Graz University of Technology

442.022 - Nonlinear Signal Processing

Assignment 1

Author: Sophie Steger, Felix Stollberger
Date: Graz, May 6, 2021

Contents

1 Universal Approximators and System Identification

1.1 Problem 1

Note that code related to plotting was removed in this report to enhance the readability of the document!

This problem deals with different universal approximators for memoryless systems. For this purpose, a training set X_{train} consisting of $N_{train} = 50$, and a test set consisting of $N_{test} = 30$ samples was given. Both sets represent i.i.d. samples of a static nonlinearity $f(\cdot)$, corrupted by additive measurement noise ν, with $\nu_i \sim \mathcal{N}(0, \sigma_\nu^2)$.

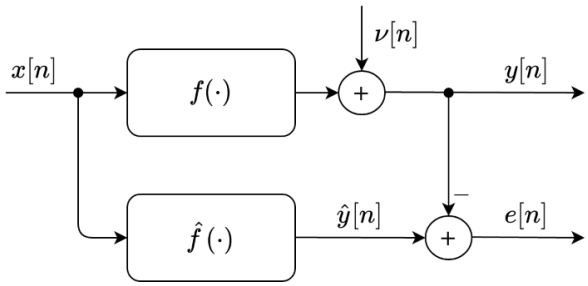


Figure 1.1: Setup for the system identification task of Problem 1

[1]:
```
%reload_ext autoreload
#%autoreload 2
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import warnings

# from src.models.mls_models import MyFancyModel
# from rbf_mlp import RbfNetwork
from src.utils.plotting import init_plot_style
%pylab
%matplotlib inline

init_plot_style()
data_dir='../../data/csv/1_1_system_identification/'

warnings.filterwarnings('ignore') # Supress warnings from polyfit
```

```
Using matplotlib backend: Qt5Agg
Populating the interactive namespace from numpy and matplotlib
```

First, let's take a look at the training and test data.

[2]:
```
training_set = pd.read_csv(data_dir + 'training-set.csv').to_numpy()
x_train, y_train = training_set[:,0], training_set[:,1]

test_set = pd.read_csv(data_dir + 'test-set.csv').to_numpy()
x_test, y_test = test_set[:,0], test_set[:,1]
```

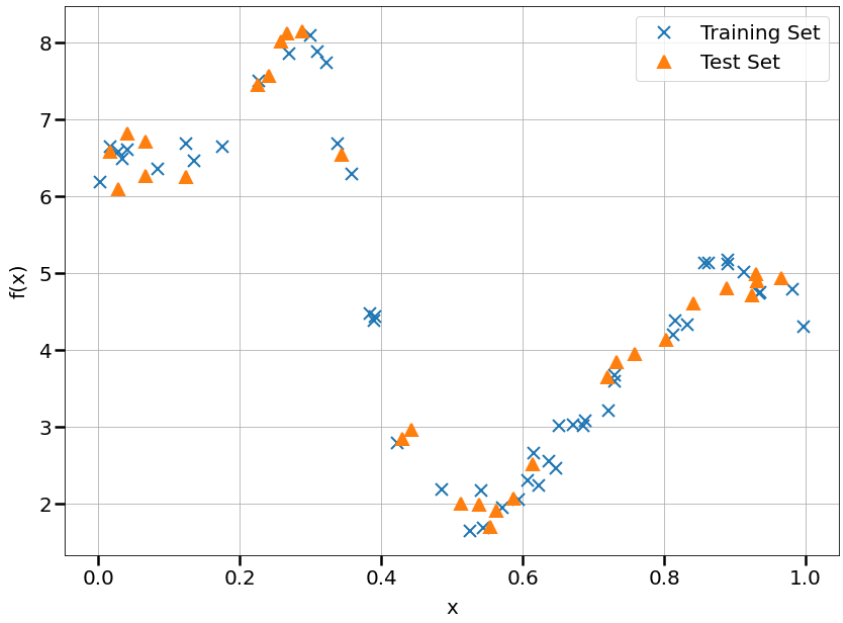


Figure 1.2: The given data sets

Run your experiments from here on!

Defenition of some functions for Task 1

[3]:
```
#Compute the MSE of the given coefficents p

def MSE(p,x,y):
        """Computing the 1D-MSE, given the polynomial coefficents p, the␣
 ↪places x and the data y  """
        y_hat = np.polyval(p,x)
        error = (y_hat-y)**2
        mse = np.mean(error)
        return mse


# Evaluate the polynomial model

def PolyModelEval(order, x_fit, y_fit, x):
        """Evaluate the poly model, fited to the data [x_fit, y_fit] at x"""
        p = np.polyfit(x_fit, y_fit, order, full=False)
        y_hat = np.polyval(p,x)
        return y_hat

# For RBF linear least squares solution
```

```
def RbfOptimizer(P, w, x):
        """Compute the matrix A for finding the linear least squares␣
 ↪solution based on the data x."""
        rbf_center = np.arange(0,1,1/P)
        A = np.ones([np.size(x), P])

        for i in range(0,np.size(x)):
            for j in range(1,P):
                A[i,j] = np.exp(1)**(-(x[i]-rbf_center[j])**2/w)
        return A
```

1.1.1 Task a)

The training set should be used to find a polynomial model:

$$\hat{f}(x) = \sum_{p=0}^{P} \alpha_p x^p \tag{1.1}$$

of order P ($P_{max} = 25$), and with the parameters $\theta = \{\alpha_p\}_{p=1}^{P}$ which is optimal with respect to the mean squared error (MSE):

$$J(\theta, P) = \frac{1}{N} \sum_{i=0}^{N-1} e_i^2 \tag{1.2}$$

[4]:
```
#First test with polyfit

P_max = 25
residuals_train = np.zeros(P_max-1)
residuals_test = np.zeros(P_max-1)

for order in range(1,P_max):

    p = np.polyfit(x_train, y_train, order, full=False)
    residuals_train[order-1] = MSE(p,x_train, y_train)
    residuals_test[order-1] = MSE(p,x_test, y_test)

train_min = np.where(residuals_train == np.amin(residuals_train)) #optimal␣
 ↪order
test_min = np.where(residuals_test == np.amin(residuals_test))    #optimal␣
 ↪order
```

Visual inspection of the results:

[5]:
```
#Plot of MSE for train and test

print('Optimal polynomial order for training set: ', train_min[0]+1, 'with␣
 ↪MSE: ', residuals_train[train_min[0]])
```

```
print('Optimal polynomial order for test set: ', test_min[0]+1, 'with MSE:␣
 ↪', residuals_test[test_min[0]])
x_eval = np.arange(0,1,0.005)
opt_train_model = PolyModelEval(train_min[0]+1, x_train, y_train, x_eval)
opt_test_model = PolyModelEval(test_min[0]+1, x_train, y_train, x_eval)
```

```
Optimal polynomial order for training set:  [23] with MSE:  [0.02241836]
Optimal polynomial order for test set:  [20] with MSE:  [0.04568998]
```

Comparing the training and test MSE curves shows a quite similar behavior for both sets. The MSE starts with a rapid decrease up to $P = 7$, followed by a slow, nearly monotonic decrease with increasing polynomial order. The difference between the MSE for the training-, and test-set is relatively small, indicating that all models generalize well up to the maximal polynomial order.

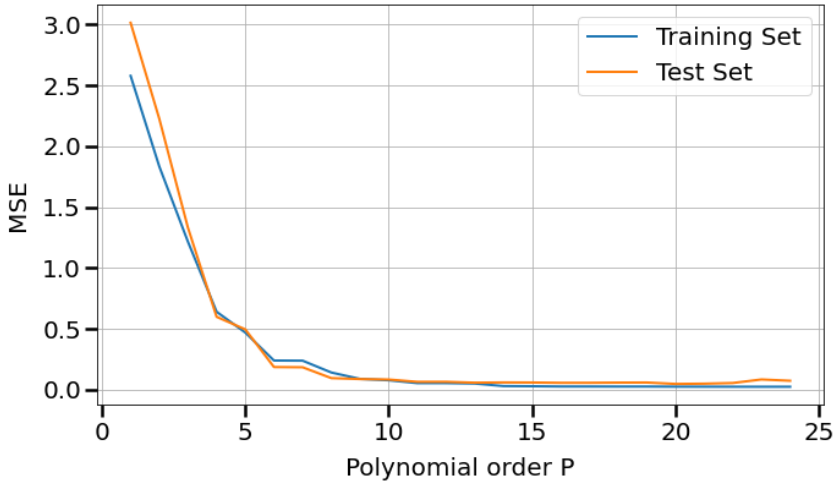


Figure 1.3: MSE of the training-, and test-set over polynomial order

Further analyzing the curve shows that the optimal training order is $P_{opt,train} = 23$, with an MSE of 0.0224. The lowest MSE on the test set is archived with $P_{opt,test} = 20$, where a MSE of 0.0457 was reached.

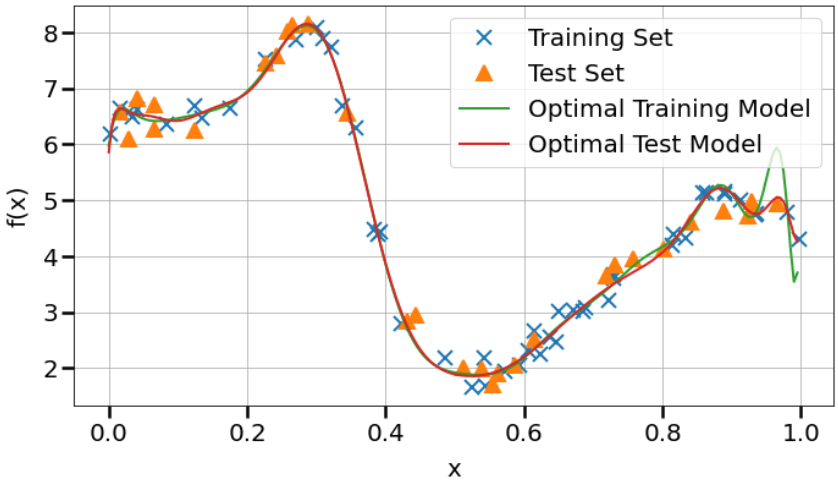


Figure 1.4: Optimal training-, and test-model

Figure 1.4 shows a comparison of the optimal training and test models over the training-, and test-samples. Both curves fit the data very well. Nevertheless, significant overfitting can be observed for the training model in the region of $x \in [0.95\ 1]$, which suggests that the model would generalize not as good as the test model. In this region, the polynomial is strongly fitted to the training samples, delivering an unrealistic behaviour besides those data points. Therefore, the test model with $P_{opt,test} = 20$ should be used to predict new samples which are not contained in the training-, or test-set.

As this behaviour is not obvious when solely comparing the MSE curve of both sets, a visual inspection of the final model (as long as possible) should be always considered. Thus, model errors based on eg. overfitting in regions with sparse data could be easily identified.

1.1.2 Task b)

Use a subset of the original training set as validation set. Training and validation set must be disjoint.

The splitting ratio between training-, and validation set can be changed by changing the following variable:

```
TV_ratio = 0.3 #in [0,1]
```

[6]:
```
TV_ratio = 0.3 #Define splitting ratio btw. training and validation set, in␣
↪[0,1]

N_sample = np.size(x_train)
Val_size = np.ceil(N_sample*TV_ratio)

rng = np.random.default_rng(7) #seed value set for repeatable results!!
pos = rng.choice(N_sample,Val_size.astype(int), replace=False) #Draw␣
↪non-repetitive random numbers

x_validation = x_train[pos]
y_validation = y_train[pos]

x_train2 = np.delete(x_train, pos)
y_train2 = np.delete(y_train, pos)

# Plot the splitted data-set
```

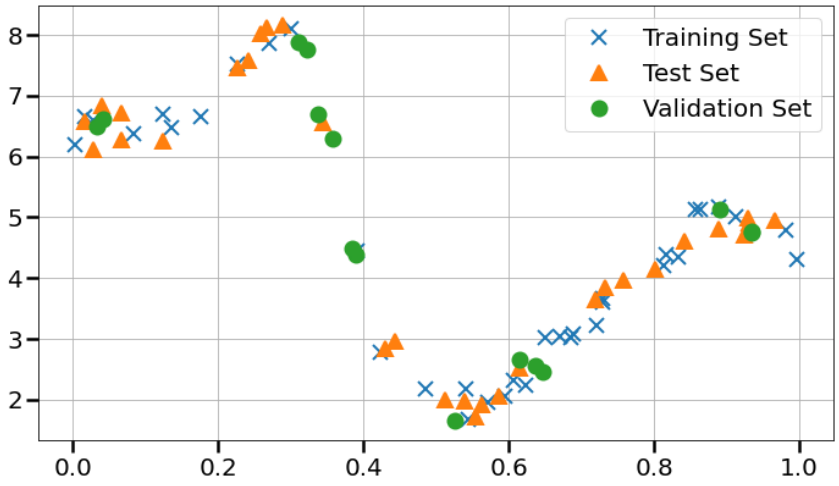


Figure 1.5: Overview training-, validation-, and test set

Evaluation based on validation set performance:

[7]:
```
P_max = 25

residuals_validation = np.zeros(P_max-1)
residuals_train = np.zeros(P_max-1)
residuals_test = np.zeros(P_max-1)

for order in range(1,P_max):

    p = np.polyfit(x_train2, y_train2, order, full=False)
    residuals_train[order-1] = MSE(p,x_train2, y_train2)
    residuals_validation[order-1] = MSE(p,x_validation, y_validation)
    residuals_test[order-1] = MSE(p,x_test, y_test)

validation_min = np.where(residuals_validation == np.
 ↪amin(residuals_validation)) #optimal order
test_min = np.where(residuals_test == np.amin(residuals_test))

print('Optimal polynomial order for validation set: ', validation_min[0]+1,␣
 ↪'with MSE: ', residuals_validation[validation_min[0]])
print('Optimal polynomial order for test set: ', test_min[0]+1, 'with MSE:␣
 ↪', residuals_test[test_min[0]])
```

```
Optimal polynomial order for validation set:  [21] with MSE:  [0.03892393]
Optimal polynomial order for test set:  [21] with MSE:  [0.05241151]
```

Visual inspection of the results:

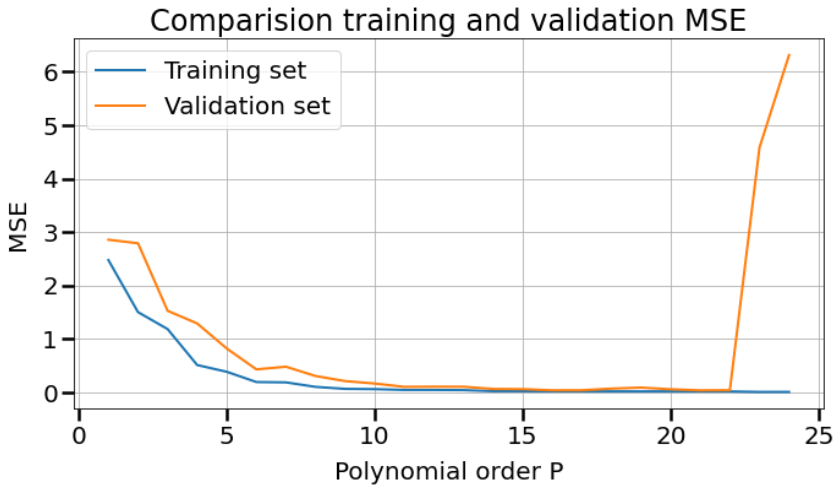


Figure 1.6: MSE of the training-, and validation-set over polynomial order

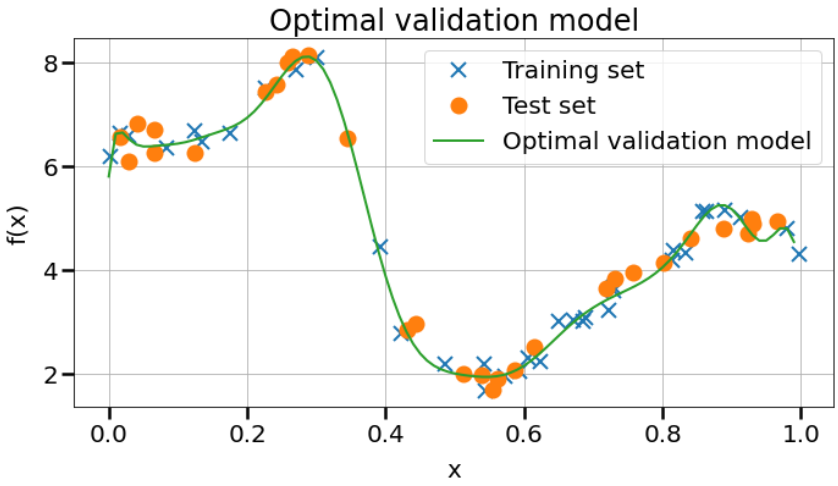


Figure 1.7: Optimal validation model

Figure 1.6. shows again the MSE of the different models, over the training-, and validation sets for increasing polynomial order P. The behaviour of the MSE for low polynomial order is quite similar to the previous case with the test set. Nevertheless, the MSE difference in this region is much larger. After $P \sim 11$, both curves converge approximately to the same value, with a minimum MSE on the validation set at $P = 21$. Further increasing P causes a rapid increase of the MSE on the validation set, while the training MSE is still decreasing. This clearly indicates overfitting to the training set.

