Implementation of Dyna-Q, One Step Actor Critic, Prioritized Sweeping

Rahul Seetharaman, Harsha Kanaka Eswar Gudipudi
{rseetharaman, hgudipudi}@umass.edu

Abstract

The project implements the Dyna-Q, One Step Actor Critic, and Prioritized Sweeping algorithms. The domains explored are 687-GridWorld, Cart Pole, and Mountain Car. One Step Actor Critic leverages a neural network (written in PyTorch) in order to predict value function estimates and policy probabilities. For the Dyna-Q and Prioritized Sweeping implementations, we make use of tile coding and discretization to map the continuous state spaces in Cart Pole, and Mountain Car into discrete state spaces. We start with an overview of each algorithm, intuitions, pseudocode, and finally we explain the design decisions behind our implementations followed by an analysis with plots such as rewards over time, number of episodes vs number of actions taken upto a point, and so on.

1 Dyna-Q

1.1 Overview

Dyna-Q belongs to a family of algorithms called Model-based RL, wherein the agent learns a model of the environment in addition to learning the optimal action value functions and policy. Dyna-Q involves a two step process, where an agent learns by interactions with the environment, followed by a second step where the agent iteratively learns a model of the environment, and uses this model to learn from *simulated* interactions. So it involves both planning and learning by actual interactions.

1.2 Algorithm Pseudocode

Algorithm 1: Dyna Q

```
Initialize $Q(s,a)$ and $Model(s,a)$ randomly $\forall s \in S, \forall a \in A$
history ← {}
while True do
    S ← (current non terminal state)
    done ← False
    while $done \neq False$ do
        Take action A, observe next state $s'$ and reward r.
        $Q(s,a) \leftarrow Q(s,a) + lr * (r + \gamma * \max_{a'} Q(s',a') - Q(s,a))$
        $Model(s,a) \leftarrow r, s'$
        Add $(s,a,r,s')$ to history
        if $s'$ is terminal: done ← True
        for $i = 1$ to $max\_history\_length$ do
            $s \leftarrow$ randomly sampled state
            $a \leftarrow$ action taken in $s$
            $s', r \leftarrow Model(s,a)$
            $Q(s,a) \leftarrow Q(s,a) + lr * (r + \gamma * \max_{a'} Q(s',a') - Q(s,a))$
        end for
    end while
end while
```

1.3 Explanation

In Dyna Q, the agent learns in two step process. In the first step, it interacts with the environment, and updates it's q-values, like most RL algorithms. It also *records* the interactions, with rewards, next states, current state, current action, etc. It uses these interactions to build a model of the environment. Using the built model of the environments, it *simulates* interactions, and updates its q-values based on the simulated experiences. Dyna-Q is particularly useful in scenarios where actual interactions of agent and environment are expensive, for eg: an RL agent that learns to do stock trading. Real interactions can potentially involves loss of millions of dollars, while the agent is learning from mistakes and improving its policy. So, in this case building a model of the environment and using simulated experiences is desirable.

1.4 Design Decisions

For the 687-Gridworld environment, which consist of discrete state and discrete action spaces, the algorithm lended itself very well and no design tradeoffs had to be incurred in terms of discretization, etc. However, for the CartPole and MountainCar domains, where the state space is continous, Tile Coding was utilized to discretize the state space. With the help of tile coding, a unique set of tiles or regions in a hash table is identified for each state in the continuous space, and q-values updates are made by retrieving the uniquely mapped hash locations for each state-action pair. The tiles3 module implemented by Sutton and Barto is used as a tile coding utility to convert continuous state spaces into discrete action space.

1.5 Results

1.5.1 687-GridWorld

The performance of Dyna-Q on 687-GridWorld is illustrated with the help of two plots - one showing the mean squared error estimate against the value function identified by value iteration, and the other showing the effectiveness of the agent, in terms of number of episodes completed as the number of actions increases.

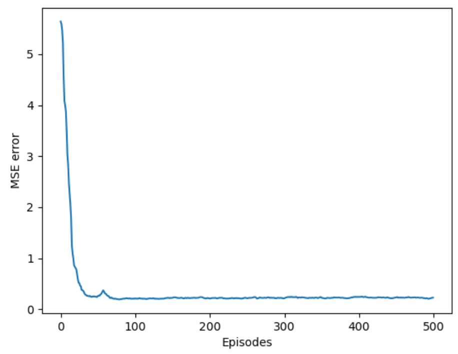


Figure 1: Plot of Mean Squared Error across 500 episodes

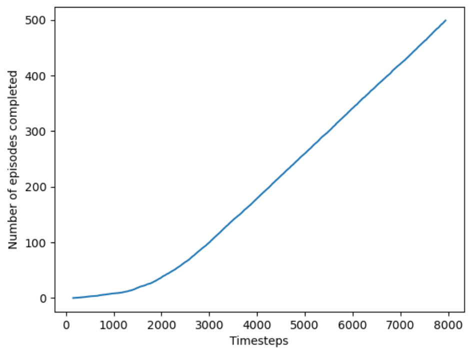


Figure 2: Number of episodes completed vs number of actions taken

Figure 1 shows the mean squared error across 500 episodes averaged across 20 runs. It is observed that the mean squared error rapidly decreases after approximately 50 episodes. This shows that the agent quickly learns the optimal set of actions for the given states. Figure 2 shows the number of episodes completed against the number of actions taken upto the given point. We observe a rapid increase in number of episodes completed roughly again at 50 episodes, as the agent picks up the optimal actions and starts to go to the goal state in lesser steps.

Hyperparameter Tuning: The following set of hyperparameters was experimented with.

1. **Learning rate/Step Size** α: The best step size was found to be 0.1. Ideally, any schedule of learning rate/step size, where the ϵ decay is faster than the α decay should be preferred, as this means the agent will always learn from observed rewards. If ϵ decays slower than α, this means that at some point, random actions may be given precedence and the agent never actually learns. The ideal step size is observed to be 0.1. Another step size that worked well in practice similar to Q-Learning and SARSA implementations was $\frac{0.25}{sqrt(i)+1}$, where i is the number of episodes completed upto that point.

2. **Epsilon (Exploration parameter)** ϵ: The ideal schedule of epsilon is found to be a starting value of 0.8, and decaying by a factor of 2, every 100 episodes. This encourages early exploration, and the exponential decay ensures the agent exploits the learned policy to get maximum rewards. As observed, the agent learns the optimal policy to a large extent fairly quickly.

