# A

# PROJECT REPORT

## ON

# STOCHASTIC CHESS PIECE MOVEMENT USING DQN


## CENTRAL INSTITUTE OF TOOL DESIGN

## HYDERABAD

BY

**K. JAYASREE**

**E. VISHWA SAI**

## UNDER THE GUIDANCE OF

(S. VAMSI, M.Tech)
(Industry4.0 Department)


DEPARTMENT OF COMPUTER ENGINEERING

# GOVERNMENT INSTITUTE OF ELECTRONICS EAST MAREDEPALLY, SECUNDERABAD

Affiliated to SBTET

MINISTRY OF MICRO SMALL & MEDIUM ENTERPRISES GOVERNMENT OF INDIA

एमएसएमई-टूलरूम,हैदराबाद
MSME-TOOL ROOM, HYDERABAD
केंद्रीय उपकरण अभिकल्प संस्थान
Central Institute of Tool Design

# BONAFIDE CERTIFICATE

This is to certify that this project report entitled "**STOCHASTIC CHESS PIECE MOVEMENT USING DQN "** submitted to MSME TOOL ROOM(CITD),BALANAGAR, is a Bonafide record of work done by **K.JAYASREE (21054-CS-036) , E.VISHWA SAI (21054-CS-038)** under my supervision from 16th December 2023 to 15th June 2024.

(S. VAMSI )
Training guide

( G. SANATH KUMAR)
Dy. Director (Trg.)

# ACKNOWLEDGEMENT

I would like to extend my heartfelt thanks to my guide S. VAMSI for his continuous assistance, guidance, support, views, ideas encouragement and feedback without this project would have been a distant dream.

I would like thank and honor our head of the department G. SANATH KUMAR, DEPUTY DIRECTOR who advised me throughout the completion of this work.

We express heart-full thanks for the employees who contributed either directly or indirectly for the successful completion of industrial training.

We thank each and every one of MSME TOOL ROOM, HYDERABAD(CITD) For their kind cooperation and all sorts of help bring out this industrial training Successfully.

K. JAYASREE (21054-CS-036)

E. VISHWA SAI(21054-CS-038)

# CONTNET

# LIST OF FIGURES

# LIST OF ABBREVATIONS

RL- Reinforcement Learning

MDP- Markov Decision Processes

DQN- Deep Q-network

DP- Dynamic Programming

GPI- Generalized Policy iteration MC- Monte Carlo

TD- Temporal difference

# ABSTRACT

Reinforcement learning (RL) is a machine learning approach where an agent learns to make decisions by interacting with an environment to maximize cumulative reward. The field has come a long way since then, evolving and maturing in several directions. Reinforcement learning has gradually become one of the most active research areas in machine learning, artificial intelligence, and neural network research.

This research explores the application of Deep Q-Network (DQN) algorithms to the problem of optimizing chess piece movement, a fundamental aspect of chess strategy. This work aims to develop an AI agent capable of making precise and effective moves.

The DQN algorithm is utilized to approximate the optimal Q-values for each possible movement, allowing the agent to learn the best actions from given board states. Training is conducted through self-play, where the agent receives feedback based on the outcomes of its moves, including rewards for advantageous positions and penalties for errors.

This study demonstrates the potential of reinforcement learning in enhancing the tactical capabilities of chess playing-agents.

# CHAPTER- 1

# INTRODUCTION

Learning from interaction is a foundational idea underlying nearly all theories of learning and intelligence.

Solving problems evaluating the designs through mathematical analysis or computational experiments. The approach,called reinforcement learning.

## 1.1 Reinforcement learning

Reinforcement learning is a learning what to do how to map situations to actions so as to maximize a numerical reward signal.The learner is not told which actions to take, as in most forms of machine learning,But instead must discover which action yield the most rewards by trying them.

Trial-and-error search and delayed reward are the two most important distinguishing features of reinforcement learning.

Reinforcement learning is defined not by characterizing learning methods, but by characterizing a learning problem.A full specification of the reinforcement learning problem in terms of optimal control of Markov decision processes.

The real problem facing a learning agent interacting with its environment to achieve a goal.The idea that we learn by interacting with our environment is probably the first to occur to us when we think about the nature of learning.

Another key feature of reinforcement learning is that it explicitly considers the whole problem of a goal-directed agent interacting with an uncertain environment.

Interaction between an active decision-making agent and its environment, within which the agent seeks to achieve a goal despite uncertainty about its environment.

All reinforcement learning agents have explicit goals, can sense aspects of their environments, and can choose actions to influence their environments.

Moreover, it is usually assumed from the beginning that the agent has to operate despite significant uncertainty about the environment it faces. The agent ' s actions are permitted to affect the future state of the environment.The agent can use its experience to improve its performance over time.

In reinforcement learning we extend ideas from optimal control theory and stochastic approximation to address the broader and more ambitious goals of artificial intelligence.

The term “optimal control” came into use in the late 1950s to describe the problem of designing a controller to minimize a measure of a dynamical system’s behavior over time.

This approach uses the concepts of a dynamical system’s state and of a value function, or “optimal return function,” to define a functional equation, now often called the Bellman equation.

One of the challenges that arise in reinforcement learning and not in other kinds of learning is the trade-off between exploration and exploitation.The agent has to exploit what it already knows in order to obtain reward, but it also has to explore in order to make better action selections in the future.

## 1.2 Elements of Reinforcement Learning

Four main sub elements of a RL system:a policy, a reward function, a value function, and, optionally, a model of the environment.

**Policy**

A policy defines the learning agent's way of behaving at a given time.A policy is a mapping from perceived states of the environment to actions to be taken when in those states.In some cases the policy may be a simple function or lookup table.The policy is the core of a reinforcement learning agent in the sense that it alone is sufficient to determine behavior.

