DHOscillator_analysis

March 12, 2024

[1]:
```
import os
import tempfile

from math import sqrt
import numpy as np
import scipy as sp
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
%matplotlib widget
from itables import init_notebook_mode
init_notebook_mode(all_interactive=True)

from sklearn.model_selection import train_test_split
import torch
from ray import train, tune
```

```
<IPython.core.display.HTML object>
```

[2]:
```
import time
```

1 Dumped Harmonic Oscillator, compare model

In this notebook we will compare the performance of the varius models on the dumped harmonic oscillator dataset.

1.1 Load data

[3]:
```
# import data
# data are generated by "src/DHOscillator_data_gen.py"
data = np.load('../data/DHOscillator_data.npy')
X = data[:,0]
Y = data[:,1:]
```

[4]:
```
def data_loader(X, Y, batch_size):
    """
    Function to load data and divide it in batches
    input: X, Y, batch_size
```

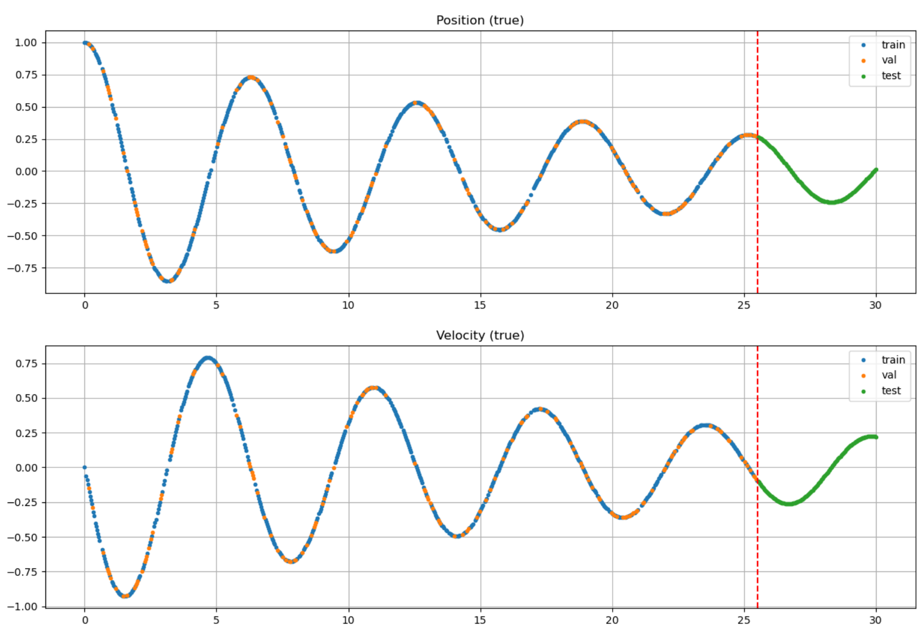
```python
    output: train_X_batches, train_Y_batches, val_X, val_Y, test_X, test_Y
    """

    # divide in train, validation and test
    train_frac = 0.7
    val_frac = 0.15
    test_frac = 0.15

    train_val_X = X[:int((train_frac+val_frac)*len(X))]
    train_val_Y = Y[:int((train_frac+val_frac)*len(X)), :]
    train_X, val_X, train_Y, val_Y = train_test_split(
        train_val_X,
        train_val_Y,
        test_size=val_frac/(train_frac+val_frac),
        random_state=42
        )

    test_X = X[int((train_frac+val_frac)*len(X)):]
    test_Y = Y[int((train_frac+val_frac)*len(X)):, :]

    # convert to torch tensor
    train_X = torch.tensor(train_X, dtype=torch.float32).view(-1, 1)
    train_Y = torch.tensor(train_Y, dtype=torch.float32)
    val_X = torch.tensor(val_X, dtype=torch.float32).view(-1, 1)
    val_Y = torch.tensor(val_Y, dtype=torch.float32)
    test_X = torch.tensor(test_X, dtype=torch.float32).view(-1, 1)
    test_Y = torch.tensor(test_Y, dtype=torch.float32)

    # divide in batches train
    train_X_batches = torch.split(train_X, batch_size)
    train_Y_batches = torch.split(train_Y, batch_size)


    return train_X_batches, train_Y_batches, val_X, val_Y, test_X, test_Y
```

[5]:
```python
# use the data loader to get the data, in this example we use only one batch
train_X_batches, train_Y_batches, val_X, val_Y, test_X, test_Y = data_loader(X,␣
 ↪Y, 595)
```