3. **History context (specific to Dyna Q)**: The ideal history context was found to be 10. This means for 10 occurrences of (s, a) (state, action) in the history taken, the learned model is inferred to produce a possible next state, and the update equation $Q(s, a) \leftarrow Q(s, a) + lr * (r + \gamma * \max_{a'} Q(s', a') - Q(s, a))$ is performed. The model in this case is a matrix that counts occurrences of (s, a, s'). The probabilities are obtained by normalizing over the matrix entries, and the next state for a given (s, a) is sampled accordingly. Over time, this matrix builds up a Maximum Likelihood estimate of the transition probabilities as the agent encounters more and more episodes.

4. **Gamma - γ (Discount parameter)** The discount factor that worked best was found to be 0.9. This value of γ is also observed to work well for most algorithms including SARSA, Q-Learning, Prioritized Sweep, etc. for this domain.

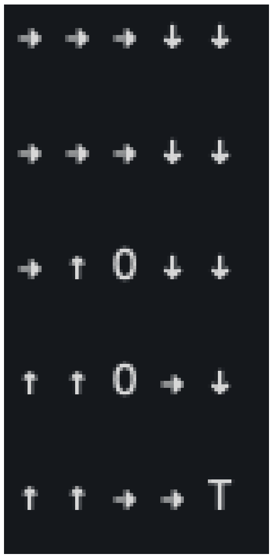


Figure 3: Policy learnt by Dyna Q

From Figure 3, we observe that Dyna-Q has mostly learnt the optimal policy obtainable through value iteration. It differs from the value iteration obtained policy in one cell (2,0) where it executes RIGHT instead of UP. However, the sequence of actions still lead it ultimately to the goal state in the same number of steps from (2,0). This owes to the fact we are able to learn a strong maximum likelihood model of the transition probabilities by allowing the agent to run many episodes, while at the same time iteratively improving our Dyna-Q model.

1.5.2 Cart Pole

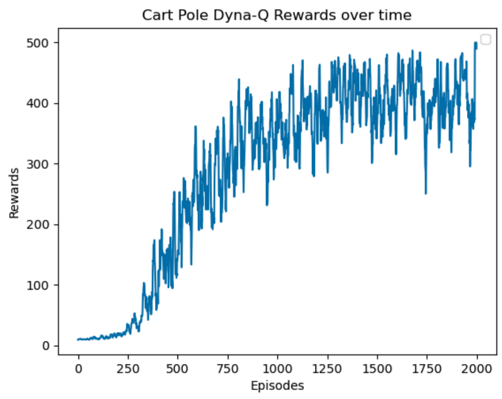


Figure 4: Cartpole - Rewards over time

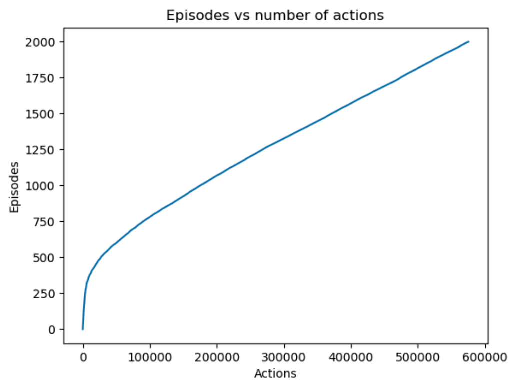


Figure 5: Episodes completed vs number of actions

Figure 4 shows the rewards obtained in the CartPole domain over a period of 500 episodes. As the number of episodes increases, the agent is found to get increasingly better at the task. Albeit a bit of variance, the agent still learns the optimal policy and will only get better with more episodes. Figure 2 shows the number of episodes completed against the number of actions. The number of episodes plateaus over time. This indicates that, the agent is learning a better policy, and as a result, balances for a longer time, and takes more actions to complete an episode (which eventually terminates after 500 actions).

Hyperparameter tuning: The following set of hyperparameters were experimented with,

1. **Learning rate/Step Size** α: The best step size was found to be 0.1. Ideally, any schedule of learning rate/step size, where the ϵ decay is faster than the α decay should be preferred, as this means the agent will always learn from observed rewards. If ϵ decays slower than α, this means that at some point, random actions may be given precedence and the agent never actually learns. The ideal step size is observed to be 0.1. Another step size that worked well in practice similar to Q-Learning and SARSA implementations was $\frac{0.25}{sqrt(i)+1}$, where i is the number of episodes completed upto that point (which is a gradual square root decay).
2. **Epsilon (Exploration parameter)** ϵ: The ideal schedule of epsilon is found to be a starting value of 0.4, and decaying by a factor of 2, every 100 episodes. This encourages early exploration, and the exponential decay ensures the agent exploits the learned policy to get maximum rewards. As observed, the agent learns the optimal policy to a large extent fairly quickly.
3. **History context (specific to Dyna Q)**: The ideal history context was found to be 20. This means for 20 occurrences of (s, a) (state, action) in the history taken, the learned model is inferred to produce a possible next state, and the update equation $Q(s, a) \leftarrow Q(s, a) + lr * (r + \gamma * \max_{a'} Q(s', a') - Q(s, a))$ is performed. The model in this case is very deterministic, as a given state and action leads to only possible next state. There are two possible actions, and the best q value is chosen amongst the left and right actions. With the next state being deterministic, it is recorded as part of the history, and this encompasses the model for Dyna-Q.
4. **Gamma - γ (Discount parameter)** The discount factor that worked best was found to be 0.99. This γ is found to work well for continuous domains in Dyna-Q and Prioritized Sweeping.

The state space is discretized using tile coding. As a result, each point in the continuous state space in the form of $(x, v, \omega, \dot{\omega})$ is mapped into a tile coding space with 8 tiles representing this point in the state space.

The Index Hash Table size is kept to be 2^{23}, as an upper bound to avoid any hash collisions. 8 tiles are used to represent $(x, v, \omega, \dot{\omega})$. These 8 tiles represent 8 unique indices in a hash table. The update happens for each of these 8 tiles during the updation of q-values. The scaling factor used is 32 for each of the values in the state space. Each point is represented uniquely in a $32 \times 32 \times 32 \times 32$ grid, with 8 tiles for each point. This makes the total dimensionality of $32 \times 32 \times 32 \times 32 \times 8$, i.e., 2^{23} as mentioned. A detailed overview of Tile Coding is given in 4.

