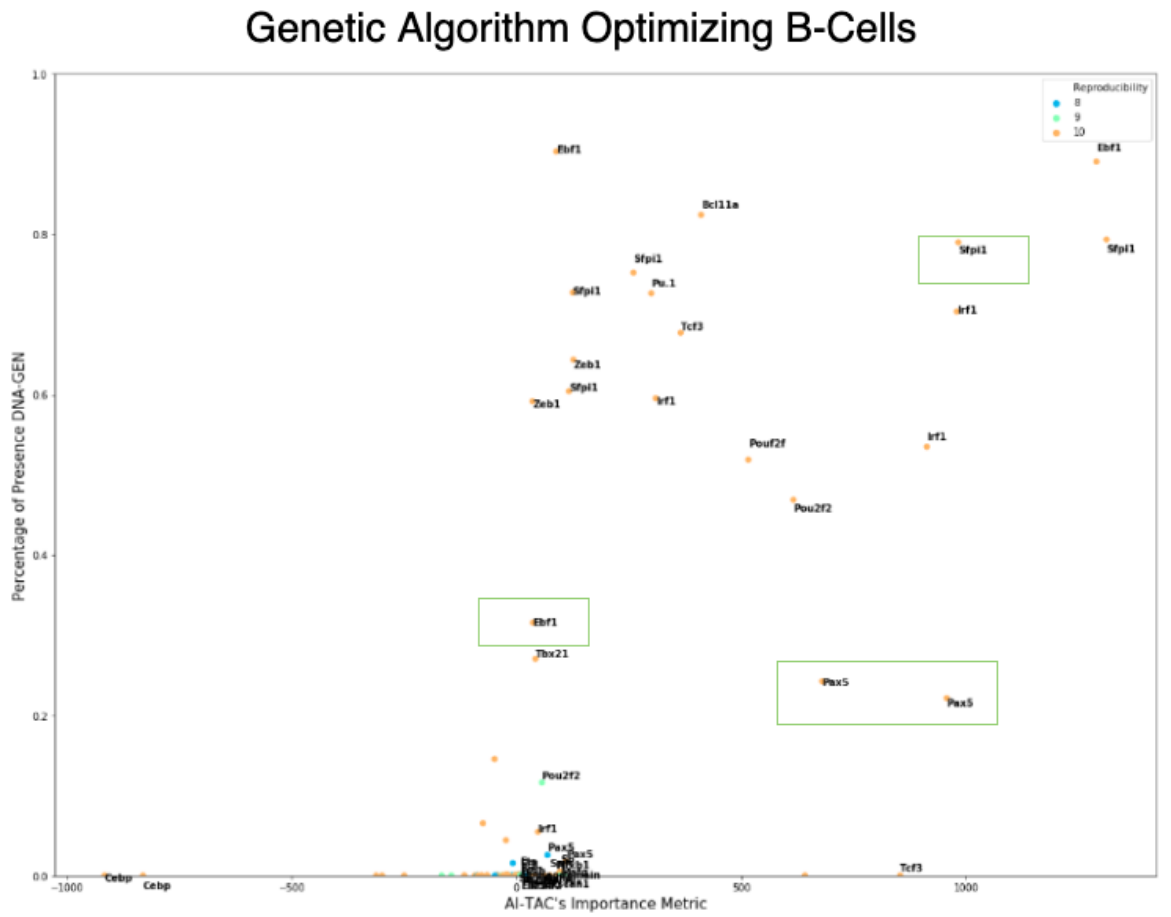
Asynchronous Advantage Actor-Critic (A3C) to generate DNA sequences

A. Introduction

1. Description

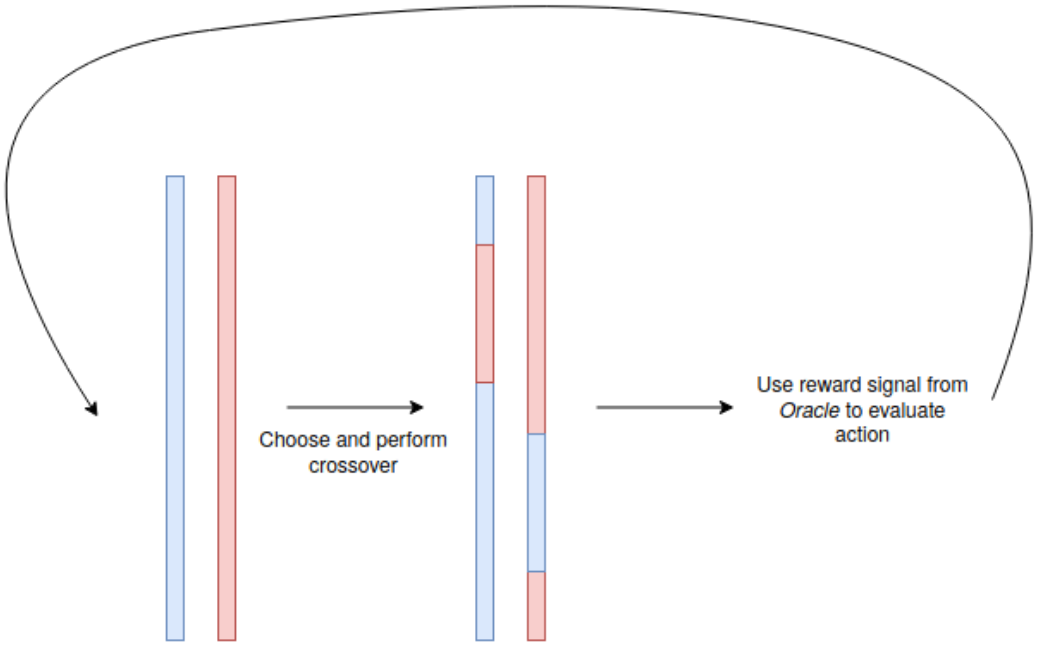
Generating relevant and unseen DNA sequences is a necessity to understand and interpret our *Oracle* network. In this document, we suggest a new approach among the DNA sequences generation methods.

The idea comes from the Genetic algorithm (GA), the relatively good performance of this model gave us the insight that repeated crossover is a good path to DNA sequence optimization. The arbitrary part (crossover defined beforehand) is frustrating though. There is a lot of arbitrary of random choices during the the process of the Genetic Algorithm, this is an issue because we can't guarantee that we are getting closer to the optimum with the number of iteration and that the model is improving with time, one of the consequences of this is that the model is hard to compare with papers using reinforcement algorithm [Google] or sampling based technique [Listgarden]. On the other side the fact that the GA is yielding very convincing results in term of motifs even in difficult optimization problems (Uncertain oracle and space bigger that the one tested in most of the concurrent papers) is encouraging and give us the intuition that it would be interesting to exploit techniques relying on crossover. We want our model to make its choices based on the previous ones and to learn a decision making model that we could use for interpretation secondly. That's why we thought that using Reinforcement Learning framework could perfectly fits our goals.



RL framework has shown being very efficient regarding decision making problem. With a robust definition of the framework components, we have the strong feeling this thrilling optimization matter can be addressed using Reinforcement Learning. Especially, one of the main challenge of this set of task is to navigate on the thin line between "exploration" and "exploitation" in order to find both diverse and and efficient solutions. Recent improvements on Policy Gradient (*PG*) and in particular Actor-Critic (*AC*) algorithm demonstrate the latter outperforms value-based methods. The strong advantage of PG is its ability to learn stochastic policy and to sample from the learned probability distribution to perform its actions (*On-Policy algorithm*). This way, the panel of good possibilities to be chosen will not be affected by the algorithm. Thus and most importantly, the model will not converge to a single high rated sequence. More on exploitation-exploration trade-off is discussed in the algorithm description.

