Last time..

Generative models

DRL
(Deep Reinforcement Learning)

Key concepts
Value, Policy and Hybrid based

Course: Deep Reinforcement Learning

Book: Reinforcement Learning: An Introduction

Agenda

- Applications
- What is RL, key concepts, RL vs. DRL
- Value-based:
  - Q-learning
  - Deep Q-learning (DQN)
  - Improvements to DQN
- Policy-based:
  - REINFORCE with Monte Carlo & Policy Gradients
- Hybrid-based / Actor Critic:
  - A2C, A3C, PPO

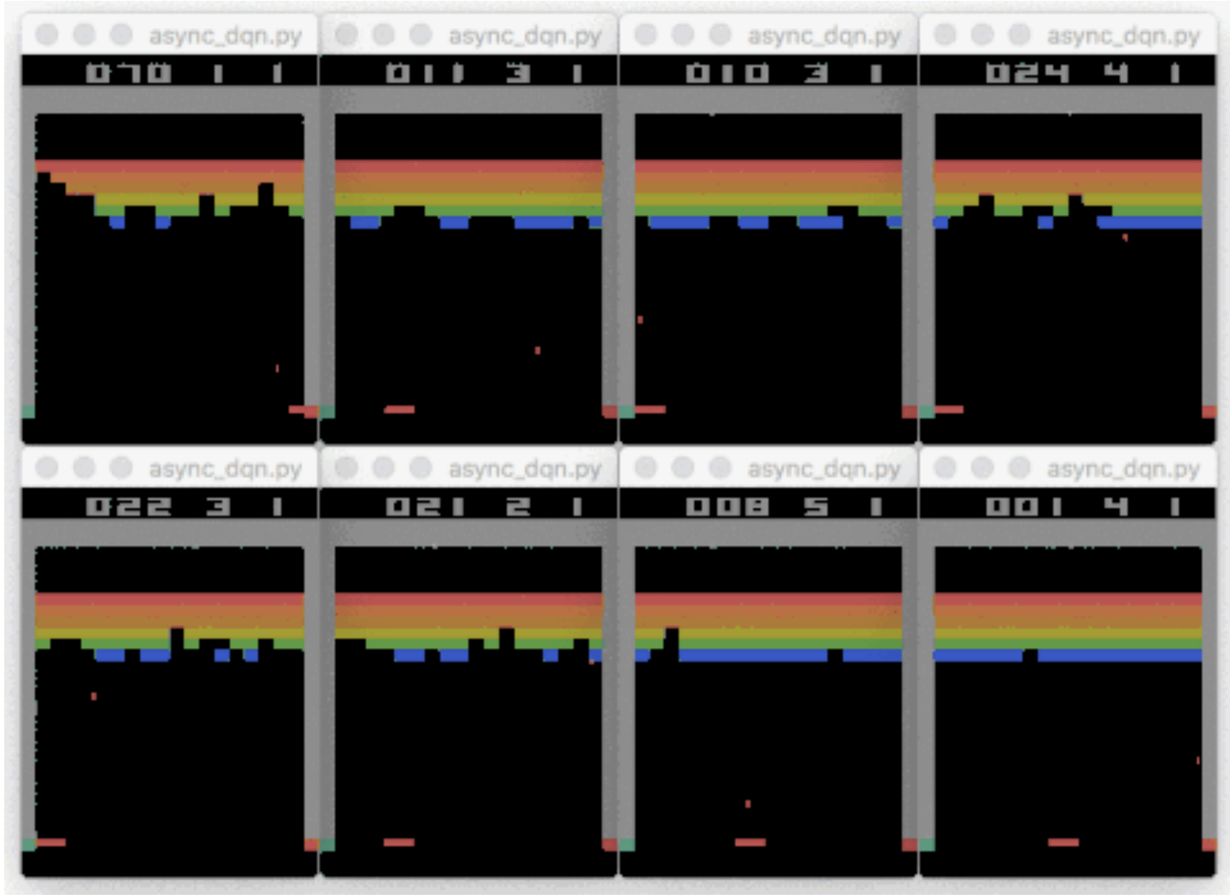
Wayve: Learning to drive in a day

Learning to Drive Smoothly in Minutes

What is RL, key concepts, DeepRL

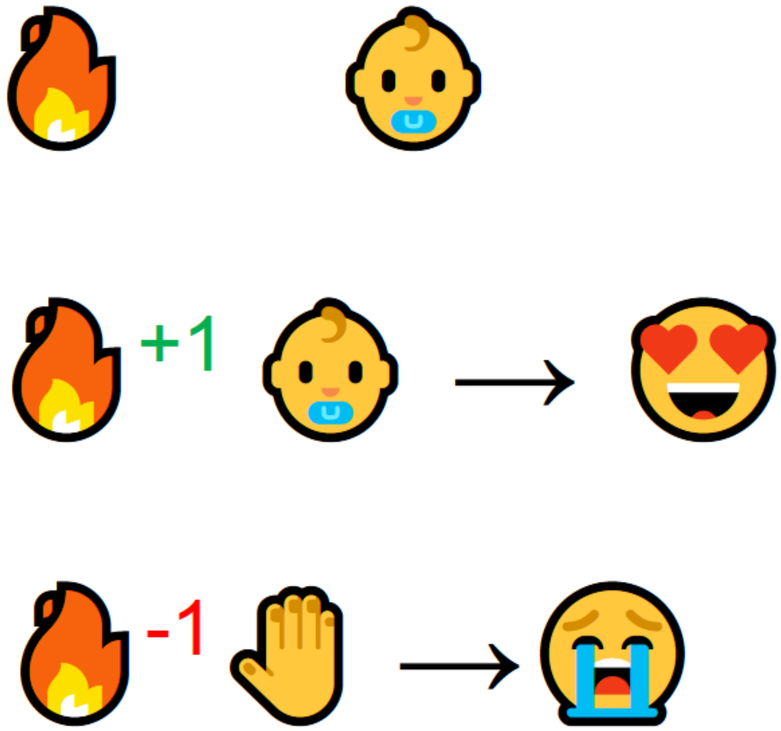
Reinforcement learning

- type of Machine Learning (in addition to supervised & unsupervised), more like humans learn, trail an error.
- where an **agent** learn how to behave in a **environment** by performing **state** related **actions** and seeing the result by getting some **reward**.
- fascinating area, e.g. AlphaGo (Zero)
- big field, focus on key methods.
- important to understand key concepts before starting to implement DRL agents.
- **idea**: an agent learn by interacting with the environment and it's states and receiving rewards for performing good actions.



Reinforcement learning (2)

- **How humans (or animals) learn:**
  - **Learning** from interaction with the environment comes from our **natural experiences,** e.g. imagine you're a **child**, in a living room, see a fireplace and approach it:
  - Close by it's worm and good, i.e. you get a positive reward +1
  - But if you try to touch you burn your hand and get a negative reward -1
  - **You learn**: **fire** is positive when you are a sufficient distance away, because it produces warmth. But get too close to it and you will be burned.
- **How machines learn:**
  - Reinforcement Learning is just a computational approach of learning from action.
  - **Idea**: an agent will learn from the environment (state) by interacting with it and receiving rewards for performing actions



The RL process

- **Example**: an agent learning to play Super Mario Bros

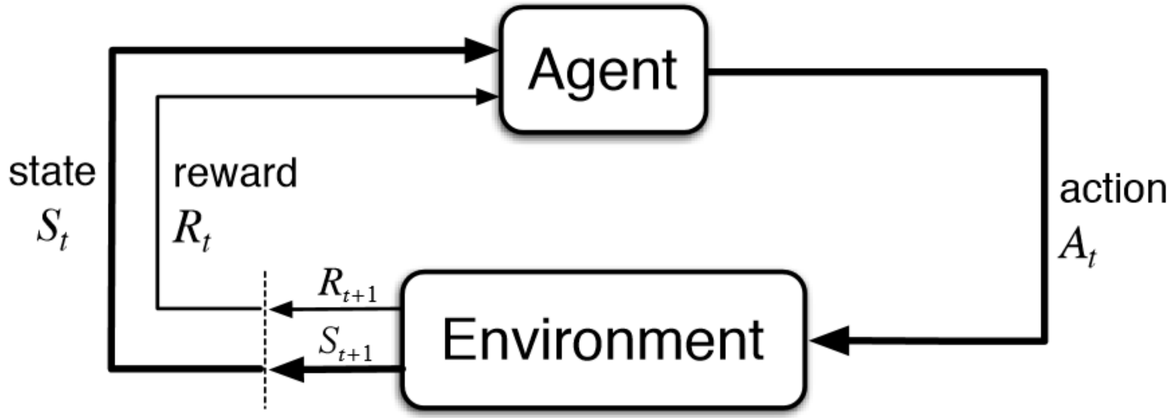
- The RL process can be modeled as a **loop** that outputs a sequence of **state, action and reward**:

  - The agent receives **state (s0)** (pixel frame) from the environment (Super Mario Bros game).

  - Based on that state, the agent takes an **action (a0)** (e.g. move right)

  - The environment gives some **reward (r1)** to the agent (e.g. not dead: +1) and transitions to a **new state (s1)** (i.e. new frame)

- The **goal** of the agent is to maximize the expected future cumulative reward.

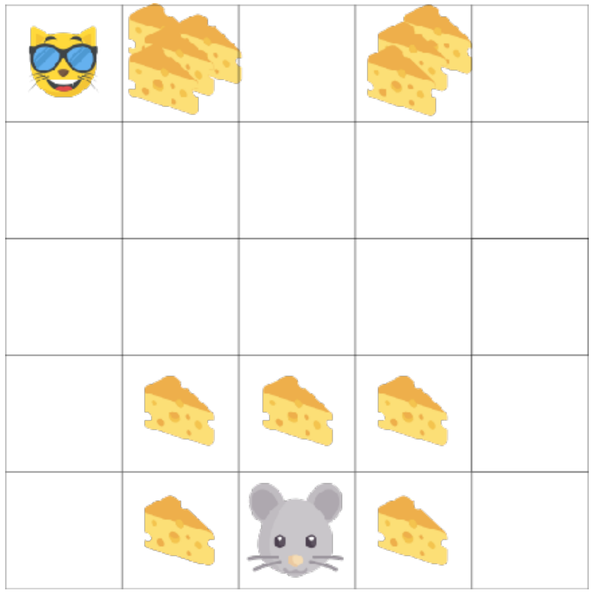


The RL process (2)

- RL is based on the **Reward Hypothesis**: all **goals** can be described by the maximization of the expected cumulative reward / to have the **best behavior** we need to maximize the expected **cumulative** reward.

$$G_t = \sum_{k=0}^{T} R_{t+k+1} \qquad G_t = R_{t+1} + R_{t+2} + \dots$$

- **Discounted rewards**: short-term rewards (close in time) are more **probable** to happen, since they are more predictable than the long term future reward.
  - **Discount** rate: gamma (between 0 and 1)
    - larger gamma (e.g. 0.9), smaller discount («rabatt»), the learning agent cares more about the **long** term reward (cheese further away).
    - smaller gamma (e.g. 0.1), bigger discount, the learning agent cares more about the **short** term reward (the nearest cheese). (Think: cum.reword if takes 3 steps to reach a 100p reward)
  - Each reward will be discounted by gamma to the exponent of the time step. As the time step increases, the cat gets closer to us, so the future reward is less and less probable to happen.

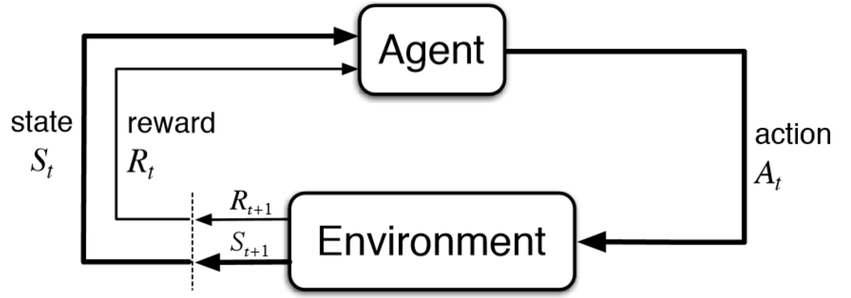


$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} where \gamma \in [0, 1)$$

$$R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \dots$$

Key concepts

- **agent**: An RL algorithm that learns to behave optimally in a given environment. Often implemented as a deep neural network.
- **environment**: The source of states and rewards. If we're building an RL algorithm to play a game, then the game is the environment.
- s: **state**: State-spaces are the set of all possible states a system can be in.
- a: **action**: The action-space is the set of all possible actions for a particular state.
- r: **reward**: signals produced by the environment that indicate the relative success of taking an action in a given state.
- Future (accumulated) rewards:
  - V(s): Value-function: the **value** of being at that state
  - Q(s,a): Q(uality)-function: the **quality** of taking an action a at a given state s
  - A(s,a): advantage-function: the **advantage** of taking that action at that state
  - pi(s) = a: (deterministic) policy: maps state to action
  - pi(a given s): Stochastic policy: outputs a probability distribution over actions given status s



$$Q(s,a) = A(s,a) + V(s)$$

Monte Carlo $$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$$

TD Learning $$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

Previous estimate; Reward t+1; Discounted value on the next step; TD Target

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

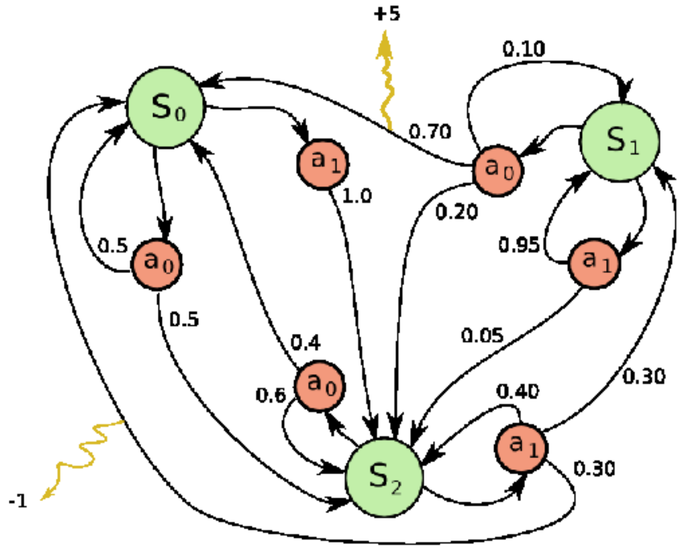
New Q value for that state and that action; Current Q value; Reward for taking that action at that state; Learning Rate; Discount rate; Maximum expected future reward given the new s' and all possible actions at that new state

Markov property / Markov Decision Process (MDP)

- **Markov property:**
  - It is possible to make the best decisions without reference to a history of prior states.
  - The future is independent of the past given the present

$$P[St+1 \mid St] = P[St+1 \mid S1, \ldots\ldots, St]$$

- **Markov Decision Process (MDP):**
  - A decision-making process (e.g. a game or any other control task) that exhibits the Markov property



Episodic or Continuous tasks

- Task: instance of a RL problem
- **Episodic task**:
  - starting point and an **ending** point (the agent reaches a “terminal state”)
  - **episode**: a list of States, Actions, Rewards, and New States
  - rewards are only received at the end of the game
  - then we start a new game with the added knowledge, the agent makes better decisions with each iteration.
  - Example: **maze** environment: 1) We always start at the same starting point. 2) We terminate the episode if the cat eats us or if we move > 20 steps. 3) At the end of the episode, we have a list of State, Actions, Rewards, and New States. 4) The agent will sum the total rewards (to see how well it did). 5) Then start a new game with this new knowledge. 6) By running more and more episodes, the agent will learn to play better and better.
  - Example: **Super Mario Bros**: an episode begin at the launch of a new Mario game and ends when you’re killed or you’re reach the end of the level.

