Deep Learning for Wireless Communications: Flexible Architectures and Multitask Learning

Sarunas Kalade

Strathclyde Software Defined Radio Laboratory
Electronic and Electrical Engineering Department
University of Strathclyde, Glasgow

February 29, 2024

This thesis is the result of the author's original research. It has been composed by the author and has not been previously submitted for examination which has led to the award of a degree.

Abstract

Demand for wireless connectivity has never been higher and continues to grow rapidly. Connecting more devices requires mindfulness in managing the limited resources of energy and radio spectrum. The advent of Software Defined Radio (SDR) has enabled breathroughs in radio configurability, enabling dynamic spectrum access and physical layer optimizations at runtime. In recent years Machine Learning (ML) has been a key enabling technology of various innovations in the wireless communications domain, taking advantage of the newfound flexibility in SDR. The new ML-based signal processing models are no longer based entirely on Digital Signal Processing (DSP) expertise, but are developed in a data-driven approach. This paradigm shift in receiver design is recent, and appropriate architectures and best model training practices have yet to be established.

This thesis explores multiple wireless communications tasks addressed with the toolbox of Deep Learning (DL), which is a subset of ML. Many existing DL solutions are hampered by the limitations of the chosen architectures, which limits their adoptability as drag-and-drop solutions by wireless system designers. Recurrent Neural Network (RNN) and Fully Convolutional Neural Network (FCN) architecture types are explored that enable the adaptability one would expect of classic DSP functions (like the filter).

The field of wireless communications boasts a wealth of data, due to the mature and feature-rich simulation software ecosystem. In Radio Frequency Machine Learning (RFML) this is regularly leveraged to produce datasets for the new data-driven models. Techniques like Multitask Learning (MTL) can exploit this simulated data even further by allowing models to be trained on their primary task, like signal classification or demodulation, while simultaneously estimating the channel quality.

The findings presented in this work show that fully convolutional architectures can be more appropriate for tasks like frame synchronization compared to commonly applied classification models. RNN-based autoencoders achieve good results as an end-to-end trainable receiver solution, however they can be challenging to apply to longer sequences. MTL is identified as an excellent technique not only for training unique models, capable of performing multiple tasks, but as a regularization technique in RFML.

Contents

List of Figures

List of Figures

List of Tables

List of Tables

Acronyms

AI Artificial Intelligence

AMC Automatic Modulation Classification

ANN Artificial Neural Network

ASK Amplitude Shift Keying

AWGN Additive White Gaussian Noise

BER Bit Error Rate

BPSK Binary Phase Shift Keying

CC Computational Complexity

CFO Carrier Frequency Offset

CNN Convolutional Neural Network

CPU Central Processing Unit

CV Computer Vision

DER Detection Error Rate

DL Deep Learning

DT Decision Tree

DNN Deep Neural Network

DSP Digital Signal Processing

EOS End Of Sequence

FC Fully Connected

FCN Fully Convolutional Neural Network

FFT Fast Fourier Transform

FIR Finite Impulse Response

FS Frame Synchronization

FSK Frequency Shift Keying

GMSK Gaussian Minimum Shift Keying

GRU Gated Recurrent Unit

IF Intermediate Frequency

LOS Line-of-Sight

LSTM Long Short-Term Memory

LTE Long Term Evolution

ML Machine Learning

MLP Multilayer Perceptron

MPSoC Multiprocessor System on a Chip

MSE Mean Squared Error

MTL Multitask Learning

NLOS Non-line-of-sight

NLP Natural Language Processing

List of Tables

NR New Radio

OFDM Orthogonal Frequency Division Multiplexing

PLL Phase Locked Loop

PSK Phase Shift Keying

QAM Quadrature Amplitude Modulation

QPSK Quadrature Phase Shift Keying

RC Raised Cosine

RF Radio Frequency

RGB Red Green Blue

ReLU Rectified Linear Unit

RGB Red Green Blue

RNN Recurrent Neural Network

RRC Root Raised Cosine

SDR Software Defined Radio

SGD Stochastic Gradient Descent

SNR Signal-to-Noise Ratio

SOS Start of Sequence

TDL Tapped Delay Line

TPU Tensor Processing Unit

Acknowledgements

I would like to thank my supervisors, Prof Bob Stewart and Dr Louise Crockett, for introducing me to the magical world of DSP and wireless communications, their unwavering positivity, continued support, and giving me so many amazing professional opportunities. None of this would have been possible without them.

The Strathclyde SDR lab is truly one of a kind and it has been a joy and privilege to work alongside so many remarkable colleagues. Thank you to my fellow PhD students for the support and all the tea time chats, it was a blast going through this journey together. I will miss our "CRC fridays" dearly.

Thanks to Neil MacEwen and Daniel Garcia-Alis for the internship opportunities at the Glasgow MathWorks office. I would not have been able to reason about combined communications and ML applications in this work without the help and expertise of the Glasgow MathWorks team. Thank you all so much.

Thanks to Graham Schelle and Patrick Lysaght for the internship opportunity at Xilinx (now AMD). It has been an incredible experience professionally and personally. The year I got to spend in Boulder was one of the best years of my life. I would like to thank Graham for the flexibility in managing my professional commitments while I completed this thesis. And a big thank you to Shane Fleming (AMD) for all the advice and moral support in completing this work.

I would like to thank my friends, family and colleagues for their support and endless encouragement I have received over the years. It has been a long journey and I am incredibly grateful for having been surrounded by such tremendous people.

Chapter 1

Introduction

Advancements in wireless communications and signal processing play a crucial part in keeping the world connected. Both individuals and large corporate enterprises require reliable connectivity in order to thrive in the modern world. The need for connectivity is growing exponentially, as even autonomous devices are being plugged into the world wide web. Modern communication systems are not magic however – they require resources, such as radio spectrum, which is finite. This necessitates engineering increasingly efficient radio transmitters and receivers, capable of better utilizing the limited resources we have to meet the growing demand for connectivity.

Wireless receivers are complex systems composed of multiple Digital Signal Processing (DSP) blocks. Wireless transmissions emitted by a transmitter can travel vast distances, are transformed by a variety of physical obstructions and electromagnetic interference. Typically the signal captured by the receiver will be a much weaker, distorted version of the original transmission. In order to mitigate these challenges, specific signal processing blocks of a receiver perform functions designed to overcome particular hinderences. For example, low-pass filters are designed to remove high-frequency, interfering signals.

It is likely that future communication systems will have many of their functions replaced by neural network equivalents, perhaps the entire system itself [1]. Deep Neural Networks (DNNs) are frequently outperforming select algorithms in radio receivers in metrics such as BERs (Bit Error Rates) and detection accuracies [2], [3], [4], however

often times they can be inflexible to their input sizes [5]. Training DNNs for wireless communications tasks also brings challenges and opportunities unique to the field [6]. The focus of this thesis is the development of DNN architectures that can be used in lieu of traditional DSP algorithms within a radio receiver as drag-and-drop solutions.

This chapter will outline the main motivations for the work carried out in this thesis – mainly why flexible DNNs, capable of operating on variable sized inputs, are desirable in the field of RFML (Radio Frequency Machine Learning) and how this idea fits into the larger vision of the future radio receiver.

1.1 Deep Learning in Wireless Communications Physical Layer

Classic radio receivers are typically designed by human experts using a systematic approach, where each signal processing block is designed and evaluated individually, before being integrated into the full system. Since the popularization of mobile devices and increasing bandwidth requirements throughout the generations of wireless technology, these receivers have also become more complex [7], [8]. Entire books have been written for understanding just a single aspect of a wireless receiver, e.g. synchronization [9], or channel equalization [10]. The resulting complexity of the full system in latest generation wireless communications standards creates a very large optimization surface with an immense amount of design choices and parameters.

On the other hand, the unprecedented number of wireless devices has caused a surge of data, and thanks to the research efforts made by the Machine Learning (ML) and data science research communities, there is a growing number of tools that can harness this information for a variety of useful endeavors [11]. One of these is to create powerful RFML models capable of learning to approximate very complex functionality that typically takes expert knowledge and time to design.

There are a few key reasons why ML is seen as an attractive option for signal processing tasks in wireless communications:

1. It is difficult to programmatically solve complex tasks, such as network traffic pre-

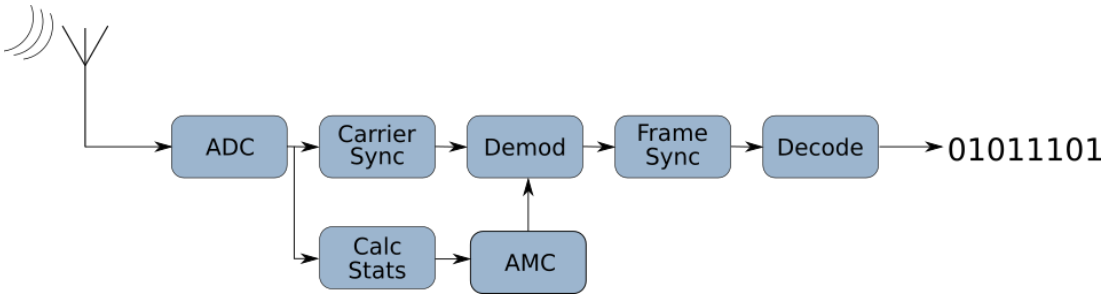
diction [12] or channel occupancy prediction [13]. Sometimes Deep Learning (DL) methods prove to be much better at the task than a human-defined algorithm (as is the case in Automatic Modulation Classification (AMC) [14]).

2. Counter-intuitively, DL techniques can sometimes be successfully deployed in a communications scenario to reduce complexity and save resources, as demonstrated for polar decoding [15]. Reduction in complexity is usually seen when the original algorithm has a highly iterative nature, requiring many loops of computation, which does not translate well to hardware. Most neural networks, on the other hand, are easy to parallelize and generally provide a prediction in a single processing forward pass.

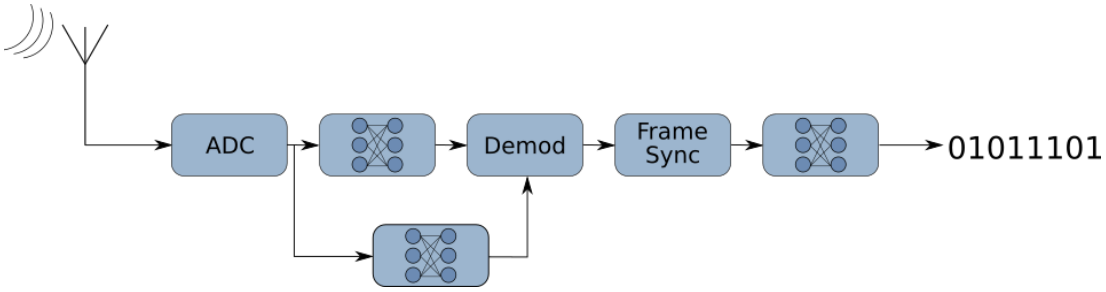
3. According to the global optimality theorem [16], a globally optimized system typically outperforms one built with separately designed and evaluated parts. Similar trends in fields such as Computer Vision (CV), where the best algorithms were initially designed with individually tested blocks, such as edge detection and color thresholding, but eventually moved on to an end-to-end trainable approach with deep CNNs (Convolutional Neural Networks) [17].

As shown in Figure 1.1 (a), a transitional period is required where only parts of the system are being replaced by proven ML solutions. There are still many concerns and issues associated with DNNs – difficulty of training and explainability [18], [19], [20] being major ones. Having unexplainable "black box" components within a mission-critical system will warrant extra validation as well [21],, for example, this is an actively researched topic in the area of driverless vehicles [22]. Due to these concerns, adoption can be slow and it is very likely that the field will remain in the transitional stage illustrated in Figure 1.1 (b) for decades to come.

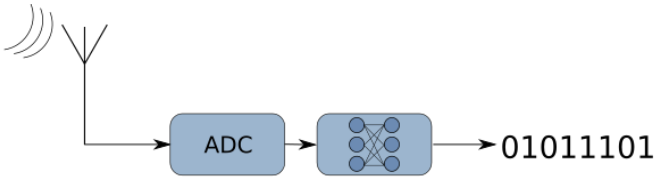
The work in this thesis addresses the challenges of the transitional stage in wireless communications. The core aim of the work is to develop training and deployment techniques for DNNs as drag-and-drop replacements for DSP blocks in wireless receiver implementations.



(a) Classic receiver processing chain – entirely based on human-engineered algorithms and DSP



(b) Transitional period of functions being replaced by DNNs, key functionality being replaced by DNNs that outperform expert designed systems



(c) Future receiver – the whole processing chain substituted with an end-to-end trainable DNN

Figure 1.1: New Signal Processing with Deep Learning.

1.2 Research Objectives

The aim of this research work is to develop DNNs that integrate well into existing system design tools and address key issues hindering broader adoption, such as the need for RF data-appropriate architectures, field-specific training and explainability. To accomplish this, the main objectives are laid out as follows:

- Develop adaptable DNN architectures capable of handling variable input sizes. This will enable their use as drag-and-drop solutions in wireless receivers.
- Establish wireless communications-specific training techniques and methodologies that leverage existing simulation tools and available channel models.

The following subsections expand on the importance of achieving these objectives in the context of the wider field of wireless communications and RFML.

1.2.1 Adaptable DNN Architectures and the Importance of Variable Input Sizes

A typical radio signal processing flowgraph is composed of DSP blocks, such as FIR (Finite Impulse Response) filters, PLLs (Phase Locked Loops), etc. Using industry standard design tools like MathWorks Simulink [23], these can be drag-and-dropped from a library into a receiver design and operate on streams of samples.

Many wireless communications problems are addressed with ML in a uniform way – a classification approach. However, this is not necessarily the most appropriate solution to every problem. In order to perform classification, the DNN will be trained on a fixed number of class outputs, using Fully Connected (FC) layers, limiting the deployment of these models to a specific number of classes. In some instances the number of classes can be very large – for example, if the DNN is mimicking a filter and the number of outputs equals the number of inputs. In these instances the model can be difficult to scale to larger inputs. Even worse, in order to apply the DNN to a larger (or smaller) input the model architecture will have to be re-tuned for the new input shape and a new model re-trained.

Re-training DNNs can be time-consuming and require DL expertise. In order to avoid this scenario, more appropriate DNN architectures can be trained and deployed on variable input sizes. Ideally these data-driven solutions should be available to non-ML experts as importable modules in a software package, with minimum effort required to integrate into a new receiver. The types of architectures capable of this versatility are explored in this thesis.

1.2.2 Unique Training Techniques in Wireless Communications

In addition to exploring novel architectures that could act as stand-ins for tried and tested DSP blocks, there is massive scope for exploiting rich simulated wireless communications datasets. Training DNNs for problems being addressed by wireless receivers

has unique challenges not present in other major areas where DL techniques are most prominent – such as CV and Natural Language Processing (NLP). Wireless receivers must deal with a variety of channel impairments such as multipath fading, frequency offsets due to oscillator mismatches, thermal noise, and non-linearities from minor component imperfections. Each environment differs, and designing a model that can generalize to various channel conditions is a significant challenge.

However, with these challenges come unique opportunities – transmitted signals are engineered and known beforehand at the transmitter, meaning that in a simulation there exists perfect knowledge of the transmitted and received versions of the signal. The channel models to evaluate these algorithms are now fairly mature and most impairments, that are encountered in practice, can be effectively simulated. All of these simulation parameters can be used as additional training data to create robust RFML solutions with techniques like Multitask Learning (MTL). Few fields have accumulated such extensive simulation libraries, so as to allow effectively generating data – perfect for ML algorithm consumption.

This research seeks to contribute to the advancement of DNN adoption in wireless receiver implementations by creating adaptable architectures and training methodologies that leverage the rich wireless datasets in ways unique to the field.

1.3 Related Work

Previous work with RNNs (Recurrent Neural Networks) in wireless communications has shown their effectiveness for demodulation of speech signals as an end-to-end training solution [24]. They have also been shown to work well for AMC tasks – the automatic identification of the modulation type of a received signal [25]. AMC is a crucial process for cognitive radio applications, where accurate sensing of the radio spectrum is essential [26].

Seq2Seq (Sequence-to-Sequence) models are an advanced arrangement of two RNNs. Initially, these models have been widely applied for language translation tasks, where input and output sequences can vary [27]. In the wireless communications domain, Seq2Seq models have been used for channel prediction [28], where the flexibility of the

encoder-decoder structure allows prediction of channels of various time spans. Seq2Seq models have been shown to work well on problems like MIMO (Multiple-Input Multiple-Output) channel state information compression [29] and RF fingerprinting [30], to determine a device's location based on signal strength measurements. A Seq2Seq model was used to predict sensor measurements in order to avoid unnecessary transmissions, showing to reducuction transmitter energy consumption by up to 90% in Wireless Sensor Networks (WSNs) [31]. Despite these varied applications, to the best of the author's knowledge, simultaneous AMC and demodulation tasks using Seq2Seq models have not been demonstrated in previous literature.

Frame synchronization is an important part of a wireless receiver, responsible for detecting transmitted packets. DL-based solutions to frame synchronization have previously been proposed by utilizing a regression CNN, where the final layer is a single neuron estimator of the packet start index [32], [33]. Another popular CNN-based approach in the literature is to treat frame synchronization as a classification problem, where the output of the CNN is equal in size to the input, and the assumption is made that there will be a single packet in an input signal [34], [35], [36]. An RNN-based approach has also been proposed, showing promising results on periodically repeating synchronization patters, however this method has not been explored on bursty communications [37]. All of the reviewed CNN-based frame synchronization models contain fully connected layers in the detector architecture, preventing the models from being deployed on inputs of any other shape than the one they were trained on. The work in this thesis addresses the fixed-size limitations of previous research with a fully convolutional architecture applied to bursty wireless communications transmissions.

To take advantage of the rich datasets available in wireless communications, training techniques like Multitask Learning (MTL) show potential to improve the performance of DNN models in the RFML domain. MTL is a popular topic of study in the fields of CV and NLP [38], where multiple related tasks can be learned while sharing the feature extraction layers of a DNN. MTL can improve the performance of individual tasks by having the network learn them at the same time – what is learned for one task can help other tasks learn better [39]. A strong case for SNR (Signal to Noise Ratio)

estimation as an MTL implementation has been made in the NLP domain for speech recognition [40], where SNR estimation is used as a secondary task, imbuing a CNN model with SNR-awareness. Using this approach, instead of training multiple models for different SNR conditions and pairing them with an SNR estimator, a single DNN can be deployed performing both tasks simultaneously.

AMC models can be sensitive to specific SNR conditions, and previous research has been conducted to address this by training multiple AMC models for different SNR ranges, then classifying the incoming signal based on an SNR estimation [41], [42]. Later works have explored a single model solution using MTL and training AMC and SNR estimation tasks simultaneously to create an SNR-aware model [43], [44]. A slightly different variation of MTL has been proposed in [45], where a denoising autoencoder is used as the secondary task to AMC – this makes the model SNR-aware, without explicitly training it to estimate SNR. AMC has been the most popular application for targeting with MTL, however more recent works are emerging on Channel Estimation (CE) and Carrier Frequency Offset (CFO) estimation for OFDM systems [46], [47].

The reviewed works on AMC treat SNR estimation as a binary or tertiary classification problem, demonstrating that even a coarse prediction of 'low', 'medium' and 'high' SNR is enough to improve AMC performance. An aspect that has not been addressed is the comparison of different SNR estimation methods and how they affect MTL-trained model performance. In this work SNR estimation, as part of an MTL investigation, is evaluated with different regression and classification estimators.

Outside of MTL, DNN-based SNR estimation has been explored in previous research [48] [49] [50] [51]. However, to the best of the author's knowledge, continuous SNR estimation using a Fully Convolutional Neural Network (FCN) and an MTL training scheme has not been explored in previous work.

1.4 Contributions

The contributions resulting from this research work are considered to be the following:

- **Introduction of a Seq2Seq autoencoder model to address the problems**

of AMC and digital baseband symbol recovery simultaneously. To the best of the author's knowledge a Seq2Seq model that combines these tasks has not been proposed in existing literature. To expedite the training process and alleviate some of the difficulty of training RNNs, a hybrid-model with a convolutional encoder was proposed. The combined Conv-Seq2Seq showed improved training convergence properties, compared with a purely RNN-based Seq2Seq model.

- **Development and evaluation a novel FCN architecture or "deep filter" for physical layer frame synchronization.** A fully convolutional architecture allows inference on input sequences of any length, which was not addressed in existing DNN-based works addressing the same problem. Historically DL solutions to frame synchronization have treated the problem using classification, rather than regression. The trained models were evaluated on different preamble lengths, showing that DL-based frame synchronization works well for very short preambles compared to correlation-based methods, especially when considering channel impairments like CFO.

- **Introduction of novel uses of MTL to train DNNs using wireless channel simulation parameters as additional training labels.** The proposed MTL methodology uses channel simulation parameters as additional labels to train additional estimator heads, which can be removed post-training for inference. The technique acts as additional regularization, guiding the model to generalize for a wider range of interferences. This method was used to train two models: AMC with SNR estimation, and frame synchronization with CFO estimation. A consistent improvement was shown over baseline models, achieved without increasing computational, time, energy, or other costs at inference time.

- **Development of a novel double-headed fully convolutional MTL implementation for frame synchronization and continuous SNR estimation.** To the best of the author's knowledge this is the first implementation of a DL-based method for continuous SNR estimation. The FCN SNR estimator outperformed the baseline model at low SNR.

- **An investigation of the tradeoffs and best criteria for MTL-related parameters are presented in this work.** Best practices for MTL in wireless communications have yet been defined and MTL-specific parameters, like individual task loss weightings have not been discussed in detail in existing literature. Additionally, a series of wireless communications dataset insights and best practices have been collected from training a variety of models for multiple different applications. As such, a methodology has been developed that is hoped to be useful for future researchers in the field of RFML.

1.5 Publications

Several works have been published as part of this research project. These are:

[a] S. Kalade, L.H. Crockett and R.W. Stewart, "Using Sequence to Sequence learning for digital BPSK and QPSK demodulation," (conference paper), in *2018 IEEE 5G World Forum (5GWF)*, pp. 317-320, IEEE, 2018.
Available: `https://doi.org/10.1109/5GWF.2018.8517049`

[b] S. Kalade, L.H. Crockett and R.W. Stewart, "Using Deep Learning for Simultaneous Classification and Demodulation of Wireless Communications Signals" (poster), Presented at *New England Workshop on Software Defined Radio*, Worcester, USA, 2018.
Abstract Available: `https://newsdr.org/workshops/newsdr2018/`

[c] S.Kalade, "Introduction to Deep Learning" in *Exploring Zynq MPSoC: With PYNQ and machine learning applications* (book chapter), Strathclyde Academic Media, 2019.
Available: `https://www.zynq-mpsoc-book.com/`

[d] S. Kalade, "Quirks and Opportunities of Training Deep Learning Systems for Future Wireless Networks" (presentation), Presented at *6G: Software Defined Radio and RF Sampling*, TechUK, UK, 2021.

Available: https://www.techuk.org/resource/slides-software-defined-radio-and-rf-sampling.html

[e] S. Kalade, L.H. Crockett and R.W. Stewart, "Training Deep Filters for Physical-Layer Frame Synchronization," (journal), in *IEEE Open Journal of the Communications Society*, vol. 3, pp. 1063-1075, IEEE, 2022.
Available: https://doi.org/10.1109/OJCOMS.2022.3185973

1.6 Thesis Organization

The remainder of this thesis is structured as follows:

- **Chapter 2** is the wireless communications background chapter that reviews the main modulation types and common channel impairments in the context of dataset generation for RFML. This chapter also summarizes the fundamentals of AMC and explains the reasoning behind the transition from ML to DL algorithms in this space.

- **Chapter 3** covers the basic theory of DL, main architecture types, common design patterns and explains the process of training a DNN.

- **Chapter 4** introduces Seq2Seq models and presents a methodology for training them for AMC and PSK demodulation. The challenges of training these networks are discussed and architectural changes proposed to alleviate these issues. This work has been published in [a],[b].

- **Chapter 5** presents the novel work done on fully convolutional architectures applied for physical layer frame synchronization, in this thesis referred to as "deep filters". An extensive architecture design study is performed, including an introspection into the individual layers and a complexity analysis. The majority of this work was published in [e].

- **Chapter 6** reviews MTL and how it can be used for wireless communications data to improve model performance or train novel models that can perform multiple

tasks simultanously. Initial results were presented in [d].

- **Chapter 7** summarizes the thesis, presents key conclusions and future work.
- **Appendix A** contains training summaries for the models trained in chapters 4-6.
- **Appendix B** describes a general methodology developed over the course of this research for training DNNs for wireless communications data.
- **Appendix C** describes the reproducibility efforts of this work and points the reader to the GitHub repository containing the jupyter notebooks used to generate the results presented in this thesis.

The next chapter will introduce the fundamentals of signal processing for wireless communications.

Chapter 2

Signal Processing Background

This chapter introduces the fundamental concepts and problems in wireless communications that are explored in this thesis.

2.1 Communications Link Overview

A basic communications system is composed of a transmitter, channel and receiver. The transmitter will typically encode the raw bits of data for redundancy purposes – this could be simple bit repetition or more advanced error-correcting codes like hamming [52]. Encoded bits are then mapped into digital symbols, where each symbol can represent a number of bits, according to the modulation scheme – this is typically referred to as baseband modulation. RF transmissions are quite stringently regulated and in order to comply with established wireless standards, for example the 5G New Radio the spectral emission mask parameters are listed in detail in the standard documents [53]. Extra processing is required to perform the band-limiting the signals – this is typically done by pulse shaping filters. Additionally, in order to actually transmit data via RF frequencies, the modulated symbols have to be upconverted to an RF frequency and then converted to an analog signal with a Digital-to-Analog Converter (DAC). Multicarrier systems will perform additional steps, such as de-serializing the bitstream and multiplexing the modulated bits onto different carriers in parallel, however single carrier configurations are mainly discussed in this thesis.

Receivers, on the other hand, must perform the inverse operation of the transmitter, which would be straightforward – if not for the channel. In addition to converting the signal back into the digital domain with an Analog-to-Digital Converter (ADC), executing the decimation, demodulation and decoding, the receiver also has to overcome various impairments introduced by the wireless channel, such as noise, frequency and phase offsets, gain imbalances, etc. Some of these impairments have little to no recourse, for example there are few practical solutions for thermal noise introduced by the physical characteristics of the components making up the receiver. Others, such as frequency offset or time delay, can be corrected by employing synchronization.

A basic overview of a communications link can be observed in Figure 2.1. Illustrated is one of the possible single carrier communications link arrangements at a high level. Note that the receiver is much more complex than the transmitter.

2.2 Digital Modulation Schemes

Wireless devices, like phones or routers, work by converting data bits into signals carried by radio waves. However radio transmissions have a cost – radio spectrum, and it is in our best interest to maximize the number of bits per Hz. The simplest example is the On-Off keying modulation scheme, where a signal being 'on' means a 1 is transmitted, 'off' means a 0. If bits are transmitted this way at a rate of **B** then a spectral efficiency **B** bits/Hz is achieved.

In a simple scenario like this, increasing the throughput is trivial – just speed up the rate at which bits are transmitted. However this incurs the cost of additional bandwidth, which is undesirable since spectrum is a finite resource. In order to transmit more bits using the same bandwidth, the bits would have to be packed into different symbols, for instance, instead of having 2 levels to represent 'on' and 'off', 4 voltage levels could be used instead to represent pairs of bits (-1V is '00', -$\frac{1}{3}$V is '01', +$\frac{1}{3}$ is '10' and +1V is '11'). Now the transmitter can emit **2B** bits/Hz. Of course there are tradeoffs, e.g. the number of voltage levels used will depend on how noisy the channel is. Some modulation schemes will be better suited for some channels than others, but this is the key idea behind modulation.

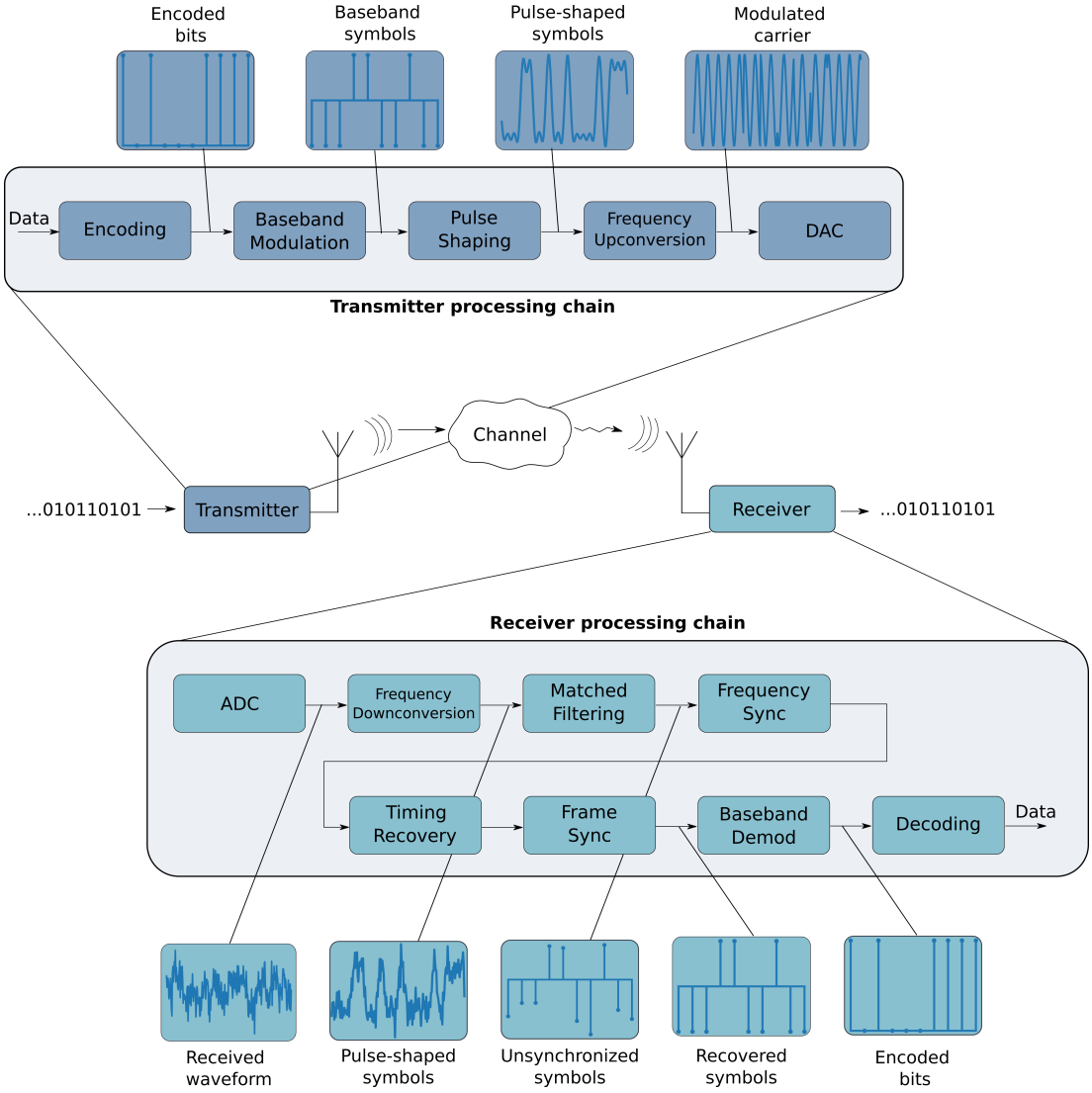


Figure 2.1: Communications link overview

Baseband modulation takes place before Intermediate Frequency (IF) and/or Radio Frequency (RF) modulation. It involves mapping data bits to symbols according to the modulation scheme. Popular schemes are Amplitude Shift Keying (ASK), Phase Shift Keying (PSK), Frequency Shift Keying (FSK) and QAM (Quadrature Amplitude Modulation), which is a combination of both ASK and PSK. This section will cover the foundations necessary to recognize and understand these modulation types.

2.2.1 Amplitude Shift Keying

The most intuitively understandable digital modulation scheme is ASK. ASK-2, ASK-4, ASK-8, etc. are examples of ASK, where the increasing number dictates how many symbols or amplitude levels are used in a modulation scheme. The advantage of ASK is that it is very simple to implement, for example ASK-2, sometimes known as OOK (On-Off keying) simply has 2 states where the transmission is in an ON state or an OFF state. The disadvantage is that, since the modulation scheme is entirely based on amplitude, it is sensitive to noise and path gain/loss effects. Since real world transmissions are generally noisy these modulation schemes are rarely seen in widely used communications standards.

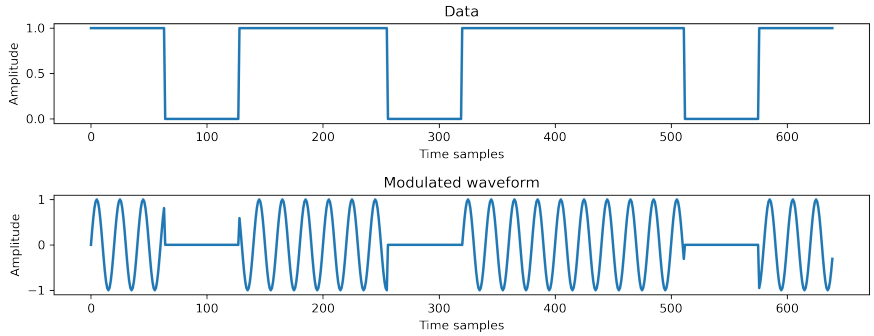


Figure 2.2: Amplitude Shift Keying (2-ASK/OOK)

Figure 2.2 shows the 2-ASK symbols and what the resulting modulation would look like once the symbols have been modulated onto a carrier. The modulated carrier signal is transmitted by an RF front end interfacing with the physical world.

2.2.2 Phase Shift Keying

BPSK, 4-PSK, 8-PSK, 16-PSK are examples of Phase Shift Keying (PSK). When data is modulated using PSK schemes the output modulated signal amplitude is constant, however phase transitions between symbols, as seen in Figure 2.3, occur at the symbol edges. The way different symbols are extracted in this scheme is by evaluating the phase differences between each symbol period. Since phase is not affected by variations in amplitude, it is more robust compared to modulation schemes that rely on amplitude.

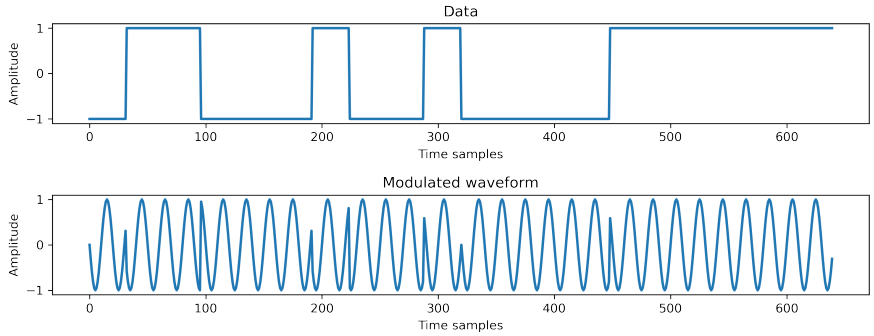


Figure 2.3: Binary Phase Shift Keying (BPSK)

Once modulated onto a carrier, subtle symbol transitions can be seen where the sine wave seemingly restarts at different points. Abrupt phase differences, producing sharp changes in the time domain, are generally undesirable. In practice the modulated symbols go through more processing steps such as pulse shaping, covered in the next section, which act to soften these transitions.

In wireless communications modulation is often performed by transmitting a Real (In-phase) and Imaginary (Quadrature) components, by modulating the signal using 2 oscillators that are 90 degrees out of phase (orthogonal to each other). The motivation for doing so is to double the spectral efficiency compared to a modulation using only a single carrier. The simplest such example is QPSK (equivalent to 4-PSK), which combines 2 orthogonal BPSK streams at the same carrier frequency, as shown in Figure 2.4. Nearly all of the communications signals in this thesis will take this form.

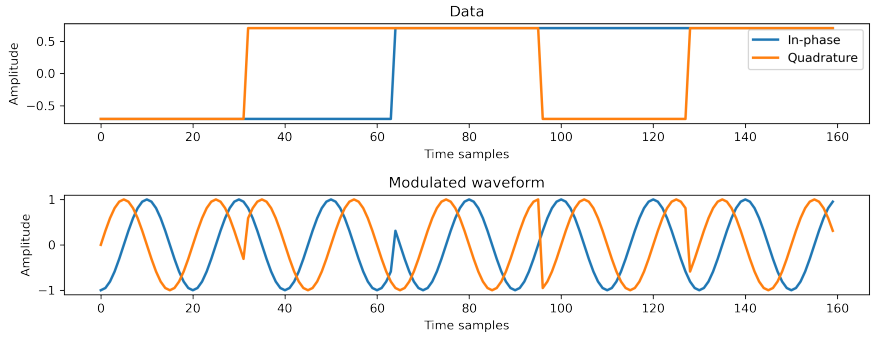
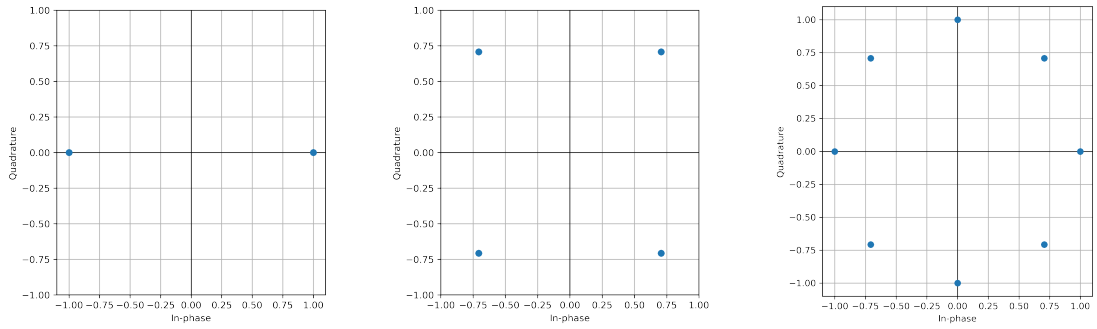


Figure 2.4: Quadrature Phase Shift Keying (QPSK)

Inspecting the symbols in time domain graphs can be difficult, especially when

modulation order increases – expanding the number of possible phase shifts. In order to better visualize and evaluate PSK modulated data, constellation diagrams are often used. The baseband symbols of BPSK, QPSK and 8-PSK modulated data can be observed in Figure 2.5. Note how BPSK sits only on the real plane, while the other modulation schemes use both I and Q channels to represent the symbols.



(a) BPSK (b) QPSK (c) 8-PSK

Figure 2.5: PSK Constellation Diagrams

PSK is still limited in efficiency – note that, in all of the subplots of Figure 2.5 the symbols occupy only the outer perimeter of the circle – no amplitude information is transmitted.

2.2.3 Frequency Shift Keying

Having covered amplitude and phase, the third key property of a carrier that can be used to convey different symbol values is frequency. An example of frequency-based digital modulation is Frequency Shift Keying (FSK). FSK is very commonly used in widely deployed standards such as Bluetooth [54]. This scheme is implemented by having each symbol be represented by a particular carrier frequency. Detecting a signal at a particular pre-defined frequency counts as receiving a symbol. Transmitted symbols, modulated by the 2-FSK modulation scheme, where two different carrier frequencies are used, can be seen in Figure 2.6.

FSK cannot be easily examined using constallation diagrams, however performing an FFT on a FSK-modulated signal results in 2 distinct spikes (for 2-FSK), showing where the symbols are located in the frequency domain (Figure 2.7). The challenge with

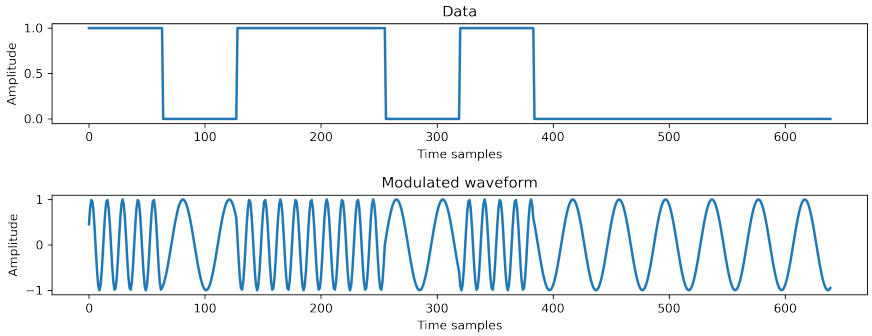


Figure 2.6: Frequency Shift Keying (2-FSK) in time

FSK is to successfully transmit and recover multiple frequencies at a limited bandwidth – spectrum leakage, noise and quantization errors can make this rather difficult, as these interference sources can 'flatten' the frequency peaks and make them less discernable.

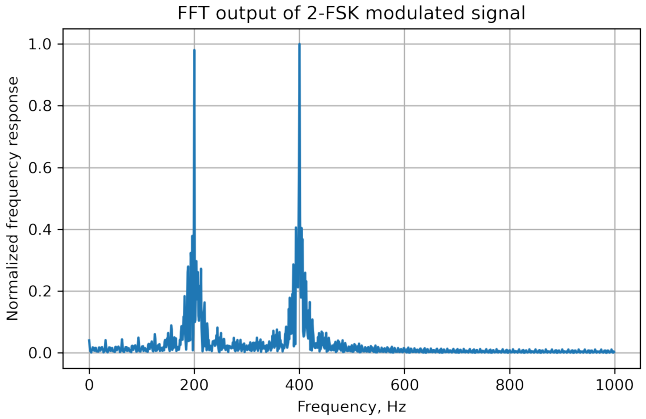


Figure 2.7: Frequency Shift Keying (2-FSK) in spectrum

2.2.4 Quadrature Amplitude Modulation

The final modulation scheme class that will be covered in this chapter is Quadrature Amplitude Modulation (QAM), a widely used modulation scheme that is a combination of both amplitude and phase shift keying. This is a very common modulation scheme, often used as part of multicarrier modulation such as OFDM (Orthogonal Frequency-Division Multiplexing) seen in WiFi and mobile standards like LTE [52, Chapter 15]. QAM can be configured with different numbers of modulation levels (in some applica-

tions up to 4096-QAM [55]). Typically, when a channel has very favorable conditions, such as low noise, a high order QAM scheme can be used to maximize the number of bits conveyed per symbol. However, when SNR drops and bit error rates reach an unaccaptable threshold, the communications link can switch to a lower order QAM modulation.

Technically the previous example of QPSK has already shown a QAM constellation, because 4-PSK, QPSK and 4-QAM are equivalent. Going to a higher modulation level, an example of 16-QAM modulated constellation is shown in Figure 2.8.

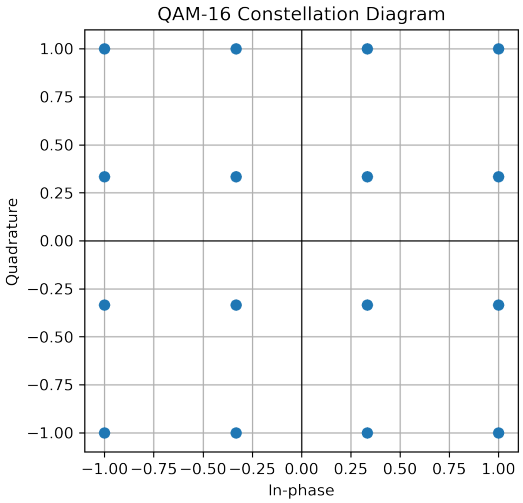


Figure 2.8: QAM-16 constellation diagram

As mentioned earlier, the modulated signals presented here exhibit discontinuities and sharp transitions between symbols. Representing such rapid transitions in the frequency domain requires frequency components that are often outside of the desired bandwidth that the signal should occupy. In order to prevent spectral leakage to adjacent communications channels results from these effects, techniques like pulse shaping are often used.

2.3 Pulse Shaping

In order to band-limit the transmitted pulses (baseband symbols), some additional filtering should be applied to the samples that are being transmitted. This pulse shaping

stage takes place after baseband modulation. Pulse shaping is required in order to prevent spectrum leakage and ensure adherence to spectral mask requirements specified in different standards [56, Chapter 21].

2.3.1 Sinc Pulse Shape

The simplest pulse shaping method is to repeat the symbol samples – this is called square pulse shaping, and has been applied implicitly in the previous examples of this chapter when modulating symbols onto a carrier. The downside of this approach is that changes at the edges of the square pulses are still sharp in the time domain, which means that pronounced side lobes will be present in the frequency domain, causing interference to nearby transmissions. A solution to these sharp transitions is the sinc-shaped filter. The sinc function is simply given by

$$sinc(x) = \frac{sin(\pi x)}{\pi x}. \tag{2.1}$$

An infinite sinc shaped filter can achieve ideal frequency limiting properties, however implementing this filter is not practically feasible since hardware resources are not infinite. Instead, an approximation is often deployed by using the sinc function and implementing a truncated version of an ideal sinc filter, as shown in Figure 2.9.

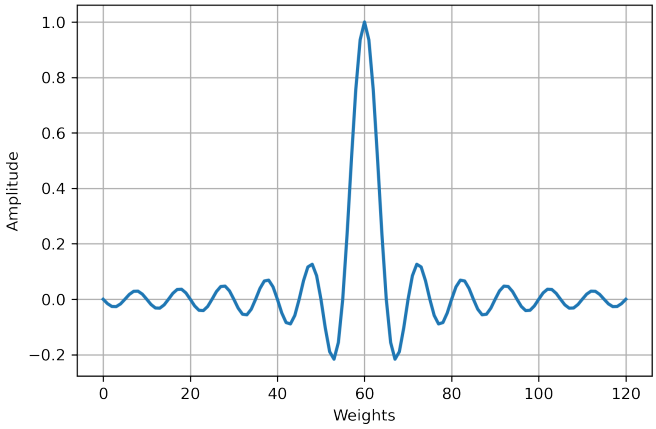


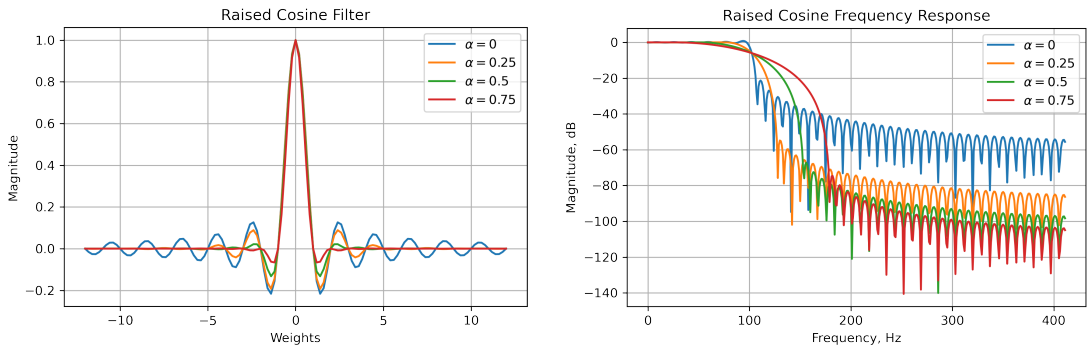
Figure 2.9: Sinc function

2.3.2 Raised Cosine

The truncated sinc filter is not ideal and can still result in sizeable side lobes in the frequency domain due to its long tapering tails. Other variants of sinc-shaped filters, such as Raised Cosine (RC) are very commonly used in wireless communcations, and allow for higher parameterization and tradeoffs in filter complexity and spectral leakage [52, Chapter 3]. The RC filter is achieved with an additional design parameter α, which governs the excess bandwidth (also referred to as roll-off factor), or how wide the main lobe of the signal will be in the frequency domain, the new equation is defined as

$$rc(x) = (\frac{sin(\pi x)}{\pi x})(\frac{cos(\alpha \pi x)}{1 - (2\alpha x)^2}). \tag{2.2}$$

Note that when $\alpha = 0$, the RC filter is equivalent to the previously defined Sinc filter. Effects of the design parameter α has on the filter weights and frequency response is shown in Figure 2.10.



