Artificial intelligence

HIMA VAMSI (22B1001)

July 31, 2023

Contents

1 Introduction of AI

1.1 Defnition of AI

Saying that AI is an artificial intelligence doesn't really tell you anything meaningful,we can define it like, what occurs is artificial and intelligence should involve some elements like

- Learning
- Reasoning
- Understanding
- Manipulating

(this list contains only some elements which comes under intelligence)
Artificial Intelligence refers to the development of computer systems that can perform tasks that would typically require human intelligence. These tasks include perception, reasoning, problem-solving, and decision-making. AI systems often employ algorithms and models to learn from data, adapt to changing environments, and improve their performance over time.

In this report, we explore the field of Reinforcement Learning (RL) and its significance in the broader context of Artificial Intelligence (AI). This introductory section provides an overview of AI and its definition, highlighting its role in solving complex problems. Additionally, we outline the various branches of AI, including supervised and unsupervised learning, which serve as building blocks for understanding RL.
For further clarity refer the book [3]

1.2 Branches of AI

AI encompasses several branches, each focusing on specific approaches and techniques for solving problems. There are some main branches of AI which i will go through:

1.2.1 Machine Learning

I will go through the machine learning briefly in the further sections of this report.Here i will give some basic defnitions.
Machine Learning is a branch of AI that involves developing algorithms capable of learning and making predictions or decisions without being explicitly programmed. It focuses on the development of statistical models that automatically improve their performance through experience.

1.2.2 Deep Learning

Deep Learning is a subfield of machine learning that utilizes deep neural networks with multiple layers to learn hierarchical representations of data. It has achieved significant breakthroughs in various domains, including image recognition, natural language processing, and speech recognition.
Deep learning is a very vast concept.There are many things to learn in deep learning. One of the main concept is Neural networks.There are several types of neural networks like

- Deep Neural networks
- Training neural networks
- Convolutional neural networks
- Recurrent neural networks

Here I am not going through these topics because my main topic was about machine learning.
Some applicationns of deep learning are:

- **Image Recognition**: Deep learning models have achieved remarkable performance in image recognition tasks, surpassing human-level accuracy in some cases. They are used in applications like autonomous vehicles, medical imaging, and facial recognition systems.
- **Natural Language Processing**: Deep learning models have revolutionized natural language processing tasks such as language translation, sentiment analysis, chatbots, and question-answering systems. They can learn semantic representations of words and sentences, enabling better understanding and generation of human language.
- **Speech Recognition**: Deep learning has significantly improved speech recognition systems, making voice assistants like Siri and Alexa more accurate and reliable. Deep neural networks, particularly recurrent neural networks and attention-based models, have been instrumental in this domain.
- **Generative Models**: Deep learning has led to the development of generative models capable of generating realistic images, videos, and music. Examples include Generative Adversarial Networks (GANs) and Variational Autoencoders (VAEs), which have applications in art, entertainment, and data augmentation.

1.2.3 Natural Language Processing (NLP)

Natural Language Processing involves the interaction between computers and human language. It encompasses tasks such as text understanding, sentiment analysis, language translation, and question-answering systems. I will go through some main topics of NLP below:

- **Language Understanding**: NLP aims to enable computers to understand and interpret human language. This involves tasks such as part-of-speech tagging, syntactic parsing, named entity recognition, and semantic role labeling. These techniques help analyze the grammatical structure and extract meaning from sentences.
- **Language Generation**: NLP also focuses on generating human-like language. This includes tasks such as text summarization, machine translation, and dialogue systems. Language generation techniques involve generating coherent and contextually appropriate text based on given input or context.
- **Sentiment Analysis**: Sentiment analysis is the process of determining the sentiment or opinion expressed in a piece of text. NLP algorithms can classify text as positive, negative, or neutral, enabling sentiment analysis in customer reviews, social media posts, and other textual data sources.
- **Speech Recognition**: NLP is also closely tied to speech recognition, which involves converting spoken language into written text. NLP algorithms process the audio input, recognize individual words or phonemes, and produce transcriptions of the spoken content.
- **Information Extraction**: Information extraction techniques involve extracting structured information from unstructured text. This includes tasks such as named entity recognition, relation extraction, and event detection. NLP algorithms identify and extract specific pieces of information, such as person names, organization names, or temporal expressions.

1.2.4 Computer Vision

Computer Vision deals with enabling computers to gain a high-level understanding of visual content. It includes tasks such as image recognition, object detection, image segmentation, and scene understanding. Computer vision has many small subtopics here i will just go through all of their defnitions:

- **Image Classification**: Computer Vision algorithms can classify images into different categories or classes. This involves training models to recognize and differentiate between objects, scenes, or patterns within images.

- **Object Detection**: Object detection aims to locate and identify specific objects within an image or video. It involves algorithms that can not only recognize objects but also draw bounding boxes around them.
- **Image Segmentation**: Image segmentation divides an image into different regions or segments, based on similarities in color, texture, or other visual properties. This technique enables the identification and separation of objects or regions of interest within an image.
- **Scene Understanding**: Scene understanding involves analyzing the overall context and content of a scene. It includes tasks such as scene classification, scene parsing, and understanding the relationships between objects in a scene.
- **Pose Estimation**: Pose estimation aims to estimate the position and orientation of objects or human bodies within an image or video. This technique finds applications in robotics, augmented reality, and human-computer interaction.
- **Face Recognition**: Face recognition algorithms identify and verify individuals by analyzing facial features. This technology is commonly used for authentication, surveillance, and personalization in various applications.
- **Video Analysis**: Computer Vision techniques extend to the analysis of videos. This includes tasks such as action recognition, activity detection, and tracking objects or people across multiple frames.
- **3D Reconstruction**: Computer Vision algorithms can reconstruct three-dimensional models of objects or scenes from two-dimensional images or videos. This technique finds applications in virtual reality, 3D modeling, and autonomous navigation.

1.2.5 Robotics

Robotics combines elements of AI, machine learning, and control systems to design and develop intelligent robots capable of perceiving and interacting with the physical world. It involves areas such as robot perception, motion planning, and robot learning.
Now I will go through some main topics of Robotics so that the reader can get an idea of those subtopics:

- **Industrial Robotics**: Industrial robots are widely used in manufacturing and production processes. They are designed to automate repetitive tasks, such as assembly, welding, packaging, and material handling, improving efficiency and productivity in industries.

- **Medical Robotics**: Medical robots are used in healthcare settings to assist with surgeries, rehabilitation, diagnostics, and patient care. These robots can perform precise and delicate procedures, provide support to medical professionals, and enhance patient outcomes.
- **Service Robotics**: Service robots are designed to assist humans in various settings, such as homes, hospitals, hotels, and public spaces. They can perform tasks like cleaning, delivery, companionship, and assistance for the elderly or individuals with disabilities.
- **Autonomous Vehicles**: Autonomous vehicles, including self-driving cars, drones, and unmanned aerial vehicles (UAVs), are a significant application of robotics. These vehicles use sensors, computer vision, and advanced control algorithms to navigate and operate without human intervention.
- **Exploration and Space Robotics**: Robotics plays a crucial role in space exploration and planetary missions. Robots are used for tasks such as data collection, sample analysis, repairs, and exploration of challenging environments where human presence is difficult or dangerous.
- **Humanoid Robotics**: Humanoid robots are designed to resemble and interact with humans. They have human-like features, including arms, legs, and facial expressions. Humanoid robots find applications in research, entertainment, education, and assistive technologies.
- **Agricultural Robotics**: Agricultural robots are used in farming and agriculture to automate tasks like planting, harvesting, spraying, and monitoring crops. These robots improve efficiency, reduce labor costs, and enable precision agriculture techniques.

2 Machine Learning

Machine Learning is a branch of AI that focuses on the development of algorithms capable of automatically learning and improving from data. It can be broadly divided into three main categories: supervised learning, unsupervised learning, and reinforcement learning.

2.1 Supervised Learning

Supervised learning involves training a model on labeled examples, where the input data is accompanied by corresponding desired outputs. The model learns from these examples to generalize and make predictions or classifications on unseen data. Classification and regression problems are commonly tackled using supervised learning techniques. Here I will write how the supervised learning works:

2.1.1 How it works

I have taken most of this information from the book [2] In supervised learning1, the dataset is the collection of labeled examples $(x_i, y_i)_{i=1}^{N}$

Each element xi among N is called a feature vector. A feature vector is a vector in which each dimension j = 1,...,D contains a value that describes the example somehow. That value is called a feature and is denoted as $x^{(j)}$. For instance, if each example x in our collection represents a person, then the first feature, $x^{(1)}$, could contain height in cm, the second feature, $x^{(2)}$, could contain weight in kg, $x^{(3)}$ could contain gender, and so on. For all examples in the dataset, the feature at position j in the feature vector always contains the same kind of information. It means that if $x_i^{(2)}$contains weight in kg in some example x_i, then $x_k^{(2)}$ will also contain weight in kg in every example x_k, k = 1,...,N. The label y_i can be either an element belonging to a finite set of classes $1, 2, ..., C$, or a real number, or a more complex structure, like a vector, a matrix, a tree, or a graph. For instance, if your examples are email messages and your problem is spam detection, then you have two classes "spam, notspam".

Example for understanding:

Let's say the problem that you want to solve using supervised learning is spam detection. You gather the data, for example, 10,000 email messages, each with a label either "spam" or "notspam" (you could add those labels manually or pay someone to do that for us). Now, you have to convert each email message into a feature vector. The data analyst decides, based on their experience, how to convert a real-world entity, such as an email message, into a feature vector. One common way to convert a text into a feature vector, called bag of words, is to take a dictionary of English words (let's say it contains 20,000 alphabetically sorted words) and stipulate that in our feature vector:

- the first feature is equal to 1 if the email message contains the word "a"; otherwise, this feature is 0;
- the second feature is equal to 1 if the email message contains the word "aaron"; otherwise, this feature equals 0;
- ...
- the feature at position 20,000 is equal to 1 if the email message contains the word "zulu"; otherwise, this feature is equal to 0.

You repeat the above procedure for every email message in our collection, which gives us 10,000 feature vectors (each vector having the dimensionality of 20,000) and a label ("spam"/"notspam"). Now you have a machine-readable input data, but the output labels are still in the form of human-readable text. Some learning algorithms require transforming labels into numbers. For example, some algorithms

require numbers like 0 (to represent the label "notspam") and 1 (to represent the label "spam"). The algorithm I use to illustrate supervised learning is called Support Vector Machine (SVM). This algorithm requires that the positive label (in our case it's "spam") has the numeric value of +1 (one), and the negative label ("notspam") has the value of -1 (minus one)

The algorithm puts all feature vectors on an imaginary 20,000- dimensional plot and draws an imaginary 20,000-dimensional line (a hyperplane) that separates examples with positive labels from examples with negative labels. In machine learning, the boundary separating the examples of different classes is called the Decision boundary.

The equation of the hyperplane is given by two parameters, a real-valued vector w of the same dimensionality as our input feature vector x, and a real number b like this:

$$wx - b = 0$$

where the expression wx means $w^{(1)}x^{(1)} + w^{(2)}x^{(2)} + ... + w^{(D)}x^{(D)}$, and D is the number of dimensions of the feature vector x
Now the final equation
Now, the predicted label for some input feature vector x is given like this:

$$y = sign(wx - b)$$

The goal of the learning algorithm — SVM in this case is to leverage the dataset and find the optimal values w* and b* for parameters w and b. Once the learning algorithm identifies these optimal values, the model f(x) is then defined as:

$$f(x) = sign(w^*x - b^*)$$

Therefore, to predict whether an email message is spam or notspam using an SVM model, you have to take a text of the message, convert it into a feature vector, then multiply this vector by w*, subtract b* and take the sign of the result. This will give us the prediction (+1 means "spam", -1 means "notspam").

