Assignment 2 - Convolutional Neural Network

Course: *Advanced Machine Learning*

January 21, 2019

1 Teammembers and Contributions

- **Bernd Menia**, *01316129*: Report, Programming, Testing
- **Maximilian Samsinger**, *01115383*: Main Programming, Testing, Plots

The difference in our contributions is because Bernd just only started working with Machine Learning, while Maximilian has already prior knowledge in the subject. Thus we worked together, but the main programming work for this assignment was done by Maximilian.

In this month's assignment we had to program the Mountain Car task, as initially described by Moore's PhD thesis in 1990 [Moo90]. Though as for the actual implementation we used the github project from Senadhi as a basis [Sen18].

2 Task i

Before the algorithm starts the policy approximation some values have to be initialized. Our initial state gets reset by calling $state = env.reset()$. Similar to that the first action gets set to 0, which corresponds to going left. The features get calculated by calling convert_to_features(). Depending on the task the flag $polynomial$ gets so that the function behaves as following: **Task i**, $polynomial = false$) Simply append the initial state with the first action and return it. **Task ii**, $polynomial = true$) Converts the state to a polynomial of degree max_degree. However, we cap the degree of action at 1, since we cannot expect higher powers of action to deliver meaningful extra features. Hereby the actions can be 0, 1 or 2, corresponding to *go left*, *do nothing* and *go right* respectively.

The main part of the code consists of 3 nested for-loops as seen in Figure 1. The only thing the outermost loop does is to completely run the code n times. This was just done for convenience so that Task iii can be done without having to manually restart the program. The two inner loops are more interesting: The middle loop represents each episode of the algorithm while the innermost loop represents the steps that are made in each generation.

Figure 1: Main part of the program with 3 nested for loops

```python
for j in range(NUM_RUNS):
    ''' Start of another run '''
    print('Run:', j)
    weights = np.random.rand(NUM_FEATURES)

    for k in range(NUM_EPISODES):
        ''' Start of another episode '''
        state = env.reset()
        action = choose_action(state, weights.copy())
        cumulative_reward = 0
        for step in range(MAX_EPISODE_STEPS):
            env.render()
            previous_state = state
            state, reward, done, info = env.step(action)
            next_action = choose_action(state, weights.copy())
            weights = update_weights(previous_state, state, action,
                                     next_action, weights.copy(), reward, done)

            cumulative_reward += reward
            if done:
                break
            action = next_action
        average_reward = cumulative_reward
        average_rewards[j,k] = average_reward
```

At the start of each episode the middle loop chooses the action that gets used, depending on which action is most suitable (see Figure 2). From the second episode onwards the action that gets used is chosen by the algorithm, depending on which action is most suitable (choose_action()). This is done creating a random number and comparing it to a threshold, epsilon, which we set to 0.33. In other words this means that in 33% of all cases the algorithm performs a random action. This is done to ensure that even when the algorithm is shifting towards a certain policy and is therefore almost fixed in its actions, there is still the possibility that it performs random actions which might lead to better results. Note that the action also gets chosen anew in each step of each episode (innermost loop).

Initially we set the percentage of random actions to just 2% which we thought would be good, but somehow the results were abysmal and the mountain car almost never got to the top. Once we increased the amount random actions to be 33% the mountain car got to the top a lot of times.

Figure 2: Choose an action with which the algorithm proceeds

```python
def choose_action(state, weights, epsilon = 0.33):
    ''' A random action will be picked with probability epsilon '''
    if epsilon > np.random.uniform():
        action = np.random.randint(0, NUM_ACTIONS)
    else:
        action = np.argmax([action_value(state, action, weights)
                   for action in range(NUM_ACTIONS)])
    return action
```

After an action has been chosen the algorithm performs a step. The step also calculates the new state and the respective reward of the previous state in combination with the action.

The reward gets calculated as follows: For each step that gets performed and after which the mountain car is not at the goal the reward is -1. On the contrary if after a step the goal got reached the reward is positive, though no exact numbers are given.

The only thing left to do is to update the weights so that the policy gets approximated to the best possible values. As long as the goal hasn't been reached the weights get updated by taking into account the action_value functions of the current- and previous states and also two not yet seen variables, learning_rate and discount. The learning rate is used to decrease the amount of calculations that we have to make. Theoretically in order to get the best possible policy we would have to perform all combinations of sets and actions that exist. However this amount of complexity is not practical and cannot be achieved in a senseful manner. So instead of trying out all combinations of states and actions we choose one as described in choose_action() and then dampen the value of the chosen action by multiplying the outcome with the learning_rate.

The other variable is the discount factor. The discount also acts as a kind of dampening factor, but it is more general than the learning_rate. The exact formula how weight update gets calculated can be seen in Figure 3.

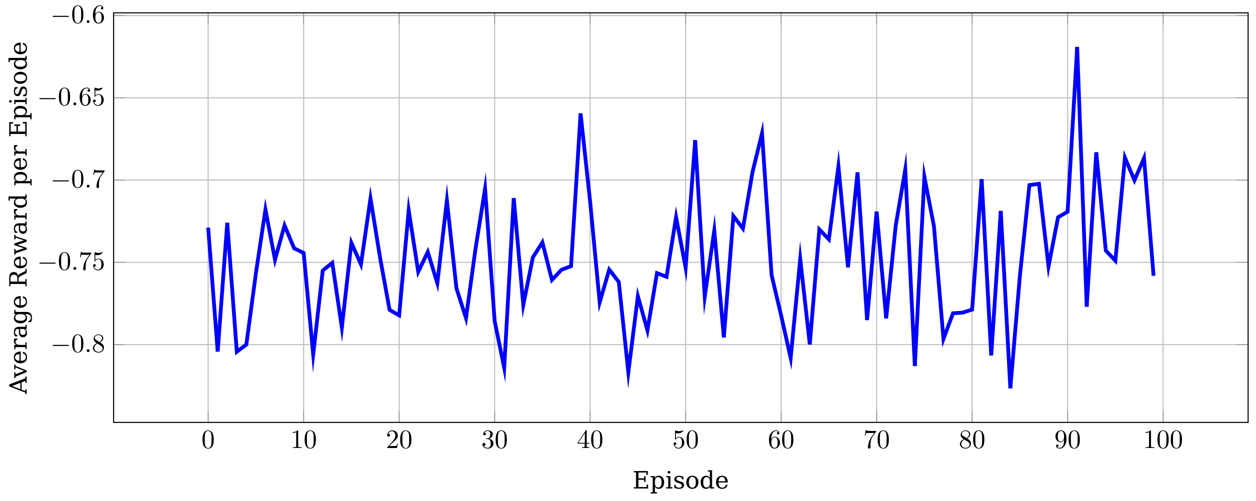
Figure 3: Function that updates the weights after a step

```python
def update_weights(previous_state, state, action, next_action,
                   weights, reward, done = False):
    gradient = convert_to_features(previous_state, action)
    if done:
        update = learning_rate * (reward
                        - action_value(previous_state, action, weights))

    else:
        update = learning_rate * (reward
                        + discount * action_value(state, next_action, weights)
                        - action_value(previous_state, action, weights))
    update *= gradient
    weights -= update # Gradient descent
    return weights
```

The results from the firs Task are visualized in Figure 4. Unfortunately there isn't much to say about the graph as it is highly unstable. The rewards appear to be randomly distributed within a a range of -0.82 and -0.68. Strange enough the rewards don't seem to converge towards a higher number with each episode which was what we expected at first. We don't know why this isn't the case, but it could very well be that a bug slipped into our code that we haven't identified yet.

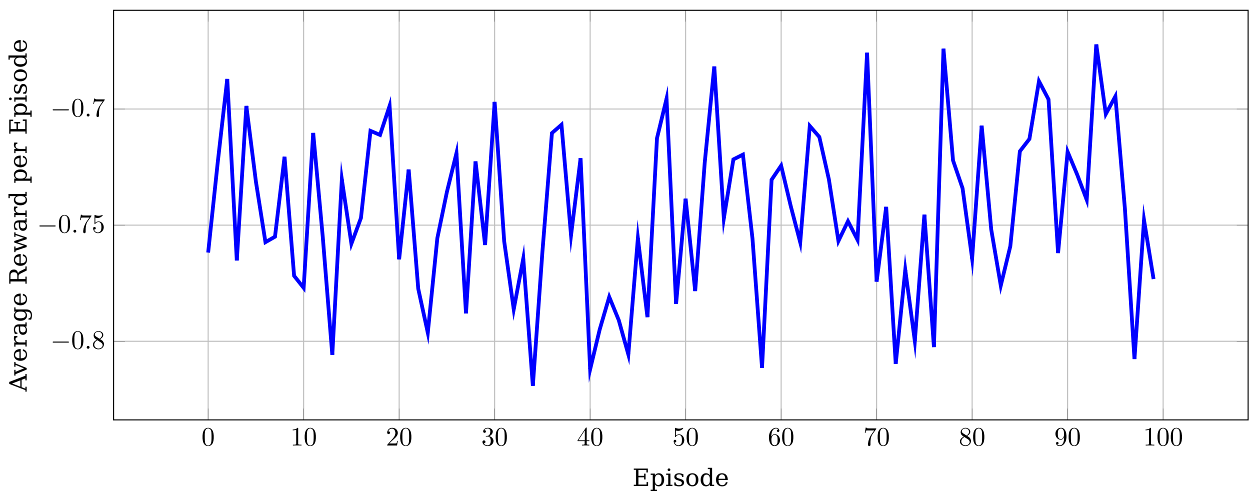
Figure 4: The average reward per episode after 50 runs (Task1)



3 Task ii & Task iii

This task is quite similar to the first one. The difference is that the weights get initialized differently. How this is done is already described in section 2. Task ii also has a highly unstable average reward, but compared to Task i the rewards seem to jitter even more.

Figure 5: The average reward per episode after 50 runs (Task2)



4 Discussion

As we have seen we ran into some problems while working on this assignment. First of all using polynomials as the initial values to approximate the action-value function is not good because it is highly likely that these values differ vastly from the actual underlying function that we try to approximate. Of course this is partly inevitable because by definition we don't know the underlying function yet and therefore have to approximate it. But there are far better methods to start the approximation, for example by using a Neural Network at first and then transferring the output to the Reinforcement Algorithms as the input, which is also the process that was used to train AlphaGo.

Another thing to mention is that our algorithm is unstable. Even after letting the algorithm run for 1000+ episodes the results from the episodes vary strongly as seen by Figure 4 and Figure 5. Unfortunately we don't know why this is the case, but we have spoke with multiple colleagues of us from the course and their programs have similar behaviour.

Finally the major problem in our code is that our algorithm is prone to unlearning previously learned solutions. It may happen that it discovers a good way to drive upon the mountain, but the solution gets overwritten by the next episodes and therefore unlearns what brought it to the goal, which might be the reason why the graphs are so jittery.

References

[Moo90] Andrew William Moore. Efficient memory-based learning for robot control. Technical report, 1990.

[Sen18] Vishma Senadhi. Mountaincar-v0. URL: (https://gym.openai.com/envs/MountainCar-v0/, August 2018.