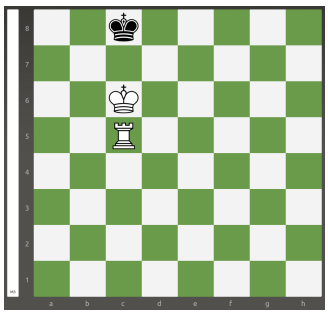
Guide to RL

1. Hi! I'm @naklecha & I love learning through examples and jumping right into things. It works well for me, it's fun and imo it's the best way to learn anything. So, that's what I'm going to do. :)
   Fuck it! Let's start by trying to solve chess.

   note: please read this blog in dark theme (the colors are nicer that way)

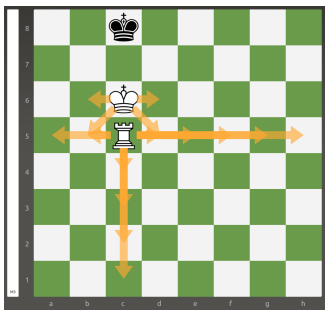
2. To start with let's call the initial *"state"* of the chessboard - *s*



3. In the above board state, white is winning because white has an extra rook! But let's look at it from a computer's pov. This is what a computer would know without calculations or lookahead:
   - the position of all the pieces on the board, aka the board state
   - which player is up next (white in this case)
   - valid moves that can be made from the current board state, aka the action space

   It doesn't matter if the player is 1 move away from the win, 30 moves away or losing in 3 moves. Without additional calculations, the computer has no idea if the position is good and by how much. Let's try and calculate how good the position is.
4. Now, for any given chess board let's say we have a magical function that can give us a True or False boolean output if the position is winning or losing. Let's call this magical function (in reinforcement learning it's called the value function) - `V(s)` , this function takes in a state of the board and returns True or False if the position is winning or not.
5. Okay, but realistically can this function ever exist for the game of chess? The branching factor in chess (the number of possible moves at each turn) is around 30-35 on average. After just 5 moves, you're already looking at millions of positions. Now think of a 20-30 move game. We are looking at roughly 2^50 possible moves. That's fucked up.

   You are going to realize that picking chess as an example was a bad idea because its branching factor is too high. But, picking bad examples helps me learn better. Later, we can move on to more complex (actually useful) algorithms that solve some of those problems.
6. For now let's ignore the part where I said this is impossible, and look at a simple chessboard state (a checkmate in 3 for example). In this example, it's white to move let's determine the value of the position, `V(s)` for white.
7. From the starting state (board position), white can make a bunch of possible moves (shown below) — 15 to exact.



8. It should be pretty intuitive that the *"value"* of the current position is somehow dependent on the value of the next position. The most naive version of this thought process would be something like:

   `V(s) = max( V(s') ) for all actions a (s → after an action a → s')`

The above equation essentially says — "the value of the chess position is the value of the chess position after making the best move (because if we are playing optimally we would always play the best chess move, aka take the best action)" which obviously makes sense.

9. Math people love to write equations, but to me, it's a lot easier to look at the above equation as just a tree search (depth-first search). There is a problem here though — even if we had infinite compute and time and traversing a chess game's tree the equation — V(s) = max(V(s')) doesn't care if the position is checkmate in 3 moves or 20 moves. Sometimes, you would prefer a position that is checkmate in 3 moves and should have a higher *"value"* than a position in 20.

   This is where I'm going to introduce you to something (in the rl world) called the discount factor (γ) which will enable us to penalize *"states"* that are further away from the "*optimal/winning state*". Now, the equation looks something like:

   V(s) = max(γ * V(s')) for all actions a (s → after an action a → s') and γ is the discount factor with a value range of [0,1].

   Let's say the *"value"* of a checkmate position is 100 when you win and -100 when you lose. The values at positions when considering the discount factor (let's say the discount factor is 0.9).

   V(checkmate position) = 100

   V(checkmate in 1 position) = 0.9 * 100 = 90

   V(checkmate in 2 position) = 0.9 * 90 = 81

   V(checkmate in 3 position) = 0.9 * 81 = 72.9

   and so on... the *"discount factor"* ensures that when a position is "more" winning or closer to a win, the *state* has a higher *value*.

10. Okay, I'll be honest in my chess example the *"discount factor"* isn't that useful. Like who cares if a position is checkmate in 20 or 2 moves? It's a guaranteed win either way. I get it! Hear me out though... Not every situation/game is deterministic.

    Think of the situation in which you are trying to solve a more complex game like Dota2 and you have to destroy the enemy's base. If you haven't played the game yet, here is a quick explanation — each team of 5 players has a base that they have to protect. The way you win a game of Dota is by destroying the enemy's base, if the enemy destroys your base you lose.

    If you play Dota2 or have played the game before, don't get mad at me I'm offensively oversimplifying here.

Dota is a lot more complex than chess. Not only is its action space MASSIVE (almost infinite in practice) but anything can happen at any time because you don't know what items (items are something you can buy during a Dota game duration and they help you win games) the enemy possesses and sometimes you don't even know the position of your enemies.

Okay, so coming back to why the discount factor is needed — in a nondeterministic game like Dota2 anything is possible. When anything is possible and the game is undeterministic, you would want to end the game ASAP!!!! The discount factor with this equation — $V(s) = \max(\gamma * V(s'))$ is great because it will help prioritize the shortest sequence of *actions* that will likely lead you to a win.

11. So far in both the chess and Dota2 examples, we have some major problems:
    a. Both games have so many possible *states* that our current algorithm $V(s) = \max(\gamma * V(s'))$ is impractical because the recursive depth of $V(s)$ is essentially unthinkably massive for both games.
    b. They are both 2 player games, it's not clear what the *value function* calculation looks like.
    c. In the Dota2 example, I talked about undeterministic states and actions. The current equation doesn't account for randomness.
    d. Dota2 has both continuous actions and discrete — you can click anywhere on the map and your hero will move there. This creates a constant 2D action space for movement, similar to how a robot might move in a 2D plane. The x and y coordinates where you can click are continuous values within the bounds of the map.

       However, many other actions are discrete: selecting which ability to use (typically 4 unique abilities per hero), choosing which item to use (6 item slots), targeting abilities (either on a unit or a location), attack commands, and shop purchases.

       ^^ This is tricky as fuck, our algorithm falls apart because we expect actions to be discrete and the resulting states to be discrete as well. This combination makes Dota2's action space what we call a "hybrid" or "mixed" action space.

       A non-hybrid but continuous action space example would be self-driving cars. When you're driving, you don't press the gas pedal in discrete steps - it's a continuous range from 0% to 100%. Similarly, the steering wheel can be turned to any angle.

12. Before I address the problems that I mentioned above I want to talk about *"rewards"* in reinforcement learning. Think about learning to play chess. If we only had `V(s) = max(γ * V(s'))` with no rewards, and we only knew that checkmate states were good or bad, we'd face a serious problem: most chess positions would appear equally valuable until we could see all the way to checkmate. This is because the value would only "flow backwards" from the final states.

    This problem can be easily addressed by having intermediate rewards. Consider the following equation:

    `V(s) = max( R(s, a) + γ * V(s') )` where R(s,a) is a reward function that outputs a reward when provided an action for the current state s.

    Btw, this equation is known as the "Bellman Equation" and is one of the most foundational building block of reinforcement learning.

    $$V(s) = \max_a(R(s,a) + \gamma V(s'))$$

    Modelling your world, states, actions and finding meaningful rewards for your problem in reinforcement learning is an art form. Chess is an extremely studied game and rewards for different states are generally standard and explored. Here is a pretty standard reward table for the game of chess:

```
chess_reward_table = {
    # Material values (traditional piece values)
    'piece_values': {
        'pawn': 1.0,
        'knight': 3.0,
        'bishop': 3.0,
        'rook': 5.0,
        'queen': 9.0
    },

    # Positional rewards
    'position_rewards': {
        'control_center': 0.2,     # Bonus for controlling center squares
        'connected_rooks': 0.3,    # Bonus for connected rooks
        'bishop_pair': 0.3,        # Small bonus for having both bishops
        'doubled_pawns': -0.2,     # Penalty for doubled pawns
        'isolated_pawns': -0.2,    # Penalty for isolated pawns
    },
```

```
    # Development rewards (early game)
    'development': {
        'piece_developed': 0.1,    # Piece moved from starting square
        'king_castled': 0.5,       # Successfully castled king
        'controlling_open_file': 0.2  # Rook on open file
    },

    # King safety
    'king_safety': {
        'pawn_shield': 0.3,        # Pawns protecting king
        'king_exposed': -0.4,      # Penalty for exposed king
    },

    # Game ending states
    'game_end': {
        'checkmate_win': 100.0,    # Winning the game
        'checkmate_loss': -100.0,  # Losing the game
        'stalemate': 0.0,          # Draw
    }
}
```

I want you to really understand that these rewards are extremely important to future algorithms and optimizations. They really help guide and speed up search exponentially. In the above example the reward table can understood like:

R(s, a) = +100 for checkmates, +9 for capturing the enemy's queen, and +9.3 (9+0.3) for capturing a queen and connecting 2 rooks together.

13. We have an updated equation with *rewards* — $V(s) = \max(R(s, a) + \gamma * V(s'))$ does this make our search more efficient? No! Not yet. Why not? Because to calculate the state's value we still have to traverse the practically infinitely massive tree of states (like chess board states).

    This problem of too many (sometimes infinite) states and actions is the heart of all reinforcement learning.

    Reinforcement learning problems can't be *"solved"* but can be *"estimated".* From now on, we will learn about the most efficient and accurate ways researchers have found to estimate problems (never really solve).

14. Okay, so the tree can't be fully traversed but what if we partially traversed a state tree by pruning or picking only "*sensible*" actions. In reinforcement learning this is called "exploration vs exploitation".

    Exploration: When learning in an uncertain environment, an agent needs to try new, potentially suboptimal actions to discover better strategies it might have missed.

Exploitation: Once an agent has discovered reliable strategies, it can exploit this knowledge by repeatedly choosing actions that lead to rewards.

But why exploration? The reason why we need exploration (and try potentially suboptimal actions) is because sometimes taking suboptimal short-term actions leads to better long-term outcomes. Examples:

a. Navigation: A longer initial route might connect to a highway that's ultimately faster than the obvious shorter path
b. Dota2: Higher initial costs for expensive items often yield better long-term returns than can win you the game.
c. Chess: Sacrificing a high-value piece (seemingly suboptimal) can lead to checkmate sometimes.
d. Game Theory: In multi-agent scenarios, appearing unpredictable through occasional suboptimal moves can prevent opponents from exploiting fixed strategies

The exploration-exploitation dilemma is fundamental to learning - like a pathfinding algorithm for road networks balancing between routing through established highways (exploitation) and investigating side streets and alternate routes (exploration). Too much exploration wastes computational effort testing every possible road combination, while pure exploitation might miss that optimal route through local roads that bypass congested highways entirely.

15. Something I want to talk about before we move on to more complex algorithms — *Q(s,a) function* also known as the action-value function!

    Q(s,a) represents state-action value - value at starting from state s, taking action a:

    Q(s,a) = R(s,a) + γ * V(s')

    reminder that V(s) = max(R(s, a) + γ * V(s'))

    which means: V(s) = max(Q(s,a)) for all actions a that can be taken at

    (remember these 3 equations, they are pretty fundamental to rl)

    Why do we need Q(s, a)? Why can't we just use V(s) everywhere? — The reason is obvious but it's never addressed in papers, books and research. Reinforcement learning primarily comprisies of algorithms and code-style logic expressed as mathematical equations. In translating algorithmic logic to mathematical notation, we need intermediate variables (like Q and V) to represent different computational steps. Just as code uses multiple variables to break down complex logic, RL math uses Q(s, a) and V(s) to express the process of evaluating actions and selecting maximums. These functions are derivatives of each other similar to how variables in code build on each other to express program logic.

    This is where I have a fundamental problem with the way reinforcement learning is taught and studied & is also partially why I wanted to write this blog in my learning style. Anyway, my rant is over let's move on.

16. Now that you know about exploration and exploitation, let's talk about an algorithm called *Real-time dynamic programming (RTDP)*:

    In RTDP, exploration is typically done using an epsilon-greedy algorithm where:

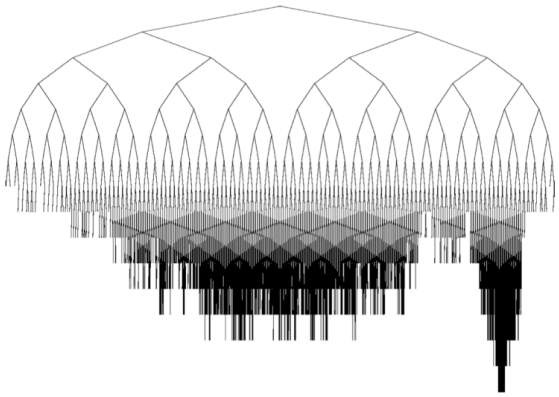
    a. With probability ε: Choose a random action

    b. With probability (1-ε): Choose the action that maximizes the expected value

    The *"exploration rate ε"* usually starts high (like 0.9) and decays over time, allowing the algorithm to:

    a. Initially explore broadly (because of the high starting exploration rate) to avoid local optima

    b. Eventually, exploit known good paths. As the *exploration rate* decays towards 0, we essentially take action greedily. We do this because over time our confidence in value estimates increases.



notice how initially the tree is wide but then as the *exploration rate* decays the tree exploits only the good nodes

Let's quickly read some pseudocode & find issues with this algorithm. Look at the code below:

```
def rtdp(state_space, actions, initial_state):
   V = {s: 0 for s in state_space}
   epsilon = epsilon_start

   gamma=0.99 # discount factor from bellman equation
   epsilon=0.9 # exploration rate, ε as described above
   max_episodes=1000 # number of iterations
   epsilon_decay = 0.995 # epsilon decay amount after each episode

   for episode in range(max_episodes):
       state = initial_state
       while not terminal(state):
           # next action decided using epsilon-greedy method
```

```
            best_action = argmax(actions, lambda a: V[get_next_state(state,
            next_action = random_action() if random() < epsilon else best_a
            next_state = transition(state, next_action)
            # Bellman update
            max_future_value = max(V[get_next_state(state, a)] for a in act
            V[state] = reward(state,next_action) + gamma * max_future_value
            state = next_state
        epsilon *= epsilon_decay
    return V
```

The code explained:

a. We run the for loop *max_episodes* times. I realize now that I haven't spoken about the term *"episodes"* yet in the context of reinforcement learning. In RL an iteration (in our case one pass of our tree) is called an episode. So next time you see the episode, think of one iteration of the algorithm.

b. For each episode, we start from the initial state and explore our tree until we hit a terminal state. Initially, our tree only comprises of the root node (the initial state) but after each iteration,n the tree grows as we compute values of new states.

c. We now compute the `next_action` in an epsilon-greedy manner. If the `random()` returns a value < 0.9 (our epsilon value) then we select a random action and if it's above 0.9 we select the best action. This means that there is a 90% chance we explore a new path and a 10% chance we exploit the nodes in our tree already. After we have the next action we then compute the `next_state` by performing that action.

d. We then update the value of the state using the bellman equation we learnt about (but, with a small modification, i.e., the `next_action` is already chosen.)

e. After this the `state` is updated to the `next_state`. we move on to the next state and then repeat steps a-d.

f. Once we reach the terminal state of our explored tree, we can update the exploration rate using the decay. Notice that the epsilon after 1000 episodes would be 0.9 * 0.995^1000 = 0.006. This means that towards the final episodes we are barely exploring (0.06% of the time) and are exploiting the states with the highest ROI.

   A chess game equivalent of the above statement is something like — after enough episodes the algorithm has largely crystallized its strategy, almost always selecting moves it's learned lead to advantageous positions rather than experimenting with novel approaches.

The problems faced by *RTDP:*

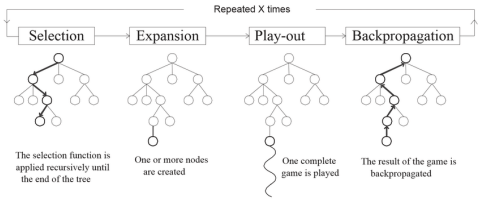
a. Initial values - Performance heavily depends on initial value function estimates

b. Coverage - May ignore relevant states if they're not frequently encountered

c. Exploration decay - Hard to set an optimal decay rate; too fast means missing important states, too slow wastes computation

17. Solving RTDP problems using Monte Carlo Tree Search (MCTS):

    The core idea behind MCTS is that instead of relying on a decaying exploration rate like RTDP does, we use the actual *value* scores of the nodes to make better decisions.



    MCTS has 4 stages:

    a. Selection Stage:

        During the first stage of the MCTS algorithm, we select a path based on a formula called the Upper Confidence Bound (UCB) scores. For each child node, we compute the UCB scores as follows:

        UCB = Q(s,a) + c * √(ln(N) / n)

        - *Q(s,a)* is the average value (quality) of taking action 'a' from state 's'
        - *N* is the total number of visits to the parent node
        - *n* is the number of visits to this specific child node
        - *c* is an exploration constant

        During tree traversal, we compute the scores for each child node. Sounds fancy, but it's intuitive and simple. The first term of the formula, Q(s,a), says "go where we've found success before" (exploitation). The second term, c * √(ln(N) / n), says "explore paths we haven't tried much" (exploration). What makes this formula clever is how it automatically balances two competing needs: exploiting moves we know are good (the Q(s,a) term) and exploring moves we haven't tried much (the square root term). The more we visit a node, the smaller the exploration bonus becomes (because the denominator n increases), naturally shifting our focus to the most promising moves.

        When a child node hasn't been visited yet, the `ln(N)/n` term is infinity because n for that node is 0. In simpler words, when a node has a child node that is also a leaf node (unvisited) we select that node to exploit in future stages.

        The exploration constant (c) is generally set to sqrt(2) why?

        The choice of c = √2 in the UCB formula is rooted in a probability theory concept called the Hoeffding inequality (I think so). The Hoeffding inequality helps us understand how far a random variable might deviate from its expected value. When we're exploring in MCTS, we're essentially trying to estimate the true value of each move based on limited samples.

We want to be confident that our estimate is reasonably close to the true value.
The value √2 comes from setting our confidence level for these estimates. I'm going to skip a bunch of math here but, √2 gives us what's called a "95% confidence interval" - meaning we can be 95% confident that our estimate is close to the true value.

Think of it like this: Using √2 means we're making an assumption that exploration decisions that will be good choices 95% of the time.

Let's look at the selection stage's pseudocode:

```
class Node:
    def __init__(self):
        self.visits = 0
        self.value = 0
        self.children = {}

def ucb_score(parent, child, c=1.414):
    if child.visits == 0:
        return float('inf')
    return (child.value/child.visits) + c * sqrt(log(parent.visits)/chi

def select(node):
    while node.children:
        child_scores = softmax([ucb_score(node, child) for child in noc
        node = select_random_child_based_on_softmax_ucb_scores(child_sc
    return node
```

Imo, the code is so simple & I'm not going to go through with it. But here are some important things to remember from this stage of MCTS —

- `float('inf')` is selected when a child hasn't been visited.
- at each level of the tree we compute the scores and then choose a random child node based on the softmax of scores.

b. Expansion Stage:

After we've selected a leaf node during the selection stage, we need to grow our tree. During expansion, we create a new child node based on available actions. Here's what makes this stage interesting - we don't usually expand all possible children at once. Instead, we add just one child node per expansion. This helps us manage our memory efficiently while still exploring the space of possibilities.

Keeping it simple like the selection stage:

```
def expand(node, possible_actions):
        # find unexplored actions
```

```
        unexplored_actions = [
        action for action in possible_actions
        if action not in node.children
    ]

    # if we have unexplored actions, randomly pick one
    if unexplored_actions:
        chosen_action = random.choice(unexplored_actions)
        node.children[chosen_action] = Node()
        return node.children[chosen_action]

    return node
```

I don't think this code needs any explaination, we select an *action* that hasn't been performed and returns a new node (the new *state*) after performing that *action*.

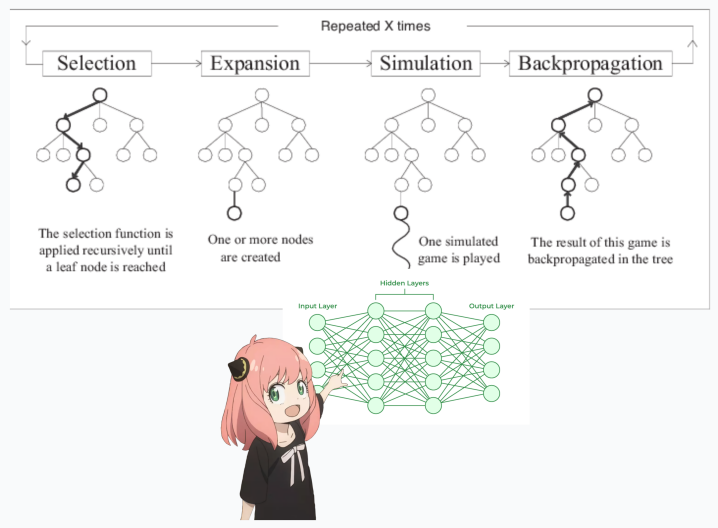
c. Playout Stage:

This is where MCTS gets really clever. After expansion, we perform what's called a "playout" or "simulation" from our newly created node. Imagine you're playing chess - instead of carefully analyzing every possible move, you just play random moves until the game ends. That's essentially what we're doing here!

During the playout, we follow a simple simulation of moves (often random) until we reach a terminal state or hit a depth limit. This gives us a rough estimate of how good our position is. While it might seem too simple to just play randomly, this approach has a fascinating property: over many iterations, these random playouts actually give us meaningful information about which moves are generally better.

Did AlphaGo simulate a game using random moves? Does that make sense?

Okay, I lied. The simulation is not often random! AlphaGo didn't use random moves in the simulation stage. Instead of random playouts, it used its value network to directly evaluate board positions. This value network was trained on millions of positions and could estimate the winning probability without needing to play out random moves. This was more efficient and accurate than traditional MCTS's random simulations since Go's massive branching factor made random playouts less reliable in evaluating positions. The image below is kinda what they did:




```
def playout(node, max_depth=50):
    current_node = node
    depth = 0
    total_reward = 0

    while not is_terminal(current_node) and depth < max_depth:
        action = random.choice(get_possible_actions(current_node))
        reward = simulate_action(action)
        total_reward += reward
        depth += 1

    return total_reward
```

d. Backpropagation Stage:

   The final stage is where we take what we learned from our playout and update our knowledge throughout the tree. Remember that value Q(s,a) we used in the UCB formula during selection? This is where that value gets updated.

   During backpropagation, we take the result from our playout and propagate it back up through all the nodes we visited during selection and expansion. Each node updates its visit count and total value. This is similar to how neural networks update their weights through backpropagation but much simpler.

```
def backpropagate(node, reward, path):
    for node in path:
        node.visits += 1
        node.value += reward
```

```
            # Some implementations use (reward - node.value) / node.visi
ts
```

The intuition behind this stage of MCTS:

- Good paths get higher values, making them more attractive in future selections
- More visited paths get lower exploration bonuses (because we do `node.visits += 1`)
- The UCB formula uses this backpropagated information to balance between proven good moves and unexplored possibilities

The real power comes from repetition - the more we play through paths, the more accurate our value estimates become. Bad moves naturally get avoided over time as their values stay low, while good moves get visited more frequently, refining our understanding of their true worth.

What makes this stage powerful is how it gradually builds up our knowledge of the game tree. Nodes that consistently lead to good outcomes will accumulate higher values, while nodes that lead to poor outcomes will have lower values. This information then feeds back into our selection stage through the UCB formula, creating a beautiful cycle of exploration and exploitation.

Advantage 1 of MCTS over RTDP with an example of chess:

In RTDP, imagine each position in the chess tree has a score written on it that stays there between moves. When a position initially looks bad (like sacrificing your queen), it gets a low score. The only way to change this score is if RTDP randomly decides to explore this "bad" position due to its epsilon-greedy exploration. Since RTDP usually picks the highest-scoring moves (that's the "greedy" part), it might rarely visit this position again. The low score might stick around for a long time, preventing RTDP from discovering that this position actually leads to a win.

In MCTS, there are no permanent scores on the tree positions. Every time MCTS runs, it builds its understanding of the position from scratch through simulations. If sacrificing the queen leads to checkmate, MCTS can discover this anytime it explores that branch, unaffected by any previous evaluations. It doesn't need to get "lucky" with random exploration to overcome a bad initial score because it runs a full simulation every time!

Advantage 2 of MCTS over RTDP again in an example of chess:

Consider a chess position where most moves look bad but one subtle move wins. RTDP might waste a lot of time randomly exploring obviously bad moves, while MCTS would quickly focus on the promising line once it discovers it, while still occasionally checking other options to make sure it hasn't missed anything.

Advantage 3 of MCTS over RTDP while solving a maze:

RTDP would have to actually try walking down each path to learn about it, while MCTS can "imagine" walking down paths through simulation to get a rough idea of which ones might lead to the exit. This ability to look ahead through simulation helps MCTS make better decisions with limited computation time.

Problems with MCTS using examples that I like:

- Memory Issue: In Starcraft 2, each unit can take multiple actions, and there are often 200+ units. Even storing just the first few levels of possible moves requires gigabytes of memory.
- Computational Cost: In Chess, if each move needs 1000 simulations for accurate evaluation, and you have 30 possible moves, that's 30,000 simulations just to evaluate one position.
- Simulation Quality: In Go, random playouts often miss subtle positional advantages. A seemingly poor move might actually be brilliant, but random play won't reveal this because it lacks strategic understanding.
- Terminal State: In some games like Dota2, reaching a terminal state through simulation is impractical - games can last hours and thousands of moves. This makes it hard to get reliable win/loss feedback for MCTS backpropagation, reducing the algorithm's effectiveness.

18. I want to pause and reflect on everything we understand so far. Idk if you noticed but all of the algorithms, formulas and examples we spoke about so far only talk about learning by experience (and computing a value table) or search and simulations (like in MCTS). But, imagine you're Netflix building a movie recommendation system. You have millions of users watching different movies, but each user only interacts with a tiny fraction of your catalogue. The challenge: how do you learn what movies they might like without forcing them to watch everything?

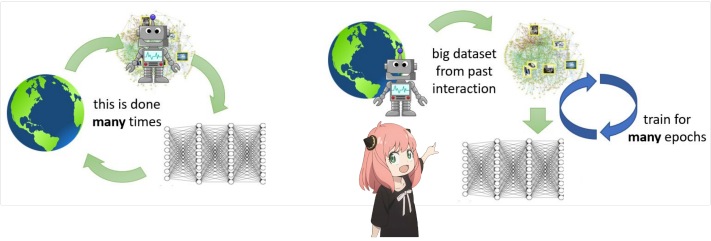
    You should have guessed that this section was coming, we haven't discussed a single algo that uses a dataset or external human feedback.

    Netflix serves 250+ million subscribers (or something idk), with a catalogue of thousands of movies. Each user watches maybe 0.1% of available content. Training a recommendation system through pure exploration (like RTDP) would mean forcing users to watch random movies - terrible UX lol.

    This brings us to a critical distinction in reinforcement learning: on-policy vs off-policy learning.

    a. *On-policy algorithms* (like basic RTDP) learn only from their own experiences and data that they create on their own.

    b. *Off-policy methods* can learn from any datasets - whether it's Netflix viewing history, robots or language models like ChatGPT.



REMEMBER THIS:

on-policy algorithm means the algorithm learns by playing/experiencing the world on its own.

on-policy algorithm means the algorithm learns by playing/experiencing the world on its own.

on-policy algorithm means the algorithm learns by playing/experiencing the world on its own.

on-policy algorithm means the algorithm learns by playing/experiencing the world on its own.

on-policy algorithm means the algorithm learns by playing/experiencing the world on its own.

on-policy algorithm means the algorithm learns by playing/experiencing the world on its own.

off-policy *algorithm means learning from datasets, human or external feedback.*

off-policy *algorithm means learning from datasets, human or external feedback.*

off-policy *algorithm means learning from datasets, human or external feedback.*

off-policy *algorithm means learning from datasets, human or external feedback.*

off-policy *algorithm means learning from datasets, human or external feedback.*

off-policy *algorithm means learning from datasets, human or external feedback.*

19. Okay, let's try and build a Netflix recommendation algorithm (can be extended to Youtube, Tiktok, Instagram etc). My assumptions here — we have a massive dataset of people's preferences and profile.

    What if we just use a giant table? For each user and movie, we store a number representing how likely they are to enjoy it. Seems simple enough, right?

    Q(user, movie) = How much user will like the movie?

    Where Q is a function that returns a score (between -1 and 1) indicating the degree to which someone likes/dislikes a movie. A score of -1 would mean they hate the movie and a score of 1 means they love it!

    Btw, *user state* contains — what they've watched, how long they watch, when they watch, their demographics, device usage, and viewing habits like "watches on weekends" or "likes action movies".

    You can already tell that the possible user states can be infinite, but let's ignore it (for now)!! :)

    Also, when someone watches a movie, we update our prediction based on:

    - Did they finish the movie?

- Did they like similar movies?
- What rating did they give?

Our goal right now is to maximize Q(user, movie), in order words, we want to find a move (this is an action in our reinforcement learning context) that the user will like most!!!

One issue with our current *quality* function (Q) is that we only take 2 parameters (user and movie). That doesn't help!! Let's take the example of a Q table below:

Q table	Interstellar	Inception	The Matrix	The Notebook	Love Actually
user_a	0.9	0.85	0.8	-0.4	-0.3
user_b	0.8	0.9	0.7	-0.2	-0.1
user_c	-0.2	-0.3	-0.1	0.9	0.8
user_d	0.1	0.2	0.3	0.7	0.6

Looking at this table, we can see clear patterns:

- Users A and B love complex sci-fi (high scores for Interstellar/Inception)
- User C strongly prefers romance movies
- User D is more neutral, slightly preferring lighter movies

Now that we have this data, let's say a new *user_e* creates an account, watches *Interstellar* and then rates it 5/5. Intuitively, we would recommend *Inception* and *The Matrix* to them.

Think about how you decide if you'll like a movie. You probably consider many different aspects: the genre, actors, director, release year, and even more subtle things like the movie's pacing or visual style. These are all features that will influence your decision. This is good point to talk about *features* in reinforcement learning. Instead of just having Q(user, movie), we could expand this to include all these features. Our Q function would look more like:

Q(user_features, movie_features) = How much user will like the movie?

Now our Q table becomes an n-dimensional array and the shape of this Q table matrix would be (count(user_features), count(movie_features)). This is where it gets interesting! When user_e watches and loves *Interstellar*, we're not just learning "this user likes Interstellar" - we're learning they might like features of this movie and recommend:

- Other sci-fi movies
- Other Christopher Nolan films
- Movies with similar visual effects
- Movies with similar pacing
- Movies released in a similar time period

Now, let's look at how we can learn from our data, take this info for example:

```
user_a = {
    'age': 26,
    'preferred_genre': 'sci-fi',
    'preferred_watch_time': 'evening',
}

interstellar = {
    'genre': 'sci-fi',
    'duration': 169,
    'director': 'Christopher Nolan',
}
```

So let's look at how our Q matrix can be updated, let's initialize all values in the Q matrix to be 0 (neutral on all movies):

a. In step 1, our Q value is 0, i.e., Q(user_a_features, movie_interstellar_features) = 0

b. To be more exact our Q matrix will look something like:

   Q[user_feature_1_index, movie_feature_1_index] = 0

   Q[user_feature_1_index, movie_feature_2_index] = 0

   Q[user_feature_1_index, movie_feature_3_index] = 0

   Q[user_feature_2_index, movie_feature_1_index] = 0

   Q[user_feature_2_index, movie_feature_2_index] = 0

   Q[user_feature_2_index, movie_feature_3_index] = 0

   Q[user_feature_3_index, movie_feature_1_index] = 0

   Q[user_feature_3_index, movie_feature_2_index] = 0

   Q[user_feature_3_index, movie_feature_3_index] = 0

   and so on... hopefully this is obvious to you now — the entire matrix is initialized to 0.

c. In step 2, let's say we want to know if 26 year olds like *Interstellar*'s genre (sci-fi) by analyzing how user_a reviewed *Interstellar*'s. I really want you to read that sentence like 3 times :)

   Okay so if we want to learn from *user_a's* rating, we would update the Q table with something like:

   Q(age=26, sci_fi) = probably some function of the rating that *user_a* provided to *Interstellar*.

   If a 26 year old end's up loving the movie alot more than the avg person (rating = 5/5), you'd want to adjust your score upward for similar movies. Similarly, if they hate it more than the avg person (rating = 0/5), you'd want to adjust downward for similar movies.

   The most intuitive way to learn this feature would be:

Q(age=26, sci_fi) = Q(age=26, sci_fi) + α * (avg_rating_of_Interstellar - user_a_rating_of_Interstellar)

Why does α exist? — Think about how humans learn from experience. When we watch a movie and either love it or hate it, we don't immediately assume all similar movies will provoke exactly the same reaction. Instead, we gradually adjust our expectations. The *learning rate α* captures this idea of gradual learning.

If α = 1, we would completely override our previous knowledge with each new rating. This would be like completely changing your opinion about sci-fi movies based on just one movie experience. For example, if a 26-year-old really loves Interstellar (rating it much higher than average), we'd immediately assume all 26-year-olds love all sci-fi movies just as much.

If α = 0, we wouldn't learn anything at all from new experiences. Our Q-values would stay at their initial values forever, making our recommendations static and unresponsive to user feedback.

d. In step 3, not only do we repeat this process for every single 26 year old that reviewed Interstellar but we do this for all combinations of user_features and movie_features for every single row in our dataset.

   If a new *user_e* enters the system:

```
user_e = {
    'age': 35,
    'preferred_genre': 'romance',
    'preferred_watch_time': 'undecided'
}
```

   then, we can compute the Q scores for this new *user_e*:

```
q_value = (
        # How much do 28-year-olds like this genre?
    Q(age=35, movie_x_genre) +
    # How well does this match with romance preference?
    Q(romance, movie_x_genre) +
    # No strong signal from watch time preference
    Q(undecided, movie_x_watch_duration)
)
```

   for the movies *Interstellar* and *The Notebook*:

```
# Interstellar recommendation calculation
interstellar_score = (
    Q(age=35, scifi)      # Might be slightly negative based on demogra
```

```
    + Q(romance, scifi)   # Likely negative - genre mismatch
    + Q(undecided, 169)   # Neutral score for duration
)

# The Notebook recommendation calculation
notebook_score = (
    Q(age=35, romance)     # Might be positive based on demographic
    + Q(romance, romance)  # Strong positive - direct genre match
    + Q(undecided, 123)    # Neutral score for duration due to undecide
)
```

Using our earlier Q-table as reference, we'd probably see something like:

1. *The Notebook* and *Love Actually* would get high scores, mainly driven by the strong genre match with *user_e's* romance preference
2. I*nterstellar, Inception,* and *The Matrix* would get lower scores due to the genre mismatch, even though the age demographic might typically enjoy these films

Btw, for when this user_e rates a move we can update the Q values again, which makes for better reocmmendations for other users.

The *genralized* equation for this enitre process can be the following:

Q(s,a) = Q(s,a) + α * (user_rating - Q(s,a))

20. There are some very glaring obvious issues with our Q matrix Netflix recommendation system algorithm that I will eventually come to! But, I want you to consider this Q matrix update method for a chess game engine. Imagine you're building an Q table for a chess engine.

    How do we represent a chess position to our matrix based Q learning algorithm? You might think we could simply use the complete board state - the position of every piece. However, this approach runs into a serious problem: the space of possible chess positions is so vast (around 10^43 positions) that we'd almost never see the exact same position twice in our training data.

    This is where feature engineering becomes crucial in reinforcement learning. Instead of raw board states, we need to extract meaningful features that capture the essential characteristics of a position for example:

    - Material count and distribution
    - Piece activity and mobility
    - King safety metrics
    - Pawn structure characteristics
    - Control of key squares and files
    - Piece coordination metrics
    - Tempo and development indicators

By representing positions through these features, we can start recognizing similar strategic situations even when the exact piece positions are different. For example, two positions might be very different in their exact piece placement but share key characteristics like "bishop pair advantage with weak dark squares in opponent's side."

So from now on when you see a *state, s* think of a feature map and when you see a matrix Q think of these features getting mapped to possible moves (Just like user features getting mapped to movies in the Netflix example).

Okay, let's continue building our chess engine. At first glance, our simple formula for Netflix recommendations might seem promising for chess. After all, we have huge databases of chess games - millions of matches with ratings, outcomes, and moves. Just like we learn user preferences for movies, couldn't we learn preferences for chess positions? Consider this formula for now:

Q(s,a) = Q(s,a) + α * (game_outcome - Q(s,a))

Let's say we're looking at a position where White has a knight sacrifice opportunity. For this example, lets say our features tell us:

a. Piece material is even (0.0)
b. White has slightly better piece activity (+0.3)
c. Black's king is somewhat exposed (+0.2)

If White wins this game, the outcome is +1. With a learning rate α of 0.1, we'd update our Q value:

Q(s, knight sacrifice) = 0.5 + 0.1 * (1.0 - 0.5) = 0.55 (remember this value, I'll compare it below)

But this update misses something crucial. After the knight sacrifice, we enter a new position where:

- White is down material (-3.0)
- Now the black's king might be very exposed (+1.5)
- White has strong attacking chances (+2.0)

Using our simple Netflix style formula, Q(s,a) = Q(s,a) + α * (game_outcome - Q(s,a)) — we might learn from the database that this sacrifice leads to a win 70% of the time. Seems good, right?

But here's where things get fascinating - this formula is critically flawed for chess, even with all that data. The problem is that our formula only considers the immediate position and final outcome. It completely misses the crucial chain of moves — Consider a 40-move chess game. Let's say White makes a brilliant move on move 10, but then makes a terrible blunder on move 35 and loses. If we only use the final outcome to learn, we're giving that brilliant move on move 10 a negative update - we're saying "this move led to a loss." But that's not really fair or accurate! The move on move 10 was great, and the game was only lost much later due to a

completely unrelated mistake. This is why our Netflix style formula with just *game_outcome* is highly flawed for games and environments with a sequence of events.

This is where we desperately need a future-looking term like in this formula:

$Q(s,a) = Q(s,a) + \alpha * [R + \gamma * \max(Q(s',a')) - Q(s,a)]$

Btw, this formula & this Q matrix method of learning from data is known as Q learning (specificly in our case Tabular Q learning).

VERY IMPORTANT NOTE: It's not obvious to many people, but when learning from a dataset. The dataset needs to contain the following information — current state(s), action(a) that was taken at current state(s) & finally the new state(s') that the action(a) resulted in. The max function in the formula including all future iterations of Q-learning take the new state s' from the dataset and find an action a' that maximizes the value (Q(s',a')) for the state s'.

Also, $R + \gamma * \max(Q(s',a'))$ is known as the target value in the context of RL. This makes sense, it's kinda like the true outcome

Now when we update our Q-matrix, we consider not just the final outcome, but the best possible follow-up moves. If γ is 0.9 and we see that the resulting position offers strong attacking moves valued at +2.0, our update becomes:

$Q(s, \text{knight sacrifice}) = 0.5 + 0.1 * (0 + 0.9 * 2.0 - 0.5) = 0.63$ (which is >0.55 that we calculated above)

This tells us that the sacrifice position is even better than we thought, not just because the game was eventually won, but because it leads to positions with concrete winning chances. Over time, our Q-table learns not just which positions are good, but which positions lead to other good positions.

21. Now we are cooking! There are 2 MASSIVE issues with our tabular Q-learning approach:
    a. First, we might miss subtle patterns that don't fit neatly into our pre-defined features. Think about a position where a seemingly small detail - like whether a knight is on chess board square e4 or f3 - completely changes the evaluation. Our simple features might not capture this nuance.
    b. Second, our features don't naturally combine with each other. In real chess, the value of piece activity deeply interacts with king safety, which interacts with pawn structure, and so on. With tabular Q-learning, we're treating these features somewhat independently.

    If only there was a way to let the computer discover patterns by itself and combine features into learnt patterns :) LMAO this is deep learning 101.

    Anyway, this brings us to another improvement to Q-learning, using neural networks! Instead of storing Q-values in a table, we train a network to predict them. This is also known as Deep Q-Learning. Sounds simple enough, let's give it a shot!

    The key insight here is replacing the Q-table with a neural network that maps states to Q-values for each action. Instead of manually engineering features, the network learns them

automatically.

In chess terms, a very simple version:

- Input: Raw board position (e.g., 8×8×12 tensor representing piece positions)
- Output: Q-values for all possible moves from that position
- Hidden layers: Hopefully automatically discover patterns like "bishop pair advantage" or "exposed king"

Let's think about how we can modify the Tabular Q-learning formula — $Q(s,a) = Q(s,a) + \alpha * [R + \gamma * \max(Q(s',a')) - Q(s,a)]$. Instead of storing Q-values in a table, we use a neural network that takes in a state and outputs Q-values for each possible action. Let's call this network $Q(s,a;\theta)$, where θ represents our network weights.

We can train this network using the same principle as before! For any state-action pair (s,a), we want:

$Q(s,a;\theta) \approx R(s,a) + \gamma * \max(Q(s',a';\theta))$. It's like saying "this move (*action, a*) should be worth a score of taking the action, $R(s,a)$ + what I think I can get in the future, $\gamma * \max(Q(s',a';\theta))$"

So how do we train this? Let's derive our loss function:

a. First, what's our target value?

   $y = R + \gamma * \max(Q(s',a';\theta))$ This is what we WANT our network to predict.

b. What's our current prediction?

   It's $Q(s,a;\theta)$. This is what our network ACTUALLY predicts.

c. The difference between these is our error:

   $error = (y - Q(s,a;\theta))$

   note: This error is also known as Temporal Difference error (TD error).

d. We square this error to get our loss (making it positive and penalizing big errors more):

   $L = (y - Q(s,a;\theta))^2$

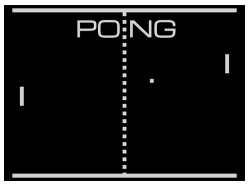
e. Now for the weight updates! Using gradient descent (this is standard deep learning):

   $\theta_{new} = \theta_{old} - \alpha * \partial L/\partial\theta$, where α is the learning rate

f. There's a subtle issue here:

   When we use the same network for both our prediction $Q(s,a;\theta)$ and calculating the future value $\max(Q(s',a';\theta))$, we create a moving target problem. Think about it: every time we update our network to make $Q(s,a;\theta)$ closer to the target, we also change the target itself because it depends on the same network!

To show you exactly why this is a problem, let's say we are training a DQN to play Pong. We're in a state s where our paddle is in the middle and the ball is moving towards us. Our network Q(s,a;θ) needs to decide between moving UP or DOWN.



Let's say in our example, moving UP is actually the right choice that leads to hitting the ball (state s'). Without a target network, here's what happens:

- Step 0:

  Initially, with random weights θ our neural network outputs the following scores:

  ```
  Q(s, UP; θ) = 0.2
  Q(s, DOWN; θ) = 0.3
  # right now our network says DOWN is the better move
  ```

- Step 1:

  Let's say we are computing the loss for Q(s, UP; θ) and reach new state s'. In this new state, our network again needs to predict values for both actions, because the formula computes max(Q(s', actions; θ)):

  ```
  Q(s', UP; θ) = 0.5
  Q(s', DOWN; θ) = 0.4
  # s' is the new state after taking action UP
  ```

  So max(Q(s', actions; θ)) = 0.5 (the UP action)

  Now we calculate our target value:

  ```
  target = R + γ * max(Q(s', actions; θ))
         = 0 + 0.9 * max(0.5, 0.4)
         = 0.9 * 0.5
         = 0.45

  # Computing loss for our action UP:
  loss = (R + γ * max(Q(s', actions; θ)) - Q(s, UP; θ))²
           = (Target - Q(s, UP; θ))²
       = (0.45 - 0.2)²
  ```

```
      = (0.25)²
      = 0.0625
```

Let's say we update our weights θ to make Q(s, UP; θ) closer to the target value of *0.45* (using θ_new = θ_old - α * ∂L/∂θ)

- Step 2:

  Because we updated our network weights, ALL Q-values change. Let's say that now in state s':

  ```
  Q(s', UP; θ) = 0.7 (increased from 0.5)
  Q(s', DOWN; θ) = 0.6 (increased from 0.4)
  ```

  Now, think of a scenario where a similar state to our original state s, is encountered in our training data. For that new state, the target value would look something like:

  ```
  new_target = R + γ * max(Q(s', actions; θ))
             = 0 + 0.9 * max(0.7, 0.6)
             = 0.9 * 0.7
             = 0.63

  # new loss
  loss = (New Target - Q(s, UP; θ))²
           = (0.63 - 0.3)²
           = (0.33)²
           = 0.1089
  ```

  You can see for the exact same state s (or similar state) the loss value of the same action is now alot higher. This creates an upward spiral of constantly increasing value estimates. When you calculate the target network with the same Q network as predictions, your model starts fogetting everything it has learnt and this is known as catastrophic forgetting.

  note: This is not the only type of DQN scenario in which we see catastrophic forgetting.

How do we fix this problem?

Instead of using the same network θ for both prediction and target calculation, we create a copy of our network with weights θ' that we update less frequently. This second network is called our target network. Let's see how this changes our training:

- Step 0 after target network:

  We start with two networks - our main network θ and target network θ', let's say with random weights we get this value:

```
# Main network (updated every step)
Q(s, UP; θ) = 0.2
Q(s, DOWN; θ) = 0.3

# Target network (updated less frequently)
Q(s, UP; θ') = 0.2  # Initially same as main network
Q(s, DOWN; θ') = 0.3
```

- Step 1 after target network:

  When calculating our target value, we now use the target network θ' instead:

```
# Target network predicts future values
Q(s', UP; θ') = 0.5
Q(s', DOWN; θ') = 0.4

target = R + γ * max(Q(s', actions; θ'))
       = 0 + 0.9 * max(0.5, 0.4)
       = 0.45

# But we still use main network for current state prediction
loss = (target - Q(s, UP; θ))²
     = (0.45 - 0.2)²
     = 0.0625
```

- Step 1 after target network:

  When we update our weights, only the main network θ changes. The target network θ' stays stable:

```
# Main network values change after update
Q(s', UP; θ) = 0.7
Q(s', DOWN; θ) = 0.6

# Target network values remain the same
Q(s', UP; θ') = 0.5
Q(s', DOWN; θ') = 0.4
```

  Now when we encounter a similar state, our target calculation remains stable because it uses the unchanged target network:

```
new_target = R + γ * max(Q(s', actions; θ'))
           = 0 + 0.9 * max(0.5, 0.4)
```

```
                = 0.45  # Same target as before!
```

The key is that we only update the target network θ' periodically, typically every C steps (like every 10,000 steps). When we do update it, we simply copy all weights from our main network θ to our target network θ'. This creates a more stable learning target because the target values don't change every single step.
This approach is called "
Fixed Q-Targets" (the key innovation in the original DQN paper). The target network acts as a temporary anchor that stabilizes our learning process.
The trade-off here is that we're learning from slightly outdated target values (because θ' lags behind θ), but this is much better than the instability caused by constantly moving targets.
In practice, this technique significantly improves the stability and convergence of DQN training.

g. I went on a massive detour, but let's get back to deriving the loss function, our new loss function after the target network is introduced:

   y = R + γ * max(Q'(s',a')) where Q' is the target network

   $L = (y - Q(s,a))^2$

h. Now for the weight update we need to compute:

   θ_new = θ_old - α * ∂L/∂θ, where α is the learning rate

i. Using the chain rule to find ∂L/∂θ:
   we get,
   ∂L/∂θ = ∂L/∂Q * ∂Q/∂θ
   ⇒ ∂L/∂θ = -1 * 2(y - Q(s,a)) =
   -2(y - Q(s,a)), this is our gradient

   note: y contains Q', but since Q' is our target network, its parameters θ' are fixed during this update, so we treat y as a constant.

j. To help visualize why this makes sense, think about what this gradient tells us:

   If y > Q(s,a), the gradient is negative, meaning we should increase Q(s,a)
   If y < Q(s,a), the gradient is positive, meaning we should decrease Q(s,a)

   The factor of 2 comes from the square in our loss function, which makes the gradient larger when the error is larger

   This is exactly what we want: the gradient pushes our prediction Q(s,a) toward the target value y. The larger the difference between them, the stronger the push.

22. In our chess and pong examples, we learn from a sequence of actions and states that happened in the past (from a dataset). Learning from a sequence of states or game actions can have 2 issues:

a. Consecutive states are highly correlated, which can cause the network to overfit to recent experiences.

b. Second, we only see each experience once before discarding it, which is inefficient.

For example in chess, if you're in the middle of an attack on the opponent's king, many consecutive positions will be very similar. If you learn only from these similar positions in sequence, your neural network might overfit to this specific type of position and "forget" other important chess concepts it learned earlier.

There is a simple fix to this problem: instead of training from samples directly, we train randomly on a batch of samples from a buffer, it's easier to understand this by looking at some code:

```
# Problem: Learning from sequences has 2 issues
# 1. Similar consecutive states cause overfitting
# 2. Each experience only seen once

buffer = []
max_size = 10000

for episode in range(num_episodes):
   state = env.reset()
   while not is_end_state(state):
       action = agent.get_action(state)
       next_state, reward, done = env.step(action)

       # Not training one state at a time, this causes the issues we spoke
       agent.train((state, action, reward, next_state))

       # Store experience
       if len(buffer) >= max_size:
           buffer.pop(0)  # Remove oldest

       buffer.append((state, action, reward, next_state))

       # Train on random past experiences
       batch = random.sample(buffer, 32)
       agent.train(batch)

       state = next_state
```

This simple buffer mechanism solves both issues by randomizing training order and reusing experiences. Instead of learning only from consecutive similar states (like 50 frames of ball moving right in Pong), the network gets trained on a diverse mix of situations.

Also, the buffer that helps us randomize events is known as the Experience Repay Buffer.

There is an optimization to this method called Prioritized Experience Repay Buffer where high-priority experiences get sampled more frequently during training. The formula for priority is:

```
priority = |TD_error|^α + ε

where:
- TD_error = reward + γ * Q(next_state) - Q(current_state)
- α controls how much prioritization is used (α = 0 is uniform sampling)
- ε is a small constant ensuring non-zero sampling probability

# when we want to sample an item from the buffer we sample based on this p
P(i) = priority(i)^α / sum(priority(j)^α for j in buffer)
```

This helps the agent learn more efficiently by focusing on the most informative experiences while still maintaining some exploration through the ε term.

23. We've fixed quite a few issues with our DQN implementation, but there's are more problems. Here is another problem: our Q-values tend to be overoptimistic! Why? Because we're using the same network to pick actions AND evaluate them. It's like grading your own homework!

    Let's see exactly why this is a problem. Remember our target calculation:

    $y = R + \gamma * \max(Q'(s',a'))$

    The max operation is both selecting which action to take AND estimating its value. This leads to a systematic overestimation because estimation errors (which can be positive or negative) are more likely to be positive when we select the maximum. Let's look at why with a concrete example from Pong. Imagine your paddle is in the middle, the ball is coming downwards, and you need to decide between moving UP or DOWN:

    ```
    # State 1: Ball below paddle, moving downward fast on opponent side
    state1 = {
       'paddle_y': 0.5,     # Middle
       'ball_y': 0.4,       # Below paddle
       'ball_x': 0.9,       # Opponent side
       'ball_speed': 0.8,   # Fast
       'ball_direction': -0.8  # Heavy downward
    }

    Q_state1 = {
       'UP': 5.0, # Bad prediction - should move down to ball
       'DOWN': 2.0
    }
    ```

```
reward = 0  # No point

# State 2: Ball continued trajectory, paddle moved wrong way
state2 = {
   'paddle_y': 0.3,     # Went down correctly in the dataset
   'ball_y': 0.3,       # Ball continued down
   'ball_x': 0.85,      # Moving toward paddle
   'ball_speed': 0.8,
   'ball_direction': -0.8
}

Q_state2 = {
   'UP': 6.0, # Wrong values because model is bad right now
   'DOWN': 3.0
}

# updating the model
# target is calculated using state 2 Q values
target = 0 + 0.99 * 6.0 = 5.94
# since the target is actually worse then the already bad state 1 predicti
# our Q update for the action UP actually get worse
Q_state1['UP'] = 5.0 + 0.1 * (5.94 - 5.0) = 5.094
```

See what happened? If our initial model is terrible or untrained, we stop learning anything. This is because our target estimation itself is biased or terrible. The model learns from itself in a way, which makes the model worse. This creates a feedback loop of increasingly optimistic predictions.

Why does this happen? We're using the same network for two different jobs:

1. Picking which action looks best
2. Telling us how good that action actually is

A good analogy here would be like asking a super enthusiastic friend "which restaurant should we go to AND how good is it?" They might say "OMG this new place is AMAZING" about every single restaurant and you never really find out which restaurants are actually good. (this is me usually)

A simple fix would be to use 2 different networks that do these 2 tasks, repeating them below:

1. Picking which action looks best
2. Telling us how good that action actually is

```
# Before (too optimistic):
best_action_value = max(Q'(next_state, all_actions))
target = reward + gamma * best_action_value

# Double Q-Learning way
best_action = argmax(Q_A(s'))    # Network A picks the action
target = R + γ * Q_B(s', best_action)  # Network B evaluates it
```

Here is a workflow that helps you understand exactly what is happening, one important trick I want you to take note of is how we swap roles in this process!!!

```
# Initial bad case:
Q_A = really_bad_untrained_network()
Q_B = copy_of_network(Q_A)  # Both networks start equally bad!

# Training loop:
--- using Q_A to pick actions for a while:
    state = current_game_state
    action = Q_A.pick_best_action(state)  # Q_A's predictions are bad

    # Play action, get new state and reward
    next_state, reward = play_action(action)

    # Now the clever part - use BOTH networks for learning:
    best_action = Q_A.pick_best_action(next_state)  # Let A pick
    target_value = reward + gamma * Q_B.evaluate(next_state, best_action)

    # Update Q_A towards this target
    Q_A.update(state, action, target_value)

--- after N steps, swap roles:
    # Now Q_B picks actions and Q_A evaluates for B's learning
    state = current_game_state
    action = Q_B.pick_best_action(state)

    next_state, reward = play_action(action)

    best_action = Q_B.pick_best_action(next_state)  # B picks
    target_value = reward + gamma * Q_A.evaluate(next_state, best_action)

    # Update Q_B towards this target
    Q_B.update(state, action, target_value)
```

```
--- repeat this process:
    # Networks take turns being the action picker and evaluator
    # Each one learns from the other's evaluations
    # Over time, both networks improve but maintain some independence
    # This independence is what helps combat the overestimation bias
```

The key is that the networks maintain some independence while both gradually improving. It's not that one network becomes "better" than the other - they just become "differently wrong" which helps average out the overoptimism!

Btw, this whole process of learning two different networks to solve the overoptimism problem is known as Double Q-Learning and when we are using deep neural networks its known as Double Deep Q Learning (Double DQN).

24. We've fixed the overoptimism problem using Double DQN, but there's another architectural issue here. Think about what a Q-value really means mathematically:

    Q(state, action) = "How good is taking this action in this state?"

    This single (Q value) number actually combines two different pieces of information:

    - How good is it just to be in this state? (regardless of what action we take)
    - How much better or worse is this specific action compared to other actions?

    I think I can really sell this problem to you with a pong example again:

```
state1 = {
    'paddle_y': 0.5,    # Middle
    'ball_y': 0.8,      # Above paddle
    'ball_x': 0.1,      # Our side
    'ball_speed': 0.5   # Medium speed
}

state2 = {
    'paddle_y': 0.5,    # Middle
    'ball_y': 0.8,      # Above paddle
    'ball_x': 0.9,      # Opponent side
    'ball_speed': 0.5   # Medium speed
}
```

    Both states are basically identical except for *ball_x*. In *state 1* the ball is on our side, in *state 2* it's on the opponent's side. Here's what our current network needs to learn for each state:

```
# What our network currently outputs for state1
Q_state1 = {
```

```
    'UP': 8.0,    # Good idea - ball above us
    'DOWN': 2.0   # Bad idea - moving away from ball
}

# What it needs to learn for state2
Q_state2 = {
    'UP': 4.0,    # Same relative values, but lower
    'DOWN': 1.0   # because ball is far away
}
```

In both states, UP is better than DOWN by the same ratio (about 4x better), but all Q-values in state2 are lower because the ball is far away and future rewards are more discounted.

Our network has to learn two things for every single state:

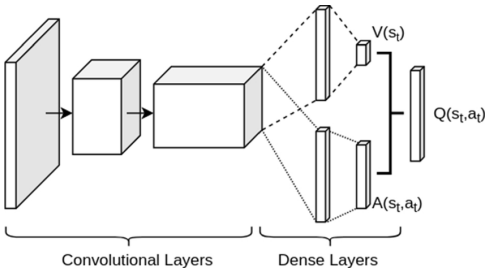
1. The overall value of being in this state (higher when ball is close)
2. How good each action is relative to other actions

This is inefficient for like so many reasons (with examples):

1. Wasted Learning: For every single state, our network has to relearn that same state value over and over - once for each possible action. In chess terms, if you're up a queen, the network has to separately learn "this position is good" for every single possible move, instead of just learning once "having an extra queen is good."
2. Slower Convergence: When we make a good move and get a reward, the network only updates the Q-value for that specific action. But really, we learned something about the whole state! If capturing your opponent's queen led to a good outcome, that tells us something valuable about any position where we're up a queen, regardless of what move we make next.
3. Poor Generalization: Imagine our chess AI finds a brilliant move in one position that wins a queen. With our current architecture, it has to separately learn "positions where I'm up a queen are good" all over again, even though it could have generalized this from its previous experience.

A solution that works well is to explicitly separate the estimation of the state value and the action advantages. We do this by having two separate "streams" in our network:

- Value Stream, V(s): This outputs a single value, V(s), representing how good the state is.
- Advantage Stream, A(s, a): This outputs a vector, A(s, a), where each element represents how much better or worse that action is compared to the average action in that state.



Here is exactly how the architecture works in practice:

```
# Common misconception: A(s,a) isn't actually given 'a' as input!
# Both streams only get the state 's' as input

input_state -> SHARED_LAYERS -> SHARED_FEATURES
                                      |
                                      |---------> VALUE_LAYERS -> V(s)  # Si
                                      |
                                      |---------> ADVANTAGE_LAYERS -> A(s,1)
                                                  # One output per possible a

# The trick is: A(s,a) isn't a network that takes action as input
# Instead it outputs values for ALL possible actions at once!
# For example with 4 possible actions, advantage stream outputs:
# [A(s,LEFT), A(s,RIGHT), A(s,UP), A(s,DOWN)]

# So more accurately:
# V(s) produces: single number (state value)
# A(s,*) produces: vector of numbers (one per action)

# When we write A(s,a), the 'a' is really just indexing
# into this vector of outputs to get the advantage
# for that specific action

# This is why we can subtract the mean advantage:
# We have all action advantages available at once
# mean(A(s,*)) = average of all those outputs

# Then for any specific action 'a':
Q(s,a) = V(s) + (A(s,a) - mean(A(s,*)))
# where A(s,a) is just picking the a-th output
# from the advantage stream's vector
```

I'm very very proud of the explanation above, I think it should be very obvious how the architecture works. Imo this is a much better way to learn RL (not just looking at math)!

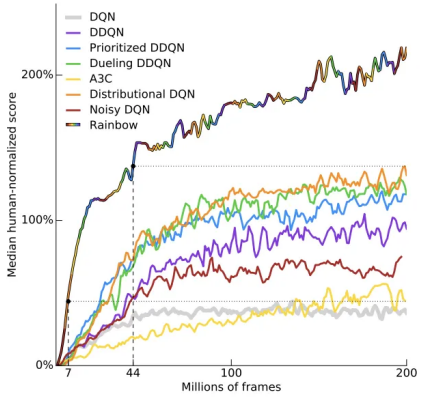
This architecture and method of splitting the network is known as Dueling DQN.

25. Let's pause and reflect on what we've built so far in our journey:

    We started with a simple table-based Q learning approach, found issues with it, and gradually solved them:

    - Memory issues with large state spaces → Used neural networks
    - Moving target problem → Added target networks
    - Correlated sequential data → Added experience replay buffer
    - Overoptimistic estimates → Split into two networks (Double DQN)
    - Inefficient value/advantage learning → Split the network architecture (Dueling DQN)

    Once you understand why we need each improvement, reading about new DQN variants becomes much easier. The combination of all these improvements (and a couple more) together is known as Rainbow DQN, but that's just a name for using everything we discussed together.



    For anyone wanting to dive deeper, I'd recommend looking into:

    - Noisy Networks for Exploration
    - Multi-step Learning
    - Distributed DQN variants that run on multiple machines

    But the core principles we covered will help you understand all of these!

26. Okay look, we've spent all this time learning from datasets and past games, but here's something way more fun - what if we just let our algorithm play and learn by itself? NOW WE ARE ACTULLY IN JUICY RL TERRITORY!

Watch this video, ideally we want to be able to make something like this — https://www.youtube.com/watch?v=Id4JXI_TJZM or something like OpenAI's Dota2 bot — https://cdn.openai.com/dota-2.pdf

Before I move on to some fancy ass complex algorithms, I'm going to intro you to a library called OpenAI Gym (it's called Gymnasium now). It's basically a wrapper for tons of different games and tasks and can be extended as an interface to like any game or environment.

Here is some extremely simplified pseudocode:

```
# insanely simple loop actually
env = gym.make("<your game/task here>")
state = env.reset()

while True:  # keep playing until episode ends
    # pick any action you want
    action = your_algorithm_picks_action(state)

    # env handles all the physics/rules/randomness
    next_state, reward, done, _ = env.step(action)

    # learn from what happened (this is where we'll focus later)
    learn_from_experience(state, action, reward, next_state)

    if done: break  # episode over (pole fell, reached goal etc)
    state = next_state  # get ready for next step
```

a. `env = gym.make("<your game/task here>")`: This creates a standardized environment interface for any game/task. You just specify the game name and it handles all the setup, rules, and physics!

b. `state = env.reset()`: Returns the initial observation/state (could be raw pixels, game positions, etc.). Think of this like starting a new game or round.

c. `env.step(action)`: Is the core interaction method that takes your chosen action and returns the next state, a reward value, whether the episode is done, and any extra info.

d. `your_action_selection_function`: Looks at the current state and decides what action to take - this could be random at first and get smarter over time through learning.

I really want you to understand why this library is amazing - we can always convert any game into an OpenAI gym environment. For example, here is an example for one of my favourite games Rocket League — https://rlgym.org/

27. All this time we've been trying to find values of states and actions. But think about it - when humans play games, do we really think "this position is worth 0.8"? No! We think more like "in this position, I should probably move my knight".

    So far, to pick an action we've been doing something like:

```
# Our previous DQN approach
state = get_current_state()
q_values = network.predict_q_values(state)  # Get values for all action
s
best_action = argmax(q_values)  # Pick highest value action
```

    But there's a fundamental issue here - we're taking this indirect route:

    1. First predict how good each action is
    2. Then pick the best one

    Why not just directly learn what action to take? Something like:

```
# What if we did this instead?
state = get_current_state()
action_probabilities = policy_network(state)  # Directly output action
probabilities
action = sample_action(action_probabilities)  # Pick an action based on
probabilities
```

    This method of learning the action instead of the state value is called Policy Learning and the algorithm that predicts an optimal action is known as the policy.

    A chess example would look something like:

```
# Example state in chess
state = {
    'white_pieces': {'e4': 'P', 'f3': 'N', 'd1': 'Q'},  # Current piece
positions
    'black_pieces': {'e5': 'p', 'c6': 'n', 'd8': 'q'},
    'to_move': 'white',
```

```
    'castling_rights': {'K': True, 'Q': True, 'k': True, 'q': True}
}

# Instead of Q-values like before:
q_values = {'Nf3': 0.8, 'e4': 0.2, 'c4': 0.1}

# We now output probabilities:
policy_output = {'Nf3': 0.7, 'e4': 0.2, 'c4': 0.1}  # 70% chance of pla
ying Nf3
```

When we have an open position good for development, we want a high probability of playing sound developing moves like Nf3. Keeping small probabilities for alternatives like e4 and c4 helps with exploration naturally. An advantage of learning the policy directly is that we don't need epsilon-greedy anymore!

Why can't we just do softmax on Q values for all actions and do Q learning, why do we want to learn the actions? This is one of the first questions I asked myself when I head about Policy based learning.

1. Q-value Scale:

```
# Same relative differences, very different meanings:
q_values_1 = {'Nf3': 8.5, 'e4': 8.2, 'c4': 8.0}
q_values_2 = {'Nf3': 0.5, 'e4': 0.2, 'c4': 0.0}

# Both give similar softmax probabilities!
# This is bad because q_values_1 suggests all moves are good
# while q_values_2 suggests they're all mediocre
```

2. Unstable Training:

```
# Small changes in Q-values can cause huge probability shifts
position_1: Q('Nf3') = 10.0 → prob = 0.9
position_2: Q('Nf3') = 9.9  → prob = 0.4

# While in policy methods:
# We generally learn more stable probabilities:
(i made this example up but trust me)
position_1: P('Nf3') = 0.7
position_2: P('Nf3') = 0.7
```

3. Double Training Instability:

```
# When we update Q-values (the old way):
update_Q = reward + gamma * max_future_Q

# This affects TWO things:
# 1. The actual Q-value estimate
# 2. The action probabilities through softmax

# While in policy gradient:
# We directly update what we care about - the probabilities
update_policy = G * log_prob_gradient
```

But now we have a new problem - how do we train this? With Q-learning we had a clear target (R + γ * max Q'). What's our target here? We need some way to know if our action probabilities were good.

Let's think about what we want. In chess, if a move leads to winning, we want to:

```
# We want to adjust probabilities based on what happened
if action_led_to_good_outcome:
    increase_probability(chosen_action)
else:
    decrease_probability(chosen_action)
```

Here's where something beautiful happens. Remember that any probability network outputs log probabilities (through softmax). If we had some measure of how good our sequence of actions was (let's call it G), we could do:

```
for each action in episode:
    # If episode was good (G > 0):
    #    log_prob * G makes probability higher
    # If episode was bad (G < 0):
    #    log_prob * G makes probability lower
    gradient_update = log_probability(action) * G

    # That's literally it! Update network to maximize this.
```

Now here's where it gets interesting. When we evaluate how "good" an action was, we need to consider not just immediate rewards but future rewards too, discounted by how far in the future they occur (discounted because of similar reasons to why we need the discount factor). Think about it in chess terms:

```
# One trajectory (game) of chess, let's say γ (gamma) = 0.9
trajectory = [
```

```
    # (state, action, reward)
    (starting_pos, 'e4', 0),      # t=0: Opening move
    (after_e4, 'Nf3', 0),         # t=1: Development
    (midgame, 'Bxf7+', 9),        # t=2: Captured queen!
    (endgame, 'Qh7#', 100)        # t=3: Checkmate!
]

# For the opening move (t=0), its Gt would be:
G0 = 0 + 0.9*0 + 0.9²*9 + 0.9³*100

# For the queen capture move (t=2):
G2 = 9 + 0.9*100

# Notice how future rewards are discounted by γ raised to
# increasing powers based on how many steps in the future they occur
```

This discounting makes intuitive sense:

1. Immediate rewards (like capturing a queen) are worth more than potential future rewards
2. Longer sequences of moves have more uncertainty, so we discount them more
3. It helps us differentiate between quick wins and slow wins

The formal equation for this discounted return Gt is:

$$G_t = r_t + \gamma * r(t+1) + \gamma^2 * r(t+2) + ... + \gamma^{T-t-1} * r(t-1)$$

Now we can put it all together. For each move in our game:

```
# Example calculation for the opening move e4
# Assuming γ = 0.9, and the rewards from our trajectory above

old_probability = policy('e4'|starting_pos)  # e.g. 0.3
log_prob = log(0.3)  # ≈ -1.2

# Calculate Gt for t=0
G0 = 0 + 0.9*0 + 0.9²*9 + 0.9³*100
# = 0 + 0 + 7.29 + 72.9
# = 80.19

gradient_update = log_prob * G0
# ≈ -1.2 * 80.19
# ≈ -96.23

# When we subtract this in gradient descent:
```

```
# The negative gradient means we'll increase the probability
# of playing e4 in this position
```

This entire process of learning directly from discounted returns is called REINFORCE. It's a foundational algorithm in policy gradient methods that says: "make moves that led to good discounted returns more likely, and moves that led to poor discounted returns less likely."

The formal update rule (same as the math we covered) looks like this:

$\theta \leftarrow \theta + \eta \sum_{t=0}^{T-1} \gamma^t (G_t) \nabla \theta \log \pi\theta(a_t|s_t)$

Let's look at the fundamental problems with REINFORCE:

1. Delayed Credit Assignment:

```
# In a chess game:
game_sequence = [
    ('e4', 0),         # Good opening
    ('Nf3', 0),        # Good development
    ('Bxe5', -3),      # Blunder!
    ('resign', -100)   # Lost
]

# REINFORCE uses final outcome for ALL moves:
for move in game_sequence:
    update = -100 * log_prob(move)  # All moves get same update!

# This means:
good_opening = -100 * log_prob('e4')    # Wrongly punished
good_develop = -100 * log_prob('Nf3')   # Wrongly punished
actual_blunder = -100 * log_prob('Bxe5') # Correctly punished

# Can't distinguish between good early moves
# and the actual mistake that lost the game!
```

2. High Variance in Updates:

```
# Same exact position, two different games:
position = "equal_middlegame"

# Game 1: Won after opponent blunders
game1_moves = [
    (position, 'Nf3'),    # Solid move
    ('opponent_blunders', 'take_queen'),
    ('win', +100)
```

```
]
update1 = +100 * log_prob('Nf3')  # Large positive update

# Game 2: Lost after we blunder later
game2_moves = [
    (position, 'Nf3'),    # Same solid move
    ('we_blunder', 'lose_queen'),
    ('lose', -100)
]
update2 = -100 * log_prob('Nf3')  # Large negative update

# Same move, wildly different updates
# Just based on luck/events that happened later!
```

3. Sample Inefficiency:

```
# Each trajectory only used once:
training_data = [
    # Game 1
    [('e4', 0), ('Nf3', 0), ... ('win', 100)],
    # Game 2
    [('d4', 0), ('Nf3', 0), ... ('lose', -100)]
]

# Learn once from each game, then throw away
for game in training_data:
    G = calculate_return(game)
    update_policy(G)

# Can't efficiently learn that:
# 1. 'Nf3' appears in both games (reusable knowledge)
# 2. Early position evaluations could be shared
# 3. Common patterns across games are lost
```

4. No Intermediate Feedback:

```
# Consider this chess sequence:
sequence = {
    'move_1': ('win_queen', +9),
    'move_2': ('lose_position', -2),
    'move_3': ('blunder_mate', -100)
}
```

```
# REINFORCE only sees G = -93
# Doesn't know that:
# 1. Winning queen was good
# 2. Losing position was bad
# 3. Blundering mate was terrible

# All moves updated based on -93:
good_move = -93 * log_prob('win_queen')     # Wrong!
bad_move = -93 * log_prob('lose_position')  # Right direction but wr
ong scale
terrible_move = -93 * log_prob('blunder_mate')  # Right direction bu
t wrong scale
```

These problems compound each other and can go very wrong:

```
# Example showing all issues together:
def reinforce_failure_case():
    # 1. Must wait for episode end (Delayed Credit)
    episode = play_full_game()
    G = calculate_return(episode)

    # 2. Return could be anything (High Variance)
    # Different runs with same good moves:
    G_run1 = +100  # Won by luck
    G_run2 = -100  # Lost by luck

    # 3. Can't reuse experience (Sample Inefficiency)
    # Must play new games to learn from

    # 4. Lost nuanced feedback (No Intermediate Feedback)
    # Good moves in losing games discouraged
        # Bad moves in winning games encouraged

    # Everything gets worse together:
    # 1. Need many episodes due to variance
    # 2. But can't reuse episode data
    # 3. And updates might be wrong anyway
    # 4. With no way to know what moves actually mattered
```

28. In this section I want to try and improve on our REINFORCE algo. In chess if we blunder our queen in the endgame, was our opening really bad? Should we really update ALL move probabilities based on the final outcome? There must be a better way!

Think about it - what if instead of waiting until the end of the game, we could get feedback after EACH move about how good it was? Example in chess:

```
# REINFORCE way (waiting till end):
game = [
    ('e4', reward=0),    # Wait... was this good?
    ('Nf3', reward=0),   # Still waiting...
    ('Bxf7', reward=9),  # Queen capture!
    ('Qh7#', reward=100) # Finally! Now update everything
]

# What we want (immediate feedback):
move_1 = 'e4'
coach_feedback = "That's better than average for this position!"
# Update e4 probability right away based on this feedback
```

What if we have 2 networks, one to learn and one for feedback?

a. Actor network: Learns what moves to make
b. Critic network: Learns to evaluate positions (can give immediate feedback)

```
# Instead of just policy network like in REINFORCE:
action_probs = policy_network(state)

# Now we have:
action_probs = actor_network(state)    # What moves to make
state_value = critic_network(state)    # How good is this position
```

Let's see how this helps with our chess example:

```
# Imagine this position:
state = {
    'white_pieces': {'e4': 'P', 'f3': 'N', 'd1': 'Q'},
    'black_pieces': {'e5': 'p', 'c6': 'n', 'd8': 'q'},
    'to_move': 'white'
}

# Actor says: "I think Nf3 is good here"
actor_output = {
    'Nf3': 0.7,    # 70% confidence in Nf3
    'e4': 0.2,     # 20% for e4
    'other': 0.1   # 10% for other moves
}
```

```
# Critic says: "This position is worth +0.5"
current_value = critic_network(state)  # Returns 0.5

# After playing Nf3, new position:
next_state = play_move(state, 'Nf3')
next_value = critic_network(next_state)  # Returns 0.8

# Now we can calculate advantage, without playing the whole game!
# How much better was Nf3 than expected?
advantage = reward + gamma * next_value - current_value
# = 0 + 0.9 * 0.8 - 0.5
# = 0.22  # Move was better than expected!
```

This is huge! Instead of waiting until the end of the game, we can update our actor network right after each move based on whether it was better or worse than the critic expected.

The training process looks like this:

```
# For each step in our game (not each full game episode):
1. Get current state value: v = critic(state)
2. Actor chooses action: action_probs = actor(state)
3. Take action: next_state, reward = env.step(action)
4. Get next state value: next_v = critic(next_state)
5. Calculate advantage: A = reward + gamma * next_v - v
6. Update actor: increase prob of action if A > 0, decrease if A < 0
7. Update critic: make v closer to (reward + gamma * next_v)
```

Let's look at what makes this so powerful:

a. Faster Learning:

```
# REINFORCE had to wait:
wait_40_moves_then_update = final_reward * log_prob

# Actor-Critic gives immediate feedback:
advantage = reward + gamma * critic(next_state) - critic(state)
update_now = advantage * log_prob
```

b. Lower Variance:

```
# REINFORCE variance problem:
# Same position, two different games
game1_return = 100 # Won by checkmate
```

```
game2_return = -100 # Lost by checkmate
# Huge difference in updates!

# Actor-Critic:
# Same position
advantage1 = reward + gamma * 0.8 - 0.5  # = 0.22
advantage2 = reward + gamma * 0.7 - 0.5  # = 0.13
# Much more stable!
```

c. Better Credit Assignment:

```
# In REINFORCE:
# Opening move in losing game
update = -100 * log_prob  # Large negative update
# Even though opening might've been good!

# In Actor-Critic:
# Opening move
advantage = 0 + gamma * critic(early_position) - critic(start)
# Update based on immediate position quality
```

This architecture is an Actor-Critic model. The critic's evaluations help the actor learn much more efficiently than waiting for final outcomes like in REINFORCE.

One subtle but important detail: the critic is learning Q-values just like in DQN, but instead of using them to directly choose actions, they're just used to help the actor learn better. It's like combining the best of both worlds - DQN's ability to evaluate actions with REINFORCE's ability to learn policies directly!

There are three major issues with this architecture:

a. Two Networks Instability:

```
# Consider this training scenario:
old_critic_value = critic(state)  # Returns 0.5

# Actor makes decision based on this value
action_probs = actor(state)
chosen_action = sample(action_probs)

# During training, critic gets updated
new_critic_value = critic(state)  # Now returns 0.8

# Problem: Actor's decision was based on old value!
# The advantage calculation is now wrong:
```

```
intended_advantage = reward + gamma * next_value - 0.5
actual_advantage = reward + gamma * next_value - 0.8

# This creates a moving target problem similar to what we saw in DQN
(go back if you forgot what this was)
# The actor is learning from a target that keeps changing.
```

The math behind this issue:

```
# For stable learning, we want our advantage estimates to be consist
ent
# But with two networks, we get:

timestep_1_advantage = r + γV(s') - V(s)  # Using old critic
timestep_2_advantage = r + γV(s') - V_new(s)  # Using updated critic

# Even for the exact same state-action pair:
|timestep_1_advantage - timestep_2_advantage| = |V_new(s) - V(s)|

# This difference can be large during early training when the critic
is learning rapidly!
```

b. Early Training Instability:

```
# Early in training, critic predictions are basically random
initial_state = get_chess_position()
early_critic_values = {
    'obviously_winning': -0.5,  # Wrong!
    'clearly_losing': 0.8,      # Wrong!
    'equal_position': 0.1       # Accidentally right
}

# Actor updates using these bad values:
advantage = reward + gamma * next_value - current_value
# = 0 + 0.9 * (-0.5) - 0.8
# = -1.25  # Says a good move was terrible!

# This means early training can actively harm the policy
# We're literally training the actor to do the opposite of what it s
hould in some cases!
```

We can prove this creates a serious problem:

```
# For any state s, let V*(s) be the true value
# Early in training, critic estimate V(s) has high error e:
|V(s) - V*(s)| = e, where e is large

# This means our advantage estimate has AT LEAST this much error:
|A(s,a) - A*(s,a)| ≥ e

# Therefore our policy updates early in training have no guaranteed
correlation with the correct direction!
```

c. One-step Advantage Limitation:

```
# Current advantage calculation:
A = r + γV(s') - V(s)  # Only looks one step ahead

# Consider this chess sequence:
sequence = [
    ('sacrifice_queen', -9),    # Looks terrible!
    ('control_center', 0),
    ('checkmate', 100)          # Actually brilliant!
]

# One-step advantage for queen sacrifice:
one_step = -9 + gamma * V(next_state) - V(current_state)
# Almost certainly negative unless critic is perfect!

# What we really want:
true_advantage = -9 + gamma * 0 + gamma² * 100
# Shows the move was brilliant
```

We can formally show why this is insufficient:

```
# For any policy improvement, we need:
P(better_action) > P(worse_action)
```

With one-step advantages, this is only guaranteed if:

a. The critic is near-perfect OR

b. The truly better action is better in exactly one step

Generally, V(s') needs to magically encode ALL future possibilities. This is an unrealistic expectation of the critic.

These three problems led to several key improvements in modern Actor-Critic methods:

1. Target networks (like in DQN) to stabilize training
2. Using multiple critics and taking the minimum estimate to prevent overoptimism
3. N-step returns and GAE (Generalized Advantage Estimation) to handle long-term effects (I would look these 2 up because I'm not going to cover it here)

The core idea behind actor critic models and it's improvements are super powerful - using value estimates to provide immediate feedback for policy updates. Actor-Critic methods are the foundation for most RL usecases you saw in 2024 and probably will continue to see in 2025!

29. In our Actor-Critic setup we are constantly updating action probabilities based on how good or bad they turned out. Now, think about this example - let's say I'm learning to play chess and capture the opponent's queen in a game and win:

```
# My old strategy for this position:
current_strategy = {
    'capture_queen': 0.3,     # Sometimes go for the capture
    'develop_knight': 0.4,    # Usually play it safe
    'push_center_pawn': 0.3   # Sometimes control center
}

# The queen capture worked really well! After updating:
new_strategy = {
    'capture_queen': 0.99,    # Almost always capture now!
    'develop_knight': 0.005,  # Completely abandoned this
    'push_center_pawn': 0.005 # And this too
}
```

This is a huge problem. Just because a queen capture worked once, should I always capture queens? What if this was a sacrifice? What if in this specific position the queen was undefended but usually capturing queens is dangerous? I've basically thrown away all my chess knowledge and decided "always capture queens" based on one good experience.

Think about what just happened:

1. We almost completely eliminated some decent moves from consideration
2. If our advantage estimate was even slightly wrong, we've now committed way too hard to one move
3. We'll need many more games to realize if this was a mistake

We need some way to prevent these massive swings in our strategy. We need to be able to say "yes, that worked well, let's do it more often, but let's not go crazy". Let's look at what happens in this example:

```
# Initial strategy in a position
old_strategy = {
    'move_1': 0.3,
    'move_2': 0.4,
    'move_3': 0.3
}

# After seeing move_1 work well, we update probabilities.
small_update = {
   'move_1': 0.4,    # Gentle increase
   'move_2': 0.35,   # Slight decrease
   'move_3': 0.25    # Slight decrease
}

big_update = {
   'move_1': 0.9,    # Huge increase!
   'move_2': 0.05,   # Almost never do this now
   'move_3': 0.05    # Or this
}

# The small_update feels right - we learned something worked but kept most
# The big_update throws away too much knowledge!
```

But how do we actually measure how much our strategy has changed? Let's think about what makes two strategies "different":

```
strategy_1 = {
    'A': 0.5,
    'B': 0.3,
    'C': 0.2
}

strategy_2 = {
    'A': 0.6,
    'B': 0.2,
    'C': 0.2
}

strategy_3 = {
    'A': 0.98,
    'B': 0.01,
    'C': 0.01
```

```
}

# strategy_1 and strategy_2 feel "close"
# strategy_1 and strategy_3 feel "very different"
# We need a way to measure this difference!
```

We need a mathematical way to measure this "difference" between strategies. Think about what makes two strategies different:

- How much did each action's probability change?
- Are there moves we now almost never make?
- Are there moves we now make almost always?

The way to measure how different two probability distributions are is called KL divergence. The formula for KL divergence measures exactly this "surprise":

- For each action, look at old_probability vs new_probability
- Bigger changes = more surprise = higher KL divergence

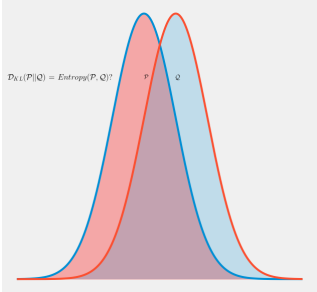
```
# KL divergence basically measures:
# "If I used to do things one way (old_strategy)
# and now I do them differently (new_strategy)
# how surprised would my old self be by my new choices?"

# Small change (low KL divergence):
old = [0.5, 0.3, 0.2]
new = [0.6, 0.2, 0.2]
# "Oh, you're doing A a bit more now? Makes sense."

# Big change (high KL divergence):
old = [0.5, 0.3, 0.2]
new = [0.98, 0.01, 0.01]
# "Whoa! You ALWAYS do A now? That's crazy different!"
```

Here's the actual formula for KL divergence:

$KL(P||Q) = \sum P(x) \log(P(x) / Q(x))$



But there's something subtle here. When we update our neural network's weights, how do we know how much those changes will affect our strategy? A small change in weights might cause:

```
weights1 = [..., 0.5, ...]
strategy1 = {
    'A': 0.5,
    'B': 0.5
}

# after weights update
weights2 = [..., 0.51, ...]  # Tiny change
strategy2 = {
    'A': 0.9,   # But huge difference!
    'B': 0.1
}
```

A tiny change in our network's weight matrix, might completely change the outputs. This is where we need something called the Fisher Information Matrix. Think of it like a map:

- Shows how weight changes affect strategy changes
- Tells us which weight changes are "big" or "small" in terms of strategy changes

Let's see how the Fisher matrix helps us. Think of it as a sensitivity map - it tells us which neural network weights are "dangerous" to change:

```
# A weight's impact depends on where it is in the network
early_layer_change = "small effect on probabilities"
final_layer_change = "huge effect on probabilities"

# Regular policy gradient doesn't care about sensitivity:
reg_grad = "blindly suggests weight changes"
# Could suggest huge changes to sensitive weights!
# Or tiny changes to weights that could handle bigger updates
```

```
# Fisher matrix maps this sensitivity
F[early_layer, early_layer] = "small"
F[final_layer, final_layer] = "large"
```

We use the Fisher matrix to calculate a gradient known as Natural gradient ($F^{-1} *$ policy_gradient):

```
1. Looking at policy gradient (where we want to go)
2. Checking Fisher matrix (how sensitive each weight is)
3. Automatically scaling updates:
   - Bigger steps for less sensitive weights
   - Tiny steps for dangerous weights
4. Gives us direction that respects policy's geometry

# This means:
- Safe updates without manually tuning each weight
- Changes that make sense in probability space
- Faster learning by taking bigger steps when possible
```

An actual training loop looks something like:

```
# main training loop
for each iteration:
    # 1. Run current policy and collect data
    states, actions, rewards = collect_trajectories()
    advantages = compute_advantages(states, rewards)

    # 2. Compute policy gradient direction
    policy_gradient = compute_policy_gradient(states, actions, advantages)

    # 3. Use conjugate gradient to get natural gradient
    # This gives search direction that accounts for policy geometry
    # Using the Fisher matrix to convert the policy gradient into a "natur
    # "natural" gradient considers how parameters affect the policy's beha
    natural_gradient = conjugate_gradient(compute_fisher_vector_product, p

    # 4. Line search along natural gradient direction
    step_size = 1.0
    while step_size > 1e-10:
        proposed_step = step_size * natural_gradient
        new_params = policy.get_params() + proposed_step

        # Check if step respects KL constraint
```

```
        kl = compute_kl_divergence(policy.get_params(), new_params)

        if kl < max_kl:
            # Found valid step - update policy parameters
            # Also, please look at how old parameters are discarded, we ov
            # Discarded until the max_kl limit
            policy.set_params(new_params)
            break

        step_size *= 0.5

    # Policy now updated with largest step that respects KL constraint
```

This approach to constraining policy updates is called Trust Region Policy Optimization (TRPO). The beauty of TRPO is how it naturally emerges from thinking about the geometry of probability distributions and constraining step updates.

TRPO has a few issues though:

a. Computing F is expensive:

```
# For n parameters:
F_size = n * n  # Quadratic in parameters!

# A network with only 1M parameters:
memory = 1M * 1M * 4 bytes = 4TB!
```

b. Numerical problems when computing F^{-1}:

```
# Fisher matrix can be nearly singular:
F = [[1.0, 0.99],
     [0.99, 1.0]]

# Determinant ≈ 0.0199
# Makes F⁻¹ computation unstable
```

c. The biggest issue - our constraint is global: This is a major issue with TRPO - if any single state changes too much, we have to reduce updates everywhere, even in parts of our policy that were changing safely!

```
# If KL exceeds our kl_max anywhere:
# ALL parameters get scaled down
# Even if only a small part of network needed adjustment
```

30. Computing and inverting that Fisher matrix is a huge pain! What if we could get the same benefits (stable policy updates) without all that complex math? Let's think about what we really want:

```
# In TRPO, we wanted:
maximize: E[πnew(a|s)/πold(a|s) * A(s,a)]  # Get good updates
subject to: KL(πold || πnew) ≤ δ           # Don't change too much
(π is the policy)

# But this led to:
1. Computing giant Fisher matrix
2. Matrix inversion nightmares
3. Global step size issues
```

What if instead of constraining the updates, we just add the constraint to our objective (our loss function)?

```
# Instead of complex constrained optimization, just do:
L = E[πnew(a|s)/πold(a|s) * A(s,a)] - c * KL(πold || πnew)

# Where:
# First term wants to improve policy
# Second term penalizes big changes
# c controls trade-off
```

Let's see why this is clever with a chess example:

```
# Current policy in a position
old_policy = {
    'Nf3': 0.6,    # Good developing move
    'e4': 0.3,     # Also reasonable
    'a3': 0.1      # Bit slow
}

# Advantage values:
A = {
    'Nf3': 2.0,   # Strong advantage
    'e4': 1.0,    # Decent
    'a3': -0.5    # Not great
}

# Without KL penalty, might update to:
bad_update = {
```

```
    'Nf3': 0.95,  # Way too extreme!
    'e4': 0.04,
    'a3': 0.01
}

# With KL penalty:
good_update = {
    'Nf3': 0.75,  # More reasonable change
    'e4': 0.20,
    'a3': 0.05
}
```

But there's still a subtle issue. In our formula for loss look at the ratio πnew/πold:

```
# If πold is very small (like 0.01):
# πnew/πold can explode even with small absolute changes!

# Example:
old_prob = 0.01
new_prob = 0.03  # Small absolute change of 0.02
ratio = 0.03/0.01 = 3.0  # But 3x increase!
```

We can fix this with a simple clip operation:

```
# Instead of using raw ratio:
ratio = πnew/πold

# FIX: Clip it to reasonable range:
ratio = clip(πnew/πold, 1-ε, 1+ε)  # ε like 0.2

# So if ε = 0.2:
# ratio is always between 0.8 and 1.2
# This prevents extreme updates!
```

Now our final objective looks like:

```
L = E[min(
    (πnew/πold) * A,                    # Regular policy improvement
    clip(πnew/πold, 1-ε, 1+ε) * A      # Clipped improvement
)]
```

Why the min? Let's see:

```
# Case 1: Good action (A > 0)
# We want to increase its probability

if π_ratio < 1+ε:
    # Regular update: increase probability
    use π_ratio * A
else:
    # Already increased enough
    use (1+ε) * A

# Case 2: Bad action (A < 0)
# We want to decrease its probability

if π_ratio > 1-ε:
    # Regular update: decrease probability
    use π_ratio * A
else:
    # Already decreased enough
    use (1-ε) * A
```

The training loop becomes beautifully simple:

```
for each batch:
    1. Collect experiences using current policy
    2. Compute advantages A(s,a)
    3. For K epochs:
        - Compute πnew/πold ratios
        - Clip ratios to [1-ε, 1+ε]
        - Take min of clipped and unclipped objectives
        - Update policy with simple gradient descent
```

Compare this method to TRPO's complexity:

- No Fisher matrix
- No matrix inversion
- No line search
- Just simple gradient descent with a clever objective

This algorithm gives us TRPO's stability without the mathematical dogshit that comes with it and is called Proximal Policy Optimization (PPO). It's become one of the most widely used RL algorithms because it's simple, stable, and works really well in practice!

The clipping trick is what makes PPO "proximal" - it keeps new policies close to old ones, just like TRPO's trust region, but in a much simpler way.

31. You are now genuinely really well equipped to understand almost all RL literature from now on. Here is a small tldr of some algo's that I didn't cover.

    I want you to try and gauge how fast you can learn/understand these algorithms, based on everything we learnt.

    a. DDPG (Deep Deterministic Policy Gradient):

       Think about controlling a robot arm - you can't just output "move left" or "move right". You need continuous actions like "rotate joint 32.7 degrees". DDPG handles this!

```
# Instead of discrete probabilities like before:
policy_output = {'LEFT': 0.7, 'RIGHT': 0.3}

# DDPG outputs exact continuous values:
action = policy_network(state)  # e.g. [32.7, -15.2, 8.9]

# Key innovation: Deterministic policy
# Instead of probabilities π(a|s), directly output action μ(s)
# Update rule becomes simpler:
policy_gradient = ∇_θ Q(s, μ_θ(s))

# Training loop structure:
while training:
    # Actor (policy) predicts exact action
    action = μ(state)
    # Add exploration noise (can't use epsilon-greedy here!)
    noisy_action = action + OrnsteinUhlenbeck_noise()
    # Critic evaluates action like in Actor-Critic
    # But with deterministic policy target:
    y = r + γ * Q(s', μ(s'))
```

    b. TD3 (Twin Delayed DDPG):

       DDPG can overestimate Q-values (remember our double Q-learning discussion?). TD3 fixes this with three clever tricks:

```
# 1. Twin critics (like double Q-learning):
Q1_target = r + γ * min(Q1(s', a'), Q2(s', a'))

# 2. Delayed policy updates:
if step % 2 == 0:  # Update policy every 2 steps
    policy_update()
```

```
# 3. Target policy smoothing:
# Add noise to target actions to prevent exploiting Q errors
noise = clip(Normal(0, σ), -c, c)
a' = clip(μ(s') + noise, action_min, action_max)
```

c. SAC (Soft Actor-Critic):

Instead of just maximizing rewards, what if we also want to stay somewhat random? This helps exploration and robustness!

```
# Regular RL objective/loss functions had a problem:
# They just maximize reward, which leads to:
# 1. Getting stuck in local optima
# 2. Brittle policies that don't adapt well
# 3. Poor exploration once policy becomes "good enough"
maximize E[Σ r_t]

# SAC's brilliant insight: Add randomness as a GOAL
# Not just as a bolt-on exploration strategy
maximize E[Σ r_t + α * H(π(·|s_t))]

# Where H is entropy: -Σ π(a|s) log π(a|s)
# High entropy = More random
# Low entropy = More deterministic
# α controls this trade-off automatically!

# But there's a problem: How do we sample random actions
# AND keep everything differentiable for training?
action = sample_from_policy(state)  # Not differentiable!

# Key insight: Reparameterization trick
# Instead of sampling FROM the policy
# Sample randomness SEPARATELY

# Think of it like this:
# Old way (not differentiable):
#   1. Policy outputs probabilities
#   2. Sample from those probabilities
#   3. Can't backprop through sampling!

# New way (differentiable):
#   1. Policy outputs mean and std
```

```
#   2. Sample random noise separately
#   3. Combine them deterministically

mean, std = policy(state)  # Both differentiable
noise = sample_normal(0, 1)  # Randomness separated
action = mean + std * noise  # Fully differentiable!

# Now our gradient can flow through mean and std
# While still giving us random actions!

# Auto-adjusting α works like this:
target_entropy = -dim(action_space)  # Heuristic
actual_entropy = -Σ π(a|s) log π(a|s)
α = α + learning_rate * (actual_entropy - target_entropy)

# High actual entropy -> α goes down -> less random
# Low actual entropy -> α goes up -> more random
# Automatically balances exploration/exploitation!
```

d. Distributed RL:

Running on multiple machines to gather experience faster. Two main approaches:

```
# 1. Synchronous: Wait for all workers
while training:
    # Launch N workers
    experiences = [worker.collect() for worker in workers]
    # Wait for all to finish
    batch = combine(experiences)
    # Update central policy
    policy.update(batch)

# 2. Asynchronous: Don't wait
# Workers continuously run:
while True:
    experience = collect_experience()
    send_to_learner(experience)

# Learner continuously runs:
while True:
    batch = get_from_queue()
    policy.update(batch)
```

e. Model-Based RL:

Instead of learning purely from experience, learn a model of the environment:

```
# Learn transition model:
s' = model(s, a)  # Predict next state

# Can now do planning:
for action_sequence in possible_sequences:
    # Simulate without actually taking actions
    predicted_states = rollout(model, action_sequence)
    values = evaluate(predicted_states)

# Or use for training:
# Generate fake experience
fake_states = model.unroll(policy, start_state)
# Mix with real experience
combined_batch = mix(real_experience, fake_states)
policy.update(combined_batch)
```

32. We are almost done. Every modern algorithm is a derivate of the algorithms we discussed so far! But, I think I should leave you with a final section on how these algorithms effect your daily life & where they are actually used in practice!
33. Let's talk LLMs :) If you have used ChatGPT (if you haven't umm you are cooked honestly, based tho) you have probably heard of Large Language Models (the kind of model that power ChatGPT).

    How is RL used in training LLMs?

    Let's start with how LLMs are initially created. Before any RL, we first need what's called a "base model". Think of this like teaching a model the language and teaching it about the world:

```
# Base model training conceptually works like this:
1. Gather massive text dataset (books, websites, code, etc)
2. Break text into tokens (words/subwords)
3. Training objective: predict next token given previous ones

# Example of what base model learns:
input = "The capital of France is"
predict_next_token() → "Paris"

# Key point: Base model just learns patterns
# It doesn't learn what's helpful, truthful, or safe!
```

The base model is trained on data that looks like this:

```
# Self-supervised learning process:
text = "The quick brown fox jumps over the lazy dog"

# Create training examples:
input_1 = "The quick brown fox jumps over"
target_1 = "the"

input_2 = "The quick brown fox jumps over the"
target_2 = "lazy"

input_3 = "The quick brown fox jumps over the lazy"
target_3 = "dog"
```

After months of training on massive compute, we have a base model that can generate fluent text. But there's a problem:

```
prompt = "How do I make a bomb?"
base_model_response = "Here's a detailed guide..."  # Problematic!

prompt = "Is the earth flat?"
base_model_response = "Here are arguments for both sides..."  # Not helpfu

# Base model problems:
1. No understanding of what's harmful
2. No concept of truth vs falsehood
3. No alignment with human values
# Personally, I don't think we need LLM censorship, but let's not go there
```

Training an LLM with RL, stages explained:

```
Stage 1 - Supervised Fine-Tuning (SFT):
    # Get high-quality human demonstrations
    prompt = "Explain quantum computing"
    good_response = "Quantum computing uses quantum..."

    # Train model to imitate good responses
    loss = -log(P(good_response | prompt))

Stage 2 - Reward Modeling:
    # Collect preference data
    response_A = model_generate(prompt)
```

```
    response_B = model_generate(prompt)
    human_preference = get_human_rating(A, B)

    # Train reward model
    # Sometimes a Human evaluator can give a score
    reward = neural_network(prompt, response)
    loss = -log(P(preferred) / (P(preferred) + P(non_preferred)))

Stage 3 - RL Fine-Tuning:
    # Now use reward model to improve responses
    while training:
        prompt = get_prompt()
        response = model_generate(prompt)
        reward = reward_model(prompt, response)

        # Update model to maximize reward
        # But don't deviate too far from SFT model
        loss = ppo_loss(reward, response_probability)
```

The PPO loss here is what I want you to focus on. We need to prevent the model from becoming too different from our SFT model:

```
# PPO loss details:
old_prob = sft_model_probability(response)
new_prob = current_model_probability(response)
ratio = new_prob / old_prob

# Clip ratio to prevent huge changes
clipped_ratio = clip(ratio, 1-ε, 1+ε)

# Take minimum to be conservative
loss = -min(ratio * reward, clipped_ratio * reward)
```

I think token level details are kinda important to understand, how we model LLMs as a RL problem:

```
# For a response like: "Python is great"
token_1 = "Python"  # First action
token_2 = "is"      # Second action
token_3 = "great"   # Third action

# But we only get ONE reward for the whole sequence!
reward = reward_model(prompt, "Python is great")
```

```
# Problem: How do we know which tokens helped/hurt?
# Was "Python" good but "is great" bad?
# Or was the whole combination good?
```

The solution is to use a reward attribution scheme like this (very simplified version):

```
# Instead of one reward at the end:
1. Generate multiple completions:
   response_1 = "Python is great"
   response_2 = "Python is okay"
   response_3 = "Python is bad"

2. Get rewards for each:
   reward_1 = reward_model(prompt, response_1)  # 0.9
   reward_2 = reward_model(prompt, response_2)  # 0.5
   reward_3 = reward_model(prompt, response_3)  # 0.2

3. Compare shared prefixes:
   # All start with "Python is"
   # But lead to different rewards
   # This tells us about the value of the next token!
```

Now for the actual training process:

```
# For each training batch:
1. Generate multiple responses with current policy
   keeping track of probability of each token

2. Compute rewards for full sequences

3. For each token position:
   # Get value estimate from critic
   value = critic_network(prompt + tokens_so_far)

   # Advantage is comparing similar sequences
   # that differ after this token
   advantage = compare_similar_sequences(position)

   # PPO update for this specific token:
   old_prob = token_probability_from_sft
   new_prob = current_token_probability
```

```
   ratio = new_prob / old_prob
   clipped_ratio = clip(ratio, 1-0.2, 1+0.2)

   # Update to maximize reward but stay close to SFT
   loss = min(ratio * advantage, clipped_ratio * advantage)

4. Only update if KL divergence from SFT is small
   # This prevents "model collapse" where responses
   # become repetitive or lose language understanding
```

This process is known as RLHF (Reinforcement Learning from Human Feedback). It's kinda important and probably influences your life of a daily basis :)

34. Using PPO for RLHF is very expensive to run, and has a few issues:

```
# PPO needs 4 separate networks in memory:
1. Policy network (current actions)
2. Reference policy network (old policy)
3. Reward network (evaluate actions)
4. Value network (estimate future rewards, the critic)

# This creates real problems:
- Massive memory usage
- Slower training loops
- Harder to implement
- More things that can go wrong

# Plus PPO does something kinda weird:
generation = "This is a good response"
# PPO treats each word as separate action:
action_1 = "This"       # Update policy for this
action_2 = "is"         # And this
action_3 = "a"          # And this
action_4 = "good"       # And this
action_5 = "response"   # And this

# But that's not really how language works!
# The whole response is ONE coherent thought
```

Cohere (another big AI lab for LLMs) created an algorithm called RLOO (REINFORCE Leave-One-Out). I want you to test yourself, I'm going to explain what the algorithm does and I want you to gague if you fully understand how it works.

```
# RLOO only needs 3 networks:
1. Policy network
2. Reference policy network
3. Reward network
# No value network needed!

# And treats the whole response as ONE action:
response = "This is a good response"
# One action, one reward, one update!

# But here's the really clever part:
# For each batch of responses to same prompt:
responses = [
    "This is good",
    "That works well",
    "Maybe try this",
    "How about that"
]

# Instead of using a value network use other responses as baseline!
for response in responses:
    # Baseline = average reward of OTHER responses
    baseline = average_reward(all_responses_except(response))
    advantage = reward(response) - baseline
```

Why does this work so well?

```
# Example with 3 responses to "Write a poem":
responses = [
    "Roses are red..." -> reward = 8.0
    "In the moonlight..." -> reward = 6.0
    "Through the forest..." -> reward = 4.0
]

# For "Roses are red":
baseline = average(6.0, 4.0) = 5.0
advantage = 8.0 - 5.0 = 3.0  # Better than others!

# For "In the moonlight":
baseline = average(8.0, 4.0) = 6.0
advantage = 6.0 - 6.0 = 0.0  # About average
```

```
# For "Through the forest":
baseline = average(8.0, 6.0) = 7.0
advantage = 4.0 - 7.0 = -3.0  # Worse than others!
```

The training process is lwokey easy to understand too:

```
# RLOO training loop:
while training:
    # 1. Generate multiple responses per prompt
    prompt = "Write a poem"
    responses = generate_k_responses(prompt)

    # 2. Get rewards for all responses
    rewards = reward_model(responses)

    # 3. For each response:
    for response in responses:
        # Calculate baseline from other responses
        baseline = average_reward(other_responses)

        # Compute advantage
        advantage = reward(response) - baseline

        # Simple REINFORCE update!
        # If advantage > 0: increase probability
        # If advantage < 0: decrease probability
        update = log_prob(response) * advantage
```

RLOO vs PPO for LLMs:

1. Uses 25% less memory than PPO (no value network!).

   In practice uses like 70% less memory because:

   - No per-token storage (treats whole sequence as one action)
   - No value network activations
   - No multi-epoch replay buffer
2. Treats language more naturally (whole responses as actions)
3. Gets baselines "for free" from batch comparisons
4. Runs 2-3x faster than PPO in practice
5. Works just as well as PPO for actual performance

35. Recently OpenAI announced their o3 model that uses reinforcement learning at test time to "think" before generating outputs. They haven't revealed exactly how it works, but let's try to imagine how we could solve this problem using what we've learned!

    Let's say we give an LLM this question: "What's 123 * 456?" A normal LLM might just try to generate an answer immediately. But humans don't work that way - we think through steps, catch our mistakes, and revise our thinking. Can we make an LLM do the same thing?

    Let's try using MCTS to reason about outputs:

```
# imagine we're the LLM getting this prompt:
prompt = "What's 123 * 456?"

# in normal LLM we'd just generate directly:
direct_response = generate("The answer is...")

# but what if instead we treated THINKING as actions?
# each "thought" is an action we can take
possible_thoughts = [
    "Let's break this into steps...",
    "First multiply 123 by 6...",
    "Then multiply 123 by 50...",
    "Finally multiply 123 by 400..."
]

# now we can use MCTS to explore these thought paths
# Selection: pick promising thinking paths using UCB
# Remember UCB from before?
UCB = thought_value + c * sqrt(log(total_visits)/thought_visits)

# Expansion: generate new possible thoughts
# Simulation: continue the thought process
# Example simulation path:
path = [
    "Let's break this down:",
    "123 * 6 = 738",
    "123 * 50 = 6150",
    "123 * 400 = 49200",
    "Adding: 738 + 6150 + 49200 = 56088"
]
```

    How do we know if a thought path is good? We need some way to evaluate it. Remember reward functions from earlier? We could use another LLM as a reward model (or another prompt that evaluates outputs) or a code interpreter:

```
# reward model could look at each path and score:
# - Does it break problem into steps?
# - Are calculations correct?
# - Does it reach a clear answer?

good_path = [
    "Let's solve step by step:",
    "First 123 * 6 = 738",
    "Then 123 * 50 = 6150"
]
reward_model(good_path) # High reward

bad_path = [
    "Well, 123 is a number",
    "And 456 is also a number",
    "So maybe..."
]
reward_model(bad_path) # Low reward

# Regular MCTS from here, we can build a reasoning tree based on the rewar
# Here is an example tree we might end up with!
# Each node has:
# - A thought/step
# - Number of visits (N)
# - Total reward (Q)
# - UCB score

# Root node (starting point)
"What's 123 * 456?"
N = 10, Q = 7.5
|
|----"Let's break this down step by step..."  # Most visited path
|    N = 5, Q = 4.8
|    |
|    |----"First, 123 * 6 = 738"             # Strong reward
|    |    N = 3, Q = 3.2
|    |    |
|    |    |----"Next, 123 * 50 = 6150"       # Continues well
|    |    |    N = 2, Q = 2.1
|    |    |    |
|    |    |    |----"Finally, 123 * 400 = 49200"  # Complete path
|    |    |         N = 1, Q = 1.0
```

```
|    |
|    |----"Let's use distributive property..."  # Alternative approach
|         N = 2, Q = 1.6
|         |
|         |----"123 = 100 + 20 + 3"
|              N = 1, Q = 0.8
|
|----"We can work right to left..."          # Less visited path
|    N = 3, Q = 1.5
|    |
|    |----"6 times 3 is 18..."               # Lower reward
|         N = 1, Q = 0.4
|
|----"Well, 123 is a number..."              # Dead end
     N = 2, Q = 0.2
     |
     |----"And 456 is also a number..."      # Very low reward
          N = 1, Q = 0.1
```

I want to be clear - this probably isn't how o3 works. Noam Brown confirmed that o3 doesn't use MCTS. But that's not the point. When you see RL in any context I want you to be able to come up with ideas. Ideas that you know work, ideas that you understand on a deeper level. I'll be pretty happy if I was able to leave you with that!

36. Honestly, this is already such so much information that we should really stop here (also I have work to do). A significant number of RL resources bring up random topics without context & jump straight into the math/code. Imo, this is a terrible way to learn (at least for me). I prefer a "I'm not going to read the rule book" style of learning. Now that we are at the end, hopefully you are convinced too! Anyway, this was really fun :)

Anyway. This is the end.

Thank you, ily <3

If you want to support my work

1. follow me on twitter https://twitter.com/naklecha
2. or, buy me a coffee https://www.buymeacoffee.com/naklecha

but honestly, if you made it this far you already made my day :)

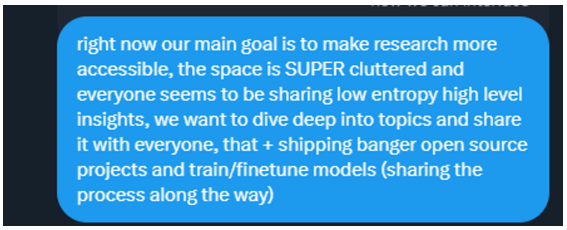
What motivates me?

My friends and I are on a mission - to make research more accessible!

We created a lab (soon to be non-profit) called A10 - AAAAAAAAAA.org

A10 twitter - https://twitter.com/aaaaaaaaaaorg

A10 thesis (this screen shot is from a year ago, we are doubling down on creating a meaningful bridge for new people to enter fields that we find fascinating)



see you soon :)