BOSTON UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

Thesis

LEARNING COMMUNICATION CHANNELS WITH AUTOENCODERS

by

FABIAN HEINRICH

B.S., Universität der Bundeswehr München, 2018

Submitted in partial fulfillment of the

requirements for the degree of

Master of Science

2020

Approved by

First Reader

Andreas Knopp, Dr. Ing.
Professor of Signal Processing

Second Reader

Sang Chin, Ph.D.
Professor of Computer Science
Director of LISP (Learning, Intelligence + Signal Processing)

Third Reader

Thomas Delamotte, Dr.-Ing.
Group Leader of Digital Satellite Payloads & Satellite Monitoring Group

Fourth Reader

Tony Colin, Dipl.-Ing.
Research Fellow

Acknowledgments

This page is dedicated to all persons who have been part of my academic career so far.

I would like to express my deep and sincere gratitude to Professor Peter Chin at the Department of Computer Science & Hariri Institute for Computing at Boston University. He has been a great mentor to me since the day we met. I am very grateful for the possibility to join the LISP research group at Boston University by Professor Chin's invitation. I was able to participate in several research projects and not to forget the huge amount of time I was able to spend at Boston University. Even though nobody could imagine that the world would turn upside down by the beginning of this year, I was pleased to mentor your classes via online teaching and I am looking forward to seeing you as soon as possible.

Furthermore, I cannot express enough thank to Professor Andreas Knopp who has been a great teacher and motivated me to see the bigger picture during class. He inspired me to extend my academic commitment and supports my wish to become a research assistant by all available means. Together with his colleagues Tony Colin and Thomas Delamotte, they advise this thesis at Universität der Bundeswehr München. Therefore, I want to express my appreciation for Tony and Thomas for providing a perfect framework by weekly online meetings and emails.

I would like to acknowledge Professor Thomas Apel, Professor Cornelius Greither and Professor Peter Hertling for offering me to work as a student assistant for more than two years. I have to admit that I did not learn the majority of my skills and knowledge in class but rather during my time as student assistant. The opportunity to work in a teaching position mentoring students during labs had a tremendous influence on my personal development and my academic success. As part of this experience, I want to thank Alessandro Cobbe, Christof Haubner, Philip Janicki and Sören Kleine.

Contents

List of Figures

List of Abbrevations

$BLER$ Block Error Rate. 21

SNR Signal-to-Noise Ratio. 19

Adam Adaptive Moment Estimation. 14

AWGN Additive white Gaussian Noise. 19

GANs Generative Adversarial Networks. 17

GD Gradient Descent. 13

GPUs Graphic Processing Units. 2

HPA High Power Amplifier. 1

ML Machine Learning. 1

NN Neural Network. 8

SGD Stochastic Gradient Descent. 13

Introduction

The success of Machine Learning (ML) algorithms in image and object recognition has shown that these methods have been outperforming expert designed algorithms in various fields of application. Due to the level of prestige that ML has achieved so far, we hope to benefit in a similar way when it comes to applications in communication systems. Most of the algorithms used in communications assume mathematical models that they rely on. However, these models can never capture real-world problems. For this reason, one part of this thesis is to show how to use existing ML algorithms to model the imperfections of the real world. For this reason we want to make use of neural networks and the huge amount of capabilities they have already given proof of in other fields of application. We will consider non-linearities caused by High Power Amplifier (HPA) that are approximated by a mathematical model and show how neural networks can be trained to model a specific hardware configuration.

Moreover, thinking about communications models recalls the picture of several tasks in a block diagram that are processed step by step and optimized separately. To cap it all, the optimization of the single components brings us back to the point mentioned before. Why? Because they use mathematical models to optimize the single components. If we can include all the components in one model, end-to-end optimization will promise the optimization of every single component with respect to the optimization of all other components. Nevertheless, we do not want to talk down all existing models and expert knowledge that it is based on but rather include this knowledge to benefit even more.

Another point that needs to be underlined is the power and speed of computation

provided by the current hardware, especially Graphic Processing Units (GPUs). Due to that, we can train models online with every new data point we get. Therefore, the algorithms can be adapted to sudden or temporary changes of the environment as well.

To conclude, we motivated a lot of possibilities ML can be applied to in communication systems. However, this thesis can only represent a small part of the potential applications. For this reason, we focus on uncoded modulation and the possibilities to get rid of a certain channel model.

Chapter 1

Information Theory

In this chapter about information theory we want to lay the foundation of notations, definitions and assumptions that are used in course of this thesis. We assume that the reader is familiar with the fundamentals of information theory. For more details, the reader is advised to refer to (McEliece, 2004) and (Cover, 1999). In this chapter, we only focus on the definitions and theorems that are relevant to the ML techniques in the next chapters.

1.1 Information and Entropy

We consider the source and receiver as discrete random variables X and Y with alphabets $\mathcal{X}$ and † and the appropriate probability mass functions $p(x)$ and $p(y)$. At this point, we want to underline that $p(x)$ and $p(y)$ are assigned to different random variables and are indeed two different probability mass functions.

Definition 1.1. Assuming that X is a discrete random variable on the finite range $\mathcal{X} = \{x_1, \ldots, x_n\}$, we define the information of an event $\{X = x\}$ by

$$I_b(x) = -\log_b P(X = x), \tag{1.1}$$

with $b \geq 2, b \in \mathbb{N}$. Furthermore, we set $-\log_b(0) = \inf$. In case of $b = 2$ we refer to $I_b(x)$ as $I(x)$. The choice of base b of the logarithm determines the unit information is measured with as well. The most common used basis are base-2 and base-e the units referring to these basis are called *bits* and *nats*.

According to the communication model we introduced at the beginning of this

thesis we need a source and a receiver to describe the exchange of messages. Without further explanation we assume the source and receiver to be discrete random variables X, Y with alphabet X, Y and the appropriate probability mass functions $p(x)$, $p(y)$. At this point we want to underline that $p(x)$ and $p(y)$ are assigned to different random variables and are indeed two different probability mass functions.

Since the source is a discrete random variable X with probability mass function $p(x)$ we can provide a more practical view of the information. As $p(x)$ can only take values in $[0, 1]$ and the fact that $-\log(p(x))$ is positive and monotonically increasing for values of $p(x) \in [0, 1]$ concludes that the higher the probability of sending a message is, the lower is the information that results from this message and vice versa.

Definition 1.2. We define the entropy of a discrete random variable X with probability mass function p by

$$H(X) = -\sum_{i=1}^{n} p_i \log p_i.$$

1.2 Conditional Entropy

Recalling our communication model, we do not know the joint probability distribution of source and receiver as we usually only observe the received symbols. However, we can still gain certain knowledge about the symbols that were sent from the source.

Definition 1.3.

$$H(X \mid Y = y) = -\sum_{x} p(x \mid y) \log p(x \mid y)$$

. As this quantity is a random variable itself, we define the *conditional entropy* of two random variables X and Y as

$$\begin{aligned} H(X \mid Y) &= \sum_{y} p(y) H(X \mid Y = y) && (1.2) \\ &= -\sum_{y} \sum_{x} p(x \mid y) \log p(x \mid y) && (1.3) \\ &= -\sum_{x,y} p(x, y) \log p(x \mid y) && (1.4) \end{aligned}$$

The conditional entropy measures the amount of uncertainty that remains about X after observing Y.

1.3 Mutual Information and Kullback-Leibler Distance

Similarly to the way we consider $H(X \mid Y)$ as the uncertainty about X after observing Y, we can consider $H(X)$ as the uncertainty about X before observing Y. Therefore, $H(X) - H(X \mid Y)$ specifies the amount of information provided about X by Y. We refer to this quantity by $I(X,Y)$, the *mutual information* between X and Y.

Theorem 1.4.

$$\begin{aligned} I(X,Y) &= H(X) - H(X \mid Y) && (1.5)\\ &= \sum_{x,y} p(x,y) \log \frac{p(y \mid x)}{p(y))} && (1.6) \end{aligned}$$

Here we can see that, $I(X,Y)$ is the mean value of $I(x,y)$ over the joint sample space of X and Y.

Definition 1.5. The Kullback-Leibler distance between two probability mass functions $p(x)$ and $q(x)$ is defined as

$$\begin{aligned} D(p||q) &= \sum_{x \in X} p(x) \log \frac{p(x)}{q(x)} && (1.7)\\ &= E\left[\log \frac{p(X)}{q(X)})\right]. && (1.8) \end{aligned}$$

In the following we want to derive an estimator for the mutual information that was introduced by (Belghazi et al., 2018). We start by using the Kullback-Leibler distance to represent a formulation for the mutual information

$$\begin{aligned} I(X,Y) &= H(X) - H(X \mid Z) && (1.9)\\ &= D(p(x,y) \mid\mid p(x)p(y)) && (1.10)\\ &= E\left[\log \frac{p(x,y)}{p(x)p(y)}\right]. && (1.11) \end{aligned}$$

Furthermore, we use the Donsker-Varadhan representation that admits a dual representation of the Kullback-Leibler distance, so that

$$I(X,Y) = \sup_{g:\Omega\to\mathbb{R}} E_p(g(X,Y)) - \log E_q(\exp g(X,Y)), \tag{1.12}$$

with the supremum over all measurable functions g with an existing mean value. Therefore, we obtain a lower bound of the mutual information, so that we can compute an estimation of the mutual information by drawing empirical and i.i.d samples from X and Y. As we can see the right-hand-side provides a lower-bound of the mutual information. For this purpose, we can compute an estimation using empirical samples that are i.i.d.

