**Deep Learning**
Summer semester ‘24

# 9. Deep Reinforcement Learning

Slide credit: Slides in parts adapted from the lecture at the PRL @ FAU Erlangen-Nürnberg (K. Breininger, T. Würfl, A. Maier, V. Christlein).

# Recap

- Basics of Reinforcement Learning;
- Only **discrete** MDPs;
- **Today:**

**Continuous** and **high-dimensional** MDPs!

# Outline

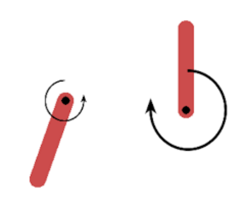
# Continuous MDPs - 1

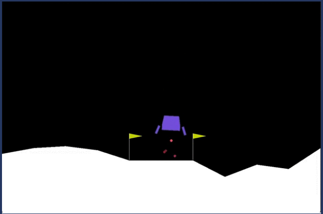
- **Discrete** MDPs:
  - $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \iota, \gamma \rangle$;
  - **discrete** state space $\mathcal{S} = \{1, \dots, N_s\} \subseteq \mathbb{N}$;
  - **discrete** action space $\mathcal{A} = \{1, \dots, N_a\} \subseteq \mathbb{N}$.
- **Continuous** MDPs:
  - $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \iota, \gamma \rangle$
  - **continuous** state space $\mathcal{S} \subseteq \mathbb{R}^d$;
  - **discrete** action space $\mathcal{A} = \{1, \dots, N_a\} \subseteq \mathbb{N}$;
  - or continuous action space $\mathcal{A} \subseteq \mathbb{R}^m$.

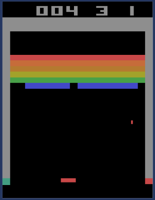
**Not every problem can be formalized as a discrete MDP!**

# Continuous MDPs - 2

**In this lecture:** MDPs with **continuous** states, **discrete** actions;








**In future lectures:** MDPs with continuous states and actions.

# Outline

# Value-function approximation - 1

- Value-function **cannot** be computed as a table;
  - too **many** states/actions to store in memory;
  - too **slow** to update each state.
- Commonly used function **approximators**:
  - Linear function;
  - Neural network;
  - Regression tree;
  - Gaussian process;
  - ...

# Value-function approximation - 2

- Use **parametric** value function estimators where the parameters are expressed as the weight vector $\boldsymbol{w} \in \mathbb{R}^d$

$$\hat{V}_{\boldsymbol{w}}(s) \approx V^{\pi}(s) \qquad \hat{Q}_{\boldsymbol{w}}(s, a) \approx Q^{\pi}(s, a);$$

- Parameters $\boldsymbol{w}$ are much **fewer** than the states (that are potentially infinite!);
- Changing weights affects the **accuracy** of the estimate of **multiple** states;
- **Improving** the accuracy of the value-function estimate of one state, may **decrease** the accuracy of the estimate of others.

# Mean squared value error

- **Accuracy** is measured as the *mean squared value error*:

$$\overline{\mathrm{VE}}(\boldsymbol{w}) \triangleq \sum_{s\in\mathcal{S}} \mu(s) \left[ V^{\pi}(s) - \hat{V}_{\boldsymbol{w}}(s) \right]^2 ,$$

with the state distribution $\mu(s) \geq 0$;

- $\sum_s \mu(s) = 1$ weighs the importance of the estimate error for each state $s$;
- For **on-policy** algorithms, $\mu(s)$ is the fraction of time spent in state $s$ while following policy $\pi$.

