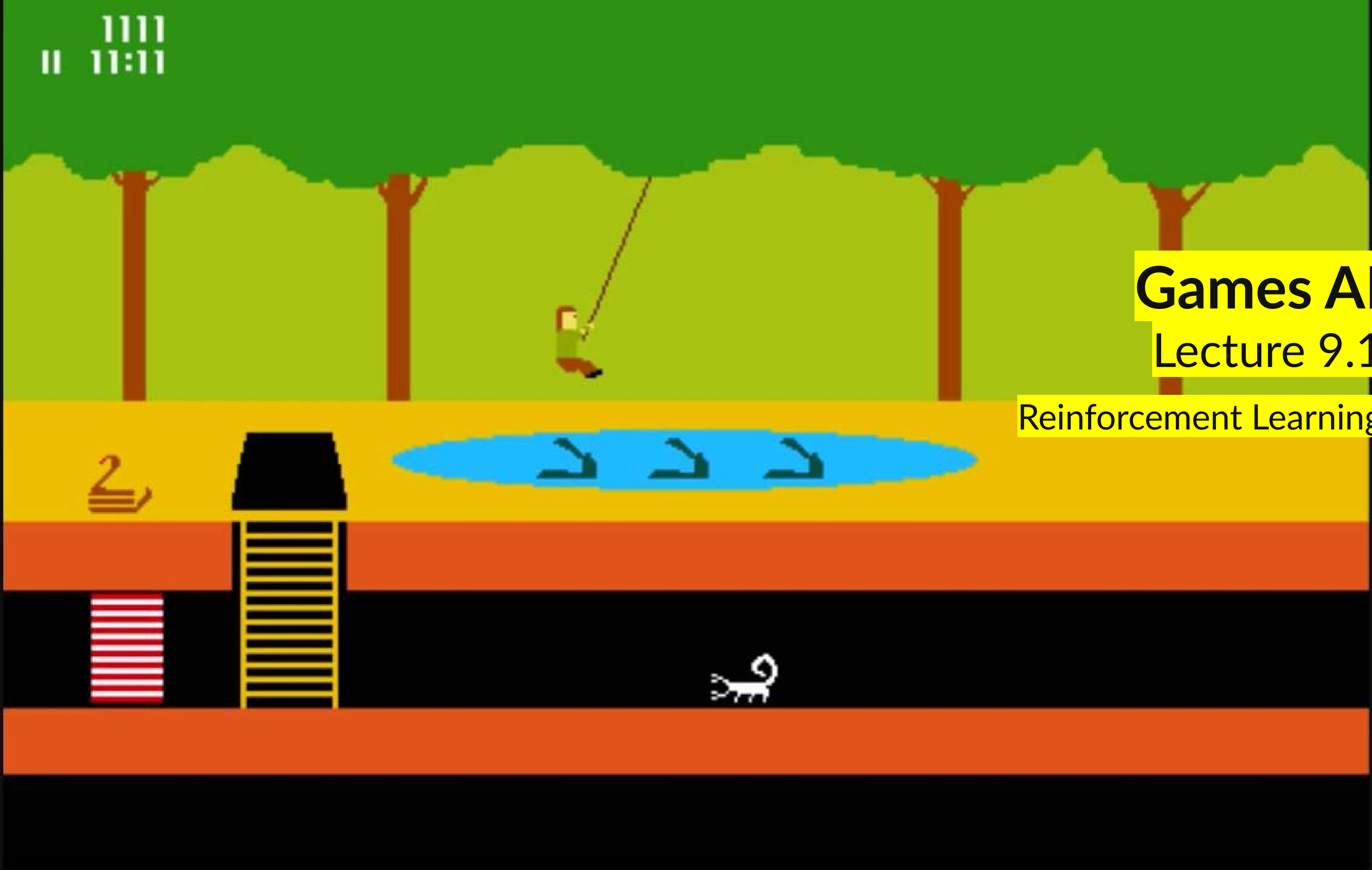


Games AI

Lecture 9.1

Reinforcement Learning

- Machine Learning (ML)
  - ML approaches progressively improve the performance of a specific task with data
  - For example, neural networks learn a function from labeled data in a training data set

- ML comes in two kinds
  - **Supervised Learning**
    - When training, each decision can be compared to a known correct value
    - Neural networks (except autoencoders)
  - **Unsupervised Learning**
    - Correct value not known
    - Agent figures out the best thing to do by itself

- Reinforcement Learning (RL)
  - RL agents learn how to act in an environment in order to maximise their cumulative reward

- RL agents observe their environment and receive rewards. This is the data that helps them to improve at their task.
  - This is why they are **learning** agents
- We do not know what actions an agent should perform to maximise their reward so **RL is unsupervised learning**
  - We only know when to give them rewards

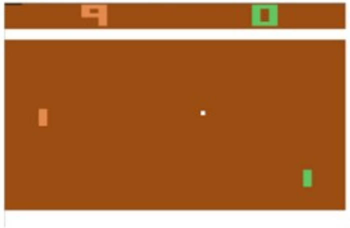
- When to use RL
  - Learning to make a **sequence of decisions** under uncertainty,
  - No or little pre-existing knowledge of environment
  - Limited knowledge, limited feedback
  - You cannot tell how good an action is immediately, but you can give a (possibly delayed) reward, e.g.
    - completing a maze
    - return on financial investments

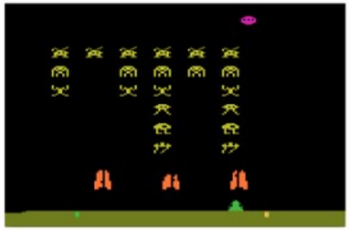
Overview


ALPHAGO
02:00:00
LEE SEDOL
01:59:41
AlphaGo
Lee Sedol


PRODUCTION
DeepMind
STARCRAFT
REPLAY
AlphaStar
DeepMind


OpenAI
Killing Spree 3x Kill Streak ended

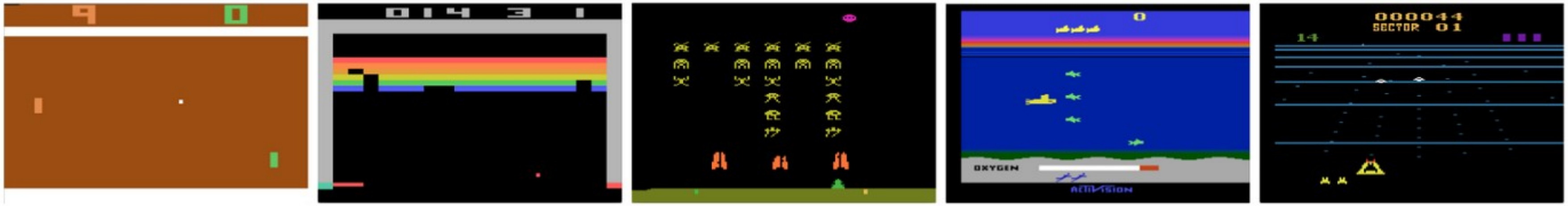





OXYGEN
ACTIVISION

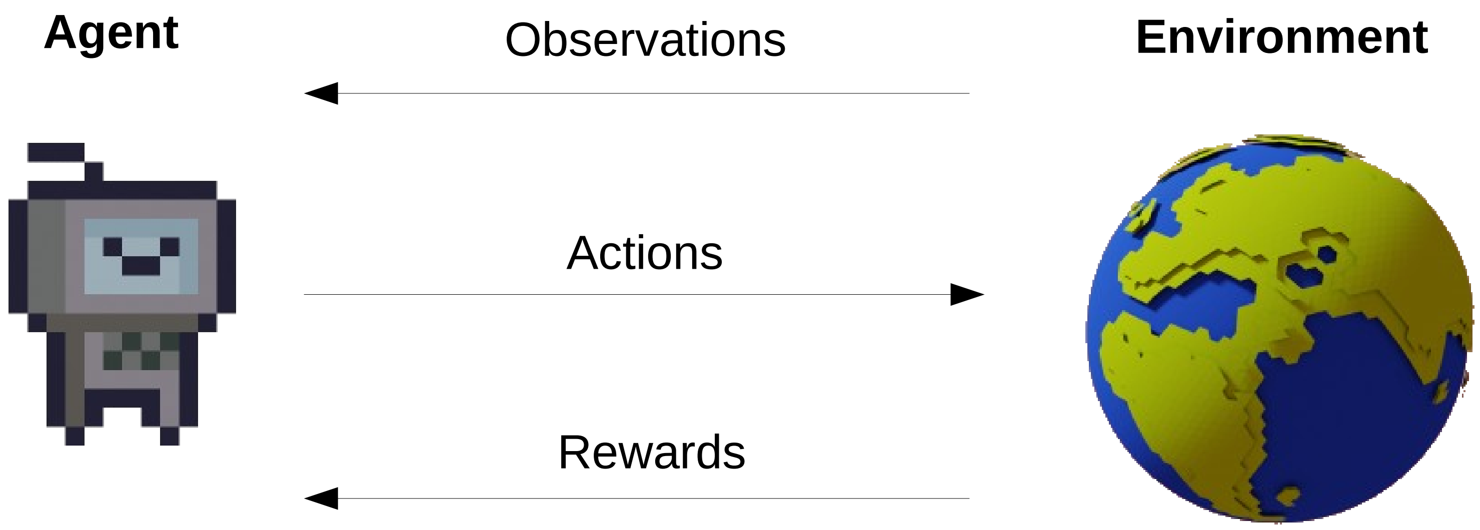

000044
SECTOR 01

- Examples
  - Learn to fly a helicopter (https://youtu.be/VCdxqn0fcnE)
  - Play Atari 2600 games (https://youtu.be/Q70ulPJW3Gk)
  - Trade shares
  - Control a power station



Structure of a Reinforcement Learning problem

Structure of an RL problem



- Time
  - Agents interact with environment over a sequence of time steps
    - t_1, t_2, t_3, t_4, ...
  - For example,
    - A game of chess takes multiple turns
    - An autonomous vehicle driving to a destination takes time
    - An algorithm trades stocks every millisecond

