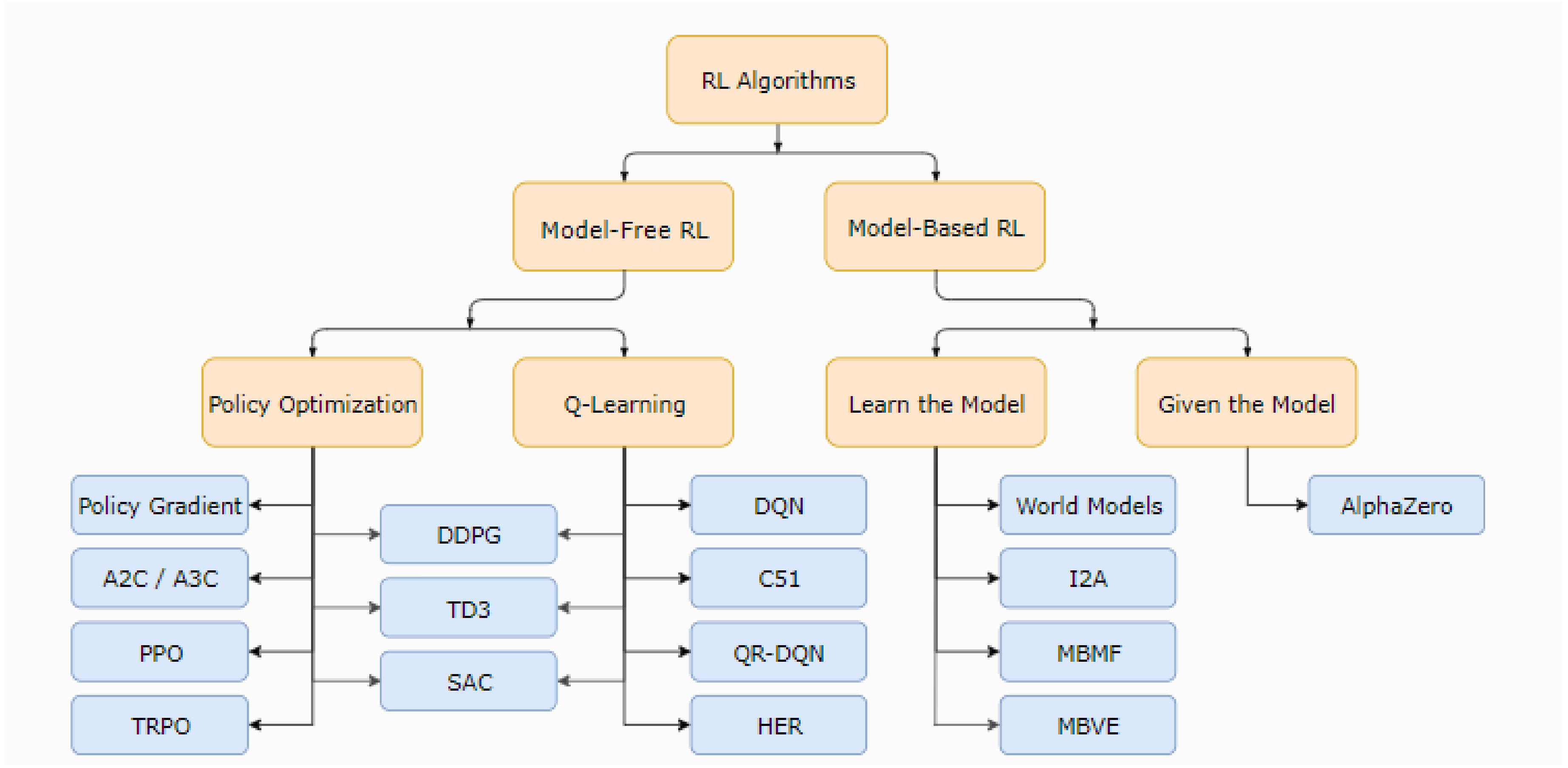
UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR TECHNISCHE INFORMATIK

RL Algorithms

- Zakaria Narjis
- Anouar Nechi

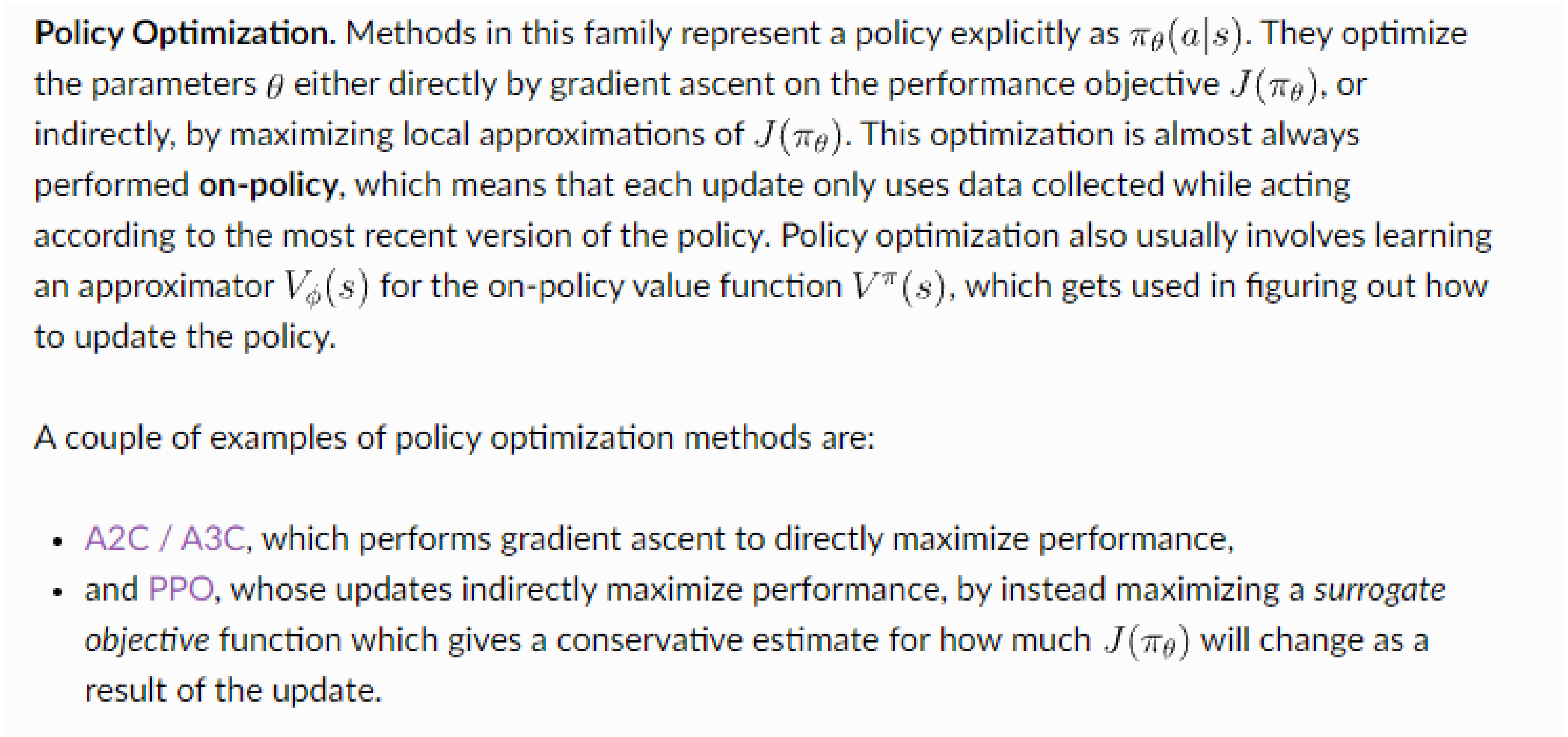
Table of contents

Taxonomy of RL algorithms



What to Learn in Model-Free RL

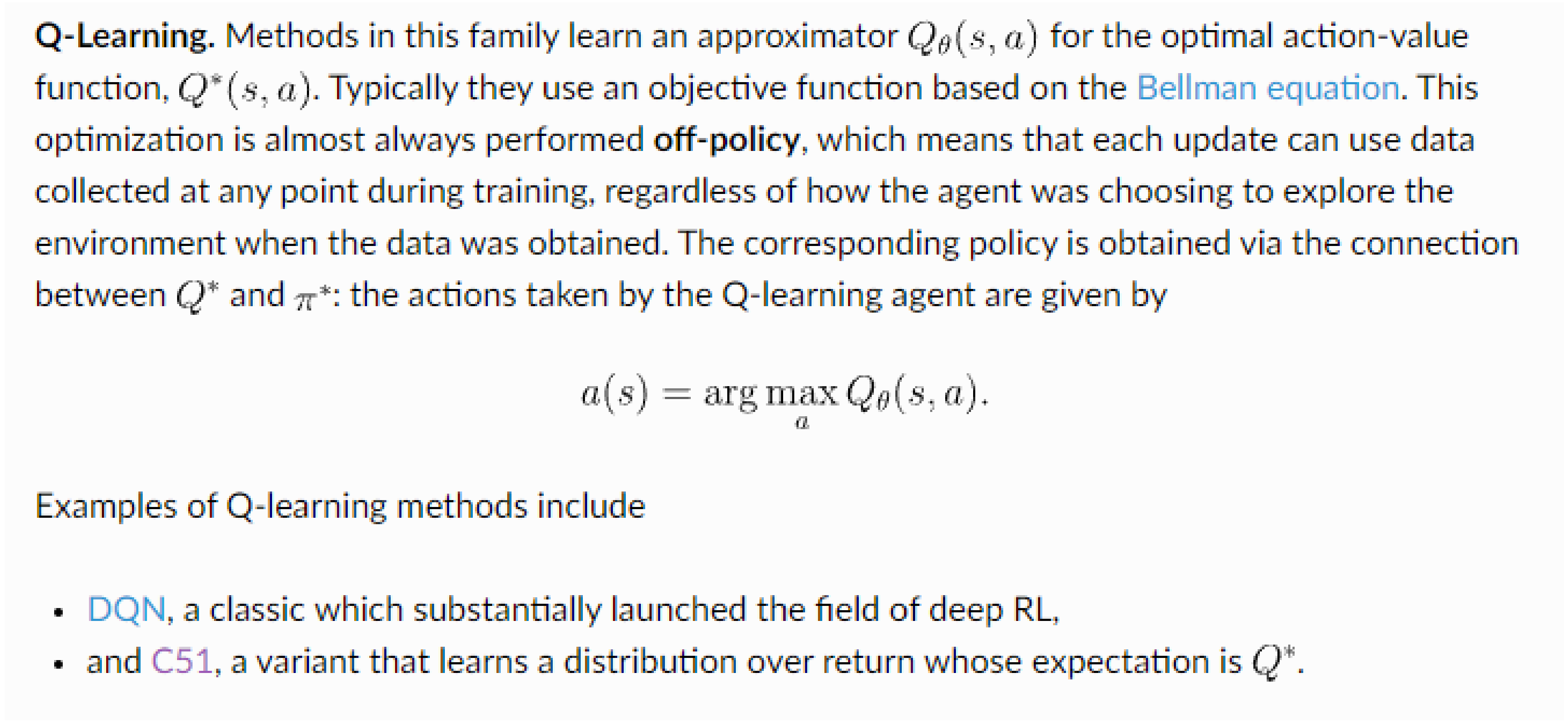
There are two main approaches to representing and training agents with model-free RL:



Policy Optimization. Methods in this family represent a policy explicitly as $\pi_\theta(a|s)$. They optimize the parameters θ either directly by gradient ascent on the performance objective $J(\pi_\theta)$, or indirectly, by maximizing local approximations of $J(\pi_\theta)$. This optimization is almost always performed **on-policy**, which means that each update only uses data collected while acting according to the most recent version of the policy. Policy optimization also usually involves learning an approximator $V_\phi(s)$ for the on-policy value function $V^\pi(s)$, which gets used in figuring out how to update the policy.

A couple of examples of policy optimization methods are:

- A2C / A3C, which performs gradient ascent to directly maximize performance,
- and PPO, whose updates indirectly maximize performance, by instead maximizing a *surrogate objective* function which gives a conservative estimate for how much $J(\pi_\theta)$ will change as a result of the update.

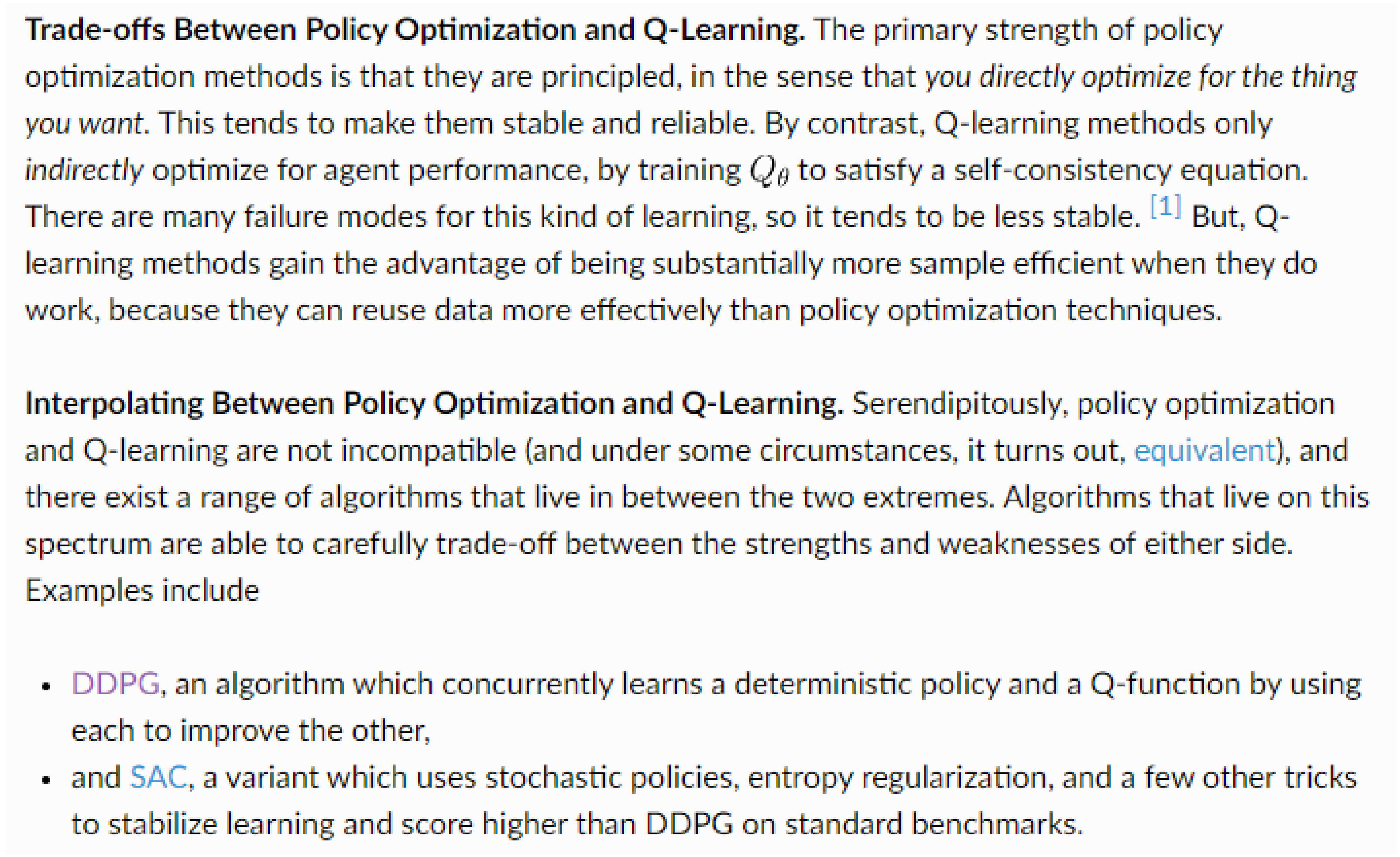


Q-Learning. Methods in this family learn an approximator $Q_\theta(s,a)$ for the optimal action-value function, $Q^*(s,a)$. Typically they use an objective function based on the Bellman equation. This optimization is almost always performed **off-policy**, which means that each update can use data collected at any point during training, regardless of how the agent was choosing to explore the environment when the data was obtained. The corresponding policy is obtained via the connection between Q^* and π^*: the actions taken by the Q-learning agent are given by

$$a(s) = \arg\max_a Q_\theta(s,a).$$

Examples of Q-learning methods include

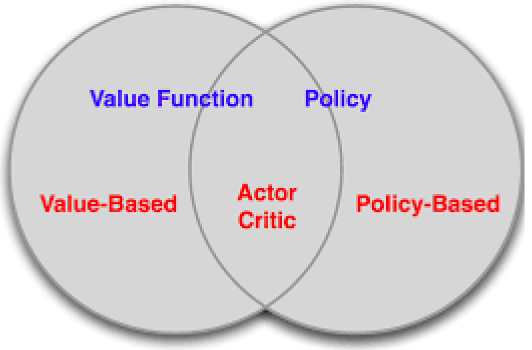
- DQN, a classic which substantially launched the field of deep RL,
- and C51, a variant that learns a distribution over return whose expectation is Q^*.



Trade-offs Between Policy Optimization and Q-Learning. The primary strength of policy optimization methods is that they are principled, in the sense that *you directly optimize for the thing you want*. This tends to make them stable and reliable. By contrast, Q-learning methods only *indirectly* optimize for agent performance, by training Q_θ to satisfy a self-consistency equation. There are many failure modes for this kind of learning, so it tends to be less stable. [1] But, Q-learning methods gain the advantage of being substantially more sample efficient when they do work, because they can reuse data more effectively than policy optimization techniques.

Interpolating Between Policy Optimization and Q-Learning. Serendipitously, policy optimization and Q-learning are not incompatible (and under some circumstances, it turns out, equivalent), and there exist a range of algorithms that live in between the two extremes. Algorithms that live on this spectrum are able to carefully trade-off between the strengths and weaknesses of either side. Examples include

- DDPG, an algorithm which concurrently learns a deterministic policy and a Q-function by using each to improve the other,
- and SAC, a variant which uses stochastic policies, entropy regularization, and a few other tricks to stabilize learning and score higher than DDPG on standard benchmarks.