Episodic or Continuous tasks (2)

- **Continuous tasks:**
  - Tasks that continue «forever» (no terminal state)
  - agent has to learn how to choose the best actions and simultaneously interacts with the environment
  - Example: **Self-driving car**: doesn't have episodes, it rides continuously (hopefully..). But it has to choose the best actions to ride safely, and simultaneously interact with the environment.
  - Example: an agent that do automated **stock trading**: for this task, there is no starting point and terminal state, the agent keeps running until we decide to stop him.

Monte Carlo vs TD Learning methods

- Two ways of **learning**: MC and TD learning
- **Monte Carlo** (MC) learning:
  - When the episode **ends** (e.g. the agent reaches a "**terminal** state"), the agent looks at the **total cumulative reward** to see how well it did. In a Monte Carlo approach, rewards are only received at the end of the game. Then, we start a **new game** with the **added knowledge**. The agent makes **better decisions** with each iteration.
  - Example: **maze:**
    - 1) We always start at the **same** starting point. 2) We terminate the episode if the cat **eats** us or if we move > 20 steps. 3) At the end of the episode, we have a **list** of State, Actions, Rewards, and New States. 3) The agent will sum the **total rewards Gt** (to see how well it did). 4) It will then **update** V(st) based on the formula above. 5) Then start a **new** game with this new knowledge. 6) By running more and more **episodes**, the agent will learn to play better and better

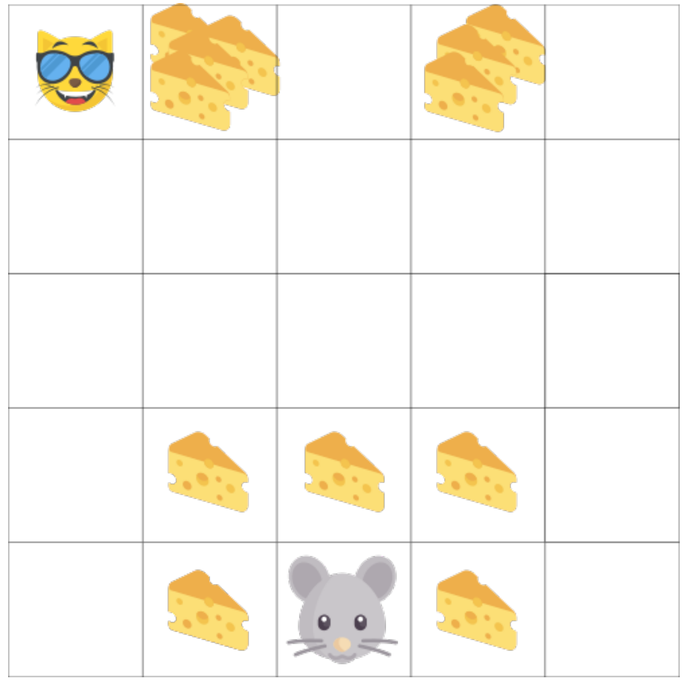
$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$$

Maximum expected future reward starting at that state

Former estimation of maximum expected future reward starting at that state

learning rate

Discounted cumulative rewards



Monte Carlo vs TD Learning methods (2)

- **Temporal Difference** Learning:
  - learning at each **time step** instead of learning only at the end of the episode
  - **update** the maximum expected future reward **estimation** V for the non-terminal states St occurring at that experience.
  - This method is called TD(0) or one step TD (update the value function after any individual step).
  - TD methods only wait until the next time step to update the value estimates. **At time t+1 they immediately form a TD target using the observed reward Rt+1 and the current estimate V(St+1).**
  - TD target is an **estimation**: in fact you update the previous estimate V(St) by updating it towards a one-step target.

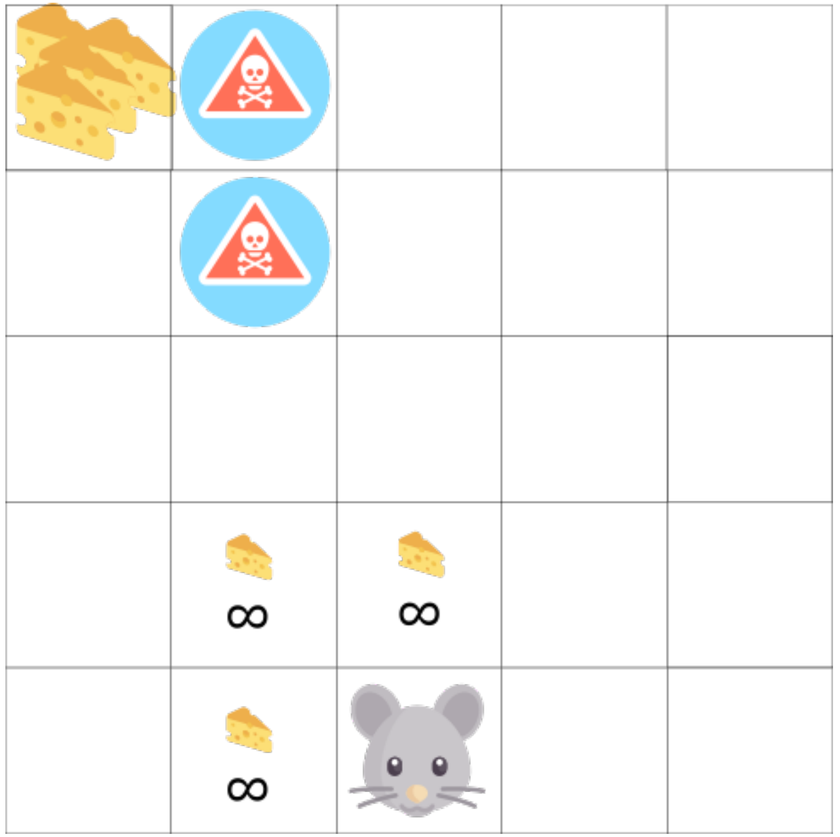
Monte Carlo $$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$$

TD Learning $$V(S_t) \leftarrow \underbrace{V(S_t)}_{\text{Previous estimate}} + \alpha[\underbrace{R_{t+1}}_{\text{Reward t+1}} + \underbrace{\gamma V(S_{t+1})}_{\text{Discounted value on the next step}} - V(S_t)]$$

TD Target

Exploration / Exploitation trade off

- **Important** topic: Exploration vs. Exploitation
  - **Exploration**: finding **more** information about the environment (to maximize futures reward).
  - **Exploitation**: exploiting **known** information to maximize the reward
- Example **trap**: infinite amount of small cheese close by (+1 each) vs. a gigantic sum of cheese further away (+1000):
  - if we only focus on reward, our agent will never reach the gigantic sum of cheese (i.e. exploration), instead it will only exploit the nearest sources of small rewards (i.e. exploitation)
  - we must define a **rule** that helps to handle the exploration/exploitation trade off

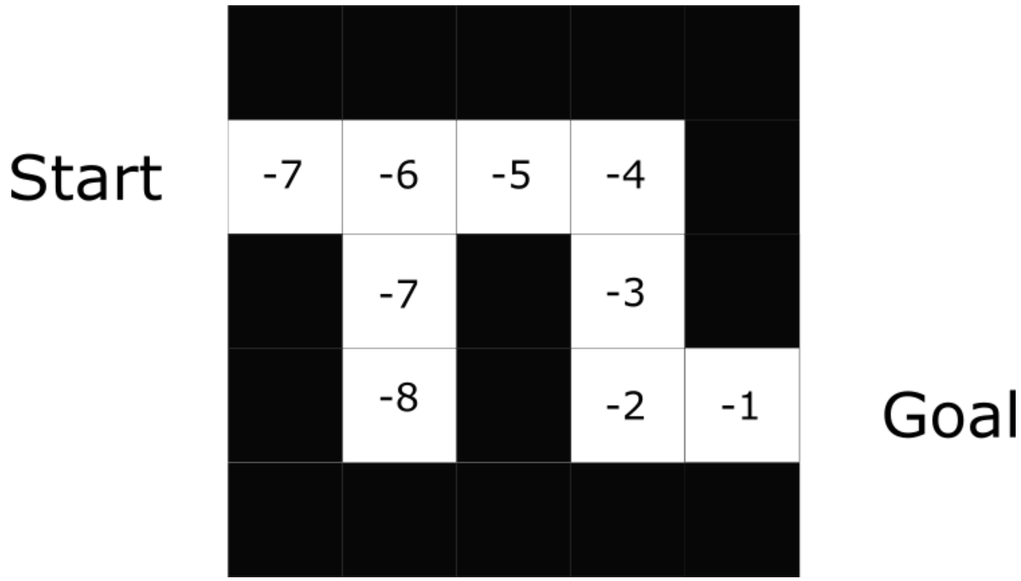


Three approaches to RL: Value-based

- Value, policy and hybrid-based RL
- **Value Based RL**:
  - **Goal**: optimize the value function V(s)
  - **Value function** for a given state: total amount of reward an agent can expect to accumulate, starting at that state.
  - The **agent** will use this value function to select which action to choose at each step. **The agent takes the action that leads to the state with the biggest value**.
  - Maze example: at each step we will take the biggest value, i.e. starting from -7, choose -6, -5 and so on.

$$v_\pi(s) = \mathbb{E}_\pi \left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \mid S_t = s \right]$$

Expected | Reward discounted | Given that state



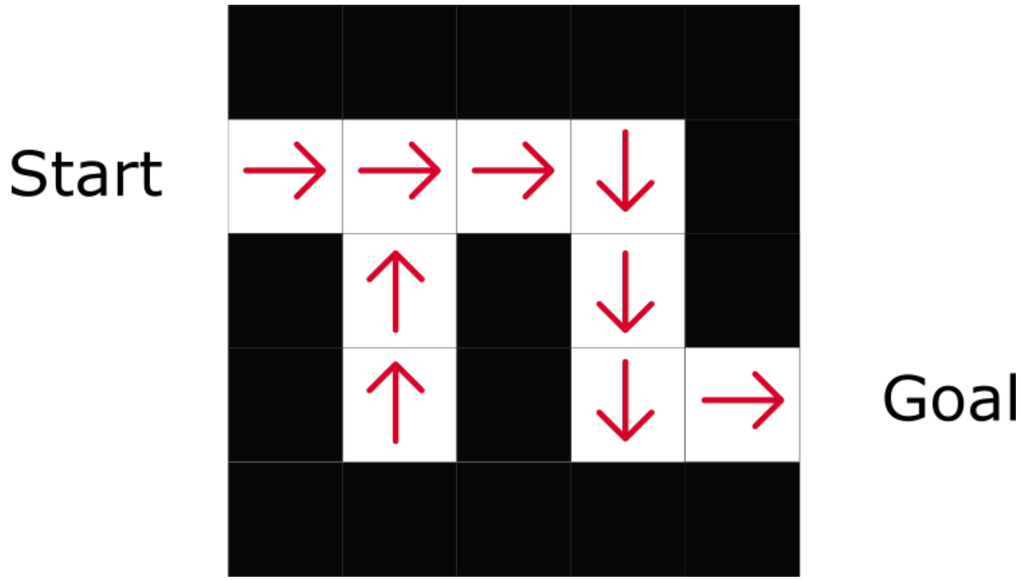
Three approaches to RL: Policy-based

- **Policy Based RL**:
  - Goal: **directly** optimize the **policy function** π(s) without using a value function
  - The policy is what defines the agent **behavior** at a given time: action = policy(state)
  - We **learn** a policy function. This lets us **map** each state to the best corresponding action.
  - We have two types of policy:
    - **Deterministic**: a policy at a given state will always return the same action
    - **Stochastic**: output a distribution probability over actions
  - Maze example: the policy directly indicates the best action to take for each steps