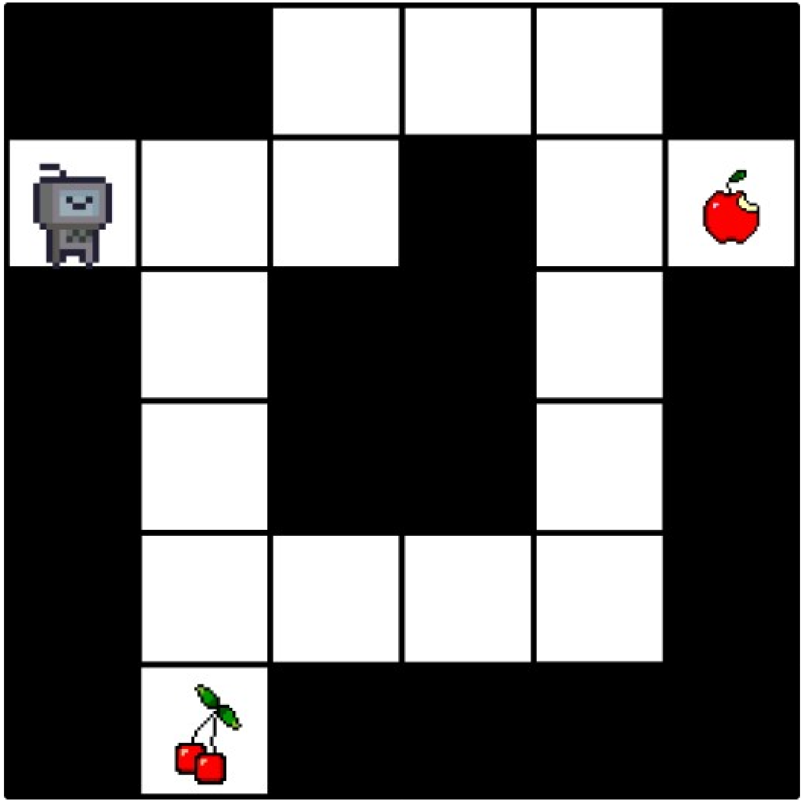
- Observations
  - At each time step, the agent can observe the environment, usually in a limited way, e.g.
    - Autonomous vehicle sensors
    - X, Y position in a maze
    - Pixels on the screen
  - In combination with rewards, this is the experience that an agent learns from

- Actions
  - (Usually) discrete set of actions to perform, e.g.
    - Move king's bishop to H3
    - Turn left 10 degrees
  - Actions affect the environment

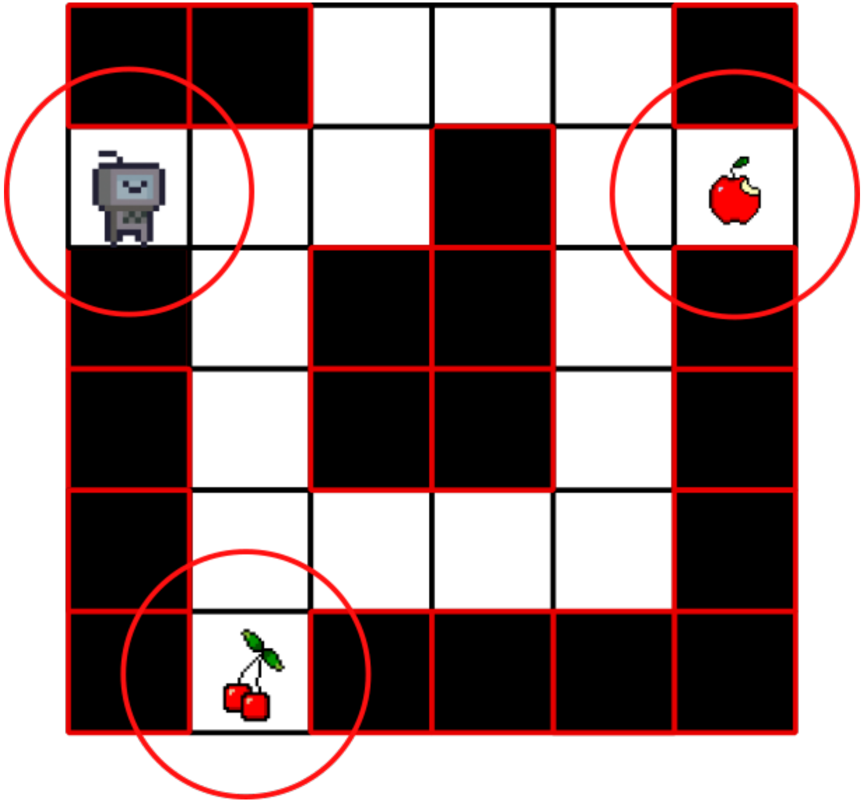
- Reward
  - Reward signal at each time step
  - Quantitative, e.g.
    - Change in score in a game
    - Distance robot has walked
    - End game reward 1 = win, -1 = loss

- From these
  - Observations
  - Actions
  - Reward
- An agent learns how to act in a way to maximise their **cumulative reward**
  - (A way of acting is called a **policy**)

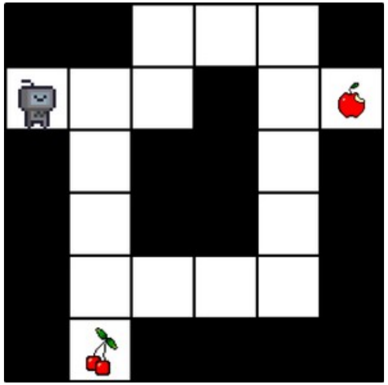
- Imagine we have a maze and our goal is to collect fruit
  - What are the:
    - Observations
    - Actions
    - Rewards

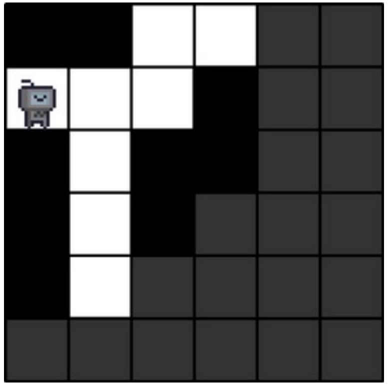


- Observations
  - The information about the environment we choose to give to the agent,
  - What the agent can "sense", e.g.
    - Position of agent
    - Positions of walls
    - Positions of fruit

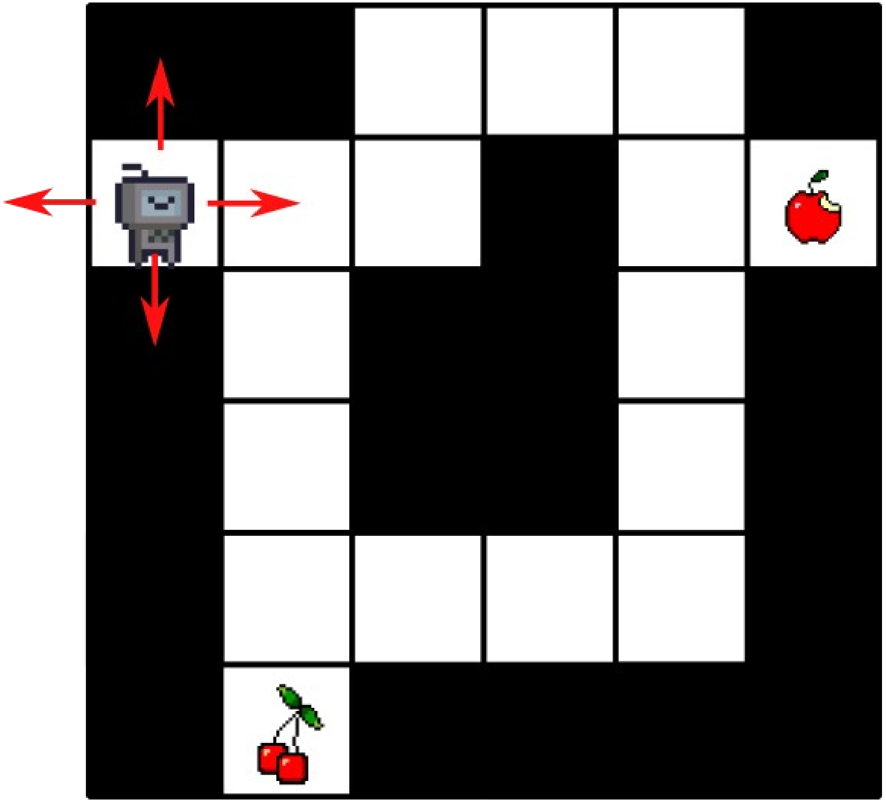


- The agent doesn’t always get all the information about the environment
  - **Fully observable**
    - Agent sees everything
  - **Partially observable**
    - Agent sees only part

