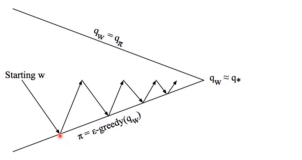
From evaluation to control: FA in GPI



- Policy evaluation: Approximate policy evaluation $\hat{Q}(s,a,\mathbf{e}) \approx Q^{pi}(s,a)$
- Policy improvement: ϵ-greedy policy improvement

DQN

DQN: Bringing Deep Learning into RL I

In principle it sounds easy: use convolutional neural networks to link screen shots to values.
Given our TD Q-learning update:

$$Q(s_t,a) \leftarrow Q(s_t,a) + \alpha[r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1},a') - Q(s_t,a)] \quad (1)$$

we want to learn $Q(s_{t+1},a';w)$ as a parametrised function (a neural network) with parameters w.
We can define the TD error as our learning target that we want to reduce to zero.

$$\text{TD error}(w) = r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1},a';w) - Q(s_t,a;w) \quad (2)$$

Taking the gradient of E wrt w we obtain:

$$\Delta w = \alpha[r + \gamma \max_{a_{t+1}} Q(s_{t+1},a_{t+1};w) - Q(s_t,a;w)]\nabla_w Q(s_t,a;w) \quad (3)$$

However, the ATARI playing problem required more engineering to solve properly

- Experience Replay
- Target Network
- Clipping of Rewards
- Skipping of Frames

DQN: Experience Replay I

The CNN is easily overfitting the latest experienced episodes and the network becomes worse at dealing with different experiences of the game world:

- Complication 1: Inefficient use of interactive experience
  - Training deep neural networks requires many updates, each has its own transition
  - But now each state transition is used only once, then discarded
  - so we would need to constantly revisit the same state transitions to train.

  This is inefficient and slow.
- Complication 2: Experience distribution
  - Interactive learning produces training samples that are highly correlated (agents recent actions reflect recent policy)
  - The probability distribution of the input data changes
  - The network forgets transitions that were in the not so recent past (overwriting past memory, in neuroscience known as "extinction")
  - Moreover, because networks are monolithic, changes in one part of the network can have side-effects on other parts of the network [Why is this not a problem for table-based RL?]

Solution: Experience Replay[2] (alternative use Hierarchical RL).

- Experiences (traces) are stored and replayed in mini-batches to train the neural networks on more than just the last episode.
- Instead of running Q-learning on each state/action pairs as they occur during simulation or actual experience, the experience replay buffer system stores the traces sampled.
- Mini-batches are only feasible when running multiple passes with the same data is somewhat stable with respect to the samples. I.e. the transitions should show low variance/entropy for the next immediate outcomes (reward, next state) given the same state, action pair.
- Replaying past data is also a more efficient use of previous experience, by learning (training the neural network) with it multiple times. This is key when gaining real-world experience is costly (e.g.simulation time) as the Q-learning updates are incremental and do not converge quickly.

The learning phase is separated from gaining experience, and based on taking random samples from the mini-batch table.
DQN interleaves the two processes - acting and learning. Improving the policy will lead to a different behaviour (and different state action pairs), resulting too large replay buffers getting stale.
In summary

- Reduction of correlation between experiences in updating DNN – mini-batches effectively make the samples more independent and identically distributed.
- Increased learning speed with mini-batches due to reuse/multi-use of epxerience
- Reusing/Replaying past transitions to avoid catastrophic forgetting of associated rewards

DQN: Experience Replay V

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

DQN: Target Network I

Whenever we run an update step on a Q-network's state we also update "near by" states (monlithic architecture + generalisation ability of CNNs).

$$Q(s_t,a) \leftarrow Q(s_t,a) + \alpha[r_{t+1} + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a)] \quad (4)$$

Issue: Unstable training resulting from bootstrapping a continuous state space representation

- We may have an unstable learning process, because changes to $Q(s,a)$ change $Q(s',a)$ (with $s - s' \approx \delta$) which changes $Q(s,a)$ on the next CNN update and then in turn $Q(s',a)$ forth. This can lead to a resonance effect
- The effect may result in a run-away bias from bootstrapping and subsequently dominating the system numerically, causing the estimated Q values to diverge.
- In calculating the TD error calculation, the target function is changing too frequently when using convolutional neural networks.

Solution: Slow things down (as resonance dampener)
Using a separate target network Q', updated every fewer time (e.g. every 1000 steps) with a copy of the latest learned weight parameters, controls this stability (relaxation time).