As mentioned, the best model for the validation set has order $P = 21$, and archives a MSE of 0.052 on the test set. This is a quite similar result as in the first task, although the overall MSE is slightly higher. Figure 1.7 shows this optimal validation model along with the given data for the (reduced) training set, and the test set.

This split size has a direct effect on the model quality, as we are working with a relatively small data set. Therefore, increasing the `TV_ratio` reduces the training set, which is necessary for basic model construction. On the other hand, the quality of the validation set improves, as more and more samples are available for model evaluation. In contrast to that, a small `TV_ratio` might not represent the actual function in the validation set, as only very few samples are available for the model evaluation. As a consequence, unfavorable models might yield a small validation MSE.

In our case, we came up with the optimal splitting ratio by varying its value, and analyzing the MSE on the training, and validation set. The lowest MSE was archived for a `TV_ratio` of 0.3, and yielded in $MSE_{valid} = 0.039$, and $MSE_{test} = 0.0524$

1.1.3 Task c)

Use a Gaussian radial basis function (RBF) model $\hat{f}$ of order P:

$$\hat{f}(x) = \alpha_0 + \sum_{p=1}^{P} \alpha_p e^{\frac{-(x-c_p)^2}{2\omega_p^2}} \tag{1.3}$$

To avoid a nonlinearity, the centers of the P-RBF are chosen evenly spaced over the support of the given data. Using the given data sets $\mathbf{x}[n]$, and $\mathbf{y}[n]$, a simplification to a linear model in the coefficients α_p is possible. This can be further rewritten to a linear matrix vector multiplication:

$$\mathbf{y}_{train} = \mathbf{A} \cdot \alpha \tag{1.4}$$

Where A is defined as:

$$A = \begin{bmatrix} 1 & e^{\frac{-(x[0]-c_1)^2}{2\omega^2}} & e^{\frac{-(x[1]-c_1)^2}{2\omega^2}} & \dots & e^{\frac{-(x[N]-c_1)^2}{2\omega^2}} \\ 1 & e^{\frac{-(x[0]-c_2)^2}{2\omega^2}} & e^{\frac{-(x[1]-c_2)^2}{2\omega^2}} & \dots & e^{\frac{-(x[N]-c_2)^2}{2\omega^2}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & e^{\frac{-(x[0]-c_P)^2}{2\omega^2}} & e^{\frac{-(x[1]-c_P)^2}{2\omega^2}} & \dots & e^{\frac{-(x[N]-c_P)^2}{2\omega^2}} \end{bmatrix} \tag{1.5}$$

And α denotes the coefficient vector, consisting of all α_i.

The optimal coefficient vector α_{opt} is determined by means of minimizing the MSE for the given model order P:

$$\alpha_{opt} = \underbrace{(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T}_{\text{Moore–Penrose inverse}} \cdot \mathbf{y}_{train} \tag{1.6}$$

We implemented the computation of matrix A in the function `RbfOptimizer(P, w, x)`, and computed the optimal coefficient vector using `np.linalg.pinv(A).dot(y_train)`. The benefit of this method is that once A was computed, the model could be easily evaluated by a multiplication with the coefficient vector α.

Model design

Three parameters must be manually chosen before the RBF-model could be evaluated. The first one, P, determines the number of Gaussian basis functions, used to approximate the nonlinear

system. It is a very crucial parameter, as it directly affects the performance of the model. Choosing P too low will yield in a very smooth model (if ω^2 is large), which is not capable of modeling any rapid changes in the evaluated system. This is a direct effect of the inherent smoothness of Gaussian RBF. On the other hand, a large P, compared with a small width parameter ω^2 will create a delta like model, which is very susceptible to overfitting. (Can be easily created by changing P to 200) Hence, we choose $P = 20$, as it seems like a reasonable number for representing the given system.

Once P is fixed, the individual positions of the basis functions c_p must be defined. This was straightforward in our case, as the RBF should be evenly spaced over the support. Therefore, their position was computed using `rbf_center = np.arange(0,1,1/P)`.

The last parameter to choose was the width parameter ω^2. It defines the width of the Gaussian basis functions, and was chosen equally for all functions ($\omega^2 = \omega_p^2$) to simplify the computation of the model. As previously mentioned, a smaller ω^2 results in narrower basis functions, which can be used to model e.g. a rapidly changing function. Nevertheless, problems with overfitting might occur. If ω^2 is too large, the the individual basis functions start to overlap significantly, and can not resolve rapid changes, or smaller structures in the original system. We have chosen a dynamic width parameter for our system, depending on the number of basis functions with $\omega^2 = \left(\frac{2}{\alpha}\right)^2$. Therefore, a sufficient overlap between the RBG could be guaranteed, while maintaining the necessary model resolution.

Problems could occur if the width parameter ω^2 is independent of the model order P, as a smaller order requires wider basis functions to cover the full domain of x. If this connection is omitted, the model would be unable to model the system. In this case, the narrow RBF's would drop so fast towards zero, that the region between adjacent RBF's could not modeled with the available functions. The problem is less important in the other case - many RBF's with large width parameter, as some RBF's could be simply suppressed by setting their corresponding α to zero.

Model evaluation

The optimal model order was determined by evaluating the model with the reduced test set, and computing the MSE on the validation set. The optimal model on the validation set was then evaluated on the test set. The location parameter c_p, and the width parameter ω^2 where chosen as previously described.

[15]:
```
#Determine the optimal order based on validation set performance
P = 26 # max number of RBF's

mse_train = np.zeros([P-1,1])
mse_valid = np.zeros([P-1,1])

for order in range(1,P):
    w2 = (2/order)**2                    #width parameter squared
    rbf_center = np.arange(0,1,1/order)  #RBF centers

    A = RbfOptimizer(order, w2, x_train2)
    c_opt = np.linalg.pinv(A).dot(y_train2)

    y_hat_train = RbfOptimizer(order, w2, x_train2).dot(c_opt)
    y_hat_valid = RbfOptimizer(order, w2, x_validation).dot(c_opt)
    mse_train[order-1] = np.mean((y_hat_train-y_train2)**2)
```

```
    mse_valid[order-1] = np.mean((y_hat_valid-y_validation)**2)

mse_min_train = np.where(mse_train == np.amin(mse_train)) #optimal order

print('Optimal RBF order for training set: ', mse_min_train[0]+1, 'with MSE:␣
 ↪', mse_train[mse_min_train[0]])

mse_min_valid = np.where(mse_valid == np.amin(mse_valid)) #optimal order
print('Optimal RBF order for validation set: ', mse_min_valid[0]+1, 'with␣
 ↪MSE: ', mse_valid[mse_min_valid[0]])
```

```
Optimal RBF order for training set:  [25] with MSE:  [[0.00609635]]
Optimal RBF order for validation set:  [13] with MSE:  [[0.04833838]]
```

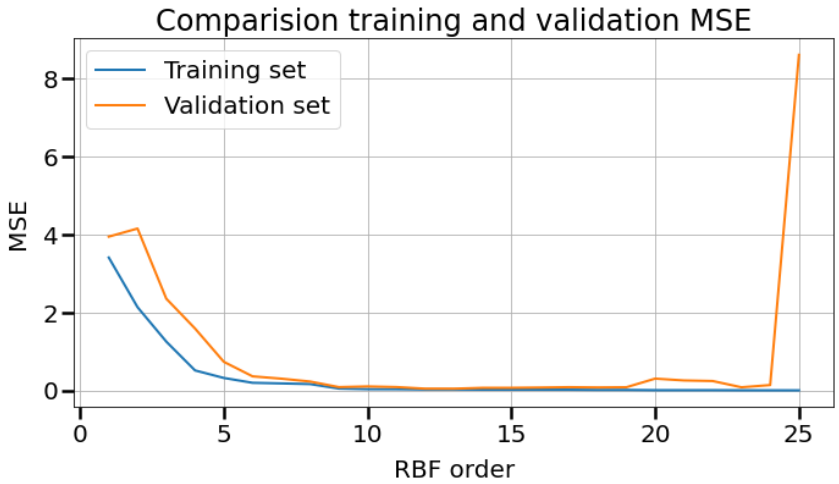


Figure 1.8: MSE of the training-, and validation-set over RBF order

Figure 1.8 shows again the MSE over the RBF order. The lowest MSE on the validation set was archived for $P = 13$. Therefore, a RBF model of this order was then evaluated on the test set:

[10]:
```
P = 13 #number of RBF's
rbf_center = np.arange(0,1,1/P)
w2 = (2/P)**2 #width parameter sqared
A = RbfOptimizer(P, w2, x_train2)
c_opt = np.linalg.pinv(A).dot(y_train2)
#Evaluate the model with test set
y_hat = RbfOptimizer(P, w2, x_test).dot(c_opt)  #for MSE compuation
mse = np.mean((y_hat-y_test)**2)

print('MSE on test set: ', mse)

x_eval = np.arange(0,1,0.01)
y_eval = RbfOptimizer(P, w2, x_eval).dot(c_opt) #for plot
```

```
MSE on test set:  0.050864889480220825
```

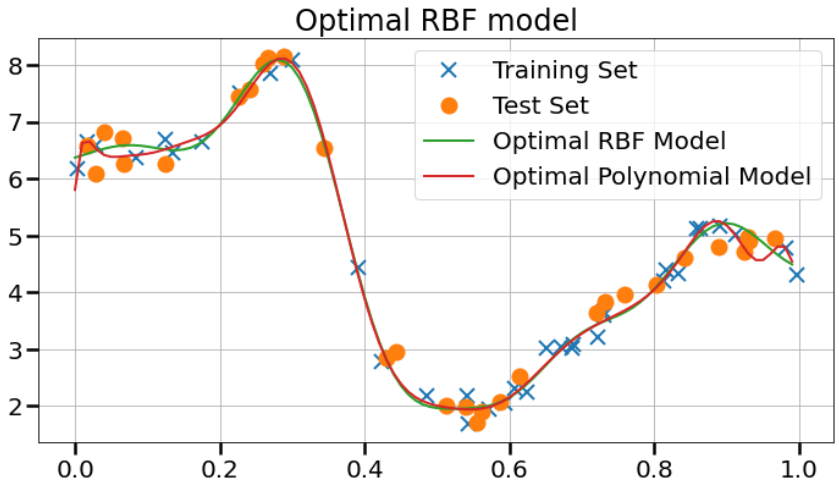


Figure 1.9: Model comparision of the optimal RBF, and polynomial model

The RBF model of order $P = 13$ archived an MSE of 0.0508 on the test set. This is slightly lower than the 0.0524, archived with the polynomial model of order $P = 21$.

Figure 6. shows now a direct comparison of the optimal (w.r.t the MSE on the validation set) RBF-, and polynomial models. Both models fit very well to the data, and behave nearly identical in the region between 0.2 and 0.9. Nevertheless, the polynomial model shows beginning overfitting in the outer regions. Hence, the RBF model should be preferred over the polynomial model for predicting samples outside the given test, and training sets.

Manually selecting the widths-, and centers will be problematic for high-dimensional input data. In this case, the evenly spaced RBF, and potential different ω^2 would create a model with a huge amount of parameters (α_p) which must be optimized. Not only will this be a very difficult task, but the high complexity is also surely not necessary. The data will most likely populate some subspace of the high dimensional input space, such that a modeling capability in the whole R^{100} is surely not needed. Therefore, learning the width-, and position parameters is an important task.

1.1.4 Task d)

Learn the optimal centers and widths using gradient descent (GD) applied on a RBF-NN.

Implementing the RBF-model in a neuronal-network (NN) is not a straightforward process. Therefore, is is always a good idea to start small, determine the dimensionality of the variables, and explicitly formulate the nonlinear transform depicted in the NN. Starting with $P = 3$, a RBF-NN for $x \in \mathrm{R}^1, y \in \mathrm{R}^1$ and $\hat{f}(x) : \mathrm{R}^1 \to \mathrm{R}^1$ would look like:

$$y = \hat{f}(x) = \alpha_0 + \sum_{p=1}^{3} \alpha_p \left[\phi(\omega_{p,1} \cdot x_1 + \omega_{p,0})\right] \tag{1.7}$$

This could be expressed for the given Gaussian basis functions (RBF):

$$y = \hat{f}(x) = \alpha_0 + \sum_{p=1}^{P} \alpha_p \mathrm{e}^{\left[\frac{-1}{2\omega_p^2}(x-c_p)^2\right]} \tag{1.8}$$

Which is equivalent to:

$$y = \hat{f}(x) = \alpha_0 + \sum_{p=1}^{P} \alpha_p \mathrm{e}^{\left(-\left[\sqrt{\frac{1}{2\omega_p}}\cdot x - c_p\sqrt{\frac{1}{2\omega_p^2}}\right]^2\right)} \tag{1.9}$$

In this case, the non linear activation function $\phi(x) = \mathrm{e}^{-[x]^2}$, as well as the linear transform:

$$z(x) = \underbrace{\sqrt{\frac{1}{2\omega_p}}}_{\omega_{p,1}} \cdot x + \underbrace{(-c_p\sqrt{\frac{1}{2\omega_p^2}})}_{\omega_{p,0}}, \quad \text{so} \quad z_p = \omega_{p,1} \cdot x + \omega_{p,2} \tag{1.10}$$

becomes visible. Following from that, a simple sketch of the NN can be drawn:

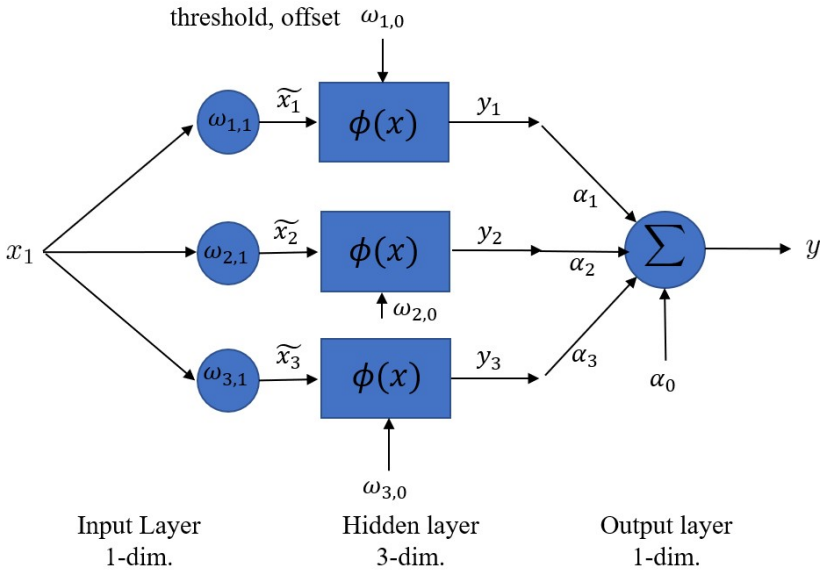


Figure 1.10: Sketch of a simple NN with order P=3

As we deal with a one dimensional input parameter, the weight matrix simplifies to a weight vector:

$$\boldsymbol{\omega}_{p_1} = \left[\sqrt{\frac{1}{2\omega_1}}, \sqrt{\frac{1}{2\omega_2}}, \sqrt{\frac{1}{2\omega_3}}\right]^T, \quad \text{and} \quad \boldsymbol{\omega}_{p_0} = \left[-c_1\sqrt{\frac{1}{2\omega_1^2}}, -c_2\sqrt{\frac{1}{2\omega_2^2}}, -c_3\sqrt{\frac{1}{2\omega_3^2}}\right]^T \tag{1.11}$$

