Assignment 3

Author: **Leonardo Colosi 1799057**

Main Contributors: **Bruno Francesco Nocera 1863075, Paolo Renzi 1887793**

Other Contributors: **Silverio Manganaro 1817504, Jacopo Tedeschi 1882789**

MARR, RL

December 14, 2023

Contents

1 Theory

1.1 Exercise

In the context of this exercise, we need to consider an environment that allows two possible actions and features a two-dimensional state representation, denoted as $x(s) \in R^2$. This scenario involves the application of the 1-step Actor-Critic Algorithm with specific policy and action-state value function approximators:

$$\begin{aligned}\pi_\theta(a=1|s) &= \sigma(\theta^T x(s)) = \frac{1}{1 + e^{-(\theta^T x(s))}}\\ Q_w(s,a=0) &= w_0^T x(s)\\ Q_w(s,a=1) &= w_1^T x(s)\end{aligned}$$

Give the initial values for the weights and the values of the hyper-parameters:

$$\begin{aligned}w_0 = (0.8,\ 1)^T&,\ w_1 = (0.4, 0)^T\\ \theta_0 &= (1, 0.5)^T\\ \alpha_w = \alpha_\theta &= \alpha = 0.1\\ \gamma &= 0.9\end{aligned}$$

As well as a defined transition:

$$x(s_0) = (1,0)^T,\ a_0 = 0,\ r_1 = 0,\ x(s_1) = (0,1)^T,\ a_1 = 1$$

The task is to compute new values of w_0, w_1 and θ after the given transition. To do this we have to follow the steps of the *Q Actor-Critic* update method where:

1. The weights of the "Critic" network **w** are updated to minimize the TD error;
2. The weights of the "Actor" network θ are updated in the direction suggested by the Critic.

As first step we must compute the TD error, this can be done by the usual formula:

$$\delta = r + \gamma Q_w(s',\ a') - Q_w(s,\ a) \tag{1}$$

In this case $(s,\ a)$ and $(s',\ a')$ are given as $x(s_0)$, a_0 and $x(s_1)$, a_1. Now we have to evaluate the Q-function approximations for the given state-action pairs:

$$Q_w(s,a=0) = w_0^T x(s) = \begin{pmatrix}0.8 & 1\end{pmatrix}\begin{pmatrix}1\\0\end{pmatrix} = 0.8$$

$$Q_w(s,a=1) = w_1^T x(s) = \begin{pmatrix}0.4 & 0\end{pmatrix}\begin{pmatrix}0\\1\end{pmatrix} = 0$$

Now we have all the numerical value to substitute in 1, after doing this we obtain:

$$\delta = 0 \,+\, 0.9 \cdot (0) \,-\, 0.8 \,=\, -0.8$$

For the update of the weight $\mathbf{w}$ we should compute the gradient of Q w.r.t. $\mathbf{w}$ itself. We can write the general formula as:

$$\mathbf{w} \,=\, \mathbf{w} \,+\, \alpha\gamma\nabla_w Q_w(s, a) \tag{2}$$

It is possible to decuple the computation of the new weights in two different equations:

$$w_0 \,=\, w_0 \,+\, \alpha\gamma\nabla_w Q_{w_0}(x(s_0), a_0) \,=\, \begin{pmatrix}0.8\\1\end{pmatrix} \,-\, 0.8 \cdot 0.1 \cdot \begin{pmatrix}1\\0\end{pmatrix} \,=\, \begin{pmatrix}0.72\\1\end{pmatrix}$$

Regarding the second equation, the gradient with respect to w_1 of $Q(x(s), a_1)$ would be used to update w_1 only if the action $a_1 = 0$ was taken at state $x(s_0)$. In the given problem statement, since the action $a_0 = 0$ was taken at state $x(s)$, we would not use the gradient of $Q(x(s), a_1)$ for the update in this step. Using instead the gradient of $Q(x(s), a_0)$ lead to no change for the vector w_1:

$$w_1 \,=\, w_1 \,+\, \alpha\gamma\nabla_w Q_{w_1}(x(s), a_0) \,=\, \begin{pmatrix}0.4\\0\end{pmatrix} \,-\, 0.8 \cdot 0.1 \cdot \begin{pmatrix}0\\0\end{pmatrix} \,=\, \begin{pmatrix}0.4\\0\end{pmatrix}$$

In certain variations of the Actor-Critic algorithm, one might update both w_0 and w_1 using the respective gradients even if one of the associated actions was not taken, as part of an off-policy learning method. This choice would lead to the following update:

$$w_1 \,=\, w_1 \,+\, \alpha\gamma\nabla_w Q_{w_1}(x(s_1), a_1) \,=\, \begin{pmatrix}0.4\\0\end{pmatrix} \,-\, 0.8 \cdot 0.1 \cdot \begin{pmatrix}0\\1\end{pmatrix} \,=\, \begin{pmatrix}0.4\\-0.08\end{pmatrix}$$

Finally we can compute the new value of θ according to:

$$\theta \,=\, \theta \,+\, \alpha\delta\nabla_\theta \,\log \pi_\theta(a|s) \tag{3}$$

In particular we need the expression for $\pi_\theta(a \,=\, 0|s)$ in order to calculate the gradient. To obtain this expression we could make the observation that, since the action space is binary and π represent a probability distribution over actions, $\pi_\theta(a = 0|s) \,=\, 1 \,-\, \pi_\theta(a = 1|s)$.

Established this we can proceed with the computation of the gradient:

$$\nabla_\theta \, \log(1 - \pi_\theta(a = 1|x(s_0))) =$$

$$\nabla_\theta \, \log\left(1 - \frac{1}{1+e^y}\right) = \nabla_\theta \, \log\left(\frac{1+e^y-1}{1+e^y}\right) = \qquad \text{where } \; y = -(\theta^T x(s))...$$

$$\nabla_\theta \, \log\left(\frac{e^y}{1+e^y}\right) = \nabla_\theta \, \log(e^y) \; - \; \nabla_\theta \, \log\left(\frac{e^y}{1+e^y}\right) = \quad \text{...splitting the logarithm ...}$$

$$\nabla_\theta y \; - \; \nabla_\theta \, \log(1+e^y) = \qquad \text{...substituting y...}$$

$$-x(s) \; + \; \frac{x(s)\cdot e^{-(\theta^T x(s))}}{1+e^{-(\theta^T x(s))}} = \qquad \text{...and taking the gradient.}$$

$$\frac{-x(s) - x(s)\cdot e^{-(\theta^T x(s))} + x(s)\cdot e^{-(\theta^T x(s))}}{1+e^{-(\theta^T x(s))}} = \qquad \text{Evaluating in } \; x(s_0), \; \theta_0...$$

$$\frac{-x(s)}{1+e^{-(\theta^T x(s))}} \; = \; -\begin{pmatrix}0.73\\0\end{pmatrix}$$

To obtain the final evaluation we should take in account that:

- $\theta^T x(s) \; = \; 1;$
- $1 + e^{-1} \; = \; 1.36;$
- $\frac{1}{1+e^{-1}} \; = \; 0.73.$

The new value for θ is given by:

$$\theta \; = \; \theta_0 \; + \; \alpha\delta\nabla_\theta \, \log\pi_\theta(a = 0|x(s_0)) \; = \; \begin{pmatrix}1\\0.5\end{pmatrix} \; + \; 0.1\cdot 0.8\cdot \begin{pmatrix}0.73\\0\end{pmatrix} \; = \; \begin{pmatrix}1.05\\0.5\end{pmatrix}$$

2 World Models

2.1 Task Overview

The task of this assignment was to solve the CarRacing-v2 gym environment using one of the proposed algorithms. A complete description of the environment is reported on gym car-racing website. The generated track is random every episode, the observation states consists of 96x96 pixels images. The reward is -0.1 every frame and $+1000/N$ for every track tile visited, where N is the total number of tiles visited in the track. This means that agent would receive higher reward for completing the track in a shorter amount of time, this also means that in general moving is better than do nothing (even if sometimes it means to move off-track). From this observation we can conclude that the reward is not a perfect metric to evaluate the performance of the car but it's still a good way to influence its behavior.

2.2 Solution Strategy

The strategy chosen among the proposed solution is the one presented on *World Models* [2]. I, along with my colleagues listed as main Contributors, have implemented a simplified version of the model proposed in the paper. The main idea expressed in their work is to combine three different models (see 1) in order to obtain the best performance from the agent.

- A Variational Auto Encoder (V) used for feature extraction and state representation (z);
- A MDN-RNN (M) as a predictive model of the future z vectors that V is expected to produce;
- A Controller responsible for selecting the action for the agent given the current observation (z) and the temporal information (h) given by M.

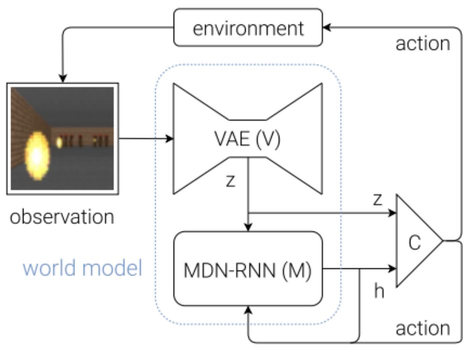


Figure 1

We have decide to reduce the size of the model by removing the M module. This choice was due to two main reasons. The firs one is that there are experimental results that shows that for this specific environment a good compression of the observation is way more relevant that the temporal "compression" provided by the RNN. The second reason relies in the complexity of efficiently training a Mixture Density Network combined with an Long Short Term Memory module. The rest of the implementation is build around the Variational Auto Encoder that we have coded from scratch and the CMA-ES training strategy for the Controller for which we have took inspiration from a Github repository that has implemented as similar version of World Models [1].

2.3 Code Structure

The assignment directory is organized in modular components according to the implementation logic of World Models. Here it is a summary of the general structure:

- **checkpoints**: shared directory to store the state dictionary of the various components of the model;
- **dataset**: local directory to store a collection of observation used to train the VAE;
- **modules**: contains the implementation of the architecture modules;
- **train**: is a directory that contains the files necessary to run the training of the NN modules;
- **utils**: is a collection of function used to perform random rollout, to gather and manipulate environment data and to handle the controller parameters;
- *main.py*: the main file for the execution of both training and evaluation of the entire model (unchanged);
- *student.py*: this file contains the implementation of the act function for the agent (*Policy*) as well as the main function to train the controller.

In the following sections is reported a detailed explanation of the most relevant components of the project.

2.4 Data Collection

The task of collecting random frames from the environment, in order to give the agent a meaningful, generalized representation of the world, is handled by *collect_data.py*. This script is responsible of performing random rollouts, moving the agent in various direction and collecting a the desired number of frames (as long as the termination state is not encountered before). The random rollouts could be performed both in continuous and discrete settings, what change between the two cases is how the random policy select the actions.

- *Discrete case*: sample one random int form {1,2,3,4} → {do nothing, steer left, steer right, gas, brake}, in this case the action {0: do nothing} is discarded because it would be of no use to collect meaningful observations.
- *Continuous case*: sample one action from the continuous action space (dim=3) of the environment. To action is then added a weighted random sample extracted from a normal distribution to increase the change of performing "strange action in order to gather more generalized information.

All data are then saved inside the local directory dataset, this directory is not contained in the zip file because of its size. The collected data are the one used to train the Neural Networks components of the architecture.

2.5 Variational Auto Encoder

The Variational Auto Encoder (VAE) implemented for this assignment is a "vanilla" auto-encoder consisting of 3 *Convolutional* layers and one *Adaptive Pooling* for the encoder, two *Linear* layers for computing μ and σ and a combination of one *Linear* layer and 3 *Transposed Convolutional* for the decoder. The encoder reduce the dimensionality of the input images to a tensor of shape (1, 1025). This vector is then used to compute the latent space z of size 32, this is a good enough compression of the original frames. The only manipulation applied to the images was a normalization of the pixels value between 0 and 1. The lightweight nature of the network has shown advantages both in training and in the application. The model required a short amount of training time while showing good result in the image compression/reconstruction.

2.5.1 VAE Training Strategy

The training phase of the auto-encoder was carried on using different evaluation functions over the reconstructed images as inputs and the original images as target. The main loss used was the Kullback-Leibler Divergence (KDL) combined alteratively with and the Mean Square Error (MSE) and the Weighted Binary Cross Entropy (BCE). The former performed best in terms of loss values (2) but the latter perform better in terms of final images reconstruction (3).
Here are the expression of the functions:

$$\text{KLD} = -\frac{1}{2}\sum_{j=1}^{J}(1 + \log((\sigma_j)^2) - (\mu_j)^2 - (\sigma_j)^2) \tag{4}$$

$$\tag{5}$$

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{6}$$

$$\tag{7}$$

$$\text{BCE} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)] \tag{8}$$

The following plots show the evolution of the loss functions during training.

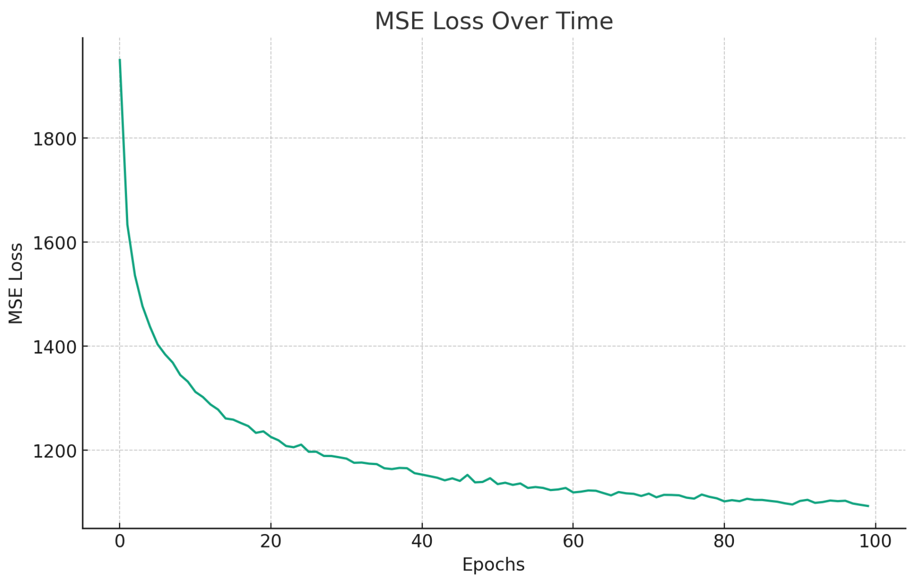


Figure 2

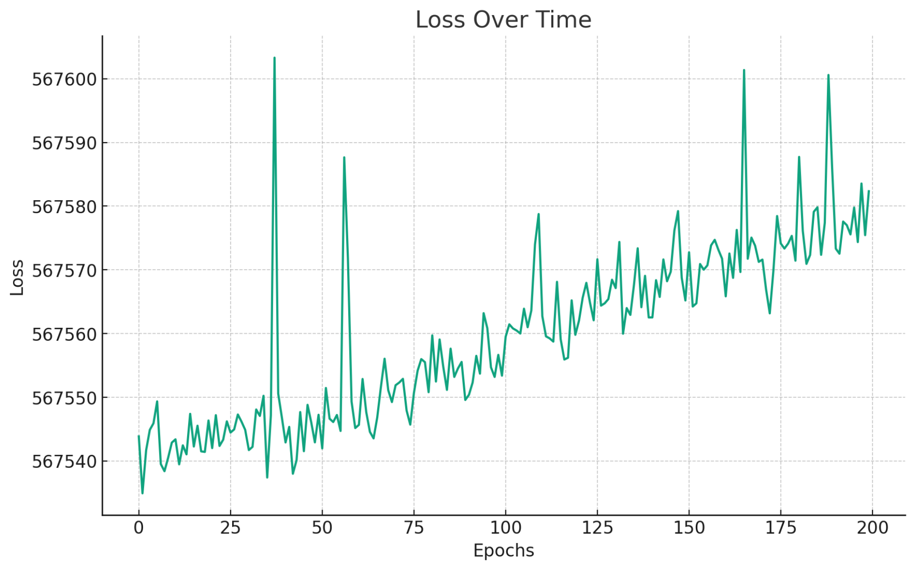


Figure 3

2.5.2 VAE Results

The following are the images reconstructed by the VAE. The original images are placed on the even columns while the reconstruction are on the odd columns. For the first set of images (4) it can be notice that when the track is straight the network is able to reconstruct it pretty well, capturing also the features of the red racing curb. On the other hand, when there is a hairpin turn the results are poor, leading to worst performance of the car in this environmental conditions.

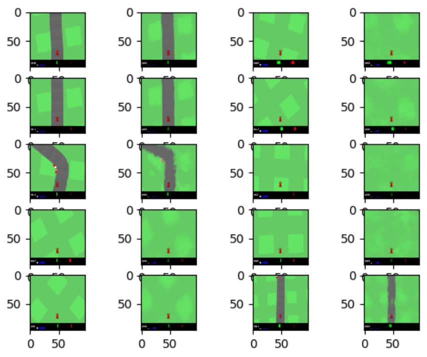


Figure 4: Frames reconstructed using a VAE trained with MSE loss.

About the second set (5) it is possible to see some improvement also with respect to the representation of hairpin turn. This has a huge impact on the final performance of the agent.

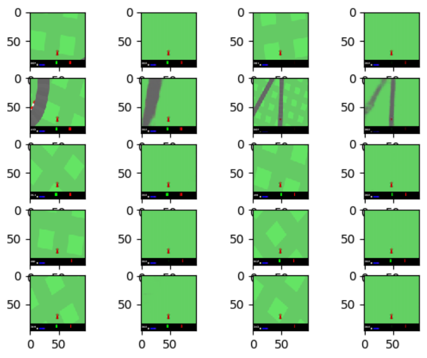


Figure 5: Frames reconstructed using a VAE trained with BCE loss.

Controller

The controller block is implemented as a simple Linear layer as suggested in the paper. There are two scripts inside the **modules** directory that represents two similar controllers, one to be used in a discrete environment (*discrete.py*) the other (*continuous.py*) for the continuous case. As we decide to not add the MDN-RNN module to the final architecture the dimensionality of the input expected by the controller coincide exactly with the size of the latent space of the VAE. The output is different for the two cases:

Discrete case: here the output is a 4-dimensional vector where each element represent the probability of choosing one action. Through the *argmax* function the index with the highest probability is selected. I have decide to crop the action-space (the discrete case has 5 possible action) in order to exclude the action do nothing, to do so a +1 is added to the output of *argmax*.

Continuous case: this case is straight forward, the output od the controller here is a 3dimensional vector, as it is the dimension of the action-space, which is direclty given to the environment step function.

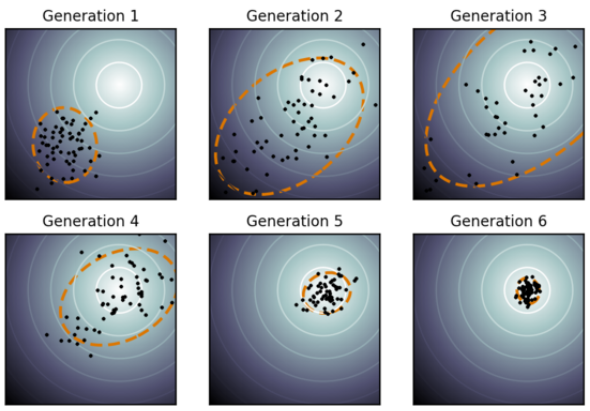
The core function of *student.py*, regarding the model evaluation, is *act*:

```python
def act(self, state):

    # convert input state to a torch tensor
    state = torch.tensor(state/255, dtype=torch.float)
    state = state.permute(0,2,1).permute(1,0,2)
    state = state.unsqueeze(0).to(self.device)

    # obs compression
    z = self.vae.get_latent(state.float())

    # get action from controller
    a = self.c(z).to(self.device)

    if not self.continuous:
        return (int(torch.argmax(a)) + 1)
    else:
        return a.cpu().float().squeeze().detach().numpy()
```

Here the policy select which action to do during a rollout of the simulation. The state observed from the environment is normalized and passed to the VAE, this will output a compressed, but still meaningful, version of the frame to the controller which return the action according to the environment set up.

2.6 Controller Training Strategy

Regarding the training of the controller module we have followed the same approach suggested by the original paper and have decided to implement the Covariance Matrix Adaptation Evolution Strategy. It's a method used for optimizing complex functions, often in situations where the function's form is not known in advance or is difficult to analyze. In our case the objective function to minimize was the difference between the max expected reward (1000) and the sum of cumulative rewards returned after the performance of a single rollout. CMA-ES doesn't make entirely random changes. Instead, it learns which changes are more likely to lead to better solutions[1]. In it is possible to see intuitively how the CMA algorithms shapes the data covariance Matrix in order to make the solution to converge to a minimum.



In order to apply this method we have imported in our code the *python cma* library. Here it is a simplified version of the evolution procedure:

```python
es = cma.CMAEvolutionStrategy(xo, sigma, pop_size)
while not es.stop():
    solutions = es.ask()
    es.tell(solutions, [cma.ff.rosen(s) for s in solutions
        ])
    es.disp()
es.result_pretty()
```

The parameters passed to CMAEvolutionStrategy represent:

- x0: initial solution, starting point, the current parameters of the controller;
- sigam: initial standard deviation, which will be changed by the algorithm;
- pop_size: the population size or number of solution computed for each generation.

The training process was scheduled following a series of steps in order to achieve the best possible results with limited resources. The bottleneck for the training of the entire model was in fact given by the training of this specific module. This is due to two main reason: one is the complexity of the genetic algorithm which is proportional to phe population size, the other is the heavy load on the cpu produced by the rollout necessary to compute the objective function. For those reasons I have decided to split the training in progressive phases iteratively saving the new best parameter and restarting the training from the saved checkpoints. The stop condition was defined by setting a a goal target for the mean cumulative reward. This approach has lead to a more manageable training and pretty good results in terms of performance. Furthermore

[1]In this context the solution returned by the cma algorithm are the controller parameters

in order to reduce the population size and average out the results I have set a fixed number of sample to take from each element of the population. This was the same method used in the code from [1] but in that context it was done with the purpose of combining the performance of multiple threads acting on the evaluation of the same solutions. In every case, since the environment is stochastic, a given solution could lead to much different results so the approach of taking more than one sample per solution seems reasonable.

2.7 Results

Below are listed the training parameters and hyper-parameters used to reach an average of 700 points as reward:

- environment setting: continuous;
- x0: initial controller parameters;
- sigma: 0.2;
- pop_size: 6
- n_sample: 3

With this training setup I needed to perform 3 consecutive run of CMA-ES, each run lasting ± 22 minutes, in order to obtain a mean cumulative reward of 700. This results has been achieve using the vae trained with the BCE loss. This results can be seen in video here.
Note: the model ist trained to works at best with a continuous environment.

References

[1] ctallec. “world-models”. In: *github repo* (). URL: `https://github.com/ctallec/world-models.git`.

[2] David Ha andJürgen Schmidhuber. “World Models”. In: *CoRR* abs/1803.10122 (2018). arXiv: `1803.10122`. URL: `http://arxiv.org/abs/1803.10122`.