# Value estimation with stochastic gradient descent

- Assume the **exact** value-function $V^\pi(s)$ is known $\forall s \in \mathcal{S}$;
- **Goal:** find approximation with a **differentiable** estimator $\hat{V}_{\mathbf{w}}(s)$;
- Value-function is updated each discrete time step $t = 0, 1, 2, \ldots;$
- Define weights vector $\mathbf{w}_t \triangleq (w_{1_t}, w_{2_t}, \ldots, w_{d_t})^T$
- A basic **step-based** update of $V$:

$$\begin{aligned} \mathbf{w}_{t+1} &\triangleq \mathbf{w}_t - \frac{1}{2}\alpha\nabla\left[V^\pi(s_t) - \hat{V}_{\mathbf{w}_t}(s_t)\right]^2 \\ &= \mathbf{w}_t + \alpha\left[V^\pi(s_t) - \hat{V}_{\mathbf{w}_t}(s_t)\right]\nabla\hat{V}_{\mathbf{w}_t}(s_t) \end{aligned} \tag{1}$$

where $\alpha > 0$ is the *learning rate* and

$$\nabla\hat{V}_{\mathbf{w}_t}(s_t) \triangleq \left(\frac{\partial\hat{V}_{\mathbf{w}_t}(s_t)}{\partial w_1}, \frac{\partial\hat{V}_{\mathbf{w}_t}(s_t)}{\partial w_2}, \ldots, \frac{\partial\hat{V}_{\mathbf{w}_t}(s_t)}{\partial w_d}\right); \tag{2}$$

- **Convergence** to local optimum if properly decaying $\alpha$.

# Semi-gradient SARSA - 1

- On-policy **control** algorithm: approximate $Q$ instead of $V$;
- **Gradient descent** update:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \left[ U_t - \hat{Q}_{\boldsymbol{w}_t}(s_t, a_t) \right] \nabla \hat{Q}_{\boldsymbol{w}_t}(s_t, a_t); \tag{3}$$

- **One-step** SARSA update:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \left[ r_{t+1} + \gamma \hat{Q}_{\boldsymbol{w}_t}(s_{t+1}, a_{t+1}) - \hat{Q}_{\boldsymbol{w}_t}(s_t, a_t) \right] \nabla \hat{Q}_{\boldsymbol{w}_t}(s_t, a_t). \tag{4}$$

# Semi-gradient SARSA - 2

**Algorithm** Semi-gradient SARSA

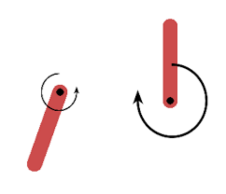
```
Input: the policy π to evaluate;
Input: a differentiable function Q̂ : S × A → ℝ;
Initialize action-value-function weights w ∈ ℝ^d arbitrarily, e.g., w = 0;
while true do
    Initialize first state s = s_0 and action a = a_0 (e.g., ε-greedy);
    while true do
        if s' is terminal then
            w ← w + α[r − Q̂_w(s, a)]∇Q̂_w(s, a);
            Terminate the episode;
        end if
        Choose action a' as a function of Q̂_w(s', ·) (e.g., ε-greedy);
        w ← w + α[r + γQ̂_w(s', a') − Q̂_w(s, a)]∇Q̂_w(s, a);
        s ← s';
        a ← a';
    end while
end while
```

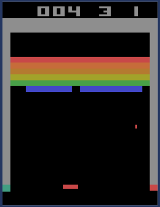
Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

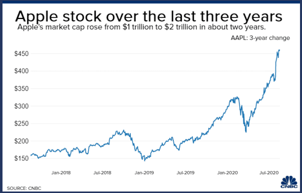
# Outline

# Feature construction for linear methods

- State vector contains representative **features** of the problem;
- For example:
  - position and angular velocity for balancing a pendulum;
  - pixels for playing a videogame;
  - stock price for finance applications.





