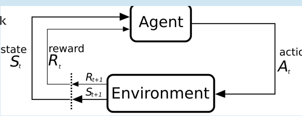
1. Introduction & Lecture 1

AI: How do we build systems that solve tasks for which humans need intelligence?
ML: The answer to the AI question: Methods, algos, data structures that *learn* to solve such tasks leveraging data.
Big Data: Encompasses methods to handle and understand large data sets - composed of Data Science (how to analyse the world in a data driven way) and Data Engineering (how to collect, store, process, clean and maintain large data).
AI usually refers to an intelligent system that embodies an entire practical system (self driving car) while ML is focused on the algo. ML puts engineering efforts on building a system to solve a problem rather than engineering a system that solves problems
Reinforcement Learning: Solves problems of control - choosing the best / optimal action at the right time.
The core of RL: Agent interacts with environment and gains knowledge, exploring and receiving *rewards*. Actions change the state of the env, we choose actions for max long-term reward.
RL is the framework by which we can teach AI to learn how to interact on its own



1.1 Probability

Probability of all events sums to 1.
Given a probability matrix: P on the left:

0.1	0.4	*Rain*
0.25	0.25	*Sun*
Windy	*Calm*	

P(Rain) = 0.5 (marginalization)
P(Rain|Windy) = 0.1/0.35 = 0.29 (conditioning)
Should I bring an umbrella if its windy (cost: -1) - getting caught in the rain without an umbrella is cost: -3. (control)
Cost of umbrella = -1. Cost of no umbrella: -3*0.29 = -0.87.
Bayes Theorem: p(AB) = p(A|B)*p(B). So: p(A|B) = p(B|A)*p(A)/p(B)
Boolean Logic: if A is true B is true. B is true - doesn't tell us abt A
Bayes Logic: if A is true, B is true. B is true, so A is more likely

Plausibilities: (encodes prior info) 3 criteria applies to them:
1) The degrees of plausibility are represented by real numbers
2) These numbers must be based on the rules of common sense
3) Plausibilities are self-consistent when found via different methods, they take into account all available data, and discovering the plausibilities is reproducible.

Posterior: p(Parameters|Data)
= p(Data|Parameters) * p(Parameters) / p(Data)
Likelihood / Evidence: p(Data|Parameters)
Prior: p(Parameters) - our prior belief over possible param values
We can visualise structure of dependencies between random variables with directed edges of causation events (with edges having P[target|source] from source to target nodes and nodes with pure source nodes reflecting prior probabilities P[source]).

Benefits of Bayes Logic: we can make inferences based on uncertain information. Probabilities can represent degrees of belief. We can make principled decisions - is P(A)*Cost(A) > P(¬A)*Cost(¬A)
Bayesian Decision Theory: $a^* \in \arg\max_a \mathbb{E}[r(a)] = \arg\max_a \sum_{j=1}^{M} r(s_j, a)p(s_j)$
Make optimal decisions by maximizing a*. a represents a decision/action, s is the state
We do this in RL, Optimal Control (Robots).
RL involves learning optimal control of an "a priori" unknown system with unknown rewards - only experience teaches us

1.2 Markov Processes

A Markov Process is a tuple <S, P>. S is a set of states and Pss′ is a state transition probability matrix P[St+1 = s′|st = s]
Markov Property: A state st is Markov iff P[st+1|st] = P[st+1|s1...st] - that is, the next state is only determined by the current state.
Square states are terminal, round are transient (can start there).
Σs ' Pss' = 1 (of course, as all transition probabilities sum to 1)
Stationary Markov Chains: if the P[st+1|st] don't depend on t, (and so only on origin/dest state) the Markov chain is stationary.
Markov Reward Process: A Markov chain that emits rewards. Formed of a tuple (S, P, R, γ). S and Pss′ are as before.
Rs = E[rt+1|St = s] is an expected immediate reward collected at time step t+1 upon departing state s.
γ ∈ [0, 1] is a discount factor.
Return: The return Rt is the total discounted reward from time-step t. Rt = rt+1 + γrt+2 + · · · = Σ^∞_k=0 γ^k rt+k+1
The γ factor is how we discount the current value of future reward. γ close to 0 leads to myopic evaluation (as looking at the return we ONLY care about the immediate reward - what's in the current state). γ close to 1 is farsighted evaluation - we factor lots of distant rewards.
Most Markov Chains are discounted, ie γ < 1:
- It avoids infinite returns in cyclic processes
- Helps take into account uncertainty about the future
- Immediate rewards can help us gain future ones (e.g: finance)
- Human & Animal decision making favours immediate rewards
- Sometimes γ = 1 is if all sequences terminate, or when they're equally long

State Value Function v(s): The expected return R starting from state s at time t; v(s) = E[Rt|St = s] by definition *(expectation notation)*. Which equals: E[rt+1 + γrt+2 + γ 2 rt+3 + . . . |St = s]
= E [rt+1 + γ(rt+2 + γrt+3 + . . .)|St = s]
= E [rt+1 + γRt+1|St = s] = E [rt+1 + γv (St+1)|St = s]
Which represents two terms - the immediate reward rt+1 and the discounted reward of the successor state γv(St+1).
Sum notation: v(s)=Rs+γ Σ_s '∈S P_ss' v (s′) (n eqs, 1 for each state)
Vector notation: v = R + γPv (v is an n-dimensional vector)
The vector equation is directly solvable;
v = (1-γP)^-1R. But matrix inversion is expensive O(n^3); direct solutions are only good for small MRPs. Dynamic Programming, Monte-Carlo Evaluation, Temporal-Difference learning faster.

Policy: A policy Πt(a,s) = P [At = a|St = s] is a conditional probability distribution (non deterministic) to execute an action a ∈ A given that one is in state s ∈ S at time t.
The policy is only deterministic if for a given state only 1 a is possible: Π(a,s) = 1 and Π(a ' ,s) = 0, ∀a≠ a '. Shorthand: Πt(s) = a

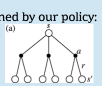
1.3) Markov Decision Processes - like MRP *but with an action*

S: State Space, A: Action Space, γ ∈ [0, 1] is discount factor,
P_ss′ ^a: Transition probability p(s_t+1|s_t,a_t). (rt+1=r(st+1,st,at))
Π: Policy. Stochastic: a ~ pΠ(a|s).
pΠ(a|s) = Π(a|s) ≡ Π(a,s) (probability)
Deterministic: a = Π(s) (indicator function)

1. State Value Function for MDPs: (A)

$$V^\pi(s) = \mathbb{E}_\pi[R_t|S_t = s] = E[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|S_t = s]$$

Where Rt is a discounted total return and rt+k+1 is imed. reward
Backup Diagrams: Traces paths from state s to successor states s′ given we chose action a. We can transfer state value info back to a state s from successor states s′ - for a back operation to find the right Value Func:
1) Find the probability of taking action a (governed by our policy: P [a|s] = Π(a,s)).
2) The probability of transition to s′ (transition probability: P[s′|s,a] = Pss′^a)
3) The reward of each s′ ((s,a,s′) = Rss′^a)
4) The value of s′ (v(s′) recursive definition).



$$\begin{aligned} &= \mathbb{E}_\pi[R_t|S_t{=}s] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|S_t{=}s\right] \\ &= \mathbb{E}_\pi\left[r_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|S_t{=}s\right] \\ &= \mathbb{E}[r_{t+1}|S_t{=}s] + \gamma\mathbb{E}\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|S_t{=}s\right] \end{aligned}$$

$$\mathrm{E}[r_{t+1}|S_t = s] = \sum_{a\in\mathcal{A}} P[a|s]\left(\sum_{s'\in\mathcal{S}} P[s'|s,a]\, r(s,a,s')\right)$$

$$\mathrm{E}\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|S_t{=}s\right] = \sum_{a\in\mathcal{A}} P[a|s]\left(\sum_{s'\in\mathcal{S}} P[s'|s,a]\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}\right)$$

$$= \sum_{a\in\mathcal{A}}\pi(a,s)\left(\sum_{s'\in\mathcal{S}}\mathcal{P}^a_{ss'}\left(\mathcal{R}^a_{ss'} + \gamma\mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|S_{t+1} = s'\right]\right)\right)$$

$$= \sum_{a\in\mathcal{A}}\pi(s,a)\sum_{s'\in\mathcal{S}}\mathcal{P}^a_{ss'}\left(\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\right)$$

We got a version of Bellman's Equation - has one unique solution. We can simply apply bellman's equations again and again to obtain better estimates for V1(s), v2(s)... (known as iterative policy eval) with a thresh for a cutoff point.