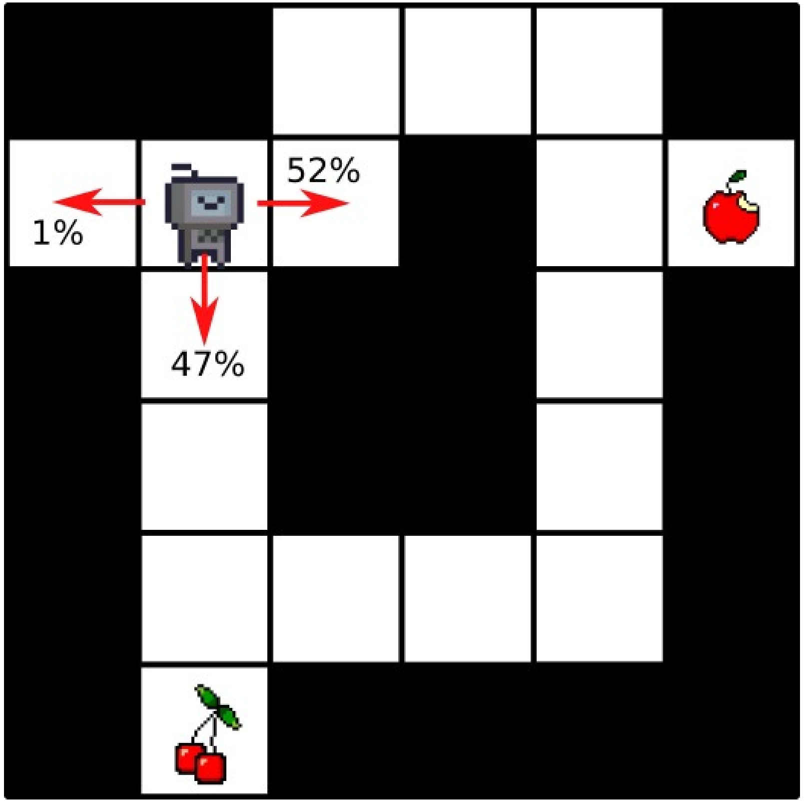




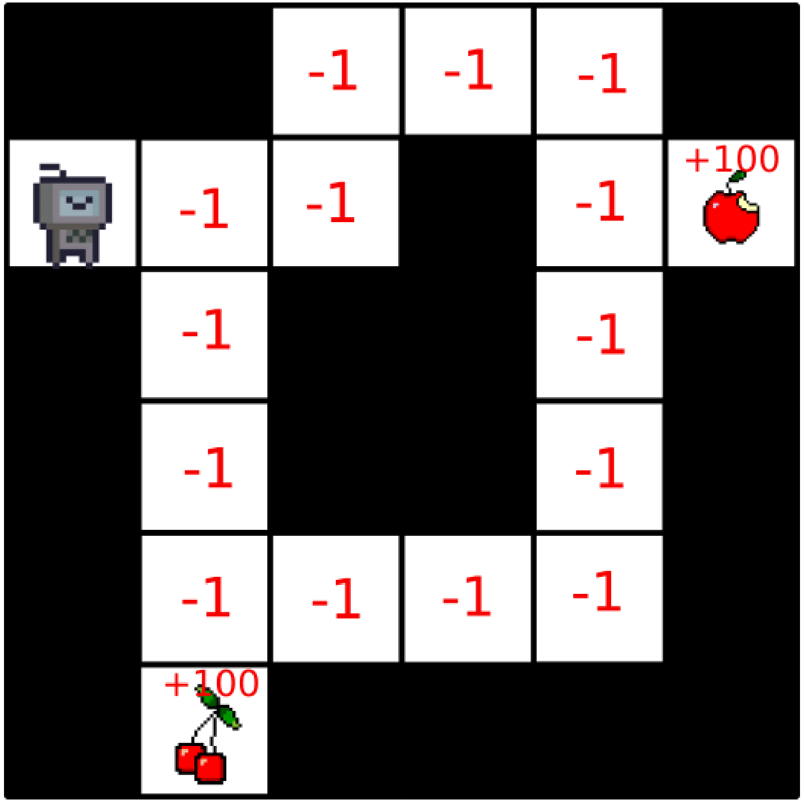
- Actions
  - The set of actions our agent can perform, e.g.
    - Move up
    - Move down
    - ...



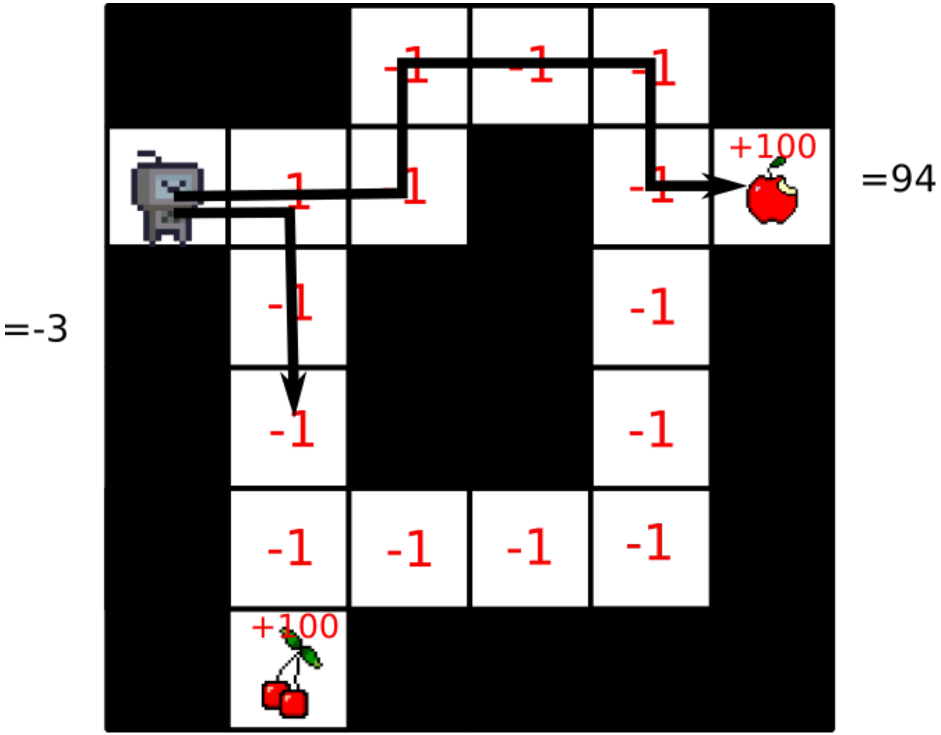
- A plan of action is called a **policy**, e.g.
  - Act randomly
  - Always turn left
  - Move to state with highest utility
- We want the agent to learn the **optimal policy**
  - There is always at least one optimal policy



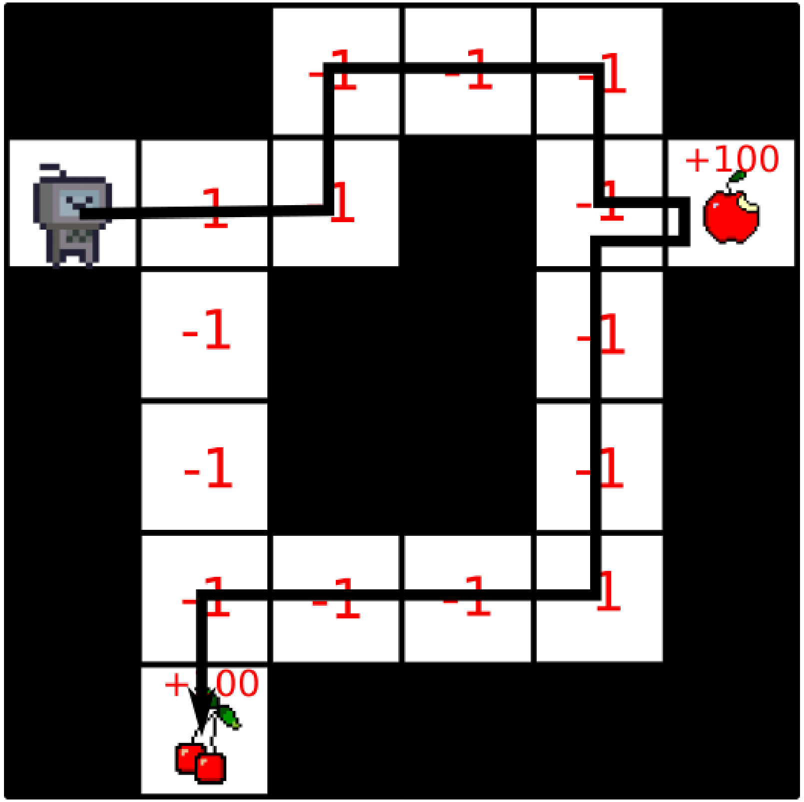
- Rewards
  - A score we give the agent when it does something we want
  - Agents often need to take several actions before they get a reward



- The agent learns by taking actions and getting rewards
  - It tries to maximise its cumulative reward
- In this example, it would (hopefully) learn to collect all the fruit with as few steps as possible

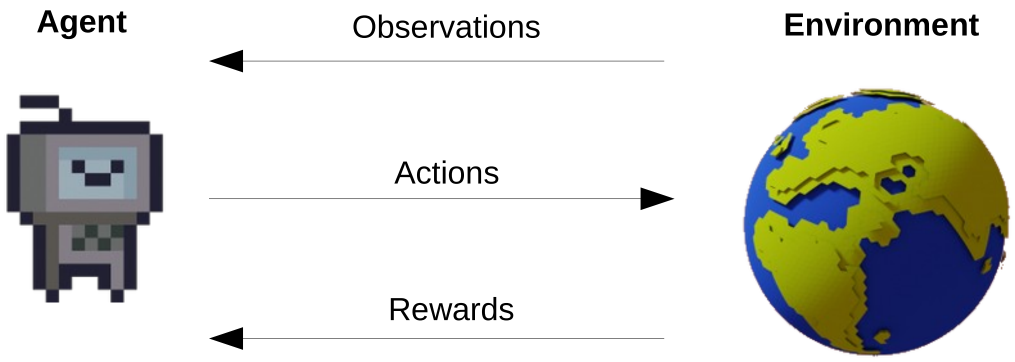


- **Return**
  - The cumulative reward over an episode (a sequence of states and actions)
- We want to learn the policy with the highest return



- Assumption:
  - Every goal can be expressed as the maximisation of some reward value, e.g.
    - Maximise game score
    - Maximise financial return
    - Optimise power output while following safety regulations
      - Here we might decide on the ‘exchange rate’ for efficiency vs. regulations

- RL algorithms use **observations** and **rewards** to decide on a **policy** for selecting **actions** in order to maximise their **return**



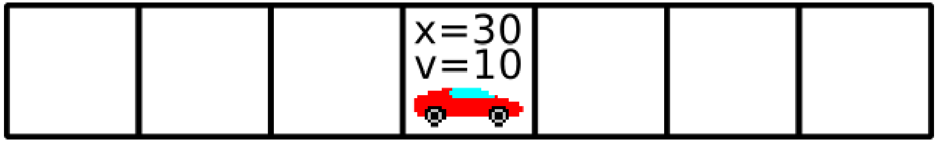
State

State

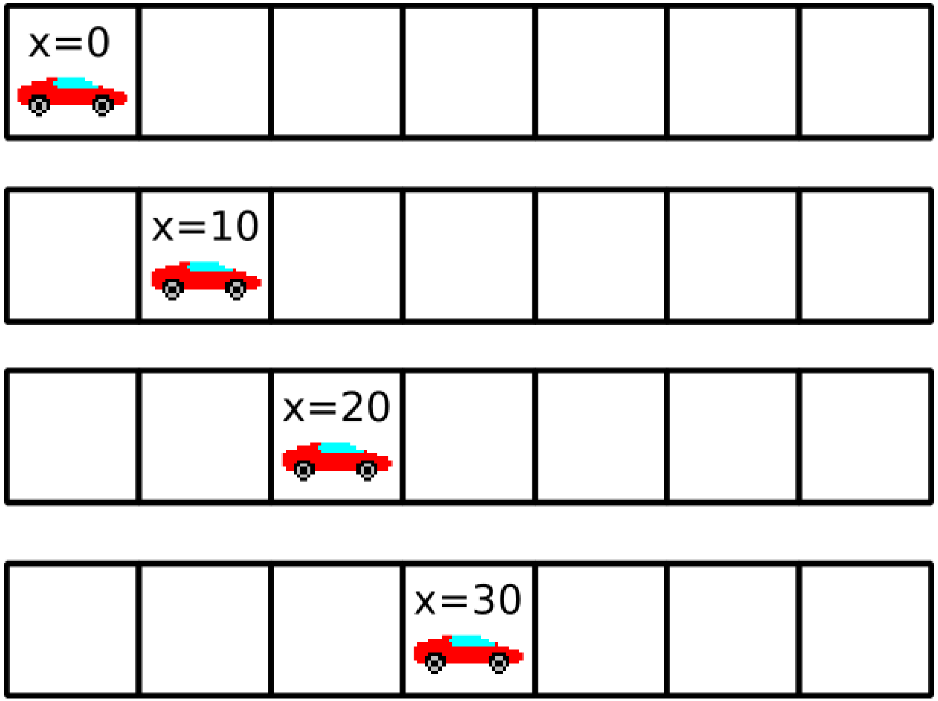
- Agents learn from their observations
- When deciding what to do, it is inefficient to recall past observations in full
  - Instead the agent abstracts a history of observations into a **state**

