Assignment_00

April 14, 2021

1 Assignment 00 - Python

As an introduction to the assignments, we will start with a few basic Python operations and concepts. If anything is unlcear or if you find errors, feel free to post in the forum set up in Ilias.

You can submit incomplete assignments that don't validate. If a test cell validates correctly, you will get the points. If not, you will receive feedback from us with some remarks about what might be wrong.

1.1 Operators (1 point)

In the lecture, we haven't talked about the operators `//`and `%`. Use the following cell to check what they do using different numbers:

[ ]:
```
a = 317
b = 17

print(a // b)
print(a % b)
```

Now that you know what these operators produce, you should see how you can use the results and the value of variable `b` to reconstruct number `a`.

Write a function that takes as input the result of operation `//` called `intdiv`, the result of operation `%` called `mod` and number `b`. It should calculate number `a` and `return` the result:

[ ]:
```
def reconstruct(intdiv, mod, b):
    # YOUR CODE HERE
    raise NotImplementedError()
```

The cell above contains a comment saying `### YOUR SOLUTION HERE` and a `NotImplemented` exception, both of which you can delete, so you can place your solution there.

You can test your function in the following cell:

[ ]:
```
a = 317
b = 17

result = reconstruct(a // b, a % b, b)
print(result)
```

The following cell contains tests. These will be used to determine if you solved the problem correctly (or creatively tricked the system). Taking a look at them might give you some hints about what is expected here, but sometimes some of the tests, or all of them, will be hidden.

When the check above yields the correct number, execute the test cell below to see if your function does the correct thing:

[ ]:
```
assert reconstruct(317 // 17, 317 % 17, 17) == 317
import random as r
import math as m
a = m.floor(100*r.random())
b = m.floor(100*r.random())
assert reconstruct(a // b, a % b, b) == a
```

If anything went wrong, the above cell will throw an `AssertionError` and show you which test failed. Take a good look at it to get some hints as to what went wrong. Always make sure your functions are as general as possible.

1.2 Modules (1 point)

Sometimes you need a combination of different functions that you imported more than once. In that case it often makes sense to write a function that wraps the functionality of this combination.

Write a function that takes a number as input and returns $\frac{1}{2}\left(\exp\left(x\right) - \exp\left(-x\right)\right)$:

[ ]:
```
# you need to import math here

def sh(x):
    # YOUR CODE HERE
    raise NotImplementedError()
```

In this case, there is a hidden test below that will check if you correctly implemented the desired functionality. Do you recognize which function is built here?

[ ]:
```
import math as m
assert sh(0) == 0.0
assert sh(m.log(0.5*(1+5**0.5))) == 0.5
```

1.3 Cancellation (1 point)

We talked about cancellation problems a little bit in the lecture. A consequence of this problem is that sometimes, when you check the result of an operation to be zero, your check will fail, since it is not exactly zero. Numerical zeros are extremely small though, so in most cases it suffices to compare the result to a very small number, the **tolerance**, that you have to define sensibly as a programmer.

Write a function that takes as input a result `res` from some calculation and a tolerance `tol`. It should return `True` if the result is close to zero within the given tolerance and return `False` otherwise:

[ ]:
```
def zero_check(res, tol):
    # YOUR CODE HERE
    raise NotImplementedError()
```

[ ]:
```
import numpy as np
assert zero_check(0.1 + 0.1 + 0.1 - 0.3, 1e-16) == True
assert zero_check(np.exp(np.arcsinh(0.5)) - 0.5 * (1+5.00005**0.5), 1e-6) ==␣
↪False
```

1.4 Lists (1 point)

Assume you have a set of values that you want to test whether they are zero with a fixed, given tolerance. These values are in the list `values`.

Create a list called `checks` that contains the checks for all values using your function `zero_check` from above. *Hint*: there is an easy way to do this in a single line.

[ ]:
```
tol = 5e-7
vals = [1.73275932e-7, 7.02650253e-7, 6.17275235e-6, -5.21986805e-6, 1.
↪25262623e-8, \
        3.31753765e-7, 8.36552362e-6, -6.08163535e-8, 8.01356532e-7, 3.
↪75065322e-6]

# YOUR CODE HERE
raise NotImplementedError()
```

[ ]:
```

```

1.5 Classes (5 points)

Although you won't have to use object-oriented programming during the course, it makes sense to understand roughly what classes do and how they work. We will use an example from FEM here and try to think of it in an object-oriented way.

In FEM, you describe a model using **nodes** and **elements** in a **mesh**. A mesh consists of nodes, which are arranged in elements. We will constrain our analysis to elements here for brevity's sake. Two classes are needed here, `Node` and `Element`. Remember that a class has attributes (things it know) and methods (actions it can take). Additionally, elements *have* nodes associated with it, so the two need to be connected somehow. We will implement a very rudimentary version of that concept here.

Implement a `Node` class. It should have as attributes a `list` of coordinates, that describes its position in the mesh. Assume a 2d mesh, so you'll only need to save two coordinates per node. As methods, the node should be able to print its own coordinates, and to add a displacement to its coordinates.

Use the following skeleton code:

[ ]:
```
class Node():
    def __init__(self, x_coord, y_coord):
        # self.coordinates = [ ?? ]
        # YOUR CODE HERE
        raise NotImplementedError()

    def print_coordinates(self):
        # print( ?? )
        # YOUR CODE HERE
        raise NotImplementedError()

    def add_displacement(self, x_disp, y_disp):
        # self.coordinates[ ?? ] += ??
        # YOUR CODE HERE
        raise NotImplementedError()
```

Check that your class does what you expect below:

[ ]:
```
node1 = Node(0.0, 0.0)
node2 = Node(0.0, 1.0)
node3 = Node(1.0, 1.0)
node4 = Node(1.0, 0.0)

node2.print_coordinates()
node2.add_displacement(0.1, 0.2)
node2.print_coordinates()
```

[ ]:
```
testnode = Node(0.5, 0.5)
assert type(testnode.coordinates) == type([])
```

[ ]:
```
testnode = Node(-0.8, 0.8)
testnode2 = Node(-0.8, 0.8)
testnode2.add_displacement(0.3, 0.7)
assert testnode2.coordinates == [testnode.coordinates[0] + 0.3, testnode.
 ↪coordinates[1] + 0.7]
```

Now write an `Element` class, that is instanciated empty, but can save a list of nodes. For adding a node, write a method that does this. Additionally, write a method that iterates over all nodes of the element and prints their coordinates. For later plotting, also write a function that returns a list of all nodal coordinates as a list called `get_nodal_coordinates()`. The

list should look like this ind the end: `[[x-coordinate_of_node1, y-coordinate_of_node1], [x-coordinate_of_node2, y-coordinate_of_node2], ...]`.

Use the following skeleton code:

[ ]:
```
class Element():
    def __init__(self):
        self.nodes = []

    def add_node(self, node):
        # YOUR CODE HERE
        raise NotImplementedError()

    def print_nodal_coordinates(self):
        # YOUR CODE HERE
        raise NotImplementedError()

    def get_nodal_coordinates(self):
        # YOUR CODE HERE
        raise NotImplementedError()
```

Check that your class does what you desire in the cell below:

[ ]:
```
# play with these lines and numbers, but
node1 = Node(0.0, 0.0)
node2 = Node(0.0, 1.0)
node3 = Node(1.0, 1.0)
node4 = Node(1.0, 0.0)

element1 = Element()

element1.add_node(node1)
element1.add_node(node2)
element1.add_node(node3)
element1.add_node(node4)

element1.nodes[1].add_displacement(0.0, 0.45)
element1.nodes[2].add_displacement(0.25, 0.25)
element1.nodes[3].add_displacement(0.35, 0.1)

element1.print_nodal_coordinates()

## you might need to adjust the x- and y-limits here:
import matplotlib.patches as patches
import matplotlib.pyplot as plt

%matplotlib notebook
```

```
fig = plt.figure()
ax = fig.add_subplot(111, aspect='equal')
ax.set_xlim(-0.1, 2.0)
ax.set_ylim(-0.1, 2.0)

print(element1.get_nodal_coordinates())

ax.add_patch(patches.Polygon(xy=[*element1.get_nodal_coordinates()],␣
↪color="darkorange"))

plt.show()
```

[ ]:
```
from random import random
from math import isclose
node1 = Node(0.0, 0.0)
node2 = Node(0.0, 1.0)
node3 = Node(1.0, 1.0)
node4 = Node(1.0, 0.0)
u_r = [random(), random()]
node3.add_displacement(*u_r)
element1 = Element()
element1.add_node(node1)
element1.add_node(node2)
element1.add_node(node3)
element1.add_node(node4)
assert isclose(sum([sum(node.coordinates) for node in element1.nodes]), 4.0 +␣
↪sum(u_r))
```

[ ]:
```
assert type(element1.nodes) == type([])
for node in element1.nodes:
    assert type(node) == type(node1)
```

[ ]:
```

```

Assignment_01

April 14, 2021

1 Assignment 02

1.1 Quadratic Forms (3 points)

Let's take a look at how different matrices produce different "distance landscapes". In the cell below, create a function called `qform()` that takes as input a matrix and two vectors (in that order!), then returns the resulting quadratic form. Make it as general as possible (it should also work for 5-dimensional matrices and vectors, for example).

[ ]:
```
import numpy as np

v1 = np.array([1.0, 1.0, 1.0])
v2 = np.array([0.0, 2.0, 7.0])

A  = np.array([[1.0, 0.0, 0.0], \
               [0.0, 1.0, 0.0], \
               [0.0, 0.0, 1.0]])

# YOUR CODE HERE
raise NotImplementedError()
```

[ ]:
```
A9, v10, v11 = np.eye(3), np.random.rand(3), np.random.rand(3)
assert qform(A, v10, v11) == v10.T @ v11
assert qform(A9, v10, v10) == v10.T @ v10
```

As mentioned in the lecture when you have a notion of *length*, you automatically have a notion of *distance* by simply measuring the length of the difference of two vectors $||\Delta\mathbf{v}|| = ||\Delta\mathbf{v}_2 - \mathbf{v}_1||$, or, in this case, applying the quadratic form to the difference of two vectors.

Let's see what the qform function does to a variety of 2d vector differences by plotting the resulting value for difference vectors that exist around the origin. For example, if the difference of two vectors is $[1, 1]$, the resulting qform result would show up in the plot at $x = 1, y = 1$.

In the following code, apply your `qform()` function from above to an array of vectors called `vecs`, that is created below:

[ ]:
```
# this makes the resulting image a bit more interactive and rotatable in Jupyter
%matplotlib notebook

# from matplotlib, we only need the "pyplot" class
```

```
import matplotlib.pyplot as plt
# we need the "axes3d" class for 3d plots, so in 2d, this can be omitted
from mpl_toolkits.mplot3d import axes3d


## define your matrix for the quadratic form here
A = np.array([[1.0, 0.0], \
              [0.0, 1.0]])


# first, a "figure" object needs to be created
fig = plt.figure(figsize=(8,5))


# this is an "axis" object, which you can use to plot.
# the "111" means, that this will only contain a single image
# later, we will use subplots with more figures, where the
# first 2 numbers indicate the number of plots per dimension,
# like "224" would indicate 2 by 2 plots. The last number gives
# the total number of plots.
# "projection='3d'" makes this "axis"-object 3d-capable
ax = fig.add_subplot(111, projection='3d')


# before something can be plotted, you need a space of inputs
mesh_points = np.linspace(-2,2,20)


# to create coordinate matrices out of this space (similar to matlab):
x, y = np.meshgrid(mesh_points, mesh_points)


# we need the vectors of these coordinates for qform to act on
vecs = np.array([x.reshape(400), \
                 y.reshape(400)]).T


# this will plot all our vectors, to see we've done it correctly
ax.scatter(*vecs.T, color='darkorange', s=0.2)


# apply qform function to the vectors
## instructions: create a variable "q" here, which will contain the results
## of the qform, applied on "vecs". Make it a numpy array. It should have
## shape (400,)
# YOUR CODE HERE
raise NotImplementedError()
```

```
print(q.shape)


# the plot functions expect matrices, so we have to reshape the results
# to fit the shape of the coordinate arrays x and y
q = q.reshape(20,20)


# this will plot the surface of the action on the vectors
ax.plot_surface(x, y, q, cmap='viridis')


# with this, you can include a projection plot, where "zdir" gives the direction
ax.contourf(x, y, q, zdir='z', offset=0, cmap='viridis')


# you might need to adjust the axes limits
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)
ax.set_zlim( 0, 8)


# this displays the plot on the screen
plt.show()
```

Feel free to try out a few different matrices and see how that changes the landscape. See for example, what the matrix [[1, 0], [0, -1]] does and how that affects the landscape, or skew it by introducing non-diagonal elements.

[ ]:

1.2 Singular Matrices (3 points)

We can calculate the pseudoinverse even of singular and non-square matrices. Take for example `B` in the following cell and calculate its pseudo-inverse `B_plus`:

[ ]:
```
import numpy as np

B = np.array([[1.0, 0.5, 1.0], \
              [0.9, 1.0, 0.0], \
              [0.0, 0.0, 0.0]])

print(np.linalg.det(B))

# YOUR CODE HERE
raise NotImplementedError()
```

Check that it is indeed the pseudo-inverse here:

[ ]:
```
print(B @ B_plus)
```

[ ]:
```
assert np.allclose(B @ B_plus, np.diag([1, 1, 1])) or \
       np.allclose(B @ B_plus, np.diag([1, 1, 0])) or \
       np.allclose(B @ B_plus, np.diag([1, 0, 0]))
```

Now let's explore what the action of the singular matrix and its pseudo-inverse on vectors looks like in 3D. Complete the following code:

[ ]:
```
%matplotlib notebook

import matplotlib.patches as patches
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d
from mpl_toolkits.mplot3d.art3d import Poly3DCollection, Line3DCollection

# this function helps building the vectors for plotting a parallelepiped
def create_parallelepiped_3d(v1, v2, v3):
    origin = np.array([0.0, 0.0, 0.0])
    return [[origin, v1, v1+v2, v2], \
            [v3, v1+v3, v1+v2+v3, v2+v3], \
            [origin, v1, v1+v3, v3], \
            [v1+v2, v2, v2+v3, v1+v2+v3], \
            [v1, v1+v2, v1+v2+v3, v1+v3], \
            [v3, v2+v3, v2, origin]]


# plot setup, this time we will create two plots
fig = plt.figure(figsize=(8,4))
ax1 = fig.add_subplot(121, projection='3d')
ax1.set_xlim(-1.0, 2.0)
ax1.set_ylim(-1.0, 2.0)
ax1.set_title(label="B")
ax2 = fig.add_subplot(122, projection='3d')
ax2.set_xlim(-1.0, 2.0)
ax2.set_ylim(-1.0, 2.0)
ax2.set_title(label="B_plus B")


# create a matrix of the basis vectors
E_vecs = np.eye(3)

## instructions: create two matrices of vectors similar to "E_vecs",
## called B_vecs and B_plus_vecs, which contain the resulting vectors
## after applying B, and after applying B_plus*B respectivly
# YOUR CODE HERE
```

```
raise NotImplementedError()


# create the parallelpipeds
E_rhombus = create_parallelepiped_3d(*E_vecs)
B_rhombus = create_parallelepiped_3d(*B_vecs)
B_plus_rhombus = create_parallelepiped_3d(*B_plus_vecs)


# plot original square
ax1.add_collection3d(Poly3DCollection(E_rhombus,
     facecolors='lightgray', linewidths=1, edgecolors='darkgray', alpha=.25))
# apply B
ax1.add_collection3d(Poly3DCollection(B_rhombus,
     facecolors='darkorange', linewidths=1, edgecolors='red', alpha=.25))


# plot B_vecs
ax2.add_collection3d(Poly3DCollection(B_rhombus,
     facecolors='lightgray', linewidths=1, edgecolors='darkgray', alpha=.25))
# apply B_plus
ax2.add_collection3d(Poly3DCollection(B_plus_rhombus,
     facecolors='darkorange', linewidths=1, edgecolors='red', alpha=.25))


plt.tight_layout()
plt.show()
```

If everything worked, you should see that a 3d shape gets reduced to a 2d shape by a singular matrix and hence, that in general, information is lost and cannot be reconstructed even with the pseudoinverse. It's a projection of the original shape onto some 2d subspace of $\mathbb{R}^3$. You will need to rotate the images to see it.