[6]:
```python
# plot the position
plt.figure(figsize=(15, 10))
plt.subplot(2, 1, 1)
plt.plot(train_X_batches[0].detach().numpy(), train_Y_batches[0][:, 0].detach().
 ↪numpy(), '.', label='train')
plt.plot(val_X.detach().numpy(), val_Y[:, 0].detach().numpy(), '.', label='val')
plt.plot(test_X.detach().numpy(), test_Y[:, 0].detach().numpy(), '.',␣
 ↪label='test')
```

```
plt.grid()
plt.title('Position (true)')
plt.axvline(x=30*0.85, color='r', linestyle='--')
plt.legend()

# plot the velocity
plt.subplot(2, 1, 2)
plt.plot(train_X_batches[0].detach().numpy(), train_Y_batches[0][:, 1].detach().
 ↪numpy(), '.', label='train')
plt.plot(val_X.detach().numpy(), val_Y[:, 1].detach().numpy(), '.', label='val')
plt.plot(test_X.detach().numpy(), test_Y[:, 1].detach().numpy(), '.',␣
 ↪label='test')
plt.grid()
plt.title('Velocity (true)')
plt.axvline(x=30*0.85, color='r', linestyle='--')
plt.legend()
```

[6]: <matplotlib.legend.Legend at 0x7fab2873a430>



1.2 Define some analysis functions

[7]:
```python
def plot_results(model, train_X_batches, train_Y_batches, val_X, val_Y, test_X,␣
↪test_Y, save=False, name='model'):
    """
    Function to plot the results of the net for the
    position and velocity of the DHOscillator

    input: model, train_X_batches, train_Y_batches, val_X, val_Y, test_X, test_Y
    output: plot of the results
    """

    # get predictions
    Y_pred_train = model(train_X_batches[0])
    Y_pred_val = model(val_X)
    Y_pred_test = model(test_X)

    # plot the position, and subplot the residue
    plt.figure(figsize=(18, 13))
    plt.subplot(2, 2, 1)

    marker='.'
    markersize=2

    plt.plot(train_X_batches[0].detach().numpy(), train_Y_batches[0][:, 0].
↪detach().numpy(), marker, label='train true', markersize=markersize)
    plt.plot(test_X.detach().numpy(), test_Y[:, 0].detach().numpy(), marker,␣
↪label='test true', markersize=markersize)
    plt.plot(train_X_batches[0].detach().numpy(), Y_pred_train[:, 0].detach().
↪numpy(), marker, label='train pred', markersize=markersize)
    plt.plot(test_X.detach().numpy(), Y_pred_test[:, 0].detach().numpy(),␣
↪marker, label='test pred', markersize=markersize)
    plt.grid()

    plt.title('Position (true and %s prediction)' % name)
    plt.axvline(x=30*0.85, color='r', linestyle='--')
    plt.legend()

    plt.subplot(2, 2, 3)
    plt.plot(train_X_batches[0].detach().numpy(), train_Y_batches[0][:, 0].
↪detach().numpy()-Y_pred_train[:, 0].detach().numpy(), marker, label='train',␣
↪markersize=markersize)
    plt.plot(test_X.detach().numpy(), test_Y[:, 0].detach().
↪numpy()-Y_pred_test[:, 0].detach().numpy(), marker, label='test',␣
↪markersize=markersize)
    plt.grid()
    plt.ylabel('residue (true - pred)')
```

```
    plt.xlabel('time')
    plt.legend()
    plt.axvline(x=30*0.85, color='r', linestyle='--')


    # new figure for the velocity
    plt.subplot(2, 2, 2)

    plt.plot(train_X_batches[0].detach().numpy(), train_Y_batches[0][:, 1].
 ↪detach().numpy(), marker, label='train true', markersize=markersize)
    plt.plot(test_X.detach().numpy(), test_Y[:, 1].detach().numpy(), marker,␣
 ↪label='test true', markersize=markersize)
    plt.plot(train_X_batches[0].detach().numpy(), Y_pred_train[:, 1].detach().
 ↪numpy(), marker, label='train pred', markersize=markersize)
    plt.plot(test_X.detach().numpy(), Y_pred_test[:, 1].detach().numpy(),␣
 ↪marker, label='test pred', markersize=markersize)
    plt.grid()

    plt.title('Velocity (true and %s prediction)' % name)
    plt.axvline(x=30*0.85, color='r', linestyle='--')
    plt.legend()

    plt.subplot(2, 2, 4)
    plt.plot(train_X_batches[0].detach().numpy(), train_Y_batches[0][:, 1].
 ↪detach().numpy()-Y_pred_train[:, 1].detach().numpy(), marker, label='train',␣
 ↪markersize=markersize)
    plt.plot(test_X.detach().numpy(), test_Y[:, 1].detach().
 ↪numpy()-Y_pred_test[:, 1].detach().numpy(), marker, label='test',␣
 ↪markersize=markersize)
    plt.grid()
    plt.ylabel('residue (true - pred)')
    plt.xlabel('time')
    plt.legend()
    plt.axvline(x=30*0.85, color='r', linestyle='--')

    if save:
        # save the figure
        plt.savefig('../plot/DHOscillator_%s_results.png' % name)
```

