Assignment3

March 9, 2021

0.1 Jeremy Kazimer

5018-1732

Assignment #3

As always, our import statements:

[1]:
```
from numpy import zeros, array, einsum, float32, sum, zeros_like, subtract
from numpy.lib.stride_tricks import as_strided
from matplotlib import pyplot as plt
```

The solution to this PDE

$$\frac{\partial^2 u}{\partial t^2} = c^2 \Delta u$$

otherwise known as the wave equation is given by the following discrete set of equations:

$$u_0(x,y) = C \tag{1}$$
$$u_{n+1}(x,y) = u_n(x,y) + u'_n(x,y)\delta t \tag{2}$$
$$u'_{n+1}(x,y) = u'_n(x,y) + (c^2 \Delta u)\delta t \tag{3}$$
$$\tag{4}$$

Here, since the equation is 2-D dimensional, it can be broken into matrix operations as a numerical solution. As such, the following code are initial conditions taken from the CompPhys directory:

[2]:
```
# Image size
N = 51
n = 50

# u and u'
u_init = zeros([N, N], dtype = float32)
up_init = zeros([N, N], dtype = float32)

# Initial condition on u
```

```
u_init[N//2, N//2] = 10
```

We can also introduce a Laplacian L, which acts as a convolution filter:

[3]:
```
L = array([[0., 1., 0.], [1., -4., 1.], [0., 1., 0.]])
```

Problem #1: **(A.)** *solution by loops*

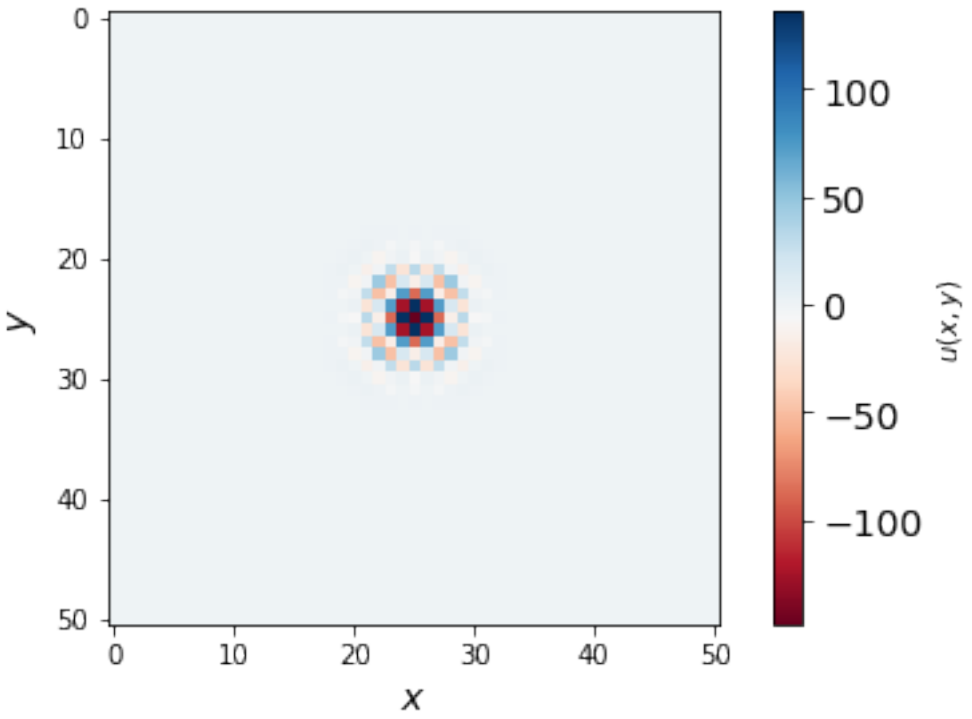
Then, we can create a function out of the numerical solution such that it can be timed. Since this method is iterative, as defined on n, we must therefore pass in some max iteration. Note that this code is also taken from the CompPhys directory, but modified to more closely suit our purpose:

[4]:
```
def convolution_solution(u, up, L, n):

    U = u.copy()
    Up = up.copy()

    Un = zeros_like(U)

    for k in range(0, n):
        for i in range(1, U.shape[0] - 2):
            for j in range(1, U.shape[1] - 2):

                Un[i + 1, j + 1] = sum(L*U[i:i + 3, j:j + 3])/8.

        U  = U + Up
        Up = Up + 1/4 * Un

    return U
```

So, what does this look like for our initial conditions?

[5]:
```
sln = convolution_solution(u_init, up_init, L, n)
```

[6]:
```
fig, ax = plt.subplots(1, 1)

I = ax.imshow(sln, 'RdBu')

ax.set_ylabel('$y$', fontsize = 14)
ax.set_xlabel('$x$', fontsize = 14)

cbar = fig.colorbar(I, label = '$u(x, y)$')
cbar.ax.tick_params(labelsize = 14)

fig.tight_layout();
```



It's great that we can solve this - that is perfectly fine. However, how long does this actually take? Since the initial conditions are precisely the same each time, we can opt to only do different runs, not iterations.

[7]:
```
%timeit -r 10 convolution_solution(u_init, up_init, L, n)
```

```
54.2 s ± 485 ms per loop (mean ± std. dev. of 10 runs, 1 loop each)
```

As to be expected with Python loops. . . but we can do better. . .

(B.) *vectorized solution*

So a few things really must be noted here. Mostly that I didn't entirely come up with the solution myself. Rather, I pieced together different bits and pieces from StackOverflow because I couldn't find a precise implementation other than `scipy`, but that felt very uneducational. As such, the StackOverflow resources are as follows:

- https://machinelearningmastery.com/padding-and-stride-for-convolutional-neural-networks/
- https://stackoverflow.com/questions/48097941/strided-convolution-of-2d-in-numpy/49062204
- https://jessicastringham.net/2018/01/01/einsum/

- https://www.w3schools.com/python/gloss_python_join_tuple.asp
- https://www.tensorflow.org/api_docs/python/tf/einsum
- https://stackoverflow.com/questions/26089893/understanding-numpys-einsum

There is probably a more clean solution, but this is the one I came up with after piecing everything together:

[17]:
```
def vectorized_solution(u, up, L, n):

    # Everything from the old function, including parameters.
    # These won't change because we still need to do the same
    # operations on them.
    U = u.copy()
    Up = up.copy()
    Un = zeros_like(U)

    # This is the new vectorized function.
    def conv_u(U, L):

        # What is 'shape'? 'shape' is a four-element tuple
        # containing the size of U (that is, the initial aray)
        # and L.  This tells us what shapes we're starting from and convolving␣
↪with.
        # We subtract 2 so that the edges are left alone.
        shape = tuple(subtract(U.shape, 2))

        # This appends (3, 3)
        shape += L.shape

        # What is a stride? For a 2-d matrix, it takes the
        # size of the 2nd axis and multiplies by 4, for 4 bits.
        # We need this for the 'as strided' function so that
        # it produces all possible multiplications.
        # We append U.strides to itself because
        # the shapes need to be consistent.
        strides = U.strides + U.strides

        # as_strided takes in U and two shapes such that
        # it produces all possible windows
        # based on the size of shape.  For our
        # data, this'll return (48, 48, 3, 3).
        all_combos = as_strided(U, shape, strides)

        # einsum is the 'tensor contraction over specified indices and outer␣
↪product' (TensorFlow).
        # this one is a bit hard to explain, but it does matrix multiplication␣
↪according to the dimensions
        # you give it.  So, 'xy' correspond to L's shape, 'uvxy' corresponds to␣
↪'all_combos''s shape.
        # So, this is (3, 3) and (48, 48, 3, 3).  Since 'xy' is repeated we're␣
↪telling them that these
```

```
        # dimensions match and that we should do something with that so that it␣
↪matches the dimensions of
        # 'uv'.  I _think_ in this case it's saying multiply along 'xy' of both␣
↪matrices and then sum
        # such that it has the dimensions of uv.

        # This is the same as:
        #(L[np.newaxis, np.newaxis, :,:]*all_combos).sum(axis = 2).sum(axis = 2)
        # But this seemed like a terrible solution so I googled around a bit and
        # found einsum.  It does the same thing but in less code.
        conv = einsum('xy, uvxy -> uv', L, all_combos)

        return conv

    # Then we iterate over the time space.
    for _ in range(n):

        # Same as before, but we exclude the edges.
        # We divide by 8 just as we do in the other
        # solution.

        Un[1:-1, 1:-1] = conv_u(U, L)/8
        U  = U + Up
        Up = Up + 1./4. * Un

    return U
```

And, of course, we can time it with the same conditions…

[10]:
```
%timeit -r 10 vectorized_solution(u_init, up_init, L, n)
```

```
218 ms ± 2.93 ms per loop (mean ± std. dev. of 10 runs, 1 loop each)
```

[19]:
```
print('vectorized code is {:.2f} times faster!'.format(54.2/(218/1000)))
```

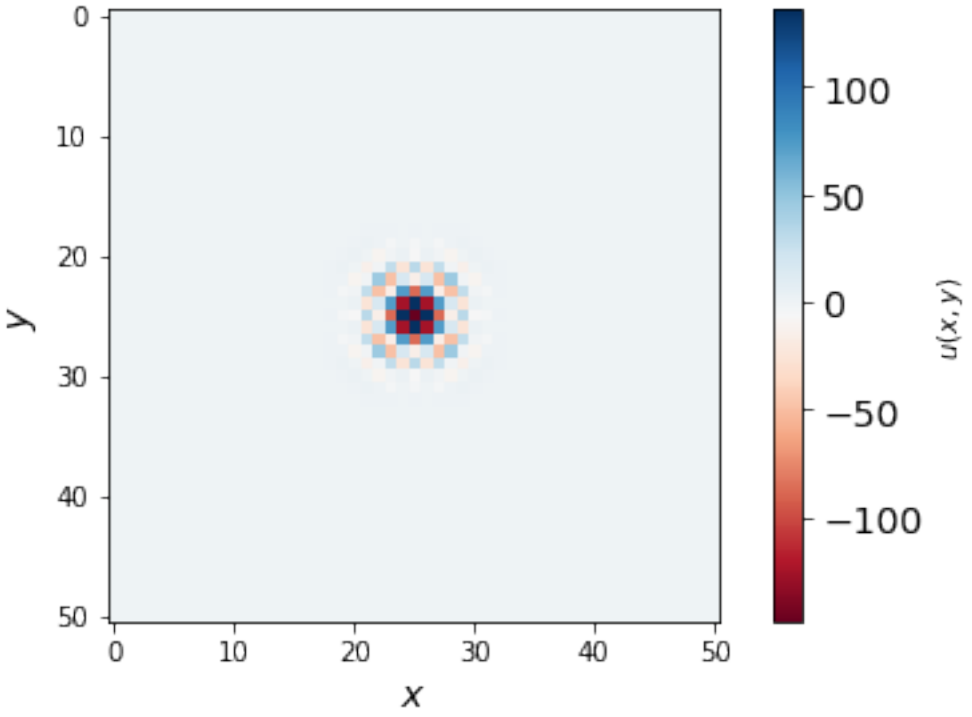
```
vectorized code is 248.62 times faster!
```

Absolutely incredible! We can plot the image as well, to show that it's the same.

[20]:
```
vec_sln = vectorized_solution(u_init, up_init, L, n)
```

[21]:
```
fig, ax = plt.subplots(1, 1)

I = ax.imshow(vec_sln, 'RdBu')

ax.set_ylabel('$y$', fontsize = 14)
ax.set_xlabel('$x$', fontsize = 14)

cbar = fig.colorbar(I, label = '$u(x, y)$')
```

```
cbar.ax.tick_params(labelsize = 14)

fig.tight_layout();
```



I don't see why anyone would make the preference for loops. However, I don't think my code is exactly adjusted such that it can handle any u. If I could make one change that would be it! Otherwise, the speed-up is really cool and not quite what I was anticipating. I thought maybe 20 times faster at most, but this is game-changing for the Pi Zero.

Problem #2: Unfortunately, I don't have any numerical results to show. Really, after seeing the speed-up from Problem 1b, I'm convinced that this is probably the fastest it'll get without coming short of damaging my device. This is not to say that I didn't do any research at all. I did, and a few hours worth.

The main thing I learned is that, at this point, nothing supports `armv6` and `armv6l` for computational purposes. I learned this because I wanted to install a Docker image such that I could try out Tensorflow Lite, but it only supported `armv7` and forward. Regardless, the way that Tensorflow described Lite, and several others did, is that it was a computationally effective and inexpensive alternative to Tensorflow. In my mind, this meant to me that it had all the basic functionality, such as matrix multiplication. But it didn't. We'll get there!

So, I first tried to install it natively. I came into no huge errors, but it did give me warnings that it needed 16 bits (bytes?) to allocate for these functions. The Pi Zero W only has 8. So, I thought maybe that I had the wrong installation. Thus, I tried the `armv6l` specific version (this was for

`armv6` before) and it gave me the exact same warnings. I eventually installed Tensorflow on my Pi Zero by creating a virtual environment and installing it there. I must say that I was not happy when I found out Lite has only about 6 functions, for optimizing models. While there, I also tested run-times for Tensorflow versus Numpy and found that Tensorflow was upwards of 10-15 times slower than Numpy, on all operations.

At this point, my main path was exhausted. I had looked into videocore before and in general I understand what they're doing there from my time in CSE 250 (Data Structures) and CSE 305 (Programming Languages), but the warning that they constantly gave is that incorrectly allocating memory can severely damage or break the device. Although Pi Zero's are inexpensive, this is something I couldn't afford to do time-wise. As such, this option was already exhausted because the risk analysis told me that the risk was too high. With that, different searches kept bringing me back to videocore or telling me to get a different Pi model. So, at this point, it seems that there is no solid hardware acceleration that can be done without the user going in and modifying things themselves.

As a result, I've opted to not attempt hardware acceleration via the only reasonable option, videocore, because even then the developers warn that it's rather unsafe for people inexperienced with memory allocation. Although I had taken those two CSE courses, they're certainly not enough to get me going for this assignment.

However, I know that this wasn't the purpose of the assignment, but I am confident in the speed-up from Problem 1b. It feels like the happy ending to an unhappy situation with hardware acceleration. Even though I am not directly modifying the hardware, this seems like the best route short of doing that. Really, at the very least Problem 2 was great in showing me the true limitations of the Pi Zero. Of course, anyone can reason that space is a huge limiting factor, but slowly learning that nothing is supported, that its speed has a cap, is probably the most education I've had about hardware this entire semester, so far. So, at least I have that... right?

For what it's worth, I've also spoken to friends familiar with this hardware on feedback for this, what I've said above, and they've been stumped by the same things. It seems that the Pi Zero W is meant for very small scale data analysis, despite being marketed as capable of machine learning.

Conclusion: There's not much for me to conclude, here. I wish I could elaborate on the theory of the PDE more, but I've not taken that course nor have I self-taught myself the necessary skills to firmly make any statements. I also wish that I was able to explain my vectorized solution a bit further in detail, but I couldn't find many resources that gave a firm answer themselves.

Other than that, I'm disappointed that I wasn't able to succeed in hardware acceleration, especially since this apparently comes up again later, but I can't always get the answer I want. Overall, a fair assignment with fair expectations. After all, you did warn me that the Pi Zero W was going to be difficult!