[ ]:

1.3 Non-square matrices (1 point)

Non-square matrices can also introduce or destroy geometric information in linear systems. See what the non-square matrix C does to a vector below and how the pseudo-inverse retrieves the original vector.

Calculate the pseudo-inverse `C_plus` of the non-square matrix `C` below.

[ ]:
```
# C is 3x2
C = np.random.rand(3,2)
```

```
# YOUR CODE HERE
raise NotImplementedError()

v = np.array([1,1])

# see how the vector v gets transformed into a higher-dimensional space
print(C @ v)

# since no information was destroyed, the pseudo-inverse of C returns the␣
 ↪original vector
print(C_plus @ C @ v)
```

You see how the non-square matrix `C` transforms the 2d vector into 3d, adding some information, that is destroyed again by `C_plus`. We will later make use of this property for example in the kernel-trick, or in neural networks in general.

[ ]:

1.4 Pandas (3 points)

Someone provided you with some data in comma-separated form. In the cell below, use `pandas` to load the file `data.csv` as the variable `dataset`. If it worked, you should get an overview of the table.

[ ]:
```
import pandas as pd

# create a variable called "dataset" to load the data from the csv-file
# YOUR CODE HERE
raise NotImplementedError()

# don't change the next line, it will print a summary table
dataset
```

[ ]:
```
assert str(type(dataset)) == "<class 'pandas.core.frame.DataFrame'>"
```

As you can see, there are a few missing data points indicated by `NaN`. We can see this in the standard plot:

[ ]:
```
dataset.plot()
```

These missing data points pose a problem for many ways in which further processing may happen. Get rid of all rows that contain a `NaN`in the following cell, and save the result in a variable called `dataset_nonan`:

[ ]:
```
# create a variable called "dataset_nonan" to save the cleaned dataset
# YOUR CODE HERE
raise NotImplementedError()
```

```
# don't change the next line, it will print a summary table
dataset_nonan
```

[ ]:
```
assert dataset_nonan.size - dataset_nonan.count().sum() == 0
```

To check how many `NaN`s there are in a dataset, you can compare the total number of elements `dataset.size` to the number of elements that aren't `NaN` with `dataset.count().sum()`:

[ ]:
```
print("\t### Original dataset:###\n")
print("Number of elements in dataset: \t\t", dataset.size)
print("Number of values in dataset: \t\t", dataset.count().sum())
print("Number of NaNs = elements - values: \t", dataset.size - dataset.count().
 ↪sum())

print("\n\n\t### Non-nan dataset:###\n")
print("Number of elements in dataset_nonan: \t", dataset_nonan.size)
print("Number of values in dataset_nonan: \t", dataset_nonan.count().sum())
print("Number of NaNs = elements - values: \t", dataset_nonan.size -␣
 ↪dataset_nonan.count().sum())

dataset_nonan.plot()
```

The plot now doesn't have any gaps like before. Let's plot one column against another:

[ ]:
```
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets

def plot_scatter(labels):
    if labels:
        dataset_nonan.plot.scatter(x='petal width (cm)', y='petal length (cm)',␣
 ↪c='target', cmap='viridis')
    else:
        dataset_nonan.plot.scatter(x='petal width (cm)', y='petal length (cm)',␣
 ↪cmap='viridis')

interact(plot_scatter, labels=False)
```

The plot seems to indicate clustering and a linear trend. We'll see plots like these again later in the course. If you check the `labels` box, the plot will color the data points according to the column `target`, which contains the *labels* of the data. We'll come back to what this means later.

In the next cell, save your `dataset_nonan` dataframe as a csv-file called `dataset_nonan.csv`:

[ ]:
```
# You don't have to provide any extra arguments to the function here
# YOUR CODE HERE
raise NotImplementedError()
```

[ ]:

Assignment_02

April 14, 2021

0.1 Assignment 02 - Statistics I (due by 2nd of December)

The assignments for this exercise were scattered around the lecture today. You can either answer them on a piece of paper, scan/photograph your notes, or solve them in Latex as a PDF document and upload them into the `Assignment_02` folder, hit `submit` on the `Assignments` tab on top, or solve them here via Latex.

You'll find a PDF with the summarized homework in the `Assignment_02` folder in your home on the Jupyterhub.

To change a cell, simply double-click and type your solution like you would in a Latex-document. Almost everything works like you know it, raw text, `\begin{..}` environments, and other features that Latex offers. Don't provide a \begin{document}'

0.2 Set Theory (2 points)

We talked about a few properties of events A, B, C from a sample space S, prove them in the following cell:

YOUR ANSWER HERE

0.3 Probability Functions (2 points)

Show that for any $A \subset S$: $P(A) = \sum_i p_i$

YOUR ANSWER HERE

0.4 Legitimate Probability Functions (2 points)

YOUR ANSWER HERE

0.5 Coin Toss

Tossing three coins, show that the occurance of *head* on any toss has no effect on any other tosses.

YOUR ANSWER HERE

0.6 Continuous Distribution Functions (2 points)

YOUR ANSWER HERE

0.7 Binomial Distribution (2 points)

Let $Y = g(x)$, where $g(x) = n - x$. What is $f_Y(y)$?

YOUR ANSWER HERE

0.8 Expectation Value I (2 points)

In the lecture, we saw 4 essential properties of the expectation value. Prove at least the first one in the following cell:

YOUR ANSWER HERE

0.9 Expectation Value II (2 points)

In the lecture, we mentioned the distance between a random variable x and a constant b as $(x-b)^2$. Show that its expectation value is $\mathrm{E}(x-b)^2 = \mathrm{E}(x-\mathrm{E}(x))^2 + (\mathrm{E}(x)-b)^2$:

YOUR ANSWER HERE

Assignment_03

April 14, 2021

0.1 Assignment 03 - Statistics II (due by 9th of December)

The assignments for this exercise were scattered around the lecture today. You can either answer them on a piece of paper, scan/photograph your notes, or solve them in Latex as a PDF document and upload them into the `Assignment_03` folder, hit `submit` on the `Assignments` tab on top, or solve them here via Latex.

You'll find a PDF with the summarized homework in the `Assignment_03` folder in your home on the Jupyterhub.

To change a cell, simply double-click and type your solution like you would in a Latex-document. Almost everything works like you know it, raw text, `\begin{..}` environments, and other features that Latex offers. Don't provide a \begin{document}'

0.2 1. Probability Calculation for 2 Distributions, page 1 (2 Points)

Put your answer in the cell below, preferably in Latex-format:

[ ]:
```
# YOUR CODE HERE
raise NotImplementedError()
```

0.3 2. Proof for Expectation Expression, page 4 (2 Points)

Put your answer in the cell below, preferably in Latex-format:

[ ]:
```
# YOUR CODE HERE
raise NotImplementedError()
```

0.4 3. Mean and Variance for Binomial Distribution, page 5 (2 Points)

Put your answer in the cell below, preferably in Latex-format:

[ ]:
```
# YOUR CODE HERE
raise NotImplementedError()
```

0.5 4. Poisson Distribution Sum, page 6 (2 points)

Put your answer in the cell below, preferably in Latex-format:

[ ]:
```
# YOUR CODE HERE
raise NotImplementedError()
```

0.6 5. Geometric Distribution Mean and Variance, and Example, page 9 (2 Points)

Put your answer in the cell below, preferably in Latex-format:

[ ]:
```
# YOUR CODE HERE
raise NotImplementedError()
```

0.7 6. Express the Integrand, page 10 (2 points)

Put your answer in the cell below, preferably in Latex-format:

[ ]:
```
# YOUR CODE HERE
raise NotImplementedError()
```

0.8 7. Two Equations, page 16 (2 Points)

Put your answer in the cell below, preferably in Latex-format:

[ ]:
```
# YOUR CODE HERE
raise NotImplementedError()
```

0.9 8. Probability Computation, page 23 (2 Points)

Put your answer in the cell below, preferably in Latex-format:

[ ]:
```
# YOUR CODE HERE
raise NotImplementedError()
```

0.10 9. 2 Derivations, page 29 (2 points)

Put your answer in the cell below, preferably in Latex-format:

[ ]:
```
# YOUR CODE HERE
raise NotImplementedError()
```

---

linreg

April 14, 2021

1 Assignment 04: Shallow ML - Linear Regression

In this assignment, you will implement the most famous shallow learning algorithm: the linear regressor. We will restrict ourselves to the univariate case, but the transfer to multiple variables is straight forward. Please also refer to the class notes for more details and derivations.

Some notes: - Each cell starts with a comment on its contents and function to help you understand what is going on - Cells in which your input is required have a ### STUDENT ### Tag in the first line - if a cell does not have such a tag, you do not need to change anything there - You can also feed your own data to the regressor, just give your file the name 'data.csv'. - If you have problems with plots not showing up, try adding either '%matplotlib inline' or '%matplotlib notebook' to the cell in question - this is called a magic function in ipython and usually helps with that

1.1 The task

Your task to apply a linear regressor to the problem of predicting which grade or number of points a student will receive on the final exam. The data may have been collected from field studies among LRT students in Stuttgart. The only input feature is the number of hours studied before the test, the only output is the points achieved. This calls for a linear regression model with one feature.

Your tasks are: - implementing the forward pass - computing the cost function - computing the gradients and updating the parameters - optimizing the model - comparing the results with the solution of the normal form of the least squares problem

Let us start by reading in the data and plotting it:

[ ]:
```
# Import bibs and helper functions
#############################################################################################

import numpy as np
from numpy import genfromtxt
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from IPython import display
import matplotlib.cm as cm
from matplotlib.colors import LogNorm
from matplotlib import rcParams
rcParams.update({'font.size': 24})
from IPython.display import Math, Latex, Markdown
```

```
##########################################################################################
# helper function, allows printing of markdown syntax
# z.B: printmd(**Fettschrift**)
def printmd(string):
    display.display(Markdown(string))


# Declare Variables
##########################################################################################
filename="data.csv"
```

[ ]:
```
# Read Training Data and plot it
##########################################################################################
data_from_file = genfromtxt(filename, delimiter=',')
# the file data.csv contains two columns, separated by comma. The first column␣
 ↪contains the no. of hours studied,
# the second column the points on the test. Each row thus contains a␣
 ↪groundtruth pair (X,Y)

X=data_from_file[:,0]

# To underline that the Ys are the (T)ruth, we call the variable tY
tY=data_from_file[:,1]

nsamples=X.size
print("Number of read in samples: ",nsamples)

# We work with a linear model with one feature, so we will have two parameters
weights=np.zeros((2,1))
print("Current weights w0 and w1: ",weights[0],weights[1])
print("Dimension of parameter vector theta",weights.shape,"\n")

print("We extend the input vector by a column of ones to account for the bias␣
 ↪term")
print("Our linear model with a single input feature x_1 and parameters w_0 and␣
 ↪w_1 is given for a sample m as:")
display.display(Math(r'y^m=w_0+w_1\,x^m=w_0\cdot 1+w_1\,x^m=[1\,\,\, x^m]␣
 ↪\begin{bmatrix}w_0 \\ w_1 \\ \end{bmatrix}'))
X=np.c_[ np.ones(nsamples), X ]
X=np.array(X)
tY=np.array(tY)
tY=tY.reshape(nsamples,1)


# making sure that the dimensions are correct, note that each row corresponds␣
 ↪to a single sample
```

```
print("Shape of the inputs X",X.shape)
print("Shape of the outputs tY",tY.shape)


# Plot the training data
#########################################################################################

plt.figure(figsize=(20, 8))
area = np.pi*50
colors = np.random.rand(nsamples)
plt.scatter(X[:,1],tY,s=area, c=colors, alpha=0.9)
plt.xlabel('Hours studied')
plt.ylabel('Points on final exam')
display.display(plt.gcf())
display.clear_output(wait=True)
plt.grid(color='k', linestyle='--', linewidth=0.5)
plt.show()
```

Just by inspecting the data, we can see that a linear model might be a good idea, but it will certainly not be perfect. And yes, there are some students with negative points as well....

Let us try the linear regression for this!

1.1.1 Task 1:

The first step in the supervised learning method is to make a prediction based on the current parameters. Implement the forward pass through the linear model, that is the prediction of the model given some input x as a function of the current parameter vector w. Your function should work on multiple samples at a time, i.e. it should accept inputs of the (:,2) like we have seen for X.

[ ]:
```
### STUDENT ###
def forwardpass(X,w):
    # YOUR CODE HERE
    raise NotImplementedError()
```

[ ]:
```
# This cell tests your function "forwardpass" with the autograder
Xtest=[[1, 3],[3. ,1]]
thetatest=[2 ,1]
np.testing.assert_array_equal([5.,7.],forwardpass(Xtest,thetatest))
```

Let us now use your forwardpass to plot the current model:

[ ]:
```
# Compute the current prediction of the model for all training samples and plot
#########################################################################################

def showprediction(X,tY,w):
    printmd("## Prediction for w0 = "+str(w[0])+" **and w1 = **"+str(w[1])+"\n")
    plt.figure(figsize=(20, 8))
```

```
    plt.scatter(X[:,1],tY,s=area, c=colors, alpha=0.9)
    plt.xlabel('Hours studied')
    plt.ylabel('Points on final exam')
    plt.grid(color='k', linestyle='--', linewidth=0.5)
    Y=forwardpass(X,w)
    plt.plot(X[:,1],Y, c='orange')
    plt.show()
```

[ ]:
```
### STUDENT ###
# You can play around with parameters below
###############################################################################################

weights[0]=100
weights[1]=-1.0
showprediction(X,tY,weights)
```

1.1.2 Task 2:

Now that we can compute the prediction for our choice of parameters w, let us compute a norm to evaluate how good a given choice is. Below, implement the cost function for the batch gradient approach.

[ ]:
```
### STUDENT ###
# Compute the cost function for the batch gradient approach
###############################################################################################

def computecost(X,tY,w,nsamples):
    C_loc=0.0

    # YOUR CODE HERE
    raise NotImplementedError()
    return C
```

[ ]:
```
# This cell tests your function "computecost" with the autograder
```

[ ]:
```
# Helper function, evaluate the cost on a parameter grid
###############################################################################################

# split parameter space in 100x100 grid
w0_vals=np.linspace(-10, 10.0, num=100)
w1_vals=np.linspace(-1, 4.0, num=100)
C_vals = np.zeros((w0_vals.size, w1_vals.size));

# compute the costs for all nodes in 100x100 grid
for i in range (0,w0_vals.size):
    for j in range (0,w1_vals.size):
        t = (w0_vals[i] , w1_vals[j]);
```

```
        C_vals[i,j] = computecost(X, tY, t,nsamples);


%matplotlib notebook
fig = plt.figure(figsize=(14,10))
w0_pos, w1_pos = np.meshgrid(w0_vals, w1_vals)
ax = fig.add_subplot(1, 1, 1,projection='3d')
p = ax.plot_surface(w0_pos, w1_pos, C_vals.T, cmap=cm.plasma)
ax.set_xlabel('w0')
ax.set_ylabel('w1')
ax.set_zlabel('Costs C')
plt.show()


# Plot the 2D version of the costs
#############################################################################################

fig = plt.figure(figsize=(8,6))
ax1 = fig.add_subplot(1, 1, 1)
ax1.set_xlabel('w0')
ax1.set_ylabel('w1')
cp=ax1.contour(w0_vals, w1_vals, C_vals.T, levels=[60, 70,75, 80, 90, 100,125,␣
 ↪150,175,250],cmap=cm.plasma,norm = LogNorm())
plt.clabel(cp, inline=1, fontsize=10)
```

1.1.3 Task 3:

In the cell below, compute the mean gradient over all samples as shown in class, and update the parameters w from this. LR corresponds to the learning rate and is a real number >0.

[ ]:
```
### STUDENT ###
def compute_Gradient_and_update_parameters(X,w,tY,nsamples,LR):
    # YOUR CODE HERE
    raise NotImplementedError()
```

[ ]:
```
# This cell tests your function "compute_Gradient_and_update_parameters" with␣
 ↪the autograder
t_weights=np.array([[1.0],[1.0]])
t_y=np.array([[12],[13],[14]])
t_samples=t_y.size
t_X=np.array([[1, 1],[1 ,2],[1 ,3]])
t_LR=120
#compute_Gradient_and_update_parameters(t_X,t_weights,t_y,t_samples,t_LR)
np.testing.assert_array_equal([[1201.0],[2401.
 ↪0]],compute_Gradient_and_update_parameters(t_X,t_weights,t_y,t_samples,t_LR))
```

[ ]:
```
# GradientDescent: Optimize the square error iteratively and plot what is␣
 ↪happening
##########################################################################################
def gradientDescent(X,tY,w,LR,niter):
    # 2D plot of error surface
    fig=plt.figure(figsize=(40, 20))
    fig, ax = plt.subplots(nrows=2, ncols=1,figsize=(15,15))
    ax[0].size=(12,12)
    ax[0].contour(w0_vals, w1_vals, C_vals.T, levels=[60, 70,75, 80, 90,␣
 ↪100,125, 150,175,250],cmap=cm.plasma,norm = LogNorm())
    # plot prediction
    N=nsamples
    area = np.pi*30
    colors = np.random.rand(N)
    ax[1].scatter(X[:,1],tY,s=area, c=colors, alpha=0.9)
    plt.xlabel('Hours studied')
    plt.ylabel('Points on final exam')

    # let us save the cost history
    C_history = np.zeros((niter, 1));

    for n in range(0,niter):

        compute_Gradient_and_update_parameters(X,w,tY,nsamples,LR)

        # save costs for cost history
        C_history[n] = computecost(X, tY, w,nsamples);
        # plot current prediction and contour plot
        if n%100 == 0:
            ax[0].scatter(w[0],w[1])
            ax[1].plot(X[:,1],np.dot(X,w),linewidth=1.0)
            display.display(plt.gcf())
            display.clear_output(wait=True)
    return(C_history)
```

[ ]:
```
### STUDENT ###
# Do the training!
##########################################################################################

%matplotlib inline

# select learning rate, initial weights and iterations.
# You can play around with this!
LR=0.015
weights[0]=-8.0
weights[1]=1.0
n_iter=2000
```

```
# Now do the actual training and observe the plots
Cost_hist=gradientDescent(X,tY,weights,LR,n_iter)

print('The optimized weights are: ',weights[0],weights[1])
print ('The optimized costs are: ',Cost_hist[-1])
# Plot the cost history after training
cost_fig=plt.figure(figsize=(30, 15))
plt.xlabel('Iteration',fontsize=24)
plt.ylabel('Cost C',fontsize=24)
plt.grid(color='k', linestyle='--', linewidth=0.5)
plt.plot(Cost_hist, linestyle='-', linewidth=4.5)
plt.show()
```

The last of the three plots above shows what is called the *cost history*, which is just the costs as a function of the training iteration. For more complex algorithms and training, looking at the cost history can reveal if the learning is successful or if e.g. overfitting is likely.

For this convex cost function, the cost drops rapidly and the algorithm converges in very few steps, given that the learning rate is not too large.

1.1.4 Task 4:

Train the model as best as you can and provide the final weights and cost function value below:

[ ]:
```
### STUDENT ###
# uncomment these lines, copy them below the <begin solution> tag and complete␣
 ↪them
# w0=
# w1=
# cost =
# YOUR CODE HERE
raise NotImplementedError()
```

[ ]:
```
# This cell tests results for w0, w1 and cost
```

We have thus seen how linear regression works in practice. Note all the steps discussed here (defining the model, computing the forward pass, computing the cost function and its derivatives, updating the weights and looping over the iterations) are also present almost any other supervised learning method, in particular in neural networks.

1.1.5 Task 5

Let us now check our results against the solution of the normal form of the linear least squares problem (see notes from class). In the box below, implement the normal equation to compute the optimal parameters directly from the matrices given by the training features X and the vector tY. You can use the numpy function for the inverse, numpy.linalg.inv if you want. Try it for yourself, but in case you run into too much trouble, a great

resource is given here: https://towardsdatascience.com/performing-linear-regression-using-the-normal-equation-6372ed3c57

[ ]:
```
### STUDENT ###
def compute_parameters_from_normal_form(X,tY):
    # YOUR CODE HERE
    raise NotImplementedError()
```

[ ]:
```
t_y=np.array([[12],[13],[14]])
t_X=np.array([[1, 1],[1 ,2],[1 ,3]])
compute_parameters_from_normal_form(t_X,t_y)
np.testing.assert_almost_equal([[11.0],[1.
 ↪0]],compute_parameters_from_normal_form(t_X,t_y))
```

This concludes assignment 4. Let's summarize some important points: - The linear regression model is linear in the parameters, but can deal with an arbitrary number of features - the number of column in X just increases - Optimizing the parameters of the model is done in a supervised learning manner: We learn from the given data points to predict the model response (for all possible inputs) - To define what we mean by optimum, we need a cost function - a norm on the error of the current predictions - This cost function is of course a function of the parameters. Thus, we can compute the gradient - There are many optimization methods, gradient descent is just the most basic one - but it works nicely for convex problems - For the linear regression with square cost function, we can check our results against the normal equation of the linear least squares problem.

Student_0_Readme

April 14, 2021

1 Readme

All exercises are designed in a way that you only need to change / add code after comments starting with "Todo:" . Please don't change other parts as this will break later parts in the notebooks. Additionally, it is strongly advised to use the variable names that are already presented in your "todo" segments as later parts build on these. It is encouraged to play around with the code after you're done with the exercises but please make sure to do this in either new cells or after duplicating the original notebooks to avoid getting stuck with broken code.

1.1 Please make sure to run your code multiple times as random numbers play a key role in this exercise!!!

1.2 The loss function is given in the beginning of the notebooks

[ ]:

Student_1_1plus1_ES

April 14, 2021

1 1+1 Evolution Strategies

Please note, the adaptive 1+1 ES are optional.

[ ]:
```
%matplotlib notebook
from math import sin, cos, sqrt, pi
from matplotlib import cm
import numpy as np
import matplotlib.pyplot as plt
```

[ ]:
```
def loss_function(x, a=10):
    dummy = a * len(x)
    for ii in range(len(x)):
        dummy += x[ii] ** 2 - a * cos(2 * pi * x[ii])
    return dummy
```

[ ]:
```
def plot_loss(ax):

    x = np.linspace(-5, 5, 200)
    y = np.linspace(-5, 5, 200)
    X, Y = np.meshgrid(x, y)

    Z = np.zeros_like(X)
    for ii in range(X.shape[0]):
        for jj in range(X.shape[1]):
            Z[ii][jj] = loss_function([X[ii][jj], Y[ii][jj]])
    img = ax.contour(X, Y, Z, levels=30, cmap=cm.coolwarm)
    plt.colorbar(img, ax=ax)

    return ax
```

[ ]:
```
# 1+1 Evolution Strategy

# Parameters
n_evolutions = 50
init_population = [2.5, 2.8]
sig = 0.6
population = np.random.normal(init_population, sig)
```

```
population_log = [population]

# Parameters for adaption
length_log = 5
log = [0]*length_log
c = 0.817

# Inits for plot
fig = plt.figure()
ax = fig.gca()

axes = plt.gca()
axes.set_xlim([-5, 5])
axes.set_ylim([-5, 5])

plt.ion()
fig.show()
fig.canvas.draw()

plot_loss(ax)


for episode in range(n_evolutions):

    # Todo: Implement 1+1 update strategy

    # Todo: Generate a random sample
    sample = 0

    # Todo: Calculate loss (function) for new sample and old population


    # Todo: Compare loss from the sample to the loss of the old population.
    #       Store in better if the new loss is smaller than the former


    # Todo: Update population
    if better:
        ax.plot(sample[0], sample[1], ".g")
    else:
        ax.plot(sample[0], sample[1], ".b")

    # Additional code here␣
↪=====================================================

    # Todo: Implement the presented update strategy for sigma.
```

```
    #       This exericse comes with further instructions to make it a little␣
 ↪more challenging.
    #       Please feel free to come back to this exercise after completing the␣
 ↪other notebooks.



    #␣
 ↪================================================================================

    # Plot population
    population_log.append(population)
    fig.canvas.draw()

population_log = np.array(population_log)

ax.plot(population_log[:,0], population_log[:,1], "-g", label="Intermediate␣
 ↪updates")
ax.plot(population_log[0,0], population_log[0,1], "xg", label="Start")
ax.plot(population_log[-1,0], population_log[-1,1], "*g", label="End")
plt.legend()
```

[ ]:

Student_2_Naive_ES

April 14, 2021

1 First steps in population based optimization

[ ]:
```
%matplotlib notebook
from math import sin, cos, sqrt, pi
from matplotlib import cm
import numpy as np
import matplotlib.pyplot as plt
```

[ ]:
```
def loss_function(x, a=10):
    dummy = a * len(x)
    for ii in range(len(x)):
        dummy += x[ii] ** 2 - a * cos(2 * pi * x[ii])
    return dummy
```

[ ]:
```
def plot_loss(ax):

    x = np.linspace(-5, 5, 200)
    y = np.linspace(-5, 5, 200)
    X, Y = np.meshgrid(x, y)

    Z = np.zeros_like(X)
    for ii in range(X.shape[0]):
        for jj in range(X.shape[1]):
            Z[ii][jj] = loss_function([X[ii][jj], Y[ii][jj]])
    img = ax.contour(X, Y, Z, levels=30, cmap=cm.coolwarm)
    plt.colorbar(img, ax=ax)

    return ax
```

[ ]:
```
# Simple Evolution Strategy
n_evolutions = 30
n_population = 50
start = [3.5, 3.8]
init_population = [start] * n_population
sig_0 = 0.4
population = np.random.normal(init_population, sig_0)
centers = [start]
```

```
# Inits for plot
fig = plt.figure()
ax = fig.gca()

axes = plt.gca()
axes.set_xlim([-5, 5])
axes.set_ylim([-5, 5])

plt.ion()
fig.show()
fig.canvas.draw()

plot_loss(ax)
fig.canvas.draw()

for episode in range(n_evolutions):
    # Population Update Strategy
    scores = []
    for ii in range(n_population):
        ax.plot(population[ii,0], population[ii,1], ".b")

    # Todo: evaluate the loss for every member of the population and save the␣
 ↪value in score
    for ii in range(n_population):
        scores.append(function(population[ii, :]))

    # Todo: Save the id of the "best" member of the population in best_id
    best_id = np.argmin(scores)

    # Todo: Append the best member of the population to centers
    centers.append(list(population[best_id, 0:2]))

    # Todo: Update the population initializing a new population around the best␣
 ↪member of the former.
    #       Keep sigma constant with respect to the initialization
    tmp_population = [
        [population[best_id, 0], population[best_id, 1]]
    ] * n_population
    population = np.random.normal(tmp_population, sig_0)

    # Plot population
    fig.canvas.draw()

centers = np.array(centers)

plt.plot(centers[:,0], centers[:,1], ".-g", label="Centers")
```

```
plt.plot(centers[0,0], centers[0,1], "or", label="Start")
plt.plot(centers[-1,0], centers[-1,1], "xr", label="End")
plt.legend()
```

[ ]:

Student_3_Eigenvectors

April 14, 2021

1 Eigenvectors

In this notebook you can explore how the covariance matrix, data and eigenvectors are related. Feel free to play around with it!

[ ]:
```
%matplotlib notebook
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse
```

[ ]:
```
def error_ellipse(ax, xc, yc, cov, sigma=1, **kwargs):
    """
    https://github.com/megbedell/plot_tools/blob/master/error_ellipse.py
    Plot an error ellipse contour over your data.
    Inputs:
    ax : matplotlib Axes() object
    xc : x-coordinate of ellipse center
    yc : x-coordinate of ellipse center
    cov : covariance matrix
    sigma : # sigma to plot (default 1)
    additional kwargs passed to matplotlib.patches.Ellipse()
    """
    w, v = np.linalg.eigh(cov)  # assumes symmetric matrix
    order = w.argsort()[::-1]
    w, v = w[order], v[:, order]
    theta = np.degrees(np.arctan2(*v[:, 0][::-1]))  # * unpacks argument␣
↪instead of [0]
    ellipse = Ellipse(
        xy=(xc, yc),
        width=2.0 * sigma * np.sqrt(w[0]),
        height=2.0 * sigma * np.sqrt(w[1]),
        angle=theta,
        **kwargs
    )
    ellipse.set_facecolor("none")
    ax.add_artist(ellipse)

    return ax
```

[ ]:
```
n_population = 200
cov = np.array([[1, 0.5], [0.5, 1]])
mean = np.array([0, 0])
population = np.random.multivariate_normal(mean, cov, n_population)

w, v = np.linalg.eig(cov)
order = w.argsort()[::-1]
w, v = w[order], v[:, order]

fig, ax = plt.subplots(1, 1)

for ii in range(population.shape[0]):
    ax.plot(population[ii, 0], population[ii, 1], ".b")

ax = error_ellipse(
    ax, mean[0], mean[1], cov, ec="green", sigma=3, zorder=9999, label="3␣
 ↪$\sigma$"
)
ax = error_ellipse(
    ax, mean[0], mean[1], cov, ec="black", sigma=2, zorder=9999, label="2␣
 ↪$\sigma$"
)
ax = error_ellipse(
    ax, mean[0], mean[1], cov, ec="red", sigma=1, zorder=9999, label="1␣
 ↪$\sigma$"
)

# Eigenvalues
ax.arrow(
    mean[0],
    mean[1],
    np.sqrt(w[0]) * v[0, 0],
    np.sqrt(w[0]) * v[1, 0],
    width=0.1,
    color="red",
    length_includes_head=True,
)
ax.arrow(
    mean[0],
    mean[1],
    np.sqrt(w[1]) * v[0, 1],
    np.sqrt(w[1]) * v[1, 1],
    width=0.1,
    color="green",
    length_includes_head=True,
)
```

```
ax.set_aspect("equal", "box")
ax.set_xlim([-5, 5])
ax.set_ylim([-5, 5])
ax.set_xlabel("x")
ax.set_ylabel("y")

plt.show()
```

[ ]:

[ ]:

Student_4_CMA_ES

April 14, 2021

1 Covariance Matrix Adaptive ES

Note this is a simplified version of the original algorithm. Make sure to check the original paper before applying this optimizer to a real-world problem!

[ ]:
```
%matplotlib notebook
from math import sin, cos, sqrt, pi
from matplotlib import cm
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse
import matplotlib.transforms as transforms
```

[ ]:
```
def loss_function(x, a=10):
    dummy = a * len(x)
    for ii in range(len(x)):
        dummy += x[ii] ** 2 - a * cos(2 * pi * x[ii])
    return dummy
```

[ ]:
```
def plot_loss(ax):

    x = np.linspace(-5, 5, 200)
    y = np.linspace(-5, 5, 200)
    X, Y = np.meshgrid(x, y)

    Z = np.zeros_like(X)
    for ii in range(X.shape[0]):
        for jj in range(X.shape[1]):
            Z[ii][jj] = loss_function([X[ii][jj], Y[ii][jj]])
    img = ax.contour(X, Y, Z, levels=30, cmap=cm.coolwarm)
    plt.colorbar(img, ax=ax)

    return ax
```

[ ]:
```
def error_ellipse(ax, xc, yc, cov, sigma=3, **kwargs):
    """
    https://github.com/megbedell/plot_tools/blob/master/error_ellipse.py
    Plot an error ellipse contour over your data.
```

```
    Inputs:
    ax : matplotlib Axes() object
    xc : x-coordinate of ellipse center
    yc : x-coordinate of ellipse center
    cov : covariance matrix
    sigma : # sigma to plot (default 1)
    additional kwargs passed to matplotlib.patches.Ellipse()
    """
    w, v = np.linalg.eigh(cov)  # assumes symmetric matrix
    order = w.argsort()[::-1]
    w, v = w[order], v[:, order]
    theta = np.degrees(np.arctan2(*v[:, 0][::-1]))  # * unpacks argument␣
↪instead of [0]
    ellipse = Ellipse(
        xy=(xc, yc),
        width=2.0 * sigma * np.sqrt(w[0]),
        height=2.0 * sigma * np.sqrt(w[1]),
        angle=theta,
        **kwargs
    )
    ellipse.set_facecolor("none")
    ax.add_artist(ellipse)

    return ax
```

[ ]:
```
def covar_helper(x1, mu_x1, x2, mu_x2):
    if len(x1) != len(x2):
        raise Exception

    tmp = 0
    for ii in range(len(x1)):
        tmp += (x1[ii] - mu_x1) * (x2[ii] - mu_x2)

    return tmp / len(x1)
```

[ ]:
```
 # Simplified CMA ES
n_evolutions = 20
n_population = 100
best_perc = 0.25
start = [3.5, 3.5]
init_population = [start] * n_population
sig_0 = 0.65
population = np.random.normal(init_population, sig_0)

centers = [start]

# Inits for plot
```

```
fig = plt.figure()
ax = fig.gca()

axes = plt.gca()
axes.set_xlim([-5, 5])
axes.set_ylim([-5, 5])

plt.ion()
fig.show()
fig.canvas.draw()

plot_loss(ax)

for episode in range(n_evolutions):

    if episode > 0:
        # confidence_ellipse(cov, mean_old, ax)
        ax = error_ellipse(
            ax, mean_old[0], mean_old[1], cov, ec="green", zorder=9999
        )

    # Population Update Strategy

    # Todo: store the loss of each particle in a list named scores


    # Todo: Find the best best_perc percent particles with respect to their␣
↪loss.
    #       Store these particles in a numpy array named best_population


    # Todo: Find the mean of the best population and save it as mean


    centers.append(list(mean))

    if episode == 0:
        mean_old = mean

    sig_xx = covar_helper(
        best_population[:, 0], mean_old[0], best_population[:, 0], mean_old[0]
    )
    sig_xy = covar_helper(
        best_population[:, 0], mean_old[0], best_population[:, 1], mean_old[1]
    )
    sig_yy = covar_helper(
        best_population[:, 1], mean_old[1], best_population[:, 1], mean_old[1]
```

```
    )

    # Todo: build a covariance matrix with the entries above and draw a new␣
↪population around the mean calculated above.


    mean_old = mean


    # Plot population
    fig.canvas.draw()

centers = np.array(centers)

plt.plot(centers[:,0], centers[:,1], ".-g", label="Centers")
plt.plot(centers[0,0], centers[0,1], "or", label="Start")
plt.plot(centers[-1,0], centers[-1,1], "xr", label="End")
plt.legend()
```

[ ]:

Student_5_Future_Work

April 14, 2021

1 Next steps

After completing the exercises you could work on a subset of the following questions but feel free to explore other questions if they seem interesting to you: * Try out different loss functions * Look up "momentum" and add it to the motion of the means for ES and CMA-ES * Add discontinouities to a loss function (for example with an if statement -> if x[0]>3: …) and observe how the optimizers handle it

[ ]:

6-1_Adam

April 14, 2021

1 Adam

In this exercise you'll implement Adam from scratch by adding code to the provided functions. All places that need adjustments are started with a comment and todo: giving instructions.

[ ]:
```
%matplotlib notebook
import matplotlib.pyplot as plt
from matplotlib import cm
import matplotlib
import numpy as np

# Parameters
x_min = -5
x_max = 5
y_min = -5
y_max = 5
```

We are going to start by defining a loss function

[ ]:
```
def function(x):
    return (x[0] ** 2 + x[1] - 11) ** 2 + (x[0] + x[1] ** 2 - 7) ** 2


# Todo:
# fill function grad - The gradient of function.
# Note, the current function body is just there to illustrate which output is␣
 ↪expected for futher code.
# As for the lecture the gradients *-1 are plotted over the contour plot of the␣
 ↪loss landscape in the next cell.
# You can use this to "test" your grad function

def grad(x):
    return [0,0]
```

[ ]:
```
# Plotting
fig = plt.figure()
ax = fig.gca()
```

```
x = np.linspace(x_min, x_max, 100)
y = np.linspace(y_min, y_max, 100)
X, Y = np.meshgrid(x, y)
Z = function(x=[X, Y])
levels = np.logspace(-1.5, 4, 30)
cset = ax.contour(
    X, Y, Z, levels=levels, cmap=cm.coolwarm, norm=matplotlib.colors.LogNorm()
)
plt.colorbar(cset)
ax.set_xlabel("$x_0$")
ax.set_xlim(x_min, x_max)
ax.set_ylabel("$x_1$")
ax.set_ylim(y_min, y_max)

x_quiver = np.linspace(x_min, x_max, 15)
y_quiver = np.linspace(y_min, y_max, 15)
X_quiver, Y_quiver = np.meshgrid(x_quiver, y_quiver)
U = np.zeros(shape=X_quiver.shape)
V = np.zeros(shape=X_quiver.shape)

for ii in range(U.shape[0]):
    for jj in range(U.shape[1]):
        U[ii, jj] = -1 * grad([X_quiver[ii, jj], Y_quiver[ii, jj]])[0]
        V[ii, jj] = -1 * grad([X_quiver[ii, jj], Y_quiver[ii, jj]])[1]

q = ax.quiver(X_quiver, Y_quiver, U, V)


plt.show()
```

1.1 Implementing the optimizer

[ ]:
```
class Adam:
    def __init__(self, theta_0, grad_fc, beta1=0.9, beta2=0.999, alpha=0.1):
        self.theta = theta_0

        self.beta1 = beta1
        self.beta2 = beta2
        self.alpha = alpha
        self.epsilon = 1e-8

        self.m_t_1 = np.zeros([1, len(theta_0)])
        self.v_t_1 = np.zeros([1, len(theta_0)])
        self.t = 0

        self.grad = grad_fc
```

```
    def step(self):
        # Todo: implement the gradient step for Adam use the class attributes
        # Note that you can use the two cells below to run your optimizer as␣
↪long as you don't change the interfaces

        return self.theta
```

[ ]:
```
theta_0 = np.random.uniform([-2, -2], [[1, 1]]).squeeze()
theta_log = [list(theta_0)]
optimizier = Adam(theta_0, grad, alpha=0.1)

for ii in range(100):
    theta = optimizier.step()
    theta_log.append(list(theta))
theta_log = np.array(theta_log)
```

[ ]:
```
# Plotting
fig = plt.figure()
ax = fig.gca()

x = np.linspace(x_min, x_max, 100)
y = np.linspace(y_min, y_max, 100)
X, Y = np.meshgrid(x, y)
Z = function(x=[X, Y])
levels = np.logspace(-1.5, 4, 30)
cset = ax.contour(
    X, Y, Z, levels=levels, cmap=cm.coolwarm, norm=matplotlib.colors.LogNorm()
)
plt.colorbar(cset)

ax.set_xlabel("$x_0$")
ax.set_xlim(x_min, x_max)
ax.set_ylabel("$x_1$")
ax.set_ylim(y_min, y_max)

ax.plot(theta_log[:, 0], theta_log[:, 1], "k.-", label="Adam")
plt.legend()
plt.show()
```

1.2 Ideas for next steps

If you're done with this exercise you might go ahead and try out extensions or additions like: * Implement SGD and momentum and compare them with Adam * Explore the trade off between learning rate and momentum factors * Implement a different loss function and respective gradient * Fix the inital guess and tune your hyperparamters parameters to reach a local optimum with as few steps as possible

6-1_Adam

April 14, 2021

1 Adam

In this exercise you'll implement Adam from scratch by adding code to the provided functions. All places that need adjustments are started with a comment and todo: giving instructions.

[ ]:
```
%matplotlib notebook
import matplotlib.pyplot as plt
from matplotlib import cm
import matplotlib
import numpy as np

# Parameters
x_min = -5
x_max = 5
y_min = -5
y_max = 5
```

We are going to start by defining a loss function

[ ]:
```
def function(x):
    return (x[0] ** 2 + x[1] - 11) ** 2 + (x[0] + x[1] ** 2 - 7) ** 2


# Todo:
# fill function grad - The gradient of function.
# Note, the current function body is just there to illustrate which output is␣
 ↪expected for futher code.
# As for the lecture the gradients *-1 are plotted over the contour plot of the␣
 ↪loss landscape in the next cell.
# You can use this to "test" your grad function

def grad(x):
    return [0,0]
```

[ ]:
```
# Plotting
fig = plt.figure()
ax = fig.gca()
```

```
x = np.linspace(x_min, x_max, 100)
y = np.linspace(y_min, y_max, 100)
X, Y = np.meshgrid(x, y)
Z = function(x=[X, Y])
levels = np.logspace(-1.5, 4, 30)
cset = ax.contour(
    X, Y, Z, levels=levels, cmap=cm.coolwarm, norm=matplotlib.colors.LogNorm()
)
plt.colorbar(cset)
ax.set_xlabel("$x_0$")
ax.set_xlim(x_min, x_max)
ax.set_ylabel("$x_1$")
ax.set_ylim(y_min, y_max)

x_quiver = np.linspace(x_min, x_max, 15)
y_quiver = np.linspace(y_min, y_max, 15)
X_quiver, Y_quiver = np.meshgrid(x_quiver, y_quiver)
U = np.zeros(shape=X_quiver.shape)
V = np.zeros(shape=X_quiver.shape)

for ii in range(U.shape[0]):
    for jj in range(U.shape[1]):
        U[ii, jj] = -1 * grad([X_quiver[ii, jj], Y_quiver[ii, jj]])[0]
        V[ii, jj] = -1 * grad([X_quiver[ii, jj], Y_quiver[ii, jj]])[1]

q = ax.quiver(X_quiver, Y_quiver, U, V)


plt.show()
```

1.1 Implementing the optimizer

[ ]:
```
class Adam:
    def __init__(self, theta_0, grad_fc, beta1=0.9, beta2=0.999, alpha=0.1):
        self.theta = theta_0

        self.beta1 = beta1
        self.beta2 = beta2
        self.alpha = alpha
        self.epsilon = 1e-8

        self.m_t_1 = np.zeros([1, len(theta_0)])
        self.v_t_1 = np.zeros([1, len(theta_0)])
        self.t = 0

        self.grad = grad_fc
```

```
    def step(self):
        # Todo: implement the gradient step for Adam use the class attributes
        # Note that you can use the two cells below to run your optimizer as␣
↪long as you don't change the interfaces

        return self.theta
```

[ ]:
```
theta_0 = np.random.uniform([-2, -2], [[1, 1]]).squeeze()
theta_log = [list(theta_0)]
optimizier = Adam(theta_0, grad, alpha=0.1)

for ii in range(100):
    theta = optimizier.step()
    theta_log.append(list(theta))
theta_log = np.array(theta_log)
```

[ ]:
```
# Plotting
fig = plt.figure()
ax = fig.gca()

x = np.linspace(x_min, x_max, 100)
y = np.linspace(y_min, y_max, 100)
X, Y = np.meshgrid(x, y)
Z = function(x=[X, Y])
levels = np.logspace(-1.5, 4, 30)
cset = ax.contour(
    X, Y, Z, levels=levels, cmap=cm.coolwarm, norm=matplotlib.colors.LogNorm()
)
plt.colorbar(cset)

ax.set_xlabel("$x_0$")
ax.set_xlim(x_min, x_max)
ax.set_ylabel("$x_1$")
ax.set_ylim(y_min, y_max)

ax.plot(theta_log[:, 0], theta_log[:, 1], "k.-", label="Adam")
plt.legend()
plt.show()
```

1.2 Ideas for next steps

If you're done with this exercise you might go ahead and try out extensions or additions like: * Implement SGD and momentum and compare them with Adam * Explore the trade off between learning rate and momentum factors * Implement a different loss function and respective gradient * Fix the inital guess and tune your hyperparamters parameters to reach a local optimum with as few steps as possible

assignment08

April 14, 2021

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
```

[ ]:
```
# Hidden parts create data x and y


x = np.linspace(-3,3, 30).reshape(-1, 1)
np.random.seed(42)
y_clean= x * -0.8545 + x**2*0.35+ x**3*0.25
y = y_clean + np.random.normal(np.zeros_like(x),1)


plt.plot(x, y, "x")
plt.xlabel("x")
plt.ylabel("y")
```

0.1 Regularisation - Ridge Regression

1. Create a polynomial input X_poly of degree 10 (without a bias term) from the sampling locations x
2. For alpha = 0, 1, 5, 10 fit a Ridge Regressor to X_poly and y
3. Plot the predictions of the respective models over x and create a legend to distinguish them

You can check X_poly in the cell below.

[ ]:
```
from sklearn.linear_model import Ridge

plt.plot(x, y, "x")
plt.xlabel("x")
plt.ylabel("y")


# YOUR CODE HERE
raise NotImplementedError()
```

[ ]:
```
# Hidden part checks X_poly to avoid simple mistakes


if is_correct
    print("X_poly is correct!")
else:
    print("X_poly is not correct!")
```

0.2 Regularisation - Decision Tree

Tune the following parameters by hand to get a feel for their effect: * max_depth: Max depth of the tree * min_samples_split: The minimum number of samples required to split an internal node * min_samples_leaf: The minimum number of samples required to be at a leaf node.

The goal is to get a reasonable regressor. The original settings are below if you want to reset your try:

```
regressor = DecisionTreeRegressor(max_depth=None, min_samples_split=2, min_samples_leaf=1, rand
```

[ ]:
```
from sklearn.tree import DecisionTreeRegressor
plt.plot(x, y, "x")
plt.xlabel("x")
plt.ylabel("y")
regressor = DecisionTreeRegressor(max_depth=None, min_samples_split=2,␣
 ↪min_samples_leaf=1, random_state=0)
regressor.fit(x,y)
y_pred = regressor.predict(x)
plt.plot(x, y_pred, label="alpha=")

# YOUR CODE HERE
raise NotImplementedError()
```

0.3 Cross Validation

To get to know cross validation you should see it in comparison to "traditional" train / test splitting. Therefore, implement the following steps:

- For a random seed from 1 to 5 split the X_poly and y into a train and test set, with a test_size of 0.33 and respective random_state. Please use sklearn's train_test_split for this": X_train, X_test, y_train, y_test = train_test_split(X_poly, y, test_size=0.33, random_state=seed). Please use X_poly from the last exercise.
- For each random seed train a ridge regressor with an alpha of your choice on the training set and calculate and print its MSE with respect to the test set
- Plot the predictions of the model for the whole range of x (not just X_train or X_test)

[ ]:
```
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
from sklearn.metrics import make_scorer
from sklearn.utils import shuffle
```

```
scorer = make_scorer(mean_squared_error)

plt.plot(x, y, "x")
plt.xlabel("x")
plt.ylabel("y")


# YOUR CODE HERE
raise NotImplementedError()

model = Ridge(alpha=10)
X_poly_, y_ = shuffle(X_poly, y)
scores = cross_val_score(model, X_poly_, y_, cv=10, scoring=scorer)
print("\n")
print(scores.mean())
```

0.4 Toy Problem

In this last part of today's assignment you can try out everything you've learned so far. In the next hidden cell a dataset is created that you should analyse and build a model for. Please use standard sklearn functions as the evaluation cell uses model.predict(test_data) to calculate the root mean square error.

Your goal is to build a model with a RMSE below 4, any model that can be evaluated with model.predict is fair game! Feel free to try different models, hyperparameter search, cross validation and other techniques to solve this exercise!

[ ]:
```
# Hidden code generates X_train and y_train

from sklearn.datasets import load_boston
X, y = load_boston(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,␣
 ↪random_state=711)
y_test = 5
X_test = 5
```

[ ]:
```
# YOUR CODE HERE
raise NotImplementedError()
```

[ ]:
```
# Hidden code uses model on test set and returns the RMSE.


print(rmse)
assert rmse < 4
```

[ ]:
```

```

Assignment_10

April 14, 2021

0.1 REMEMBER: even if a cell is locked / read only, you can always copy the contents to a new cell to modify and execute! Cells are locked here to provide you with a safe fallback, not to stop you from exploring!

0.2 Multilayer-Perceptron for MNIST

In this notebook, we will program a MLP with 1 hidden layer and apply it to the famous MNIST problem. MNIST is a freely available dataset, see e.g. http://yann.lecun.com/exdb/mnist/. It is so popular and often used that is included in the general ML frameworks like tensorflow, pytorch etc.

Here, we use the version provided by Kaggle https://www.kaggle.com/oddrationale/mnist-in-csv, as it gives the data in easily manageable CSV format.

The idea for this notebook follows the books "Make Your Own Neural Network" by T. Rashid and "Neural Networks from Scratch in Python" by Harrison Kinsley & Daniel Kukieła

0.2.1 The MNIST data

This data contains 28x28 pixel wide gray scale images of handwritten digits 0-9. There are 60k training samples, and 10k test samples. The data is arranged in rows, each row consists of 785 values: the first value is the label (0 to 9) and the remaining 784 values are the color pixel values (0 to 255). You are free to use the full set for training (set downsample =1.0 in the next cell), or stick with a reduced set. Execute the next cell to load the training and test data, select a random subset of both and plot a sample image.

Note that the label is stored as the first entry of the row, i.e. it is in sample[0], while the rest of the row are the 28x28 pixel values sample[1:]

[ ]:
```
import numpy as np
import array as arr
import matplotlib.pyplot as plt
from IPython import display
import random
%matplotlib inline

# Load the MNIST data sets, containing 60k training and 10k test data sets.
 ↪Each row contains a single
# 28x28 pixel grayscale image (0,255), with the first entry of the row being
 ↪the label
```

```
training_data_file = open("./mnist_dataset/mnist_train_60k.csv", 'r')
test_data_file = open("./mnist_dataset/mnist_test_10k.csv", 'r')

training_data_list = training_data_file.readlines()
test_data_list = test_data_file.readlines()

training_data_file.close()
test_data_file.close()

# select ratio of data to actually use in training. this of course influences␣
 ↪the training results and speed
# for downsample=0.1, we thus have 1k test data and 6k training data samples
downsample=0.1
test_data_list = random.
 ↪sample(test_data_list,int(len(test_data_list)*downsample))
training_data_list = random.
 ↪sample(training_data_list,int(len(training_data_list)*downsample))

# visualize the first [0] image from the training set and print its label
sample = training_data_list[0].split(',')
image_array = np.asfarray(sample[1:]).reshape((28,28))
plt.imshow(image_array, cmap='Greys',interpolation='None')
print ("True label = ",sample[0])
```

[ ]:
```
# Helper functions, only change them if you have ruled out any other source of␣
 ↪problems
# Sigmoid activations can run into numerical problems for their limits of 0 and␣
 ↪1. Thus, one can avoid these
# by scaling inputs and outputs.

# Scale pixel values to 0-1
def scale_inputs(image_in):
    #return (np.asfarray(image_in)/255.0 * 0.99) + 0.01
    return np.asfarray(image_in)/255.0

# one-hot encode label vectors
def mod_one_hot(labels_in):
    #labels_out = np.zeros(neurons_per_layer[2]) + 0.01
    #labels_out[int(labels_in[0])] = 0.99

    labels_out = np.zeros(neurons_per_layer[2]) + 0.0
    labels_out[int(labels_in[0])] = 1.0
    return labels_out
```

0.2.2 The neural network

We will now build an MLP to recognize the digits in MNIST and later see how it does on your own handwriting. For this, we will build a simple MLP with one hidden layer. We will feed the network with one sample at a time, and update the weights after each sample.

Since each sample has 28x28=784 pixels, these will be the input neurons. As outputs, we have a vector of length 10, each entry corresponding to the likelihood of the digit as a label. In other words, the true labels from the training data set look for example like this: [0,0,0,0,0,0,0,0,1,0]. This would represent the label "8" (0-9). The network outputs y_hat will have the same shape, but will contain numbers between 0 and 1 - the network is approximate, and will never return a label with absolute certainty. The final output of the network will then be chosen as the digits with the highest likelihood, e.g. [0.1,0.2,0.3,0.1,0.2,0.1,0.4,0,0.8,0.1] would produce a network prediction of "8".

Thus, the input neurons are fixed at 784, the output neurons are fixed at 10, and we are free to choose the number of neurons in the hidden layer. As usual, there will be no activation applied to the input, but we will use a *sigmoid* on the hidden and output layers.

[ ]:
```
# Helper functions:
# Sigmoid Activation Function:
def sigmoid(z):
    # YOUR CODE HERE
    raise NotImplementedError()

# From the lecture, we know that a=sig(z), and dsig/dz=a(1-a). Implement this␣
↪last equation here
def sigmoidgrad(a):
    # YOUR CODE HERE
    raise NotImplementedError()
```

[ ]:
```
assert np.isclose(sigmoid(0), 0.5)
assert sigmoid(1) == 0.5*np.tanh(0.5)+0.5
```

[ ]:
```
assert sigmoidgrad(0) == 0
assert sigmoidgrad(1) == 0
assert np.isclose(sigmoidgrad((1+np.sqrt(5))/2), -1)
```

[ ]:
```

# the class for an MLP with 1 hidden layer
class MLP1H:
    # neuronsperlayer is a vector with 3 entries, alpha is the learning rate
    def __init__(self,neuronsperlayer,alpha):
        self.n=neuronsperlayer
        self.n[0]=neuronsperlayer[0]
        self.n[1]=neuronsperlayer[1]
        self.n[2]=neuronsperlayer[2]
        self.alpha=alpha
```

```
        # initialize activations
        self.a_1=np.zeros((self.n[0],1))
        self.a_2=np.zeros((self.n[1],1))
        self.a_3=np.zeros((self.n[2],1))

        # initialize outputs. It is not strictly necessary to make z self here, as it is locally used only
        self.z_1=np.zeros((self.n[0],1))
        self.z_2=np.zeros((self.n[1],1))
        self.z_3=np.zeros((self.n[2],1))

        # initialize the weight matrices between the input - hidden and hidden - ouput layers
        # and fill them with random values, either normally or uniformly distributed
        # YOUR CODE HERE
        raise NotImplementedError()

        self.activation=sigmoid
        return

    def sse_cost(self,pred,truth):
        # define the summed squared error cost function for a single sample here, including the 0.5
        # YOUR CODE HERE
        raise NotImplementedError()
        return cost

    def forwardpass(self,inputs):
        # We convert the inputs from (784,) to a  true vector (784,1)
        X = np.array(inputs, ndmin=2).T
        # Implement the forward pass
        # YOUR CODE HERE
        raise NotImplementedError()
        return Yhat

    def train(self, inputs, targets):
        # Making sure the targets have the right shape
        tY = np.array(targets, ndmin=2).T
        # Implement the training:
        # first, pass the current sample through the net to fill a,z, etc.
        # next, compute the layer erros \delta, remember the direction
        # then, compute the cost function gradients
        # for tips, see accompanying lecture and notes

        # YOUR CODE HERE
        raise NotImplementedError()
```

```
        # Update the weights
        self.w23  = self.w23-self.alpha * dCd2
        self.w12  = self.w12-self.alpha * dCd1

        cost = self.sse_cost(Y_hat,tY)
        return cost
```

[ ]:
```
# We set up a small ANN here to test the implementation

neurons_per_layer=np.array([2,3,4])
learningrate=0.15
testANN = MLP1H(neurons_per_layer,learningrate)

#testing the cost function
assert testANN.sse_cost(1, 1) == 0
assert testANN.sse_cost(8.123, 12.123) == 8
assert testANN.sse_cost(np.array([2, 3]),np.array([3, 2])) == 1


#testing the correct shapes of w12, w23

# now testing the forward pass


# NOTE: There are no explicit tests for the backpropagation part. If the error␣
 ↪drops in training, that
# typically is a good sign that it works as planned. With this settings as␣
 ↪provided, a performance_on_test
# (see "Performance" section below) should be >92%.
# When debugging the backprop, think about the shapes of the objects, the␣
 ↪possible matrix operations etc.
# As a last resort, work it out by hand - for a much smaller network and a␣
 ↪single layer
```

0.2.3 Generating the Network

[ ]:
```
# now initiate the ANN with the given parameters
neurons_per_layer=np.array([784,100,10])
learningrate=0.15

ANN = MLP1H(neurons_per_layer,learningrate)
```

0.2.4 Training the ANN

[ ]:
```
# Before training the network, let us first pass an input to the untrained␣
 ↪network and see what happens
# The prediction will likely be wrong, but we can test the forward pass in␣
 ↪principal this way
sample = training_data_list[0].split(',')
image_array = np.asfarray(sample[1:]).reshape((28,28))
plt.imshow(image_array, cmap='Greys',interpolation='None')
print ("True label = ",sample[0])
inputs = scale_inputs(sample[1:])
outputs = ANN.forwardpass(inputs)
print("Predicted label = ",np.argmax(outputs))
```

[ ]:
```
# Training the network

nEpochs = 10

# Initializing the cost history plot
fig=plt.figure(figsize=(80, 40))
fig, ax = plt.subplots(figsize=(20, 10))
ax.size=(12,12)
plt.xlabel('Epochs',fontsize=24)
plt.ylabel('Costs C',fontsize=24)

# loop over epochs, remember, we train and update with single samples
for i in range(1,nEpochs+1):
    lcost=0.
    localcost=0.
    # loop over all training samples
    for record in training_data_list:
        data_list = record.split(',')
        # scale and encode inputs / labels
        inputs = scale_inputs(data_list [1:])
        labels= mod_one_hot(data_list [0:])
        # train and compute cost for each sample
        localcost=ANN.train(inputs,labels)
        lcost=lcost+localcost

    # Compute the training and test costs after each epoch
    Cost_Training = 0.
    Cost_Test     = 0.
    for record1 in training_data_list:
        # evaluate current model on the full training set and compute cost
        data_list  = record1.split(',')
        inputs = scale_inputs(data_list[1:])
        labels= mod_one_hot(data_list[0:])
```

```
        labels= np.array(labels,ndmin=2)
        y_hat=ANN.forwardpass(inputs)
        Cost_Training = Cost_Training + 0.5*np.sum(np.square(y_hat-labels.T))

    for record2 in test_data_list:
        # evaluate current model on the full test set and compute cost
        data_list= record2.split(',')
        inputs = scale_inputs(data_list[1:])
        labels= mod_one_hot(data_list[0:])
        labels= np.array(labels,ndmin=2)
        y_hat=ANN.forwardpass(inputs)
        Cost_Test = Cost_Test + 0.5*np.dot(y_hat.T-labels,(y_hat.T-labels).T)

    # normalize errors
    Cost_Training = Cost_Training / len(training_data_list)
    lcost = lcost / len(training_data_list)
    Cost_Test     = Cost_Test     / len(test_data_list)

    ax.scatter(i,Cost_Training,color='g',label='Training Data (epoch)')
    ax.scatter(i,lcost,color='k',label='Training Data (sample)')
    ax.scatter(i,Cost_Test,color='r',label='Test Data (epoch)')

    if i==1:
        ax.legend()
    display.display(plt.gcf())
    display.clear_output(wait=True)
print("training complete!")
print("Cost on test data (epoch): ",Cost_Test)
print("Cost on training data (epoch): ",Cost_Training)
print("Cost on training data (sample): ",lcost)
```

0.2.5 Querying the trained network

[ ]:
```
# let us now see if the training brought progress
sample = test_data_list[0].split(',')
image_array = np.asfarray(sample[1:]).reshape((28,28))
plt.imshow(image_array, cmap='Greys',interpolation='None')
print ("True label = ",sample[0])
inputs = scale_inputs(sample[1:])
outputs = ANN.forwardpass(inputs)
print("Predicted label = ",np.argmax(outputs))
```

0.2.6 Performance

[ ]:
```
# evaluate the network perfomance by computing the ratio of correct predictions␣
 ↪on the full test set
test_pred=np.zeros(len(test_data_list))
performance_on_test = 0
j=0
for record in test_data_list:
    data_list= record.split(',')
    inputs = scale_inputs(data_list[1:])
    labels= mod_one_hot(data_list[0:])
    correct_label = np.argmax(labels)
    y_hat=ANN.forwardpass(inputs)
    pred_label = np.argmax(y_hat)
    test_pred[j]=np.argmax(y_hat)
    j+=1
    if (pred_label == correct_label):
        performance_on_test+=1
print ("Performance = ", (performance_on_test / len(test_data_list))*100,"%")
```

[ ]:
```
# Here are random samples from the test data, and the network prediction

from random import randint

fig, ax = plt.subplots(10,10,figsize=(10,10))
ax = ax.flatten()

for i in range(100):
    r = random.randint(0, len(test_data_list))
    sample= test_data_list[r].split(',')
    image= np.asfarray(sample[1:]).reshape((28,28))
    ax[i].imshow(image, cmap='gray_r')
    ax[i].set_title(int(test_pred[r]), fontsize=15)
    ax[i].axis('off')

plt.tight_layout()
plt.show()
```

0.2.7 Your own handwriting

Check the networks prediction on your own handwriting.Either write on a piece of paper and take a picture, or use e.g. gimp to directly create a sample image. Save as png, and store in the folder indicated below. On linux, you can convert e.g. a jpg with 'convert Name.jpg Name.png'

[ ]:
```
from skimage.transform import resize
import matplotlib.image as mpimg
from skimage.color import rgb2gray
```

```
# place own image in the folder, adjust name
own_image=mpimg.imread("./mnist_dataset/own/drei.png")
plt.imshow(own_image)
own_image = rgb2gray(own_image)
own_image = resize(own_image, (28,28))

own_image = 255 - own_image.reshape(784)

inputs = scale_inputs(own_image )
outputs = ANN.forwardpass(inputs)
print("Predicted label = ",np.argmax(outputs))
```

Assignment_10

April 14, 2021

0.1 REMEMBER: even if a cell is locked / read only, you can always copy the contents to a new cell to modify and execute! Cells are locked here to provide you with a safe fallback, not to stop you from exploring!

0.2 Multilayer-Perceptron for MNIST

In this notebook, we will program a MLP with 1 hidden layer and apply it to the famous MNIST problem. MNIST is a freely available dataset, see e.g. http://yann.lecun.com/exdb/mnist/. It is so popular and often used that is included in the general ML frameworks like tensorflow, pytorch etc.

Here, we use the version provided by Kaggle https://www.kaggle.com/oddrationale/mnist-in-csv, as it gives the data in easily manageable CSV format.

The idea for this notebook follows the books "Make Your Own Neural Network" by T. Rashid and "Neural Networks from Scratch in Python" by Harrison Kinsley & Daniel Kukieła

0.2.1 The MNIST data

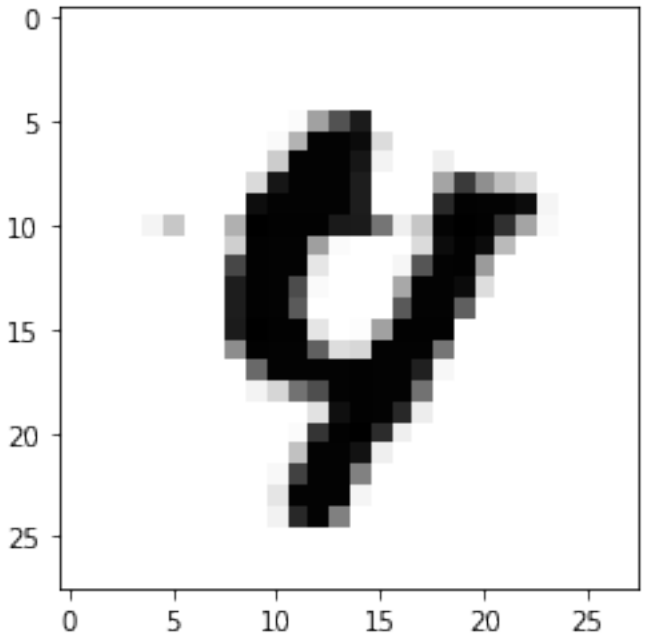
This data contains 28x28 pixel wide gray scale images of handwritten digits 0-9. There are 60k training samples, and 10k test samples. The data is arranged in rows, each row consists of 785 values: the first value is the label (0 to 9) and the remaining 784 values are the color pixel values (0 to 255). You are free to use the full set for training (set downsample =1.0 in the next cell), or stick with a reduced set. Execute the next cell to load the training and test data, select a random subset of both and plot a sample image.

Note that the label is stored as the first entry of the row, i.e. it is in sample[0], while the rest of the row are the 28x28 pixel values sample[1:]

[1]:
```
import numpy as np
import array as arr
import matplotlib.pyplot as plt
from IPython import display
import random
%matplotlib inline

# Load the MNIST data sets, containing 60k training and 10k test data sets.␣
 ↪Each row contains a single
# 28x28 pixel grayscale image (0,255), with the first entry of the row being␣
 ↪the label
```

```
training_data_file = open("./mnist_dataset/mnist_train_60k.csv", 'r')
test_data_file = open("./mnist_dataset/mnist_test_10k.csv", 'r')

training_data_list = training_data_file.readlines()
test_data_list = test_data_file.readlines()

training_data_file.close()
test_data_file.close()

# select ratio of data to actually use in training. this of course influences␣
 ↪the training results and speed
# for downsample=0.1, we thus have 1k test data and 6k training data samples
downsample=0.1
test_data_list = random.
 ↪sample(test_data_list,int(len(test_data_list)*downsample))
training_data_list = random.
 ↪sample(training_data_list,int(len(training_data_list)*downsample))

# visualize the first [0] image from the training set and print its label
sample = training_data_list[0].split(',')
image_array = np.asfarray(sample[1:]).reshape((28,28))
plt.imshow(image_array, cmap='Greys',interpolation='None')
print ("True label = ",sample[0])
```

```
True label =  4
```



[2]:
```
# Helper functions, only change them if you have ruled out any other source of␣
↪problems
# Sigmoid activations can run into numerical problems for their limits of 0 and␣
↪1. Thus, one can avoid these
# by scaling inputs and outputs.

# Scale pixel values to 0-1
def scale_inputs(image_in):
    #return (np.asfarray(image_in)/255.0 * 0.99) + 0.01
    return np.asfarray(image_in)/255.0

# one-hot encode label vectors
def mod_one_hot(labels_in):
    #labels_out = np.zeros(neurons_per_layer[2]) + 0.01
    #labels_out[int(labels_in[0])] = 0.99

    labels_out = np.zeros(neurons_per_layer[2]) + 0.0
    labels_out[int(labels_in[0])] = 1.0
    return labels_out
```

0.2.2 The neural network

We will now build an MLP to recognize the digits in MNIST and later see how it does on your own handwriting. For this, we will build a simple MLP with one hidden layer. We will feed the network with one sample at a time, and update the weights after each sample.

Since each sample has 28x28=784 pixels, these will be the input neurons. As outputs, we have a vector of length 10, each entry corresponding to the likelihood of the digit as a label. In other words, the true labels from the training data set look for example like this: [0,0,0,0,0,0,0,0,1,0]. This would represent the label "8" (0-9). The network outputs y_hat will have the same shape, but will contain numbers between 0 and 1 - the network is approximate, and will never return a label with absolute certainty. The final output of the network will then be chosen as the digits with the highest likelihood, e.g. [0.1,0.2,0.3,0.1,0.2,0.1,0.4,0,0.8,0.1] would produce a network prediction of "8".

Thus, the input neurons are fixed at 784, the output neurons are fixed at 10, and we are free to choose the number of neurons in the hidden layer. As usual, there will be no activation applied to the input, but we will use a *sigmoid* on the hidden and output layers.

[3]:
```
# Helper functions:
# Sigmoid Activation Function:
def sigmoid(z):
    # YOUR CODE HERE
    raise NotImplementedError()
```

```
# From the lecture, we know that a=sig(z), and dsig/dz=a(1-a). Implement this␣
 ↪last equation here
def sigmoidgrad(a):
    # YOUR CODE HERE
    raise NotImplementedError()
```

[4]:
```
assert np.isclose(sigmoid(0), 0.5)
assert sigmoid(1) == 0.5*np.tanh(0.5)+0.5
```

```
      ␣
↪---------------------------------------------------------------------------

       NotImplementedError                       Traceback (most recent call␣
↪last)

       <ipython-input-4-50e5db2cf28f> in <module>
   ----> 1 assert np.isclose(sigmoid(0), 0.5)
         2 assert sigmoid(1) == 0.5*np.tanh(0.5)+0.5


       <ipython-input-3-dedc2ac1237b> in sigmoid(z)
         3 def sigmoid(z):
         4     # YOUR CODE HERE
   ----> 5     raise NotImplementedError()
         6
         7 # From the lecture, we know that a=sig(z), and dsig/dz=a(1-a).␣
↪Implement this last equation here


       NotImplementedError:
```

[ ]:
```
assert sigmoidgrad(0) == 0
assert sigmoidgrad(1) == 0
assert np.isclose(sigmoidgrad((1+np.sqrt(5))/2), -1)
```

[ ]:
```
# the class for an MLP with 1 hidden layer
class MLP1H:
    # neuronsperlayer is a vector with 3 entries, alpha is the learning rate
    def __init__(self,neuronsperlayer,alpha):
        self.n=neuronsperlayer
        self.n[0]=neuronsperlayer[0]
        self.n[1]=neuronsperlayer[1]
        self.n[2]=neuronsperlayer[2]
        self.alpha=alpha
```

```
        # initialize activations
        self.a_1=np.zeros((self.n[0],1))
        self.a_2=np.zeros((self.n[1],1))
        self.a_3=np.zeros((self.n[2],1))

        # initialize outputs. It is not strictly necessary to make z self here,␣
↪as it is locally used only
        self.z_1=np.zeros((self.n[0],1))
        self.z_2=np.zeros((self.n[1],1))
        self.z_3=np.zeros((self.n[2],1))

        # initialize the weight matrices between the input - hidden and hidden␣
↪- ouput layers
        # and fill them with random values, either normally or uniformly␣
↪distributed
        # YOUR CODE HERE
        raise NotImplementedError()

        self.activation=sigmoid
        return

    def sse_cost(self,pred,truth):
        # define the summed squared error cost function for a single sample␣
↪here, including the 0.5
        # YOUR CODE HERE
        raise NotImplementedError()
        return cost

    def forwardpass(self,inputs):
        # We convert the inputs from (784,) to a  true vector (784,1)
        X = np.array(inputs, ndmin=2).T
        # Implement the forward pass
        # YOUR CODE HERE
        raise NotImplementedError()
        return Yhat

    def train(self, inputs, targets):
        # Making sure the targets have the right shape
        tY = np.array(targets, ndmin=2).T
        # Implement the training:
        # first, pass the current sample through the net to fill a,z, etc.
        # next, compute the layer erros \delta, remember the direction
        # then, compute the cost function gradients
        # for tips, see accompanying lecture and notes

        # YOUR CODE HERE
```

```
        raise NotImplementedError()

        # Update the weights
        self.w23  = self.w23-self.alpha * dCd2
        self.w12  = self.w12-self.alpha * dCd1

        cost = self.sse_cost(Y_hat,tY)
        return cost
```

[ ]:
```
# We set up a small ANN here to test the implementation

neurons_per_layer=np.array([2,3,4])
learningrate=0.15
testANN = MLP1H(neurons_per_layer,learningrate)

#testing the cost function
assert testANN.sse_cost(1, 1) == 0
assert testANN.sse_cost(8.123, 12.123) == 8
assert testANN.sse_cost(np.array([2, 3]),np.array([3, 2])) == 1


#testing the correct shapes of w12, w23

# now testing the forward pass


# NOTE: There are no explicit tests for the backpropagation part. If the error␣
 ↪drops in training, that
# typically is a good sign that it works as planned. With this settings as␣
 ↪provided, a performance_on_test
# (see "Performance" section below) should be >92%.
# When debugging the backprop, think about the shapes of the objects, the␣
 ↪possible matrix operations etc.
# As a last resort, work it out by hand - for a much smaller network and a␣
 ↪single layer
```

0.2.3 Generating the Network

[ ]:
```
# now initiate the ANN with the given parameters
neurons_per_layer=np.array([784,100,10])
learningrate=0.15

ANN = MLP1H(neurons_per_layer,learningrate)
```

0.2.4 Training the ANN

[ ]:
```
# Before training the network, let us first pass an input to the untrained␣
 ↪network and see what happens
# The prediction will likely be wrong, but we can test the forward pass in␣
 ↪principal this way
sample = training_data_list[0].split(',')
image_array = np.asfarray(sample[1:]).reshape((28,28))
plt.imshow(image_array, cmap='Greys',interpolation='None')
print ("True label = ",sample[0])
inputs = scale_inputs(sample[1:])
outputs = ANN.forwardpass(inputs)
print("Predicted label = ",np.argmax(outputs))
```

[ ]:
```
# Training the network

nEpochs = 10

# Initializing the cost history plot
fig=plt.figure(figsize=(80, 40))
fig, ax = plt.subplots(figsize=(20, 10))
ax.size=(12,12)
plt.xlabel('Epochs',fontsize=24)
plt.ylabel('Costs C',fontsize=24)

# loop over epochs, remember, we train and update with single samples
for i in range(1,nEpochs+1):
    lcost=0.
    localcost=0.
    # loop over all training samples
    for record in training_data_list:
        data_list = record.split(',')
        # scale and encode inputs / labels
        inputs = scale_inputs(data_list [1:])
        labels= mod_one_hot(data_list [0:])
        # train and compute cost for each sample
        localcost=ANN.train(inputs,labels)
        lcost=lcost+localcost

    # Compute the training and test costs after each epoch
    Cost_Training = 0.
    Cost_Test     = 0.
    for record1 in training_data_list:
        # evaluate current model on the full training set and compute cost
        data_list  = record1.split(',')
        inputs = scale_inputs(data_list[1:])
        labels= mod_one_hot(data_list[0:])
```

```
        labels= np.array(labels,ndmin=2)
        y_hat=ANN.forwardpass(inputs)
        Cost_Training = Cost_Training + 0.5*np.sum(np.square(y_hat-labels.T))

    for record2 in test_data_list:
        # evaluate current model on the full test set and compute cost
        data_list= record2.split(',')
        inputs = scale_inputs(data_list[1:])
        labels= mod_one_hot(data_list[0:])
        labels= np.array(labels,ndmin=2)
        y_hat=ANN.forwardpass(inputs)
        Cost_Test = Cost_Test + 0.5*np.dot(y_hat.T-labels,(y_hat.T-labels).T)

    # normalize errors
    Cost_Training = Cost_Training / len(training_data_list)
    lcost = lcost / len(training_data_list)
    Cost_Test     = Cost_Test     / len(test_data_list)

    ax.scatter(i,Cost_Training,color='g',label='Training Data (epoch)')
    ax.scatter(i,lcost,color='k',label='Training Data (sample)')
    ax.scatter(i,Cost_Test,color='r',label='Test Data (epoch)')

    if i==1:
        ax.legend()
    display.display(plt.gcf())
    display.clear_output(wait=True)
print("training complete!")
print("Cost on test data (epoch): ",Cost_Test)
print("Cost on training data (epoch): ",Cost_Training)
print("Cost on training data (sample): ",lcost)
```

0.2.5 Querying the trained network

[ ]:
```
# let us now see if the training brought progress
sample = test_data_list[0].split(',')
image_array = np.asfarray(sample[1:]).reshape((28,28))
plt.imshow(image_array, cmap='Greys',interpolation='None')
print ("True label = ",sample[0])
inputs = scale_inputs(sample[1:])
outputs = ANN.forwardpass(inputs)
print("Predicted label = ",np.argmax(outputs))
```

0.2.6 Performance

[ ]:
```
# evaluate the network perfomance by computing the ratio of correct predictions␣
 ↪on the full test set
test_pred=np.zeros(len(test_data_list))
performance_on_test = 0
j=0
for record in test_data_list:
    data_list= record.split(',')
    inputs = scale_inputs(data_list[1:])
    labels= mod_one_hot(data_list[0:])
    correct_label = np.argmax(labels)
    y_hat=ANN.forwardpass(inputs)
    pred_label = np.argmax(y_hat)
    test_pred[j]=np.argmax(y_hat)
    j+=1
    if (pred_label == correct_label):
        performance_on_test+=1
print ("Performance = ", (performance_on_test / len(test_data_list))*100,"%")
```

[ ]:
```
# Here are random samples from the test data, and the network prediction

from random import randint

fig, ax = plt.subplots(10,10,figsize=(10,10))
ax = ax.flatten()

for i in range(100):
    r = random.randint(0, len(test_data_list))
    sample= test_data_list[r].split(',')
    image= np.asfarray(sample[1:]).reshape((28,28))
    ax[i].imshow(image, cmap='gray_r')
    ax[i].set_title(int(test_pred[r]), fontsize=15)
    ax[i].axis('off')

plt.tight_layout()
plt.show()
```

0.2.7 Your own handwriting

Check the networks prediction on your own handwriting.Either write on a piece of paper and take a picture, or use e.g. gimp to directly create a sample image. Save as png, and store in the folder indicated below. On linux, you can convert e.g. a jpg with 'convert Name.jpg Name.png'

[ ]:
```
from skimage.transform import resize
import matplotlib.image as mpimg
from skimage.color import rgb2gray
```

```
# place own image in the folder, adjust name
own_image=mpimg.imread("./mnist_dataset/own/drei.png")
plt.imshow(own_image)
own_image = rgb2gray(own_image)
own_image = resize(own_image, (28,28))

own_image = 255 - own_image.reshape(784)

inputs = scale_inputs(own_image )
outputs = ANN.forwardpass(inputs)
print("Predicted label = ",np.argmax(outputs))
```

lorenz

April 14, 2021

1 Building a simple model with KERAS for the Lorenz System

The Lorenz system is a harmless looking system of ODEs, describing the trajectory of a point in x,y,z-space. It is however infamous, as it can have extremely erratic dynamics. Over a wide range of parameters, the solution oscillates irregularly, but never repeats itself. Remarkably, it stays in a bounded region in phase space, a so-called strange attractor - a fractal with a dimension between 2 and 3.

It is a prototype for non-linear dynamical systems, and we will use it to show two things in this notebook:

a) How simple it is to build an NN using KERAS

b) How NNs can learn complex dynamics

Note: There are no graded parts in this notebook, but you should look around still and explore!

1.1 Solving the Lorenz System with an ODE solver

Here, we show the Lorenz system, its typical solutions and solve the ODE the classical way - with a numerical integrator / quadrature from python. This serves as a generator for the training data. We will then try to build a NN to lear the dynamics of the system:

Given (x,y,z) at timestep t as an input, predict the next state (x,y,z) at timestep t+1.

Thus, we can generate the training data for the NN by shifting the solution by one timestep. Since we want to learn the dynamics and avoid overfitting, we will use many runs with different initial conditions.

[ ]:
```
# See also the blog on https://scipython.com/blog/the-lorenz-attractor/
# Christian Hill, January 2016 as well as the website http://www.databookuw.com/


%matplotlib inline
import matplotlib
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

```
import pylab
img_width, img_heigth, img_dpi = 2000, 1500, 100


# Lorenz parameters and initial conditions, changed somewhat from the standard␣
 ↪set
sigma, beta, rho = 10, 2.667, 32
(x0, y0, z0) = (2, 2, 2)

# define the ODE system
def lorenz_system(t, X, sigma, beta, rho):
    x, y, z = X
    dxdt = -sigma*(x - y)
    dydt = rho*x - y - x*z
    dzdt = -beta*z + x*y
    return dxdt, dydt, dzdt


# t_end and the number of timesteps and runs of the system
tend, n, runs = 32, 2000, 100
t = np.linspace(0, tend, n)

# solution, 3 coordinates, timesteps, runs
solution=np.zeros((3, n,runs))


for run in range(0,runs):
# Integrate the Lorenz equations with the ODE solver of python
    print("solving Lorenz system no. ",run,end='\r')
    # set up the solver, default is RungeKutta 45
    solver = solve_ivp(lorenz_system, (0, tend), (x0, y0, z0), args=(sigma,␣
 ↪beta, rho),dense_output=True)
    solution[:,:,run] = solver.sol(t)
    # randomize initial condition
    (x0, y0, z0) = 20*(np.random.rand(3)-0.5)


# Plot some of the runs
printfig=1
s, k = 1, 15 # s: plot every sth step, k: plot every kth run
print("\nPreparing plots \n")
if printfig:
    fig = plt.figure(facecolor='k', figsize=(img_width/img_dpi, img_heigth/
 ↪img_dpi))
    ax = fig.gca(projection='3d')
    ax.set_facecolor('k')
    fig.subplots_adjust(left=0, right=1, bottom=0, top=1)
```

```
    cmap = plt.cm.winter
    for run in range(0,runs-k,k):
        for i in range(0,n-s,s):

            #vals = np.linspace(0,1,256)
            #np.random.shuffle(vals)
            #cmap = plt.cm.colors.ListedColormap(plt.cm.jet(vals))

            ax.plot(solution[0,i:i+s+1,run], solution[1,i:i+s+1,run],␣
↪solution[2,i:i+s+1,run], color=cmap(i/n), alpha=0.5,linewidth=2.0)
    ax.set_axis_off()
    plt.show()
```

2 Solving the Lorenz System with an ANN

It's much easier using a package for creating and using a neural network. *Keras* was developed as a high-level API that supports multiple backends. Recently, it was incorporated into the *TensorFlow* framework from Google. Roughly, the pipeline for an ANN project is

- Define the model.
- Compile the model.
- Fit the model.
- Evaluate the model.
- Make predictions.

[ ]:
```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from sklearn.utils import shuffle

# keras is part of TensorFlow, so importing things is done via tensorflow
# Sequential is used for creating sequential models, that are built layer
# by layer as seen below
# actions in such a model are created by adding layers to a model. Dense
# layers are normal neural layers which we've seen in the lecture

# the following function clears a keras session, such that all old models
# get deleted
from tensorflow.keras.backend import clear_session

clear_session()

# We want to learn the mapping from X_t -> X_(t+1), so we shift the outputs Y␣
↪by one
X=solution[:,0:solution.shape[1]-1,:]
Y=solution[:,1:solution.shape[1],:]
```

```
X=np.transpose(X.reshape(3,-1))
Y=np.transpose(Y.reshape(3,-1))

# Shuffling is important to avoid bias in the test set
X, Y = shuffle(X,Y)
input_shape = (3,)

# Here we build the KERAS model - it should be self explanatory!
# Layers are added just by stacking them as shown below
# "Dense" adds a fully connected layer of neurons. Activation is given
# for all neurons simultaneously, although there are other options.
# input_shape in the first layer determines the shape that is
# expected from the input array.


model = keras.Sequential(
    [
        layers.Dense(3, input_shape=input_shape, name="input"),
        layers.Dense(20, activation="relu", name="layer2"),
        layers.Dense(20, activation="relu", name="layer3"),
        layers.Dense(20, activation="relu", name="layer4"),
        layers.Dense(3, name="output"),
    ]
)

# This is so called "call back". It is used to influence the model during the␣
 ↪training stage. Here, we use it
# to introduce a decay of the learning rate lr

def lr_scheduler(epoch, lr):
    decay_rate = 0.99
    decay_step = 10
    if epoch % decay_step == 0 and epoch:
        return lr * decay_rate
    return lr

# hook the call backs up
callbacks = [
    keras.callbacks.LearningRateScheduler(lr_scheduler, verbose=1)
]
# We now compile the model. You can set the loss function, the optimizer and␣
 ↪many other things here
model.compile(loss='mean_absolute_error', optimizer='adam',␣
 ↪metrics=['mean_squared_error'])

# The next line starts the training. We do not need to take care about␣
 ↪backpropagation at all.
```

```
model.fit(X, Y, epochs=20, verbose=1, validation_split=0.2,callbacks=callbacks)
print(model.summary())
```

3 Evaluate model on test run

We evalute the model using model.predict. Here, we generate a new test run from random initial conditions, solve for the exact solution with the ODE solver and then apply the NN. Note that since the NN has make predictions one step at a time, this is quite a lot slower than the training, where we could put in many samples at the same time However, here we want the net to make iterative predictions starting just from x0,y0,z0, so we are stuck with this approach:

X0-> X1=model(X0) -> X2=model(X1) ...

[ ]:
```
# Generate test run
# randomize initial condition
(x0, y0, z0) = 20*(np.random.rand(3)-0.5)
solver = solve_ivp(lorenz_system, (0, tend), (x0, y0, z0), args=(sigma, beta,␣
 ↪rho),dense_output=True)
test_solution=np.zeros((3, n))
test_solution[:,:] = solver.sol(t)
inp=np.array([[x0, y0, z0]])
predout=np.zeros((3, n))

for i in range(0,n):
    if (i % k) ==0: print("Prediction of time step ",i," of ",n,end='\r')
    predout[:,i]=model.predict(inp)
    inp=np.array(predout[:,i])[None,:]


s=1
print("\nPreparing plots \n")
if printfig:
    fig = plt.figure(facecolor='k', figsize=(img_width/img_dpi, img_heigth/
 ↪img_dpi))
    ax = fig.gca(projection='3d')
    ax.set_facecolor('k')
    fig.subplots_adjust(left=0, right=1, bottom=0, top=1)
    cmap = plt.cm.summer
    for i in range(0,n-s,s):

        #vals = np.linspace(0,1,256)
        #np.random.shuffle(vals)
        #cmap = plt.cm.colors.ListedColormap(plt.cm.jet(vals))

        ax.plot(test_solution[0,i:i+s+1], test_solution[1,i:i+s+1],␣
 ↪test_solution[2,i:i+s+1], color=cmap(1), alpha=0.4,linewidth=3.0)
```

```
        ax.plot(predout[0,i:i+s+1], predout[1,i:i+s+1], predout[2,i:i+s+1],␣
↪color='red', alpha=0.4,linewidth=3.0)
    ax.set_axis_off()
    plt.show()
```

4 Evalute on x-component of test run

[ ]:
```
print(test_solution[0,:])
print(predout[0,:])
fig = plt.figure(facecolor='w', figsize=(img_width/img_dpi, img_heigth/img_dpi))
plt.plot(t,test_solution[0,:])
plt.plot(t,predout[0,:])
```

5 Things to investigate / think about

- Build more complex networks: Which one works better: tall and skinny or short and fat ones?
- Train a simpler ODE! Look at the Lorenz system and think about how to make it less sensitive (remove / weaken nonlinearities)
- Effects of activation functions, optimizers, metrics, ...
- Which improves the training: more runs of the Lorenz system, or more steps per run (runs vs. n?)
- Influence of timestep size; can a model trained on delta_t work on 2 delta_t?
- Getting more familiar with KERAS
- Sequence methods for temporal data
- Physics informed NNs
- We have essentially replace the ODE solver of python with an NN. Are you impressed with the speed and accuracy of the NN, or not so much?

[ ]:
```
#model.save("model_val12_0.0029")
#reconstructed_model = keras.models.load_model("model_val12_0.0023")
```

lorenz

April 14, 2021

1 Building a simple model with KERAS for the Lorenz System

The Lorenz system is a harmless looking system of ODEs, describing the trajectory of a point in x,y,z-space. It is however infamous, as it can have extremely erratic dynamics. Over a wide range of parameters, the solution oscillates irregularly, but never repeats itself. Remarkably, it stays in a bounded region in phase space, a so-called strange attractor - a fractal with a dimension between 2 and 3.

It is a prototype for non-linear dynamical systems, and we will use it to show two things in this notebook:

a) How simple it is to build an NN using KERAS

b) How NNs can learn complex dynamics

Note: There are no graded parts in this notebook, but you should look around still and explore!

1.1 Solving the Lorenz System with an ODE solver

Here, we show the Lorenz system, its typical solutions and solve the ODE the classical way - with a numerical integrator / quadrature from python. This serves as a generator for the training data. We will then try to build a NN to lear the dynamics of the system:

Given (x,y,z) at timestep t as an input, predict the next state (x,y,z) at timestep t+1.

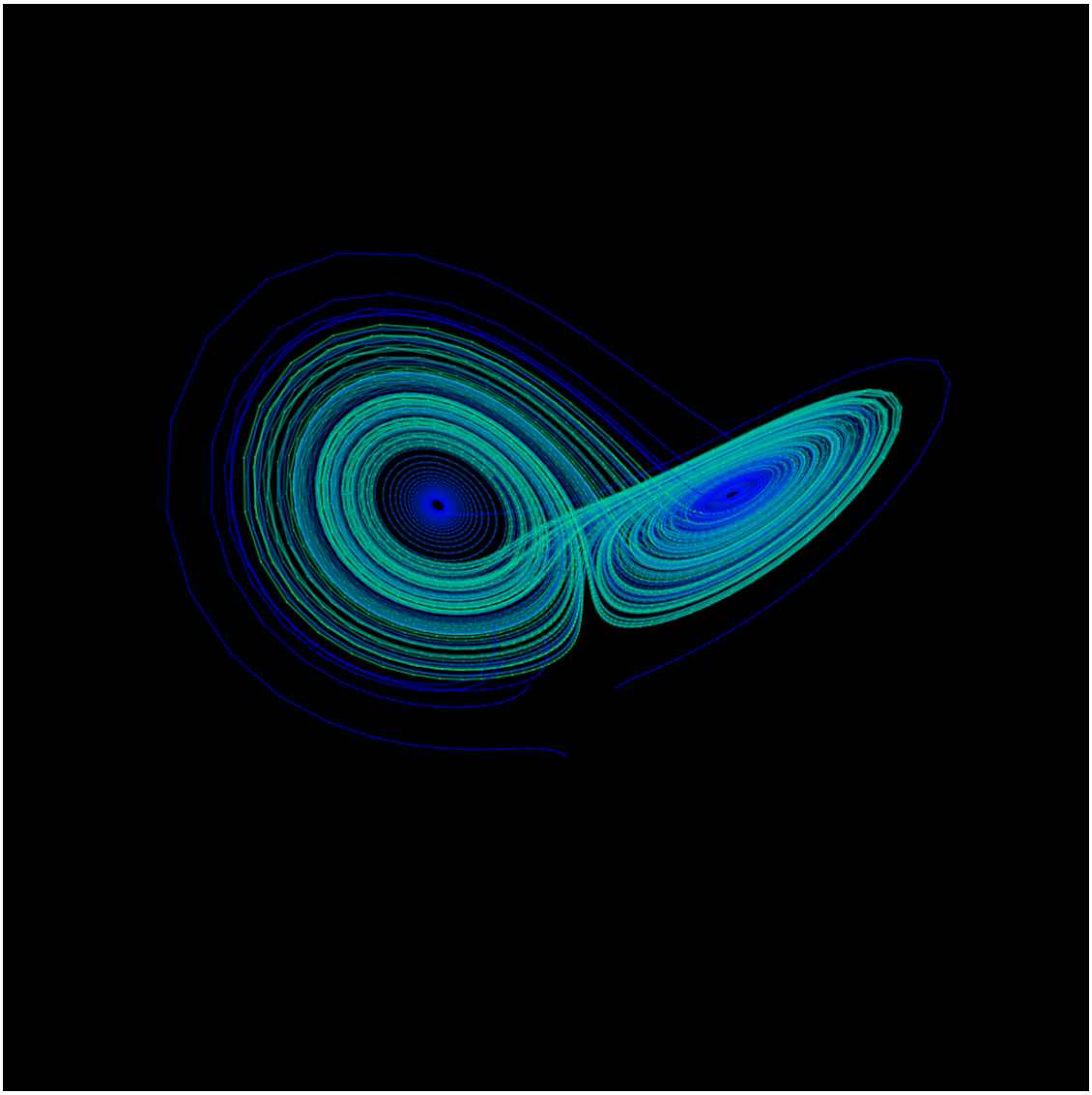
Thus, we can generate the training data for the NN by shifting the solution by one timestep. Since we want to learn the dynamics and avoid overfitting, we will use many runs with different initial conditions.

[1]:
```
# See also the blog on https://scipython.com/blog/the-lorenz-attractor/
# Christian Hill, January 2016 as well as the website http://www.databookuw.com/


%matplotlib inline
import matplotlib
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

```
import pylab
img_width, img_heigth, img_dpi = 2000, 1500, 100


# Lorenz parameters and initial conditions, changed somewhat from the standard␣
 ↪set
sigma, beta, rho = 10, 2.667, 32
(x0, y0, z0) = (2, 2, 2)

# define the ODE system
def lorenz_system(t, X, sigma, beta, rho):
    x, y, z = X
    dxdt = -sigma*(x - y)
    dydt = rho*x - y - x*z
    dzdt = -beta*z + x*y
    return dxdt, dydt, dzdt


# t_end and the number of timesteps and runs of the system
tend, n, runs = 32, 2000, 100
t = np.linspace(0, tend, n)

# solution, 3 coordinates, timesteps, runs
solution=np.zeros((3, n,runs))


for run in range(0,runs):
# Integrate the Lorenz equations with the ODE solver of python
    print("solving Lorenz system no. ",run,end='\r')
    # set up the solver, default is RungeKutta 45
    solver = solve_ivp(lorenz_system, (0, tend), (x0, y0, z0), args=(sigma,␣
 ↪beta, rho),dense_output=True)
    solution[:,:,run] = solver.sol(t)
    # randomize initial condition
    (x0, y0, z0) = 20*(np.random.rand(3)-0.5)


# Plot some of the runs
printfig=1
s, k = 1, 15 # s: plot every sth step, k: plot every kth run
print("\nPreparing plots \n")
if printfig:
    fig = plt.figure(facecolor='k', figsize=(img_width/img_dpi, img_heigth/
 ↪img_dpi))
    ax = fig.gca(projection='3d')
    ax.set_facecolor('k')
    fig.subplots_adjust(left=0, right=1, bottom=0, top=1)
```

```
    cmap = plt.cm.winter
    for run in range(0,runs-k,k):
        for i in range(0,n-s,s):

            #vals = np.linspace(0,1,256)
            #np.random.shuffle(vals)
            #cmap = plt.cm.colors.ListedColormap(plt.cm.jet(vals))

            ax.plot(solution[0,i:i+s+1,run], solution[1,i:i+s+1,run],␣
↪solution[2,i:i+s+1,run], color=cmap(i/n), alpha=0.5,linewidth=2.0)
    ax.set_axis_off()
    plt.show()
```

```
solving Lorenz system no.  99
Preparing plots
```



2 Solving the Lorenz System with an ANN

It's much easier using a package for creating and using a neural network. *Keras* was developed as a high-level API that supports multiple backends. Recently, it was incorporated into the *TensorFlow* framework from Google. Roughly, the pipeline for an ANN project is

- Define the model.
- Compile the model.
- Fit the model.
- Evaluate the model.
- Make predictions.

[ ]:
```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from sklearn.utils import shuffle

# keras is part of TensorFlow, so importing things is done via tensorflow
# Sequential is used for creating sequential models, that are built layer
# by layer as seen below
# actions in such a model are created by adding layers to a model. Dense
# layers are normal neural layers which we've seen in the lecture

# the following function clears a keras session, such that all old models
# get deleted
from tensorflow.keras.backend import clear_session

clear_session()

# We want to learn the mapping from X_t -> X_(t+1), so we shift the outputs Y␣
 ↪by one
X=solution[:,0:solution.shape[1]-1,:]
Y=solution[:,1:solution.shape[1],:]
X=np.transpose(X.reshape(3,-1))
Y=np.transpose(Y.reshape(3,-1))

# Shuffling is important to avoid bias in the test set
X, Y = shuffle(X,Y)
input_shape = (3,)

# Here we build the KERAS model - it should be self explanatory!
# Layers are added just by stacking them as shown below
# "Dense" adds a fully connected layer of neurons. Activation is given
# for all neurons simultaneously, although there are other options.
# input_shape in the first layer determines the shape that is
# expected from the input array.


model = keras.Sequential(
    [
        layers.Dense(3, input_shape=input_shape, name="input"),
        layers.Dense(20, activation="relu", name="layer2"),
        layers.Dense(20, activation="relu", name="layer3"),
        layers.Dense(20, activation="relu", name="layer4"),
        layers.Dense(3, name="output"),
    ]
)
```

```
# This is so called "call back". It is used to influence the model during the␣
 ↪training stage. Here, we use it
# to introduce a decay of the learning rate lr

def lr_scheduler(epoch, lr):
    decay_rate = 0.99
    decay_step = 10
    if epoch % decay_step == 0 and epoch:
        return lr * decay_rate
    return lr

# hook the call backs up
callbacks = [
    keras.callbacks.LearningRateScheduler(lr_scheduler, verbose=1)
]
# We now compile the model. You can set the loss function, the optimizer and␣
 ↪many other things here
model.compile(loss='mean_absolute_error', optimizer='adam',␣
 ↪metrics=['mean_squared_error'])

# The next line starts the training. We do not need to take care about␣
 ↪backpropagation at all.
model.fit(X, Y, epochs=20, verbose=1, validation_split=0.2,callbacks=callbacks)
print(model.summary())
```

```
/usr/local/lib/python3.7/dist-
packages/tensorflow/python/framework/dtypes.py:516: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorflow/python/framework/dtypes.py:517: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorflow/python/framework/dtypes.py:518: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorflow/python/framework/dtypes.py:519: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorflow/python/framework/dtypes.py:520: FutureWarning: Passing
```

```
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorflow/python/framework/dtypes.py:525: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:541: FutureWarning:
Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future
version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:542: FutureWarning:
Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future
version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:543: FutureWarning:
Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future
version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:544: FutureWarning:
Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future
version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:545: FutureWarning:
Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future
version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:550: FutureWarning:
Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future
version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])

WARNING:tensorflow:From /usr/local/lib/python3.7/dist-
packages/tensorflow/python/ops/init_ops.py:1251: calling
VarianceScaling.__init__ (from tensorflow.python.ops.init_ops) with dtype is
deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the
constructor
Train on 159920 samples, validate on 39980 samples
```

```
Epoch 00001: LearningRateScheduler reducing learning rate to
0.0010000000474974513.
Epoch 1/20
159920/159920 [==============================] - 18s 113us/sample - loss: 0.5197
- mean_squared_error: 5.2149 - val_loss: 0.1515 - val_mean_squared_error: 0.0806

Epoch 00002: LearningRateScheduler reducing learning rate to
0.0010000000474974513.
Epoch 2/20
159920/159920 [==============================] - 18s 112us/sample - loss: 0.1531
- mean_squared_error: 0.0803 - val_loss: 0.1465 - val_mean_squared_error: 0.0554

Epoch 00003: LearningRateScheduler reducing learning rate to
0.0010000000474974513.
Epoch 3/20
159920/159920 [==============================] - 18s 110us/sample - loss: 0.1217
- mean_squared_error: 0.0482 - val_loss: 0.1138 - val_mean_squared_error: 0.0369

Epoch 00004: LearningRateScheduler reducing learning rate to
0.0010000000474974513.
Epoch 4/20
159920/159920 [==============================] - 18s 110us/sample - loss: 0.1058
- mean_squared_error: 0.0360 - val_loss: 0.0994 - val_mean_squared_error: 0.0321

Epoch 00005: LearningRateScheduler reducing learning rate to
0.0010000000474974513.
Epoch 5/20
159920/159920 [==============================] - 17s 108us/sample - loss: 0.0953
- mean_squared_error: 0.0296 - val_loss: 0.1038 - val_mean_squared_error: 0.0285

Epoch 00006: LearningRateScheduler reducing learning rate to
0.0010000000474974513.
Epoch 6/20
159920/159920 [==============================] - 18s 110us/sample - loss: 0.0899
- mean_squared_error: 0.0261 - val_loss: 0.0702 - val_mean_squared_error: 0.0190

Epoch 00007: LearningRateScheduler reducing learning rate to
0.0010000000474974513.
Epoch 7/20
159920/159920 [==============================] - 18s 110us/sample - loss: 0.0857
- mean_squared_error: 0.0237 - val_loss: 0.0861 - val_mean_squared_error: 0.0220

Epoch 00008: LearningRateScheduler reducing learning rate to
0.0010000000474974513.
Epoch 8/20
159920/159920 [==============================] - 17s 107us/sample - loss: 0.0818
- mean_squared_error: 0.0215 - val_loss: 0.0787 - val_mean_squared_error: 0.0190
```

```
Epoch 00009: LearningRateScheduler reducing learning rate to
0.0010000000474974513.
Epoch 9/20
159920/159920 [==============================] - 18s 110us/sample - loss: 0.0807
- mean_squared_error: 0.0204 - val_loss: 0.0895 - val_mean_squared_error: 0.0219

Epoch 00010: LearningRateScheduler reducing learning rate to
0.0010000000474974513.
Epoch 10/20
159920/159920 [==============================] - 17s 108us/sample - loss: 0.0778
- mean_squared_error: 0.0189 - val_loss: 0.0713 - val_mean_squared_error: 0.0169

Epoch 00011: LearningRateScheduler reducing learning rate to
0.0009990000470224768.
Epoch 11/20
159920/159920 [==============================] - 17s 109us/sample - loss: 0.0752
- mean_squared_error: 0.0178 - val_loss: 0.0872 - val_mean_squared_error: 0.0226

Epoch 00012: LearningRateScheduler reducing learning rate to
0.0009990000877678394.
Epoch 12/20
159920/159920 [==============================] - 17s 107us/sample - loss: 0.0719
- mean_squared_error: 0.0162 - val_loss: 0.0712 - val_mean_squared_error: 0.0152

Epoch 00013: LearningRateScheduler reducing learning rate to
0.0009990000877678394.
Epoch 13/20
159920/159920 [==============================] - 17s 108us/sample - loss: 0.0699
- mean_squared_error: 0.0152 - val_loss: 0.0840 - val_mean_squared_error: 0.0172

Epoch 00014: LearningRateScheduler reducing learning rate to
0.0009990000877678394.
Epoch 14/20
159920/159920 [==============================] - 17s 109us/sample - loss: 0.0666
- mean_squared_error: 0.0142 - val_loss: 0.0648 - val_mean_squared_error: 0.0122

Epoch 00015: LearningRateScheduler reducing learning rate to
0.0009990000877678394.
Epoch 15/20
159920/159920 [==============================] - 18s 110us/sample - loss: 0.0656
- mean_squared_error: 0.0138 - val_loss: 0.0670 - val_mean_squared_error: 0.0124

Epoch 00016: LearningRateScheduler reducing learning rate to
0.0009990000877678394.
Epoch 16/20
159920/159920 [==============================] - 18s 111us/sample - loss: 0.0639
- mean_squared_error: 0.0133 - val_loss: 0.0503 - val_mean_squared_error: 0.0100
```

```
Epoch 00017: LearningRateScheduler reducing learning rate to
0.0009900000877678394.
Epoch 17/20
159920/159920 [==============================] - 17s 107us/sample - loss: 0.0622
- mean_squared_error: 0.0127 - val_loss: 0.0540 - val_mean_squared_error: 0.0120

Epoch 00018: LearningRateScheduler reducing learning rate to
0.0009900000877678394.
Epoch 18/20
148064/159920 [==========================>...] - ETA: 1s - loss: 0.0625 -
mean_squared_error: 0.0126
```

3 Evaluate model on test run

We evalute the model using model.predict. Here, we generate a new test run from random initial conditions, solve for the exact solution with the ODE solver and then apply the NN. Note that since the NN has make predictions one step at a time, this is quite a lot slower than the training, where we could put in many samples at the same time However, here we want the net to make iterative predictions starting just from x0,y0,z0, so we are stuck with this approach:

X0-> X1=model(X0) -> X2=model(X1) ...

[ ]:
```
# Generate test run
# randomize initial condition
(x0, y0, z0) = 20*(np.random.rand(3)-0.5)
solver = solve_ivp(lorenz_system, (0, tend), (x0, y0, z0), args=(sigma, beta,␣
 ↪rho),dense_output=True)
test_solution=np.zeros((3, n))
test_solution[:,:] = solver.sol(t)
inp=np.array([[x0, y0, z0]])
predout=np.zeros((3, n))

for i in range(0,n):
    if (i % k) ==0: print("Prediction of time step ",i," of ",n,end='\r')
    predout[:,i]=model.predict(inp)
    inp=np.array(predout[:,i])[None,:]


s=1
print("\nPreparing plots \n")
if printfig:
    fig = plt.figure(facecolor='k', figsize=(img_width/img_dpi, img_heigth/
 ↪img_dpi))
    ax = fig.gca(projection='3d')
    ax.set_facecolor('k')
    fig.subplots_adjust(left=0, right=1, bottom=0, top=1)
    cmap = plt.cm.summer
```

```
    for i in range(0,n-s,s):

        #vals = np.linspace(0,1,256)
        #np.random.shuffle(vals)
        #cmap = plt.cm.colors.ListedColormap(plt.cm.jet(vals))

        ax.plot(test_solution[0,i:i+s+1], test_solution[1,i:i+s+1],␣
↪test_solution[2,i:i+s+1], color=cmap(1), alpha=0.4,linewidth=3.0)
        ax.plot(predout[0,i:i+s+1], predout[1,i:i+s+1], predout[2,i:i+s+1],␣
↪color='red', alpha=0.4,linewidth=3.0)
    ax.set_axis_off()
    plt.show()
```

4 Evalute on x-component of test run

[ ]:
```
print(test_solution[0,:])
print(predout[0,:])
fig = plt.figure(facecolor='w', figsize=(img_width/img_dpi, img_heigth/img_dpi))
plt.plot(t,test_solution[0,:])
plt.plot(t,predout[0,:])
```

5 Things to investigate / think about

- Build more complex networks: Which one works better: tall and skinny or short and fat ones?
- Train a simpler ODE! Look at the Lorenz system and think about how to make it less sensitive (remove / weaken nonlinearities)
- Effects of activation functions, optimizers, metrics, ...
- Which improves the training: more runs of the Lorenz system, or more steps per run (runs vs. n?)
- Influence of timestep size; can a model trained on delta_t work on 2 delta_t?
- Getting more familiar with KERAS
- Sequence methods for temporal data
- Physics informed NNs
- We have essentially replace the ODE solver of python with an NN. Are you impressed with the speed and accuracy of the NN, or not so much?

[ ]:
```
#model.save("model_val12_0.0029")
#reconstructed_model = keras.models.load_model("model_val12_0.0023")
```