Advanced Techniques in Artificial Intelligence

Homework 1: An overview of (Deep) Reinforcement Learning

Pablo Felipe, Miguel Gil, Xabier Ugarte and Edu Vallejo

Abstract—

This work aims to provide a humble overview of the Reinforcement Learning (RL) field, and in specific Deep Reinforcement Learning (Deep RL). We first situate RL in a taxonomy of its neighbour and composing fields, there we talk about everything that needs to be known to start understanding RL, and we show why the RL approach makes sense. We then explain what RL algorithm paradigms exist and what are key types of methods. After that we show some trademark techniques of RL (and Deep RL) without getting into too much detail. Finally there is a discussion about exploration, which is one of the most crucial aspect of any RL algorithm.

Keywords—Reinforcement Learning , Deep Reinforcement Learning, Deep Learning, Neural Networks, Machine Learning, Optimization, Gradient Descent, AI, Agent, Environment

1 Structure of this work

This work is divided into several sections each of which builds on top of the previous one. Section 2 gives an introduction to the paper and contextualizes it. Section 3 situates Reinforcement Learning by explaining what the neighbour and composing fields are and formalizing the ideas that lead to RL. Section 4 explains briefly the core idea of RL and introduces DeepRL.Section 5 gives us a breakdown of the types of algorithms and paradigms that can be currently relevant in the RL scope. Section 6 explains some of the most popular algorithms without getting into too much detail. Finally section 7 concludes with a discussion about agent exploration, one of the most determining factors in the success or failure of any RL algorithm.

Disclaimer: we will mostly refer to RL throughout the document, but it shouldn't matter since most things can be applied to Deep RL as well.

2 Introduction

It is not news that artificial intelligence agents have achieved superhuman-like performance on a series of complex tasks[1]; tasks that most people would consider require of some real emergent creativity and cognitive ability that is not present in previous breakthrough AI works. In the center of this revolution stands Reinforcement Learning (RL), and in concrete Deep Reinforcement Learning (Deep RL)[2].

RL ,in a nutshell, consists in applying machine learning techniques to automatically learn agents that do well (with respect to some metric) in an environment by optimizing for some objective function in the space of agent policies [3]. Deep RL is just augmenting the RL field with Deep Learning concepts, which often involve (but are not limited to) deep neural network architectures and the methods to train them[4]. This is not a new idea, but has caught a lot of attention lately due to the achievements of Deep RL in training agents for game environments and game-like systems.

While RL has proven its worth in these clean-room conditions RL methods currently struggle with real-world environments where the signals are weaker, ambiguous and unpredictable; however this is currently a hot line of research that has a lot of expectations[5]. In any case, the potential of this field can not be understated and

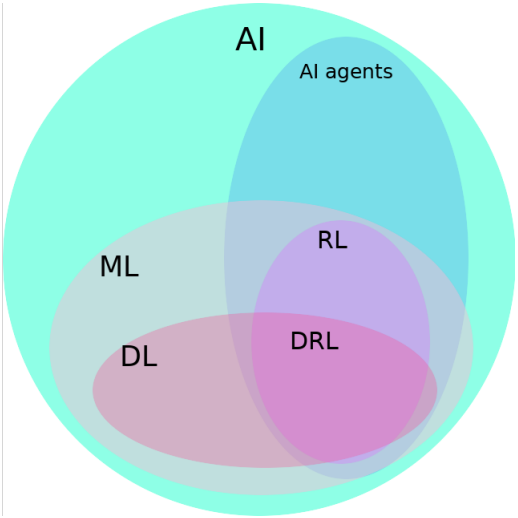

Figure 1: A taxonomy of the disciplines relevant for RL

is considered by many experts to be the path to AGI (Artificial General Intelligence)[2].

3 Situating Reinforcement Learning

As we have said, RL exists in the intersection of the AI agents and Machine Learning fields (see figure 4), the idea being that the behaviour of the agent can be learned from data about environment interactions. However this combination of fields is not as trivial as it might seem as we will explain in this section; for that, we will break down the ideas that led to the proposal of Reinforcement Learning as a technique and ultimately to the revolution that is currently happening.

3.1 AI agents and environments

Lets start, in the first place, by talking about the problem that RL was invented to solve, that is: how to construct agents that behave intelligently in a target environment. This is a very general problem, and as such a lot of problems from other fields (e.g. Game Theory, Control Theory, Swarm Intelligence, Theory of Optimal Control) can be traced back and reduced to this simple problem description. It is not surprising then that the community considers it an AI-complete problem, meaning that, solving this problem solves the core problem of AI that

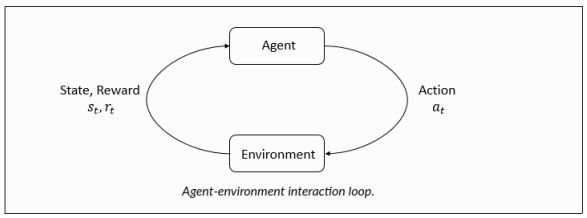

Figure 2: The feedback loop between the agent and the environment

all AI fields struggle with.

The central elements in this problem description are the agent and the environment. The agent lives in the environment and it constantly interacts with it. For the most part, the field of AI agents assumes that the interactions between agent and environment are performed at discrete time steps.

In the following subsections we will explain the concepts that are important for the AI agent problems, for that we make formalize these concepts, which we will use later.

3.1.1 *Observations*

At each time step the agent receives an observation o of the state s of the environment, this observation is a surrogate of the real state though, since it may be the case that there is incomplete information (partially-observable environments) and/or there is noise in the observation signal (real-world environments).

In any case we call the set of all possible observations the **observation space** O and we may call the set of all possible environment states the **state space** S. So at each time step t the environment is at state

$$s_t \in S$$

and the agent receives

$$f(s_t) = o_t \in O$$

, where f is a mapping between S and O,

$$f : S \longrightarrow O$$

f may preclude some information and/or introduce some noise into it (as explained previously),

for some environments f is just the identity function and in that case

$$s_t = o_t$$

, this is the case of fully-observable environments (e.g. chess).

State and observation spaces can be continuous, discrete or a mixture of both, but are almost always multidimensional; whether it is vectors, matrices or higher order tensors.

3.1.2 Actions

After receiving an observation s_t at time t an agent will decide on what action a_t to take. In this formalism the agent always takes an action, so we consider that doing nothing is taking the action *do nothing* sometimes expressed as

$$a_t = e$$

(the identity element).

In the case of actions too, we call the set of possible actions the **action space** A, which may again be discrete, continuous or something in between and will probably be multivariate.

For determining the action $a_t \in A$ to take, the agent may use all the information about the interactions with the environment, including the last observation. This information (or a surrogate of it) is stored in the memory of the agent commonly referred to as the Knowledge Base (KB) of the agent. The action a_t at time t is then a function μ of the KB_t,

$$\mu(KB_t) = a_t$$

. In most of the literature μ is called **policy**, and acts as the "brain" of the agent.

The KB gets updated when an observation o_t comes in from the environment and may be optionally updated with the action a_t the policy took in that time-step. So we have

$$KB'_t = f_{update1}(KB_t, o_t)$$

,

$$a_t = \mu(KB')$$

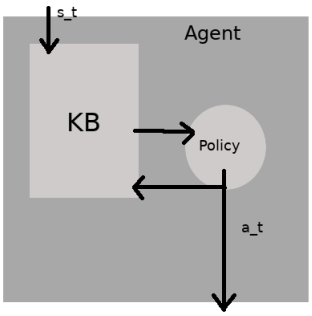

Figure 3: Simple diagram of the internals of an agent

and

$$KB_{t+1} = f_{update1}(KB'_t, a_t)$$

When the environment is fully observable there is no need for KB since the observation $o = s$ already includes all the information available. In this case there is no KB and the next action a_t to take is solely a function μ of the last observation o_t,

$$a_t = \mu(o_t)$$

, but since $o_t = s_t$ this is commonly formulated as

$$a_t = \mu(s_t)$$

We've presented deterministic policies to make things simple, but policies may also be stochastic, meaning that they don't necessarily return the same action a for the same KB (or observation s). Instead they return an action a_t under a given KB'_t (or state s_t) depending on some probability distribution. A stochastic policy gets modeled as a random variable π, and to get a_t you need to sample π given KB'_t (or s_t), that is

$$a_t \sim \pi(\cdot|KB'_t)$$

or

$$a_t \sim \pi(\cdot|s_t)$$

For the rest of this paper we will only talk about policies, Knowledge Bases are important but it's just an extra hurdle and there are a lot of detail.

3.1.3 State transitions

Between time-steps the environment state may change. This may happen because the agent acted on it or it may change on its own also.

When the state of the environment solely depends on the current state and in the action taken by the agent, we say it is a **deterministic environment** and there will be a function P (which the agent may not know) that maps from state action pairs (s_t, a_t) to new states s_{t+1},

$$s_{t+1} = P(s_t, a_t)$$

In the case of stochastic environments, a random variable P of possible next states exists which, when sampled given that state s_t and action a_t are know, returns a new state s_{t+1} with some probability of a distribution

$$s_{t+1} \sim P(\cdot|s_t, a_t)$$

If the environment is stochastic, the initial state is usually given by sampling a random variable p_0 that has a probability distribution for different starting states

$$s_0 \sim p_0$$

3.1.4 Episodes

An episode (or roll-out) τ is a sequence of contiguous states and actions $(s_0, a_0, s_1, a_1, ..., s_n)$ that represent the trajectory of the agent in the environment. It lets us talk about the agent acting in the environment in a more general way, without nitpicking in each individual interaction.

3.1.5 Utility functions

In the description of the AI agent problem we have said that we want to build agents that "behave intelligently", however what do we exactly mean by "intelligently"? Intelligent behaviour shows up as the means to accomplish an objective, thus we would like to know whether the agent accomplishes the objective and we may also want to asses the efficiency (time-steps needed, cost of the actions taken, ...) of the agent in doing so.

It is apparent that we need some sort of metric to measure how well an agent is doing in the environment. We call a metric R that measures agent performance through the duration of an episode τ an **utility function**. R is a scalar function that returns higher values for episodes that went better than for those that did not.

The utility function is not part of the environment or the agent, but a tool of the agent designer to benchmark the agent, as such, the designer has to define this function in such a way that it rewards the behaviour the designer is looking for and penalizes bad behaviour.

3.1.6 Fitness functions

An utility function R gives a measure of how well an agent did in a given episode τ, this will depend on how good the agent is on the target task, however, if there is a source of uncertainty in the agent (a stochastic policy) or in the environment (a stochastic environment), it may be the case that the agent got lucky and did better than it usually does, or that it got unlucky and did a lot worse. This variance is fine for talking about and comparing episodes, but it is not a good if we want to make meaningful statements about the agent.

A function J that measures the performance of an agent in an environment overall, without considering individual episodes, is called a **fitness function**. A fitness function J of a policy π is the expected value of the utility function R when generating episodes according to the policy of π and to the state transition function P. Assuming both the policy and the environment are stochastic (if some of them is deterministic is even simpler)

$$J(\pi) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)]$$

if we expand this expectation

$$J(\pi) = \int_\tau \mathop{P}_{\tau\sim\pi}(\tau) * R(\tau)$$

where $\mathop{P}_{\tau\sim\pi}(\tau)$ is the probability of a trajectory τ in the stochastic environment sampling actions from the stochastic policy π, so this looks like

$$\underset{\tau\sim\pi}{P}(\tau) = p_0 \prod_{t=0}^{n} P(s_{t+1}|s_t, a_t)\pi(s_t|a_t)$$

There is a problem with this expression though, the fact is we do not know the distribution (or function) P unless we have a complete model of the environment, this is rarely the case. Instead this expressions serves more as a formal mathematical baseline for the methods.

This means we can't analytically obtain J and so we can't directly calculate $J(\pi)$, however, we can do the second best thing: Form a numerical sample estimate of $J(\pi)$.

$J(\pi)$ is just the expectation of $R(\tau)$ (under policy π), to form a sample estimate $\hat{g}$ we only need to sample a set D of episodes (i.e. let the agent with policy π run in the environment for various episodes) and compute the mean of their utility values R

$$J(\pi) \sim \hat{g} = \frac{1}{|D|} \sum_{\tau_i \in D} R(\tau_i)$$

The choice of the size of D has to take into account the cost of each episode and the required estimate accuracy. A simple simulation will have a low cost so a high number of roll-outs can be recorded while the cost on a real-life environment is prohibitive and you probably will have to settle with a less accurate estimator $\hat{g}$ for $J(\pi)$.

3.1.7 The goal of agent design

With all of the agent formalism out of the way we can finally talk about the goal of this field: For some problem (environment and agent interface description), we want to find policies (and KB update functions) π that have a high fitness value. How high it needs to be depends on the needs specified in the problem. We may sometimes talk about optimality but finding optimal policies is not realistic for all but the simplest environments.

