Reinforcement Learning

S Sabarinath 24B1052

July 2025

Contents

1 Introduction

What is Reinforcement Learning(RL)?

RL(includes DRL) is a ML approach to AI focused on building programs that learn to solve complex problems, which requires intelligence. Unlike other methods, DRL learns through trial and error from feedback that is simultaneously sequential, evaluative, and sampled using a powerful nonlinear function approximation.

AI and ML

AI is a part of Computer programs which demonstrate intelligence.All computer program display intelligence come under AI.ML is a part of AI which can learn from the data. ML has three different divisions:Supervised,Unsupervised And Reinforcement Learning. In this report, we will focus on RL.Deep Learning(DL) is a part of Ml which use multiple layers of non linear Function approx..DRL is simply use of DL in RL tasks.

2 DRL

DRL

At its Core,DRL is about Complex sequential decision problems under certainity.There are two components in DRL:

- Agent
- Environment

Components

Agent

In DRl, we use computer programs that solve complex decision making problems.In DRL, these computer programs are called agents.It is the decision maker.Everything that comes after the decisions get bundled into the environment. For example, while playing chess with an opponent,agent is only your brain. Agents have internal components and process of their own.There is a 3-step process:*interaction*,*evaluate*,*improve*. Interaction gather information for learning.then there is evaluation where they evaluate their current behavior.Then they improve themselves to make the overall performance better.

Environment

Environment is everything outside agent,which agent has no control over.In case of the example of chess player can change his respective pieces rest he has no control over.A way to express decision making problems in Rl is by modeling the problem by mathematical framework known as Markov Decision Process(MDP).

1. **State Space(S_i)**: The environment is represented by a set of variables related to the problem. This set of variables and all the possible values that they can take are referred to as the state space. A state is an instantiation of the state space, a set of values the variables take.It can be finite/infinite.It is set of sets the inner set should have same size.
2. **Observation**: sometimes agent doesn't have full access they can only access part of it.The set of variables the agent perceives at any given time is *observation*
3. **Action Space(A_i)**:The set of actions agent can choose form.The set of all actions in all states is *Action space.*
4. **Transition function**:The function responsible for the transition made by action chosen by agent is *transition function.*

5. **Reward function(R_i)**: After transition,the environment may provide a reward signal as response.The function responsible for this mapping is *Reward function*.

the set of transition and reward function is known as the *model* of the environment.

Agent's Learning process

Trial and error

Agent learns through several cycles. each cycle is known as *Time Step*.Each time step agent observes the environment and takes the actions and receives a new observation and reward.The set of tuple of state,action,reward and new state is known a experience.It may take many time steps to learn and observe the process properly.

Sequential Feedback

The results of the agent's actions aren't just immediate they can affect what happens later too. That means the agent has to think ahead. It's a bit like chess: one move might not seem great now but could lead to a win later.

Evaluative Feedback

The agent isn't told the "right" thing to do. Instead, it gets feedback like this action gave you a good reward or not. It learns by judging its past actions based on how well they turned out, not by being directly used.

Sampled Feedback

The agent only learns from the actions it actually tries not all the possible actions. So, to really learn what works best, it needs to explore different choices. It's like trying different strategies in a game to see what leads to winning.

Agent Environment Interaction cycle

An agent interacts with the environment in cycles called **time steps**, where it observes the state, takes an action, and receives a reward and new state forming an *experience tuple*. Tasks can be:

- **Episodic**: Have a natural ending (e.g., a game).
- **Continuing**: No clear end.

A full sequence of time steps in an episodic task is called an **episode**, and the total accumulated reward is called the **return**.
Agents learn from rewards and also:

- Learn **policies** (mapping from states to actions),
- Learn **models** (predicting next states/rewards),
- Learn **value functions** (estimating returns).

3 MDPs: The Engine of the environment

Markov decision processes formally describe an environment for RL.where environment is fully observable.Almost all reinforcement learning (RL) problems can be formalized as MDP. Examples include:

- **Optimal control** primarily deals with *continuous MDPs*.
- **Partially observable** problems can be converted into MDPs by considering belief states.

- **Bandits** are a special case of MDPs with only one state.

later,we will use algorithms for planning MDPs.One assumption we use is distribution is stationary.

Markov Property

The **Markov Property** means that the next state depends only on the current state and the action taken, not on any of the previous states or actions.
Mathematically, if S_t is the current state and A_t is the current action, then:

$$P(S_{t+1} \mid S_t, A_t) = P(S_{t+1} \mid S_t, A_t, S_{t-1}, A_{t-1}, \ldots, S_0, A_0) \tag{1}$$

This equation (1) implies that the system's future behavior depends only on the current situation, not on how it got there. Such a system is called a *Markov Decision Process (MDP)*.A RL task which satisfies Markov Property is called as MDP.

Actions

There is a set of states for the MDP.The initial state is chosen from this set.And there is an unique state called as *terminal state*.In which all actions executed return to itself(terminal state) with probability 1,with 0 reward.In Markov Decision Processes (MDPs), the set of available actions depends on the state and is defined by a function A_t. This set can be constant or state-dependent.

- The action space may be finite or infinite, but each action must consist of a finite number of variables.
- Most environments use the same number of actions across states.
- Agents select actions either **deterministically** or **stochastically** (from a probability distribution).

Transition Function

The **transition function** in a MDP defines the probability of transitioning from one state to another given an action. It is denoted as:

Transtion Function

$$p(s'|s,a) = P(S_t = s' \mid S_{t-1} = s, A_{t-1} = a) \tag{2}$$

This represents the probability of ending up in state s' at time t, given that the agent was in state s and took action a at time $t-1$.
Since these are probabilities, they must sum to 1 over all possible next states s':

$$\sum_{s' \in S} p(s'|s,a) = 1, \quad \forall s \in S, \forall a \in A(s) \tag{3}$$

Reward Signal

The Reward function maps the tuple s',a,s to a scalar.This tells us the how good is the transition.

Reward Function

$$r(s,a) = \mathbf{E}[\mathbf{R_t}|\mathbf{S_{t-1}} = \mathbf{s}, \mathbf{A_{t-1}} = \mathbf{a}] \tag{4}$$

The reward function above is defined as the expected value of the reward received at time step t, given that the agent was in state s at time $t-1$, took action a, and transitioned to state s' at time t. This captures the dynamics of the environment's response to the agent's interaction.

$$r(s,a,s') = \mathbb{E}\left[R_t \mid S_{t-1} = s,\, A_{t-1} = a,\, S_t = s'\right], \quad R_t \in \mathcal{R} \subset \mathbb{R} \tag{5}$$

Here, $r(s,a,s')$ represents the expected immediate reward for a transition from state s to s' by taking action a. The reward R_t is a real-valued random variable drawn from the reward space $\mathcal{R}$, which is a subset of the real numbers $\mathbb{R}$.

Returns

In infinite-horizon MDPs, future rewards are discounted using a factor $\gamma \in [0,1]$, known as the **discount factor**, to reflect uncertainty and reduce variance.

- A smaller γ values immediate rewards more.
- A larger γ values future rewards more.
- $\gamma = 1$: no discount; all future rewards are equally valued.

Return and Discount

The discounted return from time step t is:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{6}$$

$$G_t = R_{t+1} + \gamma G_{t+1} \tag{7}$$

Above equation is recursive.

Several extensions of the standard MDP framework exist to model more complex environments:

Extension of MDPs

- **POMDP** (Partially Observable MDP): Agent cannot fully observe the state.
- **FMDP** (Factored MDP): Compact representation of transitions and rewards.
- **Continuous MDP**: State, action, or time are continuous.
- **RMDP** (Relational MDP): Combines probabilistic and relational knowledge.
- **SMDP** (Semi-MDP): Supports temporally extended actions.
- **MMDP** (Multi-agent MDP): Multiple agents in the same environment.
- **Dec-MDP** (Decentralized MDP): Agents collaborate to maximize a common reward.

In most of the real world applications we have POMDPs.

Policy

In RL, a **policy** π defines the agent's behavior by mapping states to actions. Policies can be:

- **Deterministic**: Assigns a specific action to each state.
- **Stochastic**: Defines a probability distribution over actions for each state.

Policy

A policy π is a distribution over actions given states, defined as:

$$\pi(a \mid s) = \mathbb{P}[A_t = a \mid S_t = s]$$

where $\pi(a \mid s)$ denotes the probability of taking action a when in state s.

The agent uses policies to act in the environment. Evaluating a policy's effectiveness allows comparison: How good is the policy? How does it compare to another policy?

Policy evaluation helps choose better strategies for achieving goals. Using this agent evalutes its behaviour and improve. we also need another function called State-Value function.

State-Value function

The **State-Value function or Value Function** $v_\pi(s)$ represents the expected return when starting from state s and following policy π.It tells how good the state under the given policy. **Key aspects:**

- Considers all future rewards.
- Depends on transition probabilities and rewards.
- Uses a discount factor $\gamma \in [0, 1]$ to reduce the weight of future rewards.

State-Value Function

$$\mathbf{v}_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{8}$$

The above equation is the expectation of return when policy π followed from state s.It can also be written in recursive form as

$$\mathbf{v}_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \tag{9}$$

Bellman Expectation Equation

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} p(s'|s, a) \left[r(s, a, s') + \gamma v_\pi(s')\right] \tag{10}$$

This equation provides a recursive way to compute the value of a state under policy π.

Action-Value and Advantage Functions

The action-value function $q_\pi(s, a)$ represents the expected return when:

- starting from state s,
- taking action a,
- and then following policy π.

Action-Value Function

$$q_\pi(s,a) = \mathbb{E}_\pi\left[G_t \mid S_t = s, A_t = a\right] \tag{11}$$

Bellman Expectation Equation

$$q_\pi(s,a) = \sum_{s'} p(s' \mid s,a)\left[r(s,a,s') + \gamma \sum_{a'} \pi(a' \mid s') q_\pi(s',a')\right]$$

Advantage Function

The advantage function $A_\pi(s,a)$ measures how much better or worse it is to take action a in state s, compared to the average value of state s under policy π:

$$A_\pi(s,a) = q_\pi(s,a) - v_\pi(s)$$

Optimality

Till now we introduced some functions which we will use to describe,evaluate and improve.This is called optimality when these function are at their best.

