Lecture Notes on Machine Learning for Communications

Georg Böcherer
Institute for Communications Engineering
Technical University of Munich
georg.boecherer@ieee.org

April 17, 2022

To cite:

G. Böcherer, *Lecture notes on machine learning for communications.* [Online]. Available: http://georg-boecherer.de/mlcomm

Contents

Acronyms

APP	a-posteriori probability
ASK	amplitude shift keying
AWGN	additive white Gaussian noise
BCE	binary cross equivocation
BER	bit error rate
BPSK	binary phase shift keying
BRGC	binary reflected Gray code
CE	cross equivocation
DAC	digital-to-analog conversion
DC	direct current
DFT	discrete Fourier transform
FIR	finite impulse response
GAN	generative adversarial network
GPU	graphical processing unit

IIR	infinite impulse response
ISI	inter symbol interference
LPR	log probability ratio
MAP	maximum a posteriori probability
MLA	max-log approximation
MSE	mean squared error
NN	neural network
PCA	principal component analysis
PME	principle of maximum entropy
PSD	power spectral density
ReLU	rectified linear unit
RNN	recursive neural network
SGD	stochastic gradient descent
SNR	signal-to-noise ratio
SPS	samples-per-symbol

1. Introduction

In recent years, we could observe very active academic and industrial research on using machine learning for the design and implementation of communication systems, and machine learning techniques are becoming a common tool for communications engineering. For example, Sionna was released recently [1], [2], an open-source Python library for link-level simulations of digital communication systems built on top of the open-source software library TensorFlow for machine learning.

These notes aim to provide an entrance into the field of machine learning for communication system design. In following, we provide a rough definition of what “machine learning” and “communications” may stand for in these notes.

1.1. Machine Learning

We consider a specific subfield of machine learning

- Machine learning
  - ...the machine is an (artificial) neural network (NN)
    - ...the NN is realized by a publicly available software package
      - ...the NN is trained by supervised learning.

There are several software packages that could serve our purpose, some of which are listed in the following table.

package	company	programming language	open source
tensorflow	Google	python/C++/CUDA	yes
pytorch	Facebook	python/C++/CUDA	yes
paddlepaddle	Baidu	python/...	yes

These software packages are all the result of a company-driven effort to develop frameworks for scalable and fast prototyping, training, and deployment of neural networks. We keep most of the material in these notes independent of a specific software package. However, in some examples and problems, we explicitly refer to `pytorch`.

Within the machine learning subfield we specified above, we may define machine learning mathematically as follows:

1. The **objective** is to realize a function

$$f\colon x \mapsto f(x) = y \tag{1.1}$$

minimizing some *loss function*

$$\ell\colon (y,t) \mapsto \ell(y,t). \tag{1.2}$$

where t is the target outcome.

2. The **approach** is to use data for learning f.

Example 1.1 (Learning from Data)**.**

- Input samples $x^n = x_0 x_1 \dots x_{n-1}$
- Target output samples $t^n = t_0 t_1 \dots t_{n-1}$
- Mean square error (MSE) loss function

$$\ell_{\text{mse}}(y,t) = |y-t|^2. \tag{1.3}$$

- Find f that minimizes cost:

$$f^* = \arg\min_f \frac{1}{n}\sum_{i=0}^{n-1} |f(x_i) - t_i|^2 \tag{1.4}$$

You may wonder, what is the big deal about Example 1.1? In the end, classic textbooks on communications also use the MSE design criterion! If you look closely, you will note that most classic textbooks focus on *linear* functions, see, e.g., [3]. In this regard, Example 1.1 is radically different, as optimization is over a general function! E.g., for real-valued functions on the reals, this includes all linear functions, all polynomials, all trigonometric functions like sin, cos, the exponential function, etc. As we will see, NNs can represent almost any function, in particular, **non-linear functions**.

In summary, the paradigm shift is from parametric models to **non-parametric models** and designing functions using machine learning allows us to put focus on

1. Identifying the **input/output** (I/O) interface.
2. Identifying the **appropriate loss** for the task at hand.

The promise is that machine learning takes care of the rest, at least most of the time.

1.2. Communications

According to Claude Elwood Shannon [4]

> "The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point."

The following diagram shows the three main components of a communication system.

$$\begin{tikzcd}
\text{message} \arrow[r] & \boxed{\text{Transmitter}} \arrow[r] & \boxed{\text{Channel}} \arrow[r] & \boxed{\text{Receiver}} \arrow[r] & \text{massage}
\end{tikzcd}$$

In these notes, we specify the different parts of a communication system via **data**, i.e., samples taken at specific points in the system.

We treat the different parts in the system successively.

- At the **receiver** we focus on the problem of detecting symbols and bits from noisy samples, using machine learning techniques commonly called *classification.*
- At the **transmitter**, we let the machine learn message representation insusceptible to channel impairments; an important concept from machine learning that we apply here is the *autoencoder.*
- For training transmitter components, we need to emulate the **channel**, and we use *generative modeling* techniques to achieve this.

1.3. Further Reading

A textbook on machine learning suitable for these notes is [5]. The textbook [6] provides a good introduction to the design of digital communication systems.

References

[1] J. Hoydis, S. Cammerer, F. Ait Aoudia, A. Vem, N. Binder, G. Marcus, and A. Keller, “Sionna: An open-source library for next-generation physical layer research,” *arXiv preprint*, 2022.

[2] *Sionna: An open-source library for next-generation physical layer research.* [Online]. Available: https://github.com/NVlabs/sionna.

[3] T. Kailath, A. H. Sayed, and B. Hassibi, *Linear estimation.* Prentice Hall, 2000.

[4] C. E. Shannon, “A mathematical theory of communication,” *Bell Syst. Tech. J.*, vol. 27, 379–423 and 623–656, 1948.

[5] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems.* O’Reilly Media, 2019.

[6] R. G. Gallager, *Principles of Digital Communication.* Cambridge University Press, 2008.

2. Probability and Information

Cross entropy is a loss function widely used in machine learning. In this chapter, we summarize the most important information measures and their properties, which serves us as a foundation for later chapters, where we relate objectives relevant for communication system design to the cross entropy loss.

2.1. Probabilistic Expectation

- Random variable X with distribution P_X on alphabet $\mathcal{X}$
- Real-valued function $f\colon \mathcal{X} \to \mathbf{R}$
- **Probabilisitc expectation**

$$\mathbb{E}[f(X)] := \sum_{a\in\mathcal{X}} P_X(a)f(a). \tag{2.1}$$

- Most important for us is the **empirical expectation**. For n samples from P_X,

$$\frac{1}{n}\sum_{i=1}^{n-1} f(x_i) \approx \mathbb{E}[f(X)]. \tag{2.2}$$

Example 2.1.

```
In [21]: pX = [0.5, 0.5]

In [22]: x = np.random.choice([-1, 1], 1000000, p=pX)

In [23]: np.mean(x)
Out[23]: 0.000152

In [24]: np.dot(pX, [-1, 1])
Out[24]: 0.0
```

2.2. Entropy

- Random variable X, distribution P_X on alphabet $\mathcal{X}$.

- The entropy of X in bits is

$$\begin{aligned}\mathbb{H}(X) = \mathbb{H}(P_X) &= \sum_{a\in\mathcal{X}} P_X(a)[-\log_2 P_X(a)] && (2.3)\\ &= \mathbb{E}[-\log_2 P_X(X)] && (2.4)\\ &\approx -\frac{1}{n}\sum_{i=1}^{n} \log_2 P_X(x_i). && (2.5)\end{aligned}$$

- The entropy is bounded by

$$0 \leq \mathbb{H}(X) \leq \log_2 |\mathcal{X}|. \tag{2.6}$$

2.3. Conditional Entropy ("Equivocation")

- Two random variable X and Y with joint distribution P_{XY} on $\mathcal{X} \times \mathcal{Y}$.
- Bayes' rule:

$$P_{XY}(ab) = P_{X|Y}(a|b)P_Y(b). \tag{2.7}$$

  - P_Y is a distribution on $\mathcal{Y}$.
  - For each $b \in \mathcal{Y}$, $P_{X|Y}(\cdot|b)$ is a distribution on $\mathcal{X}$.

- Conditional entropy

$$\begin{aligned}\mathbb{H}(X|Y) &= \sum_{a\in\mathcal{X}}\sum_{b\in\mathcal{Y}} P_{XY}(ab)[-\log_2 P_{X|Y}(a|b)] && (2.8)\\ &= \mathbb{E}[-\log_2 P_{X|Y}(X|Y)] && (2.9)\\ &\approx -\frac{1}{n}\sum_{i=0}^{n-1} \log_2 P_{X|Y}(x_i|y_i) && (2.10)\end{aligned}$$

where x^n, y^n are sampled from P_{XY}.

2.4. Mutual Information

$$\begin{tikzcd} \text{message} \arrow[r, "B"] & \boxed{\text{Transmitter}} \arrow[r, "X"] & \boxed{\text{Channel}} \arrow[r, "Y"] & \boxed{\text{Receiver}} \arrow[r, "\hat{B}"] & \text{massage} \end{tikzcd}$$

- Mutual information of X and Y is

$$\mathbb{I}(X;Y) = \mathbb{H}(X) - \mathbb{H}(X|Y). \tag{2.11}$$

- Mutual information $\mathbb{I}(X;Y)$ quantifies the rate at which we can reliably transmit over the channel.

- The data processing inequality relates $\mathbb{I}(B;\hat{B})$ and $\mathbb{I}(X;Y)$:

$$\mathbb{I}(B;\hat{B}) \leq \mathbb{I}(X;\hat{B}) \leq \mathbb{I}(X;Y). \tag{2.12}$$

$\Rightarrow$ Large $\mathbb{I}(X;Y)$ is a requirement for large reliable rate from B to $\hat{B}$.

2.5. Importance of Equivocation

$$\begin{tikzcd}\text{message} \arrow[r, "B"] & \boxed{\text{Transmitter}} \arrow[r, "X"] & \boxed{\text{Channel}} \arrow[r, "Y"] & \boxed{\text{Receiver}} \arrow[r, "\hat{B}"] & \text{massage}\end{tikzcd}$$

- Since X is emitted by the transmitter, we usually have full control of $\mathbb{H}(X)$.
- Large $\mathbb{I}(X;Y) = \mathbb{H}(X) - \mathbb{H}(X|Y)$ therefore corresponds to small $\mathbb{H}(X|Y)$.

$\Rightarrow$ The equivocation $\mathbb{H}(X|Y)$ is of central importance to us.

2.6. Cross Entropy

$$\begin{tikzcd}\text{message} \arrow[r, "B"] & \boxed{\text{Transmitter}} \arrow[r, "X"] & \boxed{\text{Channel}} \arrow[r, "Y"] & \boxed{\text{Receiver}} \arrow[r, "\hat{B}"] & \text{massage}\end{tikzcd}$$

- Frequently, we have full control of x^n, we can measure y^n, but we do not know $P_{X|Y}$ and P_Y.
- If we assume some Q_Y, we can estimate

$$-\frac{1}{n}\sum_{i=1}^{n} \log_2 Q_Y(y_i) \approx \mathbb{E}[-\log_2 Q_Y(Y)] \tag{2.13}$$

  where the expectation is with respect to P_Y. This quantity is called **cross entropy**, which we also denote by

$$\mathbb{X}(P_Y \| Q_Y) := \mathbb{E}[-\log_2 Q_Y(Y)]. \tag{2.14}$$

2.7. Cross Equivocation

$$\begin{tikzcd}\text{message} \arrow[r, "B"] & \boxed{\text{Transmitter}} \arrow[r, "X"] & \boxed{\text{Channel}} \arrow[r, "Y"] & \boxed{\text{Receiver}} \arrow[r, "\hat{B}"] & \text{massage}\end{tikzcd}$$

- If we assume some $Q_{X|Y}$, we can estimate

$$-\frac{1}{n}\sum_{i=1}^{n} \log_2 Q_{X|Y}(x_i|y_i) \approx \mathbb{E}[-\log_2 Q_{X|Y}(X|Y)] \tag{2.15}$$

where the expectation is with respect to P_{XY}. This quantity is called **cross equivocation**, which we also denote by

$$\begin{aligned} \mathbb{X}(P_{X|Y} \| Q_{X|Y} | P_Y) &:= \mathbb{E}[-\log_2 Q_{X|Y}(X|Y)] && (2.16)\\ &= \sum_{b \in \mathcal{Y}} P_Y(b) \sum_{a \in \mathcal{X}} P_{X|Y}(a|b)[-\log_2 Q_{X|Y}(a|b)] && (2.17) \end{aligned}$$

2.8. Information Inequality for Cross Entropy

- The most important property to us is the **information inequality**, which states that

$$\mathbb{X}(P_Y \| Q_Y) \geq \mathbb{H}(P_Y) \tag{2.18}$$

with equality if and only if $Q_Y = P_Y$.

$\Rightarrow$ This allows us to learn P_Y from data by solving

$$P_Y \approx \operatorname*{arg\,min}_{Q_Y} -\frac{1}{n} \sum_{i=1}^{n} \log_2 Q_Y(y_i). \tag{2.19}$$

2.9. Further Reading

The textbook [1] provides a good introduction to information measures and their properties.

References

[1] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, 2nd ed. John Wiley & Sons, Inc., 2006.

Part I.

Receiver

3. Equalization of Deterministic Impairments

In this chapter, we consider equalization, i.e., the reconstruction of a transmitted signal by filtering a received signal. Throughout this chapter, we consider the following design problem.

- **Data:**
  - At the transmitter, message bits have been mapped to the symbol sequence $\boldsymbol{x} = x_0 \dots x_{n-1}$, with
  $$x_i \in \{-3, -1, 1, 3\}. \tag{3.1}$$
  - The receiver has been synchronized to the transmitter and an oversampled, noisy sequence is available. The oversampling factor is 2 samples-per-symbol (SPS), i.e., the received sequence is
  $$\boldsymbol{y} = y_0 \dots y_{2n-1}, \quad y_i \in \mathbf{R}. \tag{3.2}$$
- **I/O specification:** We are looking for a function f that maps $\boldsymbol{y}$ to the estimate $\hat{\boldsymbol{x}} = f(\boldsymbol{y})$ of $\boldsymbol{x}$.
  - $\boldsymbol{y}$ is the input of f
  - $\hat{\boldsymbol{x}}$ is the output of f
  - $\boldsymbol{x}$ is the target output of f
- **Loss:** $\text{MSE}(\boldsymbol{x}, \hat{\boldsymbol{x}})$ should be small.

A simple device that fulfills the above I/O specification is a downsampler with downsampling factor 2. Such downsampler discardes every second sample, as illustrated in the following diagram.

$$\begin{tikzcd} {} \arrow[r, "y_0y_1y_2y_3\cdots"] & \boxed{\text{Downsampler}} \arrow[r, "y_0y_2y_4\cdots", "=\hat{x}_0\hat{x}_1\hat{x}_2\cdots"'] & {} \end{tikzcd}$$

In Figure 3.1, we display a scatterplot of the downsampled sequence $\hat{\boldsymbol{x}}$ and the transmitted sequence $\boldsymbol{x}$. As we can see, the downsampled sequence is significantly distorted. In the next section, we use a neural network (NN) to filter the received sequence $\boldsymbol{y}$ prior to downsampling.

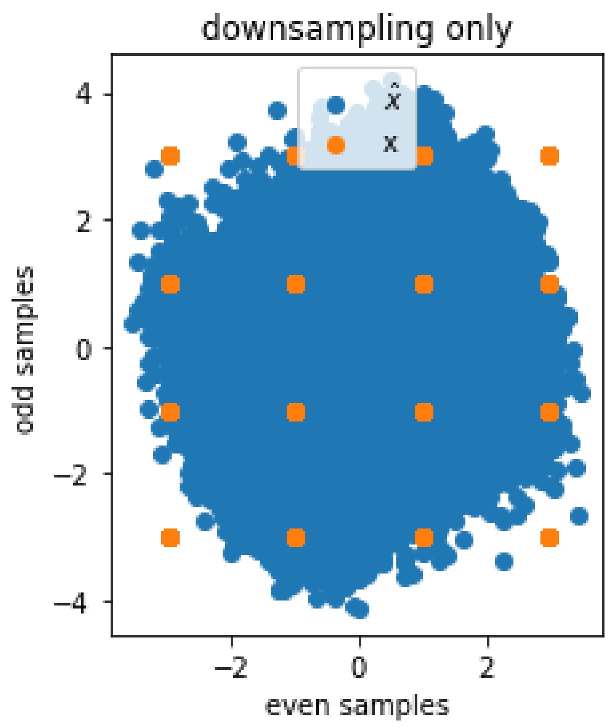


Figure 3.1.: Scatterplot of the downsampled sequence $\hat{\boldsymbol{x}}$ and the transmitted sequence $\boldsymbol{x}$.

3.1. Linear Equalization

We now make the function f more powerful by using a linear filter h prior to downsampling, as illustrated in the following diagram.

$$\begin{tikzcd}
\text{input } \boldsymbol{y} \arrow[r] & \boxed{\text{Filter } h} \arrow[r, "\boldsymbol{z}"] & \boxed{\text{Downsampler}} \arrow[r] & \text{output } \hat{\boldsymbol{x}}
\end{tikzcd}$$

f

We realize the filter h by a linear NN with weights $w_0, \ldots, w_{2m}$ and bias b and apply it at positions $i = 0, 1, 2, \ldots, 2n-1$, as shown in the following diagram.

$$\begin{tikzcd}
\cdots & y_{i-m-1} & y_{i-m} \arrow[drr, "w_0"'] & \cdots & y_i \arrow[d, "w_m"'] & \cdots & y_{i+m} \arrow[dll, "w_{2m}"] & y_{i+m+1} & \cdots \\
 & & & b \arrow[r] & \Sigma \arrow[d] & & & & \\
 & & & & z_i & & & &
\end{tikzcd}$$

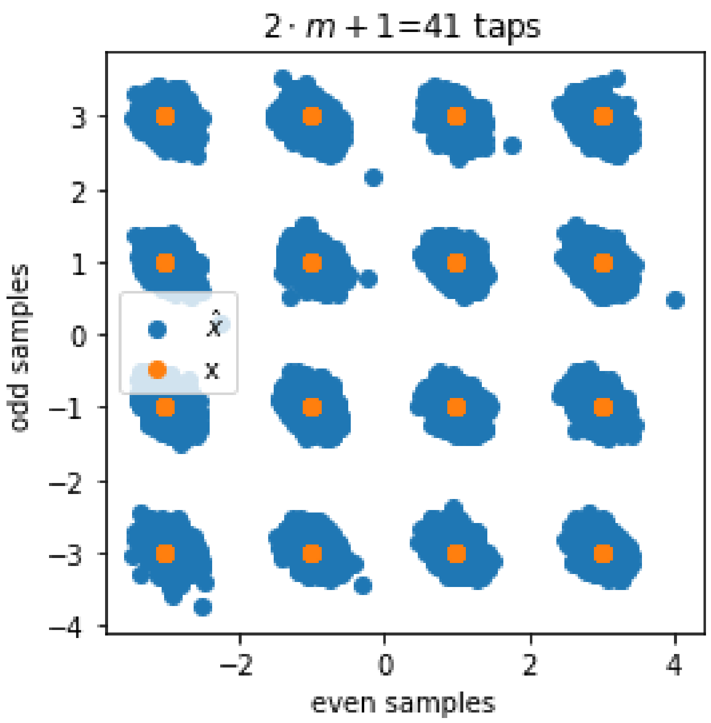


Figure 3.2.: Scatterplot of the linearly filtered and downsampled sequence $\hat{\boldsymbol{x}}$ and the transmitted sequence $\boldsymbol{x}$.

The following code implements the filter in `pytorch`.

```
filter = torch.nn.Conv1d(in_channels=1,
                         out_channels=1,
                         kernel_size=2 * m + 1,
                         padding='same')
```

To learn the weights $\boldsymbol{w}$ and the bias b, it remains to

- prepare the data
- initialize an optimizer
- run a training loop

These tasks a carried out in Problem 3.1. In Figure 3.2, we show the scatterplot of the equalized signal and the transmitted signal. In comparison with Figure 3.1, we note that linear filtering prior to downsampling has improved the quality of the equalized signal significantly.

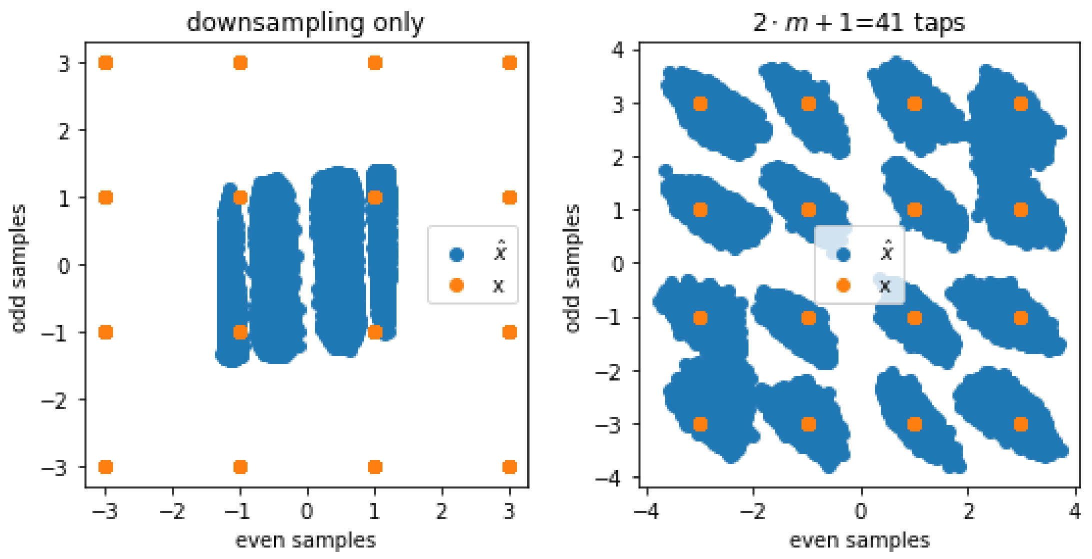


Figure 3.3.: Scatterplots of downsampling and linear filtering followed by downsampling, respectively. The received signal was subject to non-linear distortion. As we can see, the quality of the equalized signal is poor.

3.2. Non-Linear Equalization

We now consider a new data set $\boldsymbol{x}$, $\boldsymbol{y}$, where this time, the received sequence $\boldsymbol{y}$ was subject to non-linear distortion under transmission. As we can see in Figure 3.3, our linear equalizer is not not able to reconstruct the transmitted signal in a satisfactory manner. We therefore enhance the capability of our NN equalizer in two ways:

- We add *hidden layers.*
- We add *non-linear activation functions.*

3.2.1. Hidden Layers

Our linear NN from Section 3.1 has two layers, namely the input layer consisting of a real vector $\boldsymbol{y}_0$ with $2m+1$ entries, and an output layer consisting of one real value z. We can represent the calculation of the input from the output by an inner product

$$z = \boldsymbol{y}_0 \boldsymbol{w}^T + b. \tag{3.3}$$

We can generalize this input-output relation by allowing for more than one output, and by using more layers. We illustrate this by the example network displayed in Figure 3.4.

- The input is the row vector

$$\boldsymbol{y} = y_{i-1} y_i y_{i+1} \tag{3.4}$$

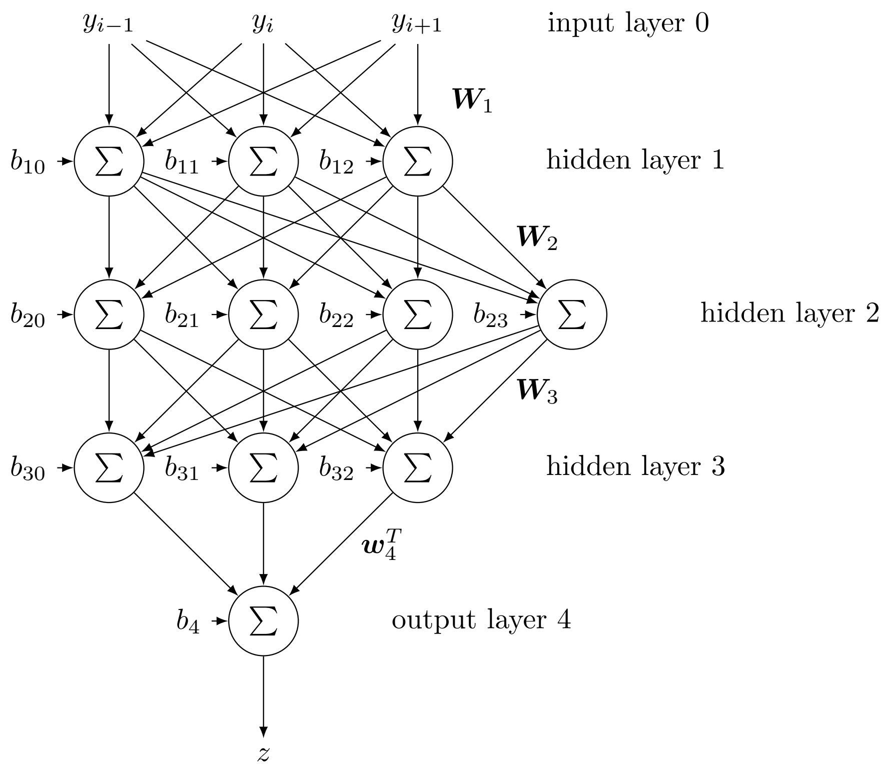


Figure 3.4.: An NN with hidden layers. Since this NN does not have non-linear activation functions, it is equivalent to a simple linear NN with 3 input unit and 1 output unit.

- The weight matrices are of size

$$\boldsymbol{W} \in \mathbf{R}^{\#\text{inputs}\times\#\text{outputs}} \tag{3.5}$$

- The output of layer k is

$$\boldsymbol{y}_k = \boldsymbol{y}_{k-1}\boldsymbol{W}_k + \boldsymbol{b}_k. \tag{3.6}$$

- $\boldsymbol{w}_4$ is a row vector, $\boldsymbol{w}_4^T$ is a column vector.