$$a = \pi(s)$$

Stochastic policy: $\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]$

Proba of — Take a particular action conditionned to a state

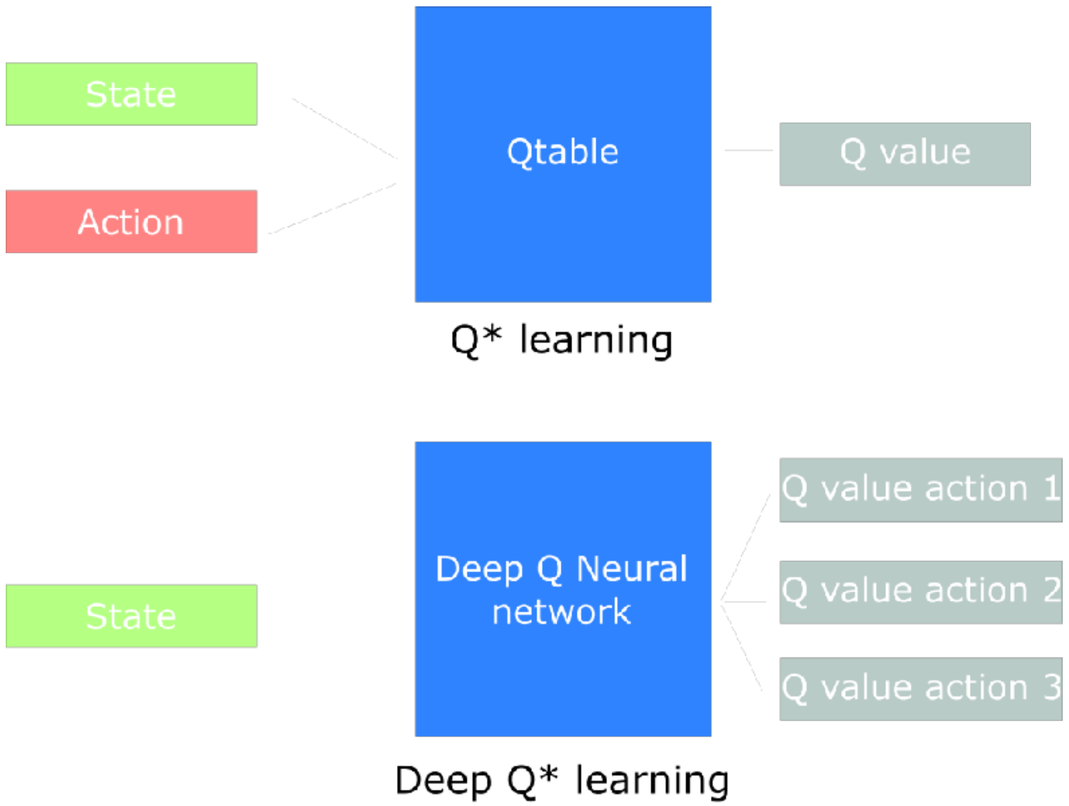


Three approaches to RL: Hybrid-based

- **Hybrid-based RL:**
  - **Goal**: best of both worlds, combination of policy and value based, overcome the shortcomings of previous methods.
  - mostly actor-critic based, SotA
- (**Other methods:** curiously-based, genetic-based etc.)
- (Model-based RL (vs. off-model)
  - **Goal**: model the environment, i.e. create a model of the behavior of the environment
  - **Problems**: each environment will need a different model representation, difficult to model real-world problems
  - **Not used** much today. Only works for very simple problems.)

Deep RL (DRL vs. RL)

- Deep RL: (the agent) uses a deep **NN** to solve RL problems .
- Example:
  - **Q-Learning**: use a traditional algorithm to create a **Q table** that helps us find what action to take for each state.
  - **Deep Q-Learning**: use a Neural Network to approximate the reward for each action given a state (**Q value**). Q for?

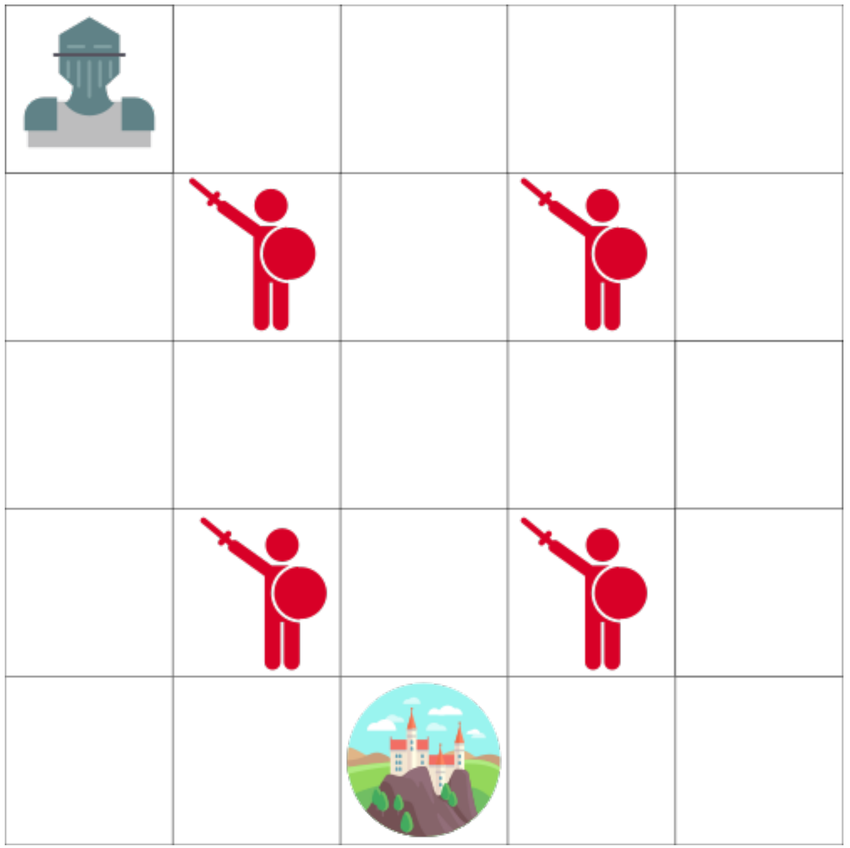


Value-based RL

Q-Learning
Deep Q-Learning
Improvements to Deep Q-Learning

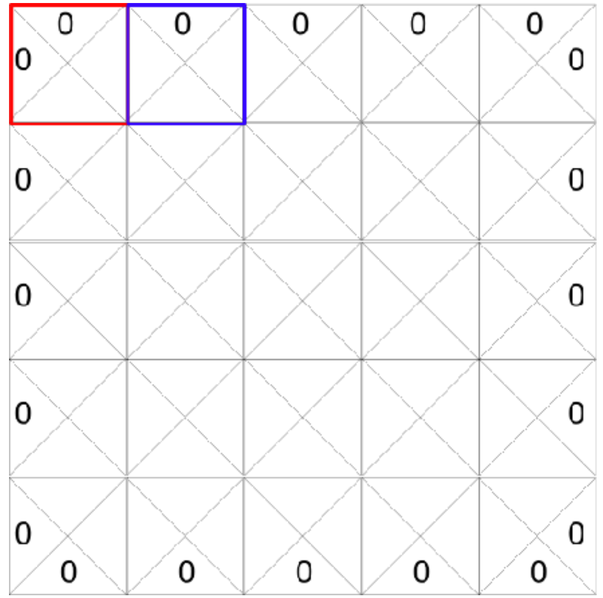
Q-Learning

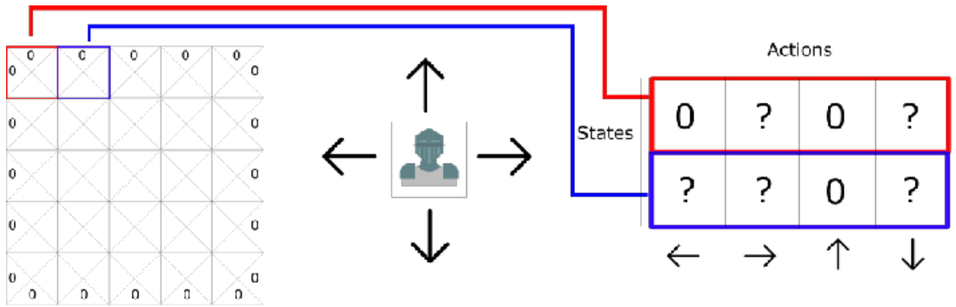
- a **value**-based RL algorithm.
- Big picture: the Knight and the Princess **example**:
  - **Goal**: save the princess trapped in the castle shown on the map as fast as possible.
  - **Rules**: You (the agent) can move one tile at a time. The **enemy** can't, but land on the same tile as the enemy, and you will **die**.
  - Can be solved using a "**points** scoring" system:
    - You lose one point **-1** at each step (losing points at each step helps our agent to be **fast**).
    - If you touch an enemy, you lose **-100** points, and the episode **ends**.
    - If you are in the castle you win, you get **+100** points and the episode **ends**.
  - How can we **create an agent** that will be able to do this?



Q-Learning: Q-table

- **Strategy**: create a **table** where we can **calculate** the maximum expected future **reward**, for each action at each state:
  - First: Transform **grid** into a **table**: Each state (i.e. tile) allows four possible actions, these are moving left, right, up, or down.
  - Use the table to find the **best action** to take for each state.
- **Q-table** (“Q” for “quality” of the action):
  - The **rows** will be the **states**. The **columns** will be the four **actions** (left, right, up, down). The value of each **cell** will be the maximum expected future **reward** for that given state-action **pair** (with the best policy given: we don’t implement a policy, instead we just improve our Q-table to always choose the best action, i.e policy involved in training: epsilon-greedy).
  - Q-table as a game “**cheat sheet**”: find the best action for a given state: find the correct line in the Q-table (=state) and then you find the highest score in that line (=action).
  - How do we **calculate** the values for each element of the Q table?





Q-Learning: Algorithm

- The **Action Value Function** (or "**Q-function**") takes two inputs: "state" and "action" and returns the expected future reward for that state-action pair.

- Q-function: a **reader** that scrolls through the Q-table to find the **line** associated with our **state**, and the **column** associated with our **action**. It returns the Q-value from the matching **cell**. This is the "expected future **reward**."

- **But**: before we **explore** the environment, the Q-table gives the same arbitrary **fixed value** (most of the time **0**). As we explore the environment, the Q-table will give us a better and better approximation by iteratively **updating** Q(s,a) using the **Bellman** Equation (next slide).

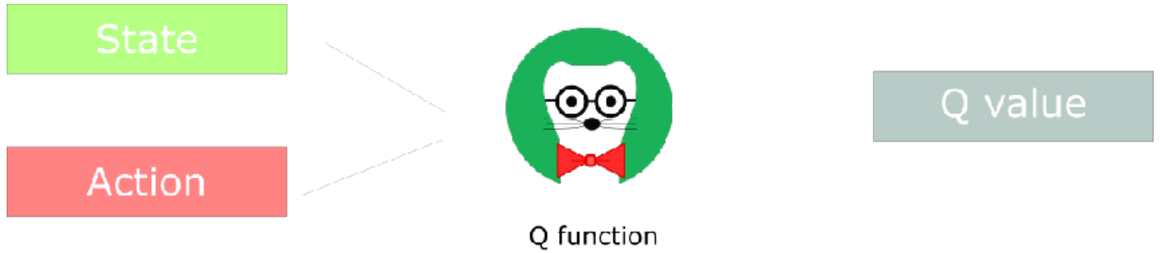
- Where in the alg. do the exploration take place and how?

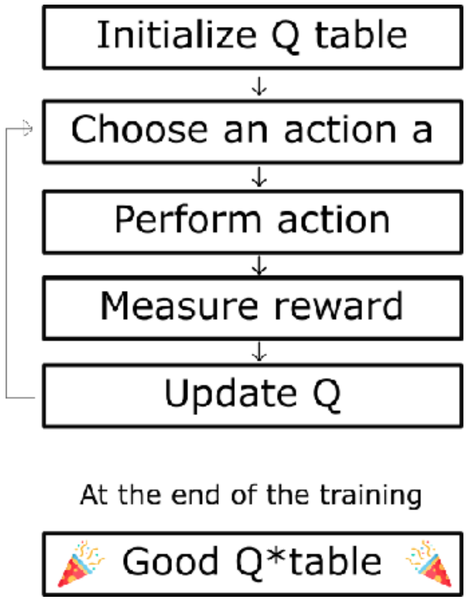
$$Q^{\pi}(s_t, a_t) = E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots | s_t, a_t]$$

Q value for that state given that action

Expected discounted cumulative reward ...

given that state and that action





Q-Learning: Algorithm (2)

- Step 1: **Initialize** Q-values: m rows (=# states), n cols (=# actions). Set all values to 0.

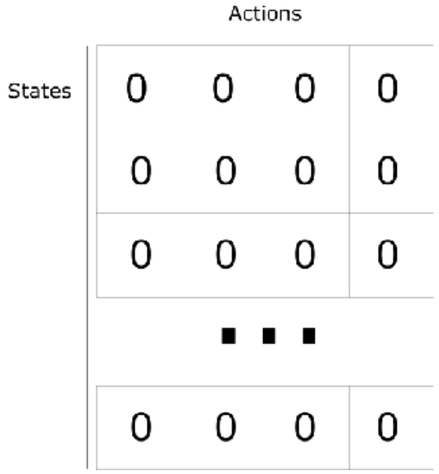
- Step 2: **Iterate** for life (or until learning is stopped)

  - Step 3: **Choose an action**: but after the initialization all Q-value equals **zero** (which is best?). This is were the **exploration/exploitation trade-off** comes in. We use the **<u>epsilon greedy</u>** strategy (policy):

    - Start with a big **epsilon** (=exploration threshold, i.e. close to 1) and reduce it progressively as the agent learns (towards 0). Generate a **random** number (uniform 0-1). <u>If</u> number **<** epsilon: **exploration** (don't know anything about the environment, choose random actions), <u>else</u>: **exploitation** (use what we already know to select the best action at each step).

  - Steps 4–5: **Evaluate**: take the **action** a that we chose in step 3, this returns us a **new state** s' and a **reward** r. Then update Q(s,a) using the **<u>Bellman equation</u>**.

