Technische Universität München

Invertible Neural Networks for MIMO Detection

Bachelor Thesis

Jonas Maas

Supervisor: Univ.-Prof. Dr.-Ing. Wolfgang Utschick

Submission: Xxx xx, 20XX

TUM Department of Electrical and Computer Engineering
Professur für Methoden der Signalverarbeitung
Univ.-Prof. Dr.-Ing. Wolfgang Utschick

Abstract

There is a lot of recent interest in *Invertible Neural Networks* for density estimation. The idea is to use the architecture of these networks for MIMO Detection. The advantage of this approach is that it does not require the use of complex methods at the receiver to reconstruct the signal. We simply run the received signal through the network in reversed direction. The network is based on *Normalizing Flows* to guarantee the invertability, where the scale and translation variables are learned with the help of *Convolutional Neural Networks* (CNNs).
We achieved very good results for low dimensionality but the system is too complex to expand it to high dimensionality at the moment.

Contents

1 Introduction

One of the main topics of communication technology is to reconstruct data at the receiver that was transmitted over a non-trivial channel.
Multiple-Input-Multiple-Output (MIMO) Detection describes the case where we have multiple transmitter signals, multiple antennas, multiple receiver signals, multiple receiver antennas, etc. The data is transferred over the channel between the transmitter and receiver. In order to reconstruct the original data at the receiver, it is necessary to invert the distortion and the noise, caused by the channel. Over the years many different methods were proposed and used. Recently a lot of researchers tried to use *Neural Networks* for this yet challenging problem. The standard way of using *Neural Networks* for this task is to map the received signals $\boldsymbol{y}$ to the transmitted signals $\boldsymbol{x}$. The idea of doing this, is to simplify the transmitter. It is only necessary to feed the trained network with the received signal $\boldsymbol{y}$ and it then computes the original signal $\boldsymbol{x}$. We tried a different approach. The transmitted signals $\boldsymbol{x}$ are mapped to the received signals $\boldsymbol{y}$ in the natural way and then the origial signals are reconstructed by inverting this mapping in the online phase. The mapping is done with the help of invertible and composable transformations. Afterwards a ML Detector is simply used on each stream and so the NP-hardness is avoided in the real-time system.
Therefore a network is being used, which is structured into sequential and invertible blocks, where each block is a transformation and has its own *Convolutinal Neural Network*. The internal Networks are used to determine the weights for the affine, autoregressive transformations.

2 Background

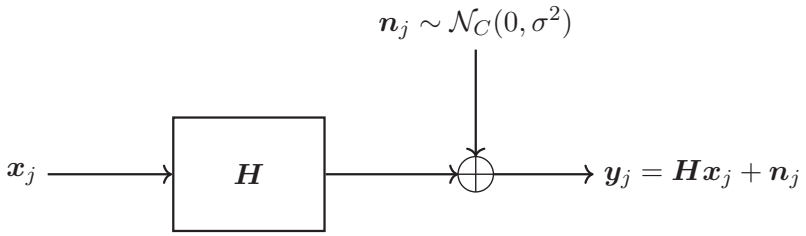
First of all it is essential to outline basic properties of the network and to define the underlying MIMO system.

2.1 Channel setup

The MIMO System consists of n transmitter and n receiver antennas. Each antenna sends a *QPSK* signal, thus we have 4^n different signals $\boldsymbol{x}$.

$$\boldsymbol{x} \in \{\pm\frac{1}{\sqrt{2}} \pm \frac{j}{\sqrt{2}}\}^n \tag{2.1}$$

These signals are transmitted over an *Additive White Gaussian Noise* (AWGN) channel. In this model the signal $\boldsymbol{x} \in \mathbb{C}^n$ is multiplied with the channel matrix $\boldsymbol{H} \in \mathbb{C}^{n\times n}$ and then the noise $\boldsymbol{n} \in \mathbb{C}^n$ is added. The resulting receiver signal is $\boldsymbol{y} \in \mathbb{C}^n$.



Without loosing generality, the channel is assumed to only consist of line-of-sight paths and the structure of the transmitter and receiver to be the same. Then there are direct paths from each transmitter antenna to the associated receiver antenna and some distorted paths from one receiver antenna to the other ones. These paths are added together and result in the following structure

$$\boldsymbol{H} = \sum_{k=1}^{K} h_k \boldsymbol{a}(\theta_k)\boldsymbol{a}(\theta_k)^H, \qquad \boldsymbol{a}(\theta_k) = [\alpha_k^0 \quad \alpha_k^1 \quad ... \quad \alpha_k^{n-1}]^T, \qquad \alpha_k = e^{-j\pi \sin\theta_k}. \tag{2.2}$$

Where θ_k and h_k are channel specific parameters and are responsible for the magnitude and the shift at the receiver.
When having a closer look at one of the receiver antennas, which is represented by one row in the $\boldsymbol{y}$ vector, the composition of the received signal becomes very clear.

$$y_{j,i} = h_{i+1,1}x_{j,0} + h_{i+1,2}x_{j,1} + ... + h_{i+1,i+1}x_{j,i} + ... + h_{i+1,n}x_{j,n-1} + n_{j,i} \quad (2.3)$$

There is one summand containing the signal of the corresponding transmitter antenna and the others summands are the damped signals of the other transmitter antennas or noise. We are interested in the, in general, dominating summand of the corresponding transitter antenna. The challenge is to invert the interference and predict the original signal.

2.2 Transformations

2.2.1 Inversion of Neural Networks

The approach of detecting signals by learning the channel and inverting it afterwards requires an *Invertible Neural Network*. Indeed every *Feed-Forward Neural Network* with any bijective activation function is invertible, if the weight matrix is not singular.

$$\begin{aligned} \boldsymbol{y} = f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) \qquad &\Leftrightarrow \qquad \boldsymbol{x} = \boldsymbol{W}^{-1}(f^{-1}(\boldsymbol{y}) - \boldsymbol{b}) \\ \text{with} \qquad &\boldsymbol{W} \in \mathbb{C}^{n\times n}, \boldsymbol{b} \in \mathbb{C}^n. \end{aligned} \quad (2.4)$$

