Evolutionary Robotics
Prof. Dario Floreano

Evolutionary Robotics Laboratory

Exercise Sheet 0: Evolutionary Algorithms

Euan Judd (euan.judd@epfl.ch)
Krishna Manaswi Digumarti (krishna.digumarti@epfl.ch)

Goal. The goal of this lab is to understand some advantages and disadvantages of evolutionary algorithms (EAs). You will observe the effects of the mutation rate, crossover rate, selection pressure, population size and number of generations in artificial evolution, and reflect on how to change these parameters when performing an evolution according to different types of fitness landscapes**.**

Rules:

- **You are expected to form a group of two people and work with one computer.**
- **You are expected to complete all exercises during these two lab hours.**
- **You should provide written answers for all exercises. There is no official submission of these written answers but it is crucial for your understanding for the following labs, the Robogen grand challenge and the final written exam.**
- **A PDF with answers will be uploaded next week, before your next class, so that you can compare them with your answers. If you have any question, you can discuss them with the teaching assistants in class or by sending us an email.**

Some information: The labs (TPs) for the Evolutionary robotics course are held in **BS 160**. You will be using your own computers. We will use the Python programming language in the first two lab sessions (this exercise sheet). We will use the browser version of the Robogen software in all subsequent labs.

You will not have to do much programming in this exercise sheet, but a basic understanding of Python is required. **If you are not familiar with Python, you can find documentation and numerous tutorials on the official python website.**

Getting Started. You will have to download the file EA exercises.zip and unzip it. This lab employs the *inspyred*[1] framework for the Python programming language. Python is not installed by default.

Navigate to the directory where you unzipped the EA_exercises.zip file and run the *setup.bat* file by double-clicking on it. This should install Python 3.6.4 on your user's account and setup

[1] http://pythonhosted.org/inspyred/

your environment with all the required dependencies for this lab. The script will ask you if you wish to install Visual Studio Code (VSCode) which is an editor that has been configured with all the required extensions. We therefore recommend that you use VSCode now, as it could increase the productivity, but you can setup VSCode later by running the *setup-vscode.bat* file.

To verify that everything is working properly, run the *ea.bat* file to open the command prompt in the right environment and run the following command[2] :

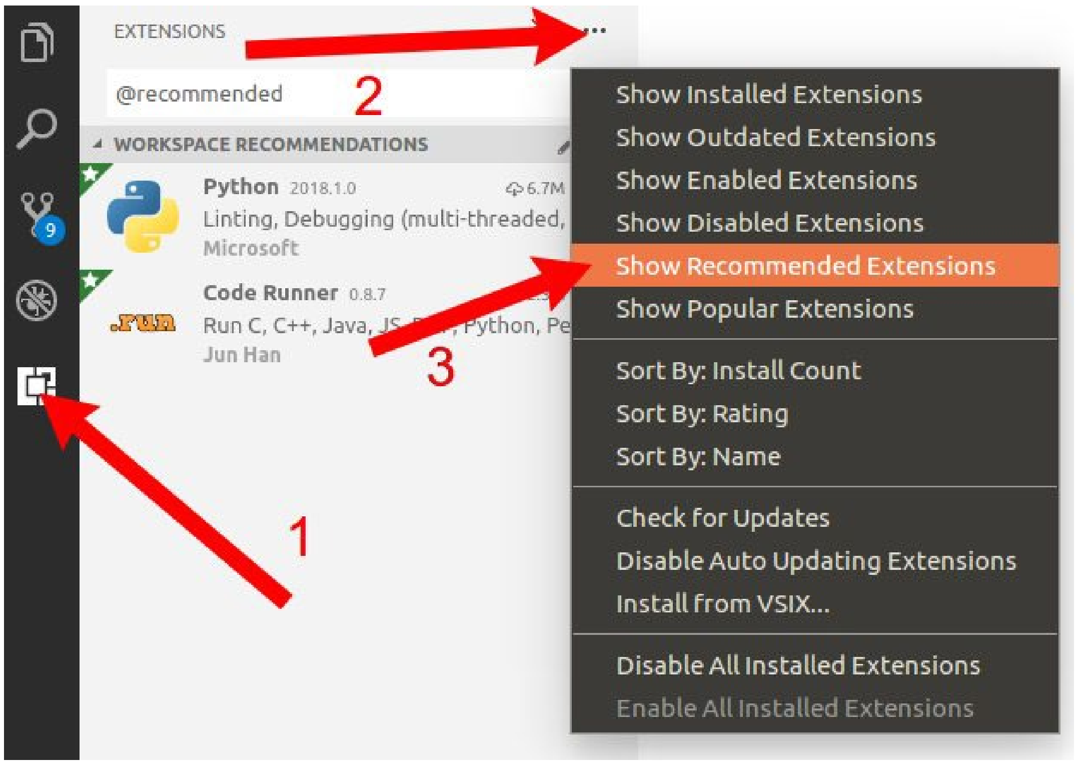
```
$>python exercises\test.py
```

If you see "All Good." Then everything is working properly. However, if you see an error, it means something went wrong and you should ask one of TAs for assistance.

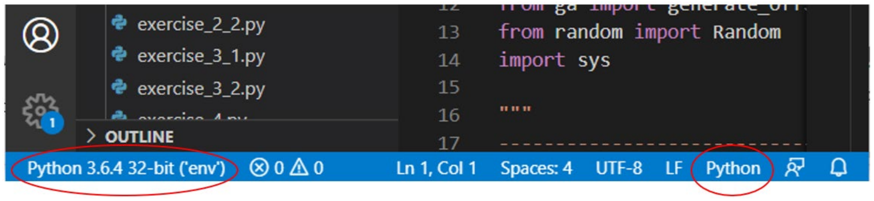
VSCode Setup

If you are using VSCode, open the File menu and select Open Folder then navigate to the exercise directory.

Install the recommended extensions when prompted. If the prompt does not appear, or if you pressed close by mistake, open the Extension view on the left side of the screen (3), open the menu (2) and select "Show Recommended Extensions" (3). Next, press the green Install button below Python and Code Runner. Press the blue Reload button or restart VSCode when the installation is finished.



If your environment in VSCode is correctly set to 3.6.4 with all the required libraries, then you should see this:



```
from random import Random
import sys

"""
```

Python 3.6.4 32-bit ('env') ⊗ 0 ⚠ 0 Ln 1, Col 1 Spaces: 4 UTF-8 LF Python

Each exercise has a corresponding .py file (named after the exercise number: exercise_1_1.py, exercise_1_2.py, etc.). To solve the exercises, you will have to open, edit, and run these .py files.

Tip. If you do not know what a Python module/class/function does, open a Python command prompt by double-clicking *ea.bat* and typing:

$>python

And then run[2]

>>>help(my_function)

from the Python command prompt.

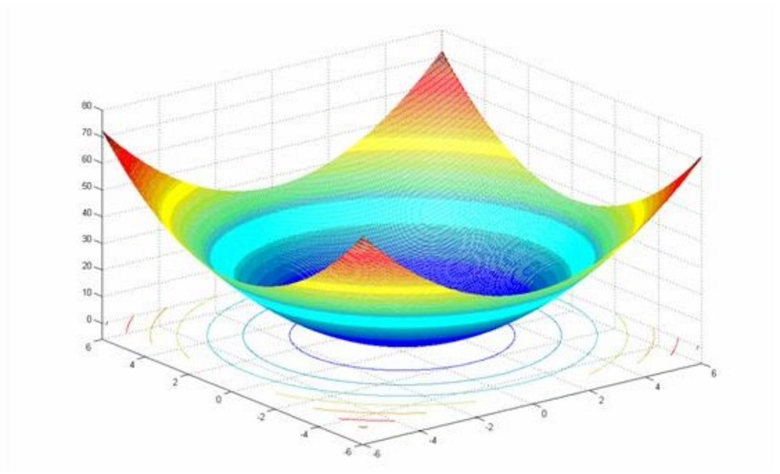
Note. In this lab, the genome of an individual is always a vector of real-valued parameters $\mathbf{x} = [x_1\ x_2\ \ldots\ x_N]$. Keep in mind that other types can be used to encode genomes but these will not be explored in this lab. The fitness of an individual is given by the fitness function $f(\mathbf{x})$. Here, the aim of the EA is to ***minimize*** the function $f(\mathbf{x})$, i.e. to find the vector $\mathbf{x}_{min}$ that has the lowest value $f(\mathbf{x})$. In other words, ***lower* values of *f*(x) correspond to a *better* fitness**.

[2] Just like we use $> to represent the DOS command prompt, we use >>> to represent the Python command prompt. **Do not retype these symbols!**

Exercise 1

In this first exercise, we will not yet run a complete EA. Instead, we consider a single parent individual $\mathbf{x}_0$, from which a number of offspring individuals are created using a Gaussian mutation operator (which adds a random number from a Gaussian distribution with mean zero and a standard deviation σ that you will set to each parameter x_i of the parent). The fitness function is defined as:

$$f(x) = \sum_{i=1}^{n} x_i^2$$



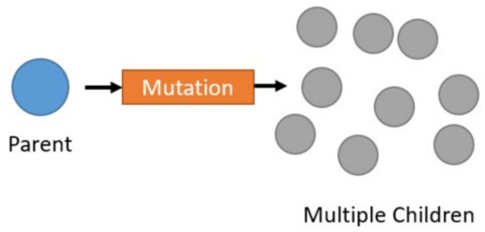
This fitness function is unimodal, it has a single global minimum at the origin. We will analyse the effects of mutations on the fitness depending on the value of the parent $\mathbf{x}_0$, the mutation magnitude (the standard deviation σ), and the number of dimensions N of the search space.

Exercise 1.1

In VSCode, open *exercise_1_1.py* and press the Run button in the upper left corner. If you are not using VSCode, double click *ea.bat* to open a terminal and type[3]:

```
$>python exercises\exercise_1_1.py
```

You can also type the above in the VSCode terminal to run the file.



In this exercise, a single parent $\mathbf{x}_0$ is mutated to generate multiple offspring using a Gaussian mutation operator. You should first consider a one-dimensional case, then two dimensions, and finally many dimensions (e.g. N

[3] **Note.** For all exercises in this lab, you may follow the name of the .py file with an integer value which will serve as the seed for the pseudo-random number generator. This will allow you to reproduce your results.

=100). For the one-dimensional case, you should try different parents, e.g. $\mathbf{x}_0$=0.1, $\mathbf{x}_0$=1 and $\mathbf{x}_0$=10, and use different mutation magnitudes (standard deviations σ). Run the algorithm and wait for it to finish. A boxplot[4] will appear which shows the fitness of the offspring. The fitness of the parent will be shown as a dashed green line on the same plot. A second figure will also appear when there are one or two dimensions which shows the parent and the offspring on the fitness landscape. As our objective is to find the global minimum, try tuning the parameters such that at least one of the children has near zero fitness. Provide written answers for the following questions:

- Do the mutations tend to improve or worsen the fitness of the parent?
- Are low or high mutation magnitudes best for improving the fitness? How does this depend on the initial value of the parent and on the number of dimensions of the search space?

Exercise 1.2

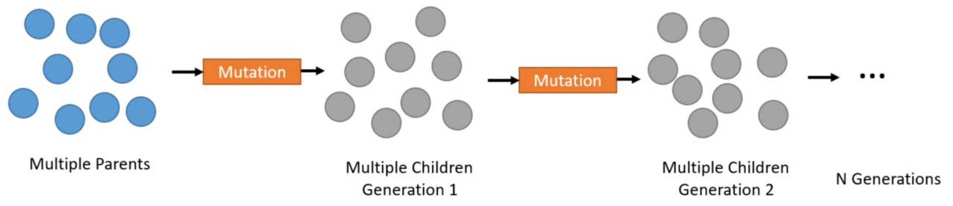
Run the exercise_1_2.py file to confirm the observations that you did qualitatively in the previous exercise. You will see boxplots plotted side-by-side to evaluate the statistical significance of observed differences. Compare different values for:

- The number of dimensions of the search space
- The value of the parent (how close it is to the optimum)
- The mutation magnitude

If you vary one of these three parameters, **make sure that you set the other two at a constant value** (otherwise it may be difficult to interpret your results). Try to confirm the answers that you gave in the previous exercise.

Exercise 2

We will now use an EA to find the minimum of the unimodal fitness function defined in the previous exercise. You should analyse how the mutation rate magnitude, dimensionality of the search space and the number of generations influence the results.