2. Policy Evaluation:

```
Input π, the policy to be evaluated
Initialize V(s) = 0, for all s ∈ S+
Repeat
    Δ ← 0
    For each s ∈ S:
        v ← V(s)
        V(s) ← Σ_a π(s,a) Σ_s' P^a_ss' [R^a_ss' + γV(s')]
        Δ ← max(Δ, |v − V(s)|)
until Δ < θ (a small positive number)
Output V ≈ V^π
```

State-Action Value function: (How good it is to be in a given state and take a given action when you follow a policy Π):

$$Q^\pi(s,a) = \mathbb{E}[R_t|S_t = s, A_t = a] = \mathbb{E}\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|S_t = s, A_t = a\right]$$

The relation between the state value function and the state-action value func is straightforward: $V^\pi(s) = \sum_{a\in\mathcal{A}}\pi(s,a)Q^\pi(s,a)$

A policy is better than or equal to another if its expected return is geq for all states: Π ≥ Π ' iff V Π(s) ≥ V Π ' (s) for all s ∈ S.
3. Optimality - Bellman's Equations: $V^*(s) = \max_\pi V^\pi(s), \forall s \in \mathcal{S}$
1. The Optimal Value Function:
Optimal VF without a specific policy:

$$\begin{aligned} V^*(s) &= \max_a \sum_{s'} P[s'|s,a]\left(r(s,a,s') + \gamma V^*(s')\right) \\ &= \max_a \sum_{s'} \mathcal{P}^a_{ss'}\left(\mathcal{R}^a_{ss'} + \gamma V^*(s')\right) \end{aligned}$$

Optimal State-Action VF: $Q^*(s,a) = \max_\pi Q^\pi(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}$

$$= \sum_{s'}\mathcal{P}^a_{ss'}\left(\mathcal{R}^a_{ss'} + \gamma\max_{a'} Q^*(s',a')\right)$$

Optimally solving the Bellman Optimality Equations gives 1 route to finding the optimal policy, to solve the RL problem. It's hard to solve because it's like an exhaustive search looking ahead at all possible traces and also relies on 3 assumptions: 1) we accurately know the env dynamics 2) We have the comp. resources to solve 3) the Markov property. Hence we often can only approximate.
Bellman Optimality Equations Convergence Theorem:
1. The Bellman Optimality Eqs have a unique soln
2. The values in iteration converge to that soln
Since the BOE is a fixed pt eqn:
1. If we start with some arb. VF we can apply the RHS of BOE to V to get a new VF: V′. Thus RHS of the BOE defines an operator K on the space of VFs.
2. BOE states the opt val func is a fixed point; V^Π* = K(V^Π*).
3. A simple process to find the fixed point K would be to begin with VF V0(s) = 0∀s. Apply K to it, to get V1: V1 <-- K(V0).
4. We can keep doing this. But that's just the iterative method.
5. This converges to a unique limit because K is a contraction; ie: given two VFs V1 and V2, applying K brings them closer: ||K(V1) - K(V2)||∞ ≤ γ||V1 - V2||∞, where V1 =/= V2 and norm ||..|| is the max norm (max over all possible states S)
6. This γ was the same discount factor from the MDP. With γ < 1 we must converge as our process brings points closer. BOEs and MDPs satisfy the Contracting Mapping Theorem which is why this works!

2.1) Dynamic Programming - computes optimal policies for MDPs

- Assume: MDP is finite & we have a perfect model of the env: we know the transition and reward functions
- DP breaks down complicated opti problems by breaking them into solving simpler optimization subproblems recursively.
  - If subproblems overlap, we can reuse our memoized results for them. The space of subproblems needs to be small.
- If a subproblem can be nested recursively inside larger ones and DP is applicable, the relationship between those smaller subproblems and the original problem is called the Bellman Eq.

Policy Improvement:
Computing the value function helps us find better policies:
- If we have the VF for an arb. determini. policy Π, we know how good it is to follow Π from s, aka: V^Π(s).
- To determine if it'd be better to change to the new policy Π′ where Π′(s) = a′ and Π(s) = a we can select the new action for state s alone else use the old policy. This is Q^Π(s, a′) by defin.
- So: test if Q^Π(s,a′) > V^Π(s). If so, the newer policy is better.

Policy Improvement Theorem:
Let Π and Π′ be two deterministic policies s.t; ∀s ∈ S
Theorem: Q^Π(s,a′) >= V^Π(s) means Π′ is as good or better than Π: V^Π′(s) >= V^Π(s), ∀s ∈ S
Proof: 1) Consider a deterministic policy Π(s) = a.
2) We're at *least* as good with Π′(s) = argmax(a∈A) Q^Π(s,a)
3) So setup a new Policy: Q^Π(s, Π′(s)) = max(a∈A) Q^Π(s,a) >= Q^Π(s, Π(s)) = V^Π(s)
4) Thus it improves the value function: V^Π′(s) >= V^Π(s) bc:

$$\begin{aligned} V^\pi(s) &\le Q^\pi(s,\pi'(s)) \\ &= \mathbb{E}[R_{t+1} + \gamma V^\pi(S_{t+1})|S_t = s]_{\pi'} \\ &\le \mathbb{E}[R_{t+1} + \gamma Q^\pi(S_{t+1},\pi'(S_{t+1}))|S_t = s] \\ &\le \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 Q^\pi(S_{t+2},\pi'(S_{t+2}))|S_t = s] \\ &\le \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \ldots|S_t = s] \\ &= V^{\pi'}(s) \end{aligned}$$

5) The halting condition is if the (3) inequality is an equality.
In this case the Bellman Optimality Equation is satisfied; V^Π(s) = max(a∈A) Q^Π(s, a) and V^Π(s) = V^Π*(s) = V*(s) for all s, and Π=Π*
Policy Iteration: We can iteratively improve policies generating a sequence of monotonically improving policies and VFs.

Bellman's Theorem of Optimality:
A policy Π(a|s) achieves optim. value from state s V^Π(s) = V*(s) iff:
1. For any state s′ reachable from s ∃a: p(s′,s,a) > 0
2. Π achieves the optimal value from state s′, V^Π(s′) = V*(s′)
Optimal Policies can be split into 2 subcomponents: 1) an optimal action a* and 2) Followed by an optimal policy from sucssr state s′

Policy Iteration Algorithm:
Generalised Policy Iteration Algorithm Repeat:
1. Policy Evaluation estimate VΠ
2. Policy Improvnt Π′ >= Π
Drawback:
Each iter involves a full policy eval (an iterative computation requiring multiple sweeps through the state set).

```
1. Initialization
   V(s) ∈ ℜ and π(s) ∈ A(s) arbitrarily for all s ∈ S

2. Policy Evaluation
   Repeat
       Δ ← 0
       For each s ∈ S:
           v ← V(s)
           V(s) ← Σ_s' P^π(s)_ss' [R^π(s)_ss' + γV(s')]
           Δ ← max(Δ, |v − V(s)|)
   until Δ < θ (a small positive number)

3. Policy Improvement
   policy-stable ← true
   For each s ∈ S:
       b ← π(s)
       π(s) ← arg max_a Σ_s' P^a_ss' [R^a_ss' + γV(s')]
       If b ≠ π(s), then policy-stable ← false
   If policy-stable, then stop; else go to 2
```

If evaluation is done iteratively then convergence to VΠ only happens at the limit. Instead, we can add a stopping condition:
1. ε-convergence of value func (for all s: V100(s) →-V101(s) <= ε)
2. Stop after k iterations of iterative policy evaluation
low k -> policy improv so less iter (eval/improv) before cnvrgnce.