1.5.3 Mountain Car

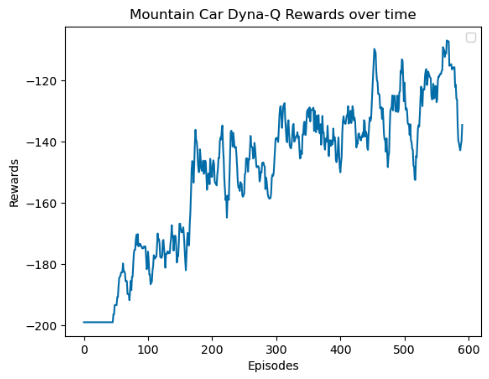


Figure 6: Mountain Car domain - rewards over time

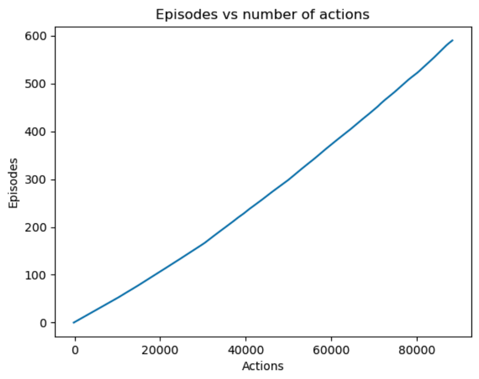


Figure 7: Episodes versus actions over time

It is observed from Figure 6 that the negative reward decreases having trained the agent for 600 episodes. It reaches a reward of -120, starting from -200. The environment gives a negative reward for every state apart from the goal. This penalizes the agent to stay longer in the environment. We can also see from 7 that the number of episodes covered by the agent increases steadily. This is because the agent is penalized to stay long in the environment due to the negative reward, and has to exit quickly, thus it learns to finish an episode quicker by reaching the goal state faster.

Hyperparameter tuning: The following set of hyperparameters were experimented with,

1. **Learning rate/Step Size** α: The best step size was found to be 0.1. Ideally, any schedule of learning rate/step size, where the ϵ decay is faster than the α decay should be preferred, as this means the agent will always learn from observed rewards. If ϵ decays slower than α, this means that at some point, random actions may be given precedence and the agent never actually learns. The ideal step size is observed to be 0.1. Another step size that worked well in practice similar to Q-Learning and SARSA implementations was $\frac{0.5}{sqrt(i)+1}$, where i is the number of episodes completed upto that point.
2. **Epsilon (Exploration parameter)** ϵ: Very similar to the CartPole domain, the ideal schedule of epsilon is found to be a starting value of 0.4, and decaying by a factor of 2, every

100 episodes. This encourages early exploration, and the exponential decay ensures the agent exploits the learned policy to get maximum rewards. As observed, the agent learns the optimal policy to a large extent fairly quickly.

3. **History context (specific to Dyna Q)**: The ideal history context was found to be 10. This means for 10 occurrences of (s, a) (state, action) in the history taken, the learned model is inferred to produce a possible next state, and the update equation $Q(s, a) \leftarrow Q(s, a) + lr * (r + \gamma * \max_{a'} Q(s', a') - Q(s, a))$ is performed. The model in this case is very deterministic, as a given state and action leads to only possible next state. There are three possible actions (accelerate left, stay still, accelerate right), and the best q value is chosen amongst the three possible actions. With the next state being deterministic, it is recorded as part of the history, and this encompasses the model for Dyna-Q in this case too.
4. **Gamma - γ (Discount parameter)** The discount factor that worked best was found to be 0.9. This value of γ is also observed to work well for most algorithms including SARSA, Q-Learning, Prioritized Sweep, etc. for this domain.

The state space is discretized using tile coding. As a result, each point in the continuous state space in the form of (x, v) is mapped into a tile coding space with 8 tiles representing this point in the state space.

The Index Hash Table size is kept to be 2^{17}, as an upper bound to avoid any hash collisions. 8 tiles are used to represent (x, v). These 8 tiles represent 8 unique indices in a hash table. The update happens for each of these 8 tiles during updation of q-values. The scaling factor used is 16 for each of the values in the state space. Each point is represented uniquely in a 16*16*64*8 grid, with 8 tiles for each point. This makes the total dimensionality of 16*16*64*8, i.e. 2^{17} as mentioned.

2 One-step Actor-Critic

2.1 Overview

One-step Actor-Critic is a model-free reinforcement learning algorithm designed to directly learn a parameterized policy for action selection and a state-value function for evaluating states. Unlike model-based approaches, such as Dyna-Q, which involve learning an explicit model of the environment, one-step Actor-Critic focuses on immediate interactions with the environment. In a single step, the agent selects an action based on the current state, receives immediate feedback in the form of a reward and the next state, and updates both the policy and the value function. This algorithm does not require the construction of an explicit model, making it suitable for scenarios where acquiring an accurate model is challenging or computationally expensive. Its emphasis on real-time learning from direct experiences makes it particularly well-suited for dynamic environments and applications with unpredictable changes.

2.2 Explanation

One-step Actor-Critic is a concise and direct reinforcement learning algorithm that operates in a single step without involving the construction of an explicit model of the environment. In each step, the agent interacts with the environment, selecting actions based on its policy, and updates both the policy and the state-value function in response to immediate feedback, including rewards and next states. This straightforward approach makes it well-suited for scenarios where real-time interactions are feasible, and there is no need for a simulated model of the environment. The algorithm excels in adapting to dynamic environments, continually refining its policy and value function based on direct experiences.

2.3 Design Decisions

For the 687-Gridworld environment, which consist of discrete state and discrete action spaces, the algorithm lended itself very well and no design tradeoffs had to be incurred in terms of discretization, etc. However, for the CartPole and MountainCar domains, where the state space is continuous, Fourier basis was utilized to represent the state features. With the help of pytorch, created two

Algorithm 2: One-step Actor-Critic (episodic)

```
Input: a differentiable policy parameterization $\pi(a \mid s, \boldsymbol{\theta})$
Input: a differentiable state-value parameterization $\hat{v}(s, \mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0, \alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$
Repeat forever:
    Initialize $S$ (first state of episode)
    $I \leftarrow 1$
    While $S$ is not terminal:
        $A \sim \pi(\cdot \mid S, \boldsymbol{\theta})$
        Take action $A$, observe $S', R$
        $\delta \leftarrow R + \gamma \hat{v}\,(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} I \delta \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w})$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla_{\boldsymbol{\theta}} \ln \pi(A \mid S, \boldsymbol{\theta})$
        $I \leftarrow \gamma I$
        $S \leftarrow S'$
    end While
```

neural networks to estimate value function and the Policy. A softmax layer is used in the Policy network to estimate the action probabilities of a given state.

State Features: The Fourier Basis The Fourier basis provides a simple and principled approach to constructing state features for learning value functions in problems with continuous state spaces. By expanding the representation of states using Fourier Basis functions, the agents can better capture the underlying structure of the environment, improving learning efficiency.

