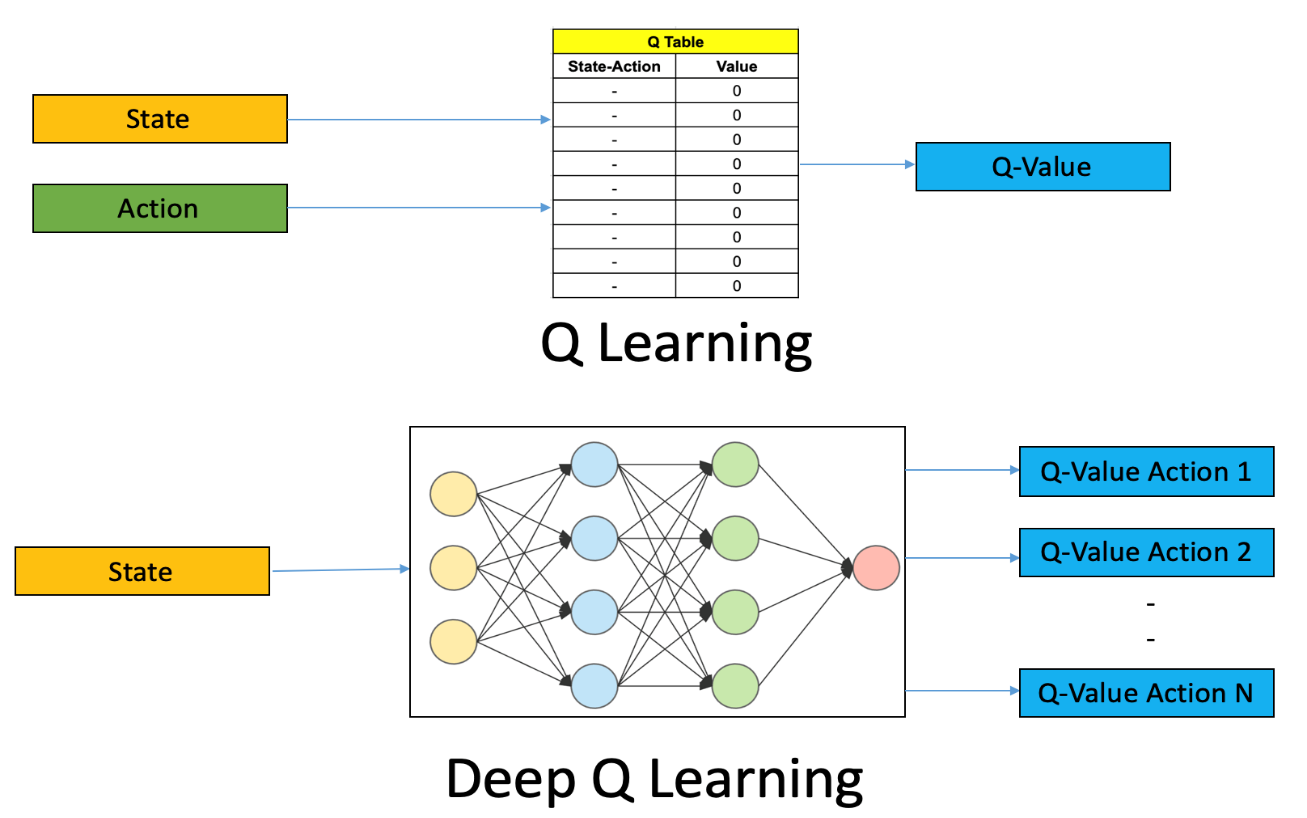
**DEEP Q LEARNING**

**-SAUMYA ARORA**



**WHAT IS REINFORCEMENT LEARNING?**

We must understand Reinforcement Learning before starting Deep Q Learning.

**Reinforcement Learning is an area of machine learning which is concerned with low intelligent agents who ought to take actions in an environment to maximize the notion of cumulative reward.** It is one of the three basic paradigms, alongside supervised learning and unsupervised learning.

Reinforcement Learning differs from supervised learning in not needing labeled input/output pairs to be presented and in not needing sub optional actions to be explicitly corrected. Instead, it focuses on finding a balance between **exploration** (of uncharted territory) and **exploitation** (of current knowledge).