**Reward Function**

A reward function defines the goal in a reinforcement learning problem.It maps each perceived state of the environment to a single number, a reward, indicating the desirability of that state.A reinforcement learning agent's objective is to maximize the total reward it receives in the long run.The reward function defines what are the good and bad events for the agent serve as a basis for altering the policy.They are the immediate and defining features of the problem faced by the agent.Rl is based on the reward hypothesis.

**Value Function**

Value function specifies what is good in the long run. The value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state.Values indicate the long-term desirability of states after taking into account the states that are likely to follow, and the rewards available in those states.Values, as predictions of rewards.Action choices are made based on value judgments.We seek actions that bring about states of highest value, not highest reward.The most important component of almost all reinforcement learning algorithms is a method for efficiently estimating values. It is much harder to determine values than it is to determine rewards.

**Model**

Model of the environment is something that mimics the behavior of the environment.The model might predict the resultant next state and next reward.Models are used for planning, by which we mean any way of deciding on a course of action by considering possible future situations before they are actually experienced.

Most of the time we move **greedily**, selecting the move that leads to the state with greatest value, that is, with the highest estimated probability.

we select randomly from among the other moves instead. These are called **exploratory** moves because they cause us to experience states that we might otherwise never see.

Reinforcement learning is the first field to seriously address the computational issues that arise when learning from interaction with an environment in order to achieve long-term goals.

The concepts of value and value functions are the key features of the reinforcement learning methods.

All of these are essential elements underlying the theory and algorithms of modern reinforcement learning.

## 1.3 Evaluative Feedback

The most important feature distinguishing reinforcement learning from other types of learning is that it uses training information that evaluates the actions taken rather than instructs by giving correct actions. Purely evaluative feedback indicates how good the action taken is, but not whether it is the best or the worst action possible.Each action has an expected or mean reward given that that action is selected,let us call this the value of that action.
At any time there is at least one action whose estimated value is greatest. We call this a greedy action.
Select a greedy action, we say that you are exploiting your current knowledge of the values of the actions.If instead you select one of the ~~non greedy~~ actions, then we say you are exploring because this enables you to improve your estimate of the non greedy action's value. Exploitation is the right thing to do to maximize the expected reward on the one play, but exploration may produce the greater total reward in the long run.
Reward is lower in the short run, during exploration, but higher in the long run because after you have discovered the better actions, you can exploit them.

### 1.3.1 Action-value methods

Estimating the values of actions and for using the estimates to make action selection decisions.the true (actual) value of action a as Q ∗ (a), and the estimated value at the t th play as Qt(a).True value of an action is the mean reward received when that action is selected.One way to estimate this is by averaging the rewards actually received when the action was selected.

$$Q_t(a) = \frac{r_1 + r_2 + \cdots + r_{k_a}}{k_a}.$$

Fig 1.1 Action-value method

The simplest action selection rule is to select the action (or one of the actions) with highest estimated action value.This method always exploits current knowledge to maximize immediate reward.

### 1.3.2 Incremental Implementation

Reward following a selection of action a requires more memory to store it and results in more computation being required to determine Qt(a).
It is easy to devise incremental update formulas for computing averages with small, constant computation required to process each new reward.

$$NewEstimate \leftarrow OldEstimate + StepSize\left[Target - OldEstimate\right]$$

Fig 1.2 Incremental Implementation

The expression Target − OldEstimate is an error in the estimate. It is reduced by taking a step toward the "Target."
The step-size parameter (StepSize) used in the incremental method described above changes from time step to time step.

## 1.4 Exploration & Exploitation

### 1.4.1 ε-greedy method:

To behave greedily most of the time, but every once in a while, say with small probability ε, instead select an action at random, uniformly, independently of the action-value estimates.Methods using this near-greedy action selection rule ε -greedy methods.This of course implies that the probability of selecting the optimal action converges to greater than 1 − ε, that is, to near certainty.

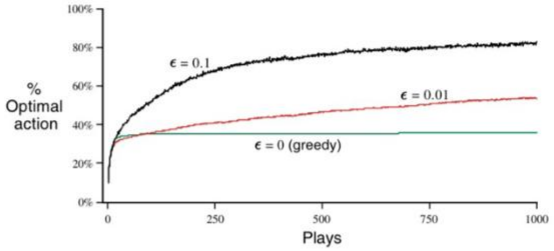


Fig 1.3 ε-greedy method

With noisier rewards it would take more exploration to find the optimal action, and ε -greedy methods would fare even better relative to the greedy method.

The advantage of ε -greedy over greedy methods depends on the task.Reinforcement learning requires a balance between exploration and exploitation.

### 1.4.2 Softmax Action Selection

Although ε -greedy action selection is an effective and popular means of balancing exploration and exploitation in reinforcement learning, one drawback is that when it explores it chooses equally among all actions.
This means that it is as likely to choose the worst-appearing action as it is to choose the next-to-best action.
The greedy action is still given the highest selection probability, but all the others are ranked and weighted according to their value estimates. These are called softmax action selection.
The most common softmax method uses a Gibbs, or Boltzmann, distribution. It chooses action a on the tth play with probability.

$$\frac{e^{Q_t(a)/\tau}}{\sum_{b=1}^{n} e^{Q_t(b)/\tau}},$$

Fig 1.4 Softmax action selection

Whether softmax action selection or ε -greedy action selection is better is unclear and may depend on the task and on human factors.
Initial action values can also be used as a simple way of encouraging exploration.

## 1.5 Markov Decision Processes

### 1.5.1 The Agent–Environment Interface

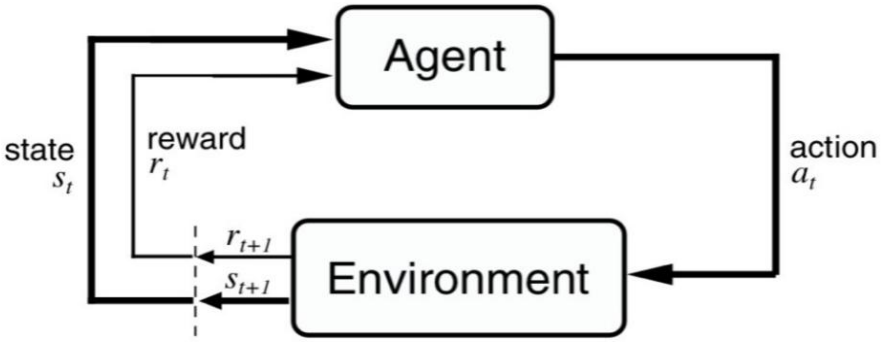


Fig 1.5 The agent–environment interaction.