The space of policies to consider is all the computable functions that agree in their signature

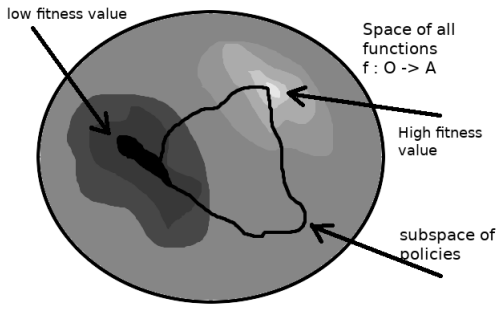

Figure 4: Diagram to show a possible look of the space of policies

with the observation and action spaces, so every μ and π that

$$\mu : O \longrightarrow A$$

and

$$\pi : O \longrightarrow P(A)$$

In the Good-Old-Fashioned Artificial Intelligence (GOFAI) days, researchers would, with a lot of work, devise pretty good algorithms for the policy (an algorithm is just an abstract implementation of a function). These methods are not only very expensive in the human resources side, but also very inflexible and not scalable at all.

For most complex problems this type of techniques have proven to be infeasible, and even for simpler problems the GOFAI solutions get stomped by the Machine Learning counterparts, which leads nicely into the next topic.

3.2 Learning policies with machine learning

In the previous section we have established that the core problem of the design of AI agents is to find policies that have high fitness functions for the target environment (i.e. that their average utility performance, the expectation, is high in the environment).

We said that the valid policies for an agent are those that agree in signature with the interfaces of the agent, that is policies of the signature $\pi : O \longrightarrow A$.

With these definitions of the space of policies and the fitness function of policies, it is only natural to formulate this as a optimization problem, where the space of solutions is the space of policies; and the objective function is the policy fitness function.

Even for small sets of O and A, the amount of computable functions is anything beyond what a human can grasp. So manually checking all policies is out of the window. However a computer can methodically check a huge amount of policies in a short span of time.

While exhaustively looking at every policy is still practically impossible (no matter how much compute power, realistically speaking), we can be more clever about how we chose to search the space.

In this section we describe the machine learning techniques we can employ to efficiently look for a good solution in a space of policies.

3.2.1 Baseline ML algorithm

The baseline ML algorithm would involve enumerating all the policies that exist in the policy space, and assessing the performance of the policies one by one, to later choose the highest fitness function in a "king-of-the-hill" type of algorithm. This algorithm is sometimes called **British Museum**. Given enough time, the algorithm will find the best policy (if the space of policies is finite). This as previously stated is not a valid approach, but serves as a theoretical baseline to compare with other algorithms.

A simple improvement to this baseline is to sample the policies at random with some probability distribution, this is called **Monte Carlo sampling**, and while it may not find the optimal policy any quicker on average, it does have lower variance between different problems (environment and agent descriptions).

These baseline algorithms will always find the globally optimal policy, but they are so costly that cannot be applied in any interesting problem. One way of obtaining easier to run algorithms is to relax the properties of these brute-force approaches. For instance, finding the absolute optimum is very hard because only one or few exist in the space, however there will probably be a lot of "close-to-optimal" policies that are 99.9% as good. If we can settle with less than optimal but still very good solutions then our task becomes easier.

If we just want a good policy (a solution that is almost as good as optimal), then Monte Carlo sampling becomes a good choice for lower dimension spaces. However as the dimension of the space goes up this approach quickly becomes unusable.

British Museum can be augmented with heuristics to guide the search of the optimal policy. With good heuristics the performance can be improved a lot and if some properties are relaxed (like for example the completeness of the algorithm), the algorithm can find decent solutions if left long enough. This is the case of some of the classic search algorithms like **Hill Climbing** or the more general **Beam Search**. The problem that these algorithms have is that their complexity grows exponentially with the dimension of the search space, much like Monte Carlo sampling.

The phenomenon that produces higher dimension spaces to be harder to deal with is called the **dimensionality problem**. It is a problem that appears all throughout Machine Learning and is well studied. The notion of this problem is that to have a good understanding of how a space looks, you need to sample the space thoroughly in such a way that in any region of space there is a tightly packed cluster of sample points. This density of samples is hard to keep when the number of dimensions go up.

To illustrate this, imagine a 10 by 10 node lattice that spans a 2 dimensional space, if we want to span a 3 dimensional space with the same amount of precision, we now need a 10 by 10 by 10 lattice, which requires ten times the amount of sample points. It is as if the amount of free room

in the space grows exponentially with dimension.

To mitigate the effects of the dimensionality problem we can restrict the search to a smaller subspace (a subset of the policy space). We would like to select the space following two criteria: first, we want to keep the size of the search space to a minimum but we want the space to contain good solutions so we have to find the balance; second, we would like the subspace to have properties that will aid us in the search.

In the next section we will see how parametrizing the definition of the policy can help us both, reduce the search space and get valuable properties in the new search subspace.

3.2.2 Parametrized policies

Most of machine learning today uses parametrized models, where the span of the parameters of the model (what are all the possible values of the parameters) defines the model space. A parametrized model is usually a function $f_\theta(x)$ which has some parametric terms θ(that are usually scalar values) that alter the behaviour of the function. So what we would like to do now is to find the parameters so that the model does what we want, i.e. minimizes some metric of loss.

For us the model is the policy. The parameters θ of the parametric policy π_θ alter the behaviour of the policy, in this case we want to find θ to maximize the fitness function of the policy. Note how we stopped looking directly for the policy π and we are now optimizing for parameters θ. As you can see we have effectively transformed a problematic function search problem into a well-defined numeric optimization problem.

It look as though we did magic and now a hard problem is an easy one, however to obtain this properties we had to give up search space. The catch is that the space that these parametrized policies span is way smaller than that of all policies, so it might be (and probably will be the case) that our subspace doesn't contain the optimal policy. However, as previously explained, hoping to find the optimal policy is not realistic. Instead, we hope that the space that the parametrized policies span, is large enough to contain *quasi-optimal* policies.

If we have selected a good function $f_\theta(x)$ to parametrize, the span of the parameters will directly correlate with the span of the policy subspace. In that case, more parameters will result in higher parameter span, which in turn results in higher policy subspace span. In short, our parametrized generic policy covers more policies in the space of policies the more parameters it has. In the Machine Learning literature this is commonly called **model capacity**, for the sake of keeping it in the field of AI agents lets just call it **policy capacity**. We would like to find a good balance for policy capacity, one that is not to small to include good solutions in the subspace and one that is not too large to hamper the search.

The selection of the parametrized function is important not only to ensure that its capacity scales with more parameters, but also to provide interesting properties that can be useful when searching for solutions. A basic property that we may ask, is for the function f_θ to be continuous for all x with respect to θ. This is an interesting property to have, specially if the objective function that we are using to asses the quality of the parametrized function f_θ is also continuous with respect to it; in which case the objective function will be continuous with respect to θ and we can optimize for it directly forgetting about f altogether.