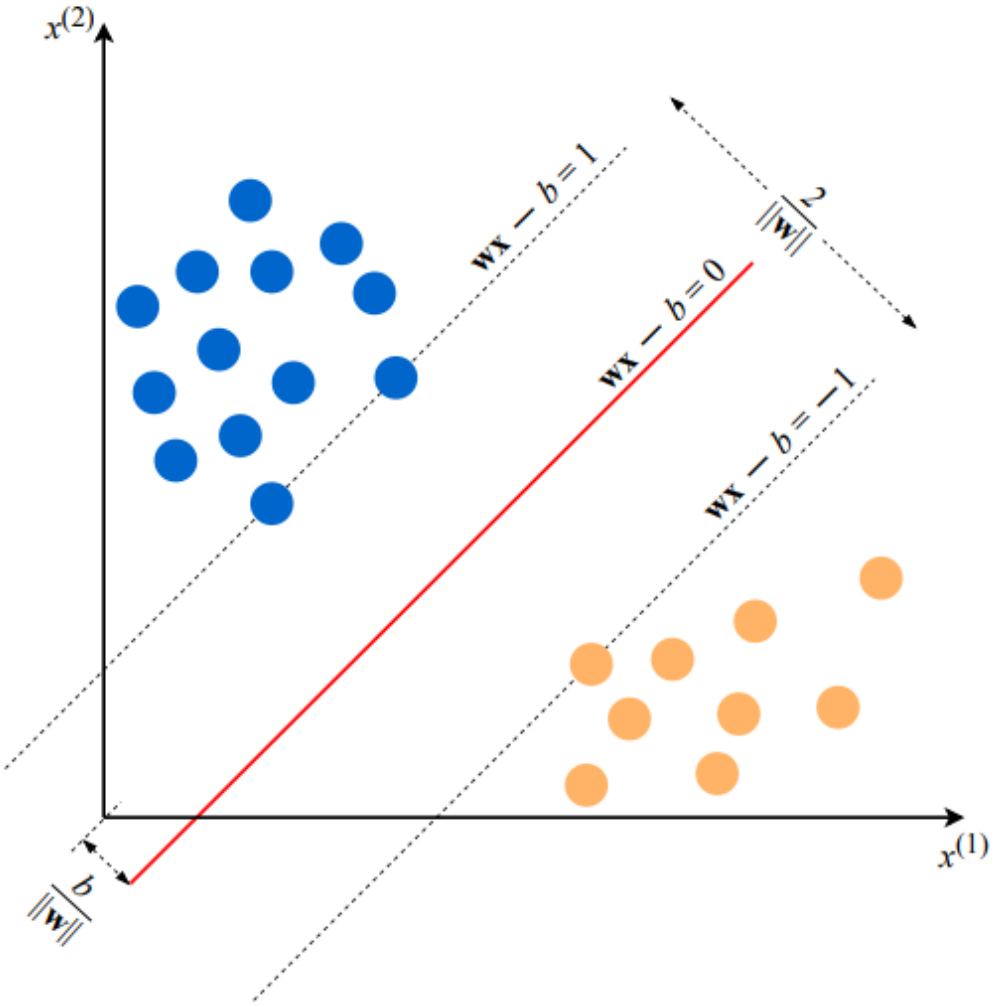


Figure 1: Example of an SVM model for 2-dimension

for largest margin we have to minimize the norm of the vector w (this is done by machine)

2.2 Unsupervised Learning

Unsupervised learning aims to discover patterns and structures within unlabeled data. Algorithms in this branch analyze the inherent structure of the data, such as clustering similar data points or finding latent representations. Unsupervised learning is useful when labeled data is scarce or unavailable.
In unsupervised learning, the dataset is a collection of unlabeled examples ${x_i}^N{}_{i=1}$

Again, x is a feature vector, and the goal of an unsupervised learning algorithm is to create a model that takes a feature vector x as input and either transforms it into another vector or into a value that can be used to solve a practical problem.

For example, in clustering, the model returns the id of the cluster for each feature vector in the dataset. In dimensionality reduction, the output of the model is a feature vector that has fewer features than the input x; in outlier detection, the output is a real number that indicates how x is different from a "typical" example in the dataset.

2.3 Reinforcement Learning

Reinforcement Learning is a subfield of machine learning that focuses on learning optimal behavior through interactions with an environment. RL agents learn to take actions to maximize cumulative rewards based on the feedback received from the environment.

2.4 Key Components of RL

There are some keywords that we should know before going to learn it clearly.
Reinforcement Learning involves several key components:

2.4.1 Agent

The agent is the entity responsible for making decisions and taking actions in the environment. It can range from simple algorithms to complex systems, depending on the problem domain.

2.4.2 Environment

The environment represents the external context in which the agent operates. It defines the state of the world, the set of possible actions, and the rules governing how the agent interacts with the environment.

2.4.3 State

The state refers to the current configuration or representation of the environment. It encapsulates all relevant information necessary for the agent to make decisions.

2.4.4 Action

Actions are the choices available to the agent in a given state. The agent selects an action based on its current policy, which can be deterministic or stochastic.

2.4.5 Reward

The reward is a scalar feedback signal that indicates the desirability of an agent's action in a particular state. The agent's goal is to learn a policy that maximizes cumulative rewards over time.

2.5 Main Topics in Reinforcement Learning

These are some main topics of reinforcement learning which we go through further briefly. Reinforcement Learning encompasses several main topics, each addressing different aspects of the learning process:

2.5.1 Value Functions

Value functions estimate the expected cumulative rewards an agent can obtain from a particular state or state-action pair. They provide a measure of the long-term desirability of being in a specific state or taking a particular action.

2.5.2 Policy Optimization

Policy optimization techniques aim to find the optimal policy that maximizes the expected cumulative rewards. These methods explore different exploration-exploitation strategies to strike a balance between exploring new actions and exploiting known actions.

2.5.3 Temporal Difference Learning

Temporal Difference (TD) learning is a class of RL algorithms that update value function estimates based on the difference between estimated and observed rewards. TD methods bridge the gap between Monte Carlo methods, which require complete episodes, and dynamic programming methods, which need a model of the environment.

2.5.4 Deep Reinforcement Learning

Deep Reinforcement Learning combines RL with deep neural networks to handle high-dimensional state spaces. Deep RL algorithms, such as Deep Q-Networks (DQN) and Proximal Policy Optimization (PPO), have achieved remarkable success in complex domains, including game playing and robotics.

3 Fundamentals of Reinforcement Learning

Most parts of this section were taken from the videos in the link [1]

3.1 k-armed Bandit Problem

This k-armed bandit problem consists of some elements like decision maker(agent) , who chooses between k different actions, and he receives a reward according to on the action he choosed.

- Reward is the result of the selected action. Each action may yield rewards following different probability distributions.

$$q_*(a) \doteq \mathbb{E}[R_t|A_t = a]$$

- q_* is the mean of the distributions for each action.The goal is to maximize the q_* function and we will choose the action according to that.
- The action values, denoted as $q_*(a)$, are estimates of the true action values. Initially, these estimates are often initialized to some initial value, such as 0 or a small random value. As the agent interacts with the bandit problem, it receives rewards for choosing different actions, allowing it to update its estimates of the action values.
- these Action values in k-armed bandit problems represent the estimated or true expected reward associated with each available action. These values are used by the agent are to make decisions and learn which actions are more rewarding.

This kind of nature of a bandit problem shows about the fact that actions and rewards occur over a sequence of time steps. The agent's decisions in earlier steps can affect future choices and outcomes. The agent aims to learn and improve its action selection strategy based on received rewards, striking a balance between exploring new options and exploiting the best-known actions.

3.2 Sample-Average Method

The agent will run many trials to learn about each action.
q_* is not known to the agent instead, we will need to find a way to estimate it. One way to estimate is to compute a sample-average. We simply record the total reward for each action and divide it by the number of times that action has been selected.

$$Q_t(a) \doteq \frac{\textbf{Sum of rewards when } a \textbf{ taken prior to } t}{\textbf{number of times } a \textbf{ taken prior to } t}$$

The greedy action is the action that currently has the largest estimated value. Selecting the greedy action means the agent is exploiting its current knowledge.
The agent would sacrifice immediate reward hoping to gain more information about the other actions.

3.2.1 Incremental Update Rule

The sample average method can be written in a recursive manner. By doing so, we can avoid storing all the previous data.

$$NewEstimate = OldEstimate + StepSize[Target - OldEstimate]$$

The error in the estimate is the difference between the old estimate and the new target.

$$Q_{n+1} = Q_n + \alpha_n(R_n - Q_n)$$

The step size can be a function of n that produces a number from zero to one.In the specific case of the sample average, the step size is equal to $\frac{1}{n}$.

3.2.2 Non-Stationary bandit Problems

- In these type of Problems the action reward distribution changes as the time changes. The agent is unaware of this change but would like to adapt to it. One option is to use a fixed step size. If α_n is constant like 0.1, then the most recent rewards affect the estimate more than older rewards.
- Then by expanding the incremental Update rule we get an Binomial distribution of the rewards in which the recent rewards contribute more.
- A constant step size parameter is used to solve a non-stationary bandit problem.

3.3 Exploration vs. Exploitation

- Exploration allows the agent to improve his knowledge about each action, leading to long-term benefit. By improving the accuracy of the estimated action values, the agent can make more informed decisions in the future.
- Exploitation exploits the agent's current estimated values. It chooses the greedy action to try to get the most reward. But by being greedy with respect to estimated values, may not actually get the most reward.

When we explore, we get more accurate estimates of our values. When we exploit, we might get more reward. We cannot however choose to do both simultaneously. How do we choose when to explore, and when to exploit?

3.3.1 Epsilon-Greedy Method

One very simple method for choosing between exploration and exploitation is to choose randomly. The probability of exploring is small value assumed to be epsilon.

$$A_t = \begin{cases} \text{argmax } Q_t(a) & \text{With Probability } 1-\epsilon \\ \text{Random } a \text{ Uniform}(a_1 ... a_k), & \text{With Probability } \epsilon \end{cases} \tag{1}$$

In the epsilon-greedy method of the agent selects actions based on a balance between exploration and exploitation. The agent chooses the action with the highest estimated value (exploitation) most of the time but also explores other actions with a certain probability (exploration).

To estimate the action values, the agent uses sample average method. Averaging over 2000 samples of different random seeds in the epsilon-greedy method helps to reduce variance and obtain more reliable estimates of the action values, enabling the agent to make better decisions in exploring and exploiting actions. As the

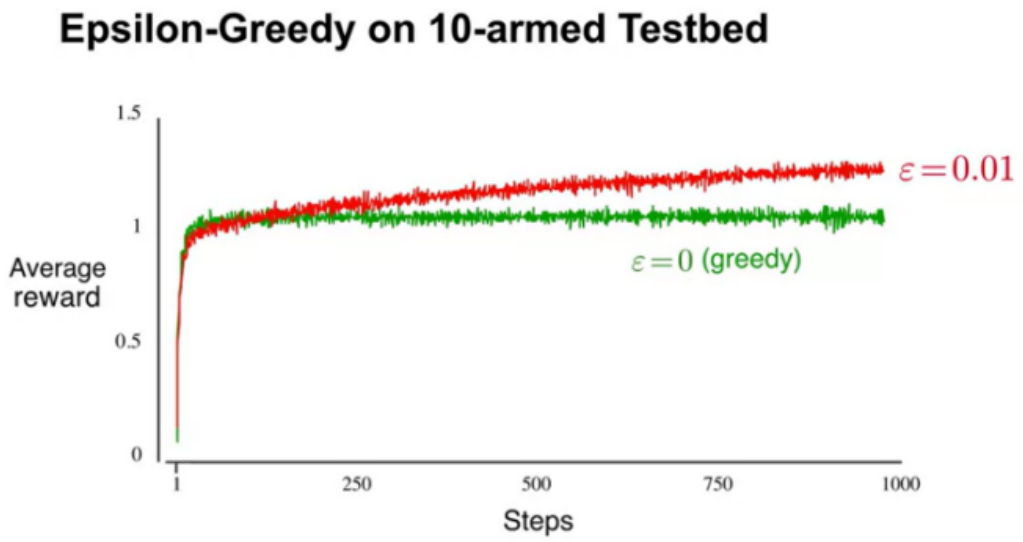


Figure 2: Epsilon-Greedy

probability to explore is increased in fig 2 the Average reward increases as the time goes on.

3.3.2 Optimistic Initial Value

In this Method agent optimistically assumes that each action is highly effective before running the trial. Agent assumes the initial value for every action is equal and greater than the expected reward. Let's assume the agent always chooses the greedy action.

Whenever the agent selects an action, the first time that action is selected the observed reward will likely be smaller than the optimistic initial estimate. The estimated value for this action will decrease, and other actions will begin to look more appealing in comparison. Optimistic initial values encourages exploration early in learning.

In early learning, the optimistic agent performs worse because it explores more. Its exploration decreases with time, because the optimism and its estimates washes out with more samples refer 3.

3.3.3 Limitations of Optimistic Initials

- optimistic initial values only drive exploration early in learning, this means agents will not continue exploring after some time.

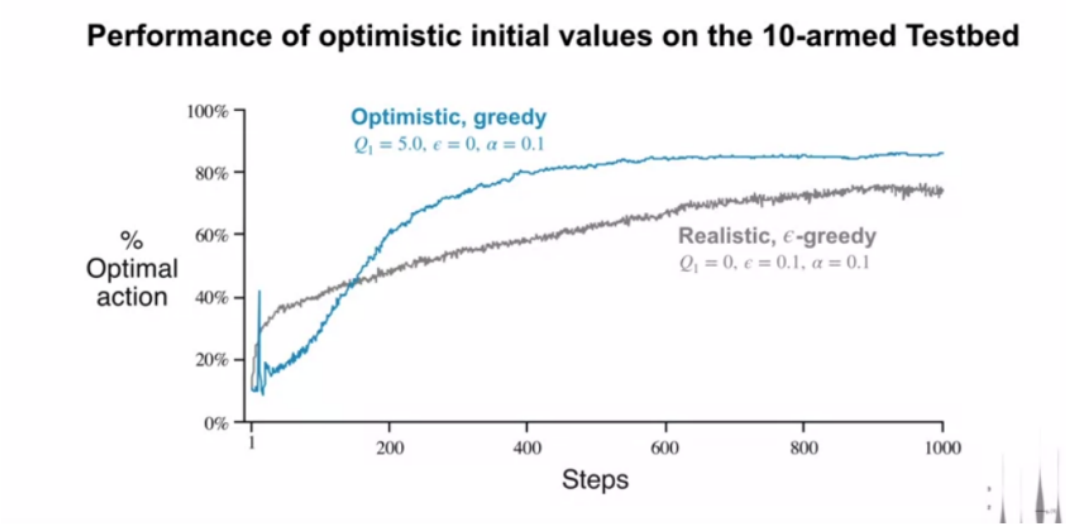


Figure 3: comparision between optimistic initial value and epsilon-greedy

- This leads to issues in non-stationary problems. For example, one of the action values may change after some number of time steps. An optimistic agent may have already settled on a particular action, and will not notice that a different action is better now.
- Determining the precise value for optimistic initialization can be challenging since it requires prior knowledge of the problem's dynamics and reward structure, which may not always be available.

