A short introduction to PyTorch

Author: Margaux Boxho, Junior Research Ingineer at Cenaero.
Date: July 2022.

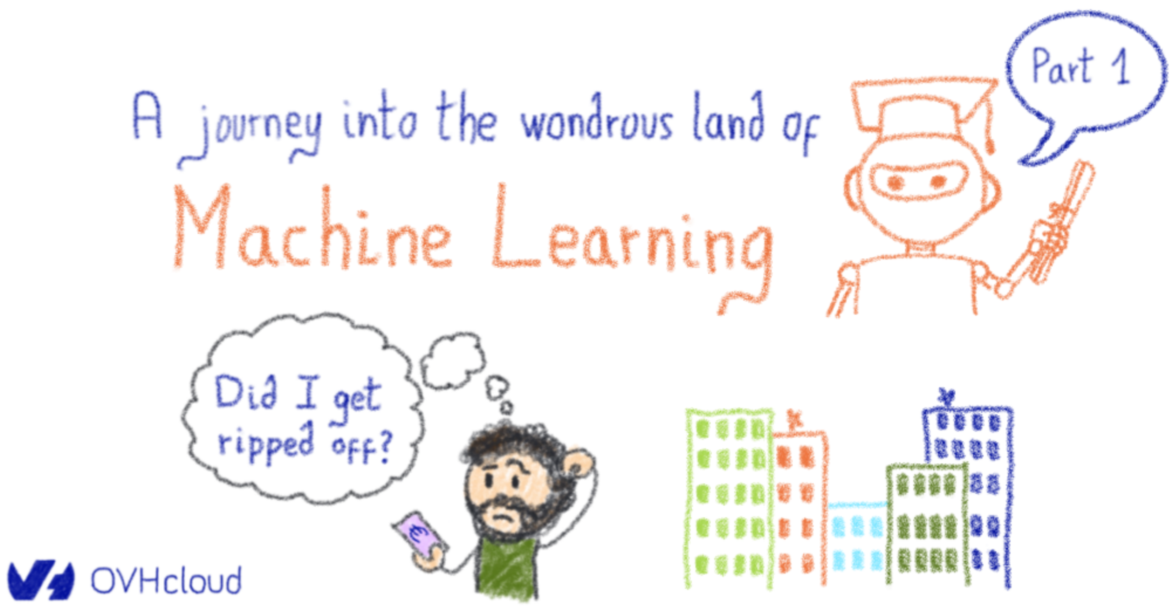


Image credits: https://blog.ovhcloud.com/a-journey-into-the-wondrous-land-of-machine-learning-or-did-i-get-ripped-off-part-1/

For a **first** and **simple** introduction to deep learning, I really recommand you this YouTube channel called **3Blue1Brown**. The guy have made four videos for explaining the concept of deep neural networks and how to train them. These four videos are listed below:

- What is a neural network?
- Gradient descent and How machines learn?
- What is backpropagation really doing?
- The backpropagation computation

This tutorial is about training neural networks with PyTorch. PyTorch is an open source machine learning framework dedicated to both research environemnt and production deployment. It is designed to offer great flexibility and increase the speed of implementation of deep neural networks. It is currently the most popular library for AI researchers and practitioners worldwide, in academia and industry. I will let you dive into this post to learn a bit more about the advantages of PyTorch overother open source libraries.

PyTorch has a large community of users that share their experiments through blogs, GitHub, posts, etc. Once you will face an issue, you will never be alone to solve it. PyTorch also has some very good tutorials ranging from the tensors basics to parallel and distributed learning on GPUs.

Through this tutorial, I will try to condense for you the main concepts of PyTorch (tensors, networks, backpropagation, ...) through simple and didactic examples. I hope you will enjoy this deep learning journey with me. **Let's start ...**

Outline

- Required Packages
- Tensors and basic operations
- `torch.autograd` Package
- Network modules, Optimization and DataLoader
- Train your first MLP!

1. Required Packages

In [1]:
```
# ---
# Torch packages used for the definition of tensors, modules and optimizers
# ---
import torch
import torch.nn as nn
import torch.optim as optim

# ---
# Fundamental package for scientific computing with Python
# ---
import numpy as np

# ---
# Figures and graphics with Python
# ---
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable
plt.rcParams['font.family'] = 'DeJavu Serif'
plt.rcParams['font.serif'] = ['Times'] #['Times New Roman']
```

2. Tensors and basic operations

In PyTorch, tensors are specialized data structures. They are objects from the class torch.Tensor. They can be assimilated to array and matrices and are similar to Numpy's ndarrays. They are used to encore the inputs, outputs of a model and the model's parameters (e.g., weights and bias). Tensors are also optimized for automatic differentiation (see section 4). Let us see,

a) how to construct a tensor in PyTorch?
b) how to extract data from them?
c) how to convert a numpy array to a tensor?
d) what are the tensor attribues?
e) which operations can be performed on tensors?

For a deep review of the various operations including arithmetic, linear algebra, matrix multiplication, sampling and more, you can click on this link

In [2]:
```
# directly from data
data   = [[0,1,2], [3,4,5], [6,7,8]]
x_data = torch.tensor(data)
print("x_data:\n", x_data)

# from a numpy array
np_array = np.array(data)
x_np     = torch.from_numpy(np_array)
print("x_np:\n", x_np)

# from other tensor
x_ones = torch.ones_like(x_data)
print("x_ones:\n", x_ones)

# given specific size
x_zeros = torch.zeros((5,3))
x_rand  = torch.rand((3,4), dtype=torch.float)
m_eye   = torch.eye(10)
print("m_eye:\n", m_eye)
```

```
x_data:
 tensor([[0, 1, 2],
        [3, 4, 5],
        [6, 7, 8]])
x_np:
 tensor([[0, 1, 2],
        [3, 4, 5],
        [6, 7, 8]])
x_ones:
 tensor([[1, 1, 1],
        [1, 1, 1],
        [1, 1, 1]])
m_eye:
 tensor([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 1., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]])
```

In [3]:
```
# attributes of a tensor
print(f"Shape of tensor: {x_rand.shape}")
print(f"Size of tensor: {x_rand.size()}")
print(f"Datatype of tensor: {x_rand.dtype}")
print(f"Device where the tensor is stored on: {x_rand.device}")
```

```
Shape of tensor: torch.Size([3, 4])
Size of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device where the tensor is stored on: cpu
```

In [4]:

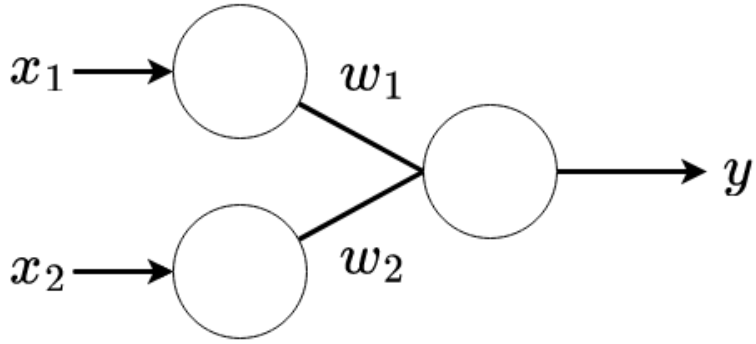
```
# dtype conversion
print("Before conversion, Datatype of x_data:", x_data.type())
x_data_real = x_data.float()
print("After  conversion, Datatype of x_data:", x_data_real.type())
print("-----------------------------------------------------------")

# sum, subtract, mult, divide
v1 = torch.randint(low=-100, high=100, size=[5])
v2 = torch.randint(low=-100, high=100, size=[5])
print("v1:\n", v1)
print("v2:\n", v2)

print("sum:\n",v1+v2)
print("sub:\n",v1-v2)
print("mult:\n",v1*v2)
print("div:\n",v1/v2)
print("-----------------------------------------------------------")

# slicing vector
print("Slice:\n",v1[2:4])
print("-----------------------------------------------------------")

# squeeze and unsqueeze
print('Before unsqueeze:\n ', m_eye.shape)
m_eye = m_eye.unsqueeze(0)
print('After unsqueeze:\n', m_eye.shape)
m_eye = m_eye.squeeze(0)
print('After squeeze:\n', m_eye.shape)
print("-----------------------------------------------------------")

# expand
v   = torch.arange(1, 7)
m_v = v.unsqueeze(1).expand(-1, 4)
print("Expand:\n", m_v)
print("-----------------------------------------------------------")

# there is no reshape() with torch.Tensor but view() instead
images = torch.randn(10, 3, 256, 256)
images_as_vectors = images.view(10, -1)
print("Shape of image:", images.shape, "\nShape after vector transformation:"
print("-----------------------------------------------------------")

# permutation
images_permute = images.permute((1,0,2,3))
print("Shape of image:", images.shape, "\nShape after permutation:", images_p
print("-----------------------------------------------------------")

# matrix multiplication
mt1 = torch.tensor([[5, 2],[0, 1]], dtype=torch.float)
mt2 = torch.tensor([[2, -2],[10, 1]], dtype=torch.float)
print("Matrix multiplication:\n", mt1@mt2)

# mean computation
print("Mean computation along axis=3\n", images.mean(3).shape)
```

```
Before conversion, Datatype of x_data: torch.LongTensor
After  conversion, Datatype of x_data: torch.FloatTensor
-----------------------------------------------------------
```

```
v1:
 tensor([-80,   5, -57,   1,  -6])
v2:
 tensor([ 47, -50,  83, -66, -88])
sum:
 tensor([-33, -45,  26, -65, -94])
sub:
 tensor([-127,   55, -140,   67,   82])
mult:
 tensor([-3760,  -250, -4731,   -66,   528])
div:
 tensor([-1.7021, -0.1000, -0.6867, -0.0152,  0.0682])
----------------------------------------------------------
Slice:
 tensor([-57,   1])
----------------------------------------------------------
Before unsqueeze:
  torch.Size([10, 10])
After unsqueeze:
 torch.Size([1, 10, 10])
After squeeze:
 torch.Size([10, 10])
----------------------------------------------------------
Expand:
 tensor([[1, 1, 1, 1],
        [2, 2, 2, 2],
        [3, 3, 3, 3],
        [4, 4, 4, 4],
        [5, 5, 5, 5],
        [6, 6, 6, 6]])
----------------------------------------------------------
Shape of image: torch.Size([10, 3, 256, 256])
Shape after vector transformation: torch.Size([10, 196608])
----------------------------------------------------------
Shape of image: torch.Size([10, 3, 256, 256])
Shape after permutation: torch.Size([3, 10, 256, 256])
----------------------------------------------------------
Matrix multiplication:
 tensor([[30., -8.],
        [10.,  1.]])
Mean computation along axis=3
```

3. `torch.autograd` Package

The training of a (deep) neural network is based on the **gradient descent**. During training, the learnable parameters are adjusted according to the minimisation of a given loss function $\mathcal{L}$. This loss function must therefore be derived w.r.t. each network parameter.



To better understand backpropagation, it is first useful to develop an intuition about the relationship between the actual output of a neuron and the correct output for a specific learning example. Let us considere a regression problem using a simple neural network with two inputs x_1 and x_2 and one single output y (see figure above). For a regression problem, there is no activation layer in the output layer and the loss function is typically a mean square error (MSE) defined as follows,

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$$

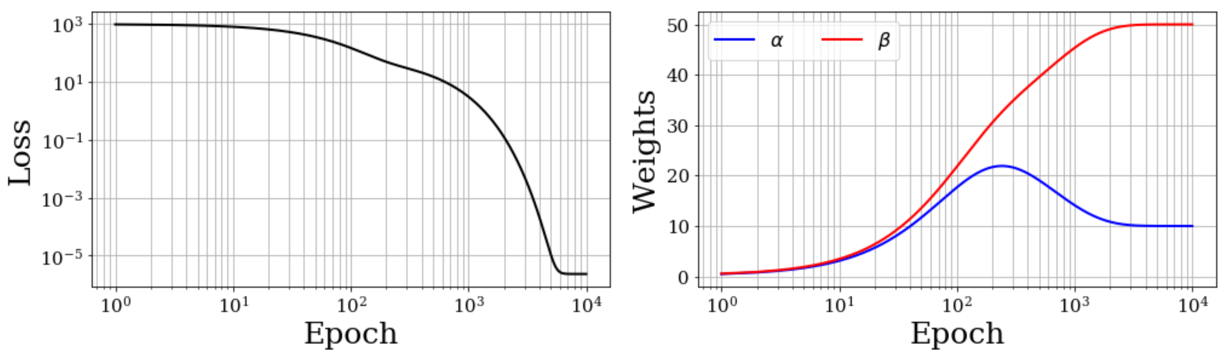
where $\hat{y}_i$ is the network prediction, y_i is the ground truth and N is the number of training data.