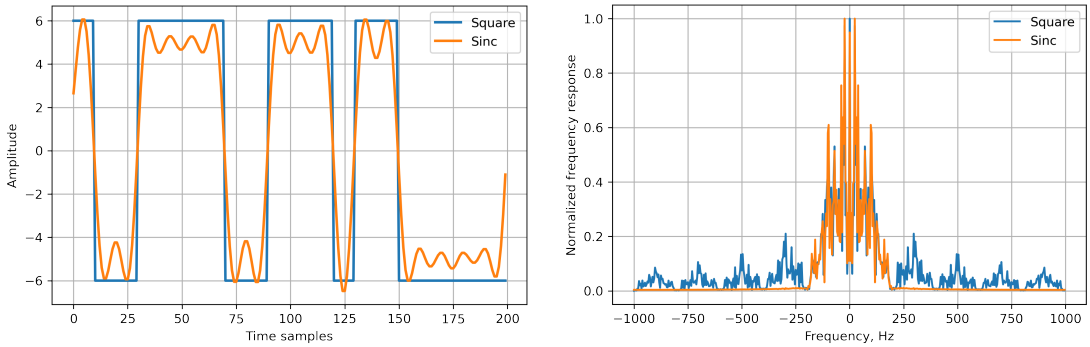
(a) Parameter α effect on filter weights

(b) Filter frequency response

Figure 2.10: Raised Cosine filter

As a concrete example, the effects of pulse shaping on spectral leakage are illustrated in Figure 2.11. Note that the square pulse shape results in many side lobes leaking energy into adjacent frequency bins – this is the reason why square pulse-shaped data is usually not transmitted over the air, as it would cause a great deal of interference to adjacent frequency channels.

Applying a pulse shaping filter to these repeated pulses results in a slightly perturbed time domain graph, however this has a very clear effect of minizing the side



(a) Time domain

(b) Frequency domain

Figure 2.11: Square and sinc pulse shaped-symbols in time and frequency domains

lobes in the frequency domain.

2.3.3 Matched Filtering

The RC filter has the desirable property of not introducing Inter Symbol Interference (ISI) to the pulse shaped data. This means that the maximum effect points (the samples that are used to determine the symbol in baseband demodulation) are unaffected by their neighbours. The raised cosine filter can be split into 2 linear filters at the transmitter and receiver, and the resulting filter is called the Raised Root Cosine (RRC) filter. Using two identical RRC filters at the transmitter and receiver is referred to as matched filtering, which reproduces the original RC transfer function, ensuring that the property of zero ISI is maintained.

An example of matched filtering with two RRC filters is demonstrated in Figure 2.12. A single RRC filter distorts the signal and introduces ISI, as seen on the left plot – the individual symbols do not perfectly overlap at the zero crossings, interfering with each other. However, once the RRC filter is applied again on the receiver side, the original RC response is achieved and the symbols at the receiver do not experience any ISI.

RRC filters are the most common pulse shaping filters and will be seen in many datasets for RFML.

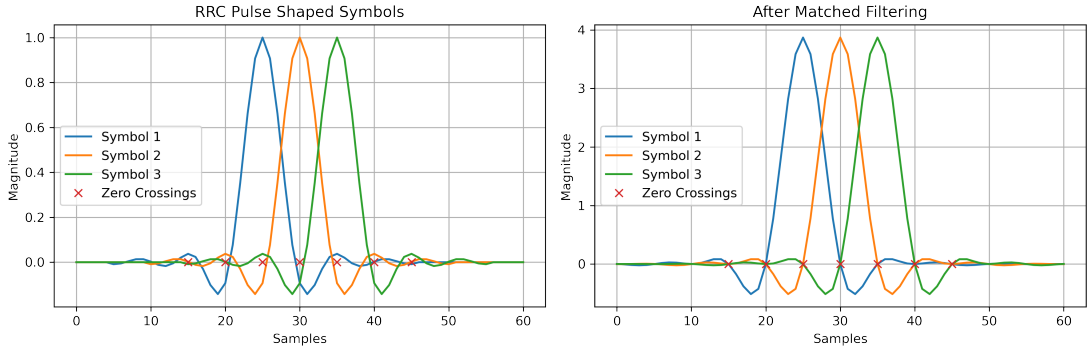


Figure 2.12: RRC matched filtering

2.4 Channel Effects

Transmitted signals travel across wide distances, get partially absorbed, transformed, bounced off various surfaces, and can arrive at irregular intervals to their destination. There are man-made obstacles, such as buildings, and cars, as well as natural foliage like trees that can act as filters. Nearby wireless devices like routers, phones or other radio-capable equipment add noise in the form of interference. All of these effects and more have to be overcome and compensated to some degree by the receiver.

Ignoring the forces of nature and cityscapes that influence the wireless world, even a clear Line of Sight (LOS) channel between transmitter and receiver will pose a great deal of issues. For example, if the receiver local oscillator does not match the carrier frequency exactly – the received signal will suffer from frequency offset errors. Additionally, it is unlikely that the phase of the received carrier signal and receiver local oscillator will match, necessitating the receiver to compensate for the phase offset. It is also important to remember that the components that make up a radio receiver will exhibit thermal noise and contribute to some loss of SNR. Local interfering transmissions will also negatively impact SNR and generally are difficult to avoid.

Understanding these channel effects is not only necessary to develop good receivers, but it is also imperative to produce comprehensive datasets for ML.

2.4.1 Signal to Noise Ratio

The concept of Signal to SNR is important in various aspects of wireless communications. Linear SNR is simply defined as the ratio of the signal power P_s and noise power P_n. Much more commonly, however, SNR is expressed in decibels (dB), and is calculated as follows:

$$SNR_{dB} = 10\log_{10}(\frac{P_s(W)}{P_n(W)}). \tag{2.3}$$

When describing power in wireless communications the ranges can vary drastically from one millionth of a watt in handheld mobile devices to hundreds of watts in operating FM stations [57]. Expressing these measurements as dB values is much simpler and more readily understood.

In wireless communications, many sources of noise are present that result in loss of SNR, such a thermal properties of the electronics, and interference from nearby transmitters. Using modern simulation software, these noise effects can be readily simulated. The most popular noise model used in wireless channel simulations is Additive White Gaussian Noise (AWGN). The noise samples are drawn from a zero-mean normal distribution $\mathcal{N}$, parameterized by the variance σ^2 as shown in Eq 2.4.

$$n \sim \mathcal{N}(0,\, \sigma^2). \tag{2.4}$$

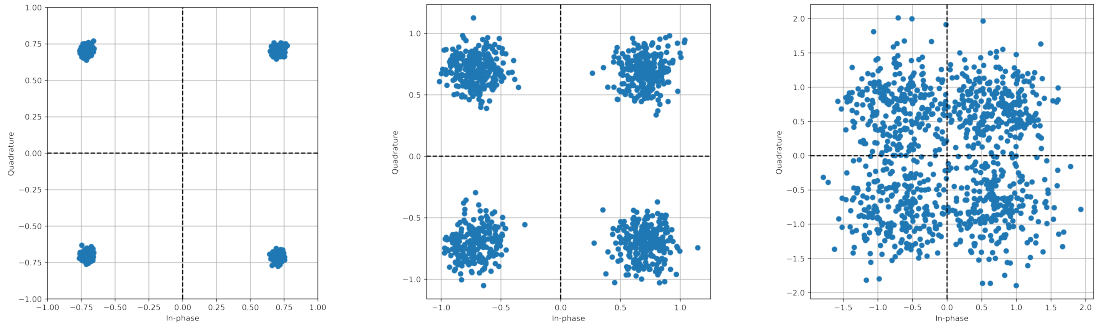
The amount of noise is determined by the variance of the normal distribution. In order to use this generated noise, it can simply be added to the signal s, producing the received, noisy signal r, as demonstrated in Eq 2.5.

$$r = s + n. \tag{2.5}$$

The simple addition in order to apply the noise is where AWGN gets the "Additive" portion of the name.

Effects of AWGN on a baseband QPSK-modulated signal at different SNR levels are illustrated in Figure 2.13. Note that a certain degree of noise in the system can be

tolerated, however as we approach 0dB, where the signal and noise power is equal, it becomes much harder to discern whether the symbol has been received in the correct quadrant.



(a) QPSK at 30dB SNR (b) QPSK at 15dB SNR (c) QPSK at 5dB SNR

Figure 2.13: Effects of AWGN on constellation diagrams

In most channels noise is an unavoidable truth and bit errors will occur – this is why source and channel coding schemes are important.

2.4.2 Carrier Offsets

To demodulate and recover the symbols being transmitted over the air, the local oscillator of the receiver must match the frequency and phase of the carrier being emitted by the transmitter. In a real-world system there will inevitably be physical differences between the transmitter and receiver, which can cause a mismatch of frequencies or Carrier Frequency Offset (CFO). Frequency offsets can also result from movement of one (or both) of the transmitting or receiving devices (such as a phone) – this is known to be caused by the Doppler effect [52, Chapter 14]. Due to the distance the carrier wave must travel, the phase of the received waveform is unpredictable and is very unlikely to initially match the local oscillator phase. Without correction, these effects can significantly impact the receiver's ability to recover the data bits of the transmitted signal.

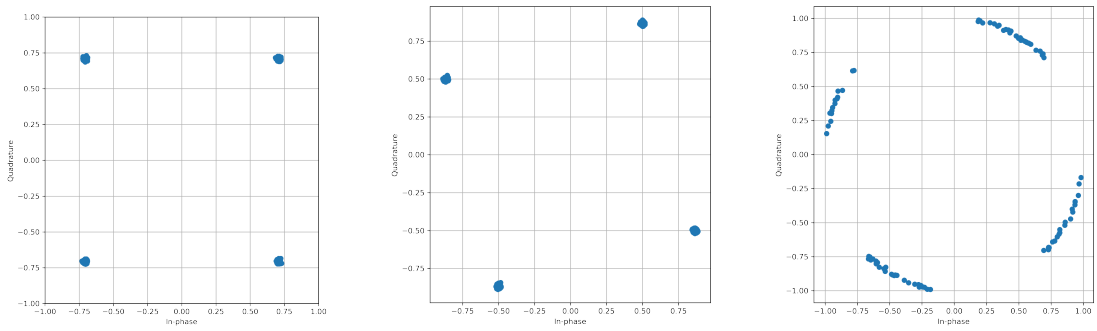
The received signal $r(t)$ with carrier offsets can be described as

$$r(t) = s(t)e^{j(2\pi f_o t + \phi)}, \tag{2.6}$$

where $s(t)$ is the transmitted signal, f_o is the carrier offset frequency, and ϕ is the phase offset.

Recall that, in the case of PSK in Section 2.2, phase offsets of a carrier wave result in a rotation along the unit circle of the complex plane – phase offset manifests the same way and the amount of rotation is defined by ϕ. A constant phase shift is easy enough to correct, however CFO as defined by the offset frequency f_o results in a constantly changing phase shift.

These effects can be easily visualized in constellation diagrams for QPSK, as shown in Figure 2.14. Figure 2.14(b) shows a constant phase offset of 15° – in this case the symbols still fall in the correct quadrants and this would not cause erroneous bits due to phase shift alone. CFO, as shown in Figure 2.14(c), results in a spinning constellation over time, meaning that eventually the symbols will cross over into a quadrant where the receiver will misclassify the received bits. A useful, more practical reference covering these types of impairments can be found in M. Lichtman's book on practical SDR [57].



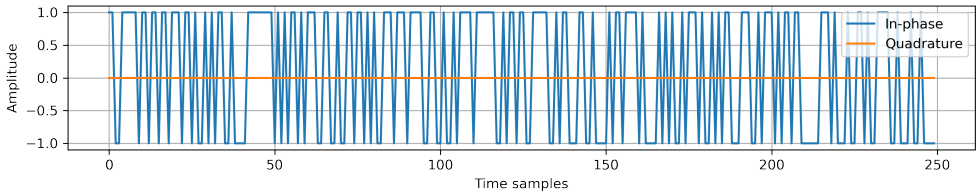
(a) No offsets (b) Phase offset (c) Frequency offset

Figure 2.14: QPSK affected by carrier offsets

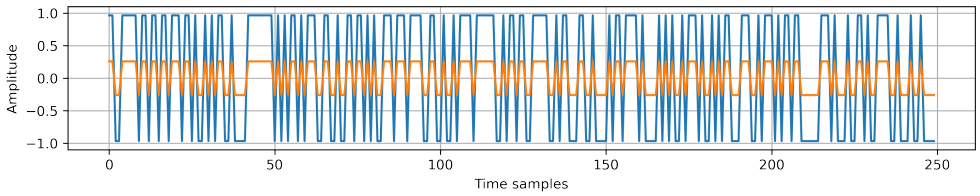
ML models commonly operate on the time domain samples of a signal, and therefore it is important to understand what the samples of a signal affected by CFO will look like with respect to time. Figure 2.15(a) shows a series of BPSK symbols, without any impairment, in this case only the in-phase channel is used to transmit the modulated symbols. In Figure 2.15(b), a modest phase offset is introduced, effectively causing some noise on the quadrature channel. However, with just the phase offset, the in-phase channel symbols are still prominently visible and it is possible to tell by eye that

it is still BPSK.

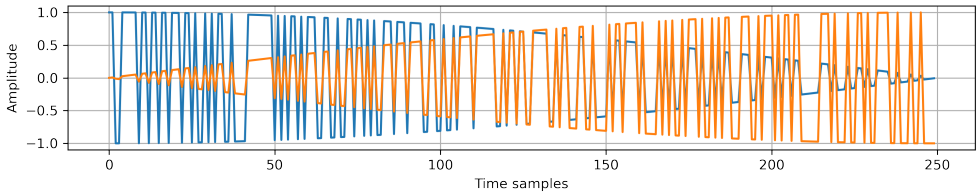
The spinning of the constellation caused by CFO represents itself as a sinusoidal pattern where the in-phase and quadrature channels "bleed" into each other over time, as shown in Figure 2.15(c). From an ML perspective, the drastic effects of this type of impairment (e.g. for a classification system) can be observed by viewing the signal at sample points between 100 and 150, where the real and imaginary samples have roughly the same amplitudes – by looking at just this segment, it would be impossible to differentiate the transmitted BPSK signal from a QPSK capture.



(a) BPSK symbols with no impairment



(b) BPSK symbols with phase offset



(c) BPSK symbols with CFO

Figure 2.15: Illustration of CFO effects on BPSK-modulated symbols in the time domain

Carrier frequency and phase offsets are common in wireless channels, and should be considered when creating any dataset for training an ML model.

2.4.3 Multipath Fading

Many physical phenomena can affect the signals that are transmitted – signals can be reflected by trees and buildings, be partially absorbed by walls, or experience attenuation due to propagation loss. The received signal is the summation of several components that arrive at the receiver at different gains and times, as a result of multiple paths taken between the transmitter and receiver. Such a multipath scenario is illustrated in Figure 2.16.

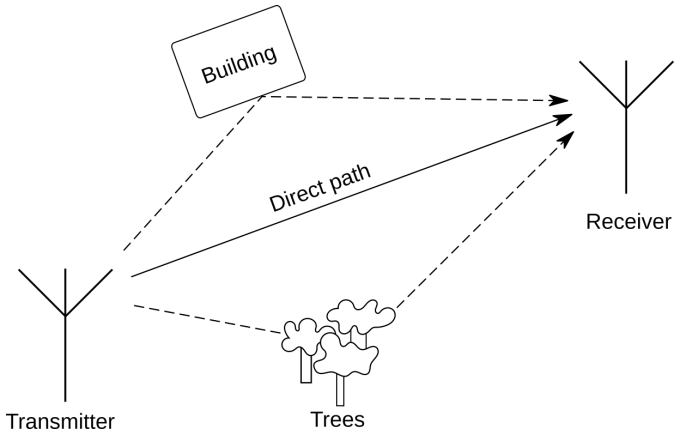


Figure 2.16: Illustration of a multipath channel

If a direct path exists, as shown in Figure 2.16, the channel is called a LOS channel. There can also be cases where the is no direct path from transmitter to receiver, and the strongest received signal may have bounced off a building, a hill or any number of other obstructions – in this case the channel is defined as non-LOS.

A common way to represent multipath channels is with the Tapped Delay Line (TDL) model [58]. Figure 2.17 shows the power delay profile of a multipath channel – where the tap at delay τ_0 is the LOS path, and each subsequent tap is a multipath component. The received signal r as a result of a signal s passing a TDL model is defined as

$$r(t) = \sum_{k=1}^{K} H_k(t)s(t - \tau_k), \tag{2.7}$$

where K is the number of multipath components (or taps), and H_k and τ_k are the

gains and delays of individual multipath components, respectively. It is common for LOS channel path gains to be derived from a Rician distribution, and non-LOS gains to be modelled after a Rayleigh distribution [52, Chapter 14].

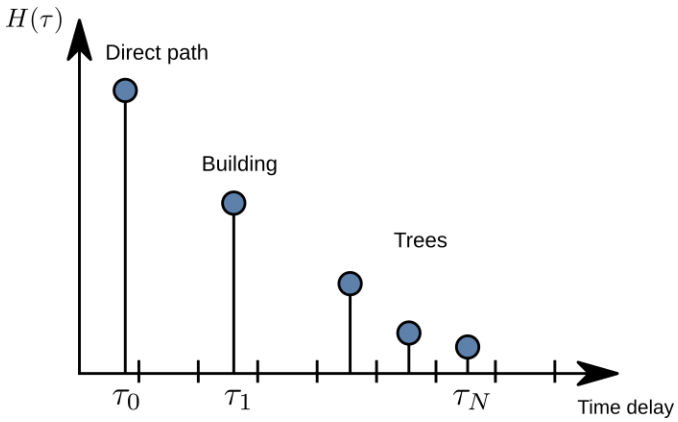


Figure 2.17: Delay taps of a multipath channel model

Channels can be classified as flat and frequency-selective. If a signal of interest occupies the entire bandwidth f as shown as shown in Figure 2.18, it experiences a frequency-selective channel – this is undesirable because deep fades like the one shown in Figure 2.18 result in the signal being distorted to the point where no data can be recovered without channel equalization. On the other hand, if the signal experiences only a flat channel response, then it suffers from a simple complex gain $H(t)$ being applied to the transmitted signal $s(t)$ as defined in Eq. 2.8.

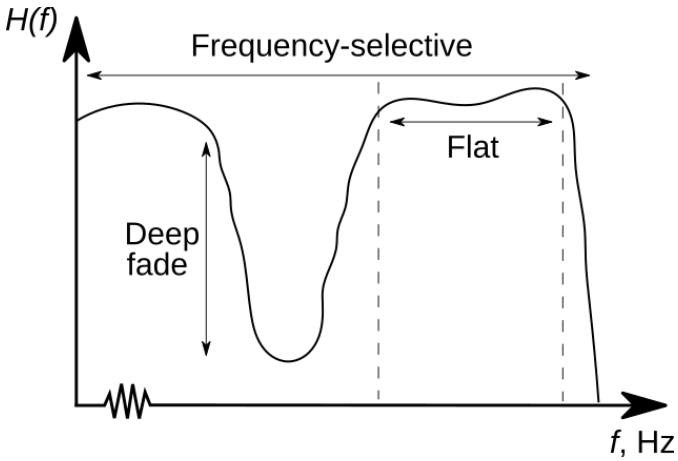
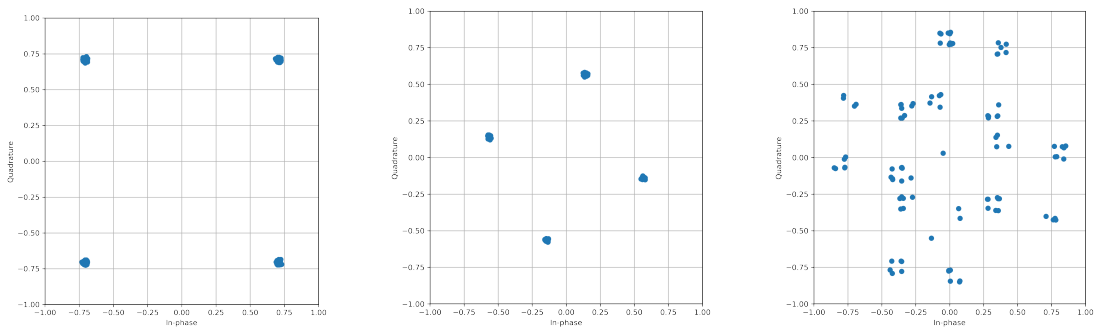


Figure 2.18: Difference between frequency-selective and flat channel responses

$$r(t) = H(t)s(t). \tag{2.8}$$

From a more practical perspective, fading channel effects are illustrated in Figure 2.19. In the case of a flat channel, because the gain is a complex number and multiplication of complex numbers causes a change in both magnitude and phase, the resulting QPSK symbols have a gain and phase shift applied to them – in some cases this may not even result in error, and is quite easy to compensate for. In the case of a multipath channel, Figure 2.19(c), the additional multipath components cause more fundamental changes to the constellation, making it quite difficult to parse by the human eye.



(a) No fading (b) Flat channel response (c) 2 tap multipath channel

Figure 2.19: QPSK affected by fading channels

Multipath fading channels are some of the most difficult impairments to overcome, and therefore there is much interest in techniques like channel estimation and equalization, or multicarrier approaches to mitigate frequency selectivity [52, Chapter 15].

2.5 Automatic Modulation Classification

Automatic Modulation Classification (AMC) is an essential function of an intelligent radio capable of reacting to its environment. The number of signals of various wireless standards is increasing every year, and being able to capture that information is valuable [59] [4].

AMC is an important part of many cognitive radio systems in both user and military applications [60]. As the name implies, its main functionality is to imbue the receiver

with the ability to automatically classify incoming signals. At the time of writing, this task is dominated by DL models, however a brief understanding of how AMC works using classic statistical methods is useful to appreciate the DL solutions.

2.5.1 Higher Order Statistics

As covered in Section 2.2, digital modulation schemes carry data by varying the different carrier properties, of amplitude, phase or frequency. Various statistical features can be extracted by monitoring these properties of an incoming signal. Historically, higher order statistical moments and cumulants have been used to achieve this [61] [62].

Moments

A moment is a statistical measure used to describe the distributions of a given dataset. First order moments, such as the mean and variance are ubiquitous in statistics – these describe the expected value and the average squared distance from the mean of a distribution respectively.

The central moment of a random variable is defined as the moment around the mean (expected value) of the distribution. An n^{th} central moment μ_n can be calculated as follows:

$$\mu_n = E[(X - \mu)^n], \tag{2.9}$$

where $E[]$ is the expected value, X is the random variable, and μ is its mean value. The second central moment is the variance of a distribution $\sigma^2 = \mu_2 = E[(X - \mu)^2]$, in other words the mean of the squared difference between the samples of the distribution and its expected value.

Because moments describe the shapes of distributions, rather than magnitudes, it is useful to standardize these values. A standardized n^{th} central moment $\hat{\mu}_n$, is calculated using

$$\hat{\mu}n = \frac{\mu n}{\sigma^n} = \frac{E[(X - \mu)^n]}{(E[(X - \mu)^2])^{\frac{n}{2}}}. \tag{2.10}$$

The first central and standardized moments are always equal to 0, because the mean around the mean is zero. And since the mean is 0, the resulting variance is 1 – this is usually the case for Gaussian processes.

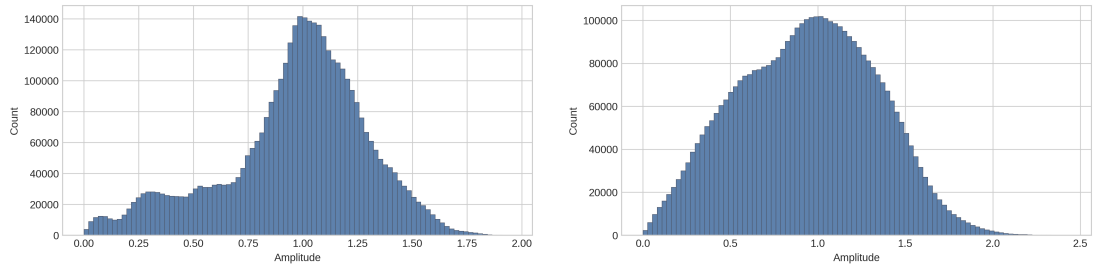
Some more interesting, named standardized moments are skewness (Eq 2.11) and kurtosis (Eq 2.12). Since skewness is an order of 3, it can be negative or positive.

$$skew(X) = \hat{\mu}3 = \frac{\mu 3}{\sigma^3} = \frac{E[(X-\mu)^3]}{(E[(X-\mu)^2])^{\frac{3}{2}}}. \tag{2.11}$$

Kurtosis is a measure that can only be positive and it indicates how the variance is affected by extreme and rare deviations at the tails of the distribution.

$$kurt(X) = \hat{\mu}4 = \frac{\mu 4}{\sigma^4} = \frac{E[(X-\mu)^4]}{(E[(X-\mu)^2])^{\frac{4}{2}}}. \tag{2.12}$$

A good example of kurtosis in wireless communications signals can be observed when comparing QPSK and QAM-16 modulation histograms of the absolute values of their instantaenous amplitudes, as demonstrated in Figure 2.18.



(a) Histogram of instantaneous amplitude of a pulse shaped QPSK signal.

(b) Histogram of instantaneous amplitude of a pulse shaped 16-QAM signal.

Figure 2.20: QPSK and QAM-16 distribution comparison

In Figure 2.20(a), the tail of the distribution of the QPSK signal is much more prominent than for the QAM-16 signal in Figure 2.20(b). 16-QAM looks closer to a normal distribution, which would result in $\hat{\mu}_{4_{qpsk}} > \hat{\mu}_{4_{qam}}$. Kurtosis of the instantaneous amplitude is just one feature that can be used to differentiate different modulation schemes. A collection of these features can be used to create flowcharts or decision trees, where an incoming signal can be algorithmically identified.

Cumulants

Just like moments, cumulants are used to describe the shape of the signal distribution. There are two main advantages to using cumulants – for Gaussian processes all higher order (over 2) cumulants tend to 0, which is helpful in discerning non-Gaussian processes. The other advantage over moments is that the joint cumulant of 2 independent random variables is the same as the sum of the two individual cumulants, this property simplifies statistical feature calculations [63].

Conveniently, cumulants can be calculated directly from the central moments using the recursive cumulant formula

$$K_n = \mu_n - \sum_{m=1}^{n-1} \binom{n-1}{m-1} k_m \mu_{n-m}. \tag{2.13}$$

where K_n is the n^{th} cumulant, $\binom{n-1}{m-1}$ is a binomial coefficient, k_m is the m^{th} cumulant, μ_n is the n^{th} central moment. This equations allows calculating a new higher order cumulant, based on previously calculated moments and cumulants. For example, the 4th order cumulant can be derived from Eq 2.13 as follows:

$$k_4 = \mu_4 - 4\mu_3\mu_1 - 3\mu_2^2 + 12\mu_2\mu_1^2 - 6\mu_1^4. \tag{2.14}$$

If the distribution was standardized to $\mu_1 = 0$ mean, then all of the equations would be simplified, since all the terms with μ_1 would be dropped. For example, the fourth order cumulant from Eq 2.14, can be simplified as:

$$k_4 = \mu_4 - 3. \tag{2.15}$$

The 4^{th} order cumulant is an interesting one, sometimes interchangeably referred to as excess kurtosis. In a Gaussian distribution kurtosis is always 3. The excess kurtosis in a gaussian distribution will always converge to 0.

2.5.2 Decision Trees

Decisions Trees (DTs) are some of the oldest and simplest classifiers. This method is one of the earliest predecessor to AMC systems before ML [64]. A DT works by simply applying a series of condition checks, each one creating two or more branches until a final node is reached indicating the class prediction of the DT model.

Different modulated waveforms have different distributions, as was shown in Figure 2.20. This means that unique modulation schemes will have typical statistical feature values, determined by mathematical theory or simulations – in short, expert knowledge. One branch of a DT can be determined by kurtosis to differentiate QPSK from 16-QAM. Given enough expert knowledge, a more intricate DT can be built that is capable of recognizing a great proportion of modulation schemes.

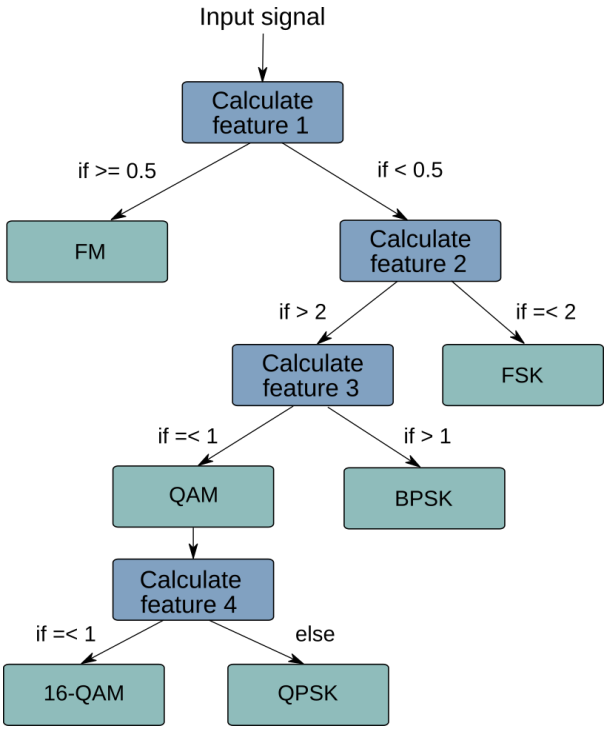


Figure 2.21: Decision tree used for AMC

A high level overview of how an AMC DT might function is illustrated in Figure 2.21. It is common to separate modulation schemes into classes first, then make decisions with finer granularity by evaluating more specific statistical features. For

example, one branch can split the decision of a received signal into broad QAM and not-QAM classes, then further decisions can be made to separate BPSK, QPSK, 16-QAM, 32-QAM, etc.

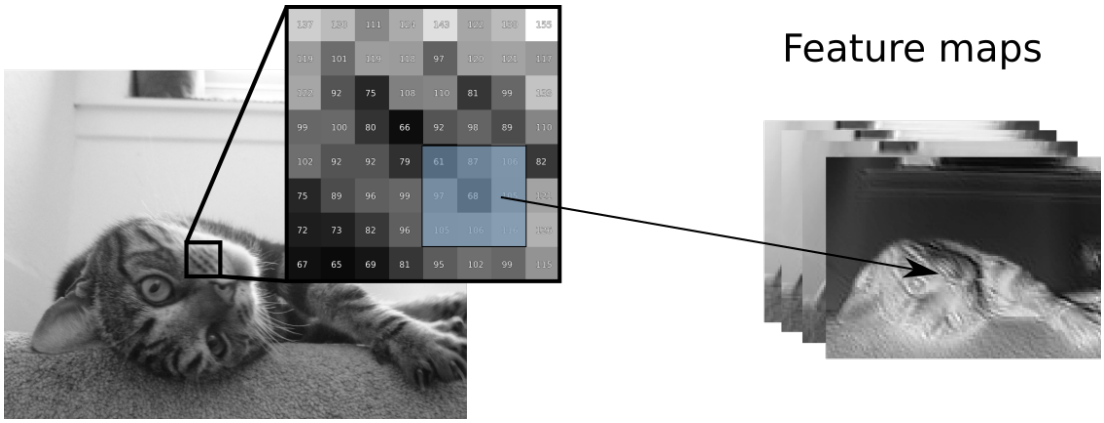
The downside of DTs is that they rely heavily on expert knowledge and require immense human engineering effort to develop the statistical features and DT design. Additional circumstances, such as SNR and other channel impairments, can affect the statistical distributions of the captured waveform and need to be accounted for as well. Fortunately, this task is somewhat easier with ML and off-the-shelf libraries like scikit-learn [65] can be used to automatically produce a DT for a given dataset, simplifying the required human efforts.

2.6 Deep Learning in Wireless Communications

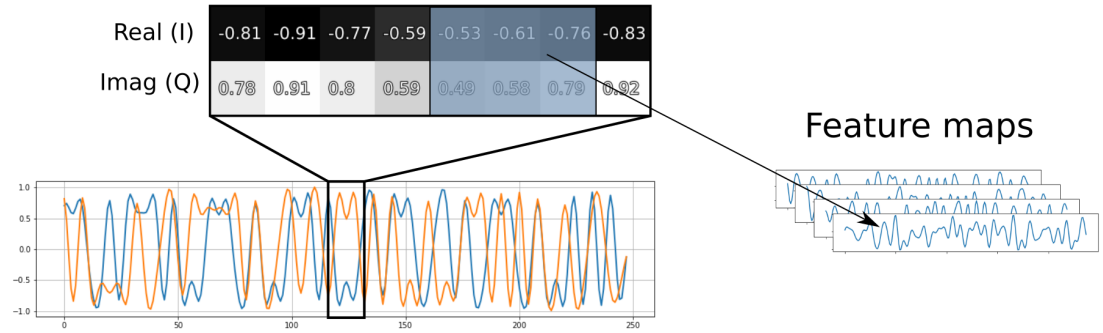
Using ML simplifies the DT approach discussed in the previous section – instead of relying on expert knowledge and carefully selecting the correct thresholds to decide modulation schemes, the features of the signal can be fed directly into a learning algorithm, which can then decide the best thresholds to use in a data-driven way. Taking it a step further, using a DL approach we can forego calculation of the statistical features alltogether, and simply feed a DNN the raw I/Q samples. A big enough DNN would implicitly learn to extract the most relevant features for its given task during training.

As seen in earlier sections, most wireless communications data exists in the form of sequences of complex numbers (I/Q samples). The most common approach of mapping this data for the consumption of ML algorithms is by interpreting it as an image. This can be done by treating the individual I and Q channels as two separate real numbers, much like pixels in pictures – this is illustrated in Figure 2.22. Once converted to matrices of the right shape, the data can be processed by CNNs or any other neural networks that are commonly used architectures in the CV or NLP fields.

Classifying wireless signals involves recognizing cliques or patterns in spatial adjacency within these matrices, much like in CV, where the focus is on detecting patterns and structures in visual data. Borrowing models from NLP and audio domains is also beneficial because concepts like SNR and sample rates are shared with wireless com-



(a) Applying a convolutional layer to an image.



(b) Applying a convolutional layer to a waveform.

Figure 2.22: Comparison of image and wireless comms data for ML.

munications. However, communications signals are not just fixed sized images, they are constantly incoming as a stream of data; phase locked loops need to lock on, frames need to be synchronized, etc. Additional channel impairments as discussed in the earlier sections make dealing with wireless communications signals distinctly unique, and the developed DL solutions should take that into account.

Unique ML Challenges in Communications

Models trained to solve problems in the wireless communications domain will often run into challenges not faced in other fields like CV or NLP. For example, wireless channel impairments are a very unique and interesting problem unique to this field. Training datasets and architectures must account for these effects to ensure performance and reliability.

A particular advantage when in the field of wireless communications when it comes to data – that can be considered a privilege by many ML scientists – is the ability to effectively generate infinite data by using rich simulation software libraries in tools like MATLAB [66] and GNU Radio [67]. With the advantages that come with mature simulation environments in the wireless communications field there come unique challenges and research opportunities in the realm of DL. For instance:

1. **Multirate processing** – many DSP systems found in wireless receivers operate in multirate modes, where the input and output dimensionality can vary. For instance, when demodulating a sequence of pulse shaped symbols, a downsampling operation is required to retrieve the individual bits of the captured waveform. This necessitates development of more flexible model architectures that can accommodate multirate processing.

2. **Robustness** – ML models in wireless communications must be robust against a wide range of variations, including signal distortion, interference, and other channel effects. The robustness of these models is crucial for reliable performance under diverse and often adverse operational conditions.

3. **Latency** – in real-time signal processing, the delay introduced by ML models can be critical. Models must be optimized not only for accuracy but also for low-latency predictions, especially in applications requiring immediate responses, such as making spectrum allocation decisions or signal demodulation.

4. **Cost** – embedded devices, such as wireless receivers often have very strict resource budgets, which necessitates smaller neural network models to be deployed. This means that the trained DNNs must not only meet the performance demand, they must also fit onto an embedded device, which means that optimizing the model architecture for efficiency is paramount.

5. **Dataset tuning** – since effectively an infinite amount of quality synthetic data for training and evaluating the neural networks can be generated using simulation software, this presents the challenges of dataset optimization – what are the best

channel impairments, how many examples should be generated, and so on. The dataset becomes another dial that can be turned to improve model performance, without changing the architecture or the hyperparameters.

Thus, while the dataset is a significant factor in improving model performance, other elements such as model latency, cost and robustness also play crucial roles in the successful application of ML in wireless communications.

2.7 Chapter Conclusion

This chapter introduced the fundamentals of digital baseband modulation and the most common modulation types. Pulse shaping was explained and it was shown how pulse shaping is used to mitigate spectral leakage. Different channel impairments like AWGN, carrier offsets and multipath fading have been presented and their importance highlighted for ML dataset generation. Also covered were some more classic statistical approaches to problems like AMC, for which modern solutions are typically ML dominated.

The main motivation of this chapter is to introduce the essential wireless communications concepts needed to understand how ML fits into solving problems in this area. The next chapter will cover the fundamentals of ML.

Chapter 3

Deep Learning Background

This chapter will review the main terminology and essentials of Deep Learning (DL) required to understand how it can be applied for signal processing and wireless communications problems tackled in the subsequent chapters of this thesis.

3.1 Introduction

DL is a subfield of Machine Learning (ML), which in turn is a subfield of Artificial Intelligence (AI) and consists of methodologies of solving problems in a data-driven approach. This means that, rather than tackling a problem by designing an algorithm based on domain knowledge, a solution can be produced by training a model with a dataset containing a set of inputs and desired outputs. In DL these inputs can be as granular as the raw input pixels of an image or samples taken straight from a radio front-end. By applying a large model and sufficient computing resources, the necessary abstractions required to make the input-output mapping can be learned.

There are multiple ML paradigms in which a problem can be approached, the primary three being: supervised learning, unsupervised learning and reinforcement learning. In this thesis the main focus is on algorithms applied in the supervised learning approach, which is covered in the following sections.

3.2 Supervised Learning

A supervised learning approach deals with labelled data. This means that we have a dataset of inputs x, corresponding desired outputs (or labels) y, and we wish to produce a model that gives the best estimate of $\hat{y}$ by implementing the function $\hat{y} = f(x, w)$, where w corresponds to our model parameters.

When designing a wireless communication system, in simulations, we generally have perfect knowledge of channel conditions and bits that have to be decoded – the trick is to produce a model capable of looking at noisy distorted waveforms and transform them in such a way that the original data can be re-interpreted. Feed the model labelled data, and if enough was provided it will learn to label new entries.

3.2.1 Linear Regression

Linear regression is used to find out some quantity, or estimate a parameter. Regression methods work well for single variable predictions, for example, regression can be used to estimate the SNR of an intercepted waveform. It can also be used to estimate matrices in tasks like channel estimation, where an input could be the received modulated data bits that were perturbed by traveling through a noisy channel and the regression model outputs the channel estimate for that input. Generally linear regression is employed for tasks where the output is not discrete or binary, and there is a certain acceptable range to operate in.

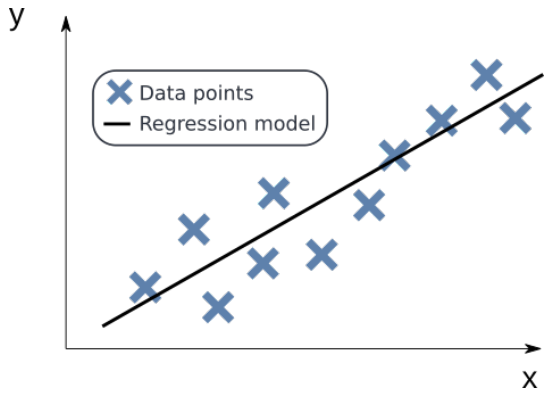


Figure 3.1: Linear Regression

3.2.2 Classification

As the name implies, classification is used to discern between classes. A binary classification task could be used to detect whether a communications channel is occupied or free to transmit on. Classifying inputs into categories is one of the most popular applications in supervised learning. It is also one of the first problems tackled in wireless communications when applied to Automatic Modulation Classification (AMC), which is by far the most popular application of DL in this field.

Classification tasks like AMC are very well suited for supervised learning, because significant amounts of data can usually be generated by using simulation software or collected with Software Defined Radio (SDR) equipment.

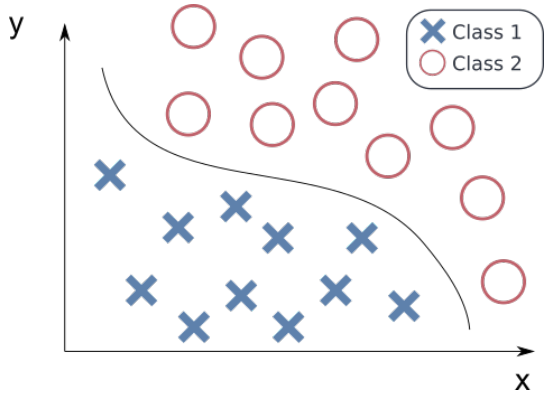


Figure 3.2: Classification

3.3 Neural Network Architectures

When talking about neural networks one should be aware that there exist different architectures, which can be more useful to some tasks than others. This section will cover the basic Artificial Neural Network (ANN) archetypes.

3.3.1 The Artificial Neuron

Before reviewing neural networks, the most basic building block should be covered – the neuron. The artificial neuron takes the entire input vector as an input, then multiplies

every element within the vector by a learnable weight. A single neuron passthrough for output y can be defined as

$$y = \sum_{i}^{N} x_i w_i + b, \tag{3.1}$$

where x is the input vector, w are the corresponding weights, and b is an added bias weight. The bias term is useful as it acts as a threshold for activation. A small bias value requires a strong correlation with the neuron weights to pass the threshold, resulting in higher selectivity. Conversely, a large bias lowers the requirement for the inputs and weights to be perfectly aligned, which increases sensitivity – ensuring the neuron 'fires' more frequently.

3.3.2 Activation Functions

In order to model complex non-linear functions that DL is known for solving, it is necessary for the neurons to be paired with an activation function to introduce some non-linearity to the model. Three of the more popular activation functions are illustrated in Figure 3.3.

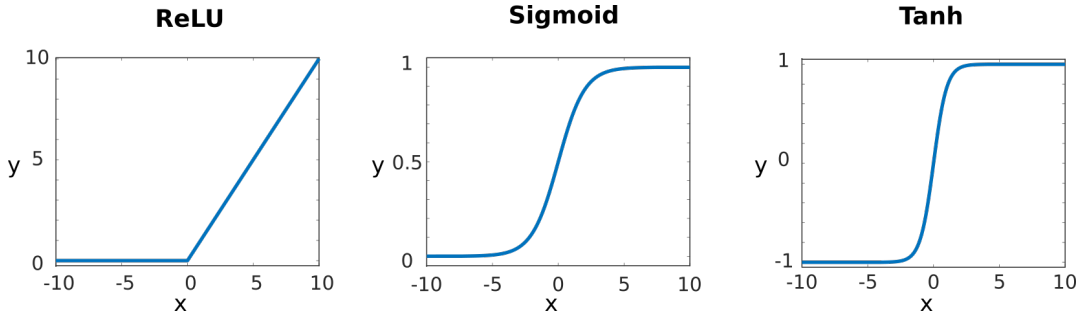


Figure 3.3: Popular activation functions

The Rectified Linear Unit (ReLU) activation function [68] is the de facto standard in many DL libraries, and unless manually specified will likely be the implemented choice for ANN layers. There are a few reasons for its popularity, one of which is the simplicity of implementation, as shown in Eq. 3.2.

$$ReLU(x) = Max(0, x). \tag{3.2}$$

Sigmoid is another popular choice that is used to squeeze any input x to an output value between 0 and 1, which can also be interpreted as a probability output – very useful for binary classification.

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{3.3}$$

More concretely, if we add a sigmoid activation σ to the previous equation, the outputs of our neuron will be squashed between the values of 0 and 1. And the new output equation will look like the following:

$$y = \sigma(\sum_{i}^{N} x_i w_i + b). \tag{3.4}$$

At this point a decision boundary can be defined, say 0.5, and if the neuron output exceeds that value it is considered an 'activation' and can be treated as a positive class. This is essentially how logistic regression works for classification.

There are no hard rules on which activation function is best to use in which architecture or problem type. Like most things in DL, these are determined empirically. While ReLU is one of the most popular options, new variants, such as Leaky ReLU or Parametric ReLU are constantly emerging from research [69].

3.3.3 The Multilayer Perceptron

The Multilayer Perceptron (MLP) network, otherwise referred to as a fully connected network or just a DNN, is one of the simplest neural network architectures in DL. It is not the most common choice for feature extraction, but it will often operate on already extracted features to learn to classify them better than an analytical model would. Very often this type of network is also incorporated into other neural networks, such as convolutional or recurrent networks, to enable classification at the output.

This type of ANN contains an input layer, at least one hidden layer and an output layer. Once you start adding more than a single hidden layer is when the network becomes "deep". The main building blocks required to understand an MLP network are fully connected layers, activation functions and the softmax output.

Fully Connected Layers

As the name implies, in a fully connected network each neuron of a layer is connected to each neuron of the subsequent layer, as shown in Figure 3.4.

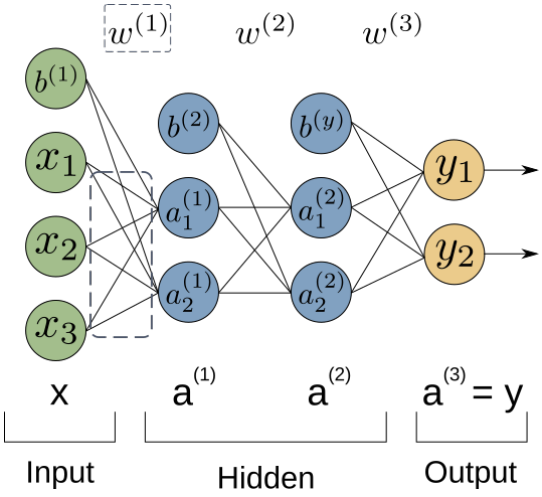


Figure 3.4: Fully Connected Neural Network

Typically, each neuron output is followed by a non-linear activation function, which is crucial for introducing non-linearity into the model. This non-linearity allows the network to learn complex patterns in data, enabling it to solve tasks that linear models cannot. Without activation functions, a neural network, regardless of its depth, essentially remains a linear model. These non-linearities are what enable the network to approximate complex functions and solve interesting problems that go beyond the capacity of linear models.

$$a^{(l)} = \sigma(w^{(l)}a^{(l-1)} + b^{(l)}), \tag{3.5}$$

where w^l and b^l are the weights and biases of layer l, and $a(l-1)$ are the outputs of the previous layer, with σ being the activation function. This type of layer is the integral building block of most neural network architectures.

