Reinforcement Learning

Indranil Basu

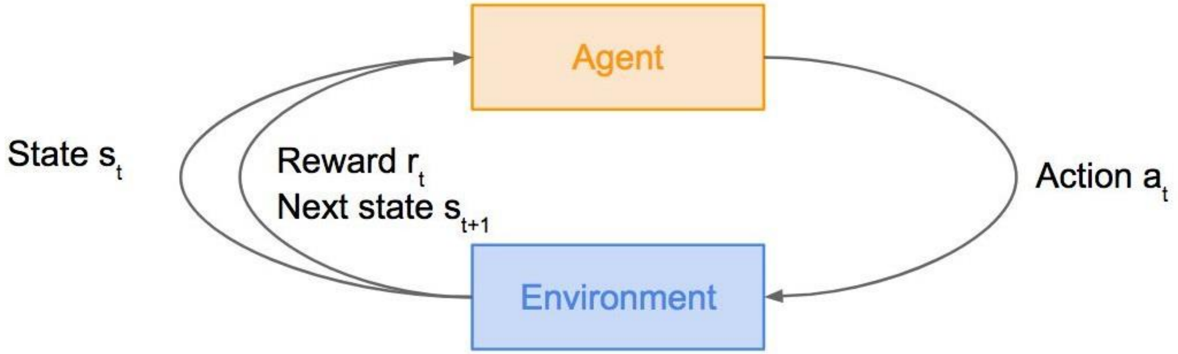
United Healthcare Group

Hyderabad

Today: Reinforcement Learning

Problems involving an **agent** interacting with an **environment**, which provides numeric **reward** signals

Goal: Learn how to take actions in order to maximize reward



Overview

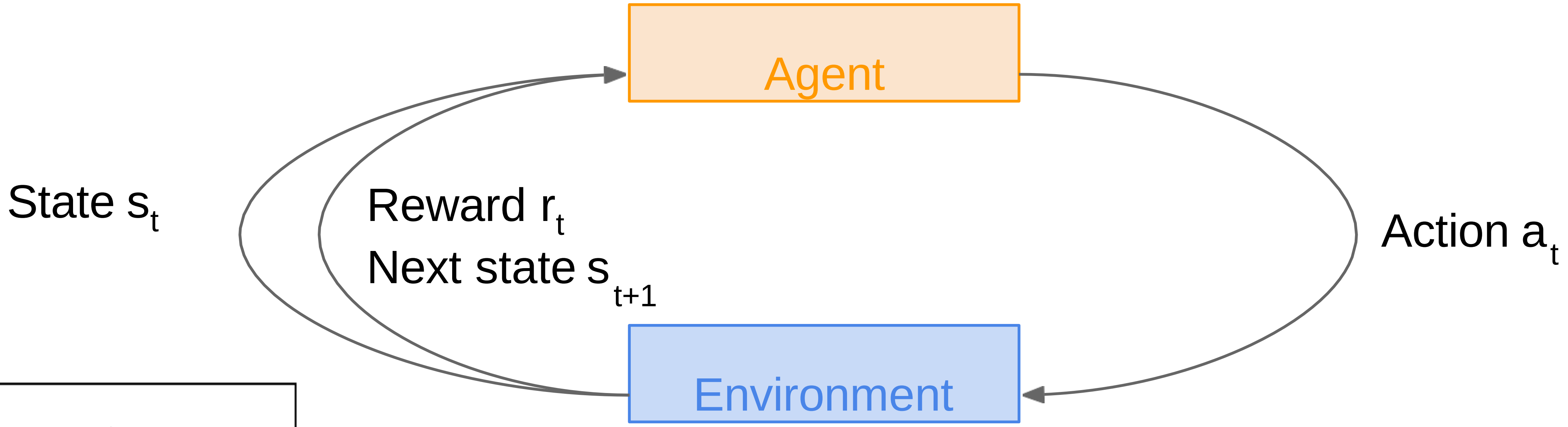
- What is Reinforcement Learning?
- Markov Decision Processes
- Q-Learning
- Policy Gradients