- Initialise two Q networks: a main Q-network (Q), and a target network (Q')
- When calculating the TD-error, use Q', not Q
- Infrequently set $Q' = Q$
- This gives the highly fluctuating Q time to settle (we can measure this so called "relaxation time" empirically and set it accordingly) before updating Q'

Clipping Rewards: Varation in rewards maeks the training unstable ⇒ Clip positive reward to 1, and negative reward to -1

DQN: Clipping rewards I

Clipping rewards:

- Each ATARI game has different score scales. For example, in Pong, players can get 1 point when winning the play. Otherwise, players get -1 point. However, in Space Invaders, players get 10-30 points when defeating invaders. This difference would make training unstable.
- Clipping Rewards: clips the rewards, with all positive rewards are set +1 and all negative rewards are set -1.

Clipping Frame to reducing computational cost and accelerating training times.

DQN: Skipping frames I

Skipping frames:

- Computer can simulate games (e.g. the ATARI simulator does this at 60 Hz) much faster than a human player would be able to react to the game, and this implies that the video game design and the required actions can be slower.
- Frame skipping uses only every 4 video game frames (i.e. 15 Hz) … (and frame stacking uses the past 4 frames as inputs), thus reducing computational cost and accelerating training times. It also makes the game run at a speed comparable to human reaction time.

Dobule Q network

Normal netowrk will produce a maximisation bias

Max at the Casino II

- So when update the value for $Q(Start, Left)$ we perform a *max* over actions in successor state Casino (e.g. from traces yielding $r(Casino, A, Hotel) = -1, r(Casino, B, Hotel) = +1, r(Casino, A, Hotel) = -1$) and immediately note down the $+1$ reward from choosing B.
- In the limit of large number of trials all actions will yield an average reward of -0.02, but that is only asymptotically reached.
- This results in a maximisation bias, namely that when taking a *max* over all actions with (very) finite data we may always overestimate values. This illustrates a general relationship (that we do not prove here), but we can see empirically.

Use the main netowrk instead of the target network to do action selection (compared with normal target network)

Double Q Learning (van Hasselt et al, 2017, AAAI) I

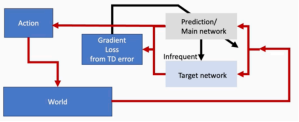
Situation: In regular Deep Q-learning we use the target network for two tasks:

- Identify action with highest Q value,
- Determine Q value of this action

Complication: The maximum Q value may be overestimated (variance-bias problem), because an unusually high value from the main network Q does not mean that there is an unusually high value from the target network Q'.
The problem with (Deep) Q-Learning is that the same samples (i.e. the Q network) are being used to decide which action is the best (highest expected reward), and the same samples are also being used to estimate that specific action-value.
Solution: In Double Q-Learning (van Hasselt et al, 2017, AAAI) we separate the two estimates: We use a target network Q' for 1. but use the regular Q for 2. (or the converse). Thus, we make the selection of the action with highest Q value, and selection of the Q value used in Bellman updates become independent from each other (different networks) so we become less susceptible to variance in each estimate. This reduces the frequency by which the maximum Q value may be overestimated, as it is less likely that both the networks are overestimating the same action.

DQN: Target Network II

$$\text{TD error}(w) = r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1},a';w) - Q(s_t,a;w)$$



Double Q network provide prediction that are closer to the final value and more stable and less biased.

Dealing with Bias II

- The two straight horizontal orange (for DQN) and blue (for Double DQN) lines in the top row are computed by running the corresponding agents after learning concluded (end of training), and averaging the actual discounted return obtained from each visited state. These straight lines would match the learning curves at the right side of the plots if there is no bias during learning and training lasted long enough.
- We thus see that learning with DDQN is much not only more realistic (closer to the final values) and more stable and less biased (far less systematic overshooting).

Policy Gradient

Why using policy based method:

For some environment that has large action space (especially continous action space), iterate through all actions might take a long time, thus have a policy that direct output action would be more efficient.