3.3.4 Upper-Confidence Bound (UCB) Action Selection

In Epsilon-Greedy method exploring probability is uniform throughout the time. In this method that probability will vary according to time.

In UCB, we follow the principle of optimism in the face of uncertainty. So it optimistically picks the action that has the highest upper bound. We can use upper-confidence bounds to select actions using the following formula;

$$A_t \doteq \text{argmax}[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}}]$$

The upper-bound term can be broken into three parts. The C parameter as a user-specified parameter that controls the amount of exploration. The first term in the sum represents the exploitation part, and the second term represents the exploration part.

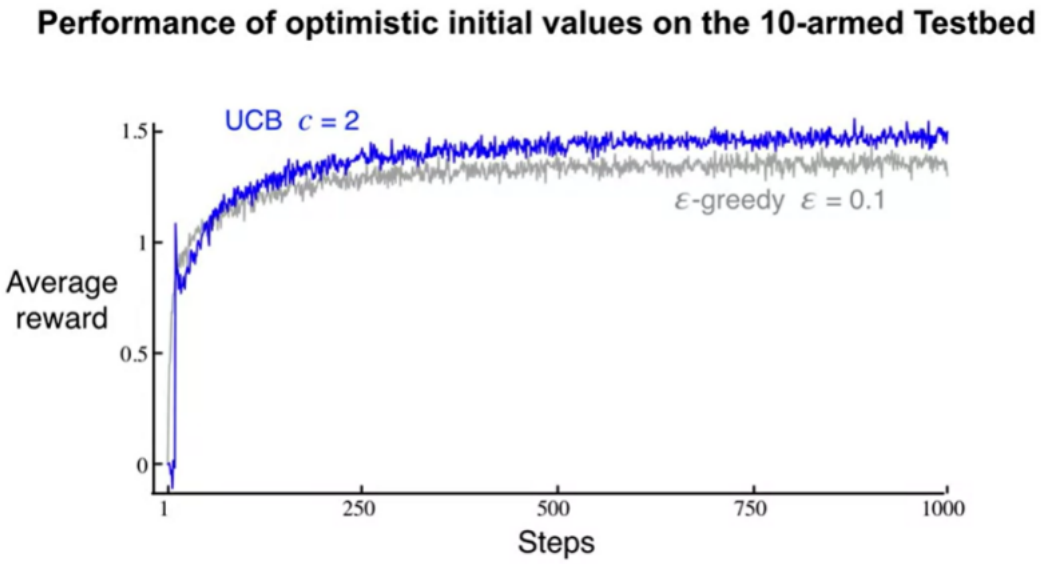


Figure 4: comparision between UCB and epsilon-greedy

Initially, UCB explores more to systematically reduce uncertainty. UCB's exploration reduces over time whereas Epsilon-greedy continues to take a random action with 10 percent of the time.

4 Conclusion

In this report, we provided an introduction to AI, exploring its definition and various branches such as supervised and unsupervised learning. We then jumped into the field of Reinforcement Learning, discussing its key components and main topics. I started reinforcement learning and done some fundementals. There are also many other things in reinforcement learning which i will cover in my next part of work. And most of my doubts are clarified from the book [4]

5 Markov Decision Process

In the K-Armed bandit problems doesn't include many aspects of real-world problems , like it does not take care about the upcoming situation of the agent after choosing a action. It just focusses on the present action and chooses in such a way to get highest possible reward.
But in markov decision process(MDPs) the agent's action also depends on the possible future states.
Let us take an example of a rabbit having brocolli and carrot in each of its two sides. Let us choose that eating carrot gives reward of 10 and eating brocoll gives reward of 3. But near the rabbit if there was a tiger then choosing the carrot results

in the bigger negative reward in the next action. We can formalize this interaction

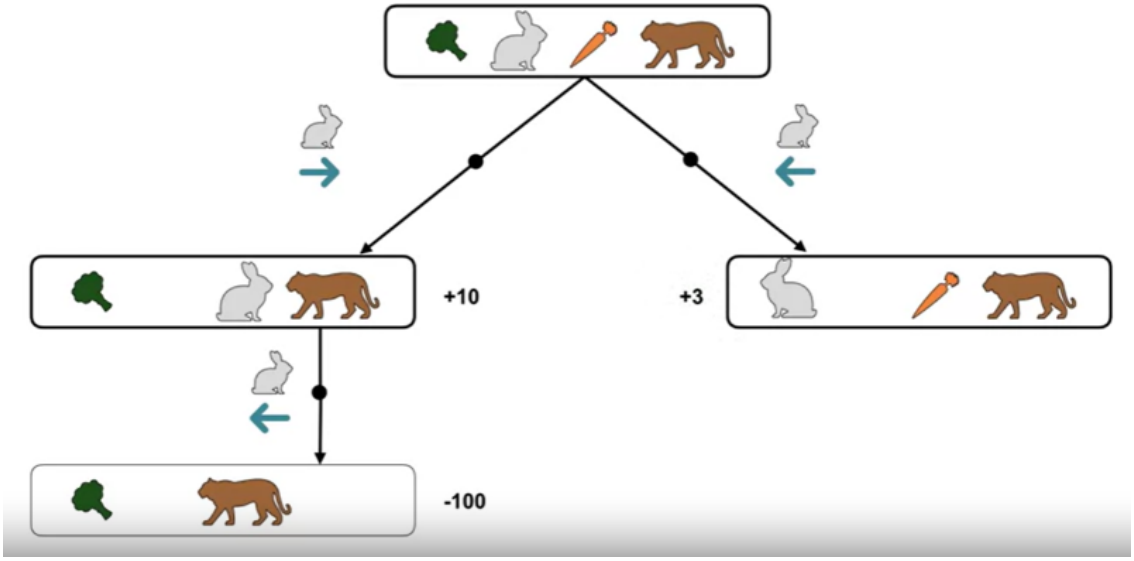


Figure 5: tiger rabbit example

with the general framework. In this framework, the agent and environment interact at discrete time steps. At each time, the agent receives a state S_t from the environment from a set of possible states, script S. The configuration shown on the slide is an example of a state. Based on this state the agent selects an action A_t from a set of possible actions. Script A of S_t is the set of valid actions in state S_t. Moving right is an example of an action. One time step later based in part on the agent's action, the agent finds itself in a new state S_{t+1} . . This diagram summarizes the agent environment interaction in the MDP framework. The agent environment interaction generates a trajectory of experience consisting of states, actions, and rewards. Actions influence immediate rewards as well as future states and through those, future rewards.

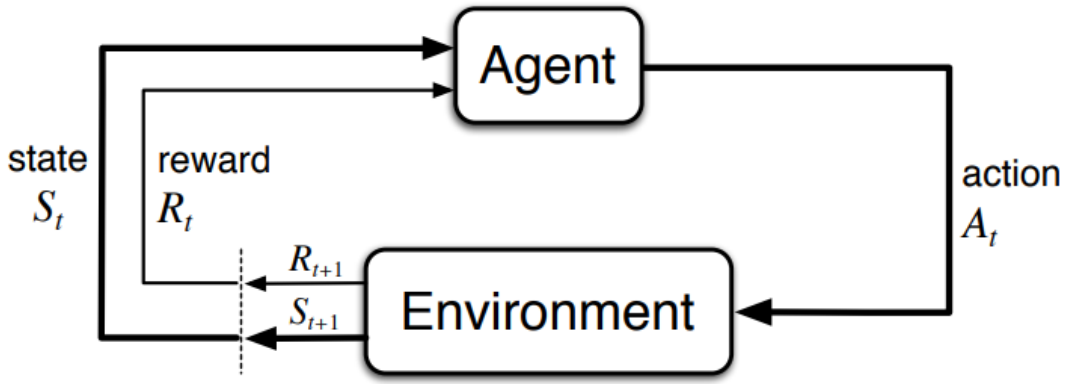
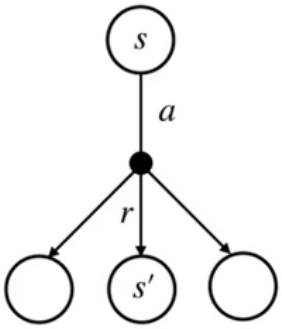


Figure 6: overview of Markov decision process

When the agent takes an action in a state, there are many possible next states and rewards. The transition dynamics function P, formalizes this notion. Given a state S and action a, p tells us the joint probability of next state S prime and reward are. In this course, we will typically assume that the set of states, actions, and rewards are finite. Since p is a probability distribution, it must be non-negative



$$p(s', r \mid s, a)$$

$$p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$$

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) = 1, \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$$

Figure 7: probabity criteria

and it's sum over all possible next states and rewards must equal one. Here the future state and reward only depends on the current state and action. This is called the **Markov property**.

5.1 An MDP Example

Consider recycling robot which collects empty soda cans in an office environment. It can detect soda cans, pick them up using his gripper, and dropped them off in a recycling bin. The robot runs in a rechargeable battery. Its objective is to collect as many cans as possible. Let's formulate this problem as an MDP. We will start with the states, actions, and rewards. Let's assume that the sensors can only distinguish two charged levels, low and high. These charged levels represent the robot's state. In each state, the robot has three choices. It can search for cans for a fixed amount of time, it can remain stationary and wait for someone to bring in a can, or it can go to the charging station to recharge its battery. We only allow recharging from the low state because recharging is pointless when the energy level is high Searching for cans when the energy level is high might reduce the energy level to low. That is the search action in the state high might not change the state. Let's say with probability α, the energy level might drop to low with probability $1 - \alpha$. In both cases, the robots search yields a reward of r_{search}. For instance, r_{search} could be $+10$ indicating that the robot found 10 cans. The robot can also wait. Waiting for cans does not drain the battery, so the state does not change.

In both cases, the wait action yields a reward of r_{wait}. For example, r_{wait} could be plus one. Searching when the energy level is low might deplete the battery, then

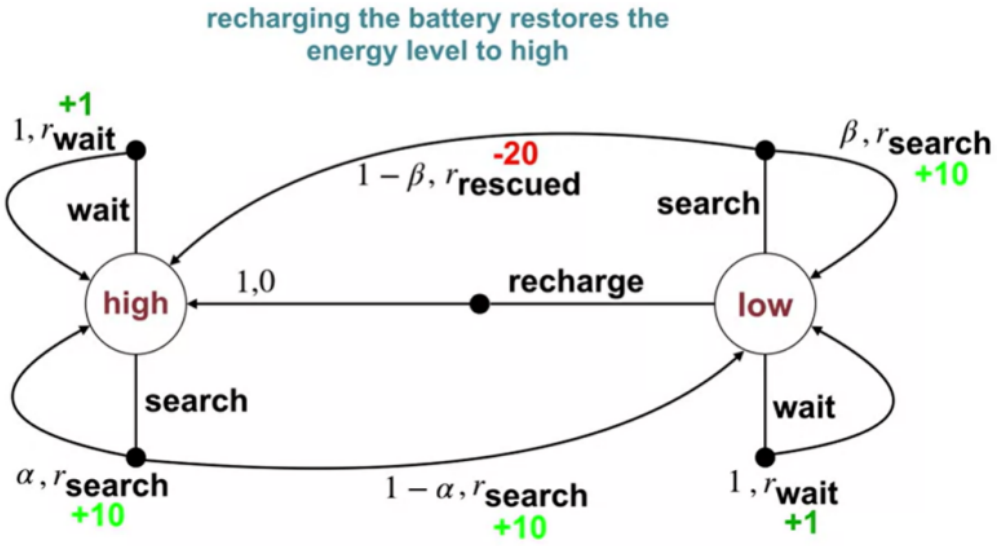


Figure 8: recyling robot example as MDP

the robot would need to be rescued. Let's write this probability as one minus Beta. If the robot is rescued then its battery is restored. However, needing rescue yields a negative reward of $r_{rescued}$. For example, $r_{rescued}$ could be -20 because we were annoyed with the robot. Alternatively, the battery might not run out. This occurs with probability beta and the robot receives a reward of r_{search}. Taking the recharge action restores the battery to the level high and receives a reward at zero.In this way we formalized this example as a MDP

5.2 Goal of reinforcement learning

In reinforcement learning, the agent's objective is to maximize future reward.Now, let's formally define what we mean by maximizing total future reward. The return at time step t, is simply the sum of rewards obtained after time step t. We denote the return with the letter G

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ...$$

The return is a random variable because the dynamics of the MDP can be stochastic. For this to be well-defined, the sum of rewards must be finite. Specifically, let say there is a final time step called T where the agent environment interaction ends the interaction naturally breaks into chunks called **episodes**. Each episode begins independently of how the previous one ended. At termination, the agent is reset to a start state. Every episode has a final state which we call the terminal state.
For example a single game of chess is an episode and next game starts independently to the previous one.