The Softmax Function

For classification tasks, the output of most neural networks are commonly followed by a softmax activation function. This function takes the outputs of the final hidden layer of a network, and squashes it into a probability distribution, as shown in Figure 3.5. This is useful for human-interpretability, but it also has computational advantages when combined with the logarithmic loss function (this will be covered in Section 3.4.2).

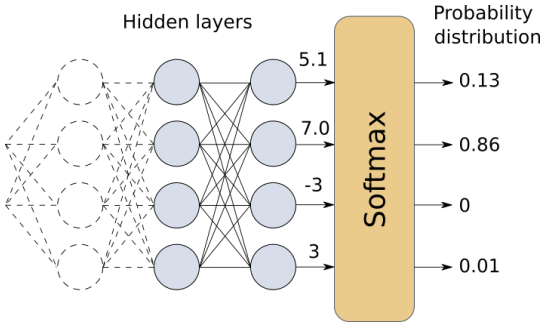


Figure 3.5: Softmax Function

The output of the softmax activation function is calculated as follows:

$$\sigma(Y)_j = \frac{e^{y_j}}{\sum_{k=1}^{K} e^{y_k}}, \tag{3.6}$$

where σ is the normalized output over all possible symbols for that time step. The j index denotes the output neuron number, while K is the number of outputs, and Y refers to the activation values coming from the fully connected layer neurons.

Typical MLP Network Arrangement

A common way this type of network can be arranged is by chaining fully connected layers together, each one followed up by a non-linear activation function, with the final layer followed by a softmax normalization, as shown in Figure 3.6. This is one of the most popular patterns seen in DNN architectures, as it is often added to the final output of the network. MLPs are excellent classifiers when provided with extracted features representing the data, however other layer types are more suited for feature extraction.

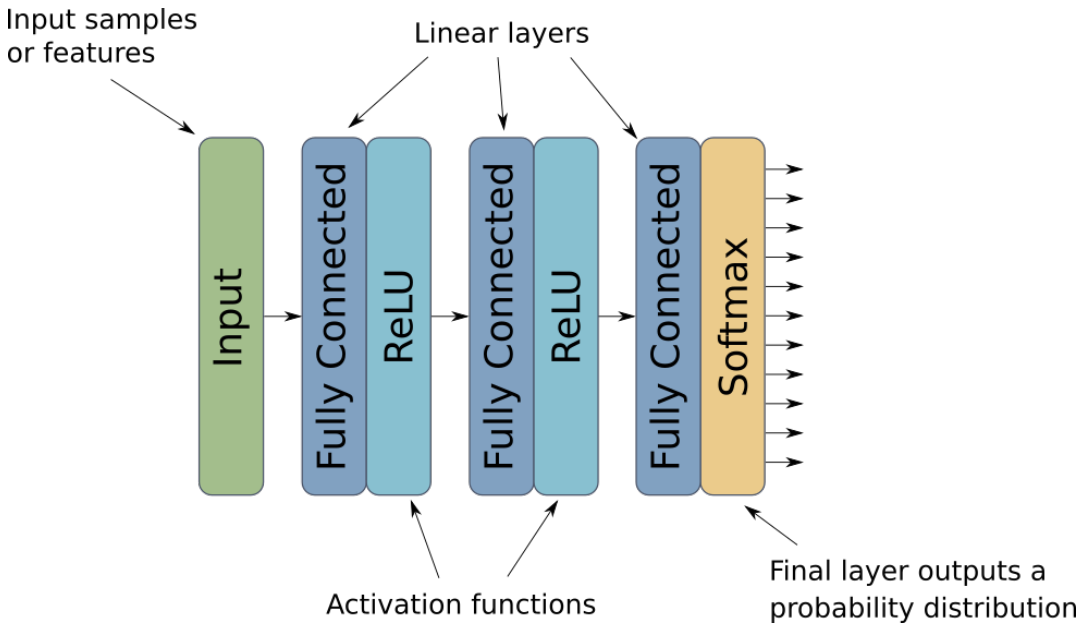


Figure 3.6: Arrangement of a typical MLP network

3.3.4 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are in some regards an improvement over standard MLP type networks and are very popular not only in computer vision applications, but signal processing as well. CNNs are employed in a great array of applications as they have multiple advantages, including a smaller memory footprint due to weight sharing and a local memory map of operation, which makes them adaptable to various shapes and sizes of input [70, Chapter 9].

Convolutional Layers

A convolutional layer is composed of multiple convolutional filters, or kernels, each containing a number of weights that are correlated with the input array, which usually represents an image. A single convolutional layer kernel producing a feature map is shown in Figure 3.7.

As a side note on implementation, while the name 'convolutional' layer implies convolution, most deep learning libraries implement these layers as the cross-correlation function.

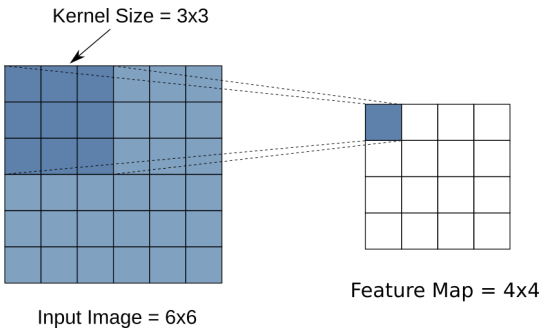


Figure 3.7: Convolution

Max Pooling

Once CNNs become fairly deep it can become costly to implement them in terms of memory required to store the network. Max pooling techniques are a form of down-sampling that reduces the memory footprint and computational cost of the resulting network.

An example of a 2x2 maxpooling operation can be seen in Figure 3.8, where the maximum values of each patch of the feature map on the left is calculated and a new downsampled feature map is created with only the maximum values.

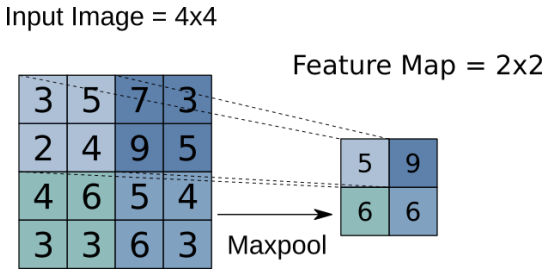


Figure 3.8: Maxpooling operation

Flatten layer

Once the features are extracted from the input data using the convolutional filters, it is common to flatten them into a single vector so that they can be fed into a fully connected layer, which is more suited for classification tasks. The flatten operation can be seen in Fig 3.9.

While most commonly this layer is seen as part of the pattern where a convolutional

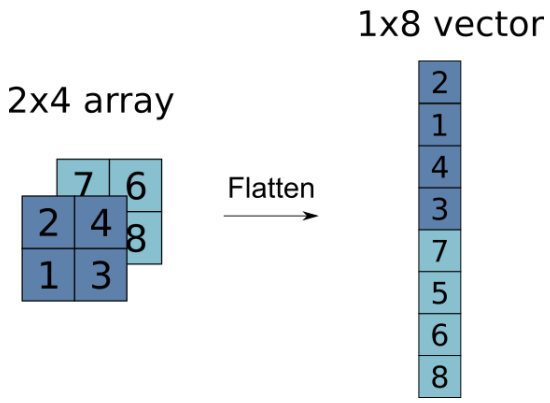


Figure 3.9: Flattening operation

layer output has to be passed to a fully connected layer, it can be applied to any data where an N-dimensional to 1-dimensional transformation is required – e.g. an RGB image can be flattened before being passed directly into a fully connected layer.

Typical CNN Arrangement

Since convolutional layers are very powerful feature extractors they are often used as the first layers of DNN architectures. The most typical arrangement will have convolutional feature extraction layers, followed by pooling layers and then non-linear activations, as shown in Figure 3.10.

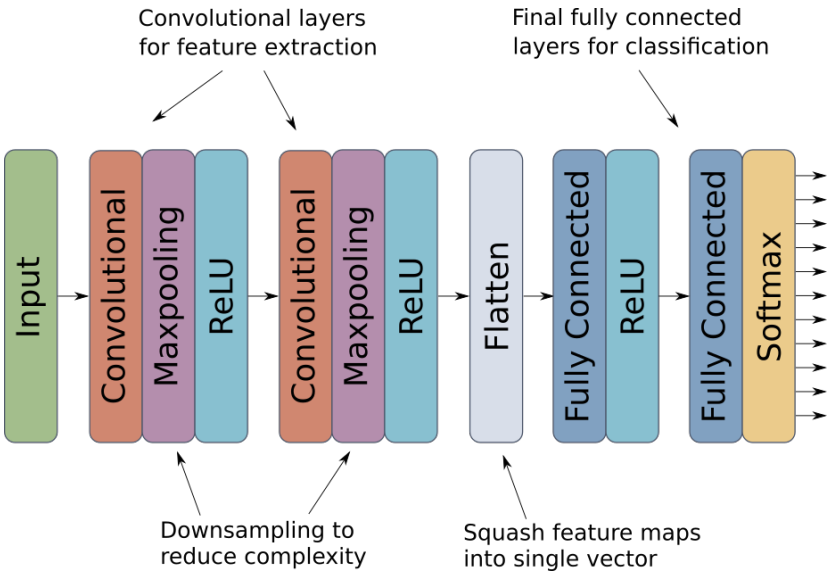


Figure 3.10: Typical arrangement of a CNN model

This arrangement is repeated multiple times and the final feature maps are squashed down into a one dimensional vector using a flatten layer so that the features can be input to further fully connected layers for classification, as was previously shown in Section 3.3.3 for the MLP example.

3.3.5 Recurrent Neural Networks

The Recurrent Neural Network (RNN) type of deep learning is often associated with sequential data, such as audio signals like speech or music, or text in the case of language translation. What differentiates RNNs from CNNs and MLPs is the presence of a hidden state, which is updated after each time step, therefore acting as additional memory for storing context about the input sequence.

RNN cells generally follow a basic update rule for hidden state, h, and output y at time step t, as given in (3.7) and (3.8) respectively, where the W terms represent input and output weight matrices respectively, and b are the biases.

$$h_t = tanh(W_h h_{t-1} + W_x x_t + b_h) \tag{3.7}$$

$$y_t = W_y h_t + b_y \tag{3.8}$$

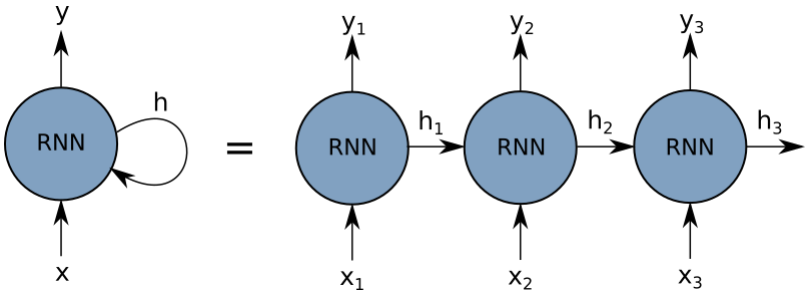


Figure 3.11: Recurrent Neural Network

RNNs have a hidden state which acts as memory, and consequently the outputs produced at time step t do not depend solely on the current input, but on all the past inputs as well. This feature of the architecture makes it a compelling choice for sequence modelling, where the context of past inputs is a useful predictor of the entire signal. An unrolled RNN representation is shown in Figure 3.11.

While RNNs are a very powerful neural network architecture, they do come with some shortcomings. One of these is the difficulty in training long sequences, due to what is known as *vanishing gradient* – when training, the error signals travelling back through the network become incredibly tiny due to constant multiplication by numbers less than 1. This phenomena prevents the network from effectively updating its weights – halting the training process.

Long Short Term Memory

The introduction of the LSTM cell [71] addresses the problem of vanishing gradients by adding gate functions to the original RNN cell and introducing an additional memory cell called the cell state. LSTMs support very long sequences, and allow even more advanced models to be developed. The update rules commonly implemented for LSTM cells [72] are defined as

$$i_t = \sigma_i(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i) \tag{3.9}$$

$$f_t = \sigma_f(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f) \tag{3.10}$$

$$y_t = \sigma_y(W_{xy}x_t + W_{hy}h_{t-1} + W_{cy}c_t + b_y) \tag{3.11}$$

$$c_t = f_t c_{t-1} + i_t tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \tag{3.12}$$

$$h_t = y_t tanh(c_t) \tag{3.13}$$

Here σ is the sigmoid function, i and f are the input and forget gates respectively, c is the cell state, and all the W terms are the corresponding weight matrices, as seen in Figure 3.12.

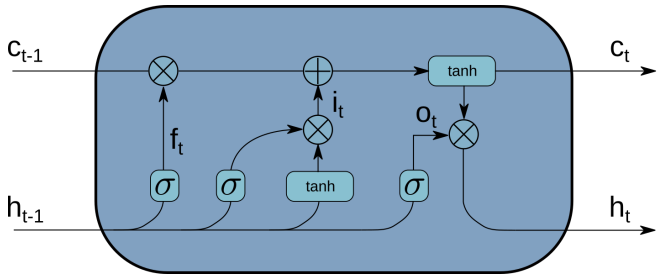
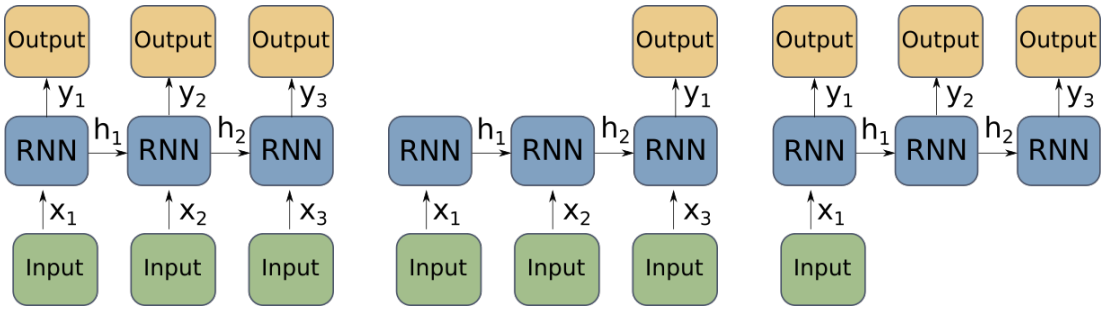


Figure 3.12: LSTM Cell

While the extra equations can make it seem like implementing LSTMs is difficult, DL libraries will typically abstract the complexity of an LSTM, and it will be made available to use as a building block just like a regular RNN cell. In practice, for example in PyTorch, one just needs to be aware that the LSTM hidden state will likely be composed of a tuple of (hidden_state, cell_state), whereas a regular RNN cell will have a single hidden state.

Typical RNN Arrangements

There are many ways of using RNNs and they can be arranged to work on a variety of data formats and lengths [73]. RNNs can be trained to perform many-to-one mappings (Fig 3.13b), which was previously shown for the MLP and CNN examples. When configured in a many-to-one mapping, it can be treated as a feature extractor like a CNN and the hidden state of the RNN can be passed to a classification MLP. However RNNs have other common arrangements, such as many-to-many (Fig 3.13a), where each input produces an output, or one-to-many (Fig 3.13c), where a single input produces a sequence of outputs. The latter two arrangements can be used in tandem for language translation, for example, where one RNN does a many-to-one transformation of a sentence in one language and a second RNN decodes the resulting intermediary representation into another language.



(a) Many-to-many: each time step of the RNN produces an output. This type of configuration can be used in streaming applications, such as video.

(b) Many-to-one: the entirety of the input sequence is processed and mapped to the latent vector space of the hidden state, which can then be passed to other layers.

(c) One-to-many: this arrangement can be used as part of a generative model [72] for creating synthetic data, or predicting next inputs in a sequence.

Figure 3.13: Typical RNN configurations

3.4 Training and Optimization

Regardless of architecture selection, neural networks are trained in much the same fashion. We have data, desired outputs (in the case of supervised learning), and we want our model to learn a meaningful relationship between the two. The training of neural networks is built on two pillars – backpropagation and gradient descent, which will be covered in this section. In the context of ANNs, 'learning' simply means given a training example (e.g. a labelled spectrogram), the neural network will adjust its weights in such a way so that next time it sees the same or a similar data sample it will be able to predict the correct class or label with more confidence.

Machine learning models, such as neural networks, are composed of a number of weights – these are parameters that are learned during training. However, associated with the trained models are a plethora of parameters that need to be set in order to facilitate this training and produce the output model – these are the hyperparameters.

Everything that is not a learned weight, or part of the neural network architecture can be considered a hyperparameter. Some of the more common ones are listed in Table 3.1. The rest of this section will expand on and focus on concepts relating to these hyperparameters.

3.4.1 Loss Functions

Training a neural network requires some evaluation metric to determine how well the DNN is doing at the task in hand. This metric can be used to determine the error, or how close the prediction of the model is to the desired outcome. Selecting an appropriate loss function for the problem is essential – an unsuitable loss function can prevent consistent weight updates of the DNN due to instability or halt the training altogether. There are a few loss functions that will perform well for most tasks, while some will work best for a specific application. Generally, the better a loss function fits the problem, the less tuning of other parameters (such as learning rate or batch size) will be required.

For regression tasks the Mean Squared Error (MSE), defined in Eq. 3.14 is a good

Table 3.1: Selection of hyperparameters

Hyperparameter	**Description**
Loss function	One of the main model performance metrics – the value that has to be minimized when training. Good selection is important to enable, or speed up training.
Optimizer	Optimization is performed using SGD, however in DL many new derivatives of SGD exist, providing advantages in convergence or training speed. Examples are ADAM [74] and RMSProp [75].
Learning rate	The size of the learning steps of the optimizer. Large values result in quicker training, but less stability. Small values can guarantee convergence to a minimum, but can take a very long time to converge.
Regularization	Regularization is achieved using various techniques. Associated hyperparameters can include, for example, regularization factor of weight decay or dropout rate [76].
Number of epochs	A single epoch corresponds to the model being trained on the entire dataset one time. This can be tens, hundreds or even thousands epochs.
Batch size	Number of training examples in a batch. It is inefficient to train on single examples at a time, so training data is usually divided into batches of training data and label pairs.

choice of loss function.

$$\mathcal{L}(y, \hat{y}) = \frac{1}{N} \sum_{n=0}^{N} (y(n) - \hat{y}(n))^2. \tag{3.14}$$

One of the downsides of MSE is that because the input terms are squared, it can give a heavy weighing to outliers in the training data, which reduces the generalization of the model trained with this loss. Another common option for these cases is the Mean Absolute Error (MAE) loss, defined in Eq 3.15.

$$\mathcal{L}(y, \hat{y}) = \frac{1}{N} \sum_{n=0}^{N} |y(n) - \hat{y}(n)|. \tag{3.15}$$

By far the most popular function for classification problems is the categorical cross-entropy (Eq 3.16). It works best when two distributions need to be compared in similarity, and, coincidentally, the softmax layer at the output of most classification models provides the exact form the cross-entropy loss function expects. Doing this

without softmax could make the training significantly harder, as the unnormalized output values from the fully connected layers will result in a bigger loss.

$$L_s(y, \hat{y}) = -\sum_i y_i \log \hat{y}_i. \tag{3.16}$$

The above defined functions are some of the most popular in a variety of fields and DL applications. There are many more, and oftentimes it can be worth optimizing a custom loss metric [77]. However, for exploratory work, choosing MSE for regression and log loss for classification are very sane starting options.

Regardless of the selection, the value given by the loss function will determine how drastically the network should adjust itself to make predictions more closely aligned with the training data. If it made a prediction very far from the ground truth, it will have to nudge its weights more drastically than if it had made the right decision. How does it determine which weights need nudging to make the correct decision? This is where backpropagation comes in.

3.4.2 Backpropagation

Backpropagation is the main enabler of deep neural network training and is the reason why this field has risen to prominence over the last couple of decades. It is the beating heart behind the training of all DNNs. While the loss function provides a metric of how a model of interest is performing, backpropagation is used to update all the weights of the model based on the error resulting from the distance between the estimated model output and desired output, or training label [78]. An overview of the entire training process can be observed in Figure 3.14.

Although its influence is vast, at its core the backpropagation algorithm is quite simple – it is simply the chain rule from calculus, applied to neural networks. This can be exemplified by applying backpropagation to a single logit, a basic classification model consisting of a weight and bias, as shown in Figure 3.15. In an ML context the logit is a broad term, but typically refers to the unprocessed output of a neuron or logistic classifier (from logistic regression).

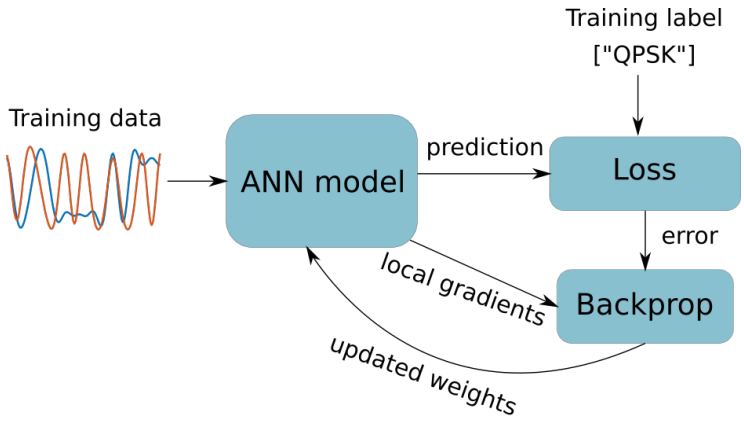


Figure 3.14: Training Overview

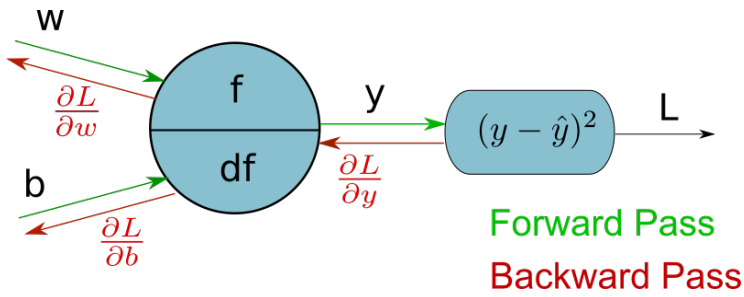


Figure 3.15: Backpropagation on a single logit and MSE loss

This logit has a single input x, weight w, bias b and activation function f, which outputs a value y, with the ground truth label being $\hat{y}$. The forward pass simply involves multiplying the inputs and passing them through the activation function $y = f(wx+b)$. The loss is then computed as $L = (y - \hat{y})^2$, which is the error signal that is then propagated backwards during backpropagation to adjust the weight and bias.

The main goal of the backward pass is to compute the partial derivatives of the tunable parameters, in this case $\frac{\partial L}{\partial w}$ and $\frac{\partial L}{\partial b}$. This can be done by computing the local gradients of the forward pass and applying the chain rule with the backpropagated gradients, as shown in Eqs 3.17, 3.18.

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial w} \tag{3.17}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial b} \tag{3.18}$$

By computing the partial derivatives with respect to every single weight, it can be determined which weight has contributed the most to the error signal given by the loss function output at the end of the computational graph.

Once the gradients are known, stochastic optimization methods, such as SGD, can be used to determine how much the weights should be nudged in the direction of the gradients.

3.4.3 Stochastic Gradient Descent

While backpropagation allows the computation of gradients that tell us how much each weight has contributed to the loss, SGD drives the training by allowing iterative updating of the weights to better fit the training set. This is also where one of the main hyperparameters – the learning rate α – is applied, which helps control exactly how much the weights should be changed.

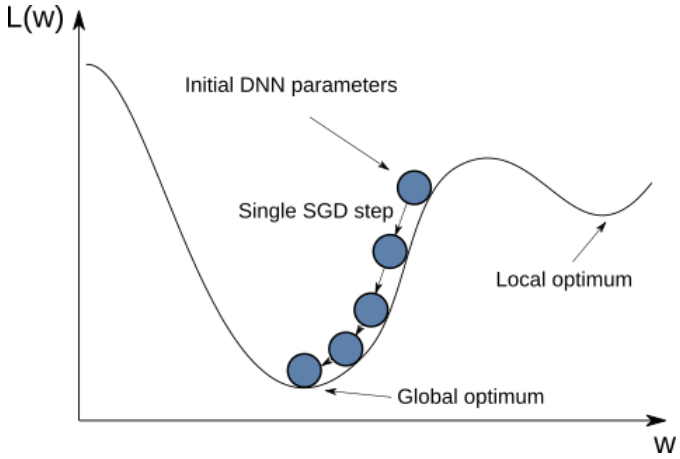
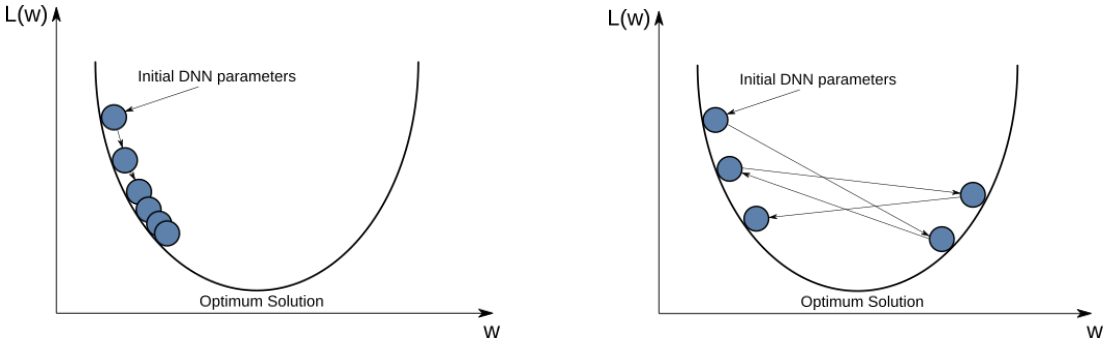


Figure 3.16: Illustration of SGD

Assuming the computed gradients from backpropagation can be represented as ∇w, the new weights can be computed by subtracting these gradients scaled by the learning rate α from the old weights, as seen in Eq 3.19. Subtraction is used so the change is in the opposite direction of the error gradient.

$$w_{(new)} = w_{(old)} - \alpha \nabla w. \tag{3.19}$$



(a) Learning rate too low

(b) Learning rate too high

Figure 3.17: Learning rate impact on convergence

Ideally the loss function, given some ANN parameters, will result in a convex plane, where the goal is to find the ideal parameters w, that will minimize the loss function $L(w)$, reaching a global optimum, like shown in Figure 3.16. The initial model will start somewhere random – technically it is possible to get extremely lucky and initialize a network at the optimum location, however this is very unlikely. As the network is fed training data, and its error gradients are computed using backpropagation, we use SGD to update the weights in the opposite direction of the error gradient, which makes it move toward the desired minimum.

Success assumes that a good learning rate α has been chosen. Too small of a value may move it towards the desired minimum, but it may take a very long time to actually converge, or converge at a non-ideal local minima (Fig 3.17a). Choosing a value that is too large can make it move towards the minima faster, but it is also likely to overshoot and never actually converge to a good solution (Fig 3.17b).

3.4.4 Regularization Techniques

Regularization is an important part of the training process introduced to combat overfitting. In order to create a robust model for deployment in the real world, just monitoring the training loss may not be enough, because DNNs, especially large ones, are susceptible to overfitting. Overfitting can be thought of as the model memorizing the entire training set, rather than learning the features required to identify the classes in the training set. This leads to the newly trained model not being able to generalize to new

problems, because it learned, for example, the specific pixels of a spectrogram image, rather than how to recognize a signal in the image. There are multiple regularization techniques available, of which a few will be introduced in this section.

Validation

To understand regularization, we should first understand the metrics often used to evaluate it during training. The first step of making sure the model is robust is to implement validation loss monitoring. This is typically achieved by splitting the training dataset into two portions, say a 80/20 split, where 80% of the data is used for training, and the remainder is used to evaluate the performance, but does not backpropagate those evaluations. This is not the only validation strategy, methods like cross-validation are also regularly used in ML, but a validation split is very common in DL.

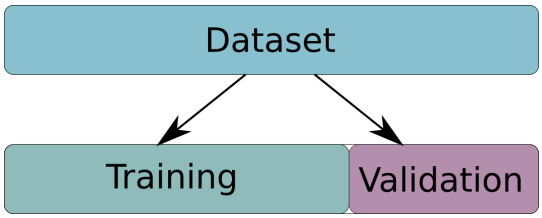


Figure 3.18: Training and validation split

Early Stopping

The easiest way to make sure the model is not overfitting is to monitor the validation loss, and, once it stops improving, halt the training. A typical training/validation loss graph is illustrated in Figure 3.19. It can be seen that the training loss in this case keeps decreasing over time to near 0, which generally would be great – the network is learning to perfectly predict everything it sees in the training set. However, if this model were to be deployed in the real world on data that has not appeared in the training dataset exactly, it would fail miserably. If we looked at the validation loss it would be evident that, past a certain point, the network started losing its ability to generalize to new data points. This is because in order to minimize the loss, sufficiently large DNNs can start learning their training data samples, including the noise present

in the dataset, which is undesirable.

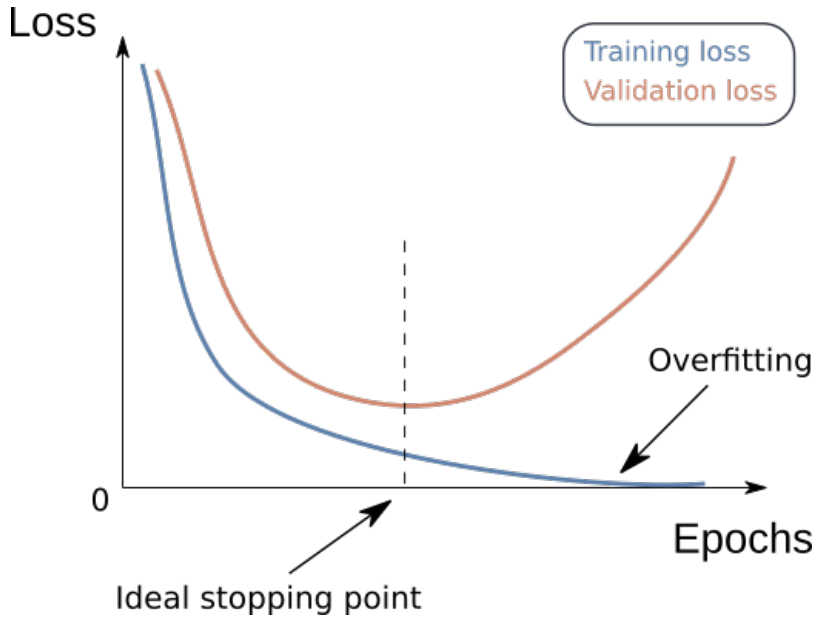


Figure 3.19: Loss over time

Ideally the DNN would learn the core statistics required to make the desired predictions and generalize to different noise distributions. This is why it is important to not just monitor the training loss, but have a separate validation set to make sure that the network is not overfitting.

Weight Decay

Weight decay, or L2 penalization, is a simple form of regularization that can be implemented by modifying the loss function. This is done adding the sum of all weights of the model, scaled by a regularization factor λ, to the loss, as seen in Eq. 3.20. In this case the loss is the previously defined MSE, but with an additional regularization term.

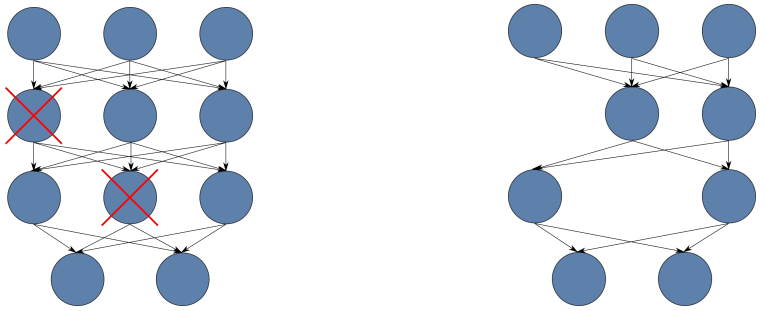
$$L_{reg}(y, \hat{y}) = \frac{1}{N}\sum_{n=0}^{N}|y(n) - \hat{y}(n)|^2 + \frac{1}{2}\lambda\sum_{i=0}^{N_w} w_i^2. \tag{3.20}$$

This regularization term penalizes the network for using weights that are large. The reason large weights are undesirable is because they might give a heavy weighing to a single feature for which that one weight is responsible, rather than using all of the

features available. Generally we want to keep the weights uniformally distributed.

Dropout

Dropout is a very popular technique, often used for large DNNs [76]. During training it drops random neurons from the computational graph and forces the DNN to adapt and use other neurons for the prediction, as shown in Figure 3.20. This, again, decreases the reliance on a single neuron responsible for one strong feature dominating the decision making of the entire model and forces the DNN to use all available weights.



(a) DNN before dropout (b) DNN after dropout

Figure 3.20: Dropout on a small ANN

3.5 Chapter Conclusion

This chapter covered the basic concepts of supervised learning and DL techniques for training DNNs. Some of the most commonly used layer types have been covered and it has been shown how they can be combined for regression and classification tasks. Training techniques with regularization were also covered and are a key part of engineering robust neural network models. These patterns will keep appearing in the following chapters when new architectures and training methods for wireless communications applications are discussed.

The next chapter will address the problems of baseband demodulation and AMC using RNNs.

Chapter 4

Sequence to Sequence Learning for Demodulation

RNNs are extremely powerful models capable of learning complex non-linear functions while operating on time series data. The unique property of RNNs, compared to other DNN architectures, is the presence of a hidden state – the ability to maintain memory for storing context on the input sequence being processed. This type of neural network has impressive malleability and can function as a building block for various models that are useful in DSP.

Additionally, RNNs can be deployed on sequences of variable length, because they process inputs on a sample-by-sample basis as opposed to operating on fixed-size frame inputs like fully connected layers. This flexibility is very useful in cognitive radio (intelligent radio systems), where a single smart receiver may need to recognize and process different wireless standards necessitating adaptability to various packet lengths.

This chapter covers the usage of RNNs as building blocks for a Sequence-to-Sequence (Seq2Seq) autoencoder model that can learn various wireless communications tasks by processing raw baseband I/Q samples. By learning to transform baseband samples into a stream of bits at the output, the model is shown to implicitly learn modulation classification, matched filtering and digital baseband demodulation. The resulting trained models also show robustness to noise, performing well at a variety of SNR levels.

The difficulty in training these models is discussed, and some alleviating techniques

proposed – such as reducing the burden on the RNN encoder by adding convolutional layers for feature extraction.

4.1 Motivation

For decades, radio receivers have been developed on an individual module basis – that is, each function of a receiver, be it a matched filter, timing synchronizer, or channel estimator, is designed and tested as an individual unit. This is fundamentally a local minimum problem within a larger system. An intelligent wireless receiver should have the capacity to learn the multiple signal processing functions required to identify and extract the information from an incoming signal as a *single module*, making it a global optimization problem.

RNNs have shown a lot of promise in their versatility and expressive power when modelling a variety of problems. When arranged in a many-to-many configuration it can be considered an autoencoder (Seq2Seq) architecture. This type of architecture has seen significant usage in sequence data, such as translation tasks from one language to another [27]. Taking an input vector representing a sentence in one language, an encoder processes each word and compresses its meaning in its hidden state. A separate decoder is then used to interpret this hidden state and decodes the meaning as a vector representing the words of an entirely different language.

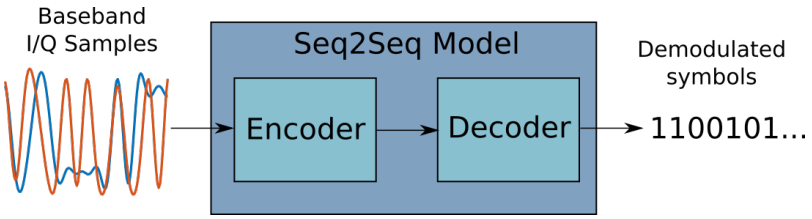


Figure 4.1: Wireless receiver as a Seq2Seq model

The same encoder-decoder principle can be applied to designing a receiver for wireless transmissions, as shown in Figure 4.1. After all, incoming I/Q samples are an example of sequential time series data, just like words in a sentence. The sequence lengths are also variable in both cases – wireless transmissions can contain short and long packets, and different standards will use different modulation schemes, upsampling

rates and synchronization sequences. For this reason, an ideal receiver would be able to cope with a variety of input sizes and conditions. RNNs are capable of operating on inputs of arbitrary length, which makes them a compelling candidate for such a receiver.

4.2 Related Work

There exists a significant body of existing research on using DL for digital demodulation. MLP-based demodulators for ASK, QAM and CPM have been suggested as early as the 90s, and shown to work well in an AWGN channel with robustness to phase offsets [79], [80], [81]. The use of MLP structures was also explored for BPSK, QPSK, and 8-PSK in [82], where each MLP worked on fixed 16 sample inputs. An MLP-based design for Frequency Shift Keying (FSK) demodulation was explored in [83], [84], both works operating on 21 sample inputs. Later, the same authors added a complex-valued hidden layer in their MLP structure, which allowed more expressibility [85]. BPSK demodulators based on an MLP have also been shown to work on over-the-air data [86], with the MLP consuming 10 fixed samples per inference.

Moving towards more complex architectures, Deep Belief Networks (DBNs) have been utilized to handle BPSK and QPSK modulation schemes in underwater acoustic communication on 30 sample inputs [87]. For the purposes of this discussion, DBNs are similar to MLPs in that they contain fully connected layers constraining the input size of the model. DBNs were also used to demodulate BPSK for short-range multipath channels [88]. A joint DBN-SVM approach was employed for QAM demodulators, on over-the-air transmissions at various modulation levels from BPSK to 256-QAM, where the DBN was used for feature extraction and an SVM model for classification, combining DL and a more traditional ML approach [89].

CNNs have been successfully applied to extract symbols from transmissions containing mixed signals, where simultaneous transmissions of BPSK, 2-ASK and 2-FSK were interfering with each other, showing that a DNN approach can effectively suppress interference noise [90]. The input size of this model was 32 samples. CNNs were also utilized to demodulate Binary Frequency Shift Keying (BFSK) in multipath chan-

nels [91]. A CNN-based demodulator was demonstrated for BPSK, QPSK and 8-PSK, processing a fixed-size 4096 sample frames using an autoencoder for feature extraction, then using individual 1024 individual small scale classifiers to recover the symbols [92]. Another CNN-based model was developed to demodulate 4096 sample sequences of 64-FSK signals for the JT65A standard [93].

On the RNN front, an LSTM-based RNN was shown to demodulate continuous FM speech signals, achieving superior performance to conventional methods [24]. A combination of CNN and RNN methods was proposed for demodulating BFSK, QPSK, and 16-QAM signals [94]. The CNN and RNN were both operating on the 100 sample input, then their outputs combined into a single vector and fed into a classification MLP. An LSTM demodulator capable of classifying and demodulating BPSK, 2-FSK, and 2-ASK was demonstrated by the authors of [95]. Their LSTM architecture used a regression output, meaning that the implementation should be capable of operating on variable length sequences, however only the single 800 sample input length was explored. Most recently an RNN-based demodulator for QPSK and 16-QAM was investigated in [96], operating on 10 sample input sequences. It showed improved performance in fading channels, however only using the RNN for feature extraction and leaving classification to a final fully connected layer.

A variety of DNN architectures have been proposed over the years for digital demodulation. They are typically optimized to specific modulation types, and often require fixed-size inputs. A common trend across the reviewed works, with the exception the few RNN-based implementations [24], [95] is that most of the reviewed DNNs emitted a single symbol prediction as a classification output.

The LSTM-based Seq2Seq architecture developed in this work was first demonstrated for BPSK and QPSK classification and demodulation in [97]. To the best of the author's knowledge, this was the first instance of a Seq2Seq model being applied for radio physical layer problem like demodulation. The work presented in this chapter aims to address the input dimensionality limitations of the reviewed architectures and further enhance DL-based demodulation efforts by including AMC into the model.

4.3 Seq2Seq Model for QPSK Demodulation

This section introduces how the Seq2Seq architecture is implemented for processing the raw I/Q samples of digital baseband signals. The first task that the Seq2Seq autoencoder is trained on is QPSK demodulation. The model trained in this section will be used as the base for more complex additional tasks, like AMC, in the rest of this chapter.

4.3.1 Baseband Demodulation Task

The first proof-of-concept task for the Seq2Seq architecture is processing a sequence of pulse shaped QPSK symbols and demapping (or classifying) the symbols. Baseband digital modulation is the mapping of data bits to symbols according to the specified baseband modulation scheme. In order to avoid spectral leakage, the symbols must be band-limited by a pulse shaping filter. As covered in Chapter 3 (page 47), RRC filters are commonly used for matched filtering in practical systems, allowing less complex implementations than if an RC filter had been implemented at the receiver instead. Due to the ubiquity of RRC filters in wireless systems, the following experiments use a Root Raised Cosine (RRC) filter to generate the baseband modulation datasets. The goal of the Seq2Seq model is illustrated in Figure 4.2.

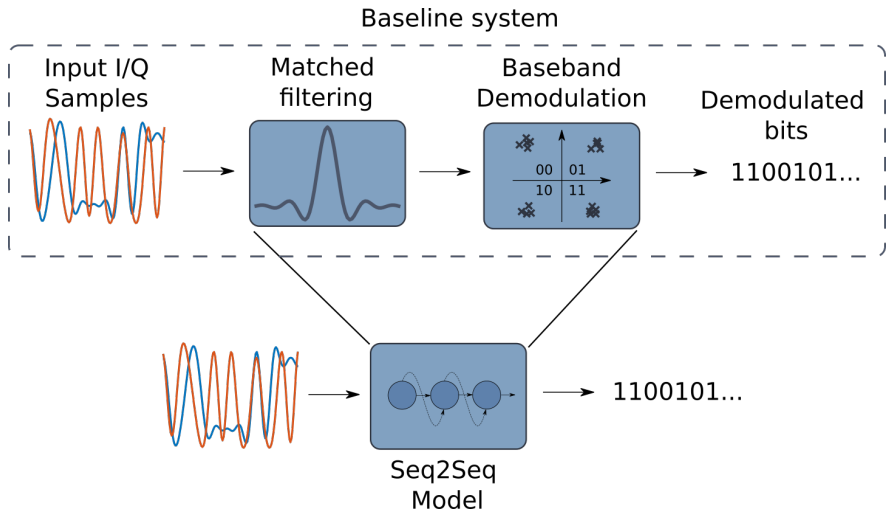


Figure 4.2: QPSK Baseline Model

The next sections will explore the architecture of the Seq2Seq model and data formatting requirements to train the model and perform predictions.

4.3.2 Data Formatting

In terms of ML, the act of symbol extraction can be considered a classification problem with M classes, where M is the number of possible received symbols. For QPSK, M = 4, where each symbol $M \in \{S_0, S_1, S_2, S_3\}$ can be represented as one-hot encoded vectors such as:

$$\begin{bmatrix} S_0 \\ S_1 \\ S_2 \\ S_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.1}$$

For example, in Figure 4.3, a pulse shaped QPSK waveform with an upsampling factor of 4 is shown. Each symbol label corresponds to 4 samples of the waveform, meaning that 6 symbols can be represented by a sequence of 24 I/Q samples or a 2×24 matrix (where in Figure 4.3 the blue dots are real and red dots imaginary channel samples respectively). An example sequence of 6 one-hot encoded labels to go along with the waveform shown in Figure 4.3 would be represented by a 6×4 label matrix.

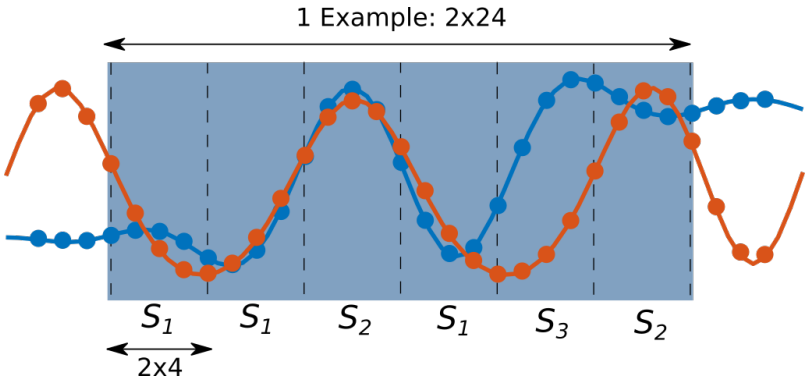


Figure 4.3: QPSK training waveform snippet

4.3.3 Architecture

An overview of the full encoder-decoder configuration is shown in Figure 4.4. The encoder and decoder must share the same cell sizes, however the number of iterations they compute is independent to one another. Each input sample x represents a complex sample (a vector containing 2 real numbers representing the I/Q channels) in a baseband QPSK modulated signal. Note that an RNN may contain an N number of layers (LSTM/GRU/basic RNN cells) stacked on top of each other, increasing the complexity and expressibility of the model.

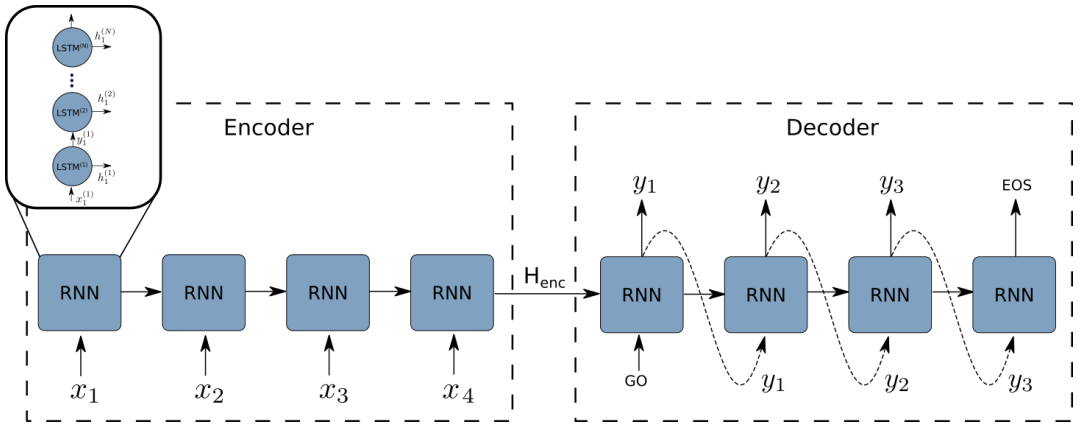


Figure 4.4: Sequence to Sequence model

Encoder

The encoder is an RNN made up of N layers of stacked LSTM cells – the cell size and network depth in terms of number of stacked layers determines the network memory capacity. The example encoder in Figure 4.5 demonstrates an RNN composed of 2 LSTM cells. The job of the encoder is to process each incoming sample to a latent space representation inside its hidden state. The values inside the hidden state will contain the context and information necessary to determine how many symbols have been received and how they should be demodulated.

At each time step the encoder network is fed a sample input x, composed of the real and imaginary components of the received baseband radio signal. The hidden state of each stacked LSTM cell is updated after every step, and once the last sample is

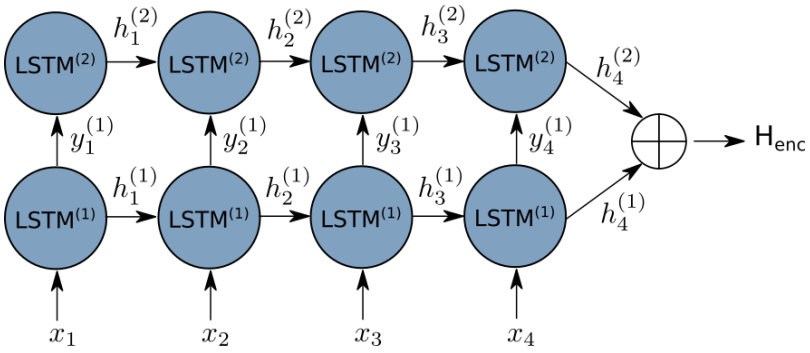


Figure 4.5: Two layer encoder structure

processed, the final hidden states are concatenated to be passed off to the decoder. The outputs of the last encoder layer are disregarded, as the only purpose of the encoder network is to accumulate information about the incoming waveform in its hidden state. This information, compressed in the encoder hidden state, is then passed to the decoder.

Decoder

The decoder has to unravel the hidden state of the encoder by outputting the best next symbol label predictions based on all the information accumulated in the hidden states of the encoder cells. The decoder cells must be of the same size as the encoder cells (to allow hidden state sharing), however the number of output steps need not correspond to the number of iterations required to encode the input waveform. Each predicted output is fed back into the decoder as the input of the next time step to assist in predicting the next symbol, as shown in Figure 4.6.

The Start of Sequence (SOS) or GO token can technically be represented by any values in a vector of the same dimensionality of the cell input. In this case it is simply represented as just a vector of 4 zeros to differentiate it from the other one-hot encoded symbols it will be generating – the network needs some initial input. Once the GO token has been passed, the network can continue outputting symbols indefinitely. Typically it is trained to output an End of Sequence (EOS) token once the sequence end has been reached. It can also run for a predetermined number of outputs without the use of an EOS token, which is how the decoder is implemented in this work.

Since demodulation is treated as a classification task, during training, each output y

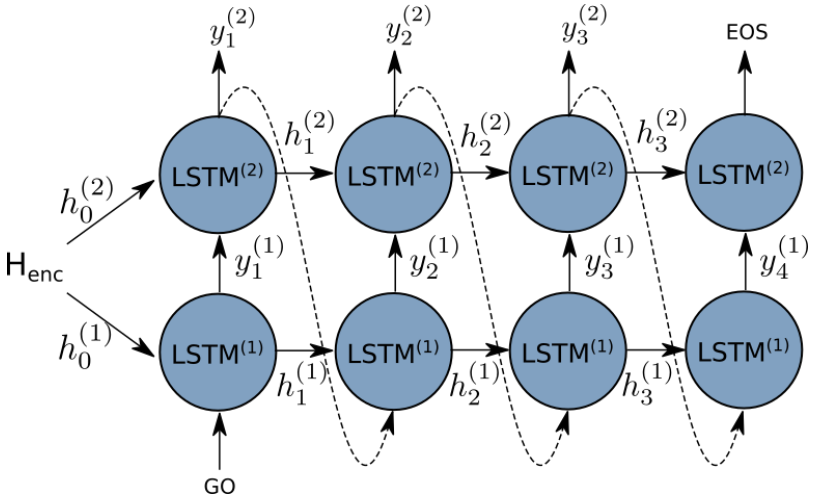


Figure 4.6: Two layer decoder structure

goes through a softmax layer where a probability distribution over all possible symbols is generated for each decoder time step, which can then be fed into a crossentropy loss function for optimization.

4.3.4 Training

Determining the model architecture that will be used to tackle the problem is only the first step – and even then, the model will have to go through multiple iterations of the model architecture parameters to reach a satisfactory final network. A DL workflow consists of many iterations and tuning of parameters empirically based on metrics like accuracy and training/validation loss. There are 3 major configurations that require tuning:

- Dataset – finding a representative training set size, and determining a good SNR level for training.
- Model parameters – determining an appropriate number of layers, and how large should cell sizes should be.
- Optimization parameters – deciding on a good loss function, learning rate, number of epochs, regularization and whether teacher forcing should be used.

Iterating over model and optimization parameters over a given dataset is a very common workflow in DL. A unique addition in RFML is the addition of dataset parameter tuning: here the dataset also needs to be generated and there exists endless possibilities in simulating various dataset configurations.

Dataset Parameters

Before embarking on training any DNNs, it is generally a good idea to take time to understand the dataset and look at any available baseline models. The problem in this case is rather simple and a baseline model consists of a pulse shaping filter and a QPSK demodulator. The baseline system accuracies are plotted in Figure 4.7. Accuracy, instead of BER plots on a logarithmic scale, are used here out of convenience – classification tasks are usually evaluated by measuring the classifier accuracy (which is just the inverse of error rate).

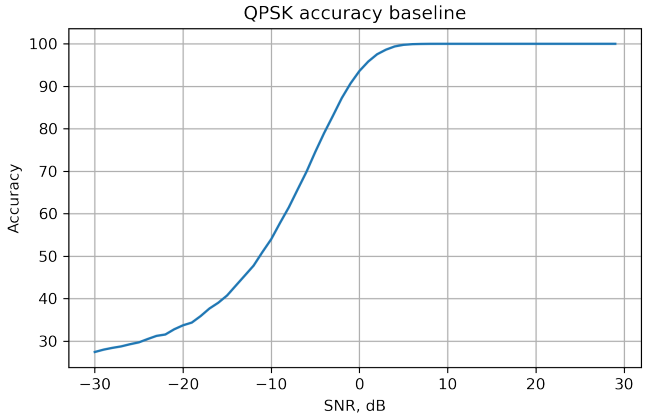


Figure 4.7: Baseline accuracy of a QPSK receiver over an SNR range

A good starting SNR for a dataset should not be too low, because then the model would just learn noise unnecessarily. It should also not be too high, because it will never learn to be robust to unseen channel effects [98]. The sweet spot can usually be found where the baseline model begins to saturate at the maximum accuracy value – in the case of the baseline QPSK receiver in Figure 4.7, an approximation of 0dB SNR can be made.

The training dataset parameters are summarized in Table 4.1. Any number of

training sequences could technically be generated, however eventually this reaches a point of diminishing returns if the model architecture remains constant [99]. While it depends on the problem, for small scale architecture exploration and understanding the model, a few thousand examples are enough, and in this case 8192 training examples are used to train each model – this was found to be a satisfactory number of examples on for training a variety of models investigated in this thesis.

Table 4.1: Dataset parameters

Parameter	**Value**
Number of sequences	8192
Samples per symbol	4
Symbols per sequence	5
RRC filter weights	65
SNR	0dB

Each QPSK modulated sample sequence consists of 5 modulated symbols, pulse shaped using an RRC filter with an oversampling factor of 4. The entire dataset is kept at a single, relatively low SNR of 0dB, which as the baseline model has shown, should be enough for the Seq2Seq model to achieve at least 90% accuracy at that SNR. The entire dataset was generated from scratch using Python, the one-hot encodings for the labels were produced using a simple dictionary conversion by mapping symbols to vectors.

Model Parameters

After generating a reasonable training dataset, the different model parameters can be explored. For RNNs the main parameters to consider are the cell sizes (hidden state) and number of layers in the encoder/decoder. Obviously larger and deeper models are preferable as they can learn to solve more interesting problems and have longer memories. The tradeoff is that large models become more difficult to train and costly to implement, which can be a significant constraint in an embedded system.

The training parameters used to investigate an optimal Seq2Seq configuration for QPSK demodulation are summarized in Table 4.2. Standard training parameters are used, such as the ADAM optimizer [74] with the default learning rate of $1e-3$. The

chosen batch size was the default Pytorch size of 32, which has been proven to be an optimal choice for many problems [100]. A sweep of architecture parameters is performed by training 5 models for each combination of cell size and number of layers. Models are trained for 100 epochs each, then their accuracies evaluated over a test SNR range from -30 to 30dB.

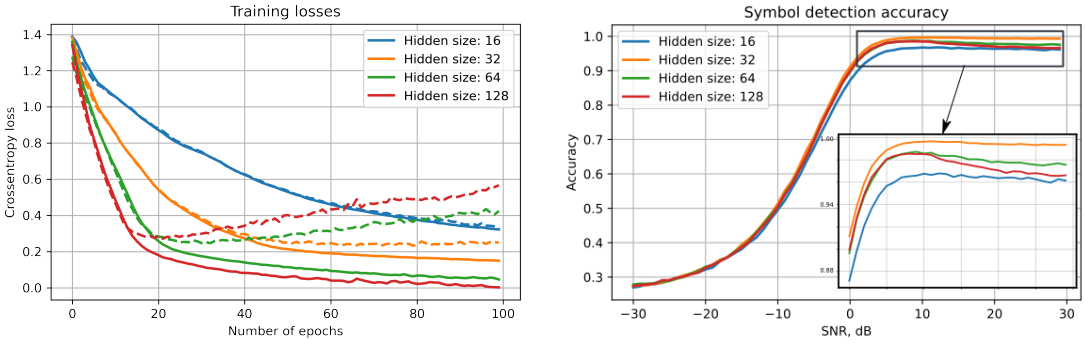
Table 4.2: Training parameters

Parameter	**Value**
Optimizer	Adam
Loss function	Cross-Entropy
Batch size	32
Learning rate	1e-3
Number of epochs	100
Weight decay	1e-4
Hidden sizes	16, 32, 64, 128
Number of layers	1, 2, 3

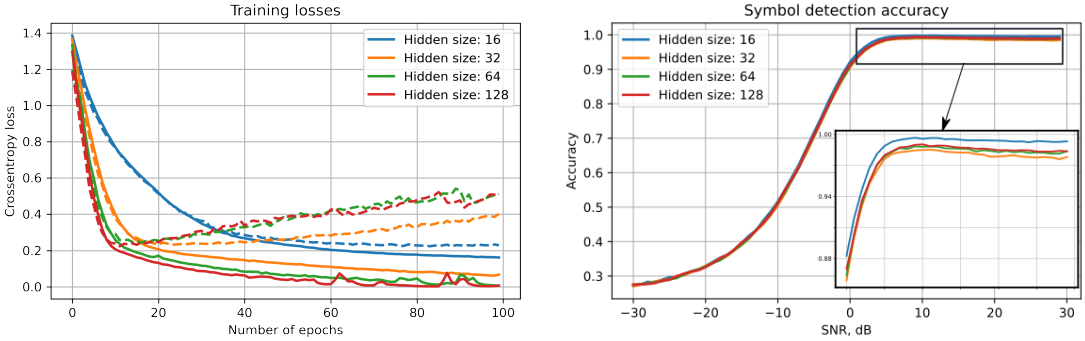
Figure 4.8 shows the averaged training (left-hand side) and testing (right-hand side) results for each model parameter configuration. Subplots (a), (b), (c) correspond to models trained with 1, 2 and 3 layers respectively.

As indicated by the training losses in Figure 4.8, there is a clear preference for large cell sizes when it comes to convergence speed, regardless of the number of layers. Large cell sizes, like 64 and 128, quickly fit the data and start overfitting at 20 epochs for a single layer configuration, or as early as 10 epochs when training deeper 3-layer RNNs.

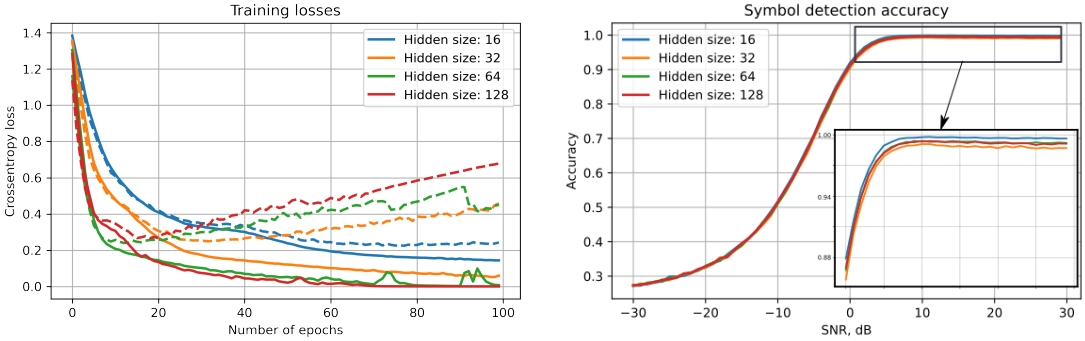
Looking at the accuracy results, on the right-hand side of Figure 4.8, the performance is surprisingly comparable, even at a cell size of 16 with just 1 layer achieving over 95% accuracy. At 2 and 3 layers deep, the smallest cell size also outperformed larger models – this makes sense for a simple problem like demodulating a very short QPSK sequence, as the smaller models are not as susceptible to overfitting because they have less capacity for memorizing the dataset. For more challenging problems and longer sequences, bigger cells will be preferable, as will be shown in the following sections.



(a) Training 1 layer



(b) Training 2 layers



(c) Training 3 layers

Figure 4.8: Training losses and accuracies of resulting models at varying number of layers and cell hidden sizes (dashed lines are validation losses)

4.3.5 Results

Multiple Seq2Seq models have been trained with varying numbers of layers and cell sizes. A summary of all model mean and best accuracies, can be observed in Table A.1 (in the appendix) and Figures A.1 to A.4. Weight decay regularization was also explored and summarized in Table A.1. It was found that too little (zero) and too much

(0.001) weight decay negatively impacts the accuracy of the achieved models, however a modest amount between 0.0001-0.0003 works well for improving generalization and achieving the best accuracies.

The best performing model for QPSK demodulation ended up being a 3 layer LSTM network with a cell size of 16. One of these models was evaluated over an SNR range from -30 to 30dB, and compared against a matched filter implementation (representing the ideal practical baseline) in Figure 4.9.

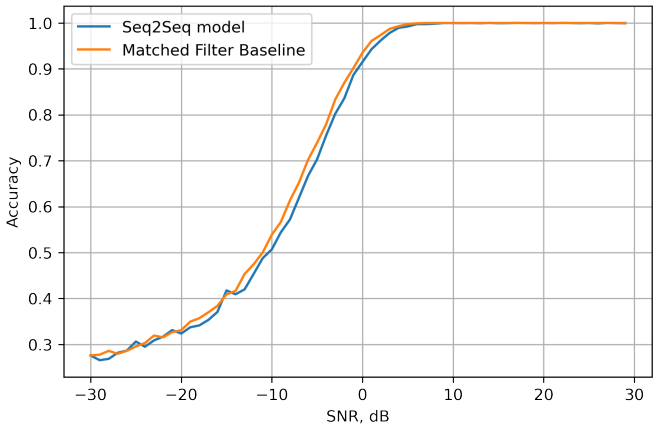


Figure 4.9: QPSK Demodulation Accuracy

While the results do not perfectly fit the baseline, it is clear from the accuracy curves of Figure 4.9 that the Seq2Seq model was capable of learning the matched filter and all of the functionality required to perform the symbol demapping, otherwise it would not have been able to so closely match the baseline result.

This section introduced how a Seq2Seq model can be trained to act as a baseband QPSK demodulator, and provided some intuition on the appropriate training set sizes and parameters. Building on the foundations of QPSK demodulation, in the next section the same architecture is used for more advanced signal processing tasks like AMC.

4.4 Simultaneous AMC and Demodulation

The beauty of an autoencoder structure, or deep networks in general, is that it can learn multiple communications tasks implicitly, without having to explicitly be told about individual necessary subtasks like matched filtering. Building on the previous task of QPSK demodulation, a new task for the Seq2Seq model to learn is added in this section – modulation classification.

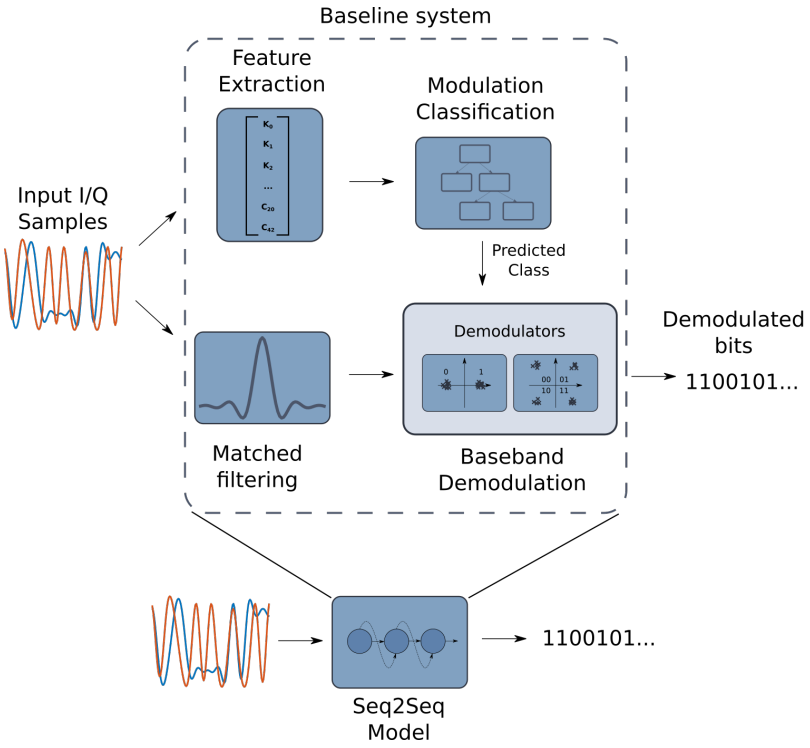


Figure 4.10: Overview of traditional flow vs Seq2Seq

A baseline system in this case, as demonstrated in Figure 4.10, is a Decision Tree (DT) trained on higher order statistics such as moments and cumulants of the modulated waveform. This would typically be a 2-step approach – the incoming I/Q sample stream would be independently consumed by the AMC pipeline and the demodulation processing chain, once the modulation scheme is determined the correct demodulator function is chosen and symbol demapping can be performed to recover the bits.

Using the proposed Seq2Seq DL approach, all of the above steps are learned by the Seq2Seq model implicitly; only one stream of I/Q samples is input to the model and it

directly outputs the detected symbols.

4.4.1 Dataset

The best performing architecture from Section 4.3 (3 layers and an LSTM cell size of 16) is used here for the combined classification and demodulation training. One difference to the architecture will be the decoder output dimensionality – the new labels will have 6 possible outputs to accommodate the additional 2 BPSK symbol possibilities. The new mapping is displayed in the following:

$$\begin{bmatrix} S_0(BPSK) \\ S_1(BPSK) \\ S_2(QPSK) \\ S_3(QPSK) \\ S_4(QPSK) \\ S_5(QPSK) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.2}$$

The dataset will also need to be modified with additional BPSK modulated waveforms. The approach is to incorporate and train on the same number of BPSK and QPSK examples – this is to ensure that both modulation classes are equally represented.

Determining the Best Training Set Size

Generally speaking, more data equals better performance. However, eventually increasing the amount of data fed to a fixed DNN can reach diminishing returns. Even though, by using simulations, an infinite amount of training data can be produced, it is desirable to find an optimum quantity that would allow the trained model to perform well enough without spending too many computational resources on generating datasets, and then spending even more resources training for an unnecessarily long period.

To determine a guideline of effective dataset sizes for training the Seq2Seq model for combined AMC and demodulation, the model is trained on a range of training sizes. Every dataset is generated with a range of SNRs. In the following experiments these include generating 128, 256, 512, 1024, 2048, 4096, 8192 and 16384 per SNR per modulation.

Table 4.3: Training and dataset parameters of AMC + demodulation model

Parameter	**Value**
Optimizer	Adam
Loss function	Cross-Entropy
Batch size	32
Learning rate	3e-4
Number of epochs	100
Weight decay	0.0001
Hidden size	16
Number of layers	3
Training SNR (dB)	-5, 0, 5, 10, 15, 20
Dataset sizes	1.5k, 3.1k, 6.2k, 12k, 24k, 49k, 98k, 196k

For example, 128 sequences $\times$ 2 modulation schemes $\times$ 6 SNR levels, results in $\approx 1.5K$ training examples in the dataset, as displayed in Table 4.3. Once a network is trained, it is evaluated over an SNR range of -30 to 30dB and the mean accuracy calculated. The mean accuracies of models trained on every dataset size are summarized in Figure 4.11. Calculating the mean accuracies provides an approximate trend line, however in future work to get a more accurate value, using percentiles would be more appropriate.

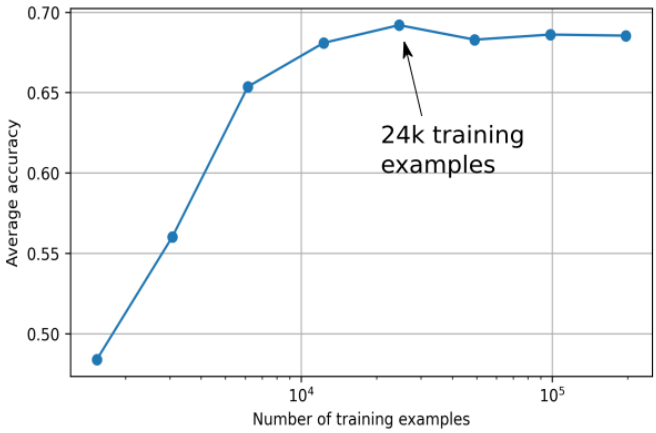


Figure 4.11: Mean test accuracy based on varying training dataset size

The results in Figure 4.11 show a clear correlation of model performance with increasing training dataset size. However, once the training data includes more than 24k examples, there are no significant average accuracy improvements. For contrast,

the authors of [95] used 1.3k examples to train an LSTM for ASK demodulation without AMC.

It is certainly possible to squeeze out more performance by amending the architecture of the model or tweaking hyperparameters, however these results are sufficient to gain the necessary intuition of the magnitudes of data required for training competent Seq2Seq models for short sequences in this domain.

4.4.2 Training

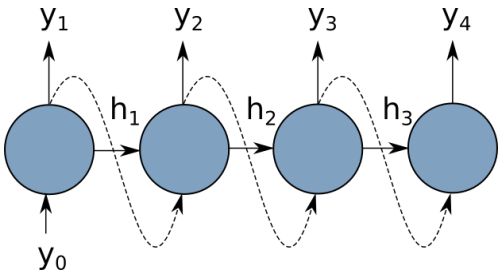
Based on the results in Section 4.3.5, we know that a higher number of layers is preferable. In this section the 24k example dataset will be used to train a Seq2Seq model for simultaneous AMC and demodulation with small (16) and large (128) cell sizes. Teacher forcing is explored as an option to speed up model training convergence and dropout is explored for regularization.

Teacher Forcing

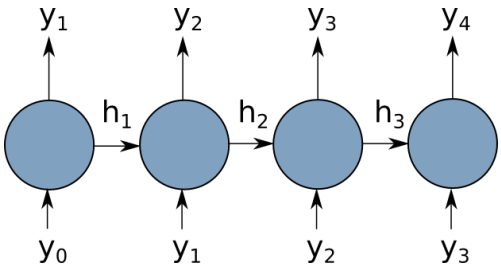
Teacher forcing is a commonly applied technique in RNNs, especially in Seq2Seq models where the optimization landscape is vast. Introduced in [101], it works by feeding the running RNN desired outputs from the previous states (i.e. labels), rather than letting it generate the next values itself. Using teacher forcing in the decoder structure is illustrated in Figure 4.12.

The advantage of using the teacher forcing method is a reduction of computation required during training, as it simplifies the learning task of the model. However, the end goal is still for the network to self-generate the next step inputs in a closed-loop operation. Teacher forcing is not a guaranteed method of improving training speed or performance however. As mentioned in [70], once deployed in closed loop mode, the network can run into issues as the generated inputs may not match exactly what was presented during training.

In practice, teacher forcing is implemented by closing the RNN loop at random during training based on a teacher forcing ratio [102]. For example, if the ratio is set to 0.25, then there is a 25% chance that the next RNN input will be fed from the ground



(a) Closed loop: the network is trained in the same configuration as it will be deployed, with the downside being that, during the beginning of training, randomly initialized weights will not produce very good decoder inputs for training, causing instabilities. This can make it more difficult to escape a local minimum in the optimization plane and halt training.

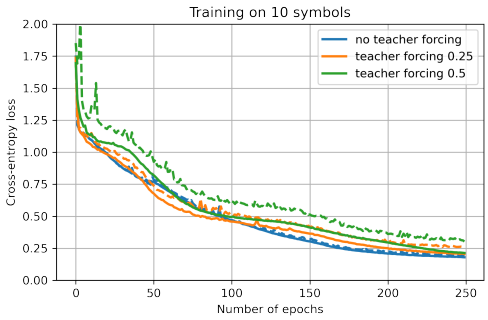


(b) Teacher forcing: instead of letting the network generate the next inputs, desired values are provided directly from the label data, which oftentimes will speed up the training process of RNNs. The downside is potential instability during testing as the the generated inputs in closed loop mode now might not exactly match what was seen in training.

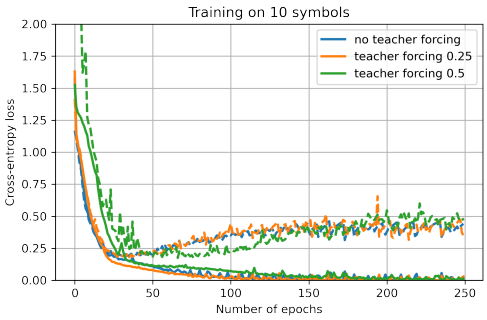
Figure 4.12: Decoder training modes

truth (labels), instead of the previous output of the RNN. A teacher forcing ratio of 0 means the network will always run in closed-loop mode, whereas a ratio of 1 will never loop the RNN predicted outputs into its inputs.

Figure 4.13 shows training loss comparisons for a small cell size of 16 and a larger cell of size 128, both being there-layer models, for an input sequence of 10 symbols. Two teacher forcing ratios were used for these experiments: 0.25 and 0.5. The full overview of the results can be found in Appendix A.1.2, which includes training losses for more symbol lengths.



(a) Hidden size 16



(b) Hidden size 128

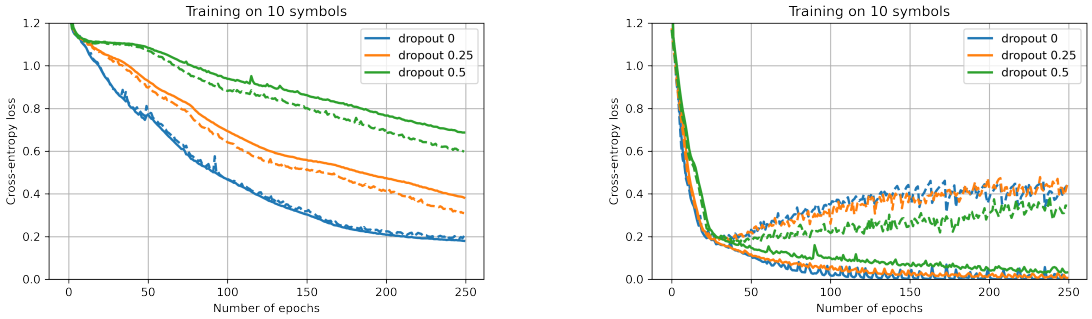
Figure 4.13: Training losses with and without teacher forcing (dashed lines are respective validation losses)

Observing the results in Figure 4.13 it is clear that teacher forcing does not help the model converge faster. In fact, a high forcing rate of 0.5, results in an overall higher validation loss for the small cell size, shown in Figure 4.13(a).

Unfortunately, teacher forcing does not seem to speed up training for either smaller or larger networks. Running both training and inference in closed loop mode for these models is a better approach. Of course, with more exploration, and some additional input processing and hyperparameter tuning, it could be possible to speed up training with or without teacher forcing.

Dropout

The effects of weight decay have been explored for QPSK demodulation and a summary is a available in Appendix A.1.1, showing that it generally helps Seq2Seq models generalize better. Another popular method used to regularize LSTM-based models is dropout. In the following experiment the same architectures (with a small and large hidden size), are explored with dropout rates of 0.25 and 0.5. The training results are summarized in Figure 4.14.



(a) Hidden size 16

(b) Hidden size 128

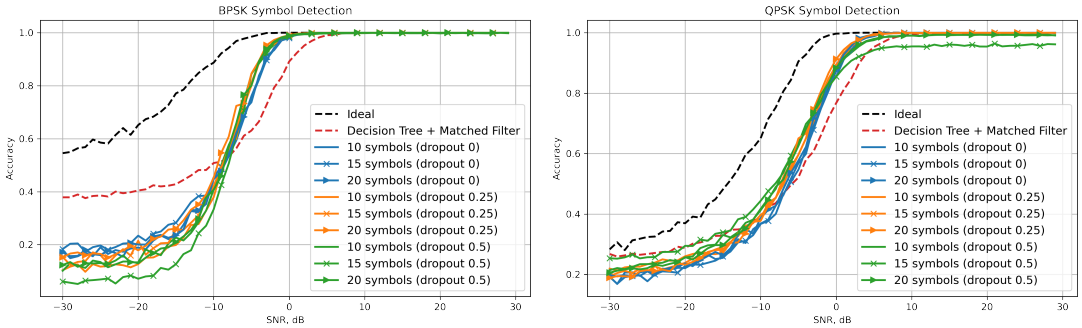
Figure 4.14: Training losses with and without dropout (dashed lines are respective validation losses)

For the model with smaller cell sizes, Figure 4.14(a), which already struggles to converge to a 10 symbol input sequence, dropout appears to exacerbate the issue, making it even harder for the model to fit the training set. Results with the larger-sized model are more promising, since it was already overfitting the training set and

could benefit from additional regularization. The dropout rate of 0.25 for the large model, Figure 4.14 (b), had little effect on training, however a dropout rate of 0.5 visibly lowered the validation loss.

4.4.3 AMC and Demodulation Results

The Seq2Seq model with a hidden size of 128, was trained and evaluated on input sequences of 10, 15 and 20 symbols (with 4 samples per symbol, corresponding to 40, 60 and 80 complex samples). Performance is compared with a Decision Tree (DT) baseline (in combination with standard baseband demodulators) as well as the ideal demodulation accuracy (using a matched filter with perfect knowledge of the incoming modulation type). The results are summarized in Figure 4.15, noting that the dashed black lines represented the ideal case where perfect information of the modulation scheme is known and only demodulation performed, and the dashed red lines are the DT baseline.



(a) BPSK symbol prediction accuracies with hidden size 128

(b) QPSK symbol prediction accuracies with hidden size 128

Figure 4.15: Combined classification and demodulation model performance

The DT model is trained using scikit-learn [65], using the same dataset as the Seq2Seq model. The selected features include 5 higher order moments (μ_2, μ_3, μ_4, μ_5, μ_6) calculated from the instantaneous amplitude, phase and frequency (15 total moments) and the first 5 complex cumulants described in [103] – totalling 20 input features. The DT prediction then directs which demodulator should be used for the received symbols – BPSK or QPSK.

The Seq2Seq results are plotted with and without dropout, which has had minimal impact on the overall model accuracy. The trained Seq2Seq model generally outperforms the DT baseline from -10dB onwards for BPSK and from around -5dB for QPSK. Interestingly the DT model wins at very low SNR values, even though neither model has been trained in those ranges. One reason for the discrepancy is due to the statistical features selected for the DT being more tolerant to low SNR, whereas the end-to-end trained Seq2Seq model needs more exposure to such channel conditions.

Both DT and Seq2Seq models could be tuned to further improve these inference results. This experiment has demonstrated, however, that a Seq2Seq model can be treated as a single unit for AMC, matched filtering and baseband modulated symbol detection. Not only that, but without much tuning it is competetive with classical ML models like the DT working jointly with classic DSP algorithms such as the matched RRC filter.

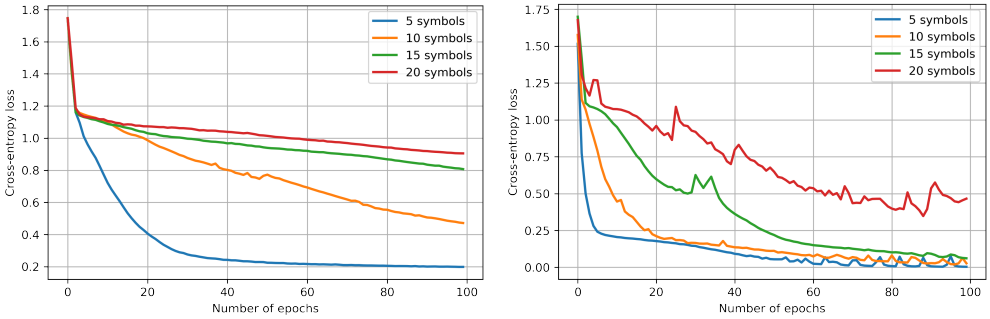
4.4.4 Scaling Discussion

One of the key advantages of RNNs is that they can be trained and operate on input sequences of arbitrary length. RNN research and the inception of cells like LSTMs [71] has made training on long sequences easier, however it still remains a difficult problem when scaling to very long inputs [104]. In signal processing it is common to deal with 100's or 1000's of samples per input frame, which models must be able to cope with for training and inference.

Looking at just the training losses (excluding validation for now), the explored Seq2Seq models have been trained on 5,10,15, and 20 symbols (represented by 20,40,60,80 I/Q samples respectively), as shown in Figure 4.16.

The smaller model in Figure 4.16 (a) can easily overfit 5 symbol sequences, however as the length increases the model increasingly struggles to fit the dataset. Increasing the cell size, and thus the complexity of the model, it can converge much faster, as shown in Figure 4.16 (b), however at 20 symbol inputs we can observe difficulties of convergence once more.

One of the reasons why it is difficult for RNNs to learn longer sequences is that,



(a) 3 layers, 16 cell size

(b) 3 layers, 128 cell size

Figure 4.16: Training loss at different input sequence lengths

as the number of input samples increases so do the number of steps required in each backpropagation pass. Recall that an RNN has to step through and compute the hidden state for each input step – that is a lot of multiplications, prone to vanishing gradients. Additionally, long input sequences will add a general computational complexity penalty, demanding more hardware resources to properly train the model.

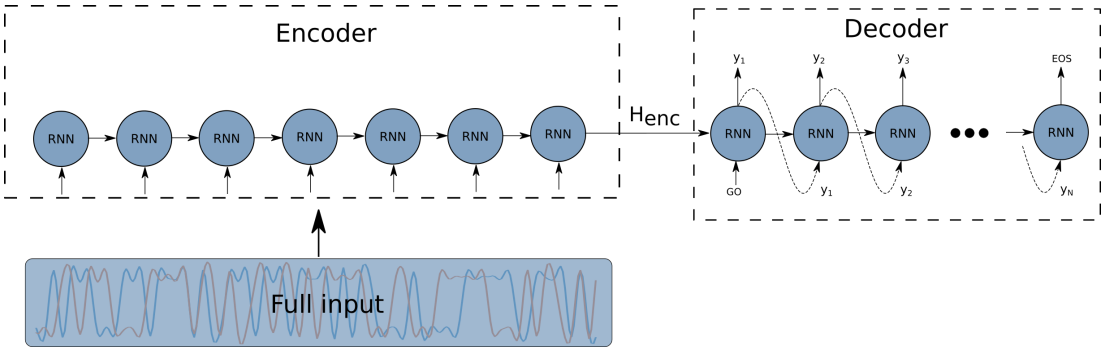
4.5 Reducing Complexity Burden with CNNs

While the Seq2Seq model has a high capacity and can learn interesting problems, it is also very complex and difficult to train. In this case the bulk of the training overhead comes from the encoder side, because it needs to compute a new hidden state for every input sample, resulting in a backpropagation signal that has to travel through potentially hundreds of nodes once unrolled. It is manageable for tiny inputs, however this does not scale well computationally to longer input sequences.

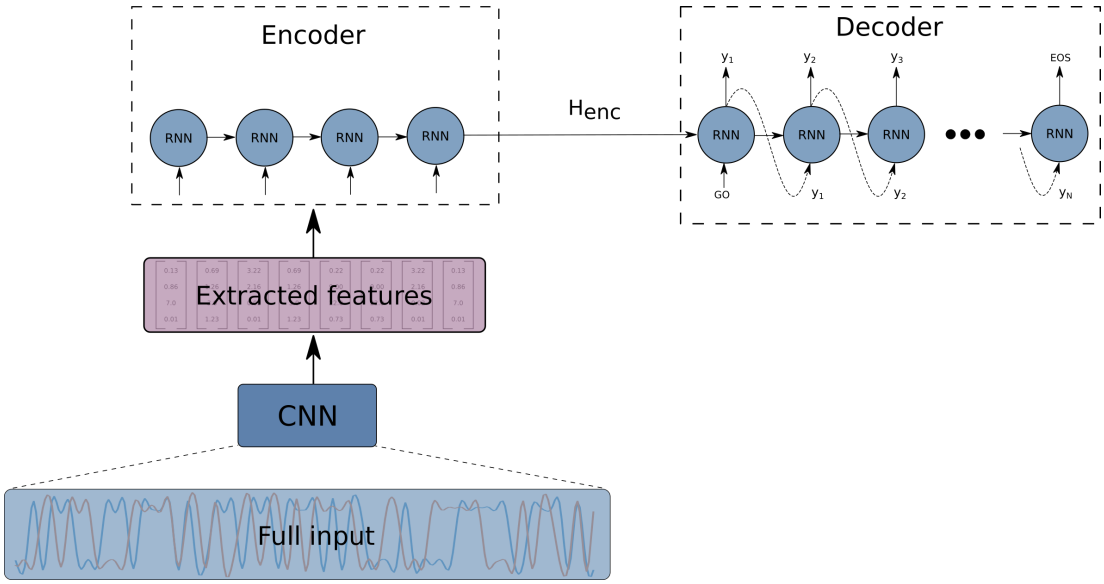
In application scenarios where the receiver is implemented as an embedded system with strict power constraints it is desirable to minimize the computational load as much as possible. For example, if the receiver is part of a battery-powered node in sensor network or an IoT application. Additionally, training DNNs can be a costly endeavour and focusing on more efficient model architectures is important to reduce data center loads as well.

One way to solve, or at least alleviate, the RNN scalability problem is to alter

the encoder architecture by adding a CNN for feature extraction, as demonstrated in Figure 4.17.



(a) RNN encoder: entire input is processed sample by sample by the RNN. Disadvantage here is that on very long sequences this can lead to vanishing gradients and make it very difficult to train the model.



(b) Conv+RNN encoder: the entire input is first pre-processed by a CNN, reducing the number of steps the RNN encoder has take in order to fully encode the input.

Figure 4.17: Using a CNN to simplify the encoding task.

The CNN can process the entire input first, and output a vector of features proportional to the input size. The RNN encoder then processes the features instead of raw I/Q samples – this eases the task of the RNN learning the features itself and drastically reduces the number of steps it needs to perform to encode the input. A lower number of steps simplifies backpropagation because the unrolled graph becomes shorter and

less susceptible to vanishing gradients.

4.5.1 Convolutional Encoder

The additional CNN is composed entirely of convolutional and pooling layers, which downsample the full input and convert it into a much smaller representation, alleviating some of the computational burden from the encoder. The encoder CNN parameters are summarized in Table 4.4, with a reference input sequence length of 100 samples – the dimensionality will scale appropriately for different sequence lengths.

Table 4.4: CNN parameters (ref input length 100 samples)

Layer	**Parameters**	**Out Shape**
Input	-	2×100
Conv1D + ReLU	9×1, 16 filters, padding=4	16×100
Maxpool1D	2	16×50
Conv1D + ReLU	9×1, 8 filters, padding=0	8×42

The simple pre-processing CNN consists of 2 convolutional layers and a single max-pooling layer and is used to feed the RNN encoder processed feature vectors of the input sequence. The first layer consists of 16 1-dimensional filters, each with a width of 9 samples. The inputs to this filter are padded to maintain the dimensionality, and prevent the first and last symbol information from getting lost due to transient responses of the filters. The maxpool layer downsamples the input tensor by a factor of 2, reducing a 100 sample input to an intermediary representation that is 50 samples long. The second convolutional layer has 8 filters with the same width of 9 samples, without padding, further decreasing the length of the input signal to a sequence of 42 steps of feature vectors containing 8 values – this is less than half of the steps the RNN encoder would have had to process otherwise.

Convolutional layers are excellent feature extractors and, in this case, input data compressors. Now the encoder RNN can step through feature vectors that are output by a convolutional layer, rather than raw I/Q samples, adding an additional abstraction layer and reducing the computation required by the RNN. From the perspective of the decoder, nothing changes because the dimensionality of the hidden state that is being output by the encoder RNN remains the same.

4.5.2 Training Results

The new Conv Seq2Seq model is trained using the same parameters as the models in Section 4.4. The only difference is that the first LSTM layer is replaced with the CNN described in Table 4.4, meaning that the new encoder consists of 2 convolutional layers and 2 LSTM layers. The training losses produced by the original RNN encoder and the new Conv+RNN encoder configuration are shown in Figure 4.18.

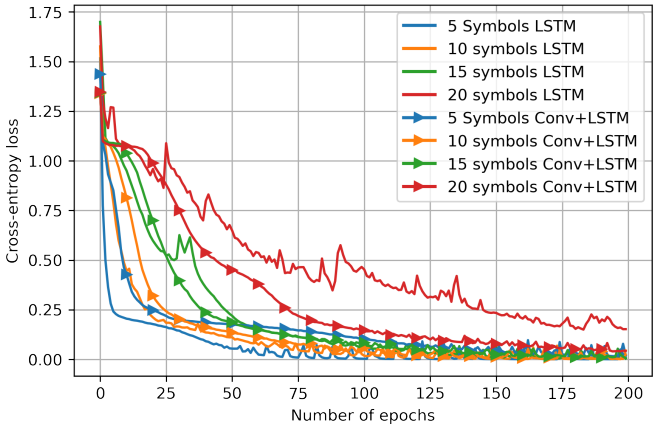
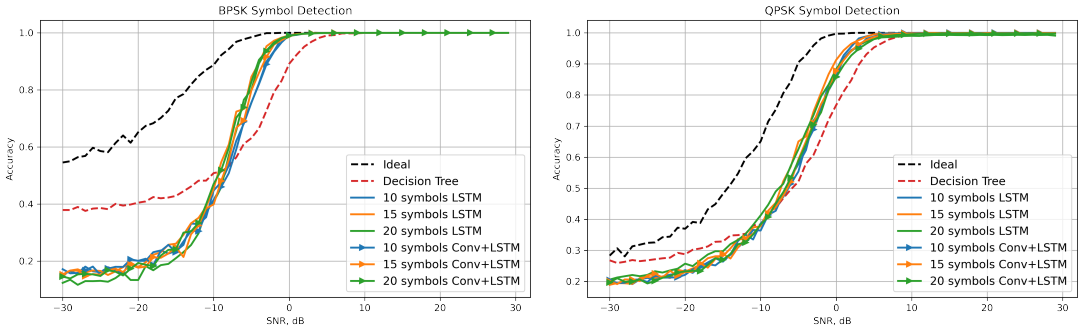


Figure 4.18: Training losses of the new Conv+RNN encoder

The differences are very minor for shorter sequences – the RNN does not struggle to backpropagate through the 5 pulse-shaped symbols, and in fact the RNN only configuration converges faster than the Conv+RNN variant. However, beyond the sequence length of 10 symbols the Conv+RNN configuration always converges faster than the RNN-only encoder.

The trained models are evaluated over an SNR range from -30 to 30dB, and compared with previously trained purely RNN-based Seq2Seq models at each input sequence length. The results are summarized in Figure 4.19.

In summary, while the convolutional encoder improves training speed, it does not have a significant impact on accuracy when compared to previously trained models.



(a) BPSK symbol prediction accuracies (b) QPSK symbol prediction accuracies

Figure 4.19: Accuracy comparison of purely LSTM and LSTM+Conv encoders

4.5.3 Runtime Complexity

The runtime complexity of the different encoder methods can be estimated by measuring the time it takes to process a single input. In this section the computational complexity of three encoders is compared: a 2-layer and 3-layer purely RNN-based encoders and the new Conv+RNN encoder proposed in this work. Each method was evaluated on a desktop AMD Ryzen 5 CPU, and individual runtimes averaged over 100 iterations illustrated in Figure 4.20.

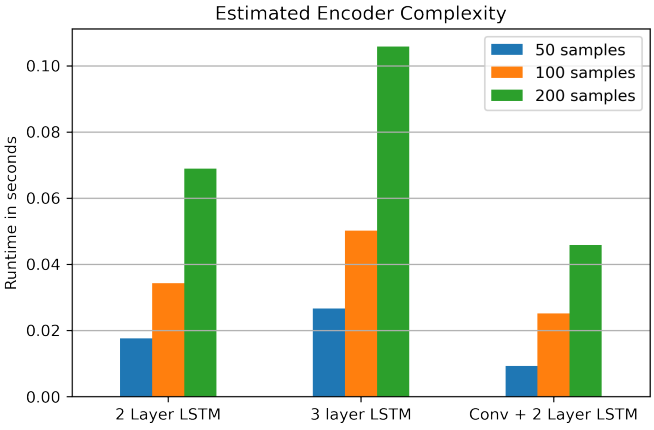


Figure 4.20: Estimated complexity of the Seq2Seq encoder implementations

The estimated Conv+RNN encoder complexity is lower than both RNN-only methods. This might seem surprising, given that the convolutional encoder consits of 2 LSTM cells as well. The reason it still beats it in inference time is due to the RNN

having to step through roughly 2 times fewer input steps (reducing the RNN computation required by a factor two).

4.6 Chapter Conclusion

A Seq2Seq autoencoder approach was demonstrated for baseband digital signal demodulation. This model was also shown to be capable of learning AMC and demodulation of multiple modulation schemes (BPSK and QPSK) simultaneously. The network was able to learn multiple subtasks implicitly in the process of learning to classify incoming modulated symbols, such as the features required to classify modulation types, matched filtering and downsampling. An important property that an RNN-based Seq2Seq model possesses is the ability to process inputs at variable sequence lengths, which is desirable for intelligent radio receivers. As was shown, it can be used as a building block and add significant malleability to a deep learning model.

The research presented in this chapter contrasts with previous works on DNN-based demodulation by allowing a single trained DNN module to operate on multiple symbols in one inference execution. This enables training of a single architecture on arbitrarily long waveforms, albeit with computational limitations. Other LSTM-based models demonstrated in [24], [95] also offer this capability, however since their implementations use a single RNN, they are only capable of a one-to-one input-output mapping. A Seq2Seq model is composed of 2 RNNs, allowing many-to-few or few-to-many configurations. This flexibility makes the Seq2Seq model compatible with a wider range of problems, and enables nice features like downsampling (e.g. a symbol that is represented by 4 samples will result in the decoder producing a single output for that 1 symbol, as opposed to 4 if a single RNN was used).

