In [19]:

```
import tensorflow as tf
# <a class="tocSkip">
!/home/per/venv/jupyterlab/bin/jupyter-nbconvert --to slides main.ipynb
!pip install pillow --user
!pip install pandas --user
import PIL.Image
```

```
[NbConvertApp] Converting notebook main.ipynb to slides
[NbConvertApp] Writing 397723 bytes to main.slides.html
Requirement already satisfied: pillow in /home/per/.local/lib/python
3.8/site-packages (7.0.0)
Requirement already satisfied: pandas in /home/per/.local/lib/python
3.8/site-packages (0.25.3)
Requirement already satisfied: numpy>=1.13.3 in /home/per/.local/lib/
python3.8/site-packages (from pandas) (1.18.1)
Requirement already satisfied: pytz>=2017.2 in /home/per/.local/lib/p
ython3.8/site-packages (from pandas) (2019.3)
Requirement already satisfied: python-dateutil>=2.6.1 in /home/per/.l
ocal/lib/python3.8/site-packages (from pandas) (2.8.1)
Requirement already satisfied: six>=1.5 in /home/per/.local/lib/pytho
n3.8/site-packages (from python-dateutil>=2.6.1->pandas) (1.13.0)
```

IKT-441: Lecture 2

Reinforcement Learning

By Per-Arne Andersen

Table of Contents

Before we begin!

Reinforcement Learning: An Introduction - [http://www.andrew.cmu.edu/course/10-703/textbook/BartoSutton.pdf (http://www.andrew.cmu.edu/course/10-703/textbook/BartoSutton.pdf)](http://www.andrew.cmu.edu/course/10-703/textbook/BartoSutton.pdf)

Part 1: Introduction

Supervised Learning

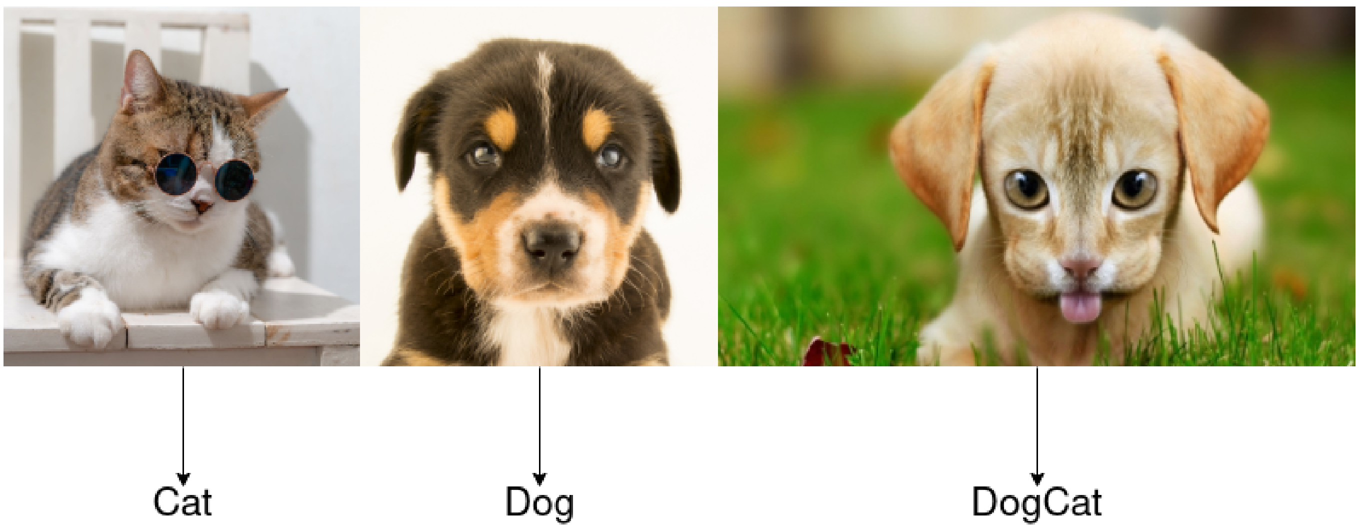
Dataset: You start with **labeled** data. The supervision is the label

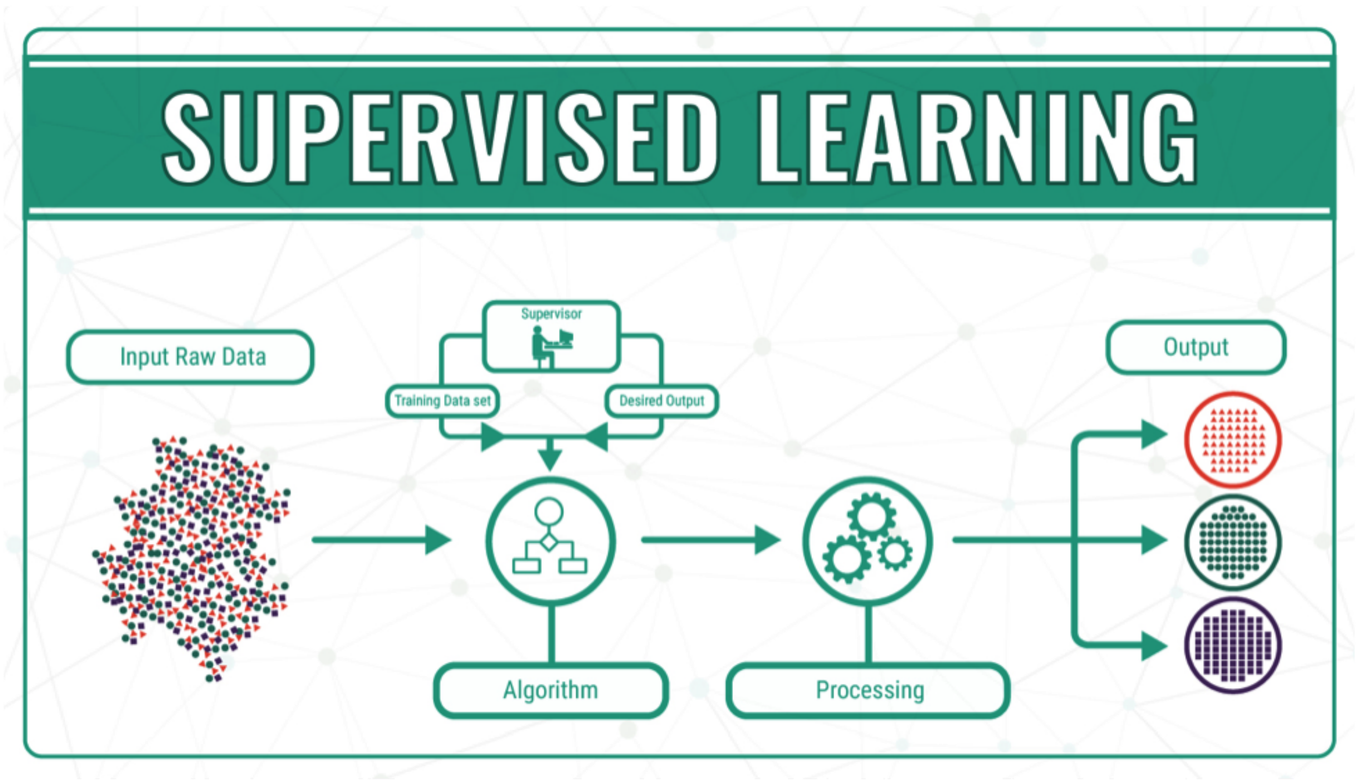
Goal: Learn a **function** that **maps** data (x) samples to its corresponding **label** (y)

Examples: Classification, regression, object detection ...etc

Algorithms: SVM, Naïve Bayes, Multilayer perceptron (NN)

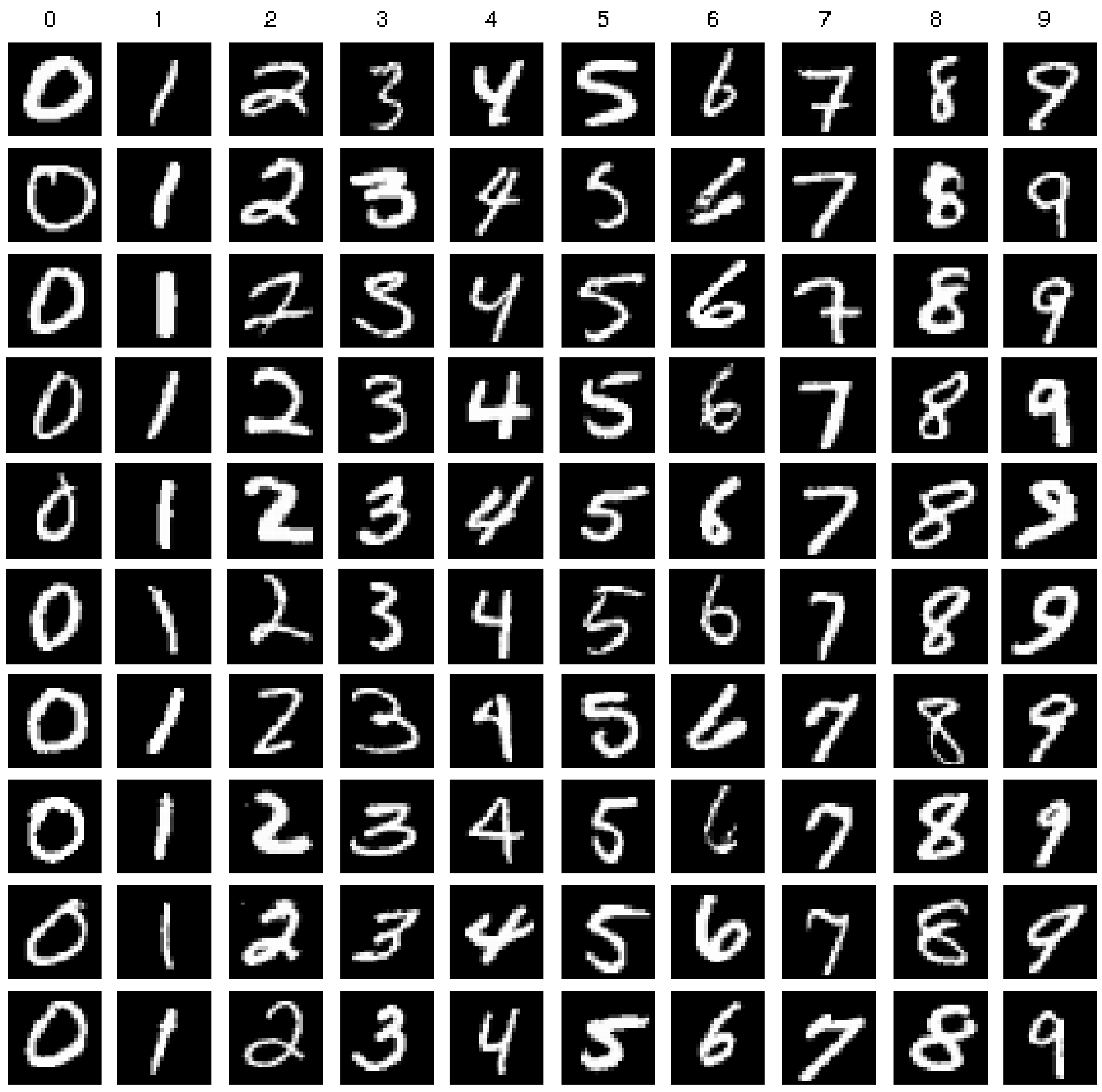
$$y = f(x)$$




SUPERVISED LEARNING
Input Raw Data
Supervisor
Training Data set
Desired Output
Output
Algorithm
Processing

Example - MNIST

- Classify handwritten digits with keras



In [20]:

```
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(1, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation='softmax')
])
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5, verbose=2)
"Error: %s, Accuracy: %s" % tuple(model.evaluate(x_test,  y_test, verbose=2))
```

```
Train on 60000 samples
Epoch 1/5
60000/60000 - 1s - loss: 2.1003 - accuracy: 0.1764
Epoch 2/5
60000/60000 - 1s - loss: 1.9696 - accuracy: 0.2064
Epoch 3/5
60000/60000 - 1s - loss: 1.9417 - accuracy: 0.2219
Epoch 4/5
60000/60000 - 1s - loss: 1.9248 - accuracy: 0.2366
Epoch 5/5
60000/60000 - 1s - loss: 1.9118 - accuracy: 0.2482
10000/10000 - 0s - loss: 1.7493 - accuracy: 0.3175
```

