ADAPTIVE TILE CODING METHODS FOR THE GENERALIZATION OF VALUE FUNCTIONS IN THE RL STATE SPACE

A THESIS

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL

OF THE UNIVERSITY OF MINNESOTA

BY

BHARAT SIGINAM

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF SCIENCE

RICHARD MACLIN

MARCH 2012

Abstract

The performance of a Reinforcement Learning (RL) agent depends on the accuracy of the approximated state value functions. Tile coding (Sutton and Barto, 1998), a function approximator method, generalizes the approximated state value functions for the entire state space using a set of tile features (discrete features based on continuous features). The shape and size of the tiles in this method are decided manually. In this work, we propose various adaptive tile coding methods to automate the decision of the shape and size of the tiles. The proposed adaptive tile coding methods use a random tile generator, the number of states represented by features, the frequencies of observed features, and the difference between the deviations of predicted value functions from Monte Carlo estimates to select the split points. The RL agents developed using these methods are evaluated in three different RL environments: the puddle world problem, the mountain car problem and the cart pole balance problem. The results obtained are used to evaluate the efficiencies of the proposed adaptive tile coding methods.

Acknowledgements

I am indebted to my advisor Dr. Richard Maclin for giving me an opportunity to work under him and for patiently guiding me during the course of my thesis-work.

I would like to thank my committee members Dr. Hudson Turner and Dr. Marshall Hampton for considering my request to evaluate the thesis.

I would also like to thank all the faculty and staff at Computer Science Department and Dr. Joseph Gallian for helping me during my course work and TA work.

Contents

List of Figures

List of Tables

CHAPTER 1

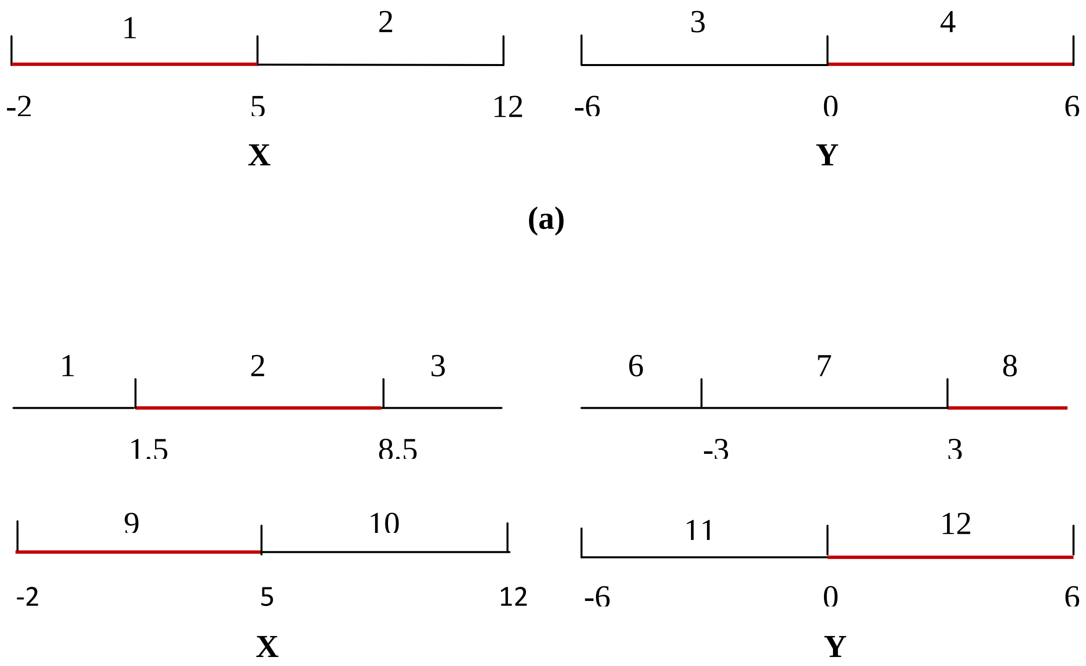
INTRODUCTION

Artificial Intelligence (AI) is a branch of computer science that studies and develops intelligent agents to learn to act in new environments. *Machine Learning* (ML) is a branch of AI that designs algorithms to generalize the observed data of an environment for automating the estimation of patterns and to use these patterns to maximize the performance of an agent in the environment. Multiple approaches like Supervised Learning, Unsupervised Learning, Semi-supervised Learning, Reinforcement Learning, etc., are available to implement ML algorithms. In this work, I used the algorithms developed using Reinforcement Learning methods.

Learning by interaction is one of the most effective and natural methods of learning. It is also one of the important inherent attributes of humans and animals. Making a move in chess, adjusting the thermostat and increasing/decreasing the speed of car are all examples of interaction-based activities. These interactions can be used by an agent to learn about the environment and improve its performance in the environment. For example, a player while making a move in chess tries to use all the experience he gained from playing chess previously. This interaction-based learning can be applied to machines by asking an environment to reward an agent for key actions performed by the agent (often just at goals). This approach is implemented as *Reinforcement Learning* (RL), a goal-directed and interactive learning mechanism wherein an agent interacts with an environment to learn what action to take given a state of the environment (Sutton and Barto, 1998). The agent perceives the state of the environment and takes one of the available actions which results in a new state and a possible reward. The objective of the agent is to map states to actions in a way that maximizes the discounted future reward. A value function is assigned to each state to store the estimated discounted cumulative reward of the state. Given a state s and a set of possible actions from s, s is mapped to an action a which leads to a state with maximum future reward (as discussed in Section 2).

Estimating the value function for RL environments with a discrete and small set of states is trivial; a table with each cell representing a state of the environment can be used to store the value functions of all the states. In the case of complex RL environments with continuous states, value functions cannot be learned for every state as only a few states are visited during training and there are too many states to assign a separate cell for each state. In these environments, a function approximation method can be used to estimate the value functions by splitting the state space into groups (features) of states (the number of groups is considerably less than the number of states) and using a common value function for all of the states in the same group. To update the value function of an observed state s, the update rule is applied to all the features that contain s. As a result, the other states which are present in these features are also affected by the update. The range of this effect is proportional to the number of common features between a state and s. Different kinds of function approximation methods have been proposed to generalize a state space; one of them is tile coding (Sutton and Barto, 1998), a function approximation method that uses grid-like structures to represent the features of a state space. The grid is chosen such that these features, when grouped, should represent the entire state space effectively. Each group of the features is known as a tiling and each feature in a tiling is known as a tile (Sutton and Barto, 1998). In single tiling only a single group (tiling) of features are used to represent the entire state space, whereas in multiple tilings more than single group (tilings) of features are used to represent the entire state space. In Figure 1.1, a two dimensional state space is represented using single tiling method and multiple tilings method, and the tiles which contain state (4,4) are highlighted in red color. Sutton and Barto, in their work proved that multiple tilings method is an efficient function approximator. Though the tiling method proposed by

Sutton and Barto has worked well for small number of dimensions where tiling is applied to each dimension individually, it was not fully applied. In this work, we implemented different adaptive tile coding methods for single tiling in a RL environment.



(a)

(b)

Figure 1.1: Tile representation in single tiling and multiple tilings methods for a two dimensional state space. The range of X is -1 to 11 and the range of Y is -5 to 5. The number of a tile is represented inside the corresponding cell and the range of tile is represented below the corresponding cell. The tiles containing the state (4,4) is highlighted in red color. (a) Representation of the states along X and Y dimensions using single tiling. (b) Representation of the states along X and Y dimension using two tilings.

1.1 Thesis Statement

In this thesis we propose to evaluate different tiling methods for generalizing the value functions of a state space. In each of these tiling methods, we start with a small number of tiles and gradually divide the tiles based on how well the learner is performing in the environment. Each tiling method follows a unique policy for selecting the target points to split. The polices used are based on random generator, the number of states represented by features, the frequencies of observed features, and the difference between the deviations of predicted value functions from Monte Carlo estimates for observed states in each tile. We used these methods to train and test RL agents in different RL environments using the RL-Glue experiment tool (discussed in Chapter 4). The performance of each RL agent is evaluated and then compared with the RL agents developed using other adaptive tile coding methods and multiple tilings.

1.2 Overview

The details related to this thesis are presented in the following order. Chapter 2 presents the background for this thesis with a detailed description of Reinforcement Learning, tile coding and other generalization methods. In Chapter 3, a detailed description of the algorithms used for implementing all the proposed adaptive tile coding methods is provided. Chapter 4 presents the methodology and the environments used to conduct the experiments. In Chapter 5, the results obtained from the experiments are provided, and the efficiencies of the proposed tile coding methods are evaluated using these results. Chapter 6 discusses the related work and the possible future work in adaptive tile coding methods. In Chapter 7, summary of the thesis and the conclusions are provided.

CHAPTER 2

Background

This chapter provides background material for this thesis. The first section gives an introduction to Reinforcement Learning, an interaction-based machine learning technique. This technique guides an agent to learn what actions to take at each perceived state in an environment to maximize the cumulative reward. The second section discusses different generalization methods that are used in complex RL environments to generalize the value functions. The third section describes in detail a generalization method called Tiling.

2.1 Reinforcement Learning

Reinforcement Learning is a goal-directed and interactive learning mechanism wherein an agent interacts with an environment to learn what action to take given a state of the environment (Sutton and Barto, 1998). The agent perceives the state of the environment and takes one of the available actions which results in a new state and a possible reward. The objective of the agent is to map states to actions in a way that maximizes the discounted future reward. The process of mapping states to actions is called learning. The learning mechanism for an agent in RL is different from other computational approaches such as supervised learning. A supervised learning agent uses a set of examples provided by the supervisor to learn an optimal policy, whereas an RL agent uses the rewards obtained to learn an optimal policy which is based on future rewards rather than immediate rewards (Sutton and Barto, 1998).

RL uses a framework of states, actions and rewards to depict the interactions between an agent and an environment. Figure 2.1 represents the RL framework which contains an agent and an environment. The learner, which gets to select an action, is called the agent. Everything in the system except the learner is called the environment (Sutton and Barto, 1998). The role of an agent is to observe the current state at the current time-step and react accordingly by selecting an action. The role of an environment is to update the state of the system using the selected action and to return a reward, which may be 0, to the agent for selecting the action. In general, this process continues until the agent reaches a goal state.

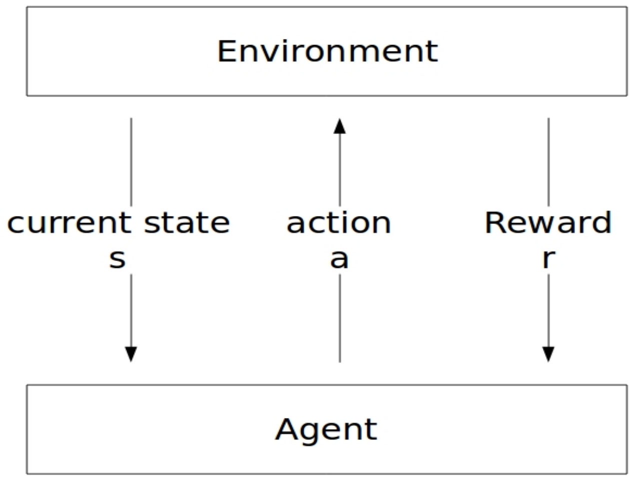


Figure 2.1: Reinforcement Learning framework: the agent perceives the current state (s) of the environment and performs an action (a). The selected action is returned to the environment, which generates a new state using a and returns a reward (r) to the agent.

In RL it is the responsibility of the agent to determine how to reach the goal state in an efficient way. The future states that an agent encounters depend on the actions previously taken by the agent. To help the agent in choosing the correct actions the environment gives feedback for each selected action in the form of a reward. If a chosen action can help in reaching the goal state in an efficient way, then the environment will return high rewards to the agent. Otherwise, it will punish the agent with low or negative rewards. To understand more about how rewards are associated with actions, a sample Grid-world problem is presented in Figure 2.2. The entire Grid-world environment is represented using a 3x3 table where each cell represents a state within the environment. The goal of an agent is to learn how to reach the goal state from the start state in an efficient way. The start state can be any state other than a mine state and the goal state. In a given state, each possible action is represented using an arrow pointing towards the resulting state for that action. The reward for each action is given next to the arrow. The value of a reward for an action depends on the resulting state. The actions which lead to the goal state receive 10 as the reward whereas the actions which lead to a mine state receive -10 as the reward; the rest of the actions receive 0 as the reward.

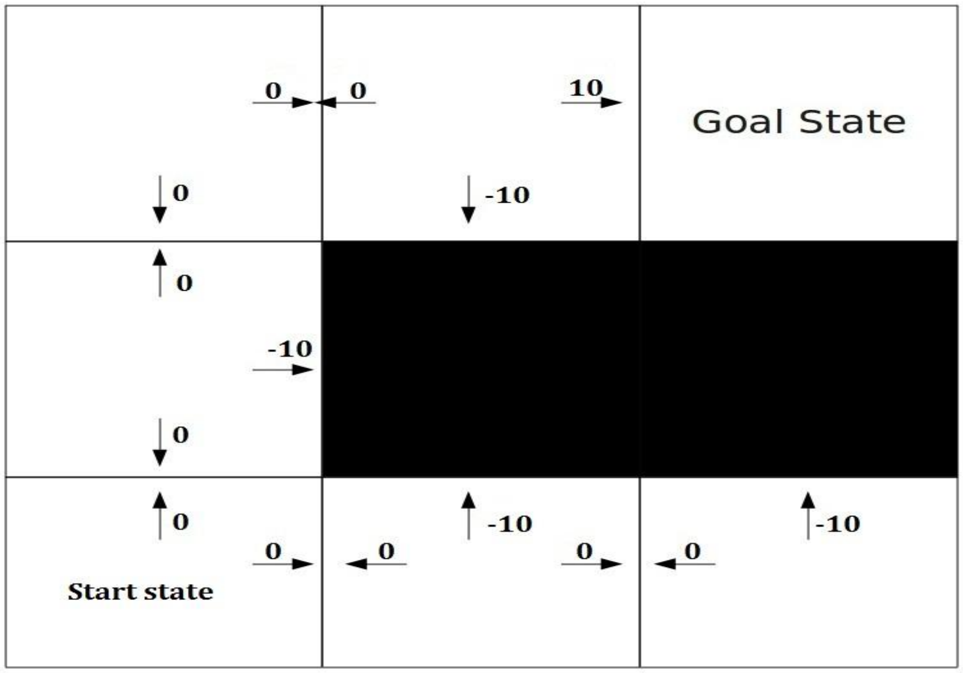


Figure 2.2: The Grid-world problem represented as RL problem: Each cell in the table represents a state in the Grid-world. Arrows represent the possible actions and the number next to them represents the rewards. Black cells represent mines.

A *policy* (π) is used to define the behavior of an agent (Sutton and Barto, 1998). It is a function that maps each state $s \in S$ and action $a \in A(s)$ to the probability ($\pi(s, a)$) of taking a when in s, where S contains all the states in an environment and $A(s)$ contains all possible actions in s. In general, multiple policies are possible in an environment and it is the responsibility of an agent to find a policy which is optimal in the sense of maximizing future cumulative reward.

An agent, while choosing an action, has to consider not only the immediate rewards but also the future rewards to maximize the *cumulative return* (Sutton and Barto, 1998):

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_T$$

where *t* is the current time step and *T* is the final time step (when the agent reaches the goal state or maximum number of steps allowed). The above function works for episodic tasks but not for continuing tasks, as the value of *T* is infinite in the latter case. A *discounted* cumulative return (Sutton and Barto, 1998) defined as:

$$\begin{aligned} R_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{T-1} r_T \\ &= \sum_{i=0}^{\infty} \gamma^i r_{t+i+1} \begin{cases} 0 \leq \gamma \leq 1 \text{ for episodic tasks} \\ 0 \leq \gamma < 1 \text{ for continuing tasks} \end{cases} \end{aligned}$$

is used to generalize cumulative return for all tasks by introducing a constant called discount factor (γ). The value of γ decides whether the future rewards are to be considered or not. If the value of γ is set to 0, future rewards are ignored while choosing an action. If the value of γ is set to 1, future rewards are as important as immediate reward while choosing an action. Generally in RL we chose a value of $\gamma > 0$ and < 1 to incorporate future rewards to push the agent to achieve those rewards sooner. The value of γ can not be set to 1 in continuous tasks as there is no final step in continuous tasks.

The *value function* (Sutton and Barto, 1998) of a state is defined as the expected cumulative reward from starting at the state and following a particular policy in choosing the future actions. The *value function* for a state s if an agent follows a policy π is:

$$V^{\pi}(s) = E_{\pi}\{R_t | s_t = s\}$$

A policy π is considered as better than or equal to policy π' if the value function of π is better than or equal to π' for all states:

$$V^{\pi}(s) \geq V^{\pi'}(s)\ \forall s \in S$$

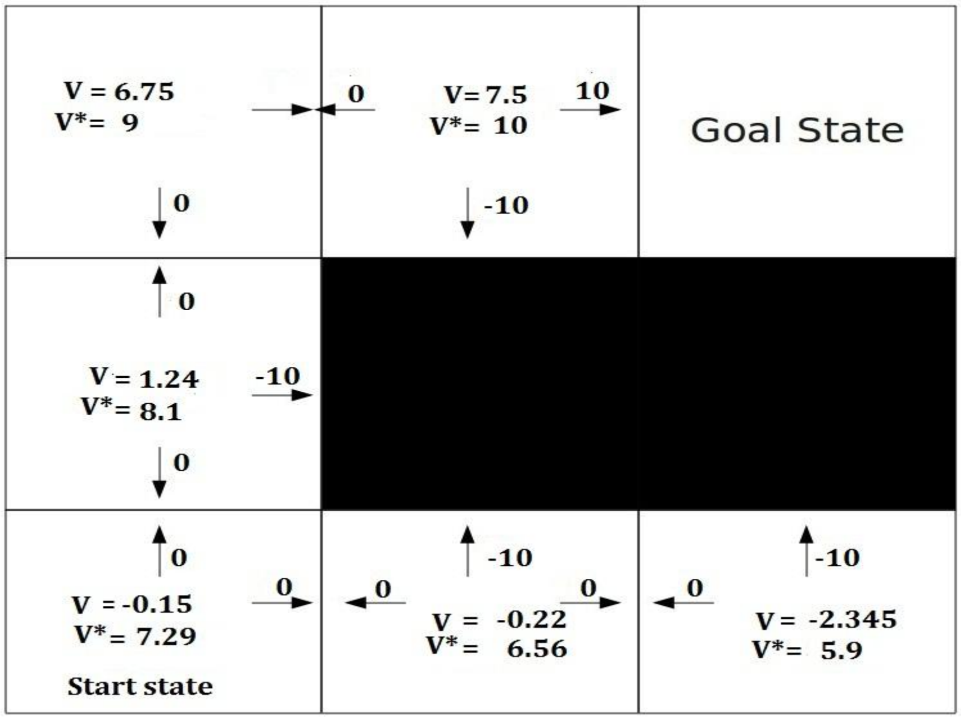


Figure 2.3: The state values in the Grid-world problem after following a policy with equal probability for all state-action pairs: V is the state value for fixed trails of a policy where each action in a state has same probability. V* is the state value for an optimal policy.

A policy whose value function is better than or equal to the value functions of all other policies at all possible states is known as an *optimal policy* (V^*) (Sutton and Barto, 1998):

$$V^*(s) = \max_{\pi} V^{\pi}(s) \quad \forall s \in S$$

A policy with the same probability for all actions was used for the above-described Grid-world problem and the results obtained are shown in Figure 2.3. The value of each state is obtained by taking an average of the state values obtained from following a fixed set of paths. The value of V* represent the optimal state values in each state.

An *action-value function* (Sutton and Barto, 1998) for an action a in a state s following a policy π is:

$$Q^{\pi}(s,a) = E_{\pi}\{R_t | s_t = s, a_t = a\}$$

The values of the action-value function are also known as *Q-values*.
An *optimal action-value function* (Sutton and Barto, 1998) (Q^*) is defined as:

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) \quad \forall s \in S \; and \; \forall a \in A(s)$$

The RL tasks are mainly divided into two types: *episodic* tasks and *continuing tasks* (Sutton and Barto, 1998). If the agent-environment interactions in a RL task naturally consist of sub-sequences, like in maze and mountain car, the RL task is considered episodic. If the agent-environment interactions in a RL task do not consist of sub-sequences, like in real time robot application, the RL task is considered continuing. According to the above given equations, to update the value functions V^{π} and Q^{π} for a policy π, an agent has to wait until the end of an episode and has to keep a record of all the rewards obtained and all the states visited. In the case of episodic tasks, updating value functions V^{π} and Q^{π} using the above equations is complex but possible, as the length of an episode is finite. But in the case of continuing tasks, updating value functions V^{π} and Q^{π} using the above equations is not feasible as an episode never ends in the continuing tasks. An alternate update rule which works fine with both continuing and episodic tasks was proposed by Bellman. According to *Bellman update rule*, the value function for an observed state-action pair following policy π, is updated in the same step by using the value function of the following state-action pair:

$$V_t^{\pi}(s) = E_{\pi}\{r_{t+1} + \gamma\big(V_{t-1}^{\pi}(s')\big)\}$$
$$Q_t^{\pi}(s,a) = E_{\pi}\{r_{t+1} + \gamma\big(Q_{t-1}^{\pi}(s',a')\big)\}$$

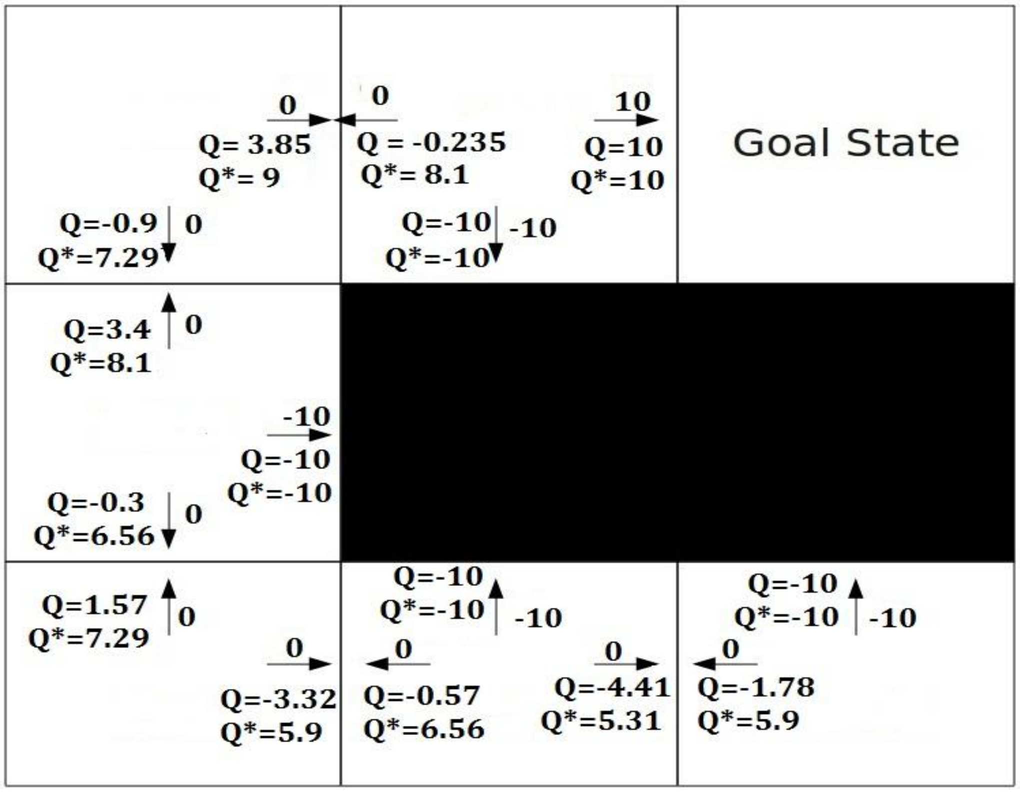


Figure 2.4: The state-action values in the Grid-world problem: Q is the state-action value for fixed trails of a policy where each action in a state has same probability. Q* is the state-action value for an optimal policy.

In Bellman update rule, the state value function of an observed state s is updated towards the sum of the immediate reward r_{t+1} and the value function $\big(V_{t-1}^{\pi}(s')\big)$ of the following state s'. Similarly, the action-value function of an observed state-action pair (s,a) is updated towards the sum of the immediate reward r_{t+1} and the action-value function $\left(Q_{t-1}^{\pi}(s', a')\right)$ of the following state-action pair (s', a').

A policy with equal probability for all state-action pairs was used to estimate the action-values functions for the above-described Grid-world problem. The obtained estimated values are shown in Figure 2.4. The Q-value in each state is obtained by taking an average of the state-action values obtained from following a fixed set of paths. The Q* in each state represents the optimal state-action values in the state.

```
1. initialize $Q(s, a)$ to some arbitrary value $\forall s \in S, a \in A(s)$
2. initialize $s$ and $a$
3. repeat (steps in an episode):
       take action $a$, observe the new state $s'$ and the reward $r$
       choose an action $a'$ in state $s'$ either stochastically or using the Q-policy
       find the difference $\delta$ between a $target$ and $Q(s, a)$, where $target$ is
                                 $r + \gamma Q(s', a')$
       increment $Q(s, a)$ by $\alpha\delta$ $\forall s \in S$ and $\forall a \in A$, where $\alpha$ is a step-rate
       update $s$ with $s'$ and $a$ with $a'$
   until $s$ is a terminal state
4. go to step 2
```

Table 2.1: The steps followed in basic Sarsa algorithm.

An agent can use the above-described *optimal state-action* values to find an optimal policy. Different algorithms are available to find the optimal Q-values for the state-action pairs in an environment. In this work I have used a modified form of the Sarsa algorithm (Sutton and Barto, 1998) to find the optimal Q-values. The $Sarsa$ algorithm is a *temporal difference* method, used to approximate the value functions of a state space using the immediate reward and the current approximated value function. It initializes all the Q-values to a user defined value and repeatedly updates the Q-values until a reasonable estimate of the Q-values is reached (A point where all the Q-values are near optimal). A brief description of the steps followed in implementing the basic form of Sarsa algorithm is given in the Table 2.1.

