Data-driven Design and Analyses of Structures and Materials (3dasm)

Lecture 10

Miguel A. Bessa | **M.A.Bessa@tudelft.nl** | Associate Professor

OPTION 1. Run this notebook **locally in your computer**:

1. Confirm that you have the 3dasm conda environment (see Lecture 1).
2. Go to the 3dasm_course folder in your computer and pull the last updates of the **repository**:

```
git pull
```

3. Open command window and load jupyter notebook (it will open in your internet browser):

```
conda activate 3dasm
jupyter notebook
```

4. Open notebook of this Lecture.

OPTION 2. Use **Google's Colab** (no installation required, but times out if idle):

1. go to **https://colab.research.google.com**
2. login
3. File > Open notebook
4. click on Github (no need to login or authorize anything)
5. paste the git link: **https://github.com/bessagroup/3dasm_course**
6. click search and then click on the notebook for this Lecture.

Outline for today

- Introduction tutorial on Gaussian Processes (no theory today!)
    - Using Scikit-learn for Gaussian Process Regression (noiseless and noisy datasets)
- Introduction tutorial on Artificial Neural Networks (no theory today!)
    - Using Keras for regression with Artificial Neural Networks (noiseless and noisy datasets)

Reading material: This notebook + (GPs in Section 17.3 of book; ANNs in Chapter 13)

Today's lecture is going to be more practical

Similarly to the previous lecture, we will focus on the practical aspects of two new ML models **without deriving them yet**.

The goal is for you to be able to use these models as *black boxes* and for you to explore them in the Midterm Project assignment (reverse engineering them!).

- So, today we will focus on how to train:
    - **Gaussin Processes** using **scikit-learn**
    - **Artificial Neural Networks** (ANNs) using **keras** and **tensorflow**

This way, you can explore these models in the Midterm Project, while in later lectures we will derive these and other models that we use in this course.

1. Short tutorial on 1D regression with Gaussian Processes

Recall the model for linear regression with polynomial basis functions (Lecture 9)

1. Observation distribution:

$$p(y|x, \mathbf{z}) = \mathcal{N}(y|\mu_{y|z} = f(x; \mathbf{z}), \sigma^2_{y|z} = \sigma^2)$$

where $f(x; \mathbf{z}) = \mathbf{w}^T \boldsymbol{\phi}(x)$ and each term is defined by:

- $\mathbf{z} = (\mathbf{w}, \sigma)$ are all the hidden rv's of the model, i.e. the model parameters.
    - the vector $\mathbf{w} = [w_0, w_1, w_2 \ldots, w_d]^T$ includes the **bias** term w_0 and the remaining **weights** w_i with $i = 1, \ldots, d$.
    - the vector $\boldsymbol{\phi}(x) = [1, x, x^2, \ldots, x^d]^T$ includes the **basis functions**, which now correspond to a polynomial of degree d.

1. A chosen Prior distribution for each hidden rv of $\mathbf{z}$.

As we will see, Gaussian Processes generalize this model by introducing nonlinear functions and by correlating the entire dataset in a very interesting way.

For now, we will just share how GPs make regression predictions for one-dimensional functions (input x and output y). We will follow Görtler et al. (2019) but with a different notation: **https://distill.pub/2019/visual-exploration-gaussian-processes**

- In Gaussian Processes **each** OUTPUT value y_i in the training data is treated as a random variable that follows a Gaussian distribution:

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$$

and where the joint distribution of all the output values $\mathbf{y}$ is also Gaussian (a multivariate Gaussian) defined by a mean vector $\boldsymbol{\mu}$ and a covariance matrix $\boldsymbol{\Sigma}$. The mean is usually assumed to be zero.

We will do the derivations later, but for now we just want to know that the expression for the covariance matrix depends on a nonlinear function called the "covariance function" or "kernel" k that is calculated for **each** pair of INPUT points x_i and x_j:

$\Sigma_{ij} = \mathrm{Cov}(y_i, y_j) = k(x_i, x_j) + \alpha_i^2 \delta_{ij} \quad \Rightarrow \quad \mathbf{\Sigma} = \mathbf{K} + \mathbf{R}$

where the kernel $k(x_i, x_j)$ can be many different kinds of functions (with some special properties) and α_i^2 is the noise level of y_i
.

- An example of a kernel function is the RBF: $k(x_i, x_j) = \eta^2 \exp\left(-\frac{||x_i - x_j||^2}{2\lambda^2}\right)$
- Every kernel function has a set of unknown parameters (in red). The RBF has 2 parameters. You can visualize different kernel functions in this figure: **https://distill.pub/2019/visual-exploration-gaussian-processes/#MultipleKernels**

In the Gaussian Process method, we start by **choosing** a kernel function with an **initial guess** for the values of the parameters. This defines our **prior** information in conjunction with the assumption that the variables follow a multivariate Gaussian distribution:

$$\begin{bmatrix} \mathbf{y} \\ y^* \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} \mathbf{K}+\mathbf{R} & \mathbf{k}_* \\ \mathbf{k}_*^T & k(x^*, x^* \end{bmatrix}\right)$$

As we saw before in the course, Multivariate Gaussian distributions have very interesting properties and it turns out that based on the above assumptions a Gaussian Process predicts both the **mean** and the **variance** of the output point y^* by the following matrix operations:

$$\mathbb{E}[y^*] = \mathbf{k}_*^T(\mathbf{K}+\mathbf{R})^{-1}\mathbf{y}$$

$$\mathrm{Cov}[y^*] = k(x^*, x^*) - \mathbf{k}_*^T(\mathbf{K}+\mathbf{R})^{-1}\mathbf{k}_*$$

- The **key concept** is that the prediction of the mean and variance of the new point y^* depends on the values of the parameters of the kernel function (which are **UNKNOWN**).
- However, despite the fact that we don't know the parameters of the kernel function, they can be obtained by **Bayesian inference**.

