703642, Advanced Machine Learning course 21st January, 2019

Assignment 3

Programming Project: Reinforcement Learning

Kamal Zakieldin, Yusuf Ipek

1. Introduction

Reinforcement Learning is an aspect of Machine learning where an agent learns to behave in an environment, by performing certain actions and observing the feedback which it gets from those actions, to build a policy that can act appropriately according to these observations.

In this report we are describing our methods we have applied to the well-known **Mountain Car** problem [1] and our experiments we have made. First, we introduce our problem and our tuning showing the experiments we had. Afterwards, we discuss our chosen parameters and the results we got. Furthermore, we compare the results we got and finally give some recommendations for future tuning.

2. Problem description

The goal is to drive the car up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum.
The agent can perform three different actions; move left, move right or do not move.

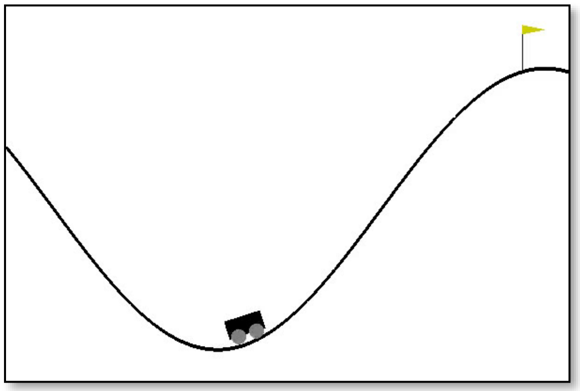


Figure1: Mountain Car environment

3. Experiments

At the beginning we started to try tabular algorithms for this problem, we have tried **tabular q-learning** algorithm and **tabular SARSA** algorithm, but after performing many tries with different parameters, such as changing the learning rates, exploration greedy (epsilon), but the agent could not learn the appropriate policy.

One of the main reasons for that was the rewarding system, giving a **-1** as a reward after every step was not a great indication, the more you spend iterations the more punishment you get. Theoretically this is a good approach, but because the agent never receives a positive reward and was not able to figure out the optimal policy till it explores the goal.

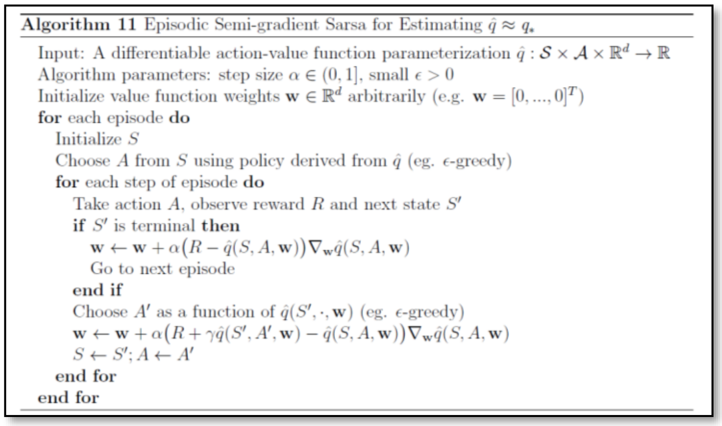
This raised the conflict of **the exploration versus the exploitation** between doing more exploration to search about the goal or be greedy and reduce the number of iterations to reduce the negative rewards. Without exploration, you will never know what is better ahead. But if it is overdone, we are wasting time. So, we have an exploration policy, like epsilon-greedy, to select the action taken in step 1. We pick the action with highest *Q* value but yet we allow a small chance of selecting other random actions. *Q* is initialized with zero. Hence, there is no specific action standing out in early training. As the training progress, more promising actions are selected and the training shift from **exploration to exploitation**.

It totally makes sense that **tabular methodologies** did not work, because tabular methodology works fine with small problems with small environment, although our environment does not look so complex, but it is not small and tabular methods doesn't achieve good results with it. thinking deeply about our problem space, we have only three actions to get observations about the state (the position and the velocity of the agent).

That's why combining the velocity and the position together generate a large environment that needs **function approximation methodologies** parameterized by some weight vector.

In these methodologies, the RL algorithms updates the parameters (weight vector) during the exploration phase. we are required to use a linear value-action function approximation.