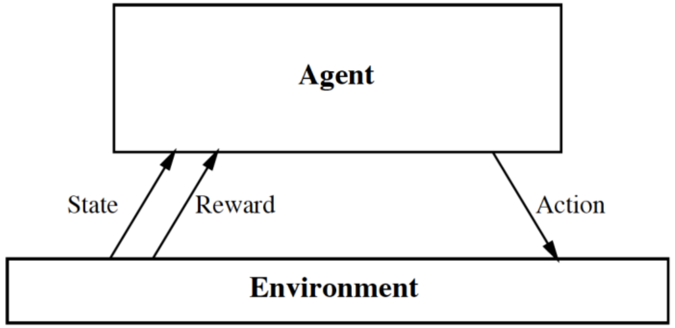
Basic Differences

- **Supervised vs Reinforcement Learning**: In supervised learning, there’s an external “supervisor”, which has knowledge of the environment and who shares it with the agent to complete the task. But there are some problems in which there are so many combinations of subtasks that the agent can perform to achieve the objective. So that creating a “supervisor” is almost impractical. For example, in a chess game, there are tens of thousands of moves that can be played. So creating a knowledge base that can be played is a tedious task. In these problems, it is more feasible to learn from one’s own experiences and gain knowledge from them. This is the main difference that can be said of reinforcement learning and supervised learning. In both supervised and reinforcement learning, there is a mapping between input and output. But in reinforcement learning, there is a reward function which acts as a feedback to the agent as opposed to supervised learning.

- **Unsupervised vs Reinforcement Leanring**: In reinforcement learning, there’s a mapping from input to output which is not present in unsupervised learning. In unsupervised learning, the main task is to find the underlying patterns rather than the mapping. For example, if the task is to suggest a news article to a user, an unsupervised learning algorithm will look at similar articles which the person has previously read and suggest anyone from them. Whereas a reinforcement learning algorithm will get constant feedback from the user by suggesting few news articles and then build a “knowledge graph” of which articles will the person like.

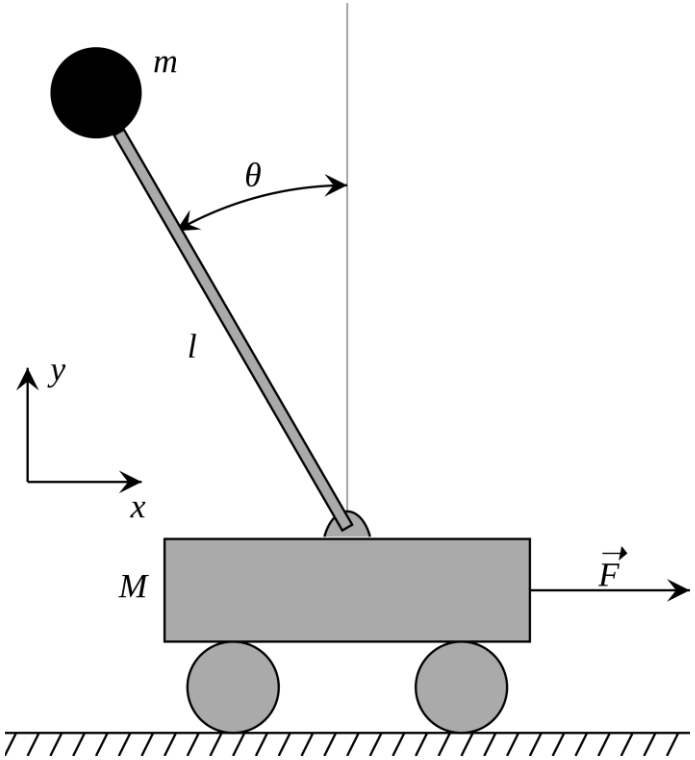
Reinforcement Learning





$$s_0 \xrightarrow[r_0]{a_0} s_1 \xrightarrow[r_1]{a_1} s_2 \xrightarrow[r_2]{a_2} \cdots$$