- State
  - A function of a history of observations
  - i.e. Abstract all the information seen so far into a useful summary that tells you ‘where you are’

- A state might include just the present observations
  - In a racing game we observe our position and velocity at each time step
  - Our state might contain our position and velocity
    - (So we can slow down at the corners)



- The state might incorporate past observations
  - In this game we now only observe position
  - To calculate velocity, we need information from past observations

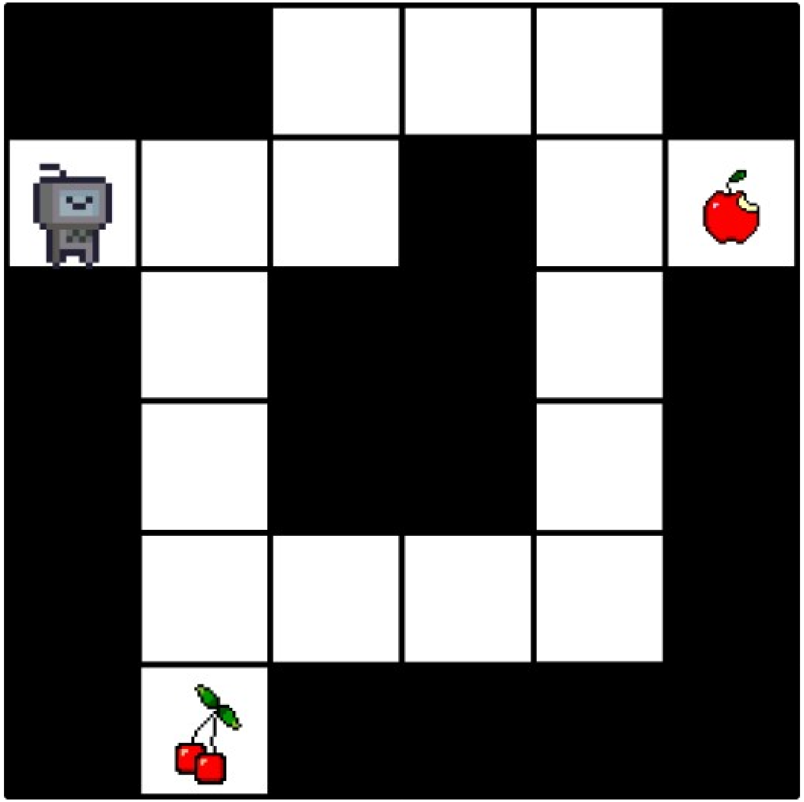


State

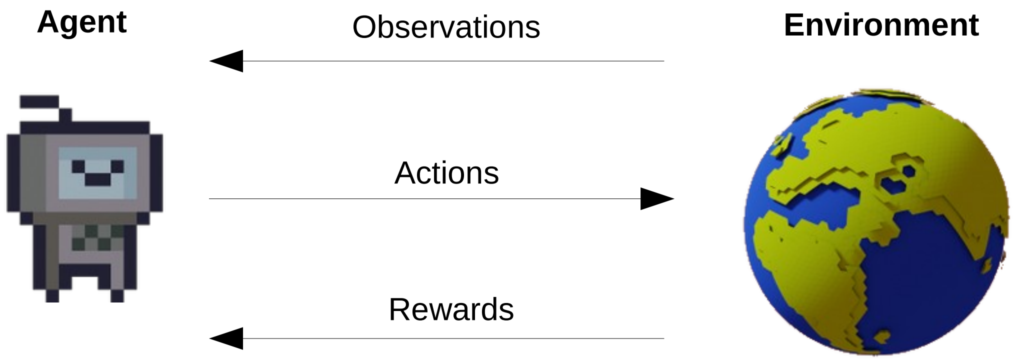
- Environment State
  - The state of the environment
    - Game state in chess
  - Often not accessible
    - e.g. autonomous vehicle

- Agent State
  - What state does the agent think they are in?
  - This represents (some of) the environment state
- As environment state is usually inaccessible, most of the time when we talk about state we mean agent state

- What is the environment state in the maze game?
- What information might we include in the agent state?



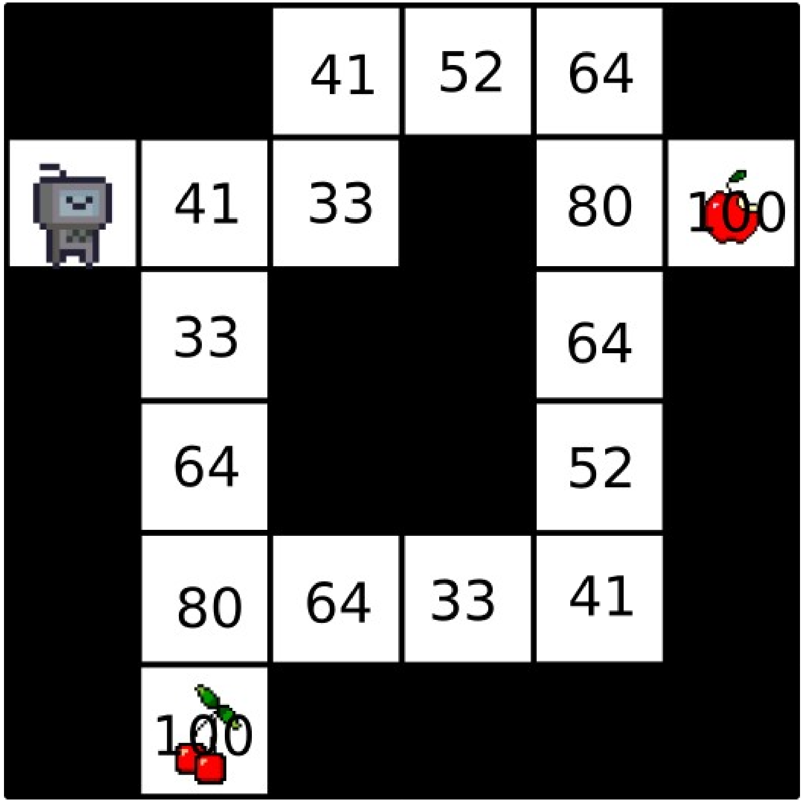
- RL algorithms abstract **observations** of the **environment** into a **state,** which tells them where they are and helps them to select **actions** to maximise their **return**



Parts of a Reinforcement Learning Agent

- There are three components that RL agents often have
  - Value Function
  - Model
  - Policy
- RL algorithms can be described by which of these it uses

- Value Function
  - A value function assigns each state a score
  - It's helpful to know how good any given state is
  - Value comes from our **total expected future reward** from that state

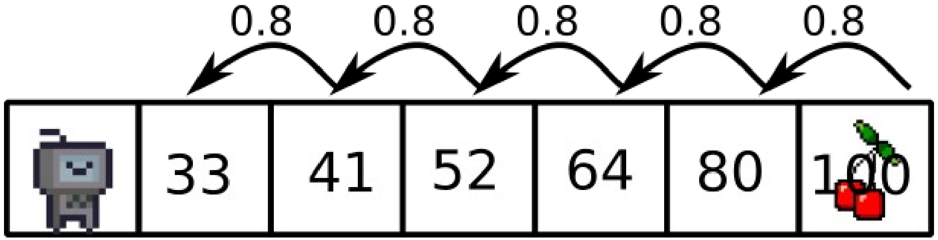


- Value combines **current and expected** future rewards

value of state = reward + value of next state

- This is a **Bellman Equation**
- For example
  - I'll get 100 points now, and I expect that – if I act optimally – I'll get 95 points in the future
  - So the value of the state is 195

- Discount factor
  - In practice, we multiply our expected future reward by a **discount factor**, e.g. 0.8
  - This decreases the importance of future rewards
- Why?
  - Future rewards are uncertain because our model is imperfect
  - Prevents infinite cumulative rewards which makes the maths harder

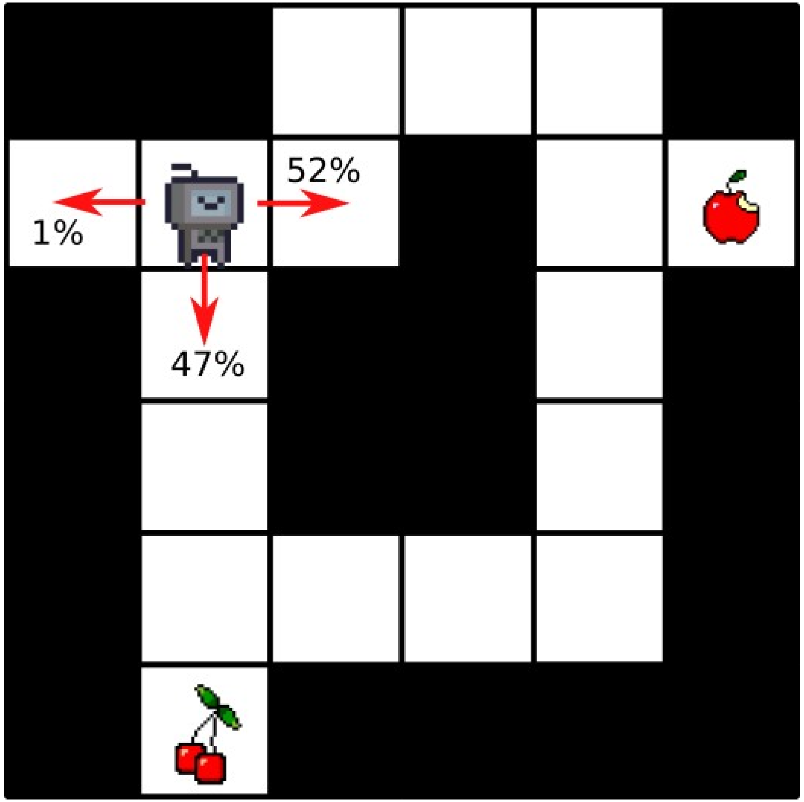


- The value of a state is thus:

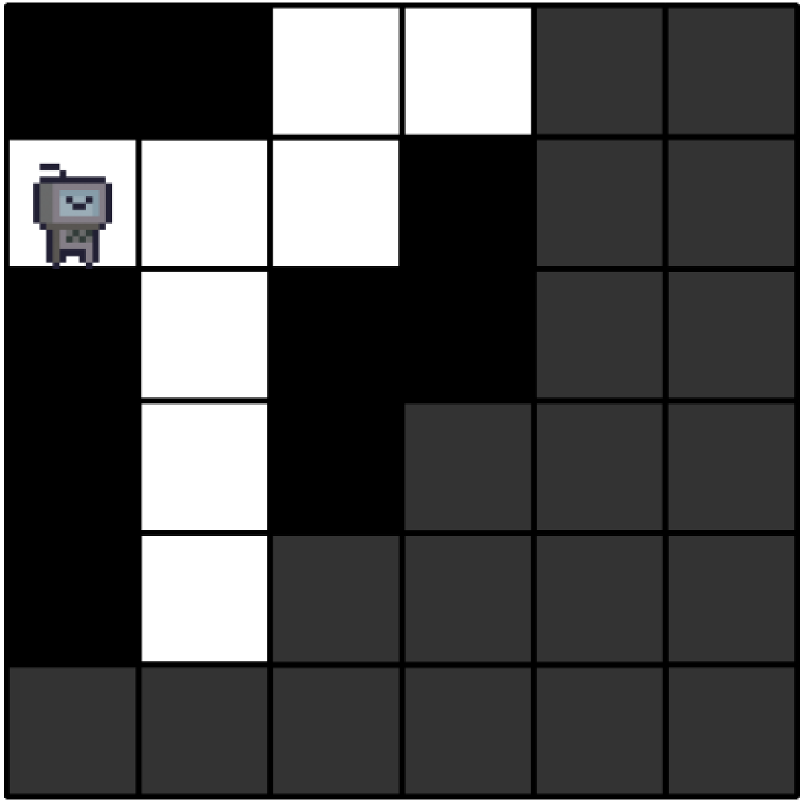
reward + discount factor * value of next state

- I'll get 100 points now, and I expect that – if I act optimally – I'll get to a state with a value of 95 in 1 time step, which is 0.8*95 = 76
  - So the value of the state is 176

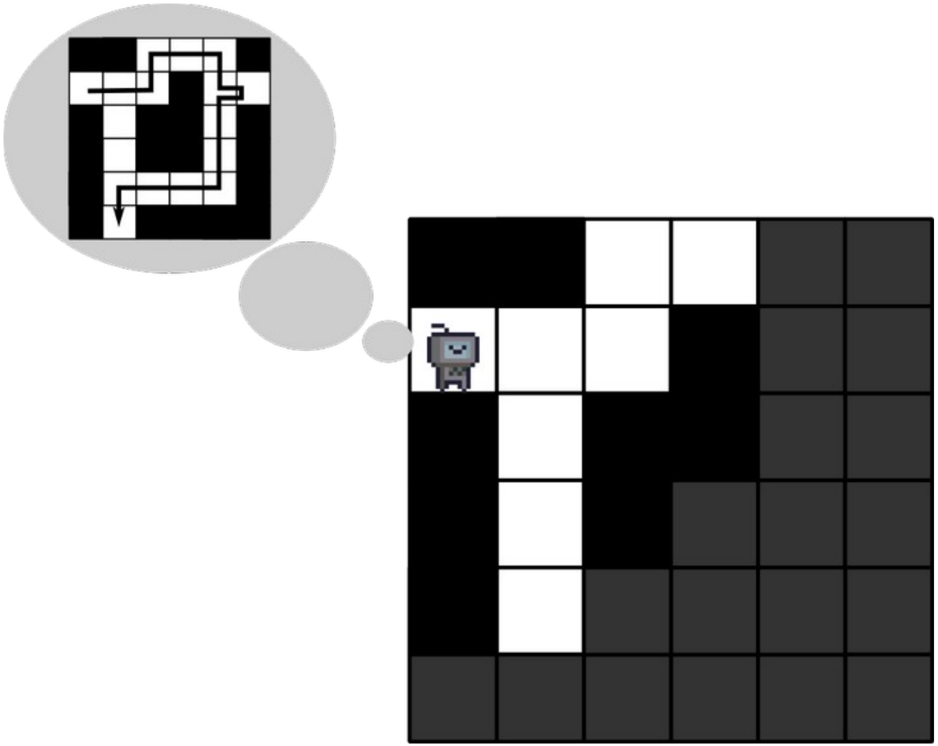
- Policy
  - For a given state, with what probability should we select the actions available to us?
  - If we have a value function, that give an **implicit policy** – always take the highest valued action



- Model
  - Does the agent try to build a model of the environment?
    - i.e. try and work out the dynamics of the environment
  - E.g. the maze agent might walk around finding all the walls to build a map of the environment



- Models allow planning
  - The maze agent could build a model and then use it to plan a route



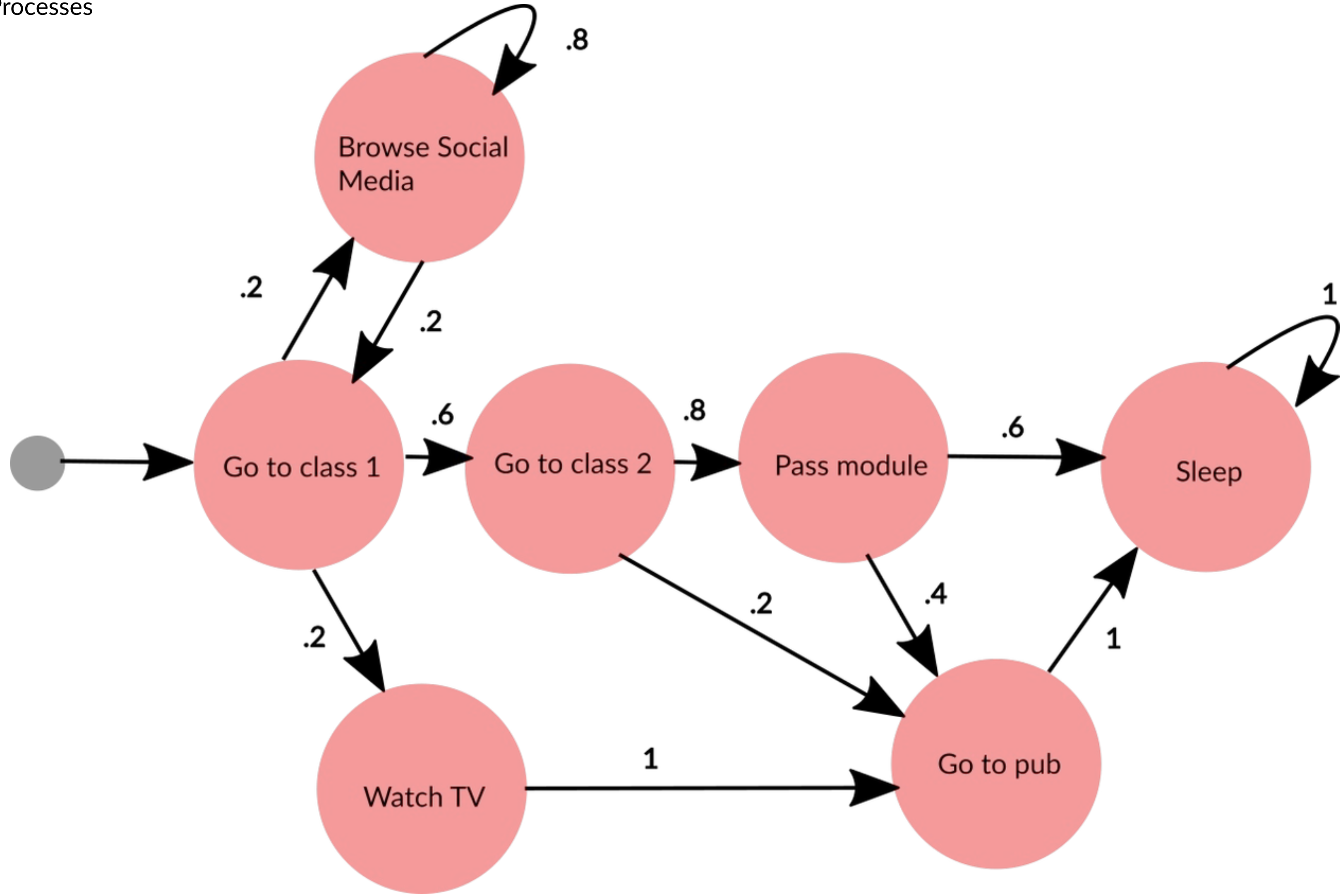
Markov Processes

- A Markov Process is a way of mathematically modeling stochastic sequences of events
  - It is a formal tool to help understand and build RL algorithms
    - (It’s widely used outside RL as well)
  - Basically a finite state machine where the transitions are uncertain

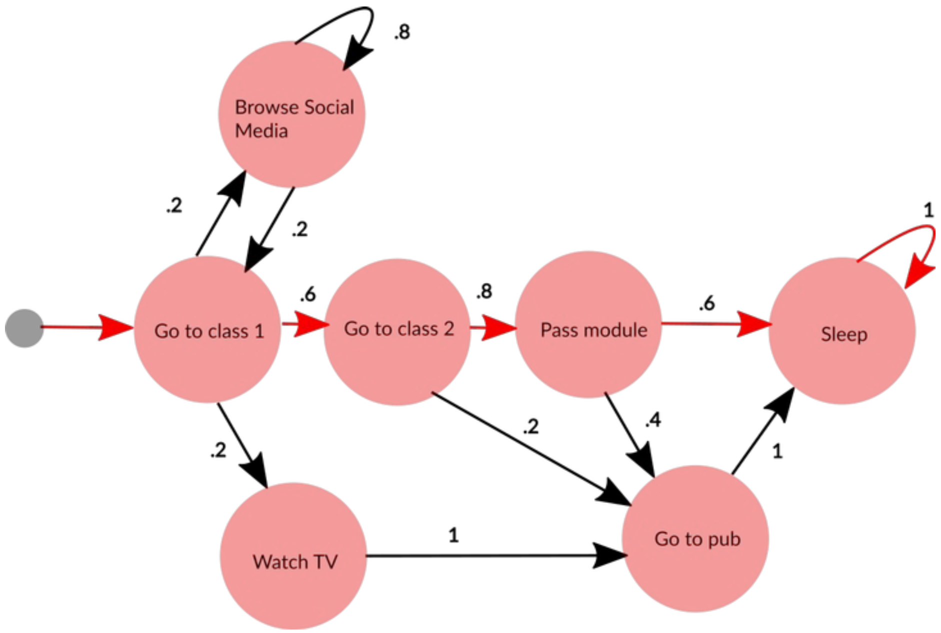
- Markov Process are sequences of states where the probability of each state is dependent only on the last state
  - (Here we have a third meaning of "state": **information state**)
- A state contains all information about past states
  - i.e. knowing the history gives you no more information about what's happening next