In the algorithm_description document I give a detailed description of the algorithm, but first, let's explain the outlines. This algorithm aims at optimizing a random DNA sequence so that it maximizes the *Oracle* return. So at each time step, we modify our sequence and we call the *Oracle* for feedback. These optimization's approaches differ from generation's ones (*i.e generate one nucleotide at each iteration*). You will find more about that below (Method Justification) but we want to show that crossover based ans thus optimization approach is well fitted here. As you may guess, the DNA sequence will be modified by crossovers, eventually our agent will learn to perform good crossover circumstantial (*i.e based on the sequences' couple*). As it is a continuous task, the sequence optimization will stop once the reward stagnate for an arbitrarily defined duration or reach a target value.

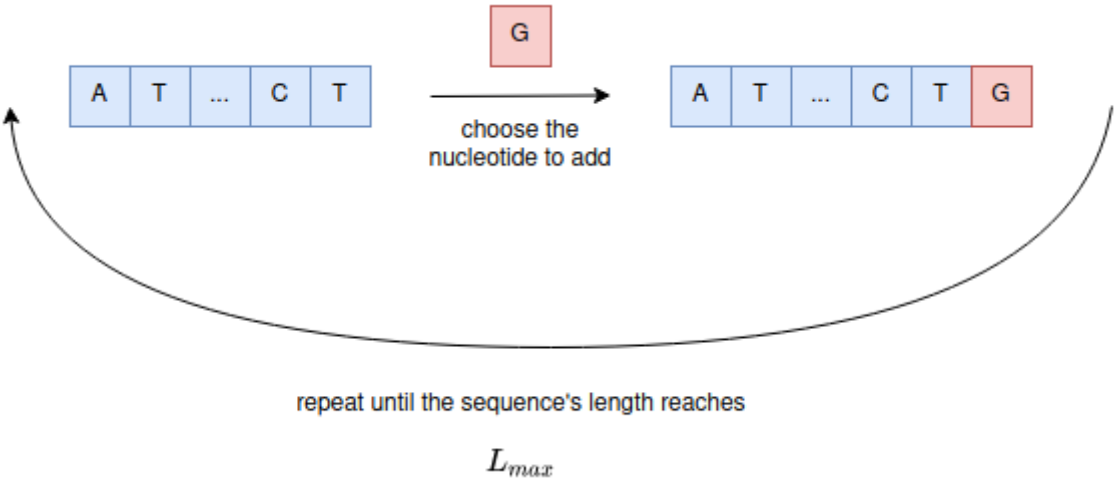


2. Related work

Model-based reinforcement learning for biological sequence design

https://openreview.net/pdf?id=HklxbgBKvr

In September 2019, Google suggest a RL-based method to generate DNA. This approach is a generation's one, at each time step the model will choose a nucleotide (*i.e {A, T, G, C}*) to stack on top of the current branch. So based on the current incomplete sequence, the model will try to find the right nucleotide to set afterward.

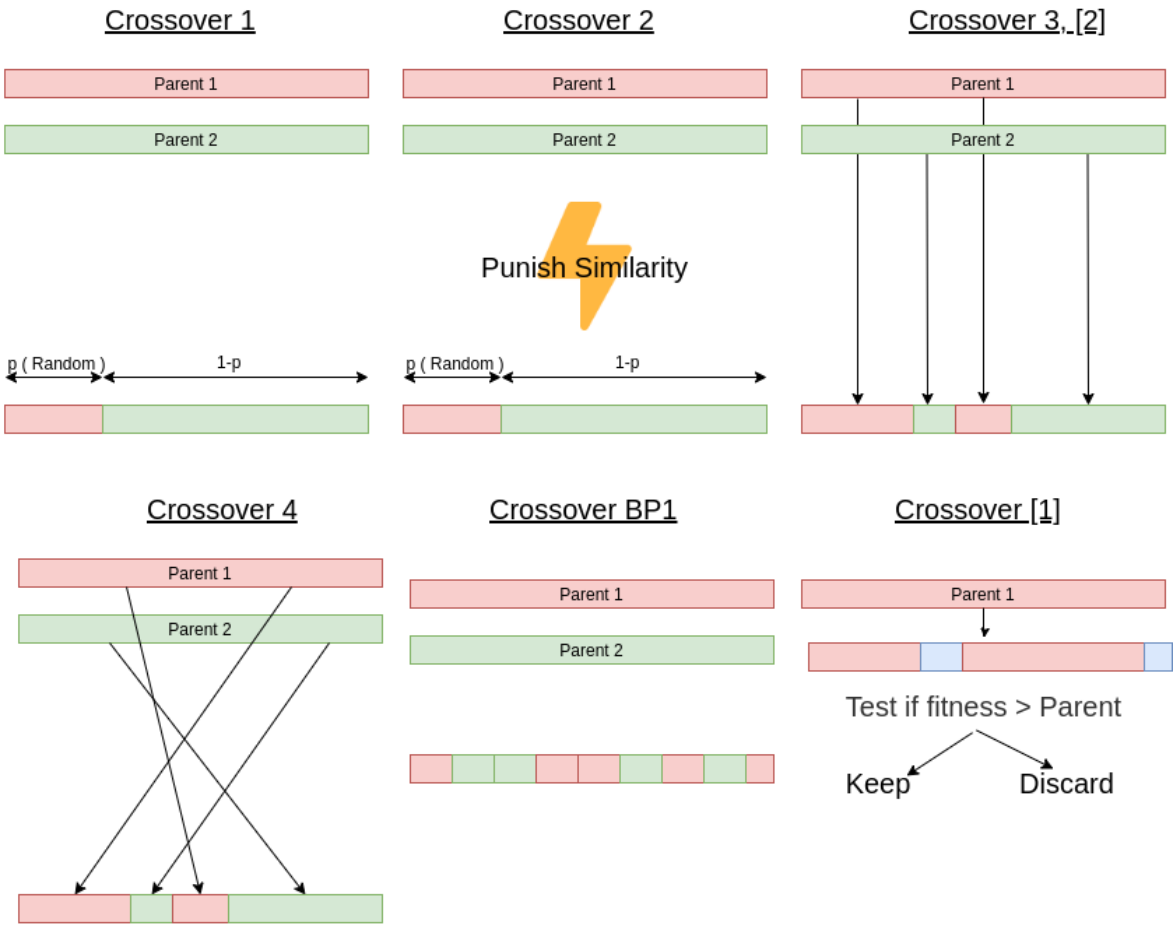


3. Method justification

In this section, we will justify our choices and explain why we think our approach is efficient. At first, we want to clearly emphasize on the fact that recent RL has proven to be able to learn complexe structure, largely driven by the neural network paradigm. (Universal approximation theorem)

4. Comparison with related techniques

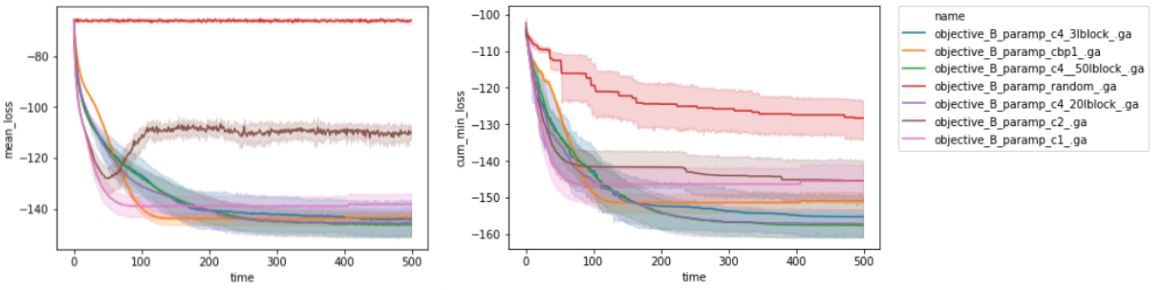
We both have the strong conviction that manipulate our sequences bloc-wise is a very efficient way compare to nucleotide-wise. Indeed, DNA interpret information not as whole but sequentially. This intuition seems actually confirmed by practical results : when we run the GA with different crossover size we observer consistently a better performance of the 19 bp block crossover.