Out[20]:

```
'Error: 1.749269065284729, Accuracy: 0.3175'
```

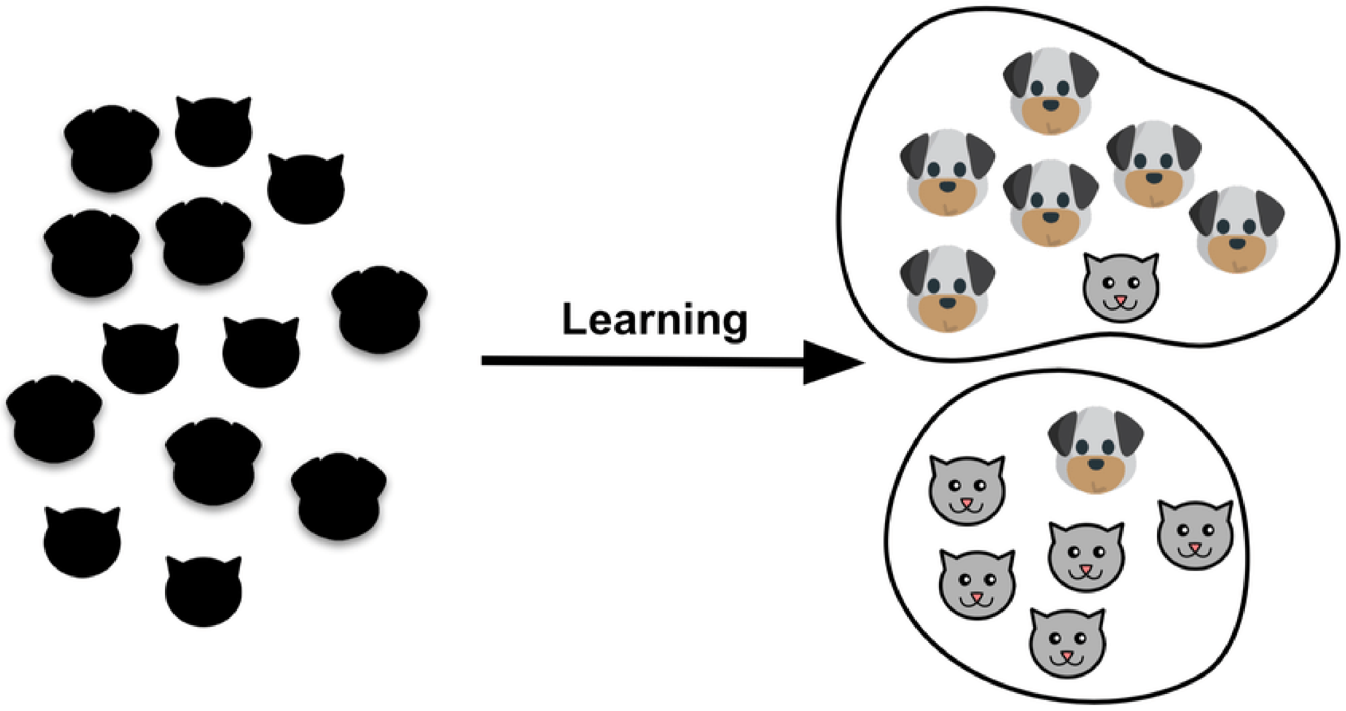
Unsupervised Learning

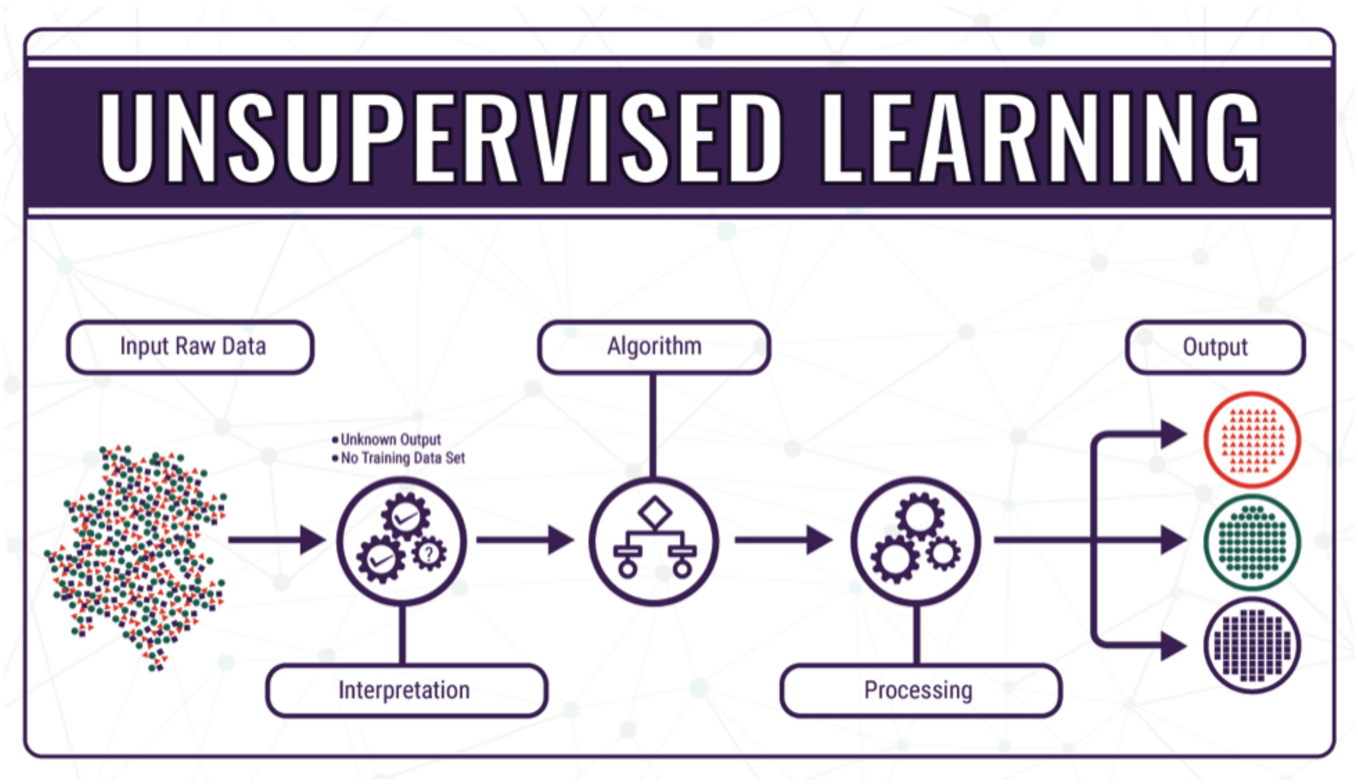
Dataset: Contains datapoints. No Labels

Goal: Find the hidden structure of the data

Examples: Clustering, Feature Learning, Dimensionality Reduction

Algorithms T-SNE, PCA


Learning



Part 2: Reinforcement Learning Applications

Cartpole

```
* Discrete or continous action-space
```

When Discrete:

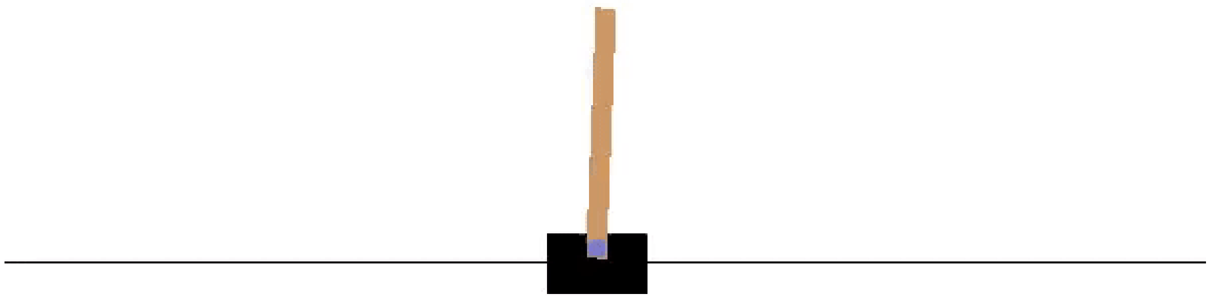
```
* 2 Actions
* Left Push, Right Push
```

When Continous:

```
* Any real value between -1.0 and 1.0
```

State-Space:

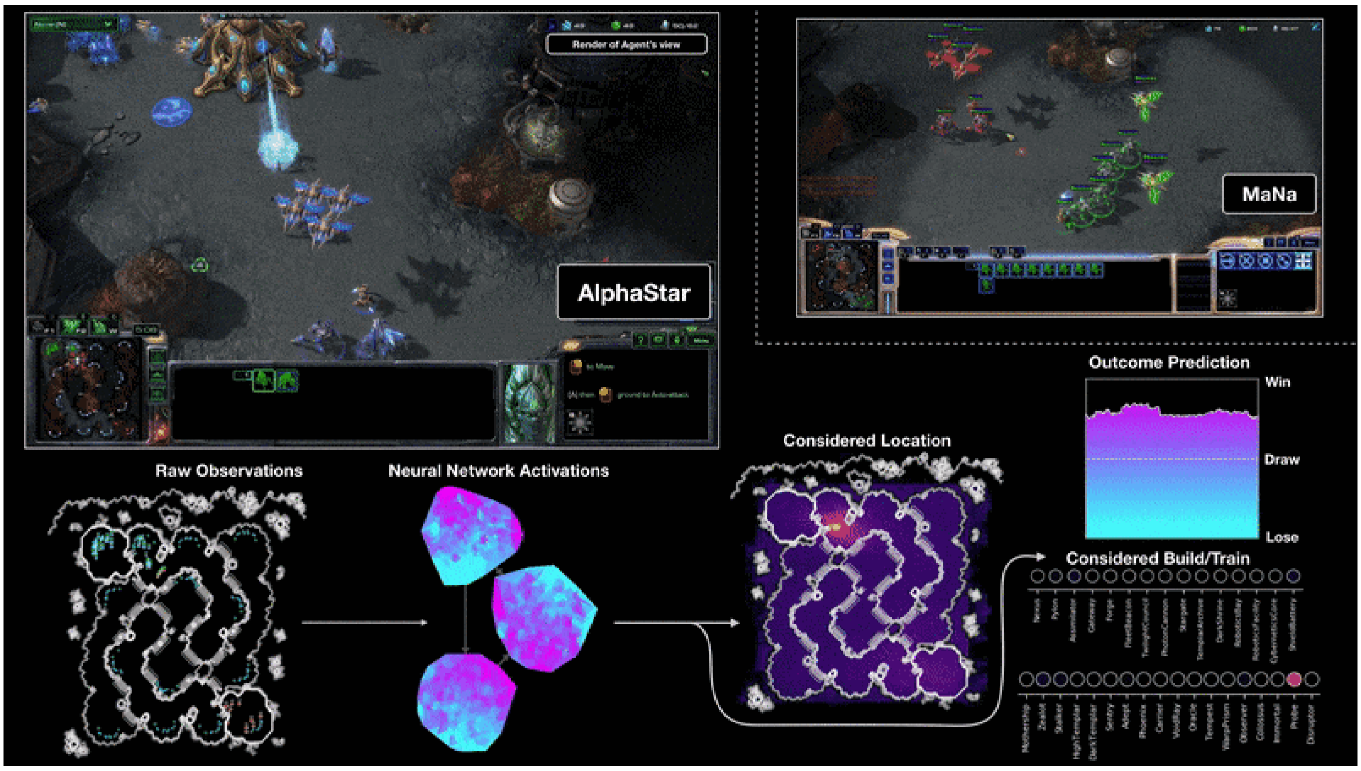
```
* Vector of 4 values.
```



Num	Observation	Min	Max
0	Position	-4.8	4.8
1	Velocity	-Inf	Inf
2	Pole Angle	-24deg	24deg
3	Pole Velocity	-Inf	Inf

Starcraft II

- Considered as one of the most difficult games to master
- Real-Time Strategy Game
- Approximately 10^{26} possible moves at any given time
- Deep Mind's AlphaStar
- Just think ∞ state-space and action-space



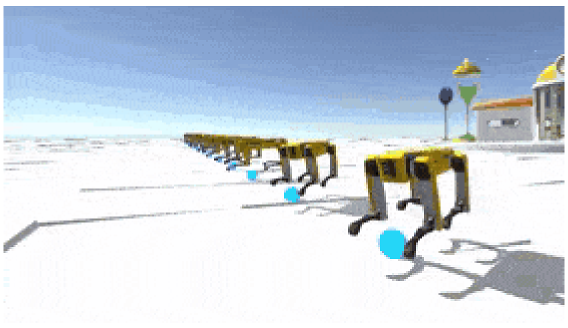
In [21]:

```python
import numpy as np

for t in [np.int8, np.float64]:
    print("Type: ", t.__name__, ", Memory: ", (10**26 * np.dtype(t).itemsize) * 1e-9, "GB")
```

```
Type:  int8 , Memory:  1.0000000000000002e+17 GB
Type:  float64 , Memory:  8.000000000000001e+17 GB
```

Locomotion

- Continous Control tasks
    - Controlling a robot arm
    - Controlling several robot limbs
- The researcher (you) need to define boundaries to the search-space
- The algorithm must find optimal behaviour in the defined search-space

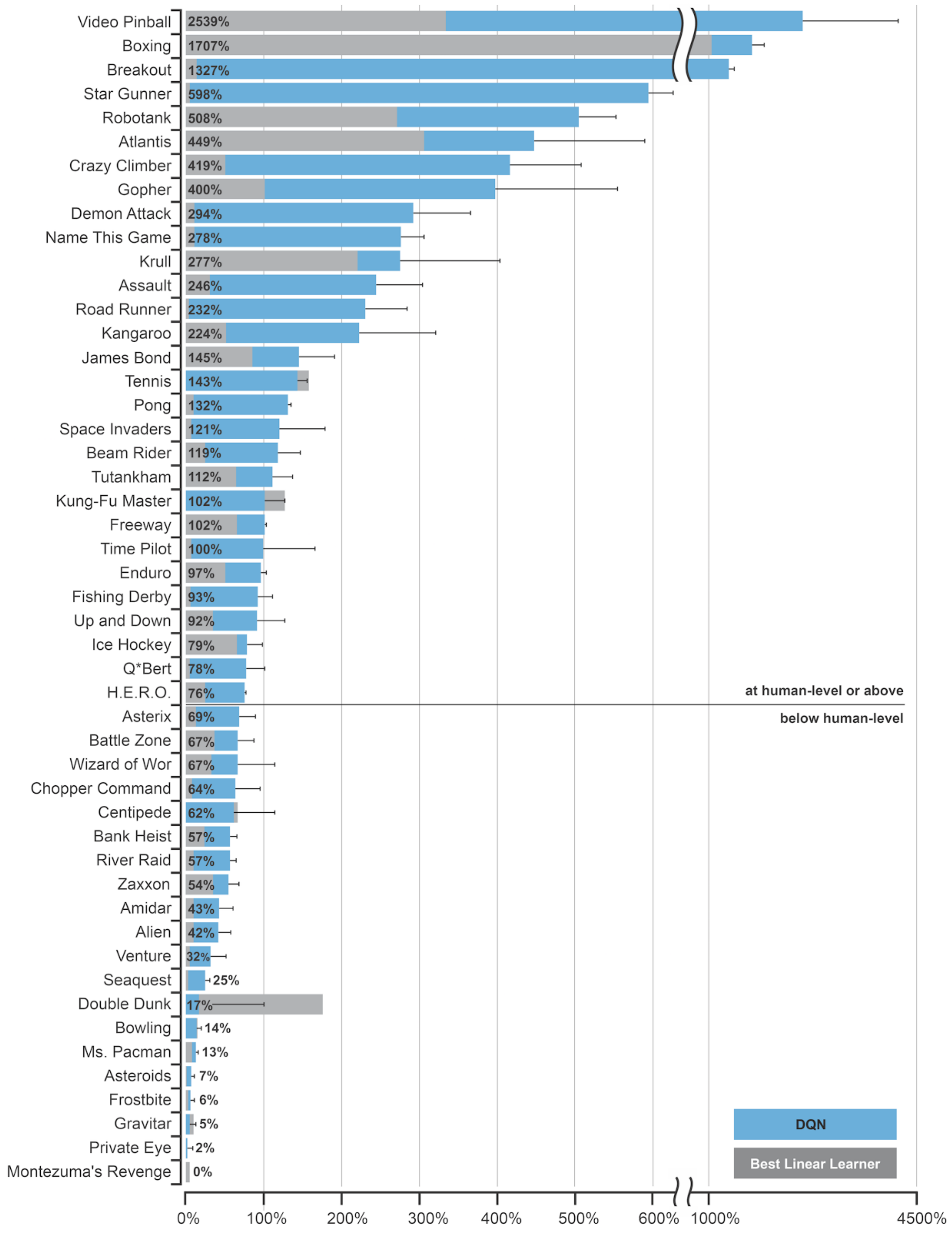


Examples

- Boston Dynamics
- RoboSchool
- PyBullet
- MuJoco

Atari 2600

- Various tasks
- Mastered by DeepMind with Deep Q-Networks
- Later "perfected" (and still ongoing) with various improvements



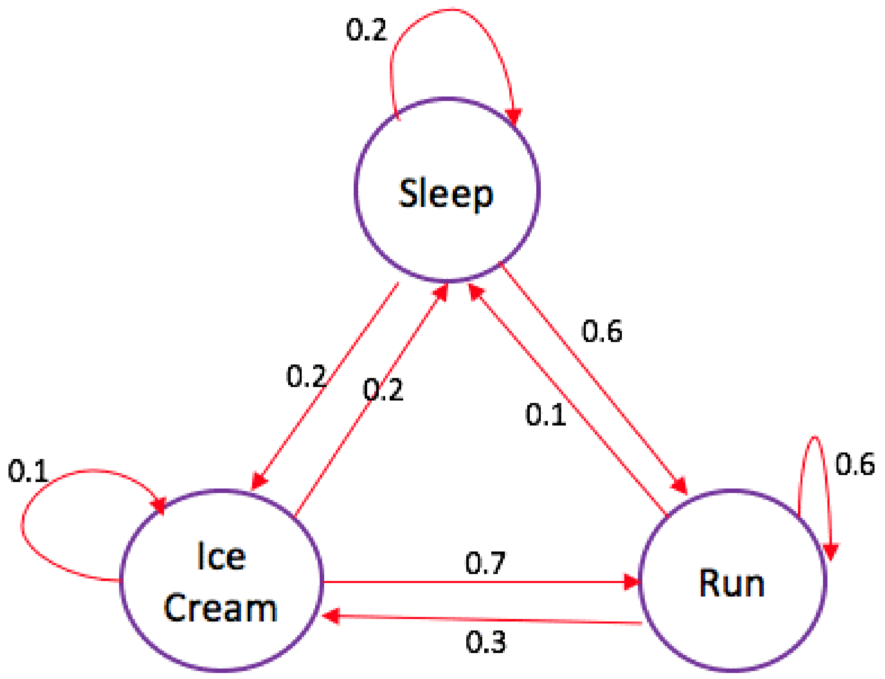
For all of the examples above.

How can we express learning of agents in a mathematical way?

Part 3: Markov Decision Processes

Makrov Process

- Also known as a Markov Chain is a tuple of <$\mathcal{S}, P$>
- $\mathcal{S}$ is a (finite) set of states
- $\mathcal{P}$ is a state transition probability matrix $\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s]$



$$\mathcal{P} = \mathit{from} \begin{array}{c} \mathit{to} \\ \begin{bmatrix} \mathcal{P}_{11} & \dots & \mathcal{P}_{1n} \\ \vdots & & \\ \mathcal{P}_{n1} & \dots & \mathcal{P}_{nn} \end{bmatrix} \end{array}$$

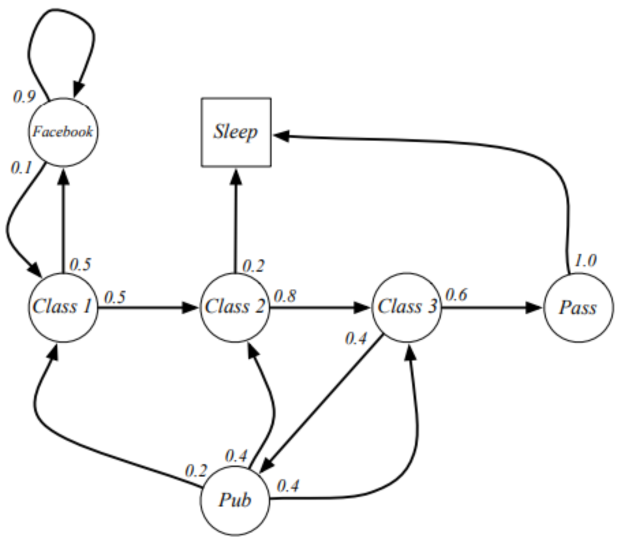
```
In [22]:
```

```python
import pandas as pd
# Sleep = 0
# Run = 1
# Ice Cream = 2
df = pd.DataFrame([    ["", "Sleep", "Run", "Ice Cream"],
     ["Sleep", 0.2, 0.6, 0.2],
     ["Run", 0.1, 0.6, 0.3],
     ["Ice Cream", 0.2, 0.7, 0.1]
])
df
```

```
Out[22]:
```

	0	1	2	3
0		Sleep	Run	Ice Cream
1	Sleep	0.2	0.6	0.2
2	Run	0.1	0.6	0.3
3	Ice Cream	0.2	0.7	0.1

Another example of a Markov Process

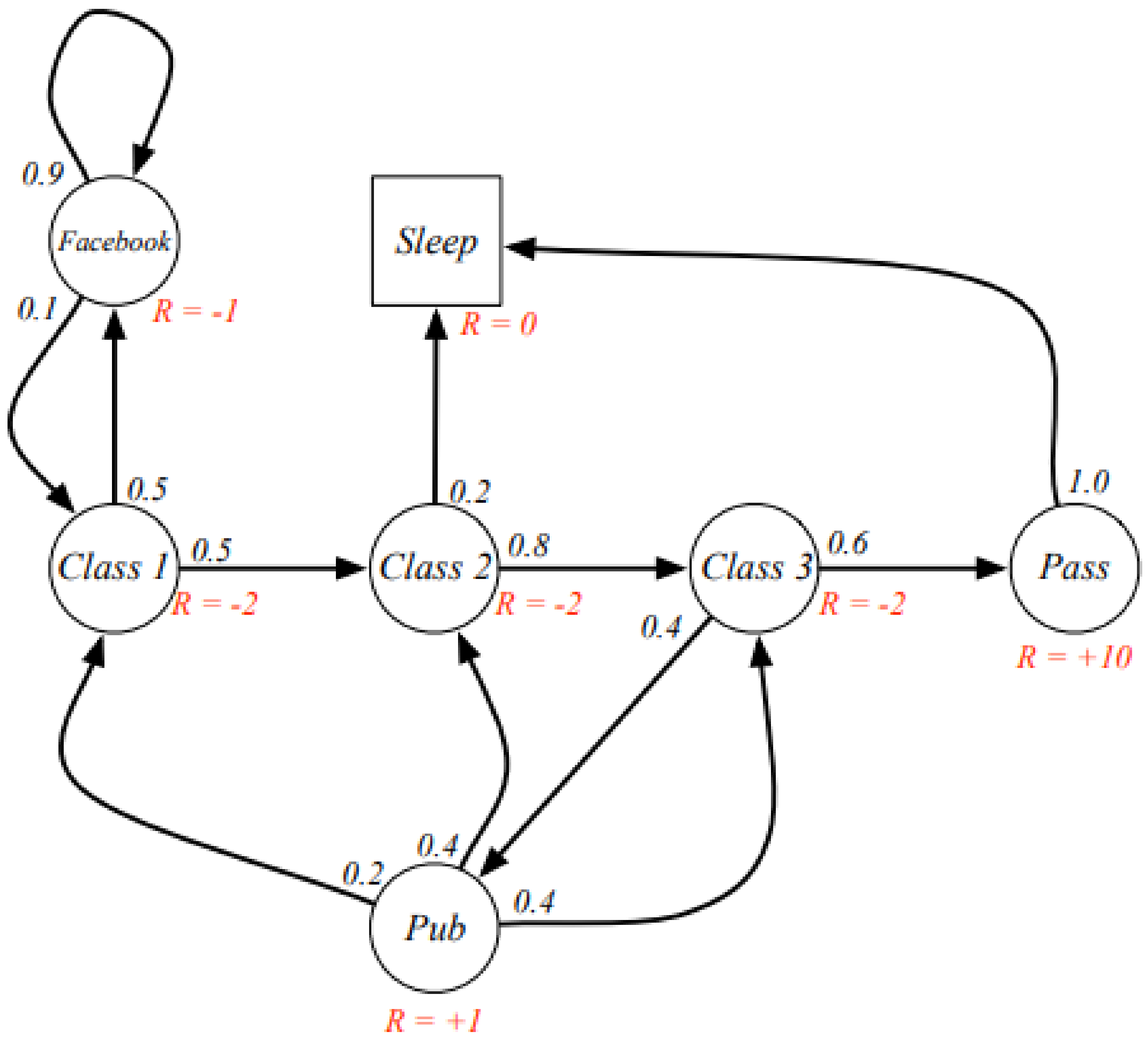


```
In [ ]:
```

Markov Reward Process

- Extends the markov process to account for rewards
- Defined by the tuple $< \mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma >$.

- $\mathcal{S}$ is a (finite) set of states
- $\mathcal{P}$ is a state transition probability matrix
- $\mathcal{R}$ is a reward function, $R_s = \mathbb{E}[R_{t+1} | S_t = s]$
- γ is the discount factor, $0 \leq \gamma \leq 1$.



Returns and Episodes

- **Epsisode:** Finite sequence of steps
- **Returns:** Total reward from time-step t until T

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T$$

- The problem with the above definition is that its possible to have continous tasks where $T = \infty$

- We can expand this notion of return to be discounted
  - G_t - Now its the total discounted reward from time-step t.