The reinforcement learning problem is meant to be a straightforward framing of the problem of learning from interaction to achieve a goal.

The learner and decision- maker is called the agent. The thing it interacts with, comprising everything outside the agent, is called the environment. These interact continually, the agent selecting actions and the environment responding to those actions and presenting new situations to the agent.

The environment also gives rise to rewards, special numerical values that the agent tries to maximize over time.

A complete specification of an environment defines a task, one instance of the reinforcement learning problem.At each time step, the agent implements a mapping from states to probabilities of selecting each possible action. This mapping is called the agent's policy and is denoted π.

Any problem of learning goal-directed behavior can be reduced to three signals passing back and forth between an agent and its environment.

One signal to represent the choices made by the agent (the actions), one signal to represent the basis on which the choices are made (the states), and one signal to define the agent's goal (the rewards).

The agent – environment boundary represents the limit of the agent ' s absolute control, not of its knowledge.The agent's goal is to maximizing not immediate reward, but cumulative reward in the long run.

The use of a reward signal to formalize the idea of a goal is one of the most distinctive features of reinforcement learning.In the simplest case the return is the sum of the rewards:

Rt =rt+1 +rt+2 +rt+3 +⋯+rT

### 1.5.2 Markov property

The agent makes its decisions as a function of a signal from the environment called the environment's state. Define a property of environments and their state signals that is of particular interest, called the Markov property. " The state " we mean whatever information is available to the agent.

A state signal that succeeds in retaining all relevant information is said to be Markov, or to have the Markov property.

That is, the best policy for choosing actions as a function of a Markov state is just as good as the best policy for choosing actions as a function of complete histories.

The Markov property is important in reinforcement learning because decisions and values are assumed to be functions only of the current state.

### 1.5.3 Markov Decision Processes

A reinforcement learning task that satisfies the Markov property is called a Markov decision process, or MDP.

If the state and action spaces are finite, then it is called a finite Markov decision process (finite MDP). A particular finite MDP is defined by its state and action sets and by the one-step dynamics of the environment. Given any state and action, s and a, the probability of each possible next state, s′, is

$$\mathcal{P}^a_{ss'} = \Pr\left\{s_{t+1} = s' \mid s_t = s, a_t = a\right\}.$$

Fig 1.6 probability of next state

These quantities are called transition probabilities. Similarly, given any current state and action, s and a, together with any next state, s′, the expected value of the next reward is

$$\mathcal{R}^a_{ss'} = E\left\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\right\}.$$

Fig 1.7 Expected value of the next reward

A transition graph is a useful way to summarize the dynamics of a finite MDP.

### 1.5.4 Value functions:

The notion of "how good" here is defined in terms of future rewards that can be expected, or, to be precise, in terms of expected return.The rewards the agent can expect to receive in the future depend on what actions it will take.

The value of a state s under a policy π, denoted Vπ(s), is the expected return when starting in s and following π .

$$V^\pi(s) = E_\pi\left\{R_t \mid s_t = s\right\} = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, s_t = s\right\},$$

Fig 1.8 State-value function

$$Q^\pi(s,a) = E_\pi\left\{R_t \mid s_t = s, a_t = a\right\} = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, s_t = s, a_t = a\right\}.$$

Fig 1.9 Action-value function

The additional concept that we need is that of discounting. According to this approach, the agent tries to select actions so that the sum of the discounted rewards it receives over the future is maximized.where γ is a parameter, 0 ≤ γ ≤ 1, called the discount rate.

If γ = 0, the agent is "myopic" in being concerned only with maximizing immediate rewards.As γ approaches 1, the objective takes future rewards into account more strong.

### 1.5.5 Optimal value function:

A policy π is defined to be better than or equal to a policy π′ if its expected return is greater than or equal to that of π ′ for all states.

There is always at least one policy that is better than or equal to all other policies. This is an optimal policy.Optimal policies also share the same optimal action-value function,

$$V^*(s) = \max_{\pi} V^{\pi}(s),$$

Fig 1.10 Optimal value function

Another way of saying this is that any policy that is greedy with respect to the optimal value function V ∗ is an optimal policy.

An MDP is "solved" when we know the optimal value function.

# CHAPTER-2

# ELEMENTARY SOLUTION METHODS

Three fundamental classes of methods for solving the reinforcement learning problem: dynamic programming, Monte Carlo methods, and temporal-difference learning.
All of these methods solve the full version of the problem, including delayed rewards.
The methods also differ in several ways with respect to their efficiency and speed of convergence.

## 2.1 Dynamic Programming

The term dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process (MDP).
DP provides an essential foundation for the understanding of the methods of RL.
DP ideas can be applied to problems with continuous state and action spaces, exact solutions are possible only in special cases.
The key idea of DP, and of reinforcement learning generally, is the use of value functions to organize and structure the search for good policies.
As we shall see, DP algorithms are obtained by turning Bellman equations such as these into assignments, that is, into update rules for improving approximations of the desired value functions.
DP algorithms such as Policy evaluation,Policy iteration,Value iteration.

### 2.1.1 Policy Evaluation

First we consider how to compute the state-value function V π for an arbitrary policy π . This is called policy evaluation in the DP.
The initial approximation, V0, is chosen arbitrarily (except that the terminal state, if any, must be given value 0), and each successive approximation is obtained by using the Bellman equation for V (π )as an update rule:

$$V_{k+1}(s) = E_\pi\left\{r_{t+1} + \gamma V_k(s_{t+1}) \mid s_t = s\right\}$$

$$= \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'}\left[\mathcal{R}^a_{ss'} + \gamma V_k(s')\right],$$

Fig 2.1 Policy Evaluation

Vk = V π is a fixed point for this update rule because the Bellman equation for V π assures us of equality in this case
The sequence {Vk} can be shown in general to converge to V π as k → ∞ under the same conditions that guarantee the existence of V π . This algorithm is called iterative policy evaluation.
It replaces the old value of s with a new value obtained from the old values of the successor states of s, and the expected immediate rewards, along all the one-step transitions possible under the policy being evaluated.
Each iteration of iterative pol- icy evaluation backs up the value of every state once to produce the new approximate value function Vk+1.
Backups done in DP algorithms are called full backups because they are based on all possible next states rather than on a sample next state.

## 2.1.2 Policy Improvement

Computing the value function for a policy is to help find better policies consider selecting a in s and thereafter following the existing policy, π .

The key criterion is whether this is greater than or less than V π (s ). If it is greater— that is, if it is better to select a once in s and thereafter follow π than it would be to follow π all the time—then one would expect it to be better still to select a every time s is encountered, and that the new policy would in fact be a better one overall.

That this is true is a special case of a general result called the policy improvement theorem. Let π and π′ be any pair of deterministic policies such that, for all s ∈ S, $Q\pi(s,\pi'(s)) \geq V\pi(s)$.

Then the policy π ′ must be as good as, or better than, π . That is, it must obtain greater or equal expected return from all states s ∈ S: $V\pi'(s) \geq V\pi(s)$.

The idea behind the proof of the policy improvement theorem is easy to understand. Starting $Q\pi(s,\pi'(s)) \geq V\pi(s)$ from, we keep expanding the Qπ side using 2.1.1 and reapplying $Q\pi(s,\pi'(s)) \geq V\pi(s)$ until we get V π′(s).

This is the same as the Bellman optimality equation (4.1), and therefore, V π ′ must be V∗ , and both π and π′ must be optimal policies.

Policy improvement thus must give us a strictly better policy except when the original policy is already optimal.

## 2.1.3 Policy Iteration

Once a policy, π , has been improved using V π to yield a better policy, π ′ , we can then compute V π ′ and improve it again to yield an even better π′′ .

$$\pi_0 \xrightarrow{E} V^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V^{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi^* \xrightarrow{E} V^*,$$

where $\xrightarrow{E}$ denotes a policy *evaluation* and $\xrightarrow{I}$ denotes a policy *improvement*.

Fig 2.2 Policy Iteration.

A finite MDP has only a finite number of policies, this process must converge to an optimal policy and optimal value function in a finite number of iterations.This way of finding an optimal policy is called policy iteration.

That each policy evaluation, itself an iterative computation, is started with the value function for the previous policy. This typically results in a great increase in the speed of convergence of policy evaluation.

## 2.1.4 Value Iteration

One drawback to policy iteration is that each of its iterations involves policy evaluation.

The policy evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration. One important special case is when policy evaluation is stopped after just one sweep (one backup of each state). This algorithm is called value iteration.

That value iteration is obtained simply by turning the Bellman optimality equation into an update rule.

**EFFICIENCY OF DP:**

If n and m denote the number of states and actions, this means that a DP method takes a number of computational operations that is less than some polynomial function of n and m. A DP method is guaranteed to find an optimal policy in polynomial time even though the total number of (deterministic) policies is m^n.

DP is sometimes thought to be of limited applicability because of the curse of dimensionality , and it requires prior knowledge of Model.

## 2.2 Generalized Policy Iteration

Policy iteration consists of two simultaneous, interacting processes, one making the value function consistent with the current policy (policy evaluation), and the other making the policy greedy with respect to the current value function (policy improvement).

As long as both processes continue to update all states, the ultimate result is typically the same—convergence to the optimal value function and an optimal policy.

The term generalized policy iteration (GPI) to refer to the general idea of interacting policy evaluation and policy improvement processes, independent of the granularity and other details of the two processes.

The key idea of DP, and of reinforcement learning generally, is the use of value functions to organize and structure the search for good policies.

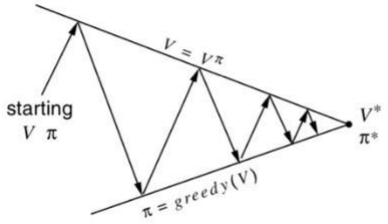


Fig 2.3 Generalized Policy Iteration

## 2.3 Monte-Carlo methods

Here we do not assume complete knowledge of the environment. Monte Carlo methods require only experience sample sequences of states, actions, and rewards from on-line or simulated interaction with an environment.
Learning from on-line experience is striking be- cause it requires no prior knowledge of the environment ' s dynamics, yet can still attain optimal behavior.
Although a model is required, the model need only generate sample transitions.Monte Carlo methods only for episodic tasks.
Monte Carlo methods are ways of solving the reinforcement learning problem based on averaging sample returns. Each of these ideas taken from DP is extended to the Monte Carlo case in which only sample experience is available.

### 2.3.1 Monte Carlo Policy Evaluation

An obvious way to estimate it from experience, then, is simply to average the returns observed after visits to that state.As more returns are observed, the average should converge to the expected value.
Each occurrence of state s in an episode is called a visit to s.By the law of large numbers the sequence of averages of these estimates converges to their expected value.
The every-visit MC method estimates $V\pi(s)$ as the average of the returns following all the visits to s in a set of episodes. Within a given episode, the first time s is visited is called the first visit to s. The first-visit MC method averages just the returns following first visits to s.
Monte Carlo methods are ways of solving the reinforcement learning problem based on averaging sample returns. An important fact about Monte Carlo methods is that the estimates for each state are independent. The estimate for one state does not build upon the estimate of any other state.
That the computational expense of estimating the value of a single state is independent of the number of states.

