

Deep Reinforcement Learning

Palacode Narayana Iyer Anantharaman

23 Aug 2018

Some notes and disclaimers

- In this slide set I have used figures from a few publicly available documents, videos. I have indicated with a note of thanks, the source of figures wherever applicable. I encourage you to look at the original source for more detailed diagrams
- Some examples need a knowledge of cricket game! As these slides were prepared for my students in India, I felt they can relate to these better than “Alpha GO” or “Atari” games.

References

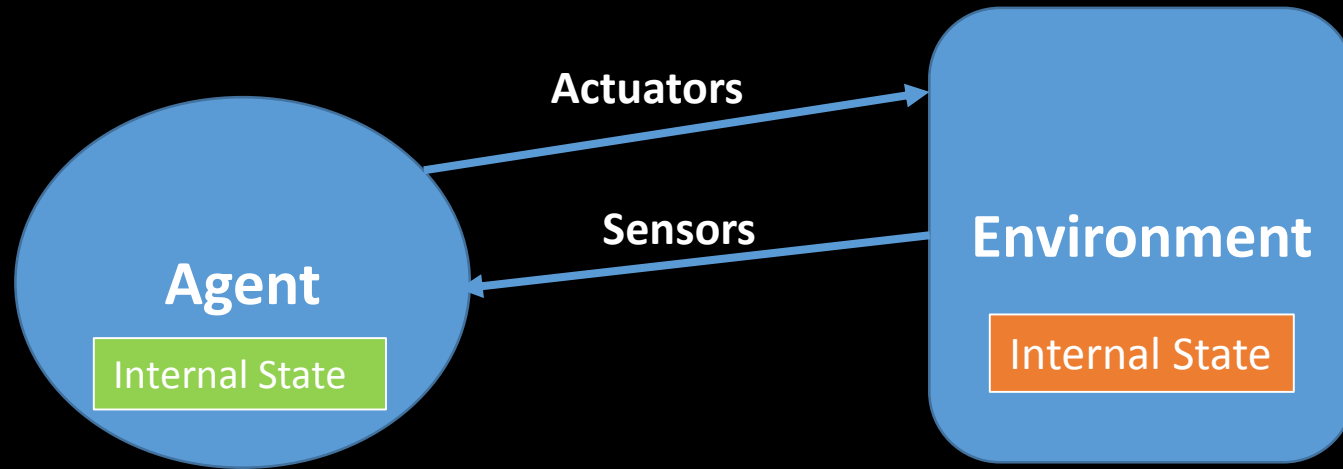
- Deep Reinforcement Learning Hands On by Maxim Lapan from Packt Publications
- Pong to Pixels
 - <http://karpathy.github.io/2016/05/31/rl/>
- Stanford CS231n
 - <https://www.youtube.com/watch?v=lvoHnicueoE>
- Center for Brains Minds and Machines CBMM
 - <https://www.youtube.com/watch?v=JNsgnD8Fr6I>

Goal of this module (2 lecture sessions)

- Provide an overview of the core principles of Deep RL that include concepts like: Markov Decision Process, Q Learning, Policy Gradients etc
- Discuss a couple of application areas
- Describe how to build such models using TensorFlow

Reinforcement Learning is a sub topic of Artificial Intelligence modelled around the paradigm of Agent, Environment interaction.

AI: Agent-Environment Model



- Agent perceives the environment (Percepts) and acts upon the environment in order to maximize achievement of the required goal (Actions)
- Looked at from this perspective, an agent is a function that maps the percepts to actions

Intelligent Agents

- Intelligent agents interact with the Environment
- Interactions through:
 - Sensors
 - Actuators

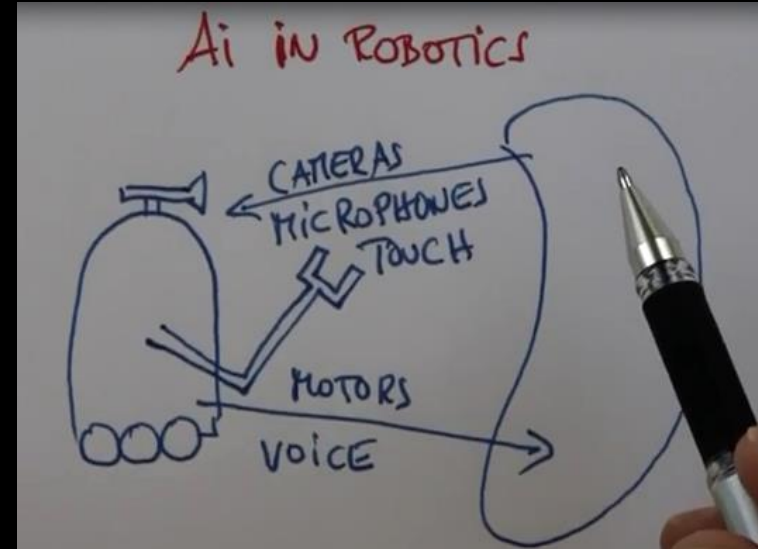


Fig Credit: Sebastian Thrun, Udacity

- The function $f(.)$ that maps the sensor to actuators is the control policy
 - Determines how the agent makes the decisions
 - Those decisions take place many many times in a loop: Perception-Action cycle
- The concept of intelligent agents is abstract. One can cast any real world problem using this model – for instance, it is possible to apply this model to search engine design.

Modelling real world problems

Can we model the following examples? Identify what the sensors are, what are the actuators and how would you describe the agent?

- Home Security Systems: Suppose we have a system that can take pictures or record video on a continuous basis and we are interested in detecting an intruder.
- The speech recognizer: Receive the speech input and transcribe
- Receive the inputs from location sensors and turn the steering wheel
- Look at the last 4 hours data on stock trend of a set of companies and buy the stock of the most promising company

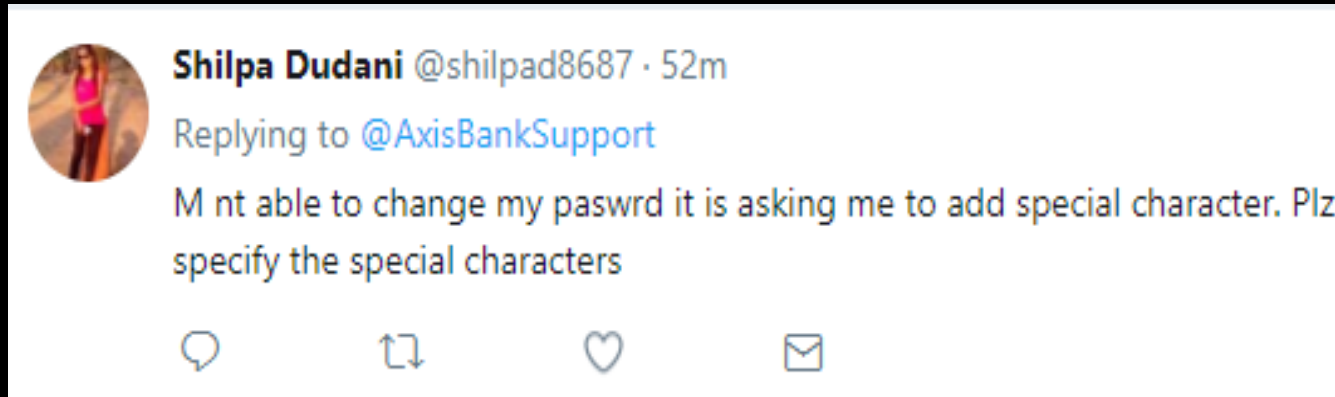
Terminology

- Fully versus Partial Observability
- Deterministic versus Stochastic
- Discrete versus continuous
- Benign versus Adversarial

Observability

- Fully versus Partially Observable
 - If what the agent senses momentarily at any time from the environment is completely sufficient to make the decisions it is fully observable
 - Examples: Chess game, Tennis service etc
- Environment has an internal state
- The agent may be able to fully observe the state or partial

Examples



- The tweets are the text examples of partially observable inputs because they are heavily abbreviated
- The occlusion on the above image is an example of partial observability when we look at images

Deterministic versus Stochastic

- Deterministic: chess moves – outcome is pre determined
- Stochastic: Can a bowler bowl a “perfect” Yorker whenever he intends one?



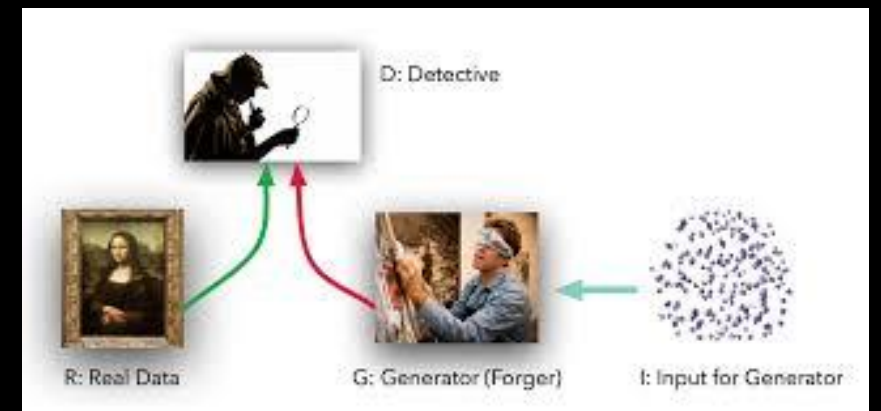
Discrete versus Continuous

- Examples of actions modelled as a continuous variable
 - Throwing a dart : infinitely many ways to angle the dart
 - Turning the steering by an angle (0 to 360 deg, real valued)
 - Magnitude of acceleration to apply
- Actions that are Discrete Variables
 - Choosing a gear (1, 2, 3, 4, Top, Reverse) in a car with manual transmission system
 - Deciding (buy, sell, wait) on stocks of a finite, small set of companies and acting
 - Deciding which elective course to sign up based on the data available and aptitude



Benign versus Adversarial

- A benign system is not attempting to defeat the agent
 - Weather may affect the actions of a self driving car – e.g: reduced visibility and hence increased uncertainty of actions. But it is not an adversary
- An adversarial system attempts to score over the agent. It tries to win over and not allow the agent to succeed.
 - A Generative Adversarial Network (GAN) is modelled as an adversarial game.