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

where $0 \leq \gamma \leq 1$.

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$$

including the possibility of $T = \infty$ or $\gamma = 1$ (but not both)

- Lower γ values penalize the future

In [ ]:

```
Show example on board:
Continous stream of rewards
Set Discount to be 0.5
R = 1
G_T = 1 + 1*0.5**1 + 1*0.5**2 + 1*0.5**3
G_T = 1 + 0.5 + 0.25+ 0.125 + .... + 0.0000000000000000001 (eventually)
```

In [23]:

```python
import numpy as np
def G(t, gamma, r_fn):
    v = 0
    for step in range(t):
        v += gamma**step * r_fn()
    return v









#
```

In [24]:

```
G(t=100, gamma=0.99, r_fn=lambda: 1)
```

Out[24]:

```
63.396765872677015
```

In [25]:

```
G(t=100000, gamma=0.99, r_fn=lambda: 1)
```

Out[25]:

```
99.99999999999925
```

In [26]:

```
G(t=100000, gamma=0.25, r_fn=lambda: 1)
```

Out[26]:

```
1.3333333333333333
```

In [27]:

```
import random
rng = random.Random()
rng.seed(42)
G(t=100, gamma=0.99, r_fn=lambda: rng.random())
```

Out[27]:

```
29.809815542048387
```

Not both because this would lead to G_t = inf. iF T = inf but gamma < 1. gamma would eventually be 0 if Gamma = 1 but T = finite, it would just be a number

Value function

- $V(s)$ denotes the state-value function
- A measurement for the "goodness" of a state

So, how do we know the value of a state?

- We have already found that there is some **return**, the discounted future reward

Bellman Equation for MRP

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots R_T \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \ldots R_T) \\ &= R_{t+1} + G_t \end{aligned}$$

$$\begin{aligned} V(s) &= \mathbb{E}[G_t | S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots | S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \ldots) | S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma G_{t+1} | S_t = s], \text{(Law of total expectation)} \\ &= \mathbb{E}[R_{t+1} + \gamma V(S_{t+1}) | S_t = s] \end{aligned}$$

- The reason we have to use expectation is that:
  - The policy can be stochastic
  - The transition matrix can be stochastic
  - Aka, there *can* be randomness in the system

The importance of the Bellman equations is that they let us express values of states as values of other states. This means that if we know the value of s_{t+1}, we can very easily calculate the value of s_t. This opens a lot of doors for iterative approaches for calculating the value for each state, since if we know the value of the next state, we can know the value of the current state. The most important things to remember here are the numbered equations.

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

$$V(s) = \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} V(s')$$

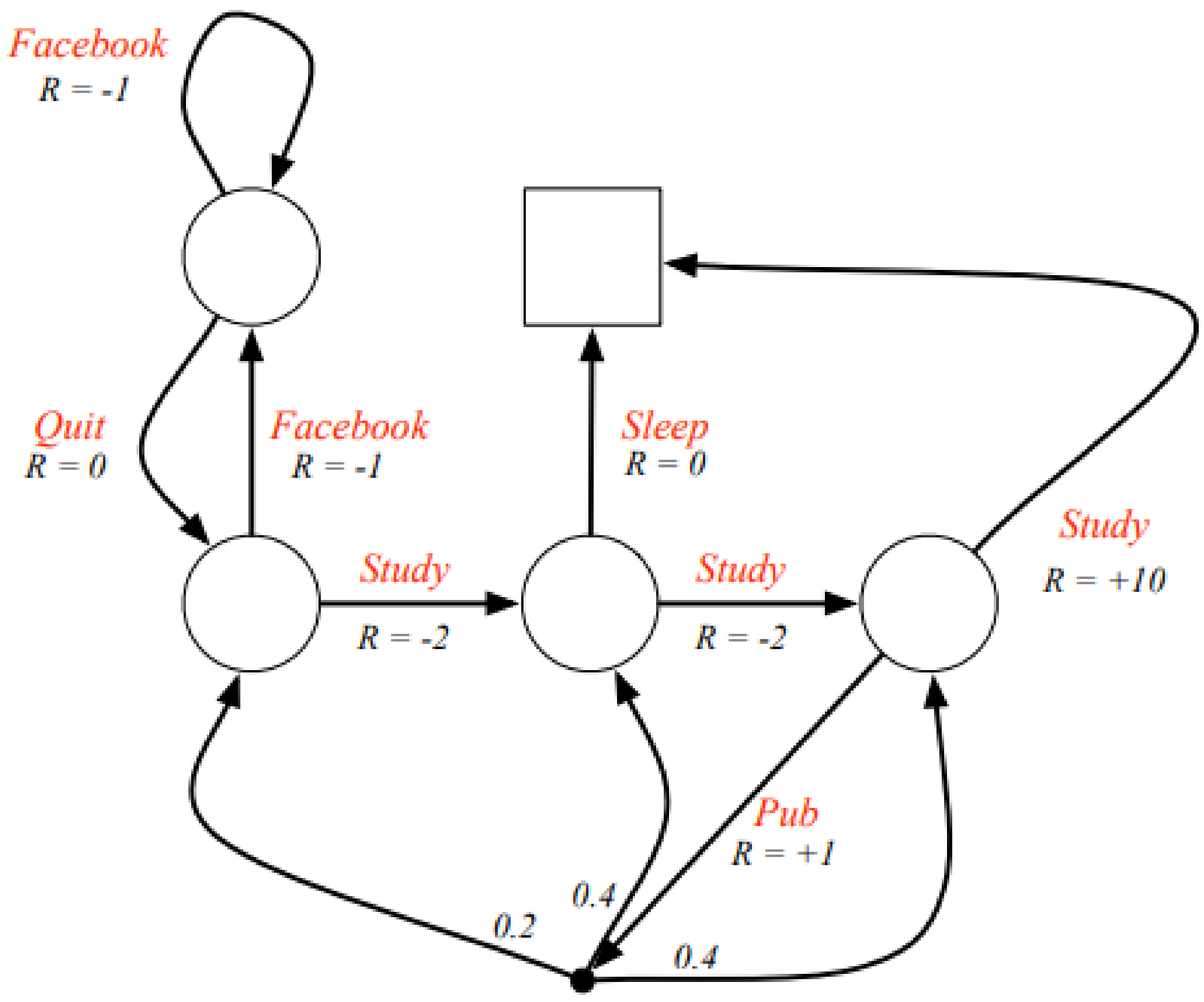
The decomposed value function **(The first)** is also called the Bellman Equation for Markov Reward Processes. This function can be visualized in a node graph **Figure**. Starting in state s leads to the value v(s). Being in the state s we have certain probability Pss' to end up in the next state s'. In this particular case we have two possible next states. To obtain the value v(s) we must sum up the values v(s') of the possible next states weighted by the probabilities Pss' and add the immediate reward from being in state s. This yields **The last equation**, which is nothing else than **The first** if we execute the expectation operator E in the equation.

Markov Decision Processes

- Mathematical formulation of reinforcement learning problems

MDP is defined as tuples <$\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma$>.

- $\mathcal{S}$ is a (finite) set of states
- $\mathcal{A}$: set of actions that the agent can take
- $\mathcal{P}$ is a state transition probability matrix \ $\mathcal{P}_{ss'}^{a} = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$
- $\mathcal{R}$ is a reward function $\mathcal{R}_s^a = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$
- γ is the discount factor, $0 \leq \gamma \leq 1$.



Transition Function Derivation

1. The transition function is the probability of landing in s' with reward r given action a in state s

$$\mathcal{P}(s', r|s, a) = \mathbb{P}[S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a]$$

2. We find the state-transition by taking the the probability of landing in state s' with reward r given that we took action a in state s. We account for all possible rewards in the transition.

$$\mathcal{P}^a_{ss'} = \mathcal{P}(s'|s, a) = \mathbb{P}[S_{t+1} = s'|S_t = s, A_t = a] = \sum_{r\in\mathcal{R}} \mathcal{P}(s', r|s, a)$$

3. The *expected* reward r of doing action a in state s. We can expand the *expecency* by summing the probability of all possible $r \in \mathcal{R}$ and $s' \in \mathcal{S}$

$$\mathcal{R}^a_s = \mathcal{R}(s, a) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a] = \sum_{r\in\mathcal{R}} r \sum_{s'\in\mathcal{S}} \mathcal{P}(s', r|s, a)$$

Policy

The brain of a RL algorithm (Agent)

A policy π is a distribution over actions given states:

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

- State to action mapping
- The agent's behaviour
- Depends only on current state **(but not the history)**
- The policy is stationary $$A_t \sim \pi(\cdot|S_t), \forall t \geq 0$$
- The policy can be deterministic or stochastic

Deterministic: $\pi(s) = a$

Stochastic: $\pi(a|s) = \mathbb{P}_\pi[\mathcal{A} = a|\mathcal{S} = s]$

- Optimal policies are denoted π^*
- A policy can be *any* behaviour:
  - $\pi_0(a|s) = rng(0, 5)$
  - $\pi_1(a|s) = Q(s, a)$

Just assume the policy to be stochastic!

State-value and Action-value functions

State-Value

- State-value function is now the *expected return starting from s following policy π

$$V_{\pi}(s) = \mathbb{E}_{\pi}[G_t | S_t = s]$$

$$\begin{aligned} V_{\pi}(s) &= \mathbb{E}_{\pi}[G_t | S_t = s] \\ &= \dots \\ &= \mathbb{E}_{\pi}[R_{t+1} + \gamma V_{\pi}(s_{t+1}) | S_t = s] \end{aligned}$$

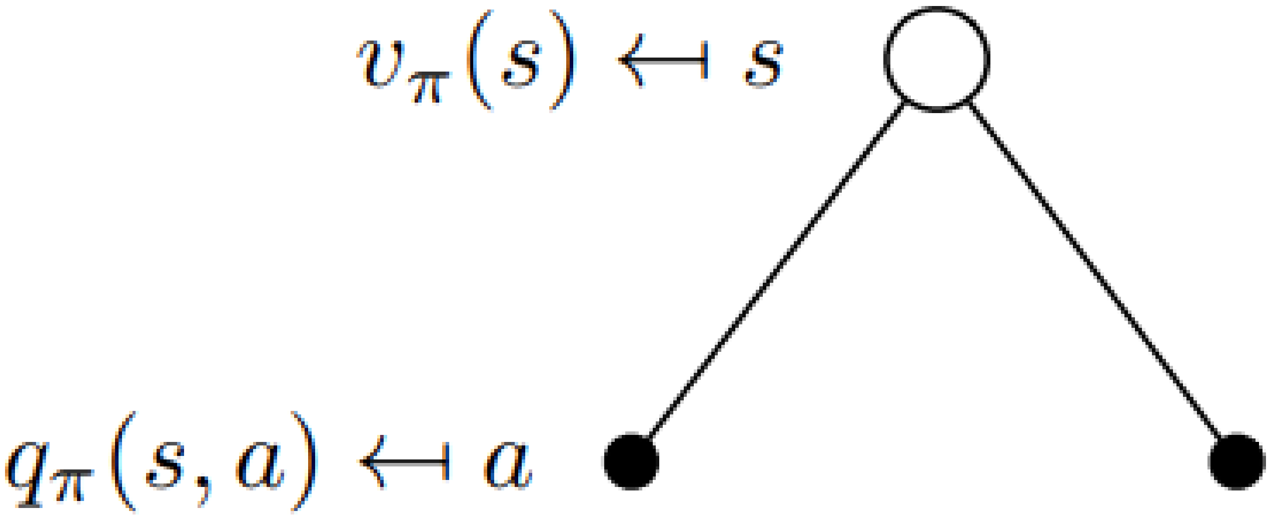
Action-Value

- Action-value function tells us how good is it to take a particular action in a particular state.

$$\begin{aligned} Q_{\pi}(s,a) &= \mathbb{E}_{\pi}[G_t | S_t = s, A_t = a] \\ &= \dots \\ &= \mathbb{E}_{\pi}[R_{t+1} + \gamma Q_{\pi}(s_{t+1}, a_{t+1}) | S_t = s] \end{aligned}$$

Advantage Function

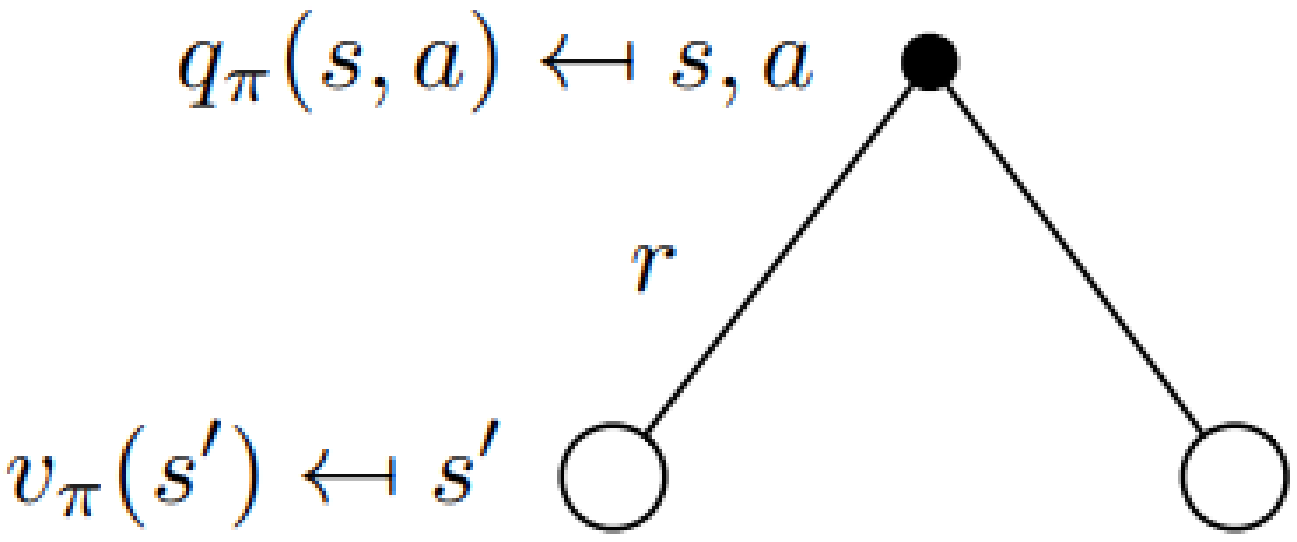
$$A_{\pi}(s,a) = Q_{\pi}(s,a) - V_{\pi}(s)$$


$v_\pi(s) \leftarrow\!\dashv s$
$q_\pi(s,a) \leftarrow\!\dashv a$

State-value function as weighted sum of action-values.