An *optimal* policy is the policy such that every state can obtain expected returns better than any other policy could get. that is any above defined function are at their maximum in this policy.There can be more than one optimal policy but there can be only one optimal *state value* function and optimal *action-value* function and optimal *advantage* function.optimal functions are given by Bellman optimality equations.

Optimality Equations

The Bellman Optimality Equations define the value of a state or state-action pair under the optimal policy:

Optimal state-value function: $\mathrm{v}_*(s) = \max_\pi v_\pi(s), \quad \forall s \in \mathbb{S}$

Optimal action-value function: $\mathrm{q}_*(s,a) = \max_\pi q_\pi(s,a), \quad \forall s \in \mathbb{S}, \forall a \in \mathbb{A}(s)$

Bellman equation for $\mathrm{v}_*(s):$ $\quad v_*(s) = \max_a \sum_{s',r} p(s',r \mid s,a)\left[r + \gamma v_*(s')\right]$

Bellman equation for $\mathrm{q}_*(s,a):$ $\quad q_*(s,a) = \sum_{s',r} p(s',r \mid s,a)\left[r + \gamma \max_{a'} q_*(s',a')\right]$

4 Dynamic Programming (DP)

Now that we've discussed the RL problem formulation, and we've defined the objective we are after, we can start exploring methods for finding this objective. Iteratively computing the equations presented in the previous section is one of the most com-mon ways to solve a reinforcement learning problem and obtain optimal policies when the dynamics of the environment, the MDPs, are known. Let's look at the methods.

Policy Evaluation

To evalute a random policy we use an algorithm to check whether it is better than a policy is known as policy Evaluation.
The policy-evaluation algorithm consists of the iterative approximation of the state-value function of the policy under evaluation. The algorithm converges as k approaches infinity.

The policy-evaluation

1. Initialize $v_0(s)$ for all s arbitrarily, and to 0 if s is terminal. Then, increase k and iteratively improve the estimates using the equation below:

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma v_k(s')\right]$$

2. Calculate the value of a state s as the weighted sum of the reward and the discounted estimated value of the next state s'.

The function below implements iterative policy evaluation.Given a fixed policy π and environment P, it computes the value function $V(s)$ for all states s, using the Bellman expectation equation.

Policy Evaluation Code

```python
def policy_evaluation(pi, P, gamma=1.0, theta=1e-10):
   prev_V = np.zeros(len(P), dtype=np.float64)
   while True:
      V = np.zeros(len(P), dtype=np.float64)
      for s in range(len(P)):
         for prob, next_state, reward, done in P[s][pi(s)]:
            V[s] += prob * (reward + gamma * prev_V[next_state] * (not done))
      if np.max(np.abs(prev_V - V)) < theta:
         break
      prev_V = V.copy()
   return V
```

Parameters:

- `pi`: The policy π, a function mapping states s to actions a.
- `P`: The environment, typically a dictionary mapping states and actions to a list of possible transitions, each defined by $(\text{prob}, \text{next\_state}, \text{reward}, \text{done})$.
- `gamma`: Discount factor $\gamma \in [0, 1]$.
- `theta`: Convergence threshold. Iteration stops when value changes are less than this.

Returns: The estimated value function $V(s)$ for all states under policy π.

Policy improvement

Here we will improve policies based on action value function values obtained from policy evaluation and the MDP.Then greedily selecting the best function value this is known as policy improvement

The policy-improvement

1. We obtain a new policy π' by taking the highest-valued action.

$$\pi'(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma v_\pi(s')\right]$$

2. Notice that this uses the action with the highest-valued Q-function.

The below code takes in state-value function of the policy to improve V and the P(MDP).

Policy Improvement Code

```python
def policy_improvement(V, P, gamma=1.0):
   Q = np.zeros((len(P), len(P[0])), dtype=np.float64)
   for s in range(len(P)):
      for a in range(len(P[s])):
         for prob, next_state, reward, done in P[s][a]:
            Q[s][a] += prob * (reward + gamma * V[next_state] * (not done))
   new_pi = lambda s: {s:a for s, a in enumerate(np.argmax(Q, axis=1))}[s]
   return new_pi
```

Policy Iteration

Now we got two functions,Now we evalute all the policies state-value function and improve it till we get the best policy.

Policy Iteration Code

```python
def policy_iteration(P,theta=1e-10):
   random_actions=np.random.choice(tuple(P[0].keys()),len(P))
   pi=(lambda s:{s:a for s,a in enumerate(random_actions)}[s])
   while true;
      old={s:pi(s) for s in range(len(P))}
      V=policy_iteration(pi,P)
      pi=policy_improvement(V,P)
      if old_pi == {s:pi(s) for s in range(len(P))}:
         break
   return V,pi
```

Regardless where you start(at which policy or state) from ,you will get the optimal policy .

Value Iteration

As the above functions iterate through all policies which will take time.But if we truncate policy evaluation after one iteration we could still find the improved policy by taking the greedy one this algorithm is known as *Value Iteration*.

Value Iteration Equation

$$v_{k+1}(s) = \max_a \sum_a \pi(a|s) \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma v_k(s')\right]$$

The above equation is combined truncated policy-evaluation step and a policy improvement.

Policy Evaluation Code

```
def value_iteration(P,theta=1e-10,gamma=1.0):
   V=np.zeros(len(P),dtype=np.float64)
   while True:
      Q=np.zeros((len(P),len(P[0])),dtype=np.float64)
      for s in range(len(P)):
         for a in range(len(P[s]):
            for prob,next_state,reward,done in P[s][a]:
               Q[s][a]+=prob*(rward+gamma*V[next_state]*(not done))
      if(np.max(np.abs(V-np.max(Q,axis=1))))<theta:
         break
      V=np.max(Q,axis=1)
   pi=lamda s:{a:a for s,a in enumerate(np.argmax(np.argmax(Q,axis=1))}[s]
   return V,pi
```

5 Exploration And Exploitation

In real-world RL, the agent does **not know in advance** how the environment reacts to actions. Unlike planning methods that assume a known model (e.g., MDPs), RL agents must learn by **trial and error**—interacting with the environment directly.

A key decision-making challenge arises: **What action should the agent take next?**

- **Exploit:** Choose the best-known action so far based on current knowledge.
- **Explore:** Try new actions to gather more information.

This is known as the **exploration-exploitation trade-off**. Balancing the two is essential:

- **Exploration** builds the knowledge needed for learning.
- **Exploitation** uses that knowledge to make optimal decisions.

Exploration builds knowledge; exploitation maximizes reward. The agent must strike a balance between both.

6 Multi-armed Bandits(MAB)

MAB are special case of RL problem where state spcae and horizon equal one.It has multiple actions and single state and greedy horizon.

Functions for MABs

MABs are MDPs with single nonterminal state and single time step per episode.

$$q(a) = \mathbf{E}[R_t | A_t = a] \tag{12}$$

$$\mathbf{v}_* = q(a_*) = \max_{a \in \mathbf{A}} q(a) \tag{13}$$

Regret

The main goal is to maximize the expected reward.As there single chance of selecting one action on each episode.Comparing agents based only on final rewards can be misleading. One might act randomly until the end, while another quickly finds the best action—but both may finish with the same score.A better way to compare them is by using **total regret**—how much reward they missed out on compared to always taking the best action. Lower regret means the agent explored better and learned faster. While the agent doesn't need to know the MDP, we do need it to measure how well it did.

Total Regret Equation

$$\mathbf{T} = \sum_{e=1}^{E} \mathbb{E}[v_* - q_*(A_e)] \tag{14}$$

Here T is the total Regret.

Approaches to Solving MAB Environments

There are three major approaches to solving Multi-Armed Bandit (MAB) problems:

1. **Random Exploration Strategies**: Introduce randomness in action selection, e.g., ϵ-greedy, where the agent mostly exploits but explores with probability ϵ.
2. **Optimistic Exploration Strategies**: Assume unvisited states are optimal, encouraging exploration. As more data is gathered, estimates converge to true values. This systematic approach is better.
3. **Information State-Space Exploration Strategies**: Encode the uncertainty as part of the environment's state, allowing the agent to distinguish between explored and unexplored states. This increases the state space and its complexity but provides a more nuanced exploration.

Q-function Estimation: For MABs, the Q-function for an action a is the average reward:

$$Q(a) = \frac{\text{Total reward from action } a}{\text{Number of times action } a \text{ is selected}}.$$

Greedy:Always Exploit

The Greedy approach always select the action with highest estimated value.

Greedy Algorithm

```python
def pure_exploitation(env,n1=5000):
   Q=np.zeros((env.action_space.n))
   N=np.zeros((env.action_space.n))
   Qe=np.empty((n1, env.action_space.n))
   returns = np.empty(n1)
   actions = np.empty(n1)
   for e in tqdm(range(n1),desc='Episodes for:'+name,leave=False):
      action=np.argmax(Q)
      a,reward,b,c=env.step(action)
      N[action]+=1
      q=Q[action]
      Q[action]=q+(reward-q)/N[action]
      Qe[e]=Q
      returns[e]=reward
      actions[e]=action
   return returns,Qe,actions
```

Random action:Always explore

This selects action randomly from the action space.sole reason is to gather information.

Random Algorithm

```python
def pure_exploration(env,n1=5000):
   Q=np.zeros((env.action_space.n))
   N=np.zeros((env.action_space.n))
   Qe=np.empty((n1, env.action_space.n))
   returns = np.empty(n1)
   actions = np.empty(n1)
   for e in tqdm(range(n1),desc='Episodes for:'+name,leave=False):
      action=np.random.randint(len(Q))
      a,reward,b,c=env.step(action)
      N[action]+=1
      q=Q[action]
      Q[action]=q+(reward-q)/N[action]
      Qe[e]=Q
      returns[e]=reward
      actions[e]=action
   return returns,Qe,actions
```

Epsilon Greedy

The Greedy approach always selects the action with highest estimated value sometimes and randomly other action otherwise.It is both exploring and Exploiting.As in greedy you keep thinking the max one is good but here it changes.