### 2.3.2 Monte Carlo Estimation of Action Values

If a model is not available, then it is particularly useful to estimate action values rather than state values.
The policy evaluation problem for action values is to estimate $Q\ \pi\ (s, a)$, the expected return when starting in state s, taking action a, and thereafter following policy $\pi$.
That the first step of each episode starts at a state–action pair, and that every such pair has a nonzero probability of being selected as the start.
This guarantees that all state–action pairs will be visited an infinite number of times in the limit of an infinite number of episodes.

### 2.3.3 Monte Carlo Control

The overall idea is to proceed according to generalized policy iteration (GPI).The value function is repeatedly altered to more closely approximate the value function for the current policy, and the policy is repeatedly improved with respect to the current value function.

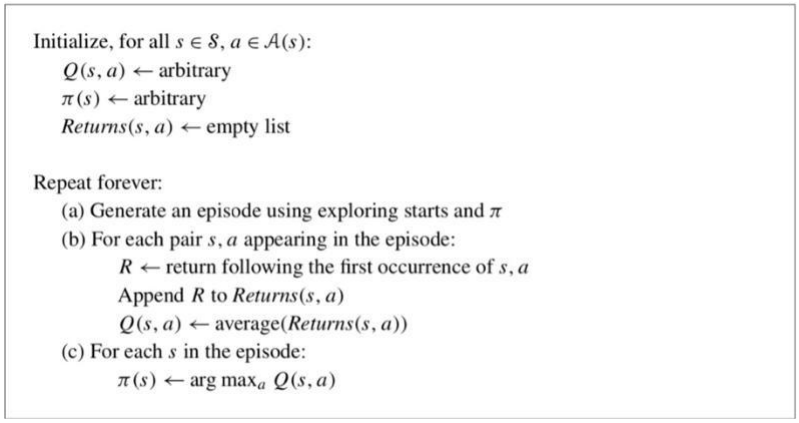


Fig 2.4 Monte Carlo ES: A Monte Carlo control algorithm assuming exploring starts.

Stability is achieved only when both the policy and the value function are optimal.
There are two approaches to ensuring this, resulting in what we call on-policy methods and off-policy methods.
On-policy methods attempt to evaluate or improve the policy that is used to make decisions.

**On-policy:**
In our on-policy method we will move it only to an ε -greedy policy. For any ε -soft policy, π , any ε -greedy policy with respect to Qπ is guaranteed to be better than or equal to π.soft, meaning that π(s, a) > 0 for all s ∈ S and all a ∈ A(s).

**Off-policy:**
In off-policy methods these two functions are separated. The policy used to generate behavior, called the behavior policy, may in fact be unrelated to the policy that is evaluated and improved, called the estimation policy.
They follow the behavior policy while learning about and improving the estimation policy. This technique requires that the behavior policy have a nonzero probability of selecting all actions that might be selected by the estimation policy.
The behavior policy chosen in can be anything, but in order to assure convergence of π to the optimal policy, an infinite number of returns suitable for use in must be obtained for each pair of state and action.

In designing Monte Carlo control methods we have followed the overall schema of generalized policy iteration (GPI) .Maintaining sufficient exploration is an issue in Monte Carlo control methods.
In on-policy methods, the agent commits to always exploring and tries to find the best policy that still explores.
In off-policy methods, the agent also explores, but learns a deterministic optimal policy that may be unrelated to the policy followed. Time complexity of MC is O(N.T^2),where T represents the length of an episodes(number of steps or transitions within that episode).Monte Carlo methods one must wait until the end of an episode, because only then is the return known.

## 2.4 Temporal difference

If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be temporal-difference (TD) learning.
TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap).
TD, and Monte Carlo methods all use some variation of generalized policy iteration (GPI). TD learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas.
TD methods learn their estimates in part on the basis of other estimates. They learn a guess from a guess—they bootstrap.
Monte Carlo methods one must wait until the end of an episode, because only then is the return known, whereas with TD methods one need wait only one time step.

### 2.4.1 TD Prediction

Given some experience following a policy π, methods update their estimate V of Vπ.
TD methods need wait only until the next time step. At time t + 1 they immediately form a target and make a useful update using the observed reward rt+1 and the estimate V (st+1). The simplest TD method, known as TD(0).
TD and Monte Carlo updates as sample backups because they involve looking ahead to a sample successor state. Thus, TD methods combine the sampling of Monte Carlo with the bootstrapping of DP.

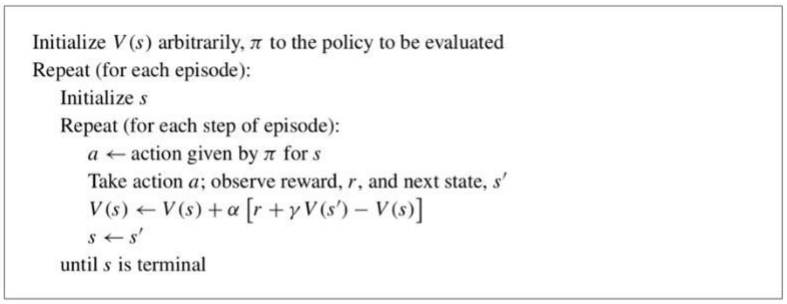


Fig 2.5 TD(0) for estimating V π .

There are exactly the errors between the estimated value in each state and the actual return.
Each error is proportional to the change over time of the prediction, that is, to the temporal difference in prediction.
There are several computational reasons why it is advantageous to learn based on your current predictions rather than waiting until termination when you know the actual return.
Td(0) is based on batch updating because updates are made only after processing each complete batch of training data.
TD(0) always finds the estimates that would be exactly correct for the maximum-likelihood model of the Markov process.
The maximum-likelihood estimate is the model of the Markov process formed in the obvious way from the observed episodes.

### 2.4.2 On-Policy TD Control

As usual, we follow the pattern of generalized policy iteration (GPI) only this time using TD methods for the evaluation or prediction part.
And again approaches fall into two main classes: on-policy and off-policy.
we consider transitions from state–action pair to state–action pair, and learn the value of state–action pairs.
In particular, for an on-policy method we must estimate Q π (s,a) for the current behavior policy π and for all states s and actions a.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[ r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right].$$

Fig 2.6 State-action values updating.

