NLP_C3_W1_lecture_nb_03_data_generators

December 16, 2023

1 Data generators

In Python, a generator is a function that behaves like an iterator. It will return the next item. Here is a link to review python generators. In many AI applications, it is advantageous to have a data generator to handle loading and transforming data for different applications.

You will now implement a custom data generator, using a common pattern that you will use during all assignments of this course. In the following example, we use a set of samples `a`, to derive a new set of samples, with more elements than the original set.

Note: Pay attention to the use of list `lines_index` and variable `index` to traverse the original list.

[ ]:
```
import random as rnd
import numpy as np

# Example of traversing a list of indexes to create a circular list
a = [1, 2, 3, 4]
b = [0] * 10

a_size = len(a)
b_size = len(b)
lines_index = [*range(a_size)] # is equivalent to [i for i in range(0,a_size)],␣
↪the difference being the advantage of using * to pass values of range␣
↪iterator to list directly
index = 0                      # similar to index in data_generator below
for i in range(b_size):        # `b` is longer than `a` forcing a wrap
    # We wrap by resetting index to 0 so the sequences circle back at the end␣
↪to point to the first index
    if index >= a_size:
        index = 0

    b[i] = a[lines_index[index]]     #  `indexes_list[index]` point to a index␣
↪of a. Store the result in b
    index += 1

print(b)
```

1.1 Shuffling the data order

In the next example, we will do the same as before, but shuffling the order of the elements in the output list. Note that here, our strategy of traversing using `lines_index` and `index` becomes very important, because we can simulate a shuffle in the input data, without doing that in reality.

[ ]:
```
# Example of traversing a list of indexes to create a circular list
a = [1, 2, 3, 4]
b = []

a_size = len(a)
b_size = 10
lines_index = [*range(a_size)]
print("Original order of index:",lines_index)

# if we shuffle the index_list we can change the order of our circular list
# without modifying the order or our original data
rnd.shuffle(lines_index) # Shuffle the order
print("Shuffled order of index:",lines_index)

print("New value order for first batch:",[a[index] for index in lines_index])
batch_counter = 1
index = 0                 # similar to index in data_generator below
for i in range(b_size):   # `b` is longer than `a` forcing a wrap
    # We wrap by resetting index to 0
    if index >= a_size:
        index = 0
        batch_counter += 1
        rnd.shuffle(lines_index) # Re-shuffle the order
        print("\nShuffled Indexes for Batch No.{} :{}".
 ↪format(batch_counter,lines_index))
        print("Values for Batch No.{} :{}".format(batch_counter,[a[index] for␣
 ↪index in lines_index]))

    b.append(a[lines_index[index]])      #  `indexes_list[index]` point to a␣
 ↪index of a. Store the result in b
    index += 1
print()
print("Final value of b:",b)
```

Note: We call an epoch each time that an algorithm passes over all the training examples. Shuffling the examples for each epoch is known to reduce variance, making the models more general and overfit less.

1.1.1 Exercise

Instructions: Implement a data generator function that takes in `batch_size, x, y shuffle` where x could be a large list of samples, and y is a list of the tags associated with those samples. Return a subset of those inputs in a tuple of two arrays `(X,Y)`. Each is an array of dimension (`batch_size`). If `shuffle=True`, the data will be traversed in a random form.

Details:

This code as an outer loop

```
while True:
...
yield((X,Y))
```

Which runs continuously in the fashion of generators, pausing when yielding the next values. We will generate a batch_size output on each pass of this loop.

It has an inner loop that stores in temporal lists (X, Y) the data samples to be included in the next batch.

There are three slightly out of the ordinary features.

1. The first is the use of a list of a predefined size to store the data for each batch. Using a predefined size list reduces the computation time if the elements in the array are of a fixed size, like numbers. If the elements are of different sizes, it is better to use an empty array and append one element at a time during the loop.
2. The second is tracking the current location in the incoming lists of samples. Generators variables hold their values between invocations, so we create an `index` variable, initialize to zero, and increment by one for each sample included in a batch. However, we do not use the `index` to access the positions of the list of sentences directly. Instead, we use it to select one index from a list of indexes. In this way, we can change the order in which we traverse our original list, keeping untouched our original list.
3. The third also relates to wrapping. Because `batch_size` and the length of the input lists are not aligned, gathering a batch_size group of inputs may involve wrapping back to the beginning of the input loop. In our approach, it is just enough to reset the `index` to 0. We can re-shuffle the list of indexes to produce different batches each time.

[ ]:
```
def data_generator(batch_size, data_x, data_y, shuffle=True):
    '''
      Input:
        batch_size - integer describing the batch size
        data_x - list containing samples
        data_y - list containing labels
        shuffle - Shuffle the data order
      Output:
        a tuple containing 2 elements:
        X - list of dim (batch_size) of samples
        Y - list of dim (batch_size) of labels
    '''
```

```
    data_lng = len(data_x) # len(data_x) must be equal to len(data_y)
    index_list = [*range(data_lng)] # Create a list with the ordered indexes of sample data


    # If shuffle is set to true, we traverse the list in a random way
    if shuffle:
        rnd.shuffle(index_list) # Inplace shuffle of the list

    index = 0 # Start with the first element
    # START CODE HERE
    # Fill all the None values with code taking reference of what you learned so far
    while True:
        X = None # We can create a list with batch_size elements.
        Y = None # We can create a list with batch_size elements.

        for i in range(batch_size):

            # Wrap the index each time that we reach the end of the list
            if index >= data_lng:
                index = None
                # Shuffle the index_list if shuffle is true
                if shuffle:
                    None # re-shuffle the order

            X[i] = None # We set the corresponding element in x
            Y[i] = None # We set the corresponding element in y
    # END CODE HERE
            index += 1

        yield((X, Y))
```

If your function is correct, all the tests must pass.

[ ]:
```
def test_data_generator():
    x = [1, 2, 3, 4]
    y = [xi ** 2 for xi in x]

    generator = data_generator(3, x, y, shuffle=False)

    assert np.allclose(next(generator), ([1, 2, 3], [1, 4, 9])),  "First batch does not match"
    assert np.allclose(next(generator), ([4, 1, 2], [16, 1, 4])), "Second batch does not match"
```

```
    assert np.allclose(next(generator), ([3, 4, 1], [9, 16, 1])), "Third batch␣
↪does not match"
    assert np.allclose(next(generator), ([2, 3, 4], [4, 9, 16])), "Fourth batch␣
↪does not match"

    print("\033[92mAll tests passed!")

test_data_generator()
```

If you could not solve the exercise, just run the next code to see the answer.

[ ]:
```
import base64

solution =␣
↪"ZGVmIGRhdGFfZ2VuZXJhdG9yKGJhdGNoX3NpemUsIGRhdGFfeCwgZGF0YV95LCBzaHVmZmxlPVRydWUpOgoKICAgIG
# Print the solution to the given assignment
print(base64.b64decode(solution).decode("utf-8"))
```

1.1.2 Hope you enjoyed this tutorial on data generators which will help you with the assignments in this course.