Compute the gradients

The gradient ∇ could be computed with respect to the weights ($\nabla\alpha$), the width parameters ($\nabla\omega$) or the centers (∇c). Each gradient is hereby defined as:

$$\nabla_x = \begin{pmatrix} \frac{\partial}{\partial x_1} \\ \frac{\partial}{\partial x_2} \\ \frac{\partial}{\partial x_3} \end{pmatrix} \tag{1.12}$$

Therefore:

$$\nabla_\omega \hat{f}(x) = \begin{pmatrix} \alpha_1 \frac{1}{\omega_1^3}(x-c_1)^2 \mathrm{e}^{\left(\frac{-1}{2\omega_1^2}(x-c_1)^2\right)} \\ \alpha_2 \frac{1}{\omega_2^3}(x-c_2)^2 \mathrm{e}^{\left(\frac{-1}{2\omega_2^2}(x-c_2)^2\right)} \\ \alpha_3 \frac{1}{\omega_3^3}(x-c_3)^2 \mathrm{e}^{\left(\frac{-1}{2\omega_3^2}(x-c_3)^2\right)} \end{pmatrix} \tag{1.13}$$

as well as:

$$\nabla_c \hat{f}(x) = \begin{pmatrix} \alpha_1 \frac{1}{\omega_1^2}(x-c_1) \mathrm{e}^{\left(\frac{-1}{2\omega_1^2}(x-c_1)^2\right)} \\ \alpha_2 \frac{1}{\omega_2^2}(x-c_2) \mathrm{e}^{\left(\frac{-1}{2\omega_2^2}(x-c_2)^2\right)} \\ \alpha_3 \frac{1}{\omega_3^2}(x-c_3) \mathrm{e}^{\left(\frac{-1}{2\omega_3^2}(x-c_3)^2\right)} \end{pmatrix} \qquad \nabla_\alpha \hat{f}(x) = \begin{pmatrix} 1 \\ \mathrm{e}^{\left(\frac{-1}{2\omega_1^2}(x-c_1)^2\right)} \\ \mathrm{e}^{\left(\frac{-1}{2\omega_2^2}(x-c_2)^2\right)} \\ \mathrm{e}^{\left(\frac{-1}{2\omega_3^2}(x-c_3)^2\right)} \end{pmatrix} \tag{1.14}$$

Model implementation

For this task, the given MLP was adapted to utilize RBFs as activation functions. This was easily implemented in the forward-pass by computing the output of the input layer with a Gaussian function `z1 = torch.exp(-torch.pow(self.hidden_layer_1(x),2))`. The model order was chosen empirically, and defined as 20. It was trained using the reduced training set to guarantee comparable results for all tasks. Afterwords, the model was evaluated on the test set to determine the archived MSE.

[11]:
```
from src.models.rbf_mlp import RbfMLP

rbf_mlp_model = RbfMLP(hidden_size = 20) #change model order with hidden size
loss_list = rbf_mlp_model.fit(x_train2, y_train2, learning_rate=1e-2,␣
 ↪max_epochs=1000)

plt.figure(figsize=(10,6))
plt.plot(list(range(1, 1 + len(loss_list))), loss_list)
plt.xlabel('Epoch')
plt.ylabel('Training Set MSE')
plt.grid()
plt.tight_layout()
plt.show()
```

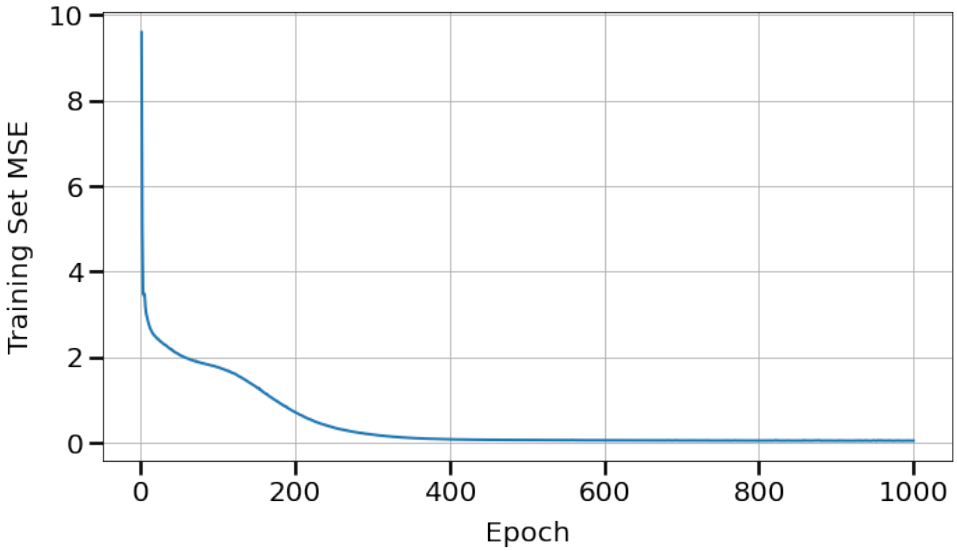


Figure 1.11: Evolution of the training MSE over the epochs.

[12]:
```
x_eval = np.arange(0,1,0.01)

approx_output = rbf_mlp_model.predict(x_eval)
y_pred = rbf_mlp_model.predict(x_test)

print(f'Training MSE is {loss_list[-1]:.4f}')
print(f'Test MSE is {np.mean((y_test-np.transpose(y_pred))**2):.4f}')
```

```
Training MSE is 0.0385
Test MSE is 0.0366
```

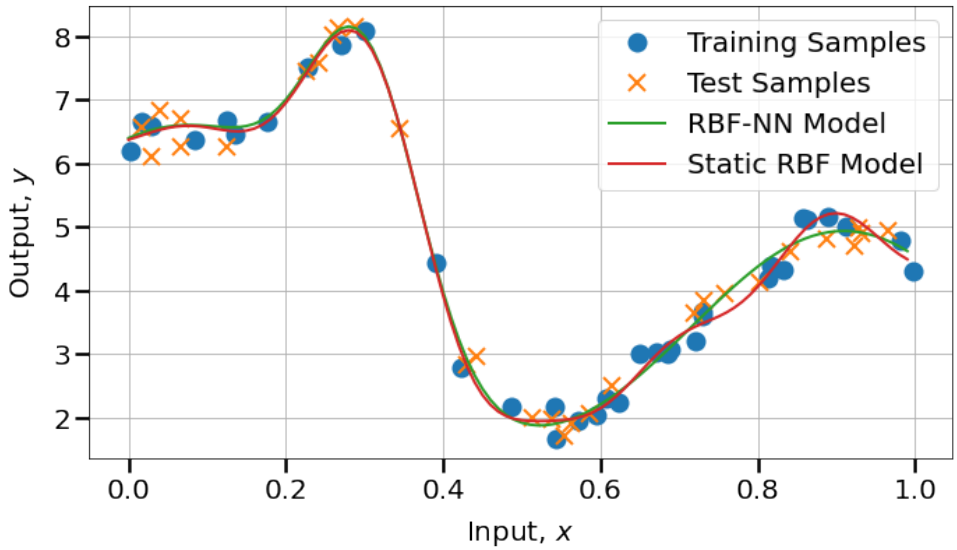


Figure 1.12: Comparison of the different RBF models

Figure 1.12 shows a comparison of the best RBF-NN model, and the best static RBF-model. The plotted RBF-NN model was found using a single layer NN with 20 neurons, 1000 epochs

of training and a learning rate of $\eta = 0.01$. The biggest difference is visible in the region of $x \in [0.7\ 1]$, where the RBF-NN model shows a smoother behaviour than the static-RBF model. This seems like a more realistic behaviour of the nonlinearity, and suggests together with significantly lower training MSE of $MSE_{train} = 0.326$, that the RBF-NN model performs better. Interestingly, the MSE is even lower on the test-, than on the training set, which suggests very good generalization.

Nevertheless, the result was not surprising, as variable position-, and width parameters clearly extend the modeling capabilities of the RBF-model. The problems of to small/large overlap between basis-functions as described in Task c), can be avoided with this method.

Finding the centers and width parameters

RBF are very common activation functions in neuronal networks. Generally, such NN are trained in a two stage process. The first stage fixes the number (P) of RBF, and determines their centres, as well as their width parameters. Afterwards, in the second step, the weighting coefficient α_i are learned to fit the network to the training data.

One approach for determining P, and $\mathbf{c}$ is to fit a RBF to each data sample. This guarantees an exact fit of the model to the training data. Nevertheless, such an approach is only useful for noise free data. Hence, other methods where developed, which are e.g. based on randomly locating the centers, or a prior clustering of the training data with algorithms as K-means. The benefit of prior clustering is that the RBF could be placed in the center of the cluster, and represent therefore the density of the training data. Yousef and Hindi [3] states additional methods which are based on genetic algorithms, decision trees, or supervised learning.

After these two parameters are fixed, the with parameter must be chosen, or learned. A common procedure is to choose ω^2 heuristically, and fixing its value for all RBF-kernels. A more formal way is to define ω_p^2 by using the euclidean distance to the adjacent RBF. A problem with this method is the increased computational effort, especially for higher dimensional data sets. One way to overcome this problem is again to fix the width for all RBF, and define the common with parameter over the maximum distance (d) between two chosen centres for all k RBF's: [3]

$$\omega^2 = \frac{d}{\sqrt{2k}} \tag{1.15}$$

Task e)

Approximation using a feed-foward neuronal network.

For this task, the previous RBF-NN was extended such that the layer size, as well as the number of neurons per layer could be chosen manually. The network is defined using the parameter `h_list` whose size define the number of layers in the network, and whose elements define the number of neurons per layer. `h_list = [20,20,20]` would for example create a 3-layer NN, with 20 neurons per layer. The activation function of the network is fixed, and could be changed if necessary in the `forward` function of the `variable_rbf_mlp.py`. Note that this model assumes at least two layers!

Training

As in the previous cases, the model was again trained with the reduced data set for a better comparability with the other models. Overall, four different activation functions where tried: the Gaussian RBF, tanh, relu, and the sigmoidal activation function. Each function was tested for selected combinations of 2-5 layers with 20-200 neurons per layer. The model was trained with a learning rate $\eta = 0.005$ for 1000 epochs.

[33]:
```
from src.models.variable_rbf_mlp import VarRbfMLP

h_list = [20, 20] # [20, 20, 20] creates a 3 layer NN with 20 neurons per␣
 ↪layer

rbf_mlp_model = VarRbfMLP(h_sizes = h_list)
loss_list = rbf_mlp_model.fit(x_train2, y_train2, batch_size=15
                              , learning_rate=9e-3, max_epochs=10000)

x_eval = np.arange(0,1,0.01)

approx_output_mlnn = rbf_mlp_model.predict(x_eval)
y_pred = rbf_mlp_model.predict(x_test)
```

[39]:
```
print('')
print(f'Training MSE is {loss_list[-1]:.4f}')
print(f'Test MSE is {np.mean((y_test-np.transpose(y_pred))**2):.4f}')
```

```
Training MSE is 0.0692
Test MSE is 0.0289
```

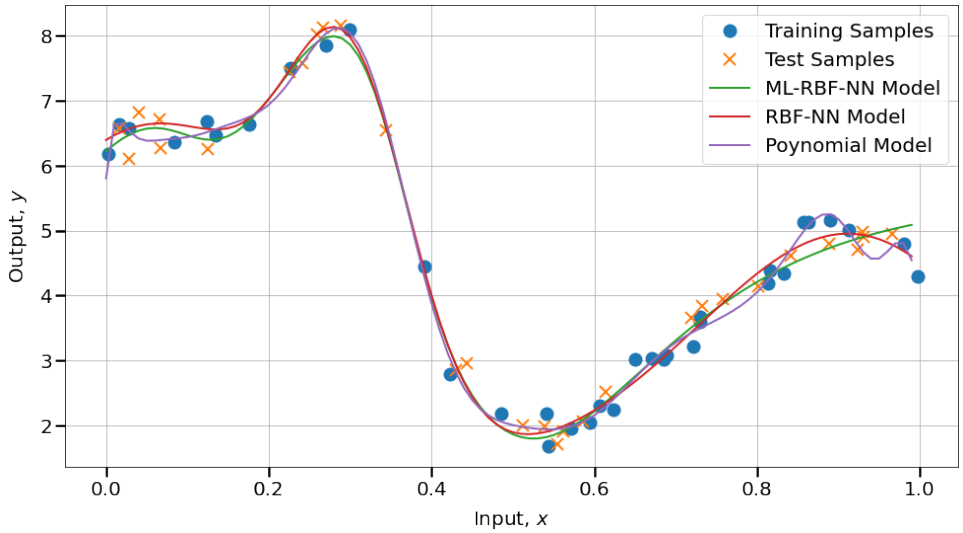


Figure 1.13: Model comparison

The final model of the multilyaer NN used a sigmoidal activation function in a 2 layer network with 20 neurons per layer. The networks was trained for 10000 epochs with $\eta = 0.009$, and reached a MSE_{test} of 0.0289. This is a smaller MSE than for the single-layer RBF-model!

Figure 1.13 shows a comparison of this model with the optimal single-layer RBF model. The increased performance on the training set can be explained when the region around $x = 0.95$ is considered, as the RBF-NN model is decreasing to adapt to the training samples, while the ML-RBF-NN model increases, and fits better to the test samples. As the model performance should not be solely determined by the MSE, the RBF-NN model should be still considered as the better model.

Task f)

Conclude your findings of the experiments.

The **sigmodidal activation function** delivered the best results beside the Gaussian RBF. A combination of a 3 layer network with 200 neurons per layer resulted in a training error MSE_{train} of 0.023, and a test error MSE_{test} of 0.054. Increasing the number of layers, as well as the number of training epochs decreased the MSE on the training set further (down to 0.198), but increased again the MSE on the test set as the model began with overfitting. Changing the model structure to a two layer network with 20 neurons increased the performance on the test set drastically, and delivered a MSE_{test} of 0.0289, which is even better than the RBF model (but see the comments in Task e)).

The **tanh activation function** performed similar (concerning the MSE) to the sigmoidal one, but delivered models which where not as smooth, which could be also interpreted as beginning overfitting. In addition to that, the MSE on the test set was with 0.930 the largest for all used activation functions.

The **rectifier-linear activation function** performed relatively good on the training set, delivering a MSE_{train} of 0.031. It outperformed also the tanh activation function on the test set, with a MSE_{test} of 0.056. The nature of the activation function was also visible in the estimated model, as is consisted of linear segments which where smoothly connected.

A multi-layer NN with **Gaussian RBF** performed approximately as good as the single layer RBF-NN. The $MSE_{test} = 0.037$ was nevertheless still larger as for the single layer NN. Increasing either the number of neurons, or the number of layers decreased the model performance as overfitting became visible.

