Color Quantization with Genetic Algorithms

Leon Rische

February 18, 2020

Contents

1 Update [2019-12-12 Thu] 1
2 Setup 1
3 Organisms & Mapping 2
4 Evolutionary Loop 4
5 Results 5

1 Update [2019-12-12 Thu]

I removed the images in the post, I hope the code is still useful to someone.
I’m not programming crystal anymore, maybe I’ll add a new image at a later time.
For more information on this, see https://www.nature.com/articles/s41565-018-0337-2.

2 Setup

If you want to follow along, you’ll need to install crystal (or rewrite the code in some other language).
We need some way to read and write png images, so run shards init and add following to your shards.yml file:

language=yaml,label=,caption=,captionpos=b,numbers=none dependencies:
	stumpy,png: github: stumpycr/stumpy,pngversion: > 4.2.0

While reading the Handbook of Neuroevolution Through Erlang, I was surprise how simple the description of genetic algorithms (page 87) was:
1. Create a seed population

2. Create a fitness function

3. Loop until termination condition is reached
   (a) Evaluate each organism’s fitness
   (b) Choose the fit organisms from the population
   (c) Remove the unfit rest
   (d) Create offspring from the fit ones
   (e) Compose a new population from the offspring and their parents

Each organism needs a **genotype** (DNA), a **phenotype** (how it looks and behaves) and a function that map from one to the other.

I’ll use color quantization, representing images with a limited number of colors, as an example. The genotype is a list of colors, the phenotype is how the input image looks after it has been reduced to these colors.

3 Organisms & Mapping

random_color generates a random RGB color (duh) and seed_random creates a palette of **size** random colors.

```ruby
require "stumpy_png"
def random_color():
  RGBARGBA.from_relative(rand, rand, rand, 1.0)
defself.seed_random(size : Int32):
  self.new(Array.new(size) random_color)
```

quantize maps from genotype to phenotype by finding the color in the palette that is closest to the input color (has the smallest distance) for each pixel in the image. To save some calculations later, find_closest returns the distance and the closest color. Using the squared distance is faster and because $0 < a < b \implies \sqrt{a} < \sqrt{b}$, it won’t affect the result.

**Note:** there are better ways to calculate the distance between two colors, this one is easy to implement good enough for now.

```ruby
def distance(a : RGBA, b : RGBA):
  Convert to Int64 first, otherwise the subtraction could underflow
dr = a.r.to_i64 - b.rdg = a.g.to_i64 - b.gdb = a.b.to_i64 - b.b
```
dr * dr + dg * dg + db * db end

def find_closest(input_color : RGBA) : RGBA, Int64 best_color = RGBA::WHITE
  best_distance = Int64::MAX
  @colors.each do |color|
    distance = distance(input_color, color)
    if distance < best_distance
      best_color = color
      best_distance = distance
    end
  end

  Now we can build the mapping function, the output / phenotype is a Canvas, a 2D grid of RGBA pixels, with quantized colors. Because `find_closest` returns a tuple, we need to use `[0]` to extract the color.

  language=Ruby,label=,caption=,captionpos=b,numbers=none ...
  def quantize(image : Canvas) : Canvas
    Canvas.new(image.width, image.height) do |x, y|
      find_closest(image[x, y])[0]
    end
  end

  def write(image : Canvas, filename : String)
    StumpyPNG.write(quantize(image), filename)
  end

  The fitness function is pretty simple, we just iterate over all pixels of the image and sum up the errors / distances. If one organism is better than another, the value `fitness()` is smaller.

  language=Ruby,label=,caption=,captionpos=b,numbers=none ...
  def fitness(image : Canvas) : Int64
    image.pixels.reduce(0,i64) |
    acc, col | acc + find_closest(col)[1]
  end

  Lastly, we need some way to create offspring for the fittest organisms in our population.
  There are various ways to do this:

  1. Mutation, copy the genes with a small chance that stuff gets messed up
  2. Crossover, combine the genes of multiple parents
  3. A combination of both

  To keep the implementation as simple as possible, I’ll only use the first one.

  language=Ruby,label=,caption=,captionpos=b,numbers=none ...
  def produce_offspring(chance) : self.new(@colors.map{|color| mutate(color, chance)})
  def mutate(color, chance)
    rand < chance ? random_color : color
  end
4 Evolutionary Loop

path points to the image we want to quantize, @population_size tells it how many organisms should be in each generation, @colors is the length of each organism’s palette / genes and @mutation_chance can be used to adjust how probable mutations are.

@average_error stores a history of fitness values for the whole population, this is useful to see if the organisms actually improve.

@generations keeps track on how many generations have passed, why this is useful will become clear later.

During each step of the evolutionary loop, we sort the organisms by their fitness, store the average fitness in @average_error for later, take the fittest half of the organisms and create a new population with one offspring each. evolve_to just calls step until the target generation is reached.

write_images iterates over all organisms in the current population and renders their phenotypes, write_average_error creates a .csv file with the average error of each generation.

Math.log10 in combination with .rjust(length, '0') is just a trick to prepend numbers with 0 so that they are all equally long (e.g. 000=...=999= instead of 0...=999=), this way image viewer display them in the correct order.
def write_images(path) Dir.mkdir_p(path) unless File.directory?(path)
  length = Math.log10(@population.size).ceil.to_i
  @population.each_with_index do |palette, i|
    name = path+i.to_s.rjust(length,’0′)+’_.png’
    palette.write(@image, name)
  end
end

def write_avg_error(path) File.open(path,”w”) do |file|
  @average_error.each_with_index do |error, gen|
    file.puts [gen, error].join(“,”)
  end
end
end

5 Results

I’ll use a quantizer with a population of 12 organisms, each with 20 colors and a 1% mutation chance as an example.

The input image is the standard test image

language=Ruby,label=,caption=,captionpos=b,numbers=none
q=Quantizer.new("./lena.png", 12, 20, 0.01) q.write_images("images/initial/")
q.evolve_to(10) q.write_images("images/10/")
q.evolve_to(100) q.write_images("images/100/")
q.evolve_to(1000) q.write_images("images/1000/")
q.write_avg_error("avg_error.csv")

Initial (random seed population)

In the first (random) generation there already is one pretty good organism, the one at the top right.

After 10 generations it is the dominant version.

Finally, after 1000 generations and lots of mutations, we arrive at something that looks surprisingly close to the input image.

To confirm that the generations get better and better, we can plot the average errors collected along the way:
Images were created using the `montage` command that ships with [ImageMagick](https://imagemagick.org/).

Full source code: [l3kn/genetic_colorquantization](https://github.com/l3kn/genetic_colorquantization)