[8]:
```
def get_losses(model, train_X_batches, train_Y_batches, val_X, val_Y, test_X,␣
 ↪test_Y):
    """
    Function to get the losses of the model for the train, validation and test␣
 ↪set
    input: model, train_X_batches, train_Y_batches, val_X, val_Y, test_X, test_Y
    output: train_loss, val_loss, test_loss
```

```
    """

    # get predictions
    Y_pred_train = model(train_X_batches[0])
    Y_pred_val = model(val_X)
    Y_pred_test = model(test_X)

    # get losses, to numpy
    train_loss = torch.mean((Y_pred_train - train_Y_batches[0])**2)
    val_loss = torch.mean((Y_pred_val - val_Y)**2)
    test_loss = torch.mean((Y_pred_test - test_Y)**2)

    # print losses
    print('Train loss:', train_loss.item())
    print('Validation loss:', val_loss.item())
    print('Test loss:', test_loss.item())

    return train_loss.item(), val_loss.item(), test_loss.item()
```

[9]:
```
# def a function that get the time for one prediction sampling 10 time the test␣
↪set, not with cuda
def get_pred_time(model, test_X, n_samples=1000):
    """
    Function to get the time for one prediction
    input: model, test_X, n_samples
    output: time
    """

    # get the time for one prediction
    times = []
    for i in range(n_samples):
        start = time.time()
        Y_pred_test = model(test_X)
        end = time.time()
        times.append((end - start)/len(test_X))

    time_pred = np.mean(times)
    time_pred_std = np.std(times)

    print('Time for one prediction:', time_pred, '+/-', time_pred_std)

    return time_pred, time_pred_std
```

[10]:
```
# Model class
class FFNN(torch.nn.Module):
    def __init__(self, n_layers, n_neurons):
```

```
        super(FFNN, self).__init__()
        layers = []
        for i in range(n_layers):
            if i == 0:
                layers.append(torch.nn.Linear(1, n_neurons))
            else:
                layers.append(torch.nn.Linear(n_neurons, n_neurons))
            layers.append(torch.nn.Tanh())
        layers.append(torch.nn.Linear(n_neurons, 2))
        self.model = torch.nn.Sequential(*layers)
    def forward(self, x):
        return self.model(x)
```

[11]:
```
def objective(config):
    net = FFNN(config["n_layers"], config["n_neurons"])

    device = "cpu"

    criterion = torch.nn.MSELoss()
    optimizer = torch.optim.Adam(net.parameters(), lr=config["lr"])
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
        optimizer,
        'min',
        factor=config["factor"],
        patience=config["patience"]
    )

    train_X_batches, train_Y_batches, val_X, val_Y, test_X, test_Y =␣
↪data_loader(data_X, data_Y, config["batch_size"], config["grid_num"])

    for epoch in range(50000):
        for i, (X, Y) in enumerate(zip(train_X_batches, train_Y_batches)):
            optimizer.zero_grad()
            X.requires_grad = True
            Y_pred = net(X)

            # get the derivatives
            dx_dt = torch.autograd.grad(Y_pred[:,0], X, grad_outputs=torch.
↪ones_like(Y_pred[:,0]), create_graph=True)[0]
            dv_dt = torch.autograd.grad(Y_pred[:,1], X, grad_outputs=torch.
↪ones_like(Y_pred[:,1]), create_graph=True)[0]

            # loss_ode and loss_ic
            loss_ode = torch.mean((dx_dt[:,0] - Y_pred[:,1])**2 + (dv_dt[:,0] +␣
↪0.1*Y_pred[:,1] + Y_pred[:,0])**2)
            loss_ic = ((Y_pred[0,0] - 1)**2 + (Y_pred[0,1] - 0)**2)
```

```
            loss = config["lambda"]*loss_ode + loss_ic

            loss.backward()
            optimizer.step()
            scheduler.step(loss)

        val_loss = criterion(net(val_X), val_Y).item()

        report(metrics={"loss": val_loss})

        if epoch % 100 == 0:
            torch.save(net.state_dict(), "./model.pth")
```

[ ]:
```
def analytical_solution(t):
    """
    Function to get the analytical solution of the DHOscillator
    with fixed initial conditions Y0 = [1, 0]
    input: t
    output: Y
    """

    # define system parameters
    m = 1.0
    k = 1.0
    c = 0.1

    Omega = sqrt(k/m - (c/(2*m))**2)
    gamma = c/(2*m)
    A = 1

    return A*np.exp(-gamma*t)*np.cos(Omega*t), -A*np.exp(-gamma*t)*(gamma*np.
 ↪cos(Omega*t) + Omega*np.sin(Omega*t))
```

[ ]:
```
def dumped_spring(t, Y):
    """
    This function calculates the derivative of the state vector Y at time t
    for a spring-mass-damper system.
    t (float): time
    Y (ndarray): state vector [position, velocity]

    Returns:
    dXdt (list): derivative of state vector
    """

    # define system parameters
    m = 1.0
    k = 1.0
```

```
    c = 0.1

    return [Y[1], -k/m*Y[0] - c/m*Y[1]]
```

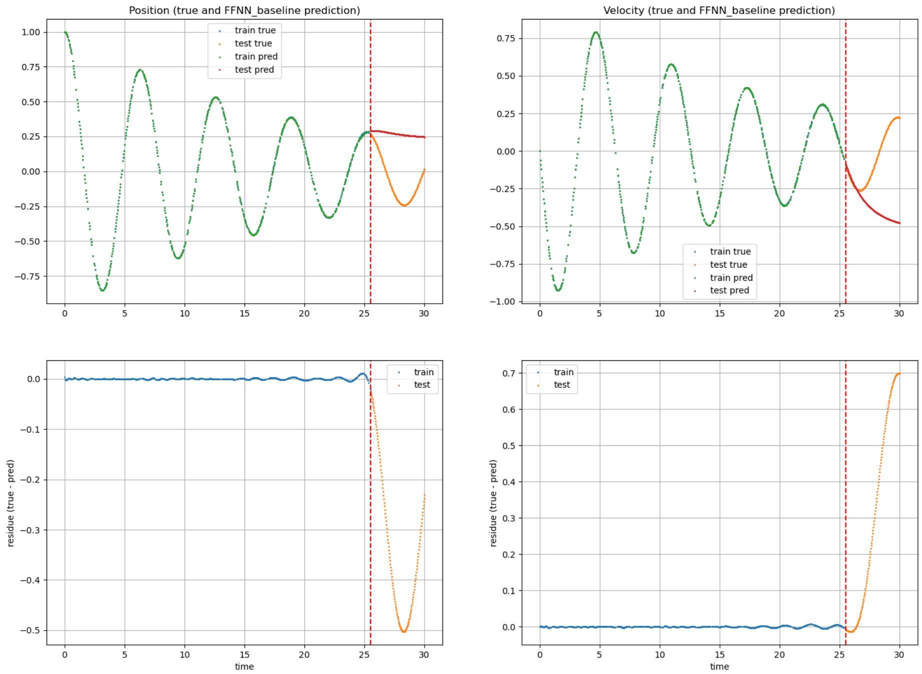
[ ]:
```
def RK5(f, Y0, t_span, dt):
    """
    Function to solve a ODE system using the RK5 method,
    based on solve_ivp but with fixed step size dt
    input: f, Y0, t_span, dt
    output: sol (solve_ivp object)
    """

    # parameters to fix the step size
    e_tol = 1000000
    atol = e_tol
    rtol = e_tol

    max_step = dt
    min_step = dt

    return solve_ivp(f, t_span, Y0, method='RK45', atol=atol, rtol=rtol,␣
 ↪max_step=max_step, min_step=min_step)
```

1.3 Baseline model

Baseline model FFNN and PINN have the same architecture and where trained for the same amount of time and data.

[12]:
```
# load the models
FFNN_baseline = torch.load('../models/DHO_FFNN_baseline.pt')
PINN_baseline = torch.load('../models/DHO_PINN_baseline.pt')
```

[13]:
```
# plot results
plot_results(FFNN_baseline, train_X_batches, train_Y_batches, val_X, val_Y,␣
 ↪test_X, test_Y, save=False, name='FFNN_baseline')
```



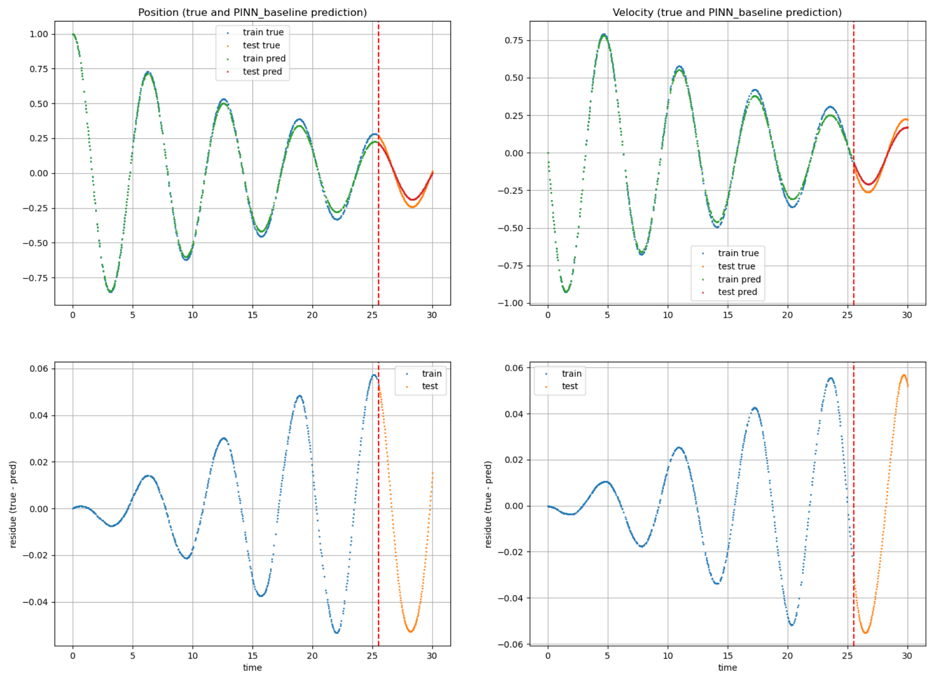
[43]:
```
# get losses
FFNN_baseline_train_loss, FFNN_baseline_val_loss, FFNN_baseline_test_loss =␣
↪get_losses(FFNN_baseline, train_X_batches, train_Y_batches, val_X, val_Y,␣
↪test_X, test_Y)
```

```
Train loss: 5.001572390028741e-06
Validation loss: 6.745639893779298e-06
Test loss: 0.1473110467195511
```

[42]:
```
# get time
FFNN_baseline_time, FFNN_baseline_time_std = get_pred_time(FFNN_baseline,␣
↪test_X, n_samples=100)
```

```
Time for one prediction: 1.7065525054931643e-06 +/- 1.2451593007279568e-06
```

[16]:
```
# plot results PINN
plot_results(PINN_baseline, train_X_batches, train_Y_batches, val_X, val_Y,␣
↪test_X, test_Y, save=False, name='PINN_baseline')
```



[17]:
```
# get losses
PINN_baseline_train_loss, PINN_baseline_val_loss, PINN_baseline_test_loss =␣
↪get_losses(PINN_baseline, train_X_batches, train_Y_batches, val_X, val_Y,␣
↪test_X, test_Y)
```

```
Train loss: 0.0006238690693862736
Validation loss: 0.0006585018127225339
Test loss: 0.0015676728216931224
```

[18]:
```
# get time
PINN_baseline_time, PINN_baseline_time_std = get_pred_time(PINN_baseline,␣
↪test_X, n_samples=100)
```

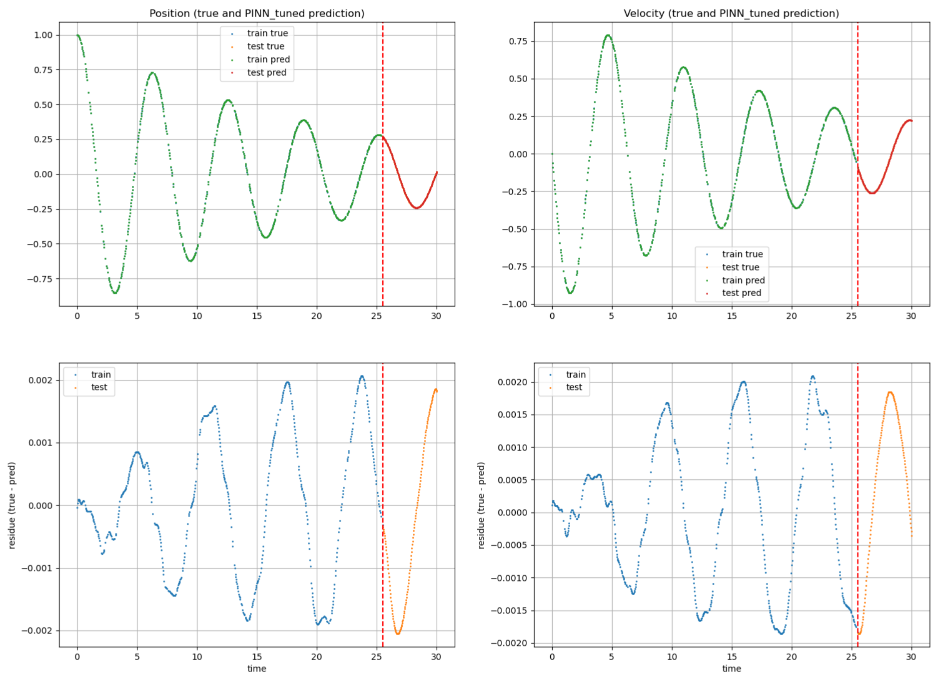
```
Time for one prediction: 1.1006037394205728e-06 +/- 2.93063350201597e-07
```

1.4 PINN Tuned model

[19]:
```
# import DHO_PINN_tuned
DHO_PINN_tuned = torch.load('../models/DHO_PINN_tuned.pt')
```

[20]:
```
# plot results
```

```
plot_results(DHO_PINN_tuned, train_X_batches, train_Y_batches, val_X, val_Y,
    test_X, test_Y, save=True, name='PINN_tuned')
```



[21]:
```
# get losses
DHO_PINN_tuned_train_loss, DHO_PINN_tuned_val_loss, DHO_PINN_tuned_test_loss =
    get_losses(DHO_PINN_tuned, train_X_batches, train_Y_batches, val_X, val_Y,
    test_X, test_Y)
```

```
Train loss: 1.2679423662120826e-06
Validation loss: 1.239517246176547e-06
Test loss: 1.7491795460955473e-06
```

[22]:
```
# time
DHO_PINN_tuned_time, DHO_PINN_tuned_time_std = get_pred_time(DHO_PINN_tuned,
    test_X, n_samples=100)
```

```
Time for one prediction: 1.8994967142740881e-06 +/- 7.45313810576931e-07
```