Value iteration is the k = 1 case: the eval func stops after one sweep of the VFunc. We turn the BOE into an update rule:
1. if we know the solution to the subproblems V*(s′) then the solution V*(s) can be found with one step lookahead:

$$V^*(s) \leftarrow \max_{a\in\mathcal{A}}\sum_{s'}\mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^*(s')]$$

2. Value iteration applies these updates iteratively. (Consider starting with final rewards and working back). It's the same as the above Policy Evaluation but we don't have an explicit policy. Instead we try all actions for each state and set V(s) to the max try, and set the policy to the argmaxes of what we did and output that policy (like the coursework & above formula

Dynamic Programming performs Full Backups
- Synchronous Backups: all states are backed up in parallel (requires two copies of VFunc)
- Asynch Backup: DP backups states target only states individually in any order (so one VFunc copy):
  - for each state apply the appropriate backup in place
  - lower computation & guaranteed convergence if all states continue to be selected.
- DP uses full-width backups (looks at all successor states)
- Every successor state and action is considered.
- Effective for mid sized problems (millions of states)
- Curse of Dimensionality: Num States grows exponentially w num state vars. Even one backup can be too expensive
- Uses Bootstrapping: update value estimates based on other value estimates

2.2) Model Free Reinforcement Learning - Monte Carlo

- Learn directly from episodes of experience; model free
- Learns from complete episodes (sample traces) - no bootstrap
- Val of state=mean ret; ONLY for episodic terminating MDPs
- We estimate from experience, avg converges w more returns

First-visit MC prediction, for estimating $V \approx v_\pi$

```
Input: a policy π to be evaluated
Initialize:
    V(s) ∈ ℝ, arbitrarily, for all s ∈ S
    Returns(s) ← an empty list, for all s ∈ S

Loop forever (for each episode):
    Generate an episode following π: S0, A0, R1, S1, A1, R2, ..., S_T−1, A_T−1, R_T
    G ← 0
    Loop for each step of episode, t = T−1, T−2, ..., 0:
        G ← γG + R_t+1
        Unless S_t appears in S0, S1, ..., S_t−1:
            Append G to Returns(S_t)
            V(S_t) ← average(Returns(S_t))
```

In first visit MC we append the return of the episode from the first occurrence of a state s.
- trace 1: (s0, r=5), (s1, r=-3), (s1, r=2), (s0, r=-1) -> terminal
- trace 2: (s0,r=4), (s2, r=0), (s0, r=6), (s1, r=-4) -> terminal
- V(s0) = v(s0) trace1 + v(s0) trace2=((5-3+2-1)+(4+0+6-4))/2 = 4.5
- V(s1) = ((-3+2-1)+(-4))/2 = -3, V(s2) = (0+6-4) = 2

In Every visit MC we append the return of the episode on every occurrence of state s in the episode. Considering same traces:
- V(s0) = ((5-3+2-1)+(-1)+(4+0+6-4)+(6-4))/4 = 2.5, should be clear

We can't backup death - a *disadvantage* of Monte Carlo.
- *Online MC* - update policy after each episode
- *Batch MC* - update policy after n episodes.
- *Vanilla MC* - update policy after all episodes.
- Online MC expensive, but freq updates is rly good at the start (as our policy is random at the beginning)

Normally we batch process data x1, x2... to find mean u
But computing the mean is O(n) on all values in the typical way, and we need to do it online (update as we find new data)
More efficient: uk = $\mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1})$ (nudges mean to new val)
This form where we're nudging by 1/k (small weighting value on a new val applied to the old mean (like a prior) to get posterior) is known as an incremental estimation computation.
Incremental Monte Carlo Updates:
We can update VFs without storing sample traces using above:
1. Update V (s) incrementally after step st, at, rt+1,st+1
2. For all states st w/ return Rt (up to this pt) & N(s) (visit cnt): N(st) <- N(st) + 1, V(st) <- V(st) + (R-V(st))/N(st)

If the world changes, a running mean is useful (gradually forget old eps). V(st) <- V(st) + a(Rt-v(st)), a is learning rate.

2.3) Temporal Difference Learning

DP - bootstraps; efficiently uses data bc of subproblem structure
MC - samples; no need to know the MDP / the model
TD Learning does both bootstrapping and sampling!!:
- TD methods learn directly from episodes of experience
- Model free; learns from incomplete eps due to bootstrapping
- Updates estimates towards estimates

Remember the *MC Rule:* V(st) <- V(st) + a(Rt-v(st))
- We update the value of V(st) towards the actual return Rt

TD Update Rule: V(st) <- V(st)+a(rt+1+γV(st+1)-V(st))
- We update V(st) towards an estimated return rt+1+γV(st+1). AKA we are combining measurement rt+1, estimate V(st+1) to produce a better estimate for V(st)!
- rt+1 + γV(st+1) - V(st) is the TD Error
- rt+1 + γV (st+1) is the TD Target (both intuitive enough).

Advantages of TD & Disadvantages of MC: Learns incrementally and without the final outcome - can backup near death. MC waits until the end of episode. TD can learn from incomplete sequences in continuing (non terminating environments. MC needs complete sequences in episoding environments.

Bias-Variance Tradeoff:
The TD target depends on one random action, transition & reward.
1. Sample return Rt = rt+1+γrt+2 + ... + γ^T+1 rT is the unbiased estimate of VΠ(st)
2. "True" TD target rt+1+γVΠ(st+1) is an unbiased estimate of VΠ(st)
3. Estimated TD target rt+1+γV(st+1) is a biased estimate of VΠ(st)

```
procedure TD-ESTIMATION(π)
    Init
        V̂(s) ← arbitrary value, for all s ∈ S.
    EndInit
    repeat(For each episode)
        Initialise s
        repeat(For each step of episode)
            a action chosen from π at s
            Take action a; observe r, and next state, s'
            δ ← r + γV̂(s') − V̂(s)
            V̂(s) ← V̂(s) + αδ
            s ← s'
        until s is absorbing state
    until Done
end procedure
```

Precision - vals close together *Accurate* - vals close to mean
TD target is much lower variance than the return: Sample return depends on many random actions, transitions, rewards. So:
- MC has high var, zero bias. Good convergence properties even with function approx, not sensitive to initial val, simple to use
- MC doesn't use Markov Property (memorylessness); better in non Markov Environments
- TD has low variance, some bias. Usually more efficient than MC, TD(0) converges to VΠ(s), but not guaranteed with function approximation. More sensitive to initial value
- TD exploits the Markov Property.

In summary TD performs bootstrapping with 1 step samples.

3. Model Free Control - optimizing the VF of an unknown MDP

Section 2.2 and 2.3 cover Model-Free Learning: estimating a value function for an unknown MDP.
We use Model Free Control in two scenarios:
1. MDP known but too big to use (Curse of D) except via sampling
2. MDP is unknown but experience is sampleable.

We use Generalised Policy Iteration (GPI) for MDPs:
- GPI keeps an approx policy and approx vf yet converges to opt
- VF is repeatedly altered to approx the VF for the curr policy
- The policy is repeatedly improved w.r.t the curr VF
- Once again: we bounce between eval and improv until convrge

Policy improvement is done by making the policy greedy w.r.t the curr value func. Here we limit ourselves to an action value function so that no model is needed to construct greedy policy.
- For any action-VF Q(s,a) the corresponding greedy policy is the one that for each s ∈ S deterministically chooses an action with maximal action-value: Π(s) = arg max(a) Q(s, a)
- Policy improv is then done by constructing each Π^k+1 as the greedy policy w.r.t QΠk
- Greedy Policy improvement over V(s) requires knowing MDP: as we need to know R and P. But since we're just picking the best action value for each state off trial runs don't need that.
- Here, we consider transitions from state-action pairs rather than states to learn the value of state-action pairs
- Assume we have infinite episodes & exploring starts. Then Monte Carlo methods converges to the optimal VF and Policy
- Practically, we can't assume inf. episodes or exploring starts.

If we avoid ass. exploring starts we can only ensure all actions are selected infinitely often by ass. the agent still picks those actions.
- On Policy methods attempt to evaluate or improve the policy used to make decisions
  - "Learn on Job" - learns abt policy Π frm exp sampled from Π
  - Soft policies have in general Π(a,s) > 0 ∀s ∈ S, ∀a ∈ A; ie we have a finite probability to explore all actions
    - e-greedy policies are a form of soft policy, where the greedy action a* (argmax action on the action-VF) has a high chance of being selected while other actions have an equal share of being picked. e.g: for epsilon in the interval [0,1], Π(a,s) = 1-e+ e/|A(s)| if a* = argmax(a) Q(s,a)
    - e/|A(s)| if a =/= a*

