Master's thesis
Master's Programme in Data Science

Iterative deep learning receiver

Roope Niemi

May 12, 2022

Supervisor(s): Assoc. Professor Arto Klami
Dr. Mikko Honkala

Examiner(s): Assoc. Professor Arto Klami
Dr. Mikko Honkala

University of Helsinki
Faculty of Science

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Degree programme
Faculty of Science	Master's Programme in Data Science

Tekijä — Författare — Author

Roope Niemi

Työn nimi — Arbetets titel — Title

Iterative deep learning receiver

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidantal — Number of pages
Master's thesis	May 12, 2022	42

Tiivistelmä — Referat — Abstract

DeepRx is a deep learning receiver which replaces much of the functionality of a traditional 5G receiver. It is a deep model which uses residual connections and a fully convolutional architecture to process an incoming signal, and it outputs log-likelihood ratios for each bit. However, the deep model can be computationally too heavy to use in a real environment. Nokia Bell Labs has recently developed an iterative version of the DeepRx, where a model with fewer layers is used iteratively. This thesis focuses on developing a neural network which determines how many iterations the iterative DeepRx needs to use. We trained a separate neural network, the stopping condition neural network, which will be used together with the iterative model. It predicts the number of iterations the model requires to process the input correctly, with the aim that each inference uses as few iterations as possible. The model also stops the inference early if it predicts that the required number of iterations is greater than the maximum amount. Our results show that an iterative model with a stopping condition neural network has significantly fewer parameters than the deep model. The results also show that while the stopping condition neural network could predict with a high accuracy which samples could be decoded, using it also increased the uncoded bit error rate of the iterative model slightly. Therefore, using a stopping condition neural network together with an iterative model seems to be a flexible lightweight alternative to the DeepRx model.

ACM Computing Classification System (CCS):
Computing methodologies → Machine learning → Machine learning approaches → Neural networks
Networks → Network types → Mobile networks

Avainsanat — Nyckelord — Keywords

deep learning, 5G, radio receiver

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Acknowledgements

I would like to thank my supervisor Arto Klami, who gave me valuable feedback throughout the process of writing this thesis. I would also like to thank Mikko Honkala, Janne Huttunen and Dani Korpi from Nokia Bell Labs, for their valuable help and support.

Contents

1 Introduction **1**

2 Neural networks **3**
2.1 Feedforward neural networks 3
2.2 Convolutional neural networks 5
2.2.1 Fully convolutional networks 7
2.2.2 Depthwise separable convolutions 7
2.2.3 Residual connections 8
2.2.4 Dilated convolutions 9
2.3 Training a neural network 9
2.3.1 Optimizers 10

3 Deep learning receiver **13**
3.1 Transmission of signals with OFDM 13
3.2 DeepRx 17
3.2.1 Iterative DeepRx 20
3.3 Related works 21

4 Stopping condition neural network **23**
4.1 Creating the training and validation data 23
4.2 Training the stopping condition neural network 26
4.3 Inference with DeepRx using the stopping condition 28

5 Results and discussion **31**
5.1 Parameters and inference time 31
5.2 Performance of the SCNN 32
5.3 Comparison to other DeepRx models 34

6 Conclusion **37**

Bibliography **39**

1. Introduction

Deep learning offers an opportunity to address various problems. In wireless networks, deep learning has been proposed to be used for various tasks, such as resource management and traffic prediction in the data link layer of the network [1], or in the physical layer for channel estimation [2], and for transmitting and receiving data [3, 4].

Transmission and reception of data in wireless systems is a complicated process which contains multiple steps. The transmitter needs to complete steps such as adding error correcting code to the transmission, turning sets of bits into *modulated symbols* with certain amplitude and phase, and combining multiple of these symbols into one signal which will be transmitted by an antenna. The receiver on the other hand has to complete multiple steps to remove the noise from the signal, and to transform the signal into bits. In MIMO (multiple-input and multiple-output), where multiple antennas are used to transmit and receive data, the receiver has to complete additional steps to process the received signals and to recover the actual transmitted signals. MIMO DeepRx [5] is a deep learning receiver which works in a setting with multiple transmitting and receiving antennas, and it replaces much of the functionality of the traditional receiver with a deep fully convolutional neural network. However, as the MIMO DeepRx is a deep model with many layers, the computations require a large amount of power and time. In this thesis we investigate how to do this more efficiently.

An iterative version of the DeepRx was recently developed by Nokia Bell Labs. The model has fewer layers which are used in an iterative fashion, and the model has fewer trainable parameters than the deep model. However, calculations with this model can be slow and heavy as well, as each computation uses a fixed number of iterations, no matter the input. To make the usage of the iterative model more flexible, we will train a separate neural network, the stopping condition neural network (SCNN), which predicts the number of iterations the iterative model requires to process the input correctly. The expected result is that an iterative model which uses an SCNN requires fewer iterations than an iterative model which does not, and that using an SCNN does not increase the errors of the iterative model significantly.

In Chapter 2 we give an overview of neural networks, with a focus on concepts which are relevant for this thesis, such as fully convolutional neural networks and train-

ing a neural network. In Chapter 3, we discuss how a sequence of bits is transformed into radio waves in OFDM-systems, and how a radio wave is transformed back into a sequence of bits in the receiver. We also discuss the deep learning receiver architecture known as DeepRx, as well as an iterative version of the DeepRx. At the end of Chapter 3, we discuss other deep learning solutions that have been proposed to be used in the receiver. In Chapter 4 we implement a stopping condition neural network that predicts the number of iterations an iterative DeepRx requires to process an input. In Chapter 5 we show the results of using a stopping condition neural network with an iterative DeepRx. We compare the performance of such a network to two models, a deep model and an iterative model which does not use an SCNN. Conclusion and discussion of future work is in Chapter 6.

2. Neural networks

Neural networks are a family of predictive computational models often used in machine learning to detect patterns in the data and perform tasks such as classification. Their simple computational structure is inspired by the structure of the brain, where units known as neurons perform tasks and send signals to other neurons [6, 7]. Neural networks, like the brain, learn the task they are given by looking for patterns in the received data, and strengthening and weakening connections between neurons. While the human brain is a much more complex network of neurons than the artificial neural networks, neural networks in machine learning have proved to be able to perform well in different tasks, for example in semantic segmentation of images [8] and machine translation [9].

In this Chapter, we discuss the different neural network architectures and concepts related to this thesis. We first give an overview of feedforward neural networks, explaining how the input propagates forward in the network, and how the output is produced. We then give an overview of convolutional neural networks and fully convolutional networks, which are special neural network architectures often used in specific tasks, when the input is an image or other multidimensional tensor. We then explain three different concepts related to convolutional neural networks: depthwise separable convolutions, residual connections and dilated convolutions. These concepts will be useful later in Chapter 3 and 4, when we discuss DeepRx. Finally, in Section 2.3 we explain how artificial neural networks are trained.

2.1 Feedforward neural networks

Feedforward neural networks consist of an input layer, one or more hidden layers, and an output layer [10, 11]. Each layer contains computational units known as neurons (as well as bias units). Each layer is connected to the next and previous layer by weights, which determine how much each input to a neuron affects the output of the neuron. In a fully connected network the layers are fully connected: each neuron is connected to all the neurons of the next and the previous layer. When we discuss fully connected layers later, we mean a feedforward neural network where the layers are fully connected. The

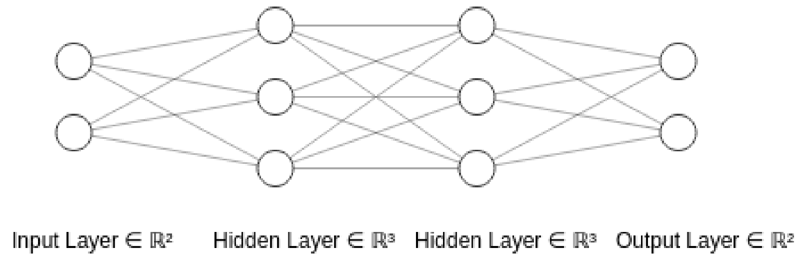


Figure 2.1: A fully connected network with two hidden layers, each containing three neurons.

size of the layers differs based on the type of the layer. In a fully connected neural network the input layer is the same size as the input vector. The hidden layers can have any size, and the output layer's size depends on the task: in a classification task, the size is typically the number of possible classes. The output of the last layer, the output layer, is the output of the whole network. A fully connected network is shown in Figure 2.1.

To make calculations effective, the inputs to a layer and the weights are presented as matrices and vectors. The hidden layers receive their input from the previous layer and calculate their output with the equation

$$h = g(\mathbf{W}^T x + b), \tag{2.1}$$

where $\mathbf{W}$ is the weight matrix, x is the input to the layer, b are the biases, and g is the activation function [10]. The result is a vector of outputs of the hidden layer, which will function as the input to the next layer.

The activation functions used in calculating the output of a layer introduce non-linearity to the model. There are multiple different activation functions, and we will briefly discuss three of them: the sigmoid function σ, the ReLU (rectified linear unit) activation function, and the softmax function. In mathematical terms, these functions are defined as

$$\mathrm{ReLU}(x) = \max(0, x), \tag{2.2}$$

$$\sigma(x) = \frac{1}{1 + \exp(-x)}, \tag{2.3}$$

$$\mathrm{softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}. \tag{2.4}$$

The ReLU function is often used in the hidden layers as it can be calculated quickly by doing a simple max-operation. The sigmoid function σ pushes the values to a range

between 0 and 1. The softmax function uses the input values to create a probability distribution, and it is often used in the output layer [10].

In addition to these layers, a batch normalization layer is often used before a hidden layer. A batch normalization layer normalizes the input to have 0 mean and unit variance, and it learns to scale and shift the input. This process has been found to increase the speed of the training and a model's accuracy [12]. In mathematical terms, batch normalization is defined as

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_\beta^2 + \epsilon}}, \tag{2.5}$$

$$y_i = \gamma \hat{x}_i + \beta, \tag{2.6}$$

where Equation 2.5 normalizes the input and Equation 2.6 scales and shifts the normalized values. In the equations, x_i is sample i in mini-batch B, μ_B and σ_β^2 are the mean and variance of the mini-batch, and γ and β are the learned scaling and shifting parameters. During inference, the mean and variance of the whole population is used instead of the mean and variance of the mini-batch, to make the output only depend on the input.