Chapter 2

Machine Learning

ML can be considered as a tool that turns information into knowledge. Therefore, it finds underlying patterns within the given data that would be undetected otherwise. After discovering the hidden patterns they can be used to predict new data and perform complex decision-making. We can classify ML problems into three main categories: supervised, unsupervised and reinforcement learning. Supervised learning models use labeled data to learn a mapping between the input set and the output set. Typical unsupervised models are classification models and regression models. As we cannot provide labeled data for all kinds of problems there are unsupervised learning algorithms that can handle input data only. Unsupervised learning takes only input data. Due to that, the problem becomes less defined since we do not how the underlying patterns look like from the input data. For this reasons, the algorithm has to find the patterns by extracting important features from the data. Common unsupervised algorithms are clustering, association and autoencoders. Reinforcement learning uses an agent to find the optimal way to achieve a certain goal. If you can design a reward for a problem, you can use it to create a feedback loop between the learning process and the problem. You can maximize the reward by applying the feedback loop over and over again. Reinforcement algorithms have been applied to teach a machine on how to play games like Go or video games.

In this chapter we want to introduce the basic concepts of ML techniques that necessary to improve the concept of autoencoders. We examine the basic principles of

feed-forward networks that were taken from (Bishop, 2006).

2.1 Feed-forward Networks

The basic Neural Network (NN) model can be characterized by a series of functional transformations. Considering the first (1) layer of a model consisting of M units we can compute the input activation $a_j,\ j = 1, \ldots, M$ of every unit and input $x_1, \ldots, x_D$ by

$$a_j^{(1)} = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}. \tag{2.1}$$

The parameters $w_{ji}^{(1)}$ are referred as *weights* where the subscript describes to connection from unit i of the previous layer to the unit j of the current layer. Additionally, $w_{j0}^{(1)}$ are called *biases* that are added to every unit on the layer. After computing the activation of each unit it will be transformed by a differentiable and nonlinear activation function $h(\cdot)$. (PyTorch, 2020a) provides a big choice for different activation functions including their documentation. Applying $h(\cdot)$ to the input activation leads to the *unit activations*

$$z_j^{(1)} = h(a_j^{(1)}). \tag{2.2}$$

To construct the outputs of our model, we combine the unit activations linearly which gives us

$$a_k^{(2)} = \sum_{i=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)}. \tag{2.3}$$

This computation leads to the activations of the second layer in the network. As we consider a two layer network we need to transform the activations of the last layer to proper network outputs y_k. Moreover, we can included the biases into the weight parameters by defining a single additional input variable. The proof for this procedure can be found in (Bishop, 2006) alongside a more detailed description of the annotations we introduced in this section.

2.2 Training of a Neural Network

In the previous, section we introduced feed-forward NNs as a general class of parametric nonlinear functions that maps an input vector x to an output vector y. As we mentioned before, the nonlinear activation functions that are used in a feed-forward network are differentiable. This property will lay the foundation of the training process that is supposed to optimize the weights.

Considering a sum-of-squares loss function (error function) over a given training set of N input vectors $\{x_n\}$ and target vectors $\{t_n\}$ we minimize the following loss function

$$E(w) = \frac{1}{2} \sum_{n=1}^{N} || \ y(x_n, w) - t_n \ ||^2 \ . \tag{2.4}$$

At this point, we want to introduce a graphical interpretation of the loss function $E(w)$ as a surface sitting over the weight space. If w_A is a local minimum, whereas w_B is the global minimum of the loss function $E(w)$. We can take a small step from w to $w + \delta w$, the change of the loss function can be denoted by $\delta E \approx \delta w^T \nabla E(w)$ where $\nabla E(w)$ is a vector pointing in the direction of the greatest increase of $E(w)$. Because $E(w)$ is a smooth continuous function of w, we can find its smallest value at a point where the gradient of the loss function vanishes so that $\nabla E(w) = 0$. If the gradient does not vanishes we can keep going into the direction of $-\nabla E(w)$ to reduce the error. However, since the loss function is highly nonlinear dependent on the weights, we may find several points the gradient vanishes. These points can be local minima, global minima or saddle points. Nevertheless, finding one of these points does not tell us whether it is a global minima or not. In the following, we will introduce algorithms that optimize continuous nonlinear functions using iterative schemes, so that

$$w^{(\tau+1)} = w^{(\tau)} + \Delta w^{(\tau)}. \tag{2.5}$$

If we think about evaluating the gradient of the loss function step by step, it makes perfect sense if we examine a local approximation the loss function to decrease complexity. For this reason, we consider a second-order Taylor approximation of the loss function $E(w)$ around a point $\hat{w}$ in the weight space

$$E(w) \approx E(\hat{w}) + (w - \hat{w})^T b + \frac{1}{2}(w - \hat{w})^T H(w - \hat{w}). \tag{2.6}$$

Here H denotes the Hessian Matrix of the loss function and b is the gradient of E evaluated at $\hat{w}$, so that

$$b = \nabla E|_{w=\hat{w}}\,, \tag{2.7}$$

$$(H)_{ij} = \left.\frac{\partial E}{\partial w_i \partial w_j}\right|_{w=\hat{w}}. \tag{2.8}$$

At this point we should think about computational complexity with regard to the total number W of weights and biases of a network as well. The computation of 2.6 depends on $O(W^2)$ parameters given by H and b. Therefore, we need to compute $O(W^2)$ function evaluation each of them requires $O(W)$ steps. Due to that the complexity of determining the minimum with (2.6) is $O(W^3)$ (Bishop, 2006).

2.3 Error Backpropagation Algorithm

The following algorithm is called *error backpropagation* or *backprop*. We can separate this approach into two different steps. The first step is to evaluate the derivatives of the loss function with respect to the weights. Afterwards we need to compute the modifications of the weights using the derivatives.

The loss functions we will examine at this point consist of a sum of terms, one for each of the N data points in the training set

$$E(w) = \sum_{n=1}^{N} E_n(w). \tag{2.9}$$

In the following we examine the evaluation for one such term in the loss function. At first, we consider an output layer with outputs y_k and a linear activation function, so that $y_k = \sum_i w_{ki} x_i$, where x_i are input variables. Applying a sum-of-squares loss function takes the form

$$E_n = \frac{1}{2} \sum_k (y_{nk} - t_{nk})^2, \tag{2.10}$$

where $y_{nk} = y_k(x_n, w)$. The gradient of a sum-of-square loss function with respect to a weight w_{ij} is given by

$$\frac{\partial E_n}{\partial w_{ji}} = (y_{nj} - t_{nj}) x_{ni}. \tag{2.11}$$

If we generalize the two layer model from 2.1 we can derive the following formulas for the input activation a_j and the unit activation z_j for general hidden layers. Therefore

$$a_j = \sum_i w_{ji} z_i, \tag{2.12}$$

$$z_j = h(a_j). \tag{2.13}$$

These two equations describe the forward propagation since the sum runs over i.
For a hidden layer that uses a nonlinear activation function we determine the gradient of the loss function by applying the chain rule for partial derivatives

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}, \tag{2.14}$$

with

$$\delta_j = \frac{\partial E_n}{\partial a_j} \tag{2.15}$$

and $\frac{\partial a_j}{\partial w_j i} = z_i$ from (2.12) we conclude

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i. \tag{2.16}$$

Using equation (2.16) we can compute the derivative of the loss function E_n with respect to weight w_{ji} by multiplying the value of δ with the value of z for the unit. Obtaining the values of δ we need to distinguish between output units and hidden units. To derive the equation for outputs units from (2.11), we need to apply the chain rule to compute the values of δ for hidden units. Therefore, for outputs units

$$\delta_k = y_k - t_k, \tag{2.17}$$

and for hidden units

$$\delta_j = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j}, \tag{2.18}$$

where the sum runs over all units k to which unit j is connected with. In the following, we will derive the final formulation for the error backpropation. Using (2.15) to substitute the outer derivative and (2.12) to substitute a_k, we conclude

$$\delta_j = \sum_k \delta_k \frac{\partial \sum_i w_{ki} z_i}{\partial a_j}. \tag{2.19}$$

Now we substitute z_i with $h(a_i)$ from (2.13) and take the partial derivative with respect to a_j, so that

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k. \tag{2.20}$$

Here we see where the name backpropagation comes from directly since the values of k in (2.20) are taken over the first index of the weights w_{kj} which refers to the units higher up in the NN that have incoming connections from unit j,whereas in (2.12) the sum runs over the second index of the weights which computes the forward propagation.

The error backpropagation algorithm has a computational complexity of $O(W)$ since a single evaluation of the loss function requires $O(W)$ operations and the effort that is used to compute the forward computation of (2.12) is surpassed by $O(W)$ since the

number of weights is much greater than the number of hidden units.

We want to make use of the gradient information to lower the overall order of the computational complexity by one. Since we receive W items of information with each evaluation of ∇E, we only need to perform $O(W)$ evaluations at the costs of $O(W)$ steps for the error backpropagation. Therefore, we can find the minimum in $O(W^2)$ steps now.

2.4 Optimization

2.4.1 Gradient Descent Optimization

A straight forward approach that uses the information given by the gradient is to conduct a small step in the direction of the negative gradient by updating the weights. This can be described by

$$w^{(\tau+1)} = w^{(\tau)} - \eta \nabla E(w^{(\tau)}), \tag{2.21}$$

where η is referred to as *learning rate*. This procedure is repeated in so called *epochs*. Therefore, it guarantees to converge to a local minimum for a non-convex loss function for a given number of epochs. Nevertheless, we want to underline that this approach uses the loss function applied on the whole training set which can cause problems if the data set exceeds the available memory resources. Furthermore, the model cannot handle a continuous stream of new data points which is referred to as *online* learning and unsustainable for a lot of applications. However, we can perform minor changes to batch methods to prevent the problems mentioned.

