SAPIENZA
UNIVERSITÀ DI ROMA

A practical study of Deep Convolutional Q-Learning

Alfano F. 1262220 - Vargiu L. 1422135

August 27, 2018

Chapter 1

Introduction

The purpose of the seminar is to prove how the reinforcement learning can be used to make an agent easily learning how to play to vintage games, the test are prompted on Atari games, Doom and Super Mario bros. The interesting part is that the input is a matrix of pixels so convolutional neural network carried the task through. During the work we analyzed some papers and we used already tested CNN alternately with custom CNN. The code is written also in a way that a future user can test his self-made CNN.
Before going in depth into the project would be useful a quick overview of the technologies we are going to use.

1.1 Reinforcement learning

Reinforcement learning has no dataset, but the agent's actions are driven by the so called "cumulative reward". The learning comes from two phases:

- **Exploration:** in this phase the agent tests some actions, possibly in a clever way, in order to get the best action sequence.
- **Exploitation:** in this phase the agent "knows" better his environment and selects a good sequence.

We also know that the result of an action can be:

- Deterministic
- Stochastic

A reinforcement learning problem is often modeled as Markov Decision Process.

1.1.1 Markov Decision Process (MDP)

A MDP is a process where the current state is independent of the previous ones. As Wikipedia states:

> *"Markov decision processes (MDPs) provide a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker. MDPs are useful for studying a wide range of optimization problems solved via dynamic programming and reinforcement learning. MDPs were known at least as early as the 1950s (cf. Bellman 1957); a core body of research on Markov decision processes resulted from Ronald A. Howard's book published in 1960, Dynamic Programming and Markov Processes. They are used in a wide area of disciplines, including robotics, automatic control, economics, and manufacturing. "* [1]

In particular a problem is defined by a 5-tuple (S, A, P_a, R_a, γ) where:

- S is a set of states
- A is a set of actions
- P_a is a set of probabilities where $P_a(s, s^1$ is the probability to go to the state s^1 if we are in the state s
- is the immediate reward after a transition
- γ represent the discount factor, which is a way to get the difference of importance between the present reward and the future reward.

The goal of a MDP is to choose a policy π in order to maximize the cumulative reward:

$$\sum \gamma_t R_a(s_t, s_{t+1}) \tag{1.1}$$

We see that the probability makes implicit the case of a **stochastic search**.

1.1.2 Convolutional Neural Network

A convolutional neural network belongs to the more general Deep Feed-Forward artificial neural network. His goal is to try to emulate the human vision using some properties that underline the features and try to give the so called space invariance. The inspiration comes from biological processes.
As we can see in the following figure, a CNN is composed by different parts:

1. **Feature maps:** are the layers in charge of extract the features of a image (edges, shadows, etc..).
2. **Pooling layer:** it gets a double benefit, this is the part which gives translation invariariance loosing some informations. The second benefit is the reduction of the images and consequently the computational time.

3. **Fully Connected Layer:** at the end of the Layers we found a Fully connected layer that elaborates the data given from the previous one in a 1-D array. The procedure which converts the data into a 1-D array is called Flattening.

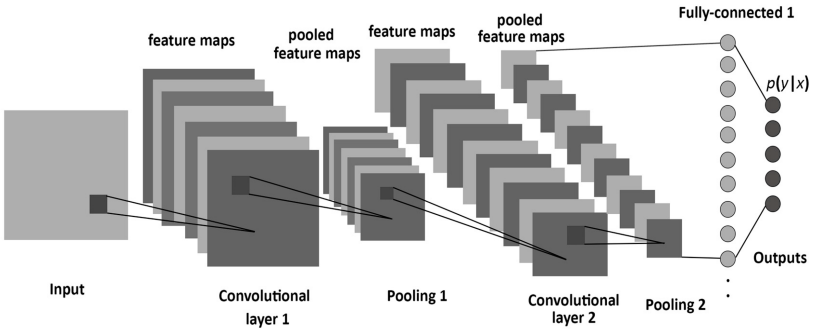

Figure 1.1: The architecture of a CNN

1.1.3 Q-Learning

It is one of the most used algorithm in reinforcement learning, his goal is to learn a good policy (eventually is optimal) to solve a problem modeled as MDP. Q is the function that returns the reward, (q stands for quality of an action). Obviously it works well in a finite space, but we can extend to a continuos space combining Q with a function approximation.

$$Q^{new}(s_t, a_t) \leftarrow (1 - \alpha) * Q(s_t, a_t) + \alpha * (r_t + \gamma * max_a Q(s_{t+1}, a)) \tag{1.2}$$

Deep Q-Learning

Using a nonlinear function approximator such as a NN to represent Q it introduces a kind of instability given from the correlation of observations. In order to face the problem is used a technique called **experience-replay:**

> *"This removes correlations in the observation sequence and smooths changes in the data distribution. Iterative update adjusts Q towards target values that are only periodically updated, further reducing correlations with the target."*[2]

Chapter 2

Setup of the environment

By the following lines, we want to show how the project has been developed underlining the libraries,the tools and the sources that have been used in its implementation.