# Feature construction for linear methods

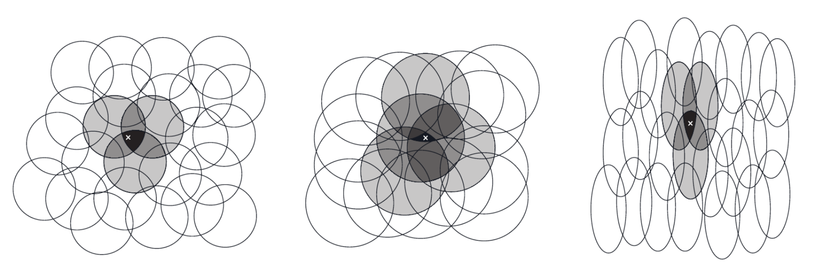
- Basic features may **not be representative** enough to capture complex behavior;
- Consider the pendulum example:
  - state $s = (s_1, s_2)$, with $s_1 \in \mathbb{R}$ the angular position and $s_2 \in \mathbb{R}$ the angular velocity;
  - a feature vector $\varphi(s) = (s_1, s_2)^T$ is a **poor** representation of the problem;
  - **interaction** between the state dimensions are not considered!
- **Representative** feature vectors consider all dimensions of the state and their (potentially complex) **interaction**;
- How to obtain **good** features?

# Polynomial features

- Polynomial features capture interaction among state dimensions by **multiplication**:
  - 1st-order: $\varphi(s) = (1, s_1, s_2, s_1 s_2)^T$;
  - 2nd-order: $\varphi(s) = (1, s_1, s_2, s_1 s_2, s_1^2, s_2^2, s_1 s_2^2, s_1^2 s_2, s_1^2 s_2^2)^T$;
  - ...
- The number of features grows **exponentially** with the number of dimensions of the state;
- Note that the approximation is still **linear** in the weights.

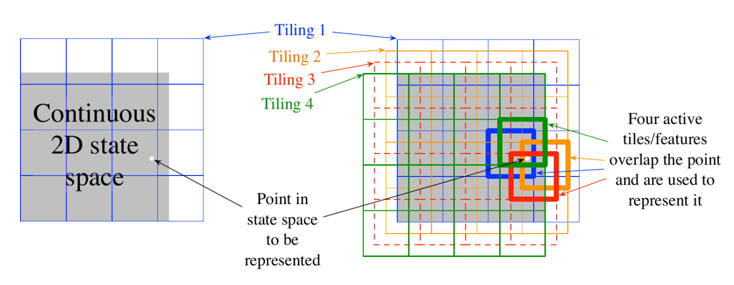
# Coarse coding

- Divide the state space in $M$ different **regions**;
- The feature vector has $M$ **binary** values;
- Given a state $s$, for each region, assign feature value 1 if the state is **inside** the region, 0 otherwise;
- 0-1 features are also called **sparse**.



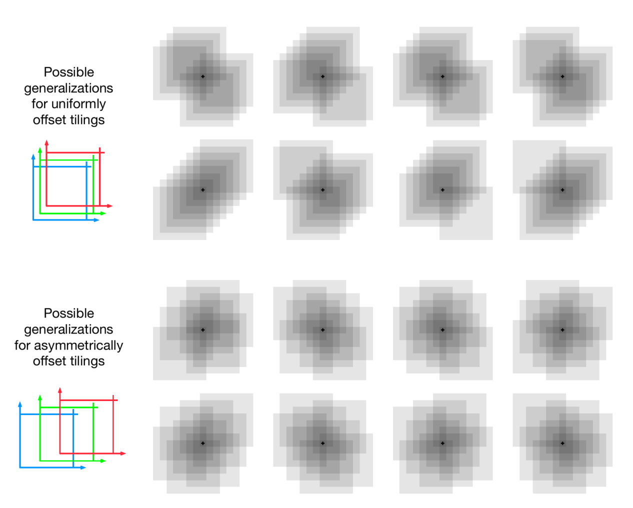
# Tile coding - 1

- **Flexible** and computationally efficient form of coarse coding;
- Use $N$ **tilings**, each one composed of $M$ **tiles**;
- The feature vector is a $N \times M$ **matrix**;
- The feature value is 1 if the state is **inside** a tile, 0 otherwise;




- Every state has the **same** number of active features.