**WHAT IS Q LEARNING?**

It is a basic form of Reinforcement Learning which uses Q -values (also called **action values**) to iteratively improve the behavior of the learning agent. It's considered off policy because the Q learning function learns from actions that are outside the current policy, like taking random actions, and therefore a policy isn't needed.

‘Q’ in Q-Learning stands for **quality**. Quality in this case represents how useful a given action is in gaining a future reward.

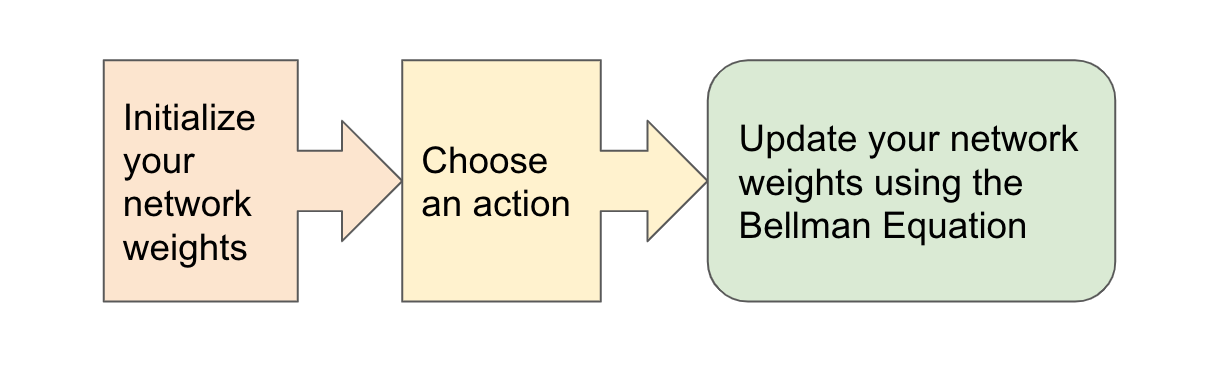
**WHAT IS DEEP-Q-LEARNING?**

Deep-Q-Learning uses user experience relay to learn in small batches to avoid skewing the dataset distribution of different states, actions, rewards, and next states that the neural network will see. Importantly the agent doesn't need to train after each step.

The process of Deep-Q-learning crates an exact matrix for the working agent which can be used to maximize its reward in the long run. Although this approach is not wrong in itself, this is only practical for very small environments and quickly losses its feasibility when the number of states and actions in the environment increases.

The solution for the above problem comes from the realization that the values in the matrix only have relative importance i.e., the values only have importance concerning other values. Thus, this thinking leads us to **Deep-Q-Learning** which uses a **deep neural network** to approximate the values. This approximation of values doesn’t hurt as long as the relative importance is preserved. The basic working step for Deep-Q-Learning is that the initial state is fed into the neural network and it returns the Q-values of all possible actions as on output.

**THE DEEP-Q-NETWORK ALGORITHM: -**



The algorithm mainly consists of 3 parts: -

1. **Initialize your main and target neural networks.**
2. **Choose an action using the Epsilon-Greedy Strategy.**
3. **Update your network weights using the Bellman Equation.**
4. **Initialize your target and main neural networks: -**

Deep learning replaces the regular Q-Table with a neural network, rather than mapping a state-action pair to a Q-value, a neural network maps inputs states to (action, Q-value) pairs.

1. **The importance of effective initialization: -**

To build a machine learning algorithm, usually, you had to define an architecture (e.g., logistic regression, support vector machine, neural network, etc.) and train it to learn parameters. Here is a common training process for a neural network: -

* 1. **Initialize the parameters**
  2. **Choose an optimizing algorithm**
  3. **Repeat these steps:**

