Lab Assignment

Note: Example codes and figures are provided at the end for your reference.

Questions pertaining to Session 1

1.a Create a python function `solver(func, lr, init_guess, n)` with the following inputs, implementation, and outputs.

Inputs:

- a **function** `func` such that its output $f(x)$ is a polynomial in a scalar input x. For clarifications, see footnote.[1] Example:

```
def func(x):
    f = x**2 - 2*x + 1
    return f
```

- `lr`: learning rate
- `init_guess`: initial guess for x (a scalar value)
- `n`: number of steps

Implementation:

- The function `solver` should find the one of the roots of the polynomial $f(x) = 0$. Note: depending on the polynomial, there maybe zero, one, or more than one roots.
- Implement the classical gradient descent optimization algorithm that we covered in the lab session (not Adam, SGD, or LBFGS) for the aforementioned problem.
- Choose an appropriate loss function to measure how far the optimizer is from the target, i.e. $f(x) = 0$. Example: `loss = f` or `f**2`.
- At every iteration, keep track of both the loss value and the current value of x.

Outputs:

- `x_history`: list of all values of x at each step of the optimization loop. Note: make sure that `init_guess` is included as the first value in `x_history`.
- `loss_history`: list of all values of `loss` at each step of the optimization loop. Make sure to detach from the computational graph to avoid overworking your computer.

[1] We will use different forms of f in other parts, that is why we want a code that can be reused for any arbitrary f. Note: yes, python functions can take other python functions as input; see Example A.

1.b Create a python function `visualize(func, x_history)` to visualize the trajectory or path taken by the optimizer. The plot should look like Example B. The function has the following inputs and implementation, but no outputs.

Inputs:

- `func`: same description as part 1.a.
- `x_history`: same description as part 1.a.

Implementation:

- Using `func`, (line-)plot $f(x)$ vs. x in the range $x \in [-5, 5]$.
- In the same figure, plot the trajectory or path that the optimizer took. The plot should look like Example B.
- Tip: Fix the x-axis and y-axis limits in the plot settings to reasonable values to avoid unreadable plots when your data will contain infinities.

We now have all the tools to start analyzing and interpreting the role of learning rate, initial guess, and of course, the form of the function $f(x)$ itself.

1.c With $f(x) = (x^2 - 2x + 1)$, `n=1000`, `init_guess=4.5`,

- plot the final loss vs. learning rate for several values (at least 100) of learning rates between 10^{-4} and 10^{-1}. Use a log-log scale for the plot. Tip: If the computations become too heavy, decrease the number of steps (`n`). Tip: Use `np.nanmin()` or `np.nanargmin()` when the operand may contain `nan` or `inf` (infinity) values.
- What should be the best value of the learning rate for this problem. Explain your interpretation and reasoning.
- Would this optimal learning rate also applicable for different problem and why?

1.d With $f(x) = (x^2 - 2x + 1)$, `n=1000`, `init_guess=4.5`, visualize the path of the optimizer for several learning rates between 10^{-4} and 10^{-1}.

- Report different cases (**not more than five**) where you observe the transition from the optimizer (i) diverging to infinity, (ii) oscillating but converging to the root of the polynomial, and (iii) converging gradually to the root of the polynomial.
- Explain your interpretation and reasoning of the optimizer behavior as learning rate changes.

1.e With $f(x) = (x - 3)^2(x + 3)^2$, `n=1000`, `init_guess=4.5`,

- plot optimizer path for `lr=`$\{$`1,2,5,8,10`$\}\times 10^{-5}$.
- For each case, explain with reasoning your observation of the minima found by the optimizer and the role of the learning rate.

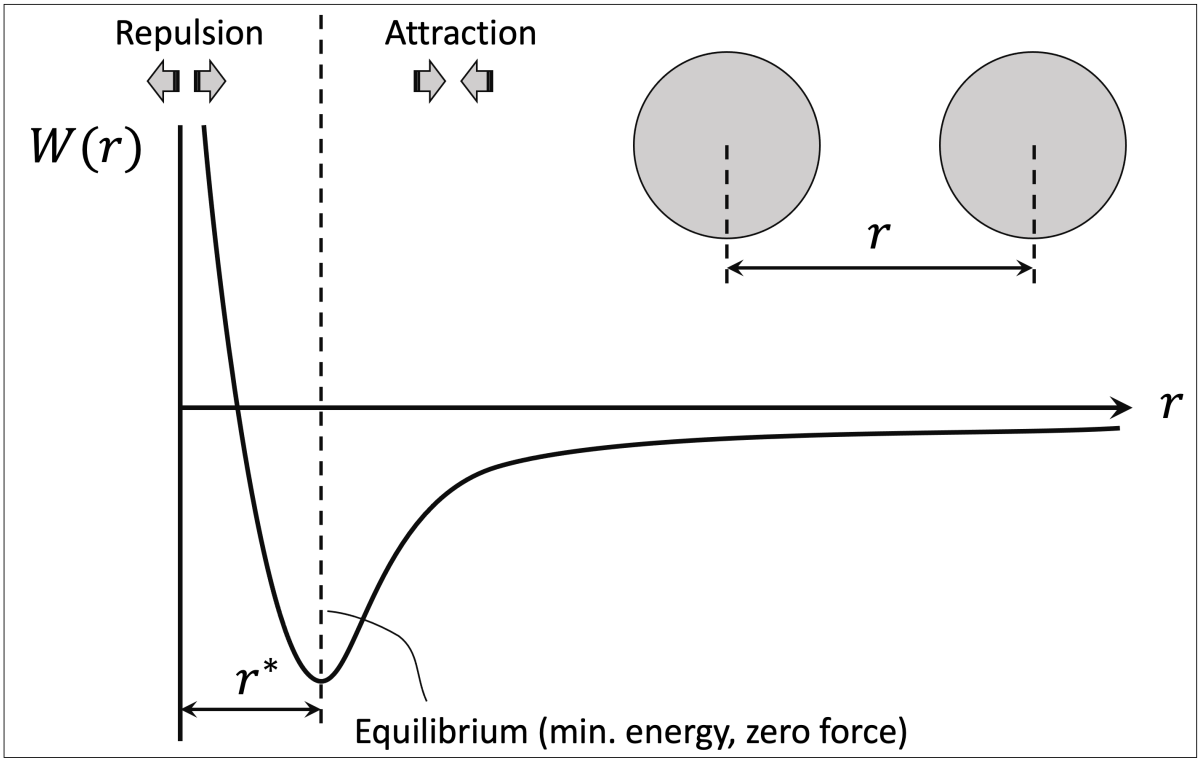
1.f With $f(x) = (x - 3)^2(x + 3)^2$, `n=1000`, `lr=`$10^{-5}$,

- plot optimizer path for `init_guess=`$\{$`-1.0, 0.0, 1.0, 2.0, 3.0, 4.5`$\}$.
- For each case, explain with reasoning your observation of the minima found by the optimizer and the role of the initial guess.

Questions pertaining to Session 2

Material properties are governed by the physics at multiple scales – from the quantum scale to the molecular scale and up to the micron scale. For this problem, we focus on the molecular scale. At this scale, interatomic forces are either attractive (when two particles are very far apart) or repulsive (when two particles are very close). Modeling these forces are important for simulating and calculating effective material properties.

For the scope of this problem, we consider a simple one-dimensional setting containing two particles separated by distance (r). The forces on both particles are derived from an interatomic potential energy (W) – i.e., the total potential energy of the system due to particles being in proximity of each other. A typical potential energy of a two particle system looks like the figure below – when the two particles are very close to each other, the energy is very large (infinite when particles are coincident); when the particles are very far apart, the energy is zero. There is an equilibrium point (denoted by distance r^*) where the energy is minimum.



The potential energy is itself calculated from quantum-level simulations. However, it is computationally intractable to study large molecular systems with quantum-level simulations. Therefore, we perform quantum-level simulations on a pair of particles in isolation at different relative distances and approximate the energy as a function of the distance between the particles. Using this computationally inexpensive approximation of potential energy, we can bypass expensive/slow quantum-level simulations and be able to perform large molecular-level simulations efficiently

For this problem, you are provided a training data (potentially obtained from quantum-level simulations) which contains the potential energy W (`energy_train.npy`) for different values of distance r (`distance_train.npy`) between the two particles. You are also provided testing data (`distance_test.npy` and `energy_test.npy`) for validation purposes. The data files are uploaded separately on Brightspace.

2.a Train a neural network which takes in as input the distance between two atoms (r) and outputs the potential energy (W). As an initial starting point, use the following architecture.

- Linear layer: Input dimension $\rightarrow$ 8
- Activation: `ReLU()`

- Linear layer: 8 $\rightarrow$ Output dimension

List all the neural network and training settings (architecture, parameters, loss function, optimizer, learning rate, number of epochs, etc.) in a table. Plot the following:

- training and testing loss histories,
- learnt energy vs. distance function (continuous line; not scatter plot) along with testing data (use scatter plot here). Use appropriate axis limits for visualization.
- predicted vs. true energy along with coefficient of determination (R^2) value with respect to a line with slope=1 and passing through origin.
  (Ref: for more details on R^2, see: https://en.wikipedia.org/wiki/Coefficient_of_determination)
  R^2 maybe calculated as the following.

```
def calculate_R2(true, pred):
    # Note:  both true and pred should be numpy 1D-array, NOT torch tensors
    true_mean = true.mean()
    ss_tot = ((true-true_mean)**2).sum()
    ss_res = ((true-pred)**2).sum()
    return 1.  - (ss_res/ss_tot)
```

Evaluate and discuss (with appropriate explanations) the performance of your trained neural network using the above plots. Also discuss if the neural network is underfitting or overfitting the data?

2.b Tune your neural network model until you achieve good accuracy. You may change one or more of the following: e.g., architecture, activations, learning rate, no. of epochs, optimizer, loss function, etc.

- In the same table as in part 2.a, list all the new neural network and training settings (architecture, parameters, loss function, optimizer, learning rate, number of epochs, etc.) in a separate column (for direct comparison).
- Make the same plots as in part 2.a and place them side-by-side (for direct comparison).
- Discuss the improvement in performance and explain the motivation behind each change.

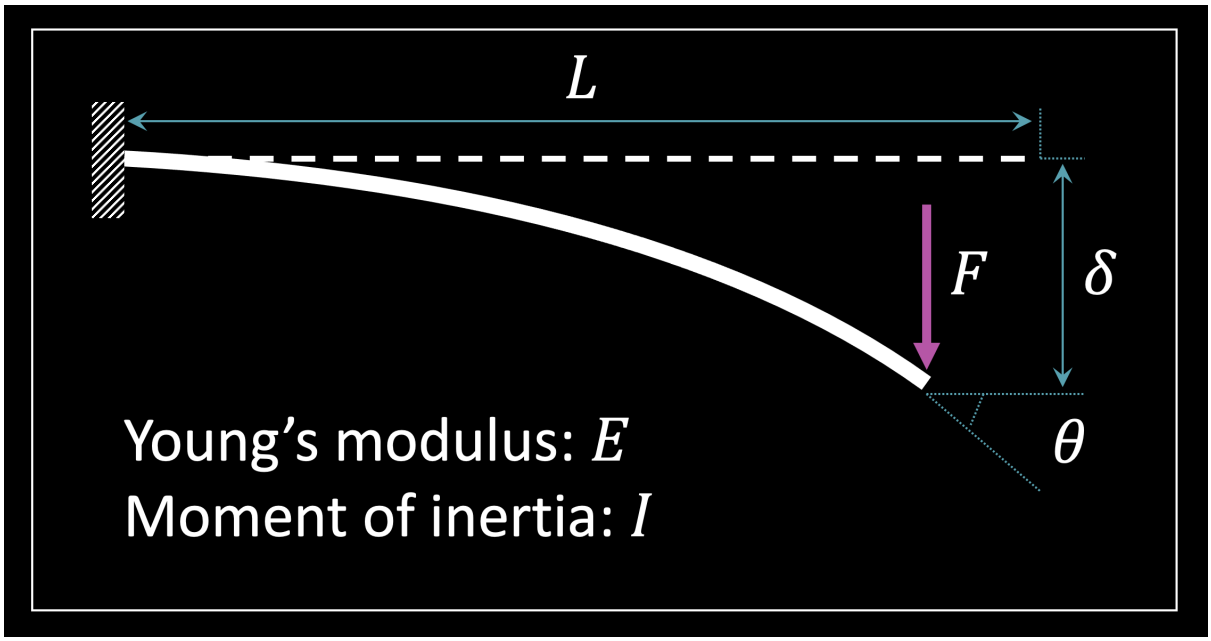
2.c At this point, you have a continuous approximate function in form of a neural network that takes distance as input and outputs the energy. We are interested in finding the equilibrium distance (r^*) between the two particles, which occurs at the minima of the energy.

- Assuming $W \approx f(r)$ where f is the neural network, solve the minimization problem:

$$r^* = \arg\min_r f(r)$$

  Note: The objective function in the above minimization problem is "$\min f$" not "$\min f^2$".
  Tip 1: You may adapt your solution to parts 1.a and 1.b here.
  Tip 2: Remember that the neural networks covered require at least two-dimensional matrix as input. So, if you want to input a single scalar `x` to a neural network, make sure `x` is of the shape `[1,1]` (not `[1]`).
- Clearly state all the settings and parameters used for the optimization.
- Report the equilibrium distance: r^* found.
- Visualize the energy vs. distance function (using a continuous line) and plot the trajectory/path of the optimizer (use line segments to show the trajectory; see figure in Example B).

Questions pertaining to Session 3



Consider a cantilever beam with Young's modulus E, length L, and moment of inertia I. A force F is applied at the tip of the beam and the resulting tip deflection δ and slope θ are measured. The units of each quantity are as following: F[N], L[m, E[GPa], I[cm^4], δ[mm], θ[°].

In the lab session, we trained a neural network to map the quantities $[F, L, E, I]$ to the response $[\delta, \theta]$. We also implemented an optimization scheme to estimate appropriate combination of $\{L, E, I\}$ for a given F and target δ. Recall the two important points that we covered: *(i)* using proper data normalization, and *(ii)* using appropriate variable transformations (e.g., $\tanh(\cdot)$) to convert a constrained optimization problem (where the variables are bounded to certain limits) into an unconstrained optimization problem.

Inverse design problems are ill-posed and admit multiple solutions that satisfy the target property requirements. E.g., several combinations of $\{L, E, I\}$ may give the same δ for a given F. The solution to the optimization problem covered in the lab session resulted in one single combination of $\{L, E, I\}$. The particular output is sensitive to the initial guess in the optimization, i.e., different initial guesses may give different final solutions for $\{L, E, I\}$. Here, we will explore this idea to discover multiple solutions to the given design problem.

The following problem is an extension of the problem covered in the lab session and as such, please **reuse the codes and data** from the session's workbooks and solutions. Please use the same values for `given_F` and `target_delta` (in both normalized/unnormalized coordinates).

3.a Adapt the code for the optimization scheme to estimate $\{L, E, I\}$ for a target δ with given F into a single function `run_beam_optimization(L,E,I)`. The inputs of the function are the initial guesses for $\{L, E, I\}$. Note that you are free to decide if you want the input initial guess to be in normalized/unnormalized and constrained/unconstrained coordinates as long as the inputs to the neural network are normalized and constrained (to e.g., $[-1, 1]$). The function outputs two things: *(i)* the optimal $\{L, E, I\}$ – in **unnormalized and unconstrained coordinates** (i.e., with physical units); *(ii)* `delta_history`: which is a list of δ values at each step of the optimization.
Perform the design optimization using the above function for 10 different initial guesses for $\{L, E, I\}$ chosen randomly between the minimum and maximum of the respective training data.

3.b Plot three histogram distributions of the training data – one each for $\{L, E, I\}$. On each respective histogram, also plot the distribution of the 10 $\{L, E, I\}$ solutions obtained in part 3.a. **Discuss your**

observations.
Tip: When using `plt.hist()` for histogram, pass the argument `density=True` for better visualization, i.e., `plt.hist(·, density=True)`.

3.c Plot the `delta_history` for all the 10 cases (each with different initial guess). Use linear scale for y-axis (i.e., δ) and log scale for x-axis (i.e., `epoch`). Do all the `delta_history` converge to the `target_delta`? Do they all reach `target_delta` in the same number of epochs – why or why not?

Questions pertaining to Session 4

The lab session introduced the spinodoid metamaterials. Note: For the scope of this lab and project, we are interested more in the relationship between the design parameters and mechanical properties of spinodoid metamaterials, rather than the physics of spinodal decomposition.

For visualization of spinodoid metamaterials in this exercise and also the project, you will require MATLAB software. In case of installation troubles, please contact the instructor.

The MATLAB code for this exercise is provided. We will use this code to visualize spinodoid metamaterials for different design parameters. Identify the 5x parameters (`density`, `beta`, `thetaX`, `thetaY`, `thetaZ`) marked with "To-do" in the `spinodoid.m` code. For the provided set of parameter values, the code generates a `.STL` file which can be viewed in any 3D software (e.g., Paraview, Solidworks, Microsoft 3D viewer). For each part, we will be try different values for the 5 parameters. Answer the questions by providing images of the .STL files in the lab report. For each part, clearly indicate the parameter values used. Also, please label/caption the images appropriately with the corresponding parameter values.

4.a Keeping the remaining parameters fixed (of your choice), try different values of `density` between $[0.3, 1]$. Report your observations (with images and brief explanation in your own words) on how it affects the geometry.

4.b Keeping the remaining parameters fixed (of your choice), try different values of `beta` between $[3.0, 7.0]$. Report your observations (with images and brief explanation in your own words) on how it affects the geometry.

4.c Keeping `thetaX`$= 0°$, `thetaY`$= 0°$, and the remaining parameters fixed (of your choice), try different values of `thetaZ` between $[15°, 90°]$. Report your observations (with images and brief explanation in your own words) on how it affects the geometry. How would you classify these designs – lamellar/columnar/cubic?

4.d Keeping `thetaX`$= 15°$, `thetaY`$= 0°$, and the remaining parameters fixed (of your choice), try different values of `thetaZ` between $[15°, 90°]$. Report your observations (with images and brief explanation in your own words) on how it affects the geometry. How would you classify these designs – lamellar/columnar/cubic?

4.e Keeping `thetaX`$= 15°$, `thetaY`$= 15°$, and the remaining parameters fixed (of your choice), try different values of `thetaZ` between $[15°, 90°]$. Report your observations (with images and brief explanation in your own words) on how it affects the geometry. How would you classify these designs – lamellar/columnar/cubic?

For the related paper, see the following link: https://www.nature.com/articles/s41524-020-0341-6 .

Example code snippets

A. Example of a python function taking another python function as argument

```
def function_A(x):
    return 2.0*x

def function_B(x):
    return x+100.0

def function_C(some_function):
    # evaluate some_function at value 1.
    return some_function(1.)

print('function_C(function_A): ',function_C(function_A))
print('function_C(function_B): ', function_C(function_B))
```

```
function_C(function_A):  2.0
function_C(function_B):  101.0
```

B. Example of plot obtained in part 1.b (Note: ignore the annotations below)