For this optimization problem we have that we want to maximize $J(f_\theta)$, and we know that the fitness function J is continuous with respect to θ. Continuity simply means that the function $J(f_{theta})$ will not have sudden jumps for similar values of θ. Armed with this property we can employ a myriad of local optimization techniques like **Evolution Strategies**, **Finite difference** methods, **Hill Climbing** (with no necessity for heuristic), **Beam search** (with no necessity for heuristic) and some global search algorithms like **Particle Swarm Optimization**, **Simulate annealing**, **Genetic Algorithms** and all other sorts of meta-heuristics and bio-inspired methods.

These techniques are far better than the baselines proposed in the previous section, however it is not what is used in the industry. See, the problem with this kind of methods is that they require to sample a lot of points of the fitness function to make any progress at all, and thus they take forever in large dimensional spaces. In the current Machine Learning state of affairs, the most used optimization methods are gradient based, which means that they compute the gradient of the objective function $\nabla_\theta J(f_\theta)|_{\theta_k}$ in the point of the current parameters θ_k (for each update of the parameters).

3.2.3 Gradient based methods

If we choose our parametrized functions carefully we can have the property of differentiability, having this property will further restrict the choice about the policy but we can always make that up with more parameters. With a differentiable policy and a differentiable fitness function we can get the gradient of the fitness function with respect to the parameters.

The gradient is useful because it tells us how to update the parameters θ to improve the fitness function $J(f_\theta)$ at each step, so we only have to keep track of one set of parameters and iteratively update them according to the gradient. This will yield progressively better parameters for the policy, which will eventually converge in a maximum. Since this is a type of local search, the maximum might be local and not global. If the fitness function were concave (in minimization convex) with respect to the parameters, we wouldn't have this problem, since there would only be one maximum and the algorithm wouldn't get stuck in another lower value local maximum.

Making sure loss functions are convex is a very important part of gradient methods and is being studied thoroughly.

We have talked about calculating the gradient of the fitness function with respect to the parameters of a policy, but how can we do that? Is it even possible?

Recall from the AI agents section that the fitness function J of a policy π (we are still going to use the stochastic policy) is

$$J(\pi) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)]$$

now if we introduce parametrized policies

$$J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[R(\tau)]$$

the gradient is then

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[R(\tau)]$$

if we expand the expectation and put the gradient inside (integral and gradient has nothing to do)

$$\nabla_\theta J(\pi_\theta) = \int_\tau \nabla_\theta \mathop{P}_{\tau\sim\pi_\theta}(\tau) R(\tau)$$

it looks like we are stuck, however we can get progress if we use what is now as the "grad-log trick" which is a reformulation of the gradient of the logarithm and the chain-rule

$$\nabla f = f \nabla \log f$$

this lets us proceed

$$\nabla_\theta J(\pi_\theta) = \int_\tau \mathop{P}_{\tau\sim\pi_\theta}(\tau) \nabla_\theta \log \mathop{P}_{\tau\sim\pi_\theta}(\tau) * R(\tau)$$

this is an expectation yet again so

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[\nabla \log P(\tau) R(\tau)]$$

if we recall the expression for P

$$\mathop{P}_{\tau\sim\pi}(\tau) = p_0 \prod_{t=0}^{n} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t)$$

the log of that will be

$$\log \mathop{P}_{\tau\sim\pi}(\tau) = \log p_0 \sum_{t=0}^{n} \log P(s_{t+1}|s_t, a_t) + \log \pi(a_t|s_t)$$

and applying the gradient, the first two terms don't depend on θ so

$$\nabla_\theta \log \mathop{P}_{\tau\sim\pi}(\tau) = \sum_{t=0}^{n} \nabla_\theta \log \pi(s_t|a_t)$$

so if we put everything together

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[\sum_{t=0}^{n} \nabla_\theta \log \pi(s_t|a_t) R(\tau)]$$

given that this is an expectation we make our estimation $\hat{g}$, the average over episodes in sample set D

$$\nabla_\theta J(\pi_\theta) \sim \nabla_\theta \hat{g} = \frac{1}{|D|} \sum_{\tau_i \in D} \sum_{t=0}^{n} \nabla_\theta \log \pi(s_t|a_t) R(\tau)$$

with the previous expression we can tackle the problem using gradient ascent. This is the most basic form of the policy gradient, and is sometimes considered RL territory, however RL methods don't use this expression.

The fact of the matter is that the fitness function is a very weak supervision signal and is too coarse grain. What we would like to have is some notion of cumulative score that gets updated every interaction step as fitness function, so as to provide a signal in every action.

This score we assign to the agent at each step is called reward and the sum of all the rewards collected during an episode is a fitness function called return. The reward concept is the main idea behind RL.

4 The idea of Reinforcement Learning

4.1 Reward

The Reinforcement Learning approach uses a reward function R that at each interaction at time step t, expressed by (s_t, a_t, s_{t+1}), computes a reward value r_t. So

$$r_t = R(s_t, a_t, s_{t+1})$$

The fitness function $R(\tau)$ of the agent is the sum of all rewards called **return** and expresses how well an agent did for the duration of an episode τ

$$R(\tau) = \sum_{t=0}^{n} r_t = \sum_{t=0}^{n} R(s_t, a_t, s_{t+1})$$

Before we had to meticulously craft a fitness function, now we need to engineer a good reward function, and the fitness function will directly depend on it.

4.2 Value functions

The goal of the agent will be to collect as much reward as possible, however it can't act greedy since making actions that give immediate reward might result in the agent going into states that have low value, i.e. the total reward that you can get from that state is low. The value of a state is formalized as **Value Function**:

$V^{\pi_\theta}(s)$ is the expected return that an agent acting according to parametrized policy π_θ will get starting from state s

$$V^{\pi_\theta}(s) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[R(\tau)|s_0 = s]$$

, this is called the **on-policy Value Function**

We may also talk about the maximum value of a state $V^*(s)$ as being the expected return that can be obtained if starting from state s and acting optimally

$$V^*(s) = \max_\theta \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[R(\tau)|s_0 = s]$$

, this is called the **optimal Value Function**

To succeed the agent has to trade-off the immediate and long term reward that the action will produce. The expected return of taking an action a in state s and proceeding according to a policy π_θ, is also formalized as **Q-function** or **Action-Value function**:

The on-policy Q-function $Q^{\pi_\theta}(a, s)$ tells us what return to expect, in average, if we take action a in state s and follow policy π_θ afterwards

$$Q^{\pi_\theta}(a, s) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[R(\tau)|s_0 = s, a_0 = a]$$

Likewise we have an **optimal Action-Value Function** $Q^*(a, s)$ which returns the expected

return but if we acted according to the optimal policy

$$Q^*(a,s) = \max_\theta \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[R(\tau)|s_0 = s, a_0 = a]$$

4.3 Q-learning

These functions are very useful to have since they give the agent the ability to plan. In the case of the optimal functions, obtaining them would allow the agent to act optimally by using these functions to determine what action will produce the most long term amount of reward and selecting the actions with that criteria. Of course we don't know any of these functions. But we can work around that:

We can estimate the on-policy functions, with simple sample estimation. The optimal ones however are a bit more complex, to get approximations of those we'll need to use **Bellman equations**. These equations are self-consistency equations that can be used to numerically approximate the Q-function using a sort of *value iteration* which is sometimes called **approximate dynamic-programming**. These techniques are standard issue in the field of RL known as **Q-learning**, but we won't get on them with much detail.

The previous technique is an off-policy optimization technique in the sense that we don't keep updating a single policy by continuously testing the policy and updating the parameters to slightly improve it. In this case we directly look for the Q-function which helps us act optimally.

These techniques are great when they work because they have very good sample efficiency, since they can use information from roll-outs of any policy. Whereas on-policy optimization techniques are limited to only using information from the roll-outs of just the last policy.

However off-policy methods are quite unstable and often don't converge, and when they converge is thanks to a lot of tricks employed to make the Q^* estimate stable. Because of that on-policy methods are more frequent as they are more stable and can work out-of-the-box in more problems.

4.4 On-policy optimization

The most basic RL version of policy gradient optimization builds upon the ML version with the fitness function, that if we recall is

$$\nabla_\theta J(\pi_\theta) \sim \nabla_\theta \hat{g} = \frac{1}{|D|}\sum_{\tau_i\in D}\sum_{t=0}^{n}\nabla_\theta \log\pi(s_t|a_t)R(\tau)$$

note that in this expression $R(\tau)$ acts as a weight that is higher for actions from good episodes and lower for actions from episodes that get low return (utility score). This is done regardless the action was a good or not, so if in a good episode the policy took a bad action this gradient will reinforce that action the same as the actions that led to the episode being good, the same happens in the opposite case.

The fact of the matter is that the actions are reinforced taking into account the past of the action, which from the point of view of how good the action was has nothing to do. An easy way to improve the above expression is to just reinforce each action depending on the future reward sum ignoring the past.

$$\frac{1}{|D|}\sum_{\tau_i\in D}\sum_{t=0}^{n}\nabla_\theta \log\pi(s_t|a_t)\sum_{t'=t}^{n} r_{t'}$$

This expression is equivalent to the previous one (the expectation is the same) but gives better approximation of the gradient with less samples, which makes convergence way faster.

4.5 Deep RL

Deep learning studies the learning capabilities of deep machine learning architectures. Deep architectures rely in function composition to get the representative power they are known for. Function composition generates a layered structure where each layer is a function that takes some input from the previous layer and computes an output which is passed to the next layer.

The layers create a hierarchical function which is expected to break down the problem in the input into simpler and simpler problems until we have a solution in the output. The layers

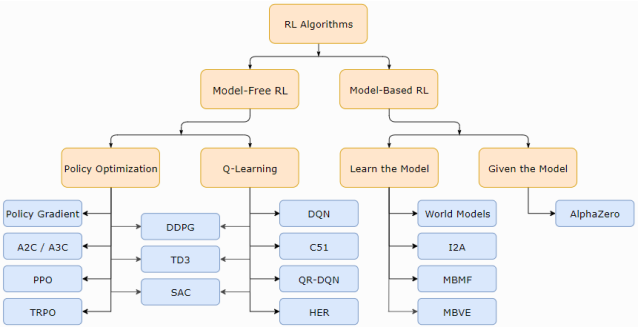

Figure 5: Taxonomy of algorithms in modern RL

are parametrized functions, so the whole thing is parametrized. The building block functions are also differentiable which makes the whole composed function differentiable.

These architectures are often depicted as computation graphs or networks, in particular they are known in the media as **Neural Networks** because the apparent similarities of both types of systems.

A deep architecture may be used in RL and in other fields to approximate functions. In Deep RL we use Neural Networks (there are a lot of types) as the parametrized policy and sometimes we may use it to approximate Value functions.

5 Kinds of RL Algorithms

A non-exhaustive, but useful taxonomy of algorithms in modern RL is presented in Figure 5.

5.1 Model-Free vs Model-Based RL

One of the most important characteristic of RL algorithms is whether the agent has access to a model of the environment. This model is actually a function which predicts state transitions and rewards.

The main advantage of using a model-based algorithm is that it allows the agent to plan, seeing what would happen for a range of possible choices and deciding between its options. Agents can use the results they obtain from thinking ahead to learn a policy.
On the other hand, it is not a common situation to have a ground-truth model of the environment. In this case, the agent has to learn the model from its experience. The biggest difficulty which this option creates is the possibility of generating a biased agent, performing well with respect to the learned model, but sub-optimally in the real environment.

Although model-free algorithms lose the potential gains in efficiency, they are more popular than model-based algorithms due to their facility to be implemented.

5.1.1 Model-Free Algorithms

There are two main approaches to represent model-free methods: Policy optimization and Q-learning.

In policy optimization algorithms the agent learns directly the policy function that maps state to action.
There are two types of policies. Deterministic policies map state to action without uncertainty. By contrast, stochastic policies output a probability distribution over actions in a given state. The type of policy depends on whether the environment is deterministic or not.
Policy optimization is mostly performed on-policy, which means that each update only uses data collected while acting according to the most recent version of the policy.

Q-learning learns the action-value function Q(s, a): how good it is to take an specific action at a particular state. Basically a scalar value is assigned to an action a given the state s. A representation of steps taken by Q-learning algorithms is shown in Figure 6.
This optimization is typically performed off-policy, which means that each update can use data collected at any point during training, regardless of how the agent was choosing to explore the environment when the data was obtained.

The primary strengths of policy optimization algorithms are their stability and reliability as they allow you to directly optimize for the thing you want.

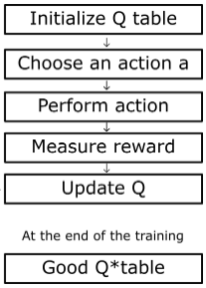

Figure 6: Steps taken by Q-learning algorithms

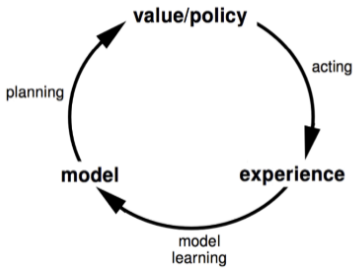

Figure 7: Representation of a generic model-based algorithm

Contrarily, Q-learning methods only indirectly optimize for agent performance, so they tend to be less stable. However, these algorithms are more sample efficient as a result of their capability to reuse data more effectively.

Policy optimization and Q-learning are not incompatible, and there exist a range of algorithms that mix features of both methods. These algorithms aim to combine the strengths of Q-learning and policy gradients.

5.1.2 Model-Based Algorithms

Model-based RL has a strong influence from control theory, and the goal is to plan through an f(s,a) control function to choose the optimal actions. There are two main approaches: learning the model or learn given the model. In Figure 7 is shown how a generic model-based algorithm learns and acts.

To learn the model a base policy is ran (e.g., random policy) while the trajectory is observed, and the model is fitted using the sampled data. A cost function is used to find the optimal trajectory with the lowest cost.

Contrarily, it might be the case where the model is already known and it can be given to the agent. For example, in many games, like Go, the rule of the game is a model (e.g., AlphaZero).

6 Policy Optimization

6.1 Overview

Once having an overview of the many RL related algorithms, we will specifically develop the Model-Free ones. As we already know, there are two main approaches to these kind of algorithms: Policy Optimization, on-policy, and Q-Learning family, off-policy. However, it is possible to use resources from both approximations in order to obtain good results. This is the case of Deep Deterministic Policy Gradient (DDPG) and Soft Actor Critic (SAC), both interpolations between Policy Optimization and Q-Learning.

On the one hand, DDPG is an algorithm that uses deterministic policies and Q-functions in order to make one improve the other. On the other hand, SAC is a variant of DDPG which instead uses stochastic policies, entropy regularization and clipped double-Q trick among others.

However, we will mainly focus on an algorithms completely based on Policy Optimization. In this part, we will need to first understand what Policy Gradients are. The idea underlying policy gradients is to push up probabilities of actions that lead to lower return, until you arrive at the optimal policy.

6.2 Vanilla Policy Gradient

The goal of VPG is to train stochastic policies in an on-policy way. Unlike off-policy algorithms, it does not use a replay buffer to store past experiences, so it learns from what the agent encounters in the environment in that exact moment. This makes it less sample efficient. It can be used in either discrete or continuous action spaces.

The action selection randomness lowers over the course of training, because it tends to exploit rewards already found, sometimes avoiding getting an optimum result.

Equation: (see Figure 8)

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t, a_t)\right]$$

Figure 8

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_{\theta_k})$$

Figure 9

π_θ: our parametrized policy.
J(π_θ): expected finite-horizon discounted return of the policy.π_θ

The way the algorithm works is by updating the policy parameters: (see Figure 9)

Policy gradient methods usually compute advantage function estimates based on a discounted return. The advantage is obtained subtracting the value function to the discounted sum of rewards:

The discounted return is the sum of all the rewards obtained during the in each timestep in the same episode, whereas the value function gives an estimate of the discount rewards from this point onward If we subtract them, we get the advantage estimate.

If the advantage estimate was positive, the actions resulted in better than average return, we increase the probability of selecting them again in the future in the same state. Else, it will do the other way around.

6.3 TRPO and PPO

A problem we encounter with Policy Gradients is that we may take a step too far from the previous policy and ruin our procedure. This problem was first tried to solve with the Trusted Region Policy Optimization (TRPO).

Equation: (see figure 10)

In order to avoid moving too far from the previous policy, the KL constraint was added: (see figure 11)

The use of this constraint sometimes leads to bad training behaviours. Finally, we arrive to

$$\mathcal{L}(\theta_k, \theta) = \mathop{\mathrm{E}}_{s,a\sim\pi_{\theta_k}}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a)\right]$$

Figure 10

$$D_{KL}(\theta||\theta_k) = \mathop{\mathrm{E}}_{s\sim\pi_{\theta_k}}\left[D_{KL}\left(\pi_\theta(\cdot|s)||\pi_{\theta_k}(\cdot|s)\right)\right].$$

Figure 11

$$\theta_{k+1} = \arg\max_\theta \mathop{\mathrm{E}}_{s,a\sim\pi_{\theta_k}}\left[L(s, a, \theta_k, \theta)\right]$$

Figure 12

the *state of the art*: Proximal Policy Optimization (PPO). There is a version of this algorithm which basically adds this KL constraint directly to the optimization objective. Just like TTPO, this approach is also trying to make really big improvements in the policy we are trying to optimize trying to avoid taking too big steps so the policy does not totally ruin.

There are two primary variants of PPO: PPO-Penalty and PPO-Clip.

PPO-Penalty approximately solves a KL-constrained update like TRPO, but penalizes the KL-divergence in the objective function instead of making it a hard constraint, and automatically adjusts the penalty coefficient over the course of training so that it's scaled appropriately.

PPO-Clip does not have a KL-divergence term in the objective and does not have a constraint at all. Instead relies on specialized clipping in the objective function to remove incentives for the new policy to get far from the old policy. This is the one we will develop.

Policy parameters update this way: (see figure 12)

where L is: (see figure 13)

6.4 Interpolations between Policy Optimization and Q-Learning

Moving to methods that involve an interpolation between Q-learning and Policy Optimization, there are two algorithms that should be taken into account.

First of all we have Deep Deterministic Policy Gradient. It uses off-policy data and the Bellman

$$L(s, a, \theta_k, \theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a),\ \ \mathrm{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon\right) A^{\pi_{\theta_k}}(s,a)\right)$$

Figure 13

$$a^*(s) = \arg\max_a Q^*(s,a).$$

Figure 14

$$Q^*(s,a) = \mathop{\mathrm{E}}_{s' \sim P}\left[r(s,a) + \gamma \max_{a'} Q^*(s',a')\right]$$

Figure 15

equation to learn the Q-function, and uses the Q-function to learn the policy. It can only be used for environments with continuous action spaces.

Just like in Q-Learning, if you know the optimal action-value function Q*(s,a), then in any given state, the optimal action a*(s) can be found by solving (see figure 14)

Bellman equation describing the optimal action-value function: (see figure 15)

We can set up a mean-squared Bellman error (MSBE) function, which tells us roughly how closely Q comes to satisfying the Bellman equation. Putting it all together, Q-learning in DDPG is performed by minimizing the following MSBE loss with stochastic gradient descent: (see figure 16)

Finally, it is to be mentioned an algorithm which optimizes a stochastic policy in a off-policy way, forming a bridge between stochastic policy optimization and DDPG-style approaches: Soft Actor Critic. A central feature of SAC is entropy regularization. The policy is trained to maximize a trade-off between expected return and entropy, a measure of randomness in the policy.

7 Agent exploration

When an agent arrives to a solution, a dilemma emerges. Should the agent keep doing what it has done or should it try something new? If the agent keeps doing what it has done, it is called **exploitation** whereas **exploration** is called when it tries something new. But how much can we exploit and explore? There is a problem in exploration which is that we don't know what the outcome will be, it could be better but it could also be worse.

If we take the reference in humans, before making the move they usually try to read reviews or ask friends to make a decision, in Reinforcement Learning on the other hand, it is not possible to do that, but there are some techniques that will help deciding the best solution.

$$L(\phi, \mathcal{D}) = \mathop{\mathrm{E}}_{(s,a,r,s',d)\sim\mathcal{D}}\left[\left(Q_\phi(s,a) - \left(r + \gamma(1-d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))\right)\right)^2\right]$$

Figure 16

7.1 Exploration vs. Exploitation

In the Reinforcement Learning setting, no one gives us some batch of data like in supervised learning. We are gathering data as we go, and the actions that we take affects the data that we see so sometimes it is worth to take different actions to get new data. If we keep what we have and we make the best decision given the current information, we will never know if there is any other better solution. That is why we talk about agent exploration instead of agent exploitation, because we need to gather more information to lead us to better decision in the future.

7.2 Regret in Reinforcement Learning - Notion of regret

It is logical to think that if we try something new and the solution is worse than the one we had before, we regret our decision of exploring. For example, if the new restaurant we try is bad we regret going there and we consider the amount we paid as a complete loss.

As we keep doing bad decisions the amount of losses grows as well as the level of regret, so one should think, can we keep the amount of losses and the level of regret to the minimum? And the answer is yes, at least in Reinforcement Learning.

Some of the methods to do that are Greedy and Epsilon Greedy exploration methods. This methods are fairly easy to understand and implement but they suffer from having a sub-optimal regret on the solution which grows linearly by the time.

7.3 Greedy

The Greedy method will lock on one action that happened but it is not really the optimal

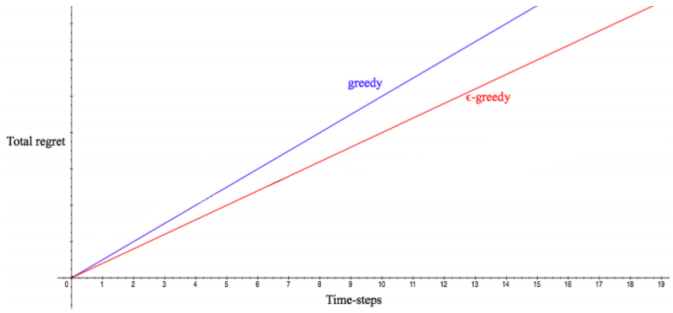

Figure 17: A difference on regret between Greedy and Epsilon Greedy

action, so Greedy will keep exploiting this action while ignoring others that might be better.

7.4 Epsilon Greedy

The epsilon greedy ($\varepsilon - greedy$), where $0<\varepsilon<1$ is a parameter controlling how much exploration or exploitation we have to do. The agent chooses exploitation with the probability of $1 - \varepsilon$, choosing the action that it believes has the best long-term effect. In the other hand, the agent chooses exploration with the probability of ε, and the action chosen is random. ε is usually a fixed parameter but can be adjusted to make the agent explore progressively less (to keep the level of regret minimized) or adaptively based on heuristics.

Even if the Epsilon Greedy looks better and maintains a better minimization of regret then the greedy method, it is not as optimal as we want it to be, as we can see in in the Figure 17.

So we can not do anything to minimize more the regret? Yes, we can, taking different approaches.

7.4.1 Optimism in Face of Uncertainty approachment

This approachm tries to enhance the exploration minimizing the total regret. Let's suppose we have three slots machine with different probability distribution that we don't know and after some trials we got this result (see Figure 18).

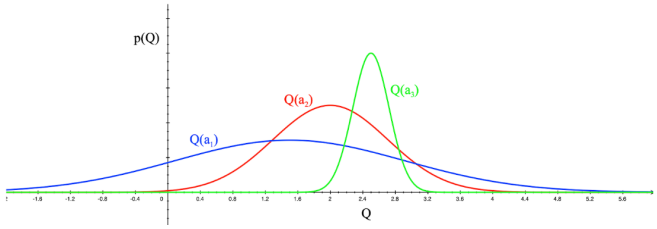

Figure 18: Three slots machine probability distribution

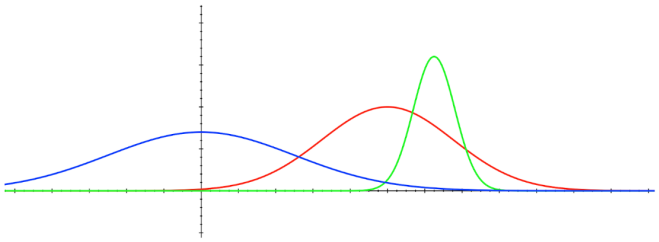

Figure 19: Three slots machine probability distribution after choosing one

The Q(a1) has the largest interval [-1.8, 5.2], so we take this because the Optimism in Face of Uncertainty principle says that we choose the action that present a higher reward than the others even if it has a higher uncertainity. Now we take the action a1 and the distributions becomes like this (see Figure 19).

Now we know the blue distribution has lesser maximum than the other two so we are less uncertain about it and the next action will be to choose any other, for example, the red one.
This is the principle but how con we implement this in reality?

7.5 Upper Confidence Bound (UCB)

It is said that the exploration is needed because there always will be uncertainty about the accuracy of the action-value estimates. It could be better to select non-greedy actions to make them more optimal, if we could estimate some upper confidence on what the value of an action could be we would optimize the action a lot. This is where $U_t(a)$ enters in action, $U_t(a)$ is a high probability upper confidence on the value of an action could be and then we could pick the action with the highest upper confidence value.
To do this, we add a term called upper confidence

t	$N_t(a)$	$U_t(a)$
100	1	2
1000	1	2.45
1001	2	1.73

Figure 20: Chart of UCB results

bound to $U_t(a)$ for each action and then select the action that has the best result.

$$a_t = argmax_{a \epsilon A}(Q_t(a) + U_t(a))$$

Obviously, Ut(a) is not constant, as time goes by we become more and more certain of what to expect of Q(a), so Ut(a) should shrink as we become more confident of Q(a).
Having the number of trials (let's call it t) and the number of times action (a) was selected (call it $N_t(a)$) we can define Ut(a) as:

$$U_t(a) = \sqrt{\frac{2 \log t}{N_t(a)}}$$

As t increases the numerator slowly increases while $N_t(a)$ increases faster so at some point $U_t(a)$ decreases sharply.
Here is an example (see Figure 20):

At the point where t=1000 $U_t(a)$ is 2.45 but adding an unit decreases the value of Ut(a) to 1.73.
With this definition, we have that:

$$a_t = argmax_{a \epsilon A}(Q_t(a) + \sqrt{\frac{2 \log t}{N_t(a)}})$$

7.6 Summary

To summary all this agent exploration methods, we can say that:
Greedy methods: We initialize the values of actions optimistically assuming everything is good until proven otherwise, and when that happen, we suppress the action value to its realistic value. This repeats for every and each action value.
Random exploration: $\varepsilon - greedy$ algorithm does this well, if we are lucky tuning it right. The problem is that if we get this wrong getting to the optimal action in the end is a very hard task.
Optimism in the face of uncertainty: This is not a safe exploration, in the real industry it is recommended not to use this. Estimating how much we don't know about the value of an action and using that to guide us towards actions that have the most potential to be good could guide us into a wrong and risky path if we are wrong.
UCB: Without any knowledge about the problem systematically performs really well. This approach of explore vs exploit guarantees to have a logarithmic regret.

7.7 Self-Play

Self-Play is a term we use when an agent only players against itself, and is one of the most advanced techniques in AI for the last decade. An example of this is AlphaZero, the algorithm that learned to play Go, chess and shogi only playing against itself, that is, AlphaZero's agent did not look at examples of well-played games or good moves nor did it know heuristics that could help it evaluate a position (for example giving a better award for going to the left than to the right). It simply played games against itself millions of times. If only played against itself, where did new information come from? that is where the agent exploration comes.
The great benefit of self-play is that you don't need to give direct supervision, the agent uses another agent (a clone of itself) to recover new information to maximize its own objectives. AlphaZero's fast success comes from all the game episodes that could simulate the paralleled algorithm, learning much faster than a true human.

7.7.1 Competitive self-play

As OpenAI has stated, the self-play will be a core part of powerful AI systems in the future.
Alternatively to standard algorithms of Reinforcement Learning that we have talked before, the self-played AIs try new phyisical skills likes ducking, catching, tackling... without explicity designing an environment with these skills in mind. This allows adaptation of differently designed AIs to alternative environments, as we have seen in AlphaZero.

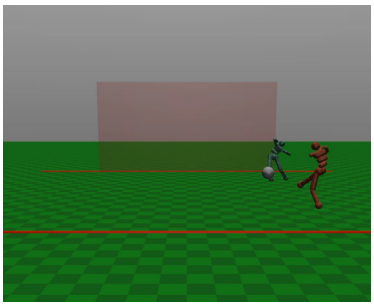

Figure 21: Self-play footbal agents

All of this is very interesting but how does this work? We will explain this with the example of two agents playing some penalty shots, sumo and standing against wind.
Firstly, agents are trained to receive high rewards for behaivours like standing and moving forward which is better called as "exploration", this are eventually annealed to zero in favor of just having a reward if they win and loose. Despite this rewards being so simple, the agents learn to make different moves to kick the ball, catch the ball, faking the move, etc... Each agent's neural network policy is trained independently with the algorithm Proximal Policy Optimization.

For example, the goalkeeper starts (see Figure 21) imitating the position of the shooter to have more probabilities to stop the ball from entering the goal and alternitavely the shooter learns to kick the ball to throw it with more velocity.
To show how a trained agent can emerge complex behaviours in a different environment, the OpenAI team took a previous work for a walking humanoid and removed the term of velocity, added the negative euclidean distance from the center of the ring and took this as a dense exploration reward for sumo agents. Then they slowly annealed it to zero so the agents learned pushing the other player out for the remaining training iterations for the competition reward. As we can see (see Figure 22), the agents starts acting almost like humans.
The hard part of this complex work is the ingenuity needed from the human designers because the agents' behaviours will be bounded in complexity by the problems that the human designer can pose for them. Developing agents

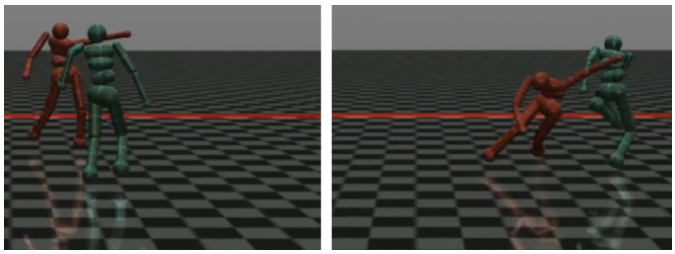

Figure 22: Self-play sumo agents

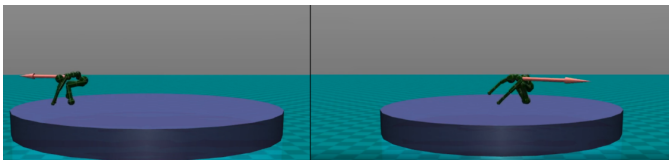

Figure 23: Wind VS Self-play trained agent

through thousands and thousands of iterations of matches against better version of themselves we can create AI systems that improve their behavours. This can be seen in Dota 2 project, where self-play let the team create a Reinforcement Learning agent that could beat top human players.

7.7.2 Difference between classical RL and self-play

A distinct characteristic of the self-play trained agents is transfer learning, with the application of skills learned in one setting to succeed in another one. For example, taking the sumo agents learn to stand and not fall while being perturbed by a unknown variable "wind" while agents trained to walk using classical Reinforcement Learning would fall immediately.
The sumo agent learn to crouch and support their weight without falling (see Figure 23), but the classically trained agents (see Figure 24) fall immediately.

8 Bibliography

- Artificial Intelligence: A Modern Approach (3rd Edition) - Peter Norvig, Stuart J. Russell
- Spinning Up in Deep RL - OpenAI
- Policy Gradient Methods for Reinforcement Learning with Function Approximation - S.Sutton

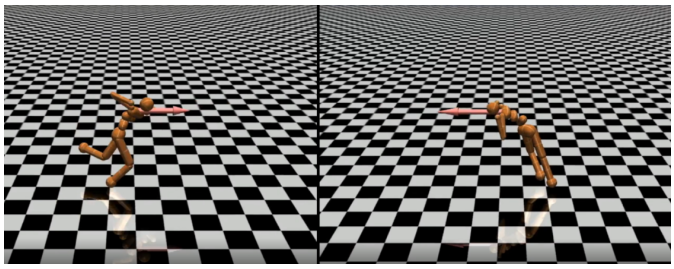

Figure 24: Wind VS Classically trained agent

- Proximal Policy Optimization Algorithms - Schulman

References

[1] *DeepMinds' AlphaZero.*
https://deepmind.com/blog/article/alphazero-shedding-new-light-grand-games-chess-shogi-and-go

[2] *OpenAI take on reinforcement learning.*
https://spinningup.openai.com/en/latest/user/introduction.html

[3] *Wikipedia RL article.*
https://en.wikipedia.org/wiki/Reinforcement$_{l}earning$

[4] *Wikipedia Deep RL article.*
https://en.wikipedia.org/wiki/Reinforcement$_{l}earning$

[5] *RL for NLP, the hardships of RL in the real world.*
https://en.wikipedia.org/wiki/Reinforcement$_{l}earning$