First Implementation: A Single-Node "Network":

- Optimizing one connection for one input :

$$O=\sigma(\phi w+b),$$

Example : $\sigma((-0.23)w+b)=0.74$, Given an input of −0.23, produce the output 0.74.
⟶ There are 2 unknown parameters w and b and just one equation, so we have infinity of solutions.

To solve this the idea is to find a solution by descending the error surface E defined by

$$E=12(O-0.74)_2,(29)$$

Libraries for Anaconda :

- TensorFlow: is an open source software library for high performance numerical computation. It comes with strong support for machine learning and deep learning and the flexible numerical computation core is used across many other scientific domains.

- the PyTorch deep learning library :
  It has nifty features such as dynamic computational graph construction as opposed to the static computational graphs present in TensorFlow and Keras.

The basics in Pytorch:

- Computational graphs : A computational graph is a set of calculations, which are called *nodes*, and these nodes are connected in a directional ordering of computation.
  The benefits of using a computational graph is that each node is like its own independently functioning piece of code, so it's useful for building neural network operations and making gradient back-propagation
- Tensors : Tensors are matrix-like data structures.

Some code to declare Tensors in Pycorch :

```
import torch
x = torch.Tensor(2, 3)    %tensor of zeros (2 lines,3 columns)
x = torch.rand(2, 3)       %tensor of random floats (2 lines,3 columns)
x = torch.ones(2,3)    %tensor of ones (2 lines,3 columns)
x[:,1]  %return the second column
```

→ Multiplying tensors, adding them and so forth is also available.

- ## Autograd in PyTorch:

The need of a mechanism where error gradients are calculated and back-propagated through the computational graph which is called Autograd in Pycorch.

The Autograd is done via the Variable Class. It contains a Tensor and allows the calculation of the gradients with the call of .backward() function.

- Declaration of Variable :

x = Variable(torch.ones(2, 2) * 2, requires_grad=True).
The requires_grad parameter means if it's true that the calculation of the gradient is allowed and this means that this Variable would be trainable.
If requires_grad = False, the Variable would not be trained.
We can declare Variable by operations on a tensor.
Example: z = 2 * (x * x) + 5 * x this is a Variable.

z.backward(torch.ones(2, 2)) → To calculate the gradient dz/dx.
The gradient is stored in the x Variable, in the property .grad.
When we do x.grad a tensor(2,2) is shown.

- ## Creating a neural network in PyTorch :

Object : create a neural network with 4-layer fully connected to classify the hand-written digits.

Using the base Class : nn.Module and the need of including it's functionality.
Code :

```
import torch.nn as nn
import torch.nn.functional as F
class Net(nn.Module):       % Inheritance from nn.Module Class
   def __init__(self):          %  initialization
      super(Net, self).__init__()    %creation of an instance
```

```
        self.fc1 = nn.Linear(28 * 28, 200)
        self.fc2 = nn.Linear(200, 200)
        self.fc3 = nn.Linear(200, 10)
```

→ A fully connected neural network layer is represented by the nn.Linear object, with the first argument in the definition being the number of nodes in layer l and the next argument being the number of nodes in layer l+1