First, we tried the **semi-episodic** algorithms to solve the problem without using any neural network architecture as a linear method.



```
Algorithm 11 Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$
  Input: A differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
  Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
  Initialize value function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g. $\mathbf{w} = [0, ..., 0]^T$)
  for each episode do
    Initialize $S$
    Choose $A$ from $S$ using policy derived from $\hat{q}$ (eg. $\epsilon$-greedy)
    for each step of episode do
      Take action $A$, observe reward $R$ and next state $S'$
      if $S'$ is terminal then
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big(R - \hat{q}(S, A, \mathbf{w})\big)\nabla_{\mathbf{w}}\hat{q}(S, A, \mathbf{w})$
        Go to next episode
      end if
      Choose $A'$ as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (eg. $\epsilon$-greedy)
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big(R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big)\nabla_{\mathbf{w}}\hat{q}(S, A, \mathbf{w})$
      $S \leftarrow S'; A \leftarrow A'$
    end for
  end for
```

Figure 2: Episodic SARSA algorithm

We have tried many different methods to construct the feature vector, first we tried the logistic function to build the features (**$\varphi(S) = S^j$** , when S is the state and j = 0, ..., n), however this feature representation did not give us good results, so we have tried the **Fourier basis method**[2] which leads to an expansion in sinusoidal functions.

We have faced some problems during constructing the feature vector **φ**, that's why, we decided to implement another approach thinking about supervised machine learning by using a **neural network** with **one linear layer** to build the agent.

We use supervised learning to fit the *Q*-value function. In RL, we search better as we explore more. So, the input space and actions we searched are constantly changing. In addition, true labels are not available, hence we substitute it with [$R_t + \gamma Q(S_{t+1},A_{t+1},w)$] as the target value as well as the label for our neural network turning the problem into a supervised learning problem solvable using gradient descent.

We have tried different models with deep Q-network algorithm, but we could not get an optimal policy as q-learning doesn't converge in linear methods, so we tried deep SARSA network algorithm, with different number of units in the hidden layer and different epsilon value, table 1 shows the results we got.

	Model	epsilon	Testing acc.
a)	DQN with 300 units (with decreased epsilon)	0.3	0.52
b)	DQN with 500 units (with decreased epsilon)	0.2	0.48
c)	DQN with 500 units (with constant epsilon)	0.2	0.55
d)	SARSA NN with 400 units (with constant epsilon)	0.3	0.55
e)	SARSA NN with 500 units (with constant epsilon)	0.2	0.53
f)	SARSA NN with 500 units (with constant epsilon)	0.3	**0.61**
g)	SARSA NN with 700 units (with constant epsilon)	0.3	0.57

Table 1: different models for a one hidden layer NN using deep Q-network (DQN) models (a – c) and deep SARSA-network algorithm models (d – g) after training for 3000 episodes.

It is important to notice that increasing the number of units in the hidden layer have made a better performance increasing the number of total successful episodes.

During this experiment we have some parameters to tune, one of the most vital parameters that needed to be tuned is the exploration **epsilon** greedy parameter, choosing the epsilon high will make the agent act greedy with less exploration, in such a problem, exploration is critical and without making enough exploration the agent may not find the goal at all.

We also tried to **adjust the epsilon** after each successful step, by slightly reducing it after each successful episode (to act less greedy and explore more) and surprisingly this increased the number of successful episodes in the training phase but reduced slightly the number of successful episodes in the testing phase. on the other hand, increasing the epsilon after every successful episode made the agent more greedy and reduce the exploration, hence, the number of successful episodes have decreased dramatically, it is not a good idea to be greedy during training the model.

It was clear that q-learning will not converge. However, although SARSA has achieved a higher accuracy, **SARSA could not also converge or reach the optimal policy but chattered a lot**.

Finally, we have tried **polynomial** methods using the **tile coding** method. and surprisingly, it has shown a very good progress, although it may take longer time for training, but overall it does not need the same number of episodes to find the goal for the first time, and after few episodes it can converge.

Using semi-episodic algorithm has achieved a real convergence if we compared with earlier experiments, as we will see in section 4.

We have used a ready code doing tiling by Richard S. Sutton[3] , the idea behind tiling is just dividing the space into number of partitions, we have chosen 16 partitions as recommended from the lectures, **each partition is a tiling**, and element in the tiling is a tile, we are creating the features of each partition by combining the position and the velocity of the object to have a feature represents these tiles in its tiling partition.

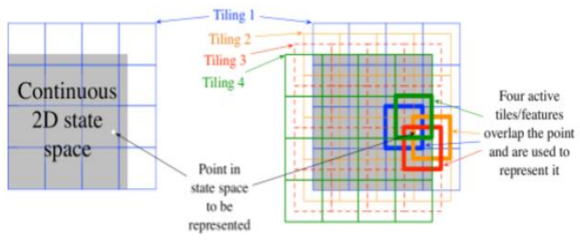

Tiling 1
Tiling 2
Tiling 3
Tiling 4
Continuous 2D state space
Point in state space to be represented
Four active tiles/features overlap the point and are used to represent it

Figure 3: Tile Coding

One important thing is that tiling is only a map from (state, action) to a series of indices, it doesn't matter whether the indices have meaning, only if this map satisfy some property. after trying some iterations, we found that our space is small to do a tiling for 16 partition, so we have tried 8 partitions and it has shown better results as shown in table 2.

