Q-Learning

main RL method

- Return after time step t is the discounted total reward of the rest of the game

$$G_t = \sum_{t'=t+1}^{T} \gamma^{t'-t-1} R_{t'}$$

- value function: expected return in state s when we're playing according to policy π: $v_\pi(s)$
- action value function: expected return in state s, when we execute action a and then use policy π for the rest of the game: $q_\pi(s,a) = Q_\pi(s,a)$

$$v_\pi(s) = \mathbb{E}_{a \sim \pi(a|s)} [Q_\pi(s,a)]$$

- Bellmann equations for q-function:

$$\boxed{Q_\pi(s,a) = \mathbb{E}_{s',R \sim p(s',R|s,a)} [R + \gamma v_\pi(s')]}$$

Specifically, for the optimal policy $\pi^*(s) := \arg\max_a Q^*(s,a)$

$$Q^*(s,a) = \mathbb{E}_{s',R \sim p(s',R|s,a)} [R + \gamma \max_a Q^*(s',a)]$$

- for the Q-function of suboptimal policy π, we can get a new policy by

$$\pi'(s) = \arg\max_a Q_\pi(s,a)$$

one can prove that $\pi'(s)$ is never worse than $\pi(s)$ => possibility of „off-policy learning":

games played with a sub-optimal policy π can still be used as training data for an improved policy π'

=> iterate this over long training sessions => $\pi' \to \pi^*$

- goal: learn $\hat{\pi} \approx \pi^*$ and $\hat{Q} \approx Q^*$ without learning a full world model $p(s',R|s,a)$, because this is very difficult

„model-free RL"

- trick: approximate expectation in the Bellmann equations by a single instance

=> temporal difference learning

$$Q^*(s,a) = Q_\pi(s,a) + [Q^*(s,a) - Q_\pi(s,a)]$$
$$= Q_\pi(s,a) + [\mathbb{E}_p [R + \gamma V^*(s')] - Q_\pi(s,a)]$$

Expect. over p, the „world"

approximate: Expectation -> single instance
$V^*(s') \to$ our current $V_\pi(s')$

$$Q^*(s,a) \approx Q_\pi(s,a) + [R + \gamma V_\pi(s') - Q_\pi(s,a)]$$

⇒ Learning rule: add a fraction of the correction (as in stochastic gradient descent)

$$Q'(s,a) \leftarrow Q_\pi(s,a) + \alpha\left[R + \gamma V_\pi(s') - Q_\pi(s,a)\right]$$

α: learning rate, hyperparameter

· algorithm:

① collect training data with the current policy π

② for all moves in TS: perform above update

Q-Learning Algorithm

⓪ initialisation $Q^{(0)}(s,a)$ (arbitrary, but good guess speeds up convergence)

important: $Q^{(0)}(s = \text{terminal state}) = 0$

① outer loop: $\tau = 1, \dots, E$ (play many games)

② inner loop: $t = 1, \dots, T_\tau$ (time steps of game τ)

Ⓘ decide about next move according to

- ε-greedy policy

$$a_{\tau,t} \sim \begin{cases} \arg\max_a Q^{(\tau-1)}(s_{\tau,t}, a) & \text{with prob } 1-\varepsilon \\ \text{uniform}(a) \text{ (random)} & \text{with prob } \varepsilon \end{cases}$$

($s_{\tau,t}$: current state)

- softmax policy

$$a_{\tau,t} \sim \text{softmax}_\beta\left(Q^{(\tau-1)}\right)$$

with hyperparam β (= inverse temperature)

Ⓛ for all $t = 1, \dots, T_\tau$: update Q-function

$$Q^{(\tau)}(s_{\tau,t}, a_{\tau,t}) \leftarrow Q^{(\tau-1)}(s_{\tau,t}, a_{\tau,t}) + \alpha\left[R_{\tau,t+1} + \gamma V^{(\tau-1)}(s_{\tau,t+1}) - Q^{(\tau-1)}(s_{\tau,t}, a_{\tau,t})\right]$$

② final policy: $\hat{\pi}(s) = \arg\max_a Q^{(E)}(s,a)$

two major variants: what's the current guess for $V^{(\tau-1)}(s_{\tau,t+1})$

· Q-learning: $V^{(\tau-1)}(s) = \max_a Q^{(\tau-1)}(s,a)$ (use move we should have done)

· SARSA algorithm: $V^{(\tau-1)}(s_{\tau,t+1}) = Q^{(\tau-1)}(s_{\tau,t+1}, a_{\tau,t+1})$ (use move we actually executed at time $t+1$)

- more accurate variant for updates: use actual rewards of multiple time steps:

n-step Q-learning

$$Q^{(\tau)}(s_{\tau,t}, a_{\tau,t}) \leftarrow Q^{(\tau-1)}(s_{\tau,t}, a_{\tau,t}) + \alpha \left[\sum_{t'=t+1}^{t+n} \gamma^{t'-t-1} R_{t'} + \gamma^n V^{(\tau-1)}(s_{\tau,t+n}) - Q^{(\tau-1)}(s_{\tau,t}, a_{\tau,t}) \right]$$

- in practice: maintain an experience buffer = collection of moves from all games so far (possibly pruned to keep size manageable)

  => update step (6) uses a random batch from the experience buffer instead of just the previous game

  => better, especially when the batch is sampled with non-uniform probabilities: prefer instances where the current guess $Q^{(\tau-1)}$ is very bad

$$\left[\left(G_{\tau,t} - Q^{(\tau-1)}(s_{\tau,t}, a_{\tau,t}) \right)^2 \right] >> 0$$

$G_{\tau,t}$: actual return; $Q^{(\tau-1)}(s_{\tau,t}, a_{\tau,t})$: current guess

Problem: If implemented naively, $\hat{Q}(s,a)$ is a huge matrix of size $|S| \times |A|$ (# of states x # of actions)

- matrix cannot be stored in praxis
- we cannot collect enough training data to fit all matrix elements

} analogeously to problems of multidimensional histograms

How to do Q-learning in practice?

(1) easy possibility: group states into k clusters where action values are similar => $\hat{Q}(cl, a)$ has only size $k \times |A|$

with handcrafted or learned clustering (good actions should be equal for all members of a cluster)

(2) reduce it to a regression problem: $\hat{y} = f(x)$

two possibilities:

I: y is a vector of length $|A|$, holding the expected return for every action possible in state $x = s$

II: y is a scalar, giving the expected return of the move $x = (s,a)$

- define response ($y[a]$ for possibility I or y for possibility II)

  - MC (Monte Carlo) value estimation:

$$y_{\tau t} = \sum_{t'=t+1}^{\infty} \gamma^{t'-t-1} R_{\tau,t'}$$ (use actual observed rewards)

- TD Q-learning: $y_{\tau,t} = R_{\tau,t+1} + \gamma \max_a Q^{(\tau-1)}(s_{\tau,t+1}, a)$

↑ temporal difference

- n-step TD Q-learning: $y_{\tau,t} = \sum_{t'=t+1}^{t+n} \gamma^{t'-t-1} R_{\tau,t'} + \gamma^n \max_a Q^{(\tau-1)}(s_{\tau,t+n}, a)$

- SARSA: $y_{\tau,t} = R_{\tau,t+1} + \gamma Q^{(\tau-1)}(s_{\tau,t+1}, a_{\tau,t+1})$

- define X: I: hand crafted feature detector $X = \psi(s)$

II: use $X = s$ and have the regression <u>learn</u> optimal features (especially effective when $f(x)$ is a neural network)

- Simplest possibility: <u>linear regression</u>

$Q(s,a) = \psi(s) \cdot \beta_a$ (one weight vector for each action a)

training by batch gradient descent:

① sample batch of training instances where action a was performed

② update $\beta_a \leftarrow \beta_a + \frac{\alpha}{N_{batch}} \sum_{\tau,t \in batch} \underbrace{\psi(s_{\tau,t})^T (y_{\tau,t} - \psi(s_{\tau,t}) \cdot \beta_a)}_{\text{derivative of squared loss}}$

- only works when features $\psi(s)$ are very expressive

=> use standard feature extraction methods (PCA, ICA, LLE) in combination with hand-designed preprocessing

=> exploit symmetries of the game [Bomberman: mirroring of s or rotation of s by 90°, 180°, 270° are also legal states]

- more powerful: <u>regression forest</u>

- can train one forest for each action a or a single forest for outputting a vector of action values

=> can learn non-linear mappings from state representation $\psi(s) \to y$

=> still needs good features $\psi(s)$, but less so than linear regression

- Deep Q-learning: typically use $X = s$ (no feature extraction) and learn optimal features in the first layers of a NN (entire network determines values as before)

- example: ATARI game agents by DeepMind: X = pixels of the current video frame ≙ same as human players

- process this with a "convolutional neural network"

- advantage: learned features are superior to hand-crafted ones

- disadvantage: find a good network architecture and train to convergence can take quite long => often does not finish by final project deadline

---

Reward shaping

- auxiliary rewards during training when official rewards are very sparse

- idea: define intermediate goals for agent

  e.g. chess: taking an opponents piece is usually good, but not always (may be a trap or sacrifice)

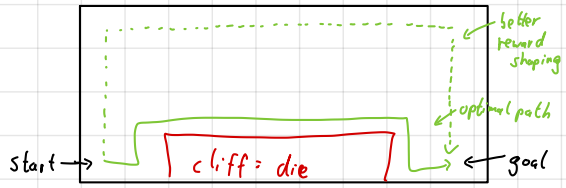
=> reward shaping must be done with care, otherwise the agent may converge to a bad policy

- theory: modify the rewards by a potential function $R'_{t,t+1} = R_{t,t+1} + \overbrace{F(S_{t,t}, S_{t,t+1})}^{\text{reward shaping}}$

$F(S_{t,t}, S_{t,t+1}) = \gamma \Phi(S_{t,t+1}) - \Phi(S_{t,t})$ $\Phi(s)$ must not depend on the action

optimal theoretical choice: $\Phi(s) = v^*(s)$, but:

- we do not know $v^*(s)$
- is not very good during exploration



bei dem optimal path kann jede random action zum Tod führen
↳ viele Fehlversuche, darum gestrichelten Pfad generieren