2.4.2 Stochastic Gradient Descent

While the Gradient Descent (GD) Optimization runs on the entire training set, the Stochastic Gradient Descent (SGD) Optimization (Ruder, 2016) updates the weights

on one data point with index n at a time, so that

$$w^{(\tau+1)} = w^{(\tau)} - \eta \nabla E_n(w^{(\tau)}). \tag{2.22}$$

This approach causes a higher variance in the change of the loss function. Therefore, it can jump out of a local minimum and to find a potentially better local minimum. However, this degrades the convergence to the global minimum. To avoid jumping between local minima without converging we can decrease the learning rate during training to stabilize the optimization process. It is clear that Stochastic Gradient Descent optimization enables online learning in contrast to the batch methods. Nevertheless, bypassing one problem usually causes a new problem. To reduce the variance of the updated weights and to ensure more stable convergence we can simply take a subset of the training set to compute the weight updates. The size of this subset is referred to as *batch size*. The batch size has to be adapted to the general training set properties and especially to the number of target classes.

2.4.3 Adaptive Moment Estimation

Visualizing the way how SGD approaches a local minimum can help to understand other weaknesses when it comes to the convergence of SGD. Since it is typical for the surface of the loss function to change more steeply in one dimension than in another close to a local minimum, SGD has issues with these areas. One of the algorithms that copes with this problem by using moments is called Adaptive Moment Estimation (Adam) (Kingma and Ba, 2014). It computes an adaptive learning rate and uses exponentially decreasing first and second moments of the previous gradient

$$m_1^{(\tau+1)} = \beta_1 m_1^{(\tau)} + (1-\beta_1)\nabla E_n(w), \tag{2.23}$$

$$m_2^{(\tau+1)} = \beta_2 m_2^{(\tau)} + (1-\beta_2)(\nabla E_n(w))^2. \tag{2.24}$$

Due to numerical reasons and the initialization, the moments need to be corrected, so that

$$\hat{m}_1^{(\tau)} = \frac{m_1^{(\tau)}}{1-\beta_1^{\tau}} \tag{2.25}$$

$$\hat{m}_2^{(\tau)} = \frac{m_2^{(\tau)}}{1-\beta_2^{\tau}}. \tag{2.26}$$

Finally it yields in the following update rule

$$w^{(\tau+1)} = w^{(\tau)} - \frac{\eta}{\sqrt{\hat{m}_2^{(\tau)}} + \epsilon} \hat{m}_1^{(\tau)} \tag{2.27}$$

Common default values that were determined in (Kingma and Ba, 2014) are $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$.

2.5 Loss Function

There are several loss functions that can be used to optimize the trainable parameters. However, they need to match the output unit activation function and the type of problem to be solved. For regression, we use the same functions as for the example in 2.3, linear outputs and sum-of-squares error. Binary classifications use logistic sigmoid outputs and cross-entropy loss functions. Moreover, we use softmax outputs and multiclass cross-entropy loss functions for multiclass classifications. The mathematical definitions and documentation of these loss functions a lot more can be found here (PyTorch, 2020b). We recommend to check the definition and implementation of loss functions carefully since they sometimes vary from environment to environment.

2.6 Autoencoder

Autoencoders have already been examined since the late 80s (Ballard, 1987). In this section we want to introduce the general idea of autoencoders to extend this idea

with aspects of communication systems later on.

An autoencoder is a NN that is supposed to reproduce its input x as its output y. Thereby, it has a hidden layer h that is supposed to compute a different representation of the input. For this reason, it consists of two parts. The first part is referred to as *encoder*, whereas the second part is the *decoder*. Both can be considered as functions that conduct the appropriate mappings. The encoder functions f applies the mapping from the input to the computed representation $h = f(x)$, whereas the decoder function g takes the input h and maps it to a reconstruction $r = g(h)$. The desired scenario is to obtain the input given to the encoder at the output, so that $g(f(x)) = x$. Furthermore, we have to put certain restrictions to the hidden layer h since we do not want the NN to simply copy the input. The most common restrictions that are applied with respect to the dimension that is used for the representation. Of course, it is desired to find a low-dimensional representation of the input. A good example of a task that takes regard of the principle of an autoencoder would be the compression of a text file. Thereby, we usually remove redundancies in the file and extract the important features which refer to the restricted dimension of representation given by the hidden layer h. Nevertheless, we can restore the original file without any losses. At this point, we want to highlight the question whether the learning process of an autoencoder is to be considered as supervised or unsupervised. We cannot give a precise answer to this question since the autoencoder does not use distinct target values. The labels are actually the inputs and due to that it is often referred to as unsupervised training. However, the algorithms we use to train are used for supervised training.

Autoencoders have been playing an important role in image denoising and feature extraction for many years and offer a lot of possibilities to do research. Nonetheless, we do not focus on autoencoders in general since we want to focus on the aspect of

communication systems that make use of autoencoders and can therefore compete with already existing expert-designed algorithms.

2.7 Mutual Information Neural Estimation

Recalling the estimator for the mutual information (1.12), (Belghazi et al., 2018) showed that we can choose a NN that is defined by its trainable parameters θ as a function family T_θ. Due to that we obtain the estimator

$$I(X,Y) \geq \sup_{\theta \in \Theta} E_p \left[T_\theta(X,Y)\right] - \log \left[E_{pq}(\exp T_\theta(X,Y))\right]. \tag{2.28}$$

Furthermore, they showed that this estimator is consistent in converging to the true value for an increasing number of samples drawn.

2.8 Generative Adversarial Networks

Generative Adversarial Networks (GANs) were first introduced by (Goodfellow et al., 2014). Since then they have gained a lot of interest in the ML society. They play an import role in all kinds of computer vision problems as they can be used to create graphical models or to up-scale low-resolution pictures. However, we do not want to study GANs in general at his point. We want to use GANs as a component in the overall concept of a communication model based on an autoencoder. For this reason, we will only refer to a very basic description of GANs and we will not provide all mathematical formulations needed to describe GANs. Nevertheless, we refer the reader to the given literature for more details.

GANs consist of a generator and a discriminator. The generator is supposed to generate new data that is similar to the expected data. Therefore, it uses random noise to mimic the data by creating fake samples. The discriminator has to decide whether the input belongs to the data or not. Therefore, it distinguishes the generated data

between fake and real. Both parts are modeled by NNs and trained simultaneously so that the generateor and the discriminator improve their performance by competing against each other (minimizing the the probability of generated samples being detected as fake). A common example that illustrates the procedure of GANs is the comparison of the generator with an art forger and the discriminator with an art expert. The generator tries to tamper the painting of a certain painter and the discriminator has to decide whether a given painting is fake or not.

In our case we want the generator to generate samples of a certain joint probability distribution $p(x, y)$ from random noise. To do so we train the discriminator on samples of the conditional probability distribution $p(y \mid x)$. Therefore, we use the generator to create samples of $p(x, y)$. The discriminator has been trained with samples of $p(y \mid x)$ and decides if the generated the samples of $p(x, y)$ are fake or real.

Chapter 3

Machine Learning in Communication Systems

3.1 Channel Models

3.1.1 Additive White Gaussian Noise Channel

The Additive white Gaussian Noise (AWGN) channel is a very common and simple way to model certain impairments to communication during a transmission. It adds white noise with a constant spectral density and a Gaussian distributed amplitude. If we consider a discrete random variable X as input to an AWGN channel, we can describe the output Y by

$$X = Y + N \tag{3.1}$$

where N is normally distributed with zero mean variance σ^2 denotes a normal distribution with zero mean and variance σ^2. Furthermore, σ^2 denotes the power of the noise. We will use the Signal-to-Noise Ratio (SNR) value to describe the AWGN channel,

$$\begin{aligned} SNR &= \frac{signal\ power}{noise\ power} && (3.2)\\ &= \frac{E_s}{N_0} && (3.3)\\ &= \frac{k\ n\ E_b}{N_0}, && (3.4)\\ SNR_{dB} &= \frac{E_b}{N_{0\,dB}} + 10\log_{10} k\ n, && (3.5) \end{aligned}$$

where k denotes the number of bits per symbol and n the number of complex baseband symbols. For an autoencoder with $n = 1$ we consider a complex symbol as an two dimensional vector. Therefore, we add AWGN with the standard deviation $\sigma = \sqrt{\frac{10^{-SNR_{dB}/10}}{2}}$ to each component of the vector.

3.1.2 Transmission High Power Amplifier

We want to extend the AWGN channel to a more complex channel simulation including nonlinearities. Due to that, we consider upsampling, pulse-shaping, an high power amplifier, AWGN, downsampling and matched filtering. block diagram of the channel simulation can be seen below.

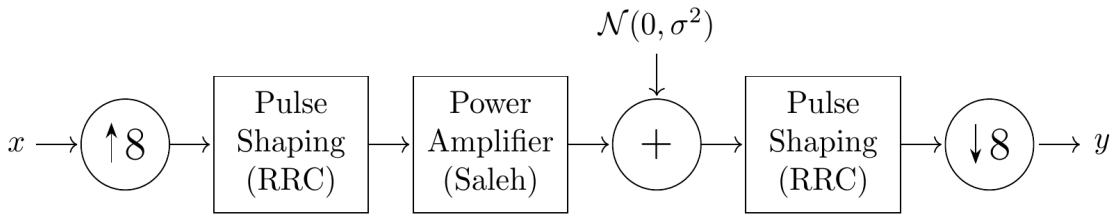


Figure 3·1: Transmission Model including HPA

We use an upsampling factor of 8, an square-root-cosine-filter with roll-off-factor of 0.3 and a span of 16. After pulse shaping we pass the stream through a normalized memoryless Saleh amplifier model. Next, we apply AWGN to the stream and perform matched filtering. The last step is to perform the downsampling to obtain the received symbols.