**1.3.1) Forward propagate an input**

**1.3.2) Compute the cost function**

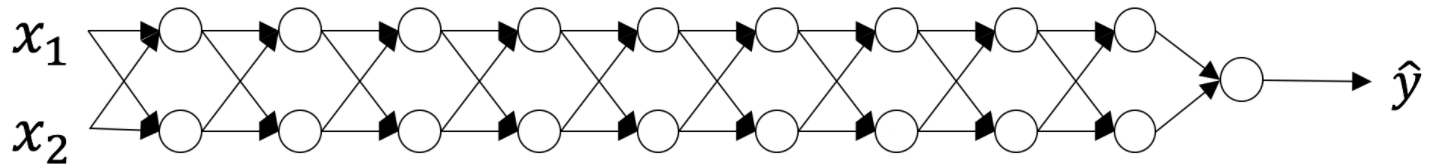
**1.3.3) Compute the gradients of the cost concerning parameters using backpropagation**

**1.3.4) Update each parameter using gradients, according to the optimization algorithm.**

Then given a new data point you can use the model to predict its class. The initializing step can be critical to the model’s ultimate performance and it requires the right method.

1. **The problem of exploding or vanishing gradients: -**

Consider this 9-layer neural network



At every iteration of the optimization loop (forward, cost, backward, update), we observe that backpropagated gradients are either amplified or minimized as you move from the output layer towards the input layer. This result makes sense if you consider the following example.

Assume all the activation functions are linear (identity function). Then the output activation is:

Were,

***L*=10**

are all matrices of size (2,2) because layers [1] to [L-1] have 2 neurons and receive 2 inputs. With this in mind, and for illustrative purposes, if we assume the output prediction is

### **Case 1: A too-large initialization leads to exploding gradients**

Consider the case where every weight is initialized slightly larger than the identity matrix.

This simplifies to and the values of  increase exponentially with **l**. When these activations are used in backward propagation, this leads to the exploding gradient problem. That is, the gradients of the cost with the respect to the parameters are too big. This leads the cost to oscillate around its minimum value.

### **Case 2: A too-small initialization leads to vanishing gradients**

Similarly, consider the case where every weight is initialized slightly smaller than the identity matrix.

This simplifies to and the values of the activation   decrease exponentially with **l**. When these activations are used in backward propagation, this leads to the vanishing gradient problem. The gradients of the cost concerning the parameters are too small, leading to convergence of the cost before it has reached the minimum value.

All in all, initializing weights with inappropriate values will lead to divergence or a slow-down in the training of your neural network. Although we illustrated the exploding/vanishing gradient problem with simple symmetrical weight matrices, the observation generalizes to any initialization values that are too small or too large.

1. **How to find appropriate initialization values**

To prevent the gradients of the network’s activations from vanishing or exploding, we will stick to the following rules of thumb:

* 1. **The *mean* of the activations should be zero.**
  2. **The *variance* of the activations should stay the same across every layer**.

Under these two assumptions, the backpropagated gradient signal should not be multiplied by values too small or too large in any layer. It should travel to the input layer without exploding or vanishing.

More concretely, consider a *layer ll*. Its forward propagation is:

We would like the following to hold:

Ensuring zero-mean and maintaining the value of the variance of the input of every layer guarantees no exploding/vanishing signal, as we'll explain in a moment. This method applies both to forward propagation (for activations) and backward propagation (for gradients of the cost to activations). The recommended initialization is Xavier initialization (or one of its derived methods), for every layer L:

**}**

In other words, all the weights of layer **l**are picked randomly from a *normal distribution* with mean *μ*=0 and variance ​ where  is the number of neurons in layer **l-1**. Biases are initialized with zeros.