5.3 Reward hypothesis

The reward hypothesis suggests that intelligent behaviour can be achieved by designing agents to maximize cumulative rewards.By this method agents can autonomously learn and adapt their behaviour to effectively navigate complex and changing environment, leading to make intelligent decision making.
Some times in the mathematical form of cumulative rewards, the terms are multiplied by a geometric series of a number called α which is called **Discount factor**. This reflects in a decrease of contribution in the future rewards.

5.4 Continuing tasks

Before discussing about continuing tasks lets go through the characterstics of episodic tasks. Episodic tasks are independent, they have a terminal state and their interaction naturally breaks into episodes. But in continuing tasks its quite opposite and in the episodic tasks the return value G is the sum of rewards upto timestep T but in the continuing tasks there is a quantity called discount factor which we already discussed in the reward hypothesis.

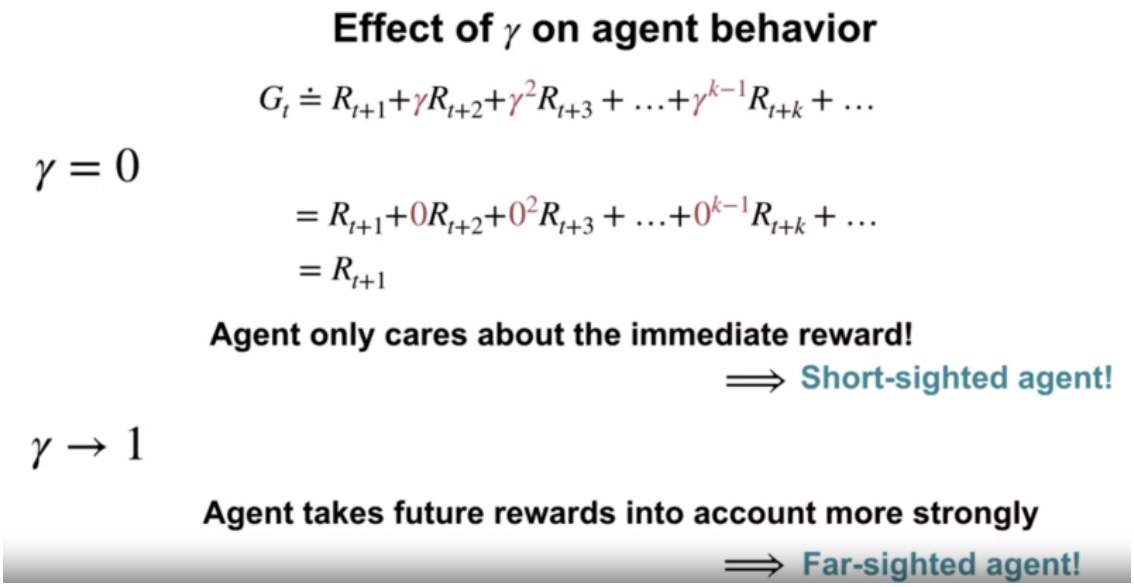


Figure 9: Continuing tasks and its features

6 Value functions and Policy functions

6.1 Policies

A Policy is a distribution over actions for each state
In the simplest case, a policy maps each state to a single action. This kind of policy

is called the **deterministic policy**. We will use the fancy Greek letter π to denote a policy. π of S represents the action selected in state S by the policy. Simply a deterministic policy is a mapping between states and actions.

A **stochastic policy** is one where multiple actions may be selected with non-zero probability. The sum over all action probabilities must be one for each state, and each action probability must be non-negative. It's important to remember that policies depend only on the current state, not on other things like time or previous states.

6.2 Value functions

A value function is a function that assigns a value to each possible state or state-action pair in an environment. It represents the expected return or cumulative reward that an agent can expect to obtain by starting from a particular state or taking a specific action in that state.

In simpler terms, a value function helps an agent evaluate how good or valuable a particular state or action is. It provides a measure of the potential long-term rewards that can be achieved from that state or action. By using the value function, an agent can make decisions that maximize its expected return, leading to optimal decision-making in the given environment.

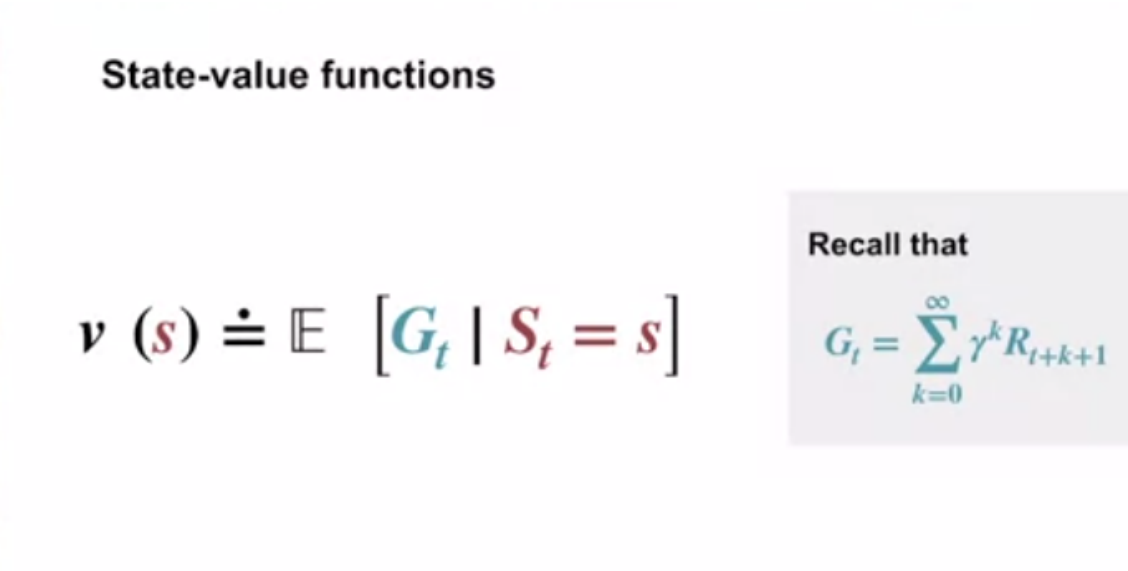


Figure 10: State value function

- **State value function**: A state value function is the future award an agent can expect to receive starting from a particular state. More precisely, the

state value function is the expected return from a given state

So a value function is defined with respect to a given policy. The subscript Π indicates the value function is contingent on the agent selecting actions according to Pi. Likewise, a subscript π on the expectation indicates that the expectation is computed with respect to the policy Pi

- **Action value function**:An action value function is the expected return when the agent first selects a particular action. More formally, the action value of a state is the expected return if the agent selects action A and then follows policy Pi.

Value functions are crucial in reinforce learning, they allow an agent to query the quality of its current situation instead of waiting to observe the long-term outcome.Value function enable us to judge the quality of different policies.

Chess Example(Episodic task):

Figure 11: Use of value functions

For example, consider an agent playing the game of chess. Chess has an episodic MDP, the state is given by the positions of all the pieces on the board, the actions are the legal moves, and termination occurs when the game ends in either a win, loss, or draw. We could define the reward as plus one for winning and zero for all the other moves. This reward does not tell us much about how well the agent is playing during the match, we'll have to wait until the end of the game to see any non-zero reward. The value function tells us much more. The state value is equal

to the expected sum of future rewards. Since the only possible non-zero reward is plus one for winning, the state value is simply the probability of winning if we follow the current policy Pi.

In this two player game, the opponent's move is part of the state transition. For example, the environ moves both the agents piece, circled in blue, and the opponent's piece, circled in red. This puts the board into a new state, S prime. Note, the value of state S prime is lower than the value of state S. This means we are less likely to win the game from this new state assuming we continue following policy Pi. An action value function would allow us to assess the probability of winning for each possible move given we follow the policy Pi for the rest of the game.

Grid Example(continuing task):

This is a simple continuing MDP. The states are defined by the locations on the

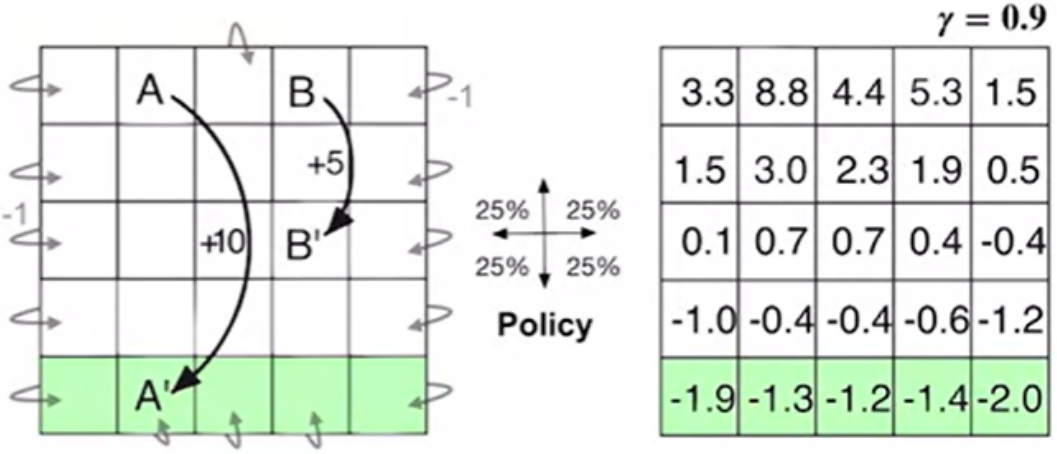


Figure 12: Use of value functions

grid, the actions move the agent up, down, left, or right. The agent cannot move off the grid and bumping generates a reward of minus one. Most other actions yield no reward. There are two special states however, these special states are labeled A and B. Every action in state A yields plus 10 reward and plus five reward in state B. Every action in state A and B transitions the agents to states A prime and B prime respectively. Remember, we must specify the policy before we can figure out what the value function is. Let's look at the uniform random policy. Since this is a continuing task, we need to specify Gamma, let's go with 0.9.

Later, we will learn several ways to compute and estimate the value function, but this time we'll be nice to you and computed for you. On the right, we have written the value of each state. First, notice the negative values near the bottom, these values are low because the agent is likely to bump into the wall before reaching the distance states A and B. Remember, A and B are both the only sources of positive reward in this MDP. State A has the highest value, notice that the value is less than 10 even though every action from state A generates a reward of plus 10, why? Because every transition from A moves the agent close to the lower wall and near the lower wall, the random policy is likely to bump and get negative reward. On the other hand, the value of state B is slightly greater than five. The transition from B moves the agent to the middle. In the middle, the agent is unlikely to bump and is close to the high-valued states A and B.

Upto now we understood the use of value functions indetermining the policies. Now we will focus on how to compute value functions

7 Bellman Equations

Bellman equations relate current and future values. That means these equations establish a relation between the present value functions and future value functions.

To derive this relationship we use definitions of the state value function and action value function and return. For derivation refer figure[11]

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi\left[G_t \mid S_t = s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s\right] \\ &= \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a)\left[r + \gamma \mathbb{E}_\pi\left[G_{t+1} \mid S_{t+1} = s'\right]\right] \\ &= \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a)\left[r + \gamma v_\pi(s')\right] \end{aligned}$$

Figure 13: Derivation for state value function

Action-value Bellman equation

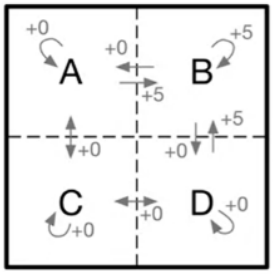
$$
\begin{aligned}
q_\pi(s,a) &\doteq \mathbb{E}_\pi\left[G_t \mid S_t = s, A_t = a\right] \\
&= \sum_{s'}\sum_{r} p(s',r\mid s,a)\left[r + \gamma \mathbb{E}_\pi\left[G_{t+1} \mid S_{t+1} = s'\right]\right] \\
&= \sum_{s'}\sum_{r} p(s',r\mid s,a)\left[r + \gamma \sum_{a'} \pi(a'\mid s')\mathbb{E}_\pi\left[G_{t+1} \mid S_{t+1} = s', A_{t+1} = a'\right]\right] \\
&= \sum_{s'}\sum_{r} p(s',r\mid s,a)\left[r + \gamma \sum_{a'} \pi(a'\mid s') q_\pi(s',a')\right]
\end{aligned}
$$

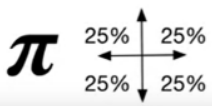
Figure 14: Derivation for action value function

7.1 Calculating value functions using Bellman equations

I will show an Grid example to explain this.
This example consists of just four states, labeled A, B, C and D on a grid. The action space consists of moving up, down, left and right. Actions which would move off the grid, instead keep the agent in place. Say for example we start in state C, moving up would take us to state A. If we then try to move left we would hit a

Example: Gridworld





$$V_\pi(s) = \sum_a \pi(a\mid s)\sum_r\sum_{s'} p(s',r\mid s,a)[r + \gamma V_\pi(s')]$$

$$V_\pi(A) = \frac{1}{4}(5 + 0.7V_\pi(B)) + \frac{1}{4}0.7V_\pi(C) + \frac{1}{2}0.7V_\pi(A)$$

$$V_\pi(B) = \frac{1}{2}(5 + 0.7V_\pi(B)) + \frac{1}{4}0.7V_\pi(A) + \frac{1}{4}0.7V_\pi(D)$$

$$V_\pi(C) = \frac{1}{4}0.7V_\pi(A) + \frac{1}{4}0.7V_\pi(D) + \frac{1}{2}0.7V_\pi(C)$$

$$V_\pi(D) = \frac{1}{4}(5 + 0.7V_\pi(B)) + \frac{1}{4}0.7V_\pi(C) + \frac{1}{2}0.7V_\pi(D)$$

Figure 15

wall and stay in state A. The reward is 0 everywhere except for any time the agent lands in state B. If the agent lands in state B, it gets a reward of +5.