2.1 System configurations

2.1.1 Hardware used

The code has been developed and tested on two different machines with different configurations:

1. **First machine:** mounting a **CPU** *Intel Core i5-7200u* .
2. **Second machine:** mounting a **CPU** *Intel Core i7-6700HQ* and a dedicated **GPU** *Nvidia GTX960M* .

The heterogeneity of the components listed above has allowed greater generalization of the code developed in such a way as to fully exploit the hardware that is available during the training and test phases of the agent.
Each machine uses a GNU-Linux-based operating system.

2.1.2 Software used

The project has been fully developed using *python-2.7* as source code language, utilizing *PyCharm*[3] as the main IDE for launching and deploying the code.
Project's deployments and updates took place by utilizing the services provided by the site *BitBucket*[4].
In order to operate in a controlled environment under similar conditions, it has also been used *Anaconda* and in particular the *conda*[5] command to setup the additional python libraries required by the project.
Where possible, it has also been installed *CUDA*(9.2) in order to take advantage of the graphic card during the computations.

2.2 Environment configuration

As described before, an *Anaconda* environment has been set up on each system, defining *python-2.7* as the default version of the python compiler that had to be installed and used for each code execution.
A new environment set and start up can be achieved by launching in a terminal:

Listing 2.1: Environment creation and activation,

```
#!/bin/bash
source /.../activate root
conda create -n envname python=2.7
conda activate envname
```

where *envname* is the desired environment name and the last command-line allows the execution of the later launched commands directly in the environment. Further installations of the required libraries, that will be described one by one in the next subsection, took place by utilizing directly conda or by using *Python-pip*, a package management system used to install and manage software packages written in Python[6].
A generalization of the commands launched to install the additional software can be summarized in the following commands:

Listing 2.2: Conda package installation,

```
#!/bin/bash
conda install [-c] [optional-repository] packagename
```

Listing 2.3: Python-pip package installation.

```
#!/bin/bash
pip install packagename
```

Some of the packages also required some local compilation obtained by using *C* compilers, such as *cmake*, *make*, *gcc* and *g++*, previously residing on the machine.

2.2.1 Main External libraries

The following are described the external libraries used in the project:

- **Boost and Boost-python**: a set of libraries for the C++ programming language that provide support for tasks and structures such as linear algebra, pseudorandom number generation, multithreading, image processing, regular expressions, and unit testing.[7]
- **OpenAI Gym***(gym)*: a toolkit for developing and comparing reinforcement learning algorithms. It makes no assumptions about the structure of the agent, exposing via its framework a wide and expandable collection of testing enviroments that easily process, simulate and return the obtained observations and rewards of a submitted action.[9]

  - **gym[atari]**: a sub-packet of gym containing all enviroments related to the atari-games. Can be also addressed to **atari-py**.
  - **gym-super-mario-bros**: an *OpenAI Gym* environment for Super Mario Bros and Super Mario Bros 2 (Lost Levels) on The Nintendo Entertainment System *(NES)* using the **nes-py** emulator.[10]
  - **ppaquette-gym-doom**: an environment bundle for OpenAI Gym based on the 1993 Doom game engine, using old version of VizDoom. In particular the package collects, compiles and installs also the **doom-py** package, that works as the core of the simulation engine.[11]

- **Numpy**: a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.[8]

- **Pytorch** : an open source machine learning library for Python, based on *Torch* (**torch** and **torchvision** packages), used for applications such as natural language processing[12]. Pytorch comes with different releases based on the version of Python, the operating system and in particular on the presence of *CUDA* on the machine and its version. If the CUDA version is installed, also the packages **cudatoolkit** and **cudann** have to be installed for the correct functioning of the extension.

- **SciPy**: a free and open-source Python library used for scientific computing and technical computing. In particular contains modules for optimization, linear algebra, integration, interpolation, special functions, signal and image processing.[13]

Chapter 3

Solution in practice

By the following subsections there will be introduced and commented parts of the attached code that have a significant relevance in the learning process of the AI.
The rest of the code of the project, especially if discussed previously, will be presented in a more general way by following the source files content.

3.1 Pre-existing code

Some parts of the solution are just a reuse of code, this is the section where we'll have an overview of this code. Most of the reused code comes from the SuperDataScience website where they tried to solve problems similar to ours using the same technique.

Experience Replay

[14] It is a standard technique used when a CNN solves a reinforcement learning problems. The way how it reaches the results is giving the Neural Network a kind of memory, so it keeps a list of frames and evaluates the performaces basing on them. The most common implementations uses a list to keep fragments and performs a his evaluation on a mini-batch taken randomly.

3.1.1 Asyncronous N Step

[14] This is the algorithm on the paper and the one we are going to implement, the idea is simple, instead of evaluate the q function value on a single event, we evaluate on N events, it improves drammatically the efficency giving even better performances. The only difference with the paper is the use of the softMax function to choose the action to take instead the ϵ-greedy function.

