dl-projectile-solver

January 9, 2020

0.1 Deep Learning Projectile Motion Solver

This notebook implements an artifical neural network to solve projectile motion problems when only three of the variables x, y, v0x, v0y, vy, and t are known. An solution engine utilizing kinematic equations of motion is used to calculate exact solutions to problems used to both train the neural network and evaluate the accuracy of its results. The domain of the problems is constrained to simplify the coding of the exact solution engine. Specifically, v0x and v0y are always >=0, y is always <=0, and vy is always an unkown.

0.1.1 Import useful packaged including TensorFlow 2.0 and Keras.

The notebook utilizes tensorflow >= 2.0, which now includes keras, a package of high level wrappers designed to make building and training deep learning models easier.

[1]:
```
# import packages
import tensorflow as tf
import tensorflow.keras as keras
import numpy as np
import random
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D

from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.models import Sequential
from tensorflow.keras import optimizers
```

0.1.2 Check to see if a Tensorflow is installed with GPU support and if a GPU is available.

[2]:
```
if not tf.test.is_gpu_available():
    print('No GPU found. Training will be slower.')
else:
    print('Default GPU {} found.'.format(tf.test.gpu_device_name()))
```

```
Default GPU /device:GPU:0 found.
```

0.1.3 Define helper functions.

[3]:
```python
# randomly select a set of three knowns and set all unknowns initially to zero.
def seed_inputs():

    # need at least one known for the x-dimension, and one from the y␣
↪dimension, but three in all.
    # to make imaginary and multiple solutions impossible, voy and v0x are␣
↪always >=0,
    # y is always <= 0, and vy is always unknown.
    variables = ['x', 'y', 'v0x', 'v0y', 'vy', 't']
    knowns = random.sample(['v0x', 'x'],1) + random.sample(['v0y', 'y'],1)
    remaining = set(['x','y','v0x','v0y','t']).difference(set(knowns))
    knowns = knowns + random.sample(remaining,1)
    vals = [round(random.uniform(0,1000),2), round(random.uniform(0,-500),2),
            round(random.uniform(50,100),2), round(random.uniform(50,100),2),
            round(random.uniform(0,-100),2), round(random.uniform(5,20),2)]
    ins = {variables[i]: vals[i] for i in range(len(variables))}

    for key in ins:
        if not key in knowns:
            ins[key] = 0

    return knowns, ins

# engine to solve projectile motion problem using kinematic equations of motion.
# this may not be the most efficient or ellagent implementation but it works.
def solve_projectile(knowns, ins):
    outs = dict.copy(ins)
    g = -9.8


    if 'x' in knowns and 'v0x' in knowns: # find t
        outs['t'] = outs['x']/outs['v0x']
        knowns = knowns + ['t']

    if not 'y' in knowns:
        if 'v0y' in knowns and 'vy' in knowns: # if not T
            outs['t'] = (outs['vy'] - outs['v0y'])/g
        elif 'v0y' in knowns and 't' in knowns: # if not VY
            outs['vy'] = outs['v0y'] + g*outs['t']
        elif 'vy' in knowns and 't' in knowns: # if not VOY
            outs['v0y'] = outs['vy'] - g*outs['t']
        outs['y'] = outs['v0y']*outs['t'] + 0.5*g*outs['t']**2
    elif 'y' in knowns and 't' in knowns:
        outs['v0y'] = (outs['y'] - 0.5*g*outs['t']**2)/outs['t']
        outs['vy'] = outs['v0y'] + g*outs['t']
```

```
    elif 'y' in knowns and 'vy' in knowns:
        outs['v0y'] = np.sqrt(outs['vy']**2 - 2*g*outs['y']) # always positive
        outs['t'] = (outs['vy']-outs['v0y'])/g
    elif 'y' in knowns and 'v0y' in knowns:
        outs['vy'] = -np.sqrt(outs['v0y']**2 + 2*g*outs['y'])
        outs['t'] = (outs['vy']-outs['v0y'])/g
    if 'x' in knowns and not 'v0x' in knowns:
        outs['v0x'] = outs['x']/outs['t']


    if 'v0x' in knowns and not 'x' in knowns:
        outs['x'] = outs['v0x']*outs['t']


    for key in outs:
        outs[key] = round(outs[key],2)

    return outs

# generate N projectile motion problems and their solutions.
def generate_data(N):
    x = np.zeros((N,6))
    y = np.zeros((N,6))
    for ii in range(N):
        knowns, ins = seed_inputs()
        outs = solve_projectile(knowns, ins)
        x[ii,:] = list(ins.values())
        y[ii,:] = list(outs.values())

    return x, y

# function to test the solution engine so individual solutions can be checked␣
↪for correctness.
def test_solution_engine():
    variables = np.array(['x', 'y', 'v0x', 'v0y', 'vy', 't'])
    knowns, ins = seed_inputs()
    outs = solve_projectile(knowns, ins)

    unknowns = list(set(variables).difference(set(knowns)))

    ins = list(ins.values())
    outs = list(outs.values())

    print('knowns:')
    for each in knowns:
        ii = list(np.where(variables==each))[0][0]
        print(each, '=', ins[ii])
```

```
    print('\nsolved uknowns:')
    for each in unknowns:
        ii = list(np.where(variables==each))[0][0]
        print(each, '=', outs[ii])

    return

# function to calculate a projectiles trajectory from v0x, v0y, and x
def calc_trajectory(solution):
    theta = np.arctan(solution['v0y']/solution['v0x'])
    v0 = np.sqrt(solution['v0x']**2 + solution['v0y']**2)
    xrange = np.arange(0,solution['x'],0.1)
    yrange = xrange*np.tan(theta) - 9.8*np.multiply(xrange,xrange)/(2*v0**2*np.
 ↪cos(theta)**2)

    return xrange, yrange
```

