TDT4171 Artificial Intelligence Methods

Lecture 10 – Reinforcement Learning

Norwegian University of Science and Technology

Helge Langseth
Gamle Fysikk 255
helge.langseth@ntnu.no

NTNU

Outline

1. Reinforcement Learning
   - Relation to MDPs
   - *Q*-learning
   - Active RL: Being explorative
2. Deep RL
   - Motivation
   - DQN
   - Policy-based models
3. Summary

Notes to self

- Fyr opp flappy: `./demo/train_me.command` i Terminal
- Testing etterpå: `./demo/run_me.command`.

Learning goals for this reinforcement learning-part

Being familiar with:

- **Motivation** for reinforcement learning
- **Relation to MDP** – and what makes RL difficult
- **Learning part**, at least Q-learning for tabular problems
- **DRL** – motivation for general function approximators. High-level understanding of DQN and policy-based models.

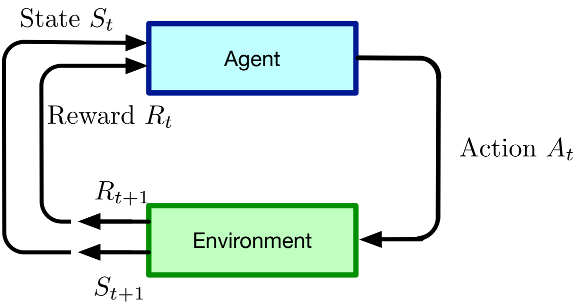
Note! Topics on these slides are relevant for exam – even if not covered by book curriculum.

Recall: Types of learning

1. **Unsupervised Learning**
   - No environmental feedback concerning correctness
   - Learning system detects patterns in the data without attaching right/wrong status to them.
2. **Supervised Learning**
   - Frequent environmental (e.g. teacher) feedback that includes the correct action/response.
   - Many classic learning algorithms rely on this constant feedback.
3. **Reinforcement Learning (RL)**
   - Occasional environmental feedback of form right/wrong or good/bad.
   - Feedback often comes at the end of a long sequence of actions.

RL + Unsupervised Learning are most common in the real world.
Good evidence for both in the brain.

RL: Top-level view



Loop:

1. The agent receives a percept: Current state and reward
2. The agent updates its understanding of the environment.
3. The agent decides on what to do next
4. The selected action is executed in the environment.
5. The environment produces a new percept (state and reward)

Edge of tomorrow (2014)

Lee Sedol: The face of RL's success

Lee Sedol vs. AlphaGo (2016)

Reinforcement learning

- Learning to act in **unknown environments** with only **occasional** feedback.
  - Feedback often comes at the end of a long sequence of actions.
- Agent learns from **its own experience** in the environment.
- RL involves learning the **whole problem at once**, not via combined sub-problems.
- Balance of **exploration -vs- exploitation** is key to getting complete information about the environment.
- RL often argued to be **"general purpose and easy to build"**: Lots easier to say "Winning is 1 point, loosing is 0" than to provide vast amounts of training examples.
- **Assumptions:**
  - The environment is a Markov Decision Process
  - Markov assumption $\rightarrow S_t$ must hold all (relevant) info

Flavors of RL research

Dimension 1: Models for transitions?

Model-based RL: Learn (or get) an explicit transition-model, $P(s'|a, s)$. Often learn a reward-model, too.

Model-free RL: No (explicit) transition-model:

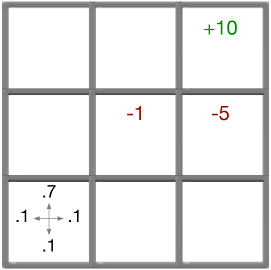
- Utility-based: Learn the utility / reward-to go.
- Policy-based: Learn the policy $\pi(s)$.

Dimension 2: Fixed policy?

Passive RL: Assumes $\pi(s)$ fixed, and estimates utility for that π.

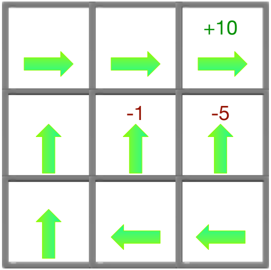
Active RL: Learns policy by exploration etc., utilizing Passiv-RL techniques to estimate utility.

Recap: Value Iteration in the "maze"



1. Agent gets **rewards** when entering some states. **Rewards known in advance**
2. **Infinite** time horizon; future rewards **discounted** (factor γ)
3. Action-outcomes random, but **known** $P(S_{t+1} = s' | S_t = s, a)$.
4. A **solution** defines action to choose in each state s to maximize expected discounted cumulative reward.