2.2 Convolutional neural networks

Convolutional neural networks (CNNs) are a special neural network architecture used in tasks where the input to the network is multidimensional, for example an image. A typical CNN consists of three types of layers: convolutional layers, pooling layers and fully connected layers [10, 13, 14]. An input tensor is given as input to the first convolutional layer, which extracts features from the input using kernels and a convolution operation (Figure 2.2). The kernels are small tensors with an equal number of channels as the input, and during training they learn to detect certain features in the input, such as an edge in an image. During the convolution operation, the inner product of a kernel and the input at each point is calculated, and output to its corresponding position in the output tensor (called a feature map), which has a depth of one. Typically many kernels are used per convolutional layer, with each kernel learning to detect one feature in the input, and each kernel outputting a feature map. Using kernels instead of fully connected layers reduces the number of parameters in the network significantly, as a single kernel is used for all the positions in the input, and it is much smaller in size than the input (for example 3×3). In mathematical terms, the output of the convolution operation for a pixel of a 2D-image at position x, y is defined as

$$S(x,y) = (K * I)(x,y) = \sum_{m}^{\hat{m}} \sum_{n}^{\hat{n}} I(x-m, y-n)K(m,n), \tag{2.7}$$

where I is the image, K is the kernel, $\hat{m}$ and $\hat{n}$ the width and height of the kernel.

After all the feature maps have been created, the feature maps are passed through a non-linear activation function. Because each feature map has a depth of one, the output of the convolution layer is a tensor with depth equal to the number of kernels used, and with a width and height that depend on the stride and the padding parameters. The stride parameter defines how much a window is moved after each calculation. With padding, the feature map received as input is padded with zeroes, helping with calculations close to the edges of a tensor.

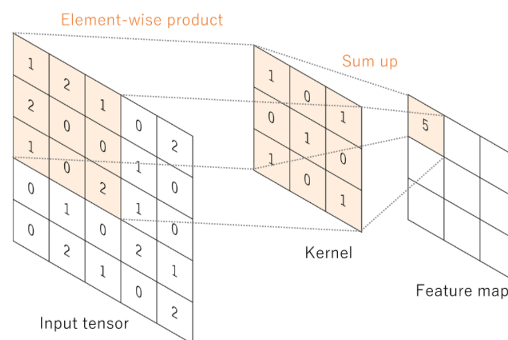


Figure 2.2: The kernel operation calculates the inner product of the kernel and the input at each position, and outputs the value to a feature map [14].

The pooling layer follows a convolutional layer, and it is used to reduce the size of the feature maps. The intuition of the pooling layers is that they capture the local information of the feature maps and reduce it to a single number, reducing the size of the feature map and making the information in the feature maps more coarse. This is achieved by sliding a window through the feature maps and by using a pooling operation. The pooling operation looks at the values of a feature map inside the window, and depending on the pooling operation used, outputs a single value that best describes all those values. In the case of max pooling operation, the value is simply the maximum of all the values inside the window, whereas the average pooling operation outputs the average of the neighborhood. Similar to the convolutional layer, stride and padding parameters can be defined by the user.

After the convolutional layers and pooling layers are the fully connected layers. The output of the last layer before the fully connected layers is reshaped to a fixed-size vector, which the fully connected layers process, resulting in the output of the whole network [10, 13, 14].

2.2.1 Fully convolutional networks

While CNNs are useful for different tasks related to images, they do have a limitation. Because a feedforward neural network requires fixed-size inputs, the reshaping of the output of the last layer before the fully connected layers must always result in a vector of the same size. This means that the input image must always be the same size. Another problem is that with typical convolutional neural networks, tasks such as pixel-level classification are quite difficult to do, because local areas in the input are represented by a single value due to the pooling operations.

Fully convolutional neural networks can be used to overcome these limitations. In fully convolutional networks the size of the input image does not have to be fixed. The fully connected layers can be turned into 1×1 convolutional layers, or they can be removed completely. When the fully connected layers are turned into 1×1 convolutional layers, the coarse output of these latter layers can be fused with the finer-grained outputs of earlier layers of the network. This leads to an output tensor that is the same size as the input [8]. A fully convolutional network might not have pooling layers at all, in which case the convolutionalization and fusing of outputs of different layers might not be required at all. If only convolutional layers are used, the width and height of the input can remain constant, and only the depth of the tensor changes. An example of such a network is the DeepRx [15], which we will discuss more in Chapter 3.

In the next three subsections we will discuss depthwise separable convolutions, residual connections and dilated convolutions, which are special concepts and additions related to convolutional neural networks, and which we will be using later in Chapter 3 and Chapter 4.

2.2.2 Depthwise separable convolutions

The convolution operation can be optimized to require less parameters and less computation time. Because the kernels of a convolutional layer have the same number of channels as the input image, to output a feature map with depth one, the kernels handle both spatial correlations and cross-channel correlations [16]. With depthwise separable convolutions, the calculation of these two correlations is decoupled. Instead of one inner product of two tensors, we will first perform a spatial convolution for each of the channels separately. After this a 1×1 pointwise convolution is performed depthwise. Computing the convolution this way reduces the parameters required significantly, and improves the speed of the computations.

2.2.3 Residual connections

Training very deep networks can cause the training and test loss to be high, even without overfitting. The loss can be reduced by implementing residual connections between layers, so that the input to a layer is also passed around it [17]. The residual connections can skip multiple layers by stacking the layers of the network into residual blocks. In mathematical terms, if no residual connections are used, the output of multiple layers is marked as $y = F(x)$. With residual connections, the input to the residual block is included in the equation, changing the equation to $y = F(x) + x$, where the addition is done elementwise.

Using residual connections introduces one challenge: since the addition is done elementwise, the output of the residual block must be the same size as the input to the residual block, making the use of pooling layers inside residual blocks difficult. This problem can be fixed by using a 1×1 convolution: if the output $F(x)$ of the residual block has a different number of channels than its input x, the channel size of x can be changed to equal the channel size of $F(x)$ by using a 1×1 convolution with the correct number of output channels. If other dimensions are different, the size of them can be reduced by increasing the stride of the 1×1 convolution, though this causes some loss of information.

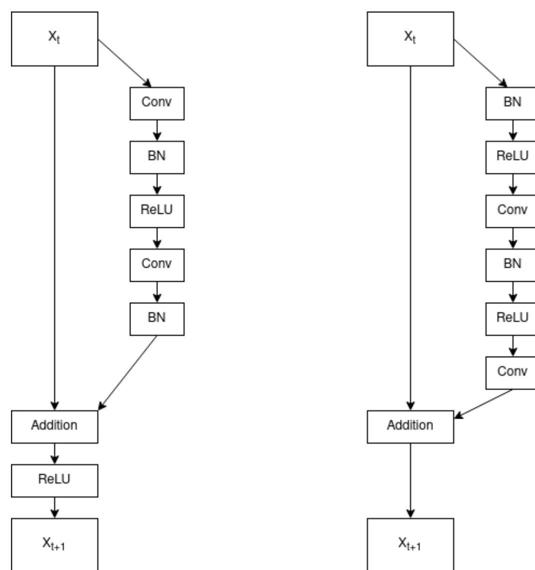


Figure 2.3: A typical residual block with 2 convolutional layers on the left, a residual block with equal number of convolutional layers and pre-activation on the right. Using pre-activation, the batch normalization and activation layers are before the convolutional layer in the residual block [18].

A pre-activation architecture [18] can be used with deep residual models to improve their ability to generalize and to make their training easier (Figure 2.3). With pre-activation, the architecture of a residual block is changed. In a typical residual

block, the convolutional layer is first, followed by batch normalization and an activation function. With pre-activation, the batch normalization is first, followed by an activation layer and a convolutional layer.

2.2.4 Dilated convolutions

In normal convolutions, the convolution operation is done between the kernel and an area on the image that is the same size as the kernel. To allow the kernels to use an area larger than their size without increasing the number of parameters, dilated convolutions can be used. In dilated convolutions, a set number of zeros is introduced between the values of a kernel. The number of zeros depends on the dilation factor: dilation factor 1 means a kernel with no added zeros, dilation factor 2 means we add one zero between the values [19]. An example of this can be seen below, where the matrix on the left is a normal kernel, and the matrix on the right is a 2-dilated kernel.

$$f_1 = \begin{bmatrix} 3 & 5 & 5 \\ 2 & 1 & 4 \\ 2 & 7 & 9 \end{bmatrix}, f_2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 5 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 1 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 7 & 0 & 9 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

2.3 Training a neural network

During training a neural network is given training examples x, y repeatedly, where x is the input to the network and y is the target value, the value we want the network to output. The examples are given one by one or in batches, and they pass through the network and produce an output $\hat{y}$. Using the output of the network $\hat{y}$ and the target value y, the loss function calculates the loss of the network. The loss is propagated back through the network using backpropagation, and the gradient of the loss is calculated. Using the gradient $\nabla_W L(y, \hat{y})$ and an optimizer function, all the weights $\mathbf{W}$ are updated slightly (defined by the learning rate hyperparameter) to a direction that reduces the loss. While the backpropagation algorithm is an important part of learning in neural networks, we will not discuss its details here. For a more detailed explanation, see section 6.5 of *Deep Learning* [10].

The training of a model continues for multiple epochs by constantly feeding training examples to the model and updating the weights with an optimizer. One epoch consists of all the training data, and once all the training data has been fed to the network, another epoch begins. A separate validation set is used to see how the model performs with data that is not in the training set, and to see if the model overfits. Overfitting occurs when a model learns a training set too well and does not work well with data outside the training set. When this happens, the training loss keeps decreasing but the validation loss starts increasing. To avoid this, different regularization methods can be used, three of which are important for this thesis: early stopping, weight decay and dropout. In the early stopping method we keep track of the validation set loss, and once the validation loss starts increasing, the training is stopped [10, 11]. The weight decay is used to control the size of the weights. The size of the weights decreases a certain amount during each update, forcing the network to learn to use smaller weights. In addition to the decaying of weights, L1-regularization can be used to introduce sparsity into the weight matrix, setting some weights to zero. The last regularization method, dropout, is used during training to select a random set of neurons during each mini-batch, effectively removing those neurons from the network for that mini-batch. This forces the network to optimize only those parameters that are left in the network [10].

2.3.1 Optimizers

There are multiple different optimizers that can be used to optimize the weights. Common iterative optimizer functions are stochastic gradient descent (SGD) [10] and Adam [20]. In SGD the gradients of the loss of a batch of inputs are averaged and the weights are updated using this average:

$$W = W - \epsilon\Big(\frac{1}{b}\nabla_W \sum_{i}^{b} L(y^i, \hat{y}^i)\Big), \tag{2.8}$$

where b is the batch size, $L(y^i, \hat{y}^i)$ is the loss of the i:th output and i:th target value, ∇_W is the gradient of the loss wrt. all the weights of the network, and ϵ the learning rate [10].

By using a *momentum*-hyperparameter v [21], the direction of the updates during each time step is affected by the updates of the previous time step. In other words, with momentum the updates are applied using a moving average of the gradients of each time step:

$$v = \alpha v - \epsilon \nabla_W \Big(\frac{1}{b} \sum_i^b L(y^i, \hat{y}^i) \Big), \tag{2.9}$$

$$W = W + v, \tag{2.10}$$

where v is the *momentum*-hyperparameter and α defines how much the previous value of v affects the current value [10].

The Adam optimizer works by keeping track of the moving average of the gradients (first moment estimate m_t, similar to momentum) and the square of the gradients (second order estimate v_t). The first moment estimate is an estimate of the mean of the gradients, and v_t an estimate of its uncentered variance. Unlike in SGD with momentum, where momentum is a single scalar value, in Adam each parameter has its own first and second order moments that are adapted during training. Adam uses two separate hyperparameters during training to decay both m_t and v_t. Before applying the updates to the weights, Adam does two more steps. First it does some bias-correction to the estimates m_t and v_t, as they are initialized as vectors of 0s, which biases them towards 0. Then it calculates the ratio r_t of the bias-corrected terms, using this and the step-size parameter α to update the gradients. The ratio r_t effectively means distance from an optimum, with a higher ratio meaning a larger distance from an optimum, allowing larger steps [10, 20].

We will be using a more recent optimizer, the LAMB optimizer [22], which can be used to enable better learning in a distributed environment where the total batch size is larger. Due to the effects a too large total batch size has on the generalization performance of the model, however, the total batch size cannot grow too large [23]. The LAMB optimizer uses Adam as its base algorithm, and it uses layer-wise learning rates which are adapted during training. The equations of the LAMB-algorithm for a single time step t are shown in the equations below [22].

$$g_t = \frac{1}{b} \sum_{j}^{b} \nabla_l(y^j, \hat{y}^j), \tag{2.11}$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t, \tag{2.12}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2, \tag{2.13}$$

$$m_t = \frac{m_t}{1 - \beta_1^t}, \tag{2.14}$$

$$v_t = \frac{v_t}{1 - \beta_2^t}, \tag{2.15}$$

$$r_t = \frac{m_t}{\sqrt{v_t + \epsilon}}, \tag{2.16}$$

$$x_{t+1}^{(i)} = x_t^{(i)} - \eta_t \frac{\phi ||x_i^{(i)}||}{||r_t^{(i)} + \lambda x_t^{(i)}||} (r_t^{(i)} + \lambda x_t^{(i)}). \tag{2.17}$$

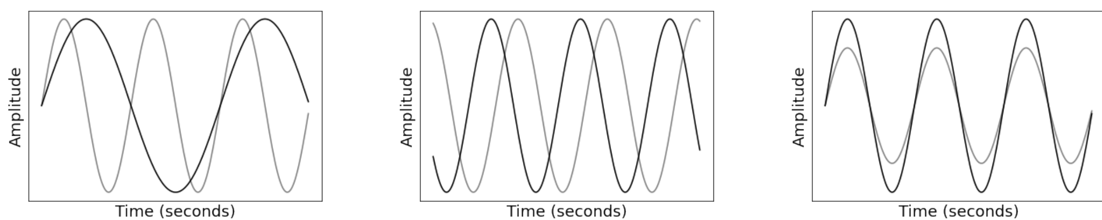
In Equation 2.11, the gradients of the loss wrt. to the weights of time step t are first calculated and stored into the variable g_t. In Equation 2.12 and 2.13, parameters β_1 and β_2 are used to first decay the previous values of m_t and v_t, after which m_t and v_t are updated using the calculated gradient g_t and the parameters β_1 and β_2. Equations 2.14, 2.15 and 2.16 handle the bias correction of m_t and v_t, and the calculation of their ratio r_t. Equation 2.17 is specific to the LAMB-algorithm. It applies the updates to the weights one layer i at a time, normalizing the update and decaying weights. In the update line, r_t^i stands for the calculated ratio r_t of layer i, and λ is the weight decay parameter.

3. Deep learning receiver

The DeepRx model, which we will be working with in this thesis, replaces much of the functionality of the traditional receiver. To understand better what the DeepRx does, and what the purpose of our research is, we need to first explain how data has been traditionally transmitted and received. In this Chapter, we give an overview of how data is transmitted and received when using OFDM, the DeepRx architecture, and discuss some other deep learning methods and solutions for receivers that have been researched.

3.1 Transmission of signals with OFDM

Before discussing how bits are turned into radio waves and back, we will briefly discuss the radio waves themselves. The properties of a radio wave that are relevant for this thesis are *frequency*, *amplitude* and *phase*. The base unit for frequency is hertz (Hz), and it describes how many cycles, or repeating events there are per second. A simple example of this is a sine wave, where the trip from the top of a wave to the bottom and back up again is one hertz. The amplitude is the height of the wave, and phase describes how much the wave is shifted horizontally. These properties are shown in Figure 3.1. The scale of amplitude and time are purposefully not shown, to make the figures as simple as possible.



(a) Two time domain signals with different frequencies.

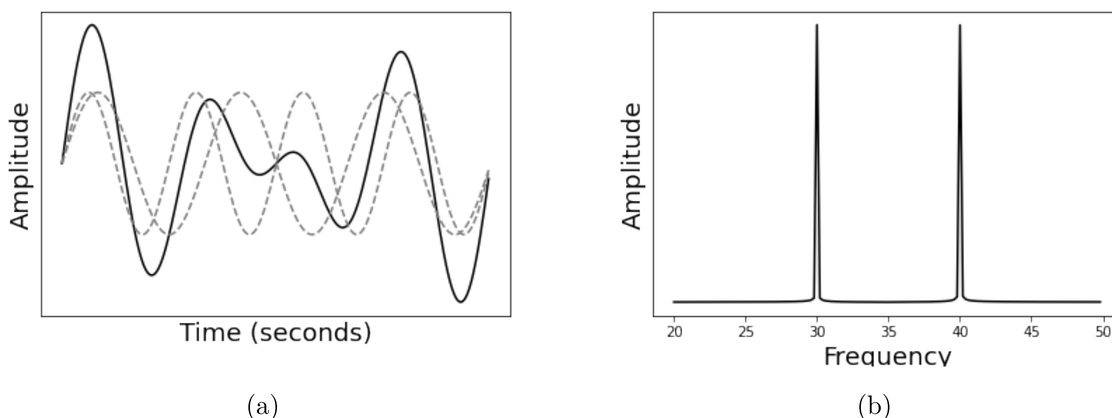
(b) Two time domain signals with the same frequency but a different phase.

(c) Two time domain signals with the same frequency but a different amplitude.

Figure 3.1

A signal can be represented in frequency domain or time domain. The signals in Figure 3.1 are an example of time domain signals: time is on the x-axis, and the y-axis describes the changes in the signal. When the signal is represented in the frequency domain, the frequency is on the x-axis and the power, or amplitude is on the y-axis. A frequency domain representation describes how much power each frequency has (Figure 3.2(b)).

In OFDM the frequencies of a transmission are selected so that they are orthogonal to each other; when the power of one frequency peaks, the power of others is zero. By using an operation known as inverse fast Fourier transform (IFFT) multiple frequencies can be combined into a single time domain signal, which can be then separated back to the frequency components with fast Fourier transform (FFT) [24, 25, 26]. An example of this is shown in Figure 3.2, where (a) shows a single time domain signal consisting of two different time domain signals, and (b) shows the results of the FFT. After FFT, the single signal has been separated into its frequency components (Figure 2.1(b)).



(a) (b)

Figure 3.2: On the left, a single time domain signal (black) consisting of two different frequencies (grey dotted lines). On the right, the same signal split into its frequency components in frequency domain, after FFT.

The process of transmitting and receiving data in 5G is shown in Figure 3.3. When transmitting data, the first step is the channel coding, where error detecting bits are attached to the data, and the number of bits transmitted is matched with the available resources[26]. These are shown in Figure 3.3 as *LDPC-encoding*, which is the error detection part, and *rate matching*, which handles the matching of the number of transmitted bits to available resources. In Figure 3.3, channel coding is followed by *symbol mapping*, which means modulating the data with a modulation and coding scheme (MCS) and mapping the resulting symbols to physical resource blocks (PRBs).

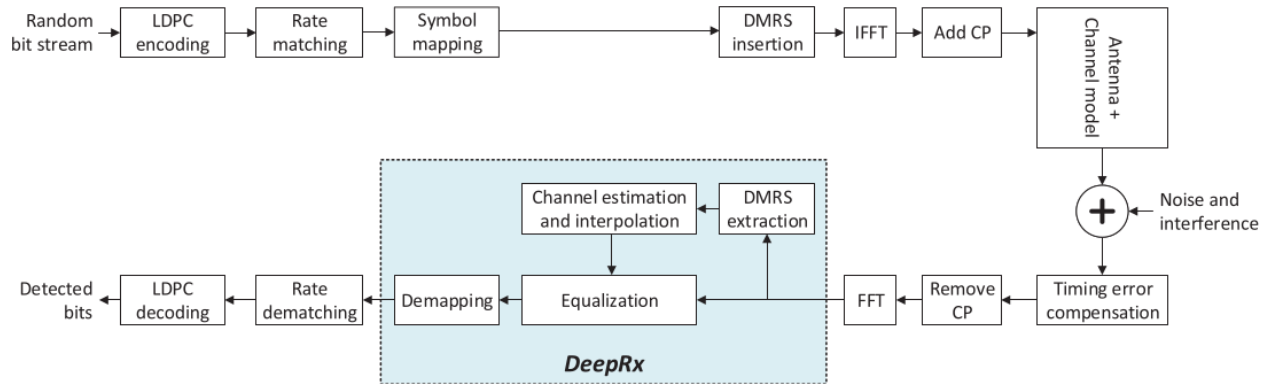


Figure 3.3: Flowchart showing how data is transmitted and received in 5G, and which parts DeepRx replaces. The top shows the transmitter, and the bottom the receiver. Picture from [15], © 2021 IEEE (reprinted with permission).

Modulation is a complicated operation, and as it is not the focus of this thesis, we will only give a simplified explanation. There are multiple ways to modulate the data, QAM-modulation being one of the options. By using QAM-modulation, different sets of bits can be represented by varying the amplitude and phase of a signal [26]. Each combination of amplitude and phase is unique, and the signal it results in is called a modulated symbol. The selected MCS determines how many bits a modulated symbol represents; in 16-QAM a modulated symbol represents 4 bits, whereas in 64-QAM symbols this is 6 bits [24, 25, 26]. A 16-QAM and a 64-QAM constellation are shown side by side in figure 3.4.

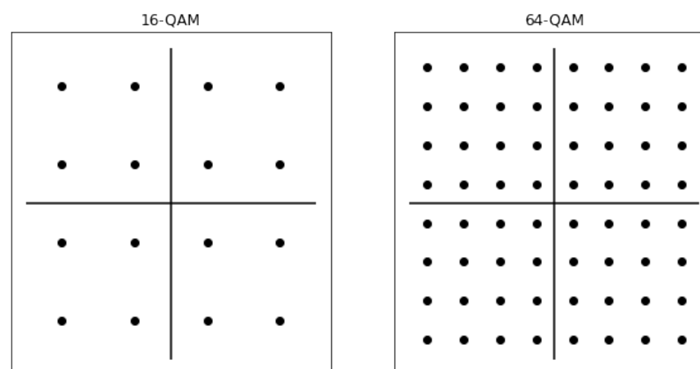


Figure 3.4: 16-QAM and 64-QAM constellations. A simplified explanation for QAM-modulation is that the x-axis determines the amplitude of the signal and the y-axis determines the phase.

In Figure 3.4 each point (symbol) represents a set of bits. While 16-QAM uses

fewer bits per symbol, the amplitude and phase difference between different symbols is larger. This makes it easier for the receiver to determine what symbol is received, even if the received symbol is noisy [26]. Generally, a lower MCS is better for channels with bad conditions, where the received data is very noisy.

In MIMO the modulated symbols can be transmitted in parallel using *spatial multiplexing*. In spatial multiplexing, the data is split into *MIMO layers*, which are transmitted in parallel using multiple antennas and same frequencies [27]. The resources of a single antenna are divided into time-frequency grids called physical resource blocks (PRBs). A single PRB in 5G has space for 14 OFDM symbols on the x-axis and 12 subcarriers (frequencies separated by a set amount, 15kHz in our case) on the y-axis, with a total space of 168 resource elements (REs) in the grid. Each modulated symbol is mapped to a single RE, and pilot symbols are added to specific REs (*DMRS insertion* in Figure 3.3). The pilot symbols are symbols that are known, and they are used in the receiver to estimate the state of the channel. In MIMO, each MIMO layer has different pilot symbol positions, selected so that the positions are not in use in other MIMO layers. Figure 3.5 shows an example of the PRBs and pilot symbols used in a transmission with two MIMO layers.

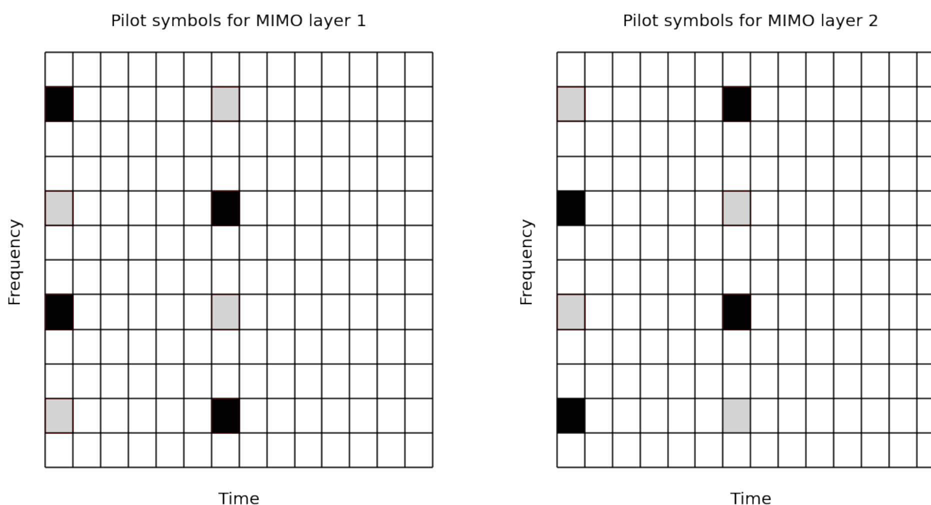


Figure 3.5: A figure showing the positions of pilot symbols (black) and modulated data symbols (white) on the PRBs when two MIMO layers are used. Each square is a resource element. On y-axis are the subcarriers. The positions of pilot symbols of one MIMO-layer are not in use in the other MIMO layer (gray squares). Inverse Fast Fourier Transform is used on a column of the PRB to create an OFDM symbol, which is a time domain signal.

After the addition of the pilot symbols, inverse Fast Fourier Transform (IFFT) is used to turn the data of the different subcarriers of a column into a single time domain

OFDM symbol. After this, a cyclic prefix (CP), which is a piece copied from the end of a symbol, is added to the beginning of each OFDM symbol. Cyclic prefix is used to help with the issue of inter-symbol interference (ISI), which occurs when a received symbol interferes with another received symbol, and the symbols are mixed [28, 29, 15].

On the receiver side there are similar steps but in reverse order. The cyclic prefixes are removed, and Fast Fourier Transform is used to turn the received time domain signal into the frequency domain components it consisted of. Once the frequency domain symbols are retrieved, the receiver uses the pilot symbols received to estimate the channel. The channel estimate is used to remove the effects of ISI on the received data symbols, using a process known as *equalization*. Finally, the log-likelihood ratios (LLRs) for the received symbols are calculated in the *demapping* process. We define the LLRs as

$$L_{ijl} = \log\left(\frac{Pr(c_l = 0|\hat{x}_{ij})}{Pr(c_l = 1|\hat{x}_{ij})}\right), \tag{3.1}$$

where $Pr(c_l = b|\hat{x}_{ij})$ is the conditional probability that a bit at position l of a received symbol is b, given the received symbol $\hat{x}_{ij}$. Once calculated, the LLRs are fed through the *rate dematching* and *LDPC decoding* processes. If the LDPC decoding succeeds, all the bits were received successfully [15].

3.2 DeepRx

DeepRx [15] is a deep fully convolutional neural network that is used to replace much of the functionality of a traditional receiver (see Figure 3.3). It replaces the channel estimation, equalization and LLR calculation parts of the traditional receiver with a neural network, taking as input the signal after the Fast Fourier Transform, and outputting the LLRs of bits. The neural network introduced in the original DeepRx paper is restricted to a SIMO (single-input, multiple-output) system, where the expected number of transmitting antennas is 1. However, we will base our iterative model on the MIMO version of the DeepRx, where there are multiple transmitting antennas, and where each layer, meaning a data stream from a transmitting antenna to a receiving one, has their own pilot symbols [5]. The input to the SIMO DeepRx is similar to that of the MIMO DeepRx, so we will briefly discuss that first, as understanding it helps understanding the MIMO model better.

The input $\mathbf{Z}$ to the SIMO DeepRx is an $S \times F \times 2N_c$ grid, where S is the number of OFDM symbols and F is the number of subcarriers used. To explain N_c, we will first discuss the three parts that the input consists of. $\mathbf{Y}$ is the received signal after Fast Fourier Transform, including the pilot symbols. This is a grid of shape $S \times F \times N_r$,

where S and F are as above and N_r is the number of receiving antennas. The second part, $\mathbf{X}_p$, is a grid of shape $S \times F$, which contains zeros in all positions except those where the pilot symbols are positioned in the input. The third part, $\mathbf{H}_r$, is also a grid of shape $S \times F \times N_r$, and contains the precomputed raw channel estimates for the pilot positions. The result of concatenating $\mathbf{Y}$, $\mathbf{X}_p$ and $\mathbf{H}_r$ is a tensor of shape $S \times F \times N_c$, where $N_c = 2N_r + 1$. The received signal is a complex number, so the real and imaginary parts of the complex numbers are separated into different channels. This doubles the number of input channels, resulting in the final input shape of $S \times F \times 2N_c$. The network outputs a tensor with shape $S \times F \times B$, where B is the number of bits per modulated symbol [15].

The input to the MIMO DeepRx is different to the SIMO DeepRx because the MIMO DeepRx network contains additional preprocessing layers, preDeepRx and multiplicative layer, which are trained together with the actual DeepRx model [5]. The input to the first preprocessing layer consists of the concatenation of two tensors: the tensor containing received data signals, which has the shape $F \times S \times N_R$ and the tensor containing the raw-channel estimates, which has the shape $F \times S \times N_R \times N_T$, where N_T is the number of transmitting antennas. Unlike the SIMO DeepRx however, the real and imaginary parts of the complex numbers are not separated into separate channels. The concatenation results in a tensor with shape $F \times S \times N_R(1 + N_T)$. The preDeepRx layer is a 3-block residual network, and its output is fed into the multiplicative layer. The multiplicative layer learns which inputs and channels need to be multiplied, and it scales the imaginary parts of the complex numbers of the channels. The output of the multiplicative network is fed to the actual DeepRx model [5]. The architecture of the MIMO DeepRx is shown in Figure 3.6

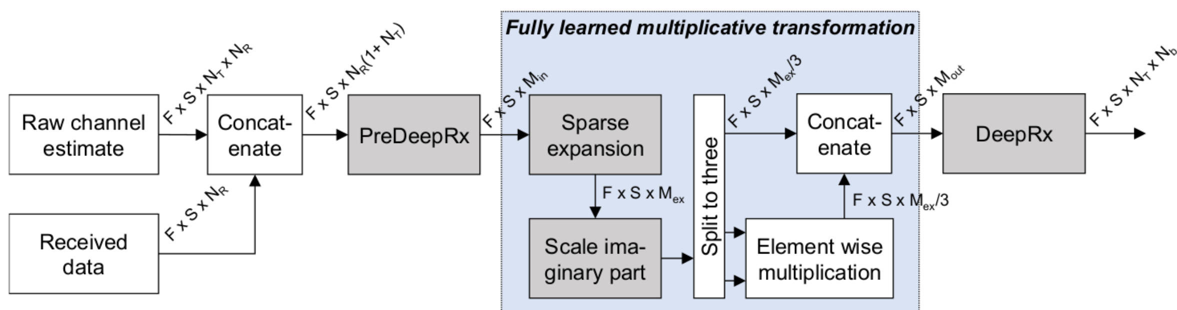


Figure 3.6: The architecture of the MIMO DeepRx. The received data and raw channel estimates are concatenated and fed as input to the preprocessing layers. The output of the multiplicative layer is used as input to the actual DeepRx model. Blocks where trainable weights are used are shown in gray. Picture from [15], © 2021 IEEE (reprinted with permission).

The actual DeepRx model is similar in both the MIMO and SIMO case, with

the exception that the MIMO model has more channels. The model is a deep fully convolutional network with depthwise separable convolutions, residual connections and batch normalization. It consists of multiple residual blocks and a final layer, which will output the final output of the network. The model does not contain pooling layers to keep the resolution constant ($S \times F$) throughout the network. The model also makes use of dilations [15, 5]. A single residual block is shown in Figure 3.7.

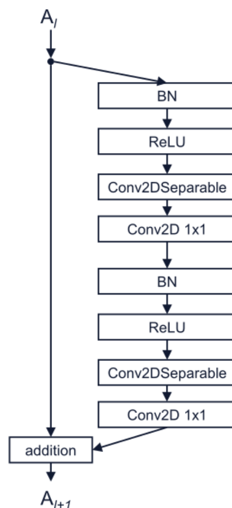


Figure 3.7: A single residual block of DeepRx. One block contains depthwise separable convolutions, batch normalization operations and non-linearity operations (ReLU). Picture from [15], © 2021 IEEE (reprinted with permission).

The loss function used for the model is

$$L_q(\theta) = \log_2(1 + s_q)CE_q(\theta), \tag{3.2}$$

where s_q is the signal-to-noise ratio of the qth sample, and $CE_q(\theta)$ represents the binary sigmoid cross-entropy for the qth sample and weight vector θ. The binary sigmoid cross-entropy is defined as

$$CE(\theta) = -\frac{1}{DB} \sum_{i,j \in D} \sum_{l=0}^{B-1} \left(b_{ijl} \log(\hat{b}_{ijl}) + (1 - b_{ijl} \log(1 - \hat{b}_{ijl})) \right). \tag{3.3}$$

In the equation above, D stands for the number of data carrying symbols received, B is the number of bits per modulated symbols, b_{ijl} is the lth bit of the symbol in position (i,j) of the received data matrix. The last term in Equation 3.3, $\hat{b}_{ijl}$, is the DeepRx model's output for the same bit, passed through the sigmoid function (Equation 2.3). Simply put, $\hat{b}_{ijl}$ estimates the probability that the bit is one [15]. The network predicts for each received bit the probability that the bit is one, and by using binary sigmoid

cross-entropy the network learns to give high probability to bits that are one, and low probability to bits that are not.

To allow the network to work with different QAM schemes (where number of bits per received symbol varies), the model makes use of the hierarchies between different constellations. The hierarchies mean that 4 points (symbols) in a higher modulation and coding scheme can be mapped to a single point in a lower modulation and coding scheme. The number of outputs for a symbol is set to 8 (256-QAM), and the outputs of the network are masked so that only the actual bits used in modulation are used. The MIMO DeepRx configuration used in this thesis is shown in Table 3.1.

Layer	Output channels	Additional parameters
PreDeepRx residual block 1	128	kernel size=3, dilation=(1,1)
PreDeepRx residual block 2	128	kernel size=3, dilation=(1,1)
PreDeepRx residual block 3	128	kernel size=3, dilation=(1,1)
Multiplicative layer	52	
Residual block 1	128	kernel size=3, dilation=(1,1)
Residual block 2	128	kernel size=3, dilation=(1,1)
Residual block 3	256	kernel size=3, dilation=(3,2)
Residual block 4	256	kernel size=3, dilation=(3,2)
Residual block 5	256	kernel size=3, dilation=(3,2)
Residual block 6	256	kernel size=3, dilation=(6,3)
Residual block 7	256	kernel size=3, dilation=(3,2)
Residual block 8	256	kernel size=3, dilation=(3,2)
Residual block 9	256	kernel size=3, dilation=(3,2)
Residual block 10	128	kernel size=3, dilation=(1,1)
Residual block 11	128	kernel size=3, dilation=(1,1)
Output convolutional layer	16	

Table 3.1: Configuration of the MIMO DeepRx.

3.2.1 Iterative DeepRx

The DeepRx, as shown earlier, is a deep fully convolutional neural network. In our case, the DeepRx has 3 residual blocks in the PreDeepRx part, and 11 residual blocks in the DeepRx part (Table 3.1). While a deep model with a fixed number of layers can be powerful, it does contain a large number of parameters and it can be computationally very expensive to use. A possible solution for this is to use an iterative model. Nokia Bell Labs has developed an iterative version of the MIMO DeepRx model with only a

few residual blocks that have a recurrent connection to the beginning. The difference between this iterative model and a non-iterative model is shown in Figure 3.8. The output of the last residual block is used by the output layer during each iteration to get the LLRs of the iteration. The output of the last residual block is concatenated with the LLRs and concatenation is used as input for the next iteration. During training the model is trained with a fixed number of iterations and loss is calculated after each iteration using the LLRs, and summed at the end to get the final loss of the network. In mathematical terms, the loss function of an iterative model is

$$CE_{iter} = CE(b, \hat{b}^{(0)}) + CE(b, \hat{b}^{(1)}) + ... + CE(b, \hat{b}^{(N)}), \tag{3.4}$$

where $CE(b, \hat{b}^{(k)})$ is the binary sigmoid cross-entropy loss with predicted bit probabilities $\hat{b}^{(k)}$ of kth iteration.

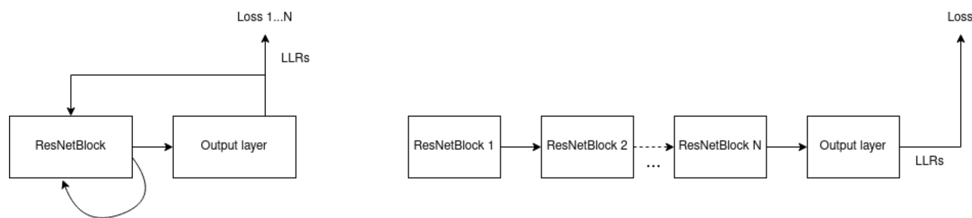


Figure 3.8: An iterative DeepRx model with one residual block on the left, a non-iterative model with N-ResNetBlocks on the right. In the iterative model the loss is calculated after each iteration, and the final loss is the sum of the losses.

3.3 Related works

The use of deep learning methods in the receiver is an active field of research. Deep learning solutions for receivers can be categorized into two categories, solutions that replace or enhance single parts of the receiver, and solutions that replace multiple parts with a single machine learning model [30, 3]. Examples of the former category are using deep learning for tasks such as MIMO-detection [31] or channel estimation and signal detection [2]. Examples of the latter category are works such as the already discussed DeepRx, as well as the Deep-Waveform [4], where the used machine learning architecture is a deep complex-valued convolutional neural network (DCCN). The DCCN takes as input a synchronized time-domain signal, and outputs the LLRs of the bits.

A hybrid approach is to replace multiple parts in the receiver with multiple machine learning models. Goutay et al. propose an ML-enhanced receiver architecture for multi-user MIMO [30], which replaces multiple components of the traditional receiver with machine learning models. The architecture utilizes convolutional neural

networks, using separate convolutional neural networks for channel estimation and for calculating the LLRs. By copying the machine learning models for every user and sharing the trainable parameters between the copies, the architecture scales to work with multi-user MIMO. All the machine learning components are trained jointly using the estimated LLRs and binary sigmoid cross-entropy, similar to DeepRx. The architecture uses pre-activation residual blocks similar to DeepRx, with depthwise separable convolutions and dilated convolutions. Due to the fact that it only uses machine learning during channel estimation and calculating the LLRs, the ML-enhanced receiver is computationally less expensive than the original DeepRx.

Deep learning solutions have also been proposed for tasks different than the receiver tasks we have discussed so far. For example, Hanna et al. [32] discuss the possibility of using a Dual Path Network (DPN) in the blind decoding of symbols and classification of modulation and coding schemes. In this setup the receiver works blindly, receiving data without having information such as the modulation and coding scheme used, the length of the received signal or the preamble (a signal used in tasks such as time synchronization and channel estimation [29]). The DPN is an architecture consisting of a signal path and a feature path. The signal path uses parameters provided by the feature path to remove effects such as ISI and noise from the signal. The feature path predicts these parameters by using neural networks, using the original signal and the outputs of different steps of the signal path as input.

Some previous work has researched the training of both the transmitter and the receiver jointly using the autoencoder architecture [33, 3]. Autoencoder is an architecture where the network has two parts, an encoder and a decoder. The encoder learns to encode its input to some (typically) lower dimensional representation, and the decoder learns to decode this representation back to the original input. Using autoencoders, the transmitter functions as the encoder which encodes the transmitted data, and the receiver functions as a decoder which decodes the encoded data. This type of architecture does seem like a good solution for transmitting data, as the output of the receiver should be exactly what was transmitted.

4. Stopping condition neural network

While an iterative version of the DeepRx has fewer trainable parameters than the deep version, using the iterative DeepRx can lead to unnecessary computations. Because the iterative model is trained with a fixed number of iterations, during inference the model keeps iterating until this fixed number of iterations has been completed, even if the signal could already be decoded earlier. In an ideal case the model should iterate only as much as is required for the signal to be decoded successfully. To support varying levels of iteration, we train a separate neural network that takes the output of an iteration as input and predicts how many iterations are required before its input can be decoded. We will call this separate neural network the stopping condition neural network (SCNN).

4.1 Creating the training and validation data

The SCNN is a model which uses the output of an iteration of the iterative DeepRx, and predicts how many iterations are required. The goal is to train a model which can accurately predict the number of iterations required. Before training an SCNN, however, an iterative model needs to be trained first, and the training and validation sets for the SCNN need to be created. To create the data sets we will make use of a pretrained iterative DeepRx, that is trained with a fixed number of iterations. To enable performance comparisons with the MIMO DeepRx model, we will train the fixed iterations network with simulated data that is generated for a 4x2 MIMO case, using a simulator created with Matlab's 5G Toolbox [5]. The data contains different levels of noise, with the signal-to-noise ratio (SNR) varying from -3 dB (very noisy signal received) to 31dB (very good quality signal received). The QAM-modulation used for the data is 16-QAM (4 bits per symbol).

We train a single model with 6 iterations using data simulated for a 4x2 MIMO case (4 transmitting antennas and 2 receiving antennas). The architecture of the iterative model is shown in Table 4.1. The PreDeepRx-layers consist of one residual

Layer	Output size
PreDeepRx residual block 1	$312{\times}14 \times 128$
Multiplicative layer	$312{\times}14 \times 54$
DeepRx residual block 1	$312{\times}14 \times 128$
DeepRx residual block 2	$312{\times}14 \times 128$
DeepRx residual block 3	$312{\times}14 \times 128$
DeepRx residual block 4	$312{\times}14 \times 112$
Output layer	$312{\times}14 \times 16$

Table 4.1: Configuration of the iterative DeepRx used to generate the data.

block with 128 kernels. The DeepRx part consists of four residual blocks and one output layer. The convolutional layers of the first three DeepRx residual blocks have 128 kernels of size 3x3. The convolutional layers of the last DeepRx residual block have 16 fewer kernels, because its output is concatenated with the output layer's output, which has 16 kernels, resulting in a total output size of $312 \times 14 \times 128$. This tensor is used as input during the next iteration. The LLRs that are fed to the LDPC-decoder are in the output layer's output, reshaped to size $2 \times \times 312 \times 14 \times 8$. The first DeepRx residual block uses 3-dilated kernels in the horizontal direction and 2-dilated kernels in the vertical direction. The next three DeepRx residual blocks use 1-dilated kernels (normal kernels) in both horizontal and vertical directions.

Once the model is trained, we will feed the trained model its training and validation data sets as input and create the training and validation data sets for the SCNN using Algorithm 1 shown below.

The algorithm describes a case where batch-processing is not used, and the network only takes a single input at a time. The output of each iteration is fed into an LDPC-decoder (line 6). If the decoding is a success, the number of iterations so far and the output of the iteration are saved, and the execution of the first loop is stopped. If the decoding fails, the output of the iteration is saved, and the loop continues (lines 7-11).

After the first loop, if the decoding was not successful at any point, the number of iterations required to decode is set to be higher than the maximum number of iterations (lines 12-13). After this, a second loop is started, during which the number of iterations until decoding was successful is combined with the output of an iteration, to create a final training data point (lines 14-18). The final data point will contain the output of an iteration, and the number of iterations remaining until that output was successfully decoded.

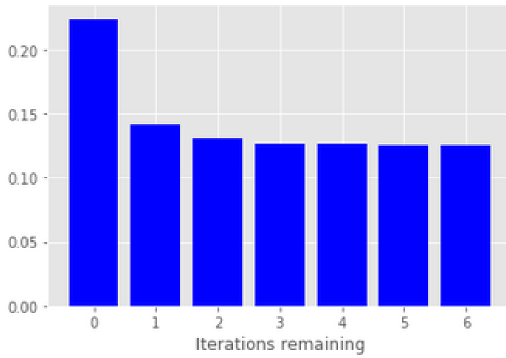
Algorithm 1: Create SCNN data set

```
1  outputs = array[maxIter]
2  input = inputToNetwork
3  target = 0
4  for iter = 1 to maxIter do
5      input = runIteration(input)
6      ldpc = ldpcDecode(input)
7      if ldpc == 1 then
8          target = iter
9          outputs[iter] = input
10         break
11     outputs[iter] = input
12 if target == 0 then
13     target = maxIter + 1
14 for iter = 1 to maxIter do
15     if target > iter then
16         item = outputs[iter]
17         numItersRemaining = target − iter
18         saveTrainingDataPoint(item, numItersRemaining)
```

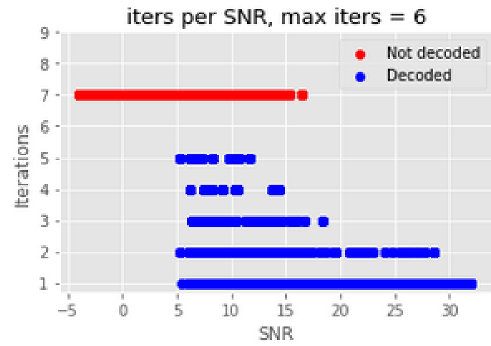
In Figure 4.1, we show some statistics about the training data set generated with this method using the 4x2 MIMO model. As is seen in Figure 4.1 (a), the distribution of the target value is quite even. The only exception is target category 0, which is nearly double the size of the other values. The reason for this is seen in Figure 4.1 (c), where it is shown that most of the decoded samples were decoded after just one iteration. Because of the second loop, each sample that required more than one iteration also contributes to the lower values, as the loop continues until that sample is decoded and $numItersRemaining$ on line 17 is zero. However, samples that were decoded after one iteration only iterate the second loop once, increasing only the number of samples in category 0. A scatter plot showing the SNRs and number of iterations of samples is shown in Figure 4.1 (b). Many samples that were decoded, especially samples with higher SNR, were decoded with just 1 iteration. However, some successfully decoded samples with lower SNR required more iterations.

The distribution of successful and unsuccessful decoding per SNR is shown in Figure 4.1 (d). The plot contains information about the amount of data successfully decoded or unsuccessfully decoded per SNR. The distribution of the samples is skewed to the right towards the lower SNRs. This explains the high number of values in column

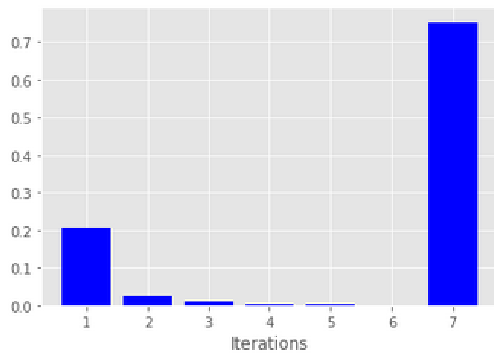
7 of Figure 4.1 (c), because a large portion of the data is data with SNR lower than 2, below which the model produces poor results. Samples with SNR higher than 13 were almost always decoded successfully, and samples that fall between 2 and 13 were either decoded or not, with a higher number of samples not decoded.



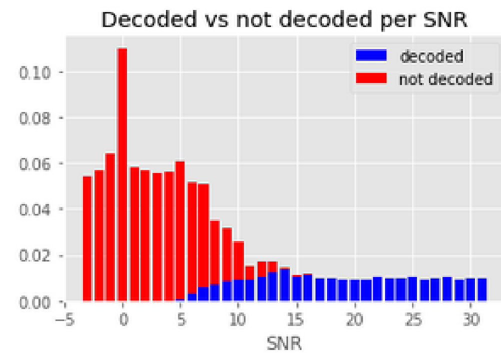
(a) Distribution of iterations remaining until decoding was successful. This is used as target when training the SCNN.



(b) Scatter plot showing the iterations required for decoding the signal per SNR. Higher SNR data could be decoded with 1 or 2 iterations, some lower SNR samples required more iterations.



(c) Distribution of iterations required. Most samples that were decoded successfully were decoded after 1 iteration.



(d) Distribution of successfully decoded and unsuccessfully decoded data, with SNR on the x-axis.

Figure 4.1

4.2 Training the stopping condition neural network

The architecture of the SCNN is a residual network with k residual blocks similar to what is shown in Figure 3.7. The size of the tensor should be reduced before it is fed to a fully connected network later, so we will use an average pooling operation after each residual block to gradually reduce the size of the data. After the residual blocks, there are the fully connected layers and an output layer. The output layer contains t

units, where t is the size of the target array. We use softmax as the activation function of the output layer to create a probability distribution for the outputs. The goal of training the SCNN is to maximize the probability of the correct output and minimize the probability of other outputs. The loss function used for the network is categorical cross-entropy.

We will train an SCNN using data generated with the iterative model. A single input to the SCNN models has dimensions $312 \times 14 \times 128$, and the target is a one-hot encoded array of size 7. The model uses average pooling layers between residual blocks, aiming to reduce the dimensions as much as possible without losing too much information. Because fully connected layers introduce a lot of parameters to the network, the dimensions of the data need to be as small as possible, to allow for smaller fully connected layer sizes. The configuration of the SCNN is shown in Table 4.2.

Layer	**Output size**	**Additional parameters**
Residual block 1	$312\times14 \times 128$	kernel size=5, dilation=(1,1)
Average Pooling	$156\times7 \times 128$	Pool size=(2,2)
Residual block 2	$156\times7 \times 128$	kernel size=5, dilation=(1,1)
Average Pooling	$78\times4 \times 128$	Pool size=(2,2)
Residual block 3	$78\times4 \times 128$	kernel size=5, dilation=(1,1)
Average Pooling	$39\times2 \times 32$	Pool size=(2,2)
Residual block 4	$39\times2 \times 32$	kernel size=5, dilation=(1,1)
Average Pooling	$20\times1 \times 32$	Pool size=(2,2)
Fully connected layer 1	64	dropout rate=0.3
Fully connected layer 2	64	dropout rate=0.3
Softmax layer	7	

Table 4.2: Configuration of the stopping condition network which uses average pooling operations between residual blocks.

The average pooling layers of the SCNN gradually reduce the size of the input, halving the width and height of the input each time. The result is a tensor with dimensions $20 \times 1 \times 32$. The tensor is flattened to a vector of size 640 that is fed as input to the fully connected layers. The fully connected network of the SCNN has 2 hidden layers with 64 neurons, and an output layer with 7 neurons. The hidden layers use a dropout rate of 0.3 as a regularization method.

The model is trained for 200 000 iterations, where one iteration equals one training sample. Learning rate is increased gradually for 800 iterations until it reaches 0.00001 per sample. The learning rate stays at 0.00001 per sample until 30% of the iterations

have been completed, after which it is decreased linearly to 0. Validation is run every 20 000 samples to keep track of the validation loss. In addition to validation loss, we will keep track of two other metrics, validation accuracy and validation probability. Validation accuracy is the number of correct predictions in a batch divided by the batch size. A prediction is correct if the number of iterations with the highest probability equals the target value. Validation probability is the probability of the predicted value. In an ideal case the probability of prediction is always 100%, however it will be less in most, if not all of the validation cases. The probability will be used later in the inference phase, where the prediction probability must be higher than a given threshold.

4.3 Inference with DeepRx using the stopping condition

There are two ways to use the SCNN together with the iterative DeepRx, both shown below in Algorithm 2 and 3. Intuitively, Algorithm 2 is an actual stopping condition, as during each iteration the SCNN is used to check whether to continue iterating or to quit. Algorithm 3, on the other hand, only uses the SCNN once, or until a prediction has a high enough probability. Using the SCNN only once is an ideal choice, as it results in fewer computations during inference, however we will test both methods to see how they perform.

Algorithm 2: DeepRx inference, run SCNN after every iteration

```
1  inputToNextIter = input
2  for iter = 1 to maxIter do
3      inputToNextIter, LLRs of iteration = runIteration(inputToNextIter)
4      numItersRemaining, probability = SCNN(inputToNextIter)
5      if probability < threshold then
6          continue
7      if numItersRemaining == 0 then
8          return LLRs of iteration
9      if (numItersRemaining + iter) > maxIter then
10         return LLRs of iteration
11 return LLRs of iteration
```

Predicting the number of iterations after each iteration requires the setting of two stopping conditions for the prediction. If the prediction is zero iterations, the iteration can be stopped instantly, as the output of the iteration can be decoded al-

ready. Similarly, if the prediction leads to a higher number of iterations than the maximum number of iterations, the iteration is stopped, as the output of the iteration cannot be decoded within the maximum number of iterations, making continuing pointless. In addition, a probability threshold can be set, so that if the probability of the prediction is lower than the threshold, the prediction is ignored and the iteration continued. This way the more uncertain predictions are not taken into account, and only the predictions which are more certain affect the stopping of the network. However, finding a good threshold does require some experiments with the model.

Algorithm 3: DeepRx inference, run SCNN once

```
1  stopIter = null
2  inputToNextIter = input
3  for iter = 1 to maxIter do
4      inputToNextIter, LLRs of iteration = runIteration(inputToNextIter)
5      if stopIter == null then
6          numItersRemaining, probability = SCNN(inputToNextIter)
7          if probability > threshold then
8              stopIter = iter + numItersRemaining
9      if stopIter == iter then
10         return LLRs of iteration
```

Predicting the number of iterations once is a more straightforward process: after the first iteration the SCNN is called with the output of the iteration, and the number of iterations run is capped to the output of the SCNN. In terms of performance this is the ideal way to use the SCNN, as the number of additional computations is minimal. As with the previous case, a probability threshold can be set to only accept predictions that are highly probable. If a threshold is used however, the SCNN might be called multiple times, until the threshold is crossed or the maximum number of iterations is completed.

5. Results and discussion

In the first section of this Chapter we compare the total amount of parameters between a deep model, an iterative model, and an iterative model which uses an SCNN. Then we will test the accuracy of the predictions of the SCNN, using the Algorithms 2 and 3, with and without a threshold. In the last section we will test how an iterative model with an SCNN performs compared to a deep model and an iterative model without an SCNN, by comparing the uncoded bit error rates of the models. Finally, we compare the amount of time an inference takes on average with different models.

5.1 Parameters and inference time

As stated in Chapter 4, the iterative model should be more lightweight than the deep model, even when it is using an SCNN. The deep model we compare our models to is a MIMO DeepRx model with DeepRx layers similar to what was described in the original DeepRx paper, however with the added PreDeepRx layers that the MIMO case requires. The configuration of this network is shown in Table 3.1, while the iterative model is the model used to create the training data for the SCNN, shown in Table 4.1.

A comparison of the parameters of the different models is shown in Table 5.1. The deep model has close to 3000 000 parameters, whereas the iterative model has less than a third of that. Adding the SCNN to the iterative model increases the number of parameters by a few percent, setting its total amount to approximately 36% of the number of parameters of the deep model. Clearly The iterative model is a more lightweight model than the deep model, even when using the SCNN.

Model	Number of parameters	% of deep model parameters
Deep model	2,952,720 (2.9M)	100%
Iterative model	920,999 (0.92M)	31.19%
Iterative model + SCNN	1,062,638 (1.06M)	35.99%

Table 5.1: Comparison of the number of trainable parameters of each model.

While the smaller number of parameters makes it a more lightweight model, the computations can still take as much time as with the deep model, if the number of iterations used is, on average, large. We test the performance of an iterative model which uses an SCNN (which we call NN_{scnn}) using the validation set. We are interested in the distribution of iterations completed when the SCNN is used, as well as the distribution of the decoding results of the samples.

5.2 Performance of the SCNN

In this section we discuss the accuracy of the SCNN when used with an iterative model. More specifically, we discuss the false negatives and positives, as well as true negatives and positives. We consider a prediction of the SCNN to be correct if a sample is decoded and is processed in 1-6 iterations, or if a sample is not decoded successfully and predicted to take 7 iterations. Similarly, an incorrect prediction is one where the sample is not decoded but is predicted to take 1-6 iterations, or the sample is decoded and predicted to take 7 iterations. We refer to samples which were not decoded, and which are in categories 1-6 as false positives, and decoded samples in category 7 as false negatives. We consider 4 scenarios: predict once during inference with no threshold and with threshold 0.5 (probability over 50%), and predict after each iteration with no threshold and threshold of 0.5. For each scenario, we ran over 80000 inferences with NN_{scnn} using the validation set. The number of iterations each inference required was saved and mapped to the result of the decoding. Table 5.2 shows the results for the first two scenarios, where the SCNN is only called once (until probability exceeds threshold).

As seen in the table, with or without a threshold the SCNN was able to predict with relatively high accuracy which samples could not be decoded, with over 90% of the undecoded samples predicted to be in category 7. Similarly, in both scenarios the most successfully decoded samples required only 1 iteration, with a smaller number of samples requiring 2 or 3 iterations. In both scenarios none of the samples were predicted to take 4 or 5 iterations. Without a threshold, 21 samples required 6 iterations but were not decoded successfully, whereas with a threshold this category was empty. Some samples that used 1 to 3 iterations could not be decoded, implying that they either would have required more iterations, or the SCNN should have predicted a category 7. The 21 samples in category 6 can be considered to be outliers, as the training data had no samples that belonged to category 6.

Predicting once with a threshold of 0.5 resulted in a distribution of iterations quite similar to the distribution of the no threshold case, with the most notable differences being in categories 1, 2 and 7. In category 1 using a threshold reduces the number of

Predict once, no threshold		
Iterations	Decoded/not decoded samples	% out of 84600
1	48882/2338	57.78%/2.76%
2	488/235	0.58%/0.28%
3	212/43	0.25%/0.05%
4	0/0	0%/0%
5	0/0	0%/0%
6	0/21	0%/0.02%
7	42/32339	0.05%/38.23%
Predict once, threshold 0.5		
Iterations	Decoded/not decoded samples	% out of 84600
1	48360/1758	57.16%/2.08%
2	1334/382	1.58%/0.45%
3	148/128	0.17%/0.15%
4	0/21	0%/0.02%
5	0/0	0%/0%
6	0/42	0%/0.05%
7	233/32194	0.28%/38.05%

Table 5.2: Table showing the number of samples decoded or not decoded when using an SCNN with an iterative DeepRx. On top, no threshold was used, on bottom, a threshold of 0.5 was used.

false positives (that is, samples that should be in category 7, but are in categories 1-6) by nearly a 1000. The number of true positive samples in category 2 is increased by 1%, with a small increase in false positives however, and a decrease in the true positives of category 1.

Table 5.3 shows the results of using Algorithm 2, for a case when no threshold was used, and when a threshold of 0.5 was used. The results are similar to the ones above, with the iterations having a similar distribution as them. Using a threshold caused an increase in the false negatives in category 7, as well as false positives in categories 2-3.

The number of false positives in categories 1-3, as well as the near empty categories of 4-6 could be explained by the uneven distribution of iterations in the training data. If we compare the distribution of Figures 5.2 and 5.3 to the distribution of iterations in the training data (Figure 4.1(c)), we see that in all distributions iteration 1 has the most samples out of all the categories 1-6. However, while the training data has samples in columns 4-5, the SCNN predicted close to 0 samples in those columns. It can be reasonable to assume that the samples that weren't decoded in 1-3 iterations are samples that might belong to categories 4-5 (or category 7), as the number of it-

Predict during each iteration, no threshold		
Iterations	Decoded/not decoded total	% out of 84600
1	48830/2279	57.72%/2.69%
2	637/214	0.75%/0.25%
3	64/0	0.08%/0.03%
4	0/0	0%/0%
5	0/0	0%/0%
6	0/0	0%/0%
7	63/32513	0.07%/38.43%
Predict during each iteration, threshold 0.5		
Iterations	Decoded/not decoded total	% out of 84600
1	48376/1761	57.18%/2.08%
2	1381/379	1.63%/0.45%
3	85/85	0.10%/0.10%
4	0/0	0/0
5	0/0	0/0
6	0/0	0/0
7	233/32300	0.28%/38.18%

Table 5.3: A table similar to 5.2, but SCNN was run after each iteration.

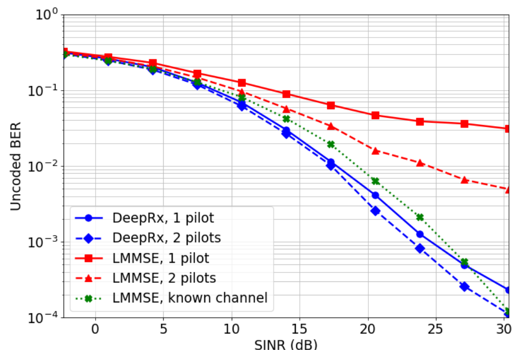
erations predicted can be expected to follow a somewhat similar distribution to the one shown in Figure 4.1. The uneven distribution in the training data can affect the performance of the SCNN. Because an iterative model which uses an SCNN should have a similar distribution of iterations as the training data, it could be the case that the SCNN has predicts the samples which took 4-5 iterations incorrectly, to improve overall performance and to generalize better.

5.3 Comparison to other DeepRx models

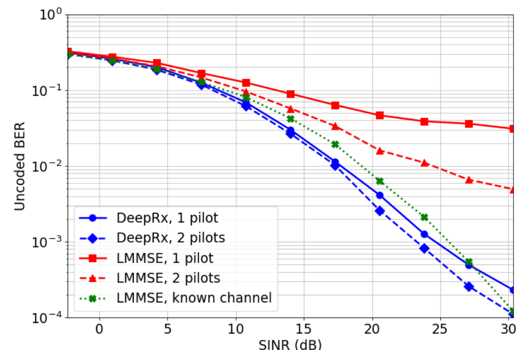
To get a better understanding of how well the NN_{scnn}, we will next compare its performance to two other models: the deep MIMO DeepRx NN_{deep}, and an iterative model NN_{fixed} which does not use an SCNN. We will use the uncoded BER (bit error rate) as a metric. Before calculating the uncoded BER, the LLRs need to be turned into bits. Each LLR in the output is turned into a bit using the following equation:

$$bit = \begin{cases} 1, & \text{if } LLR < 0 \\ 0, & \text{otherwise.} \end{cases}$$

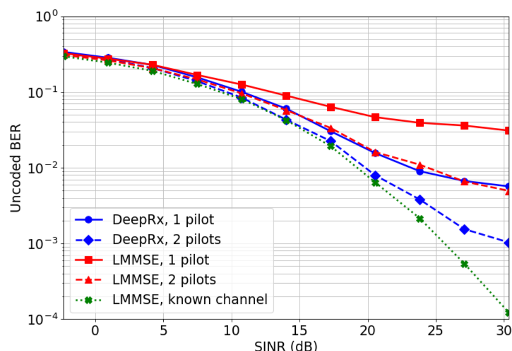
The resulting bits are compared to the target bits, and the ratio of false predictions and total bits define the uncoded BER [15]. Figure 5.1 shows the uncoded BER of the different models. The plots show SINR (signal to interference and noise) on the x-axis, and the uncoded BER on the y-axis. In our case the SINR will equal SNR, because in the data the interference is 0.



(a) Uncoded BER of NN_{deep}.



(b) Uncoded BER of NN_{fixed}.



(c) Uncoded BER of NN_{scnn}.

Figure 5.1: Uncoded BERs of the different DeepRx models compared to the uncoded BER of a traditional receiver (described in Section 3.1). Green line shows the uncoded BER of a traditional receiver which has full channel information.

The results in the figure show that NN_{deep} and NN_{fixed} have similar uncoded BERs, with uncoded BER decreasing as SNR increases. With NN_{scnn} the results are different, as while the uncoded BER does decrease as SNR increases, the decrease is smaller than with the other two models. NN_{scnn} performs worse than the LMMSE-receiver when the channel is known (green line). The reason for the results can be found in the SCNN: as was shown in Figure 4.1(b), the higher SNR data points were decoded with only 1 or 2 iterations. Because the iterative model always completes at least 1 iteration, the false positives of category 1 (Table 5.2 and 5.3) could be samples of high SNR that would have been decoded on iteration 2. As inference was stopped after the completion of the first iteration, the sample could not be decoded perfectly,

leading to an increase in the uncoded BER in the high SNR range. The same reasoning applies to false positives in category 2. They might be samples that could be decoded in 3 iterations.

Model	Inference time avg.	Inference time sd
NN_{deep}	34.3ms	5.76ms
NN_{fixed}	50ms	5.82ms
NN_{scnn}, predict once, 0 threshold	24.55ms	4.08ms
NN_{scnn}, predict once, 0.5 threshold	25.22ms	3.72ms
NN_{scnn}, predict continuously, 0 threshold	25.12ms	5.26ms
NN_{scnn}, predict continuously, 0.5 threshold	25.34ms	4.53ms

Table 5.4: Averages and standard deviations of inference times of each model.

We also compared the time an inference required on average with each model using the validation set and a batch size of 1. While the inference time itself depends on the hardware, as well as the configurations of the different networks, the difference in the averages can provide useful information, such as how much faster the iterative model is when using a stopping condition network. The average inference times and standard deviations (sd) are shown in table 5.4.

Using an SCNN is clearly faster than both NN_{deep} and NN_{fixed}, requiring only half as much time as the fixed model, and close to 30% less time than the deep model. NN_{fixed} is the slowest, as it needs to run the four residual blocks and the output layer 6 times during each inference. While the inference time of NN_{fixed} can be reduced by using a configuration with less iterations, as long as there are samples which can be decoded with less iterations than the maximum amount, using an SCNN should be faster than not using one.

To summarize all the results, using an iterative DeepRx model with an SCNN can lead to models with significantly fewer parameters and faster inference time. The downside is that the SCNN can increase the uncoded bit error rate of the iterative DeepRx. It is possible this could be fixed by using an iterative DeepRx configuration where the iterations required to decode the samples are distributed more evenly.

6. Conclusion

In this thesis we considered the use of a stopping condition neural network in combination with an iterative DeepRx model. The motivation behind our work was that when combined with a separate stopping condition neural network, the iterative model would be able to vary the number of iterations it uses, using fewer iterations for good quality signals and more iterations for noisier signals. In the case of a very poor quality signal, which the receiver cannot decode, the stopping condition neural network would be able to predict that decoding the signal is not possible and would stop the iterative model from processing the input any further.

To train the stopping condition neural network we first trained an iterative model with a fixed number of iterations. We then created the training and validation sets for the stopping condition neural network. This we achieved by doing inference on the training and validation sets of the fixed model, using an LDPC-decoder after each iteration and saving the output of each iteration. We considered the task a multi-class classification problem. We implemented the stopping condition neural network as a residual convolutional neural network, with average pooling layers between residual blocks, and fully connected layers at the end of the network.

To test how the iterative model worked in combination with a stopping condition neural network, we compared its performance to a deep model, and an iterative model without a stopping condition neural network. We showed that the use of the stopping condition neural network does not significantly increase the number of trainable parameters. Similarly, we showed that the use of an iterative model in combination with a stopping condition neural network does have significantly less parameters than a deep model. We showed that using a stopping condition neural network does reduce the number of iterations the iterative DeepRx uses. The results showed that while the error rate of the stopping condition neural network is relatively small, it does increase the uncoded BER of the iterative DeepRx, especially in samples with high SNR. We theorized that the uncoded BER of the high SNR samples was most affected due to the uneven distribution of iterations in the training data, and due to misclassifications by the stopping condition neural network. We suggested some improvements, such as finding a better configuration for the iterative DeepRx, which would have a more even

distribution of iterations. In addition to this, our work still left room for other future work. Combining the categories into larger ones could simplify the classification task and reduce the classification errors in the high SNR samples. The task could possibly be modeled as a binary classification task, if instead of predicting the number of remaining iterations, the model would predict if iteration is stopped or not (similar to Algorithm 2, but with two classes). Finally, the threshold used for the predictions of the stopping condition neural network requires further investigations, firstly to find out if it is even viable to use a threshold, and secondly to find out that in the case it is viable, what is the correct threshold.

Bibliography

[1] Q. Mao, F. Hu, and Q. Hao, “Deep learning for intelligent wireless networks: A comprehensive survey,” *IEEE Communications Surveys Tutorials*, vol. 20, no. 4, pp. 2595–2621, 2018.

[2] H. Ye, G. Y. Li, and B.-H. Juang, “Power of Deep Learning for Channel Estimation and Signal Detection in OFDM Systems,” *IEEE Wireless Communications Letters*, vol. 7, no. 1, pp. 114–117, 2018.

[3] T. O’Shea and J. Hoydis, “An Introduction to Deep Learning for the Physical Layer,” *IEEE Transactions on Cognitive Communications and Networking*, vol. 3, no. 4, pp. 563–575, 2017.

[4] Z. Zhao, M. C. Vuran, F. Guo, and S. D. Scott, “Deep-Waveform: A Learned OFDM Receiver Based on Deep Complex-Valued Convolutional Networks,” *IEEE Journal on Selected Areas in Communications*, vol. 39, no. 8, pp. 2407–2420, 2021.

[5] D. Korpi, M. Honkala, J. M. Huttunen, and V. Starck, “DeepRx MIMO: Convolutional MIMO Detection with Learned Multiplicative Transformations,” in *ICC 2021-IEEE International Conference on Communications*, pp. 1–7, IEEE, 2021.

[6] S. Agatonovic-Kustrin and R. Beresford, “Basic concepts of artificial neural network (ANN) modeling and its application in pharmaceutical research,” *Journal of pharmaceutical and biomedical analysis*, vol. 22, no. 5, pp. 717–727, 2000.

[7] O. I. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, N. A. Mohamed, and H. Arshad, “State-of-the-art in artificial neural network applications: A survey,” *Heliyon*, vol. 4, no. 11, p. e00938, 2018.

[8] J. Long, E. Shelhamer, and T. Darrell, “Fully Convolutional Networks for Semantic Segmentation,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3431–3440, 2015.

[9] S. P. Singh, A. Kumar, H. Darbari, L. Singh, A. Rastogi, and S. Jain, "Machine translation using deep learning: An overview," in *2017 International Conference on Computer, Communications and Electronics (Comptelix)*, pp. 162–167, 2017.

[10] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016. http://www.deeplearningbook.org.

[11] D. Svozil, V. Kvasnicka, and J. Pospichal, "Introduction to multi-layer feed-forward neural networks," *Chemometrics and intelligent laboratory systems*, vol. 39, no. 1, pp. 43–62, 1997.

[12] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in *International conference on machine learning*, pp. 448–456, PMLR, 2015.

[13] W. Rawat and Z. Wang, "Deep Convolutional Neural Networks for Image Classification: A Comprehensive Review," *Neural Computation*, vol. 29, pp. 1–98, 06 2017.

[14] R. Yamashita, M. Nishio, R. K. G. Do, and K. Togashi, "Convolutional Neural Networks: An Overview and Application in Radiology," *Insights into Imaging*, vol. 9, no. 4, pp. 611–629, 2018.

[15] M. Honkala, D. Korpi, and J. M. Huttunen, "DeepRx: Fully Convolutional Deep Learning Receiver," *IEEE Transactions on Wireless Communications*, vol. 20, no. 6, pp. 3925–3940, 2021.

[16] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1251–1258, 2017.

[17] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[18] K. He, X. Zhang, S. Ren, and J. Sun, "Identity Mappings in Deep Residual Networks," in *European conference on computer vision*, pp. 630–645, Springer, 2016.

[19] F. Yu and V. Koltun, "Multi-Scale Context Aggregation by Dilated Convolutions," in *International Conference on Learning Representations (ICLR)*, May 2016.

[20] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[21] B. T. Polyak, “Some methods of speeding up the convergence of iteration methods,” *USSR Computational Mathematics and Mathematical Physics*, vol. 4, no. 5, pp. 1–17, 1964.

[22] Y. You, J. Li, S. Reddi, J. Hseu, S. Kumar, S. Bhojanapalli, X. Song, J. Demmel, K. Keutzer, and C.-J. Hsieh, “Large Batch Optimization for Deep Learning: Training BERT in 76 Minutes,” in *International Conference on Learning Representations (ICLR)*, 2020.

[23] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour,” *arXiv preprint arXiv:1706.02677*, 2017.

[24] J. Armstrong, “OFDM for Optical Communications,” *Journal of lightwave technology*, vol. 27, no. 3, pp. 189–204, 2009.

[25] N. Cvijetic, “OFDM for Next-Generation Optical Access Networks,” *Journal of lightwave technology*, vol. 30, no. 4, pp. 384–398, 2011.

[26] W. Stallings, *5G Wireless: A Comprehensive Introduction.* Addison-Wesley Professional, 2021.

[27] E. Dahlman, S. Parkvall, and J. Skold, *5G NR: The Next Generation Wireless Access Technology.* Academic Press, 2018.

[28] P. Sure and C. M. Bhuma, “A survey on OFDM channel estimation techniques based on denoising strategies,” *Engineering Science and Technology, an International Journal*, vol. 20, no. 2, pp. 629–636, 2017.

[29] G. L. Stuber, J. R. Barry, S. W. Mclaughlin, Y. Li, M. A. Ingram, and T. G. Pratt, “Broadband MIMO-OFDM Wireless Communications,” *Proceedings of the IEEE*, vol. 92, no. 2, pp. 271–294, 2004.

[30] M. Goutay, F. A. Aoudia, J. Hoydis, and J.-M. Gorce, “Machine Learning for MU-MIMO Receive Processing in OFDM systems,” *IEEE Journal on Selected Areas in Communications*, vol. 39, no. 8, pp. 2318–2332, 2021.

[31] N. Samuel, T. Diskin, and A. Wiesel, “Deep MIMO Detection,” in *2017 IEEE 18th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, pp. 1–5, 2017.

[32] S. Hanna, C. Dick, and D. Cabric, “Signal Processing-Based Deep Learning for Blind Symbol Decoding and Modulation Classification,” *IEEE Journal on Selected Areas in Communications*, vol. 40, no. 1, pp. 82–96, 2022.

[33] A. Felix, S. Cammerer, S. Dörner, J. Hoydis, and S. Ten Brink, "OFDM-Autoencoder for End-to-End Learning of Communications Systems," in *2018 IEEE 19th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, pp. 1–5, 2018.