Deep Reinforcement Learning

From Markov Decision Processes to Deep-Q Networks

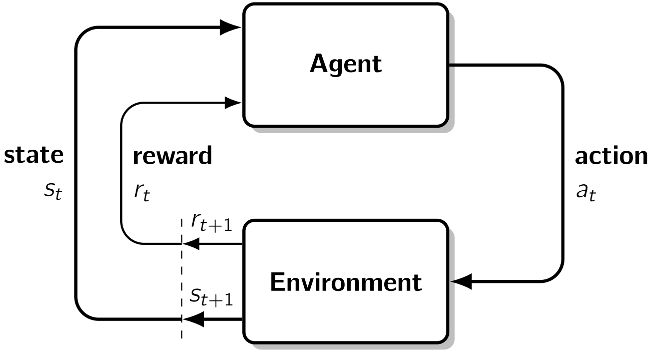
Matthia Sabatelli

March 21st, 2022

Today's Agenda

1. Markov Decision Processes (MDPs)
2. Learning Value Functions
3. Function Approximation
4. Deep Reinforcement Learning

Markov Decision Processes



Markov Decision Processes

The mathematical framework of Reinforcement Learning:

- A set of possible states $\mathcal{S}$ where $s_t \in \mathcal{S}$ is the current state
- A set of possible actions $\mathcal{A}$ where $a_t \in \mathcal{A}$ is the current action
- A transition function $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$
- A reward function $\Re : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ which returns r_t

These components allows us to define a Markov Decision Process: $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, r \rangle$.

Markov Decision Processes

$\Rightarrow$ What is so special about MDPs?

In a Markovian environment the conditional distribution of the next state of the process only depends from the current state of the process.

$$p(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \dots) = p(s_{t+1}|s_t, a_t).$$

Interestingly, the same property also holds for the reward that the agent will get:

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

Markov Decision Processes

How does an agent interact with its environment?

Through a probability distribution over $a \in \mathcal{A}(s)$ for each $s \in \mathcal{S}$:

$$\pi(a|s) = \Pr\{a_t = a | s_t = s\}, \text{ for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A}.$$

which more simply can be seen as a mapping from states to actions that determines the behaviour of the agent:

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

Markov Decision Processes

Once we have policy π we can let the agent interact with the environment, which results in an episode:

$$\langle (s_t, a_t, r_t, s_{t+1}) \rangle, t = 0, \dots, T-1$$

where T is a random variable defining the length of the episode

One $\langle s_t, a_t, r_t, s_{t+1} \rangle$ is called a trajectory τ

Markov Decision Processes

Why do we want an agent to interact with the environment?

- We would like it to master a certain task
- Ideally it could discover solutions that are unknown to us
- Biologically plausible form of learning

How do we formalize this mathematically?

In its easiest form we can define such goal as maximizing the sequence of r_t returned by $\Re$

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

Markov Decision Processes

However the definition of G_t has some limitations:

- Assumes that episodes are finite
- Many real world applications are instead continuous, therefore $T = \infty$
- As a result G_t can be infinite as well

We need a slightly more complex definition of G_t based on the concept of discounting modeled by γ

Markov Decision Processes

γ allows us to define the notion of discounted cumulative reward

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

- γ determines the value of future rewards as $0 \leq \gamma < 1$
- Makes G_t finite as long as $\gamma < 1$ and r_t is constant

$$G_t = \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1 - \gamma}$$

Markov Decision Processes

One of the most important concepts of Reinforcement Learning: value

- Directly connected to the notion of G_t
- We can define the value of a state s, of an action a, and even of a policy π
- Allows the introduction of value functions
  - state-value function $V(s)$
  - state-action value function $Q(s, a)$

Markov Decision Processes

The state-value function

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}\Big[G_t \;\Big|\; s_t = s, \pi \Big] \\ &= \mathbb{E}\Bigg[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \;\Bigg|\; s_t = s, \pi \Bigg] \end{aligned}$$

- The easiest value function to learn
- Intuitively it tells us *"how good/bad"* every state s visited by policy π is
- This *"goodness"* is expressed with respect to G_t

Markov Decision Processes

The state-action value function

$$
\begin{aligned}
Q^{\pi}(s,a) &= \mathbb{E}\left[G_t \,\middle|\, s_t = s,\ a_t = a, \pi \right] \\
&= \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, s_t = s,\ a_t = a, \pi \right]
\end{aligned}
$$

- Is much more informative compared to $V^{\pi}(s)$
- Intuitively tells us *how good/bad* taking action a in state s is
- Plays a crucial role in the development of Reinforcement Learning algorithms

Markov Decision Processes

An interesting property of these value functions is that they satisfy a recursive equality

$$\begin{aligned} V^\pi(s) &= \mathbb{E}\left[\sum_k^\infty \gamma^k r_{t+k+1} \,\middle|\, s_t = s, \pi \right] \\ &= \sum_a \pi(s,a) \sum_{s_{t+1}} p(s_{t+1}|s,a) \left[\Re(s_t, a, s_{t+1}) + \gamma V^\pi(s_{t+1}) \right] \end{aligned}$$

Markov Decision Processes

An interesting property of these value functions is that they satisfy a recursive equality

$$
\begin{aligned}
V^{\pi}(s) &= \mathbb{E}\left[\sum_{k}^{\infty} \gamma^{k} r_{t+k+1} \,\middle|\, s_t = s, \pi \right] \\
&= \sum_{a} \pi(s, a) \sum_{s_{t+1}} p(s_{t+1}|s, a) \left[\Re(s_t, a, s_{t+1}) + \gamma V^{\pi}(s_{t+1}) \right]
\end{aligned}
$$

Markov Decision Processes

Where does this recursive property come from?

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

Markov Decision Processes

Where does this recursive property come from?

$$\begin{aligned} V^\pi(s) &= \mathbb{E}\Big[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \Big| s_t = s, \pi\Big] \\ &= \mathbb{E}\Big[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \Big| s_t = s, \pi\Big] \end{aligned}$$

.