In [5]:

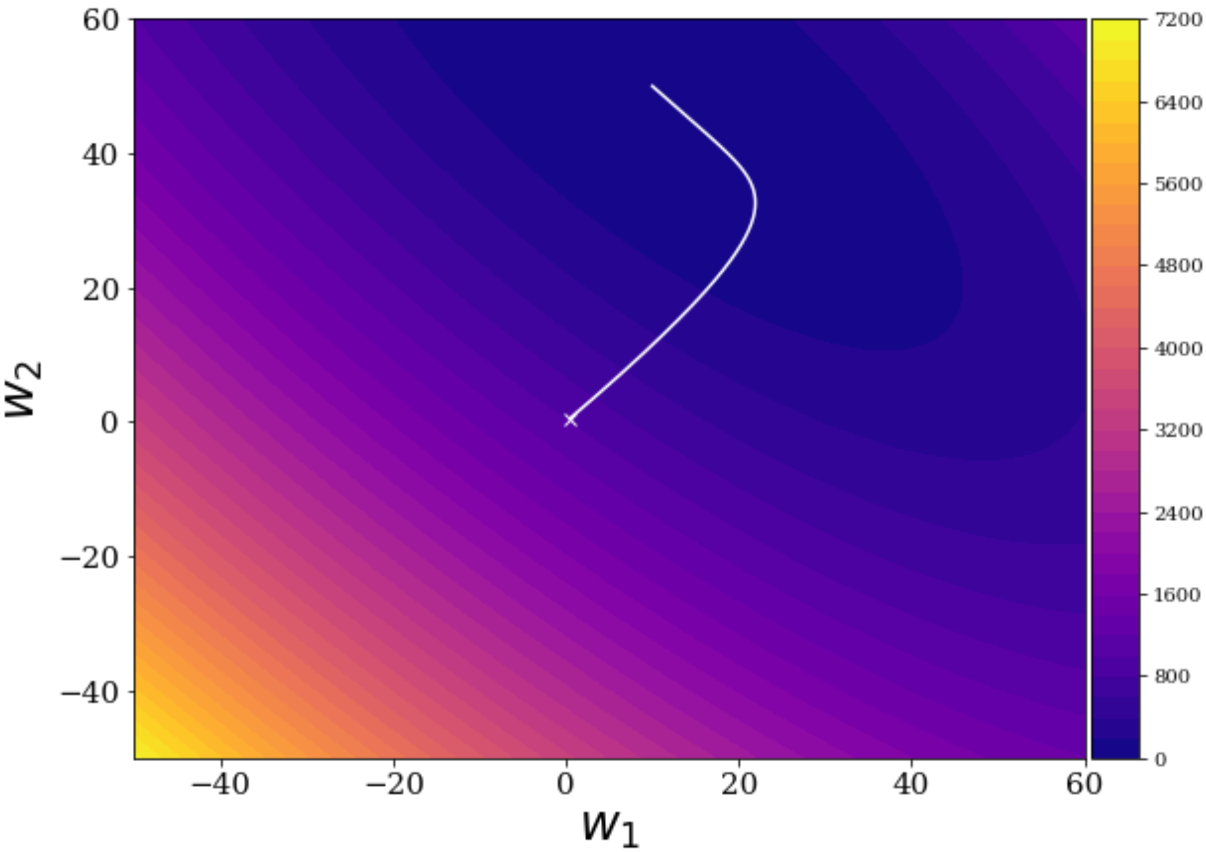
```
# ---
# Loss function
# ---
def MSE(x1,x2,y,w1,w2):
    return np.mean((w1*x1 + w2*x2 - y)**2)

# ---
# Derivative of the loss w.r.t w1 and w2
# ---
def dMSE(x1,x2,y,w1,w2):
    dldw1 = np.mean(2*x1*(w1*x1+w2*x2-y))
    dldw2 = np.mean(2*x2*(w1*x1+w2*x2-y))
    return dldw1, dldw2

# ---
# Training data
# ---
x1      = np.random.rand(100)
x2      = np.random.rand(100)

# ---
# Parameters to fit
# ---
alpha   = 10
beta    = 50

# ---
# Small additional error
# ---
epsilon = np.random.rand(100)*0.005

# ---
# Outputs
# ---
y       = x1*alpha + x2*beta + epsilon

# ---
# Training though gradient descent
# ---
nbEpochs = 10000
w1       = np.zeros(nbEpochs+1)
w1[0]    = np.random.rand(1)
w2       = np.zeros(nbEpochs+1)
w2[0]    = np.random.rand(1)
loss     = np.zeros(nbEpochs)
step     = 0.01

for epoch in range(nbEpochs):
    loss[epoch]  = MSE(x1,x2,y,w1[epoch],w2[epoch])
    dldw1, dldw2 = dMSE(x1,x2,y,w1[epoch],w2[epoch])
    w1[epoch+1]  = w1[epoch] - step*dldw1
    w2[epoch+1]  = w2[epoch] - step*dldw2

print("End of the training.")
```

```
End of the training.
```

In [6]:
```
fig, ax = plt.subplots(1,2, constrained_layout=True)
ax[0].loglog(np.arange(1,1+nbEpochs), loss, 'k-', linewidth=2)

ax[0].set_ylabel("Loss", fontsize=25)
ax[0].set_xlabel("Epoch", fontsize=25)
ax[0].grid(True, which='both')

ax[1].semilogx(np.arange(1,1+nbEpochs+1), w1, 'b-', linewidth=2, label="$\\al
ax[1].semilogx(np.arange(1,1+nbEpochs+1), w2, 'r-', linewidth=2, label="$\\be

ax[1].legend(loc="best", fontsize=16, ncol=2)
ax[1].set_ylabel("Weights", fontsize=25)
ax[1].set_xlabel("Epoch", fontsize=25)
ax[1].grid(True, which='both')

for i in range(2):
    for tick in ax[i].get_xticklabels():
        tick.set_fontsize(15)

    for tick in ax[i].get_yticklabels():
        tick.set_fontsize(15)

fig.set_size_inches(14,4)
plt.show()
```



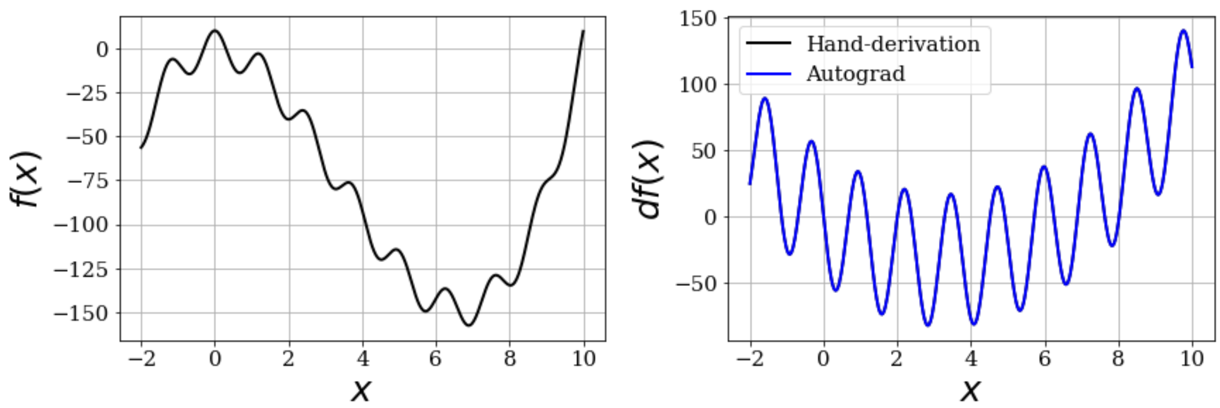
In [7]:
```
nw1    = 500; nw2    = 500
W1, W2 = np.meshgrid(np.linspace(-50,60,nw1), np.linspace(-50,60,nw2))
LOSS   = np.zeros(W1.shape)
for i in range(nw1):
    for j in range(nw2):
        LOSS[i,j] = MSE(x1,x2,y,W1[i,j],W2[i,j])

fig, ax = plt.subplots(1,1, constrained_layout=True)
im = ax.contourf(W1, W2, LOSS, levels=40, cmap='plasma')
ax.plot(w1,w2,'w-')
ax.plot(w1[0],w2[0],'wx')

ax.set_xlabel("$w_1$", fontsize=25)
ax.set_ylabel("$w_2$", fontsize=25)

for tick in ax.get_xticklabels():
    tick.set_fontsize(15)

for tick in ax.get_yticklabels():
    tick.set_fontsize(15)

divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="5%", pad=0.05)

plt.colorbar(im, cax=cax)

fig.set_size_inches(8,6)
plt.show()
```



Deriving the full gradient is easy for the case treated above but for classical deep neural networks containing billion of parameters, it would be a nightmare. This is where **backpropagation** comes in. Essentially, backpropagation is an algorithm to train neural networks based on the chain rule, which makes it more efficient than any naive direct computation of the gradient with respect to each individual weight. To compute those gradients, PyTorch uses automatic differentiation through the build-in differentiation engine called `torch.autograd` . The autograd package creates a computational graph that can be later used to compute derivatives of the output quantities w.r.t the input and other intermediate computations steps. If you want to go into a gentle introduction to `torch.autograd` by PyTorch, I encourage you to read this. For the moment, let us see how `torch.autograd` can be used to derive an analytical function.

In [8]:

```
def f(x):
    return x**3 + 10*torch.cos(5*x) - 10*x**2

def df(x):
    return 3*x**2 - 50*torch.sin(5*x) -20*x

fig, ax = plt.subplots(1,2, constrained_layout=True)
x = torch.linspace(-2,10,500)
ax[0].plot(x.detach().numpy(), f(x).detach().numpy(), 'k-', linewidth=2)

df_autograd = torch.zeros(x.shape)
for i in range(x.shape[0]):
    xx = x[i].clone().detach().requires_grad_(True)
    df_autograd[i] = torch.autograd.grad(f(xx), xx)[0].item()

ax[1].plot(x.detach().numpy(), df(x).detach().numpy(), 'k-', linewidth=2, lab
ax[1].plot(x.detach().numpy(), df_autograd.detach().numpy(), 'b-', linewidth=

ax[0].set_ylabel("$f(x)$", fontsize=25)
ax[1].set_ylabel("$df(x)$", fontsize=25)
ax[1].legend(loc="best", fontsize=15, ncol=1)

for i in range(2):
    ax[i].grid(True)
    ax[i].set_xlabel("$x$", fontsize=25)

    for tick in ax[i].get_xticklabels():
        tick.set_fontsize(15)

    for tick in ax[i].get_yticklabels():
        tick.set_fontsize(15)

fig.set_size_inches(12,4)
plt.show()
```



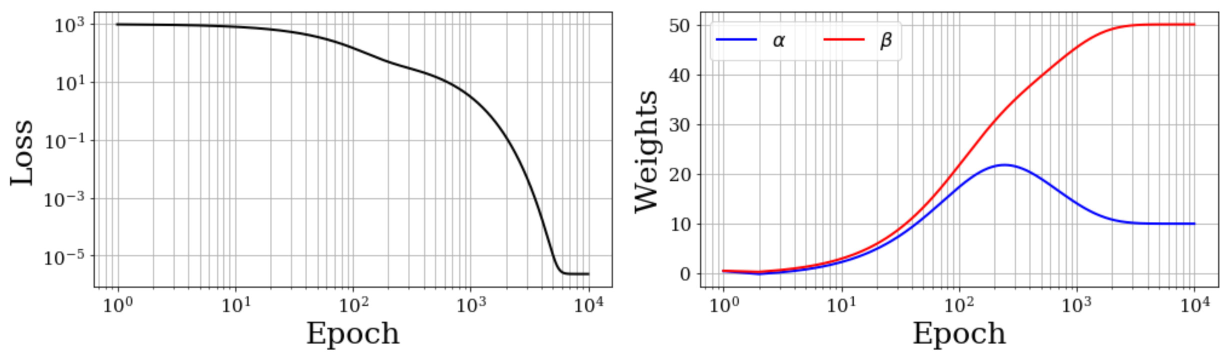
Now we can reuse `torch.autograd` to optimise the previous regression porblem. Instead of simply using `torch.autograd.grad` on the MSE, we can directly go one step further by using the function `.backward()` without forgetting to reset the values of the gardients to zero with the `.zero_()` function.