To see which function is realized by the network, we carry out the vector matrix multiplications:

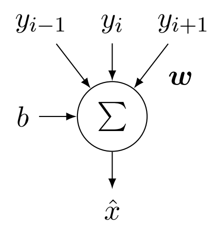
$$\hat{x} = (((\boldsymbol{y}\boldsymbol{W}_1 + \boldsymbol{b}_1)\boldsymbol{W}_2 + \boldsymbol{b}_2)\boldsymbol{W}_3 + \boldsymbol{b}_3)\boldsymbol{w}_4^T + b_4 \tag{3.7}$$

$$= ((\boldsymbol{y}\boldsymbol{W}_1\boldsymbol{W}_2 + \boldsymbol{b}_1\boldsymbol{W}_2 + \boldsymbol{b}_2)\boldsymbol{W}_3 + \boldsymbol{b}_3)\boldsymbol{w}_4^T + b_4 \tag{3.8}$$

$$= (\boldsymbol{y}\boldsymbol{W}_1\boldsymbol{W}_2\boldsymbol{W}_3 + (\boldsymbol{b}_1\boldsymbol{W}_2 + \boldsymbol{b}_2)\boldsymbol{W}_3 + \boldsymbol{b}_3)\boldsymbol{w}_4^T + b_4 \tag{3.9}$$

$$= \boldsymbol{y}\,\underbrace{\boldsymbol{W}_1\boldsymbol{W}_2\boldsymbol{W}_3\boldsymbol{w}_4^T}_{=:\boldsymbol{w}^T} + \underbrace{((\boldsymbol{b}_1\boldsymbol{W}_2 + \boldsymbol{b}_2)\boldsymbol{W}_3 + \boldsymbol{b}_3)\boldsymbol{w}_4^T + b_4}_{=:b}. \tag{3.10}$$

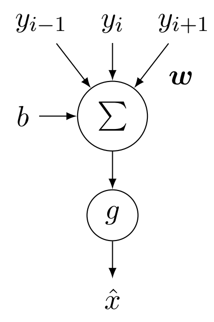
We note that the complicated network in Figure 3.4 is equivalent to the simple network



Thus, using hidden layers does not enhace our linear NN from Section 3.1 at all! The essential missing ingredients are non-linear activation functions, which we are going to discuss next.

3.2.2. Non-Linear Activation Functions

The key step to turn the redundant hidden layers into powerful tools is to pass the outputs of each linear layer through a non-linear activation function g, which is applied entrywise. The building block of a non-linear NN is



which we can express mathematically by

$$\hat{x} = g(\boldsymbol{y}\boldsymbol{w}^T + b). \tag{3.11}$$

Common linear activation functions are the rectified linear unit (ReLU), the hyperbolic tangent (tanh), and the logistic function, which we define and plot in Figure 3.5.

3.2.3. Non-linear Equalizer

The code in Figure 3.6 specifies a non-linear equalizer using three hidden layers and the ReLU activation function. The scatterplot of the signal equalized by this non-linear equalizer is displayed in Figure 3.7. As we can see, the equalized signal has high quality, despite the non-linear distortions during transmission. The implementation and evaluation of a non-linear equalizer is treated in Problem 3.2.

ReLU	tanh	logistic
$\max\{0, x\}$	$\frac{e^x - e^{-x}}{e^x + e^{-x}}$	$\frac{1}{1+e^{-x}}$

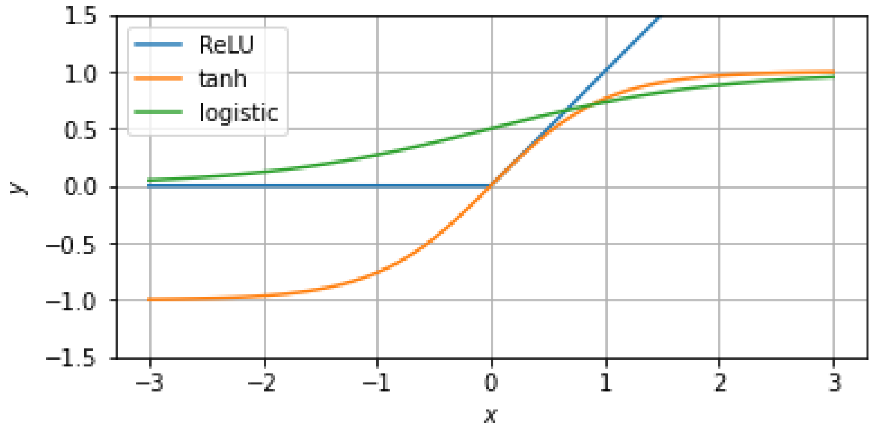


Figure 3.5.: Common non-linear activation functions.

```python
class EQnonlinear(torch.nn.Module):
    def __init__(self, num_taps):
        super().__init__()
        self.num_taps = num_taps
        self.model = torch.nn.Sequential(
            nn.Linear(num_taps, 25),
            nn.ReLU(),
            nn.Linear(25, 25),
            nn.ReLU(),
            nn.Linear(25, 25),
            nn.ReLU(),
            nn.Linear(25, 1))

    def forward(self, y):
        return self.model(y)
```

Figure 3.6.: Nonlinear equalizer in `pytorch`.

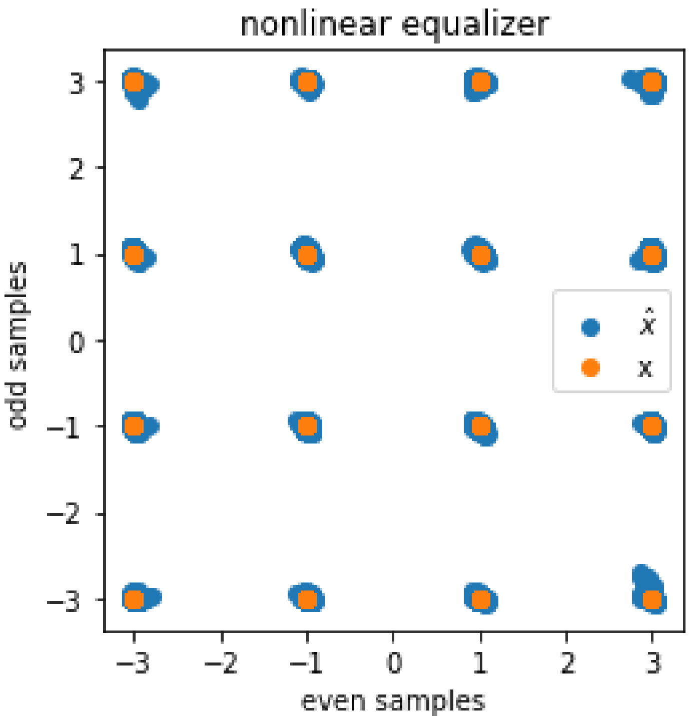


Figure 3.7.: Scatterplot of nonlinear filtering followed by downsampling. In comparison to linear filtering (see Figure 3.3), the signal equalized by nonlinear filtering has a much better quality.

3.3. Further Reading

3.4. Problems

Problem 3.1. For a provided data set, implement and train a linear equalizer using the mean squared error (MSE) loss function. Visualize the equalized signal and the target signal using a scatterplot.

Problem 3.2. For a provided data set, implement and train a non-linear equalizer using the MSE loss function. Visualize the equalized signal and the target signal using a scatterplot.

4. Demapper

$$\begin{tikzcd} X \in \mathcal{X} \arrow[r] & \boxed{\text{Channel}} \arrow[r] & Y \arrow[r] & \boxed{\text{Demapper}} \arrow[r] & \text{decision } Q_{X|Y}(\cdot|Y) \end{tikzcd}$$

Figure 4.1.: Communication system with soft demapper outputting for each symbol in the input alphabet an estimate of the probability that it was transmitted, given the observed channel output Y. The probabilities are collected in the distribution $Q_{X|Y}(\cdot|Y)$ on the channel input alphabet $\mathcal{X}$.

A symbol X from an input alphabet $\mathcal{X}$ is transmitted over a channel, which outputs Y. The purpose of a demapper is to provide, based on its channel output observation Y, a decision on which symbol from the input alphabet was transmitted.

Hard Decision If the channel mapping from input to output is deterministic, then the demapper may output a hard decision, i.e., a single value $\hat{x}(Y)$ from the input alphabet $\mathcal{X}$.

Soft Decision If the mapping from input to output realized by the channel is not deterministic, i.e., if it is random, then a correct decision may not always be possible. In this case, the demapper may take a soft decision, i.e., instead of outputting one single symbol from the input alphabet, the demapper provides for each symbol $a \in \mathcal{X}$ an estimate of the probability that it was transmitted, given the channel output observation Y. We collect these probability estimates in a distribution $Q_{X|Y}(\cdot|Y)$ on $\mathcal{X}$. If a hard decision (which is occasionally wrong) is required, then the most probable symbol from the input alphabet may be chosen, i.e.,

$$\hat{x}(Y) = \operatorname*{arg\,max}_{x \in \mathcal{X}} Q_{X|Y}(x|Y).. \tag{4.1}$$

The decision rule (4.1) is called the maximum a posteriori probability (MAP) rule.

4.1. Cross Equivocation Loss Function

4.1.1. Channel Input as Target Value

We now devise a strategy to train a demapper whose output approximates the true a-posteriori probability (APP) distribution $P_{X|Y}(\cdot|Y)$. For training, consider the samples

$x^n = x_0 \dots x_{n-1}$ and $y^n = y_0 \dots y_{n-1}$ of channel input and output, respectively. Also, let $Q_{X|Y}(\cdot|y)$ be the actual distribution output by the demapper. We estimate the cross equivocation of $P_{X|Y}$ and $Q_{X|Y}$ by

$$-\frac{1}{n}\sum_{i=0}^{n-1} \log_2 Q_{X|Y}(x_i|y_i) \approx \mathbb{E}[-\log_2 Q_{X|Y}(X|Y)]. \tag{4.2}$$

By the information inequality, the cross equivocation is lower bounded by the equivocation

$$\mathbb{E}[-\log_2 Q_{X|Y}(X|Y)] \geq \mathbb{E}[-\log_2 P_{X|Y}] = \mathbb{H}(X|Y). \tag{4.3}$$

with equality in (4.3) if and only if $Q_{X|Y} = P_{X|Y}$. Thus, we can use the left-hand side of (4.2) as cost function to train the demapper, i.e., training consists in minimizing the cross equivocation estimate

$$Q^*_{X|Y} = \underset{Q_{X|Y}}{\arg\min} -\frac{1}{n}\sum_{i=0}^{n-1} \log_2 Q_{X|Y}(x_i|y_i). \tag{4.4}$$

4.1.2. Message as Target Value

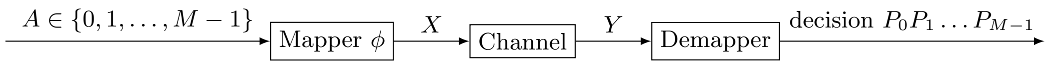


Figure 4.2.: Communication system with soft demapper outputting for message $a \in \{0,\dots,M-1\}$ an estimate p_a of the probability that it was transmitted, given the observed channel output Y. The probabilities are collected in the distribution $\boldsymbol{p}(Y) = \boldsymbol{P} = P_0P_1\dots P_{M-1}$ on the message set $\{0,1,\dots,M-1\}$.

We would now like to formulate (4.4) in a more generic form. To this end, we represent the channel input symbol X by the message $A \in \{0,1,\dots,M-1\}$, i.e., $X = \phi(A)$. Also, we define the random vector output by the demapper as

$$P_a = p_a(Y) = Q_{X|Y}(\phi(a)|Y), \quad a \in \{0,1,\dots,M-1\}. \tag{4.5}$$

In consistency with common practice in machine learning programming frameworks, we switch our unit of information from bits to nats, so that we can use the natural logarithm log insted of the binary logarithm $\log_2$. (We can always convert nats to bits by dividing by $\log(2)$). We can now define the cross equivocation (CE) loss function by

$$\mathrm{ce}(a, \boldsymbol{p}) = -\log p_a \quad \text{[nats]} \tag{4.6}$$

and the CE cost function by

$$\mathrm{CE}(a^n, \boldsymbol{p}^n) = \frac{1}{n}\sum_{i=0}^{n-1} \mathrm{ce}(a_i, \boldsymbol{p}_i). \quad \text{[nats]} \tag{4.7}$$

4.2. NN Demapper Outputting Probabilities

For input y, an NN demapper should output a distribution $\boldsymbol{p}(y)$ on the message set $\{0, 1, \dots, M-1\}$. This translates into the following requirements for the NN demapper output

1. The NN demapper must have M outputs.
2. Each output $p_a(y)$ must represent a probability, i.e., a value between 0 and 1.
3. The outputs should sum to 1.

To fulfill requirement 1., we use an output layer with M neurons. To ensure requirement 2., we use the logistic activation function

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{4.9}$$

in the output layer. For requirement 3., we need to normalize, which requires processing all output values jointly. The complete NN output layer is displayed in Figure 4.3. From inputs $\boldsymbol{\ell} = \ell_0 \dots \ell_{M-1}$, its outputs are calculated by

$$p_a = \frac{\sigma(\ell_a)}{\sum_{j=0}^{M-1} \sigma(\ell_j)}, \quad a \in \{0, 1, \dots, M-1\}. \tag{4.10}$$

Note that the input of the output layer is unconstrained, i.e., the activation function of the output layer transforms any real vector with M entries, possibly taking negative and positive values, into a distribution on $\mathcal{X}$.

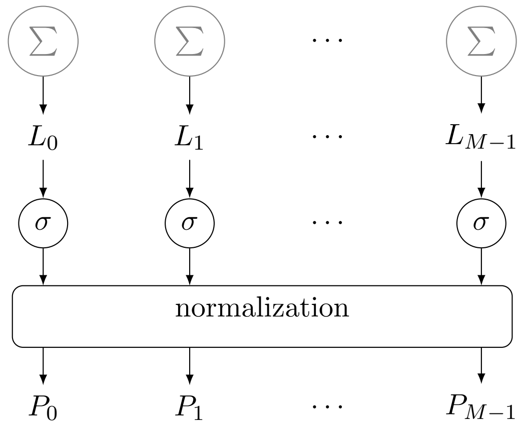
The same is achieved by the softmax activation

$$\operatorname{softmax}(\ell_0 \ell_1 \dots \ell_{M-1}) = \frac{1}{\sum_{j=0}^{M-1} e^{\ell_j}} (e^{\ell_0}, e^{\ell_1}, \dots, e^{\ell_{M-1}}) \tag{4.11}$$

which realizes the transformation of real vectors into probability distributions. The ath output of softmax is the probability

$$p_a = \operatorname{softmax}(\ell_0 \ell_1 \dots \ell_{M-1})_a = \frac{e^{\ell_a}}{\sum_{j=0}^{M-1} e^{\ell_j}}. \tag{4.12}$$

An NN terminated by a softmax layer is shown in Figure 4.4.

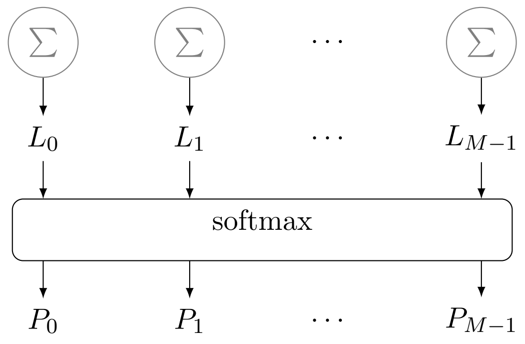


Demapper outputting probabilities

- NN is terminated by a linear layer, followed by logistic activation and normalization.
- Output $\boldsymbol{P}$ is a probability distribution on the message set $\{0, 1, \ldots, M-1\}$.
- For training, use cross equivocation loss on probabilities

$$\operatorname{ce}(a, \boldsymbol{p}) = -\log p_a \quad \text{[nats]} \tag{4.8}$$

Figure 4.3.: NN termination transforming intermediate values L_a into non-negative probabilities summing to one by using logistic activations followed by a normalization layer. In most cases, it is reasonable to choose L_a as output of a linear layer, which is indicated by the linear units in gray. The linear units have bias and inputs; these are omitted in the diagram.



Demapper outputting probabilities

- NN is terminated by a linear layer, followed by a softmax layer.
- Output $\boldsymbol{P}$ is a probability distribution on the message set $\{0, 1, \ldots, M-1\}$.
- For training, use cross equivocation loss on probabilities

$$\operatorname{ce}(a, \boldsymbol{p}) = -\log p_a \quad \text{[nats]}$$

Figure 4.4.: NN termination transforming intermediate values L_i into non-negative probabilities summing to one by using a softmax layer.

Demapper outputting log probabilities

- NN is terminated by a linear layer.
- Output $\boldsymbol{L}$ is a vector of unnormalized log probabilities on the message set $\{0, 1, \ldots, M-1\}$.
- For training, use cross equivocation loss on log probabilities

$$\operatorname{ce}_{\log}(a, \boldsymbol{\ell}) = -\log \frac{e^{\ell_a}}{\sum_{j=0}^{M-1} e^{\ell_j}}.$$

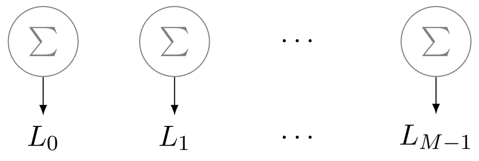


Figure 4.5.: NN termination outputting unnormalized log probabilities.

4.3. NN Demapper Outputting Log Probabilities

Let's consider again the loss (4.6) in the case when the probabilities are calculate by a softmax layer. We have

$$\begin{aligned}
\mathrm{ce}(a, \boldsymbol{p}) &= -\log p_a && (4.13)\\
&= -\log \frac{e^{\ell_a}}{\sum_{j=0}^{M-1} e^{\ell_j}} && (4.14)\\
&= -\ell_a + \log\Bigl(\sum_{j=0}^{M-1} e^{\ell_j}\Bigr) \quad [\text{nats}]. && (4.15)
\end{aligned}$$

In fact, a scaled version of the intermediate value ℓ_a plus a normalization term is output! This suggests to directly define the loss on the intermediate value ℓ_a. Up to normalization, we can interpret the ℓ_a as log probabilities. NN outputting log probabilities have several advantages

1. The NN has one layer of non-linear activations less (i.e., we save the logistic functions in Figure 4.3 and the softmax layer in Figure 4.4), which makes it less complex for deployment (during training, complexity is unchanged, we just moved the non-linear activation layer from the NN into the loss function).

2. ML literature often states that (4.15) is numerically more stable. The reason is that e^{ℓ} may be very small or very large, saturating $\log e^{\ell}$ to $\pm\infty$, while ℓ is a finite value. The normalization term is a log-sum-exp expression, which can be realized by

$$\log\Bigl(\sum_{j=0}^{M-1} e^{\ell_j}\Bigr) = \ell^* + \log\Bigl(\sum_{j=0}^{M-1} e^{\ell_j - \ell^*}\Bigr)$$
$$\text{with } \ell^* = \max\{\ell_0, \dots, \ell_{M-1}\} \qquad (4.16)$$

   which also alleviates the problem of saturating the log function. See Problem 4.1.

3. In communication systems, the soft decision output of the demapper is fed to a soft decision FEC decoder, which in most cases work on log probabilities.

4.3.1. Cross Equivocation for Log Probabilities

For training NN demappers outputting log probabilities, the cross equivocation loss is

$$\mathrm{ce}_{\log}(a, \boldsymbol{\ell}) = -\log \frac{e^{\ell_a}}{\sum_{j=1}^{M-1} e^{\ell_a}} \quad [\text{nats}]. \qquad (4.17)$$

(Of course, a numerically stable implementation of the right-hand side should be chosen). The corresponding cost function is

$$\mathrm{CE}_{\log}(a^n, \boldsymbol{\ell}^n) = \frac{1}{n}\sum_{i=0}^{n-1} \mathrm{ce}_{\log}(a_i, \boldsymbol{\ell}_i) \quad [\text{nats}]. \qquad (4.18)$$

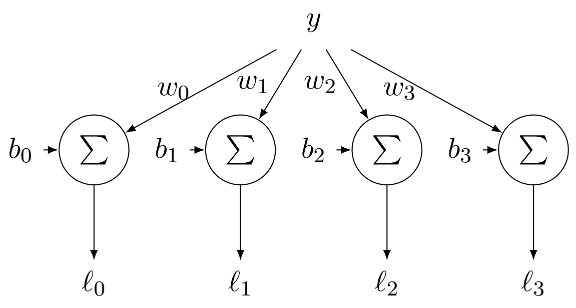


Figure 4.6.: Simple NN demapper for the AWGN channel with input alphabet size 4, outputting unnormalized log probabilities.

4.4. NN Demapper for AWGN Channel

We now describe a simple demapper that is able to calculate soft decisions for AWGN channels. For n channel uses, let $x^n = x_0 \dots x_{n-1}$ denote real-valued channel inputs. The channel outputs are

$$Y_i = x_i + Z_i, \quad i = 0, \dots, n-1 \tag{4.19}$$

where the Z_i are independent and zero mean Gaussian with variance σ_z^2, which we abbreviate by $Z_i \sim \mathcal{N}(0, \sigma_z^2)$. Since the Z_i are random and independent, we can not learn the noise realization Z^n, but since the Z_i are identically distributed, we can learn the variance σ_z^2, since for large enough n, we have

$$\frac{1}{n} \sum_{i=0}^{n-1} z_i^2 \approx \mathbb{E}[Z^2] = \sigma_z^2. \tag{4.20}$$

The input samples x_i take values in a finite set of real numbers, for instance, input and output of the mapper may be given by

message A	0	1	2	3
symbol X	-3	-1	1	3

We now want to devise an NN demapper that takes as input the channel output sample y and outputs soft decisions in the form of unnormalized log probabilities $\ell_a = \log p_a + K$, where p_a is a probability and where K is some constant. The simplest such demapper is shown in Figure 4.6. It merely consists of one linear output layer with $M = 4$ neurons. Problem 4.2 compares the demapper in Figure 4.6 to an optimal demapper specified by analytical expressions.

4.5. Further Reading

In his youtube video [1], Aurélien Géron, the author of the great textbook on machine learning [2], talks about entropy and cross-entropy. Effectively, he is talking about what we call equivocation and cross equivocation in these notes. The conditioning on the classifier's observation Y (a picture of a cat) of the object X (a cat) to be classified is left implicit by Aurélien. This is common practice in the machine learning community and the reason why what we call cross equivocation loss is called cross entropy loss in the machine learning programming frameworks, e.g., pytorch and tensorflow. In these notes, we prefer to emphasize the conditioning on the classifiers observation and therefore use the term cross equivocation loss.

References

[1] A. Géron, *A short introduction to entropy, cross-entropy, and kl-divergence*, 2018. [Online]. Available: https://youtu.be/ErfnhcEV1O8.

[2] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems.* O'Reilly Media, 2019.

4.6. Problems

Problem 4.1. (Numeric stability)

1. Consider the code

```python
import torch

def f64(x):
    x = x.type(torch.float64)
    x = torch.exp(x)
    return torch.log(x)

def f32(x):
    x = x.type(torch.float32)
    return x
```

2. What is the smallest integer value `x` for which `f64(x)!=x`?

3. For the value `x` determined in 1., do you have `f32(x)=x`?

4. Repeat the analysis for 1. and 2. to compare a `float64` implementation of the log-sum-exp expression on the left-hand side of (4.16) to a `float32` implementation of the right-hand side of (4.16).

Problem 4.2. (How good is the simple AWGN NN Demapper?)

1. Calculate $P_{A|Y}(\cdot|y)$ in terms of channel output y, mapper function ϕ, and noise variance σ_Z^2.

2. Suppose A is uniformly distributed. The signal-to-noise ratio (SNR) in dB is

$$\mathrm{SNR}_{\mathrm{dB}} = 10 \log_{10} \frac{\mathbb{E}(X^2)}{\mathbb{E}(Z^2)}. \tag{4.21}$$

   For $\mathrm{SNR}_{\mathrm{dB}} = 5, 6, 7, \dots, 10$, use sample sequences to calculate an equivocation estimate $\hat{\mathbb{H}}(A|Y)$ and plot the mutual information estimate in bits

$$\hat{\mathbb{I}}(A;Y) = \mathbb{H}(A) - \hat{\mathbb{H}}(A|Y) \quad \text{[bits]}. \tag{4.22}$$

3. For each SNR value from 2., train an AWGN NN demapper as in Figure 4.6 using the cross equivocation cost for log probabilities (4.18).

4. For the trained NN demappers, calculate equivocation estimates in bits from input and output sequences. Plot the corresponding mutual information estimate in bits. What is the gap in mutual information and in SNR, compared to the analytical demapper from 2.?

5. For $\text{SNR}_{\text{dB}} = 5$, sample a channel output sequence y^n and pass it to the trained demapper to obtain unnormalized log probabilities $\boldsymbol{\ell}^n$. Assume

$$\ell_{i,a} = \log p_{i,a} + K, \quad i = 0, \dots, n-1, \quad a = 0, \dots, 3 \tag{4.23}$$

where $p_{i,a}$ is a probability and K some constant. Convert $\boldsymbol{\ell}^n$ into probability distributions $\boldsymbol{p}^n$. What value do you find for K?

6. For $\text{SNR}_{\text{dB}} = 5$ and $-4 \leq y \leq 4$, plot the distributions $P_{M|Y}(i|y)$ as function of y for $i = 0, 1, 2, 3$. For the same values of y, also plot the probabilities calculated from the log probabilities output by the NN demapper trained for $\text{SNR}_{\text{dB}} = 5$. How well does the NN demapper approximate the analytical probabilities?

Problem 4.3. (Log probabilities versus probabilities)

1. Repeat steps 3., 4., and 6. of Problem 4.2 for the demapper in Figure 4.6 terminated as in Figure 4.3, using the cross equivocation cost (4.7). Do you observe differences to the results from Problem 4.2?

2. Repeat steps 3., 4., and 6. of Problem 4.2 for the demapper in Figure 4.6 terminated as in Figure 4.4, using the cross equivocation cost (4.7). Do you observe differences to the results from Problem 4.2?

5. Bitwise Demapper

5.1. Binary Classification

$$\begin{tikzcd}
B \in \{0,1\} \arrow[r] & \boxed{\text{Mapper}} \arrow[r, "X"] & \boxed{\text{Channel}} \arrow[r, "Y"] & \boxed{\text{Demapper}} \arrow[r] & Q_{B|Y}(1|Y)
\end{tikzcd}$$

Figure 5.1.: Binary message transmitted over a channel. The demapper outputs the probability $Q_{B|Y}(1|Y)$, from which $Q_{B|Y}(0|Y) = 1 - Q_{B|Y}(1|Y)$ can be calculated. Thus, effectively, the demapper provides us with the distribution $Q_{B|Y}(\cdot|Y)$ on the binary message set $\{0, 1\}$.

We now consider the special case when the channel input alphabet $\mathcal{X}$ contains two symbols, i.e., if $|\mathcal{X}| = 2$. In this case, the demappers we developed in the last chapter would have two outputs. However, this is redundant, since one probability can be calulated from the other, as they sum to one. Therefore, in the binary case, we usually use a demapper with a single output. Following the common practice in the machine learning community, we let the binary demapper output the probability that the transmitted bit was 1, i.e.,

$$P_1 = p_1(Y) = Q_{B|Y}(1|Y). \tag{5.1}$$

(This choice is arbitrary, outputting the accordingly defined probability P_0 would work equally well).

5.1.1. Log Probability Ratio

In the log domain, binary distributions can be defined conveniently by the log probability ratio (LPR)

$$L = \log \frac{P_1}{P_0}. \tag{5.2}$$

Using that $P_0 + P_1 = 0$, we can recover the probabilities from L by

$$P_1 = \frac{1}{1 + e^{-L}} = \sigma(L) \tag{5.3}$$

$$P_0 = \frac{1}{1 + e^{L}} = \sigma(-L). \tag{5.4}$$

Thus, the logistic function evaluated in L provides us P_1 and the logistic function evaluated in $-L$ provides us P_0.

5.1.2. Hard Decision and Bit Error Rate

Having defined the I/O specification of a binary demapper, it is in place to take a closer look at how to interprete the output provided to us by the demapper. Suppose the demapper makes a hard decision $\hat{B}$, based on its observation Y of the channel output. We would like to minimize the probability of wrong decision and equivalently, we decide for the most probable bit value, i.e.,

$$\hat{B} = \arg\max_{b \in \{0,1\}} P_{B|Y}(b|Y) \tag{5.5}$$

which is an instance of the MAP rule. In terms of P_1, the decision rule is

$$\hat{B} = \begin{cases} 1, & \text{if } P_1 \geq 0.5 \\ 0, & \text{if } P_1 < 0.5. \end{cases} \tag{5.6}$$

In terms of L, the rule becomes

$$\begin{aligned} \hat{B} &= \begin{cases} 1, & \text{if } L \geq 0 \\ 0, & \text{if } L < 0 \end{cases} && (5.7) \\ &= \frac{1}{2}(1 + \operatorname{sign} L). && (5.8) \end{aligned}$$

Thus, the hard decision is encoded in the sign of L. Furthermore, the decision is correct with probability $P_{B|Y}(\hat{B}|Y)$, so that the bit error rate (BER) is

$$\begin{aligned} \text{BER} &= \mathbb{E}[\min\{P_{B|Y}(1|Y), 1 - P_{B|Y}(1|Y)\}] && (5.9) \\ &\approx \mathbb{E}[\min\{P_1, 1 - P_1\}] && (5.10) \end{aligned}$$

where we used $\approx$, because P_1 output by the demapper is only an estimate of $P_{B|Y}(1|Y)$. In terms of L, we can write the BER as

$$\begin{aligned} \text{BER} &\approx \mathbb{E}[\min\{\sigma(L), \sigma(-L)\}] && (5.11) \\ &= \mathbb{E}[\sigma(-|L|)]. && (5.12) \end{aligned}$$

Thus, the BER is encoded in the amplitude of L, in other words, $|L|$ reflects how confident the demapper is about it's decision. In (5.14), we can replace $\approx$ by $=$ only if the demapper output is exact, i.e., if $L = \log \frac{P_{B|Y}(1|Y)}{P_{B|Y}(0|Y)}$.

We summarize.

1. The hard decision of the binary demapper is encoded in the sign of the LPR L, i.e.,

$$\hat{B} = \frac{1}{2}(1 + \operatorname{sign}(L)). \tag{5.13}$$

2. The confidence of the binary demapper about it's decision is encoded in the amplitude of the LPR L, i.e.,

$$\text{BER} \approx \mathbb{E}[\sigma(-|L|)]. \tag{5.14}$$

3. To assess the BER for performance evaluation, we should *not* use (5.14), because of it's dependence on how exact the demapper is. Instead, we should resort to the empirical expectation

$$\hat{\text{BER}} = \frac{1}{n}\sum_{i=0}^{n-1} \mathbb{1}(b_i \neq \hat{b}_i) \tag{5.15}$$

where $\mathbb{1}(\texttt{True}) = 1$ and $\mathbb{1}(\texttt{False}) = 0$. We should use (5.15) whenever the actually transmitted sequence b^n is available, which is the case, e.g., in the design phase. In the machine learning community, BER is often called *accuracy*.

5.2. Binary Cross Equivocation Loss

We now want to specify NNs for binary classification that output the probability P_1 and the LPR L, respectively, together with the appropriate loss functions for training.

5.2.1. Binary NN Demapper Outputting Probabilities

As discussed above, for a binary demapper, it is sufficient to output the probability estimate that the transmitted message is equal to 1. Thus, the requirements are

1. Output a single value.

2. The output must represent a probability, i.e., a value between 0 and 1.

The first requirement is achieved by a layer with a single linear neuron outputting the intermediate value $\ell \in [-\infty, \infty]$, and the second requirement is achieved by terminating the NN by a logistic activation. Thus, the output of the NN is

$$p_1 = \sigma(\ell) = \frac{1}{1+e^{-\ell}}. \tag{5.18}$$

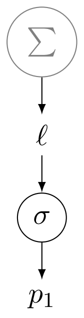
The BCE loss is

$$\text{bce}(b, p_1) = \begin{cases} -\log p_1, & \text{if } b = 1 \\ -\log(1-p_1), & \text{if } b = 0. \end{cases} \tag{5.19}$$

The loss can be compactly written as

$$\text{bce}(b, p_1) = -b\log p_1 - (1-b)\log(1-p_1). \tag{5.20}$$

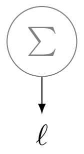
We summarize in Figure 5.2.



- NN is terminated by a linear layer followed by a logistic activation.
- Output p_1 is the probability that the message took the value 1.
- For training, use binary cross equivocation (BCE) defined on the probability of 1

$$\text{bce}(b, p_1) = \begin{cases} -\log p_1, & \text{if } b = 1 \\ -\log(1 - p_1), & \text{if } b = 0 \end{cases} \tag{5.16}$$

Figure 5.2.: Binary NN demapper outputting probability p_1.



- NN is terminated by a linear layer.
- Output is the LPR $\ell = \log p_1/p_0$.
- For training, use BCE defined on the LPR

$$\text{bce}_{\log}(b, \ell) = \begin{cases} -\log \frac{e^{\ell}}{e^{\ell}+1}, & \text{if } b = 1 \\ -\log \frac{1}{e^{\ell}+1}, & \text{if } b = 0. \end{cases} \tag{5.17}$$

Figure 5.3.: NN demapper outputting LPR $\ell = \log p_1/p_0$.

5.2.2. Binary NN Demapper Outputting Log Probability Ratios

Let's consider again the loss (5.19). By (5.3) and (5.4), we have

$$\begin{aligned}
\operatorname{bce}(1, p_1) &= -\log p_1 = -\log \sigma(\ell) && (5.21)\\
&= -\log \frac{1}{1+e^{-\ell}} && (5.22)\\
\operatorname{bce}(0, p_1) &= -\log(1-p_1) = -\log p_0 = -\log \sigma(-\ell) && (5.23)\\
&= -\log \frac{1}{1+e^{\ell}}. && (5.24)
\end{aligned}$$

Thus, we can use an NN terminated by one linear neuron outputting $\ell \in [-\infty, \infty]$ and train it by the BCE on LPR loss, which we define as

$$\operatorname{bce}_{\log}(b, \ell) = \begin{cases} -\log \sigma(\ell), & \text{if } b = 1\\ -\log \sigma(-\ell), & \text{if } b = 0. \end{cases} \tag{5.25}$$

The loss can be compactly written as

$$\operatorname{bce}_{\log}(b, \ell) = \log(1 + e^{(1-2b)\ell}). \tag{5.26}$$

5.2.3. Binary Cross Equivocation Lower Bound

Suppose bit B is transmitted over the channel, the demapper observes channel output Y and outputs LPR L. By Section 4.1, the BCE cost for sequences b^n and ℓ^n is lower bounded as

$$\begin{aligned}
\operatorname{BCE}_{\log}(b^n, \ell^n) &= \frac{1}{n} \sum_{i=0}^{n-1} \operatorname{bce}_{\log}(b_i, \ell_i) && (5.27)\\
&\gtrapprox \log(2) \cdot \mathbb{H}(B|Y) && (5.28)
\end{aligned}$$

that is, the BCE cost in bits is lower bounded by the equivocation $\mathbb{H}(B|Y)$, and the better the demapper approximates the exact LPR $\log \frac{P_{B|Y}(1|y)}{P_{B|Y}(0|y)}$, the closer the BCE cost gets to the equivocation $\mathbb{H}(B|Y)$.

5.3. Binary Labels

We now consider message sets $\{0, 1, \ldots, M-1\}$ that contain more than 2 messages, i.e., $M > 2$. By using

$$m = \lceil \log_2 |M| \rceil \tag{5.29}$$

bits, we can define a binary label of the message set. For instance, a natural approach is to label message a with the binary representation of a, which is instantiated in Table 5.1 for the case when $M = 8$. The question is now what a good label could be. Intuitively,

Message	0	1	2	3	4	5	6	7
natural label	000	001	010	011	100	101	110	111

Table 5.1.: Natural binary label of a message set.

Message	0	1	2	3	4	5	6	7
Gray label	000	001	011	010	110	111	101	100

Table 5.2.: Binary Gray label of a message set.

Symbol	-7	-5	-3	-1	1	3	5	7
Gray label	000	001	011	010	110	111	101	100

Table 5.3.: Binary Gray label of a real alphabet.

for two messages that are close, we would like the corresponding labels to be close as well. Considering the natural label in Table 5.1, we see that while the messages 1 and 2 differ by 1, their labels 001 and 010 differ in two bits. We next define Gray labels, which do not have this issue.

5.3.1. Gray Label for Message Sets

A binary Gray label assigns to neighboring messages labels that differ in only 1 bit. In Table 5.2, a binary Gray label for $M = 8$ is displayed.

5.3.2. Gray Label for Real Alphabets

Gray labels can also be defined for alphabets of real numbers. For real numbers, we say that two numbers r, s are close when their distance $|r - s|$ is small. For a set of real numbers $\mathcal{X}$, we define the neighbor(s) of r as the symbols in $\mathcal{X}$ that are closest to r, i.e.,

$$\text{neighbor}(r) = \mathop{\arg\min}_{\substack{s\in\mathcal{X}\\ s\neq r}} |s - r| \tag{5.30}$$

A binary Gray label now has the property that labels of neighbors differ in 1 bit. In Table 5.3, we display a Gray label for a channel input alphabet with 8 real symbols. As we will see in Problem 5.5, by using a Gray label on the channel input alphabet and bitwise demapping, we virtually lose nothing compared to symbolwise demapping as specified in the previous chapter. In a later chapter, we will revisit the problem of labeling channel input alphabets. By defining an arbitrary label and then learning the alphabet, we will recover the optimality of Gray labels for bitwise demapping.

5.3.3. Gray Code Algorithm

Gray labels are not unique. Here, we use Gray label as synonym for binary reflected Gray codes (BRGCs), which are defined by a recursive procedure. We list an implementation in python in Figure 5.4

```python
import numpy as np

def graylabel(m):
    """
    Calculate binary reflected Gray code (BRGC) for m bits.
    """
    if m == 1:
        return np.array([[0], [1]], dtype='uint8')
    else:
        label = graylabel(m -1)
        half_1 = np.hstack((np.zeros((2**(m - 1), 1), dtype='uint8'),
                            label))
        half_2 = np.hstack((np.ones((2**(m - 1), 1), dtype='uint8'),
                            np.flipud(label)))
        return np.vstack((half_1, half_2))
```

Figure 5.4.: Algorithm to calculate BRGC.

$$\begin{tikzcd} \boldsymbol{B} \in \{0,1\}^m \arrow[r] & \boxed{\text{Mapper}} \arrow[r, "X"] & \boxed{\text{Channel}} \arrow[r, "Y"] & \boxed{\text{Demapper}} \arrow[r] & \boldsymbol{L} \end{tikzcd}$$

Figure 5.5.: Bitwise demapper outputting a vector $\boldsymbol{L} = L_0 L_1 \dots L_{m-1}$ of LPRs, one for each of the bits in the binary label $\boldsymbol{B} = B_0 B_1 \dots B_{m-1}$.

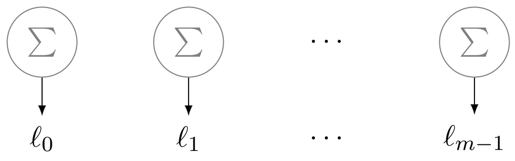


Figure 5.6.: Termination of a bitwise NN demapper outputting m LPRs.

5.4. Bitwise Cross Equivocation Loss

We now consider the setup in Figure 5.5. The message is represented by a binary label $\boldsymbol{B}$ consisting of m bits and the demapper outputs m LPRs, one for each bitlevel in the binary label. In Figure 5.11, we show the termination of an NN demapper outputting m LPRs.

Consider now sample sequence $\boldsymbol{b}^n$ and $\boldsymbol{\ell}^n$, where for $i = 0, \dots, n-1$, $\boldsymbol{b}_i$ is a transmitted binary label of m bits and where $\boldsymbol{\ell}_i$ is the vector of m LPRs output by the demapper. The bitwise cross equivocation loss is

$$\text{bce}_{\log}(\boldsymbol{b}, \boldsymbol{\ell}) = \frac{1}{m} \sum_{j=0}^{m-1} \text{bce}_{\log}(b_j, \ell_j) \tag{5.31}$$

and the bitwise cross equivocation cost is

$$\mathrm{BCE}_{\log}(\boldsymbol{b}, \boldsymbol{\ell}) = \frac{1}{nm} \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} \mathrm{bce}_{\log}(b_{ij}, \ell_{ij}). \tag{5.32}$$

The outer sum averages over the n samples and the inner sum averages over the m bitlevels.

5.4.1. Bitwise Cross Equivocation Lower Bound

By Section 4.1, we can lower bound the bitwise cross equivocation similar to (5.28). We have

$$\begin{aligned} \mathrm{BCE}_{\log}(\boldsymbol{b}, \boldsymbol{\ell}) &= \frac{1}{nm} \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} \mathrm{bce}_{\log}(b_{ij}, \ell_{ij}) && (5.33)\\ &\gtrapprox \frac{1}{m} \sum_{j=0}^{m-1} \mathbb{H}(B_j|Y) && (5.34) \end{aligned}$$

that is, the bitwise cross equivocation cost is lower bounded by the average of the equivocations $\mathbb{H}(B_j|Y)$, $j = 0, \dots, m-1$, and the better the demapper outputs L_j approximate the exact LPRs $\log \frac{P_{B_j|Y}(1|Y)}{P_{B_j|Y}(0|Y)}$, the closer the bitwise cross equivocation loss approaches the lower bound.

5.5. Bitwise NN Demapper for AWGN

We now define a bitwise NN demapper for additive white Gaussian noise (AWGN) channels. We consider amplitude shift keying (ASK) constellations (input alphabets) parametrized as

$$\mathcal{X}_{2^m\mathrm{ASK}} = \{\pm 1, \pm 3, \dots, (\pm 2^m - 1)\} \tag{5.35}$$

where $m = \log_2 |\mathcal{X}|$. For instance,

$$\mathcal{X}_{4\mathrm{ASK}} = \{-3, -1, 1, 3\}. \tag{5.36}$$

For the remainder of this chapter, we assume ASK constellations labelled by Gray codes.
This parametrization may look restrictive, however

- In the next chapter, we will combine equalization and demapping, so that the equalizer prepares the received signal such that a demapper assuming an ASK constellation can take over.
- In a later chapter, we will consider transmitter optimization, where we will learn constellations from scratch.

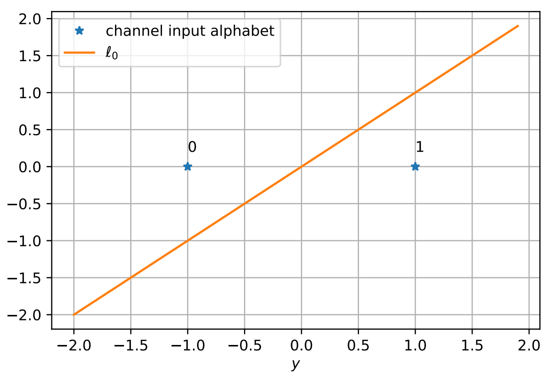


Figure 5.7.: Binary demapper.

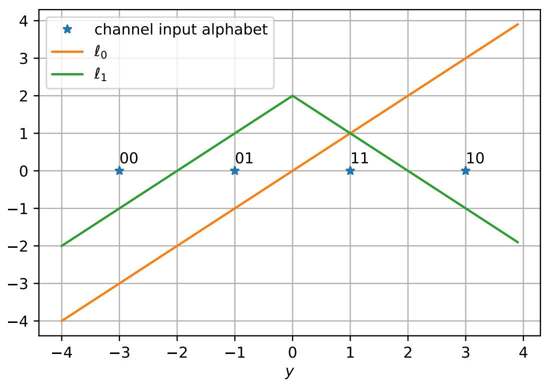


Figure 5.8.: Bitwise demapper for two bits labeling 4 symbols.

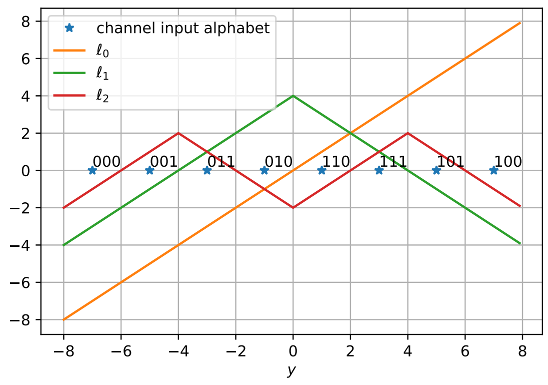


Figure 5.9.: Bitwise demapper for three bits labeling 8 symbols.

5.5.1. Bit 0

We start by considering the binary demapper graphically. In Figure 5.7, the channel input alphabet $\{-1, 1\}$ with the binary label $\{0, 1\}$ is displayed, together with the log probability $\ell_0 = \log p_1/p_0$. For $y < 0$, the symbol -1 is closest to the channel output observation y, and correspondingly, $\ell_0 < 0$, indicating $p_1 < p_0$. At $y = 0$, both symbols are equally distant from y and $\ell_0 = 0$, indicating $p_1 = p_0$. Finally, for $y > 1$, we have $\ell_0 > 0$, which indicates that $p_1 > p_0$.

5.5.2. Bit 1

In Figure 5.8, we display a bitwise demapper for 2 bits Gray labelling 4 symbols. The demapper output ℓ_0 for the first bitlevel b_0 is basically equal to the output of the binary demapper we just discussed, the reason is that for $y < 0$, symbols with $b_0 = 0$ are closest and for $y > 0$, symbols with $b_0 = 1$ are closest. We now would like to define a corresponding demapper output ℓ_1 for the second bitlevel b_1. The symbol -3 to the left has $b_1 = 0$, then the two symbols $\{-1, 1\}$ in the middle have $b_1 = 1$, and the symbol 3 on the right again has again $b_1 = 0$. Thus, ℓ_1 should be negative to the left and right and positive in the middle. The displayed ℓ_1 has these properties and is piecewise linear.

We now devise an efficient way to calculate both ℓ_0 and ℓ_1 using function composition.

$$\ell_0(y) = y \tag{5.37}$$

$$\ell_1(\ell_0) = -|\ell_0| + 2. \tag{5.38}$$

We make two observations.

1. While ℓ_0 plays the role of the function *value* in (5.37), it acts as a function *argument* in (5.38). This is the key principle of defining functions by composition.

2. The absolute value $|\cdot|$ acts like a mirror, i.e., the negative argument values are reflected on the horizontal axis.

5.5.3. Bit 2

To illustrate the power of function composition further, we now consider 3 bits Gray labeling 8 symbols. The three curves in Figure 5.9 can be calculated as

$$\ell_0(y) = y \tag{5.39}$$

$$\ell_1(\ell_0) = -|\ell_0| + 4 \tag{5.40}$$

$$\ell_2(\ell_1) = -|\ell_1| + 2. \tag{5.41}$$

For better understanding how the calculation of ℓ_2 works, we display in Figure 5.10 ℓ_2 as function of ℓ_2, as function of ℓ_1, and as function of ℓ_0. Note that in Figure 5.9, the number of linear sections of $\ell_i(y)$ doubles with i: $\ell_0(y)$ is a straight line and has 1 section, $\ell_1(y)$ has 2 sections and $\ell_2(y)$ has 4 sections. This is an example of how powerful function composition can be, and directly relates to the "deep" in deep learning, namely the use of many layers, which corresponds to the composition of many functions.

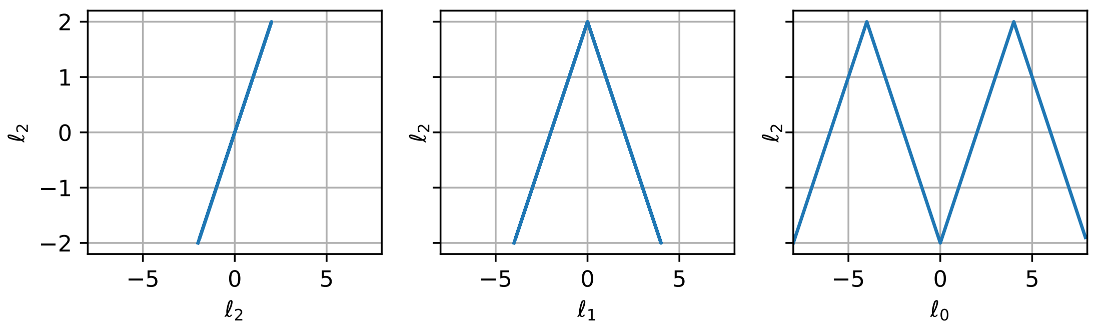
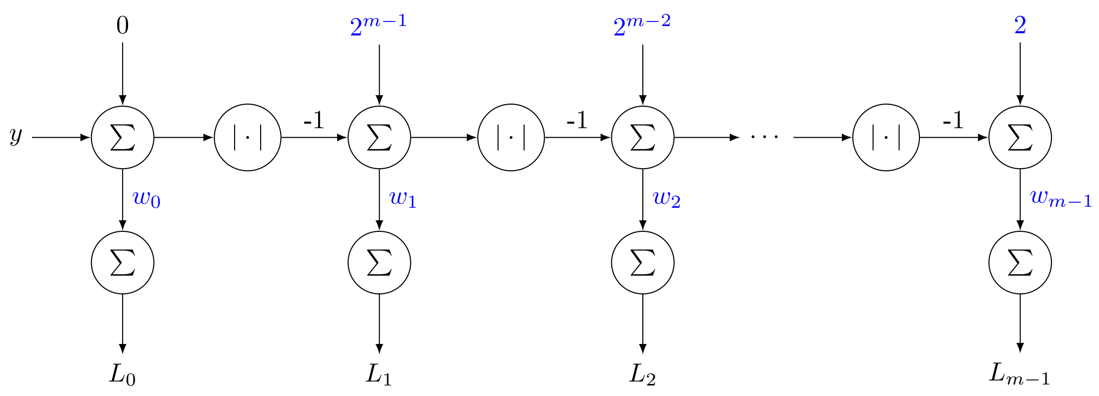


Figure 5.10.: The LPR ℓ_2 as function of the LPRs ℓ_2, ℓ_1, and ℓ_0.

5.5.4. Complete Demapper

In Figure 5.11, a general bitwise NN demapper for AWGN is displayed that works for Gray labelled ASK constellations with 2^m symbols.



https://gitlab.lrz.de/gb/mlcomm:

```
from mlcomm import demapper

dm = demapper.BitwiseNNDemapper(num_bitlevels=m)
```

Figure 5.11.: Bitwise NN demapper for Gray labelled ASK constellations in AWGN. The parameters in blue are trainable.

5.6. Problems

Problem 5.1. (Binary cross equivocation loss) Verify the compact forms of the BCE loss (5.20) and (5.26).

Problem 5.2. (Optimality of Binary NN Demapper for AWGN) Consider an AWGN channel

$$Y = X + Z \tag{5.42}$$

with binary phase shift keying (BPSK) input X uniformly distributed on $\{-1, 1\}$, Gaussian noise $Z \sim \mathcal{N}(0, \sigma_Z^2)$. Assume the binary label $B = b(X)$ with $b(-1) = 0$ and $b(1) = 1$.

1. Show that

   $$\ell = \log \frac{P_{B|Y}(1|y)}{P_{B|Y}(0|y)} \tag{5.43}$$

   is given by

   $$\ell = \lambda \cdot y. \tag{5.44}$$

   What is the value of λ?

2. For SNR $= 0, 1, 2, 3, 4, 5$, train the binary NN demapper in Figure 5.3. Compare the weight w after training with the value of λ that you calculated in 1.

Problem 5.3. (Function composition) Use the function compositions developed in Section 5.5 to plot Figure 5.9.

Problem 5.4. (ReLU activation) Design an NN using linear neurons and ReLU activation that for input y outputs $a \cdot |y| + b$.

Problem 5.5. (Bitwise NN demapper) Consider the Gray labeled channel input alphabet

symbol	-3	-1	1	3
label	00	01	11	10

1. For SNR $= 5, 6, \ldots, 10$, train the bitwise NN demapper from Figure 5.11.

2. Plot the LPRs $\ell_0(y)$ and $\ell_1(y)$ output by the trained NN demapper for $y \in [-4, 4]$.

3. For the trained NN demapper, estimate the sum of the bitwise equivocations

   $$\mathbb{H}(B_0|Y) + \mathbb{H}(B_1|Y) \quad \text{[bits]}. \tag{5.45}$$

   How does it compare to the equivocation achieved by the symbolwise NN demapper from Problem 4.2?

6. Equalization in Random Noise

In this chapter, we bring together the signal reconstruction by equalization considered in Chapter 3, and the message classification by demapping considered in Chapters 4 and 5.

$$\begin{tikzcd} {} \arrow[r, "A"] & \boxed{\text{Mapper}} \arrow[r, "X"] & \boxed{\text{Channel}} \arrow[r, "Y"] & \boxed{\text{Equalizer } f} \arrow[r, "f(Y)"] & \boxed{\text{Demapper}} \arrow[r, "L"] & {} \end{tikzcd}$$

Recall that in Chapter 3, we considered the transmission of a real signal X over a channel. The receiver observed the channel output Y representing a distorted version of X. We then devised an equalizer f with the aim to reconstruct X from Y. We considered the case where the residual error

$$X - f(Y) \tag{6.1}$$

is virtually zero, i.e., when the equalizer can reconstruct the transmitted signal from the channel output. We called this scenario equalization of deterministic impairements and we used as criterion to train our equalizer the MSE loss

$$\text{mse}(x, f(y)) = |x - f(y)|^2. \tag{6.2}$$

In this chapter, we consider the case when the residual error (6.1) is non-zero and random, e.g., because the channel impairement is in part random. Thus, the reconstruction $\hat{X} = f(Y)$ at the equalizer output will be a noisy version of X, and it will not be deterministic, i.e., transmitting the same signal twice may result in different reconstructions.

The fundamental question is now

$$\text{What is a good reconstruction } f(Y) \text{ for random impairements?} \tag{6.3}$$

The key observation is that, in the end, the purpose of a communication system is to reconstruct the transmitted message A. Based on this observation, we will develop two approaches to answer question (6.3):

Approach 1: We concatenate the equalizer f with a demapper proxy outputting a soft decision L. A good equalizer reconstructs the transmitted signal so that when this reconstruction is fed to the demapper, the soft decision L output by the demapper minimizes the equivocation of message A given L.

Approach 2: We don't even try to reconstruct the transmitted signal. Instead, we directly aim for outputting a soft decision L that minimizes the equivocation of message A given L, effectively integrating equalization and classification in one device.

Most of this chapter discusses a toy example in detail, namely a BPSK signal that is impaired by AWGN and then attenuated.

6.1. Equalization of Attenuated BPSK Signal

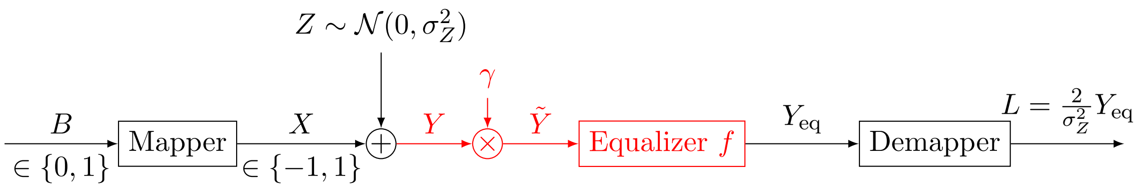


Figure 6.1.: Bit B is mapped to the BPSK signal X and impaired by AWGN. The noisy signal is attenuated by γ. An equalizer reconstructs the signal and a demapper outputs a soft decision on the transmitted bit, given the equalizer output Y_{eq}.

We consider the setup in Figure 6.1. A binary message B is mapped to a BPSK signal $X \in \{-1, 1\}$. The signal is corrupted by additive noise and the noisy signal is attenuated by γ, i.e.,

$$\tilde{Y} = \gamma(X + Z) \tag{6.4}$$

is received, where $Z \sim \mathcal{N}(0, \sigma^2)$, i.e., Z is zero mean Gaussian with variance σ^2 and where γ is a positive scalar. We will now proceed in two steps.

1. We set $\gamma = 1$, i.e., we assume the absence of attenuation, and we derive the optimal binary demapper.

2. We use the demapper derived in step 1. and let γ take an arbitrary positive value. We then insert an equalizer between channel and demapper whose purpose is to compensate the attenuation γ, so as to guarantee optimal functioning of the demapper.

The central finding of this section will be that

- MSE $\mathbb{E}[|X - Y_{\text{eq}}|^2]$ is not the appropriate equalizer criterion in this scenario.

- The equalizer should minimize the equivocation of the message B given the demapper output L.

6.1.1. Optimal Demapper in the Absence of Attenuation

We start by considering the case when there is no attenuation ($\tilde{Y} = Y$) and no equalizer ($Y_{\text{eq}} = \tilde{Y}$). By Problem 5.2, we know that the optimal demapper outputs

$$L = \frac{2}{\sigma_Z^2} Y. \tag{6.5}$$

We fix the demapper to

$$\ell(y_{\text{eq}}) = \frac{2}{\sigma_Z^2} y_{\text{eq}} \tag{6.6}$$

and since $Y_{\text{eq}} = \tilde{Y} = Y$, this demapper is optimal in the absence of attenuation and explicit equalization.

6.1.2. Optimal Equalizer

We now let γ take an arbitrary non-zero value. Informally, to ensure that demapper (6.6) is optimal, we require

$$y_{\text{eq}} = y \tag{6.7}$$

$$\Rightarrow f_{\text{optimal}}(\tilde{y}) = y \tag{6.8}$$

$$\Rightarrow f_{\text{optimal}}(\tilde{y}) = \frac{1}{\gamma} \tilde{y} \tag{6.9}$$

so the optimal equalizer is linear and scales it's input by $1/\gamma$, fully compensating the attenuation.

Formal Derivation Formally, let's write the demapper output as probabilities

$$Q_{B|Y_{\text{eq}}}(1|y_{\text{eq}}) = \sigma\left(\frac{2}{\sigma_Z^2} y_{\text{eq}}\right), \quad Q_{B|Y_{\text{eq}}}(0|y_{\text{eq}}) = \sigma\left(-\frac{2}{\sigma_Z^2} y_{\text{eq}}\right). \tag{6.10}$$

The corresponding probabilities output by the optimal demapper are

$$P_{B|Y}(1|y) = \sigma\left(\frac{2}{\sigma_Z^2} y\right), \quad P_{B|Y}(0|y) = \sigma\left(-\frac{2}{\sigma_Z^2} y\right). \tag{6.11}$$

Also, we have

$$Y_{\text{eq}} = f(\tilde{Y}) = f(\gamma Y). \tag{6.12}$$

Thus, by the information inequality, we have

$$\mathbb{E}\left[-\log Q_{B|Y_{\text{eq}}}(B|f(\gamma Y))\right] \geq \mathbb{E}\left[-\log P_{B|Y}(B|Y)\right] \tag{6.13}$$

with equality if and only if

$$Q_{B|Y_{\text{eq}}}(1|f(\gamma y)) = P_{B|Y}(1|y) \tag{6.14}$$

$$\Leftrightarrow \sigma\left(\frac{2}{\sigma_Z^2} f(\gamma y)\right) = \sigma\left(\frac{2}{\sigma_Z^2} y\right) \tag{6.15}$$

$$\Leftrightarrow f(\gamma y) = y \tag{6.16}$$

$$\Leftrightarrow f(\tilde{y}) = \frac{1}{\gamma}\tilde{y}. \tag{6.17}$$

Thus, the optimal equalizer (6.9) is recovered when we minimize the left-hand side of (6.13), which is effectively the CE loss. For binary message sequences b^n, channel output $\tilde{y}^n$, and demapper output ℓ^n, the BCE cost is

$$\text{BCE}_{\log}(b^n \ell^n) = \frac{1}{n}\sum_{i=0}^{n-1} \text{bce}_{\log}(b_i, \ell_i) \tag{6.18}$$

$$= \frac{1}{n}\sum_{i=0}^{n-1} \text{bce}_{\log}\left(b_i, \frac{2}{\sigma_Z^2} f(\tilde{y}_i)\right). \tag{6.19}$$

and we obtain the optimal equalizer by minimizing the last term over f. Note that we made no restrictions on f, in particular, we did not require it to be linear.

6.1.3. Linear MSE Equalizer

We now consider a linear equalizer minimizing the MSE

$$\mathbb{E}[(f(\tilde{Y}) - X)^2]. \tag{6.20}$$

A linear equalizer calculates $f(\tilde{y}) = \alpha\tilde{y}$. We thus look for the α that minimizes the MSE, i.e., we calculate the derivative w.r.t. α and set it equal to zero. By Problem 6.1

$$\begin{aligned} f_{\text{linear, mse}}(\tilde{y}) &= \frac{1}{\gamma(1+\sigma_Z^2)}\tilde{y} \\ &= \frac{1}{\gamma(1+\frac{1}{\mathsf{snr}})}\tilde{y} \end{aligned} \tag{6.21}$$

which is different from the optimal equalizer (6.9). With increasing signal-to-noise ratio (SNR), the linear MSE equalizer approaches the optimal equalizer.

6.1.4. Non-Linear MSE Equalizer

Suppose now that we impose no restriction on the equalizer f, in particular, we do not restrict the equalizer f to be linear. In this case, for each value of $\tilde{y}$, we can choose $f(\tilde{y})$ so that the MSE is minimized. Conditioned on $\tilde{Y} = \tilde{y}$, we write the MSE as

$$\mathbb{E}[(f(\tilde{y}) - X)^2|\tilde{Y} = \tilde{y}] = P_{X|\tilde{Y}}(-1|\tilde{y})(\underbrace{f(\tilde{y})}_{=:a}+1)^2 + P_{X|\tilde{Y}}(1|\tilde{y})(\underbrace{f(\tilde{y})}_{=a}-1)^2. \tag{6.22}$$

We can now minimize the MSE by calculating the derivative of the right-hand side w.r.t. a and setting it equal to zero. By Problem 6.1, the MSE is minimized by

$$a = 2 \cdot P_{X|\tilde{Y}}(1|\tilde{y}) - 1. \tag{6.23}$$

By Problem 6.1, we can use (5.3) to express $P_{X|\tilde{Y}}(1|\tilde{y})$ in terms of $\tilde{y}/\gamma$ to finally obtain

$$f_{\text{non-linear, mse}}(\tilde{y}) = \tanh\left(\mathsf{snr}\frac{\tilde{y}}{\gamma}\right). \tag{6.24}$$

6.1.5. Discussion

1. In Figure 6.2, we plot the I/O characteristics of the optimal, the linear MSE, and the non-linear MSE equalizers for $\gamma = 0.7$ and $\text{SNR} = 7\,\text{dB}$.

2. In Figure 6.3, we plot the SNR after equalization for $\gamma = 0.7$ against the SNR after optimal equalization. Comparing to Figure 6.4, we see that the SNR after MSE equalization cannot be used for predicting the signal quality in terms of information content. In particular, the SNR after non-linear MSE equalization highly overestimates the signal quality.

3. In Figure 6.4, we plot the equivocation in bits for the three considered equalizers for $\gamma = 0.7$. For high SNR, the linear MSE equalizer does not lose much compared to the optimal equalizer, while the non-linear MSE equalizer is more than 1 dB SNR worse than the optimal equalizer. The reason is that it removes soft information from the received signal, i.e., the tanh characteristics of the non-linear MSE equalizer acts rather like a hard-decision than a soft-decision.

6.2. Demapper Proxies for Equalizer Design

6.2.1. Symbolwise Demapping

In Figure 6.5, we display a symbolwise demapper proxy for equalizer training.

6.2.2. Bitwise Demapping

In Figure 6.6, we display a bitwise demapper proxy for equalizer training.

6.3. Integrated Equalization and Demapping

Alternative to using an equalizer with a single output representing a signal reconstruction followed by a demapper proxy, we can integrate equalization and demapping. In this case, the equalizer input is processed by one NN, which outputs the demapper output. In particular, there is no explicit signal reconstruction, so the integrated NN may not have an intermediate layer with a single neuron representing the reconstructed signal.

The integrated equalizer and demapper should be trained as a demapper, i.e., the symbolwise CE or the bitwise BCE loss should be used for training.

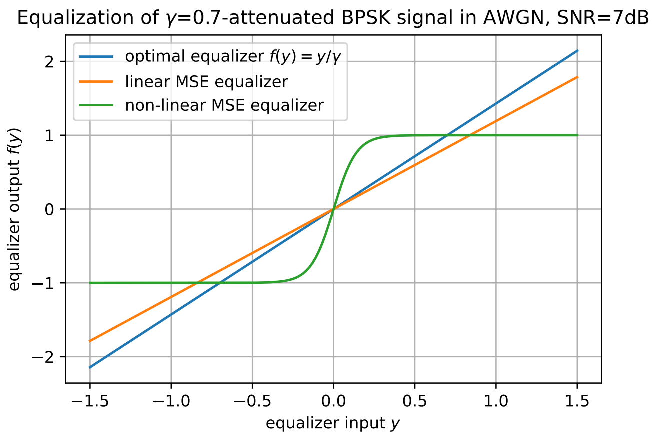


Figure 6.2.: Equalizer characteristics at 7 dB SNR.

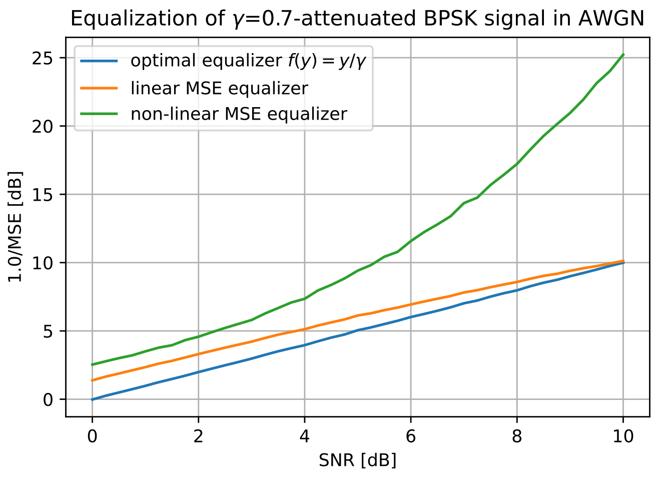


Figure 6.3.: The SNR estimate 1/MSE after equalization versus the SNR after optimal equalization.

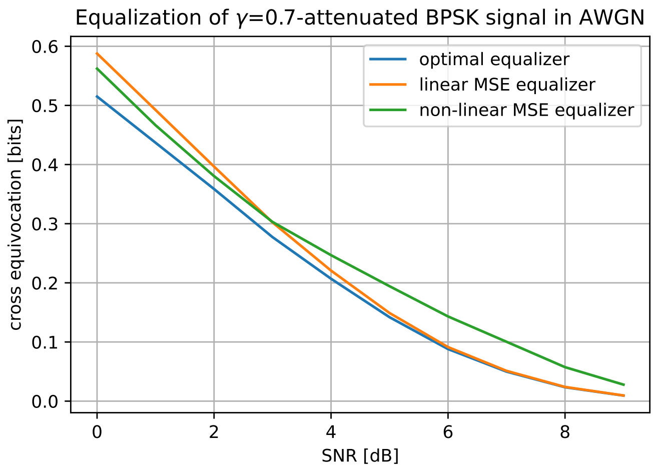
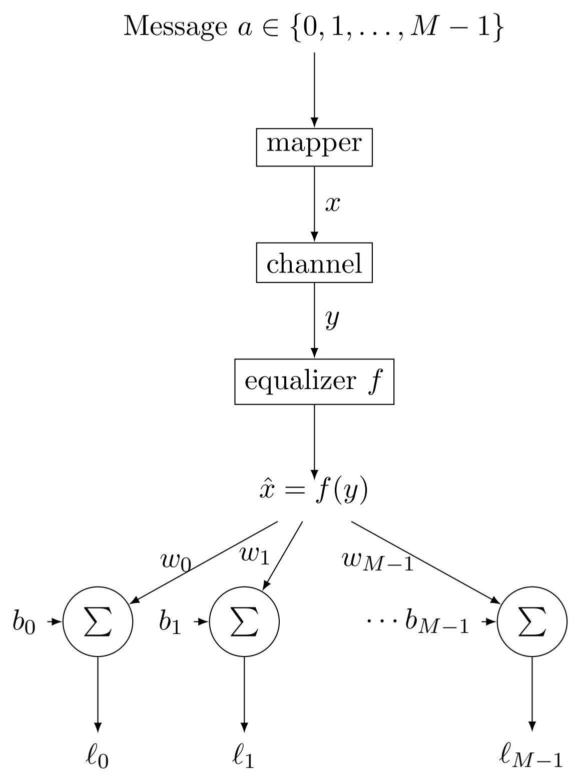


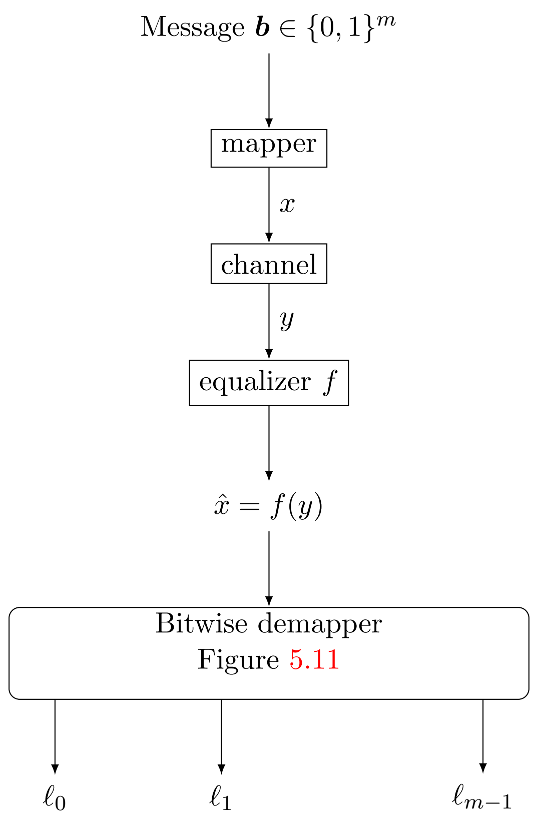
Figure 6.4.: Cross equivocation versus SNR.



Train the equalizer f by minimizing the CE loss

$$\mathrm{ce}_{\log}(a, \boldsymbol{\ell}) = -\log \frac{e^{\ell_a}}{\sum_{j=0}^{M-1} e^{\ell_j}} \tag{6.25}$$

Figure 6.5.: Symbolwise demapper proxy for equalizer training.



Train the equalizer f by minimizing the BCE loss

$$\mathrm{bce}_{\log}(\boldsymbol{b}, \boldsymbol{\ell}) = \frac{1}{m} \sum_{j=0}^{m-1} \mathrm{bce}_{\log}(b_j, \ell_j) \tag{6.26}$$

$$= \frac{1}{m} \sum_{j=0}^{m-1} \log(1 + e^{(1-2b_j)\ell_j}) \tag{6.27}$$

Figure 6.6.: Bitwise demapper proxy for equalizer training.

6.4. Further Reading

In [1], a non-linear equalizer based on Volterra series is trained w.r.t. the BCE loss via a demapper proxy based on the max-log approximation (MLA). Also, a NN integrating equalization and bitwise demapping is considered and trained w.r.t. the BCE loss.

References

[1] M. Schädler, G. Böcherer, and S. Pachnicke, “Soft-demapping for short reach optical communication: A comparison of deep neural networks and Volterra series,” *J. Lightw. Technol.*, vol. 39, no. 10, pp. 3095–3105, 2021.

6.5. Problems

Problem 6.1. (MSE equalizers for attenuated BPSK in AWGN)

1. Show that for channel (6.4), the optimal linear MSE equalizer is given by (6.21).
2. Show that for channel (6.4), the optimal non-linear MSE equalizer is given by (6.24).

Problem 6.2. (NN Equalizers for attenuated BPSK in AWGN) In this problem, you train various equalizers for channel (6.4) with $\gamma = 0.7$ and you compare the theoretical findings of Section 6.1 to the input-output characteristics of trained NN equalizers.

1. **Linear MSE equalizer:** implement a linear NN equalizer f consisting of a single linear neuron and train it w.r.t. the MSE loss $|X - f(\tilde{Y})|$. Plot the I/O function f of the trained NN and compare it to (6.21).
2. **Non-linear MSE equalizer:** implement a non-linear NN equalizer f consisting of several hidden layers with non-linear activations and train it w.r.t. the MSE loss $|X - f(\tilde{Y})|$. Plot the I/O function f of the trained NN and compare it to (6.24).
3. **Optimal equalizer:** Use the same non-linear NN as in 2. as equalizer. Concatenate it with a single linear neuron demapper whose parameters you fix according to Section 6.1.1. Train the NN equalizer across the demapper proxy w.r.t. the BCE $\mathrm{bce}_{\log}(B, L)$. Plot the I/O function f of the trained NN and compare it to (6.9).
4. Use the NN equalizer from 1.–3. concatenated with the single linear neuron demapper fixed according to Section 6.1.1 to verify the curves in Figure 6.4.

Part II.

Transmitter

7. Mapper

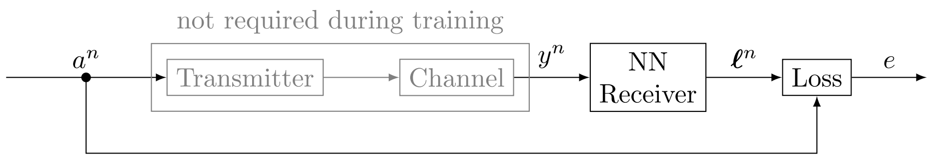


Figure 7.1.: Receiver training

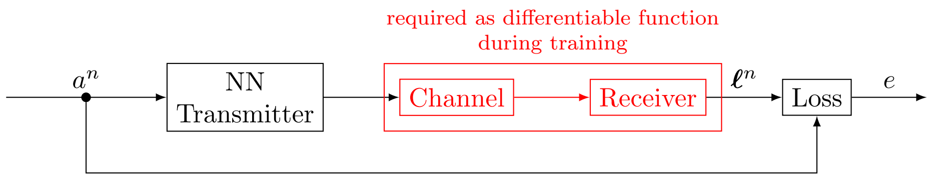


Figure 7.2.: Transmitter training.

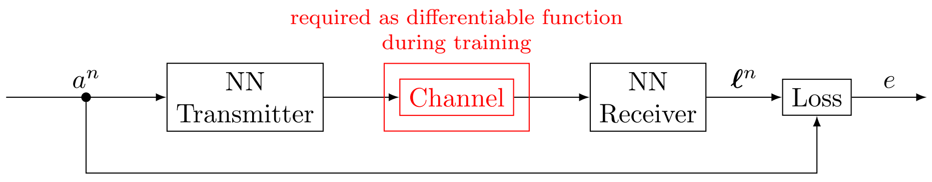


Figure 7.3.: End-to-end training.

In Part 1, we considered the design of receiver components. For training, it was sufficient to have a message sequence a^n and an observation y^n of the channel output, as illustrated in Figure 7.1. These two sequences can be recorded prior to training so that during training, there is no need to access transmitter and channel. In particular, message and channel output can be measured in a lab experiment and receiver training can then be done offline.

In contrast, for transmitter training, the loss gradient needs to be backpropagated all the way through receiver and channel to the transmitter, so that the transmitter parameters can be updated. Therefore, channel and receiver must be available as differentiable functions during training. This is illustrated in Figure 7.2. The preparation of a differentiable channel model, possibly by training based on experimental measurements, is a challenging task of its own, which we will consider in a later chapter.

For now, we simply assume the existence of a differentiable channel model and focus our attention on training the transmitter. In fact, we will go a step further and consider

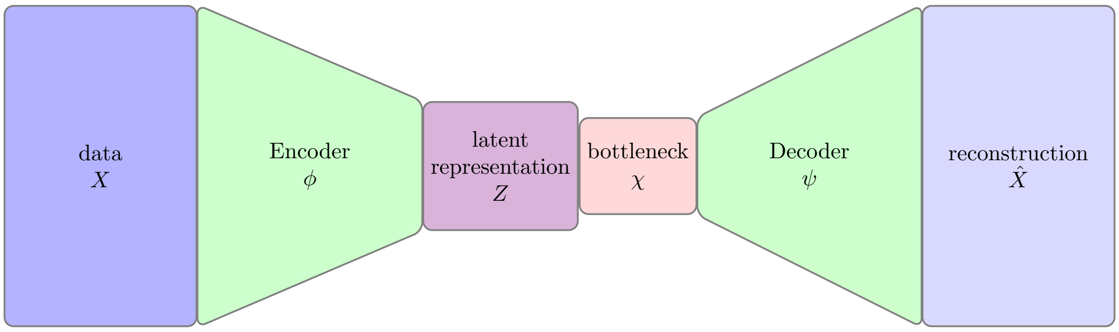


Figure 7.4.: Autoencoder.

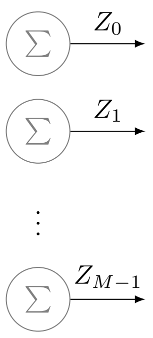
end-to-end training, i.e., we will train transmitter and receiver jointly. The reason for doing so is the following: when designing the receiver, we assumed specific transmitter configurations like symbol constellations and binary labels as fixed and we then designed the receiver for this specific transmitter, which, in turn, results in a specific receiver. Thus, to gain insights about how optimal transmitters may look like, we also need to optimize the receiver.

We will start by introducing the autoencoder, which is widely applied in machine learning. Closely following the autoencoder paradigm, we will then discuss the mapper design, i.e., the mapping of messages to channel input symbols. We will discuss mappers for two different message representations, namely the one-hot representation and the binary representation.

7.1. Autoencoder

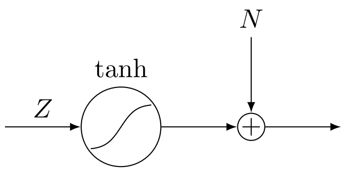
We display the basic architecture of an autoencoder in Figure 7.4. Data X is encoded by ϕ to a latent representation Z, which is passed through a bottleneck χ. The bottleneck output is decoded by ψ to $\hat{X}$, a reconstruction of X. Encoder ϕ (and thereby latent representation Z) and decoder ψ are chosen jointly. Thus, without the bottleneck, encoder and decoder would simpy copy their input to the output, so that $\hat{X}$ is a perfect reconstruction of X. With the bottleneck, the encoder needs to compress the data to a latent representation that is robust w.r.t. the bottleneck, so that the decoder can reconstruct the original data X from the bottleneck output. The bottleneck appears in many variations. In the following, we will discuss a few bottleneck examples relevant for communications.

7.1.1. Dimensionality Reduction Bottleneck



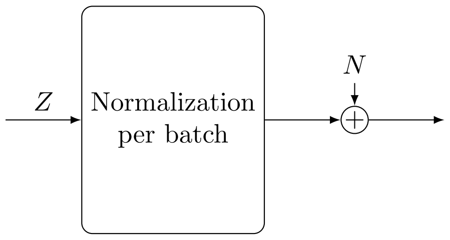
Suppose data X is an N-dimensional vector. Dimensionality reduction corresponds to enforcing an $M < N$ dimensional latent representation Z. In a NN, the dimensionality reduction bottleneck can be realized by a layer with M output units, as displayed above. We will discuss this in Section 7.3, where we consider the case when encoder ϕ and decoder ψ are linear functions, which is called principal component analysis (PCA).

7.1.2. Amplifier Saturation Bottleneck



An ideal amplifier would scale a signal x by some scalar α, i.e., it would output $\alpha \cdot x$, independent of the amplitude of x. For physical reasons, this is impossible in practice. Instead, the output amplitude is bounded by some maximum value. Typically, the amplifier characteristics is linear for low amplitudes and becomes non-linear and saturates for large amplitudes, with a smooth transition from the linear region to saturation. This amplifier saturation can be modelled by the tanh function. With the tanh function alone, the encoder could scale the signal down to the linear region and the decoder could scale the signal up again, so that the tanh would effectively not present any real bottleneck. By scaling down the signal, the tanh output would effectively have low power, which is impractical. To enforce the encoder to use the full range of the amplifier, we therefore add Gaussian noise N with a fixed variance to the amplifier output. As a result, we have two competing impairments: a low power signal is linearly amplified, and the dominant impairment is the additve noise, while for a high power signal, the non-linear amplification is the dominant impairment. Thus, optimizing encoder and decoder also balances between non-linear amplifier impairment and noise corruption.

7.1.3. SNR Bottleneck



Communication systems are frequently subject to an average power constraint P. As displayed in the diagram above, such average power constraint can be implemented in an NN by a normalization layer. Since stochastic gradient descent (SGD) works on batches, we consider the average power per batch, i.e., for a batch size n_{batch}, the normalization layer outputs

$$\text{normalization}(z^{n_{\text{batch}}})_i = z_i \cdot \sqrt{\frac{P}{\sum_{i=0}^{n_{\text{batch}}-1} |z_i|^2}}. \tag{7.1}$$

Similar to the amplifier saturation, this normalization has effectively no effect if the signal is not corrupted by noise. Therefore, after normalization, we add Gaussian noise with variance σ^2. A convenient parametrization of the noise variance is via the SNR by

$$\mathsf{snr} = \frac{P}{\sigma^2} \Rightarrow \sigma^2 = \frac{P}{\mathsf{snr}}. \tag{7.2}$$

7.2. Mapper

The task of the mapper is to map a message $a \in \mathcal{A} = \{0, 1, \dots, M-1\}$ to a real symbol $x(a)$. In the autoencoder framework, the mapper plays the role of an encoder ϕ and the real symbol plays the role of the latent representation. We have considered mappers before, for instance, we used the ASK constellation

$$\mathcal{X} = \{\pm 1, \pm 3, \dots, \pm(M-1)\} \tag{7.3}$$

indexed by the message a. The mapping to ASK symbols can also be realized by an affine mapping from $\mathcal{A}$ to $\mathcal{X}$ by

$$a \mapsto 2 \cdot a - M + 1. \tag{7.4}$$

For instance, for $M = 4$, the mapping (7.4) yields

a	0	1	2	3
$2 \cdot a - 3 = x(a)$	-3	-1	1	3

Consider now the slightly modified constellation

$$\mathcal{X} = \{-4, -1, 1, 4\}. \tag{7.5}$$

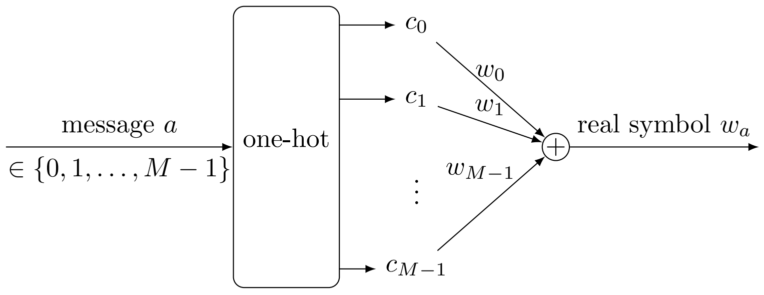


Figure 7.5.: Symbol mapper using one hot representation followed by a linear neuron. This mapper is suitable for training end-to-end w.r.t. CE loss on the message and the demapper output.

Let's try to map a message $a \in \{0,1,2,3\}$ to $\mathcal{X}$ by the affine mapping $a \mapsto \alpha \cdot a + \beta$. We have

$$\alpha \cdot 0 + \beta = -4 \Rightarrow \beta = -4 \tag{7.6}$$

$$\alpha \cdot 1 - 4 = -1 \Rightarrow \alpha = 3 \tag{7.7}$$

$$3 \cdot 2 - 4 = 2 \ \lightning \tag{7.8}$$

This system of equation has no solution! As we want to learn the constellation $\mathcal{X}$, we need a mapping from messages to real symbols that can realize *any* real constellation $\mathcal{X}$. We will next present two such mappings, one that is suitable for symbolwise demapping and one that is suitable for bitwise demapping.

7.2.1. Symbol Mapper for One-Hot Message Representations

To easily realize a mapping from messages to arbitrary real constellations, we represent the message $a \in \mathcal{A} = \{0,1,2,\dots,M-1\}$ by the one-hot representation

$$\boldsymbol{c} = c_0 c_1 \dots c_{M-1} \tag{7.9}$$

which is a vector with M entries, which are all zero, except for one, which is equal to 1 ("hot"). The message value determines which entry is hot, i.e.,

$$c_i(a) = \mathbb{1}(i = a) = \begin{cases} 1, & \text{if } i = a \\ 0, & \text{if } i \neq a \end{cases} \tag{7.10}$$

For instance, for $M = 4$, the one-hot representation is given in the following table.

message a	one-hot representation $\boldsymbol{c}$
0	1000
1	0100
2	0010
3	0001

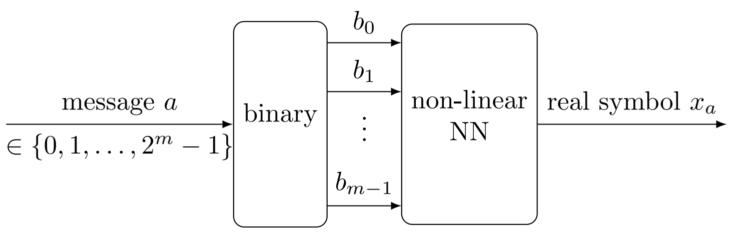


Figure 7.6.: Bit mapper using binary representation followed by a non-linear NN. The bit mapper is suitable for training end-to-end w.r.t. BCE loss on the message bits and the output of a bitwise demapper.

A symbol mapper now consists of a one-hot representation followed by a linear neuron as displayed in Figure 7.5. By Problem 7.1, the symbol mapper can map to any real constellation and is suitable for learning constellations by end-to-end training w.r.t. the CE loss, see, e.g., Problem 7.2.

7.2.2. Mapper for Binary Message Representations

Suppose now we want to use a bitwise demapper and the BCE loss for learning a constellation by end-to-end training. We represent message $a \in \mathcal{A} = \{0, 1, 2, \ldots, 2^m - 1\}$ by m bits $\boldsymbol{b} = b_0 b_1 \ldots b_{m-1}$. By Problem 7.1, to map the binary representation to an arbitrary constellation $\mathcal{X}$, we need a non-linear NN with m inputs and 1 output. If the non-linear NN is sufficiently rich, we can learn any constellation of real signal points, see also Problem 7.3. We display the bit mapper in Figure 7.6.

7.3. Principal Component Analysis

PCA considers dimensionality reduction of data by a linear mapping, followed by linear reconstruction, with the aim to minimize the MSE between the data and its reconstruction. PCA is a linear autoencoder with dimensionality reduction as bottleneck. Specifically, consider data consisting of n samples of N dimensional row vectors, i.e.,

$$\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_{n-1}, \quad \boldsymbol{x}_i \in \mathbf{R}^N. \tag{7.11}$$

The encoder ϕ maps sample $\boldsymbol{x}$ to an $M < N$ dimensional vector $\boldsymbol{z}$ by an affine mapping, i.e.,

$$\boldsymbol{z} = \boldsymbol{x}\boldsymbol{R} + \boldsymbol{r}, \quad \boldsymbol{R} \in \mathbf{R}^{N \times M}. \tag{7.12}$$

The decoder ψ calculates from $\boldsymbol{z}$ a reconstruction $\hat{\boldsymbol{x}}$ by an affine mapping, i.e.,

$$\hat{\boldsymbol{x}} = \boldsymbol{z}\boldsymbol{Q} + \boldsymbol{q}, \quad \boldsymbol{Q} \in \mathbf{R}^{M \times N}. \tag{7.13}$$

The goal is to minimize the MSE between data and reconstruction, i.e.,

$$\min_{\boldsymbol{Q},\boldsymbol{R},\boldsymbol{q},\boldsymbol{r}} \sum_{i=0}^{n-1} \|\boldsymbol{x}_i - [(\boldsymbol{x}_i \boldsymbol{R} + \boldsymbol{r})\boldsymbol{Q} + \boldsymbol{q}]\|_2^2. \tag{7.14}$$

PCA can be realized by a linear NN with two linear layers. The first linear layer plays the role of the encoder ϕ and has N inputs and M outputs, which correspond to the latent representation of reduced dimensionality. The second layer plays the role of the decoder ψ and has M inputs and N outputs. Matrix $\boldsymbol{R}$ and vector $\boldsymbol{r}$ are weights and bias of the encoding layer, respectively; $\boldsymbol{Q}$ and $\boldsymbol{q}$ are weights and bias, respectively, of the decoding layer. In Problem 7.4, we consider a simple example to illustrate the principle of PCA.

7.4. Further Reading

The work [1] promotes the idea of interpreting a communication system as an autoencoder that can be trained end-to-end. In [2, Figure 4], symbol- and bit mappers for complex signal points subject to an SNR constraint are trained end-to-end using CE and BCE loss, respectively. BCE loss trained bit mappers for 1D peak power constraints and 2D average power constraints are presented in [3]. PCA is treated in detail, e.g., in [4, Chapter 12]. In [5, Chapter 8], PCA using Scikit-Learn is discussed.

References

[1] T. O'Shea and J. Hoydis, "An introduction to deep learning for the physical layer," *IEEE Trans. Cognitive Commun. Netw.*, vol. 3, no. 4, pp. 563–575, 2017.

[2] S. Cammerer, F. A. Aoudia, S. Dörner, M. Stark, J. Hoydis, and S. Ten Brink, "Trainable communication systems: Concepts and prototype," *IEEE Trans. Commun.*, vol. 68, no. 9, pp. 5489–5503, 2020.

[3] M. Schädler, S. Calabrò, F. Pittalà, G. Böcherer, M. Kuschnerov, C. Bluemm, and S. Pachnicke, "Neural network assisted geometric shaping for 800Gbit/s and 1Tbit/s optical transmission," in *Proc. Optical Fiber Conf. (OFC)*, 2020.

[4] C. M. Bishop, *Pattern recognition and machine learning*. Springer, 2006.

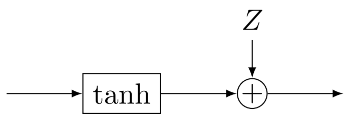
[5] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019.

7.5. Problems

Problem 7.1. (One hot and binary representations) Consider a mapper that maps a message $a \in \mathcal{A} = \{0, 1, \dots, M-1\}$ with $M = 2^m$ to a constellation $\mathcal{X} = \{x_0, x_1, \dots, x_{M-1}\}$, i.e., $a \mapsto x_a$. The constellation is arbitrary and fixed.

1. Let $\boldsymbol{c} = c_1 c_2 \dots c_{M-1}$ be a one-hot representation of message a, i.e., $c_i(a) = \mathbb{1}(i = a)$. Show that a linear neuron with M inputs can realize the mapping $\boldsymbol{c}(a) \to x_a$, $a \in \mathcal{A}$, for any constellation $\mathcal{X}$.
2. For $M = 8$, sample the symbols in $\mathcal{X}$ uniformly at random from the interval $[-8, 8]$ and train a linear neuron that realizes $\boldsymbol{c}(a) \mapsto x_a$ using the MSE loss.
3. Let $\boldsymbol{b} = b_0 b_1 \dots b_{m-1}$ be a binary representation of message a. Identify a constellation $\mathcal{X}$ for which a linear neuron with m inputs cannot realize the mapping $\boldsymbol{b}(a) \to x_a$.
4. For the input alphabet you identified in 3., train a linear unit with m inputs using the MSE loss to verify that the mapping $\boldsymbol{b}(a) \to x_a$ can indeed not be realized.
5. For the input alphabet from 3., devise a non-linear NN with m inputs that approximates the mapping $\boldsymbol{b}(a) \to x_a$ well w.r.t. the MSE loss.

Problem 7.2. (Amplifier saturation) In this problem, we consider the amplifier saturation bottleneck, as discussed in Section 7.1.2. Consider the channel



where Z is zero mean Gaussian noise.

1. Implement the channel as differentiable function supporting `torch.autograd` using functions provided by `torch`.
2. Use a linear neuron with M inputs and 1 output as transmitter and a linear neuron with 1 input and M outputs as receiver. Sample a message $a \in \{0, 1, \dots, M-1\}$ and use its one-hot representation $\boldsymbol{c}$ as transmitter input. Train transmitter and receiver w.r.t. the cross equivocation $\text{ce}(a, \boldsymbol{\ell})$, where $\boldsymbol{\ell}$ is the receiver output.
3. Plot the learned channel input alphabet and compare it to the scatterplot of the channel output.

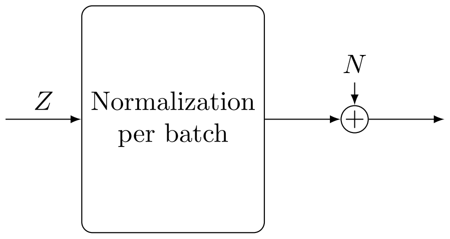
Problem 7.3. (SNR Constraint) In this problem, we consider an SNR constraint as introduced in Section 7.1.3. Specifically, we consider a complex signal, which we represent by a real signal where two successive real symbols are the inphase and quadrature components, respectively, of one complex symbol, i.e.,

$$x_{2i} x_{2i+1} \leftrightarrow c_i = x_{2i} + j \cdot x_{2i+1}. \tag{7.15}$$

We normalize the average power of the signal, i.e., for a batch size of n_{batch} complex symbols, we impose

$$\frac{1}{2n_{\text{batch}}} \sum_{i=0}^{2n_{\text{batch}}-1} x_i^2 = P. \tag{7.16}$$

After normalization, we add zero mean Gaussian noise. The resulting bottleneck is



The task is now to learn a 2D constellation with 16 complex symbols.

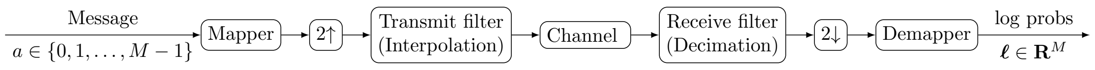
1. Implement the normalization layer followed by AWGN as differentiable function in `pytorch`.
2. For a one-hot representation of the message, implement a linear unit with $M = 16$ inputs and 2 outputs as a mapper at the transmitter, and a linear unit with 2 inputs and $M = 16$ outputs as demapper at the receiver.
3. Jointly train mapper and demapper w.r.t. CE loss for SNR equal to $0, 2, \dots, 10$ dB and plot the learned constellations.
4. For a binary representation of the message, implement a non-linear NN with $m = 4$ inputs and 2 outputs as mapper at the transceiver and a non-linear NN with 2 inputs terminated by a linear layer with $m = 4$ outputs as demapper at the receiver.
5. Jointly train mapper and demapper w.r.t. BCE loss on LPRs for SNR equal to $0, 2, \dots, 10$ dB and plot the learned constellations.
6. Verify if the learned bit mappings are Gray.

Problem 7.4. (PCA) Generate a data set consisting of $n = 10\,000$ vectors of dimension $N = 2$ by sampling from a joint Gaussian distribution with covariance matrix and mean vector given by

$$\boldsymbol{C} = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 2 \end{bmatrix}, \quad \boldsymbol{\mu} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}. \tag{7.17}$$

1. Implement a PCA autoencoder with dimensionality reduction bottleneck with $M = 1$. Train the NN w.r.t. to the MSE and scatterplot the data and its reconstruction, i.e., its principal component.
2. Compare to what you get when using `sklearn.decomposition.PCA` from the Scikit-Learn package.

8. Spectral Shaping



Message
$a \in \{0, 1, \dots, M-1\}$
Mapper
$2\uparrow$
Transmit filter (Interpolation)
Channel
Receive filter (Decimation)
$2\downarrow$
Demapper
log probs
$\boldsymbol{\ell} \in \mathbf{R}^M$

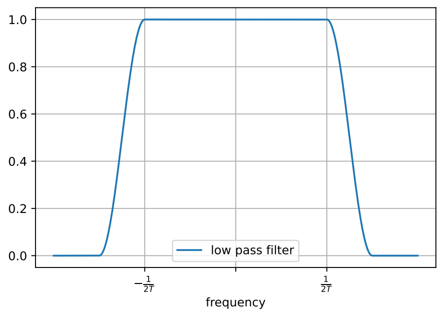
Figure 8.1.: System to learn end-to-end transmit filter, receive filter, and demapper, for making the transmitted signal fulfill the spectral requirements.

Practical communication systems are often subject to constraints on the frequency spectrum of the signal dissipated by the transmitter. For instance, only certain frequencies may be allowed for use. The transmitter must therefore spectrally shape the signal, so as to meet the spectral requirements. This is achieved in part in the analog frontend. To allow for the use of cheap analog hardware, the spectrum is fine-tuned before the digital-to-analog conversion (DAC) by first upsampling the digital signal and then using a digital interpolation filter, which shapes the signal spectrally. Correspondingly, at the receiver, a decimation filter corrects the signal spectrum, which is followed by downsampling and demapping.

In this chapter, we will first discuss examples of spectral requirements. We will then give a short summary of mathematical tools for spectral analysis, which we need for monitoring the filter training. Next, we will discuss the effects of upsampling, filtering, and downsampling in the spectral domain. Finally, we will discuss filter design by end-to-end training.

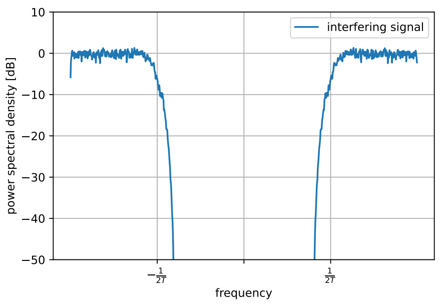
8.1. Spectral Requirements

8.1.1. Low Pass Filter



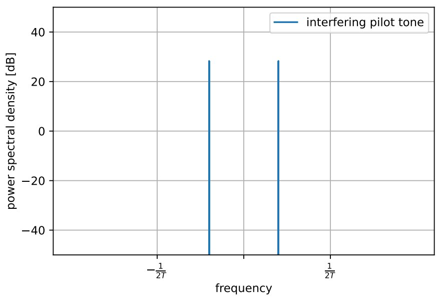
Suppose the transmitter's analog frontend acts like a low pass filter, which attenuates high frequencies. In this case, the digital transmit filter should place most of the signal's power in the low frequency regime, where the signal is not attenuated by the analog frontend.

8.1.2. Interference



Suppose some other transmissions is going on in the neighboring frequency bands. To cope with this interference, the digital transmit filter should avoid placing power in the neighboring frequency bands and instead allocate most of the signal power in the center part of the spectrum, where there is no interference.

8.1.3. Pilot Tone



Suppose some other device transmits a narrow band pilot tone. To cope with this pilot tone, the transmit filter should avoid allocating power to the pilot tone frequency, and the receive filter should filter out the pilot tone.

8.2. Spectral Analysis

8.2.1. Discrete Fourier Transform

The discrete Fourier transform (DFT) of a signal $\boldsymbol{x}$ of length n is

$$x_f(k) = \sum_{i=0}^{n-1} x(i) e^{-2\pi j \frac{ik}{n}}, \quad j = \sqrt{-1}, \quad k = 0, 1, \ldots, n-1. \tag{8.1}$$

The DFT is implemented, e.g., in `numpy.fft.fft` and `torch.fft.fft`. The inverse DFT is given by

$$x(i) = \frac{1}{n}\sum_{k=0}^{n-1} x_f(k) e^{2\pi j \frac{ki}{n}}, \quad i = 0, 1, \dots, n-1 \tag{8.2}$$

and it is implemented, e.g., by `numpy.fft.ifft` and `torch.fft.ifft`. We interpret $\boldsymbol{x}$ as a *time signal*, i.e., its entries $x_0, x_1, \dots, x_{n-1}$ are samples taken at different time instances. The DFT decomposes the time signal $\boldsymbol{x}$ into a sum of n tones

$$e^{2\pi j \frac{ki}{n}} = \cos\left(2\pi\frac{ki}{n}\right) + j\sin\left(2\pi\frac{ki}{n}\right), \quad k = 0, \dots, n-1. \tag{8.3}$$

The kth tone has frequency k/n cycles per sample (for other frequency units, see Section 8.2.4). The frequency coefficient $x_f(k)$ tells us how much the kth frequency is weighted in the time signal $\boldsymbol{x}$. Thus, $\boldsymbol{x}_f$ is a *frequency signal*.

8.2.2. Power Spectral Density

Recall that we defined the power of our time signal $\boldsymbol{x}$ as

$$P = \frac{1}{n}\sum_{i=0}^{n-1} |x_i|^2. \tag{8.4}$$

If we draw the samples x_i from a distribution P_X, then the empirical distribution in (8.4) approximates P_X and we have

$$P \approx \mathbb{E}(|X|^2). \tag{8.5}$$

We now want to know how the power in the signal $\boldsymbol{x}$ is distributed to the n frequencies. If the samples in the time signal are random variables X_i, then also the samples $X_f(k)$ in the frequency signal are random variables. Consequently, using probabilistic expectation, we can define the power spectral density (PSD) of $\boldsymbol{x}$ as

$$S_{xx}(k) = \frac{1}{n}\mathbb{E}[|X_f(k)|^2], \quad k = 0, 1, \dots, n-1. \tag{8.6}$$

The PSD and the signal power relate via

$$\mathbb{E}(|X|^2) = \sum_{k=0}^{n-1} S_{xx}(k). \tag{8.7}$$

For replacing the probabilistic expectation in (8.6) by an empirical expectation, we consider n_{seg} time signal segments $\boldsymbol{x}_0, \dots, \boldsymbol{x}_{n_{\text{seg}}-1}$, each of length n_{perseg}. For each segment, we calculate the DFT

$$\boldsymbol{x}_{f\ell} = \text{DFT}(\boldsymbol{x}_\ell), \quad \ell = 0, 1, 2, \dots, n_{\text{seg}} - 1. \tag{8.8}$$

We can now estimate the PSD via

$$S_{xx}(k) \approx \frac{1}{n_{\text{perseg}}} \underbrace{\frac{1}{n_{\text{seg}}} \sum_{\ell=0}^{n_{\text{seg}}-1} |x_{f\ell}(k)|^2}_{\approx \mathbb{E}[|X_f(k)^2|]}, \quad k = 0, 1, \ldots, n_{\text{perseg}} - 1. \tag{8.9}$$

Note that we considered in total $n = n_{\text{seg}} \cdot n_{\text{perseg}}$ samples. By using large n_{perseg} (few long segments), we get a high resolution in the frequency domain, while using large n_{seg} (many short segments) makes our PSD estimate more confident. Thus, for a fixed total number n of samples, we have to trade resolution for confidence. In Problem 8.1, we compare (8.9) to a refined version called Welch's method, which is implemented, e.g., in `scipy.signal.welch`.

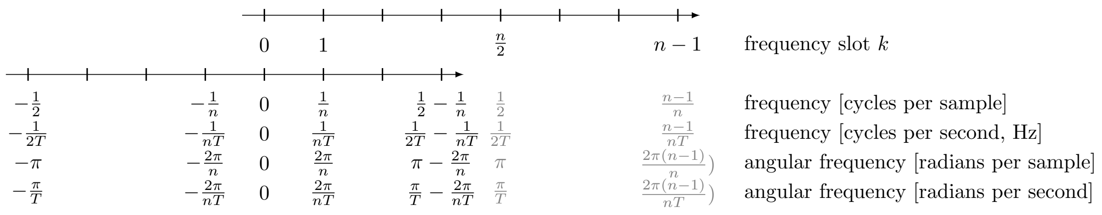
8.2.3. White Signals

A random signal $\boldsymbol{X} = X_0 X_1, \ldots, X_{n-1}$ is *white*, if the samples X_i are independent, have zero mean, and variance σ^2. Equivalently, the PSD is constant and given by

$$S_{xx}(k) = \frac{1}{n}\sigma^2, \quad k = 0, \ldots, n-1. \tag{8.10}$$

In Problem 8.1, we consider the PSD of white and colored (non-white) signals.

8.2.4. Frequency Units



0
1
$\frac{n}{2}$
$n-1$
frequency slot k
$-\frac{1}{2}$ $-\frac{1}{n}$ 0 $\frac{1}{n}$ $\frac{1}{2}-\frac{1}{n}$ $\frac{1}{2}$ $\frac{n-1}{n}$
frequency [cycles per sample]
$-\frac{1}{2T}$ $-\frac{1}{nT}$ 0 $\frac{1}{nT}$ $\frac{1}{2T}-\frac{1}{nT}$ $\frac{1}{2T}$ $\frac{n-1}{nT}$
frequency [cycles per second, Hz]
$-\pi$ $-\frac{2\pi}{n}$ 0 $\frac{2\pi}{n}$ $\pi-\frac{2\pi}{n}$ π $\frac{2\pi(n-1)}{n})$
angular frequency [radians per sample]
$-\frac{\pi}{T}$ $-\frac{2\pi}{nT}$ 0 $\frac{2\pi}{nT}$ $\frac{\pi}{T}-\frac{2\pi}{nT}$ $\frac{\pi}{T}$ $\frac{2\pi(n-1)}{nT})$
angular frequency [radians per second]

When plotting the spectrum calculated by the DFT (8.1), we often want to choose a more informative frequency unit than simply the frequency slot index k. Furthermore, as $k = 0$ corresponds to the frequency 0 cycles per sample, we often prefer to place frequency slot $k = 0$ in the middle of the displayed spectrum. By defining

$$k = k' \bmod n \tag{8.11}$$

we can choose any range of n successive integers for k', and we can then plot $x_f(k' \bmod n)$ against k'. Usually, the range $k' \in [-\frac{n}{2}, \frac{n}{2} - 1]$ is chosen. A convenient tool to facilitate the conversion of the frequency signal $\boldsymbol{x}_f$ to this range is implemented in `numpy.fft.fftshift` and `torch.fft.fftshift`.

$$\begin{tikzcd}
{} \arrow[r, "\boldsymbol{x} = x_0 x_1 \dots x_{n-1}"] & \boxed{2\uparrow} \arrow[r, "\boldsymbol{u} = u_0 u_1 \dots u_{2n-1}"] & \boxed{\text{Filter } \boldsymbol{h}} \arrow[r, "\boldsymbol{y} = y_0 y_1 \dots y_{2n-1}"] & \boxed{2\downarrow} \arrow[r, "\boldsymbol{d} = d_0 d_1 \dots d_{n-1}"] & {}
\end{tikzcd}$$

Figure 8.2.