On-policy first-visit MC control (for ε-soft policies), estimates $\pi \approx \pi_*$

```
Algorithm parameter: small ε > 0
Initialize:
    π ← an arbitrary ε-soft policy
    Q(s,a) ∈ ℝ (arbitrarily), for all s ∈ S, a ∈ A(s)
    Returns(s,a) ← empty list, for all s ∈ S, a ∈ A(s)

Repeat forever (for each episode):
    Generate an episode following π: S0, A0, R1, ..., S_T−1, A_T−1, R_T
    G ← 0
    Loop for each step of episode, t = T−1, T−2, ..., 0:
        G ← γG + R_t+1
        Unless the pair S_t, A_t appears in S0, A0, S1, A1 ..., S_t−1, A_t−1:
            Append G to Returns(S_t, A_t)
            Q(S_t, A_t) ← average(Returns(S_t, A_t))
            A* ← argmax_a Q(S_t, a)                  (with ties broken arbitrarily)
            For all a ∈ A(S_t):
                π(a|S_t) ← { 1 − ε + ε/|A(S_t)|   if a = A*
                           { ε/|A(S_t)|           if a ≠ A*
```

- Off Policy methods evaluate or improve a policy different from that used to generate the data
  - "Look over Someone's Shoulder" - learn about policy Π from experience sampled from policy Π′

Greedy in the Limit with Infinite Exploration (GLIE):
- All state-action pairs are explored infinitely many times: lim k-->∞ Nk(s,a) = ∞.
- The policy converges on a greedy policy lim k-->∞ Πk(a,s) = (a == arg max (a′ ∈A) Qk(s, a′))
- GLIE basically tells us when a schedule for adapting the exploration parameter is suficient to ensure convergence.
- The e-greedy operation is GLIE if e reduces to 0 with ek = 1/k

MC methods can batched, when the transitions are collected and then the estimates are computed once based on a large number of transitions. Can be iterative too

```
procedure MONTECARLOBATCHOPTIMISATION(n)
    Init
        Q̂(s,a) ← arbitrary value, for all s ∈ S, a ∈ A.
        π ← ε-greedy(Q̂)
    EndInit
    repeat (For each batch)
        Returns(s,a) ← an empty list, for all s ∈ S.
        for i = 1 to n do
            Get trace, τ, using π.
            for all (s,a) appearing in τ do
                R ← return from first appearance of (s,a) in τ
                Append R to Returns(s,a)
        for all s ∈ S, a ∈ A do
            Q̂(s,a) ← average(Returns(s,a))
        π ← ε-greedy(Q̂)
    until Done
    return greedy(Q̂)
```

```
procedure MONTECARLOITERATIVEOPTIMISATION(n)
    Init
        Q̂(s,a) ← arbitrary value, for all s ∈ S, a ∈ A.
        π ← ε-greedy(Q̂)
    EndInit
    for i = 1 to n do
        Get trace, τ, using π.
        for all (s,a) appearing in τ do
            R ← return from first appearance of (s,a) in τ.
            Q̂(s,a) ← Q̂(s_t,a_t) + α[R − Q̂(s_t,a_t)]
        π ← ε-greedy(Q̂)
    return greedy(Q̂)
```

Summary:
- Rather than using a model we average the many returns that start in that state. Approximates the truth
- Maintaining sufficient exploration is an issue in Monte Carlo control methods, but we need to balance exploration v exploit.
- Exploring Starts can be done by arranging this to happen, but unlikely in practice.
- In on policy methods the agent commits to always exploring, and tries to find the best policy that still explores. In off-policy methods, the agent also explores, but learns a deterministic optimal policy that may be unrelated to the policy followed.

3.2) Temporal Difference Control

As we discussed, TD has advantages over MC. So we can use TD in our control loop instead of MC:
- Apply TD to Q(s,a), use an e-greedy policy improvement
- update the policy every timestep

We learn V with SARSA for TD. Converges as normal:
- Q(S,A) <-- Q(S, A) + a(r+γQ(S′, A′) - Q(S,A))

```
Initialize Q(s,a), ∀s ∈ S, a ∈ A(s), arbitrarily, and Q(terminal-state, ·) = 0
Repeat (for each episode):
    Initialize S
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Repeat (for each step of episode):
        Take action A, observe R, S'
        Choose A' from S' using policy derived from Q (e.g., ε-greedy)
        Q(S,A) ← Q(S,A) + α[R + γQ(S',A') − Q(S,A)]
        S ← S'; A ← A';
    until S is terminal
```

SARSA converges to optimal action-VF under these conditions:
- GLIE sequence of policies Πk(a, s)
- Robbins-Munro sequence of step sizes at ❶ $\sum_{t=1}^{\infty}\alpha_t = \infty$ ❷ $\sum_{t=1}^{\infty}(\alpha_t)^2 < \infty$

So, we have to control both the exploration ϵ and the learning rate α over time. If the series of α is the reciprocals of powers of 2 we satisfy this for example. If reciprocals of positives, diverges.
Other approaches, such as Hindsight Experience Replay (HER) convert unsuccessful episodes into artificially rewarded ones (we basically failed to do X, but we have achieved Y)
Off Policy methods have a policy Π′ we follow and generate episodes from but we want to estimate for policy Π.
- Π is the target policy. Π′ is the behaviour policy,.
- we call it off-policy because we learn about a policy given only experience off (aka not following) that policy
- To use episodes of Π′ to estimate for Π we must require every action taken under Π is also taken under Π′ at least sometimes. ie: we need Π(a, s) > 0 ==> Π′(a, s) > 0 - this is known as the assumption of coverage.

Q-Learning: Consider off-policy learning of action-values Q(s,a):
- The next action is chosen using behaviour policy at+1 ~ Π′(.|st)
- But consider an alternative successor action a′ ~ Π(.|st)
- Then we update Q(st, at) in direction of the value of our alternative better action max (a) Q(st+1, a)
- So we now allow both behaviour & target policies to improve
- The target policy Π is greedy w.r.t Q(s,a):
  - Π(St+1) = arg max Q(st+1, a′)
- The behaviour policy Π′ is e-greedy w.r.t Q(s, a):
- The Q Learning target simplifies to: rt+1 + max(a′) γ Q(st+1, a′)
- ie: Q(S, A) <-- Q(S, A) + α(R + γ max(a′) Q(S′, a′) - Q(S, A)).

```
Initialize Q(s,a), ∀s ∈ S, a ∈ A(s), arbitrarily, and Q(terminal-state, ·) = 0
Repeat (for each episode):
    Initialize S
    Repeat (for each step of episode):
        Choose A from S using policy derived from Q (e.g., ε-greedy)
        Take action A, observe R, S'
        Q(S,A) ← Q(S,A) + α[R + γ max_a Q(S',a) − Q(S,A)]
        S ← S';
    until S is terminal
```

- No explicit policies - just representation of Q.
- Target policy is implied via the max(a) bit.
- Behaviour policy is the e-greedy version of the target
- Both policies updated on each step as update Q on each step

Asynchronous Agents: instead of experience replay, train multiple agents in parallel. Update weights asynchronously resulting in improved execution. xp replay = off-policy training.

4.1) Function Approximation

Need to scale up the model-free methods for prediction & cntrl.
- Large MDPs have too many states (Curse of D).
- Solution: Estimate the VF with function approximation
- We update a parameter w using MC or TD learning:
- Estimate VF with function approximation:
  - VΠ(s) ≈ V^(s, w) QΠ(s, a) ≈ Q^(s, a, w)
  - Generalise from seen states to unseen states
  - Update function approx param *w* with MC or TD learning
  - Function approximators: linear combs, NN, decision tree
- We only care about differentiable functions
- Our training method must be work w/ stationary non iid data

Gradient Descent: $J(\mathbf{w}) = \mathbb{E}\left[(V^\pi(s) - \hat{V}(s,\mathbf{w}))^2\right]$
- Goal: find parameter vector w that mins MSE b/w approxim. value function V^(s,w) and true value function V Π(s).
- GDesc: finds a local min by sliding down the gradient:
- $\Delta\mathbf{w} = -\frac{1}{2}\alpha\nabla_\mathbf{w}J(\mathbf{w}) = \alpha\mathbb{E}\left[(V^\pi(s) - \hat{V}(s,\mathbf{w}))\nabla_\mathbf{w}\hat{V}(s,\mathbf{w})\right]$ The LR -1/2α where LR α ctrls the step size. -1/2 cancels the derivative; neg = grad descn

Stochastic GD samples the grad and the avg update = full grad update: $\Delta w = \alpha\left(V^\pi(s) - \hat{V}(s,\mathbf{w})\right)\nabla_\mathbf{w}\hat{V}(s,\mathbf{w})$

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

```
Input: the policy π to be evaluated
Input: a differentiable function v̂ : S × ℝ^d → ℝ
Algorithm parameter: step size α > 0
Initialize value-function weights w ∈ ℝ^d arbitrarily (e.g., w = 0)

Loop forever (for each episode):
    Generate an episode S0, A0, R1, S1, A1, ..., R_T, S_T using π
    Loop for each step of episode, t = 0, 1, ..., T − 1:
        w ← w + α[G_t − v̂(S_t,w)]∇v̂(S_t,w)
```