Epsilon Greedy Code

```
def epsilon_greedy(env,epsilon=5000,n1=5000):
   Q=np.zeros((env.action_space.n))
   N=np.zeros((env.action_space.n))
   Qe=np.empty((n1, env.action_space.n))
   returns = np.empty(n1)
   actions = np.empty(n1)
   for e in tqdm(range(n1),desc='Episodes for:'+name,leave=False):
      if(np.random.random() > epsilon):
         action=np.argmax(Q)
      else:
         action=np.random.randint(len(Q))
      a,reward,b,c=env.step(action)
      N[action]+=1
      q=Q[action]
      Q[action]=q+(reward-q)/N[action]
      Qe[e]=Q
      returns[e]=reward
      actions[e]=action
   return returns,Qe,actions
```

Decaying epsilon-greedy

Initially, the agent explores more due to limited experience. As it learns, it should exploit more. A common approach is the decaying epsilon-greedy strategy—start with a high epsilon (1) and reduce it over time. This helps balance exploration and exploitation. There two methods given below:

Linearly decaying epsilon-greedy strategy

```
def lin_decep(env,n1=5000,init_epsilon=1.0,min_epsilon=0.01,decay_ratio=0.05):
   Q=np.zeros((env.action_space.n))
   N=np.zeros((env.action_space.n))
   Qe=np.empty((n1, env.action_space.n))
   returns = np.empty(n1)
   actions = np.empty(n1)
   for e in tqdm(range(n1),desc='Episodes for:'+name,leave=False):
      de_epsiodes=n1*decay_ratio
      epsilon = 1 - e / de_episodes
      epsilon *= init_epsilon - min_epsilon
      epsilon += min_epsilon
      epsilon = np.clip(epsilon, min_epsilon, init_epsilon)
```

Linearly decaying epsilon-greedy strategy

```
        if(np.random.random() > epsilon):
            action=np.argmax(Q)
        else:
            action=np.random.randint(len(Q))
        a,reward,b,c=env.step(action)
        N[action]+=1
        q=Q[action]
        Q[action]=q+(reward-q)/N[action]
        Qe[e]=Q
        returns[e]=reward
        actions[e]=action
    return returns,Qe,actions
```

Expontentially decaying epsilon-greedy strategy

```
def exp_decep(env,n1=5000,init_epsilon=1.0,min_epsilon=0.01,decay_ratio=0.05):
    Q=np.zeros((env.action_space.n))
    N=np.zeros((env.action_space.n))
    Qe=np.empty((n1, env.action_space.n))
    returns = np.empty(n1)
    actions = np.empty(n1)
    decay_episodes = int(n1 * decay_ratio)
    rem_episodes = n1 - decay_episodes
    epsilons = 0.01
    epsilons /= np.logspace(-2, 0, decay_episodes)
    epsilons *= init_epsilon - min_epsilon
    epsilons += min_epsilon
    epsilons = np.pad(epsilons, (0, rem_episodes), 'edge')

    for e in tqdm(range(n1),desc='Episodes for:'+name,leave=False):
        if(np.random.random() > epsilon):
            action=np.argmax(Q)
        else:
            action=np.random.randint(len(Q))
        a,reward,b,c=env.step(action)
        N[action]+=1
        q=Q[action]
        Q[action]=q+(reward-q)/N[action]
        Qe[e]=Q
        returns[e]=reward
        actions[e]=action
    return returns,Qe,actions
```

Optimistic initialization

A way to balance exploration and exploitation is to assume unexplored actions are very good actions for better results this idea is called optimism in the face of uncertainty. One example is optimistic initialization: start by setting all Q-values high so the agent is tempted to try everything at least once.

Optimistic initialization strategy

```python
def optimistic_initialization(env,
 optimistic_estimate=1.0,
 initial_count=100,n1=5000):
   Q = np.full((env.action_space.n),optimistic_estimat,dtype=np.float64)
   N = np.full((env.action_space.n),optimistic_estimat,dtype=np.float64)
   Qe=np.empty((n1, env.action_space.n))
   returns = np.empty(n1)
   actions = np.empty(n1)
   for e in tqdm(range(n1),desc='Episodes for:'+name,leave=False):
      action=np.argmax(Q)
      a,reward,b,c=env.step(action)
      N[action]+=1
      q=Q[action]
      Q[action]=q+(reward-q)/N[action]
      Qe[e]=Q
      returns[e]=reward
      actions[e]=action
   return returns,Qe,actions
```

The best performing strategy in the experiment five two-armed Bernoulli bandit environments with probabilities α and β initialized uniformly at random, and five seeds is the optimistic with 1.0 initial Q-values and 10 initial counts. All strategies perform pretty well, and these weren't highly tuned, so it's just for the fun of it and nothing else. Head to chapter 4's Notebook and play, have fun.

7 Strategic Exploration

While ε-greedy and similar methods focus on random exploration, strategic methods try to choose actions that yield useful information. We will discuss:

- Softmax Strategy
- Upper bound Confidence(UCB)
- Thompson Sampling

Softmax Strategy

This Method selects a specific action with a probability which depends on Q-function(action value function).In this we add a hyperparameter called temperature represented by τ.As it approaches infinity preference of all actions become same.we use very high and very low positive real numbers and normalize these.

Softmax Exploration Strategy

$$\pi(a) = \exp(Q(a)/\tau) / \sum_{b=0}^{B} \exp(Q(b)/\tau) \tag{15}$$

Softmax Strategy

```
def soft_max(env,
 init_temp=1.0,
 min_temp=0.01,decay_ratio=0.04,n1=5000):
   Q = np.zeros((env.action_space.n), dtype=np.float64)
   N = np.zeros((env.action_space.n), dtype=np.int)
   Qe=np.empty((n1, env.action_space.n))
   returns = np.empty(n1, dtype=np.float64)
   actions = np.empty(n1, dtype=np.int)
   init_temp = min(init_temp,sys.float_info.max)
   min_temp = max(min_temp,np.nextafter(np.float32(0), np.float32(1)))
   for e in tqdm(range(n1),desc='Episodes for:'+name,leave=False):
      decay_episodes = n1 * decay_ratio
      temp = 1 - e / decay_episodes
      temp *= init_temp - min_temp
      temp += min_temp
      temp = np.clip(temp, min_temp, init_temp)

      scaled_Q = Q / temp
      norm_Q = scaled_Q - np.max(scaled_Q)
      exp_Q = np.exp(norm_Q)
      probs = exp_Q / np.sum(exp_Q)
      assert np.isclose(probs.sum(), 1.0)

      action = np.random.choice(np.arange(len(probs)),
                          size=1,
                          p=probs)[0]
      a,reward,b,c=env.step(action)
      N[action]+=1
      q=Q[action]
      Q[action]=q+(reward-q)/N[action]
      Qe[e]=Q
      returns[e]=reward
      actions[e]=action
   return returns,Qe,actions
```

UCB

Select actions with high estimated potential reward and uncertainty.

UCB

UCB balances exploration and exploitation by selecting the action a with the highest upper confidence bound:

$$A_e = \arg\max_a [Q_e(a) + c\sqrt{lne/N_e(a)}] \tag{16}$$

UCB

```python
def UCB(env,c=3,n1=5000):
   Q = np.zeros((env.action_space.n), dtype=np.float64)
   N = np.zeros((env.action_space.n), dtype=np.int)
   Qe=np.empty((n1, env.action_space.n))
   returns = np.empty(n1, dtype=np.float64)
   actions = np.empty(n1, dtype=np.int)
   for e in tqdm(range(n1),desc='Episodes for:'+name,leave=False):
      action = e
      if(e>=len(Q)):
         U=c*np.sqrt(np.log(e)/N)
         action = np.argmax(Q+U)
       a, reward, b, c = env.step(action)
      N[action] += 1
      Q[action] = Q[action] + (reward - Q[action])/N[action]

      Qe[e] = Q
      returns[e] = reward
      actions[e] = action
   return returns,Qe,actions
```

Thompson Sampling

Thompson Sampling is a Bayesian approach. For each action, maintain a distribution over expected rewards and sample from these to choose actions. Commonly, Beta distributions are used for Bernoulli rewards.

Thonpson Sampling

```python
def thompson_sampling(env,
                alpha=1,
                beta=0,
                n_episodes=1000):
   Q = np.zeros((env.action_space.n), dtype=np.float64)
   N = np.zeros((env.action_space.n), dtype=np.int)

   Qe = np.empty((n_episodes, env.action_space.n), dtype=np.float64)
   returns = np.empty(n_episodes, dtype=np.float64)
   actions = np.empty(n_episodes, dtype=np.int)
   for e in tqdm(range(n_episodes),
            desc='Episodes for: ' + name,
            leave=False):
      samples = np.random.normal(
         loc=Q, scale=alpha/(np.sqrt(N) + beta))
      action = np.argmax(samples)
      _, reward, _, _ = env.step(action)
      N[action] += 1
      Q[action] = Q[action] + (reward - Q[action])/N[action]
      Qe[e] = Q
      returns[e] = reward
      actions[e] = action
   return name, returns, Qe, actions
```

8 Policy Value Estimation

In this part we will learn about two important policy Value estimation method.

- Monte Carlo(mc)
- Temporal Difference Learning(TD)

Monte Carlo method

The goal of policy evaluation is to estimate how much total reward an agent can expect by following a given policy. MC methods achieve this by running the policy over multiple episodes, collecting trajectories, and averaging the returns for each state.

Each trajectory is a sequence of experience tuples $(S_t, A_t, R_{t+1}, S_{t+1})$. For every state visited, the return is computed as the sum of discounted rewards from that point to the end of the episode. This return is then averaged over multiple visits to estimate the state value.

