Project 2 CompSci program, deadline May 31

CompSci program

Department of Physics, University of Oslo, Norway

Feb 21, 2023

Paths for project 2

Possible paths for project 2. We discuss here several paths as well as data sets for the second project.

1. The computational path: Here we propose a path where you develop your own code for a neural networks (or CNNs or RNNs) and apply this to data of your own selection. The code should be object oriented and flexible allowing for eventual extensions by including different Loss/Cost functions and other functionalities. Feel free to select data sets from those suggested below here. This code can also be extended upon by adding for example autoencoders. You can compare your own codes with implementations using TensorFlow(Keras)/PyTorch or other libraries.

2. The differential equation path: Here we propose a set of differential equations (ordinary and/or partial) to be solved first using neural networks (using either your own code or TensorFlow/Pytorch or similar libraries). Thereafter we plan to extend the set of methods for solving these equations to recurrent neural networks and autoencoders. All these approaches can be expanded into one large project. This project can also be extended into including Physics informed machine learning. Here we can discuss neural networks that are trained to solve supervised learning tasks while respecting any given law of physics described by general nonlinear partial differential equations.

3. The application path: Here you can use the most relevant method(s) (say neural networks, convolutional neural networks for images) and apply this(these) to data sets relevant for your own research.

4. And finally we propose also a partial differential equation path.

You can propose own data sets that relate to your research interests or just use existing data sets from say

1. Kaggle
2. The University of California at Irvine (UCI) with its machine learning repository.
3. For the differential equation problems, you can generate your own datasets, as described below.
4. If possible, you should link the data sets with existing research and analyses thereof. Scientific articles which have used Machine Learning algorithms to analyze the data are highly welcome. Perhaps you can improve previous analyses and even publish a new article?
5. A critical assessment of the methods with ditto perspectives and recommendations is also something you need to include.

The approach to the analysis of these new data sets should follow to a large extent what you did in project 1. That is:

1. Whether you end up with a regression or a classification problem, you should employ at least two of the methods we have discussed among **linear regression (including Ridge and Lasso)**, **Logistic Regression**, **Neural Networks**, **Convolution Neural Networks**, **Recurrent Neural Networks**, **Adversarial Neural Networks**, **Support Vector Machines** and **Decision Trees, Random Forests, Bagging and Boosting**.
2. The estimates you used and tested in project 1 should also be included, that is the $R2$-score, **MSE**, confusion matrix, accuracy score, information gain, ROC and Cumulative gains curves and other, cross-validation and/or bootstrap if these are relevant.
3. Similarly, feel free to explore various activations functions in deep learning and various approachs to stochastic gradient descent approaches.

All in all, the report should follow the same pattern as project 1, with abstract, introduction, methods, code, results, conclusions etc.

Solving differential equations with neural networks

Here we describe the possible differential equations we can study first with neural networks and thereafter with recurrent neural networks and/or AE and/or GANs.

The differential equations are given by the so-called Lorenz attractor model, and read

$$\frac{dx}{dt} = \sigma\left(y - x\right),$$

where $\sigma = 10$ is a constant

$$\frac{dy}{dt} = x\left(\rho - z\right) - y,$$

with $\rho = 28$ and

$$\frac{dz}{dt} = xy - \beta z$$

with $\beta = 8/3$ as our final constant.

The following function is a simple function which sets up the solution using the ordinary differential library which follows **NumPy**. Here we have fixed the time sted $\Delta t = 0.01$ and the final time $t_f = 8$.

The program sets 100 random initial values and produces inputs and outputs for a neural network calculations. The inputs are given by the values of the array $\boldsymbol{x}$ (which contains x, y, z as functions of time) for the time step $\boldsymbol{x}_t$. The other array defined by $\boldsymbol{x}_{t+1}$ contains the outputs (or targets) which we want the neural network to reproduce.

```
# Common imports
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
import os

# Where to save the figures and data files
PROJECT_ROOT_DIR = "Results"
FIGURE_ID = "Results/FigureFiles"
DATA_ID = "DataFiles/"

if not os.path.exists(PROJECT_ROOT_DIR):
    os.mkdir(PROJECT_ROOT_DIR)

if not os.path.exists(FIGURE_ID):
    os.makedirs(FIGURE_ID)

if not os.path.exists(DATA_ID):
    os.makedirs(DATA_ID)

def image_path(fig_id):
    return os.path.join(FIGURE_ID, fig_id)

def data_path(dat_id):
    return os.path.join(DATA_ID, dat_id)

def save_fig(fig_id):
    plt.savefig(image_path(fig_id) + ".png", format='png')
```

```
# Selection of parameter values and setting array for time
dt =0.01; tfinal = 8
t = np.arange(0,tfinal+dt, dt)
beta =8.0/3.0; rho = 28.0; sigma = 10.0

# define the inputs and outputs for the neural networks
nninput = np.zeros((100*len(t)-1,3))
nnoutput = np.zeros((100*len(t)-1,3))
# Define the equations to integrate
def lorenz_derivative(xyz, t0, sigma=sigma,beta=beta,rho=rho):
    x, y, z = xyz
    return [sigma*(x-y), x*(rho-z)-y, x*y-beta*z]

# generate 100 random initial values
x0 = -15.0+30.0*np.random.random((100,3))

# Use odeint functionality by sending in derivative function
# Feel free to change the choice of integrator
x_t = np.asarray([odeint(lorenz_derivative, x0_j, t)
                  for x0_j in x0])

# define the inputs and outputs for the neural networks
for j in range(100):
    nninput[j*(len(t)-1):(j+1)*(len(t)-1),:] = x_t[j,:-1,:]
    nnoutput[j*(len(t)-1):(j+1)*(len(t)-1),:] = x_t[j,1:,:]
```

The input and output variables are those we will start trying our network with. Your first taks is to set up a neural code (either using your own code or TensorFlow/PyTorch or similar libraries)) and use the above data to a prediction for the time evolution of Lorenz system for various values of the randomly chosen initial values. Study the dependence of the fit as function of the architecture of the network (number of nodes, hidden layers and types of activation functions) and various regularization schemes and optimization methods like standard gradient descent with momentum, stochastic gradient descent with batches and with and without momentum and various schedulers for the learning rate.

Feel free to change the above differential equations. As an example, consider the following harmonic oscillator equations solved with the Runge-Kutta to fourth order method. This is a one-dimensional problem and it produces a position x_t and velocity v_t. You could now try to fit both the velocities and positions using much of the same recipe as for Lorenz attractor. You will find it convenient to analyze one set of initial conditions first. The code is included here.

This code is an example code that solves Newton's equations of motion with a given force and produces an output which in turn can be used to train a neural network

```
# Common imports
import numpy as np
import pandas as pd
from math import *
import matplotlib.pyplot as plt
import os

# Where to save the figures and data files
PROJECT_ROOT_DIR = "Results"
FIGURE_ID = "Results/FigureFiles"
DATA_ID = "DataFiles/"

if not os.path.exists(PROJECT_ROOT_DIR):
    os.mkdir(PROJECT_ROOT_DIR)

if not os.path.exists(FIGURE_ID):
    os.makedirs(FIGURE_ID)

if not os.path.exists(DATA_ID):
    os.makedirs(DATA_ID)

def image_path(fig_id):
    return os.path.join(FIGURE_ID, fig_id)

def data_path(dat_id):
    return os.path.join(DATA_ID, dat_id)

def save_fig(fig_id):
    plt.savefig(image_path(fig_id) + ".png", format='png')


def SpringForce(v,x,t):
#   note here that we have divided by mass and we return the acceleration
    return  -2*gamma*v-x+Ftilde*cos(t*Omegatilde)


def RK4(v,x,t,n,Force):
    for i in range(n-1):
# Setting up k1
        k1x = DeltaT*v[i]
        k1v = DeltaT*Force(v[i],x[i],t[i])
# Setting up k2
```

```
        vv = v[i]+k1v*0.5
        xx = x[i]+k1x*0.5
        k2x = DeltaT*vv
        k2v = DeltaT*Force(vv,xx,t[i]+DeltaT*0.5)
# Setting up k3
        vv = v[i]+k2v*0.5
        xx = x[i]+k2x*0.5
        k3x = DeltaT*vv
        k3v = DeltaT*Force(vv,xx,t[i]+DeltaT*0.5)
# Setting up k4
        vv = v[i]+k3v
        xx = x[i]+k3x
        k4x = DeltaT*vv
        k4v = DeltaT*Force(vv,xx,t[i]+DeltaT)
# Final result
        x[i+1] = x[i]+(k1x+2*k2x+2*k3x+k4x)/6.
        v[i+1] = v[i]+(k1v+2*k2v+2*k3v+k4v)/6.
        t[i+1] = t[i] + DeltaT


# Main part begins here

DeltaT = 0.001
#set up arrays
tfinal = 20 # in dimensionless time
n = ceil(tfinal/DeltaT)
# set up arrays for t, v, and x
t = np.zeros(n)
v = np.zeros(n)
x = np.zeros(n)
# Initial conditions (can change to more than one dim)
x0 =  1.0
v0 = 0.0
x[0] = x0
v[0] = v0
gamma = 0.2
Omegatilde = 0.5
Ftilde = 1.0
# Start integrating using Euler's method
# Note that we define the force function as a SpringForce
RK4(v,x,t,n,SpringForce)

# Plot position as function of time
fig, ax = plt.subplots()
ax.set_ylabel('x[m]')
ax.set_xlabel('t[s]')
```

```
ax.plot(t, x)
fig.tight_layout()
save_fig("ForcedBlockRK4")
plt.show()
```

The next step is to include recurrent neural networks. These will be discussed in connection with coming lectures.

Finally we add a so-called differential equation path as well.

Solving partial differential equations with neural networks

For this variant of project 2, we will assume that you have some background in the solution of partial differential equations using finite difference schemes. In the lecture slides from weeks 6 and 7 you may find additional material. We will study the solution of the diffusion equation in one dimension using a standard explicit or implicit scheme and neural networks to solve the same equations.

For the explicit and/or implicit schemes, you can study for example chapter 10 of the lecture notes in Computational Physics or alternative sources.

For the machine learning part you can use the functionality of for example **Tensorflow/Keras**.

Part a), setting up the problem. The physical problem can be that of the temperature gradient in a rod of length $L = 1$ at $x = 0$ and $x = 1$. We are looking at a one-dimensional problem

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t}, t > 0, x \in [0, L]$$

or

$$u_{xx} = u_t,$$

with initial conditions, i.e., the conditions at $t = 0$,

$$u(x, 0) = \sin(\pi x) \quad 0 < x < L,$$

with $L = 1$ the length of the x-region of interest. The boundary conditions are

$$u(0, t) = 0 \quad t \geq 0,$$

and

$$u(L, t) = 0 \quad t \geq 0.$$

The function $u(x, t)$ can be the temperature gradient of a rod. As time increases, the velocity approaches a linear variation with x.

We will limit ourselves to the so-called explicit forward Euler algorithm with discretized versions of time given by a forward formula and a centered difference in space resulting in

$$u_t \approx \frac{u(x,t+\Delta t)-u(x,t)}{\Delta t} = \frac{u(x_i,t_j+\Delta t)-u(x_i,t_j)}{\Delta t}$$

and

$$u_{xx} \approx \frac{u(x+\Delta x,t)-2u(x,t)+u(x-\Delta x,t)}{\Delta x^2},$$

or

$$u_{xx} \approx \frac{u(x_i+\Delta x,t_j)-2u(x_i,t_j)+u(x_i-\Delta x,t_j)}{\Delta x^2}.$$

Write down the algorithm and the equations you need to implement. Find also the analytical solution to the problem.

Part b). Implement the explicit scheme algorithm and perform tests of the solution for $\Delta x = 1/10$, $\Delta x = 1/100$ using Δt as dictated by the stability limit of the explicit scheme. The stability criterion for the explicit scheme requires that $\Delta t/\Delta x^2 \leq 1/2$.

Study the solutions at two time points t_1 and t_2 where $u(x,t_1)$ is smooth but still significantly curved and $u(x,t_2)$ is almost linear, close to the stationary state.

Part c) Neural networks. Study now the lecture notes on solving ODEs and PDEs with neural network and use either your own code or the functionality of tensorflow/keras to solve the same equation as in part b). Discuss your results and compare them with the standard explicit or implicit scheme. Include also the analytical solution and compare with that.

Part d). Finally, present a critical assessment of the methods you have studied and discuss the potential for the solving differential equations and eigenvalue problems with machine learning methods.

Introduction to numerical projects

Here follows a brief recipe and recommendation on how to write a report for each project.

- Give a short description of the nature of the problem and the eventual numerical methods you have used.
- Describe the algorithm you have used and/or developed. Here you may find it convenient to use pseudocoding. In many cases you can describe the algorithm in the program itself.

- Include the source code of your program. Comment your program properly.
- If possible, try to find analytic solutions, or known limits in order to test your program when developing the code.
- Include your results either in figure form or in a table. Remember to label your results. All tables and figures should have relevant captions and labels on the axes.
- Try to evaluate the reliabilty and numerical stability/precision of your results. If possible, include a qualitative and/or quantitative discussion of the numerical stability, eventual loss of precision etc.
- Try to give an interpretation of you results in your answers to the problems.
- Critique: if possible include your comments and reflections about the exercise, whether you felt you learnt something, ideas for improvements and other thoughts you've made when solving the exercise. We wish to keep this course at the interactive level and your comments can help us improve it.
- Try to establish a practice where you log your work at the computerlab. You may find such a logbook very handy at later stages in your work, especially when you don't properly remember what a previous test version of your program did. Here you could also record the time spent on solving the exercise, various algorithms you may have tested or other topics which you feel worthy of mentioning.

Format for electronic delivery of report and programs

The preferred format for the report is a PDF file. You can also use DOC or postscript formats or as an ipython notebook file. As programming language we prefer that you choose between C/C++, Fortran2008 or Python. The following prescription should be followed when preparing the report:

- Send us an email in order to hand in your projects with a link to your GitHub/Gitlab repository.
- In your GitHub/GitLab or similar repository, please include a folder which contains selected results. These can be in the form of output from your code for a selected set of runs and input parameters.

Finally, we encourage you to collaborate. Optimal working groups consist of 2-3 students. You can then hand in a common report.

Software and needed installations

If you have Python installed (we recommend Python3) and you feel pretty familiar with installing different packages, we recommend that you install the following Python packages via **pip** as

1. pip install numpy scipy matplotlib ipython scikit-learn tensorflow sympy pandas pillow

For Python3, replace **pip** with **pip3**.

See below for a discussion of **tensorflow** and **scikit-learn**.

For OSX users we recommend also, after having installed Xcode, to install **brew**. Brew allows for a seamless installation of additional software via for example

1. brew install python3

For Linux users, with its variety of distributions like for example the widely popular Ubuntu distribution you can use **pip** as well and simply install Python as

1. sudo apt-get install python3 (or python for python2.7)

etc etc.

If you don't want to install various Python packages with their dependencies separately, we recommend two widely used distrubutions which set up all relevant dependencies for Python, namely

1. Anaconda Anaconda is an open source distribution of the Python and R programming languages for large-scale data processing, predictive analytics, and scientific computing, that aims to simplify package management and deployment. Package versions are managed by the package management system **conda**

2. Enthought canopy is a Python distribution for scientific and analytic computing distribution and analysis environment, available for free and under a commercial license.

Popular software packages written in Python for ML are

- Scikit-learn,
- Tensorflow,
- PyTorch and
- Keras.

These are all freely available at their respective GitHub sites. They encompass communities of developers in the thousands or more. And the number of code developers and contributors keeps increasing.