1.)$\phi(s) = [1, \cos(1\pi x), \cos(2\pi x), ..., \cos(M\pi x), \cos(1\pi v), \cos(2\pi v), ..., \cos(M\pi v), \cos(1\pi\omega), \cos(2\pi\omega), ..., \cos(M\pi\omega), \cos(1\pi\dot{\omega}), \cos(2\pi\dot{\omega}), ..., \cos(M\pi\dot{\omega})]^T$.

2)$\phi(s) = [1, \sin(1\pi x), \sin(2\pi x), ..., \sin(M\pi x), \sin(1\pi v), \sin(2\pi v), ..., \sin(M\pi v), \sin(1\pi\omega), \sin(2\pi\omega), ..., \sin(M\pi\omega), \sin(1\pi\dot{\omega}), \sin(2\pi\dot{\omega}), ..., \sin(M\pi\dot{\omega})]^T$.

2.4 Results

2.4.1 687-GridWorld

The performance of One-Step Actor-Critic on 687-GridWorld is illustrated. As we keep training on more episodes the MSE between the estimated and original value function decreases. As the number of episodes increases the agent takes less number of actions to reach the terminal state.

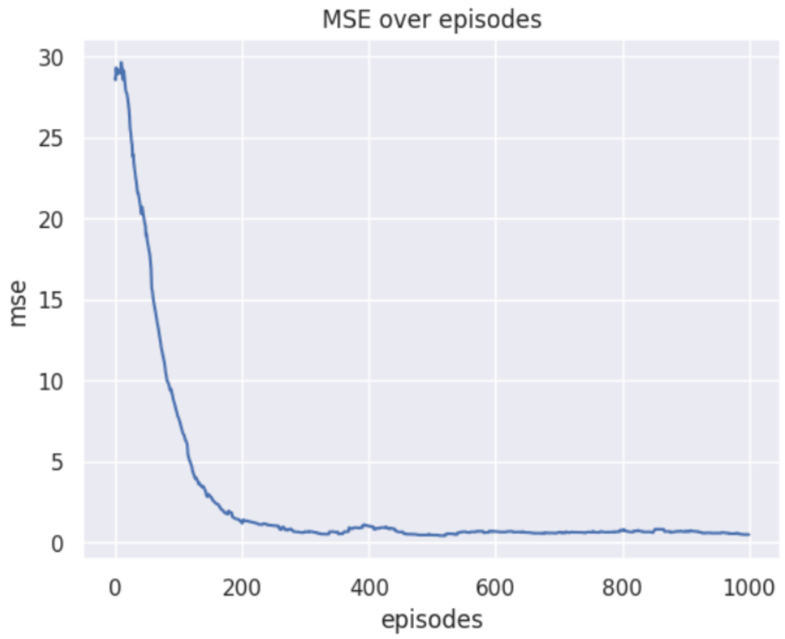


Figure 8: MSE across 500 episodes

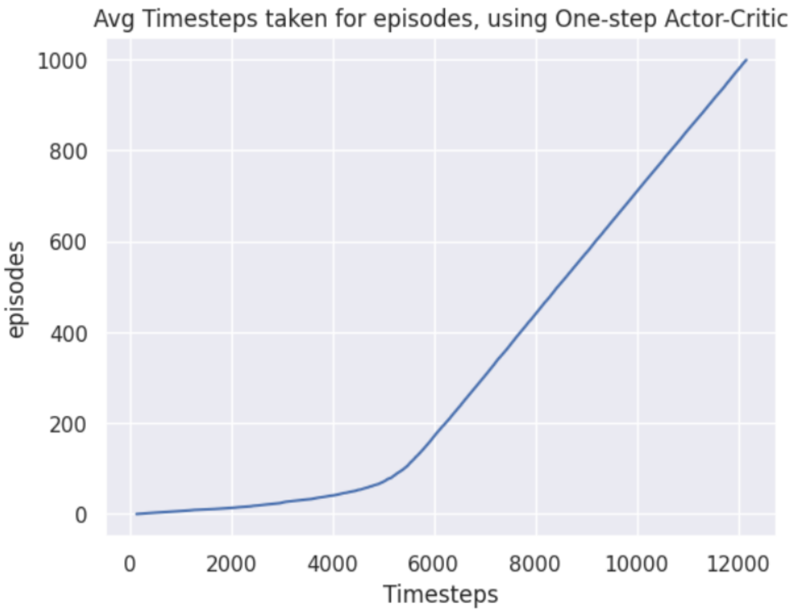
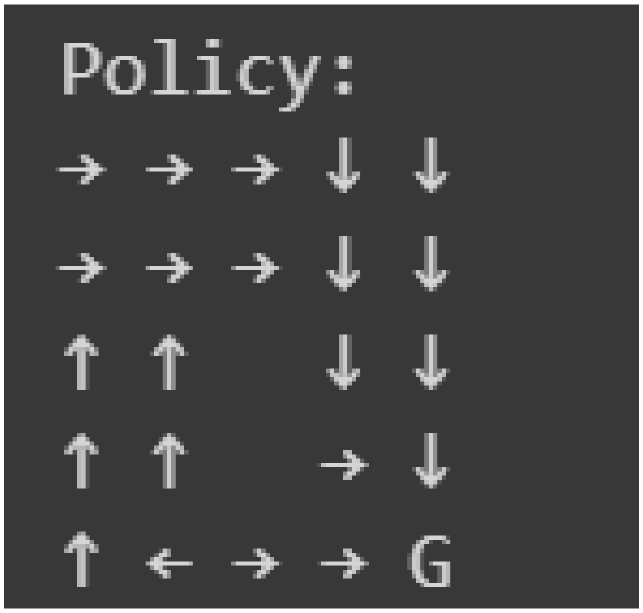


Figure 9: Episodes vs no of actions over time



```
Policy:
→ → → ↓ ↓
→ → → ↓ ↓
↑ ↑   ↓ ↓
↑ ↑   → ↓
↑ ← → → G
```

Figure 10: Policy

Hyperparameter Tuning: The following set of hyperparameters was experimented with.

1. **Learning rate/Step Size** α: The best step size was found to be 0.5, 0.5 for both policy and value function estimates. The step size decayed by half over five hundred episodes. For higher step size like 0.8, 0.9 more fluctuations are observed over the trials and it takes more episodes to converge.

2. **Gamma - γ (Discount parameter)** The discount factor that worked best was found to be 0.9. This value of γ is also observed to work well for most algorithms including SARSA, Q-Learning, Prioritized Sweep, etc. for this domain.

