Problem1

April 8, 2021

0.1 Jeremy Kazimer

0.1.1 5018-1732

0.1.2 Assignment #5

Import Statements As always, import statements are necessary:

[1]:
```
import numpy as np
from numpy.random import choice
import matplotlib.pyplot as plt
import tensorflow as tf
```

```
/home/kzmr/anaconda3/lib/python3.7/site-
packages/tensorflow/python/framework/dtypes.py:526: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/home/kzmr/anaconda3/lib/python3.7/site-
packages/tensorflow/python/framework/dtypes.py:527: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/home/kzmr/anaconda3/lib/python3.7/site-
packages/tensorflow/python/framework/dtypes.py:528: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/home/kzmr/anaconda3/lib/python3.7/site-
packages/tensorflow/python/framework/dtypes.py:529: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/home/kzmr/anaconda3/lib/python3.7/site-
packages/tensorflow/python/framework/dtypes.py:530: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
```

```
/home/kzmr/anaconda3/lib/python3.7/site-
packages/tensorflow/python/framework/dtypes.py:535: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])
```

[2]:
```
session = tf.compat.v1.Session()
tf.compat.v1.disable_eager_execution()
tf.global_variables_initializer().run(session = session)
```

1D Simulation In a one-dimensional random walk, the general premise is that the person starts at the origin. Then, if viewed on a Cartesian set of axes, the person is always moving forward one step along the y-axis. However, at each step they can either move to the left, right, or stay in their current location. This is encoded by:

[3]:
```
choices = tf.constant([-1, 0, 1])
```

We define the `choices` as this because we only allow them to move in multiples of 1. Of course, in the real world, a person's stride is variant depending on a variety of factors. However, for the purposes of simplicity, it will be kept this way. A simple 1D simulation can be seen below:

[4]:
```
'''
    dims -> the number of dimensions.
    n_walkers -> the number of iterations.
    n_steps -> the number of steps each walker will take.
'''

dims = 1
n_walkers = 100
n_steps = 500
```

We also have to define a shape for an array so that each step can be recorded:

[5]:
```
steps = tf.random.uniform(minval = -1, maxval = 2, shape = (n_walkers, n_steps,␣
 ↪dims), dtype = tf.int64)
```

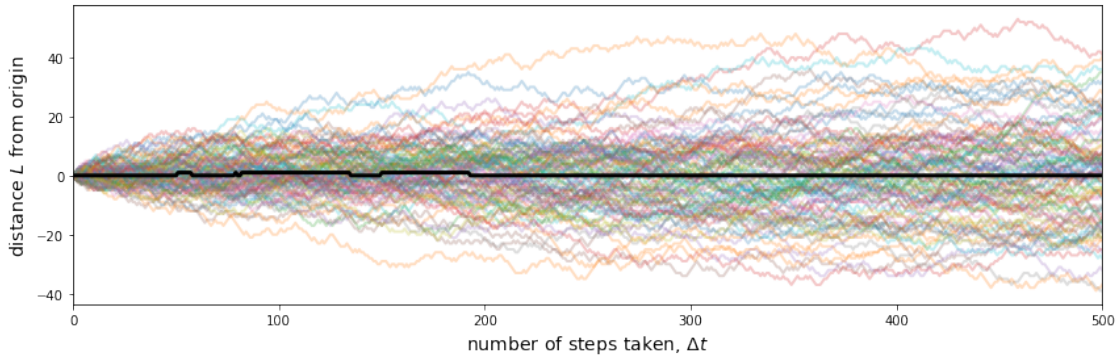
And then we take the cumulative sum, since this will give us the distance from the origin at any step:

[6]:
```
distance_traveled = tf.math.cumsum(steps, axis = 1).eval(session=tf.compat.v1.
 ↪Session())
```

Of course, what is a Jupyter notebook without its plots? Nothing!

[7]:
```
fig, ax = plt.subplots(1, 1, figsize = (12, 4))

L = tf.math.reduce_mean(distance_traveled, axis = 0).eval(session=tf.compat.v1.
 ↪Session())
dt = np.arange(n_steps) + 1
```

```
ax.plot(dt, distance_traveled[:, :, 0].T, lw = 2, alpha = 0.25);
ax.plot(dt, L, lw = 3, c = 'black')

ax.set_xlim(0, n_steps)

ax.set_ylabel('distance $L$ from origin', fontsize = 14)
ax.set_xlabel(r'number of steps taken, $\Delta t$', fontsize = 14)

fig.tight_layout()
```



Note that the faded lines represent each individual walker, whereas the black line represents the average of those walkers.

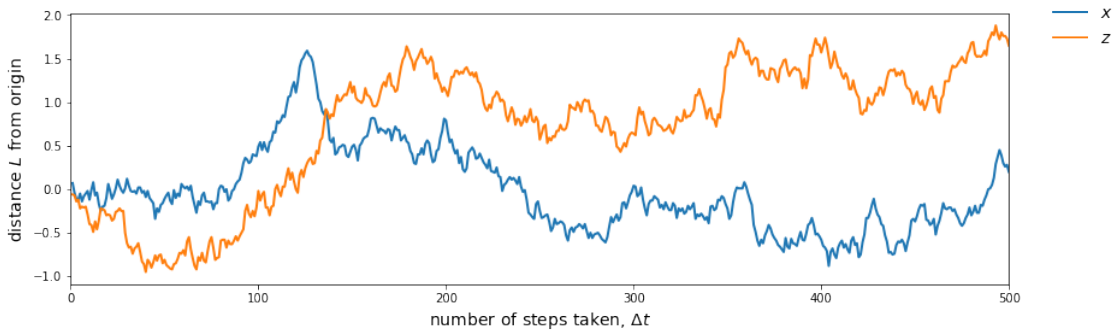
This is effectively a Markov Chain Monte Carlo simulation; basically, a random process - choosing which direction to walk - is done N times to observe some greater phenomena. This is the Monte Carlo portion. The Markov Chain aspect is then that the walker is transitioning from one 'state' to another, where the state is said walker's location. This is a probabilistic process as well. However, each option in our simulation has equal probability.

Note that this can easily be generalized to higher dimensions, which we can do as a function. However, instead of viewing each individual walk, we'll only look at the average from this point forward:

[8]:
```
def random_walk(n_steps, n_walkers, dims):
    steps = tf.random.uniform(minval = -1, maxval = 2, shape = (n_walkers,␣
 ↪n_steps, dims), dtype = tf.int64)
    distance_traveled = tf.math.cumsum(tf.cast(steps, dtype = tf.float64), axis␣
 ↪= 1)
    return distance_traveled
```

2D Simulation Now, suppose the same person from the previous section can now move in another direction. Say, the z axis. The analogy falls apart here, since a person cannot just levitate. Oftentimes, random walks generalize to particle motion for higher motion because of this fallacy. Regardless, we can observe the individual components below:

[9]:
```
fig, ax = plt.subplots(1, 1, figsize = (12, 4))

L_2D = tf.math.reduce_mean(random_walk(n_steps, n_walkers, 2), axis = 0)
dt = np.arange(n_steps) + 1

ax.plot(dt, L_2D.eval(session=tf.compat.v1.Session()) , lw = 2)

ax.set_xlim(0, n_steps)

ax.set_ylabel('distance $L$ from origin', fontsize = 14)
ax.set_xlabel(r'number of steps taken, $\Delta t$', fontsize = 14)

fig.legend((r'$x$', r'$z$'), fontsize = 14, loc = 'lower left', frameon =␣
↪False, bbox_to_anchor = [1.00, 0.8])

fig.tight_layout()
```

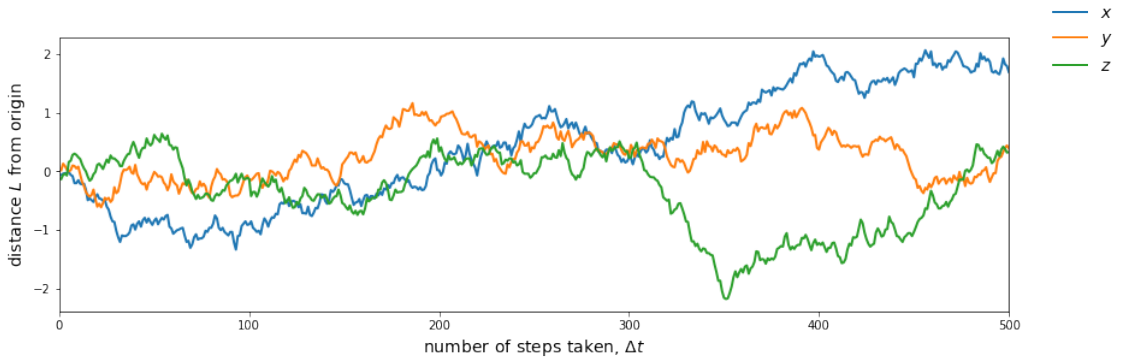


Since their motions are independent of each other, it is anticipated that they'll have similar form. Now, if it were dependent on each other then this function would look different. The same can be said for 3D motion:

3D Simulation 3D motion in space is really analoguous to a particle moving freely, since, once again, humans cannot attain such a motion without extremely high and lethal velocity. As such, we can simulate it normally:

[10]:
```
fig, ax = plt.subplots(1, 1, figsize = (12, 4))

L_3D = tf.reduce_mean(random_walk(n_steps, n_walkers, 3), axis = 0)
dt = np.arange(n_steps) + 1

ax.plot(dt, L_3D.eval(session=tf.compat.v1.Session()), lw = 2)

ax.set_xlim(0, n_steps)
```

```
ax.set_ylabel('distance $L$ from origin', fontsize = 14)
ax.set_xlabel(r'number of steps taken, $\Delta t$', fontsize = 14)

fig.legend((r'$x$', r'$y$', r'$z$'), fontsize = 14, loc = 'lower left', frameon␣
↪= False, bbox_to_anchor = [1.00, 0.8])

fig.tight_layout()
```



Nothing surprising, honestly. We can, however, take this a step further and calculate the diffusion coefficient.

The Diffusion Coefficient The diffusion coefficient, D, is effectively the rate at which the person is traveling. For particle physics, this is the product of particles colliding into each other and impeding movement. Theoretically, the value of this should be

$$\sigma^2 = 2dDt \tag{1}$$

where σ^2 is the variance of the displacement from the origin, squared, d is the dimension, and t is our step size. The formulation for this comes from this paper. However, the root-mean squared variance is expected to be

$$\sqrt{\sigma^2} = \sqrt{d} \tag{2}$$

As such, it would follow that

$$\sigma^2 = d = 2dDt \leftrightarrow D = \frac{1}{2t} \tag{3}$$

So, regardless of dimension, it is anticipated that $D = \frac{1}{2}$. However, we can use experimental observations of σ^2 such that

$$\sigma^2 = \sqrt{E[L^4] - E[L^2]^2} \tag{4}$$

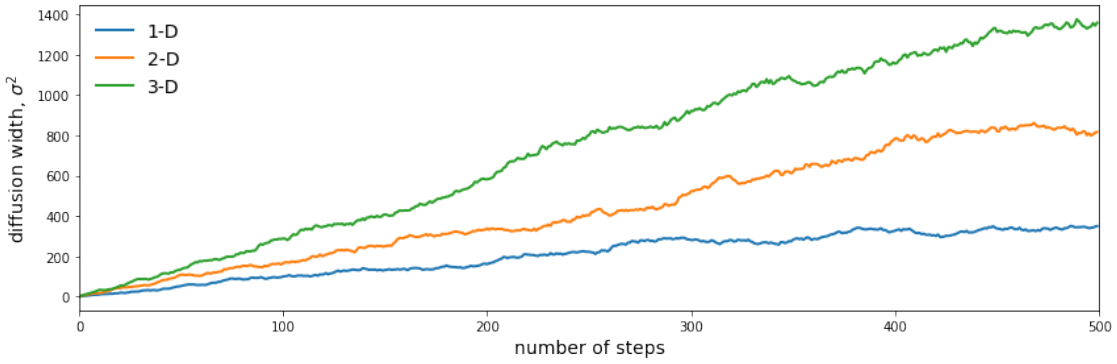
from the course notes. Applying this to all dimensions, first looking at σ^2:

[11]:
```
def sigma(L):

    L2 = tf.reduce_mean(L**2, axis = 0)**2
    L4 = tf.reduce_mean(L**4, axis = 0)

    return tf.math.reduce_sum(tf.math.sqrt(L4 - L2), axis = 1)
```

[12]:
```
L_1 = random_walk(n_steps, n_walkers, 1)
L_2 = random_walk(n_steps, n_walkers, 2)
L_3 = random_walk(n_steps, n_walkers, 3)
```

[13]:
```
s_1 = tf.reshape(sigma(L_1), (n_steps, 1))
s_2 = tf.reshape(sigma(L_2), (n_steps, 1))
s_3 = tf.reshape(sigma(L_3), (n_steps, 1))
```

[14]:
```
fig, ax = plt.subplots(1, 1, figsize = (12, 4))

ax.plot(s_1.eval(session=tf.compat.v1.Session()) , label = '1-D', lw = 2)
ax.plot(s_2.eval(session=tf.compat.v1.Session()) , label = '2-D', lw = 2)
ax.plot(s_3.eval(session=tf.compat.v1.Session()) , label = '3-D', lw = 2)

ax.set_xlim(0, n_steps)

ax.set_ylabel('diffusion width, $\sigma^2$', fontsize = 14)
ax.set_xlabel('number of steps', fontsize = 14)

ax.legend(loc = 'upper left', fontsize = 14, frameon = False)

fig.tight_layout();
```



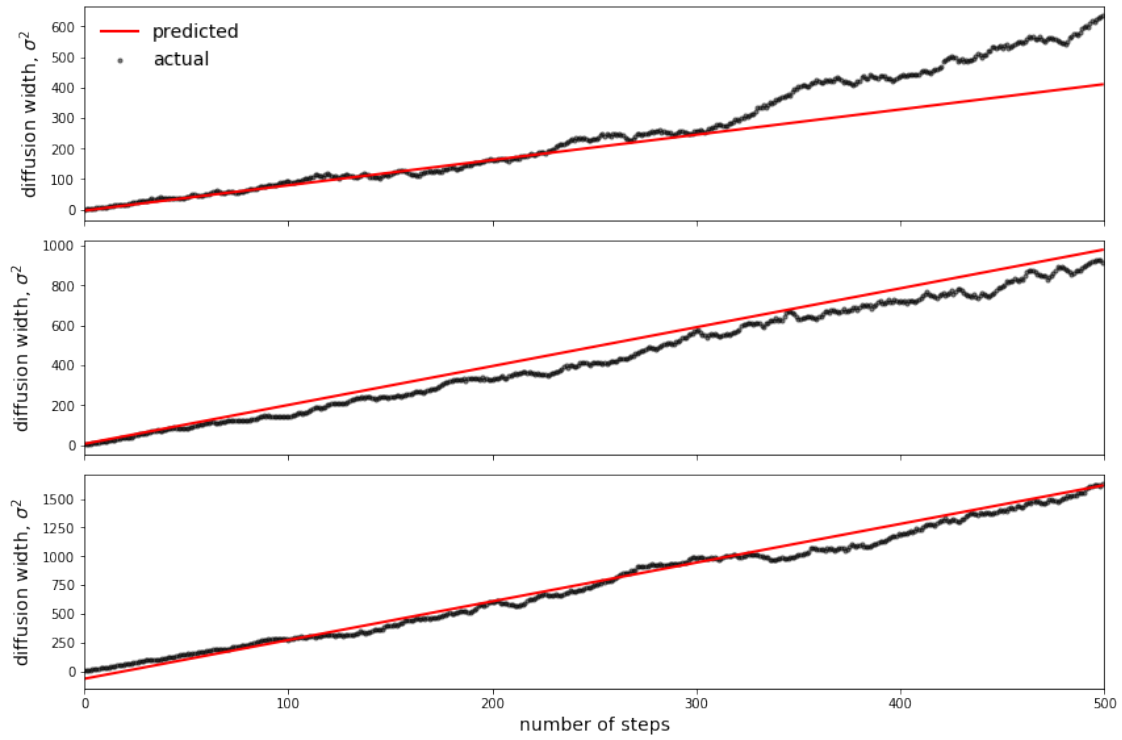
Since this is rather linear, we can also a linear fit on the basis of the diffusion width, squared, and the number of steps. Since `tensorflow` in this version does not offer a `polyfit()` function, we can create our own on the basis of least-squares fitting:

[15]:
```
def tf_least_squares(x, y):
    one_pads = tf.reshape(tf.ones(x.shape[0], dtype = tf.float64), (x.shape[0],␣
 ↪1))

    A = tf.concat((one_pads, tf.reshape(x, (x.shape[0], 1))), axis = 1)
    At = tf.transpose(A)

    lhs = tf.linalg.matmul(At, A)
    rhs = tf.linalg.matmul(At, y)

    return tf.linalg.solve(lhs, rhs)
```

[16]:
```
steps = tf.range(0, n_steps, 1, dtype = tf.float64)

b_1, m_1 = tf_least_squares(steps, s_1).eval(session = session)
b_2, m_2 = tf_least_squares(steps, s_2).eval(session = session)
b_3, m_3 = tf_least_squares(steps, s_3).eval(session = session)
```

[17]:
```
plotting_steps = steps.eval(session = session)
```

[18]:
```
fig, axs = plt.subplots(3, 1, figsize = (12, 8), sharex = True)

axs[0].scatter(plotting_steps, s_1.eval(session=session), s = 10, alpha = 0.5,␣
 ↪label = 'actual', c = 'black')
axs[1].scatter(plotting_steps, s_2.eval(session=session), s = 10, alpha = 0.5,␣
 ↪c = 'black')
axs[2].scatter(plotting_steps, s_3.eval(session=session), s = 10, alpha = 0.5,␣
 ↪c = 'black')

axs[0].plot(plotting_steps, m_1*plotting_steps + b_1, lw = 2, label =␣
 ↪'predicted', c = 'red')
axs[1].plot(plotting_steps, m_2*plotting_steps + b_2, lw = 2, c = 'red')
axs[2].plot(plotting_steps, m_3*plotting_steps + b_3, lw = 2, c = 'red')

axs[0].set_ylabel('diffusion width, $\sigma^2$', fontsize = 14)
axs[1].set_ylabel('diffusion width, $\sigma^2$', fontsize = 14)
axs[2].set_ylabel('diffusion width, $\sigma^2$', fontsize = 14)

axs[0].legend(loc = 'upper left', fontsize = 14, frameon = False)

axs[2].set_xlabel('number of steps', fontsize = 14)
axs[2].set_xlim(0, n_steps)
fig.tight_layout()
```



Note that this fit would be better if the number of walkers were increased. But due to space and time limitations, this is not possible. The predicted diffusion rate is then just the slope of the fit s in place of σ^2 in the earlier equation. This is because our step size is 1:

[19]:
```
D_1 = m_1/2
D_2 = m_2/4
D_3 = m_3/6
```

[20]:
```
print(D_1, D_2, D_3)
```

```
[0.41407725] [0.48773634] [0.56168534]
```

It appears that they're rather close. The main issue being that there's just not enough to smooth out the rough patches. Otherwise, it would ideally converge to $\frac{1}{2}$. Of course, technical limitations will prevent this from ever happening, at least on the Pi Zero W.

Conclusion Overall, the main barrier in converting to `tensorflow` from `numpy` was not the functions for me, but the types. When trying to calculate averages, it kept returning an `int64`. Then, I realized…`tensorflow` doesn't have implicit type conversion! The issue arises because my solution to the lack of a direct equivalent of `np.random.choice` was to generate a random set of `int64`'s from the set $\{-1, 0, 1\}$. This basically does the same thing, although averaging over several different walkers becomes an issue because they should never be `int64`'s, but rather `float64`'s, unless there's only a few walkers and they share a path.

So, `tf.cast` was an easy enough solution. The greater difficulty in this assignment was under-

standing the literature and originally setting this up. Unfortunately, not everything can be done in `tensorflow` due to plotting libraries not being yet compatible with the library itself. However, I got pretty close…one day.

Creating my own `polyfit()` was fun, although this is effectively a repeat of what I did for the final project last semester. Working with `tensorflow` is certainly a change. I'll get used to it, I'm more familiar with `pytorch` (unfortunately).