8.3. Spectral Filter Properties

In this section, we discuss the spectral effects of upsampling, filtering, and downsampling in the system displayed in Figure 8.2. For an example signal, we display the spectra after each step in Figure 8.3.

8.3.1. Upsampling

$$\begin{tikzcd}
{} \arrow[r, "\boldsymbol{x} = x_0 x_1 \dots x_{n-1}"] & \boxed{2\uparrow} \arrow[r, "\boldsymbol{u} = u_0 u_1 \dots u_{2n-1}"] & {}
\end{tikzcd}$$

Upsampling by factor 2 inserts 0s between samples:

$$u(i) = \begin{cases} x(i/2) & \text{if } i \text{ is even} \\ 0 & \text{if } i \text{ is odd} \end{cases} \tag{8.12}$$

that is,

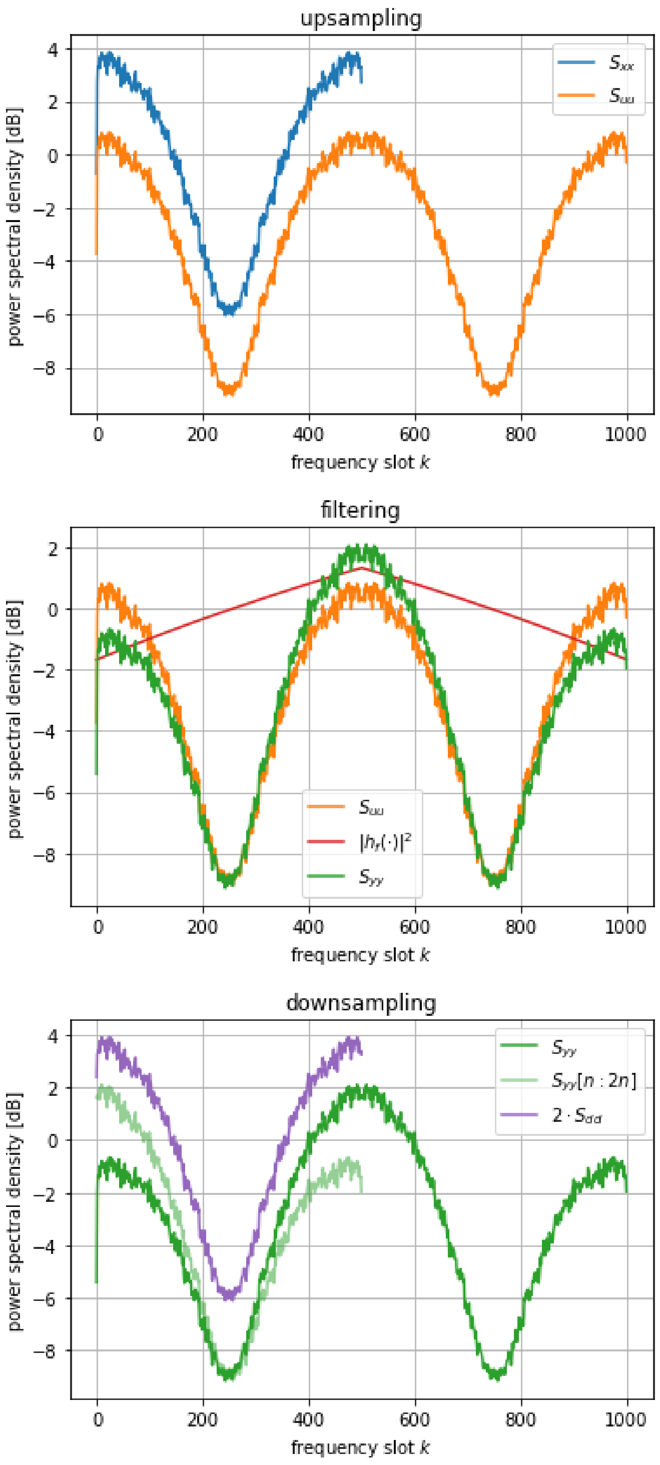
$$x_0 x_1 x_2 \dots \mapsto x_0 0 x_1 0 x_2 0 \dots \tag{8.13}$$

In frequency domain, upsampling by 2 corresponds to repeating the spectrum twice. This follows by

$$\begin{aligned} u_f(k) &= \sum_{i=0}^{2n-1} u(i) e^{-2\pi j \frac{ik}{2n}} && (8.14)\\ &= \sum_{\ell=0}^{n-1} u(2\ell) e^{-2\pi j \frac{2\ell k}{2n}} && (8.15)\\ &= \sum_{\ell=0}^{n-1} x(\ell) e^{-2\pi j \frac{\ell k}{n}} && (8.16)\\ &= \sum_{\ell=0}^{n-1} x(\ell) e^{-2\pi j \ell \frac{k \bmod n}{n}} && (8.17)\\ &= x_f(k \bmod n), \quad k = 0, \dots, 2n-1. && (8.18) \end{aligned}$$

8.3.2. Filtering

$$\begin{tikzcd}
{} \arrow[r, "\boldsymbol{u} = u_0 u_1 \dots u_{n-1}"] & \boxed{\text{Filter } \boldsymbol{h}} \arrow[r, "\boldsymbol{y} = y_0 y_1 \dots y_{n-1}"] & {}
\end{tikzcd}$$



upsampling
S_{xx}
S_{uu}
filtering
S_{uu}
$|h_f(\cdot)|^2$
S_{yy}
downsampling
S_{yy}
$S_{yy}[n:2n]$
$2 \cdot S_{dd}$
power spectral density [dB]
frequency slot k

Figure 8.3.

Suppose we have a signal $\boldsymbol{u}$ and a filter $\boldsymbol{h}$. The circular convolution of $\boldsymbol{u}$ and $\boldsymbol{h}$ is

$$(\boldsymbol{u} \star \boldsymbol{h})_i = \sum_{k=0}^{n-1} h_k u_{(i-k) \bmod n}, \quad i = 0, 1, \ldots, n-1. \tag{8.19}$$

Circular convolution in time domain corresponds to multiplication in frequency domain, i.e.,

$$y_f(k) = u_f(k) h_f(k), \quad k = 0, 1, \ldots, n-1. \tag{8.20}$$

The PSD of a filtered signal is given by

$$S_{yy}(k) = S_{uu}(k) |h_f(k)|^2, \quad k = 0, 1, \ldots, n-1. \tag{8.21}$$

8.3.3. Downsampling

$$\begin{tikzcd} {} \arrow[r, "\boldsymbol{y} = y_0 y_1 \ldots y_{2n-1}"] & \boxed{2\downarrow} \arrow[r, "\boldsymbol{d} = d_0 d_1 \ldots d_{n-1}"] & {} \end{tikzcd}$$

Downsampling corresponds to dropping the odd samples:

$$d(i) = y(2i), \quad i = 0, 1, \ldots, n-1. \tag{8.22}$$

In frequency domain, the spectrum $\boldsymbol{d}_f$ is the sum of spectrum $\boldsymbol{y}_f$ from 0 to $n-1$ and $\boldsymbol{y}_f$ from n to $2n-1$, i.e.,

$$d_f(k) = \frac{1}{2}[y_f(k) + y_f(k+n)], \quad k = 0, 1, \ldots, n-1. \tag{8.23}$$

This can be seen by the following derivation:

$$\begin{aligned}
d(i) &= y(2i) && (8.24)\\
&= \frac{1}{2n} \sum_{k=0}^{2n-1} y_f(k) e^{2\pi j \frac{2ik}{2n}} && (8.25)\\
&= \frac{1}{2n} \left[\sum_{k=0}^{n-1} y_f(k) e^{2\pi j \frac{ik}{n}} + \sum_{k=0}^{n-1} y_f(k+n) e^{2\pi j \frac{i(k+n)}{n}} \right] && (8.26)\\
&= \frac{1}{2n} \left[\sum_{k=0}^{n-1} y_f(k) e^{2\pi j \frac{ik}{n}} + \sum_{k=0}^{n-1} y_f(k+n) e^{2\pi j \frac{ik}{n}} \right] && (8.27)\\
&= \frac{1}{n} \sum_{k=0}^{n-1} \frac{1}{2}[y_f(k) + y_f(k+n)] e^{2\pi j \frac{ik}{n}}, \quad i = 0, \ldots, n-1. && (8.28)
\end{aligned}$$

8.3.4. Intersymbol Interference

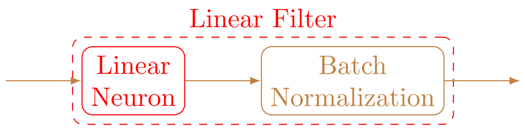
Suppose now the samples of a random signal $\boldsymbol{X} = X_0 X_1, \ldots, X_{n-1}$ have zero mean and variance σ^2, however, the PSD is not constant, i.e., the signal is not white. By Section 8.2.3, this implies that the samples are not independent, which is often called inter symbol interference (ISI).

The PSD provides us with a simple test for the presence of ISI. Suppose the transmitted signal $\boldsymbol{x}$ in Figure 8.2 is white. Then, if the PSD of the reconstructed signal $\boldsymbol{d}$ at the receiver is not constant, then we know that $\boldsymbol{d}$ is corrupted by ISI. In other words, a constant PSD is a necessary condition for the reconstruction of white signals.

8.4. Filter Design by End-to-End Training

In this section, we discuss the end-to-end training w.r.t. the CE loss of a communication system as in Figure 8.1 in presence of a low pass filter as introduced in Section 8.1.1.

8.4.1. Filter



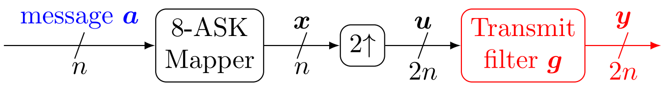
```
class LinearFilter(torch.nn.Module):
    def __init__(self, num_taps):
        super().__init__()
        self.conv = torch.nn.Conv1d(in_channels=1,
                                    out_channels=1,
                                    kernel_size=num_taps,
                                    padding='same',
                                    bias=False)
    def forward(self, y):
        y = self.conv(y.reshape(1, 1, -1)).reshape(-1)
        return 1 / torch.sqrt(torch.mean(y**2)) * y
```

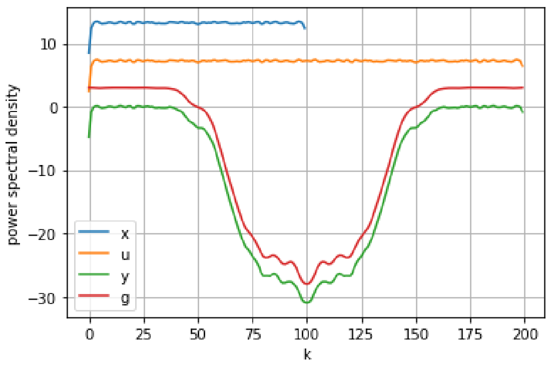
At transmitter and receiver, we consider linear filters with output normalized to power 1. As we have several trainable devices in our transceiver chain, the output normalization helps to avoid arbitrarily scaled intermediate signals, which would be difficult to interprete. The filter implementation is differentiable, and by passing

`LinearFilter.parameters()`

to the optimizer, the linear neuron is trainable. Note that the bias is set to zero, to avoid direct current (DC) components in the output signal.

8.4.2. Transmitter

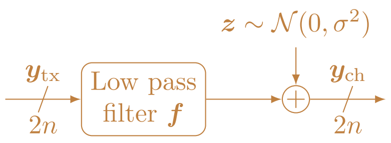


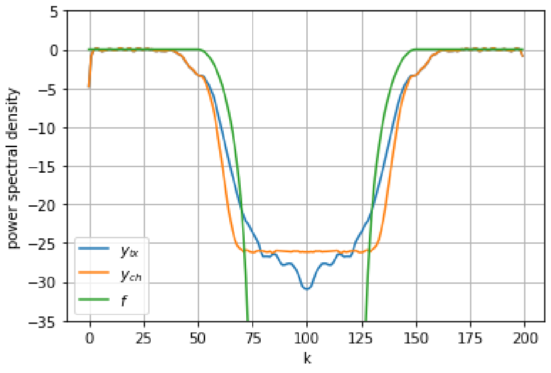


We consider an 8-ASK mapper that maps the message $a \in \{0, 1, \dots, 7\}$ to the 8-ASK constellation $\mathcal{X} = \{\pm 1, \pm 3, \pm 5, \pm 7\}$. The message will be passed as label to the CE loss function, which we indicate by the color blue. The mapper output is then upsampled with oversampling factor 2 and filtered by the transmit filter, which is an instance of the linear filter as discussed in Section 8.4.1. Of the transmitter chain, the transmit filter needs to be trainable, while the other devices before the filter don't require to be implemented differentiable, as the loss gradient will be backpropagated only until reaching the transmit filter.

In the figure, we show the PSD of the signals at the transmitter after training. We plot against the frequency slot k, and we chose for $\boldsymbol{x}$ a frequency resolution of $n_{\text{perseg}} = 100$ and for the upsampled signals $\boldsymbol{u}, \boldsymbol{y}_{\text{tx}}$, we chose $n_{\text{perseg}} = 200$, so that we can display the signals before and after upsampling in the same plot. For using other frequency units for the horizontal axis, see Section 8.2.4. Note that for putting the low frequencies in the middle, we would need to shift $\boldsymbol{x}$ to the left by 50, and the upsampled signals by 100.

8.4.3. Channel


$\boldsymbol{z} \sim \mathcal{N}(0, \sigma^2)$
$\boldsymbol{y}_{\text{tx}}$
$2n$
Low pass filter $\boldsymbol{f}$
$\boldsymbol{y}_{\text{ch}}$
$2n$

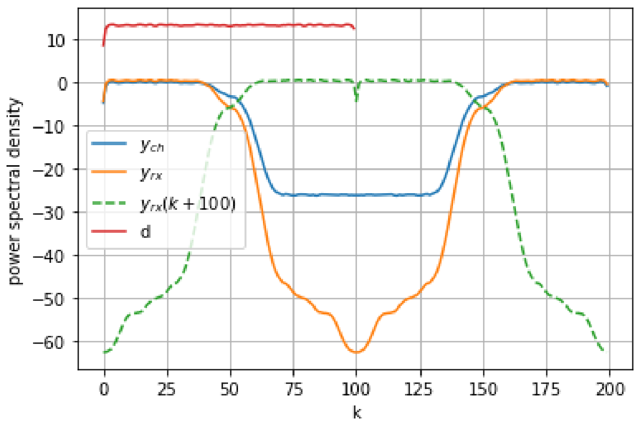


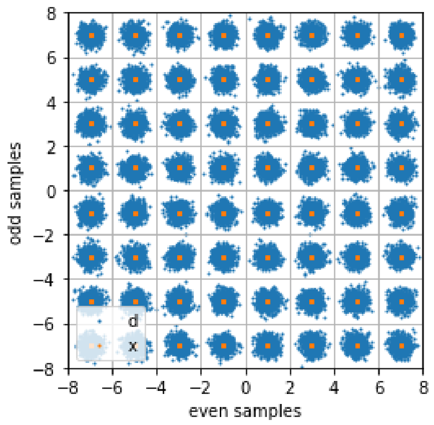
The channel consists of a low pass filter followed by additive Gaussian noise. All devices in the channel chain must be differentiable, as the loss gradient is backpropagated through the channel to the transmitter.

We observe that while the low pass filter $\boldsymbol{f}$ completely attenuates the high frequencies, the chanel output signal $\boldsymbol{y}_{\text{ch}}$ has high frequency components, indicated by the PSD value slightly below 25 dB. This is the noise floor due to the noise added after filtering.

8.4.4. Receiver

$$\begin{tikzcd} {} \arrow[r, "\boldsymbol{y}_{\text{ch}}", "2n"'] & \boxed{\text{Receive Filter } \boldsymbol{h}} \arrow[r, "\boldsymbol{y}_{\text{rx}}", "2n"'] & \boxed{2\downarrow} \arrow[r] & \boxed{\text{Normalize to } \mathbb{E}(|X|^2)} \arrow[r, "\boldsymbol{d}", "n"'] & \boxed{\text{Demapper}} \arrow[r, "\boldsymbol{\ell}", "n \times 8"'] & {} \end{tikzcd}$$





The receive filter is an instance of the linear filter discussed in Section 8.4.1. After downsampling, we normalize to the power of the transmitted signal $\boldsymbol{x}$, to facilitate the comparison between the reconstruction $\boldsymbol{d}$ and the original signal $\boldsymbol{x}$. As demapper, we use a linear NN outputting 8 log probabilities $\boldsymbol{\ell}$, one for each possible message value. The demapper output will be passed as prediction to the CE loss function, which we indicate by the color blue. Both the receive filter and the demapper are trainable. All devices in the receiver chain must be implemented differentiable, as the loss gradient is backpropagated through the channel to the transmitter.

In the figure, we show the PSD of the signals after training. To illustrate the effect of downsampling, we also display by a dashed line the PSD of the filtered signal $\boldsymbol{y}_{\mathrm{rx}}$ shifted by 100 (see also the discussion in Section 8.3.3). We observe that the downsampled signal $\boldsymbol{d}$ has a flat PSD similar to the PSD of the transmitted signal $\boldsymbol{x}$ (See also the discussion in Section 8.2.3).

8.5. Further Reading

[1] treats the DFT in Chapter 4, the PSD in Section 8.4 and interpolation and sampling in Chapters 9 and 11. Welch's method for PSD estimation was originally published in [2]. In [3], mapper and interpolation filter are trained end-to-end in an autoencoder-based communication system.

References

[1] P. Prandoni and M. Vetterli, *Signal Processing for Communications*. 2008.

[2] P. Welch, "The use of fast Fourier transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms," *IEEE Trans. Audio Electroacoustics*, vol. 15, no. 2, pp. 70–73, 1967.

[3] T. Uhlemann, S. Cammerer, A. Span, S. Dörner, and S. ten Brink, "Deep-learning autoencoder for coherent and nonlinear optical communication," in *In Proc. ITG-Symposium Photonic Networks*, VDE, 2020.

8.6. Problems

Problem 8.1. (PSD and white signals)

1. Implement the PSD estimator (8.9).

2. Sample the length $n = 100, 1000, 10\,000$ time signal $\boldsymbol{x}$ with independent normal Gaussian entries. For each signal length n and $n_{\text{perseg}} = 100$, plot its PSD using $n_{\text{seg}} = n/n_{\text{perseg}}$ segments of length $n_{\text{perseg}} = 100$.

3. Consider now the signal $\boldsymbol{y}$ with

$$y_i = x_i + 0.5 \cdot x_{(i-1) \bmod n}. \tag{8.29}$$

   Are the entries of $\boldsymbol{y}$ independent?

4. Plot the PSD of $\boldsymbol{y}$ for $n_{\text{perseg}} = 100$ and $n = 100, 1000, 10\,000$. Can you decide from the PSD if the entries of $\boldsymbol{y}$ are independent?

5. Compare the plots of 2. and 4. to using

```
from scipy import signal
_, Syy = signal.welch(y, nperseg=nperseg, return_onesided=False)
```

Problem 8.2.

1. Carry out the end-to-end filter design described in Section 8.4 and plot the PSDs of the filtered signals.

2. Remove the low pass filter from the channel. Retrain the system and plot the PSDs of the filtered signals.

3. Exchange the low pass filter in the channel by interference as described in Section 8.1.2. Retrain the system and plot the PSDs of the filtered signals.

4. Exchange the low pass filter in the channel by the pilot tone as described in Section 8.1.3. Retrain the system and plot the PSDs of the filtered signals.

Part III.

Channel

9. Deterministic Channel Modeling

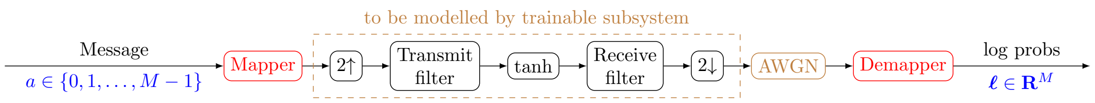

to be modelled by trainable subsystem
Message
$a \in \{0, 1, \dots, M-1\}$
Mapper
2↑
Transmit filter
tanh
Receive filter
2↓
AWGN
Demapper
log probs
$\boldsymbol{\ell} \in \mathbf{R}^M$

Figure 9.1.: Communication system example. To train the transmitter-sided mapper, we substitute the inner subsystem by a differentiable model, which allows us to backpropagate the loss gradient through the model to the transmitter.

As we have seen in Chapter 7, for end-to-end training of transmitter components, we must backpropagate the loss gradient to the transmitter through the inner subsystem, which may consist of the transmitter frontend, the channel, and the receiver frontend. This backpropagation requires the inner subsystem to support automatic differentiation, which is often not the case. For instance, the subsystem of interest may be available only as a non-differential implementation. In some cases, the software providing the subsystem may be available to us only in compiled form, with no access to the source code. In another scenario, the subsystem may not be available as a runable software at all, but only in the form of input and output sequences recorded in a transmission experiment.

In this chapter, we identify the basic challenges of subsystem modeling and how they can be handled. To this end, we focus on setups similar to the system in Figure 9.1. Specifically, we model a subsystem implemented in software, and we assume we can run it as often as we want. In particular, we assume that we can freely choose input sequences, transmit them over the subsystem, and record the output sequence. This allows us to try out different types of input sequences and obtain insights about which sequences are best suited for learning a representative model.

9.1. Noisy Channel Modeling Challenge

$$\begin{tikzcd} \arrow[r, "x"] & \boxed{f} \arrow[r, "y"] & \boxed{g} \arrow[r, "\hat{x}"] & {} \end{tikzcd}$$

$$\begin{tikzcd} \arrow[r, "x"] & \boxed{\hat{f}} \arrow[r, "\hat{y}"] & {} \end{tikzcd}$$

The basic channel modeling task is as follows: for a channel f, find a *surrogate channel* $\hat{f}$, so that $\hat{f}(x) \approx f(x)$, for all input values x of interest. Note how this differs from

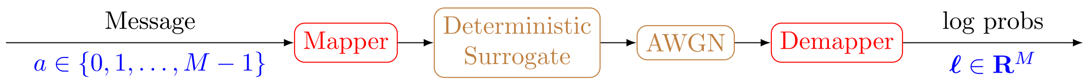

Message
$a \in \{0,1,\ldots,M-1\}$
Mapper
Deterministic Surrogate
AWGN
Demapper
log probs
$\boldsymbol{\ell} \in \mathbf{R}^M$

Figure 9.2.: Differentiable subsystem model consisting of a deterministic model followed by Gaussian noise. Note that the noise source is placed after the deterministic model, in difference to the original subsystem, where the noise source is followed by the receive filter and downconversion. This may result in differences between the original subsystem and its model.

equalization, which we considered in previous chapters. In equalization, for a channel f, we want to find g so that $g(f(x)) \approx x$, i.e., $g \approx f^{-1}$. In other words, while channel modeling tries to imitate a channel, equalization tries to invert it.

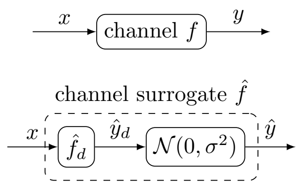
The task of imitating a given channel can be intricate. Consider the subsystem in Figure 9.1. We observe that the subsystem f of interest contains a noise source. In particular, $f(x)$ is random, also for known input x. In particular, $x_1 = x_2$ does *not* imply $f(x_1) = f(x_2)$. Thus, the requirement for surrogate $\hat{f}$ is not exactly $\hat{f}(x) = f(x)$, rather, $\hat{f}(x)$ and $f(x)$ should be “similar enough”.

9.2. Deterministic Channel Surrogate

Assume now that the channel surrogate $\hat{f}$ is deterministic, so that also $\hat{f}(x)$ is deterministic and $x_1 = x_2$ implies $\hat{f}(x_1) = \hat{f}(x_2)$, a property that the original noisy channel does not have. Thus, the deterministic surrogate $\hat{f}$ does not model well the noisy channel f. We therefore consider a channel surrogate that consists of a deterministic surrogate followed by additive noise, i.e., our surrogate is

$$\hat{f}(x) = \hat{f}_d(x) + Z \tag{9.1}$$

where $\hat{f}_d$ is deterministic and Z is zero mean Gaussian noise.



In Figure 9.2, we display the system from Figure 9.1 with the inner subsystem replaced by a surrogate channel.

For training the surrogate channel, we use training sequences

$$x^n = x_0 x_1 \ldots, x_{n-1} \tag{9.2}$$
$$y^n = y_0 y_1 \ldots y_{n-1}, \quad y_i = f(x_i) \tag{9.3}$$

and we learn the channel surrogate in two steps:

1. We train the deterministic surrogate $\hat{f}_d$ w.r.t. MSE:

$$\hat{f}_d = \arg\min_{\hat{f}_d} \frac{1}{n}\sum_{i=0}^{n-1} |y_i - \hat{f}_d(x_i)|^2 \tag{9.4}$$

2. We estimate the variance of the residual error $y_i - \hat{f}_d(x_i)$:

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i=0}^{n-1} |y_i - \hat{f}_d(x_i)|^2. \tag{9.5}$$

   Note that the variance $\hat{\sigma}^2$ of the residual error is exactly the MSE after training f_d.