This update is done after every transition from a nonterminal state St. If St+1 is terminal, then Q(st+1, at+1) is defined as zero.

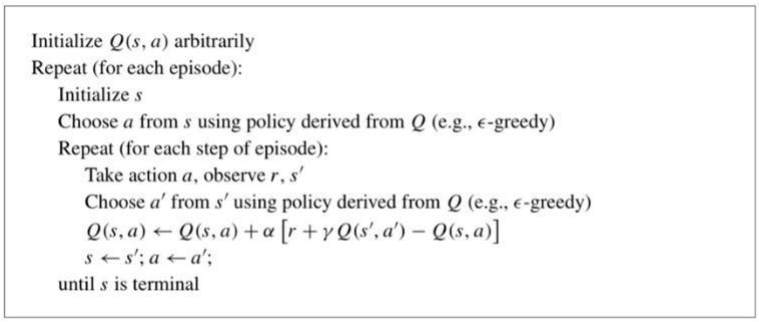


Fig 2.7 SARSA on-policy TD control algorithm.

This rule uses every element of the quintuple of events, (st, at, rt+1, st+1, at+1), that make up a transition from one state– action pair to the next.
This quintuple gives rise to the name **Sarsa** for the algorithm.
There are several computational reasons why it is advantageous to learn based on your current predictions rather than waiting until termination when you know the actual return.
Td(0) is based on batch updating because updates are made only after processing each complete batch of training data.
TD(0) always finds the estimates that would be exactly correct for the maximum-likelihood model of the Markov process.

# CHAPTER-3
# ALGORITHMS IN
# STOCHASTIC CHESS PIECE MOVEMENT USING DQN

## 3.1 Q-Learning: Off-Policy TD Control:

One of the most important breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as Q-learning (Watkins, 1989).

The algorithm is explained below:
The learned action-value function, Q, directly approximates Q ∗ , the optimal action-value function, independent of the policy being followed.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[ r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right].$$

Fig 3.1 Q-values off-policy updating.

1. The learned action-value function, Q, directly approximates Q ∗ , the optimal action-value function, independent of the policy being followed.
2. The policy still has an effect in that it determines which state–action pairs are visited and updated.
3. However, all that is required for correct convergence is that all pairs continue to be updated.
4. Under this assumption and a variant of the usual stochastic approximation conditions on the sequence of step-size parameters, Qt has been shown to converge with probability 1 to Q∗ .

The Q-learning algorithm is shown in procedural form :

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize $s$
  Repeat (for each step of episode):
    Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
    Take action $a$, observe $r$, $s'$
    $Q(s, a) \leftarrow Q(s, a) + \alpha \left[ r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$
    $s \leftarrow s'$;
  until $s$ is terminal

Fig 3.2 Q-learning: An off-policy TD control algorithm.

5. The backup is also from action nodes, maximizing over all those actions possible in the next state.
6. It balances exploration and exploitation to efficiently learn the best actions to take in each state to maximize cumulative rewards.

**Efficiency:**

Q-learning relies on a Q-table to store the values for all state-action pairs. As the state and action spaces grow, the table becomes prohibitively large, requiring significant memory and computational resources.Q-learning relies on a Q-table to store the values for all state-action pairs. As the state and action spaces grow, the table becomes prohibitively large, requiring significant memory and computational resources.

## 3.2 Deep Q-Network (DQN) :

DQN is an extension of Q-learning that leverages to handle environments with large or continuous state spaces Developed by DeepMind, DQN combines Q-learning with deep neural networks, enabling the algorithm to approximate the Q-values for high-dimensional input spaces.

### Key Concepts of DQN:

1. Q-Value Function Approximation:

In traditional Q-learning, the Q-values are stored in a table, which becomes impractical for large state spaces. DQN uses a deep neural network to approximate the Q-value function Q(s,a;θ)Q(s,a;θ), where θ are the parameters (weights) of the neural network.

2. Experience Replay:

Instead of learning from each experience tuple (s,a,r,s′) sequentially, DQN stores them in a replay buffer (or experience replay memory). During training, mini-batches of experiences are randomly sampled from the buffer. This breaks the correlation between consecutive experiences and leads to more stable learning.

3. Target Network:

DQN maintains a separate target network with parameters θ − , which are copied from the main Q-network θ periodically. The target network is used to compute the target Q-value in the Bellman equation, reducing oscillations and divergence in the training process.

4. Updating the Q-Network:

The Q-network is updated using gradient descent to minimize the loss function

$$L(\theta) = \mathbb{E}_{(s,a,r,s')\sim D}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)\right)^2\right]$$

Fig 3.3 Updated gradient descent.

The algorithm is explained below:

1. Initialize the Q-network with random weights θ.Initialize the target network with the same weights θ−← ←θ.Initialize the replay buffer D.
2. Interaction with the Environment:For each episode:Initialize the starting state s Repeat for each step of the episode:With probability ϵ, select a random action a (exploration).
3. Execute action a and observe reward r and next state s′.
4. Store the experience tuple (s,a,r,s ′ ) in the replay buffer D.
5. Sample a mini-batch of experiences.
6. Compute the target Q-value for each sampled experience.
7. Update the Q-network by minimizing the loss.
8. Periodically update the target network.

# CHAPTER-4

# COMPONENTS

## 4.1 Software required

### 1.Python 3.10.9:

Python 3.10.9 is a version of the Python programming language, part of the Python 3 series. It includes improvements, bug fixes, and security enhancements compared to previous versions. Key features of Python 3.10.9 include enhanced type hinting

capabilities, the introduction of structural pattern matching with the match statement, performance optimizations, and updates to the standard library. This version also addresses known bugs and security vulnerabilities present in earlier releases, ensuring a stable and reliable programming environment. Python 3.10.9 maintains backward compatibility with previous Python 3.x versions, allowing developers to upgrade and leverage the latest features while ensuring compatibility with existing code and libraries. Overall, Python 3.10.9 is arecommended update for users looking to benefit from the latest improvements andenhancements in the Python language.