1. Initialize Q-values ($Q(s, a)$) arbitrarily for all state-action pairs.
2. For life or until learning is stopped...
3. Choose an action (a) in the current world state (s) based on current Q-value estimates ($Q(s, \cdot)$).
4. Take the action (a) and observe the the outcome state (s') and reward (r).
5. Update $Q(s, a) := Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]$



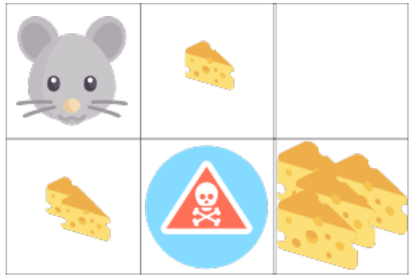
$$NewQ(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma max\, Q'(s',a') - Q(s,a)]$$

New Q value for that state and that action | Current Q value | Learning Rate | Reward for taking that action at that state | Discount rate | Maximum expected future reward given the new s' and all possible actions at that new state

```
New Q value =
   Current Q value +
   lr * [Reward + discount_rate * (highest Q value between possible
actions from the new state s' ) – Current Q value ]
```

Q-Learning: Example

- **Scoring** system: One cheese: **+1**, Two cheese: **+2**, Eat rat poison: **-10** (end of the episode), Big pile of cheese: **+10** (end of episode).
- Hyper-parameters: alfa (lr): 0.1, discount rate: 0.9, epsilon: 1 (start).
- Algorithm:
  - Step 1: **Init** the Q-table: with zeros, Step 2: **Iterate:**
    - Step 3: **Choose** an **action**: from starting position: going right or down, big epsilon: 1, choose action randomly: e.g. move right
    - Step 4: Take the action a = «move right»: r = 1, s' = «small cheese»



	←	→	↑	↓
Start	0	0	0	0
Small cheese	0	0	0	0
Nothing	0	0	0	0
2 small cheese	0	0	0	0
Death	0	0	0	0
Big cheese	0	0	0	0

	←	→	↑	↓
Start	0	0	0	0
Small cheese	0	0	0	0
Nothing	0	0	0	0
2 small cheese	0	0	0	0
Death	0	0	0	0
Big cheese	0	0	0	0

Q-Learning: Example (2)

- Step 5: **Update** the Q-function:
  - First, we calculate the **change** in Q value ΔQ(start, right): 1
  - Then we add the learning rate multiplied by the change to the initial Q value: 0.1
  - We've just updated the first Q value, now we need to do that again and again until the learning is stopped.

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

New Q value for that state and that action | Current Q value | Learning Rate | Reward for taking that action at that state | Discount rate | Maximum expected future reward given the new s' and all possible actions at that new state

$$NewQ(start,right) = Q(start,right) + \alpha[\Delta Q(start,right)]$$

$$\Delta Q(start,right) = R(start,right) + \gamma\ max\ Q'(1cheese,a')) - Q(start,right)$$

$$\Delta Q(start,right) = 1 + 0.9 * max(Q'(1cheese,left), Q'(1cheese,right), Q'(1cheese,down)) - Q(start,right)$$

$$\Delta Q(start,right) = 1 + 0.9 * 0 - 0 = 1$$

$$NewQ(start,right) = 0 + 0.1 * 1 = 0.1$$

	←	→	↑	↓
Start	0	0.1	0	0
Small cheese	0	0	0	0
Nothing	0	0	0	0
2 small cheese	0	0	0	0
Death	0	0	0	0
Big cheese	0	0	0	0

Implementation: Q-learning with numpy and OpenAI Taxi-v2

Deep Q-Learning (DQL)

- **Q-Learning**: an algorithm which produces a **Q-table** that an agent uses to find the best action to take given a state.
  - Producing and updating a **Q-table** can become **ineffective** in **big state space** environments.
- **Deep Q-Learning**: Instead of using a Q-table, we'll implement a NN (DQN) that takes a **state** and approximates **Q-values** for each action based on that state.
  - Example application: create an agent that learns to play Doom or other video games.

Deep Q* learning

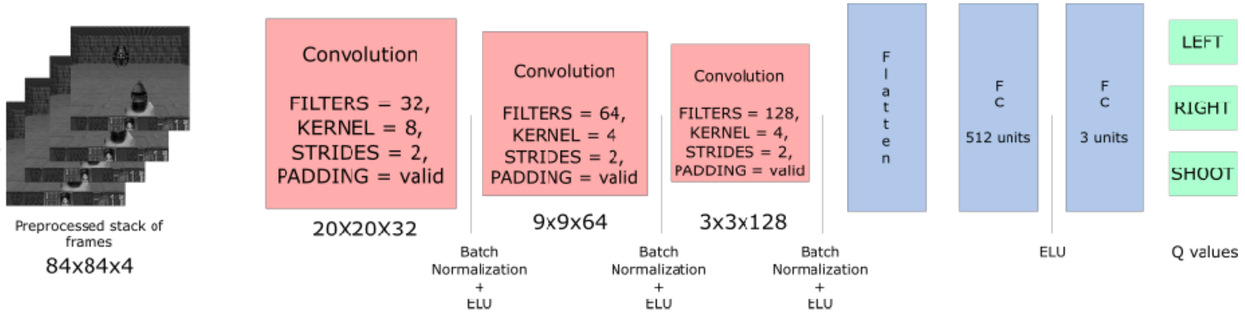
DQL: Architecture

- **Overall**: The Deep Q Network (DQN) takes a **stack of four frames** as an input and output a **vector of Q-values** for each action possible in the given state. We need to take the **biggest** Q-value of this vector to find our **best** action. In the beginning the agent does really **badly**, but over time it begins to **associate** frames (states) with best actions to do.

- CNN **Architecture**:

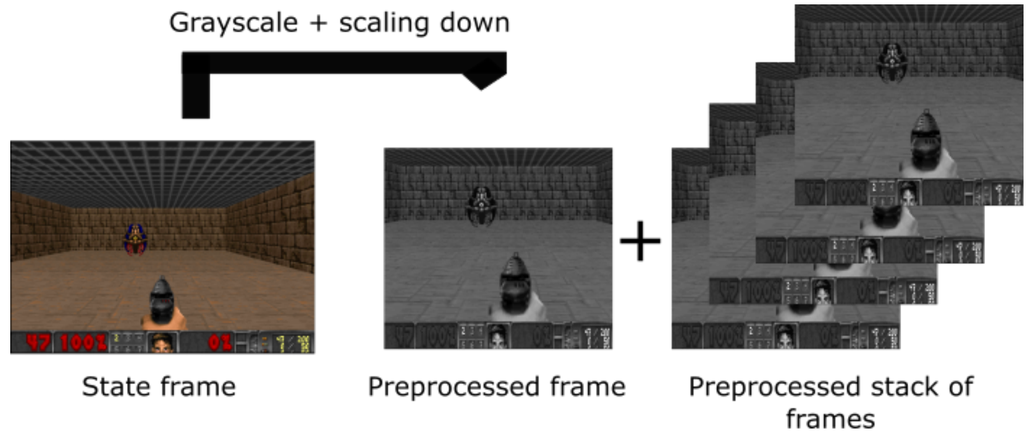
  - 3 convolutional layers w/ ELU activation: exploit spatial (& temporal) relationships in images (& stack)

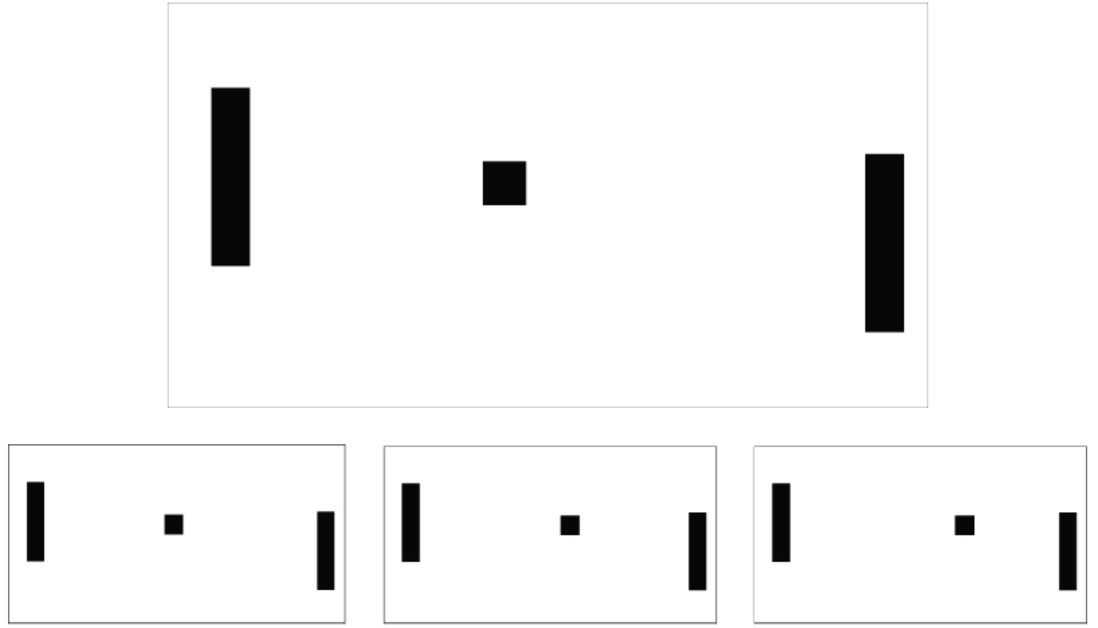
  - 2 FC-layers: produces the Q-value estimation for each action



DQL: Architecture (2)

- **Preprocessing**: important to reduce state complexity to reduce the training time.
  - **grayscale**: from 3 to 1 channel, color not important here.
  - **crop**: seeing the roof in each frame is not useful.
  - reduce the **size** of the frame, and **stack** four sub-frames together.
- **Temporal** limitation:
  - Game of Pong: one frame is not enough to have a sense of **motion**. Adding two or more frames we can see that the ball is going to the right.
  - The agent needs to know **where** and how **fast** objects are moving to make a correct decision.
  - Alternatives to stacked frames?





DQL: Experience Replay

- Avoid **forgetting** previous experiences:
  - RL process: At each time step, we receive a tuple (state, action, reward, new_state). We learn from it (we feed the tuple to our neural network), and then throw this experience away.
  - Our problem is that we give **sequential** samples from interactions with the environment to our neural network. And it tends to **forget** the previous experiences as it overwrites with new experiences. For instance, if we are in the first **level** and then the second (which is totally different), our agent can forget how to behave in the first level.
  - **Solution**: create a “replay buffer” that stores experience tuples while interacting with the environment, and then we sample a small batch of tuples to feed to our neural network. This prevents the network from only learning about what it has immediately done.

DQL: Experience Replay (2)

- Reducing **correlation** between experiences:
  - Every action affects the next state and this outputs a **sequence** of experience **tuples** which can be highly **correlated**.
  - If we train the network in sequential order, we risk our agent being influenced by the effect of this correlation.
  - By sampling from the replay buffer at random, we can break this correlation. This prevents action values from oscillating or diverging catastrophically.
  - Example: if we shoot a monster the probability that the next monster comes from the same direction is 70%. Problem: approach increases the value of using the right gun through the entire state space.
  - Strategy: stop learning while interacting with the environment (helps avoid being fixated on one region of the state space and prevents reinforcing the same action over and over).

DQL: Algorithm

- Remember that we **update** our Q value for a given state and action using the **Bellman** equation.

- For DQL we want to update the NN **weights** to reduce the error.

- The **TD error** is calculated by taking the difference between our **Q_target** (maximum possible value from the next state) and **Q_value** (our current prediction of the Q-value).

- Algorithm consists of two processes:

  - We sample the environment by performing actions and store the observed experience tuples in a **replay** memory.

  - Select a small batch of tuples randomly and learn from it using a gradient descent update step.

<u>Implementation: DQL & Space Invaders</u>

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

New Q value for that state and that action

Current Q value

Learning Rate

Reward for taking that action at that state

Discount rate

Maximum expected future reward given the new s' and all possible actions at that new state

$$\Delta w = \alpha[(R + \gamma\, max_a\, \hat{Q}(s',a,w)) - \hat{Q}(s,a,w)]\, \nabla_w \hat{Q}(s,a,w)$$

Change in weights

learning rate

Maximum possible Qvalue for the next_state (= Q_target)

Current predicted Q-val

TD Error

Gradient of our current predicted Q-value

```
Initialize Doom Environment E
Initialize replay Memory M with capacity N (= finite capacity)
Initialize the DQN weights w
for episode in max_episode:
    s = Environment state
    for steps in max_steps:
         Choose action a from state s using epsilon greedy.
         Take action a, get r (reward) and s' (next state)
         Store experience tuple <s, a, r, s'> in M
         s = s' (state = new_state)

         Get random minibatch of exp tuples from M
         Set Q_target = reward(s,a) +  γmaxQ(s')
         Update w =  α(Q_target - Q_value) *  ∇w Q_value
```

Improvements to Deep Q-Learning

- Deep Q-Learning was **introduced** in 2014. Since then, a lot of **improvements** have been made.
- The following are four strategies that improve the training and the results of our DQN agents significantly:
  - fixed Q-targets
  - double DQNs
  - dueling DQN (aka DDQN)
  - Prioritized Experience Replay (aka PER)

Improvements: Fixed Q-targets

- We saw in the **Deep Q Learning part** that, when we want to calculate the **TD error** (aka the loss), we calculate the difference between the **TD target** (Q_target) and the current **Q value** (estimation of Q).

- But we don't have any idea of the real TD target. We need to **estimate** it. Using the **Bellman** equation, we saw that the TD **target** is just the **reward** of taking that action at that state plus the discounted highest Q **value** for the next state.

- However, the problem is that we are using the **same parameters (weights)** for estimating the **target** and the Q **value**. As a consequence, there is a big **correlation** between the TD target and the parameters (w) we are changing.