3.2 Channel Autoencoder

Channel Autoencoder were introduced by (O'Shea et al., 2016), (O'Shea and Hoydis, 2017) and gained a lot of interest in the following years (Dörner et al., 2017). If we want to send a message $s \in \mathcal{M} = \{1, \ldots, M\}$ to a receiver, the transmitter applies a

function $f : \mathcal{M} \to \mathbb{C}^n$ to a message s where n denotes the number complex baseband symbols. Moreover, there is a power constraint on the transmitted element $x = f(s)$, e.g. $E(\|x\|^2) \leq n$. When x is transmitted via the channel, we assume that it will be altered in a certain way. For this reason, we obtain a new element $y \in \mathbb{C}^n$ after the channel. We will implement the channel models introduced in this chapter and study their influence on the autoencoders learned constellation. Receiving the altered symbol, forces to find a function $g : \mathbb{C}^n \to \mathcal{M}$ that computes an estimation $\hat{s} = g(y)$ for the transmitted message s since we have no explicit knowledge about the modifications at the channel. For this reason, we want the receiver to make the right decision for y to match $\hat{s} = s = g(f(s))$. At this point, we can see an analogy to autoencoders introduced in 2.6. We can see the transmitter as the encoder and the receiver as the decoder. The channel model is new contributing to the name *channel autoencoder*.

To realize the model by using an autoencoder we need to train a NN to apply the functional mappings that are done by the transmitter and receiver. Therefore we replace f and g by NNs that perform the desired mappings. We denote the encoder mapping as $f_{\theta_E} : \mathcal{M} \to \mathbb{C}^n$ and the decoder mapping as $g_{\theta_D} : \mathbb{C}^n \to \mathcal{M}$ where θ_E and θ_D describe the trainable parameters of the NNs.

In 2.6 we denoted the represent ion h as the output of the encoder. For the channel autoencoder this representation is the constellation that is learned by the autoencoder. We use the constellation to visualize what the autoencoder is learning. Due to the constellation we can compare autoencoders with existing constellation schemes as well. To evaluate the performance of an autoencoder we use the Block Error Rate ($BLER$) with one block consisting of one symbol. Due to that we simulate a transmission of a certain length n and count the errors n_{errors} that occur, so that

$$BLER = \frac{n_{errors}}{n}. \tag{3.6}$$

3.3 Mutual Information Neural Estimation Autoencoder

A crucial point for the performance of the autoencoder is the channel model used during training. Classical channel models can never embrace the real channel since they do not capture the real world. Adding a MINE can be promising since the optimal transmission rate is a function of the mutual information $I(X, Y)$ between source X and receiver Y. If we recall the definition of the mutual information from 1.3, we know that $I(X, Y)$ is the amount of information about X provided by Y. Therefore, if we obtain $y \in Y$ at the receiver and we have an algorithm to maximize the mutual information we can find out the transmitted symbol $x \in X$. (Fritschek et al., 2019) introduces an approach that separated the encoder from the decoder to optimize the encoder without regard to the decoder. Therefore, it can be trained separately using the optimization from the MINE that draws empirical samples.
Moreover, we introduce a different approach thats keeps the training of encoder and decoder combined. As we receive the best performance when $I(X, Y)$ is maximized, we can use $-I(X, Y)$ as a loss function.

3.4 Generative Adversarial Networks Autoencoder

A crucial point for the performance of the autoencoder is the channel model used during training. Classical channel models can never embrace the real channel since they are simplified. One approach to avoid this problem by using GANs was introduced by (O'Shea et al., 2019) and (O'Shea et al., 2018). In this section, we show how GANs can be used to determine the probability distribution of a channel by drawing empirical samples.
We denote $\hat{y}$ as the estimation of y. The channel approximating network h_{θ_h} with trainable parameters θ_h applies the mapping $\hat{y} = h(x)$ to the encoder output x. Referring to the GAN model introduced in 2.8 the channel approximation network takes

part in the generator. The discriminator is represented by another NN d_{θ_d} with trainable parameters θ_d. Being able to learn the distribution of an arbitrary channel, we show how to include this approach in our existing autoencoder structure.

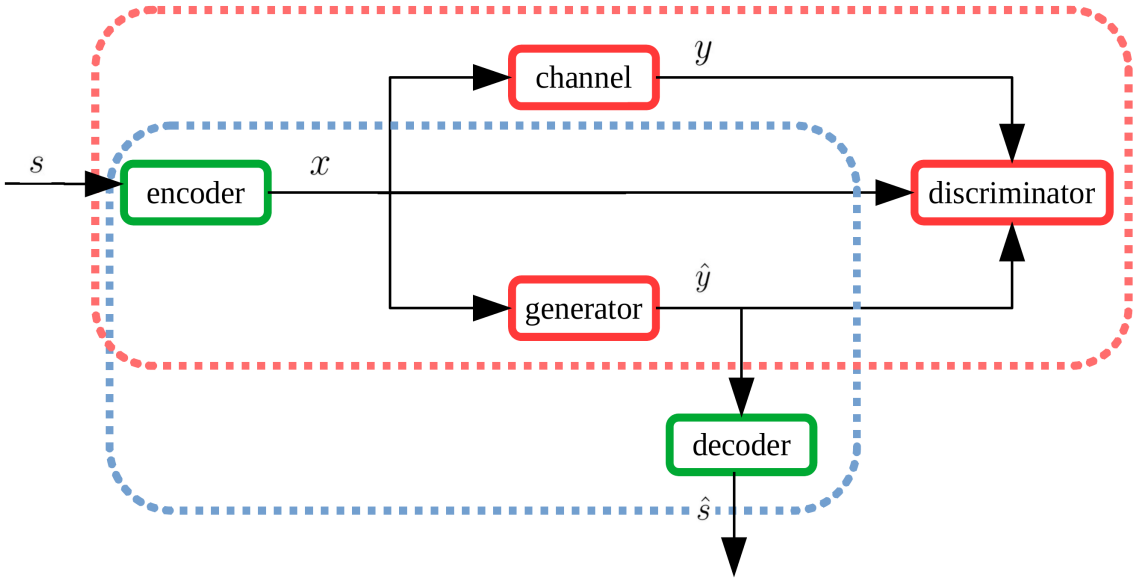


Figure 3·2: Model GAN and autoencoder combined

Chapter 4

Simulations and Results

4.1 Implementation Remarks

In this section, we present our thoughts behind the implementation process and state information concerning the code.

There are several programming libraries that offer state-of-the-art ML algorithms that can be integrated into an existing Python environment easily, e.g. Tensorflow, Keras, Scikit or PyTorch. For the implementation in this thesis we decided to use PyTorch. PyTorch was released in September 2016 by the Facebook AI Research lab. From our point of view, PyTorch offers a lot of advantages as it is clear in its structure and combines the best of all other libraries in one. We want to state a few of the main reasons that have been supporting our decision for PyTorch. A big advantage of PyTorch is the implementation of the actual training process which is usually defined in a for-loop with all training steps listed separately. Therefore, it makes training more transparent since the training is not performed by a single function call. Moreover, the step-by-step implementation offers great possibilities for debugging. Another point is the level of data parallelism and straight forward support of GPU based computation.

We provide all implementations that are used in this thesis on GitHub (Heinrich, 2020). We implemented all models using an object-oriented structure. All models are separated in a different class that inherits all necessary methods. Thereby, we created two parent classes. One containing class specific methods and the other one

features all PyTorch specific methods to keep the classes neat. The object oriented approach makes our code reusable and sustainable to maintain. Therefore, all models are created as a separate instance. When creating an instance of a class the whole network architecture is handed over. Due to that, we can create different instances of the same model with a different architecture to compare the influence of the architecture. Furthermore, our implementation supports the use of a GPU by default implementation. Additionally, we added an animated plot to each training function that shows how the training process actually changes the parameters of the model. We think that this is an important part when it comes to understanding how the ML methods work and how they learn since it gives a visual output instead of just plotting the loss function. For this reason, we highly recommend downloading the code from GitHub (Heinrich, 2020) and run it by yourself and to make use of these animated plots.