MC methods are simple and rely only on complete episodes, making them easy to implement and understand. Some helper functions are given below:

Decay Function

```
def decay_schedule(init_value, min_value, decay_ratio, max_steps, log_start=-2,
    log_base=10):
   decay_steps = int(max_steps * decay_ratio)
   rem_steps = max_steps - decay_steps
   values = np.logspace(log_start, 0, decay_steps, base=log_base, endpoint=True)
       [::-1]
   values = (values - values.min()) / (values.max() - values.min())
   values = (init_value - min_value) * values + min_value
   values = np.pad(values, (0, rem_steps), 'edge')
   return values
```

Generating full trajectories

```
def generate_trajectory(pi, env, max_steps=200):
   done, trajectory = False, []
   while not done:
      state = env.reset()
      for t in count():
         action = pi(state)
         next_state, reward, done, _ = env.step(action)
         experience = (state, action, reward, next_state, done)
         trajectory.append(experience)
         if done:
            break
         if t >= max_steps - 1:
            trajectory = []
            break
         state = next_state
   return np.array(trajectory, np.object)
```

MC Prediction

```
def mc_prediction(pi, env, gamma=1.0,init_alpha=0.5,min_alpha=0.01,
    alpha_decay_ratio=0.5,n_episodes=500, max_steps=200,first_visit=True):
   nS = env.observation_space.n
   discounts = np.logspace(0, max_steps,num=max_steps,base=gamma,endpoint=False)
   alphas = decay_schedule(init_alpha,min_alpha, alpha_decay_ratio,n_episodes)
   V = np.zeros(nS, dtype=np.float64)
   V_track = np.zeros((n_episodes, nS), dtype=np.float64)
   targets = {state:[] for state in range(nS)}
   for e in tqdm(range(n_episodes), leave=False):
      trajectory = generate_trajectory(pi, env,max_steps)
      visited = np.zeros(nS, dtype=np.bool)
      for t, (state, _, reward, _, _) in enumerate(trajectory):
         if visited[state] and first_visit:
            continue
         visited[state] = True
         n_steps = len(trajectory[t:])
         G = np.sum(discounts[:n_steps] trajectory[t:, 2])
         targets[state].append(G)
         mc_error = G - V[state]
         V[state] = V[state] + alphas[e] * mc_error
      V_track[e] = V
   return V.copy(), V_track, targets
```

Temporal Difference Learning

In MC,one must wait until an episode ends to compute returns, making them high-variance and time taking. TD methods address this by using bootstrapping: they update value estimates based on the immediate reward and the estimated value of the next state.

Instead of using the full return, TD uses the target:

Some changes from MC

$$\text{target} = R_{t+1} + \gamma V(S_{t+1}),$$

Now the TD error will change and this also affects the V(s)

$$\mathbf{v}_\pi(s) = \mathbb{E}_\pi[G_{t:T}|S_t = s]$$

$$\mathbf{v}_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma \mathbf{v}_\pi(S_{t+1})|S_t = s]$$

$$TD\_error = R_{t+1} + \gamma \mathbf{V}_t(S_{t+1}) - \mathbf{V}_t(S_t)$$

$$\mathbf{V}_{t+1}(S_t) = V_t(S_t) + \alpha_t[R_{t+1} + \gamma V_t(S_{t+1} - V_t(S_t)]$$

allowing learning from incomplete episodes and enabling faster, lower-variance updates. This makes TD methods more efficient and practical in many reinforcement learning settings.

TD Prediction

```
def td(pi, env, gamma=1.0,init_alpha=0.5,min_alpha=0.01,alpha_decay_ratio=0.5,
    n_episodes=500):
   nS = env.observation_space.n
   V = np.zeros(nS, dtype=np.float64)
   V_track = np.zeros((n_episodes, nS), dtype=np.float64)
   targets = {state:[] for state in range(nS)}
   alphas = decay_schedule(init_alpha, min_alpha,alpha_decay_ratio, n_episodes)
   for e in tqdm(range(n_episodes), leave=False):state, done = env.reset(),
       False
      while not done:
         action = pi(state)
         next_state, reward, done, _ = env.step(action)
         td_target = reward + gamma * V[next_state] * (not done)
         targets[state].append(td_target)
         td_error = td_target - V[state]
         V[state] = V[state] + alphas[e] * td_error
         state = next_state
      V_track[e] = V
   return V, V_track, targets
```

MC methods estimate the value of a state by sampling complete episodes and computing the actual return at the end. These methods are unbiased but tend to have high variance because they rely on the full sampled return.Additionally, MC methods delay learning until the episode ends, which may not be ideal in all environments.

TD methods address the variance issue by updating estimates based on a single step: they use the observed reward and bootstrap from the estimated value of the next state. This results in lower variance and faster updates but introduces bias because the update depends on the current value function estimate.which more than actual value

N-step TD learning

Between MC and TD lies a family of methods called n**-step returns**. Instead of updating after one step (TD) or waiting until the end (MC), n-step methods wait for n steps before bootstrapping. These methods generalize the idea of temporal difference learning:

- For $n = 1$, we recover TD(0).
- For $n = \infty$ (or the episode length), we recover MC.
- For intermediate n, we get a trade-off between bias and variance.

This allows us to balance how much we rely on actual returns (low bias, high variance) versus bootstrapped estimates (high bias, low variance), tuning our method to the needs of the problem.

N-Step TD

The **n-step return** is defined as:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n})$$

The value function update rule using the n-step return is:

$$V_{t+n-1}(S_t) \leftarrow V_{t+n-1}(S_t) + \alpha \left[G_{t:t+n} - V_{t+n-1}(S_t)\right]$$

N step TD

```
def ntd(pi, env, gamma=1.0,init_alpha=0.5,min_alpha=0.01,alpha_decay_ratio=0.5,
    n_step=3,n_episodes=500):
   nS = env.observation_space.n
   V = np.zeros(nS, dtype=np.float64)
   V_track = np.zeros((n_episodes, nS), dtype=np.float64)
   discounts = np.logspace(0, n_step+1, num=n_step+1, base=gamma, endpoint=False
       )
   alphas = decay_schedule(init_alpha, min_alpha,alpha_decay_ratio, n_episodes)
   for e in tqdm(range(n_episodes), leave=False):
      state, done, path = env.reset(), False, []
      while not done or path is not None:
         path = path[1:]
         while not done and len(path) < n_step:
            action = pi(state)
            next_state, reward, done, _ = env.step(action)
            experience = (state, reward, next_state, done)
            path.append(experience)
            state = next_state
            if done:
               break
         n = len(path)
         est_state = path[0][0]
         rewards = np.array(path)[:,1]
         partial_return = discounts[:n] * rewards
         bs_val = discounts[-1] * V[next_state] * (not done)
         ntd_target = np.sum(np.append(partial_return, bs_val))
         ntd_error = ntd_target - V[est_state]
         V[est_state] = V[est_state] + alphas[e] * ntd_error
         if len(path) == 1 and path[0][3]:
            path = None
      V_track[e] = V
   return V, V_track
```

TD(λ) Method

In last part we discussed about N step TD but N value can be anything we don't know which N value gives us the best result. Instead of committing to a specific n, TD(λ) uses a weighted combination of all n-step targets as a single target? I mean, our agent could go out and calculate the n-step targets corresponding to the one-, two-, three-, . . . , infinite-step target, then mix all of these targets with an exponentially decaying factor.

This is what a method called *forward-view TD(λ)* does. Forward-view TD(λ) is a prediction method that combines multiple n-steps into a single update. In this particular version, the agent will have to wait until the end of an episode before it can update the state-value function estimates. However, another method, called *backward-view TD(λ)*, can split the corresponding updates into partial updates and apply those partial updates to the state-value function estimates on every step.

Forward-view TD(λ)

$$G_{t:T}^{\lambda} = (1-\lambda)\sum_{n=1}^{T-t-1}\lambda^{n-1}G_{t:t+n} + \lambda^{T-t-1}G_{t:T}$$

$$G_{t:t+1} = R_{t+1} + \gamma V_t(S_{t+1}) \qquad \text{Weight: } (1-\lambda)$$

$$G_{t:T} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t-1}R_T \qquad \text{Weight: } \lambda^{T-t-1}$$

The issue with this approach is that you must sample an entire trajectory before you can calculate these values. you have the V will become available at time T:

$$V_T(S_t) = V_{T-1}(S_t) + \alpha_t\left[G_{t:T}^{\lambda} - V_{T-1}(S_t)\right]$$

Backward-view TD(λ)

When you encounter a state S_t,Here E is the Eligibility vector:,

$$E_t(S_t) = E_t(S_t) + 1$$

I'm not using a $V(s)$, but a V instead. Because we're multiplying by the eligibility vector, all eligible states will get the corresponding credit.

$$V_{t+1} = V_t + \alpha_t\left[R_{t+1} + \gamma V_t(S_{t+1}) - V_t(S_t)\right]E_t$$

Finally, we decay the eligibility:

$$E_{t+1} = E_t\gamma\lambda$$

N step TD

```python
def td_lambda(pi, env, gamma=1.0,init_alpha=0.5,min_alpha=0.01,alpha_decay_ratio
    =0.5,lambda_=0.3,n_episodes=500):
   nS = env.observation_space.n
   V = np.zeros(nS, dtype=np.float64)
   E = np.zeros(nS, dtype=np.float64)
   V_track = np.zeros((n_episodes, nS), dtype=np.float64)
   alphas = decay_schedule(init_alpha, min_alpha, alpha_decay_ratio, n_episodes)
   for e in tqdm(range(n_episodes), leave=False):
      E.fill(0)
      state, done = env.reset(), False
      while not done:
         action = pi(state)
         next_state, reward, done, _ = env.step(action)
         td_target = reward + gamma * V[next_state] * (not done)
         td_error = td_target - V[state]
         E[state] = E[state] + 1
         V = V + alphas[e] * td_error * E
         E = gamma * lambda_ * E
         state = next_state
      V_track[e] = V
   return V, V_track
```

9 Control in Reinforcement Learning

Control problems use action-value functions $Q(s, a)$ instead of state-values $V(s)$.
Unlike $V(s)$, the function $Q(s, a)$ enables policy improvement without an explicit model (MDP).This refers to problem of finding optimal policies. The control problem is usually solved by following the pattern of generalized policy iteration (GPI), where the competing processes of policy evaluation and policy improvement progressively move policies towards optimality.