Recap: Value Iteration in the "maze"

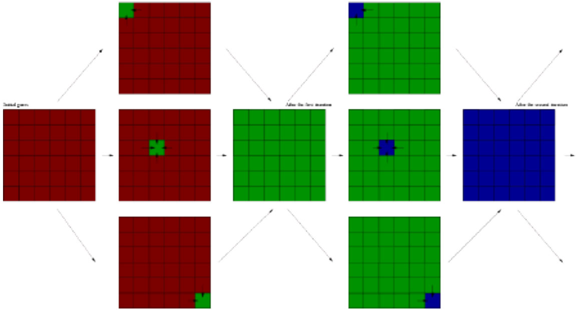


Solution:

1. Allocate **utilities** $U^*(s)$ to each state; $U^*(s)$ is the optimal expected discounted future reward when starting in s.
2. For utilities we have
   $U^*(s) = \max_a \left[R(a,s) + \gamma \cdot \sum_{s'} P(s' \mid a,s)\, U^*(s')\right]$
3. **Action selection:** We follow MEU. In this example: "Move to the neighbouring state with highest utility".

Recap: Value iteration algorithm

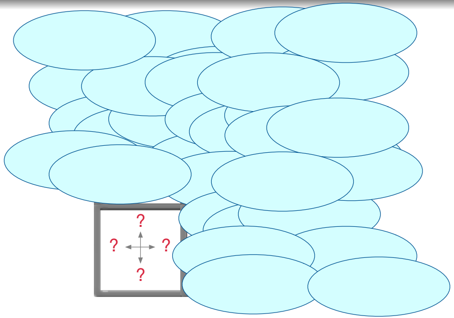
Start with **initial guess** of $U^*(s)$, and **iteratively refine** it:



Things to note:

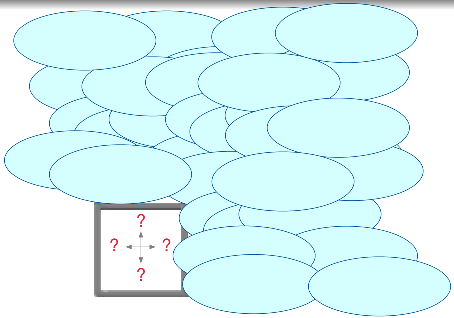
1. **Update rule:**
   $U^{j+1}(s) := \max_a \left[R(a,s) + \gamma \cdot \sum_{s'} P(s' \mid a,s)\, U^j(s')\right]$.
2. $U^j(s)$ converges to the "true" utilities $U^*(s)$
3. Everything can be calculated **directly from the model**.

Reinforcement learning in the "maze"



1. The agent gets **rewards** when entering some states. **Rewards UNKNOWN**
2. **Infinite** time horizon; future rewards **discounted** (factor γ)
3. Model for outcome of actions is **UNKNOWN**, but the agent may spend (unbounded) time on **learning** (trial & error)
4. A **solution** defines action to choose in each state s to maximize expected discounted cumulative reward.

Reinforcement learning in the "maze"



Requirements:

We need something similar to value iteration, but more clever:

1. The agent must **explore** the domain ("maze") on its own
2. The effect of actions in each state (both state-change and the rewards) must be **learned**

Can we use same techniques as before?

General idea for Value Iteration:

1. Define utility $U^*(s)$ as accumulated discounted reward starting from s and following optimal policy thereafter.
2. Calculate utility U_j iteratively so that $U_j(s) \xrightarrow{j\to\infty} U^*(s)$.
3. Define policy so that π_j is the **MEU** choice wrt. U_j.

Can we use the same setup when we do not know:

- The transfer distribution $P(s'|s,a)$
- The reward $R(a,s)$

Discuss with your neighbour for a couple of minutes.

Can we use same techniques as before?

General idea for Value Iteration:

1. Define utility $U^*(s)$ as accumulated discounted reward starting from s and following optimal policy thereafter.
2. Calculate utility U_j iteratively so that $U_j(s) \xrightarrow{j\to\infty} U^*(s)$.
3. Define policy so that π_j is the **MEU** choice wrt. U_j.

Problems:

- **Transfer distribution:** Cannot use MEU without $P(s'|s,a)$.
- **Rewards:** Cannot *calculate* $U_j(s)$-values without $R(a,s)$.

Can we use same techniques as before?