In this equation $\boldsymbol{W}$ is the weight-matrix, $\boldsymbol{b}$ is the bias vector, $\boldsymbol{x}$ is the input signal and f is the activation function.
The problem is the expense and the numerical instability of the inversion. The larger the network, the smaller the elements of the weight matrix, which causes numerical instability. The requirements for the network, we want to build ,are contrary to these problems. The goal is to avoid complexity and errors, caused by the inversion, in the online phase.
The idea is to use an *Invertible Neural Network* based on simple and composable transformations. Every transformation has to be bijectice with a simple inversion but still has to be capable of learning a highly nonlinear mapping.

$$\boldsymbol{z} = T(\boldsymbol{u}) \qquad \Leftrightarrow \qquad \boldsymbol{u} = T^{-1}(\boldsymbol{z}), \qquad \text{with} \qquad \boldsymbol{z}, \boldsymbol{u} \in \mathbb{C}^n. \quad (2.5)$$

Such transformations are called *diffeomorphisms*. They are invertible, composable, differentiable and are often used in density estimation tasks. There they have to be differentiable to compute *Jacobian matrices*. For our problem it is only assumed that

they are invertible and composable.
Given two of these transformations T_1 and T_2, their composition $T_2 \circ T_1$ is also invertible

$$(T_2 \circ T_1)^{-1} = T_1^{-1} \circ T_2^{-1}. \tag{2.6}$$

Consequently multiple instances of simple tranformations can be composed without loosing invertability. This allows us to construct complex functions out of very simple ones. Each function T_k transforms $\boldsymbol{z}_{k-1}$ into $\boldsymbol{z}_k$, assuming $\boldsymbol{z}_0$ is the input and $\boldsymbol{z}_K$ is the target.

$$T = T_K \circ ... \circ T_1 \qquad \text{with} \qquad \boldsymbol{z}_k = T_k(\boldsymbol{z}_{k-1}) \tag{2.7}$$

Regarding the problem of learning the channel, these transformations allow us to use simple components as a base and combine them. The result is a system that is able to represent a complex mapping without loosing the ability of a simple inversion.

2.2.2 Affine transformation

The most simple transformations are affine transformations.

$$T_i(z_i, \boldsymbol{h}_i) = \alpha_i z_i + \beta_i \qquad \boldsymbol{h}_i = \{\alpha_i, \beta_i\}. \tag{2.8}$$

Equation 2.8 descibes a scaling and a location transformation. The α_i controls the scale of the input and β_i determines the location. To guarantee invertability α_i must not be zero ($\alpha_i \neq 0$). This can be achieved by defining α_i the following way: $\alpha_i = \exp(\tilde{\alpha}_i)$. Since $\exp(\cdot)$ can not become zero the invertability of the transformation is guaranteed. There are a lot of ways to implement the *conditioner* $\boldsymbol{h}_i$. For this task each parameter $\tilde{\alpha}_i$ and β_i is learned by a seperate *Neural Network*.
This causes linear growth in complexity with an increasing number of composed transformations. In fact we should be aware of using as little as possible of these transformations.

2.2.3 Related Work

The underlying concept defined above relates to the work in distribution estimation. Especially affine transformations were used a lot in popular papers on this subject, like *non-linear independent components estimation (NICE)*, *real non-volume preserving flow (Real NVP)*, *masked autoregressive flow (MAF)* and *Glow* [1, 2, 3, 4]. As shown in these papers the networks constructed are based on these transformations and scaled very good. When comparing the underlying problem of all of these networks to the one coverred in this thesis, one finds a lot of similarities. So if you are interested in knowing more about the subject, it's highly recommended to read these papers. More information and much more complex approaches about the transformations itself are covered in *normalizing flows for probabilistic modeling and inference* [5].

3 Problem specification

3.1 Project goal

As initially described, there are 4^n different signals when using *QPSK* and n receiver and transmitter antennas. The signals get transferred over the *AWGN channel*, which leads to distorted and noisy signals at the receiver.
One of the main goals of signal processing is to invert the distortion and noise to get the best possible estimation of the original signal. The upper-bound of this operation can be defined the following way, assuming the use of a trivial channel, whose channel matrix is the identity matrix. In other words, there is no distortion, thus $\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{n}$. It is not possible to get any better, because every signal is separated from each other and there is no way of controlling the noise $\boldsymbol{n}$. In this case, the ML-detector can be obtained as

$$\hat{\boldsymbol{x}}_j = \underset{\boldsymbol{x}_m}{\operatorname{argmin}} \|\boldsymbol{x}_m - \boldsymbol{y}_j\| \qquad \text{with} \qquad \boldsymbol{y}_j = (\boldsymbol{x}_j + \boldsymbol{n}_j), \tag{3.1}$$

where $\boldsymbol{x}_j$ is the transmitted signal, $\boldsymbol{n}_j$ the noise added to the signal and $\boldsymbol{x}_m$ is one of the 4^n possible signals. The goal is to get as close as possible to the upper-bound.

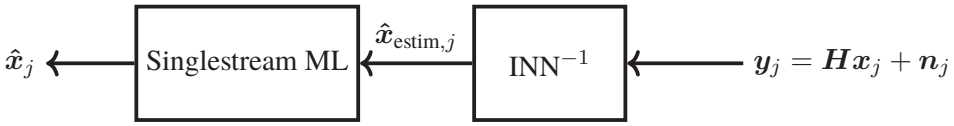
3.2 Approach

Recently a lot of work has gone into the idea of using *Neural Networks* for MIMO Detection.
The basic idea is to map the signal $\boldsymbol{y}$ to the originial signal $\boldsymbol{x}$. These networks replace the classical filter structures by trying to learn the inversion of the channel. When using *Neural Networks* for detection, the complexity of the receiver can be reduced by simply using the pre-trained network and not having to use complex filters to transform the data. The network gets trained offline with pairs of samples $(\boldsymbol{y}_j; \boldsymbol{x}_j)_{j=1}^{J}$ and then in the online phase the network sees samples of $\boldsymbol{y}$, it has never seen before and tries to compute the corresponding $\boldsymbol{x}$.
Lately *Invertible Neural Networks* have been a subject of high interest, especially in distribution estimation. They are typically used to transform a simple base distribution,

like the *Gaussian*, into a target distribution of which only samples are available. This is done by learning the mapping with the help of the networks to be able to invert it afterwards and get more samples of the target distribution.
In this paper it is tried to use the same principle for *MIMO-Detection*. Instead of mapping $\boldsymbol{y}$ to $\boldsymbol{x}$ we try to connect them in the natural way. The idea of mapping $\boldsymbol{x}$ to $\boldsymbol{y}$ is to avoid the normally much more complex inverse mapping. Indeed it is not avoided completely, because in the actual detection phase the inverse is used to reconstruct the original signal. However this is done by inverting the system built based on easy transformations and these transformations are very simple to invert. A comparison with the *Invertible Neural Networks for density estimation* at hand points out a lot of similarities. We also want to learn a mapping of input data $\boldsymbol{x}$ to the target data $\boldsymbol{y}$ with only a limited amount of samples provided. Furthermore the inverse transformation is applied to the received data $\boldsymbol{y}$ to get back the transmitted data $\boldsymbol{x}$.

When now trying to learn the mapping between $\boldsymbol{x}$ and $\boldsymbol{y}$ the following problem is encountered. The signal $\boldsymbol{x}$ is part of a discrete set and the signal $\boldsymbol{y}$ is a continuous variable. A bijective function between those two variables is therefore impossible. Considering this, the idea is to map $\boldsymbol{x} + \boldsymbol{n}$ to $\boldsymbol{y}$ with the help of *affine diffeomorphisms*. Then the input and target data are continous and the problem is solved at the expense of complexity. The proposed mapping is highly non-linear and there is no formal proof that the transformations used are universal approximators, like discussed in [5]. However, in this case simulations took proof of the capability.

Another advantage is that the only information we need to know about the channel is a finite amount of samples. There is no need to know the channel matrix itself or what type of channel it is used. The architecture of the network is independent of the architecture of the channel.



In conclusion the approach can be described as follows. The idea is to train an *Invertible Neural Network* to be able to map $\boldsymbol{x} + \boldsymbol{n}$ to $\boldsymbol{y}$. For this purpose bijective

transformations are used to afterwards being able to invert the mapping. When detecting, we receive the signal $\boldsymbol{y}_j$ and want to get an estimation for the original signal $\boldsymbol{x}_j$. Therefore the signal $\boldsymbol{y}_j$ is used as input for the system, which outputs $\hat{\boldsymbol{x}}_{\text{estim},j}$ by inverting the transformations. $\hat{\boldsymbol{x}}_{\text{estim},j}$ gets further processed to receive $\hat{\boldsymbol{x}}$, which is one of the 4^n signals again. This is done by a *Singlestream-ML*, which is a less complex version of a standard *ML-Detector*. Its use is another benefit of the proposed network. The exact working routine and the advantage of this type of ML is discussed in chapter 5.

3.3 Setup for the learning phase

In the learning phase, we use sets of finite amounts of samples of our actual system to learn the mapping. The learning phase is split into 3 parts, the training phase, the validation phase and the test phase. Every phase uses a different and disjunct set of samples.

$$\begin{aligned}\{(\boldsymbol{x}_{j,tr} + \boldsymbol{n}_{j,tr}, \boldsymbol{y}_{j,tr})\}_{j=1}^{N} \cap \ \{(\boldsymbol{x}_{j,val} + \boldsymbol{n}_{j,val}, \boldsymbol{y}_{j,val})\}_{j=1}^{M} \\ \cap \{(\boldsymbol{x}_{j,test} + \boldsymbol{n}_{j,test}, \boldsymbol{y}_{j,test})\}_{j=1}^{L} = \varnothing.\end{aligned} \tag{3.2}$$

Training samples are used to determine the weights and biases of the network. Validation samples are used to compare the performance and avoid overfitting in the training phase. Test samples are used to benchmark the overall performace of the trained network in comparison to other networks. A commonly used segmentation is 60-20-20 in favour of the training samples. Our learning process does not contain the actual test of the inversion while training. It is all about fitting the input data to the target data and trying to find a balance between performance and overfitting in order to get a function as smooth as possible without large deviations.

Invertible Neural Network for MIMO Detection

4

4.1 Main structure of the Invertible Neural Network

To sum things up till this point, the concept of the forward path can be described as follows. We try to learn a highly nonlinear mapping by combining affine transformations, whose parameters are learned with the help of *Neural Networks*. In the beginning it was pointed out, that these very simple transformations need to be coupled with each other to be capable of representing such complex mappings.

To do so the system is structured into K sequential blocks, where each block contains an affine transformation and the output of one block is the input of the next block. By doing this the overall transformation is obtained like described above, $T_{total} = T_K \circ ... \circ T_1$. Inside the blocks, the set of parameters $\{\alpha_i, \beta_i\}$ for the transformation needs to be determined. This is where the actual intelligence of the system comes to play. The parameters are determined by a *Deep Neural Network*, where each block and each paramter has its own network. The reason for doing this is to avoid correlations of the individual parameters between each other. On the other hand there has to be a solution for the quick increase of complexity. While training, each block that is added, consequents in more networks and therefore $m + o$ more parameters to be determined. o and m represent the complexity of the networks to compute α and β. Thus it appears that there is a tradeoff between complexity and accuracy, like in almost every technical application. Another problem is that an increasing number of blocks causes error propagation. The more sequential blocks, the more room for error and chance of propagating it through the entire system. There are plenty of options where the errors come from. In our system the most problems are caused by numerical instability. When increasing the number of blocks, one consequence is that inevitably the weights

of the network shrink.

$$\begin{aligned} \boldsymbol{y} = T(\boldsymbol{x}+\boldsymbol{n}) = \boldsymbol{\alpha}_K \odot \boldsymbol{x}_{K-1} + \boldsymbol{\beta}_K &= \boldsymbol{\alpha}_K \odot (\boldsymbol{\alpha}_{K-1} \odot \boldsymbol{x}_{K-2} + \boldsymbol{\beta}_{K-1}) + \boldsymbol{\beta}_K \\ &= \boldsymbol{\alpha}_K \odot \boldsymbol{\alpha}_{K-1} \odot \boldsymbol{x}_{K-2} + \boldsymbol{\alpha}_K \odot \boldsymbol{\beta}_{K-1} + \boldsymbol{\beta}_K \\ &= ..., \end{aligned} \tag{4.1}$$

where $\odot$ is the *Hadamard product* or *element-wise product*. With shrinking weights, the fixed data type can become a problem, because the accuracy may not be sufficient enough. Another issue is the correlation of the weights. When changing one weight the whole transformation is changed which in fact requires a lot of fine tuning and optimization. Again the balance between the achievable complexity of the network and the error propagation has to be found.
Inside each block there are two paths, the *forward* and the *backward* path. The forward path is used in the actual training phase to determine the parameters of the network. This path represents the natural way of mapping $\boldsymbol{x} + \boldsymbol{n}$ to $\boldsymbol{y}$ and is therefore the actual forward transformation. The backward path on the other hand is used in the online-phase to invert the received signals. It transforms the received signal $\boldsymbol{y}$ in an estimation for the original signal $\boldsymbol{x}$ and represents the inverse transformation. Both paths work the exact same way. The input data gets pre-processed in order to then serve as the input of the two *Neural Networks* to determine the transformation parameters $\{\alpha_i, \beta_i\}$. The parameters are used to transform the input data z_i into the output data z_{i+1} on the forward path and to transform the input data z_{i+1} into the output data z_i on the inverse path.

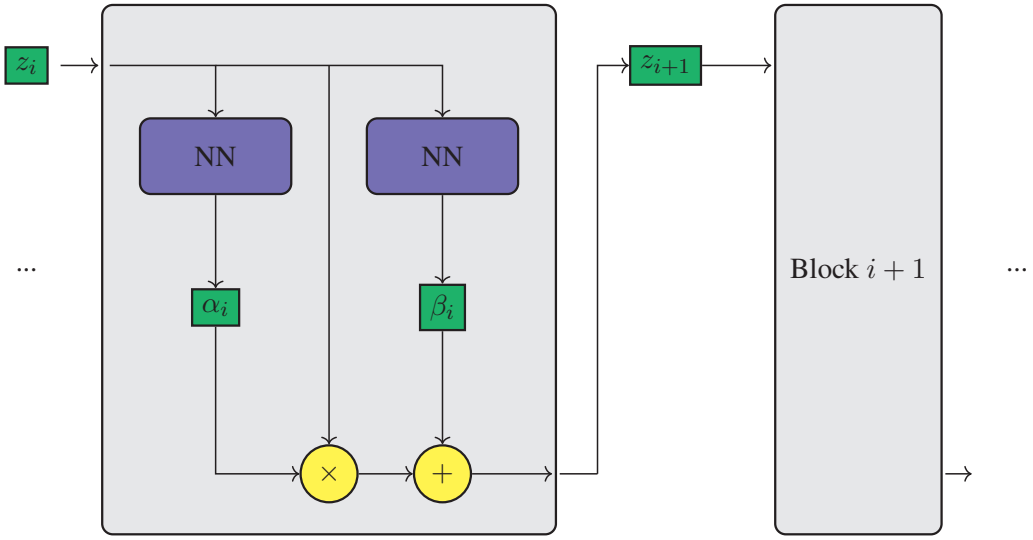


Figure 4.1: Structure of the system

Unlike most other *Neural Networks* it does not make a huge difference to compute its parameters on a GPU. The reason is that the network is not very parallel at all. The sequential structure requires sequential computing. In other words we have to handle one block after another, which is the opposite of a parallel structure thus it is not necessary to run the system on GPUs.

4.2 Coupling Layers

The following concept was first proposed in the paper [6]. Like most of the other concepts used, its origins are in density estimation and is modified to apply it on to signals.
Each of the sequential structured blocks explained in section 4.1 is a *Coupling Layer*. The extension compared to the way we looked at these blocks until now is the technique used to pre-process the input data. Instead of transforming all of the input data, now only parts of the input is transformed and the rest is being run through the block without modifying it.

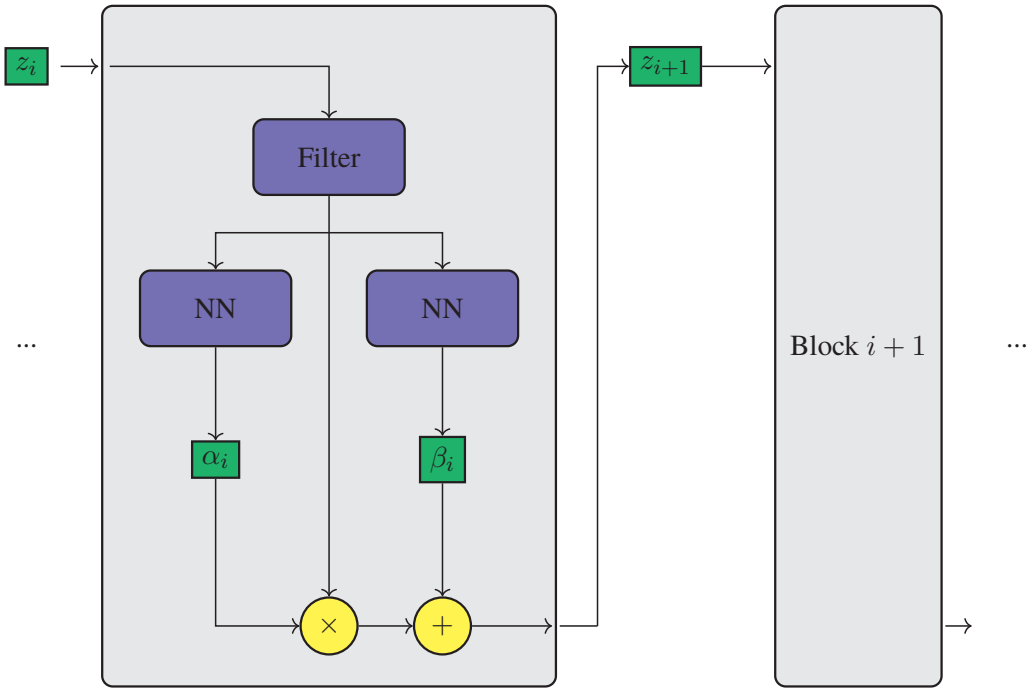


Figure 4.2: Structure of a *Coupling Layer*

To get this done, a binary mask $\boldsymbol{b} \in \{0,1\}^n$ is applied to the input data of each block. The mask gets generated randomly by distributing a special amount of ones across a zero-vector. The randomness is required to make sure every dimension of the input is transfomed at least once along its path through the system. In other words the unchanged dimensions are exchanged in every other block.
Another way of making sure to vary the transformed dimensions is to always transform the same dimensions inside one block but to apply *permutations* onto the data between several blocks.

When looking at the special case used for the simulations, it needs at least six layers to make sure every entry of the input vector gets processed together with every other entry in the input vector. The six sequential layers are the lower-bound. But to consider the randomness caused by the generation of the masks and the fact that the mapping is very complex it is necessary to add some more blocks to make sure every input is transformed more than once.

The use of this type of pre-processing routine does not affect the invertability. The

forward and the backward paths are still very similar, which is good because we want our backward path to be computable as fast as possible.

$$\begin{aligned} \boldsymbol{y} &= \boldsymbol{b} \odot \boldsymbol{x} + (1 - \boldsymbol{b}) \odot (\boldsymbol{x} \odot \exp(s(\boldsymbol{b} \odot \boldsymbol{x})) + t(\boldsymbol{b} \odot \boldsymbol{x})) \\ \boldsymbol{x} &= \boldsymbol{b} \odot \boldsymbol{y} + (1 - \boldsymbol{b}) \odot (\boldsymbol{y} - t(\boldsymbol{b} \odot \boldsymbol{x}) \odot \exp(-s(\boldsymbol{b} \odot \boldsymbol{x}))) \end{aligned} \tag{4.2}$$

$s(\cdot)$ and $t(\cdot)$ stand for the scale and translation. Note that computing the inverse path does not require the invertability of the functions $s(\cdot)$ and $t(\cdot)$. This means the whole non-linearity can be realized with the help of these two variables. In this approach both of these variables are computed with the help of *Convolutional Neural Networks*, which are highly nonlinear.
Like explained above, each *Coupling Layer* transforms a part of the input data and all of the layers are combined together to learn the channel.

4.3 Internal Neural Networks for parameter determination

Every block in the sequential structure contains two *Neural Networks* to determine the parameters for the transformation. The networks are the most important part of the system and are responsible for good results.

The problem can be addressed in two different ways.
First of all it can been seen as a regression task, which means the task is to map $\boldsymbol{x} + \boldsymbol{n}$ to $\boldsymbol{y}$ as close as possible. For regression problems the most popular and most common networks are *Feed-Forward Networks*. These networks contain alternately a linear transformation followed by a non-linear activation function and was the first approach. The results were not very good, because it seemed like the approach was not complex enough.
The other way to look at the problem is to see it as a classification problem. When doing classification, normally the goal is to find clusters in the input data and map them to the pre-defined labels. From this perspective our problem could be seen as mapping 4^n clusters to 4^n distorted clusters, which represent the labels. n denotes the number of antennas.
Classification uses completely different networks but also completely different loss-functions in the training phase. These loss-functions are based on determination of the likelihood of the input data belonging to each of the pre-defined labels. Such loss-functions can not be used, because the network is trained to then be able to run the received data through the inverse network in its natural form. That means the label of the received data is not known. If it was, the problem would be solved already. Instead, we want to be able to use the actual received data and run it through the network without any pre-processing routine.
This leads to a mixture of regression and classification. The network itself works on the base of a classification network but the loss function remains a regression one.
In image classification, which is obviously a classical classification task, most networks are *CNNs*. These networks are known for their ability of being able to find structures in pictures. For our input data we use *1D-Convolutional Neural Networks* to detect clusters and, in consequence, determine the paramters for the transformations.

The structure of our network is shown in Figure 4.3. First of all a *Convolutional Layer* is applied to the pre-processed input data of each block. Like explained above the convolution tries to find a pattern in the input data. Next a non-linear activation function is applied to the data, followed by a *Max-Pooling Layer*. Like in every other

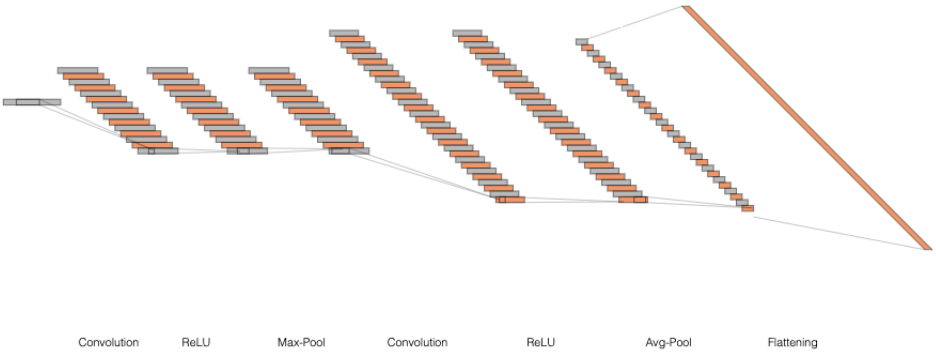


Figure 4.3: Architecture of the CNN for determination of the parameters $\{\alpha_i, \beta_i\}$

network the non-linearity of the activation function is the actual intelligence of the network and the *Max-Pool operation* is used to filter the important information and to restrict the complexity. The same structure is applied twice with the difference that instead of the *Max-Pool operation* an *Average-Pool operation* is applied the second time. Following this the network scaled even better. Afterwards the data gets flattened and further processed by a *Feed-Forward Neural Network* and brought in the right shape to calculate the transformation. In the network we are using *Dropout Layers*, to make the network more robust and to prevent the network from overfitting. This concept was proposed in [7] and is a commonly used technique in machine learning. An observation made while training the network was, that if more channels are used in the *CNN* than there are different signals, the system scaled very good. As you can see the system itself is very complex, which supports the initial assumption of a highly non-linear mapping.

5 Detection

The purpose of the whole system is to simplify the detection of the signal at the receiver. To do this, nearly every receiver uses a *ML-Detector*, which solves the following optimization problem

$$\hat{\boldsymbol{x}}_{ML} = \underset{\boldsymbol{x}}{\operatorname{argmax}} f_{\boldsymbol{X}}(\boldsymbol{y}; \boldsymbol{x}). \tag{5.1}$$

The optimizer searches for the signal $\boldsymbol{x}$, which maximizes the likelihood that this signal was sent when $\boldsymbol{y}$ is received. In the case where the channel is an *AWGN channel* like here, the equation simplifies to

$$\hat{\boldsymbol{x}}_{ML} = \underset{\boldsymbol{x}}{\operatorname{argmin}} \|\boldsymbol{y} - \boldsymbol{x}\|. \tag{5.2}$$

To solve this optimization problem every possible signal has to be tested and the one with the smallest norm is the one expected to be the best fit. When having *QPSK* signals and n antennas, 4^n signals have to be tested. Consequently the problem is *NP-hard*, which means the complexity of the optimization problem is exponentially growing. If the problem is high-dimensional the detection process gets very costly and will take a lot of time. It has to be accepted that this problem has this large complexity and it is not possible to solve a NP-hard problem exactly with lesser effort.

There are two ways of getting a quicker receiver. The first way is to approximate the solution and to not use the exact solution. The other way is to shift the *NP-hardness* away from the receiver. Then the receiver gets faster but on the expense of slowing down some other parts of the system.
In our approach the *NP-hardness* is shifted into the offline learning phase and, in addition, the solution is not exact. This is a consequence of using *Neural Networks* for approximating the solution. By doing this a relatively good solution is obtained and instead of handling the complexity at the receiver it is shifted into the offline training phase. To detect the signal, it is simply ran through the system in the inverse direction. The disadvantage of the approach is that we have to deal with the *NP-hardness* in the offline phase. In other words the more antennas and the more different signal-points

the longer it will take to train the network. The network gets larger because it has to get extended for recognizing more clusters than before. Even the size of the training set increases, when the amount of samples of each signal should maintain the same. In conclusion there is a bigger network with a bigger set of training data which causes an increase in training time.
The advantage, on the other hand, is that the detecting process is much more simple. By inverting the channel, there is no more interference between the single dimensions. Thus, the *NP-hard* problem is avoided at the detector and a Singlestream-ML can be applied. A *Singlestream-ML* is a *ML-Detector* which is applied to each dimension alone without taking account of the other dimensions.

$$\hat{\boldsymbol{x}}_{i,ML} = \underset{\boldsymbol{x}}{\operatorname{argmin}} \|\hat{\boldsymbol{x}}_{\text{estim},i} - \boldsymbol{x}_{QPSK}\| \qquad i \in \{i \in \mathbb{Z} \mid 0 \leq i < n\} \tag{5.3}$$

Where i is one of the n dimensions caused by the antennas and x_{QPSK} is one of the 4 signals transmitted on each stream. Contrary to the ML-Detection explained above the complexity now is only linear in n and not exponential. The complexity could also be seen as constant when considering that the detector can work on each dimension simultaneously. This can be done beacause there is no more interference between the dimensions.
In conclusion it can be said that the concept presented in this thesis, has a much faster detecting process with a constant complexity in contrast to the exponential one in standard applications. On the other hand the NP-hardness is shifted into the training phase and with more dimensions it takes much more time to learn the mapping.

6 Experiments

6.1 Data Generation

The signals we want to transmit are $QPSK$ signals, as described above. These signals are complex natured signals, which leads to the following problem. Most of the toolboxes used for dealing with *Neural Networks* are not able to handle complex signals. Therefore it is necessary to transform the signals into a real valued representation. In [8] two transformations were proposed. Here, the first one with the following formula is used.

$$\tilde{\boldsymbol{x}} = \begin{bmatrix} \Re(\boldsymbol{x}) \\ \Im(\boldsymbol{x}) \end{bmatrix}, \qquad \grave{\boldsymbol{A}} = \grave{\Re}(\boldsymbol{A}) = \begin{bmatrix} \Re(\boldsymbol{A}) & -\Im(\boldsymbol{A}) \\ \Im(\boldsymbol{A}) & \Re(\boldsymbol{A}) \end{bmatrix} \tag{6.1}$$

In this equation vector $\boldsymbol{x}$ and matrix $\boldsymbol{A}$ are complex valued. $\tilde{\boldsymbol{x}}$ and $\grave{\boldsymbol{A}}$ are the real-valued equivalents to the complex valued vector and matrix.

The transformation doubles the dimensions of the inputs and thus the complexity increases. Indeed this is no surprise, because each complex valued signal carries two separate information. There is the real part and the imaginary part and both parts contain an information. In fact the transformation gives each information of the complex signal a single real-valued dimension.

In this case the original signals are two-dimensional which leads to four dimensional input signals for the network. With the approach of two dimensional signals and the use of $QPSK$ there are 16 different possible signals to transmit. In other words there is a classification task with 16 different targets or the network has to learn the mapping between 16 input and 16 target signals. The reason why we choose to only use such litte dimensionality is, that in the beginning it was not even clear if the approach with invertible neural networks will work. To our knowledge nobody has ever tried such an approach and so the demand of this work was to show the possibilities of this approach. In consequence we decided to use very low noise with zero mean and variance of $\frac{1}{5}$. With such little noise the noisy signals are spread around the original data points, without interference between the data points. That means, if applying the ML-Detector on the noisy points, it achieves an accuracy of a hundred percent. The upper bound with a hundred percent is very hard to achieve.

The other variables we have to determine are the channel variables. There are the stream-weights h_k and the angles θ_k. We use equal weights which means that each stream has the same power and the same damping. This does not restrict the generality. The angles are also freely selectable. But if you are using *supplementary* angles the matrix becomes singular. *Supplementary* angles mean that the angles sum up to 180 degrees.
Data generation is done in *Matlab* and then the data is converted into the shape the *python* network requests it to be. Like described above there are four different data sets for training, validation, testing and for the inverse path. All of these data sets use the same channel and signal parameters but contain different instances.
For better scaling of the network the data gets *normalized*. This kind of pre-processing routine shifts the whole data into the interval $[-1; 1]$. Here *normalizing* is done by searching for the maxima of $\boldsymbol{x}_i + \boldsymbol{n}_i$ and then dividing the whole data by the maxima. We are doing this for each dataset on its own. The used approach is the most trivial one and can not handle huge additive noise. Assuming there is a huge noise value $\boldsymbol{n}_i$ added to the corresponding signal value $\boldsymbol{x}_i$ in the training data and the other noise values are comparatively little. Then the whole pre-processing routine in this dataset is dominated by the bigger one. This causes irregular distribution of the noisy datapoints over the interval $[-1; 1]$. In the other datasets the distribution is normal, which means the datapoints are spread over the interval regular. Then the network learns some strange mapping and when applying the network on the other data sets it scales poorly. So we have to pay attention to big noise values and in general to noise with big variance when using this approach.

The whole pre-processing routine is done to avoid inputs with large magnitude. If there are inputs with small and big magnitude, the ones with big magnitude have more impact on the weight updates than the other ones. By normalizing them, all inputs have nearly the same impact on the weight updates.

$$\triangle w_i \sim x_i \tag{6.2}$$

The 60-20-20 rule is used to determine how many samples of each signal there are in each data set. These samples are divided into batches, where every batch contains 20 samples. The samples are distributed randomly over the batches, to, in the best case, have one sample of each signal in every batch. We tried to distribute the samples in different orders over the batches but the random distribution worked best. The batches are then used to train the network.

6.2 Network

The structure of the system used for detection is described above. We are using sequential *Coupling Layers* with affine transformations to learn the mapping between $\boldsymbol{x} + \boldsymbol{n}$ and $\boldsymbol{y}$. The parameters of the transformations are determined by internal *Convolutional Neural Networks* in the corresponding block.

There are a lot of *hyperparameters* in the system and in this chapter the effect of the most important ones is explained.
First of all the network has to be initialized. Orthogonal initialization is used. It is popular for the improvement in performance and for a better abilitiy to converge. The effect of orthogonal initialization was profen in *Provable Benefit of Orthogonal Initialization in Optimizing Deep Linear Networks* [9].

One of the main *hyperprameters* is the number of sequential blocks. The number of blocks represents the number of combined transformations and thereby the complexity of the mapping. We are using 15 or more blocks to be able to get a system that is complex enough to represent the mapping. With an increasing number of blocks the complexity and consequently the amount of time to learn the networks increases rapidly. At about 30 sequential blocks the positive effect of being able to represent a more complex mapping and the negative effect of propagating errors through the network compensate each other. The best results were achieved when using 15 blocks.

The next parameter worth a more intense study is the amount of hidden units. This number comes to play when talking about the complexity of the *Feed-Forward-Network* where the flattened outcome of the *Convolutional Neural Network* gets passed to. With the number of hidden units the complexity can be further determined. The disadvantage in increasing the amount of hidden units is the chance to overfit the network. The network looses generality and tries to learn features which are not even there. The best results were achieved with 500 to 700 hidden units.

The architecture of the internal *Convolutional Neural Networks*, is very important for the performance of the system. The *CNNs* try to find features in the input data and determine the variables of the transformations to recognize them. To get good results, a good choice of its parameters is necessary. There are a lot of internal parameters but we highlight only a few of them.
The number of channels made a huge difference in performance. Channels are the

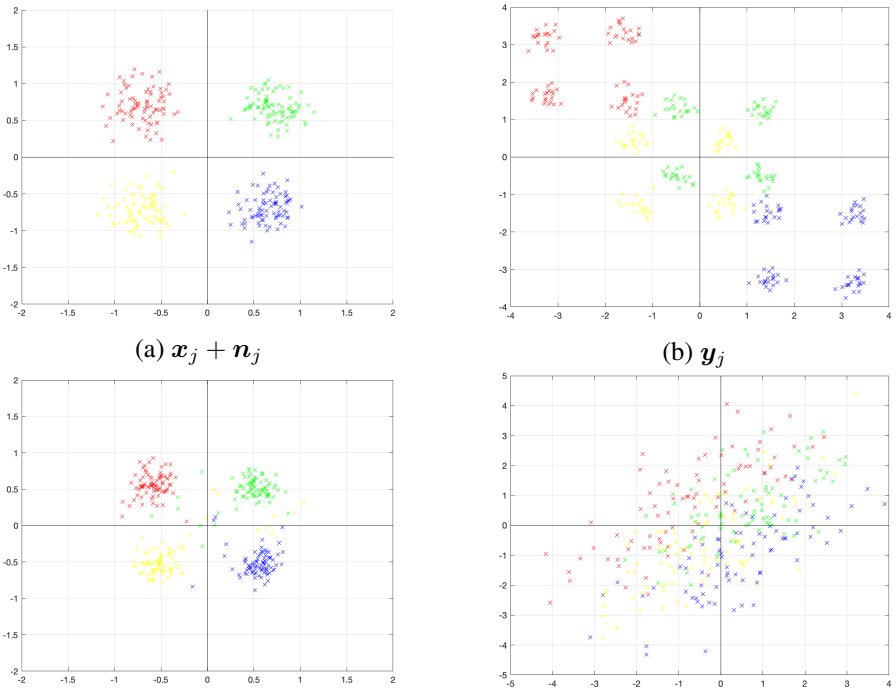
result of the convolution operations and represent different features of the input data. This means the more channels the network has the more features it can probably learn. Like said before, an increase in channels makes a huge improvement in the accurancy of the outcome. Especially more channels than different signals scaled very good.
Filter size is also worth a mention. It determines the number of input dimensions processed together. The network then tries to find features in those parts of the data. Our approach was to look at as much dimensions as possible at the same time, because each input dimension influences every other target dimension.
Other than that there are a lot of other parameters but we have not tried a lot of different configurations regarding them.

A commonly discussed topic when talking about *Neural Networks* is the activation function. It is the actual intelligence of the network and there are a lot of options to choose from. In most cases the *ReLU*, the *sigmoid* or the *tanh* function is used. The *ReLU* is the most common one for this kind of problem and it also worked best here. Regarding the performance, the difference between the functions were not large but the *ReLU* was the most consistent one.

The network used, achieved an accurancy of arround 96 percent in the best case. Due to the statistical component of the network it does not perform the same way on every simulation but varies a bit in performance. However the used *hyperparameters* allowed us to constantly achieve around 95 percent. In these simulations the system consisted of 15 sequential blocks, 500 hidden layers, *ReLU* activation function, 60 channels for the first convolutional operation and 120 channels for the second one.

In figure 6.1 the real-part of both transmitted signals is plotted against each other. The first subplot 6.1a shows the input of the system $\boldsymbol{x}_j + \boldsymbol{n}_j$. The four different clusters, caused by the four different real parts, can clearly be seen. The noise is *Gaussian* with very small variance. When taking account of the imaginary-parts too, there are 16 input clusters which have to be mapped to the 16 target clusters. They can be seen in 6.1b. The yet challenging problem is the adjacency of the target clusters. Some of them are very close to each other which causes a lot of trouble for the network. Especially with a high variance it gets hard for the network to differentiate between the different clusters.

In figure 6.1c the actual performance of the network can be seen. The signals plotted are the estimations of the network. Therefore the trained network got the received signal $\boldsymbol{y}_j$ as an input and the backward path was used to compute an prediction for



(a) $\boldsymbol{x}_j + \boldsymbol{n}_j$ (b) $\boldsymbol{y}_j$

(c) $\boldsymbol{x}_{\text{estim},j}$ (d) $\boldsymbol{x}_{\text{linear},j} = \boldsymbol{H}^{-1}\boldsymbol{y}_j$

Figure 6.1: Real part of the signals

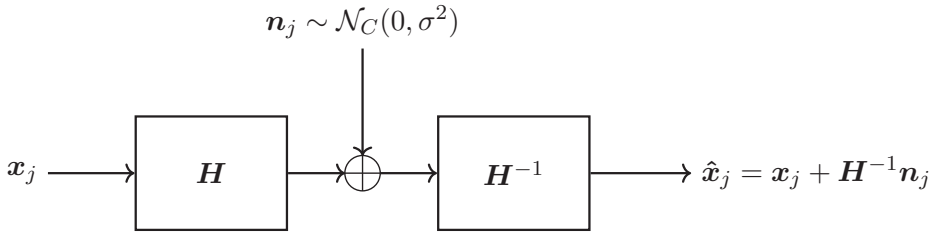
the transmitted signal $\boldsymbol{x}_j$. After applying a *Singlestream-ML* to the signal $\boldsymbol{x}_{\text{estim},j}$ the estimated transmitter signal $\hat{\boldsymbol{x}}_j$ is obtained. The plot shows, that the performance of the system is actually very good. Most of the data points got reconstructed very well. The clusters of the estimated signals are even not as spread as the original ones, which means that the network was able to supress parts of the noise. You can also identify a few outliers, especially caused by the yellow and the green signal points. These are the consequence of the mentioned adjacency of these clusters, see subplot 6.1b.

Like one would imagine, the *channel-matrix* has a huge impact on the distribution of the target clusters. The matrix can be changed by varying the angles θ_k or the weights h_k. With some parameters the clusters are very separated from each other and with others they are very close. A special case is when the *channel-matrix* gets singular. The network presented in this chapter was tested with a lot of different combinations and even performed very good on inverting the singular matrix.

6.3 Discussion

In this section we want to compare our results to results of other approaches and then discuss them afterwards.

At first we will have a closer look to the most simple, the linear approach.



When doing this, the received signal $\boldsymbol{y}_j$ gets multiplied by the inverse *channel-matrix* $\boldsymbol{H}^{-1}$ to receive the estimation $\hat{\boldsymbol{x}}_j = \boldsymbol{H}^{-1}\boldsymbol{y}_j$. The result of this operation can be seen in figure 6.1d . As you can see the result was very poor. The detection accuracy was between 30 and 40 percent. In the figure you can see that this operation was not able to reconstruct the original clusters. The linear transformation was not complex enough to get good results. Another disadvantage in comparison to the proposed approach is the missing ability to handle singular or nearly singular *channel-matrices*. In this case the linear transformation simply does not work at all.

Like mentioned in the beginning the normal way to use *Neural-Networks* for *MIMO-Detection* would be to map the received signal $\boldsymbol{y}$ to $\boldsymbol{x}$. We want to compare the results of the *Invertible Neural Network* to the results of the standard approach. Therefore we used the structure of the best performing network. But instead of using both the forward and backward path, only the forward path was used for the training as well as the detection phase. In the training phase the network tried to learn the mapping between $\boldsymbol{y}$ and $\boldsymbol{x}$ and in the detection phase the network got samples it has never seen before and tried to apply the mapping. The results were very bad and very inconsistent in comparisson to the results of the *Invertible Neural Network*. It could be concluded that mapping $\boldsymbol{y}$ to $\boldsymbol{x}$ is much more complicated than mapping $\boldsymbol{x} + \boldsymbol{n}$ to $\boldsymbol{y}$. There is no formal proof for this assumption but the simulation results validated it, at least for the kind of system we are using.

In summary we have shown that *Invertible Neural Networks* are usable for *MIMO-Detection*. The proposed network achieved remarkable results and could be improved to achieve state of the art accurancy. However the system has only been tested on low dimensionality. We ran a few simulations with the extension of the system to four antennas. These simulations took proof of the rapidly increasing complexity and in consequence the learning phase was very time consuming. The results were not as good as the ones from the two dimensional system but the system was not optimated yet.
To be able to extend the network to higher dimensions without the need of infinite time for the learning phase, we have to find a better structure for the network. The structure should be less complex, to decrease the amount of time the network needs to be trained. On the other hand the structure has to be as capable, regarding the ability to represent a complex mapping, as the structure proposed here. Another way of improving the results is to find a better pre-processing routine without impairing the performance when normalizing. The exact problem is described in section 6.1.

In conclusion it can be said that the approach has a lot of good aspects and is promising. It still needs a lot of optimization to be able to compete against standard systems and it has to be tested if the system is usable for more dimensions.

Bibliography

[1] L. Dinh, J. Sohl-Dickstein, and S. Bengio, "Density estimation using real NVP," *CoRR*, vol. abs/1605.08803, 2016. [Online]. Available: http://arxiv.org/abs/1605.08803

[2] L. Dinh, D. Krueger, and Y. Bengio, "Nice: Non-linear independent components estimation," 2014.

[3] D. P. Kingma, T. Salimans, and M. Welling, "Improving variational inference with inverse autoregressive flow," *CoRR*, vol. abs/1606.04934, 2016. [Online]. Available: http://arxiv.org/abs/1606.04934

[4] G. Papamakarios, T. Pavlakou, and I. Murray, "Masked autoregressive flow for density estimation," in *Advances in Neural Information Processing Systems*, 2017, pp. 2338–2347.

[5] G. Papamakarios, E. Nalisnick, D. J. Rezende, S. Mohamed, and B. Lakshminarayanan, "Normalizing flows for probabilistic modeling and inference," 2019.

[6] L. Ardizzone, J. Kruse, S. Wirkert, D. Rahner, E. W. Pellegrini, R. S. Klessen, L. Maier-Hein, C. Rother, and U. Köthe, "Analyzing inverse problems with invertible neural networks," *arXiv preprint arXiv:1808.04730*, 2018.

[7] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint arXiv:1207.0580*, 2012.

[8] C. Hellings and W. Utschick, "Measuring impropriety in complex and real representations," *Signal Processing*, vol. 164, pp. 267–283, 2019.

[9] W. Hu, L. Xiao, and J. Pennington, "Provable benefit of orthogonal initialization in optimizing deep linear networks," *arXiv preprint arXiv:2001.05992*, 2020.