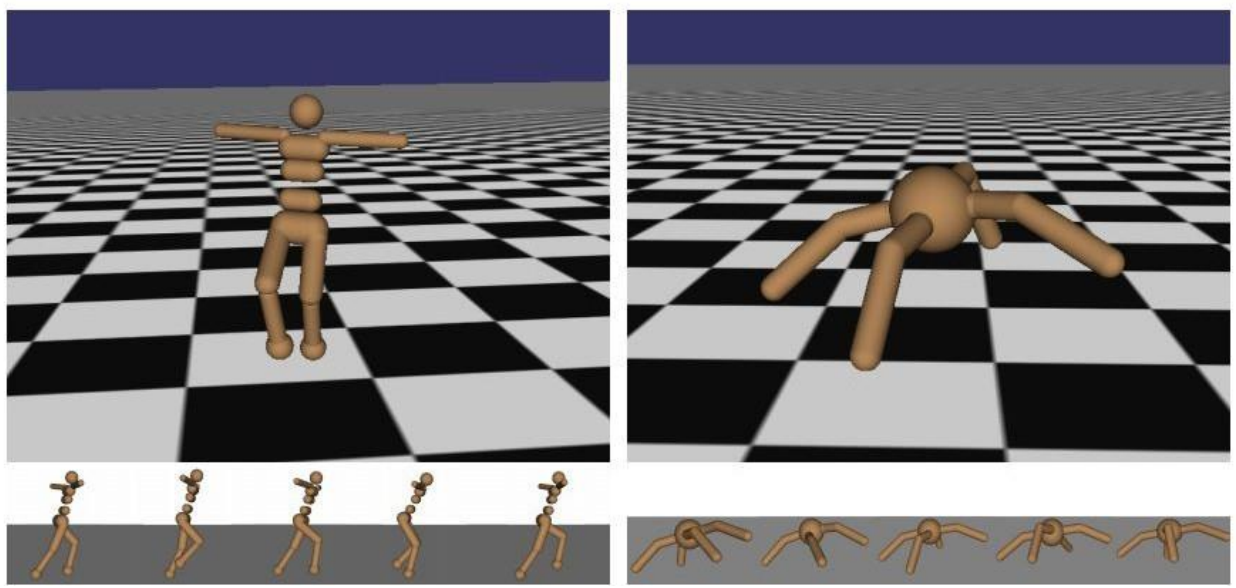
Cart-Pole Problem



Objective: Balance a pole on top of a movable cart

State: angle, angular speed, position, horizontal velocity
Action: horizontal force applied on the cart
Reward: 1 at each time step if the pole is upright