This includes starting in state B and hitting a wall to remain there. Let's consider the uniform random policy, which moves in every direction 25 percent of the time. The discount factor gamma 0.7.
Using the Bellman equation, we can write down an expression for the value of state A in terms of the sum of the four possible actions and the resulting possible successor states. We can simplify the expression further in this case, because for each action there's only one possible associated next state and reward. That's the sum over s prime and r reduces to a single value. If we go right from state A, we land in state B, and receive a reward of +5. This happens one quarter of the time under the random policy. If we go down, we land in state C, and receive no immediate reward. Again, this occurs one-quarter of the time. If you go either up or left, we will land back in state A again. Since they both land in state A and received no reward, we combine them into a single term with factor of 1 over 2. Finally, we arrived at the expression shown here for the value of state A. We can write down a similar equation for each of the other states, B, C, and D. Now we have a system of for equations for four variables.
By solving those equations we get

$V_\pi(A)$=4.2 ,$V_\pi(B)$=6.1 ,$V_\pi(C)$=2.2,$V_\pi(D)$=4.2.
Here we solved 4 linear equations to compute the value of value functions. But we may face many incidents like we have to solve some large number of linear equations like in case of a chess example which includes lot of actions and lots of types of rewards and states.To solve such large system of equations we use algorithms.

7.2 Optimal policies

Optimal policy is a policy that is atleast as good as every other policy.To define an optimal policy, we first have to understand what it means for one policy to be better than another. Here we can see the value of two policies plotted across states.

This plot illustrates that in some states, π_1 achieves a higher value, and in other states π_2 achieves a higher value. So it does not make much sense to say π_1 is better than π_2, or that π_2 is better than π_1. We will say policy π_1 is as good as or better than policy π_2, if and only if the value under π_1 is greater than or equal to the value under π_2 for every state. There's always at least one optimal policy, but there may be more than one. We'll use the notation π_* to denote any optimal policy.

We could combine these policies into a third policy π_3, which always chooses actions according to whichever of policy π_1 and π_2 has the highest value in the current state. π_3 will necessarily have a value greater than or equal to both π_1 and π_2 in every state. So we will never have a situation we're doing well in one state require sacrificing value in another. Because of this, there always exists some policy which

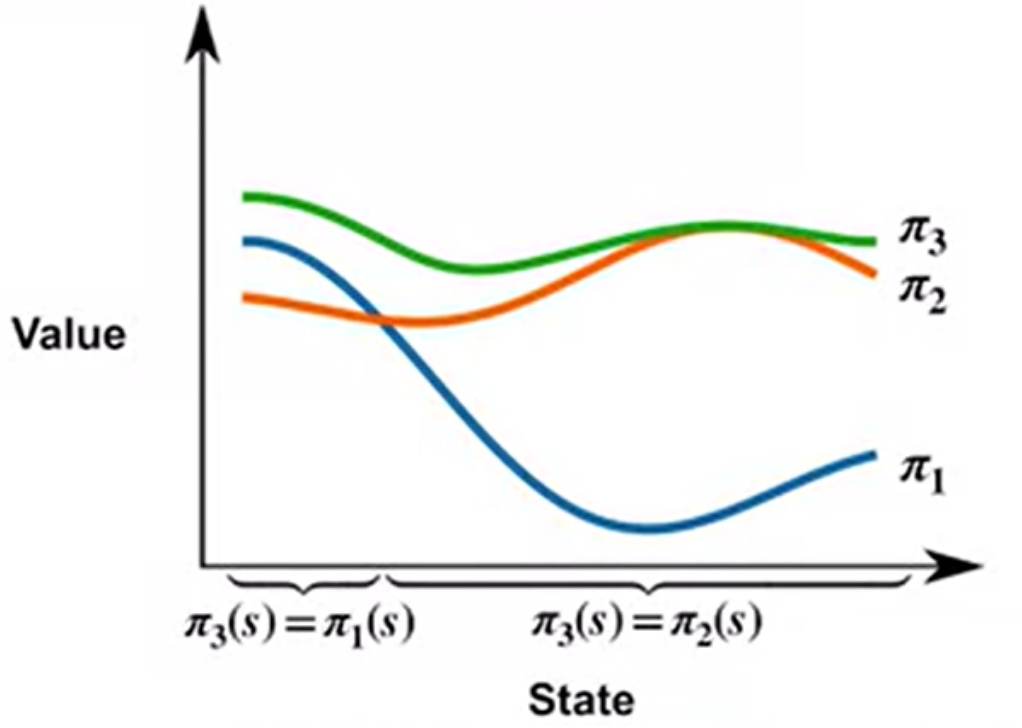


Figure 16: An illustration to define optimal policy

is best in every state.

7.2.1 Example on optimal policies

This example consists of three possible states. here the two possible deterministic policies are choosing state A_1 or A_2. we have to calculate the values for both of them to decide which is optimal policy.That calculation was shown in the figure below.

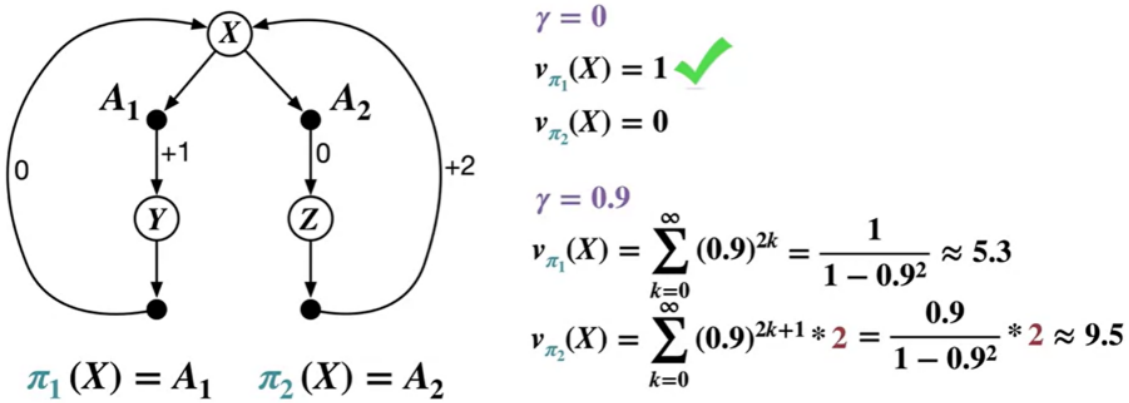


Figure 17: An example to decide optimal policy

In the first case when the discount factor is small π_1 is optimal policy and in the second case π_2 is the optimal policy. In general, it will not be so easy. Even if we limit ourselves to deterministic policies, the number of possible policies is equal to the number of possible actions to the power of the number of states. We could use a brute force search where we compute the value function for every policy to find the optimal policy. But it's not hard to see this will become intractable for even moderately large MDPs. Luckily, there's a better way to organize our search of the policy space. The solution will come in the form of yet another set of Bellman equations, called the Bellman's Optimality equations.

7.3 Bellman's Optimality equations

Just replace the policy part in the bellman equations by optimal policy functions.Remember there always exists an optimal deterministic policy, one that selects an optimal action in every state. Such a deterministic optimal policy will assign Probability 1, for an action that achieves the highest value and Probability 0, for all other actions. We can express this another way by replacing the sum over π_* with a max over a. Notice that π_* no longer appears in the equation. we discussed

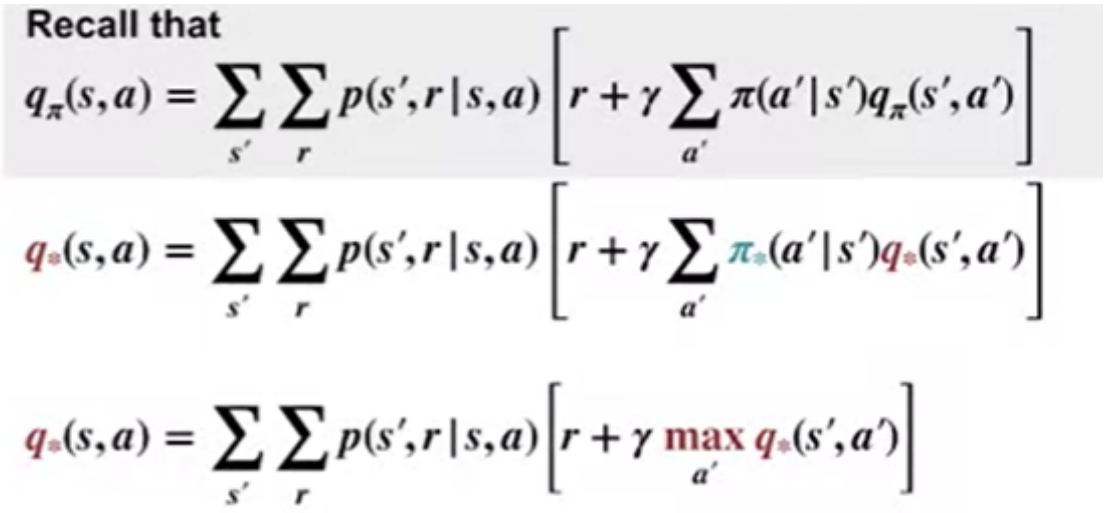


Figure 18: Bellman's Optimality equations

how the Bellman equations form a linear system of equations that can be solved by standard methods. The Bellman's optimality equation gives us a similar system of equations for the optimal value. One natural question is, can we solve this system in a similar way to find the optimal state-value function? Unfortunately, the answer is no. Taking the maximum over actions is not a linear operation. So standard techniques from linear algebra for solving linear systems won't apply.

7.3.1 Calcualting optimal values in a grid

Let's just look at an example where we have already worked out the optimal state value function, v star. Specifically, let's take another look at the grid world we introduced earlier .The reward is zero everywhere else except for -1, for bumping into the walls.The discount factor is 0.9. To see the estimated optimal values see the figure. Notice that unlike before, the values along the bottom are not

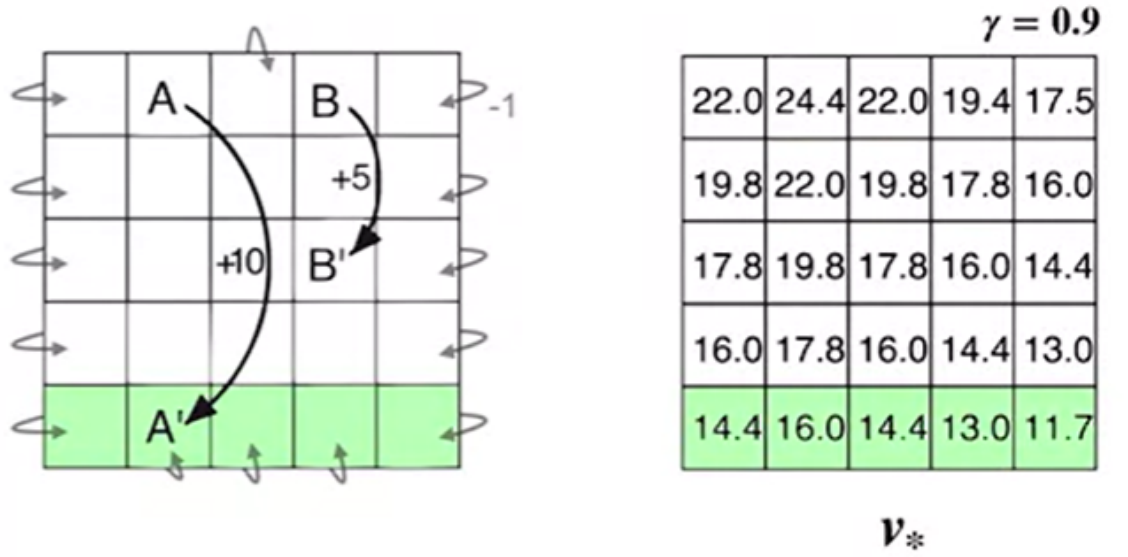


Figure 19: optimal values of each grid

negative. Unlike the uniform random policy, the optimal policy won't ever choose to bump into the walls. As a consequence, the optimal value of state A is also much higher than the immediate reward of +10. Remember that in general, the dynamics function p can be stochastic, so it might not always be so simple. However, as long as we have access to p, we can always find the optimal action from v star by computing the right-hand side of the Bellman optimality equation for each action and finding the largest value.