Generalized Policy Iteration (GPI)

Continuous interplay of:

- Policy Evaluation(Estimate value functions)
- Policy Improvement(Make policy greedier with respect to estimated values)

Anatomy of RL Agents

1. **Experience:**
   - Via direct environment interaction or learned models.
   - Gathering and learning from data are distinct challenges.
2. **Estimate From Data:**
   - Value functions, models, or returns.
   - Can use MC, TD, n-step, λ-returns.
   - Policy gradient agents estimate policies directly.
3. **Improve Policy:**
   - Value-based: Improve policy by making better value estimates.
   - Policy-based: Directly improve using gradients or sampled returns.
   - Model-based: Use learned models for planning or policy/value learning.

Exploration in GPI

- Without access to full MDP, complete greedification is not possible.
- Use partial policy improvement to maintain exploration (e.g., ϵ-greedy).

MC control

The MC control is similar to MC prediction. The two main differences is that we now estimate the action-value function Q, and that we need to explore.

MC control

```
def mc_control(env,gamma=1.0,init_alpha=0.5,min_alpha=0.01,alpha_decay_ratio
    =0.5,init_epsilon=1.0,min_epsilon=0.1,epsilon_decay_ratio=0.9,n_episodes
    =3000,max_steps=200,first_visit=True):
   nS, nA = env.observation_space.n, env.action_space.n
   discounts = np.logspace(0, max_steps, num=max_steps, base=gamma, endpoint=
       False)
   alphas = decay_schedule(init_alpha, min_alpha, alpha_decay_ratio, n_episodes)
   epsilons = decay_schedule(init_epsilon,min_epsilon, epsilon_decay_ratio,
       n_episodes)
   pi_track = []
   Q = np.zeros((nS, nA), dtype=np.float64)
   Q_track = np.zeros((n_episodes, nS, nA), dtype=np.float64)
   select_action = lambda state, Q, epsilon: np.argmax(Q[state]) if np.random.
       random() > epsilon else np.random.randint(len(Q[state]))
   for e in tqdm(range(n_episodes), leave=False):
      trajectory = generate_trajectory(select_action,Q,epsilons[e],env,
          max_steps)
      visited = np.zeros((nS, nA), dtype=np.bool)
      for t, (state, action, reward, _, _) in enumerate(trajectory):
         if visited[state][action] and first_visit:
            continue
         visited[state][action] = True
         n_steps = len(trajectory[t:])
         G = np.sum(discounts[:n_steps] * trajectory[t:, 2])
         Q[state][action] = Q[state][action] + alphas[e] * (G - Q[state][action
             ])
      Q_track[e] = Q
      pi_track.append(np.argmax(Q, axis=1))
   V = np.max(Q, axis=1)
   pi = lambda s: {s:a for s, a in enumerate(np.argmax(Q, axis=1))}[s]
   return Q, V, pi, Q_track, pi_track
```

SARSA

Here it uses TD for policy evaluation part instead of MC (Which was used in MC control)this algorithm is implemented in SARSA(Improving Policies after each step).

SARSA

```
def sarsa(env,gamma=1.0,init_alpha=0.5,min_alpha=0.01,alpha_decay_ratio=0.5,
    init_epsilon=1.0,min_epsilon=0.1,epsilon_decay_ratio=0.9,n_episodes=3000):
   nS, nA = env.observation_space.n, env.action_space.n
   pi_track = []
   Q = np.zeros((nS, nA), dtype=np.float64)
   Q_track = np.zeros((n_episodes, nS, nA), dtype=np.float64)
   select_action = lambda state, Q, epsilon: np.argmax(Q[state]) \
      if np.random.random() > epsilon \
      else np.random.randint(len(Q[state]))
   alphas = decay_schedule(init_alpha, min_alpha, alpha_decay_ratio, n_episodes)
   epsilons = decay_schedule(init_epsilon,min_epsilon,epsilon_decay_ratio,
       n_episodes)
```

SARSA

```
    for e in tqdm(range(n_episodes), leave=False):
        state, done = env.reset(), False
        action = select_action(state, Q, epsilons[e])
        while not done:
            next_state, reward, done, _ = env.step(action)
            next_action = select_action(next_state, Q, epsilons[e])
            td_target = reward + gamma * Q[next_state][next_action] * (not done)
            td_error = td_target - Q[state][action]
            Q[state][action] = Q[state][action] + alphas[e] * td_error
            state, action = next_state, next_action
        Q_track[e] = Q
        pi_track.append(np.argmax(Q, axis=1))

    V = np.max(Q, axis=1)
    pi = lambda s: {s:a for s, a in enumerate(np.argmax(Q, axis=1))}[s]
    return Q, V, pi, Q_track, pi_track
```

10 Decoupling Behavior from Learning

In TD,it updates equation for state-value functions as

$$R_{t+1} + \gamma V(S_{t+1})$$

as the TD target. However, if you see at the TD update equation for action-value functions instead, which is

$$R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}),$$

you may notice there are a few more possibilities there. Look at the action being used and what that means. In RL,development of the ***Q-learning*** algorithm, a model-free off-policy bootstrapping method that directly approximates the optimal policy despite the policy generating experiences.

Q-learning

The **SARSA** algorithm is a sort of learning on the job. The agent learns about the same policy it uses for generating experience. This type of learning is called **on-policy**. **Off-policy learning**, on the other hand, is sort of learning from others.The agent learns about a policy that's different from the policy-generating experiences.

In off-policy learning, there are two policies:

- **Behavior policy**: used to generate experiences through interaction with the environment.
- **Target policy**: the policy we're learning about.

SARSA is an on-policy method; Q-learning is an off-policy one.

Q-learning and SARSA

In SARSA,it updates

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha_t \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right]$$

But in Q-learning it updates as

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha_t \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

Q-learning

```
def q_learning(env, gamma=1.0,init_alpha=0.5,min_alpha=0.01,alpha_decay_ratio
    =0.5,init_epsilon=1.0,min_epsilon=0.1,epsilon_decay_ratio=0.9,n_episodes
    =3000):
   nS, nA = env.observation_space.n, env.action_space.n
   pi_track = []
   Q = np.zeros((nS, nA), dtype=np.float64)
   Q_track = np.zeros((n_episodes, nS, nA), dtype=np.float64)
   select_action = lambda state, Q, epsilon: np.argmax(Q[state]) \
      if np.random.random() > epsilon \
      else np.random.randint(len(Q[state]))
   alphas = decay_schedule(init_alpha, min_alpha, alpha_decay_ratio, n_episodes)
   epsilons = decay_schedule(init_epsilon, min_epsilon, epsilon_decay_ratio,
       n_episodes)
   for e in tqdm(range(n_episodes), leave=False):
      state, done = env.reset(), False
      while not done:
         action = select_action(state, Q, epsilons[e])
         next_state, reward, done, _ = env.step(action)
         td_target = reward + gamma * Q[next_state].max() * (not done)
         td_error = td_target - Q[state][action]
         Q[state][action] = Q[state][action] + alphas[e] * td_error
         state = next_state

      Q_track[e] = Q
      pi_track.append(np.argmax(Q, axis=1))

   V = np.max(Q, axis=1)
   pi = lambda s: {s:a for s, a in enumerate(np.argmax(Q, axis=1))}[s]
   return Q, V, pi, Q_track, pi_track
```

Double Q-learning

Q-learning often **overestimates** the value function.As in every step, we take the maximum over the estimates of the action-value function of the next state. But what we actually need is the value of the maximum, not the maximum of the values.

This approach introduces a well-known problem called **maximization bias**. That is, the use of a maximum of biased estimates as the estimate of the maximum value. Why is this a problem?

- These action-value estimates are typically noisy or biased due to sampling.
- The max operator selects the highest among these biased estimates.
- This leads to persistent overestimation and thus inaccurate targets in the TD update.

Double Q-learning

```
def double_q_learning(env,gamma=1.0,init_alpha=0.5,min_alpha=0.01,
    alpha_decay_ratio=0.5,init_epsilon=1.0,min_epsilon=0.1,epsilon_decay_ratio
    =0.9,n_episodes=3000):
   nS, nA = env.observation_space.n, env.action_space.n
   pi_track = []
   Q1 = np.zeros((nS, nA), dtype=np.float64)
   Q2 = np.zeros((nS, nA), dtype=np.float64)
   Q_track1 = np.zeros((n_episodes, nS, nA), dtype=np.float64)
   Q_track2 = np.zeros((n_episodes, nS, nA), dtype=np.float64)
   select_action = lambda state, Q, epsilon: np.argmax(Q[state]) if np.random.
       random() > epsilon else np.random.randint(len(Q[state]))
   alphas = decay_schedule(init_alpha,min_alpha,alpha_decay_ratio,n_episodes)
   epsilons = decay_schedule(init_epsilon, min_epsilon, epsilon_decay_ratio,
       n_episodes)
   for e in tqdm(range(n_episodes), leave=False):
      state, done = env.reset(), False
      while not done:
         action = select_action(state, (Q1 + Q2)/2, epsilons[e])
         next_state, reward, done, _ = env.step(action)

         if np.random.randint(2):
            argmax_Q1 = np.argmax(Q1[next_state])
            td_target = reward + gamma * Q2[next_state][argmax_Q1] * (not done)
            td_error = td_target - Q1[state][action]
            Q1[state][action] = Q1[state][action] + alphas[e] * td_error
         else:
            argmax_Q2 = np.argmax(Q2[next_state])
            td_target = reward + gamma * Q1[next_state][argmax_Q2] * (not done)
            td_error = td_target - Q2[state][action]
            Q2[state][action] = Q2[state][action] + alphas[e] * td_error
         state = next_state

      Q_track1[e] = Q1
      Q_track2[e] = Q2
      pi_track.append(np.argmax((Q1 + Q2)/2, axis=1))

   Q = (Q1 + Q2)/2.
   V = np.max(Q, axis=1)
   pi = lambda s: {s:a for s, a in enumerate(np.argmax(Q, axis=1))}[s]
   return Q, V, pi, (Q_track1 + Q_track2)/2., pi_track
```

Here we have **Double Q-learning** is one such technique. It reduces overestimation by decoupling action selection from value estimation, using two value functions instead of one. One way of dealing with maximization bias is to track estimates in two Q-functions. At each time step, we choose one of them to determine the action, to determine which estimate is the highest according to that Q-function. But, then we use the other Q-function to obtain that action's estimate. By doing this, there's a lower chance of always having a positive bias error. Then, to select an action for interacting with the environment, we use the average, or the sum,across the two Q-functions for that state. That is, the maximum over $Q_1(S_{t+1}) + Q_2(S_{t+1})$, forinstance. The technique of using these two Q-functions is called double learning, and the algorithm that implements this technique is called double Q-learning. In a few chapters, you'll learnabout a deep reinforcement learning algorithm called double deep Q-networks (DDQN),which uses a variant of this double learning technique.[3][8][6][4]

11 Introduction-second Half

In this part of SoS,I learnt about *Eligibility Traces* and tried to implement a CHESS ENGINE using DRL by reading a thesis[10].