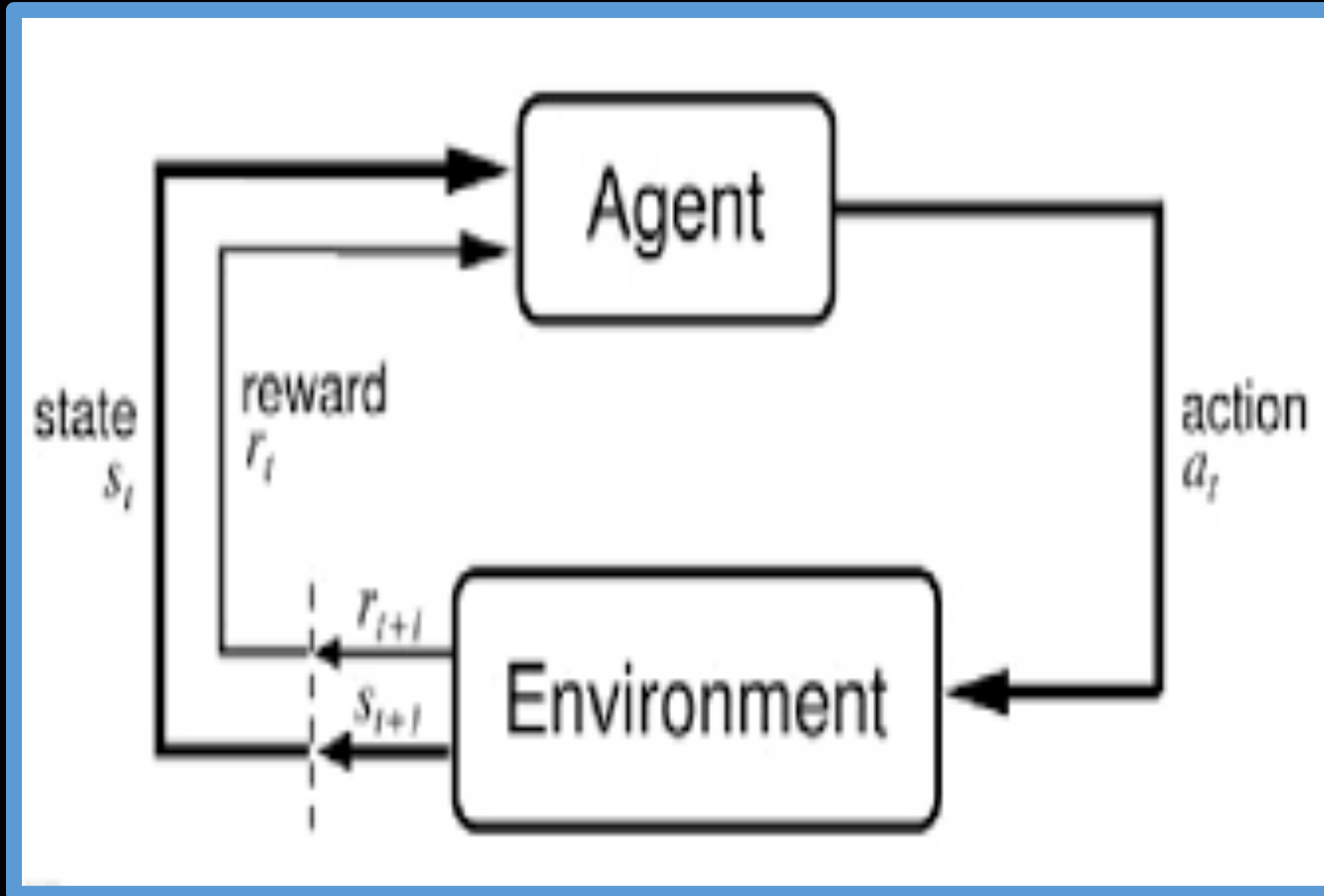
AI as uncertainty management

- What to do when you don't know what to do?
- Reasons for uncertainty
 - Sensor faults, limitations
 - Adversaries : we don't know what it will do
 - Stochastic Environments
 - Ignorance

Casting the problem as a Deep RL problem

- As we observed (in the previous slide), many real world problems can be cast as an agent, environment interaction.
- Reinforcement Learning is a natural choice for a number of applications such as gaming, robotics, dialogue agents etc, where the agent can learn from the feedback from the environment
- As we will see later, a critical component of RL is the policy that agent needs to follow to maximize rewards. A deep neural network can guide the policy. Hence the term Deep Reinforcement Learning.

RL Problem



At each time step t :

Agent:

1. Observes the state from environment
2. Receives a reward from environment
3. Takes an action

Environment:

1. Receives the action
2. Emits the next state ($t+1$)
3. Emits the next reward ($t+1$)

Example

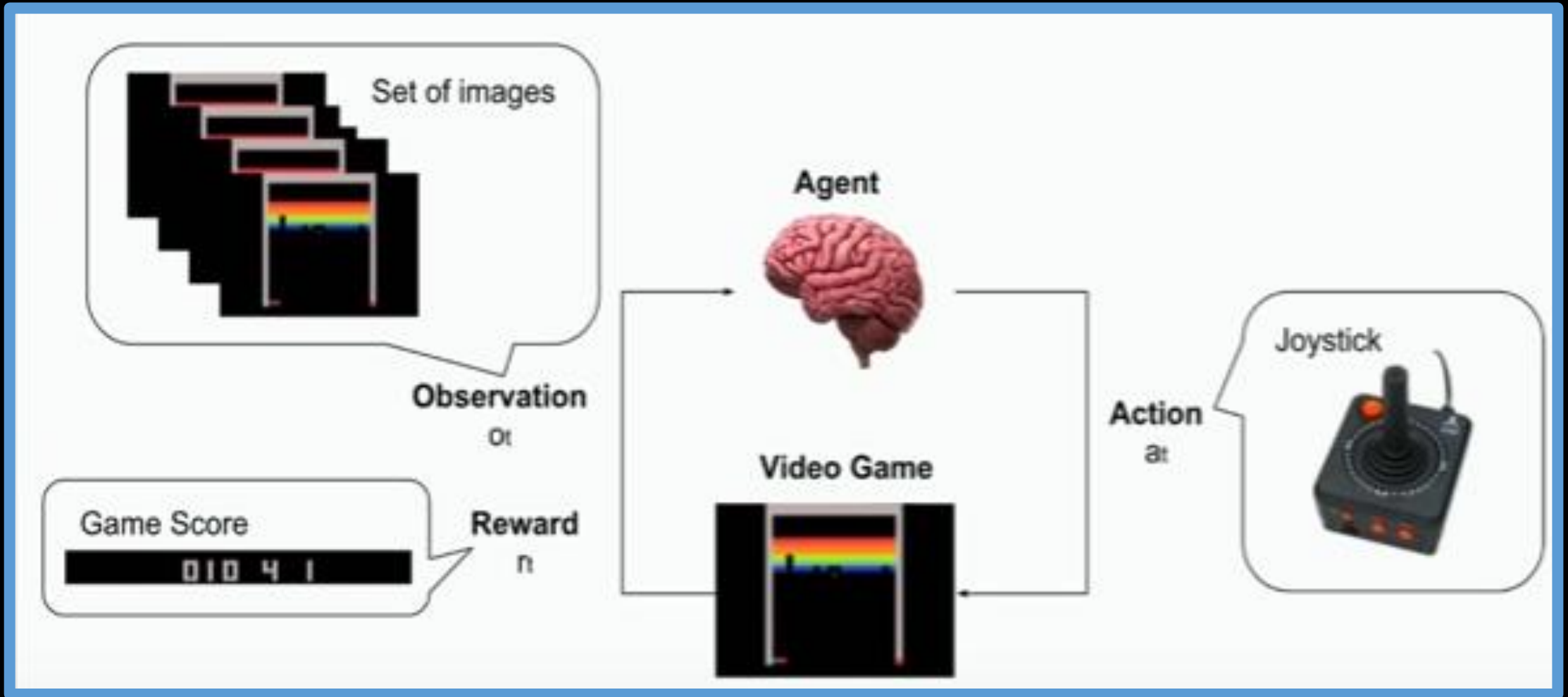


Fig Credits: Center for Brains Minds + Machines (YouTube Video)

Supervised Learning and RL

Supervised Learning:

- Observations are emitted independently of previous agent's actions. Sequence length = 1.
- Immediate feedback (no delayed reward).

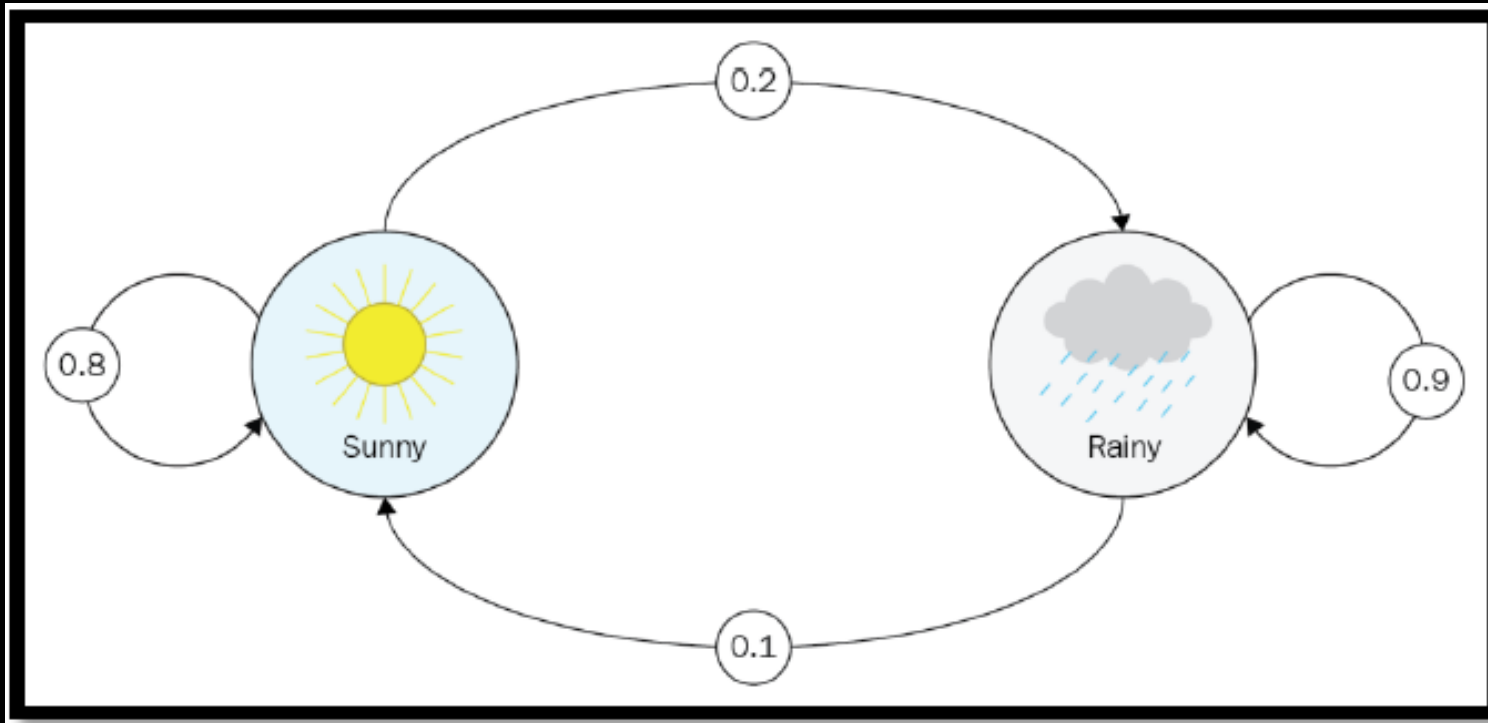


Formulation of Markov Decision Processes

It is helpful for our understanding to view the MDP as an evolution of:

- Markov Process (MP) or Markov Chain
 - All possible states of a system is called “State Space”
 - States are directly observable, sequences of these “observations” form a chain of states
 - Future system dynamics depends only on the current state (Markov Property)
 - Notion of state transition table
- Markov Rewards Process (MRP)
 - All properties same as MP but the notion of rewards included
 - Imagine this to be 2 tables: A state transition table and a rewards table
- Markov Decision Process
 - All the properties same as MRP but we add “actions” to the above
 - Imagine the state transition table as a cube indexed with (i, j, a) where the index “a” is the action

Markov Process Example#1



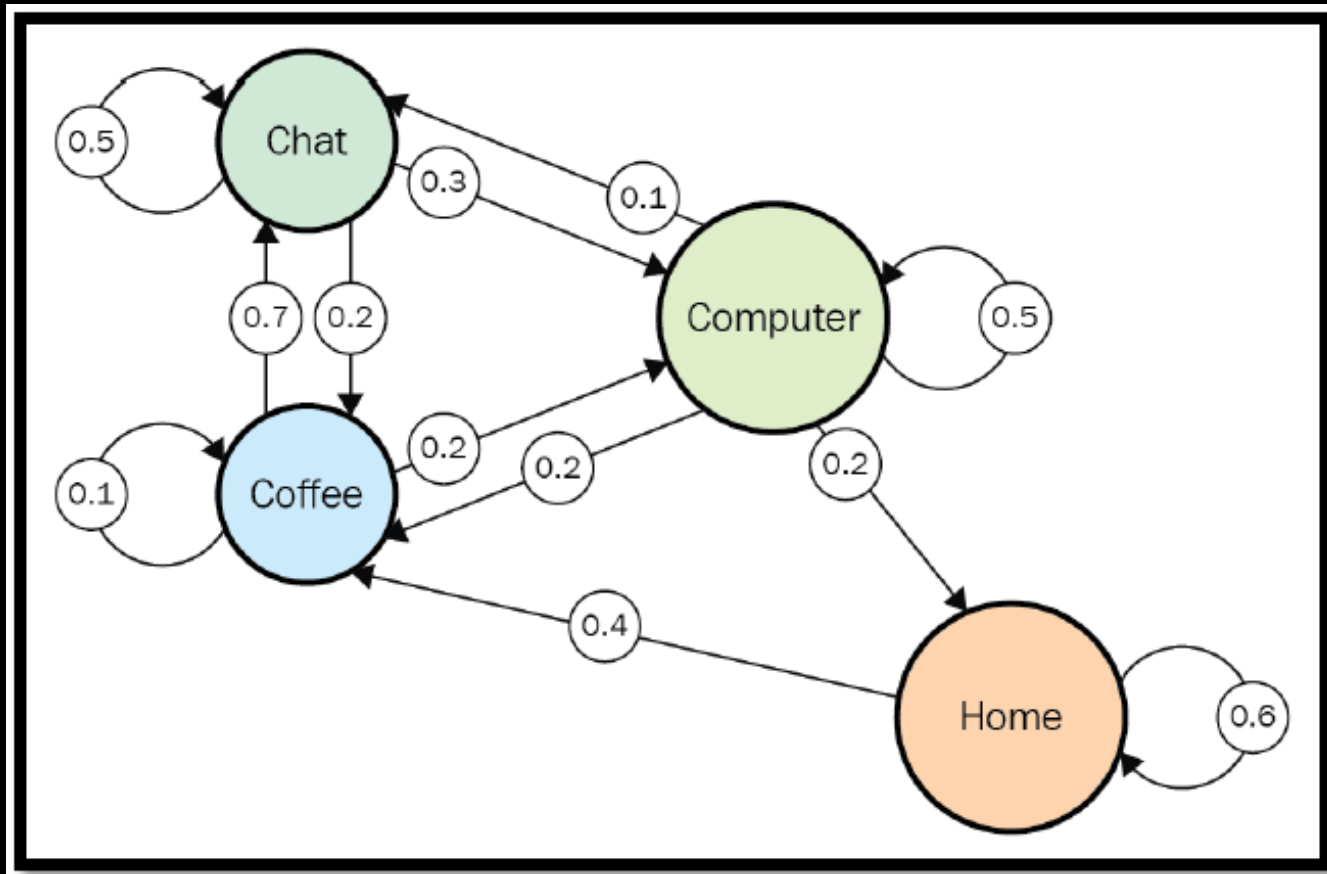
	Sunny	Rainy
Sunny	0.8	0.2
Rainy	0.1	0.9

What is the probability of the sequence:
S, S, S, R, R, S, R

Note: MP states are directly observable.
In a general case, the states and observations are distinct, where states typically model the “cause” and observations “effect”

Fig Credits: Deep Reinforcement Learning Hands On by Maxim Lapan from Packt Publications

Example#2 : Model of Office Worker



	Home	Coffee	Chat	Laptop
Home	0.6	0.4	0	0
Coffee	0	0.1	0.7	0.2
Chat	0	0.2	0.5	0.3
Laptop	0.2	0.2	0.1	0.5

Can you write a typical sequence constituting an episode or history?
What is the minimum sequence length you can form and the max?

Office Worker Example with MRP annotation

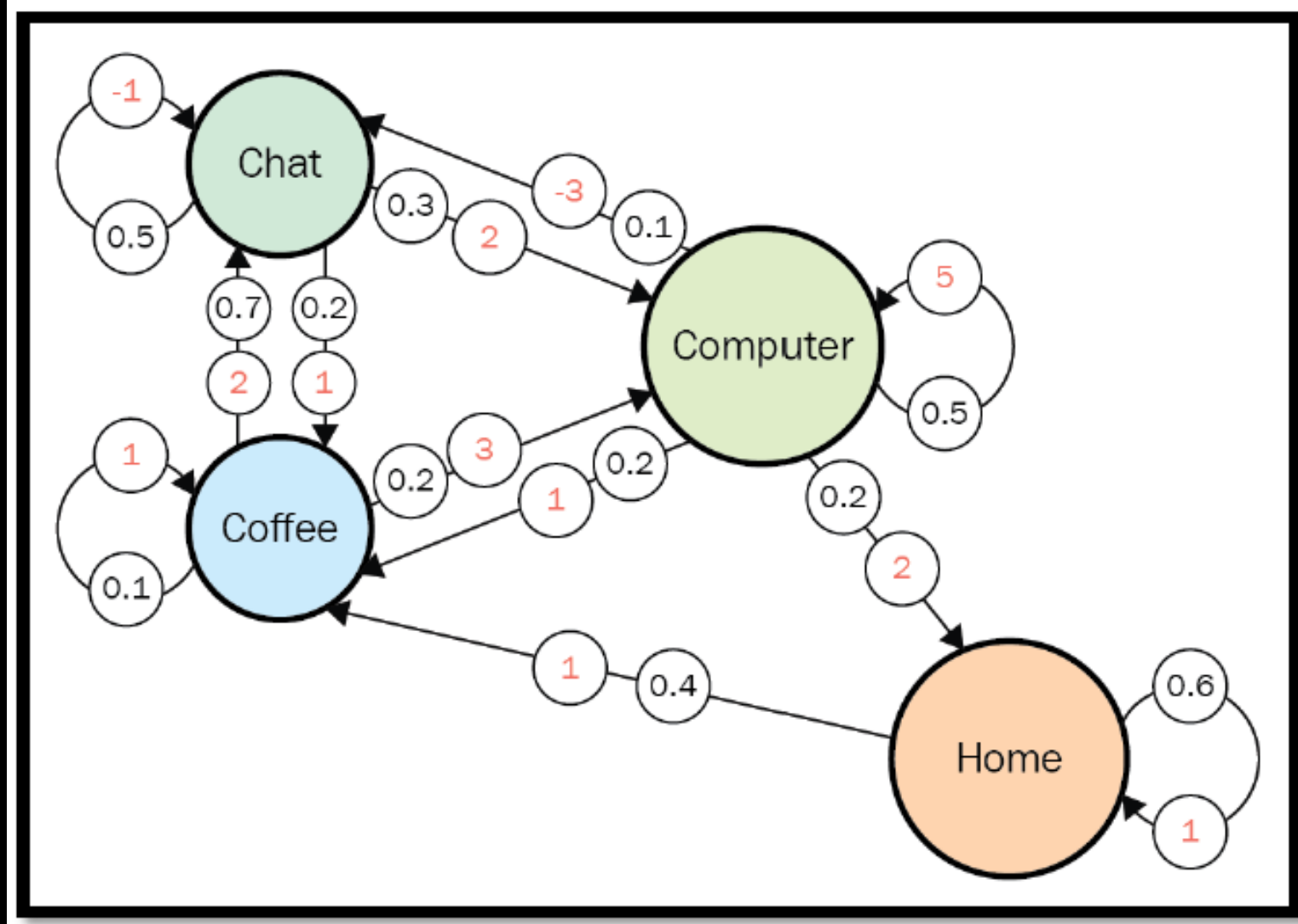


Fig Credits: Deep Reinforcement Learning Hands On by Maxim Lapan from Packt Publications