0.1.4 Run the following cell to test the solution engine.

[4]:
```
test_solution_engine()
```

```
knowns:
v0x = 99.08
y = -266.76
v0y = 76.62

solved uknowns:
t = 18.57
x = 1839.78
vy = -105.35
```

0.1.5 Generate training, test, and validation data sets.

[5]:
```
x_train, y_train = generate_data(50000)
x_test, y_test = generate_data(5000)
x_validate, y_validate = generate_data(5000)
```

0.1.6 Define the ANN model graph.

[6]:
```
# learning_rate is a hyperparameter
learning_rate = 0.0001
```

```
# four hidden layers - three fully connected and a dropout layer - and an␣
↪output layer
# linear activation is utilized on the final layer to get numerical results␣
↪commensurate with the type of
# problem we are considering.
model = Sequential()
model.add(Dense(128, activation='relu', input_dim=6))
model.add(Dense(256, activation='relu'))
model.add(Dropout(rate=0.05))
model.add(Dense(256, activation='relu'))
model.add(Dense(6, activation='linear'))


# the error function utilized is the sum of the mean squared error.
model.compile(loss='mean_squared_error',
              optimizer=keras.optimizers.Adam(lr=learning_rate),
              metrics=['accuracy'])

class AccuracyHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.acc = []
        self.val_acc = []

    def on_epoch_end(self, batch, logs={}):
        self.acc.append(logs.get('accuracy'))
        self.val_acc.append(logs.get('val_accuracy'))

history = AccuracyHistory()
```

