THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

A real-time deep learning OFDM receiver

Citation for published version:
Brennsteiner, S, Arslan, T, Thompson, J & McCormick, A 2021, 'A real-time deep learning OFDM receiver', *ACM Transactions on Reconfigurable Technology and Systems*, vol. 15, no. 3, 26, pp. 1-25. https://doi.org/10.1145/3494049

Digital Object Identifier (DOI):
10.1145/3494049

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
ACM Transactions on Reconfigurable Technology and Systems

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 08. Jun. 2022

A real-time deep learning OFDM receiver

STEFAN BRENNSTEINER, University of Edinburgh, UK
TUGHRUL ARSLAN, University of Edinburgh, UK
JOHN THOMPSON, University of Edinburgh, UK
ANDREW MCCORMICK, Alpha Data Parallel Systems Ltd, UK

Machine learning in the physical layer of communication systems holds the potential to improve performance and simplify design methodology. Many algorithms have been proposed; however, the model complexity is often unfeasible for real-time deployment. The real-time processing capability of these systems has not been proven yet. In this work, we propose a novel, less complex, fully connected neural network to perform channel estimation and signal detection in an orthogonal frequency division multiplexing (OFDM) system. The memory requirement, which is often the bottleneck for fully connected neural networks, is reduced by ≈ 27 times by applying known compression techniques in a three-step training process. Extensive experiments were performed for pruning and quantizing the weights of the neural network detector. Additionally, Huffman encoding was used on the weights to further reduce memory requirements. Based on this approach, we propose the first FPGA-based, real-time capable neural network accelerator, specifically designed to accelerate the OFDM detector workload. The accelerator is synthesized for a Xilinx RFSoC FPGA, uses small-batch processing to increase throughput, efficiently supports branching neural networks, and implements superscalar Huffman decoders.

CCS Concepts: • **Computer systems organization** → *Real-time system architecture*; **Real-time systems**; • **Applied computing** → **Telecommunications**; • **Hardware** → **Digital signal processing**; **Reconfigurable logic applications**; **Hardware accelerators**; • **Computing methodologies** → *Neural networks.*

Additional Key Words and Phrases: neural networks, OFDM, FPGA, physical layer processing, machine learning acceleration, real-time

ACM Reference Format:
Stefan Brennsteiner, Tughrul Arslan, John Thompson, and Andrew McCormick. 2021. A real-time deep learning OFDM receiver. 1, 1 (October 2021), 25 pages.

1 INTRODUCTION

One of the main signal processing tasks in the physical layer of any communication system is channel estimation and signal detection. In recent years much research interest developed to replace or extend conventional channel estimation and signal detection algorithms with machine learning-based algorithms. Speculations on the high importance of physical layer machine learning in future communication networks such as 6G have been made too [31, 53]. The reason for this high interest is fourfold:

Authors' addresses: Stefan Brennsteiner, University of Edinburgh, King's Buildings, Alexander Crum Brown Road, Edinburgh, UK, stefan.brennsteiner@ed.ac.uk; Tughrul Arslan, University of Edinburgh, King's Buildings, Alexander Crum Brown Road, Edinburgh, UK, tughrul.arslan@ed.ac.uk; John Thompson, University of Edinburgh, Kings Buildings, Thomas Bayes Road, Edinburgh, UK, john.thompson@ed.ac.uk; Andrew McCormick, Alpha Data Parallel Systems Ltd, 160 Dundee St, Edinburgh, UK, am@alpha-data.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2021 Association for Computing Machinery.
XXXX-XXXX/2021/10-ART $15.00
https://doi.org/

- Higher performance over complex or unmodeled channels [47]
- Automatic compensation of hardware impairments [38]
- Increased performance due to the removal of functional block boundaries of traditional algorithms [47][38]
- Ease of design and processing due to well-established machine learning(ML) techniques

Various architectures have been proposed to realize these benefits. O'Shea et al. proposed an autoencoder-based system, in which the transmitter and the receiver are both implemented via neural networks. The transmission channel is modeled as a layer of the autoencoder and the system is optimized in an end-to-end training session [2] [37]. The autoencoder networks have the advantage of being able to choose the symbol alphabet depending on the channel conditions. Other approaches assume a conventional communication system on the transmitter side and focus on signal detection via machine learning in the receiver. The authors in [45] present a deep learning-based Viterbi detector.

One modulation technique of special interest is Orthogonal Frequency Division Multiplexing (OFDM), as it is used in many communication systems such as in the downlink of long term evolution (advanced) (LTE(A)) [43] and in the downlink of fifth generation (5G) [4]. Various proposals to perform machine learning-based channel estimation and signal detection in an OFDM system have been made. The authors in [15] present an OFDM-autoencoder capable of inserting pilots as required. A receiver-only neural network is presented in [49]. The authors present a comb-type OFDM system with fixed positions of pilots across the frequency axis. The channel estimation and signal detection is performed in a single neural network for eight sub-carriers (SC) at once. The proposals in [15] and [49] incorporate a minimal amount of assumptions about the underlying communication system and the detectors are purely machine learning-based. In contrast, model-driven proposals incorporate expert knowledge in the receivers and combine it with machine learning techniques. The authors in [16] extend a classic OFDM receiver containing a least square channel estimation and a zero-forcing detection algorithm with a neural network to increase receiver performance.

A survey laying out opportunities and challenges for deep learning in the physical layer can be found in [47]. For model-driven approaches, [22] provides a comprehensive survey. Yao et al. describe applications in the 5G Radio Access Technology Layer [48]. Broader surveys on machine learning in communications, which also include the physical layer, are presented in [52] and [35].

Despite the algorithmic advancements described above, the deployment of neural networks in actual receiver hardware has been very limited. In [13] an autoencoder is used in an over-the-air OFDM system. The autoencoder is first trained offline with a mathematical channel model and then deployed via two software-defined radios (SDRs). In the deployment stage, the SDRs act as front ends, and the online training and neural network inference is performed on Graphics Processing Units (GPUs). However, the inference is not performed in real-time and the feasibility of the hardware setup for actual real-time deployment is not assessed.

We notice a lack of real-time processing of neural-network-based communication systems in the literature. The term real-time is often used loosely in the literature. According to [29], a real-time system is defined as one whose logical correctness is based on both the correctness of the outputs and their timeliness. This implies that (a). the processing latency has to be below this predetermined time, and (b). the throughput of the processing system has to be aligned with the rate of data input to avoid the build-up of input data. Real-time processing is challenging as the model complexity of many proposed neural network receivers is infeasibly high. Also, the throughput and latency requirements of communication systems are challenging. In the LTE(A) downlink the OFDM SC spacing is 15 kHz which leads to a symbol duration of $\approx 66.7\mu s$ [43]. With the new numerologies of

up to 240 kHz SC spacing in 5G, the symbol duration can be as low as $\approx 4.17\mu s$ [4]. It is not strictly required to process each symbol within one symbol time, however, latency is a major performance metric of any receiver.

Field Programmable Gate Arrays (FPGAs) are a popular choice for implementing neural network inference as they provide the computational power required to accelerate the workload. For the purpose of accelerating the first real-time OFDM neural network detector, an FPGA is ideal as it also provides flexibility to explore custom accelerator designs at a low cost compared to application specific integrated circuits (ASICs). Further, FPGAs are extensively used in base stations, which could be targeted for future deployment of neural network detectors. Many accelerator designs have been proposed in the literature in recent years. Multiple factors determine the optimal accelerator design, including the neural network type, the chosen loop unrolling strategy, the on-chip memory and off-chip memory characteristics, and the available computing resources amongst others. However, a typical accelerator will have common elements such as on-chip buffers, computational units, control logic, host interfaces, and memory interfaces. Surveys on previously proposed FPGA accelerators can be found in [18] [44].

1.1 Contribution

A novel structure of a neural network for OFDM signal detection and channel estimation in the receiver is proposed. The algorithmic exploration undertaken in this work is based on [49], however, the new neural network structure significantly reduces the computational complexity and memory requirement, making real-time processing possible.

To further reduce the memory requirement, we analyze whether the methods of deep compression can be applied to our neural network [20]. Deep compression seeks to reduce a given neural network's memory requirement by applying three techniques: 1) pruning, 2) quantization, and 3) Huffman coding. Pruning eliminates small weight values which do not contribute much to the network's output. Quantization means to represent weights with a low number of bits while maintaining accuracy. Huffman coding compresses a stream of weights by taking the probability of each word and assigning shorter codes to more likely occurring words.

Based on the algorithmic improvements and the findings on deep compression, we develop a custom hardware accelerator specifically for the detector workload. The accelerator processes one or more neural network detectors. It supports small batch processing, superscalar Huffman decoding and branching neural networks, making it ideal for detector acceleration.

In short, we claim the first real-time neural network based OFDM detector, enabled by:

- A reduced memory OFDM neural network by a.) improved network structure and b.) compression of the network
- A Real-time capable hardware accelerator tailored to the neural-network-based OFDM detector workload

1.2 Limitations

Our work focuses on a generic OFDM system for channel estimation and decoding. However, to put the work in perspective, we provide a comparison to the cellular standard of LTE(A) and 5G wherever appropriate. We focus on the real-time processing of the neural network detector workload rather than on being fully compatible with 3GPP standards and acknowledge the following main limitations towards LTE(A) compliance:

- The position of reference signals in the two-dimensional resource element lattice in time and frequency cannot be chosen dynamically as required by the 3GPP standard for LTE [6]. Instead, at every time instance, the decoder assumes the same interleaving of reference- and

data- symbols in frequency (i.e. comb-type OFDM system). Further, reference symbols are not chosen according to the 3GPP standard.
- The modulation scheme is fixed to Quadrature Phase Shift Keying (QPSK), as the neural network is limited to that. It is not clear in the current state of research how to extend the training and inference process of neural networks in an efficient way to higher modulation schemes such as Quadrature Amplitude Modulation (QAM)64 or higher.

Despite these limitations, the work presented is important in that it evaluates the possibility of deploying full neural-network-based receivers in the future.

1.3 Outline

The rest of this paper is organized as follows. Section 2 introduces the OFDM system we study in this work as well as the used machine learning techniques, focusing on inference in fully connected neural networks. In Section 3 we present a reduced complexity neural network and expand on quantization and pruning techniques used. Section 4 lays out accelerator design considerations to support the detector workload. In Section 5 we present the proposed FPGA accelerator on circuit level. Results are presented in Section 6 and conclusions are drawn in section 7.

2 BACKGROUND

This section first discusses the orthogonal frequency division multiplexing (OFDM) system used in this work which is derived from previous work in the literature. A brief background is provided on neural networks focusing on fully connected (FC) layers as the proposed network consists of only fully connected layers.

2.1 Orthogonal Frequency Division Multiplexing

Orthogonal Frequency Division Multiplexing (OFDM) is a commonly used multiple access scheme in the downlink of wireless communication systems such as LTE(A) and 5G [4, 43]. Symbols can be arranged along the time- and frequency axis, forming a resource grid containing resource blocks (RBs). Each RB consists of multiple sub-carriers (SC) on which the transmitted information is modulated upon. To acquire channel state information, known reference symbols (i.e. pilot symbols) are typically embedded in equidistant intervals in the resource grid. These reference symbols are used by the receiver to estimate the channel and aid symbol detection. The neural network proposed in Subsection 3.1 handles these two tasks in a joint fashion. For a more detailed introduction to OFDM, channel estimation, and signal detection we refer the interested reader to [36].

2.2 Fully Connected Neural Networks

Neurons are the fundamental computational unit in neural networks and the output of a simple neuron can be written as

$$out = \Phi\left(\sum_{i=1}^{N}(\mathbf{w}_i \times \mathbf{in}_i) + b\right). \tag{1}$$

Where $\mathbf{w} \in \mathbb{R}^{N,1}$ is the weight vector, $\mathbf{in} \in \mathbb{R}^{1,N}$ is the input vector and $b, out \in \mathbb{R}$ are the bias value and the neuron's output respectively. The activation function Φ calculates the neuron's output and is often a non-linear function. The non-linearity is important as it allows a neural network consisting of multiple layers to act as a universal function approximator [12]. Common activation functions are Rectified Linear Unit (ReLU) and Sigmoid functions [8]. Multiple neurons can be connected to the same inputs. Such a grouping is often referred to as a layer, which can be expressed as the following:

$$\mathbf{out} = \Phi(W \times \mathbf{in} + \mathbf{b}). \tag{2}$$

In a fully connected network, each layer's inputs are connected to the previous layer's outputs or to the network's input in the case of the first layer. The last layer's outputs $\mathbf{out}_M$ are the neural network's output. We do not go into further detail regarding training processes but refer the interested reader to [8, 17]. For inference, the calculations are performed as shown in Algorithm 1. As processing and memory resources are limited, the loops cannot be implemented fully in parallel in hardware. A loop unrolling strategy has to be conceived to determine which parts of the algorithm are computed in parallel. We present our strategy in Section 5.

Algorithm 1: Inference in a multi-layered, fully connected neural network considering multiple input samples in a batch

input : Data: multiple $\mathbf{x} \in \mathbb{R}^{N_0}$
input : Weights: $W = \{W_1 \in \mathbb{R}^{N_0,N_1}, W_2 \in \mathbb{R}^{N_1,N_2}, ..., W_M \in \mathbb{R}^{N_{M-1},N_M}\}$
input : Bias: $B = \{\mathbf{b}_1 \in \mathbb{R}^{N_1}, \mathbf{b}_2 \in \mathbb{R}^{N_2}, ..., \mathbf{b}_M \in \mathbb{R}^{N_M}\}$
output: multiple $\mathbf{y}_M \in \mathbb{R}^{N_M}$

```
Y = {y_0 ∈ R^{N_0}, y_1 ∈ R^{N_1}, ..., y_M ∈ R^{N_M}};
foreach x do // Batch Loop
    y_0 ← x ;
    for m ← 1 to M do // Layer Loop
        for n ← 1 to N_M do // Neuron Loop
            y_{m,n} ← 0 ;
            for i ← 0 to N_{M−1} do // Input Loop
                y_{m,n} ← y_{m,n} + y_{m−1,i} × W_{m,n,i} ;
            end
            y_{m,n} ← Φ(y_{m,n} + B_{m,n}) ;
        end
    end
end
```

3 OFDM SYSTEM AND NEURAL NETWORK DETECTOR

This section introduces the OFDM system used in this work. Then a novel neural network structure is proposed which significantly reduces the computational complexity. Further, we use a method to train this new structure efficiently in a three-step process. First, the network is trained with floating-point weights and activations. Then, after training with pruned weights has approached a sufficient accuracy, quantization is introduced in the forward pass of the data through the computational graph. Thirdly, pruning is introduced in the training process. In the forward pass, a mask is applied, masking out a predefined percentage of the total number of weights, depending on the magnitude of the weight.

3.1 OFDM System

Conceptually we follow the OFDM system introduced in [49] and the software scaffold for this OFDM system is taken from [21]. The data stream is converted from serial to parallel and then Quadrature Phase Shift Keying (QPSK) modulated. Pilot and data symbols are then interleaved for each symbol-time to form the transmit vector

$$\mathbf{x}[p] := \mathbf{PilotValue}[n];\ n \in \{1, ..., N_p\}, \tag{3}$$
$$\mathbf{x}[d] := \mathbf{Data}[n];\ n \in \{1, ..., N_d\}. \tag{4}$$

Where N_p is the number of pilot symbols in the **PilotValue** vector and $p \in P := \{1, P_s, 2P_s, N_pP_s\}$ are the equidistant indices of the pilots spaced by P_s. The complex pilot symbols can be chosen randomly from the symbol alphabet, but remain the same for all packages passed to the detector. In a similar manner data symbols are assigned to transmit vector elements where $d \in \{1, ..., N\} \land d \notin P$. The total number of elements in $\mathbf{x}$ is N. Next, the transmit vector is mapped to 15kHz spaced sub-carriers via the Inverse Discrete Fourier Transformation (IDFT) and a cyclic prefix is added as

$$\mathbf{x_t}[k] = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} \mathbf{x}[n] e^{j\frac{2\pi n(k-L_{CP})}{N}}, \tag{5}$$

where $k \in \{0, ..., N + L_{CP} - 1\}$ and L_{CP} is the cyclic prefix duration in samples [32].

The transmit signal $\mathbf{x_t}$ is convolved with the channel impulse response of the same WINNER II channel model $\mathbf{h}$ as in [49] and noise is added, forming the receive signal:

$$\mathbf{y_r}[\mathbf{k}] = \mathbf{x_t}[k] \circledast \mathbf{h}[k] + \mathbf{n}[k] \tag{6}$$

The noise $\mathbf{n} \in \mathbb{C}^N$ is assumed to be Additive White Gaussian Noise(AWGN). On the receiver side eq. (5) is reversed by removing the cyclic prefix and by performing a Discrete Fourier Transformation (DFT) to recover the complex frequency domain symbols $\mathbf{y} \in \mathbb{C}^N$. In order to allow a real-valued neural network to perform detection and channel estimation, the complex-valued receiver signals are converted to a real representation by interleaving real and imaginary values:

$$\mathbf{y_{real}}[2i] := \Re(\mathbf{y}[i]); \tag{7}$$
$$\mathbf{y_{real}}[2i + 1] := \Im(\mathbf{y}[i]);\ i \in \{1, .., N\} \tag{8}$$

Where $\mathbf{y_{real}} \in \mathbb{R}^{2N}$ is the real input vector. The neural network then recovers an estimate of the transmitted symbol

$$\widehat{\mathbf{x}} = f_{NN}(\mathbf{y_{real}}). \tag{9}$$

The target of the neural network thus is to recover the transmit vector $\mathbf{x}$ with as few errors as possible. The structure of the used OFDM system for training and evaluation is depicted in fig. 1.

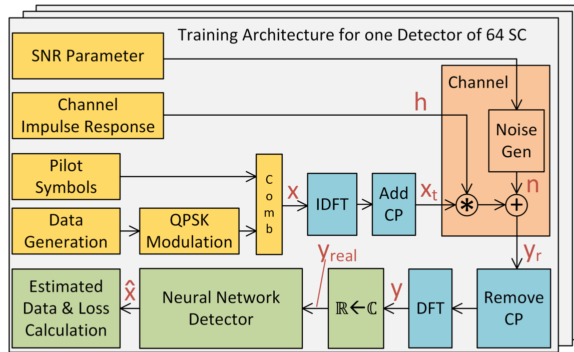


Fig. 1. Neural Network training and evaluation setup for a single neural network detector. Each detector processing 64 SC has to be trained independently. Conceptually based on [49].

3.2 Algorithmic Complexity Reduction

The proposed receiver consists of D neural network detectors, each of which processes $B \times S$ SC. Where B is the number of branches in one detector and S is the number of SC processed per branch. The neural network structure differs significantly from previous work. An entire neural network needs to be deployed in [49] for the detection of 8 SC, leading to a high number of neural network weights. We propose to share the first two layers of the neural network detector and then branch out in the last two layers for each set of SC, as seen in fig. 2. Our experiments show similar performance of this architecture and the architecture presented in [49], while complexity is significantly reduced.

On the other hand, if the first two layers are not shared but reduced in size, such that the sum overall complexity of the neural network is the same as the proposed branching neural network, we observe a complete loss of performance. This indicates that the first two layers in the branching neural network are capable of extracting common features important to the detection in all branches. Our branching neural network takes advantage of this observation.

We chose the number of SC per branch to be eight (denoted by $S = 8$) and the number of branches per detector to be eight as well (denoted by $B = 8$), leading to 64 SC to be processed per detector. Each detector is completely independent of the other detectors, i.e. each detector has its own reception chain as outlined in fig. 1. This is not ideal as multiple narrow-band reception chains are required on the system level. Future work will investigate the possibilities to integrate all detectors within one broad-band reception chain (e.g. IDFT/FFT size of 1024 instead of currently 64). For now, we focus on the real-time processing of the neural networks. We evaluate our design by choosing D to be equal or less than 19, which gives us a maximum total of 1216 SC processed. Not considering the 16 last SC of the 19th detector, this results in 1200 effective SC and the sum-bandwidth of the occupied channels is 18.015 MHz including the DC sub-carrier. This means that the 19 detectors can cover all standard bandwidths as specified in LTE-(A).

The resulting graph for each detector has multiple outputs and for each of the B branches a mean square error loss value is calculated as

$$L_b = \frac{1}{2S} \sum_{n=1}^{2S} (\widehat{\mathbf{x}}_\mathbf{b}(n) - \mathbf{x}[b \times 2S + n])^2, \tag{10}$$

$$\widehat{\mathbf{x}}_\mathbf{b}(n) = \widehat{\mathbf{x}}[b \times 2S + n], \tag{11}$$

for $b \in \{1, ..., B\}$, and where $\widehat{\mathbf{x}}$ is the estimated symbol value (i.e. the neural network output). The by the transmitter sent symbol value is $\mathbf{x}$ (i.e. the training label). The loss functions are summed to calculate the total loss as used in the optimization algorithm as

$$L_{total} = \sum_{b=1}^{B} L_b. \tag{12}$$

In our experiments, we train only one detector for 64 SC for simplicity, however, the proposed hardware accelerator is capable of accelerating multiple detectors with different trained weights and activations. Results on performance and memory reduction are presented in Subsection 6.1.

3.3 Quantization

Changing the floating-point model parameters to fixed point data types is often referred to as model quantization. We use the abbreviation BW to mean the bit-width for weights and BA to mean the bit-width of activations. Two approaches for quantization are common. Firstly, post-training quantization takes the parameters after training and applies quantization to it. This often affects the

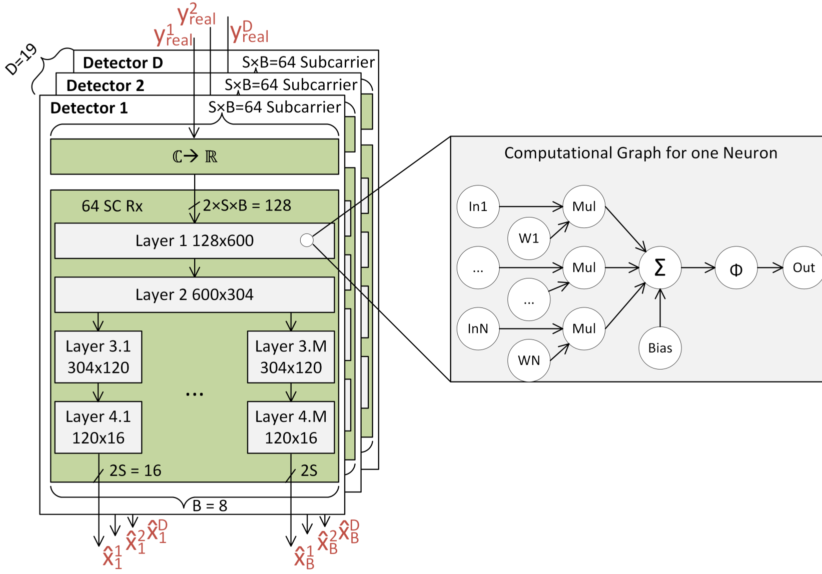


Fig. 2. Neural network receiver structure consisting of D detectors, each consisting of a branching neural network. On the right the computational graph of one neuron is highlighted.

performance of the network, especially when quantizing to low bit-widths. Secondly, quantization aware training applies quantization during the training process. Our training approach is based on [26] where each parameter is still saved as a floating-point number during training. In the forward-pass, a quantized version of the parameters is used to calculate the loss. In the back-propagation step, the floating-point parameters are updated. The floating-point representation of the parameters is required to allow small updates during training. Quantization results are presented in Subsection 6.1

3.4 Pruning

Pruning seeks to reduce the number of total weights in a neural network by considering only weights with large magnitudes, as weights with small magnitudes don't contribute much to the output value of a neuron. In our setup, we use the pruning framework provided by google-research [1]. During the building of the computational graph additional masking nodes are added, which mask out selected weights in the forward-pass during training. The pruning framework accepts a target sparsity parameter which sets a percentage of weights to be masked. In other words, the pruning is not dependent on an absolute threshold value, but rather removes the specified percentage of weights with the lowest magnitudes. The masking can be delayed by an arbitrary number of global training steps. This is useful as first a high accuracy neural network can be trained before gradually increasing the sparsity until the target sparsity is reached. Using a gradual increase of sparsity makes converging to a good solution more likely. Pruning results are presented in Subsection 6.1.

4 ACCELERATOR DESIGN CONSIDERATIONS

A well-established model to classify neural network accelerators is the roofline model [51]. It allows classifying the limiting factor in the performance of a design as either memory limited or computationally limited. The proposed neural network exclusivity consists of fully connected layers and the number of weights is large. This leads to a memory limited design. Many considerations in the presented architecture take the memory limitation into account and are measures to mitigate the effect of a memory limited accelerator. The presented architecture is specifically tuned to the task of accelerating the above-introduced OFDM detector neural network, which is also the reason for designing a neural network accelerator instead of relying on a typical abstraction framework (e.g. [24]). The accelerator distinguishes itself by the following main features:

(1) Each weight memory interface transfers multiple Huffman encoded data-streams in parallel from memory. The bit-width of the memory interface is shared among multiple weight-streams and decoupled via First-In, First-Out storage structures (FIFOs), such that each Huffman decoder can work on a small number of bits, reducing the performance requirement per Huffman decoder. Details on the memory interface and the Huffman decoders are described in Subsection 4.1.
(2) To make the most efficient use of any memory-transferred weights, each weight is used multiple times on different OFDM symbols. First, multiple OFDM symbols are collected, before they are processed in small batches. Batch processing is a known technique to increase performance in certain inference accelerators [41, 50]. We see the application in communication systems as a good target for batch processing, as it allows a trade-off between latency and throughput with a given memory performance. Details are described in Subsection 4.2
(3) The computational graph of the proposed OFDM-detector branches after the second fully connected layer. Switching between two intermediate result buffers in the accelerator allows efficient storing of intermediate results while processing the remaining layers. Section 5.1 expands on this.

The above considerations lead to the following loop-unrolling strategy. The batch level loop in Algorithm 1 is unrolled to the extent that buffering and batch processing of symbols is required. We call this unrolling factor *PAR_S*. As the second unrolled dimension we chose the neuron loop, such that multiple neuron outputs can be calculated in parallel. This is advantageous as the available memory bandwidth can be easily split up into multiple parallel weight streams and as a single DSP block in the FPGA can be used as a Multiply and ACcumulate (MAC) unit. This parameter is named *PAR_N*. As the final unrolled dimension we chose the innermost loop of input weights. This is to reduce the number of required input FIFOs and Huffman decoders, as one chain of FIFOs and Huffman Decoders can provide up to two weights per clock cycle. This unrolling parameter is named *PAR_I*.

To showcase the proposed real-time neural network detector architecture we chose a cutting edge FPGA-platform, namely Alpha Data's RFSoC ADM-XRC-9R1 [33]. The platform provides the Xilinx Zynq Ultrascale+ XCZU27DR-2 FPGA and two 8Gb DDR4-2400 SDRAM memory chips connected to the FPGA fabric [23]. Each memory is capable of providing a theoretical throughput of 64 bits of data at a rate of 300MHz. On top of that, the FPGA provides 22.5 Mbit of on-chip Ultra Ram which we utilize as weight memory for some detectors. BRAM resources are used for various buffers as well as for the storage of bias values.

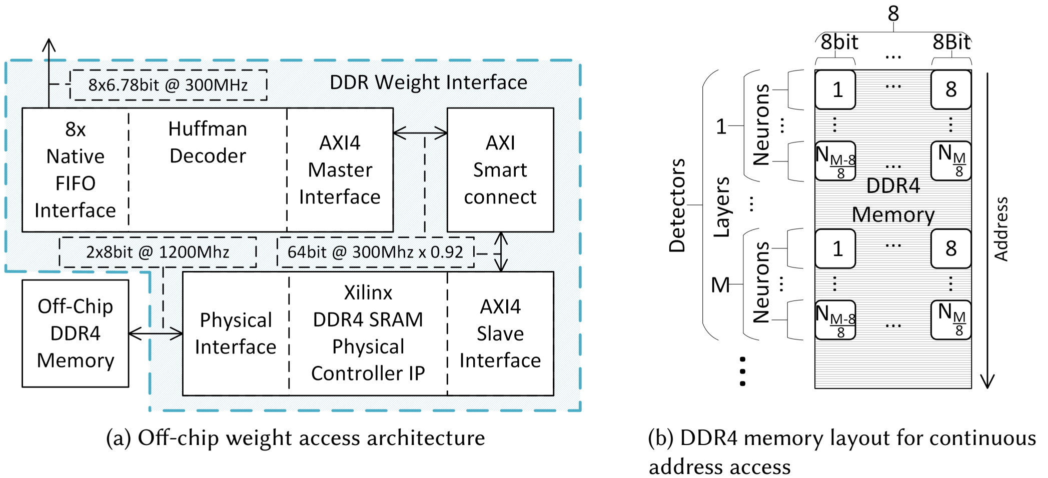
As will be seen in Subsection 6.1, a quantization of 6 bits for the weights and 16 bits for the activations provides a good performance/compression trade-off. Some of the below design considerations assume this quantization, but may also apply to other quantization schemes.

4.1 Memory Interface and Huffman Decoding

As on-chip memory of even large FPGAs is not sufficient to hold all weights for all detectors, off-chip memory access is required. The off-chip memory access is the processing bottleneck in most designs and the memory interface has to be at the center of all design considerations. In the chosen example platform, two state-of-the-art memory chips are available and the design has to match the maximum memory performance of 64 bits of data at a rate of 300MHz per memory chip as close as possible. Utilizing more parallel memory chips or other memory technology, such as HBM or GDDR5, higher performance could be achieved, but would not show the feasibility of a neural network based detector on a standard, off-the-shelf platform.

The physical interface to the DDR4 memory is handled by a Xilinx UltraScale Architecture-Based FPGAs Memory IP [25]. It consists of a physical layer, interfacing the DDR4 memory, a memory controller, and a user interface block. The user interface provides a standard AXI4 slave interface as shown in fig. 3a. The AXI interface allows a maximum of 256 words to be transferred per burst before another addressing phase has to be performed. This limits the effective throughput. The measured throughput is 256 words per 278 clock cycles or 0.92 words per clock cycle.

The weights are ordered in memory as shown in fig 3b. The memory bandwidth of 64 bits is divided into eight 8-bit streams. All weight streams are independent of each other. Each detector's weights are serialized into the memory with a linearly increasing address.



(a) Off-chip weight access architecture

(b) DDR4 memory layout for continuous address access

Fig. 3. Memory layout and access structure for accessing off-chip weights.

Huffman Decoding is a lossless coding scheme that takes the probability of encoded source symbols into account [46]. The more common a source symbol is, the shorter the assigned codes are. This is a good strategy as the typical weight distribution in our detector network follows a normal distribution. Most neural network accelerators for FPGAs do not consider weight compression via Huffman decoders. Han et al. propose an accelerator for deep compression according to [20], however, in the accelerator design Huffman decoding is not considered [19]. There is a research interest in using Huffman decoding for resource-limited devices [39]. More advanced compression methods such as the Deflate-algorithm use Huffman coding as part of the algorithm and high-speed hardware implementations have been proposed [7, 30]. However, we are not aware of hardware

implementations of these compression algorithms specifically for neural network accelerators. Our work implements Huffman coding as we follow the deep compression approach from [20].

The eight bit-wide memory streams are chosen to simplify the Huffman decoder. If the bit-width of the input of the Huffman decoder is much larger than the average length of the codeword and a new input can be provided every clock cycle, then the Huffman decoder has to decode multiple codewords per clock cycle to match the throughput of the input. Since the Huffman decoder is a variable-length code, this would result in high circuit complexity. On the other hand, if the memory word is divided into too many independent streams, many independent Huffman decoders are needed. Each of which needs input and output FIFOs for compensating throughput fluctuations. As a result, the design would require a large number of FPGA resources. To strike this balance we chose eight Huffman decoders per 64-bit interface. The encoded data is organized as shown in fig.3b and each of the eight streams is encoded/decoded independently. The architecture of the proposed Huffman decoder is similar to the parallel decoder of [9], with the difference that up to two source symbols can be recovered per clock cycle. For simplicity of the hardware, we chose to concatenate all layers per detector, and then Huffman code the resulting streams. This reduces the efficiency of the Huffman coding slightly as the per-decoder weight variance is larger than each single per-layer weight variance. In our trained and quantized network with the above-reported performance, we achieve a compression rate of 0.780 while the average per-layer compression rate lays at 0.746. For 6 bit quantization, this leads to an average length of 4.68 bits per weight.

It is important to match the Huffman decoder's throughput with the memory-, and MAC-throughput to not create throughput bottlenecks. Since each Huffman decoder is provided with 8 bits at $\approx 92\%$ of all clock cycles and an average of 4.68 bits are required to encode one weight, the throughput of the Huffman decoder would be too low if it only decoded one weight per clock cycle. The proposed architecture can recover up to two source symbols per clock cycle and is detailed in fig. 4a. Each decoder has an input and output FIFO to decouple datarate fluctuations. A counter keeps track of the valid symbols in the buffer register. If the counter is too low, then eight new bits are written to the buffer register after the last valid data in the register. Two look-up-based decoders are available, both work fully in parallel, and can decode up to one codeword per clock cycle each. The second decoder is connected to 4 bits below the MSB of the buffer register. It speculates that the first decoder decodes a 4-bit codeword and only validates its results once this is confirmed. The probability of decoding a 4-bit codeword of any Huffman codebook is

$$P_{Dec4bitmatch} = \frac{\sum P_{all4bitcodes}}{\sum P_{allcodes}}. \tag{13}$$

With the fully trained weights of one of our detectors $P_{Dec4bitmatch} = 0.52$. The total average throughput thus is

$$\begin{aligned} H_{thr} &= (P_{Dec1Match} + P_{Dec4bitmatch}) \times AvgCodelen \\ &= (1 + 0.5) \times 4.68 = 7.02 \ [bits/clockcycle]. \end{aligned} \tag{14}$$

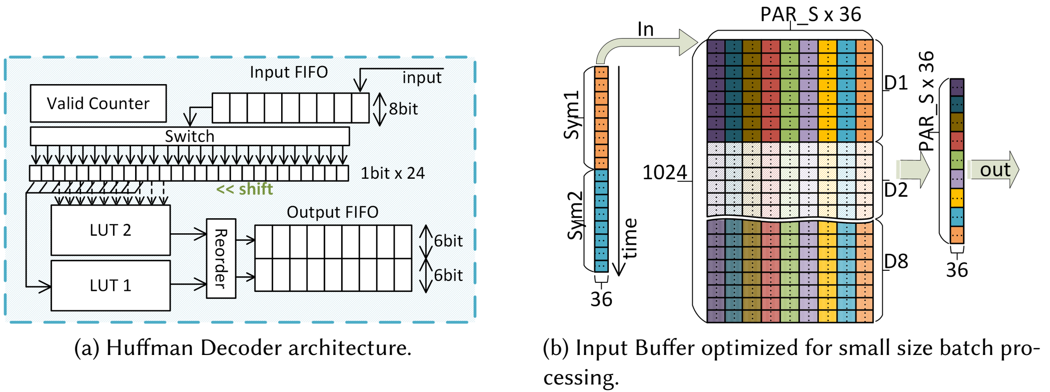
The goal of the Huffman decoder is to match its throughput H_{thr} as closely as possible to the memory interface's throughput. The memory interface throughput is $8\ bit \times 0.92 = 7.36\ [bits/clockcycle]$. This is slightly larger than the Huffman decoder's throughput but is a close enough match for good pipeline performance.

4.2 Small Batch Processing

As the design is memory-bound for off-chip weight access, the throughput of processed OFDM symbols can be increased by using any read weight on multiple OFDM symbols at once. This is termed batch processing and is used in certain accelerator designs for model inference [40, 50]. This

work adds another application to the list where batch processing can be useful. The drawback of batch processing is of course higher detection latency. The work by Posewsky and Ziener proposes a hardware implementation of a batch parallel FPGA accelerator [41]. The accelerator in this work is most noticeable different in that it uses not only batch parallelism, but also other parallelisms as laid out in the above loop unrolling strategy. Also, the Huffman coding, reordering of input symbols and multiple buffers for branching is demanded by the communication workload and distinguishes the proposed accelerator from [41]. The work in [50] focuses on batch processing in CNNs.

The batch processing starts at the symbol-input-buffer which temporarily stores incoming symbols to be processed later in parallel as shown in fig. 4b. The input buffer accepts two input 16-bit words simultaneously and 128 input words are written per detector and per OFDM symbol consecutively. For the DRAM accelerator, the input buffer is larger, as it can hold up to nine OFDM symbols for up to eight different detectors in memory. As the URAM accelerators are working on one neural network only, a much smaller input buffer is required, which can hold up to 2 symbols. All input buffers address the BRAM storage in such a way that previously stored OFDM symbols can be processed by the accelerator in parallel per detector. Writing and reading to the input buffer are independent, such that while data of one detector is read for processing, the data of another detector is written to the buffer to be processed next. As the Huffman decoder provides up to two weight values per clock cycle, the input buffer provides two input values per symbol too. The loop unrolling strategy also implies that the inputs have to be streamed to the processing unit multiple times, which is controlled by an input to the buffer.



(a) Huffman Decoder architecture.

(b) Input Buffer optimized for small size batch processing.

Fig. 4. Huffman Decoder and Input Buffer Architecture.

4.3 Intermediate result buffer with branching support

The proposed accelerator processes each layer consecutively, i.e. no layer-level parallelism is supported. This is a common strategy but requires the buffering of one layer's results such that they can be used in the next layer. In our application, the layer results are small enough to be stored in an on-chip BRAM buffer - the so-called intermediate result buffer. It consists of two BRAM blocks that can support the size of up to 1024 Neurons for up to *PAR_S* parallel symbols. The Accelerator controller chooses which buffers to use in accordance with Table 1. The two buffers support branching in the computational graph of the neural network model without performance degradation. The result of the last common layer, layer 2 in this case, is stored in Buffer 2. Buffer 2 is not overwritten until a new detector is calculated. The result in Buffer 2 is used in all calculations

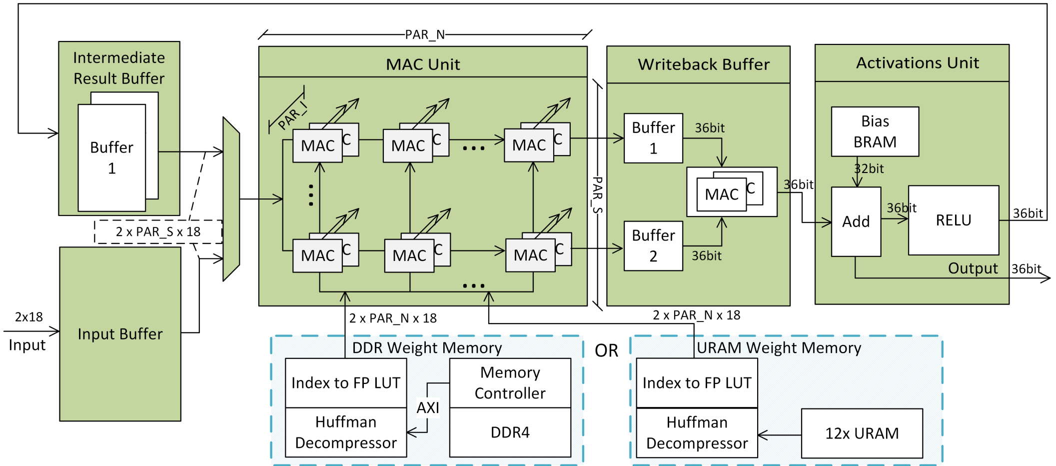


Fig. 5. Proposed Neural Network Accelerator with Batch processing support, Huffman decoders and multiple intermediate result buffers for branching support

of layer 3. If branching was not supported by the intermediate result buffer, costly off-chip memory writes would degrade the detector performance, as each layer would overwrite the layer 2 results.

Table 1. Buffer usage for the branching computational graph

Layer	1	2	3.1	4.1	..	3.8	4.8
Source Buffer	Input	B1	B2	B1	..	B2	B1
Target Buffer	B1	B2	B1	Out	..	B1	Out

5 REAL-TIME ACCELERATOR DESIGN

By taking the above design considerations into account, we propose a novel neural network accelerator specifically for the workload in communications. The accelerator has two versions. One version holds the weight data in off-chip DDR4 memory banks. This version is ultimately limited by the memory throughput. The second version holds the weights in on-chip Ultra RAM (URAM) buffers. The available memory bandwidth from URAM to DSP blocks is large (up to 72 bits per clock cycle per URAM). The design is ultimately limited by the number of parallel URAM blocks per detector. Apart from the weight-access, the biggest difference between the versions of the accelerator is the setting of PAR_N and PAR_S. The proposed accelerator architecture is depicted in fig. 5.

As will be seen in Subsection 6.1, pruning does not provide a major performance benefit as for a reasonable pruning rate the detector performance is degraded heavily. For this reason, and in order not to increase the hardware complexity further, pruning is not considered in the accelerator design.

5.1 Architecture for off-chip weights

In the DRAM accelerator $PAR_N = 8$ and $PAR_S = 9$. The input buffer provides access to collected symbols for the multiply and accumulate (MAC)-unit. With $PAR_S = 9$, 18 input-values of 9 different symbols are provided to the MAC unit per clock cycle. Similarly, the Huffman lookup

tables provide 16 weights of 8 weight streams per clock cycle. As the weights are reused on each symbol it results in 288 MAC calculations per clock cycle. Once the calculations for one set of output neurons are completed in the MAC unit, the 288 results are transferred to the writeback buffer in one clock cycle.

The Huffman decoder provides weight indices from 0 to 63 to the accelerator for the 6-bit quantization, which strikes a good trade-off between accuracy and quantization as shown in Section 6. These indices are used to lookup an 18-bit fixed point weight value. Per Huffman decoder, one dual-port index translation ROM is instantiated which can provide separate lookup values for each layer of the accelerator. As each layer is quantized separately during the training phase, it is important for performance to also follow this quantization strategy in hardware.

Once the MAC results have been written in the writeback buffer, the 128 values of the two buffers are added one per clock cycle to form 8 neuron values for 9 symbols. The writeback of data through the activation unit and into the intermediate result buffer can be executed in series as the layers of the neural network are large enough to not require the last output data of the previous layer immediately. On the pass through the activation unit, a bias value is added to each result. This bias value is stored in a dual-port ROM as an 18-bit fixed point value. For each layer and each detector, different bias values are stored. All our layers use the RELU activation function, except the output layers. The RELU function is easy to implement in hardware. The Sigmoid activation function of the output layer is not implemented in the accelerator, as its main function is to produce a predictive value between zero and one. It can be implemented in successive hardware or as software function as the timing of it is not critical.

5.2 Architecture for on-chip weights

The URAM detector architecture is similar to the DRAM one in that it uses the same Huffman decoders, the same principle of intermediate result and writeback buffering, and the same activations unit. Apart from using on-chip URAM weight storage, one of the main differences is that $PAR_N = 18$ and $PAR_S = 2$. Because of the size limitation in the URAM, each accelerator only accelerates one detector network. As the processing latency is low, $PAR_S = 2$ is sufficient to process the detector network in real-time. For best usage of the URAM resources, we chose the URAM memory data-with as 144 bits, utilizing two parallel URAM blocks. The 144 bits translate into 18 Huffman decoder modules, each receiving 8 bits of the data. $PAR_N = 18$ follows, and a total of 36 MAC units are utilized. The writeback, activation, and intermediate result buffer are similar to the DRAM version of the accelerator except for adaptations to the differing parameters PAR_N and PAR_S.

6 RESULTS AND PERFORMANCE

This section reports the results of the previous design steps from detection performance in terms of Bit Error Ratio (BER) to real-time capability and accelerator circuit metrics. Table 2 summarizes the key parameters used to generate the presented results.

6.1 Detection Performance

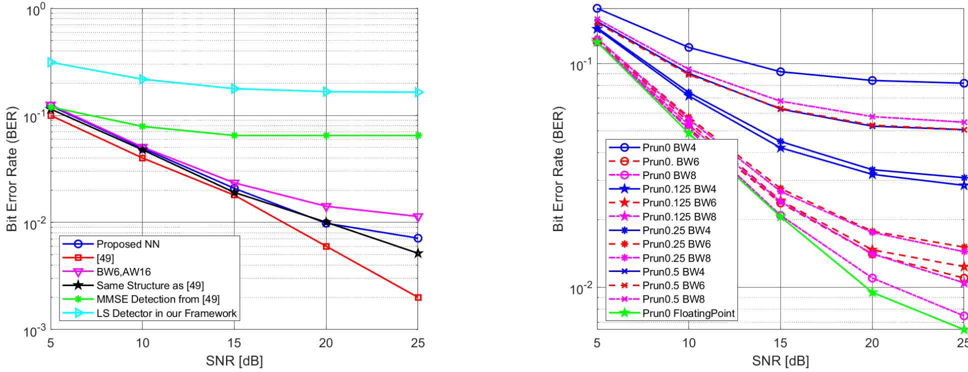
The training is performed across the range of SNR values from 5dB to 25dB. In our experiments, we did not notice any significant performance improvement when training separately for each SNR value. For this reason, SNR values are picked randomly from the discrete set of SNR values in the training range, such that the receiver is trained for all SNR values in the range simultaneously. In fig. 6a the comparison with [49] shows that in our training, the performance of the same neural network structure leads to a 2 dB SNR loss at a BER of 10^{-2}. The source of this discrepancy is not clear to us. It might have to do with extensive training times, as in [49] 20000 epochs are trained, whereas in our work a maximum of 2000 epochs are trained for practical reasons. Although our

Table 2. Key Setup Parameters

Channel Model	WINNER II channel model as in [49]
SNR Range	5-25 dB
Pilot- to Data- Symbol Ratio	1/8
Number of training channel realizations	3 Million
Number of testing channel realizations	1 Million
Modulation Type	QPSK
Cyclic Prefix length	16 Samples
Number of OFDM SC per Detector	64
Number of Layers	4
Number of Neurons	566,400
FPGA Type	Xilinx xczu27dr-ffve1156-2-i
Total Memory Throughput	37.2 GB/s
Clock Frequency	300 MHz

training efforts could not reach the performance of [49], it still outperforms traditional approaches by a large amount.

Another small performance degradation can be observed in fig. 6a by comparing the proposed reduced complexity network with the neural network structure from [49] which was trained in our framework.



(a) Detection performance comparison for various neural networks for the range of 0dB to 25 dB.

(b) Detection performance loss due to simultaneous pruning and quantization.

Fig. 6. Performance for quantized and prunned neural network detectors.

Quantization with a weight bit-width of 6 bits and activation bit-width of 16 bits leads to another small performance degradation at high SNR values. We investigate the effects of quantization to the network in depth and define the accuracy-loss with respect to the floating-point BER for each configuration as

$$QErr = \frac{BER_{Quant}}{BER_{FP}} - 1. \tag{15}$$

In table 2a the *QErr* when the detector is trained in a signal to noise ratio (SNR) range from 5 to 25 dB is provided. In table 2b the *QErr* at 25dB is shown. It can be seen that for higher SNR values the impact of quantization is larger.

(a) Average performance degradation *QErr* from eq. (15) in percent, when training the detector with SNR values from 5 to 25dB.

			Bitwidth Weights [bits]				
			4	5	6	7	8
Quantization Aware Training	Bitwidth Activations [bits]	6	4057%	2383%	2153%	2001%	297%
		8	1610%	505%	297%	226%	223%
		10	1167%	259%	85%	68%	52%
		16	1044%	193%	54%	10%	5%
Post Training Quantization	Bitwidth Activations [bits]	6	4749%	3111%	2111%	2085%	2740%
		8	3052%	1131%	524%	318%	306%
		10	2516%	751%	229%	85%	66%
		16	2277%	558%	112%	19%	-8%

(b) Average performance degradation *QErr* from eq. (15) in percent, when training the detector at 25dB SNR.

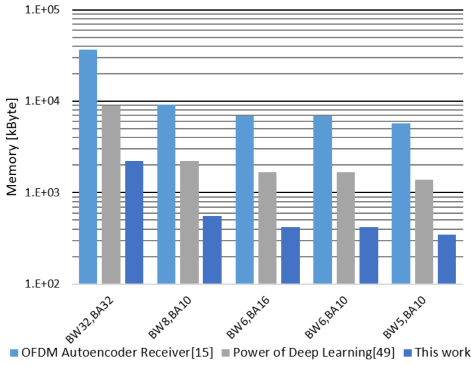
			Bitwidth Weights [bits]				
			4	5	6	7	8
Quantization Aware Training	Bitwidth Activations [bits]	6	657.0%	422.0%	388.0%	373.8%	67.1%
		8	263.5%	98.8%	67.1%	58.7%	59.8%
		10	186.4%	46.3%	18.7%	14.7%	12.9%
		16	163.0%	33.6%	7.2%	1.7%	2.4%
Post Training Quantization	Bitwidth Activations [bits]	6	768.3%	489.1%	398.1%	368.7%	487.0%
		8	479.5%	192.0%	99.7%	75.7%	71.4%
		10	390.1%	121.8%	38.5%	20.0%	16.2%
		16	345.3%	73.5%	9.7%	6.3%	0.3%

Table 3. Impact of quantization on performance.

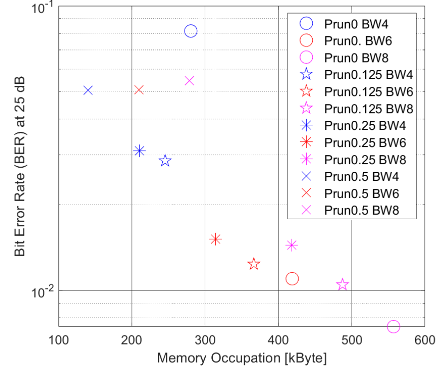
Fig. 6b shows the detection performance when pruning is introduced on top of quantization. The pruning rates of 12.5%, 25%, and 50% are compared to the unpruned, quantized, and floating point networks. For pruned networks, quantization is started after 500 epochs and pruning after 750 epochs. The total number of trained epochs is 2000. For 8 bit weight quantization, pruning of 12.5% leads to a performance loss of $\approx$ 5 dB at 25 dB. This is similar to the performance loss when quantizing weights with 6 bits. The network with 6 bit quantized weights suffers from performance loss of $\approx$ 2.5 dB and $\approx$ 6 dB by pruning it to 12.5% and 25% respectively. The memory requirement for the configuration of Prun0.125BW8 is $\approx$ 16% higher than for Prun0BW6 as seen in fig. 7b. Because of this and the fact that pruning adds additional hardware complexity, pruning is not considered in the accelerator design.

6.2 Memory Requirement

As a baseline for memory-requirement comparison, we assume single-precision floating-point parameters with 32 bits per parameter according to [3]. For comparison we include the autoencoder from [13]. In this design, each sub-carrier requires one autoencoder, making it the most complex design. Only the receiver network of the autoencoder is considered. Fig. 7a shows significant savings both from reduced complexity models as well as from quantization. The novel model results in a memory reduction of $\approx$ 4.02 times and the quantization with BW=6 and BA=16 results in another memory reduction of $\approx$ 5.3 times as compared to 32 bit floating-point weights. Huffman decoding further reduces the memory requirement by $\frac{1}{0.780} \approx 1.28$, leading to a total memory reduction of $4.02 \times 5.30 \times 1.28 \approx 27.27$ times.



(a) Memory requirement to process 64 sub-carriers in comparison to previous work in the literature.



(b) Memory requirement and detection performance of various quantization and pruning options. All activation weights are assumed to be 16 bit except floating point activations are 32 bits.

Fig. 7. Memory requirement to process 64 SC for various combinations of quantization and pruning settings.

6.3 Scheduling and Latency

The performance of the presented accelerator is evaluated on the Xilinx Zynq Ultrascale+ XCZU27DR-2 FPGA for up to 1200 SC. We evaluate the performance for multiple numbers of SC commonly used within standardized LTE numerology [43]. The largest configuration of 1200 SC provides a 20MHz channel bandwidth at a sub-carrier spacing of 15kHz. One of the main performance metrics of the accelerator is the detection latency in symbol durations. The measurement in symbol durations is useful as it makes the latency independent of the underlying OFDM numerology. With 15kHz, the symbol duration is $T_S = \frac{1}{15kHz} \approx 66.7\mu s$.

As described in Section 5.1 and 5.2, the accelerators allow for buffering and simultaneous processing of multiple symbols. While one detector is processed, all detectors including the currently processed one, buffer the newly received symbols. This batch processing has an impact on system latency. The number of detectors is the main factor for the maximum latency besides the processing time:

$$L_{DDR_max} = \lceil N_{DDR_det} \times T_{DDR_proc} \rceil + T_{DDR_proc} \quad (16)$$

N_{DDR_det} is the number of parallel detectors per accelerator. T_{DDR_proc} is the processing time for each detector. The ceiling function indicates that processing cannot start while a symbol is not completely received yet. The average detection latency for each of the DRAM accelerators is:

$$L_{DDR_avg} = \frac{\sum_{k=1}^{n} k}{n} + T_{DDR_proc} = \frac{n+1}{2} + T_{DDR_proc}\ ; with\ n = \lceil N_{DDR_det} \times T_{DDR_proc} \rceil. \quad (17)$$

Leading to a combined detection latency of:

$$L_{avg} = \frac{N_{DDR_det} \times L_{DDR_avg}}{N_{det}} + \frac{N_{URAM_det} \times L_{URAM_avg}}{N_{det}}. \quad (18)$$

As each URAM accelerator processes only one detector ($N_{det} = 1$), L_{URAM_avg} is equal to T_{URAM_proc}. Because each accelerator processes one or more detectors which are assigned during design time, each detector's throughput has to be

$$TP_k = \frac{PAR_S_k}{T^k_{proc} \times N^k_{det}} > 1. \tag{19}$$

Where k is the detector index. If the throughput falls below the value of 1, symbols will build up at the input and the processing latency will continuously increase. Since the physical input buffers have limited capacity it eventually will result in data loss, which violates the real-time condition. As the assignment of detectors to accelerators is fixed, a build-up of symbols at one or more detectors might even happen if the total system throughput satisfies the following condition:

$$TP = \sum_{k=1}^{N_{acc}} TP_k > 1. \tag{20}$$

Table 4 shows the accelerator allocation for maximum performance for different numbers of processed SC. First URAM accelerators are allocated to maximize performance. The latency is given according to eqs. (16) to (18). The throughput per detector and the total throughput in terms of symbols according to eqs. (19) and (20) are shown respectively. Up to 13 detectors can be deployed while the real-time condition still holds. The average throughput per detector is 1.4 symbols per T_S and all symbols can be processed in time. A maximum symbol latency of 11.52 symbols, and an average latency of 3.67 symbols results. The 13 detectors amount to a total of 832 QPSK modulated SC allowing a bitrate of $832 \times 15kHz \times 2 = 24.96\frac{Mbits}{s}$ including pilot symbols. For 900 SC, the throughput in the DRAM accelerators is too small in our configuration, although the average throughput per detector would be sufficient as indicated by the coloring of the last row in table 4. For 1200 SC, each of the DRAM accelerator's throughput, as well as the average throughput, is too low as indicated by the red coloring.

6.4 Circuit Metrics

Each DRAM accelerator is clocked at 300 MHz. This clock is provided by the physical DDR interface IP and is dependent on the connected memory chips. The memory chip is clocked at 1200 MHz and transfers 8 bit of data per positive and negative clock-edge. This translates into a bus with 64 bits data-width at a transfer rate of 300 MHz (single clock-edge). The URAM accelerator's maximum frequency is 300 MHz as well. The critical paths are manifold, with the most noticeable ones being in the accelerator's control logic and in the flexible shift mechanic of the Huffman decoder's buffer register.

Table 5 shows the average resource usage per single accelerator and fig. 8 shows the average allocation of resources per submodule per accelerator. We notice that for the DRAM accelerator, more than half of all CLB resources and slightly less than half the BRAM resources are used by the physical memory interfaces. The DRAM accelerator uses more resources, but can also process multiple detectors.

6.5 Accelerator Efficiency Evaluation

To assess the implementation efficiency of the proposed accelerator, we compare the measured performance in terms of operations per second with its theoretical maximum performance. This theoretical maximum performance assumes that all computational units perform useful computations every clock cycle and that the available memory bandwidth transports data continuously at its maximum capacity. The measure of operations per second is adequate to assess performance as it

Table 4. Accelerator performance in terms of processing latency, throughput and real-time capability for different number of SC. Latency is calculated according to eqs. (16) to (18) and throughput according to eqs. (19) and (20). Green fields indicate sufficiently large throughput for real-time processing according to eq. (19).

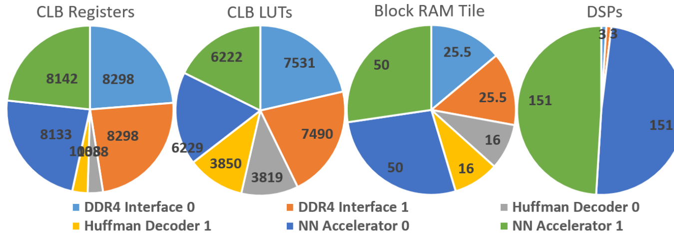
Bandwidth[MHz]	1.4	3	5	10	11.52	15	20
Number of SC	72	180	300	600	832	900	1200
Required Accelerators (Number of SC / 64)	1.125	2.8125	4.6875	9.375	13	14.0625	18.75
URAM Accelerator Processing Latency [Sym]	1.087						
URAM Accelerator Throughput [Sym/TS]	1.840						
DRAM Accelerator Processing Latency [Sym]	2.519						
Allocated URAM accelerators	2	3	5	6	6	6	6
Allocated DRAM accelerators	0	0	0	2	2	2	2
Detectors per DRAM accelerator1/accelerator2	0/0	0/0	0/0	3/1	3/3	5/4	7/6
Maximum DRAM accelerator latency [Sym]	-	-	-	11.52	11.52	17.52	23.52
DRAM accelerator 1 throughput [Sym/TS]	-	-	-	1.19	1.19	0.71	0.51
DRAM accelerator 2 throughput [Sym/TS]	-	-	-	3.57	1.19	0.89	0.60
Average DRAM accelerator latency [Sym]	-	-	-	4.76	6.26	8.51	11.51
Average symbol latency [Sym]	1.09	1.09	1.09	2.56	3.67	5.54	8.22
Maximum throughput [Sym/TS]	4.35	6.52	10.87	18.19	18.19	18.19	18.19
Average throughput per detector [Sym/TS]	2.17	2.17	2.17	1.82	1.40	1.21	0.96

Table 5. Average FPGA resource utilization for each accelerator version. Non-integer number of resources result from averaging the resources over multiple accelerators.

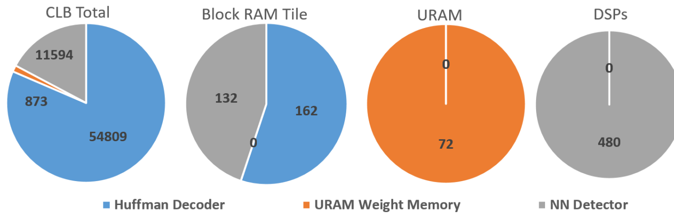
	CLB LUTs	CLB Registers	Block RAM	URAM Blocks	DSPs
DRAM Accelerator	17.5k	17.5k	91.5	0.00	154
URAM Accelerator	8k	3.7k	49	12.00	80.7

directly relates to the throughput of an accelerator for a given neural network. Table 6 lays out the maximum achievable performance given the DSP utilization and the available memory bandwidth. It can be seen that both accelerators are memory-bound, as expected, for fully connected network accelerators. The measured detector performance was obtained by digital simulation and was found to be 9.6% and 15.3% lower than the theoretical maximum performance of the DRAM and URAM accelerator respectively. This inefficiency is mostly due to the varying length of the weight streams after Huffman coding. Some streams can have regions where the decoders can only decode one datum per clock which leads to a stall of the processing elements. The benefit of the Huffman compression still outweighs this inefficiency though.

Table 7 compares our work with previously proposed neural network accelerators. It is important to notice that the latency and throughput figures have been extracted for fully connected layers from the previous work for a fair comparison. The utilization figures have been extracted where possible. From the comparison, it can be noticed that even though our DRAM accelerator has much smaller memory bandwidth available, its throughput is considerably larger than the throughput in [34] and [42]. This can be attributed to a few factors: the weight quantization to 6 bit, the Huffman compression, and the batch-processing, to which we attribute the highest performance gain. In



(a) Resource utilization for two DRAM accelerators



(b) Resource utilization for six URAM accelerators

Fig. 8. Resource usage per sub module per type of accelerator for two DRAM accelerators and six URAM accelerators.

Table 6. Theoretical and measured accelerator performance

Accelerator Performance	DRAM	URAM
Measured Detector Performance without Activations		
Number of weights in detector network	566400	566400
Processed symbols per second [Sym/s]	15000	15000
Accelerator throughput [Sym/Ts]	1.19	1.84
Detectors per accelerator	3	1
Performance [GOP/s]	30.36	15.64
Theoretical Maximum Performance based on DSP Utilization		
Accelerator frequency	3E+08	3E+08
Number of DSPs per Accelerator	154	80.67
Theoretical maximum performance based on the number of DSP [GOP/s]	46.2	24.20
Theoretical Maximum Performance based on Memory Bandwidth		
Buswidth [Bytes]	8	18
Busspeed [MHz]	300	300
Memory throughput [MB/s]	2400	5400
AXI-Interface throughput reduction factor	0.91	-
Average number of bits per weight [bits]	4.68	4.68
Average number of weights processed per second [MWeights/s]	3733.33	9230.8
Maximum batch size	9	2
Theoretical maximum performance based on memory bandwidth [GOP/s]	33.60	18.46

Table 7. Accelerator performance comparison

	[34]	[42]	[10]	This Work	
				One DRAM Accelerator	All Accelerators
Model	Alexnet	VGG16-SVD	Alexnet	OFDM Detector	
Quantization	fixed 8 bit	dynamic 16 bit	fixed 16 bit	fixed 6 bit	
FPGA	Stratix-V GXA7	Zynq XC7Z045	Arria 10	RFSoC (Ultrascale+)	
Feature Node	28nm	28nm	20nm	16nm	
Clock Frequency[MHz]	100	150	300	300	
External Memory	2 DDR3 x64	1 DDR3 x64	1 DDR4 x64	1 DDR4 x8	2 DDR4 x8 + 6 URAM x18
Throughput					
Memory Bandwidth [GB/s][a]	12.8-29.87[b]	4.2	17	2.40	37.2
Weights in the FC Layers	59M	73M	59M	566k	7.4M
Average Latency FC Layers [ms]	2.83	61.18	125.5[c]	0.0187	0.24
FC Layer Throughput [GOP/s]	20.72	1.20	1382.00	30.33	154.56
Utilization					
DSP Utilization	256	780	1476	154	788
Logic Utilization[d]	15.48K[e]	310.27K	246K	35.04K	139.88K
BRAM/URAM Utilization	213/-	486/-	2487/-	91.5/-	477/72

[a] 1GB $\hat{=}$ 1e9B

[b] Not clearly specified, estimation based on DDR3 specification

[c] Processes the FC layers in batches of 128 after calculating all preceding layers first. As the FC layers are at the end of the network we can get a rough estimate based on the average image latency ($\frac{1}{1020\ \text{img/s}} = 0.98$ ms/img) multiplied by the batch size of 128.

[d] Xilinx FPGAs in Total CLBs (Registers + LUTs) and Altera FPGAs in ALMs

[e] Resources directly associated to FC6&FC7&FC8 Layers

addition, also the older technology nodes of 28 nm and the lower operating frequencies in [34] and [42] are contributing factors. The design in [10] makes use of batch processing for its FC layers and uses a batch size of 128. We estimate the latency for the Alexnet calculation in [10] to be 125.5 ms as compared to the latency for the same neural network in [34] of 12.75 ms. This shows how a large batch size like this trades off throughput for latency. Our design achieves ≈ 9 times lower throughput than [10] with ≈ 2 times the total memory bandwidth. However, it's batch size is ≈ 14 times and 64 times smaller for the DDR- and URAM- accelerator respectively.

6.6 Comparison with Traditional Approaches

6.6.1 Computational Resources. The neural network based detector outperforms traditional approaches significantly as seen in fig. 6a. This gain in performance comes at the expense of computational complexity. For example, a modified Minimum Mean Square Error (MMSE) FPGA implementation is reported in [14]. Timing is not reported in detail, but resource utilization of LUTs, Registers, and DSP blocks is very low. The memory requirement with ≈ 150 kilo bytes is within the same order of magnitude as what is needed in our design to process 64 SC. The accelerator in [14] however, processes 840 SC. The work in [11] presents a linear MMSE detector that shows reduced complexity by low-rank-approximating via singular value decomposition. Assuming a rank of 6, and 8 pilots per 64 SC, the number of complex multiplications for estimating the channel transfer function is reported as 105. This is much lower than the required number of operations in our neural network. The channel estimator and OFDM equalizer provided in Mathworks' Wireless HDL toolbox implement basic least square channel estimation and zero-forcing or minimum mean

square error equalization [27, 28]. Both designs achieve a clock frequency of 244.6 MHz on the Xilinx Zynq- 7000 ZC706 evaluation board. Their hardware resource requirements are shown in table 8. The resource usage in terms of LUTs is comparable to the URAM accelerator, while DSP and memory resources are far lower.

Table 8. Resource usage in Mathworks' OFDM channel estimator and equalizer

Resource	Channel Estimator [28]	Equalizer [27]	Total [28] + [27]	URAM Accelerator
Slice LUTs	2684	7380	10064	7976
Slice Registers	1184	8063	9247	3656
DSPs	6	24	30	80
Block RAMs	1.5	0	1.5	49
URAMs	-	-	-	12

6.6.2 Latency. The LTE-A user-plane latency is typically specified to be less than 10ms [43, Chapter 27]. However the processing time is specified by 3GPP as 1.5ms for UEs [5]. Comparing this time with the maximum latency for the DRAM accelerator of 11.52 $symbols \mathrel{\hat{=}} 0.77\ ms$ and the URAM accelerator of 2.52 $symbols \mathrel{\hat{=}} 0.17\ ms$, this seems possible. However, a typical OFDM receiver consists of more than just channel estimation and detection circuits. The latency of the traditional approaches in [27, 28] depends on the detailed setting of the block, but is within the range of a few clock cycles up to 951 clock cycles for more advanced settings. This shows, that in comparison to the neural network based accelerator, there is a large gap in latency. Whether or not this latency is acceptable will be determined by the specific design and use case.

7 CONCLUSION

This paper proposes the first neural network detector for an OFDM communication system with real-time capability. The proposed detector utilizes a simplified structure of a previously reported neural network performing efficient channel estimation and detection. Three main techniques of deep compression were explored, namely pruning, quantization, and weight encoding. The network complexity could be reduced this way, however, reduced performance was observed when pruning and quantization are applied simultaneously. As memory throughput and memory size have been limiting factors, our efforts achieved a memory requirement reduction of ≈ 27 times. A neural network accelerator specifically designed to accelerate the targeted fully connected neural network was developed. The accelerator has three features making it uniquely suitable for processing the proposed neural network in real-time. It allows for processing multiple symbols simultaneously in a batch. Loaded weights are used on multiple symbols and throughput is increased. Multiple intermediate result buffers allow for efficient branching in the neural network. High-performance Huffman decoders with a decoding rate larger than 1 allow efficient usage of weight buffering FIFOs. Our design can be expanded by instantiating more accelerators, however, in our platform containing the Xilinx RFSoC, we achieve real-time processing for 832 sub-carriers with an average detection latency of ≈ 3.7 OFDM symbols. The thereby enabled datarate including pilots is just under 25 Mbit/s.

We consider the presented work an important step towards deployable, neural network based detector solutions in communication systems. Even though the application of compression techniques reduced the memory requirement significantly, the computational complexity and memory requirements is still far beyond that of traditional approaches. The improved performance can

justify this much larger complexity to some extend, however we also recognize the need for further research in the area. The computational complexity and memory requirements need to be reduced further. Techniques that combine well-established communication algorithms with deep learning techniques (i.e. model-driven designs) could help with this. Also in terms of flexibility, improvements are needed. Frame structures as specified in mobile communication standards and higher modulation orders such as QAM64 need to be supported. Form an FPGA perspective, the trend to highly integrated high bandwidth memory (HBM) and larger on-chip memories can mitigate the current memory limitations.

ACKNOWLEDGMENTS

This work is supported by the Engineering and Physical Sciences Research Council of the United Kingdom and Alpha Data Parallel Systems Ltd. under Grant No.: EPS513799/1.

REFERENCES

[1] [n.d.]. google-research/google-research. https://github.com/google-research/google-research Library Catalog: github.com.

[2] 2018. Unsupervised Deep Learning for MU-SIMO Joint Transmitter and Noncoherent Receiver Design. *IEEE Wireless Communications Letters* (2018), 1–1. https://doi.org/10.1109/LWC.2018.2865563

[3] 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (July 2019), 1–84. https://doi.org/10.1109/IEEESTD.2019.8766229 Conference Name: IEEE Std 754-2019 (Revision of IEEE 754-2008).

[4] 3GPP. 2019. *3GPP TR 21.915*. Technical Report 3GPP TR 21.915 V15.0.0 (2019-09). https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3389

[5] 3GPP. 2020. *3GPP TR 36.912*. Technical Report 3GPP TR 36.912 V16.0.0 (2020-07-14). https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=2584

[6] 3GPP. 2021. *3GPP TR 36.211*. Technical Report 3GPP TR 36.211 V16.6.0 (2021-06-30). https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=2425

[7] Mohamed S. Abdelfattah, Andrei Hagiescu, and Deshanand Singh. 2014. Gzip on a chip: high performance lossless data compression on FPGAs using OpenCL. In *Proceedings of the International Workshop on OpenCL 2013 & 2014 - IWOCL '14*. ACM Press, Bristol, United Kingdom, 1–9. https://doi.org/10.1145/2664666.2664670

[8] Charu C. Aggarwal. 2018. *Neural Networks and Deep Learning: A Textbook*. Springer International Publishing, Cham. https://doi.org/10.1007/978-3-319-94463-0

[9] Z. Aspar, Z. Mohd Yusof, and I. Suleiman. 2000. Parallel Huffman decoder with an optimized look up table option on FPGA. In *2000 TENCON Proceedings. Intelligent Systems and Technologies for the New Millennium (Cat. No.00CH37119)*, Vol. 1. 73–76 vol.1. https://doi.org/10.1109/TENCON.2000.893543

[10] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. 2017. An OpenCL Deep Learning Accelerator on Arria 10. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. Association for Computing Machinery, New York, NY, USA, 55–64. https://doi.org/10.1145/3020078.3021738

[11] Tzi-Dar Chiueh and Pei-Yun Tsai. 2007. *OFDM baseband receiver design for wireless communications*. John Wiley and Sons (Asia), Singapore ; Hoboken, NJ. OCLC: ocn137222764.

[12] Balázs Csanád Csáji. 2001. Approximation with Artificial Neural Networks. /paper/Approximation-with-Artificial-Neural-Networks-Cs%C3%A1ji/5b111b750c2c0e55bdb7a7bd78d69648c03fb4a5 Library Catalog: www.semanticscholar.org.

[13] S. Dörner, S. Cammerer, J. Hoydis, and S. t Brink. 2018. Deep Learning Based Communication Over the Air. *IEEE Journal of Selected Topics in Signal Processing* 12, 1 (Feb. 2018), 132–143. https://doi.org/10.1109/JSTSP.2017.2784180

[14] Khaled ElWazeer, Mohamed M. Khairy, Hossam A. H. Fahmy, and S. E. D. Habib. 2009. FPGA implementation of an improved channel estimation algorithm for mobile WiMAX. In *2009 International Conference on Microelectronics - ICM*. 280–283. https://doi.org/10.1109/ICM.2009.5418629 ISSN: 2159-1679.

[15] A. Felix, S. Cammerer, S. Dörner, J. Hoydis, and S. Ten Brink. 2018. OFDM-Autoencoder for End-to-End Learning of Communications Systems. In *2018 IEEE 19th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*. 1–5. https://doi.org/10.1109/SPAWC.2018.8445920

[16] Xuanxuan Gao, Shi Jin, Chao-Kai Wen, and Geoffrey Ye Li. 2018. ComNet: Combination of Deep Learning and Expert Knowledge in OFDM Receivers. *IEEE Communications Letters* 22, 12 (Dec. 2018), 2627–2630. https://doi.org/10.1109/LCOMM.2018.2877965

[17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. The MIT Press, Cambridge, Massachusetts.
[18] Yunhui Guo. 2018. A Survey on Methods and Theories of Quantized Neural Networks. *arXiv:1808.04752 [cs, stat]* (Dec. 2018). http://arxiv.org/abs/1808.04752 arXiv: 1808.04752.
[19] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. *arXiv:1602.01528 [cs]* (Feb. 2016). http://arxiv.org/abs/1602.01528 arXiv: 1602.01528.
[20] Song Han, Huizi Mao, and William J. Dally. 2015. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *arXiv:1510.00149 [cs]* (Oct. 2015). http://arxiv.org/abs/1510.00149 arXiv: 1510.00149.
[21] haoyye. 2020. Github Software https://github.com/haoyye/OFDM_DNN. https://github.com/haoyye/OFDM_DNN original-date: 2017-10-09T02:14:52Z.
[22] Hengtao He, Shi Jin, Chao-Kai Wen, Feifei Gao, Geoffrey Ye Li, and Zongben Xu. 2019. Model-Driven Deep Learning for Physical Layer Communications. *IEEE Wireless Communications* 26, 5 (Oct. 2019), 77–83. https://doi.org/10.1109/MWC.2019.1800447 Conference Name: IEEE Wireless Communications.
[23] Xilinx Inc. 2018. Zynq UltraScale+ RFSoC Data Sheet: Overview.
[24] Xilinx Inc. 2019. DPU for Convolutional Neural Network v3.0, DPU IP Product Guide. (2019), 54.
[25] Xilinx Inc. 2020. UltraScale Architecture-Based FPGAs Memory IP v1.4 LogiCORE IP Product Guide.
[26] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2017. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *arXiv:1712.05877 [cs, stat]* (Dec. 2017). http://arxiv.org/abs/1712.05877 arXiv: 1712.05877.
[27] MathWorks United Kingdom. 2021. Equalize OFDM data using channel estimates. https://uk.mathworks.com/help/wireless-hdl/ref/ofdmequalizer.html
[28] MathWorks United Kingdom. 2021. Estimate channel using input data and reference subcarriers. https://uk.mathworks.com/help/wireless-hdl/ref/ofdmchannelestimator.html
[29] Phillip A. Laplante. 2012. *Real-time systems design and analysis: tools for the practitioner* (4th edition. ed.). Wiley-IEEE Press, Hoboken, N.J. http://www.ezproxy.is.ed.ac.uk/login?url=http://onlinelibrary.wiley.com/book/10.1002/9781118136607
[30] Morgan Ledwon, Bruce F. Cockburn, and Jie Han. 2020. High-Throughput FPGA-Based Hardware Accelerators for Deflate Compression and Decompression Using High-Level Synthesis. *IEEE Access* 8 (2020), 62207–62217. https://doi.org/10.1109/ACCESS.2020.2984191 Conference Name: IEEE Access.
[31] Khaled B. Letaief, Wei Chen, Yuanming Shi, Jun Zhang, and Ying-Jun Angela Zhang. 2019. The Roadmap to 6G: AI Empowered Wireless Networks. *IEEE Communications Magazine* 57, 8 (Aug. 2019), 84–90. https://doi.org/10.1109/MCOM.2019.1900271 Conference Name: IEEE Communications Magazine.
[32] H. Lin. 2015. Flexible Configured OFDM for 5G Air Interface. *IEEE Access* 3 (2015), 1861–1870. https://doi.org/10.1109/ACCESS.2015.2480749 Conference Name: IEEE Access.
[33] Alpha Data Ltd. [n.d.]. ADM-XRC-9R1: FPGA COTS board: Xilinx Zynq RFSoC - 8x 14-bit 4/5Gsps ADCs, 8x 14-bit 6.5/10Gsps DACs. https://www.alpha-data.com/esp/products.php?product=adm-xrc-9r1
[34] Yufei Ma, Naveen Suda, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. 2018. ALAMO: FPGA acceleration of deep learning algorithms with a modularized RTL compiler. *Integration, the VLSI Journal* 62 (June 2018), 14–23. https://doi.org/10.1016/j.vlsi.2017.12.009
[35] Qian Mao, Fei Hu, and Qi Hao. 2018. Deep Learning for Intelligent Wireless Networks: A Comprehensive Survey. *IEEE Communications Surveys Tutorials* 20, 4 (2018), 2595–2621. https://doi.org/10.1109/COMST.2018.2846401
[36] Andreas F. Molisch. 2012. *Wireless Communications*. John Wiley & Sons. Google-Books-ID: 877tFGeQo5oC.
[37] Timothy J. O'Shea, Tugba Erpek, and T. Charles Clancy. 2017. Deep Learning Based MIMO Communications. *arXiv:1707.07980 [cs, math]* (July 2017). http://arxiv.org/abs/1707.07980 arXiv: 1707.07980.
[38] T. O'Shea and J. Hoydis. 2017. An Introduction to Deep Learning for the Physical Layer. *IEEE Transactions on Cognitive Communications and Networking* 3, 4 (Dec. 2017), 563–575. https://doi.org/10.1109/TCCN.2017.2758370
[39] Chandrajit Pal, Sunil Pankaj, Wasim Akram, Amit Acharyya, and Dwaipayan Biswas. 2018. Modified Huffman based compression methodology for Deep Neural Network Implementation on Resource Constrained Mobile Platforms. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–5. https://doi.org/10.1109/ISCAS.2018.8351234 ISSN: 2379-447X.
[40] Thorbjörn Posewsky and Daniel Ziener. 2016. Efficient deep neural network acceleration through FPGA-based batch processing. In *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 1–8. https://doi.org/10.1109/ReConFig.2016.7857167
[41] Thorbjörn Posewsky and Daniel Ziener. 2018. Throughput Optimizations for FPGA-based Deep Neural Network Inference. *Microprocessors and Microsystems* 60 (July 2018), 151–161. https://doi.org/10.1016/j.micpro.2018.04.004 arXiv: 1810.00722.

[42] Jiantao Qiu, Sen Song, Yu Wang, Huazhong Yang, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, and Ningyi Xu. 2016. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '16*. ACM Press, Monterey, California, USA, 26–35. https://doi.org/10.1145/2847263.2847265

[43] Stefania Sesia, Issam Toufik, and Matthew Baker. 2011. *LTE - The UMTS Long Term Evolution: From Theory to Practice*. John Wiley & Sons. Google-Books-ID: beIaPXLzYKcC.

[44] A. Shawahna, S. M. Sait, and A. El-Maleh. 2019. FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review. *IEEE Access* 7 (2019), 7823–7859. https://doi.org/10.1109/ACCESS.2018.2890150

[45] Nir Shlezinger, Yonina C. Eldar, Nariman Farsad, and Andrea J. Goldsmith. 2019. ViterbiNet: Symbol Detection Using a Deep Learning Based Viterbi Algorithm. In *2019 IEEE 20th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*. 1–5. https://doi.org/10.1109/SPAWC.2019.8815457 ISSN: 1948-3244.

[46] Jan Van Leeuwen. 1976. On the Construction of Huffman Trees. 382–410.

[47] Tianqi Wang, Chao-Kai Wen, Hanqing Wang, Feifei Gao, Tao Jiang, and Shi Jin. 2017. Deep learning for wireless physical layer: Opportunities and challenges. *China Communications* 14, 11 (Nov. 2017), 92–111. https://doi.org/10.1109/CC.2017.8233654

[48] Miao Yao, Munawwar Sohul, Vuk Marojevic, and Jeffrey H. Reed. 2019. Artificial Intelligence Defined 5G Radio Access Networks. *IEEE Communications Magazine* 57, 3 (March 2019), 14–20. https://doi.org/10.1109/MCOM.2019.1800629

[49] Hao Ye, Geoffrey Ye Li, and Biing-Hwang Juang. 2018. Power of Deep Learning for Channel Estimation and Signal Detection in OFDM Systems. *IEEE Wireless Communications Letters* 7, 1 (Feb. 2018), 114–117. https://doi.org/10.1109/LWC.2017.2757490

[50] Hanqing Zeng, Ren Chen, Chi Zhang, and Viktor Prasanna. 2018. A Framework for Generating High Throughput CNN Implementations on FPGAs. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '18)*. Association for Computing Machinery, New York, NY, USA, 117–126. https://doi.org/10.1145/3174243.3174265

[51] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *FPGA*. https://doi.org/10.1145/2684746.2689060

[52] Chaoyun Zhang, Paul Patras, and Hamed Haddadi. 2019. Deep Learning in Mobile and Wireless Networking: A Survey. *arXiv:1803.04311 [cs]* (Jan. 2019). http://arxiv.org/abs/1803.04311 arXiv: 1803.04311.

[53] Lin Zhang, Ying-Chang Liang, and Dusit Niyato. 2019. 6G Visions: Mobile ultra-broadband, super internet-of-things, and artificial intelligence. *China Communications* 16, 8 (Aug. 2019), 1–14. https://doi.org/10.23919/JCC.2019.08.001 Conference Name: China Communications.