Deep Reinforcement Learning Overview of the State of the Art

Submitted: 5th October 2018; accepted: 16th November 2018

Youssef Fenjiro, Houda Benbrahim

DOI: 10.14313/JAMRIS_3-2018/15

Abstract:
Artificial intelligence has made big steps forward with reinforcement learning (RL) in the last century, and with the advent of deep learning (DL) in the 90s, especially, the breakthrough of convolutional networks in computer vision field. The adoption of DL neural networks in RL, in the first decade of the 21 century, led to an end-to-end framework allowing a great advance in human-level agents and autonomous systems, called deep reinforcement learning (DRL). In this paper, we will go through the development Timeline of RL and DL technologies, describing the main improvements made in both fields. Then, we will dive into DRL and have an overview of the state-of-the-art of this new and promising field, by browsing a set of algorithms (Value optimization, Policy optimization and Actor-Critic), then, giving an outline of current challenges and real-world applications, along with the hardware and frameworks used. In the end, we will discuss some potential research directions in the field of deep RL, for which we have great expectations that will lead to a real human level of intelligence.

Keywords: *reinforcement learning, deep learning, convolutional network, recurrent network, deep reinforcement learning*

1. Introduction

Reinforcement learning [1], [2] is an AI sub-domain allowing agent to fulfill a given goal while maximizing a numerical reward signal. It was developed within three main threads. The first is the concept of learning by trial and error, discovered during researches undertaken in psychology and neuroscience of animal learning. The second concept is the problem of optimal control developed in the 1950s using a discrete stochastic version of the environment known as Markovian decision processes (MDP) and adopting a concept of a dynamical system's state and optimal return function (Reward) and defining the "Bellman equation" to optimize the agent behavior over the time (Dynamic programming). The last concept concerns the temporal-difference methods, which become the mainstream, and was boosted by the actor-critic architecture. This topic is detailed in the first section.

Deep Learning (DL) [3] is a machine learning sub-domain, based on the concept of artificial neural networks that imitates human brain while processing data and creating patterns for use in decision-making. DL enables automatic features engineering and end-to-end learning through gradient descent and back-propagation. There are many types of DL nets, which usage depend on their application and the nature of the problem being treated. For time sequences like speech recognition, natural language processing we use recurrent neural network. For extracting visual features, like image classification and object detection, we use convolutional neural network. For data pattern recognition like classification and segmentation, we use feed-forward networks, and for some complex tasks like video processing, object tracking and image captioning, we use a combination of those nets. This topic is detailed in the second section.

The link between RL and DL technologies was made, while AI researchers were seeking to implement a single agent that can think and act autonomously in the real world, and get rid of any hand-engineered features. In fact, in 2015, Deepmind succeed to combine RL, which is a decision-making framework and DL [4], which is a representation learning framework allowing visual features extraction, to create the first end-to-end artificial agent that achieves human-level performance in several and diverse domains. This new technology named deep reinforcement learning is used now, not only to play ATARI games, but also to design the next generation of intelligent self-driving cars like Google with Waymo, Uber, and Tesla.

In summary, this paper will give an outline of RL and DL technologies (in sections II and III respectively), which are the basis of the deep RL. Section IV will focus on dissecting the different approaches and improvements that had a significant impact on building a human-level autonomous agents, by giving (a) an overview of state-of-the-art deep RL algorithms and achievements in recent years and (b) an outline of the current challenges of DRL and its applications in industry, and (c) an introduction to the latest toolkits and framework libraries that can be used to develop deep RL approaches.

Finally, we open a discussion related to deep RL and then inherently raise different directions for future studies in the conclusion.

2. Preliminary: Reinforcement Learning

In this section, we begin with an introduction to the fundamental concepts of reinforcement learning [1] like Markov decision process, Bellman equation,

and exploration vs exploitation dilemma. Then we will review the main algorithms and methods developed that represent the key breakthroughs of contemporary RL, allowing autonomous human-level agents to reach the actual state-of-the-art DRL.

2.1. Reinforcement Learning and Markov Decision Process

Reinforcement Learning is an AI domain inspired by behaviorist psychology, it is based on a mechanism that learns through trial and error by interacting with a stochastic environment. It is built up on the concept of Markov Decision Process MDP (see Fig. 1), a sequential decision-making problem based, defined by a 5-tuple: A set of states and actions (S,A), reward model R, state transition probability matrix P (from all states s to all their successor s') and discounted factor γ ∈ [0,1], which allows to give more importance to recent rewards compared to future rewards. An environment is said to be MDP when the state S contains all information the agent needs to act optimally.

Fig. 1. Markov decision process

The state transitions of an MDP is memoryless, so, we say that it satisfies the Markov property (1). RL agents behave under this assumption, so the effects of an action taken in a state **depend only on that state** and **not on the prior history:**

$$P(S_{t+1}=s' \mid S_t=s_t, A_t=a_t, S_{t-1}=s_{t-1}, A_{t-1}=a_{t-1}, \ldots) = P(S_{t+1}=s' \mid S_t=s_t, A_t=a_t) \quad (1)$$

If MDP is episodic, the state is reset after each episode of length T. **Reward R_t** defines what are the good and bad events for the agent, and **Cumulative reward G_t (2)** is the discounted sum of reward accumulated throughout an episode of T steps:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{T} \gamma^k R_{t+k+1} = R_{t+1} + \gamma G_{t+1} \quad (2)$$

π is the policy function that maps each possible state **s** of the agent to its selective action **a**, π: S → p(A = a|S). The agent try to learn an optimal policy **π*** in order to take the best actions that maximize the cumulative reward $\mathbf{G_t}$ [reinforcement feedback from the environment].

2.2. Reinforcement Learning and Bellman Equations

To find the optimal policy **π*** that achieves the maximum cumulative reward, RL algorithms involve estimating the following value functions:

- State-Value function **V(s)** (3) estimate how good (future rewards) it is, to be in a state **s**. **V(s)** under policy **π** is denoted $\mathbf{V_\pi(s)}$:

$$V_\pi(s) = E_\pi(G_t \mid S_t = s) = E_\pi\left(\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S = s\right), \quad \text{for all } \boldsymbol{s} \in \boldsymbol{S} \quad (3)$$

- Action-value function $Q(s,a)$ (4) how good (future rewards) it is to perform an action a in a state s. $Q(s,a)$ under policy π is denoted **:**

$$Q_\pi(s,a) = E_\pi(G_t \mid S_t = s, A_t = a) = E_\pi\left(\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right), \quad \text{for all } s \in S \text{ and } a \in A \quad (4)$$

From the two relation above, we infer the **Bellman equations** that break RL problems into sub-problems by expressing an iterative relationship between the value of a state s_t and the values of its successor states s_{t+1}, with r_t the expected reward from s_t to s_{t+1} by following a_t, we have the equations (5):

$$V_\pi(s_t) = \sum_{a_t} \pi(a_t \mid s_t) \sum_{s_{t+1}, r_t} P(s_{t+1}, r_t \mid s_t, a_t)[r_t + \gamma . V_\pi(s_{t+1})]$$
$$Q_\pi(s_t, a_t) = \sum_{a_t} \pi(a_t \mid s_t) \sum_{s_{t+1}, r_t} P(s_{t+1}, r_t \mid s_t, a_t)[r_t + \gamma . Q_\pi(s_{t+1}, a_{t+1})] \quad (5)$$

We then infer the **Bellman optimality equations** (6) (7) for V and Q under the optimal policy as below:

$$V^*(s_t) = \max_\pi [V_\pi(s_t)] = \max_{a_t} \sum_{s_{t+1}, r_t} P(s_{t+1}, r_t \mid s_t, a_t)[r_t + \gamma, V_\pi(s_t)] \quad (6)$$

$$Q^*(s_t, a_t) = \max_\pi [Q_\pi(s_t, a_t)] = \sum_{s', r} P(s_{t+1}, r_t \mid s_t, a_t)[r_t + \gamma, \max_{a_{t+1}} Q_\pi(s_{t+1}, a_{t+1})] \quad (7)$$

Optimal Value function = immediate reward r + discounted value of successors state $\gamma V(S_{t+1})$.

The optimal policy can be found using two different modes depending on:

- **On-policy** agent learns policy, and action is performed by the current policy instead of the greedy policy.
- **Off-policy** agent learns the optimal values (V,Q), and action is performed by the greedy policy (**max** operator in Bellman equation → Optimal policy).

In RL, there are two main types of algorithms:

- **Model-based** algorithms that use a model to predict the unobserved portion of the environment like Dynamic Programming, but they suffer from Bellman's curse of dimensionality problem (use full-width backups), since knowing all the elements of the MDP is a tough task, especially when we have infinite states or almost.
- **Model-free** algorithms that skip learning a model and directly learn what action to do and when, by estimating the value function of a certain policy without a concrete model. The most known

methods are Monte-Carlo, Temporal difference learning and its variants Q-learning and SARSA.

In this paper, we will focus on model-free RL.

2.3. The Exploration/Exploitation Dilemma

In real life, the best long-term strategy may involve short-term sacrifices to gather enough information so as to make the best overall decisions. For RL problems, to avoid being stuck in a local maximum, we have to balance between the two concurrent approaches Exploration and Exploitation (see Fig. 2):

- **Exploit** (using deterministic search): in this case, the search is deterministic and the RL agent chooses actions that he has already attempted in the past (from the history of trials) and which maximized the cumulative reward and proved to be the most efficient.
- **Explore** (using non-deterministic search): to gather more information and discover such actions, it has to try actions (weighted by a probability of correctness) that it has not selected before, and so allow the exploration of the other possibilities, in order to make better action selections in the future.

Fig. 2. Exploitation vs Exploration

The most known approaches to exploration use the following action-selection strategies [5]:

- **Greedy Approach:** Agent is exploiting its current knowledge to choose at any time the action which he expects to provide the greatest reward.
- **ε-greedy Approach:** forces the non-greedy actions to be tried (exploration) with no preference for nearly greedy ones or particularly uncertain (chooses equally among all actions). ε is the probability of exploration typically **5** or **10%** (see Fig. 3).

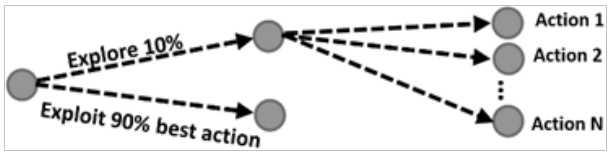


Fig. 3. ε-greedy approach

- **Softmax Approach:** All the actions are ranked and weighted according to their values estimates, but the selection probability of Greedy actions is the highest. A random action is selected, by taking into account the weight of each action. In practice, we use an additional temperature parameter (**τ**) applied to Softmax, to lower the low probabilities and higher the high probabilities.
- **Bayesian Approach:** we add a probability distribution to the neural network weights by repeatedly sampling from a network with dropout [6]; thus, the distribution variance provides an estimate of the uncertainty of each action.

2.4. Monte-Carlo Learning

Monte Carlo method [1] relies on repeated random sampling to obtain numerical results. By the law of large numbers, the expected value of a random variable can be approximated, by taking the sample mean of independent samples of the variable.

MC methods are used in RL to solve episodic problems by averaging sample returns and learning directly from complete episodes of experience without bootstrapping. MC methods are insensitive to initial value since they learn only from complete sequences, the return is known only at the end of the episode and not before.

MC is used for prediction by learning the state-value function Vπ(s) following a given policy π, and for control by estimating the policy using Generalized Policy Iteration GPI and the action-value function Q. The MC control algorithm starts with an arbitrary policy π and iterates between the two steps until converging toward the optimal policy $\boldsymbol{\pi^*}$:

- **Policy evaluation:** use the current policy π to estimate $\boldsymbol{Q^{\pi}}$ or $\boldsymbol{V^{\pi}}$.
- **Policy improvement:** making a better policy π by deterministically choosing actions with maximal action-value: $\boldsymbol{\pi(s)} = \mathbf{arg}\ \boldsymbol{max_a q(s,a)}$.

2.5. Temporal-Difference Learning

Temporal-Difference Learning [1], [6] is model-free methods that act by deriving its information from experiences without having complete knowledge of the environment. TD combines Monte Carlo methods, by learning directly from raw experience, with dynamic programming methods, by updating value function estimates through bootstrapping from its current estimate.