- Therefore, it means that at every step of training, our Q **values** shift but also the **target** value shifts. So, we're getting closer to our target but the target is also moving. It's like **chasing a moving target!** This lead to **oscillations** in training.

$$\Delta w = \alpha[(R + \gamma\, max_a\, \hat{Q}(s', a, w)) - \hat{Q}(s, a, w)]\, \nabla_w \hat{Q}(s, a, w)$$

Change in weights

learning rate

Maximum possible Qvalue for the next_state (= Q_target)

Current predicted Q-val

TD Error

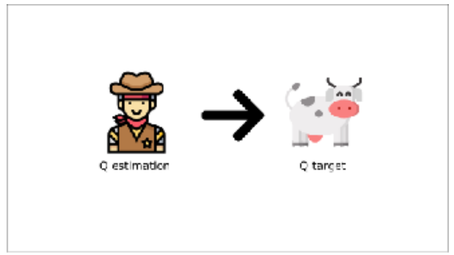
Gradient of our current predicted Q-value

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

Q target

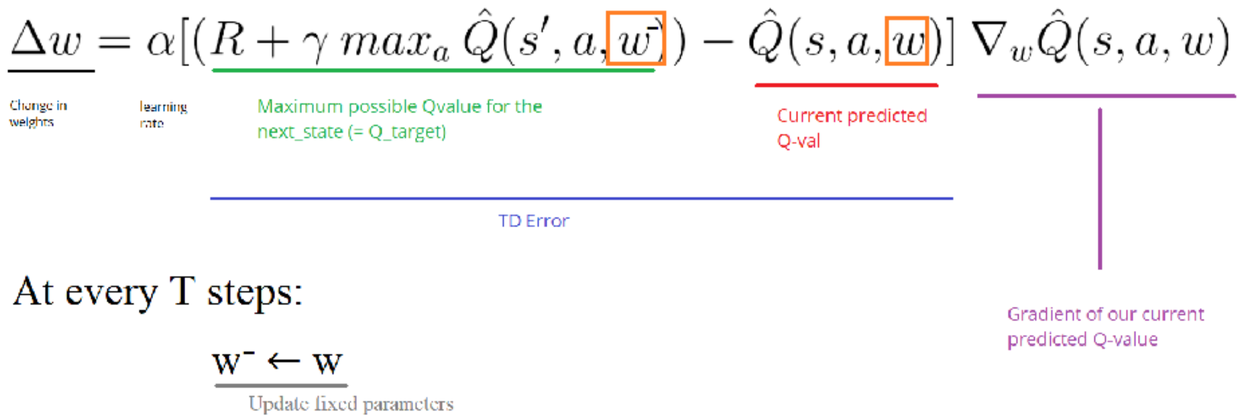
Reward of taking that action at that state

Discounted max q value among all possibles actions from next state.



Improvements: Fixed Q-targets (2)

- **Solution**: we can use the idea of a **fixed Q-targets** introduced by DeepMind:
  - Using a separate network with fixed parameters (let's call it w-) for estimating the TD target.
  - At every Tau step, we copy the parameters from our DQN network to update the target network.
- Thanks to this procedure, we'll have more **stable** learning because the target function stays fixed for a while.



Improvements: Double DQNs

- Double DQNs, or double Learning was introduced to handles the problem of **overestimating** the Q-values

- To understand this problem, remember how we calculate the TD Target. By calculating the TD target, we face a simple **problem**: how are we sure that the **best action** for the next state is the action with the **highest Q-value**?

- We know that the accuracy of q values depends on what action we tried and what neighboring states we explored. As a consequence, at the **beginning** of the training we don't have enough information about the best action to take. Therefore, taking the maximum q value (which is noisy) as the best action to take can lead to false positives. If non-optimal actions are regularly given a higher Q value than the optimal best action, the learning will be complicated.

$$\underbrace{Q(s,a)}_{\text{Q target}} = \underbrace{r(s,a)}_{\text{Reward of taking that action at that state}} + \underbrace{\gamma max_a Q(s',a)}_{\text{Discounted max q value among all possibles actions from next state.}}$$

$$\underbrace{Q(s,a)}_{\text{TD target}} = r(s,a) + \gamma Q(s', \underbrace{argmax_a Q(s',a)}_{\text{DQN Network choose action for next state}})$$

Target network calculates the Q value of taking that action at that state

Improvements: Double DQNs (2)

$$\underbrace{Q(s,a)}_{\text{Q target}} = \underbrace{r(s,a)}_{\text{Reward of taking that action at that state}} + \underbrace{\gamma max_a Q(s',a)}_{\text{Discounted max q value among all possibles actions from next state.}}$$

- The **solution** is: when we compute the Q target, we use **two networks** to decouple the **action selection** from the **target** Q value generation. We:
  - use our **DQN network** to select what is the best **action** to take for the next state (the action with the highest Q value).
  - use our **target network** to calculate the target **Q value** of taking that action at the next state.

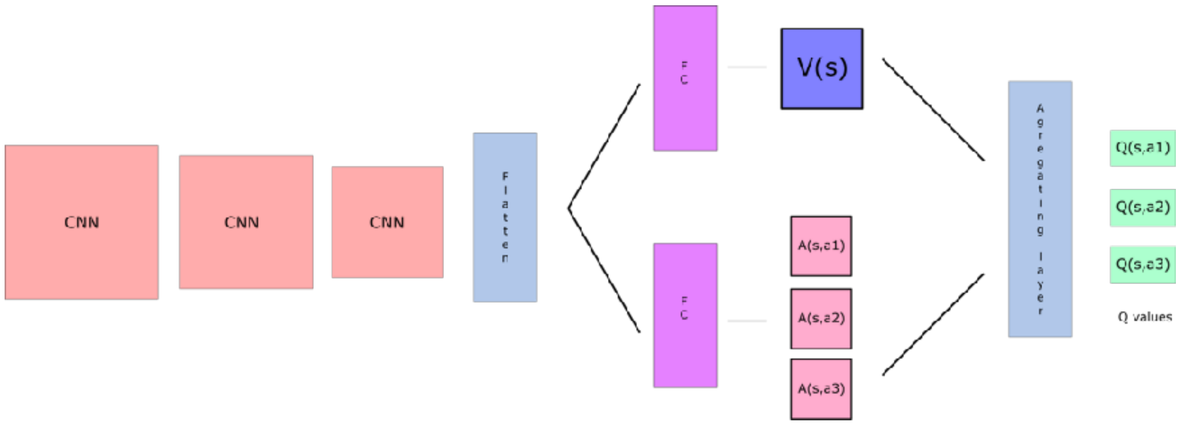
$$\underbrace{Q(s,a)}_{\text{TD target}} = r(s,a) + \gamma Q(s', \underbrace{argmax_a Q(s',a)}_{\text{DQN Network choose action for next state}})$$

Target network calculates the Q value of taking that action at that state

Improvements: Dueling DQN (DDQN)

- Remember that **Q-values** correspond to how **good** it is to be in a given **state** s and take an **action** a at that state Q(s,a).
- So we can **decompose** Q(s,a) as the sum of:
  - V(s): the **value** of being at that state
  - A(s,a): the **advantage** of taking that action at that state (how much **better** is to take this action versus all other possible actions at that state).
- With DDQN, we want to **separate** the estimator of these two elements, using two new streams:
  - one that estimates the state value V(s)
  - one that estimates the advantage for each action A(s,a)
- And then we combine these two streams through a special **aggregation** layer to get an estimate of Q(s,a).

$$Q(s,a) = A(s,a) + V(s)$$

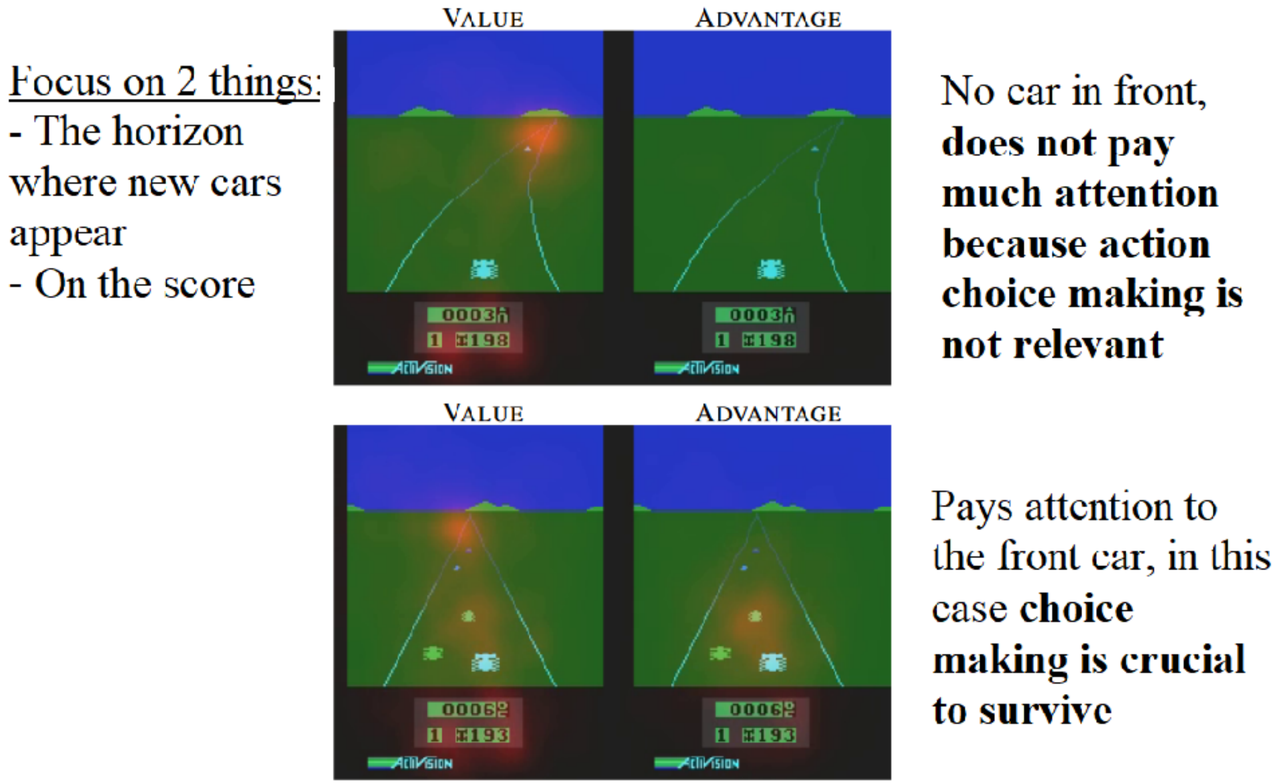


Improvements: Dueling DQN (DDQN) (2)

- But **why** do we need to calculate these two elements **separately** if then we **combine** them?
  - By decoupling the estimation, **intuitively** our DDQN can learn which states are (or are not) **valuable without** having to learn the effect of each **action** at each state (since it's also calculating V(s)).
  - With our **normal DQN**, we need to calculate the value of **each** action at that state. But what's the point if the value of the state is **bad**? What's the point to calculate all actions at one state when all these actions lead to **death**?
  - As a consequence, by decoupling we're able to **calculate V(s)**. This is particularly useful for states where their **actions do not affect the environment** in a relevant way. In this case, it's unnecessary to calculate the value of each action. For instance, moving right or left only matters if there is a risk of **collision**. And, in most states, the choice of the action has no effect on what happens.

Improvements: Dueling DQN (DDQN) (3)

- It will be clearer if we take a look at the **example** to the right.
  - We see that the value **network streams** pays attention (the orange blur) to the **road**, and in particular to the **horizon** where the cars are spawned. It also pays attention to the **score**.
  - On the other hand, the **advantage stream** in the first frame on the right does not pay much attention to the road, because there are no cars in front (so the action choice is practically irrelevant). But, in the second frame it pays attention, as there is a car immediately in front of it, and making a choice of action is crucial and very relevant.



Improvements: Dueling DQN (DDQN) (4)

- Concerning the **aggregation layer**, we want to generate the q values for each action at that state. We might be tempted to **combine** the streams as shown below. But if we do that, we'll fall into the issue of **identifiability**, that is—given Q(s,a) we're unable to find A(s,a) and V(s).

- And not being able to find V(s) and A(s,a) given Q(s,a) will be a problem for our **back-propagation.** To avoid this problem, we can **force our advantage function estimator to have 0 advantage at the chosen action.** To do that, we **subtract the average advantage** of all actions possible of the state.

- Therefore, this architecture helps us **accelerate the training**. We can calculate the value of a state without calculating the Q(s,a) for each action at that state. And it can help us find much more reliable Q values for each action by decoupling the estimation between two streams.

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha)$$

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \frac{1}{\mathcal{A}}\sum_{a'} A(s,a';\theta,\alpha)\right)$$

Common network parameters

Value stream parameters

Advantage stream parameters

Average advantage

Improvements: Prioritized Experience Replay

- Prioritized Experience Replay (PER) was introduced in 2015. The **idea** is that some experiences may be more **important** than others for our training, but might **occur less frequently**.

- Because we **sample** the batch **uniformly** (selecting the experiences randomly) these **rich** experiences that occur rarely have practically **no chance** to be selected.

- That’s why, with PER, we try to change the sampling distribution by using a criterion to define the priority of each tuple of experience.

- We want to take in priority experiences where there are a **big difference** between our prediction and the TD target, since it means that we have a lot to **learn** about it.

Improvements: Prioritized Experience Replay (2)

- We use the absolute value of the **magnitude of our TD error**.
- And we put that **priority** in the experience of each **replay buffer**.
- But we can't just do **greedy** prioritization, because it will lead to always training the **same** experiences (that have big priority), and thus over-fitting. So we introduce **stochastic prioritization**, which generates the **probability** of being chosen for a replay.
- As consequence, during each time step, we will get a batch of samples with this probability distribution and train our network on it.

$$p_t = |\delta_t| + e$$

Magnitude of our TD error

Constant assures that no experience has 0 probability to be taken.

$$P(i) = \frac{p_i^a}{\sum_k p_k^a}$$

Priority value

Normalized by all priority values in Replay Buffer

Hyperparameter used to **reintroduce some randomness in the experience selection for the replay buffer**

If **a = 0** pure uniform randomness

If **a = 1** only select the experiences with the highest priorities

Improvements: Prioritized Experience Replay (3)

- But, we still have a problem here. Remember that with normal Experience Replay, we use a stochastic update rule. As a consequence, the **way we sample the experiences must match the underlying distribution they came from.**

- When we do have **normal** experiences, we select our experiences in a **normal** distribution—simply put, we select our experiences **randomly**. There is no bias, because each experience has the same chance to be taken, so we can update our weights normally.

