Reinforcement learning for large state spaces ("RL in practice")

- so far: by learning state-value function $Q^*(s,a)$, we can derive an optimal policy: $\pi^*(s) = \arg\max_a Q^*(s,a)$
- problem: Q^* was represented as a table of size $|s| \times |a|$

  this may be impractical: - too big to be stored explicitly
  - impossible to get enough training data to fill each entry (≙ analogous to multi-dimensional histogram)

$\Rightarrow$ learn Q^* (or rather: $\hat{Q}$ approx.) by regression:
- generalize from events that we have seen in TS to events we have not seen, provided there is sufficient similarity
- define features:
  - $X \equiv s$ (or $X \equiv o$ in partially observable problems) and have regression alg. figure out what is important in s

    example: deep Q-learning of Atari games (famous paper by DeepMind): s is content of the screen, $X \equiv s$, deep neural network automatically learns more informative features from the raw pixel values
  - $X = \Psi(s)$ or $X = \Psi(o)$ with some hand-crafted or learned feature extraction function $\Psi(.)$
- define response: $Y = \mathbb{E}\left[\sum_{t'=t}^{\infty} \gamma^{t'-t-1} r_{t'}\right]$ expected return

examples: - MC value estimation: $Y_{\tau t} = \sum_{t'=t+1}^{\infty} \gamma^{t'-t-1} \underbrace{R_{\tau,t'}}$

actually observed rewards in time step t' of episode τ ($\tau \hat{=}$ game index, $t \hat{=}$ move index)

- TD (temporal difference) Q-learning

$$Y_{\tau,t} = R_{\tau,t} + \gamma \max_a Q(s_{\tau,t+1}, a)$$

current guess of Q-function

- n-step TD Q-learning

$$Y_{\tau,t} = \sum_{t'=t+1}^{t+n} \gamma^{t'-t-1} R_{\tau,t'} + \gamma^n \max_a Q(s_{\tau,t+n}, a)$$

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

- optimization algorithm: alternating optimization

repeat until convergence:

1. compute target response $Y_{\tau,t}$ using current guess for Q
2. optimize Q using squared loss

$\Rightarrow$ new guess $Q' = \arg\max_{\hat{Q}} \sum_{TS} \mu(s_{\tau,t}) \left(Y_{\tau,t} - \hat{Q}(s_{\tau,t}, a_{\tau,t})\right)^2$

weights

usually solved by batch gradient descent

random subset of TS — improve our current by updating parameters

- common update strategies: – "online": apply a gradient step after every move
  - "episodic": gradient step(s) after every game
  - "offline": TS is an "experience buffer" of size K
    (when new experiences arrive (= moves are executed), new data are added to TS and old possibly removed (if K would otherwise be exceeded))

elements in TS: $(s_{\tau,t}, a_{\tau,t}, s_{\tau,t+1}, r_{\tau,t+1})$

in each update round ("experience replay"):
- sample a batch of size K' from TS
- compute $X_{\tau,t}, Y_{\tau,t}$ for selected elements
- perform gradient descent with squared loss

variant: prioritized experience replay:
- sample batch of size $K'' > K'$
- compute $X_{\tau,t}, Y_{\tau,t}$ and residual $Y_{\tau,t} - Q(X_{\tau,t}, a_{\tau,t})$
- only keep K' instances with biggest squared residual $\hat{=} \mu(s_{\tau,t})$ big for difficult decisions
- gradient descent with squared loss

- alternating optimization: keep two models: old guess $Q \Rightarrow$ compute $Y_{\tau,t}$
  working guess $Q' \Rightarrow$ apply updates to
  every once in a while, update $Q \Leftarrow Q'$ (frequency of this is a hyper-parameter)

Models for Q-function regression:

- linear value approximation

$$Q(X(s), a) = X(s) \cdot \beta_a$$ for each action: one parameter vector β_a

this only works if we define good features $X = X(s)$

(generally a hard engineering problem)

training updates: training batch $B = (B_{a_1}, \ldots, B_{a_N})$ ← sub batch for each action

⇒ gradient update:

$$\beta_a \leftarrow \beta_a + \underbrace{\alpha}_{\text{training rate (hyper param)}} \frac{1}{\underbrace{N_{B_a}}_{\text{size of subbatch } B_a}} \sum_{\tau, t \in B_a} X(s_{\tau,t})^T \left(Y_{\tau t} - \underbrace{X}_{\text{fixed feature extraction function}}(s_{\tau t}) \cdot \beta_a \right)$$

- feature definition: usual preprocessing methods like PCA and clustering and their kernel-version can be applied before manual feature design (or sometimes instead)
- (monte carlo) tree search: roll-out each game (from current state) to the end with random actions ⇒ game out-come gives information on the current state
- data augmentation: exploit the symmetries of a game: in Bomberman, rotating by 90° or mirroring are still valid game states
  ⇒ each training move gives information about 8 game states
  (alternative: define features to be invariant under rotation and mirroring)

- Regression Forest: two possibilities 1) use as input $(X, a) \Rightarrow$ predict $Q(X, a)$
  2) use as input X and predict a vector ~~of~~ values, ~~for~~ one entry for each action
  - problem: updating an existing forest is difficult (on-line learning of RF open probl.)
    $\Rightarrow$ learn a new RF from scratch in every iteration & ~~...~~ batch size must be big enough to get good results ↯
    $\Rightarrow$ in ensemble of trees: use different batches for every tree, ~~or~~ can even keep some trees from the current forest and ~~update~~ retrain only the rest
- Gradient Boosting: alternative method to create an ensemble:
  - RF: each tree is independent of all other trees
  - GB: a new member of the ensemble is trained to correct the mistakes of the ensemble so far $\Rightarrow \underbrace{h_{i+1}}_{\text{new member}} = f(TS, \{h_i\}_{i=1}^{i})$
    conditioned on old members, not independent
  - ~~modified~~ modification to alternating optimization alg.:
    instead of doing gradient updates on the existing model, train a new ensemble member on the current residuals, keeping existing ensemble fixed
    - current residuals $\delta_{\tau,t} = Y_{\tau t} - \underbrace{Q(s_{\tau t}, a_{\tau t})}_{\text{current ensemble response}}$
      $h_{i+1} = \arg\min_h \sum_{\text{batch}} p_{\tau t} (\delta_{\tau,t} - h(s_{\tau t}, a_{\tau t}))^2$
    $\Rightarrow$ new ensemble $Q'(s, a) = Q(s, a) + \underbrace{w_{i+1}}_{\text{weight}} h_{i+1}(s, a)$

advantage: h_i can be very simple non-linear regressors, e.g.

- decision stumps ($\hat{=}$ decision trees with very small depth, e.g. 1)
- rounding of a linear model (sign$(X\beta)$)

weight schedule: $w_i = \dfrac{w_0}{1+\mu \cdot i}$: new ensemble members get lower weight

- Deep Q-learning (DQN $\hat{=}$ deep Q network)
  - doesn't define features manually, but sets $X \equiv S$ (or $X \equiv O$) and have the first layers of the network learn good features
  - $Q(s, a)$ is represented as $Q_\Theta(s, a)$ with Θ: the parameters of the neural network
  - gradient updates:

value estimates compared with stored guess $Q_{\Theta'}$

$$\Theta \leftarrow \Theta - \alpha \nabla_\Theta \left[\frac{1}{|B|} \sum_{\tau, t \in B} \left(Y_{\tau t} - Q_\Theta(s_{\tau t}, a_{\tau t}) \right)^2 \right]$$

current parameters

use autodiff in pytorch or tensorflow

batch as before

(use a stored older guess to compute $Y_{\tau t} = r_{\tau t+1} + \gamma\, Q_{\Theta'}(s_{\tau, t+1}, a_{\tau, t+1})$ SARSA

$Y_{\tau t} = r_{\tau, t+1} + \gamma \max_a Q_{\Theta'}(s_{\tau, t+1}, a)$ Q-learning

every once in a while, update $\Theta' \leftarrow \Theta$)

Training strategies

- self play: by playing with your current agent against itself or an older version of itself, create arbitrary amount of training data
  - crucial: ensure that ~~ac~~ action selection is partially random to avoid bad convergence to local optima ⇒ do not use greedy policy $\pi(s) = \arg\max_a Q(s,a)$

    common choice: - softmax policy $\pi(s) \sim \text{softmax}_a\, Q(s,a)$

    $$\text{softmax}(u) = \frac{e^{u_i}}{\sum_{i'} e^{u_{i'}}} \quad \text{or with "temperature"} \quad \frac{e^{u_i/S}}{\sum_{i'} e^{u_{i'}/S}}$$

    $$\hat{=} \text{ discrete}_a(\exp(Q(s,a)))$$

    ($S = 0$ ⇒ greedy policy, $S \Rightarrow \infty$ uniform random actions)

  - IAUU exploration: improved ε-greedy strategy

    $$a \sim \begin{cases} \arg\max_a Q(s,a) & \text{with prob } 1-\varepsilon \\ \underbrace{\text{softmax}(-\text{count}(s,a)/S)}_{\text{prefer actions we haven't tried much in the past}} & \text{with prob } \varepsilon \end{cases}$$

- reward shaping: often, rewards are sparse ($\hat{=}$ rare, e.g. just win/loss at the end of each episode)
  ⇒ hard to learn, slow convergence
  - idea: add auxiliary ~~a~~ rewards during training to speed-up convergence
    but: bad auxiliary rewards lead to bad policies

common failure modes: - infinite loop of positive local awards

A=start	B
C=monster	D=treasure

$r_{A \to B} = r_{B \to A} = 1$

$\Rightarrow$ agent can make infinite return by going in an infinite loop $A \leftrightarrow B$

- reward gaming: intended effect of the auxiliary reward is put to rediculous extremes

  ex: robot should learn to stretch out its arm $\Rightarrow$ auxiliary reward for placing an object far away from the "body"

  unintended consequenc: robot learns to throw objects instead of placing them softly

- theory: potential-based reward shaping is necessary and sufficient for the optimal policy to remain invariant

  shaped rewards: $r'_{\tau,t+1} = r_{\tau,t+1} + F(s_{\tau,t}, s_{\tau,t+1})$

$$F(s_{\tau,t}, s_{\tau,t+1}) = \gamma\, \phi(s_{\tau,t+1}) - \phi(s_{\tau,t})$$

potential: only depends on a single state and independent of actions

- under a greedy policy $\phi(s) = V^*(s)$ (optimal value function) is best
- But: in an exploration/exploitation setting during training, this may lead to many losses (eaten by monster): to avoid this, the auxiliary rewards should be biased towards "safe" behavior $\Rightarrow$ good $\phi(s)$ are an "art".