TD updates values using recent trends so as to capture the effect of a certain state. It learns online after every step from **incomplete episodes of experience** by **sampling** the environment according to a given policy and approximating its current estimate based on previously learned estimates. The general rule (8) can be summarized as follow: $\mathbf{V_{New} \leftarrow V_{Old} + StepSize^* [Target - V_{Old}]}$

$$V(S_t) \leftarrow V(S_t) + \underbrace{\alpha(\underbrace{R_{t+1} + \gamma V(S_{t+1})}_{\text{TD target}} - V(S_t)}_{\text{TD Error}} \qquad (8)$$

In TD learning, instead of computing the update every step, we can postpone it after N steps. The N-step return, in this case, is calculated as follows:

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n V(R_{t+n}) \qquad (9)$$

N-Step TD formula becomes:

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t^{(n)} - V(S_t)] \qquad (10)$$

In TD learning, we also integrate a neurologic phenomenon called **eligibility trace (ET)**, This RL mechanism uses a short-term memory that stores the steps state-action history called **traces**. Those traces mark the state as eligible for learning, reinforcing the event that contributed to getting to the reward. It decays gradually over time if the given state is not enough visited. So, ET extends what the agent learned at t+1 also to previous states, by tracking where he has been (prior states) and back-upping of reward for a longer period, so to reinforce the most visited states and accelerate learning.

Eligibility traces [1] **implement a** memory trace that is usually an exponential function, with a **decay parameter**. The three most known Eligibility traces implementations are:

- **Accumulating traces:** accumulate each time the state is visited, then fades away gradually when the state is not visited.

$$e_{t(s)} = \begin{cases} \gamma\lambda e_{t-1(s)} \text{ if } s \neq s_t \\ \gamma\lambda e_{t-1(s)} + 1 \text{ if } s = s_t \end{cases} \quad (11)$$

- **Replacing traces:** each time a state is visited, the trace is reset to 1, regardless to present or prior trace

$$e_{t(s)} = \begin{cases} \gamma\lambda e_{t-1(s)} \text{ if } s \neq s_t \\ 1 \text{ if } s = s_t \end{cases} \quad (12)$$

- **Dutch traces:** intermediate between accumulating and replacing traces, depending on the step-size parameter $\boldsymbol{\alpha}$

$$e_{t(s)} = \begin{cases} \gamma\lambda e_{t-1(s)} \text{ if } s \neq s_t \\ (1-\alpha)\gamma\lambda e_{t-1(s)} + 1 \text{ if } s = s_t \end{cases} \quad (13)$$

$\mathbf{e_t(s)}$ is the **eligibility trace** function for state **s** and time **t**, On each step, **it** decays by $\boldsymbol{\gamma\lambda}$ for all non-visited states.

2.6. Q-learning

Q-learning [7] is an **Off-Policy** algorithm for **TD-Learning control** (MDP environment) used in reinforcement learning. The learned action-value function Q approximates directly the optimal action-value function Q*, regardless of the policy being followed. One-step Q-learning is defined by (9):

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\{R_{t+1} + \gamma max_a[Q(S_{t+1}, a)] - Q(S_t, A_t)\} \quad (14)$$

Q-learning combined with eligibility trace become Q($\boldsymbol{\lambda}$):

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha e_t(s,a)\{R_{t+1} + \gamma \max_a[Q(S_{t+1}, a)] - Q(S_t, A_t)\} \quad (15)$$

With three known implementations of eligibility trace [1] for Q($\boldsymbol{\lambda}$): Watkins Q(λ) (10) [8], Peng Q(λ) which doesn't distinct between exploratory and greedy actions and Naïve Q(λ) which is similar to Watkins's method, except that the traces are not set to zero on exploratory actions.

$$e_t(s,a) = \begin{cases} \gamma\lambda e_{t-1}(s,a), \text{if } \forall s \in S, s \neq S_t, a \neq a_t \\ \gamma\lambda e_{t-1}(s,a) + 1, \text{if } (s,a) = (S_t, a_t) \text{ and} \\ Q_{t-1}(s_t, a_t) = max_a[Q_{t-1}(s_t, a)] \\ 0, \text{if } Q_{t-1}(s_t, a_t) \neq max_a[Q_{t-1}(s_t, a)] \end{cases} \quad (16)$$

2.7. SARSA

SARSA (**S**tate–**A**ction–**R**eward–**S**tate–**A**ction)[9] is an on-Policy algorithm for TD-Learning control (MDP environment) used in the reinforcement learning, it learns an action-value function of [state, action] pairs that depends on the quintuple (s_t, a_t, r_t, s_t+1, a_t+1).

What makes the difference with Q-Learning, is that with **SARSA,** the maximum reward for the next state is not necessarily used for updating the Q-values; instead, a new action (& reward), is selected using the same policy that determined the original action:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \quad (17)$$

SARSA combined with eligibility trace become SARSA(λ):

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha e_t(s,a)[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \quad (18)$$

In SARSA, the **policy π** is updated at each visit choosing the action with the highest state-action value **$\mathbf{argmax_a Q(s_t, a_t)}$** making the policy greedy.

2.8. Actor-Critic

Actor-Critic (AC) algorithms were inspired by neuroscience and animal learning [10], it's a hybrid control methods that combine the policy gradient method and the value function method together. The Actor-Critic algorithm (see Fig. 5) introduces a **Critic** that judges the actions of the actor. The **Actor** is the source of high variance and the critic provides low-variance feedback on the quality of the performance, which balanced the equation. Adding the Critic component reduces variance and higher the likelihood of convergence of the policy gradient methods.

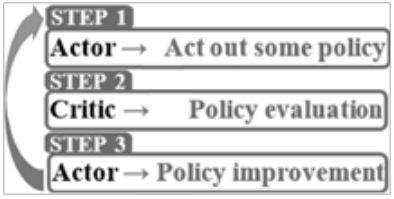


Fig. 5. ***Actor-Critic Algorithm steps***

AC **methods** are considered part of TD methods since the critic is an implementation of the TD(0) algorithm and it is updated following the same rule:

- **Actor**: policy function, produces the action for a given input "current state" **s** of the environment
- **Critic**: value function, criticizes the actions made by the actor. Input information obtained through sensors (state estimation), and it receives rewards

2.9. RL limitations and Function approximator

In RL the value functions V(s) or Q(s,a) are represented by a state transition table (lookup table), where every state **s,** has an entry V(s) or every state-action pair (s,a) has an entry Q(s,a). When the Markov decision process is large, we will have too many states and actions to store in memory and it is too slow to learn the value of each state individually. For instance, in Computer Go, we use 106 parameters to learn about 10170 positions[11], [12].

Instead of having a lookup table with explicit values for all the (state, action) space, the idea is to use a function approximator like a neural network that will replace this lookup table. Therefore, we will estimate value functions with function approximation $\hat{V}(s,w) \approx V_{\pi}(s)$ or $\hat{q}(s,a,w) \approx q_1$ where π indicate the neural network. The gain is that it allows generalizing from seen states to unseen states and reuse reinforcement learning framework (MC, TD learning,…) to update the weights **w** [13].

With the breakthrough made in deep learning in computer vision, we won't be only using a Feed-forward neural network to approximate the value functions used in RL, but also a convolutional neural network that allows to get ride off hand-engineered visual features, and directly capture the environment visual state. Optionally, we can also use a recurrent neural network to keep in memory the relevant events during the agent life cycle, which can help to get an optimal experience.

3. Preliminary: Deep Learning

Deep learning is a branch of machine learning based on deep (> 2 hidden layers) and wide (many input/hidden neurons) neural networks, that model high-level abstractions in data, based on an architecture composed of multiple non-linear layers of neurons. Each neuron of the hidden layers performs a linear combination of its inputs and applies a non-linear function (Relu, Softmax, Sigmoid, tanh, …) to the result, which allows neurons from the next layer to separate classes with a curve (hypercurve/hyperplane) and no more with a simple line (see Fig. 6), thus, hidden layers learn hierarchical features. The deeper the layers, the more complex the learned features are [14].

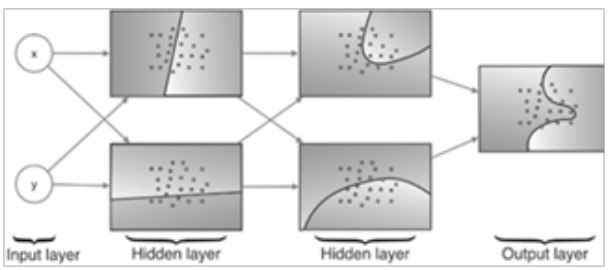


Fig. 6. neural network learns to separate classes with complex curves, thanks to the hierarchy of layers

3.1. Backpropagation and Gradient Descent

Unlike machine learning where features are crafted by hand, with deep learning, features are automatically learned to be optimal for the task. To achieve the process of learning, DL uses cost/loss function like the mean square error MSE or Cross entropy CE (19):

- **MSE Loss:**

$$L_{MSE} = \sum \tfrac{1}{2}[target(y) - prediction(\hat{y})] \qquad (19)$$

- **CE Loss:** $L_{CE} = \sum [target(y) * \log(prediction(\hat{y}))]$

We use these losses to measure how well the neural network performs to map training examples to correct output (in the classification case), and then tweak his parameters (weights and biases) using backpropagation processes based on gradient descent (GD) optimization methods [15] that finds the minimum error:

- **Batch gradient descent:** calculate gradient for the entire training dataset to update parameters
- **Stochastic gradient descent (SGD):** calculate the gradients for each training sample xi of the dataset
- **Mini-batch gradient decent:** tradeoff of the two methods, mini-batch sizes range ϵ [50, 256] (can vary).

3.2. Learning Rate

Learning rate is a hyper-parameter used by GD methods to control the adjustment rate of the network's weights with respect to the loss gradient. The learning speed is slow when the rate is low, but can diverge when the rate is too high, the most popular learning rates are:

- **Momentum** [16]: accelerates SGD convergence in the relevant direction while reducing oscillations, by adding a parameter **γ** (usually 0,9) of the updated vector of the previous step to the current update vector.

$$\mathbf{V_t = \gamma.V_{t-1} + \eta.\nabla_\theta L(\theta, x_i)} \text{ and } \mathbf{\theta = \theta - V_t} \qquad (20)$$

  where θ is the vector that represents the network's parameters and L is the loss function
- **Adagrad** [17]: adapts the learning rate to the parameters, by making larger updates for infrequent parameters (small historical gradients) and smaller updates for frequent ones (bigger historical gradients).

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i}$$

$$\text{where } g_{t,i} = \sum_i \left[\nabla_{\theta_i} L(\theta_i)\right]^2 \qquad (21)$$

- **RMSprop** [18] (Root Mean Squared): is an adaptive learning rate method, an extension of Adagrad proposed by Geoffrey Hinton (γ = 0.9 , η = 0.01)

$$\theta_{t+1,i} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t ,$$
$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2 \qquad (22)$$

- **Adadelta** [19]: is an improvement of Adagrad that prevent learning rate from convergence to zero with time. It restricts the accumulated past gradients to only a recent time window of fixed size.

$$\theta_{t+1} = \theta_t - \frac{\sqrt{E\left[\Delta\theta^2\right]_t + \epsilon}}{\sqrt{E\left[g^2\right]_t + \epsilon}} g_t ,$$
$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2 ,$$
$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2 \qquad (23)$$

- **Adam** [20] (Adaptive Moment Estimation): computes adaptive learning rates for each parameter. It keeps an exponentially decaying average of past gradients $m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$, similar to momentum. It stores both exponentially decaying average of past gradients and squared gradients like **Adadelta**.

$$\theta_{t+1} = \theta_t - \frac{\sqrt{E\left[\Delta\theta^2\right]_t + \epsilon}}{\sqrt{E\left[g^2\right]_t + \epsilon}} g_t ,$$
$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2 ,$$
$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2 \qquad (24)$$

3.3. Hyper-Parameters Optimization

For DL, Hyper-parameters include the number of layers, the size of each layer, nonlinearity functions, weights initialization, decay term, learning rate, loss function, and input batch size. Optimization is done by measuring performance on independent data set and choose the optimal ones that maximize this measure, Most known Optimization algorithms are Grid search, Random search [21], Bayesian optimization [22], Gradient-based optimization [23], Genetic algorithms.

3.4. Neural Networks and Overfitting

Neural networks fight overfitting by applying commonly used approaches like validation data, data augmentation or early stopping (during training phase), in addition, it uses two main methods that became widespread:

- Dropout method applied at every forward pass during the training process, by randomly dropping nodes and their connections from hidden or input layers (with the same probability), which prevent the network from becoming sensitive to the weights of nodes and make it more robust.
- Regularization using batch normalization that normalizes the inputs of each layer before applying the activation function, in order to have a mean output activation of zero and standard deviation of one.