Some of the downsides to the Seq2Seq autoencoder, however, are a difficulty in training and implementation. Transfering these models to hardware is a non-trivial problem as explored by previously published works due to the inherently recurrent nature of RNNs, making them difficult to parallelize [105], [106]. Even though LSTMs were introduced as a way to mitigate vanishing gradients and allow models to be trained

on longer sequences, it is still a non-trivial task. For digitally modulated baseband signals, one method identified in this chapter to alleviate the problem was the addition of convolutional layers to the encoder, which decreased the number of iterations required by the RNN, easing the training of longer sequences, and reducing complexity. Convolutional layers are very good at feature extraction, providing a more dense output, but with fewer time steps for the RNN to iterate over. The fortunate thing about convolutional layers is that, like RNNs, they can work on arbitrarily long sequences of data as well, maintaining the original input size flexibility of the Seq2Seq model.

There are a number of techniques not explored in this work that could further improve the results and potentially help with training convergence, such as attention, which has been shown allow the RNN to focus on parts of the input while making the output prediction, which improves training convergence [107]. Bidirectional RNNs [108], where the input sequence is processed simultanously in the positive and negative time dimension, resulting in richer context vectors. Additionally more advanced data configurations using padding and End-Of-Sequence (EOS) tokens were not explored to showcase the full malleability of the Seq2Seq architecture. A large scope for optimization is available for future studies of the Seq2Seq architecture.

In this chapter convolutional layers were introduced as a method to reduce the complexity of an RNN-based autoencoder. The next chapter focuses on fully convolutional architectures to improve the flexibility and lower complexity of existing DNN solutions to problems like frame synchronization.

Chapter 5

Fully Convolutional Neural Networks for Frame Synchronization

This chapter introduces a Fully Convolutional Network (FCNs) architecture for the problem of phyiscal layer Frame Synchronization (FS). By increasing the probability of successfully detecting a transmitted packet, the overall throughput of the system can be enhanced. Additionally, by reducing the required preamble length for successful packet recovery, due to improved receiver performance, the transmitter can save power by having to emit fewer redundant bits - this can have significant implications in wireless sensor networks, where conserving power can be really important.

A key advantage of adopting the proposed FCN architecture is that it can be deployed on arbitrary input lengths, whereas more common CNN architectures used for FS will require the input dimensionality during inference to match exactly the dimensionality of the data used during training.

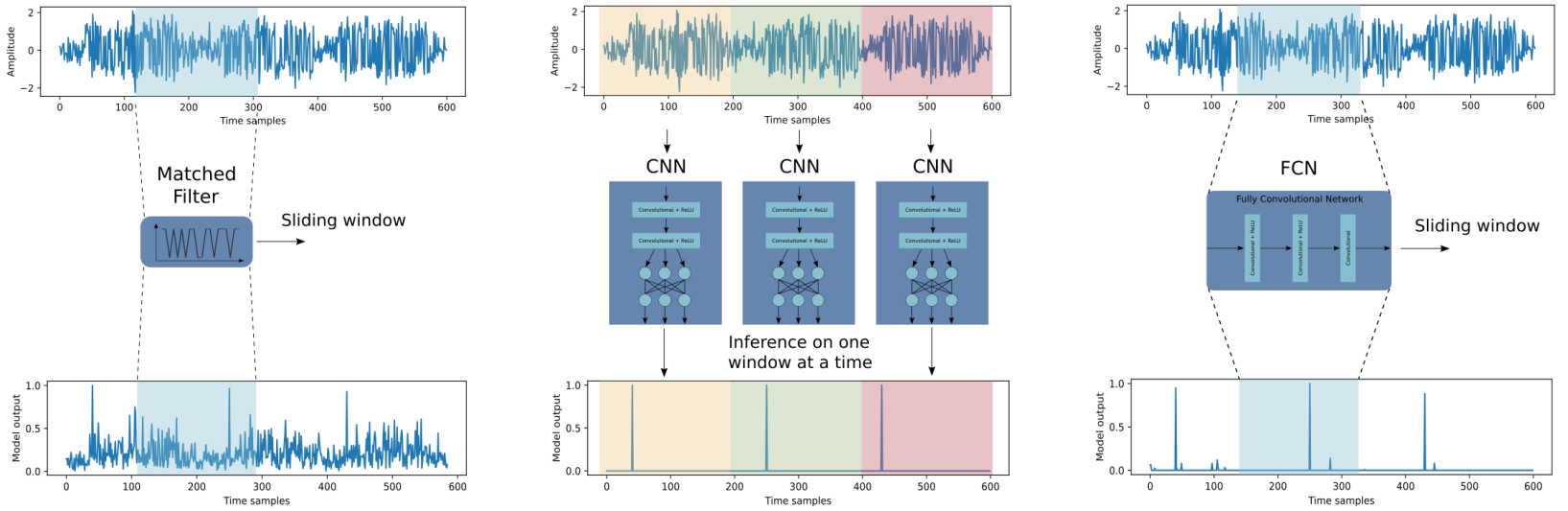
The trained FCN models are evaluated on a variety of synchronizing sequences of different lengths and channel conditions. Best practices for training these models and a deeper introspection of how they work will be discussed, and complexity tradeoffs evaluated and compared with more typical CNN approaches.

5.1 Motivation

Modulation and coding schemes are nearing the theoretical limit of what is achievable in terms of Bit Error Rate (BER). However in modern communication systems there still exist significant overheads due to the additional signalling required for various synchronization stages in order to successfuly retrieve over-the-air transmitted data. This overhead, caused by sending additional redundant bits and preambles, introduces inefficiencies in overall throughput, power and RF spectrum usage. Reducing or eliminating these overheads by implementing smarter transceivers will have a big impact on power and utilization of available radio spectrum [109].

DL is a promising technology in addressing the issue of commmunications overhead and has been shown to work well for problems like FS. Existing DL solutions treat FS as a classification problem, and apply CNNs to solve it – review of existing CNN solutions is presented in Section 5.2. While standard CNNs (composed of convolutional and fully connected layers) are shown to produce very good results across many applications, for some communications problems it can be appropriate to investigate architectures more suitable to the field – such as FCNs. FCNs can be trained and configured to work much like digital filters, and could slot into existing systems with minimal pain caused to the system architect.

The inference differences of a standard matched filter, CNN and FCN are captured in Figure 5.1. Figure 5.1(a) shows standard correlation, where a sliding matched filter produces the output peaks at the packet frame locations. Figure 5.1(b) demonstrates the typical DL approach using a CNN – classification CNNs will usually have fully connected layers, which means that the input has to be separated into windows of a pre-determined length. Finally, in Figure 5.1(c) the proposed FCN model is illustrated – because there are no fully connected layers it can be deployed as a standard filter, much like 5.1(a), however it also benefits from data-driven DL techniques for enhanced performance – combining the best of both worlds.



(a) Correlation: the standard matched filter approach, where the correlation peaks are correctly indicating the packet start indices. The resulting output is fairly noisy compared to DL methods.

(b) CNN: a typical CNN configuration is shown for FS as a classification task. The output will generally be very clean and performance good, however not as flexible as correlation.

(c) FCN: demonstrates the proposed FCN approach, combining the sliding window of correlation and data-driven driven methods of DL to create an accurate and flexible solution.

Figure 5.1: Comparison of classical and DL-based frame synchronization methods.

One of the main advantages of using an FCN model for wireless communications, such as FS, is that once this model is trained it can effectively be treated as any other filter encountered in DSP. While a CNN needs to be trained and deployed on exactly the same input sequence length, FCNs can be trained on small or large inputs, and then deployed as a sliding window on inputs of any length.

5.2 Frame Synchronization

FS is an important step performed by the receiver when recovering transmitted data packets. Conventionally it is estimated by taking the cross-correlation of a known preamble sequence and the captured waveform. In a bursty (meaning data can arrive in non-periodic intervals of time) communications scenario the packet, containing a known preamble and the data samples, will be received together with "dead air" samples containing no information, as shown in Figure 5.2.

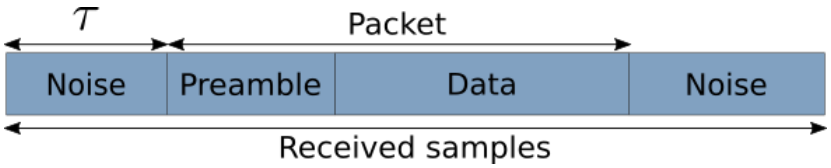


Figure 5.2: Packet data sizes in bursty communication

Finding the time offset index, τ, of the received waveform $r = [r_1, r_2, ..., r_M] \in \mathbb{C}^M$ is the main goal of the frame synchronizer in a wireless receiver. The most common way of performing this detection is to use a noncoherent correlation detector (1) and estimate the peak $\hat{\tau}$ from the absolute cross correlation response,

$$\hat{\tau} = argmax \left(\left| \sum_{i=0}^{M-1} r_i p_i^* \right| \right), \tag{5.1}$$

where $p = [p_1, p_2, ..., p_{N_p}] \in \mathbb{C}^{N_p}$ is the preamble symbol sequence of length N_p, M is the number of samples in the received signal and $*$ the complex conjugate. Of course correlation methods can be improved further by adding correction terms to (5.1), for example as shown in [110].

Common preamble sequences used for this task are Barker codes, PN (Pseudorandom Noise) and Zadoff-Chu sequences as seen in LTE and the New Radio 5G standard [111] [112]. Ideally a preamble sequence will have good autocorrelation properties, where the cross-correlation with itself is very strong, but not when correlating with random data, which the payload is assumed to consist of. As an example, the Barker sequence, illustrated in Figure 5.3, exhibits such properties.

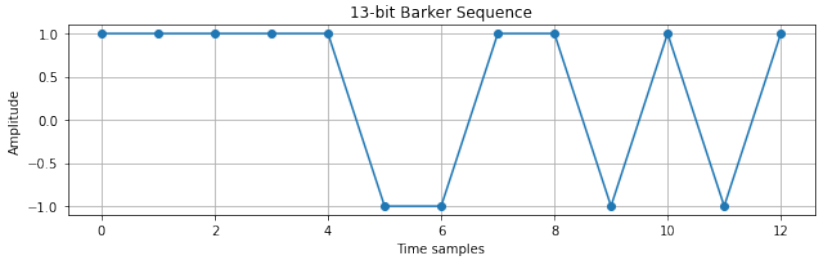


Figure 5.3: Barker sequence

When correlated with itself the barker sequence produces an output shown in Figure 5.4. It has a strong response once the sequence overlaps exactly, and at every other sample interval it produces small outputs between 0 and 1.

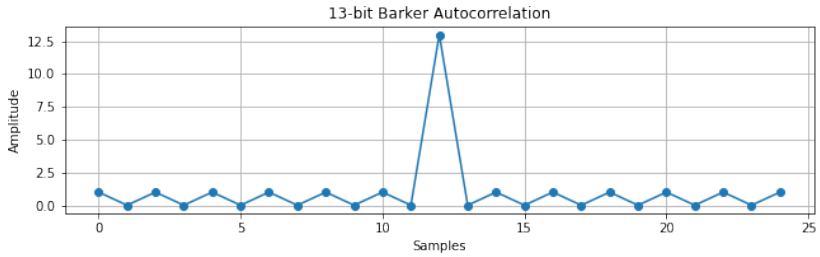


Figure 5.4: Barker autocorrelation

An example waveform containing a packet with a 64-bit payload of random bits and a 13-bit barker sequence preamble is shown in Figure 5.5.

Running the waveform from Figure 5.5 through the correlator receiver, the output generates a clear peak, indicating the start of the data packet, as can be observed in Figure 5.6. Because the sequence is short relative to the payload size, the output does not manifest in the ideal response shown in Figure 5.4. Instead the correlation outputs non-negligible values due to partial matches with the preamble. If the Barker

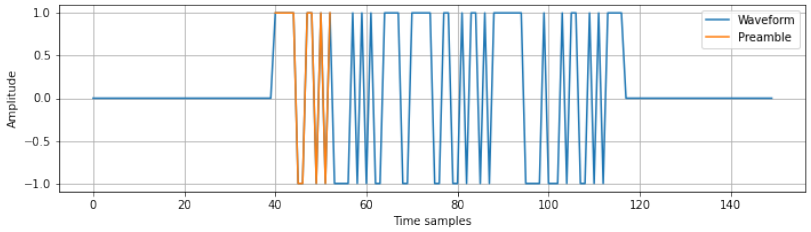


Figure 5.5: Transmitted waveform containing preamble and payload

sequence (or something close to it) appeared within the payload symbols, it could result in a false positive detection and the entire packet would be lost, which is why frame synchronization is such an important step in the receiver chain.

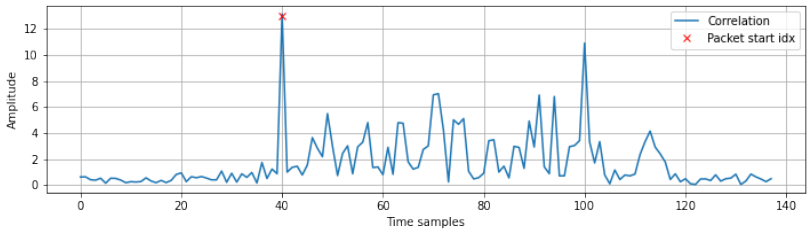


Figure 5.6: Correlation Receiver Output

The preamble detection peak as observed in Figure 5.6 is not always as prominent and detection can be susceptible to noise, carrier offsets and other channel effects. The negative effects can be mitigated by increasing the length of the preamble, however the tradeoff is reduced efficiency of the overall communications system (more power, less throughput, etc.). Another way of counteracting these effects is by introducing more complex receivers, as will be demonstrated with DL-based receivers in the following sections.

5.3 Related Work

Early research on improving FS using ML was investigated for a Radio Frequency Identification (RFID) system in [113], where Multi-Instance Learning (MIL) was used in combination with a Support Vector Machine (SVM) to predict whether an input frame contained a preamble sequence. The MIL approach showed an improvement in

accuracy over correlation methods at small carrier frequency offsets on input sequences of 62 samples.

An RNN-based architecture for blind frame location estimation, where the preamble sequence was not known beforehand by the receiver, was developed for an eavesdropping system on BPSK modulated sequences [37]. Interestingly, in the cited work the RNN is not pretrained with a known preamble sequence; the model relies on the repetitiveness of the synchronizing sequence of the continuous transmission. The RNN-based synchronizer showed promising results in AWGN channels, however such deployment is not suited for bursty traffic.

Previous work using CNNs for synchronization was undertaken for carrier offset estimation as well as timing offset estimation in [32]. The authors proposed a CNN architecture with a single output regression, however it did not outperform the baseline correlation approach in the tested AWGN and fading channels. A CNN with a single regression output was later also developed for IEEE 802.11ah standard packet detection [33]. Their approach outperformed baseline correlation approaches, and showed improved performance in low SNR scenarios. The authors stated that the full flexibility of the CNN architecture was not explored in order to simplify the complexity evaluations, which is why the CNN used fixed-length filters and inputs.

FS has also been researched as a classification problem. A minimal CNN model was built with a softmax classifier to predict the location of the preamble sequence, as part of an autoencoder design [34]. Previous work on MLP and CNN architectures, both configured as classifiers, was conducted with an AWGN channel in [35]. Their results showed that the CNN easily outperformed a simple MLP, however the study was limited as none of the results were compared with a more traditional method, like correlation as a baseline. Another CNN implementation was used to enhance a traditional correlator output by operating on standard correlation output samples [36]. While the proposed CNN-enhanced approach showed an average improvement of 2dB over the baseline on an impressive input size of 10k samples, the implementation requires reshaping the correlation output into a 100×100 matrix to be compatible with their CNN architecture, which is impractical to repurpose for different input lengths.

In review, the problem of frame synchronization has been approached from an ML perspective to enhance existing algorithms. It has also been researched as both a regression and classification problem for fixed-size input sequences – these works operate as illustrated in Figure 5.1 (b), where the assumption is that only 1 preamble sequence can exist in a received frame. To re-apply these implementations to a different deployment scenario, for example, requiring even a slight change in input shape would require reproducing a new dataset and training an entire new model.

While RNNs seem like an appropriate solution to the fixed-size problem, existing works are very specialized towards continuous transmissions and have not been shown to work in bursty communications [37]. Additionally, RNNs can be difficult to train, as was demonstrated in Chapter 4 of this thesis. The work presented in this chapter aims to address the fixed-size problem present in existing DNN solutions, while providing the improved detection performance that comes with data-driven approaches.

Preliminary results of the work presented in this chapter were published in [114], highlighting the flexibility of the FCN architecture and demonstrating improved performance over traditional methods. Additional contributions in the following sections include a more comprehensive model parameter search and introspection of the architecture. The lessons learned on training FCN models for FS can be generally applied to many DL applications in the wireless communications domain.

5.4 Training DL models

This section will review the FS data preparation, FCN architecture and parameter search, as well as the training setup, including discussion on hyperparameter selection.

5.4.1 Dataset

Packet detection by utilizing FCNs is treated as a regression problem, where the deep filter has to output peaks at locations where a preamble is located within a captured frame of samples. The training process will attempt to create a model that suppresses all of the noisy samples that are not the true preamble index, while maximizing the

desired output.

Just like Chapter 4, all of the data in the following sections has been generated from scratch with Python and NumPy. The labels are generated from random distributions in the range of possible packet offsets. Each training example consists of the I/Q samples of a simulated bursty transmission of a single BPSK modulated packet that includes a preamble and random data payload, as shown in Figure 5.7.

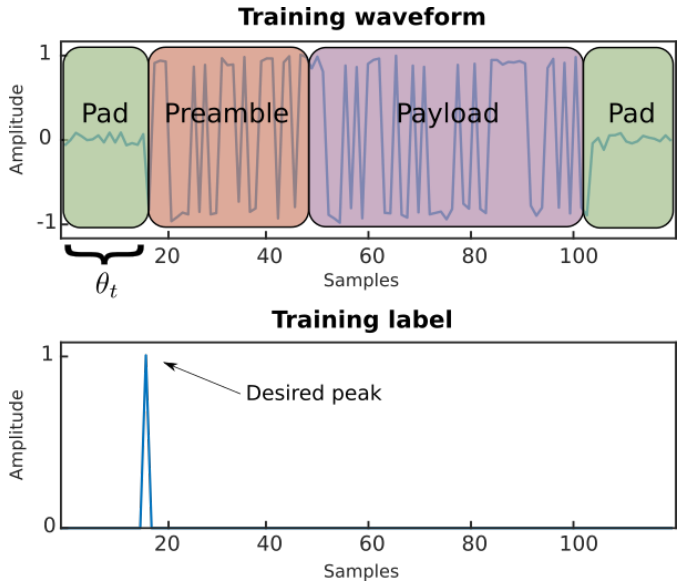


Figure 5.7: Training data example

Even though the FCN performs a regression during inference, the dataset is perfectly suited for tackling as a classification problem with softmax – as shown in Figure 5.7, the label is an ideal response with a single peak at the time delay index, corresponding to the actual start of the packet, and the value of 0 elsewhere. While the FCN does not treat it as a classification, during training a cross-entropy loss function can be appropriate. Loss function selection will be explored in more detail in Section 5.4.3.

Selecting the best initial training SNR

When generating training data there are many variables to consider. Does it have multipath channel effects, non-linearities, what is the SNR and, if not a single SNR, then what range of SNR values should be included? Ideally the training data will be

representative of the types of perturbations the received signal will have experienced. Practically, it is very likely the model may overfit to the training set if there is not enough data, or if the range of channel conditions is insufficiently represented. One way of approaching this (and producing a model that is at least as good as a baseline model) is to train it on relatively clean data (high SNR) and apply regularization, such as weight decay.

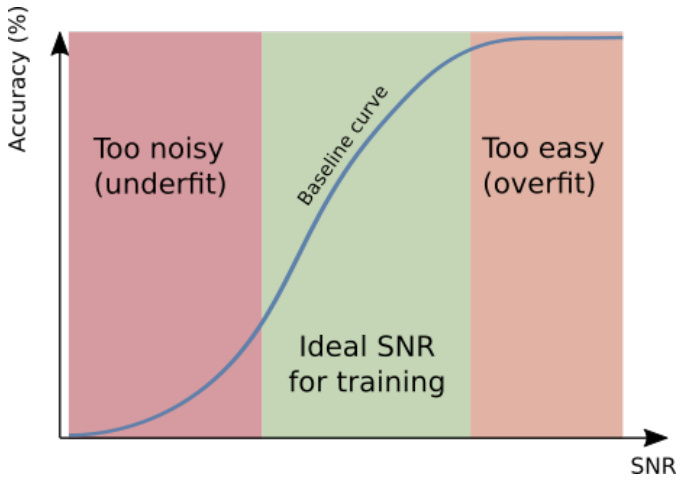


Figure 5.8: SNR selection based on baseline performance

A heuristically derived rule of thumb (from the work carried out in this thesis) for selecting the starting training SNR level is to inspect the baseline accuracy curve and choose the SNR where performance is beginning to saturate to the maximum value. In the context of frame synchronization, one would be looking at the accuracy of the noncoherent correlation of the synchronizing sequence – the SNR at which it reaches roughly 90% accuracy is generally a good starting point for generating training data.

To reinforce this claim, a 32-bit PN sequence detector was trained on a dataset generated for SNRs ranging from -30 to 30dB in steps of 5dB. For each SNR level, 5 models were trained with different random seeds, and each of those models was evaluated under an SNR range of -10dB to 10dB. The plot in Figure 5.9 shows the results of this experiment – each blue bubble is the mean accuracy achieved by the 5 models trained at that SNR. Similarly to Figure 5.8, it is clear that a 'sweet spot' SNR range exists where the trained model shows ideal performance. When there is too

much noise (up to -15dB SNR) the model fails to fit the training set and the test set accuracy drops to nearly 0%, and when the data is too clean (from 10dB upwards) the model most likely starts overfitting to the clean dataset, failing to generalize to unseen data.

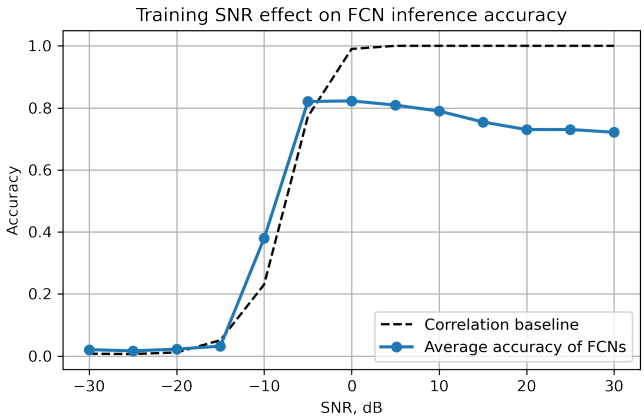


Figure 5.9: Mean accuracy of each model trained at a different SNR

An important takeaway from this result is that higher SNR datasets, at least initially, should be preferred. Too low an SNR can easily result in "garbage in garbage out" learning, where there is very little signal in the data from which the model can learn meaningful patterns. That said, some noise is still necessary to produce a robust solution.

5.4.2 Architecture

FCNs are most prominently used for dense pixel-wise classification in image segmentation tasks [115]. This type of network is constructed only from convolutional and pooling layers. Since they have no fully connected layers, they are not constrained to a fixed sized input, as is the case with the feedforward networks, that work by performing matrix multiplication of their learned weights by the entire input at once. This section covers an initial exploration of some key architecture parameters to uncover insights as to what type of FCN configurations perform favourably on FS.

Architecture Sweep

There are many architectural decisions that can be made when building a DNN – number of layers and filters, individual kernel dimensions, activation functions, additional layers like pooling, or regularization layers like batchnorm. The appropriate architecture might also change depending on the training dataset (some architectures could work better with certain impairments than others). The approach in this section is to approximate a "good enough" architecture for a single dataset by generating a number of different models with varying numbers of layers, filters and filter widths. Seeing how different models perform on the FS problem will give some insight on which parameters work best, and give a clearer guidance of how to structure DNNs for this task going forward.

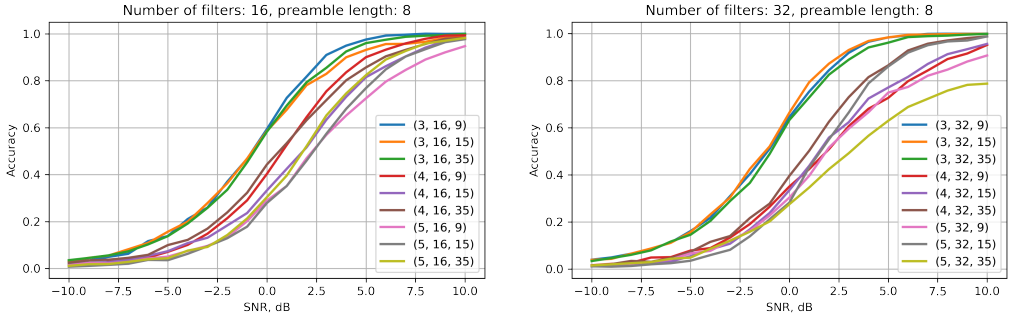
An initial FCN architecture sweep was performed by training and evaluating a number of models for 3 different PN sequence preamble lengths, for a set of configurations with varying numbers of layers, number of filters per layer, and widths of filters. For each configuration, 5 models were trained and evaluated at an SNR range from -10 to 10dB. Every model was trained on a dataset containing 1024 examples with a consistent SNR of 5dB, the datasets are discussed in more detail in the next subsection. Architecture sweep parameters are summarized in Table 5.1.

Table 5.1: Architecture sweep parameters

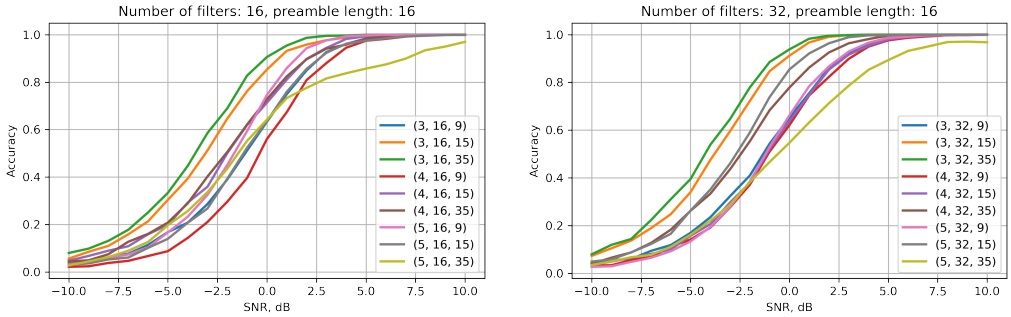
Parameter	**Values**
Preamble length	8, 16, 32
Number of layers	3, 4, 5
Number of filters	16, 32
Filter width	3, 9, 15, 35
Number of models	5

Resulting accuracies for each of the 5 trained models per configuration are averaged and displayed in Figure 5.10. For clarity, results for filter widths of 3 having been excluded from these figures, but can be accessed in Appendix A.2 – short filter widths have proven to work very poorly on this problem.

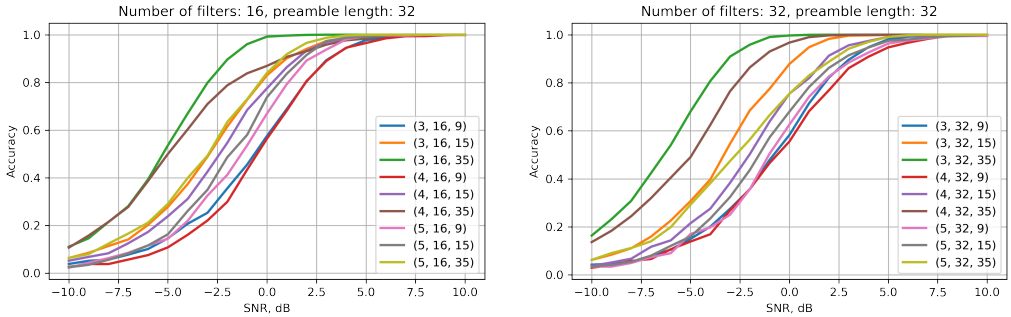
A few concrete conclusions can be drawn from the architecture sweep. Firstly, long filter widths are preferred to shorter ones. In the case where 32 filters are applied



(a) 8-bit preamble detection



(b) 16-bit preamble detection



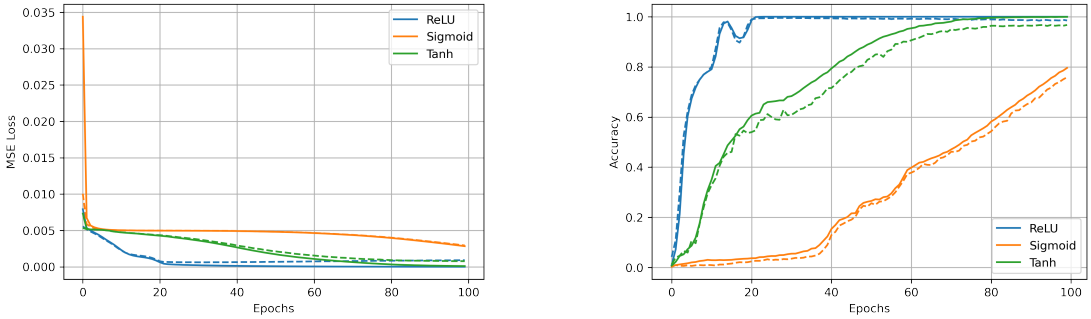
(c) 32-bit preamble detection

Figure 5.10: Accuracies achieved with different FCN model parameters. Legend key should be read as (number of layers, number of filters, individual filter widths).

per layer, the architectures with 35 sample wide filters worked best for each preamble length. More layers can make it more difficult for the FCN to converge to a solution; based on results in Figure 5.10, 3 layers has been shown to be sufficient in all cases. It is worth noting that these results have been achieved using a modest dataset of 1024 examples, in order to obtain some intuition on what works. The dataset, architecture and optimization strategy can always be iterated on to improve performance.

Activation Functions

One of the major buildings blocks for DNNs are the various activation functions introduced to the individual layers to add nonlinearity to the system. In the following experiment 5 models are trained per activation functions for the three functions being explored: ReLU, Sigmoid and Tanh. The same 1024 example dataset is used from the architecture sweep. The results are captured in Figure 5.11.



(a) Training losses

(b) Training accuracies

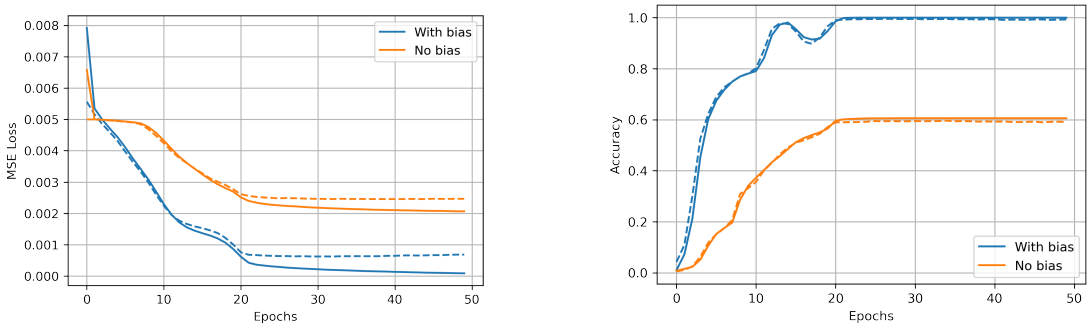
Figure 5.11: Training results on an 8-bit preamble dataset using three different types of activation functions (dashed lines show validation losses)

From the results in Figure 5.11 it is clear that ReLU demonstrates the fastest convergence times and reaches the maximum training and validation accuracies in the least number of epochs. Sigmoid and Tanh functions tend to limit the output to a maximum value of 1, weakening some of the stronger error signals being backpropagated during training. The transformation required to convert input samples to a vector containing sparse peaks (or in this case a single peak) is quite simple and the complexity added by these activation functions (Sigmoid and Tanh) may be unnecessary.

Bias

What differs between typical convolutional layers in a DNN, and receivers based on correlation, is the addition of the bias term. Since standard correlation does not have an extra bias added to the equation, the question arises whether learning correlation would be easier for the DNN without this addition.

Figure 5.12 shows the average results of 5 trained models with the FCN bias enabled

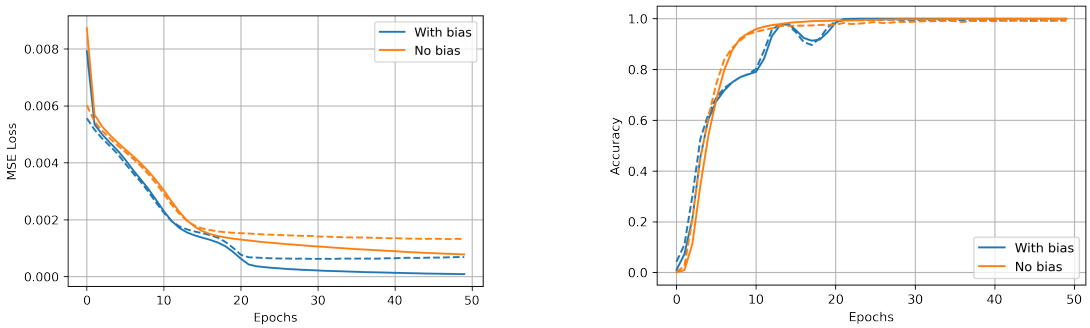


(a) Training losses (b) Training accuracies

Figure 5.12: FCN training results with and without bias in all convolutional layers

and disabled. It is evident that removing bias from all the layers reduces the predictive power of the DNN and degrades the training process. This makes sense – the number of parameters has been reduced, which reduces the expressivity of the model.

What about just the first layers? As will be shown near the end of this chapter, the first layers of the FCN tend to learn approximations of the preamble sequence, mimicking individual matched filters. In order to mimic correlation, one does not need the bias term. Another experiment is run where 5 additional models are trained with only the first layer having the bias parameter disabled. The results of this experiment are shown in Figure 5.13.



(a) Training losses (b) Training accuracies

Figure 5.13: FCN training results with and without bias in the first layer

From the results in Figure 5.13, it is clear that with the bias of the first layer omitted, the training process will be slower than the default alternative, albeit the FCN will still be able to converge to a solution. If trying to optimize the DNN size

in the parameter space, some biases could be thrown away to make minimal memory savings, however compared to the number of parameters in convolutional and fully connected layers, number of terms representing biases are negligible.

Selected FCN and CNN architectures

This chapter mainly focuses on FCNs, but for comparison a CNN baseline model was also produced for performance comparisons. A high level overview of the two architectures can be observed in Figure 5.14.

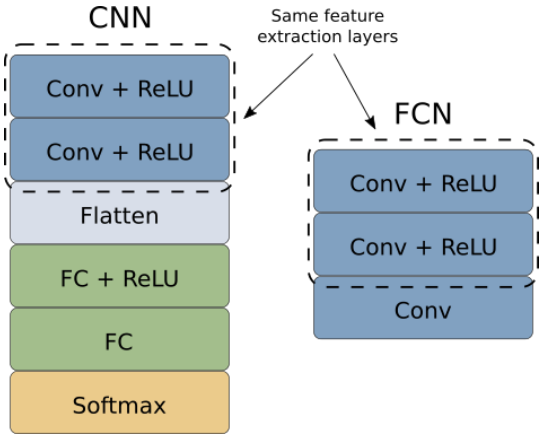


Figure 5.14: CNN and FCN architectures overview

The FCN is composed entirely of convolutional layers, where each layer has enough zero-padding to ensure that the outputs are of the same length as the input. In total, the model contains 3 layers, where the first two are followed up with ReLU activation functions in order to add non-linearity to the model. 2D convolutions are used so the first layer can take advantage of the phase information of the signal by processing in-phase and quadrature samples simultaneously. Kernels of sample width of 35 have shown best response to multiple preamble lengths (from results in Figure 5.10) and will be used for all convolutional layers. The final layer outputs a single channel response from which the frame offset can be inferred.

The reference CNN model shares the exact same first feature extraction layers as the FCN model, however these are then flattened into a single vector representation to be consumed by an additional 2 FC layers for further processing. The output of

the final FC layer is followed by a softmax activation, as is typical of a classification architecture. The CNN is of comparable size to those presented in [33], [35].

Table 5.2: FCN Parameters

Layer	**Parameters**	**Output Shape**
Input		$2 \times N$
Conv2D + ReLU	35×2, 32 filters	$32 \times 1 \times N$
Conv2D + ReLU	35×1, 32 filters	$32 \times 1 \times N$
Conv2D	35×1, 1 filter	$1 \times 1 \times N$

Table 5.3: CNN Parameters for input length 200

Layer	**Parameters**	**Output Shape**
Input		2×200
Conv2D + ReLU	35×2, 32 filters	$32 \times 1 \times 200$
Conv2D + ReLU	35×1, 32 filters	$32 \times 1 \times 200$
Flatten		1×6400
FC + ReLU	Size = 128	1×128
FC + Softmax	Size = 200	1×200

A summary of the FCN model parameters is presented in Table 5.2 and, for comparison, a CNN model is summarized in Table 5.3. The dimensionality and the number of neurons in each FC layer of the CNN will change based on input size, hence the parameters are specified for a reference input of 200 samples.

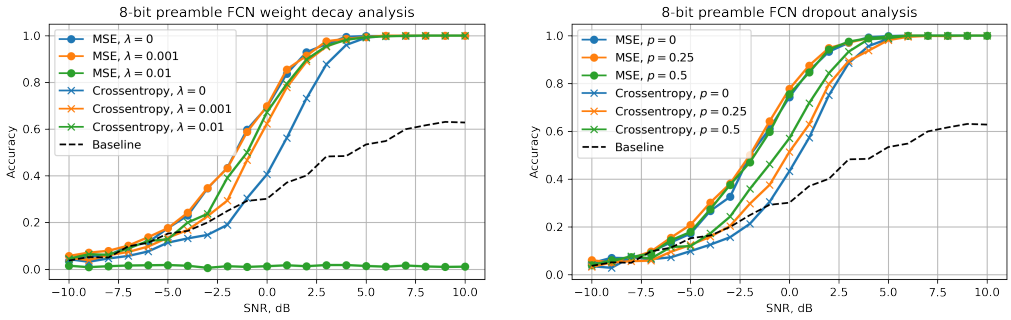
5.4.3 Training

This section will consider some of the remaining key hyperparameter choices for training the FCNs. Regularization is an important part of training a robust model and there are many techniques, but in this section some of the most popular choices for weight decay and dropout will be explored.

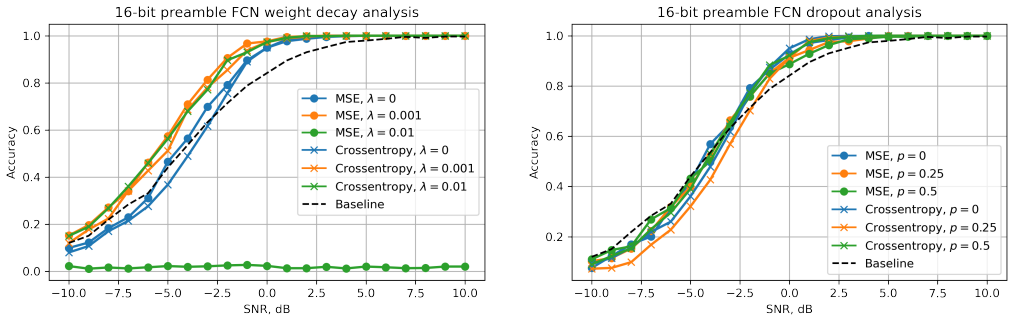
Loss Functions and Regularization

The way the training data is formatted lends itself perfectly to categorical cross-entropy, since the desired labels are one-hot encoded due to the nature of the dataset. While not addressed in this chapter, one caveat of training with cross-entropy is that it may not perform as well if the training data consists of more than a single packet, making

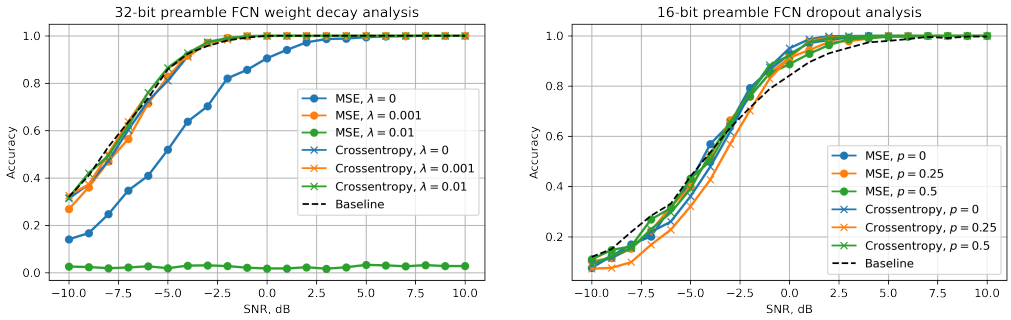
the data no longer one-hot encoded. In this case, MSE might be more appropriate. Nonetheless, both loss functions have been evaluated and compared with two regularization techniques – weight decay and dropout. In the following experiments regularization factors $\lambda \in [0, 0.001, 0.1]$ are used for weight decay and probabilities $p \in [0.25, 0.5]$ for dropout.



(a) 8-bit preamble detection



(b) 16-bit preamble detection



(c) 32-bit preamble detection

Figure 5.15: Accuracies achieved using different regularization techniques (left-hand side shows weight decay results, and dropout on the right).

In order to evaluate the effects weight decay has on the FCN models, for each regularization factor 5 models have been trained and evaluated over an SNR range of -

10dB to 10dB for each of the 6 regularization/loss combinations. The mean accuracy of all 5 models per regularization/loss function configuration is displayed in the left-hand side graphs of Figure 5.15.

Interestingly, MSE tends to work better for shorter preambles (Figure 5.15 (a)), then as the preamble length increases cross-entropy becomes a clear winner, as seen in Figure 5.15 (c). It is evident that MSE is much more sensitive than cross-entropy to weight decay: once λ is set to 0.01, none of the FCN models are able to converge to a solution for any preamble length. MSE with moderate ($\lambda = 0.001$) weight decay performs better than cross-entropy, on average, however for the longest 32-bit preamble it cannot provide a satisfactory solution, since the accuracy result falls below baseline correlation.

Similarly to weight decay, dropout has also been evaluated for dropout probabilities of $p \in [0.25, 0.5]$. The average accuracy of 5 trained models per dropout configuration and loss function are displayed in the graphs on the right-hand side of Figure 5.15. The results of using dropout are firmly worse than weight decay, this might be due to the fact that the FCN models are fairly small and the missing connections caused by dropout may hinder the training too much by not having enough neurons to successfully learn the task. Since the FCN models are small, compared to other common DL architectures, weight decay appears like a more appropriate solution.

In general terms, it seems that using cross-entropy and having a modest amount of weight decay will provide best overall results.

Early Stopping

A simple form of regularization that can often be overlooked is early stopping or checkpointing. Early stopping is implemented by constantly monitoring the validation loss and saving the weights of the network only when a lower loss has been computed than the previous best result – an example of early stopping based on validation monitoring for the CNN model is illustrated in Figure 5.16.

Without early stopping, it can be easy to overfit the model to the training set, making it incapable of generalizing to new inputs. The produced model for inference

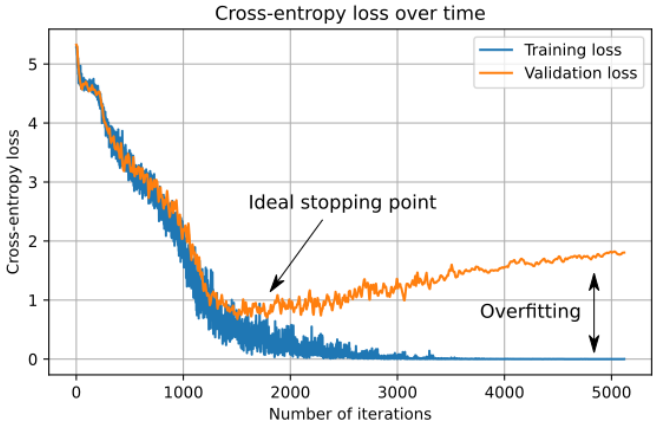


Figure 5.16: CNN training and validation losses over batch iterations

is the one that has resulted in the lowest validation loss.

Training Parameter Summary

The training details for both FCN and CNN models are summarized in the Table 5.4. The models were trained for 3 different preamble sequence lengths of 8, 16 and 32 bits. Each model was trained for 30 epochs on datasets of decreasing SNR level based on preamble length – detecting longer preambles is an easier problem, hence it is more tolerant of noise. The training model sizes in terms of parameters were kept as close as possible for both FCN and CNN architectures for a fair comparison. Some hyperparameters, such as optimizer choice, learning rate and batch size have not been explored, however the defaults proved to be good enough for training performant models. Of course many further optimizations are possible to the architectures, dataset quality and training hyperparameters.

A key difference between the two sets of architecture training parameters is the number of training examples – using the outlined training parameters, it was difficult to achieve the same average accuracy on a CNN compared to an FCN without increasing the dataset size. For this reason the dataset size for CNN is 4x the amount used for the FCN training. In these experiments the dataset has been synthetically generated, however this is not always the case. Additionally, achieving the same or better result with less data is a desirable architectural property.

Table 5.4: Training hyperparameters

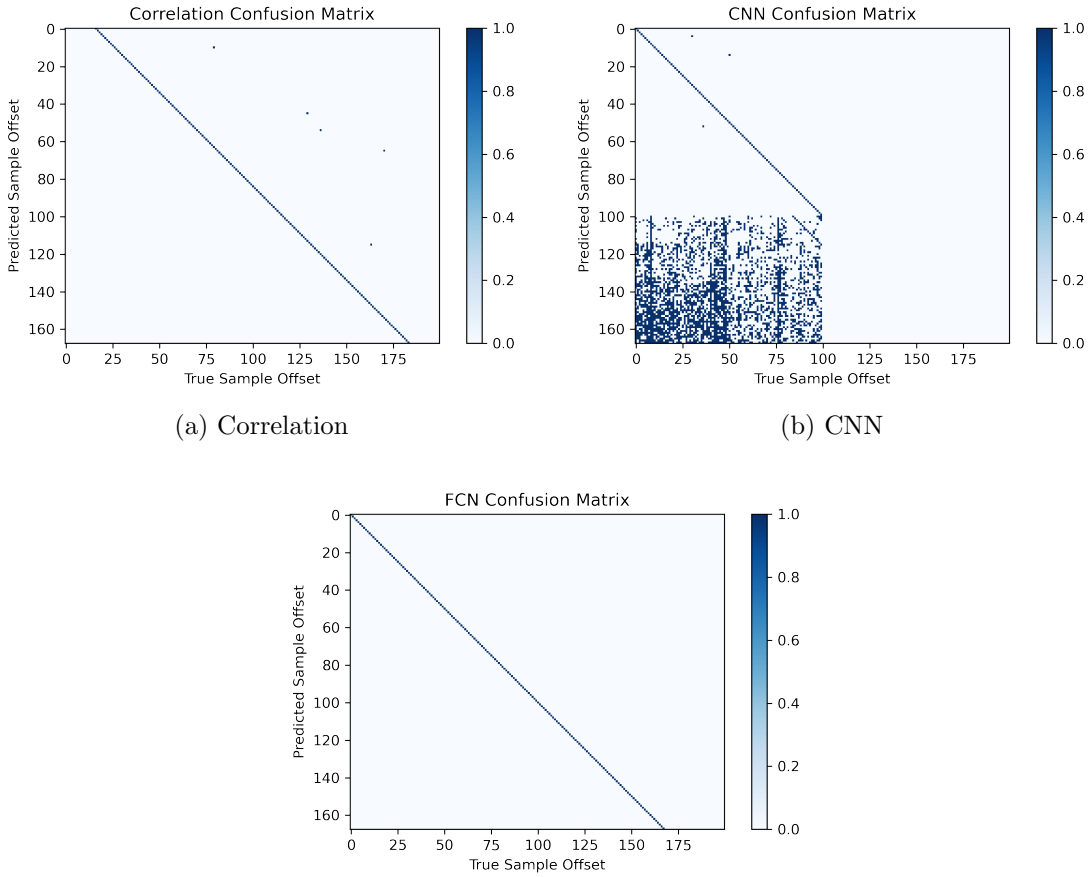
Parameter	**FCN**	**CNN**
Optimizer	ADAM	ADAM
Loss function	Cross entropy	Cross entropy
Number of training examples	8192	32768
Batch size	32	32
Number of epochs	30	30
Learning rate (α)	0.001	0.001
Regularization Factor (λ)	0.01	0.01
Training SNRs (dB) for 8, 16, 32 bit preambles	10, 5, 0	10, 5, 0

Class Imbalance Constraints on CNNs

In many ML applications, the class imbalance problem manifests itself when there is an unequal distribution of all classes in a training dataset. Meaning that the trained DNN will not be as good at predicting the under-represented classes [116]. While deep CNN models can show excellent performance on frame synchronization tasks, they are very sensitive to class imbalance – if a single time offset is not present in the training dataset, the network will not be able to generalize and make that prediction – all classes (time offsets) have to be present in the training set to make sure the final layer weights going into softmax are not 'dead' (0 weights). For this reason, the CNN requires more training data to match the FCN performance, as summarized in Table 5.4.