9.3. Linear Channel Surrogate

Suppose now we consider an affine surrogate. For a memoryless scalar channel, $\hat{f}_d(x) = a \cdot x + b$ (for $b \neq 0$, $\hat{f}_d$ is affine and for $b = 0$, $\hat{f}_d$ is linear) and the MSE is

$$\frac{1}{n}\sum_{i=0}^{n-1} |y_i - a \cdot x_i - b|^2. \tag{9.6}$$

For channels where inputs $\boldsymbol{x}$ and outputs $\boldsymbol{y}$ are column vectors with r and c entries, respectively, the affine surrogate is

$$\hat{f}_d(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}, \quad \boldsymbol{A} \in \mathbf{R}^{c\times r}, \quad \boldsymbol{b} \in \mathbf{R}^{c}. \tag{9.7}$$

For a scalar channel with memory, we may use as surrogate channel a linear filter $\boldsymbol{a}$ with m entries so that

$$f_d(x_i) = (\boldsymbol{a} \star \boldsymbol{x})_i + b. \tag{9.8}$$

Linear channel surrogates linearize the channel in the neighborhood of x^n. Therefore, we should choose the training sequence x^n similar to the signal of interest. For instance, we may use for x^n an appropriately scaled M-ASK signal, i.e.,

$$\begin{gathered} x_i = \lambda\tilde{x}_i, \quad \tilde{x}_i \in \{\pm 1, \pm 3, \dots, \pm(M-1)\} \\ i = 0, 1, \dots, n-1. \end{gathered} \tag{9.9}$$

In other words, to train linear channel surrogates, the important property of the training sequences is the range in which the values lie, rather than how finely this range is sampled by the entries of the training sequence (as we will see, this is different for non-linear channel surrogates). In summary,

- Linear channel surrogates may yield a good approximation in the neighborhood of the training sequence x^n.

- The approximation may be bad outside of the neighborhood of x^n, if the original channel is non-linear.
- If the original channel is linear, then the linear channel surrogate can be very good, even outside of the neighborhood of x^n. Thus, in principle, it can generalize well to unseen data.

9.4. Non-Linear Channel Surrogate

For a non-linear surrogate, the MSE is

$$\frac{1}{n}\sum_{i=0}^{n-1}|y_i - f_d(x_i)|^2. \tag{9.10}$$

Suppose now the training sequence takes values in a finite set $\mathcal{X}$ (e.g., an ASK constellation), as we may use to train linear surrogates. We can now write the MSE as

$$\frac{1}{n}\sum_{i=0}^{n-1}|y_i - f_d(x_i)| = \frac{1}{n}\sum_{x\in\mathcal{X}}\sum_{i\colon x_i=x}|y_i - f_d(x)|^2. \tag{9.11}$$

Minimizing this objective, the surrogate may approximate the original channel f very well for $x \in \mathcal{X}$, but the *approximation may be arbitrarily bad elsewhere!* As countermeasure, we should therefore sample the training sequence continuously from $\mathbf{R}$. Of course, a training sequence of length n has at most n distinct values, which is far from continuous. Fortunately, most of the common non-linear activation functions like ReLU and tanh are continuous function, which comes to our rescue here. Besides the requirement to sample finely, the range of the training samples is also important. We need to ensure that the training sequence covers the range of interest, or rather exceeds it at least slightly, as a non-linear surrogate may generalize arbitrarily poorly to unseen data.

9.5. Further Reading

In [1, Section II.C], training sequences for learning channel surrogates are discussed.

References

[1] A. Smith and J. Downey, "A communication channel density estimating generative adversarial network," in *IEEE Cognitive Commun. Aerosp. Appl. Workshop (CCAAW)*, 2019.

9.6. Problems

In this promblem set, we consider the non-linear channel

$$Y = f(X) = \tanh(X) + Z \tag{9.12}$$

where Z is zero mean Gaussian with variance $\sigma^2 = 7 \times 10^{-3}$. We model the channel by

$$\hat{Y} = g(X) + \hat{Z} \tag{9.13}$$

where g is a deterministic function and $\hat{Z}$ zero mean Gaussian with variance $\hat{\sigma}^2$.
Problem 9.1. Let X be a real random variable and define

$$g(X) = \arg\min_{a \in \mathbf{R}} \mathbb{E}\big[|a - f(X)|^2 \big| X\big]. \tag{9.14}$$

1. Show that $g(X) = \tanh(X)$.
2. Show that the residual error $\hat{Z} = g(X) - f(X)$ is zero mean Gaussian with variance σ^2.

Problem 9.2. (Linear model)

1. Implement g as a linear neuron.
2. Use the 4-ASK alphabet $\mathcal{X} = \{\pm 1, \pm 3\}$. Sample $\tilde{x}^n = \tilde{x}_0 \dots \tilde{x}_{n-1}$ uniformly from $\mathcal{X}$ and generate the input and output sequences

$$x^n = 0.2 \cdot \tilde{x}^n, \quad y_i = f(x_i), \quad i = 0, \dots, n-1. \tag{9.15}$$

   Train your linear model g using x^n as input and y^n as target output using the MSE loss function on $g(x^n)$ and y^n.
3. For your trained model, plot $g(x)$ and $\tanh(x)$ for $x \in [-2, 2]$. In which range of x does g approximate tanh well?

Problem 9.3. (Non-linear model)

1. Implement g by a non-linear NN using several hidden layers with RelU activation.
2. Repeat the steps of Problem 9.2 for your non-linear model. How well does g approximate tanh?
3. Repeat the steps of Problem 9.2 for your non-linear model using, using for x^n zero mean Gaussian samples with variance equal to the variance of x^n in Problem 9.2. How well does g approximate tanh now?

Problem 9.4. (Model-based transmitter design) We revisit Problem 7.2.

1. Train a mapper following Problem 7.2 using your model from Problem 9.3.
2. Evaluate your mapper on the true channel. How does it's performance compare to the performance of a mapper that is directly trained on the true channel?

10. Stochastic Channel Modeling

To model a noisy channel, we introduced in the last chapter a pragmatic approach: we first learn a deterministic model, and we then add noise, which models the residual error of the deterministic model. As we learn the deterministic model using MSE loss, the residual error has zero mean. We estimate the variance of the residual error and we then model the noise as zero mean Gaussian with that variance, i.e.,

$$Z \sim \mathcal{N}(0, \hat{\sigma}^2). \tag{10.1}$$

This is a parametric model, with the parameter σ^2, and to then generate such noise, we need a random number generator that accepts this parameter and then generates Gaussian random variables accordingly.

In this chapter, we consider a non-parametric approach, i.e., we want to learn a noise source from data. More specifically,

- We assume we have some source of randomness, e.g., a random number generator from which we can sample s^n, where the s_i are independent and uniformly distributed.
- We want to learn a fake source via learning a deterministic transformation f, so that $\hat{z}^n = f(s^n)$ statistically has the desired properties.
- We estimate the desired properties from our observation z^n of the true source. These properties can be any values that we can estimate from z^n, for instance, mean, variance, and higher moments.
- We would like to have a loss function to which we can pass samples z^n from the true source and samples $\hat{z}^n$ from the fake source, and which make the fake source look like the true source after convergence.

This is an active field of research and significant progress has been made in the last years, see Section 10.3 for some references.

We will next take a close look at how this can be realized when the true source is in fact a Gaussian density.

10.1. Principle of Maximium Entropy

So far, we have defined the Gaussian density formally via the formula

$$p_Z(z) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{z^2}{2\sigma^2}}. \tag{10.2}$$

To see how we can learn a Gaussian source, we may look for an operational definition. For instance, the central limit theorem [1] provides one operational definition of the Gaussian density. As we are interested in modeling communications channel, we will consider an information-theoretic operational definition based on the principle of maximum entropy (PME) [2]:

> "The **principle of maximum entropy** states that the probability distribution which best represents the current state of knowledge about a system is the one with largest entropy, in the context of precisely stated prior data (such as a proposition that expresses testable information)."

As the Gaussian source we want to learn is a continuous random variable, "probability distribution" in the PME is in our case a probability density function and accordingly, "entropy" is differential entropy, which we are defining next.

10.1.1. Differential Entropy

For a probability density function $p_{\boldsymbol{X}}$ on $\mathbf{R}^d$, differential entropy [3, Chapter 8] is defined as

$$\mathrm{h}(p_{\boldsymbol{X}}) = \mathbb{E}[-\log p_{\boldsymbol{X}}(\boldsymbol{X})] \tag{10.3}$$

$$= \int p_{\boldsymbol{X}}(\boldsymbol{t})[-\log p_{\boldsymbol{X}}(\boldsymbol{t})]\,\mathrm{d}\boldsymbol{t}. \tag{10.4}$$

By Problem 10.2, the differential entropy of a Gaussian is

$$Z \sim \mathcal{N}(\mu, \sigma^2), \quad \mathrm{h}(p_Z) = \frac{1}{2}\log(2\pi e\sigma^2) \tag{10.5}$$

in particular, the differential entropy does not depend on the mean.

10.1.2. Towards a Loss Function for Learning Gaussian Noise

In our case, the "precisely stated prior data" in the PME consists in mean and variance, i.e.,

$$\mathbb{E}(Z) = 0, \quad \mathbb{E}(Z^2) = \sigma^2 \tag{10.6}$$

and according to the PME, the density we are looking for is

$$p = \operatorname*{arg\,max}_{p\colon\ \mathbb{E}(Z)=0,\ \mathbb{E}(Z^2)\le\sigma^2} \mathrm{h}(p). \tag{10.7}$$

We claim that a zero mean Gaussian density with variance σ^2 solves this problem. To show this, we use the information inequality in the form

$$\mathbb{D}(p_Y \| p_Z) = \mathbb{E}\left[\log \frac{p_Y(Y)}{p_Z(Y)}\right] \geq 0 \tag{10.8}$$

with equality if and only if $pY = p_Z$. The information measure $\mathbb{D}(p_Y \| p_Z)$ is called *informational divergence*, also known as *Kullback-Leibler divergence*.

Let now p_Z be zero mean Gaussian and let p_Y be any other zero mean density and assume that both p_Z and p_Y have variance σ^2. We have

$$\begin{aligned}
\mathbb{D}(p_Y \| p_Z) &= \mathbb{E}\left[\log \frac{p_Y(Y)}{p_Z(Y)}\right] && (10.9)\\
&= \mathbb{E}\left[-\log p_Z(Y)\right] - \mathrm{h}(p_Y) && (10.10)\\
&= \mathbb{E}\left[\frac{Y^2}{2\sigma^2}\log(e)\right] + \frac{1}{2}\log(2\pi\sigma^2) - \mathrm{h}(p_Y) && (10.11)\\
&= \frac{1}{2}\log(2\pi e\sigma^2) - \mathrm{h}(p_Y) && (10.12)\\
&\geq 0 && (10.13)
\end{aligned}$$

By the information inequality (10.8), we have equality in (10.13) if and only if $p_Y = p_Z$, which proves that among all densities with variance σ^2, the Gaussian density achieves the maximum differential entropy, and this maximum is equal to $\frac{1}{2}\log(2\pi e\sigma^2)$.

Line (10.11) provides us an alternative formulation of the optimization problem. Let's drop now the restriction that the variance of p_Y should be equal to σ^2. Still, by the information inequality (10.8)

$$\mathbb{E}\left[\frac{Y^2}{2\sigma^2}\log(e)\right] + \frac{1}{2}\log(2\pi\sigma^2) - \mathrm{h}(p_Y) \geq 0 \tag{10.14}$$

with equality iff $p_Y = p_Z$. As we know by now that p_Z solves (10.7), we can now minimize the left-hand side of (10.14) to solve (10.7). We have

$$\begin{aligned}
p_Z = &\operatorname*{arg\,min}_{p_Y\colon\, \mathbb{E}(Y)=0} \frac{\log(e)}{2\sigma^2}\mathbb{E}(Y^2) + \frac{1}{2}\log(2\pi\sigma^2) - \mathrm{h}(p_Y) && (10.15)\\
= &\operatorname*{arg\,min}_{p_Y\colon\, \mathbb{E}(Y)=0} \frac{\log(e)}{2\sigma^2}\mathbb{E}(Y^2) - \mathrm{h}(p_Y). && (10.16)
\end{aligned}$$

The last line is almost the loss function we are looking for. We can replace $\mathbb{E}(Y^2)$ by an empirical expectation, however, we need to find a way for how to estimate the differential entropy $\mathrm{h}(p_Y)$ from a sample sequence y^n. We will do exactly that in the next section.

10.2. Differential Entropy Estimator

In this section, we summarize the nearest neighbor differential entropy estimator as analyzed in [4]. To stay close to [4], we describe the estimator a bit more general then

Dimension d	Volume of a ball of radius R	Volume of the unit ball
0	$V_0(R) = 1$	$V_0 = 1$
1	$V_1(R) = 2R$	$V_1 = 2$
2	$V_2(R) = \pi R^2$	$V_2 = \pi$

Table 10.1.: Volumes of low dimensional Euclidean balls.

what we will actually need. That is, we consider the general k-nearest neighbor, while in our implementation, we use the nearest neighbor ($k = 1$).

10.2.1. Volume of a Euclidean Ball

The d-dimensional volume of a Euclidean ball of radius R in d-dimensional Euclidean space is [5]:

$$V_d(R) = \frac{\pi^{\frac{d}{2}}}{\Gamma(\frac{d}{2}+1)} R^d. \tag{10.17}$$

In Table 10.1, we display the first few low dimensional volumes. We denote the volume of a unit ball with radius 1 by

$$V_d = V_d(1). \tag{10.18}$$

In [4], [6] the Euclidean ball of radius R around $\boldsymbol{x}$ and its volume are denoted by

$$B(\boldsymbol{x}, R) = \{\boldsymbol{z} \colon \|\boldsymbol{z} - \boldsymbol{x}\|_2 \leq R\} \tag{10.19}$$

$$\lambda(B(\boldsymbol{x}, R)) = V_d(R). \tag{10.20}$$

10.2.2. k-Nearest Neighbor

Let $\boldsymbol{y}^{(k)}(\boldsymbol{x})$ be the k-th nearest neighbor of $\boldsymbol{x}$ among $\boldsymbol{y}_1, \dots, \boldsymbol{y}_n$, i.e.,

$$\|\boldsymbol{y}^{(1)}(\boldsymbol{x}) - \boldsymbol{x}\|_2 \leq \|\boldsymbol{y}^{(2)}(\boldsymbol{x}) - \boldsymbol{x}\|_2 \leq \dots \leq \|\boldsymbol{y}^{(n)}(\boldsymbol{x}) - \boldsymbol{x}\|_2. \tag{10.21}$$

10.2.3. d-dimensional Data

We consider n samples $\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_n$, where each sample is a d-dimensional vector with d entries, i.e.,

$$\boldsymbol{x}_i \in \mathbf{R}^d = \underbrace{\mathbf{R} \times \mathbf{R} \times \dots \mathbf{R}}_{d \text{ times}}, \qquad i = 1, \dots, n. \tag{10.22}$$

10.2.4. Density Estimator

By [6, Definition 3.1], the k-nearest neighbor density estimate from n d-dimensional samples $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_n$ is defined by

$$f_n^{(k)}(\boldsymbol{x}) = \frac{k}{n \cdot V_d \cdot \left\|\boldsymbol{y}^{(k)}(\boldsymbol{x}) - \boldsymbol{x}\right\|_2^d} \tag{10.23}$$

Example 10.1. For $k = 1$ and $d = 1$, the nearest neighbor density estimate is

$$f_n(x) = \frac{1}{n \cdot 2 \cdot |y^{(1)}(x) - x|}. \tag{10.24}$$

10.2.5. Differential Entropy Estimator

By [4, Section 1], the principle of differential entropy estimation is as follows. Let $p_{\boldsymbol{X}}$ be the actual density and $\hat{p}$ an approximation of $p_{\boldsymbol{X}}$. Then,

$$h(p_{\boldsymbol{X}}) = \mathbb{E}[-\log p_{\boldsymbol{X}}(\boldsymbol{X})] \approx \frac{1}{n}\sum_{i=0}^{n-1} -\log p_{\boldsymbol{X}}(\boldsymbol{x}_i) \tag{10.25}$$

$$\approx \frac{1}{n}\sum_{i=0}^{n-1} -\log \hat{p}(\boldsymbol{x}_i). \tag{10.26}$$

We now use the k-nearest neighbor estimate for $\hat{p}$. The following observation requires attention:

We use $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$ both for *estimating* $\hat{p}$ as well as for *evaluating* $\hat{p}$.

To account for this, for each $i = 1, \ldots, n$, we estimate $\hat{p}(\boldsymbol{x}_i)$ using the other samples $\boldsymbol{x}_j$, $j \neq i$, i.e., samples from the punctured set

$$\mathcal{X}_i = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{i-1}, \boldsymbol{x}_{i+1}, \ldots, \boldsymbol{x}_n\}. \tag{10.27}$$

Let $R_{i,k}$ be the distance of $\boldsymbol{x}_i$ to its k-nearest neighbor in $\mathcal{X}_i$. The density estimate is then

$$\hat{p}(\boldsymbol{x}_i) = \frac{k}{n \cdot V_d \cdot R_{i,k}^d} \tag{10.28}$$

and the entropy estimate becomes

$$h(p_{\boldsymbol{X}}) \approx \frac{1}{n}\sum_{i=0}^{n-1}\left(-\log \frac{k}{n \cdot V_d \cdot R_{i,k}^d}\right) \tag{10.29}$$

$$= \frac{1}{n}\sum_{i=0}^{n-1} \log\left(\frac{n}{k} \cdot V_d \cdot R_{i,k}^d\right) \tag{10.30}$$

$$\tag{10.31}$$

```python
import torch
from mlcomm import estim

n = 10_000
d = 4
x = torch.randn(n, d)
h_hat = estim.differential_entropy_estimator_dD(x)
```

https://gitlab.lrz.de/gb/mlcomm

Figure 10.1.: `mlcomm.estim.differential_entropy_estimator_dD` implements the Kozachenko-Leonenko estimator for $k = 1$, i.e., using the nearest neighbor. The differential entropy estimate is differentiable and can be used in loss functions, for instance, in the loss function (10.16) that we derived for learning Gaussian noise.

By [4] the Kozachenko-Leonenko entropy estimator is given by

$$\hat{h}_{n,k}(\boldsymbol{x}^n) = \log k - \psi(k) + \frac{1}{n}\sum_{i=0}^{n-1} \log\left(\frac{n}{k} \cdot V_d \cdot R_{i,k}^d\right) \tag{10.32}$$

where $\log k - \psi(k)$ is a bias-correcting term and ψ is the digamma function [7] with

$$\psi(1) = -\gamma \tag{10.33}$$

$$\gamma = \int_0^\infty e^{-t} \ln t \,\mathrm{d}t. \tag{10.34}$$

The value γ is called the Euler-Mascheroni constant [8].

Example 10.2. For $k = 1$ and $d = 1$, the Kozachenko-Leonenko entropy estimator becomes

$$\hat{h}_{n,1}(x^n) = \log 1 - \psi(1) + \frac{1}{n}\sum_{i=1}^{n} \log\left(\frac{n}{1} \cdot V_1 \cdot R_{i,1}^1\right) \tag{10.35}$$

$$= \gamma + \frac{1}{n}\sum_{i=1}^{n} \log\left(n \cdot 2 \cdot R_i\right) \tag{10.36}$$

which recovers [9, Eq. (18)]. Note that $R_i := R_{i,k}^1$ is given by

$$R_i = \arg\min_{x \in \mathcal{X}_i} |x - x_i|. \tag{10.37}$$

In Figure 10.1, we refer an implemtation of the estimator for $k = 1$ and any dimension d.

10.3. Further Reading

The learning of fake noise is an example of deep generative modeling. A landmark paper is [10], which introduces the generative adversarial network (GAN). In [11] and many other works, GANs have been used for modeling communication channels. A GAN consists of a generator and a discriminator. In Problem 10.4, we implement such a discriminator for assessing the quality of our trained fake source.

References

[1] *Central limit theorem.* [Online]. Available: `https://en.wikipedia.org/wiki/Central_limit_theorem`.

[2] *Principle of maximum entropy.* [Online]. Available: `https://en.wikipedia.org/wiki/Principle_of_maximum_entropy`.

[3] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, 2nd ed. John Wiley & Sons, Inc., 2006.

[4] J. Jiao, W. Gao, and Y. Han, "The nearest neighbor information estimator is adaptively near minimax rate-optimal," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, 2018, pp. 3160–3171.

[5] [Online]. Available: `https://en.wikipedia.org/wiki/Volume_of_an_n-ball`.

[6] G. Biau and L. Devroye, *Lectures on the nearest neighbor method.* Springer, 2015, vol. 246.

[7] [Online]. Available: `https://en.wikipedia.org/wiki/Digamma_function`.

[8] [Online]. Available: `https://en.wikipedia.org/wiki/Euler-Mascheroni_constant`.

[9] J Beirlant, E. D. L. Györfi, and E. van der Meulen, "Nonparametric entropy estimation: An overview," 2001.

[10] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," *Advances in neural information processing systems*, vol. 27, 2014.

[11] T. J. O'Shea, T. Roy, N. West, and B. C. Hilburn, "Physical layer communications system design over-the-air using adversarial networks," in *2018 26th European Signal Processing Conference (EUSIPCO)*, 2018, pp. 529–532.

10.4. Problems

Problem 10.1. (Linear filter recap) Consider the filter

$$\boldsymbol{h} = (0, 0, 3, 2, 1). \tag{10.38}$$

1. Implement a function that realizes the circular convolution $\boldsymbol{h} \star \boldsymbol{x}$ for a length n input vector $\boldsymbol{x}$.

2. Realize the circular convolution from 1. using an NN with a linear layer provided by `torch.nn.Linear`. Verify the correctness of your implementation by comparing your implementation from 1. for several test sequences $\boldsymbol{x}$.

3. Realize the convolution from 1. using an NN with a convolutional layer provided by `torch.nn.Conv1d`. Is this convolution circular? Test against 1. and 2. Where in the output sequence do the sequences differ?

Problem 10.2. (differential entropy)

1. Show that the differential entropy of a Gaussian random variable $Z \sim \mathcal{N}(\mu, \sigma^2)$ is given by $\mathrm{h}(p_Z) = \frac{1}{2}\log(2\pi e \sigma^2)$.

Problem 10.3. (differential entropy estimation) Sample $n = 10\,000$ vectors $\boldsymbol{z}_0, \boldsymbol{z}_1, \dots, \boldsymbol{z}_{n-1}$, each with d entries from a zero mean Gaussian density $p_{\boldsymbol{X}}$ with covariance matrix $\sigma^2 \cdot \boldsymbol{I}$, where $\boldsymbol{I}$ is the $d \times d$ identity matrix.

1. What is the differential entropy of the d-dimensional random vector $\boldsymbol{X}$?

2. Plot the differential entropy for $d = 1, 2, 3, 4$ and $\sigma^2 = 0.1, 0.2, \dots, 1$.

3. Use the differential entropy estimator

   `mlcomm.estim.differential_entropy_estimator_dD`

   from https://gitlab.lrz.de/gb/mlcomm to estimate the differential entropy from the sample sequences $\boldsymbol{z}^n$.

Problem 10.4. (fake Gaussian source) In this problem, you will implement an NN that transforms a uniform random variable into fake Gaussian noise.

1. Implement a nonlinear NN f with scalar input and scalar output. As input to this network, sample u^n from a density p_U that is uniform on $[0, 1]$. You can use `torch.rand(n, 1)`. Plot a histogram of the output $\hat{z}^n = f(u^n)$ of the initialized NN and compare it with the histogram of samples z^n from a zero mean Gaussian density with variance $\sigma^2 = 1$.

2. Train your network using as loss the MSE. Plot the histogram. Can you explain the trained f mathematically?

3. To enforce the required mean and variance. Train your network using as loss

$$|\,\mathbb{E}(\hat{Z})|^2 + |\,\mathbb{E}(\hat{Z}^2) - \sigma^2|^2 \tag{10.39}$$

In your implementation, replace probabilistic expectation $\mathbb{E}(\cdot)$ by empirical expectation. (You can also try using $|\cdot|$ instead of $|\cdot|^2$ in the above loss). Plot the resulting histogram.

4. Train now your network using the loss (10.16), i.e.,

$$|\,\mathbb{E}(\hat{Z})|^2 + \frac{\log(e)}{2\sigma^2}\,\mathbb{E}(\hat{Z}^2) - \mathrm{h}(\hat{Z}) \tag{10.40}$$

where we have added $|\,\mathbb{E}(\hat{Z})|^2$ to enforce zero mean. In your implementation replace probabilistic expectations by empirical expectations and use the differential entropy estimator from Problem 9.3.3 to estimate $\mathrm{h}(\hat{Z})$.

5. We now want to assess how good our fake source is. To this end, we train a discriminator using the BCE loss on LPRs. The discriminator is a non-linear network with scalar input and scalar output. As we use the BCE on LPRs, the output layer is linear.

   a) Training: We feed the discriminator with sequences z^n_{true} from the true source with label $b = 1$, and sequences z^n_{fake} from our fake source with label $b = 0$. Denoting the discriminator output by ℓ, the cost can be calculated by

$$\mathrm{BCE}_{\log}(b^{2n}, \ell^{2n}) = \frac{1}{2n}\sum_{i=0}^{2n-1} \mathrm{bce}_{\log}(b_i, \ell_i) \tag{10.41}$$

$$= 0.5 \cdot \frac{1}{n}\sum_{i=0}^{n-1} \mathrm{bce}_{\log}[0, \ell(z_{\text{fake},i})] + 0.5 \cdot \frac{1}{n}\sum_{i=0}^{n-1} \mathrm{bce}_{\log}[1, \ell(z_{\text{true},i})]. \tag{10.42}$$

   b) Evaluation: calculate the probability $P_{B|Z}(0|z)$ from the tained discriminator output and plot z against $P_{B|Z}(0|z)$. For which values of z can the discriminator not distinguish between true and fake?

   c) Evaluation: estimate $\mathbb{H}(B|Z)$ using the trained discriminator. What would be the value of $\mathbb{H}(B|Z)$ for a perfect fake source? How much in bits is our fake source different from a perfect fake source?

Part IV.

Supplements

11. Filters with Memory

So far, we have considered filters with a finite number of taps. Such filters are also called finite impulse response (FIR) filters. For instance, to equalize or demap the ith sample, the filter processes not only its observation of the noisy ith sample, bu also n_{pre} predecessors and n_{post} successors, where $n_{\text{taps}} = 1 + n_{\text{pre}} + n_{\text{post}}$. In certain situations, the dependence of the current sample on past and future observations can be realized more efficiently, in particular, if this dependency extends far into the past and/or far into the future, which would require a filter with a very large number of taps.