3.5. Neural Networks Main Types

In this section, we will give a brief outline of three types of DL used in DRL: feed-forward neural network, convolutional neural network, and recurrent neural network, to introduce thereafter, in the next section, the deep reinforcement learning concepts.

3.5.1. Feed-forward Neural Network

A feedforward neural network [24] (Multilayer Perceptron) is a non-linear artificial neural network where the information moves in only one direction, it solves classification problems and is composed of three main parts: the input layer, N hidden layers, and an output layer. Each layer can contain a given number of neurons. Neurons of hidden & output layers use **non-linear activation function**, to distinguish data that is not linearly separable, for this purpose, we use mainly Relu or sigmoid functions. The learning is carried out through minimization of the loss function, using cross-entropy or mean square error functions. An appropriate decaying learning rate is used to avoid local minima issues and **backpropagation** of the error to change **connection weights** using gradient descent algorithm (most of time Stochastic gradient descent or Mini-batch gradient descent) in a way to get the best fit values of those weights that will lead to an optimal error.

3.5.2. Convolutional Neural Network

Convolutional neural network (CNN, or ConvNet) [25], [26] is a class of deep feed-forward artificial neural network that has successfully been applied to analyzing visual imagery. CNN is used in supervised learning for classification and object recognition/detection purposes, in unsupervised learning for image compression and image segmentation, and finally as visual features extractor in deep reinforcement learning.

CNN is composed of four basic components :

- **Convolutional layers:** the layer is no more fully connected like in feed-forward nets, instead, it learns **2D** square-shaped matrix of neurons called **filters** (or kernels, Eg. 9 neurons for a kernel of 3×3 pixels), that scans the whole image searching for a pattern (**localized feature**), by applying effects such as **blurring**, **sharpening**, **outlining**, **embossing, etc.**, to extract visual features. Each neuron of a kernel in the hidden layer will be connected to a small region (E.g. 3×3 pixels) of the input image (Ex. 200×200 pixels) given by the previous layer, called the ***local receptive field***. Each **kernel** leverage these ideas :
  - **2D Convolution:** Convolution is an image processing operation that is a weighted multiplication between the image matrix

representation ***i*** with the kernel's matrix (filter k of size Nk*Nk) to extract visual features from the image, by generating an output image $\boldsymbol{i}_{\mathbf{conv}}$ called feature map:

$$i_{conv}(x,y) = \sum_{x_k=0}^{N_k-1}\sum_{y_k=0}^{N_k-1} i(x-x_k, y-y_k) * k(x_k, y_k) \quad (25)$$

Two additional parameters for 2D convolution:

- **Zero Padding:** Put zeros on the image border to allow the convolutional output size to be the same as the input size image.
- **Strides**: How many pixels the kernel window will slide at each step while scanning the image.

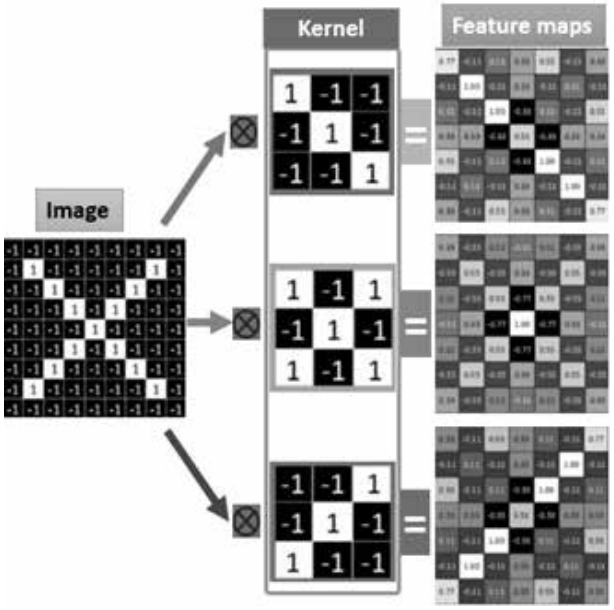


Fig. 7. Feature maps are the result of the convolution of two matrices (image and kernel)

- **Parameter sharing:** neurons of the same kernel share the same weights
- **Local connectivity**: each input neuron in kernels only receives input from a small local group of the pixels in the input image called local receptive field by Cutting connections to the other pixels. input neurons represent overlapping receptive fields that form a complete map of visual space.
- **Feature extraction:** each kernel can detect just a single kind of localized feature. So, if we want to look for 10 different patterns we must have 10 kernels in the convolutional layer, each one looking for a particular pattern on the image.
- **Hierarchical features learning:** inspired by the organization of the animal visual cortex, multi-convolutional layers network allow to learn hierarchical visual features, the deeper is the layer, the more complex is the detected feature.

- **Pooling layers:** are used immediately after convolutional layers to shrink the output image using non-linear down-sampling and add to some amount of translation invariance. For instance, each unit of pooling layer summarizes a region of N×N neurons in the previous layer (ex. 3×3). There are several implementations of pooling like Average pooling which calculates the average value of the N×N matrix and Max-pooling (26) which is the most common and takes the max value of the N×N matrix and Mixed Pooling.

$$i_{\max-pool}(x,y) = \max_{(x,y)\in region\ NxN} i(x,y) \quad (26)$$

No learning takes place on the pooling layers. With back-propagation, only convolutional layers are concerned and we do not use pooling when we want to "learn" some object specific positions like in reinforcement learning.

- **Fully connected layer:** is a normal feed-forward network layer that makes the connection between each input dimension (pixel location) and each output class, mixes signals received from feature learning layers and decides on classification based on the whole image
- **Normalization layer:** apply batch normalization [27] on input and hidden layers to rescale the input data $\hat{x} = \frac{x - E(x)}{\sqrt{Var(x)}}$, which helps to avoid vanishing/exploding gradient descent problem and to have deeper a network.

3.5.3. CNN Improvements and CapsNet

The first functional Conv net was **Lenet** [28] implemented by Yann Lecun in 2006. Then came the **AlexNet** [29] in 2012 a deeper and much wider version (5 Convolutional + 3 Maxpooling + 3 Fully-connected) of the LeNet, which integrate RELU (Rectified Linear Unit) activation function and Reduce the over-fitting, by using a Dropout layer after every FC layer. In 2014 **VGGNET** [30] adapts more layers (16 Convolutional + 5 Maxpooling + 3 Fully-connected) and lower dimension for convolution filters are 3×3 (instead of Instead of the 9×9 or 11×11 filters for AlexNet).

In the same year, Google came out with **GoogLeNet** (22 Convolutional layers) [31], which reuses 1x1 convolutions, introduced by NiN [32] to perform dimensionality reduction, and bring in the new concept of inception module (see Fig. 8), which allow CNN network to use many kernels dimensions (5×5, 3×3, 1×1)

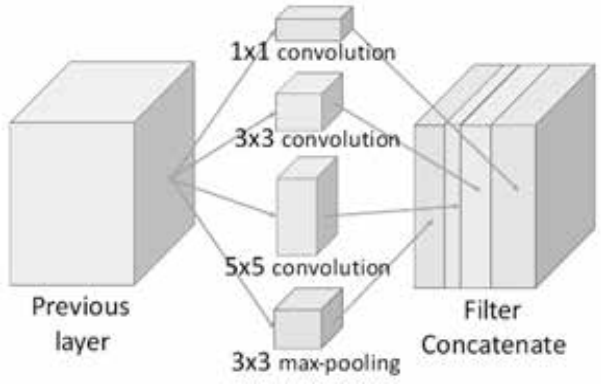


Fig. 8. Inception module

and pooling methods in the same layer and choosing itself the best filter through backpropagation process.

In 2015, **ResNets** [33] launched by Microsoft allows deeper neural networks (152 convolutional layers) by adding Identity connections to the traditional neural network, every two convolutional layers. The layers can start as the identity function and gradually transform to be more complex and more efficient.

Even if CNN has made a great breakthrough in the computer vision domain, some drawbacks remain:

- Orientation and relative spatial relationships between hierarchical features are not important, for example, the two representations below are both faces.
- Pooling layers don't have learnable parameters that learn how to pool in a dynamic way, that predict which low-level features (ex. nose, eyes, mouth) would be routed to the higher level features (ex. face).
- Training needs a large amount of data to reach an acceptable accuracy.

Capsule network [35] came as a solution to solve these problems with an architecture composed from an Encoder (1 Conv + 2 Capsule layers) and decoder (3 FC), and the use of two principles :

- **Activity vector & Equivariance:** neuron are replaced by capsules (a group of neurons) and activity vector for object detection with additional equivariant features (orientation, lighting, …). Changes in object position lead to changes in the orientation, without any change in vector length and probability.
- **Dynamic routing:** It replaces max pooling by adding an intermediate level a weight matrix $\mathbf{W_{ij}}$, that learns how to pool using dynamic routing of the capsule of layer N to the appropriate parent in the layer N+1, and encodes the relationship between the input features $\mathbf{u_i}$ to get their predicted position relative to each other. The higher level capsules combine objects parts and encode their relative positions, so an object can be accurately detected not only from the presence of the parts but also their right locations.

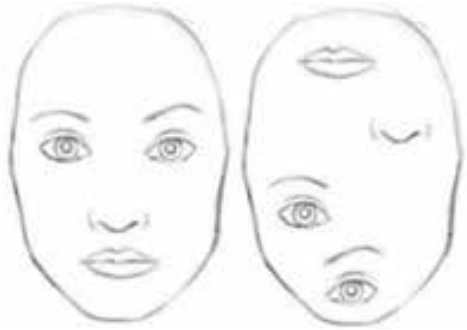


Fig. 9. The two figures are the same for CNN

3.5.4. Recurrent Neural Network

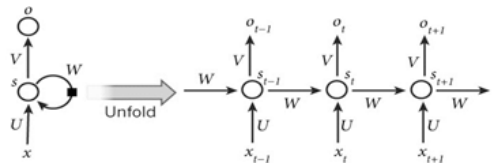
RNN is a deep network that extracts temporal features while processing sequences of inputs like text, audio or video. It's used when we need history/context to be able to provide the output based on previous inputs, like for video tracking, Image captioning, Speech-to-text, Translation, Stock forecasting, etc.

RNN neuron uses its internal memory to maintain information about the previous inputs and update the hidden states accordingly, which allows them to make predictions for every element of a sequence.

RNN maintain a state vector $\mathbf{s_t = g\,(x_t, x_{t-1}, x_{t-2}, \ldots, x_2; x_1)}$ that contains data features with the history of all previous input sequences.

RNN can be converted into a feedforward network by unfolding it over the time to many layers that share the same weights W:

$$s_t = \sigma(Ux_t + Ws_{t-1}) \quad \text{and} \quad o = g(Vs_t) \qquad (27)$$



x_t: input at time t
s_t: hidden state at time t (memory of the network)
f: is an activation function (e.g, tanh() and ReLUs)
U, V, W: network parameters (same across time)
g: activation function for the output layer (softmax)

Fig. 10. RNN Cell unfolded

RNNs can learn to use the past information when the context is small, as that gap grows, RNNs become unable to learn to connect the information, due to Vanishing and Exploding gradient problem.

LSTM (LONG SHORT-TERM MEMORY) [36], [37] is a variant of RNN, that came with a solution, by replacing simple RNN node by a complex cell composed of 4 layers, which allow to remove or add information to the cell state, judiciously regulated by three **gates** that conditionally decides what information to keep, what information to update, and what information to throw away:

- **Input Gate:** selectively update cell state values by adding information about the new input.
- **Forget gate**: forget irrelevant parts of previous states, depending on the relevance of the stored information.
- **Output Gate:** select information from the current cell state and show it out.

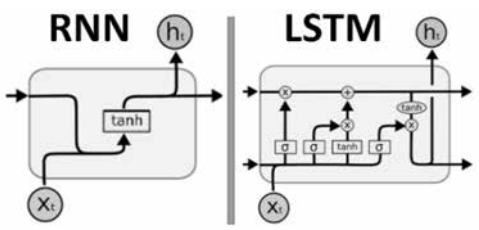


Fig. 11. RNN Cell vs LSTM Cell

3.5.5. Transfer Learning for Deep Learning

Transfer learning [38], [39] (TL) is the ability of a system to apply knowledge and skills learned in previous tasks to novel tasks in new domains. In DL, we reuse pre-trained models as a starting initialization for

new models to speed up training phase, which become a tuning phase where the new model is refined on the input-output pair data available for the new task.

The TL process tends to work if the features are general and both original and target tasks are not too far. For CNN, in case training data of the new model is similar to pre-trained model data, so all CNN Layers are fixed and only the FC layers are trained, otherwise, only lower CNN Layers that contain basic features are fixed and higher CNN layers and FC layers are trained on the new model dataset.