To illustrate the point, an FCN and CNN have been trained on a dataset containing waveforms that are 200 samples long, with maximum reprsented offset τ of 100, with a 32-bit PN sequence preamble. Then both networks, together with standard correlation for reference, are tested for each time offset from 0 to 168 at 0dB SNR. The results of this experiment are summarized in Figure 5.17.

Figures 5.17 (a) and (c) show an ideal diagonal where the model predictions (y-axis) match the expected offset value (x-axis). Correlation has a "snowy" pattern above the diagonal – the reason the misclassifications exist only above the diagonal is that there are payload symbols following the preamble, and the random bits may almost match the preamble sequence, causing some false positives. The noise is not high enough to cause false positive below the diagonal, where no data bits are being transmitted.



(a) Correlation

(b) CNN

(c) FCN

Figure 5.17: Confusion matrices of Correlation, CNN and FCN approaches to frame sync.

More striking is the result in Figure 5.11 (b), where the CNN is unable to generalize to any offset past $\tau = 100$. This is an expected result – the training data did not have a single example (class) above a time offset of 100 samples, so the weights connected those 100 neurons never needed to be updated. Since the dataset is synthetic, this could be solved by adding more training examples – however it incurs more training-time costs. In cases where it is not as trivial to acquire more data (because it was collected off the air or from a third party), methods like SMOTE (Synthetic Minority Over-sampling Technique), used for balancing a dataset where each class is not equally represented, could provide a solution [117]. In the meantime, the FCN model does not have this limitation, and processes the input like a filter would.

5.5 Empirical Evaluation

In the following sections, the FCN architecture is tested under various impairments and the DERs (Detection Error Rates) compared with baseline results from correlation methods and implementations of the improved noncoherent correlation method from [110] – this baseline reference will be denoted as 'expert' in the legends of result graphs.

The impairments that the FCN will be tested on include AWGN, carrier phase and frequency offsets, and fading channels. In total 3 preambles based on PN sequences are investigated, and for each preamble 5 FCN models trained, with the best model chosen for evaluation. Data packet simulated transmissions are assumed to be 200 samples long, with the payload containing 128 BPSK-modulated symbols.

5.5.1 AWGN and Phase Offset

One of the most fundamental impairments for evaluating models is an AWGN channel, which allows an analysis of the model at various SNR levels and an initial assessment of model robustness. This basic test provides some insight into how FCNs could be used quite effectively for short preambles. FCNs trained for 8,16 and 32-bit preamble sequences are evaluated at an SNR range from -10 to 15dB, and random phase offsets. They are also compared with standard noncoherent correlation and an improved method from [110], with the results summarized in Figure 5.18.

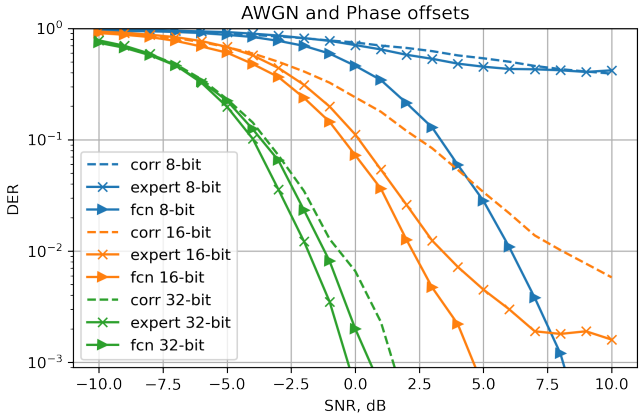


Figure 5.18: DER results with AWGN and random phase offsets

It is clear that the FCNs work very well for smaller preambles – in these situations a lot of false positives can degrade performance because of the non-zero likelihood of the preamble sequences appearing in the 128-bit payload data. This becomes less of a problem for longer preamble sequences and the advantage of using a DNN becomes less apparent.

Careful not to Overfit Phase!

Phase offsets do not affect existing methods in terms of DER. A neural network, on the other hand, suffers greatly if it is not trained on a variety of phase offsets, as it can easily overfit to a single phase. This is demonstrated in Figure 5.19, where an FCN trained exclusively on data with $\phi = 0$ is compared against the standard correlation method. The DER achieved by correlation does not change, regardless of the phase offset applied. However the FCN performance at an unseen phase is drastically affected.

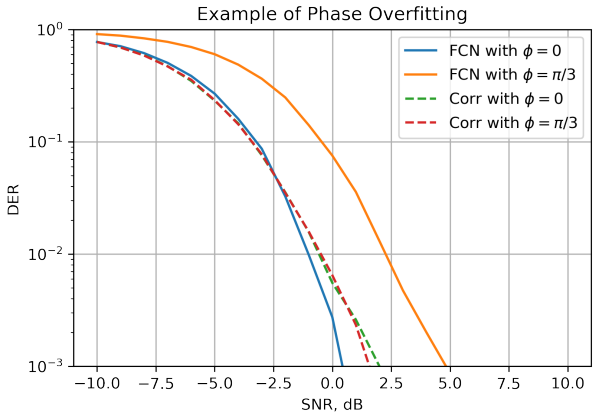


Figure 5.19: Single phase overfitting

Concretely, one can think of bursty BPSK transmissions in a high SNR channel – if there is no phase offset, the network will only have seen actual data in the in-phase channel, whereas anything on the quadrature line will be noise. Once there is a phase offset, some of the in-phase channel data will "leak" into the quadrature channel, a situation which the DNN has never seen before. To the neural network this situation is merely the reduction of power on the in-phase channel, degrading the receiver performance. This is why it is important to train and test any data-driven

receiver solution on various phase offsets.

5.5.2 Carrier Frequency Offset

While phase offsets do not affect standard correlation implementations, CFO certainly does. In any realistic scenario there will be some CFO effects to compensate for. Depending on the communications link setup, the first step in order to compensate for the CFO effects might be frame synchronization. For example, in 5G NR communications, the PSS (Primary Synchronization Signal) is detected first, which enables coarse carrier frequency synchronization [112, Chapter 16]. This makes it all the more important that the method of reception is robust to carrier offsets.

Much like AWGN channel experiments previously, in Figure 5.20 the same methods are compared for multiple PN sequence lengths and a 128-bit payload, all BPSK modulated. Except in addition to the random phase offsets, the waveforms are suffering from a 10kHz CFO, sampled at a rate of 1MHz sampling rate.

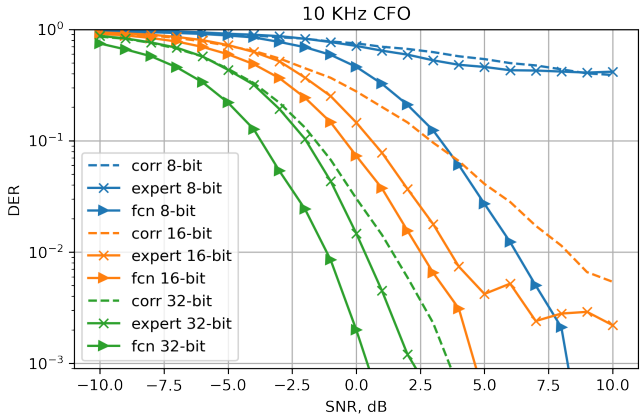


Figure 5.20: DER with CFO of 10KHz

Now the advantages of the DNN approach are a little more apparent, even at the longer 32-bit preamble, a ∼3dB gain in performance is achieved. As before, the shorter preambles are detected with a much greater accuracy than the classical methods. It is clear that the DNN approach works well for CFO. To further evaluate how well, it is pertinent to evaluate the sensitivity to CFO for the FCN and reference methods.

In order to determine the CFO sensitivity of the proposed method, the same ex-

periment was run for a 32-bit preamble sequence, with the SNR fixed to 5dB and an increasing CFO value. In this instance the FCN is compared against another DNN model - the CNN, as well as the previous noncoherent correlation methods. The results are summarized in Figure 5.21.

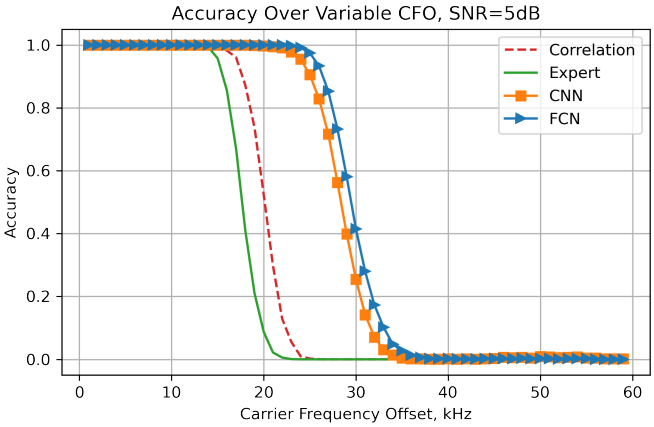


Figure 5.21: CFO Sensitivty Comparison

Clearly there is a performance gap between the traditional approaches and both DNN models, with the DNN models performing better. While both traditional approaches perform similarly in this test, the improved reference method actually showed more sensitivity to CFO and had a quicker drop off in accuracy as the frequency offset was increased. The DNN models show an almost 10kHz greater tolerance to frequency offset than the classical approaches.

5.5.3 Fading Channels

Some of the most challenging channels encountered in a deployment will have some form of fading. In this subsection, the FCN model and reference methods are evaluated under a non-line-of-sight (NLOS) and line-of-sight (LOS) channels modelled by flat fading and multipath fading channels, respectively.

Flat Fading

The same preamble and payload configuration is used in this experiment as previously. However, instead of carrier offsets the signal is transformed by a single tap complex

gain that follows a Rayleigh distribution. Since a single tap channel simply applies a gain and a phase offset, the results with flat fading (Figure 5.22) are similar to the ones seen with just AWGN and phase offsets in Section 5.5.1.

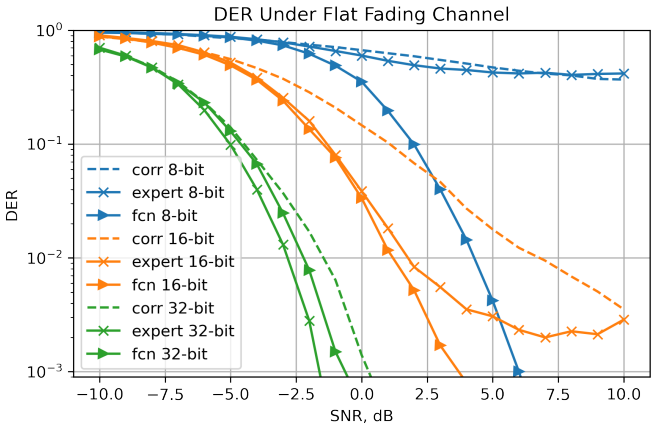


Figure 5.22: Detection Error Rate under Flat Fading Channel

In this instance the expert correlation-based method outperforms the FCN for the longer preamble sequence of 32 bits. The FCN approach consistently outperforms baseline methods for shorter preamble lengths, matching 16 bit correlation performance with just an 8 bit preamble at higher SNR.

Multipath Fading

Multipath fading is more complex than a single tap flat fading channel, and can be very detrimental to single carrier transmissions. Because there are multiple paths, the expected performance will be even worse than flat fading. Nonetheless it is still important to confirm that the FCN model is capable of generalizing to this unseen impairment in the dataset (training data only included CFO and phase offsets).

The multipath channel that the frame synchronization methods are being tested with follows a Rician distribution, and consists of 3 channel gains at time offsets of [0, 5e-6, 10e-6] seconds and average channel gains of [0dB, -3dB, -6dB], with a sampling rate of 1MHz. The evaluation results under this channel are summarized in Figure 5.23.

Again, the most marked improvement can be seen from the 8-bit preamble detec-

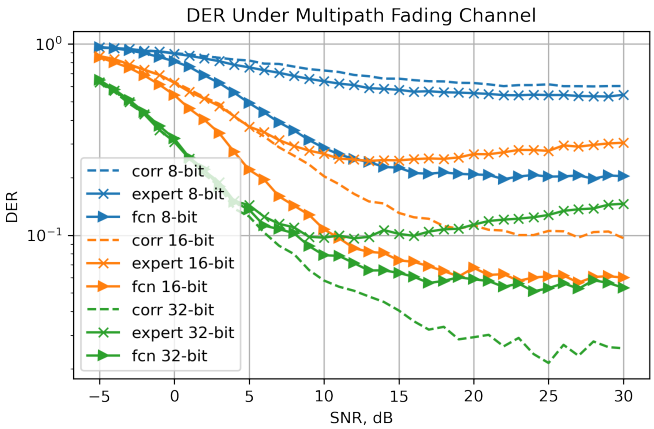


Figure 5.23: Detection Error Rate under Multipath Fading Channel

tions by the FCN. As the preamble length increases, the FCN results exhibit progressively less improvement over the traditional approaches. One important point to note is that the FCN model has never seen flat or multipath fading in its training dataset. Given this fact, it is still capable of not only matching but showing improvements over traditional methods for shorter preamble lengths.

5.6 FCN Architecture Introspection

Most DNNs fall in the category of "black box" models, meaning that even though they can achieve very good results, their interpretability is low [20]. The opaqueness of DNNs complicates fault diagnosis and regulatory compliance, crucial in environments where precise control and explanation of algorithmic decisions are essential. There are ongoing research efforts in explainable AI that show potential to solving these issues, such as LIME (Local Interpretable Model-Agnostic Explanations), which determines the positive and negative impact individual features have on a classifier's decision, by approximating a local interpretable model [18]. Another example is SHAP (SHapley Additive exPlanations) in which an importance value is added to each feature for a specific prediction, revealing how each contributes to the outcome [19]. Explainability is fundamental to the transparency and trustworthiness of complex models, and should be necessary for faster adoption of DNNs in production.

In this section we will take a closer look at the individual learned filters of the FCN layers to see how it is abstracting the incoming complex signals in order to make decisions. A multi-packet inference analysis is also carried out to visualize the FCN model transients, which are similar to regular DSP filters, albeit different.

5.6.1 Learned Filters

Passing a 200 sample frame with a single preamble located at an offset of 40 samples, and a payload of 128 bits, the intermediatery outputs of the FCN are inspected in Figure 5.24. Interestingly, looking at the filter outputs of the first convolutional layer, layer 1, the output seems very noisy and there are many peaks, however not a single filter actually outputs a strong correlation at the expected offset of 40. This is a different from the second layer outputs, where it seems like many filter outputs are stacked at the expected offset location.

Figure 5.25 provides a closer look at what the learned filters actually look like. The first layer filters in Figure 2.25 (a) are almost recognizable as the expected preamble sequence. To make this a bit clearer, one of the filters has been overlaid with the 32-bit PN sequence in Figure 5.26. Clearly the pattern is there, however not matching in its entirety.

Recall that the FCN has never seen a clean preamble sequence, and in this particular instance it has been trained on 0dB SNR data. Given that other first layer filters exhibit similar patterns and offset peaks at various locations, it is safe to assume that the first layer network learns multiple representations of the preamble sequence at different positions.

The second layer filters, shown in Figure 5.26 (b) are much more difficult to interpret, as they are multi-dimensional, now with 32-channels (however only 16 are displayed for clarity) rather than just the two for in-phase/quadrature samples. Nonetheless, some patterns can be observed by taking a closer look at individual filter outputs, as opposed to the learned weights, in Figure 5.27. Looking at the upper row, Filter 2 and 3, the outputs are what one would anticipate, with a clean peak at the expected sample offset of 40; the next filter over does not give as clean a prediction, but the

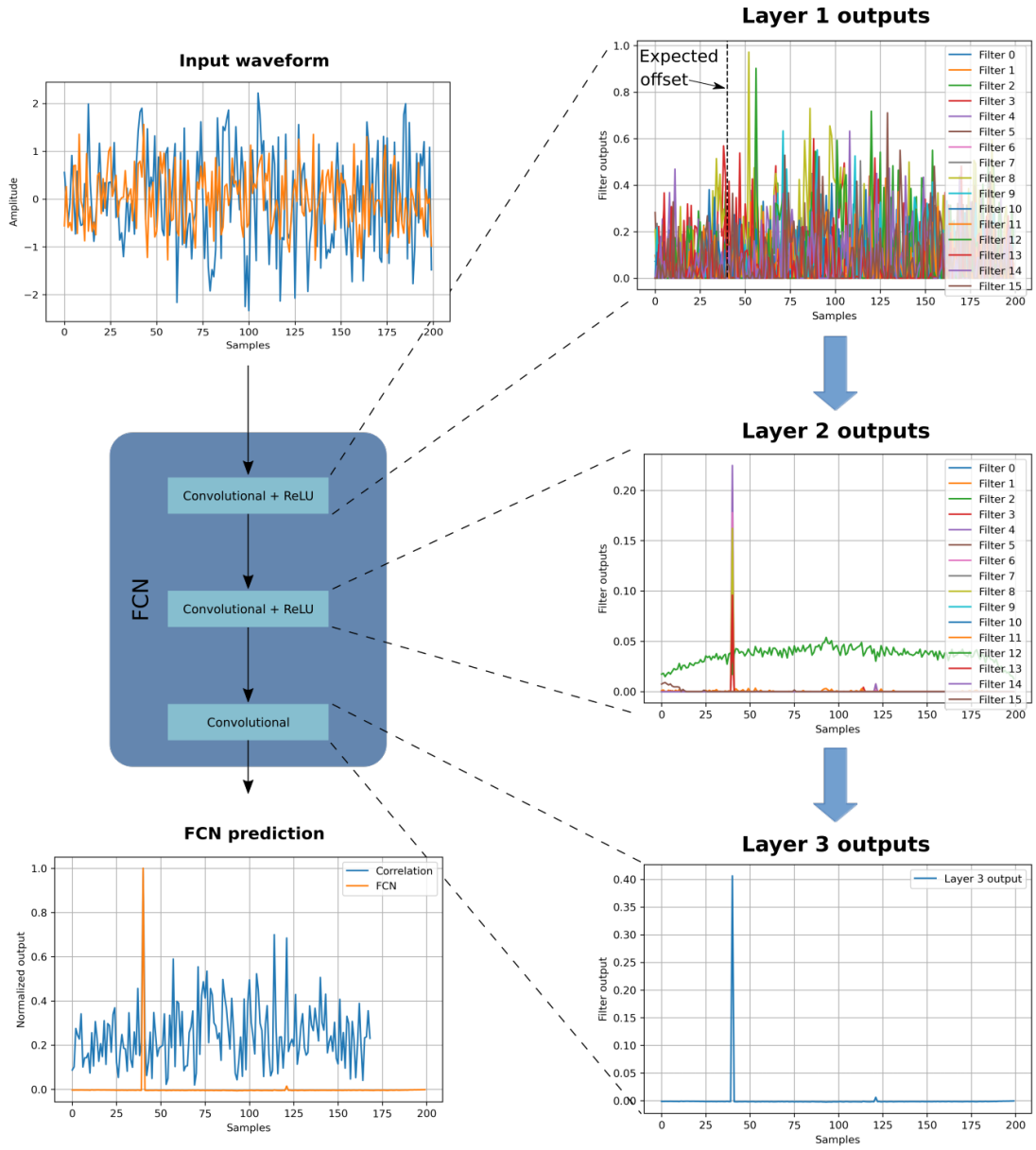
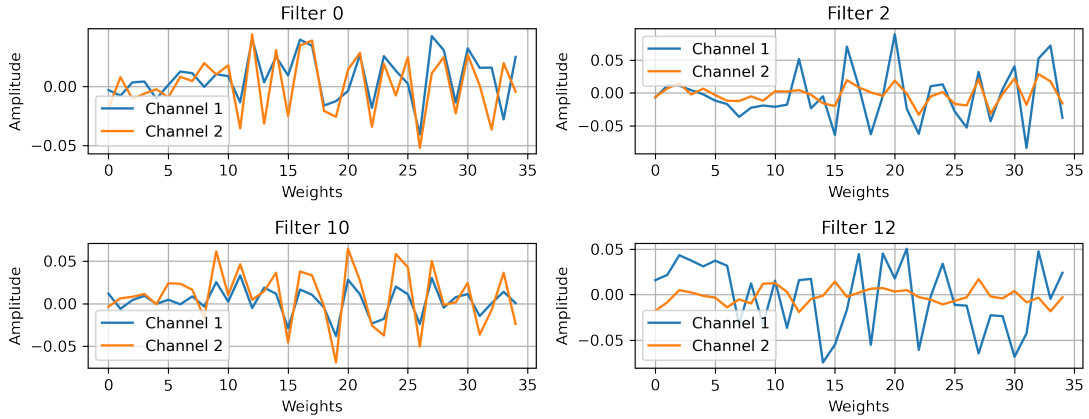


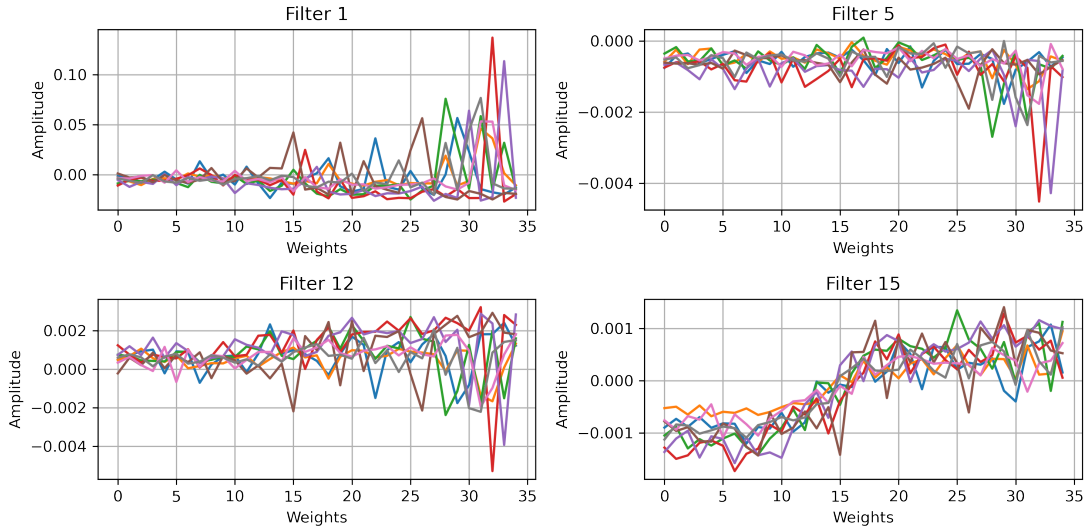
Figure 5.24: FCN introspection – looking at the outputs of individual layers

argmax value is still the correct offset.

More interestingly, the bottom two subplots in Figure 5.27, for Filters 5 and 12, show some unexpected peculiarities. Filter 5 produces absolutely no output, rather a vector of zeros – this could be the equivalent of a dead neuron, and with further optimization could be pruned from the network to save resources. An alternative interpretation is that whatever stimulus it is responsible for reacting to simply was not present in the



(a) First FCN layer



(b) Second FCN layer

Figure 5.25: Individual learned filter weights

input signal (perhaps this particular filter is only sensitive to certain phase offsets).

The other peculiar observed output comes from the last filter in Figure 5.27, Filter 12, which initially was dismissed as just producing noise. However upon further inspection, it was seen to produce an inverse peak at the expected output. This was confirmed by performing an argmin function on the filter output.

One reason as to why this particular kernel decided to learn to produce an inverse detection is that a DNN in the early layers does not particularly care whether a correlation is positive or negative; after weight initialization by random chance it could

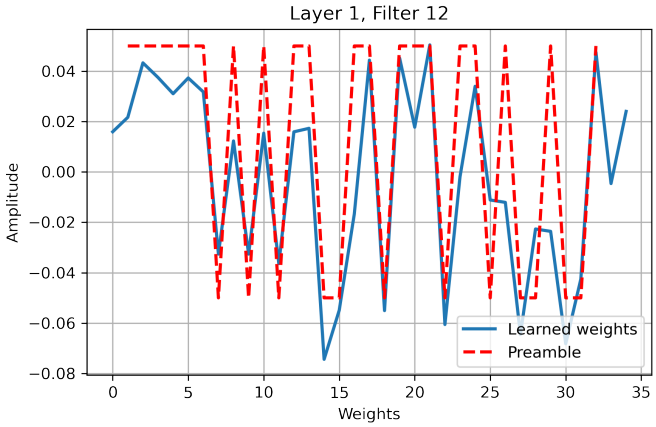


Figure 5.26: First layer filter similarity to preamble in training set

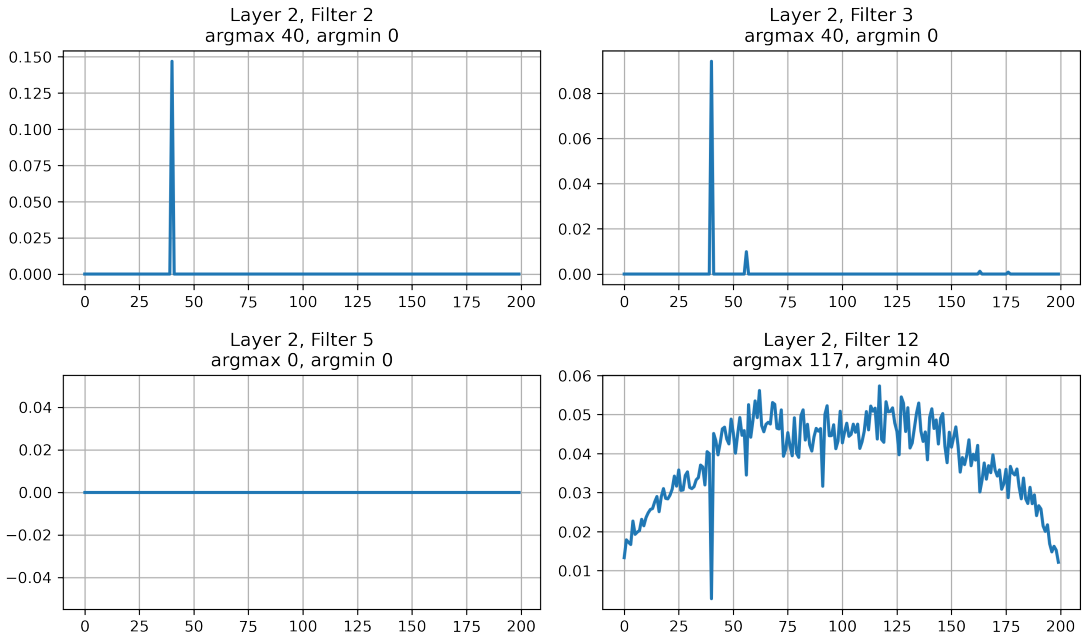


Figure 5.27: Second layer filter outputs

have started out with weights that resulted in a strong negative correlation. If a kernel is already leaning towards a negative correlation, during training it might be easier to reinforce this response and have one of the higher level layers invert the output of this filter, rather than retrain the whole kernel.

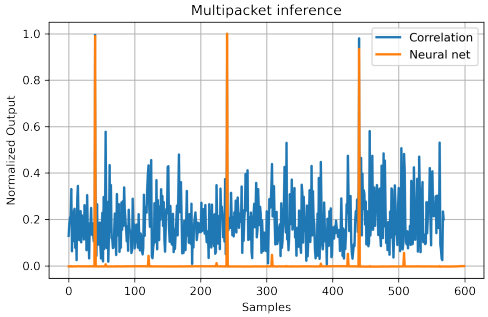
Being able to dissect DL models can give valuable information and insights into how they are learning. The FCN presented in this work is only a 3-layer convolutional

network, where each layer contains no more than 32 filters. The small size of the network makes introspection and visualization easier than some other DNNs. In future work a deeper dive with a tool like the DALEX toolkit [118] could prove to be even more useful into gaining more insight of how the model operates.

5.6.2 Multi-Packet Inference

One of the biggest advantages of using an FCN over a fully connected network is that it is actually possible to run inference on a frame containing more than 1 packet. As an additional benefit of using a non-linear model to achieve this, it can be observed that the FCN performs additional denoising functions when compared to the baseline correlation output. Inference of a trained FCN (on a 32-bit preamble sequence) is demonstrated in Figure 5.28 (a).

The FCN was trained on 200 sample sequences with one preamble in each one of them. In this case it is presented with a sequence spanning 600 samples, containing 3 packets. The output, by comparison to the correlation approach, is a lot cleaner with 3 distinct peaks corresponding to the correct predictions of packet locations.



(a) Inference on a waveform containing multiple packets

(b) Averaged predictions to visualize the noise floor

Figure 5.28: Multi-packet inference

To quantify the denoising capabilities that the FCN models are demonstrating, the same experiment was run on 1000 different frames at an SNR of 0dB, then the correlation and FCN outputs were collectively normalized and averaged over all runs. The result is presented in Figure 5.28 (b). Given that the FCN has such a low prediction

noise floor, compared to the standard correlation method, allows a practical advantage of threshold setting. Usually to avoid false positives a reasonable threshold value is selected which, when exceeded, signals the receiver that a packet has been detected. From the graphs in Figure 5.28 it seems like this threshold could be set much lower for the FCN.

One more observation from the FCN outputs is that, interestingly, the transients of the FCN and correlation are inverted; this could be a byproduct of the FCN implementation – in order to produce an output of equal length to the input length, the input signal is padded with zeros. The sharp transition from the padded zeros to the noisy samples might be triggering an edge detection-like response.

5.7 Complexity Analysis

When building a wireless communications system, performance is not the only metric to consider – implemented algorithm complexity is part of the tradeoff. The word *complexity* can mean multiple things, and there are a number of ways of evaluating it, depending on the hardware the algorithm is being implemented on. In this section, Computational Complexity (CC) is defined as the number of flops (floating point operations) required to perform a single inference on one waveform. Computational complexity can also be approximated by running the model on a CPU and measuring the time it takes to run.

For DL models it is also very important to evaluate the required memory capacity – DNNs typically require many weights to be stored on-device or even in on-chip memory. Additional memory requirements will incur a higher cost and potentially energy expenditure due to more memory accesses being required.

It should also be acknowledged that performing a fair comparison between 2 DNN models is challenging, because there are so many deployment modes and ways of implementing the architectures. In this work, a best effort was made to keep the number of parameters (for 200 sample inputs) as even as possible between the FCN and CNN. Due to the huge optimization space involved it cannot be guaranteed that either model is optimal.

5.7.1 Computational Complexity

For basic network layers it is fairly straightforward to calculate the CC in flops. For example, as defined in [119], a 2D convolutional layer can be determined by

$$CC_{conv} = (2 * (c * l * h)) * k * W * H, \tag{5.2}$$

where CC_{conv} is the computational complexity of a 2D convolutional layer, c, l and h, are the convolutional layer kernel parameters for input channels, filter length and height respectively, k represents the number of filters, and W and H are the width and height of the input signal, or feature map. Similarly, the complexity of a fully connected layer CC_{fc} is calculated using

$$CC_{fc} = 2 * w_{in} * w_{out}, \tag{5.3}$$

where CC of a fully connected layer is the product of input vector length w_{in} and number of output neurons w_{out}. Note that equations (5.2) and (5.3) do not take into account other building blocks of the architecture, such as activation functions. Using these equations does help estimate the complexity impact the DNN will have on the overall system.

For baseline reference, the CC associated with correlation is simply calculated as

$$CC_{corr} = 2 * c * l * W * H, \tag{5.4}$$

where c and l are the number of channels and length of the matched filter respectively, and W and H are the width and height of the input signal. The matched filter only needs to be evaluated once, since there are no layers, as would be typical of a DNN.

Using the FCN and CNN parameters from Table 5.1 and calculating the CC of each layer using the above equations, the number of flops per model is calculated and summarized in Figure 5.29. While this does not perfectly reflect the realities of hardware [120], it provides an approximation of the expected computational costs for these types of architectures.

Computational complexity can also be estimated by measuring the computation time on a target device, such as a CPU. In this instance, each method was evaluated on

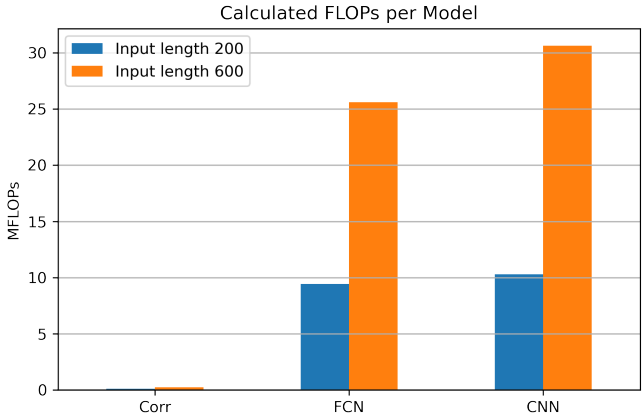


Figure 5.29: Calculated computational complexity for 200 and 600 samples of evaluated FS methods

a desktop AMD Ryzen 5 CPU, with results as summarized in Figure 5.30. This by no means reflects the realities of other typical hardware platforms that a DNN might be implemented on, such as TPU (Tensor Processing Units), GPUs or FPGAs. However, it is useful to compare this with the expected trends of the analyzed detection methods.

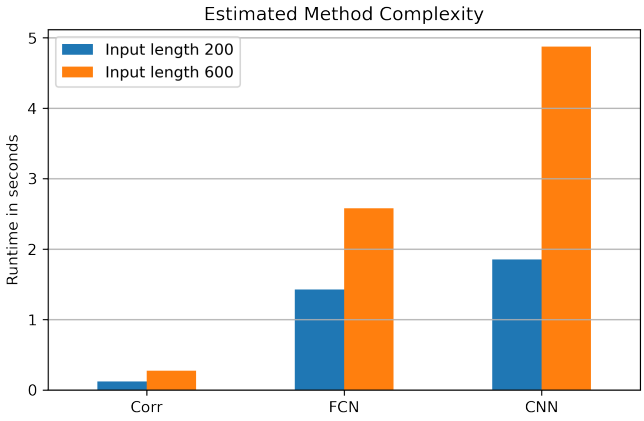


Figure 5.30: Estimated computation complexity in CPU runtime

Looking at Figure 5.30 the trends are quite similar to those of the calculated results displayed in Figure 5.29. Clearly both DNN models are more costly than standard correlation, however the FCN seems to require much less computation as the input size scales to 600 samples. The higher cost of the CNN when scaling the input size can be explained by additional activation functions, such as softmax, and the increasing

number of neurons in the final layer required to represent the many potential outputs.

5.7.2 Memory Requirements

When comparing DL models, a very important metric to consider is the model size, or number of weights that are required to store the DNN layers in memory. For convolutional and fully connected layers, this is very straightforward to calculate – simply sum up all of the weights in the model. While the computational complexity is comparable between the FCN and CNN models, looking at Figure 5.31 it is clear that the CNN is much more costly in terms of memory.

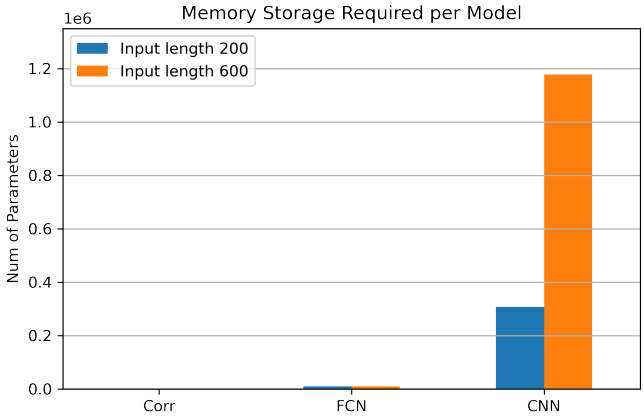


Figure 5.31: Required number of parameters for each detection method

It should also be noted that, since the correlation method uses a matched filter equal to the size of the preamble (32 samples), it is neglibible and does not even show up on the graph. Since the FCN employs only convolutional layers, its number of parameters does not scale as with the CNN. The fully connected layers of the CNN scale linearly with the input size and do not exhibit the weight sharing property, which is an important aspect of convolutional layers when considering implementation [121].

The scaling effect can be demonstrated by plotting the the number of parameters over an increasing input length, as shown in Figure 5.32. This compares the FCN, and the discussed CNN architecture with a softmax output layer, in Figure 5.32 denoted as “CNN + classification”, as well as a CNN with a single output neuron regression denoted as “CNN + regression”, scaling accordingly as the input size increases.

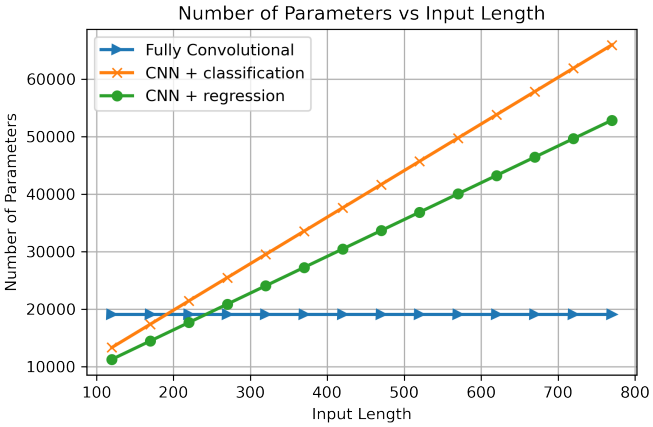


Figure 5.32: Parameters required with increasing input size

Based on these experiments, a case can be made for using FCNs when compared to CNNs. Power and memory constraints should be taken into account – while more compact than a CNN, FCNs are still an order of magnitude more demanding in both computational power and storage than traditional approaches like the matched filter. DNN deployment is an active research area and techinques like pruning and quantization [122] have been shown to be effective at minimizing the cost. As the field progresses more specialized hardware will also be made available, again, making DNN deployment more practical in many scenarios.

5.8 Chapter Conclusion

FCNs are relatively easy to train compared to similar DNN models, and work very well when detecting packets in bursty single carrier communications, such as those found in IoT transmissions. In the experiments of this chapter, it was shown that the FCN models excel when detecting shorter preambles, which is highly beneficial when trying to eliminate signalling overhead for a more efficient communications link. The DER improvements were most significant in channels containing CFO. While the training dataset never contained any fading channel data, the FCN models were still able to match the performance of baseline correlation methods and outperform these methods for shorter preamble lengths.

Previous research done on frame synchronization using models like CNNs and MLPs have also demonstrated performance improvements in various channel conditions, making a strong argument for the use of data-driven models for frame synchronization in future wireless receiver implementations. The work presented in this chapter showed how to harness the performance gains that have been displayed in previous DL implementations, while contributing novel flexibility to these DL models. A trained FCN can be treated like a deep filter and deployed on inputs of any size – this is a distinct advantage over other DNN methods, that contain fully connected layers. Importantly, this work demonstrates that the FCN is less likely to overfit to the training set than a CNN of a similar size, making this architecture easier to train and further reducing adoption costs.

While FCNs look promising compared to other DNN models, when compared to traditional approaches such as the noncoherent correlation detector, they are significantly more costly to implement. As the ML field progresses, more accelerators and techniques will become available for quantizing and deploying the DNNs efficiently. Eventually DL-based receiver designs should become a cost efficient alternative to traditional methods that can be deployed on ubiquitous hardware.

Next chapter will focus on training techniques that can improve DNN model performance with zero added implementation costs post-training.

Chapter 6

Multitask Learning with Channel Impairment Estimation

This chapter introduces the concept of Multitask Learning (MTL) for wireless communications tasks. Generally the data generation process for RFML will involve signal movement through multiple transmitter signal processing blocks, as well as multiple channel impairments – the transmitter functions and impairments will have parameters associated with them and can be treated as additional training data.

Usage of MTL in this chapter has a twofold motivation: improving model performance by introducing expert driven regularization resulting from additive losses; and saving resources by sharing common feature extraction layers between tasks, eliminating the need of training multiple models for different tasks.

The previous chapter introduced the FCN architecture for frame synchronization, this chapter explores how the FCN architecture can be enhanced even further with MTL. Examples of improving AMC and frame synchronization accuracies using MTL with additional impairment estimation are explored. MTL expands on some of the FCN functionality from the previous chapter by enabling continuous SNR estimation. Using MTL adds additional complexity to the training process, requiring to tune additional hyperparameters, however it is a promising new vector of optimization for wireless communications tasks like AMC.

6.1 Motivation

The term "multitask learning" was introduced in 1997 by R. Caruana [39]. Beforehand, the concept of MTL existed in the early 90's and has been referred to as "training hints", the idea being to give the model hints in the form of complementary tasks to learn, making the process of learning helpful features more explicit. The key principle behind MTL is that one can train multiple tasks (of the same domain) using a single DNN with a set of shared feature extraction layers. Which tasks are appropriate to group together is an actively researched ML topic [123]. Training related tasks simultaneously forces the shared layers to learn features that can be beneficial to other tasks, yet may not have been learned otherwise. It can be viewed as an expert-driven transfer of knowledge that occurs during training time.

MTL is an actively pursued area of research in many applications where machine learning is used. It is especially prominent in the field of CV [38]; most CNNs that work with images always have to learn basic features like edge detection or color thresholding – for tasks like image segmentation and contour detection, the shared layers will have many functions in common. MTL can also be challenging to implement, as it can be difficult to determine whether the tasks are complimentary and the gradients will align [124]. Additionally, a very important topic in MTL is the question of loss weighting – one task can be weighted higher than another, and balancing these weightings is necessary to achieve good results [125].

One of the reasons why MTL can be a good fit for wireless communications is the built-in capability of generating many additional labels, based on simulation parameters. Take the most typical scenario of AMC – by far the most popular approach to this problem is to use supervised learning, and generate data samples and labels, as shown in Figure 6.1. Normally some noise is added to the dataset, perhaps multiple different levels of SNR. The simulated channel used when generating the data could also introduce a carrier phase offset. While generating the dataset there could be a plethora of parameters used to generate specific waveforms, however the only parameters saved would be the labels ('BPSK', 'QPSK', 'GMSK', etc.) and the actual I/Q samples.

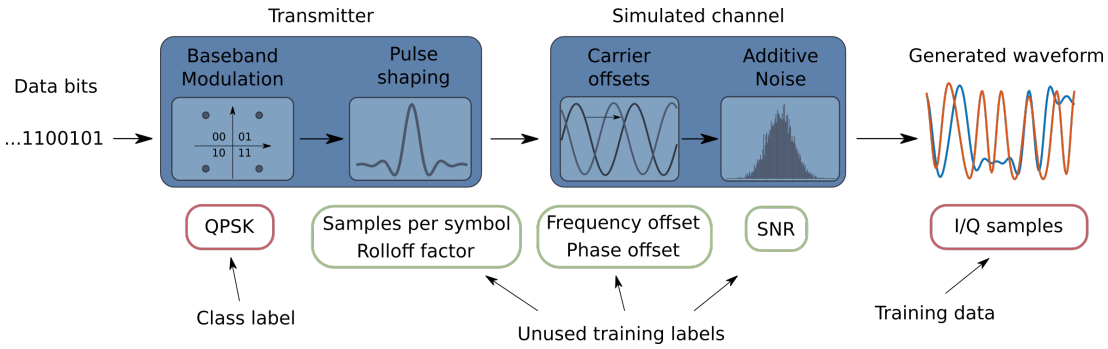


Figure 6.1: Typical data generation process for AMC, highlighting potential untapped data

What if, instead of throwing away the parameters that were used to generate the dataset (like SNR levels, or carrier offset values), these were used as additional training data for the DNNs? These parameters are not used in operation and will be discarded anyway after the dataset has been generated and the DNN has been trained. This is essentially free data, that is a byproduct of generating a conventional ML dataset. With MTL this does not need to be the case; the parameters used to generate the training data can be stored and re-used as additional labels for supplementary tasks within the same DNN. It should be mentioned that there are recent efforts in the open source community to create richer data format standards like SigMF [126] that can be used to store all of these parameters in new RFML datasets. Doing so would enable new interesting MTL research in signal processing, however, at the time of writing, these standards are yet to be widely adopted by the research community.

A high level overview of MTL is shown in Figure 6.2. The trunk of the DNN, closest to the input, has a set of feature extraction layers that are shared between each of the prediction heads that are responsible for learning their particular task. Each head can have its own unique architecture (different layer and activation function types), independent of the other heads. Importantly, but not necessarily, they can also have separate loss functions – for example, a classification head might use categorical cross-entropy, whereas a regression head with a single output would use MSE or similar.

The DNN will try to optimize all of its weights to reduce the sum of the losses, and so backpropagation will update the shared weights in such a way that, ideally, reaches

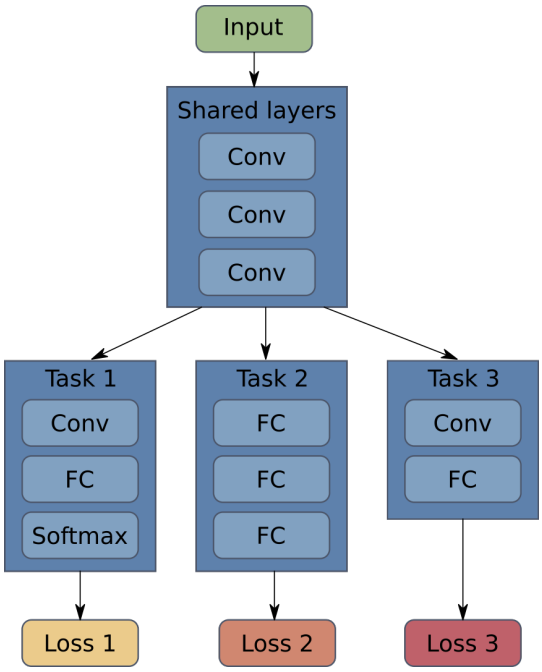


Figure 6.2: High level overview of MTL

a compromise between the prediction heads. The different prediction head tasks should be closely related, and learning one will benefit the other. Some tasks may be more important than others, which is where interesting tradeoffs in loss weighting can be evaluated. This will be discussed in more detail in the following sections.

