Deep Learning

Lecture 10: Deep reinforcement learning

Guest lecture by Matthia Sabatelli
m.sabatelli@uliege.be

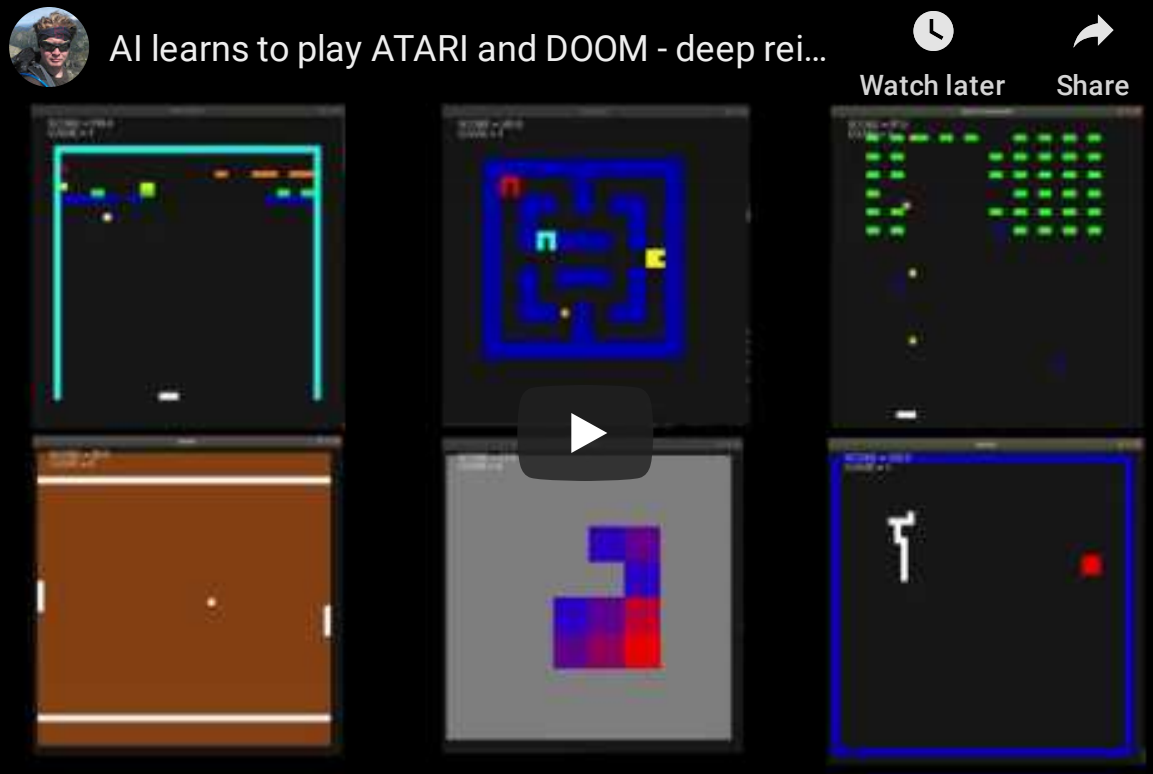
LIÈGE université

Today

Understand the field of Reinforcement Learning (RL) and see how it can be combined with neural networks.

- Markov Decision Processes
- Value functions and optimal policies
- Temporal Difference Learning
- Function approximators

Reinforcement Learning


AI learns to play ATARI and DOOM - deep rei...
Watch later
Share

Markov Decision Processes

Markov Decision Processes (MDP) are a classical formalization when it comes to sequential decision making problems.

MDPs allow us to mathematically define RL problems for which precise and sound statements can be made.

An MDP consists of the following elements:

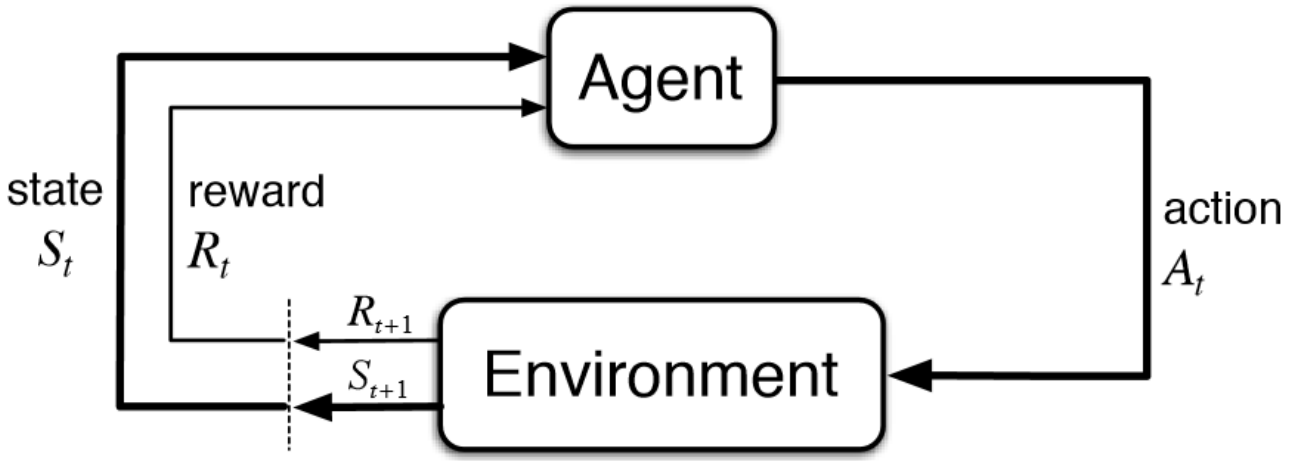
- a set of possible states $\mathcal{S}$,
- a set of possible actions $\mathcal{A}$,
- a reward signal $R(s_t, a_t, s_{t+1})$,
- a transition probability distribution $p(s_{t+1} | s_t, a_t)$.

When it comes to most RL settings, the MDPs can come in a more challenging form than the one they have when we use planning or dynamic programming algorithms.

Specifically we consider cases in which the following information is not known:

- the transition probability distribution $p(s_{t+1}|s_t, a_t)$
- the reward $R(s_t, a_t, s_{t+1})$

In practice this means that we do not know beforehand which states are good or bad, and therefore also do not know what actions to take.



The agent-environment interface

- The agent corresponds to the learner, sometimes also defined as the decision maker, which has the ability to continually interact with the environment.
- Each time an action is performed the environment has the ability to change and will present new situations to the agent.

Credits: Sutton and Barto, 2018.

Differently from supervised learning, in RL we have to deal with the component of time:

- At each discrete time-step $t = 0, 1, 2, 3, \ldots$ the agent receives a state representation s_t, selects an action a_t, and receives a numerical reward $r_t \in \mathbb{R}$ after which it will find itself in a new state s_{t+1}.
- This gives rise to trajectories

$$s_t, a_t, r_t, s_{t+1}, a_{t+1}, r_{t+1}, s_{t+2}, \ldots$$

- s_t and a_t at time-step t give all the necessary information that is required for predicting to which state the agent will step next.
- This is related to the fact that the environment is Markovian, where an action a_t only depends on the current state s_t

$$\begin{aligned} p(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, ...) &= p(s_{t+1}|s_t, a_t) \\ &= T(s_t, a_t, s_{t+1}). \end{aligned}$$

  For predicting the future it does not matter how an agent arrived in a particular current state.
- Similarly, the reward that is obtained is only determined by the previous action and not by the history of all previously taken actions,

$$p(r_t|s_t, a_t, ..., s_1, a_1) = p(r_t|s_t, a_t).$$

Goals and returns

So far we have properly defined how an agent can interact with an environment but have not seen **why** this should be done.

- The purpose of an RL agent is formalized by $r_t \in \mathbb{R}$, which is a numerical quantity that we want to maximize.
- We do not want to maximize the immediate reward but rather the cumulative reward

$$G_t = r_t + r_{t+1} + r_{t+2} + \ldots + r_T.$$

- Mathematically this can be seen as maximizing the expected value of the cumulative sum of a scalar signal.

To properly define the concept of return we need one additional component: the discount factor γ.

- The idea of discounting allows our agent to select actions which will maximize the sum of discounted rewards, therefore maximizing the discounted return:

$$\begin{aligned} G_t &= r_t + \gamma r_{t+1}, \gamma^2 r_{t+2} + \dots \\ &= \sum_{k-0}^{\infty} \gamma^k r_{t+k+1}. \end{aligned}$$

- The discount factor $0 \leq \gamma \leq 1$ and controls the trade-off between immediate and long-term rewards.

Policies and value functions

Basically all RL algorithms involve the concept of value function, functions that are able to estimate how good or bad it is for an agent to be in a particular state.

- The goodness of a state is defined in terms of future rewards that can be expected by being in state s.
- Just being in a good state is not enough, since the rewards will depend on which actions will be performed in the future.
- We need the concept of policy:

$$\pi : \mathcal{S} \rightarrow \mathcal{A}$$

RL methods specify how the agent's policy is changed as a result of its experience.

When it comes to value functions there are two popular value functions we care about

- The state-value function:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \middle| s_t = s, \pi\right]$$

- The state-action value function:

$$Q^{\pi}(s, a) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \middle| s_t = s, a_t = a, \pi\right].$$

Both value functions can compute the desirability of being in a specific state.

The goal of a RL agent is to find an optimal policy that realizes the optimal expected return

$$V^*(s) = \max_\pi \ V^\pi(s), \ \text{for all } s \in \mathcal{S}$$

and the optimal state-action value function

$$Q^*(s,a) = \max_\pi \ Q^\pi(s,a) \ \text{for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A}.$$

Both of these optimal value functions satisfy the Bellman optimality equation, and can be learned via Monte-Carlo or Temporal-Difference learning methods.

What do value functions represent in practice?

- Remember that the environment is unknown.
- We can see value functions as some sort of knowledge representation of an agent.
- Through its interactions with the environment, the agent accumulates knowledge which is stored by updating its value function.
- If a value function is accurate an agent will know everything he needs to know for interacting with an environment.

Monte Carlo (MC) methods

- The first method than can be used for learning a value function without assuming complete knowledge of the environment: model-free RL.
- The only requirement of MC methods is experience: sampling sequences of states, actions and rewards from an environment which can be simulated.

The idea is to compute the real return once an episode terminates

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

and keep track of the value of a single state based on the amount of times this state has been visited

$$V(s_t) = \frac{\sum_{i=1}^{k} G_t(s)}{N(s)}.$$

This estimate can then get updated as follows:

$$V(s_t) := V(s_t) + \alpha[G_t - V(S_t)].$$

Drawbacks

- We need to wait until an episode is terminated because only then G_t will be known.
- Convergence is slowed down.

Temporal Difference (TD)-Learning

- TD-Learning is a combination of Monte Carlo ideas and dynamic programming ideas. It is the most central and novel idea of entire Reinforcement Learning
- Just like MC methods TD-Learning approaches can simply learn from raw experiences without the need for a model of the environment. Like DP techniques, the update estimates are based (in part) on other learned estimates.
- This means we do not have to wait until the end of an episode anymore but instead rely on bootstrapping:

$$V(S_t) := V(S_t) + \alpha[r_t + \gamma V(S_{t+1}) - V(s_t)]$$

The core idea of TD-Learning is the concept of TD-error, or sometimes called target

$$\delta_t = r_t + \gamma V(s_{t+1}).$$

- It corresponds to the only information that is needed in order to update our value estimates (remember that we do not have access to $V^*(s)$ and we want to overcome waiting for G_t!)
- We are learning $V^*(s)$ by guessing the value estimates that the exact same function provides at s_{t+1}

We have seen the simplest form of how to update an estimate based on another estimate, but is this really a good idea?

- TD-Learning methods do not require a model of the environment, its rewards nor the s_{t+1} probability distributions
- They are implemented in an online, fully incremental fashion.
- We only need to wait one-step before starting learning (might also be a drawback!).
- TD methods are also sound and convergence of any fixed policy π to V^{π} is guaranteed. This policy might however not be the optimal one!
- A convergence proof on the speed of TD methods vs MC methods is still missing!

Eligibility Traces

So far we have seen that we can update a value function from the real final return G_t obtained at the end of an episode, or based on an immediate future estimate $V(s_{t+1})$.

- There is however an intermediate approach between MC and TD-Learning called TD-(λ):

$$G_t^n = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^n V_t(s_{t+n})$$

- With setting $0 \leq \gamma \leq 1$ we are able to compute updates based on n-step returns.
- The problem is that we would have to wait indefinitely for computing G_t^∞. This is also known as the problem of the forward view of TD(λ)

We could however partially overcome this problem by incrementally updating the eligibility trace of a state, which nicely allows us to perform n-step backups in an elegant way.

- For each state $s \in \mathcal{S}$ a trace $e_t(s)$ is kept in memory and initialized at 0.
- At each state we update $e_t(s)$ as

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1 & \text{if } s = s_t \end{cases}$$

The trace of each state is increased every time that particular state is visited and decreases exponentially otherwise due to λ.

If we again consider the TD-error as

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t),$$

then on every step we want to update a state in proportion to its eligibility trace, which results in the following update:

$$V(s_t) := V(s_t) + \alpha \delta_t e_t(s_t).$$

This results in a generic mechanism for learning from n-step returns.

- For $\lambda = 1$ we have MC-Learning.
- For $\lambda = 0$ we have TD-Learning.

Exploration vs Exploitation

- We know that if a complete model of an environment is given it is easy to compute an optimal policy (like Dynamic-Programming).
- As we have seen so far in the general RL setting this is unfortunately not the case, therefore learning an optimal policy becomes as process of trial and error.
- During this process the only feedback that is available to the agent is the reward that is obtained at the end of an action

So why is the exploration-exploitation dilemma so challenging?

- In RL the amount of feedback that the agent gets compared to e.g. SL is much less. There is no direct relationship between a learning sample and its output which allows us to evaluate a general performance
- In SL we usually deal with static datasets, similarly in Unsupervised Learning we learn a hopefully useful partition of an unlabeled dataset. In RL we have to deal with time.
- The RL "dataset" is considered as a moving target which makes it hard to quantify how well e.g. an objective function is minimized.

Let us assume that we have learned the optimal Q function:

$$Q^*(s,a) = \max_{\pi} \; Q^{\pi}(s,a) \text{ for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A}.$$

We know that this equation satisfies the Bellman optimality equation as given by:

$$Q^*(s_t,a_t) = \sum_{s_{t+1}} p(s_{t+1}|s_t,a_t) \Big[R(s_t,a_t,s_{t+1}) + \gamma \; \max_{a} \; Q^*(s_{t+1},a) \Big].$$

If such a function is learned it is straightforward to derive and optimal policy which does not require exploration

$$\pi^*(s_t) = \operatorname*{argmax}_{a \in \mathcal{A}} \; Q^{\pi}(s_t,a_t).$$

Unfortunately we first need to learn Q.

ϵ-greedy exploration

- An agent which always learns from the same experience will learn fast but will never increase its knowledge and performance.
- But once the agent has learned enough we do not want it to make sub-optimal decisions anymore since deviating from a greedy policy can cause some loss.
- The most popular way of dealing with this dilemma is the ϵ-greedy approach

$$a_t = \begin{cases} \max_a Q(s_t, a_t) & \text{with prob 1-}\epsilon \\ \text{random action with prob } \epsilon \end{cases}$$

where ϵ is annealed linearly over time to encourage exploration in the early training stages.

On-policy vs Off-policy learning

- We have seen how important it is to learn a policy and how this governs the behavior of an agent.
- Policies also define the underlying RL algorithm which we use when learning a value function.
- Specifically they are of interest when we need to compute a TD-error

$$\begin{aligned}\delta_t &= r_t + \gamma V(s_{t+1}) \\ &= r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a).\end{aligned}$$

- The first TD-error defines an on-policy RL algorithm since the estimate at $V(s_{t+1})$ will always be defined by the current policy the agent is following
- The second TD-error defines an off-policy RL algorithm since the TD-error $r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a)$ is always greedy because it is defined by the $\max$ operator. Remember that because of the exploration-exploitation trade-off the agent might not follow this greedy policy in practice
- Overall we can see off-policy algorithms as methods which learn many policies whereas on-policy ones only learn one policy. Both methods come with their pros and cons and the choice of a particular algorithm depends on the problem at hand.
- This difference starts to play a significant role when neural networks are used.

Q-Learning

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha \left[r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

- The most popular RL algorithm.
- Based on a variation of the simplest form of TD-Learning.
- Learns the Q function in an off-policy learning setting.
- (In the limit) Converges to the optimal policy regardless of the exploration strategy used.
- Suffers from numerous biases.

SARSA

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha\left[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right]$$

- An on-policy variation of Q-Learning.
- The TD-error is given by the estimate at s_{t+1} which is based on the current policy.
- When function approximators are used it diverges less when compared to Q-Learning.

QV-Learning

Jointly learns the state-value function V and the state-action value function Q

$$V(s_t) := V(s_t) + \alpha\big[r_t + \gamma V(s_{t+1}) - V(s_t))\big]e_t(s)$$

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha\big[r_t + \gamma V(s_{t+1}) - Q(s_t, a_t)\big]$$

- Learns on-policy.
- Learning two value functions might accelerate learning.
- Uses the same TD-error to learn two value functions.

Actor-Critic Learning

A branch of TD methods which keep the policy (Actor) from a learned value function (Critic).

The TD error

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

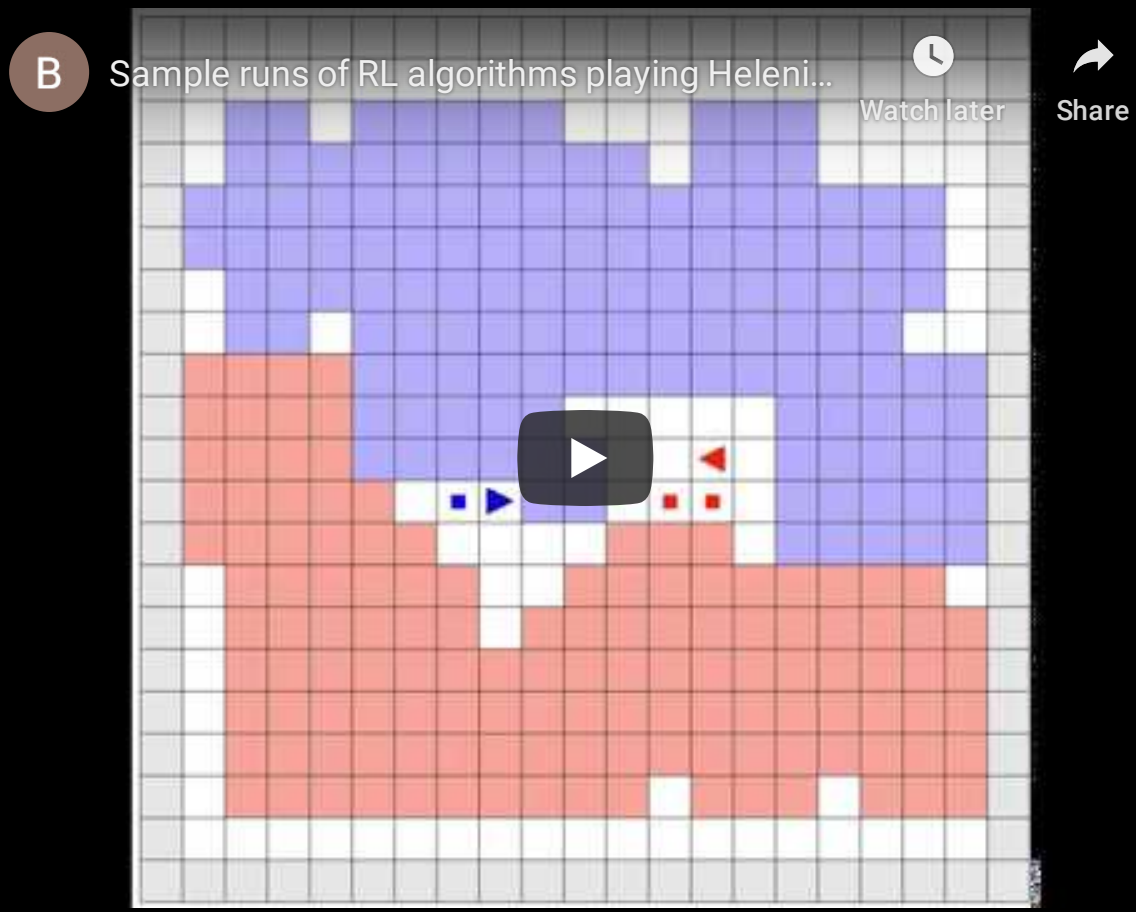
is used by the critic to evaluate the action which was taken by the actor.

With this error we can strengthen or weaken the selection of an action by modifying the preference of selecting an action

$$p(s_t, a_t) := p(s_t, a_t) + \beta \delta_t$$

where β determines the size of the update.

A practical example



Deep Reinforcement Learning (DRL)

The need for function approximators

- All previously mentioned RL algorithms work well when the size of the MDP is relatively small.
- In practice value functions are usually represented by a look-up table where for example each state-action pair has an entry representing $Q(s, a)$.
- Storing such data structures is however not possible when the state-action space is too large.
- We want to replace a value function with a function approximator e.g. a neural network:
  - $V(s; \theta) \approx V^{\pi}(s)$
  - $Q(s, a; \theta) \approx Q^{\pi}(s, a)$

Using a function approximator allows us to drastically increase the complexity of the MDP

- TD-Gammon: 10^{20} possible states
- Alpha-Go: 10^{20} possible states
- Many real world situations ranging from autonomous driving cars, to personalized web-services

In these examples RL methods with function approximators are mostly used in combination with other AI techniques e.g. tree-search

There are no restrictions on the type of function approximator that is used:

- Linear vs Non-Linear
- Neural Networks
- Regression Trees

Deep neural networks as function approximators

- Despite the recent hype of DRL the combination between MLPs and RL algorithms is not new.
- Before DRL existed this research field was known as Connectionist Reinforcement Learning
- We start speaking of DRL when more complex neural architectures are used as function approximators, one above all Convolutional Neural Networks (CNNs).
- CNNs are then combined with other techniques which make DRL algorithms stable and will correspond to a DRL cooking recipe.

Most of the DRL algorithms we will see from now on use a CNN as a function approximator

- Universal function approximators
- Powerful feature extractors
- This allows us to learn an appoximation of a value function from raw dimensional feature inputs.

The CNN itself will be directly modeling the value function, or in case of policy gradient methods it will represent the policy of our agent.

One question needs to be answered, how do we exactly train a neural network on a RL problem?

Let us consider the Q-Learning algorithm

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha \left[r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a) - Q(s_t, a_t) \right].$$

This update rule is very different from the objective functions which we have encountered so far in the course:

- There are no parameters θ defining a neural network.
- There is no loss function $\mathcal{L}(\theta)$.
- What should we minimize?

The answer comes when considering the TD-error that defines Q-Learning update's rule

$$\delta_t = r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a) - Q(s_t, a_t).$$

This quantity is telling us to update our current $Q(s_t, a_t)$ estimate with respect to the greedy s_{t+1} one, which is an idea that resembles the Mean Squared Error loss

$$\mathcal{L}(y, f(x)) = (y - f(x))^2$$

which we can adapt to obtain the following objective function:

$$\mathcal{L}(\theta) = \left(r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta) - Q(s_t, a_t; \theta)\right)^2$$

where θ represents the neural network approximating the Q function.

DQN

The popular DQN algorithm integrates two additional components into the previous objective function which ensure stable and robust training:

- Experience Replay
- Target Networks

$$\mathcal{L}(\theta) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)} \left[\left(r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta^-) - Q(s_t, a_t; \theta) \right)^2 \right]$$

Given a training iteration i, differentiating this objective function with respect to θ gives the following gradient:

$$\nabla_{\theta_i} y_t^{DQN}(\theta_i) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)} \left[\left(r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta_{i-1}^-) - Q(s_t, a_t; \theta_i) \right) \nabla_{\theta_i} Q(s_t, a_t; \theta_i) \right]$$

Integrating the popular Q-Learning algorithm with an experience replay memory buffer and a separate network ensures that training is stable only until a certain point.

- DRL algorithms suffer from the same problems that characterize their tabular counterparts.
- This is especially true for TD methods which are built upon biased expectations.
- Biases get even more enhanced because of the use of function approximators which can make training even more unstable.

An example of these biases is the overestimation bias of the Q function that characterizes Q-Learning and therefore DQN.

DDQN

DQN's objective function tells us that the same set of actions is used when selecting and evaluating an action.

This becomes clearer if we look at DQN's target:

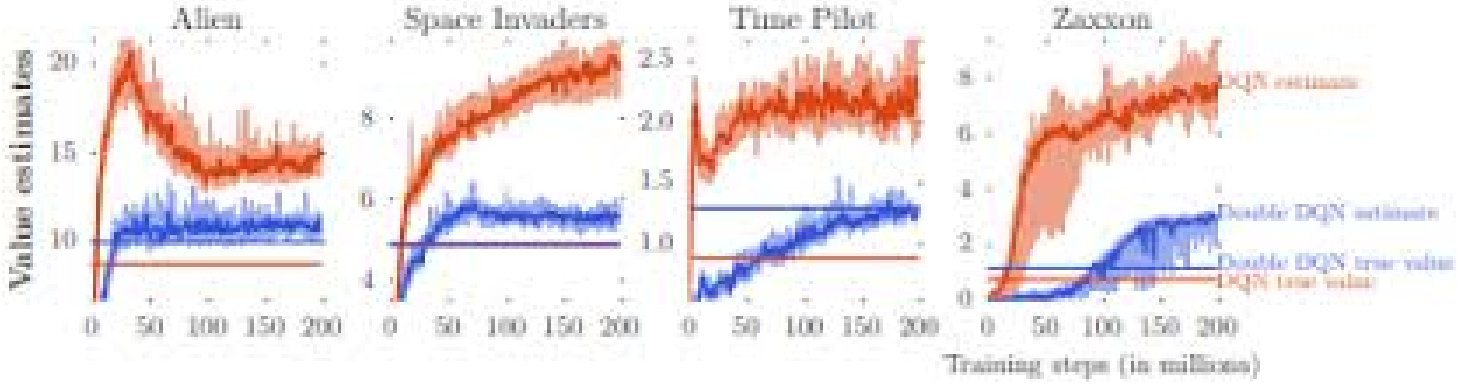
$$\delta_t = r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta^-)$$

and rewrite it as:

$$\delta_t = r_t + \gamma \, Q(s_{t+1}, \operatorname*{argmax}_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta); \theta^-).$$

DQN tends to approximate the expected maximum value of a state, instead of its maximum expected value, resulting in Q values that are overestimated.

DDQN partially solves this problem by untangling the selection and the evaluation of an action by taking advantage of the target network.



The overestimation in practice.

Credits: Van Hasselt et al, 2016.

Prioritized Experience Replay (PER)

The original formulation of experience replay memory buffer presented some limitations:

- A large amount of the $\langle s_t, a_t, r_t, s_{t+1} \rangle$ trajectories that are stored in the buffer might correspond to similar situations.
- Each trajectory is treated as equally important, but when it comes to learning there might be some situations which are more informative than others.
- Assuming there is more informative trajectories than others, uniform sampling techniques will still treat all trajectories equally.

PER addresses all these remarks by introducing a stochastic sampling method that interpolates between pure greedy prioritization and uniform random sampling.

The probability of sampling a transition i is defined as

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

where p_i is the priority of a transition and α is the degree of prioritization we want to use (if $\alpha = 0$ we are in the uniform sampling case).

The priority p_i is defined by the TD-error associated to a transition $p_i = |\delta_i| + \epsilon$.

The DQV-Family of Algorithms

So far we have considered neural networks which approximate the state-action value function only. We know that in addition to the Q function there also is the V which provides us with significant information.

The idea of jointly approximating two value functions over one is what characterizes the DQV-family of DRL algorithms.

- $V(s_t)$ is simpler to learn than $Q(s_t, a_t)$
- The convergence of $V(s_t)$ can help the convergence of $Q(s_t, a_t)$
- Regressing the Q function towards itself leads to biased Q-estimates

Defines two neural networks which are responsible for each learning an approximation of either the state-value function or the state-action value function

- $V(s_t; \Phi) \approx V(s)$
- $Q(s_t, a_t; \theta) \approx Q(s, a)$

DQV-Learning is an on-policy DRL algorithm which learns the state-value function with the simplest form of TD-Learning

$$\mathcal{L}(\Phi) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)} \left[\left(r_t + \gamma V(s_{t+1}; \Phi^-) - V(s_t; \Phi) \right)^2 \right]$$

while the state-action value function is learned as follows:

$$\mathcal{L}(\theta) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)} \left[\left(r_t + \gamma V(s_{t+1}; \Phi^-) - Q(s_t, a_t; \theta) \right)^2 \right].$$

DQV has the interesting property of requiring the computation of one TD-error which can be used for learning two value functions simultaneously.

The DQV-Family of Algorithms

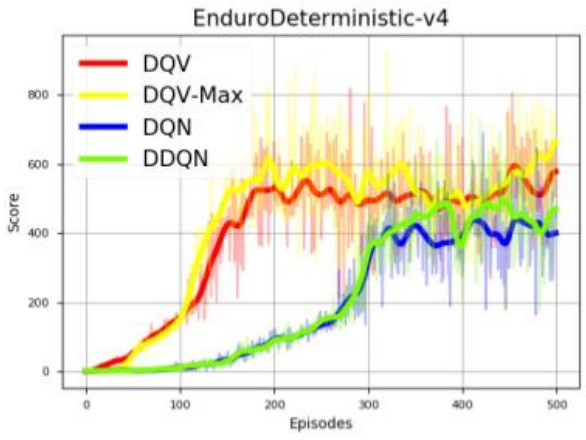
DQV-Max Learning is an off-policy DRL algorithm which combines ideas from DQV and DQN. The idea is to reintroduce DQN's $\max_{a \in \mathcal{A}} Q(s_{t+1}, a)$ operator and use it for learning the state-value function

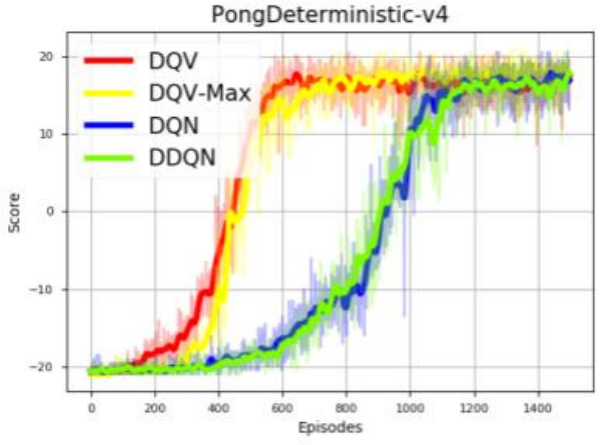
$$\mathcal{L}(\Phi) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)} \left[\left(r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta^-) - V(s_t; \Phi) \right)^2 \right]$$

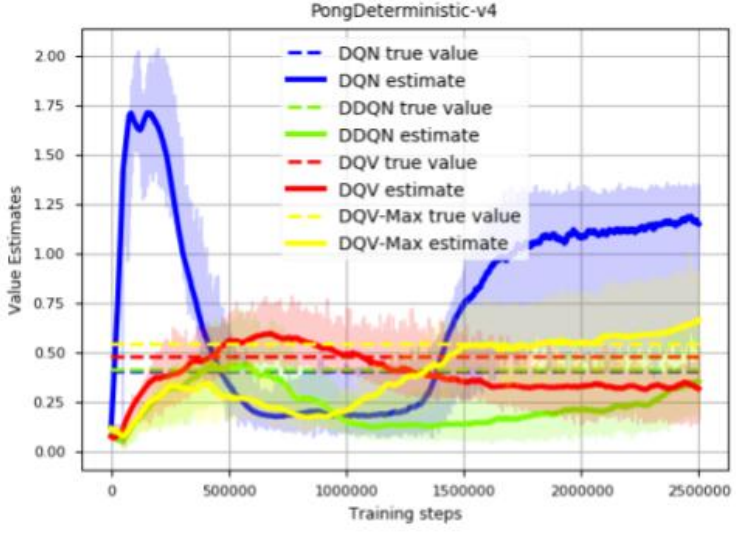
while we keep learning the state-action value function as we did with DQV

$$\mathcal{L}(\theta) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)} \left[\left(r_t + \gamma V(s_{t+1}; \Phi) - Q(s_t, a_t; \theta) \right)^2 \right].$$

Note that we are now computing two TD-errors and not one anymore







Credits: Sabatelli et al, 2019.

The DQV family of algorithms is characterized by the idea of learning different value estimates and then transfer them in the form of TD-errors from one value function to another.

This presents some nice benefits:

- Overall obtain higher cumulative rewards on popular DRL benchmarks
- Converge significantly faster than methods which only learn one value function
- They suffer less from the overestimation bias (especially DQV)

But comes at the price of being:

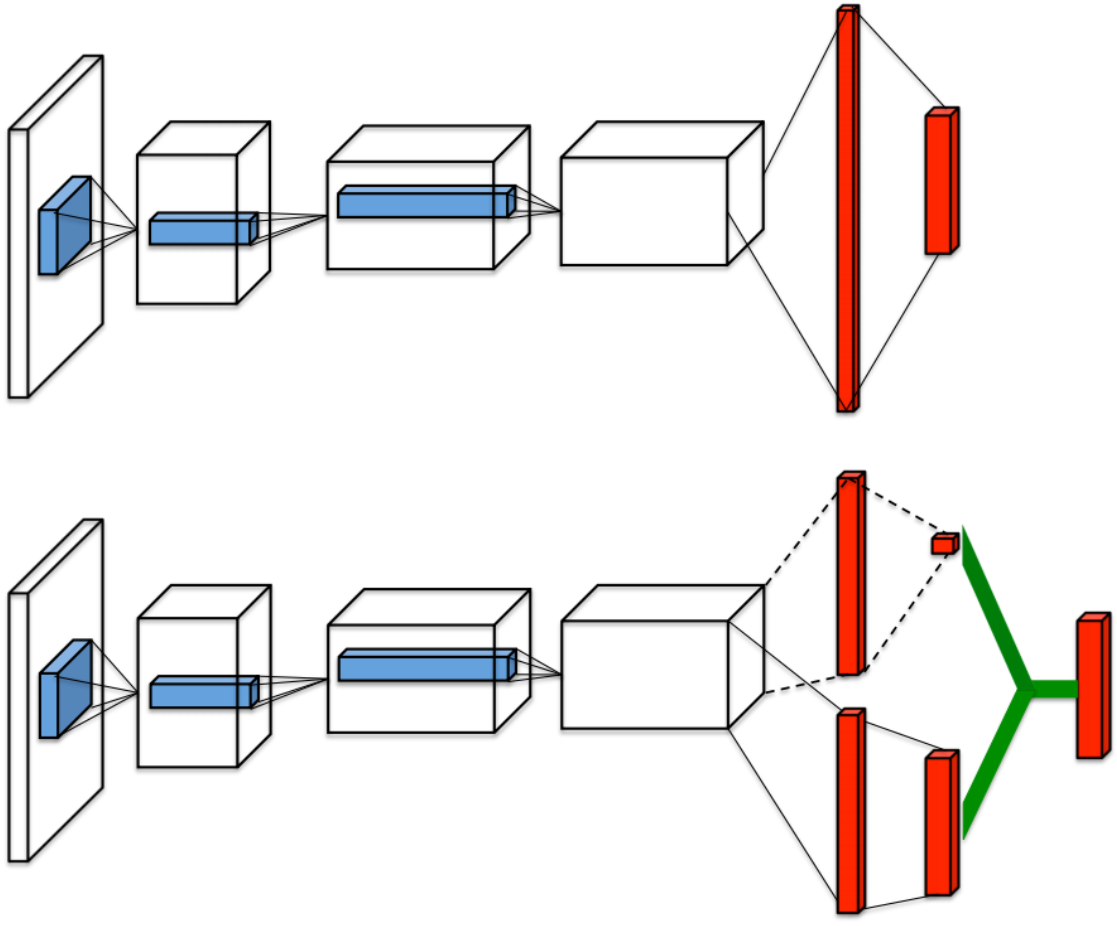
- Memory wise more expensive
- Computationally more expensive
- Greater risk of encountering divergence issues

Dueling Architectures

The idea of learning multiple value functions is also used by the Dueling network architecture. In addition to the Q and V estimates this kind of method also learns the Advantage function:

$$A^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t)$$

- Intuitively $A^{\pi}(s_t, a_t)$ tells us how much of a good idea it was to select action a_t in state s_t.
- All estimates are computed within the same neural network (θ), which has different streams that are responsible for the different value estimates



Credits: Wang et al, 2015.

The different Q values will depend from all the different "heads" of the network and have to satisfy the following forward mapping

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

If this equality is satisfied this intuitively means that the Q values that are learned are maximizing the correct state-value estimates (identifiability issue).

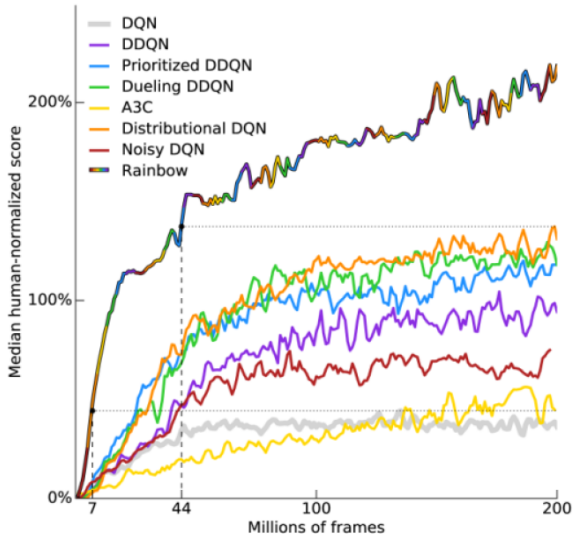
- Dueling Architectures are trained in a DDQN fashion
- Uses Prioritized Experience Replay
- Achieved SOA results on the popular Atari-2600 benchmark

Rainbow

We have seen that DRL is characterized by several independent improvements which combine RL theory and deep neural networks.

The idea of the Rainbow agent is to combine all these improvements into a super-agent which gets the best out of six different DRL contributions:

- DDQN
- PER
- Dueling Architectures
- A3C
- Distributional DRL
- Noisy Exploration



After many experiments and lots of hyperparameter tuning the Rainbow agent significantly outperforms all previous DRL methods while also being more data efficient.

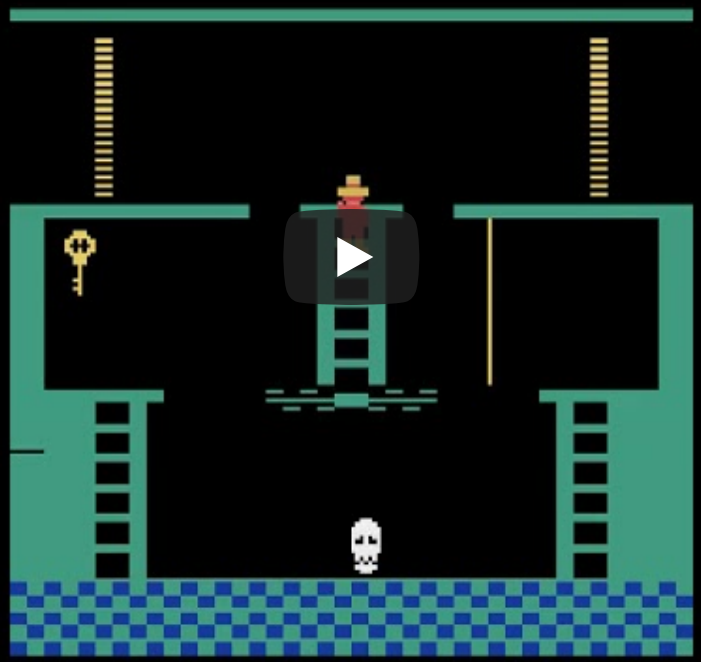
Despite all the improvements Rainbow still failed to master some of the games of the Atari benchmark. This has only recently been overcome with the Agent-57 algorithm.

Credits: Hessel et al, 2017.

Montezuma's Revenge in 2012

DQN playing Montezuma's Revenge

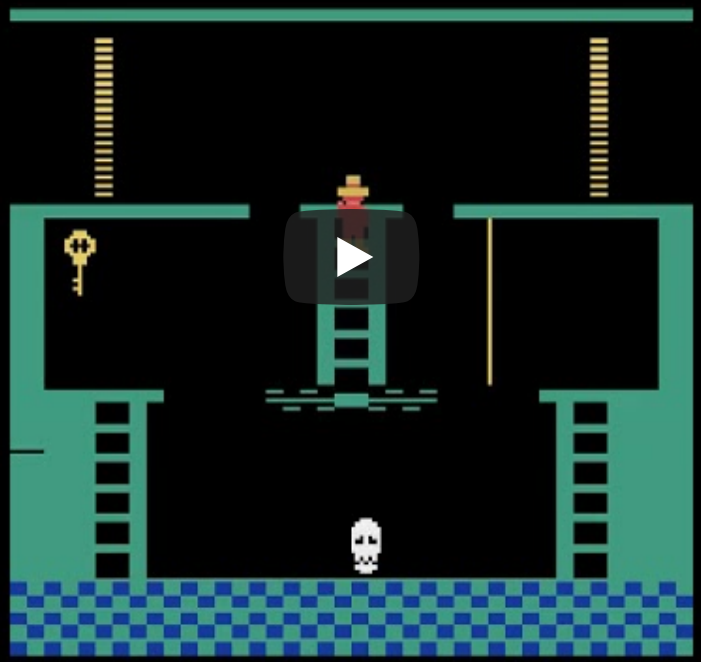
1/2 Watch later Share



Montezuma's Revenge in 2020

DQN playing Montezuma's Revenge

1/2 Watch later Share



The Deadly Triad of DRL

But is DRL as hard and tricky as we have been warning you? :P

- So far I have presented you a set of results that seem to make it easy to be a successful DRL researcher, but why is it then that DRL became so popular only recently?
- If we go back to tabular RL algorithms there is no mention of
  - Experience Replay memory buffers
  - Target networks
  - PER

DRL can work well but it requires RL algorithms to be coupled with a lot of additional techniques which ensure that training can be done in a stable way.

The cause of potential issues is known as the Deadly Triad of DRL.

- Function Approximators: we are learning an approximation of a value-function.
- Bootstrapping: when the estimates of a value function are learned wrt other estimates which come from the same value function.
- Off-policy training: training on trajectories that are different from the one which are being followed.

If any of the three elements is absent then instability can be avoided, but it is hard to choose which components to discard.

- Function Approximators: clearly we cannot give up on this, if we want to scale RL up we need at least a linear function approximator
- Bootstrapping: it is possible to find alternatives (n-step updates or MC methods) but this has to be done at the cost of computational data and efficiency, besides TD-Learning seems to work better in practice
- Off-policy Learning: we can create on-policy DRL algorithms (Deep-SARSA, DQV, ...) but training them in a robust way makes these methods closer to off-policy algorithms than one might think. Furthermore if we want to learn in parallel off-policy methods are the only way to go.

The end.

References

- Richard Sutton and Samuel Barto, Reinforcement Learning: an introduction, second edition
- Richard Sutton Learning to predict by the methods of temporal differences
- Marco Wiering and Martijn van Otterlo, Reinforcement Learning
- Watkins Christopher and Peter Dayan, Q-Learning
- Hado Van Hasselt Double Q-Learning
- Marco Wiering and Hado Van Hasselt The QV Family Compared to Other Reinforcement Learning Algorithms.html)
- Mnih et al. Human-level control through deep reinforcement learning
- Van Hasselt et al. Deep Reinforcement Learning with Double Q-learning
- Sabatelli et al. Deep Quality-Value (DQV) Learning
- Sabatelli et al. Towards characterizing a new family of Deep Reinforcement Learning algorithms
- Schaul et al. Prioritized Experience Replay
- Hessel et al. Rainbow: Combining Improvements in Deep Reinforcement Learning
- Rashid et al. QMIX: Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning
- Van Hasselt et al. Deep Reinforcement Learning and the Deadly Triad