In [9]:

```
def parametric_function(x1,x2,w1,w2):
    return x1*w1 + x2*w2

# ---
# Transfer to torch.Tensor
# ---
x1_torch = torch.from_numpy(x1)
x2_torch = torch.from_numpy(x2)
y_torch  = torch.from_numpy(y)
param    = torch.randn(2,requires_grad=True)
# ---
# Training though gradient descent
# ---
nbEpochs      = 10000
w1_auto       = np.zeros(nbEpochs+1)
w2_auto       = np.zeros(nbEpochs+1)
loss_auto     = np.zeros(nbEpochs)
lr            = 0.01

for epoch in range(nbEpochs):
    y_pred            = parametric_function(x1_torch,x2_torch,param[0],param[1
    mse               = ((y_torch-y_pred)**2).mean()
    loss_auto[epoch] = mse.item()
    mse.backward()
    param.data      -= lr*param.grad.data
    param.grad.data.zero_()
    w1[epoch+1]   = param.data[0].item()
    w2[epoch+1]   = param.data[1].item()

print("End of the training.")
```

```
End of the training.
```

In [10]:
```
fig, ax = plt.subplots(1,2, constrained_layout=True)
ax[0].loglog(np.arange(1,1+nbEpochs), loss, 'k-', linewidth=2)

ax[0].set_ylabel("Loss", fontsize=25)
ax[0].set_xlabel("Epoch", fontsize=25)
ax[0].grid(True, which='both')

ax[1].semilogx(np.arange(1,1+nbEpochs+1), w1, 'b-', linewidth=2, label="$\\al
ax[1].semilogx(np.arange(1,1+nbEpochs+1), w2, 'r-', linewidth=2, label="$\\be

ax[1].legend(loc="best", fontsize=16, ncol=2)
ax[1].set_ylabel("Weights", fontsize=25)
ax[1].set_xlabel("Epoch", fontsize=25)
ax[1].grid(True, which='both')

for i in range(2):
    for tick in ax[i].get_xticklabels():
        tick.set_fontsize(15)

    for tick in ax[i].get_yticklabels():
        tick.set_fontsize(15)

fig.set_size_inches(14,4)
plt.show()
```



We end up with the same results but no derivative of the loss has been written down to train this regression probleme. The gradients are computed through `mse.backward()` and the parameters are ajusted accordingly with a given learning rate `lr`. Here, the learning rate is kept constant throughout the learning process. However, there are procedures to adapt it dynamically during learning. We will see this later in Section 5.

4. Network modules, Optimization and DataLoader

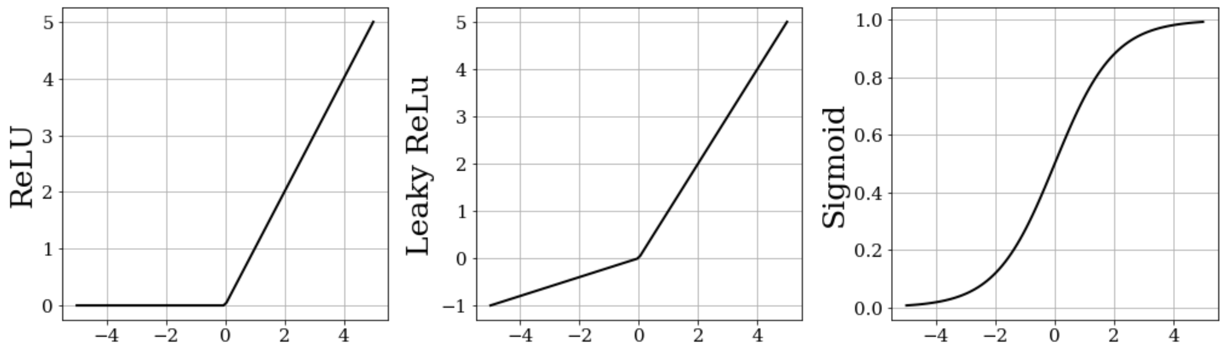
Now that we have learn how to use `torch.autograd` to optimize a regression problem, we still need to construct the neural networks as LEGO blocks. Indeed, PyTorch offers a wide library of basic building blocks (e.g., activation layer, pooling, shuffle, normalization, convolution layer, linear layer, module, ...) for graphs through `torch.nn` (see here). Let us dive into these packages to construct our first neural network.

4.1. `nn.Functional`

It contains convolution functions, pooling functions, non-linear activation functions, linear function, dropout functions, sparse functions, distance functions, loss functions, vision functions and dataParallel functions (multi-GPU, distributed). You can have an access to the complete list here. All these functions are the basic blocks to construct our own neural network. Let us have a look at various activation functions.

In [17]:

```
fig, ax = plt.subplots(1,3, constrained_layout=True)

x = torch.linspace(-5, 5, 100)

y1 = nn.functional.relu(x)
ax[0].plot(x, y1, 'k-', linewidth=2)
ax[0].set_ylabel("ReLU", fontsize=25)

y2 = nn.functional.leaky_relu(x, negative_slope=.2)
ax[1].plot(x, y2, 'k-', linewidth=2)
ax[1].set_ylabel("Leaky ReLu", fontsize=25)

y3 = torch.sigmoid(x) # nn.functional.sigmoid is deprecated. Use torch.sigmoi
ax[2].plot(x, y3, 'k-', linewidth=2)
ax[2].set_ylabel("Sigmoid", fontsize=25)

for i in range(3):
    ax[i].grid(True)
    for tick in ax[i].get_xticklabels():
        tick.set_fontsize(15)

    for tick in ax[i].get_yticklabels():
        tick.set_fontsize(15)

fig.set_size_inches(14,4)
```



4.2. nn.Module and nn.Sequential

The class `nn.Module` is used to build complex neural networks. Any new networks (or sub-class) which inherits from `nn.Module`, will automatically keep track of the parameters of their components (or properties). To define such a sub-class, it is required to implement the `forward()` method and the constructor `__init__()`. Let us dive into an example.

In [24]:
```
class MyLinearModel(nn.Module):
    def __init__(self, in_features, out_features):
        # --- inheritance
        super(MyLinearModel, self).__init__()

        # --- parameters
        self.in_features  = in_features
        self.out_features = out_features

        # --- linear layer
        self.w = nn.Linear(in_features=self.in_features,
                           out_features=self.out_features,
                           bias=True)
    # --- forward pass
    def forward(self, x):
        return self.w(x)
```

In [25]:
```
# instantiate an object of MyLinearModel with its two inputs arguments
linearModel = MyLinearModel(4,2)

# nn.Module offers useful instructions such as user-friendly printing
# of the module which summarize the modules contained in it:
print(linearModel)

# look at the parameters themselves
for param in linearModel.parameters():
    print(param)
```

```
MyLinearModel(
  (w): Linear(in_features=4, out_features=2, bias=True)
)
Parameter containing:
tensor([[-0.1879,  0.0553,  0.0924,  0.3530],
        [ 0.1341, -0.1475, -0.4913, -0.4788]], requires_grad=True)
Parameter containing:
tensor([ 0.3331, -0.1007], requires_grad=True)
```

Now if we want to construct more complex neural network, we will use `nn.Sequential` that automatically chains modules with each others. Let us construct an multi-layer perceptron (MLP) containing L layers composed of h_l neurons in the layer l with an arbitrary input and output size (`in_size` and `out_size`).

In [29]:
```
class MySequentialMLP(nn.Module):
    def __init__(self, in_size, hidden_units, out_size):
        super(MySequentialMLP, self).__init__()

        self.hidden_units = hidden_units
        modules           = []

        # --- input layer
        modules.append(nn.Linear(in_size, hidden_units[0]))
        modules.append(nn.ReLU())

        # --- hidden layers
        for i in range(len(hidden_units)-1):
            modules.append(nn.Linear(hidden_units[i], hidden_units[i+1]))
            modules.append(nn.ReLU())

        # --- output layer
        modules.append(nn.Linear(hidden_units[-1], out_size))

        self.net = nn.Sequential(*modules)

    # --- forward pass
    def forward(self, x):
        out = self.net(x)
        return out
```

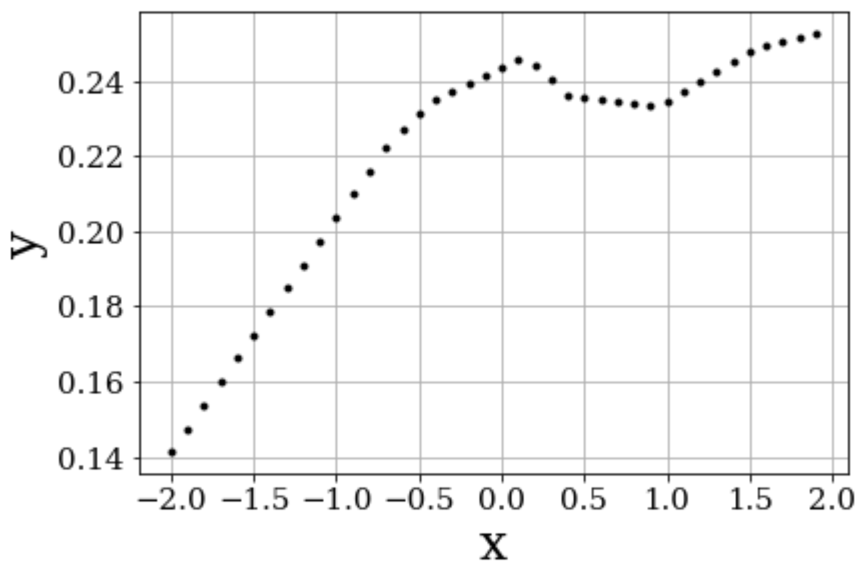
In [31]:
```
# ---
# Get the number of parameters of each trained neural network
# ---
import functools
import operator

# ---
# https://gist.github.com/ihoromi4/aa16085532358f9fc7937941526d827c
# ---
def get_n_params(model: nn.Module) -> int:
    return sum((functools.reduce(operator.mul, p.size()) for p in model.param
```

In [33]:
```
in_size      = 1
out_size     = 1
hidden_units = [10,15,5]
mlp          = MySequentialMLP(in_size,hidden_units,out_size)
print("MLP:", mlp)
print("Number of learnable parameters: %d" %(get_n_params(mlp)))
```

```
MLP: MySequentialMLP(
  (net): Sequential(
    (0): Linear(in_features=1, out_features=10, bias=True)
    (1): ReLU()
    (2): Linear(in_features=10, out_features=15, bias=True)
    (3): ReLU()
    (4): Linear(in_features=15, out_features=5, bias=True)
    (5): ReLU()
    (6): Linear(in_features=5, out_features=1, bias=True)
  )
)
Number of learnable parameters: 271
```

In [38]:

```
# batch of inputs are usually given to train a neural network, because the ne
# we need to reshape the tensor to get torch.Size([40, 1]) compatible with th
x = torch.arange(-2, 2, .1).unsqueeze(1)

# the forward pass is performed and to plot the output, we need to use detach
# as numpy matrix (which is implicitely made when you plot a tensor).
y = mlp(x).detach()

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

ax.plot(x, y, 'k.', linewidth=2)

ax.set_xlabel("x", fontsize=25)
ax.set_ylabel("y", fontsize=25)

ax.grid(True)

for tick in ax.get_xticklabels():
    tick.set_fontsize(15)

for tick in ax.get_yticklabels():
    tick.set_fontsize(15)
```



4.3. `torch.optim`

We have learn how to construct a neural network using a combination of linear layer and activation function. We have aslo learn how to optimize a very simple network using the gradient descent and the `torch.autograd` package. As explained, the computation of the gradient is needed to update the parameters of the neural network and to converge through an optimal solution (not necessary a global minimum). This update procedure is performed by `torch.optim` (here).

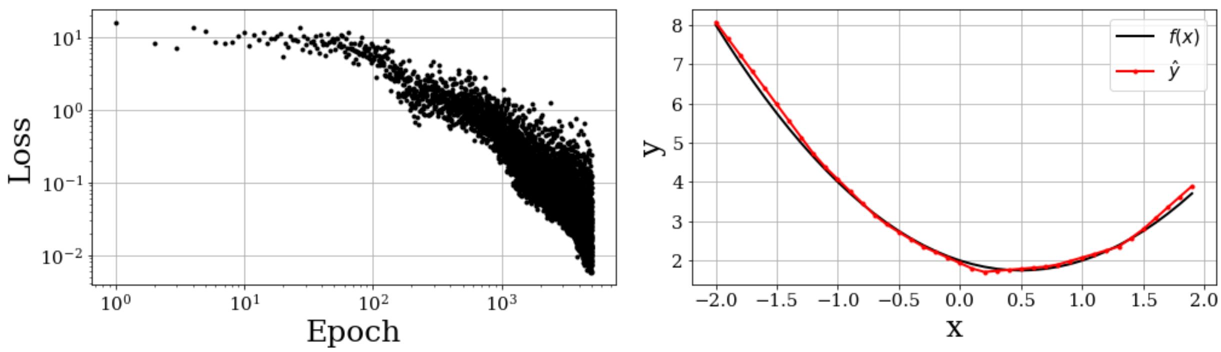
According to the documentation of PyTorch: `torch.optim` *is a package implementing various optimization algorithms. Most commonly used methods are already supported, and the interface is general enough, so that more sophisticated ones can be also easily integrated in the future.*

Let us consider an example to illustrate the behavior of the stokastic gradient descent (SGD) for the fitting of a quadratic function $y := f(x) = x^2 - x + 2$

In [40]:

```
# unknown function
def fun(x):
    return x**2 - x + 2

# creation of an instance of the SGD class
sgd_optimizer = optim.SGD(params=mlp.parameters(), lr=.001)

# learing steps
nbEpochs = 5000

loss = np.zeros(nbEpochs)

for i in range(nbEpochs):
    # randomly generate x values
    x = torch.randn(100, 1)

    # prediction
    y_pred = mlp(x)

    #  set all the grad values of the parameters of our net to zero
    sgd_optimizer.zero_grad()

    # computation of the loss and its gradient
    loss_item = ((fun(x) - y_pred)**2).mean()
    loss_item.backward()

    # always monitor your loss !
    loss[i] = loss_item.item()

    # update of the parameters
    sgd_optimizer.step()
```

In [50]:
```
fig, ax = plt.subplots(1,2, constrained_layout=True)

ax[0].loglog(np.arange(1,nbEpochs+1), loss, 'k.', linewidth=2)
ax[0].set_xlabel("Epoch", fontsize=25)
ax[0].set_ylabel("Loss", fontsize=25)

x = torch.arange(-2, 2, .1).unsqueeze(1)
y = mlp(x).detach()

ax[1].plot(x, fun(x), 'k-', linewidth=2, label='$f(x)$')
ax[1].plot(x, y, 'r.-', linewidth=2, label='$\hat{y}$')
ax[1].legend(loc="upper right", fontsize=15)

ax[1].set_xlabel("x", fontsize=25)
ax[1].set_ylabel("y", fontsize=25)

for i in range(2):
    ax[i].grid(True)

    for tick in ax[i].get_xticklabels():
        tick.set_fontsize(15)

    for tick in ax[i].get_yticklabels():
        tick.set_fontsize(15)

fig.set_size_inches(14,4)
```



4.4. DataLoader

It is not a good training because the loss oscillates a lot. We can reduce these oscillations by changing the optimizer but also by correctly defined the training data and the test data. Indeed, while training a (deep) neural network, you need to have three distinct sets:

1. a **training** set,
2. a **test** set and,
3. a **validation** set.