At the start of an experiment, the Q-values $Q(s, a)$ for all the state-action pairs are initialized to some user defined value. At the start of each episode, s and a are initialized randomly to help the agent to visit different state-action pairs. The agent perceives the current state s (initialized by environment) and takes the action a either stochastically or using a Q-policy:

$$a = max_{a'} Q(s, a') \quad \forall a' \in A(s)$$

A constant ε is used to decide if an action has to be taken stochastically or using the Q-policy. If the value of ε is zero, an action is taken using the Q-policy. If the value of ε is non-zero, an action is taken stochastically with a probability ε and using the Q-policy with a probability $1 - \varepsilon$. After taking the action, the environment updates the current state to the resulting state s' obtained by taking the action a. The agent again has to choose an action a' in the current state s'. The agent has to update the values of observed state-action pairs after each step, to improve the accuracy of the estimated value function. As discussed before, Bellman update rule is used to update the value function of an observed state-action pair. The agent updates $Q(s, a)$ to minimize the difference between the Q-value of resulting state action-pair and $Q(s, a)$:

$$\delta = r + \gamma Q(s', a') - Q(s, a)$$
$$Q(s, a) = Q(s, a) + \alpha\delta$$

where α is a learning parameter and γ is a discount factor. The steps in an episode are repeated until a terminal state is reached, at which point a new episode is started at step 2.

The Sarsa algorithm only considers the reward obtained and the discounted Q-value of the current state to assess the Q-value of the previous state. It is also known as *1-step* backup. In this way, an agent does not have to wait until the end of an episode to update the Q-values of visited states. The efficiency of this approach depends on the accuracy of the Q-values. At the start of training, the Q-values for all the state-action pairs in a state space are filled arbitrarily (generally as 0); updates based on these Q-values are not reliable and will result in a poor training. It is also a proven fact that learning rate of 1-step backup is slow. An alternate solution is to use a *Monte Carlo* method (Sutton and Barto, 1998) at the start of training. A Monte Carlo method waits until it receives a final cumulative reward R before updating the Q-values. The discounted value of R is used to update the Q-values of all the state-action pairs visited in that episode. This way the updates to Q-values are reliable, as R is an observed result. Though the estimates of this method are more reliable, implementing these methods for continuing tasks is not feasible because of the reasons which are explained before.

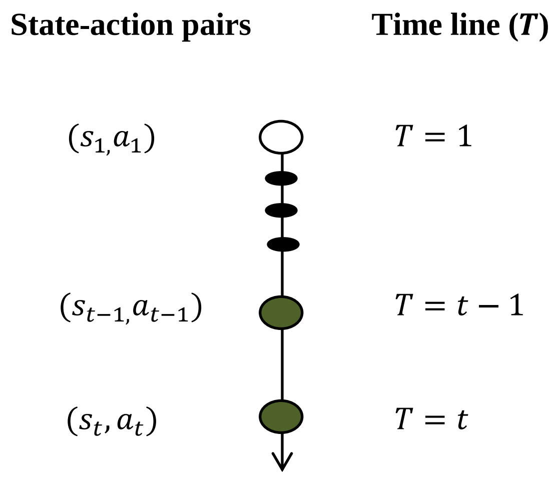


Figure 2.5: An outline of the two-step backup. In the State-action pairs column, the observed state-action pairs at time step T=1, t-1 and t are shown. The state-action pairs which correspond to the green colored ellipses are eligible for a Q-value update at time step T=t. Under the Time line column, the sequence of time step values until t is shown.

It is verified that the most accurate Q-values are not approximated by using either TD method or Monte Carlo method, but by using a generalized form of these two methods called an *n-step* method (Sutton and Barto, 1998). In the n-step method, not only the value function for current observed state-action pair is adjusted towards the target, but also the value functions for the previous n state-action pairs are adjusted towards the target; hence it is also known as n-step backup. In Figure 2.5, the outline of the two-step backup at a time step t is given. The eligible state-action pairs for which value functions are updated at time step t are $(s_{t-1,} a_{t-1})$ and (s_t, a_t). In case of 1-step backup only (s_t, a_t) is updated at time step t.

The value of n at which the approximated value functions are most accurate is not same for all the environments. Instead of using environment specific n value, a better alternative is to take the average of all possible n-step returns, by assigning weights to each n-step return, and making sure that the sum of the weights is equal to 1. Here, the value functions of all the state-actions pairs are adjusted towards target by variable amounts. The amount of adjustment depends on how close a state-action pair is temporally to the current state-action pair. This method can be implemented by introducing *eligibility traces* (Sutton and Barto, 1998) in the Sarsa algorithm; the resulting algorithm is called as the Sarsa (λ) algorithm. The Sarsa (λ) algorithm combines the efficiency of TD learning and the reliability of Monte Carlo method. In this method, an observed reward is used not only to update the Q-value of the current state but also to update the Q-values of the states which lead to the current state. From an implementation point of view, at any time step, the Q-values of all the states which are most recently observed are also updated along with the Q-value of the current state. A variable associated with each state is used to store the temporal closeness of the states with the current state. Eligibility traces are an array consisting of the above defined variables for all the states in a state space. At the start of training, eligibility trace value $e(s, a)$ for each state-action pair is initialized to 0:

$$e(s, a) = 0 \quad \forall s \in S, \forall a \in A(s)$$

At the start of an episode, the value of eligibility trace for the most recent observed sate-action pair is incremented by 1, as it is temporally closest to the current state and is most responsible for reaching the current state:

$$e(s,a) = e(s,a) + 1$$

where (s,a) is the most recent observed state-action pair.
At the end of each step during an episode, the values of eligibility traces for all the state-action pairs are reduced by a constant factor to indicate that the credit to award for all previously visited state-action pairs for reaching the current state decreases with every time step:

$$e(s,a) = \gamma\lambda e(s,a)\ \forall s \in S, \forall a \in A(s)$$

where λ is a *trace-decay parameter* and γ is a discount factor.

The update rule for Q-value using eligibility traces is given below:

$$\delta = r + \gamma Q(s',a') - Q(s,a)$$
$$e(s,a) = e(s,a) + 1$$
$$Q(s,a) = Q(s,a) + \alpha\delta e(s,a)\ \ \forall s \in S, \forall a \in A(s)$$
$$e(s,a) = \gamma\lambda e(s,a)\ \forall s \in S, \forall a \in A(s)$$

where $Q(s',a')$ is the maximum estimated action-value function in the current state s, $Q(s,a)$ is the value function of the previous state-action pair that agent has chosen.

In the above equation, using the eligibility traces to update the Q-values of the state-action pairs allows to reward all those state-action pairs which lead to the current state. The amount of credit given to a state-action pair depends on how close it is temporally to the current state. This general method is called as the Sarsa (λ) algorithm and the steps followed in this algorithm are given in Table 2.2. It is evident from the table that most of the steps in Sarsa (λ) are same as the Sarsa algorithm except for updating the Q-values of the state-action pairs. Here, not just the value function of the previously observed state-action pair, but the value functions of all the state-action pairs which are observed during the previous steps of the episode are adjusted towards the current target by variable amount. The amount by which the value function of a state-action pair is adjusted depends on the corresponding eligibility trace associated with the state-action pair. The values of eligibility traces for recently visited state-action pairs are relatively higher than the rest of the state-action pairs.

```
1. initialize $Q(s,a)$ to some arbitrary value $\forall s \in S, a \in A(s)$
2. initialize $e(s,a)$ to 0 $\forall s \in S, a \in A(s)$ , initialize $s$ and $a$
3. repeat (steps in an episode):
       take the action $a$, observe a new state $s'$ and a reward $r$
       choose an action $a'$ in state $s'$ either stochastically or using the Q-policy
       increment the value of $e(s,a)$ by 1
       update target to $r + \gamma Q(s',a')$
       find the difference $\delta$ between target and $Q(s,a)$
       increment $Q(s,a)$ by $\alpha\delta e(s,a)$ $\forall s \in S, \forall a \in A$
       decay $e(s,a)$ by a factor of $\forall s \in S, \forall a \in A$
       update $s$ with $s'$ and $a$ with $a'$
   until $s'$ is a terminal state
4. go to step 2
```

Table 2.2: The steps followed in the Sarsa (λ) algorithm.

The learning rate of an agent which uses eligibility traces to estimate the value functions is more when compared to the learning rate of an agent which does not use eligibility traces to estimate the value functions. To understand more about how eligibility traces affect the learning rate of an agent, the Grid-world problem discussed in section 2.3 is considered here. Initially all the Q-values of state space are set to 0. The estimated Q-values of a state space by Sarsa and Sarsa(λ) algorithms, after an agent completed the path consisting of states 1, 2, 3, 5, 6 and Goal, are given in Figure 2.6 and Figure 2.7.

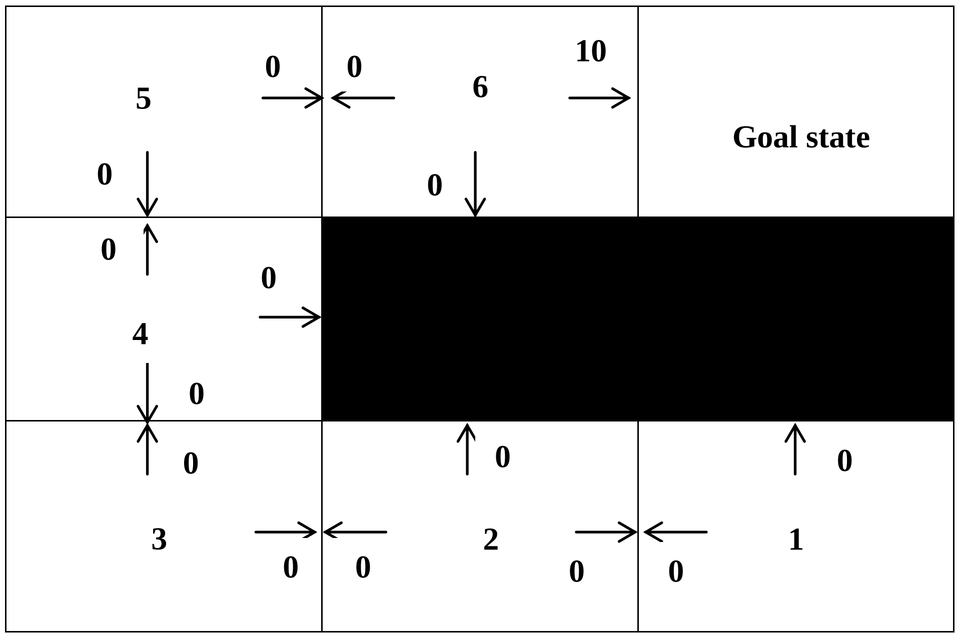


Figure 2.6: The estimated Q-values by the Sarsa algorithm after first episode for the Grid-world problem discussed in section 2.2.

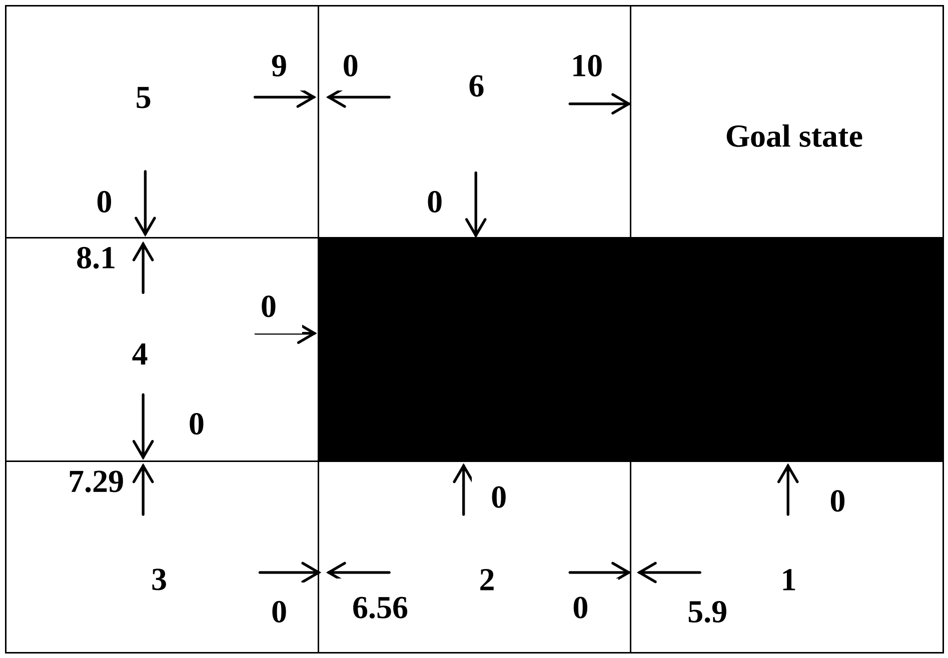
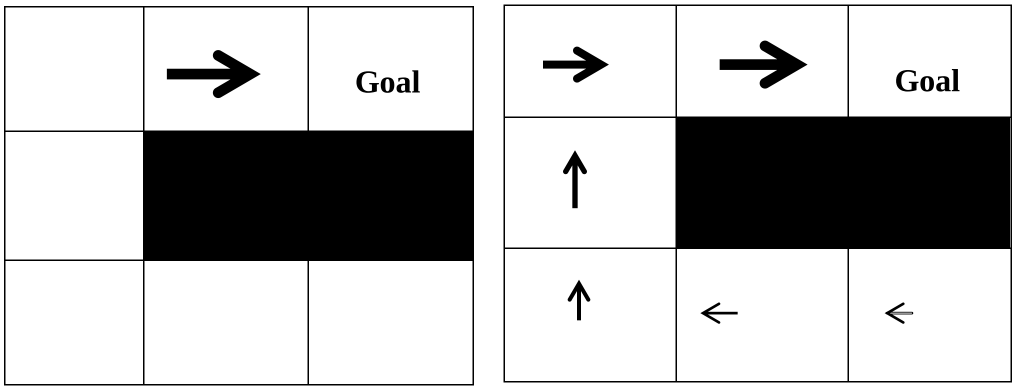


Figure 2.7: The estimated Q-values by the Sarsa (λ)algorithm, after first episode for the Grid-world problem discussed in section 2.2.

The numbers at the center of each cell in both the tables represent the states. In Figure 2.8, the effect of the completed episode in the state space is shown for both the algorithms, and it is clear from the figure that by using eligibility traces the learning rate is higher for Sarsa(λ) algorithm when compared to the Sarsa algorithm. In this work, all the RL environments that are used to evaluate the performance of various agents are episodic; hence details about applying Monte Carlo methods to continuing tasks are not provided here. In all the experiments, during training, for the first few episodes Q-values are updated using Monte Carlo method and for the rest of the episodes Q-values are updated using Bellman update rule and eligibility traces.



(a) **(b)**

Figure 2.8: The graphical representation of the Q-values from Figures 2.6 and 2.7.
(a) Representation of knowledge gained using the Sarsa algorithm.
(b) Representation of knowledge gained using the Sarsa (λ) algorithm.

The above-described Sarsa (λ) algorithm works fine for environments with small and discrete state space, but if the state space is large, table cannot be used to represent the state values and the state-action values. In such cases, generalization methods are followed to represent the Q-values in a state space. These generalization methods are discussed in detail in the next section.

2.2 Generalization Methods

The representation of value functions for an RL environment comprising discrete and limited number of states is simple and straightforward; a table can be used to represent the values with a row for each state and a column for each action. But in general, an RL environment contains a large state space. It is not feasible to represent value functions for these state spaces using a table**.** Even if a large table is used to represent the state space, the value functions in most of the cells are not updated for the reason that the number of states visited is far less than total number of states present in the environment; hence an agent has to generalize the value function obtained from a limited state space to the entire state space.

Several generalization methods are available to approximate the value functions for a large state space. In this thesis I have used a *function approximation* method (Sutton and Barto, 1998) to generalize the value functions of a state space. The function approximation method is used to generalize a target function from a set of examples. More about the implementation of this method will be discussed in the following sections.

2.2.1 Basic Idea of Generalization Methods

In general, the difference in feature values of the states which are close enough in a large state space is minimal. A generalization method exploits this fact and uses the feature values of visited states to approximate the feature values of nearby states. This allows an RL agent to train efficiently on a small set of states and apply this knowledge to a larger set. The problem with this approach is the boundary of generalization. If the boundary is too small then only the value functions of those states which are really close enough to a visited state are approximated. It improves the accuracy of the results, but it affects the extent of generalization. If the boundary is too big then the extent of generalization is improved, but the states which are not close enough might end in the same group. Different generalization methods use different approaches to solve the boundary problems. In the following sections, function approximation, a common generalization approach and various other generalization methods which are obtained by extending function approximation method are discussed in detail.

2.2.2 Function Approximation

A *function approximation* is a supervised learning technique which approximates a target function from a set of examples (Sutton and Barto, 1998). For example, given a set of examples of the form $\left(x, f(x)\right)$, the function approximation has to approximate the target function f. In this thesis, I am using the function approximation method to approximate a parameterized Q-value function Q for the state-actions pairs in a state space S using a limited set of Q-values $Q(s,a)$, where $s \in S$ and $a \in A(s)$, estimated for observed state-actions pairs. In this context, x is an observed state action pair (s,a) and $f(x)$ is a Q-value $Q(s,a)$ for action a taken in state s. The generalized function Q has a parameter $\vec{\theta}$, a variable vector with each component representing a group of states in the state space. The values of the components in the variable vector are used to compute the Q-values of all the state-action pairs in a state space. The above specified parameter vector can be implemented in different ways. In this thesis I have used a linear model of gradient descent method to implement the parameter vector.

In an environment with a discrete and small state space, a table is used to represent the entire state space. Each cell of the table represents a unique state in the state space, and it is easy to keep track of value functions for each state. If a state space is continuous then a table cannot be used to represent the states of the state space as it is not feasible to allocate separate cell for each state in the state space. Assuming a very large table is allocated for the entire state space, value functions for most of the cells in the table are not estimated, as number of states visited during training is far less than the total number of states in the state space. A function approximation method can solve the above two problems of representing a continuous state space. Since a parameter vector $\vec{\theta}$ represents an entire state space using far less number of vector components, representation of state space gets simple. Since each component of $\vec{\theta}$ represents a group of states in the state space; to estimate the value functions for the entire state space, we just have to make sure that reasonable number of states in each component of $\vec{\theta}$ is visited during training. The value function of a state is obtained using the values of the vector components in which the state is present. As the number of components in a vector is far less than the actual count of states, it is possible to visit reasonable number of states in each component of $\vec{\theta}$.

To understand more about how parameter vector $\vec{\theta}$ generalize value functions, the state space of a mountain car is represented in the form of a generalized function below. The state space of mountain car consists of two dimensions: the position of car (X), and the velocity of the car (Y). The range of X is from -1.2 to 0.6 and range of Y is from -0.07 to 0.07; and both X, Y are continuous. Each component in $\vec{\theta}$ is called a feature and each feature contains a set of states from the state space. As $\vec{\theta}$ represents all the states of an environment, all values of X and Y must be present in at least one component/feature of $\vec{\theta}$. Assuming that ϕ_i represents the i^{th} component of $\vec{\theta}$, one way to generalize the features X and Y is given below:

$$\vec{\theta} = (\phi_1, \phi_2, \phi_3, \phi_4 \ldots\ldots, \phi_9)$$

ϕ_1 = [-1.21, -0.9)

ϕ_2 = [-1.0, -0.3)

ϕ_3 = [-0.5, 0.1)

$\phi_4 = [0.05, 0.5)$

$\phi_5 = [0.3, 0.61)$

$\phi_6 = [-0.07, -0.03)$

$\phi_7 = [-0.05,\ 0.02)$

$$\phi_8 = [0.0,\ 0.05)$$
$$\phi_9 = [0.03, 0.071)$$

For the sake of simplicity, in the above vector, only the values of a single dimension are placed in each feature. The components $\phi_1, \phi_2, \phi_3, \phi_4,$ and ϕ_5 are used to represent the features comprising X and the components $\phi_6, \phi_7, \phi_8,$ and ϕ_9 are used to represent the features comprising Y. If an observed state has x = 0.3 and y = 0.01 then only the Q-values of components ϕ_4, ϕ_5, ϕ_7 and ϕ_8 are adjusted towards a target but the Q-values of the rest of the components are not disturbed. Updating the Q-values of ϕ_4, ϕ_5, ϕ_7 and ϕ_8 effects the value functions of states in $[0.1, 0.5)$, [0.3, 0.61) for X and $[-0.05,\ 0.02)$, [0.0, 0.05) for Y. It is important to make sure that the components of every feature in $\vec{\theta}$ are closely related. Otherwise, the accuracy of the estimated value function is affected. Various methods are available to group the states into different components of $\vec{\theta}$.

2.2.3 Gradient-Descent Methods

Gradient-Descent is a form of function approximation method which uses a column vector filled with real value components as the parameter $\vec{\theta}$ (Sutton and Barto, 1998). The number of elements n in $\vec{\theta}$ is fixed at the start of training and is a lot less than the number of states in a state space. At each time step, the difference between a target and estimated Q-value of an observed state-action pair is used to update $\vec{\theta}$ to reduce the mean standard error in approximation:

$$\vec{\theta} = \begin{pmatrix} \theta(1) \\ \theta(2) \\ : \\ : \\ \theta(n) \end{pmatrix}$$

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha(target - Q(s_t, a_t))\vec{e}_t$$

where α is a learning rate parameter, $\vec{e}_t$ is an eligibility trace vector at time *t*. The value of $target$ depends on the learning mechanism used; in a Monte Carlo method, $target$ is a discounted cumulative reward R_t, whereas in Temporal Difference method, *target* is the sum of observed reward r_t and the best Q-value $Q(s_{t+1}, a_{t+1})$ for the next state s_{t+1}. It is also required to update $\vec{e}$ at each time step to adjust the weights of $\vec{\theta}$:

$$\vec{e}_t = \gamma\lambda\vec{e}_{t-1} + \nabla_{\vec{\theta}_t} Q(s_t, a_t)$$

where γ is a discount factor, λ is an eligibility trace component, and $\nabla_{\vec{\theta}_t} Q(s_t, a_t)$is a vector derivative of Q with respect to $\vec{\theta}_t$. In this method, Q-value function is obtained by applying a differentiable function to the parameter vector$\overrightarrow{\theta_t}$. Different types of gradient-descent methods apply various differentiable functions to $\overrightarrow{\theta_t}$.

2.2.3.1 Coarse Coding

Coarse coding is a linear form of gradient-descent function approximation method (Sutton and Barto, 1998). It uses several instances of a geometrical shape G to represent the features in a state space S, where each feature consists of a subset of states in S. Each instance of G represents a feature in the state space and a component in the feature vector $\vec{\phi}$. If a set of states have some particular property in common, they are all grouped into one feature. If a state s lies within the boundaries of an instance of G then s has the feature corresponding to that instance. The *presence* of a feature in a state can be represented in different ways, but in general a binary digit is used. The number of features shared by two different states is equal to the number of instances common for both the states. The scope of generalization depends on the shape and size of an instance.

For example, in Figure 2.9 generalization along X-axis is more prominent when compared to the generalization along Y-axis; hence the rectangles with length along X-axis are used to represent the features. If **A** is observed during a training, then generalization from **A** to **C** is better when compared to generalization from **A** to **D**, as **A** has three features in common with **C** but only two with **D**. Moreover there will be no generalization from **A** to **E** as there are no common features between **A** and **E**. With large size instances, generalization is quick and broad but not accurate. With small size instances, generalization is slow and narrow but accuracy is much better. Using large numbers of moderate size instances allows generalization to be quick, broad and near optimal.

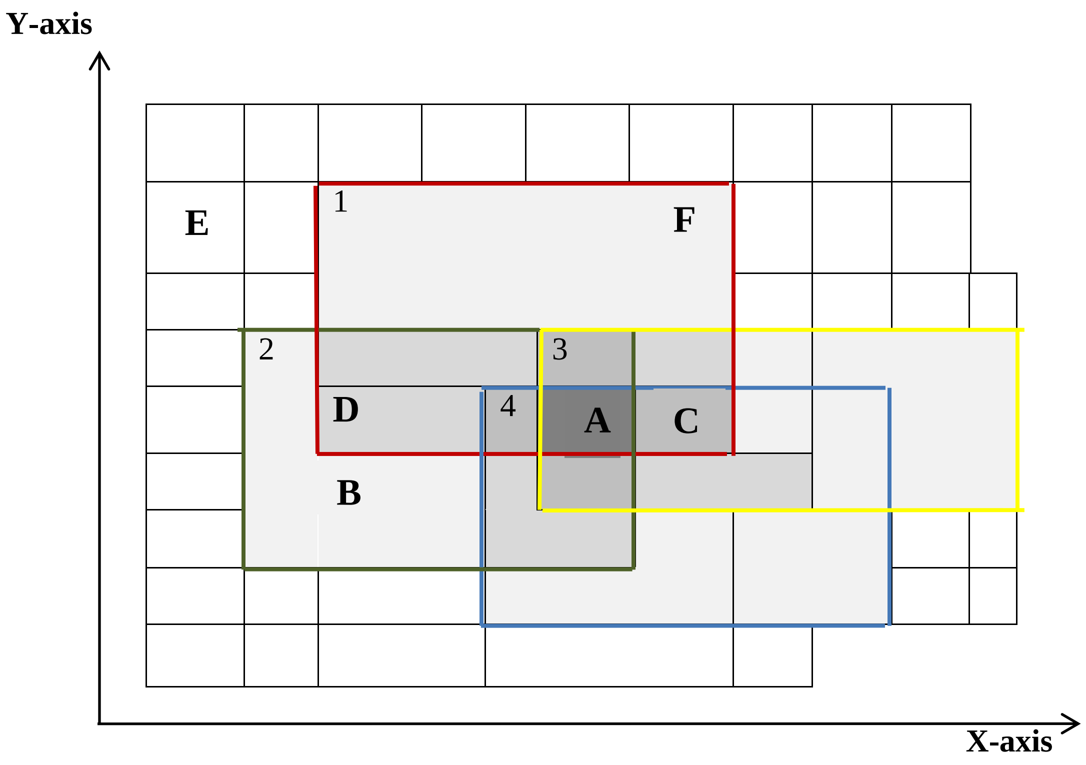


Figure 2.9: Coarse coding using rectangles: A, B, C and D are different states in a state space. The number of features shared by any two states in the state space is equal to the number of rectangles which contains both the states.

The use of binary numbers to represent the presence of a feature in a state is not an efficient way of generalization. If an observed state x and some other states y and z have only one common instance i then generalization is same for y and z as the feature value for i is set to one for both y and *z*. A better generalization method should consider how x, y and z are related with respect to i before generalizing from x to y and z. For example, in Figure 2.9, assuming state **D** was observed during training, if binary numbers are used to represent the feature presence, as **D** has only one feature in common with both **B** and **F**, while adjusting the values of features, both **B** and **F** are adjusted by same amount. It is pretty clear from the figure, state **D** is present at somewhere near the corner of the feature which contains both **D** and **F** (feature 1), whereas it is present near to the middle point of the feature which contains both **D** and **B** (feature 2). It means the commonality between **D** and **B** is more when compared to the **D** and **F**. Actually **B** should be adjusted more towards the target when compared to **F**, but it is not possible with binary features. The following method resolves this problem by not only considering the presence of a state in the feature but also the distance between the observed state and the state while adjusting the value function of the state towards the target.

2.2.3.2 Radial Basis Functions

A function whose values depend only on the distance from a particular point is known as a radial basis function. Radial basis functions are used in coarse coding method to estimate the degree of presence of a feature in a state. The degree of presence of a feature f in a state s is represented using a value between 0 and 1. If an instance i is used to represent f then the feature value depends on how far s is from the center c of i relative to the width w of i (Sutton and Barto, 1998):

$$\phi_s(i) = \exp\left(-\frac{||s-c||^2}{2w^2}\right)$$

For example, in Figure 2.9, if we consider radial basis function instead of binary numbers to estimate the presence of a feature in a state S, and assume that the width of feature 1 is w' , distance between state **D** and the center of feature 1 is c', width of feature 2 is w'' and the distance between state **D** and the center of feature 2 is c'', the resultant radial basis functions are:

$$\phi_{D1}(i) = \exp\left(-\frac{||c'||^2}{2w'^2}\right)$$

$$\phi_{D2}(i) = \exp\left(-\frac{||c''||^2}{2w''^2}\right)$$

It is evident from the figure that $c' > c''$ and $w' \approx w''$, which means the value inside exp is more negative for feature 1 when compared to feature 2. As a result, the value of $\phi_{D1}(i)$ is less than $\phi_{D2}(i)$, from which we can deduce that state **D** is closely associated with feature 2 when compared to feature 1. Since **D** is the observed state, the adjustment of value functions towards target for states in feature 2 is higher when compared to that of value functions for states in feature 1.

2.3 Tile Coding

Tile coding is an extension of the coarse coding generalization method (Sutton and Barto, 1998; Stone, Sutton, and Kuhlmann, 2005). In tile coding, a state space is generalized using a set of features which represent a partition of the state space. The set of features are grouped in such a way that each single group contains non-overlapping features that represent the entire state space. The group of non-overlapping features that represent the entire state space is also known as a *tiling*. The accuracy of the generalization in tile coding is improved with the number of tilings. Each feature in a tiling is called a *tile*, and each tile has an associated binary value to indicate whether the corresponding feature is present in the current state or not. The direction and accuracy of a generalized state space depends primarily on the shape and size of the tiles. In general, the shape of a tile affects the direction of generalization whereas the size of a tile affects the range of generalization. In Figure 2.10, single tiling is applied to generalize two different state spaces. Tiles with different shapes and sizes are used to represent the features of these state spaces. It is evident from the figure that the direction of generalization is along Y for S1, and along X for S2. In the following sections, these two tile coding methods are discussed in detail.

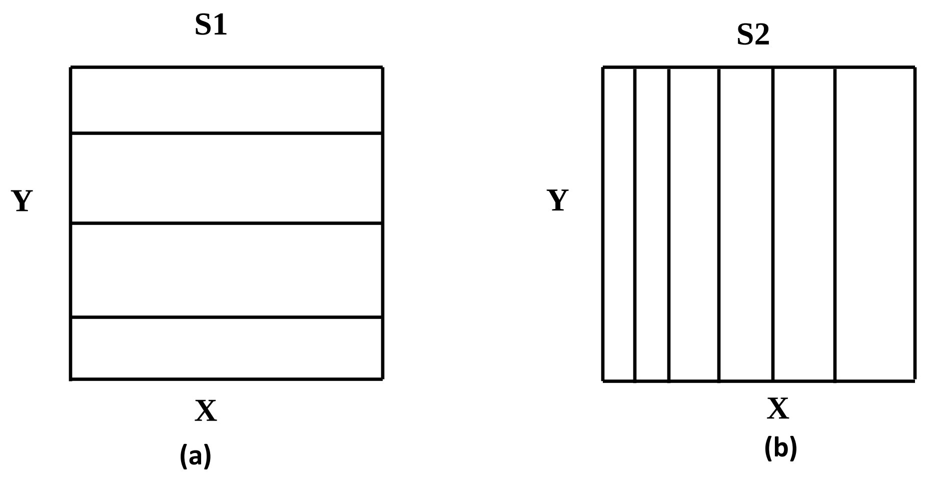


Figure 2.10: Partition of two different state spaces S1 and S2 using tile coding.
(a) S1 is generalized along Y direction.
(b) S2 is generalized along X direction.

The update rule used by a linear generalization method to estimate the value functions is:

$$Q_t(s) = \sum_{i=1}^{n} \theta_t(i)\phi_s(i)$$

where $Q_t(s)$ is an estimated value function, $\overrightarrow{\theta_t}$ is a parameter vector and $\overrightarrow{\phi_s}$ is a component vector.

In a linear generalization method which uses gradient-descent approximation to estimate value function, the gradient of $Q_t(s)$ with respect to $\overrightarrow{\theta_t}$ is used to update the value function:

$$\nabla_{\overrightarrow{\theta_t}} Q_t(s) = \overrightarrow{\phi_s}$$

Tile coding uses an extended form of the above specified gradient-descent update rule to estimate the value functions:

$$\nabla_{\overrightarrow{\theta_t}} Q_t(s) = \sum_{\phi(i)\neq 0} \theta(i)$$

since the presence of a feature is represented using a binary value associated with the corresponding component of a feature. Unlike in coarse coding, where in the number of features observed at a point depends on the current observed state, tile coding always has same number of features for any state. It simplifies the update rule, as α can be set to a constant value. The number of features present in an observed state always depends on the number of tilings used to represent the features of a state space. In the following sections, the implementation of single tiling and multiple tilings are discussed in detail. In this thesis, I have used only single tiling for implementing adaptive tile coding.

2.3.1 Single Tiling

In *single tiling*, all the features of a state space are represented using one tiling. Any state in the state space must correspond to only one of the tiles in the tiling. During an experiment, at each step, the feature that contains the observed state is selected and the value function of a corresponding tile is updated according to the rewards obtained in the following steps. When the value function of a tile is updated, the value functions of all states within the tile are equally affected as the observed state. Hence, while choosing features for a state space, states with identical value functions are place in a same tile. Since, every state in the state space corresponds to only one of the features of the tiling and only a single tiling is available, the number of the features selected at any step is a constant and is equal to one. A binary variable is associated with each tile to indicate the presence of a corresponding feature at each step. The binary variable of a tile corresponding to the observed feature is set to 1 and the binary variables of the rest of the tiles are set to 0. In case of single tiling, the above specified tile coding update rule can update only one feature at each step, as the number of tiles selected is restricted to one per tiling and the number of available tilings is one.

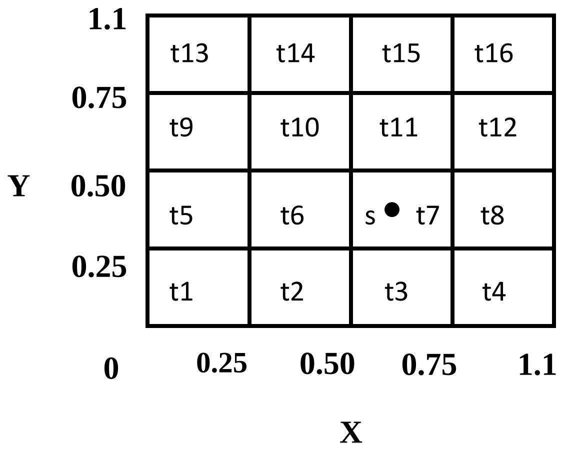


Figure 2.11: The features of the simple Grid-world problem represented using single tiling. A two-dimensional grid is used to represent the non-overlapping features, with x-axis representing X values and y-axis representing Y values. Each cell represents a tile with width and height equal to 0.25. The numbers inside each cell indicates the tile number.

To understand more about how to generalize a state space using single tiling, the simple Grid-world problem discussed earlier is revisited. The Grid-world problem contains two dimensions X and Y with values in the range 0 to 1. The state space is partitioned into multiple non-overlapping features using single tiling as shown in Figure 2.11. Each cell (t1, t2, …, t16) of the table represents a tile in the tiling. In this case, the parameter vector $\overrightarrow{\theta_t} = (t1, t2, t3, \dots\dots., t16)$ and the value of component vector $\overrightarrow{\phi_s}$ is 1 for the tiles that correspond to features which contain *s* and 0 for the rest of the tiles. At any step, an observed state correspond to only one of the 16 available tiles. For example, assume that a current observed state (represented using a dot in the above figure) s has **X** value equal to 0.59 and **Y** value equal to 0.37. To update the value function of s, the value function of t7 which contains s has to be updated. As a result of the update, the value functions of all those states which are present in t7 are also affected.

To improve the accuracy of a generalization, a state space has to be partitioned in such a way that the resulting tiles contain only closely associated states, which in return affects the range of generalization. As an alternative, another tile coding technique called multiple tiling is used to increase the range of generalization without affecting the accuracy of approximation.

2.3.2 Multiple Tilings

In *multiple tilings,* the features of a state space are represented using more than one tiling. Any state in the state space must correspond to only one of the tiles in each tiling. During an experiment, at each step, a tile that contains the observed state is selected in each tiling and the value functions of those tiles are updated according to the rewards obtained in the following steps. Unlike in the single tiling, updating the value function of tiles does not equally affect the value function of other states which correspond to the same tiles. The value functions of the states which appear in most of the active tiles are affected more when compared to the value functions of the states which appear in few of the active tiles. It improves the range of generalization without limiting the accuracy of generalization, as updating multiple tiles at the same time affects more number of states, and only the states which have more number of features in common with an observed state are adjusted more towards the target when compared to the other states which have only few features in common with an observed state. Since, every state in the state space correspond to only one of the features of a tiling, the number of the features selected at any step is a constant and is equal to the number of tilings. A binary variable is associated with each tile to indicate the presence of a corresponding feature at each step. The binary variable of the tiles corresponding to the observed features are set to 1 and the binary variables of the rest of the tiles are set to 0.

To understand more about the multiple tilings, the state space of the above discussed Grid-world problem is partitioned into multiple non-overlapping features using multiple tilings as shown in Figure 2.12. Two tilings are used to represent all the features of the state space. Each cell in the tables represents a tile in one of the two tilings. In this case, the parameter vector $\overrightarrow{\theta_t} = (t1, t2, t3, \dots \dots., t32)$ and the value of component vector $\overrightarrow{\phi_s}$ is 1 for the tiles that correspond to features which contain the current observed state and 0 for the rest of the tiles. At any step, an observed state correspond to one of the tiles in tiling 1 and tiling 2. For example, assume that a current observed state (represented using a dot in the above figure) S has **X** value equal to 0.59 and **Y** value equal to 0.37. The two tiles, one from tiling 1 and another from tiling 2, that contain s are highlighted in the figure. These value functions of these two tiles are adjusted towards the target value. In the figure, the states which correspond to both of the highlighted tiles are affected more when compared to the tiles which correspond to only one of the highlighted tiles.

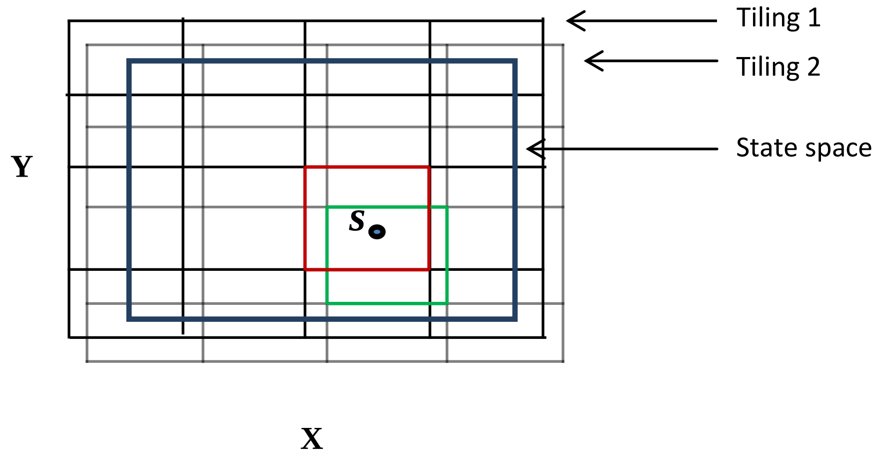


Figure 2.12: The features of a simple Grid-world problem represented using multiple tilings. Two different two-dimensional grids are used to represent the non-overlapping features, with x-axis representing X values and y-axis representing Y values. Each cell in both the tables represents a tile with width and height equal to 0.25.

Another fact to notice is the difference in α value for single tiling and multiple tilings to update the Q-values at a same step rate. The value of step-size parameter α in multiple tiling is different from that of single tiling. In single tiling, the value of α is set to x if the required step size is x. In multiple tiling, the value of α is set to x/m where m is the number of tilings if the required step size is x. The division by m is to balance the repeated update performed on the value functions of the eligible states towards the same target for m times (once for each tile in a tiling).

2.4 Review of Problem

The efficiency of tile coding method depends on how well the tiles can partition the state space into groups of similar states. Sutton and Barto, in their work, applied single tiling and multiple tilings methods to generalize different RL state spaces, and suggested that multiple tilings is an efficient method to generalize state spaces with lower-dimensions. They used a manual approach to group the states (for generalization) of an RL state space. The drawback of this approach is that user must have a thorough understanding of environment to partition the state space effectively. An alternate approach is to give the control to the RL agent to automate the process of grouping the states. This approach is also known as adaptive tile coding. Whiteson, Taylor and Stone used the Bellman update rule to implement this approach. In this thesis, we proposed different adaptive tile coding methods and evaluated their efficiencies.

CHAPTER 3

Proposed Solutions

In this chapter I will present several adaptive tile coding methods that I have used as an alternative for multiple tilings to solve RL problems. An *adaptive tile coding* method is a technique used to automate the feature defining process of tile coding (Whiteson *et al*., 2007). The first section describes the tile split concept and its advantages. In the second section the algorithm and the implementation details of different adaptive tile coding methods like *random tile coding*, *feature-based tile coding, value-based tile coding, smart tile coding* and *hand-coded tiling* are provided. In *random tile coding,* a tile is selected randomly from a set of available tiles for further division. In *feature-based tile coding*, tile division is based on the feature's size (the portion of a dimension represented by the feature). In *value-based tile coding*, tile division is based on the frequency of the observed states. In *smart tile coding,* split points selection is based on the estimated Q-values deviation. In *hand-coded tiling* split points are selected manually.

3.1 Tile Coding a State Space

In tile coding, the direction (the dimension along which generalization is prominent) and the accuracy (how closely related are the states in the same tile) of a generalized state space depend primarily on the shape and size of the tiles representing the state space. To manually partition a state space, significant knowledge of the state space is required. An alternate solution is to *automate* the partitioning, by using the knowledge gained during learning to partition the state space. This mechanism is also known as *adaptive tile coding* (Whiteson *et al.,* 2007). The steps followed in adaptive tile coding are given in Table 3.1.

In linear adaptive tile coding, different set of tiles are used to represent each feature of a state space. At the start, a minimal number of tiles (often two) are used to represent each feature in a state space. Using RL mechanism, the action-value functions of the available tiles are approximated. The approximated action-value functions of the available tiles represent the generalized action-value functions for the entire state space. As the minimal number of tiles is used, states with conflicting action-value functions might correspond to the same tile. A policy based on parameters like action-value functions and frequency of the observed states is used to select a point which separates most conflicting states. A tile which contains the point is divided into two separate tiles by splitting at the point. The action-value functions are approximated again using available tiles, and a tile which contains a point that separates most conflicting states is split at the point into two separate tiles. The process is repeated until the policy determines that splitting can no longer improve generalization or the splitting count reaches the required value.

```
1.    Choose minimal number of tiles for each feature in a state space.
2.    Repeat (tile split):
      a.    Approximate the action-value functions for all states in the state space
            using available tiles.
      b.    A policy is used to find a state which divides the states that have
            conflicting value functions.
      c.    Find the tile containing the selected state and split the tile into two tiles at
            the selected state.
      Until the policy determines splitting can no longer improve generalization.
```

Table 3.1: General steps followed in an adaptive tile coding method.

Figure 3.1 shows how an imaginary two-dimensional state space might be tiled using adaptive tile coding. Initially, each feature in the state space is represented using two tiles. Assuming that an adaptive tile coding policy has selected a tile containing feature x

equal to 0.9 for further division, tile 2 is split at 0.9 into tile 5 and tile 6. In the second pass, the tile containing feature y equal to 0.1 is selected for further division; hence tile 3 is divided at 0.1 into tile 7 and tile 8. Similarly, in the last pass, tile 4 having feature y equal to 1.2 is divided at 1.2 into tile 9 and tile 10.

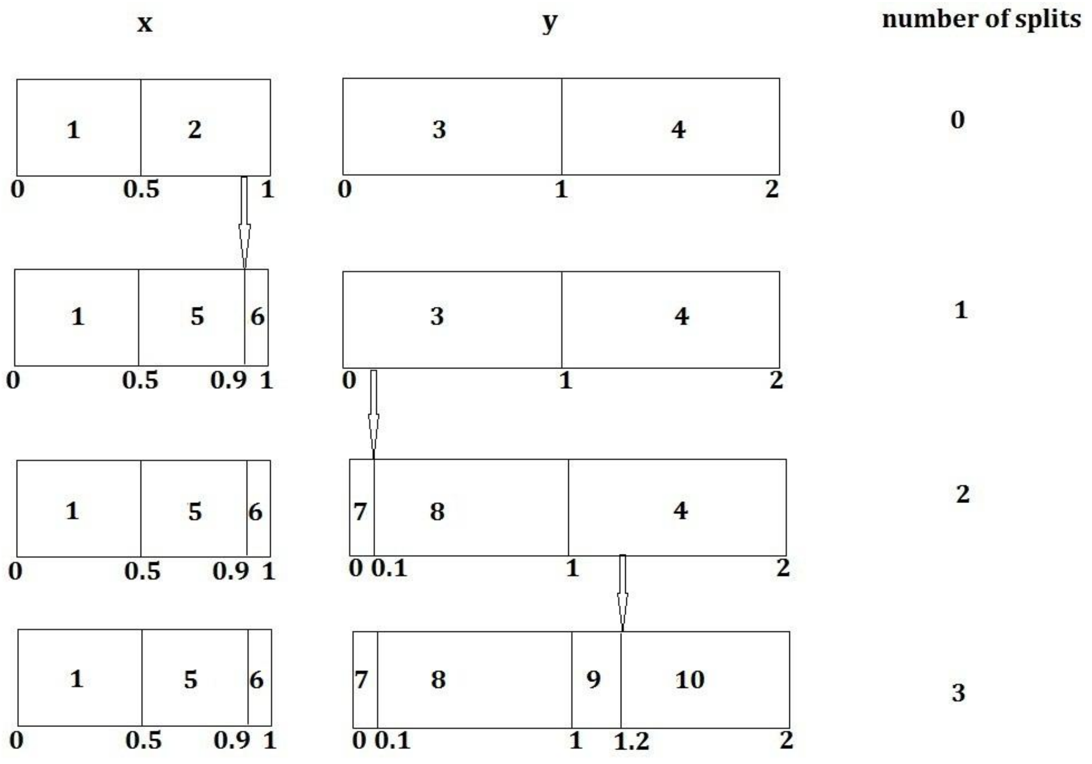


Figure 3.1: A generalized two-dimensional state space using adaptive tile coding. On the left, each cell represents a tile that generalizes a group of states specified at the bottom of the cell, for features x and y. On the right, the number of splits performed on the state space .

In adaptive tile coding, generalization at the start of learning is *broad* and not reliable, as only a few tiles are used to partition a state space. During the course of learning, by splitting tiles, generalization becomes narrow but it does not guarantee an improvement in the accuracy of generalization. The accuracy of a generalization depends on how effectively a policy partitions a state space. Different policies are used to split tiles; most of them use approximated value functions. In the paper *Adaptive tile coding using value function approximation* (Whiteson *et al*., 2007); a policy based on *Bellman error* was used for selecting the split points. In the next section, I described different policies that are used to implement adaptive tile coding.

3.2 Adaptive Tile Coding Methods

Different methods use different approaches to select a split point in adaptive tile coding. In this work, I implemented methods based on random selection, feature size, frequency of the observed states, difference between estimated Q-value deviations of states in a tile, and hand-coding. The algorithms used to implement these methods are described in the following sections. All these algorithms are derived from the Sarsa (λ) algorithm. In the next chapter, the efficiencies of these methods, obtained by evaluating the performance of RL agents developed using the methods, are provided.

3.2.1 Adaptive Tile Coding Implementation mechanism

The implementation of the following adaptive tile coding methods is a repeated process of training (exploring) an agent using available tiles, testing (exploiting) the performance of the agent, and splitting a tile selected by the specific adaptive tile coding method. The implementation of training phase and testing phase in all adaptive tile coding methods are similar, and use a modified form of the Sarsa(λ) algorithm. A generalized algorithm to control and implement the above three phases is provided in Table 3.2. All the steps of the algorithm except for splitting_method() are same for all of the following adaptive tile coding methods. The implementation of splitting_method() is unique for each adaptive tile coding method. It defines the approach used by an adaptive tile coding method to partition the tiles. In feature size based tile coding and smart tile coding, during a testing phase, additional steps are required in the Sarsa(λ) algorithm at the end of each pass. The additional steps required for these adaptive tile coding methods are discussed in separate sections.

The input to the generalized algorithm is the number of episodes (N) for exploration, the number of episodes (M) for exploitation, the maximum number of splits (P) to perform, the minimum number of tiles ($startT$) to start with, the maximum number of steps ($maxS$) to perform, the initial Q-values (Q), the function (tilingFunction) to retrieve corresponding tiles for the current state, and the values for different RL parameters. The algorithm has an exploration/training phase, an exploitation/testing phase and a splitting phase. For each split, the exploration phase runs for N episodes and the exploitation phase runs for M episodes. The exploration phase and exploitation phase use the modified form of Sarsa(λ) algorithm provided in Table 3.3.

```
runExperiment($P, N, M, startT, maxS, trainP, Q, tilingFunction, \gamma, \lambda, \alpha, \varepsilon$)
  where
     $P$ – number of splits to use
     $N$ – number of training episodes
     $M$ – number of testing episodes
     $startT$ – number of tiles to represent each dimension at the start
     $maxS, trainP, Q$, tilingFunction, $\gamma, \lambda, \alpha$ , $\varepsilon$ – parameters for runEpisode (see
                                                                     Table 3.3)
     for $p = 1$ to $P$
       for $n = 1$ to $N$
         runEpisode($maxS, true, Q, tilingFunction, \gamma, \lambda, \alpha, \varepsilon$)          // Training
       rof
       $total = 0$
       for $m = 1$ to $M$
         $R$ = runEpisode($maxS, false, Q, tilingFunction, \gamma, \lambda, \alpha, \varepsilon$) // Testing
         $total = total + R$
       rof
       if ($p < P$)
         splitting_method()          // Depends on the adaptive tile coding method
       fi
     rof
```

Table 3.2: General algorithm used to implement an adaptive tile coding method.

```
runEpisode($maxS, trainP, Q$, tilingFunction, $\gamma, \lambda, \alpha$ , ε) returns $R$
where
  maxS – max steps to take in this episode (even if goal not found)
  trainP – a flag to indicate whether the agent is training or testing
  $Q$ – a set of factors for each possible action, one for each tile
  tilingFunction – a callable function that transforms a general state description into set of tiles
  γ – a trace decay factor for the traces
  λ – discount factor for calculating discounted reward
  α – learning rate , ε – value used for ε-greedy learning

  $s_{curr} =$ initialState()                         // Current state vector
  $t_{curr} = null,\ a_{curr} = null,\ e(t) = 0\ \ \forall\, t \in$ Tilings
  $step = 0$                                          // Number of steps so far
  $valueCount(t) = 0\ \ \ \forall\, t \in$ Tilings        // Used in value-based tile coding only
  repeat
     $step = step + 1$
     $s_{prev} = s_{curr},\ t_{prev} = t_{curr},\ a_{prev} = a_{curr}$
     $t_{curr} =$ tilingFunction($s_{curr}$)          // $t$ is the tile vector for the current state
     calculate $Q_a(t_{curr})\ \ \forall a \in$ Actions
     $e(t_{curr})$ ++
     if ($trainP == false$)  $valueCount(t_{curr})$++       // value-based tile coding only
     if ($trainP == true$)  ε-greedyPolicy()             // $\varepsilon - greedy\ select\ action$
     else    $a_{curr} = max_a\, Q_a(t_{curr})\ \ \ \forall a \in$ Actions  // action with max Q-value
     fi
     $< s_{curr}, r_{curr} > =$ performAction($a_{curr}$)
     $R = r_{curr} + \lambda R$

     if ($a_{prev}$ != null)
       $\delta = R - Q_{a_{prev}}(t_{prev})$
       $Q_a(t) = Q_a(t) + \alpha\, \delta\, e(t)\ \ \ \forall a \in$ Actions, $\forall t \in$ Tilings
       $e(t) = \gamma\, \lambda\, e(t)\ \ \ \forall t \in$ Tilings
     fi
  until (($step \geq$ maxS)|  |($s_{curr} \in$ GoalStates))
  $R = r_{curr} + \lambda R$
  $\delta = R - Q_{a_{prev}}(t_{prev})$
  $Q_a(t) = Q_a(t) + \alpha\, \delta\, e(t)\ \ \forall a \in$ Actions, $\forall t \in$ Tilings // Update for final action
```

Table 3.3: A modified form of the Sarsa(λ) algorithm to generalize a state space using tile coding.

```
ε-greedyPolicy() returns $a_{curr}$
    if (randomProb() < ε)    // randProb returns a random value between 0 and 1
        $a_{curr}$ = randomAction(Actions) // returns a random action
    else
        $a_{curr} = max_a Q_a(t_{curr})$    $\forall a \in$ Actions
    fi
```

Table 3.4: Algorithm to implement the ε-greedy policy. It is used to select an action during training.

The implementation of the Sarsa(λ) algorithm is discussed in Chapter 2. At the start of an experiment, the starting state of an agent is initialized randomly, with every possible state having non-zero probability**.** The tiles representing the current observed state is returned from tilingFunction. During the exploration phase, an ε-greedy policy (Table 3.4) is used to train an agent and approximate Q-values for the tiles. The Q-value for a tile is approximated only if it contains at least one point which represents a feature of an observed state. Hence, the value of ε has to be chosen carefully, so that at least few points of all available tiles are visited during training. During the exploitation phase, Q-policy is used to exploit the knowledge gained during training and evaluate the performance of the agent. In some adaptive tile coding methods, the data required for splitting a tile is also gathered during this phase. During the splitting phase, a tile is selected from the set of current tiles following the policy of specific adaptive tile coding method. The selected tile is partitioned into two different tiles. A variable $total$ is used to store the cumulative reward received during an exploitation phase of each split. The sequence of exploration, exploitation and splitting is repeated for P times.

3.2.2 Randomly Splitting Tiles

In random tile coding method, the tiles are chosen randomly for further partition. At the start, each feature in a state space is represented using the minimal number of tiles (two in this case). At the end of each pass, a tile is selected randomly and is further divided into two new tiles of equal size. The algorithm used for implementing the RL mechanism of smart tile coding is given in Table 3.2. It includes a modified form of Sarsa(λ) algorithm (explained in section 2.3) given in Table 3.3. The implementation of splitting_method() is provided in the Table 3.5. At the start of the algorithm, two variables $target\_point$ and $target\_tile$ are defined to store the point and the tile at which a split would occur; both the variables are initialized to 0. The value of $target\_tile$ is set to a tile selected randomly from the set of current tiles. The value of $target\_point$ is set to the mid-point of $target\_tile$. At the end, $target\_tile$ is split at $target\_point$ into two new separate tiles.

```
splitting_method()
  $target\_point = 0$
  $target\_tile = 0$
  $target\_tile =$ randomTiling(Tilings)   // selects a tile randomly from Tilings
  $target\_point =$ midPoint($t$)
  split $target\_tile$ at $target\_point$
```

Table 3.5: Algorithm to implement splitting_method() in random tile coding.

To understand more about how random tile coding is performed, an example is provided in Figure 3.2. The state space consists of two features X and Y, with X ranging from -4 to 4 and Y ranging from 0 to 2. At the start, all the features in a state space are partitioned into two tiles of equal size. At the end of first pass, tile 4 is selected randomly and is partitioned into tile 4 and tile 5. At the end of second pass, tile 1 is selected randomly and partitioned into tile 1 and tile 6. At the end of third pass, tile 1 is selected randomly and is partitioned into tile 1 and tile 7.

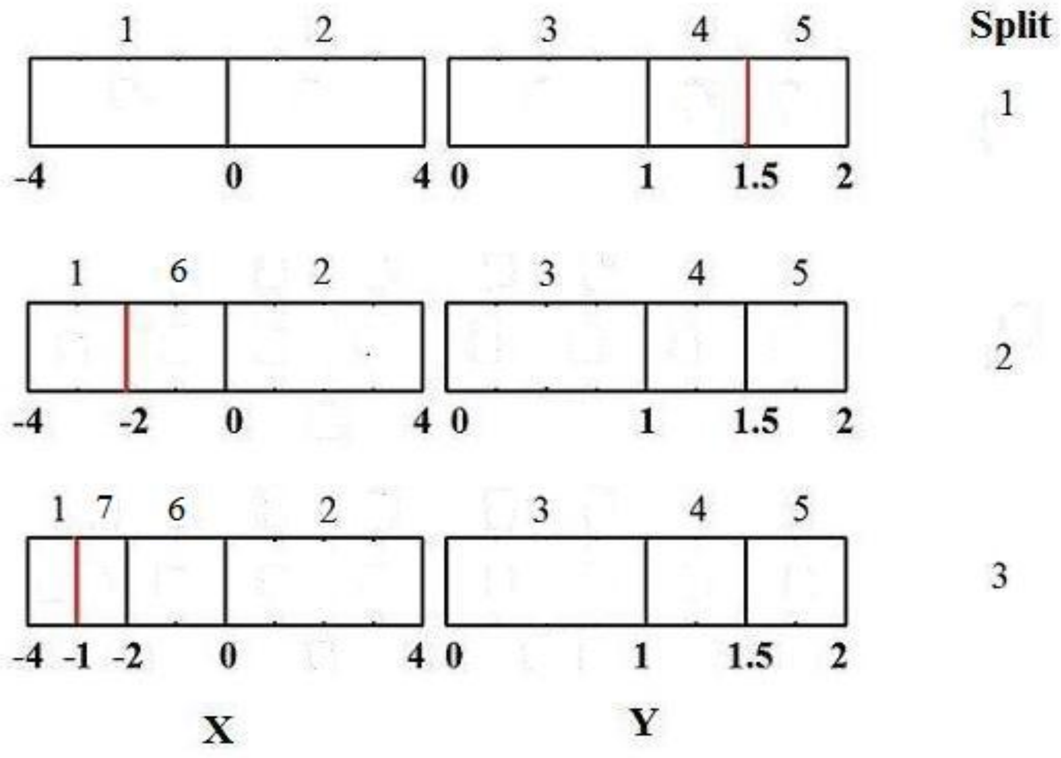


Figure 3.2: Example for random tile coding. The states space contains features X and Y. Each row represents the state of tiles and spits performed so far. Each cell represents a tile whose number is shown at the top of cell. The range of a tile is shown at the bottom of corresponding cell. During each pass, a split is performed randomly and it is represented using a red line.

3.2.3 Splitting Based on the "Size" of Features

In this method, splitting of tiles is based on the size (its extent in terms of its magnitude from the highest and lowest value it can represent) of features in a state space. At the start, each dimension is represented using a minimal number of tiles. During the course of the experiment, for each pass, a single tile is split into two new tiles of equal size. In feature-base tile coding, only the tiles for which the ratio of tile size and corresponding dimension size is maximum are eligible for splitting. If there is only one eligible tile, it is split into two new tiles of equal size. If there are multiple eligible tiles, one of the eligible tiles is selected randomly and split into two new tiles of equal size. For implementation, all the tiles in a state space are organized into groups of different levels, with newly formed tiles being grouped one level higher than its predecessor. A tile from any level (except first level) is allowed to split further only after splitting all the tiles at lower levels. If more than one tile is present at a level which is eligible for splitting, one of the tiles from that level is selected randomly.

```
create an integer variable $currentLevel$ and an integer counter $levelCounter$ during the
  first pass of the algorithm
initialize $level\_completed$ to false and $index$ to 0
$target\_point = 0, \; target\_tile = 0$
create an integer array $eligible\_tiles$ to store the numbers of tiles eligible for splitting
while ($index < levelCounter.length$ && ! $level\_completed$)
    if ($levelCounter[index] == currentLevel$)
        set $level\_completed$ to true
if ($level\_completed$)
    set $index$ to 0
    while $index < levelCounter.length$
        if ($levelCounter[index] == currentLevel$)
            append $index$ to $eligible\_tiles$
    set $target\_tile$ to a value drawn randomly from $eligible\_tiles$
    set the value of $levelCounter$ for new tiles to one level higher than
      $levelCounter$ of $target\_tile$
else
    set $index$ to 0 and increment $currentLevel$
    set $target\_tile$ to a value drawn randomly from all of the available tiles
    set the value of $levelCounter$ for new tiles to one level higher than
      $levelCounter$ of $target\_tile$
    set $tartget\_point$ to the midpoint of $target\_tile$
    split $target\_tile$ at $target\_point$ into two seperate tiles
```

Table 3.6: Algorithm to implement splitting_method() in feature size based tile coding.

The algorithm used for implementing the RL mechanism of feature size based tile coding is given in Table 3.2. It includes a modified form of the Sarsa(λ) algorithm (explained in section 2.3) given in Table 3.3. The implementation of splitting_method() for feature size based tile coding is provided in Table 3.6. During the first pass, it defines two integer variables to store the eligibility level of each available tile and the current eligibility level. The first variable is a dynamic integer array called $levelCounter$ with size equal to the total number of available tiles. The size of the array increases as new tiles are added to the state space. Each tile in the state space is associated with a cell of $levelCounter$. The value of cells in $levelCounter$ is used to keep track of current level of tiles in a state space. Initially the values of all the cells in a $levelCounter$ are set to 1, so that all available tiles at the start of algorithm are placed at level 1. The second variable is an integer variable called $currentLevel$, which represents the level of tiles eligible for a split. The value of $currentLevel$ is initialized to 1 at the start of a learning. It means that any tile which is at level 1 is eligible for a split. A tile is eligible for a split, only if the value of its corresponding cell in $levelCounter$ is equal to $currentLevel$.

The algorithm to implement splitting_method() in feature-based tile coding is provided in Table 3.6. At the start of the algorithm, two variables $target\_point$ and $target\_tile$ are defined to store the point and the tile at which a split would occur; both the variables are initialized to 0. During splitting, the values of all the cells in $levelCounter$ are compared with $currentLevel$, to check if there is at least one tile which is eligible for splitting. If there is only one tile which has the same value as $currentLevel$, $target\_tile$ is set to the tile and $target\_point$ is set to the mid-point of the tile. If there is more than one tile which has the same value as $currentLevel$, all eligible tiles are added to a new set T' and a tile is selected randomly from T'. The $target\_tile$ is set to the selected tile and the target point is set to the mid-point of the selected tile. In case if all the tile at $currentLevel$ are already partitioned, no tile will have $levelCounter$ value equal to $currentLevel$. In this case, the value of $currentLevel$ is incremented by one and a tile is selected randomly from all available tiles in state space. The value of $target\_tile$

is set to the selected tile and the value of $target\_point$ is set to the mid-point of the selected tile. In above all cases, the values of corresponding indexes in $currentLevel$ for new tiles are set to one level higher than that of their parent tile level. This avoids the possibility of splitting newly formed tiles with less size before splitting tiles with bigger size. At the end, $target\_tile$ is split at $target\_point$ into two new tiles. The sequence of exploration, exploitation and splitting is repeated for P times.

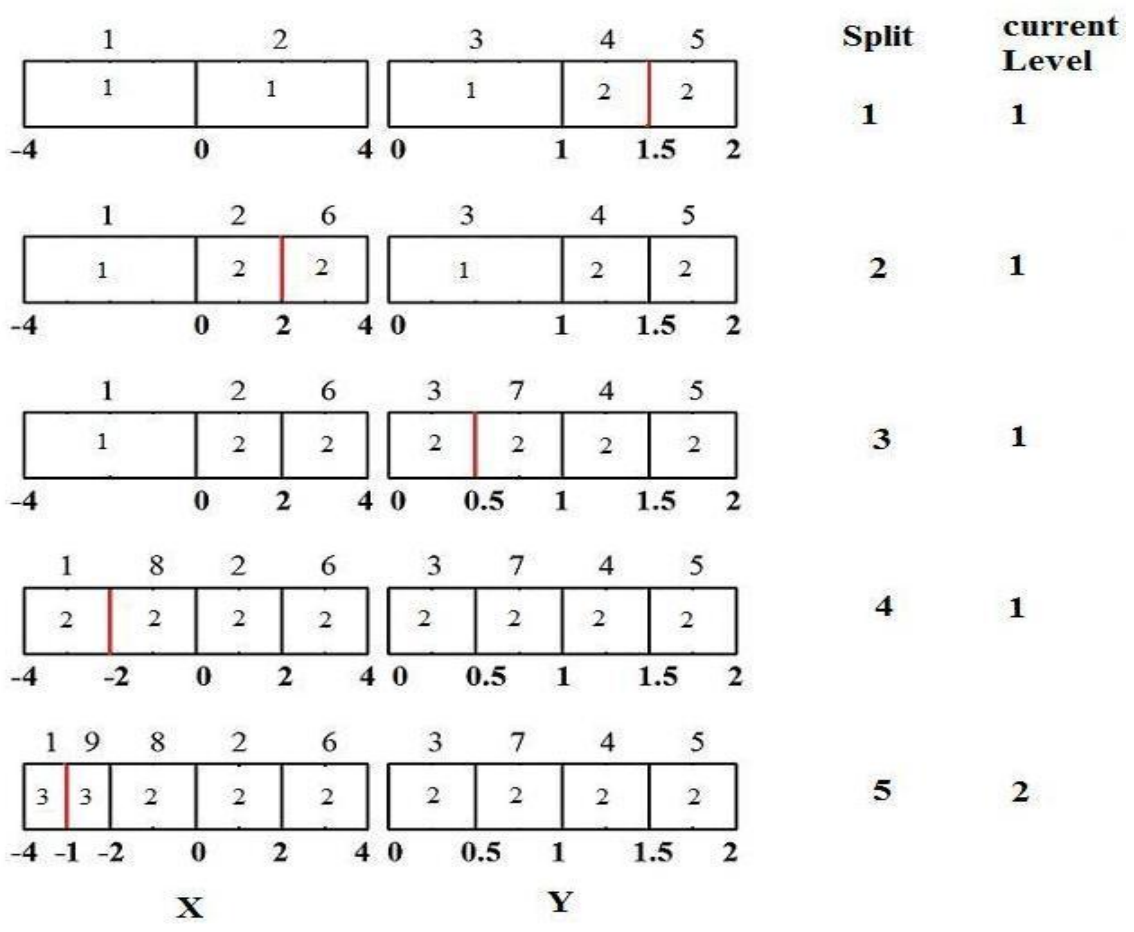


Figure 3.3: Example for feature size based tile coding. The states space contains two dimensions X and Y. Each row represents the state of tiles, splits performed so far and $currretLevel$. Each cell indicates a tile with tile number on the top, range at the bottom and $levelCounter$ at the center.

To understand more about how feature size based split tile coding is performed, an example is provided in Figure 3.3. At the start, all the features for X and Y in a state space are partitioned into two tiles of equal size, and all the four tiles are set to level 1. According to the algorithm, only the tiles at $currentLevel$ have to be partitioned; hence an eligible tile set contains only those tiles whose level is equal to $currentLevel$. At the start eligible tile set contains tiles 1, 2, 3 and 4. At the end of first pass, tile 4 is selected randomly from the eligible tile set. Tile 4 is split into two different tiles (tile 4 and tile 5) of equal size; the levels of both the tiles are set to 2. At the end of second pass, tile 2 is selected randomly from the eligible tile set containing tiles 1, 2 and 3. Tile 2 is split into two different tiles (tile 2 and tile 6) of equal size; the levels of both the tiles are set to 2. At the end of third pass, tile 3 is selected randomly from an eligible tile set containing tiles 1 and 3. Tile 3 is split into two different tiles (tile 3 and tile 7) of equal size; the levels of both the tiles are set to 2. At the end of fourth pass, the eligible tile set contains only tile 1, and it is selected automatically. Tile 1 is partitioned into two different tiles (tile 1 and tile 8) of equal size; the levels of both the tiles are set to 2. At the end of fifth pass, the eligible tile set contains no tiles. As all the tiles from level 1 are already partitioned and are updated to level 2, the value of $currentLevel$ is incremented to 2 and all available tiles (since all available tiles are at level 2) are added to the eligible tile set . Tile 1 is selected randomly from the eligible tile set and is partitioned into tile 1 and tile 9, whose levels are set to 3.

3.2.4 Splitting Based on the Observed Frequency of Input Value

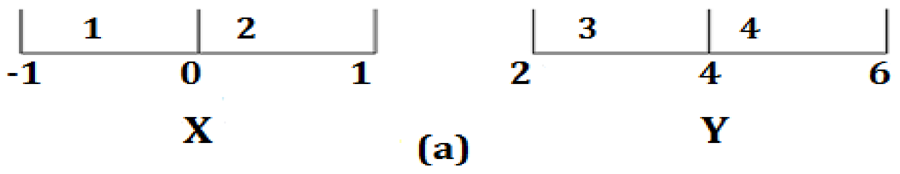
In this method, splitting of the tiles is based on the observed frequency of the states in a state space. At the start, each dimension in a state space is represented using a minimal number of tiles. During each pass, a tile with maximum number of observed states is partitioned into two tiles of equal size. The algorithm used for implementing a value-based split tile coding is given in Table 3.2. It includes a modified form of $Sarsa(\lambda)$

algorithm (explained in section 2.3) given in Table 3.3. At the start of a testing phase, the above specified Sarsa(λ) algorithm is extended to include an integer array called $valuecount$. The size of $valuecount$ is equal to the total number of available tiles. The array is used to store the frequency of observed states, tile wise. At each step, during the testing phase, the value of a cell in $valuecount$ with index equal to the tile number which contains the observed state is incremented by one. The frequency of a tile is obtained by adding the frequencies of all observed points represented by the tile. At the end of testing phase, $valuecount$ contains frequency of all the observed states, arranged tile wise. The algorithm used to implement splitting_method() in value-based tile coding is provided in Table 3.7. At the start of the algorithm, two variables $target\_point$ and $target\_tile$ are defined to store the point and the tile at which a split would occur; both the variables are initialized to 0. The values of all the cells in $valuecount$ are checked to find a cell with maximum frequency. The value of $target\_tile$ is set to the number of tile corresponding to the cell with maximum frequency. The value of $target\_point$ is set to the mid-point of $target\_tile$. At the end, $target\_tile$ is split at $target\_point$ into two new tiles.

```
splitting_method()
    $target\_tile = 0$
    $target\_point = 0$
    $target\_tile = max_{tile}(valueCount(tile)) \quad \forall tile \in$ Tilings
    $target\_point =$ midPoint($target\_tile$)        // Returns $target\_tile$ mid-point
    split $target\_tile$ at $target\_point$
```

Table 3.7: Algorithm to implement splitting_method() in value-based coding.

To illustrate how value-based tile coding is performed, an example with assumed data is provided in Figure 3.4. At the start, features X and Y, in a state space S, are partitioned into two tiles of equal size, as shown in Figure 3.4(a). At the end of first pass, cells in *valuecount* are filled with frequencies of corresponding tiles as shown in the table from Figure 3.4(b). It is evident from the table that tile 0 has highest frequency. Hence tile 0 is partitioned into two tiles of equal size, by splitting at midpoint -0.5, as shown in Figure 3.4(c).



Tile	Frequency
1	121
2	64
3	92
4	93

(b)

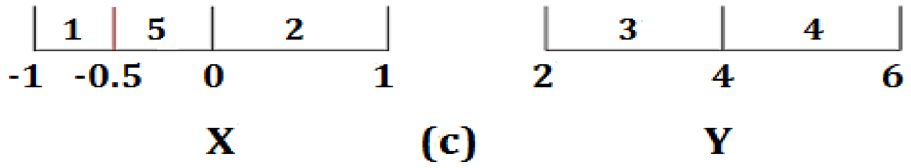


Figure 3.4: Example for value-based tile coding: (a) Initial representation of states space with features X and Y; each feature is represented using two tiles of size. X ranges from -1 to 1 and Y ranges from 2 to 6. (b) A table showing tile number and observed frequencies of points in each tile, after completion of *split* phase. (c) The representation of state space after splitting tile 1 into two tiles of same size, by splitting at -0.5.

3.2.5 Smart Tile Coding

In smart tile coding, splitting of tiles is based on the deviation between predicted Q-values and observed Monte Carlo Estimates. Each pass consists of training, testing and splitting phases. During training, the Q-values of tiles representing the observed states are updated using ε-greedy policy. During testing, the predicted Q-values are used by Q-policy to select the actions. For every observed state, the difference between Monte Carlo Estimate and predicted Q-value is used to guess the error in Q-value approximation (Q-deviation) for the points representing the observed state. At the end of testing, all observed points are grouped tile wise. For each tile, a point with maximum difference between Q-deviation of all the points on the left side and the right side of the point (in the tile) is selected. The tile whose selected point contains maximum Q-deviation difference among all the tiles is split at the selected point into two new tiles.

The algorithm used for implementing the RL mechanism of smart tile coding is given in Table 3.2. It includes a modified form of the Sarsa*(λ)* algorithm (explained in section 2.3) given in Table 3.3. In smart tile coding, splitting_method() require Q-deviation values for all the states observed in testing phase. The Q-deviation of an observed state is obtained by finding the difference between Monte Carlo Estimate and predicted Q-value of the state. The steps used to calculate Q-deviation values are provided in the Table 3.8. A data structure ($qError$) is used to store the Q-deviation values of all the observed states. In $qError$, observed states are saved as keys and the estimated Q-deviations of the corresponding observed states are stored as values for the keys. At the end of each pass, $qError$ is searched to find a matching key for the current state (s_{prev}). If a match is not found, s_{prev} is added as a key, with value equal to the difference between predicted Q-value ($Q_{a_{prev}}(t_{curr})$) and Monte Carlo Estimate (R) of s_{prev}. If a match is found, the value of the key is updated to the weighted average of the current value of the key ($Error(s_{prev})$) and the difference between predicted Q-value ($Q_{a_{prev}}(t_{curr})$) and Monte

Carlo Estimate (R) of s_{prev} .The above specified steps are executed at the end of each step during a testing phase, to record the Q-deviation values of the observed states.

```
$if\ (qError(s_{prev}))$
    $qError(s_{prev}) = weightedaverage(qError(s_{prev}), R - Q_{a_{prev}}(t_{curr}))$
else
    $qError(s_{prev}) = R - Q_{a_{prev}}(t_{curr})$
```

Table 3.8: Additional steps required in the Sarsa(λ) algorithm to approximate the estimated Q-values deviations during a testing phase.

The algorithm used to implement splitting_method() in smart tile coding is provided in Table 3.9. In splitting_method(), the predicted Q-value deviations of observed states are used to select a point for splitting. Initially, all the observed Q-value deviations are arranged tile wise. For each tile with an observed state, a point with maximum difference between Q-value deviations of either sides of the point is selected. The tile which contains a point with maximum value among all the tiles is selected for further division at the point. At the start of the algorithm, two variables $target\_point$ and $target\_tile$ are defined to store the point and the tile at which a split would occur; both the variables are initialized to 0. All observed states S are arranged tile wise following the order of value.
Starting from the least state, $left\_error$ and $right\_error$ are calculated tile wise for each observed state. To calculate $left\_error$ and $right\_error$ for an observed state, only the Q-deviations within the tile containing the observed state are considered. The $left\_error$ for a state is defined as the difference between the sum of all the Q-deviations on the left side of the state(including the current state) and the product of mean and number of elements on the left side of the state(including the current state). To calculate $left\_error$ for a state, a count of number of states on the left side of the state is required. The number of elements on left side of the state, including the current state is stored in $lcount$. The $right\_error$ for a state is defined as the difference between sum of

all the errors on the right side of the state and the product of $mean$ and number of elements on the right side of the state. To calculate $right\_error$ for a state, a count of number of states on the right side of the state is required. The number of elements on right side of the state is stored in $rcount$.

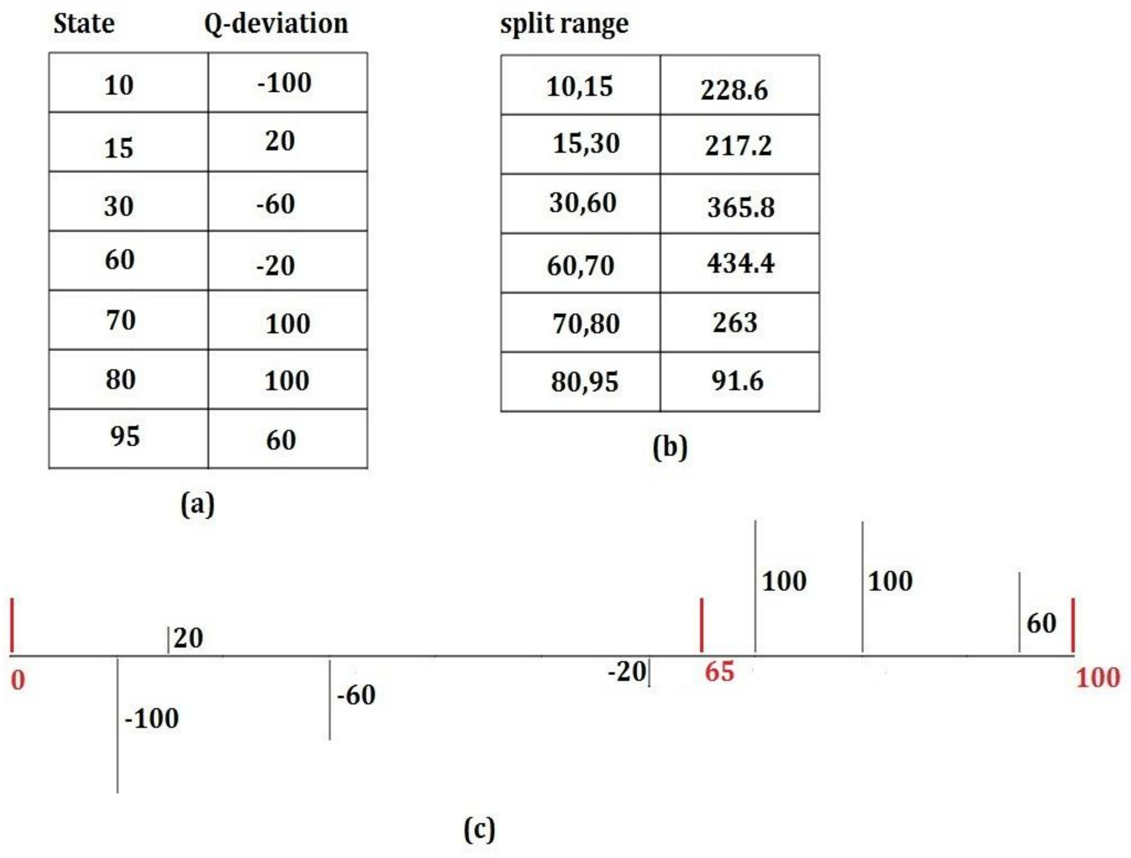
```
Arrange all observed states ($s$) tile-wise.
$target\_point = 0$
$target\_tile = 0$
for each tile $t \in T$:
    initialize $Lcount, Rcount, left\_errors, right\_errors$ $and$ max $\_diff$ to 0.
    set $mean$ to $average(Q\_error(s)), \forall f \in t$.
    sort observed features of $t$ in ascending order.
    for each $s'$ in $t$:
        $Lcount =$ number of elements to the left of $s' + 1$
        $Rcount =$ number of elements to the right of $s'$
        Set $left\_errors$ to sum of all $Q\_errors$ to the left of $s'$ and
            $Q\_error(s')$.
        update $left\_errors$ to difference of $left\_errors$ and $Lcount * mean$.
        set $right\_errors$ to sum of all $Q\_errors$ to the right of $s'$.
        update $right\_errors$ to difference of $right\_errors$ and $Rcount * mean$.
        if $abs(left\_errors - right\_errors) >$ max $\_difference$
                $max\_diff = abs(left\_errors - right\_errors)$.
            $target\_point = midpoint(s', s'')$, where $s''$ is the feature next
                        to $s'$
                $target\_tile = t$.
            split $target\_$tile into two tiles at $target\_point$.
```

Table 3.9: Algorithm to implement splitting_method() in smart tile coding.

A state L with maximum absolute difference between $left\_error$ and $right\_error$ among all observed states in a state space is selected. The range of points between L and next observed state R in the same tile is expected to contain most conflicting states on either side of it. The value of $target\_tile$ is set to the tile which contained the state L.

The above described method only finds the range of points but not a specific point for splitting. In this work, the mid-point of the obtained range is considered as the value of $target\_point$. For example, in an observed state list for tile t, if state x is followed by state y and the difference between $left\_error$ and $right\_error$ is maximum at x then split point lies in between x and y in tile t. In this work, I have chosen mid-point of x and y as the split point. The main algorithm splits $target\_tile$ at $target\_point$ into two separate tiles. The exploration, exploitation and splitting phases are repeated with new tiles until the number of splits performed is equal to P.

To illustrate how smart tile coding is performed, an example with assumed data for a tile is provided in Figure 3.5. In Figure 3.5(a) the approximated Q-value deviations for points representing observed states in a tile are represented using a table, and in Figure 3.2 (b) the difference between Q-deviations of either side of portion between observed consecutive feature pairs is represented using a table. To comprehend how Q-deviation difference is measured for an eligible split range, let us consider the split range which lies in between $(10{,}15)$. The only observed point to the left of $(10{,}15)$ is 10; hence $Lcount$ is set to 1 and $left\_error$ is set to the Q-deviation estimated at 10, which is -100. Since $mean$ of Q-deviations for observed points in the tile is 14.28, $left\_error$ is updated to -114.28, after subtracting $Lcount * mean$ from initial $left\_error$. The observed points to the right of $(10{,}15)$ are $15, 30, 60, 70, 80\ and\ 95$; hence $Rcount$ is set to 6 and $right\_error$ is set to the sum of Q-deviations of above points, which is 200. After subtracting $Rcount * mean$ from initial $right\_error$, $right\_error$ is updated to -114.32. The difference in *Q-deviations* between left, right portions of $(10{,}15)$ is equal to the absolute difference between $left\_error$ and $right\_error$, which is 228.6. The difference between Q-deviations of right side and left side portions of remaining eligible split ranges are also calculated in a similar way. It is evident from table in Figure 3.2(b) that split range $(60{,}70)$ contains maximum Q-deviation difference between left side and right side; hence split point for the tile lies in between points representing features 60 and 70. In this algorithm, I have used the midpoint of selected split range as the split point. Hence the split point in this case is 65, which is a midpoint for 60 and 70. In Figure 3.2(c), the tile is split into two new tiles at point 65.



State	Q-deviation
10	-100
15	20
30	-60
60	-20
70	100
80	100
95	60

(a)

split range	
10,15	228.6
15,30	217.2
30,60	365.8
60,70	434.4
70,80	263
80,95	91.6

(b)

Figure 3.5: Example for smart tile coding: (a) Representation of approximated Q-deviations for observed points in a tile, using a table. In each row, left cell represents an observed point and right cell represents a corresponding estimated Q-deviation. (b) Representation of difference in Q-deviations between right and portions of eligible split ranges using a table. In each row, the left cell represents an eligible split range and right cell represents the corresponding Q-deviation difference. (c) Splitting the parent tile at point 65 into new tiles.

3.2.6 Hand coded

In this method, split points are selected manually at the start of algorithm. The algorithm used for implementing the RL mechanism of hand-coded tiling is given in Table 3.2. It includes a modified form of the Sarsa*(λ)* algorithm (explained in section 2.3) given in Table 3.3. At the start, each feature in a state space is represented using minimal number of tiles. In splitting_method() , manually provided split points are used to split the tiles. The process of exploration, exploitation and splitting is repeated for P times.

CHAPTER 4

Experiment Setup

In this chapter I will explain the testing environment used for evaluating different adaptive tile coding mechanisms. In the first section, I will give a brief introduction to RL-Glue, a software tool used to test standard RL problems (Tanner and White, 2009). In the second section, a detailed description about multiple environments that are used for evaluating RL agents is provided.

4.1 RL-Glue

RL-Glue is a software tool used to simplify and standardize the implementation of RL experiments. The implementation of each module and establishing a communication mechanism between the modules of an RL framework (discussed in section 2.0) is complex. RL-Glue provides a simple solution to this problem by providing a common platform to implement RL agents, RL environments and RL experiments as isolated modules. It takes care of establishing and maintaining the communication mechanism between the modules. Implementing the agent, environment and experiment using the RL-Glue protocol reduces the complexity and the amount of code considerably. It allows the use of an RL agent developed using a particular policy with a variety of RL environments. In a similar way, it allows the use of a particular RL environment with a variety of RL agents. It is a perfect platform for comparing the performance and efficiency of different agents using a common environment.

RL-Glue considers the experiment setting, the environment and the agent as separate modules and requires separate programs to implement each module. A function calling mechanism is used to establish a connection between the modules. Figure 4.1 provides the standard framework used in RL-Glue.

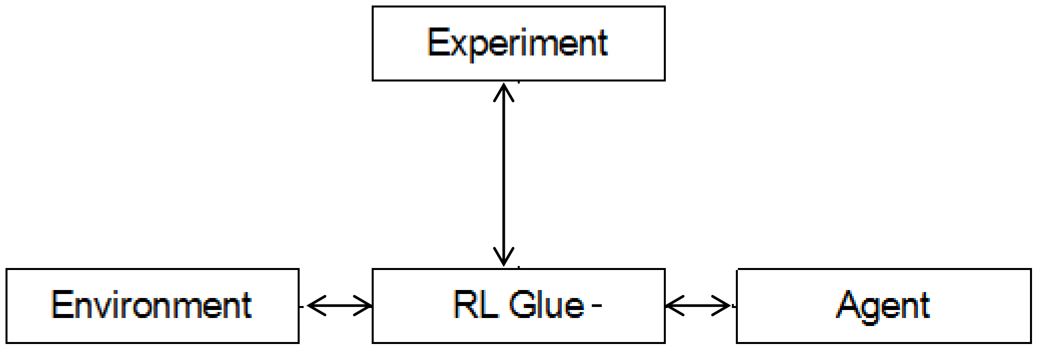


Figure 4.1: RL-Glue framework indicating separate modules for Experiment, Environment and Agent programs. The arrows indicates the flow of information.

4.1.1 The Experiment Module

The experiment module is responsible for starting an experiment, controlling the sequence of other module interactions, extracting the results, evaluating the performance of an agent and ending the experiment. It has to follow the RL-Glue protocol to send a message to either the agent module or the environment module. According to the protocol, it cannot access the functions of other modules and can interact with them only by calling the predefined RL-Glue functions. For example, it requires the use of $RL\_environment\_message()$ to send a message to the environment module and $RL\_agent\_message()$ to send a message to the agent module. $RL\_episode()$ is used to start an episode and to restrict the maximum number of steps allowed in an episode. $RL\_return()$ is called to extract the cumulative reward received for the actions of agent in the most recently finished episode. The general sequence of steps followed in the experiment module is provided in Table 4.1. At the start of an experiment, $RL\_init()$ is called to initialize both the environment and action modules. To start a new episode $RL\_episode(max\_steps\_allowed)$ is called with a parameter equal to the maximum number of steps allowed in the episode. After calling$RL_episode(max_steps_allowed)$, the control returns to the experiment module only after either a terminal state is reached or the number of steps executed in the episode is equal to $max\_steps\_allowed$. The return value ($reward$) of the above function is the cumulative reward received by the agent after completing the last episode, which is used to evaluate the performance of the agent. At the end of the experiment, $RL\_cleanup()$ is called to trigger cleanup functions in the agent module and the environment module to free any resources being held by the modules.

```
$RL\_init()$
$episode\_count = 0$
while $episode\_count < episodes\_limit$
   $RL\_episode(max\_steps\_allowed)$
   $reward = reward + RL\_return()$
        Use $reward$ to evaluate the agent
        $RL\_cleanup()$
```

Table 4.1: General sequence of steps followed in the experiment module.

The sequence of steps implemented by RL-Glue during the call to $RL\_episode(max\_steps\_allowed)$ is provided in Table 4.2. At the start of a new episode, $environment\_start()$ is called to set the current state ($currentState$) to some random state in the state space. It is followed by a call to $agent\_start(currentState)$ with the details of the current state. The agent, using its policy, will choose an action ($selectAction$) to perform at the current state. The value of $selectAction$ is used as a parameter to call $environment\_step(selectAction)$. The environment will update $currentState$ to a new state after applying $selectAction$ to $currentState$. A terminal flag ($terminalFlag$) is set to true if $currentState$ is a terminal state. The reward ($rewardAction$) for leading to $currentState$ is gauged. The value of an integer variable $stepsCount$ is updated by 1. The value of $rewardAction$ is added to a variable ($cumulativeReward$). If $terminalFlag$ is set or $stepsCount$ exceeds max_steps_allowd, $cumulativeReward$ is returned to the experiment module. Otherwise, $currnetState$ is sent to the agent using $agent\_step(currentState)$. The agent, using its policy will update the value of $selectAction$. Except for the first two steps, the process is repeated.

```
$RL\_episode(max\_steps\_allowed)$
  $currentState = environment\_start()$
  $selectAction = agent\_start(currentState)$
  repeat
    $rewardAction, currentState, terminalFlag = environment\_step(selectAction)$
    $cummulativeReward = cummulativeReward + rewardAction$
    $stepsCount = stepsCount + 1$
    if($terminalFlag \,||\, stepsCount >$ max $\_steps\_allowed$)
      return $cummulativeReward$
    $selectAction = agent\_step(currentState)$
```

Table 4.2: The sequence of steps followed by RL-Glue during a call to $\boldsymbol{RL\_episode}()$.

During an experiment, the experiment module can send messages to both the agent module and the environment module using $RL\_agent\_message(message)$ and $RL\_environment\_message(message)$.

4.1.2 The Environment Module

The environment module is responsible for initializing and maintaining all of the parameters describing a state space. It uses different functions to initialize and update the environment. Different RL-Glue functions used by the environment module are provided in Table 4.3. At the start of an experiment, $environment\_init()$ is called to initialize the environment data and to allocate resources required. At the start of every episode,

$environment\_start()$ is called to initialize the current state to some random state in the state space. The value of the current state is returned to the agent through RL-Glue. At each step of an episode, $environment\_step$ is called with the current selected action as a parameter. The environment performs $selectAction$ in the current state to get a new state and then the current state is updated to the resulting new state. If the new state is a terminal state the terminal flag is set to true. It also finds the reward for moving to the new state. It returns the terminal flag and the reward to RL-Glue and the current state to the agent through RL-Glue. At the end of an experiment, $environment\_cleanup()$ is called to free any resources allocated to the environment.

```
$environment\_init()$                    // Allocate required resources
$environment\_start()$                   // Initialize the current state arbitrarily
$environment\_step(selectAction)$
    • update the current state to a resulting state obtained after performing
      $selectAction$ in the current state
    • set the terminal flag to true if the resulting state is a terminal state
    • Find a reward for reaching the current state and return the reward to the agent.
$environment\_cleanup()$                 // Free the resources
```

Table 4.3: The common RL-Glue functions used in the environment module.

4.1.3 Agent

The role of the agent program is to select an action to be taken in the current state. The agent can use either a well-defined policy or just a random number generator to select an action for the current observed state. In general, it uses a policy to keep track of the desirability of all the states in a state space and chooses an action based on the desirability of the following state. The previously discussed adaptive tile coding mechanisms and other tile coding methods are implemented in the agent program. The performance of an agent depends on the policy being used by the agent to choose actions. Different RL-Glue functions used by the environment module are provided in Table 4.4. At the start of an experiment, $agent\_init()$ is called to initialize the agent data and to allocate resources required. At the start of every episode, $agent\_start()$ is called to select an action to be performed in the starting state. The action value is returned to the environment through RL-Glue. At each step of an episode, $agent\_step$ is called with the current state and previous reward as a parameter. The agent uses the reward to estimate the desirability of the previous action and chooses an action to perform in the current state. At the end of an experiment, $agent\_cleanup()$ is called to free any resources allocated to the agent.

```
$agent\_init()$                          // Allocate required resources
$agent\_start()$                         // Select an action to perform in the starting state
$agent\_step(currentState, reward)$
    • use the reward to update the desirability of the previous action
    • choose an action to perform in the current state
$agent\_cleanup()$                       // Free the resources
```

Table 4.4: The common RL-Glue functions used in the agent program.

4.2 Environments

Different environments are available in the RL-Glue library which can be used for evaluating the performance of an RL agent. In this work, I have used three different environments from the RL-Glue library to evaluate the performance of various tile coding mechanisms that I have implemented. The three different environments used are the puddle world problem, the mountain car problem, and the cart pole problem. All the parameters used in these environments are set according to the values given in the RL-Glue library specifications.

4.2.1 The Puddle World Problem

Puddle world is the one of the most basic environments used in evaluating the performance of an RL agent. It is a two-dimensional continuous state space with x and y parameters. The range of values is from 0 to 1 for both parameters. A portion of the state space in puddle world has puddles with variable depths. The puddles are spread around the line segment which starts at (0.1, 0.75) and ends at (0.45, 0.75), and the line segment which starts at (0.45, 0.4) and ends at (0.45, 0.8). A typical puddle world would look like the one shown in Figure 4.2.

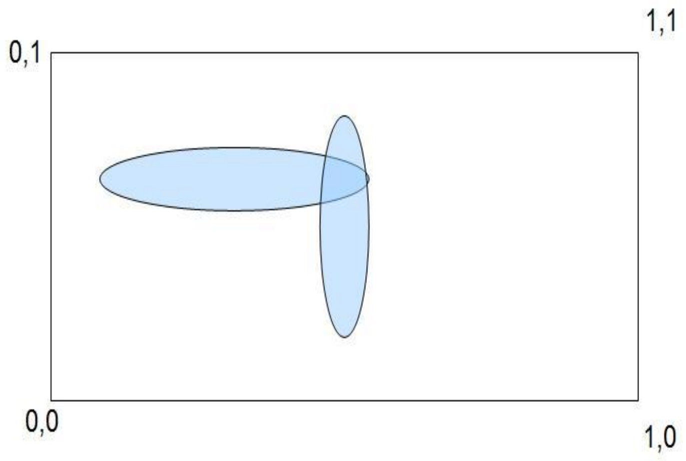


Figure 4.2: The standard puddle world problem with two puddles. The values along x and y are continuous from 0 to 1.

4.2.1.1 Actions

In the puddle world, four different actions are possible in each state. Given a state, it is the responsibility of an agent to choose one of the four possible actions. Table 4.5

represents the numbers associated with each action. According to the Table 4.5, 0 represents the action to move right, 1 represent the action to move left, 2 represents the action to move up, and 3 represents the action to move down.

action	number
right	0
left	1
top	2
down	3

Table 4.5: The possible actions and the corresponding integers used in the puddle world environment.

4.2.1.2 Goal State

An agent is said to reach a goal state if both the x-position and y-position of the agent is greater than or equal to 0.95 (in the upper right corner):

$$x_{pos} \geq 0.95 \; and \; y_{pos} \geq 0.95$$

4.2.1.3 Rewards

In puddle world, a negative reward is used to help an agent in state-action mapping. The action of an agent, which does not lead to an immediate puddle, receives a reward of -1 if it does not lead to an immediate goal state, and a reward of 0 if it leads to an immediate goal state. If the action taken by an agent leads to an immediate puddle, a large negative reward is received. The value of negative reward depends on the position of the agent in puddle. The negative reward for a position which is in the puddle and is closer to the center is higher when compared to a position which is in the puddle but near the boundary of the puddle. Hence, the negative reward for a position in the puddle is proportional to the distance between the center of the puddle and the position.

4.2.1.4 Objective

The goal of an agent is to reach the terminal state, from an arbitrary starting position in the least possible number of steps possible from the starting position while avoiding the puddles. At each pass, the agent starts from an arbitrary position within the state space. The Q-value of an action at a particular instance of a feature need not be constant; it might depend on the value of other feature. For example, the Q-value for moving right (towards goal state) has to be higher than the rest of the possible actions at x equal to 0.4. It does not hold true always; if the value of y for the state lies between 0.4 and 0.8, then agent has to either move up or bottom to avoid the puddle which is located towards the right side of the current state. Hence, an agent has to consider the effect of other features while determining the Q-value of an action for a feature. The maximum number of steps allowed during an episode is limited to 1000, if an agent is unable to reach the terminal state within 1000 steps, it is forced to quit the episode and a new episode is started.

4.2.2 The Mountain Car Problem

Mountain car environment is based on the mountain car task used by Sutton and Barron(1998). In this environment, an agent has to drive a car uphill to reach a goal position located at the top of a hill. The two dimensional state space has features for car position(x) and velocity(v). The value of x ranges from $-1.2\ to\ 0.6$ and the value of v ranges from $-0.07\ to\ 0.07$. The slope of the hill and the gravity makes sure that a car cannot simply reach the top position by only using full acceleration in forward direction from starting point. It has to build some momentum by moving backward. Only after attaining proper momentum the car can press forward with full acceleration to reach the goal state. During update, if the x value becomes less than -1.2, it is reset to -1.2.

Similarly if v crosses any of its boundaries, it is reset to the nearest range limit. Figure 4.3 shows a typical two-dimensional mountain car problem.

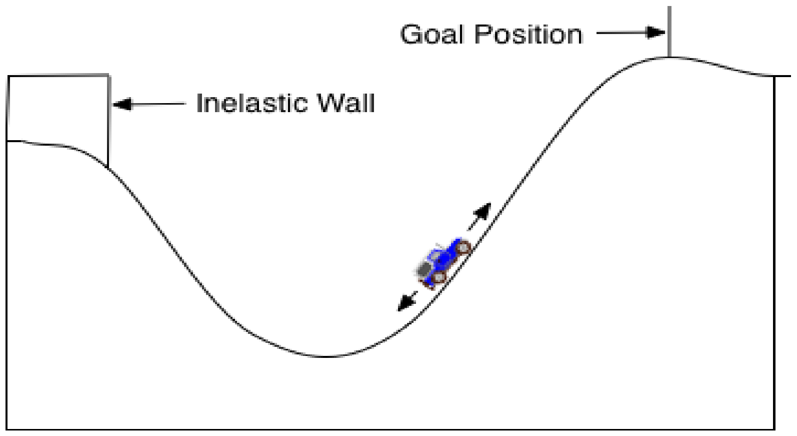


Figure 4.3: The standard view of mountain car environment. The goal of an agent is to drive the car up to the goal position. The inelastic wall ensures that car stays within the boundaries.

action	number
forward	2
neutral	1
backward	0

Table 4.6: The possible actions and the corresponding numbers in the mountain car environment.

4.2.2.1 Actions

Three different actions are possible at every given state in a mountain car problem. An agent can drive a car forward, backward and keep neutral. An agent has to choose one of the above three actions at each state. A set of integer values are used to represent the actions, Table 4.6 shows the possible actions and their corresponding integers.

The forward action pushes car towards the goal state, the backward action pushes car away from a goal state, and neutral action does not exert any external force to the car.

4.2.2.2 Goal State

An agent successfully completes an episode if it can drive the car to a position where x is greater than or equal to 0.6 (at the top of the hill):

$$x \geq 0.6$$

4.2.2.3 Rewards

In the mountain car problem, a negative reward is used to help an agent in state-action mapping. An agent is awarded -1 for each step in an episode. An agent receives more negative reward if it takes more steps to complete the episode, and receives less negative reward if it takes fewer steps to complete the episode. It forces the agent to learn how to reach the goal state in less number of steps, in order to get fewer negative rewards.

4.2.2.4 Objective

An agent has to learn how to drive a car to the top of the hill from any given arbitrary position in least possible steps possible from the starting position. The mountain car is a perfect RL problem to evaluate the performance of an RL agent. Here, the Q-value of a feature changes with the value of another feature. For example, the Q-value for a forward action and backward action at x is equal to 0.1 depending on the value of v. If the value of v is near to the upper limit, the Q-value for the forward action is higher when compared to the backward action and vice versa if the value of v is near to lower limit.

To generalize the state space for the mountain car problem is not trivial. Depending on the car position and velocity, an agent may have to learn to drive a car away from the goal state initially to build some momentum and then drive towards the goal state. The maximum number of steps allowed during an episode is limited to 1000. If an agent cannot reach the goal state before 1000 steps, it is forced to quit the episode.

4.2.3 The Cart Pole Problem

The cart pole problem is based on the pole-balancing task used by Sutton and Barron(1998) . In this environment, an agent has to control a cart moving along a bounded plain surface, so that a pole attached perpendicularly at the center of the cart lies within a certain angle range, which assures balance of the pole. The four-dimensional continuous state space has features for cart position(x), pole angle(a), car velocity(xv), and pole angular velocity (av) . The value of x ranges from -2.4 to 2.4, the value of a ranges from -12 degrees to 12 degrees. The balance of pole is lost, if x value becomes less than -2.4 or greater than 2.4, and if the value of a becomes less than -12 degrees or greater than 12 degrees. Figure 4.3 shows a typical four dimensional cart pole problem.

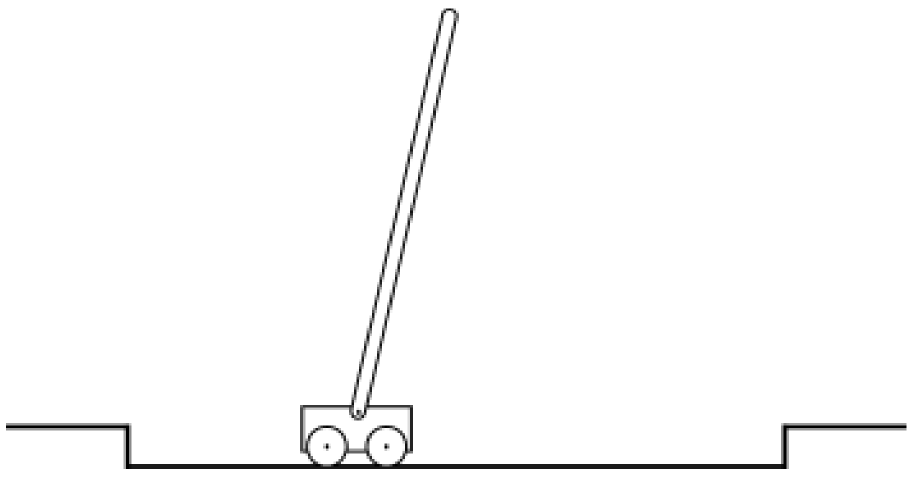


Figure 4.4: The standard view of the cart pole balance environment.

4.2.3.1 Actions

Two different actions are possible at every given state in a cart pole problem. An agent can push the cart towards the left or right. An agent has to choose one of the above two actions at each state. A set of integer values are used to represent the actions, Table 4.7 shows the possible actions and their corresponding integer number.

action	number
forward	1
backward	0

Table 4.7: Possible actions and corresponding integers used in a cart pole environment.

4.2.3.2 Goal State

The goal state of an agent is to successfully maintain the cart with in $[-2.4,2.4]$ and pole with in $[-12°, 12°]$ for as long as possible:

$$-2.4 \leq x \leq 2.4 \; and - 12° \leq a \leq 12°$$

4.2.3.3 Rewards

In cart pole, positive rewards are used to help an agent in state-action mapping. An agent is awarded 1 for each successful step in an episode. An agent receives more positive reward, if it maintains cart position and pole angle with in specified limits for more steps. It forces an agent to learn how to keep a pole balanced for long duration to receive higher rewards.

4.2.3.4 Objective

In all episodes, the initial position of the cart is at the center of bounded plain surface and the initial angle of pole is zero. An agent has to learn how to keep the pole in the balanced state for maximum possible number of steps. Cart pole is a complex problem because of the number of dimension being used in the environment. Similar to the above discussed puddle world and mountain car task, the Q-value of an action for a feature in cart pole depends on the value of the remaining features. In the earlier problems, the tile coding has to generalize the dependencies between two dimensions. In this problem, a feature action value function might depend on three other features, which is hard to generalize. For example, the Q-value for a forward action and backward action at x equal to -2.3 depends on the value of v, av and xv. If the value of v is near to lower limit, the Q-value for forward action has to be lower when compared to the backward action, but if the value of xv is also very high, then taking a backward action might cause the cart to cross the boundary, even if the pole is balanced. Hence while generalizing the state space for a cart pole problem, the dependencies between all the features should be considered and also priority should be given to a feature which might off balance the pole, rather than those which might disturb the balance. The maximum number of steps allowed during an episode is limited to 2500. If an agent can balance a pole for 2500 steps, the episode is considered successful and agent has to quit the episode.

CHAPTER 5

Experiments

In this chapter, I will analyze the efficiency of the proposed adaptive tile coding methods by evaluating the performance of RL agents trained using these methods. The examined adaptive tile coding methods include smart tile coding, value-based tile coding, feature-based tile coding and random tile coding. The testing environments used to evaluate the above tile coding methods are the mountain car problem, the cart pole balance problem and the puddle world problem. In the first section, I will discuss a general methodology that I have used for implementing all of the experiments. In the second section, I provide the results obtained from the above experiments and analyze the efficiency of the proposed adaptive tile coding methods using these results.

5.1 Methodology

The methodology that I used for implementing all the experiments in this work is similar for each experiment. An outline of these methodologies is provided in Table 5.1. In general, an experiment has three different phases: training phase, testing phase and splitting phase. During the training phase, an RL agent is trained to learn a near optimal policy for a given environment. In the process of learning a policy, the agent estimates the Q-values for all visited state action pairs. In order to estimate Q-values for state action pairs at different regions in a state space, an ϵ-greedy policy is used to introduce some randomness in action selection. The accuracy of the approximated Q-values depends on the generalization achieved by the corresponding adaptive tile coding method.

```
Input: Number of episodes for exploration (N), for exploitation (M), for splitting (K), for
       using discounted cumulative reward to update Q-values (L), and number of splits
       to perform (P)

• Repeat
  o Training
    ▪ Initialize Q-values for all possible state action pairs to 0
      • Repeat(episode)
        ▪ Repeat (step)
          ➢ Initialize start state arbitrarily
          ➢ Follow ϵ-greedy policy to choose an action
          ➢ If number of episodes is less than L
              Adjust Q-values for all state-action pairs using cumulative reward
          ➢ Else
              Adjust Q-values for all state-action pairs using immediate reward
              and the Q-value of the following state-action pair
        ▪ Until a terminal state is reached
      • Until number of episodes reach N
  o Testing
    ▪ Initialize cumulative_reward to 0
      • Repeat(episode)
        ▪ Repeat(step)
          ➢ Initialize start state arbitrarily
          ➢ Follow Q-policy to choose an action
          ➢ Increment cumulative_reward  by obtained reward
        ▪ Until a terminal state is reached
      • Until number of episodes reach M
  o Splitting
    ▪ Use an Adaptive tile coding method to select a split point
    ▪ Split the tile containing split point into two tiles, by splitting at split point
    ▪ Until number of splits performed is less than P
```

Table 5.1: The general methodology used to perform the experiments on testbed. The methodology has three different phases: exploration, exploitation and splitting.

During the testing phase, the performance of the trained agent is evaluated by allowing the agent only to choose an action with maximum estimated Q-value among all possible actions (Q-policy). The Q-values of the observed state action pairs are not updated during this phase. The performance of the agent depends on the accuracy of the Q-values estimated during the training phase. During the splitting phase, the RL data required by the corresponding adaptive tile coding method is gathered while following the Q-policy. The number of episodes used in each phase depends on the environment and is provided manually. For experiments implementing random tile coding, feature-based tile coding, and hand coded tiling, RL data is not required to split a tile; hence the splitting phase is not implemented for these methods. During the training phase and the splitting phase, random start-states are used for each split of an experiment to allow an agent to explore more states. During the testing phase, the same start states are used for each split to maintain the consistency in agent evaluation at different splits.

5.2 Baseline

In this work, I proposed various adaptive tile coding methods as an alternative to the existing multiple tilings method. Hence, the performance of the multiple tilings method is used as the baseline to evaluate the performances of the proposed adaptive tile coding methods in each environment. In an environment, if the performance of an adaptive tile coding method is above the baseline of the environment, the adaptive tile coding method is considered to outperform the multiple tilings method in the environment and vice-versa. A detailed description about multiple tilings is provided in Section 2.3.2. The code used for implementing multiple tilings is an extension of *Tile coding software*[1] developed by Sutton. In this software, a hashing technique is used for efficient allocation of the tiles to state-action pairs. A total of 5000 tiles with ten tilings are used to implement the multiple tilings. The values used for the RL parameters are environment dependent; different combinations of values are tried for each environment and a combination for

[1] http://webdocs.cs.ualberta.ca/~sutton/tiles2.html

which an agent performs best is chosen. The RL parameter values used for each environment are provided in the next section. Another instance of multiple tiling with a much smaller number of tiles (70 tiles with 8 tilings) is also implemented, to compare the performance of the adaptive tile coding methods and multiple tilings method for equal number of tiles. In both the cases, experiment is repeated 25 times and the average of obtained cumulative rewards is used as the baselines. The baselines for each environment are provided in the results sections of the environments. As mentioned before, I have used these baselines to evaluate the performances of different adaptive tile coding methods.

5.3 Results

In this section, the results obtained from the experiments performed on the testbed are used to evaluate the performance of the adaptive tile coding methods by comparing these results with multiple tilings baselines. The results contain the performance of the trained adaptive tile coding agents after each split. Here, I only provided the performances observed after every 5 splits, starting from 0 and until 60. The reason for choosing a step interval of 5 is to simplify the data representation and to smooth the final performance graph. The RL parameter values used in each environment are provided in the methodology section of the environments. In each environment, at the end, I compared the results of all adaptive tile coding methods to determine which adaptive coding method performs better than the rest in the environment. While analyzing the results in each environment, I attempted to answer the following questions related to a trained adaptive tile coding agent:

Is an agent able to improve the performance considerably by the end of an experiment?
Is the rate of improvement in performance consistently positive?
Is an agent able to perform better than a multiple tiling agent with same number of tiles?
Is an agent able to perform better than a multiple tiling agent with large number of tiles?
Is an agent able to perform better than the remaining adaptive tile coding agents?

Is an agent able to perform well in a state space with large number of features?

I also attempted to validate the following claims basing on the results:

- *The performance of trained smart tile coding agent can match the performance of a multiple tiling agent.*
- *Over the long run, the performance of a trained smart tile coding agent is better than the performance of inefficient adaptive tile coding methods like feature-based tile coding and random tile coding.*

5.3.1 Puddle World Experiments

The setup for the puddle world environment is given in Section 4.2.1. The outline of each phase with the order of execution is provided in Table 5.1. At the start of an experiment, each feature is divided into two tiles of equal size and the Q-values for all the available tiles are initialized to 0. The sequence of training, testing and splitting is repeated for 60 times to split the tiles 60 times. Since the puddle world environment has only two features and each feature is represented using two tiles, at the start of an experiment the state space is represented using four tiles and by the end of the experiment it is represented using 64 tiles (the original four features and the resulting split features created from the 60 splits). In this experiment, I used the same RL parameter values for all adaptive tile coding methods and multiple tilings method. Each experiment is repeated for 100 times and the final performance at each split is obtained by taking the average of all the observed performances at the split. The observed performances of the proposed tile coding agents in the puddle world environment are shown in Table 5.2. To simplify the data representation, only the performances observed after every 5 splits are shown in the table. The first column in the table contains the split count, the remaining columns contains the average rewards observed for random tile coding, feature-based tile coding, value-based tile coding, smart tile coding and hand-coding in order.

Number of splits	Average rewards				
	Random	Feature	Value	Smart	Handcoded
0	**-239.21**	**-216.01**	**-222.94**	**-223.68**	**-253.34**
5	-281.28	-283.61	-109.93	-194.42	-309.32
10	-270.47	-348.7	-120.87	-195.93	-159.26
15	-286.16	-377.81	-125.9	-151.94	-91.48
20	-270.32	-284.45	-151.07	-138.49	-89.39
25	-295.00	-198.36	-160.16	-130.18	-90.9
30	-297.91	-141.6	-160.02	-127.01	-94.96
35	-271.87	-130.69	-173.95	-129.38	-97.03
40	-243.48	-168.81	-186.52	-119.77	-98.99
45	-233.04	-142.09	-187.32	-106.24	-100.53
50	-249.91	-132.07	-179.05	-109.79	-111.55
55	-251.55	-118.29	-184.08	-104.62	-87.23
60	**-225.38**	**-117.46**	**-177.23**	**-97.31**	**-86.4**

Table 5.2: The average rewards received after every 5 splits in various adaptive tile coding methods in the puddle world environment. The first column in the table represents the number of splits performed. The remaining columns in the table contain the average rewards received at various splits for Random, Feature, Value, Smart and Manual tile codings in order.

5.3.1.1 Random Tile Coding

Methodology

The algorithm used to implement random tile coding method is provided in Section 3.2.2. The values of the different RL parameters used are: learning rate (α) = 0.01, discount rate (γ) = 0.9, decay rate (λ) = 0.9. The training phase is repeated for 1000 steps and the testing phase is repeated for 200 steps. There is no need for a splitting phase.

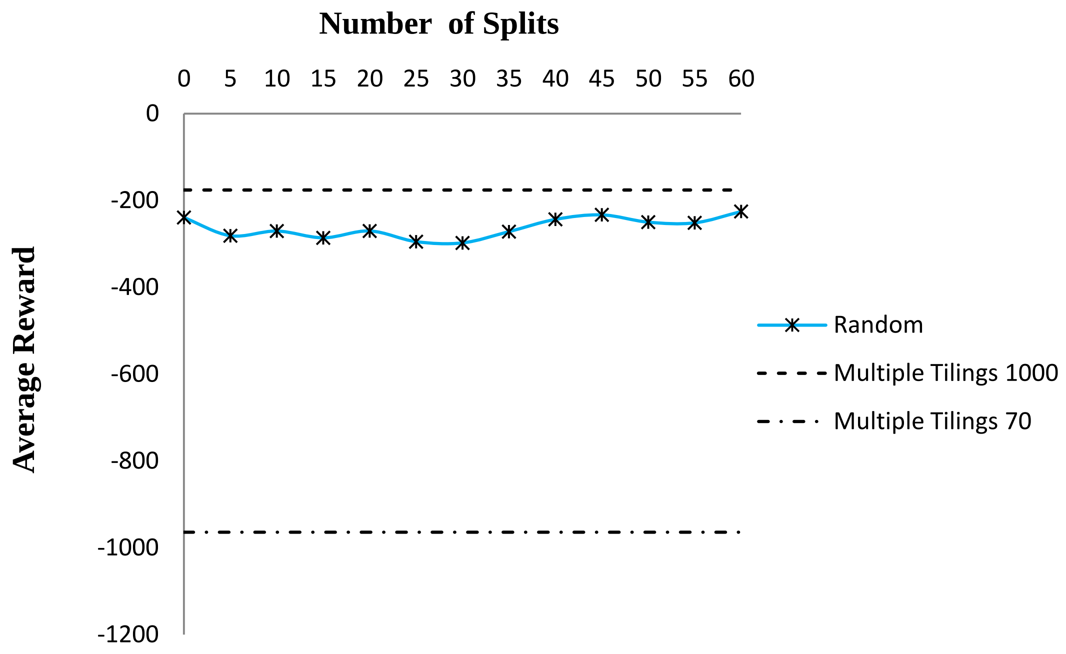


Figure 5.1: The graphical representation of the performance of random tile coding agent vs multiple tile coding agents in the puddle world environment. The X-axis represents the number of splits. The Y-axis represents the average reward received at each split.

Results

The average reward obtained after every five splits is show under the Random column in Table 5.2. In each row, the values in the Random column represent the performance of a random tile coding agent at a split specified in the first cell. In Figure 5.1, the baselines and the performance of random based tile coding method (from Table 5.2) in the puddle world are represented using graphs.

Analysis

At the end of first pass, before performing any split, the agent received an average reward of -239.21 which is slightly less than -225.38, the average reward received after 60 splits. It indicates that there is no considerable improvement in the performance of the random tile coding agent towards the end of experiment. It is evident from the sequence of observed average rewards that there is no consistency in the performance of agent, as the performances of the agent at most of the split intervals are less than the performance at the start. It is evident from these observations that the rate of performance improvement of the random tile coding agent is not consistently positive. According to Figure 5.1, a multiple tiling agent trained with 5000 tiles is able to achieve an average reward of -174.9 which is higher than the average reward received by the random tile coding agent after 60 splits. The performance achieved by a multiple tiling agent with 70 tiles is less than that of the random tile coding agent. It suggests that random tile coding agent cannot match the performance of multiple tilings agent in the puddle world environment.

5.3.1.2 Feature-based Tile Coding

Methodology

The algorithm used to implement feature-based tile coding method is provided in Section 3.2.3. The values of different RL parameters used are: learning rate (α) = 0.01, discount rate (γ) = 0.9, decay rate $(\lambda) =$ 0.9. The training phase is repeated for 1000 steps and the testing phase is repeated for 200 steps. There is no need of splitting phase as features are split based on the relative size of the tiles which does not require RL data.

Results

The average reward obtained after every five splits is show under the Feature column in Table 5.2. In each row, the values in the Feature column represent the performance of a feature tile coding agent at a split specified in the first cell. In Figure 5.2, the baselines and the performance of feature-based tile coding method (from Table 5.2) in the puddle world are represented using graphs.

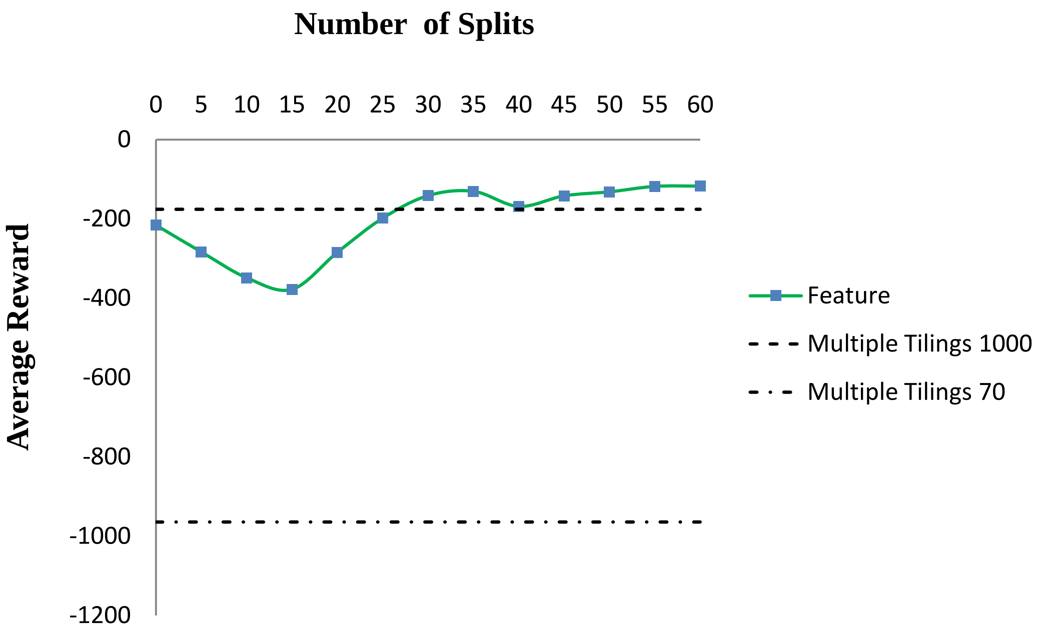


Figure 5.2: The graphical representation of performance of a feature-based tile coding agent vs multiple tile coding agent in the puddle world environment. The X-axis represents the number of splits. The Y-axis represents the average reward received at each split.

Analysis

At the start of the experiment, before performing any split, the agent achieved an average reward of -216.01 which is less than -117.31, the average reward received after 60 splits. It proves that, in the puddle world environment, there is a considerable improvement in the performance of the feature-based tile coding agent by the end of experiment. It is also evident from the table that the rate of improvement in performance of the agent is not consistently positive. At the start of experiment (after 0th, 5th and 10th split) and later at the 35th split a big dip in the performance is observed.

According to the graphs in Figure 5.2, the average reward received (-117.46) by the feature-based tile coding agent after 60 splits is higher than the baseline (-174.9). The performance achieved by a multiple tiling agent with 70 tiles is lot less than the rest of the two agents. It suggests that, in the puddle world, a feature-based tile coding agent can match the performance of a multiple tiling agent.

5.3.1.3 Value-based Tile Coding

Methodology

The algorithm use to implement value-based tile coding method is provided in Section 3.2.4. The values of different RL parameters used are: learning rate (α) = 0.01, discount rate (γ) = 0.9, decay rate (λ) = 0.9. The training phase is repeated for 1000 steps. The testing phase and splitting phase is repeated for 200 steps.

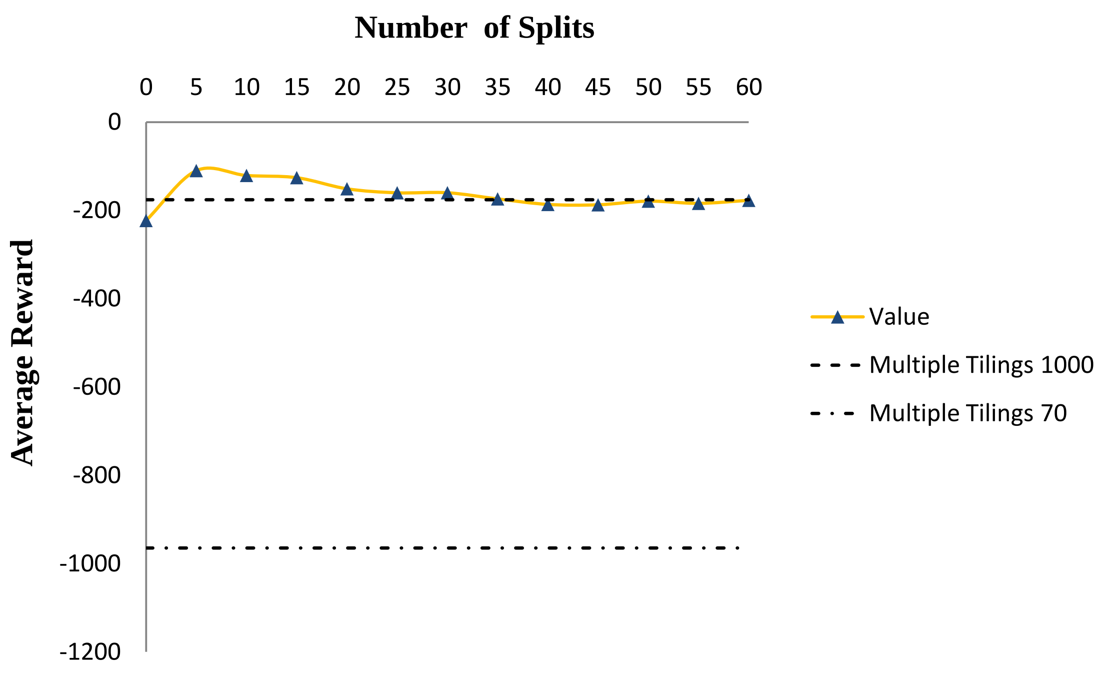


Figure 5.3: The graphical representation of the performance of value-based tile coding agent vs multiple tilings agents in the puddle world environment. The X-axis represents the number of splits. The Y-axis represents the average reward received at each split.

Results

The average reward obtained after every five splits is show under the Value column in Table 5.2. In each row, the values in the Value column represent the performance of a value tile coding agent at a split specified in the first cell. In Figure 5.3, the baselines and the performance of value-based tile coding method (from Table 5.2) in the puddle world are represented using graphs.

Analysis

At the start of the experiment, before performing any split, the agent achieved an average reward of -222.94 which is less than -177.23, the average reward received after 60 splits. In indicates that in the puddle world environment, there is a considerable improvement in the performance of the value-based tile coding agent by the end of experiment. It is evident from the graph that the rate of improvement in performance of the agent is sharp and positive for the first 5 splits, and negative for large part of the next 50 splits. It indicates that the performance of value-based tile coding agent is not consistent in the puddle world environment.

According to the graphs in Figure 5.3, a multiple tiling agent trained with 5000 tiles is able to achieve an average reward of -174.9 which is pretty close to the average reward received by the value-based tile coding agent after 60 splits. The performance achieved by a multiple tiling agent with 70 tiles is lot less than the rest of the two agents. It suggests that, in the puddle world environment, a value-based tile coding agent can match the performance of a multiple tiling agent.

5.3.1.4 Smart Tile Coding

Methodology

The algorithm used to implement smart tile coding method is provided in Section 3.2.5. The values of different RL parameters used are: learning rate (α) = 0.01, discount rate (γ) = 0.9, decay rate (λ) $=$ 0.9. The training phase is repeated for 1000 steps. The testing phase and splitting phase is repeated for 200 steps.

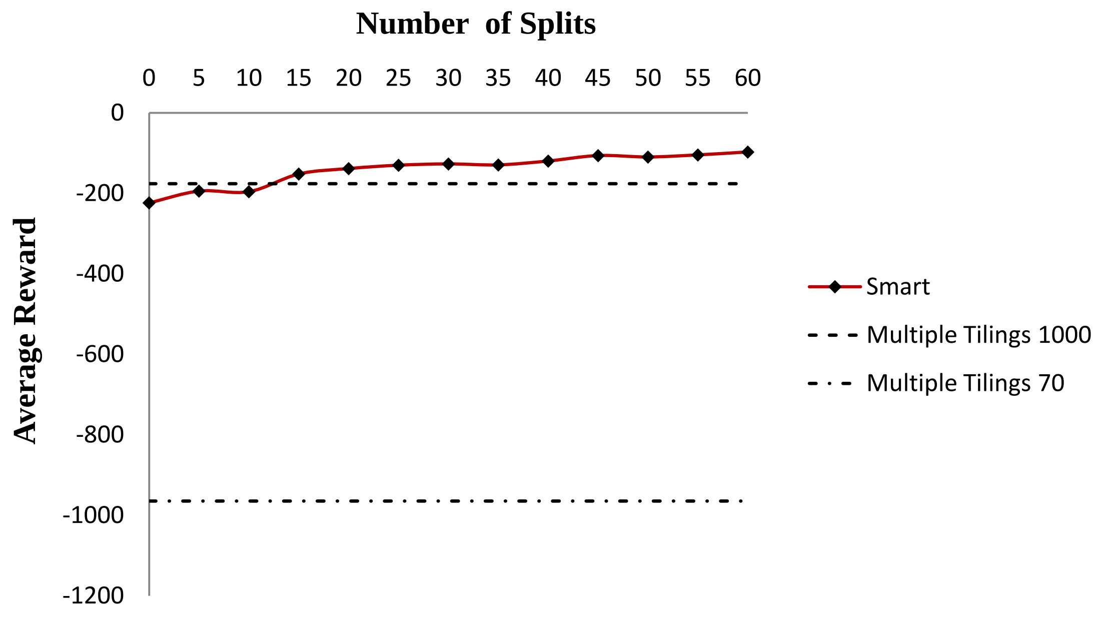


Figure 5.4: The graphical representation of performance of smart tile coding agent vs multiple tile coding agents in the puddle world environment. The X-axis represents the number of splits. The Y-axis represents the average reward received at each split.

Results

The average reward obtained after every five splits is show under the Smart column in Table 5.2. In each row, the values in the Smart column represent the performance of a smart tile coding agent at a split specified in the first cell. In Figure 5.4 the baselines and the performance of value based tile coding method (from Table 5.2) in the puddle world are represented using graphs.

Analysis

At the start of the experiment, before performing any split, the agent achieved an average reward of -223.68 which is less than -97.31, the average reward received after 60 splits. It proves that, in the puddle world environment, there is a considerable improvement in the performance of the smart tile coding agent towards the end of experiment. It is also evident from the table that the rate of improvement in performance of the agent is consistently positive at all split intervals except after 30 and 45, where a small dip in the performance is observed.

According to the graph in Figure 5.4, a multiple tiling agent trained with 5000 tiles is able to achieve an average reward of -174.9 which is less than the average reward received by the smart tile coding agent after 60 splits. The performance achieved by a multiple tiling agent with 70 tiles is lot less than the rest of the two agents. It suggests that, in the puddle world, a smart tile coding agent can match the performance of a multiple tiling agent.

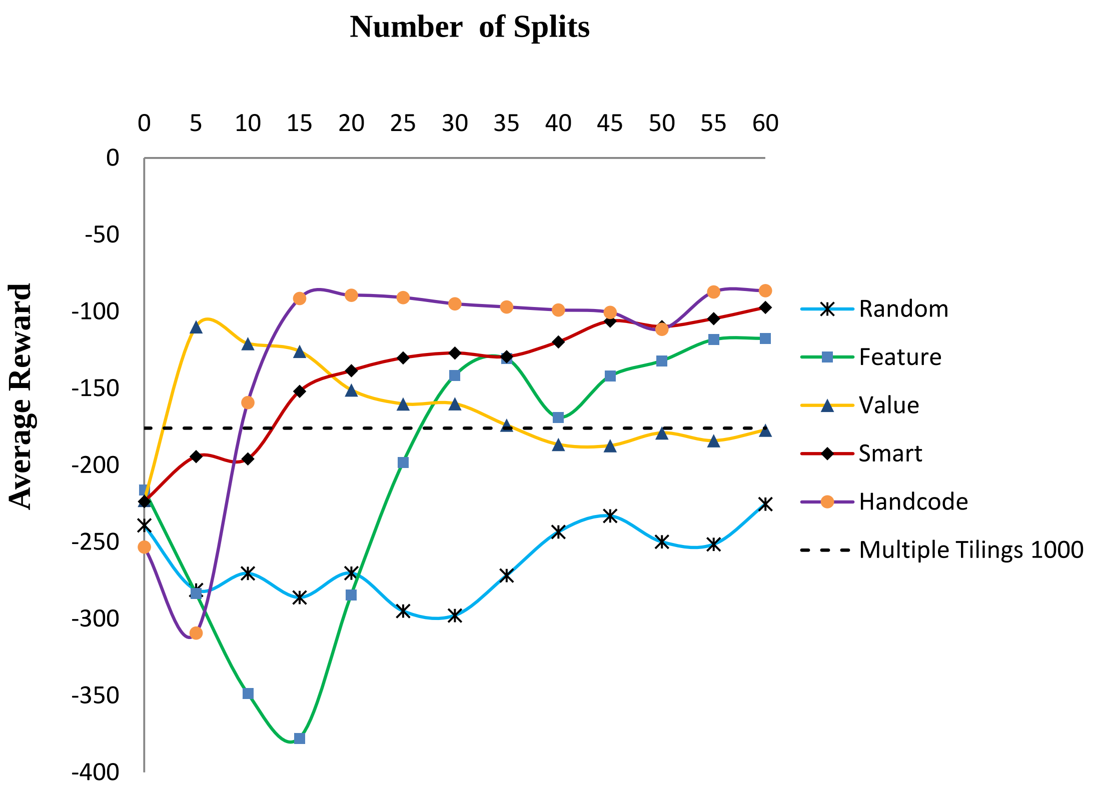


Figure 5.5: The graphical representation of performance of various adaptive tile coding methods, hand-coding method and multiple tilings method in the puddle world environment. The X-axis represents the number of splits. The Y-axis represents the average reward received at each split.

5.3.1.5 Claim Validation

In Figure 5.5, the performances of various tile coding algorithms discussed in the above sections are represented using graphs to validate the claims raised in section 5.3. The performance of hand-code algorithms is also included in the final graph to complete the evaluation.

It is evident from the graph that, in the puddle world environment:

- Smart tile coding algorithm has outperformed all of the remaining adaptive tile coding methods.
- Since the performance of smart tile coding agent is better than the performance of random tile coding and feature-based tiled coding agents, the claim that a smart tile coding algorithm outperform pseudo adaptive tile coding algorithms is validated in the puddle world environment.
- Since the smart tile coding agent has outperformed the multiple tilings agent, the claim that a smart tile coding method matches the performance of multiple tile coding method is validated in the puddle world.

5.3.2 Mountain Car Experiments

The setup for the mountain car environment is given in Section 4.2.2. The outline of each phase and their order of execution are provided in Table 5.1. At the start of an experiment, each feature is divided into two tiles of equal size and the Q-values for all the available tiles are initialized to 0. The sequence of training, testing and splitting is repeated for 60 times to split the tiles 60 times.

Number of splits	Average rewards				
	Random	**Feature**	**Value**	**Smart**	**Handcoded**
0	**-804.31**	**-825.23**	**-731.61**	**-848.25**	**-940.35**
5	-715.54	-734.97	-894.11	-570.77	-762.02
10	-655.04	-582.26	-490.08	-241.16	-636.58
15	-682.23	-555.19	-513.57	-256.96	-764.27
20	-620.04	-459.79	-542.64	-265.99	-869.45
25	-651.07	-559.45	-494.51	-196.25	-506.1
30	-526.54	-421.35	-309.24	-232.69	-434.04
35	-651.79	-417.92	-289.14	-169.94	-372.99
40	-551.85	-383.48	-382.14	-215.67	-439.95
45	-628.71	-374.02	-317.34	-183.74	-418.05
50	-556.71	-412.33	-375.94	-212.58	-360.06
55	-537.99	-419.18	-345.38	-193.97	-355.76
60	**-489.13**	**-413.35**	**-317.63**	**-166.14**	**-355.86**

Table 5.3: The average rewards received after every 5 splits for various adaptive tile coding methods in the mountain car environment. The first column in the table represents the number of splits performed. The remaining columns in the table contain the average rewards received at a split for Random, Feature, Value, Smart and Manual tile codings in order.

Since the mountain car environment has two features and each feature is represented using two tiles, at the start of an experiment the state space is represented using four tiles and towards the end of the experiment the state space is represented using 64 tiles (the original four features and the resulting split features created from the 60 splits. Same RL parameter values are used for all adaptive tile coding methods and multiple tilings method. Each experiment is repeated for 100 times and the final performance at each split is obtained by taking the average of all the observed performances at the split. The observed performance of the proposed tile coding methods in the mountain care environment is shown in Table 5.3. To simplify the data representation, only the performances observed for every 5 splits are shown in the table. The first column in the table contains the split count, the remaining columns represents the average rewards recorded for random tile coding, feature-based tile coding, value-based tile coding, smart tile coding and hand-coding in order.

5.3.2.1 Random Tile Coding

Methodology

The algorithm used to implement random tile coding method is provided in section 3.2.2. The values of different RL parameters used are: learning rate (α) = 0.01, discount rate (γ) = 0.99, decay rate $(\lambda) = 0.9$. The training phase is repeated for 1000 steps and the testing phase is repeated for 200 steps. There is no need for a splitting phase.

Results

The average reward obtained after every five splits is shown under the Random column in Table 5.3. In each row, the values in the Random column represent the performance of the random tile coding agent at a split specified in the first cell. In Figure 5.6, the baselines and the performance of random based tile coding method (from Table 5.3) in the mountain car are represented using graphs.

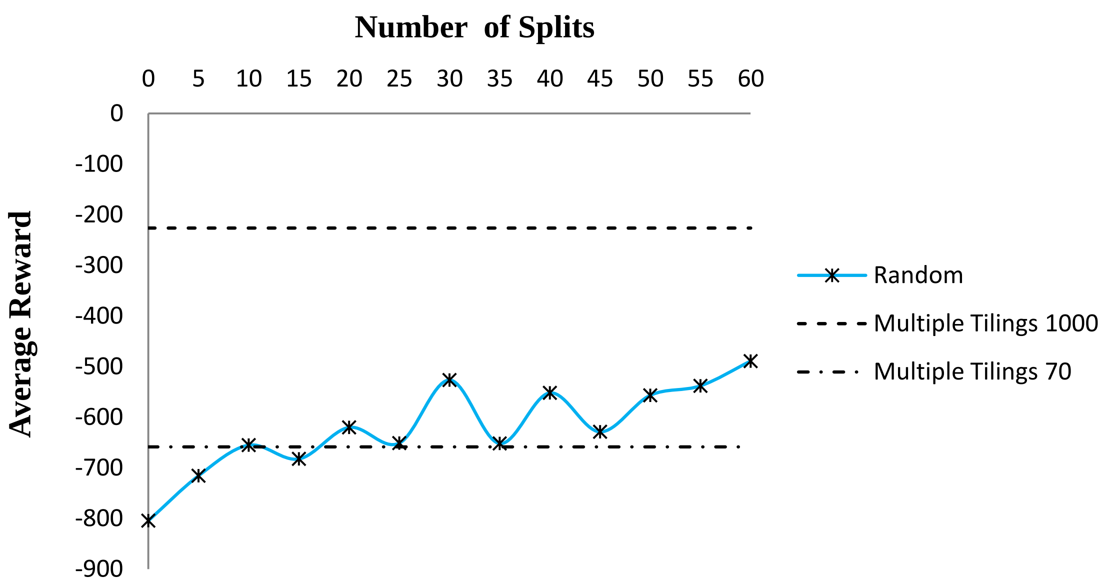


Figure 5.6: The graphical representation of performance of a random tile coding agent vs multiple tile coding agents in the mountain car environment. The X-axis represents the number of splits. The Y-axis represents the average reward received at each split.

Analysis

At the end of the first pass in the experiment, the agent received an average reward of -804.31 which is less than -481.38, the average reward received after 60 splits. It indicates that there is a considerable improvement in the performance of the random tile coding agent by the end of the experiment. It is evident from the graph that the performance of the agent is not consistent over the course of the experiment, but still there is an improvement in the performance.

The multiple tilings agent trained with 5000 tiles is able to achieve an average reward of -226.2 which is greater than the average reward received by the random tile coding agent after 60 splits. The performance achieved by a multiple tiling agent with 70 tiles (-658.26) is less than the performance of random tile coding agent after 60 splits. It suggests that, in the mountain car environment, a random based tile coding agent cannot match the performance of a multiple tilings agent.

5.3.2.2 Feature-based Tile Coding

Methodology

The algorithm used to implement feature-based tile coding method is provided in Section 3.2.3. The values of different RL parameters used are: learning rate (α) = 0.5, discount rate (γ) = 0.9, decay rate $(\lambda) =$ 0.9. The training phase is repeated for 1000 steps and the testing phase is repeated for 200 steps. There is no need for a splitting phase as the features are split based on the relative size of tiles which does not require RL data.

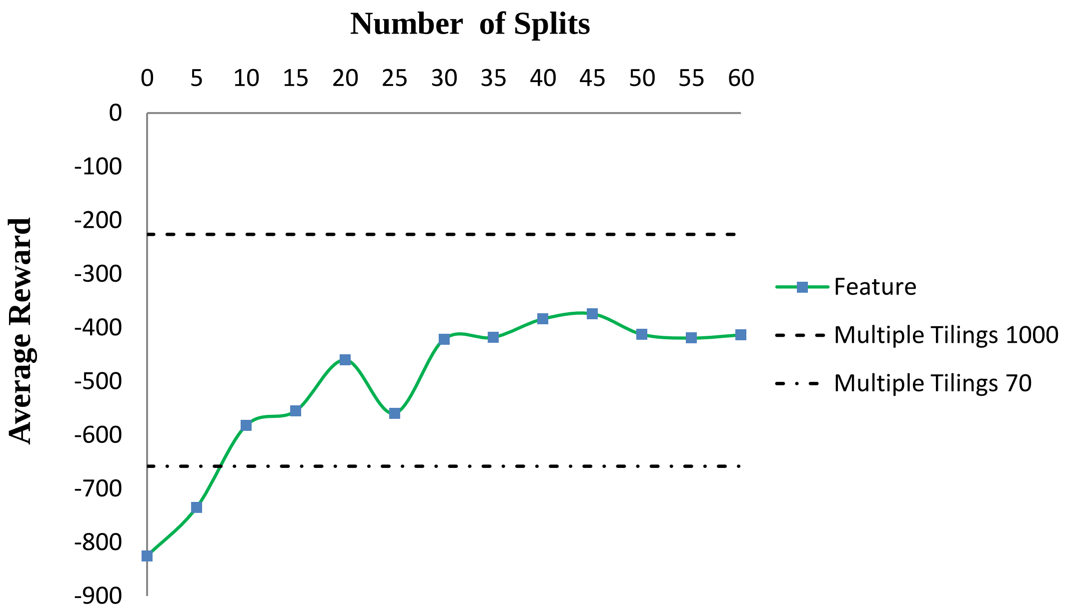


Figure 5.7: The graphical representation of the performance of feature-based tile coding agent vs multiple tile coding agent in the mountain car environment. The X-axis represents the number of splits. The Y-axis represents the average reward received at each split.

Results

The average reward obtained after every five splits is shown under the Feature column in Table 5.3. In each row, the values in the Feature column represent the performance of the feature tile coding agent at a split specified in the first cell. In Figure 5.7, the baselines and the performance of feature-based tile coding method (from Table 5.3) in the mountain car are represented using graphs.

Analysis

At the start of the experiment, before performing any split, the agent achieved an average reward of -825.23 which is less than -413.35, the average reward received after 60 splits. It indicates that, in the mountain car environment, there is a considerable improvement in the performance of the feature-based tile coding agent by the end of experiment. It is also evident from the table that the performance of the agent is consistent for most part of the experiment except in between splits 20 and 25, where a considerable dip in performance is observed.

According to the graph given in Figure 5.7, the average reward received (-413.35) by the feature-based tile coding agent after 60 splits is lower than the baseline (-226.2). The performance achieved by a multiple tiling agent with 70 tiles (-658.26) is lot less than the rest of the two agents. It suggests that, in the mountain car environment, a feature-based tile coding agent cannot match the performance of a multiple tiling agent.

5.3.2.3 Value-based Tile Coding

Methodology

The algorithm used to implement value-based tile coding method is provided in Section 3.2.4. The values of different RL parameters used are: learning rate (α) = 0.5, discount rate (γ) = 0.9, decay rate (λ) $=$ 0.9. The training phase is repeated for 1000 steps. The testing phase and splitting phase are repeated for 200 steps.

Results

The average reward obtained after every five splits is shown under the Value column in Table 5.3. In each row, the values in the Value column represent the performance of the value tile coding agent at a split specified in the first cell. In Figure 5.8 the baselines and the performance of value-based tile coding method (from Table 5.3) in the mountain car are represented using graphs.

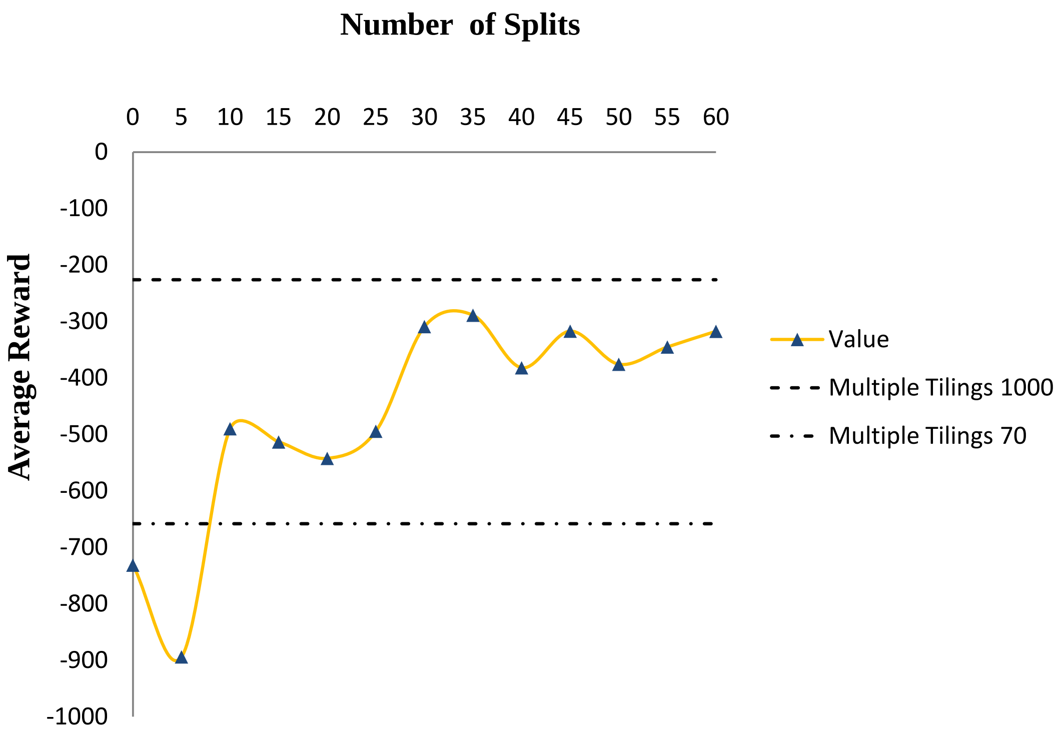


Figure 5.8: The graphical representation of performance of value-based tile coding agent vs multiple tile coding agents in the mountain car environment. The X-axis represents the number of splits. The Y-axis represents the average reward received at each split.

Analysis

At the start of the experiment, before performing any split, the agent achieved an average reward of -731.61 which is less than -317.63, the average reward received after 60 splits. It proves that, in the mountain car environment, there is a considerable improvement in the performance of the value-based tile coding agent towards the end of experiment. It is evident from the table that performance improvement rate of the agent is not consistently positive.

According to the graphs in Figure 5.8, the baseline with value equal to -226.20 is greater than the average reward received by the value-based tile coding agent after 60 splits. The performance achieved by a multiple tiling agent with 70 tiles is lot less than the rest of the two agents. It suggests that, in the mountain car environment, a value-based tile coding agent cannot match the performance of a multiple tiling agent.

5.3.2.4 Smart Tile Coding

Methodology

The algorithm used to implement smart tile coding method is provided in Section 3.2.5. The values of different RL parameters used are: learning rate (α) = 0.5, discount rate (γ) = 0.9, decay rate $(\lambda) = 0.9$. The training phase is repeated for 1000 steps. The testing phase and splitting phase are repeated for 200 steps.

Results

The average reward obtained after every five splits is shown under the Smart column in Table 5.3. In each row, the values in the Smart column represent the performance of the smart tile coding agent at a split specified in the first cell. In Figure 5.9 the baselines and the performance of smart tile coding method (from Table 5.3) in the mountain car are represented using graphs.

Analysis

At the start of the experiment, before performing any split, the agent achieved an average reward of -848.25 which is less than -166.14, the average reward received after 60 splits. It proves that, in the mountain car environment, there is a considerable improvement in the performance of the smart tile coding agent by the end of experiment. It is also evident from the table that performance of the agent improved sharply from 0 to 10 splits and it is consistent for most of the split intervals except at few split points where a small dip in the performance is observed.

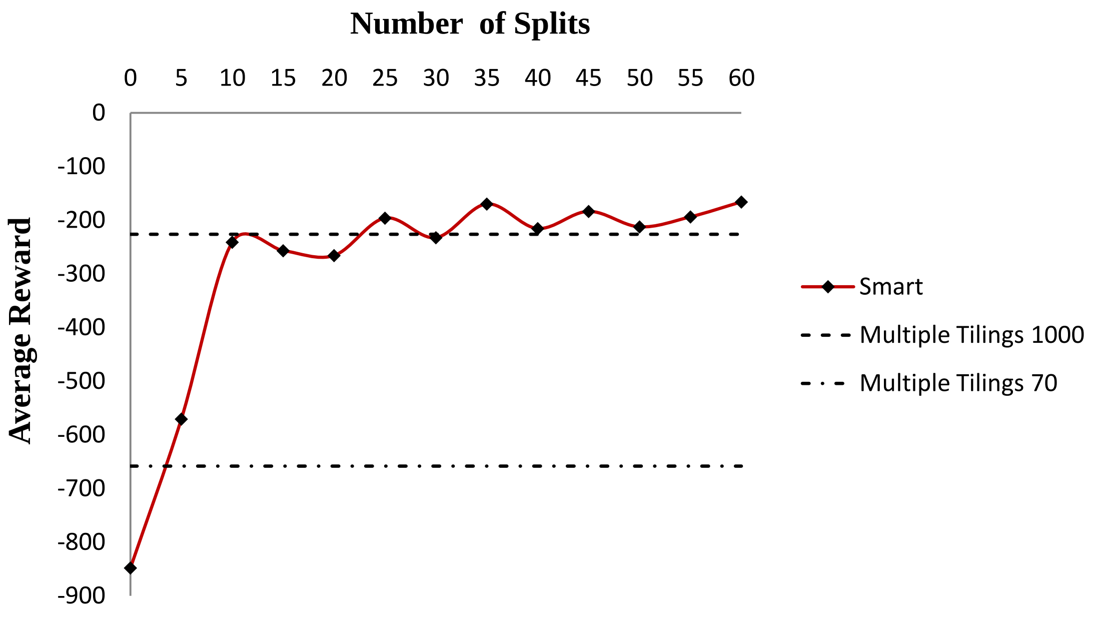


Figure 5.9: The graphical representation of the performance of smart tile coding agent vs multiple tile coding agents in the mountain car environment. The X-axis represents the number of splits. The Y-axis represents the average reward received at each split.

According to the graphs in Figure 5.9, a multiple tiling agent trained with 5000 tiles is able to achieve an average reward of -226.2 which is less than the average reward received by the smart tile coding agent after 60 splits. The performance achieved by a multiple tiling agent with 70 tiles is lot less than the remaining two agents. It suggests that, in the mountain car, a smart tile coding agent can match the performance of a multiple tiling agent.

5.3.2.5 Claim Validation

In Figure 5.10, the performances of various tile coding algorithms discussed in the above sections are represented using graphs to validate the claims raised in section 5.3. I have also included the performance of hand-code algorithms to complete the evaluation. The performance of all the proposed adaptive tile coding methods except smart tile coding failed to exceed the baseline.

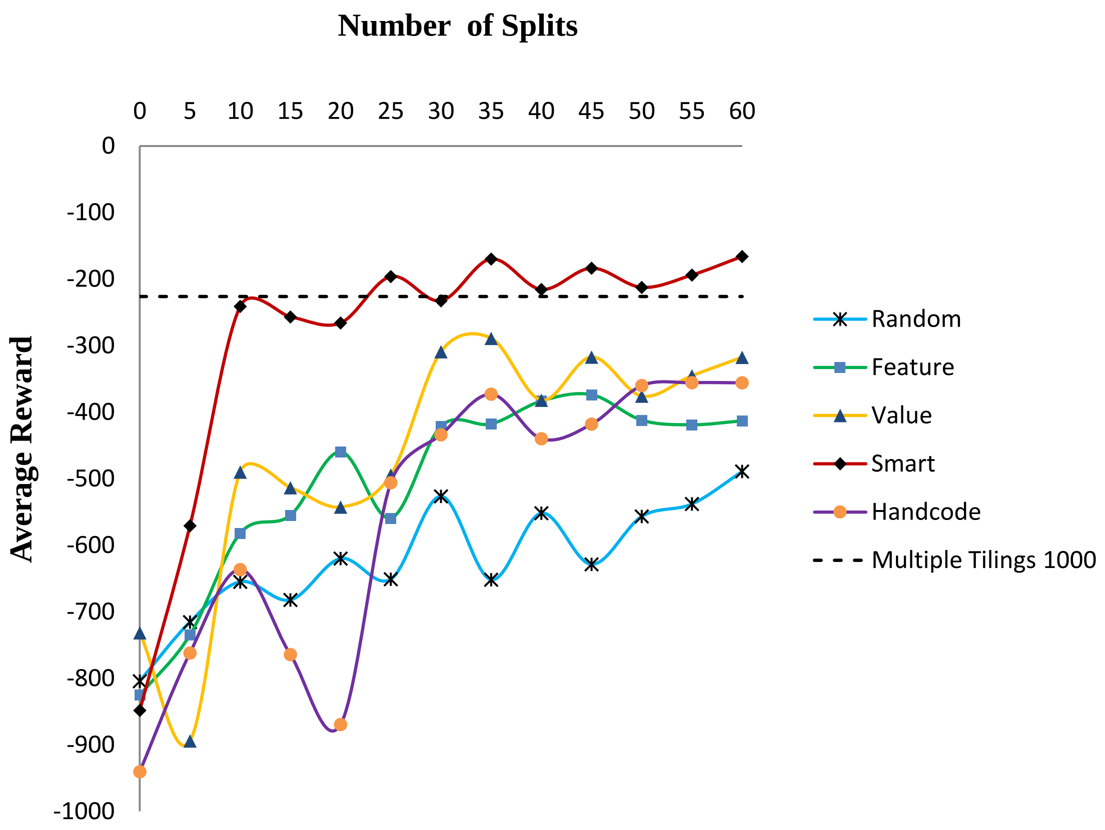


Figure 5.10: The graphical representation of performance of various adaptive tile coding methods, hand-coding method and a multiple tile coding method in the mountain car environment. The X-axis represents the number of splits. The Y-axis represents the average reward received at each split.

It is evident from the above graph that in the mountain car environment:

- Smart tile coding algorithm has outperformed all other adaptive tile coding methods in the mountain car environment.
- Since the performance of the smart tile coding agent is better than the performance of random tile coding and feature-based tiled coding, the claim that a

smart tile coding algorithm outperform pseudo adaptive tile coding algorithms is validated in the mountain car environment.

- Since the smart tile coding agent has outperformed the multiple tilings agent, the claim that a smart tile coding method matches the performance of multiple tile coding method is validated in the mountain car environment.

5.3.3 Cart Pole Balance Experiments

The setup for the cart pole balance environment is given in Section 4.2.3. The outline of each phase and their order of execution are provided in Table 5.1. At the start of an experiment, each feature is divided into two tiles of equal size and the Q-values for all the available tiles are initialized to 0. The sequence of training, testing and splitting is repeated for 60 times to split the tiles 60 times. Since the car pole balance environment has four features where each feature is represented using two tiles, at the start of an experiment the state space is represented using eight tiles and by the end of the experiment the state space is represented using 68 tiles (the original eight features and the resulting split features created from the 60 splits). In this experiment, the same RL parameter values are used for all adaptive tile coding methods and multiple tilings method. Each experiment is repeated for 100 times and the final performance at each split is obtained by taking the average of all the observed performances at the split. The observed performance of the proposed tile coding agents in the cart pole balance environment is shown in Table 5.4. To simplify the data representation, only the performances observed after every 5 splits are shown in the table. The first column in the table contains the split count, the remaining columns represents the average rewards recorded for random tile coding, feature-based tile coding, value-based tile coding, smart tile coding and hand-coding in order.

Number of splits	**Average rewards**				
	Random	**Feature**	**Value**	**Smart**	**Handcoded**
0	**183.89**	**201.94**	**209.73**	**234.96**	**168.76**
5	158.65	146.65	165.68	230.16	196.19
10	131.38	147.2	228.39	292.02	261.95
15	141.4	139.53	562.62	299.39	260.54
20	168.34	124.66	520.21	402.56	252.66
25	156.11	98.41	405.7	500.33	252.44
30	169	128.28	465.2	451.05	259.06
35	172.41	152.52	559.87	510.27	231.02
40	162.48	195.5	696.51	490.21	116.53
45	164.47	188.23	753.35	603.34	188.37
50	195.06	173.13	872.8	678.12	281.32
55	190.08	130.91	955.08	776.26	475.3
60	**211.38**	**151.32**	**966**	**801.88**	**557.31**

Table 5.4: The average rewards received after every 5 splits in various adaptive tile coding methods in the cart pole balance environment. The first column in the table represents the number of splits performed. The remaining columns in the table contain the average rewards received at a split for Random, Feature, Value, Smart and Manual tile codings in order.

5.3.3.1 Random Tile Coding

Methodology

The algorithm used to implement random tile coding method is provided in section 3.2.2. The values of different RL parameters used are: learning rate (α) = 0.01, discount rate (γ) = 0.99, decay rate (λ) = 0.9. The training phase is repeated for 1000 steps and the testing phase is repeated for 200 steps. There is no need for a splitting phase.

Results

The average reward obtained after every five splits is shown under the Random column in Table 5.4. In each row, the values in the Random column represent the performance of the random tile coding agent at a split specified in the first cell. In Figure 5.11 the baselines and the performance of random based tile coding method (from Table 5.4) in the cart pole balance are represented using graphs.

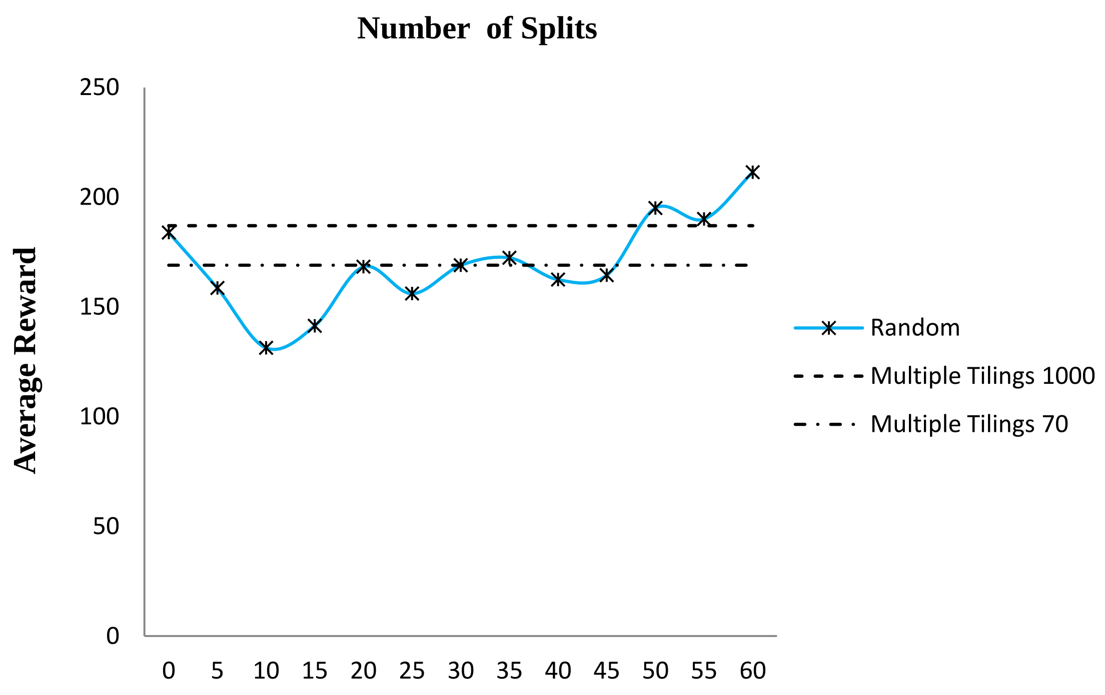


Figure 5.11: The graphical representation of the performance of random tile coding agent vs multiple tile coding agents in the cart pole balance environment. The X-axis represents the number of splits. The Y-axis represents the average reward received at each split.

Analysis

At the end of the first pass in the experiment, the agent received an average reward of 183.89 which is less than 211.38, the average reward received after 60 splits. It indicates that there is a slight improvement in the performance of the random tile coding agent by the end of the experiment. It is evident from the table that the performance of the agent is not consistent over the course of the experiment, more often than not the performance of agent has dipped below the initial performance.

The multiple tilings agent trained with 5000 tiles is able to achieve an average reward of 187; which is less than the average reward received by the random tile coding agent after 60 splits. The performance achieved by a multiple tiling agent with 70 tiles is 169 and it is less than the performance of random tile coding agent. It suggests that, in the cart pole balance, a random based tile coding agent can match the performance of a multiple tiling agent.

5.3.3.2 Feature-based Tile Coding

Methodology

The algorithm used to implement feature-based tile coding method is provided in Section 3.2.3. The values of different RL parameters used are: learning rate (α) = 0.01, discount rate (γ) = 0.99, decay rate (λ) = 0.9. The training phase is repeated for 1000 steps and the testing phase is repeated for 200 steps. There is no need for a splitting phase.

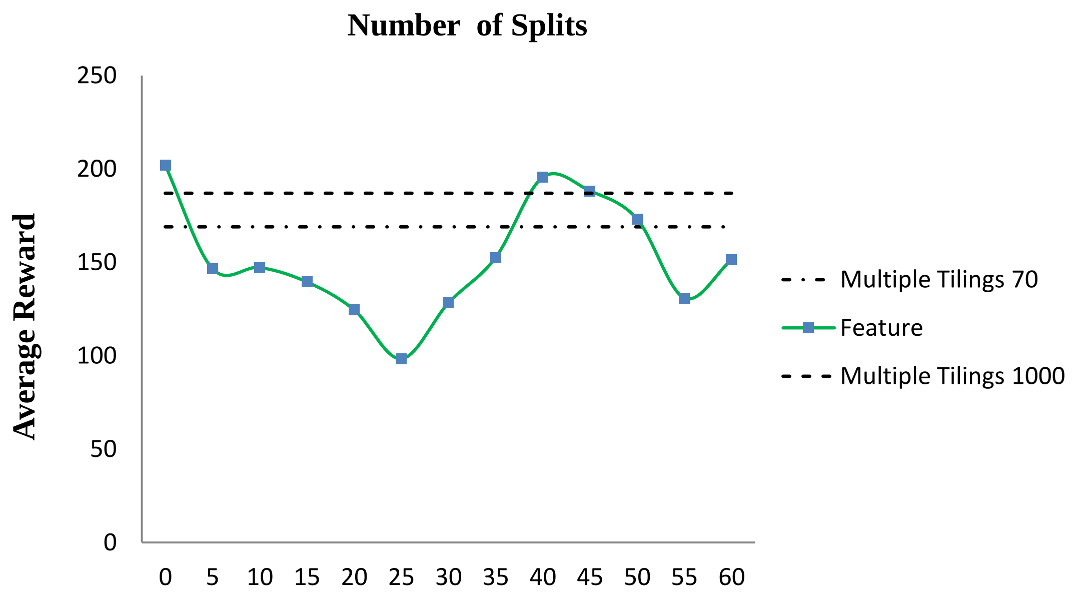


Figure 5.12: The graphical representation of the performance of feature-based tile coding agent vs multiple tile coding agent in the cart pole balance environment. The X-axis represents the number of splits. The Y-axis represents the average reward received at each split.

Results

The average reward obtained after every five splits is shown under the Feature column in Table 5.4. In each row, the values in the Feature column represent the performance of the feature tile coding agent at a split specified in the first cell. In Figure 5.12, the baselines and the performance of feature-based tile coding method (from Table 5.4) in car pole balance are represented using graphs.

Analysis

At the start of the experiment, before performing any split, the agent achieved an average reward of 201.94 which is greater than 151.32, the average reward received after 60 splits. It indicates that, in the cart pole balance environment, there is no considerable improvement but a dip in the performance of the feature-based tile coding agent by the end of experiment. It is also evident from the table that the performance of the agent is inconsistent for most part of the experiment.

According to the graphs given in Figure 5.12, the average reward received (151.32) by the feature-based tile coding agent after 60 splits is less than the base line (187). In fact, the performance achieved by a multiple tiling agent with 70 tiles (169) is lot lesser than the rest of the two agents. It suggests that, in the cart pole balance environment, a feature-based tile coding agent cannot match the performance of a multiple tiling agent.

5.3.3.3 Value-based Tile Coding

Methodology

The algorithm used to implement value-based tile coding method is provided in Section 3.2.4. The values of different RL parameters used are: learning rate (α) = 0.01, discount rate (γ) = 0.99 decay rate (λ) $=$ 0.9. The training phase is repeated for 1000 steps. The testing phase and splitting phase is repeated for 200 steps.

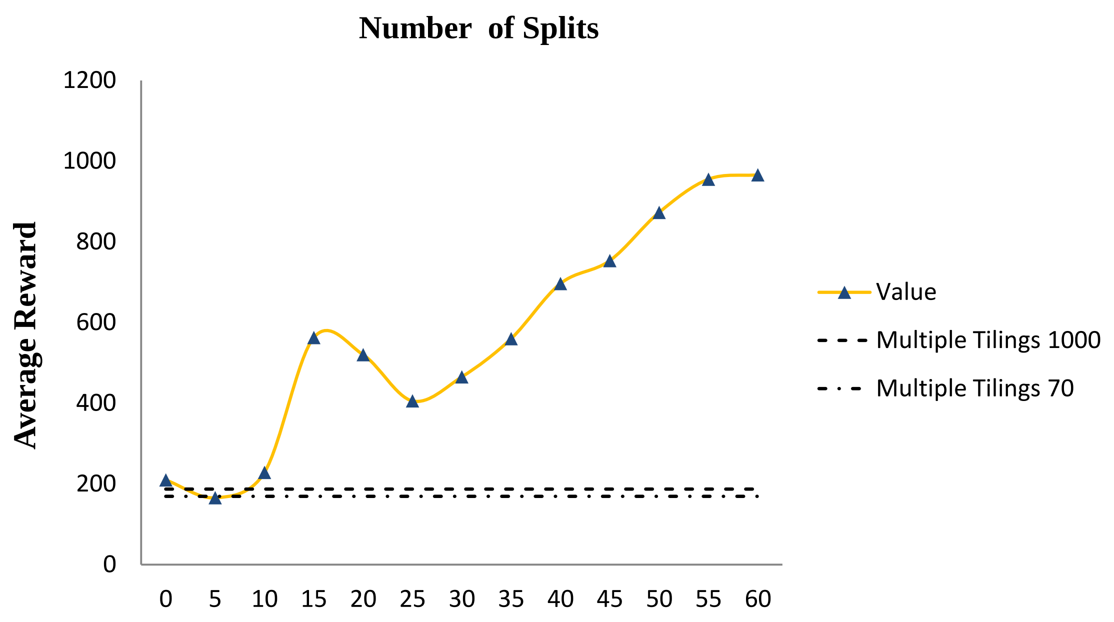


Figure 5.13: The graphical representation of the performance of value-based tile coding agent vs multiple tile coding agents in the pole balance environment. The X-axis represents the number of splits. The Y-axis represents the average reward received at each split.

Results

The average reward obtained after every five splits is shown under the Value column in Table 5.4. In each row, the values in the Value column represent the performance of the value tile coding agent at a split specified in the first cell. In Figure 5.13 the baselines and the performance of value-based tile coding method (from Table 5.4) in pole balance are represented using graphs.

Analysis

At the start of the experiment, before performing any split, the agent achieved an average reward of 209.73 which is less than 966, the average reward received after 60 splits. It proves that, in the cart pole balance environment, there is a considerable improvement in the performance of the value-based tile coding agent towards the end of experiment. It is evident from the table that performance improvement rate of the agent is positive consistently, except at the start of the experiment and in between 20^{th} and 30^{th}split. The value-based tile coding agent was able to improve the performance at a better rate when compared to the rest of the adaptive tile coding methods.

According to the graphs in Figure 5.13, the baseline with value equal to 187 is lot less than the average reward received by the value-based tile coding agent after 60 splits. The performance achieved by a multiple tiling agent with 70 tiles is also lot less than the value-based tile coding agent. It indicates that in the care pole balance environment a value-based tile coding agent can not only match but outperform a multiple tiling agent.

5.3.3.4 Smart Tile Coding

Methodology

The algorithm used to implement smart tile coding method is provided in Section 3.2.5. The values of different RL parameters used are: learning rate (α) = 0.01, discount rate (γ) = 0.99, decay rate $(\lambda) = 0.9$. The training phase is repeated for 1000 steps. The testing phase and splitting phase are repeated for 200 steps.

Results

The average reward obtained after every five splits is shown under the Smart column in Table 5.4. In each row, the values in the Smart column represent the performance of the smart tile coding agent at a split specified in the first cell. In Figure 5.14 the baselines and the performance of smart tile coding method (from Table 5.4) in the cart pole balance are represented using graphs.

Analysis

At the start of the experiment, before performing any split, the agent achieved an average reward of 234.96 which is less than 801.88, the average reward received after 60 splits. It proves that, in the cart pole balance environment, there is a considerable improvement in the performance of the smart tile coding agent by the end of experiment. The performance improvement rate of agent is consistently positive at most of the split intervals except for a split interval after 25th and 35th split, where a small dip in the performance is observed.

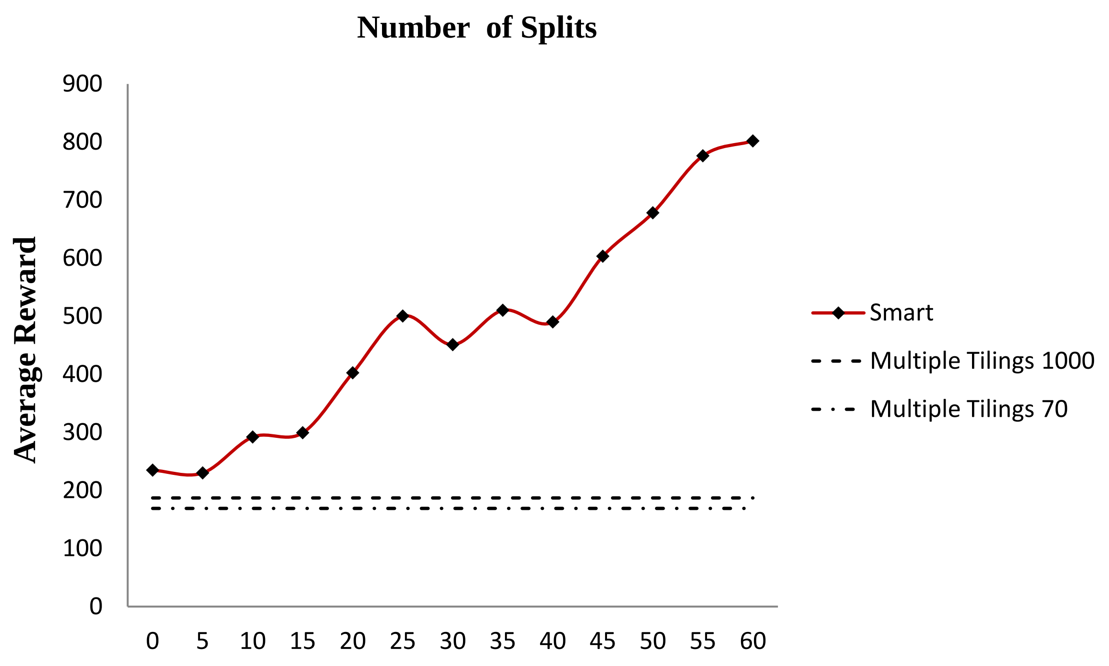


Figure 5.14: The graphical representation of the performance of smart tile coding agent vs multiple tile coding agents in the cart pole balance environment. The X-axis represents the number of splits. The Y-axis represents the average reward received at each split.

According to the graphs in Figure 5.14, a multiple tiling agent trained with 5000 tiles is able to achieve an average reward of 187 which is lot less than the average reward received by the smart tile coding agent after 60 splits. The performance achieved by a multiple tiling agent with 70 tiles is lot less than the remaining two agents. It suggests that, in the pole balance environment, a smart tile coding agent can not only match but outperform the performance of a multiple tiling agent.

5.3.3.5 Claim Validation

In Figure 5.15, the performances of various tile coding algorithms discussed in the above sections are represented using graphs to validate the claims raised in section 5.3, for the cart pole balance environment. I have also included the performance of hand-code algorithms to complete the evaluation. The performances of all adaptive tile coding methods except feature-based tile coding were able to exceed the baseline.

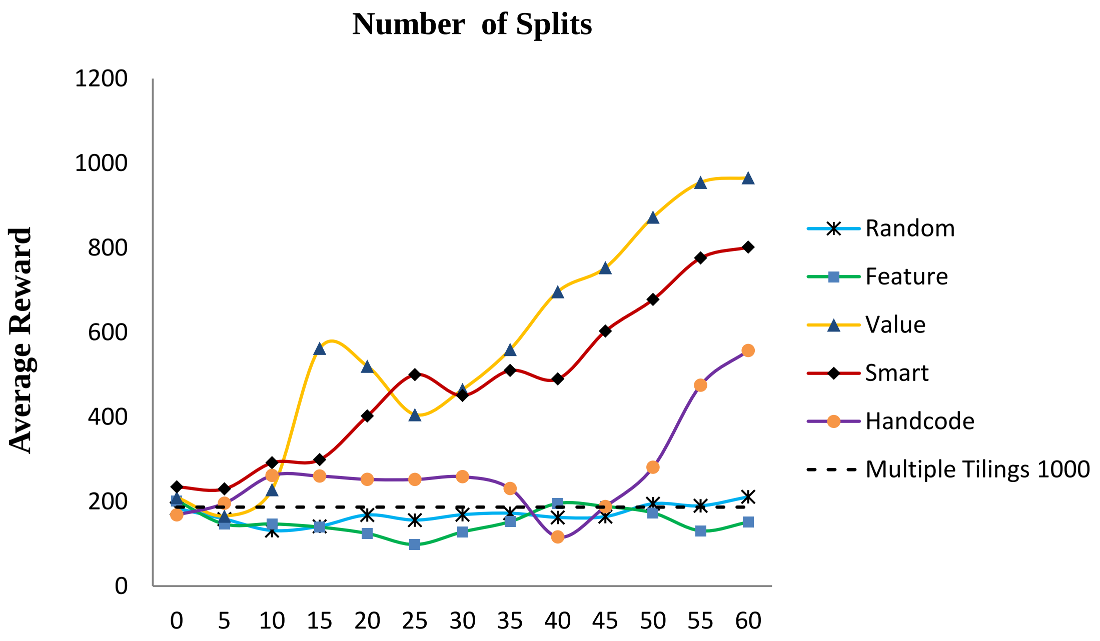


Figure 5.15: The graphical representation of the performance of various adaptive tile coding methods, hand-coding method and multiple tilings method in the cart pole balance environment. The X-axis represents the number of splits. The Y-axis represents the average reward received at each split.

It is evident from the graphs that, in the cart pole balance environment:

- Value-based tile coding algorithm has outperformed all other adaptive tile coding methods in the cart pole balance environment.

- Since the performance of the smart tile coding agent is better than the performance of random tile coding and feature-based tile coding, the claim that a smart tile coding algorithm outperform pseudo adaptive tile coding algorithms is validated in the cart pole balance environment.
- Since the smart tile coding agent has outperformed the multiple tilings agent, the claim that a smart tile coding method matches the performance of multiple tile coding method is validated in the cart pole balance environment.
- The poor performance of the multiple tilings agent validates the claim that the multiple tilings method performance drops with increase in dimensions, and is not suitable for environments with higher-dimensions.
- The good performance of smart tile coding method in the cart pole balance environment validates the claim that an adaptive tile coding method can perform well even in relatively higher-dimensional environments.

5.3.4 Summary

The performances of various adaptive tile coding algorithms are discussed in the previous sections. In this section, I will give an overview of how different adaptive coding methods behaved in different test environments and try to validate the claim that adaptive tile coding is an efficient tile coding mechanism which can match the performance of multiple tilings method using a lot less number of tiles. In Table 5.5, the presence of desirable attributes is checked for random tile coding method in different testing environments. According to the table, in majority of the cases, random tile coding method has failed to: improve the performance, maintain consistency in performance improvement rate, and match the performance of multiple tilings. In Table 5.6, the presence of desirable attributes is checked for feature-based tile coding method in different testing environments. According to the table, in majority of the cases, feature-based tile coding method is able to improve the performance considerably but it failed to maintain the consistency in performance improvement rate and to match the performance of multiple tilings method.

Attributes	**Puddle world**	**Mountain car**	**Pole balance**
Considerable improvement in overall performance after splitting	no	yes	no
Consistent positive improvement of performance	no	no	no
Better performance compared to mtiling_5000	no	yes	yes
Better performance compared to mtiling_70	yes	yes	yes
Better performance compared to all other tile coding methods	no	no	no

Table 5.5: Observed attributes values for random tile coding.

Attributes	**Puddle world**	**Mountain car**	**Pole balance**
Considerable improvement in overall performance after splitting	yes	yes	no
Consistent positive improvement of performance	no	yes	no
Better performance compared to mtiling_5000	yes	no	no
Better performance compared to mtiling_70	yes	yes	no
Better performance compared to all other tile coding methods	no	no	no

Table 5.6: Observed attributes values for feature-based tile coding.

In Table 5.7, the presence of desirable attributes is checked for value-based tile coding method in different testing environments. According to the table, in majority of the cases, value-based tile coding method is able to improve the performance considerably but failed to maintain consistency in performance improvement rate and to match the performance of multiple tilings.

Attributes	**Puddle world**	**Mountain car**	**Pole balance**
Considerable improvement in overall performance after splitting	no	yes	yes
Consistent positive improvement of performance	no	no	yes
Better performance compared to mtiling_5000	no	no	yes
Better performance compared to mtiling_70	yes	yes	yes
Better performance compared to pseudo adaptive tile coding	no	yes	yes
Better performance compared to all other tile coding methods	no	no	yes

Table 5.7: Observed attributes values for value-based tile coding.

Attributes	**Puddle world**	**Mountain car**	**Pole balance**
Considerable improvement in overall performance after splitting	yes	yes	yes
Consistent positive improvement of performance	yes	yes	yes
Better performance compared to mtiling_5000	yes	yes	yes
Better performance compared to mtiling_70	yes	yes	yes
Better performance compared to pseudo adaptive tile coding	yes	yes	yes
Better performance compared to all other tile coding methods	yes	yes	no

Table 5.8: Observed attributes values for smart tile coding.

In Table 5.8, the presence of desirable attributes is checked for smart tile coding method in different testing environments. According to the table, in all the cases, the smart based tile coding method is able to improve the performance considerably, maintain consistency in performance improvement rate, and outperform the performance of multiple tilings method. Hence it can be safely deduced from Table 5.8 that smart tile coding algorithm is better than all other proposed adaptive tile coding methods (as other tile coding methods failed to satisfy all attributes) and can match the multiple tilings method.

Chapter 6

Related and Future Work

Applying the function approximation methods to generalize the value functions of a state space in an RL environment allows RL to work on problems with continuous and large state spaces. Various function approximation methods are available to generalize the value functions of a state space. The majority of our current work is to automate one such function approximation method called tile coding. The most related work to ours is the work of Whiteson et al. (2007) on adaptive tile coding for value function approximation. They used a different policy from ours to decide when to split and where to split. In our work, we split a tile for every fixed number of steps; they split a point only if the observed Bellman error of the last updated tile is not less than the current minimum Bellman error of the tile for a fixed number of steps which allows the agent to use the updates optimally for generalization. We split at a point with maximum difference between the observed Q-value deviations the tile on either side of the point; they split at a point with maximum difference in weights of the potential sub tiles formed by splitting at the point. They are only considering the mid points of local range for each dimension in a tile as potential split points which might requires extra steps to find the exact point where the difference is substantial. The work of Sherstov and Stone (2005) has proved that the rate of change in the level of generalization has an effect on the performance in tile coding. Munos and Moore (2002) addressed variable resolution state abstraction for state spaces involving continuous time and space. They used Kd-trie to represent the space and a top-down approach with different criteria to split the cells.

The proposed adaptive tile coding methods can be extended to find an efficient solution for RL problems involving higher-dimensional states. Performing splits at fixed intervals, as in this work, might slow down the overall generalization process (if newly formed sub tiles learn the near optimal value functions quickly) or limit the area of generalization (if the Bellman error of newly form sub tiles is still high), depending on the previous split. The proposed methods can be modified to consider the generalization achieved by the sub tiles formed after previous split point before performing a new split. For every environment, there is a threshold value for the number of splits to perform; beyond that point new splits might reduce the generalization. In this work, I performed a fixed number of splits for each tile coding method in all environments. For better generalization, the number of splits performed can be changed to a variable which depends on the environment and the tile coding method. Implementing multi-dimensional representation of a state space would capture the features in a better way when compared to the single dimensional representation used in this work.

Chapter 7

Summary and Conclusions

In this thesis, I implemented four different adaptive tile coding methods to generalize the value functions of an RL state space. The proposed adaptive tile-coding methods use a random tile generator, the number of states represented by features, the frequencies of observed features, and the difference between the deviations of predicted value functions from Monte Carlo estimates for observed states in each tile. The above proposed methods are tested on the following RL environments: the puddle world problem, the mountain car problem and the cart-pole balance problem using RL-Glue. The performances of these methods in the above tests are evaluated against the baselines obtained from multiple tilings method.

The random tile coding method had inconsistent performance in the experiments which suggests that random splitting of the tiles will result in a poor generalization in many cases. The poor performance of feature-size based tile coding in the cart-pole problem suggests that giving equal priority to all the dimensions in a state-space might slow down the generalization for high-dimension state spaces. The performance of value-based tile coding is better for environments with high number of observed states at critical regions. The performance of smart tile coding is consistent in all three environments and it proves that the difference in Q-values deviation is a valid criteria in splitting the tiles. After observing the results of all the experiments, I conclude that an efficient adaptive tile coding method with far less number of tiles can match the performance of multiple tilings method in a RL environment.

Bibliography

[Andre and Russell, 2001] Andre, D., Russell, S. J. 2001. *Programmable reinforcement learning agents*. In T. K. Leen, T. G. Dietterich, and V.Tresp (Eds.), Advances in neural information processing systems (Vol. 13, pp. 1019–1025). Cambridge, MA: MIT Press.

[Andre and Russell, 2002] Andre, D., Russell, S. J. 2002. *State abstraction for programmable reinforcement learning agents*. In R. Dechter, M. Kearns and R. S. Sutton (Eds.), Proceedings of the 18th National Conference on Artificial Intelligence Mento Park (pp. 119–125). CA: AAAL Press.

[Baird, 1995] Baird, L. 1995. *Residual algorithms: Reinforcement learning with function approximation*. In proceedings of the Twelfth International Conference on Machine Learning, 30–37. Morgan Kaufmann.

[Bradtke and Duff, 1995] Bradtke, S. J., and Duff, M. O. 1995. *Reinforcement learning methods for continuous-time Markov decision problems*. In G. Tesauro, D. Touretzky, and T. Leem (Eds.), Advances in neural information processing systems (Vol. 7, pp. 393–400). San Mateo, CA: Morgan Kaufmann.

[Dietterich, 2000] Dietterich, T. G. 2000. *Hierarchical reinforcement learning with the maxq value function decomposition*. Journal of Artificial Intelligence Research, 13, 227–303.

[Gordon, 2001] Gordon, G. 2001. *Reinforcement learning with function approximation converges to a region*. In T. K. Leen, T. G. Dietterich, and V. Tresp (Eds.), Advances in neural information processing systems (Vol. 13, pp. 1040–1046). Cambridge, MA: The MIT Press.

[Kaelbling, Littman and Moore, 1996] Kaelbling, L. P., Littman, M.L., Moore, A.W. 1996. *Reinforcement Learning: A Survey*. Journal of Artificial Intelligence Research, Vol 4, (1996), 237-285.

[Koller and Parr, 1999] Koller, D., and Parr, R. 1999. *Computing factored value functions for policies in structured MDPs*. In T. Dean (Ed.), Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99) (pp1332–1339). Morgan Kaufmann.

[Lanzi, Loiacono and Wilson, 2006] Lanzi, P. L., Loiacono, D., Wilson, S. W., and Goldberg, D. E. 2006. *Classifier prediction based on tile coding*. In proceedings of the Eighth Annual Conference on Genetic and Evolutionary Computation, 1497–1504.

[Mahadevan, 2005] Mahadevan, S. 2005. *Automating value function approximation using global state space analysis*. In proceedings of the Twentieth National Conference on Artificial Intelligence.

[Mitchell, 1997] Mitchell, T. 1997. *Machine Learning*, McGraw Hill.

[Moore and Atkeson, 1995] Moore, A. W., and Atkeson, C. G. 1995. *The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces*. Machine Learning 21(3):199–233.

[Neal, 1993] Neal, R. 1993. *Probabilistic Inference Using Markov Chain Monte Carlo Methods*. Technical Report, University of Toronto.

[Stone, Sutton and Kuhlmann, 2005] Stone, P., Sutton, R., and Kuhlmann, G. 2005. *Reinforcement learning in robocup-soccer keepaway*. Adaptive Behavior 13(3):165–188.

[Sutton, 1996] Sutton, R. 1996. *Generalization in reinforcement learning: Successful examples using sparse coarse coding*. Advances in Neural Information Processing Systems 8, 1038–1044.

[Sutton and Barto, 1998] Sutton, R. S., and Barto, A. G. 1998. *Reinforcement Learning: An Introduction*. Cambridge, Massachussets: MIT Press.

[Sutton et al, 2000] Sutton, R., McAllester, D., Singh, S., and Mansour, Y. 2000. *Policy gradient methods for reinforcement learning with function approximation*. In S. A. Solla, T. K. Leen, and K. R. Muller (Eds.) Advances in neural information processing systems, (Vol. 12, pp. 1057–1063). Cambridge, MA: The MIT Press.

[Tsitsiklis and Van Roy, 1997] Tsitsiklis, J. N., and Van Roy, B. 1997. *An analysis of temporal-difference learning with function approximation.* IEEE Transactions on Automatic Control, 42, 674–690.

[Watkins, 1989] Watkins, C. 1989. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge.

[Whiteson and Stone, 2006] Whiteson, S., and Stone, P. 2006. *Evolutionary function approximation for reinforcement learning*. Journal of Machine Learning Research 7(May): 877–917.