Markov Decision Processes

Where does this recursive property come from?

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}\Big[\sum_{k=0}^{\infty}\gamma^k r_{t+k}\Big|s_t = s, \pi\Big] \\ &= \mathbb{E}\Big[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \Big| s_t = s, \pi\Big] \\ &= \mathbb{E}\Big[r_t + \gamma(r_{t+1} + \gamma r_{t+2} + \ldots)\Big| s_t = s, \pi\Big] \end{aligned}$$

.

Markov Decision Processes

Where does this recursive property come from?

$$
\begin{aligned}
V^{\pi}(s) &= \mathbb{E}\big[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \big| s_t = s, \pi\big] \\
&= \mathbb{E}\big[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots \big| s_t = s, \pi\big] \\
&= \mathbb{E}\big[r_t + \gamma(r_{t+1} + \gamma r_{t+2} + \dots)\big| s_t = s, \pi\big] \\
&= \mathbb{E}\big[r_t + \gamma V^{\pi}(s_{t+1})\big| s_t = s, \pi\big]
\end{aligned}
$$

.

Markov Decision Processes

Where does this recursive property come from?

$$\begin{aligned}
V^{\pi}(s) &= \mathbb{E}\Big[\sum_{k=0}^{\infty}\gamma^{k} r_{t+k}\Big| s_t = s, \pi\Big] \\
&= \mathbb{E}\Big[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \Big| s_t = s, \pi\Big] \\
&= \mathbb{E}\Big[r_t + \gamma(r_{t+1} + \gamma r_{t+2} + \ldots)\Big| s_t = s, \pi\Big] \\
&= \mathbb{E}\Big[r_t + \gamma V^{\pi}(s_{t+1})\Big| s_t = s, \pi\Big] \\
&= \sum_{a}\pi(a|s)\sum_{s_{t+1}} p(s_{t+1}|s, a)\Big[\Re(s_t, a, s_{t+1}) + \gamma V^{\pi}(s_{t+1})\Big].
\end{aligned}$$

Markov Decision Processes

Solving a Reinforcement Learning problem intuitively means finding a policy π that achieves a lot of reward:

- What makes a certain policy π better than another policy π'?
- How can we rank different policies?

We can answer these questions thanks to the $V(s)$ and the $Q(s, a)$ functions. There is always one policy that is better or equal than all other policies

$$\pi \geq \pi' \text{ iff } V^{\pi}(s) \geq V^{\pi'}(s) \text{ for all } s \in \mathcal{S}$$

The best possible policy is the optimal policy π^*

Markov Decision Processes

- How do we find the optimal policy π^*?
- By maximizing the state-value and state-action value functions results in the optimal value functions

$$V^*(s) = \max_\pi V^\pi(s) \qquad Q^*(s,a) = \max_\pi Q^\pi(s,a)$$

which, if expressed in a recursive form, result in the Bellman optimality equations

Learning Value Functions

In the typical Reinforcement Learning scenario no complete knowledge about the MDP is available:

- Set of possible states $\mathcal{S}$ ✓
- Set of possible actions $\mathcal{A}$ ✓
- Transition Function $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ ✗
- Reward Function $\Re : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ ✗

Learning Value Functions

$\Rightarrow$ As a result ...

- We need to overcome the lack of information of $\mathcal{M}$
- We can do this through experience
- Gathering experience means sampling states s, actions a and rewards r from the environment
- Recall the concept of trajectory τ $\langle s_t, a_t, r_t, s_{t+1} \rangle$

The transition function $\mathcal{P}$ and the reward function $\Re$ are usually called the model of the environment

$\mathcal{P}$ and $\Re$ can be learned $\Rightarrow$ model-based Reinforcement Learning

Learning Value Functions

We start by considering model-free Reinforcement Learning techniques:

- We learn without any prior knowledge of the environment
- Trajectories, and therefore experience, is sufficient for learning
- We "only" need a value function $Q(s, a)$, which is arguably easier to learn than the model

The effectiveness of model-free Reinforcement Learning algorithms highly depends from how much experience the agent is able to gather!

Learning Value Functions

Monte Carlo methods:

- Can be used for learning $V^\pi(s)$ as well as $Q^\pi(s, a)$
- Although in this lecture we only focus on learning $V^\pi(s)$
- The key idea is to learn through sampling returns

We assume that we are always dealing with episodic tasks i.e. episodes eventually terminate.

$$\langle (s_t, a_t, r_t, s_{t+1}) \rangle, t = 0, \dots, T - 1$$

Learning Value Functions

Let us again consider the notion of expected 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 goal is to learn the state-value function of a given policy $V^\pi(s)$: Monte Carlo Prediction
- We do this with respect to the G_t that is obtained by following π

Learning Value Functions

Learning $V^{\pi}(s)$ involves the following steps:

- Before learning, each state has its own value $V(s_t)$
- We follow policy π until an episode terminates
- We compute the discounted return G_t that was obtained by π

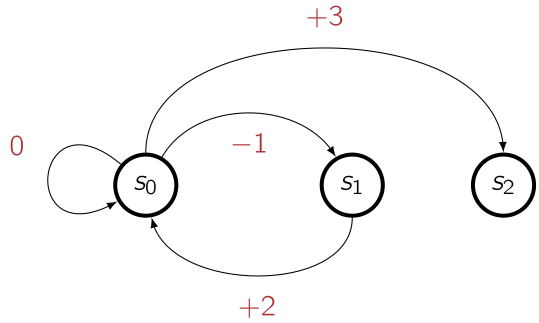
We can change the value of each state $V(s_t)$ based on G_t

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

where $\alpha \in [0, 1]$ is the learning rate parameter.