If instead we have access to q_*, it's even easier to come up with the optimal policy.

In this case, we do not have to do a one step look ahead at all. We only have to select any action a, that maximizes q_* of s and a.

The action-value function caches the results of a one-step look ahead for each action. In this sense, the problem of finding an optimal action-value function corresponds to the goal of finding an optimal policy.

8 Policy evaluation and Control

- Defnitions

Policy evaluation is the task of determining the value function for a specific policy.
Control is the task of finding a policy to obtain as much reward as possible. In other words, finding a policy which maximizes the value function.
We already learnt that by knowing the values π,p,γ we can find the value of v_π, by writing bellman equation for each state and solving those linear system of equations.v
Instead pf linear system solver we now use dynamic programming to find the value function.
Control is the task of improving a policy. recall that a policy π_2 is considered as good as or better than π_1 if the value under π_2 is greater than or equal to the value under π_1 in every state.

We say π_2 is strictly better than π_1 if π_2 is as good as or better than π_1 and there's at least one state where the value under π_2 is strictly greater than the value under π_1. The goal of the control task is to modify a policy to produce a new one which is strictly better.Moreover, we can try to improve the policy repeatedly to obtain a sequence of better and better policies. When this is no longer possible, it means there is no policy which is strictly better than the current policy. And so the current policy must be equal to an optimal policy. And we can consider the control task complete.

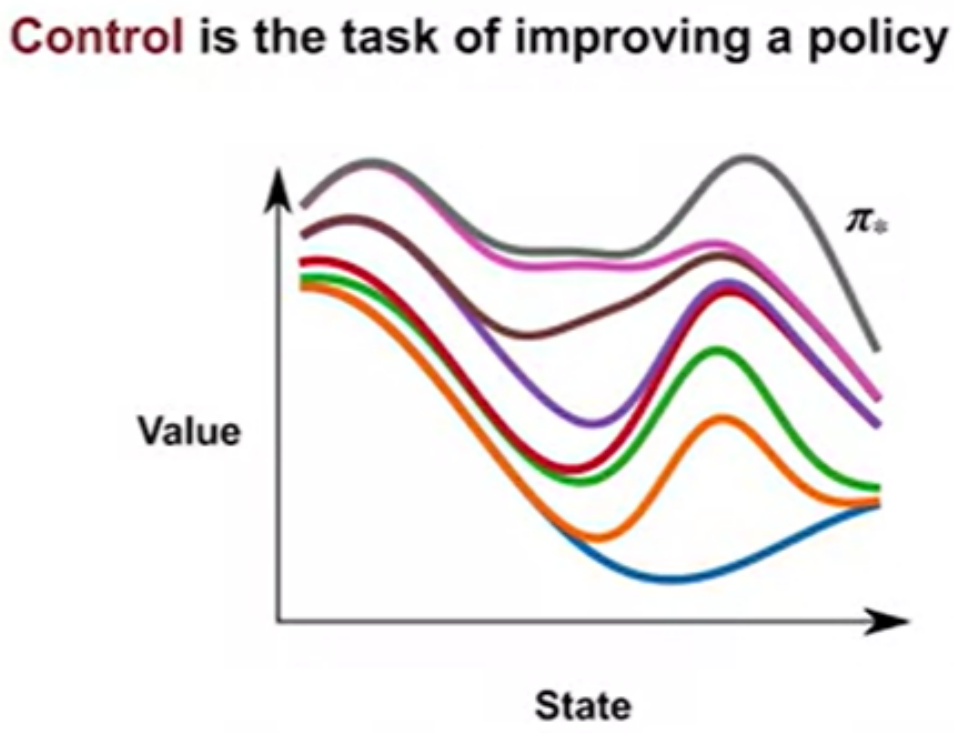


Figure 20: Improving the policy(Control)

8.1 Iteracy policy evaluation

The idea of iterative policy evaluation is so simple that at first it might seem a bit silly. We take the Bellman equation and directly use it as an update rule.We begin with an arbitrary initialization for our approximate value function, let's call this v_0. Each iteration then produces a better approximation by using the update rule shown at the top of the slide. Each iteration applies this updates to every state, S, in the state space, which we call a sweep. Applying this update repeatedly leads to a better and better approximation to the state value function v_Π. n fact, it can be proven that for any choice of v_0, v_k will converge to v_Π in the limit as k approaches infinity. To implement iterative policy evaluation, we store two arrays,

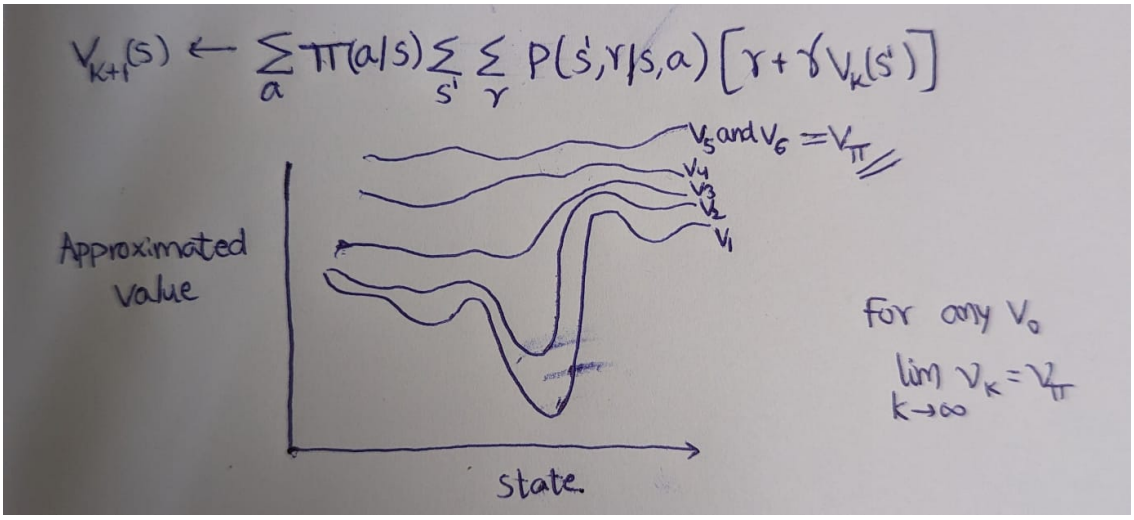


Figure 21

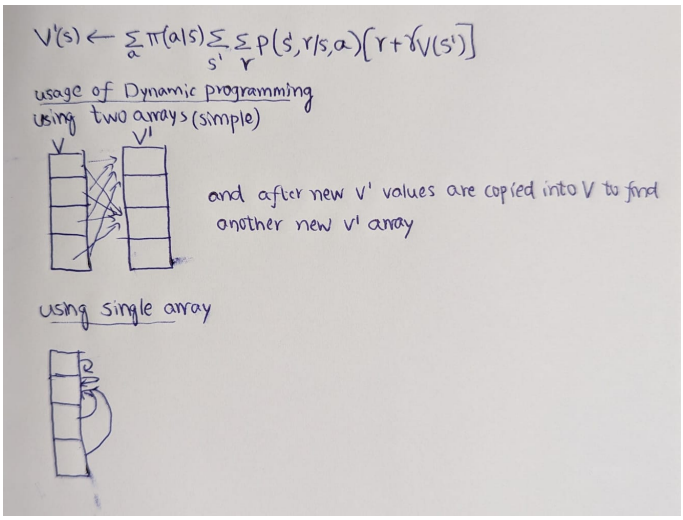


Figure 22

each has one entry for every state. One array, which we label V stores the current approximate value function. Another array, V prime, stores the updated values. By using two arrays, we can compute the new values from the old one state at a time without the old values being changed in the process.At the end of a full sweep, we can write all the new values into V; then we do the next iteration. It is also possible to implement a version with only one array, in which case, some updates will themselves use new values instead of old. This single array version is still guaranteed to converge, and in fact, will usually converge faster.

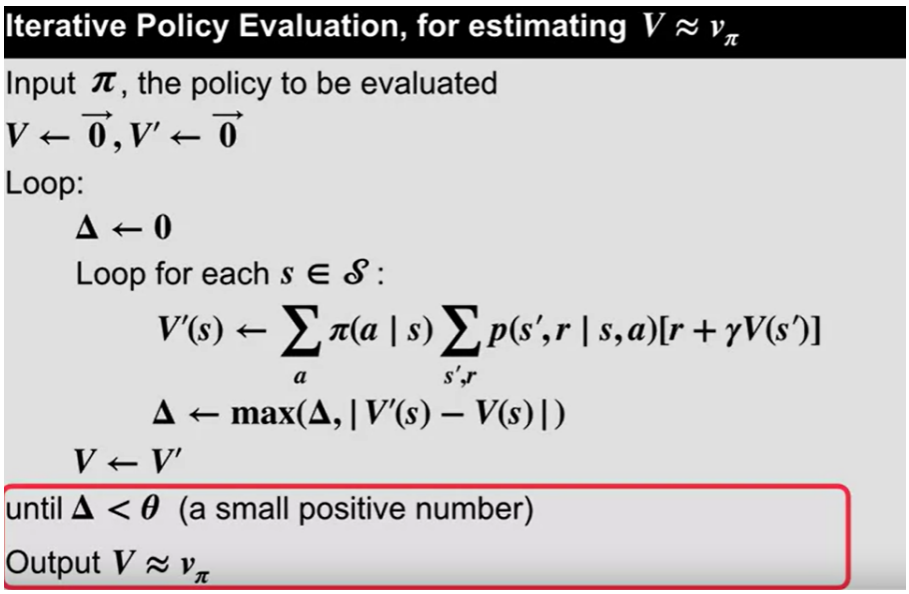


Figure 23

8.2 Policy improvement

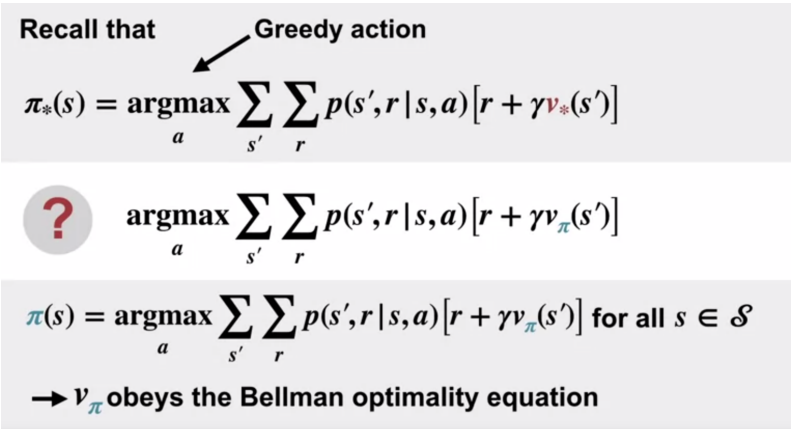


Figure 24: Equation showing policy improvement

Previously, we showed that given V*, we can find the optimal policy by choosing the Greedy action. The Greedy action maximizes the Bellman's optimality equation in each state. Imagine instead of the optimal value function, we select an action which is greedy with respect to the value function v_Π of an arbitrary policy Π. What can we say about this new policy? That it is greedy with respect to v_Π. The first thing to note is that this new policy must be different than Π . If this greedification doesn't change Π, then Π was already greedy with respect to its own value function. This is just another way of saying that v_Pi obeys the Bellman's optimality equation. In which case, Π is already optimal. In fact, the new policy obtained in this way must be a strict improvement on Π, unless Π was already optimal. This is a consequence of a general result called the policy improvement theorem. Policy Π' is at least as good as Π if in each state, the value of the action selected by Π' is greater than or equal to the value of the action selected by Π. Policy π prime is strictly better if the value is strictly greater and at least one state.