Adding Actions to the state transition table

- **MP and MRP models don't incorporate mechanism to alter the environment: We can understand the environment but not affect it!**
- **Choosing an action (MDP) allows the agent to affect the probabilities of next state and future dynamics!**

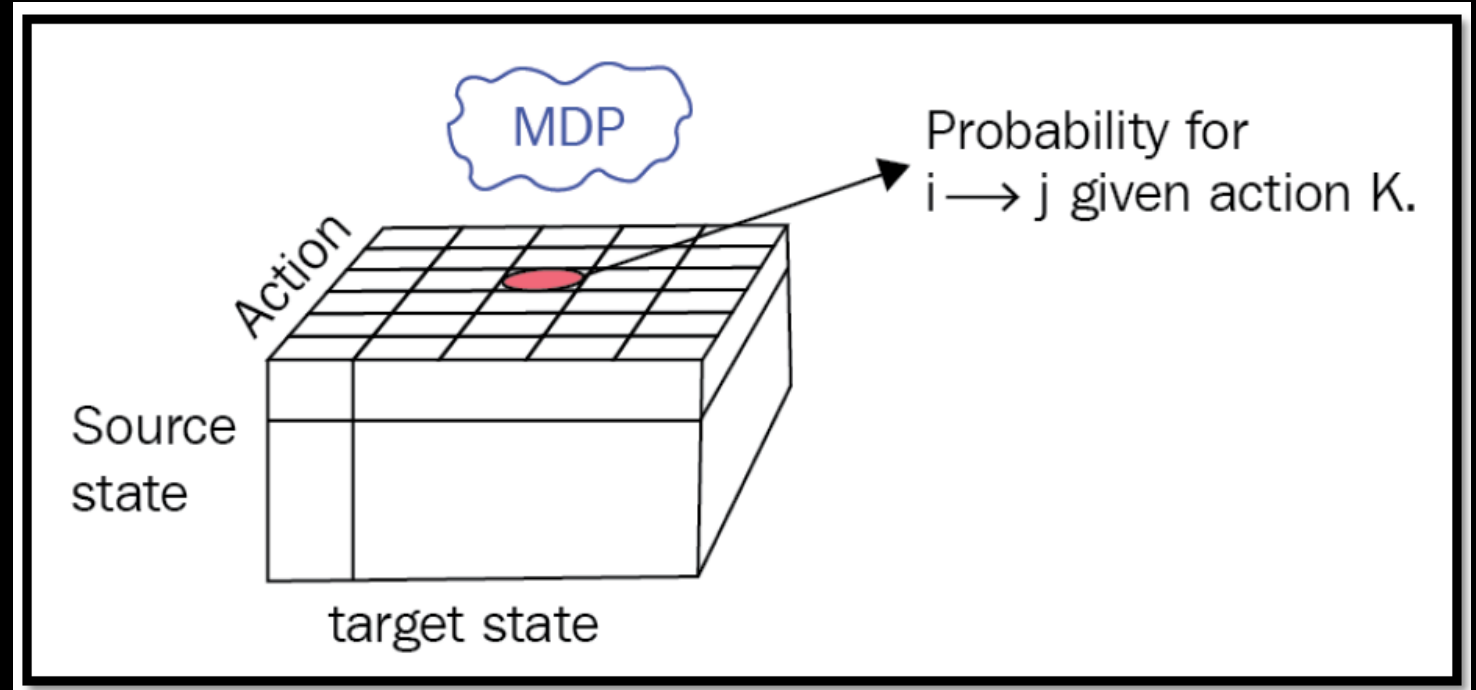
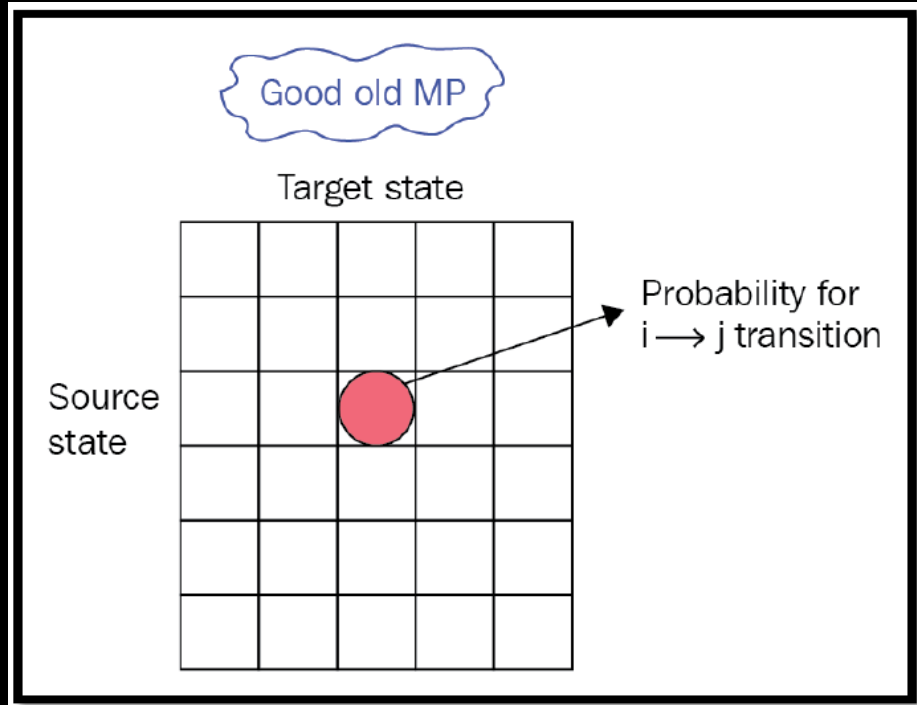


Fig Credits: Deep Reinforcement Learning Hands On by Maxim Lapan from Packt Publications

Mathematical Formulation of RL problem

- We use Markov Decision Process (MDP) to formulate the RL problem
- MDP : (S, A, R, P, γ) where:
 - S : Set of states
 - A : Set of possible actions
 - R : Distribution of reward given (State, Action) pair
 - P : Transition Probability – that is distribution over next state given the current state and action
 - γ : Discount Factor
- MDP satisfies the Markov property: The next state is conditionally independent of all past states given the current state

Markov Decision Process

- At time step $t=0$, environment samples initial state $s_0 \sim p(s_0)$
- Then, for $t=0$ until done:
 - Agent selects action a_t
 - Environment samples reward $r_t \sim R(\cdot | s_t, a_t)$
 - Environment samples next state $s_{t+1} \sim P(\cdot | s_t, a_t)$
 - Agent receives reward r_t and next state s_{t+1}
- A policy π is a function from S to A that specifies what action to take in each state
- **Objective:** find policy π^* that maximizes cumulative discounted reward: $\sum_{t \geq 0} \gamma^t r_t$

States and Trajectories

- An episode (or experience) is a sequence as below:

$$s_0, a_0, r_0, s_1, a_1, r_1, \dots$$

- The sequence may be finite number of steps or can be an infinite sequence
 - E.g. A video game terminates after a finite number of steps while a robot that performs surveillance may be interacting with the environment continually. A single game can be thought of as one episode
- Sometimes the rewards may be delayed and not instantaneous
 - E.g. A move in a chess game may or may not be indicative of a win or loss. Result is known in the end. Similarly the final result of a game for an action by a batsman for a given ball may or may not be known instantly

Project Idea (Suggestion)

- Build a simulator for simulating a cricket game – this should be the proxy for the environment
- You should use Deep Learning to build the simulator itself
- Build an agent that uses Deep RL in order to take actions
- Visualize with graphics

Demo

- <https://youtu.be/V1eYniJ0Rnk>

Agent Policy

- The goal of the agent is to win the game
- How can it strategize in order to win?
- One option of an agent is a deep neural network
- How can a neural network learn to play a video game similar to us?

Policy

- Policy is a way of choosing an action in accordance with the current situation (state)
- The policy is a function map from the state to the action
- Application of a policy can result in an action that can be emitted deterministically or stochastically
 - Choose what action to take (Policy)
 - The action may be emitted as per a probability distribution

$$a_t = \pi(s_t) \quad (\text{deterministic})$$
$$\pi(a_t|s_t) = P(a_t|s_t) \quad (\text{stochastic})$$

Value Function

- Value function indicates the value of a given state, under a given policy
- This means that: “How much reward will I get if I am in the state s_t ?” That is, how good is this state?
- This can only be defined as an expectation value, as the future has not happened yet and so we don't know what reward we will get in the end
- Example: In a 50 over game, what is the expected run rate after the innings if the batting side is 300 for 3 after 40th over?

$$V^\pi(s_t) = \mathbb{E}_\pi[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots | s_t]$$

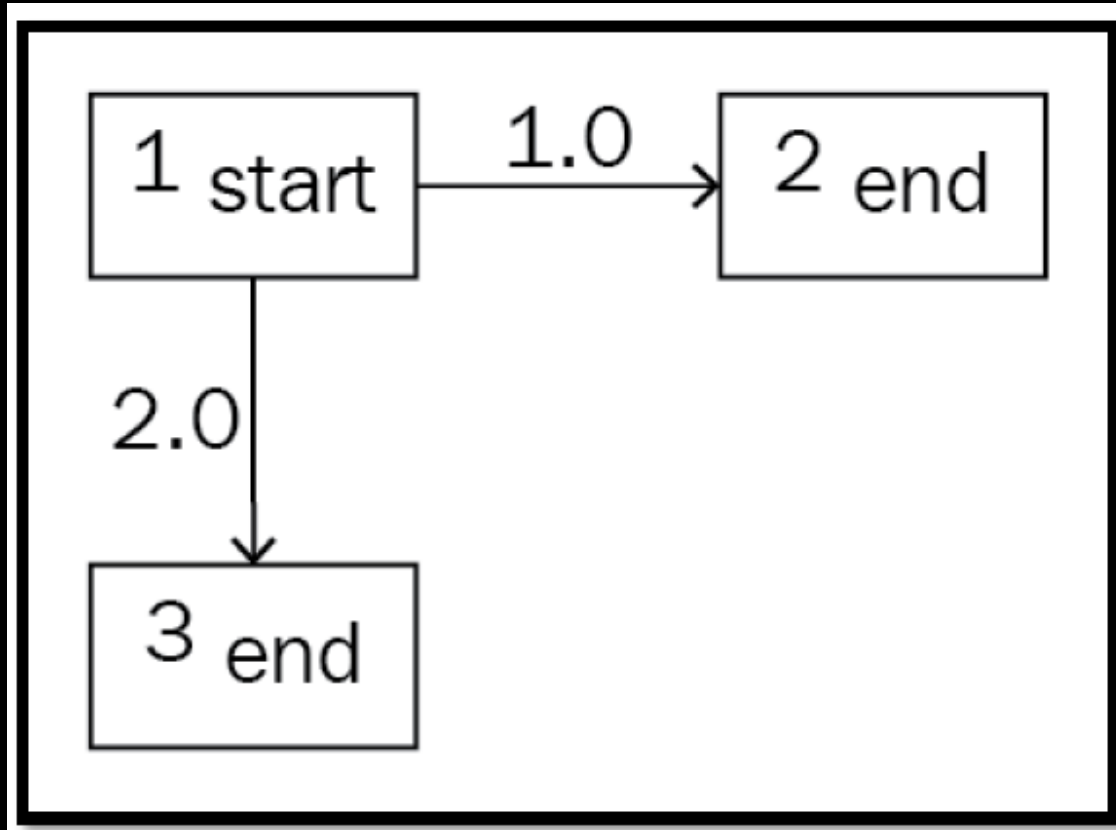
$$0 \leq \gamma < 1$$

(discounted framework)

$$V^\pi(s) = \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, \pi \right]$$