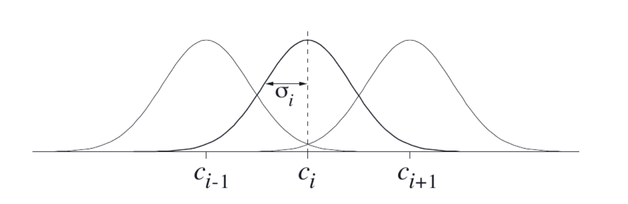
# Tile coding - 2



# Radial basis functions

- **Generalization** of coarse coding;
- Feature values are **real** numbers in $[0, 1]$;
- The **Gaussian** distribution is a typical RBF with mean $c_i$ and standard deviation $\sigma_i$

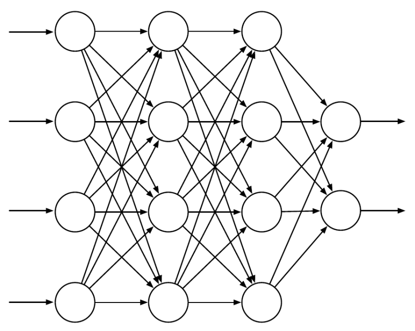
$$\varphi_i(s) = \exp\left(-\frac{\|s - c_i\|^2}{2\sigma_i^2}\right); \qquad (5)$$



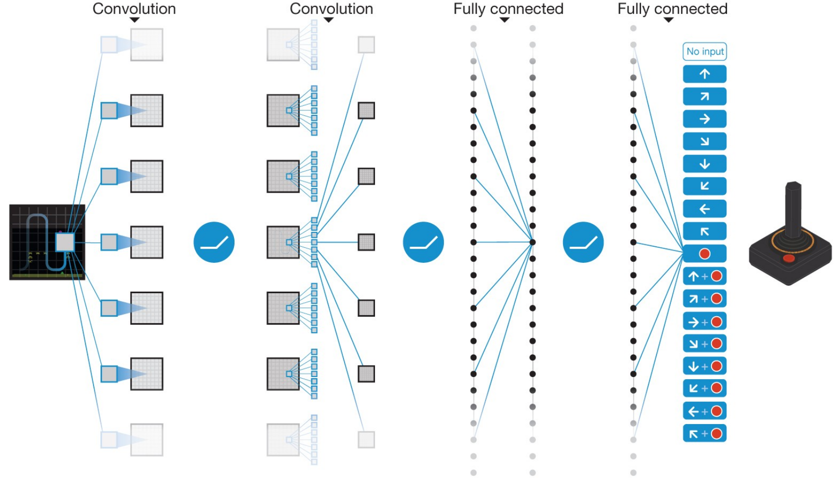

.

# Neural networks

- What if constructing features **by hand** is difficult or impractical?
- Use **neural networks**!
  - **Automatically** extract features in hidden layers;
  - Enable processing **high-dimensional** data.



# Deep neural networks



# Outline

# Semi-gradient off-policy TD(0)

- **Recall:** perform importance sampling to change the expectation from a **target** policy $\pi$ to the **behavioral** distribution $b$;
- Define the *importance sampling ratio*

$$\rho_t = \frac{\pi(a_t|s_t)}{b(a_t|s_t)}; \tag{6}$$

- The one-step semi-gradient **off-policy** TD(0) update is

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\rho_t\delta_t\nabla\hat{V}_{\mathbf{w}_t}(s_t); \tag{7}$$

where, e.g., $\delta_t = r_{t+1} + \gamma\hat{V}_{\mathbf{w}_t}(s'_t) - \hat{V}_{\mathbf{w}_t}(s_t)$.

# Off-policy divergence

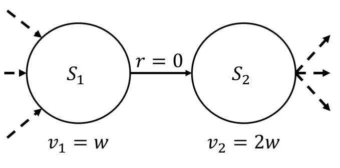
- Updating value functions following the **on-policy** distribution is important for convergence;
- The distribution of updates does **not** match the on-policy distribution;
- Off-policy algorithms with approximation can **diverge**!

# Example of off-policy divergence - 1

- Consider this transition as part of a **bigger** MDP;
- Consider $\boldsymbol{w}_0 = [w_0] = [10]$;
- Suppose $\gamma \approx 1$ and $\alpha = 0.1$;
- **Step 1:**
  - Take the transition, update $w_t$:

$$\begin{aligned}\delta_0 &= r_1 + \gamma \hat{V}_{\boldsymbol{w}_0}(s_2) - \hat{V}_{\boldsymbol{w}_0}(s_1)\\ &= 0 + \gamma 2 w_0 - w_0\\ &= (2\gamma - 1) w_0 \approx 10\end{aligned}$$

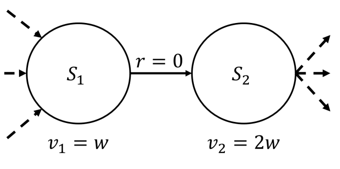
$$\begin{aligned}w_1 &= w_0 + \alpha \rho_0 \delta_0 \nabla \hat{V}_{\boldsymbol{w}_0}(s_1)\\ &= w_0 + \alpha \cdot 1 \cdot (2\gamma - 1) w_0 \cdot 1\\ &= (1 + \alpha(2\gamma - 1))\, w_0 \approx 11\end{aligned}$$



# Example of off-policy divergence - 2

- **Step 2:**
  - Take the transition, update $w_t$:

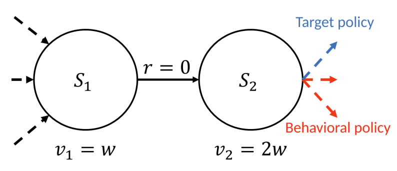
$$\begin{aligned}\delta_1 &= r_2 + \gamma \hat{V}_{\mathbf{w}_1}(s_2) - \hat{V}_{\mathbf{w}_1}(s_1)\\ &= 0 + \gamma 2 w_1 - w_1\\ &= (2\gamma - 1) w_1 \approx 11\\ w_2 &= w_1 + \alpha \rho_1 \delta_1 \nabla \hat{V}_{\mathbf{w}_1}(s_1)\\ &= w_1 + \alpha \cdot 1 \cdot (2\gamma - 1) w_1 \cdot 1\\ &= (1 + \alpha(2\gamma - 1))\, w_1 \approx 12.1\end{aligned}$$




  - The value of the updated parameter **diverges** if $1 + \alpha(2\gamma - 1) > 1$.

# Example of off-policy divergence - 3

- Divergence happens because the distribution of update is **different** from the on-policy distribution;
- *w* is updated **only** during the given transition, not after;



# The deadly triad

- **Instability** and **divergence** arise for methods based on the following elements:
  - **function approximation**;
  - **bootstrapping**;
  - **off-policy training**.
- Can we get rid of one of them without disadvantages?
  - function approximation is necessary to **scale** to large problems;
  - bootstrapping is important for data **efficiency**;
  - off-policy training is essential for learning from **heterogeneous** experience.

# Batch reinforcement learning

- Up to now, we have seen mostly **online** RL algorithm;
- **Batch** (a.k.a., **Offline**) RL methods use a previously collected dataset of transitions:

$$\mathcal{D} = \langle s_i, a_i, r_i, s_i' \rangle_{i=1}^{T}. \tag{8}$$

# Fitted $Q$-Iteration

- Given a dataset $\mathcal{D} = \langle s_i, a_i, r_i, s_i' \rangle_{i=1}^{T}$, solve a **sequence of regression** problems;
- **Stability** guarantees hold for particular regression methods:
  - regression trees;
  - kernel averaging.

**Algorithm** FQI

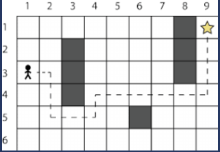
**Input:** dataset of transitions $\mathcal{D} = \langle s_i, a_i, r_i, s_i' \rangle_{i=1}^{T}$;
**Input:** $N = 0$, an initial $\hat{Q}_N(s, a) = 0$;
**while** true **do**
    $N \leftarrow N + 1$;
    Build training set $\mathcal{T} = \{\langle s_i, a_i, r_i + \gamma \max_{a \in \mathcal{A}} \hat{Q}_{N-1}(s_i', a) \rangle\}_{i=1}^{T}$;
    Use regression algorithm to build $\hat{Q}_N(s, a)$;
**end while**