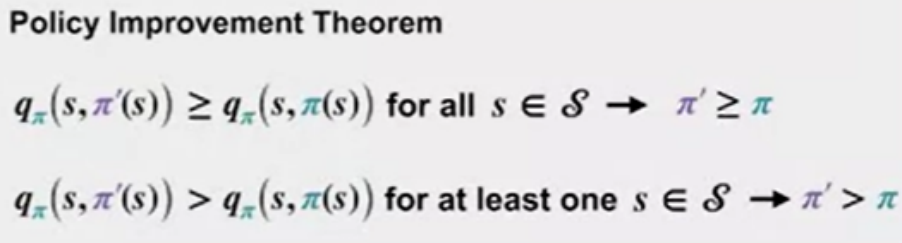


Figure 25: Policy Improvement Theorem

8.3 Policy Iteration

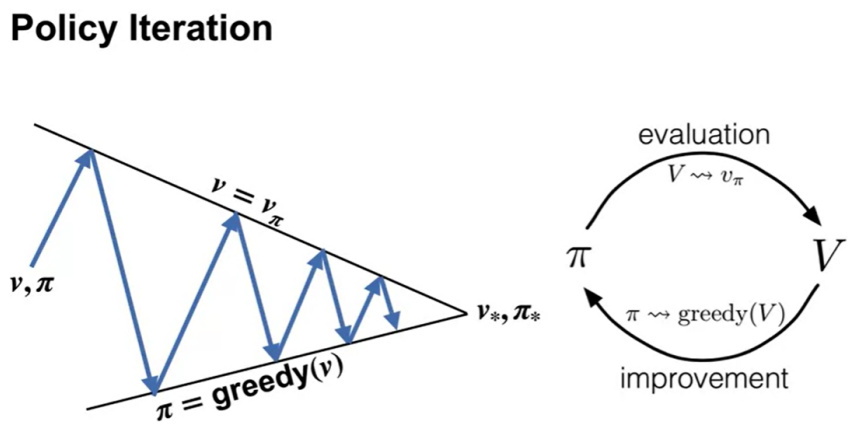


Figure 26: Policy Iteration

Let's say we begin with the policy Pi 1. We can evaluate Pi 1 using iterative policy evaluation to obtain the state value, V Pi 1. We call this the evaluation step. Using the results of the policy improvement theorem, we can then greedify with respect to v Pi 1 to obtain a better policy, Pi 2. We call this the improvement step. We can then compute V Pi 2 and use it to obtain an even better policy, Pi 3. This gives us a sequence of better policies. Each policy is guaranteed to be an improvement on the last unless the last policy was already optimal. So when we complete an iteration, and the policy remains unchanged, we know we have found the optimal policy. At that point, we can terminate the algorithm. Each policy generated in this way is deterministic. There are finite number of deterministic policies, so this iterative improvement must eventually reach an optimal policy. This method of finding an optimal policy is called policy iteration. Policy iteration consists of two distinct steps repeated over and over, evaluation and improvement. This dance of policy and value proceeds back and forth, until we reach the only policy, which is greedy with respect to it's own value function, the optimal policy. At this point, and only at this point, the policy is greedy and the value function is accurate. Here's what this procedure looks like in pseudocode.

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
      $\Delta \leftarrow 0$
      Loop for each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))[r + \gamma V(s')]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      old-action $\leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
      If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Figure 27: Policy Improvement Theorem

8.4 Generalized Policy Iteration

The policy iteration algorithm runs each step all the way to completion. Intuitively, we can imagine relaxing this. Imagine instead, we follow a trajectory like this. Each evaluation step brings our estimate a little closer to the value of the current policy but not all the way. Each policy improvement step makes our policy a little more greedy, but not totally greedy.We will use the term generalized policy iteration to refer to all the ways we can interleave policy evaluation and policy improvement. This brings us to our first generalized policy iteration algorithm, called value iteration. In value iteration, we still sweep over all the states and greedify with respect to the current value function. However, we do not run policy evaluation to completion. We perform just one sweep over all the states. After that, we greedify again. We can write this as an update rule which applies directly to the state value function. The update does not reference any specific policy, hence the name value iteration. The full algorithm looks very similar to iterative policy evaluation. Instead of updating the value according to a fixed falsey, we update using the action that maximizes the current value estimate. Value iteration still converges to V star in the limit.The full algorithm looks very similar to iterative policy evaluation. Instead of updating the value according to a fixed falsey, we update using the action that maximizes the current value estimate. Value iteration still converges to V star in the limit. We can recover the optimal policy from the optimal value function by taking the argmax. Value iteration sweeps the entire state space on each iteration just like policy it-

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $v \leftarrow V(s)$
| $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
| $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \text{argmax}_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$

Figure 28: Algorithm for Value iteration

eration. Methods that perform systematic sweeps like this are called synchronous. This can be problematic if the statespace is large. Every sweep could take a very long time. Asynchronous dynamic programming algorithms update the values of states in any order, they do not perform systematic sweeps. They might update a given state many times before another is updated even once. In order to guarantee

convergence, asynchronous algorithms must continue to update the values of all states. for example, the algorithm updates the same three states forever ignoring all the others. This is not acceptable because the other states cannot be correct if they are never updated at all. Asynchronous algorithms can propagate value information quickly through selective updates. Sometimes this can be more efficient than a systematic sweep. For example, an asynchronous method can update the states near those that have recently changed value.

8.4.1 Alternative methods for Policy evaluation and Policy iteration

For policy evaluation

- **Montecarlo Method**:we gather a large number of returns under pi and take their average.This will eventually converge to the state value,this is called the Monte Carlo method.However, if we do it this way, we may need a large number of returns from each state.Each return depends on many random actions, selected by pi, as well as many random state transitions due to the dynamics of the MDP. We could be dealing with a lot of randomness here,each return might be very different than the true state value. So we may need to average many returns before the estimate converges, and we have to do this for every single state.
- **Boot strapping**:The key insight of dynamic programming is that we do not have to treat the evaluation of each state as a separate problem. We can use the other value estimates we have already worked so hard to compute.This process of using the value estimates of successor states to improve our current value estimate is known as bootstrapping. This can be much more efficient than a Monte Carlo method that estimates each value independently.

For Finding optimal policy

- **Brute-force search**:This method simply evaluates every possible deterministic policy one at a time, we then pick the one with the highest value. There are a finite number of deterministic policies, and there always exists an optimal deterministic policy. So brute-force search will find the answer eventually, however, the number of deterministic policies can be huge.A deterministic policy consists of one action choice per state. So the total number of deterministic policies is exponential in the number of states. Even on a fairly simple problem, this number could be massive, this process could take a very long time.
- **Dynamic programming**: policy iteration is guaranteed to find the optimal policy in time polynomial in the number of states and actions.Thus, dynamic programming is exponentially faster than the brute-force search of the policy

space.In practice, dynamic programming is usually much faster, even in this worst-case guarantee.

8.5 An example describing Iteracy policy evaluation

At first we follow uniform random policy, 25 percent in all directions.In the below figure the values on grid on the left side are assumed as initial values(V_0).And stepping into the -1 valued grids will give a reward of -1 and stepping into other two grids will give value of 0. This means the optimal policy should show the shortest path to the zero valued grids. After some iterations, the values become

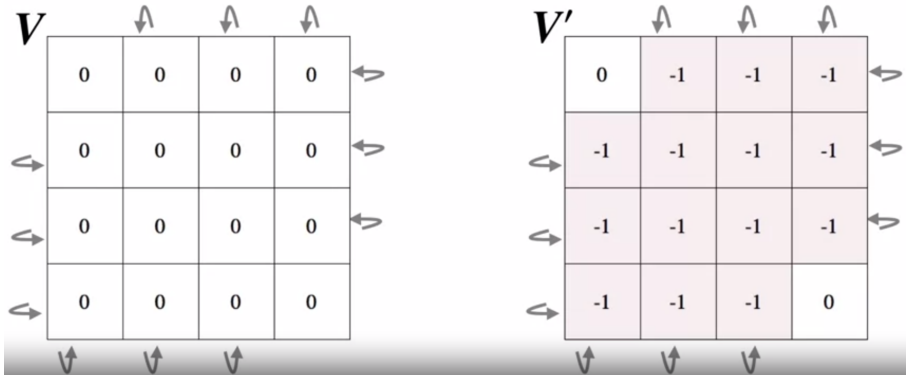


Figure 29: First iteracy policy evaluation step

constant and will not change and those are the values of this example following uniform random policy As we already learnt policy iteration i am going to show

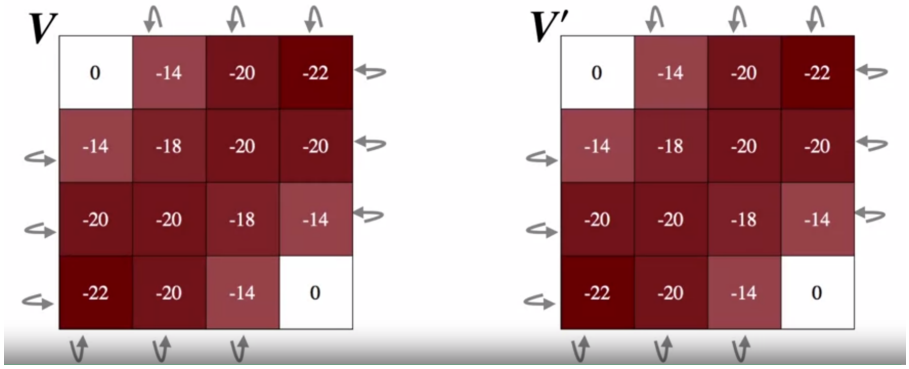


Figure 30: After many steps

the greedy policy with respect to the final constant value functions in the following figure. We can observe that this final policy turns out to be the optimal policy because those arrows showed will represent the shortest path to the reward 0 grids.

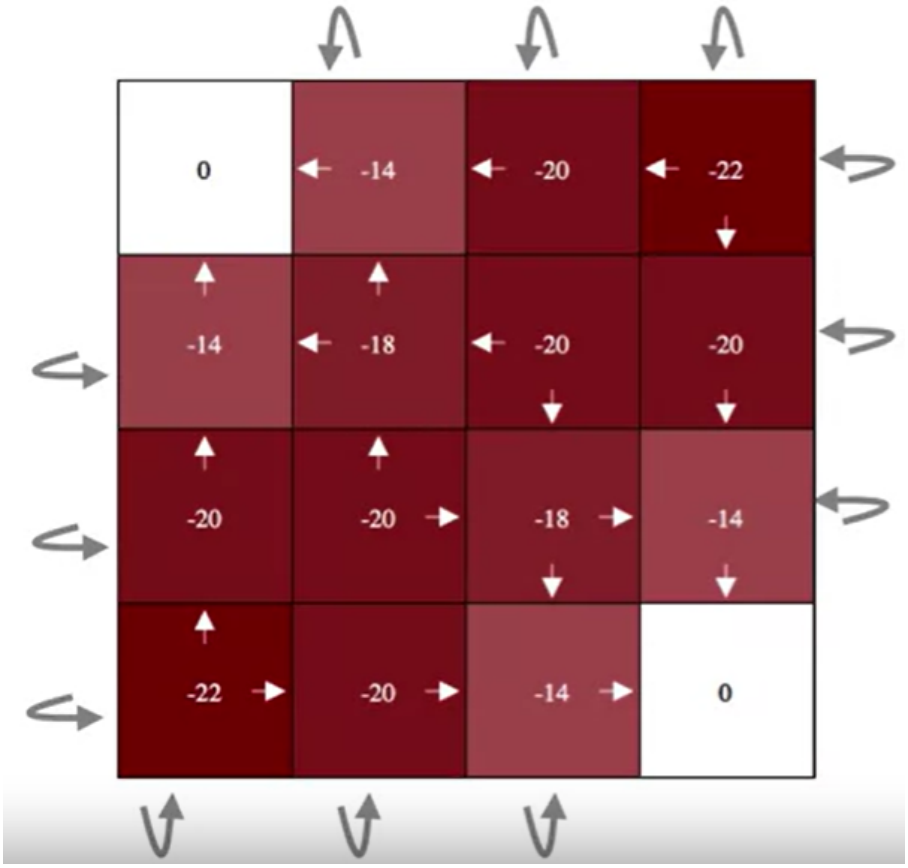


Figure 31: Greedy policy with respect to final values

9 Sample based learning methods

9.1 Montecarlo methods

The term Monte Carlo is often used more broadly for any estimation method that relies on repeated random sampling. In RL Monte Carlo methods allow us to estimate values directly from experience, from sequences of states, actions and rewards.

So a Monte Carlo method for learning a value function would first observe multiple returns from the same state. Then, it average those observed returns to estimate the expected return from that state. As the number of samples increases, the average tends to get closer and closer to the expected return. The more returns the agent observes from a state, the more likely it is that the sample average is close to the state value. These returns can only be observed at the end of an episode. So we will focus on Monte Carlo methods for episodic tasks.

9.1.1 For finding State value function

Now let's look at an algorithm for estimating the state value function of a policy. The Monte Carlo algorithm has to keep track of multiple observed returns. Let's introduce a list and returns one for each state. Each list holds the returns observed from state S. Then we generate an episode by following our policy. For each date in the episode, we compute the return and start in the list of returns. By working

MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated
Initialize:
    $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
    $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):
    Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        Append G to $Returns(S_t)$
        $V(S_t) \leftarrow$ average$(Returns(S_t))$

Figure 32: Montecarlo algorithm for episodic tasks

backwards from the terminal time-step, we can efficiently compute the returns for each state encountered during the episode. The first return in just the last rewards. So we add the last reward to the list of returns for S_{t-1}. Then, we set the value of S_{t-1} to be the average of returns S_{t-1}. On the previous time step t-2, we calculate the return as before then added to the list of returns for S_{t-2}. Finally we update the value of S_{t-2}. If we continue this loop until the end we'll, have updated the values for all the states visited in the current episode. Then we can repeat the whole process over many episodes and eventually learn a good estimate for the value function.

9.1.2 For finding Action value functions

Learning action values is almost exactly the same process as learning state values. Recall that we learned the value of a state by averaging sample returns from that state.But when we are using this method to find values for a given policy , the policy may be like avoiding a particular action every time. Then this makes calculation of action values using montecarlo method becomes difficult.This is the problem of maintaining exploration in reinforcement learning.One way to maintain exploration is called **exploring starts**. In **exploring starts**, we must guarantee that episodes start in every state-action pair. Afterwards, the agent simply follows its policy.And the algorithm is similar to the algorithm for finding statevalues.