12 Eligibility Traces

Eligibility Traces are one of the basic mechanisms of RL.There are two types of Eligibility Traces:

- Forward View
- Backward View

Forward View of TD(λ)

The forward view interprets them as a conceptual bridge between Monte Carlo and TD methods, offering a spectrum of intermediate algorithms that often perform better than either extreme. There is not only use of backup not just towards return of n-steps,They set of returns can be averaged.The composite return possesses an error reduction property similar to that of individual n-step returns.TD(λ) involves averageing of n-step backups.The λ-return of TD(λ) is

TD(λ)

$$\mathbf{L}_t = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_t^{t+n}(V_t(S_{t+n}))$$

On each step, t, it computes an increment, $\Delta_t(S_t)$, to the value of the state occurring on that step:

$$\Delta_t(S_t) = \alpha[L_t - V_t(S_t)]$$

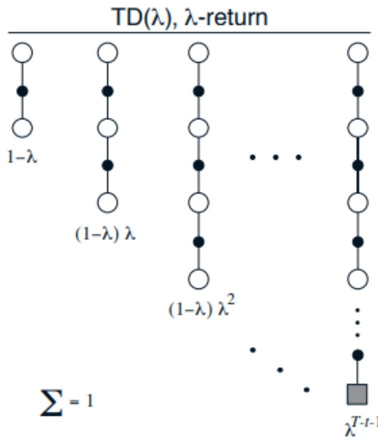


Figure 1: The backup diagram for TD(λ)

When there is a intermediate value of N then it gives the best performance.The approach that we have been taking so far is what we call the forward, view of a learning algorithm. For each state visited, we look forward in time to all the future rewards and decide how best to combine them.Once we look ahead and update a state, we move on and no longer need to revisit it. However, future states are seen and processed multiple times—each time from the perspective of a state that came before.In offline case,λ-return is the TD(λ) algorithm.

Backward View of TD(λ)

The backward view of TD(λ) takes a more mechanistic approach by introducing a new variable—an *accumulating eligibility trace*—for each state. This trace stores how recently and frequently a state has been visited and is used to distribute TD error backward in time.

TD(λ) Backward View

Eligibility traces are updated at each time step t as:

$$E_t(s) = \begin{cases} \gamma\lambda E_{t-1}(s) + 1, & \text{if } s = S_t \\ \gamma\lambda E_{t-1}(s), & \text{otherwise} \end{cases}$$

The TD error is:

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

All state values are updated as:

$$V(s) \leftarrow V(s) + \alpha\delta_t E_t(s)$$

[8]

This approach is practical for online learning. As each new state is visited, traces for all states decay, and the current state's trace is incremented. Then, the TD error is applied to all previously visited states, weighted by how "eligible" they are for learning based on the trace.It is oriented in backward direction in time.

TD(λ)

For each episode:

- Initialize $V(s)$ arbitrarily
- Initialize $E(s) \leftarrow 0$ for all s
- Observe initial state S

For each step:

- Take action, observe R, S'
- Compute $\delta \leftarrow R + \gamma V(S') - V(S)$
- $E(S) \leftarrow E(S) + 1$
- For all s: $\quad V(s) \leftarrow V(s) + \alpha\delta E(s) \quad$ and $\quad E(s) \leftarrow \gamma\lambda E(s)$
- $S \leftarrow S'$

Equivalence of Forward and Backward Views

Although they seem different, the forward and backward views of TD(λ) result in the same algorithm under certain conditions. The forward view looks ahead and computes expected returns, while the backward view incrementally updates past states based on eligibility traces. For online updates, the backward view is more practical, and both views converge to the same value function in the limit.

SARSA(λ)

SARSA(λ) is an on-policy control algorithm that extends SARSA with eligibility traces. The algorithm updates action-value estimates $Q(s, a)$ based on TD errors and traces of previously visited state-action pairs.

It supports accumulating or replacing traces and works well in real-time settings. The update rule is:

TD(λ) Backward View

$$\delta_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)$$
$$Q(s, a) \leftarrow Q(s, a) + \alpha \delta_t E_t(s, a)$$

(Accumulating) $\mathrm{E}_t(s, a) = \gamma\lambda E_{t-1}(s, a) + \mathbb{I}_{\{S_t=s\}}\mathbb{I}_{\{A_t=a\}}$

(Dutch) $\mathrm{E}_t(s, a) = (1 - \alpha)\gamma\lambda E_{t-1}(s, a) + \mathbb{I}_{\{S_t=s\}}\mathbb{I}_{\{A_t=a\}}$

(Replacing) $\mathrm{E}_t(s, a) = (1 - \mathbb{I}_{\{S_t=s\}}\mathbb{I}_{\{A_t=a\}})\gamma\lambda E_{t-1}(s, a) + \mathbb{I}_{\{S_t=s\}}\mathbb{I}_{\{A_t=a\}} where \mathrm{E}_t(s, a)$ is the eligibility trace for (s, a).

Watkins's Q(λ)

Watkins's Q(λ) is an off-policy method that integrates Q-learning with eligibility traces. Unlike SARSA(λ), it cuts traces when a non-greedy action is taken. This ensures off-policy correctness while still benefiting from multi-step returns when following the greedy policy. The core idea is: - Maintain traces like in SARSA(λ) - Set all traces to zero if a non-greedy action is selected

Implementation Issues

Although eligibility traces seem memory-heavy, efficient implementations use sparse representations — only storing non-zero traces. Techniques include:

1. Using hash tables to track active traces
2. Clearing small traces after they fall below a threshold
3. Choosing between accumulating and replacing traces for stability

Variable λ

Rather than using a constant λ, dynamic adjustment can improve learning. A higher λ leads to longer backups (more Monte Carlo–like), while lower values favor immediate TD learning. For example, λ_t can decay with time or be adjusted based on the variance of TD errors:

Variable λ

Definition

$$E_t(s) = \begin{cases} \gamma\lambda_t E_{t-1}(s), & \text{if } s \neq S_t \\ \gamma\lambda_t E_{t-1}(s) + 1, & \text{if } s = S_t \end{cases}$$

where $E_t(s)$ is the eligibility trace for state s at time t, and λ_t is a possibly time-varying trace decay parameter.

$$\lambda_t = \begin{cases} 1 - \frac{1}{t+1}, & \text{if variable} \\ \text{constant}, & \text{otherwise} \end{cases}$$

Conclusions-Eligibility Traces

Eligibility traces unify TD and Monte Carlo learning, offering a spectrum of methods controlled by the parameter λ. TD(λ) and its variants (SARSA(λ), Q(λ)) provide practical tools for both prediction and control. Choosing appropriate settings for λ, α, and the type of trace (accumulating or replacing) can significantly affect learning performance.

13 RL CHESS ENGINE

13.1 Introduction

In this part we will be discussing how to develop chess engine which can play against human.Chess is a two player game.It consists of various pieces which is movable on board with certain rules.[11] here pieces represented as

Symbol	**White Piece**	**Symbol**	**Black Piece**
K	King	k	King
Q	Queen	q	Queen
R	Rook	r	Rook
B	Bishop	b	Bishop
N	Knight	n	Knight
P	Pawn	p	Pawn

13.2 Chess Move Input Format

- The player who plays **White** will start the game.
- Each move input given by the player will follow the format:

  `{from_square}{to_square}`

- Each square is represented as:

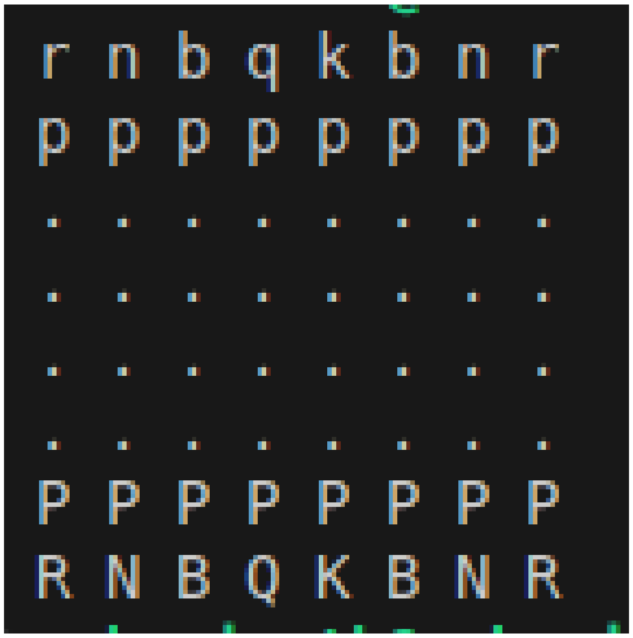
  `{file}{rank}`

  where:
  - `file` is a horizontal alphabet from `a` to `h`.
  - `rank` is a vertical number from `1` to `8`.
- No spaces, dashes, or other separators should be used in the move.
- Each move input must be exactly 4 characters long.[1]

Examples

- `e2e4` – Move a piece from square `e2` to `e4`.
- `g1f3` – Move a knight from `g1` to `f3`.
- `e7e8` – Pawn reaches the last rank, possible promotion.

We will represent our board as in form of above piece symbols.See the below diagram:



```
r n b q k b n r
p p p p p p p p
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
P P P P P P P P
R N B Q K B N R
```

Figure 2: Enter Caption

13.3 Game Results and End Conditions

- The result of the game will be one of the following:
  - **Win**
  - **Lose**
  - **Draw**
- The game can end under the following conditions:
  - **Checkmate** – One player wins by putting the opponent's king in an inescapable check.
  - **Stalemate** – The player whose turn it is has no legal moves and is not in check. The game ends in a draw.
  - **Threefold repetition** – The same position occurs three times with the same player to move. The game ends in a draw.
  - **Other draw conditions** – Such as the fifty-move rule or insufficient mating material.

13.4 Chess Engine

A chess engine is a computer program that analyzes positions in chess or chess variants, and generates a list of moves that it regards as strongest. Given any chess position, the engine will estimate the winner of that position based on the strength of the possible future moves up to a certain depth. The strength of a chess engine is often determined by the number of positions, both in depth and breadth, that the engine can evaluate. This means that with time, as computational power increases, chess engines will keep getting stronger.

13.5 How do traditional chess engines work?

The Minimax Algorithm

Minimax is a game tree search algorithm used to minimize the opponent's maximum payoff. The tree is traversed to a certain depth and values are propagated based on whether it is the maximizing or minimizing player's turn.

13.5.1 The Evaluation Function

A heuristic function $f(n)$ evaluates the strength of a position, combining static evaluation of pieces, control, safety, etc.

Pseudocode

```
def minimax(node, depth, maximizingPlayer):
   if depth == 0 or node is terminal:
      return heuristic_value(node)

   if maximizingPlayer:
      value = -float('inf')
      for child in node.children:
         value = max(value, minimax(child, depth - 1, False))
      return value
   else:
      value = float('inf')
      for child in node.children:
         value = min(value, minimax(child, depth - 1, True))
      return value
```

13.5.2 Alpha-Beta Pruning

Alpha-beta pruning reduces nodes evaluated by pruning branches that cannot affect the final decision. This makes minimax feasible for deeper searches.

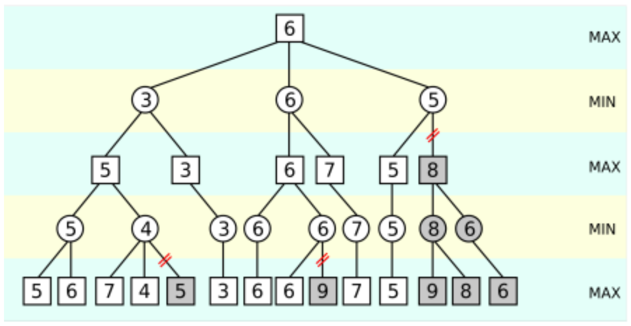


Figure 3: Example of alpha-beta pruning in minimax

13.6 Monte Carlo Tree Search (MCTS)

MCTS is a search algorithm that builds a search tree using random exploration of the most promising moves. Each execution of the algorithm is called an *MCTS simulation* and consists of four steps:

Selection

From the root node, select child nodes based on a formula (we will use Upper Confidence Bound - UCB and q value) until a leaf node is reached. If the root is a leaf, skip to Expansion.

Expansion

If the leaf node is non-terminal, create children for all possible actions.

Simulation / Rollout

Simulate the game from the newly expanded node using random moves.

Backpropagation

Propagate the simulation result up the tree. Each node updates its visit count and win count.

Note: MCTS in chess is costly, as simulating full games is inefficient. Hence, the selection strategy must balance exploration and exploitation.

13.7 AlphaZero

AlphaZero, developed by DeepMind, combines MCTS with deep neural networks to master games like Go, Chess, and Shogi. It differs from traditional engines by learning purely through self-play.We develop using some ideas used to build AlphaZero to build our Engine.[7]

13.8 Coding Part

13.9 MCTS & Tree Structure

The MCTS algorithm contains various steps which were explained in previous steps.To select a move, the agent observes the current board state and uses the `MCTS` class to construct a search tree rooted at that state. The MCTS algorithm runs 1000 simulations(may change depending how many you want). More simulations yield better accuracy but increase computation time.

Each simulation uses the neural network to evaluate the position, outputting a **policy** (probability distribution over moves) and a **value** (estimated outcome). These are used to update the search tree.

After all simulations, the agent selects a move:

- **Stochastic selection**: Samples a move based on visit count distribution. Useful for training due to added diversity.
- **Deterministic selection**: Chooses the most visited move. Preferred for evaluation or competitive play.

The Tree consists of Chess Board positions which is stored in nodes and the data related to moves made between positions are stored in Edge.

node.py

This file contains the class *Node*, which stores information about the current state of the chessboard using the following parameters:

- **state (Position)**: A string representation of the board using FEN notation [12].
- **turn**: A boolean indicating the player to move next (1 for White, 0 for Black).
- **value**: The evaluation value of the node.
- **N**: The visit count of the node.
- **edges**: A list of edges (actions) connecting this node to its children.
- **add_child(child,action,prior)**:Adds a child node resulting from a move. Returns the `Edge` created between the current node and the child.
- **get_all_children**:Recursively returns a list of all the nodes which are descendant connected to the current node via its edges.
- **get_edge(action)**: Returns the `Edge` from the current node that corresponds to the given move (`action`).
- **step**: Executes a move and returns the resulting `new_state`.

edge.py

This file defines the *Edge* class, which connects two `Node` objects in the MCTS tree and represents an action taken on the chessboard.

- **in_node, out_node**: References to the parent and child `Node` objects connected by this edge.
- **action**: A `chess.Move` object representing the move taken from `in_node` to `out_node`.
- **player_turn**: A boolean indicating if it is White's turn at the source node.
- **N**: The number of times this edge has been visited during simulations.
- **W**: The total action value from simulations passing through this edge.
- **P**: The prior probability of selecting this action, given by the neural network.
- **ucb(noise)**: Computes the Upper Confidence Bound (UCB) value used during selection in MCTS. It combines exploitation (Q-value) and exploration (based on prior and visit counts). The UCB formula depends on whether the player is White or Black.

mcts.py

This file defines the *MCTS* (Monte Carlo Tree Search) class, which helps the agent select moves by simulating future game states and evaluating them with a neural network.

- **root**: The root `Node` of the current search tree, representing the starting board state.
- **stochastic**: A boolean that, if true, adds Dirichlet noise at the root to encourage more exploration .
- **run_simulations(n)**: Performs n MCTS simulations starting from the root node. Each simulation expands the tree and backpropagates values.
- **select_child(node)**: Follows the very good path down the tree using the UCB (Upper Confidence Bound) formula until a leaf node is reached.

$$UCB = \Big(\log(\frac{(1 + N_{\text{parent}} + C_{\text{base}})}{C_{\text{base}}}) + C_{\text{init}} \Big) \cdot P \cdot \frac{\sqrt{N_{\text{parent}}}}{(1 + N)} \tag{1}$$

- C_{base} and C_{init} are constants that can be changed in the config file. The same values as AlphaZero were used.
- N_{parent} is the visit count of the input node
- N is the visit count of the edge
- P is the prior probability of the edge

The selection step combines this UCB formula with the edges action-value and visit count:

$$Q = \frac{W}{N + 1} \tag{2}$$

$$V = \begin{cases} UCB + Q & \text{if white} \\ UCB - Q & \text{if black} \end{cases} \tag{3}$$

Figure 4: Value used for selection

- **expand(leaf)**: Generates possible moves from the leaf node, uses the neural network to evaluate them, and adds resulting nodes to the tree.
- **backpropagate(end_node, value)**: Updates visit counts and total values along the path from the leaf node back to the root.
- **map_valid_move(move)**: Converts a legal move into a tuple of plane index, row, and column—used to match the move with the policy output from the neural network.
- **probabilities_to_actions(p, board)**: Converts the policy tensor (neural network output) into a dictionary of legal moves and their associated probabilities.
- **get_policy_distribution()**: Returns a normalized distribution over legal moves, based on how often each was visited during simulations.
- **best_move()**: Selects the move from the root with the highest visit count—used during actual gameplay.
- **get_policy(board, simulations)**: Runs MCTS on a given board for a fixed number of simulations and returns a probability distribution over legal moves.

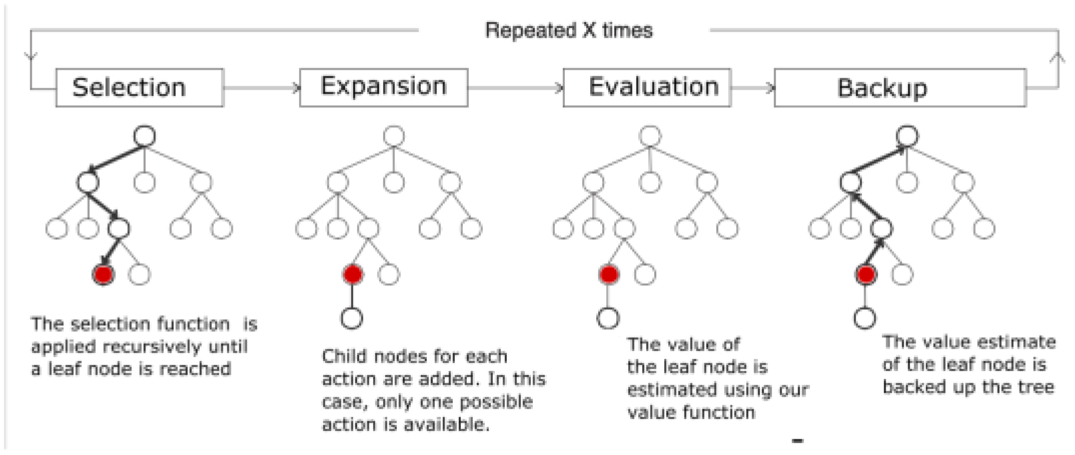

Repeated X times
Selection
Expansion
Evaluation
Backup
The selection function is applied recursively until a leaf node is reached
Child nodes for each action are added. In this case, only one possible action is available.
The value of the leaf node is estimated using our value function
The value estimate of the leaf node is backed up the tree

Figure 5: MCTS Algorithm

13.10 Neural Network Input & ChessEnv

chessEnv.py

This file defines the *ChessEnv* class, which serves as a lightweight interface to interact with the `chess.Board` object. It provides utility methods for board manipulation and converting board states into neural network input.

- **fen**: Stores the initial position of the board in FEN format.
- **reset()**: Resets the board to the initial FEN position.
- **step(action)**: Applies a given `chess.Move` to the board and returns the updated board.
- **state_to_input(fen)**: Converts a board position (FEN) into a tensor input for the neural network. It encodes piece positions, castling rights, turn, en passant ,etc.
- **estimate_winner(board)**: Roughly estimates which side is winning based on material count. Returns a small numerical bias indicating advantage.

- **get_piece_amount(board)**: Returns the total number of pieces on the board.[2]

The input to the network is a tensor of shape $19 \times 8 \times 8$, The function in ChessEnv state_to_input helps encoding the current game state with the following 19 planes:[7]

- **1 plane** indicating the current player's turn:
  - All entries are 1 if it's White's turn, 0 if it's Black's.
- **4 planes** for castling rights:
  - White kingside, White queenside, Black kingside, Black queenside.
  - Each plane is filled with 1 if the corresponding right is still available.
- **1 plane** for the 50-move rule:
  - Filled with 1 if a draw is possible due to 50 moves without a pawn move or capture.
- **12 planes** encoding piece positions:
  - One plane for each piece type and color (6 per side: P, N, B, R, Q, K).
  - A cell is 1 if the corresponding piece is present at that square, 0 otherwise.
- **1 plane** for en passant:
  - The en passant target square is marked with 1 if a pawn capture is possible.

rlmodelbuilder.py

This file defines the deep learning model architecture used for training the chess-playing agent. The model follows an similar one to AlphaZero design, using convolutional layers with two output heads—one to predict the best move, and another to evaluate how good the current board position is.[2]

- **RLModelBuilder**: A class that builds the full neural network.
- **build_model()**: Creates and compiles the full model with 19 residual layers, one head that suggests the next move (policy), and another that evaluates the board (value).
- **build_convolutional_layer(input)**: Creates a basic convolutional block with batch normalization and ReLU activation.
- **build_residual_layer(input)**: Builds a residual block that allows deeper architectures by passing information through shortcut connections.
- **build_policy_head()**: Produces a probability distribution over all possible moves on the board.
- **build_value_head()**: Outputs a single value estimating the likely outcome of the current board position.
- **Main script**: When run directly, this script creates and saves the model based on command-line arguments .

agent.py

This file defines the `Agent` class, which acts as the main decision-maker in the system. It connects the neural network with MCTS to decide on the best moves during a game.

- **Agent**: Loads a trained neural network model and sets up the MCTS. It works from a given starting board position.
- **build_model()**: Creates a fresh copy of the model architecture.
- **save_model(timestamped)**: Saves the current model to the folder.
- **predict(data)**: Takes the current board state (as input data) and returns two things: a set of move probabilities (what to play), and a value estimate (how good the position is). This uses local, fast prediction.

train.py

This file handles the training process for the neural network. It defines a `Trainer` class that takes in a compiled model and performs training on data collected from self-play or the folder. The training data includes FEN positions, target move probabilities, and value estimates for each position. The trainer splits the data into input tensors and output labels, trains the model in batches , and tracks training progress. At the end of training, the model is saved with a timestamp.

game.py

This file implements the `Game` class, which handles the gameplay loop between two agents. It supports MCTS-based self-play and data collection for training. Key components include:

- **Game(env, white, black)**: Initializes a new game between two agents using the given environment.
- **reset()**: Resets the board and turn to the initial state.
- **get_winner(result)**: Returns a numerical value representing the winner based on the PGN result string.
- **play_move(stochastic, previous_moves, save_moves)**: Plays a single move using MCTS; supports reuse of the tree across turns. Selects the move based on visit counts or probabilities.
- **save_to_memory(state, moves)**: Records the board state and move distribution into memory for training.

main.py

This file provides a command terminal interface for a human player to play against the agent. The core functionality is in the `Main` class.

- **Main(player, model_path)**: Initializes the game based on whether the human plays as white or black. It loads the AI model and sets up the environment.You can use to play between two agents also.
- **play_game()**: Runs the main game loop, alternating between two players. It checks for game termination and prints the board after each move.
- **get_player_move()**: Accepts a move from the human in UCI format validates it, and plays it if legal.
- **opponent_move()**: Lets the agent make a move using MCTS with deterministic selection.
- **___main___ block**: Parses command-line arguments to determine who plays first and loads the model and asks whether it is self play. Starts the game session.

13.11 Helper files

config.py

All of the project's important values and file paths are stored in this file. It contains configurations for neural network architecture, training parameters, directories. The system is simpler to set up, test, and maintain when these values are centralized.

local_prediction.py

This contains one function given below: **predict_local(model, args)**: Efficiently runs the model on given inputs using TensorFlow's fast graph execution.

mapper.py

This file defines how different types of chess moves are converted into specific index positions in the neural network's output tensor. These mappings allow the network to understand and predict moves based on direction and piece type.

- **QueenDirection, KnightMove, UnderPromotion**: These are enums representing the possible directions a queen can move which can used to refer rook ,pawn,king and bishop also, the unique L-shaped patterns for a knight, and underpromotion piece types (to knight, bishop, or rook).
- **Mapping.mapper**: A dictionary that assigns each movement type to one or more plane indices.
- **get_queenlike_move(from, to)**: Returns in which direction and how far a piece like a queen, rook, or bishop moved, given the starting and ending squares.
- **get_knight_move(from, to)**: Returns which of the eight knight patterns a move follows.
- **get_underpromotion_move(type, from, to)**: the cases where a pawn promotes to a piece other than a queen, and returns the piece type and direction for mapping.
- **knight_mappings**: A predefined list of how knights move, represented as numerical differences between board squares.

utils.py

This file provides utility functions to convert chess moves into tensor formats compatible with the neural network's output layer.

- **moves_to_output_vector(moves, board)**: Converts a dictionary of move probabilities into a $73 \times 8 \times 8$ tensor. Each plane represents a different move type or direction.
- **move_to_plane_index(move, board)**: Maps a UCI move string and a given board state to its corresponding output tensor location in terms of plane index, row, and column.

dummydata.py

This script will:

- Simulate 1 full chess game.
- Save each position as a tuple: `(FEN, policy, result)`.
- Store the output in a timestamped file located in:

  ```
  data/selfplay/selfplay-YYYY-MM-DD_HH-MM-SS.npy
  ```

13.12 Running Instructions

1. Environment Setup

Install Requirements

```
pip install tensorflow keras numpy python-chess tqdm pandas python-dotenv
```

2. Create a Model

Run this to create the initial model:

```
python3 rlmodelbuilder.py --model-folder models/ --model-name {model_name}
```

This saves `model_name.h5` in the `models/` folder.

3. Generate Training Data (Self-play)

Run the script to generate fake self-play data:

```
python3 dummydata.py
```

Saved in `data/selfplay/`.

4. Train the Model

Train the model on the generated data:

```
python3 train.py --model models/{model_name}.h5 --data-folder data/selfplay
```

5. Play the Game

a. Play vs Agent

As White:

```
python3 main.py --model models/{model_name}.h5 --player white
```

As Black:

```
python3 main.py --model models/{model_name}.h5 --player black
```

b. Self-play (Agent vs Agent)

```
python3 main.py --model models/{model_name}.h5 --selfplay
```

13.13 Links

You can checkout my code in [5].
To connect gpu with tensorflow gpu support check this [9]

References

[1] Niklas Fiekas. *python-chess: A chess library for Python*. 2015–2024. URL: `https://python-chess.readthedocs.io/`.

[2] Zachary Jefferis. *chess-deep-rl: Deep Reinforcement Learning for Chess using MCTS and Neural Networks*. 2021. URL: `https://github.com/zjeffer/chess-deep-rl`.

[3] Miguel Morales. *Grokking Deep Reinforcement Learning*. Manning Publications, 2020.

[4] Miguel Morales. *Grokking Deep Reinforcement Learning*. GitHub repository. 2020. URL: `https://github.com/mimoralea/gdrl`.

[5] Sabarinath S. *RL Chess*. 2025. URL: `https://github.com/Sabari293/RL-chess-1`.

[6] David Silver. *Reinforcement Learning Course*. `https://www.davidsilver.uk/teaching/`. Lecture series by DeepMind and University College London. 2015.

[7] David Silver et al. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". In: *arXiv preprint arXiv:1712.01815* (2017). URL: `https://arxiv.org/abs/1712.01815`.

[8] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2nd. MIT Press, 2018.

[9] TensorFlow. *TensorFlow GPU Support*. 2024. URL: `https://www.tensorflow.org/install/gpu`.

[10] Tuur Vanhoutte. "How to Create a Chess Engine with Deep Reinforcement Learning: A Critical Look at DeepMind's AlphaZero". Bachelor's thesis. Howest University of Applied Sciences, 2022.

[11] Wikipedia contributors. *Chess*. Wikipedia. May 2022. URL: `https://en.wikipedia.org/w/index.php?title=Chess&oldid=1085844722`.

[12] Wikipedia contributors. *Forsyth–Edwards Notation*. Feb. 2022. URL: `https://en.wikipedia.org/w/index.php?title=Forsyth%E2%80%93Edwards_Notation&oldid=1069540048`.