Figure Credits: Stanford Fei Fei Li et al CS231n

Example



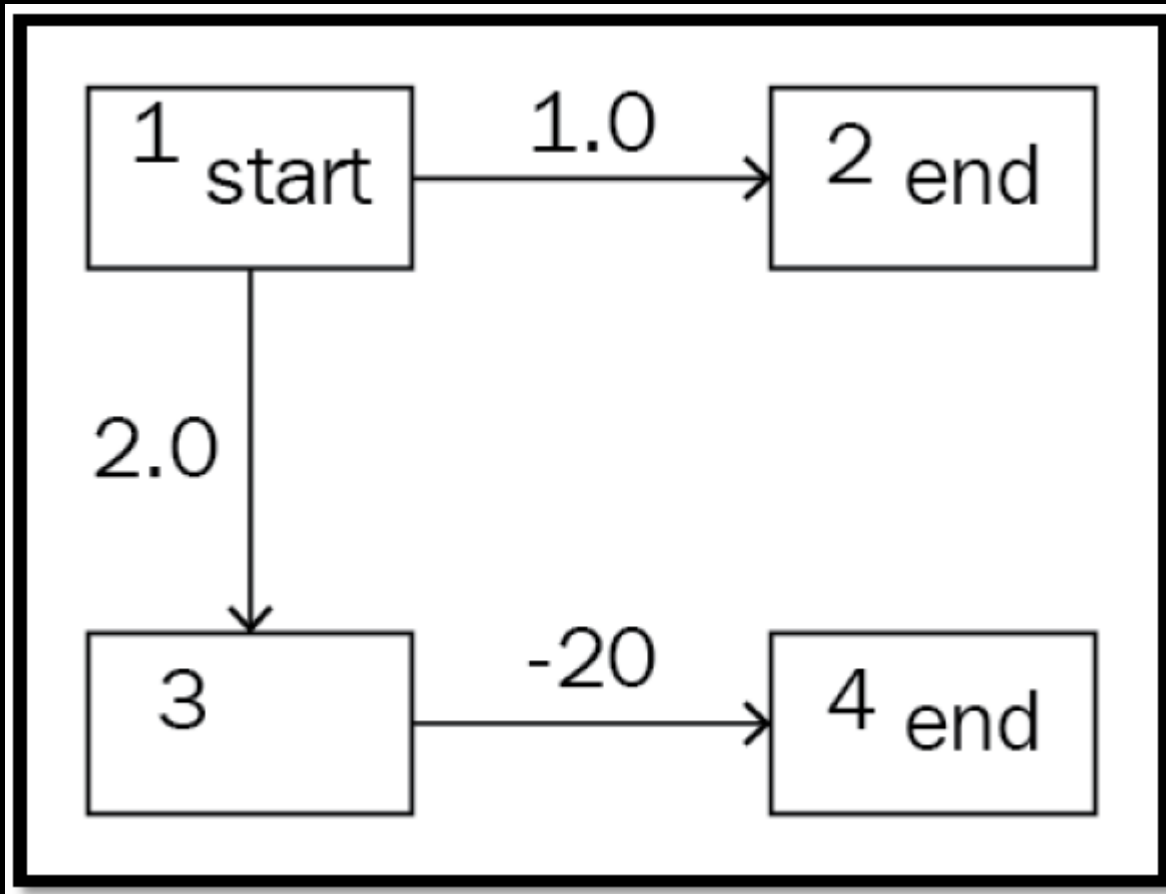
- Consider an environment with 3 states as below:
- The agents initial state is state 1
- The final state after executing an action “right” is state 2 and has a reward 1.0
- The final state after executing “down” is state 3 and has a reward 2.0

Example

- Assume: Environment is always deterministic, start state is always 1, system terminates on reaching the final state
- How to compute the value for state 1?
- The above depends on the policy
- The space of policies is potentially infinite, some possible policies:
 - Always go right
 - Always go down
 - Go right or down with uniform distribution of probabilities (0.5, 0.5 in this example)
 - Go right 10% of the time, down 90%

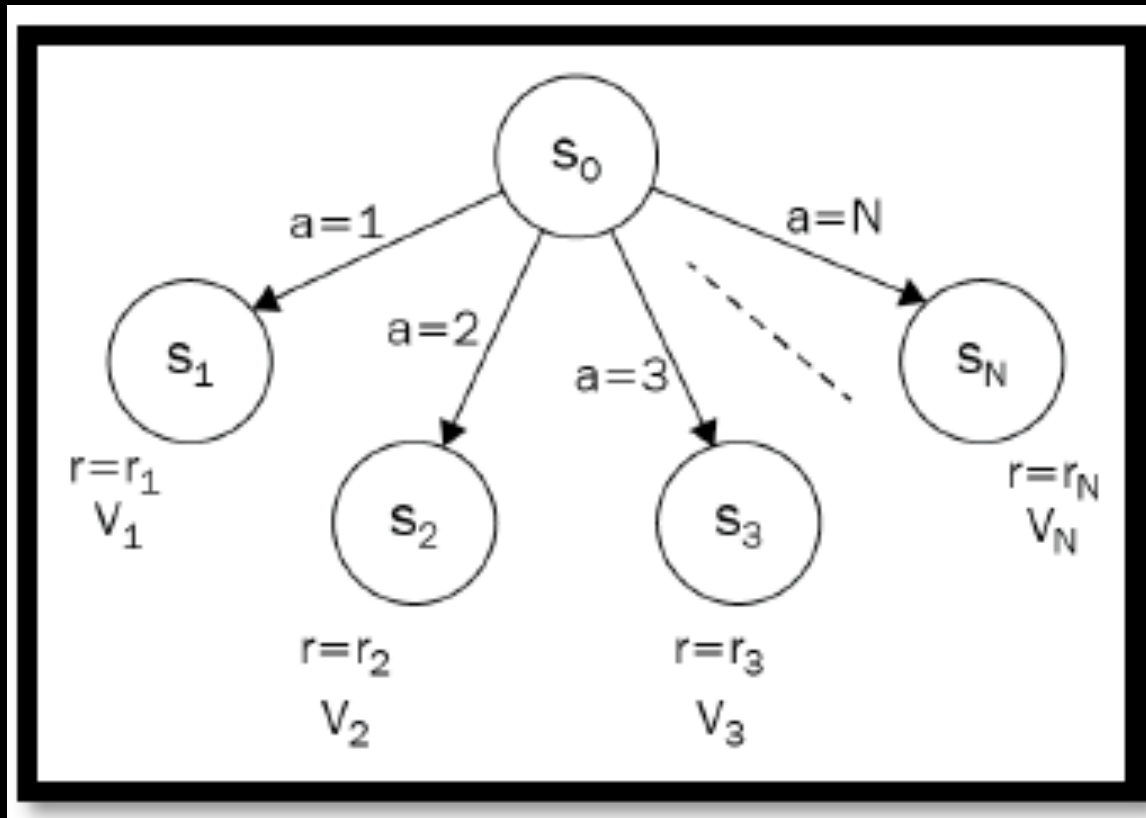
Compute the value of State 1 under each of the policies above. For the above case, what is the optimal policy to choose from?

Example (Contd)



- Compute the value of State 1 under different policies for the modified example and determine the optimal policy

States reachable from initial states (deterministic)

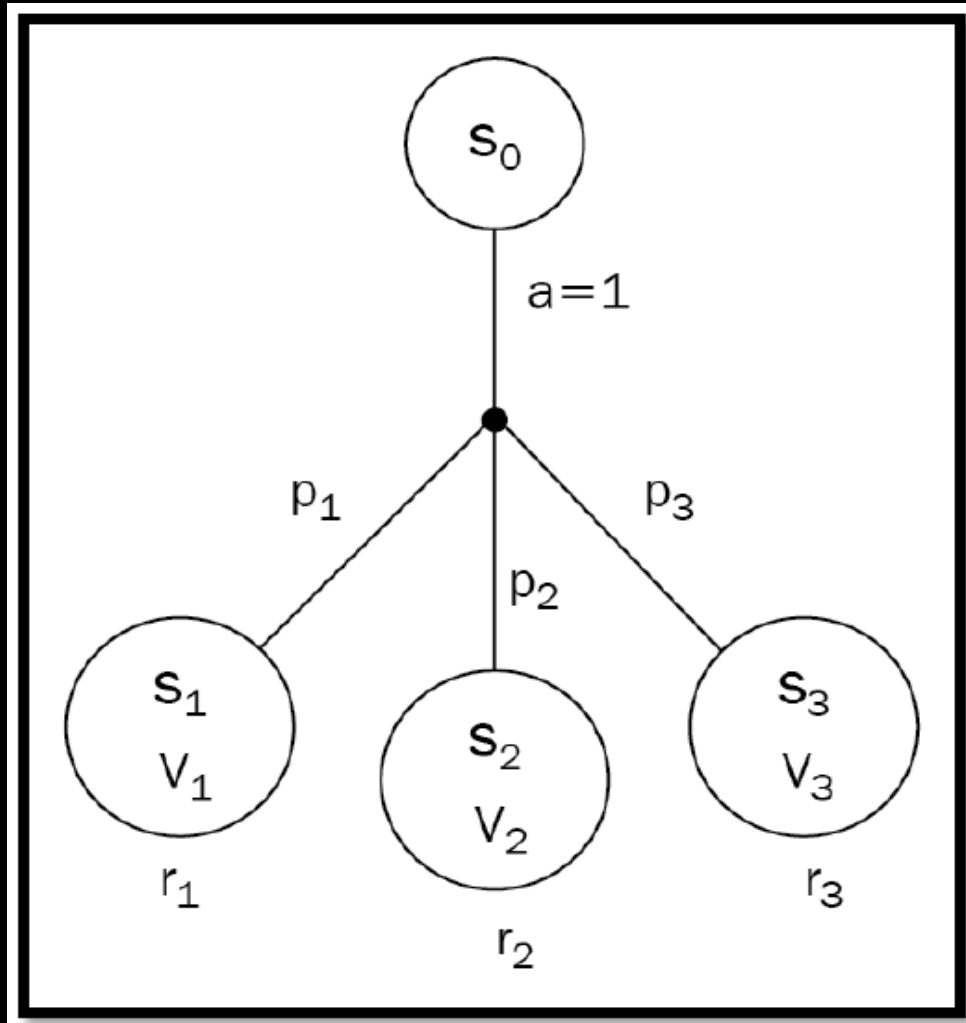


- The figure depicts an environment where:
1. There are N actions that lead us deterministically to N states
 2. Respective rewards are r_1 to r_n
 3. If we choose a concrete action a_i and calculate the value for this action at start state, what is the value for $V_0(a=a_i)$?