	Model	epsilon	*Train Acc.	Test Acc.
a)	SARSA Tile-Coding (8 partitions, decreased epsilon) 500 steps**	0.2	0.366	0.66
	SARSA Tile-Coding (8 partitions, constant epsilon), 500 steps**	0.3	0.386	**0.97**
c)	SARSA Tile-Coding (8 partitions, decreased epsilon), 1000 steps**	0.2	0.812	**100% (converge)**
d)	SARSA Tile-Coding (8 partitions, constant epsilon), 1000 steps**	0.3	0.806	**100% (converge)**

Table 2: different tile coding agents with different epsilon values.
**Training accuracy is the ratio of the number of succeeded episodes during training the agent.*
*** number of steps per episode*

It was clear that with few tuning, **semi-episodic SARSA with Tile-Coding has found the optimal policy** and can converge if it could do enough exploration, in section 4 we are giving a complete discussion about choosing the steps the agent can perform before termination at each episode.

4. comparisons

In this section we are going to compare between the results we got in task (i) by applying a linear neural network, and the results we got in task (ii) by applying polynomial tile method.

First, we discuss our observations for the **SARSA neural network** algorithm, Figure 4 shows the agent interacting with the environment 500 steps per episode. The agent starts to find the goal shortly after around 10 episodes and after 100 episodes it finds the goal regularly, so the policy gets better. However, we can observe that the agent fluctuated to find the goal in continuative sequence of episodes. We increased the environment interaction of the agent by increasing the number of steps before terminating, which leads the agent to find the goal more often in the first 100 episodes, as can be seen in Figure 5. Again, we can observe that the agent fluctuated in finding the goal in continuative sequence of episodes. An interesting fact is, the policy in Figure

5 has a higher average reward (best average reward around **-0.2**) compared to the policy in Figure 4(best average reward is around **-0.3**), which is probably due to the higher exploration possibility.

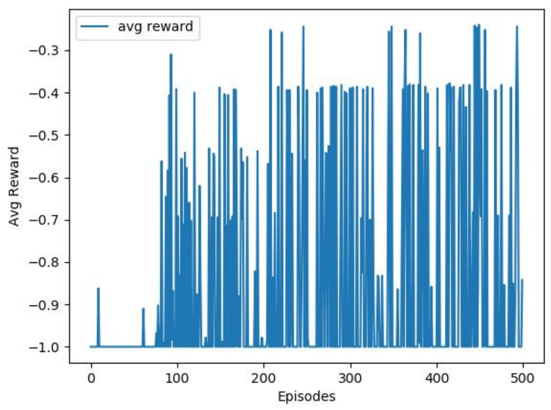


Figure 4: Neural Network SARSA – Average rewards per episode with 500 steps for 500 episodes

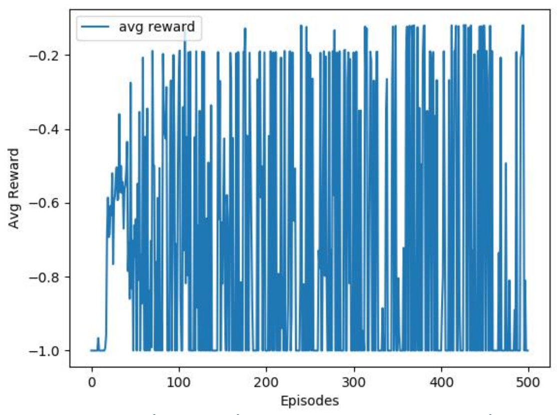


Figure 5: Neural Network SARSA – Average reward per episode with 1000 steps for 1000 episodes

These plots have shown that the agent has found a good policy to reach the goal, but sometimes it may miss it, which proves **it is not the optimal policy.**

On the other hand, in **SARSA with tile coding**, Figure 6 shows the policy where the agent interacted with the environment 500 steps per episode and found the goal after more than 100 episodes. However, reaching the optimal policy starts around 500 episodes, to prove that the policy starts to converge at that point we increased the episodes to 1000 and we observed that the policy got much better, which can be seen in Figure 7. The reason why the policy started to converge toward the optimal policy such late is due to the amount of interaction steps of the agent with the environment per episode.