0.1.7 Train the model.

[7]:
```
# hyperparameters
batch_size = 64
epochs = 64

# train the model
model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_validate, y_validate),
          callbacks=[history])
score = model.evaluate(x_test, y_test, verbose=0)
print(score)
```

```
Train on 50000 samples, validate on 5000 samples
```

```
Epoch 1/64
50000/50000 [==============================] - 2s 44us/sample - loss:
226825.0267 - accuracy: 0.9069 - val_loss: 156662.0082 - val_accuracy: 0.9374
Epoch 2/64
50000/50000 [==============================] - 1s 30us/sample - loss:
205561.2255 - accuracy: 0.9245 - val_loss: 154005.6033 - val_accuracy: 0.9298
Epoch 3/64
50000/50000 [==============================] - 2s 31us/sample - loss:
203286.3120 - accuracy: 0.9226 - val_loss: 151467.9689 - val_accuracy: 0.9310
Epoch 4/64
50000/50000 [==============================] - 1s 29us/sample - loss:
201363.5741 - accuracy: 0.9252 - val_loss: 149307.5514 - val_accuracy: 0.9360
Epoch 5/64
50000/50000 [==============================] - 1s 30us/sample - loss:
199268.3278 - accuracy: 0.9322 - val_loss: 146704.5826 - val_accuracy: 0.9442
Epoch 6/64
50000/50000 [==============================] - 1s 30us/sample - loss:
197005.7727 - accuracy: 0.9422 - val_loss: 143848.7816 - val_accuracy: 0.9544
Epoch 7/64
50000/50000 [==============================] - 1s 30us/sample - loss:
194245.8185 - accuracy: 0.9510 - val_loss: 141031.0822 - val_accuracy: 0.9584
Epoch 8/64
50000/50000 [==============================] - 2s 31us/sample - loss:
192099.9537 - accuracy: 0.9558 - val_loss: 137642.2182 - val_accuracy: 0.9634
Epoch 9/64
50000/50000 [==============================] - 2s 31us/sample - loss:
189560.1260 - accuracy: 0.9592 - val_loss: 135169.8088 - val_accuracy: 0.9682
Epoch 10/64
50000/50000 [==============================] - 1s 30us/sample - loss:
186618.8667 - accuracy: 0.9618 - val_loss: 131401.5580 - val_accuracy: 0.9684
Epoch 11/64
50000/50000 [==============================] - 2s 30us/sample - loss:
184006.7029 - accuracy: 0.9632 - val_loss: 127511.9567 - val_accuracy: 0.9688
Epoch 12/64
50000/50000 [==============================] - 2s 30us/sample - loss:
180792.4118 - accuracy: 0.9634 - val_loss: 124000.7154 - val_accuracy: 0.9696
Epoch 13/64
50000/50000 [==============================] - 2s 32us/sample - loss:
177997.3625 - accuracy: 0.9639 - val_loss: 119403.0051 - val_accuracy: 0.9716
Epoch 14/64
50000/50000 [==============================] - 2s 31us/sample - loss:
174703.7560 - accuracy: 0.9647 - val_loss: 114992.7253 - val_accuracy: 0.9696
Epoch 15/64
50000/50000 [==============================] - 1s 30us/sample - loss:
171292.8219 - accuracy: 0.9651 - val_loss: 110787.3508 - val_accuracy: 0.9718
Epoch 16/64
50000/50000 [==============================] - 2s 30us/sample - loss:
167286.9927 - accuracy: 0.9659 - val_loss: 105729.2192 - val_accuracy: 0.9714
```

```
Epoch 17/64
50000/50000 [==============================] - 2s 31us/sample - loss:
163514.1763 - accuracy: 0.9652 - val_loss: 102613.4849 - val_accuracy: 0.9702
Epoch 18/64
50000/50000 [==============================] - 2s 33us/sample - loss:
160216.6553 - accuracy: 0.9642 - val_loss: 97338.6637 - val_accuracy: 0.9710
Epoch 19/64
50000/50000 [==============================] - 2s 31us/sample - loss:
156521.5830 - accuracy: 0.9647 - val_loss: 93666.7120 - val_accuracy: 0.9716
Epoch 20/64
50000/50000 [==============================] - 2s 32us/sample - loss:
154229.6766 - accuracy: 0.9654 - val_loss: 89676.0055 - val_accuracy: 0.9674
Epoch 21/64
50000/50000 [==============================] - 2s 32us/sample - loss:
150622.7727 - accuracy: 0.9646 - val_loss: 89117.3173 - val_accuracy: 0.9728
Epoch 22/64
50000/50000 [==============================] - 2s 31us/sample - loss:
147872.7199 - accuracy: 0.9663 - val_loss: 82350.9531 - val_accuracy: 0.9740
Epoch 23/64
50000/50000 [==============================] - 2s 32us/sample - loss:
144762.6736 - accuracy: 0.9665 - val_loss: 77609.5376 - val_accuracy: 0.9690
Epoch 24/64
50000/50000 [==============================] - 1s 30us/sample - loss:
140882.5599 - accuracy: 0.9658 - val_loss: 75497.1645 - val_accuracy: 0.9726
Epoch 25/64
50000/50000 [==============================] - 1s 30us/sample - loss:
139670.0150 - accuracy: 0.9656 - val_loss: 71643.9692 - val_accuracy: 0.9734
Epoch 26/64
50000/50000 [==============================] - 2s 34us/sample - loss:
134934.0436 - accuracy: 0.9664 - val_loss: 68505.9194 - val_accuracy: 0.9746
Epoch 27/64
50000/50000 [==============================] - 2s 31us/sample - loss:
132604.8318 - accuracy: 0.9667 - val_loss: 66166.1641 - val_accuracy: 0.9774
Epoch 28/64
50000/50000 [==============================] - 1s 30us/sample - loss:
128116.7638 - accuracy: 0.9673 - val_loss: 63429.5566 - val_accuracy: 0.9770
Epoch 29/64
50000/50000 [==============================] - 1s 30us/sample - loss:
127712.3832 - accuracy: 0.9671 - val_loss: 58173.8889 - val_accuracy: 0.9774
Epoch 30/64
50000/50000 [==============================] - 1s 30us/sample - loss:
124037.8837 - accuracy: 0.9672 - val_loss: 61002.8354 - val_accuracy: 0.9716
Epoch 31/64
50000/50000 [==============================] - 2s 31us/sample - loss:
122256.5941 - accuracy: 0.9678 - val_loss: 55248.4380 - val_accuracy: 0.9746
Epoch 32/64