# CHAPTER-5
# IMPLEMENTATION

## 5.1 Process flow

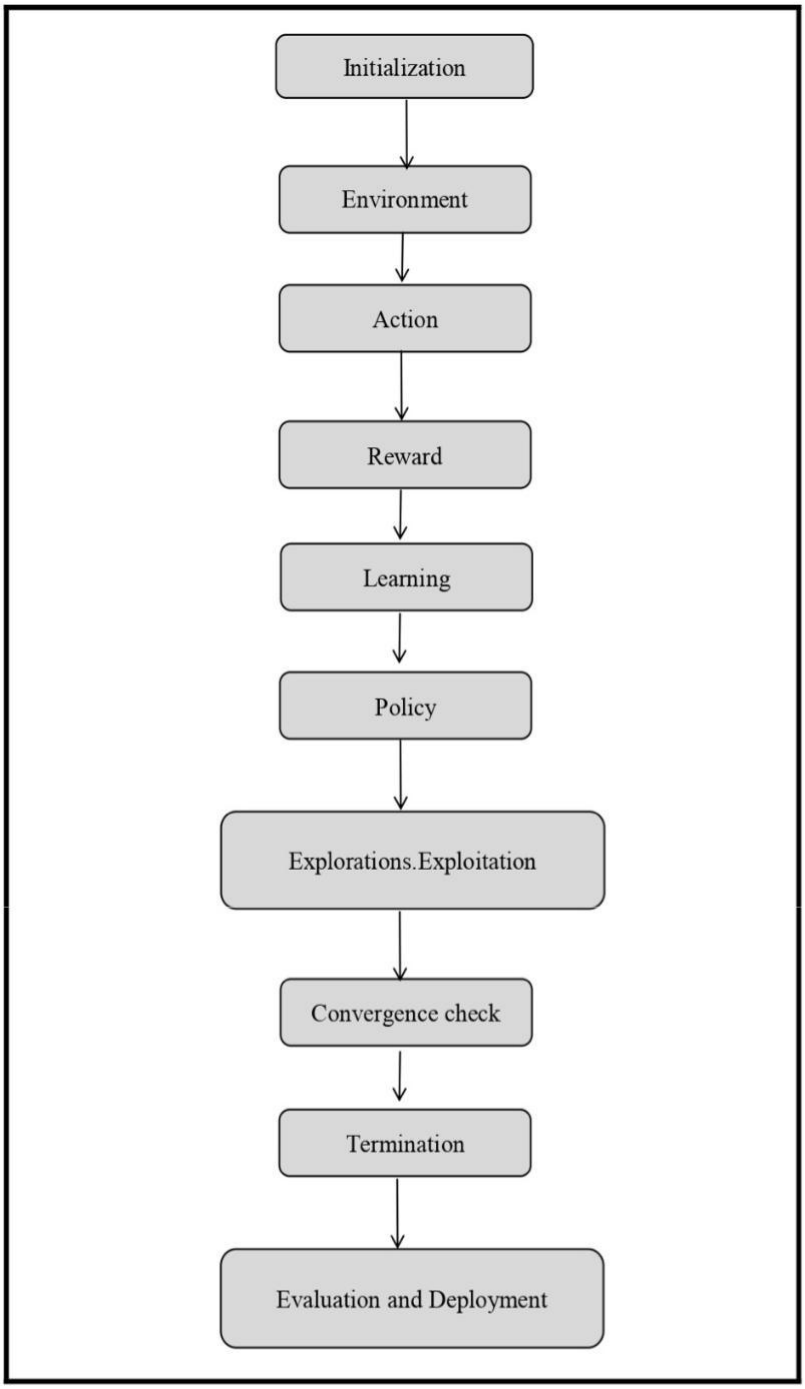


Fig 5.1 Process flow

## 5.2 Chess piece movement using DQN

```
import chess
import numpy as np
import random
from collections import deque
import torch
import torch.nn as nn
import torch.optim as optim

# Neural Network for DQN class
DQNNetwork(nn.Module):
    def __init__(self): super(DQNNetwork,
        self).__init__()
        self.fc1 = nn.Linear(64, 128) # Three fully connected layers fc1 64 neurons self.fc2 =
        nn.Linear(128, 128) # fc2 and fc3 two hidden layers 128 neurons
        self.fc3 = nn.Linear(128, 4672)  # 4672 is an over-approximation of all possible chess moves

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return self.fc3(x)

# Replay Buffer used store and manage experience class
ReplayBuffer:
    def __init__(self, capacity): self.buffer =
        deque(maxlen=capacity)

    def push(self, state, action, reward, next_state, done):
        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        state, action, reward, next_state, done = zip(*random.sample(self.buffer, batch_size))
        return state, action, reward, next_state, done

    def __len__(self):
        return len(self.buffer)

# DQN Algorithm two neural network model and training_model class DQN:
    def __init__(self, state_dim, action_dim, lr=0.001, gamma=0.99, epsilon_start=1.0, epsilon_end=0.01,
epsilon_decay=0.995, buffer_size=10000, batch_size=64):
        self.state_dim = state_dim self.action_dim =
        action_dim self.gamma = gamma self.epsilon
        = epsilon_start self.epsilon_end = epsilon_end
        self.epsilon_decay = epsilon_decay
        self.batch_size = batch_size
```

```
        self.model = DQNNetwork() # model for selecting actions
        self.target_model = DQNNetwork() # target_model computing target Q-values self.optimizer
        = optim.Adam(self.model.parameters(), lr=lr) self.replay_buffer = ReplayBuffer(buffer_size)
        self.update_target_network()


    def update_target_network(self):
        self.target_model.load_state_dict(self.model.state_dict())

    def select_action(self, state,board):
        if random.random() < self.epsilon:
            legal_moves = list(board.legal_moves)
            return random.choice(legal_moves).uci() # Select and return UCI of a random legal move else:
            with torch.no_grad():
                state = torch.tensor(state, dtype=torch.float32).unsqueeze(0)
                q_values = self.model(state)
                return q_values.argmax().item()

    def train(self):
        if len(self.replay_buffer) < self.batch_size:
            return

        state, action, reward, next_state, done = self.replay_buffer.sample(self.batch_size)

        state = torch.tensor(state, dtype=torch.float32)
        action = torch.tensor(action, dtype=torch.int64)
        reward = torch.tensor(reward, dtype=torch.float32)
        next_state = torch.tensor(next_state, dtype=torch.float32)
        done = torch.tensor(done, dtype=torch.float32)

        q_values = self.model(state).gather(1, action.unsqueeze(1)).squeeze(1)
        next_q_values = self.target_model(next_state).max(1)[0]
        target_q_values = reward + (1 - done) * self.gamma * next_q_values

        loss = nn.MSELoss()(q_values, target_q_values)

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        if self.epsilon > self.epsilon_end:
            self.epsilon *= self.epsilon_decay

# Function to get a valid move from the user in UCI format def
get_user_move_uci(board):
    while True: try:
            user_input = input("Enter your move in UCI format (e.g., e2e4, g1f3): ").strip() move =
            chess.Move.from_uci(user_input)
            if move in board.legal_moves: return move
            else:
```

```
            print("Illegal move, please try again.")
        except ValueError:
            print("Invalid move format, please try again.")

# Function to display the board def
display_board(board):
    print(board)

# Main function to handle the game loop def
play_game():
    board = chess.Board()
    dqn = DQN(state_dim=64, action_dim=4672)

    while not board.is_game_over():
        display_board(board)

        # Get move from user for White print("White's
        turn:")
        user_move = get_user_move_uci(board)
        board.push(user_move)

        if board.is_game_over():
            break
        if board.is_check():