Monte Carlo (MC) Methods

Let us consider the following MDP:



- We have policy $\pi : s0 \rightarrow s1 \rightarrow s0 \rightarrow s2$
- π results in the sequence of rewards $-1, +2, +3$
- The starting value of each state is 0, $\gamma = 0.99$ and $\alpha = 0.5$

Monte Carlo (MC) Methods

We know that G_t for starting in $s0$ is

$$
\begin{aligned}
G_t &= \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \\
&= -1 + \gamma 2 + \gamma^2 3 \approx 3.92
\end{aligned}
$$

Therefore

$$
\begin{aligned}
V(s_t) &:= V(s_t) + \alpha \left[G_t - V(s_t)\right] \\
V(0) &:= V(0) + \alpha \left[3.92 - V(0)\right] \approx 1.96
\end{aligned}
$$

Learning Value Functions

Pros & Cons of Monte Carlo Methods:

- Yield unbiased updates thanks to G_t ✓
- Scale well to function approximators ✓
- Learning can be very slow as one has to wait until the very end of an episode ✗
- There can be large variance in the value updates ✗

Learning Value Functions

An alternative option to Monte Carlo Methods is based on Temporal Difference (TD)-Learning a family of methods where we do not have to wait until the end of an episode before updating a value estimate

- We only need to wait until the next step
- At $t + 1$ we immediately create a target for learning called the TD-target
- We do this by using the observed reward r_t and a future value estimate e.g. $V(s_{t+1})$

Learning Value Functions

$\Rightarrow$ Let us again consider the problem of estimating $V^\pi(s)$: We change the value of each state $V(s_t)$ with respect to $t+1$ only:

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

- It is clear that we only learn by *looking ahead* in the future one single step
- This is called TD(0) or *one-step TD*

Learning Value Functions

The key idea of TD-Learning it to learn through bootsrapping:

- We update the value of a state with respect to the value of its successor state only
- Ideally we would like to use $V^{\pi}(s_{t+1})$ for learning but it is unfortunately unknown

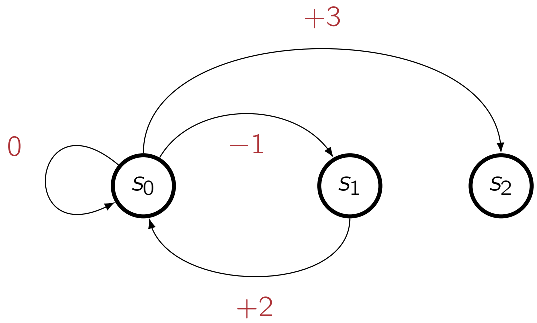
$$V(s_t) := V(s_t) + \alpha\big[r_t + \gamma V^{\pi}(s_{t+1}) - V(s_t)\big].$$

Therefore we replace it with a guess instead:

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

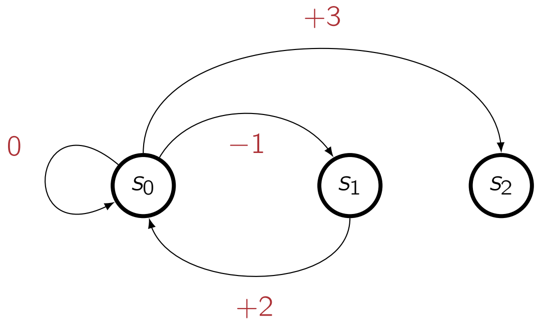
Temporal Difference (TD) Learning

Let us go back to the previous MDP:



- We again have policy $\pi : s0 \rightarrow s1 \rightarrow s0 \rightarrow s2$
- π results in the sequence of rewards $-1, +2, +3$
- The starting value of each state is 0, $\gamma = 0.99$ and $\alpha = 0.5$

Temporal Difference (TD) Learning



Our first state transition is $s0 \rightarrow s1$

$$
\begin{aligned}
V(s_t) &:= V(s_t) + \alpha\big[r_t + \gamma V(s_{t+1}) - V(s_t)\big] \\
V(s0) &:= V(s0) + \alpha\big[r_t + \gamma V(s1) - V(s0)\big] \\
V(s0) &:= -0.5
\end{aligned}
$$

Learning Value Functions

Let us now consider the problem of learning the state-action value function $Q^{\pi}(s, a)$

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

The arguably most popular algorithm for learning $Q^{\pi}(s, a)$ is Q-Learning

Learning Value Functions

Q-Learning

- Is an *off-policy* learning algorithm
- Able of converging to the optimal state-action value function $Q^*(s, a)$ with probability 1
- Works by keeping track of an estimate of the state-action value function $Q : \mathcal{S} \times \mathcal{A} \rightarrow \Re$

The update rule of each visited state-action pair used by Q-Learning is:

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

Learning Value Functions

How does Q-Learning work?

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

- We create the TD-error δ_t by using the $\max_{a \in \mathcal{A}}$ operator

Learning Value Functions

How does Q-Learning work?

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

- We create the TD-error δ_t by using the $\max_{a \in \mathcal{A}}$ operator
- We always update $Q(s_t, a_t)$ with respect to a greedy policy

Learning Value Functions

How does Q-Learning work?

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

- We create the TD-error δ_t by using the $\max_{a \in \mathcal{A}}$ operator
- We always update $Q(s_t, a_t)$ with respect to a greedy policy
- Even if the agent is exploring the environment, learning is done greedily $\rightarrow$ *off-policy* learning

Learning Value Functions

We can also learn the $Q^{\pi}(s, a)$ function in a way which is more similar to how we learned $V^{\pi}(s)$ beforehand: SARSA

- Is an *on-policy* learning algorithm
- Also works by keeping track of $Q : \mathcal{S} \times \mathcal{A} \rightarrow \Re$
- Has different convergence properties

The update rule of each visited state-action pair used by SARSA is:

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

Learning Value Functions

Pros & Cons of TD-Learning methods:

- They do not yield unbiased estimates ✗
- But there is small variance in the updates ✓
- Learning starts faster ✓
- It can be complicated to combine them with non-linear function approximators ✗

Both approaches can be combined resulting in TD(λ) algorithms!

$\Rightarrow$ TD-Learning methods empirically work better than MC algorithms but a formal proof about why this is the case is missing ...

