Assignment_7_Novelty_Search_[netid]

October 11, 2021

1 Assignment 7: Novelty Search

In our last assignment, we explored the idea of measuring diversity. This week we'll turn it up to eleven, and directly incentivize diversity by playing around with novelty search.

While not quite a deceptive landscape, we'll see how novelty search interacts with a rugged fitness landscape by revisiting our prior work on NK-landscapes from Assignment 3.

[ ]:
```
# imports
import numpy as np
import copy
import matplotlib.pyplot as plt
plt.style.use('seaborn')

import scikits.bootstrap as bootstrap
import warnings
warnings.filterwarnings('ignore') # Danger, Will Robinson! (not a scalable␣
↪hack, and may surpress other helpful warning other than for ill-conditioned␣
↪bootstrapped CI distributions)

import scipy.stats # for finding statistical significance

import time
```

Our NK fitness landscape function from Assignment 3.

[ ]:
```
class Landscape:
    """ N-K Fitness Landscape
    """



    def __init__(self, n=10, k=2):
        self.n = n # genome length
        self.k = k # number of other loci interacting with each gene
        self.gene_contribution_weight_matrix = np.random.rand(n,2**(k+1)) # for␣
↪each gene, a lookup table for its fitness contribution, which depends on␣
↪this gene's setting and also the setting of its interacting neighboring loci
```

```
    # find values of interacting loci
    def get_contributing_gene_values(self, genome, gene_num):
        contributing_gene_values = ""
        for i in range(self.k+1): # for each interacing loci (including the␣
↪location of this gene itself)
            contributing_gene_values += str(genome[(gene_num+i)%self.n]) # for␣
↪simplicity we'll define the interacting genes as the ones immediately␣
↪following the gene in question.  Get the values at each of these loci
        return contributing_gene_values # return the string containing the␣
↪values of all loci which affect the fitness of this gene

    # find the value of a partiuclar genome
    def get_fitness(self, genome):
        gene_values = np.zeros(self.n) # the value of each gene in the genome
        for gene_num in range(len(genome)): # for each gene
            contributing_gene_values = self.
↪get_contributing_gene_values(genome, gene_num) # get the values of the loci␣
↪which affect it
            gene_values[gene_num] = self.
↪gene_contribution_weight_matrix[gene_num,int(contributing_gene_values,2)] #␣
↪use the values of the interacting loci (converted from a binary string to␣
↪base-10 index) to find the lookup table entry for this combination of genome␣
↪settings
        return np.mean(gene_values) # define the fitness of the full genome as␣
↪the average of the contribution of its genes (and return it for use in the␣
↪evolutionary algoirthm)
```

1.0.1 Q1: Baseline implementation

Let's copy our usual `Individual` and `evolutionary_algorithm` setup from before. For simplicity in future questions, let's simply our algorithm as much as possible, working with bit-string (as per the NK fitness function), mutation only (just flipping one bit) and no crossover, and simple truncation selection rather than tournament selection. Like last week, let's also record genotypic diverity over time.

[ ]:
```
class Individual:

    def __init__(self, fitness_function, bit_string_length):
        ...

    def eval_fitness(self):
        ...
```

[ ]:
```

```

```
def evolutionary_algorithm(fitness_function=None, total_generations=100,␣
↪num_parents=10, num_children=10, bit_string_length=10,␣
↪num_elements_to_mutate=1, crossover=True):
    """ Evolutinary Algorithm (copied from the basic hillclimber in our last␣
↪assignment)

        parameters:
        fitness_funciton: (callable function) that return the fitness of a␣
↪genome
                          given the genome as an input parameter (e.g. as␣
↪defined in Landscape)
        total_generations: (int) number of total iterations for stopping␣
↪condition
        num_parents: (int) the number of parents we downselect to at each␣
↪generation (mu)
        num_childre: (int) the number of children (note: parents not included␣
↪in this count) that we baloon to each generation (lambda)
        bit_string_length: (int) length of bit string genome to be evoloved
        num_elements_to_mutate: (int) number of alleles to modify during␣
↪mutation (0 = no mutation)
        crossover (bool): whether to perform crossover when generating children

        returns:
        fitness_over_time: (numpy array) track record of the top fitness value␣
↪at each generation
    """

    # initialize record keeping
    solution = None # best genome so far
    solution_fitness = -99999 # fitness of best genome so far
    fitness_over_time = np.zeros(total_generations)
    solutions_over_time = np.zeros((total_generations,bit_string_length))
    diversity_over_time = np.zeros(total_generations)

    # the initialization proceedure
    ...

    # get population fitness
    ...

    ... # repeat

        # the modification procedure
        ...
            # inheretance
            ...
```

```
            # crossover
            # N/A

            # mutation
            ...

        ...

        # the assessement procedure
        ...

        # selection procedure
        ...

        # record keeping
        ...

    return fitness_over_time, solutions_over_time, diversity_over_time
```

Initialize recordkeeping

[ ]:
```
experiment_results = {}
solutions_results = {}
diversity_results = {}
```

1.0.2 Q1b: Baseline Results

Let's pull all the pieces together and run `20` repitions of `100` generations of a population with `20` parents and `20` children. Let's use a NK-landscape with a bitstring length (`N`) of `15` and a highly rugged landscape of `K = 14`. (My repitions take about 1.5 second each)

[ ]:
```
num_runs = 20
total_generations = 100
num_elements_to_mutate = 1
bit_string_length = 15
num_parents = 20
num_children = 20

n = bit_string_length
k = bit_string_length - 1

...
```

1.0.3 Q1c: Plotting

Please plot both the fitness over time and diversity over time of this run.

```
[ ]: def plot_mean_and_bootstrapped_ci_over_time(input_data = None, name = "change␣
     ↪me", x_label = "change me", y_label="change me", y_limit = None,␣
     ↪plot_bootstrap = True):
         """

         parameters:
         input_data: (numpy array of shape (max_k, num_repitions)) solution metric␣
     ↪to plot
         name: (string) name for legend
         x_label: (string) x axis label
         y_label: (string) y axis label

         returns:
         None
         """

         ...
```

```
[ ]: # plotting
     ...
```

1.0.4 Q2: Analysis

What do your results look like? In what ways, if any, do you expect that they'll change if we search for novelty instead of fitness? Why?

insert text here

1.0.5 Q3: Implementing Novelty

Let's implement novelty search! First, modify your `Individual` class to record `novelty` as an attribute (in additon to `fitness`).

Hint: As ususal, you may want to skip ahead to the modification of your `evolutionary_algorithm` function in `Q4` and come back to fill in the helper functions in `Q3` once you have a better idea of when they'll be used (they just appear first to establish defniitions in case you `Restart and Run All`).

```
[ ]: class Individual:

         def __init__(self, fitness_function, bit_string_length):
             ...
```

```
    def eval_fitness(self):
        ...
```

1.0.6 Q3b: Caclulate Novelty

Let's define the novelty of a solution to be the average hamming/euclidean distance (i.e. number of differing bits) between the closest `k` genomes to it within an archive of prior solutions. It may be helpful to define a helper function to calculate this quanity.

[ ]:
```
def get_novelty(solution_archive, individual, k):
    ...

    return ...
```

1.0.7 Q3c: Selecting for Novelty

Please modify your evolutionary algorithm code to select (again, using the truncation selection as above) for the most novel solutions in our population (according to the novelty metric defined above), regardless of their fitness.

In order to keep down the cost of computing the distance between a new genome and all those that have previously existed, let's also set a finite size to our novelty archive (as a parameter we can pass to the algorithm). When we trying to add new genomes to the novelty archive, only add the the new individual if it has a higher novelty score than an individual already in the novelty archive (and remove that prior individual from the archive to keeep the archive size the same). Let's also say that the novelty of an individual will be its novelty score when first being considered for additon to the archive (i.e. we do not have to re-calculate it in the future as the makeup of the archive changes).

It may also be helpful to build a helper function for (potentially) updating the archive with a new individual.

Feel free to use the indidivuals in the current generation for your archive calculation or not, whichever is more convenient for your implementation.

Hint: If you've sorted the population by novelty for selection, don't forget that the population will no longer be in order of fitness when you go to record the fitness of most fit indidivual for record keeping!

[ ]:
```
def update_archive(solution_archive, individual, max_archive_length):
    ...

    return solution_archive
```

[ ]:
```
def evolutionary_algorithm(fitness_function=None, total_generations=100,␣
↪num_parents=10, num_children=10, bit_string_length=10,␣
↪num_elements_to_mutate=1, crossover=True, novelty_k = 0):
```

```
    """ Evolutinary Algorithm (copied from the basic hillclimber in our last␣
↪assignment)

        parameters:
        fitness_funciton: (callable function) that return the fitness of a␣
↪genome
                          given the genome as an input parameter (e.g. as␣
↪defined in Landscape)
        total_generations: (int) number of total iterations for stopping␣
↪condition
        num_parents: (int) the number of parents we downselect to at each␣
↪generation (mu)
        num_childre: (int) the number of children (note: parents not included␣
↪in this count) that we baloon to each generation (lambda)
        bit_string_length: (int) length of bit string genome to be evoloved
        num_elements_to_mutate: (int) number of alleles to modify during␣
↪mutation (0 = no mutation)
        crossover (bool): whether to perform crossover when generating children

        returns:
        fitness_over_time: (numpy array) track record of the top fitness value␣
↪at each generation
    """

    ...

    return fitness_over_time, solutions_over_time, diversity_over_time
```

1.0.8 Q4: Run

Run novelty search with the same hyperparameter settings as above, and with a `novelty_k` value of `5` nearest neighbors for calculating the novelty metrics, from an archive of the `100` most novel individuals found thus far. (My runs take about 3 seconds for each repitition)

[ ]:
```
num_runs = 20
total_generations = 100
num_elements_to_mutate = 1
bit_string_length = 15
num_parents = 20
num_children = 20
novelty_k = 5

n = bit_string_length
k = bit_string_length-1

...
```

1.0.9 Q4b: Plot

Please visualize the fitness and diversity over time for novelty search vs. fitness-based search

[ ]:
```
# plotting
...
```

1.0.10 Q5: Analysis

How does novelty search perform in this domain? Is it what you expected? If no, why might that be the case?

insert text here

1.0.11 Q6: Larger Neighborhoods

How might you expect the result to change if you were to use a larger neighborhood (`novelty_k` value) for calculating a soluton's novelty.

Let's try it! Please run and plot the same settings as above, but with a `novelty_k` of `100` (i.e. using the full archive for novelty calcuation).

[ ]:
```
num_runs = 20
total_generations = 100
num_elements_to_mutate = 1
bit_string_length = 15
num_parents = 20
num_children = 20
novelty_k = 100

n = bit_string_length
k = bit_string_length-1

...
```

[ ]:
```
# plotting
...
```

1.0.12 Q6b: Analysis

What happened? Did it work better or worse? Is this what you expected (and why)?

insert text here

1.0.13 Q7: Mixed Fitness and Novelty

As suggested in class, perhaps the best version of an evolutionary algorithm is not one that selects just for fitness or one that selects just for novelty, but one that considers both in an attempt to carefully tradeoff exploration and exploitation.

We may not be the most careful and inentional with our tradeoffs here, but let's start with perhaps the simplest combination of selecting for both novelty and fitness one could think of. Let's select some of the individuals in our population on the basis of novelty and some on the basis of fitness.

In particular, pleae define a new parameter `novelty_selection_prop` that defines what proportion of the parents for the next generation will be selected by novelty (and choose the most novel solutions to occupy that portion of the new generation) while the remainder of the new population (`1-novelty_selection_prop`) gets selected on the basis of fitness – resulting in the same `num_parents` as before heading into the next generation.

[ ]:
```
def evolutionary_algorithm(fitness_function=None, total_generations=100,␣
↪num_parents=10, num_children=10, bit_string_length=10,␣
↪num_elements_to_mutate=1, crossover=True, novelty_k = 0,␣
↪novelty_selection_prop = 0):
    """ Evolutinary Algorithm (copied from the basic hillclimber in our last␣
↪assignment)

        parameters:
        fitness_funciton: (callable function) that return the fitness of a␣
↪genome
                          given the genome as an input parameter (e.g. as␣
↪defined in Landscape)
        total_generations: (int) number of total iterations for stopping␣
↪condition
        num_parents: (int) the number of parents we downselect to at each␣
↪generation (mu)
        num_childre: (int) the number of children (note: parents not included␣
↪in this count) that we baloon to each generation (lambda)
        bit_string_length: (int) length of bit string genome to be evoloved
        num_elements_to_mutate: (int) number of alleles to modify during␣
↪mutation (0 = no mutation)
        crossover (bool): whether to perform crossover when generating children

        returns:
        fitness_over_time: (numpy array) track record of the top fitness value␣
↪at each generation
    """

    ...

    return fitness_over_time, solutions_over_time, diversity_over_time
```

1.0.14 Q8: Experimentation

Let's try running this mixed selection criteria for a 50/50 split between suvivors/parents for the next generation selected via novelty vs. fitness. Let's do this with our original novelty neighbhorhood of size 5, and all other parameters the same.

As usual, please plot fitness and diversity afterwards.

[ ]:
```
num_runs = 20
total_generations = 100
num_elements_to_mutate = 1
bit_string_length = 15
num_parents = 20
num_children = 20

novelty_k = 5
novelty_selection_prop = 0.5
max_archive_length = 100

n = bit_string_length
k = bit_string_length-1

...
```

[ ]:
```
# plotting
...
```

1.0.15 Q9: Analysis

What happened (to both fitness and diversity)? Are you surprised? Why would this be?

insert text here

1.0.16 Q10: Balancing Novelty and Fitness

Let's run this again with a different balance of novelty vs. fitness. Please run it with 90% of survivors selected via novelty and just 10% selected via fitness, and also vise versa with just 10% novelty and 90% fitness at each generation. Which do you expect to work better?

insert text here

1.0.17 Q9b: Running and Visualization

Let's findout!

[ ]:
```
num_runs = 20
total_generations = 100
```

```
num_elements_to_mutate = 1
bit_string_length = 15
num_parents = 20
num_children = 20

novelty_k = 5
novelty_selection_prop = ...
max_archive_length = 100

n = bit_string_length
k = bit_string_length-1

...
```

[ ]:
```
# plotting
...
```

1.0.18 Q10: Analysis

Did the experiment turn out the way you thought it would? Why or why not? What does this imply about the use of novelty vs. fitness in expoitation vs. exploration? Do the diveristy plots support this idea?

insert text here

1.0.19 Q11: The Effect of Ruggedness

How much do you think the conclusions you came to above are the result of the particular (maximally rugged) fitness landscape we experiemented with? What would happen if we used a much smoother landscale (e.g. a NK landscape with `K=0`)

insert text here

1.0.20 Q12: Experiementation

Let's find out! Please pick your best ratio of novelty vs. fitness selection, and compare it to purely fitness and purely novelty selection on a NK landscale with `K=0`. Please plot your results.

[ ]:
```
num_runs = 20
total_generations = 100
num_elements_to_mutate = 1
bit_string_length = 15
num_parents = 20
num_children = 20

novelty_k = 5
```

```
novelty_selection_prop = ...
max_archive_length = 100

n = bit_string_length
k = 0


...
```

[ ]:
```
# plotting
...
```

1.0.21 Q12b: Analysis

Were you right? Was novelty, or novelty+fitness helpful? Was it harmful? What was the effect on diversity?

insert text here

1.0.22 Q13: Future Work

In this assignment we explored just one (very simple) way to combine novelty and fitness, how else might you want to do this that could be more effetive (and why)?

insert text here

1.0.23 Congratulations, you made it to the end!

Nice work – and hopefully you're starting to get the hang of these!

Please save this file as a .ipynb, and also download it as a .pdf, uploading **both** to blackboard to complete this assignment.

For your submission, please make sure that you have renamed this file (and that the resulting pdf follows suit) to replce `[netid]` with your UVM netid. This will greatly simplify our grading pipeline, and make sure that you receive credit for your work.

Academic Integrity Attribution During this assignment I collaborated with:

insert text here