States can be represented as feature vecs x(s) = (x1(s)...xn(s))^T
ie: dist of robot from landmarks, piece configs in a chess game
Crafting Features:
1. Coarse Coding: if state is in circle the feature has value 1 otherwise 0. A state's features coarsely code for its location.
2. Tile Coding: Used for computers and efficient on-line learning. The receptive fields of the features are grouped into partitions called tilings of the input space. Then being within a tile indicates we have the feature. Hashing reduces memory requirements by collapsing large tiling into many smaller tiles
3. RBFs: natural generalization of coarse coding to continuous features. Features are in [0,1] indicating various degrees to which the elem is present. $\phi_s(i) = \exp\left(-\frac{|s - c_i|^2}{2\sigma_i^2}\right)$ each ϕs(i) is the ith basis function mapping a continuous state space variable s. The RBFs are centered at ci with width σi.
4. Function Approximations can remove the bias of the person making the RL algo, identify complex patterns or weak patterns that humans can't detect.
   - Linear Function Approximation: $\Delta w = \alpha\left(V^\pi(s) - \hat{V}(s,\mathbf{w})\right)\nabla_\mathbf{w}\hat{V}(s,\mathbf{w})$ to calculate V^(s,w) we can use a linear func with weight w
   - $\hat{v}(s,\mathbf{w}) = \mathbf{x}(s)^\top\mathbf{w} = \sum_{j=1}^{n} x_j(s)w_j$ • Update rule is particularly simple $\nabla_\mathbf{w}\hat{V}(s,\mathbf{w}) = \nabla_\mathbf{w}(\mathbf{x}(s)^\top\mathbf{w}) = \mathbf{x}(s)$ $\Delta\mathbf{w} = \alpha(V^\pi(s) - \hat{V}(s,\mathbf{w}))\mathbf{x}(s)$
- Update = learning step size* predic error*feature value

Monte Carlo with VF Approximation:
- The return Rt is an unbiased noisy sample of the true V^Π(St)
- We apply supervised learning to "training data" of state return trace: (s1, r1), (s2, r2)... (sT, rT)
- Monte-Carlo evaluation converges to a local optimum, even when using non-linear value function approximation

Using linear MC policy eval the update rule is as follows:

$$\begin{aligned} \Delta\mathbf{w} &= \alpha(R_t - \hat{V}(s,\mathbf{w}))\nabla_\mathbf{w}\hat{V}(s_t,\mathbf{w}) \\ &= \alpha(R_t - \hat{V}(s,\mathbf{w}))\mathbf{x}(s_t) \end{aligned}$$

TD Learning with VF:
- The TD target Rt+1+γV^(st+1, w) is a biased single sample of the true value V^Π(st)
- We still perform supervised learning on "digested" training data: $(s_1, r_2 + \gamma\hat{V}(s_2,\mathbf{w})), (s_2, r_3 + \gamma\hat{V}(s_3,\mathbf{w})), \ldots, (s_T, r_T)$ (10)
- Using Linear TD: $\Delta\mathbf{w} = \alpha(r + \gamma\hat{V}(s',\mathbf{w}) - \hat{V}(s,\mathbf{w}))\nabla_\mathbf{w}\hat{V}(s_t,\mathbf{w}) = \alpha(r + \gamma\hat{V}(s',\mathbf{w}) - \hat{V}(s,\mathbf{w}))\mathbf{x}(s_t)$

LinearTD converges "close" to the global optimum. Doesn't extend to non-linear TD.

4.2) Deep Learning

In Deep Learning we have an input layer (one neuron per feature), a number of hidden layers, and an output layer.
Depth = num hidden layers.
Neurons take input from the previous layer: Σ_i wi xi + b.
Output feeds into *activation func before* feeding to next layer.

Activation Functions: sigmoid: σ(z) = 1 + / (1 + e^(-z))
tanh(z) = (e^z-e^(-z)) / (e^z + e^(-z)) ReLU(z) = max(0, z)

Forward Pass: evaluates / produces output given the input.
Layer one: $h_j(x) = f\left(b_j + \sum_{i=1}^{D} x_i\omega_{ji}\right)$ Layer two: $g\left(b_j + \sum_{i=1}^{D} h(x_i)\omega_{ji}\right)$
Using the output (o) and the target output (y) we compute *Loss*

Backpropagation: Uses the loss to update weights to improve the model. This is how we train.
Loss Functions: $\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=0}^{N}(o_i, y_i)^2$ (MSE $\mathcal{L}(\theta) = -\sum_{i=0}^{N} y_i \cdot \log(o_i)$ (cross entropy)

Gradient Descent: w = w - α * dJ / dw (derivative of error function w.r.t the weights)

Advantages of Neural Networks:
- Implicitly detects complex relations • Fault Tolerant
- Works with incomplete knowledge. • Concurrency

Disadvantages:
- Black box • Trial and error to find the "right" network arch
- Unclear where the error lies

Does Representation Learning - the representations aren't human detectable. With those representations, can do:
1) Classification and 2) Regression (continuous)
3) *Deep Reinforcement Learning*: takes as input the state, outputs a policy which we try on the env to gain a reward which we use to update the weights and the state.

5.1) Deep Q Learning

To do *Deep RL* we do Off-Policy Function Approximation TD:
- Function Approximation: Allows generalising from a state space much larger than memory & computational resources
- Bootstrapping: Updates targets w/ existing estimates rather than just relying on actual rewards and complete returns
- Off-Policy: Train on a distribution of transitions other than that produced by the target policy. Sweeping through the state space and updating all statuses uniformly (like in DP) doesn't specifically follow the target policy & so is Off Policy.

NNs: good function approximators. Usually supervised learners
- Learn features directly from lots of data
- Stacking layers enables learning hierarchical features
- Deep Learning is a subset of Representation Learning
- So, we don't need to hand-engineer state space features, instead we learn the relevant features from state data!

Atari DQN: • Q(s, a) is approximated by our NN
- Process Raw Pixels with conv layers for feature extraction
- Actions are from a discrete set (joystick commands)

We use CNNs to link screenshots to values.
- Given our TD Q-learning update: $Q(s_t,a) \leftarrow Q(s_t,a) + \alpha[r_{t+1} + \gamma\max_{a_{t+1}} Q(s_{t+1},a') - Q(s_t,a)]$
- We want to learn Q(st+1, a′; w) as a parametrised function (NN) with parameters w.
- We can define TD error as learning target to reduce to zero. TD error(w) = $r_{t+1} + \gamma\max_{a_{t+1}} Q(s_{t+1},a';w) - Q(s_t,a;w)$

Taking the gradient of E wrt w we obtain: $\Delta w = \alpha[r + \gamma\max_{a_{t+1}} Q(s_{t+1},a_{t+1};w) - Q(s_t,a;w)]\nabla_w Q(s_t,a;w)$
But to solve the ATARI playing problem we need optimizations:
Optimization 1: Experience Replay:
- Problem: the CNN is overfitting to the latest experienced episodes and becomes worse at dealing with different experiences of the game world
  - *We inefficiently use interactive experience:*
    - Training deep NNs requires many updates. Each has its own transition, but each is used once then discarded.
    - So we need to constantly revisit same state transitions to train. This is inefficient and slow.
  - *Experience distribution is not good:*
    - Interactive learning produces training samples that are very correlated (agent recent acts reflect recent policy)
    - The probability distribution of the input data changes
    - The network forgets old transitions
    - Because networks are monolithic changes in one part of the network can affect other parts
- *Experience Replay* solves this (or Hierarchical RL):
  - Traces are stored and replayed in mini-batches to train NNs on more than just the latest episode.
  - Instead of running Q-learning on each state/action pair as they occur during simulation/experience, the experience replay buffer system stores traces sampled
  - Mini-batches are only feasible when running multiple passes with the same data is stable with respect to the samples. AKA: transitions should have low variance for the next immediate outcomes (reward, next_state) given the same state and action pair
  - Replaying past data is also an efficient use of past exp. This is because gaining real-world exp / simulating can be costly, as Q-learning is incremental and slowly converges.
  - In summary, we separate the learning phase from the gaining experience phase, and the learning phase is based on random samples from the mini-batch table.
    - DQN interleaves acting and learning. Improving the policy will lead to diff behaviour (and so diff state-action pairs) resulting in large replay buffers going stale.
  - Consequences of Experience Replay:
    - Increased Learning Speed
    - Reduction of correlation between experiences in updating DNN - minibatching makes samples more iid
    - Reusing old transi avoids forgetting of assoc. rewards