50000/50000 [==============================] - 2s 33us/sample - loss:
116502.9261 - accuracy: 0.9687 - val_loss: 51723.7252 - val_accuracy: 0.9782
```

```
Epoch 33/64
50000/50000 [==============================] - 2s 34us/sample - loss:
113723.5618 - accuracy: 0.9697 - val_loss: 55236.0293 - val_accuracy: 0.9786
Epoch 34/64
50000/50000 [==============================] - 2s 30us/sample - loss:
116418.9519 - accuracy: 0.9699 - val_loss: 43481.1027 - val_accuracy: 0.9814
Epoch 35/64
50000/50000 [==============================] - 2s 31us/sample - loss:
111896.7498 - accuracy: 0.9706 - val_loss: 44488.4195 - val_accuracy: 0.9788
Epoch 36/64
50000/50000 [==============================] - 2s 34us/sample - loss:
110896.6180 - accuracy: 0.9711 - val_loss: 39506.6367 - val_accuracy: 0.9802
Epoch 37/64
50000/50000 [==============================] - 2s 34us/sample - loss:
108678.1261 - accuracy: 0.9704 - val_loss: 37298.3340 - val_accuracy: 0.9746
Epoch 38/64
50000/50000 [==============================] - 2s 33us/sample - loss:
103491.8128 - accuracy: 0.9717 - val_loss: 37589.3822 - val_accuracy: 0.9826
Epoch 39/64
50000/50000 [==============================] - 2s 32us/sample - loss:
103134.0231 - accuracy: 0.9699 - val_loss: 32574.8370 - val_accuracy: 0.9828
Epoch 40/64
50000/50000 [==============================] - 2s 32us/sample - loss:
101741.5785 - accuracy: 0.9709 - val_loss: 33616.2368 - val_accuracy: 0.9826
Epoch 41/64
50000/50000 [==============================] - 2s 32us/sample - loss: 99322.1724
- accuracy: 0.9711 - val_loss: 29588.1351 - val_accuracy: 0.9810
Epoch 42/64
50000/50000 [==============================] - 2s 32us/sample - loss: 97225.4852
- accuracy: 0.9703 - val_loss: 30238.4709 - val_accuracy: 0.9812
Epoch 43/64
50000/50000 [==============================] - 2s 32us/sample - loss: 96541.6528
- accuracy: 0.9702 - val_loss: 28331.8182 - val_accuracy: 0.9812
Epoch 44/64
50000/50000 [==============================] - 2s 32us/sample - loss: 93585.9041
- accuracy: 0.9723 - val_loss: 24742.7860 - val_accuracy: 0.9812
Epoch 45/64
50000/50000 [==============================] - 2s 31us/sample - loss: 91557.1265
- accuracy: 0.9722 - val_loss: 23917.4806 - val_accuracy: 0.9810
Epoch 46/64
50000/50000 [==============================] - 2s 32us/sample - loss: 86673.8677
- accuracy: 0.9705 - val_loss: 24981.9717 - val_accuracy: 0.9794
Epoch 47/64
50000/50000 [==============================] - 2s 31us/sample - loss: 87267.1953
- accuracy: 0.9706 - val_loss: 25024.6665 - val_accuracy: 0.9794
Epoch 48/64
50000/50000 [==============================] - 2s 32us/sample - loss: 84550.4746
- accuracy: 0.9705 - val_loss: 26502.9428 - val_accuracy: 0.9786
```