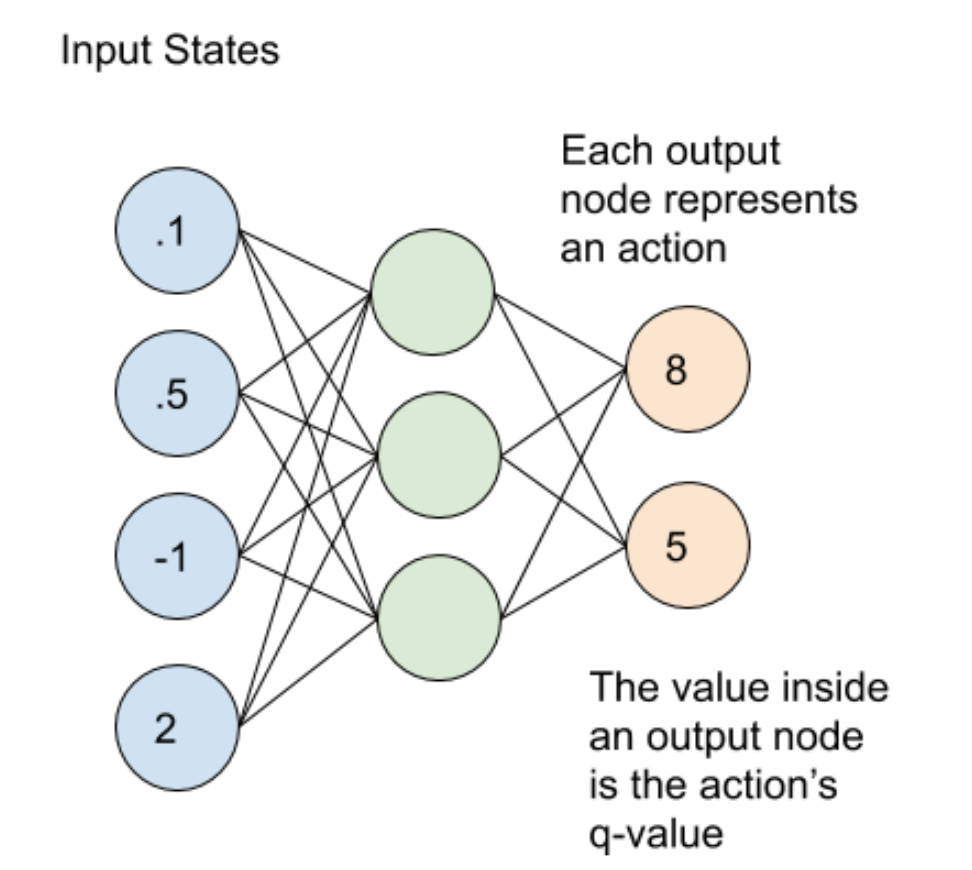
# Justification for Xavier initialization

Xavier Initialization keeps the variance the same across every layer. Here we will assume that our layer’s activations are normally distributed around zero. Sometimes it helps to understand the mathematical justification to grasp the concept, but you can understand the fundamental idea without the math.

A common math trick is to extract the summation outside the variance. To do this, we must make the following three *assumptions*:

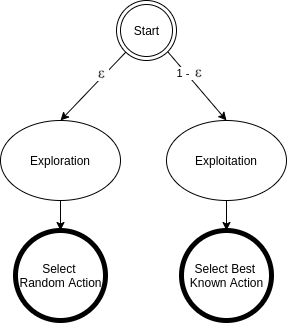
* 1. **Weights are independent and identically distributed**
  2. **Inputs are independent and identically distributed**
  3. **Weights and inputs are mutually independent**

**5. MAPPING OF STATES (ACTION, Q-VALUES) PAIRS**



The main and target neural networks map input states to an **(action, q-value)** pair. In this case, each output node (representing an action) contains the action’s q-value as a **floating-point number.**

1. **Choose an action using the Epsilon-Greedy Exploration Strategy: -**



**In epsilon-greedy action selection, the agent uses both exploitations to take advantage of prior knowledge and exploration to look for new options.** The epsilon greedy approach selects the action with the highest estimated reward most of the time. The aim is to have a balance to have some room for trying new things, sometimes contradicting what we have already learned.

With a small probability of what we choose to explore, i.e., not to exploit what we have learned so far. In this case, the action is selected randomly, independent of the action-value estimates.

**If we make infinite trials, each action is taken an infinite number of times. Hence, the epsilon-greedy action selection policy discovers the optimal actions for sure.**

**An epsilon-greedy algorithm is easy to understand and implement. Yet it’s hard to beat and works as well as more sophisticated algorithms.**We need to keep in mind that using other action selection methods is possible. Depending on the problem at hand, different policies can perform better.

For example, the **SoftMax** action selection strategy controls the relative levels of exploration and exploitation by mapping values into action probabilities. Here we use the same formula from the SoftMax activation function, which we use in the final layer of classifier neural networks.

## **1). Epsilon-Greedy Q-learning Parameters: -**

In this type of method, we have three parameters i.e., **alpha gamma, and the third one is epsilon (epsilon greedy action selection).**

* 1. **ALPHA (α): -**

Similar to other machine learning algorithms, **alpha (α)** **defines the learning rate or step size.** As we can see from the equation above, the new Q-value for the state is calculated by incrementing the old Q-value by alpha multiplied by the selected action’s Q-value.

Alpha is a real number between zero and one **(0 < α ≤ 1).** If we set alpha to zero, the agent learns nothing from new actions. Conversely, if we set alpha to 1, the agent completely ignores prior knowledge and only values the most recent information. Higher alpha values make Q-values change fast.

* 1. **GAMMA (ɣ): -**

**Gamma (ɣ) is the discount factor.** In Q-learning, gamma is multiplied by the estimation of the optimal future value. The next reward’s importance is defined by the gamma parameter.

Gamma is a real number between 0 and 1 **(0 ≤ ɣ ≤ 1).** If we set gamma to zero, the agent completely ignores the future rewards. Such agents only consider current rewards. On the other hand, if we set gamma to 1, the algorithm would look for high rewards in the long term. A high gamma value might prevent conversion: summing up non-discounted rewards leads to having high Q-values.

* 1. **EPSILON (): -**

**Epsilon () parameter is related to the epsilon-greedy action selection procedure in the Q-learning algorithm.** In the action selection step, we select the specific action based on the Q-values we already have. The epsilon parameter introduces randomness into the algorithm, forcing us to try different actions. This helps not getting stuck in a local optimum.

If epsilon is set to 0, we never explore but always exploit the knowledge we already have. On the contrary, having the epsilon set to 1 forces the algorithm to always take random actions and never use past knowledge. Usually, epsilon is selected as a small number close to 0.

1. **UPDATE YOUR NETWORK WEIGHTS USING THE EPSILON EQUATION: -**

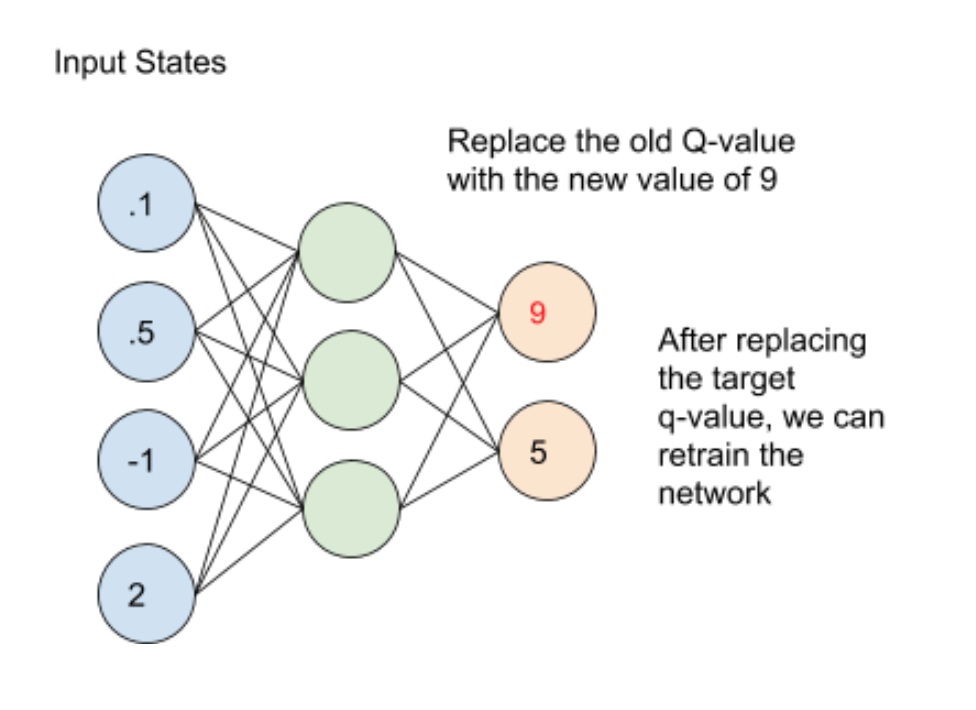
After choosing an action, it’s time for the agent to perform the action and update the Main and Target networks according to the Bellman equation. Deep Q-Learning agents use Experience Replay to learn about their environment and update the Main and Target networks.

