CSCI 4831/5722 Computer Vision
Instructor: Fleming, Spring 2019
Homework 1
Due: 6:00 pm, Saturday January 26th

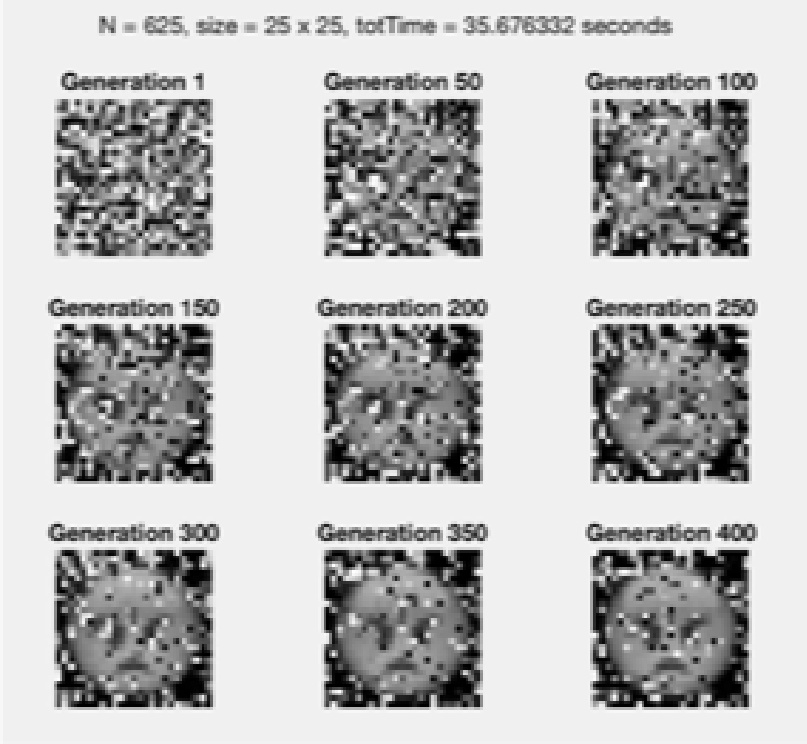
Genetic Algorithms - an Introduction to Artificial Intelligence

Objectives: this homework is meant to give you an introduction to Matlab and working with images

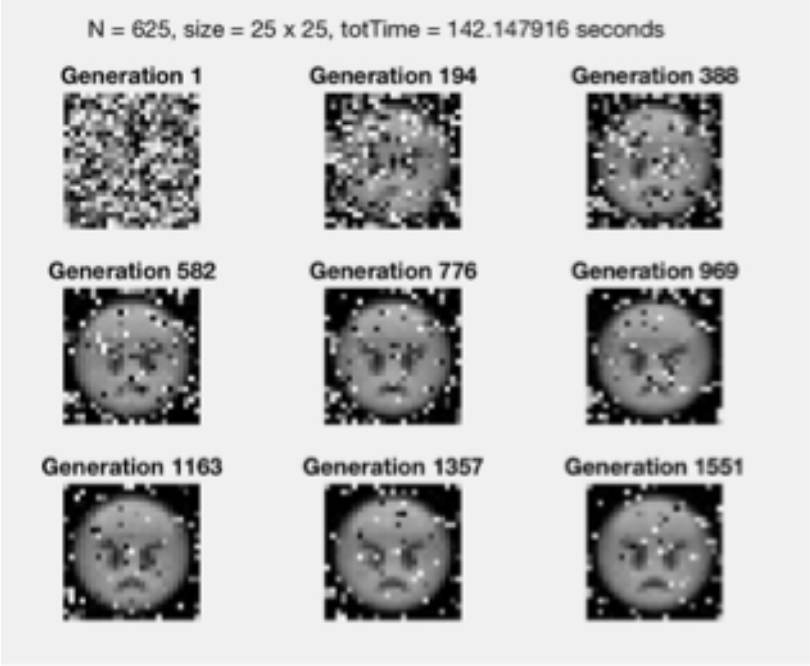
- Create scripts and functions
- Understand how images are represented in the matlab environment: matrix size, data type, supported image formats, grayscale vs RGB
- Upload, modify, display and save images
- Understand and practice vectorization: avoid loops, nested loops, recursion and replace them by vectorized matrix operations

The concept of Artificial Intelligence (AI) has been intensely studied by many computer scientists since the early 1950s. However, due to the enormous computing power required to build and run AI algorithms, we have only starting seeing serious applications of these technologies in the last decade or so.

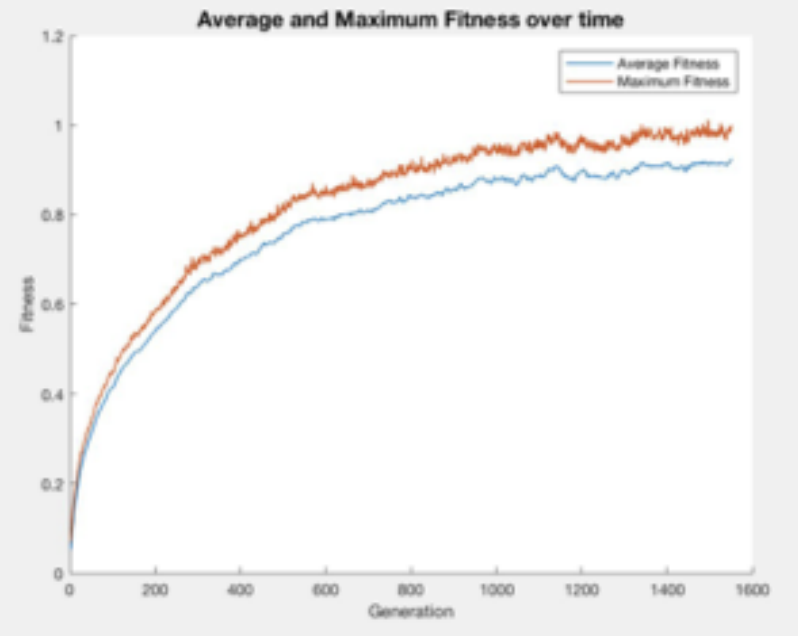
Many of our designs for AI systems are inspired directly from nature, such as how neural networks, the widely publicized method used to create Google's *Alpha Go* , roughly model neurons in human brains. In this project we will be considering another model of machine learning called ' *generative algorithms* '. The concept is derived directly from Charles Darwin's 'survival of the fittest' theory of evolution: we are going to 'breed' a picture from completely random data over many generations by assessing the 'fitness' of each member of the population. An example of breeding the angry emoji over 400 generations (in black and white) can be seen below:



While the generated image may not be *perfect* , it is clear to see that after roughly 300 generations we can begin to discern the basic features of the angry emoji. Here is the same image bred over more generations:



After a certain point there are clearly diminishing returns in running the algorithm any further. The maximum and average fitness can be seen to stagnate in this graph:



Monkeys on a typewriter

Before we tackle images we will first build the fundamental components of the genetic algorithm to breed a text phrase.

There is an age-old thought experiment pertaining to the nature of randomness and time called the 'Infinite Monkey Theorem'. It states that if you put a monkey on a typewriter and let it start hitting random keys, it will start typing random characters, and maybe occasionally a complete word or two... but you'd certainly have to wait a very long time to get anything *meaningful* typed from this process. However, if we gave the monkey an *infinite* amount of time, it could type out any given text we're looking for, such as a Shakespeare play.

In the real world we do not have infinite amounts of time, so we'd instead have to think of creative ways to speed things up. We could start by adding more monkeys on more typewriters, giving rewards to the monkeys when they type complete words, or even limiting the amount of keys certain monkeys had access to.

Because computers essentially have the 'intelligence' of a million monkeys banging on typewriters, we need to consider these sorts of optimization strategies to get the computer to 'think' for us. This is an **open-ended project** , and while you will get many suggestions of how to do things along the way, you may discover your own tweaks to make your program even *more* efficient. You are encouraged to experiment at every step, but still stick

to the core ideas presented here. More will be discussed on how to make modifications or implement your own ideas in Tasks 1.7 and 2.5.

The basic outline of the generative algorithm we are going to use is as follows:

1) Initialize population members with random DNA
2) Calculate fitness of each member in population
3) Build mating pool based on fitness of each member
4) Select two parents from mating pool and breed child
5) Apply some random mutation to DNA of child
6) Repeat steps 4 and 5 until new population (new generation) has been bred
7) Repeat steps 2 through 6 for each new population until goal is reached

First, we are going to try and get the computer to generate the phrase 'To be or not to be'.

Task 1.1: Randomly generate initial population of strings

We are going to write a function *buildPopulation* that will initialize 200 random strings (all the same length as the target phrase) containing upper/lowercase letters and spaces. You should store these population members in a single cell, which your function should output. In this function you should also define the population size as a variable, an important quantity that we will come back to often in this project.

To generate random characters, we simply have to generate random integers which correspond to text characters in the ASCII table (http://www.asciitable.com/), and use the *char( )* typecast to turn them into characters.

Following the biology analogy, our population (cell) now contains organisms (strings) with DNA bits (individual characters) that were randomly generated. We are going to try and match that DNA to the target phrase's DNA.

Help links:

- Functions - https://www.mathworks.com/help/matlab/function-basics.html
- Cell arrays - https://www.mathworks.com/help/matlab/cell-arrays.html
- char() - https://www.mathworks.com/help/matlab/ref/char.html
- Randsample - https://www.mathworks.com/help/stats/randsample.html

Task 1.2: Calculate fitness of each member in the population

In this example, we will simply define fitness as how many characters the current member of the population has correct when compared to our target phrase. For example, if our target phrase is '`Hello World`', and we are assessing the fitness of the string '`qK lfdRoPLd`', we would compare it as follows:

Target:	H e l **l** o W **o** r l **d**
Test string:	q K **l** f d R **o** P L **d**
DNA match:	0 0 0 1 0 0 0 1 0 0 1

The test string would have a fitness score of 3, due to having 3 DNA matches to the target phrase. A phrase with 100% fitness (ie. the phrase '`Hello World`') would have a fitness score of 11 in this example.

You can see that with a longer target string, the maximum possible fitness goes up. Furthermore, having a fitness of 3 in a phrase like 'Hello World' is much less significant to having a fitness of 3 in a shorter phrase like 'Hello'. Therefore, it is going to be more useful for us to look at the string's fitness as a **percentage** , where 0 has no matches, and 1 is completely matching (also known as **normalizing** the fitness score). In this case the test string has a fitness of approximately 0.273, or 27.3%.

We will now write a function *calculateFitness* which calculates the fitness of the entire population in the manner discussed above.

Task 1.3: Build a mating pool

Now that we know the fitness of each member in the population, we are going to build a 'mating pool'. This is essentially a crude way of selecting two parents from the population based on fitness, where fitter members are more likely to be picked.

You are going to write a function called *buildMatingPool* . The basic idea is that the mating pool will be like a raffle that we can pick parents from. When setting up the mating pool raffle, we will simply give the fitter phrases more tickets to be chosen by.

We could achieve this by multiplying the fitness (which is a value from 0 to 1) by some sort of ' **mating factor** ' (say 10), and add that many tickets to the raffle. For example, if the population member has a fitness of 0.2, it will get 0.2*10 = 2 raffle tickets added. If the next member has more matches and has a fitness of 0.8, it will get 8 tickets, and thus will more likely be chosen when parents are selected.

One drawback to this method of building the 'raffle' is that it doesn't enter many tickets at all if the entire population's fitness is low (such as when you first start the algorithm with random text). To account for this, we will again normalize our population's fitness values so they go from 0 to 1, where 1 corresponds to the maximum fitness of the current generation. Now the fittest member will always be given 10 tickets into the raffle, even if it only has a fitness of <0.001.

Your completed mating pool outputted by your function should only contain indexes of the population members added to it, not entire copies of the string (this will take up a lot of unnecessary memory). It is also important to have your mating factor as a variable that can be easily adjusted later.

Task 1.4: Breed a child from two parents
Note: at this stage you may wish to begin on your script outlined in Task 1.6 to begin testing some of these components.

We are going to write a function which will breed two of our organisms (strings) by combining their DNA (characters). To do this we try two methods. In the first method we will choose a random midpoint in the two phrases, and split the two parents' DNA up accordingly. For example, if we were breeding the phrases '`Heqqq`' and '`R llo`', and were lucky enough to randomly choose the midpoint 2, the child would look like:

Parent 1 DNA:	**H e** q q q
Parent 2 DNA:	R **l l o**
DNA from 1 (midpoint = 2):	1 1 0 0 0
DNA from 2 (opposite of 1):	0 0 1 1 1
Child:	H e l l o

Our second method will be to take a completely random selection of one parent's DNA, followed by taking the remaining DNA from the other. For example, if we were breeding the phrases '`HrllL`' and '`pe qo`', the result may look like this:

Parent 1 DNA:	**H** r **l l** L
Parent 2 DNA:	p **e** q **o**
DNA from 1 (randomly selected):	1 0 1 1 0
DNA from 2 (opposite of 1):	0 1 0 0 1
Child:	H e l l o

Again, this would be a very lucky case where the randomly selected DNA for each parents spelled out the word '`Hello`'.

Write a function called *breed* which takes two parents as an input, and an integer (0 or 1) and returns a child as an output. The function should contain both methods of breeding a child, one to be used when the third input value is 0, and the other when the thiurd input value is 1.

Task 1.5: Cause DNA mutation

The key to any evolutionary process is the fact that DNA can randomly mutate, and sometimes those mutations lead to improvements in an organism that two parents' DNA alone could have never provided.

We will write a function called *causeMutation* which will randomly mutate the DNA of the child outputted by *breed* by randomly changing some characters. Start by defining a mutation rate which determines how often mutation should occur. If a character is selected to be mutated, simply change it to a new random value (A-z, or space). A typical mutation rate is around 1%.

Task 1.6 : Bring it all together in a script!
We now have all of the components of a genetic algorithm. In a script, we can now have them work together to produce our target phrase, '`To be or not to be`'.

1) Define target
2) Generate initial population
3) While the target is not found OR maximum generations has not been reached, do the following:
    a) Calculate fitness
        Every 10 generations, print to the Command Window the following
        - The generation number
        - The maximum fitness member from the current generation
        - The string with the maximum fitness
    b) Build mating pool
    c) Breed a new population
        - Randomly select two unique parents from mating pool
        - Breed child from parents
        - Mutate child
        - Replace old population members with children until entirely new generation has been bred
4) Print total generations and total time elapsed to Command Window (*hint: use the keywords ' tic ' and ' toc ' to time a segment of code*)
5) Plot maximum and average fitness over generations
6) Plot 'genetic diversity' (maximum fitness - average fitness) over generations
7) Save best phrase, maximum fitness, average fitness, and genetic diversity for each generation to a text file (make sure it is WELL-FORMATTED so it is READABLE).

Task 1.7: Experiment!

You will now have the opportunity to test your creation by changing some of the parameters introduced in the making of this program. Note that all of these variables come

together in quite a complicated manner to determine how well your evolution progresses. Therefore it is best to only vary one variable at a time when trying to discover its effect on overall performance and overall time taken. Try varying the parameters in the list below, taking note of what combination of values seem to work best, as well as any observations made along the way. For every parameter value, or every combination of values, write your analysis in the report. Try to answer the questions below and motivate your choice for the most suitable value for each parameter:

1) Change the length of the target phrase
2) Change the number of population members
3) Change the mutation rate
4) Change the range of possible characters being considered (include numbers, etc)
5) Try changing between the two breeding methods - which one works better?
6) Change the mating factor - what benefit might we get from increasing this? What's a reasonable value for it?
7) Change the maximum generations - what happens to fitness over time?
8) Which function takes the longest to run? Can you improve its runtime at all?

Note: we recommend you allow the user to input values for these parameters, along with your suggested value for each. For example, you program can ask the user to enter a value for the population size:

```
What is your population size (suggested value: 200):
```

To make our code much more efficient at selecting the best parents, we can force our population to favor slightly fitter members much more than everyone else. After calculating our fitness, which are numbers from 0 to 1, we can raise it to some power, let's say 3. Now values which are *close* in fitness will be much easier to differentiate between, for example:

Original fitness:	0.33	0.30	0.27
fitness^3:	0.036	0.027	0.019

Although the fitness values themselves got smaller, the *relative difference* between them became larger. Remembering that the mating pool normalizes fitnesses to the maximum fitness, this will now award 0.33 *even more* tickets in the raffle compared to 0.30 or 0.27. In your script, try raising your entire fitness vector to different values before the mating pool is built (we'll call this the ' **exponential factor** '). How high can you make this value before it stops becoming beneficial? How might you want to adjust your mating factor (from Task 1.3) after introducing this exponential factor? Write your observations in the report.

Final thoughts

You may have noticed from Task 1.7 that if you make values such as the population size large enough, your code will run arbitrarily well. This is because if we are trying to breed a

20 character phrase, it won't be too hard to find perfect matches on all of those characters within a random population of 2000+ phrases. In machine learning, this is referred to as 'overfitting' your model: it's performing better than it should because you've given it more resources than it deserves. In the following tasks, you will have to keep the issue of overfitting in mind, in particular by not making the population size too large.

As we move from text to images none of the fundamental concepts are going to change, but serious increases in data and computational requirements are to be expected. We must keep this in mind when dealing with pictures, and be sure to avoid the following as much as possible to save on execution time: nested loops, recursion, and cells/structs . Using MATLAB's matrix capabilities will save time at all steps.

Painting the Mona Lisa

We will now be converting our above methodology into a generative algorithm that can breed pictures. Provided on Moodle are several small pictures to test your algorithm out on, including various sizes of the Mona Lisa. It is highly recommended that you start this section with black and white images only , it will save you lots of time in testing.

Task 2.1: Convert your code so it processes images instead of strings.

You will want to increase your population size to accommodate for the much larger genetic diversity you will need (note: try to **limit** the population value to the total size of your image to avoid overfitting). You may also want to find a new exponential factor that works for you.

Instead of printing our best phrase to the Command Window, we will now only print the generation number, the current maximum fitness, and the current average fitness of the population. Then, as seen on the first page, we will want to plot a sample of 9 maximum fitness members from evenly spaced generations (first subplot being Generation 1, last subplot being the final generation).

Your function should save the absolute maximum fitness organism seen during all generations to an image file. You should still save maximum fitness, average fitness, and genetic diversity for each generation to a text file.

Task 2.2: Improving the fitness function

Because we are now dealing with images, we won't necessarily want to have our fitness be calculated by *exact* matches of DNA. Instead, if a pixel's colour is 'close enough' (say +-10 on a scale of 0->255), we will count it as 'fit'. Add a new variable ***tolerance*** in your *calculateFitness* function to achieve this.

Because we are now dealing with images, there are new ways to analyze the fitness other than simply checking if the pixel is a match or not. We will add two new methods to evaluate fitness:

- by checking the average values around the pixels, and
- checking the rate of change between pixels (how quickly the color changes).

For the first method, you will first want to create a *meanFilter* function (preferably with no loops, to increase the speed) - https://en.wikipedia.org/wiki/Geometric_mean_filter
We will then compare the *meanFilter* version of each image to each other, with a similar tolerance as before.

For the second method, you may want to use the built-in function *diff* . This method should compare the rate of change between pixels for each image **in both directions** (up/down, left/right). You may want a different tolerance for this test than the previous two.

You will now have the four fitness values calculated:

1. Based on percent of values that fit within a tolerance range from the target image
2. Based on percent of mean values that fit within a tolerance range from the mean values of the target image
3. Based on rate of change up/down as compared with the target image
4. Based on rate of change left/right as compared with the target image

All these four values are from 0 -> 1, so in order to keep the total fitness between 0 and 1 by adding them in 'quadrature', which is a fancy word for Pythagoras' Theorem:

$$fit_{total} = \sqrt{fit_1^2 + fit_2^2 + fit_3^2 + fit_4^2}$$

Try experimenting with different combinations of your fitness evaluations. Which methods are the most effective? Which ones take the longest to run (remember tic-toc)?

Task 2.3: Improving mutation

Similar to the reasons why we introduced the tolerance variable in Task 2.2, we may also want our mutation to be more subtle than simply randomly mutating the pixel. Instead, we can change the mutation process to brightening or darkening a pixel by a random amount set within a range (say 30). This ***mutationRange*** should be saved as a variable.

We still do want *some* random mutation however, as certain bad spots that never were previously 'bred out' are close to impossible to mutate into a 'fit' range with the above process. Make it so there is still a 1/4 chance that a mutation will be completely random (you will want to have the ability to change this ' **random mutation rate** ' later, so as usual, save it in a variable).

Task 2.4: Convert to color if you have not already!
Hint : treat a color image as 3 separate black and white images for each R-G-B channel.

Task 2.5: Experiment...again!

Congratulations, you have just brought a whole new meaning to the term 'image generator'. I would highly recommend saving this copy of the code somewhere and create a second copy to make modifications and do experiments on! From here you can essentially change any variable that you've defined along the way your heart desires, and see which combination of parameters produces the most efficient image generator. Or come up with an innovative way to calculate fitness, or better ways to mutate, or anything! This is a largely unexplored problem you are now working on, and there are no 'right answers'.

The best/most innovative generation techniques will be awarded with extra credit.

We should keep in mind that we should not implement any methods which require knowing how good any *specific piece* of DNA is. All we can know in a genetic context is the overall fitness the the organism, and resulting statistics such as average fitness over the whole population. Simply saying 'keep the pixel if it is found to be good' is be too easy of an implementation to really simulate any sort of evolving process.

Submission Guidelines:

Create a report that should include the following:

1. Your name, student ID
2. Your analysis from task 1.7
3. Your analysis from task 2.5
3. A brief description of what you learned from working on this project. Be thoughtful, do not just restate the method.

Zip all your files (.m files, input and output image files, output text files, and report), name the zip file as <lastname>_<firstname>_GA.zip, and upload it to Moodle by Saturday January 26th, at 6 pm.

Additional resources:

Acknowledgement: This project was created by a former student/TA, Julian Lambert. See his notes below.

This project was inspired by Daniel Shiffman's Nature of Code. A video tutorial of Task 1.X is available on YouTube, although the language used to code is JavaScript. However, if you are struggling with anything conceptually, I would highly recommend watching this video:
https://www.youtube.com/watch?v=RxTfc4JLYKs&index=2&list=PLRqwX-V7Uu6bJM3VgzjNV5YxVxUwzALHV