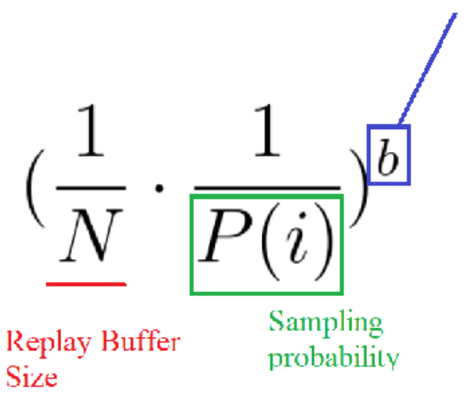
- But, because we use priority sampling, purely random sampling is abandoned. As a consequence, we introduce bias toward high-priority samples (more chances to be selected).

- And, if we update our weights normally, we take have a risk of over-fitting. Samples that have high priority are likely to be used for training many times in comparison with low priority experiences (= bias). As a consequence, we'll update our weights with only a small portion of experiences that we consider to be really interesting.

- To correct this bias, we use importance sampling weights (IS) that will adjust the updating by reducing the weights of the often seen samples.

- The weights corresponding to high-priority samples have very little adjustment (because the network will see these experiences many times), whereas those corresponding to low-priority samples will have a full update.

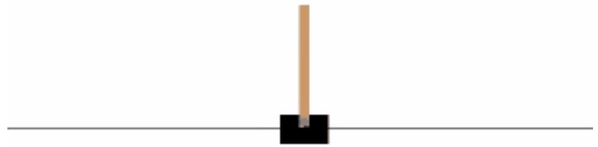
- The role of b is to control how much these importance sampling weights affect learning. In practice, the b parameter is annealed up to 1 over the duration of training, because these weights are more important in the end of learning when our q values begin to converge. The unbiased nature of updates is most important near convergence.



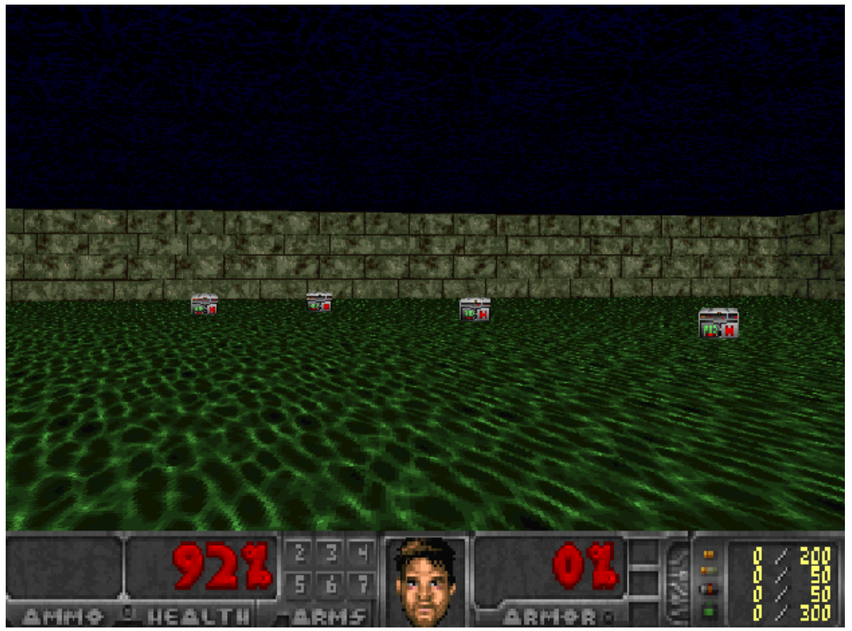
Policy-based RL

Policy Gradient (PG)

- **Previously**: value-based RL algorithms: To choose which action to take given a state, we take the action with the **highest Q-value** (maximum expected future reward). As a consequence, in value-based learning, a **policy** exists only because of these action-value estimates.



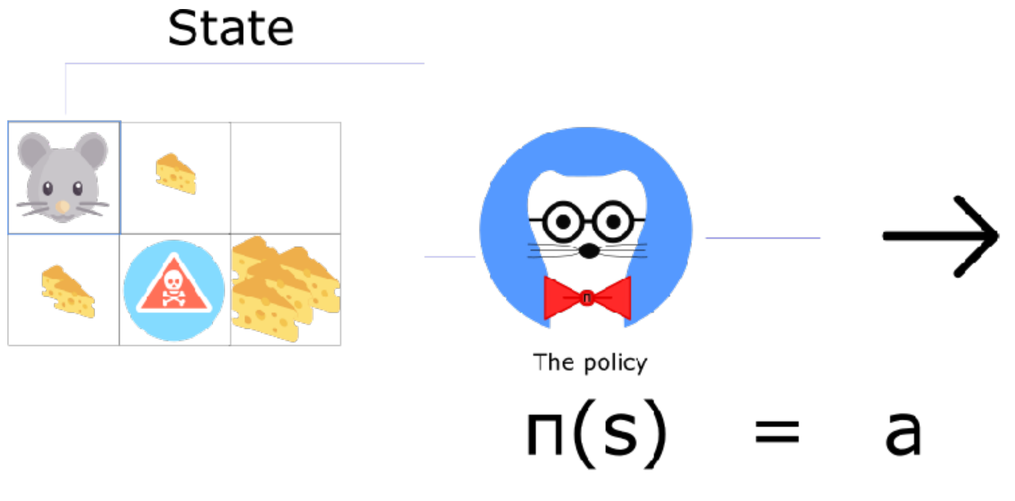
- **Now**: policy-based RL technique called **Policy Gradients:** In policy-based methods, instead of learning a value function that tells us what is the expected sum of rewards given a state and an action, we learn **directly** the policy function that **maps state to action** (select actions without using a value function). It means that we directly try to optimize our policy function π without worrying about a value function.



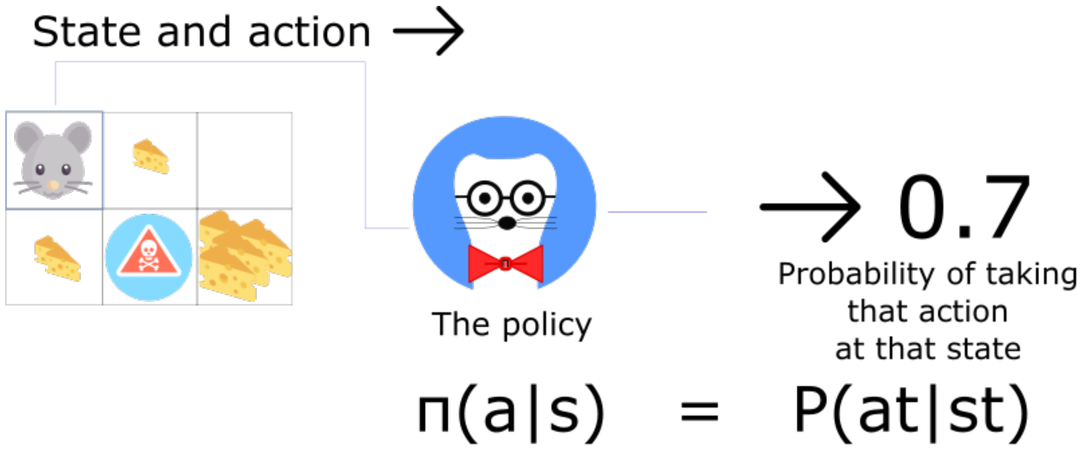
- Example agents: Cartpole and Doom

PG: Types

- A policy can be either **deterministic or stochastic**.
- **Deterministic** policy: maps state to action, give the function a state and it returns an action to take.
  - used in deterministic environments: actions taken determine the outcome, no uncertainty, e.g. chess play.
- **Stochastic** policy: outputs a **probability distribution** over actions.
  - Instead of being sure of taking action a (for instance left), there is a probability we'll take a different one (in this case 30% that we take south).
  - The stochastic policy is used when the environment is **uncertain**. We call this process a Partially Observable Markov Decision Process (**POMDP**).
  - Most of the time we'll use this second type of policy.



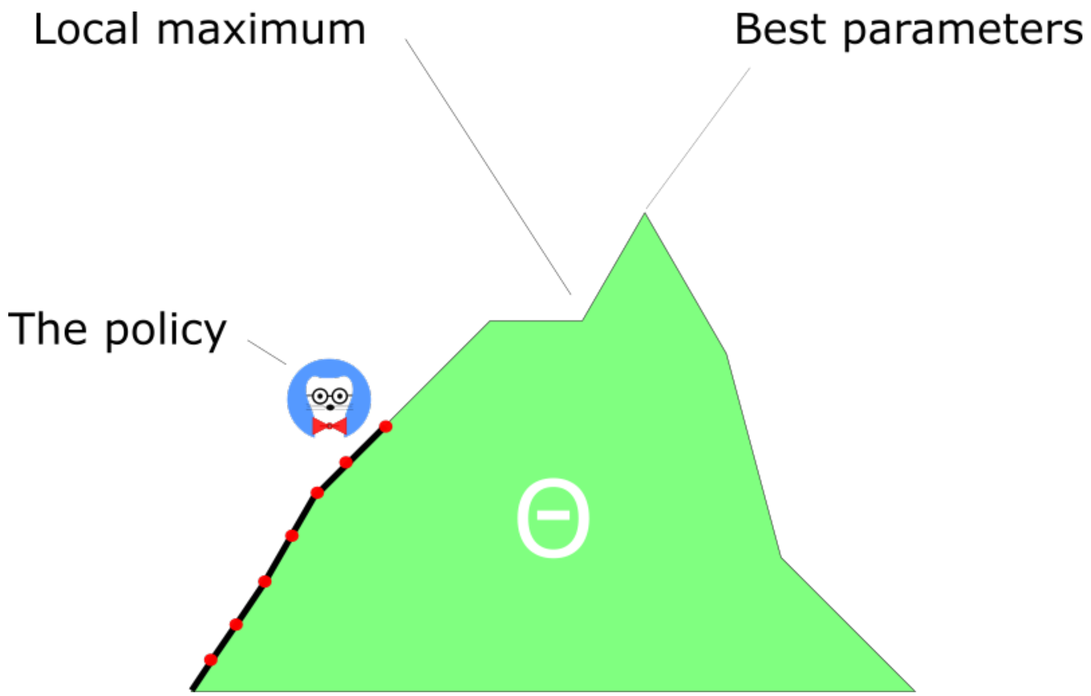
Deterministic policy



Stochastic policy

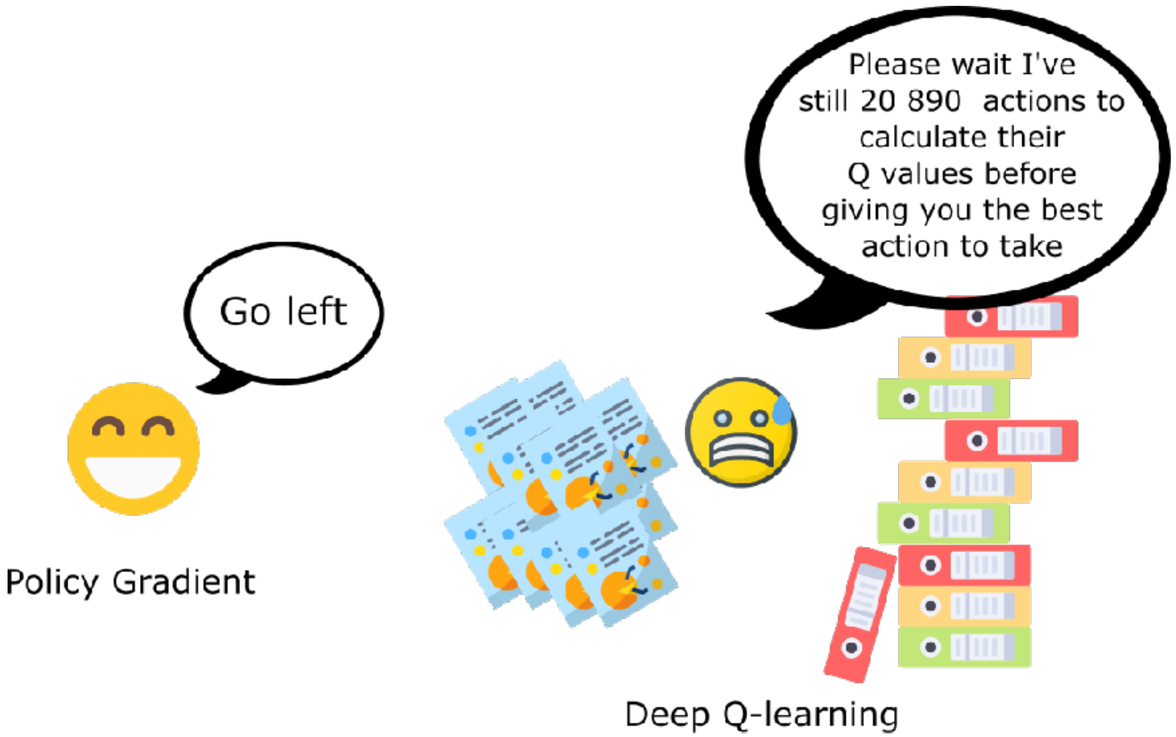
PG: Advantages

- **Convergence**:
  - Policy-based methods have **better convergence** properties
  - Value-based methods can have big **oscillation** while training. This is because the choice of **action** may change dramatically for an arbitrarily small change in the estimated **action values**.
  - Policy gradient follow the **gradient** to find the best parameters. We see a **smooth** update of our policy at each step.
    - Because we follow the gradient to find the best parameters, we're guaranteed to converge on a **local** maximum (worst case) or global maximum (best case).