Overall, this problem dealt with four different approaches to model a nonlinear system, which was observed trough noisy measurements. The complexity of the individual approaches increased, and started with a simple polynomial fit. The benefit of this method is its simplicity, compared with the acceptable results. Nevertheless, polynomial models tend to overfitting, and are unbound outside the domain of the data.

The next approach used equally spaced Gaussian basis functions with predefined width-parameters. This approach has also a very low computational complexity, as the scaling coefficients for the RBF's can be found by a matrix inversion. The performance of the static-RBF model was slightly better than the polynomial model. Additionally, a RBF model decays towards zero outside the domain due to the nature of the Gaussian RBF's.

The width-, and position parameters of the RBF model could be learn in a NN. With this, a much better model performance could be reached, as the individual RBFs are adapted to the behaviour of the data. This model performed best in therms of generalization for the the given data set. The biggest drawback is the higher computational cost for optimizing-, and training the NN.

The last method used a multi-layer NN. This is the most powerfull approach, which delivered also the lowest MSE of all models on the test set ($MSE_{min} = 0.0289$). Albeit the lowest MSE, the resulting model looked not as good as the ML-RBF model, as it deviated from the possible trajectory of the original system. Multi-layer NN show an even higher computational complexity, and might not be the best choice for such a simple problem as in this task. This could be also seen by the fact that the best model was still relatively small (2 layer, 20 neuron), and nearly all larger models showed some kind of overfitting.

2 Harmonic Analysis and Equalization

2.1 Problem 2

In this problem, a memoryless system

$$y[n] = f(x[n]) \tag{2.1}$$

should be approximated with a 3^{rd}-order Taylor series expansion. The input signal is sinusoidal $x[n] = \cos(\theta n)$ and the approximation has the general form

$$\hat{f}\left(x\left[n\right]\right) = \alpha_0 + \alpha_1 \cos(\theta n) + \alpha_2 \cos(2 \cdot \theta n) + \alpha_3 \cos(3 \cdot \theta n) \tag{2.2}$$

2.1.1 Task a)

In Task a) the coefficients $\alpha_0, \ldots, \alpha_3$ are derived analytically. The coefficients can be interpreted as the coefficients of a Fourier Series.

$$\hat{f}\left(x\left[n\right]\right) = \overbrace{f(z)|_{z=c}}^{:=f_c} + \overbrace{\left.\frac{df\left(z\right)}{dz}\right|_{z=c}}^{:=f_c'}\left(x\left[n\right]-c\right) + \frac{1}{2}\overbrace{\left.\frac{d^2 f\left(z\right)}{dz^2}\right|_{z=c}}^{:=f_c''}\left(x\left[n\right]-c\right)^2 + \tag{2.3}$$

$$+ \frac{1}{6}\underbrace{\left.\frac{d^3 f\left(z\right)}{dz^3}\right|_{z=c}}_{:=f_c'''}\left(x\left[n\right]-c\right)^3 \tag{2.4}$$

Inserting $x[n] = A \cdot \cos(\theta n)$ leads to

$$\hat{f}\left(x\left[n\right]\right) = f_c + f_c'\left(x[n]-c\right) + \frac{1}{2}f_c''\left(x[n]-c\right)^2 + \frac{1}{6}f_c'''\left(x[n]-c\right)^3 \tag{2.5}$$

$$= f_c + f_c'\left(x\left[n\right]-c\right) + \frac{1}{2}f_c''\left(x^2\left[n\right] - 2x\left[n\right]c + c^2\right) + \tag{2.6}$$

$$+ \frac{1}{6}f_c'''\left(x^3\left[n\right] - 3x^2\left[n\right]c + 3x\left[n\right]c^2 - c^3\right) \tag{2.7}$$

$$= f_c + f_c'\left(A \cdot \cos(\theta n) - c\right) + \frac{1}{2}f_c''\left(A^2 \cdot \cos^2(\theta n) - 2A \cdot \cos(\theta n)c + c^2\right) + \tag{2.8}$$

$$+ \frac{1}{6}f_c'''\left(A^3 \cdot \cos^3(\theta n) - 3A^2 \cdot \cos^2(\theta n)c + 3A \cdot \cos(\theta n)c^2 - c^3\right) \tag{2.9}$$

With the trigonometric identities

$$\cos^2\varphi = \frac{1}{2}\left(\cos\left(2\varphi\right) + 1\right) \qquad \text{and} \qquad \cos^3\varphi = \frac{1}{4}\left(3\cos\varphi + \cos\left(3\varphi\right)\right)$$

the term above can be rewritten.

$$\hat{f}\left(x\left[n\right]\right) = f_c + f_c'\left(A\cdot\cos(\theta n) - c\right) + \tag{2.10}$$

$$+\frac{1}{2}f_c''\left(\frac{A^2}{2}+\frac{A^2}{2}\cos\left(2\theta n\right) - 2A\cos\left(\theta n\right)c + c^2\right) + \tag{2.11}$$

$$+\frac{1}{6}f_c'''\left(\frac{3}{4}A^3\cos\left(\theta \mathrm{n}\right) + \frac{A^3}{4}\cos\left(3\theta \mathrm{n}\right) - \frac{3}{2}A^2c - \frac{3}{2}A^2\cos\left(2\theta \mathrm{n}\right)c + \right. \tag{2.12}$$

$$\left. + 3A\cos\left(\theta \mathrm{n}\right)c^2 - c^3\right) \tag{2.13}$$

Finally, the term is factorized to obtain the coefficients $\alpha_0, \ldots, \alpha_3$.

$$\hat{f}\left(x\left[n\right]\right) = 1\cdot\underbrace{\left[f_c - f_c'\,c + f_c''\left(\frac{c^2}{2}+\frac{A^2}{4}\right) + f_c'''\left(-\frac{A^2c}{4}-\frac{c^3}{6}\right)\right]}_{:=\alpha_0} + \tag{2.14}$$

$$+\cos(\theta n)\underbrace{\left[f_c'A - f_c''Ac + f_c'''\left(\frac{A^3}{8}+\frac{Ac^2}{2}\right)\right]}_{:=\alpha_1} + \tag{2.15}$$

$$+\cos(2\theta n)\underbrace{\left[f_c''\frac{A^2}{4} - f_c'''\frac{A^2c}{4}\right]}_{:=\alpha_2} + \tag{2.16}$$

$$+\cos(3\theta n)\underbrace{\left[f_c'''\frac{A^3}{24}\right]}_{:=\alpha_3} \tag{2.17}$$

2.1.2 Task b)

In Task b) the results from Task a) are used to derive the third order Taylor approximation around the centers $c \in \{0, \ln(2)\}$.

Sigmoid function and derivatives

$$f(z) = \frac{1}{1+e^{-z}} \qquad f''(z) = -\frac{e^z(e^z-1)}{(e^z+1)^3}$$

$$f'(z) = \frac{e^{-z}}{(1+e^{-z})^2} \qquad f'''(z) = \frac{e^z(-4e^z+e^{2z}+1)}{(e^z+1)^4}$$

Center: $c = 0$

$$f(z)\big|_{z=c} = f(0) = \frac{1}{1+e^{-0}} = \frac{1}{2} \qquad f''(z)\big|_{z=c} = f''(0) = -\frac{e^0(e^0-1)}{(e^0+1)^3} = 0$$

$$f'(z)\big|_{z=c} = f'(0) = \frac{e^{-0}}{(1+e^{-0})^2} = \frac{1}{4} \qquad f'''(z)\big|_{z=c} = f'''(0) = \frac{e^0(-4e^0+e^0+1)}{(e^0+1)^4} = -\frac{1}{8}$$

Center: $c = \ln(2)$

$$f(z)\big|_{z=c} = f(\ln(2)) = \frac{1}{1+e^{-\ln(2)}} = \frac{2}{3}$$

$$f'(z)\big|_{z=c} = f'(\ln(2)) = \frac{e^{-\ln(2)}}{(1+e^{-\ln(2)})^2} = \frac{2}{9}$$

$$f''(z)\big|_{z=c} = f''(\ln(2)) = -\frac{e^{\ln(2)}(e^{\ln(2)}-1)}{(e^{2\ln(2)}+1)^3} - \frac{2}{27}$$

$$f'''(z)\big|_{z=c} = f'''(\ln(2)) = \frac{e^{\ln(2)}(-4e^{\ln(2)}+e^{2\ln(2)}+1)}{(e^{\ln(2)}+1)^4} = -\frac{2}{27}$$

2.1.3 Task c)

Now that all analytic derivations are performed, it is time to verify the results for both approximations from Task b) numerically for the input signal parameters $A \in \{1, 3\}$ and $\theta = \frac{2\pi}{5}$. The length N of the discrete signal $x[n]$ was chosen such that it contains an integer multiple of periods in order to avoid leakage effects when computing the Discrete Fourier Transform. Both time domain and two-sided spectrum of the input signal are shown.

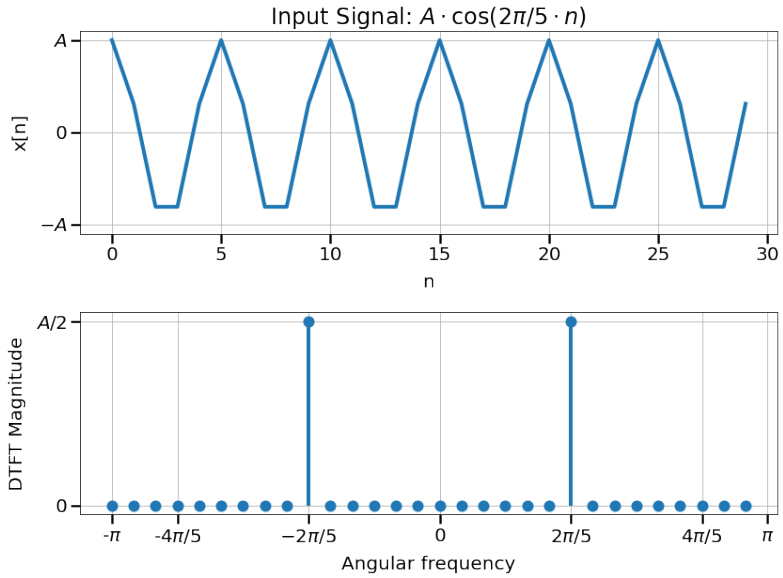


Figure 2.1: Time and frequency domain representation of input signal $x[n]$.

Then the approximation for $A = 1$ and $A = 3$ is performed around the centers $c = \{0, \ln(2)\}$. The results of the approximations are shown in Figure 2.3 and Figure 2.4 along with the true system output.

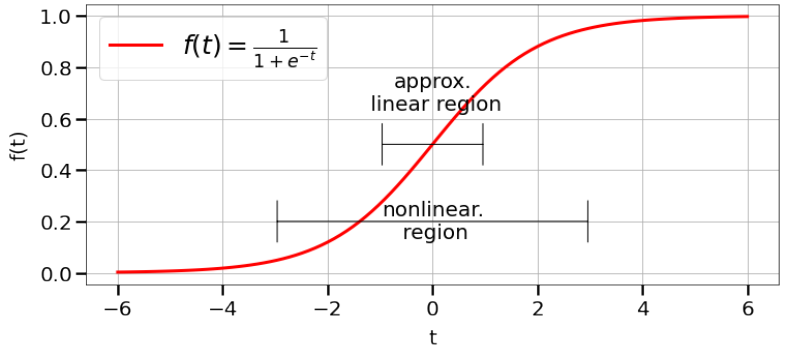


Figure 2.2: Sigmoid function showing approximately linear and nonlinear region.

Figure 2.3 shows the approximation of the system for an amplitude of $A = 1$. In this amplitude region the sigmoid function $f(t)$ can be assumed to be approximately linear as shown in Figure 2.2. The output signal now has an offset, as $f(t)$ maps the input values onto a positive range $(0, 1)$. This is also visible in the frequency domain - now the original output spectrum consists of an additional frequency bin at $\theta = 0$, representing the DC value of the signal. All other frequency components of the signal (apart of the input angular frequency $\theta = \frac{2\pi}{5}$ are close to zero. This observation confirms the assumption that in this region the sigmoid is approximately linear and the Taylor Series approximation with coefficients $\alpha_0, \ldots, \alpha_3$ works sufficiently well.

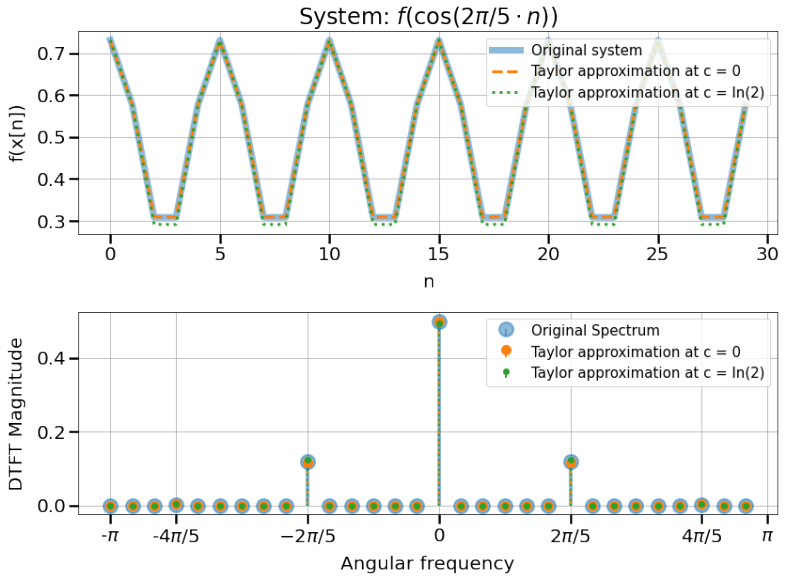


Figure 2.3: True and approximate system output for $A = 1$, approximate linear region.

The approximation is repeated with an input amplitude of $A = 3$ and its results are shown in Figure 2.4. Already in the spectrum it is visible that the sigmoid is nonlinear in this amplitude range as there are now non-zero frequency components at $\pm\frac{4\pi}{5}$. The Taylor Series approximation with coefficients $\alpha_0, \ldots, \alpha_3$ is not able to model the nonlinear behavior of the system as accurately as before in the approximately linear case.

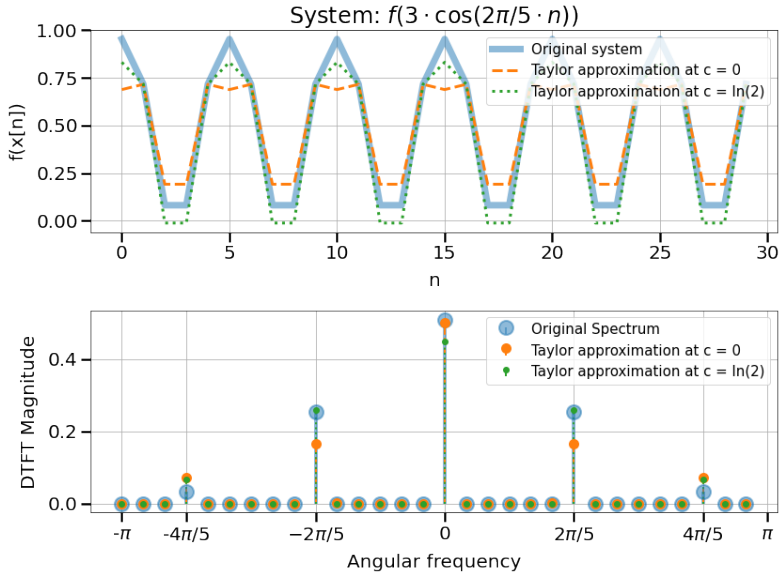


Figure 2.4: True and approximate system output for $A = 1$, nonlinear region.

