Seminars on Overparametrized Machine Learning: Hand-in assignment 3

Antônio H. Ribeiro, Dave Zachariah, Per Mattsson

Due: 2nd of December 2021, 23:59

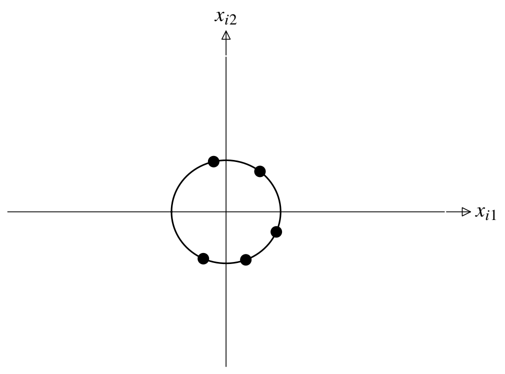
The items in the assignment can be implemented in the programming language of choice. Nonetheless, we recommend the usage of Python as a programming language, since we might include suggestions of functions and code snippets in the exercise description.

The setup of this problem resembles that from Figure 2 in (Jacot et al., 2018). We try to recover the function $f(x_i) = x_{i1}x_{i2}$ where $x_i = (x_{i1}, x_{i2})$ is a given input.

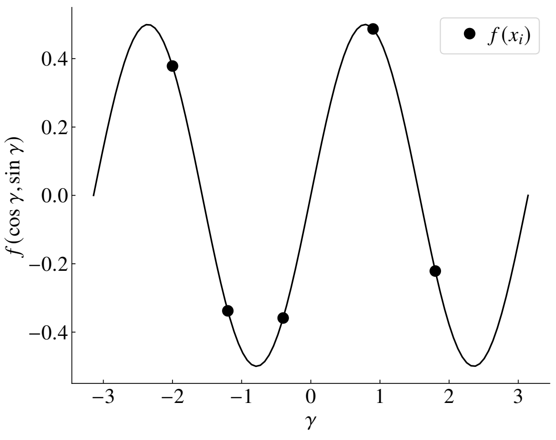
Dataset: We consider a dataset consisting only of $n_o = 5$ points (x_i, y_i), $i = 1, 2, 3, 4, 5$. The inputs are two dimensional $x_i \in \mathbb{R}^2$ and the output are computed using the function f is defined as $y_i = f(x_i) = x_{i1}x_{i2}$. All the inputs used for training lie on the unitary circle, and have the format

$$x_i = (\cos(\gamma_i), \sin(\gamma_i)). \tag{1}$$

Where $[\gamma_1, \gamma_2, \gamma_3, \gamma_4, \gamma_5] = [-2, -1.2, -0.4, 0.5, 1.5]$. In Figure 1, we illustrate where the points x_i lie in the plane $\mathbb{R}^2$ and in the one dimensional plot, as a function of the angle γ_i.



(a) Inputs, x



(b) Function f

Figure 1: In a), we show the inputs two dimentional inputs x_i, $i = 1, \cdots, 5$ together with the unitary circle. In b). we show the function $f(x)$ as we vary the angle γ, the points $(x_i, f(x_i))$ that will be used for training are indicated by disks.

Notation: We consider the following notation

- **Obseved points**: We consider $n_o = 5$ points were observed and will be used to train the model. We denote the stacked X_o the matrix containing in its rows the observed inputs x_i, $i = 1, \cdots, 5$ and y_o the vector containing the corresponding outputs. In general, the sub-index $_o$ will be used to denote quantities obtained from these points.
- **Evaluation points**: We consider $n_e = 100$ evaluation points, obtained for γ uniformly spaced in the interval $[-\pi, \pi]$. These points are close together and will be used to generate plots that look like they are a continuous evaluation of the fuction (i.e., they were used to generate the full line in Figure 1(b)). Here, we denote by X_e the matrix containing in its rows the observed evaluation inputs and by y_e the vector containing the corresponding outputs. In general, the sub-index $_e$ will be used to denote quantities obtained from these points.

Model: As a model, we will consider a neural network with $L = 4$ hidden layers. The neural network will be a function $f_\theta : \mathbb{R}^2 \to \mathbb{R}$. Let $\alpha^{(\ell)} \in \mathbb{R}^{n_\ell}$ denote the activations of the neurons at ℓ-th layer. Defining, $\alpha^{(0)} = x$, and, recursively,

$$\alpha^{(\ell+1)}(x) = \sigma\left(\frac{1}{\sqrt{n_\ell}} W^{(\ell)} \alpha^{(\ell)}(x) + \beta b^{(\ell)}\right), \text{ for } \ell = 0, 1, \cdots, L-1, \tag{2}$$

the neural network output is an affine transformation of the activations in the last layer $f_\theta(x) = W\alpha^{(L-1)}(x) + b$, where we use $W = W^{(L)}$ and $b = b^{(L)}$ to denote the weights and bias in the last layer. The weights $W^{(\ell)}$ and bias of $b_i^{(\ell)}$ have dimension $n_{\ell+1} \times n_\ell$ and $n_{\ell+1}$, respectively. At initialization, we consider it entries are are $W_{ij}^{(\ell)} \sim \mathcal{N}(0,1)$ and $b_i^{(\ell)} \sim \mathcal{N}(0,1)$. As in the original, paper $\beta = 0.1$ and the nonlinearity $\sigma(\cdot)$ is the ReLU activation function, i.e. $\sigma(z) = \max(z, 0)$ applied elementwise.

For simplicity, in this exercise, we consider all hidden layers have the width 1000. Hence, $n_0 = 2$ and $n_\ell = 1000$ for $\ell = 1, \cdots, L-1$.

Part 1: Neural Network as a Gaussian Process

Neural networks at the initialization tend to Gaussian Processes (GP) in the infinite width limit. This is stated in Proposition 1 from (Jacot et al., 2018) and will be numerically verified here.

Neural network with untrained hidden layers: Here we consider a neural network where only the last layer is trained. The weights and bias are fixed at initialization values $W_{ij}^{(\ell)} \sim \mathcal{N}(0,1)$ and $b_i^{(\ell)} \sim \mathcal{N}(0,1)$. And the only parameter that is trained is the parameter vector at the last layer $\theta = \begin{bmatrix} W & b \end{bmatrix}^\mathsf{T}$. Let by $A_o \in \mathbb{R}^{5x(1000+1)}$ the matrix containing in its rows the activations in the last layer plus a last colum containin ones. It does it for the 5 observed points,

$$A_o = \begin{bmatrix} \alpha^{(L-1)}(x_1) & 1 \\ & \vdots & \\ \alpha^{(L-1)}(x_5) & 1 \end{bmatrix}, \tag{3}$$

we can estimate the parameter θ by a simple linear regression problem, similar to the previous assignments:

$$\theta = (A_o^\mathsf{T} A_o)^+ A_o^\mathsf{T} y_o, \tag{4}$$

Which, in the overparametrized regime, corresponds to finding the minimum ℓ_2-norm solution to the problem that minimizes the sum of square error predictions (as used in the previous assignments).

Note: scipy.linalg.lstsq does yield the desired behaviour.

Gaussian Process (GP) Given a kernel $k : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$ that maps a pair of inputs x_i and x_j to a real value, i.e. $k(x_i, x_j) \mapsto K_{i,j}$. A Gaussian Process is defined by saying that, when applied to a inputs $\{x_i\}_{i=1}^n$, the output denoted here by $h(x)$ is jointly normal, that is

$$\begin{bmatrix} h(x_1) \\ h(x_2) \\ h(x_3) \\ \vdots \\ h(x_n) \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \begin{bmatrix} k(x_1,x_1) & k(x_1,x_2) & \cdots & k(x_1,x_n) \\ k(x_2,x_1) & k(x_2,x_2) & \cdots & k(x_2,x_n) \\ k(x_3,x_1) & k(x_3,x_2) & \cdots & k(x_3,x_n) \\ \vdots & \vdots & \ddots & \vdots \\ k(x_n,x_1) & k(x_n,x_2) & \cdots & k(x_n,x_n) \end{bmatrix},\right) \tag{5}$$

From this hypotesis we can make prediction for unseen points from the one that have been observed. In our example, $n_o = 5$ points have been observed and assume that we want to evaluate the function on n_e points. Assume that $n = n_o + n_e$ and denote $h_\text{o} \in \mathbb{R}^{n_o}$ the observed points, $h_\text{e} \in \mathbb{R}^{n_e}$ the points where the function will be evaluated. Assume the observed points are points correspond to the first predictions and the evaluation points to the last ones then we can write as block matrices:

$$\begin{bmatrix} h_\text{o} \\ h_\text{e} \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} K_{oo} & K_{oe} \\ K_{oe}^\mathsf{T} & K_{ee} \end{bmatrix},\right) \tag{6}$$

Here K_{oo}, K_{oe} and K_{ee} have dimension $n_o \times n_o$, $n_o \times n_e$ and $n_e \times n_e$ respectively. Using basic properties from the normal distribution, see (Lindholm et al., 2022, Section 9.A, Theorem 9.3), h_e conditioned on $h_\text{o} = y_o$ is normally distributed

$$h_\text{e} \mid (h_\text{o} = y_o) \sim \mathcal{N}\left(K_{oe}^\mathsf{T} K_{oo}^{-1} y, K_{ee} - K_{oe}^\mathsf{T} K_{oo}^{-1} K_{oe}\right) \tag{7}$$

Note: We provide the code for implementation of a Gaussian process in Appendix A.

Neural Network GP Here we consider the kernel from Proposition 1 from (Jacot et al., 2018). Where $\Sigma(x_i, x_j) = \Sigma^{(L)}(x_i, x_j)$ for $\Sigma^{(\ell)}(\bullet, \bullet)$ defined recursively as:

$$\begin{aligned} \Sigma^{(1)}\left(x_i, x_j\right) &= \frac{1}{n_0} x^T x' + \beta^2 \\ \Sigma^{(\ell+1)}\left(x_i, x_j\right) &= \mathbb{E}_{f \sim \mathcal{N}\left(0, \Sigma^{(\ell)}\right)}\left[\sigma(f(x)) \sigma\left(f\left(x'\right)\right)\right] + \beta^2 \end{aligned} \tag{8}$$

We ask you to implement this kernel above. In the case of ReLU activation function there is an analytical solution for the above, see for instance (Lee et al., 2020, Appendix C) or Cho and Saul (2009). Nonetheless, in this exercise we ask you to approximate the expectation using the average of sampled values. The approximation can be implemented using the following steps:

1. The first layer can be computed exactly, hence just use $\hat{\Sigma}^{(1)}(\bullet, \bullet) = \Sigma^{(1)}(\bullet, \bullet)$.
2. Now, for $\ell = 1, 2, \cdots, L-1$ to obtain the kernel.
   (a) Given the kernel $\hat{\Sigma}^{(\ell)}(\bullet, \bullet)$ approximate and a set of points $\{x_i\}_{i=1}^n$ build the covariance matrix:

$$\hat{\Sigma}^{(\ell)} = \begin{bmatrix} \hat{\Sigma}^{(\ell)}(x_1, x_1) & \hat{\Sigma}^{(\ell)}(x_1, x_2) & \cdots & \hat{\Sigma}^{(\ell)}(x_1, x_n) \\ \hat{\Sigma}^{(\ell)}(x_2, x_1) & \hat{\Sigma}^{(\ell)}(x_2, x_2) & \cdots & \hat{\Sigma}^{(\ell)}(x_2, x_n) \\ \hat{\Sigma}^{(\ell)}(x_3, x_1) & \Sigma^{(\ell)}(x_3, x_2) & \cdots & \hat{\Sigma}^{(\ell)}(x_3, x_n) \\ \vdots & \vdots & \ddots & \vdots \\ \hat{\Sigma}^{(\ell)}(x_n, x_1) & \Sigma^{(\ell)}(x_n, x_2) & \cdots & \hat{\Sigma}^{(\ell)}(x_n, x_n) \end{bmatrix} \tag{9}$$

   (b) Sample $K = 1000$ values from the normal distribution $\mathcal{N}\left(0, \hat{\Sigma}^{(\ell)}\right)$. Let us call these values $z_k \in \mathbb{R}^n$, where $k = 1, 2, \cdots, K$. Where we will call the i-th entrie of the k-th vector $z_{ki} \in \mathbb{R}$

   Note: You can use scipy.stats.multivariate_normal to sample from this multivariate normal distribution.

   (c) Now, approximate:

$$\hat{\Sigma}^{(\ell+1)}\left(x_i, x_j\right) = \frac{1}{K} \sum_{k=1}^{K} \sigma(z_{ki}) \sigma(z_{kj}) + \beta^2 \tag{10}$$

3. Obtain the neural network kernel at the last layer: $\Sigma = \Sigma^{(L-1)}$

We can then making predictions considering a Gaussian process with this kernel. Let us denote $f^{(0)}$ the neural network at the initialization, in the infinitely wide limit,

$$f^{(0)}\left(\begin{bmatrix} X_o \\ X_e \end{bmatrix}\right) \sim N\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \Sigma_{oo} & \Sigma_{oe} \\ \Sigma_{oe}^{\mathsf{T}} & \Sigma_{ee} \end{bmatrix}\right) \tag{11}$$

Then, the neural network at the initilization at the evaluation points conditioned on the observed points being equal to the observed output is also normally distributed:

$$f^{(0)}\left(X_e\right) \mid \left(f^{(0)}(X_o) = y_o\right) \sim N\left(\Sigma_{oe}^{\mathsf{T}} \Sigma_{oo}^{-1} y, \Sigma_{ee} - \Sigma_{oe}^{\mathsf{T}} \Sigma_{oo}^{-1} \Sigma_{oe}\right)$$

Exercise

Generate a plot with:

- Five solution from different random seeds of the neural network with untrained hidden layers (trained using least squares) and width 1000 and $L = 4$ hidden layers.
- Five solutions sampled from the neural network GP above, together with the credibility intervals of the Gaussian process. (One example with GP solutions and credibility's intervals is displayed in Figure 2 together with the code).

This plot will contain a lot of information, so be care full with the color, widths, line and marker styles, so all the elements are clear.

Part 2: Neural tangent kernel

In (Jacot et al., 2018) they show that as the wide tend to infinity, performing gradient descent in a neural network approximate kernel gradient descent for a space defined by a kernel they called "Neural Tangent Kernel". Here we numerically verify this notion.

Neural network: Now we ask you to train the same neural network as before but now using backpropagation and updating all the layers with gradient descent. Let's define the mean of square errors over all training samples ($n_o = 5$):

$$V = \frac{1}{2n_o}\sum_{i=1}^{n_o}(f_\theta(x_i) - y_i)^2, \tag{12}$$

and denote by $\nabla_\theta V$ the gradient of the cost function regarding all the parameters of the neural network $\theta = \{W^{(1)}, \cdots, W^{(L)}, b^{(1)}, \cdots, b^{(L)}\}$. We update the parameter $\hat{\theta}_t$ iteratively using gradient descent:

$$\hat{\theta}_{t+1} = \hat{\theta}_t - \lambda \nabla_\theta V. \tag{13}$$

We do it for $T = 1000$ iterations (aka epochs) and using a learning rate $\lambda = 1$, as in (Jacot et al., 2018). The parameters should be intialized as in Part 1 for iteration $t = 0$.

Note: Notice that the setup here is using *gradient descent* and all samples are used to compute the gradient. This contrast with a more common scenario used when training with larger dataset where *stochastic gradient descent* is often used instead.

Note: We provide the code for implementation of a neural network in Appendix B.

Neural tangent kernel (NTK): Implement a function that computes the neural tangent kernel:

$$\begin{aligned} \Theta^{(1)}(x_i, x_j) &= \Sigma^{(1)}(x_i, x_j) \\ \Theta^{(\ell+1)}(x_i, x_j) &= \Theta^{(\ell)}(x_i, x_j)\,\dot{\Sigma}^{(\ell+1)}(x_i, x_j) + \Sigma^{(\ell+1)}(x_i, x_j) \end{aligned} \tag{14}$$

where,

$$\dot{\Sigma}^{(\ell+1)}(x_i, x_j) = \mathbb{E}_{f\sim\mathcal{N}\left(0,\Sigma^{(\ell)}\right)}\left[\dot{\sigma}(f(x_i))\dot{\sigma}\left(f\left(x_j\right)\right)\right] \tag{15}$$

and $\Sigma^{(\ell)}(\bullet,\bullet)$ is defined as in Eq. (8). In the computation of $\dot{\Sigma}^{(\ell+1)}(\bullet,\bullet)$, use the procedure described in Part 1 to approximate $\mathbb{E}_{f\sim\mathcal{N}\left(0,\Sigma^{(\ell)}\right)}[\bullet]$.

NTK in regression: Let's denote, $\Theta(x_i, x_j) = \Theta^{(L)}(x_i, x_j)$ and

$$\Theta = \begin{bmatrix} \Theta(x_1,x_1) & \Theta(x_1,x_2) & \cdots & \Theta(x_1,x_n) \\ \Theta(x_2,x_1) & \Theta(x_2,x_2) & \cdots & \Theta(x_2,x_n) \\ \Theta(x_3,x_1) & \Theta(x_3,x_2) & \cdots & \Theta(x_3,x_n) \\ \vdots & \vdots & \ddots & \vdots \\ \Theta(x_n,x_1) & \Theta(x_n,x_2) & \cdots & \Theta(x_n,x_n) \end{bmatrix}. \tag{16}$$

Dividing it into submatrices

$$\Theta = \begin{bmatrix} \Theta_{oo} & \Theta_{oe} \\ \Theta_{oe}^\mathsf{T} & \Theta_{ee} \end{bmatrix} \tag{17}$$

We have that the gradient descent solution of the gradient descent in neural tangent kernel converges, in the limit, to the function $f^{(\infty)}$. It has a closed expression, derived in (Jacot et al., 2018, Section 5),

$$f^{(\infty)}(X_e) = \Theta_{oe}^\mathsf{T}\Theta_{oo}^{-1}y_o + \left(f^{(0)}(X_e) - \Theta_{oe}^\mathsf{T}\Theta_{oo}^{-1}f^{(0)}(X_o)\right). \tag{18}$$

Here $f^{(0)}$ is the neural network at the initialization, which in the width limit, is normally distributed (See Eq. (11)). Hence the equation above is just affine transformation of a normal distribution and some standard manipulation of normal multivariate variables (see Lindholm et al. (2022, Section 9.A, Corollary 9.2)) yields that $f^{(\infty)}(x_e)$ is normally distributed,

$$f^{(\infty)}(X_e) \sim \mathcal{N}(\mu, C) \tag{19}$$

with

$$\begin{aligned} \mu &= \Theta_{oe}^\mathsf{T}\Theta_{oo}^{-1}y_o \\ C &= \Sigma_{ee} + \Theta_{oe}^\mathsf{T}\Theta_{oo}^{-1}\Sigma_{oo}\Theta_{oo}^{-1}\Theta_{oe} - \Theta_{oe}^\mathsf{T}\Theta_{oo}^{-1}\Sigma_{oe} - \Sigma_{oe}^\mathsf{T}\Theta_{oo}^{-1}\Theta_{oe}. \end{aligned}$$

Note: Notice that the solution of the neural tangent kernel is not the same solution that would be obtained by considering a Gaussian process with the kernel $\Theta(\bullet,\bullet)$.

Exercise

Generate a plot with

- Five solution from different random seeds of the neural network (trained using gradient descent, see Appendix B) and width 1000 and $L = 4$ hidden layers.
- Five solutions that could be obtained from NTK solution above. Plot also the credibility intervals.

This plot will contain a lot of information, so be care full with the color, widths, line and marker styles, so all the elements are clear.

The Submission

Your submission should have a single page of content (a4paper, fontsize=10pt, margin=2cm, both single and double column are acceptable...). Include your name, the two plots (one for Part 1 and one for Part 2), a short description of the experiment parameters that you used and a paragraph of discussion/conclusion. You can assume that whoever will read your report has both read the paper from (Jacot et al., 2018) and the entire description above, so there is no need for repeating it...

All requested plots should have proper figure captions, legends, and axis labels. You should submit two files, one pdf-file with the report and a standalone script (or jupyter notebook) that can be used to run the code and generate the plots (Write as comments the packages/libraries versions and additional requirements as comments at the top of the script). Compress the two files as a single zip (containing pdf + script) and mail it to antonio.horta.ribeiro@it.uu.se. You will receive a confirmation mail back.

References

Y. Cho and L. Saul. Kernel methods for deep learning. *Advances in Neural Information Processing Systems 22*, 2009.

A. Jacot, F. Gabriel, and C. Hongler. Neural Tangent Kernel: Convergence and Generalization in Neural Networks. *Advances in Neural Information Processing Systems 31*, 2018.

J. Lee, L. Xiao, S. S. Schoenholz, Y. Bahri, R. Novak, J. Sohl-Dickstein, and J. Pennington. Wide neural networks of any depth evolve as linear models under gradient descent. *Journal of Statistical Mechanics: Theory and Experiment*, 2020(12):124002, Dec. 2020. ISSN 1742-5468. doi: 10.1088/1742-5468/abc62b.

A. Lindholm, N. Wahlström, F. Lindsten, and T. B. Schön. *Machine Learning: A First Course for Engineers and Scientists*. Cambridge University Press, 2022. ISBN 978-1-108-84360-7. http://smlbook.org/

A Implementing a Gaussian process

The code below implements a Gaussian process for a Gaussian kernel:

$$k(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2l^2}\right) \tag{20}$$

in you implementation of Part 1, you will need to replace this kernel by the kernel $\hat{\Sigma}(x_i, x_j)$ defined there.

```python
import numpy as np

# Small value it will be added to the covariance matrix to avoid numerical
# instabilities yielding a non positive definite covariacne matrix
EPS = 1e-7


def expkernel(l=1):
    def kernel(x1, x2):
        d = np.linalg.norm(x1 - x2, 2)
        out = np.exp(- d ** 2 / (2 * (l ** 2)))
        return out
    return kernel


def cov(x, kernel):
    """Receives a matrix (n_points, dimension) and return a kernel (n_points, n_points)"""
    n_points, dim = x.shape
    C = np.zeros((n_points, n_points))
    for i in range(n_points):
        for j in range(n_points):
            C[i, j] = kernel(x[i, :], x[j, :])
    return C


def conditioning(c, y_obs):
    n_obs = len(y_obs)
    if n_obs == 0:
        return np.zeros(c.shape[0]), c
    cov_obs = c[:n_obs, :n_obs]
    cov_eval_obs = c[n_obs:, :n_obs]
    cov_eval = c[n_obs:, n_obs:]

    inv_cov = np.linalg.inv(cov_obs)
    m = cov_eval_obs @ inv_cov @ y_obs
    cc = cov_eval - cov_eval_obs @ inv_cov @ cov_eval_obs.T
    return m, cc


if __name__ == "__main__":
    from scipy.stats import multivariate_normal
    import matplotlib.pyplot as plt

    # Train data
    gamma_obs = np.array([-2, -1.2, -0.4, 0.9, 1.8])
    X_obs = np.stack([np.cos(gamma_obs), np.sin(gamma_obs)]).T
    Y_obs = X_obs.prod(axis=1)

    gamma_eval = np.linspace(-np.pi, np.pi, 100)
    X_eval = np.stack([np.cos(gamma_eval), np.sin(gamma_eval)]).T
    Y_eval = X_eval.prod(axis=1)

    # Generate plot before conditioning
    # Get covariance matrix
    c = cov(np.vstack([X_obs, X_eval]), kernel=expkernel(l=1))
    m, cc = conditioning(c, Y_obs)
    normal = multivariate_normal(mean=m.flatten(), cov=cc + EPS*np.eye(cc.shape[0]))
    Y_sols = normal.rvs(5)
    plt.plot(gamma_eval, Y_sols.T, color=str(0.4), ms=10)
    plt.plot(gamma_obs, Y_obs.T, '*', color='black', ms=10)
    plt.plot(gamma_eval, Y_eval.T, '-', color='black', ms=10)
    for nn in [3, 2, 1]:
        plt.fill_between(gamma_eval, m - nn*np.sqrt(np.diag(cc)),
                         m + nn*np.sqrt(np.diag(cc)), color=str(0.4+0.15 * nn), alpha=0.5)
    plt.xlabel('gamma')
    plt.ylabel('f')
    plt.savefig('gaussian_process.pdf')
    plt.show()
```

Listing 1: Implementing a Gaussian process (available on github.com/uu-sml/seminars-overparam-ml)

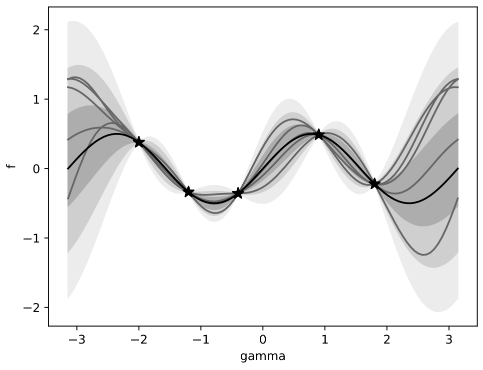


Figure 2: Output of the code. In the x-axis we have the angle γ, and on the y-axis the output of the function to an input $(\cos(\gamma_i), \sin(\gamma_i))$. The markers indicate the points that were observed and the black line the function f, used to generate them. The grey lines indicate possible solutions from the Gaussian Process. The shaded regions indicate the 68%, 95%, and 99.7% percentiles credibility intervals (i.e., 1, 2 and 3 standard deviations)

B Implementing a neural network

The code bellow implements the neural network described in Section 2 in PyTorch, together with the gradient descent training.

Note: This code can be adapted and used in Part 2. The training method for Part 1 is different and does not rely on gradient descent.

```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import tqdm


class FeedforwardNetwork(nn.Module):
    """Neural network implemented using NumPy."""

    def __init__(self, input_size, width=1000, beta=0.1, depth=4):
        super(FeedforwardNetwork, self).__init__()
        self.width = width
        self.beta = beta
        self.depth = depth
        self.ws = [input_size] + [self.width for _l in range(self.depth)] + [1]

        self.weights = []
        self.biases = []
        for i in range(depth + 1):
            wi = nn.Parameter(torch.randn(self.ws[i], self.ws[i + 1]))
            bi = nn.Parameter(torch.randn(self.ws[i + 1]))
            self.register_parameter('weight_{}'.format(i), wi)
            self.register_parameter('bias_{}'.format(i), bi)
            self.weights += [wi]
            self.biases += [bi]

    def forward(self, X):
        """Return output"""
        Xf = self.get_features(X)
        pred = 1/np.sqrt(self.ws[-2]) * Xf @ self.weights[-1] + self.beta * self.biases[-1]
        return pred.flatten()

    def get_features(self, X):
        activ = X
        for i in range(self.depth):
            preactiv = 1/np.sqrt(self.ws[i]) * activ @ self.weights[i] + self.beta * self.
    biases[i]
            activ = torch.relu(preactiv)
        return activ


```

```python
if __name__ == "__main__":
    import matplotlib.pyplot as plt
    N_eval = 100
    n_layers = 4
    beta = 0.1
    epochs = 1000
    width = 1000
    lr = 1
    seed = 1  # Change here for neural networks with different initializations

    # Train data
    gamma_obs = np.array([-2, -1.2, -0.4, 0.9, 1.8])
    X_obs = np.stack([np.cos(gamma_obs), np.sin(gamma_obs)]).T
    Y_obs = X_obs.prod(axis=1)

    # Test data
    gamma_eval = np.linspace(-np.pi, np.pi, N_eval)
    X_eval = np.stack([np.cos(gamma_eval), np.sin(gamma_eval)]).T
    Y_eval = X_eval.prod(axis=1)

    # Define neural network
    torch.manual_seed(seed)
    net = FeedforwardNetwork(2, beta=beta, depth=n_layers, width=width)

    # Convert tensors from numpy to pytorch
    X_obs_pth = torch.Tensor(X_obs)
    Y_obs_pth = torch.Tensor(Y_obs)
    X_eval_pth = torch.Tensor(X_eval)
    # Train
    msg = 'Epoch = {} - loss = {:0.2f}'
    pbar = tqdm.tqdm(initial=0, total=epochs, desc=msg.format(0, 0))
    optimizer = optim.SGD(net.parameters(), lr=lr)
    for i in range(epochs):
        optimizer.zero_grad()
        Y_pred = net(X_obs_pth)
        error = (Y_obs_pth - Y_pred)
        loss = 1/2 * (error * error).mean()
        loss.backward()
        optimizer.step()
        # Update pbar
        pbar.desc = msg.format(i, loss)
        pbar.update(1)
    pbar.close()
    # Predict on test
    with torch.no_grad():
        Y_pred_eval = net(X_eval_pth).detach().numpy()

    # Plot on test data
    plt.plot(gamma_eval, Y_pred_eval, color='grey')
    plt.plot(gamma_eval, Y_eval, color='black')
    plt.plot(gamma_obs, Y_obs, '*', color='black', ms=10)
    plt.xlabel('gamma')
    plt.ylabel('f')
    plt.show()
```

Listing 2: Implementing a neural network (available on github.com/uu-sml/seminars-overparam-ml)

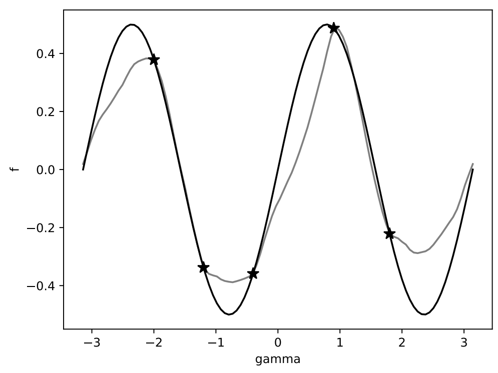


Figure 3: Output of the code. In the x-axis we have the angle γ, and on the y-axis the output of the function to an input $(\cos(\gamma_i), \sin(\gamma_i))$. The markers indicate the points that were observed and the black line the function f, used to generate them. The grey line is the neural network solution.