As we saw previousy, Bayesian inference is possible by using Bayes rule to find the **posterior** information. This involves doing Marginalization and Conditioning. We will do this in the next lectures.

Today, we are focusing on learning how to do Gaussian Process regression for one-dimensional datasets.
Let's start with a noiseless case.

In [2]:

```python
from sklearn.model_selection import train_test_split

# Function to "learn"
def f(x):
    return x * np.sin(x)

n_data = 50 # number of points in our dataset
testset_ratio = 0.90 # ratio of test set points from the dataset
x_data = np.linspace(0, 10, n_data) # uniformly spaced points
y_data = f(x_data) # function values at x_data

X_data = np.reshape(x_data,(-1,1)) # a 2D array that scikit-learn likes

seed = 1987 # set a random seed so that everyone gets the same result
np.random.seed(seed)

# Let's split into 10% training points and the rest for testing:
X_train, X_test, y_train, y_test = train_test_split(X_data,
                                    y_data, test_size=testset_ratio,
                                    random_state=seed)

x_train = X_train.ravel() # just for plotting later
x_test = X_test.ravel() # just for plotting later

print("Here's a print of X_train:\n", X_train)
```

```
Here's a print of X_train:
 [[6.12244898]
 [8.57142857]
 [7.14285714]
 [2.85714286]
 [8.97959184]]
```

Gaussian Process Regression (GPR) for noiseless datasets

Let's use the RBF kernel for our predictions:

$$k(x_i, x_j) = \eta^2 \exp\left(-\frac{||x_i - x_j||^2}{2\lambda^2}\right)$$

with an initial guess for the parameters as: $\eta = 1$ and $\lambda = 10$.

In [3]:

```python
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, Matern, ExpSineSquared, ConstantKernel

# Define the kernel function
kernel = ConstantKernel(1.0, (1e-3, 1e3)) * RBF(10, (1e-2, 1e2)) # This is the standard RBF kernel
#kernel = 1.0 * RBF(10, (1e-2, 1e2)) # Same kernel as above (scikit-learn assumes constant variance if you just
                                     # write RBF without the constant kernel or without multiplying by 1.0)

# Other examples of kernels:
#kernel = ExpSineSquared(length_scale=3.0, periodicity=3.14,
#                        length_scale_bounds=(0.1, 10.0),
#                        periodicity_bounds=(0.1, 10)) * RBF(3.0, (1e-2, 1e2))
#kernel = Matern(length_scale=1.0, length_scale_bounds=(1e-2, 1e2),nu=1.5)

gp_model = GaussianProcessRegressor(kernel=kernel, alpha=1e-10, n_restarts_optimizer=20) # using a small alpha

# Fit to data to determine parameters
gp_model.fit(X_train, y_train)

# Make the prediction on the entire dataset (for plotting)
y_data_pred, sigma_data = gp_model.predict(X_data, return_std=True) # also output the uncertainty (std)

# Predict for test set (for error metric)
y_pred, sigma = gp_model.predict(X_test, return_std=True) # also output the uncertainty (std)
```

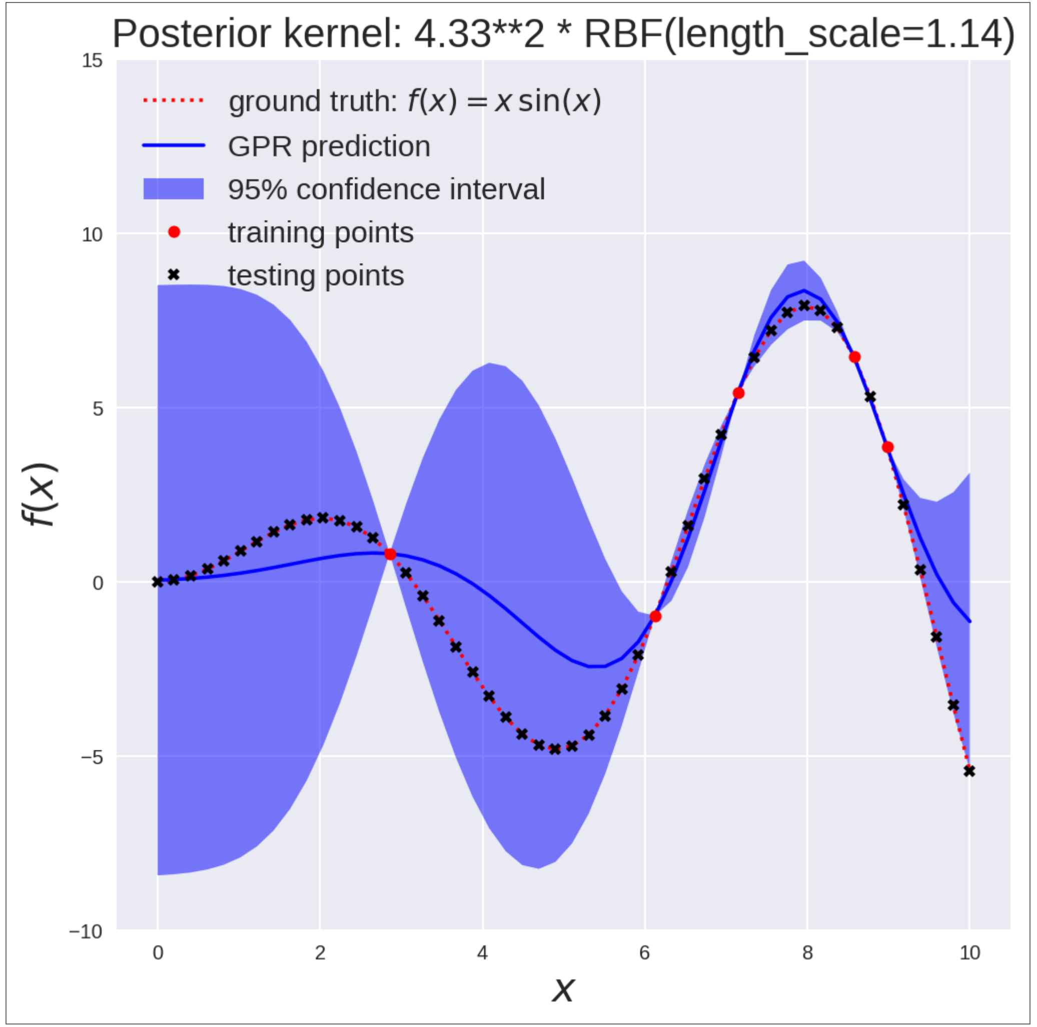
In [4]:

```
             # Plot the function, the prediction and the 95% confidence interval
fig1, ax1 = plt.subplots()

ax1.plot(x_data, y_data, 'r:', label=u'ground truth: $f(x) = x\,\sin(x)$') # function to learn

ax1.plot(x_data, y_data_pred, 'b-', label="GPR prediction")
ax1.fill(np.concatenate([x_data, x_data[::-1]]),
         np.concatenate([y_data_pred - 1.9600 * sigma_data,
                        (y_data_pred + 1.9600 * sigma_data)[::-1]]),
         alpha=.5, fc='b', ec='None', label='95% confidence interval')

ax1.plot(x_train, y_train, 'ro', markersize=6, label="training points") # noiseless data
ax1.plot(x_test, y_test, 'kX', markersize=6, label="testing points") # Plot test points

ax1.set_xlabel('$x$', fontsize=20)
ax1.set_ylabel('$f(x)$', fontsize=20)
ax1.set_title("Posterior kernel: %s"
              % gp_model.kernel_, fontsize=20) # Show in the title the value of the hyperparameters
ax1.set_ylim(-10, 15) # just to provide more space for the legend
ax1.legend(loc='upper left', fontsize=15)
fig1.set_size_inches(8,8)
plt.close(fig1) # close the plot to see it in next cell
```

In [5]:

```
fig1 # plot figure.
```

Out[5]:



Exercise 1

Fit a polynomial of degree 4 (like we did last class) and compute the error metrics for that model as well as the above mentioned Gaussian process.

In [6]:

```
# Write your code for Exercise 1:

# until here.
```

Gaussian Process Regression approximates the function much better...

- However, note that the **choice of kernel** used in GPR affects the quality of the prediction.

Gaussian Process regression for noisy datasets

Let's recreate the noisy dataset from $f(x) = x \sin x$, as we did in Lecture 9:

In [7]:

```
# Now let's also create the noisy dataset:
random_std = 0.5 + 1.0 * np.random.random(y_data.shape) # np.random.random returns random number between [0.0, 1.0)
noise = np.random.normal(0, random_std) # sample vector from Gaussians with random standard deviation
y_noisy_data = y_data + noise # Perturb every y_data point with Gaussian noise

# Pair up points with their associated noise level (because of train_test_split):
Y_noisy_data = np.column_stack((y_noisy_data,noise))

# Split into 10% training points and the rest for testing:
X_train, X_test, Y_noisy_train, Y_noisy_test = train_test_split(X_data,
                                    Y_noisy_data, test_size=testset_ratio,
                                    random_state=seed) # "noisy_train" is a great name for a variable, hein?
# NOTE: since we are using the same seed and we do train_test_split on the same X_data and y_noisy_data is
#       just y_data + noise, we are splitting the dataset exactly in the same way! This is nice because we
#       want to keep the comparison as fair as possible.

# Finally, for plotting purposes, let's convert the 2D arrays into 1D arrays (vectors):
x_train = X_train.ravel()
x_test = X_test.ravel()
y_noisy_train = Y_noisy_train[:,0]
noise_train = Y_noisy_train[:,1]
y_noisy_test = Y_noisy_test[:,0]
noise_test = Y_noisy_test[:,1]

print("Note that X_train and X_test are the same data that we used for the noiseless case.")
print("Here's a print of X_train:\n", X_train)
```

```
Note that X_train and X_test are the same data that we used for the noiseless case.
Here's a print of X_train:
 [[6.12244898]
 [8.57142857]
 [7.14285714]
 [2.85714286]
 [8.97959184]]
```

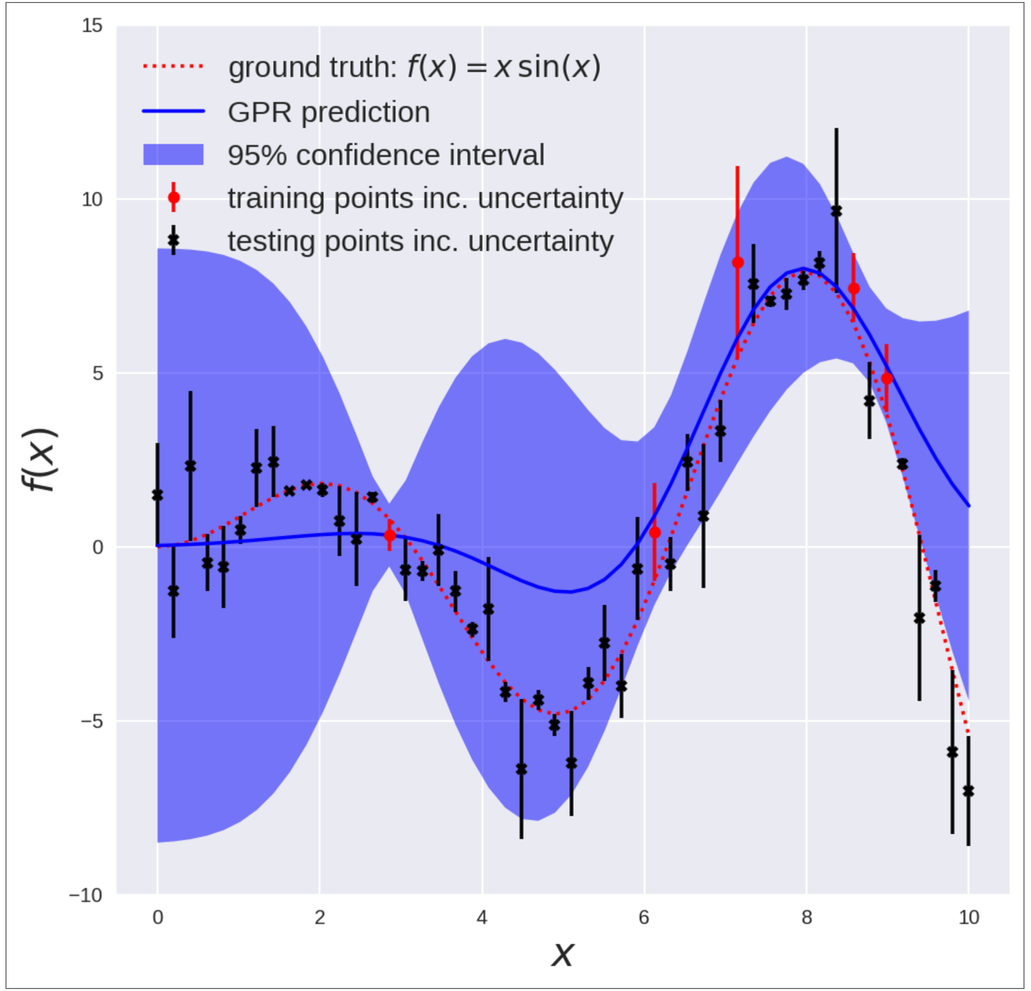
In [8]:

```python
# Instanciate a Gaussian Process model
kernel = ConstantKernel(1.0, (1e-3, 1e3)) * RBF(10, (1e-2, 1e2))

# Fitting for noisy data, if we have access to the uncertainty at the training points (usually we don't!), then
# we can include the noise level at the alpha parameter
gp_model = GaussianProcessRegressor(kernel=kernel, alpha=noise_train**2, n_restarts_optimizer=5)

# Fit to data to determine the parameters of the model
gp_model.fit(X_train, y_noisy_train)

# Make the predictions
y_noisy_pred, sigma_noisy = gp_model.predict(X_test, return_std=True) # predictions including uncertainty (std)
y_noisy_data_pred, sigma_noisy_data = gp_model.predict(X_data, return_std=True) # for plotting

# Plot the function, the prediction and the 95% confidence interval
fig1, ax1 = plt.subplots() # This opens a new figure

ax1.plot(x_data, f(x_data), 'r:', label=u'ground truth: $f(x) = x\,\sin(x)$') # function to learn
ax1.errorbar(x_train, y_noisy_train, noise_train, fmt='ro', markersize=6, label=u'training points inc. uncertainty')
ax1.errorbar(x_test, y_noisy_test, noise_test, fmt='kX', markersize=6, label=u'testing points inc. uncertainty')

ax1.plot(x_data, y_noisy_data_pred, 'b-', label="GPR prediction")
ax1.fill(np.concatenate([x_data, x_data[::-1]]),
         np.concatenate([y_noisy_data_pred - 1.9600 * sigma_noisy_data,
                        (y_noisy_data_pred + 1.9600 * sigma_noisy_data)[::-1]]),
         alpha=.5, fc='b', ec='None', label='95% confidence interval')
ax1.set_xlabel('$x$', fontsize=20)
ax1.set_ylabel('$f(x)$', fontsize=20)
ax1.set_ylim(-10, 15) # just to provide more space for the legend
ax1.legend(loc='upper left', fontsize=15)
fig1.set_size_inches(8,8)
plt.close(fig1)
```

In [9]:

```
fig1 # plot figure.
```

Out[9]:



Exercise 2

Fit a polynomial of degree 4 (like we did last class) and compute the error metrics for that model as well as the above mentioned Gaussian process.

In [10]:

```
# Exercise 2.

# until here.
```

Well done...

In the Midterm Project you will empirically explore Gaussian Processes.

2. Short tutorial on 1D regression with Artificial Neural Networks

ANNs for regression are not that different when compared to linear regression.

Consider again the observation distribution as we wrote it in the beginning of the lecture.

$$p(y|x, \mathbf{z}) = \mathcal{N}(y|\mu_{y|z} = f(x;\, \mathbf{z}), \sigma^2_{y|z} = \sigma^2)$$

But now, instead of modeling the mean of the distribution using polynomial basis functions by $f(x;\, \mathbf{z}) = \mathbf{w}^T \boldsymbol{\phi}(x)$, consider a simple extension where the **basis functions also have their own parameters**:

$$f(x;\, \mathbf{z}) = \mathbf{w}^T \boldsymbol{\phi}(x;\, \mathbf{z}_2)$$

where $\mathbf{z} = (\mathbf{z}_1,\, \mathbf{z}_2)$ and $\mathbf{z}_1 = (\mathbf{W},\, \mathbf{b})$ are all the parameters of the model.

Instead of stopping here,

$$f(x; \mathbf{z}) = \mathbf{w}^T \boldsymbol{\phi}(x; \mathbf{z}_2)$$

we can also repeat this recursively L times to create more complex transformations:

$$f(x; \mathbf{z}) = f_L\left(f_{L-1}\left(\cdots\left(f_1\left(x\right)\right)\cdots\right)\right)$$

where $f_l(x) = f(x; \mathbf{z}_l)$ is the function at layer l of our recursive transformation.
This is the key idea behind **deep neural networks**.

- Confused?

Let's try to see this for a very simple case. Consider again the observation distribution:

$$p(y|x, \mathbf{z}) = \mathcal{N}(y|\mu_{y|z} = f(x; \mathbf{z}), \sigma^2_{y|z} = \sigma^2)$$

The simplest extension comes from using a nonlinear function f_1 acting on our well-known linear model:

$$\mu_{y|z} = f(x; \mathbf{z}) = f_1\left(\mathbf{w}^T \boldsymbol{\phi}(x)\right) = f_1\left(f_0\left(x; \mathbf{z}_0\right)\right) = f_1\left(b_0 + w_0 x\right)$$

where we are calling our linear regression model as $f_0\left(x; \mathbf{z}_0\right)$.

So, our model for the mean of the observation distribution becomes:

$$\mu_{y|z} = f_1\left(b_0 + w_0 x\right)$$

But why stop here?

Why not introducing more parameters?

$$\mu_{y|z} = b_1 + w_1 f_1 \left(b_0 + w_0 x \right)$$

But why stop here?

What if we apply another nonlinear function f_2?

$$\mu_{y|z} = f_2 \left(b_1 + w_1 f_1 \left(b_0 + w_0 x \right) \right)$$

What about doing this recursively L times to create more complex transformations?

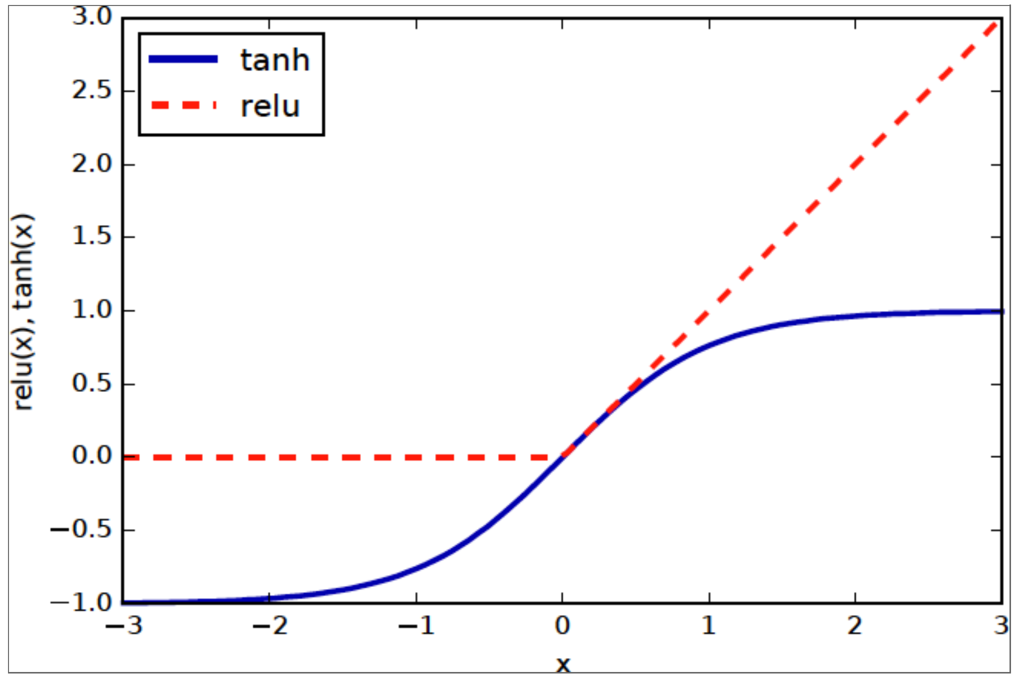
$$f(x; \mathbf{z}) = f_L \left(f_{L-1} \left(\cdots \left(f_1 \left(b_0 + w_0 x \right) \right) \cdots \right) \right)$$

This is one of the simplest examples of an Artificial Neural Network architecture called feedforward architecture.

We will look into this model carefully in a few lectures.
For now, I just want to draw a schematic so that you understand the number of parameters that starts appearing!

- Draw on the board a feedforward ANN with 2 hidden layers for 1D case.
    - First hidden layer with 3 neurons and second hidden layer with 2 neurons.

Two examples of these nonlinear functions.



Let's now focus on how to create an ANN model for 1D regression using:

1. A feedforward architecture with 2 dense hidden layers
2. The ReLu activation function
3. Adam optimizer

In [11]:

```
               # Let's create a function defining our Artificial Neural Network.
from tensorflow import keras # fast library for ANNs
from tensorflow.keras.optimizers import Adam # import the optimizer you want to use to calculate the parameters
from keras.models import Sequential # to create a feedforward neural network
from keras.layers.core import Dense # to create a feedforward neural network with dense layers
#
# Function to create the ANN model (in this case we are creating )
def create_ANN(input_dimensions=1, # number of input variables
               neurons1=3, # number of neurons in first hidden layer
               neurons2=2, # number of neurons in second hidden layer
               activation='relu', # activation function
               optimizer='adam'): # optimization algorithm to compute the weights and biases
    # create model
    model = Sequential() # Feedforward architecture
    model.add(Dense(neurons1, input_dim=input_dimensions, activation=activation)) # first hidden layer
    model.add(Dense(neurons2, activation=activation)) # second hidden layer
    model.add(Dense(1)) # output layer with just one neuron because we have only one output (1D problem!)
    model.compile(loss='mse', # error metric to measure our NLL (loss)
                  optimizer=optimizer)
    return model
```

In addition, let's introduce something important: dataset preprocessing.
Standardizing our dataset is good practice and can be important for many ML algorithms (ANNs included).

In [12]:
```
               # Standardizing your dataset is good practice and can be important for ANNs!
from sklearn.preprocessing import StandardScaler # standardize the dataset with scikit-learn
#
scaler = StandardScaler().fit(X_train) # Check scikit-learn to see what this does!
#
X_train_scaled=scaler.transform(X_train)
X_test_scaled=scaler.transform(X_test)
X_data_scaled=scaler.transform(X_data)
#
```

In [13]:

```
                from keras.wrappers.scikit_learn import KerasRegressor # a new version will use scikeras
# Now create your first ANN model!
neurons1=200 # number of neurons for the first hidden layer
neurons2=10 # number of neurons for the second hidden layer
batch_size = len(X_train) # considering the entire dataset for updating the weights and biases in each epoch
optimizer = Adam(learning_rate=0.001) # specifying the learning rate value for the optimizer (PLAY WITH THIS!)
ANN_model = KerasRegressor(build_fn=create_ANN, neurons1=neurons1, neurons2=neurons2,
                           batch_size=batch_size, epochs=150, optimizer=optimizer,
                           validation_data=(scaler.transform(X_test), y_test))
```

```
/tmp/ipykernel_12147/1298898400.py:7: DeprecationWarning: KerasRegressor is deprecated, use Sci-Keras
(https://github.com/adriangb/scikeras) instead.
  ANN_model = KerasRegressor(build_fn=create_ANN, neurons1=neurons1, neurons2=neurons2,
```

In [14]:

```python
# Now that we created our first ANN model, let's fit it to our (scaled) dataset!
history = ANN_model.fit(X_train_scaled, y_train)
```

```
2022-03-19 19:40:45.012623: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:939] successful NU
MA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returni
ng NUMA node zero
2022-03-19 19:40:45.122952: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:939] successful NU
MA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returni
ng NUMA node zero
2022-03-19 19:40:45.123476: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:939] successful NU
MA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returni
ng NUMA node zero
2022-03-19 19:40:45.123895: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1900] Ignoring visible
gpu device (device: 0, name: GeForce GTX 560M, pci bus id: 0000:01:00.0, compute capability: 2.1) wit
h Cuda compute capability 2.1. The minimum required Cuda capability is 3.5.
2022-03-19 19:40:45.125169: I tensorflow/core/platform/cpu_feature_guard.cc:151] This TensorFlow bina
ry is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instruction
s in performance-critical operations:  SSE4.1 SSE4.2 AVX
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
```

```
Epoch 1/150
1/1 [==============================] - 1s 523ms/step - loss: 17.6255 - val_loss: 13.9907
Epoch 2/150
1/1 [==============================] - 0s 28ms/step - loss: 17.5265 - val_loss: 13.9696
Epoch 3/150
1/1 [==============================] - 0s 29ms/step - loss: 17.4302 - val_loss: 13.9507
Epoch 4/150
1/1 [==============================] - 0s 27ms/step - loss: 17.3398 - val_loss: 13.9466
```

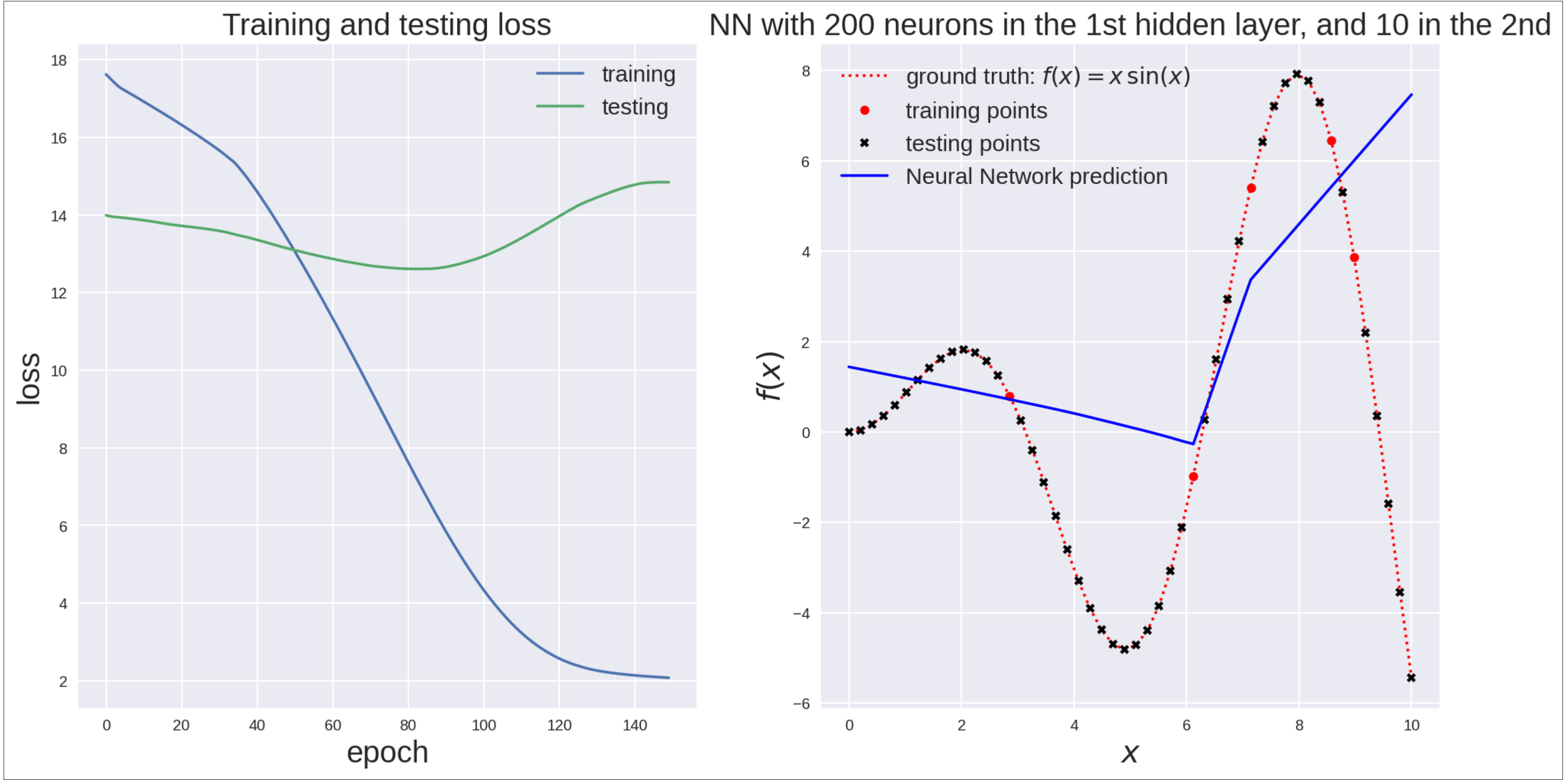
```
Epoch 145/150
1/1 [==============================] - 0s 27ms/step - loss: 2.1057 - val_loss: 14.8369
Epoch 146/150
1/1 [==============================] - 0s 30ms/step - loss: 2.0995 - val_loss: 14.8424
Epoch 147/150
1/1 [==============================] - 0s 30ms/step - loss: 2.0932 - val_loss: 14.8460
Epoch 148/150
1/1 [==============================] - 0s 34ms/step - loss: 2.0868 - val_loss: 14.8474
Epoch 149/150
1/1 [==============================] - 0s 35ms/step - loss: 2.0806 - val_loss: 14.8467
Epoch 150/150
1/1 [==============================] - 0s 33ms/step - loss: 2.0745 - val_loss: 14.8447
```

In [15]:

```
fig_ANN, (ax1_ANN, ax2_ANN) = plt.subplots(1,2)
# Create a plot for the loss history
ax1_ANN.plot(history.history['loss']) # plot training loss
ax1_ANN.plot(history.history['val_loss']) # plot testing loss
ax1_ANN.set_title('Training and testing loss', fontsize=20)
ax1_ANN.set_ylabel('loss', fontsize=20)
ax1_ANN.set_xlabel('epoch', fontsize=20)
ax1_ANN.legend(['training', 'testing'], loc='upper right', fontsize=15)

# Create a plot for the ANN prediction
ax2_ANN.plot(x_data, f(x_data), 'r:', label=u'ground truth: $f(x) = x\,\sin(x)$') # show ground truth function
ax2_ANN.plot(x_train, y_train, 'ro', markersize=6, label="training points") # show training data
ax2_ANN.plot(x_test, y_test, 'kX', markersize=6, label="testing points") # show testing data

y_pred = history.model.predict(X_data_scaled) # predict all data points with ANN

ax2_ANN.plot(x_data, y_pred, 'b-', label="Neural Network prediction") # plot prediction
ax2_ANN.set_title(r'NN with '+str(neurons1)+' neurons in the 1st hidden layer, and '+str(neurons2)+' in the 2nd',
                 fontsize=20)
ax2_ANN.set_xlabel('$x$', fontsize=20)
ax2_ANN.set_ylabel('$f(x)$', fontsize=20)
ax2_ANN.legend(loc='upper left', fontsize=15)

# Create figure with specified size
fig_ANN.set_size_inches(16, 8)
plt.close(fig_ANN) # do not plot the figure now. We will show it in the next cell
```

```
In [16]:    fig_ANN # show figure now.
```

```
Out[16]:
```



Not the most amazing model you have ever seen, right?

- Try again but now using 200 neurons for the first hidden layer and 10 for the second hidden layer.
    - spoiler alert: a bit better, but far from amazing...

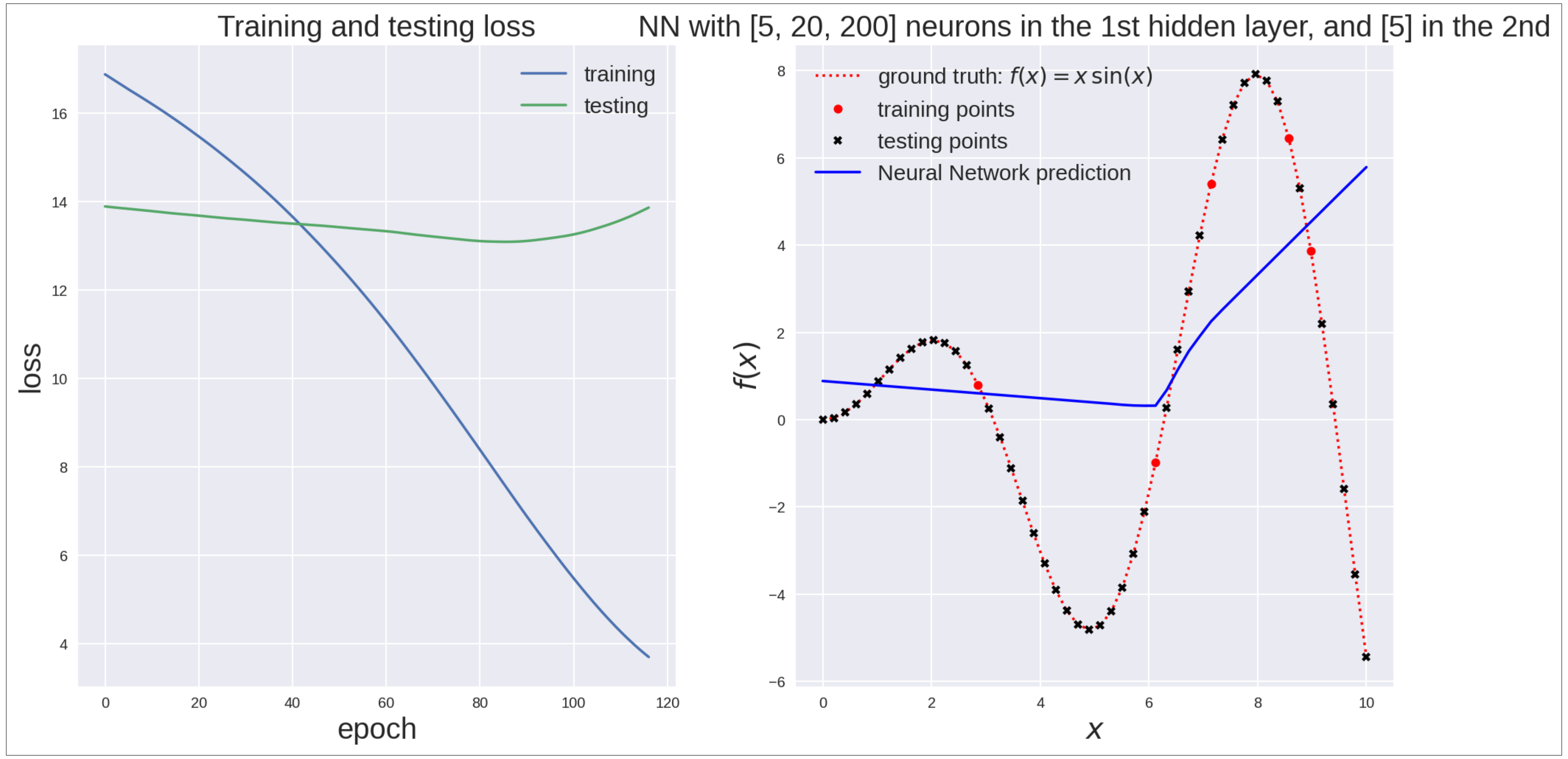
It's possible to try to find better hyperparameters (and ANNs have many!).
The notes below (not shown in presentation) contain a simple code to do this by "brute force" in a procedure called grid search.
There are some Bonus questions in the Midterm Project where you can use this strategy to find better hyperparameters...

In [18]:
```
fig_ANN, (ax1_ANN, ax2_ANN) = plt.subplots(1,2)
# Create a plot for the loss history
ax1_ANN.plot(history.history['loss']) # plot training loss
ax1_ANN.plot(history.history['val_loss']) # plot testing loss
ax1_ANN.set_title('Training and testing loss', fontsize=20)
ax1_ANN.set_ylabel('loss', fontsize=20)
ax1_ANN.set_xlabel('epoch', fontsize=20)
ax1_ANN.legend(['training', 'testing'], loc='upper right', fontsize=15)

# Create a plot for the ANN prediction
ax2_ANN.plot(x_data, f(x_data), 'r:', label=u'ground truth: $f(x) = x\,\sin(x)$') # show ground truth function
ax2_ANN.plot(x_train, y_train, 'ro', markersize=6, label="training points") # show training data
ax2_ANN.plot(x_test, y_test, 'kX', markersize=6, label="testing points") # show testing data

y_pred = history.model.predict(X_data_scaled) # predict all data points with ANN

ax2_ANN.plot(x_data, y_pred, 'b-', label="Neural Network prediction") # plot prediction
ax2_ANN.set_title(r'NN with '+str(neurons1)+' neurons in the 1st hidden layer, and '+str(neurons2)+' in the 2nd',
                 fontsize=20)
ax2_ANN.set_xlabel('$x$', fontsize=20)
ax2_ANN.set_ylabel('$f(x)$', fontsize=20)
ax2_ANN.legend(loc='upper left', fontsize=15)

# Create figure with specified size
fig_ANN.set_size_inches(16, 8)
plt.close(fig_ANN) # do not plot the figure now. We will show it in the next cell
```

In [19]:

```
fig_ANN # show figure now.
```

Out[19]:



This neural network approximation is also not brilliant... Were you expecting this?

HOMEWORK

Redo the neural network regression but now for the noisy dataset (use the same network we used for the noiseless dataset).

You will explore these and other things in Part 1 of the Midterm Project...

Have fun!