Also, the influence of the center c can be observed and discussed. The approximation of the system has the same value as the true system when $x[n] = c$. For the center $c = 0$, the signal $x[n] = \cos\left(\frac{2\pi}{5}n\right) = 0$ for a value of $n = \frac{5}{4} \cdot m$ for $m = 1, 3, \ldots,$ which obviously cannot be reached as $n \in \mathbb{N}$. It is visible in Figure 2.1 showing the input signal that the value $x[n] = 0$ is never reached with a signal bin.
The same is valid for the center $c = \ln(2) = 0.6931$. The value n^* where $x[n^*] = \ln(2)$ is the index (if $n^* \in \mathbb{N}$) where the approximation is equal to the true system output. Again, looking at Figure 2.1 this value is never reached exactly by $x[n]$.

3 Statistical Analysis

3.1 Problem 3

Consider the random variable X with density

$$p_X(x) = \frac{\lambda}{2} e^{-\lambda|x|} \quad x \in [-\infty, \infty]$$

3.1.1 Task a)

Find the corresponding characteristic function $\Phi(\omega)$

$$\begin{aligned}
\phi_X\{\omega\} &= \int_{\mathbb{R}} p_X(\xi)\, e^{j\xi\omega} d\xi && (3.1)\\
&= \int_{-\infty}^{\infty} \frac{\lambda}{2} e^{-\lambda|\xi|} e^{j\xi\omega} d\xi && (3.2)\\
&= \int_{-\infty}^{0} \frac{\lambda}{2} e^{-\lambda(-\xi)} e^{j\xi\omega} d\xi + \int_{0}^{\infty} \frac{\lambda}{2} e^{-\lambda\xi} e^{j\xi\omega} d\xi && (3.3)\\
&= \frac{\lambda}{2}\left[\int_{-\infty}^{0} e^{\xi(\lambda+j\omega)} d\xi + \int_{0}^{\infty} e^{-\xi(\lambda-j\omega)} d\xi\right] && (3.4)\\
&= \frac{\lambda}{2}\left[\frac{1}{\lambda+j\omega}\ e^{\xi(\lambda+j\omega)}\Big|_{-\infty}^{0} - \frac{1}{\lambda-j\omega}\ e^{-\xi(\lambda-j\omega)}\Big|_{0}^{\infty}\right] && (3.5)\\
&= \frac{\lambda}{2}\left[\frac{1}{\lambda+j\omega} + \frac{1}{\lambda-j\omega}\right] && (3.6)\\
&= \frac{\lambda^2}{\lambda^2+\omega^2} && (3.7)
\end{aligned}$$

$$\boxed{\phi_X\{\omega\} = \frac{1}{1+\frac{\omega^2}{\lambda^2}}} \qquad \phi_X(0) = 1 \quad \checkmark \qquad |\phi_X(\omega)| \leq 1 \quad \checkmark$$

☞ Moments of the random variable X with probability density function $p_X(\xi)$:

$$\left.\frac{d\phi_X(\omega)}{d\omega}\right|_{\omega=0} = -\left.\frac{2\lambda^2\omega}{(\lambda^2+\omega^2)^2}\right|_{\omega=0} = 0$$

➥ Mean value: $\mathbb{E}(X) = \mu = 0$

$$\left.\frac{d^2\phi_X(\omega)}{d\omega^2}\right|_{\omega=0} = -\lambda^2\left(\frac{8\omega}{(\lambda^2+\omega^2)^3} - \frac{2}{(\lambda^2+\omega^2)^2}\right)\Bigg|_{\omega=0} = -\frac{2}{\lambda^2} = j^2\mathbb{E}(X^2)$$

➥ Total power: $\mathbb{E}(X^2) = \frac{2}{\lambda^2}$

$$\left.\frac{d^3\phi_X(\omega)}{d\omega^3}\right|_{\omega=0} = -\lambda^2\left(\frac{24\omega}{(\lambda^2+\omega^2)^3} - \frac{48\omega^3}{(\lambda^2+\omega^2)^4}\right)\Bigg|_{\omega=0} = 0 = j^3\mathbb{E}(X^3)$$

➥ Skewness: $\mathbb{E}(X^3) = 0$

3.1.2 Task b)

Show that all odd-order moments of X are zero.

$$\mathbb{E}(X^n) = \int_{-\infty}^{\infty} \xi^n p_X(\xi) d\xi \tag{3.8}$$

$$= \int_{-\infty}^{0} \xi^n p_X(\xi) d\xi + \int_{0}^{\infty} \xi^n p_X(\xi) d\xi \tag{3.9}$$

$$= -\int_{0}^{-\infty} \xi^n p_X(\xi) d\xi + \int_{0}^{\infty} \xi^n p_X(\xi) d\xi \tag{3.10}$$

$$\text{Let } u = -\xi \quad \Rightarrow d\xi = -du \quad \text{and} \quad p_X(\xi) = p_X(-\xi) \text{ (even symmetry)} \tag{3.11}$$

$$\mathbb{E}(X^n) = -\int_{0}^{\infty} (-u)^n p_X(-u)(-du) + \int_{0}^{\infty} \xi^n p_X(\xi) d\xi \tag{3.12}$$

$$= \int_{0}^{\infty} (-\xi)^n p_X(\xi) d\xi + \int_{0}^{\infty} \xi^n p_X(\xi) d\xi \tag{3.13}$$

$$\text{For } n \in \{1, 3, 5, \dots\} \;\Rightarrow\; (-\xi)^n = -(\xi)^n \tag{3.14}$$

$$\mathbb{E}(X^n) = -\int_{0}^{\infty} (\xi)^n p_X(\xi) d\xi + \int_{0}^{\infty} \xi^n p_X(\xi) d\xi = 0 \tag{3.15}$$

3.2 Problem 4

Let X and Y be random variables that take values in $\mathcal{X}$ and $\mathcal{Y}$, respectively. Let $Z = X + Y$, and assume that X and Y are statistically independent, i.e. their joint distribution factorizes as $p_{X,Y}(x, y) = p_X(x) \cdot p_Y(y)$

3.2.1 Task a)

Version 1

If two random variables X and Y are independent, the joint probability distribution can be factorized.

$$p_{XY}(x, y) = p_X(x) \cdot p_Y(y) \tag{3.16}$$

In order to obtain the marginal probability distribution p_Y, the joint probability distribution is integrated over the support $\mathcal{X}$:

$$p_Y(y) = \int_{x \in \mathcal{X}} p_{XY}(x, y) dx \tag{3.17}$$

A realization $x \in \mathcal{X}$ of the random variable X is fixed and the change of variable formula is used for the function $Z = g(X, Y) = X + Y$. With the inverse $Y = g^{-1}(X, Z) = Z - X$, the probability density of Z can be determined as the following

$$p_Z(z) = \int_{x \in \mathcal{X}} p_{XY}(x, z - x) \left| \frac{d}{dz} g^{-1}(x, z) \right| dx \tag{3.18}$$

$$= \int_{x \in \mathcal{X}} p_{XY}(x, z - x) \cdot 1 \; dx \tag{3.19}$$

$$= \int_{x \in \mathcal{X}} p_X(x) p_Y(z - x) dx \tag{3.20}$$

$$= p_X \star p_Y \tag{3.21}$$

or equivalently for a fixed $y \in \mathcal{Y}$:

$$p_Z(z) = \int_{y \in \mathcal{Y}} p_Y(y) p_Y(z - y) dy \tag{3.22}$$

$$= p_X \star p_Y \tag{3.23}$$

Version 2

The cumulative distribution function (CDF) of a random variable X is defined as an integral of the probability density function (PDF):

$$P_X(x) = \int_{-\infty}^{x} p_X(\tilde{x}) d\tilde{x} \tag{3.24}$$

In our case, the CDF can be expressed using the given joint probability distribution $p_{X,Y}$. The boundary of the integral is determined using the change of variable formula for the function Z = X + Y, while fixing a realization of X ($x \in \mathcal{X}$). With the inverse $Y = g^{-1}(X, Z) = Z - X$, the cumulative distribution function can be expressed as follows.

$$P_Z(z) = \int_{-\infty}^{z-x} \int_{-\infty}^{\infty} p_{X,Y}(x, y)\ dxdy \tag{3.25}$$

$$= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \Theta\left((z - x) - y\right) p_{X,Y}(x, y)\ dxdy \tag{3.26}$$

In Equation 3.26, the Heaviside-step function $\Theta(y)$ is introduced to expand the integration over the full domain $\mathcal{D}$. The function is defined as:

$$\Theta(y) = \begin{cases} 0 & \text{if } y < 0 \\ 1 & \text{if } y \geq 0 \end{cases} \tag{3.27}$$

Inverting equation 3.24 allows to compute the PDF by deriving the CDF with respect to the variable z:

$$p_Z(z) = \frac{\partial}{\partial z} P_Z(z) = \frac{\partial}{\partial z} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \Theta\left((z - x) - y\right) p_{X,Y}(x, y)\ dxdy \tag{3.28}$$

$$= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \frac{\partial}{\partial z} \Theta\left((z - x) - y\right) p_{X,Y}(x, y)\ dxdy \tag{3.29}$$

$$= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \delta\left((z - x) - y\right) p_{X,Y}(x, y)\ dxdy \tag{3.30}$$

$$= \int_{-\infty}^{\infty} p_{X,Y}(x, z - x)\ dx \tag{3.31}$$

$$= \int_{-\infty}^{\infty} p_X(x) p_Y(z - x)\ dx \tag{3.32}$$

$$= p_X \star p_Y \tag{3.33}$$

Which proves that the PDF of Z is the convolution of the PDF of X and Y.

3.2.2 Task b)

Version 1

Since the probability density function of the random variable $Z = X + Y$ is given as the convolution $p_Z = p_X \star p_Y$, the characteristic function (Fourier transform of the pdf) is the multiplication.

$$\phi_Z = \phi_{X+Y} = \phi_X \cdot \phi_Y \tag{3.34}$$

$$\ln(\phi_Z) = \ln(\phi_X) + \ln(\phi_Y) \tag{3.35}$$

Taking the derivative and setting the variable ω to zero, the cumulants of the sum $Z = X + Y$ are given as the sum of the individual cumulants of X and Y.

$$\frac{d^p}{d(j\omega)^p}\ln(\phi_{X+Y}(\omega))\bigg|_{\omega=0} = \frac{d^p}{d(j\omega)^p}\left[\ln(\phi_X(\omega)) + \ln(\phi_Y(\omega))\right]\bigg|_{\omega=0} \tag{3.36}$$

$$= \frac{d^p}{d(j\omega)^p}\ln(\phi_X(\omega)) + \frac{d^p}{d(j\omega)^p}\ln(\phi_Y(\omega))\bigg|_{\omega=0} \tag{3.37}$$

☞ $$c_p^Z = c_p^{X+Y} = c_p^X + c_p^Y \tag{3.38}$$

Version 2

The relation can be proved using the definition of the characteristic function, as well as the fact that Z = X+Y:

$$c_p^Z = \log\big(\mathbb{E}\{e^{j\omega z}\}\big) \tag{3.39}$$

$$= \log\Big(\mathbb{E}\{e^{j\omega(x+y)}\}\Big) \tag{3.40}$$

$$= \log\big(\mathbb{E}\{e^{j\omega x}\cdot e^{j\omega y}\}\big) \tag{3.41}$$

$$= \log\Big(\mathbb{E}\{e^{j\omega x)}\Big) + \log\Big(\mathbb{E}\{e^{j\omega y)}\}\Big) \quad \text{as X and Y are statistically independent} \tag{3.42}$$

$$= c_p^X + c_p^Y \quad \square \tag{3.43}$$

4 Propagation of Uncertainty and Normalizing Flows

4.1 Problem 5

In the context of an indoor localization problem, the position of an agent should be estimated in a certain area around a base station. The agent is located at position $\mathbf{z} = (x, y)^T$, and transmits a signal to the base station at $\mathbf{z_0} = (0, 0)^T$. The base station estimates the polar coordinates $\boldsymbol{\omega}$ of the agent using a nonlinear transform $f(\mathbf{z})$. Zero-mean, additive noise $\boldsymbol{\nu} \sim p_{\boldsymbol{\nu}}$ is used to model the uncertainty of the measurement process:

$$\boldsymbol{\omega} = f(\mathbf{z}) = \begin{pmatrix} \sqrt{x^2 + y^2} \\ \arctan y/x \end{pmatrix} \tag{4.1}$$

$$\hat{\boldsymbol{\omega}} = \boldsymbol{\omega} + \boldsymbol{\nu} = f(\mathbf{z}) + \boldsymbol{\nu} \tag{4.2}$$

To estimate the position of the agent, the inverse transform $f^{-1}(\hat{\boldsymbol{\omega}})$ is applied:

$$\hat{\mathbf{z}} = f^{-1}(\hat{\boldsymbol{\omega}}) = \begin{pmatrix} \hat{r} \cdot \cos \hat{\phi} \\ \hat{r} \cdot \sin \hat{\phi} \end{pmatrix} \tag{4.3}$$

Remarks: Notation

Let X and Y be two continuous random variables taking values in $\mathcal{X}$ and $\mathcal{Y}$, then the joint probability density function is denoted as $p_{X,Y}(x, y)$.

The marginal distributions can be computed from the joint distribution as follows.

$$p_X(x) = \int_{\mathcal{Y}} p_{X,Y}(x, y)dy \qquad\qquad p_Y(y) = \int_{\mathcal{X}} p_{X,Y}(x, y)dx$$

The conditional distribution of Y given $X = x$ is then defined as

$$p_{Y,X=x}(y) = \frac{p_{X,Y}(x, y)}{p_X(x)} \ .$$

According to the assignment sheet, the conditional distribution is also denoted as $p_{y|x}$ with no distinction between random variable X and realization x with respect to capital and small letters. Therefore, the following notation for the conditional density is found in our assignment (although it is noted that there is in theory a difference between X and x).

$$p_{Y,X=x}(y) = p_{y|x} = p(y|x) = p_y(y|x)$$

4.1.1 Task a)

Derive an expression for the conditional density $p_{\hat{\boldsymbol{\omega}}|\mathbf{z}}$ of the measurements $\hat{\boldsymbol{\omega}}$ when observing the true position $\mathbf{z}$.

The conditional probability can be found by first computing $p(\hat{\boldsymbol{\omega}}|\mathbf{z}, \boldsymbol{\nu})$, and a subsequent marginalization over the probability density of $\boldsymbol{\nu}$.