The training and the test sets are used during the optimization of the network. The **training set** contains the data that are used to fit the learnable parameters while the **test set** contains data that are used to evaluate the performance of the current parameters. We hence obtain a training loss and a testing loss. In a good training, the training and testing losses should be closed to each other and should decrease with the number of epochs. However, at some point, the testing loss will start to increase. There, your neural network overfit the data (called the **overfitting** regime). Indeed, the network starts to learn the noise into the training data and it leads to a deteriorization of the prediction on the training set. The underfitting and overfitting regime are more generally called the bias-variance trade-off in Machine Learning. This issue can be depicted as in the following Figure 1, where the horizontal axis represents the model complexity (number of hidden layers, number of neurons, degrees of interpolation, etc). The bias error is relatively high at the beginning of the training and then decreases progressively. On the contrary, the variance error depicts the reverse behavior, it is quite low at the beginning and then increases during the training. There is a balance to find between bias and variance error. Underfitting data (on the left, in light blue box) have the highest bias error. Those data are less variable (see graph on the left lower corner). They exhibit a large training error coupled with a high bias error. On the other hand, overfitting data (on the right, light yellow box) result in a low bias and high variance error. The model is too complex and tries to capture the noise and so other nonphysical information. Such a complex model will show very good behavior over the training data but will be bad at generalization. The optimum lies in the red box, where the sum of the bias error and variance error is minimized.

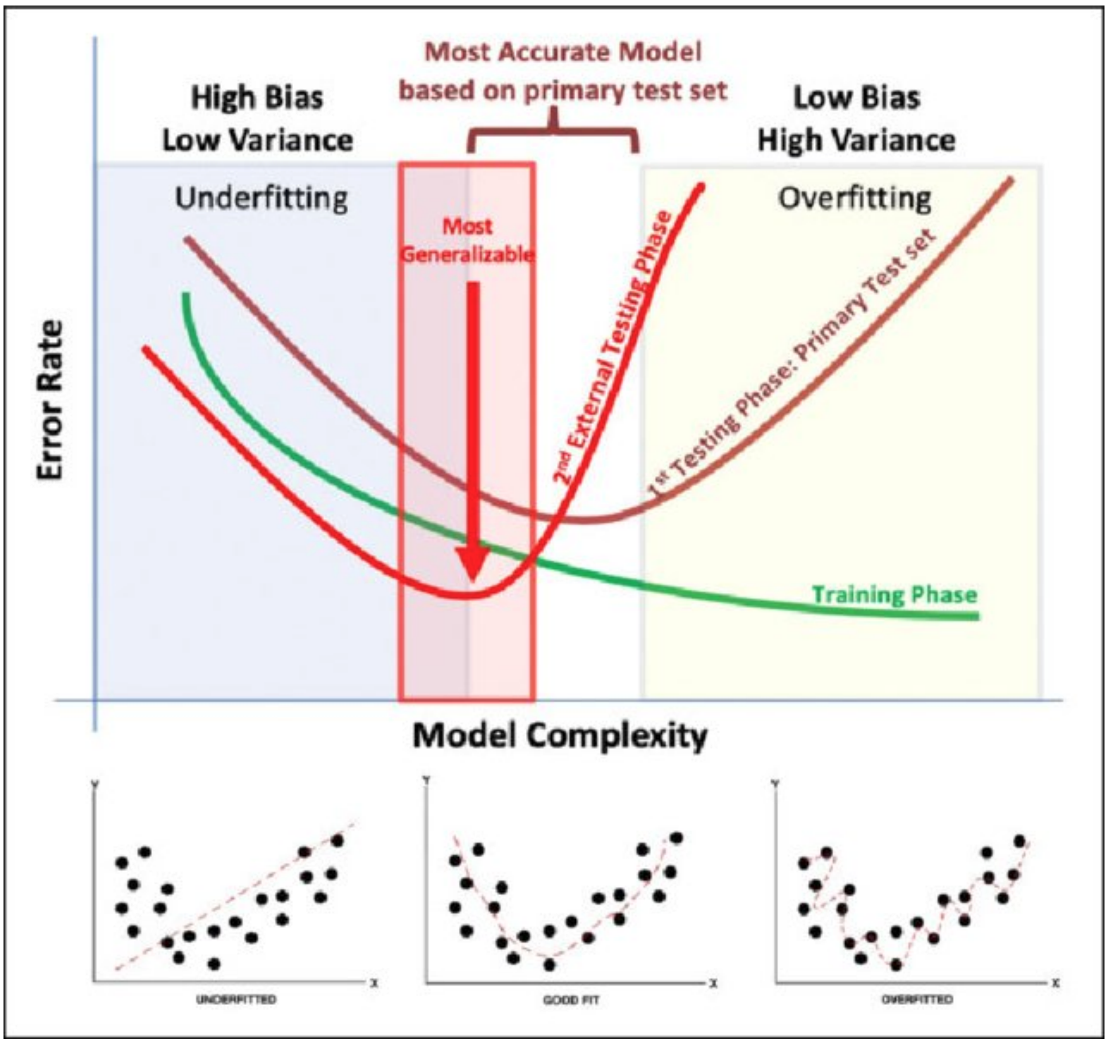


Figure 1. Bias-variance trade-off in machine learning extracted from Rashidi, 2019 [1].

The **validation set** is used to evaluate a priori the behavior our your model. The data it contains is neither training set data nor test set data. It can be data of the *same nature* or data extracted from another set up to evaluate the robustness and the generalization capabilities of the neural network.

There exists online databases that you can download and use as it is. However, most of the time, you have our own data and you want to make them accessible for the training. PyTorch offers the `DataLoader` class. For a complete review of `DataLoader` , I encourage you to read this. To summarize quicky, it exists two different type of datasets: (a) the map-style datasets, and (b) iterable-style datasets. According to the documentation,

(a) *A map-style dataset is one that implements the `__getitem__()` and `__len__()` protocols, and represents a map from (possibly non-integral) indices/keys to data samples.*

(b) *An iterable-style dataset is an instance of a subclass of IterableDataset that implements the `__iter__()` protocol, and represents an iterable over data samples. This type of datasets is particularly suitable for cases where random reads are expensive or even improbable, and where the batch size depends on the fetched data.*

Let us construct an iteratble-style dataset for the fitting of our quadratique function.

In [97]:
```
class MyIterableDataset(torch.utils.data.IterableDataset):
    def __init__(self, x, perc, train=True):
        super(MyIterableDataset).__init__()

        self.nbData = x.shape[0]
        self.x      = x
        self.y      = fun(self.x).view((self.nbData,1))

        self.train  = train
        self.test   = not(train)
        self.size   = int(self.nbData*(1-perc))*self.test + int(self.nbData*p
        self.offset = int(self.nbData*(1-perc))*self.train


    def __iter__(self):
        worker_info = torch.utils.data.get_worker_info()
        if worker_info is None:  # single-process data loading, return the fu
            iter_start = self.offset
            iter_end   = self.offset + self.size
        else:
            print("not implemented, read the documentation")
        index = torch.arange(iter_start,iter_end)
        buf   = torch.cat((self.x[index,:], self.y[index,:]), 1)
        return iter(buf)
```

In [116…

```
nbData = 2000
x      = torch.linspace(-2,2,nbData).view((nbData,1))
rand_indx = torch.randperm(len(x))

trainData   = MyIterableDataset(x[rand_indx], perc=0.9, train=True)
trainLoader = torch.utils.data.DataLoader(trainData, batch_size=256, num_work

testData   = MyIterableDataset(x[rand_indx], perc=0.9, train=False)
testLoader = torch.utils.data.DataLoader(testData, batch_size=256, num_worker
```

In [117…

```
# computation of the test loss
def test_loss(testLoader):
    test_loss = 0
    nb        = 0
    for data in testLoader:
        batch_sz = data.shape[0]
        x        = data[:,0].view((batch_sz,1))
        y        = data[:,1].view((batch_sz,1))
        y_pred   = mlp(x)
        batch_loss = ((y - y_pred)**2).mean()
        test_loss += batch_loss.detach()
        nb         += 1

    return test_loss/nb

# model
mlp = MySequentialMLP(in_size,hidden_units,out_size)

# creation of an instance of the SGD class
sgd_optimizer = optim.SGD(params=mlp.parameters(), lr=.001)

# learing steps
nbEpochs = 5000

loss     = np.zeros(nbEpochs)
lossTest = np.zeros(nbEpochs)

for i in range(nbEpochs):
    mlp.train()

    train_loss = 0
    nb         = 0
    for data in trainLoader:
        batch_sz = data.shape[0]
        x        = data[:,0].view((batch_sz,1))
        y        = data[:,1].view((batch_sz,1))
        y_pred   = mlp(x)

        #  set all the grad values of the parameters of our net to zero
        sgd_optimizer.zero_grad()

        # computation of the loss and its gradient
        batch_loss = ((y - y_pred)**2).mean()
        batch_loss.backward()

        # update of the parameters
        sgd_optimizer.step()

        # save loss
        train_loss += batch_loss.detach()
        nb         += 1

    # always record your loss
    loss[i]     = train_loss/nb
    lossTest[i] = test_loss(testLoader)
    if(i%10==0):
        print("Epoch: %6d, train_loss=%.6e, test_loss=%.6e" %(i,loss[i],lossT
```

```
Epoch:      0, train_loss=1.359354e+01, test_loss=1.214143e+01
Epoch:     10, train_loss=8.636514e+00, test_loss=7.427392e+00
Epoch:     20, train_loss=2.507225e+00, test_loss=1.984247e+00
Epoch:     30, train_loss=1.084867e+00, test_loss=1.064852e+00
Epoch:     40, train_loss=9.236277e-01, test_loss=9.135685e-01
Epoch:     50, train_loss=8.042253e-01, test_loss=7.888074e-01
Epoch:     60, train_loss=7.050731e-01, test_loss=6.857672e-01
Epoch:     70, train_loss=6.184344e-01, test_loss=5.974071e-01
Epoch:     80, train_loss=5.410020e-01, test_loss=5.195839e-01
Epoch:     90, train_loss=4.712429e-01, test_loss=4.502456e-01
Epoch:    100, train_loss=4.082098e-01, test_loss=3.883186e-01
Epoch:    110, train_loss=3.516787e-01, test_loss=3.331867e-01
Epoch:    120, train_loss=3.013818e-01, test_loss=2.845173e-01
Epoch:    130, train_loss=2.571938e-01, test_loss=2.419450e-01
Epoch:    140, train_loss=2.188046e-01, test_loss=2.051224e-01
Epoch:    150, train_loss=1.858501e-01, test_loss=1.736444e-01
Epoch:    160, train_loss=1.579241e-01, test_loss=1.471259e-01
Epoch:    170, train_loss=1.345620e-01, test_loss=1.250910e-01
Epoch:    180, train_loss=1.152603e-01, test_loss=1.070274e-01
Epoch:    190, train_loss=9.949448e-02, test_loss=9.242795e-02
Epoch:    200, train_loss=8.673394e-02, test_loss=8.071829e-02
Epoch:    210, train_loss=7.645849e-02, test_loss=7.135048e-02
Epoch:    220, train_loss=6.821416e-02, test_loss=6.392075e-02
Epoch:    230, train_loss=6.161729e-02, test_loss=5.803070e-02
Epoch:    240, train_loss=5.633116e-02, test_loss=5.337977e-02
Epoch:    250, train_loss=5.208220e-02, test_loss=4.968487e-02
Epoch:    260, train_loss=4.854724e-02, test_loss=4.666844e-02
Epoch:    270, train_loss=4.561868e-02, test_loss=4.423484e-02
Epoch:    280, train_loss=4.317870e-02, test_loss=4.222362e-02
Epoch:    290, train_loss=4.111768e-02, test_loss=4.053034e-02
Epoch:    300, train_loss=3.935010e-02, test_loss=3.907191e-02
Epoch:    310, train_loss=3.781131e-02, test_loss=3.779181e-02
Epoch:    320, train_loss=3.645479e-02, test_loss=3.663711e-02
Epoch:    330, train_loss=3.523899e-02, test_loss=3.559908e-02
Epoch:    340, train_loss=3.413494e-02, test_loss=3.466276e-02
Epoch:    350, train_loss=3.312512e-02, test_loss=3.379760e-02
Epoch:    360, train_loss=3.219513e-02, test_loss=3.298957e-02
Epoch:    370, train_loss=3.133250e-02, test_loss=3.223281e-02
Epoch:    380, train_loss=3.053148e-02, test_loss=3.150913e-02
Epoch:    390, train_loss=2.978478e-02, test_loss=3.081596e-02
Epoch:    400, train_loss=2.908574e-02, test_loss=3.015228e-02
Epoch:    410, train_loss=2.842778e-02, test_loss=2.952672e-02
Epoch:    420, train_loss=2.780756e-02, test_loss=2.893357e-02
Epoch:    430, train_loss=2.721959e-02, test_loss=2.837839e-02
Epoch:    440, train_loss=2.666123e-02, test_loss=2.785773e-02
Epoch:    450, train_loss=2.613063e-02, test_loss=2.736471e-02
Epoch:    460, train_loss=2.562708e-02, test_loss=2.688911e-02
Epoch:    470, train_loss=2.514809e-02, test_loss=2.643190e-02
Epoch:    480, train_loss=2.468925e-02, test_loss=2.599955e-02
Epoch:    490, train_loss=2.424838e-02, test_loss=2.558876e-02
Epoch:    500, train_loss=2.382437e-02, test_loss=2.519041e-02
Epoch:    510, train_loss=2.341557e-02, test_loss=2.480196e-02
Epoch:    520, train_loss=2.302213e-02, test_loss=2.442184e-02
Epoch:    530, train_loss=2.264063e-02, test_loss=2.405341e-02
Epoch:    540, train_loss=2.226980e-02, test_loss=2.369553e-02
Epoch:    550, train_loss=2.190953e-02, test_loss=2.334149e-02
Epoch:    560, train_loss=2.155813e-02, test_loss=2.299505e-02
Epoch:    570, train_loss=2.121492e-02, test_loss=2.265520e-02
Epoch:    580, train_loss=2.087952e-02, test_loss=2.232229e-02
Epoch:    590, train_loss=2.055130e-02, test_loss=2.199531e-02
```

```
Epoch:    600, train_loss=2.023000e-02, test_loss=2.167374e-02
Epoch:    610, train_loss=1.991509e-02, test_loss=2.135796e-02
Epoch:    620, train_loss=1.960630e-02, test_loss=2.104825e-02
Epoch:    630, train_loss=1.930329e-02, test_loss=2.074455e-02
Epoch:    640, train_loss=1.900584e-02, test_loss=2.044680e-02
Epoch:    650, train_loss=1.871392e-02, test_loss=2.015456e-02
Epoch:    660, train_loss=1.842749e-02, test_loss=1.986760e-02
Epoch:    670, train_loss=1.814654e-02, test_loss=1.958597e-02
Epoch:    680, train_loss=1.787091e-02, test_loss=1.930933e-02
Epoch:    690, train_loss=1.760039e-02, test_loss=1.903775e-02
Epoch:    700, train_loss=1.733506e-02, test_loss=1.877125e-02
Epoch:    710, train_loss=1.707487e-02, test_loss=1.851021e-02
Epoch:    720, train_loss=1.682058e-02, test_loss=1.825422e-02
Epoch:    730, train_loss=1.657129e-02, test_loss=1.800338e-02
Epoch:    740, train_loss=1.632681e-02, test_loss=1.775768e-02
Epoch:    750, train_loss=1.608714e-02, test_loss=1.751597e-02
Epoch:    760, train_loss=1.585229e-02, test_loss=1.727851e-02
Epoch:    770, train_loss=1.562241e-02, test_loss=1.704540e-02
Epoch:    780, train_loss=1.539772e-02, test_loss=1.681633e-02
Epoch:    790, train_loss=1.517758e-02, test_loss=1.659134e-02
Epoch:    800, train_loss=1.496188e-02, test_loss=1.637017e-02
Epoch:    810, train_loss=1.475071e-02, test_loss=1.615328e-02
Epoch:    820, train_loss=1.454366e-02, test_loss=1.593995e-02
Epoch:    830, train_loss=1.434068e-02, test_loss=1.573033e-02
Epoch:    840, train_loss=1.414151e-02, test_loss=1.552412e-02
Epoch:    850, train_loss=1.394581e-02, test_loss=1.532141e-02
Epoch:    860, train_loss=1.375375e-02, test_loss=1.512239e-02
Epoch:    870, train_loss=1.356503e-02, test_loss=1.492690e-02
Epoch:    880, train_loss=1.337932e-02, test_loss=1.473461e-02
Epoch:    890, train_loss=1.319679e-02, test_loss=1.454366e-02
Epoch:    900, train_loss=1.301721e-02, test_loss=1.435522e-02
Epoch:    910, train_loss=1.284045e-02, test_loss=1.416933e-02
Epoch:    920, train_loss=1.266646e-02, test_loss=1.398622e-02
Epoch:    930, train_loss=1.249501e-02, test_loss=1.380600e-02
Epoch:    940, train_loss=1.232614e-02, test_loss=1.362841e-02
Epoch:    950, train_loss=1.215972e-02, test_loss=1.345354e-02
Epoch:    960, train_loss=1.199565e-02, test_loss=1.328097e-02
Epoch:    970, train_loss=1.183358e-02, test_loss=1.311063e-02
Epoch:    980, train_loss=1.167387e-02, test_loss=1.294285e-02
Epoch:    990, train_loss=1.151652e-02, test_loss=1.277744e-02
Epoch:   1000, train_loss=1.136148e-02, test_loss=1.261406e-02
Epoch:   1010, train_loss=1.120865e-02, test_loss=1.245321e-02
Epoch:   1020, train_loss=1.105812e-02, test_loss=1.229469e-02
Epoch:   1030, train_loss=1.090966e-02, test_loss=1.213814e-02
Epoch:   1040, train_loss=1.076312e-02, test_loss=1.198370e-02
Epoch:   1050, train_loss=1.061872e-02, test_loss=1.183163e-02
Epoch:   1060, train_loss=1.047649e-02, test_loss=1.168175e-02
Epoch:   1070, train_loss=1.033615e-02, test_loss=1.153397e-02
Epoch:   1080, train_loss=1.019714e-02, test_loss=1.138817e-02
Epoch:   1090, train_loss=1.006020e-02, test_loss=1.124498e-02
Epoch:   1100, train_loss=9.925528e-03, test_loss=1.110266e-02
Epoch:   1110, train_loss=9.792741e-03, test_loss=1.096236e-02
Epoch:   1120, train_loss=9.661847e-03, test_loss=1.082415e-02
Epoch:   1130, train_loss=9.532942e-03, test_loss=1.068827e-02
Epoch:   1140, train_loss=9.405991e-03, test_loss=1.055373e-02
Epoch:   1150, train_loss=9.280846e-03, test_loss=1.042095e-02
Epoch:   1160, train_loss=9.157511e-03, test_loss=1.029035e-02
Epoch:   1170, train_loss=9.036094e-03, test_loss=1.016192e-02
Epoch:   1180, train_loss=8.916344e-03, test_loss=1.003523e-02
Epoch:   1190, train_loss=8.798246e-03, test_loss=9.910546e-03
```

```
Epoch:   1200, train_loss=8.681867e-03, test_loss=9.787889e-03
Epoch:   1210, train_loss=8.567161e-03, test_loss=9.667186e-03
Epoch:   1220, train_loss=8.454103e-03, test_loss=9.548422e-03
Epoch:   1230, train_loss=8.342583e-03, test_loss=9.431434e-03
Epoch:   1240, train_loss=8.232746e-03, test_loss=9.316373e-03
Epoch:   1250, train_loss=8.124380e-03, test_loss=9.203057e-03
Epoch:   1260, train_loss=8.017582e-03, test_loss=9.091646e-03
Epoch:   1270, train_loss=7.912272e-03, test_loss=8.981962e-03
Epoch:   1280, train_loss=7.808513e-03, test_loss=8.874428e-03
Epoch:   1290, train_loss=7.706146e-03, test_loss=8.768788e-03
Epoch:   1300, train_loss=7.605054e-03, test_loss=8.664844e-03
Epoch:   1310, train_loss=7.505404e-03, test_loss=8.562731e-03
Epoch:   1320, train_loss=7.407157e-03, test_loss=8.462343e-03
Epoch:   1330, train_loss=7.310229e-03, test_loss=8.363623e-03
Epoch:   1340, train_loss=7.214669e-03, test_loss=8.266531e-03
Epoch:   1350, train_loss=7.120485e-03, test_loss=8.170403e-03
Epoch:   1360, train_loss=7.027851e-03, test_loss=8.074711e-03
Epoch:   1370, train_loss=6.936394e-03, test_loss=7.980456e-03
Epoch:   1380, train_loss=6.846137e-03, test_loss=7.887858e-03
Epoch:   1390, train_loss=6.756969e-03, test_loss=7.796893e-03
Epoch:   1400, train_loss=6.668999e-03, test_loss=7.707625e-03
Epoch:   1410, train_loss=6.582027e-03, test_loss=7.619912e-03
Epoch:   1420, train_loss=6.496351e-03, test_loss=7.532579e-03
Epoch:   1430, train_loss=6.411858e-03, test_loss=7.445869e-03
Epoch:   1440, train_loss=6.328413e-03, test_loss=7.360889e-03
Epoch:   1450, train_loss=6.245806e-03, test_loss=7.277423e-03
Epoch:   1460, train_loss=6.164170e-03, test_loss=7.195521e-03
Epoch:   1470, train_loss=6.083393e-03, test_loss=7.115046e-03
Epoch:   1480, train_loss=6.003361e-03, test_loss=7.036073e-03
Epoch:   1490, train_loss=5.924754e-03, test_loss=6.955853e-03
Epoch:   1500, train_loss=5.847479e-03, test_loss=6.873725e-03
Epoch:   1510, train_loss=5.771092e-03, test_loss=6.791784e-03
Epoch:   1520, train_loss=5.695606e-03, test_loss=6.710395e-03
Epoch:   1530, train_loss=5.620905e-03, test_loss=6.628755e-03
Epoch:   1540, train_loss=5.546979e-03, test_loss=6.547786e-03
Epoch:   1550, train_loss=5.473888e-03, test_loss=6.466211e-03
Epoch:   1560, train_loss=5.401280e-03, test_loss=6.386069e-03
Epoch:   1570, train_loss=5.329575e-03, test_loss=6.307579e-03
Epoch:   1580, train_loss=5.259353e-03, test_loss=6.229300e-03
Epoch:   1590, train_loss=5.190357e-03, test_loss=6.151570e-03
Epoch:   1600, train_loss=5.122247e-03, test_loss=6.075537e-03
Epoch:   1610, train_loss=5.055159e-03, test_loss=6.001296e-03
Epoch:   1620, train_loss=4.989076e-03, test_loss=5.928508e-03
Epoch:   1630, train_loss=4.924130e-03, test_loss=5.856674e-03
Epoch:   1640, train_loss=4.860502e-03, test_loss=5.784893e-03
Epoch:   1650, train_loss=4.797818e-03, test_loss=5.714616e-03
Epoch:   1660, train_loss=4.736115e-03, test_loss=5.645905e-03
Epoch:   1670, train_loss=4.675335e-03, test_loss=5.578553e-03
Epoch:   1680, train_loss=4.615508e-03, test_loss=5.512651e-03
Epoch:   1690, train_loss=4.556561e-03, test_loss=5.448101e-03
Epoch:   1700, train_loss=4.498747e-03, test_loss=5.384162e-03
Epoch:   1710, train_loss=4.442092e-03, test_loss=5.320437e-03
Epoch:   1720, train_loss=4.386721e-03, test_loss=5.256355e-03
Epoch:   1730, train_loss=4.332285e-03, test_loss=5.193247e-03
Epoch:   1740, train_loss=4.278699e-03, test_loss=5.131437e-03
Epoch:   1750, train_loss=4.226040e-03, test_loss=5.070983e-03
Epoch:   1760, train_loss=4.174262e-03, test_loss=5.011879e-03
Epoch:   1770, train_loss=4.123234e-03, test_loss=4.954141e-03
Epoch:   1780, train_loss=4.073028e-03, test_loss=4.897700e-03
Epoch:   1790, train_loss=4.023655e-03, test_loss=4.842581e-03
```

```
Epoch:   1800, train_loss=3.975133e-03, test_loss=4.788748e-03
Epoch:   1810, train_loss=3.927710e-03, test_loss=4.734747e-03
Epoch:   1820, train_loss=3.881133e-03, test_loss=4.681355e-03
Epoch:   1830, train_loss=3.835646e-03, test_loss=4.627767e-03
Epoch:   1840, train_loss=3.791004e-03, test_loss=4.575410e-03
Epoch:   1850, train_loss=3.747078e-03, test_loss=4.524093e-03
Epoch:   1860, train_loss=3.703888e-03, test_loss=4.473913e-03
Epoch:   1870, train_loss=3.661778e-03, test_loss=4.423642e-03
Epoch:   1880, train_loss=3.620546e-03, test_loss=4.374330e-03
Epoch:   1890, train_loss=3.580046e-03, test_loss=4.326123e-03
Epoch:   1900, train_loss=3.540244e-03, test_loss=4.278965e-03
Epoch:   1910, train_loss=3.501193e-03, test_loss=4.232358e-03
Epoch:   1920, train_loss=3.462995e-03, test_loss=4.185957e-03
Epoch:   1930, train_loss=3.425487e-03, test_loss=4.140623e-03
Epoch:   1940, train_loss=3.388558e-03, test_loss=4.096316e-03
Epoch:   1950, train_loss=3.352449e-03, test_loss=4.052346e-03
Epoch:   1960, train_loss=3.317071e-03, test_loss=4.008842e-03
Epoch:   1970, train_loss=3.282287e-03, test_loss=3.966392e-03
Epoch:   1980, train_loss=3.248235e-03, test_loss=3.924151e-03
Epoch:   1990, train_loss=3.214904e-03, test_loss=3.882498e-03
Epoch:   2000, train_loss=3.182127e-03, test_loss=3.841705e-03
Epoch:   2010, train_loss=3.149865e-03, test_loss=3.801813e-03
Epoch:   2020, train_loss=3.118330e-03, test_loss=3.762302e-03
Epoch:   2030, train_loss=3.087583e-03, test_loss=3.722736e-03
Epoch:   2040, train_loss=3.057460e-03, test_loss=3.683396e-03
Epoch:   2050, train_loss=3.027993e-03, test_loss=3.644417e-03
Epoch:   2060, train_loss=2.999191e-03, test_loss=3.605691e-03
Epoch:   2070, train_loss=2.970908e-03, test_loss=3.567505e-03
Epoch:   2080, train_loss=2.943289e-03, test_loss=3.529401e-03
Epoch:   2090, train_loss=2.916115e-03, test_loss=3.492114e-03
Epoch:   2100, train_loss=2.889356e-03, test_loss=3.455617e-03
Epoch:   2110, train_loss=2.863045e-03, test_loss=3.419898e-03
Epoch:   2120, train_loss=2.837322e-03, test_loss=3.384464e-03
Epoch:   2130, train_loss=2.812116e-03, test_loss=3.349446e-03
Epoch:   2140, train_loss=2.787376e-03, test_loss=3.315085e-03
Epoch:   2150, train_loss=2.763185e-03, test_loss=3.280766e-03
Epoch:   2160, train_loss=2.739398e-03, test_loss=3.247128e-03
Epoch:   2170, train_loss=2.716003e-03, test_loss=3.214197e-03
Epoch:   2180, train_loss=2.692944e-03, test_loss=3.181915e-03
Epoch:   2190, train_loss=2.670306e-03, test_loss=3.150319e-03
Epoch:   2200, train_loss=2.648014e-03, test_loss=3.119343e-03
Epoch:   2210, train_loss=2.626054e-03, test_loss=3.088962e-03
Epoch:   2220, train_loss=2.604475e-03, test_loss=3.059253e-03
Epoch:   2230, train_loss=2.583297e-03, test_loss=3.030166e-03
Epoch:   2240, train_loss=2.562428e-03, test_loss=3.001589e-03
Epoch:   2250, train_loss=2.542007e-03, test_loss=2.973205e-03
Epoch:   2260, train_loss=2.521992e-03, test_loss=2.945095e-03
Epoch:   2270, train_loss=2.502289e-03, test_loss=2.917602e-03
Epoch:   2280, train_loss=2.482864e-03, test_loss=2.890544e-03
Epoch:   2290, train_loss=2.463767e-03, test_loss=2.864052e-03
Epoch:   2300, train_loss=2.444992e-03, test_loss=2.838139e-03
Epoch:   2310, train_loss=2.426494e-03, test_loss=2.812648e-03
Epoch:   2320, train_loss=2.408250e-03, test_loss=2.787657e-03
Epoch:   2330, train_loss=2.390312e-03, test_loss=2.763214e-03
Epoch:   2340, train_loss=2.372649e-03, test_loss=2.739183e-03
Epoch:   2350, train_loss=2.355268e-03, test_loss=2.715660e-03
Epoch:   2360, train_loss=2.338107e-03, test_loss=2.692601e-03
Epoch:   2370, train_loss=2.321217e-03, test_loss=2.669938e-03
Epoch:   2380, train_loss=2.304644e-03, test_loss=2.647789e-03
Epoch:   2390, train_loss=2.288253e-03, test_loss=2.625934e-03
```

```
Epoch:   2400, train_loss=2.272130e-03, test_loss=2.604506e-03
Epoch:   2410, train_loss=2.256255e-03, test_loss=2.583480e-03
Epoch:   2420, train_loss=2.240603e-03, test_loss=2.562954e-03
Epoch:   2430, train_loss=2.225171e-03, test_loss=2.542848e-03
Epoch:   2440, train_loss=2.209922e-03, test_loss=2.523056e-03
Epoch:   2450, train_loss=2.194963e-03, test_loss=2.503761e-03
Epoch:   2460, train_loss=2.180176e-03, test_loss=2.484743e-03
Epoch:   2470, train_loss=2.165584e-03, test_loss=2.466114e-03
Epoch:   2480, train_loss=2.151195e-03, test_loss=2.447824e-03
Epoch:   2490, train_loss=2.137062e-03, test_loss=2.429896e-03
Epoch:   2500, train_loss=2.123104e-03, test_loss=2.412234e-03
Epoch:   2510, train_loss=2.109328e-03, test_loss=2.394847e-03
Epoch:   2520, train_loss=2.095765e-03, test_loss=2.377797e-03
Epoch:   2530, train_loss=2.082403e-03, test_loss=2.361017e-03
Epoch:   2540, train_loss=2.069218e-03, test_loss=2.344598e-03
Epoch:   2550, train_loss=2.056197e-03, test_loss=2.328428e-03
Epoch:   2560, train_loss=2.043364e-03, test_loss=2.312522e-03
Epoch:   2570, train_loss=2.030712e-03, test_loss=2.296861e-03
Epoch:   2580, train_loss=2.018224e-03, test_loss=2.281451e-03
Epoch:   2590, train_loss=2.005891e-03, test_loss=2.266291e-03
Epoch:   2600, train_loss=1.993725e-03, test_loss=2.251370e-03
Epoch:   2610, train_loss=1.981758e-03, test_loss=2.236740e-03
Epoch:   2620, train_loss=1.969931e-03, test_loss=2.222141e-03
Epoch:   2630, train_loss=1.958293e-03, test_loss=2.207806e-03
Epoch:   2640, train_loss=1.946768e-03, test_loss=2.193640e-03
Epoch:   2650, train_loss=1.935452e-03, test_loss=2.179798e-03
Epoch:   2660, train_loss=1.924197e-03, test_loss=2.166124e-03
Epoch:   2670, train_loss=1.913113e-03, test_loss=2.152727e-03
Epoch:   2680, train_loss=1.902183e-03, test_loss=2.139530e-03
Epoch:   2690, train_loss=1.891384e-03, test_loss=2.126518e-03
Epoch:   2700, train_loss=1.880731e-03, test_loss=2.113720e-03
Epoch:   2710, train_loss=1.870177e-03, test_loss=2.101064e-03
Epoch:   2720, train_loss=1.859788e-03, test_loss=2.088652e-03
Epoch:   2730, train_loss=1.849541e-03, test_loss=2.076409e-03
Epoch:   2740, train_loss=1.839381e-03, test_loss=2.064206e-03
Epoch:   2750, train_loss=1.829391e-03, test_loss=2.052214e-03
Epoch:   2760, train_loss=1.819497e-03, test_loss=2.040380e-03
Epoch:   2770, train_loss=1.809736e-03, test_loss=2.028780e-03
Epoch:   2780, train_loss=1.800101e-03, test_loss=2.017362e-03
Epoch:   2790, train_loss=1.790603e-03, test_loss=2.005799e-03
Epoch:   2800, train_loss=1.781245e-03, test_loss=1.994404e-03
Epoch:   2810, train_loss=1.772026e-03, test_loss=1.983188e-03
Epoch:   2820, train_loss=1.762855e-03, test_loss=1.972012e-03
Epoch:   2830, train_loss=1.753807e-03, test_loss=1.961054e-03
Epoch:   2840, train_loss=1.744888e-03, test_loss=1.950249e-03
Epoch:   2850, train_loss=1.736030e-03, test_loss=1.939549e-03
Epoch:   2860, train_loss=1.727295e-03, test_loss=1.929033e-03
Epoch:   2870, train_loss=1.718654e-03, test_loss=1.918681e-03
Epoch:   2880, train_loss=1.710082e-03, test_loss=1.908380e-03
Epoch:   2890, train_loss=1.701631e-03, test_loss=1.898241e-03
Epoch:   2900, train_loss=1.693294e-03, test_loss=1.888237e-03
Epoch:   2910, train_loss=1.684987e-03, test_loss=1.878256e-03
Epoch:   2920, train_loss=1.676780e-03, test_loss=1.868412e-03
Epoch:   2930, train_loss=1.668697e-03, test_loss=1.858725e-03
Epoch:   2940, train_loss=1.660709e-03, test_loss=1.849191e-03
Epoch:   2950, train_loss=1.652738e-03, test_loss=1.839697e-03
Epoch:   2960, train_loss=1.644876e-03, test_loss=1.830353e-03
Epoch:   2970, train_loss=1.637129e-03, test_loss=1.821169e-03
Epoch:   2980, train_loss=1.629441e-03, test_loss=1.812043e-03
Epoch:   2990, train_loss=1.621828e-03, test_loss=1.803005e-03
```

```
Epoch:   3000, train_loss=1.614299e-03, test_loss=1.794099e-03
Epoch:   3010, train_loss=1.606870e-03, test_loss=1.785333e-03
Epoch:   3020, train_loss=1.599496e-03, test_loss=1.776598e-03
Epoch:   3030, train_loss=1.592191e-03, test_loss=1.767962e-03
Epoch:   3040, train_loss=1.584962e-03, test_loss=1.759460e-03
Epoch:   3050, train_loss=1.577808e-03, test_loss=1.751073e-03
Epoch:   3060, train_loss=1.570705e-03, test_loss=1.742728e-03
Epoch:   3070, train_loss=1.563683e-03, test_loss=1.734520e-03
Epoch:   3080, train_loss=1.556704e-03, test_loss=1.726454e-03
Epoch:   3090, train_loss=1.549817e-03, test_loss=1.718551e-03
Epoch:   3100, train_loss=1.542982e-03, test_loss=1.710730e-03
Epoch:   3110, train_loss=1.536214e-03, test_loss=1.703016e-03
Epoch:   3120, train_loss=1.529490e-03, test_loss=1.695376e-03
Epoch:   3130, train_loss=1.522834e-03, test_loss=1.687849e-03
Epoch:   3140, train_loss=1.516275e-03, test_loss=1.680375e-03
Epoch:   3150, train_loss=1.509790e-03, test_loss=1.672831e-03
Epoch:   3160, train_loss=1.503375e-03, test_loss=1.665378e-03
Epoch:   3170, train_loss=1.497002e-03, test_loss=1.657991e-03
Epoch:   3180, train_loss=1.490704e-03, test_loss=1.650703e-03
Epoch:   3190, train_loss=1.484458e-03, test_loss=1.643451e-03
Epoch:   3200, train_loss=1.478276e-03, test_loss=1.636286e-03
Epoch:   3210, train_loss=1.472119e-03, test_loss=1.629187e-03
Epoch:   3220, train_loss=1.466021e-03, test_loss=1.622186e-03
Epoch:   3230, train_loss=1.460000e-03, test_loss=1.615273e-03
Epoch:   3240, train_loss=1.454017e-03, test_loss=1.608360e-03
Epoch:   3250, train_loss=1.448076e-03, test_loss=1.601501e-03
Epoch:   3260, train_loss=1.442190e-03, test_loss=1.594737e-03
Epoch:   3270, train_loss=1.436364e-03, test_loss=1.588039e-03
Epoch:   3280, train_loss=1.430606e-03, test_loss=1.581397e-03
Epoch:   3290, train_loss=1.424913e-03, test_loss=1.574824e-03
Epoch:   3300, train_loss=1.419233e-03, test_loss=1.568247e-03
Epoch:   3310, train_loss=1.413597e-03, test_loss=1.561754e-03
Epoch:   3320, train_loss=1.408025e-03, test_loss=1.555356e-03
Epoch:   3330, train_loss=1.402522e-03, test_loss=1.549021e-03
Epoch:   3340, train_loss=1.397076e-03, test_loss=1.542764e-03
Epoch:   3350, train_loss=1.391625e-03, test_loss=1.536497e-03
Epoch:   3360, train_loss=1.386212e-03, test_loss=1.530273e-03
Epoch:   3370, train_loss=1.380867e-03, test_loss=1.524130e-03
Epoch:   3380, train_loss=1.375571e-03, test_loss=1.518049e-03
Epoch:   3390, train_loss=1.370321e-03, test_loss=1.512051e-03
Epoch:   3400, train_loss=1.365101e-03, test_loss=1.506099e-03
Epoch:   3410, train_loss=1.359928e-03, test_loss=1.500174e-03
Epoch:   3420, train_loss=1.354785e-03, test_loss=1.494300e-03
Epoch:   3430, train_loss=1.349701e-03, test_loss=1.488502e-03
Epoch:   3440, train_loss=1.344652e-03, test_loss=1.482750e-03
Epoch:   3450, train_loss=1.339639e-03, test_loss=1.477056e-03
Epoch:   3460, train_loss=1.334683e-03, test_loss=1.471438e-03
Epoch:   3470, train_loss=1.329779e-03, test_loss=1.465874e-03
Epoch:   3480, train_loss=1.324899e-03, test_loss=1.460315e-03
Epoch:   3490, train_loss=1.320028e-03, test_loss=1.454777e-03
Epoch:   3500, train_loss=1.315210e-03, test_loss=1.449302e-03
Epoch:   3510, train_loss=1.310434e-03, test_loss=1.443888e-03
Epoch:   3520, train_loss=1.305699e-03, test_loss=1.438534e-03
Epoch:   3530, train_loss=1.301015e-03, test_loss=1.433241e-03
Epoch:   3540, train_loss=1.296340e-03, test_loss=1.428013e-03
Epoch:   3550, train_loss=1.291686e-03, test_loss=1.422787e-03
Epoch:   3560, train_loss=1.287074e-03, test_loss=1.417605e-03
Epoch:   3570, train_loss=1.282504e-03, test_loss=1.412508e-03
Epoch:   3580, train_loss=1.277981e-03, test_loss=1.407466e-03
Epoch:   3590, train_loss=1.273463e-03, test_loss=1.402457e-03
```

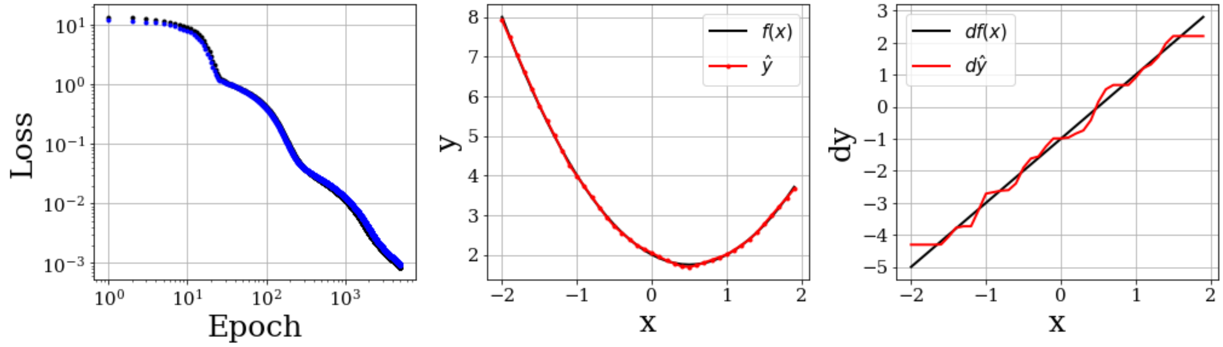
```
Epoch:   3600, train_loss=1.268999e-03, test_loss=1.397512e-03
Epoch:   3610, train_loss=1.264574e-03, test_loss=1.392619e-03
Epoch:   3620, train_loss=1.260182e-03, test_loss=1.387744e-03
Epoch:   3630, train_loss=1.255812e-03, test_loss=1.382891e-03
Epoch:   3640, train_loss=1.251454e-03, test_loss=1.378100e-03
Epoch:   3650, train_loss=1.247137e-03, test_loss=1.373323e-03
Epoch:   3660, train_loss=1.242862e-03, test_loss=1.368592e-03
Epoch:   3670, train_loss=1.238627e-03, test_loss=1.363905e-03
Epoch:   3680, train_loss=1.234441e-03, test_loss=1.359268e-03
Epoch:   3690, train_loss=1.230253e-03, test_loss=1.354665e-03
Epoch:   3700, train_loss=1.226091e-03, test_loss=1.350083e-03
Epoch:   3710, train_loss=1.221961e-03, test_loss=1.345492e-03
Epoch:   3720, train_loss=1.217872e-03, test_loss=1.340964e-03
Epoch:   3730, train_loss=1.213822e-03, test_loss=1.336475e-03
Epoch:   3740, train_loss=1.209781e-03, test_loss=1.332031e-03
Epoch:   3750, train_loss=1.205764e-03, test_loss=1.327649e-03
Epoch:   3760, train_loss=1.201788e-03, test_loss=1.323324e-03
Epoch:   3770, train_loss=1.197851e-03, test_loss=1.319035e-03
Epoch:   3780, train_loss=1.193940e-03, test_loss=1.314780e-03
Epoch:   3790, train_loss=1.190015e-03, test_loss=1.310522e-03
Epoch:   3800, train_loss=1.186116e-03, test_loss=1.306304e-03
Epoch:   3810, train_loss=1.182253e-03, test_loss=1.302126e-03
Epoch:   3820, train_loss=1.178427e-03, test_loss=1.298013e-03
Epoch:   3830, train_loss=1.174641e-03, test_loss=1.293937e-03
Epoch:   3840, train_loss=1.170875e-03, test_loss=1.289890e-03
Epoch:   3850, train_loss=1.167121e-03, test_loss=1.285870e-03
Epoch:   3860, train_loss=1.163398e-03, test_loss=1.281881e-03
Epoch:   3870, train_loss=1.159713e-03, test_loss=1.277942e-03
Epoch:   3880, train_loss=1.156047e-03, test_loss=1.273994e-03
Epoch:   3890, train_loss=1.152401e-03, test_loss=1.270079e-03
Epoch:   3900, train_loss=1.148755e-03, test_loss=1.266206e-03
Epoch:   3910, train_loss=1.145140e-03, test_loss=1.262371e-03
Epoch:   3920, train_loss=1.141555e-03, test_loss=1.258592e-03
Epoch:   3930, train_loss=1.137999e-03, test_loss=1.254834e-03
Epoch:   3940, train_loss=1.134464e-03, test_loss=1.251133e-03
Epoch:   3950, train_loss=1.130949e-03, test_loss=1.247450e-03
Epoch:   3960, train_loss=1.127454e-03, test_loss=1.243782e-03
Epoch:   3970, train_loss=1.123977e-03, test_loss=1.240141e-03
Epoch:   3980, train_loss=1.120519e-03, test_loss=1.236465e-03
Epoch:   3990, train_loss=1.117098e-03, test_loss=1.232812e-03
Epoch:   4000, train_loss=1.113714e-03, test_loss=1.229196e-03
Epoch:   4010, train_loss=1.110330e-03, test_loss=1.225601e-03
Epoch:   4020, train_loss=1.106975e-03, test_loss=1.222051e-03
Epoch:   4030, train_loss=1.103653e-03, test_loss=1.218538e-03
Epoch:   4040, train_loss=1.100352e-03, test_loss=1.215055e-03
Epoch:   4050, train_loss=1.097081e-03, test_loss=1.211591e-03
Epoch:   4060, train_loss=1.093834e-03, test_loss=1.208152e-03
Epoch:   4070, train_loss=1.090578e-03, test_loss=1.204714e-03
Epoch:   4080, train_loss=1.087336e-03, test_loss=1.201273e-03
Epoch:   4090, train_loss=1.084120e-03, test_loss=1.197873e-03
Epoch:   4100, train_loss=1.080930e-03, test_loss=1.194502e-03
Epoch:   4110, train_loss=1.077764e-03, test_loss=1.191144e-03
Epoch:   4120, train_loss=1.074626e-03, test_loss=1.187833e-03
Epoch:   4130, train_loss=1.071493e-03, test_loss=1.184549e-03
Epoch:   4140, train_loss=1.068382e-03, test_loss=1.181295e-03
Epoch:   4150, train_loss=1.065299e-03, test_loss=1.178053e-03
Epoch:   4160, train_loss=1.062242e-03, test_loss=1.174848e-03
Epoch:   4170, train_loss=1.059214e-03, test_loss=1.171667e-03
Epoch:   4180, train_loss=1.056200e-03, test_loss=1.168410e-03
Epoch:   4190, train_loss=1.053197e-03, test_loss=1.165072e-03
```

```
Epoch:   4200, train_loss=1.050210e-03, test_loss=1.161776e-03
Epoch:   4210, train_loss=1.047247e-03, test_loss=1.158496e-03
Epoch:   4220, train_loss=1.044310e-03, test_loss=1.155241e-03
Epoch:   4230, train_loss=1.041399e-03, test_loss=1.152018e-03
Epoch:   4240, train_loss=1.038505e-03, test_loss=1.148836e-03
Epoch:   4250, train_loss=1.035608e-03, test_loss=1.145669e-03
Epoch:   4260, train_loss=1.032734e-03, test_loss=1.142518e-03
Epoch:   4270, train_loss=1.029882e-03, test_loss=1.139390e-03
Epoch:   4280, train_loss=1.027062e-03, test_loss=1.136289e-03
Epoch:   4290, train_loss=1.024263e-03, test_loss=1.133210e-03
Epoch:   4300, train_loss=1.021490e-03, test_loss=1.130087e-03
Epoch:   4310, train_loss=1.018693e-03, test_loss=1.126969e-03
Epoch:   4320, train_loss=1.015919e-03, test_loss=1.123855e-03
Epoch:   4330, train_loss=1.013171e-03, test_loss=1.120783e-03
Epoch:   4340, train_loss=1.010444e-03, test_loss=1.117733e-03
Epoch:   4350, train_loss=1.007743e-03, test_loss=1.114706e-03
Epoch:   4360, train_loss=1.005057e-03, test_loss=1.111726e-03
Epoch:   4370, train_loss=1.002393e-03, test_loss=1.108649e-03
Epoch:   4380, train_loss=9.997521e-04, test_loss=1.105536e-03
Epoch:   4390, train_loss=9.971303e-04, test_loss=1.102446e-03
Epoch:   4400, train_loss=9.945279e-04, test_loss=1.099386e-03
Epoch:   4410, train_loss=9.919484e-04, test_loss=1.096383e-03
Epoch:   4420, train_loss=9.893763e-04, test_loss=1.093386e-03
Epoch:   4430, train_loss=9.868196e-04, test_loss=1.090382e-03
Epoch:   4440, train_loss=9.842539e-04, test_loss=1.087315e-03
Epoch:   4450, train_loss=9.817046e-04, test_loss=1.084285e-03
Epoch:   4460, train_loss=9.791764e-04, test_loss=1.081288e-03
Epoch:   4470, train_loss=9.766789e-04, test_loss=1.078332e-03
Epoch:   4480, train_loss=9.741896e-04, test_loss=1.075410e-03
Epoch:   4490, train_loss=9.717217e-04, test_loss=1.072485e-03
Epoch:   4500, train_loss=9.692705e-04, test_loss=1.069577e-03
Epoch:   4510, train_loss=9.668068e-04, test_loss=1.066680e-03
Epoch:   4520, train_loss=9.643740e-04, test_loss=1.063832e-03
Epoch:   4530, train_loss=9.619614e-04, test_loss=1.061010e-03
Epoch:   4540, train_loss=9.595582e-04, test_loss=1.058188e-03
Epoch:   4550, train_loss=9.571754e-04, test_loss=1.055385e-03
Epoch:   4560, train_loss=9.548044e-04, test_loss=1.052587e-03
Epoch:   4570, train_loss=9.524540e-04, test_loss=1.049806e-03
Epoch:   4580, train_loss=9.500787e-04, test_loss=1.046982e-03
Epoch:   4590, train_loss=9.477182e-04, test_loss=1.044196e-03
Epoch:   4600, train_loss=9.453704e-04, test_loss=1.041417e-03
Epoch:   4610, train_loss=9.430441e-04, test_loss=1.038662e-03
Epoch:   4620, train_loss=9.407334e-04, test_loss=1.035933e-03
Epoch:   4630, train_loss=9.384473e-04, test_loss=1.033224e-03
Epoch:   4640, train_loss=9.361753e-04, test_loss=1.030540e-03
Epoch:   4650, train_loss=9.338923e-04, test_loss=1.027883e-03
Epoch:   4660, train_loss=9.316203e-04, test_loss=1.025221e-03
Epoch:   4670, train_loss=9.293647e-04, test_loss=1.022583e-03
Epoch:   4680, train_loss=9.271254e-04, test_loss=1.019957e-03
Epoch:   4690, train_loss=9.249062e-04, test_loss=1.017351e-03
Epoch:   4700, train_loss=9.227000e-04, test_loss=1.014763e-03
Epoch:   4710, train_loss=9.205118e-04, test_loss=1.012199e-03
Epoch:   4720, train_loss=9.182978e-04, test_loss=1.009625e-03
Epoch:   4730, train_loss=9.160879e-04, test_loss=1.007034e-03
Epoch:   4740, train_loss=9.139023e-04, test_loss=1.004476e-03
Epoch:   4750, train_loss=9.117357e-04, test_loss=1.001947e-03
Epoch:   4760, train_loss=9.095859e-04, test_loss=9.994252e-04
Epoch:   4770, train_loss=9.074489e-04, test_loss=9.968808e-04
Epoch:   4780, train_loss=9.053340e-04, test_loss=9.943473e-04
Epoch:   4790, train_loss=9.032315e-04, test_loss=9.918311e-04
```

```
Epoch:   4800, train_loss=9.011173e-04, test_loss=9.893219e-04
Epoch:   4810, train_loss=8.990159e-04, test_loss=9.868311e-04
Epoch:   4820, train_loss=8.969358e-04, test_loss=9.843587e-04
Epoch:   4830, train_loss=8.948625e-04, test_loss=9.819217e-04
Epoch:   4840, train_loss=8.928005e-04, test_loss=9.794740e-04
Epoch:   4850, train_loss=8.907596e-04, test_loss=9.770488e-04
Epoch:   4860, train_loss=8.887326e-04, test_loss=9.746322e-04
Epoch:   4870, train_loss=8.867017e-04, test_loss=9.722043e-04
Epoch:   4880, train_loss=8.846738e-04, test_loss=9.697562e-04
Epoch:   4890, train_loss=8.826634e-04, test_loss=9.673198e-04
Epoch:   4900, train_loss=8.806609e-04, test_loss=9.649015e-04
Epoch:   4910, train_loss=8.786701e-04, test_loss=9.624938e-04
Epoch:   4920, train_loss=8.766902e-04, test_loss=9.600890e-04
Epoch:   4930, train_loss=8.747188e-04, test_loss=9.576948e-04
Epoch:   4940, train_loss=8.727662e-04, test_loss=9.553223e-04
Epoch:   4950, train_loss=8.708000e-04, test_loss=9.529570e-04
Epoch:   4960, train_loss=8.688479e-04, test_loss=9.506042e-04
Epoch:   4970, train_loss=8.669023e-04, test_loss=9.482831e-04
Epoch:   4980, train_loss=8.649692e-04, test_loss=9.459605e-04
Epoch:   4990, train_loss=8.630547e-04, test_loss=9.436455e-04
```

In [138…

```python
fig, ax = plt.subplots(1,3, constrained_layout=True)

ax[0].loglog(np.arange(1,nbEpochs+1), loss, 'k.', linewidth=2)
ax[0].loglog(np.arange(1,nbEpochs+1), lossTest, 'b.', linewidth=2)
ax[0].set_xlabel("Epoch", fontsize=25)
ax[0].set_ylabel("Loss", fontsize=25)

x  = torch.arange(-2, 2, .1).unsqueeze(1)
y  = mlp(x).detach()

ax[1].plot(x, fun(x), 'k-', linewidth=2, label='$f(x)$')
ax[1].plot(x, y, 'r.-', linewidth=2, label='$\hat{y}$')
ax[1].legend(loc="upper right", fontsize=15)
ax[1].set_xlabel("x", fontsize=25)
ax[1].set_ylabel("y", fontsize=25)

df = 2*x-1
dy = np.gradient(y.squeeze().detach().numpy(), x.squeeze().detach().numpy())
ax[2].plot(x, df, 'k-', linewidth=2, label='$df(x)$')
ax[2].plot(x, dy, 'r-', linewidth=2, label='$d\hat{y}$')
ax[2].legend(loc="upper left", fontsize=15)
ax[2].set_xlabel("x", fontsize=25)
ax[2].set_ylabel("dy", fontsize=25)

for i in range(3):
    ax[i].grid(True)

    for tick in ax[i].get_xticklabels():
        tick.set_fontsize(15)

    for tick in ax[i].get_yticklabels():
        tick.set_fontsize(15)

fig.set_size_inches(14,4)
```

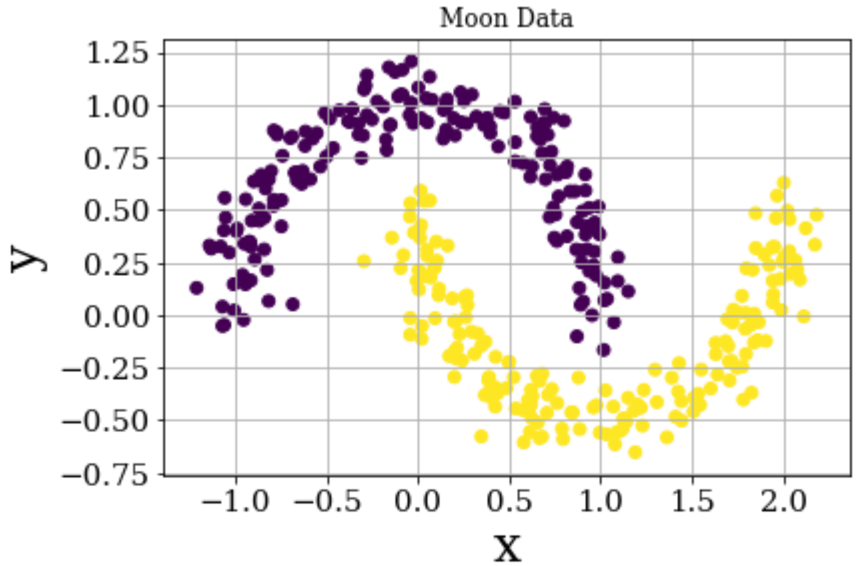


The loss is very nice, it does not present crazy oscillations as previously. We observe also the nice fit of the network with respect to the analytical expression of the quadratic function $f(x)$. If you observe correctly the red curve, you will see that it appears as three *straight lines*. It is due to the activation function used in this training. The ReLU is a piecewise linear function and the prediction also appears as a *piecewise linear function* (roughly speaking). To enhance the smoothness of the prediction, you can add more hidden layers into the model. It will increase the complexity of the model by increasing the number of parameters.

5. Train your first MLP!

Now it's your turn. During this tutorial, we have learn how to construct a neural network using `nn.Module` and `nn.Sequential` , how to train it and easily update the network's parameters using `torch.optim` and how to create the train and test set with `DataLoader` . You have the necessary knowledge to train your first neural network to perform a classification problem. We first import a toy example dataset from the sklearn library, and then split it for the training.

In [5]:
```
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
import numpy as np

X, Y = make_moons(500, noise=0.1) # create artificial data

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.25, ran

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

ax.scatter(X_train[:,0], X_train[:,1], c=Y_train)

ax.set_xlabel("x", fontsize=25)
ax.set_ylabel("y", fontsize=25)

ax.grid(True)

for tick in ax.get_xticklabels():
    tick.set_fontsize(15)

for tick in ax.get_yticklabels():
    tick.set_fontsize(15)

plt.title('Moon Data')
fig.set_size_inches(6,4)

plt.show()
```



The neural network has to distinguish between the purpule points and the yellow ones. The neutwork will try to somehow predict a curve boundary between the two set of data. Because it is a classification porblem, you will have to use `torch.nn.CrossEntropyLoss` (see the documentation here). For the optimizer, you can choose between between SGS and Adam, for example. Concerning the network, you can test different activation function `nn.functional.relu` , `nn.functional.elu` , `torch.sigmoid` , and `nn.functional.tanh` . You can play with the size of your neural network (both the number of layers and the number of neurons in each layer).

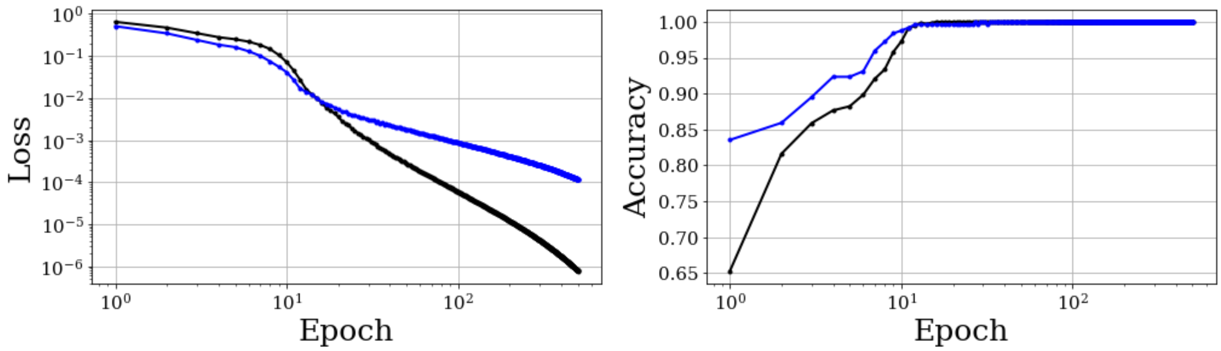
During the training stage, it is important to keep track whether your model will improve over the different iterations. It is therefore good practice to monitor whether the loss (training and testing one) you are minimizing decreases over time, and whether the overall performance of the model (training and testing accuracy) increases across epochs. (see for example figures below).

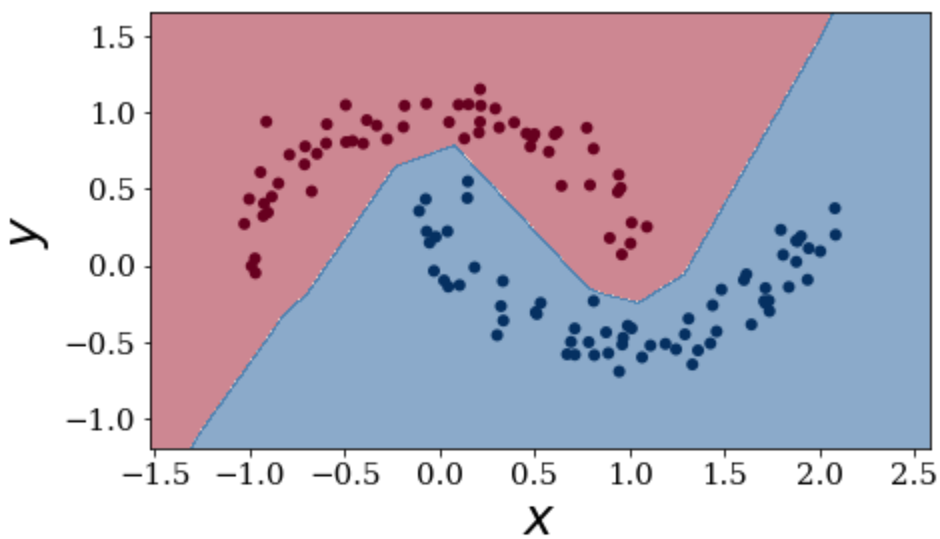
In [ ]:
```
# your code here
```

I propose you one possible solution in `classification_solution.py` but I first encourage you to code your own one before looking at the solution. **Good luck ...**

In [1]:
```
from classification_solution import *
```

In [7]:
```
# parameters
nbEpochs     = 500+1
hidden_layer = [10,20,10]

# training
net, training_loss, testing_loss, training_acc, testing_acc = training(hidden
                                                                        X_trai
                                                                        X_test
                                                                        nbEpoc

# loss and results
plot_loss_accuracy(nbEpochs, X_test, Y_test, net,
                   training_loss, testing_loss,
                   training_acc, testing_acc)
```





Bibliography

[1] Hooman H. Rashidi, Nam K. Tran, Elham Vali Betts, Lydia P. Howell, and Ralph Green. Artificial Intelligence and Machine Learning in Pathology: The Present Landscape of Supervised Methods. Academic Pathology, 6:237428951987308, January 2019.