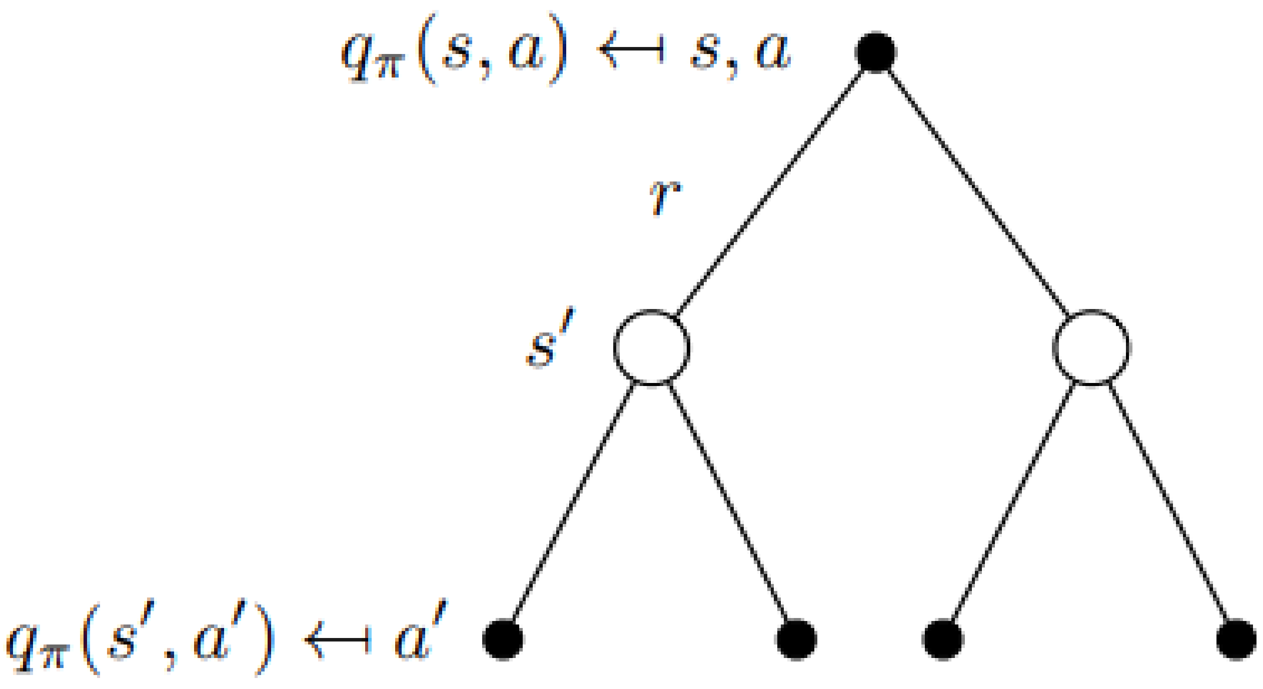
$$V_{\pi}(s) = \sum_{a \in A} \pi(a|s) Q_{\pi}(s,a)$$

Relationship between Action and State values



$$Q_\pi(s,a) = R_s^a + \gamma \sum_{s' in S} \mathcal{P}_{ss'} V_\pi(s')$$

If we plug in the recurisve bits



$$Q_\pi(s,a) = R_s^a + \gamma \sum_{s' in S} \mathcal{P}_{ss'} \sum_{a' \in A} \pi(a'|s') Q_\pi(s',a')$$

Optimal State-Value

$$\begin{aligned} V_*(s) &= \max_{\pi} V_{\pi}(s) \\ \pi_* &= \arg\max_{\pi} V_{\pi}(s) \\ V_{\pi_*}(s) &= V_*(s) \\ V_*(s) &= \max_{a \in \mathcal{A}}(R_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a V_*(s')) \end{aligned}$$

Optimal Action-Value

$$\begin{aligned} Q_*(s,a) &= \max_{\pi} Q_{\pi}(s,a) \\ \pi_* &= \arg\max_{\pi} Q_{\pi}(s,a) \\ Q_{\pi_*}(s,a) &= Q_*(s,a) \\ Q_*(s,a) &= R_s^a + \gamma \sum_{s' in S} \mathcal{P}_{ss'} \max_{a' \in \mathcal{A}} Q_*(s',a') \end{aligned}$$

The most important topic of interest in deep reinforcement learning is finding the optimal action-value function q. *Finding q* means that the agent knows exactly the quality of an action in any given state. Furthermore the agent can decide upon the quality which action must be taken. Lets define that q* means. The best possible action-value function is the one that follows the policy that maximizes the action-values:

1. Value function is the Expected future reward following policy PI from state s. The action value is the same taking into account action as well
2. The optimal V* is the the policy that yields highest possible V
3. We can find the optimal policy pi* by selecting the value function that yields highest values
4. and we can write it as V pi* is the optimal V function
5. The advantage function is the difference between state and value function

Markov Decision Processes in Reinforcement Learning

- A mathematical framework for modeling reinforcement learning problems
- Usually
  - Transition matrix is not available
  - Rewards are not available (we have to create a reward function)

Challenges

- How to model the reward?
- How to learn behaviour (policy) based on experience?
- How to explore?

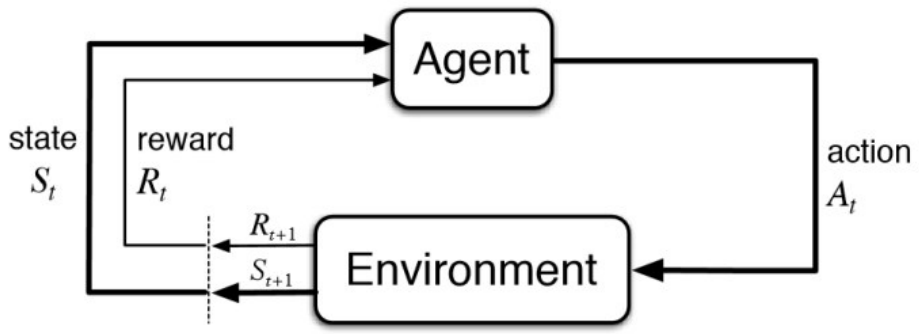
Part 3: Reinforcement Learning

- Its a agent-envrionment interface
- Agent selects actions and the environment respond, continually

Action Any decision we want to learn how to make

State Knowledge that might be useful in making actions

- While we have tried to separate agent and envrionment, these are often tightly linked together.
  - For example. You can form a agent and environment from a robot arm



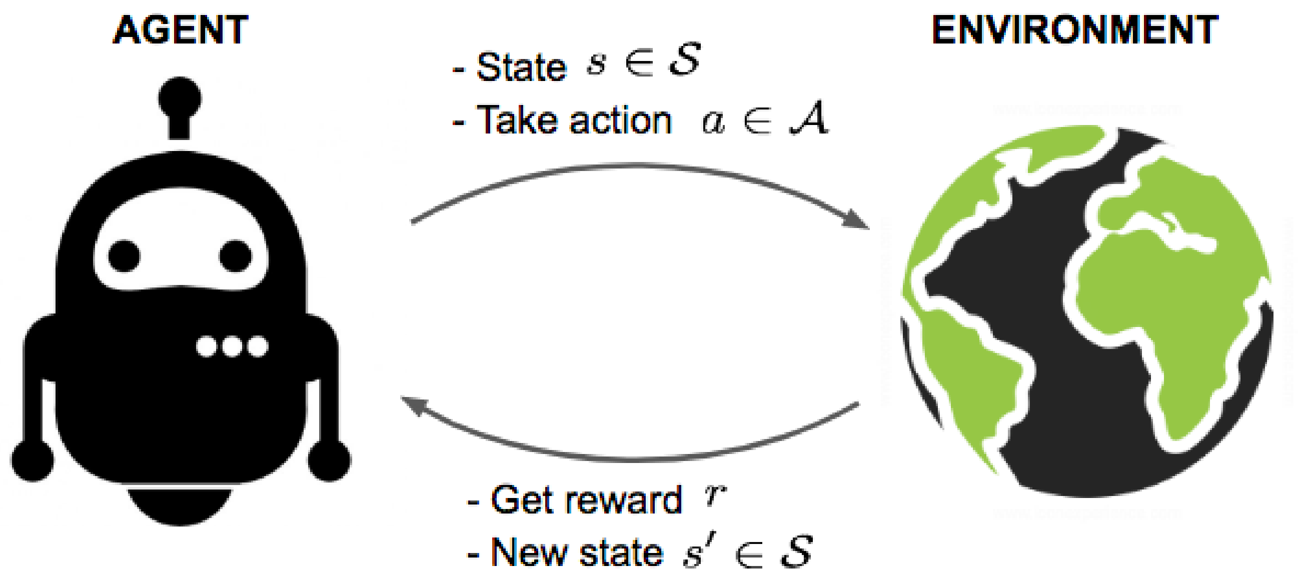
State S_t: The current picture or situation in the environment

Action A_t: The action the agent takes

Environment (P**)**: How the environment reacts to the action

Reward R_t: The feedack that the environment "sends" to the agent

We can write this as a tuple (S, A, P, R)


AGENT
ENVIRONMENT
- State $s \in \mathcal{S}$
- Take action $a \in \mathcal{A}$
- Get reward r
- New state $s' \in \mathcal{S}$

The agent ought to take actions so as to maximize cumulative rewards. In reality, the scenario could be a bot playing a game to achieve high scores, or a robot trying to complete physical tasks with physical items; and not just limited to these.