3.2 Image preprocessing

It is a simple function that takes the image as a pixel matrix and elaborates it as the CNN wants, making that of the right size and setting if it has to be coloured or in a grayscale

3.3 Implemented code

3.3.1 ai_utils.py

This is the file which acts as a container of utility classes of objects used mostly by the methods, there are some utilities for plot drawing, but the more relevant for the logic are those three:

- **CNN**
- **AI**
- **SoftmaxSelector**

CNN

This is the class used to actually build the Neural Network so the method by method analysis is:

- The constructor takes as input two arrays and builds two *ModuleList* in order to make the network as modular as possible.
- The forward function builds the real neural network.
- The goal of the function *count neurons* is just to count how many neurons are the output of the last convolutional layer.

```
class CNN(nn.Module):

    def __init__(self, cnn_layers, dnn_layers):
        super(CNN, self).__init__()
        self.convolution = nn.ModuleList(cnn_layers)
        self.fullyconnected=nn.ModuleList(dnn_layers)

    def count_neurons(self, image_dim):
        x = Variable(torch.rand(1, *image_dim))
        for layer in self.convolution:
            x = layer(x)
        size =  x.data.view(1, -1).size(1)
        return size

    def forward(self, x):
        for layer in self.convolution:
            x = layer(x)
        for layer in self.fullyconnected:
            x = layer(x)
        return x
```

Figure 3.1: The code of CNN class.

SoftMax Selector

This class is given to get a real value in the interval $[0, 1]$ as the output, the value is read as the probability of the state to be the best one, so it can be used as the probability that the considered state is the next one. We can also see the T value that represents the *temperature*, in other words is the likelihood of the system to make Exploration.

```
class SoftmaxSelector(nn.Module):

    def __init__(self, T):
        super(SoftmaxSelector, self).__init__()
        self.T = T

    def forward(self, outputs):
        probs = F.softmax(outputs * self.T)
        actions = probs.multinomial(1)
        return actions
```

Figure 3.2: The code of SoftMaxSelector class.

AI

The AI merges the Neural network with the action selector. In the forward function, given an image it propagates the signals into the network and returns the actions. It makes its work combining the two previous forward functions

```
class AI:

    def __init__(self, nn, selector):
        self.nn = nn
        self.selector = selector

    def __call__(self, inputs):
        input = Variable(torch.from_numpy(np.array(inputs, dtype=np.float32)))
        output = self.nn(input)
        actions = self.selector(output)
        return actions.data.numpy()
```

Figure 3.3: The code of AI class.

Other support classes

The classes which are not previously discussed are:

- ***PlotDynamicUpdate***: offers methods to generate and update a dynamic chart during the execution of the code;
- ***Flatten***: implements a pyTorch module for a flatten layer in the convolutional network structure.
- ***Average_movements***: adds and maintains the list of rewards and calculates the average reward of the last elements over a fixed size.

3.3.2 ai.py

The source file exposes some main global variables that identifies test enviroments IDs and names for a general access in the application.
It also implements three major functions that compose the core of the agent configuration and execution:

- **env_selector**
- **asynchronous_n_step_eligibility_batch**
- **execute**

Env Selector

Figure 3.4: env_selector code.

A simple method that taken the environment ID, the selected frame-skip and some image parameters configures and returns a Gym environment object ready for the execution of the agent. It lays also the possibility of choosing a move-set if the environment admits more than one.

Asynchronous N-Step Eligibility Batch

The method is a python implementation of an algorithm[15] used to speed up the learning rate of an actor usually in an asynchronous deployment. In our code, having only a mono-thread implementation, the code is configured and performed in a synchronous setting.

```
def asynchronous_n_step_eligibility_batch(batch, cnn):
    gamma = 0.99
    inputs = []
    targets = []
    for series in batch:
        input = Variable(torch.from_numpy(np.array([series[0].state, series[-1].state], dtype=np.float32)))
        output = cnn(input)
        cumul_reward = 0.0 if series[-1].done else output[1].data.max()
        for step in reversed(series[:-1]):
            cumul_reward = step.reward + gamma * cumul_reward
        state = series[0].state
        target = output[0].data
        target[series[0].action] = cumul_reward
        inputs.append(state)
        targets.append(target)
    return torch.from_numpy(np.array(inputs, dtype=np.float32)), torch.stack(targets)
```

Figure 3.5: asynchronous_n_step_eligibility_batch code.

The function works by taking in input a *CNN* object, representing the neural network used by the agent, and a batch of several inputs and targets, returning in output sets of more refined inputs and targets that later on the AI will use to minimize the square distance between predictions and targets in order to select the best strategy to apply in the environment. The inputs in the output set are the base to obtain the full set of predictions by simulating them on the environment during this process.
The *for* cycle in the code takes from the batch in input each series of transitions and takes the inputs states of the first and last transitions of the subset, storing them in a *numpy.array* that then is converted first in a *torch.tensor* and then in a more general *torch.Variable* (*input* variable) for further calculations.
With the obtained input the predictions are computed by storing them in the variable *output* as output signals of the *CNN* fed with the *input*.
After that the cumulative reward is defined checking whether the state of the final step is terminal or not and then is updated basing the choice on each step of the reversed series of transitions and by applying a factor *gamma*, here defined as 0.99, on the previous cumulative reward while adding the current step reward.
When the cumulative reward computation finishes for the series, the input and the prediction of the first step are memorized(*state* and *target* variables) and the target obtained by the specific action of the first state is updated with the cumulative reward.
Finally the first input step and the updated first target step are appended to the outputs, passing then to the next calculations on the next series in the batch.