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode = 1, M **do**
  Initialise state s_t
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(s_t, a; \theta)$
    Execute action a_t and observe reward r_t and state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Set $s_{t+1} = s_t$
    Sample random minibatch of transitions (s_t, a_t, r_t, s_{t+1}) from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } s_{t+1} \\ r_j + \gamma \max_{a'} Q(s_{t+1}, a'; \theta) & \text{for non-terminal } s_{t+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(s_t, a_j; \theta))^2$
  end for
end for

Optimization 2: Target Network:
- Whenever we do an update step on a Q-network's state we also update "nearby" states (monolithic & generalisation)
- *Problem*: unstable training resulting from bootstrapping a continuous state space representation:
  - We may have an unstable learning process because changes to Q(s, a) change Q(s', a) (for s-s' <= δ), which changes Q(s, a) on the next update and back and forth. Causes resonance where one update mirrors many times
    - This can result in run-away bias due to bootstrapping, and bias may dominate the system. Q vals then diverge.
  - In calcing the TD error, the target func changes too often
- *Slowing things down (Resonance Damping) solves this:*
  - Have a seperate target NN Q', updated less often (e.g. every 1000 steps) with a copy of the latest learned weight parameters, controls this stability (relaxation time).
    - Init two Q nets, a main Q net and a target net Q'
    - When calculating TD error use Q' not Q
    - Infrequently set Q' = Q
    - Gives the fluctuating Q relaxation time to settle before updating Q'. Relax time can be hyperparam optimized.

Optimization 3: Clipping of Rewards:
- Different tasks have different reward scales, to have an RL model generalise across all of them we need to normalize
- We clip rewards: positives are set to 1, negatives set to -1

Optimization 4: Skipping Frames:
- Computers can simulate games (60Hz). Since games are designed for humans that react much more slowly, we don't need to react at 60Hz - games designed that way!
- *Frame Skipping* - for ATARI DQN we use only every 4 frames (so operate at 15Hz) (and frame stacking uses past 4 frames as inputs) reducing computational costs and speeding up training time. Game runs closer to human reaction pace.

6.1) Advanced DQNs - avoiding falling Q-Values while Training
High variance MDPs can confuse the Q learner. Sometimes a path gives really high reward, othertimes its negative.
Since we update with a max over actions of succ state:
$Q(s_t, a) \leftarrow Q(s_t, a) + \alpha[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a)]$
we can have maximization bias: When taking the max over all actions with finite data we may overestimate.
Situation: the target network does 2 tasks:
1. We use the target network to identify the action with best Q
2. We also use it to find the Q value of this action
Problem of this: max Q value may be overestimated (var-bias problem) because an unusually high value from the main net Q doesn't mean there's an unusually high val from target net Q'
Solution: Double Q Learning
- We use a target network Q' for 1 but regular Q for 2 (or vice-versa). So the selection of the action with highest Q and the selection of Q value become independent. So we're less susceptible to the variance of each estimate. Reduces freq of overestimating the max Q value as its less likely both nets overestimate the same action.
- DDQN is also more realistic to the final values.

6.2) Policy Gradient - find optimal policy without Q/V function
Examples: Policy-Based Algorithms (like Policy Gradient), REINFORCE, DDPG, Soft-Actor-Critic.
Before we used value based methods. Here we instead use policy based methods which have a faster convergence and considered better for continuous / stochastic environments:
- We learn directly a parameterised policy Π selecting actions without a value function with policy weight parameters θ
- We optimize by looking at traces that a policy Πθ rolls out to correlate them with the rewards they incur
- Probability τ to observe trace p depends on params of policy $p(\tau) = p(s_1, a_1, \ldots, s_T, a_T)$

The optimal policy isn't from an optimal VF V* or Q* but optimal parameters θ*:
$$= p(s_1)\pi_\theta(a_2|s_1)p(s_2|s_1, a_2)p_\theta(s_2, a_2, \ldots, s_T, a_T)$$
$$= \ldots$$
$$= p(s_1)\prod_{t=1}^{T}\pi_\theta(a_t|s_t)p(s_{t+1}|s_t, a_t)$$
aka those that give us a maximum average return:

Rationale: Continuous Parameterisation will be good for continuous spaces which are otherwise too expensive for value-based approaches.

$$\theta^* = \arg\max_\theta \mathbb{E}\left[\sum_{t=1}^{T} r(s_t, a_t)\right]_{\tau \sim p_\theta(\tau)}$$

The Policy Gradient Method:
- Uses gradient ascent to move towards the direction suggested by the gradient to find the best policy
- VFs can be used to learn policy weights but not for selection
- Policy Gradient Methods are methods for learning the policy weights using the gradient of a perf measure w.r.t weights
- Search directly for Π* optimising params θ:

infinite horizon case: $\theta^* = \arg\max_\theta \mathbb{E}\left[\sum_t r(s_t, a_t)\right]_{\tau \sim p_\theta(\tau)}$
finite horizon: $\theta^* = \arg\max_\theta \sum_{t=1}^{T} \mathbb{E}\left[r(s_t, a_t)\right]_{(s_t, a_t) \sim p_\theta(\tau)}$

Any supervised learning model can be used to learn θ
Finding the Gradient of a Policy:
- *Notation*: Π(a|s,θ) is the probability the action a is taken in state s given policy weight vector θ
- The performance measure J(θ) is the value of the start state under the parameterised policy: $J(\theta) = V^\pi(s_0) = \mathbb{E}\left[\sum_{t=1}^{T} r(s_t, a_t)\right]_{\tau \sim p_\theta(\tau)}$
- To optimize θ we need the deriv of J(θ) which depends on traces τ. This means we have:
  - derivatives on action selection (determined by Π(a|s,θ))
  - derivatives on the distribution of states p(s), indirectly determined by Π(a|s,θ).

1. Finding the derivative: Finite Difference Policy Gradient
Vector difrntion of an M dim vector x is the partial derivs: $\nabla_{\mathbf{x}} f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_M} \end{bmatrix}$
- Compute the partial deriv of the perf measure J(θ) w.r.t θ using finite diff gradient approximation
- Then compute J(θ)

Steps:
1. Loop over all M dims of θ with index k, set uk as unit vector 1 in the kth component, 0 elsewhere; e.g u2 = [0 1 0 0 ...]^T
2. Perturb params θ by small ε in kth dim only and compute J(θ+uk∈)
3. Then the partial deriv is approximated by finite diffs: dJ/dθk = (J(θ+uk∈) - J(θ)) / ε
4. We then update θk = θk + αdJ/dθk

This is simple but noisy & ineff. Works for non-differen policies

2. Finding the Derivative: Direct Policy Gradients
We have the policy gradient theorem: $\theta^* = \arg\max_\theta \mathbb{E}\left[\sum_{t=1}^{T} r(s_t, a_t)\right]_{\tau \sim p_\theta(\tau)}$

Average return is approximated by empirical mean over N traces

$$J(\theta) = \mathbb{E}\left[\sum_{t=1}^{T} r(s_t, a_t)\right]_{\tau \sim p_\theta(\tau)} \approx \frac{1}{N}\sum_{i=1}^{N}\sum_t r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \quad (10)$$

$$\theta^* = \arg\max_\theta \underbrace{E_{\tau \sim p_\theta(\tau)}\left[\sum_t r(\mathbf{s}_t, \mathbf{a}_t)\right]}_{J(\theta)}$$

a convenient identity: $\pi_\theta(\tau)\nabla_\theta \log \pi_\theta(\tau) = \pi_\theta(\tau)\frac{\nabla_\theta \pi_\theta(\tau)}{\pi_\theta(\tau)} = \nabla_\theta \pi_\theta(\tau)$

$$J(\theta) = E_{\tau \sim \pi_\theta(\tau)}[r(\tau)] = \int \pi_\theta(\tau) r(\tau) d\tau$$

$$\sum_t r(\mathbf{s}_t, \mathbf{a}_t)$$

$$\nabla_\theta J(\theta) = \int \nabla_\theta \pi_\theta(\tau) r(\tau) d\tau = \int \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) r(\tau) d\tau = E_{\tau \sim \pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau) r(\tau)]$$

$$\theta^* = \arg\max_\theta J(\theta)$$

$$J(\theta) = E_{\tau \sim p_\theta(\tau)}[r(\tau)] \qquad \log \text{ both sides} \qquad \log \pi_\theta(\tau) = \log p(\mathbf{s}_1) + \sum_{t=1}^{T} \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) + \log p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$$

$$\nabla_\theta J(\theta) = E_{\tau \sim \pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau) r(\tau)]$$

$$\nabla_\theta \left[\log p(\mathbf{s}_1) + \sum_{t=1}^{T} \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) + \log p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)\right]$$

$$\nabla_\theta J(\theta) = E_{\tau \sim \pi_\theta(\tau)}\left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t)\right)\left(\sum_{t=1}^{T} r(\mathbf{s}_t, \mathbf{a}_t)\right)\right]$$

We arrived at this final eq, a version of the policy gradient theorem, by expanding the state sequence from a fixed starting point s1. Generally we can obtain results for the stationary distribution of starting states.