Dataset: *None* Continous interaction with the environment (You can, however collect off-policy experiences, which is some form of dataset)

Goal: Learn the agent a good strategy from trial and error (Minimize error, Learn fast)

Examples: Game playing, Robot Control

Algorithms: Q-Learning, Policy Optimization

A few notable components in RL

- **Agent**
  - Actions
  - Action-Space
    - Discrete
    - Continous
  - Exploration/Exploitation
    - ϵ-greedy
  - Policy
    - Random
    - Algorithm
  - On-Policy
  - Off-Policy

- **Environment**
  - Model
  - State-Space
    - Stochastic/Deterministic
  - Fully-observable
  - Partially Observable

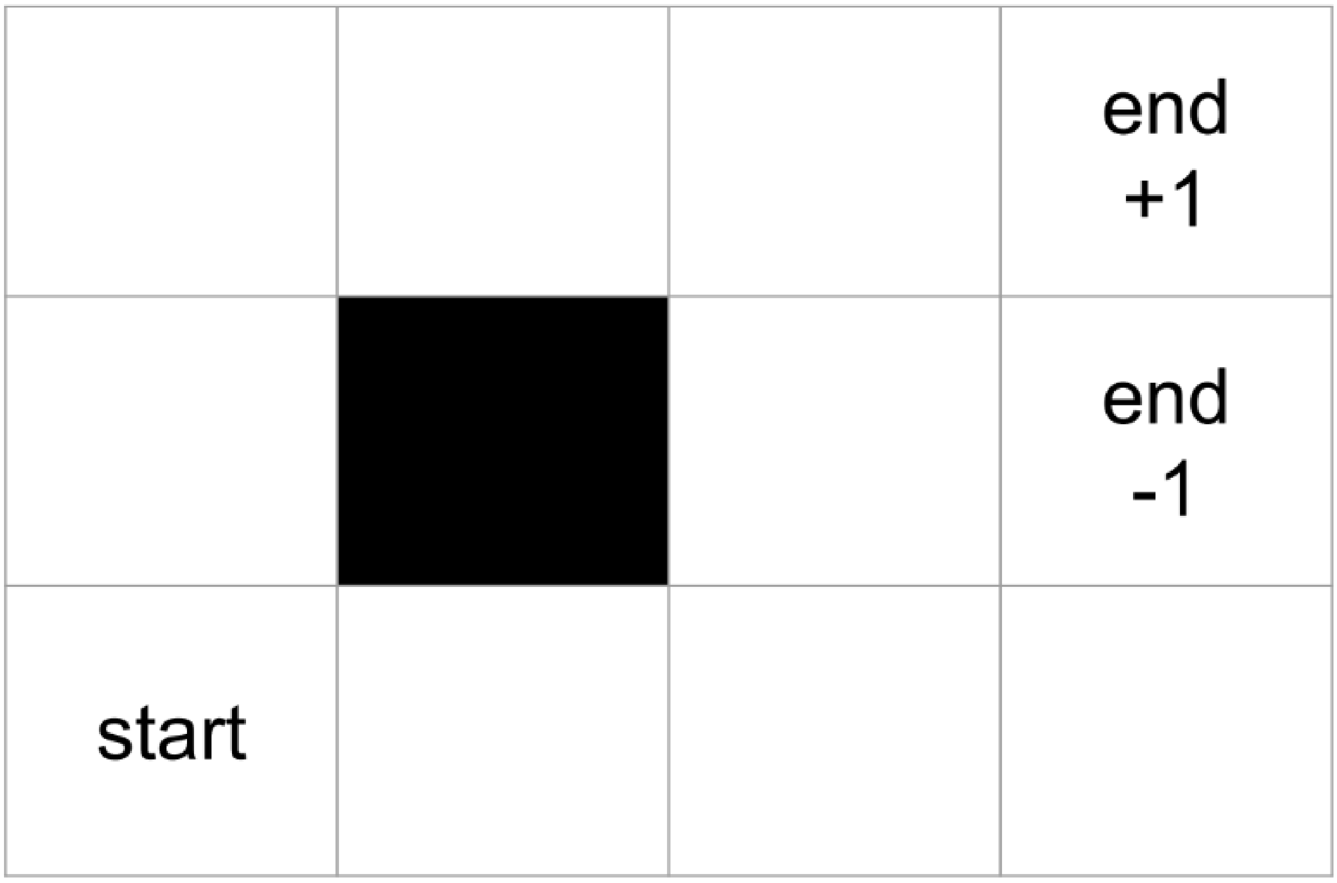
```
In [ ]:
```

```

```

Environment

- Where the agent "lives" and make actions
- The environment describes the physics/dynamics that the agent must live by
- Can be **stochastic** or **deterministic**
- Can be **fully-observable** or **partially-observable**
- Most of the time, we can say that
  - Fully observable = MDP
  - Partially observable = POMDP
- Each state is associated with a value function $V(s)$

Example



Model

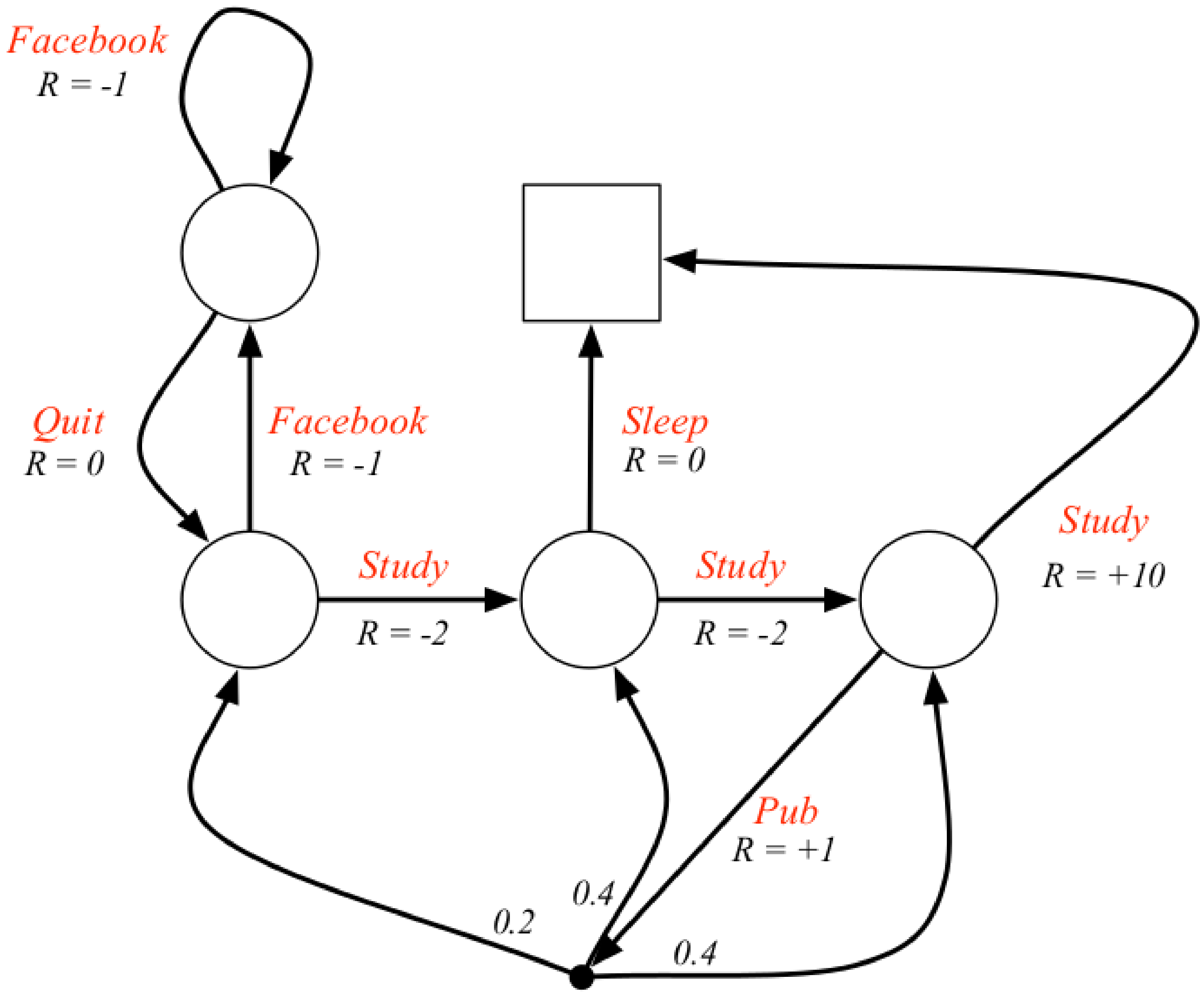
A **model** is a **description** of how the **environment** function A model has two parts:

- P - Transition Probability Matrix
- R - Reward Function

Transition

A transition happens when the agent is in **state s** and takes **action a** and arrive at next **state s'** with **reward r**

Defined as the tuple: **(s, a, s', r)**.



Agent

Description: Software that interacts with an environment

Goal Maximise some notion of cumulative reward.

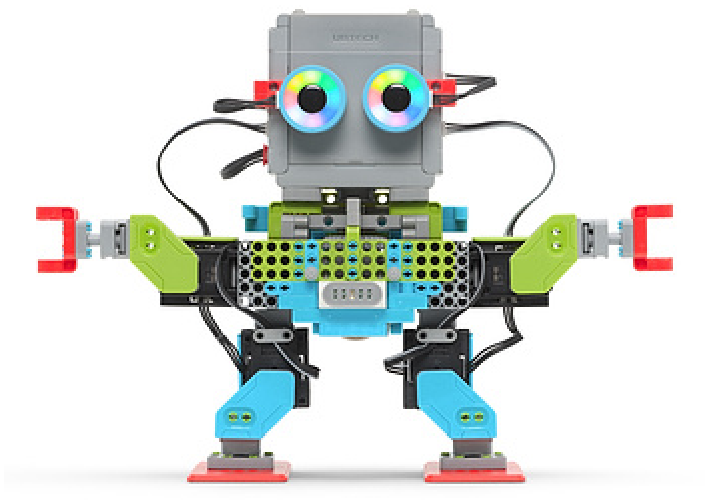
Example:

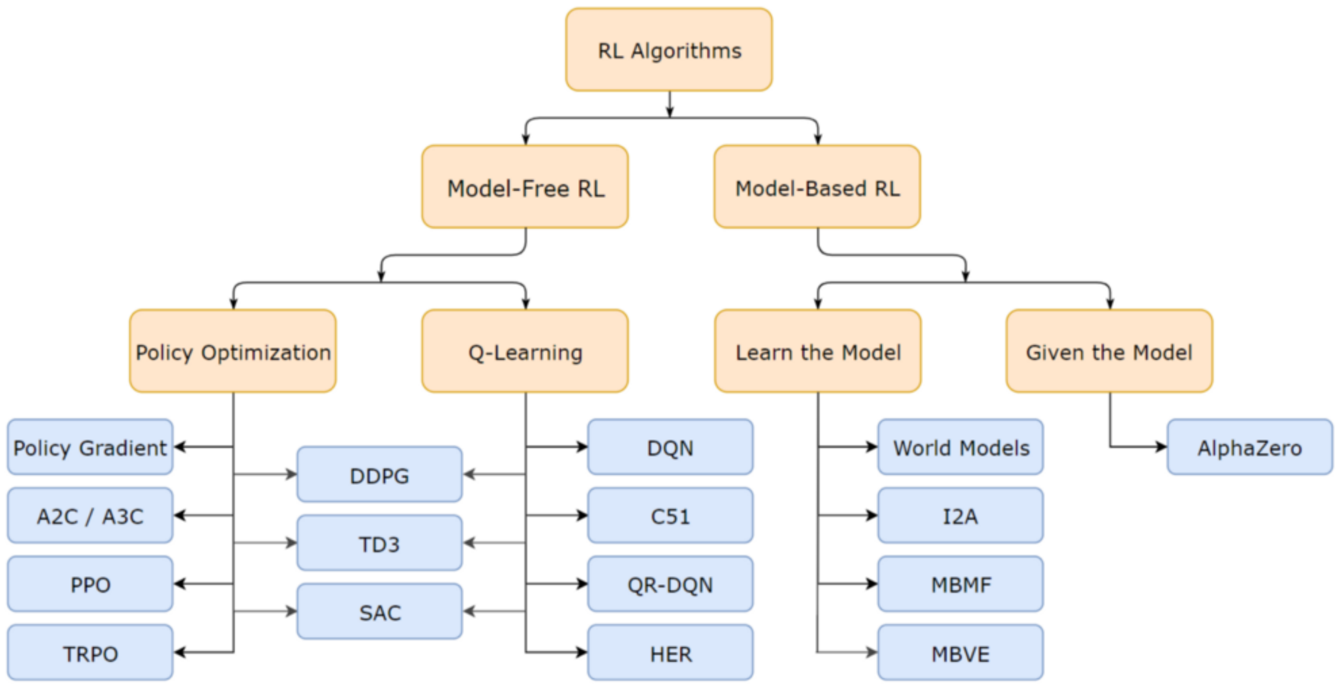
- The player of a game
- Part of a system, robot arm
- The whole robot
- Any part that can **act**

Exploration: A agent can explore the environment to better learn its dynamics

- ϵ-greedy
- decaying ϵ-greedy
- There are many ways of doing this....

Exploitation: The agent can exploit what it learns during interaction (learning)





Goals and Rewards

Goal to maximise reward

Reward a simple scalar value: $R_t \in \mathbb{R}$

Examples:

Maze: -1 for all steps, +1 for goal

Collect: 0 for all steps +1 for pickup, perhaps -1 for messing up

Chess: +1 for winning, 0 for rest.

Beware for rewarding subgoals as it often leads to sub-optimal policies

Rewards

- The reward function R is important!
- Its what fuels learning in most algorithms
- Sometimes you dont have access to design the reward, but sometimes you do.