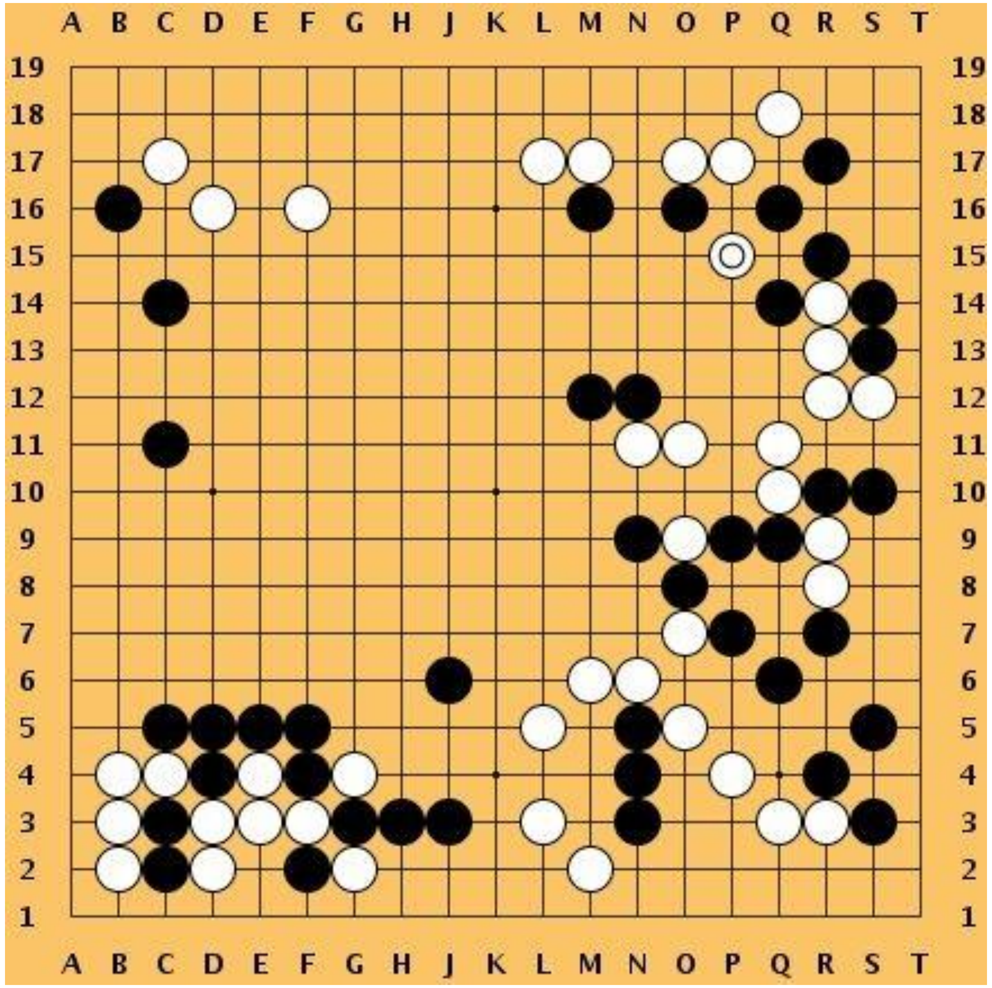
Robot Locomotion



Objective: Make the robot move forward

State: Angle and position of the joints
Action: Torques applied on joints
Reward: 1 at each time step upright + forward movement

Go



Objective: Win the game!

State: Position of all pieces
Action: Where to put the next piece down
Reward: 1 if win at the end of the game, 0 otherwise

Markov Decision Process

- Mathematical formulation of the RL problem
- **Markov property**: Current state completely characterises the state of the world

Defined by: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{P}, \gamma)$

$\mathcal{S}$: set of possible states
$\mathcal{A}$: set of possible actions
$\mathcal{R}$: distribution of reward given (state, action) pair
$\mathbb{P}$: transition probability i.e. distribution over next state given (state, action) pair
γ : discount factor

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 $r_t \sim R(. \mid s_t, a_t)$
  - Environment samples next state $s_{t+1} \sim P(. \mid s_t, a_t)$
  - Agent receives reward r_t and next state s_{t+1}

- A policy u is a function from S to A that specifies what action to take in each state
- **Objective**: find policy u* that maximizes cumulative discounted reward: $\sum_{t \geq 0} \gamma^t r_t$

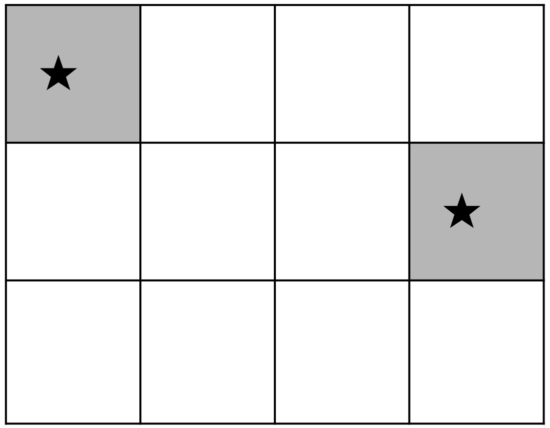
A simple MDP: Grid World

actions = {

1. right
2. left
3. up
4. down

}

states



Set a negative “reward”
for each transition
(e.g. $r = -1$)

Objective: reach one of terminal states (greyed out) in least number of actions

The optimal policy

We want to find optimal policy u* that maximizes the sum of rewards.

How do we handle the randomness (initial state, transition probability…)?
Maximize the **expected sum of rewards!**

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

Definitions: Value function and Q-value function

Following a policy produces sample trajectories (or paths) s_0, a_0, r_0, s_1, a_1, r_1, …

How good is a state?
The **value function** at state s, is the expected cumulative reward from following the policy from state s:

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

How good is a state-action pair?
The **Q-value function** at state s and action a, is the expected cumulative reward from taking action a in state s and then following the policy:

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

Bellman equation

maximum total long term reward that one can obtain starting from state s and performing action a is the sum of current reward r plus the maximum total long term rewards that can be obtained from the next state s' weighted by the discount factor γ. Bellman equation is the central theoretical concept that is used in almost all formulations of reinforcement learning.