$$P(S_t \mid S_{t-1}) = P(S_t \mid S_1, S_2, \ldots S_{t-1})$$

Markov Processes

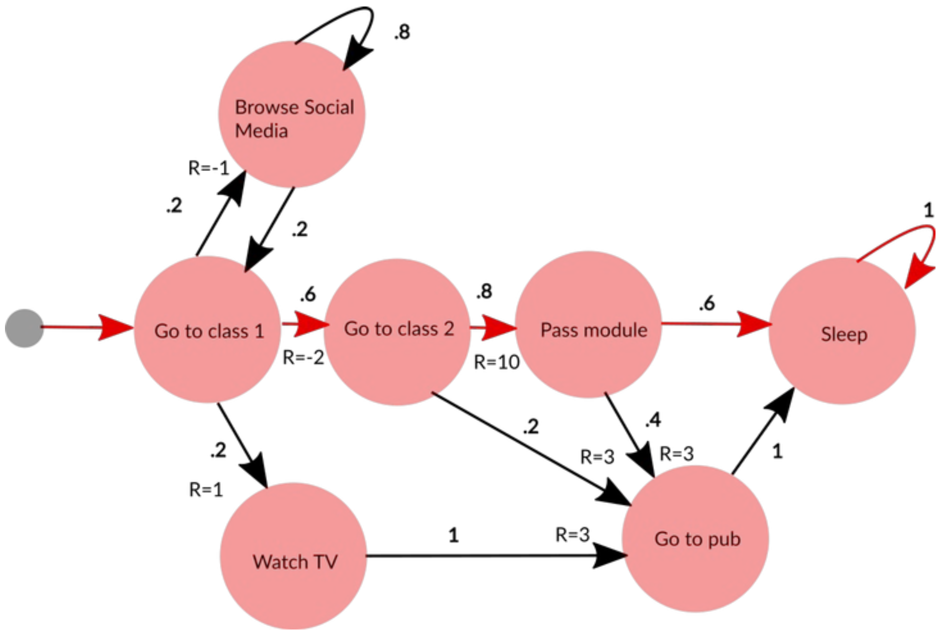


- We can work out the probabilities of taking different paths through a Markov Process
  - This path has probability
    .6 x .8 x . 6 = 0.288

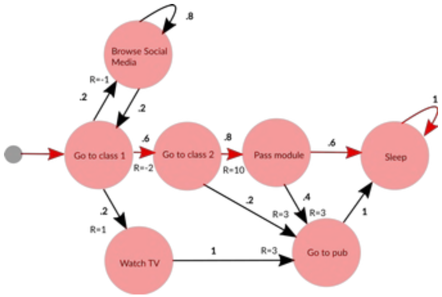


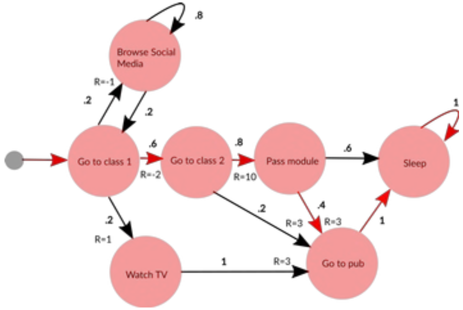
- Lets add rewards to make this a Markov Reward Process
  - Now we can work out the **return** of this path by adding the rewards

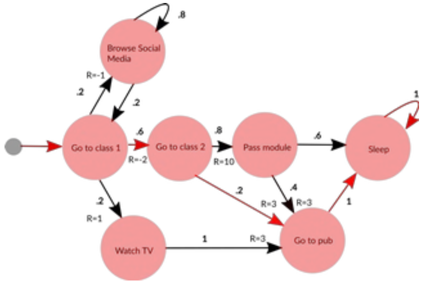
    -2 + 10 = 8



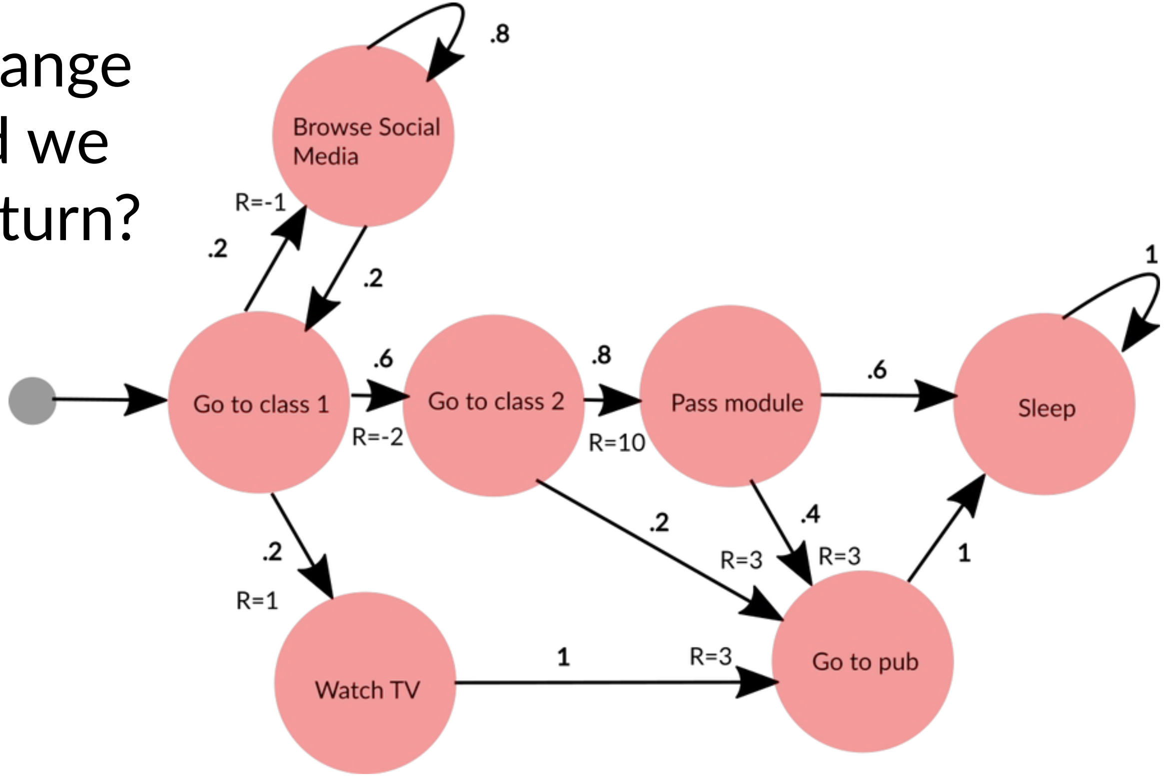
- We can also work out the expected cumulative future reward from a state
  - Considering **probability** and **return** of each path



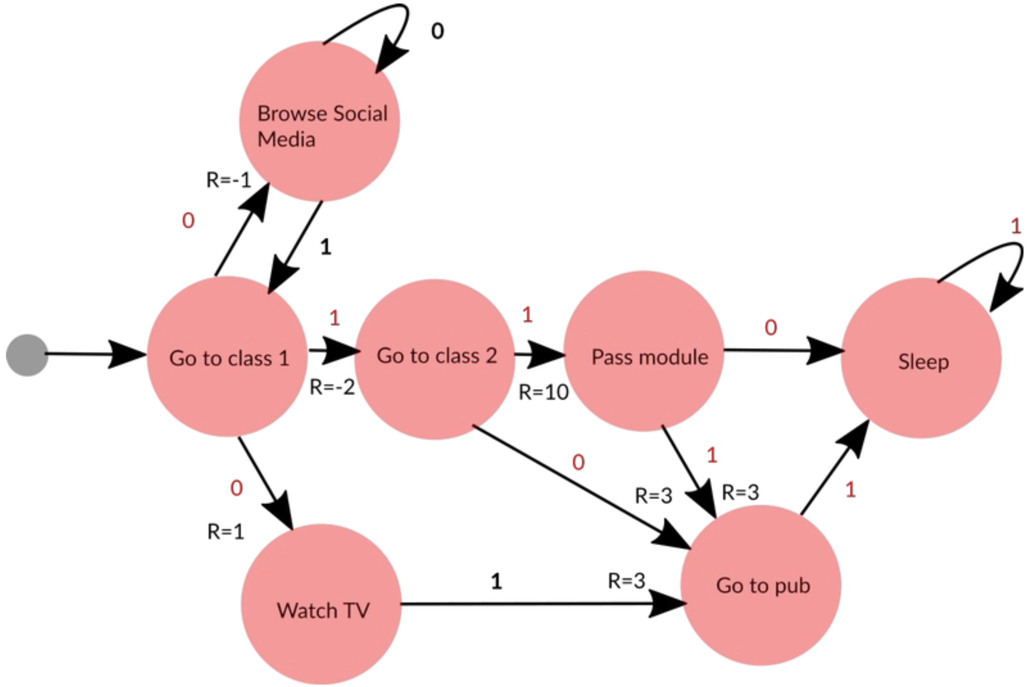




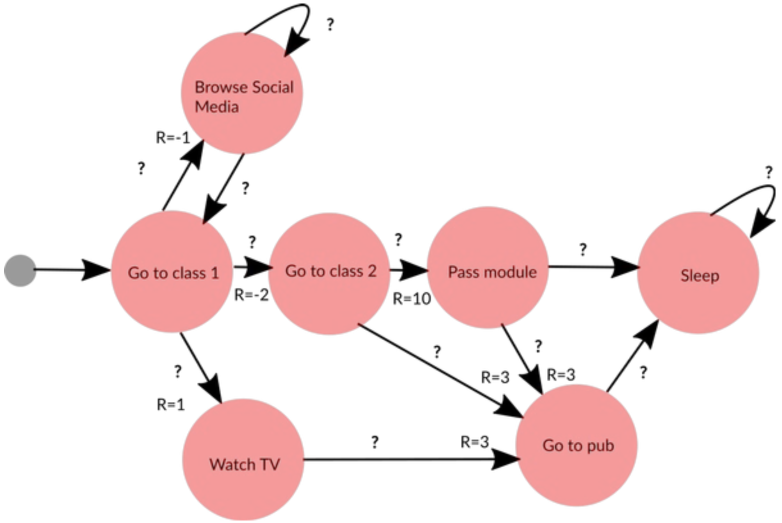
- **Question:** If we could change these probabilities, could we improve our expected return?

