Reinforcement Learning (2022/2023)

Assignment 3

Student Name: Gabriele G. Di Marzo
Student ID: 2012633

---

1. **REINFORCE**
following the pseudo-code of the REINFORCE algorithm, at each iteration (for t=1 to T-1) the update rule is:

$$\theta \leftarrow \theta + \alpha \nabla_\theta log(\pi_\theta(s_t, a_t))G_t \tag{1}$$

Considering the policy $\pi(a = 1|s) = \sigma(\theta^T x(s))$, we have first to calculate the gradient of $log(\sigma(\theta^T x(s))$ and of $log(1 - \sigma(\theta^T x(s)))$ (the second one for $\pi(a = 0|s)$).

$$\nabla_\theta log(\sigma(\theta^T x(s)) = \frac{x(s)}{e^{-\theta^T x(s)} + 1} \tag{2}$$

$$\nabla_\theta log(1 - \sigma(\theta^T x(s)) = \frac{-x(s)}{e^{-\theta^T x(s)} + 1} \tag{3}$$

From the assignment we have a $T = 2$, so we have 2 update, for $t = 0$ and $t = 1$.
The general form of the return G_t is:

$$\sum_{k=1}^{T} \gamma^{k-1} r_k \tag{4}$$

At $t = 0$ the return G_0 can be calculated as $\sum_{k=2}^{1} \gamma^{k-1} r_k = r_1 + \gamma r_2 = 0.9$.
At $t = 1$ G_1 has the value equals to $r_2 = 1$.

Finally, the total update for t=0, 1 can be calc using the data of the assignment, we get:

$$\theta_t = \begin{bmatrix} \theta_{1,t-1} \\ \theta_{2,t-1} \end{bmatrix} + \alpha \begin{bmatrix} x_1(s_t) \\ x_2(s_t) \end{bmatrix} \frac{(+/-)1}{e^{-\theta^T x(s)} + 1} G_t \tag{5}$$

The final result of the computation is $\theta_{t=2} = [0.7670, 0]^T$

2. **CarRacing with World Model**
project struct:

   (a) **data-preprocessing**: file and class for generate datasets
   (b) **model-weight**: it contains 4 torch model, (model.torch is the one that will be load into Policy)
   (c) **notebook**: series of notebook used for training the model
   (d) **rollouts-dataset**: image data (sub-folder) and dataframe
   (e) **memory˙module.py**: architecture of memory module
   (f) **vision˙module.py**: architecture of vision module
   (g) **student.py, main.py**

The WorldModel algorithm wants to create a representation of the environment, and use this representation to train a relatively lean controller (with few internal parameters).
The algorithm falls within the model-base classification, for which we the gained experience is used to train the environment model, and than on this model carry out the planning.

The reference paper use 3 main component: V, M and C (vision component, memory component and controller component). The 3 components are trained separately, that's a blessing for my resource but, as highlighted in several post on github, this could spoil the model performance.

The vision module it's a Variational Autoencoder, the memory module is made on a LSTM and a MDN (mixture density network).
By choice of the authors, the controller is the lightest module, made of a simple linear layer. The controller is also the only module that has access to the reward of the agent.
In my project, all the module has been trained on Colab, so for training details (loss funct, hyper-parameters etc) check the file inside the notebook directory.

Dataset generation
For start the project, i generate several random image of the environment, using the script `data_rollout.py`. The image are saved in jpg format in the folder `rollouts/CarRacing_random`, and during the simulation the action taken by the agent was totaly random. In a first attempt, the action was chosen from a list of discrete action (go right, go left, accelerate). Later on, i add a noise variable (`eps` variables) for having less bias. The range of this noise reflect the range of the parameters. This action will be concatenate to a vector and give in input to the memory module, so i prefer noisy action respect a small set. The action are saved in a csv file in the form of `img_name, action`

I would like to emphasize how I consider the method used to generate the dataset to be heavily biased, as it is difficult through random actions to get past the first curve. Better implementations could use datasets containing a greater variety of images or datasets generated by expert model.

Vision module

The task of this module was to construct a small-scale representation of the environment (z, 32-digit). For this we use a Variational Autoencoder, trained to reconstruct the input image. Later, from this network will be used only the encoder part. This is build on several conv2d layer plus 3 fully connected layer (for implement the so call "sampling trick"). The main advantage of the VAE respect other kind of encoder is the ability of organize the internal representation as a Normal distribution. The loss should be modify carfully for admit this representation, perhaps the loss used by the module is the sum of the crossEntropyLoss and the KL-loss, that measure the distance from a normal distribution.

The training was done on Colab, see VAE.ipynb for details.
The file vision-module.py contains only the reference architecture.
All the parameters for the model comes by the appendix of the original paper.
For the implementation i was inspired by: `https://github.com/sksq96/pytorch-vae`

Memory module and Memory Dataset
The second component of the project is the memoryModule, a model with a RNN (LSTM) and a MDN (Mixture density network). Using the output of the LSTM as input of the MDN, it's possible generate a Gaussian Mixture distribution that takes into account the history of the process. Than, it's possible sample from the distribution a possible hidden state z_{t+1}, the hidden rapresentation of the next frame. As input of this module (LSTM + MDN), we use the hidden state of the Vision module, concatenate with the action taken at time t.

For the training procedure, i thought it would be useful having a Dataset class that encapsulate the data, and return at each call the input state z_t and the future one z_{t+1}. Such class can be find inside the data preprocessind dir

A particular problem that i encounter in the training of the module was a numerical underflow of the weights of the net, so the loss function has been modified for numerical stability

Controller, evolution strategy and CMA-ES

The last module is the controller, that acting between agent and the environment, it's capable of get information about the rewards. Takes as input the concatenation of z (hidden state) and h(memory from lstm). Still using the paper as reference, i used a linear layer train by the CovarianceMatrixAdaptation algorithm, a evolution strategy that work particularly well for non-convex problem. This was also trained on colab, using a really low number of generation and population, because after some run colab went out of RAM. The reference for the script was the video of the next link, where it's show how a cma can be used for minimize a function. Considering that my problem was the opposite (maximize the reward) and that looking inside the library i cannot find the solution of this inverted problem, and on the documentation i found that the CMA don't work well with negative value, i define a evaluation function that calc the distance for a fixed score. The fixed value was chosen using the minimum score for consider the task solved (900)

Since the implementation of the CMA-ES work on a 1D vector, i extracted manually the weights from the layer, concatenate it and use it as population value
The rollouts procedure (for calc the cumulative reward) follows the original paper, here the pseudo-code:

```
obs = env.reset()
cum_rew = 0
init_h (memory)
while not done:
  obs = transform(obs) #to tensor 64x64, 1 channel
  z = vae.encode(obs)
  act = controller([z|h])
  env.step(act)
  update_reward
  update_h([z|a])
```