3.6. Deep Learning Challenges

Even if Deep Learning has made great steps, it always requires large dataset, hence long training period and big clusters of CPUs and GPUs (graphics processing units). Moreover, the learned features are often difficult to understand. In addition, we have to pay attention to overfitting, notably, when the number of parameters greatly exceeds the number of independent observations. Finally, DL is sensitive to what we call adversarial attacks, when small variations in the input data, leads to radically different outcomes, causing a lack of robustness and making them unstable.

4. Deep Reinforcement Learning: Literature Review

As seen in paragraph III, traditional reinforcement learning use lookup table to store states and actions, which is too slow, since it learns the value of each state individually, and it is memory consuming, especially when we deal with large or infinite problems, and this is due to what Richard Bellman called the curse of dimensionality. The solution is to estimate value function using differentiable function approximators, trained using reinforcement learning algorithm.

By leveraging deep learning algorithms, especially, convolutional neural networks, it became possible for RL algorithm not only act but to be totally autonomous and learn to see and act, a new technology is born called Deep Reinforcement Learning (DRL) (see DL, RL and DRL Timeline Fig. 12).

We have three main types of DRL algorithms:

- **Value optimization:** the algorithm optimizes the Value function V or Q, or the advantage function A.
- **Policy optimization:** the algorithm optimizes the policy directly function $\pi(\theta)$ representing the neural network.
- **Actor-critic** incorporates the advantages of each of the above, by learning value function with Implicit policy:
  - **Policy gradient** component **"Actor"** which calculates policy gradients
  - **Value function** component **"Critic"** that observes the performance of the actor and decides when the policy needs to be updated and which action should be preferred

In the following section, we will have an outline on the Value optimization and Actor-critic algorithms (see Fig. 13), and try to understand their mechanisms and functioning.

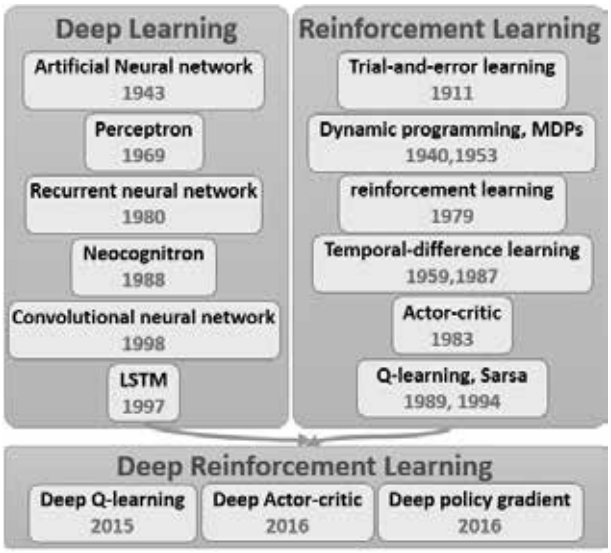


Fig. 12. DL, RL and DRL Timeline

Fig. 13. Deep Reinforcement Learning algorithms

4.1. Value Optimization Algorithms

4.1.1. Deep Q-Learning (DQN in detail)

Deep Q Learning [4] is the first application of Q-learning to deep learning, performed by Google DeepMind in 2015, it succeeded to play 2600 Atari games at expert human level. DQN is a concentrate of technologies that uses many tips and tricks:

- **Tricky Architecture network:** in the standard Q-learning algorithm, the input is composed of the state s and the action a, which will require a separate forward pass to compute Q-value $Q(a_i)$ of each action ai. Instead, we will use the state s as the only input, with as many outputs as possible actions a_i. Therefore, the network will generate a Q-value probability for each available action, immediately with a single forward pass.
- **The neural network as a function approximator:** three convolutional layers to detect visual features and to learn a hierarchical representation of the state space + two fully connected layers to estimate Q values from images, pooling layers is not used in DQN because we want CNN to be sensitive to the location of objects in the image.
- **3D convolution:** process a 2D convolution of the four frames of the input, then average them all.

Frame skipping [40] (see Fig. 14): as an initial input we have a video stream of 30 screenshots/s 210×160×3 pixels of 128 colors, which we crop, shrink and turn into greyscale to have 84×84. But, processing all the 30 image/s of the video stream is not really relevant and also needs more computation and time, so the trick is to take only 2 consecutive frames frame of each N frame and skip the others, and for these 2

frames we apply the component-wise maximum function to get 1 frame: $\mathbf{Fr_{cw}(i,j)=max(Fr1(i,j), Fr2(i,j))}$

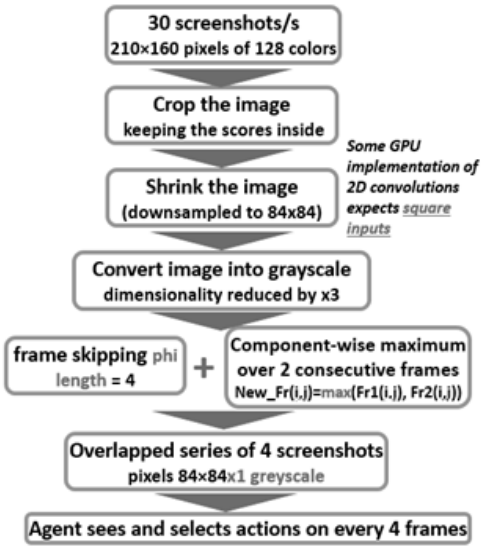


Fig. 14. image processing of the input video, before feeding the DQN

- **Phi length parameter:** to help the network to Detect the motion and catch speed information, we stack a number of frames "Phi length" of a history to produce the input of the DQN network, most of time Phi length = 4 or 5.
- **Target Network** (see Fig. 15)**:** at every training step, the DQN's values must shift due to backpropagation that changes the network's weights, but shifting constantly the set of values to adjust the network will destabilize it, which will fall into feedback loops between the target and estimated Q-values. The idea is to use a separate network to estimate the target-Q values that will be used to compute the loss for every action:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\{R_{t+1} + \gamma \max_a[Q(S_{t+1}, a)] - Q(S_t, A_t)\}$$

$$L_{MSE} = \frac{1}{2}\{R_{t+1} + \gamma \max_a[Q(S_{t+1}, a)] - Q(S_t, A_t)\}^2 \qquad (28)$$

This target network has the same architecture as the function approximator but with fixed weights, every T steps (ex. 1000), weights from the Q network are copied to the target network, which provides more stability to the DQN. An improvement of this update has been applied by using "soft" target updates [41], rather than directly copying weights, Target network weights slowly track the learned networks: $\theta \leftarrow \tau\theta + (1 - \tau)\theta'$ with $\tau << 1$.

- **Action Repetition:** define the granularity at which agents can control gameplay, by repeatedly executing a chosen action A for a fixed number of time steps k (instead of every frame), the last action is repeated on skipped frames**.** Computing the action once every k time steps and hence operate at higher speeds, thus achieving real-time performance. Two modes can be used, Static frame skip rate where Action output from the network is repeated for a fixed number of frames regardless of the current state, and Dynamic Frame skip which is an improvement [42] of the first mode which makes the frame skip rate a dynamic learnable parameter, choose the number of times an action is to be repeated based on the current state.
- **Clipping Rewards** [−1, 1]: due to the high variance of score from game to game in ATARI, all positive rewards are fixed to 1 and all negative rewards to −1, leaving 0 rewards unchanged, this technic limits the scale of the error derivatives and makes it easier to use the same learning rate across multiple games, but the major drawback is that Agent doesn't differentiate between rewards of different magnitude.
- **Experience replay:** DQN suffers from 2 main problems, the first is that in online learning, data are not i.i.d, samples frames arrive in the order, so they are highly correlated, which leads the network to overfitting and failure to generalize properly, the second concern **Catastrophic interference** [43] where Neural Network abruptly forgets what was previously learned when learning new things. To address those issues, instead of learning online, by updating Network from the last transition, we store agent experience (s_t, a_t, r_t, s_t+1) in replay memory D, then we train our network on random mini-batch of transitions (s, a, r, s') as input, which are sampled from the replay memory D. Experience replay break Similarity of subsequent training samples that might drive the network into a local minimum and solves the challenge of 'data correlation' and 'non-stationary data distributions.
- **No-ops vs human starts**: two modes are possible to initialize and populate the Experience replay memory: First, we have the no-ops mode, where actions are provided randomly at the beginning, until the Memory Replay is full enough to sample from it, and second, we have the human start mode, where actions are provided by a human user at the beginning (an expert), until the Memory Replay is full enough to sample from it. This last mode gives the network a more efficient initialization that helps to accelerate learning.
- **Actions Selection:** for Exploration vs Exploitation dilemma, DQN uses ε-greedy Approach which forces the non-greedy actions to be tried (exploration) with no preference for nearly greedy or particularly uncertain ones (chooses equally among all actions). ε is the probability of exploration (typically 5 or 10%). Most of the times ε decay through time, example: $\varepsilon = \varepsilon_{min} + (\varepsilon_{max} - \varepsilon_{min}).e^{-\lambda t}$, where λ controls the speed of decay.

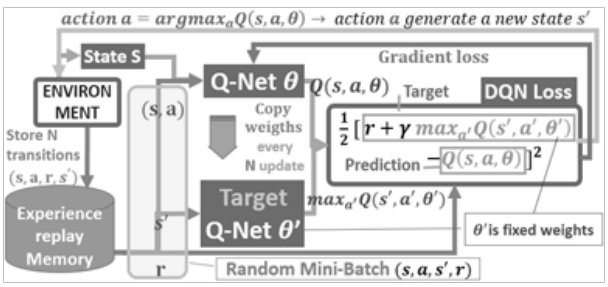


Fig. 15. DQN global architecture

4.1.2. GORILA

GoRiLa [44] is a General Reinforcement Learning architecture and a massively distributed and parallelized version of the DQN algorithm, achieved by introducing parallelization along three axes:

- **Actors:** Gorila supports N_{act} actors operating in parallel on N_{act} instantiations of the same environment. Each actor's experience can be stored in local/global memory.
- **Learners:** Gorila supports N_{learn} concurrent learners sample experience from the local or global store. Learners apply RL (DQN) to a replica of Q-network to generate gradient **gi** that updates master Q-net
- **Parameter server:** N_{param} master nodes Servers maintains a distributed Q-net(θ) splitted across the Nparam servers, they receive learner's gradient & applies appropriate updates to the subset of θ and periodically sends an updated copy of the Q-net to each learner.

4.1.3. Deep Recurrent Q-Network(DQRN)

DQRN [45]**:** A DQN agent can only see its closest area. by augmenting DQN with Recurrent neural nets and replacing DQN's last fully connected layer with recurrent LSTM layer of the same dimension, DRQN agent remembers the bigger picture and where things are, in fact, LSTM provides a selective memory of past game states allowing to improve the agent experience and efficiency. With this LSTM layer, the agent receives only one frame at once from the environment, and thanks to the hidden state of the LSTM, it can change its output depending on the temporal pattern of observations it receives.

4.1.4. Double DQN

Double DQN [46]**:** Being very noisy, DQN tends to overestimate action values as the training progresses. Due to the max term in the Bellman equation, the highest positive error is selected and this value is subsequently propagated further to other states. To overcome this issue, Double DQN uses two function approximators, Network QA and Network QB, one for selecting the best action and the other for calculating the value of this action; the two networks are symmetrically updated by switching their roles after each training step of the algorithm (see Fig. 16). By decoupling the maximizing action from its value, we can eliminate the maximization bias.

Fig. 16. Double DQN use 2 Networks QA and QB that switch their roles after each training step

4.1.5. Prioritized Experience Replay (PER)

PER [47]**:** Neuroscience has shown that the brain "replays" the past experience during awake resting or sleep, and more frequently sequences which are linked to the reward and to unexpected transition that have largest TD-error which have the highest opportunity of learning progress. PER increase the replay probability of transitions with the highest |TD-errors|, by changing the sampling distribution, and then store experience in priority queue ranked using the criterion TD-error.

4.1.6. Dueling DQN

Dueling DQN [48]**:** the goal is to produce separate estimations of state value V(s) which shows how good it is to be in any given state and advantage A(s,a), which shows much better it is, taking a certain action **a** in a state **s**, than was expected on average (see Fig.17). To achieve it, we use a single Q-net with 2 streams V and A (see Fig. 18). This decomposition allows a more robust estimate of state value by decoupling it from the necessity of being attached to specific actions. Dueling reuse also the Double DQN and PER principles.

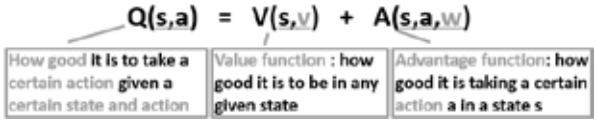


Fig. 17. The relation between the Action-value Q(s,a), the state value V(s) and the Advantage A(s,a)

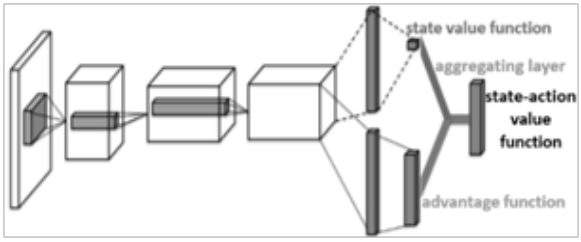


Fig. 18. Dueling DQN architecture

4.1.7. Noisy Nets for Exploration

Noisy Nets for Exploration [49]**:** for tackling Exploration/Exploitation dilemma in RL, there are two most commonly used ways, either we introduce a decaying randomness in the choice of the action (ex. Epsilon greedy), or we punish our model for being too certain in its actions (ex. Softmax with temperature parameter τ), this 2 methods have their drawbacks, since they need to be adjusted to the environment and don't take into account the current situation agent is experiencing. **Noisy Nets** came with a 3rd approach, by introducing a Gaussian noise function (σ,μ) that perturbs the last (fully-connected) layers of the network, with 2 ways:

- **Independent Gaussian Noise:** every weight of noisy layer is independent and has its own μ and σ, learned by the model.
- **Factorised Gaussian Noise:** we multiply 2 noise vectors, which respectively have the length of the input and output of the noisy layer, the result is used as a random matrix, which is added to the weights.

4.1.8. Rainbow

RAINBOW [50] is made by combining the following Improvements in Deep Reinforcement Learning, Double Q-learning, Prioritized replay, Dueling networks, Multi-step learning, Distributional RL and Noisy Nets. A ranking based on the degree of influence that has been made, by removing elements one by one from Rainbow. Experience shows that Prioritized replay and multi-step learning were the two most crucial components of Rainbow. Removing either component caused a large drop in performance. Then comes, the distributional Q-learning ranked immediately below, but have no influence on early learning stage. In the third place, we have Noisy nets and dueling network and double Q-learning that haven't much significant impact on the whole model.

Prioritized replay > Multistep >> Distributional > Noisy Nets >> dueling net > double DQN

4.2. Policy Optimization Algorithms (Actor-Critic)

Policy optimization is RL techniques that aim to optimize a parameterized policies $\pi(\theta)$, represented by a neural network, with respect to the expected return by using 1st or 2nd order optimization methods.

For Neural networks, gradient descent methods applied to the loss function, based on first order approximation, and used to update weights through backpropagation, reached their limits in term of performance. Optimization methods using Newton methods with second-order Taylor polynomial as a better approximation of the loss function and adopting various approximation of the Hessian H are explored as an alternative for more improvements (28):

$$L(\theta_k+\delta)=\frac{1}{2}\delta^T B(\theta_k)\delta+\nabla L(\theta_k)^T\delta+L(\theta_k) \quad (28)$$

where B is an approximation of the hessian.

However, the calculation of the Hessian approximation (Generalized Gauss-Newton matrix, Fisher information matrix, Hessian-free, …) remain complex and time-consuming, especially for high dimension space environment.

The use of second-order optimizers like with Natural gradient descent algorithm, significantly reduces the number of iterations, with the high-quality curvature matrices, it passes from ~102 iterations, instead of 104 iterations with SGD (stochastic gradient descent).

In the following section, we will have an overview of seven algorithms, of which five are first order: A3C, UNREAL, DDPG, PPO and ACER and two are second order: TRPO and ACKTR.

4.2.1. Advantage Asynchronous Actor-Critic Agents (A3C)

Advantage Asynchronous Actor-Critic Agents (A3C) [51] is a DRL algorithm that relies on the following principles:

- **Asynchronous**: by reusing Gorila parallelization and running multiple agents in parallel (see Fig. 19), each with its own copy of the environment, so their experiences are diverse, independent and not correlated, as result, we don't need experience replay memory anymore.

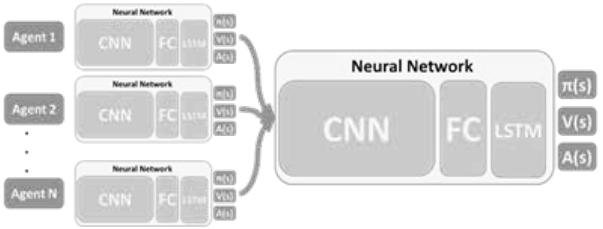


Fig. 19. A3C network architecture

- **Generalized Advantage Estimation GAE** [52]**:** by reusing Duel DQN principal, since we won't be determining the Q values directly in A3C, we use the discounted returns R as an estimate of Q(s,a) to allow us to generate an estimate of the **advantage**: **R = r + γV(s') ~ Q(s,a) à A(s,a) = Q(s,a) – V(s)= r + γV(s') – V(s),** and using GAE to reduce variance, by taking exponentially weighted average **λ**:

$$\hat{A}_t^{(n)}=\sum_{k=0}^{n-1}\gamma^k r_{t+k}+\gamma^n V(s_{t+n})-V(s_t) \text{ and}$$
$$\hat{\boldsymbol{A}}_t^{\boldsymbol{\lambda}}=\hat{A}_t^{(1)}+\lambda\hat{A}_t^{(2)}+\lambda^2\hat{A}_t^{(n)}+\ldots=\boldsymbol{\delta}_t+\boldsymbol{\gamma\lambda}\hat{\boldsymbol{A}}_{\boldsymbol{t+1}}^{\boldsymbol{\lambda}} \quad (29)$$

- **Exploration: H** is the entropy of the policy **π** is used as a mean of to **improving exploration**, by encouraging the model to be conservative regarding its **sureness of the correct action:** $\mathbf{H_{entropy}(\pi)}$ **=** $-\Sigma(P(x)\log(P(x))$. **H** is the entropy of the policy **π**, which reflect the spread of action probabilities, the entropy will be high when we have similar probabilities, and will be low when we have a single action with a large probability.
- **Actor-critic:** Each agent is sharing two networks: the **Critic Net** evaluates the present states using the value function **V(s)**, while the **Actor Net** evaluates the possible values in the present state to make decisions using **π(s)**. The global loss includes 2 parts: the value loss related to the predictions of the critic and the policy loss (which include H entropy) related to the predictions of the actor. The policy loss then combine the 2 loss in the Global Loss, with L_{value} is set to 50% to make policy learning faster than value learning:

$\mathbf{H_{entropy}(\pi)}$ **=** $-\Sigma(P(x)\log(P(x))$, $\mathbf{L_{value}}$ **=** $\Sigma(R - V(s))^2$
and $\mathbf{L_{policy}}$ **=** $-\log(\boldsymbol{\pi}(a|s)) * A(s)$ **– β*H(π)**

$$L_{global}=\frac{1}{2}L_{value}-L_{policy}=0.5*\Sigma(R—\mathbf{V(s)})^2$$
$$-\log(\boldsymbol{\pi}(\mathbf{a}\mid\mathbf{s}))*\mathbf{A(s)}-\beta*H(\boldsymbol{\pi}(\mathbf{a}\mid\mathbf{s})) \quad (30)$$

These two losses will be backpropagated into the neural network, and then reduced with an optimizer through stochastic gradient descent.

4.2.2. Umsupervised Reinforcement and Auxiliary Learning (UNREAL)

UNSUPERVISED Reinforcement and AUXILIARY Learning (UNREAL) [53]**:** the idea is to augment the on-policy A3C with off-policy auxiliary tasks to learn a better representation without influencing directly the main policy control. These tasks share the same network parameters [CNN, FC, LSTM], but with different outputs.

The network is composed of 4 modules of which A3C is the main one:

- **A3C Module:** is the main on-policy module that feeds the Experience replay memory, from which the auxiliary tasks get their inputs.
- **Pixel Control Module:** it learns how your actions affect what you will see rather than just prediction, to change different parts of the screen, and how to control the environment. It's based on the idea that changes in the perceptual stream often correspond to important events in an environment. Auxiliary policies Qaux produced using Deconvolutional [54] neural network (which was used first for image segmentation), are trained to maximize the change in pixel intensity of different regions of input. Auxiliary control loss $L_{PC} = \sum_c L_Q^{(c)}$
- **Reward Prediction Module:** learn to predict future reward based on rewarding histories. Auxiliary reward prediction loss $\boldsymbol{L_{RP}}$ is optimized from rebalanced replay data.
- **Value Function Replay:** predicts the n-step return from the current state to promote faster value iteration. Replayed value loss L_{VR} is optimized from replayed data. A global loss function:

$$L_{UNREAL} = L_{A3C} + \lambda_{VR} L_{VR} + \lambda_{PC} L_{PC} + \lambda_{RP} L_{RP} \qquad (30)$$

With $\boldsymbol{\lambda_{VR}}$, $\boldsymbol{\lambda_{PC}}$, $\boldsymbol{\lambda_{RP}}$ are weighting terms on the individual loss components.

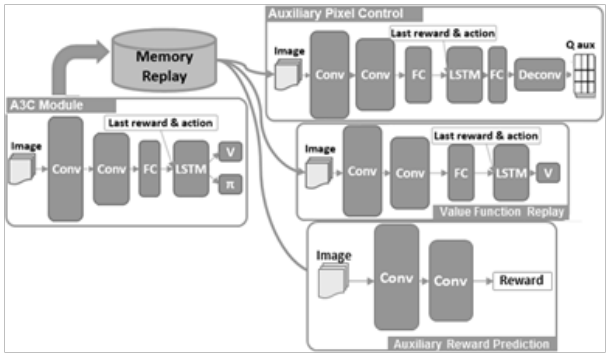


Fig. 20. UNREAL network architecture

4.2.3. Deep Deterministic Policy Gradients (DDPG)

DDPG (Deep Deterministic Policy Gradients) [41] is an actor-critic, off-policy gradient RL algorithm for continuous action space. It uses two neural networks, one for the critic and one for the actor which compute action predictions for the current state and minimize separately their two losses LActor and LCritic and follow Estimates a **deterministic** target policy. DDGP re-use DQN tricks:

- **Experience replay buffer** to solve the issue related to correlated data
- **Target Network**, make copies (Q',μ') of the Actor and Critic networks (Q,μ)and soft updates to enable training stability: $\theta^{Q'} \leftarrow \tau.\theta^{Q} + (1-\tau)\theta^{Q'}$ and $\theta^{\mu'} \leftarrow \tau.\theta^{\mu} + (1-\tau)\theta^{\mu'}$ with $\tau \ll 1$
- **Exploration** by adding noise to actor actions $\mu_{\text{Exploration}}(s_t) = \mu_\theta(s_t) + N_t$.

Even if DDPG has shown good performance but it needs to tweak the step size manually, so as to fall into the right range (too small → slow – too large → overwhelmed by the noise, bad performance).

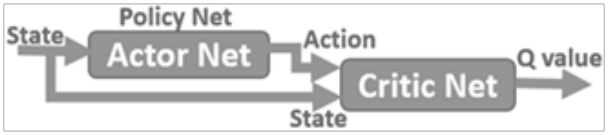


Fig. 21. DDPG network architecture

4.2.4. ACER (Actor-Critic with Experience Replay)

ACER [55]**:** is a model-free, off-policy, Asynchronous multi-agent, continuous control algorithm with actor-critic architecture. It is the off-policy counterpart of the A3C, with the addition of Experience replay memory. ACER uses the Retrace(λ) [56], which is an off-policy, Multi-step, value-based RL algorithm that reweights samples with a truncated importance sampling coefficient $C_s = \lambda min\left(1, \frac{\pi(a_s \mid x_s)}{\mu(a_s \mid x_s)}\right)$ to estimate Qπ, and thus, ensure low variance and safe efficient updates :

$$Q_{k+1}(x,a) = Q_k(x,a) + \alpha_k \sum_{t\geq 0} \gamma^t \prod_{1\leq s\leq t} \lambda \mathbf{min}\left(\mathbf{1}, \frac{\pi_k(a_s|x_s)}{\mu_k(a_s|x_s)}\right) \times \left(r_t + \gamma E_\pi\left(Q_k(x_{t+1},.)\right) - Q_k(x_t,a_t)\right) \qquad (31)$$

4.2.5. TRPO (Trust Region Policy Optimization)

TRPO [57] is a model-free, on-policy, continuous control algorithm that works for both discrete and continuous action space with actor-critic architecture. TRPO doesn't support including noise (e.g.dropout) or parameter sharing (between policy and value function, or auxiliary tasks). TRPO use natural gradient algorithm [58] to choose automatically the right step to apply for updating the policy network, which was done manually in DDPG.

TRPO uses an objective function $\eta(\pi_\theta) = \eta(\pi_{\theta old}) + E_{s_0, a_0, \ldots}\left[\sum_{t=0}^{\infty} \gamma^t A_{\pi_{old}}(s_t, a_t)\right]$, which is the expected return of policy π in terms of the **advantage $\mathbf{A_\pi}$** over the old policy πold, and with MM algorithm principle, TRPO create and try to maximize a surrogate function $L(\pi)$ which is a local first order approximation of $\eta(\pi_\theta)$

with an importance sampling term $\frac{\pi_\theta(s|a)}{\pi_{\theta old}(s|a)}$ to reduce variance.

The objective function is optimized when the surrogate function is optimized.

$$L(\pi_\theta) = E_{\pi_{old}}\left[\frac{\pi_\theta(s|a)}{\pi_{\theta old}(s|a)} A^{\pi_{\theta old}}(s|a)\right]$$
$$\nabla_\theta L(\pi_\theta)|_{\theta_{old}} = \nabla_\theta \eta(\pi_\theta)|_{\theta_{old}} \quad (32)$$

Maximize $L(\pi_\theta)$ under :

$$_{KL}\left(\pi_{\theta_{old}}, \pi_\theta\right) = \sum \pi_{\theta_{old}}(\ ) \log\left(\frac{_{old}(\ )}{(\ )}\right) \le \delta \quad (33)$$

So that the approximation remains valid and also avoid dramatic decrease in performance due to large changes from the previous policy, TRPO limits the size of the update step of the policy network's parameters, by applying KL divergence constraint, that measures the average distance between output distribution of the old policy network $\pi_{\theta old}$ and new policy network π_θ. KL constrain keep the step size within a "trust region" defined by δ, and allows modifying network parameters unequally, each one changes according to how much it affects the net output distribution regarding the KL constrain. So, KL divergence between the two networks will be as high as the difference between the outputs probabilities.

$$Maximize\ L(\pi_\theta)\ under, D_{KL}\left(\pi_{\theta_{old}}, \pi_\theta\right)$$
$$\sum \pi_{_{old}}(\ ) \log\left(\frac{_{old}(\ )}{(\ )}\right) \le \delta \quad (34)$$

By using the 2nd order Taylor series approximation for the KL divergence : $D_{KL}(\pi_\theta, \pi_{\theta+\Delta\theta}) \approx \frac{1}{2}\Delta\theta^T \boldsymbol{F} \Delta\theta$, with F, is the fisher information matrix (FIM) as the Hessian, and the 1st order Taylor series of $L(\pi_\theta)$ is $L(\theta) \approx L(\theta_{old}) + (\theta - \theta_{old}).\nabla L(\theta_{old})$. So we a have a constrained problem to optimize, and then we turning it to an unconstrained one using Lagrangian multipliers method:

$$L(\theta_{old}) + (\theta - \theta_{old}).\nabla L(\theta_{old}) \text{ and}$$
$$\frac{1}{2}(\theta - \theta_{old})^T \boldsymbol{F}(\theta - \theta_{old}) \le \delta$$
$$L(\theta) + (\theta - \theta_{old}).\nabla L(\theta) + \frac{1}{2}\lambda(\theta - \theta_{old})^T \boldsymbol{F}(\theta - \theta_{old}) \quad (35)$$

To minimize the quadratic function, we use conjugate gradient algorithm (CG) that allows to approximately solve the equation without forming the full FIM matrix, followed by a line search.

$$\nabla L(\theta) + \lambda \boldsymbol{F} \Delta\theta = 0 \Rightarrow \theta = \theta_{old} - \frac{1}{\lambda}\boldsymbol{F}(\theta_{old})^{-1}\nabla L(\theta_{old})$$
$$\theta_{new} = \theta_{old} + \frac{1}{\lambda}\boldsymbol{F}(\theta_{old})^{-1}.\nabla_\theta L_{\theta_{old}}(\pi_\theta) \quad (36)$$

TRPO resolved the step size problem but suffers from its extremely complicated computation and implementation, especially with FIM and CG.

4.2.6. PPO & PPO2 (Proximal Policy Optimization)

PPO & PPO2 [59] get rid of the computations in TRPO created by KL divergence constraint during the optimization process, as it proposes a new surrogate objective function $L^{CLIP}(\theta)$ by clipping the probability ratio $r_t(\theta)$, which removes the incentive for moving r_t outside of the interval [1 − ε, 1 + ε], it modifies TRPO's objective function by adding a penalty that sanction large policy updates :

$$L^{CLIP}(\theta) = E_t\left[\min(r_t(\theta)\hat{A}_t, clip[r_t(\theta), 1-\varepsilon, 1+\varepsilon]\hat{A}_t\right],$$
$$\text{with } r_{\pi_\theta} = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)} \quad (37)$$

PPO switch between sampling data from the policy and performing several epochs of optimization on the sampled data, while optimizing the policy.

PPO2 is the GPU-enabled implementation of PPO that runs roughly 3X faster than the original version of PPO.

4.2.7. ACKTR (Kronecker-Factored Approximation)

ACKTR [60] uses Natural gradient with K-FAC applied on the whole network (convolution layers and fully connected layers) [61], [62], which is a sophisticated approximation to the Fisher information matrix used in TRPO, to optimize both the actor and the critic. Combined with A2C architecture, where the two networks, Actor and Critic, share lower-layer representations but have distinct output layers to avoid instability during the training.

- **Actor:** use natural gradient with KL divergence constrain to update the network within a trusted region, adopting the same approach of TRPO with Fisher matrix, conjugate gradient, and line search. The K-FAC.
- **Critic:** least-squares loss using Gauss-Newton second-order approximation, the Gauss-Newton matrix G = E[JT J] where J is the Jacobian of the loss, is a positive semi-definite approximation of the Hessian and is equivalent to the Fisher matrix which allows applying K-FAC to the critic as well.

A correction is used for the inaccuracies of the local quadratic approximation of the objective, by adding (λ + η)I a Tikhonov damping term to the curvature matrix FIM, before multiplying −∇L by its inverse, which corresponds to imposing a spherical trust-region on the update.

Be a hidden layer k : $s_i = W_i a_{i-1}$, and $a_i = f_{act}(s_i)$ fisher matrix for this layer under the approximation that activations and derivative are independent :

$$F_{(i,j)(i',j')} = E\left(\frac{\partial L}{\partial w_{ij}}\frac{\partial L}{\partial w_{i'j'}}\right) = E\left(a_j \frac{\partial L}{\partial s_i} a_{j'}\frac{\partial L}{\partial s_{i'}}\right) \approx$$

$$E(a_j a_{j'}) E\left(\frac{\partial L}{\partial s_i}\frac{\partial L}{\partial s_{i'}}\right) = E(aa^T) \otimes E([\nabla_s L][\nabla_s L]^T) \tag{38}$$

With $\Omega_{k-1} = \mathrm{Cov}(a), \Gamma_k = \mathrm{cov}\left(\frac{\partial L}{\partial s}\right)$, in Kronecker vectorized form $F_k = \boldsymbol{\Omega}_{k-1} \otimes \boldsymbol{\Gamma}_k$. In practice, a two different Tikhonov damping terms are added to the Kronecker factors Ω_{k-1} and Γ_k:

$$\mathbf{F}'_k = \left[\boldsymbol{\Omega}_{k\ 1} + \alpha_i \mathbf{I}\right] \otimes \left[\boldsymbol{\Gamma}_k + \beta_i \mathbf{I}\right] = \boldsymbol{\Omega}'_{k+1} \otimes \boldsymbol{\Gamma}'_k \tag{39}$$

Under the approximation that layers are independent:

$$F_{Net} = \begin{bmatrix} \boldsymbol{\Omega}'_0 \otimes \boldsymbol{\Gamma}'_1 & 0 & 0 \\ 0 & \dots & 0 \\ 0 & 0 & \boldsymbol{\Omega}'_{L-1} \otimes \boldsymbol{\Gamma}'_L \end{bmatrix}$$

$$F_{Net}{}^{-1}\nabla L = \begin{pmatrix} vec(\boldsymbol{\Gamma}'^{-1}_1 [\nabla_{W_1} L] \boldsymbol{\Omega}'^{-1}_0) \\ \dots \\ vec(\boldsymbol{\Gamma}'^{-1}_L [\nabla_{W_L} L] \boldsymbol{\Omega}'^{-1}_{L-1}) \end{pmatrix} \tag{40}$$

4.3. DRL challenges

4.3.1. Credit assignment & Feedback Sparsity

Reinforcement learning gives good results in many use cases and applications, but it often fails in areas where the feedback is sparse.

Conceiving a reward function is a delicate task and generally, sparse discrete reward function is easier to define (e.g. get +1 if you win the game, else 0). However, sparse rewards also slow down learning, since the agent needs to take many actions before getting any reward, which is known as the credit assignment problem.

To speed up reinforcement learning algorithms and avoid spending a lot of time in areas, that likely won't help agent to achieve the assigned goal, it is usually mandatory to craft a continuous reward function, by shaping it smartly, depending on the environment and the goal to reach. Instead of having a sparse step function, we have a smooth continuous gradient function, which gives the agent information about the closeness to the goal.

Reward shaping is done by replacing the original reward function R of an MDP M={S, A, P, γ, R} by R' of transformed MDP M'={S, A, P, γ, R}, where R'=R+F with function F(s,a,s'): SxAxS → |R. To determine the right shape of the reward function F(s,a,s'), there are two relevant methods:

- Craft function reward manually [63], [64] like in Robotic, where F become usually, a function of distance and time.
- Use inverse reinforcement learning [65] by deriving a reward function (and the goals to achieve) from observed expert behavior (like using imitation learning to find the right policy) as in supervised learning (see Fig. 22).

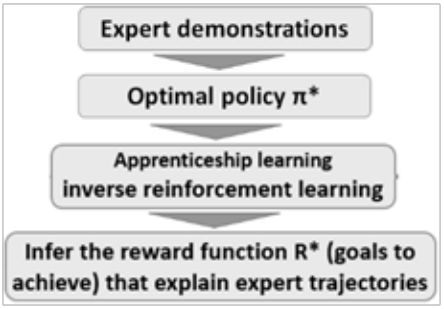


Fig. 22. Inverse reinforcement learning process

Shaping reward must take into account the fact that positive rewards encourage to keep going to accumulate reward and avoid terminals unless they yield very high reward, while negative rewards push the agent to reach a terminal state as soon as possible to avoid accumulating penalties. If the staged reward function is becoming large and complex, this is a good sign you should consider using concept networks instead.

4.3.2. Slow learning

DRL has well-performed in ATARI games and other real world tasks, but the pace of learning remain very slow, for instance, humans after 15 minutes tend to outperform DDQN after 115 hours. Many attempts has been made and are still made to bridge this gap, like the one-shot imitation learning [66], [67], whose goal is to learn in supervised mode, from very few demonstrations of any given task, and to be able to generalize to new situations of the same task, by learning to embed a higher-level representation of the goal without using absolute task and use transfer learning to communicate the higher level task, without retraining the model from scratch, another attempt has been made using Model-Agnostic Meta-Learning [68] where the agent called meta-learner trains the model or learner on a training set of large number of different tasks, so as to learn the common features representations of all the tasks, then, for a new task, the model with its prior experience provided by a good initialization (weights transfer) of its parameters, will be fine-tuned using the small amount of the new training data brought by that task with fewer number of gradient steps while avoiding overfitting that may happen when using a small dataset.

4.3.3. Complex Task

An important issue in RL is the learning ability to solve complex tasks, the main approach is using the principle of "divide and conquer", by using meta-learning principle, the goal is decomposed to a long chain of sub-goals, and learns to accomplish those sub-goals and recompose them, to define the overall solution. Many solutions have been proposed in that sense like:

- Hierarchical Deep Reinforcement Learning [69] or H-DQN where meta-controller learns the optimal goal policy and provides it to the controller that learns the optimal action policy or sub-policy. The meta-controller works at a slower pace than the controller and receives external feedback from the environment and provides incremental feedback for the controller.
- Concept network [70] where concepts are distinct aspects of a task that can be trained separately, and then combined using a selector concept (or meta-controller) to compose the complete solution.

4.3.4. Generalization and Meta-Learning

Current AI systems excel at mastering a single skill with lower versatility level, the challenge is to generalize over unseen instructions and over longer sequences of instructions. For the last two years, a lot of research has been made on the meta-learning topic, whose goal is to make a model that better generalize. In optimization, we have the example of DeepArchitect [71] that allow to automatically choose the architecture and hyperparameters for complex spaces, in meta-learning we have the deep meta-reinforcement learning (RL^2) that has been developed independently by Deepmind [72] and Openai [73], whose key ingredient in a Meta-RL system is a Recurrent Neural Network (RNN). The RNN-based agent is trained in supervised mode, to learn meta-policy that allows to exploit the structure of the problem dynamically and learn to solve new problems without retraining the model, but only by adjusting its hidden state instead of using backpropagation.

4.3.5. Variance & Biases Trade-off

In traditional supervised learning we have:

- **Biased** model generalizes well, but doesn't fit the data perfectly (under-fitting)
- **high-variance** model fits the training data perfectly but doesn't generalize well for new data (overfitting)

In RL, bias and variance measures show how close the reinforcement signal sticks to the true reward structure of the environment:

- **Bias**: refers to good stability with inaccuracy for the value estimate.
- **Variance**: refers to good accuracy with instability (noisy) for the value estimate.

Assigning credit to an RL agent acting in an environment can be done with different approaches, each with different amounts of variance or bias, for example:

- **High-Variance Monte-Carlo Estimate:** policies we are learning are stochastic because of a certain level of noise. This stochasticity leads to variance in the rewards received in any given trajectory
- **High-Bias Temporal Difference Estimate:** By relying on a value estimate instead of a Monte-Carlo rollout the stochasticity in the reward signal is reduced since the value estimate is relatively stable over time. However, we fall in another issue since the signal became biased, due to the fact that our estimate is never completely accurate. In addition, for DQN, Q-estimates are computed using the target network which is an old copy of the network, providing an older Q-estimates, with a very specific kind of bias.

There is a number of approaches that attempt to mitigate the negative effect of too much bias or too much variance in the reward signal:

- **Advantage Learning (reduced variance):** Actor-Critic methods are used to provide a lower variance reward signal to update the actor. $A\pi(st,at) = Q\pi(st,at) - V\pi(st)$, indicates how much better the agent actually performed than was expected on average, with Q(s, a) Monte-Carlo sampled reward signal, and V(s) parameterized value estimate. The high variance of the actor is balanced by the low-variance feedback on the quality of the performance supplied by the critic.
- **Generalized Advantage Estimate**: allows to balance between pure TD learning (bootstrapping method that add bias) and pure Monte-Carlo sampling (that add variance) by using a parameter λ. To produce better performance by trading off the bias of V(s) with the variance of the trajectory, we choose $\lambda \in [0.9, 0.999]$.

$$\hat{A}_t^{\lambda} = \sum_{k=0}^{\infty} (\gamma\lambda)^k \delta_{t+k} = \delta_t + \gamma\lambda \hat{A}_{t+1}^{\lambda} \Rightarrow$$
$$\Rightarrow \begin{cases} GAE\,\lambda = 0 : \hat{A}_t^{\lambda} = \delta_t \Rightarrow \textbf{TD Learning} \\ GAE\,\lambda = 1 : \hat{A}_t^{\lambda} = \sum_{k=0}^{\infty} \gamma^k \delta_{t+k} \Rightarrow \textbf{MC Sampling} \end{cases} \tag{41}$$

- **Value-Function Bootstrapping & Time Horizon**: Bootstrapping allows estimation of the Value-Function distribution using sampling methods. To make a compromise between Monte Carlo that uses all the episode steps for estimation and single-step TD methods that bootstrap, we act on the trajectories length to propagate the reward signal in a more efficient way. Time horizon corresponds to the number of steps of experience we collect before adding it to the experience buffer, it must be large enough to catch all the relevant behaviors within a sequence of an agent's actions. When the time horizon threshold is reached before the end of an episode, a value estimate is used to predict the expected total reward from the agent's current state. So, long time horizon leads to a less biased, but higher variance estimate and short time horizon leads to more biased, but less varied estimate. In cases where there are frequent rewards within an episode or episodes are extremely large, a smaller time horizon is more adapted.

4.3.6. Partial Observability Markov Decision Process (POMDP)

In Full MDP case, the agent has access to all the information about the environment it might need in order to take an optimal action, but real world problems do not meet this standard. Environments that present themselves in a limited way to the RL agent

are referred to as Partially Observable Markov Decision Processes (POMDPs) [74]. In a POMDP, the agent receives information that is spatially and temporally limited, so it partially describes the current state St, therefore, it is replaced by the observation Ot. The agent then attempts to predict what it has not sensed, by using other available information.

The main trick used to deal with POMDP is to augment the DRL net with an RNN/LSTM layers [45] that we position between Convolutional layers and fully connected layers, to keep in memory a history of the visual features that compensate for the lack of information. The Markov property is broken since the agent is no more memoryless.

4.4. DRL & RL Applications in the Industry

In the industry, DRL applications are diverse [75], depending on the purpose of the use, it can be split into three categories, first usage is for control like in robotics, Factory automation, and Smart grids, then second usage is for optimization like Supply chain, Demand forecasting, Warehouse operations optimization (picking), and finally for monitoring and maintenance like Quality control, Fault detection and isolation and Predictive maintenance. DRL lifecycle is composed of two phases, training phase, where we use a rough simulation that run fast and when it reaches the accuracy threshold desired, we switch to a higher fidelity simulation and retrain the model until it gets the targeted accuracy. For the deployment phase, the trained model is used in ground truth and tuned on physical equipment in the real world.

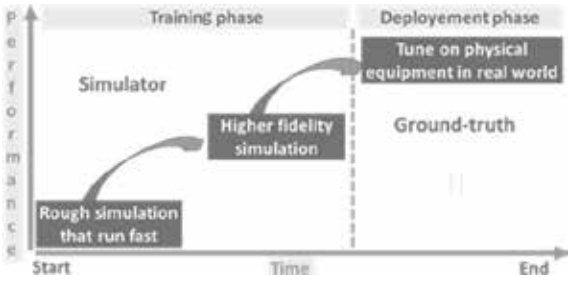


Fig. 23. The development cycle of DRL in the industry

Below some simulators used for RL/DRL in industry, see table 1 below:

Table 1. Most known RL/DRL simulators

Self-Drive-Fly	Mechanic & Electric	Robots & Drones
TORCS/Speed Dreams DeepDrive Udacity Simulator Unreal Engine simulator Unity XVEHICLE FlightGear AirSim	Matlab Simulink Sinumerik Wolfram SystemModeler OpenModelica	Gazebo MuJoCo RobotStudio RobotExpert Ardupilot NVIDIA Robotics simulator
Logistics	**Medical & Chemistry**	**Security & Networking**
Anylogistix Simutrans OpenTTD RinSim MovSim	CHEMCAD ParmEd PharmaCalc SOFA SimTK ArtiSynth SimCyp	VIRL NeSSI2 NS3 CupCarbon INET Conflict Simulation Laboratory

4.6. Deep RL Hardware

Neural network tasks like preprocessing input data, training the model, storing the trained model and deploying it, require intense hardware resource, and above all, training task is by far the most time and effort consuming, with the multiple forward and backward passes that are essentially matrices multiplications. The number of these operations can explode with a large network, for instance, VGG16 [30] a CNN of 16 hidden layers has ~140 million parameters (weights & biases).

To reduce the time of training, we can parallelize these computations. Thus, we often have the reflex to think about the CPU, however, the latter has few cores (e.g. 24 cores for INTEL E7-8890 v4) with a huge and a complex instruction set that handles every single operation (Calculation, memory fetching, IO, interrupts, ...). But the GPU contains by far, much more cores (e.g. 5120 for Nvidia Titan V and 4096 for AMD Radeon Vega 64), each of these cores has simpler instruction set and is specialized and optimized to do more calculation. In addition, Nvidia and AMD simplify the usage of GPU [76] for deep learning frameworks, by releasing and supporting high-level languages Cuda and OpenCL supported and included in these frameworks, helping researchers to write more efficient programs for their algorithms.

Since GPUs [77] are optimized for video games and not deep learning, they have some downsides like its higher power draw. Here is 2 alternatives to GPU, The first is FPGA which stands for Field programmed gate array, it's a highly configurable processor that allows tweaking the chip's function at the lowest level, it can be tailored specifically for deep nets application, so it consumes much less power than GPU, but they need highly specialized engineer to be configured. The second is called ASIC (Application-Specific Integrated Circuit) that is custom-designed chips optimized for deep learning, for instance, those made by Google named TPU (tensor processor unit), and the Nervana Engine built by Intel. To summarize, in terms of performance and power efficiency we have: ASICs >> FPGA > GPU >> CPU.

4.7. Deep RL Frameworks

General framework libraries that can be used to develop deep RL algorithms are Gym and Universe of

OpenAI, DeepMind Lab of Google, Project Malmo of Microsoft. Regarding the Deep Learning frameworks, we have the most known ones mentioned in the following table:

Table 2. Most known RL/DRL simulators

Dev Tools	Supporters	Pros	Cons
Tens or flow	Google, Uber	Community, ressources, documentation, CNN++, TensorBoard for visualization. Good for huge network	Slowness
Keras	Goosle, Kassle	Community, documentation, compatibility with tensorflow, CNTK and Theano as hish level API.	
Caffe2	Facebook, Twitter	Fast implementation and execution	RRN&GAN
Torch PyTorch	Facebook, Twitter, Nvidia	Community, documentation, Fast implementation & execution, CNN++.	
CNTK	Microsoft	RNN++ and NLP	community support
Paddle	Baidu		community support
Deep Learnings	JAVA Community	Use of Java, massively distributed	
MXNet	AMAZON, Microsoft	CNN++, massively distributed	NLP
Xeon	INTEL (Nervana)	Fast execution	community support
Power AI	IBM	Compatibily with IBM Watson	community support

5. CONCLUSION

Since the birth of Artificial Intelligence in the 50s, researchers in AI, machine learning, cognitive science, and neuroscience have wanted to build systems that learn, think and act like humans. Deep reinforcement learning has made great steps towards the creation of artificial general intelligence (AGI) systems that can interact and learn from the environment, which leverage three main points:

- Great idea and concepts: many of them were discussed in this paper like prioritized replay memory, Multi-step learning, reward shaping, imitation learning, meta-learning/generalization, Natural gradient with K-FAC.
- High-level libraries/API (Keras, Tensorflow, Pytorch) and simulation environment (Openai gym, Universe and Mujoco) that provide excellent testbeds for RL agents and simplify development and research.
- Powerful hardware (GPU & TPU) and high-level frameworks (Cuda & OpenCL) make it possible to achieve a significant gain in time and efforts.

However, DRL algorithms still suffer from the same drawbacks inherited from deep learning, so we still suffer from long training time, slow learning pace, catastrophic forgetting (of old tasks when training on new tasks), opacity of Black-box algorithms (since the chain of reasons for the action choice is not humanly-comprehensible). In addition to this, we have RL drawbacks like credit assignment problem, reward sparsity, variance and bias tradeoff, complex task management, complexity of meta-learning mechanisms and partial observability of the environment.

All these weak points are opportunities for improvement, and great challenges to overcome, which open widely the field of research for new ideas and breakthroughs that will one day lead to realizing the dream of seeing a perfectly autonomous and human-like intelligence in the real world.

AUTHORS

Youssef Fenjiro* – National School of Computer Science and Systems Analysis (ENSIAS), Mohammed V University, Rabat, Morocco.
Email: fenjiro@gmail.com.

Houda Benbrahim – National School of Computer Science and Systems Analysis (ENSIAS), Mohammed V University, Rabat, Morocco
Email: benbrahimh@hotmail.com.

*Corresponding author

REFERENCES

[1] "Sutton & Barto Book: Reinforcement Learning: An Introduction." Available at: http://incompleteideas.net/book/the-book-2nd.html
[2] Stuart J. Russell, Peter Norvig, Artificial Intelligence: A Modern Approach, 3rd edition. ISBN-13: 978-0136042594
[3] Y. LeCun, Y. Bengio, G. Hinton, "Deep learning", Nature, vol. 521, no. 7553, May 2015, pp. 436–444.
DOI: 10.1038/nature14539.
[4] V. Mnih et al., "Human-level control through deep reinforcement learning", Nature, vol. 518, no. 7540, pp. 529–533, Feb. 2015.
DOI: 10.1038/nature14236.
[5] A. D. Tijsma, M. M. Drugan, M. A. Wiering, "Comparing exploration strategies for Q-learning in random stochastic mazes". In: 2016 IEEE Symposium Series on Computational Intelligence (SSCI), Athens, Greece, 2016, pp. 1–8.
DOI: 10.1109/SSCI.2016.7849366.
[6] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," Jul. 2012. ArXiv:1207.0580 [Cs].