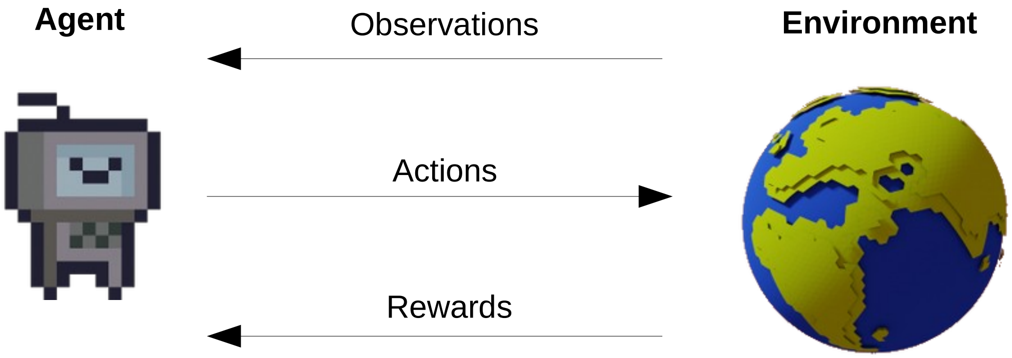


- The probabilities of taking each action are our agent's **policy**
- Our RL agent wants to **learn the best probabilities** to use
  - (Because we're making decisions, it's now called a Markov Decision Process)



- So, an RL algorithm, faces a **Markov Decision Process** and needs to work out the action **probabilities** to maximise their **expected return**
  - (The example here was fully observable)





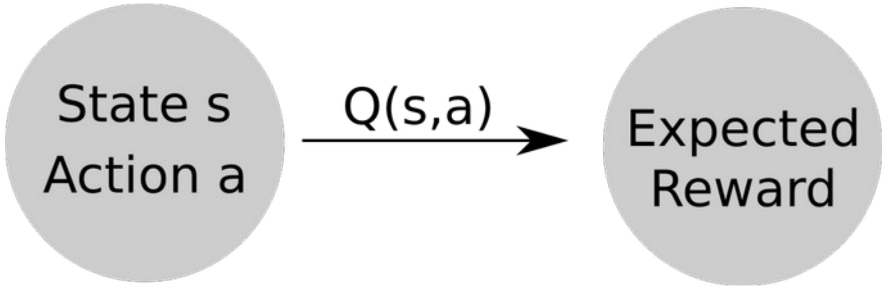
Q Learning

- Necessary functions in an RL algorithm
  - Reward Prediction: Predicting expected reward in given state
  - Choice: Optimal action selection
- Q learning is one approach to solving this

- Approach: Learn a **value function** (Q function) for each state-action pair
  - i.e. what's the value of doing action A in state S?
- Take actions with highest value
  - An implicit policy
- **Challenge**: representing and fitting the Q function

- Q learning uses Temporal Difference (TD) Learning
  - Derive reward predictions from delayed rewards
    - Learn by bootstrapping from estimates of value function
    - Sample the environment
    - Update based on current estimates
  - Model free

- Q Learning
  - Learn a function Q that take an **action** a ∈ A in a given **state** s ∈ S and returns the **expected future reward**
    - Q: S x A → R



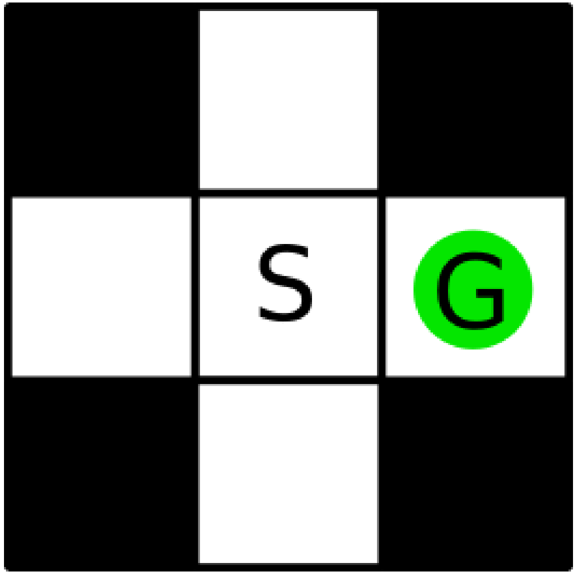
- Simplest approach is to use a Q table
  - Represent every combination of inputs (state-action pairs) with their output
  - i.e. for each state and action, what is the expected future reward?

	a_1	a_2	a_3	a_4
s_1	24	57	7	98
s_2	42	0		56
s_3	1		76	0
s_4	24	34	55	
s_5	5	3	62	3

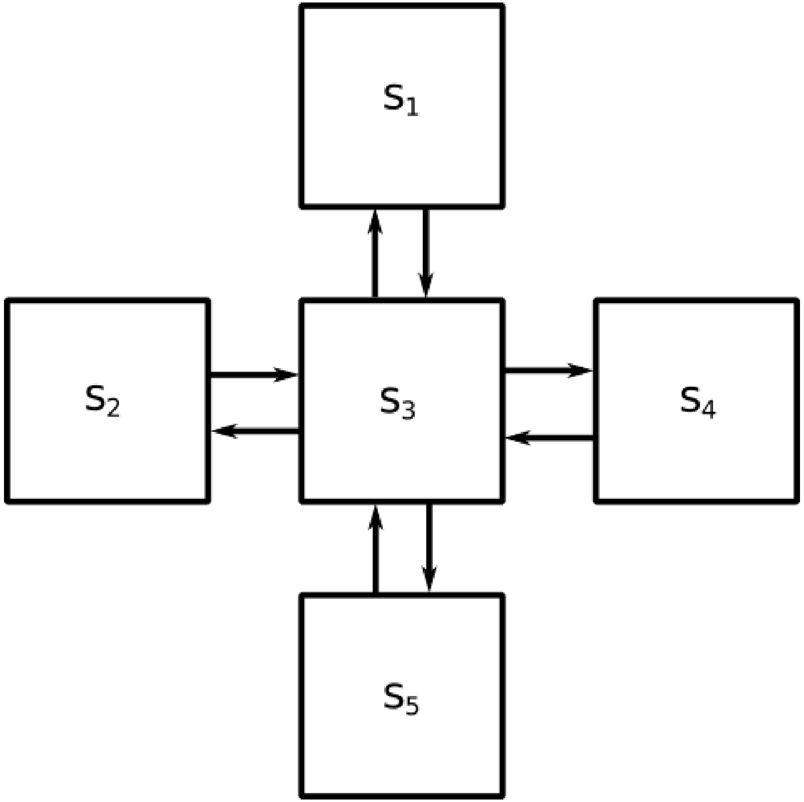
- Start with arbitrary values (e.g. 0)
- Bootstap the correct values by a series of increasingly accurate approximations
  - Start using arbitrary values and improve them when we can

	a_1	a_2	a_3	a_4
s_1	0	0	0	0
s_2	0	0	0	0
s_3	0	0	0	0
s_4	0	0	0	0
s_5	0	0	0	0

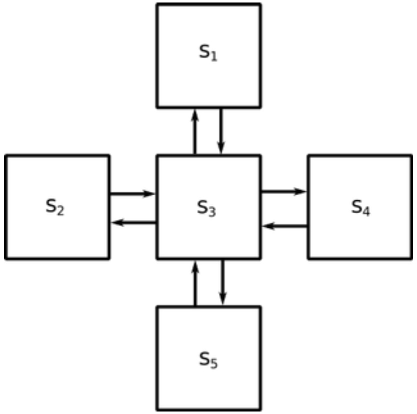
- Imagine we have a simple maze game
  - The agent starts somewhere on the grid shown (S)
  - They can move up, down, left, or right
- If they move to a goal (G) they get a reward of 100 points



- Let's represent this as 5 states and 4 actions
  - States $s_1, s_2 \ldots \in S$
  - Actions $a_{up}, a_{down} \ldots \in A$



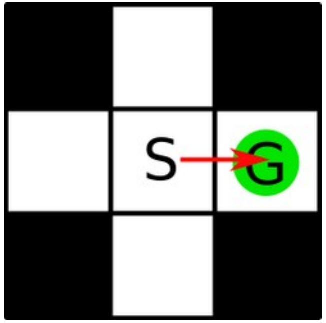
- Create a table of states/action pairs (**Q table**)
  - Stores the utility of each action from each state
  - Initialise with arbitrary values

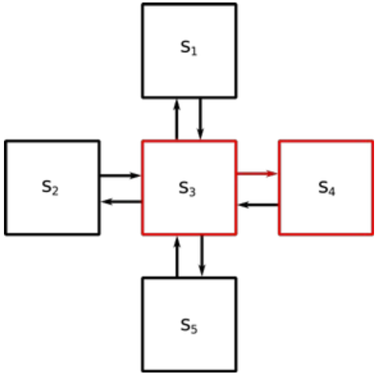


	a_{up}	a_{down}	a_{left}	a_{right}
s_1	0	0	0	0
s_2	0	0	0	0
s_3	0	0	0	0
s_4	0	0	0	0
s_5	0	0	0	0

- Imagine we start in state s_3
  - We pick an action to perform at random
    - For demonstration purposes, let's choose a_{right} (highlighted)
  - Taking this action takes us to the goal, so we give a reward of 100
- We've learned something about the value of that action in that state
  - It's pretty good!
  - We need to update $Q(s_3, a_{right})$

	a_{up}	a_{down}	a_{left}	a_{right}
s_1		0		
s_2				0
s_3	0	0	0	0
s_4			0	
s_5	0			





- We need to update $Q(s_3, a_{right})$
  - We had an expectation of its value
  - We now have better estimate
    - We have now visited it and know what **reward** we got
    - We can look up the **expected future reward** (from the **best-scoring action** from the new state)
  - We multiply the difference by a learning rate (e.g. 0.8)
- We use a **Bellman Equation**:

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Bigg(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Bigg)}^{\text{temporal difference}}$$

- So, to update $Q(s_3, a_{right})$
  - $Q^{old}(s_3, a_{right}) = 0$
  - Learning rate = 0.8
  - Reward = 100
  - Discount factor = 0.8
  - Max(0) = 0
  - $Q^{new}(s_3, a_{right}) = 0 + 0.8 * (100 + 0.8 * 0 - 0) = 80$

	a_{up}	a_{down}	a_{left}	a_{right}
s_1		0		
s_2				0
s_3	0	0	0	0
s_4			0	
s_5	0			

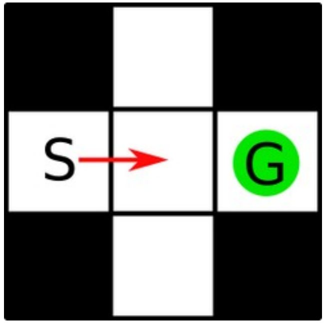
$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\underbrace{\Big(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Big)}^{\text{temporal difference}}$$