In [28]:

```
from IPython.display import IFrame
IFrame('https://www.youtube.com/embed/tlOIHko8ySg?t=20&rel=0&amp;controls=1&amp;
showinfo=0', 700, 700)
```

Out[28]:

Dynamic Programming

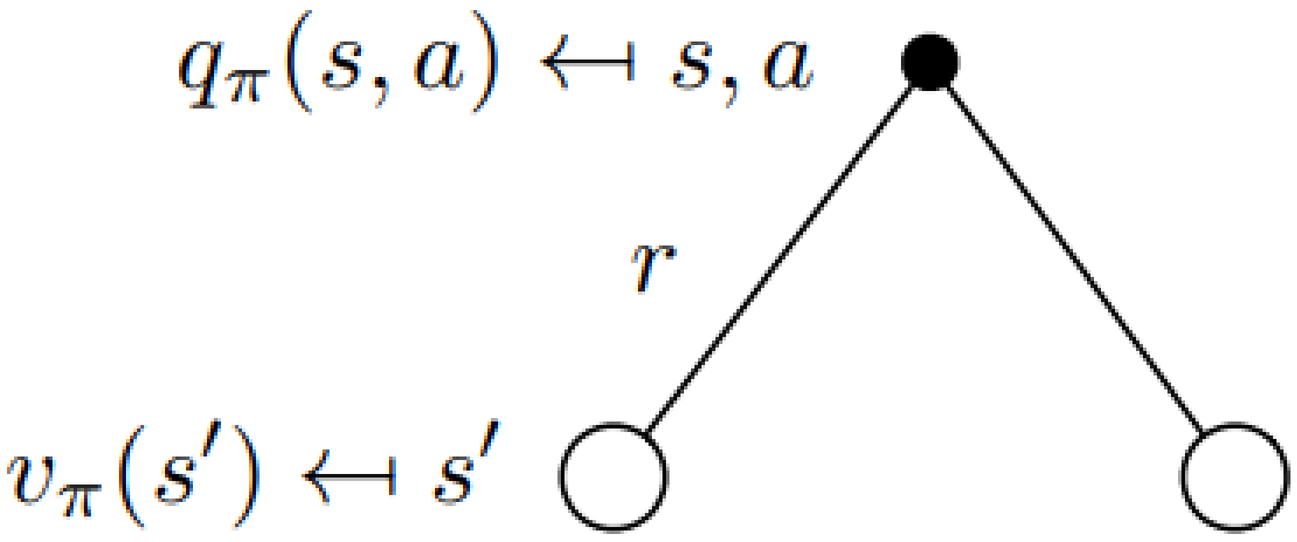
Description

- Utilizes value-functions to find good policies
- **perfect information envrionments** meaning that you need a model of the environment

General Algorithm: You start with an arbitrary policy (No defined behaviour)

1. Policy evaluation
   - Compute V_π from π
2. Policy Improvement
   - Compute improved π from V_π
3. Repeat

Repeat the process until convergence



Monte Carlo Methods

- In the other end of the scope, we have have Monte Carlo Methods
- Learns from **complete** episodes.
- Collects a trajectory of $S_1, A_1, R_2 \ldots S_T$ to compute $G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$

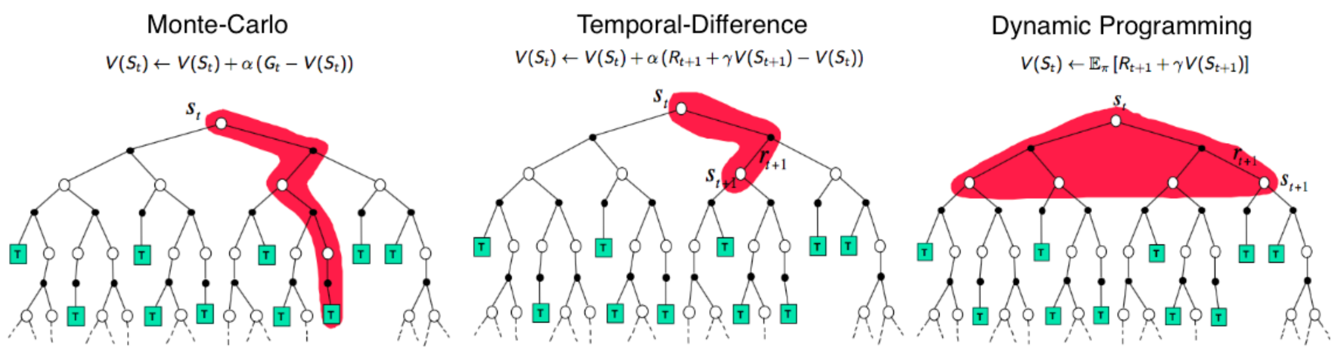
Recall that: $V(s) = \mathbb{E}[G_t | S_t = s]$

The empirical mean return for state s is:

$$V(s) = \frac{\sum_{t=1}^{T} 1[S_t=s]G_t}{\sum_{t=1}^{T} 1[S_t=s]}$$

The empirical action-value

$$Q(s,a) = \frac{\sum_{t=1}^{T} 1[S_t=s,A_t=a]G_t}{\sum_{t=1}^{T} 1[S_t=s,A_t=a]}$$


Monte-Carlo
$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t))$
Temporal-Difference
$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$
Dynamic Programming
$V(S_t) \leftarrow \mathbb{E}_\pi[R_{t+1} + \gamma V(S_{t+1})]$
s_t
r_{t+1}
s_{t+1}
T

Temporal-Difference Learning

- Model-free, like Monte-Carlo Methods but can learn from incomplete episodes like Dynamic Programming

- Aka, something in-between

- Learns by **bootstrapping**. Updates from previous estimates

- Key idea is to **update** the value function towards an estimated return.

$$\begin{aligned}
V(S_t) &\leftarrow (1-\alpha)V(S_t) + \alpha G_t \qquad \text{Multiplication} \\
V(S_t) &\leftarrow V(S_t) + \alpha(G_t - V(S_t)) \qquad \text{Expand G} \\
V(S_t) &\leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))
\end{aligned}$$

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

Q-Learning: Off-Policy TD Control

$$Q(s_t,a_t) \leftarrow \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{\underbrace{R_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \max_a Q(s_{t+1},a)}_{\text{TD-Target}} - Q(s_t,a_t)}^{\text{learned value}} \right)$$

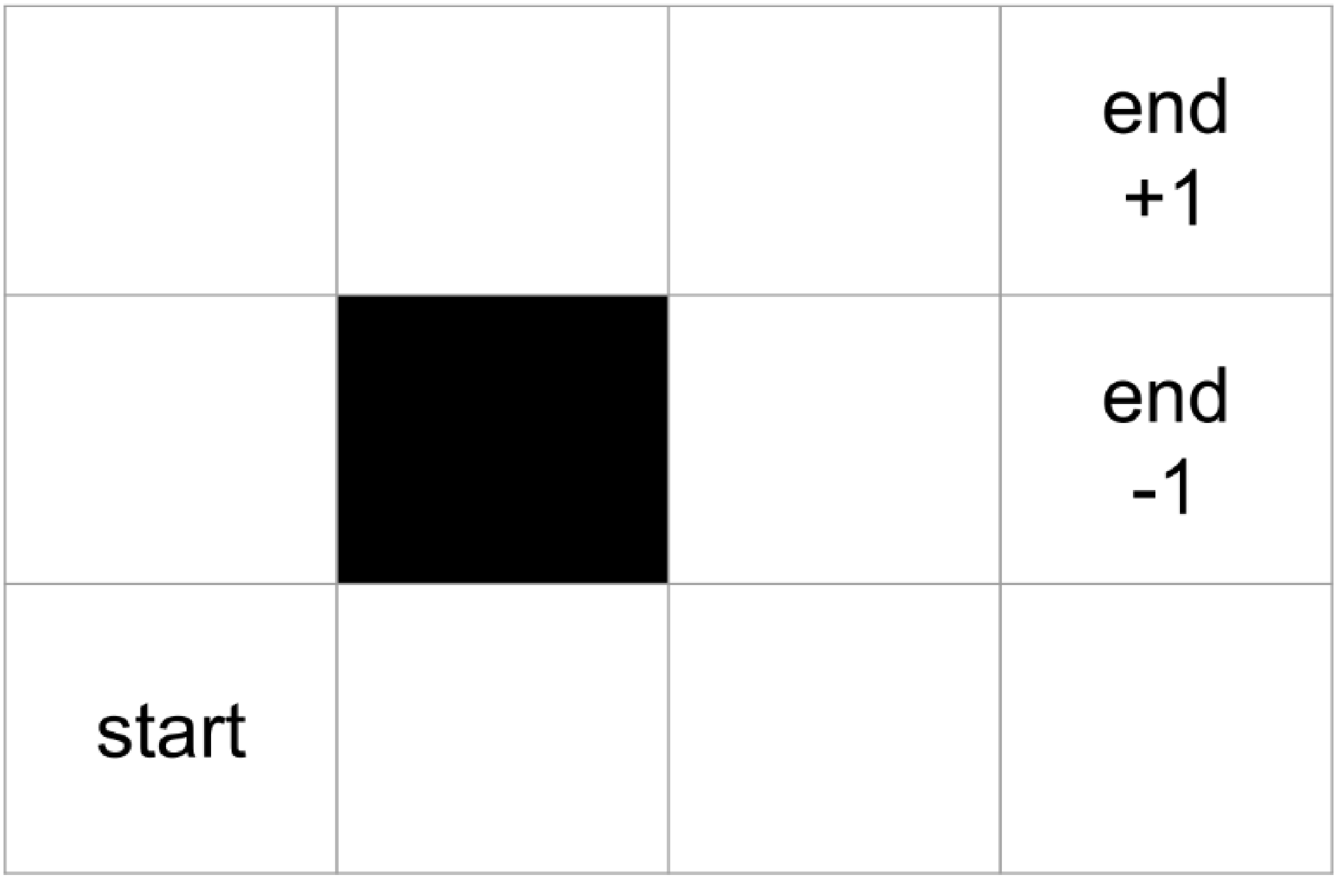
Reinforcement Loop

1. Agent decides next action based on its most recent observation
2. Environment transition to next state and yields a feedback signal
3. Agent process feedback and adapt behaviour
4. Agent observes latest sensory information

A Simple RL Loop

In [ ]:

```
agent = SomeAlgorithm()
env = SomeEnvironment()

for i in EPISODES:
    state = env.reset()
    while not terminal:
        action = agent.predict(state)
        state_1, reward, terminal, _ = env.step(action)
        agent.learn(state, action, state_1, reward, terminal)  # The learning al
gorithm
        state = state_1
```



Gridworld (Maze) environment

Control: Agent can move in either direction (not diagonals). One Timestep = one square

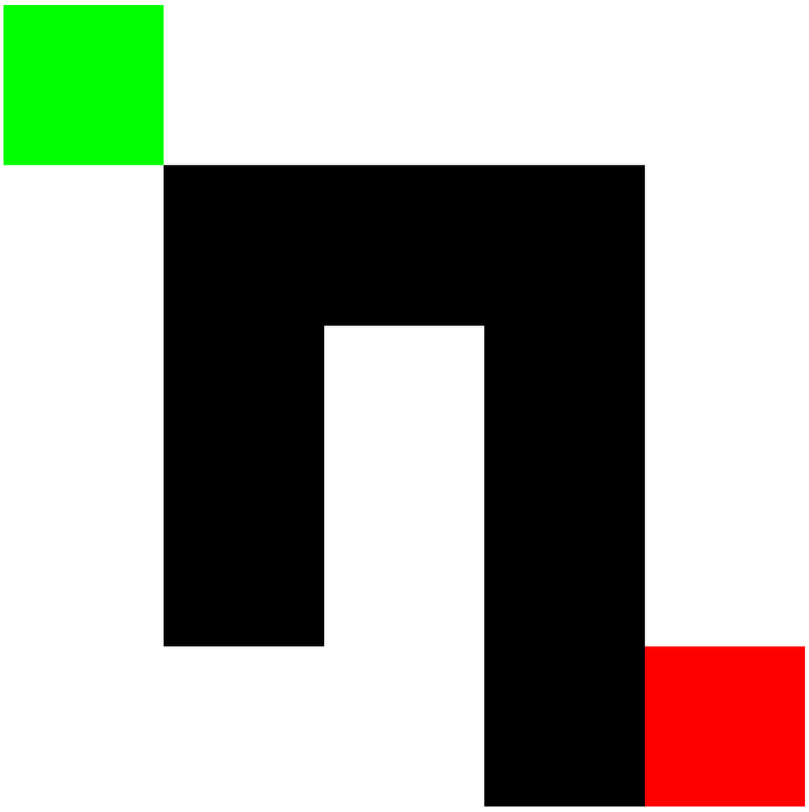
Reward: -0.1 for non terminal steps, +1 for terminal steps

Goal: Enter the red square

In [29]:

```
#!pip install git+https://github.com/cair/gym-maze.git --user --upgrade --force-
reinstall  --no-cache-dir > /dev/null
import gym
import gym_maze
env_original = gym.make("Maze-5x5-NormalMaze-v0")  # substitute environment's na
me
random.seed(42)  # Need this to have static env
env = gym_maze.StateArrayWrapper(env_original)
PIL.Image.fromarray(gym_maze.StateRGBWrapper(env_original).reset())\
.resize((400, 400) )\
.transpose(PIL.Image.FLIP_LEFT_RIGHT)
```