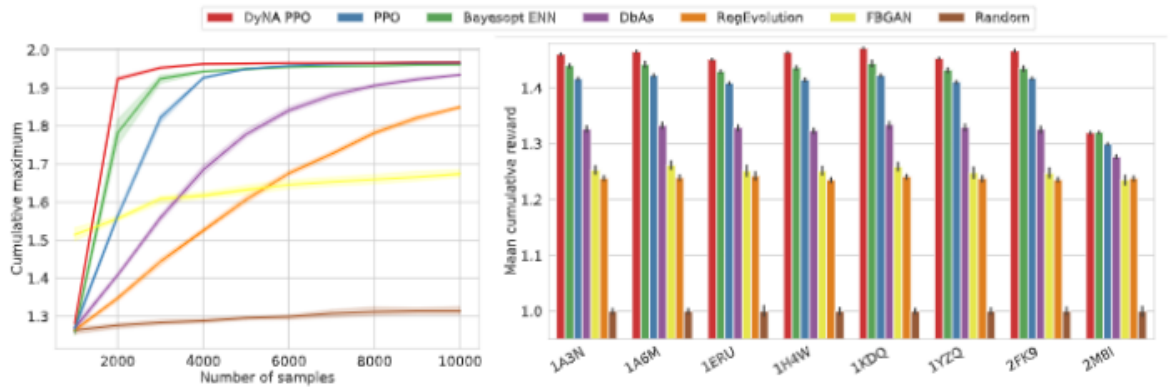
[1] Human 5′ UTR design and variant effect prediction from a massively parallel translation assay

[2] Model-Based Reinforcement Learning for Biological sequence design

GA Crossover Comparison



Hence our will to stand out from Google approach. We believe out Genetic Algorithm has relatively good result thanks to his unit-wise modification. As mentioned, we want to add artificial decision making to this system. We divert from the GA approach in a sense because we are not using a whole population but a single DNA sequence. This choice is driven by multiple reasons. The first and most obvious one is to relieve our RL model from a lot of information to handle. The agent will make assumption taking 2 sequences whether than a N-sized population. Also, in terms of *Oracle*'s call, each of them will be used for the following decision. This bootstrapping method gains the maximum potential of our *Oracle*. As shown on the figure below the number of sequence evaluated by the Oracle during the generation process is the core comparison of the papers we are comparing with. Thus, optimizing the sequences with a minimal number of step and using only two evaluations by step is a key asset to increase the credibility of our technique.



	DyNA PPO	PPO	BO-GP	DbAs	RegEvol	FBGAN	Random
Cumulative maximum	**6.4**	5.8	5.0	3.7	3.7	2.2	1.3
Fraction optima found	**6.8**	5.6	5.4	3.3	3.3	2.5	1.0
Mean hamming distance	**5.6**	5.4	4.0	2.5	1.0	2.5	7.0

5. Interpretation advantages

Concerning interpretability, our algorithm will try to model the cross over action with respect to *Oracle* evaluation. From a biological point of view, a real interpretation on crossover effect leading to new assumptions is a possible outcome. On top of that, having a natural link (*i.e the crossovers*) between our model and biology could make the *Oracle* interpretation a lot easier. Crossover based technique have a strong advantage regarding interpretability has it is linked to our current vision of the activity regulation mechanisms, by "forcing" the generated sequences to rely on motifs / large blocks to optimize the objective we increase the chance to find interesting motif but also importantly decrease the chances to build adversarial examples : in substance the generated sequences not only have to be a set of nucleotide that maximize or objective but also to be composed by subset of nucleotides that locally have an impact.

B. Algorithm description

1. Markovian Decision process

For this algorithm, we are going to optimize a DNA sequence iteratively.

A Markov Decision Process (MDP) is a 6-tuple (S, A, P, R, γ, D), where S is the state space of the process,A is the action space, P is a Markovian transition model $P(s'|s, a)$ denotes the probability of a transition to state s' when taking action a in state s, R is a reward function $R(s, a)$ is the expected reward for taking action a in state s, $\gamma \in (0, 1)$ is a discount factor for future rewards, and D is the initial state distribution. A deterministic policy π for an MDP is a mapping $\pi : S \to A$ from states to actions; $\pi(s)$ denotes the action choice in state s.

State

Our state s_t is going to be the current sequence to be optimized at time t, let's name it S_{opt}. The latter has a fixed length named L_{opt}. To perform modifications, the agent will use crossover and thus we'll need a **unmutable** DNA sequence called S_{co} with $L_{co} \geqslant L_{opt}$. This sequence could be define randomly or maybe in a more judicious manner, such as the concatenation of all

combination of A, T, G, C in a range of k (*e.g k from 1 to 5*). In this vision, S_{co} would not be in the state s_t as the modification won't affect the sequence.

Alternatively, the state s_t could be seen as the concatenation of both S_{opt} and S_{co} with $L_{opt} = L_{co}$. That would allow multiple enhancements for larger computation cost. First crossovers could modify S_{co} permitting a wider modification range. Moreover, that makes us free of choosing arbitrarily a S_{co}.

Action and Transition function

The action is defined in a **discrete 3-dimensional** space such as $a_t = (l, i_{opt}, i_{co})$, where l is the crossover length, i_{opt} and i_{co} are respectively the crossover starting point (*i.e the index*) for S_{opt} and S_{co}. In that way we can perform all types of crossover. The transition function $P(s'|s, a)$ is then purely deterministic. I describe the function approximation architecture in section 3.

Reward

The reward function is handle by a ***Oracle***. This neural network takes a DNA sequence as input and output its estimation of how good is it. So $r_t = Oracle(s_t)$ with $r_t \in [0, 1]$. However, I have the strong feeling that a relative reward will be more meaningful. That is why , I'll also use the difference between 2 consecutive scores as reward, $r_t = Oracle(s_t) - Oracle(s_{t-1})$. Accordingly, now the cross over is rewarded if it has enhanced the sequence's score (given by the ***Oracle***) and vice versa.

2. Algorithm

1. initialize policy parameter θ_0, value function parameter w_0, and both step size α_a, α_c . Let γ be our discount factor, m be our population size and r_{target} be our score target
2. Initialize s_0 (*i.e a random series of shape* $(L_{opt},)$)
3. for $t = 0, 1, 2, 3...$T
    1. the actor selects action a_t, sampling from $\hat{\pi}(.|s_t)$.
    2. perform a_t, observe s_{t+1}.
    3. compute r_{t+1} with the *Oracle* such that $r_t = Oracle(s_t)$.
    4. if $r_t > r_{target}$: *break from loop*
    5. Using critic, compute both $\hat{V}(s_t)$ and $\hat{V}(s_{t+1})$.
    6. Compute the TD target: $\delta_t = r_{t+1} + \gamma * \hat{V}(s_{t+1})$ and the TD error: $\delta_e = \delta_t - \hat{V}(s_t)$.
    7. update actor minimizing $L_a = -log(\hat{\pi}(a_t|s_t)) * \delta_e$. (*Entropy penalty is discussed in 5.*)
        1. $\theta_{t+1} = \theta_t + \alpha_a * \frac{\partial L_a}{\partial \theta_t}$.

       update critic minimizing $L_c = (\delta_t - \hat{V}(s_t))^2$.
        1. $w_{t+1} = w_t + \alpha_c * \frac{\partial L_c}{\partial w_t}$.
    8. $s_t = s_{t+1}$.