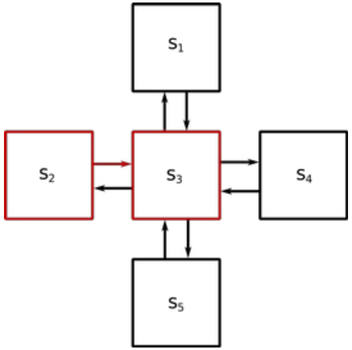
- We have updated the Q table

	a_{up}	a_{down}	a_{left}	a_{right}
s_1		0		
s_2				0
s_3	0	0	0	80
s_4			0	
s_5	0			

- Imagine we start again from state s_2
  - We pick an action to perform at random
    - Let's choose a_{right} (highlighted)
  - We don't get a reward
- We've learned something about the value of that action
  - There was no reward, but it gets us to a state from which there is a good action
  - We need to update $Q(s_2, a_{right})$

	a_{up}	a_{down}	a_{left}	a_{right}
s_1		0		
s_2				0
s_3	0	0	0	80
s_4			0	
s_5	0			





- We need to update $Q(s_2, a_{right})$
  - $Q^{old}(s_2, a_{right}) = 0$
  - Learning rate = 0.8
  - Reward = 0
  - Discount factor = 0.8
  - Max(0, 0, 0, 80) = 80
  - $Q^{new}(s_2, a_{right}) = 0 + 0.8 * (0 + 0.8 * 80 – 0) = 64$

	a_{up}	a_{down}	a_{left}	a_{right}
s_1		0		
s_2				0
s_3	0	0	0	80
s_4			0	
s_5	0			

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\bigg(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \bigg)}^{\text{temporal difference}}$$

- We have updated the Q table

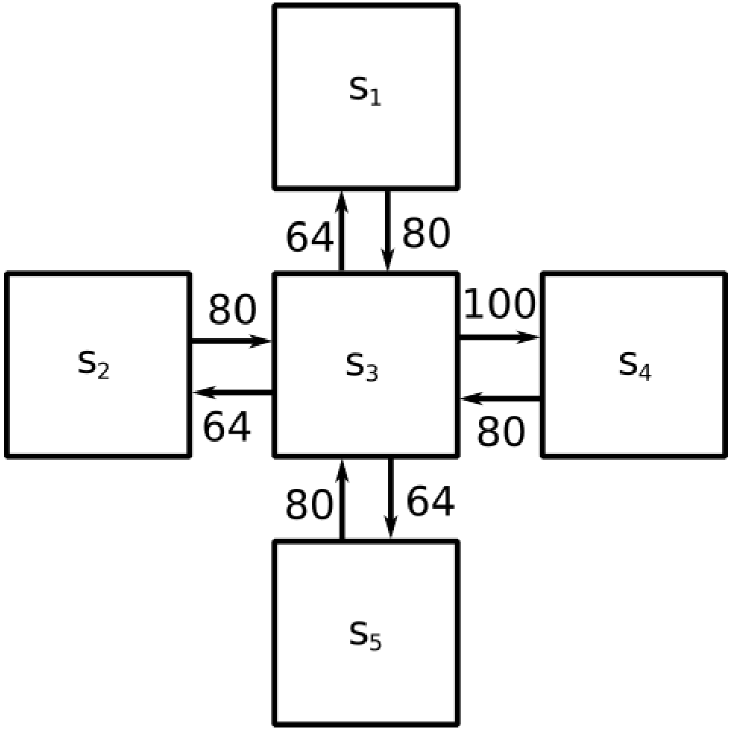
	a_{up}	a_{down}	a_{left}	a_{right}
s_1		0		
s_2				64
s_3	0	0	0	80
s_4			0	
s_5	0			

- If we keep randomly restarting and updating the table, eventually we'll converge on a table of values like those shown
- We can normalise these values by dividing by the highest (500)
  - (and multiplying by 100 to avoid decimals)

	a_{up}	a_{down}	a_{left}	a_{right}
s_1		400		
s_2				400
s_3	320	320	320	500
s_4			400	
s_5	400			

	a_{up}	a_{down}	a_{left}	a_{right}
s_1		80		
s_2				80
s_3	64	64	64	100
s_4			80	
s_5	80			

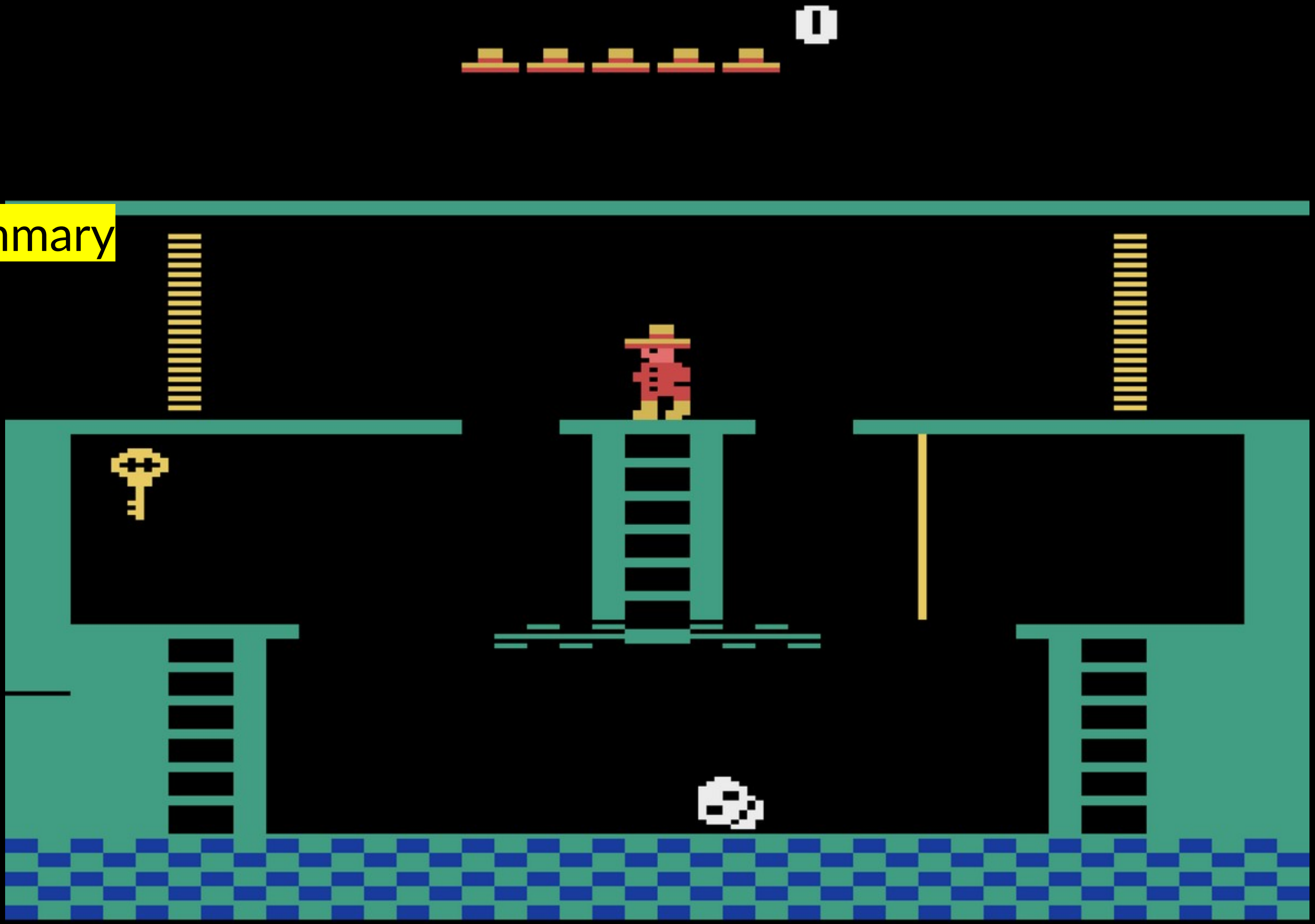
- This table gives defines the Q function that we have learned
  - Gives utility of state/action pairs
- We can use this to take the optimal move from each state by always taking the action with the highest value



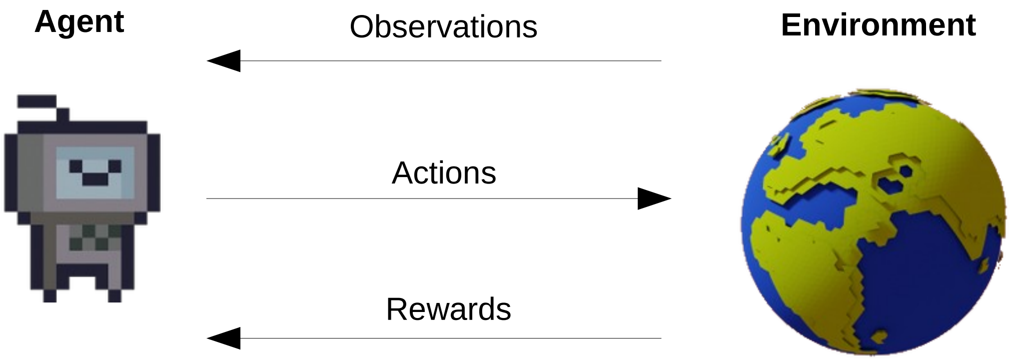
- It doesn't always make sense to explore the whole state space first
- We could choose to always start from the same state
  - Then we would explore different paths to the goal
  - Then we balance
    - Exploration – trying new/untried actions
    - Exploitation – picking high-value actions to focus effort on promising directions

- Often the state space is too big to use a table
- Instead of learning a function Q expressed as a table (exhaustively calculating and storing every value), Q can be a function approximation
  - Neural networks learn to approximate functions
  - Deep reinforcement learning uses a deep neural network as a function approximator
    - Deep Q-Learning is used in AlphaGo

Summary



- RL agents learn from interacting with an environment and getting rewards
- They represent their history of observations in a **state**
  - **Value functions** assign a value to a state
  - They decide how to act using a **policy**
  - They might learn a **model** of the environment and do planning



- One approach is to learn a Q function that gives the expected future reward from each state-action pair
  - Store a value for each pair in a table
  - Update these as you explore and discover which actions are rewarded
- A neural network could be used to approximate the Q function

- Recommended Reading
  - AlphaGo Documentary
    - https://youtu.be/WXuK6gekU1Y
  - Lecture series taught by David Silver of DeepMind
    - https://youtu.be/2pWv7GOvuf0
  - RL in Unity using ML Agents
    - https://medium.com/nerd-for-tech/an-introduction-to-machine-learning-with-unity-ml-agents-af71938ca958
    - https://github.com/Unity-Technologies/ml-agents/blob/release_18_docs/docs/Installation.md#clone-the-ml-agents-toolkit-repository-optional
  - Q Learning worked example:
    - https://people.revoledu.com/kardi/tutorial/ReinforcementLearning/Q-Learning-Example.htm