Out[29]:



In [30]:

```
random.seed(42)# Need this to have static env
print("Env\n", env.reset())
print("Player", env.env.env.player)
print("Target", env.env.env.target)
```

```
Env
 [[2 0 0 0 0]
 [0 1 1 1 0]
 [0 1 0 1 0]
 [0 1 0 1 0]
 [0 0 0 1 3]]
Player (0, 0)
Target (4, 4)
```

In [31]:

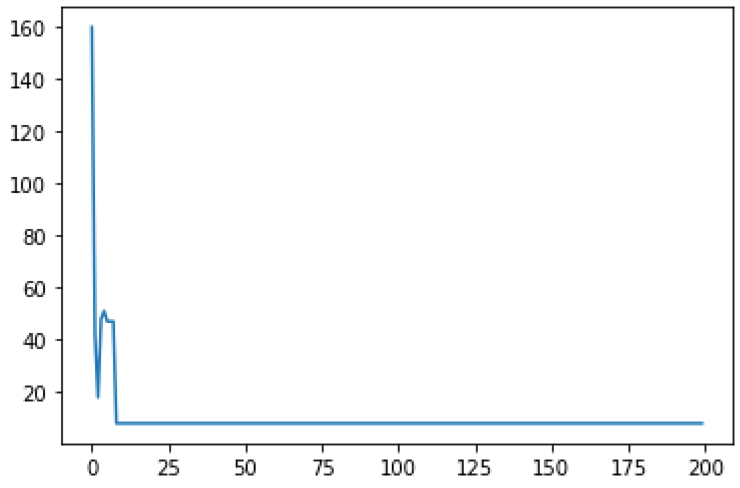
```
_env = env.env.env # MazeGame Instance

Q = np.zeros(
    (_env.width, _env.height, env.env.action_space),
    dtype=np.float32
)
e_start = 1.0 # Epsilon start
e_dec = -0.01 # Epsilon decay
lr = 0.99 # Learning rate
gam = 0.95 # Discount factor
def predict(s, steps):
    if (e_dec * steps) > random.uniform(0, 1):
        return random.randint(0, 4-1)
    else:
        return np.argmax(Q[s[0], s[1]])

def bootstrap(s, s1, a, r):
    x, y = s
    x1, y1 = s1
    Q[x, y, a] += lr * \
    (r + gam * np.max(Q[x1, y1]) - Q[x, y, a])

EPISODES = 200
MAX_STEPS = 200
```

In [32]:

```
import matplotlib.pyplot as plt
results = []
for i in range(EPISODES):
    random.seed(42)# NEED THIS FOR STATIC ENV!!!!
    _, steps, t = env.reset(), 0, False
    s = _env.player
    while not t and steps <= MAX_STEPS:
        a = predict(s, i)
        _, r, t, _ = env.step(a)
        s1 = _env.player
        bootstrap(s, s1, a, r)  # The learning algorithm
        s = s1
        steps += 1
    results.append(steps)
plt.plot(np.arange(len(results)), results)
```

Out[32]:

```
[<matplotlib.lines.Line2D at 0x7f06bc547a60>]
```



In [ ]:

```
def bootstrap(x, y, x1, y1, a, r):
    Q[x, y, a] += lr * (r + gam * np.max(Q[x1, y1]) - Q[x, y, a])
```

- Off-Policy
- Epsilon-greedy
- Learning rate
- Discount Factor

Algorithms (incomplete list)

Algorithm	Description	Model	Policy	Action Space	State Space	Operator
Monte Carlo	Every visit to Monte Carlo	Model-Free	Off-policy	Discrete	Discrete	Sample-means
Q-learning	State-action-reward-state	Model-Free	Off-policy	Discrete	Discrete	Q-value
SARSA	State-action-reward-state-action	Model-Free	On-policy	Discrete	Discrete	Q-value
Q-learning - Lambda	State-action-reward-state with eligibility traces	Model-Free	Off-policy	Discrete	Discrete	Q-value
SARSA - Lambda	State-action-reward-state-action with eligibility traces	Model-Free	On-policy	Discrete	Discrete	Q-value
DQN	Deep Q Network	Model-Free	Off-policy	Discrete	Continuous	Q-value
DDPG	Deep Deterministic Policy Gradient	Model-Free	Off-policy	Continuous	Continuous	Q-value
A3C	Asynchronous Advantage Actor-Critic Algorithm	Model-Free	On-policy	Continuous	Continuous	Advantage
NAF	Q-Learning with Normalized Advantage Functions	Model-Free	Off-policy	Continuous	Continuous	Advantage
TRPO	Trust Region Policy Optimization	Model-Free	On-policy	Continuous	Continuous	Advantage
PPO	Proximal Policy Optimization	Model-Free	On-policy	Continuous	Continuous	Advantage
TD3	Twin Delayed Deep Deterministic Policy Gradient	Model-Free	Off-policy	Continuous	Continuous	Q-value
SAC	Soft Actor-Critic	Model-Free	Off-policy	Continuous	Continuous	Advantage

Assignment

In this assignment you will build a compatible gym environment. You will then implement Q-Learning and SARSA and perform tests on the environment you created.

Task 1 - Questions

1. Is Q-Learning is a off-policy algorithm?. Explain why (not)
2. Is Sarsa a off-policy algorithm?. Explain why (not)
3. What is a policy?
4. What is a state-value function?
5. What is a action-value function?
6. What is a MDP and why do we use them in RL
7. What is the benefit of using discounted returns?
8. Do you need a model the environment in Dynamic Programming
9. Can the environment be continous in Monte Carlo Methods
10. How do we denote optimal functions in reinforcement learning? For example, a optimal policy.

Task 1 - Environment

Create the environment of your choice. Example of such environments is the GridWorld environment. You can here do whatever you want, but* I recommend that you create a **deterministic** environment. Use the template provided (class Environment)

In [33]:

```python
class Environment:

    def __init__(self):
        # Define objects here, such as map.. etc
        pass

    def step(self, action):
        # Code the game logic here
        return self.render()

    def reset(self):
        # Code a reset function here
        return self.render()

    def render(self, mode='human', close=False):
        pass
```

Task 2 - Agents

Implement SARSA and Q-Learning, use the supplied template(s) for the implementation. You can choose to use the same class for both agents if you desire to do so.

Bonus You can also implement any other algorithm in the domain of (Deep) Reinforcement Learning.

In [34]:

```python
class Agent:

    def __init__(self, lr, discount):
        pass

    def learn(self, s, a, r, s1, t):
        pass

    def predict(s):
        pass

class SarsaAgent(Agent):

    def __init__(self, lr, discount):
        super().__init__(lr, discount)

class QLearningAgent(Agent):
    def __init__(self, lr, discount):
        super().__init__(lr, discount)
```

Task 3

Run your your agents against your environment and report back the results. Did the agents learn the optimal policy (behaviour). Why, why not?

-

-

-

-

-

-

-

In [35]:

```python
import matplotlib.pyplot as plt
results = []  # Record results in this list
EPISODES = 200 # Set number of episodes here
MAX_STEPS = 200 # Max number of steps per ep
LR = None  # Set the learning rate
GAMMA = None # Set the discount factor

agents = [
    SarsaAgent(LR, GAMMA),
    QLearningAgent(LR, GAMMA)
]
env_args = () # Arguments to the environment
```

In [ ]:

```
for agent in agents:
    env = Environment(*env_args)

    for i in range(EPISODES):

        s, step, t = env.reset(), 0, False

        while not t and step <= MAX_STEPS:
            a = agent.predict(s)
            s1, r, t, _ = env.step(a)
            agent.learn(s, a, r, s1, t)
            s = s1
            step += 1

        results.append(step)

    plt.plot(np.arange(len(results)), results)
```

Example Implementation

Shortest-Path Memory Agent

In [37]:

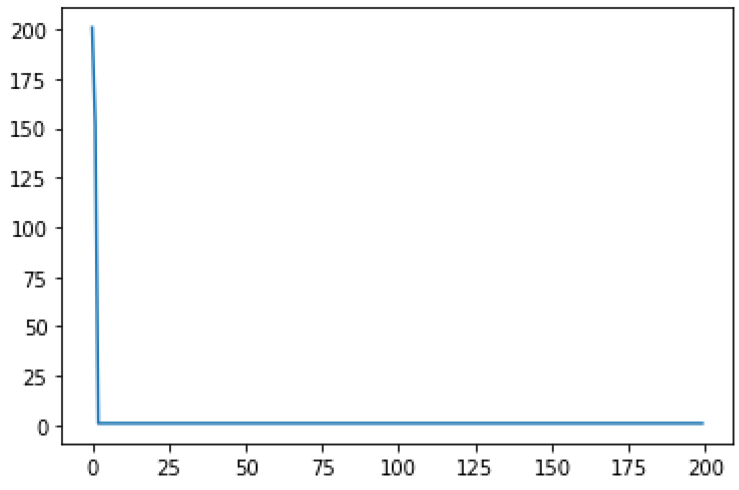
```
import numpy as np
import random
class Environment:
    class Actions:
        id_up, id_down, id_left, id_right = (0, 1, 2, 3)
        UP = (0, -1)
        DOWN = (0, 1)
        LEFT = (-1, 0)
        RIGHT = (1, 0)

    def __init__(self, width, height, obstacles, goals):
        # Define objects here, such as map.. etc
        self.width = width
        self.height = height
        self.obstacles = obstacles
        self.goals = goals
        self.goal = None # Initialized in reset()
        self.state = None  # Initialized in reset()

        self.reset()

    def step(self, action):
        # Code the game logic here

        CAN_RIGHT = self.state[0] < self.width
        CAN_LEFT = self.state[0] > 0
        CAN_UP = self.state[1] > 0
        CAN_DOWN = self.state[1] < self.height
        if any([CAN_RIGHT, CAN_LEFT, CAN_UP, CAN_DOWN]):

            if action == Environment.Actions.id_up and CAN_UP:
                self.state = tuple(np.array(self.state) + np.array(Environment.A
ctions.UP))
            elif action == Environment.Actions.id_down and CAN_DOWN:
                self.state = tuple(np.array(self.state) + np.array(Environment.A
ctions.DOWN))
            elif action == Environment.Actions.id_left and CAN_LEFT:
                self.state = tuple(np.array(self.state) + np.array(Environment.A
ctions.LEFT))
            elif action == Environment.Actions.id_right and CAN_RIGHT:
                self.state = tuple(np.array(self.state) + np.array(Environment.A
ctions.RIGHT))

        reward = 0
        terminal = False

        if self.state in self.obstacles:
            terminal = True
            reward = -1.0
        elif self.state == self.goal:
            terminal = True
            reward = 1.0

        # Set state to none when terminal
        # Return: s1, r, t, {}
        return self.render(), reward, terminal, {}

    def reset(self):
        # Code a reset function here

        while self.state is None:
            spawn_pos = (random.randint(0, self.width), random.randint(0, self.h
eight))
            if spawn_pos not in self.obstacles:
                self.state = spawn_pos

        self.goal = random.choice(self.goals)

        return self.render()

    def render(self, mode='human', close=False):
        return self.state
```

In [38]:

```
class MyAgent:
    # Agent that search random for the goal, when found he stores the location a
nd goes there greedily

    def __init__(self, lr, discount):
        self.goals = set()
        self.selected_goal = None
        self.my_pos = None

    def learn(self, s, a, r, s1, t):
        if t:
            self.goals.add(s1)


    def predict(self, s):
        if len(self.goals) == 0:
            # No known goals
            return random.randint(0, 3)

        if self.selected_goal is None:
            self.selected_goal = random.choice(list(self.goals))

        dx = self.selected_goal[0] - s[0]
        if dx > 0:
            return Environment.Actions.id_left
        elif dx < 0:
            return Environment.Actions.id_right

        dy = self.selected_goal[1] - s[1]
        if dy > 0:
            return Environment.Actions.id_up
        elif dy < 0:
            return Environment.Actions.id_down
```

In [41]:

```
import matplotlib.pyplot as plt
results = []  # Record results in this list
EPISODES = 200 # Set number of episodes here
MAX_STEPS = 200 # Max number of steps per ep

agents = [
    #SarsaAgent(LR, GAMMA),
    #QLearningAgent(LR, GAMMA)
    MyAgent(0, 0)
]

for agent in agents:
    env = Environment(10, 10, [(1, 1), (9, 9, (8, 8))], [
        (2, 2),
        (8, 3),
        #(5, 5),
        #(2, 10)
    ])

    for i in range(EPISODES):

        s, step, t = env.reset(), 0, False

        while not t and step <= MAX_STEPS:
            a = agent.predict(s)
            s1, r, t, _ = env.step(a)

            agent.learn(s, a, r, s1, t)
            s = s1
            step += 1

        results.append(step)

    plt.plot(np.arange(len(results)), results)
```



200
175
150
125
100
75
50
25
0
0
25
50
75
100
125
150
175
200