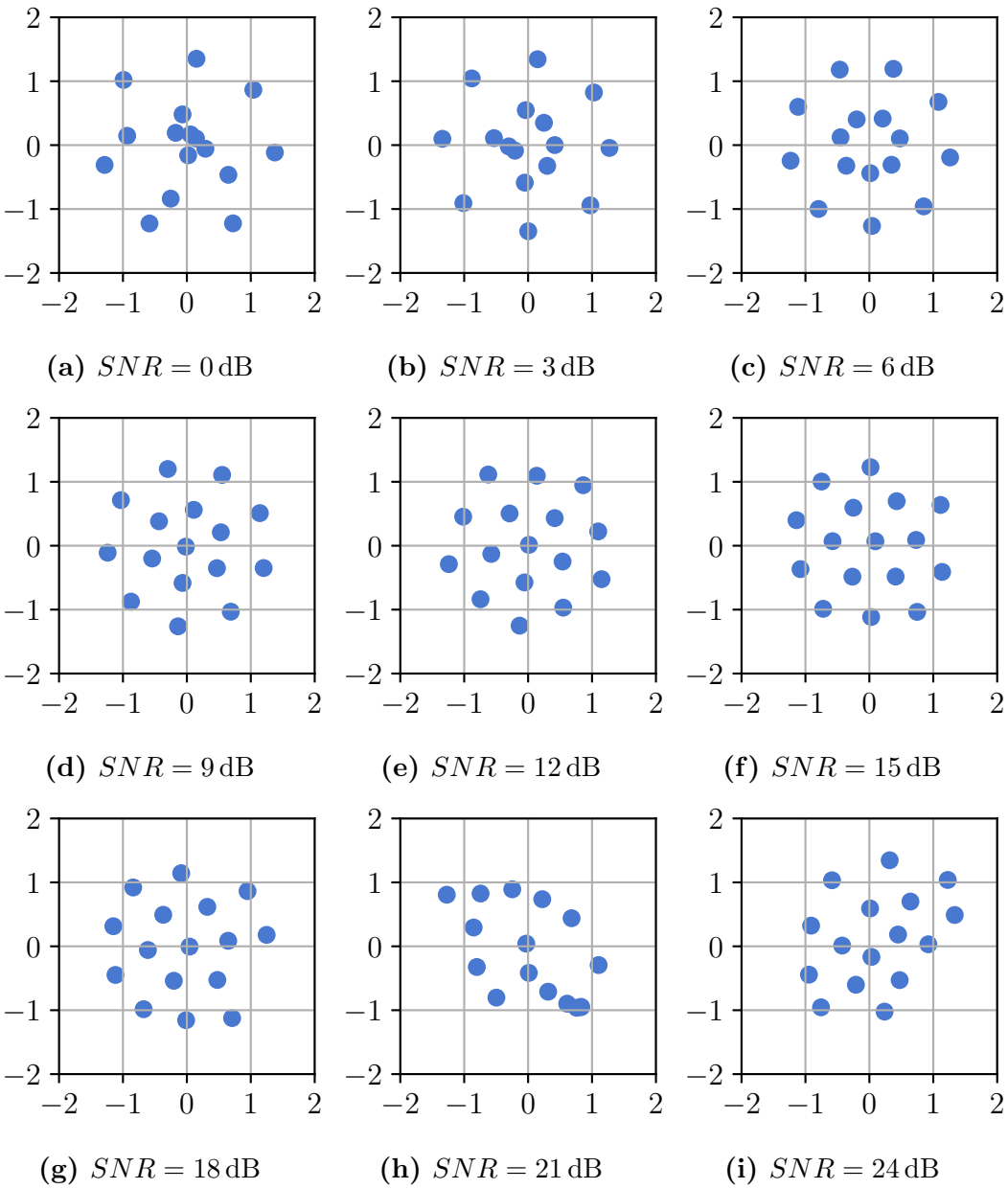
4.2 Implementation Channel Autoencoder

4.2.1 Implementation Additive White Gaussian Noise Channel

We determined the hyperparameters for training by using a graphical presentation of the learned constellation and the loss during the training process. The most important point was that we had to ensure that the training converges. Therefore, we adapted the encoder architecture step by step until we could observe convergence. As the decoder just has to classify the channel outputs, we chose a deep and wide architecture to make sure that it can cope with that problem.

To make convergence more likely we deiced to use an adaptive learning rate during training. Because of that we trained all models 40 epochs with a learning rate of $\eta = 0.01$, 20 epochs with $\eta = 0.005$ and finally 20 epochs with $\eta = 0.001$. Furthermore, we implemented a custom weight initialization at the beginning of each

training since our distinct models trained in a for-loop for different parameters using the CUDA environment by PyTorch. The problem here is that due to the memory allocation that is used by CUDA, the weights as well as the optimizer have a pointer the previous instances in the loop. That caused a lot of interference between models during training. Unfortunately, it is not possible to free the memory of the GPU consistently in this case which forced us the initialize the parameters new at the beginning of each step in the loop. Furthermore, we want to underline that the channel layer uses random generator numbers to apply the AWGN to the symbols. Therefore, the initialization of a seed for the random number generator is only suitable to a limited extent. For this reason, we used random seeds for the weight initialization and in front of the for-loop mentioned before.



(a) $SNR = 0\,\text{dB}$ **(b)** $SNR = 3\,\text{dB}$ **(c)** $SNR = 6\,\text{dB}$

(d) $SNR = 9\,\text{dB}$ **(e)** $SNR = 12\,\text{dB}$ **(f)** $SNR = 15\,\text{dB}$

(g) $SNR = 18\,\text{dB}$ **(h)** $SNR = 21\,\text{dB}$ **(i)** $SNR = 24\,\text{dB}$

Figure 4·1: Learned constellations for $M = 16$ with different SNR values on training

We see that there is an influence of the SNR value that is used during training on the learned constellation. As we can notice that there are three different constellations learned by the autoencoder that assume convergence. At low SNR values, we can see that some symbols are placed on very close to each other. This is caused by the

variance of the noise added on the channel layer which is too high compared to the power of the symbols that is limited by the power constraint. Furthermore, we assume the autoencoder to overfit since the variance of the noise is to low. (You et al., 2019) showed that adding random noise to an autoencoder can prevent overfitting.

The first learned constellation occurs at $SNR = 6\,\text{dB}$ values and consists of two circles centered on the point of origin. The inner circle consistcs of seven symbols, the outer circle of eleven. Both circles are aligned regularly. The second constellation is learned on higher SNR values of 9 dB and 15 dB. It consists of two rings and a single symbol at the point of origin. At 9 dB the inner ring contains six symbols which are arranged as a regular hexagon, whereas at 15 dB there seems to be an outlying point. The third constellation occurs in between the previous ones at $SNR = 2\,\text{dB}$. The depicted graphic for an SNR=12 dB shows a regular pentagon for the inner and the outer circle as well as a single symbol in the point of origin.

The fact that for high SNR values one symbol moves to the point origin seems consistent since the power of this symbol is approximately zero which allows a higher power for the remaining symbols due to the average power constraint. Furthermore, for high SNR values it is not a disadvantage for the symbols in the inner ring to be closer to each other since the spread caused by the AWGN channel is not that wide. In the following we see how the different training SNR and the related constellation affects the $BLER$.

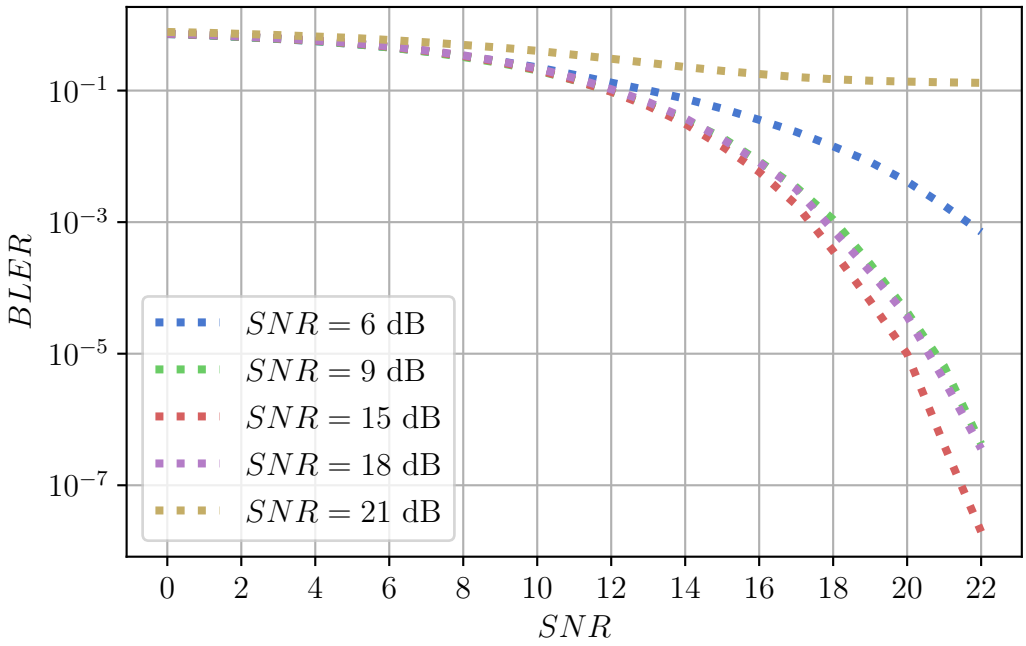


Figure 4·2: $BLER$ for $M = 16$ with different SNR values on training

It shows that the best performing SNR value is 15 dB altough it has one outlier. Since the $BLER$ for both 9 dB and 18 dB are higher than for 15 dB, we assume the optimum SNR to be between 9 dB and 18 dB and close to 15 dB To limit the range for an optimum training SNR, we conducted a series of computation with and plotted the results. Each plot shows the $BLER$ at a certain SNR value over all training $SNRs$.

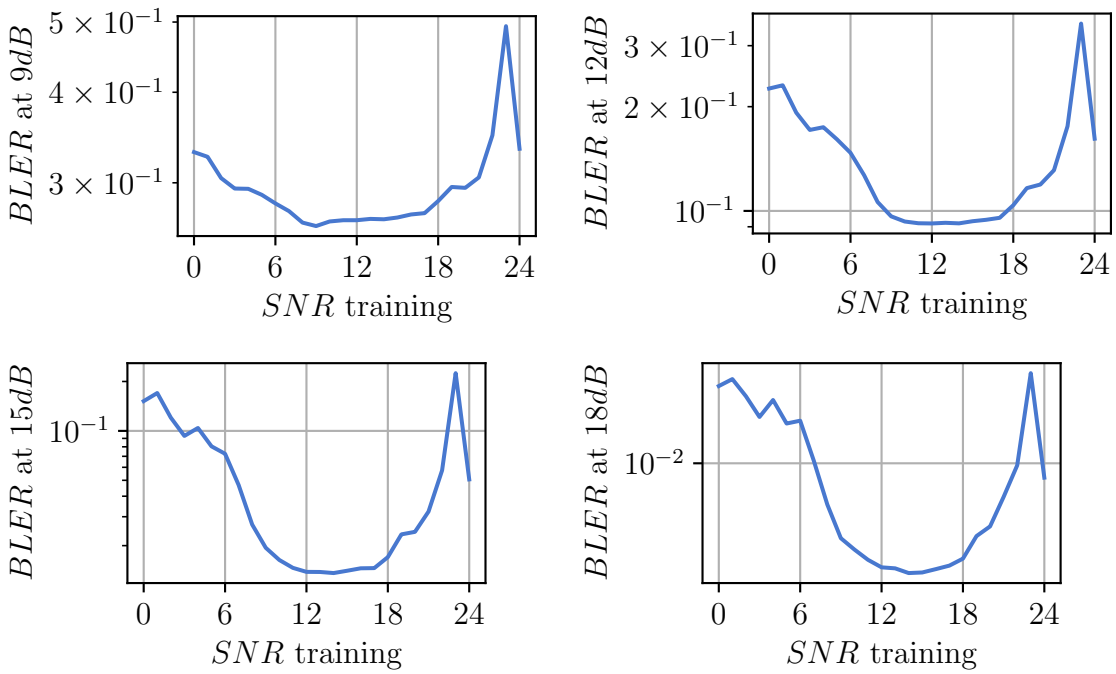


Figure 4·3: $BLER$ at a certain SNR value for $M = 16$ with different SNR values on training