3. Function Approximation

In this section we will talk about both **critic** and **actor** networks.

Actor

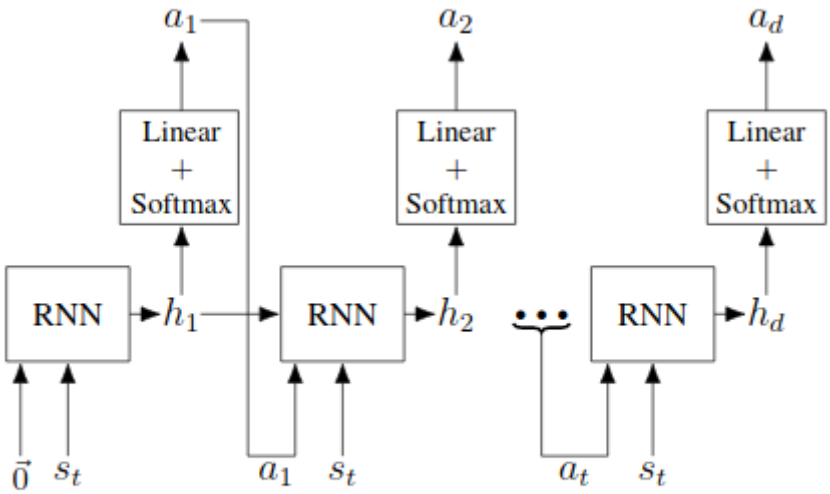
Autoregressive

In this section, we will assume $|a_1| = \cdots = |a_d| = L_{opt}$ with $d = 3$. In the section 5, we talk about dealing with different sub-action's dimension size.

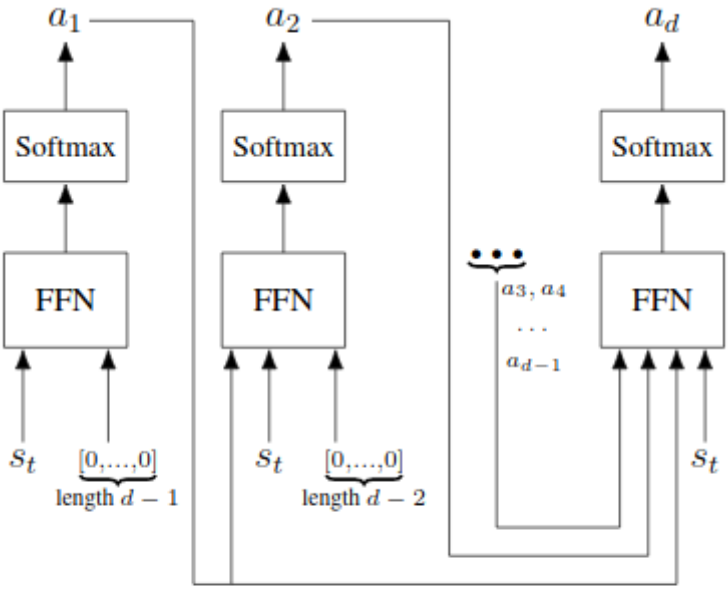
The actor needs to generate a probability distribution over the set of actions $\mathcal{A}$. In a 1-dimensional action space setting, the classical approach is to use a SoftMax output of dimension $|\mathcal{A}|$. In our specific case, $\mathcal{A} \in \mathbb{N}^3$ with $a_i < L_{opt}\ \forall\ a_i\ in\ \mathcal{A}$. That implies, generating a multivariate probability distribution over the action space. The stake of the model to use is real here for computational efficiency purpose.

Indeed, when dealing with multidimensional discrete action space, the number of possible action is exponential with respect to the number of dimension. Let's recall that our action is defined by the 3-tuple $a_t = (l, i_{opt}, i_{co})$, let's imagine we are going perform crossover on S_{opt} with $L_{opt} = 50$. Each of l, i_{opt}, i_{co} can take a positive discrete value inferior to 50. Since using all possible combination with order and repetition (and thus 50^3 different actions) would force the neural network to handle a 125000 output layer shape, I'm quite unwilling to go this way.

Recent work have shown autoregressive model can be used to sample sequentially $A_i = (a_1, a_2, \ldots, a_i)$ from our policy. In that way sampling from model only requires summing over $O(3 * L_{opt})$ effort whereas the aforementioned model requires $O((L_{opt})^3)$.



(a) The RNN architecture. To generate a_i, we input s_t and a_{i-1} into the RNN and then pass the resulting hidden state h_i through a linear layer and a softmax to generate a distribution, from which we sample a_i.

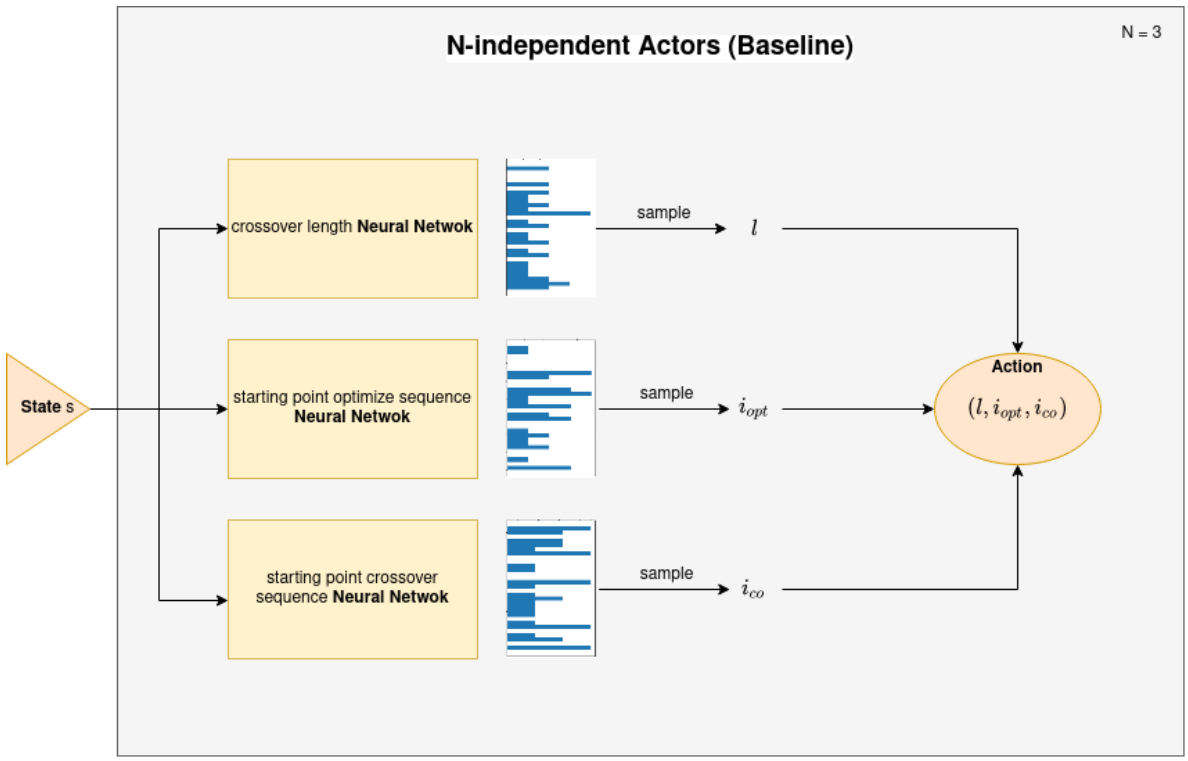


(b) The MMDP architecture. To generate a_i, we input s_t and $a_1, a_2, \ldots, a_{i-1}$ into a FFN. The output is passed through a softmax layer, providing a distribution from which we sample a_i. Since the input size of the FFN is fixed, when generating a_i, constants 0 serve as placeholders for $a_{i+1}, \ldots, a_{d-1}$ in the input to the FFN.

