Toward machines that see like us : eye movements for deep recurrent neural networks

YAX Nicolas, DOERIG Adrien & KIETZMANN Tim,
Kietzmann Lab, Donders Center for Cognition Brain & Behaviour

$02/04 \rightarrow 27/08$

General context

Deep neural networks (DNN) are more and more used in computer vision. They have been used for many purposes such as face recognition but also as brain modelisation tools. However actual DNNs are still pretty far from human vision system. They see images globally where human only see tiny parts of them and integrate information through time using eye movements.

Mnih and al (2015) tried to give DNNs eye movements on a small scale few years ago. Our goal is to extend this idea on a large scale with state of the art machine learning methods.

Studied Problem

The aim of the project is to build a pipeline that would learn to classify images using eye movements. This means that instead of viewing the complete image directly the DNN will only see crops of it and move it scope several times to integrate the whole image as humans do. This is a very interesting project because it would make it possible to have a better understanding of human behaviour. Moreover it would allow DNNs to classify high resolution images and might limit adversarial examples attacks.

This project is very innovant especially because very few researchers thought about using eye movements as an important component of human vision and it's a very hard objective.

Contribution

I have been the first one on the team to work on this long project and it is supposed to be continued by several other neuroscientists after me. As a computer scientist working with neuroscientists my first objective was to propose a simple and efficient interface to create such ML systems in order for next researchers to quickly implement their ideas.

Then my job has been to get some results with simple systems. However due to coronavirus health crisis I had to run everything on my small computer which means I had to simplify even more to get few results.

Validity of the method

Implementing such a pipeline is especially hard with actual ML frameworks because it's a very original system and they haven't been conceived for this kind of purpose. Thus I think it was a good idea to create this framework because it has been rather hacky to get the pipeline working in practice. This way it opens the way for neuroscientists to try ideas without limitations due to technical computer science problems and unlocks the next phase of the research program.

My few results on this pipeline are quite impressive due to the internship context (I had to run everything on my small i5 computer without any GPU) and I still got pretty good classification accuracies. I used a simplistic version of the expected pipeline but it's still pretty complex compared to similar DNN achitectures and all of it runs on my tiny computer. This shows how efficient is the pipeline I have been working on.

Prospects

In the end my work has unlocked the next stage of the project : the framework works and a simple version of the pipeline runs smoothly on it which confirms our intuition about how to train efficiently such a complex and chaotic ML system.

The next steps will be to implement the whole version of the pipeline and to train it on GPU clusters and to tune learning parameters until it converges perfectly (we need to get rid of the chaotic aspect of the pipeline - and I almost got rid of it in my small version of the system). Finally if it works very well we could test adversarial attacks as we thought this system would be very likely to be immune to them (which would be an awesome discovery and a breakthrough in computer vision).

Toward machines that see like us : eye movements for deep recurrent neural networks

YAX Nicolas, DOERIG Adrien & KIETZMANN Tim,
Kietzmann Lab, Donders Center for Cognition Brain & Behaviour

$02/04 \rightarrow 27/08$

NOTE : underlined words are defined in appendix if needed.

1 General view of the project

In last few years Deep Neural Networks (DNNs) have become a major tool in computer vision. They have been used both to solve engineering issues such as face recognition as well as brain modelisation. However this modelization lacks several crucial aspects of biological neural computation.

Most computer vision problems aim at replicating human skills such as image classification. Thus it's often effective to mimic biologic mechanics and to modelise neurosciences. However modelising also means simplifying and thus choosing what matters most in biological systems. Investigating this gap between models and real systems is a way to understand both better as well as improving efficiency of models. In DNNs several big differences have been spotted and we will focus on two of them :

- DNNs are mostly feed forward whereas brain computation is heavily based on recurrences which makes it possible to integrate information through time.
- DNNs see images globally whereas humans only focus on small parts and through eye movements they progressively integrate the whole image

Incorporating these biological details to artifical systems may improve the efficiency of DNNs used for computer vision.

Thus our work will focus on building recurrent neural networks able to use saccadic eye movements and integrate visual information over time that can classify images only by seeing small crops of them.

Lab context For this internship I have been working with Adrien Doerig (Postdoc researcher - Internship supervisor) and Tim Kietzmann (Assistant Professor - Team supervisor). This team has been created in September 2019. This is a new project (I'm the first one with Adrien to work on it) and it will be continued for a few years. My job will then be to start this project by creating tools for the next ones to work on it and to get first results with a simple version of our original pipeline.

Due to the coronavirus health crisis I have been working from home with 3 videoconferences each week (two for my work and one with the team). I didn't have access to servers to run my experiments which means I had to simplify the original topic and limit the number of experimental results.

2 Approach

Our goal is to make artificial systems more biologically plausible by adding eye movements. This is an original idea that has never been tried on this scale before. The closest work with using saccades we have found was from Mnih and al (2015) where they have trained a recurrent system to learn on MNIST dataset with a very simple system using a basic reinforcement learning algorithm. The system was provided 3 crops with different zooms to classify images which is too far from human vision for us.

Our idea is to go further than this and to train more biologically plausible networks with only one crop (and not 3 with different zooms) with state of the art recurrent neural networks developped by the lab, state of the art reinforcement learning algorithms and we will try to scale it to more realistic data.

Our pipeline will combine reinforcement learning to teach the neural network where to look in the image and standard supervised learning algorithms to learn how to integrate information through time. This should result in a classifier/saccader system which classifies images as sequences of crops and makes it a very original interaction between 2 sides of machine learning that rarely cross.

This interaction is so special that most reinforcement learning framework aren't optimized for it. My job will firstly focus on how to hack those frameworks from outside to implement such a pipeline and then implement my own framework to make it easier for the others to run custom experiments. Secondly I will use this new framework to get first results on a simple dataset (similar to Mnih et al.) as a proof of concept that it works. Due to the coronavirus health crisis I had to run everything on my own computer which means I mainly focused on a very modest system, with small and simple images and only few neurons.

3 Prerequisite knowledge

As explained before we used reinforcement learning and recurrent neural networks. These notions will be briefly explained in next paragraphs in order to give the overall idea about what these things are. If you need more details about math there are further explanations in appendices.

Please note this is state of the art machine learning techniques which means it is very technical and complex. Thus I have chosen to put a very simple explanation about concepts in the report and more complex ideas in appendices. However even in appendices I wanted to keep it short and to focus more on concepts than math behind them for people who don't know machine learning to not be lost in mathematical details.