First of all, we can see that for 9 dB, 12 dB and 15 dB the lowest $BLER$ can be found at the SNR value the autoencoder was trained with. The fact that this does not hold for 18 dB verifies that the learning process is disturbed due to the low variance in the noise as we mentioned before. Furthermore, we see that for 9 dB, 12 dB and 15 dB the optimal SNR value is centered around 12 dB. However, for 12 dB and 15 dB it seems to be that the optimal SNR is between 12 dB and 18 dB.

Since there is still some variance on the range of optimal SNR values we want to introduce another channel layer for the AWGN autoencoder that draws the training SNR for each batch from a uniform distribution $\mathcal{U} \sim (12, 15)$.

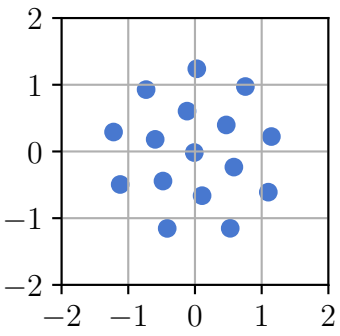


Figure 4·4: $BLER$ for $M = 16$ autoencoder with uniform AWGN channel

We notice that the constellation that was learned using a channel layer of uniform sampled SNR values does not result in a completely new constellation.

At the end of this section we want to compare the $BLER$ of the trained autoencoder with the $BLER$ of the 16QAM constellation from (MathWorks, 2020).

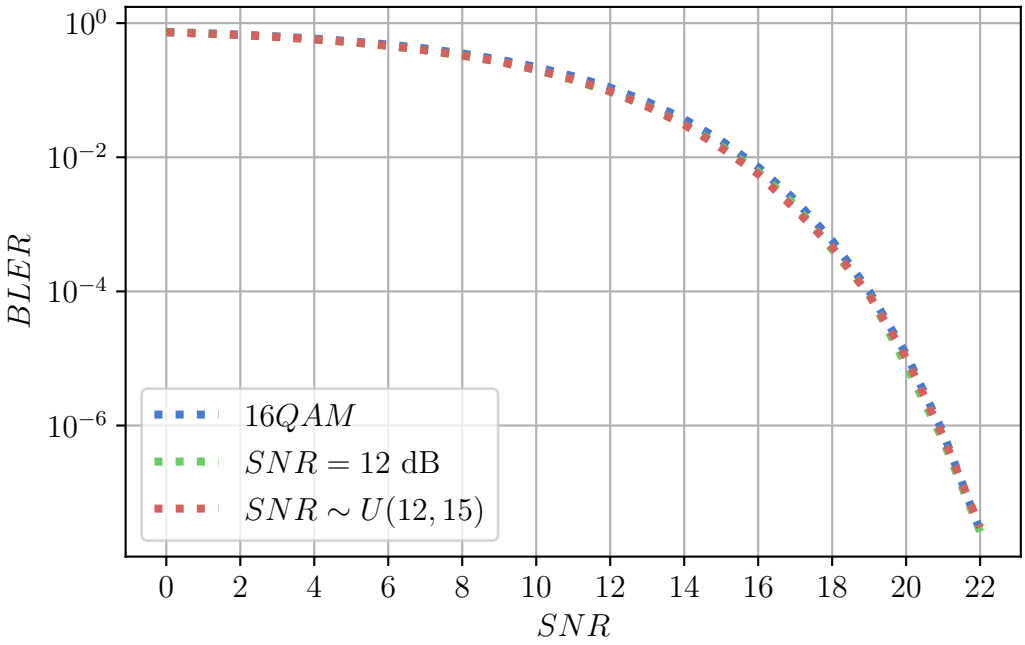


Figure 4·5: $BLER$ for $M = 16$ with different SNR values on training

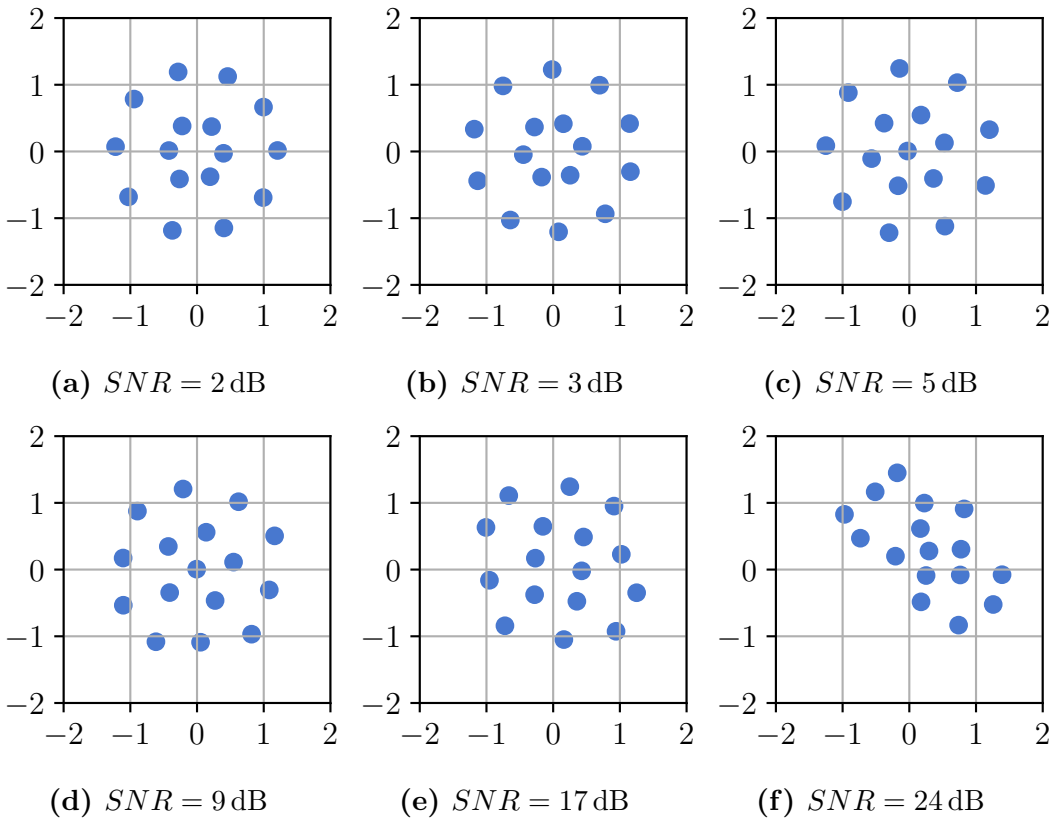
We can see that the autoencoder can compete with the 16QAM and even performs

better at high SNR values.

To sum up this section, we want to highlight that the constellations depicted are not deterministic since we use a random variable in the channel layer. That is the reason why setting seeds for the random number generator might not be useful. One could say that this is a poor property of autoencoders since you cannot reproduce results by one hundred percent. However, from our point of view, this is not a disadvantage at all. It is the exact opposite since it shows how the autoencoder can adopt to a change of the channel it is trained with.

4.2.2 Implementation Transmission High Power Amplifier

In the following we want to use a channel layer for autoencoders that use the simulations introduced in 3.1.2.



(a) $SNR = 2\,\mathrm{dB}$ **(b)** $SNR = 3\,\mathrm{dB}$ **(c)** $SNR = 5\,\mathrm{dB}$

(d) $SNR = 9\,\mathrm{dB}$ **(e)** $SNR = 17\,\mathrm{dB}$ **(f)** $SNR = 24\,\mathrm{dB}$

Figure 4·6: Learned constellations for $M = 16$ with different SNR values on training an input-backoff of 0 dB

We can say that the autoencoder adapts a whole different channel model quite good and derives similar patterns as on the AWGN channel.

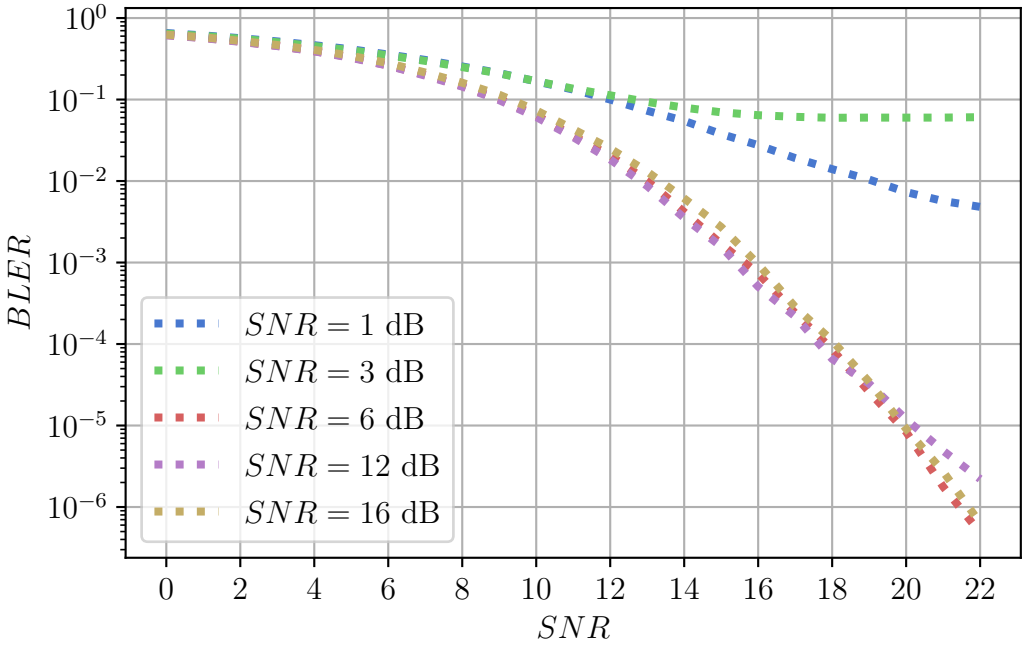
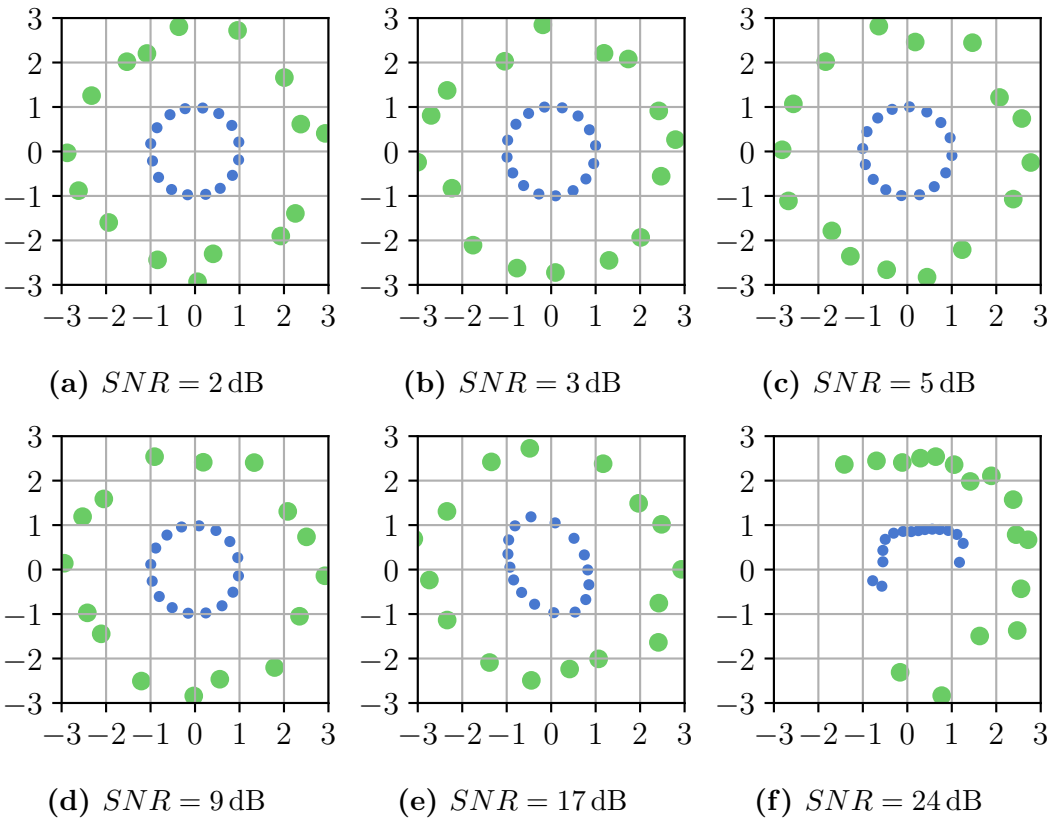


Figure 4·7: BER for $M = 16$ with different E_b/N_0 values on training and input-backoff of 0 dB



(a) $SNR = 2\,\text{dB}$ **(b)** $SNR = 3\,\text{dB}$ **(c)** $SNR = 5\,\text{dB}$

(d) $SNR = 9\,\text{dB}$ **(e)** $SNR = 17\,\text{dB}$ **(f)** $SNR = 24\,\text{dB}$

Figure 4·8: Learned constellations for $M = 16$ with different SNR values on training an input-backoff of 10 dB

These plots show the constellations for an input-backoff of 10 dB. The blue dots are marking the constellation prior to the nonlinearities and the green ones refer to the constellation after the non-linearities were applied. We can see that the autoencoder learns a similar constellation for all SNR values since the non-linear effects are too dominant.

4.3 Implementation Mutual Information Neural Estimation

In this section, we will show the results of our implementation of MINE. These plots show how the MINE estimates the mutual information of a 4QAM and 16QAM constellation on an AWGN channel with different SNR values.

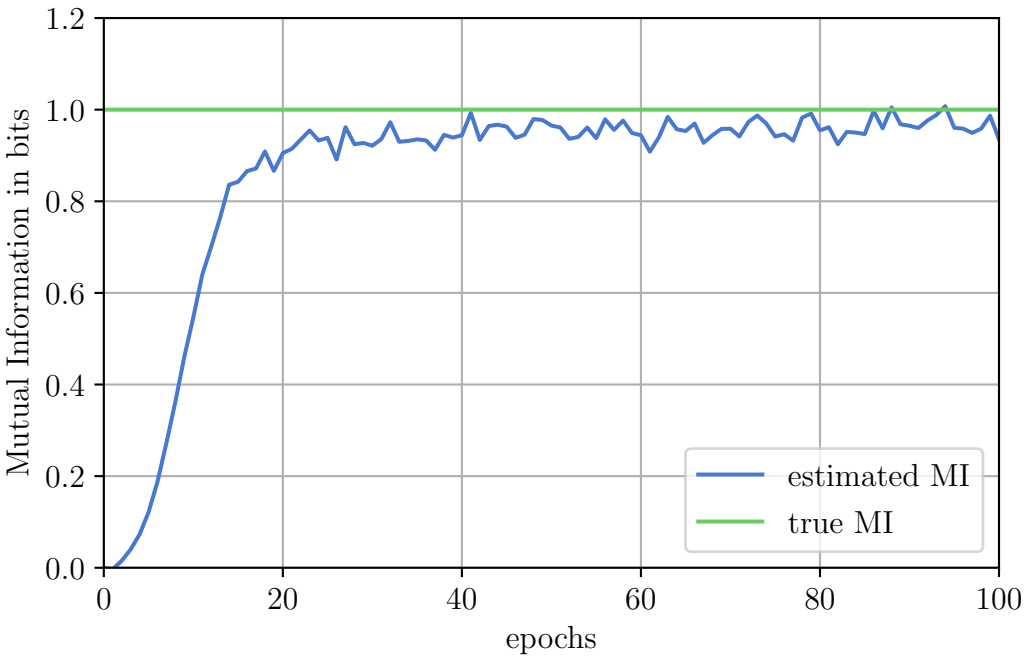


Figure 4·9: MINE estimation of $I(X, Y)$ for 4QAM at $SNR = 0\,\mathrm{dB}$

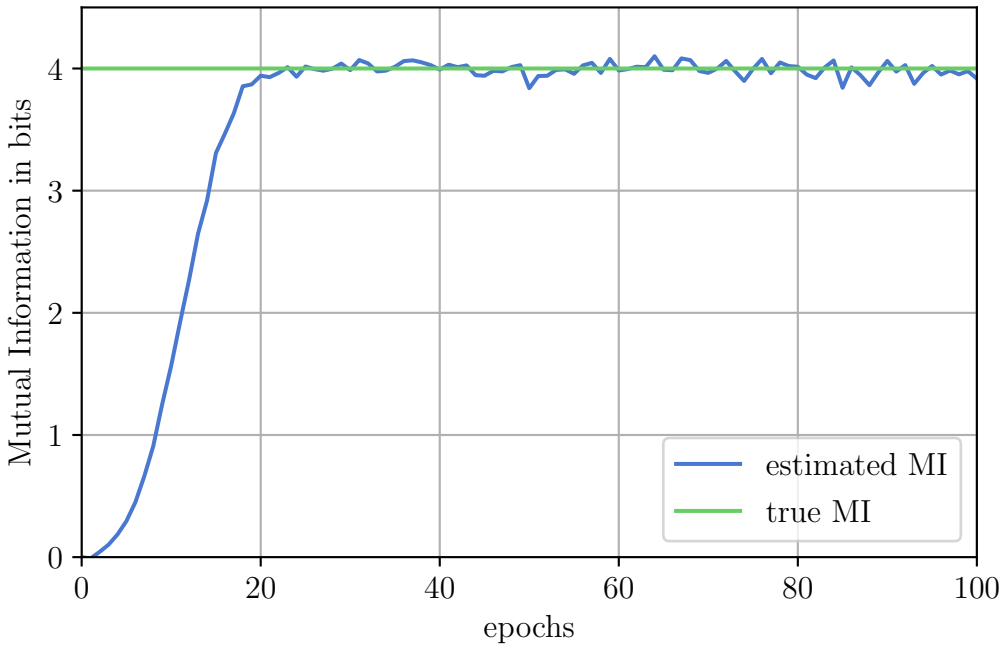
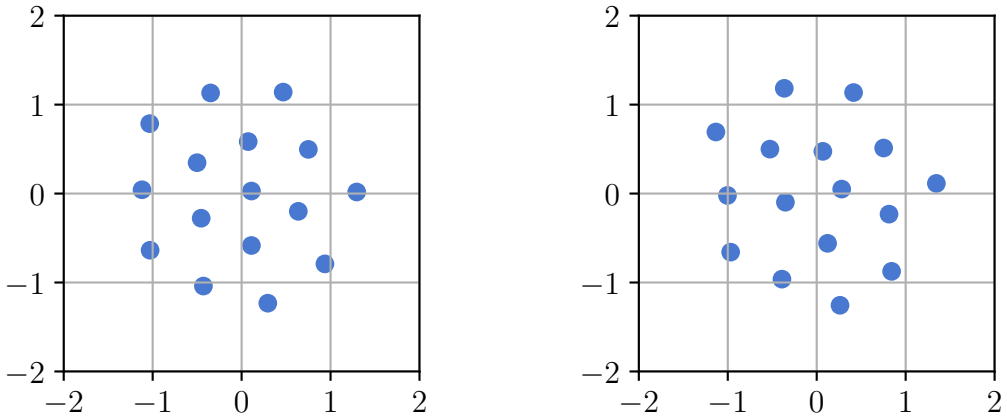


Figure 4·10: MINE estimation of $I(X, Y)$ for 16QAM at $SNR = 20\,\mathrm{dB}$ using MINE

The following plots show the results for using a MINE estimation as loss function and to train the decoder separately.