$$V_0 = \max_{a \in 1..N} (r_a + \gamma V_a)$$

The above is the Bellman equation of value (deterministic case)

States reachable from initial states Stochastic)



- Suppose we choose action $a = 1$ and there are 3 possible states.

$$V_0(a = 1) = p_1(r_1 + \gamma V_1) + p_2(r_2 + \gamma V_2) + p_3(r_3 + \gamma V_3)$$

$$V_0(a) = E_{s \sim S}[(r_{s,a} + \gamma V_s)] = \sum_{s \in S} p_{a,0 \rightarrow s} (r_{s,a} + \gamma V_s)$$

Bellman Optimality Equation for General Case:

$$V_0 = \max_{a \in A} E_{s \sim S}[(r_{s,a} + \gamma V_s)]$$

$$= \max_{a \in A} \sum_{s \in S} p_{a,0 \rightarrow s} (r_{s,a} + \gamma V_s)$$

Q-Function

- Q Function indicates how good is a state-action pair
- That is, from a given state, take an action to reach the next state. From then on continue to follow the policy. Compute the total cumulative rewards starting from the current state. The difference between the value function and the Q function is that the value function considers any allowable action from current state and follows the policy. The Q function fixes a particular action in the current state and follows the policy from the next state.

$$Q^{\pi}(s, a) = \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t \mid s_0 = s, a_0 = a, \pi \right]$$

Value Function

$$V^{\pi}(s_{t+1})$$

$$V^{\pi}(s_t) = \mathbb{E}_{\pi}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots | s_t]$$

Markov assumption:

$$P(s_{t+1} | s_t, \pi(s_t)) \quad \text{Transition probability to state } s_{t+1} \text{ from } s_t \text{ after action } \pi(s_t)$$

$$V^{\pi}(s_t) = \mathbb{E}_{a_t \sim \pi}[r_t + \gamma \sum_{s_{t+1}} P(s_{t+1} | s_t, a_t) V^{\pi}(s_{t+1})]$$

Source: Center for Brains Minds + Machines (YouTube Video)

Q-function

$$V^{\pi}(s_t) = \mathbb{E}_{a_t \sim \pi} [r_t + \gamma \sum_{s_{t+1}} P(s_{t+1} | s_t, a_t) V^{\pi}(s_{t+1})]$$

$$Q^{\pi}(s_t, a_t) = r_t + \gamma \sum_{s_{t+1}} P(s_{t+1} | s_t, a_t) V^{\pi}(s_{t+1})$$

Source: Center for Brains Minds + Machines (YouTube Video)

Bellman equation

The optimal Q-value function Q^* is the maximum expected cumulative reward achievable from a given (state, action) pair:

$$Q^*(s, a) = \max_{\pi} \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t \mid s_0 = s, a_0 = a, \pi \right]$$

Q^* satisfies the following **Bellman equation**:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s', a') \mid s, a \right]$$

Intuition: if the optimal state-action values for the next time-step $Q^*(s', a')$ are known, then the optimal strategy is to take the action that maximizes the expected value of $r + \gamma Q^*(s', a')$

The optimal policy π^* corresponds to taking the best action in any state as specified by Q^*

Figure Credits: Stanford Fei Fei Li et al CS231n

Optimal Policy

- Suppose we have a space of a number of policies to choose from
- The optimal policy is one that maximizes the expected rewards
- Formally:

$$\pi^* = \arg \max_{\pi} \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | \pi \right] \text{ with } s_0 \sim p(s_0), a_t \sim \pi(\cdot | s_t), s_{t+1} \sim p(\cdot | s_t, a_t)$$

Variants of RL

- Value Based RL
 - Estimate the expected returns to choose the policy
- Policy Based RL
 - Directly optimize the policy to get the return
- Model Based RL
 - Build a model of the environment and choose action based on the model

Example

- Value based RL:
 - Suppose we are chasing a target of 300 runs, we can model this as different states in terms of each run. Thus we have 300 states. The state can also include the wickets in hand, number of overs remaining etc.
 - Compute the value of each state and policy can choose the optimal actions
 - Note that the state space is huge and can potentially be infinite
- Policy based RL
 - Suppose we have a space of a large number of policies
 - We can directly optimize for the best policy that maximizes sum of rewards

Solving for the optimal policy

Value iteration algorithm: Use Bellman equation as an iterative update

$$Q_{i+1}(s, a) = \mathbb{E} \left[r + \gamma \max_{a'} Q_i(s', a') | s, a \right]$$

Q_i will converge to Q^* as $i \rightarrow \infty$

What's the problem with this?

Not scalable. Must compute $Q(s, a)$ for every state-action pair. If state is e.g. current game state pixels, computationally infeasible to compute for entire state space!

Solution: use a function approximator to estimate $Q(s, a)$. E.g. a neural network!

Q-learning: Use a function approximator to estimate the action-value function

$$Q(s, a; \theta) \approx Q^*(s, a)$$

 function parameters (weights)

If the function approximator is a deep neural network => **deep q-learning!**

Solving for the optimal policy: Q-learning

Remember: want to find a Q-function that satisfies the Bellman Equation:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s', a') | s, a \right]$$

Forward Pass

Loss function: $L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot)} \left[(y_i - Q(s, a; \theta_i))^2 \right]$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a \right]$

Iteratively try to make the Q-value close to the target value (y_i) it should have, if Q-function corresponds to optimal Q^* (and optimal policy π^*)

Backward Pass

Gradient update (with respect to Q-function parameters θ):

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot); s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \right] \nabla_{\theta_i} Q(s, a; \theta_i)$$

[Mnih et al. NIPS Workshop 2013; Nature 2015]

Case Study: Playing Atari Games



Objective: Complete the game with the highest score

State: Raw pixel inputs of the game state

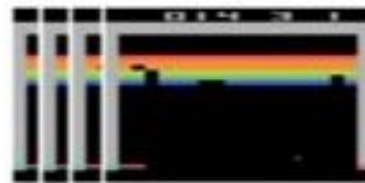
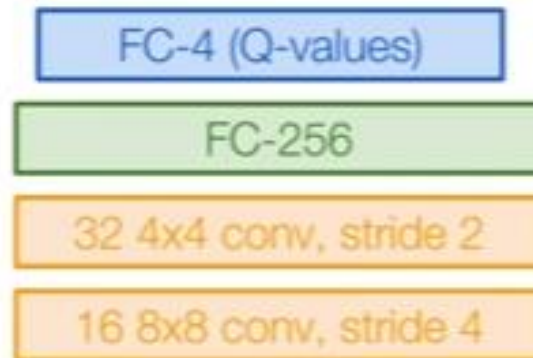
Action: Game controls e.g. Left, Right, Up, Down

Reward: Score increase/decrease at each time step

Q-network Architecture

$Q(s, a; \theta)$:
neural network
with weights θ

A single feedforward pass
to compute Q-values for all
actions from the current
state => efficient!



Current state s_t : 84x84x4 stack of last 4 frames
(after RGB->grayscale conversion, downsampling, and cropping)

← Last FC layer has 4-d
output (if 4 actions),
corresponding to $Q(s_t, a_1)$, $Q(s_t, a_2)$, $Q(s_t, a_3)$,
 $Q(s_t, a_4)$

Number of actions between 4-18
depending on Atari game

Training the Q-network: Experience Replay

Learning from batches of consecutive samples is problematic:

- Samples are correlated => inefficient learning
- Current Q-network parameters determines next training samples (e.g. if maximizing action is to move left, training samples will be dominated by samples from left-hand size) => can lead to bad feedback loops

Address these problems using **experience replay**

- Continually update a **replay memory** table of transitions (s_t, a_t, r_t, s_{t+1}) as game (experience) episodes are played
- Train Q-network on random minibatches of transitions from the replay memory, instead of consecutive samples

Each transition can also contribute
to multiple weight updates
=> greater data efficiency

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N

Initialize action-value function Q with random weights

for episode = 1, M **do**

 Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$

for $t = 1, T$ **do**

 With probability ϵ select a random action a_t

 otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

 Execute action a_t in emulator and observe reward r_t and image x_{t+1}

 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}

 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}

 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3

end for

end for


← Initialize state
(starting game
screen pixels) at the
beginning of each
episode

Finance Domain Case Study: Stock Trading

- Key problem involved in Financial Markets: “Can we predict future price movements?”
 - Accurate prediction entails huge rewards as it answers the questions of where and when to invest in.
 - This is the full time business of finance consultants, managers of investment funds and so on
- Can we cast this as an RL problem and find “optimal actions” that maximize profits?
 - We have an observation of the market and we want to decide between the actions: (buy, sell, wait)
 - If we buy before price goes up we make profit, otherwise we get a negative reward
 - Our goal is to maximize the profits as much as possible

Casting the problem : Fund Manager Agent

- To cast a real world problem as an RL problem, we need to define 3 things:
 - Space of Observations, Space of Actions to take, Reward System
- Actions are: (buy, wait, sell)
- Rewards can be expressed in different ways. We may assume a reward only on selling or we can get a reward at every time step we own the share.
- We can represent the price as relative movement (e.g increase by 5%)