If you are really interested in these notions you can read references papers at the end of the report for exact definitions, theorems and demonstrations.

3.1 Reinforcement Learning

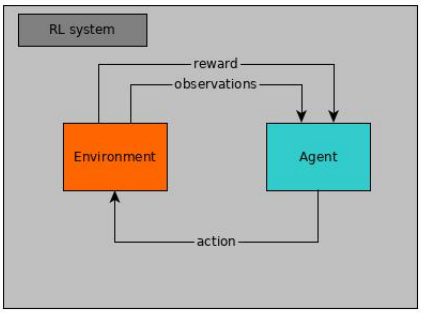

Figure 1: RL system : the agent will see its environment (collect observations) and learn to react to it (chose action) in order to get the highest cumulated reward

This part of machine learning involves involves an agent / environment system where the agent will have to learn to behave in order to maximize a reward 1 .

Reinforcement learning is a *Markov Decision Process*, the environment has states observed by the agent and transitions are triggered by actions from the agent. Each transition returns a reward that will be sent back to the agent which will try to maximize its total reward over a run.

Initially the agent doesn't know the environment, will take actions in it and observe the consequences of those actions in order to learn how to optimize its behaviour. Its aim is to find the best paths to get the highest possible reward.

The function defining the behaviour of the agent (function which from an observation returns the action) is called its *policy*.

Policy Gradient and REINFORCE The philosophy of policy gradient methods in deep learning (there are also non-deep methods) is to see the policy as a function that will be directly learnt by a neural network. The algorithm will proceed in 2 phases : collecting batches of trajectories and fitting the policy network to learn the best way to optimize rewards. To optimize it it will compute a loss function and apply a gradient descent on it through the neural network. REINFORCE was a popular early implementation of this philosophy [2]. This is the RL algorithm chosen by Mnih and al [1].

Proximal Policy Optimization (PPO) PPO is a state of the art Policy Gradient algorithm designed by OPENAI [3]. It's one of the best RL algorithm for several tasks in RL, including continuous actions [4] which we will use. It has been widely used in last few years and many RL research team try to improve it. It aims at upgrading REINFORCE by adding important features in the loss function. First when using an algorithm such as REINFORCE, the policy might change too much when fitting. Let's assume the environment is divided into several phases (summer, automn, winter, spring for example) where results of actions could be very different from a phase to another. This kind of learning might result in a policy that would change too much and overfit on each phase, forgetting everything learnt during the previous phase. There is no perfect solution for this issue.

PPO adds a term in the loss that will try to not move too much from the previous policy each time it updates in order to avoid such a behaviour.

3.2 Computer Vision

One of the main problems addressed in computer vision in which we will be interested is the classification problem. Given an image representing something among a set of defined classes (such as [cats,dogs]), the algorithm needs to find the right class to which the image belongs. Modern methods used to solve this problem strongly rely on neural networks.

Convolutional Neural Networks Convolutional neural networks are currently state of the art models for image classification. Their architecture is inspired by the internal structure of visual areas (LGN,V1,...) in brain. It makes it possible to classify images very efficiently.

The architecture of CNNs means that their activity is equivariant under translations of the visual input. this means that, if they learn to detect an object in one part of the image, they don't need to learn it again at other locations. For this reason, CNNs need fewer parameters and learn visual information much better than other architectures without these properties.

3.3 Recurrent Neural Networks

Recurrent neural networks rely on an internal state system to integrate information through time. Each time activations of a layer are computed, it takes into account internal states of those neurons to bias this activation. After the activation is computed, the internal state of each neurone is updated with this new activation. It means that each neuron is initialised with an initial internal state and each time it gets an input it will also take it state into account as input to compute its output. Once the output is computed, the state is updated with this output to be ready for next call.

This way, the network can keep track of previous calls through this internal state system and makes it possible to integrate information through time by learning how to encode important information across calls and how to decode it.

4 Detailed approach

This part will be separated in 2 subparts : first we will examine the structure of the classification system used to train both networks and results achieved for each structure. Then we will take a look at the framework I have designed and developed in order to see which tools I have been using and how I have implemented them.

4.1 Contribution - Structure of the system

The main idea is to combine 2 networks : a recurrent classifier that would see crops and try to guess the label and a saccader that would determine which sequences of eye movements (crops) would be the best for the classifier to classify the image. This sequence will be computed step by step by alternating calls to the classifier and the saccader.

Classifier The classifier will take crops as input (9x9 grayscale images) as they are centered on a pixel. It will then process it through several layers. My best results weren't with convolutions but with classic recurrent layers because MNIST is very simple and is more efficient to learn on it with simple layers. Obviously to scale the pipeline to realistic images, convolution will be a must have. The output is an array of size 10 because there are 10 classes in MNIST. Each scalar represents the probability for crops processed to represent the class of the scalar's index in the array.

It is then possible to compute how much the network is sure about its prediction by computing the crossentropy of the output for the given label. If it's close to 0 it means the network is very sure about its prediction. An entropy of 0 means that the predicted label is the right one and the predicted probability for this label to be the right one is 1.

Saccader We want the saccader to understand where to crop for the classifier to be able to classify correctly. Thus it might be a good idea for the saccader to not have the crop as input but an internal layer of the classifier. This way it should make it possible for the saccader to understand the actual state of the classifier and to return the right crop to complete partial information held by the classifier. For my internship the saccader isn't recurrent, only the classifier is. The output of the saccader is a vector of size 2 representing x and y normalised coordinates for the next crop. The saccader will learn with PPO which means we need to define a reward. We thought about many possibilities for this, including :

1. Issue each iteration - reward of 1 if the prediction is right each time the classifier is called, 0 else

2. Issue at the end - reward of 1 only for the final crop if the final prediction is right, 0 for the other iterations and if the final one is wrong

3. Minus crossentropy each iteration - the crossentropy is the loss of the classifier for its prediction - it represents how much the classifier is sure about its prediction and about if the prediction is right

4. Inverse crossentropy - a custom reward equal to $tanh(\frac{1}{crossentropy})$). The main reason behind this is when the crossentropy is very close to 0, the network is very sure about its prediction and that's a thing the saccader should encourage. This means to increase the reward, the saccader will need to look at positions which will confort the classifier in its prediction. Thus taking the inverse of the crossentropy is relevant as it will limit noise about not sure predictions as well as highly increase the reward when the classifier is absolutely sure about it prediction.After a long training, the classifier should be pretty sure about its predictions which means the reward might be too high. Clipping it with a tanh is a way to to avoid diverging high rewards.