Policy gradients interim summary I

- Using gradient ascent, we can move toward the direction suggested by the gradient to find the best policy that produces the highest return.
- A value function may be used to learn the policy weights but this is not required for action selection
- Policy gradient methods are methods for learning the policy weights using the gradient of some performance measure with respect to the policy weights
- Search directly for the optimal policy π^* by optimising policy parameters θ
- Policy gradient methods seek to maximise performance and so the policy weights are updated using gradient ascent

Objective

Towards policy gradient derivation I

Recall that the optimal policy is the policy that achieves maximum expected (future) return

- Find

$$\theta^* = \arg\max_\theta \mathbb{E}\left[\sum_t r(s_t,a_t)\right]_{\tau \sim \rho_\theta(\tau)}$$

(infinite horizon case)

- $$\theta^* = \arg\max_\theta \sum_{t=0}^{T} \mathbb{E}\left[r(s_t,a_t)\right]_{(s_t,a_t)\sim\rho_\theta(\tau)}$$

(finite horizon case)

We can use any parametric supervised machine learning model to learn policies $\pi(a|s;\theta)$ where θ represents the learned parameters

Finit difference are simple, inefficient(especially when have tons of parameters), and sometime efficient (it can even work for non-differentiable ones)

Approach 1: Finite Difference Policy Gradient III

- The partial derivative of the k-th component of θ is approximated by finite differences

$$\frac{\partial J}{\partial \theta_k} \approx \frac{J(\boldsymbol{\theta} + \boldsymbol{u}_k\epsilon) - J(\boldsymbol{\theta})}{\epsilon}$$

- Update rule $\theta_k = \theta_k + \alpha(\frac{J(\theta+u_k\epsilon)-J(\theta)}{\epsilon}))$

This is a simple, noisy and inefficient procedure that is sometimes effective, as it works for arbitrary policies (even non-differentiable ones).
Could we find a less empirical/numerical approach to optimise the policy gradient?

Directly calculating the policy gradient

Direct Policy Gradients: Derivation idea 2/2

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

$$J(\theta) = E_{\tau\sim\pi_\theta(\tau)}[r(\tau)] \qquad \text{log of both sides} \quad \frac{\pi_\theta(s_1,a_1,\ldots,s_T,a_T)}{\pi_\theta(\tau)} = p(s_1)\prod_{t=1}^{T}\pi_\theta(a_t|s_t)p(s_{t+1}|s_t,a_t)$$

$$\nabla_\theta J(\theta) = E_{\tau\sim\pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau) r(\tau)] \qquad \log \pi_\theta(\tau) = \log p(s_1) + \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t,a_t)$$

$$\nabla_\theta \left[\log p(s_1) + \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t,a_t)\right]$$

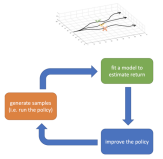
$$\nabla_\theta J(\theta) = E_{\tau\sim\pi_\theta(\tau)}\left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\right)\left(\sum_{t=1}^{T} r(s_t,a_t)\right)\right]$$

We arrived at this final equation, a version of the policy gradient theorem, by expanding the state sequence from a fixed starting point $\mathbf{s}_1$. More generally we can obtain results for the stationary distribution of starting states (not discussed here).

REINFORCE algorithms

REINFORCE-ing good trajectories over bad ones

A direct implementation of the Policy Gradient Theorem result is the classic 3-step Policy Gradient algorithm REINFORCE.



- Before we had to learn a value function through function approximation and then derive a corresponding policy
- Often learning a value function can be intractable (more unstable / was, at the time, intractable for large state spaces).
- REINFORCE provided a way to directly optimize policies to get around this problem.

REINFORCE algorithm:

1. sample $\{\tau^1\}$ 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. $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$

- The method suffer from high variance in the sampled trajectories, thus stabilising model parameters is difficult.

What was the impact of REINFORCE? I

- Before: we had to learn a value function through function approximation and then derive a corresponding policy
- But, often learning a value function can be intractable (more unstable, computationally demanding for large state spaces, observability).
- REINFORCE: provided a way to directly optimize policies to get around this problem. It was the direct result of rendering the PG theorem useful after its derivation

REINFORCE was the crucial first step at popularising Policy Gradients (PGs).

REINFORCE algorithms suffer greatly from variance, a single erratic trajectory can cause a suboptimal shift into wired area of optimisation.

REINFORCE

- In the limit of large amounts of data, the model will converge to the optimal parameters
- But, the REINFORCE method suffers from high variance in the sampled trajectories, thus stabilising model parameters θ can be difficult.
- Any erratic trajectory can cause a sub-optimal shift in the policy distribution if by chance generates high rewards or vice versa.
- Many subsequent algorithms were proposed to reduce the variance while keeping the bias unchanged by being smarter correlating rewards with trajectories. These may involve subtracting a baseline from the reward term (so as to keep values smaller) and by using a so called advantage term (Schulman et al, 2016).