To summarize, the **main network** samples and trains on a batch of past experiences every 4 steps. The main network weights are then copied to the **target network** weights every 100 steps.

# Experience Replay: -

**Experience Replay** is the act of storing and replaying game states (the state, action, reward, next state) that the RL algorithm can learn from. Experience Replay can be used in **Off-Policy** algorithms to learn in an offline fashion. Off-policy methods can update the algorithm's parameters using saved and stored information from previously taken actions. **Deep Q-Learning uses Experience Replay to learn in small batches to avoid skewing the dataset distribution of different states, actions, rewards, and next states that the neural network will see.** Importantly, the agent doesn't need to train after each step. In our implementation, we use Experience Replay to train on small batches once every 4 steps rather than every single step. We found this trick to help speed up our Deep Q-Learning implementation.

**BELLMAN EQUATION: -**



**A Bellman equation is a necessary condition for optimality associated with the mathematical optimization method known as dynamic programming.** It writes the "value" of a decision problem at a certain point in time in terms of the payoff from some initial choices and the "value" of the remaining decision problem that results from those initial choices.

This breaks a dynamic optimization problem into a sequence of simpler subproblems, as Bellman's “principle of optimality” prescribes.

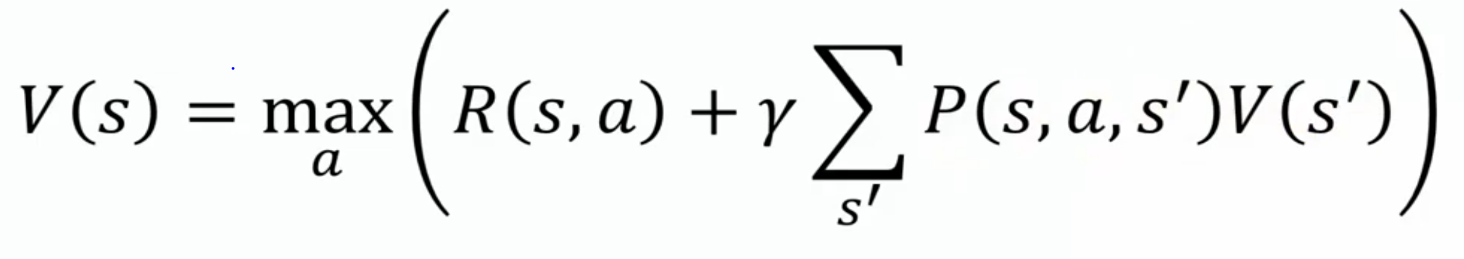
The term 'Bellman equation' usually refers to the dynamic programming equation associated with discrete-time optimization problems. In continuous-time optimization problems, the analogous equation is a partial differential equation that is called the **Hamilton–Jacobi–Bellman equation.**

In discrete-time, any multi-stage optimization problem can be solved by analyzing the appropriate Bellman equation. The appropriate Bellman equation can be found by introducing new state variables (state augmentation). However, the resulting augmented-state multi-stage optimization problem has a higher dimensional state space than the original multi-stage optimization problem - an issue that can potentially render the augmented problem intractable due to the “curse of dimensionality”. Alternatively, it has been shown that if the cost function of the multi-stage optimization problem satisfies a **"backward separable"** structure then the appropriate Bellman equation can be found without state augmentation.

**The optimal value function V\*(S) yields the maximum value.**

The value of a given state is equal to the max action (action which maximizes the value) of thereward of the optimal action in the given state and add a discount factor multiplied by thenext state’s Value from the Bellman Equation.

Let's understand this equation, **V(s)** is the value for being in a certain state. **V(s’)** is the value for being in the next state that we will end up in after taking action a**. R (s, a)** is the reward we get after taking action an in states. As we can take different actions so we use maximum because our agent wants to be in the optimal state. **γ** is the discount factor as discussed earlier. This is the Bellman equation in the deterministic environment. It will be slightly different for a non-deterministic environment or a stochastic environment.



