Jypyter (Python3) template for training neural network

This jupyter notebook provides an example workflow for training a neural network to predict forces on a particle in an optical trap. To use this file, you will need to modify the file for your specific problem/data, see comments bellow. The output will be a trained network, saved in a `.h5` file which can be loaded back into python or into another language (such as Matlab).

This template is based on the paper *Machine learning enables fast statistically significant simulations of optically trapped microparticles* (Lenton et al., 2019). This file is released under the GPL-v3 license. If you find this simulation useful, please cite it.

Load the data

You will need a data set which sufficiently covers the parameter space and provides estimates for the force at each position in the parameter space. For example, this data could include particle position, refractive index and orientation and estimate the force/torque for choices of input parameters. The data set could be experimental data or simulation data.

Depending on how your data is saved the following procedure will vary. The two snippets bellow show loading data saved in a Matlab `.mat` file and loading data saved in a `.csv` file.

In [ ]:

```
## Load from .mat file

from scipy.io import loadmat
import numpy as np

# Load from mat file
# you may need to tarnspose depending on how the data is stored
positions = np.transpose(loadmat('input.mat')['positions']) # [Nx3]
radius = np.transpose(loadmat('input.mat')['radius']) # [Nx1]
height = np.transpose(loadmat('input.mat')['height']) # [Nx1]
forces = np.transpose(loadmat('input.mat')['forces']) # [Nx3]
torque = np.transpose(loadmat('input.mat')['torque']) # [Nx3]

# Join input and outputs into single array and store number of features
feature_number = 5
data = np.concatenate((positions, radius, height, forces, torques), axis=1)
```

In [ ]:

```
## Load from .csv file

import csv
import numpy as np

positions = np.empty([0, 3])
forces = np.empty([0, 3])

with open('input.csv') as fp:
    csvfile = csv.reader(fp, delimiter=',')
    for line in csvfile:
        positions.append([[line[0], line[1], line[2]], axis=0)
        forces.append([[line[3], line[4], line[5]]], axis=0)

# Join input and outputs into single array and store number of features
feature_number = 3
data = np.concatenate((positions, forces), axis=1)
```

Split validation and training data and shuffle

We randomly split the data into validation and training data. We use 10% of the data set for validation and reserve the rest for training.

In [ ]:

```
total_points = data.shape[0]
number_val_samples = round(0.1*total_points)
number_train_samples = total_points - number_val_samples

np.random.shuffle(data)

train_data = data[:number_train_samples, :feature_number]
train_targets = data[:number_train_samples, feature_number:]

val_data = data[number_train_samples:number_train_samples + number_val_samples, :featur
e_number]
val_targets = data[number_train_samples:number_train_samples + number_val_samples, feat
ure_number:]
```

Setup keras model

For most tasks we found that a dense neural network with only a few layers was sufficient to accurately predict the forces on the particle. The following shows how we could create a network with 3 hidden layers, 1 layer for the inputs and 1 layer for the outputs. For training, we use the Adam (https://keras.io/optimizers/#adam) optimiser and we use mean squared error for the loss function. We store mean average error and mean average percentage error.

In [ ]:

```
from keras import models
from keras import layers

sz = 256;  # Number of nodes per hidden layer

model = models.Sequential()
model.add(layers.Dense(sz, activation='relu', input_shape=(train_data.shape[1],)))
model.add(layers.Dense(sz, activation='relu'))
model.add(layers.Dense(sz, activation='relu'))
model.add(layers.Dense(train_targets.shape[1]))

model.compile(optimizer='adam', loss='mse', metrics=['mae', 'mape'])

model.summary()
```

Train the model

We train the model using batches of increasing size. The batch size determines how many trial points are evaluated before updating the network weights. For each batch size, we pass over the entire data set `num_epochs` times, storing the error after each epoch.

In [ ]:

```
mae = []
mape = []
val_mae = []
val_mape = []

batches = [32, 128, 1024, 4096, 16384]
num_epochs = 100;   # Fixed number for each batch

for epochs, batch_size in zip([num_epochs]*len(batches), batches):
    print(">>> ")
    print(">>> ", batch_size, " <<<")
    print(">>> ")

    history = model.fit(train_data,
                        train_targets,
                        epochs=epochs,
                        batch_size=batch_size,
                        shuffle=True,
                        validation_data=(val_data, val_targets))

    mae.extend(history.history['mean_absolute_error'])
    mape.extend(history.history['mean_absolute_percentage_error'])
    val_mae.extend(history.history['val_mean_absolute_error'])
    val_mape.extend(history.history['val_mean_absolute_percentage_error'])
```

Save the result

The output files can be loaded back into python or Matlab for later use. We could also visualise the error inside this script, for example, see `example-3dof-dataset` .

In [ ]:

```
save_file_name = "output.h5"
model.save(save_file_name)

import pickle

save_file_name_pkl = "results.pkl"
with open(save_file_name_pkl, 'wb') as f:
    pickle.dump([mae, mape, val_mae, val_mape], f)
```