[7] R. Sutton, "Learning to Predict by the Method of Temporal Differences," Mach. Learn., vol. 3, pp. 9–44, Aug. 1988. DOI: 10.1007/BF00115009
[8] K. M. Gupta, "Performance Comparison of Sarsa(λ) and Watkin's Q(λ) Algorithms," p. 8. Available at: https://pdfs.semanticscholar.org/ccdc/3327f4da824825bb990ffb693ceaf7dc89f6.pdf.
[9] G. A. Rummery, M. Niranjan, "On-Line Q-Learning Using Connectionist Systems," 1994, CiteSeer.
[10] Yuji Takahashi, Geoffrey Schoenbaum, Yael Niv, "Silencing the Critics: understanding the effects of cocaine sensitization on dorsolateral and ventral striatum in the context of an Actor/Critic model", Front. Neurosci., 15 July 2008, pp. 86–99. DOI: 10.3389/neuro.01.014.2008l
[11] D. Silver et al., "Mastering the game of Go with deep neural networks and tree search", Nature, vol. 529, no. 7587, pp. 484–489, Jan. 2016. DOI: 10.1038/nature16961
[12] S. Hölldobler, S. Möhle, A. Tigunova, "Lessons Learned from AlphaGo," p. 10. S. H¨olldobler, A. Malikov, C. Wernhard (eds.): YSIP2 – Proceedings of the Second Young Scientist's International Workshop on Trends in Information Processing, Dombai, Russian Federation, May 16–20, 2017, published at http://ceur-ws.org.
[13] David Silver, Deepmind, "UCL Course on RL"
[14] Luis Serrano, A friendly introduction to Deep Learning and Neural Networks. https://www.youtube.com/watch?v=BR9h47Jtqyw
[15] S. Ruder, "An overview of gradient descent optimization algorithms," arXiv:1609.04747 [cs], Sep. 2016.
[16] N. Qian, "On the momentum term in gradient descent learning algorithms," Neural Netw., vol. 12, no. 1, pp. 145–151, Jan. 1999. DOI: 10.1016/S0893-6080(98)00116-6.
[17] J. Duchi, E. Hazan, Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization", JMLR, vol. 12(Jul), 2011, pp. 2121–2159.
[18] "Rmsprop: Divide the gradient by a running average of its recent magnitude – Optimization: How to make the learning go faster," Coursera.
[19] M. D. Zeiler, "ADADELTA: An Adaptive Learning Rate Method," ArXiv1212.5701 Cs, Dec. 2012.
[20] D. P. Kingma, J. Ba, "Adam: A Method for Stochastic Optimization," ArXiv1412.6980 Cs, Dec. 2014.
[21] J. Bergstra and Y. Bengio, "Random Search for Hyper-parameter Optimization", J. Mach. Learn. Res., vol. 13, pp. 281–305, Feb. 2012. ISSN: 1532-4435
[22] J. Snoek, H. Larochelle, R. P. Adams, "Practical Bayesian Optimization of Machine Learning Algorithms", p. 9. https://arxiv.org/pdf/1206.2944.pdf
[23] Yoshua Bengio, "Gradient-Based Optimization of Hyperparameters." DOI: 10.1162/089976600300015187.
[24] M. Sazli, "A brief review of feed-forward neural networks", Commun. Fac. Sci. Univ. Ank., vol. 50, pp. 11–17, Jan. 2006. DOI: 10.1501/0003168.
[25] Salman Khan, Hossein Rahmani, Syed Afaq Ali Shah, A Guide to Convolutional Neural Networks for Computer Vision. DOI: 10.2200/S00822ED1V01Y201712COV015
[26] Hamed Habibi Aghdam, Elnaz Jahani Heravi, Guide to Convolutional Neural Networks A Practical Application to Traffic-Sign Detection and Classification, Springer 2017.
[27] S. Ioffe, C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," ArXiv1502.03167 Cs, Feb. 2015.
[28] Y. Lecun, L. Bottou, Y. Bengio, P. Haffner, "Gradient-Based Learning Applied to Document Recognition". In: Proceedings of the IEEE, 1998, pp. 2278–2324. DOI: 10.1109/5.726791.
[29] A. Krizhevsky, I. Sutskever, G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks". In: Advances in Neural Information Processing Systems, 25, 2012, pp. 1097–1105. DOI: 10.1145/3065386.
[30] K. Simonyan, A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," ArXiv1409.1556 Cs, Sep. 2014.
[31] C. Szegedy et al., "Going Deeper with Convolutions," ArXiv1409.4842 Cs, Sep. 2014. DOI: 10.1109/CVPR.2015.7298594.
[32] M. Lin, Q. Chen, S. Yan, "Network In Network," ArXiv1312.4400 Cs, Dec. 2013.
[33] K. He, X. Zhang, S. Ren, J. Sun, "Deep Residual Learning for Image Recognition," ArXiv151203385 Cs, Dec. 2015. DOI: 10.1109/CVPR.2016.90.
[34] G. Huang, Z. Liu, L. van der Maaten, K. Q. Weinberger, "Densely Connected Convolutional Networks," ArXiv1608.06993 Cs, Aug. 2016.
[35] S. Sabour, N. Frosst, and G. E. Hinton, "Dynamic Routing Between Capsules," ArXiv1710.09829 Cs, Oct. 2017.
[36] S. Hochreiter and J. Schmidhuber, "Long Short-term Memory," Neural Comput., vol. 9, pp. 1735–80, Dec. 1997. DOI: 10.1162/neco.1997.9.8.1735
[37] "Understanding LSTM Networks". Colah's blog. 27/08/2015. https://colah.github.io/posts/2015-08-Understanding-LSTMs/.
[38] J. Yosinski, J. Clune, Y. Bengio, H. Lipson, "How transferable are features in deep neural networks?," ArXiv1411.1792 Cs, Nov. 2014.
[39] N. Becherer, J. Pecarina, S. Nykl, K. Hopkinson, "Improving optimization of convolutional neural networks through parameter fine-tuning", Neural Comput. Appl., pp. 1–11, Nov. 2017. DOI: 10.1007/s00521-017-3285-0.
[40] "Frame Skipping and Pre-Processing for Deep Q-Nets on Atari 2600 Games", Daniel Takeshi blog, 25/11/2016 https://danieltakeshi.github.io/2016/11/25/frame-skipping-and-preprocessing-for-deep-q-networks-on-atari-2600-games/.
[41] T. P. Lillicrap et al., "Continuous control with deep reinforcement learning," ArXiv1509.02971 Cs Stat, Sep. 2015.
[42] A. S. Lakshminarayanan, S. Sharma, B. Ravindran, "Dynamic Frame skip Deep Q Network," ArXiv1605.05365 Cs, May 2016.

[43] S. Lewandowsky, S.-C. Li, Catastrophic interference in neural networks: Causes, solutions, and data, Dec. 1995.
DOI: 10.1016/B978-012208930-5/50011-8
[44] A. Nair et al., "Massively Parallel Methods for Deep Reinforcement Learning", ArXiv1507.04296 Cs, Jul. 2015.
[45] M. Hausknecht, P. Stone, "Deep Recurrent Q-Learning for Partially Observable MDPs," ArXiv1507.06527 Cs, Jul. 2015.
[46] H. van Hasselt, A. Guez, D. Silver, "Deep Reinforcement Learning with Double Q-learning," ArXiv1509.06461 Cs, Sep. 2015.
[47] T. Schaul, J. Quan, I. Antonoglou, D. Silver, "Prioritized Experience Replay," ArXiv1511.05952 Cs, Nov. 2015.
[48] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, N. de Freitas, "Dueling Network Architectures for Deep Reinforcement Learning", ArXiv1511.06581 Cs, Nov. 2015.
[49] M. Fortunato et al., "Noisy Networks for Exploration," ArXiv1706.10295 Cs Stat, Jun. 2017.
[50] M. Hessel et al., "Rainbow: Combining Improvements in Deep Reinforcement Learning," ArXiv1710.02298 Cs, Oct. 2017.
[51] V. Mnih et al., "Asynchronous Methods for Deep Reinforcement Learning," ArXiv1602.01783 Cs, Feb. 2016.
[52] J. Schulman, P. Moritz, S. Levine, M. Jordan, P. Abbeel, "High-Dimensional Continuous Control Using Generalized Advantage Estimation," ArXiv1506.02438 Cs, Jun. 2015.
[53] M. Jaderberg et al., "Reinforcement Learning with Unsupervised Auxiliary Tasks," ArXiv1611.05397 Cs, Nov. 2016.
[54] H. Noh, S. Hong, B. Han, "Learning Deconvolution Network for Semantic Segmentation", ArXiv1505.04366 Cs, May 2015.
DOI: 10.1109/ICCV.2015.178.
[55] Z. Wang et al., "Sample Efficient Actor-Critic with Experience Replay", ArXiv1611.01224 Cs, Nov. 2016.
[56] R. Munos, T. Stepleton, A. Harutyunyan, M. G. Bellemare, "Safe and Efficient Off-Policy Reinforcement Learning", ArXiv1606.02647 Cs Stat, Jun. 2016.
[57] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, P. Abbeel, "Trust Region Policy Optimization," ArXiv1502.05477 Cs, Feb. 2015.
[58] S. M. Kakade, "A Natural Policy Gradient," p. 8. https://papers.nips.cc/paper/2073-a-natural-policy-gradient.pdf
[59] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, "Proximal Policy Optimization Algorithms," ArXiv1707.06347 Cs, Jul. 2017.
[60] Y. Wu, E. Mansimov, S. Liao, R. Grosse, Ba, "Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation," ArXiv1708.05144 Cs, Aug. 2017.
[61] J. Martens, R. Grosse, "Optimizing Neural Networks with Kronecker-factored Approximate Curvature," ArXiv1503.05671 Cs Stat, Mar. 2015.
[62] R. Grosse, J. Martens, "A Kronecker-factored approximate Fisher matrix for convolution layers," ArXiv1602.01407 Cs Stat, Feb. 2016.
[63] Bonsai "Writing Great Reward Functions" Youtube https://www.youtube.com/watch?v=0R3PnJEisqk
[64] X. Guo, "Deep Learning and Reward Design for Reinforcement Learning," p. 117.
[65] A. Y. Ng, S. Russell, "Algorithms for Inverse Reinforcement Learning". In: ICML 2000 Proc. Seventeenth Int. Conf. Mach. Learn., May 2000. ISBN:1-55860-707-2
[66] Y. Duan et al., "One-Shot Imitation Learning," ArXiv1703.07326 Cs, Mar. 2017.
[67] "CS 294 Deep Reinforcement Learning, Fall 2017", Course.
[68] C. Finn, P. Abbeel, S. Levine, "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks," ArXiv1703.03400 Cs, Mar. 2017.
[69] D. Kulkarni, R. Narasimhan, "Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation." arXiv:1604.06057 Cs.
[70] A. Gudimella *et al.*, "Deep Reinforcement Learning for Dexterous Manipulation with Concept Networks," ArXiv1709.06977 Cs, Sep. 2017.
[71] R. Negrinho and G. Gordon, "DeepArchitect: Automatically Designing and Training Deep Architectures," ArXiv1704.08792 Cs Stat, Apr. 2017.
[72] J. X. Wang et al., "Learning to reinforcement learn", ArXiv1611.05763 Cs Stat, Nov. 2016.
[73] Y. Duan, J. Schulman, X. Chen, P. L. Bartlett, I. Sutskever, P. Abbeel, "RL^2: Fast Reinforcement Learning via Slow Reinforcement Learning," ArXiv1611.02779 Cs Stat, Nov. 2016.
[74] M. T. J. Spaan, "Partially Observable Markov Decision Processes," Reinf. Learn., p. 27.
DOI: 10.1007/978-3-642-27645-3_12
[75] Bonsai, M. Hammond, Deep Reinforcement Learning in the Enterprise: Bridging the Gap from Games to Industry", Youtube. https://www.youtube.com/watch?v=GOsUHlr4DKE
[76] Emine Cengil, Ahmet Çinar, "A GPU-based convolutional neural network approach for image classification".
DOI: 10.1109/IDAP.2017.8090194
[77] "Why are GPUs necessary for training Deep Learning models?", Analytics Vidhya, 18-May-2017.