We have tested each of them (see results).

System Here is how the whole pipeline works :

1. For each image, an initial crop which is centered (coordinates [0.5,0.5]) is computed.
2. The crop is then given to the classifier that will try to classify it (but at this stage its probability to be correct is low) and will compute the input for the saccader (internal layer of the classifier). The classifier can then compute new position for the crop and it will be computed. If 6 crops have already been computed it stops, else it jumps to 2.

We have chosen to take 5 saccades (6 crops) as it's not a lot (efficient to compute quickly) but still far enough to classify correctly each digit (Mnih and Al used up to 8 crops).

Each time an image is processed by the classifier, its input, predicted label and real label are stored. Once several images have been treated, both networks will fit on these data : the classifier will backprop on its loss : a sparse_categorical_crossentropy loss function using its predicted label and the real label. Its expression is

$$L(s) = -\sum_{c \in C} 1_c(s).log(p_s)$$

where C is the set of classes, s is the predicted label
and p_s is the probability associated with this predicted label

The saccader will keep track of its observations, actions and rewards. It will then backprop on the PPO loss to optimize its policy.

Those trainings are done at the same time on both networks to keep them synchronized.

Chaotic system This classifier + saccader learning system is very unstable : let's assume the saccader always makes saccades to the top left corner. The classifier will only get crops from this location. This means it will fit on top left crops and won't develop knowledge about the rest of the image space. If it only sees the top left part of digits, when it sees an other crop it won't understand it and won't be able to classify it. This way when the saccader explores, crops won't be understood by the classifier which will result in low rewards. Thus the system will be stuck on the top left spot and won't move anymore even if this spot isn't optimal. This means the function that is minimized through this process has a lot of local minima and the starting point and the way optimization is done are crucial. To cope with this difficulty it's important to be carefull about learning parameters such as how much the saccader will explore and initialization of both networks.

4.2 Contribution - Framework

As the first one to actively work on the project, my job is to create a reliable framework for the next ones working on it to easily implement new things, without worrying about learning different coding frameworks used to handle the RL part and the supervised part. My goal is thus to create an easy to learn, easy to understand and easily expandable framework that can hide hacks of RL frameworks and handle them for the user without them having to understand how they work.

RL frameworks we have used are recent which means the documentation is very slim and often hard to find why your program doesn't work with information found in the documentation or on the internet. To understand how to implement such an innovative and original system I had to read the source code of huge libraries to produce a small code that

would actually work fine. This lack of documentation was a real issue and I've spent most of my time reading and printing everything in RL frameworks to really understand how to hack those into our original system.

4.2.1 GRLF - Graph Reinforcement Learning Framework

We want to be able to implement this mechanics of two networks interacting and learning together. It's fairly easy to draw this pipeline with a graph and its also a versatile way to change its structure for future improvements. It makes it possible to easily express the synergy between both networks and to efficiently modify the dynamics of the system only by moving few edges.

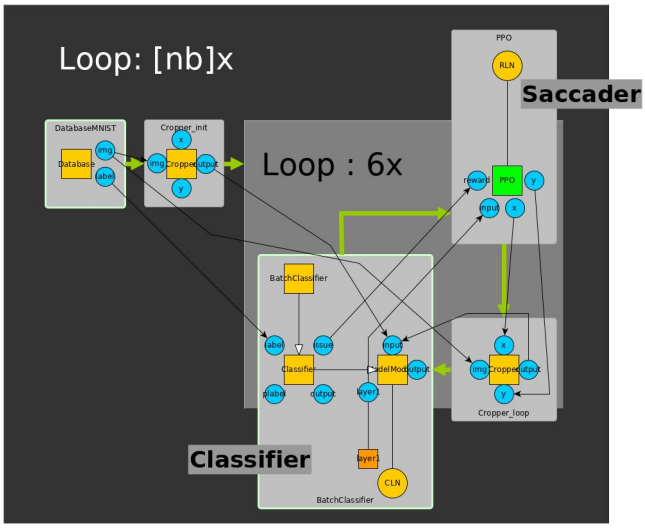

Figure 2: Graph design of the pipeline

The framework I've designed is graph based. We want to write the pipeline we have discused before with a graph.

You can find the same structure we have seen in previous section expressed as a graph 2. Database and initial crop are at the top left hand corner while the inner loop is on the right with the saccader, the classifier and a cropper to crop the image according the the saccader's outputs. The MNIST database loads images and send them to the first cropper (left part) to get the initial centered crop. This crop will then be classified in order to compute the input for the saccader and to get an other crop with the second cropper (right part). This new crop will loop again until the classifier has seen 6 crops (1 initial and 5 from the saccader).

Aim of the framework To sum things up my main concern about making this framework is to

- Find a way to implement the original pipeline with both networks learning together using reinforcement learning librairies

Moreover my additional objectives about making this framework are to make it:

1. Easy to understand (learn and read code from others)
2. Easy to extend (everyone can add code)
3. Change the RL framework easily (versatile)

Thus we will first see the structure of RL frameworks and why reinforcement learning librairies are not made for this kind of applications and how I have come up with a way to make it work and then how I have designed the framework to implement these hacks.

4.2.2 Structure of RL framework

During this internship I have worked with 2 RL frameworks : OPENAI baselines (creators of PPO) [5] and RLLIB (more scalable than OPENAI) [6] but what we will see in the next paragraph applies to most RL frameworks. Most RL framework uses the same global structure with 2 very different objects : Agents and Environments.

Environments Environments only provide observations and rewards. They are worlds used by agents to learn. The most famous standard of environment is the gym environment (made by OPENAI) whose specification is fairly straightforward :

```
class GymEnv:
  def __init__(self):
    #intializer
    self.observation_space = #type and shape of the observation space
    self.action_space = #type and shape of the action space
  def step(self,action):
    #modifies the state of the environment and returns obs,rew,done=(if the env
    needs to be reset),info=debug
    return next_observation,reward,done,info
  def reset(self):
    #resets the environment
    return initial_obs
```

This standard is widely used and all RL frameworks are compatible with this type of environment. RL frameworks have their own internal data structure for environments and have a function to convert gym environments into this representation. Usually when people give a gym environment for the training, the framework will create an internal environment that will represent an array of duplicated gym environments. This way the agent gets batches of observations and its more efficient to compute a batch of actions with neural networks and a GPU than actions one by one. Thus you cannot create a gym environment returning a batch of observations directly because the framework will considere the batch as a single observation. This means that in RL frameworks, Gym environments are strictly less expressive than the inner environment structure.

Because of the universal usage of Gym Environments it might be a good idea to use them especially because we want to switch between RL frameworks easily (secondary objective 3). However they are less expressive than inner environment structure (that will change from a framework to an other) which might become an issue.

Agents In most RL frameworks, agents have the main loop during the training which means people call the train method of the agent and it will call the environment accordingly without any action from the user.

There isn't any universal standard for agents which means agents data reprensentations will change from a framework to an other. That's why we will need to be particularly careful to set a compatibility system so that any data structure from any RL framework is compatible with ours.

4.3 RL libraries used

During this project I have been using 2 RL libraries : OPENAI baselines [5] and RLLIB [6]. OPENAI are designers of PPO so we thought it would be a good choice. However this library is more a test library (to verify results from the paper) rather than a development library. Its structure is terrible to develop on and its architecture makes it difficult to optimize it.

In fact the RL part of our system is only the saccader. It's the only neural network that learns with RL when the classifier learns by supervised learning. This means our agent will have the saccader as neural network and the classifier will be part of the environment. However if we use gym environments, the framework will convert it to its own environment representation by creating a batch of environments. We can make sure all these environments share the same classifier but it means that the inner environment will work with batches to make it possible for the saccader to learn faster. Thus the latter will return a batch of actions that will be distributed in each environment according to its index. The classifier will then be called by the environment on inputs one by one which is very inefficient. Thus it's not possible to efficiently use gym environments but we will need to sacrifice additional objective 3.

For each RL framework we will need to create a conversion system to transform our graph based environment into the internal environment data representation for each framework used instead of just creating one for gym envionments.

Nevetheless even with this trick it's still pretty slow (see results).Moreover it doesn't handle GPU clusters which will be needed for next steps of the project. We thus decided to move to RLLIB which makes it possible to run our RL training on multiple workers in order to optimize it.

RLlib is more complex : it won't create a unique internal environment but several that will run on parallel workers (see it as computers in a network). This means the latter trick won't work especially because it's super hard to share the classifier between workers (tensorflow isn't made for that). This means we will have to put all our neural networks in the hub together. Both of them will be in the agent which can only have one neural network due to the agent structure TFPolicies in RLLIB. This means we will have to create a branched network that will take the standard input for the classifier and create outputs of both networks at once but only apply the RL algorithm on a single output. RLLib makes it possible for TFPolicies to define a custom loss function where we can cut the gradient of the RL loss on the classifier's outputs. Thus by adding a second term in the loss being the standard supervised

loss for the classifier it's possible to train both networks at once with a gym environment (there isn't any network in the environment anymore).

With RLLib the framework runs faster and is more versatile : it's possible to change the loss function for PPO and the classifier very easily and defining a single branched network is easier than two separate definitions.

4.3.1 Structure of the framework

Now that we have seen hacks I've been using to make it work let's see the structure of my framework to hide those hacks.

General system To be sure to be compatible with RL frameworks it's important to keep the agent/environment separation as it's the only common structure in all RL framework. The framework must make it possible for the user to define a system that will be compiled in an agent/environment system to call libraries accordingly. This way the user doesn't need to understand how RL framworks function in depth to build its system.

Graph dynamics The main idea is to propose an interface based on a graph which will be defined by the user describing the dynamics of the system **??**. This graph will be composed of modules (nodes) that will be connected between each other.

Modules A module is an object that can be connected to other modules through its *ports* (analogy with networks). A *port* is a value from a module that can be accessed from other modules and shared with them. Basically a module gets information from other modules through its ports, computes something with thesd values and stores outputs of the computation in its ports to be sent to other modules. Thus the following architecture will be used :

```
class Module:
  def __init__(self):
    #intializer
  def run(self):
    #Compute things
```

Ports are defined using python objects as dictionaries : 3 prefix keys will be used to interact with ports :

- '*' : define a port and give its type
- ':' : redefine the type
- 'ϵ' : get the value of the port
- '¿' : used to create links between nodes (we will get back to this later)

Here is an example of the definition of a module :

```
class DoubleModule:
  def __init__(self):
    #intializer
    self["*a"] = float #definition of port a as float
    self["*b"] = float #definition of port b as float
  def run(self):
    #Compute things
    self["b"] = self["a"]*2 #Use the value of port a to double it and put it in
    port b
d = DoubleModule()
print(d["a"])
>>> None
d["a"] = 5.
d.run()
print(d["b"])
>>> 10.
d[":a"] = int #Will autoconvert the initial value in a to int
d.run()
>>> Exception : port b has type float but received int value
d[":b"] = object #Universal type
d.run()
print(d["b"])
>>> 10
```

As you can see I've added a type system that doesn't exist in Python to make it more convenient for users. It's possible to bypass it if needed. Several modules can be defined this way to compute complex functions.

Connecting modules Modules can be connected with the following syntax :

```
#Define module 1
module1 = Module()
module1["*a"] = int
#Define module 2
module2 = Module()
module2["*a"] = int
#Connect module1["a"] to module2["a"] (-> direction of arrows)
module1[">a"]([module2[">a"]])
```

Loop and Graph object Loop makes it possible to handle the system we have talked about to compute modules and ports dynamics :

```
#Define module 1
loop = Loop([module1,module2],nb=1) #Modules will run in this order (nb=1
    iteration of the loop)
module1["a"] = 5
print(module2["a"])
>>> None
loop.run() #run module1.run (empty) then send module1["a"] to module2["a"] and
    run module2.run() (empty)
print(module2["a"])
>>> 5
```

With this loop system it's possible to run a sequence of modules several times.

Graph and agent/environment system Graph object makes it possible to compute the agent/environment system. First we need to define a RL_Module :

```
import tensorflow as tf #Main ML library -> tf.Tensor is an array
class RL_Module(Module):
  def __init__(self,input_shape,output_shape):
    self["*input"] = (tf.Tensor,input_shape) #Observation : type and shape
    self["*output"] = (tf.Tensor,output_shape) #Action : type and shape
    self["*reward"] = (tf.Tensor,(None,1)) #Reward : type and shape
  def run(self):
    #Compute things
```

Once a RL_Module is defined in a Loop object it can be upgraded to a Graph object in order to get the agent/environment system. This is a dummy example with empty modules to show how it works :

```
#Definition of Modules (nodes)
rl_module = RL_Module((10,),(2,))
m1 = Module()
m1["*action"] = tf.Tensor
m1["*a"] = tf.Tensor
m2 = Module()
m2["*a"] = tf.Tensor
m2["*obs"] = tf.Tensor
m2["*reward"] = tf.Tensor
#Definition of connections
rl_module[">output"]([m1[">action"]])
m1[">a"]([m1[">a"]])
m2[">obs"]([rl_module[">input"]])
m2[">reward"]([rl_module[">reward"]])
#Definition of the graph
loop = Loop([m1,m2,rl_module],nb=10) #Run 10 times the list
graph = Graph(loop)
#Run the graph
print(graph.environment)
>>> GymEnv at XXXXXXXX
print(graph.agent)
>>> Agent at XXXXXXXX
graph.run(1000) #Run 1000 times the loop
```

It's then possible to only use the agent or the environment from the graph object with an other framework if needed. The agent is compatible with GymEnvironments and the environement is a GymEnvironment. You can also run both together with the graph.run method. We'll now see how to implement RL_Modules more in depth to then go further in difficulties of implementation.

RL_Modules more in depth In the team we aren't professionals of RL which means we will prioritize using existing powerful RL frameworks in order to be sure about the efficiency of the implementation of the algorithm. In most RL frameworks, agents are objects with a run method and an environment parameter. Calling this method will run the main training loop. However in GRLF the definition of the graph contains both the agent and the environment with a loop system including both. This means it's won't be easy to extract the environment and agent functions from this graph structure that include them both.

We have added a new method for modules : the run_after method that will be called right after the run method. Now the RL_Modules code looks like this :

```
import tensorflow as tf #Main ML library -> tf.Tensor is an array
class RL_Module(Module):
  def __init__(self,input_shape,output_shape):
    self["*input"] = (tf.Tensor,input_shape) #Observation : type and shape
    self["*output"] = (tf.Tensor,output_shape) #Action : type and shape
    self["*reward"] = (tf.Tensor,(None,1)) #Reward : type and shape
    self.learn_fn = #Function from the RL framework : (env,nb_timesteps) ->
    object
  def run(self):
    #Compute things
    raise RLInput(self["input"]) #raises the exception
  def run_after(self):
    #Compute things with self["output"] = output of the agent in the RL framework
```

4.3.2 Baseline modules

There are several objects modules can heritate from (which creates a type system over modules) : FitableModules (contains a datastructure that can learn) and PointerModules.

MNIST Database This module doesn't have any port used as input. It will fill its "img" port with a MNIST image and the "label" port with its label. Images and labels are changed each time the run method is called. Images are chosen in the order of the database (n call is the n-th image of the dataset).

Cropper This module is very simple : it gets as input an image "img" and crop coordinates "x","y" and the run method will return in "output" the smaller image corresponding to "img" cropped at those coordinates.

ModelModule - FitableModule A ModelModule has an internal neural network and two ports : "input", "output". Calling run calls the network on "input" and returns the result in "output". The user can also access any layer in the neural network through the port that has the same name.

Classifier - PointerModule FitableModule A Classifier module is a PointerModule to a FitableModule and will have 4 ports : a reshaped "output", a "label" port for the real label, a "plabel" port for the predicted label and a "issue" port with a boolean value about whether the predicted label is the same as the real label. Calling run will call the ModelModule run method, reshape the output to a given shape (it's more convenient this way), computes the predicted label from the output of the neural network to then fill the "issue" port.

BatchCapsule - PointerModule This pointer module is more complex : it will capture each input in "input" and "label" ports of the pointed module and store them in a batch. Once enough data collected (treshold defined by the user) it will fit the pointed module on collected data (it should point to a FitableModule).

PPOSaccader - PointerModule,FitableModule A RL_Module with a learn_fn for PPO which points toward a FitableModule. It will compute x and y coordinates for the saccade from the output of its FitableModule.

As you can see those modules makes it possible to train both networks and the framework stays pretty easy to use and to learn.

There are 2 different kind of RL frameworks : standard ones with simple execution and frameworks using workers to optimize the learning on GPU clusters. I've worked with both types of frameworks and developped a version of my own framework for each of these execution types.

5 Results

I have tested 2 types of neural architecture :

- Feed Forward : the classifier isn't recurrent (easier to debug) which means the task of the saccader isn't to complete information the classifier needs to decide but to find the best spot for the classifier to classify with one crop.
- Recurrent : classic pipeline.

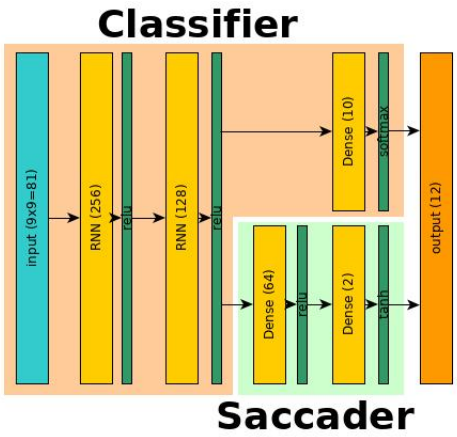

Figure 3: Structure of the network used

The structure of the network used is described in 3. For OPENAI baseline both networks are separed and for RLLib they are in the same tensorflow model.

Here are accuracies (%) for training/test data using this network architecture after having seen 100 000 images . Those values are the average of 3 run for each (accuracies were very close to each other).

	Feed Forward	Recurrent
baseline center	80/80	88/86
baseline random	28/28	44/44

	γ	λ	Feed Forward	Recurrent
Binary reward	low (0.9)	low (5.10^{-5})	77/75	90/90
Each step binary reward	low (0.9)	low (5.10^{-5})	79/79	91/89
	low (0.9)	medium (1.10^{-3})	83/77	92/92
	high (0.99)	low (5.10^{-5})	84/75	91/90
	high (0.99)	medium (1.10^{-3})	87/79	94/91
Crossentropy reward	low (0.9)	low (5.10^{-5})	63/54	89/84
Custom crossentropy reward	low (0.9)	low (5.10^{-5})	77/61	90/89

Table 1: Accuracies got from the pipeline

Those results are similar with openai baselines and rllib except that with openai I need hours to get results where 20min are enough with rllib. It's also important to notice that I'm runing this on my own computer with a bad configuration for machine learning (dual core and no gpu).

This table shows that giving a reward only at the end of the task seems to perform better (that's why I have chosen this one to test with more parameters). In fact crossentropy rewards are too noisy to truely learn on them. Those rewards are float numbers where binary rewards are booleans. Float rewards will vary a lot because each time the classifier fits they will all change a little bit which could completely change the optimal policy. Thus boolean rewards make it less chaotic and are better for learning.

Then, giving a reward for each intermediate prediction is more vague for the agent because what matters the most is the final result. Obviously it's better if the classifier finds the right label quickly is perfect (what the standard binary reward is better for) but this reward doesn't help the system to know that what is truely important is the last prediction.

Lastly the main issue with this end boolean reward is that it's sparser than the other one : rewards are delayed. The agent will need to do more before being rewarded. The path is thus more vague and the agent might not be able to understand the link between actual actions and future reward associated with those actions. It seems like a higher γ gives better results and that's the exact parameter influencing this sparse aspect of the reward. In the return function, gamma expresses the way future rewards will be taken into account to make choices. Thus to make it work with this reward it's important to have an adapted γ and 0.99 seems to be a good value.

6 Discussion

In the end, my goal has been achieved : the framework is done, it respects most objectives and I've pretty good accuracies as well. They are not as good as Mnih and al but I think it's due to the fact that we only have 1 crop as input, I haven't been able to grid search for optimal parameters (very important in RL) and to run longer trainings due to coronavirus health crisis. I have run one long training with low γ and λ which reached 98% accuracy after few hours.

Even if accuracies are good if the training runs for too long, fixations tend to focus on a single spot. This means the system learns fast but overfits rather quickly too. This is weird because it means the loss function isn't perfect and it won't learn optimal fixations. It will start by learning them and will finally be pushed aside to be replaced by a very simple saccade behaviour but a trained classifier for these fixations (overfitted classifier). I think this

problem is directly linked to the chaotic behaviour of the training which can very quickly deviate and limit its learning. This issue only appear on long training as fixations are sparse on short trainings (the saccader doesn't have enough time to overfit on the classifier).
The convergence speed is also pretty impressive (similar to Mnih and Al). Accuracy and speed are the two ingredients needed to scale up, so my work has unlocked the next stage in the research program !
This new stage will be about improving the pipeline with state of the art neural architectures I haven't been able to try because of the coronavirus health crisis such as recurrent classifiers developped by the lab. Then it should be about analysing fixations and trying adversarial attacks to verify if this new pipeline is more robust than classic DNNs.

References

[1] Mnih Volodymyr, Heess Nicolas, Graves Alex, and Kavukcuoglu Koray. Recurrent models of visual attention. 2015.

[2] Richard S. Sutton, McAllester David, Singh Satinder, and Mansour Yishay. Policy gradient methods for reinforcement learning with function approximation. 2010.

[3] Schulman John, Wolski Filip, Dhariwal Prafulla, Radford Alec, and Klimov Oleg. Proximal policy optimization algorithms. 2017.

[4] Marcin Andrychowicz, Anton Raichuk, Piotr Stańczyk, Manu Orsini, Sertan Girgin, Raphael Marinier, Léonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, Sylvain Gelly, and Olivier Bachem. What matters in on-policy reinforcement learning? a large-scale empirical study. 2020.

[5] Openai baselines github - branch tf2. https://github.com/openai/baselines/tree/tf2.

[6] Rllib github. https://github.com/ray-project/ray/blob/master/python/ray/rllib.

7 Appendix

7.1 Neural Networks

Neural networks are one of the most widely used methods in machine learning, and are a crucial part of this project". A neural network is a function from $\mathbb{R}^m$ to $\mathbb{R}^p$ with n parameters $(w_1, ..., w_n)$.

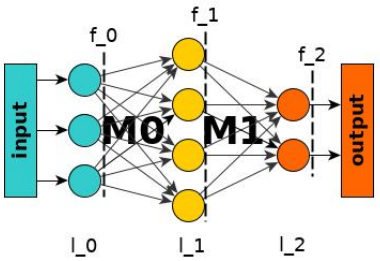

Figure 4: Small neural network with layers (l_0, l_1, l_2), weight matrix (M_0, M_1) and activation functions (f_0, f_1, f_2). Biases aren't represented for the sake of clarity.

Neural networks are arranged in layers which communicate through weight matrices 4. Layers are usually linear operations which means at the end of each layer a non linear function will be applied such as relu :

$$relu(x) = max(0, x)$$

This last one is particularly famous because it is easy to compute and performs well in practice.

$$\begin{cases} l_{i+1} = f_i(M_i.l_i + b_i) \\ l_0 = input \\ l_q = output \end{cases}$$

with l_i the i^{th} layer ($i \leq q$), M_i the matrix to compute the excitatory signal leaving layer l_i toward layer l_{i+1}, f_i the activation function of layer l_i (compute the activation of neurones from the excitatory signal) and b_i is a bias.
The whole function is then

$$out = f_q(M_q....f_i(M_i....f_0(M_0.input + b_0)... + b_i)... + b_q)$$

Parameters are matrix's weights and biases. They represent connection's strength between neurons and threshold to excited neurons. We'll see in next paragraph how to tune parameters to learn something to the neural network.

Supervised Learning In the supervised learning setting, a neural network learns to classify images using labeled data :

$$((x_0, y_0), (x_1, y_1), ..., (x_s, y_s)) \in \mathbb{R}^{2^s} (s \in \mathbb{N})$$

The goal is to make the neural neural fit on these values. This means that our model f should verify $\forall i \leq s, f(x_i) = y_i$ or at least $f(x_i) \approx y_i$. In practice the first case is almost impossible to

reach and we try to minimize a *loss function* that will define the '$\approx$' semantics. The simplest loss function is probably the squared L2 norm : $L = \sum_{i \leq s}(f(x_i) - y_i)^2$.
To minimize this loss with a neural network we can compute the gradient of this loss for each parameter of the network and apply the gradient descent algorithm. Gradient descent step :

$$\forall i \leq n, w_i = w_i - \lambda \frac{\partial L}{\partial w_i}$$

$$\textit{learning rate} : 0 < \lambda$$

If the gradient for a parameter w is positive it means that the loss function should increase as the parameter increases. Thus by decreasing the value of this parameter the loss will decrease and the model will better fit the base. If it's negative the loss should decrease as the parameter increases. By applying several steps of this algorithm it's possible for a neural network to approximate a base with a pretty good accuracy (L very low).
Defining the loss function is crucial as it will define the way gradients will be optimized which makes it one of the most important thing to work on in machine learning.

7.1.1 Convolutional Neural Networks

Convolutional neural networks are inspired by the internal structure of visual areas (LGN,V1,...) in brain. It makes it possible to compute images very efficiently. It's easier to understand convolutional neural networks by seeing states of layers as images (2 or 3D arrays) rather than a simple vector (1D array) as it was described previously.

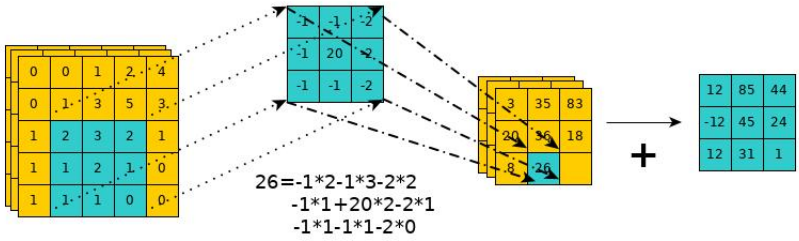

Figure 5: Convolution computation

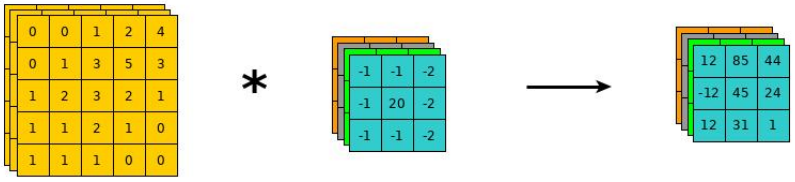

Figure 6: Convolution layer computation

Computation of a convolution with a kernel is represented in 5. A convolution product is applied for each of the c channels of the image and are then sumed to get a single channel image. This image is on of the channels of the output 6. Thus from an input with dimensions

(h,w,c) and n kernels, the output has shape (h,w,n) (images are padded so that h and w doesn't vary). Each layer has an internal matrix known as convolution kernel with few weights. This convolution product makes it possible to have a system invariant to translation which is particularly powerful when it comes to process images.
The role of convolutions is to extract features from images : some kernels will catch straight lines where others might react more to circles. This way it makes it possible to extract more and more precises features from the images such as faces and specific textures.
However with only convolution layers, the shape of the image doesn't vary. It's very interesting to have more and more features as it is processed to have representations of many objects in order to classify them properly. That's why the number of channels must increase for deepest layers and the size of the image decrease as it makes it possible to have more focused features.
This is why people also use pooling layers combined with convolutions.

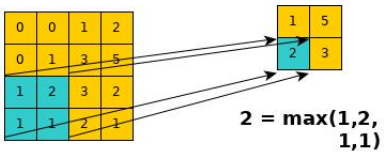

Figure 7: Max Pooling layer computation

Max Pooling layers reduce the size of the image as it goes through the network 7 by only keeping the sharpest details.
Usually people stack several convolutions, one pooling layer and a relu activation several times to make their network.

7.1.2 Recurrent Neural Networks

A recurrent neural network works similarly than a simple neural network excepts that it will keep an internal state that will interfer with next calls of the model.

$$\begin{cases} l_{i+1,t} = f_{i+1}(M_{i+1}.l_{i,t} + S_{i+1}.l_{i+1,t-1} + b_{i+1}) \\ l_{0,t} = input_t \\ l_{q,t} = output_t \end{cases}$$

There is now a temporal system where the activation of the layer during the previous timestep is needed to compute timestep t. The first timestep is computed with an arbitrary value for the previous timestep (usually 0). This makes it possible for the network to remember previous information and to integrate information through time. Gradient descent works similarly by backpropagating through timesteps. However the network might forget rather quickly the first timesteps when the training is long (high number of timesteps).
That's why there are many variants for the way to use the internal state of recurrent networks, such as LSTM (Long Short Term Memory) that will make it possible for the network to control which parts of the internal state to remember or discard.

7.2 Reinforcement Learning

Reinforcement learning involves an agent / environment system where the agent will have to learn to behave in order to maximize a reward.
More formally the environment contains a set of states S, a set of initial states $S_I \in S$, a set of final states $S_f \in S$, a finite set of actions that can be taken in any state A, a stochastic transition function P being a probability distribution over events S ($P(S_{i+1}|S_i, A_i)$ reprensents the probability when taking an action A_i in state S_i that the environment goes in state S_{i+1})and a reward function R:(S,A)$\rightarrow \mathbb{R}$ telling which reward is associated with a transition. This is part of the environment, the agent doesn't know S,A,P nor R and will explore them.
A trajectory in the environment is a finite run from an initial state to a final state : $(S_0, A_0, R_0), ..., (S_i, A_i, R_i), ..., (S_q, A_q, R_q)$. The final state is often excluded of the trajectory as no action is taken in it. Trajectory must be admissible ($\forall i \leq n, P(S_{i+1}|S_i, A_i) > 0$). The set of trajectories will be called T. Trajectories might be infinite if the environment allows it but in practice they are finite.
Initially the agent jumps in the environment with 0 knowledge about it and it will have to explore to learn where best rewards are and which trajectories in P are the most interesting ones. It will have a memory M that is the set of previous trajectories in the environment. The agent behaviour will be modelised by a function π:$(S, \mathbb{P}(T)) \rightarrow [0, 1]^A$ called the *policy* associating a probability distribution over the action space to each state and memory. This function will be learnt.
Each step, the agent is in a state S_i and chooses an action $A_i = \pi(S_i \| M + ((S_0, A, R_0), ..., (S_{i-1}, A_{i-1}, R_{i-1})))$ which changes its state to $S_{i+1} = T(S_i, A_i)$ and returns the reward R_i= R(S_i,A_i). If the new state is final, the trajectory is over, if it's not, the agent continues to step. Initially the agents starts in an initial state S_0 (chosen arbitrarily of randomly) and steps.
We can define a *return* over a trajectory as the sum of rewards for this trajectory. However in most cases it is more interesting to use a discounted sum of rewards as it gives an idea of neighbourhood to the gain function:

$$R_\pi(\tau) = \sum_{(S_i, A_i, R_i) \in \tau} \gamma^i . R_i$$

$$\gamma \in]0, 1[$$

This quantity defines the reward obtained over trajectory τ by following policy π (π isn't used in the computation, it's just for notations : this *return* represents reliability of π).
To improve the agent will run several trajectories always trying to understand better its environment and finding the best policy to optimize its gain. After each trajectory π might be updated by the RL algorithm for this purpose.
The aim of RL is to find the best algorithm to maximize the sum of gain only giving the agent a specified period of time to improve. In practice it could be a robot trying to find a path in ruins of a collapsed building to find survivors. It needs to understand in real time the shape of the environment to map it and to avoid getting hurt. This means it will very often have to choose between exploring or exploiting what the agent already knows. If it explores all the time it won't be able to get enough reward especially if time is short whereas if it explores a bit and after only exploits it knowledge without learning more it may get far less reward than it could.

In practice it's almost impossible to get a complete understanding of the environment (think about Chess and Go which are too complex games with too many states / actions to be fully explored). This means the policy needs to extrapolate its knowledge understanding a logics behind the environment. This can be acheived using tools such as Neural Networks. There are many ways to optimize π function but to make it short we will only explore the Policy Gradient method.
In practice observation space and action space are arrays of float values.
In RL there are 2 additional usefull functions that will be used later : Q and V.

$$\forall s \in S, Q_\pi(s,a) = \mathbb{E}_{\tau \| \tau[0]=(s,a,\_)}(R_\pi(\tau))$$

$$\forall (s,a) \in S \times A, V_\pi(s) = \mathbb{E}_{\tau \| \tau[0]=(s,\_,\_)}(R_\pi(\tau))$$

Q_π represents the value of an action in a specific state after taking this action with policy π whereas V represents the value of a state while following π. We then finaly define optimal V and Q :

$$\forall s \in S, V_*(s) = max_\pi V_\pi(s)$$

$$\forall (s,a) \in S \times A, Q_*(s,a) = max_\pi Q_\pi(s,a)$$

7.2.1 Policy Gradient - REINFORCE

The philosophy of policy gradient methods is to see the policy π as a function that will be learnt by supervised learning on batches of observations. The aim of policy gradient is to use a neural network as policy model and to use a specific loss function to increase the score of the agent. This policy will have a probability of chosing a random action as exploration parameter instead of returning its estimated best action.
Usually the algorithm will proceed in 2 phases : first it will collect information from the environment (n trajectories) and then it will fit the policy on those collected data. The policy will be modelised by a neural network which will, given an observation (a vector), return the associated action (a vector).
REINFORCE is probably the simplest Policy Gradient algorithm and is directly using the sum of the estimated expectation over the batch of collected data as loss for the neural network.

$$-\mathbb{E}(\sum_\tau R_\pi(\tau))$$

The algorithm collects a batch of X trajectories to fit the neural network on it. This process is repeated several time until convergence. This function can be easily computed in practice as well as its gradient especially because the gradient only depends on the policy and not on π. This process of collecting / fitting is repeated several times.

7.2.2 PPO

PPO is a state of the art Policy Gradient algorithm designed by OPENAI. It aims at improving REINFORCE by adding few features in the loss function. First when using an algorithm such as REINFORCE, the policy might change too much when fitting. Let's assume the environment is divided into several phases (summer,automn,winter,spring for example) where results of actions could be very different from a phase to another. This kind of learning

might result in a policy that would change too much and overfit on each phase, forgetting everything learnt during the previous phase. There is no perfect solution for this issue.
PPO adds a term in the loss that will try to not move too much from the previous policy in order to avoid such a behaviour. This term uses a more complex structure than REINFORCE. The policy will be composed of 2 neural networks : the policy network which returns actions and the value network. Both uses the same input and that's why very often they form a single network with 2 outputs : the action vector (policy network) and an additional output (float) for the value network. The role of the value network is to estimate the value of the actual state ($V(s)$ we have discussed about earlier). We can then define the *advantage function* as $\hat{A}_\pi(s,a) = Q_\pi(s,a) - V_\pi(s)$. It represents the interest of taking action a in state s rather than a'. If it's positive it means that a is better than the average.
Finally we need a last definition : $\hat{r}_{\pi,\pi_{old}}(s,a) = \frac{\pi(a|s)}{\pi_{old}(a|s)}$ the ratio between actual policy and the previous one. This ratio quantifies movements in the policy and this term will allow the loss to minimize them as well as improving the policy. This trade-off between both stabilizes the learning. We can finally define the PPO loss :

$$L_{clip} = \mathbb{E}(\sum_{(s_i,a_i,r_i)\in\tau} (\min(\hat{r}_{\pi,\pi_{old}}(s_i,a_i)\hat{A}_t, clip(\hat{r}_{\pi,\pi_{old}}(s_i,a_i), 1-\epsilon, 1+\epsilon)\hat{A}_t)))$$

It's a very complex loss that will be maximized instead of minimized. To understand what it does let's consider the inner term :

$$\min(\hat{r}_{\pi,\pi_{old}}(s_i,a_i)\hat{A}_t, clip(\hat{r}_{\pi,\pi_{old}}(s_i,a_i), 1-\epsilon, 1+\epsilon)\hat{A}_t)$$

The first part is

$$\hat{r}_{\pi,\pi_{old}}(s_i,a_i)\hat{A}_t$$

If $\hat{A}_t$ is positive, then the action that has been taken is better than expected which means its probability should be increased. By increasing the loss, $\hat{r}_{\pi,\pi_{old}}(s_i,a_i)$ will increase which means the probability for a_i to be taken in s_i will tend to be more likely than in previous policy.
Now let's take a look at the other term :

$$clip(\hat{r}_{\pi,\pi_{old}}(s_i,a_i), 1-\epsilon, 1+\epsilon)\hat{A}_t)$$

This clipping makes it possible for the loss to stay constant if $\hat{r}_{\pi,\pi_{old}}$ is too far from 1.

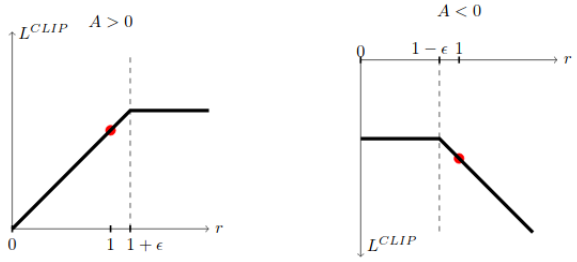

Figure 8: PPO inner term

This means that it will flatten the curve of the loss in order to prevent the algorithm to try to go too far from previous policy. The min operator will then ensure the continuity of curves 8.
This algorithm has proven very efficient lately.