**In a stochastic environment when we take an action it is not confirmed that we will end up in a particular next state and there is a probability of ending in a particular state. P (s, a, s’) is the probability of ending is state’s from s by taking action a. This is summed up to a total number of future states.** For example, if by taking an action we can end up in 3 states s₁, s₂, and s₃ from states with a probability of 0.2, 0.2, and 0.6. The Bellman equation will be

**V(s) = maxₐ (R (s, a) + γ(0.2\*V(s₁) + 0.2\*V(s₂) + 0.6\*V(s₃) )**

We can solve the Bellman equation using a special technique called dynamic programming.

**Dynamic Programming**

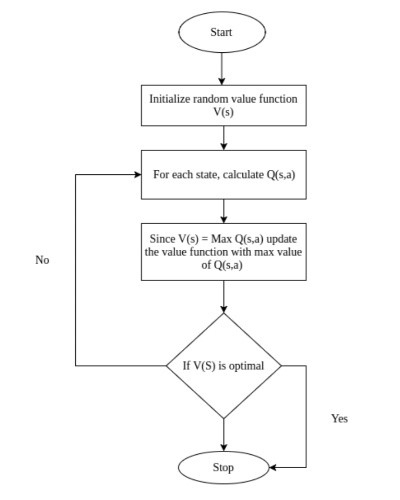
Dynamic programming (DP) is a technique for solving complex problems. In DP, **instead of solving complex problems one at a time, we break the problem into simple subproblems, then for each sub-problem, we compute and store the solution**. If the same subproblem occurs, we will not recompute, instead, we use the already computed solution.

We solve a Bellman equation using two powerful algorithms:

* **Value Iteration**
* **Policy Iteration**

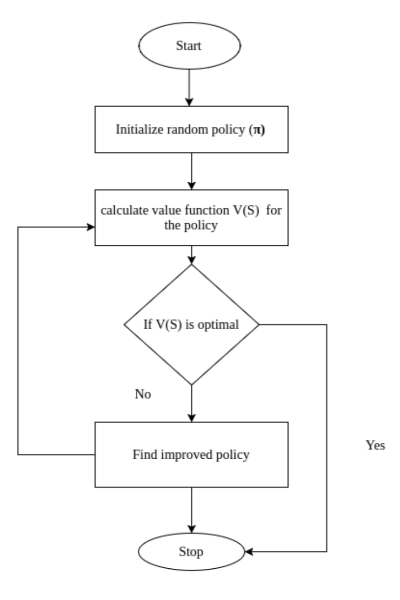
**Value Iteration**

We will learn it using diagrams and programs.



In value iteration, we start with a random value function. As the value table is not optimized if randomly initialized, we optimize it iteratively.

**Policy Iteration**



In Policy Iteration the actions which the agent needs to take are decided or initialized first and the value table is created according to the policy.

**Codes: -**

Initialize your main and target neural networks, Epsilon-Greedy Strategy, Bellman Equation (**Jupyter notebook**)

**Resources: -**

https://www.deeplearning.ai/ai-notes/initialization/ **initialization the main neural network** https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/ **main part**.

https://scholar.google.co.in/scholar?q=deep+q+learning&hl=en&as\_sdt=0&as\_vis=1&oi=scholart **for articles**

http://proceedings.mlr.press/v48/gu16.pdf **main part**

https://www.geeksforgeeks.org/what-is-reinforcement-learning/ **reinforcement learning**

https://www.geeksforgeeks.org/q-learning-in-python/#:~:text=Q%2DLearning%20is%20a%20basic,the%20action%20at%20the%20state%20. **Q learning**

https://www.baeldung.com/cs/epsilon-greedy-q-learning#:~:text=In%20epsilon%2Dgreedy%20action%20selection,balance%20between%20exploration%20and%20exploitation. **epsilon greedy**

https://medium.com/analytics-vidhya/bellman-equation-and-dynamic-programming-773ce67fc6a7 **bellman equation**