General idea for Value Iteration:

1. Define utility $U^*(s)$ as accumulated discounted reward starting from s and following optimal policy thereafter.
2. Calculate utility U_j iteratively so that $U_j(s) \xrightarrow{j\to\infty} U^*(s)$.
3. Define policy so that π_j is the **MEU** choice wrt. U_j.

Problems:

- **Transfer distribution:** Cannot use MEU without $P(s'|s,a)$.
- **Rewards:** Cannot *calculate* $U_j(s)$-values without $R(a,s)$.

Solution:

- Define $Q(a,s)$: accumulated discounted reward starting by **doing action** a in s and following optimal policy thereafter.
- Explore the domain to *estimate* $Q(a,s)$ (and $U^*(s)$, too).

Q-learning – deterministic world

We will solve the problem using "Q-learning":

- $Q(a, s)$ is expect discounted cumulative rewards if we start by doing a in state s, and follow **optimal** policy thereafter.
- **Assume for now that** when doing a in a state s the agent always moves to the same state denoted $\delta(a, s)$.

So, $U^*(s) = \max_{a'} Q(a', s)$, and we have

$$\begin{aligned} Q(a, s) &= R(a, s) + \gamma \cdot U^*(\delta(a, s)) \\ &= R(a, s) + \gamma \cdot \max_{a'} Q(a', \delta(a, s)) \end{aligned}$$

Q-learning – deterministic world

We will solve the problem using "*Q*-learning":

- $Q(a, s)$ is expect discounted cumulative rewards if we start by doing a in state s, and follow **optimal** policy thereafter.
- **Assume for now that** when doing a in a state s the agent always moves to the same state denoted $\delta(a, s)$.

So, $U^*(s) = \max_{a'} Q(a', s)$, and we have

$$\begin{aligned} Q(a, s) &= R(a, s) + \gamma \cdot U^*(\delta(a, s)) \\ &= R(a, s) + \gamma \cdot \max_{a'} Q(a', \delta(a, s)) \end{aligned}$$

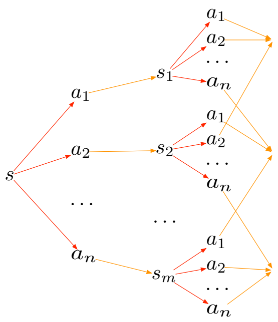
The updating function / Bellman equation

$$\hat{Q}(a, s) \leftarrow R(a, s) + \gamma \cdot \max_{a'} \hat{Q}(a', \delta(a, s))$$

We are now sure $\hat{Q}(a, s)$ converges to the "true" utilities

Deterministic Q-learning – In pictures

$$Q(A_t, S_t) = R(A_t, S_t) \quad + \gamma \cdot R(A_{t+1}, S_{t+1}) \; + \quad \ldots$$

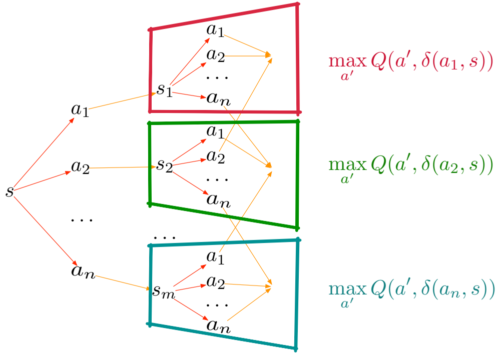


$S_t : s \quad A_t : a \quad S_{t+1} : \delta(a, s)$

- States given deterministically, actions chosen according to π.
- For given π, expanding a path until termination gives $Q(a_t, s_t)$.

Deterministic Q-learning – In pictures

$$Q(A_t, S_t) = R(A_t, S_t) \qquad + \gamma \cdot R(A_{t+1}, S_{t+1}) \;+ \qquad \ldots$$

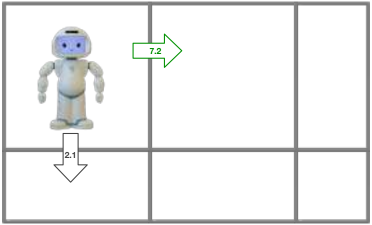


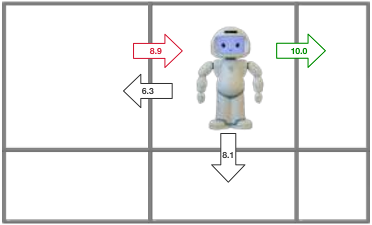
$S_t : s \quad A_t : a \quad S_{t+1} : \delta(a, s)$

- Alternative: Utility from $\delta(a, s)$ downwards is $Q(a', \delta(a, s))$.
- Rational π is MEU, so $\max_{a'} Q(a', \delta(a, s))$ tells us all.

Updating $\hat{Q}$ — An example

An agent located in state s_1 (Left Fig.) is doing Q-learning. It performs a_{right} and ends up in s_2 (Right Fig.).





Use this info to update $\hat{Q}\left(a_{\text{right}}, s_1\right)$:

$$
\begin{aligned}
\hat{Q}\left(a_{\text{right}}, s_1\right) &:= R(a_{\text{right}}, s_1) + \gamma \cdot \max_{a'} \hat{Q}(a', s_2) \\
&= -0.1 + 0.9 \; \max\{6.3, 8.1, 10.0\} = 8.9
\end{aligned}
$$

Demo: RL-sim

RL-Sim, Q-leaning with maze `8_big.maze`.
Parameters: PJOG=epsilon=0.0. **Animate=Off**
Compare update-path for one episode with one update in Value Iteration, same maze, same parameters.

Nondeterministic Case

Q learning generalizes to **nondeterministic worlds**:

$$\hat{Q}_n(a,s) \leftarrow (1-\alpha_n)\hat{Q}_{n-1}(a,s) + \alpha_n[R(a,s) + \gamma \cdot \max_{a'} \hat{Q}_{n-1}(a',s')]$$

where $\alpha_n = \frac{1}{1+\text{visits}_n(a,s)}$ and we observed the move $(a,s) \rightarrow s'$.

Do you think this is a meaningful way of doing it?
... and if so, what is the intuition behind this specific update?

Discuss with your neighbour for a couple of minutes

Nondeterministic Case

Q learning generalizes to **nondeterministic worlds**:

$$\hat{Q}_n(a,s) \leftarrow (1-\alpha_n)\hat{Q}_{n-1}(a,s) + \alpha_n[R(a,s) + \gamma \cdot \max_{a'} \hat{Q}_{n-1}(a',s')]$$

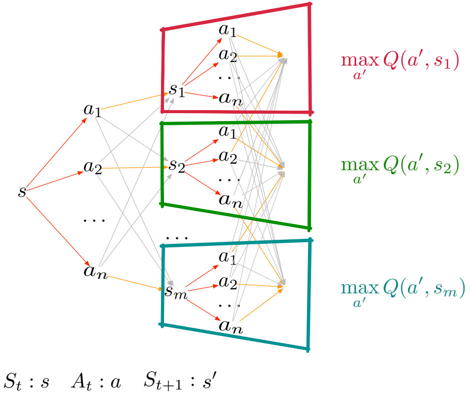
where $\alpha_n = \frac{1}{1+\text{visits}_n(a,s)}$ and we observed the move $(a,s) \to s'$.

Intuition:

- Setting $\hat{Q}_n(a,s) = R(a,s) + \gamma \cdot \max_{a'} \hat{Q}_{n-1}(a',s')$ is bad. Depends on the draw of $s' \sim P(s'|a,s)$.
- $\hat{Q}_{n-1}(a,s)$ is the accumulated knowledge of what action a in state s leads to. It is based on $\text{visits}_n(a,s)$ observations.
- Do it once more, and sample $R(a,s) + \gamma \cdot \max_{a'} \hat{Q}_{n-1}(a',s')$.
- $\hat{Q}_n(a,s)$: A weighted average of old and new information.
- Weights proportional to the number of trials: Old info is $\text{visits}_n(a,s)$ as important as the new (single observation) info.
- After normalization, new info is weighted α_n, old with $1-\alpha_n$.

Non-deterministic Q-learning – In a picture

$$Q(A_t, S_t) = R(A_t, S_t) \quad + \gamma \cdot R(A_{t+1}, S_{t+1}) \; + \quad \ldots$$



- Step through environment with $s' \sim P(S_{t+1} | S_t = s, A_t = a)$.
- Gives a sample for $Q(a, s)$, in general not "correct" value.

TD Learning

We already found a way to generalize Q learning to nondeterministic worlds:

$$\hat{Q}(a,s) \leftarrow (1-\alpha)\hat{Q}(a,s) + \alpha\left[R(a,s) + \gamma \cdot \max_{a'} \hat{Q}(a',s')\right].$$

Re-order terms to get the TD (temporal difference) formulation:

$$\hat{Q}(a,s) \leftarrow \hat{Q}(a,s) + \alpha\left[R(a,s) + \gamma \max_{a'} \hat{Q}(a',s') - \hat{Q}(a,s)\right]$$

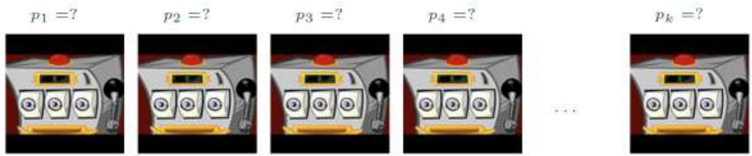
- We interpret $R(a,s) + \gamma \max_{a'} \hat{Q}(a',s') - \hat{Q}(a,s)$ as "error". Should be zero (in expectation) if $\hat{Q}$ follows Bellman!
- We interpret α as a "learning rate". Sometimes kept fixed.
- We denote $R(a,s) + \gamma \max_{a'} \hat{Q}(a',s')$ the *TD(0)-target*. We can do more, e.g., $R(a,s) + \gamma R(a',s') + \gamma^2 \max_{a''} \hat{Q}(a'',s'')$.

Active RL: Greedy vs. Explorative

Simplified domain to understand Active RL:

The k-bandit problem (no time-structure)

Confronted with k slot-machines we need to decide on a policy: How to earn the most?



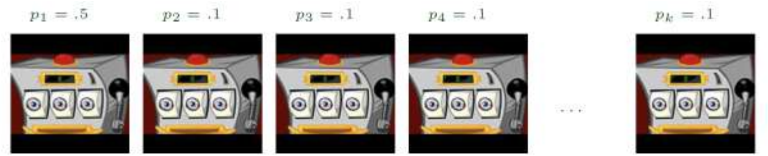
- Each has an unknown probability of giving \$10 payout.

Active RL: Greedy vs. Explorative

Simplified domain to understand Active RL:

The k-bandit problem (no time-structure)

Confronted with k slot-machines we need to decide on a policy: How to earn the most?

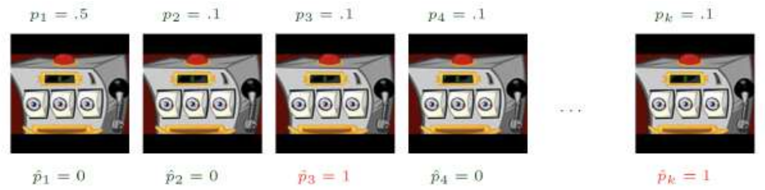


- Each has an unknown probability of giving \$10 payout.
- Natural to use 1 coin on each machine (?)

Active RL: Greedy vs. Explorative

Simplified domain to understand Active RL:
The k-bandit problem (no time-structure)

Confronted with k slot-machines we need to decide on a policy: How to earn the most?



- Each has an unknown probability of giving \$10 payout.
- Natural to use 1 coin on each machine (?)
- We will have **rough estimates** of how good each machine is (here: probability of winning \$10). **How to keep learning?**

Action selection – Exploitation vs. Exploration

In a state s we should base our action selection on $\hat{Q}$:

Greedy: Choose $\arg\max_a \hat{Q}(a,s)$ (**No!!**)

Random: Choose an action on random, all equally likely (**No!!**)

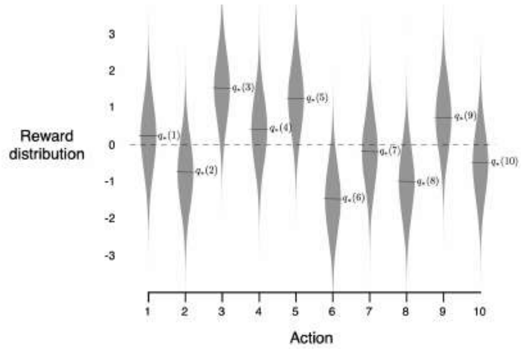
ϵ-greedy: With probability ϵ choose a random action, with $1-\epsilon$ be greedy. (**OK**, but finding a good ϵ not easy)

Guided: Each action a is chosen with probability proportional to $k^{\hat{Q}(a,s)}$ where $k \geq 1$ typically grows as agent learns more. (**Sure**, but finding k can be tricky)

UCB: Choose $\arg\max_a \; \hat{Q}(a,s) \; + \; c\sqrt{\frac{\log(N_s)}{N_{a,s}}}$. Typical value for c is $c \sim 2$. (**Yes**)

(…and there are other techniques as well)

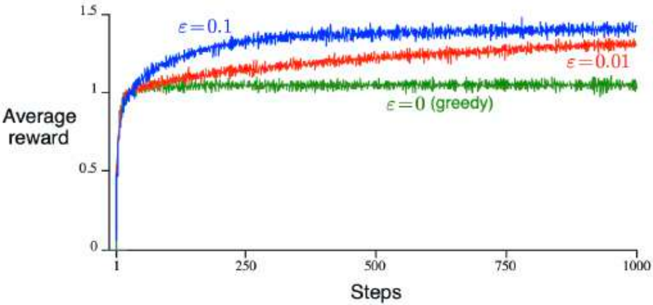
Action selection – Example w/ continuous rewards



- There are $k = 10$ bandits to choose from.
- Each has a **separate random reward scheme** (mean values differ, standard deviation is 1 for all bandits).
- Best choice is to go for Bandit 3 all the time, but the agent doesn't know that $\Rightarrow$ **Exploration!**

Action selection – Example w/ continuous rewards

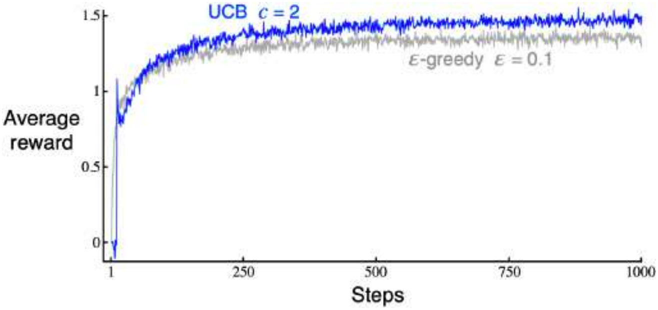
ϵ-greedy:



- Probability ϵ: Greedy; Probability $1 - \epsilon$: Random.
- Any $\epsilon > 0$ will solve the problem (eventually); $\epsilon = 0$ fails – stays with winner from first round.
- Too small ϵ makes exploration slow; too high and you keep wasting money on unguided (suboptimal) exploration.

Action selection – Example w/ continuous rewards

UCB:



- Upper Confidence Bound: $\arg\max_a \; \hat{Q}(a,s) \; + \; c\sqrt{\frac{\log(N_s)}{N_{a,s}}}$.
- UCB better than best ϵ-greedy here – *not* uncommon!
- UCB is typically not *that* sensitive to the c - value.

Full algorithm: Q-learning in discrete state-spaces

Implementation of $\pi(s)$ using tabular Q-learning:

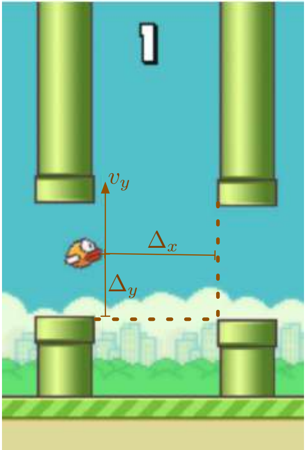
function Q-LEARNING-AGENT(*percept*) **returns** an action
  inputs: *percept*, a percept indicating the current state s' and reward signal r
  persistent: Q, a table of action values indexed by state and action, initially zero
    N_{sa}, a table of frequencies for state–action pairs, initially zero
    s, a, the previous state and action, initially null

  if s is not null **then**
    increment $N_{sa}[s, a]$
    $Q[s,a] \leftarrow Q[s,a] + \alpha(N_{sa}[s,a])(r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$
  $s, a \leftarrow s', \text{argmax}_{a'} f(Q[s',a'], N_{sa}[s',a'])$
  return a

Note:

- Update of Q using TD; α *can* depend on no. (s, a)-visits
- f implements action selection. Again *can* use no. (s, a)-visits; this is needed by UCB (and some others).

Example of a full system: Flappy Bird



State-description:

- Bird's distance to nearest pipe:
  - Along x-axis: Δ_x
  - Along y-axis: Δ_y
- Bird's velocity along y-axis: v_y

Other info:

- Legal actions known: Flap or not
- Rewards:
  - Incentivized to live;
  - Incentivized to pass pipes;
  - Punished if it dies;
  - Crash is terminal.

The plan: Q-learning!

- State is a tuple with **discretized** values of Δ_x, Δ_y, and v_y;
- Description incomplete $\rightarrow$ Consider domain non-deterministic.

