ReadMe for project 2 in numerical mathematics

30.10.2020

List of files and functionality

Important Code

1. `ann.py`
   - File containing code related to the ANN.
   - Forward sweep, back propagation, function approximator and gradient
2. `support.py`
   - Some support functionality for the ANN
3. `intmethods.py`
   - Numerical integrator code
   - Functions for symplectic Euler and Størmer-Verlet

Testing code

1. `test_training_ann.py`
   - Test of how to train ANN for a 1d function
2. `test_loading_theta.py`
   - Showcasing how to load from a pickled theta and construct functions from it
3. `test_resuming_training.py`
   - Display of how to train ANN, pickle the result and unpickle to train some more
4. `test_train_with_trajectory_data.py`
   - Test which loads data from the .csv files and trains the ANN on them
5. `test_with_trajectory_data.py`
   - Test which loads previously trained thetas and uses them to find T and V
6. `test_actual_hamiltonian.py`
   - Demonstrating how to load previously trained theta, then use the gradients to integrate with Euler or Størmer to get the actual path

Training the ANN and getting the approximator and gradient

Training the ANN is done with the `train_ANN_and_make_model_function` function in `ann.py`. This function invokes the underlying `trainANN` function which performs the actual iteration and updating of theta.

Due to lacking abilities of precognition, the ANN training function returns functions which can be used as the approximator and gradient of the approximator. However, these are relics of simpler times and should not be used.

Since the training function returns theta (or the tuple `(W, b, w, mu)`), training can be resumed and performed on alternative data. The returned functions are to be disregarded. Instead, theta should be inputted to the `make_scaled_modfunc_and_grad` function, which will return virtually the same functions as the one returned from the training function.

The reason for this convoluted way of things is that the `pickle` packages has some issues with storing functions to be loaded at a later time. If one seeks to resume the training or store the ANN for use as a function approximator (to circumvent retraining the ANN each time one needs it), it would be nice to be able to serialize the function, or at least the theta, to save it. Using `make_scaled_modfunc_and_grad` is therefore consideres best practice, as `pickle` cooperates better.

A step-by-step guide to get the approximator and gradient can be the following:

1. Load all data `Y` and `c` for which the ANN should be trained
2. Find the maximal and minimal values of the training data and exact data as to scale the ANN properly.

3. Decide on some reasonable values for
   - `d`, the dimension of the hidden layers
   - `K`, the number of hidden layers
   - `h`, the stepsize between layers
   - `it_max`, when to forcefully abort training
   - `tol`, the targetted error for which the training is considered complete
4. Call `train_ANN_and_make_model_function` with these parameters.
5. Decide what to do with the return values:
   - `scaled_modfunc` can be disregarded
   - `gradient_modfunc` can also be disregarded
   - `Js` is the evolution of the error, which can be plotted to see the decline
   - `(W, b, w, mu)` is theta, this should definitely be kept as it *is* the ANN
6. Pass theta and the min/max values to `make_scaled_modfunc_and_grad` to construct normal functions for the approximator and gradient.

Example of usage, training the network for a 1d function

Running the script `test_training_ann.py` performs this.

```
import numpy as np
import matplotlib.pyplot as plt
import itertools
import ann

def F_exact(y):
    return 1/2 * np.linalg.norm(y)**2

def dF_exact(y):
    return np.array([y[0], y[1]]) * np.linalg.norm(y)

n, y_min, y_max = 5, 0, 1
_Y = np.linspace((y_min, y_min), (y_max, y_max), n).T
x, y = np.linspace(0, 1, n), np.linspace(0, 1, n)
Y = np.array(list(itertools.product(x, y))).T
c = np.array([F_exact(Y[:, i]) for i in range(np.shape(Y)[1])])
c_min, c_max = np.min(c), np.max(c)

d, K, h, tau = 4, 4, 1, .1
it_max, tol = 10000, 1e-4

(_, _, Js, theta) = ann.train_ANN_and_make_model_function(
    Y, c, d, K, h, it_max, tol,
    tau=tau, y_min=y_min, y_max=y_max, c_min=c_min, c_max=c_max)

(F, dF) = ann.make_scaled_modfunc_and_grad(
    theta, y_min, y_max, c_min, c_max, h=h)

t = 101
ys = np.linspace((0, 0), (1, 1), t).T
cs = np.array([F_exact(ys[:, i]) for i in range(t)])
dcs = np.array([dF_exact(ys[:, i]) for i in range(t)])
Fs, dFs = F(ys), dF(np.reshape(ys, (2, t)))

bErr, bFn, bdFn = False, False, True

if bErr:
    plt.plot(Js)
    plt.yscale("log")
    plt.show()
if bFn:
    plt.plot(cs)
```

```
    plt.plot(Fs)
    plt.show()
if bdFn:
    plt.plot(dcs[:, 0], label="dF, 0-th component")
    plt.plot(dcs[:, 1], label="dF, 1-st component")
    plt.plot(dFs[0, :], label="dF~, 0-th component")
    plt.plot(dFs[1, :], label="dF~, 1-st component")
    plt.legend(loc="best")
    plt.show()
```