$p(\hat{\boldsymbol{\omega}}|\mathbf{z}, \boldsymbol{\nu})$ is hereby found by identifying that a specific $\hat{\boldsymbol{\omega}}$ for a given $\mathbf{z}$ and $\boldsymbol{\nu}$ can only be reached if the following relation holds:

$$\hat{\boldsymbol{\omega}} = f(\mathbf{z}) + \boldsymbol{\nu} \tag{4.4}$$

Therefore, the only possible distribution for $p(\hat{\boldsymbol{\omega}}|\mathbf{z}, \boldsymbol{\nu})$ is a delta-distribution, peaked at $f(\mathbf{z})+\boldsymbol{\nu}$:

$$p(\hat{\boldsymbol{\omega}}|\mathbf{z}, \boldsymbol{\nu}) = \delta(\hat{\boldsymbol{\omega}} - f(\mathbf{z}) - \boldsymbol{\nu}) \tag{4.5}$$

Following from this expression, the conditional density can be easily computed:

$$\begin{aligned} p(\hat{\boldsymbol{\omega}}|\mathbf{z}) &= \int_{\mathcal{D}} p(\hat{\boldsymbol{\omega}}|\mathbf{z}, \boldsymbol{\nu}) \cdot p(\boldsymbol{\nu}|\mathbf{z}) \; d\nu && (4.6)\\ &= \int_{\mathcal{D}} \delta(\hat{\boldsymbol{\omega}} - f(\mathbf{z}) - \boldsymbol{\nu}) \cdot p(\boldsymbol{\nu}|\mathbf{z}) \; d\nu && (4.7)\\ &= p_{\boldsymbol{\nu}}(\hat{\boldsymbol{\omega}} - f(\mathbf{z})|\mathbf{z}) && (4.8) \end{aligned}$$

This derivation is valid for the general case in which the noise $\boldsymbol{\nu}$ can be correlated with the true position $\mathbf{z}$. Assuming, that the noise and true position are independent, the following holds:

$$p_{\boldsymbol{\nu}}(\hat{\boldsymbol{\omega}} - f(\mathbf{z})|\mathbf{z}) = p_{\boldsymbol{\nu}}(\hat{\boldsymbol{\omega}} - f(\mathbf{z})) = p_{\boldsymbol{\nu}}(\boldsymbol{\nu}) \tag{4.9}$$

4.1.2 Task b)

Compute the conditional expectation $\mathbb{E}_{\hat{\boldsymbol{\omega}}|\mathbf{z}}(\hat{\boldsymbol{\omega}})$ of the measurements for a given true position. Is this an unbiased estimate for the true polar coordinates $\boldsymbol{\omega}$?

$$\begin{aligned} \mathbb{E}_{\hat{\boldsymbol{\omega}}|\mathbf{z}}(\hat{\boldsymbol{\omega}}) &= \int_{\mathbb{D}} \hat{\boldsymbol{\omega}} \; p(\hat{\boldsymbol{\omega}}|\mathbf{z}) \; d\hat{\boldsymbol{\omega}} && (4.10)\\ &= \int_{\mathbb{D}} (f(\mathbf{z}) + \boldsymbol{\nu}) \; p(\hat{\boldsymbol{\omega}}|\mathbf{z}) \; d\hat{\boldsymbol{\omega}} && (4.11)\\ &= \int_{\mathbb{D}} f(\mathbf{z}) \; p(\hat{\boldsymbol{\omega}}|\mathbf{z}) d\hat{\boldsymbol{\omega}} + \int_{\mathbb{D}} \boldsymbol{\nu} \; p(\hat{\boldsymbol{\omega}}|\mathbf{z}) \; d\hat{\boldsymbol{\omega}} && (4.12)\\ &= \int_{\mathbb{D}} f(\mathbf{z}) \; p_{\boldsymbol{\nu}}(\underbrace{\hat{\boldsymbol{\omega}} - f(\mathbf{z})}_{\boldsymbol{\nu}}) \; d\hat{\boldsymbol{\omega}} + \int_{\mathbb{D}} \boldsymbol{\nu} \; p_{\boldsymbol{\nu}}(\underbrace{\hat{\boldsymbol{\omega}} - f(\mathbf{z})}_{\boldsymbol{\nu}}) \; d\hat{\boldsymbol{\omega}}, \quad \text{and } d\hat{\boldsymbol{\omega}} = d\boldsymbol{\nu} && (4.13)\\ &= \int_{\mathbb{D}} f(\mathbf{z}) \; p_{\boldsymbol{\nu}}(\boldsymbol{\nu}) \; d\boldsymbol{\nu} + \cancel{\int_{\mathbb{D}} \boldsymbol{\nu} \; p_{\boldsymbol{\nu}}(\boldsymbol{\nu}) \; d\boldsymbol{\nu}} && (4.14)\\ &= f(\mathbf{z}) \underbrace{\int_{\mathbb{D}} p_{\boldsymbol{\nu}}(\boldsymbol{\nu}) \; d\boldsymbol{\nu}}_{=1} && (4.15)\\ &= f(\mathbf{z}) \quad \square && (4.16) \end{aligned}$$

The estimator is unbiased for zero-mean additive noise.

4.1.3 Task c)

Compute the conditional expectation $\mathbb{E}_{\hat{\boldsymbol{\omega}}|\mathbf{z}}(f^{-1}(\hat{\boldsymbol{\omega}}))$, assuming independently distributed components of the noise vector:

$$\boldsymbol{\nu} = (\nu_r \ \nu_\phi)^T \quad \text{with } \nu_r \sim p_{\nu_r}, \nu_\phi \sim p_{\nu_\phi} \text{ and } p_{\boldsymbol{\nu}} = p_{\nu_r} \cdot p_{\nu_\phi} \tag{4.17}$$

Is this an unbiased estimate for the true coordinates $\mathbf{z} = f^{-1}(\boldsymbol{\omega})$?

The following box shows the occurring variables, and functions, as their dimensionalitys might not be evident in this problem:

Dimensionalitys of the variables

$\hat{\boldsymbol{\omega}} \in \mathbb{R}^2$ $\qquad$ $f^{-1}(\hat{\boldsymbol{\omega}}) : \mathbb{R}^2 \to \mathbb{R}^2$ $\qquad$ $\mathbb{E}_{\hat{\boldsymbol{\omega}}|\mathbf{z}}\big(f^{-1}(\hat{\boldsymbol{\omega}})\big) \in \mathbb{R}^2$

As indicated, the conditional expectation is $\in \mathbb{R}^2$, hence a component-wise consideration of the problem becomes necessary. Therefore, each component of Equation 4.3 is integrated independently over the given noise distribution $p_{\boldsymbol{\nu}}$.

$$\mathbb{E}_{\hat{\boldsymbol{\omega}}|\mathbf{z}}(f^{-1}(\hat{\boldsymbol{\omega}})_{\mathbb{I}}) \equiv \int_{\mathbb{D}}\int \hat{r} \cdot \cos\hat{\phi} \ p_{\nu_r} \cdot p_{\nu_\phi} d\nu_r d\nu_\phi \tag{4.18}$$

$$\mathbb{E}_{\hat{\boldsymbol{\omega}}|\mathbf{z}}(f^{-1}(\hat{\boldsymbol{\omega}})_{\mathbb{II}}) \equiv \int_{\mathbb{D}}\int \hat{r} \cdot \sin\hat{\phi} \ p_{\nu_r} \cdot p_{\nu_\phi} d\nu_r d\nu_\phi \tag{4.19}$$

The computation is done using $\mathbb{E}_{\hat{\boldsymbol{\omega}}|\mathbf{z}}(f^{-1}(\hat{\boldsymbol{\omega}})_{\mathbb{I}})$, and can be identically repeated for the second component.

$$\mathbb{E}_{\hat{\boldsymbol{\omega}}|\mathbf{z}}(f^{-1}(\hat{\boldsymbol{\omega}})_{\mathbb{I}}) = \int_{\mathbb{D}}\int \hat{r} \cdot \cos\hat{\phi} \ p_{\nu_r} \cdot p_{\nu_\phi} d\nu_r d\nu_\phi, \quad \hat{r} = r + \nu_r \text{ and } \hat{\phi} = \phi + \nu_\phi \tag{4.20}$$

$$= \int_{\mathbb{D}}\int (r + \nu_r) \cdot \cos(\phi + \nu_\phi) \ p_{\nu_r} \cdot p_{\nu_\phi} d\nu_r d\nu_\phi \tag{4.21}$$

$$= \int_{\mathbb{D}}\int (r + \nu_r) \cdot [\cos\phi\cos\nu_\phi - \sin\phi\sin\nu_\phi] \ p_{\nu_r} \cdot p_{\nu_\phi} d\nu_r d\nu_\phi \tag{4.22}$$

$$= \int_{\mathbb{D}}\int \Big(r \cdot [\cos\phi\cos\nu_\phi - \sin\phi\sin\nu_\phi] \tag{4.23}$$

$$+ \nu_r \cdot [\cos\phi\cos\nu_\phi - \sin\phi\sin\nu_\phi] \Big) \ \cancel{p_{\nu_r}} \cdot p_{\nu_\phi} \cancel{d\nu_r} d\nu_\phi \tag{4.24}$$

$$= \int_{\mathbb{D}_\phi} r \cdot [\cos\phi\cos\nu_\phi - \sin\phi\sin\nu_\phi] \cdot p_{\nu_\phi} \ d\nu_\phi \tag{4.25}$$

$$= \int_{\mathbb{D}_\phi} r \cdot \cos\phi\cos\nu_\phi \cdot p_{\nu_\phi} \ d\nu_\phi - \int_{\mathbb{D}_\phi} r \cdot \sin\phi\sin\nu_\phi \cdot p_{\nu_\phi} \ d\nu_\phi \tag{4.26}$$

$$= r \cdot \cos\phi \cdot \mathbb{E}(\cos\nu_\phi) - r \cdot \sin\phi \cdot \mathbb{E}(\sin\nu_\phi) \tag{4.27}$$

The conditional expectation would be only for $\mathbb{E}(\cos\nu_\phi) = 1$, and $\mathbb{E}(\sin\nu_\phi) = 0$ unbiased. It can be very easily shown, that this is not generally the case by expanding the sine, and cosine function using a Taylor series:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots \quad \sin(x) = x - \frac{x^3}{3!} + \frac{x^4}{5!} - \frac{x^7}{7!} + \dots \tag{4.28}$$

Therefore:

$$\mathbb{E}(\cos\nu_\phi) = \mathbb{E}\big(1 - \frac{\nu_\phi^2}{2!} + \frac{\nu_\phi^4}{4!} - \frac{\nu_\phi^6}{6!} + \dots\big) \tag{4.29}$$

$$= 1 - \frac{1}{2}\mathbb{E}(\nu_\phi^2) + \frac{1}{4!}\mathbb{E}(\nu_\phi^4) - \frac{1}{6!}\mathbb{E}(\nu_\phi^6) + \dots \tag{4.30}$$

$$\mathbb{E}\big(\sin\nu_\phi\big) = \mathbb{E}(\nu_\phi - \frac{\nu_\phi^3}{3!} + \frac{\nu_\phi^5}{5!} - \frac{\nu_\phi^7}{7!} + \dots\big) \tag{4.31}$$

$$= \underbrace{\cancel{\mathbb{E}(\nu_\phi)}}_{\text{zero mean}} - \frac{1}{3!}\mathbb{E}(\nu_\phi^3) + \frac{1}{5!}\mathbb{E}(\nu_\phi^5) - \frac{1}{7!}\mathbb{E}(\nu_\phi^7) + \dots \tag{4.32}$$

☞ All higher order moments of p_{ν_ϕ} must vanish for an unbiased estimation result!

4.1.4 Task d)

Assume you where given multiple i.i.d. measurements $\{\hat{\boldsymbol{\omega}}_n\}_{n=1}^N$ for the same agent position $\mathbf{z}$, and two different estimators $\hat{\mathbf{z}}_\mathbf{A}$ and $\hat{\mathbf{z}}_\mathbf{B}$:

$$\hat{\mathbf{z}}_\mathbf{A} = f^{-1}\Big(\frac{1}{N}\sum_n \hat{\boldsymbol{\omega}}_n\Big) \qquad \hat{\mathbf{z}}_\mathbf{B} = \frac{1}{N}\sum_n f^{-1}(\hat{\boldsymbol{\omega}}_n) \tag{4.33}$$

Which estimator should be chosen and why?

To solve this task, the results of task b) and c) are used:

$$\hat{\mathbf{z}}_\mathbf{A} = f^{-1}\Big(\frac{1}{N}\sum_n \hat{\boldsymbol{\omega}}_n\Big) \qquad \hat{\mathbf{z}}_\mathbf{B} = \frac{1}{N}\sum_n f^{-1}(\hat{\boldsymbol{\omega}}_n) \tag{4.34}$$

$$\xrightarrow{N\to\infty} f^{-1}\Big(\mathbb{E}(\hat{\boldsymbol{\omega}}_n)\Big) = f^{-1}\Big(\mathbb{E}(\boldsymbol{\omega}_n + \boldsymbol{\nu}_n)\Big) \qquad \xrightarrow{N\to\infty} \mathbb{E}\big(f^{-1}(\hat{\boldsymbol{\omega}})\big) \tag{4.35}$$

$$= f^{-1}\Big(\mathbb{E}(\boldsymbol{\omega}_n) + \mathbb{E}(\boldsymbol{\nu}_n)\Big) \qquad \neq \mathbf{z} \tag{4.36}$$

$$= f^{-1}(\boldsymbol{\omega} + 0) = f^{-1}\big(f(\mathbf{z})\big) = \mathbf{z} \tag{4.37}$$

☞ Estimator $\hat{\mathbf{z}}_\mathbf{A}$ should be chosen, as it deliveries unbiased estimation results in our case (estimator is asymptotically unbiased).

To archive this result, the mean value of the additive noise must vanish, thus $\mathbb{E}(\boldsymbol{\nu}_n) = \mu_{\boldsymbol{\nu}_n} \overset{!}{=} 0$. Estimator $\hat{\mathbf{z}}_\mathbf{B}$ is generally biased, as previously shown in subsection 4.1.3.

Again, our results can be verified numerically. Noisy position measurements are simulated with additive zero-mean Gaussian noise with unit variance.

First, we analyze the behavior of estimator $\hat{\mathbf{z}}_\mathbf{A}$. The noisy measurements for distance $\hat{r}$ and angle $\hat{\phi}$ are plotted as a histogram in Figure 4.1. The weighted sum in the equation for $\hat{\mathbf{z}}_\mathbf{A}$ is obtained by taking the mean value of all samples. As indicated in the figure, the mean value corresponds to the true value of the measurement for N large enough. Therefore, the inverse $f^{-1}(.)$ leads to accurate estimation results.

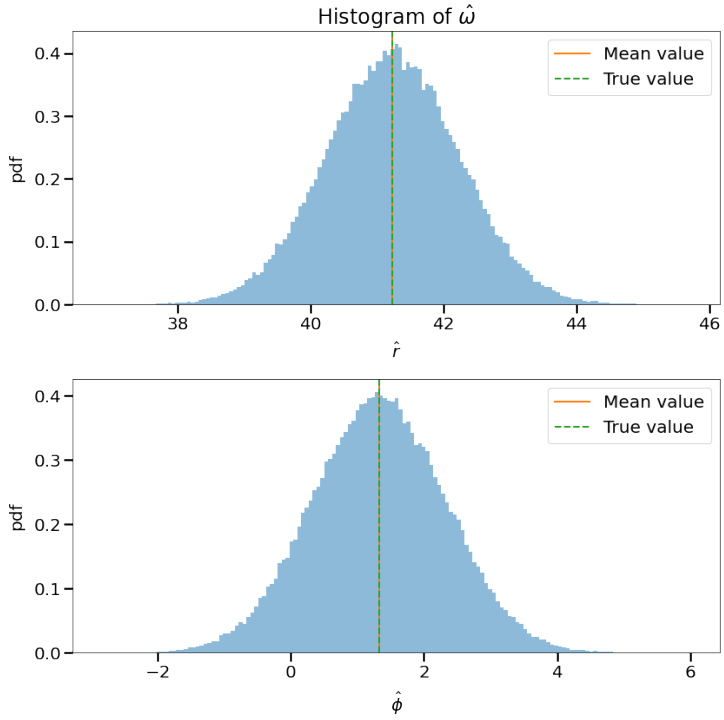


Figure 4.1: Histogram of samples $\hat{\boldsymbol{\omega}}_n$ for estimator $\hat{\mathbf{z}}_\mathbf{A}$

Next, the estimation with estimator $\hat{\mathbf{z}}_\mathbf{B}$ is performed. Noisy distance and angle measurements are generated, transformed by f^{-1} and plotted in a histogram in Figure 4.2. The mean value is computed and shown alongside the true value for both elements of $\hat{\mathbf{z}}$. It is clearly visible that the mean value deviates much from the true value - the estimator is biased.

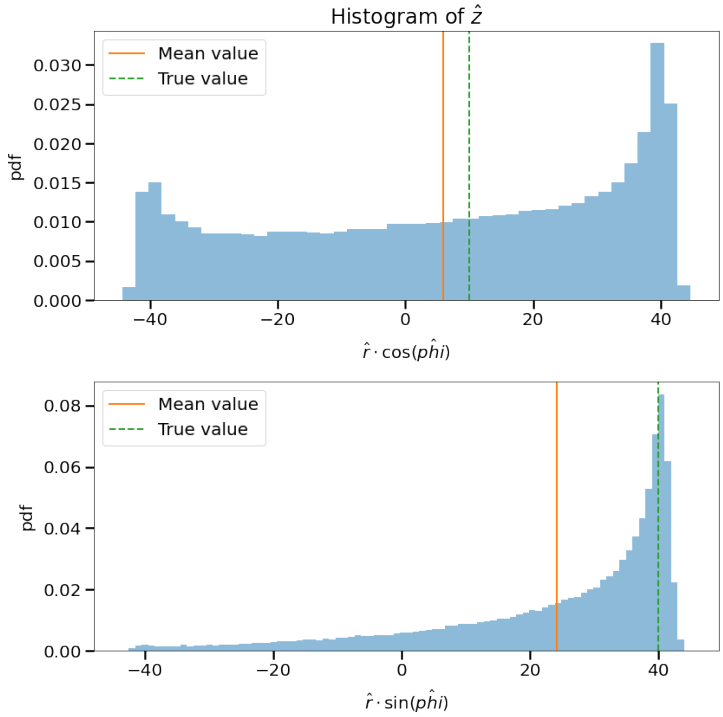


Figure 4.2: Histogram of samples $f^{-1}(\hat{\boldsymbol{\omega}}_n)$ for estimator $\hat{\mathbf{z}}_\mathbf{B}$

4.2 Problem 6

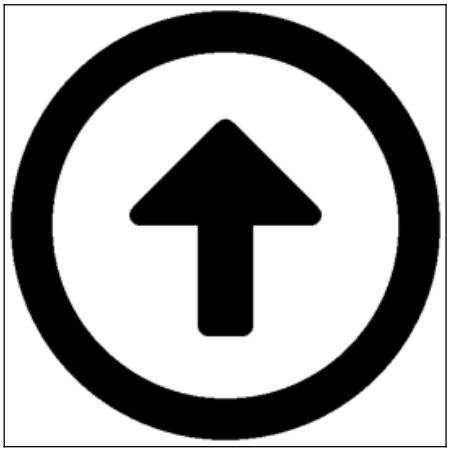
Normalizing flows are generative models that allow efficient sampling and density estimation. They can be used for image generation, which is in this problem for a very simple binary image. In our example, bivariate distributions are used to allow easy visualization.

In this task, the normalizing flows do not have to be implemented from scratch. **Pyro** and **Pytorch** are used to construct the normalizing flows. In a first step, all necessary libraries and modules are imported.

[1]:
```
%reload_ext autoreload
%autoreload 2
%matplotlib inline

import warnings
warnings.filterwarnings('ignore')

from IPython.display import set_matplotlib_formats
set_matplotlib_formats('png', 'pdf')

import matplotlib.pyplot as plt
import numpy as np
import pyro.distributions as dist
import torch
from matplotlib import cm
from src.utils.io import load_image, img_to_bw

import pyro
import pyro.distributions as dist
import pyro.distributions.transforms as T

from src.utils.plotting import init_plot_style, show_grayscale_img
#%pylab

init_plot_style()
data_dir='../../data/img/'
```

Data preparation If necessary, we first have to convert our grayscale icons to a binary image.

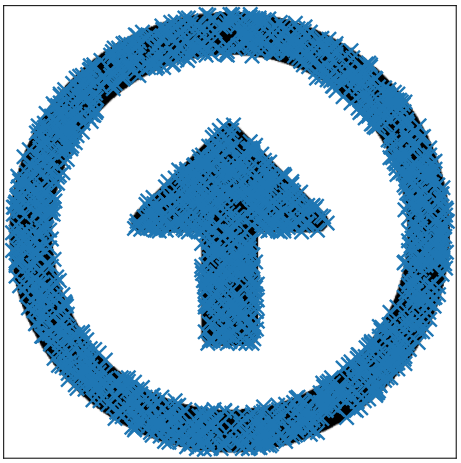
[2]:
```
img = img_to_bw(data_dir + 'up.png', data_dir + 'up_bw.png')
show_grayscale_img(img,figsize=(5, 5))
```



Now we can load the binary image and generate samples from it as described in the assignment sheet.

[3]:
```
# load image
img = load_image(data_dir + 'up_bw.png')
height, width = img.shape
print(f'Our image format is {width} x {height}.')

# generate training samples
n_samples = 2000 # number of samples to generate
data = np.zeros((n_samples, 2))
rng = np.random.default_rng(seed=0)
i=0
while i < n_samples:
    row = rng.integers(0, height)
    col = rng.integers(0, width)
    if img[row, col] == 0:
        data[i,0] = col
        data[i,1] = row
        i += 1

# plot image with generated samples
show_grayscale_img(img,figsize=(7, 7))
plt.plot( data[:,0], data[:,1], 'x')
plt.xlim([0, width])
_ = plt.ylim([0, height])
```

```
Our image format is 512 x 512.
```



Pre-processing

Our image format is 512x512 px. But in order to further process the samples of the image efficiently using normalizing flows, the samples are normalized to [-1,1].

The normalizing flow is implemented using Pyro (see this Tutorial). For the coupling transform `SplineCoupling` is used. The input parameter *bound=K* determines the bounding box ([−K,K]×[−K,K]) of the spline - the default value is K=3.

The training process of the normalizing flow works efficiently when the input is normalized to [-1,1].

[4]:
```
data_normalized = (data/512-0.5)*2
print(f'Our pre-processed samples lie in the range [{np.
 ↪min(data_normalized)} , {np.max(data_normalized)}].')
```

```
Our pre-processed samples lie in the range [-0.9609375 , 0.9609375].
```

4.2.1 Task a) Implement a coupling flow (spline coupling flow)

First, the base distribution is specified. In our case it is chosen as a random variable X with the distribution of a simple multivariate Gaussian with zero mean and unit covariance matrix:

$$\mathbf{X} \sim \mathcal{N}(\mu = \mathbf{0}_{2x1}, \Sigma = I_{2x2})$$

The random variable $\mathbf{X}$ is then transformed by a bijective function: $\mathbf{Y} = \mathbf{g}(\mathbf{X})$. This base distribution $p_{\mathbf{X}}(\mathbf{x})$ is transformed with the *change-of-variables* formula as such:

$$p_{\mathbf{Y}}(\mathbf{y}) = p_{\mathbf{X}}\left(\mathbf{g}^{-1}(\mathbf{y})\right) \left| \det \left(\frac{d\mathbf{g}^{-1}(\mathbf{y})}{d\mathbf{x}} \right) \right|. \tag{4.38}$$

The function $\mathbf{g}(.)$ typically is created by composition of several function $\mathbf{g}(\mathbf{x}) = \mathbf{g}_k(\mathbf{g}_{k-1}(\ldots(\mathbf{g}_1(\mathbf{x}))))$ - this allows us to gain more expressibility.

In our case it is formed by a spline coupling layer (see SplineCoupling) with the following parameters

- *input_dim* = 2
- *hidden_dim* = [input_dim * 30] * 10
- *count_bins* = 32
- *bound* = 3

which led to appropriate results.

[5]:
```
base_dist = dist.Normal(torch.zeros(2), torch.ones(2))
input_dim = 2
hidden_dims = [input_dim * 30]*10
transform_coupling = T.spline_coupling(input_dim=2, hidden_dims=hidden_dims,␣
↪count_bins=32, bound=3)
flow_dist_coupling = dist.TransformedDistribution(base_dist,␣
↪[transform_coupling])
```

Training: Then the parameters of the spline-coupling layer are learnt in a stochastic gradient descent where the log-likelihood is maximized (ML principle).

The parameters of the stochastic gradient descent are chosen as such:

- *number of samples* = 2000
- *epochs* = 2000
- *learning_rate* = 2e-2

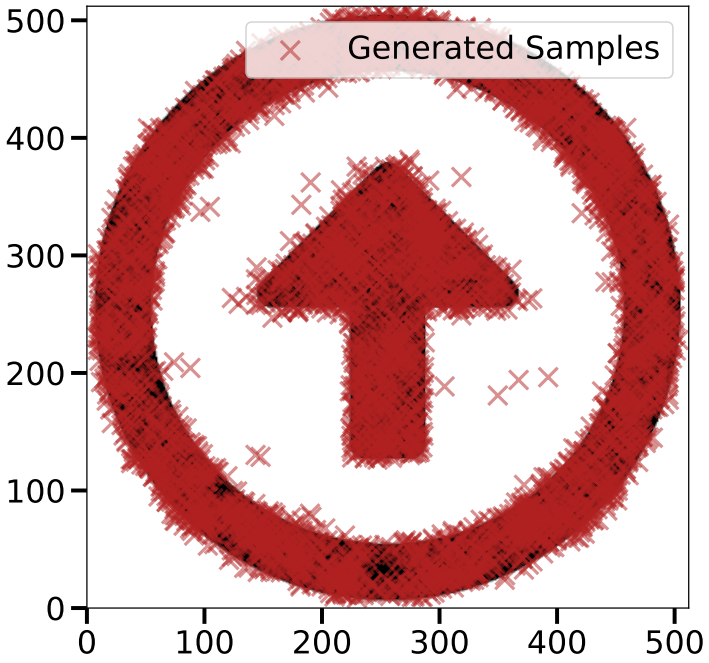
[6]:
```
%%time

steps = 2_000

dataset = torch.tensor(data_normalized, dtype=torch.float)
optimizer = torch.optim.Adam(transform_coupling.parameters(), lr=2e-2)

for step in range(steps+1):
    optimizer.zero_grad()
    loss = -flow_dist_coupling.log_prob(dataset).mean()
    loss.backward()
    optimizer.step()
    flow_dist_coupling.clear_cache()

    if step % 500 == 0:
        print('step: {}, loss: {}'.format(step, loss.item()))

print('Achieved log-likelihood: ', -loss.item())
```

```
step: 0, loss: 4.658216953277588
step: 500, loss: 0.4221729636192322
step: 1000, loss: 0.40775859355926514
step: 1500, loss: 0.37266847491264343
step: 2000, loss: 0.37579163908958435
Achieved log-likelihood:  -0.37579163908958435
Wall time: 1min 16s
```

Now that the function $\mathbf{g}(.)$ is learnt, we can sample from the trained flow distribution. In total 5000 samples are generated and shown in a scatter plot with the original b/w picture. We can observe, that the quality of the sampling process is acceptable. Most samples lie in the correct region with some small deviation.

[7]:
```
%%time

# generate samples from the learned distribution
samples_coupling = flow_dist_coupling.sample(torch.Size([5000,])).detach().
 ↪numpy()
samples_coupling = (samples_coupling/2+0.5)*512

fig, ax = plt.subplots(figsize=(7, 7))
ax.imshow(img, cmap='gray', vmin=0, vmax=255, origin='lower')
ax.scatter(samples_coupling[:,0], samples_coupling[:,1], color='firebrick',
           label='Generated Samples', alpha=0.5,  marker='x')
ax.set_xlim([0,512])
ax.set_ylim([0,512])
_ = ax.legend(loc="upper right")
```

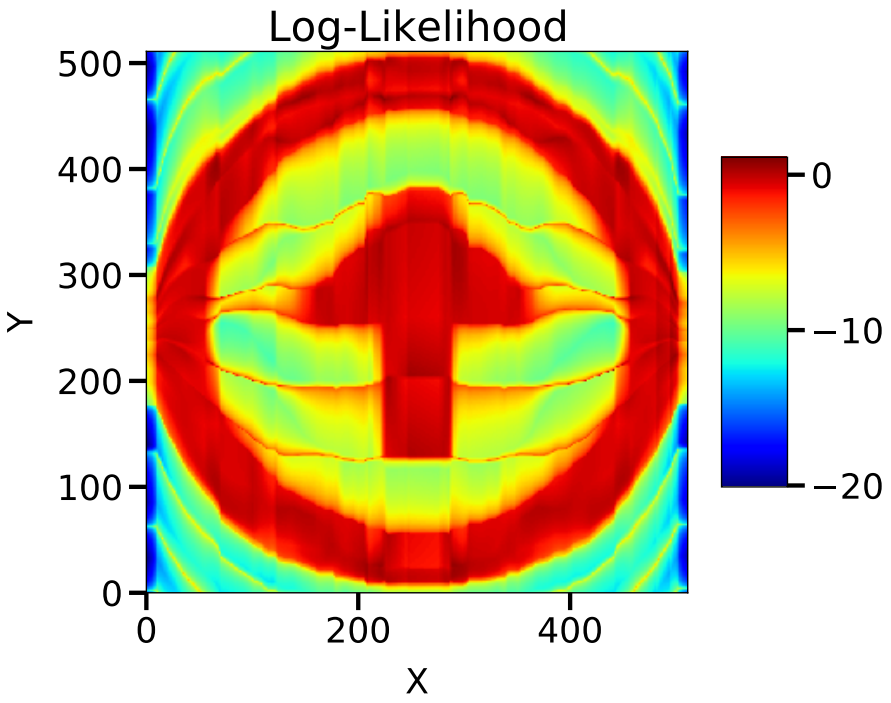
```
Wall time: 72.6 ms
```



We also want to have a look at the log-likelihood of our model!

[8]:
```
%%time

# compute the log-likelihood at a discrete grid
rows = np.arange(0, width)
cols = np.arange(0, height)

X, Y = np.meshgrid(cols, rows)

# flatten the grid points and evaluate their log-liklihoods
coordinates = torch.FloatTensor(np.stack((X,Y), axis=2)).view(-1, 2)
log_likelihood_coupling = flow_dist_coupling.log_prob((coordinates/512*2-1)).
 ↪detach().view(-1,width).numpy()

# finally, let's plot the log-likelihood
fig, ax = plt.subplots(figsize=(7, 7))
levels = np.linspace(-18, 3, 20)
c = plt.imshow(log_likelihood_coupling, cmap = 'jet',
            vmin = np.min(log_likelihood_coupling), vmax = np.
 ↪max(log_likelihood_coupling),
            extent =[X.min(), X.max(), Y.min(), Y.max()],
            interpolation ='nearest', origin ='lower')

plt.colorbar(c, shrink=0.5, aspect=5)

ax.set_xlabel('X')
ax.set_ylabel('Y')
_ = ax.set_title('Log-Likelihood')
```

```
Wall time: 1.16 s
```



4.2.2 Task b) Implement an autoregressive flow (spline autoregressive flow)

Again, the base distribution is specified as a multivariate Gaussian with zero mean and unit covariance matrix:

$$\mathbf{X} \sim \mathcal{N}(\mu = \mathbf{0}_{2x1}, \Sigma = I_{2x2})$$

The transformation of the random variable $\mathbf{X}$ with distribution $p_{\mathbf{X}}(\mathbf{x})$ by a function $\mathbf{g}(.)$ is performed by an autoregressive flow. The autoregressive layer in our solution is implemented with spline functions. In task c) the *coupling* and *autoregressive* flow are explained in more detail and compared to each other.

For the autoregressive spline layer the following parameters are specified:

- *input_dim* = 2
- *hidden_dim* = [input_dim * 30] * 10
- *count_bins* = 32
- *bound* = 3

which led to appropriate results.

[9]:
```
from pyro.nn import AutoRegressiveNN

input_dim = 2
count_bins = 32
base_dist = dist.Normal(torch.zeros(input_dim), torch.ones(input_dim))
hidden_dims = [input_dim * 30]*10
param_dims = [count_bins, count_bins, count_bins - 1, count_bins]
hypernet = AutoRegressiveNN(input_dim, hidden_dims, param_dims=param_dims)
transform_auto = T.SplineAutoregressive(input_dim, hypernet,␣
 ↪count_bins=count_bins, bound=3)
flow_dist_auto = dist.TransformedDistribution(base_dist, [transform_auto])
```

Training: Again, the parameters of the splines are learnt in a stochastic gradient descent where the log-likelihood is maximized (ML principle).