            print("Black is in CHECK.")

        display_board(board)

        # Generate and make a DQN move for Black print("Black's
        turn:")
        state = np.array(board.board_fen()).reshape(-1) # Simplified representation black_move =
        dqn.select_action(state,board)
        print(f"Black plays: {black_move}")
        board.push(chess.Move.from_uci(black_move))

        if board.is_game_over():
            break
        if board.is_check():
            print("White is in CHECK.")

        # DQN Training Step
        next_state = np.array(board.board_fen()).reshape(-1)
        reward = 0 # Define a reward function here
        done = board.is_game_over()
        dqn.replay_buffer.push(state, black_move, reward, next_state, done)
        dqn.train()

        if done:
            dqn.update_target_network()

    # Display the final board and result
    display_board(board)
    if board.is_checkmate():
        print("CHECKMATE!")
```

```
    elif board.is_stalemate():
        print("Stalemate.")
    elif board.is_insufficient_material():
        print("Draw due to insufficient material.")
    elif board.is_seventyfive_moves():
        print("Draw due to seventy-five-move rule.")
    elif board.is_fivefold_repetition():
        print("Draw due to fivefold repetition.")
    elif board.is_variant_draw():
        print("Draw due to variant-specific rules.")
    else:
        print("Game over!")
    print(f"Result: {board.result()}")

# Run the game play_game()
```

## OBSERVATION/ RESULTS :

```
Enter your move in UCI format (e.g., e2e4, g1f3): f3f7
r n b q k . n r
. p p p p Q b p
. . . . . . . .
p . . . . . p .
. . B . P . . .
. . . . . . . .
P P P P . P P P
R N B . K . N R
CHECKMATE!
Result: 1-0
```

## 5.2 Challenges

Balancing the need to explore new actions to find potentially better strategies (exploration) and using known actions that yield high rewards (exploitation) is a fundamental challenge.
The current action space (4672 moves) Handling this large action space efficiently while ensuring only legal moves are considered is a significant challenge.
Techniques such as target network updates, experience replay, and appropriate loss functions need to be carefully implemented and tuned to ensure stable and efficient training.
Continuous evaluation against standard chess engines and human players is needed to track progress and identify areas for improvement.

# CHAPTER-6

# CONCLUSION

This project presents a foundational implementation of a chess-playing AI using Deep Q-Learning (DQN). The AI leverages a neural network to approximate the Q-values for different board states and potential moves, while a replay buffer and DQN algorithm facilitate continuous learning and improvement. Through interactive gameplay, the AI competes against a human player, learning from experiences and refining its strategies in real-time.

Experimental results indicate that the DQN-based approach can successfully learn and execute sophisticated piece movements, providing a foundation for more advanced strategic play.

Ultimately, this project demonstrates the feasibility of applying deep reinforcement learning to chess, providing a basis for future research and development. By incorporating the outlined future directions, the AI can evolve into a highly competitive and sophisticated system, contributing valuable insights to the field of AI and game playing.

# FUTURE SCOPE

Future improvements can focus on advanced neural network architectures, sophisticated state representations, and refined reward functions. Additionally, exploring more effective exploration strategies, integrating transfer learning, and leveraging self-play can significantly boost the AI's performance. Implement convolutional neural networks (CNNs) to better capture spatial relationships on the chessboard.Incorporate techniques to handle special moves like castling, promotion, and en passant more effectively.

By addressing these areas, the AI can evolve into a more competitive and intelligent system, capable of playing at higher levels and offering a richer user experience.

# BIBLIOGRAPHY


1. Reinforcement learning - Richard S. Sutton
2.YouTube-://youtube.com/playlist?list=PLqYmG7hTraZDM-OYHWgPebj2MfCFzFObQ&si=3oNk8qinONoWdUTE
3.ChatGPT-3.5 https://chat.openai.com/
4.Medium https://medium.com/
5.Data Analytics Using Python -Bharti Motwani
6..Higher Engineering Mathematics-Hk-das