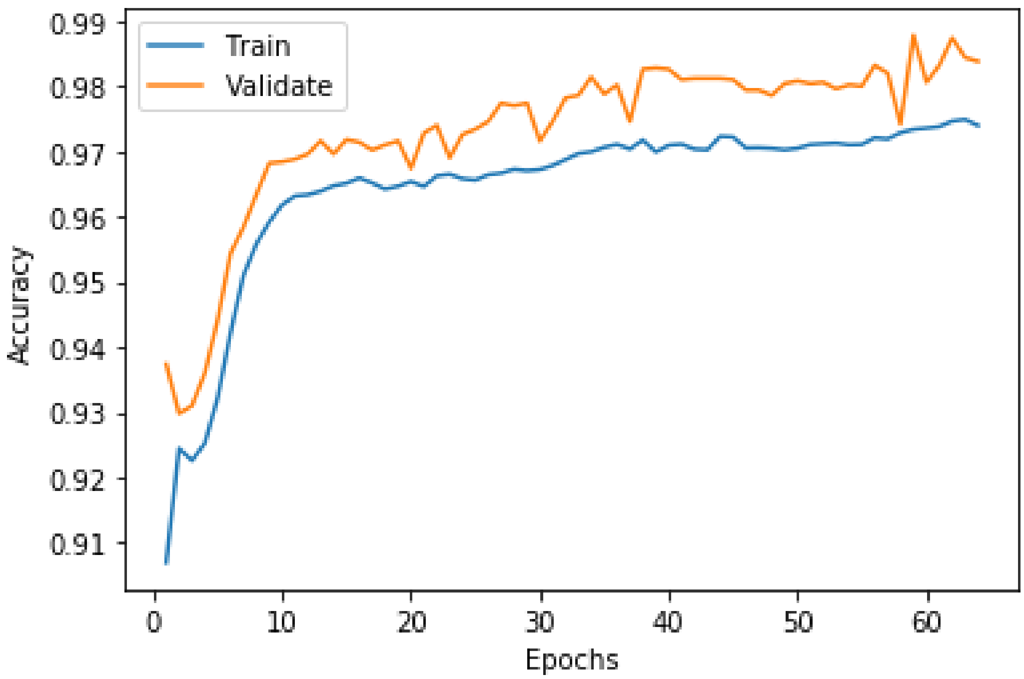
```
Epoch 49/64
50000/50000 [==============================] - 2s 31us/sample - loss: 83723.3058
- accuracy: 0.9702 - val_loss: 18467.7277 - val_accuracy: 0.9804
Epoch 50/64
50000/50000 [==============================] - 2s 32us/sample - loss: 80032.0324
- accuracy: 0.9705 - val_loss: 17431.4114 - val_accuracy: 0.9808
Epoch 51/64
50000/50000 [==============================] - 2s 31us/sample - loss: 81359.5918
- accuracy: 0.9710 - val_loss: 19448.4440 - val_accuracy: 0.9804
Epoch 52/64
50000/50000 [==============================] - 2s 31us/sample - loss: 81454.9648
- accuracy: 0.9712 - val_loss: 18295.1779 - val_accuracy: 0.9806
Epoch 53/64
50000/50000 [==============================] - 2s 31us/sample - loss: 73807.0647
- accuracy: 0.9713 - val_loss: 16917.5558 - val_accuracy: 0.9796
Epoch 54/64
50000/50000 [==============================] - 2s 31us/sample - loss: 81913.1900
- accuracy: 0.9710 - val_loss: 23103.9868 - val_accuracy: 0.9802
Epoch 55/64
50000/50000 [==============================] - 2s 31us/sample - loss: 73979.3202
- accuracy: 0.9711 - val_loss: 17476.6455 - val_accuracy: 0.9800
Epoch 56/64
50000/50000 [==============================] - 2s 31us/sample - loss: 76632.6042
- accuracy: 0.9721 - val_loss: 17569.1226 - val_accuracy: 0.9832
Epoch 57/64
50000/50000 [==============================] - 2s 31us/sample - loss: 72044.8257
- accuracy: 0.9719 - val_loss: 15260.2575 - val_accuracy: 0.9820
Epoch 58/64
50000/50000 [==============================] - 2s 31us/sample - loss: 72098.7199
- accuracy: 0.9729 - val_loss: 14909.3710 - val_accuracy: 0.9742
Epoch 59/64
50000/50000 [==============================] - 2s 31us/sample - loss: 66594.7389
- accuracy: 0.9734 - val_loss: 26212.2037 - val_accuracy: 0.9878
Epoch 60/64
50000/50000 [==============================] - 2s 32us/sample - loss: 69767.9273
- accuracy: 0.9736 - val_loss: 16098.3586 - val_accuracy: 0.9806
Epoch 61/64
50000/50000 [==============================] - 2s 31us/sample - loss: 63826.5568
- accuracy: 0.9738 - val_loss: 16299.9440 - val_accuracy: 0.9832
Epoch 62/64
50000/50000 [==============================] - 2s 31us/sample - loss: 66000.5024
- accuracy: 0.9746 - val_loss: 39307.8232 - val_accuracy: 0.9874
Epoch 63/64
50000/50000 [==============================] - 2s 31us/sample - loss: 67496.6115
- accuracy: 0.9749 - val_loss: 16970.3528 - val_accuracy: 0.9844
Epoch 64/64
50000/50000 [==============================] - 2s 31us/sample - loss: 66953.0000
- accuracy: 0.9740 - val_loss: 26566.4506 - val_accuracy: 0.9838
```

```
[9655.62845546875, 0.984]
```

0.1.8 Graph the accuracy of the model using the training data set versus the validation data set.

[8]:
```
print('Test loss:', score[0])
print('Test accuracy:', score[1])
plt.plot(range(1,epochs+1), history.acc, range(1,epochs+1), history.val_acc)
plt.legend(['Train','Validate'])
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.show()
```

```
Test loss: 9655.62845546875
Test accuracy: 0.984
```

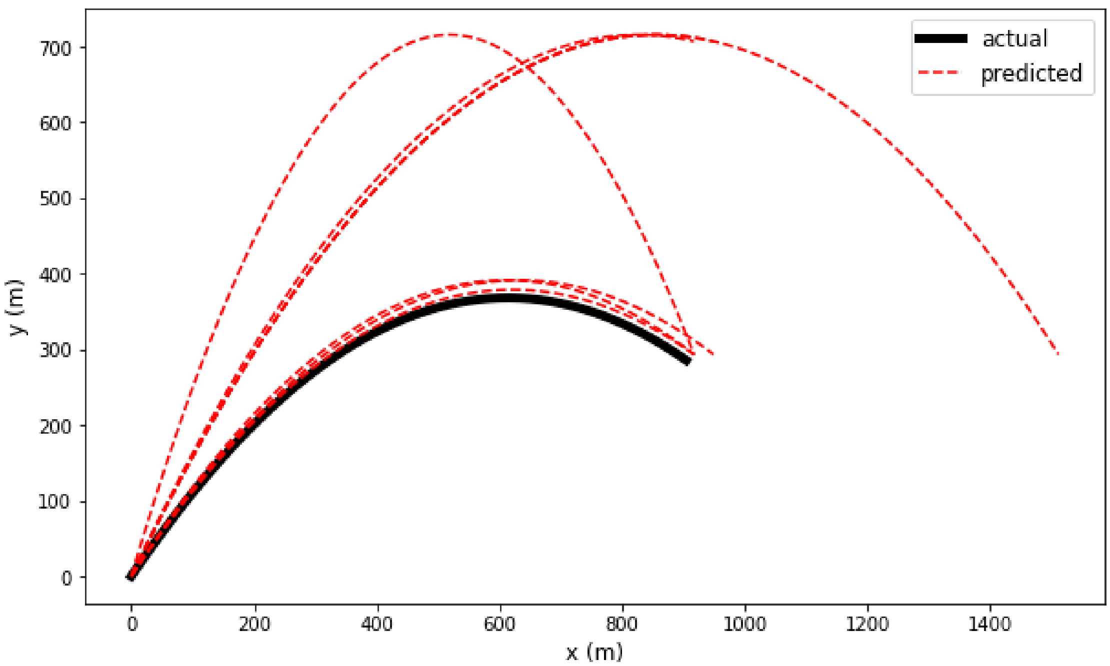


0.1.9 Visualize solutions to the projectile motion problem predicted by the ANN

The solutions to the projectile motion problems predicted by the trained neural network are not exact solutions. In fact, the nature of the model can result in even the knowns fed into the network being adusted. So, each prediction made by the ANN from a set of three knowns acually defines a variety of trajectories, which are visualized here.

[12]:

```
variables = np.array(['x', 'y', 'v0x', 'v0y', 'vy', 't'])
knowns, ins = seed_inputs()
outs = solve_projectile(knowns, ins)
outs_prime = model.predict([list(ins.values())])
outs_prime = {variables[i]: round(outs_prime[0][i],2) for i in␣
↪range(len(variables))} # convert to dict
knowns_perms = [['v0x', 'v0y', 'x'], ['v0x', 'v0y', 'y'], ['v0x', 'v0y', 't'],
                ['v0x', 'y', 'x'], ['v0x', 'y', 't'],
                ['x', 'v0y', 'y'], ['x', 'v0y', 't'],
                ['x', 'y', 't']]
plt.figure(figsize=(10,6))

xr, yr = calc_trajectory(outs)
actual = plt.plot(xr, yr, 'k', linewidth=5)

for each in knowns_perms:
    solution = solve_projectile(each, outs_prime)
    xrange, yrange = calc_trajectory(solution)
    plt.plot(xrange,yrange,'r', linestyle='dashed')

plt.xlabel('x (m)', fontsize=12)
plt.ylabel('y (m)', fontsize=12)


legend_elements = [Line2D([0], [0], color='k', linewidth=5), Line2D([0], [0],␣
↪color='r', linestyle='dashed')]
labels = ['actual','predicted']

plt.legend(legend_elements, labels, fontsize=12)
plt.savefig('projectile.png')
plt.show()
```


actual
predicted
y (m)
x (m)
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200
1400

[ ]: