Regression with Deep Neural Networks

Contents

```python
import numpy as np
np.set_printoptions(precision=3)
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
sns.set(rc={"figure.dpi":100, "savefig.dpi":300})
sns.set_context("notebook")
sns.set_style("ticks")

import scipy
import scipy.stats as st
import urllib.request
import os

def download(
    url : str,
    local_filename : str = None
):
    """Download a file from a url.

    Arguments
    url            -- The url we want to download.
    local_filename -- The filemame to write on. If not
                      specified
    """
    if local_filename is None:
        local_filename = os.path.basename(url)
    urllib.request.urlretrieve(url, local_filename)
```

Objectives

- Understand the basics of `PyTorch`.
- Set up and train regression DNNs with `PyTorch`

References

- Reading Activity 24
- Chapters 6, 7, and 8 of https://www.deeplearningbook.org/
- Deep Learning with PyTorch: A 60 minute blitz and in particular:
  - What is PyTorch?
  - Autograd: Automatic differentation
  - Neural networks

What is PyTorch and why are we using it?

- PyTorch is an alternative to Numpy that can harness the power of GPUs.
- PyTorch provides some core functionality for Neural Networks:

- PyTorch provides some core functionality for Neural Networks:
    - Some basic elements for building them up like linear layers, activation functions, etc.
    - Automatic differentation for getting the derivative of loss functions with respect to parameters.
    - Some stochastic optimization algorithms for minimizing loss functions
    - ...

I am not going to provide here a complete tutorial of PyTorch. You are advised to go over the first three topics of the Deep Learning with PyTorch: A 60 minute blitz prior to beginning this hands-on activity. Otherwise, it is unlikely that you understand the code that follows.

PyTorch was developed by the Facebook AI Research Group. There is another powerful alternative developed by Google Brain: TensorFlow. I find PyTorch easier to use than TensorFlow and that's why we only use this in this class.

Making neural networks in PyTorch

PyTorch is fairly flexible in allowing you to make any type neural network you like. You have absolute freedom on how your model looks like. However, it does provide a super easy way to make dense neural networks with a fixed activation function. That's what we are going to start with. First, import torch:

```
import torch
```

The submodule `torch.nn` is where the neural network building blocks reside:

```
import torch.nn as nn
```

First, let me show you how you can make a single linear layer: $$y = Wx + b.$$ The weights are selected randomly if not specified. Here you go:

```
layer = nn.Linear(1, 20)
```

This is now a function that takes one dimensional inputs and spits out 20 dimensional outputs. Here is how it works:

```
x = torch.rand(10, 1) # 10 randomly sampled one dimensinal inputs
print(x)
```

```
tensor([[0.3243],
        [0.7808],
        [0.2846],
        [0.2802],
        [0.5175],
        [0.1368],
        [0.8318],
        [0.8738],
        [0.1703],
        [0.8635]])
```

```
y = layer(x)
print(y)
```

```
tensor([[ 0.9553, -0.0708,  0.0977,  0.1812, -0.8013, -0.7836, -0.5528,  0.6642,
          0.1388,  1.2068, -0.1274, -0.1646, -0.8330, -0.7469, -0.4537, -0.9398,
          0.1926,  0.9901,  0.3872,  1.0143],
        [ 1.1075, -0.2200,  0.4106,  0.6055, -1.1514, -0.6292, -0.8909,  0.6650,
          0.3702,  1.5170, -0.4881, -0.2010, -1.2772, -0.8869, -0.4392, -1.1719,
          0.5615,  1.0600,  0.6605,  1.3452],
        [ 0.9420, -0.0578,  0.0705,  0.1442, -0.7708, -0.7970, -0.5233,  0.6641,
          0.1187,  1.1798, -0.0960, -0.1615, -0.7943, -0.7347, -0.4550, -0.9196,
          0.1605,  0.9840,  0.3634,  0.9855],
        [ 0.9405, -0.0564,  0.0674,  0.1401, -0.7674, -0.7985, -0.5201,  0.6641,
          0.1164,  1.1768, -0.0925, -0.1611, -0.7900, -0.7333, -0.4551, -0.9174,
          0.1569,  0.9833,  0.3608,  0.9823],
        [ 1.0197, -0.1340,  0.2301,  0.3607, -0.9494, -0.7183, -0.6959,  0.6645,
          0.2367,  1.3381, -0.2801, -0.1800, -1.0209, -0.8061, -0.4476, -1.0380,
          0.3487,  1.0197,  0.5029,  1.1544],
        [ 0.8927, -0.0095, -0.0308,  0.0068, -0.6575, -0.8470, -0.4139,  0.6638,
          0.0438,  1.0794,  0.0208, -0.1497, -0.6505, -0.6893, -0.4597, -0.8445,
          0.0411,  0.9614,  0.2750,  0.8784],
        [ 1.1245, -0.2367,  0.4455,  0.6529, -1.1905, -0.6120, -0.9286,  0.6650,
          0.3961,  1.5516, -0.5284, -0.2050, -1.3268, -0.9025, -0.4376, -1.1979,
          0.6027,  1.0678,  0.6911,  1.3822],
        [ 1.1385, -0.2504,  0.4743,  0.6920, -1.2227, -0.5978, -0.9597,  0.6651,
          0.4173,  1.5802, -0.5616, -0.2084, -1.3676, -0.9154, -0.4362, -1.2192,
          0.6366,  1.0743,  0.7162,  1.4126],
        [ 0.9039, -0.0205, -0.0079,  0.0380, -0.6832, -0.8356, -0.4387,  0.6639,
          0.0607,  1.1022, -0.0057, -0.1524, -0.6831, -0.6996, -0.4586, -0.8615,
          0.0682,  0.9665,  0.2950,  0.9027],
        [ 1.1350, -0.2470,  0.4672,  0.6823, -1.2148, -0.6013, -0.9521,  0.6651,
          0.4121,  1.5731, -0.5534, -0.2075, -1.3575, -0.9122, -0.4366, -1.2140,
          0.6282,  1.0727,  0.7100,  1.4051]], grad_fn=<AddmmBackward0>)
```

```python
print(y.shape)
```

```
torch.Size([10, 20])
```

So, this took us to 10, 20 dimensional outputs. Looks good.

But where are the weights and the bias term? Here they are:

```python
layer.weight
```

```
Parameter containing:
tensor([[ 0.3335],
        [-0.3268],
        [ 0.6854],
        [ 0.9296],
        [-0.7669],
        [ 0.3381],
        [-0.7406],
        [ 0.0017],
        [ 0.5069],
        [ 0.6794],
        [-0.7903],
        [-0.0796],
        [-0.9730],
        [-0.3067],
        [ 0.0318],
        [-0.5085],
        [ 0.8080],
        [ 0.1532],
        [ 0.5987],
        [ 0.7248]], requires_grad=True)
```

```python
layer.bias
```

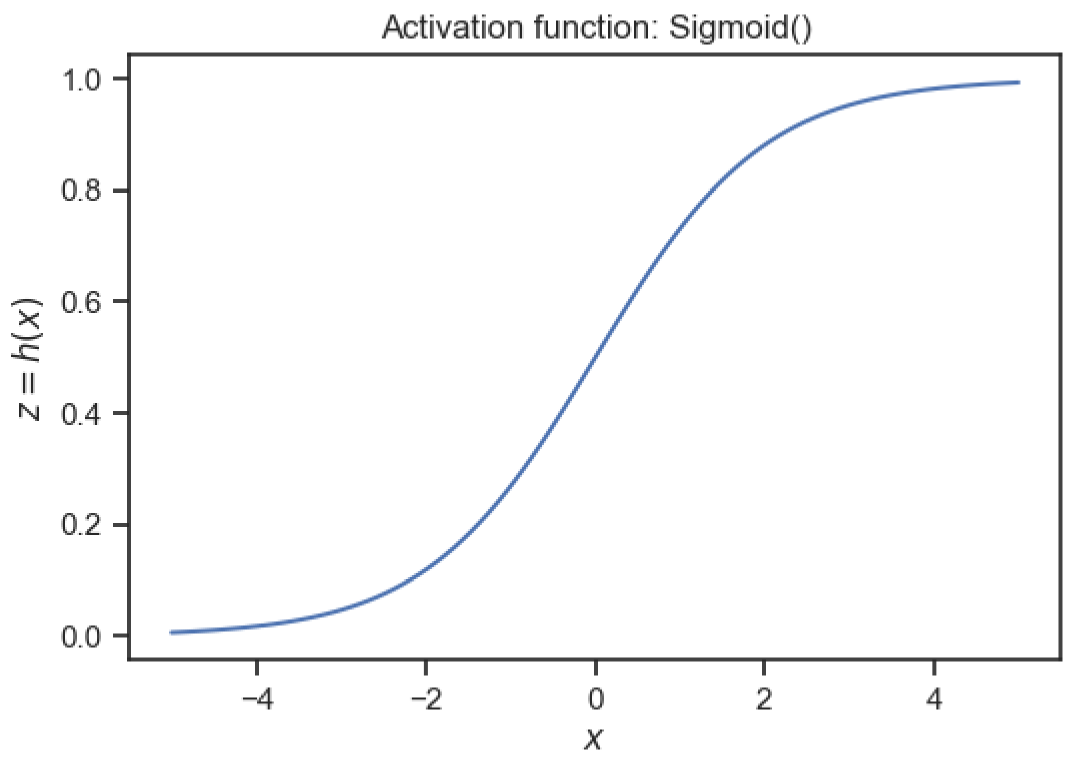
```
Parameter containing:
tensor([ 0.8471,  0.0352, -0.1246, -0.1203, -0.5526, -0.8932, -0.3126,  0.6636,
        -0.0256,  0.9865,  0.1289, -0.1388, -0.5174, -0.6474, -0.4640, -0.7749,
        -0.0695,  0.9404,  0.1930,  0.7792], requires_grad=True)
```

You can directly change them if you wish. Notice the `requires_grad=True` flag. This is because PyTorch knows that these are parameters to be optimized.

There is a little bit of flexibility on `nn.Linear`. For example, you can completly drop the bias if you wish. For the complete list of possibilities, you should always check the docs.

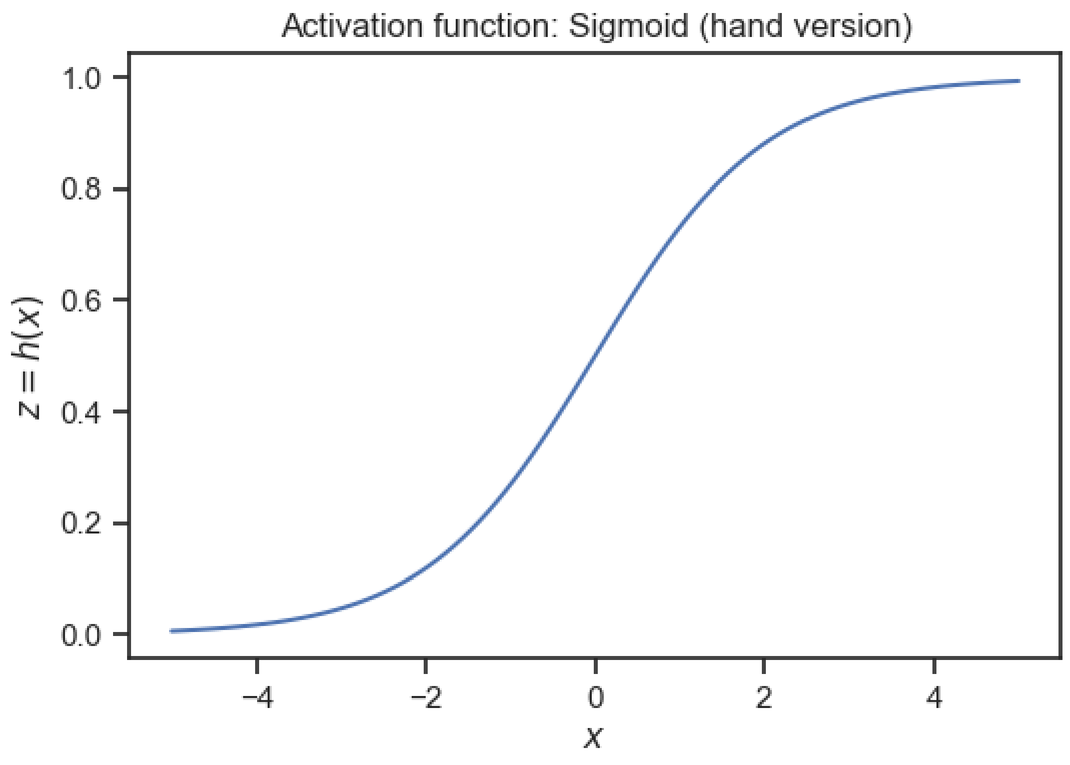
Now, let's get to the activation functions. There are a lot already in `torch.nn`. Here is the sigmoid:

```
h = nn.Sigmoid()

fig, ax = plt.subplots()
x = torch.linspace(-5, 5, 100)[:, None]
ax.plot(x, h(x))
ax.set_xlabel('$x$')
ax.set_ylabel('$z=h(x)$')
ax.set_title('Activation function: ' + str(h));
```



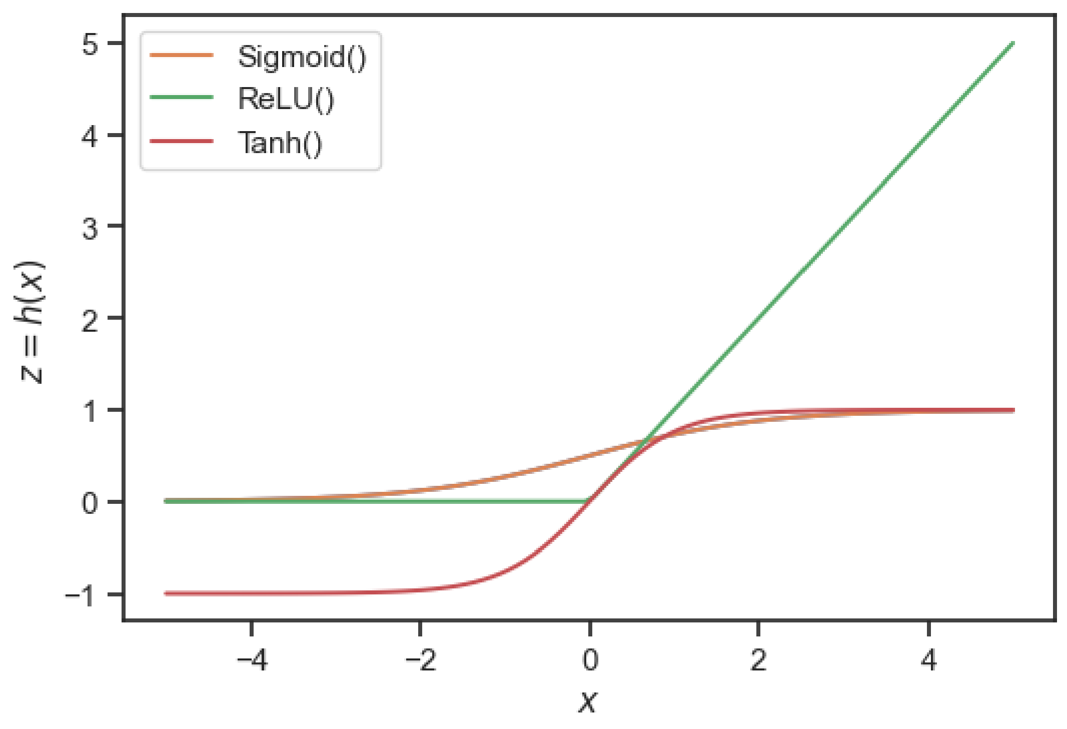
Now, you could also implement the activation fuction by hand. The only restriction is that you should be using `PyTorch` functions instead of `numpy` functions. Here is how we would do it for the sigmoid:

```
# Here is how you could do this by hand:
h_by_hand = lambda x: torch.exp(x) / (1.0 + torch.exp(x))

fig, ax = plt.subplots(dpi=100)
ax.plot(x, h_by_hand(x))
ax.set_xlabel('$x$')
ax.set_ylabel('$z=h(x)$')
ax.set_title('Activation function: Sigmoid (hand version)');
```



Here are now some of the most commonly used activation functions in `torch.nn`:

```
fig, ax = plt.subplots(dpi=100)
ax.plot(x, h_by_hand(x))

for Func in [nn.Sigmoid, nn.ReLU, nn.Tanh]:
    h = Func()
    ax.plot(x, h(x), label=str(h))

ax.set_xlabel('$x$')
ax.set_ylabel('$z=h(x)$')
plt.legend(loc='best');
```



Now that we have a linear layer and an activation function here is how we can combine them to make a function that takes us from the input to the internal neurons:

```
h = nn.Sigmoid()
z_func = lambda x: h(layer(x))
```

This is pretty much it. And that's now a function:

```
z_func(x)
```

```
tensor([[0.3057, 0.8415, 0.0279,  ..., 0.5435, 0.0573, 0.0549],
        [0.3129, 0.8370, 0.0298,  ..., 0.5473, 0.0607, 0.0589],
        [0.3202, 0.8325, 0.0319,  ..., 0.5512, 0.0642, 0.0631],
        ...,
        [0.9204, 0.1776, 0.9595,  ..., 0.8423, 0.9555, 0.9860],
        [0.9228, 0.1728, 0.9621,  ..., 0.8443, 0.9580, 0.9870],
        [0.9252, 0.1681, 0.9645,  ..., 0.8464, 0.9603, 0.9879]],
       grad_fn=<SigmoidBackward0>)
```

Now, for regression, we would like to bring this back to a scalar output. To do this, we need to add one more linear layer taking the 20 internal neurons, back to one dimension.

```
final_layer = nn.Linear(20, 1)
f = lambda x: final_layer(z_func(x))
print(f(x).shape)
```

```
torch.Size([100, 1])
```

Instead of doing this manually, we can can use the class `nn.Sequential` of PyTorch:

```
f = nn.Sequential(layer, nn.Sigmoid(), final_layer)
```

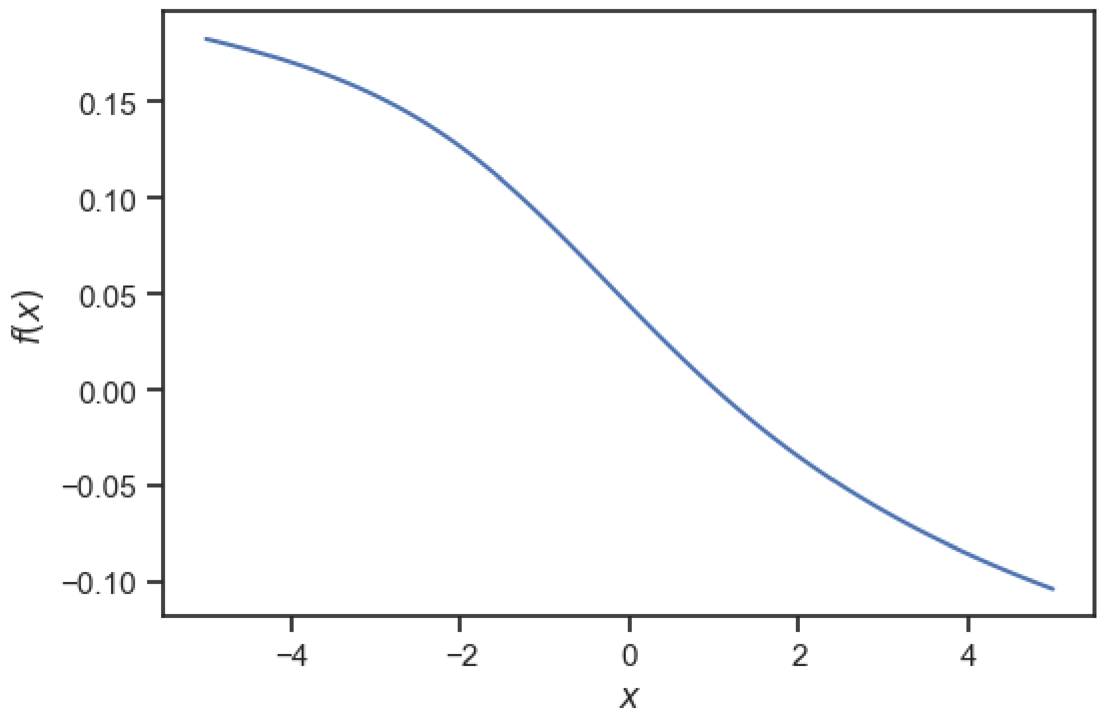
This is the recommended way, because `nn.Sequential` adds some additional functionality which I will show you in a while. You can evaluate this as a function, and you can also plot it. But to plot it, you have to turn the output into a proper numpy array. This is because matplotlib doesn't like PyTorch tensors that depend on parameters. Here is what you need to do:

```
y = f(x).detach().numpy() # detach freezes the parameters to whatever they are
                          # numpy returns a proper numpy array
print(type(y))
print(y.shape)
```

```
<class 'numpy.ndarray'>
(100, 1)
```

And here is how it looks like (remember the weights are random):

```
fig, ax = plt.subplots(dpi=100)
ax.plot(x, f(x).detach().numpy())
ax.set_xlabel('$x$')
ax.set_ylabel('$f(x)$');
```



The class `nn.Sequential` is very convenient, because it allows us to build very deep networks really quickly. Here is a 5-layer network that starts from one input, takes us through 3 layers with 20 neurons each, and ends on a single output:

```
f = nn.Sequential(nn.Linear(1, 20),
                  nn.ReLU(),
                  nn.Linear(20, 20),
                  nn.ReLU(),
                  nn.Linear(20, 20),
                  nn.ReLU(),
                  nn.Linear(20, 20),
                  nn.ReLU(),
                  nn.Linear(20, 1))
```

Where are the parameters in an object created in this way? Here they are:

```
for theta in f.named_parameters():
    print(theta)
```

```
('0.weight', Parameter containing:
tensor([[-0.9494],
        [-0.5797],
        [ 0.7302],
        [-0.0819],
        [ 0.5270],
        [ 0.3091],
        [-0.6618],
        [ 0.6813],
        [ 0.9285],
        [-0.6157],
        [ 0.3522],
        [ 0.7152],
        [ 0.3459],
        [ 0.9119],
        [ 0.9230],
        [-0.8903],
        [-0.5736],
        [-0.1883],
        [ 0.8873],
        [ 0.1048]], requires_grad=True))
('0.bias', Parameter containing:
tensor([ 0.9366,  0.2265,  0.0545, -0.2765,  0.6935, -0.9344,  0.1544, -0.3876,
        -0.9403,  0.5065, -0.4656,  0.4354,  0.1429, -0.0789, -0.4995,  0.0885,
         0.3439,  0.7998, -0.4361, -0.0108], requires_grad=True))
('2.weight', Parameter containing:
tensor([[ 1.1286e-01,  2.1704e-01,  4.2998e-02, -1.6689e-01, -1.0710e-01,
          1.3086e-01, -4.3783e-02,  1.7829e-02, -2.1267e-01,  1.7514e-01,
         -5.2786e-02, -1.8603e-01,  1.5589e-01, -5.7996e-02,  1.7400e-02,
         -1.4377e-01, -1.3804e-01, -5.2051e-02,  4.4931e-02, -1.0996e-01],
        [ 1.1133e-01,  1.9811e-01,  3.3683e-02,  3.7340e-03,  3.1732e-02,
          2.1707e-01, -1.6556e-01,  1.4816e-01,  8.3423e-02, -1.3365e-01,
          9.3574e-02, -5.0519e-02, -9.6680e-02,  1.8784e-01, -1.0347e-01,
         -7.9437e-02, -2.0011e-01, -1.9553e-01,  4.1709e-03,  1.5020e-01],
        [-1.2307e-01, -1.8057e-01,  1.3571e-01,  6.7772e-03,  3.7170e-02,
         -2.2328e-01,  2.1837e-01,  1.4359e-01,  1.8750e-01,  1.1246e-01,
         -2.1440e-01, -2.1217e-01, -6.9194e-02, -8.0945e-02, -1.9293e-01,
          3.9454e-02, -1.0073e-02, -1.7747e-01,  1.3459e-01,  1.7334e-01],
        [-2.1778e-01, -3.6912e-03, -2.0595e-01,  1.6718e-01, -1.5405e-01,
         -5.0630e-02,  1.8191e-01, -1.6468e-01, -1.1635e-01, -9.9457e-02,
         -1.0173e-01,  4.8890e-02, -7.8059e-02,  6.2275e-02,  1.5725e-01,
         -1.2319e-01,  3.3154e-02, -6.1029e-02, -1.0042e-01, -1.4759e-01],
        [ 4.3685e-02, -1.8858e-01,  4.1474e-03,  1.7385e-01, -1.9662e-01,
         -2.8209e-02, -1.7429e-01,  1.7103e-01, -1.2515e-01, -1.2538e-02,
         -4.8820e-02,  1.5188e-01, -1.2229e-01, -7.5595e-02, -2.0628e-01,
          4.0096e-03,  1.6177e-01,  6.4609e-02, -5.9149e-02,  1.3351e-01],
        [ 6.2034e-03, -1.7223e-01, -8.6363e-02, -1.6962e-01,  5.0966e-02,
         -1.9551e-01,  1.5754e-01,  5.1253e-03, -1.3364e-01, -1.1217e-02,
          1.2231e-01,  6.0869e-02, -1.5712e-01,  1.2291e-01,  1.7721e-01,
          2.3785e-02, -3.5375e-02, -2.4353e-02, -1.7061e-01, -7.0361e-02],
        [ 1.2540e-01,  7.9837e-02, -9.9634e-02, -1.4632e-01,  1.7631e-01,
         -9.2228e-03, -8.5045e-02,  1.1987e-01, -5.0555e-03, -2.0261e-01,
          4.4456e-03,  2.2257e-01,  1.1104e-01,  9.2840e-02,  2.0610e-01,
          1.2556e-01,  1.9042e-01, -1.9941e-02,  2.4621e-02, -7.0047e-02],
        [-3.4598e-02, -2.4020e-02, -1.8868e-01, -7.1706e-02, -1.6486e-01,
          1.8154e-01, -7.1214e-02, -1.1328e-01,  1.1141e-01, -1.8487e-01,
         -7.9832e-02,  4.0899e-02,  1.0203e-01, -2.0976e-01, -2.2335e-02,
          1.9190e-02, -1.5422e-01,  4.5026e-02,  1.1360e-01,  7.7903e-02],
        [ 2.0710e-01,  1.8301e-01, -5.3161e-02,  5.7054e-02, -8.5815e-02,
         -3.5122e-02,  1.5379e-01,  9.6770e-02,  2.1063e-01,  1.2809e-01,
         -6.4203e-02, -2.5752e-02,  1.8056e-01, -1.1398e-01,  9.4691e-02,
         -1.5673e-01, -2.0717e-01, -1.5335e-02, -1.0189e-01, -1.6567e-01],
        [ 1.4791e-01, -3.1689e-04, -1.5587e-01, -5.8724e-02, -1.6262e-01,
         -1.4971e-01,  9.0718e-02, -2.8154e-02, -1.5353e-01,  1.7517e-02,
          2.7428e-02,  1.8820e-01,  1.7678e-01,  1.6277e-01,  9.3257e-02,
         -3.1919e-02,  3.2292e-02, -1.9462e-01, -1.1886e-01,  2.0275e-01],
        [ 9.5722e-02,  1.7455e-01,  2.1917e-01, -6.8434e-02,  1.9427e-02,
         -1.5570e-01,  2.0874e-01,  1.3993e-01,  1.7747e-01, -1.3116e-01,
         -2.1111e-01, -1.0630e-01, -1.5332e-01,  2.1300e-02, -2.0511e-01,
          7.3674e-02,  1.4580e-01,  3.3157e-03,  7.1780e-02,  3.0687e-02],
        [-2.0726e-01,  1.2718e-01, -5.4548e-02, -1.0127e-01, -3.4740e-02,
         -1.9979e-01, -1.2061e-01,  4.9397e-05, -1.8856e-01,  1.6381e-01,
         -1.3714e-01, -1.2616e-01, -1.0659e-02, -1.5977e-01, -1.8985e-01,
          1.1887e-01,  2.0250e-01,  1.7879e-02, -2.1805e-01,  8.5303e-02],
        [ 1.8055e-01,  1.3110e-01, -1.6771e-01,  9.6013e-02,  6.1389e-02,
         -1.4238e-01,  7.6929e-02,  2.1691e-01,  1.6883e-01, -1.4136e-01,
         -2.0983e-01, -3.8846e-02,  1.3120e-01,  1.1396e-01,  8.3410e-02,
          8.9229e-03, -1.3163e-02, -1.3073e-01, -4.4455e-02, -1.4802e-01],
        [ 4.3334e-02,  2.1183e-01,  1.9789e-01, -3.4310e-02, -1.0832e-01,
          1.5490e-01, -8.3753e-03,  6.6017e-02,  2.0844e-01,  1.9448e-01,
         -1.9939e-01, -7.0833e-02, -9.0175e-02, -1.7535e-02, -1.8780e-01,
          7.9414e-02,  7.0135e-02,  5.2264e-02,  4.6232e-02, -1.5627e-01],
        [ 8.0567e-02,  2.0681e-01, -1.1085e-01,  2.1964e-01, -8.2196e-02,
         -1.9615e-02, -2.4021e-02,  1.5977e-01, -1.4433e-01,  7.5737e-02,
          1.5061e-02,  5.0628e-02,  1.3028e-01, -2.0920e-01, -9.9662e-02,
          1.2866e-01,  1.6162e-01, -6.8576e-02,  1.6176e-01,  4.2476e-04],
        [-1.3241e-01, -1.6485e-01, -1.5353e-01,  2.2188e-01,  2.1032e-03,
         -1.6531e-01, -8.2625e-02, -1.6550e-01,  6.5604e-02, -1.7354e-01,
         -9.7439e-02, -3.3844e-02,  1.0473e-02, -1.5339e-01, -1.2989e-02,
         -1.2958e-01, -9.2275e-02,  3.7533e-02,  2.0390e-01, -1.9238e-02],
        [ 4.0128e-02, -1.3448e-01,  1.3085e-01,  8.6927e-02,  1.3317e-01,
         -2.6826e-02, -2.2098e-01, -4.0641e-02,  5.6359e-02, -2.7514e-02,
          5.0890e-02, -1.2523e-01,  1.5787e-01,  2.1587e-01,  1.2051e-02,
          6.4355e-02, -1.0784e-01, -1.1378e-01, -7.9827e-02,  9.5914e-02],
        [-2.1966e-01, -5.9495e-02,  5.9051e-02,  8.9960e-02, -1.7609e-01,
         -1.8529e-01,  1.0821e-01, -1.5086e-01, -1.0097e-01, -4.0589e-02,
         -5.5297e-02,  9.7004e-02, -1.1090e-01,  1.7612e-01, -7.8378e-02,
          5.9476e-02,  2.2248e-01, -1.3457e-01, -1.7888e-01,  7.2249e-02],
        [ 3.6224e-03, -1.8087e-01,  1.2755e-01, -4.2859e-02, -7.5791e-02,
         -3.1071e-02, -1.1255e-01, -4.3793e-02, -7.7903e-02,  1.9396e-01,
          4.4949e-02, -5.3819e-02, -1.3944e-01,  3.6411e-03,  1.4437e-01,
          7.8240e-02, -7.8829e-02,  1.9536e-01, -1.1017e-01, -2.0907e-01],
        [ 1.3089e-01, -1.2091e-01,  2.1527e-01, -1.7779e-01, -1.6516e-01,
         -1.7247e-01, -1.5313e-01, -2.6268e-02, -5.5227e-03,  4.4491e-03,
         -4.5666e-02,  2.8763e-02, -1.2080e-02, -2.6132e-02,  2.2254e-01,
         -2.0683e-01,  1.1622e-01, -1.2685e-01, -7.7828e-02, -1.7918e-02]],
       requires_grad=True))
('2.bias', Parameter containing:
tensor([-0.1870, -0.2230,  0.0459, -0.1241,  0.0816, -0.0142, -0.2159,  0.1151,
         0.1728, -0.0444,  0.1517, -0.0792,  0.0751,  0.0144, -0.0273,  0.0424,
         0.0081, -0.0068,  0.0274, -0.0150], requires_grad=True))
('4.weight', Parameter containing:
tensor([[-9.6847e-02, -5.6609e-02,  1.4832e-02, -2.0790e-01,  1.1784e-01,
         -1.9573e-01,  1.4837e-01,  1.4806e-01, -2.2333e-01, -1.7930e-01,
          2.1774e-01, -4.6770e-02,  1.3684e-02, -1.6730e-01,  9.7650e-02,
         -8.0361e-02,  1.8753e-01,  2.3283e-02, -8.7366e-02, -1.3421e-01],
        [ 4.0833e-02, -1.1366e-01,  1.1291e-02,  5.5637e-02,  1.7909e-01,
          3.4869e-02, -1.2493e-01, -6.9102e-02, -1.9032e-01, -2.0830e-02,
          1.5688e-01,  1.5676e-01, -1.5358e-01,  1.3369e-02,  1.9418e-01,
         -1.7481e-01,  1.1164e-01, -1.4498e-01, -1.8586e-01, -2.1647e-01],
        [-8.7459e-02, -6.3400e-02,  1.9542e-01,  1.1394e-01, -1.4834e-02,
          1.7434e-01,  9.3055e-02, -1.0985e-01,  9.6212e-03,  1.0128e-01,
         -1.2157e-01,  1.3659e-01, -1.2553e-01,  1.6471e-01, -4.8534e-02,
         -1.9297e-01,  1.1777e-01, -1.4457e-01,  1.1136e-01, -1.9716e-01],
        [ 8.8831e-02,  7.7287e-02, -1.6985e-01,  1.6007e-01, -1.5286e-02,
         -2.1538e-02, -2.0325e-01,  2.1236e-01,  7.4149e-02,  1.7445e-01,
         -2.0231e-01, -1.8788e-01,  1.8054e-01, -2.1319e-01, -1.9425e-01,
          1.0980e-01, -1.2339e-01, -1.3589e-01,  3.7366e-02, -9.8113e-02],
        [-1.6725e-01,  1.0001e-01,  2.0642e-01,  7.4240e-02,  2.2290e-01,
         -3.9082e-02,  2.0197e-01, -2.1445e-01,  1.5392e-01,  1.2907e-01,
          9.4991e-02, -1.2032e-01, -9.4073e-02,  5.2802e-02,  1.9041e-01,
         -3.9512e-02, -1.7052e-01,  1.0917e-01, -1.6112e-01,  1.3965e-01],
        [-1.2193e-01,  1.4918e-01, -1.5171e-01,  1.0974e-01,  3.8227e-02,
         -1.1070e-01,  2.2128e-01,  9.5196e-02, -6.6172e-02, -1.8404e-01,
         -1.2421e-02,  9.6313e-03, -2.0323e-01, -1.8187e-01,  5.7903e-02,
         -5.2772e-02,  1.4909e-01, -1.7297e-01, -2.1788e-02,  1.2104e-01],
        [-1.5676e-01, -2.0907e-01, -1.4597e-01,  1.2363e-01,  1.9149e-01,
         -1.8121e-01,  1.8128e-01, -3.3656e-02,  1.5283e-01, -1.5276e-01,
         -2.0010e-01, -1.8852e-01, -1.3173e-01, -7.6001e-02, -1.3193e-02,
          5.7346e-02,  7.8066e-02,  5.6380e-03,  1.1428e-01,  1.6908e-01],
        [ 1.2299e-01,  1.3663e-01,  1.0213e-02,  2.0559e-02,  1.4025e-02,
         -9.8833e-02, -1.2489e-01, -1.6195e-01,  6.8887e-02,  3.3679e-02,
          8.9374e-02,  4.1261e-02,  2.0279e-01,  9.7451e-02, -6.0554e-02,
         -2.1639e-01,  1.9170e-01,  9.7324e-02,  9.5817e-02,  2.0641e-01],
        [-1.1409e-01,  1.4918e-01, -2.1018e-01, -1.6794e-02, -2.0630e-01,
          1.2541e-01, -5.8778e-02, -1.7874e-01, -5.7427e-02, -1.4398e-01,
         -1.7068e-01,  7.2487e-02, -5.2767e-02, -1.6759e-01,  1.0157e-01,
         -1.3053e-01, -1.9637e-01,  4.8483e-02, -9.9126e-02, -1.9988e-01],
        [-1.7796e-01,  2.0220e-02, -7.5836e-02,  1.9323e-01, -1.1579e-01,
         -2.1248e-01,  1.3793e-01, -5.0671e-02, -7.5531e-02, -1.6262e-01,
          8.2822e-02,  8.3245e-02, -1.1648e-01, -1.5781e-01,  1.3572e-01,
         -1.8831e-01, -1.1904e-02, -1.1767e-01,  7.7996e-02, -1.1504e-02],
        [ 9.8615e-02,  6.1469e-02,  6.2539e-03, -3.1780e-02,  3.7049e-02,
          1.5249e-01, -1.6889e-02,  1.2469e-01,  6.0953e-02, -1.9427e-01,
         -5.5644e-03,  1.1016e-01, -2.0366e-01, -8.4626e-02, -1.6244e-01,
          1.6428e-01,  1.2293e-01, -1.4791e-01, -1.1174e-01, -3.1043e-02],
        [-1.2941e-01,  1.4192e-02, -1.3035e-01, -5.8555e-02,  3.2495e-02,
          1.4941e-01, -1.6754e-01,  5.9224e-02, -1.5277e-01, -4.2130e-02,
         -8.6223e-02, -5.2132e-02, -2.0355e-01,  6.8886e-02, -1.2402e-01,
          1.5132e-02, -4.9344e-02,  3.5268e-02,  1.4050e-01,  2.1856e-02],
        [ 1.8799e-02,  8.6963e-02,  8.9779e-02,  2.6612e-02,  1.6774e-01,
          2.9601e-02, -5.7513e-02,  1.5845e-01,  9.5999e-02,  2.2463e-02,
          1.7251e-01,  5.4597e-02,  4.6717e-02,  2.0614e-01,  6.2908e-02,
          1.5727e-01, -1.1923e-01, -1.7977e-01, -5.4555e-02,  2.0142e-01],
        [-1.3478e-01, -1.1001e-01, -1.9024e-01, -3.1866e-02,  1.0433e-01,
         -5.8677e-02, -1.8623e-02,  1.6527e-01, -6.4278e-02,  7.8923e-02,
          2.1935e-01, -1.3046e-01,  1.2272e-01, -8.1481e-02,  1.5925e-01,
         -1.2281e-01,  1.5033e-01, -4.7790e-02,  8.1679e-02,  1.2781e-01],
        [ 9.3912e-04,  1.4835e-01, -9.7169e-02,  1.6384e-02,  2.1641e-01,
         -8.4439e-02, -2.1867e-01, -1.4203e-01, -5.2197e-03, -9.7610e-02,
         -5.8008e-02, -5.1534e-02, -5.1376e-02, -2.1071e-01,  1.7606e-01,
         -8.4334e-03,  1.5021e-01, -1.6697e-01,  2.1946e-01,  2.0718e-01],
        [ 6.0677e-02,  7.1970e-02, -2.0755e-01,  1.4563e-01, -5.7799e-02,
          7.8304e-02,  1.1828e-01, -6.7052e-02,  1.9272e-01,  1.5369e-02,
         -1.8272e-01,  1.2251e-01, -7.3870e-02,  6.1877e-02,  1.0187e-01,
          6.7067e-02, -1.0409e-01, -5.2496e-02, -1.8570e-01, -2.2436e-02],
        [ 3.3533e-02,  2.0753e-01,  4.1880e-02, -9.9971e-02, -1.5790e-01,
         -1.8910e-01,  1.4144e-01, -1.4015e-01,  2.0959e-01,  2.1754e-02,
         -1.1255e-01, -1.2617e-02, -1.9756e-01,  8.0401e-02, -2.1066e-02,
         -1.4583e-01, -1.2884e-01,  1.5609e-01, -1.6801e-01,  1.5217e-01],
        [ 1.2761e-01,  9.8483e-02,  1.9434e-02,  6.0762e-02, -1.8425e-01,
          2.1850e-01,  2.2598e-02, -1.9889e-01,  7.0827e-02, -8.6826e-02,
         -9.9342e-02, -1.7476e-01, -1.0504e-01, -2.0723e-01,  1.0367e-01,
          1.1532e-04, -9.5670e-02, -1.0173e-01, -8.4498e-02,  1.4743e-01],
        [ 5.7112e-02, -1.0820e-01, -6.2666e-02,  1.9715e-01,  1.2779e-01,
          8.1203e-02,  9.8768e-02, -1.5818e-01,  1.7736e-01, -1.2674e-01,
         -8.8281e-02,  9.6624e-02,  1.2806e-01,  9.6456e-02,  7.3434e-02,
         -2.1838e-01, -6.2891e-02, -8.3443e-03, -1.2257e-01,  9.2625e-02],
        [-1.7925e-01, -1.2288e-02,  1.9865e-01, -6.7854e-02,  1.6948e-01,
          2.2044e-01, -5.4705e-02,  5.5376e-02, -6.1081e-02,  5.9426e-02,
         -2.0691e-01,  6.7655e-02, -7.1237e-02,  1.5789e-01, -1.1197e-01,
          1.9715e-01,  1.8662e-01, -1.5829e-01,  7.6815e-02, -2.1017e-01]],
       requires_grad=True))
('4.bias', Parameter containing:
tensor([-0.0742, -0.1579,  0.0432,  0.1443,  0.1270, -0.2077, -0.1090,  0.0617,
         0.1050,  0.0394, -0.2080, -0.0351, -0.1365, -0.0497, -0.1778, -0.1794,
        -0.1578,  0.0237, -0.0437, -0.0285], requires_grad=True))
('6.weight', Parameter containing:
tensor([[-4.5287e-02, -8.3195e-02, -2.8556e-02,  1.8254e-02, -1.7835e-01,
         -4.4115e-02, -1.0819e-01,  2.1279e-01, -1.3458e-01,  2.2885e-02,
          6.8459e-02, -5.8238e-03, -3.5342e-02, -1.9900e-01, -1.5968e-01,
         -1.0425e-01,  1.4322e-01,  5.2466e-02, -1.5056e-02, -1.0785e-01],
        [-1.1748e-01,  1.0911e-01,  1.1239e-01,  1.3801e-02,  4.8429e-02,
          2.4115e-02, -2.8121e-02, -1.9609e-01, -1.7712e-01, -1.3917e-01,
         -1.5776e-01,  1.9293e-01,  1.2237e-01, -1.8227e-01,  1.3411e-01,
         -7.6240e-02, -1.4857e-01,  1.9466e-02,  9.5656e-02,  1.7965e-01],
        [-5.7639e-03, -7.9621e-02,  1.8111e-01,  1.0664e-01,  2.0911e-01,
          3.3159e-02,  9.0655e-02, -6.1233e-03, -1.7931e-01, -1.5654e-01,
         -4.5451e-02,  1.4444e-01, -2.1498e-01, -1.8836e-01,  1.4234e-01,
         -4.1664e-02,  1.9534e-01,  2.1781e-01,  6.1657e-03, -8.0940e-02],
        [ 2.0470e-01,  3.4865e-02,  1.1076e-01,  5.7243e-02, -2.1765e-02,
          2.0680e-04,  1.7717e-01,  3.4384e-02, -1.3667e-01,  1.4554e-01,
         -1.3640e-01,  1.0556e-01, -3.3661e-02,  1.6271e-01, -9.6543e-02,
         -5.4877e-02, -1.0541e-01,  1.4933e-01,  1.5578e-01,  1.9258e-01],
        [-1.0964e-01, -1.3449e-01,  1.6418e-02,  5.8185e-02,  1.5291e-01,
          2.0077e-01,  5.0078e-02,  2.0811e-01, -1.7925e-01,  1.4647e-01,
         -4.7401e-02,  9.2006e-02, -2.1833e-01,  4.2239e-02,  4.8425e-02,
         -7.0881e-02,  6.8589e-02,  1.8503e-01, -3.7403e-02,  1.7701e-01],
        [-6.0265e-02,  1.5256e-01,  1.7724e-01,  2.3339e-02,  1.4334e-01,
          1.0833e-01, -1.5352e-01,  8.1396e-02, -4.8733e-02, -1.6502e-01,
         -3.6828e-02,  1.2026e-01,  1.0016e-02,  1.7699e-01, -1.2020e-01,
         -2.0657e-01, -1.8100e-02, -3.9588e-03,  3.6084e-02,  9.6596e-02],
        [ 1.4788e-01,  8.3125e-02,  2.0546e-02, -2.1950e-01, -4.0661e-02,
         -2.2046e-01,  1.9431e-01,  3.0987e-03, -2.1717e-01, -2.1275e-01,
          1.4533e-01, -2.9126e-02, -1.5291e-01,  7.7955e-02,  1.6924e-01,
         -6.1226e-02,  1.4351e-01,  1.2701e-01, -6.9871e-02, -1.4789e-01],
        [-5.8042e-02,  1.4076e-01,  1.7691e-03, -2.1006e-01,  5.4621e-02,
          4.4105e-02, -1.5726e-01, -1.3111e-01,  7.9072e-02,  8.1666e-02,
         -1.3852e-01,  1.6350e-01, -1.2772e-01,  1.2542e-01, -1.8528e-01,
         -1.8601e-01,  7.4122e-02,  9.3150e-03,  1.2091e-01, -1.1776e-01],
        [ 1.8582e-01,  1.7252e-01, -1.4762e-01,  1.0139e-01, -1.2392e-01,
          7.5408e-02, -2.1546e-01, -1.6713e-01, -1.0545e-02,  7.9118e-03,
          4.6670e-02,  1.2832e-01,  6.4337e-02,  6.6786e-02,  1.8039e-01,
          6.5942e-03, -2.2263e-01, -1.0125e-01, -2.2086e-01,  2.1005e-01],
        [ 2.1290e-01, -8.1663e-03,  5.2744e-02,  1.0967e-01,  1.2434e-01,
         -1.9883e-01, -1.8110e-01, -1.8356e-01,  1.7305e-01, -9.5379e-02,
          1.6559e-02, -4.5791e-02, -8.4984e-02, -9.5903e-02,  1.5696e-01,
          4.4359e-02, -1.3970e-01,  1.9010e-02, -1.1388e-01,  2.1662e-01],
        [ 2.8528e-02,  2.0214e-01, -8.1869e-02, -2.7515e-02, -4.4848e-02,
          2.1481e-01,  1.4178e-01, -6.8197e-03,  1.5899e-01, -1.0869e-01,
          1.1900e-01,  7.5389e-02, -4.8459e-02,  1.1407e-01, -1.9847e-01,
         -1.7301e-01,  1.9690e-01, -1.4534e-02, -8.5117e-02,  3.6505e-02],
        [ 1.9238e-01,  6.9071e-04, -3.1796e-02, -2.4310e-02, -1.4493e-01,
         -1.0768e-01,  1.5244e-02,  1.7272e-01, -1.1210e-01, -9.1478e-02,
         -3.8587e-02, -1.0627e-01,  2.1208e-01, -6.6862e-02, -2.1042e-01,
         -2.1005e-01,  2.1363e-01, -5.7925e-02, -8.0585e-02, -2.8352e-02],
        [ 2.0862e-01, -1.8417e-01,  2.6705e-02,  8.4730e-02,  1.9516e-01,
         -1.5765e-01,  1.9030e-01, -5.1109e-02,  1.9100e-01,  3.3822e-02,
         -5.9636e-02, -4.6285e-02,  1.9125e-03, -7.3613e-02,  4.7574e-02,
          1.1894e-02,  1.0204e-01,  1.5095e-01, -2.0040e-01,  1.0191e-01],
        [-1.9472e-01, -9.7581e-02,  1.4861e-01, -3.2456e-02,  2.2149e-01,
          1.4704e-01,  1.1195e-01, -7.1284e-02,  7.3337e-02, -1.0149e-01,
          1.7256e-01, -1.6171e-02, -1.0782e-01,  1.1917e-01, -1.6484e-01,
         -1.2842e-01, -1.8489e-01,  6.0597e-02, -2.1468e-01, -8.2782e-02],
        [-9.8085e-02,  7.2005e-02,  1.5927e-02, -1.9203e-01, -8.8372e-02,
          3.9464e-02,  8.5400e-02, -5.6142e-02, -1.4675e-01,  1.2983e-01,
          1.5170e-01,  5.4788e-02,  3.2388e-03, -3.3445e-02,  6.6920e-02,
         -3.1059e-02,  4.6318e-02,  1.2596e-01,  1.7106e-01, -1.7698e-01],
        [-2.2281e-01,  5.7994e-02,  3.2657e-02,  1.0413e-01,  2.1453e-01,
         -9.6278e-02, -1.6712e-01,  1.3642e-01,  6.1346e-02, -9.6854e-02,
         -1.0650e-01,  2.2055e-01,  1.5724e-01,  8.1113e-02,  1.6177e-01,
          3.4003e-02, -1.3707e-01,  1.8064e-01, -1.3560e-01, -7.9006e-02],
        [-1.5643e-01, -1.0474e-02, -1.5855e-01, -8.3977e-02, -1.0564e-01,
          1.9140e-01, -1.0760e-01, -1.6723e-01,  8.0814e-02,  1.5609e-01,
          5.5449e-03, -9.6396e-04, -1.5040e-01,  2.0071e-01, -9.9433e-02,
          9.3875e-03,  3.6308e-02,  3.9853e-02,  2.8754e-02,  2.1989e-02],
        [-5.2658e-02, -1.1017e-01,  1.4543e-02, -1.2602e-01,  9.6084e-02,
         -6.7551e-02,  1.1702e-01, -1.1347e-01,  8.7280e-02,  1.4479e-01,
         -1.7608e-01,  1.4192e-01,  1.5857e-01, -1.1922e-01, -1.9090e-01,
          3.9634e-02,  1.9651e-01, -1.2236e-01,  1.4349e-01, -3.9810e-02],
        [ 6.7728e-02,  2.2213e-01, -8.5013e-02,  8.3224e-02, -1.1423e-01,
         -7.1883e-02,  1.5125e-01,  9.3376e-02,  1.2651e-01,  1.8770e-01,
          1.9229e-01,  3.0308e-02, -1.3999e-01,  1.7240e-03, -4.1024e-02,
         -1.2860e-02, -7.4642e-02,  6.6454e-02, -1.2193e-01, -5.2616e-02],
        [-6.4060e-02,  1.9528e-01,  1.1405e-01,  1.6761e-02, -4.3079e-02,
          8.9329e-02, -1.8684e-01,  2.0815e-01,  7.9431e-02,  1.0215e-01,
          3.9483e-02, -7.3377e-02, -1.8116e-01,  9.2645e-02, -1.8345e-01,
          1.1427e-01, -1.0790e-02,  1.7912e-01, -1.0356e-01, -1.4191e-01]],
       requires_grad=True))
('6.bias', Parameter containing:
tensor([ 0.1343,  0.1400,  0.1955,  0.2233,  0.2236,  0.0229, -0.0282,  0.1888,
         0.1885, -0.1224, -0.0165, -0.1306, -0.0932,  0.1611,  0.1641, -0.2067,
        -0.2125, -0.1150,  0.1468,  0.0748], requires_grad=True))
('8.weight', Parameter containing:
tensor([[-0.0924, -0.2204,  0.2068, -0.2059, -0.1919,  0.1608, -0.1115,  0.0401,
         -0.1463,  0.2155, -0.1146,  0.1280,  0.0047,  0.0684, -0.0690,  0.1451,
         -0.0257,  0.0228,  0.2163,  0.2113]], requires_grad=True))
('8.bias', Parameter containing:
tensor([-0.1322], requires_grad=True))
```

And that's why we love PyTorch. Because it does all the dirty work for us. Imagine having to keep track of all these parameters by hand.

For those of you who want to know what is going on inside `nn.Linear`, just note that it is a special case of a PyTorch neural network module, see nn.Module. The latter is what you would directly inherit when writing your own class for a non-standard neural network. We are not going to cover it in this class, but you can find plenty of examples here.

Making a loss function

Let's now make the loss function that we want to minimize. It needs to be a `PyTorch` function as well. For regression problems, we can think of the loss as a function of the model predictions and the observed data. That is the depends on the parameters comes through the predictions. Let's write down the mean square error (MSE) loss in this form. It is:

$$L_{\text{MSE}}(\theta) = L_{\text{MSE}}(y_{1:n}, f(x_{1:n};\theta)) = \frac{1}{n}\sum_{i=1}^{n}\left[y_i - f(x_i;\theta)\right]^2,$$

where $x_{1:n}$ are the observed inputs (features), $y_{1:n}$ are the observed outputs (targets), and $f(x_{1:n};\theta)$ contains the model predictions on the observed inputs.

You can implement the MSE loss like this:

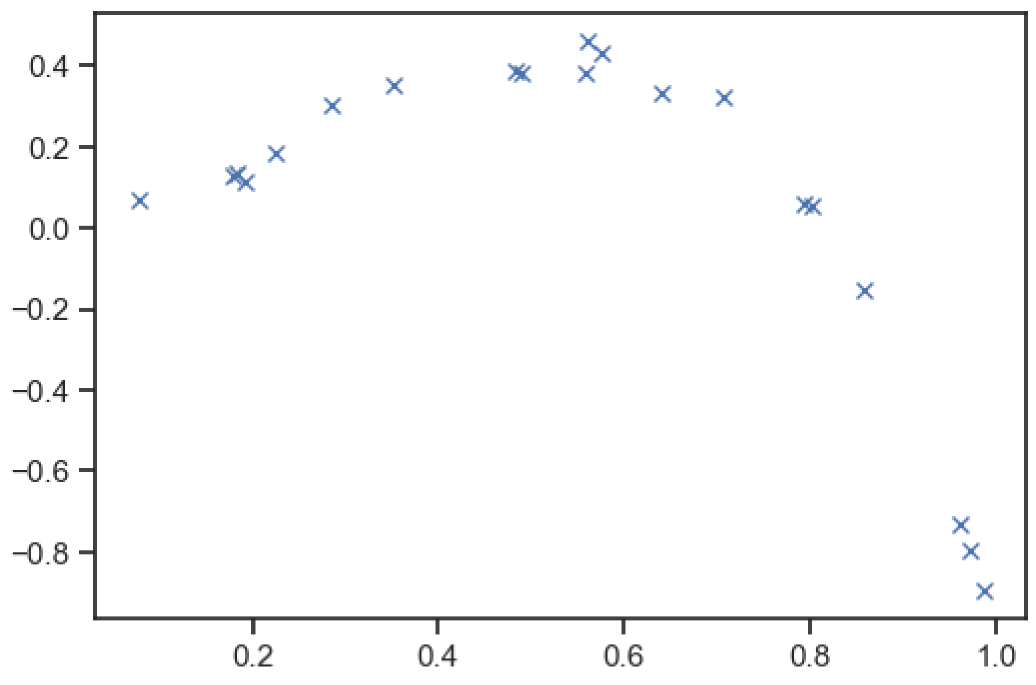
```
mse_loss_ours = lambda y, f: torch.mean((y - f) ** 2)
```

Or we can use built-in PyTorch functionality:

```
mse_loss = nn.MSELoss()
```

Let's evaluate it for some random data:

```
# The number of fake observations
n = 20
# Some fake observed features
x_fake = torch.rand(n, 1)
# Some fake observed targets
y_fake = 4 * x_fake ** 2 - 5 * x_fake ** 3 + 0.1 * torch.rand(n, 1)
fig, ax = plt.subplots(dpi=100)
ax.plot(x_fake, y_fake, 'x');
```



And here is how to calculate the loss (for the random parameters that our net started with):

```
# Predict with the net:
y_pred = f(x_fake)
# Evaluate the loss
our_loss = mse_loss_ours(y_fake, y_pred)
built_in_loss = mse_loss(y_fake, y_pred)
print(our_loss)
print(built_in_loss)
```

```
tensor(0.2232, grad_fn=<MeanBackward0>)
tensor(0.2232, grad_fn=<MseLossBackward0>)
```

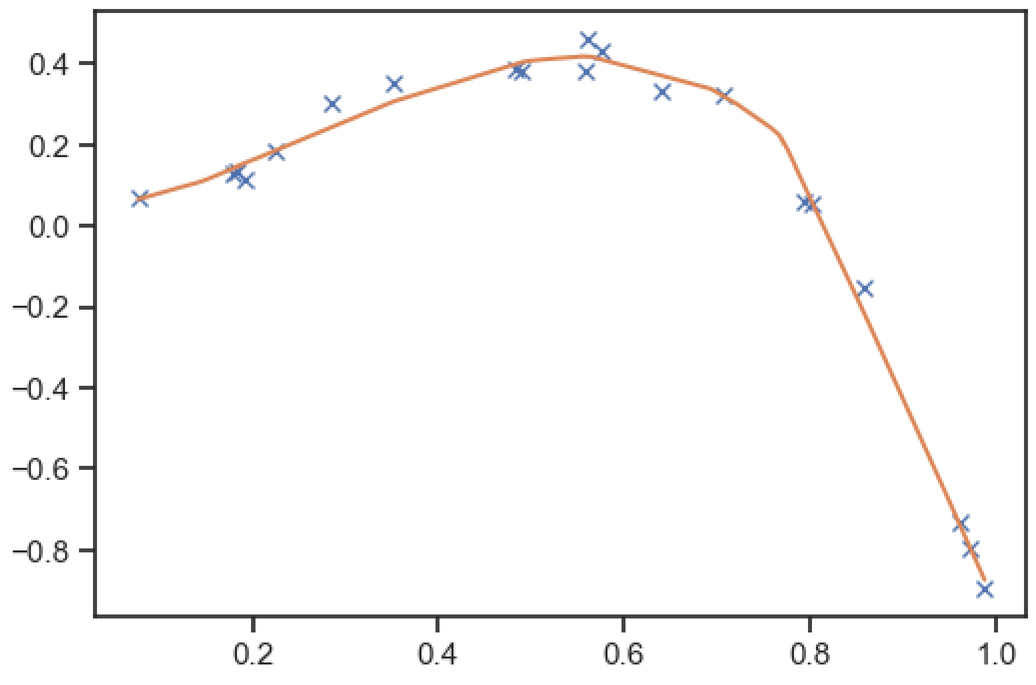
Now, let’s just minimize the MSE loss for these fake data and see what kind of fit we are going to get. Here is how you do this in PyTorch. Since I don’t have a lot of data, I will just use gradient descent - no randomly subsampling the data. I will show you how you can use stochastic gradient descent in the next example.

```
# Reinitialize the net:
f = nn.Sequential(nn.Linear(1, 20),
                  nn.ReLU(),
                  nn.Linear(20, 20),
                  nn.ReLU(),
                  nn.Linear(20, 20),
                  nn.ReLU(),
                  nn.Linear(20, 20),
                  nn.ReLU(),
                  nn.Linear(20, 1))

# Initialize the optimizer - Notice that it needs to know about the 
# parameters it is optimizing
optimizer = torch.optim.SGD(f.parameters(), lr=0.01) # lr is the learning rate
# Some place to hold the training loss for visualizing it later
training_loss = []
# Iterate the optimizer. Let's just do 10 iterations.
for i in range(10000):
    # This is essential for the optimizer to keep
    # track of the gradients correctly
    # It is using some buffers internally that need to
    # be manually zeroed on each iteration.
    # This is because it doesn't know when you are done with the
    # calculation of the loss
    optimizer.zero_grad()
    # Make predictions
    y_pred = f(x_fake)
    # Evaluate the loss - That's what you are minimizing
    loss = mse_loss(y_fake, y_pred)
    # Evaluate the derivative of the loss with respect to
    # all parameters - It knows how to do it because of
    # PyTorch magick
    loss.backward()
    # And now you are ready to make a step
    optimizer.step()
    # Save the training loss of later visualization
    training_loss.append(loss.item())
    # Print the loss every one hundend iterations:
    if i % 1000 == 0:
        print('it = {0:d}: loss = {1:1.3f}'.format(i, loss.item()))
```

```
it = 0: loss = 0.183
it = 1000: loss = 0.157
it = 2000: loss = 0.111
it = 3000: loss = 0.033
it = 4000: loss = 0.009
it = 5000: loss = 0.002
it = 6000: loss = 0.001
it = 7000: loss = 0.001
it = 8000: loss = 0.001
it = 9000: loss = 0.001
```

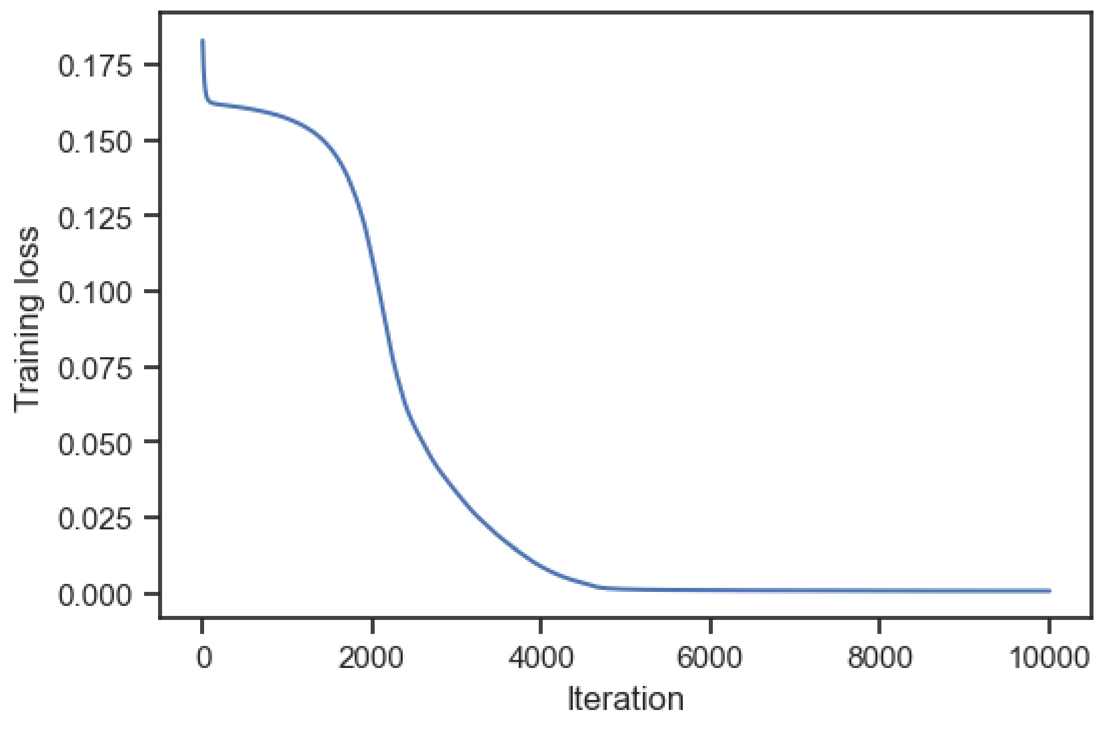
Let’s plot the predictions of this model on the fake data:

```
fig, ax = plt.subplots(dpi=100)
ax.plot(x_fake, y_fake, 'x');
xx = torch.linspace(x_fake.min(), x_fake.max(), 100)[:, None]
yy = f(xx).detach().numpy()
ax.plot(xx, yy);
```



Now, this may or may not work depending on what random seed you start with. If you run it a few times it may get stack at some local minimum. Unless we do stochastic optimization, i.e., subsampling the data, this is not a very good algorithm. Here how the loss changes with each iteration:

```
fig, ax = plt.subplots(dpi=100)
ax.plot(training_loss)
ax.set_xlabel('Iteration')
ax.set_ylabel('Training loss');
```



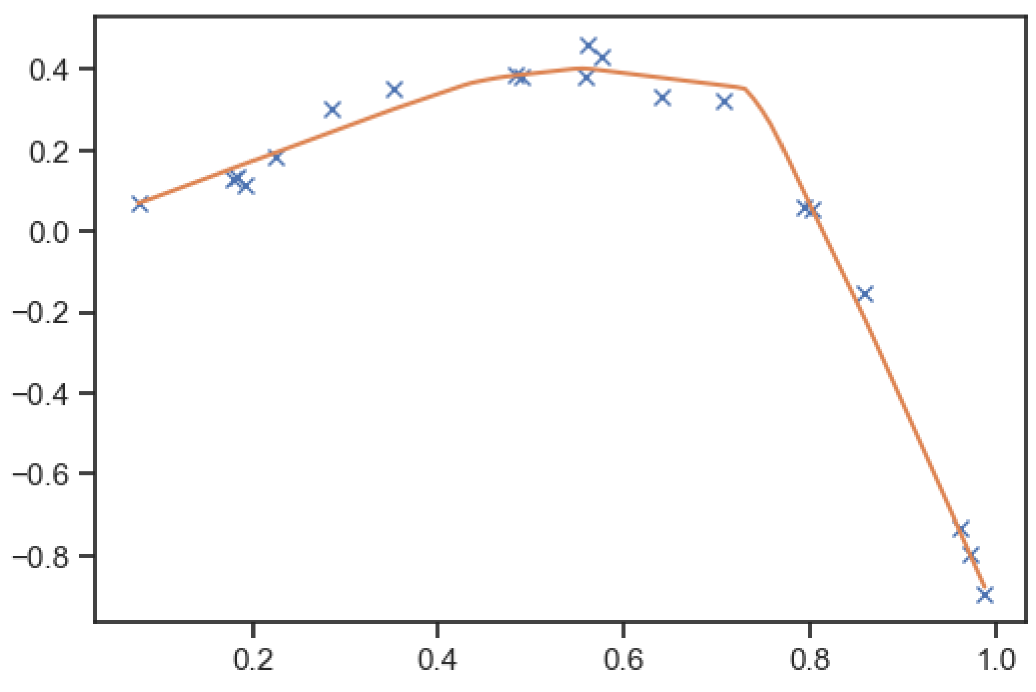
The problem is the plato we have at the beginning of the optimization.

Let’s redo this thing with stochastic optimization. For stochastic optimization we need to subsample the data during each iteration. We can either do this manually or using PyTorch functionality. First, let’s do it manually.

```
# Pick a subsampling batch size
m = 5

# Reinitialize the net:
f = nn.Sequential(nn.Linear(1, 20),
                  nn.ReLU(),
                  nn.Linear(20, 20),
                  nn.ReLU(),
                  nn.Linear(20, 20),
                  nn.ReLU(),
                  nn.Linear(20, 20),
                  nn.ReLU(),
                  nn.Linear(20, 1))

# Reinitialize the optimizer
optimizer = torch.optim.SGD(f.parameters(), lr=0.01)
# Keep track of the training loss
training_loss_sgd = []
# Iterate the optimizer. Let's just do 10 iterations.
for i in range(10000):
    # Zero out the gradient buffers
    optimizer.zero_grad()
    # Sample m observation indices at random
    idx = np.random.randint(0, n, m)
    # Here is the subsample of the data
    x_batch = x_fake[idx]
    y_batch = y_fake[idx]
    # Make predictions
    y_pred = f(x_batch)
    # Evaluate the loss - That's what you are minimizing
    loss = mse_loss(y_batch, y_pred)
    # Evaluate the derivative of the loss with respect to
    # all parameters - It knows how to do it because of
    # PyTorch magick
    loss.backward()
    # And now you are ready to make a step
    optimizer.step()
    # Keep track of the training loss
    training_loss_sgd.append(loss.item())
    # Print the loss every one hundend iterations:
    if i % 1000 == 0:
        print('it = {0:d}: loss = {1:1.2e}'.format(i, loss.item()))
```

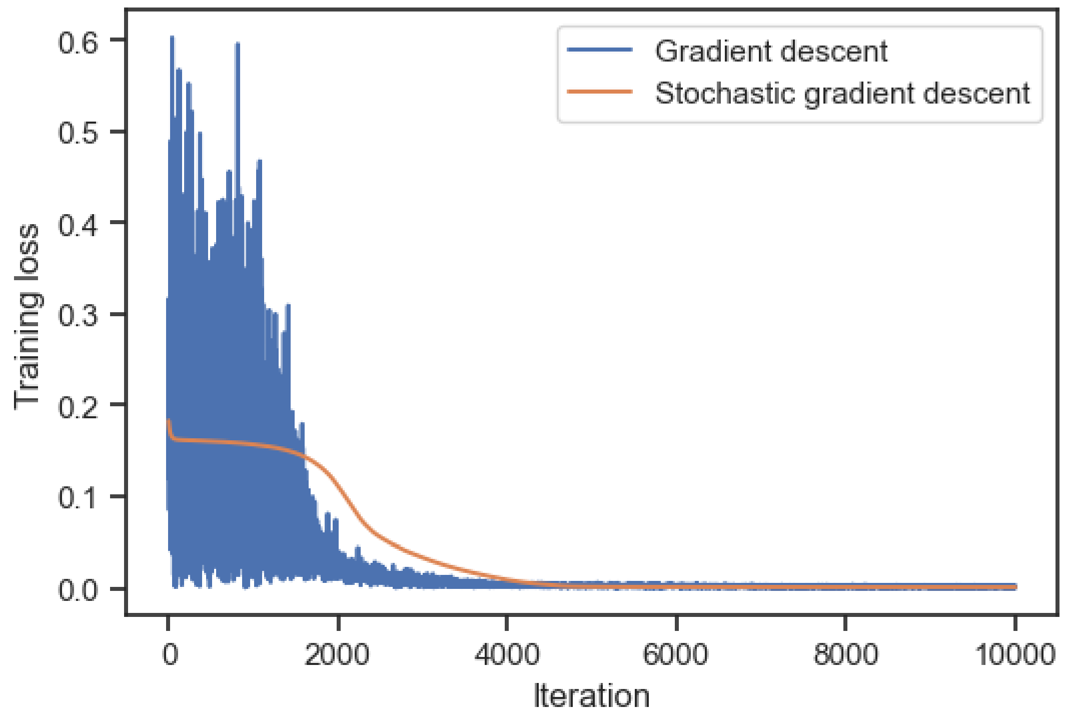
```
it = 0: loss = 2.15e-01
it = 1000: loss = 1.52e-01
it = 2000: loss = 4.05e-02
it = 3000: loss = 4.89e-03
it = 4000: loss = 2.33e-03
it = 5000: loss = 1.82e-03
it = 6000: loss = 1.93e-03
it = 7000: loss = 2.63e-03
it = 8000: loss = 1.96e-03
it = 9000: loss = 6.10e-04
```

```
fig, ax = plt.subplots(dpi=100)
ax.plot(x_fake, y_fake, 'x');
xx = torch.linspace(x_fake.min(), x_fake.max(), 100)[:, None]
yy = f(xx).detach().numpy()
ax.plot(xx, yy);
```



This fit does look a little bit better. Let's now compare the training loss of stochastic gradient descent to the previous one:

```
fig, ax = plt.subplots(dpi=100)
ax.plot(training_loss_sgd, label='Gradient descent')
ax.plot(training_loss, label='Stochastic gradient descent')
ax.set_xlabel('Iteration')
ax.set_ylabel('Training loss')
plt.legend(loc='best');
```



It is this wiggly nature of stochastic gradient descent that allows it to escape bad local minima.

Questions

- Rerun the stochastic gradient descent with one sample per iteration ($m = 1$). Does it converge? Do you need less or more iterations? Is it more or less wiggly?
- Rerun the stochastic gradient descent with 10 samples per iteration. How doe it perfom now?

Example - Motorcyle Data

Let's now use the motorcycle data to do regression with DNNs. This will help us demonstrate some best practices and specifically:

- Standarizing the data
- Splitting in training and test subsets

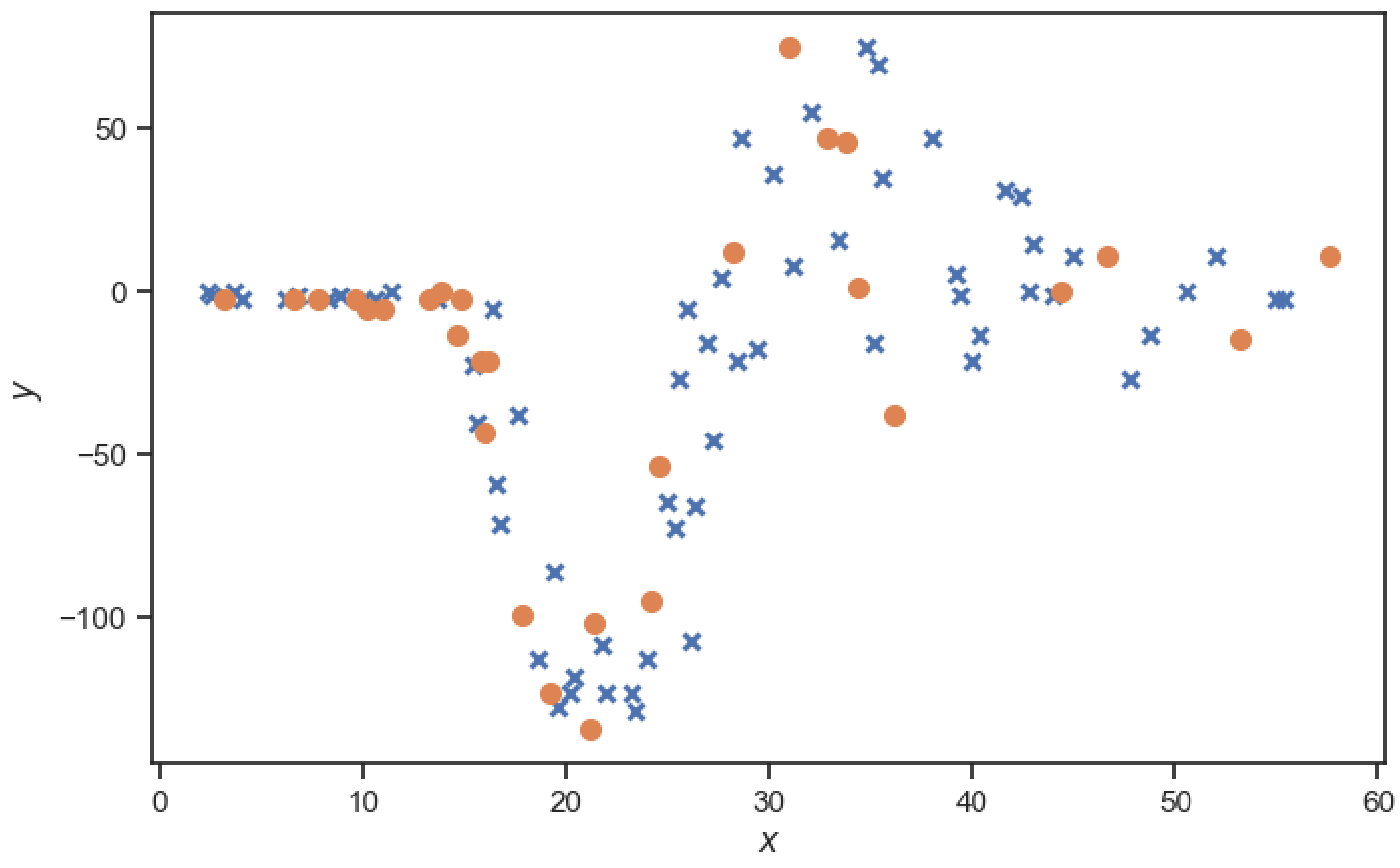
First, start by loading the dataset:

```
url = "https://github.com/PredictiveScienceLab/data-analytics-
se/raw/master/lecturebook/data/motor.dat"
download(url)
```

```
# Load the data
data = np.loadtxt('motor.dat')
x = data[:, 0][:, None]
y = data[:, 1][:, None]
```

```
# Split into training and test datasets
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3)
```

```
# Visualize them
fig, ax = plt.subplots(figsize=(8, 5))
ax.plot(x_train, y_train, 'x', markeredgewidth=2, label='Training data')
ax.plot(x_test, y_test, 'o', markeredgewidth=2, label='Test data')
ax.set_xlabel('$x$')
ax.set_ylabel('$y$');
```



```
# Turn the data into torch tensors:
x_train = torch.tensor(x_train, dtype=torch.float)
y_train = torch.tensor(y_train, dtype=torch.float)
x_test = torch.tensor(x_test, dtype=torch.float)
y_test = torch.tensor(y_test, dtype=torch.float)
```

Please note that the specification of `dtype=torch.float` is absolutely needed here. If you don't include it the code is not going to work. The problem is that the `x_train` etc. are all numpy arrays and that numpy arrays have 64-bit floating point numbers by default. PyTorch is using 32-bit floating point numbrs by default. We need at some point make the two compatible.

Now we are ready to train the network. Let's give it a shot. We will use the same architecture as before. The only difference is that I will be printing the validation loss instead of the training loss.

```
# The number of training samples
n = x_train.shape[0]

# Pick a subsampling batch size
m = 5

# Reinitialize the net:
f = nn.Sequential(nn.Linear(1, 20),
                  nn.ReLU(),
                  nn.Linear(20, 20),
                  nn.ReLU(),
                  nn.Linear(20, 20),
                  nn.ReLU(),
                  nn.Linear(20, 20),
                  nn.ReLU(),
                  nn.Linear(20, 1))

# Reinitialize the optimizer
optimizer = torch.optim.SGD(f.parameters(), lr=0.01)
# Keep track of the training loss and the test loss
training_loss = []
test_loss = []
# Iterate the optimizer. Let's just do 10 iterations.
for i in range(10000):
    # Zero out the gradient buffers
    optimizer.zero_grad()
    # Sample m observation indices at random
    idx = np.random.randint(0, n, m)
    # Here is the subsample of the data
    x_batch = x_train[idx]
    y_batch = y_train[idx]
    # Make predictions
    y_pred = f(x_batch)
    # Evaluate the loss - That's what you are minimizing
    loss = mse_loss(y_batch, y_pred)
    training_loss.append(loss.item())
    # Evaluate the derivative of the loss with respect to
    # all parameters - It knows how to do it because of
    # PyTorch magick
    loss.backward()
    # And now you are ready to make a step
    optimizer.step()
    # Evaluate the test loss
    y_pred_test = f(x_test)
    ts_loss = mse_loss(y_test, y_pred_test)
    test_loss.append(ts_loss.item())
    # Print the loss every one hundend iterations:
    if i % 1000 == 0:
        print('it = {0:d}: loss = {1:1.2e}'.format(i, ts_loss.item()))
```

```
it = 0: loss = 4.05e+06
it = 1000: loss = nan
it = 2000: loss = nan
it = 3000: loss = nan
it = 4000: loss = nan
it = 5000: loss = nan
it = 6000: loss = nan
it = 7000: loss = nan
it = 8000: loss = nan
it = 9000: loss = nan
```

The above code may not work at all, giving you nan's. Or it may work and get you nowhere. The problem here is the scale of both the inputs and the outputs and the assumptions that have been made about them when we initialize the weights of the net and when we picked the learning rate of the optimization algorithm. The easiest way to overcome this problem is to *standarize the data*. This is achieved by subtracting the empirical mean and dividing by the empirical standard deviation both the inputs and the outputs. By standarizing the data, we are making the default paramerameters (for weight initialization and stochastic gradient descent) valid again. Standarization is such a common process that it is already implemented in sklearn.preprocessing.StandardScaler. Here is how it works:

```
from sklearn.preprocessing import StandardScaler

feature_scaler = StandardScaler().fit(x)
target_scaler = StandardScaler().fit(y)
```

The `feature_scaler.transform()` is a function: \$$x \to \frac{x-\mu}{\sigma}, where \backslash mu and \backslash$sigma\$ are the empirical mean and standard deviation of the features. Here they are:

```
# The mean:
feature_scaler.mean_
```

```
array([25.9])
```

```
# The standard deviation:
feature_scaler.scale_
```

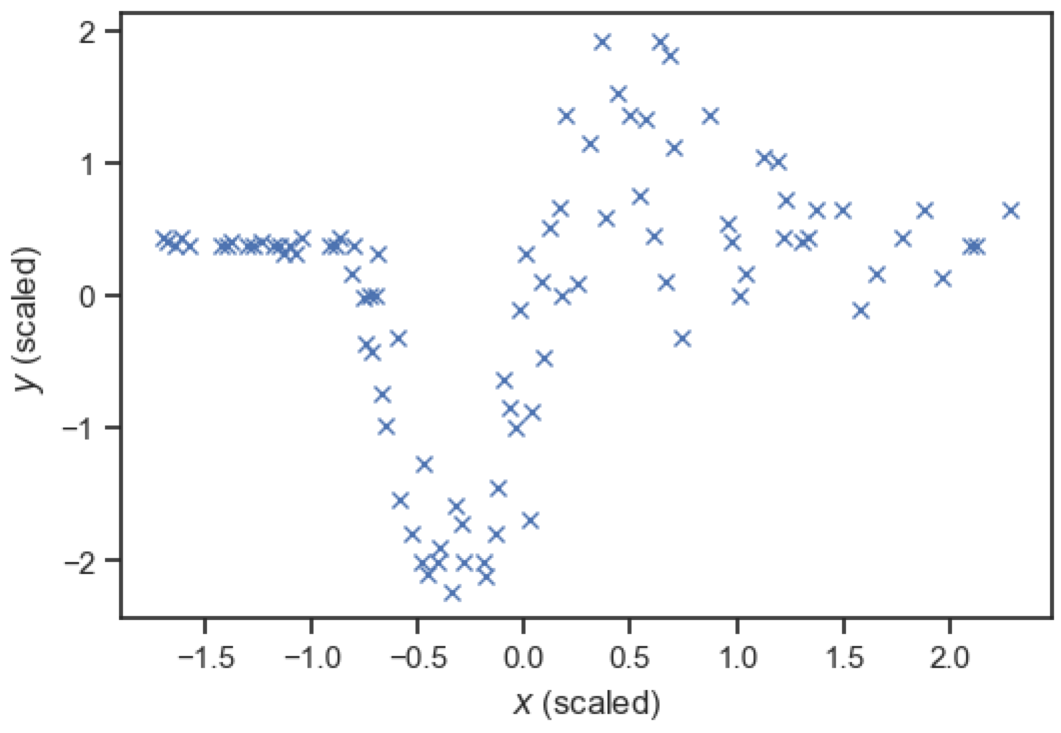
```
array([13.88])
```

And here is how the scalers work:

```
x_scaled = feature_scaler.transform(x)
y_scaled = target_scaler.transform(y)
```

The data are now scaled, see this fig:

```
fig, ax = plt.subplots(dpi=100)
ax.plot(x_scaled, y_scaled, 'x')
ax.set_xlabel('$x$ (scaled)')
ax.set_ylabel('$y$ (scaled)');
```



We will train the net using `x_scale` and `y_scaled`. We can always go back to the original scales at the end. Let's see if it works.

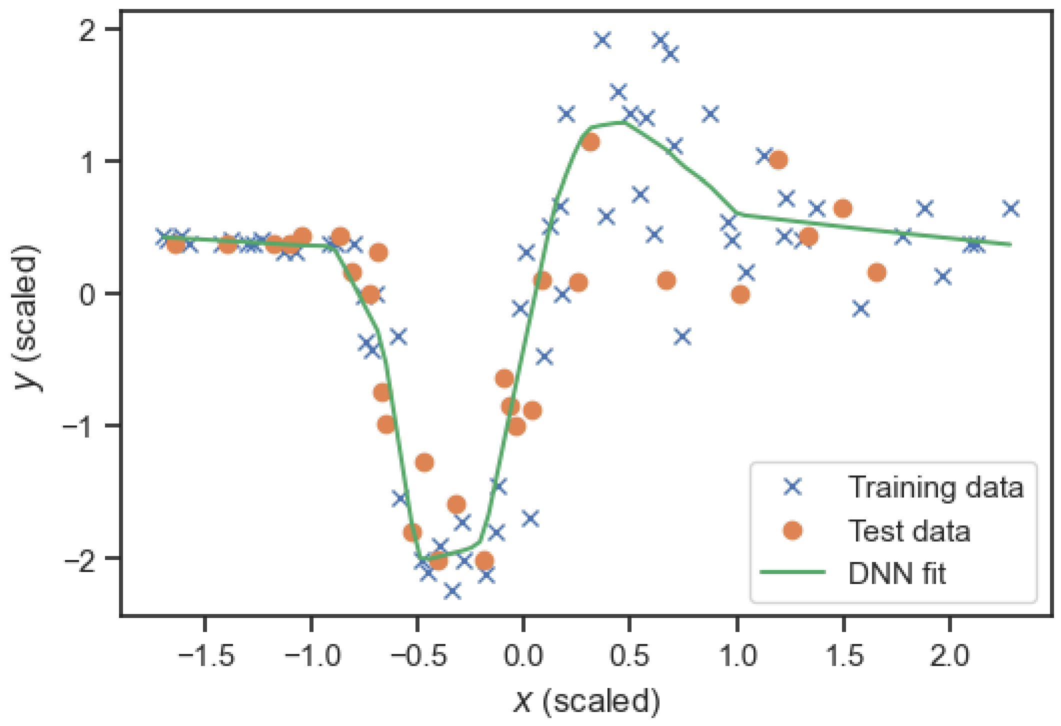
```
# Split in training and test
x_s_train, x_s_test, y_s_train, y_s_test = train_test_split(x_scaled, y_scaled,
                                                            test_size=0.3)

# Turn the data into torch tensors:
x_s_train = torch.tensor(x_s_train, dtype=torch.float)
y_s_train = torch.tensor(y_s_train, dtype=torch.float)
x_s_test = torch.tensor(x_s_test, dtype=torch.float)
y_s_test = torch.tensor(y_s_test, dtype=torch.float)

# The number of training samples
n = x_train.shape[0]

# Pick a subsampling batch size
m = 5

# Reinitialize the net:
f = nn.Sequential(nn.Linear(1, 20),
                  nn.ReLU(),
                  nn.Linear(20, 20),
                  nn.ReLU(),
                  nn.Linear(20, 20),
                  nn.ReLU(),
                  nn.Linear(20, 20),
                  nn.ReLU(),
                  nn.Linear(20, 1))

# Reinitialize the optimizer
optimizer = torch.optim.SGD(f.parameters(), lr=0.01)
# Keep track of the training loss and the test loss
training_loss = []
test_loss = []
# Iterate the optimizer. Let's just do 10 iterations.
for i in range(10000):
    # Zero out the gradient buffers
    optimizer.zero_grad()
    # Sample m observation indices at random
    idx = np.random.randint(0, n, m)
    # Here is the subsample of the data
    x_batch = x_s_train[idx]
    y_batch = y_s_train[idx]
    # Make predictions
    y_pred = f(x_batch)
    # Evaluate the loss - That's what you are minimizing
    loss = mse_loss(y_batch, y_pred)
    training_loss.append(loss.item())
    # Evaluate the derivative of the loss with respect to
    # all parameters - It knows how to do it because of
    # PyTorch magick
    loss.backward()
    # And now you are ready to make a step
    optimizer.step()
    # Evaluate the test loss
    y_pred_test = f(x_s_test)
    ts_loss = mse_loss(y_s_test, y_pred_test)
    test_loss.append(ts_loss.item())
    # Print the loss every one hundend iterations:
    if i % 1000 == 0:
        print('it = {0:d}: loss = {1:1.2e}'.format(i, ts_loss.item()))
```

```
it = 0: loss = 9.26e-01
it = 1000: loss = 3.26e-01
it = 2000: loss = 2.00e-01
it = 3000: loss = 1.99e-01
it = 4000: loss = 1.75e-01
it = 5000: loss = 1.62e-01
it = 6000: loss = 1.76e-01
it = 7000: loss = 1.84e-01
it = 8000: loss = 1.84e-01
it = 9000: loss = 2.30e-01
```