As they are quite easy to implement with nowadays deep learning framework, the goal here would be to test both RNN and MMDP as autoregressive model for the **actor**.

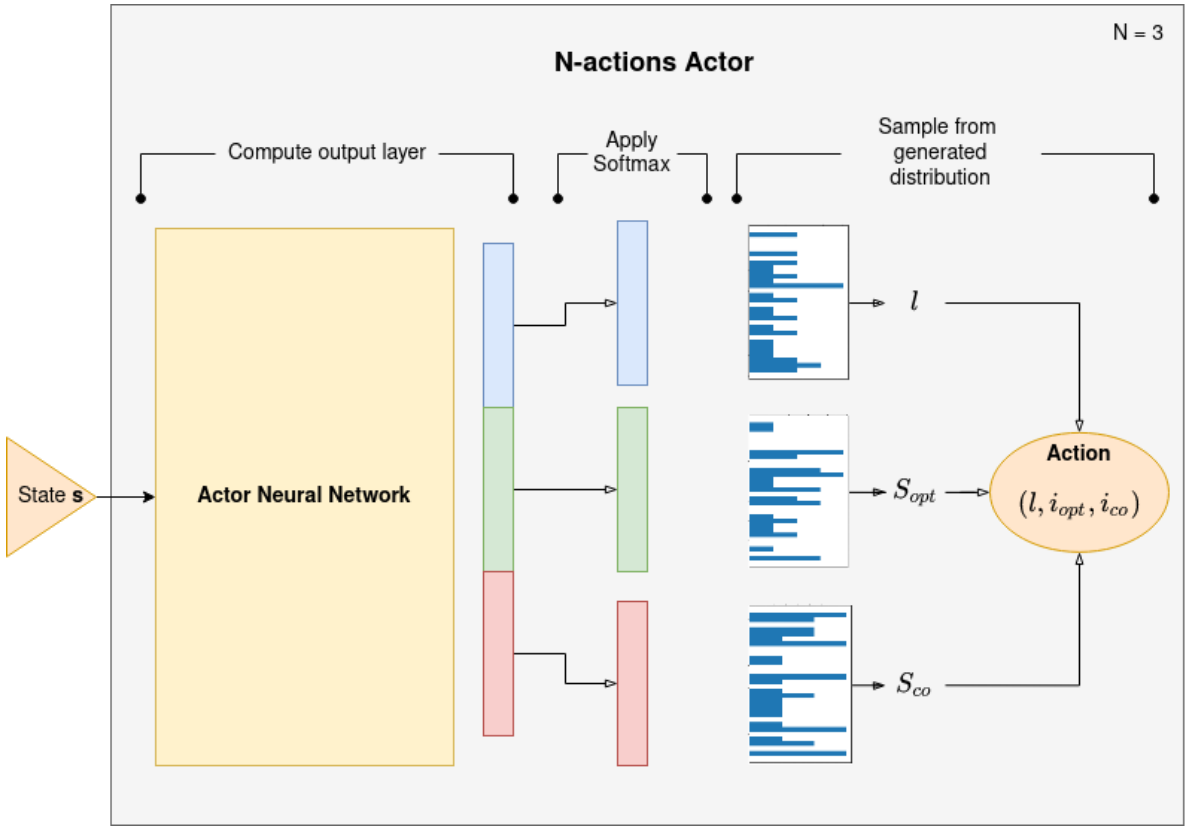
N independent Actors (Baseline)

Since we want to compare our autoregressive model with a baseline, I choose to implement a trivial actor composed of N (=3) actors each of them powered by its personal neural network. Thus each of them estimates a probability distribution over 1 action dimension (crossover length, and both starting points). They really are independent in the sense that no one is affected by one another in the learning process, they all use the same reward to update their weights independently. Of course, the overall result should be poor since there is no learning between the different action dimensions. Indeed choosing the crossover length regardless the starting point is intuitively a bad way of doing. Though, this approach will serve as a baseline.



N actions Actor

Another way of predicting multivariate distribution is to use a single neural network with multiple independent softmax on the last layer. Each piece of action is now generated with regards to the 2 other ones. This architecture could be interesting to compare with the autoregressive one.

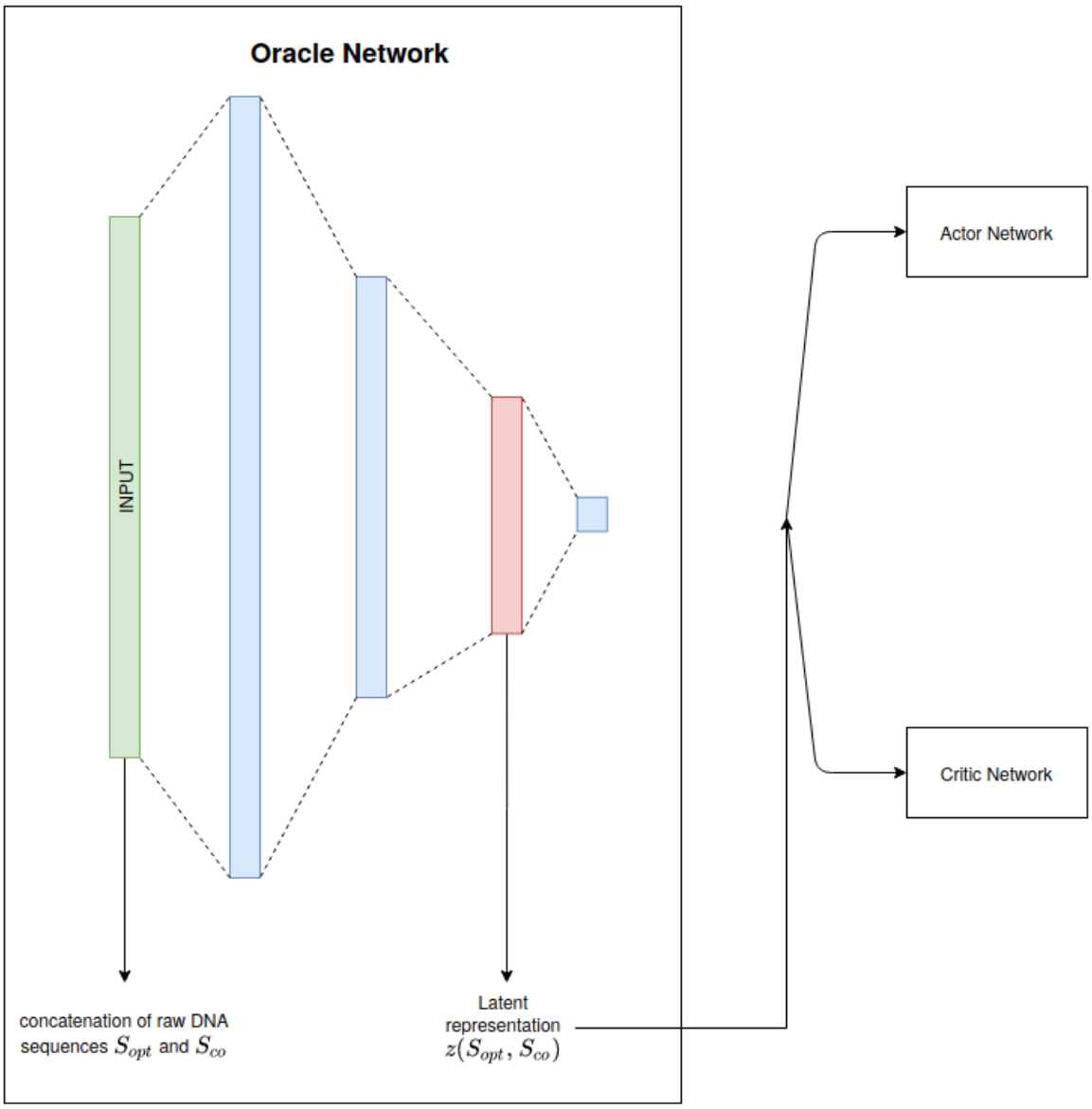


Critic

It is a much more easier network to handle, since it only estimates the value of a particular state s_t. A simple neural network taking the state s_t as input and outputting a scalar will make the job.

4. Feature engineering

Dealing with large state space can slow learning, and more precisely generalization over state. One way to manage this difficulty is to learn good latent representations to construct the states with meaningful information. The *Oracle* has been trained to grasp the DNA architecture, by taking a latent space of this network we could find a better representation of our state.



5. Exploration and Entropy

Good exploration method is a important point with large action space, the entropy penalty is quite efficient and easily scalable to our actor network. The entropy is a measure of uncertainty within a certain probabilistic distribution, it is the average “element of surprise” or amount of information when drawing from the probability distribution. When the agent is learning its policy and an action returns a positive reward for a state, it might happen that the agent will always use this action in the future because it knows it produced *some* positive reward. There could exist another action that yields a much higher reward, but the agent will never try it because it will just

exploit what it has already learned. This means the agent can get stuck in a local optimum because of not exploring the behavior of other actions and never finding the global optimum. This is where entropy comes handy: we can use entropy to encourage exploration and avoid getting stuck in local optima.

Entropy

To abbreviate notations, we write $p_\theta(a)$ for $\pi_\theta(a|s_t)$ and a_i for $(a_1, a_2, .., a_i)$. We consider auto-regressive models whereby the sample components $a_i,\ i = 1, 2, .., d$ are sequentially generated, with $d = 2$ in our case. In particular, after obtaining $a_1, a_2, \ldots, a_{i-1}$, we will generate $a_i \in \mathcal{A}_i$ from some parameterized distribution $p_\theta(. |a_{i-1})$ defined over the one-dimensional set $\mathcal{A}_i$. After generating the distribution $p_\theta(. |a_{i-1}) \forall\ i$ and sample the action component $a_1, a_2, .., a_d$ sequentially, we then define $p_\theta(a) = \prod_{i=1}^{d} p_\theta(a_i|a_{i-1})$.

$$H_{\theta_t} = -\sum_{a_i \in \mathcal{A}} p_\theta(a) log(p_\theta(a))$$

$$H_{\theta_t} = -\mathbb{E}_{A \sim p_\theta}[log\, p_\theta(A)]$$

$$H_{\theta_t} = -\sum_{i=1,..,d} \mathbb{E}_{A \sim p_\theta}[log\, p_\theta(A_i|A_{i-1})]$$

Crude unbiased estimator

During training within an episode, for each state st, the policy generates an action a. We refer to this generated action as the episodic sample. A crude approximation of the entropy bonus is:

$$H_\theta^{crude} = -log\, p_\theta(a) = -\sum_{i=1}^{d} p_\theta(a_i|a_{i-1})$$

This approximation is an unbiased estimate of H_θ but its variance is likely to be large. To reduce the variance, we can generate multiple action samples when in s_t and average the log action probabilities over the samples. However, generating a large number of samples is costly, especially when each sample is generated from a neural network, since each sample requires one additional forward pass.

Smoothed Estimator

This section proposes an alternative unbiased estimator for Hθ which only requires the one episodic sample and accounts for the entropy along each dimension of the action space:

$$\tilde{H}_\theta(a) := -\sum_{i=1}^{d} \sum_{\alpha \in \mathcal{A}_i} p_\theta(\alpha|a_{i-1})\, log\, p_\theta(\alpha|a_{i-1})$$

$$\tilde{H}_\theta(a) = \sum_{i=1}^{d} H_\theta^{(i)}(a_{i-1})$$

with

$$H_\theta^{(i)}(a_{i-1}) := -\sum_{\alpha \in \mathcal{A}_i} p_\theta(\alpha|a_{i-1})\, log\, p_\theta(\alpha|a_{i-1})$$

which is the entropy of $\mathcal{A}_i$ conditioned on a_{i-1}. This estimate of the entropy bonus is computationally efficient since for each dimension i, we would need to obtain $p_\theta(\cdot|a_{i-1})$, its log and gradient anyway during training. We refer to this approximation as the smoothed entropy.

Gradient Estimator

To make a lighter document, I will not defined mathematically the gradient of the Entropy. The aforesaid mathematical aspect gives good insights on the entropy choice whereas the gradient formula here is less relevant. You can find the whole definition in this paper.

6. Differences in each action component dimension

We might want to restrict the crossover size and thus L_{opt} to a certain range (*e.g from 15 to 20*). This would improve learning drastically without restraining too much the learning spectrum. Indeed, a 1-sized cross over is really not the vision here nor a 40-sized which would have a substantial effect to the DNA sequences.

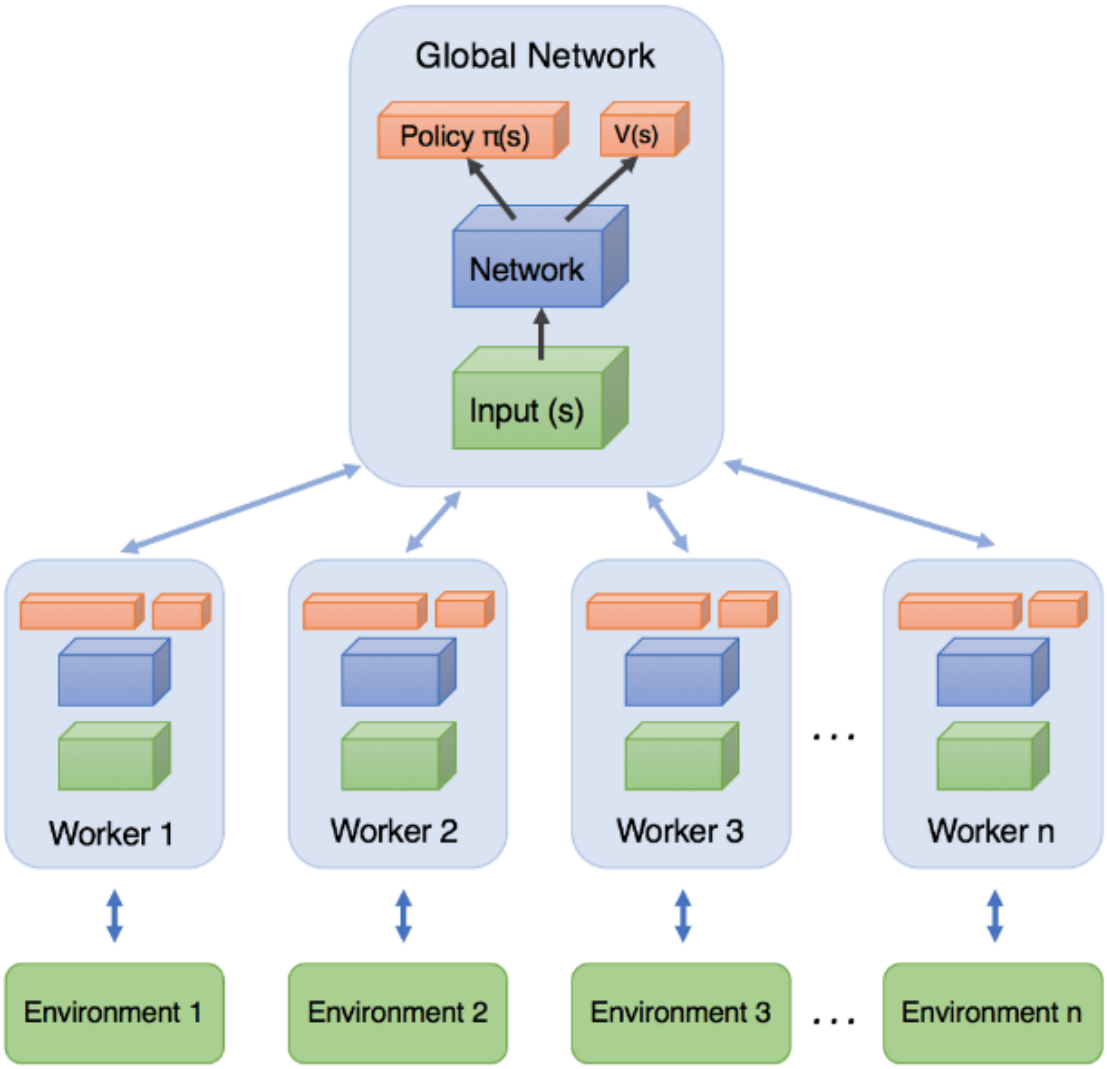
The issue is that, as we are going to use a autoregressive model (*e.g LSTM network*), the output layer size will be set. The idea behind using inconsequential actions has been largly used in the RL community recently (*e.g AlphaGo*). So we are going to proceed using this 2 rules:

- Not selectable actions are tremendously penalized
- Not selectable actions have no effect on the state

7. Asynchronous

For a better understanding, the algorithm describes in 2. does not mention the asynchronous aspect. A3C consists of **multiple independent agents** (networks) with their own weights, who interact with a different copy of the environment in parallel. Thus, they can explore a bigger part of the state-action space in much less time. The agents (or workers) are trained in parallel and update periodically a global network, which holds shared parameters. The updates are not happening simultaneously and that's where the asynchronous comes from. After each update, the agents resets their parameters to those of the global network and continue their independent exploration and training for n steps until they update themselves again.

We see that the information flows not only from the agents to the global network but also between agents as each agent resets his weights by the global network, which has the information of all the other agents.



C. Result (In-work part!)

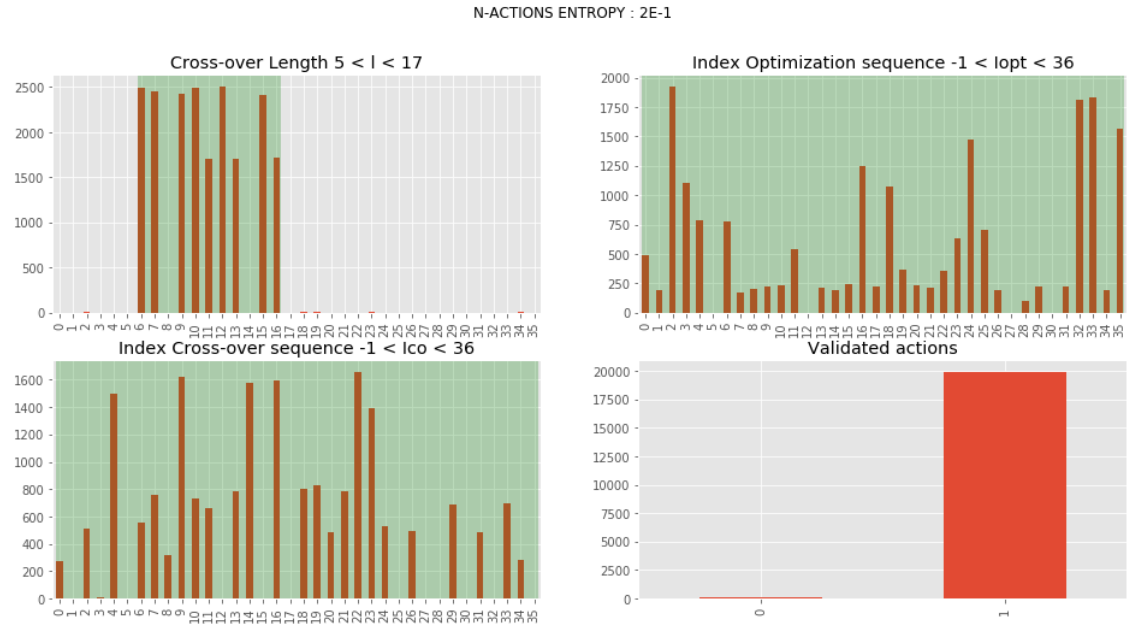
By discussing with specialists, we set the possible number of actions (*i.e crossovers*) to 150, meaning that the agent has 150 crossovers to improve as much as it can the sequence before the episode ends and the branch to be optimized is reinitialized. In the earliest step of the learning this has a low effectiveness since the agent is exploring quasi randomly. On the contrary this has its importance for the latest step. This forces the agent to improve quickly. To measure the agent performance, we focus on different metrics. Since the loss function in Deep Reinforcement Learning framework is hard to interpret (because of bootstrapping) we must take a look on others numbers.

Learning Constraints

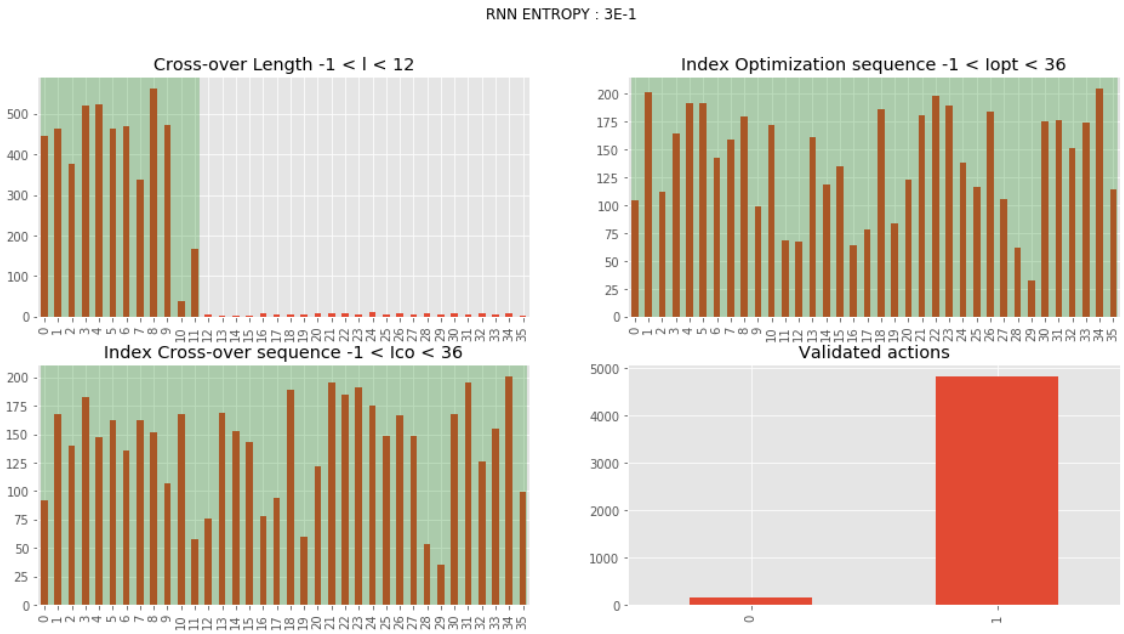
The first verification I made was about the constraints. If my agent could learn from the constraint penalty signal, then the whole learning pipeline - from the reward signal to the action distributions generated - would be validated. After which, the following part would be for my neural networks to understand the complexity of the DNA structure.

- *constraint on a single action*

Here I force the cross over length to be between 6 and 16 included, the other action dimensions are constraint-less. Here we will compare our 2 different actor's approaches: RNN actor and N-Actions actor. Below you can see their action distributions during the training, the green area stands for the constraint validity.



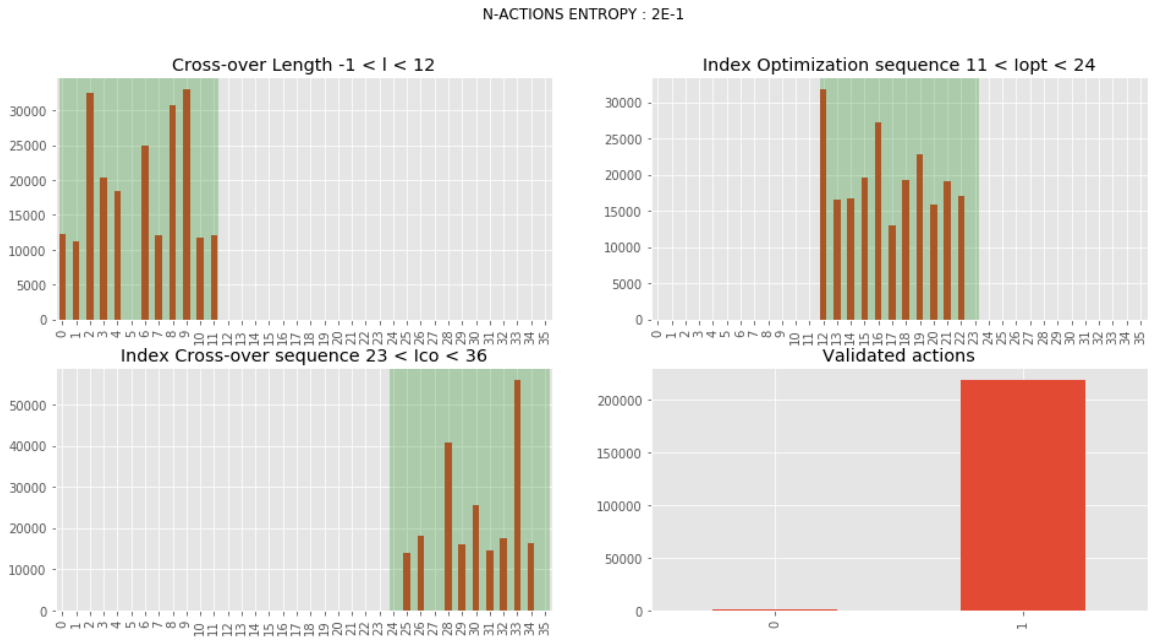
This particular constraint is learned in less than 10 episodes, meanings that a penalty reward is well understand by my agent. A better suited entropy learning rate would make the agent more explorer and thus the 8 and 14 cross over length would be more used. But the goal here is fulfill since the unauthorized actions are sidelined.

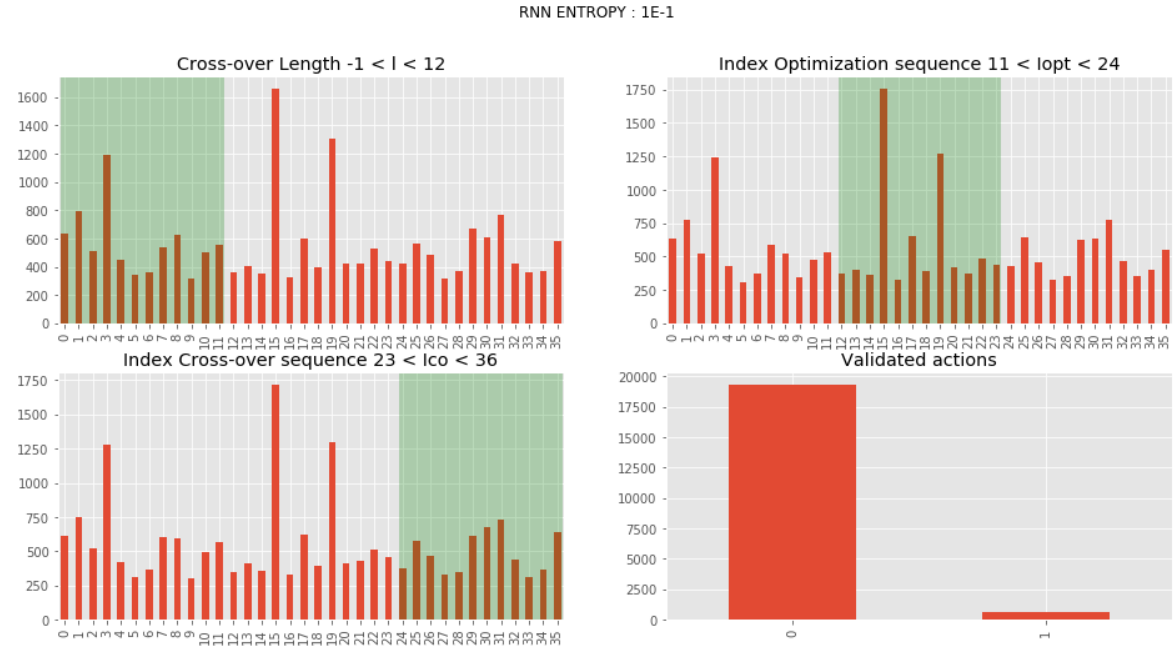


The RNN architecture does not help the constraint learning process along. Indeed, the hidden vector plays a very important role since it has to restrict outputs when equals to zero (it is initialized to 0.0) and forget this restriction in the other passes. It has a certain memory among the actions given that its neurons are the same for each action distribution generated. A longer learning process with a strong exploration seems necessary to sturdy learn the constraints.

- *Constraints on multiple actions*

When imposed to more complex constraints, the agent succeed to avoid not allowed actions. Though, a larger exploration and more training example are thus used. To learn completely constraints on multiple actions is longer since the number of unauthorized action is exponential with respect to the number of constraints. The bar plots below is generated after 1000 episodes. Even with a long training the agent has difficulties to use all the possible set of actions (*e.g the 5th action for the first action dimension etc..*)





RESSOURCE

- https://arxiv.org/pdf/1806.00589.pdf
- https://theaisummer.com/Actor_critics/