REINFORCE: implements the Policy Gradient Theorem result:
- loop: improve the policy -> generate samples (run policy) -> fit a model to estimate the return -> improve the policy...
  1. sample $\{\tau^i\}$ from $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ (run the policy)
  2. $\nabla_\theta J(\theta) \approx \sum_i \left(\sum_t \nabla_\theta \log \pi_\theta(\mathbf{a}_t^i|\mathbf{s}_t^i)\right)\left(\sum_t r(\mathbf{s}_t^i, \mathbf{a}_t^i)\right)$
  3. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
- Before we had to learn a VF through function approxim. then derive a corresponding policy
- But learning the VF could be intractable.
- REINFORCE allows us to directly optimize policies to avoid this.
  - REINFORCE suffers from high variance in samples trajectories, stabilising model params is difficult
  - Any erratic trajectory can cause a sub-optimal shift in policy distribution if it by chance gets low/high rewards
  - To solve this we use a baseline or actor-critics.
  - REINFORCE: good first attempt at policy-grad with 0 bias but high variance is a problem. Future algorithms reduce variance by being smarter with correlating rewards with trajectories. Involve subtracting a baseline from reward (to reduce rewards), or using an advantage term
  - Converges in the limit of large amounts of data

Causality: A way to reduce variance:
- Apply Causality to lower amt of total data (and so variability):

$$\nabla_\theta J(\theta) = \sum_{t=1}^{N}\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_t^i|\mathbf{s}_t^i)\right)\left(\sum_{t=1}^{T} r(\mathbf{s}_t^i, \mathbf{a}_t^i)\right)$$

- The policy at time t' > t obviously can't affect the reward at t so rewrite the terms entering the sums as:

$$\nabla_\theta J(\theta) = \sum_{t=1}^{N}\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_t^i|\mathbf{s}_t^i)\left(\sum_{t'=t}^{T} r(\mathbf{s}_{t'}^i, \mathbf{a}_{t'}^i)\right)$$

J(θ) breaks down as: $J(\theta) = \sum_s p^\pi(s) V^\pi(s) = \sum_s p^\pi(s) \sum_a \pi_\theta(a|s) Q^\pi(s, a)$
Its gradient is: $\nabla_\theta J(\theta) = \nabla_\theta \sum_s p^\pi(s) \sum_a Q^\pi(s, a) \pi_\theta(a|s)$
$$\propto \sum_s p^\pi(s) \sum_a Q^\pi(s, a) \nabla_\theta \pi_\theta(a|s)$$

But then the gradient depends on: the action Πθ
- The stationary distribution of states pΠ(s)' (resulting indirectly from Πθ via the dynamics p(s'|s,a)).
- The env (dynamics+rewards) is unknown, so it's hard to find the impact on the state dist by a policy update (gradient step on θ).
- The Policy Gradient Theorem then allows us to break the complication up; it transforms the deriv of the cost-to-go J(θ) into the deriv of the policy which is easy to compute.

We need to parameterise the policy Πθ for a distrib of actions:
- We use the Gaussian Dist - good dist for CRVs
- The multivariate Gaussian distribution is fully characterized by a mean vector μ and a covariance matrix Σ and defined as:

$$p(\boldsymbol{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = (2\pi)^{-\frac{D}{2}}|\boldsymbol{\Sigma}|^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right),$$

where x∈R^D is a random variable. We write $\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma})$ or $\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$

Lets spell out the elements of the 2-D Gaussian distribution of two random variables $\boldsymbol{x}, \boldsymbol{y}$, where $\boldsymbol{\Sigma}_{xx} = \mathrm{Cov}[\boldsymbol{x}, \boldsymbol{x}]$ and $\boldsymbol{\Sigma}_{yy} = \mathrm{Cov}[\boldsymbol{y}, \boldsymbol{y}]$ are the marginal covariance matrices of $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively, and $\boldsymbol{\Sigma}_{xy} = \mathrm{Cov}[\boldsymbol{x}, \boldsymbol{y}] = \boldsymbol{\Sigma}_{yx}$ is the cross-covariance matrix between $\boldsymbol{x}$ and $\boldsymbol{y}$.

In robotics and other real-world control systems, continuous policies are best. Gaussian policies Π(a|s) can be done as dists:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})\right)\left(\sum_{t=1}^{T} r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})\right)$$

example: $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t) = \mathcal{N}(f_{\text{neural network}}(\mathbf{s}_t); \Sigma)$

$$\log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) = -\frac{1}{2}\|f(\mathbf{s}_t) - \mathbf{a}_t\|_\Sigma^2 + \text{const}$$

$$\nabla_\theta \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) = -\frac{1}{2}\Sigma^{-1}(f(\mathbf{s}_t) - \mathbf{a}_t)\frac{df}{d\theta}$$

We can interpret the term *f(st, θ) - at* as a curious form of supervised learning - where we fit the network output computed on the states f (s) to the actions a.
It is curious because pure maximum-likelihood-type supervised learning (fitting states and actions to each other) is actually reward-weighted fitting in Policy-Gradient-based RL.

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})\right)\left(\sum_{t=1}^{T} r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})\right) \qquad \text{maximum likelihood:} \quad \nabla_\theta J_{\text{ML}}(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\nabla_\theta \log \pi_\theta(\tau_i)$$

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\nabla_\theta \log \pi_\theta(\tau_i) r(\tau_i) \qquad \sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})$$

These avgs can be interpreted as weighted avg over policies:
- Increase the weight of trajectories that are good in reward
- Decrease the weight of trajectories that are bad in rewards

7.1) Actor Critic
Policy Based Methods are known as actors. (REINFORCE)
Value Based Methods are known as critics (DQN, Q Learning)
Actor-Critic Methods combine the two.
- Actor takes input state; outputs action. 'acts out'; controls how agent behaves by learning the optimal policy
- Critic evals the action by computing the value function
- The two "models" compete & get better at their own role.
- The combo learns more efficiently than 2 separate models.

Deep Actor-Critic Methods offer the best of both worlds:
- Policy Gradient agents can take many actions during an episode, so updates have high variance as assigning the correct credit to many actions is difficult
- So it'll take a lot of updates for your policy to converge.
- Also, PG method only works in episodic regimes (cf. Monte Carlo tabular RL). If our agent isn't acting in an episodic environment, you don't get updated ("can't backup death")

We use the advantages of TD Learning and define a critic:
- Critic predicts long term val of the state s, or a state-action pair (s, a). We use value Q(s, a) instead of empirical returns.

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r_t(s, a) + \max_{a'} \gamma Q(s', a') - Q(s, a)]$$

- Lower variance, as update our params at each step using Q values (bootstrapping) ... but higher bias (cf. TD learning).
- Faster convergence, we update the policy within an episode
- We bootstrapping from our own predictions of Q-values.
- Moreover, we can learn in non-episodic domains.

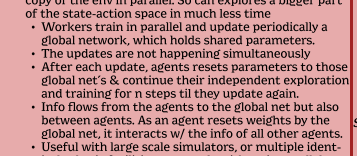
So, in A-C (e.g: deep learning as function approximators):
- The Actor updates the policy dist in the dir suggested by the Critic (such as w/ policy gradients by updating ε).
- The Critic estimates the value function. This could be the state-action-value (Q function) or state-value (V function), (such as Function Approximation/DQN style updates of w).

A simple Q-driven policy-gradient actor-critic.