Mutation
Mutation
...
Multiple Parents
Multiple Children Generation 1
Multiple Children Generation 2
N Generations

[4] Visit https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.boxplot.html if you are unfamiliar with boxplots

Exercise 2.1

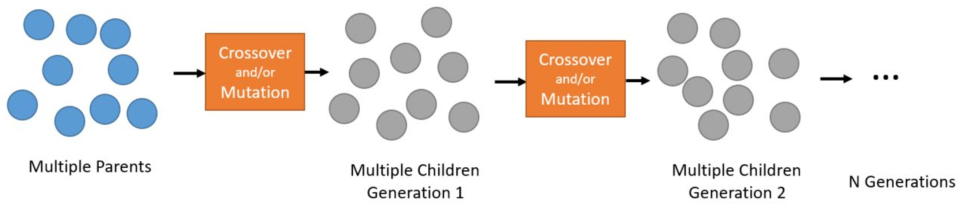
You will now use a basic, mutation-only EA. Run the exercise_2_1.py file which is initially set to optimize for a one-dimensional function. How close is the best individual from the global optimum? Increase the dimensionality of the search space to two-dimensions and higher. Now how close are the best individuals from the global optimum? Can you get as close as you did in the one-dimensional case by modifying the mutation magnitude and/or the number of generations?

Exercise 2.2

Run the exercise_2_2.py file to do three batches of 30 runs of the EA with different mutation magnitudes (it may take more time than previous runs). The boxplot that appears compares the best fitness values obtained in the three conditions. Try to explain the results.

Exercise 3

In this exercise we will analyse the effect of crossover in the EA. Using crossover, a new individual is formed by making a new vector where each entry x_i is randomly selected from one of the two parent individuals $\mathbf{x}_1$ and $\mathbf{x}_2$. The EA has a parameter defining the fraction of offspring that is created using crossover at each generation (the remaining individuals are created via asexual reproduction).


Multiple Parents
Crossover and/or Mutation
Multiple Children Generation 1
Crossover and/or Mutation
Multiple Children Generation 2
...
N Generations

Exercise 3.1

Using the same setup as in the first two exercises, the exercise_3_1.py file does 30 runs using mutation only (as in the previous exercises) and 30 runs using crossover only. The boxplots compare the best fitness values obtained in the two cases. See if you can explain the results.

Exercise 3.2

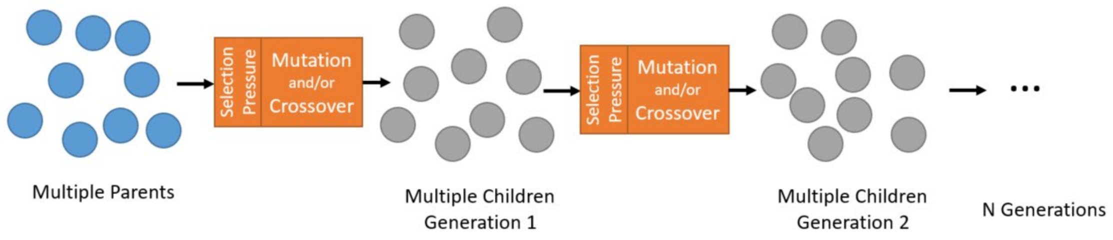
Run exercise_3_2.py to compare the best finesses obtained by varying the fraction of offspring created using crossover (while using a fixed mutation probability of 0.5, i.e. each loci x_i of each genome will have a 50% chance of being mutated).

Is there an optimal crossover fraction for this fitness function? Interpret the results.

Note: You can also set the mutation rate to zero in this exercise to realize that effect of crossover only.

Exercise 4

We will now investigate the effect of the selection pressure.


Selection Pressure
Mutation and/or Crossover
Selection Pressure
Mutation and/or Crossover
...
Multiple Parents
Multiple Children Generation 1
Multiple Children Generation 2
N Generations

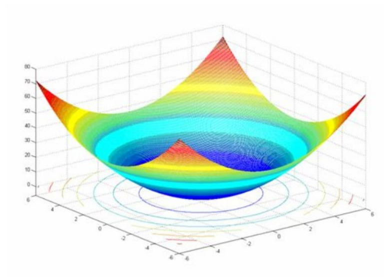
Run the exercise_4.py file to compare the best fitness values and the distances from the global optimum obtained using tournaments of size 2 and 10. Try each for both the sphere and Rastrigin[5] fitness function. **Note:** In the previous exercises, we were using tournament selection with a tournament size of 2.

- Which tournament size gives better results for the sphere fitness function and why?
- Which tournament size is better for the Rastrigin fitness function and why?

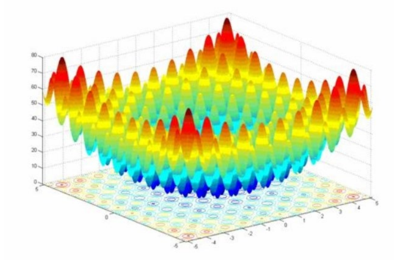
Exercise 5: Let's explore!

In this exercise you will investigate running the EA on many test functions commonly used to benchmark optimization algorithms. Run the EA on the benchmark functions shown below (especially the multimodal functions) and adapt the mutation magnitude, crossover rate, selection pressure, and population size to get the best results. If you run the code as provided, it will initialize and bound the values of your population vectors to suitable ranges. You may comment/uncomment certain lines to alter this behaviour. See the comments in exercise_5.py for further details.

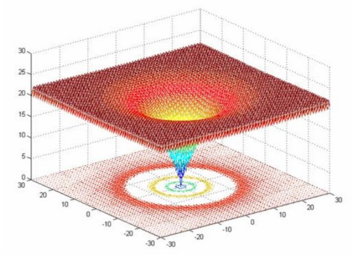
5 http://pythonhosted.org/inspyred/reference.html#inspyred.benchmarks.Rastrigin



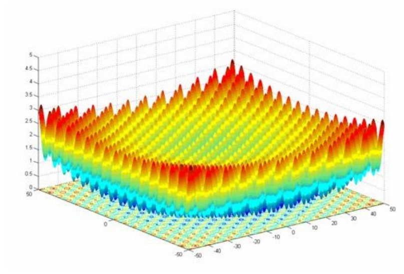
Sphere Function



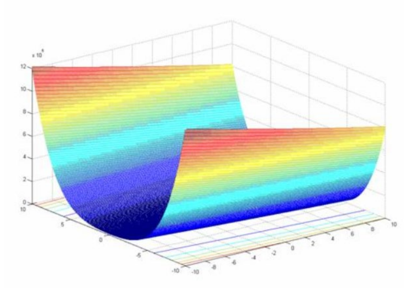
Rastrigin Function



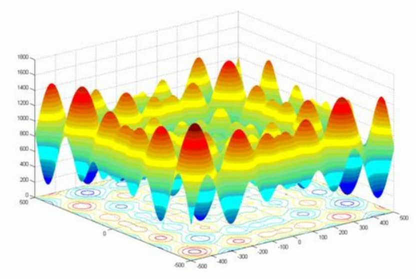
Ackley Function



Griewank function



Rosenbrock Function



Schwefel Function

You should first try the 1D or 2D cases so that the fitness landscape can be visualized. However, keep in mind that sometimes the resolution of the plot is not sufficient to accurately represent a function.

Exercise 6

In this exercise you will investigate running an EA to evolve the parameters of an artificial neural network (ANN). While there are other ways to learn the weights of an ANN, using evolution is an effective means in many circumstances. We will first evolve the weights of a simple feed forward neural network and then we will look at evolving the weights of more complex, recurrent neural nets.

N.B. All hidden and output neurons of these neural networks use the logistic activation function:

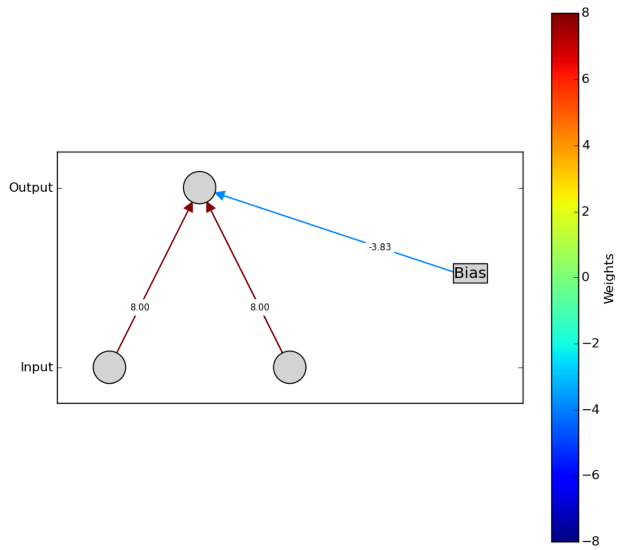
$$f(x) = \frac{1}{1 + e^{-x}}$$

Exercise 6.1

We begin by evolving the weights of a minimal neural network to solve the Or problem. That means we will use a neural network that has two inputs, and one output, which should produce the logical Or function of the two input values:

Input 1	Input 2	Target Output
1	1	1
1	0	1
0	1	1
0	0	0

Run exercises_6_1.py to evolve the weights for this Or network. The fitness here is the sum of squared errors between the network's output and the target output across each of the four input patterns. If you see the best fitness approach zero (e.g. a fitness of less than 0.1) then you have found a network able to solve this problem. This most likely looks similar to:



Here the neural network is depicted with its weights and biases shown by the corresponding color. If you were able to solve the Or problem, look at the weights of the neural network and think about/compute how it behaves when given different input patterns. It is important to think about this now, because it will be difficult to keep track of what our neural networks are doing once we start using more complex topologies.

If you were not able to solve the Or problem, try modifying some of the parameters of the EA until you are able to do so.

Once you are able to solve Or, try solving the AND problem instead (change *problem_class = Or* to be *problem_class = And*). Did the same parameters that you used for Or work for And as well? If not, modify them until you are able to solve And.

Now that we can solve Or & And, we will try something a little more challenging. Change the problem_class to be Xor, so that we are now trying to solve the Exclusive Or (Xor) function:

Input 1	Input 2	Target Output
1	1	0
1	0	1
0	1	1
0	0	0

This function has one small, but crucial difference from Or, as can be seen by comparing their truth tables.

Try running the code again after change the problem to Xor. Can you solve it? What if you change some parameters of the EA, such as the population size, crossover rate, max generations, etc.? If you are unable to solve it, why is that? Think back to what you saw in the lecture.

Here we offer you one new parameter to modify that you did not see in the previous exercises, and that is the number of hidden units of the neural network. Try changing this parameter from 0 to 1. Does this allow you to solve the problem? What if you change this value to 2 or some higher number? How many hidden neurons are required to solve this problem? Can you provide an explanation for why that is the case?

When you find a network that does compute Xor, once again see if you can understand how the network does so.

Exercise 6.2

So far we have looked at networks where you give them an input vector and they immediately produce an output. However, there are many tasks that we want to use neural networks for that should not base their output solely on one instance of time but should instead respond to changing inputs over time. In this exercise we will begin to investigate these types of problems and you will see many more of them later in the course when we begin to use neural networks for controlling robots.

First, we start with a modified version of the Or problem, that is called "Temporal Or". While the basic Or problem involved evolving a neural network that would give a single output when provided two simultaneous inputs, in Temporal Or, there is only a single input neuron and the input values are provided in sequence. Therefore, the network will have to remember the first input when seeing the second in order to output the correct value.

Try running exercise_6_2.py to solve Temporal Or. Is the EA able to solve this problem? If not, try repeating some of your strategies from the previous exercise such as modifying the EA parameters or adding more hidden neurons. Can you now evolve a successful network?

If you still cannot evolve a network able to compute Temporal Or, notice that there is one new parameter that you can modify: "recurrent". This parameter is a Boolean flag, that says whether the network is recurrent (in this case an Elman network) or not (in which case it is feed forward). If you set recurrent to be True, can you now evolve a successful network? Why might recurrence be important for being able to solve a temporal problem such as this?

Once you have been able to evolve a network capable of solving Temporal Or, you can change the *problem_class* to *TemporalAnd* and repeat. Did the same parameters that solved Temporal Or also work for Temporal And? Why or why not?

Finally, change the *problem_class* to *TemporalXor* to attempt solving a temporal version of Xor. Run the code again. Are you able to find a successful network? If not, think back to what you just saw as well as the previous exercise. What combination of recurrence and number of hidden nodes is needed to solve Temporal Xor and why is this the case?

Conclusions

- Are there optimal parameters for an EA?
- What are the advantages and disadvantages of
    - Low/high mutation rates?
    - Low/high selection pressure?
- Based on these observations, do you think there is an optimal mutation rate for a biological organism? Do mutations typically improve or worsen the fitness of a biological organism? In which situations do you think low/high mutation rates are advantageous for a population of bacteria?
- What is the genotype and what is the phenotype in the problems considered in this lab?
- Why are hidden units sometimes needed for a neural network to solve a given task? What is the defining characteristic of problems that networks without hidden units are unable to solve?
- Why are recurrent connections needed to solve certain problems? What is the defining characteristic of problems that networks without recurrent connections are unable to solve? Are there problems that require recurrent connections and multiple hidden units?