1.4.1 Dataframe of configurations hyperparameters and results

[23]:
```python
restored_tuner = tune.Tuner.restore('/home/luigi/Documents/PHYSICS/ML/Project1/
 ↪tune/DHO_PINN_tuning', objective)
restored_results = restored_tuner.get_results()

restored_df = restored_results.get_dataframe()
restored_df
```

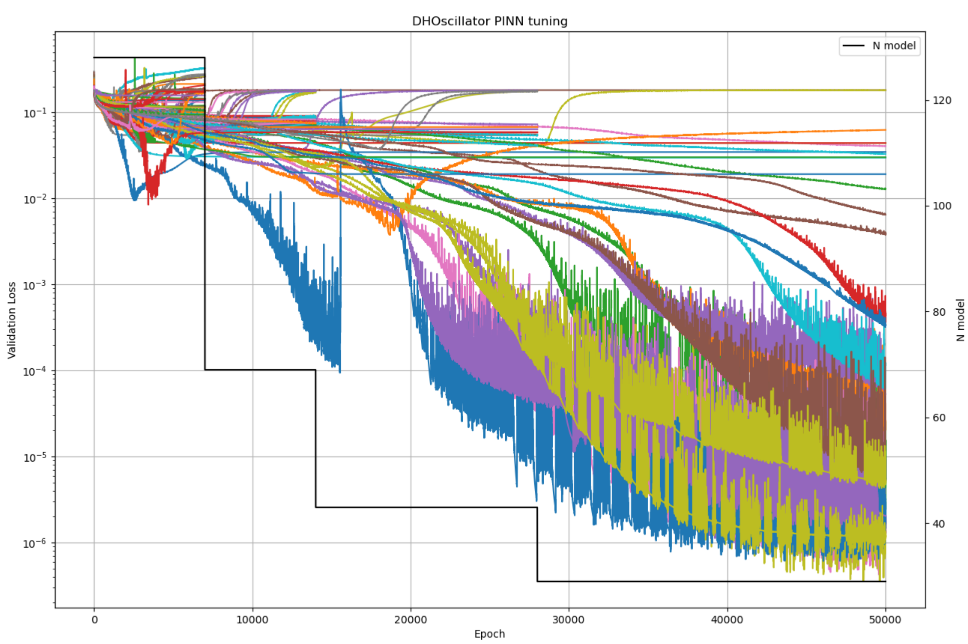
[23]:
```
         loss   timestamp checkpoint_dir_name  done  training_iteration  \
0    0.094820  1710247709                None  True                7000
1    0.102514  1710247424                None  True                7000
2    0.000015  1710248027                None  True               50000
3    0.178123  1710246995                None  True                7000
4    0.000001  1710248556                None  True               50000
..        …          …                  …    …                  …
123  0.175027  1710255092                None  True                7000
124  0.157358  1710255262                None  True                7000
125  0.185765  1710255140                None  True                7000
126  0.082898  1710255341                None  True               14000
127  0.177847  1710255407                None  True               28000

        trial_id                 date  time_this_iter_s  time_total_s   pid  \
0    9b065_00013  2024-03-12_13-48-29          0.006425     36.962976  5343
1    9b065_00011  2024-03-12_13-43-44          0.020860    128.469085  5329
2    9b065_00000  2024-03-12_13-53-47          0.009690    467.900811  5326
3    9b065_00002  2024-03-12_13-36-35          0.013086    103.356977  5330
4    9b065_00012  2024-03-12_14-02-36          0.008346    383.581897  5329
..         …                  …               …           …   …
123  9b065_00123  2024-03-12_15-51-32          0.049581    209.852490  5344
124  9b065_00124  2024-03-12_15-54-22          0.039076    314.353094  5335
125  9b065_00125  2024-03-12_15-52-20          0.013357    132.105762  5342
126  9b065_00126  2024-03-12_15-55-41          0.014910    227.598845  5343
127  9b065_00127  2024-03-12_15-56-47          0.003831    178.210384  5344

     … iterations_since_restore config/n_layers  config/n_neurons  \
0    …                     7000               2                32
1    …                     7000               2                38
2    …                    50000               5                33
3    …                     7000               3                37
4    …                    50000               4                28
..   …                      …             …                …
123  …                     7000               4                30
124  …                     7000               2                28
125  …                     7000               5                24
126  …                    14000               4                39
127  …                    28000               5                27
```

```
     config/lr  config/factor  config/patience  config/batch_size  \
0     0.002414       0.729229              176                541
1     0.002808       0.869355              123                119
2     0.001432       0.739127              616                894
3     0.005881       0.730474              887                118
4     0.004121       0.749430              618                946
..         …              …                 …                  …
123   0.002230       0.953331              969                117
124   0.008645       0.911701              270                115
125   0.007100       0.810383              223                299
126   0.007937       0.951226              102                624
127   0.001029       0.930717              567                520

     config/grid_num  config/lambda       logdir
0                329      51.560005  9b065_00013
1                464      29.381437  9b065_00011
2                269      81.953152  9b065_00000
3                121      47.442912  9b065_00002
4                303      23.923369  9b065_00012
..               …            …           …
123              368      16.457702  9b065_00123
124              998      22.106460  9b065_00124
125              587      21.774259  9b065_00125
126              787      38.707303  9b065_00126
127              127      19.321857  9b065_00127

[128 rows x 23 columns]
```