2.4.2 Cart Pole

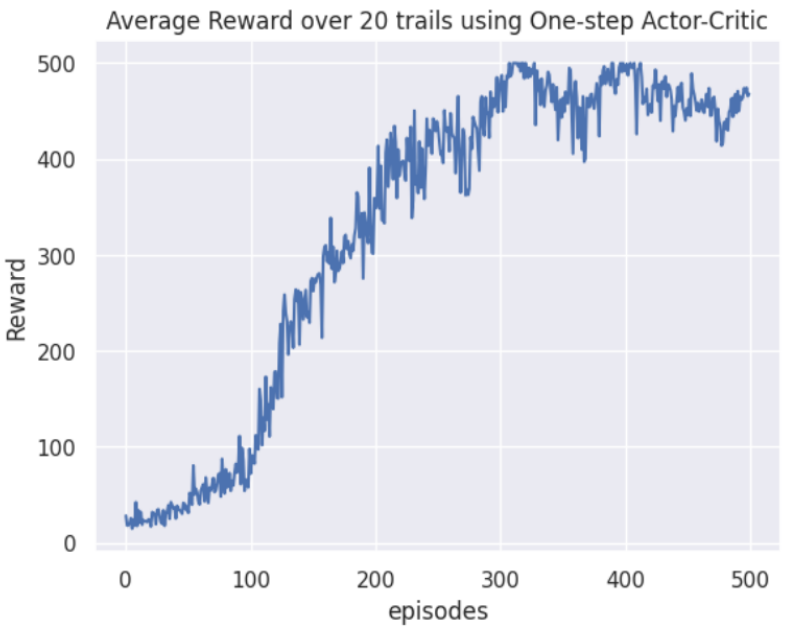


Figure 11: Rewards over time

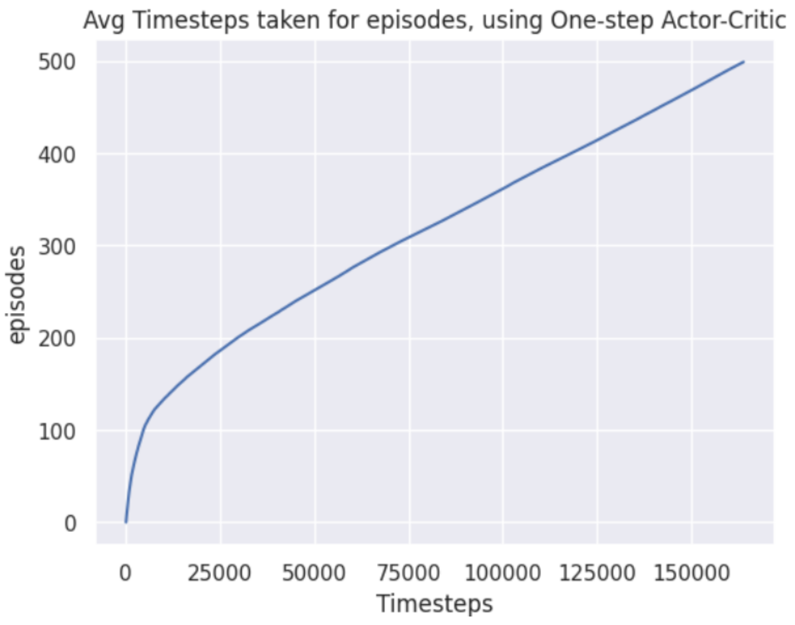


Figure 12: Episodes vs no of actions over time

Figure 11 shows the rewards obtained in the CartPole domain over a period of 500 episodes. As the number of episodes increases, the agent is found to get increasingly better at the task. The fluctuations are partly due to the initial state when doing reset of environment.

Hyperparameter tuning: The following set of hyper parameters were experimented with,

1. **Learning rate/Step Size** α: The best step size was found to be 0.01. Ideally, any schedule of learning rate/step size, where the ϵ decay is faster than the α decay should be preferred, as this means the agent will always learn from observed rewards. If ϵ decays slower than α, this means that at some point, random actions may be given precedence and the agent never actually learns. The ideal step size is observed to be 0.1. Another step size that worked well in practice similar to Q-Learning and SARSA implementations was $\frac{0.25}{sqrt(i)+1}$, where i is the number of episodes completed upto that point.

2. **Fourier basis** M: Tried different values of M, for smaller values of M like 2,3 the model is not able understand the environment and it is not reaching the optimal value. Similarly for high values of M above 8 as the parameters increases, it takes more memory to store and also more time to update the parameter. Out of all the values M= 5 is proved out to be the best. Between sine and cosine, cosine is performing better than sine basis.

3. **Temperature (temp)**: The temperature increases entropy, making the probability distribution more uniform. For policy neural network we added temperature parameter to vary between exploration and exploitation. Initially the temperature is kept high (temp =8) meaning exploration is preferred as the model doesn't know about the environment. Temperature is reduced by a factor of sqrt of two every hundred episodes to exploit more on acquired knowledge.

4. **Gamma - γ (Discount parameter)** The discount factor that worked best was found to be 0.99. For smaller values of γ it is observed to take more time to find optimal policy, higher γ helps to find the optimal policy faster.

The state space is expanded using cosine fourier basis $(x, v, \omega, \dot{\omega})$ is mapped into 4*M+1 space where M = 5 is optimal.

2.4.3 Mountain Car

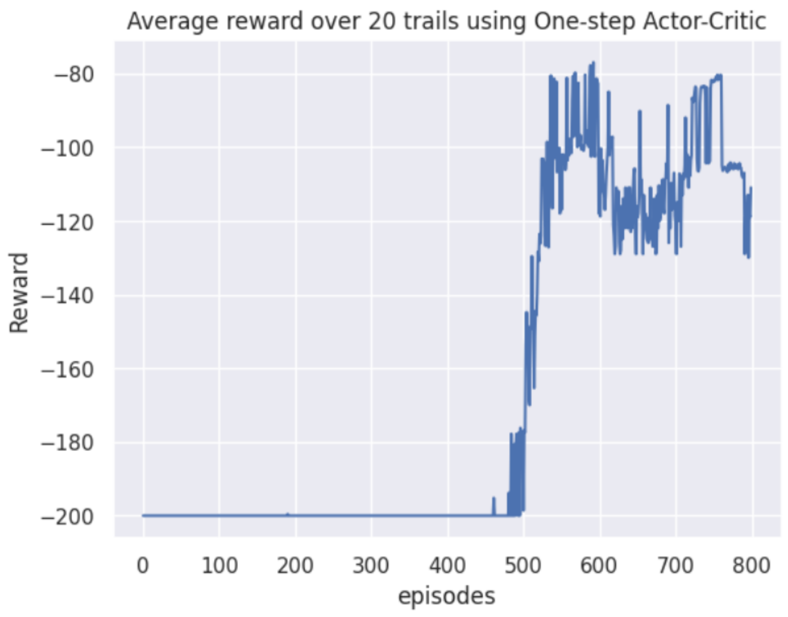


Figure 13: Rewards over time

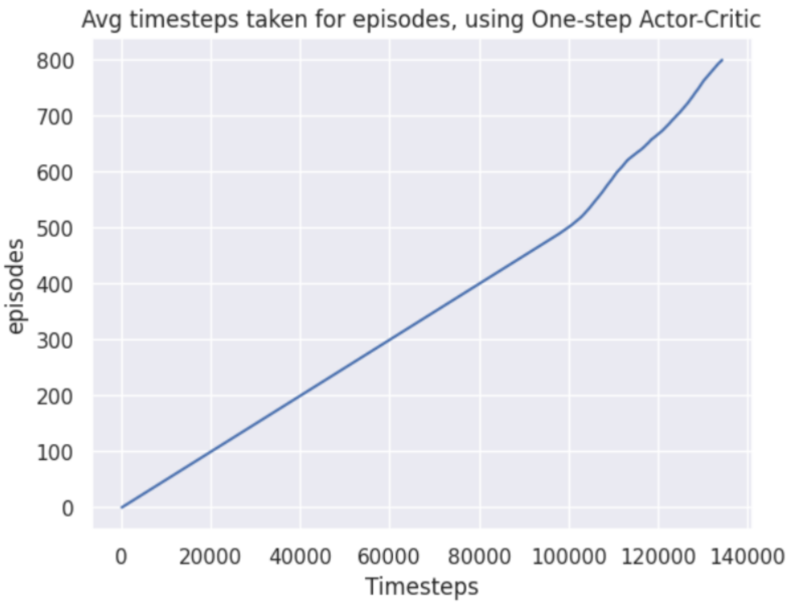


Figure 14: Episodes vs actions over time

It is observed from Figure 13 that the negative reward decreases having trained the agent for 600 episodes. It reaches a reward of -79, starting from -200. The environment gives a negative reward for every state apart from the goal. This penalizes the agent to stay longer in the environment. We can also see from 14 that the number of episodes covered by the agent increases steadily. This is because the agent is penalized to stay long in the environment due to the negative reward, and has to exit quickly, thus it learns to finish an episode quicker by reaching the goal state faster.

Hyperparameter tuning: The following set of hyper parameters were experimented with,

1. **Learning rate/Step Size** α: The best step size was found to be 0.01. Ideally, any schedule of learning rate/step size, where the ϵ decay is faster than the α decay should be preferred, as this means the agent will always learn from observed rewards. If ϵ decays slower than α, this means that at some point, random actions may be given precedence and the agent never actually learns. The ideal step size is observed to be 0.1. Another step size that worked well in practice similar to Q-Learning and SARSA implementations was $\frac{0.25}{sqrt(i)+1}$, where i is the number of episodes completed upto that point.
2. **Fourier basis** : Tried different values of M, for smaller values of M like 2,3 the model is not able understand the environment and it is not reaching the optimal value. Similarly for high values of M above 8 as the parameters increases, it takes more memory to store and also more time to update the parameter. Out of all the values M= 5 is proved out to be the best. Between sine and cosine, cosine is performing better than sine basis.
3. **Temperature (temp)**: The temperature increases entropy, making the probability distribution more uniform. For policy neural network we added temperature parameter to vary between exploration and exploitation. Initially the temperature is kept high (temp =2) meaning exploration is preferred as the model doesn't know about the environment. Temperature is reduced by a factor of sqrt of two every hundred episodes to exploit more on acquired knowledge.
4. **Gamma - γ (Discount parameter)** The discount factor that worked best was found to be 0.99. For smaller values of γ it is observed to take more time to find optimal policy, higher γ helps to find the optimal policy faster.

The state space is expanded using cosine fourier basis $(x, v, \omega, \dot{\omega})$ is mapped into 4*M+1 space where M = 5 is optimal.

The state space is expanded using cosine fourier basis $(x, v, \omega, \dot{\omega})$ is mapped into 2*M+1 space where M = 5 is optimal.

3 Prioritized Sweeping

3.1 Overview

Prioritized belongs to a family of algorithms called Model-based RL, wherein the agent learns a model of the environment in addition to learning the optimal action value functions and policy. Just like Dyna-Q, it involves a two step process, where an agent learns by interactions with the environment, followed by a second step where the agent iteratively learns a model of the environment, and uses this model to learn from *simulated* interactions. However, it builds upon Dyna-Q, by referring to its history context priority-wise. Each update is assigned a priority based on the TD error obtained, and updates are prioritized according to their TD error, with higher TD error updates performed first. Once an update is performed for a given state action pair, this update would have to get reflected in all the parent states as well. As a result, parent state, action pairs are also pushed into the priority queue, for future updations. So it involves both planning and learning by actual interactions. By prioritizing the updates, the agent learns the optimal policy in lesser iterations.

3.2 Algorithm Pseudocode

Algorithm 3: Prioritized Sweeping

```
Initialize $Q(s,a)$ and $Model(s,a)$ randomly $\forall s \in S, \forall a \in A$
PriorityQueue $\leftarrow \{\}$
while True do
    S $\leftarrow$ (current non terminal state)
    done $\leftarrow$ False
    while $done \neq False$ do
        Take action A according to policy (eg: $\epsilon$-greedy on Q) observe next state $s'$ and reward r.
        $Q(s,a) \leftarrow Q(s,a) + lr * (r + \gamma * \max_{a'} Q(s',a') - Q(s,a))$
        $Model(s,a) \leftarrow r, s'$
        $p \leftarrow |Q(s,a) + lr * (r + \gamma * \max_{a'} - Q(s,a))|$
        if $p \geq \theta$ then insert to PriorityQueue with priority p
        Add $(s,a,r,s')$ to history
        if $s'$ is terminal: done $\leftarrow$ True
        for $i = 1$ to $max\_history\_length$ while PriorityQueue is not empty do
            $s \leftarrow$ state
            $a \leftarrow$ action taken in $s$
            $s', r \leftarrow Model(s,a)$
            $Q(s,a) \leftarrow Q(s,a) + lr * (r + \gamma * \max_{a'} - Q(s,a))$
            for All states and actions $\overline{s}$, $\overline{a}$ leading to $s$ do
                $r \leftarrow$Model(s,a)
                $p \leftarrow |Q(\overline{s},\overline{a}) + lr * (r + \gamma * \max_{a'} Q(s,a') - Q(\overline{s},\overline{a}))|$
                if $p \geq \theta$ then insert to PriorityQueue with priority p
                Add $(\overline{s},\overline{a},r,s)$ to history
            end for
        end for
    end while
end while
```