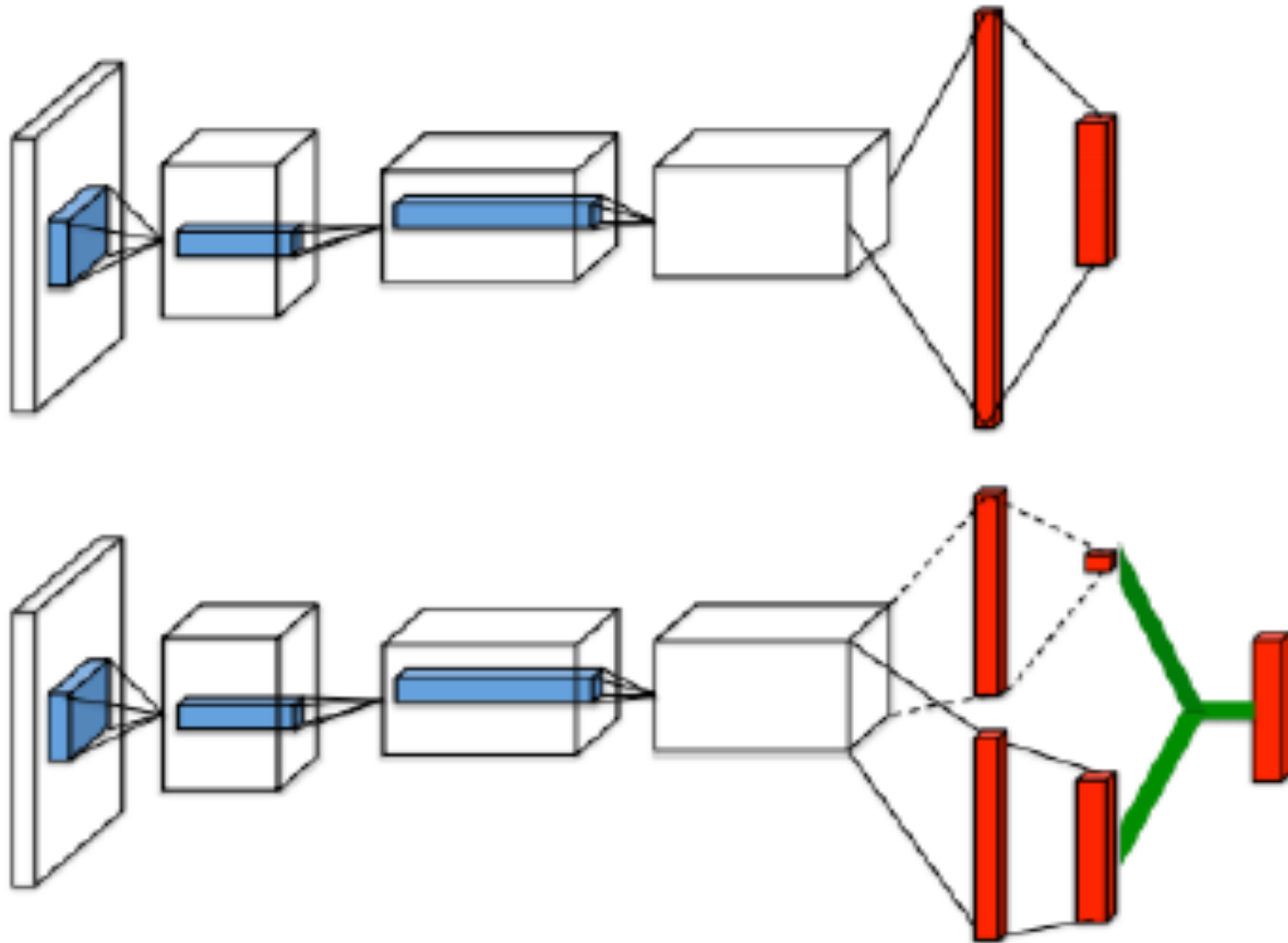
Indiabulls Integrated Services Limited

date	open	high	low	last	close
2018-09-04	670	695.9	670	676	677.5
2018-09-03	655	690	644.15	662.95	662.8
2018-08-31	689	704	660.5	668	678.05

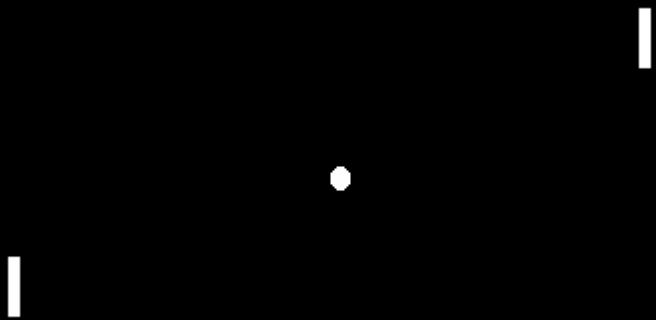
Duelling DQN

- Q Values $Q(s, a)$ that the DQN is trying to estimate can be thought of as a sum of 2 terms: $V(s) + A(s, a)$
- $V(s)$ as we know is the discounted expected reward achievable from the state s
- The Advantage term $A(s, a)$ is meant to bridge the gap between $V(s)$ to $Q(s, a)$. The advantage term is just a delta that says how much extra reward a given action brings us.
- $A(s, a)$ is a scalar value that can be positive or negative

DQN and Duel DQN Architecture

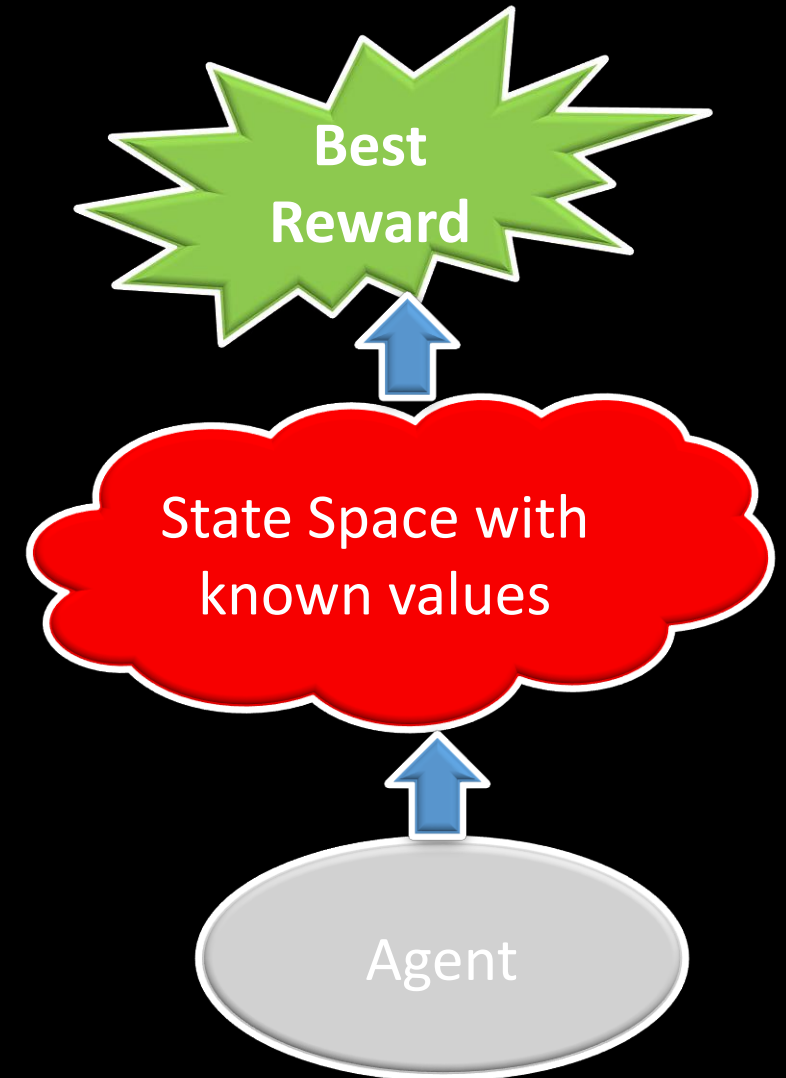


Pong Game using PG (Ref A Karpathy's blog)



Motivation for Policy Gradient

- Q Learning discussed till now is centred around computing “value” for each state or state-action pair but what we actually need is the Policy that tells us the right action to take.
- Assuming we know all the states and their values, the optimal policy is the one that obtains maximum discounted future rewards
- Thus, values govern how we should navigate through the state space
- Values of states stand between the agent and the best reward
- The policy is indirectly got through:
$$\pi(s) = \operatorname{argmax}_a Q(s, a)$$



Motivation for Policy Gradient (Contd)

- The cross product of state space and the space of actions can be huge
- We approximated the Q value of a state using a function approximator (DQN)
- Thus, our focus had been to train the DQN to predict the Q values accurately and also choose the other key parameters like gamma, learning rate optimally
- We used the predicted Q Values to infer the policy, instead of directly computing it
- Why can't we build a model that directly generates the policy as opposed to just predicting Q Values?

References for Policy Gradients

- For a good theoretical treatment see Stanford CS231n_2017 lectures. Credits: Some of the slides to follow contain figures taken from this course material
- For a more practical understanding, please go through: Deep Reinforcement Learning Hands On by Maxim Lapan, Packt Publications. Credits: A good number of figures are used from this book.
- Andrej Karpathy's blog on Reinforcement Learning (Pong from Pixels)

Theoretical Foundations (Stanford CS231n)

Policy Gradients

Formally, let's define a class of parametrized policies: $\Pi = \{\pi_\theta, \theta \in \mathbb{R}^m\}$

For each policy, define its value:

$$J(\theta) = \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | \pi_\theta \right]$$

We want to find the optimal policy $\theta^* = \arg \max_{\theta} J(\theta)$

How can we do this?