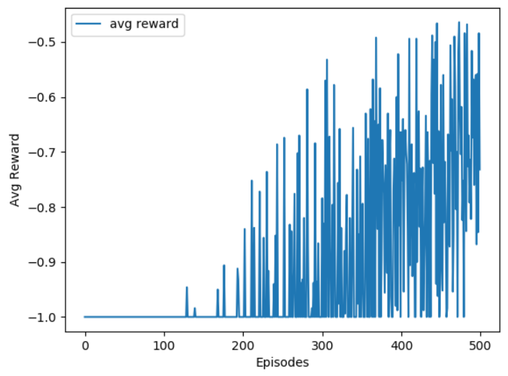


Figure 6: Tile SARSA – Average rewards per episode with 500 steps for 500 episodes

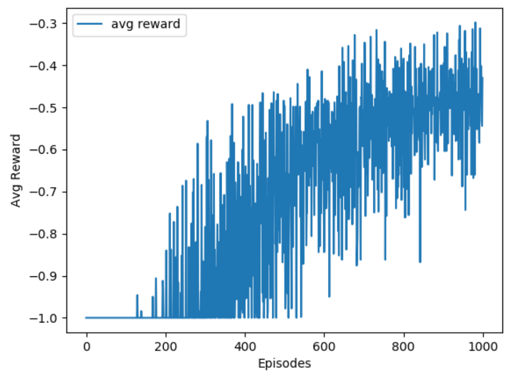


Figure 7: Tile SARSA – Average rewards per episode with 500 steps for 1000 episodes

In order to find the optimal policy in fewer episodes we increased the interaction step per episode to 1000 steps. The improvement can be observed in Figure 8. With the configuration the agent started to reach the goal during the first 100 episodes. Furthermore, the policy has started to converge toward the optimal policy for episodes higher than 100, which is due to the higher exploration possibilities of the agent. The agent spends more time to explore the environment. In order to show that the optimal policy is found, we increased the episodes to 1000, we can see that the policy doesn't increase further as reflected in Figure 9. Another interesting fact is that the new policy finds the goal in fewer steps, where the best average reward is less than **-0.2**, in contrast to the policy in Figure 7, where the best average reward is around **-0.3**.

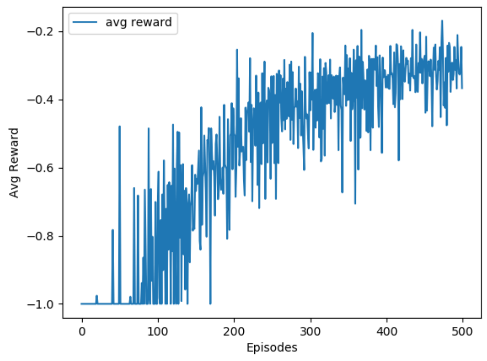


Figure 8: Tile SARSA – Average rewards per episode with 1000 steps for 500 episodes

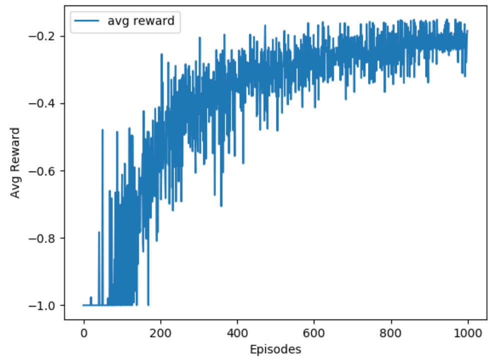


Figure 9: Tile SARSA – Average rewards per episode with 1000 steps for 1000 episodes

These plots have shown that **the agent found the optimal policy** and regularly can reach the goal, however, in some cases it may take longer to find the goal but eventually it reaches it.

5. Conclusion

In conclusion, we have tried some approaches hopefully to find the optimal policy, we have found some interesting information to share, first tabular algorithms don't work in all problems, second, Q-learning algorithm usually doesn't converge, third, SARSA can chatter around the optimal policy, fourth, tile coding has shown great results compared with the SARSA NN agent, and we think mainly the reason for that is the way of choosing the feature vector, in tile coding, we try to choose a small number of tiles to represent the state according to the partitions we have generated . We also have thought about some other approaches that we have not considered, which may lead to better results such as investigating more in constructing the feature vector or trying different polynomial method such as radial basis functions RBF.

6. References

[1] https://gym.openai.com/envs/MountainCar-v0/
[2] https://en.wikipedia.org/wiki/Basis_function#Fourier_basis
[3] http://incompleteideas.net/tiles/tiles3.html
[4] https://tomaxent.com/2017/07/05/On-policy-Prediction-with-Approximation/
[5] https://github.com/vmayoral/basic_reinforcement_learning
[6] https://danieltakeshi.github.io/2016/10/31/going-deeper-into-reinforcement-learning-understanding-q-learning-and-linear-function-approximation/