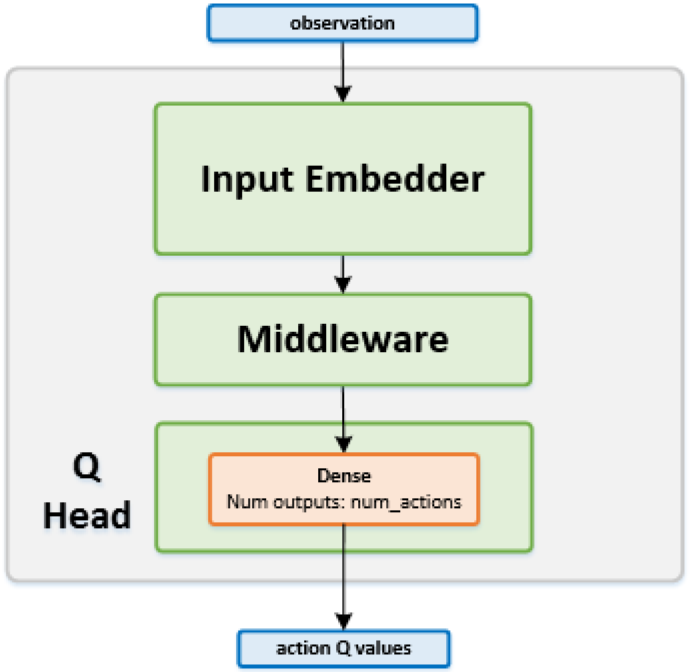
Algorithms

Classification of algorithms

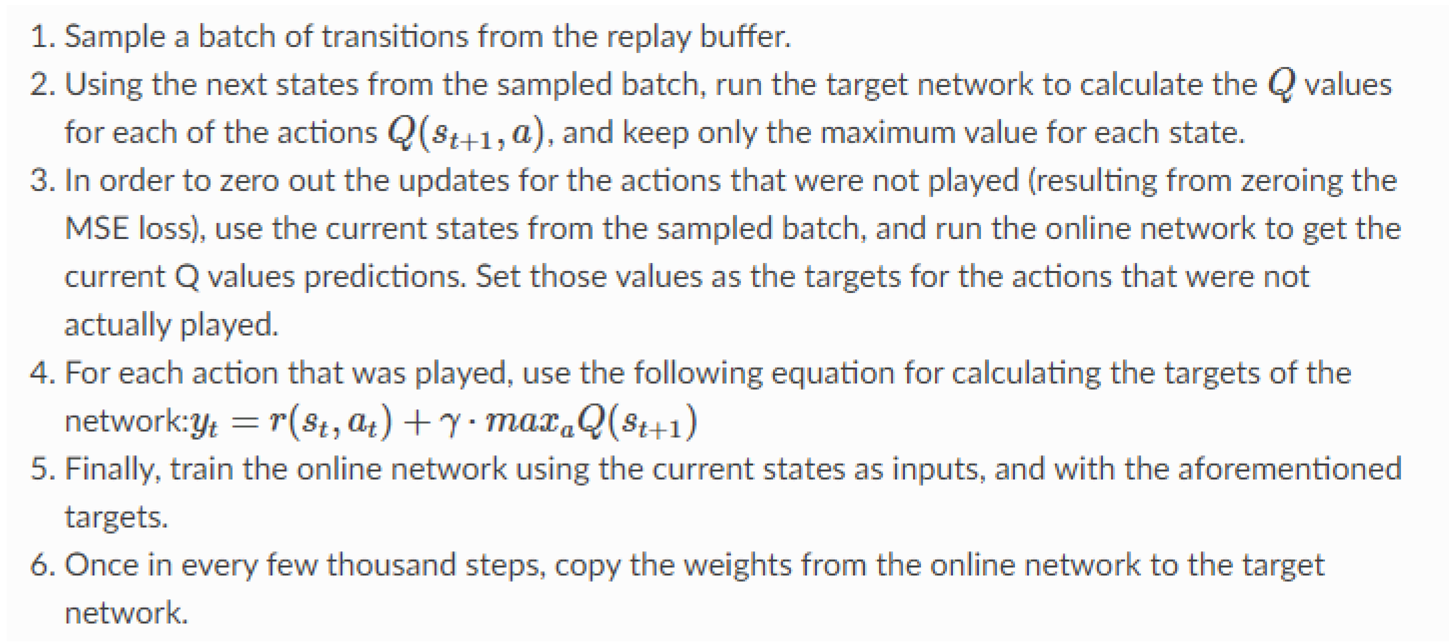
	Value Based	Policy Based	Actor-Critic
On-Policy	• Monte Carlo Learning (MC) • TD(0) • SARSA • Expected SARSA • n-Step TD/SARSA • TD(λ)	• REINFORCE • REINFORCE with Advantage	• A3C • A2C • TRPO • PPO
Off-Policy	• Q-Learning • DQN • Double DQN • Dueling DQN		• DDPG • TD3 • SAC • IMPALA

DQN

Network Structure



Training the network



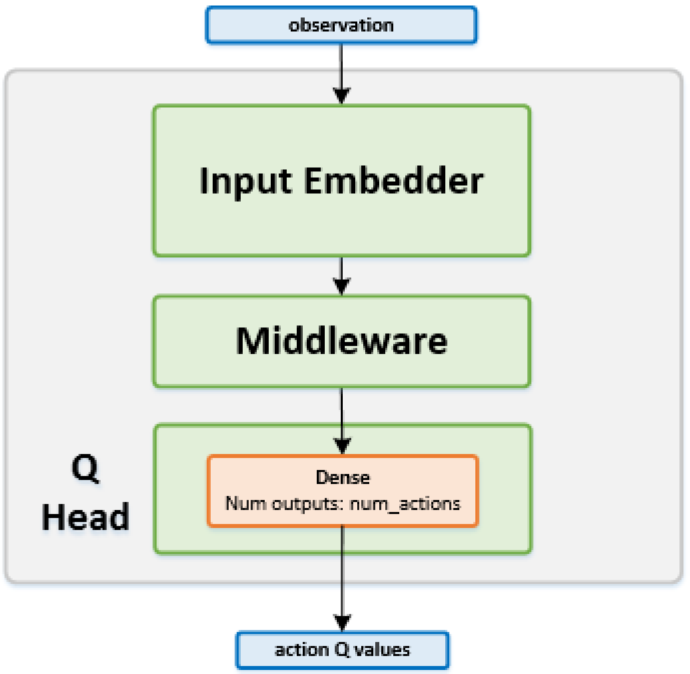
1. Sample a batch of transitions from the replay buffer.
2. Using the next states from the sampled batch, run the target network to calculate the Q values for each of the actions $Q(s_{t+1}, a)$, and keep only the maximum value for each state.
3. In order to zero out the updates for the actions that were not played (resulting from zeroing the MSE loss), use the current states from the sampled batch, and run the online network to get the current Q values predictions. Set those values as the targets for the actions that were not actually played.
4. For each action that was played, use the following equation for calculating the targets of the network:$y_t = r(s_t, a_t) + \gamma \cdot max_a Q(s_{t+1})$
5. Finally, train the online network using the current states as inputs, and with the aforementioned targets.
6. Once in every few thousand steps, copy the weights from the online network to the target network.

DDQN(double DQN)

A **Double Deep Q-Network**, or **Double DQN** utilizes Double Q-learning to reduce overestimation by decomposing the max operation in the target into action selection and action evaluation. We evaluate the greedy policy according to the online network, but we use the target network to estimate its value.

DDQN(double DQN)

Network Structure



Training the network

1. Sample a batch of transitions from the replay buffer.
2. Using the next states from the sampled batch, run the online network in order to find the Q maximizing action $argmax_a Q(s_{t+1}, a)$. For these actions, use the corresponding next states and run the target network to calculate $Q(s_{t+1}, argmax_a Q(s_{t+1}, a))$.
3. In order to zero out the updates for the actions that were not played (resulting from zeroing the MSE loss), use the current states from the sampled batch, and run the online network to get the current Q values predictions. Set those values as the targets for the actions that were not actually played.
4. For each action that was played, use the following equation for calculating the targets of the network: $y_t = r(s_t, a_t) + \gamma \cdot Q(s_{t+1}, argmax_a Q(s_{t+1}, a))$
5. Finally, train the online network using the current states as inputs, and with the aforementioned targets.
6. Once in every few thousand steps, copy the weights from the online network to the target network.

Dueling DQN

A **Dueling Network** is a type of Q-Network that has two streams to separately estimate (scalar) state-value and the advantages for each action. Both streams share a common convolutional feature learning module. The two streams are combined via a special aggregating layer to produce an estimate of the state-action value function Q as shown in the figure to the right.

$$Q(s,a,\theta,\alpha,\beta) = V(s,\theta,\beta) + \left(A(s,a,\theta,\alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a';\theta,\alpha)\right)$$

This formulation is chosen for identifiability so that the advantage function has zero advantage for the chosen action, but instead of a maximum we use an average operator to increase the stability of the optimization.

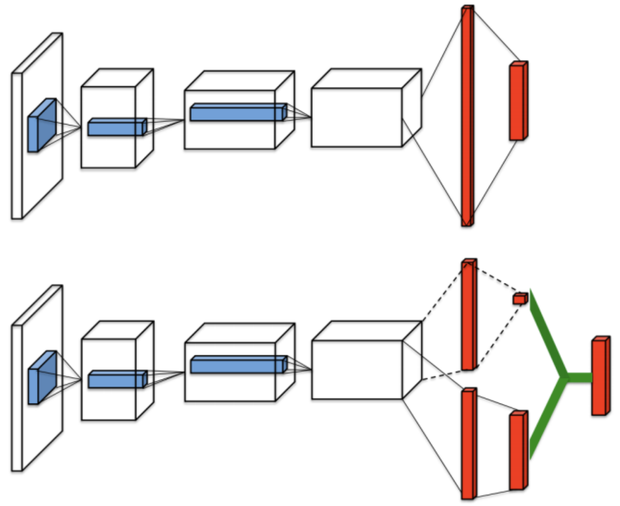
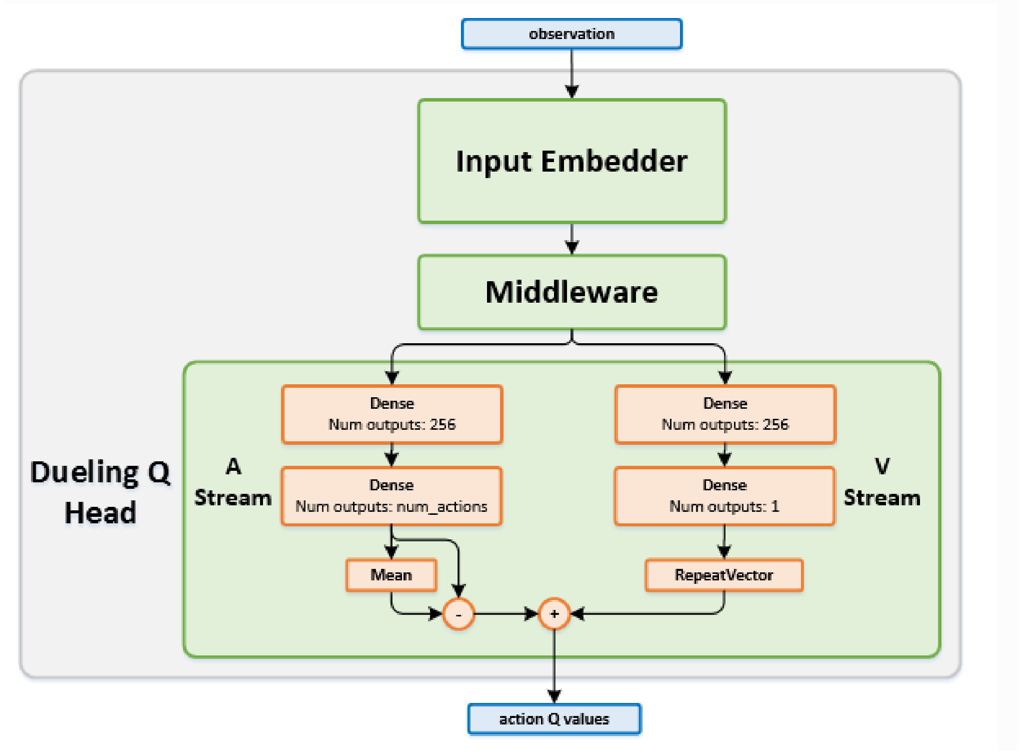


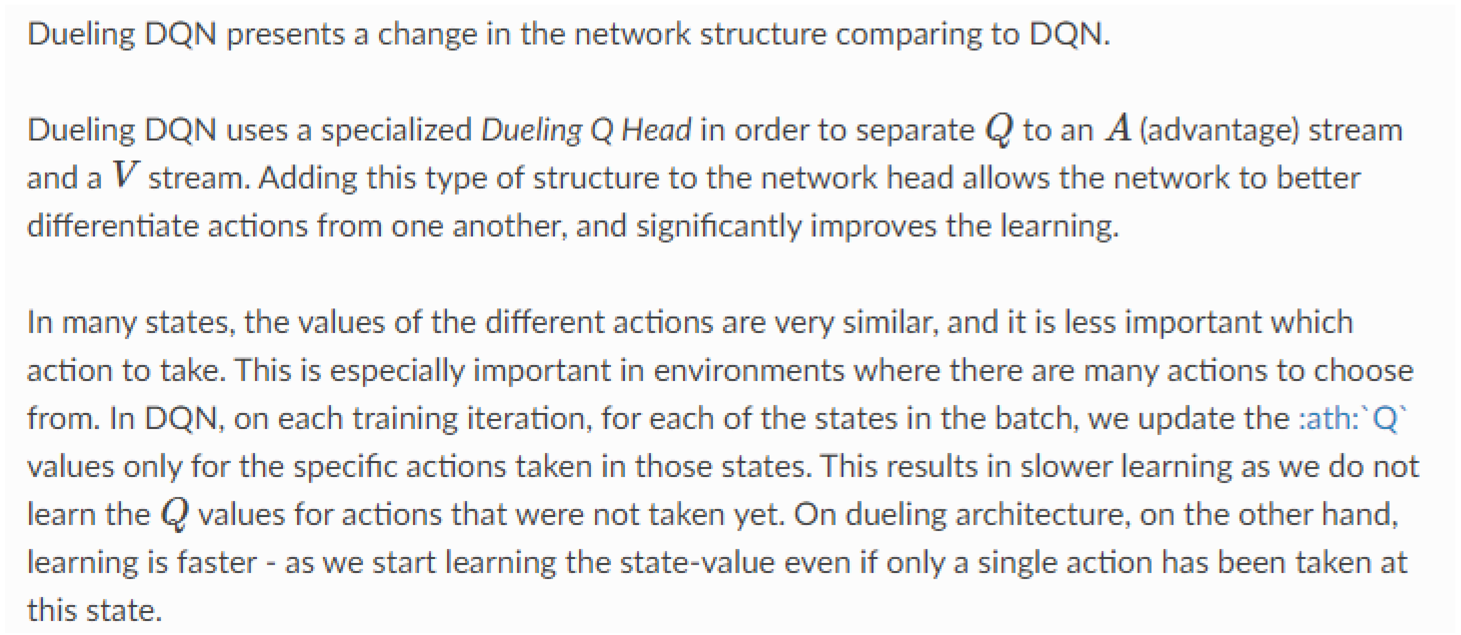
Figure 1. A popular single stream Q-network (**top**) and the dueling Q-network (**bottom**). The dueling network has two streams to separately estimate (scalar) state-value and the advantages for each action; the green output module implements equation (9) to combine them. Both networks output Q-values for each action.

Dueling DQN

Network Structure



Training the network



Dueling DQN presents a change in the network structure comparing to DQN.

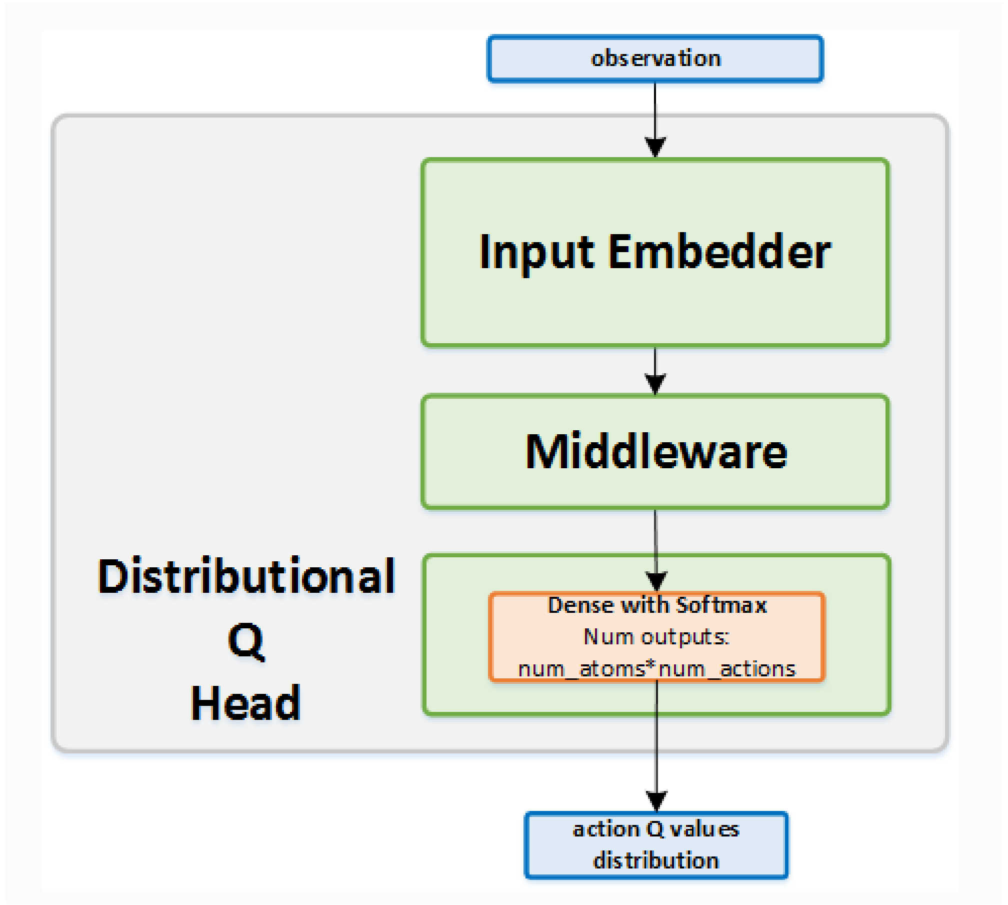
Dueling DQN uses a specialized *Dueling Q Head* in order to separate Q to an A (advantage) stream and a V stream. Adding this type of structure to the network head allows the network to better differentiate actions from one another, and significantly improves the learning.

In many states, the values of the different actions are very similar, and it is less important which action to take. This is especially important in environments where there are many actions to choose from. In DQN, on each training iteration, for each of the states in the batch, we update the :ath:`Q` values only for the specific actions taken in those states. This results in slower learning as we do not learn the Q values for actions that were not taken yet. On dueling architecture, on the other hand, learning is faster - as we start learning the state-value even if only a single action has been taken at this state.

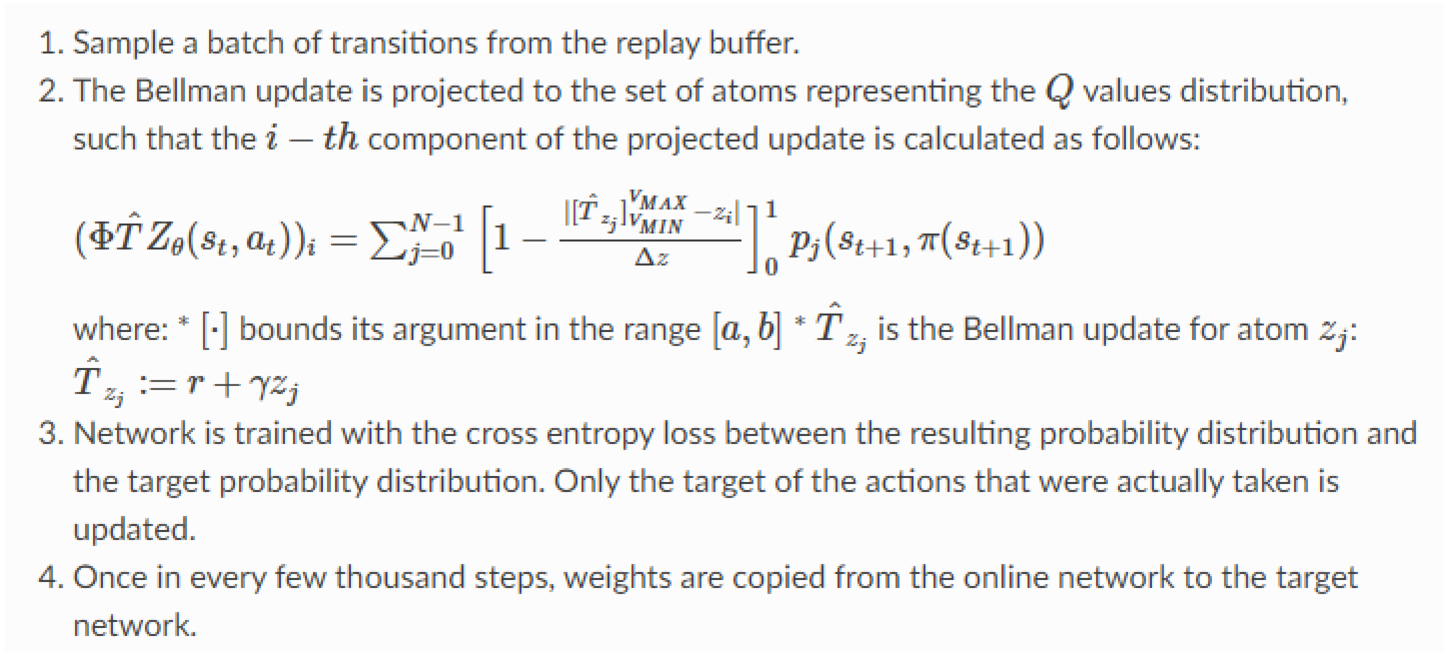
UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR TECHNISCHE INFORMATIK

Categorical DQN

Network Structure



Training the network



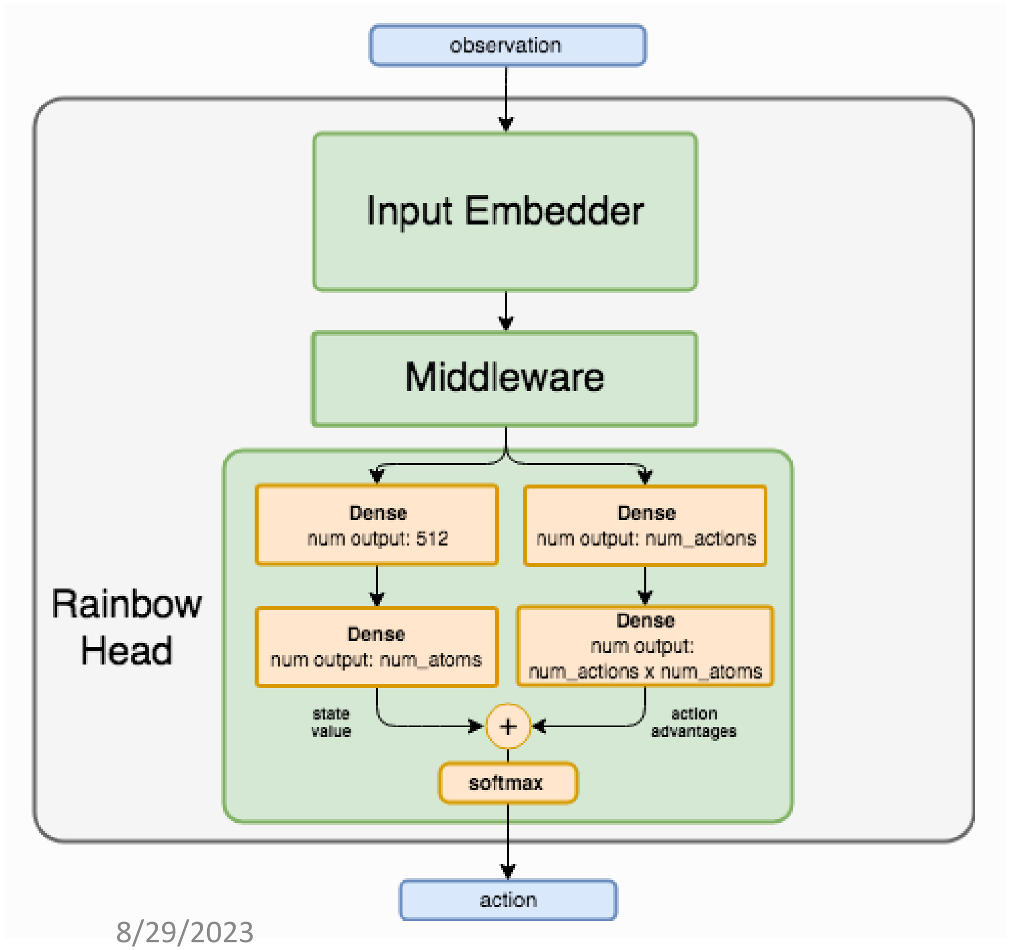
1. Sample a batch of transitions from the replay buffer.
2. The Bellman update is projected to the set of atoms representing the Q values distribution, such that the $i-th$ component of the projected update is calculated as follows:

   $$(\Phi\hat{T}Z_\theta(s_t,a_t))_i = \sum_{j=0}^{N-1}\left[1-\frac{|[\hat{T}_{z_j}]_{V_{MIN}}^{V_{MAX}}-z_i|}{\Delta z}\right]_0^1 p_j(s_{t+1},\pi(s_{t+1}))$$

   where: * $[\cdot]$ bounds its argument in the range $[a,b]$ * $\hat{T}_{z_j}$ is the Bellman update for atom z_j: $\hat{T}_{z_j} := r+\gamma z_j$
3. Network is trained with the cross entropy loss between the resulting probability distribution and the target probability distribution. Only the target of the actions that were actually taken is updated.
4. Once in every few thousand steps, weights are copied from the online network to the target network.

Rainbow

Network Structure



Rainbow combines 6 recent advancements in reinforcement learning:

- N-step returns
- Distributional state-action value learning
- Dueling networks
- Noisy Networks
- Double DQN
- Prioritized Experience Replay

Training the network

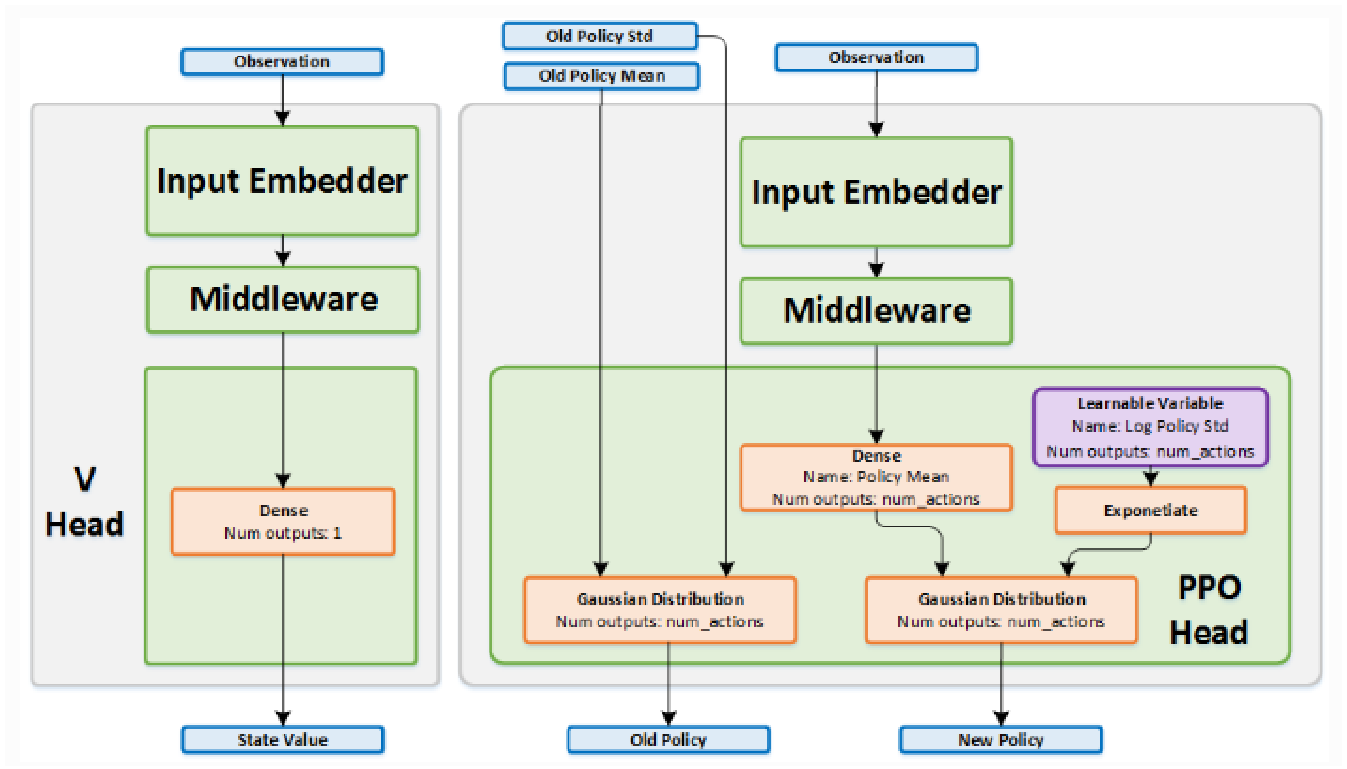
1. Sample a batch of transitions from the replay buffer.
2. The Bellman update is projected to the set of atoms representing the Q values distribution, such that the $i-th$ component of the projected update is calculated as follows:

   $$(\Phi\hat{T}Z_\theta(s_t,a_t))_i = \sum_{j=0}^{N-1}\left[1-\frac{|[\hat{T}_{z_j}]_{V_{MIN}}^{V_{MAX}}-z_i|}{\Delta z}\right]_0^1 p_j(s_{t+1},\pi(s_{t+1}))$$

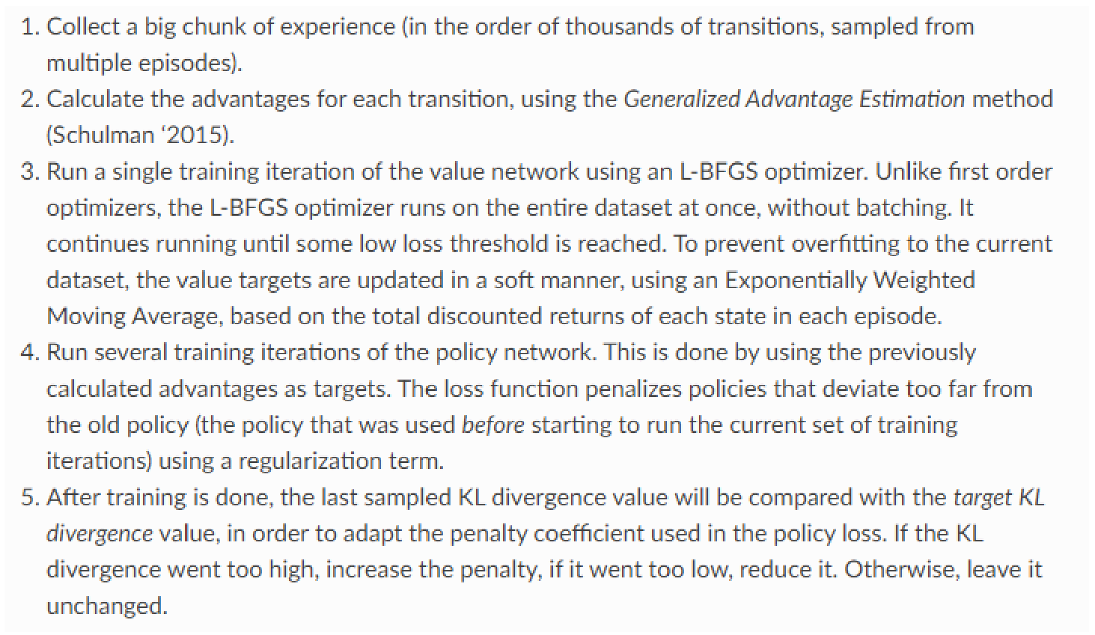
   where: * $[\cdot]$ bounds its argument in the range $[a,b]$ * $\hat{T}_{z_j}$ is the Bellman update for atom z_j: $\hat{T}_{z_j} := r_t + \gamma r_{t+1} + \ldots + \gamma r_{t+n-1} + \gamma^{n-1} z_j$
3. Network is trained with the cross entropy loss between the resulting probability distribution and the target probability distribution. Only the target of the actions that were actually taken is updated.
4. Once in every few thousand steps, weights are copied from the online network to the target network.
5. After every training step, the priorities of the batch transitions are updated in the prioritized replay buffer using the KL divergence loss that is returned from the network.

Proximal Policy Optimization (PPO)

Network Structure

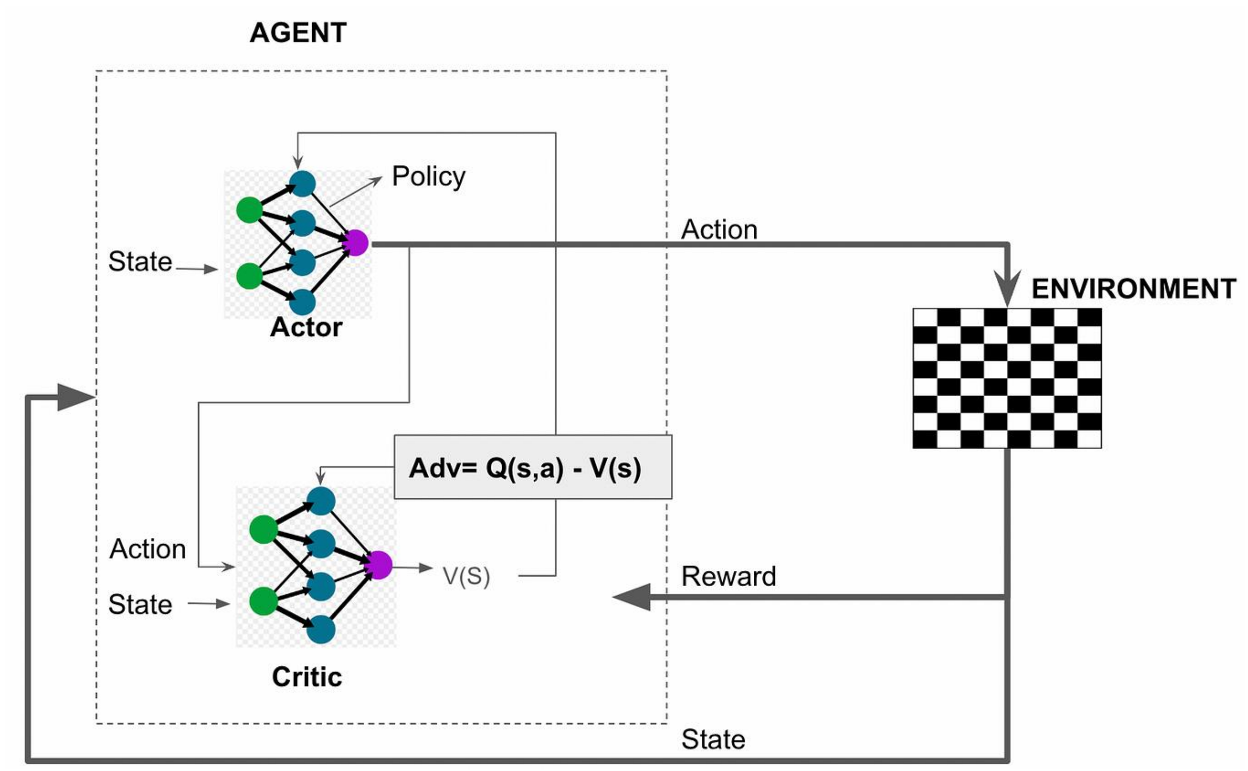


Training the network

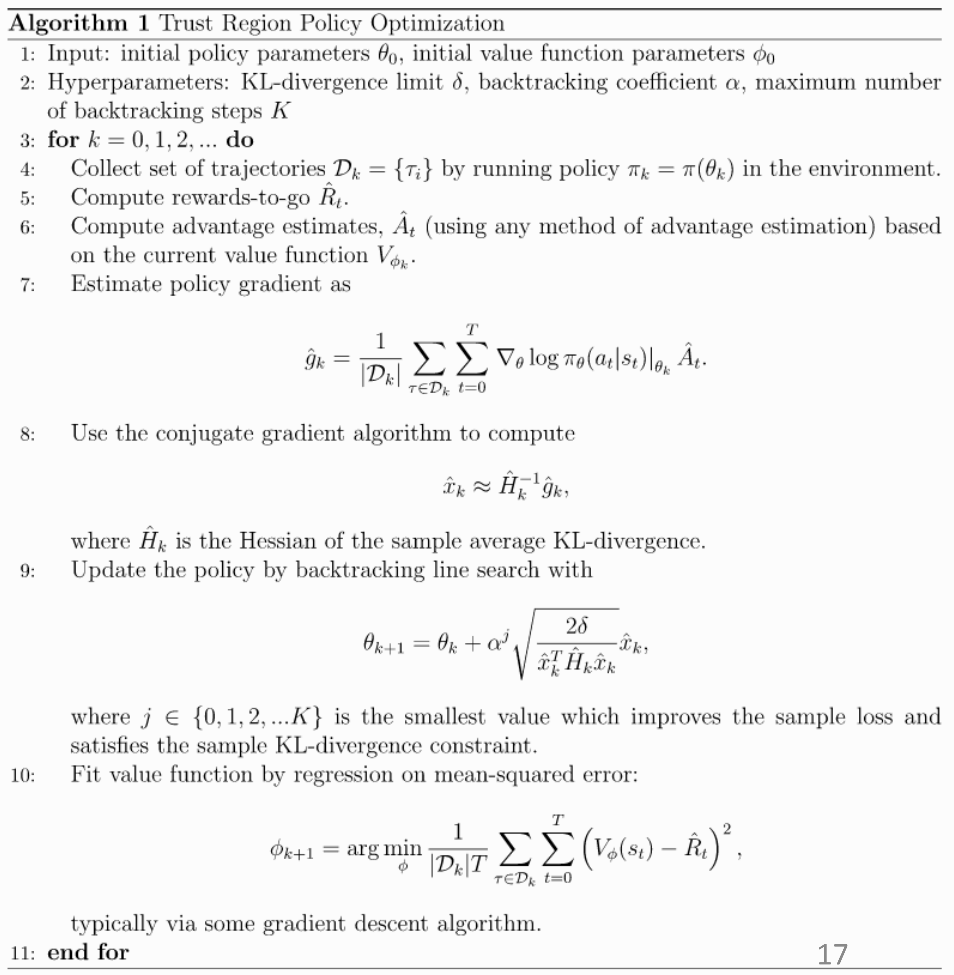

1. Collect a big chunk of experience (in the order of thousands of transitions, sampled from multiple episodes).
2. Calculate the advantages for each transition, using the *Generalized Advantage Estimation* method (Schulman '2015).
3. Run a single training iteration of the value network using an L-BFGS optimizer. Unlike first order optimizers, the L-BFGS optimizer runs on the entire dataset at once, without batching. It continues running until some low loss threshold is reached. To prevent overfitting to the current dataset, the value targets are updated in a soft manner, using an Exponentially Weighted Moving Average, based on the total discounted returns of each state in each episode.
4. Run several training iterations of the policy network. This is done by using the previously calculated advantages as targets. The loss function penalizes policies that deviate too far from the old policy (the policy that was used *before* starting to run the current set of training iterations) using a regularization term.
5. After training is done, the last sampled KL divergence value will be compared with the *target KL divergence* value, in order to adapt the penalty coefficient used in the policy loss. If the KL divergence went too high, increase the penalty, if it went too low, reduce it. Otherwise, leave it unchanged.

Trust region policy optimization (TRPO)

Network Structure



Training the network

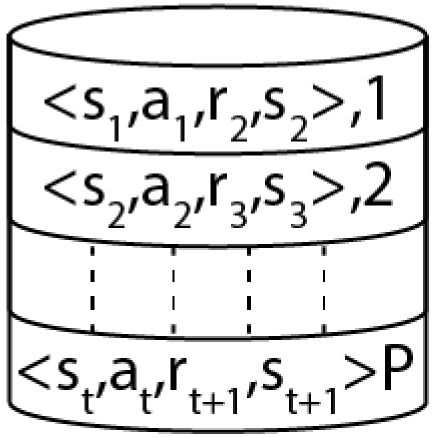


```
Algorithm 1 Trust Region Policy Optimization
1: Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
2: Hyperparameters: KL-divergence limit $\delta$, backtracking coefficient $\alpha$, maximum number of backtracking steps $K$
3: for $k = 0, 1, 2, ...$ do
4:     Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
5:     Compute rewards-to-go $\hat{R}_t$.
6:     Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function $V_{\phi_k}$.
7:     Estimate policy gradient as
           $$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$
8:     Use the conjugate gradient algorithm to compute
           $$\hat{x}_k \approx \hat{H}_k^{-1} \hat{g}_k,$$
       where $\hat{H}_k$ is the Hessian of the sample average KL-divergence.
9:     Update the policy by backtracking line search with
           $$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{x}_k^T \hat{H}_k \hat{x}_k}} \hat{x}_k,$$
       where $j \in \{0, 1, 2, ...K\}$ is the smallest value which improves the sample loss and satisfies the sample KL-divergence constraint.
10:    Fit value function by regression on mean-squared error:
           $$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left( V_\phi(s_t) - \hat{R}_t \right)^2,$$
       typically via some gradient descent algorithm.
11: end for
```

Other key concepts

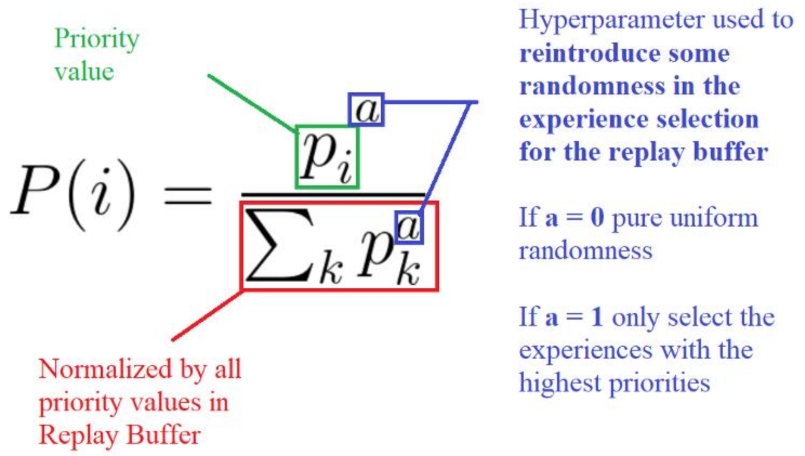
Experience Replay

Experience Replay is a replay memory technique used in reinforcement learning where we store the agent's experiences at each time-step, $e_t = (s_t, a_t, r_t, s_{t+1})$in a data-set $D = e_t$,... e_N, pooled over many episodes into a replay memory. We then usually sample the memory randomly for a minibatch of experience, and use this to learn off-policy, as with Deep Q-Networks. This tackles the problem of autocorrelation leading to unstable training, by making the problem more like a supervised learning problem.



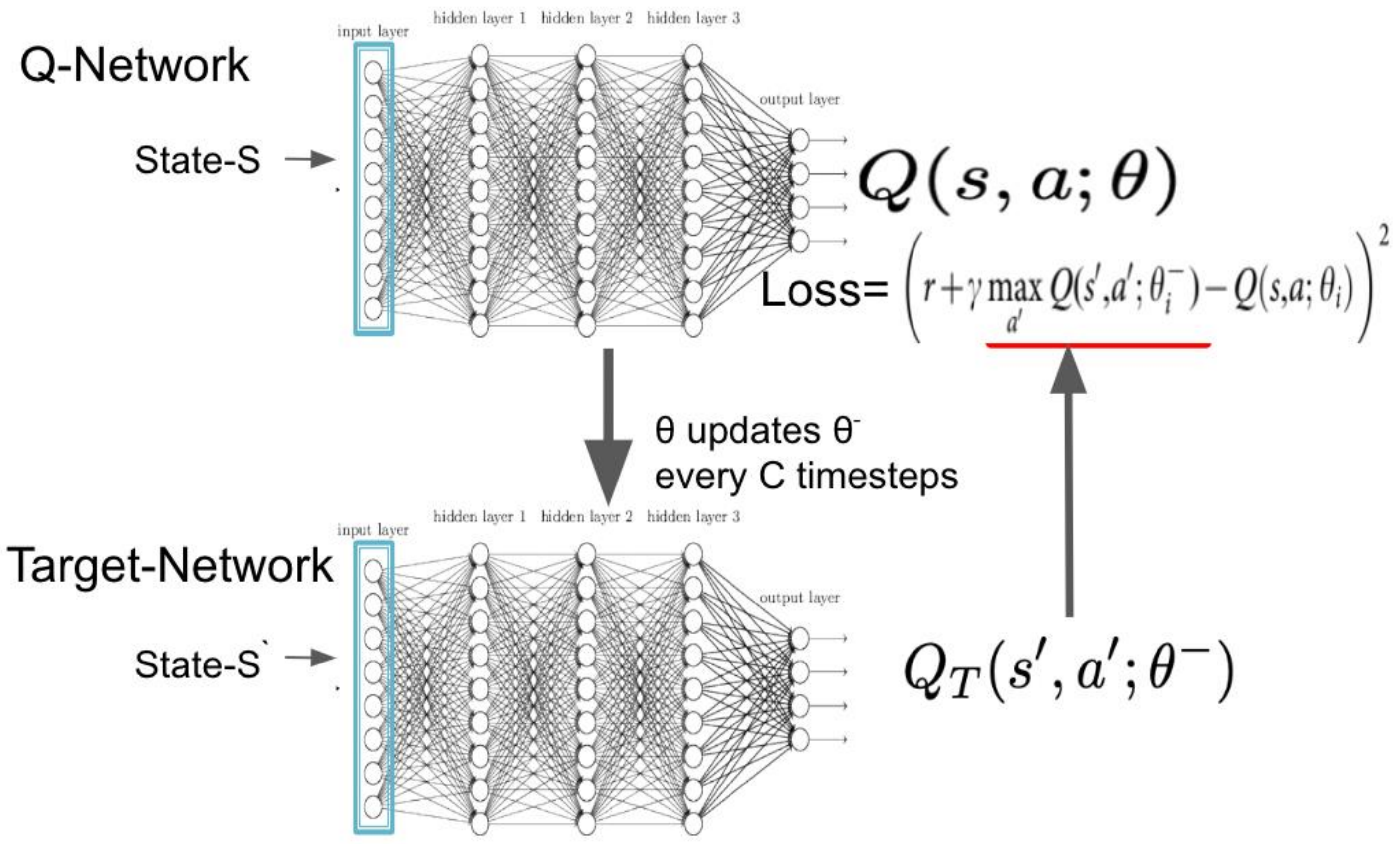
Prioritized Experience Replay

Prioritized Experience Replay is a type of experience replay in reinforcement learning where we more frequently replay transitions with high expected learning progress, as measured by the magnitude of their temporal-difference (TD) error. This prioritization can lead to a loss of diversity, which is alleviated with stochastic prioritization, and introduce bias, which can be corrected with importance sampling.

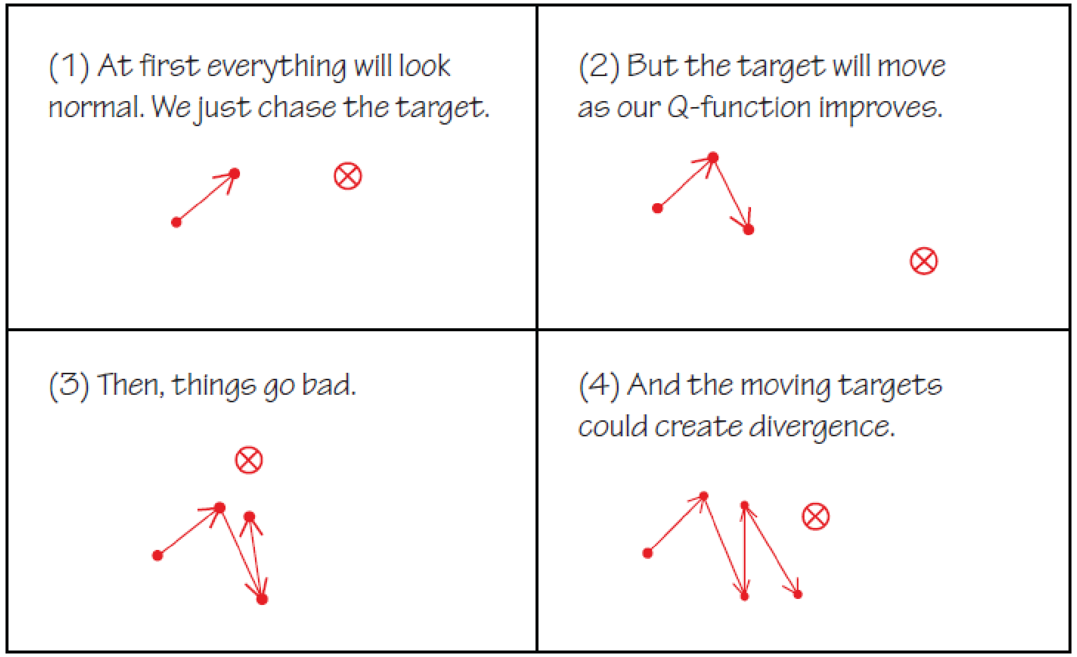

Priority value
$P(i) = \frac{p_i^a}{\sum_k p_k^a}$
Normalized by all priority values in Replay Buffer
Hyperparameter used to **reintroduce some randomness in the experience selection for the replay buffer**
If **a = 0** pure uniform randomness
If **a = 1** only select the experiences with the highest priorities

where $P(i) > 0$ is the priority of transition i. The exponent a determines how much prioritization is used, with $a = 0$corresponding to the uniform case.

Target network



Target network



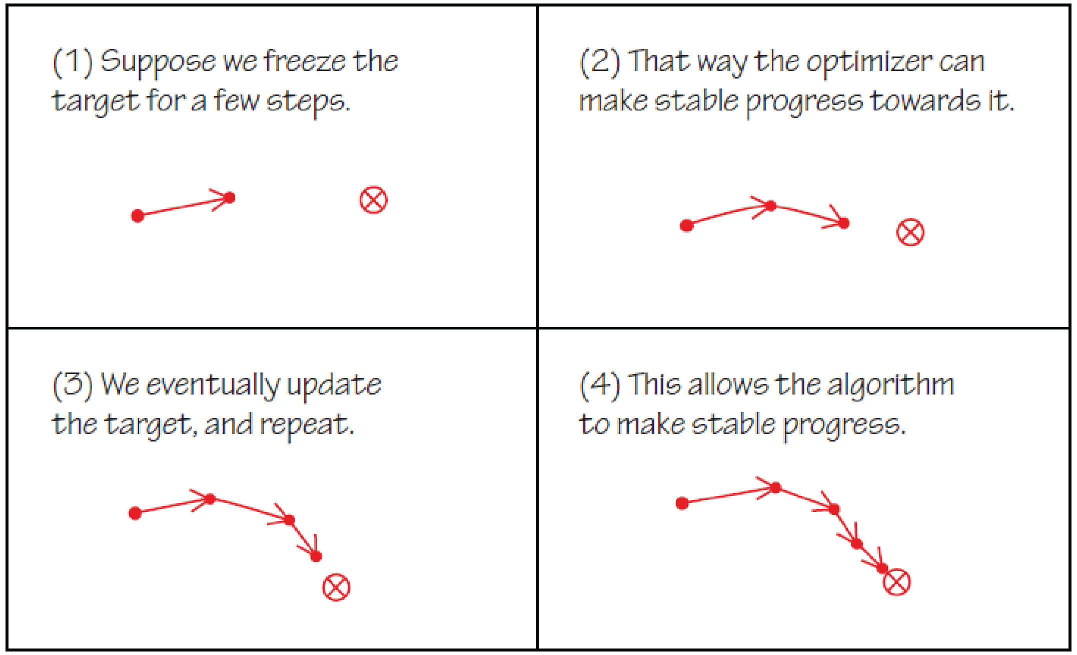
(1) At first everything will look normal. We just chase the target.

(2) But the target will move as our Q-function improves.

(3) Then, things go bad.

(4) And the moving targets could create divergence.

Without target network



(1) Suppose we freeze the target for a few steps.

(2) That way the optimizer can make stable progress towards it.

(3) We eventually update the target, and repeat.

(4) This allows the algorithm to make stable progress.

With target network

Key papers in Deep RL

https://spinningup.openai.com/en/latest/spinningup/keypapers.html

Ressources

- https://spinningup.openai.com/en/latest/spinningup/rl_intro.html (educational resource produced by OpenAI)
- https://intellabs.github.io/coach/components/agents/index.html (educational resource and RL library produced by Intel)