9.1.3 Using Montecarlo methods for Generalised policy iteration

For the policy improvement step, we can make the policy greedy with respect to the agent's current action value estimates. For the policy evaluation step, we will use a Monte Carlo method to estimate the action values. Remember that in the GPI framework, the value estimates need only improve a little, not all the way to the correct action values. All that is required for convergence is that the estimates continue to improve. Monte Carlo control methods combined policy improvement and policy evaluation on an episode-by-episode basis.

Now, let's put these two together in one algorithm. Let's start with our Monte Carlo method for learning action values. You use exploring starts so that each episode begins with a randomly selected state and action.

Then, the agent generates an episode by following his policy, keeping track of

Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

Initialize:
$\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$
$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Loop forever (for each episode):
Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0
Generate an episode from S_0, A_0, following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
Append G to $Returns(S_t, A_t)$
$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
$\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$

Figure 33: Montecarlo algorithm for episodic tasks

the states, actions, and rewards along the way. Once the episode is complete, it computes each return starting from the end of the episode. Then, it adds the return to the appropriate list. The list of returns are then averaged to update the action value estimates for each state-action pair. This completes the policy evaluation step. After policy evaluation, it's time to do policy improvement. We simply update the policy to take the greedy action with respect to our updated action values. We do this in every state observed during the episode.

9.1.4 Exploration for Montecarlo methods

Let's think about some situations where we cannot use exploring starts this algorithm must be able to start from every possible State action pair. Otherwise the

age of may not explore enough and could converge to a suboptimal solution in many problems. It can be difficult to randomly sample an initial State action pair.For example, how would you randomly sample the initial State action pair for a self-driving car.

So, how can we learn all the action values without exploring starts? Remember Epsilon greedy expiration.Epsilon greedy policies are a subset of a larger class of policies called Epsilon soft policies Epsilon soft policies take each action with probability at least Epsilon over the number of actions. For example, both policies shown on the slide are valid Epsilon soft policies. The uniform random policy is another notable Epsilon Soft policy.Epsilon soft policies Force the agent to continually explore that means we can drop the exploring starts requirement from the Monte Carlo control algorithm an Epsilon soft policy assigns nonzero probability to each action in every state because of this Epsilon soft agents continue to visit all state action pairs indefinitely. exploring starts can be used to find the optimal policy.

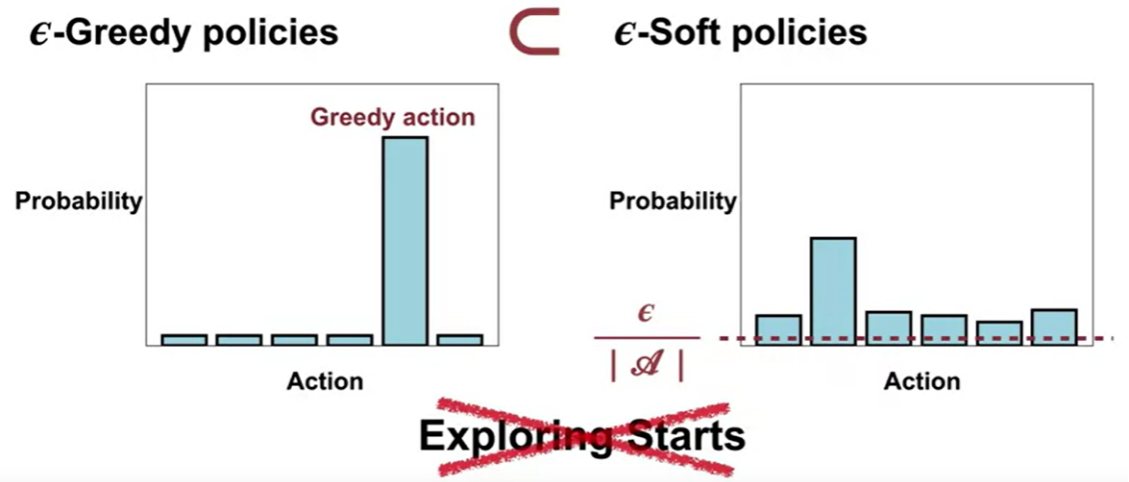


Figure 34: Montecarlo algorithm for episodic tasks

But that's on soft policies can only be used to find the optimal Epsilon soft policy. That is the policy with the highest value in each state out of all the Epsilon soft policies.This policy performs worse than the optimal policy in general. However, it often performs reasonably. Well and allows us to get rid of exploring starts. Even though this algorithm does not find optimal policy. It does find the optimal epsilon soft policy.

9.2 Off Policy learning

The disadvantage of Epsilon soft policies is that they are suboptimal for both acting and learning Epsilon soft policies are neither optimal policies for obtaining reward nor are the optimal for exploring to find the best actions so far throughout this

MC control (for ε-soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ε-soft policy

$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow$ average$(Returns(S_t, A_t))$

$A^* \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken arbitrarily)

For all $a \in \mathcal{A}(S_t)$:

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

Figure 35: Algorithm for ϵ soft policies

On-Policy and Off-Policy

- **On-Policy: improve and evaluate the policy being used to select actions**
- **Off-Policy: improve and evaluate a *different* policy from the one used to select actions**

Figure 36: on policy and off policy

course, we have implicitly been discussing on policy learning though. For example, you could learn the optimal policy while following a totally random policy we call the policy that the agent is learning the target policy because it is the target of the agents learning the target policy is usually denoted by π. The value function that the agent is learning is based on the target policy one example of a Target policy is the optimal policy we call the policy that the agent is using to select actions the behavior policy because it defines our agents Behavior. The behavior policy is usually denoted by B. The behavior policy is in charge of selecting actions for the agent. One key rule of off policy learning is that the behavior policy must cover the target policy? In other words, if the target policy says the probability of selecting

an action a given State s is greater than zero then the behavior policy must say the probability of selecting that action in that state is greater than 0

9.2.1 Importance sampling

Importance sampling,an example of off-Policy learning, allows us to do off-policy learning, learning with one policy while following another. The importance sampling ratio$\rho(x) = \pi(x)/b(x)$ is introduced to correct the expectation under B and rewrite it as an expectation under π. To estimate the expected value under π from data sampled from B, a weighted sample average is computed using the importance sampling ratio as weights. This allows us to approximate the desired expectation despite using data from a different distribution. Formula for Importance Sampling is: where, x is random variable sampled from behaviour policy B, $\rho(x)$ is impor-

$$\begin{aligned}\mathbb{E}_\pi[X] &\doteq \sum_{x\in X} x\pi(x) \\ &= \sum_{x\in X} x\pi(x)\frac{b(x)}{b(x)} \\ &= \sum_{x\in X} x\rho(x)b(x)\end{aligned}$$

Figure 37: derivation for formula of importance sampling

tance sampling ratio ,the ratio between the target distribution π and the behavior distribution B.

$$\mathbb{E}_\pi[X] \approx \frac{1}{n}\sum_{i=1}^{n} x_i\rho(x_i)$$
$$x_i \sim b$$

Figure 38

9.2.2 Off policy montecarlo algorithm

when we try to estimate the value under target policy, π, using returns following a behavior policy, b. If we simply average the returns, we saw from state s under the behavior b, we will not get the right answer. We have to correct each return in the average. This is just what important sampling is for. All we have to do is figure out the value of ρ for each of the sampled returns. ρ, here, is the probability of the trajectory under π divided by the probability of the trajectory under b. This ρ corrects the distribution over entire trajectories, and so corrects the distribution over returns. Using this correction, we get back what we want, the expectation of the return under π. Let's consider the probability distribution over trajectories.

$$\rho = \frac{\mathbb{P}(\textbf{trajectory under } \pi)}{\mathbb{P}(\textbf{trajectory under } b)}$$

$$V_\pi(s) = \mathbb{E}_b\left[\rho G_t \,|\, S_t = s\right]$$

Figure 39: like this the return terms are multiplied by ρ

We read this probability as, given that the agent is in some state as t, what is the probability that it takes action A_t then ends up in state S_{t+1}, then, it takes action A_{t+1} and ends up in S_{t+2} , and so on, until termination at time T , All of the actions are sampled according to behavior b. Because of the Markov property, we can break this probability distribution into smaller chunks. The first chunk is the probability that the agents likes action A_t in state S_t times the probability that the environment transitions into state $S_t + 1$. The second chunk gives the probability of the next time step of experience, and so on. we can take these

$$P(A_t, S_{t+1}, A_{t+1}, \ldots, S_T \,|\, S_t, A_{t:T}) =$$

$$b(A_t \,|\, S_t)\, p(S_{t+1} \,|\, S_t, A_t)\, b(A_{t+1} \,|\, S_{t+1})\, p(S_{t+2} \,|\, S_{t+1}, A_{t+1}) \ldots p(S_T \,|\, S_{T-1}, A_{T-1})$$

Figure 40

probabilities and multiply them by the importance sampling ratio. The transition dynamics of the environment cancel out on each time step. This leaves us with only a product over the ratios between policies at each time step. Now, let's get back to estimating V_π off-policy. The agent observes many returns, each according

to the behavior policy b. We can estimate V_π using these returns by correcting each return with ρ. Now, let's look at how we would implement this. First, recall

$$\mathbb{P}(\textbf{trajectory under } b) = \prod_{k=t}^{T-1} b(A_k \mid S_k)\, p(S_{k+1} \mid S_k, A_k)$$

Figure 41

$$\rho_{t:T-1} \doteq \prod_{k=t}^{T-1} \frac{\pi(A_k \mid S_k)}{b(A_k \mid S_k)}$$

Figure 42

the on-policy Monte Carlo prediction algorithm. There are two changes that will need to be made here. First, the episode will not be generated following π but will be generated following b. Second, the return will need to be corrected using the product of the important sampling ratios. By making these changes, we end up with the off-policy Monte Carlo prediction algorithm. Notice the episode is now

Off-policy every-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated
Initialize:
$V(s) \in \mathbb{R}$, **arbitrarily, for all** $s \in S$
$Returns(s) \leftarrow$ **an empty list, for all** $s \in S$
Loop forever (for each episode):
Generate an episode following $b : S_0, A_0, R_1, S_1 \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$ $W \leftarrow 1$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$
$G \leftarrow \gamma W G + R_{t+1}$
Append G **to** $Returns(S_t)$
$V(S_t) \leftarrow$ **average**$(Returns(S_t))$
$W \leftarrow W \frac{\pi(A_t \mid S_t)}{b(A_t \mid S_t)}$

Figure 43: off policy monte carlo algorithm

generated following the behavior policy. The return is corrected by a new term W,

which is the accumulated product of important sampling ratios on each time step of the episode. We can compute ρ from t to T-1 incrementally. To see why, let's write out the product at each time step. Recall that the Monte Carlo algorithm loops over time steps backwards. So on the first step of the algorithm, W is set to ρ on the last time step. On the next time step, W_2 is the second last ρ times the last ρ, and so on. Each time step adds one additional term to the product and reuses all previous terms. We can compute this recursively without having to store all past values of ρ.

$$\rho_{t:T-1} \doteq \prod_{k=t}^{T-1} \frac{\pi(A_k \mid S_k)}{b(A_k \mid S_k)}$$

$$= \rho_t \rho_{t+1} \rho_{t+2} \cdots \rho_{T-2} \rho_{T-1}$$

$$W_1 \leftarrow \rho_{T-1}$$
$$W_2 \leftarrow \rho_{T-1}\rho_{T-2}$$
$$W_3 \leftarrow \rho_{T-1}\rho_{T-2}\rho_{T-3}$$

Figure 44

9.3 Temporal difference learning

Recall the definition of the discounted return. We saw a while back that this can be written recursively like so. The value of a state at time t is the expected return at time t. We can replace the return inside this expectation with our recursive definition. We can further split up this equation because the linearity of expectation. We then get the expectation of the return on the next step, which is just the value of the next state. Now we have written the value function recursively as well. Let's go back to our incremental Monte Carlo update rule. We want to update toward the return but we don't want to wait for it. We can replace the return at time t with the reward plus the estimate of the return in the next state. We can think of the value of the next state as a stand-in for the return until the end of the episode. So we don't have to wait until the end of the episode, but we still have to wait to the next step. We call this the t target. The terms inside the brackets resemble an error, which we call the TD error. We will often see the TD error denoted by δ_t. TD takes the policy to evaluate as input, it also requires a step size parameter and an initial estimate of the value function. Every episode begins in some initial state S, and from there the agent takes actions according to its policy until it reaches the terminal state. On each step of the episode, we update the values with the TD

$$V(S_t) \leftarrow V(S_t) + \alpha\left[G_t - V(S_t)\right]$$

$$V(S_t) \leftarrow V(S_t) + \alpha\left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right]$$

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

Figure 45

learning rule. We only need to keep track of the previous state to make the update.

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\left[R + \gamma V(S') - V(S)\right]$
    $S \leftarrow S'$
  until S is terminal

Figure 46: TD(0) algorithm

References

[1] URL: https://www.coursera.org/learn/machine-learning.

[2] Andriy burkov. *The hundered page book of machine learning.*

[3] Luca massaron John paul mueller. *Artificial intelligence for dummies.*

[4] russell and norvig. *Artficial intelligence a modern approach.*