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

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

In practice, a derivative form of the Bellman equation in many implementations. This is an iterative updatin algorithm called the *Temporal Difference Learning alg* Here, upon the agent making a transition from state s s', the Q values are updated through the equation:

Q(s, a) = (1 – α) Q(s, a) + α (r + γ r

Learning Rate

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

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

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 u* corresponds to taking the best action in any state as specified by Q*

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 -> infinity

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

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

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

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

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

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

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**
  Initialise 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

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?

Gradient ascent on policy parameters!

Anomaly Detection through RL

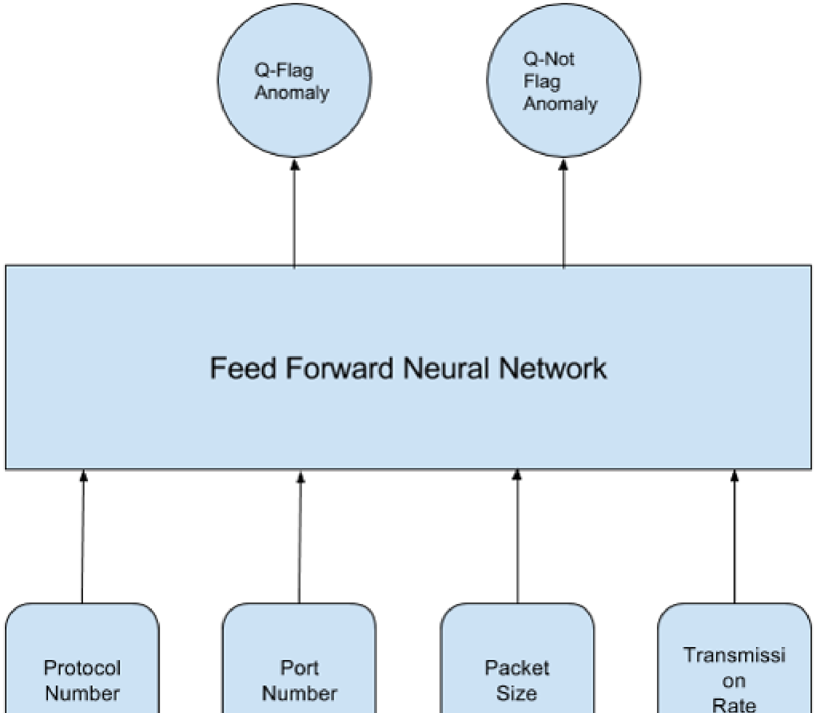
Objective is to find patterns in a dataset that do not conform to expected normal behaviour.

One can formulate the anomaly detection problem as a *reinforcement learning* problem, where an *autonomous agent* interacts with the environment and takes actions (such as allowing or denying access) and gets rewards from the environment (positive rewards for correct predictions of anomaly and negative rewards for wrong predictions) and over a period of time learns to predict anomalies with a high level of accuracy.

$Q(s, a) = (1 - \alpha)\, Q(s, a) + \alpha\, (r + \gamma \max_a Q(s', a'))$

In Bank Transaction or Credit Card default case, features are transaction amount, time of transaction, merchant name, Geography etc

Environment is created OpenAI Gym Toolkit



Implementation of Fraud Detection Learning Process

1. Initialize all the weights in DNN with random values.
2. Initialize the total accumulated reward to zero.
3. Get an initial state from the environment created using the OpenAI Gym and Credit Card dataset.
4. Repeat many episodes of learning, wherein each episode performs a series of explorations of the environment as follows:
   1. Start with the state obtained in the previous step.
   2. Perform a feed forward of the current state using DNN, and get the predicted Q(s, a) values.
   3. Take an action of either flag or not flag from the current state, according to the Q(s, a) values given by the output of the DNN in the previous state and in an ε-greedy manner.
   4. Get the reward and next state from the environment.
   5. Pass the new state also through the DNN, to compute the target Q(s, a) values using the Bellman's equation.
   6. Perform a training of the DNN by back propagation of the error of prediction, where the difference between target Q(s, a) and predicted Q(s, a) in step 4.2 is taken as the error of prediction.
   7. Compute the new cumulative total reward.
   8. Repeat steps 4.1 to 4.7 for a finite number of explorations.