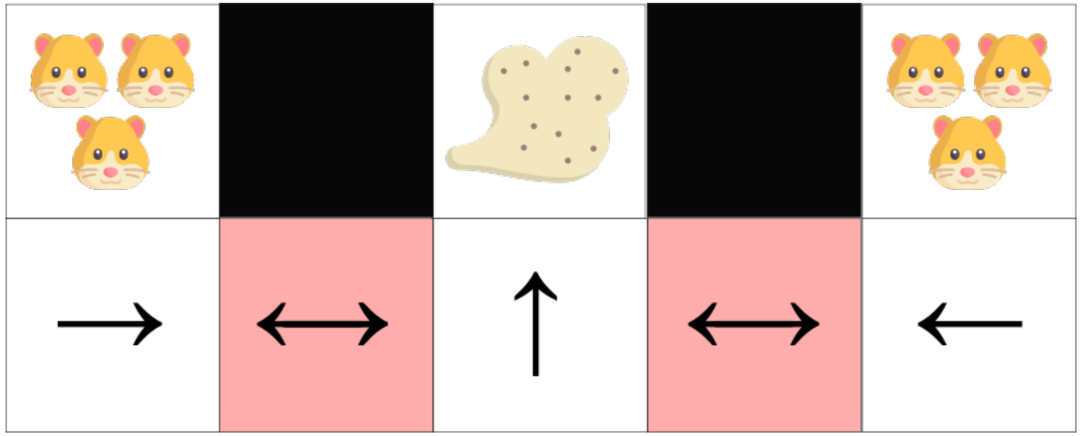
PG: Advantages (2)

- More **effective** in **high** dimensional action spaces**,** or when using **continuous** actions:
  - DQL: the predictions assign a **score** (maximum expected future reward) for each possible action, at each time step, given the current state. What if we have an **infinite** possibility of actions, e.g. a self driving car and steering wheel predictions.
  - On the other hand, in policy-based methods, you just **adjust** the parameters directly: thanks to that you'll start to understand what the maximum will be, rather than computing (estimating) the maximum directly at every step.



PG: Advantages (3)

- Can learn **stochastic** policies:
  - Policy gradient can **learn** a stochastic policy, while value functions can't and this has two consequences:
    - We don't need to implement an exploration/exploitation **trade-off** because it outputs a probability **distribution** over actions.
    - We also get rid of the problem of **perceptual aliasing**: we have two states that seem to be (or actually are) the same, but need different actions.
- **Disadvantages**:
  - A lot of the time, they converge on a **local** maximum rather than on the global optimum.
  - Instead of Deep Q-Learning, which always tries to reach the maximum, policy gradients converge **slower**, step by step. They can take longer to train.



π = (wall UP and DOWN | Go LEFT) = 0.5
π = (wall UP and DOWN | Go RIGHT) = 0.5

PG: Policy Search

- We have our policy π that has **parameters** θ. This π outputs a probability **distribution** of actions.

- How do we know if our **policy is good?** We must find the best parameters (θ) to maximize a **score function**, J(θ).

- There are two steps:

  - Measure the **quality** of a π (policy) with a **policy score function** J(θ)

  - Use **policy gradient ascent** to find the best parameter θ that improves our π.

- The main idea here is that J(θ) will tell us how **good** our π is. Policy gradient ascent will help us to find the **best policy parameters** to maximize the sample of good actions.

$$\pi_\theta(a|s) = P[a|s]$$

$$J(\theta) = E_{\pi\theta}[\sum \gamma r]$$

PG: First Step: the Policy **Score** function J(θ)

- To measure how **good** our policy is, we use a function called the **objective** function (or Policy Score Function) that calculates the expected **reward** of policy.

- **Three** methods work equally well for optimizing policies. The choice depends only on the **environment** and the objectives you have.

- 1: **Episodic** environment

  - Calculate the mean of the return from the first time step (G1). This is the cumulative discounted reward for the entire episode.

  - The idea is simple. If I always start in some state s1, what's the total reward I'll get from that start state until the end?

  - We want to find the policy that maximizes G1, because it will be the optimal policy. This is due to the reward hypothesis.

$$J_1(\theta) = E_\pi[\underbrace{G_1 = R_1 + \gamma R_2 + \gamma^2 R_3 + ...}_{\text{Cumulative discounted rewards starting at start state}}] = \underbrace{E_\pi(V(s_1))}_{\text{Value of state 1}}$$

Equivalent

PG: First Step: the Policy **Score** function J(θ)

- 2: **Continuous** environment:
  - We can use the **average** value, because we can't rely on a specific start state.
  - Each state value is now weighted (because some happen more than others) by the probability of the occurrence of the respected state.
- 3: we can use the **average** reward per time step.
  - The idea here is that we want to get the most reward per time step.

$$J_{avgv}(\theta) = E_\pi(V(s)) = \sum d(s)V(s)$$

$$where\ d(s) = \frac{N(s)}{\sum_{s'} N(s')}$$

Number of occurrences of the state

Total nb occurences of all states

$$J_{avR}(\theta) = E_\pi(r) = \sum_s d(s) \sum_a \pi\theta(s,a)\ R_s^a$$

Probability that I'm in state s

Probability that I take this action a from that state under this policy

Immediate reward that I'll get

PG: Second step: Policy gradient ascent

- We have a Policy **score** function that tells us how **good** our policy is. Now, we want to find a **parameters** θ that maximizes this score function. Maximizing the score function means finding the **optimal** policy.

- To maximize the score function J(θ), we need to do gradient ascent on policy parameters. Gradient ascent is the inverse of gradient descent. Remember that gradient always points to the steepest change. In gradient descent, we take the direction of the steepest decrease in the function. In gradient ascent we take the direction of the steepest increase of the function.

- Why gradient ascent and not gradient descent? Because we use gradient descent when we have an error function that we want to minimize. But, the score function is not an error function! It's a score function, and because we want to maximize the score, we need gradient ascent. The idea is to find the gradient to the current policy π that updates the parameters in the direction of the greatest increase, and iterate.

$$Policy: \ \pi_\theta$$
$$Objective\ function\ : J(\theta)$$
$$Gradient: \ \nabla_\theta J(\theta)$$
$$Update: \theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

PG: Second step: Policy gradient ascent

- We want to find the best parameters θ*, that maximize the score.
- Our score function can be defined as J(θ)
  - Which is the total summation of expected reward given policy.
- Our score function J(θ) can be also defined as J1(θ)
  - We wrote the function in this way to show the problem we face here.
  - We know that policy parameters change how actions are chosen, and as a consequence, what rewards we get and which states we will see and how often.
  - So, it can be challenging to find the changes of policy in a way that ensures improvement. This is because the performance depends on action selections and the distribution of states in which those selections are made.
  - Both of these are affected by policy parameters. The effect of policy parameters on the actions is simple to find, but how do we find the effect of policy on the state distribution? The function of the environment is unknown.
  - As a consequence, we face a problem: how do we estimate the ∇ (gradient) with respect to policy θ, when the gradient depends on the unknown effect of policy changes on the state distribution?

$$\theta^* = \underset{\theta}{argmax}\, \underbrace{E_{\pi\theta}[\sum_t R(s_t, a_t)]}_{J(\theta)}$$

$$J(\theta) = \underbrace{E_\pi}_{\text{Expected given policy}}[\underbrace{R(\tau)}_{\text{Expected future reward}}]$$

s0, a0, r0, s1, a1, r1...

$$J_1(\theta) = V_{\pi\theta}(s_1) = E_{\pi\theta}[v_1] = \underbrace{\sum_{s\in S} d(s)}_{\text{State distribution}} \underbrace{\sum_{a\in A} \pi_\theta(s,a) R_s^a}_{\text{Action distribution}}$$

NB: It's ok to not understand all the math on this slide

PG: Second step: Policy gradient ascent

- The solution will be to use the Policy Gradient Theorem. This provides an analytic expression for the gradient ∇ of J(θ) (performance) with respect to policy θ that does not involve the differentiation of the state distribution.

- Remember, we're in a situation of stochastic policy. This means that our policy outputs a probability distribution π(τ ; θ). It outputs the probability of taking these series of steps (s0, a0, r0…), given our current parameters θ.

- But, differentiating a probability function is hard, unless we can transform it into a logarithm. This makes it much simpler to differentiate.

- Here we'll use the **likelihood ratio trick** that replaces the resulting fraction into log probability.

- Now let's convert the summation back to an expectation.

- As you can see, we only need to compute the derivative of the log policy function.

$$J(\theta) = E_\pi[R(\tau)]$$

s0, a0, r0, s1, a1, r1...

Expected given policy

Expected future reward

$$\nabla_\theta J(\theta) = \nabla_\theta \sum_\tau \pi(\tau;\theta) R(\tau)$$

$$= \sum_\tau \nabla_\theta \pi(\tau;\theta) R(\tau)$$

$$\pi(\tau;\theta)\frac{\nabla_\theta \pi(\tau;\theta)}{\pi(\tau;\theta)} \qquad \nabla log\, x = \frac{\nabla x}{x}$$

Likelihood ratio trick

$$= \sum_\tau \pi(\tau;\theta)\nabla_\theta(log\pi(\tau;\theta))R(\tau)$$

$$\nabla_\theta J(\theta) = E_\pi[\nabla_\theta(log\pi(\tau|\theta))R(\tau)]$$

Policy function

Score function

$$Policy\ gradient: E_\pi[\nabla_\theta(log\pi(s,a,\theta))R(\tau)]$$

Policy function

Score function

$$Update\ rule: \Delta\theta = \alpha * \nabla_\theta(log\pi(s,a,\theta))R(\tau)$$

Change in parameters

Learning rate

PG: Second step: Policy gradient ascent

- We can now conclude about policy gradients.
- This Policy gradient is telling us how we should shift the policy distribution through changing parameters θ if we want to achieve an higher score.
- R(tau) is like a scalar value score:
  - If R(tau) is high, it means that on average we took actions that lead to high rewards. We want to push the probabilities of the actions seen (increase the probability of taking these actions).
  - On the other hand, if R(tau) is low, we want to push down the probabilities of the actions seen.
- This policy gradient causes the parameters to move most in the direction that favors actions that has the highest return.

$$Policy\ gradient: E_\pi[\nabla_\theta(log\pi(s,a,\theta))R(\tau)]$$

Policy function — Score function

$$Update\ rule: \Delta\theta = \alpha * \nabla_\theta(log\pi(s,a,\theta))R(\tau)$$

Change in parameters — Learning rate

PG: Monte Carlo Policy Gradients

- We use the Monte Carlo approach to design the policy gradient algorithm as the tasks can be divided into episodes.

- But we face a problem with this algorithm. Because we only calculate R at the end of the episode, we average all actions. Even if some of the actions taken were very bad, if our score is quite high, we will average all the actions as good.

- So to have a correct policy, we need **a lot** of samples, which results in **slow** learning.

- How can this be **improved**:

  - **Actor Critic** (AC): a hybrid between policy-based and value-based algorithms.

  - **Proximal Policy Optimization (PPO)**: ensures that the deviation from the previous policy stays relatively small.

```
Initialize θ
for each episode τ = S0, A0, R1, S1, …, ST:
    for t <-- 1 to T-1:
        Δθ = α ∇theta(log π(St, At, θ)) Gt
        θ = θ + Δθ

For each episode:
    At each time step within that episode:
         Compute the log probabilities produced by our policy
function. Multiply it by the score function.
         Update the weights
```

Hybrid / Actor-Critic based RL

Advantage Actor Critic (A2C)

- So far we have studied two different RL methods:
  - **Value based methods** (Q-learning, Deep Q-learning): where we learn a **value function** that will map each **state action pair to a value**. Thanks to these methods, we find the **best action** to take for each state - the action with the **biggest value**. This works well when you have a **finite set of actions**.
  - **Policy based methods** (REINFORCE with Policy Gradients): where we **directly** optimize the policy without using a value function. This is useful when the **action space is continuous or stochastic**. The main problem is finding a **good score function** to compute how good a policy is. **We use total rewards of the episode.**
- But both of these methods have big **drawbacks**. We will therefor look at new type of RL method which we can call a “hybrid method”: **Actor Critic**. The method uses two NNs:
  - an **Actor** that controls how our agent behaves (policy-based)
  - a **Critic** that measures how good the action taken is (value-based)
- Mastering this architecture is essential to understanding state of the art algorithms such as Proximal Policy Optimization (aka **PPO**). PPO is based on Advantage Actor Critic (**A2C**).

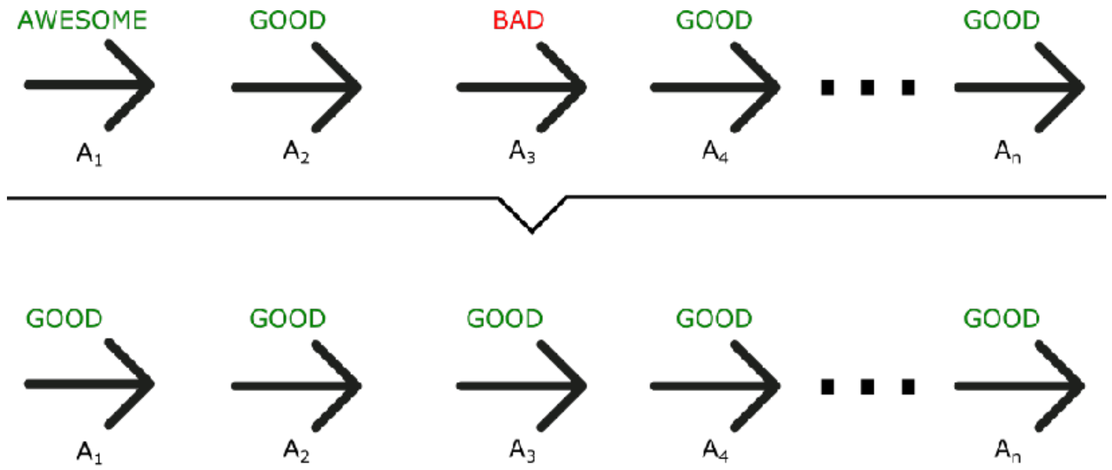
A2C: The problem with Policy Gradients

- The **Policy Gradient** method has a big **problem**. We are in a situation of **Monte Carlo**, waiting until the **end** of the episode to calculate the **reward**. We may conclude that if we have a **high reward** (R(t)), **all actions** that we took were good, even if some were really **bad**.