REINFORCE Algorithm : slide 1 of 4

REINFORCE algorithm

Mathematically, we can write:

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\tau \sim p(\tau; \theta)} [r(\tau)] \\ &= \int_{\tau} r(\tau) p(\tau; \theta) d\tau \end{aligned}$$

Where $r(\tau)$ is the reward of a trajectory | $\tau = (s_0, a_0, r_0, s_1, \dots)$

REINFORCE Algorithm : slide 2 of 4

REINFORCE algorithm

Expected reward: $J(\theta) = \mathbb{E}_{\tau \sim p(\tau; \theta)} [r(\tau)]$

$$= \int_{\tau} r(\tau) p(\tau; \theta) d\tau$$

Now let's differentiate this: $\nabla_{\theta} J(\theta) = \int_{\tau} r(\tau) \nabla_{\theta} p(\tau; \theta) d\tau$

Intractable! Gradient of an expectation is problematic when p depends on θ

However, we can use a nice trick: $\nabla_{\theta} p(\tau; \theta) = p(\tau; \theta) \frac{\nabla_{\theta} p(\tau; \theta)}{p(\tau; \theta)} = p(\tau; \theta) \nabla_{\theta} \log p(\tau; \theta)$

If we inject this back:

$$\begin{aligned} \nabla_{\theta} J(\theta) &= \int_{\tau} (r(\tau) \nabla_{\theta} \log p(\tau; \theta)) p(\tau; \theta) d\tau \\ &= \mathbb{E}_{\tau \sim p(\tau; \theta)} [r(\tau) \nabla_{\theta} \log p(\tau; \theta)] \end{aligned}$$

Can estimate with
Monte Carlo sampling

REINFORCE Algorithm : slide 3 of 4

REINFORCE algorithm

$$\begin{aligned}\nabla_{\theta} J(\theta) &= \int_{\tau} (r(\tau) \nabla_{\theta} \log p(\tau; \theta)) p(\tau; \theta) d\tau \\ &= \mathbb{E}_{\tau \sim p(\tau; \theta)} [r(\tau) \nabla_{\theta} \log p(\tau; \theta)]\end{aligned}$$

Can we compute those quantities without knowing the transition probabilities?

We have: $p(\tau; \theta) = \prod_{t \geq 0} p(s_{t+1} | s_t, a_t) \pi_{\theta}(a_t | s_t)$

Thus: $\log p(\tau; \theta) = \sum_{t \geq 0} \log p(s_{t+1} | s_t, a_t) + \log \pi_{\theta}(a_t | s_t)$

And when differentiating: $\nabla_{\theta} \log p(\tau; \theta) = \sum_{t \geq 0} \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$

Doesn't depend on
transition probabilities!

Therefore when sampling a trajectory τ , we can estimate $J(\theta)$ with

$$\nabla_{\theta} J(\theta) \approx \sum_{t \geq 0} r(\tau) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$$

REINFORCE Algorithm : slide 4 of 4

Intuition

Gradient estimator:
$$\nabla_{\theta} J(\theta) \approx \sum_{t \geq 0} r(\tau) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$$

Interpretation:

- If $r(\tau)$ is high, push up the probabilities of the actions seen
- If $r(\tau)$ is low, push down the probabilities of the actions seen

Might seem simplistic to say that if a trajectory is good then all its actions were good. **But in expectation, it averages out!**

However, this also suffers from high variance because credit assignment is really hard. Can we help the estimator?

Hands On REINFORCE algorithm

- Reference: Deep Reinforcement Learning Hands On by Maxim Lapan, Packt Publications. Credits: Figures in the next slide and the figure for Action Critic architecture are from this book.

Policy Gradient Architecture

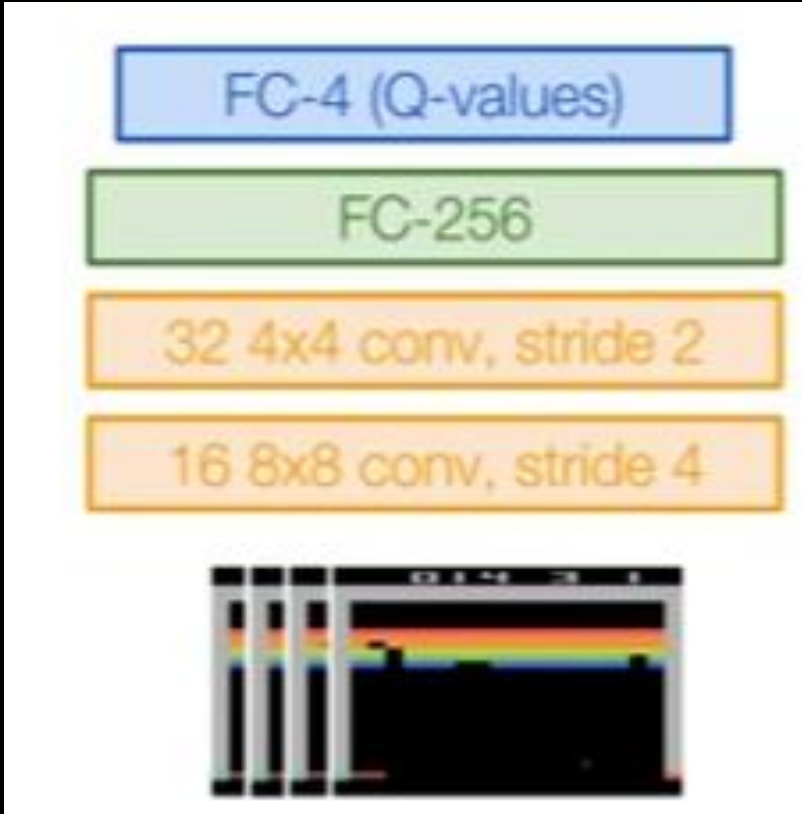


Fig (a)

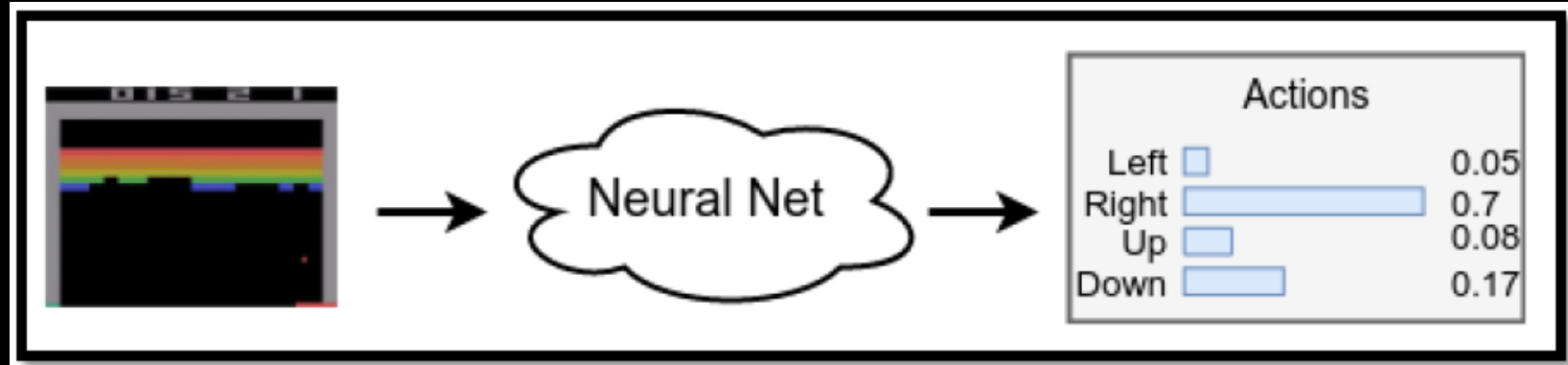


Fig (b)

- The figures (a) and (b) illustrate the DQN and PGN respectively
- PGN generates a probability distribution over actions, given the observation and the parameters of the model

$$P(a = a_i | s = s_j; \theta) = PGN(s_j)$$

Why Policy Gradient Approach?

- While we do care about total reward, the exact value of a state is less important compared to the action we need to take from a given state. Q Learning addresses our need indirectly by computing the value and it is left to us to determine the action from the predicted value.
 - E.g. For the Cartpole assume we got Q Values of 20, 40 for left and right moves. We may take the action of right because it's Q value is higher. The action = right is the output we are interested in as opposed to $Q_{\text{left}} = 20$, $Q_{\text{right}} = 40$. Since the Q Values are not probability distributions, there is an arbitrariness to the way we select the action.
- When the environment supports a large number of actions or in the extreme a continuous action space, Q Learning approach may be less feasible
- PG method allows us to sample actions with some stochasticity as per the distribution it produces.

Policy Gradients and Loss Function

We need to maximize:

$$J(\theta) = E[Q(s, a) \log(\pi(a|s; \theta))]$$

$$\nabla J(\theta) = E[Q(s, a) \nabla \log(\pi(a|s; \theta))]$$

We can write this as a problem of minimizing the loss function:

$$L(\theta) = -Q(s, a) \log(\pi(a|s; \theta))$$

The loss function can be minimized using SGD

REINFORCE Algorithm

1. Initialize the network with random weights
2. Play full episodes, saving (s, a, r, s')
3. For every step t of every episode k , calculate the discounted total reward for subsequent steps: $Q_{k,t} = \sum_{i=0} \gamma^i r_i$
4. Calculate the loss function for all transitions: $\mathcal{L} = -\sum_{k,t} Q_{k,t} \log(\pi(s_{k,t}, a_{k,t}))$
5. Perform SGD update of weights for minimizing loss
6. Repeat from step 2 until convergence

DQN learning and REINFORCE Differences

- No explicit exploration needed
 - In Q Learning we had a epsilon-greedy approach to select the action in order to explore the environment and prevent our agent from getting stuck (Exploration versus exploitation)
 - With PGN returning probabilities, we now can sample the actions from this distribution
- No replay buffer is used
 - PG methods require full episodes obtained from on line interaction with environment
- No target network is needed
 - Here, though we use Q Values, they are obtained from our experience in the environment for the full episode

REINFORCE issues

- Full episodes are required
 - The reason why this is needed is to be able to estimate the Q values
 - In DQN approach, we have a neural network to predict $V(s)$ for future while in PG method, we compute this from the complete episode trajectory
- High Gradients Variance
 - $\nabla J(\theta) = E[\nabla Q(s, a) \nabla \log(\pi(a|s; \theta))]$: Gradient is proportional to the discounted reward Q
 - Range of this reward depends on the environment
 - Large differences impact training dynamics
 - Plain vanilla REINFORCE: Use a baseline that we subtract from Q value
 - The baseline may be the moving average of discounted rewards, better still, it can be $V(s)$
- Exploration
 - In DQN we used epsilon-greedy approach
 - One may also use Entropy Bonus: $H(\pi) = -\sum \pi(a|s) \log \pi(a|s)$
 - Subtract entropy from the loss function in order to avoid the agent getting stuck

Variance

Actor Critic Method