Function Approximation

Q-Learning like algorithms are typically implemented in a tabular fashion:

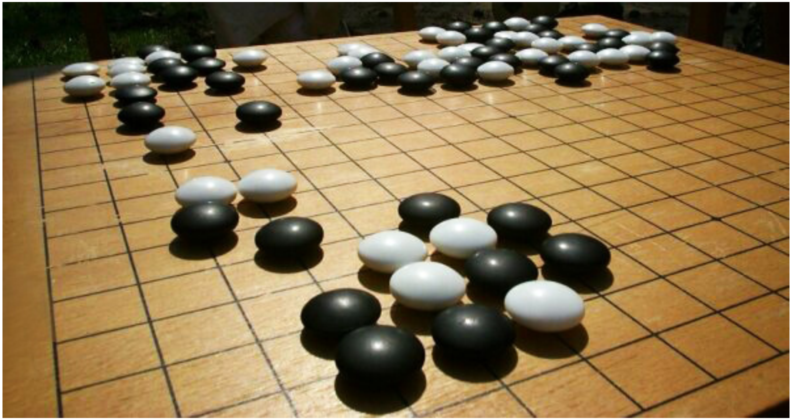
- If the goal is to learn $V^{\pi}(s)$ we use a table of size $|\mathcal{S}|$
- If the goal is to learn $Q^{\pi}(s, a)$ we use a table of size $|\mathcal{S} \times \mathcal{A}|$

It is easy to see the limitations of this approach:

1. Unfeasible when the state-action space is large
2. Impossible to use in a continous setting
3. Requires a discretization of the environment
4. Lacks generalization

Function Approximation

A simple example: the game of Go



Function Approximation

A simple example: the game of Go

- The AlphaGo and AlphaZero programs are a typical example of the recent success of Reinforcement Learning
- Both programs heavily rely on a function approximator

Why?

- The size of the Go board is 19×19
- On each location there can, or can't, be a stone (white or black)
- State space $|\mathcal{S}| = 3^{19 \times 19} = 3^{361}$

Impossible to learn any value function!

Function Approximation

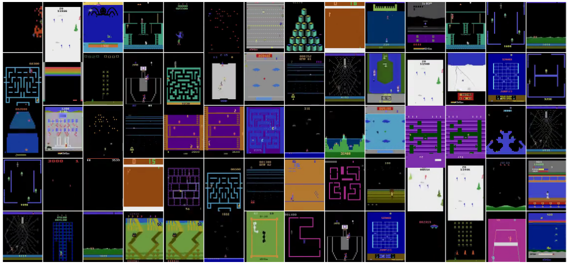
Another example: the Atari-2600 benchmark

nature
THE INTERNATIONAL WEEKLY JOURNAL OF SCIENCE
LEARNING CURVE
Self-taught AI software attains human-level performance in video games
PAGES 486 & 529

Function Approximation

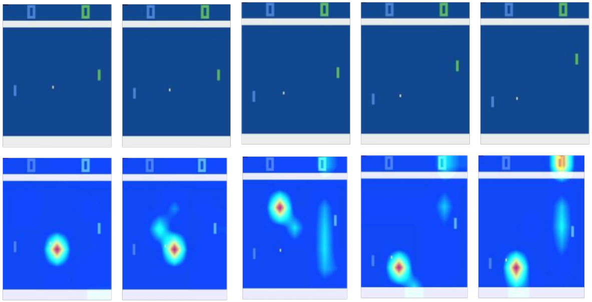
Another example: the Atari-2600 benchmark

- The first Deep Reinforcement Learning agent of mastering these games is based on a Convolutional Neural Network
- The network was trained on the raw input pixels of the game



Function Approximation

Let's take a look at some states of the `Pong` game:



- All states look very similar among each other
- Small portions of the state-space are actually informative
- Despite some pre-processing operations the state-space stays highly dimensional

Function Approximation

Fortunately we can overcome the aforementioned issues by including a function approximator in the reinforcement learning cooking recipe!

Function Approximators

We do not represent a value function as a table anymore, but rather with a parametrized functional form with weight vector $\mathbf{w} \in \mathbb{R}^d$.
Therefore we are now interested in learning:

$$V^{\pi}(s) \approx \hat{V}^{\pi}(s; \mathbf{w})$$

Function Approximation

We will see that $\hat{V}(s; \mathbf{w})$ can come in numerous forms but the key ideas behind using a function approximator are always the same:

1. We want to overcome the computational burdens that come from large state $|\mathcal{S}|$ and action $|\mathcal{A}|$ spaces
2. We wish to represent states through informative features
3. Ideally we would like to learn an approximation of a value function which generalizes well across states

$\Rightarrow$ Now that we have built some intuition around why function approximation is useful let us dive deeper into this family of techniques ...

Function Approximation

Any function approximator comes is the following form:

$$y = f(\mathbf{x}; \mathbf{w})$$

The easiest form of a function approximator that we can use is a linear function which is linear in the components of $\mathbf{x}$:

For example we can represent the value of a state as a two-dimensional feature vector: $\mathbf{x} = [f_1(s), f_2(s), ..., f_i(s)] \in \mathbb{R}^2$ which we can train via SGD

Function Approximation

A simple example:

- We wish to evaluate the state-value function $V^{\pi}(s)$
- We assume that the correct values of a state are known
- We know the true value of a state under policy π

SGD in practice

We wish to minimize the estimates made by $\hat{V}(s; \mathbf{w})$ with respect to $V^{\pi}(s)$:

$$\begin{aligned}\mathbf{w}_{t+1} &\doteq \mathbf{w}_t - \frac{1}{2}\alpha\nabla\Big[V^{\pi}(s_t) - \hat{V}(s; \mathbf{w}_t)\Big]^2 \\ &= \mathbf{w}_t + \alpha\Big[V^{\pi}(s_t) - \hat{V}(s_t; \mathbf{w}_t)\Big]\nabla\hat{V}(s; \mathbf{w}_t)\end{aligned}$$

where α is the step-size and $\nabla f(\mathbf{w})$ is the vector containing all partial derivatives wrt to the components of $\mathbf{w}$:

$$\nabla f(\mathbf{w}) \doteq \left(\frac{\partial f(\mathbf{w})}{\partial w_1}, \frac{\partial f(\mathbf{w})}{\partial w_2}, \ldots, \frac{\partial f(\mathbf{w})}{\partial w_d}\right).$$

Function Approximation

Linear functions allow us to overcome the curse of dimensionality issue, but:

1. Their representational power is still limited
2. Require a careful feature engineering stage

$\Rightarrow$ Therefore non-linear functions such as neural networks are preferred!

- They follow the same principles of linear functions
- Are powerful feature extractors
- Benefit from being universal function approximators

Are **extremely hard and slow** to train!

Function Approximation

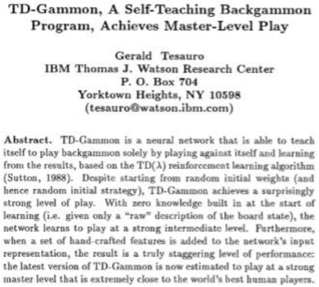
Before AlphaGo and AlphaZero there was TD-Gammon:

1. The first computer program of mastering a boardgame
2. The first successful combination of Reinforcement Learning and neural networks
3. The first practical application of TD-Learning

$\Rightarrow$ Until $\approx$ 15 years ago TD-Gammon was arguably the most (and only) successful application combining neural networks and Reinforcement Learning

Function Approximation

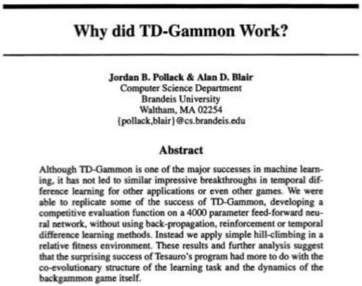
Some Reinforcement Learning history ...



TD-Gammon, A Self-Teaching Backgammon Program, Achieves Master-Level Play

Gerald Tesauro
IBM Thomas J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598
(tesauro@watson.ibm.com)

Abstract. TD-Gammon is a neural network that is able to teach itself to play backgammon solely by playing against itself and learning from the results, based on the TD(λ) reinforcement learning algorithm (Sutton, 1988). Despite starting from random initial weights (and hence random initial strategy), TD-Gammon achieves a surprisingly strong level of play. With zero knowledge built in at the start of learning (i.e. given only a "raw" description of the board state), the network learns to play at a strong intermediate level. Furthermore, when a set of hand-crafted features is added to the network's input representation, the result is a truly staggering level of performance: the latest version of TD-Gammon is now estimated to play at a strong master level that is extremely close to the world's best human players.



Why did TD-Gammon Work?

Jordan B. Pollack & Alan D. Blair
Computer Science Department
Brandeis University
Waltham, MA 02254
{pollack,blair}@cs.brandeis.edu

Abstract

Although TD-Gammon is one of the major successes in machine learning, it has not led to similar impressive breakthroughs in temporal difference learning for other applications or even other games. We were able to replicate some of the success of TD-Gammon, developing a competitive evaluation function on a 4000 parameter feed-forward neural network, without using back-propagation, reinforcement or temporal difference learning methods. Instead we apply simple hill-climbing in a relative fitness environment. These results and further analysis suggest that the surprising success of Tesauro's program had more to do with the co-evolutionary structure of the learning task and the dynamics of the backgammon game itself.

Function Approximation

In 2013 things started to change ⇒ Deep-Q Networks (DQN) were introduced!

1. The community started to focus on using Convolutional Neural Networks
2. Powerful networks able of serving as function approximators as well as feature extractors
3. The first step was to adapt the Q-Learning algorithm

⇒ Let's see how DQN intuitively works!

Function Approximation

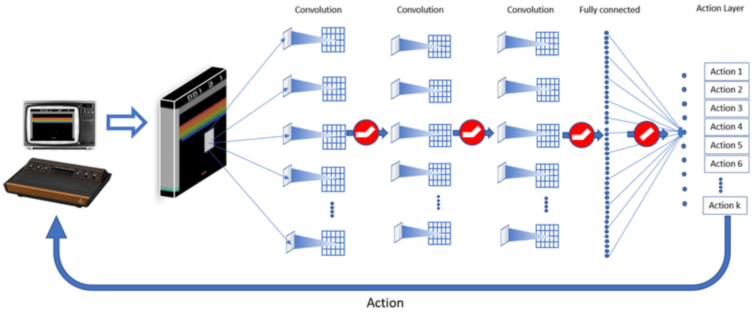


Figure: Image courtesy of Patel et al. (2019)

Function Approximation

How do we train such a system?

$$Q(s, a; \theta) \approx Q^*(s, a)$$

Function Approximation

How do we train such a system?

$$Q(s, a; \theta) \approx Q^*(s, a)$$

Function Approximation

How do we train such a system?

$$Q(s, a; \theta) \approx Q^*(s, a)$$

$\Rightarrow$ We reshape the Q-Learning algorithm into an objective function that can be used for learning θ

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

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

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

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

Function Approximation

A closer look at DQN's objective function:

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

1. Uses Q-Learning's TD-target:
   $y_t^{DQN} = r_t + \gamma \max_{a\in\mathcal{A}} Q(s_{t+1}, a; \theta^-)$.
2. Requires an additional component called Experience Replay: D

$\Rightarrow$ We are reducing the reinforcement learning problem to a supervised learning problem ...

Deep Reinforcement Learning

$\Rightarrow$ The idea of Experience Replay is to collect RL trajectories $\tau\langle s_t, a_t, r_t, s_{t+1}\rangle$ and then use them for minimizing $\mathcal{L}$

$$D = \begin{pmatrix} s_t & a_t & r_t & s_{t+1} \\ s_t & a_t & r_t & \vdots \\ s_t & a_t & r_t & \vdots \\ s_t & a_t & \vdots & \vdots \\ s_t & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

$\Rightarrow$ We are constructing a dataset of experiences ...