3.3 Explanation

In Prioritized Sweeping, the agent learns in two step process. In the first step, it interacts with the environment, and updates it's q-values, like most RL algorithms. It also *records* the interactions, with rewards, next states, current state, current action, etc. It uses these interactions to build a model of the environment. Using the built model of the environments, it *simulates* interactions, and updates its q-values based on the simulated experiences. Unlike Dyna-Q, where the updates are based on randomly chosen state, action pairs from the history, the updations are prioritized, where the priority is assigned based on the absolute value of $Q(s,a) + lr * (r + \gamma * \max_{a'} - Q(s,a'))$. A threshold θ is often used to decide whether to add a state action pair to the priority queue in the first place. This θ threshold is in practice mostly set to 0, meaning all updates are added.

3.4 Design Decisions

For the 687-Gridworld environment, which consist of discrete state and discrete action spaces, the algorithm lended itself very well and no design tradeoffs had to be incurred in terms of discretization, etc. However, for the CartPole and MountainCar domains, where the state space is continous, Fourier basis was utilized to represent the state features. With the help of tile coding, a unique set of tiles or regions in a hash table is identified for each state in the continuous space, and q-values updates are made by retrieving the uniquely mapped hash locations for each state-action pair. The tiles3 module implemented by Sutton and Barto is used as a tile coding utility to convert continuous state spaces into discrete action space.

3.5 Results

3.5.1 687-GridWorld

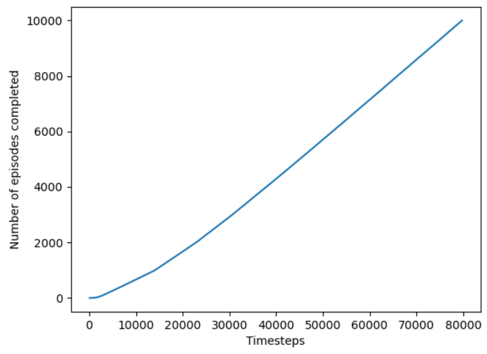


Figure 15: Episodes versus number of actions

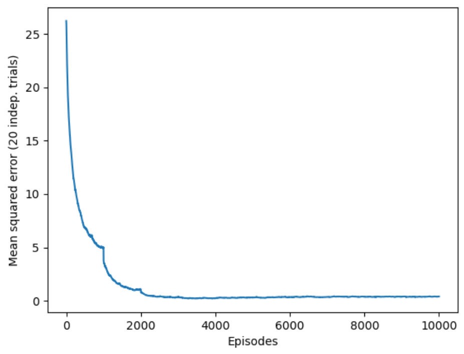


Figure 16: Mean squared error

Figure 16 shows the mean squared error across 10000 episodes averaged across 20 runs. It is observed that the mean squared error rapidly decreases after approximately 2000 episodes. This shows that the agent quickly learns the optimal set of actions for the given states. Figure 15 shows the number of episodes completed against the number of actions taken upto the given point. We observe a rapid increase in number of episodes completed roughly again at 2000 episodes, as the agent picks up the optimal actions and starts to go to the goal state in lesser steps.

Hyperparameter Tuning: The following set of hyperparameters was experimented with.

1. **Learning rate/Step Size** α: The best step size was found to be 0.1. Ideally, any schedule of learning rate/step size, where the ϵ decay is faster than the α decay should be preferred, as this means the agent will always learn from observed rewards. If ϵ decays slower than α, this means that at some point, random actions may be given precedence and the agent never actually learns. The ideal step size is observed to be 0.1. Another step size that worked well in practice similar to Q-Learning and SARSA implementations was $\frac{0.5}{sqrt(i)+1}$, where i is the number of episodes completed upto that point.
2. **Epsilon (Exploration parameter)** ϵ: The ideal schedule of epsilon is found to be a starting value of 0.8, and decaying by a factor of 2, every 1000 episodes. This encourages early exploration, and the exponential decay ensures the agent exploits the learned policy to get maximum rewards. As observed, the agent learns the optimal policy to a large extent fairly quickly.
3. **History context (specific to Dyna Q and Prioritized Sweeping)**: The ideal history context was found to be 100. This means for 100 occurrences of (s, a) (state, action) in the history taken, the learned model is inferred to produce a possible next state, and the update equation $Q(s, a) \leftarrow Q(s, a) + lr * (r + \gamma * \max_{a'} Q(s', a') - Q(s, a))$ is performed. The model in this case is a matrix that counts occurrences of (s, a, s'). The probabilities are obtained by normalizing over the matrix entries, and the next state for a given (s, a) is sampled accordingly. Over time, this matrix builds up a Maximum Likelihood estimate of the transition probabilities as the agent encounters more and more episodes.
4. **Gamma - γ (Discount parameter)** The discount factor that worked best was found to be 0.9. This value of γ is also observed to work well for most algorithms including SARSA, Q-Learning, Prioritized Sweep, etc. for this domain.

The policy learnt is shown in Figure 17

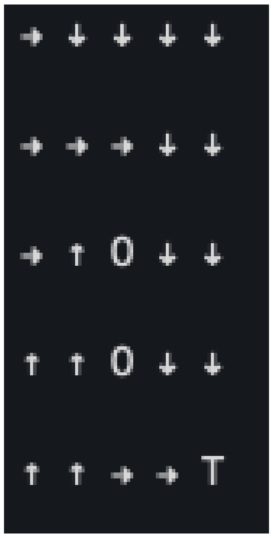


Figure 17: Policy learnt by Prioritized Sweeping

From Figure 3, we observe that Prioritized Sweeping has mostly learnt the optimal policy obtainable through value iteration. It differs from the value iteration obtained policy in two cells (0,1) and (0,2) (DOWN instead of RIGHT) and (2,0) (RIGHT instead of UP). However, the sequence of actions still lead it ultimately to the goal state in the same number of steps from the differing cells. This owes to the fact we are able to learn a strong maximum likelihood model of the transition probabilities by allowing the agent to run many episodes, while at the same time iteratively improving our Prioritized Sweeping model.

3.5.2 CartPole

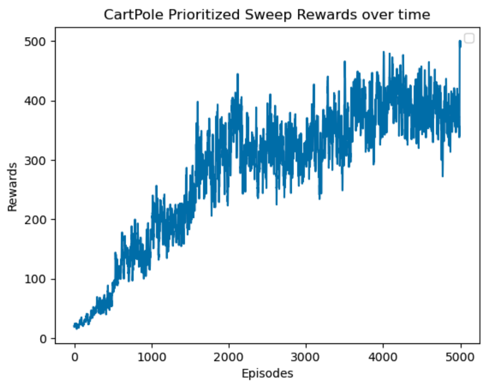


Figure 18: Rewards obtained over time - CartPole

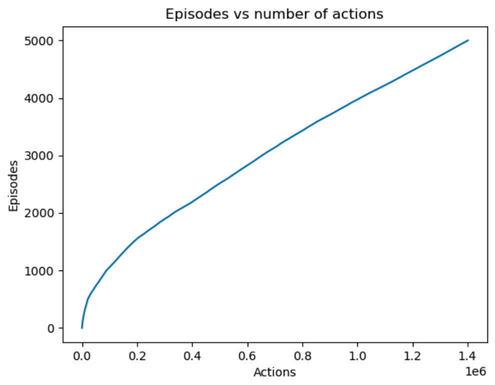


Figure 19: Episodes vs actions in prioritized sweeping

Figure 18 shows the rewards obtained in the CartPole domain over a period of 500 episodes. As the number of episodes increases, the agent is found to get increasingly better at the task. Albeit a bit of variance, the agent still learns the optimal policy and will only get better with more episodes. Figure 2 shows the number of episodes completed against the number of actions. The number of episodes plateaus over time. This indicates that, the agent is learning a better policy, and as a result, balances for a longer time, and takes more actions to complete an episode (which eventually terminates after 500 actions).

Hyperparameter tuning: The following set of hyperparameters were experimented with,

1. **Learning rate/Step Size** α: The best step size was found to be 0.1. Ideally, any schedule of learning rate/step size, where the ϵ decay is faster than the α decay should be preferred, as this means the agent will always learn from observed rewards. If ϵ decays slower than α, this means that at some point, random actions may be given precedence and the agent never actually learns. The ideal step size is observed to be 0.1. Another step size that worked well in practice similar to Q-Learning and SARSA implementations was $\frac{0.5}{sqrt(i)+1}$, where i is the number of episodes completed upto that point.

2. **Epsilon (Exploration parameter)** ϵ: The ideal schedule of epsilon is found to be a starting value of 0.8, and decaying by a factor of 2, every 100 episodes. This encourages early exploration, and the exponential decay ensures the agent exploits the learned policy to get maximum rewards. As observed, the agent learns the optimal policy to a large extent fairly quickly.
3. **History context (specific to both Dyna Q and Prioritized Sweep)**: The ideal history context was found to be 10. This means for 20 occurrences of (s, a) (state, action) in the history taken, the learned model is inferred to produce a possible next state, and the update equation $Q(s, a) \leftarrow Q(s, a) + lr * (r + \gamma * \max_{a'} Q(s', a') - Q(s, a))$ is performed. The model in this case is very deterministic, as a given state and action leads to only possible next state. There are two possible actions, and the best q value is chosen amongst the left and right actions. With the next state being deterministic, it is recorded as part of the history, and this encompasses the model for Dyna-Q.
4. **Gamma - γ (Discount parameter)** The discount factor that worked best was found to be 0.99. This value is found to work well for continous domains.

The state space is discretized using tile coding. As a result, each point in the continuous state space in the form of $(x, v, \omega, \dot{\omega})$ is mapped into a tile coding space with 8 tiles representing this point in the state space.

The Index Hash Table size is kept to be 2^{23}, as an upper bound to avoid any hash collisions. 8 tiles are used to represent $(x, v, \omega, \dot{\omega})$. These 8 tiles represent 8 unique indices in a hash table. The update happens for each of these 8 tiles during updation of q-values. The scaling factor used is 32 for each of the values in the state space. Each point is represented uniquely in a 32*32*32*32 grid, with 8 tiles for each point. This makes the total dimensionality of 32*32*32*32*8, i.e. 2^{23} as mentioned.

4 Tile Coding

Tile Coding is a discretization scheme, where in continuous input space can be represented in discretized form. An M*M-dimensional state space for eg. can be mapped to an N*N-dimensional tile space, where N can be much lesser than M.

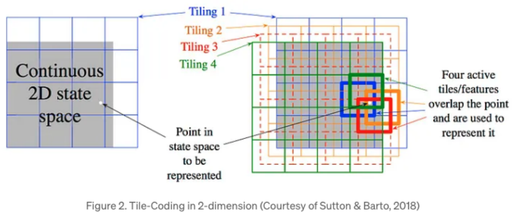

Tiling 1
Tiling 2
Tiling 3
Tiling 4
Continuous 2D state space
Point in state space to be represented
Four active tiles/features overlap the point and are used to represent it
Figure 2. Tile-Coding in 2-dimension (Courtesy of Sutton & Barto, 2018)

Figure 20: Tile Coding, illustration - from Sutton and Barto

Some advantages it gives are:

1. **Discretization:** Some domains, like mountain car, cartpole have continous state space, however the state spaces can be effectively binned, as the optimal agent action to take in each bin is fairly the same. Given the action space is quite small, tile coding works very effectively, both in terms of space and time complexity, as well as the time taken to learn the optimal policy.
2. **Overlapping in Tiles:** The basic premise in tile coding, is that overlapping features have overlapping tiles. This means that features with slight overlap will have fewer number of tiles in common as opposed to features with more overlap. The non-zero overlap is a very useful feature, which helps in the agent generalizing to unseen values in the continuous state space.

3. **Hash Table and Space Efficiency:** By converting a sparse N-Dimensional matrix into hash table blocks which are one-dimensional, we can more effectively manage the memory and avoid excessive memory wastage, something very common in sparse matrices. The Index Hash table, is responsible for mapping the discretized state space values into unique locations in the hash table. By making the number of the blocks in the table to a sizeable value, we can minimize the probability of hash collisions to a large extent.

By using Tile Coding, we can extend continuous domains to algorithms that only traditionally work in a tabular setting, i.e. Dyna-Q, Prioritized Sweeping, etc. By discretization and provided the discretized state is exhaustive enough, and at the same time, the space of policies is not very large, tile coding is very effective is continuous domains, as illustrated by the performance of Dyna Q and Prioritized Sweeping.

References

A Appendix

You may include other additional sections here.