Example of usage, construction from existing theta

Running the script `test_loading_theta.py` performs this.

```
import numpy as np
import ann
import matplotlib.pyplot as plt
import get_traj_data

all_data = get_traj_data.concatenate(0, 50)

Q_min = np.min(all_data["Q"])
Q_max = np.max(all_data["Q"])
V_min = np.min(all_data["V"])
V_max = np.max(all_data["V"])

import pickle

# Remember to set a path to a valid theta.pickle
with open("path/to/theta.pickle", "rb") as file:
    theta = pickle.load(file)

(V, dV) = ann.make_scaled_modfunc_and_grad(theta, Q_min, Q_max, V_min, V_max)

Qs, Vs = all_data["Q"], all_data["V"]
V_ann = V(Qs)

k = 10000
plt.plot(V_ann[0:k])  # Plotting the first k values of V from ANN
plt.plot(Vs[0:k])  # Plotting the first k values of V from traj_data
plt.show()
```

Continuing the training of an ANN

It is possible to resume the training of an ANN. Doing this requires that one sets the optional `theta` parameter when calling `train_ANN_and_make_model_function`. If training the ANN on different sets of data, it is important to pass correct max/min arguments in order to scale the ANN properly. Loading all datasets to find the global max/min values for the input and exact values and using these values as the scale range ensures that the ANN is scaled properly.

Failing to provide these values forces the ANN to compute the max/min values from the given dataset. If max/min varies between datasets, these values are re-computed per dataset and the ANN has to train to approximate these new initial and terminal values.

This is rather wasteful and gives a false indication of rate of convergence as the approximation to fit the new values happens very quickly.

Example of training resuming and saving of theta

Running the script `test_resume_training.py` performs this. But be warned that with the current settings, the training of the ANN is quite sensistive to the random initialization of theta and the results may vary wildly.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
import ann
import pickle

def Fe(y):
    return 1 - np.cos(y)

def dFe(y):
    return np.sin(y)

n = 11
y_rng = np.pi / 1.5
Y1 = np.reshape(np.linspace(-y_rng, .5 * y_rng, n), (1, n))
Y2 = np.reshape(np.linspace(-.5 * y_rng, y_rng, n), (1, n))

y0 = min(np.min(Y1), np.min(Y2))
yf = max(np.max(Y1), np.max(Y2))
c1, c2 = Fe(Y1), Fe(Y2)
c0 = min(np.min(c1), np.min(c2))
cf = max(np.max(c1), np.max(c2))

d, K, h, tau = 4, 4, 1, .05
it_max, tol = 5000, 1e-4

(_, _, Js1, theta1) = ann.train_ANN_and_make_model_function(
    Y1, c1, d, K, h, it_max, tol, tau=tau,
    y_min=y0, y_max=yf, c_min=c0, c_max=cf)

(F1, dF1) = ann.make_scaled_modfunc_and_grad(
    theta1, y0, yf, c0, cf, h=h)

# b is for binary, which is needed when SERIALIZING
with open("test_theta.pickle", "wb") as file:
    pickle.dump(theta1, file)

# b is for binary, which is needed when SERIALIZING
with open("test_theta.pickle", "rb") as file:
    saved_theta = pickle.load(file)

(_, _, Js2, theta2) = ann.train_ANN_and_make_model_function(
    Y2, c2, d, K, h, it_max, tol, tau=tau,
    y_min=y0, y_max=yf, c_min=c0, c_max=cf, theta=saved_theta)

(F2, dF2) = ann.make_scaled_modfunc_and_grad(
    theta2, y0, yf, c0, cf, h=h)

t = 101
ys = np.reshape(np.linspace(y0, yf, t), (1, t))
cs, dcs = np.reshape(Fe(ys), (1, t)), dFe(ys)
Fs1, Fs2 = F1(ys), F2(ys)
dFs1, dFs2 = dF1(ys), dF2(ys)

bErr, bFn, bdFn = True, True, True

if bErr:
    plt.plot(np.concatenate((Js1, Js2)))
    plt.yscale("log")
    plt.show()
if bFn:
    plt.plot(ys.T, cs.T)
    plt.plot(ys.T, Fs1.T)
    plt.plot(ys.T, Fs2.T)
```

```
    plt.show()
if bdFn:
    plt.plot(ys.T, dcs.T)
    plt.plot(ys.T, dFs1.T)
    plt.plot(ys.T, dFs2.T)
    plt.show()
```

Using the ANN with the given trajectory data

The script `get_traj_data.py` gets the trajectory data from the folder `project_2_trajectories` which holds all the `.csv` files for the trajectories. The trajectories imported with the `concatenate` function from this script give the basis for training the ANN of the actual trajectories and using a previously trained ANN on data within the bounds of these sets.

The ANN as a function approximator works best as an interpolation of the phase space, and the global min/max values discovered from the datasets are the appropriate values to perform Euler's or Størmer-Verlet's methods within.

Example of training the ANN with the trajectory data

Running the script `test_train_with_trajectory_data.py` performs this.

```
import numpy as np
import matplotlib.pyplot as plt
import ann
import get_traj_data
import pickle

first_set = get_traj_data.concatenate(0, 1)

Q_min, Q_max = np.min(first_set["Q"]), np.max(first_set["Q"])
V_min, V_max = np.min(first_set["V"]), np.max(first_set["V"])

P_min, P_max = np.min(first_set["P"]), np.max(first_set["P"])
T_min, T_max = np.min(first_set["T"]), np.max(first_set["T"])

d, K, h, tau = 4, 4, 1, .01
it_max, tol = 1000, 1e-4

(V, dV, JsV, theta_V) = ann.train_ANN_and_make_model_function(
    first_set["Q"], first_set["V"], d, K, h, it_max, tol, tau = tau,
    y_min = Q_min, y_max = Q_max, c_min = V_min, c_max = V_max, log=True)

(T, dT, JsT, theta_T) = ann.train_ANN_and_make_model_function(
    first_set["P"], first_set["T"], d, K, h, it_max, tol, tau = tau,
    y_min = P_min, y_max = P_max, c_min = T_min, c_max = T_max, log=True)

# theta_V and theta_T can now be pickled
```

Example of using an already trained ANN to get T and V

Running the script `test_with_trajectory_data.py` performs this.

```
import numpy as np
import ann
import matplotlib.pyplot as plt

import get_traj_data

all_data = get_traj_data.concatenate(0, 50)

Q_min, Q_max = np.min(all_data["Q"]), np.max(all_data["Q"])
V_min, V_max = np.min(all_data["V"]), np.max(all_data["V"])
```

```
P_min, P_max = np.min(all_data["P"]), np.max(all_data["P"])
T_min, T_max = np.min(all_data["T"]), np.max(all_data["T"])

import pickle

with open("test_pickles/theta_T.pickle", "rb") as file:
    theta_T = pickle.load(file)

with open("test_pickles/theta_V.pickle", "rb") as file:
    theta_V = pickle.load(file)

(T, dT) = ann.make_scaled_modfunc_and_grad(theta_T, P_min, P_max, T_min, T_max)
(V, dV) = ann.make_scaled_modfunc_and_grad(theta_V, Q_min, Q_max, V_min, V_max)