Gaussian Policies

In robotics and many other real-world control systems it makes sense to have continuous policies. A simple way is to introduce Gaussian policies, i.e. $\pi(a|s)$ as a Gaussian distributions

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

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 $f(\mathbf{s}_t,\theta) - \mathbf{a}_t$ as a curious form of supervised learning - where we fit the network output computed on the states $f(s)$ to the actions a.

Policy gradients is trial-and-arror (like MC)

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 reinforcement learning.

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

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

These averages can be reinterpreted as weighted average over policies

- We increase the weight of trajectories that are good in reward
- We decrease the weight of trajectories that are bad in rewards

The action in future cannot affect the actions in the past.

Ways to reduce variance: Causality I

- Apply Causality to reduce the amount of total data (and thus variability):
- Strictly speaking $\nabla_\theta J(\theta) = \sum_{t'=1}^{T}(\sum_{t=1}^{t'}\nabla_\theta \log \pi_\theta(\mathbf{a}_t^i|\mathbf{s}_t^i))(\sum_{t'=1}^{T} r(\mathbf{s}_{t'}^i,\mathbf{a}_{t'}^i))$
- but, the policy at time $t' > t$ cannot affect the reward in the past (time t), so we can restrict the terms entering the sums tp $\nabla_\theta J(\theta) = \sum_{t=1}^{T}(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_t^i|\mathbf{s}_t^i))(\sum_{t'=t}^{T} r(\mathbf{s}_{t'}^i,\mathbf{a}_{t'}^i))$

Actor Critics

Policy based method has low bias but larger variance

Actors & Critics

- Policy-Based methods (actors)
  - Find the optimal policy directly without the Q/V-function.
  - Policy-based are considered to have a faster convergence and are better for continuous and stochastic environments.
  - Examples: Policy Gradients algorithms such as REINFORCE
- Value Based methods (critics)
  - Find/approximate optimal value function (mapping action to value).
  - These are considered more sample efficient and steady.
  - Examples: Q Learning, Deep Q Networks, Double Deep Q Networks, etc.

Best of both worlds with Deep Actor-Critic Methods I

Why combine deep RL methods as Actor-Critics?

- Policy Gradient agents might take a lot of actions over the course of an episode, it is hard to assign correct credit to many actions, which means that these updates have a high variance
- Therefore, it may take a lot of updates for your policy to converge.
- Moreover, PG method only works in episodic regimes (cf. Monte Carlo tabular RL). If your agent is not acting in an episodic environment, it will never get an update (cf. "You cannot backup from death")

TD Error

- Generate the trace
- Update the policy value
- calcualte the TD error
- update the Q value error

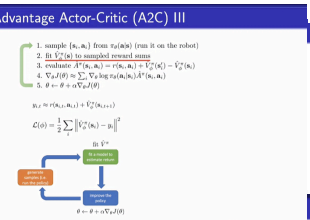
Naive, Deep, Q actor-critic (Weng, 2019) I

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

Advantage Actor-Critic (A2C) III



$$y_{i,t} \approx r(\mathbf{s}_{i,t},\mathbf{a}_{i,t}) + \hat{V}_\phi^\pi(\mathbf{s}_{i,t+1})$$

$$\mathcal{L}(\phi) = \frac{1}{2}\sum_i \left\|\hat{V}_\phi^\pi(\mathbf{s}_i) - y_i\right\|^2$$

- The agent learns Advantage values instead of Q values.

Advantage Actor-Critic (A2C) IV

- That way the evaluation of an action is based not only on how good the action is, but also how much better it can be.
- This reduces high variance of policy networks and stabilises the model during training.
- Side note, Advantages also allow us to reduce variance without introducting bias in policy gradient methods (we weight a trajectory rollout of the policy with the advantage, not its return)

A2C is the parallel and asynchronous version of A2C, the there is a global network which takes the gradient of each agent, and the agent will sometimes update their states according to global network.

Asynchronous Advantage Actor-Critic (A3C) II

- The agents (or workers) are trained in parallel and update periodically a global network, which holds shared parameters.

- The updates are not happening simultaneously and thatâs where the asynchronous comes from.

DDPG (Lillicrap et al.,2015) I

Deep Deterministic Policy Gradients (DDPG) 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 (boostrapping part 1)
- uses the Q-function to learn the policy π (bootstrapping part 2)

We will next look first at the algorithm (as re-printed from Lillicrap et al, 2015) and then what ideas it contains. Note, Lillicrap 2015's notation differs from ours: θ_Q is the parameters of the value function approximation (we used w) and θ_μ is the parameters of the policy function approximation (we used θ). R is the replay buffer (we used it for the total Return).

DDPG (Lillicrap et al.,2015) I

To get DDPG to work well a few special elements were introduced:

- Exploration in continuous action spaces
  - Use Gaussian distributed noise that perturbs a mean action $\mu'(s) = \mu_\theta(s) + \mathcal{N}$ (i.e. Gaussian Policies)
  - This is almost equivalent to discrete action ϵ-greediness (only that we explore more often and much closer to the intended action μ).
- To control learning variability DDPG uses replay buffers/minibatches & target networks (just as in DQN)
- Error-based learning on the mean-squared Bellman error (MSBE)

  - MSBE (here denoted by L, tells us how closely our approximation of Q comes to satisfying the Bellman equation (compare this to function approximation Q-learning/DQN). $L = \frac{1}{N}\sum_i (y_i - Q(s_i,a_i|\theta_Q))^2$
- Policy gradient based learning on the actor network
  - Here we are using a deterministic action version of the PG theorem, which proof (not examinable here) was published in Silver et al (2014).
  - DQN updates the target network in jumps with many episodes remaining frozen. This does not work well in continuous control cases, where small changes in policy need to remain linked to small changes in control.
  - DDPG therefore uses "smooth" updates on the parameter vector θ (that is shared by both actor and critic) which updates slowly but steadily over steps t: $\theta^{t+1} = \beta\theta^t + (1-\beta)\theta^w, 0 < \beta \ll 1$

    - 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 $t+1$.

Just like the Policy Gradient theorem can be thought of a policy-centered equivalent of the Bellman Theorem, DDPG can considered the continuous action version of the discrete action DQN. Why?

Desired qualities for Reinforcement Learning

- Stable learning & little sensitivity to hyperparameters: we want to avoid parameter tuning which is (computationally) expensive and more an art than a science. Ideally, RL algorithms work out of the box.
- Sample Efficiency: Good sample complexity is the prerequisite for efficient (and in some cases effective) skill acquisition.
- Most algorithms we covered so far are relatively poor at sample efficiency (although we have seen how introducing features imporaved their stability of learning). This is ok for simulation settings or "toy"-problems such as video games.
- Many real-world applications, e.g. robots require us to put this second aspect, good exploration at the centre of our attention, as interactive learning is costly and does not scale well.

An operational definition of the sample complexity of exploration for a reinforcement learning algorithm is the number of time steps in which the algorithm does not select near-optimal actions.

SAC - Objective Function I

In the Soft Actor-Critic we use the entropy concept to boost its sample complexity. The primary goal of SAC is to maximize a modified expected cumulative reward, which includes an entropy term to encourage exploration:

$$J(\pi) = \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t)\sim\rho_\pi}\left[\sum_t \gamma^t\left(r(\mathbf{s}_t,\mathbf{a}_t) + \lambda\mathcal{H}(\pi(\cdot|\mathbf{s}_t))\right)\right] \quad (7)$$

Here,

- $J(\pi)$ is the cost-to-go objective function,
- $r(\mathbf{s}_t,\mathbf{a}_t)$ is the immediate reward function,
- $\rho_\pi = \rho_\pi(s,a)$ is the state-action distribution under policy π,
- γ is the discount factor,
- λ is a temperature parameter controlling the importance of the entropy term $\mathcal{H}$,
- $\pi(\cdot|\mathbf{s}_t)$ is the policy distribution given state $\mathbf{s}_t$.

SAC - Critic Update I

SAC employs two Q-value functions, Q_{w_1} and Q_{w_2} (cf. target networks in DDQN), to mitigate maximisation 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:

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

where $\mathcal{D}$ is the replay buffer and ρ_{dyn} represents the environment dynamics. Note $\mathbf{a}'$ comes from the current policy vπ, not the replay buffer

SAC - Critic Update II

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

The Eq. can be interpreted as follows: Across all sampled states from our experience replay buffer, decrease the squared difference between the prediction of our Q-value network and the expected prediction of the Q function plus the entropy of the policy function π measured here by the negative log of the policy function (note, not to be confused with the log π as a Policy Gradient term).

DDPG (Lillicrap et al.,2015) I

Deep Deterministic Policy Gradients (DDPG) 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 (boostrapping part 1)
- uses the Q-function to learn the policy π (bootstrapping part 2)

We will next look first at the algorithm (as re-printed from Lillicrap et al, 2015) and then what ideas it contains. Note, Lillicrap 2015's notation differs from ours: θ_Q is the parameters of the value function approximation (we used w) and θ_μ is the parameters of the policy function approximation (we used θ). R is the replay buffer (we used it for the total Return).

DDPG (Lillicrap et al.,2015) I

To get DDPG to work well a few special elements were introduced:

- Exploration in continuous action spaces
  - Use Gaussian distributed noise that perturbs a mean action $\mu'(s) = \mu_\theta(s) + \mathcal{N}$ (i.e. Gaussian Policies)
  - This is almost equivalent to discrete action ϵ-greediness (only that we explore more often and much closer to the intended action μ).
- To control learning variability DDPG uses replay buffers/minibatches & target networks (just as in DQN)
- Error-based learning on the mean-squared Bellman error (MSBE)

  - MSBE (here denoted by L, tells us how closely our approximation of Q comes to satisfying the Bellman equation (compare this to function approximation Q-learning/DQN). $L = \frac{1}{N}\sum_i (y_i - Q(s_i,a_i|\theta_Q))^2$
- Policy gradient based learning on the actor network
  - Here we are using a deterministic action version of the PG theorem, which proof (not examinable here) was published in Silver et al (2014).
  - DQN updates the target network in jumps with many episodes remaining frozen. This does not work well in continuous control cases, where small changes in policy need to remain linked to small changes in control.
  - DDPG therefore uses "smooth" updates on the parameter vector θ (that is shared by both actor and critic) which updates slowly but steadily over steps t: $\theta^{t+1} = \beta\theta^t + (1-\beta)\theta^w, 0 < \beta \ll 1$

    - 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 $t+1$.

Just like the Policy Gradient theorem can be thought of a policy-centered equivalent of the Bellman Theorem, DDPG can considered the continuous action version of the discrete action DQN. Why?

Desired qualities for Reinforcement Learning

- Stable learning & little sensitivity to hyperparameters: we want to avoid parameter tuning which is (computationally) expensive and more an art than a science. Ideally, RL algorithms work out of the box.
- Sample Efficiency: Good sample complexity is the prerequisite for efficient (and in some cases effective) skill acquisition.
- Most algorithms we covered so far are relatively poor at sample efficiency (although we have seen how introducing features imporaved their stability of learning). This is ok for simulation settings or "toy"-problems such as video games.
- Many real-world applications, e.g. robots require us to put this second aspect, good exploration at the centre of our attention, as interactive learning is costly and does not scale well.

An operational definition of the sample complexity of exploration for a reinforcement learning algorithm is the number of time steps in which the algorithm does not select near-optimal actions.

SAC - Objective Function I

In the Soft Actor-Critic we use the entropy concept to boost its sample complexity. The primary goal of SAC is to maximize a modified expected cumulative reward, which includes an entropy term to encourage exploration:

$$J(\pi) = \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t)\sim\rho_\pi}\left[\sum_t \gamma^t\left(r(\mathbf{s}_t,\mathbf{a}_t) + \lambda\mathcal{H}(\pi(\cdot|\mathbf{s}_t))\right)\right] \quad (7)$$

Here,

- $J(\pi)$ is the cost-to-go objective function,
- $r(\mathbf{s}_t,\mathbf{a}_t)$ is the immediate reward function,
- $\rho_\pi = \rho_\pi(s,a)$ is the state-action distribution under policy π,
- γ is the discount factor,
- λ is a temperature parameter controlling the importance of the entropy term $\mathcal{H}$,
- $\pi(\cdot|\mathbf{s}_t)$ is the policy distribution given state $\mathbf{s}_t$.

SAC - Critic Update I

SAC employs two Q-value functions, Q_{w_1} and Q_{w_2} (cf. target networks in DDQN), to mitigate maximisation 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:

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

where $\mathcal{D}$ is the replay buffer and ρ_{dyn} represents the environment dynamics. Note $\mathbf{a}'$ comes from the current policy vπ, not the replay buffer

SAC - Critic Update II

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

The Eq. can be interpreted as follows: Across all sampled states from our experience replay buffer, decrease the squared difference between the prediction of our Q-value network and the expected prediction of the Q function plus the entropy of the policy function π measured here by the negative log of the policy function (note, not to be confused with the log π as a Policy Gradient term).