[24]:
```python
def get_alive_model(df, max_epoch):
    """
    Function to get the number of alive models at each epoch
    input: df, max_epoch
    output: alive_model
    """
    # get traininig_iteration vector
    training_iteration = df["training_iteration"]
    training_iteration = training_iteration.to_numpy()
    # alive_model = number of entries of training_iteration > epoch
    # epoch = (0, max_epoch)
    alive_model = np.zeros(max_epoch)
    for i in range(max_epoch):
        alive_model[i] = np.sum(training_iteration > i)
    return alive_model

alive_model = get_alive_model(restored_df, 50000)
```

[25]:

```
# show results
dfs = {result.path: result.metrics_dataframe for result in restored_results}

# twinx plot alive_model and validation loss
fig, ax1 = plt.subplots(figsize=(15, 10))
# plot the validation loss
for path, df in dfs.items():
    ax1.plot(df["training_iteration"], df["loss"], label=path)
ax1.set_yscale("log")
ax1.set_xlabel("Epoch")
ax1.set_ylabel("Validation Loss")
ax1.grid()

# plot the alive model
ax2 = ax1.twinx()
ax2.plot(alive_model, label="N model", color="black")
ax2.set_ylabel("N model")
ax2.legend()
ax2.grid()


plt.title("DHOscillator PINN tuning")
plt.grid()
# save the figure
plt.savefig('../plot/DHOscillator_PINN_tuning.png')
```



1.5 Compare with RK5

[30]:
```
# get analytical solution at t = 25
t_test_start = 25
t_test_end = 30
Y_start = analytical_solution(t_test_start)
t_span = [t_test_start, t_test_end]
```

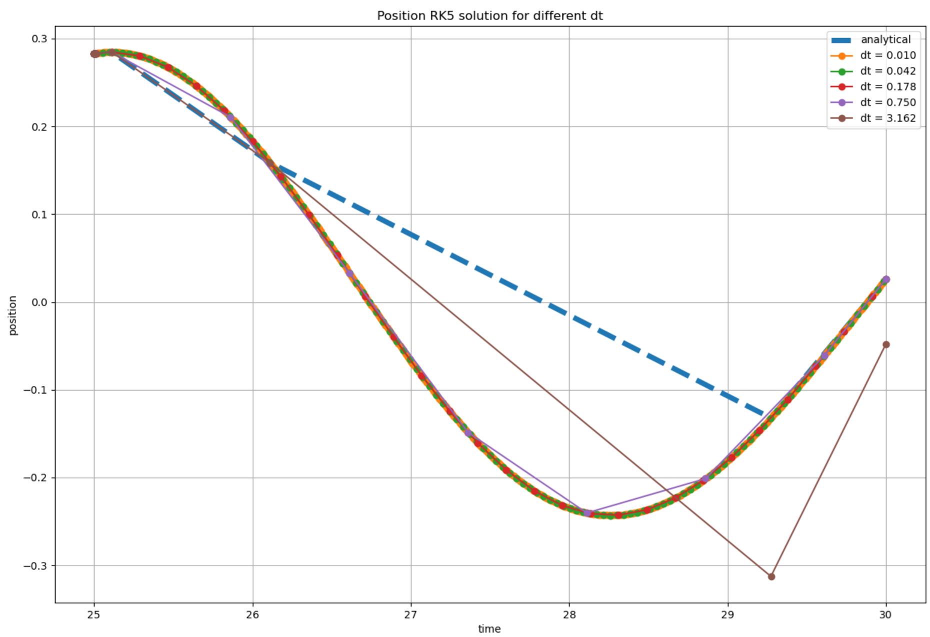
[31]:
```
# let dt_steps = [0.1, 0.01, 0.001, 0.0001]
# for each dt_step solve the ode and get the mse for t > 25
# and the time of execution
sols = []
mses = []
exec_times = []

# linspae log scale
dt_steps = np.logspace(-2, 0.5, 5, endpoint=True)

t_span=(25,30)

for dt_step in dt_steps:
    start_time = time.time()
    sol = RK5(dumped_spring, Y_start, t_span, dt_step)
    end_time = time.time()
    exec_times.append(end_time - start_time)
    sols.append(sol)
    mse = np.mean((sol.y[0,:] - analytical_solution(sol.t)[0])**2 + (sol.y[1,:]
 ↪- analytical_solution(sol.t)[1])**2)
    mses.append(mse)
```

```
/home/luigi/anaconda3/envs/ray/lib/python3.9/site-
packages/scipy/integrate/_ivp/common.py:39: UserWarning: The following arguments
have no effect for a chosen solver: `min_step`.
  warn("The following arguments have no effect for a chosen solver: {}."
```

[35]:
```
# plot sols
plt.figure(figsize=(15, 10))
c=0
plt.plot(sol.t, analytical_solution(sol.t)[0], label='analytical',
 ↪linestyle='--', linewidth=5)

for sol in sols:
```

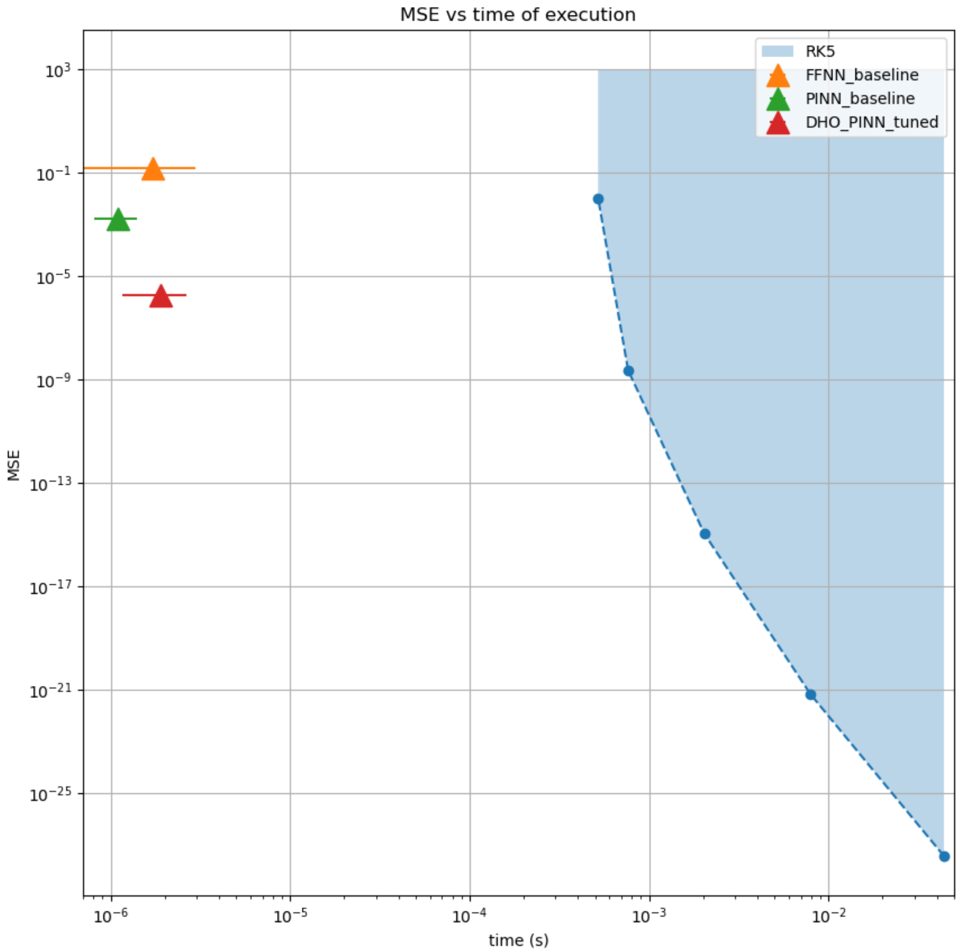
```
    plt.plot(sol.t, sol.y[0], label='dt = %.3f' %dt_steps[c], marker='o')
    c+=1
# add analytical solution
t = np.linspace(t_test_start, t_test_end, 1000)
# bigger linwidth
plt.grid()
plt.legend()
plt.title('Position RK5 solution for different dt')
plt.xlabel('time')
plt.ylabel('position')
```

[35]: Text(0, 0.5, 'position')



[49]:
```
# plot MSE vs time of execution for RK5
plt.figure(figsize=(10, 10))
plt.plot(exec_times, mses, 'o--')
plt.fill_between(exec_times, mses, 1000, alpha=0.3, label='RK5')

# plot the execution time of the models
marker = '^'
markersize = 15
```

```
plt.errorbar([FFNN_baseline_time], [FFNN_baseline_test_loss], xerr=FFNN_baseline_time_std, fmt=marker, markersize=markersize, label='FFNN_baseline')
plt.errorbar([PINN_baseline_time], [PINN_baseline_test_loss], xerr=PINN_baseline_time_std, fmt=marker, markersize=markersize, label='PINN_baseline')
plt.errorbar([DHO_PINN_tuned_time], [DHO_PINN_tuned_test_loss], xerr=DHO_PINN_tuned_time_std, fmt=marker, markersize=markersize, label='DHO_PINN_tuned')
plt.xscale('log')
plt.yscale('log')
plt.xlabel('time (s)')
plt.ylabel('MSE')
plt.grid()
plt.legend()
plt.title('MSE vs time of execution')
plt.xlim(7e-7, 5e-2)

# save the figure
plt.savefig('../plot/DHOscillator_MSE_vs_time.png')
```



MSE vs time of execution
RK5
FFNN_baseline
PINN_baseline
DHO_PINN_tuned
MSE
time (s)

[ ]:

[ ]: