Homework 4: Higher-order Functions

Due: 02.19.2021 11:59 PM ET

This homework primarily covers defining and using higher-order functions.

Goals

In this homework you will:

1. Write higher order functions that accept one or more functions as an input
2. Write higher order functions that return a function as output
3. Develop a basic signal processing algorithm (cross-correlation)

Background

Higher Order Functions

This homework will get you familiar with several ways to build and use higher order functions. Before attempting it, please review the notes from class on higher order functions. Feel free use and adapt any of the code that you have seen in class.

Stencils and cross-correlation

In Problem 2, we will ask you to write two functions. The first is a generic *stencil* function, and the second is a function that generates *box filters* that, when combined, will let you perform cross-correlation.

Stencils

A stencil (in the most general sense) is a way of computing a value for a given data point by using the values of neighboring data points. Suppose we have a *stencil function*, `f`. `f` is a function that accepts a list of some fixed length (for our purposes, let's say 3) and outputs a single value that is some combination of the values in that list. Here is one example:

```
def f(l) :
    return l[0] * l[1] + l[2]
```

Applying a stencil function means treating the stencil function like a "window" and "sliding" it across a list, computing the result of the stencil function at each point. So if I start with the following list:

```
[a, b, c, d, e, f, g]
```

Applying the stencil function produces the following output list:

```
[f([a, b, c]), f([b, c, d]), f([c, d, e]), f([d, e, f]), f([e, f, g])]
```

> Note two things: 1. The length of the output list is smaller than the length of the input list. If the input list is length `k`, the output list will be length `k - width + 1`, where `width` is the width of the stencil that `f` applies. 2. You can think of a stencil as a generalization of map that uses neighboring elements to compute the output instead of just the element itself. If `f` has a window of width 1, the output is basically like using map.

We will ask you to write a function for applying stencils that takes in an input list, a stencil function, and a third parameter that tells you how wide the stencil function is.

Box filters and cross correlation

A specific kind of stencil function is called a *box filter*. This is a filter of length `k` that specifies a list of `k` numbers (for a 1-D box filter) and computes its result by multiplying the first element of the filter by the first element of the input, the second element of the filter by the second element of the input, and so on, and then adding all the results together.

So if you have a box filter with elements `b[0], b[1], b[2]` and you applied it to the list `[a, b, c, d, e, f, g]`, you would get:

```
[b[0] * a + b[1] * b + b[2] * c,
 b[0] * b + b[1] * c + b[2] * d,
 b[0] * c + b[1] * d + b[2] * e,
 b[0] * d + b[1] * e + b[2] * f,
 b[0] * e + b[1] * f + b[2] * g]
```

In one dimension, applying a box filter in this way computes the (discrete) cross-correlation of the signal in the input list with the signal in the filter. This is a measure of similarity between the two signals.

In two dimensions, box filters can be used to perform image-processing tasks like blurring and edge detection. These types of box filters are also one of the key steps in deep neural networks that operate on images (they form the core of the "convolution" layers).

In the second part of Problem 2, we will ask you to write a function that, when given a list of box filter elements, *creates a stencil function* from those elements.

Instructions

1) Problem 1: Higher-order functions

In this problem, we ask you to fill in the code for a number of missing functions in `problem1.py`:

1. `composeMap`: This takes in *two* functions, `fun1` and `fun2`, and a list, `l`, as its arguments and returns a list, which consists of applying `fun1` then `fun2` to each element of the input list, `l`. **Note that you must apply the functions in that order: call `fun1` first, then `fun2` on the output of `fun1`** for each element in the input list `l`.
2. `tripleMap`: This takes in a function, `fun`, and returns a list, which consist of applying `fun` **thrice** to each element of the input list, `l`. You may find it useful to implement `tripleMap` using `composeMap`.
3. `compose`: This takes in two functions and returns a new function. The new function takes in a single argument, `i`, then calls `fun2` followed by `fun1` on that argument, returning the result. Note again that the order matters.
4. `repeater`: This takes in a function, `fun` and an integer, `num_repeats`, and returns a new function. The new function takes in an input `x` and calls `fun` on it `num_repeats` times (in other words, you call `fun` on the output of calling `fun` on the output of calling `fun` ... on the input, repeated as many times as `num_repeats`).

Testing

If your code works, and you run the test code provided in `problem1.py`, you should get the following output:

```
[1, 7, -23, -17, -17, -9, -5, 15, 13, -15]
[-2, 4, -26, -20, -20, -12, -8, 12, 10, -18]
[16, 40, -80, -56, -56, -24, -8, 72, 64, -48]
```

```
2
[-2, 4, -26, -20, -20, -12, -8, 12, 10, -18]
5
[1, 7, -23, -17, -17, -9, -5, 15, 13, -15]
repeat 0 times: 5
repeat 1 time: 10
repeat 2 times: 20
repeat 3 times: 40
```

2) Problem 2: Stencils

In this problem, we ask you to fill in the two functons in `problem2.py`:

1. `stencil(data, f, width)`: This function takes in a list, `data`, a function, `f`, and an int, `width`, and returns a list. This function returns the result (an output list of length `k-width+1`) of applying the stencil function `f` to the input list `data` as described in the Background section.
2. `createBox(box)`: This function accepts a list, `box`, and returns two outputs: a new function and a width (int). The width is the length of the box (the number of elements that the filter looks at). The new function is a stencil function that operates on `len(box)` items from an input list and applies a box filter to them as described in the Background section. The function should check, at the beginning, if the length in the input parmeter is the same as the length of `box`. If not, the following error should be printed:

   ```
   Calling box filter with the wrong length list.  Expected
   list of length should be n.
   ```

   where `n` is the length of `box`. For example if `n=7`, the printed warning should be:

   ```
   Calling box filter with the wrong length list.  Expected
   list of length should be 7.
   ```

Testing

If your code works, and you run the test code provided in `problem2.py`, you should get the following output:

```
[-1.0, -4.0, -8.0, -5.666666666666667, -3.6666666666666665, 1.6666666666666667, 5.3333333333
[227, 232, 208, 189, 204, 191]
[-1.0, -3.9999999999999996, -7.999999999999999, -5.666666666666666, -3.6666666666666665, 1.6
[-4.5, -6.0, 3.5, 3.0, 8.0, 5.5, -2.5]
```

(The floating point numbers may be rounded a little bit differently, depending on exactly how you implement your box filter.)

What to Submit

For Problem 1, please submit `problem1.py` with all the appropriate functions filled in.

For Problem 2, please submit `problem2.py` with all the appropriate functions filled in.

Submitting your code

Your final submission should have modified `problem1.py` and `problem2.py` files. Ensure to push all changes to your repository by 11:59PM on 2/19/2021.