Qs, Vs = all_data["Q"], all_data["V"]
Ps, Ts = all_data["P"], all_data["T"]

k = 10000

V_ann = V(Qs)[0:k]
T_ann = T(Ps)[0:k]
H_ann = V_ann + T_ann
Hs = (Vs + Ts)[0:k]

plt.plot(H_ann)
plt.plot(Hs)
plt.show()
```

Using Euler and Størmer-Verlet with the gradients from the ANN

To use Euler or Størmer-Verlet, one must import `intmethods.py`.

The vectors given as the initial position and momentum of the particle must be submitted as a n-vector with a defined second axis size. This means it must have the numpy size `v.shape = (n, 1)` which can be ensured by running the function `np.reshape(v, (n, 1))`.

The return from the `intMeth` function is a tuple of vector-collections. One for momentum, one for position. These numpy arrays have the size `(n, it_max)`, where `n` is the dimension of the submitted vector.

`intMeth` is a general function which will perform its stepping using the provided function `func`. This can either be `symEuler`, `stVerlet` or any other function designed to work with the machinery of `intMeth`.

Example of using the integrator methods

Running the script `test_actual_hamiltonian.py` performs this.

```
import numpy as np
import get_traj_data as gtd
import ann
import pickle
import matplotlib.pyplot as plt
import time
import intmethods as im

if __name__ == "__main__":
    data = gtd.generate_data(44)

    (qmin, qmax, vmin, vmax, pmin, pmax, tmin, tmax) = gtd.get_data_bounds()

    with open("thetas/V_4x5.pickle", "rb") as file:
        th_V = pickle.load(file)
```

```
with open("thetas/T_4x5.pickle", "rb") as file:
    th_T = pickle.load(file)

(T, dT) = ann.make_scaled_modfunc_and_grad(th_T, pmin, pmax, tmin, tmax)
(V, dV) = ann.make_scaled_modfunc_and_grad(th_V, qmin, qmax, vmin, vmax)

trueV, trueT = data["V"], data["T"]
trueH = trueV + trueT

Qs, Ps = data["Q"], data["P"]
annV, annT = V(Qs), T(Ps)
annH = annV + annT

euler, strVer = im.symEuler, im.stVerlet

k, l, off = len(Qs.T), 1, 0
its = l * k - off
dt = 20 / its
p0 = np.reshape(data["P"][:, off], (3, 1))
q0 = np.reshape(data["Q"][:, off], (3, 1))

bEuler = True
bStrVer = False

if bEuler:
    eulPs, eulQs = im.intMeth(p0, q0, dT, dV, its, euler, dt)
    eulerV, eulerT = V(eulQs), T(eulPs)
    eulerH = eulerV + eulerT
    eulQNorm = np.array([np.linalg.norm(y) for y in eulQs.T])

if bStrVer:
    strPs, strQs = im.intMeth(p0, q0, dT, dV, its, strVer, dt)
    strVerV, strVerT = V(strQs), T(strPs)
    strH = strVerV + strVerT
    strQNorm = np.array([np.linalg.norm(y) for y in strQs.T])

trueQNorm = np.array([np.linalg.norm(y) for y in Qs.T])

t0, tf = np.min(data["t"]), np.max(data["t"])
trueTime = np.linspace(t0, tf, k)
intTime = np.linspace(t0, tf, l * k + 1)[off:]

bPlotHamilton = True
bPlotPath = True

if bPlotHamilton:
    plt.plot(trueTime, trueH[:k], label="True Hamiltonian")
    plt.plot(trueTime, annH[:k], label="Hamiltonian from ANN")
    if bEuler:
        plt.plot(intTime, eulerH, label="Euler Hamiltonian")
    if bStrVer:
        plt.plot(intTime, strH, label="Størmer-Verlet Hamiltonian")
    plt.legend(loc="best")
    plt.show()
if bPlotPath:
    plt.plot(trueTime, trueQNorm, label="Norm of Q")
    if bEuler:
        plt.plot(intTime, eulQNorm, label="Norm of Euler's Q")
    if bStrVer:
        plt.plot(intTime, strQNorm, label="Norm of Størmer-Verlet's Q")
    plt.legend(loc="best")
```

```
plt.show()
```