ECE60146: Deep Learning HW1

Joseph Lee

January 2026

1 Questions

1.1 Question 1

```python
import math

#q1
class SignalProcessor(object) :
    def __init__ (self, data) :
        self.data = data
    def __eq__ (self, other):
        if len(self.data) != len(other.data):
            raise ValueError("Two signals are not equal in length!")
        count = 0 # keep track of number of "equal" values within tolerance
        for i,j in zip(self.data, other.data):
            #need to make it within tolerance of 0.01
            count += (i == j) or (abs(i-j) < 0.01)
        return count
```

As outlined in the assignment, the first question requires to create a base class called `SignalProcessor` that upon initialization takes in data. Since it is the parent class, it does not inherit from any previously created classes, which is why it's `SignalProcessor(object)`. The variable `data` is saved as `self.data` so that it be used for future functions. For initialization, the inclusion of the data parameter without a default means we require a data input upon creation.

NOTE: Other parts of the class are there for working with other parts of the homework.

1.2 Question 2

```
28  #q2
29  class SineWaveFunction(SignalProcessor):
30      def __init__(self, amplitude, frequency):
31          SignalProcessor.__init__(self, None)
32          self.amplitude = amplitude
33          self.frequency = frequency
34      def __call__(self, duration):
35          ls = [i for i in range(duration)] #keep track of indices to use in func eq.
36          self.data = [self.amplitude * math.sin(2 * math.pi * self.frequency * n) for n in ls]
37          print(f'Sinewave data is: {self.data}') #print created data
38          return self.data
39      def __len__(self):
40          return len(self.data)
41      def __iter__(self):
42          return SineWaveFunctionIterator(self)
```

This time, for Q2, we do inherit from a previously created class, so we signify in Line 29 of the source code that `SineWaveFunction` is calling from `SignalProcessor`. We also initalize the parent class in Line 31 so we can use its functionality if needed. Moreover, since the question asks to save two inputted parameters (amplitude and frequency) upon initialization, we save them via using `self.` for future use (similar to Q1).

NOTE: Same as Q1, other parts of the class outside of `__init__` are not required for this question, but are used in subsequent problems.

1.3 Question 3

```
28  #q2
29  class SineWaveFunction(SignalProcessor):
30      def __init__(self, amplitude, frequency):
31          SignalProcessor.__init__(self, None)
32          self.amplitude = amplitude
33          self.frequency = frequency
34      def __call__(self, duration):
35          ls = [i for i in range(duration)] #keep track of indices to use in func eq.
36          self.data = [self.amplitude * math.sin(2 * math.pi * self.frequency * n) for n in ls]
37          print(f'Sinewave data is: {self.data}') #print created data
38          return self.data
39      def __len__(self):
40          return len(self.data)
41      def __iter__(self):
42          return SineWaveFunctionIterator(self)
43
44  #q3
45  print("#========== Q3 ==========#") #need this, looks to disorganized otherwise
46  SG = SineWaveFunction(amplitude=2.0, frequency=0.1)
47  SG(duration=5)
48  print(len(SG))
```

Using the same class as in Q2, we can now explain the `__call__` function's usage. Whenever a created class is used in a manner of a function call, i.e., `SineWaveFunction(...)`, it will look for a `__call__` function. `__call__` for this class requires a duration parameter, which we use to implement the formula provided in the homework question. The first line of the function creates a list of indices using Python list comprehension in the range of 0 to duration-1, which

will mimic n in the equation given in the homework. The second line loops over these indices and calculates the equation on a per-element basis. Likewise, list comprehension is used here to create the list in a compact manner, where it populates the list `len(duration)` times, utilizing the equation as the populated value. Finally, we print the sine-wave data via the saved `self.data` variable.

Next, when we call `len(class)`, it will look for the class' `__len__` function. For this function, since `self.data` is a list, we can just use the in-built `len()` function to find the number of items in the data list. Alternatively, a for-loop over the list could've been done, should a more manual method been preferred.

With these implementations, doing the last two lines of the above source code should (i) give us a list with 5 elements, with an oscillating behavior characteristic of a sine function, and (ii) print 5, as that is the length of the duration. We see from the screenshot below of the output, both of these characteristics hold with this implementation.

```
#========== Q3 ==========#
Sinewave data is: [0.0, 1.1755705045849463, 1.902113032590307, 1.9021130325903073, 1.1755705045849465]
5
```

1.4 Question 4

```
16  class SineWaveFunctionIterator: #use as an iterator to loop through items in other classes
17      def __init__(self, xobject):
18          self.items = xobject.data
19          self.index = 0
20      def __iter__(self):
21          return self
22      def __next__(self):
23          self.index += 1
24          if self.index < len(self.items)+1:
25              return self.items[self.index-1] # need to do -1 since index was +1ed
26          else: # we reached the end of the list
27              raise StopIteration
```

```
28  #q2
29  class SineWaveFunction(SignalProcessor):
30      def __init__(self, amplitude, frequency):
31          SignalProcessor.__init__(self, None)
32          self.amplitude = amplitude
33          self.frequency = frequency
34      def __call__(self, duration):
35          ls = [i for i in range(duration)] #keep track of indices to use in func eq.
36          self.data = [self.amplitude * math.sin(2 * math.pi * self.frequency * n) for n in ls]
37          print(f'Sinewave data is: {self.data}') #print created data
38          return self.data
39      def __len__(self):
40          return len(self.data)
41      def __iter__(self):
42          return SineWaveFunctionIterator(self)
```

```
50  #q4
51  print("#========== Q4 ==========#") #need this, looks to disorganized otherwise
52  SG = SineWaveFunction(amplitude=2.0, frequency=0.1)
53  SG(duration=5)
54  print([val for val in SG])
```

Since the initialization and `SG(duration=5)` is the same as the previous problem, the first output from the above code snippet should be same as the data output from Question 1.3, which we see to be true.

For the iteration over the values in `SG`, we would like to iterate over the data, with the created list in the last line of code producing the same list as the first output. Firstly, we see that the `__iter__` function for `SineWaveFunction`, which is called when it is looped over as done with `val for val in SG`, it calls a `SineWaveFunctionIterator`, which is designed to specifically loop over the data. Since the `self` in `SineWaveFunctionIterator(self)` in the `__iter__` function for `SineWaveFunction` is actually a `SineWaveFunction`, the `xobject` in the iterator class is a `SineWaveFunction`. Therefore, it can access the items in `data`, which it can save in its own variable called `items`. Then, as we loop over the items in the iterable, it's calling `__next__` repeatedly, which does two things: returning the appropriate item in the item/data list, and updating the index position so that we progress through the data list. In addition, when we have read through the entire data, we raise a `StopIteration` to indicate that we have iterated through all of the data. As we can see from the output below, the second list that utilizes the iterable does indeed give the same list out.

```
#========= Q4 ==========#
Sinewave data is: [0.0, 1.1755705045849463, 1.902113032590307, 1.9021130325903073, 1.1755705045849465]
[0.0, 1.1755705045849463, 1.902113032590307, 1.9021130325903073, 1.1755705045849465]
```

1.5 Question 5

```python
#q5
class SquareWaveFunction(SignalProcessor):
    def __init__(self, amplitude, frequency):
        SignalProcessor.__init__(self, None)
        self.amplitude = amplitude
        self.frequency = frequency
    def __call__(self, duration):
        ls = [i for i in range(duration)]
        self.data = []
        for n in ls:
            # if equation >= 0, use positive else use negative of amp.
            val = self.amplitude if math.sin(2*math.pi*self.frequency * n) >= 0 \
                else -self.amplitude
            self.data.append(val)
        print(f'SquareWave data is: {self.data}') #print created data
        return self.data
    def __len__(self):
        return len(self.data)
    def __iter__(self):
        return SineWaveFunctionIterator(self)

print("#=========== Q5 ===========#") #need this, looks to disorganized otherwise
SW = SquareWaveFunction(amplitude=3.0, frequency=0.1)
SW(duration=5)
print(len(SW))
print([val for val in SW])
```

We now create a new class, the `SquareWaveFunction`, which now implements the square wave, which records positive amplitude if the given equation is positive for an element n, or its negative otherwise. That is why in the `__call__` function of this class, we update the value of `val` to be positive or negative given the value of `math.sin(2 * math.pi * self.frequency * n)`. Lastly, in `__call__`, since the data is a list structure, we can use the in-built append function to add elements to the list one by one as we iterate over the indices. The approach to implementing the other functions / attributes of this class follow the same flow as previous questions. Firstly, since we made the duration of `SW` equal to 5, the printing of the length of `SW` should give 5, which we see to be the case in the screenshot below. We also should see that the iterating over the class should give the same list as the printed output from the `__call__` function, which is the case. The behavior of the output is also consistent of a square wave, as you expect a sequence of same values before sharply dropping or increasing, creating a square-like shape.

```
#=========== Q5 ===========#
SquareWave data is: [3.0, 3.0, 3.0, 3.0, 3.0]
5
[3.0, 3.0, 3.0, 3.0, 3.0]
```

1.6 Question 6

```python
import math

#q1
class SignalProcessor(object) :
    def __init__ (self, data) :
        self.data = data
    def __eq__ (self, other):
        if len(self.data) != len(other.data):
            raise ValueError("Two signals are not equal in length!")
        count = 0 # keep track of number of "equal" values within tolerance
        for i,j in zip(self.data, other.data):
            #need to make it within tolerance of 0.01
            count += (i == j) or (abs(i-j) < 0.01)
        return count
```

```python
#q6
print("#========== Q6 ==========#") #need this, looks to disorganized otherwise
SG1 = SineWaveFunction(amplitude=2.0, frequency=0.1)
SG1(duration=5)
SG2 = SineWaveFunction(amplitude=2.0, frequency=0.15)
SG2(duration=5)
print(f'SG1 == SG2 test: {SG1 == SG2}')
# now do it on SquareWaveFunction
SW1 = SquareWaveFunction(amplitude=3.0, frequency=0.1)
SW1(duration=5)
SW2 = SquareWaveFunction(amplitude=3.0, frequency=0.1)
SW2(duration=5)
print(f'SW1 == SW2 test: {SW1 == SW2}')

try: # do try-catch so that script doesnt end early at expected error
    SG3 = SineWaveFunction(amplitude=2.0, frequency=0.1)
    SG3(duration=3)
    print(SG1 == SG3)
except ValueError as e:
    print(f"caught ValueError for SineWave: {e}")
try: # do try-catch so that script doesnt end early at expected error
    SW3 = SquareWaveFunction(amplitude=3.0, frequency=0.1)
    SW3(duration=3)
    print(SW1 == SW3)
except ValueError as e:
    print(f"caught ValueError for SquareWave: {e}")
```

Firstly, to implement signal comparison between two instances of the same class, we need to implement the `__eq__` function, which is called when `==` is done. Since both are children of the `SignalProcessor` class, it is simpler to have the parent implement the function, with it being inherited by the children, especially since they both work in the same manner, taking in a duration and creating a list with the length of the duration. When `__eq__` is called, it takes in a parameter of both class instances, which is `self` and `other`. We can access their data parameters and then iterate over them in a loop, checking if they are equal or within the threshold outlined in the assignment. `count += (i == j) or (abs(i-j) < 0.01)` is a simple way to implement the counter in a compact manner, as in

Python, a true statement gives 1, with a false one being 0. Therefore, if one of two conditions is true, and therefore considered equal, it gives 1 and appropriately increments the counter. In the `for` loop, `zip` is used as it allows for easy iteration over two items that are the same type and length and access the same indices simultaneously. Looking at the screenshot of the code output below, given the two `SineWave` data, we should expect an output of 2 given that two entries are considered equal, which we see to be true. Then, for the `SquareWave` data, since all entries are 3.0, we should see a comparison output that is the length of the duration, i.e., 5. We see that holds as well.

Next, we go back to the `__eq__` function, and note the first two lines of the function. If the length of the two data signals are not of the same length, we cannot do a meaningful comparison. Therefore, when their lengths are mismatched (checked via first line of function), we throw an error, with the message: "Two signals are not equal in length!" Since this raises an error, it will by default end the script, but with a try-except block, we can catch the error, and print out the exact error without pre-maturely ending the script. Testing this out with both `SW` and `SG`, we see that the error is successfully caught with the error message printed.

```
#========== Q6 ==========#
Sinewave data is: [0.0, 1.1755705045849463, 1.902113032590307, 1.9021130325903073, 1.1755705045849465]
Sinewave data is: [0.0, 1.618033988749895, 1.9021130325903073, 0.618033988749895, -1.175570504584946]
SG1 == SG2 test: 2
SquareWave data is: [3.0, 3.0, 3.0, 3.0, 3.0]
SquareWave data is: [3.0, 3.0, 3.0, 3.0, 3.0]
SW1 == SW2 test: 5
Sinewave data is: [0.0, 1.1755705045849463, 1.902113032590307]
caught ValueError for SineWave: Two signals are not equal in length!
SquareWave data is: [3.0, 3.0, 3.0]
caught ValueError for SquareWave: Two signals are not equal in length!
```

1.7 Question 7

```
110  #q7
111  print("#========== Q7 ==========#") #need this, looks to disorganized otherwise
112  #1. sine and square waves of varying amplitudes
113  amp_list = [1.0, 2.5, 5.0]
114  print(f"Used amplitudes: {amp_list}")
115  for amp in amp_list:
116      SG = SineWaveFunction(amplitude=amp, frequency=0.1)
117      SW = SquareWaveFunction(amplitude=amp, frequency=0.1)
118      SG(duration=10)
119      SW(duration=10)
120
121  #2. sine and square waves of varying frequencies
122  frq_list = [0.1, 0.25, 0.5]
123  print(f"\nUsed frequencies: {frq_list}")
124  for frq in frq_list:
125      SG = SineWaveFunction(amplitude=2.0, frequency=frq)
126      SW = SquareWaveFunction(amplitude=2.0, frequency=frq)
127      SG(duration=10)
128      SW(duration=10)
```

The question asks for testing the implementation with varying amplitudes and frequencies. Varying amplitudes of 1.0, 2.5, and 5.0 are explored, with varying

frequencies of 0.1, 0.25, and 0.5. The chosen duration is 10, as a larger number results in excessively cluttered print outputs.

In the print output below we can note two things. Firstly, as amplitude increases, so does the peaks and bottoms of the waves. For example, with SineWave at an amplitude of 2.5, we note a peak and bottom of 2.37 / -2.37. At amplitude of 5.0, it becomes 4.75 / -4.75. The same occurs with the SquareWave where it evolves from 2.5 / -2.5 to 5.0 / -5.0. This indicates that the peaks and bottoms of the waves scale linearly with the increase (or decrease) of the amplitude. Secondly, with frequency, it seems directly tied to the rate of oscillation. When frequency is increased, the rate of oscillation is likewise more frequent. While seen in both Square and Sine waves, it is more easily noticeable in SquareWaves. For example, while a frequency of 0.1 only causes it to go from 2 to -2 once, a frequency of 0.5 exhibits this behavior multiple times.

```
#========== Q7 ==========#
Used amplitudes: [1.0, 2.5, 5.0]
Sinewave data is: [0.0, 0.5877852522924731, 0.9510565162951535, 0.9510565162951536, 0.5877852522924732,
1.2246467991473532e-16, -0.587785252292473, -0.9510565162951535, -0.9510565162951536, -0.5877852522924734]
SquareWave data is: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0, -1.0, -1.0, -1.0]
Sinewave data is: [0.0, 1.469463130731183, 2.3776412907378837, 2.377641290737884, 1.4694631307311832,
3.061616997868383e-16, -1.4694631307311825, -2.3776412907378837, -2.377641290737884, -1.4694631307311834]
SquareWave data is: [2.5, 2.5, 2.5, 2.5, 2.5, 2.5, -2.5, -2.5, -2.5, -2.5]
Sinewave data is: [0.0, 2.938926261462366, 4.755282581475767, 4.755282581475768, 2.9389262614623664,
6.123233995736766e-16, -2.938926261462365, -4.755282581475767, -4.755282581475768, -2.938926261462367]
SquareWave data is: [5.0, 5.0, 5.0, 5.0, 5.0, 5.0, -5.0, -5.0, -5.0, -5.0]

Used frequencies: [0.1, 0.25, 0.5]
Sinewave data is: [0.0, 1.1755705045849463, 1.902113032590307, 1.9021130325903073, 1.1755705045849465,
2.4492935982947064e-16, -1.175570504584946, -1.902113032590307, -1.9021130325903073, -1.1755705045849467]
SquareWave data is: [2.0, 2.0, 2.0, 2.0, 2.0, 2.0, -2.0, -2.0, -2.0, -2.0]
Sinewave data is: [0.0, 2.0, 2.4492935982947064e-16, -2.0, -4.898587196589413e-16, 2.0,
7.347880794884119e-16, -2.0, -9.797174393178826e-16, 2.0]
SquareWave data is: [2.0, 2.0, 2.0, -2.0, -2.0, 2.0, 2.0, -2.0, -2.0, 2.0]
Sinewave data is: [0.0, 2.4492935982947064e-16, -4.898587196589413e-16, 7.347880794884119e-16,
-9.797174393178826e-16, 1.2246467991473533e-15, -1.4695761589768238e-15, 1.7145055188062944e-15,
-1.959434878635765e-15, 2.204364238465236e-15]
SquareWave data is: [2.0, 2.0, -2.0, 2.0, -2.0, 2.0, -2.0, 2.0, -2.0, 2.0]
```

2 Bonus

2.1 Question 1

```python
#-------------------------#
#-------------------------#
print("#========== EC ==========#") #need this, looks to disorganized otherwise
#extra 1
class CompositeSignalFunction(SineWaveFunction, SquareWaveFunction):
    def __init__(self, inputs):
        self.sineWave = inputs[0]
        self.squareWave = inputs[1]
    def __call__(self, duration):
        self.data = [sine + square for sine, square in
                     zip(self.sineWave(duration), self.squareWave(duration))]
        return self.data

SG = SineWaveFunction(amplitude=1.0, frequency=0.1)
SW = SquareWaveFunction(amplitude=0.5, frequency=0.05)
CSG = CompositeSignalFunction(inputs=[SG, SW])
CSG(duration=10)
print(f'SineWave Data: {SG.data}')
print(f'SquareWave Data: {SW.data}')
print(f'Composite Data: {CSG.data}')
```

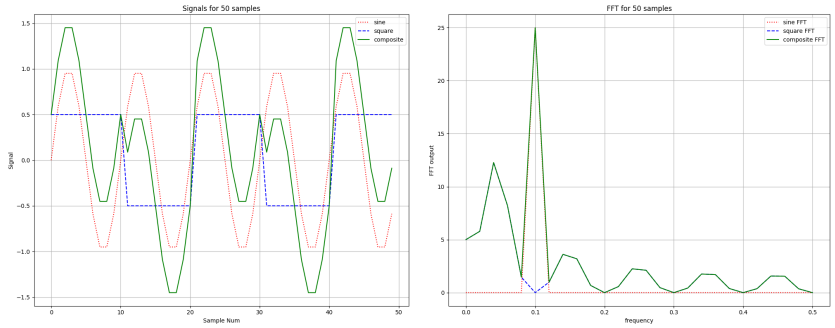
```
#========== EC ==========#
Sinewave data is: [0.0, 0.5877852522924731, 0.9510565162951535, 0.9510565162951536, 0.5877852522924732,
1.2246467991473532e-16, -0.587785252292473, -0.9510565162951535, -0.9510565162951536, -0.5877852522924734]
SquareWave data is: [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5]
SineWave Data: [0.0, 0.5877852522924731, 0.9510565162951535, 0.9510565162951536, 0.5877852522924732,
1.2246467991473532e-16, -0.587785252292473, -0.9510565162951535, -0.9510565162951536, -0.5877852522924734]
SquareWave Data: [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5]
Composite Data: [0.5, 1.0877852522924731, 1.4510565162951536, 1.4510565162951536, 1.0877852522924734,
0.5000000000000001, -0.08778525229247303, -0.45105651629515353, -0.45105651629515364, -0.08778525229247336]
```

For the composite signal function, we need to create a new class that combines their data. For initialization, we are required by the handout to take in a list of a SineWave and SquareWave instance. Since they are elements of a list, we can take the first and second elements of the list (index 0 and 1) and save them during initialization, as we do during `__init__`, which will save these instances of SG and SW for later use. Then, when we call on `CSG` with a given duration, we also input that duration into the saved instances of the SquareWave and SineWave (`self.sineWave(duration)` and `self.squareWave(duration)`) and iterate over their returned data. Similar to a previous question, we use a zip over both to access the appropriate item in the list for both SG and SW simultaneously, which then we add together (`sine + square`). Then we return the data. Printing `CSG.data` indeed shows the implementation works properly, where each entry of CSG's data is the sum of the corresponding entries from SW and SG (e.g., the second entry of the composite data is 1.08, resulting from 0.5 + 0.58).

2.2 Question 2

```
150  #extra 2
151  import numpy as np
152  import matplotlib.pyplot as plt
153  SG_50 = SineWaveFunction(amplitude=1.0, frequency=0.1)
154  SW_50 = SquareWaveFunction(amplitude=0.5, frequency=0.05)
155  CSG_50 = CompositeSignalFunction(inputs=[SG_50, SW_50])
156  CSG_50(duration=50)
157
158  plt.figure(figsize=(10,8))
159  plt.plot(range(50), SG_50.data, label='sine', color='red', linestyle='dotted')
160  plt.plot(range(50), SW_50.data, label='square', color='blue', linestyle='dashed')
161  plt.plot(range(50), CSG_50.data, label='composite', color='green')
162  plt.title('Signals for 50 samples')
163  plt.grid(True)
164  plt.legend()
165  plt.xlabel('Sample Num')
166  plt.ylabel('Signal')
167  plt.tight_layout()
168  plt.savefig('/home/lee3296/signal_plot.png')
169  plt.close()
170
171  #fft
172  #using documentation from: https://numpy.org/doc/stable/reference/routines.fft.html
173  frequencies = np.fft.rfftfreq(50)
174  plt.figure(figsize=(10,8))
175  plt.plot(frequencies, np.abs(np.fft.rfft(SG_50.data)), label='sine FFT',
176           color='red', linestyle='dotted')
177  plt.plot(frequencies, np.abs(np.fft.rfft(SW_50.data)), label='square FFT', color='blue',
178           linestyle='dashed')
179  plt.plot(frequencies, np.abs(np.fft.rfft(CSG_50.data)), label='composite FFT', color='green')
180  plt.title('FFT for 50 samples')
181  plt.grid(True)
182  plt.legend()
183  plt.xlabel('frequency')
184  plt.ylabel('FFT output')
185  plt.tight_layout()
186  plt.savefig('/home/lee3296/signal_fft.png')
187  plt.close()
```

NOTE: Rest of problem on next page.



For this question, we are required to generate two plots, one for plotting the three types of signals and another using FFT analysis, with a minimum of 50 samples. For the plotting of the three type of signals, we start the code (on previous page) off by initializing all three types of signals with a duration of 50. To plot the data, we use `matplotlib`. `.figure` sets up the figure with a defined size (10 $\times$8 here), and .plot actually draws the lines into the plot. `range(50)` sets up the x-axis by going from 0 to 49, and the data is the y. We also set the color and linestyle different for each type of data, as required for this plot. Other parts of the plot setup (e.g., `legend()`) are self explanatory. From the generated plot above (leftmost figure), we see the expected behavior for all three signals (the red sine curves up and down, the blue square jumps up and down forming a square shape, and the composite looks like a sine with an offset due to the square with some small fluctuations when the sine and square values meet).

Next, when looking at the frequency domain using FFT analysis, we follow a similar pattern to plotting the data. However, to get the needed x and y axis values, we utilize numpy's in-built FFT functionality to plot the frequencies (`np.fft.rfftfreq`) and the y-axis (`np.fft.rfft`). `rfft` plots only positive frequencies compared to `fft`, but this is fine because the data we are dealing with is not complex and only contains real numbers. From the FFT analysis, we see that the sine and square peak at frequencies 0.1 and 0.05 respectively, which are the frequency values we initialized them as. The sine wave seems to stick mostly at 0, while the square wave slowly oscillates down to 0. The composite signal mimics mostly the square wave with the exception of 0.1 frequency, where the sine wave is not at 0. As outlined in a comment in the source code, the documentation used for the numpy functionality was found at the NumPy documentation site for FFT.