Algorithm 1 Q Actor Critic

Initialize parameters s, θ, w and learning rates α_θ, α_w; sample $a \sim \pi_\theta(a|s)$.
for $t = 1 \ldots T$: **do**
  Sample reward $r_t \sim R(s, a)$ and next state $s' \sim P(s'|s, a)$
  Then sample the next action $a' \sim \pi_\theta(a'|s')$
  Update the policy parameters: $\theta \leftarrow \theta + \alpha_\theta Q_w(s, a)\nabla_\theta \log \pi_\theta(a|s)$; Compute the correction (TD error) for action-value at time t:
    $\delta_t = r_t + \gamma Q_w(s', a') - Q_w(s, a)$
  and use it to update the parameters of Q function:
    $w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s, a)$
  Move to a $\leftarrow a'$ and s $\leftarrow s'$
end for

Reducing Variance: Baselines
- We are summing up our PG terms for 3 trajectories we evaluated: ∇θlogΠtheta(τ)= 1/4, 1/5, 1/6 and 0.4
- Our 3 trajectories have returns r(τ) = 100, 101 and 102
- So the variability is SD(1/4 * 100, 1/5 * 101, 1/6 * 102) = SD(25, 20.2, 17) ~ 4.
- If we were to subtract a baseline of b = 100 from the returns r(τ), so r(τ) - b = 0, 1 and 2, we obtain terms 1/4 * 0, 1/5*1, 1/6*2, which have SD(0, 1 5 , 1 3) = 0.17
- We thus reduce variability by subtracting a good baseline
  - Putting this into practice gives *Advantage-Actor Critic*
    - The key element is the Advantage Element. It tells us what the advantage amount is by picking a in state s.
    - Q functions are decomposed into 2 pieces: the original state Value function V(s) and the advantage value A(s, a) of taking a specific action given a state: Q(s, a) = V (s) + A(s, a)
    - Advantage func says how much better an action a is compared to the others at s, while as we know the VF captures how good it is to be at this state.
      - A(s,a) = Q(s,a) - V(s) A(s,a) = r + γ V(s^) - V(s)



- So the eval of an action is based on how good the action is + how much better it can be
- Reduces the high var of policy nets; stabilises training
- Reduces var without introducing bias in Policy Grad
- Asynchronous Advance Actor Critic has multiple indep agents (nets) with own weights, who interact with a diff copy of the env in parallel. So can explores a bigger part of the state-action space in much less time
  - Workers train in parallel and update periodically a global network, which holds shared parameters.
  - The updates are not happening simultaneously
  - After each update, agents resets parameters to those global net's & continue their independent exploration and training for n steps til they update again.
  - Info flows from the agents to the global net but also between agents. As an agent resets weights by the global net, it interacts w/ the info of all other agents.
  - Useful with large scale simulators, or multiple identical robotic facilities, to run algorithms in parallel.

7.2 Deep Deterministic Policy Gradient (DDPG):
- This is a deterministic policy, model-free, off policy, actor-critic algorithm which
  - learns the value model (Q-function) and a policy Π.
  - learns by off-policy methods and uses the Bellman equation to learn the Q-function (bootstrapping part 1)
  - Uses Q-function to learn policy Π (bootstrapping part 2)

Algorithm: (θQ is params of the VF approx instead of w, θu is params of policy func approx (we used θ), R is replay buffer.

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for $t = 1, T$ **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \underbrace{\nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}}_{\text{Deterministic version of PG Theorem}}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

DDPG Optimizations to make it work:
1. *Exploration in continuous action spaces:*
   - Use Gaussian Distributed Noise that perturbs a mean action μ'(s) = μθ(s) + N - similar to discrete action e-greediness (but here we explore closer to μ)
2. *Replay Buffers, Minibatches & Target Networks*: controls the learning variability
3. *Error-based learning* on the mean-squared Bellman error
   - MSBE (L) tells us how close we came to satisfying the BE: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta_Q))^2$
4. *Policy Gradient Based Learning on an Actor Network:*
- We use a deterministic action version of the PG Theorem
  - DQN updates the target net in jumps, with many remaining episodes being frozen. Doesn't work in continuous control; small policy changes must cause small control changes.
  - So instead, DDPG uses smooth updates on the param vec θ that is shared by both the actor and the critic which updates slowly but steadily over steps t:
    θ'_t+1 = βθ't + (1-β)θ_t, 0 < β < 1
  - The parameter β controls the degree of blending (or forgetting) between the current parameter estimate θ and its update θ' at t to determine the parameter vector at the next t + 1.

Like how the Policy Gradient theorem is a policy-centered equivalent of the Bellman Theorem, DDPG is the continuous action version of the discrete action DQN.

Desired qualities for Reinforcement Learning
1. Stable learning & not reliant on hy-params: we don't want costly h-param tuning - RL algos should work immediately.
2. Sample efficiency: Good space complexity is a prerequisite for efficient skill acquisition
   - Most algorithms covered so far have poor sample efficiency (though introducing features improved learning stability). Fine for simulation settings or "toy"-problems such as video games
   - Real-world applications, like Robots require us to be efficient with our exploration, as interactive learning is costly and does not scale well.
   - *Sample Complexity of Exploration*: Number of time steps where the algo doesn't select near-optimal actions.

Soft-Actor-Critic: Advanced RL Algorithm for continuous action spaces. It combines:
- the strengths of actor-critic
- with a maximum entropy framework for exploration
- results in a balanced tradeoff bw exploration & exploitation
- The policy is trained to maximize tradeoff bw expected return and entropy, a measure of randomness in the policy
- Increasing entropy = more exploration; accelerates learning
- Can also prevent the policy from prematurely converging to a bad local optimum (thus, avoiding having to go in-depth the e-greedy decay schedule tuning)

Entropy of a Discrete RV x: $\mathcal{H}(x) = -\sum_{i=1}^{n} p(x_i)\log_2 p(x_i)$
- It's units are bits, e.g. a fair coin has the entropy of 1bit.
- Max entropy is when all outcomes xi are equally probable, min (zero) entropy is when outcome is deterministic.

For Continuous Random Variables: $\mathcal{H}(x) = -\int p(x)\log_2 p(x) dx$

Objective Functions in Soft-Actor Critic:
- We use the entropy to boost its sample complexity.
- The primary goal of SAC is to max expected cumulative reward, by including entropy to encourage exploration:

$$J(\pi) = \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \rho_\pi}\left[\sum_t \gamma^t \left(r(\mathbf{s}_t, \mathbf{a}_t) + \lambda \mathcal{H}(\pi(\cdot|\mathbf{s}_t))\right)\right]$$

- J(Π) is the cost-to-go objective function
- r(st, at) is the immediate reward function,
- pΠ = pΠ(s, a) is the state-action distribution under policy pΠ,
- γ is the discount factor,
- λ is a temperature parameter controlling the importance of the entropy term H,
- p(·|st) is the policy distribution given state st.

Soft-Actor Critic continued:
- Soft Actor Critic has two Q-Value Functions,Qw1 and Qw2. This mitigates maximization bias in policy improvement.
- In this entropy-regularized reinforcement learning, the agent gets an bonus reward at each time step proportional to the entropy of the policy at that timestep.
- The critic (Q-functions) are updated to minimize the Bellman residual:

$$J_Q(\theta_i) = \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \mathcal{D}}\left[\frac{1}{2}\left(Q_{\theta_i}(\mathbf{s}_t, \mathbf{a}_t) - \left(r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \mathbb{E}_{\mathbf{s}_{t+1} \sim \rho}\left[\min_{j=1,2} Q_{\theta_j}(\mathbf{s}_{t+1}, \mathbf{a}') - \lambda \log \pi(\mathbf{a}'|\mathbf{s}_{t+1})\right]\right)\right)^2\right]$$

- Where D is the replay buffer and pdyn represents the environment dynamics. Note a' comes from the current policy vΠ, not the replay buffer
- The Equation can be interpreted as follows:
  - Across all sampled states from our exp replay buffer, decrease the squared difference between the prediction of our Q-value net & the expected prediction of the Q function plus the entropy of the policy function p measured here by the negative log of the policy function

The policy or actor is updated by maximizing:

$$J_\pi(\theta) = \mathbb{E}_{\mathbf{s}_t \sim \mathcal{D}}\left[\log \pi_\theta(a_\theta(\mathbf{s}_t)|\mathbf{s}_t) - Q_w(\mathbf{s}_t, a_\theta(\mathbf{s}_t))\right]$$

where Πθ is the policy parameterized by Φ. We have also introduced a function aθ allowing us to rewrite the expectation over actions into an expectation over noise (random variables), there by removing the dependence on parameters.