11.1. Recursive Filter

Suppose sequence $\boldsymbol{y} = \dots y_{-1}y_0y_1\dots$ is transmitted over a channel and $\boldsymbol{x} = \dots x_{-1}x_0x_1\dots$ is received, where

$$x_i = y_i + \alpha y_{i-1}. \tag{11.1}$$

Our aim is to recover $\boldsymbol{y}$ from $\boldsymbol{x}$.

First Attempt We first use an FIR filter. We solve (11.1) for y_i. We have

$$\begin{aligned} y_i &= x_i - \alpha y_{i-1} \\ &= x_i - \alpha(x_{i-1} - \alpha y_{i-2}) \\ &= x_i - \alpha(x_{i-1} - \alpha(x_{i-2} - \alpha y_{i-3})) \\ &= x_i - \alpha(x_{i-1} - \alpha(x_{i-2} - \alpha(x_{i-3} - \alpha y_{i-4}))) \\ &= x_i - \alpha x_{i-2} + \alpha^2 x_{i-2} - \alpha^3 x_{i-3} + \cdots \\ &= \sum_{j=0}^{\infty} x_{i-j} \underbrace{(-\alpha)^j}_{=:h_j} = (\boldsymbol{x} \star \boldsymbol{h})(i) \end{aligned}$$

We observe that $\boldsymbol{h}$ is actually *not* an FIR filter, as it is the infinitely long filter

$$\boldsymbol{h} = (1, -\alpha, \alpha^2, -\alpha^3, \dots). \tag{11.2}$$

Such filters are called infinite impulse response (IIR) fitelrs. We observe that if $\alpha < 1$, then the filter coefficients vanish. We therefore may truncate $\boldsymbol{h}$ to finite length, i.e., an FIR filter. Note, however, that truncating $\boldsymbol{h}$ leads to an approximation error, i.e., with an FIR filter, we can only approximately recover $\boldsymbol{y}$ from $\boldsymbol{x}$.

Second Attempt We again solve (11.1) for y_i and get

$$y_i = x_i - \alpha y_{i-1}.$$

We note that at time instant i, x_i is the filter input and y_i the desired filter output. Correspondingly, y_{i-1} is the desired filter output at time instant i. Define now the filter function f by

$$f(x_i) = x_i - \alpha f(x_{i-1}). \tag{11.3}$$

This defines a recursive filter. Assuming that $f(x_0) = y_0$, this recursive filter allows us to perfectly recover $x_0 x_1 \dots$ from $y_0 y_1 \dots$.

11.2. Recursive Filter as Filter with State

We now specify a recursive filter by a filter with state as follows.

- Input x, output y, state h.
- Update state:

$$h = f_h(x, h) \tag{11.4}$$

- Update output:

$$y = f_y(h) \tag{11.5}$$

The updates of state and output are executed in this order, i.e., we first update the state and then the output.

Example 11.1. Consider the state and output updates

$$f_h(x, h) = x - \alpha h \tag{11.6}$$
$$f_y(h) = h. \tag{11.7}$$

By these update rule, we have

$$\begin{aligned} y_i &= h_i & (11.8)\\ &= x_i - \alpha h_{i-1} & (11.9)\\ &= x_i - \alpha y_{i-1}. & (11.10) \end{aligned}$$

That is, the state and output updates implement (11.3).

11.2.1. Initialization

With time index t, the update rules are

$$h_t = f_h(x_t, h_{t-1}) \tag{11.11}$$
$$y = f_h(h_t). \tag{11.12}$$

If we start at time t_0, we must choose some initial value for state h_{t_0-1}.

- At the beginning of time, we may choose $h_{t_0-1} = 0$ (or any other reasonable value).
- If we process a sequence in chunks, we may use the last state from the previous chunk as initial state for the current chunk.

11.3. `pytorch`'s RNN: Elman Network

`pytorch` provides recursive neural networks (RNNs) [1], which implement the update rules

$$\begin{aligned} \boldsymbol{h}_t &= g(\boldsymbol{W}_{ih}\boldsymbol{x}_t + \boldsymbol{b}_{ih} + \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{hh}) \\ \boldsymbol{y}_t &= f(\boldsymbol{h}_t) \end{aligned} \tag{11.13}$$

where

- i refers to input.
- t is the time index.
- h refers to hidden.
- g is the activation function, supported are tanh and relu.
- weights $\boldsymbol{W}_{ih}, \boldsymbol{W}_{hh}$.
- bias $\boldsymbol{b}_{ih}, \boldsymbol{b}_{hh}$.

RNNs of the kind (11.13) are also called *Elman networks* [2].

We can stack several RNN layers: the output of layer j becomes the input of layer $j + 1$.

An *unidirectional* RNN has one state sequence, where state h_t is updated based on state h_{t-1}. A *bidirectional* RNN has two state sequences where the first sequence is updated in forward direction (h_t^1 is updated based on h_{t-1}^1) and the second sequence is updated in backward direction (h_{t-1}^2 is updated based on h_t^2).

11.4. Further Reading

The effectiveness of RNNs for certain equalization problems has already been observed almost 30 years ago [3]. More recently, RNNs have been applied for equalization and demapping in optical communications, e.g., in [4].

References

[1] [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.RNN.html.

[2] [Online]. Available: https://en.wikipedia.org/wiki/Recurrent_neural_network\#Elman_networks_and_Jordan_networks.

[3] G. Kechriotis, E. Zervas, and E. S. Manolakos, "Using recurrent neural networks for adaptive communication channel equalization," *IEEE Trans. Neural Netw.*, vol. 5, no. 2, pp. 267–278, 1994.

[4] M. Schaedler, G. Böcherer, F. Pittalà, S. Calabrò, N. Stojanovic, C. Bluemm, M. Kuschnerov, and S. Pachnicke, "Recurrent neural network soft-demapping for nonlinear ISI in 800Gbit/s DWDM coherent optical transmissions," *J. Lightw. Technol.*, vol. 39, no. 16, pp. 5278–5286, 2021.

11.5. Problems

Problem 11.1. (Recursive Filter) For $n = 100\,000$, consider a BPSK sample sequence x^n where the entries x_i are sampled independent and uniformly distributed on $\{-1, 1\}$. The sequence is filtered and the filter output y^n is observed, where

$$y_i = x_i + \alpha x_{i-1}, \quad i = 0, 1, \dots n-1 \tag{11.14}$$

with $\alpha = 0.5$. For initialization, we assume $x_{-1} = x_{n-1}$. Our goal is to recover x^n from y^n.

1. Plot the PSD of x^n and y^n.
2. Show that the recursive filter $f(y_i) = y_i - \alpha f(y_{i-1})$ recovers x^n.
3. Implement the recursive filter. How does the residual error $f(y_i) - x_i$ look like when you initialize with $f(y_{i-1}) = 0$ and $f(y_{i-1}) = x_{i-1}$.
4. Implement the recursive filter in an NN by using one linear layer with 2 inputs and one output. Train the NN and evaluate the residual error for initializing with $f(y_{i-1}) = 0$ and $f(y_{i-1}) = x_{i-1}$.
5. Implement the recursive filter by using `torch.nn.RNN` with

   ```
   input_size = 1
   hidden_size = 1
   num_layers = 1
   nonlinearity = 'tanh'
   ```

   Train the NN and evaluate the residual error for initializing with $f(y_{i-1}) = 0$ and $f(y_{i-1}) = x_{i-1}$.

Problem 11.2. (Colored Noise) Consider a sampled BPSK sequence x^n. A channel adds colored noise, i.e.,

$$y_i = x_i + z_i, \quad i = 0, \dots, n-1 \tag{11.15}$$

$$z_i = \tilde{z}_i + \alpha \cdot \tilde{z}_{i-1} \tag{11.16}$$

$$\tilde{z}_{-1} = \tilde{z}_{n-1} \tag{11.17}$$

where the $\tilde{z}_i$ are independent zero mean Gaussians with variance σ^2 and $\alpha = 0.9$. Our goal is to soft demap the bit $b = \phi(x)$, where $\phi(-1) = 0$ and $\phi(1) = 1$.

1. What is the variance of Z_i? What is the optimal memoryless demapper? (the memoryless demapper assumes that the Z_i are independent zero mean Gaussian). Plot the BER against $1/\sigma^2$ in dB over a range of values, e.g., $\alpha = 0.05, 0.1, 0.3, 0.5, 1.0$.
2. Implement a linear NN demapper with $m = 21$ filter taps. Train the demapper and plot the BER curve.

3. Implement a recursive NN demapper using `torch.nn.RNN` with `input_size=1`. Try out several values for `hidden_size` and `num_layers` and try out both `nonlinearity = 'tanh'` and `nonlinearity = 'relu'`.
   Try out `bidirectional=False` and `bidirectional=True`. In all cases, terminate the NN with a linear layer. Train the demapper, plot the BER curve, and compare it to the BER curve of the linear demapper implemented in 2.

Part V.

Appendix

A. Training Neural Networks

In this chapter, we consider the task of traning an NN. Let's recall the structure of an NN: It is a network of neurons. Each neuron calculates a sum of its inputs $\boldsymbol{x} = x_0 x_1 \dots x_{k-1}$ weighted by the weights $\boldsymbol{w} = w_0 w_1 \dots, w_{k-1}$, adds a bias b, and passes this intermediate value through an activation function g. The resulting neuron output is then the input of succeeding neurons, and so on. Training an NN based on data $\boldsymbol{x}$ and labels $\boldsymbol{a}$ consists in iteratively updating the parameters w_i and b, with the aim to minimize the cost $\Lambda(\boldsymbol{a}, \boldsymbol{y})$, where $\boldsymbol{y}$ is the NN output. At first sight, this sounds like a complicated task: for instance, standard image recognition networks have millions of parameters.

The key to translate NN training into a manageable task is to interprete the structure of NNs as a computational graph and to split the computation of parameter updates into a series of simple local computations.

In this chapter, we will first interprete the NN output calculation as forward propagation through a computational graph. We will then formulate the gradient descent algorithm for updating parameters, and we will show how this can be realized by back propagating the loss gradient through the NN via local computations. Finally, we will discuss SGD, which enables computationally efficient training.

Thoughout this chapter, we will "think local" and formulate all math in terms of local computations. Thanks to this approach, the resulting formulas will be surprisingly simple.

A.1. Forward Propagation

Consider the calculation of the output in a deployed NN. We interprete the NN as a computational graph on which local computations are performed. Specifically, we repetitively apply the following local computation until all neurons in the network are processed:

1. While there are unprocessed neurons:
   a) Select an unprocessed neuron.
   b) Denote neuron input, weight, bias by $\boldsymbol{x}$, $\boldsymbol{w}$, b, respectively.
   c) If $\boldsymbol{x}$ is ready:
      i. Calculate $z_1 \leftarrow \boldsymbol{x}^T \boldsymbol{w}$.

ii. Calculate $z_2 \leftarrow z_1 + b$.

iii. Calculate output $y = g(z_2)$.

iv. Mark neuron as processed and output y as ready.

Because of the layered structure of an NN, we can also define a specific schedule. For d layers,

- For $i = 0, 1, \ldots, d-1$:
  - Process all neurons in layer i.

That is, the fresh values propagate in forward direction through the network, and this is why the output calculation is also called *forward propagation*.

A.2. Gradient Descent

A.2.1. Derivative

Consider a real-valued function

$$\begin{aligned} f\colon \mathbf{R} &\to \mathbf{R} \\ x &\mapsto y = f(x). \end{aligned} \tag{A.1}$$

We represent the function by the graph

$$\begin{tikzcd} {} \arrow[r, "x"] & \boxed{f} \arrow[r, "y"] & {} \end{tikzcd}$$

The derivative of f in a is

$$\frac{\mathrm{d}y}{\mathrm{d}x}(a) := \lim_{\epsilon \to 0} \frac{f(a+\epsilon) - f(x)}{(a+\epsilon) - a}. \tag{A.2}$$

The derivative is itself a real function defined on the real line, i.e.,

$$\frac{\mathrm{d}y}{\mathrm{d}x} \colon \mathbf{R} \to \mathbf{R}. \tag{A.3}$$

A.2.2. Minimizing a Real Function by Gradient Descent

Suppose now we want to minimize a real function f. For some initial guess a, we consider the derivative in a.

1. If $\frac{\mathrm{d}y}{\mathrm{d}x}(a) > 0$, then the function value *increases* when we move slightly to the right of a, and it decreases if we move slightly to the left of a.

2. If $\frac{\mathrm{d}y}{\mathrm{d}x}(a) < 0$, then the function value *decreases* when we move slightly to the right of a, and it increases if we move slightly to the left of a.

This suggests to update a by repeating until convergence

$$a \leftarrow a - \gamma \frac{\mathrm{d}y}{\mathrm{d}x}(a), \quad \gamma > 0. \tag{A.4}$$

1. Algorithm (A.4) is called *gradient descent*.
2. The step size $\gamma > 0$ in (A.4) is called *learning rate*.
3. The algorithm has converged to a local minimum a^* if a^* remains unchanged under the update (A.4), i.e., a^* is a *fixed point* of the function

   $$h(a) = a - \gamma \frac{\mathrm{d}y}{\mathrm{d}x}(a). \tag{A.5}$$

   This can be seen as follows.

   $$\begin{aligned} a &= h(a) && \text{(A.6)} \\ \Leftrightarrow a &= a - \gamma \frac{\mathrm{d}y}{\mathrm{d}x}(a) && \text{(A.7)} \\ \Leftrightarrow \frac{\mathrm{d}y}{\mathrm{d}x}(a) &= 0. && \text{(A.8)} \end{aligned}$$

The choice of the (potentially adaptive) learning rate γ is sensitive, we provide references in Section A.6.

A.3. Backpropagation

We have seen in Section A.1 that the NN output can be calculated by *forward* propagation. In this section, we will show that the gradient of the loss function w.r.t. a parameter of interest can be calculated by *backpropagation*. The two important structures in the NN computation graph that we need to address are sums of real functions and chains of real functions.

A.3.1. Sum Rule

Consider the function composition

$$\begin{tikzcd}
 & \boxed{f} \arrow[dr, "z_0"] & & \\
x \arrow[ur] \arrow[dr] & & \boxed{+} \arrow[r, "y"] & {} \\
 & \boxed{g} \arrow[ur, "z_1"'] & &
\end{tikzcd}$$

that is, y is given as the sum of two functions of x via

$$y = f(x) + g(x). \tag{A.9}$$

We calculate the derivative $\mathrm{d}y/\,\mathrm{d}x$ in a as the sum of two local derivatives by

$$\frac{\mathrm{d}y}{\mathrm{d}x}(a) = \frac{\mathrm{d}z_0}{\mathrm{d}x}(a) + \frac{\mathrm{d}z_1}{\mathrm{d}x}(a) \tag{A.10}$$

which is the *sum rule* for derivatives.

Back Propagating Through a Sum Suppose in the NN graph, a variable x connects to variable y via a sum of d functions f_i, which defines d variables z_i via

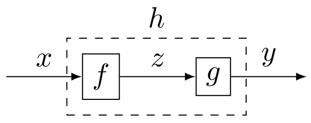
$$z_i = f_i(x), \quad i = 0, \dots, d-1. \tag{A.11}$$

We can now calculate the derivative $\mathrm{d}y/\,\mathrm{d}x$ in a by the following algorithm.

1. For $i = 0, \dots, d-1$:
   - Calculate $b_i = \frac{\mathrm{d}z_i}{\mathrm{d}x_i}(a)$.
2. Calculate $\sum_{i=0}^{d-1} b_i$. This is the desired derivative $\frac{\mathrm{d}y}{\mathrm{d}x}(a)$.

A.3.2. Chain Rule

Consider the function composition



Suppose that we want to calculate $\mathrm{d}y/\,\mathrm{d}x$ in a. We have

$$\begin{aligned}
\frac{\mathrm{d}y}{\mathrm{d}x}(a) &= \frac{h(a+\epsilon) - h(a)}{\epsilon} && \text{(A.12)}\\
&= \frac{g[f(a+\epsilon)] - g[f(a)]}{\epsilon} && \text{(A.13)}\\
&= \frac{g[f(a+\epsilon)] - g[f(a)]}{\epsilon} \cdot \frac{f(a+\epsilon) - f(a)}{f(a+\epsilon) - f(a)} && \text{(A.14)}\\
&= \frac{g[f(a+\epsilon)] - g[f(a)]}{f(a+\epsilon) - f(a)} \cdot \frac{f(a+\epsilon) - f(a)}{\epsilon} && \text{(A.15)}\\
&= \frac{\mathrm{d}y}{\mathrm{d}z}[f(a)] \cdot \frac{\mathrm{d}z}{\mathrm{d}x}(a). && \text{(A.16)}
\end{aligned}$$

Thus, we have just established the chain rule

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mathrm{d}y}{\mathrm{d}z} \cdot \frac{\mathrm{d}z}{\mathrm{d}x}. \tag{A.17}$$

Back Propagating Through a Chain Suppose now in the NN graph, a forward path connects variable x to variable y via d functions f_i, which defines $d+1$ variables z_i via

$$\begin{aligned}
z_0 &= x && \text{(A.18)}\\
z_i &= f_{i-1}(z_{i-1}), \quad i = 1, \dots, d && \text{(A.19)}\\
y &= z_d. && \text{(A.20)}
\end{aligned}$$

For instance, for $d = 1$, we have $y = f_0(x)$ and for $d = 2$, we have $y = f_1(f_0(x))$. We can now calculate $\frac{\mathrm{d}y}{\mathrm{d}x}(a_0)$ by the following algorithm.

1. (Forward propagation) For $i = 1, \dots, d$:
   - Calculate $a_i = f_{i-1}(a_{i-1})$.
2. Initialize $b = 1$.
3. (Backpropagation) For $i = d, d-1, \dots, 1$:
   - $c \leftarrow \frac{\mathrm{d}z_i}{\mathrm{d}z_{i-1}}(a_i)$.
   - $b \leftarrow c \cdot b$.

A.3.3. Example: Back Propagating to a Neuron Weight

We illustrate by an example that thanks to the sum rule and the chain rule, computing gradients of real functions locally is really all we need. Consider the neuron

$$y = g(\boldsymbol{x}^T \boldsymbol{w} + b) \tag{A.21}$$

with

- input $\boldsymbol{x} = x_0 x_1 \dots x_{k-1}$.
- weights $\boldsymbol{w} = w_0 w_1 \dots w_{k-1}$.
- bias b.
- activation function g.
- output y.

Based on data samples $\boldsymbol{x}^n = \boldsymbol{x}_0 \dots \boldsymbol{x}_{n-1}$, we want to update weight w_j using gradient descent. To this end, we express the neuron output y as a function of weight w_j as follows.

$$\begin{align}
y &= \frac{1}{n} \sum_{i=0}^{n-1} g(\boldsymbol{x}_i^T \boldsymbol{w} + b) \tag{A.22}\\
&= \frac{1}{n} \sum_{i=0}^{n-1} g\Big(\underbrace{x_{ij} w_j + \sum_{r \neq j} x_{ir} w_r + b}_{=:f_i(w_j)}\Big) \tag{A.23}\\
&= \frac{1}{n} \sum_{i=0}^{n-1} g[f_i(w_j)]. \tag{A.24}
\end{align}$$

Thus, y is indeed a sum of chains of real functions of w_j, and the derivative $\mathrm{d}y/\,\mathrm{d}w_j$ can be calculated by backpropagation using the sum rule and the chain rule.

A.4. NN Training

We now consider the use of gradient descent for updating NN parameters based on data. For label a and NN output y, the loss is $\lambda(a, y)$ and for n samples $a^n = a_0a_1 \dots a_{n-1}$, $y^n = y_0y_1 \dots y_{n-1}$, the cost is

$$\Lambda(a^n, y^n) = \frac{1}{n}\sum_{i=0}^{n-1} \lambda(a_i, y_i). \tag{A.25}$$

A.4.1. Gradient Descent

To minimize the cost w.r.t. to all parameters $\boldsymbol{w} = w_0w_1 \dots w_{n_{\text{parameters}}-1}$, we may apply gradient descent to the cost as follows.

1. Define $z = \Lambda(a^n, y^n)$.
2. Initialize $\boldsymbol{v}$.
3. for $i = 0, \dots, n_{\text{epochs}} - 1$:
    - for $j = 0, \dots, n_{\text{parameters}} - 1$:
        - Calculate $v_j \leftarrow v_j - \gamma \frac{\mathrm{d}z}{\mathrm{d}w_j}(v_j)$

An *epoch* refers to passing the entire data set a^n, y^n one time forward and backward through the network.

A.4.2. Stochastic Gradient Descent

Calculating the gradient based on the cost $\Lambda(a^n, y^n)$ requires calculating n times the derivative of the loss $\lambda(a_i, y_i)$, before updating the parameters for the first time. This may imply unnecessary calculations, e.g., considering only $n_{\text{batch}} \ll n$ samples (a *batch*) at a time may already approximate the derivative well enough for making a parameter update, i.e., for the first batch,

$$\gamma \frac{\mathrm{d}\Lambda(a^{n_{\text{batch}}}, y^{n_{\text{batch}}})}{\mathrm{d}w}(v) \approx \gamma \frac{\mathrm{d}\Lambda(a^n, y^n)}{\mathrm{d}w}(v). \tag{A.26}$$

By splitting the data set into batches of size n_{batch}, within one epoch, we would update the parameters n/n_{batch} times.

In the extreme case, we could use a batch size of $n_{\text{batch}} = 1$ and calculate the derivatives sample by sample, each time updating the parameter, i.e.,

$$v \leftarrow v - \gamma \frac{\mathrm{d}\lambda(a_i, y_i}{\mathrm{d}w}(v), \quad i = 0, \dots, n-1. \tag{A.27}$$

This way, we would update the paramters n times for each epoch.

Calculating parameter updates based on batches is called SGD and it is widely used in practice. Some remarks are in place.

1. The learning rate γ and the batch size n_{batch} should be chosen together, i.e., smaller n_{batch} increases the variation of the gradient estimate, which a smaller γ can compensate.

2. A good choice of the batch size n_{batch} may depend on the hardware, e.g., for training on a graphical processing unit (GPU), a specific batch size may be processed most efficiently.

3. Criteria for choosing learning rate and batch size are
   - The training time required to achieve a certain loss. Faster is better.
   - The minimum achieved loss. Lower is better.
   - Whether or not the loss converges to a stable minimum.

A.5. Training, Validation, Testing

For several reasons (e.g., to detect overfitting), it is best practice to split your data set into a training set, a validation set and a test set.

- A good starting point is the 80/10/10 ratio, i.e., to use 80% for training, 10% for validation, and 10% for testing.

- Training set: parameter updates during training based on gradient descent are exclusively done by using data from the training set.

- Validation set: for monitoring *during* training, the loss on the validation set is periodically calculated and reported and/or recorded for later assessment. NB: data from the validation set must *not* be used for training!

- Test set: the loss of the NN *after* training is calculated on the test set. The test loss is the proclaimed performance of the NN. NB: the test loss must *not* be calculated during training!

A.6. Further Reading

More details on backpropagation in NNs can be found, e.g., in [1, Chapter 7]. A good summary of training NNs in `pytorch` is provided in [2, Chapter 5]. In particular,

- [2, Section 5.5] discusses the implementation of backpropagation in `pytorch`.
- [2, Section 5.5.2] lists several advanced variations of SGD including the popular `torch.optim.Adam` optimizer.
- [2, Section 5.5.3] discusses training and validation sets.

The last section of [3, Chapter 1] discussed training, validation, and test sets.

References

[1] R. Rojas, *Neural Networks: A Systematic Introduction*. Springer Science & Business Media, 1996. [Online]. Available: `https://page.mi.fu-berlin.de/rojas/neural/`.

[2] E. Stevens, L. Antiga, and T. Viehmann, *Deep learning with PyTorch*. Manning Publications, 2020.

[3] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019.

A.7. Problems

Problem A.1. (Gradients) Consider the computational graph

$$\begin{tikzcd} {} \arrow[r, "x"] & \boxed{f} \arrow[r, "y"] & {} \end{tikzcd}$$

Calculate the gradient $\mathrm{d}y/\,\mathrm{d}x$ for f being

1. The absolute value $|\cdot|$.
2. The ReLU activation.
3. The logistic activation.
4. The tanh activation.

In the following, treat the message b as constant.

5. For the BCE loss on probabilities $y = \mathrm{bce}(b, p)$, calculate $\mathrm{d}y/\,\mathrm{d}p$.
6. For the BCE loss on LPRs $y = \mathrm{bce}_{\log}(b, \ell)$, calculate $\mathrm{d}y/\,\mathrm{d}\ell$.

Problem A.2. (Gradient descent) Consider a single linear neuron as demapper for BPSK in AWGN. Initialize the weight to $w = 1$, the bias to $b = 0.1$ and assume message bit $a = 1$, channel input $x = 1$, channel output $y = 0.5$.

1. Calculate the NN output ℓ by forward propagation.
2. Calculate the gradient
$$\frac{\mathrm{d}\,\mathrm{bce}_{\log}(a, \ell)}{\mathrm{d}w}(1) \tag{A.28}$$
by back propagation.
3. Redo steps 1. and 2. using `torch.tensor` variables with `requires_grad=True` and calling the `.backward()` method. Compare the local gradients to the local gradients you calculated by hand.
Hint: to access the gradient of intermediate variable `z`, call `z.retain_grad()`.
4. For learning rate $\gamma = 0.01$, calculate an update for w using gradient descent.
5. Use `torch.optim.SGD` with `lr=0.01` to calculate an update for w and compare to the update you calculated by hand in 4.

Problem A.3. (Stochastic Gradient Descent) In this problem, you train an equalizer for a dataset `x.txt` (the transmitted symbols) and `y.txt` (the received distorted signal oversampled with 2 samples per symbol.)

1. Preparation:

- Split the data set into a training set, a validation set, and a test set using the 80/10/10 ratio.
- Use the optimizer `torch.optim.SGD` with `lr=learning_rate`.
- Write a training loop with an outer loop over `n_epochs` epochs and an inner loop over batches of size `batch_size`. Use `torch.utils.data.DataLoader` for sampling batches.
- For each epoch, store the loss of the last batch and the loss on the validation set.
- Exit the training loop after 10s.

2. Plot training loss and validation loss against the epoch index. Also, display in the same plot the test loss against the last epoch index.

3. Search for the combination of `learning_rate` and `batch_size` that results in the best validation loss (after 10s of training).

4. Do you achieve a better validation loss when using `torch.optim.Adam` as optimizer?