1.1 PyTorch Tensor Indexing

We will work with the tensor-objects of PyTorch. They can be understood as multidimensional lists (e.g. vectors, matrices, ...) and are similar to the arrays of NumPy. A tensor can be manipulated in lots of different ways. The tensor syntax is important to understand and needed for solving the following exercises. Therefore, we prepared an small example showing important techniques and properties.
To open the example:

1. Open Google Colab
2. Select *open Notebook* and then the tab *GitHub*
3. Search: TomF98/torchphysics
4. Select the branch: *Workshop* and then Exercise1_1.ipynb

If you want to work on your own laptop, download the notebook from the above link and open it locally.
a) Tensor operations: Given the following tensor t, what do the commands in the below table produce?

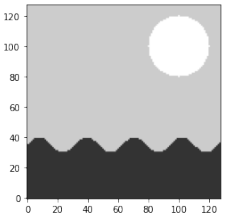
```
t = torch.tensor([[[0, 1], [4, 2], [8, 1]], [[4, 7], [5, 3], [1, 1]]])
```

Command	Output	Command	Output	Command	Output
`t.shape`		`t[1, 2, 0]`		`t[1, 3, 0]`	
`t[0]`		`t[:1]`		`t[:1, :1, :]`	
`t[0].shape`		`t[:1].shape`		`t[:1, :1, :].shape`	
`t[[True, False]]`		`t.reshape(2, 6)`			

Command	`t[1, 2, 0] = 6`	`t[1, :, 0] *= 2`	`t[1, 2:, 0] = 6`	`t[1, -1, 0] += 6`
New Tensor				

b) Indexing with Boolean Values: Indexing a tensor with Boolean values is helpful to modify large and more complex tensors. Here, we want to use this property to change the values of a tensor in specify positions. We start with a blank tensor called `image`, with a width and height of 128 pixels. We want to *draw* the following:

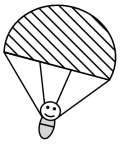
- A sun (disk) at the pixel position $x = 100, y = 100$ with a radius of 20 pixel
- Mountains where the pixel coordinates (x, y) fulfill the condition $35 + 5\sin(0.2x) >= y$



A template for this, is provided at the end of Exercise1_1.ipynb.

1.2 Data driven function approximation

Assume you are jumping from a plane with a parachute, then (under some simplifications) your fall can be described by the following ODE:



$$\begin{aligned} \partial_t^2 u(t) &= D(\partial_t u(t))^2 - g \\ u(0) &= H \\ \partial_t u(0) &= 0 \end{aligned} \tag{1}$$

Here, $u : [0, T] \to \mathbb{R}$ is your height at time t, H the starting height, g the gravity constant, $\partial_t u$ your velocity, $\partial_t^2 u$ your acceleration and D a friction coefficient that is proportional to the cross section of your parachute.
It is possible to analytically solve (1), to obtain the solution

$$u(t; D) = \frac{1}{D}\left(\ln\left(\frac{1 + e^{-2\sqrt{Dg}t}}{2}\right) - \sqrt{Dg}t\right) + H. \tag{2}$$

Our goal is now to train a neural network that, given a value D and time point t, should return the corresponding height $u(t; D)$.
A code template for the following tasks can be found in Exercise1_2.ipynb, open it with the same steps as in exercise 1.1. The tasks below, are also explained inside the notebook.

a) **Creating the Dataset**: For the training of the neural network, we need to create a fitting dataset. Create four tensors `input_training, output_training, input_testing, output_testing`. With shapes and data:

```
input_training.shape = [N_D_train, N_t_train, 2], output_training.shape = [N_D_train, N_t_train, 1]
input_testing.shape = [N_D_test, N_t_test, 2], output_testing.shape = [N_D_test, N_t_test, 1]
input_training[i, k] = (D_i, t_k), output_training[i, k] = u(t_k; D_i) (same for testing)
```

Here, the input tensors contain combinations of t, D (the input of the model) and the output tensors the corresponding expected function values.

b) **Defining the Neural Network**: Build a network, using the `torch.nn` module, that has 2 input neurons for the values of t, D and 1 output neuron for u, two hidden layers of size 20 and Tanh-activation in between.

c) **Writing the Training Loop**: The last step is to create the training loop, where the neural network learns from the data. The example implementation on the PyTorch page is helpful for this task.
Once you have finished the implementation, run the code and start the training. At the end of the notebook an accuracy check is already implemented.

⋆) **Bonus**[1]: How far can one reduce the size of the neural network and the training dataset and still obtain good generalization and approximation by the neural network?

1.3 Physics-informed function approximation

Previously, we used that we know the solution (2) of the ODE (1). Now, we assume that the solution is not analytically known. Thus, we have to utilize the differential equation (1) in the training, which leads us to physics-informed neural networks.
For simplification of the implementation, we start with a fixed value of $D = 0.02$. Again, a notebook outlining the code is prepared: Exercise1_3.ipynb

a) **Working with Autograd**: Verify, with the help of `torch.autograd.grad`, by numerically computing the derivatives and inserting them into the ODE, that the previously given function (2) solves the Equation (1).

b) **Implementing the physics-informed Loss**: Use `torch.autograd.grad` to complete the training loop, by implementing the loss for the differential equation and both initial conditions.

⋆) **Bonus**: At the start we fixed D, extend your code to also work for different D–values similar the exercise 1.2.

⋆) **Bonus**: To compute the derivatives, automatic differentiation (via backpropagation) of PyTorch was used. We could also approximate the derivatives of the neural network with a finite difference scheme. Instead of using `torch.autograd.grad`, change your code to use finite differences to implement the ODE.
Hint: If you have a point grid $t_0, ..., t_N$ with equidistant step size Δt, then:

$$\partial_t u(t_i) \approx \frac{u(t_{i+1}) - u(t_{i-1})}{2\Delta t}, \quad \partial_t^2 u(t_i) \approx \frac{u(t_{i+1}) - 2u(t_i) + u(t_{i-1})}{\Delta t^2}.$$

At the left boundary, use a ghost node u_{-1} that can be determined with the initial velocity condition. For the value $u(t_N)$ you can either not use any condition or try to implement a one sided differential quotient.

[1] We recommend to solve first all other exercises and only then try the bonus tasks.