Deep Reinforcement Learning

At training time we uniformly sample $\sim U(D)$ from the buffer and construct a mini-batch of samples for learning

$$D = \begin{pmatrix} s_t & a_t & r_t & s_{t+1} \\ s_t & a_t & r_t & \vdots \\ s_t & a_t & r_t & \vdots \\ s_t & a_t & \vdots & \vdots \\ s_t & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

Deep Reinforcement Learning

At training time we uniformly sample from the buffer and construct a mini-batch of trajectories for learning

$$D = \begin{pmatrix} s_t & a_t & r_t & s_{t+1} \\ s_t & a_t & r_t & \vdots \\ s_t & a_t & r_t & \vdots \\ s_t & a_t & \vdots & \vdots \\ s_t & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

Deep Reinforcement Learning

At training time we uniformly sample from the buffer and construct a mini-batch of trajectories for learning

$$D = \begin{pmatrix} s_t & a_t & r_t & s_{t+1} \\ s_t & a_t & r_t & \vdots \\ s_t & a_t & r_t & \vdots \\ s_t & a_t & \vdots & \vdots \\ s_t & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

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

Deep Reinforcement Learning

$\Rightarrow$ The Experience Replay memory buffer plays a crucial role in DRL: without it, it would be impossible to train an agent

1. It makes agents more sample efficient ✓
2. Helps generalization ✓
3. Goes against the principle of online learning ✗
4. Reinforcement learning $\rightarrow$ Supervised learning ✗

Is only one among the many "tricks" that are necessary if we want to successfully combine Reinforcement Learning algorithms with deep neural networks.

Deep Reinforcement Learning

The DQN algorithm is known to suffer from the overestimation bias of the Q-function:

One can rewrite DQN's TD-target as:

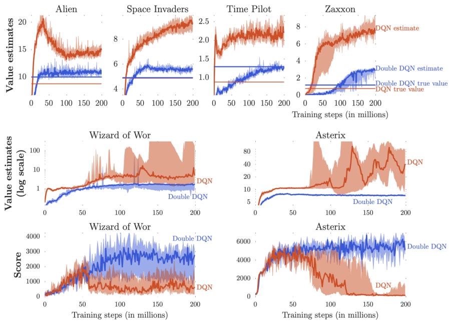
$$y_t^{DQN} = r_t + \gamma \, Q(s_{t+1}, \arg\max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta); \theta^-).$$

As a result, DQN tends to approximate the expected maximum

value of a state, instead of its maximum expected value

Deep Reinforcement Learning

- Deep Double Q Learning (DDQN) untangles the action selection process from its evaluation by taking advantage of the target network θ^{-}
- It does so by symmetrically updating the two sets of weights (θ and θ^{-}) by regularly switching their roles throughout learning



Deep Reinforcement Learning

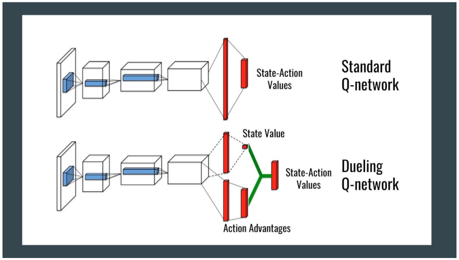
An extension of DDQN which builds on top of some ideas stemming from multi-task learning is the Dueling Architecture which considers the advantage function:

$$A^{\pi}(s, a) = Q^{\pi}(s, a) - V^{\pi}(s),$$

which gets estimated alongside $V^{\pi}(s)$ and $Q^{\pi}(s, a)$ with a network consisting of three separate streams:

Deep Reinforcement Learning

To successfully estimate state values, advantages, and state-action values, the network requires a specific architecture consisting of three separate streams



Formally ...

$$Q(s, a; \theta^{(1)}, \theta^{(2)}, \theta^{(3)}) = V\big(s; \theta^{(1)}, \theta^{(3)}\big) + \\ \big(A(s, a; \theta^{(1)}, \theta^{(2)}) - \max_{a_{t+1} \in \mathcal{A}} A(s, a_{t+1}; \theta^{(1)}, \theta^{(2)})\big).$$

Deep Reinforcement Learning

The idea of considering the state-value function $V^{\pi}(s)$ next to the state-action value function $Q^{\pi}(s, a)$ plays a crucial role in DQV-Learning and DQV-Max Learning

$\Rightarrow$ DQV-Learning minimizes the following objective function when learning $V^{\pi}(s)$

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

while the following loss is minimized for learning $Q^{\pi}(s, a)$

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

Deep Reinforcement Learning

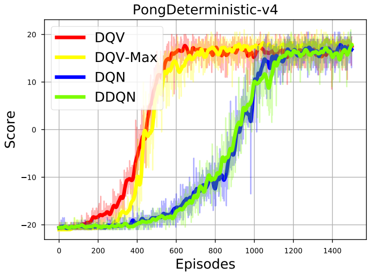
$\Rightarrow$ DQV-Max instead, minimizes the following objective function when learning $V^{\pi}(s)$

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

The way the state-value function is learned is the same as in DQV

Both algorithms converge significantly faster than DQN and DDQN

Deep Reinforcement Learning



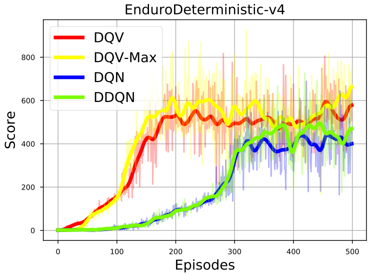
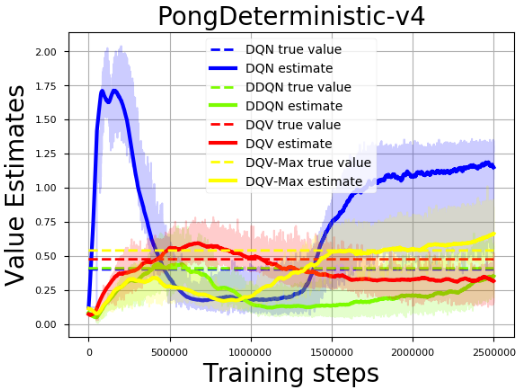


Figure: Learning curves obtained during training on three different `Atari` games by DQV and DQV-Max, and DQN and DDQN. We can observe that on these games both DQV and DQV-Max converge significantly faster than DQN and DDQN and that they obtain higher cumulative rewards on the `Enduro` environment. The shaded areas correspond to $\pm$ 1 standard deviation obtained over 5 different simulation rounds.

