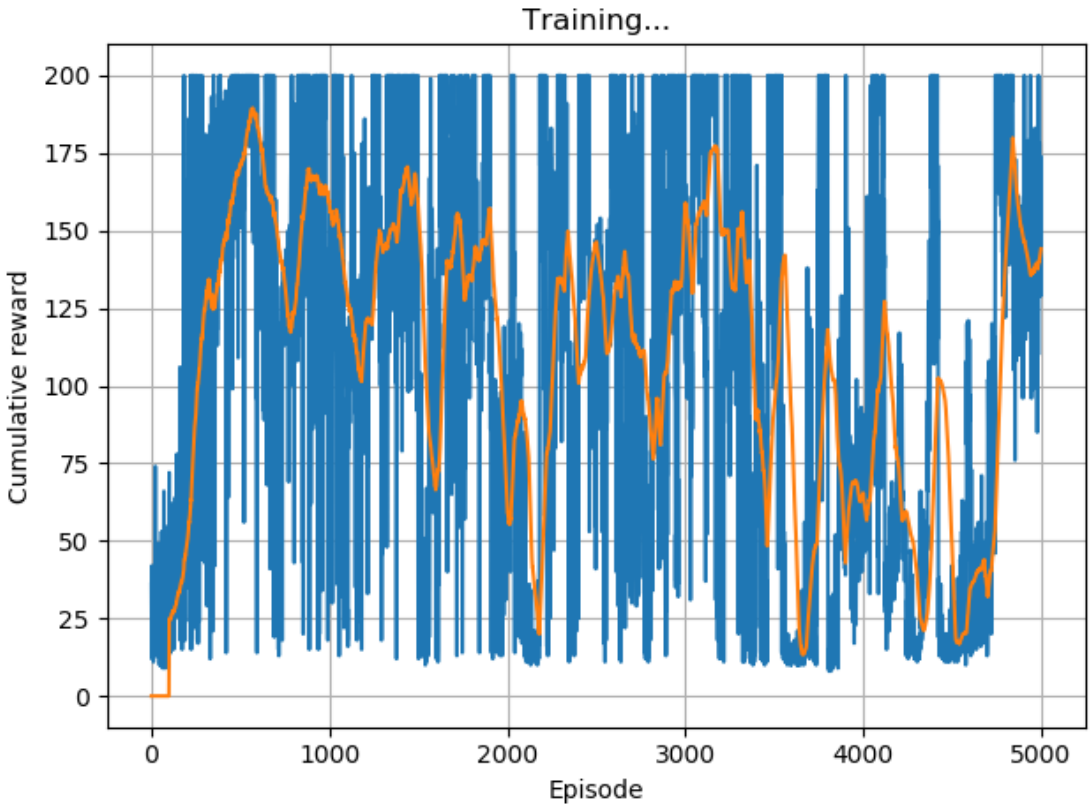


Figure 4: Training process of Cartpole environment using DQN

DQNs can only handle discrete, low-dimensional action spaces. In Q-learning, the goal is to learn a single deterministic action from a discrete set of actions by finding the maximum value.

Question 3.2 - Which steps of the algorithm would be difficult to compute in case of a continuous action space? If any, what could be done to solve them?

If we want to case of continuous action space, we need to approximate the continuous action space with discretisation. There is another kind of method called policy gradients, in which the goal is to learn a map from state to action, which can be stochastic, and works in continuous action spaces. Policy gradients seeks to directly optimizes in the policy space. The neural network (or other function approximators) can be used to directly model the action probabilities. Since the mapping function in policy gradient has to be some kind of approximator in practice, I think dicretisation of action space in Q-learning method can also be a good choice.

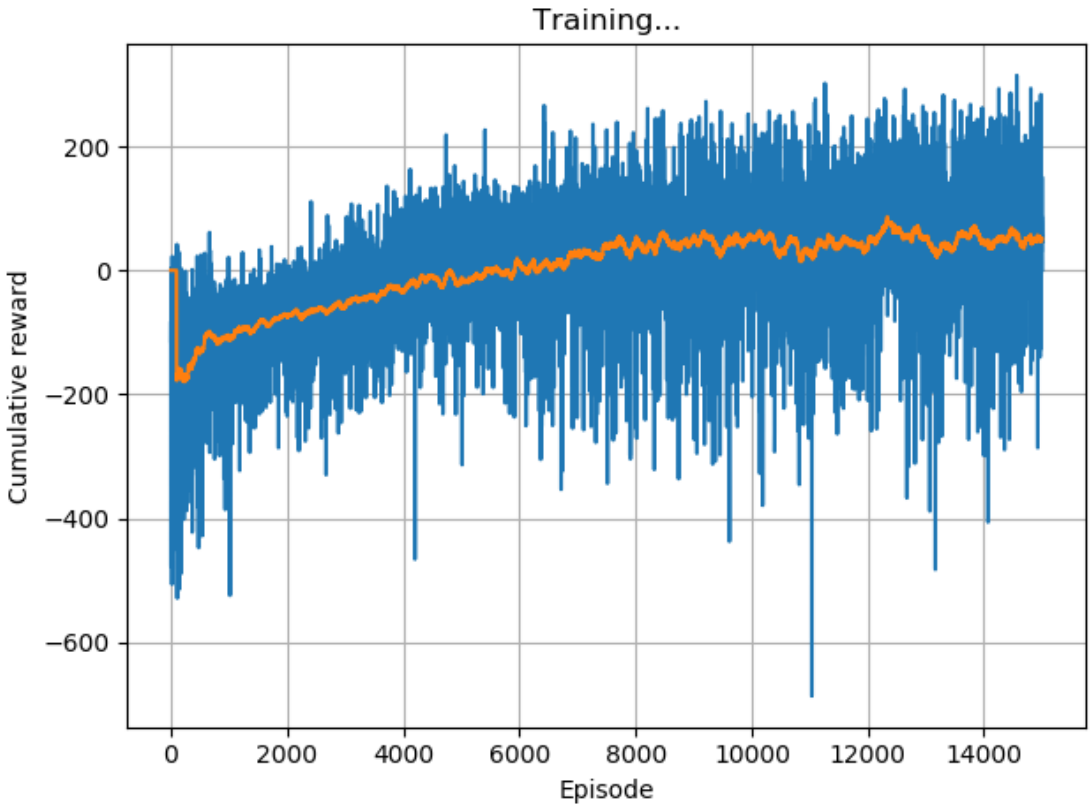


Figure 5: Training process of LandLunar environment using DQN