3.3.3 custom_nns.py

The file exposes two major functions *createSequentialLayer* and *makeNN* that dinamically create the pyTorch modules and in the final phase the convolutional neural network that would be used in order to gain the features from the input image obtained from the environment during the training of the AI.
The entire process is based on the analysis of a string input, representing the layer, and other parameters that compose the selected features. An entire layer can be composed by multiple functions in order to obtain a certain number of

features in output, so the input for a single layer is passed via a list of lists, where each element represents a pyTorch module.
There can be also underlined that the CNN generation tuning is also based on the input image dimensions, that can be varied maintaining the same module structure. Input dimensions changes affects the number of neurons that are obtained in output from the convolutional layers and are the input of the the fully connected ones. By this fact, when the construction of the first fully connected layer takes place, it is left the possibility of calculating the number of its input features by simulating the output of a dummy network that takes a generated input based on the wanted image dimensions. This is achieved by utilizing the function *count_neurons* exposed in the *CNN* class in **ai_utils.py** file.

3.3.4 default_nns.py

The source file contains four methods that define and return four different default convolutional neural networks that have been used during the testing phase of the agent.

```
def default_cnn1(number_actions, image_dim):
    convolution1 = [
        ["Conv2d-1", image_dim[0], 32, 5],
        ["ReLU-1"],
        ["MaxPool2d-1", 3, 2]
    ]
    convolution2 = [
        ["Conv2d-2", 32, 32, 3],
        ["ReLU-2"],
        ["MaxPool2d-2", 3, 2]
    ]
    convolution3 = [
        ["Conv2d-3", 32, 64, 2],
        ["ReLU-3"],
        ["MaxPool2d-3", 3, 2]
    ]
    fc1 = [
        ["Flatten-1"],
        ["Linear-1", -1, 40],
        ["ReLU-1"]
    ]
    fc2 = [
        ["Linear-2", 40, number_actions],
    ]
    return makeNN([convolution1, convolution2, convolution3], [fc1, fc2], image_dim)
```

Figure 3.6: Example of default neural network construction.

Each function returns a *CNN* object of the **ai_utils.py** source file obtained by the conversion of the textual lists of parameters using the *makeNN* function of **custom_nns.py** file.

3.3.5 main.py

It's the main file of the package that have to be launched to execute the agent. It shows all the possible parameters that can be set to launch the environments and execute the tests:

The most pecular aspect of the main function is the possibility gained by

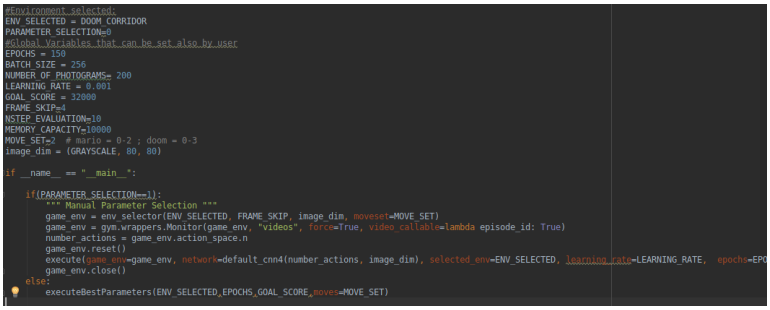

Figure 3.7: Content of **main.py** file.

the value of the variable *PARAMETER_SELECTION* that gives the possibility of execute the code with manually selected parameters or with optimal tuned parameters, following the directives contained in **parameters_auto_tuning.py** file.

3.3.6 parameters_auto_tuning.py

The main purpose of the tools and methods written in the file is to choose the best combination of tuning parameters for the agent.
The main function is *executeBestParameters*.

Figure 3.8: Function *executeBestParameters*.

As we can see from the code, the method simply check if a list of parameters does exist from a previous launch or otherwise starts to calculate it and finally applies the result to the execution of the AI.
The choice of parameters takes place in the *test_parameters* function:
where every combination of the established parameters is tested and the one

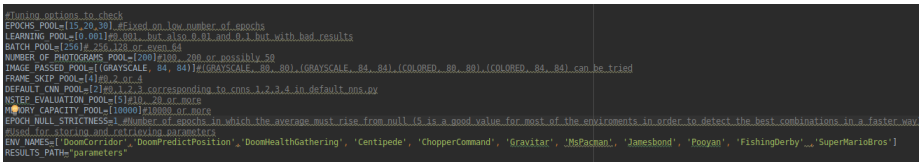

Figure 3.9: Parameters' pools,

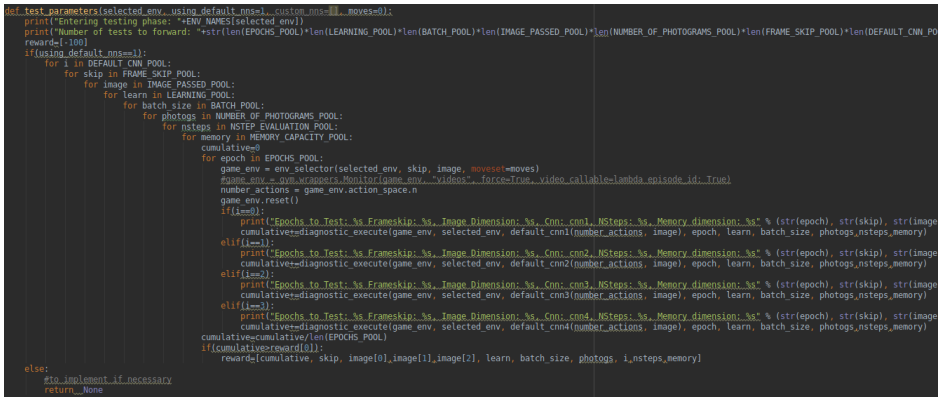

Figure 3.10: Parameters tries and selection.

with the maximum average reward on a fixed set of tries with different length in epochs is returned as the best choice. It follows that more parameters have to be tested, more time it would take to complete the evaluations. The function uses *diagnostic_execute* to simulate an execution of the agent in the environment in a similar way of the one that can be seen in the **ai.py** file, without generating charts and videos of the training and returning the final average reward of the run.

Chapter 4

The tests

By the following sections will be described the tests and the results that have been obtained by executing the AI on three different environments.

4.1 Doom-Corridor

The first game we are going to test is Doom, it is extremely easy and it's the good starting point to see if our code works. Due to its simplicity, the majority of our tests give us a positive result, in fact all of them complete the game in no more than 30 epochs.
In this case is interesting to show the performance tuning the image preprocessing, in particular, the listed tests are two:

1. Colored image
2. Grayscale image

The only CNN we are going to use in this case is the default one number one into the *default_nns.py* file, as we see it's composed by 5 layers.
The parameters of the test are:

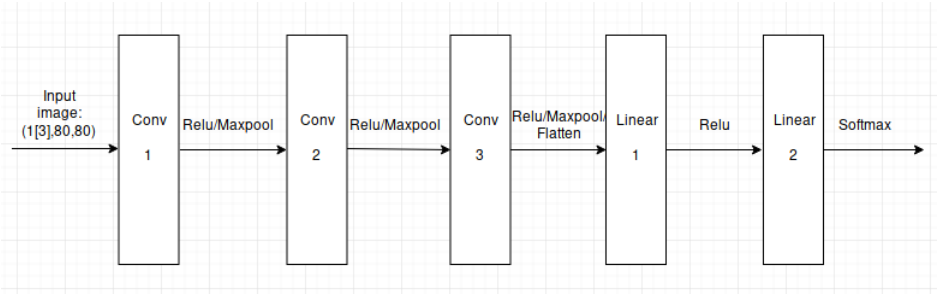

Figure 4.1: The architecture of the CNN

- BATCH_SIZE = 256
- NUMBER_OF_PHOTOGRAMS= 200
- LEARNING_RATE = 0.001
- FRAME_SKIP=4
- NSTEP_EVALUATION=10
- MEMORY_CAPACITY=10000
- COLOR = GRAYSCALE-COLORED

Using these parameters the AI finishes the level after 16 epochs in average, it needs about 1500 as reward for finishing the level, but as we can see, it improves his rewards, and after 30 epochs it's higher than 2100.

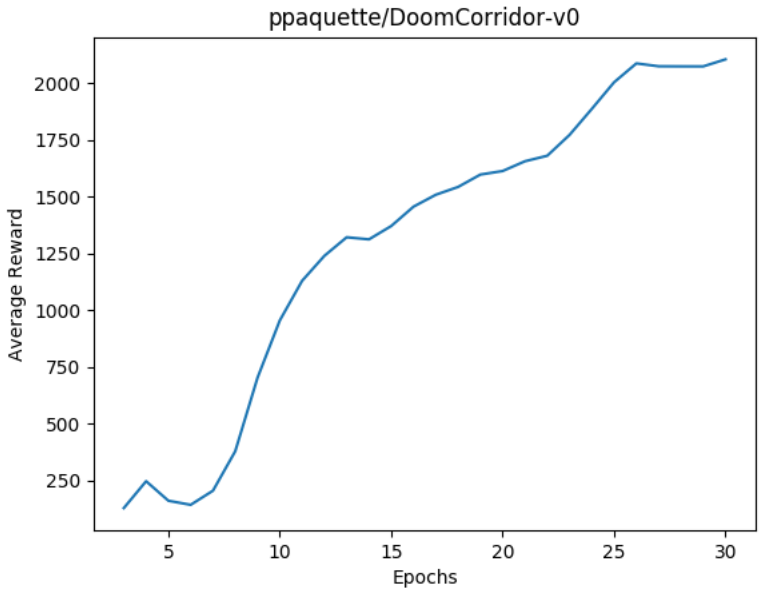

Figure 4.2: The plot of the doom corridor's results in grayscale

Let's make the same test, but using colors.

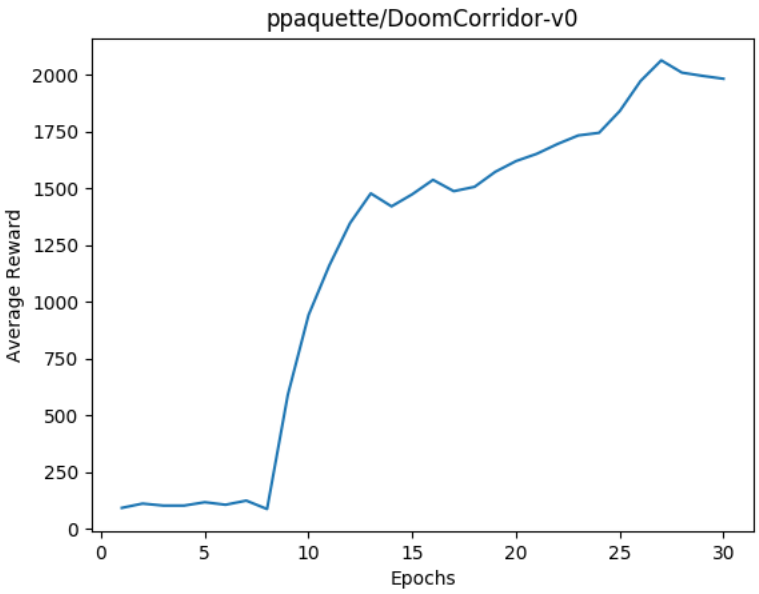

Figure 4.3: The plot of the doom corridor's results with colors

As we can see, the result in this case doesn't change, but what changes is the computing time, in this case, the cpu needs twice the case of grayscale.
Looking into the videos directory, is interesting to see what choice the AI makes to finish this level, in fact, during the exploration phase, it realizes that the best way to maximize the reward is to run until the end without waste a single bullet.
The video of some samples is at the following link:
`https://youtube.com/watch?v=A-qAn2ATwdU`

4.2 Centipede

The second test has been done on the *Atari* game *Centipede*, based on an environment where the player has three lives and the target of maximizing the score by shooting most enemies as possible with four grades of freedom in the movement on the screen.
Unlike the previous test a reward goal isn't provided, thus tests took place under 100 epochs in order to take information about the evolution in the agent's choices.
As before, tests were taken under two different tuning of the image processing:

1. Colored image
2. Grayscale image

The CNN we are going to use in this case is the default one number two into the *default_nns.py* file, as we see it's composed by 4 layers. The parameters of

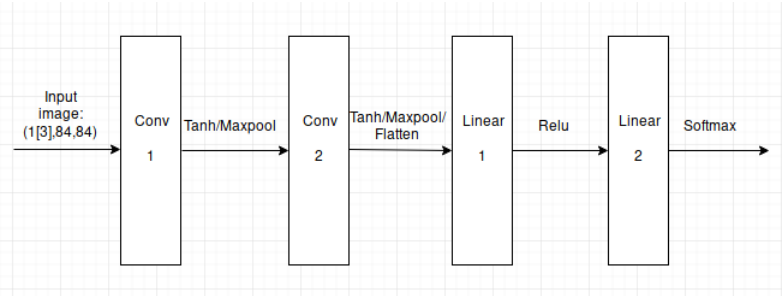

Figure 4.4: The architecture of the CNN

the test are:

- BATCH_SIZE = 128
- NUMBER_OF_PHOTOGRAMS= 200
- LEARNING_RATE = 0.001
- FRAME_SKIP=4
- NSTEP_EVALUATION=10
- MEMORY_CAPACITY=10000
- COLOR = GRAYSCALE-COLORED

Using these parameters the AI around after 20 epochs in average starts on improving the average reward reaching an good final result, especially with the grayscale input image where the final result outclasses also the highest peak obtained during the first random phases. We can also observe that the average reward exposes a same value maintained during some epochs. This takes place because a game doesn't end during an epoch with the passed batch of actions, and so the average reward doesn't change until the player loses and restarts a new game.
In the plots is also shown the development of the average loss around each epoch. In both the tests it appears to have initially an high peak and then drastically decreases to certain range of lower values that mantain a slow increasing rate parallel to the average reward's one.

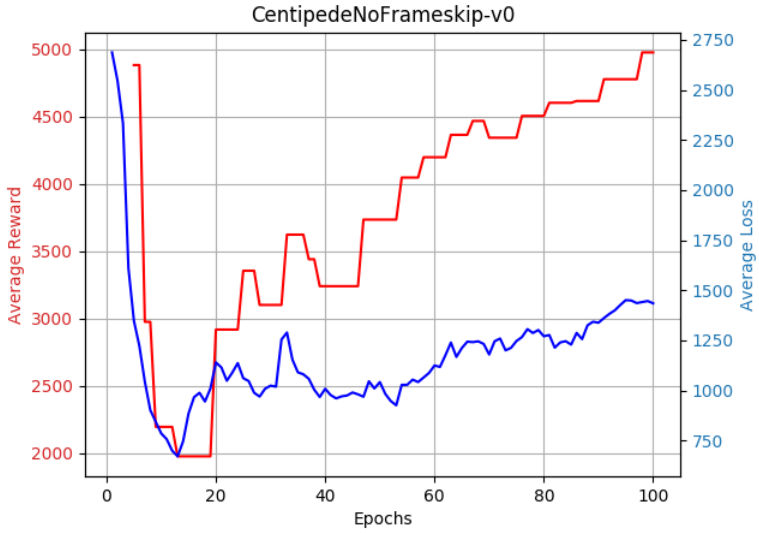

Figure 4.5: The plot of the centipede's results in grayscale

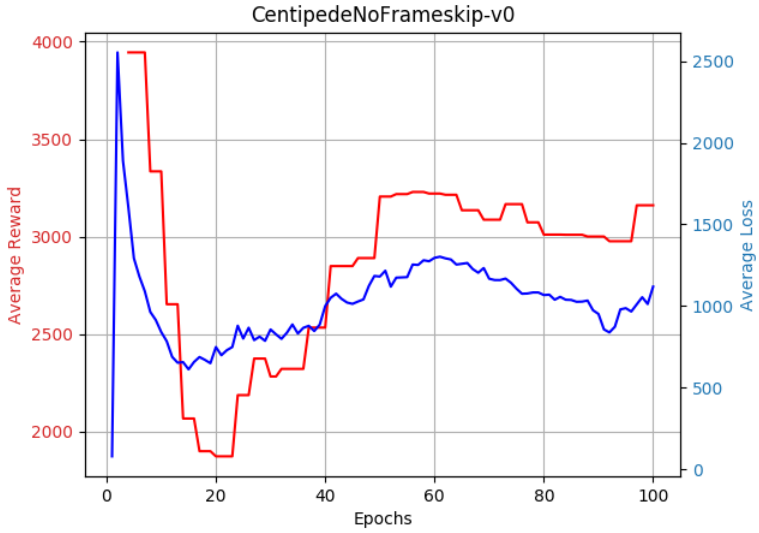

Figure 4.6: The plot of the centipede's results with colors

From the results we can deduce that with the same type of parameters the grayscale input gives an higher difference in the evolution between the average reward and the average loss if compared to the colored input's one. We can also

say that the results obtained from a grayscale input in the last epoch are far higher than the ones of the colored counterpart.

A peculiar behavior shown in both cases by the agent after a certain amount of epochs is to position of the player in one of the corners of the screen from which only little movements are made in order to avoid the enemies. The video of the test is available at the following link:
`https://youtu.be/B4wFsZ3ke1E`

4.3 Super Mario

The third test has been perpetrated on the first level of *Super Mario Bros*, modifying the default environment library in a way that the player has only one life and the obtained reward reduces the death penalties having traveled a greater distance, thus favoring the achievement of more distant targets compared to the beginning of the level and speeding up learning. Compared to the previous environments, the need for longer times has been recorded in the various tests so that the agent actually starts to improve the average reward, having also longer times during the image processing in the neural network. Since a reward goal wasn't provided from the library and for the previous causes, tests took place under 100 epochs like under the previous environment.
As before, tests were taken under two different tuning of the image processing:

1. Colored image
2. Grayscale image

The CNN we are going to use in this case is the default one number three into the *default_nns.py* file, as we see it's composed by 4 layers. The parameters of

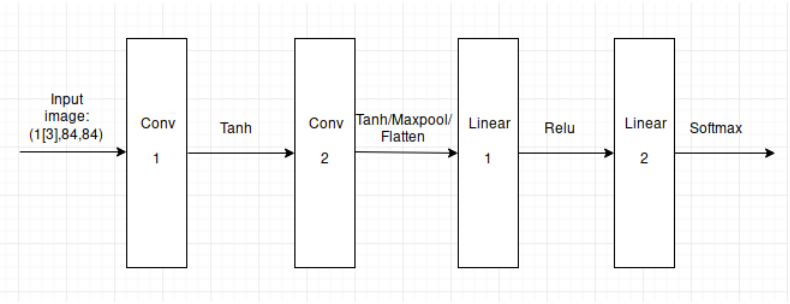

Figure 4.7: The architecture of the CNN

the test are:

- BATCH_SIZE = 512
- NUMBER_OF_PHOTOGRAMS= 400

- LEARNING_RATE = 0.001
- FRAME_SKIP=6
- NSTEP_EVALUATION=5
- MEMORY_CAPACITY=100000
- COLOR = GRAYSCALE-COLORED

During these tests also the **Temperature** parameter related to the Softmax function has been tuned to a greater value than the default one (from 1.0 to 5.0) in order to raise during the execution the probability of choice of actions with lower expected reward preventing that the system gets stuck for high periods of time on a certain sequence of actions.
Using these parameters the AI around the first 60 epochs in the average starts on improving the average reward reaching a medium point between the lowest value and the highest peak in the last epoch.
In the plots it is also shown the development of the average loss around each epoch. As we can see after a peak during the exploration phase with random choices in the actions selected the average loss takes a drastic fall in the reported values and after that maintains a constant value.
It can be deduced from the charts that when the loss is maintained at almost stable values the agent has the possibility to increase the average reward as this condition also manifests around 60 epochs.

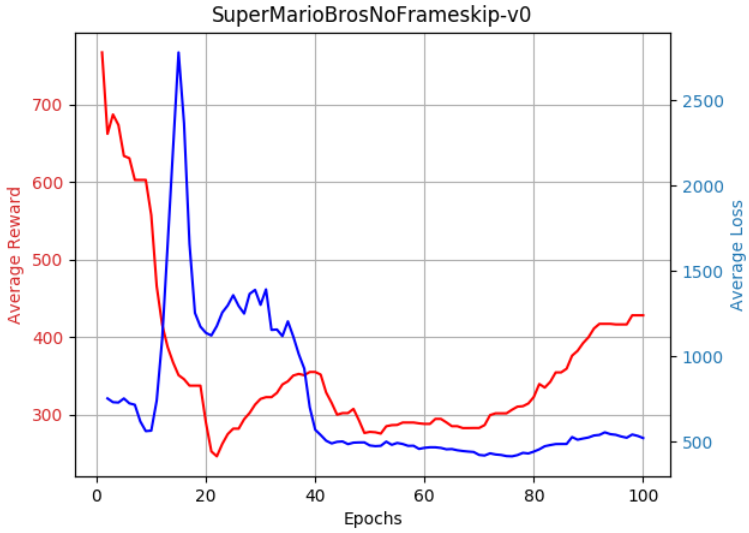

Figure 4.8: The plot of the super mario's results in grayscale

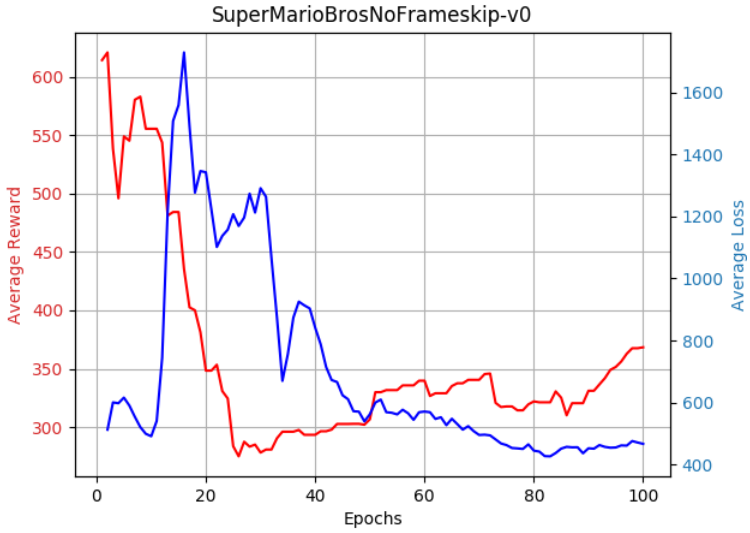

Figure 4.9: The plot of the super mario's results with colors

As we can see, the result in both cases doesn't change, but what changes is the computing time of the machine, also using *CUDA*, that raises about a 5% in the case of color.

The video of the test is available at the following link:
`https://youtu.be/j2AKAIspfow`

Bibliography

[1] Wikipedia, *Markov Decision Proces*, https://en.wikipedia.org/wiki/Markov_decision_process

[2] Wikipedia, *Q-Learning*, https://en.wikipedia.org/wiki/Q-learning

[3] JetBrains, *PyCharm,* https://www.jetbrains.com/pycharm/

[4] BitBucket, https://bitbucket.org/

[5] Conda package manager, https://anaconda.org/anaconda/conda

[6] GitHub, *pip(package manager)*, https://github.com/pypa/pip

[7] Wikipedia, *Boost*, https://en.wikipedia.org/wiki/Boost_(C\%2B\%2B_libraries)

[8] Wikipedia, *Numpy*, https://en.wikipedia.org/wiki/NumPy

[9] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[10] Christian Kauten. Super Mario Bros for OpenAI Gym. https://github.com/Kautenja/gym-super-mario-bros, 2018.

[11] GitHub, *ppaquette-gym-doom*, https://github.com/ppaquette/gym-doom

[12] Wikipedia, *Pytorch*, https://en.wikipedia.org/wiki/PyTorch

[13] Wikipedia, *SciPy*, https://en.wikipedia.org/wiki/SciPy

[14] SuperDatascience, *superdatascience*, https://www.superdatascience.com/artificial-intelligence/

[15] *Asynchronous Methods for Deep Reinforcement Learning*, Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, Koray Kavukcuoglu, 16 June 2016, page 13, https://arxiv.org/pdf/1602.01783.pdf