The parameters of the stochastic gradient descent are chosen the same as in task a):

- *number of samples* = 2000
- *epochs* = 2000
- *learning_rate* = 2e-2

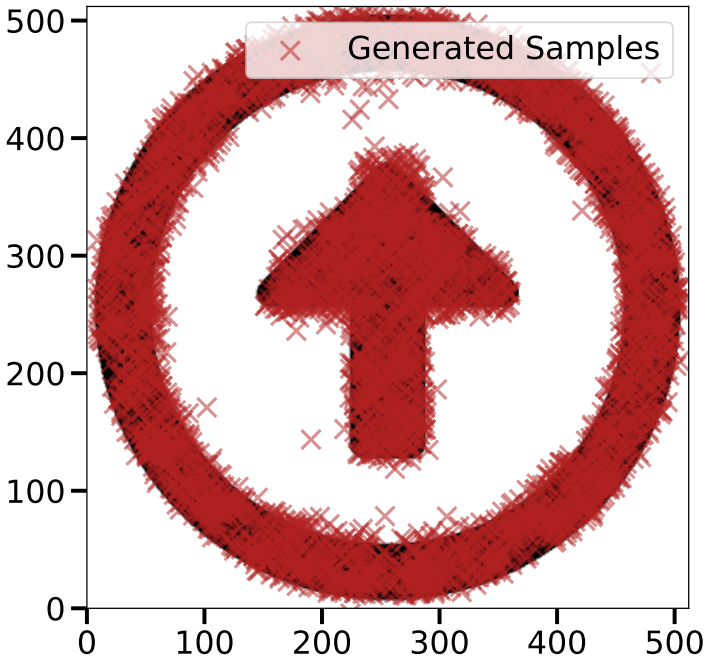
[10]:
```
%%time

steps = 2_000

dataset = torch.tensor(data_normalized, dtype=torch.float)
optimizer = torch.optim.Adam(transform_auto.parameters(), lr=2e-2)

for step in range(steps+1):
    optimizer.zero_grad()
    loss = -flow_dist_auto.log_prob(dataset).mean()
    loss.backward()
    optimizer.step()
    flow_dist_auto.clear_cache()

    if step % 500 == 0:
        print('step: {}, loss: {}'.format(step, loss.item()))

print('Achieved log-likelihood: ', -loss.item())
```

```
step: 0, loss: 2.1732373237609863
step: 500, loss: 0.43861255049705505
step: 1000, loss: 0.43714672327041626
step: 1500, loss: 0.4343864321708679
step: 2000, loss: 0.49865832924842834
Achieved log-likelihood:  -0.49865832924842834
Wall time: 2min 47s
```

Then 5000 samples are generated from the trained flow distribution and shown in a scatter plot with the original b/w picture. Also for the autoregressive flow the generated samples are quite reasonable given the simple implementation of the normalizing flow with a quite short execution time. The are some outliers, but the location of most samples does not deviate too much.

[11]:
```
%%time

# generate samples from the learned distribution
samples_auto = flow_dist_auto.sample(torch.Size([5000,])).detach().numpy()
samples_auto = (samples_auto/2+0.5)*512

fig, ax = plt.subplots(figsize=(7, 7))
ax.imshow(img, cmap='gray', vmin=0, vmax=255, origin='lower')
ax.scatter(samples_auto[:,0], samples_auto[:,1], color='firebrick',␣
 ↪label='Generated Samples', alpha=0.5,  marker='x')
ax.set_xlim([0,512])
ax.set_ylim([0,512])
_ = ax.legend(loc="upper right")
```

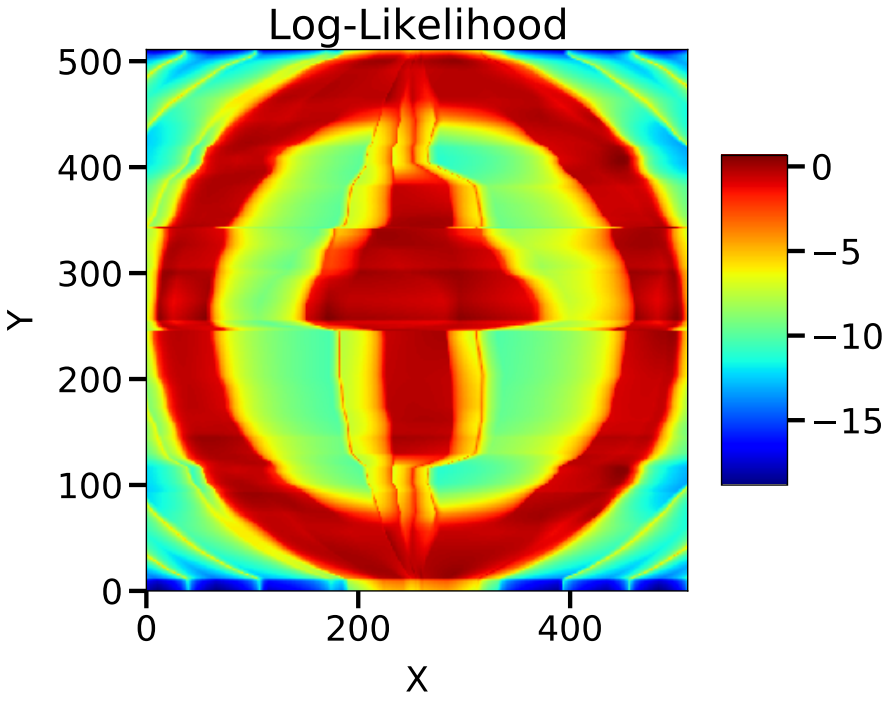
```
Wall time: 69.4 ms
```



We also want to have a look at the log-likelihood of our model!

[12]:
```
%%time

# compute the log-likelihood at a discrete grid
rows = np.arange(0, width)
cols = np.arange(0, height)

X, Y = np.meshgrid(cols, rows)

# flatten the grid points and evaluate their log-liklihoods
coordinates = torch.FloatTensor(np.stack((X,Y), axis=2)).view(-1, 2)
log_likelihood_auto = flow_dist_auto.log_prob((coordinates/512*2-1)).
 ↪detach().view(-1,width).numpy()

# finally, let's plot the log-likelihood
fig, ax = plt.subplots(figsize=(7, 7))
levels = np.linspace(-18, 3, 20)
c = plt.imshow(log_likelihood_auto, cmap = 'jet', vmin = np.
 ↪min(log_likelihood_auto), vmax = np.max(log_likelihood_auto),
            extent =[X.min(), X.max(), Y.min(), Y.max()],
            interpolation ='nearest', origin ='lower')
plt.colorbar(c, shrink=0.5, aspect=5)

ax.set_xlabel('X')
ax.set_ylabel('Y')
_ = ax.set_title('Log-Likelihood')
```

```
Wall time: 6.88 s
```



4.2.3 Task c) Comparison: coupling vs autoregressive flow

As mentioned earlier, normalizing flows make use of the *change-of-variables* formula in order to approximate a probability distribution given some samples drawn from it. Starting point is a simple base distribution, e.g. a zero-mean, unit variance Gaussian distribution $\mathbf{X} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ that can be generated easily. This random variable is then transformed by a bijective function $\mathbf{g}(.)$ with an inverse $\mathbf{f}(.)$ that is also differentiable.

$$p_{\mathbf{Y}}(\mathbf{y}) = p_{\mathbf{X}}\left(\mathbf{g}^{-1}(\mathbf{y})\right) \left| \det \left(\frac{d\mathbf{g}^{-1}(\mathbf{y})}{d\mathbf{x}} \right) \right|. \tag{4.39}$$

The target distribution is then approximated by transforming the base distribution by this function $\mathbf{g}(.)$ (generator). The target distribution $p_{\mathbf{X}}(\mathbf{x})$ is often also denoted as "*noise*". The function can be determined as an optimization problem by maximizing the log-likelihood e.g. with a stochastic gradient descent.

In order to generate samples from the approximate distribution $p_{\mathbf{Y}}(\mathbf{y})$, we simply need to generate samples from the base distribution and transform them with the generator function $\mathbf{y} = \mathbf{g}(\mathbf{x})$. The opposite direction is given by the inverse function $\mathbf{f}(.)$, that transforms the complex distribution back to the base distribution. As the base distribution is often chosen to be a normal distribution, the whole procedure got the name *Normalizing Flows*. [2]

The function $\mathbf{g}(.)$ can be chosen as a composition of k invertible functions

$$\mathbf{g}(\mathbf{x}) = \mathbf{g}_k(\mathbf{g}_{k-1}(\ldots(\mathbf{g}_1(\mathbf{x})))).$$

Then it can be shown that $\mathbf{g}(.)$ is bijective and its inverse exists: [2]

$$det\left(\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right) = \prod_{i=1}^{k} det\left(\frac{\partial \mathbf{f}_i}{\partial \mathbf{x}}\right) \quad .$$

Coupling flows:

The vector valued derivative of the inverse function is also known as the Jacobian matrix. The complexity of the computation of the determinant of the Jacobian matrix scales as $\mathcal{O}(n^3)$ with n being the dimension of the data ($\mathbf{x} \in \mathbb{R}^n$). Therefore, we would like to have some efficient implementation of the determinant of the Jacobian. In coupling flows we make use of the fact that the determinant of a lower (or upper) triangular matrix is the product of the diagonal entries. In addition to that, coupling flows allow the construction of very expressive functions to transform the base distribution.

For a more detailed description of coupling flows see the paper **Neural Spline Flows**, the following information about coupling flows is taken from there. [1]

Overview:

- The input vector $\mathbf{x}$ is split into two disjoint parts: $[\mathbf{x}_{1:d},\ \mathbf{x}_{d+1:D}]$
- The first part of the vector is mapped directly to the output vector, i.e. $\mathbf{y}_{1:d} = \mathbf{x}_{1:d}$
- Using the first part of the vector, we compute parameters $\Theta = c(\mathbf{x}_{1:d})$, where $c(.)$ can be a deep neural network
- These parameters are used to parameterize a function $\mathbf{g}_\Theta$ that is used to map the second part of the input vector to the output vector $\mathbf{y}_i = g_{\Theta_i}(\mathbf{x}_i)$ for $i = d+1, \ldots, D$. This element-wise mapping can be performed in parallel for each i.

As the first part of the input vector is mapped one-to-one directly to the first part of the output vector, the Jacobian of the function $g(.)$ is an upper diagonal matrix of the form

$$\frac{\partial \mathbf{g}}{\partial \mathbf{x}} = \begin{pmatrix} \mathbf{I} & \mathbf{0} \\ \frac{\partial c}{\partial \mathbf{x}} & \frac{\partial g_\Theta}{\partial \mathbf{x}} \end{pmatrix}$$

and the computation of the determinant simplifies to

$$det\left(\frac{\partial \mathbf{g}}{\partial \mathbf{x}}\right) = \prod_{i=d+1}^{D} \frac{\partial g_{\Theta_i}(x_i)}{\partial x_i} \quad .$$

Note that the splitting of the input vector has to be changed for the coupling flow to achieve expressibility, e.g. through permutations.

Now the question might arise, how the parameterized function g_{Θ_i} is chosen. Two popular ways are presented here, both are invertible elementwise transformations:

- **Affine/additive**: An affine transformation is given as $g_{\Theta_i} = \alpha_i x_i + \beta_i$ with the parameter vector $\Theta_i = [\alpha_i,\ \beta_i]$ (special case: additive transformation for $\alpha_i = 1$). It can be easily understood, that this transform is simple to invert but might fail in modelling complex distributions, e.g. multimodal distributions. Therefore we want a function that is

both quite simple to invert, to differentiate and is expressive in order to model complex distributions.

- **Spline transforms**: Monotonic piecewise polynomials can be used to overcome the above problem of lack of expressibility. Polynomial splines are functions that are defined piecewise by polynomials and are often preferred due to their simplicity and ability to approximate complex shapes. Then the input domain split into K bins and g_{Θ_i} is defined as a polynomial segment, again parameterized by Θ_i. The polynomial segments have to be continuous and smooth at the bin boundaries. In our implementation using *pyro* and **SplineCoupling** according to the documentation linear and quadratic spline segments are used.

In **summary**, coupling flows have several advantages: + efficient computation of the determinant of the Jacobian + the inverse can be computed easily in a single pass by repeating the same steps in the opposite direction with $g_{\Theta_i}^{-1}$ (conditioning the parameter vector Θ stays the same as the first part of the input vector is directly mapped to the output, thus $c(\mathbf{x}_{1:d}) = c(\mathbf{y}_{1:d}) = \Theta$) + no sequential computation + the conditioning function $c(.)$ can be arbitrarily complex (e.g. DNN) as it does not have to be considered in the calculation of the determinant due to the structure of the Jacobian matrix

The coupling transform can be viewed as a *special case of the autoregressive transform* with 2 splits instead of D splits of the input vector. [1]

Autoregressive flows:

All following information about autoregressive flows is taken from [2].

Overview:

Autoregressive models can be interpreted as normalizing flows. An autoregressive model is a function $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^n$ mapping the input vector $\mathbf{x} \in \mathbb{R}^D$ to the output vector $\mathbf{y} \in \mathbb{R}^D$., where the output y_i depends on the actual and previous entries of the input vector $x_{1:i} = [x_1 \dots x_i]$, for $i = 2, \dots, n$.

Let the conditioning function $c_i(.)$ be an arbitrary complex function, e.g. a DNN. The parameter vector is a function of previous entries of the input vector: $\Theta_i = c(x_{1:i-1})$. The first parameter is a constant value: $\Theta_0 = constant$.

Then, each entry of the output vector $\mathbf{y}$ is computed by a bijective function $h_{\Theta_i}(.)$ parameterized by the corresponding parameter vector entry: $y_i = h_{\Theta_i}(x_i)$.

Again, it can be shown that the Jacobian of the function $\mathbf{g}(.)$ via an autoregressive flow is a triangular matrix. The determinant can be computed efficiently and has the following form:

$$det\left(\frac{\partial \mathbf{g}}{\partial \mathbf{x}}\right) = \prod_{i=d+1}^{D} \frac{\partial h_{\Theta_i}(x_i)}{\partial x_i}$$

Given the input vector $\mathbf{x}$, the computation of $\mathbf{z}$ (*forward path*) can be done in parallel. However, the *backward path* has to be computed sequentially due to the following recursion:

$$x_1 = h_{\Theta_1}^{-1}(x_1), \qquad \Theta_1 = c_1(x_0) \tag{4.40}$$
$$x_2 = h_{\Theta_2}^{-1}(x_2), \qquad \Theta_2 = c_2(x_0, x_1) \tag{4.41}$$
$$\vdots \qquad\qquad \vdots \tag{4.42}$$
$$x_i = h_{\Theta_i}^{-1}(x_i), \qquad \Theta_i = c_i(x_0, \ldots, x_{i-1}), \qquad i = 2, \ldots, D \tag{4.43}$$

Therefore, sampling (*forward path*) can be performed in parallel but density estimation (*backward path*) is inherently sequential.

Note: *The whole implementiation can be flipped, which leads to* "inverse autoregressive flows", *where sampling is sequential and density estimation can be done in parallel.*

Summary: [1]

- Coupling flows allow the analytic computation of the inverse in one pass, but are in general less expressive than autoregressive flows
- Autoregressive flows are generally very expressive but slow for sampling or density evaluation due to the sequential nature in one direction Using spline transforms, also coupling flows can be very flexible in modelling complex distributions. Thus, the performance of coupling and autoregressive flows is very similiar.

In our implementation, there were no noticable differences in the quality of the different flow implementations. It could be observed for the autoregressive flow that the run time for sampling is a lot smaller than for density estimation. Overall, the autoregressive flow does have a longer run time while training and during the density estimation in comparison to the coupling flow. It would be interesting to have a look at inverse autoregressive flows and compare the run time. Sadly, for this assignment there was not enough time left looking until the deadline.

List of Figures

Bibliography

[1] Durkan, C., Bekasov, A., Murray, I., and Papamakarios, G. (2019). Neural spline flows.

[2] Kobyzev, I., Prince, S., and Brubaker, M. (2020). Normalizing flows: An introduction and review of current methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, page 1–1.

[3] Yousef, R. and Hindi, K. (2005). Training radial basis function networks using reduced sets as center points. *International Journal of Information Technology*, 2(1):21–35.