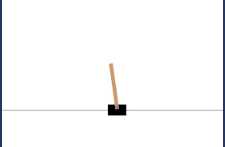
# Outline

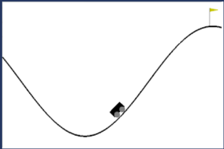
1. Continuous MDPs
2. Approximated on-policy methods
3. Feature construction for linear methods
4. Approximated off-policy methods
5. Value-based deep RL - DQN algorithm

# High-dimensional RL problems - 1

- Dimension of state and action space is **critical** in RL;
- **Curse of dimensionality:** theoretical and practical issues arising from **high**-dimensional problems;
- The RL methods we discussed can only handle **low**-dimensional problems.





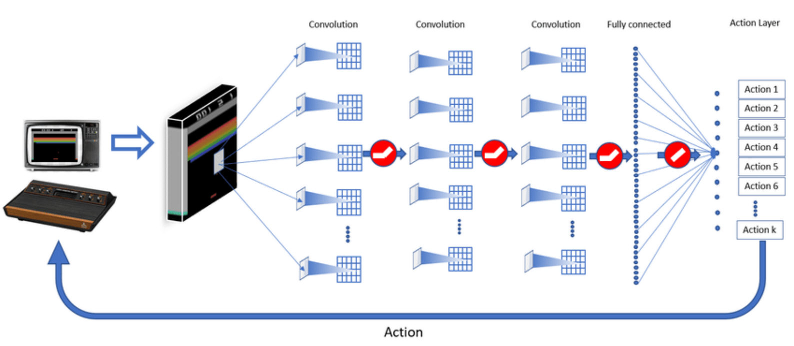
How to enable RL to solve **more** complex problems?

# High-dimensional RL problems - 2

- **High**-dimensional problems have a state space $\mathcal{S}$ with:
  - **Impractically** large number of discrete values (e.g., each pixel of an image has an integer value $\in [0, 255]$);
  - **More** than $8 - 10$ dimensions (e.g., position and velocity of joints for a robot).
- The action space $\mathcal{A}$ could be either **discrete** or **continuous**:
  - commands to a videogame;
  - torque to joints of a robot.

# Deep neural networks for RL

- High-dimensional state/action spaces can be handled with **deep neural networks** and the use of **deep learning** techniques.



Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

# Deep $Q$-Learning - 1

- Recall the definition of action-value function:

$$Q_\pi(s,a) = \mathbb{E}_\pi \left[ \sum_t \gamma^t r_{t+1} \middle| s_0 = s, a_0 = a \right]; \qquad (9)$$

- Suppose the state space $\mathcal{S}$ is **high**-dimensional;
- **Goal:** approximate the action-value function using a **deep** neural network with parameters $\theta$:

$$\hat{Q}(s,a;\theta) \approx Q_\pi(s,a). \qquad (10)$$

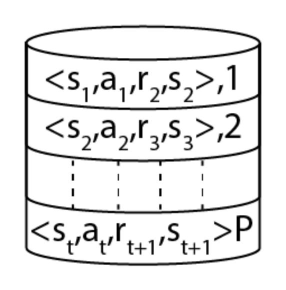
# Deep $Q$-Learning - 2

- **Minimize** the loss

$$\mathcal{L}_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(r + \gamma \max_{a'} \hat{Q}(s', a'; \theta_i) - \hat{Q}(s, a; \theta_i)\right)^2\right]; \tag{11}$$

- **Problems:**
  - this loss contains bootstrapping, off-policy, and of course function approximation $\rightarrow$ **deadly triad**!
  - an **offline** dataset of transitions is assumed **unavailable** due to the complexity of the problems $\rightarrow$ we cannot use offline algorithms (e.g., FQI);
  - data have to be collected online $\rightarrow$ training neural networks in online RL can lead to **catastrophic forgetting**.

# Deep $Q$-Learning - Replay buffer




- Collect and **store** past transitions to **reuse** them for update;
- Store transitions in queue, a.k.a. **replay buffer**, of finite capacity;
- Off-policy updates allow to reuse transitions out of the sampling distribution;
- Reduces the negative impact of **distribution shift**.

