University of Alberta
Computing Science Department
Neuro-Symbolic Programming - W23

Assignment 3
20 Marks
Due date: 10/04 at 23:59

Overview

In this assignment you will implement two local search algorithms for finding programmatic policies in the Karel domain. We will search in the latent space learned with the LEAPS system. Each vector in the latent space will be decoded into a program through LEAPS's decoder. Once the program is obtained, we can compute its reward in the StairClimber environment used in the LEAPS paper. The goal of this assignment is to give you some experience with the synthesis of programmatic policies by learning a latent space.

You should submit two files: a zip file with your implementation and a pdf with the answers to the questions of this assignment.

Dependencies

You will need the following installed in your machine to run the starter code: torch (1.13.1), Pillow, matplotlib, h5py, numpy, pandas, tqdm. All these dependencies can be installed by running

`pip install -r requirements.txt`.

Implement the Cross Entropy Method (10 Marks)

The starter code provides most of the implementation of the Cross Entropy Method (CEM). Your task is to implement the function `search` of the class `LatentSearch` (see `latent_search.py`).

The signature of the `search` method is as follows.

```
def search(self) -> tuple[str, float]:
```

All the parameters you need are given through the command line and are accessible through class `Config` (see method `__init__` if `LatentSearch` for details). The search will return a program in string format that maximizes the reward in the StairClimber environment and the reward value the program is able to achieve.

The `search` function should implement the CEM algorithm used in the LEAPS paper, which is shown in the pseudocode below. CEM initializes a population of individuals (line 2). In our case, each individual is a vector in the latent space; the starter code provides this function for you (see `init_population`). For each iteration of the algorithm, CEM evaluates the current population by decoding each vector into a program, which is evaluated in the environment. The starter code also provides the implementation of a function to evaluate the population (see `execute_population`). This function receives a set of vectors and returns the programs the vectors represent and their reward values. During evaluation, each program is tested on 16 Markov decision processes (MDPs). The reward returned is the average reward obtained across all MDPs.

After evaluating the population, CEM selects the top k programs according to their reward values (line 5). In the starter code, the variable `self.n_elite` determines the value of k. The top k programs are referred to as the "elite" of the population. CEM then computes the average vector of the elite, which is used to

generate the next population (lines 8–11). Each individual is generated by adding a random vector to the `mean_elite` vector. The noise is given by a parameter `sigma` (`self.sigma` in the starter code) multiplied by a vector of values sampled from a normal distribution with 0 mean and variance 1.

The algorithm returns the best program (in terms of reward) encountered throughout the search.

```
1 def CEM(population_size, number_iterations, k, sigma):
2   p = initialize_population(population_size)
3   for _ in range(number_iterations):
4     evaluate_population(p)
5     elite = select the best k individuals in p
6     mean_elite = compute-mean-vector(elite)
7     p = []
8     for _ in range(population_size):
9       # N(0, 1) samples from a normal distribution with mean 0 and variance 1
10      individual = mean_elite + sigma * N(0, 1)
11      p.append(individual)
12  return best individual encountered in search
```

The starter code also includes a function, `save_gifs`, that receives a program in string format and saves one gif with the execution of the program for each MDP used in the experiments. This function allows you to better understand the programs the search is able to find.

Throughout development, you can use the following command to evaluate your implementation.

```
python main_latent_search.py --env_seed 1 --disable_gpu --search_number_iterations 20
--search_population_size 1028 --env_is_crashable
```

The option `env_is_crashable` is to use to original Karel configuration where the robot crashes as it hits the wall. The LEAPS paper uses a different configuration, where the robot "flips" as it hits the wall. The "crashable" configuration makes the problems harder to solve.

Implement Variant of CEM (3 Marks)

Next, you will implement a variant of CEM. Instead of reducing the elite to its mean value to create the next population, we will generate each individual of the next population by randomly sampling an individual from the elite and applying noise to it, exactly how we applied noise to the mean vector in the previous version of CEM. While implementing this version of CEM we will introduce the flag `--search_reduce_to_mean` that is set to False by default in the starter code. Whenever we pass this flag, the code should run the original version of CEM; if the flag is omitted, then it should run this new variant of the algorithm.

Experiments and Results (7 Marks)

You should test the two algorithms you have implemented with 5 different seeds (1, 2, 3, 4, and 5). Each run will perform 20 iterations with 1028 individuals in each population. Then answer the following questions.

1. (3 Marks) Discuss the results you obtained in terms of the reward value each search algorithm was able to obtain.

2. (3 Marks) If you were given the task of writing a Karel program so that the robot is able to climb the stair until the marker is reached, which program (out of all programs the two systems synthesized in your experiments) is closer to what you would have written? Visualizing the gifs could help to answer this question.

3. (4 Marks) What is the enhancement that you could be implemented to improve the results of both systems? Note that CEM is a stochastic search algorithm. Explain your answer.