Deep Reinforcement Learning

DQV-Learning and DQV-Max Learning also suffer less from the overestimation bias of the Q function:



Deep Reinforcement Learning

A strong limitation of the Experience Replay memory buffer D is that it considers all trajectories within the buffer as equally important:

- However there might be states that are more informative than others
- Particularly problematic when dealing with sparse reward environments
- Due to its uniform sampling nature it is very inefficient

$\Rightarrow$ Prioritized Experience Replay solves this by sampling a trajectory τ based on:

$$P(\tau) = \frac{p_\tau^\alpha}{\sum_k p_k^\alpha}$$

Deep Reinforcement Learning

The goal of DQN, DDQN, Dueling-Networks, DQV and DQV-Max has always been that of learning $Q(s, a)$:

$\Rightarrow$ First we learn a value function, and then we derive the policy π

$\Rightarrow$ We have never seen how to learn π directly!

Deep Reinforcement Learning

Methods that approximate a value function are harmed by the Deadly Triad of Deep Reinforcement Learning, a combination of elements which can prevent algorithms from learning.

The Deadly Triad components are:

- Function Approximators
- Bootstrapping (e.g. TD-Learning)
- Off-Policy Learning

$\Rightarrow$ Understanding which component to remove is a very active area of research

Deep Reinforcement Learning

There is a family of techniques which tries to learn $\pi(a|s;\theta)$ directly:

Policy Gradient Methods

$\Rightarrow$ They learn a parametrized policy that learns how to select actions without having to consult a value function:

$$\pi(a|s;\theta) = \Pr\{a_t = a, s_t = s\,;\theta_t = \theta\}$$

The parameters θ usually correspond to the weights of a neural network

Deep Reinforcement Learning

Training policy gradient methods significantly differs from training a Deep-Q Network:

$\Rightarrow$ Deep-Q Networks aim at minimizing the TD-error:

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

$\Rightarrow$ Policy Gradient methods instead seek to maximize some scalar performance measure $J(\theta)$ and update the parameters via gradient ascent!

$$\theta_{t+1} = \theta + \alpha\widehat{\nabla J(\theta_t)}$$

Deep Reinforcement Learning

It is also possible to learn $\pi(a|s;\theta)$ in combination with a value function!

$\Rightarrow$ These algorithms come with the name of **Actor-Critic** methods:

- It can be hard to learn a policy π directly
- We would like to tell our agent learning who is learning π how good its policy is
- To do so we can use the state-value function $V^{\pi}(s)$

Deep Reinforcement Learning

How do Actor-Critic methods intuitively work?

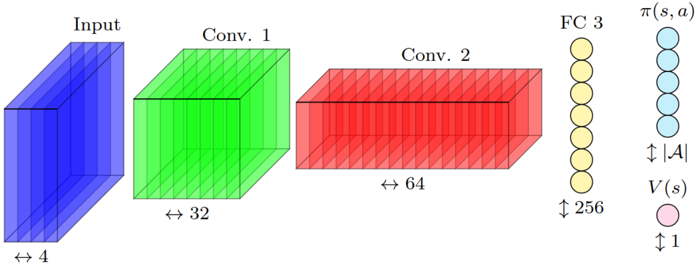


Figure: Image courtesy of Van de Wolfshaar (2017)

Deep Reinforcement Learning

How do we train this network?

$$\begin{aligned}\theta_{t+1} &= \theta + \alpha\Big(r_t + \gamma V(s_{t+1};\phi) - V(s_t;\phi)\Big)\frac{\nabla\pi(a_t|s_t;\theta_t)}{\pi(a_t|s_t;\theta_t)}\\ &= \theta_t + \alpha\delta_t\frac{\nabla\pi(a_t|s_t;\theta_t)}{\pi(a_t|s_t;\theta_t)}\end{aligned}$$

$\Rightarrow$ Note that a value function (ϕ) helps learning the policy parameters θ but is not required for action selection purposes.

Deep Reinforcement Learning

$\Rightarrow$ Nowadays actor-critic algorithms have become very popular in Deep Reinforcement Learning:

1. They are theoretically motivated thanks to the *policy gradient theorem*
2. The critic can technically learn any value function
3. Can be massively parallelized (see A3C algorithm)

$\Rightarrow$ However, compared to action-value based methods, actor-critic algorithms are less well understood (maybe because learning a policy π is still more complex than learning a value function?)

Deep Reinforcement Learning

$\Rightarrow$ Actor-Critic algorithms, just like action value based methods are also model-free Reinforcement Learning algorithms. As a result we have never even attempted learning the transition function $\mathcal{P}$ of the Markov Decision Process $\mathcal{M}$.

- Recall that the $\mathcal{P}$ and $\Re$ components of $\mathcal{M}$ are usually called the model of the environment
- If they are known we can use Dynamic Programming algorithms like value iteration :)
- However, in the typical Reinforcement Learning scenario this is never the case :(

Deep Reinforcement Learning

What to do?

- In Model-Based Reinforcement Learning the goal is to learn the model of the environment through experience!
- The idea is to learn a function that comes in the following form: $f(s_t, a_t) = s_{t+1}$
- If learned f would give us $p(s_{t+1}|s_t, a_t)$
- We can do somethinf similar for learning $p(r_t|a_t, s_t)$

$\Rightarrow$ The task of learning a model of the environment corresponds to a supervised learning problem!