(a) constellation using MINE estimation as loss function at $SNR = 12\,\mathrm{dB}$

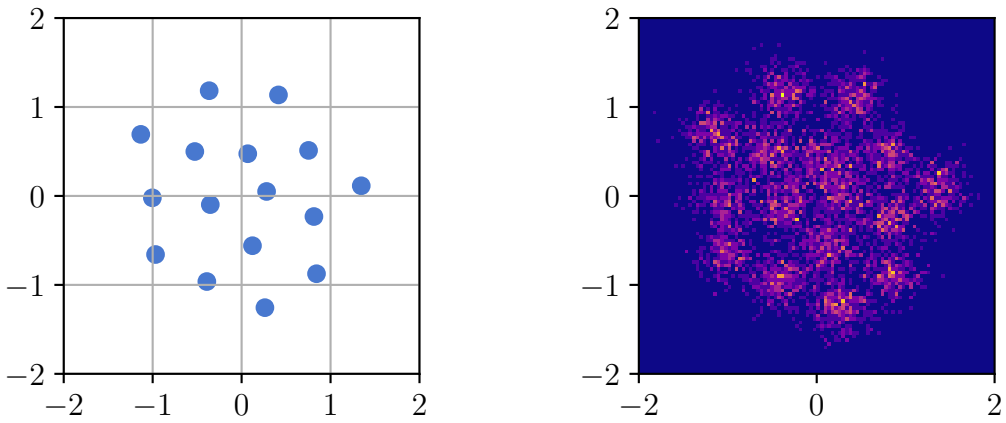
(b) constellation using MINE estimation to train encoder separately at $SNR = 12\,\mathrm{dB}$

Figure 4·11: Learned constellations for $M = 16$ with different SNR values on training

We can see that using MINE as loss functions seems to work fine since we obtain a regular constellation that is similar to the constellations in 4.2.1. Whereas, the separated training did not perform that good. This might be caused by non-optimal training parameters that can be improved with further testing.

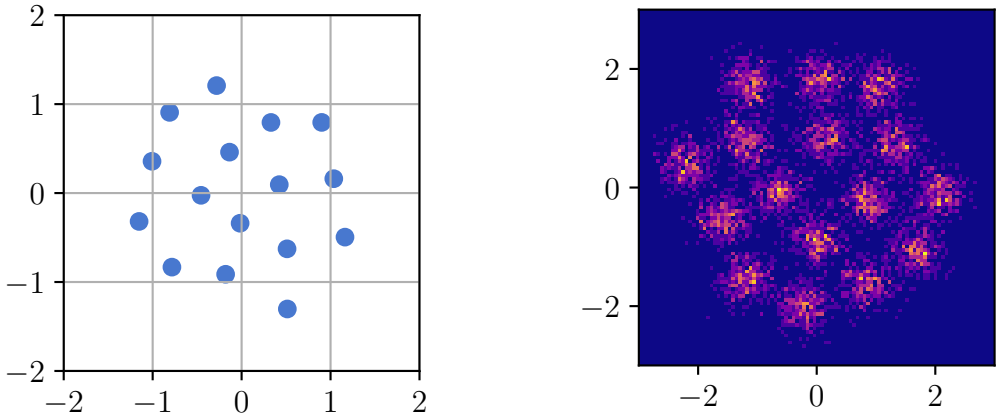
4.4 Implementation Generative Adversarial Network

In this section, we present the results of a GAN combined with an autoencoder based on the description in Figure 3·2. First, we pretrain GAN and autoencoder separately on the channel. Afterwards we put them together to train combined. A general implementation of a GAN that learns the channel distribution without an autoencoder can be found in the code as well.



(a) constellation **(b)** channel estimation

Figure 4·12: GAN and autoencoder combine training on AWGN at $SNR = 12\,\mathrm{dB}$



(a) constellation **(b)** channel estimation

Figure 4·13: GAN and autoencoder combine training on transmission with HPA at $SNR = 10\,\mathrm{dB}$ and input-backoff of $0\,\mathrm{dB}$

Conclusion

At the beginning of this thesis, we motivated a lot of applications in communications that machine learning can be applied to. However, we picked only a small part to be the topic of this thesis.

We used PyTorch as a rather untypical environment for communication problems to implement all kinds of autoencoders. Nevertheless, the quality and reusability of the code benefits from our choice. We started with simple channel models as AWGN and moved on to a more complex one including non-linearities. The results showed great performance that can be compared with expert-designed procedures. The next step was to detach an explicit channel model from the autoencoder to provide a more adaptable model that can be adjusted online and keeps its level of performance. For this approach we implemented further machine learning models to keep an end-to-end optimization ensured. As a result we hope that the existing models we implemented will be applied to real world data and prove their performance in the future.

Furthermore, future work can include the improvement of the existing autoencoder models. Great work has been published recently by (Cammerer et al., 2020) who proposes a bit-wise autoencoder instead of a symbol-wise to introduce coded modulation as well. (Dörner et al., 2020) has introduced a promising modification to the GAN and (Stark et al., 2019) proposed an autoencoder architecture that performs probabilistic and geometric shaping as well.

References

Ballard, D. H. (1987). Modular learning in neural networks. In *AAAI*, pages 279–284.

Belghazi, M. I., Baratin, A., Rajeswar, S., Ozair, S., Bengio, Y., Courville, A., and Hjelm, R. D. (2018). Mine: mutual information neural estimation. *arXiv preprint arXiv:1801.04062*.

Bishop, C. M. (2006). *Pattern recognition and machine learning*. springer.

Cammerer, S., Aoudia, F. A., Dörner, S., Stark, M., Hoydis, J., and Ten Brink, S. (2020). Trainable communication systems: Concepts and prototype. *IEEE Transactions on Communications*.

Cover, T. M. (1999). *Elements of information theory*. John Wiley & Sons.

Dörner, S., Cammerer, S., Hoydis, J., and Ten Brink, S. (2017). Deep learning based communication over the air. *IEEE Journal of Selected Topics in Signal Processing*, 12(1):132–143.

Dörner, S., Henninger, M., Cammerer, S., and Brink, S. t. (2020). Wgan-based autoencoder training over-the-air. *arXiv preprint arXiv:2003.02744*.

Fritschek, R., Schaefer, R. F., and Wunder, G. (2019). Deep learning for channel coding via neural mutual information estimation. In *2019 IEEE 20th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, pages 1–5. IEEE.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680.

Heinrich, F. (2020 (accessed September 11, 2020)). *Fabian Heinrichs GITHUB*.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

MathWorks (2020 (accessed September 11, 2020)). *BERAWGN MATLAB DOCUMENTATION*.

McEliece, R. J. (2004). *The theory of information and coding*. Number 86. Cambridge University Press.

O'Shea, T. J., Karra, K., and Clancy, T. C. (2016). Learning to communicate: Channel auto-encoders, domain specific regularizers, and attention. In *2016 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, pages 223–228. IEEE.

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

O'Shea, T. and Hoydis, J. (2017). An introduction to deep learning for the physical layer. *IEEE Transactions on Cognitive Communications and Networking*, 3(4):563–575.

O'Shea, T. J., Roy, T., and West, N. (2019). Approximating the void: Learning stochastic channel models from observation with variational generative adversarial networks. In *2019 International Conference on Computing, Networking and Communications (ICNC)*, pages 681–686. IEEE.

PyTorch (2020 (accessed September 11, 2020)a). *PyTorch Activation Functions*.

PyTorch (2020 (accessed September 11, 2020)b). *PyTorch Loss Functions*.

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.

Stark, M., Aoudia, F. A., and Hoydis, J. (2019). Joint learning of geometric and probabilistic constellation shaping. In *2019 IEEE Globecom Workshops (GC Wkshps)*, pages 1–6. IEEE.

You, Z., Ye, J., Li, K., Xu, Z., and Wang, P. (2019). Adversarial noise layer: Regularize neural network by adding noise. In *2019 IEEE International Conference on Image Processing (ICIP)*, pages 909–913. IEEE.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, die Zitate ordnungsgemäß gekennzeichnet habe und keine anderen als die im Literaturverzeichnis angegebenen Quellen und Hilfsmittel benutzt wurden. Ferner habe ich vom Merkblatt über die Verwendung von Bachelor- und Abschlussarbeiten Kenntnis genommen und räume das einfache Nutzungsrecht an meiner Bachelor-Arbeit der Universität der Bundeswehr München ein.

.....

(Unterschrift)