Let’s visualize the fit:

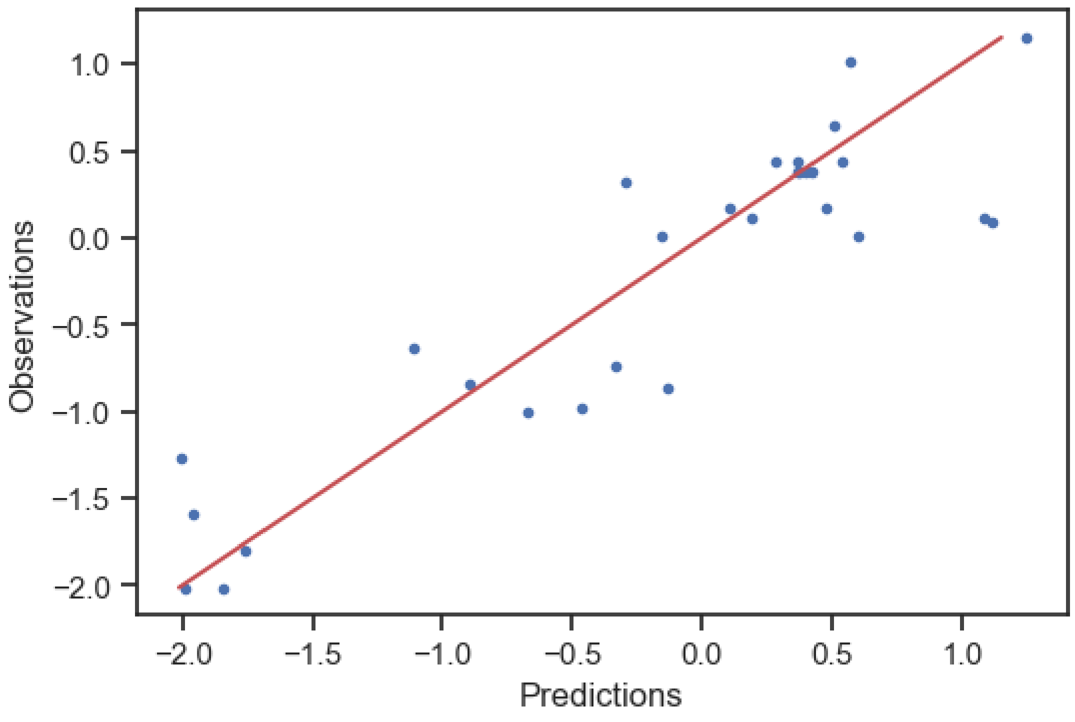
```
xx_scaled = torch.linspace(x_scaled.min(), x_scaled.max(), 100)[:, None]
yy_scaled = f(xx_scaled).detach().numpy()
fig, ax = plt.subplots(dpi=100)
ax.plot(x_s_train, y_s_train, 'x', label='Training data')
ax.plot(x_s_test, y_s_test, 'o', label='Test data')
ax.plot(xx_scaled, yy_scaled, label='DNN fit')
ax.set_xlabel('$x$ (scaled)')
ax.set_ylabel('$y$ (scaled)')
```

```
plt.legend(loc='best');
```



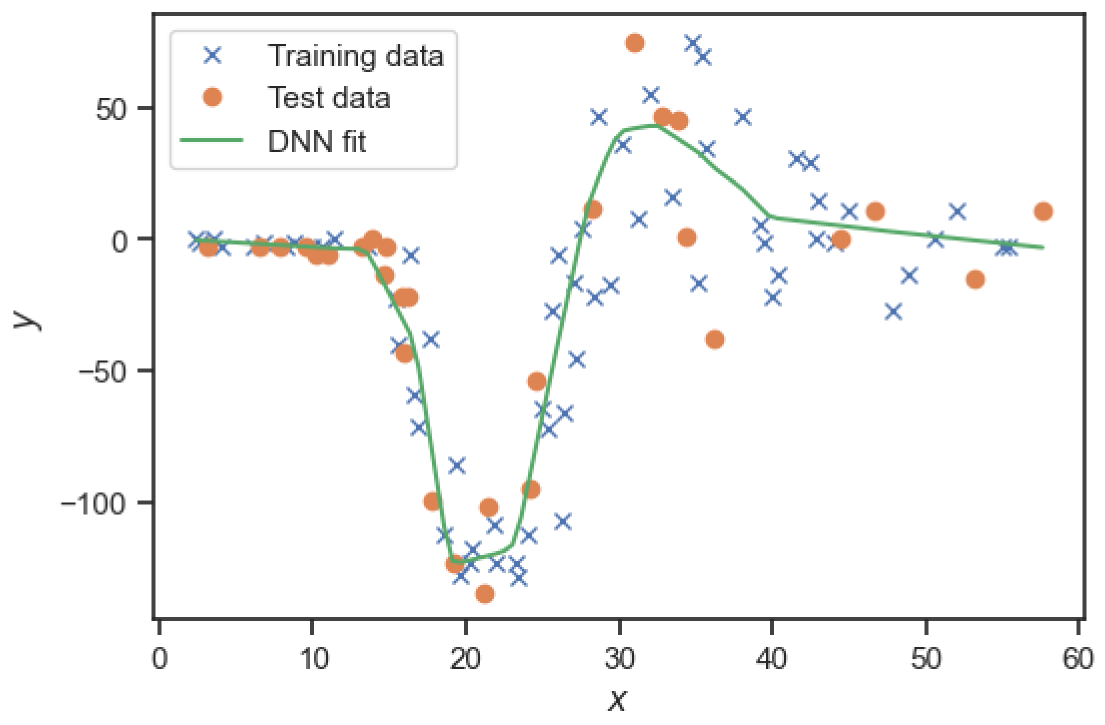
And here is predictions-observations plot on the test data set:

```
y_pred_test = f(x_s_test).detach().numpy()
fig, ax = plt.subplots(dpi=100)
ax.plot(y_pred_test, y_s_test, '.')
yys = np.linspace(y_s_test.min(), y_s_test.max(), 10)
ax.plot(yys, yys, 'r')
ax.set_xlabel('Predictions')
ax.set_ylabel('Observations');
```



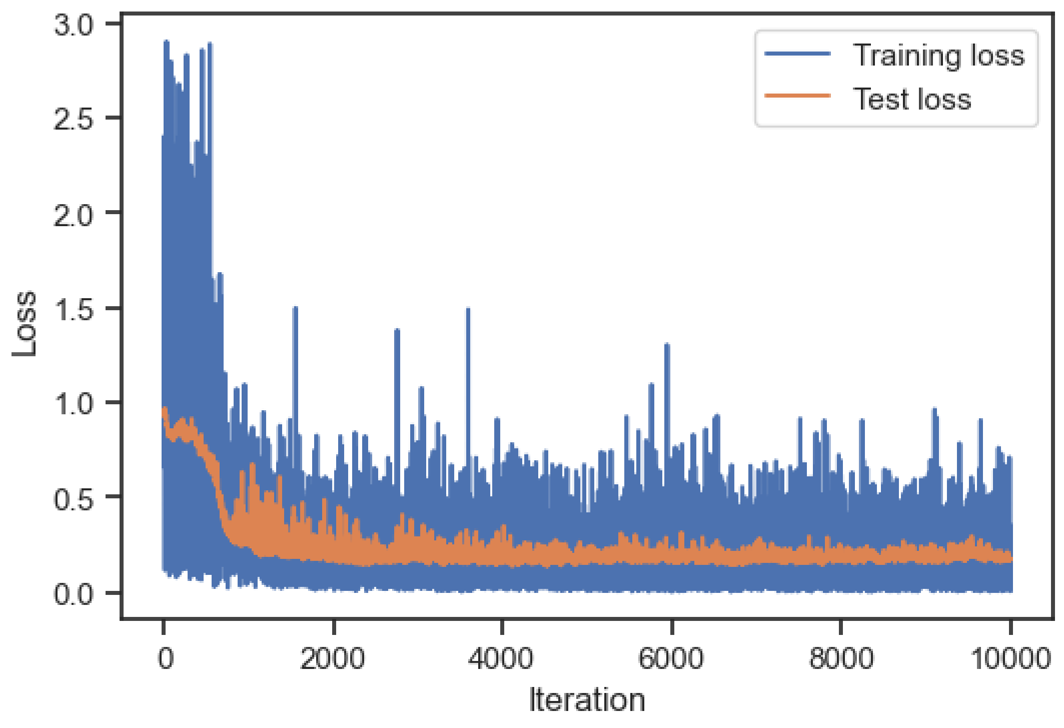
Also, if you wish, you can scale the predictions back to the original units:

```
xx = feature_scaler.inverse_transform(xx_scaled)
yy = target_scaler.inverse_transform(yy_scaled)
fig, ax = plt.subplots(dpi=100)
ax.plot(x_train, y_train, 'x', label='Training data')
ax.plot(x_test, y_test, 'o', label='Test data')
ax.plot(xx, yy, label='DNN fit')
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
plt.legend(loc='best');
```



It is instructive to observe how the training and test losses evolve as a function of the optimization iteration:

```
fig, ax = plt.subplots(dpi=100)
ax.plot(training_loss, label='Training loss')
ax.plot(test_loss, label='Test loss')
ax.set_xlabel('Iteration')
ax.set_ylabel('Loss')
plt.legend(loc='best');
```



The wiggliness is, of course, due to the stochastic nature of the optimization. The training error converges to a minimum as you keep iterating. This is direct consequence of the Robbins-Monro theorem. You will reach a local minimum of the training error eventually. However, this is not the case for the test error. The test error will reach a minimum at some point and then it will start going up! It will always do this when you are training networks by just minimizing a loss function. What happens is that the algorithm, by paying more attention to minimizing the training loss it will start, eventually, overfitting the training data and it will not be able to generalize correctly for the test data. There are ways around this. We are going to learn about the basic one in the next lecture (*weight regularization* and *early stopping*). There are some advanced ways to avoid overfitting (e.g., *dropout*, *Bayesian neural networks*) which we are not going to cover in the class.

Questions

- Change the activation function from `nn.ReLU` to `nn.Tanh`. Are you getting a better fit or a worse fit?
- Rerun the code above for 100,000 iterations. Does it start to overfit the training data? What happens to the test loss?
- Rerun the code for 5,000 iterations. How does the prediction look like now? Early-stopping would stop at about this point.

By Ilias Bilionis (ibilion[at]purdue.edu)