Assignment_3_Tunably_Rugged_Landscapes_[netid]

September 13, 2021

1 Assignment 3: Tunably Rugged Landscapes

In our assignment last week we got our first hillclimber up and running, while in class week started to talk about fitness landscapes to begin thinking about search spaces, and population-based evolutionary algorithms to start complexifying how we traverse these search spaces. In this week's assignment, we'll start to put these two things together and begin toying around with the pandora's box of algorithmic experimentation.

In particular, we'll explore the idea of generating parameterized fitness functions to being to explore the relationship between the type of problem we're trying to solve, and what features our evolutionary algorithm should have to solve it.

Note: I know this looks like a lot of coding! While we are building valuable infastructure here, much of the solutions here are modifications on prior work (from eariler in this assignment or the last one), and can largely be copy-and-pasted here, or written once as a function to call again later. Despite this, it's still always a good idea to start in on assignments early (even if just reading through all the questions to estimate how long it might take you to complete)

[ ]:
```
# 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
```

1.0.1 N-K Landscape

In general, you'll be more likely to have a problem provided to you, rather than have to design a fitness funciton by hand. So in this week's assignment, I'll provide the full fitness-landscape-generating function for you. The below function implements Kaffman's N-K Landscape. While it's not entirely necessary for you to understand every implementaton detail below, the N-K landscape idea is chosen because it's a particularly interesting toy problem – and more reading on it can be found via many online resources (e.g. Kauffman and Weinberger's *The NK model of rugged fitness landscapes and its application to maturation of the immune response* – inlcuded in the assigment zip folder as it is firewalled online)

The main things to know about the NK model are that: It is a model of a tunably rugged fitness landscape, that means we have parameters that can affect the shape and ruggedness of the fitness landscape produced by this model. While there are many variations, here we follow the original (simplest) model that includes just two parameters: **N** defines the length of the binary bit string genome, while **K** defines the ruggedness of the landscape (in particular how the fitness of each allele depends on other loci (nearby genes) in the genotype.

Note: This is fully implemented and no action is needed from you, besides running the code block.

[ ]:
```
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)
```

1.0.2 Hillclimber

Based on the hillclimber function from you last assignment (and informed by the posted solution, if you wish), copy an slightly modify the hillclimber to use this fitnes function. For sake of running multiple trials, also please modify the record keeping to reutrn the solutions after the completion of the algorithm rather than printing them out during evolution.

Hint: In python, functions can be treated as objects (e.g. passed as an argument to another function)

[ ]:
```
def hillclimber(total_generations = 100, bit_string_length = 10,␣
↪num_elements_to_mutate= 1, fitness_function=None):
    """ Basic hillclimber, copied from last assignment

        parameters:
        total_generations: (int) number of total iterations for stopping␣
↪condition
        bit_string_length: (int) length of bit string genome to be evoloved
        num_elements_to_mutate: (int) number of alleles to modify during␣
↪mutation
        fitness_funciton: (callable function) that return the fitness of a␣
↪genome
                                       given the genome as an input parameter (e.g. as␣
↪defined in Landscape)

        returns:
        solution: (numpy array) best solution found
        solution_fitness: (float) fitness of returned solution
        solution_generation: (int) generaton at which most fit solution was␣
↪first discovered
    """

    # the initialization proceedure
    ...

    # initialize record keeping
    ...

```

```
    ... # repeat

        # the modification procedure
        ...

        # the assessement procedure
        ...

        # selection procedure
        ...

        # record keeping
        ...

    return solution, solution_fitness, solution_generation
```

1.0.3 Q1: Landscape Ruggedness's effect on Hillclimbing

In class we discussed the potential for the fitness landscape to greatly affect a given search algorithm. Let't start by generating varyingly rugged landscapes, and investigating how this impacts the effectiveness of a standard hillclimber.

For each value of `k = 0..14` and a genome legnth of `15` please generate 100 unique fitness landscapes, and record the fitness value and time to convergence (when the most fit solution was found) for the hillclimber algorithm above on that landscape. Print out the mean results for each `k` as you go to keep track of progress. This output may look something like this:

[ ]:
```
# hyperparameters
n=15; max_k=15; repetitions = 100

# initialize array to record results over different settings of k and repeated␣
↪trials
solutions_found = np.zeros((max_k,repetitions,n))
fitness_found = np.zeros((max_k,repetitions))
generation_found = np.zeros((max_k,repetitions))

# initilize output
print(' k  mean fitness  mean generation found')
print('--  ------------  ---------------------')

... # for many values of k
    ... # for many repeated (independent -- make sure your results differ each␣
↪run!) trials
        ... # generate a random fitness landscape with this level of ruggeddness
        ... # run a hillclimber
```

```
        # record outputs
        ...

    # print average results for all repitions of this k
    ...
```

Let's also record this result in a nested dictionary to be able to recall it later (for comparison to other results). There is an implementation given below, but you're welcome to use `pandas` if you're more comforatable with that library for data manipulation and visualization.

```
[ ]: experiment_results = {}
     experiment_results["hillclimber"] = {"solutions_found":solutions_found,␣
      ↪"fitness_found":fitness_found, "generation_found":generation_found}
```

1.0.4 Q2: Plotting Results

Please visualize the above terminal output in a figure (feel free to recycle code from previous assignments). You'll be generating this same plot many time (and even comparing multiple runs on a single figure), so you may want to invest in implementing this as a function at some point during this assigment – but that is not strictly necessary now, and fell free to ignore the code stub below.

In particular, please plot the `Time to Convergence (Generations)` and `Fitness` values (as you vary `K`) as two separate figures, as a single figure with multiple y-axes is messy and confusing. Please include 95% boostrapped confidence intervals over your 100 repitions for eack `K`. Please also inlcude the title of each experiment as a legend (for now just `hillclimber` is sufficient for this baseline case, and titles will make more sense in follow up experimental conditions).

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

         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
         """

         ...
```

1.0.5 Q3: Analysis of Hillclimber on Varying Ruggedness

What do you notice about the trend line? Is this what you expected? Why or why not?

insert answer here

1.0.6 Q4: Random Restarts

One of the methods we talked about as a potential approach to escaping local optima in highly rugged fitness landscapes was to randomly restart search. Using the same number of total generations (`100`), please implement a function which restarts search to a new random initialization every `20` generations (passing this value as an additoinal parameter to your hillclimber function). Feel free to just copy and paste the hillclimber code block here to modify, for the sake of simplicity and easy gradability.

```
[ ]: def hillclimber(total_generations = 100, bit_string_length = 10,␣
     ↪num_elements_to_mutate= 1, fitness_function=None, restart_every = None):
         """ Basic hillclimber, copied from last assignment

             parameters:
             total_generations: (int) number of total iterations for stopping␣
     ↪condition
             bit_string_length: (int) length of bit string genome to be evoloved
             num_elements_to_mutate: (int) number of alleles to modify during␣
     ↪mutation
             fitness_funciton: (callable function) that return the fitness of a␣
     ↪genome
                                         given the genome as an input parameter (e.g. as␣
     ↪defined in Landscape)
             restart_every: (int) how frequently to randomly restart the hillclimber

             returns:
             solution: (numpy array) best solution found
             solution_fitness: (float) fitness of returned solution
             solution_generation: (int) generaton at which most fit solution was␣
     ↪first discovered
         """

         ...

         return solution, solution_fitness, solution_generation
```

1.0.7 Q4b: Run Experiment

Slightly modify (feel free to copy and paste here) your experiment running code black above to analyze the effect of modifying `K` on `Time to Convergence (Generations)` and `Fitness`, again print progress and plotting results. Please also save these results (and subsequent new ones) to your `experimental_results` dictionary for later use.

[ ]:
```
# hyperparameters
n=15; max_k=15; repetitions = 100


...
```

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

1.0.8 Q5: Analysis of Random Restarts

What trends do you see? Is this what you were expecting? How does this compare to the original hillclimber algorithm without random resets (please not any y-axis differences when comparing values/shapes of the curves)?

insert text here

1.0.9 Q6: Modifying mutation size

We've talked about a number of other potential modifications/complexifications to the original hillclimber aglorithm in class, so let's experiment with some of them here. Here, please modying your above a hillclimber (again please just copy and paste the code block here) to mutate multiple loci when generating the child from a parent.

Hint: Be careful of the difference between modifying multiple genes and modifying the same gene multiple times

[ ]:
```
def hillclimber(total_generations = 100, bit_string_length = 10,␣
↪num_elements_to_mutate= 1, fitness_function=None, restart_every = None):
    """ Basic hillclimber, copied from last assignment

        parameters:
        total_generations: (int) number of total iterations for stopping␣
↪condition
        bit_string_length: (int) length of bit string genome to be evoloved
        num_elements_to_mutate: (int) number of alleles to modify during␣
↪mutation
        fitness_funciton: (callable function) that return the fitness of a␣
↪genome
                                    given the genome as an input parameter (e.g. as␣
↪defined in Landscape)
        restart_every: (int) how frequently to randomly restart the hillclimber

        returns:
        solution: (numpy array) best solution found
```

```
        solution_fitness: (float) fitness of returned solution
        solution_generation: (int) generaton at which most fit solution was␣
↪first discovered
    """

    ...

    return solution, solution_fitness, solution_generation
```

1.0.10 Q6b: Expectations

In this experiment, let's set the number of elements to be mutated to `5` when generating a new child.

Before running the code, what do (did) you expect the result to be based on the results of the original hillclimber, the random restart condition, and the implications that a larger mutatoin rate may have?

insert text here

1.0.11 Q7: Run experiment

Run the experiment and visualize (similar to **Q4b**, and feel free to copy a paste here again) to analyze the effect of a larger mutation size on the realationship between `K` and `Time to Convergence (Generations)` / `Fitness`.

[ ]:
```
...
```

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

1.0.12 Q7b: Analysis

Is this what you expected/predicted? If not, what is different and why might that be?

insert text here

1.0.13 Q8: Accepting Negative Mutations

Another way we might be able to get out of local optima is by taking steps downhill away from that optima. Add another arguement (`downhill_prob`) to your `hillclimber` function, which accepts a child with a negative mutataion with that given probability.

[ ]:
```

```

```python
def hillclimber(total_generations = 100, bit_string_length = 10,␣
↪num_elements_to_mutate= 1, fitness_function=None, restart_every = None,␣
↪downhill_prob=0):
    """ Basic hillclimber, copied from last assignment

        parameters:
        total_generations: (int) number of total iterations for stopping␣
↪condition
        bit_string_length: (int) length of bit string genome to be evoloved
        num_elements_to_mutate: (int) number of alleles to modify during␣
↪mutation
        fitness_funciton: (callable function) that return the fitness of a␣
↪genome
                                given the genome as an input parameter (e.g. as␣
↪defined in Landscape)
        restart_every: (int) how frequently to randomly restart the hillclimber
        downhill_prob: (float) proportion of times when a downhill mutation is␣
↪accepted

        returns:
        solution: (numpy array) best solution found
        solution_fitness: (float) fitness of returned solution
        solution_generation: (int) generaton at which most fit solution was␣
↪first discovered
    """

    ...

    return solution, solution_fitness, solution_generation
```

1.0.14 Q8b: Run the experiment

Same as above (run and plot), but now investigating the effect of a `downhill_prob` of `0.1` (10% chance) on relationship between ruggedness and performance

[ ]:
```python
...
```

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

1.0.15 Q9: Visualizing Mulitple Runs

On the same plot (which may require modifying or reimplementing your plotting function, if you made one above), please plot the curves for all 4 of our experiments above on a single plot (including bootsrapped confidence intervals for all).

Hint: Legends are especially important here!

Hint: It may be convenient to iterate over the dictionaries, turning them into lists before plotting (depending on your plotting script)

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

1.0.16 Q9b: Analyzing Mulitple Runs

Do any new relationships or questions occur to you as you view these?

insert text here

1.0.17 Q10: Statistical Significance

Using the `ranksums` test for significance, please compare the values for each algorithm at `K=14` using your saved `experiment_results`, reporting the p-value for each combination of the 4 experiments. Please do this for both the resulting fitness values, and the generation for which that solution was found. The output may look something like this:

[ ]:
```
# test for statistical significance across treatments
k = 14


...
```

1.0.18 Q11: Hyperparameter Search

Its cool to see the differences that these approaches have over the baseline hillcimber, but the values for each parameter that we've asked you to investigate are totally arbitrarily chosen. For example, who's to say that doing random resets every `20` generations is ideal? So let's find out!

Please modify the code above for which you varied `K` to see the effect on `Fitness` and `Time to Convergence (Generations)`, to now keep a constant `K=14` and vary how frequently do you random resets within the fixed `100` generations of evolution. Explore this relationship for values of resets ranging from never (`0`) up to every `29` generations.

[ ]:
```
...
```

1.0.19 Q11b: Visualization

Similar to before (with `K`), please plot `Fitness` and `Time to Convergence (Generations)` as a funciton of how frequently we apply random restarts (`Restart Every`)

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

1.0.20 Q11c: The effect of ruggedness

The above plots are for a single value of `K`=14. Repeat this same experiment below, just changing the value of `K` to `0`, to see what this experiment looksl like on a less-rugged landscape.

[ ]:
```
...
```

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

1.0.21 Q12: Analysis

What trends to you see from the figs for `K=14` vs. `K=0`? Are you surprised by this? What does it imply about the relationship between ruggedness and random restarts? Does it make you want to try and other experiments (what would be the next thing you'd investigate)?

insert text here

1.0.22 Congratulations, you made it to the end!

Wow that was a bit of a long one. Hopefully you enjoyed the open-ended experimentation though

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