- As we can see in this example, even if A3 was a bad action (led to negative rewards), all the actions will be averaged as good because the total reward was important.

- As a consequence, to have an optimal policy, we need **a lot** of samples. This produces **slow** learning, because it takes a lot of time to converge.

- What if we can do an update at **each** time step instead?



A2C: The Actor Critic model

- The Actor Critic model is a **better score function**. Instead of waiting until the **end of the episode** as we do in Monte Carlo REINFORCE, we make an **update at each step** (TD Learning).

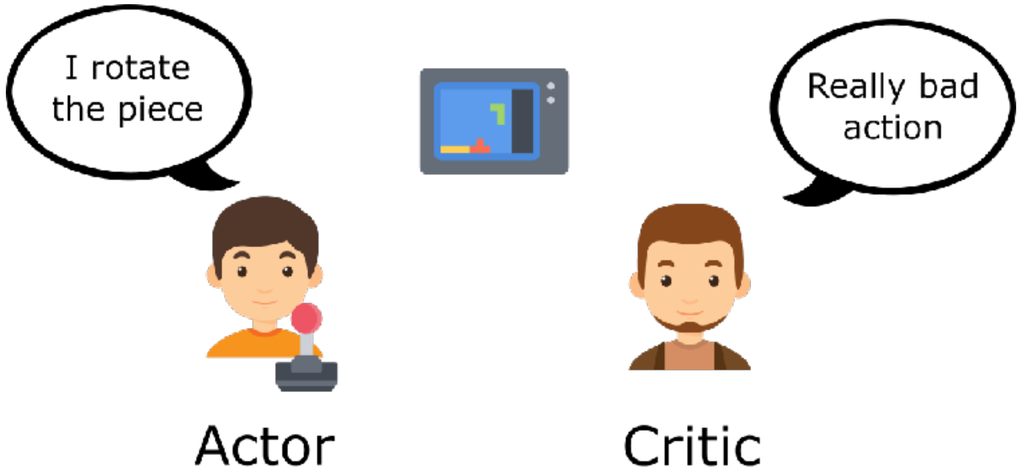
- Because we do an update at each time step, we can't use the **total rewards** R(t). Instead, we need to train a **Critic model** that approximates the **value function** (remember that value function calculates what is the maximum expected future reward given a state and an action). This value function replaces the **reward function** in policy gradient that calculates the rewards only at the end of the episode.

Policy Update: $\Delta\theta = \alpha * \nabla_\theta * (log\ \pi(S_t, A_t, \theta)) * \cancel{R(t)}$

New update: $\Delta\theta = \alpha * \nabla_\theta * (log\ \pi(S_t, A_t, \theta)) * \boxed{Q(S_t, A_t)}$

A2C: How Actor Critic works

- Imagine **you play a video game** with a **friend** that provides you some **feedback**. **You're** the **Actor** and your **friend** is the **Critic**.
  - At the **beginning**, you don't know how to play, so you **try** some action randomly. The Critic observes your action and provides **feedback**.
  - **Learning** from this **feedback**, you'll **update** your policy and become better at playing the game.
  - On the other hand, your **friend (Critic)** will also **update** his way to provide feedback so he can be better next time.
- As we can see, the **idea** of Actor Critic is to have **two NN**. We **estimate** both and both run in parallel.
- Because we have **two models** (Actor and Critic) that must be **trained**, it means that we have **two set of weights** (θ for our action and w for our Critic) that must be **optimized separately**



$$\pi(s, a, \theta) \qquad \hat{q}(s, a, w)$$

ACTOR : A policy function, controls how our agent acts

CRITIC : A value function, measures how good these actions are

Policy Update: $\Delta\theta = \alpha \nabla_\theta(log\ \pi_\theta(s,a))\hat{q}_w(s,a)$

q learning function approximation (estimate action value)

Value update: $\Delta w = \beta\ (R(s,a) + \gamma\hat{q}_w(s_{t+1}, a_{t+1}) - \hat{q}_w(s_t, a_t))\nabla_w\hat{q}_w(s_t, a_t)$

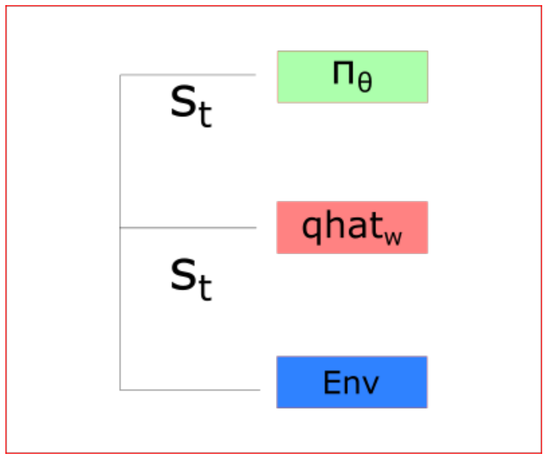
Policy and value have different learning rates

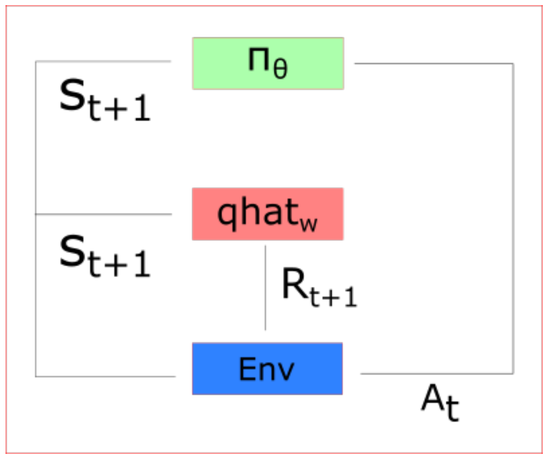
TD error

Gradient of our value function

A2C: The Actor Critic Process

- At **each time-step t**, we take the current **state** (St) from the environment and pass it as an input through our **Actor** and our **Critic**.
- Our **Policy** takes the **state**, outputs an **action** (At), and were receives a **new state** (St+1) and a **reward** (Rt+1) from the environment.
- Based on that:
  - the **Critic** computes the value of taking that action at that state
  - the **Actor** updates its policy parameters (weights) using this q value
- Thanks to its updated parameters, the Actor produces the next action to take at At+1 given the new state St+1. The **Critic** then updates its value parameters.





$$\Delta\theta = \alpha \nabla_\theta(log\ \pi_\theta(s,a))\hat{q}_w(s,a)$$

$$\Delta w = \beta\left(R(s,a) + \gamma\hat{q}_w(s_{t+1},a_{t+1}) - \hat{q}_w(s_t,a_t)\right)\nabla_w\hat{q}_w(s_t,a_t)$$

A2C: the Advantage function

- Value-based methods have high **variability**. To reduce this problem we can use the **advantage function** instead of the **value function**.

- The **advantage** function will will tell us the **improvement** compared to the average. In other words, this function calculates the **extra reward** we get if we take this action. The extra reward is that **beyond the expected** value of that state.

  - If A(s,a) > 0: (our action does better than the average value of that state) our gradient is pushed in that direction.

  - If A(s,a) < 0: (our action does worse than the average value of that state) our gradient is pushed in the opposite direction.

- The problem of implementing this advantage function is that is requires two value functions—Q(s,a) and V(s). Fortunately, **we can use the TD error as a good estimator of the advantage function.**

$$A(s,a) = Q(s,a) - V(s)$$

q value for action a in state s

average value of that state

$$A(s,a) = Q(s,a) - V(s)$$

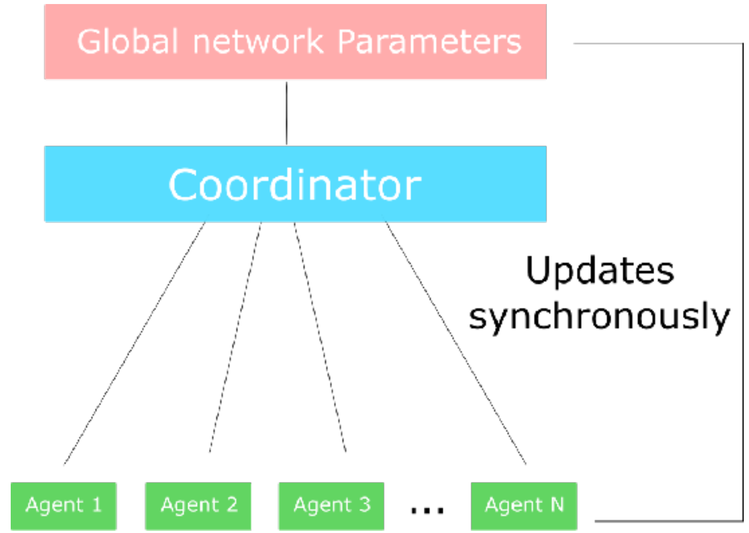
$$r + \gamma V(s')$$

$$A(s,a) = r + \gamma V(s') - V(s)$$

TD Error

A2C: Synchronous vs. Asynchronous

- We have two different strategies to implement an Actor Critic agent:
  - A2C: (Synchronous) Advantage Actor Critic
  - A3C: Asynchronous Advantage Actor Critic
- In A3C, we don't use experience replay as this requires lot of memory. Instead, we asynchronously execute different agents in parallel on multiple instances of the environment. Each worker (copy of the network) will update the global network asynchronously.
  - On the other hand, the only difference in A2C is that we synchronously update the global network. We wait until all workers have finished their training and calculated their gradients to average them, to update our global network.
- Because of the asynchronous nature of A3C, some workers (copies of the Agent) will be playing with older version of the parameters. Thus the aggregating update will not be optimal.
  - That's why A2C waits for each actor to finish their segment of experience before updating the global parameters. Then, we restart a new segment of experience with all parallel actors having the same new parameters.



A2C: in practice

- The synchronous nature of A2C means we don’t need different versions (different workers). Each worker in A2C will have the same set of weights since A2C updates all their workers at the same time. In fact, we create multiple versions of the environments (let say eight) and then execute them in parallel.

- The process will be the following:

  - Creates a vector of n environments using the multiprocessing library

  - Creates a runner object that handles the different environments, executing in parallel.

  - We two versions of the network:

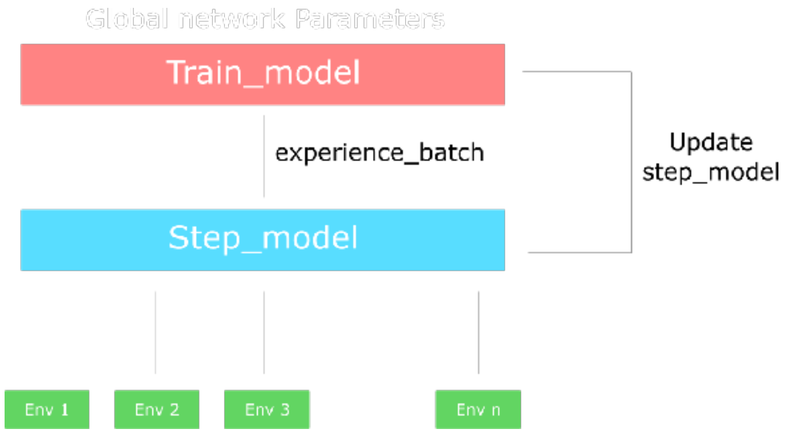
    - step_model: that generates experiences from environments

    - train_model: that trains the experiences

  - When the runner takes a step (single step model), this performs a step for each of the n environments. This outputs a batch of experiences.

  - Then we compute the gradient all at once using train_model and our batch of experience (Remember that computing the gradient all at once is the same thing as collecting data, calculating the gradient for each worker, and then averaging).

  - Finally, we update the step model with the new weights.



Proximal Policy Optimization (PPO)

- **Breakthrough**: OpenAI beat some of the best **Dota2** players of the world with OpenAI five, a team of 5 agents.

    - This was made possible thanks to a strong **hardware** architecture and by using the state of the art's algorithm: **s** aka Proximal Policy Optimization.

- The central idea of Proximal Policy Optimization is to **avoid having too large policy update**. To do that, we use a **ratio** that will tells us the difference between our new and old policy and clip this ratio from 0.8 to 1.2. Doing that will ensure that our policy update will not be too large.

- Moreover, PPO introduced another innovation which is training the agent by running K epochs of gradient descent over the sampling mini batches.

PPO: problem with Policy Gradient Objective function

- Remember the **Policy Objective Function or Policy Loss** used in a Policy Gradient agent. The idea was by taking a gradient ascent step on this function (which is equivalent of taking gradient descent of the negative of this function) we will push our agent to take actions that lead to higher rewards and avoid bad actions.

- However, the problem comes from the step size:

  - Too **small**, the training process is too **slow**

  - Too **high**, there is too much **variability** in the training.

- That's where PPO is useful, the idea is that PPO improves the **stability** of the **Actor** training by **limiting** the policy **update** at each training step.

- To be able to do that PPO introduced a new objective function called "**Clipped surrogate objective function**" that will constraint the policy change in a small range using a clip.

$$L^{PG}(\theta) = E_t[\log \pi_\theta(a_t|s_t) * A_t]$$

Policy Loss

Expected

log probability of taking that action at that state

Advantage if A>0, this action is better than the other action possible at that state

Next time..

Traditional ML approach to CV

- **Feature engineering:**
  - Harris, SIFT, RANSAC (points)
  - Viola Jones, HOG, DPM (objects)
- Classification (orientation / apply):
  - KNN, SVM (supervised)
  - k Means, Mean shift, PCA (unsupervised / clustering)