Deep Reinforcement Learning

$\Rightarrow$ The overall learning strategy is very simple:

1. We start with a random policy $\pi(a_t|s_t)$
2. This policy results in a dataset of trajectories $\mathcal{D} = \{(s, a, s')_i\}$
3. We learn the dynamics of the model by minimizing $\sum_i ||f(s_i, a_i) - s'_i||$
4. Plan through the model and go back to step 2.

Deep Reinforcement Learning

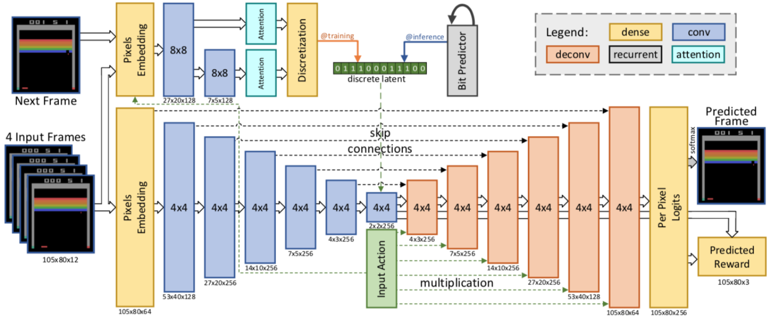


Figure: Image courtesy of Kaiser et al. (2020)

Deep Reinforcement Learning

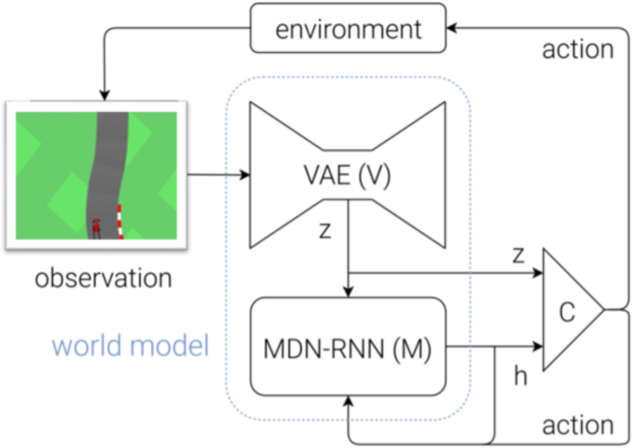


Figure: Image courtesy of Ha et al. (2019)

Deep Reinforcement Learning

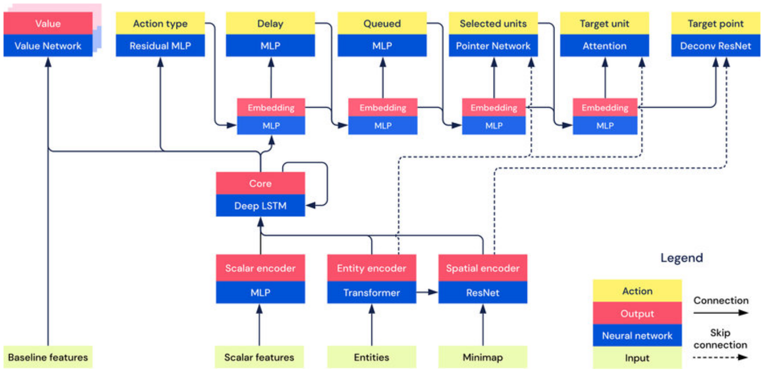


Figure: Image courtesy of Vinyals et al. (2019)

Deep Reinforcement Learning

Beyond games ...

Figure: Image courtesy of Bellemare et al. (2020)

Deep Reinforcement Learning

Beyond games ...

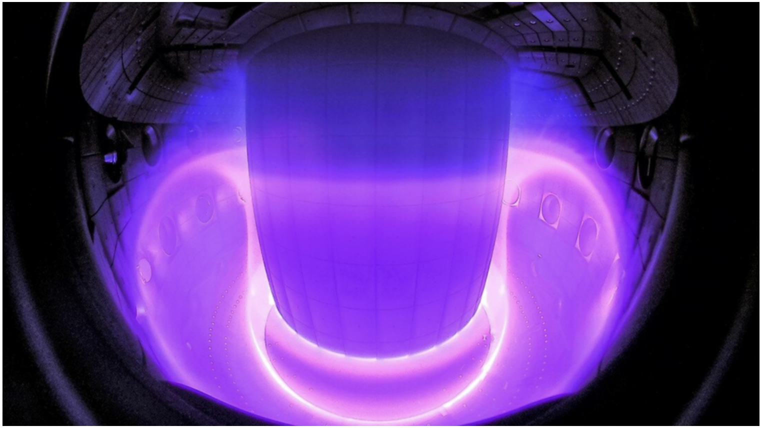


Figure: Image courtesy of Degrave et al. (2022)

Final References

- Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." nature 518.7540 (2015): 529-533.
- Van Hasselt, Hado, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning." Proceedings of the AAAI conference on artificial intelligence. Vol. 30. No. 1. 2016.
- Wang, Ziyu, et al. "Dueling network architectures for deep reinforcement learning." International conference on machine learning. PMLR, 2016.
- Hessel, Matteo, et al. "Rainbow: Combining improvements in deep reinforcement learning." Thirty-second AAAI conference on artificial intelligence. 2018.

Final References

- Van Hasselt, Hado, et al. "Deep reinforcement learning and the deadly triad." arXiv preprint arXiv:1812.02648 (2018).
- Sabatelli, Matthia, et al. "Deep quality-value (DQV) learning." arXiv preprint arXiv:1810.00368 (2018).
- Horgan, Dan, et al. "Distributed prioritized experience replay." arXiv preprint arXiv:1803.00933 (2018).
- Sabatelli, Matthia, et al. "The deep quality-value family of deep reinforcement learning algorithms." 2020 International Joint Conference on Neural Networks (IJCNN). IEEE, 2020.
- Mnih, Volodymyr, et al. "Asynchronous methods for deep reinforcement learning." International conference on machine learning. PMLR, 2016.