Deep RL

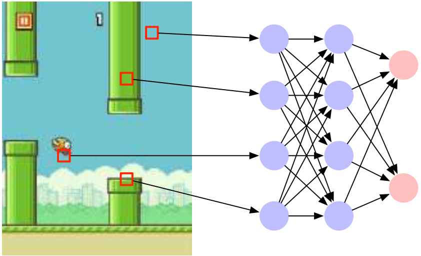
- **Flappy** needs a discretized state-description:
  - A coarse discretization ($\rightarrow$ imprecise state-representation) will **prevent** optimal behaviour.
  - Finer discretization of s means $Q(a, s)$ will be **harder to learn**.
  - Would be better to **not discretize the state**, and rather learn some function $f_a(\cdot)$ so that $f_a(s) = Q(a, s)\ \forall s$.

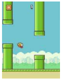
Deep RL

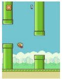
- **Flappy** needs a discretized state-description:
  - A coarse discretization ($\rightarrow$ imprecise state-representation) will **prevent** optimal behaviour.
  - Finer discretization of s means $Q(a, s)$ will be **harder to learn**.
  - Would be better to **not discretize the state**, and rather learn some function $f_a(\cdot)$ so that $f_a(s) = Q(a, s)\ \forall s$.
- **Idea:** Make a neural network to approximate $Q(a, s)$
  - **Network input:**
    - **In general:** State description rich enough to ensure the Markov assumption.
    - **For Flappy:** Raw measurements (or screen-grab/s of game)
  - **Network output:**
    - **In general:** One output per action, defined so that for input s, output-node j gives $\hat{Q}(a_j, s)$. A **single** NN covers all actions. Weight-sharing speeds up learning.
    - **For Flappy:** Two outputs, one for "flap" and one for "nothing"
  - This setup is known as the **Deep Q-Network** (DQN), works for general s, discrete action-space.

Deep RL

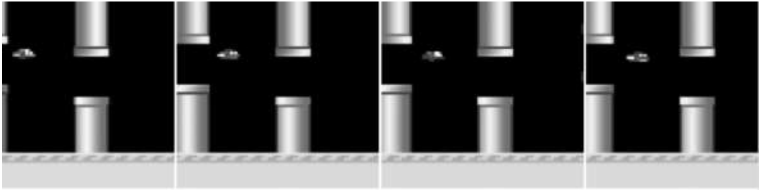
- **Flappy** needs a discretized state-description:
  - A coarse discretization ($\rightarrow$ imprecise state-representation) will **prevent** optimal behaviour.
  - Finer discretization of s means $Q(a, s)$ will be **harder to learn**.
  - Would be better to **not discretize the state**, and rather learn some function $f_a(\cdot)$ so that $f_a(s) = Q(a, s)\ \forall s$.



$\hat{Q}_{\boldsymbol{\theta}}\Big($  $, a = \text{Flap}\Big)$

$\hat{Q}_{\boldsymbol{\theta}}\Big($  $, a = \text{Nothing}\Big)$

Deep Flappy



- **Input:** 4 last screen-grabs from the game:
  - Each screen-grab is a (width $\times$ height) - matrix of grey-scale values (floats between 0 and 1)
  - We have four of them, giving a tensor of size $(80 \times 80 \times 4)$
- **Model:** Some conv.layers, some dense layers. ϵ-greedy.
- **Output:** Values for $\hat{Q}(a = \text{nothing}, s)$ and $\hat{Q}(a = \text{flap}, s)$.
- **Learning:** Tune weights so that we ensure

$$\left[R(a,s) + \gamma \cdot \max_{a'} \hat{Q}(a', s')\right] - \hat{Q}(a,s) \approx 0.$$

- **Results:** Runs "forever"! (Do `run_me.command` if time.)

Tables vs. functional approximators

Recall how tabular learning relates to TD-error:

$$\hat{Q}(a,s) \leftarrow \hat{Q}(a,s) + \alpha \left[R(a,s) + \gamma \max_{a'} \hat{Q}(a',s') - \hat{Q}(a,s) \right]$$

- Updates are **local** and isolated to the given s.
- No such thing as isolated updates in a neural network.
  - **Must** assume that "similar states" have similar Q-values.
  - What *"similar states"* means is **decided during learning**.
- Makes sense to learn a model that minimizes TD error

  $$R(a,s) + \gamma \max_{a'} \hat{Q}(a',s') - \hat{Q}(a,s),$$

  and "hope" the representation of s enforces these similarities.
- In practice, one uses L_2 (or Huber) loss on the TD error.

DQN top-level

Rough algorithm:

- For $t = 0, \ldots$:
  - Choose action: $a_t \leftarrow \max_a \hat{Q}_{\boldsymbol{\theta}}(s_t, a)$ (or random; ϵ-greedy)
  - Execute in environment: $\langle s_{t+1}, r_t \rangle \leftarrow \text{Execute}(a_t, s_t)$.
  - The observation $\langle s_t, a_t, r_t, s_{t+1} \rangle$ can be used for training! Gradient-step for the model defined by

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \, \boldsymbol{\nabla}_{\boldsymbol{\theta}} \Big(y_t - \hat{Q}_{\boldsymbol{\theta}}(a_t, s_t) \Big)^2 ,$$

where $y_t = r_t + \gamma \max_a \hat{Q}_{\boldsymbol{\theta}}(a, s_{t+1})$.

Note!

- $Q(a, s)$ means a specific output on the network. Only gradient from selected a_t used during learning.
- DQN (and DRL in general) is often explained using $V(s) = \max_a Q(a, s)$; I follow the original DQN paper here.
- **Tweaks:** Learning is always done using a *replay buffer* and typically with a stabilizing *second model* to get y_t.

Policy-based DRL

Setup

- DQN changes its weights to **minimize TD-error**. Selecting $\pi(s|\boldsymbol{\theta}) \leftarrow \max_a \hat{Q}_{\boldsymbol{\theta}}(a, s)$ produces (close to) optimal agents.
- Policy-based methods rather learn $\pi_{\boldsymbol{\theta}}$ directly – weights chosen to **optimize performance**.

Positives

- Can handle continuous and/or high-dim action spaces;
- Computationally more efficient (sometimes);
- Useful for stochastic policies.

Negatives

- We need an objective to replace the TD-error;
- We need the gradient of that objective function.

Policy Gradient Theorem

- $\mathcal{J}(\boldsymbol{\theta}) := \mathbb{E}_{s_0, a_t \sim \pi_{\boldsymbol{\theta}}(s_t)} V_{\pi_\theta}(s_0)$: Expected value following $\pi_{\boldsymbol{\theta}}$.
- We will **maximize** $\mathcal{J}(\boldsymbol{\theta})$ — corresponding loss is $-1 \cdot \mathcal{J}(\boldsymbol{\theta})$.

Policy Gradient Theorem

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a Q_\pi(a, s) \boldsymbol{\nabla}_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(s)[a]$$

where

- $\mu(s)$: Frequency of state-visits to s;
- $Q_\pi(a, s)$: Value of doing a at s then follow policy $\pi_{\boldsymbol{\theta}}$;
- $\pi_{\boldsymbol{\theta}}(s)[a]$: Policy's affinity for action a in state s.

Notice how nicely this works!

Gradients focus on state-action-pairs where states are frequently visited ($\mu(s)$), we have high value ($Q(a, s)$) and the effect on the policy is high ($\boldsymbol{\nabla}_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(s)[a]$).

Policy Gradient Theorem

- $\mathcal{J}(\boldsymbol{\theta}) := \mathbb{E}_{s_0, a_t \sim \pi_{\boldsymbol{\theta}}(s_t)} V_{\pi_\theta}(s_0)$: Expected value following $\pi_{\boldsymbol{\theta}}$.
- We will **maximize** $\mathcal{J}(\boldsymbol{\theta})$ — corresponding loss is $-1 \cdot \mathcal{J}(\boldsymbol{\theta})$.

Policy Gradient Theorem

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}}\, \mathcal{J}(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a Q_\pi(a, s) \boldsymbol{\nabla}_{\boldsymbol{\theta}}\, \pi_{\boldsymbol{\theta}}(s)[a]$$

where

- $\mu(s)$: Frequency of state-visits to s;
- $Q_\pi(a, s)$: Value of doing a at s then follow policy $\pi_{\boldsymbol{\theta}}$;
- $\pi_{\boldsymbol{\theta}}(s)[a]$: Policy's affinity for action a in state s.

Still some way to go. . .

There are a number of tricks to train DRL systems with policy gradients, but these are out of scope for us.

RL implementation

- `drl_minimal.py`: Interface to environments through `gymnasium`. Defining your own environment is immediate.
- Agents in `stable_baselines3`: `DQN` and `PPO` straight out of the box. Tweaking them is quite simple.

Summary

- **RL: Learning to operate in unknown environments**:
  - Setup fairly similar to the sequential decisions we looked at earlier, but changed focus to **Q-learning**: Find $Q(a, s)$ for action-state-pairs
  - Where Value Iteration learns an optimal policy using the MEU principle directly, an RL agent does not have this luxury, and must **explore** its domain.
  - Classic RL techniques assume discrete (preferably small!) state and action-spaces. When that does not hold: **Deep RL**.