Final assignment

by Tyfenn ELOY

Part 1

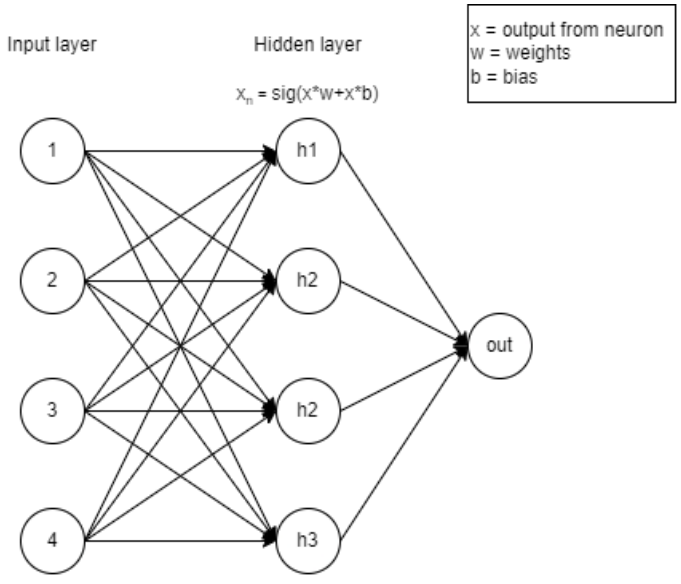
Introduction

"Implement a two-layer perceptron with the backpropagation algorithm to solve the parity problem. The desired output for the parity problem is 1 if an input pattern contains an odd number of 1's; otherwise, the output is 0".

In order to do so we had to implement a 2 layer neural network (4 input, and a hidden layer of 4).

As a set of additional instruction we were asked:

- to initialize all the weights and biases at random between [-1; 1],
- Use a sigmoid with a = 1 as activation for all units
- Run some tests by tweaking the learning rate from 0.05 to 0.5 with a step of 0.05
- Stop the algorithm if the error (difference) is below 0.05 on every pattern (a pattern is a set of 4 0 or 1)



Tests on the learning rate:

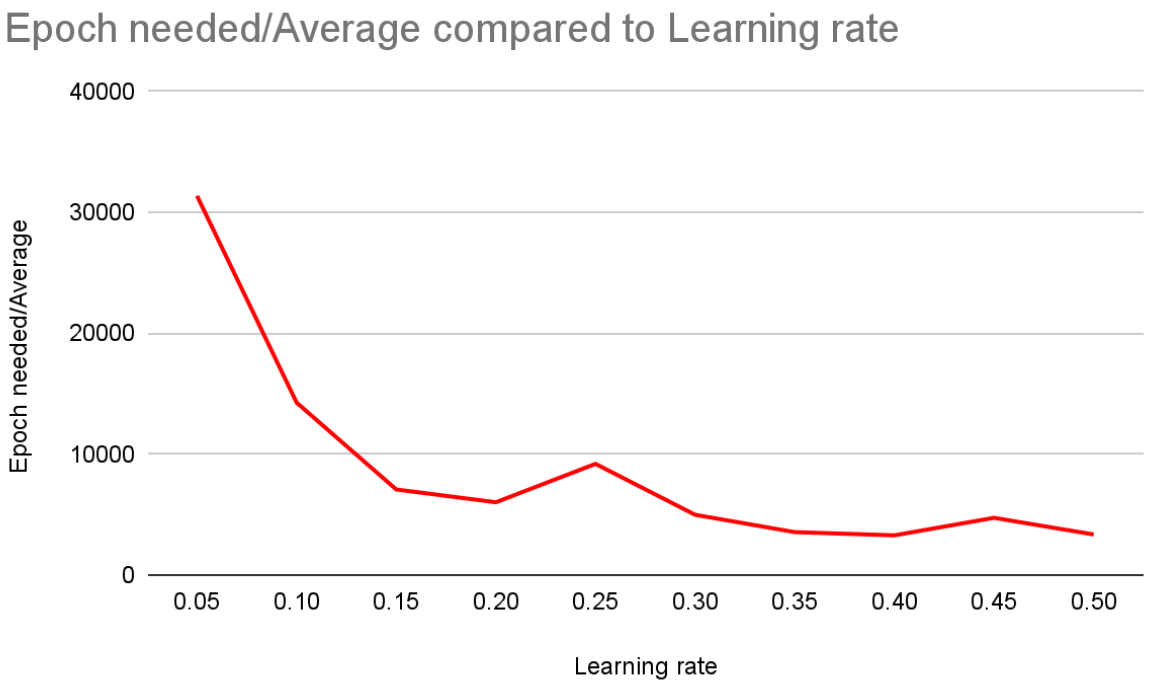
On each possible value I ran the algorithm 3 times, with a maximum number of epochs (iterations) = 50.000 for the training.

It is to note that there were on average 0.7 errors by value tested, but with a learning rate above 0.45 the errors were almost non-existent, as test demonstrated later.

On the graph below we can see the evolution of the number of epochs needed to reach an error rate of 0.05 which stops the algorithm.

We can clearly see that a learning rate below 0.15-0.20 takes way more epoch to reach the error (difference) of 0.05 on each pattern. Going toward values above 0.20. We can see a small decrease of the number of epoch needed over the increase of the learning rate.(even if we can see spikes, those can be explained by bad-luck more tests should be run to increase precision of the graph).

As I quickly mentioned a earlier learning rate seems to highly decreases the amount of error per tests (tested in range [0.45, 0.5])



Part 2

Introduction

We were given the task to implement and solve the MDVRP (multi-depot Vehicle routing problem) problem, using a NP-Hard Genetic algorithm. The goal was to reduce as much as possible the travel distance of vehicles that had to deliver things to customers. With added constraints of vehicle capacity being enough for all customers on the route.

We were asked to implement code to read data from a given format (cf. cell number 4 of the notebook).

What I did

As a main source of inspiration I looked up to a github project coded in Java.

The first attempt.

My first implementation of the solution (scrapped the 17/12/2021) managed to produce results far below the expected result.

I chose to mix OOP to create class representing elements, to simple functions outside classes.

In this solution, the individuals were a set of routes (as a list) each route contained a set of customers (randomly assigned, as a Customer object).

The chromosomes were the routes themselves, and this implementation is the error that made me refactor my project entirely. As I managed to code more functions needed to run the GA, I realized that my sets of data were way too messy to develop.

The fitness of a solution was defined by the sum of distance to be traveled by a vehicle (we consider that a route only has one vehicle) the bigger, the less fit it is.

The crossover function chose randomly two individuals (the fittest have better chances to be picked), and created two children with the DNA (set of routes) of the parent split in two (randomly between the 1st and 3rd quantile) and sicked together after in the 2 children. I also added a way to ensure that parent with capacity over the limit (too much demand on a route) were 2 times more unlikely to be picked by doubling their fitness

The mutate function simply swapped a customer to another depot.

This is where I noticed how bad this first solution was as, when run the algorithm would just remove customers from routes to gain fitness, so in the end I had a fitness of ~150 but almost no customers left, so I decided to refactor all my code and way of thinking.

The second attempt.

Still using OOP I created the following classes:

Class	Role	Alternative name
Individual	Contains a list of depots and it’s own fitness	Solution
Depot	Contains a list of routes, its coordinates	**Genome**
Route	Contains a list of customers, it’s capacity, coordinates	**Chromosome**/vehicle
Customers	Contains its own coordinates and demand	**Genes**

When I restarted my project I decided to keep some code as I thought not all the ideas were to be thrown away. For example the fitness, and individual is still considered the **fittest** if it’s **score is low**, said score being defined by the total **sum of distances** traveled by the vehicles on said sets of routes.

While it’s still possible to have **unvalid routes** because of the customer demand exceeding the vehicle capacity. Individuals with such chromosomes (**overloaded roads**) will disappear over the generations as we choose on each of those to multiply their cost by a certain factor (Goal is to diminish the individual’s fitness).

When it comes to the **crossover** function I changed it. Because as you probably noticed, instead of having a set of routes as an individual I chose to create a specific class that would contain each depot and each depot containing it’s routes. So now the crossover function works by **selecting a random route** from each of the two given individuals henceforth designated as parents (said parents are still selected depending on their fitness influenced by the capacity of each of it’s routes and a random factor). The child is created by taking the second given parent, **removing each customer it has in common** with the first given parent and attempting to **place it at the best spot it possibly can** (depending on the cost of the route after insertion).

The **mutation** function is fairly simple (a few were tried and fewer were the good results), **We simply take a customer and assign it to another depot**. The function is called randomly depending on a given parameter.

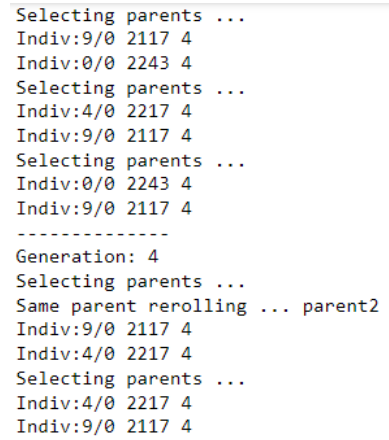
It is to be noted that the routes created on the first generation are not random, each customer is assigned to the closest depot from them.

The final results

While I could not make an algorithm able to fully predict the given solution, we can clearly see an evolution throughout the generations. Sadly however after a few generations we can notice that the algorithm remains stuck and keeps trying to reproduce the same individual over and over thus evolution is stopped fairly early.

We can still notice that modifying certain parameter changes the output, testing on the file p01.txt:

- Having more generations simply means looping over the crossover function of the same individuals over and over after the 6th-8th generation
- Having less individuals augment the chances to have the same individuals in the next generation, thus stopping evolution faster as we can see on the figure on the right side (**fitness is the number before 4**)



My idea to improve the code would be to **improve the mutation function**, as increasing the amount of mutation (while it should) does not resolve our problem. Perhaps **refactoring** the code of the **crossover** function to select the best place to insert the customer would be a good idea too, but I am fairly sure the problem does not come from here.

Had I had the chance to I would have implemented a **crossover rate** to make some parent return themselves instead of offsprings to the next generation, but I don't think the problem would be solved by doing so.

I still think that the main idea behind the algorithm would work, even if our crossover and mutation function would **create infeasible solutions** (because of the weight), they would tend to **disappear** after a few generations, as it **doubles their fitness** thus making them less likely to be selected for reproduction.

As a final comment I would like to highlight the fact that I have found a git repository answering the exact same problem but in java. And while I drew some inspiration, all the code I present to you was made by myself, and not a direct translation from java to python

Sources:

Part 1:

- How to Code a Neural Network with Backpropagation In Python (from scratch)
- Backpropagation from scratch with Python - PyImageSearch

Part 2:

- GitHub - markusmkim/GA-MDVRP: Genetic algorithm for Multi-Depot Vehicle Routing Problem optimization
- Picking a random item based on probabilities - Stack Overflow

Github repository:

https://github.com/ItsaiKara/machineLearning