# Deep $Q$-Learning - Target network

- Keep a **copy** of the neural network updating it periodically;
- Every $C$ steps, the weights $\theta$ of the online network are **copied** in the target network as $\theta'$ and kept fixed for the next $C$ steps;
- The copy of the neural network, the **target network**, is used to compute the **target** of the mean squared TD-error

$$\mathcal{L}_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(r + \gamma \max_{a'} \hat{Q}(s', a'; \theta') - \hat{Q}(s, a; \theta_i)\right)^2\right]; \tag{12}$$

- Avoids **instability** due to function approximation.

# Deep $Q$-Learning - Minibatch updates

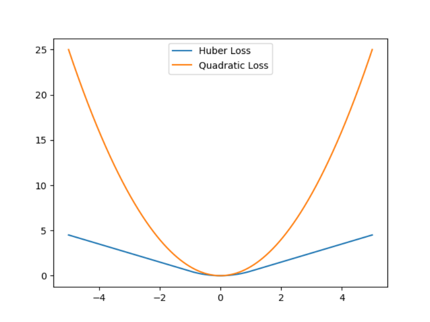
- At each step, uniformly **sample** a minibatch of $N$ transitions from the replay buffer

$$\mathcal{L}_i(\theta_i) = \sum_{i=1}^{N} \left[ \left( r_i + \gamma \max_{a'} \hat{Q}(s'_i, a'; \theta') - \hat{Q}(s_i, a_i; \theta_i) \right)^2 \right]; \qquad (13)$$

- Improves **efficiency** w.r.t. training on all transitions.

# Deep $Q$-Learning - Reward and target clipping

- **Clip reward** between $-1$ and $1$;
- **Clip the error** term of the update $r + \gamma \max_{a'} \hat{Q}(s', a'; \theta') - \hat{Q}(s, a; \theta)$ between $-1$ and $1$;
- It is sufficient to use **Huber loss**!




- Improve **stability** of the optimization.

# Deep $Q$-Learning - Pseudocode

**Algorithm** Deep Q-Learning

Initialize replay buffer $\mathcal{D}$ to capacity $N$;
Initialize action-value function $\hat{Q}$ with random weights $\theta$;
Initialize target action-value function weights $\theta' = \theta$;
**while** true **do**
  Initialize state $s_0$;
  **for** $t = 1, \ldots, T$ **do**
    Sample action $a_t$ with $\varepsilon$-greedy using $\hat{Q}$;
    Execute $a_t$ and observe reward $r_t$ and $s'_t$;
    Store transition $\langle s, a, r, s' \rangle$ in $\mathcal{D}$;
    Uniformly sample minibatch of $M$ transitions $\langle s_i, a_i, r_i, s'_i \rangle_{i=1}^{M}$ from $\mathcal{D}$;
    $$y_i = \begin{cases} r_i & \text{if episode terminates at step } i+1 \\ r_i + \gamma \max_{a'} \hat{Q}(s'_i, a'; \theta') & \text{otherwise} \end{cases}$$
    Perform gradient descent step on $\left(y_i - \hat{Q}(s_i, a_i; \theta)\right)^2$ w.r.t. weights $\theta$;
    Every $C$ steps do $\theta' = \theta$;
  **end for**
**end while**

# Deep $Q$-Learning - Applications

- DQN is a powerful algorithm
  - https://www.youtube.com/watch?v=W2CAghUiofY
  - https://www.youtube.com/watch?v=XMo0899Nz7o
  - https://www.youtube.com/watch?v=V1eYniJ0Rnk
- But this comes at a **cost**!
  - Requires many **samples**;
  - Highly sensitive to hyperparameter **tuning**;
  - High **computation time**.

# Wrap-up

- How to handle continuous Markov Decision Processes;
- How to build features for linear value function approximation;
- How to handle high-dimensional Markov Decision Processes;
- Deep $Q$-Network algorithm.