6.2 Related Work

Applying the MTL technique to tasks like AMC with a secondary SNR estimation task has been shown to improve the quality of the resulting classifiers. A multi-headed DNN for AMC was explored in [43], with one head being used to perform a binary SNR classification – SNR is high or low. This work showed that even a coarse level of SNR estimation can be beneficial to the system as a whole, achieving better accuracies than baseline models. Similarly, the trained AMCs that rely on 3 SNR approximation buckets – low/mid/high SNR – have been explored in previous studies [42], [44]. A slightly different variation of MTL was proposed in [45], where a denoising autoencoder is used as the secondary task to AMC, which in itself does not estimate SNR, but the

training effect of making the model SNR-aware is similar.

Some MTL specific concerns are not addressed in the cited works – such as an analysis of different task loss function weightings. Training multiple tasks typically involves tuning their loss function weightings to find a compromise achieving the best results for both tasks. The work in this thesis aims to expand on this missing knowledge, and conduct a more thorough exploration of loss weighting tradeoffs for multiple SNR estimator types.

CFO should be considered in practical scenarios, because small deltas between the transmitter and receiver in the individual analogue components, or even movement of one, or both of the devices can cause a non-negligible carrier offset. Previous work on CFO estimation using MTL has shown good results when comparing coarse and fine estimators as different trainable regression tasks for an OFDM system, where the individual losses were summed up without any loss weightings [47]. Very recent work on CFO estimation in an MTL configuration has been conducted by combining a Channel Estimation (CE) task with a CFO estimation task [46]. The ideal loss weightings for the CE/CFO tasks were found to be 0.9/0.1, biasing the learning strongly towards the CE task, which was shown to be harder to learn.

None of the cited works discussed improving frame synchronization with MTL, specifically with CFO as a secondary task. Additionally, to the best of the author's knowledge, a combined frame synchronization and SNR estimation model trained in an MTL scheme has not been proposed in past literature. The work described in this chapter aims to expand the collective knowledge of training fully convolutional architectures with MTL.

6.3 Automatic Modulation Classification with SNR Estimation

AMC models commonly employ convolutional layers as the first feature extraction layers of the network, promptly followed with a flatten and subsequent FC layers, ending with a softmax. Feature extractor networks are common in a variety of disciplines, and

often learn the same first layers, regardless of the task (e.g. in computer vision it is very common to see the first CNN layers learn edge detection, which is a useful feature for most vision applications [127]). This fact can be exploited by sharing the resources of the feature extractors and re-using them for multiple tasks under the same model.

Since training is often performed on datasets with varying SNR levels, the labels for SNR estimation are already available. Many higher order statistical features are affected by noise, which is why SNR estimation is often employed before AMC, with each AMC model being trained for a particular SNR [41] [42]. MTL has been used for AMC and SNR estimation in an effort to improve performance in [43], [44], however the SNR estimation head predictions were fairly coarse, containing 2-3 SNR level classes.

The MTL-assisted AMC approach in this section treats the SNR estimation as a fine-grained regression task, that can estimate the SNR more precisely, alongside the AMC task. The network is summarized in Figure 6.3. The AMC DNN architecture is taken from [128], with the only modification being the attachment of the SNR estimation head.

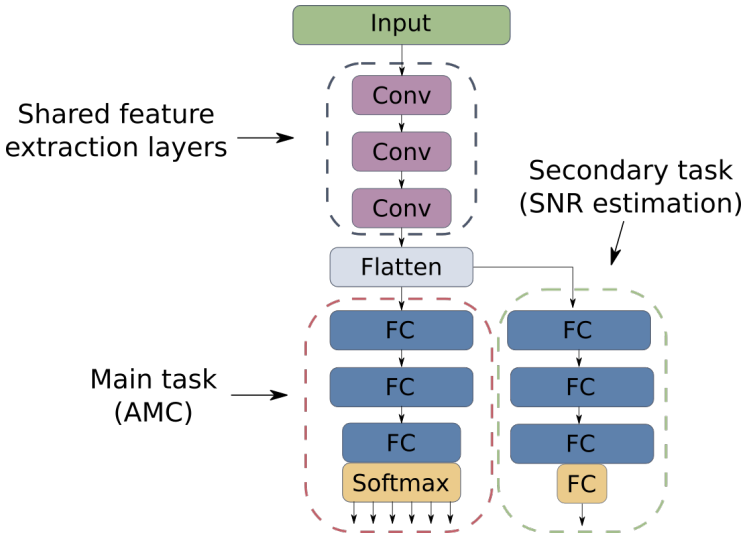


Figure 6.3: MTL Architecture with AMC and SNR estimation heads

A multitask learning model will have some shared layers near the input, and these usually work as feature extractors which can be useful for many tasks. Training neural networks for AMC, for example, we can use this method to make them more robust to

noise by adding a noise estimation task.

6.3.1 Dataset

In this section, an example AMC application is examined with 5 modulation schemes based on phase and amplitude modulation: BPSK, QPSK, 8-PSK, 16-QAM and 4-ASK. One training example contains 1024 samples of pulse shaped and baseband-modulated symbols. To demonstrate the efficacy of MTL, in this case the only channel impairment considered is AWGN at SNR levels between 0 and 20dB. Examples of each modulation scheme are presented in Figure 6.4.

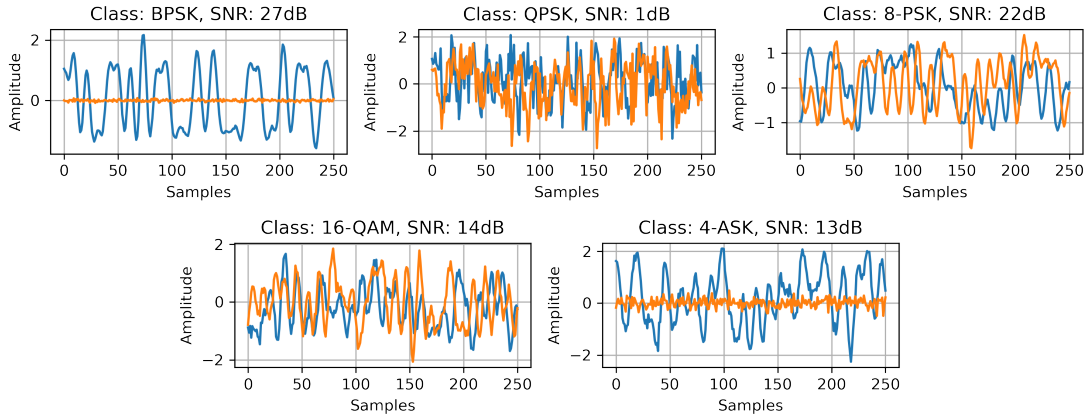


Figure 6.4: Overview of modulation classes in the time domain

6.3.2 Architecture

The base DNN in this chapter is a VGG-type [129] model modified for AMC in [128] by changing the filter sizes to be compatible with a 1024×2 input shape, summarized in Table 6.1. It consists of 7 Convolutional layers along with a Maxpool operation after each convolution. The classification head consists of 3 FC layers with the output of the last layer being fed into a Softmax activation function.

For the MTL implementation, the main DNN (Table 6.1) is modified by attaching a new head, composed of a set of new layers, strictly for SNR estimation, as illustrated in Figure 6.3. The goal of this sub-network is to perform a regression estimation of the noise present in the input waveform. The SNR estimator consists of four FC layers,

Table 6.1: AMC-VGGNet Parameters

Layer	**Parameters**	**Output Shape**
Input	-	1024×2
Conv1D + ReLU	3×1, 64 filters	1024×2
Maxpool1D	2	512×64
Conv1D + ReLU	3×1, 64 filters	512×64
Maxpool1D	2	256×64
Conv1D + ReLU	3×1, 64 filters	256×64
Maxpool1D	2	128×64
Conv1D + ReLU	3×1, 64 filters	128×64
Maxpool1D	2	64×64
Conv1D + ReLU	3×1, 64 filters	64×64
Maxpool1D	2	32×64
Conv1D + ReLU	3×1, 64 filters	32×64
Maxpool1D	2	16×64
Conv1D + ReLU	3×1, 64 filters	16×64
Maxpool1D	2	8×64
Flatten	–	512×1
FC + SeLU	128	128×1
FC + SeLU	128	128×1
FC + Softmax	5	5×1

the first three followed up with ReLU activation functions with the last one remaining linear, which allows representation of both negative and positive values. The parameters of the SNR estimation head are summarized in Table 6.2.

Table 6.2: Noise Estimator Head Parameters

Layer	**Parameters**	**Output Shape**
Input (from flatten)	–	512×1
FC + ReLU	512	512×1
FC + ReLU	128	128×1
FC + ReLU	64	64×1
FC	1	1×1

6.3.3 SNR estimation

Noise is fundamental and appears in all fields of signal processing. Some examples are grainy pictures, recordings with background noise, and perturbed communications signals due to channel effects. In ML, mislabelled data is often referred to as noise within

a training dataset [130]. Due to noise being such a pervasive phenomena, a plethora of techniques have been developed to estimate SNR and enable many signal processing applications to work correctly, for example, practical (Minimum Mean Squared Error) MMSE receivers require a noise variance estimate for channel equalization [131]. Noise estimation is also utilized in error correcting alogirhtms to set soft decision probabilities [132].

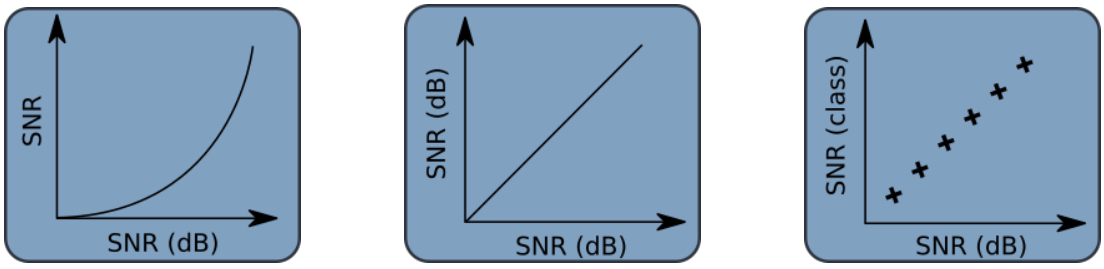
SNR estimation is broadly divided into two main categories: data-aided and non-data-aided [133], [134]. Recent DL approaches to SNR estimation have focused on tackling the problem using a non-data-aided methodology [50], [48], [49].

There are multiple ways to approach SNR estimation with a DNN, purely in terms of data formatting. For instance, the labels can simply be the linear SNR value to be predicted as a regression [48], [50], [51], or the dB values can be estimated directly in the logarithmic domain by placing them into buckets/classes and then performing a classification [43], [49], [40]. To gain some intuition on how best to represent the SNR labels, some of the formatting methods have been evaluated using a smaller version of the DNN architecture described in Table 6.3, which is simply a minimized version of the architecture in Table 6.1. Regression methods seem most natural, allowing better scaling and inference precision, however placing each SNR level into its own bucket and performing classification with categorical crossentropy for learning will also be explored.

For this experiment a total of three variations of SNR labelling will be examined: two regressions and one variation of classification. The architecture and optimization hyperparameters have not been thoroughly investigated here, as the primary objective from this exercise is to gain some insight of the best way of formatting the training data for the SNR estimation task, as opposed to creating an optimal SNR estimator. The conclusions from this experiment will help develop an SNR estimation head for an MTL-aided AMC model.

The three different SNR representations that are explored in this section as illustrated in Figure 6.5, and are summarized as follows:

1. **Linear SNR**. This value is calculated simply by dividing the signal power by the noise power – the definition of SNR. Intuitively, it could make sense to teach



(a) Linear SNR (b) SNR in dBs (c) Each SNR as a class

Figure 6.5: Different SNR representations used to train an estimator

the network a linear relationship to noise, as opposed to having it do additional internal transformations to represent the amount of noise on a log scale.

2. **SNR in dBs**, arguably the simplest approach – passthrough of SNR dB integer values, i.e. -4, -2, 0, 2, etc. The task will be treated as a regression and the DNN will have to output the estimated SNR in decibels. This is simple to implement because most DSP software packages prompt the user to specify SNR on a logarithmic scale, or display it by default, requiring little to no preprocessing in the training loop implementation.

3. **Each SNR as a class**, i.e. each dB value is predicted by a different neuron – a classification approach with categorical crossentropy as the loss function. In this case the SNR estimator will have an output of 16 neurons, each predicting an SNR value in the range from -15 to 15dB, with a step size of 2dB (a total of 16 classes). The disadvantage of this approach is that the classes are quite rigid, and do not allow fractions (e.g. 3.5dB would fall into the closest bucket of 3dB), unlike the regression-focused formats. Of course the range and precision can be improved by increasing the number of classes, which adds complexity to the architecture.

Training of a Minimal Estimator

The SNR estimator network is smaller version of the AMC DNN, and is summarized in Table 6.3. The core of the network is the same for all cases, with the exception of the final layer, which needs to be modified based on the label format and loss function. In

the logarithmic domain, where SNR is represented in dBs, the output is a single linear neuron with no activations, free to output negative and positive values. For the linear SNR representation, the output is a single neuron with a ReLU activation – the ReLU activation prevents it from outputting negative values, as linear SNR is only a ratio. For the classification approach, the only change is that the output must be 16 neurons for each SNR bucket, and of course the loss function in this case will be categorical cross-entropy as opposed to MSE in the first two cases.

Table 6.3: SNR estimator parameters

Layer	**Parameters**	**Output Shape**
Input	-	1024×2
Conv1D + ReLU	3×1, 64 filters	1024×2
Maxpool1D	2	512×64
Conv1D + ReLU	3×1, 64 filters	512×64
Maxpool1D	2	256×64
Conv1D + ReLU	3×1, 64 filters	256×64
Maxpool1D	2	128×64
Flatten	–	8192×1
FC + ReLU	512	512×1
FC + ReLU	64	64×1
FC (SNR in dBs)	1	1×1
FC + ReLU (Linear SNR)	1	1×1
FC (SNR as classes)	16	16×1

The network was trained on an AMC dataset with the modulation labels replaced with SNR labels instead. This includes 40k example waveforms of SNRs from -15dB to 15dB. The single neuron versions were trained using MSE and the 16-neuron revision of the network was evaluated using cross-entropy. Each method was used to train three models to ensure that the results were consistent, and, barring the final layer, the random seeds for weight initialization were identical when switching methods. The training parameters are summarized in Table 6.4.

Training and validation losses were averaged for each of the three models and are shown in Figure 6.6. It is clear that estimating linear SNR, with no normalization, resulted in the poorest fit to the training set, with the initial MSE loss being extremely

Table 6.4: Noise estimator training details

Parameter	**Value**
Optimizer	ADAM
Loss function	MSE & Cross entropy
Number of training examples	40k
Batch size	32
Number of epochs	30
Learning rate (α)	0.0005
Regularization Factor (λ)	0.001
Modulation schemes	BPSK, QPSK, 4-ASK, 8-PSK, 16-QAM
Training SNRs (dB)	-15dB to 15dB (in steps of 2dB)

large. Estimating the SNR dB values directly achieved a better fit, however looking at the validation loss, it is clearly overfitting. The most consistent results, with the training and validation cross-entropy losses nearly coinciding, came from the third method of treating the SNR levels as classes.

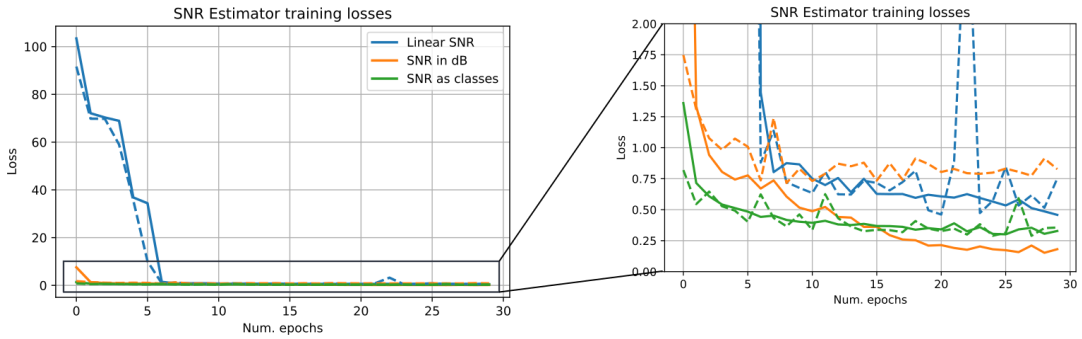


Figure 6.6: SNR estimator training losses (dashed lines are validation losses)

Inference Results

For each method the trained models were individually evaluated and their average results plotted in Figure 6.7 and Figure 6.8. Each model was evaluated by processing 64 frames of 1024 I/Q samples per modulation ($64 \times 5 = 320$), for each SNR level. Linear predictions are reverted to dB with $SNR_{dB} = 10 \times \log \hat{z}$, where $\hat{z}$ is the predicted linear SNR. The dB predictions are plotted as is, with no additional processing. In the case of classification the dB values are retrieved with a dictionary/lookup table.

At very low SNR all methods were suboptimal, however, all estimators perform

relatively well at SNRs above -10dB.

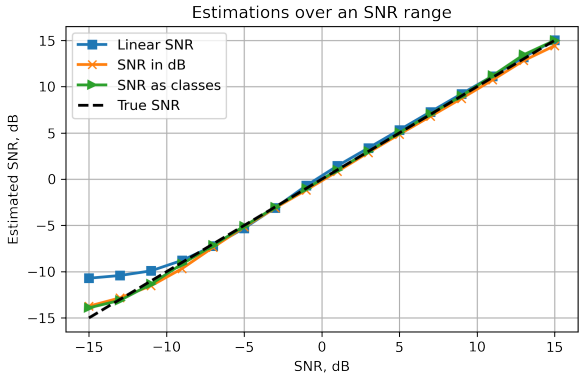


Figure 6.7: SNR estimation results

Each trained model was further evaluated by sweeping through the SNRs and calculating individual SNR dB prediction closeness to the expected value using MSE, shown in Figure 6.8. As expected the linear SNR estimation does very poorly at low SNRs. By comparison the dB SNR estimator performs best at extremely low SNRs. Interestingly, the classification approach resulted in near perfect estimation results in the range between -5dB and 10dB, much like the other methods it is poorest at lower SNRs (below -10dB).

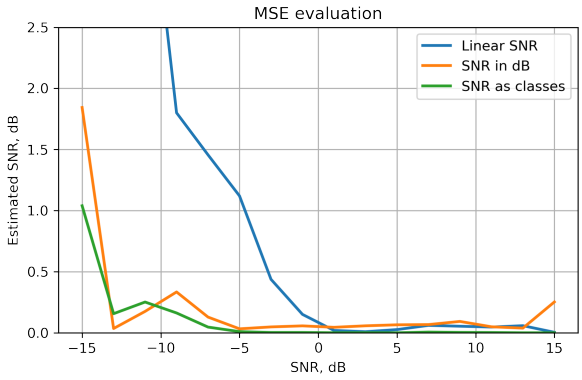


Figure 6.8: SNR estimation results, closer look at MSE

Observations

A quick conclusion to draw is that, even with poorly formatted training data (e.g. unnormalized linear SNR), a DL approach can still provide a "good enough" estimator that will work well for a large range of SNRs, even when it is not the optimum estimator training strategy. Estimating SNRs directly as dB values proved to be easiest to implement (no pre/post processing) and produced a competetive model.

Classification was the most consistent, and resulted in the lowest loss, which, aside from performance, is an important consideration for MTL – it is undesirable to have very large losses, which could overshadow other tasks and prevent learning.

6.3.4 AMC and SNR Estimation Loss Tradeoff

It is imperative that the network learns both tasks adequately – if no learning occurs on the secondary task there is no point in MTL because the shared layers will not be influenced by the additional labels. With this, however, comes the need to balance the priority of the individual learning tasks. Since the tasks have individual loss functions which are added before backpropagation, it would be undesirable for one loss to send a significantly stronger signal than the others, which could result in learning being halted on other tasks.

Necessarily, additional hyperparameters need to be introduced to resolve this issue, namely the individual loss function weightings. The weighting for secondary tasks can be as high as 0.8 in some CV tasks [124] or as low as 0.006 in speech signals [40]. For the AMC and SNR estimation tasks, the new parameters w_{AMC} and w_{SNR} are introduced, respectively, which are the weightings for the two loss functions $\mathcal{L}_{AMC}$ and $\mathcal{L}_{SNR}$ in order to calculate the overall loss $\mathcal{L}$ for the DNN as defined in Eq. 6.1.

$$\mathcal{L} = w_{AMC}\mathcal{L}_{AMC} + w_{SNR}\mathcal{L}_{SNR}. \tag{6.1}$$

As shown in [124], the primary task loss weighting can be set to 1, and the secondary weighting is determined empirically by continuously training and evaluating the same model using different secondary task loss weightings. Setting $w_{AMC} = 1$, Eq. 6.1 is

simplified to:

$$\mathcal{L} = \mathcal{L}_{AMC} + w_{SNR}\mathcal{L}_{SNR}. \tag{6.2}$$

where now only w_{SNR} needs to be determined. A hyperparameter exploration on this weight parameter is conducted by training and evaluating the same DNN while progressively reducing w_{SNR} from 0.9 to 0.1. First of all, five baseline models are trained (Table 6.1) with set weight initializations. Then, for each w_{SNR}, five more models are trained using exactly the same weight initializations for the shared layers as the baseline models, but with the additional SNR estimation head.

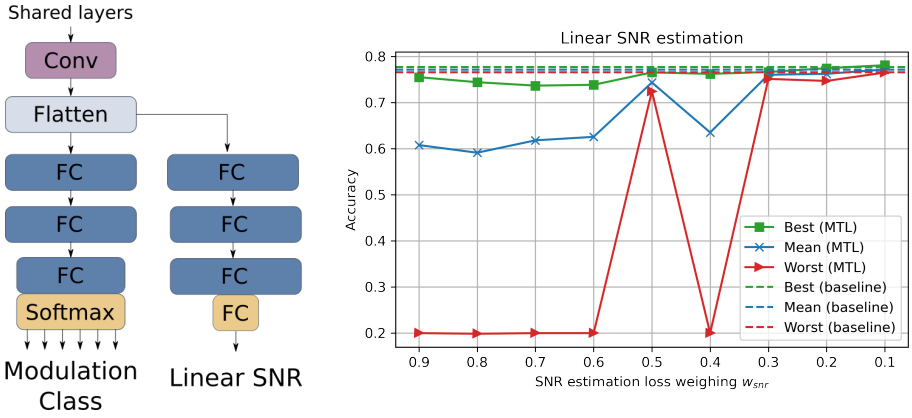
Training Results

In total, to evaluate each SNR estimator head, 45 MTL models were trained (5 models $\times$ 9 w_{SNR} values). The five baseline models were the same for all subfigures, and each and every MTL model was trained on an equivalent dataset as the baseline models. The details of training the DNNs are summarized in Table 6.5. These are similar to Table 6.4 for the noise estimator models, but with a lower regularization factor λ to make it easier for models to converge to a solution.

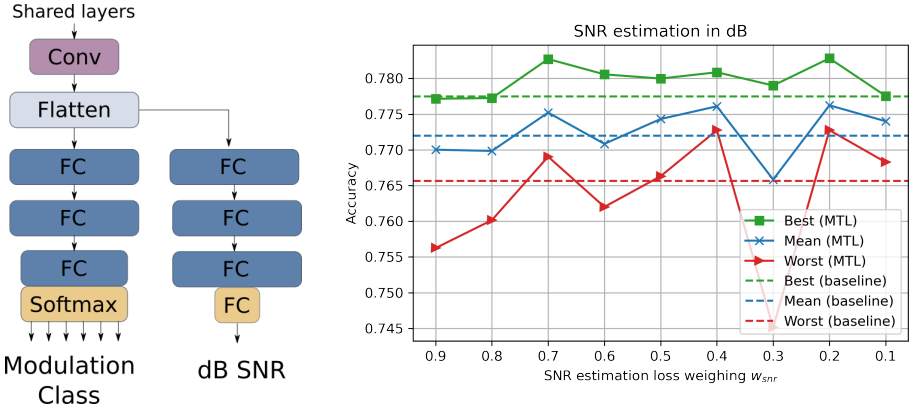
Table 6.5: AMC-SNR MTL model training details

Parameter	**Value**
Number models	5
Optimizer	ADAM
Loss function	MSE & Cross entropy
Batch size	32
Number epochs	30
Learning rate (α)	0.0005
Regularization Factor (λ)	0.0001
Number training examples	40k
Number validation examples	5k
Modulation schemes	BPSK, QPSK, 4-ASK, 8-PSK, 16-QAM
Training SNRs (dB)	-15dB to 15dB (in steps of 2)

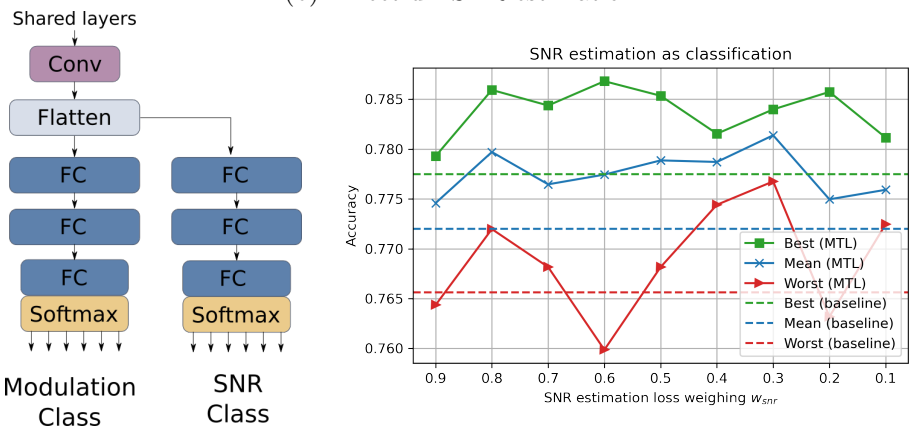
The results for every MTL AMC model for each SNR estimation head are shown in Figure 6.9. The mean accuracy of each trained model was evaluated in the SNR range



(a) Linear SNR estimation.



(b) Direct dB SNR estimation.



(c) SNR as classification.

Figure 6.9: SNR estimator configurations and loss weighting results

of -15dB to 15dB, for the modulation schemes listed in Table 6.5. To obtain greater insight into the performance and variance of these MTL models, the best, worst and mean accuracies of the five models per w_{SNR} are plotted together.

Figure 6.9 (a), due to high initial loss the model often fails to converge. This is mitigated as the SNR loss weighting trends down, but never outperforms the best baseline models (this might explain why the authors of [40] had an SNR loss weighting of 0.006 with their linear SNR estimator head). As shown in Figure 6.9 (b), at a loss weighting of 0.4, a consistent improvement is demonstrated, where even the worst model performs better than baseline. As shown in Figure 6.9 (c), the best model consistently beats the baseline, likely due to the classification-based estimator head outperforming other estimators, and cross-entropy being the most similar to the AMC training loss.

Observations

Surprisingly there does not seem to be a trend in performance as the w_{SNR} parameter is reduced, emphasizing AMC loss and de-emphasizing SNR estimation loss. The hypothesis was that, as the SNR loss weighting is decreased, higher priority is given to the AMC task, with enough SNR awareness to boost performance at lower SNRs. Looking at the most successful experiment in Figure 6.9 (c), the performance seems to have been boosted across the board at all SNR weightings, although the best possible model was achieved at $w_{SNR} = 0.6$.

6.3.5 Results on AMC

Instead of looking at the averages, the accuracy over the tested SNR range for the configuration that produced the best possible model (classification-based SNR estimator with $w_{SNR} = 0.6$) is shown in Figure 6.10. Interestingly, making the model SNR-aware did not only improve performance in the lower SNR bracket, but shifted the whole accuracy curve to the left across all SNRs. This suggests that using MTL acts as an additional source of regularization.

The average performance of the classifier, by using MTL, improved by a little over 0.9%, and the best MTL model achieved an overall accuracy improvement of 1.4%. While these gains do not appear to be groundbreaking, as shown in Figure 6.9 (c), they are consistent. These results were achieved by keeping the training hyperparameters, dataset, and weight initializations identical to the baseline non-MTL models, with the

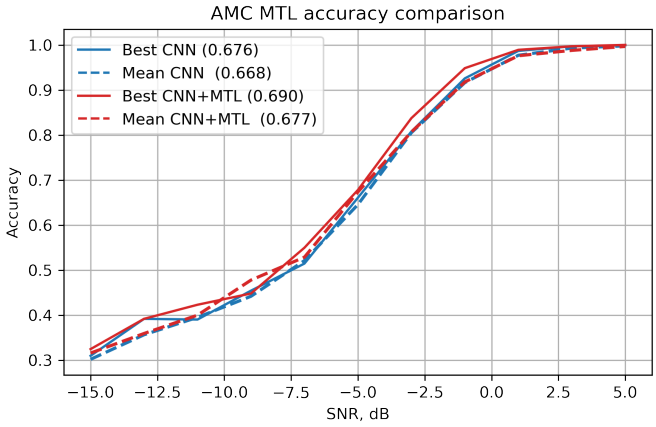


Figure 6.10: AMC mean accuracy results post MTL training. The MTL (red) curves showing better performance across entire SNR range.

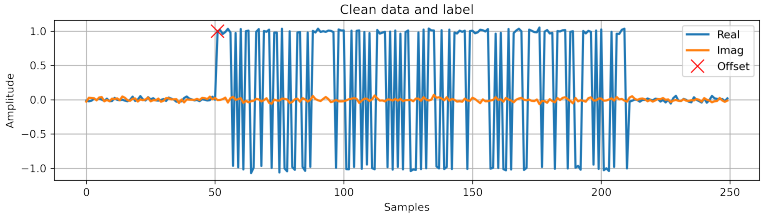
only change during training being the addition of an SNR estimation head, which is not used during AMC inference. Post-training, the SNR estimation head can be removed, and the remaining AMC network will maintain the performance boost without costing any additional resources at inference time.

6.4 Frame Synchronization with CFO Estimation

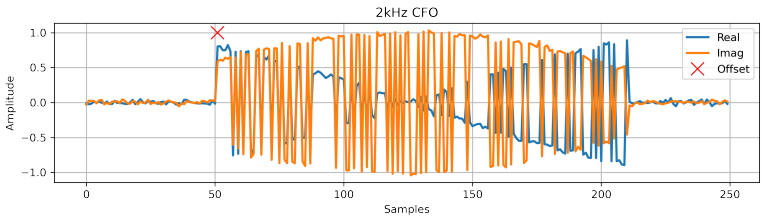
Chapter 5 presented how DL and specifically the FCN architecture can be used for frame synchronization, to improve the performance of the FS task. The DL methods were especially effective in scenarios where CFO was prevalent. Even though the FCNs have shown improved performance under such impairments, the capabilities of this receiver can be improved even further by implementing MTL in the training loop.

6.4.1 Dataset

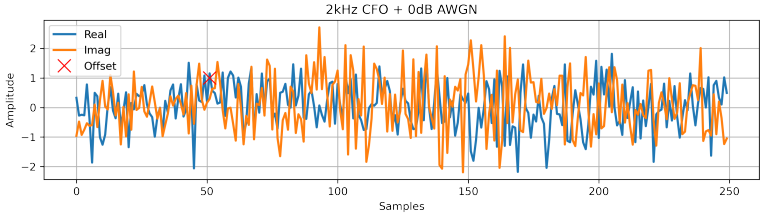
The parameters of the dataset for the FCN MTL models explored here are similar to Section 5.3.1, key differences being that the CFO parameter is saved as an additional training label, and only a single preamble length of 32-bits is explored. Additionally, the DNNs are now trained on a wider range of CFOs – for the simple reason that the FCN presented in the earlier chapter is capable of performing better at even larger CFOs – and MTL is used here to push that boundary a bit further.



(a) Clean data frame and label



(b) Added 2kHz CFO



(c) Added 2kHz CFO + 0dB AWGN

Figure 6.11: Illustration of FS+CFO dataset example generation

To gain a more intuitive understanding of what the dataset looks like, the generation steps for a single example are summarized in Figure 6.11. In (a) the 32-bit preamble and payload are combined into a packet and inserted at a random offset of an empty tensor (all values of the array set to 0). Subplot (b) shows the effects of perturbing this frame with a CFO offset of 2kHz (multiplying by a 2kHz sinusoid). Finally, (c) shows this waveform being passed through an AWGN channel of 0dB (the training SNR). At this point it is very difficult for the human eye to make out the structure of the initial waveform from (a), but luckily we have neural networks to do this for us.

6.4.2 Architecture

The architecture of the base FS DNN is exactly the same as described previously in Table 5.1. The key difference is an additional CFO estimation head attached to the output of the second convolutional layer of the original FCN model, as shown in Figure

6.12. The final output of the CFO estimation head is a single scalar value representing the normalized frequency offset.

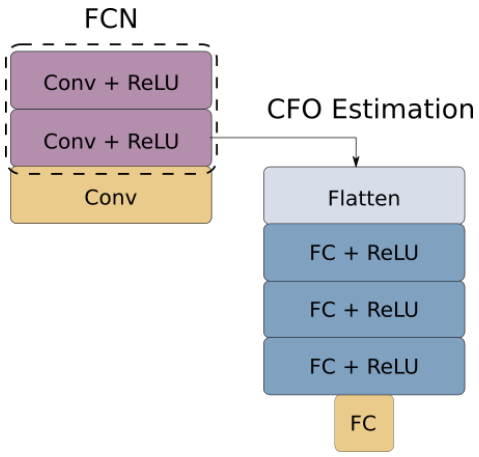


Figure 6.12: MTL Architecture with FS and CFO estimation heads

The CFO estimation head parameters are summarized in Table 6.6, similar to the single SNR estimation heads. The second shared convolutional layer output is flattened so that it can be used by the FC layers of the CFO estimator. Each FC layer is followed by a ReLU activation function, barring the final layer which is linear.

Table 6.6: CFO Estimator Parameters

Layer	**Parameters**	**Output Shape**
Input (from flatten)	–	512 × 1
FC + ReLU	512	512 × 1
FC + ReLU	128	128 × 1
FC + ReLU	64	64 × 1
FC	1	1 × 1

6.4.3 CFO Estimation

In realistic systems some level of CFO is unavoidable, and tasks like FS are very sensitive to CFO. Conventionally once a preamble is detected, the known sequence containing pilot symbols can be used to estimate and correct for CFO. After estimation and correction, the payload containing data bits can be demodulated and decoded. For this to happen, it is essential that a receiver is capable of functioning at various frequency offsets (how tolerant it has to be will depend on the wireless transmission standard).

As with other estimation tasks, DL has been successfully applied in the past for CFO estimation. Both CNNs and RNNs have been explored in [33], where the CFO estimation task was treated as a single output regression and showed very good results at frequency offsets up to 15kHz. Similarly, simple feedforward DNNs, CNNs and RNNs have been explored in [135], where the output was treated as a single value regression. Very good results using an RNN were also achieved in [136], again, using a single output regression and scalar CFO values as labels.

Training CFO Estimators

Existing literature on DL-based CFO estimation suggests that there is not as high a variance of possible labeling variations as there are with SNR estimation. With that in mind, in this section various CFO label formats will not be explored, but a single architecture for an estimator head will be chosen and trained as a sanity check – the estimator does not need to be ideal, just "good enough" to assist the main FS task by providing CFO-awareness.

Table 6.7: CFO estimator training details

Parameter	**Value**
Optimizer	ADAM
Loss function	MSE CFO
Number of training examples	8192
Batch size	32
Number of epochs	30
Learning rate (α)	0.001
Regularization Factor (λ)	0.001
Training SNR	0dB, 5dB, 10dB
Maximum CFO	35kHz

The 32-bit preamble-detecting FCNs in the previous chapter were trained on simulated captures of 0dB SNR, which is roughly where the detector reaches the acceptable DER of 1e-3. Ideally the CFO estimator should work well enough at this SNR, but to gain more insight into the model's ability to cope with different SNR levels, five models will be trained at SNR levels of 0, 5 and 10dB. All of the other parameters are still the same as in Table 6.6 for the purposes of integrating this CFO detector head into the

previously defined FCN model. The averaged training results at each training SNR are summarized in Figure 6.13.

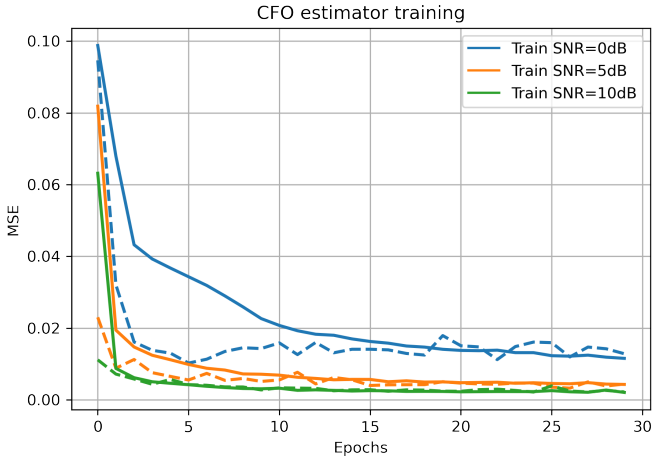


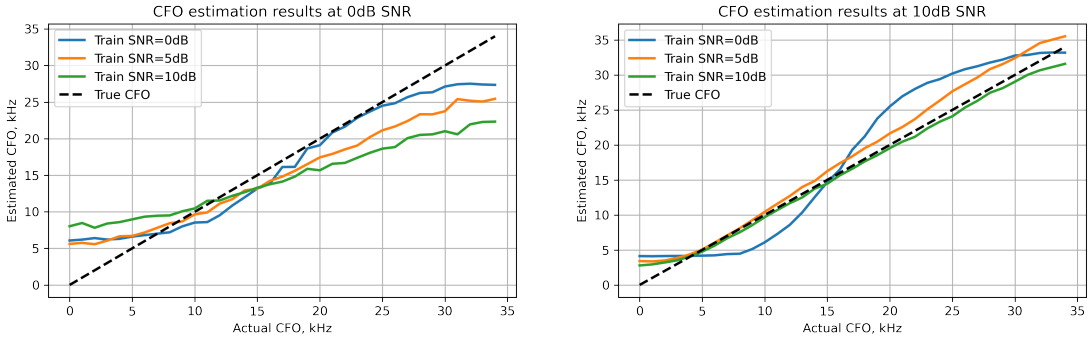
Figure 6.13: CFO estimator training losses (dashed lines are validation losses)

The training results do not stray too far away from expectations – the higher the SNR the easier it is to fit a model, resulting in lower MSE loss (with the 10dB dataset being easiest to fit). Of note is the fact that the 0dB iterations show overfitting, with the validation loss quickly running away past the 5th epoch (meaning that with early stopping implemented, this will be the quickest to train). The reason this is happening is possibly due to the low SNR dataset being too difficult; the DNN might memorize some noisy examples, but will not be able to generalize to unseen high CFO instances. Looking at the 5 and 10dB curves, the higher SNR datasets result in closer training and validation losses, indicating less over or underfitting.

Inference Results

Training and validation losses showing downward trends are a great indication that learning is happening, and that there are no severe failures in the training configuration. In order to better gauge performance, each of the trained estimators are evaluated at a range of CFOs at low and high SNRs, as shown in Figure 6.14.

None of the estimators work perfectly at low SNR, as indicated in Figure 6.14 (a), with the biggest drop-offs past 25kHz, and surprisingly, at the lower CFOs (up



(a) Estimator results at low SNR (0dB) (b) Estimator results at high SNR (10dB)

Figure 6.14: CFO estimator results

to 10kHz). Evaluation at a high SNR shows near perfect performance by the estimator that was trained at that 10dB SNR, with the other higher SNR (5dB) trained estimator still doing fairly well, however the 0dB trained estimator showing an intriguing, almost sinusoidal pattern of underestimating lower CFOs and overshooting on the higher CFOs. The undershoting and overshooting of the 0dB trained model could be explained by the fact that training at low SNR the CFO the model learns a rough "low" or "high" estimation, since there is too much noise for predicting at a finer scale.

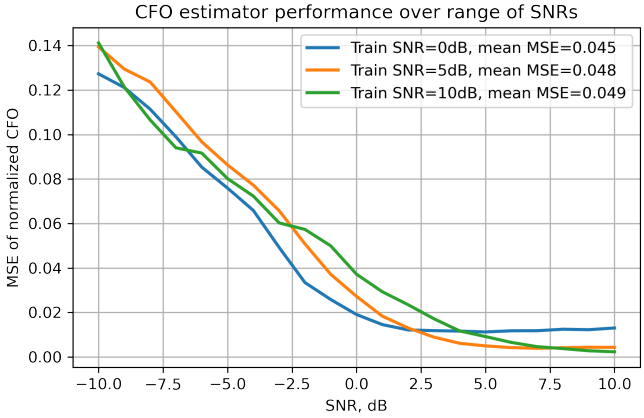


Figure 6.15: CFO estimator MSE evaluation over an SNR range

To gauge how sensitive each model is to SNR, an additional series of tests are carried out where each model is evaluated on 256 frames, each representing a random CFO from 0 to 35kHz, for an SNR sweep in the range of -10dB to 10dB. The results are

summarized in Figure 6.15.

While the models were trained on higher SNRs (5dB and 10dB), the best overall MSE was achieved the by the DNN trained on 0dB data. Fortunately this is the same SNR as the original FCNs were trained on in Chapter 5 for 32-bit preamble lengths, which empirically proved to be the best SNR for training FCN preamble detectors for that preamble length. Poor CFO estimation performance at high SNR is not as much of an issue since 32-bit preambles at high SNR have very low detection error rates.

Observations

Clearly the trained CFO estimators are not ideal: previously referenced work [33], has shown much better results using RNNs. The fact that all estimators show poor performance at the very low CFO values (Figure 6.14) is unexpected and might warrant some future investigation. There is certainly scope for tuning hyperparameters and the dataset. The goal is to produce a "good enough" CFO estimator to attach as an additional head on an FCN to make it CFO-aware, and as such this estimator should be sufficient to prove out the concept.

6.4.4 Frame Synchronization and CFO Estimation Loss Tradeoff

Similar to the AMC and SNR losses in Section 6.3.4, loss weightings are introduced for FS loss ($\mathcal{L}_{FS}$) and CFO estimation loss ($\mathcal{L}_{CFO}$) as w_{FS} and w_{CFO}, respectively. This relationship is captured in Eq. 6.3.

$$\mathcal{L} = w_{FS}\mathcal{L}_{FS} + w_{CFO}\mathcal{L}_{CFO}. \tag{6.3}$$

Generally w_{FS} and w_{CFO} are constrained to $w_{FS}, w_{CFO} \in \{0, 1\}$. The experiment of obtaining the most effective loss weighting w_{CFO} is done here in exactly the same fashion as for w_{SNR} in the Section 6.3.4. The FS loss weighting is kept at a constant value of $w_{FS} = 1$, while sets of five MTL models are trained for each w_{CFO} in steps of 0.1. The same dataset, summarized in Table 6.8, is used for every MTL and non-MTL model, and is not re-shuffled for new models, i.e. each model is fed the training data in the same order. Since five models are trained for each weightings configuration, five

unique random seeds are used for weight initialization, meaning that the only differences between models will be the actual loss values influenced by the CFO estimator head (or lack thereof for the baseline models).

Table 6.8: FS-CFO MTL training details

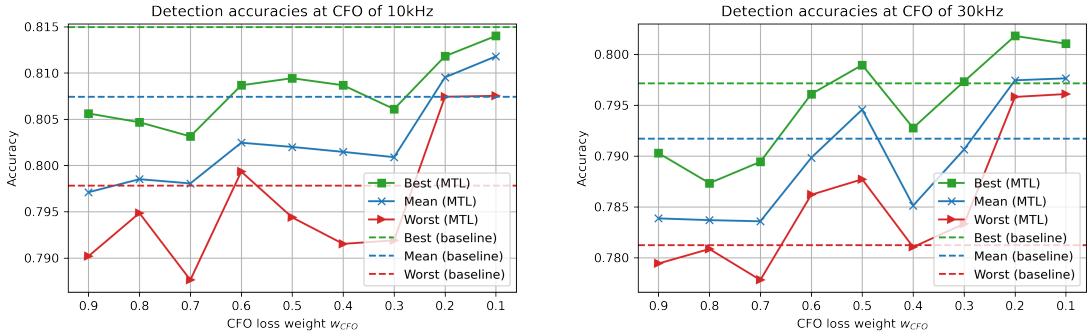
Parameter	**Value**
Optimizer	ADAM
Loss function	Cross-entropy (FS) & MSE (CFO)
Number of training examples	8192
Batch size	32
Number of epochs	30
Learning rate (α)	0.001
Regularization Factor (λ)	0.001
Training SNR	0dB
Maximum CFO	35kHz
Preamble length	32
Payload size	128
Capture size	200
Modulation	BPSK

Training Results

For each weightings configuration, every model is evaluated by finding the average detection accuracy rate (the reverse of DER) over an SNR range from -10dB to 10dB. The best, mean and worst overall accuracies are captured in Figure 6.16. This is done at two fixed CFOs of 10kHz and 30kHz being used as impairments during these experiments – the models trained at w_{CFO} will be evaluated across the whole SNR range, but only on the single CFO value. Two distinct CFOs are used to make sure the models have not overfit to a single CFO range. The dashed baseline guidelines in Figure 6.16 are based on five FCN models, without MTL, trained on the same dataset, and using the same weight initializations as the MTL models.

Unlike the MTL with SNR results for the AMC task, there is a clear upward trend as w_{CFO} decreases, for both fixed CFO cases. However, for a 10kHz CFO there is no w_{CFO} where even one MTL model outperforms the baselines.

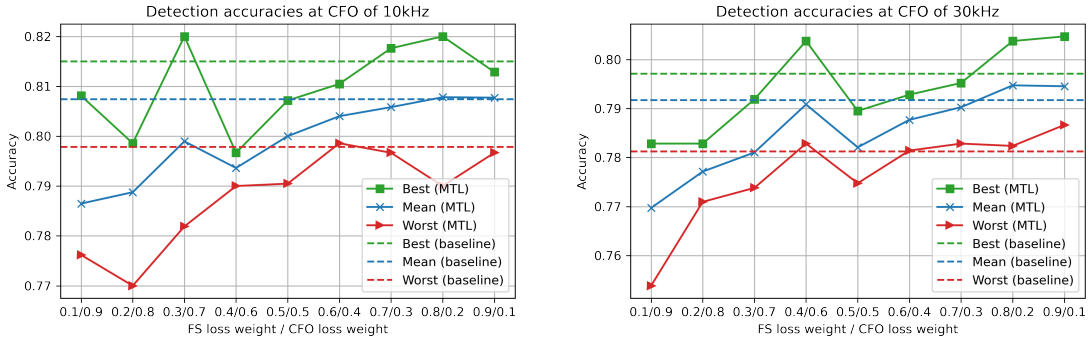
Expanding the search space by sweeping through both w_{FS} and w_{CFO}, a further experiment was run, where another set of 45 MTL models were trained, this time with



(a) Detection accuracies at CFO of 10kHz (b) Detection accuracies at CFO of 30kHz

Figure 6.16: FS-CFO MTL detection accuracy results at decreasing CFO loss weighting w_{CFO}

w_{FS} being increased from $0.1 \rightarrow 0.9$, and w_{CFO} reduced from $0.9 \rightarrow 0.1$. With the same training parameters and evaluation criteria, the new results are summarized in Figure 6.17.



(a) Detection accuracies at CFO of 10kHz (b) Detection accuracies at CFO of 30kHz

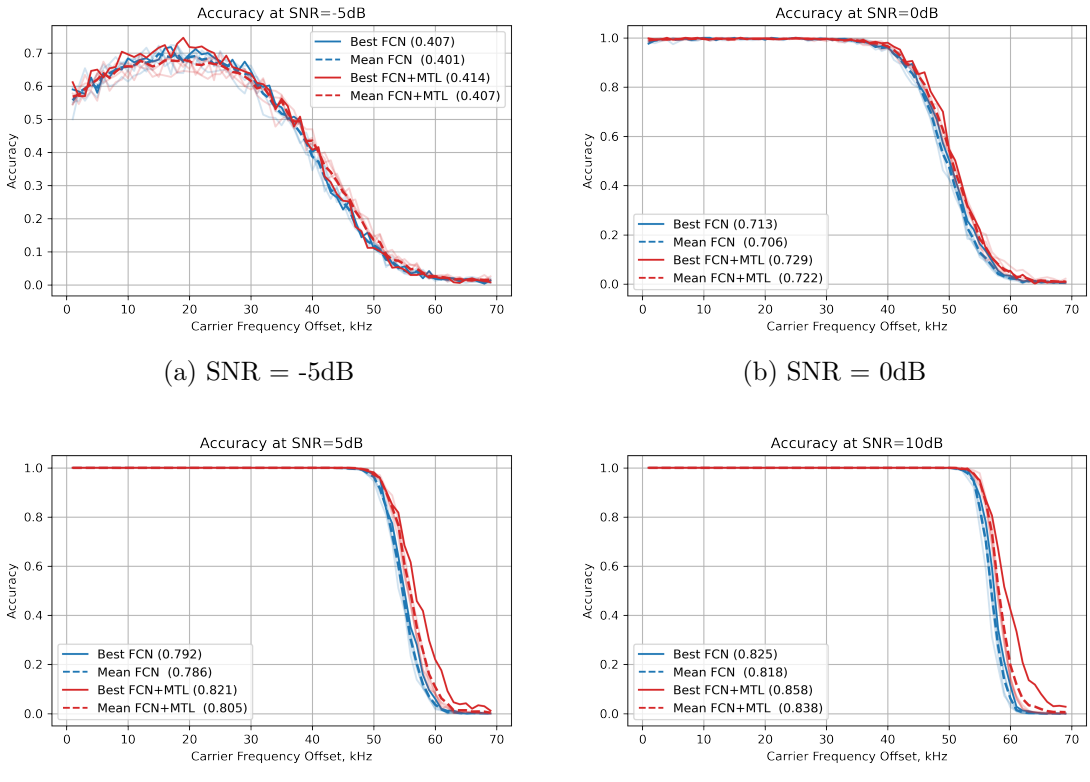
Figure 6.17: FS-CFO MTL detection accuracy results with a parameter sweep of both w_{FS} and w_{CFO} loss weightings

It is quite clear from the new results, that as the main task weighting w_{FS} is increased, the overall results generally trend upwards (since FS accuracy is the main test metric). Interestingly, even at a very low weighting of $w_{FS} = 0.1$, none of MTL FCN models failed to converge completely and still produced better-than-guessing results. The most consistently good configuration proved to be $w_{FS}/w_{CFO} = 0.8/0.2$ (interestingly, very similar to the reported split of 0.9/0.1 in [46]) – these parameters will be used for further analysis over a wider CFO sweep.

6.4.5 Frame Synchronization Results Over Varying CFOs

In order to evaluate how well the trained models are able to generalize to unseen CFOs at different SNR levels, the next experiments consist of evaluating the models over a large CFO sweep at a few fixed SNR values. The CFOs tested range from 0 to 70kHz in steps of 1kHz. The SNRs of interest here are -5, 0, 5 and 10 dB. The same baseline models as earlier are used for these tests, denoted in the legends of Figure 6.18 as "FCN", whereas the models denoted as "FCN+MTL" are the models trained with loss weightings of $w_{FS}/w_{CFO} = 0.8/0.2$.

As seen in Chapter 5, it is expected that a 32-bit preamble in a bursty communications scenario starts performing well above an SNR of 0dB. The results in Figure 6.18 indicate that these new MTL models exhibit a similar trend with all results over 0dB converging to 100% inference accuracy below 35kHz.



(a) SNR = -5dB

(b) SNR = 0dB

(c) SNR = 5dB

(d) SNR = 10dB

Figure 6.18: Comparison of FCN performance at a range of frequency offsets trained with and without MTL (legend includes mean accuracies for entire CFO sweep)

In most instances, both FCN+MTL mean and best accuracies outperformed the baselines of just using FCN models. At a high SNR of 10dB, the mean performance of all models was 3% better over the tested CFO range. At a lower SNR of 0dB, the gain was around 2%, whereas at the lowest SNR of -5dB, the two methods are comparable.

Similar to the CFO estimator results in Section 6.4.3, at -5dB the detection accuracy is actually worse at the lower offset values, gradually improving up to a CFO of 20kHz, before starting to degrade again. It is worth noting that both baseline and MTL models suffer from this, indicating that it may be more of a dataset problem. This might be an overfitting effect of training on a single SNR value of 0dB, and perhaps tuning the datasets by including more noisy examples could alleviate this issue – these are explorations ripe for future work on DL CFO estimators.

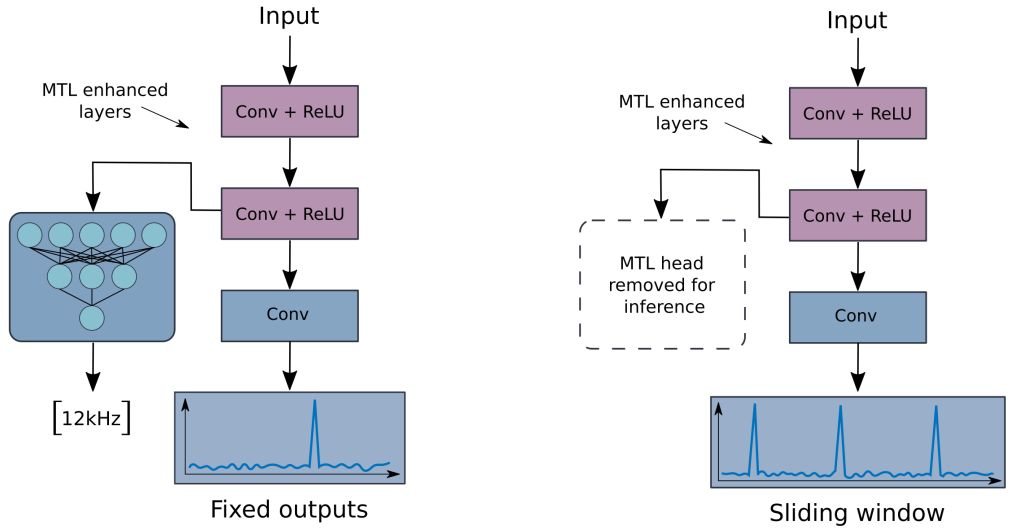
Again, while the MTL results in the past few sections have not been groundbreaking, the gains achieved are consistent and, in environments where CFO is a real issue, this technique can benefit a variety of DL models by adding more CFO-awareness without any additional inference-time cost (assuming that the CFO estimator head is omitted post-training).

6.4.6 Deployment Discussion

The major disandvantage of using MTL and attaching FC-based estimator heads to an FCN architecture is that the wonderful ability of the FCN to adapt to an input of any size is now lost. The question becomes, is CFO estimation important? Perhaps the communication protocol uses a data-aided CFO estimation method, rather than blind estimation with a DNN, in which case, after detecting the packet the pilot symbols can be used to determine the CFO and compensate for it.

If it is important to retain the sliding window property of the FCN is important to keep, then the simplest thing to do is as illustrated in Figure 6.19. Training can occur on fixed-size frames, and then post-training the estimator can be removed, with only the FCN implemented for inference, which now works on inputs of any size again.

A good DL-based CFO estimator for blind estimation may still be useful as a byproduct, which could be implemented by copying the trunk of the MTL DNN, and



(a) During training: guided training forces shared layers to learn features necessary to recognize CFO

(b) Post-training: the additional CFO estimation head is cut off, but the learning outcomes remain

Figure 6.19: MTL training and deployment

deploying the CFO estimator as a standalone model. This is another useful property of MTL – if two separate models are required, one can achieve training efficiencies, by training both of them simultaneously on a single dataset. A caveat to this statement is that the weightings of the individual loss functions should be well understood beforehand, otherwise one risks a massive hyperparameter optimization effort.

6.5 Fully Convolutional MTL

MTL enables a unique quality that can be achieved by using FCNs – continuous classification and monitoring of signal features. Additionally, the combination of MTL with the FCN architecture enables additional interesting applications, such as continuous per-sample SNR estimation. By having the learning task heads completely convolutional, the deployment discussion becomes much simpler because now the entire multi-headed FCN can be deployed for continuous, sliding-window-like inference.

6.5.1 Dataset

To ease the task of SNR estimation, a continuous transmission scheme, rather than bursty communication as seen in the previous FCN experiments, is used for data generation – this ensures that the signal power throughout the simulated frame is uniform. In this configuration data is continuously transmitted and, intermittently, a preamble will appear at a random sample offset within the data symbols. A single training data example with two labels is illustrated in Figure 6.20.

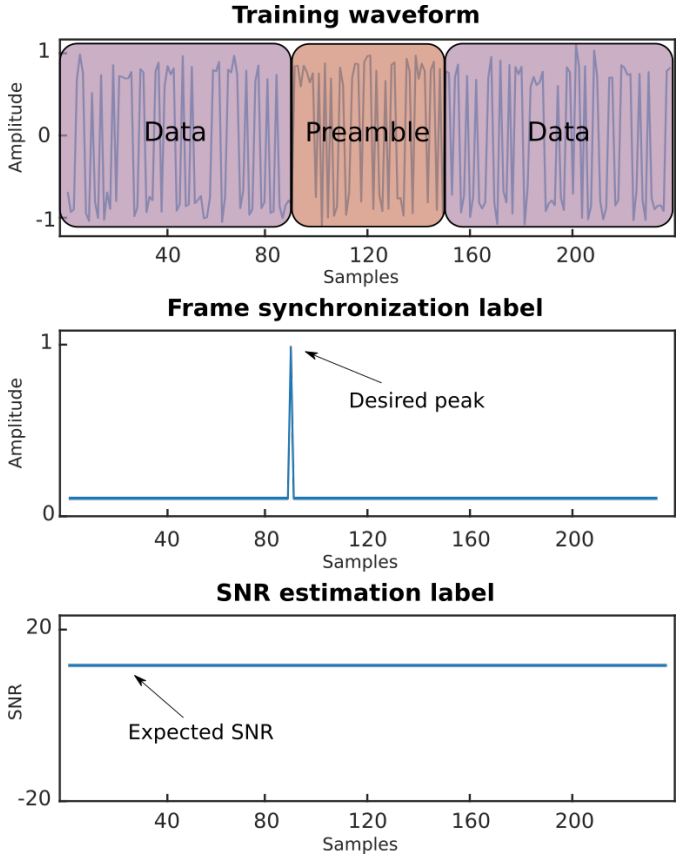


Figure 6.20: Single training waveform in continuous transmission mode

Note the two labels – one for the FS task, and one for SNR estimation. The FS label is the peak corresponding to the ideal receiver response denoting the beginning of the packet. The SNR label is new for this architecture, and different to previous SNR estimator labels, spanning the length of the entire input sequence. Rather than estimating a single SNR value for the entire frame, the FCN estimator head will attempt

to estimate it on a per-sample basis, based on the receptive field of the convolutional layer filters.

Training dataset details are summarized in Table 6.9. The preamble lengths are now 16, 32 and 64, to make it easier to detect the frames in the noise of data via both baseline and FCN methods. The training set size is the same as previous FCN models, but the individual capture frames are twice as long at 400 samples per frame. Since the SNR estimator requires a range of SNRs to be effective, each of the 8k examples is set to a randomly chosen SNR value from a range of -10dB to 10dB. In these experiments, no phase or frequency offsets are applied to the training samples.

Table 6.9: FS-SNR MTL training details

Parameter	**Value**
Number of training examples	8192
Training SNRs	-10dB to 10dB
Preamble lengths	16, 32, 64 bits
Payload size	128
Capture size	400
Modulation	BPSK
Phase offset	0
Carrier offset	0

6.5.2 Architecture

Both frame synchronization and SNR estimation tasks are now being treated as sliding window regressions. This means that the SNR estimator head can be represented as a convolutional layer, in this case with exactly the same parameters as the frame synchronization head. To accommodate the longer preambles, the convolutional layer filters are set to a width of 55 samples, otherwise maintaining the same number of channels and using the same activation functions as previous FCN experiments in earlier sections.

The double-headed FCN MTL architecture details are summarized in Table 6.10. Both FS and SNR estimation heads share the weights of the initial two convolutional layers, producing linear outputs (which are free to emit negative and positive values) – this is important for the SNR estimator where the dB estimation of SNR can be

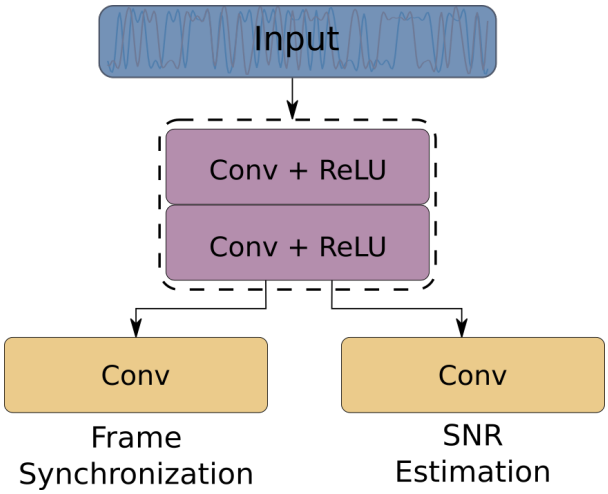


Figure 6.21: FCN FS+SNR Estimation MTL Architecture

positive and negative.

Table 6.10: Double headed FCN parameters

Layer	**Parameters**	**Output Shape**
Input		2 × N
Conv2D + ReLU	55 × 2, 32 filters	32 × 1 × N
Conv2D + ReLU	55 × 1, 32 filters	32 × 1 × N
Conv2D (FS head)	55 × 1, 1 filter	1 × 1 × N
Conv2D (SNR head)	55 × 1, 1 filter	1 × 1 × N

6.5.3 Training

Similar to the previous MTL implementations, the double-headed FCN architecture is trained with a variety of loss weightings. Keeping the weighting w_{FS} of the main task as 1, and gradually decreasing w_{SNR}, a sweep of five trained models per configuration is performed and evaluated over a test SNR range of -10 to 10dB.

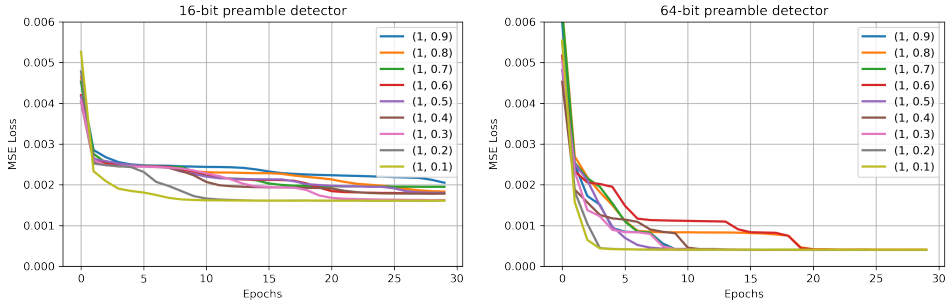
Training details for the MTL models are summarized in Table 6.11. The baseline FCN models with a single FS head are trained using exactly the same parameters, and random seeds for weight initialization, as the MTL models, the only difference being that they do not have an SNR estimation head attached. Both heads are evaluated

Table 6.11: FS-SNR MTL training details

Parameter	**Value**
Num. models	5
Loss weightings w_{FS}	1
Loss weightings w_{SNR}	0.1 to 0.9
Optimizer	ADAM
Loss function	MSE (both heads)
Batch size	32
Num. epochs	30
Learning rate (α)	0.001
Regularization Factor (λ)	0.001

using MSE loss.

Figure 6.22 shows the combined training losses for all weighting configurations for each of the 16-bit and 64-bit preamble detectors. It is evident that reducing w_{SNR} allows the FCN to converge more quickly to a solution. Interestingly, this becomes much slower when the preamble being detected is shorter, because the task is already significantly more difficult.



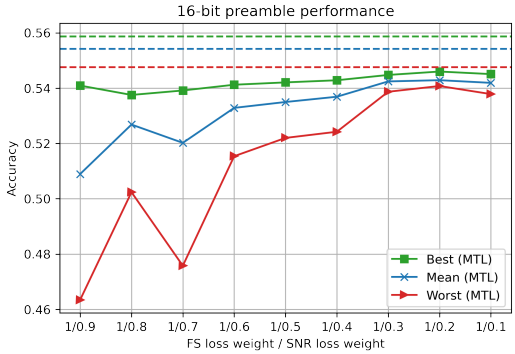
(a) 16-bit preamble losses (b) 64-bit preamble losses

Figure 6.22: Training losses of FS-SNR MTL models

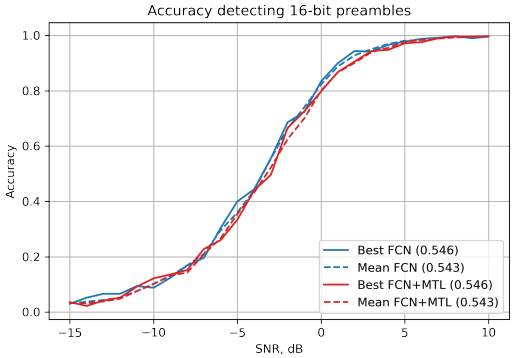
Frame Synchronization Results

In addition to the five baseline models, with five models per configuration, a total of 45 models per preamble length are evaluated over an SNR range of -10dB to 10dB. A high level overview of the performance of all models is shown in the left column subfigures (a), (c), (e) in Figure 6.23. The right column (subplots (b),(d),(f)) then show the

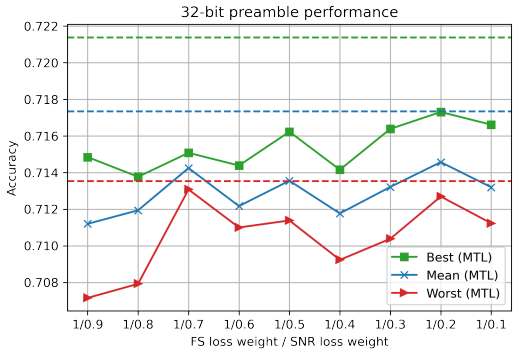
performance of the best models of all configurations compared with the baseline FCN models that were trained with no MTL.



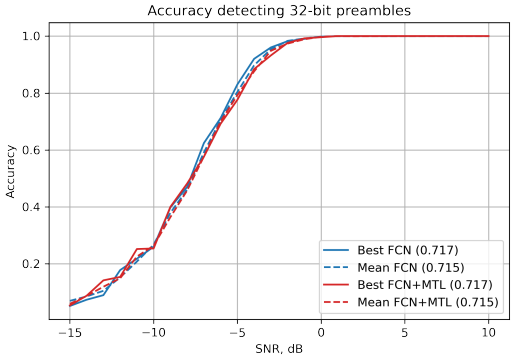
(a) 16-bit preamble loss tradeoffs



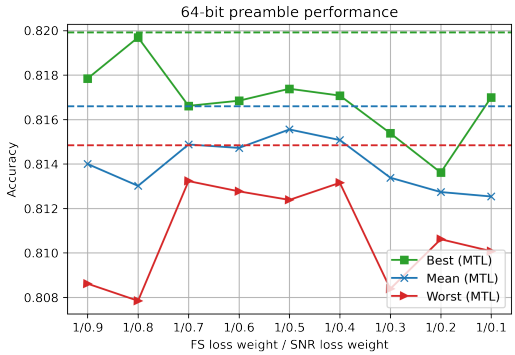
(b) FCN+MTL $w_{FS} = 1$, $w_{SNR} = 0.2$



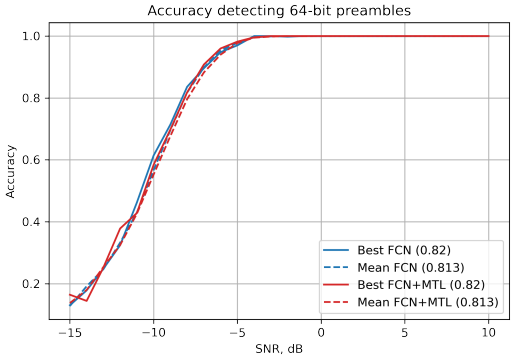
(c) 32-bit preamble loss tradeoffs



(d) FCN+MTL $w_{FS} = 1$, $w_{SNR} = 0.2$



(e) 64-bit preamble loss tradeoffs



(f) FCN+MTL $w_{FS} = 1$, $w_{SNR} = 0.8$

Figure 6.23: Comparison of mean accuracies achieved at different MTL loss weightings. Dashed lines indicate accuracies of baseline non-MTL FCN models.

As is evident from the results shown in Figure 6.23, adding SNR-awareness with

MTL has not provided any significant gains, and for the most part it has actually been detrimental on performance. The double-headed FCN architecture is a relatively shallow network, not containing a great deal of shared weights – perhaps to take full advantage of the induced features learned with the additional SNR estimation task, a deeper DNN is required, as in Section 6.3 with AMC.

6.5.4 Continuous Inference

As mentioned earlier in the chapter, MTL is used not only for improving performance, but also for saving resources by allowing two tasks to share feature extraction layers (i.e. the same DNN trunk). To explore simultaneous inference of the MTL FCN architecture, both heads are evaluated at the same time, on various frame lengths and SNR levels. For better visualization, all evaluations are performed with the FCN trained on 64-bit preambles.

Simultaneous FS and SNR Estimation

In this section two scenarios are investigated where FS and SNR estimation are performed simultaneously. First of all, a typical transmission with periodic packet transmissions is illustrated in the top subplot of Figure 6.24. The FCN and correlation methods are shown to detect the packets in this channel successfully. However, the FCN performs the additional job of an SNR monitoring network. As seen in the bottom subplot of Figure 6.24, the second FCN estimator head performs a per-sample estimation of the SNR. This is significant because it demonstrates that the FCN can easily function in a time-varying channel.

Furthermore, a sparser transmission is simulated in Figure 6.25. In this case there are only three packets, instead of five, and they only show up in the noisy portions of the time series signal. Even though there are no preambles in some portions of the waveform (e.g. in the ranges of 200-400 and 600-800 samples), the SNR estimator is still able to successfully predict SNRs close to the expected values – this gives confidence that the preamble does not need to exist for the SNR estimator head to be functional. Ideally the two tasks should not become co-dependent, and SNR estimation should

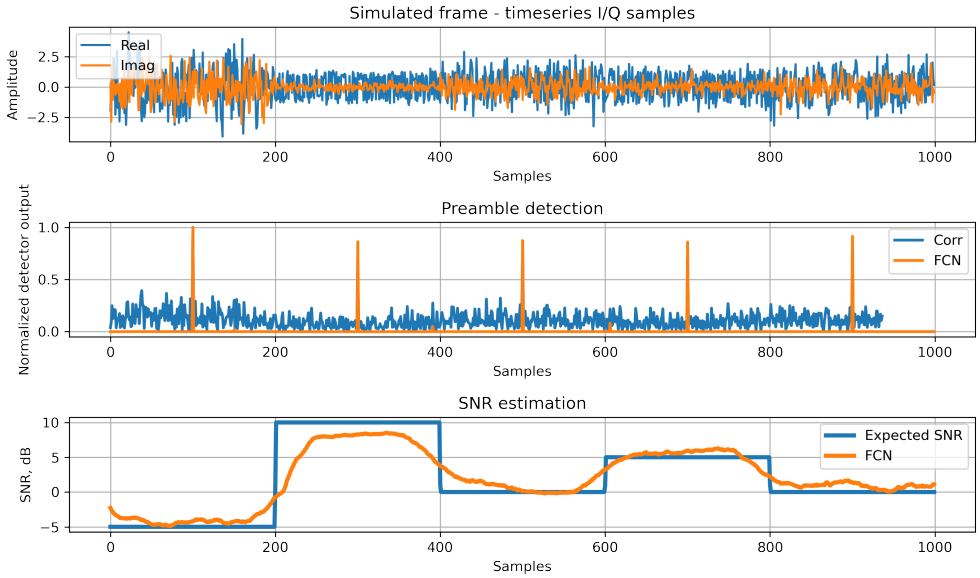


Figure 6.24: Periodic preamble detection

work exactly as expected without relying on preambles – the aim was to train a non-data-aided estimator.

This result shows an amazing level of generalization by the FCN, because the training set included only examples containing a single preamble in a transmission, it was not fed data without preambles during training. Figure 6.25 shows that it can not only be extended past the training example input length, but it can also estimate SNR without activating at least some of the neurons that are responsible for FS.

It should be noted that more conventional methods using statistical measures of the received signal can also achieve per-sample SNR estimations [132]. However, modern DNN-based methods are not capable of continuous estimation [50], [48], [49] and typically treat it as a classification problem on the entire input sequence, this is similar to how modern DL solutions for FS operate in reviewed works in Chapter 5.

A Closer Look at SNR Estimation

Examining the SNR estimation results in Figure 6.26, which shows the FCN estimation output over 400 samples, we can see that the outputs are quite irregular. This is likely due to the small receptive field of the FCN, which is 55 samples wide, being unable

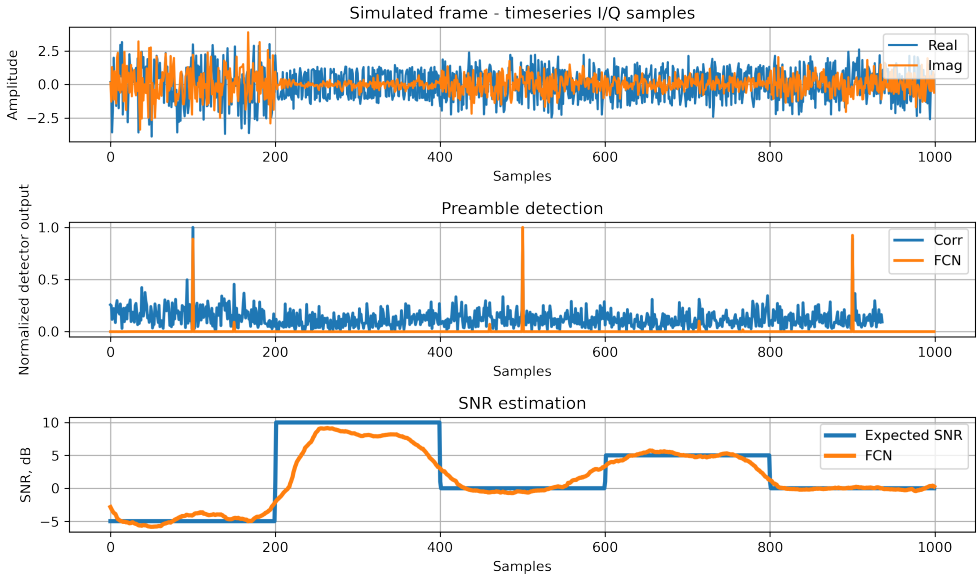


Figure 6.25: Sparse preamble detection

to calculate the statistics of the entire window of data. In an AWGN channel, by definition, the amount of noise per sample will vary, meaning that the short window the FCN has access to is only an estimation of SNR in that subset of samples, and not the entire frame.

The expected SNR is the average AWGN channel SNR in dB. When the FCN output is averaged, the dashed orange line in the figure shows that the FCN-estimated SNR is $\approx 5.4dB$, very close to the expected SNR of 5dB.

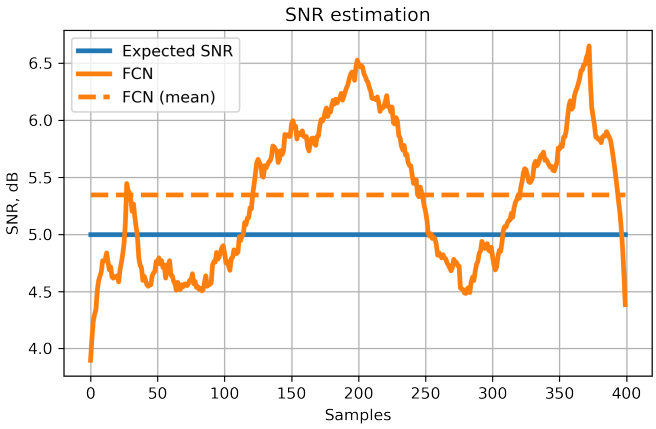
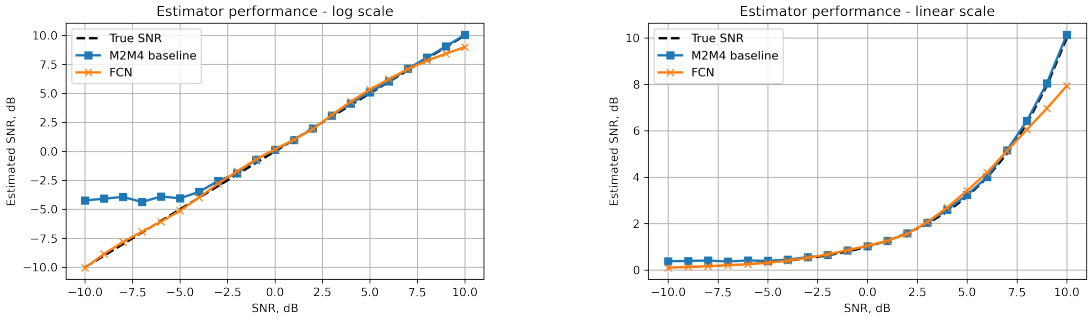


Figure 6.26: Closer look at FCN SNR estimation

The SNR estimator head was further evaluated quantitatively over the range of SNRs it was trained under, and compared to a baseline estimator based on the M2M4 algorithm [134], commonly used as a benchmark for blind SNR estimation of PSK modulated data.

The SNR estimator performance is summarized in Figure 6.27, where the FCN SNR estimator performance is shown in both linear and log scales. The estimated SNR was calculated by taking the mean of all 400 predictions of a single frame. This was done 500 times for each SNR level.



(a) Estimator performance in log scale

(b) Estimator performance in linear scale

Figure 6.27: SNR estimator performance comparison

By inspection of the predictions in Figure 6.27 (a), the FCN predictor is actually very performant and competently predicts SNR in high noise cases (under -5dB), especially compared to the M2M4 algorithm. As seen from DNN-based SNR estimators in Section 6.3.3, it does not perform as well at high SNR scenarios, which is clearer on a linear scale in Figure 6.27 (b). This can likely be alleviated by training on more higher SNR data. That said, accuracy at low SNR scenarios can be more important, as those are the channels where noise estimation is most beneficial to determine how reliable other estimators in the system are [132].

6.5.5 Observations

While training was primarily biased towards learning the FS task, the SNR estimation task still performed surprisingly well compared to baseline methods. It is worth noting that the FCN was never trained on configurations as shown in Figures 6.24 and 6.25,

training was performed on singular SNR captures per frame.

The numerical DER results of applying MTL and making the FCN SNR-aware did not seem to have any consistent performance increases, as seen in with SNR MTL in Section 6.3 or CFO MTL in Section 6.4. The advantage that this MTL architecture motivates is the resource savings of implementation due to the shared trunk of the DNN – gaining the ability of SNR estimation only required attaching a single convolutional layer to an FS FCN. An alternative would be training a whole new separate model like in Section 6.3.3, and running them in parallel, which may not be desirable, for example, in a resource-constrained embedded system.

6.6 Chapter Conclusion

When exploring DL architectures and techniques for wireless communications applications, the data driving these algorithms is mostly generated in simulation. Production of datasets requires setting various parameters for transmitter configurations and channel conditions, however this information is generally discarded. In this chapter it was demonstrated that, instead of discarding the channel impairment information to produce the dataset, it can be used as additional training data in the form of extra labels for MTL.

To prove out the concept, MTL was used for simultaneous AMC and SNR estimation, as well as FS with CFO estimation. Both showed an average of 1-3% consistent boost in evaluation metrics, such as average detection accuracy, over baseline DNN models. One big advantage is that, if the secondary task is not necessary for actual inference, it can be discarded and the trained DNN can be used for inference at no additional cost at runtime. The additional computational cost is paid entirely during training time.

Additionally, the combination of an FCN architecture and MTL training was shown for continuous per-sample inference on FS and SNR estimation tasks. MTL can be used effectively to reduce the number of resources required when deploying multiple DNN-based models in a communications system by having the individual task heads share the resources of the common feature extraction layers.

While good, consistent results were achieved for AMC and FS tasks, this comes with a large additional computation cost at training time. Secondary task architecture exploration, and loss weighting selection require training multiple additional models to determine what will improve performance. As the field matures, it is hoped that these searches will become narrower and the contributions presented in this work will drive research in this direction.

MTL with channel impairment estimation as a side-task could prove itself to be an effective general regularization technique in the wireless communications domain, especially when training models on simulated data.

Chapter 7

Conclusions

This thesis has presented multiple deep learning models for addressing wireless communications problems in the radio physical layer. A Seq2Seq autoencoder model was presented to address AMC and digital baseband demodulation. An FCN architecture was demonstrated for frame synchronization. Novel uses of MTL for wireless communications data were introduced to further improve data-driven model performance using channel simulation parameter as additional labels to improve regularization. The architectures and techniques produced in this work are intended to be easily re-purposed for a variety of new wireless communications tasks with little friction.

7.1 Resume

The main focus of the work described in this thesis was to investigate architectures and training methodologies that would be a good fit for the problems encountered in wireless communications. Many existing works have shown that replacing a standard DSP method with a DNN can improve the performance of the system. However, the majority of proposed DNNs in the wireless communications field are classification models, with very constrained deployment parameters, such as a fixed input size and specific channel conditions. In their seminal paper introducing DL for a variety of radio applications, T. O'Shea and J. Hoydis identified scalability of DNNs to larger input block sizes to be one of the biggest issues in the new DL-enabled communications paradigm [6]. While they were specifically talking about autoencoders, this has held true for other problems

– like synchronization and demodulation tasks.

Many existing DNNs applied in the wireless communications field cannot be deployed on arbitrary length inputs. While this may not be necessary for every problem, for tasks like frame synchronization a flexible architecture is a much more appropriate option [114]. If the DNN does not support variable input sizes, one option would be to treat it as an FFT block and allow designers to choose from a number of pre-trained models for various inputs, which can be an appropriate solution, albeit more costly to implement and maintain a larger number of DNNs. In order to lessen the friction of the adoption of DL methods for wireless receiver design, more flexible architectures need to be researched for appropriate problems.

One such architecture was introduced in Chapter 4 – a Seq2Seq model based on two RNNs arranged as an encoder-decoder structure. Using this approach an end-to-end system was developed, capable of tasks like AMC, matched filtering and baseband demodulation encapsulated in a single DNN. A convenient feature of the encoder-decoder arrangement is that the input and output sizes can be of arbitrary length, allowing the Seq2Seq model to perform matched filtering as well as downsampling in a single trainable module. Though the RNN-based autoencoder model is a powerful predictor, it is still difficult to train, even when applying LSTM cells, which were designed specifically to help train RNNs on longer sequences [71]. One solution to alleviate this training problem was identified in Section 4.5 – this included adding convolutional layers to the encoder of the Seq2Seq model, which reduced the computational burden on the RNN encoder by offloading the feature extraction job to a series of convolutional layers.

In wireless sensor networks transmission accounts for the majority of energy expenditure [109]. The authors of [137] have shown that reducing an IEEE 802.15.4 standard preamble transmission by as little as 2 bytes allow significant power savings on the battery-limited sensor node. In Chapter 5, the developed FCN model was shown to consistently outperform correlation-based methods for very short preamble lengths (8 and 16 bits), and, at high SNR, even matching 16-bit baseline performance with an 8-bit preamble. In contrast with existing DNN architectures used for frame synchronization, this approach allowed treating the FCN as a deep filter and deploying it on

inputs of arbitrary lengths.

In the wireless communications domain, simulation software for modeling complex channels with various impairments is quite mature and feature-rich. This means that, effectively, an infinite amount of data can be generated for training DL models. However it was shown in Chapter 4 that for a given model, providing more data will reach the point of diminishing returns. Many simulation parameters can be re-used as additional labels, enabling techniques like MTL. Chapter 6 presented a way of harnessing SNR and CFO estimation to improve AMC and frame synchronization performance of CNN and FCN models. The research presented in Chapters 5 and 6 has enabled the development of a continuous SNR estimator and frame synchronizer as a single DNN module. The trained double-headed FCN model showed comparable performance with standard SNR estimation methods like the M2M4 algorithm, while maintaining a similar frame detection accuracy to that achieved by the FCNs in Chapter 5.

Since every contribution has addressed a different problem, they can be treated as modular building blocks in future works. Seq2Seq models can be used for demodulation, while FCNs in Chapter 4 and 5 have shown to work well for feature extraction as well as frame synchronization. Since MTL has been proven to work for FCNs with other estimators in Chapter 6, it would be feasible to train a new autoencoder model with demodulation and synchronization as subtasks in an MTL-enabled training scheme. The demonstrated architectures and techniques enable various interesting new approaches to RFML problems.

7.2 Key Conclusions

The primary goals of this research were to identify and develop DL architectures that are more appropriate to wireless communications tasks than commonly deployed classification CNNs, and to investigate training techniques that are particularly well suited to this field.

Two architectures were identified showing the desired flexibility – the Seq2Seq autoencoder and the FCN. Additionally, MTL proved to be an effective training technique, which can add significant training overhead, yet using it as proposed in this work does

not impact implementation costs at inference time.

7.2.1 Seq2Seq Models

Seq2Seq models proved to be very capable of learning multiple tasks implicitly and Chapter 4 showed that it can outperform traditional ML-based AMC models to achieve an overall more accurate receiver that performs AMC and demodulation as a single DNN. It was also demonstrated how this architecture can be applied to different input/output lengths – with the input being a pulse shaped I/Q signal in the time domain, and the resulting output being the downsampled and demapped PSK symbols.

Nonetheless, RNNs can be more difficult to train than MLP or CNN-type models. The work in this thesis addresses the training difficulty to a degree by combining RNNs with CNNs for AMC and demodulation tasks. However, scaling to larger inputs and complex receivers remains a challenge.

7.2.2 FCNs for Frame Synchronization

In the context of IoT, wireless sensor networks and the rising need to preserve radio spectrum, there is an ever growing need to reduce the redundant symbols emitted by transmitters – one way to address this is with more complex receivers. The FCN architecture introduced in Chapter 5 proved to work surprisingly well for very short preamble lengths, outperforming correlation-based methods. The FCNs presented in this work showed robustness to various channel impairments, like phase offsets and CFO, as well as showing comparable performance to correlation in unseen fading channels.

Since the FCN architecture contains no fully connected layers, it can also be trained on a variety of input sizes, then deployed on arbitrary inputs, much like a DSP filter. Additionally, since these FCN models are not very large (only 3 convolutional layers), they should be straighforward to port to many hardware accelerators for practical deployment.

7.2.3 Training with MTL

With so many simulation tools there are countless possibilities for generating a dataset for training interesting DNN models. The work in Chapter 6 demonstrated how channel simulation parameters, namely SNR and CFO, can be used to improve the model performance. Without changing the architecture or dataset (other than saving the simulation parameters as additional labels) the tested CNN and FCN models consistently achieved 1-3% better accuracy in tasks like AMC and frame synchronization.

Of course, the tradeoff of achieving these gains is a much more complex training loop. The process involves determining a secondary task head architecture and tuning the additional loss weighting parameters. The hope is that, as the field matures, these hyperparameters will be understood better, and have the same level of documentation and guidelines as learning rate or weight decay. In future wireless systems, MTL could be considered as an important regularization technique, and appear as a standard feature in many communications-focused DL libraries. Hopefully the learnings gained from the MTL experiments in this thesis will contribute to the understanding of these techniques and help move the field towards that direction.

7.3 Limitations and Further Work

In any DL-based study it is difficult to evaluate every possible permutation of an architecture, hyperparameter, and in the case of wireless communications, every channel condition. The focus of this thesis was to present novel architectures and training techniques in problem domains, where commonly used DL solutions produce good results, but might not be the best fit for the task. As such, less emphasis was put on standard hyperparameter tuning, such as optimizer choice, learning rate, batch size, etc.

The accuracy improvements shown by applying MTL could potentially be replicated or even exceeded, by a different optimizer choice or putting more effort into regularization selection and tuning. The point of the study in Chapter 6, however, was to show that MTL is another valid ‘knob’ to dial, just like weight decay or learning

rate. A future study should include comparing MTL-derived gains to that of various regularization methods, with the most likely outcome being that a combination of these would produce the best models.

There is plenty of scope to further improve the explored architectures. The Seq2Seq model in this work has not been pushed to its limit and can benefit from further architectural modifications in future studies. For example, bidirectional RNNs [108] have been shown to outperform the unidirectional LSTM as was chosen for the Seq2Seq autoencoder in Chapter 4. These models also benefit greatly from adding an attention mechanism [107]. It is likely that even better accuracies would have been achieved with these enhancements. While the presented work showed good results for the considered problems, these architectural improvements would be worthwhile to pursue in future research.

That said, over the last several years RNNs have been falling out of favor, since the seminal "Attention is all you need" publication [138]. Given the prominence of LLMs (Large Language Models) and recent NLP trends, it is possible that transformer models will be the next go-to architectures for implementing wireless communications autoencoders. Early work on digital demodulation using transformers is already emerging [139]. It is unclear, at the time of writing, if transformers would be an appropriate replacement for RNN and CNN-based autoencoder networks in wireless communications. Nonetheless, given the success in the field of NLP, the new transformer models merit exploration.

The frame synchronization work in this thesis focused primarily on very short preambles meant for IoT and wireless sensor network applications. In these applications it is important reduce the overhead incurred by sending redundant bits, like preamble symbols, because transmissions cost the most energy and in battery-powered nodes this is a limited resource [109]. While the presented FCNs excel in applications for shorter preamble lengths (8 and 16 bits), the results for longer preambles showed none to minimal improvement in AWGN channels. In future work, bigger FCN models could be explored and their performance evaluated for more challenging scenarios like detecting the PSS signals of the 5G NR standard.

Besides architectural and training improvements, the next logical step to progress the implementation of these works into practical systems is the exploration of hardware deployment. From the explored architectures FCNs, in particular, would be easiest to adapt to real world applications due to their flexibility to various input sizes and robustness to channel effects, as shown in Chapter 5. From the introspection results in Section 5.6 it was determined that the FCNs have some 'dead' neurons, which indicates that techniques like pruning could be effective in further reducing the size of these models [122]. An analysis of the FCN models at various quantized bit-precision levels would be necessary to evaluate the readiness for hardware deployment.

7.4 Final Remarks

The need for connectivity is ever growing and radio spectrum is a precious limited resource. Advancements in receiver technology are essential in order to fully utilize the spectrum we have. Deep Learning is a key enabling technology that fuels many of the recent innovations in this area. In the author's opinion, the future wireless receiver will be built, in its entirety, using data-driven models. However, in the meantime, we still live in the transitionary period, and a lot of work remains to be completed in order to achieve the final receiver.

This thesis has presented a series of novel architectures and training methodologies to solve set of wireless communications problems. While the Seq2Seq architecture may soon be succeeded by transformers, FCNs seem like a very promising 'intermediary' model to replace existing receiver functions for frame synchronization, SNR estimation, and more. Given the breadth of available data in this field, MTL was shown to be a viable method of improving model performance, in a similar way to regularization.

It is hoped that this work will inspire future investigations into more flexible DNN architectures and more ways of utilizing untapped dataset generation parameters using MTL to further drive the field forward.

Appendix A

Training Runs

A.1 Seq2Seq

This section summarizes some of the training results from Chapter 4.

A.1.1 QPSK Demodulation

Tables A.1 and A.2 contain the QPSK demodulation model training results without and with teacher forcing respectively. The model with the best overall test accuracy is highlighted.

Table A.1: Seq2Seq for QPSK demodulation architecture sweep, 5 input symbols

Num. layers	**Hidden size**	**Weight decay**	**Best loss**	**Mean acc**	**Best acc**
1	16	0	0.203	72.88	72.88
1	32	0	0.201	72.96	72.96
1	64	0	0.214	72.24	72.24
1	128	0	0.218	71.61	71.61
1	16	0.0001	0.203	72.95	73.01
1	32	0.0001	0.201	72.91	72.96
1	64	0.0001	0.214	72.24	72.35
1	128	0.0001	0.218	71.61	71.66

1	16	0.0003	0.203	72.9	72.95
1	32	0.0003	0.201	72.99	73.01
1	64	0.0003	0.214	72.29	72.34
1	128	0.0003	0.218	71.87	71.91
1	16	0.001	0.203	72.87	72.96
1	32	0.001	0.201	72.96	72.98
1	64	0.001	0.214	72.56	72.64
1	128	0.001	0.218	72.2	72.26
1	16	0.003	0.203	56.53	72.82
1	32	0.003	0.201	41.06	72.96
1	64	0.003	0.214	25.03	25.09
1	128	0.003	0.218	24.92	24.95
2	16	0	0.191	72.99	72.99
2	32	0	0.199	72.4	72.4
2	64	0	0.212	71.86	71.86
2	128	0	0.214	72.18	72.18
2	16	0.0001	0.191	72.99	73.03
2	32	0.0001	0.199	72.37	72.45
2	64	0.0001	0.212	71.95	72.14
2	128	0.0001	0.214	72.15	72.22
2	16	0.0003	0.191	73.03	73.07
2	32	0.0003	0.199	72.53	72.61
2	64	0.0003	0.212	72.13	72.25
2	128	0.0003	0.214	72.3	72.31
2	16	0.001	0.191	73.0	73.03
2	32	0.001	0.199	72.78	72.83
2	64	0.001	0.212	72.63	72.79
2	128	0.001	0.214	72.56	72.66
2	16	0.003	0.191	25.08	25.11
2	32	0.003	0.199	25.01	25.02

2	64	0.003	0.212	24.95	25.03
2	128	0.003	0.214	24.98	24.99
3	16	0	0.19	72.86	72.86
3	32	0	0.194	72.3	72.3
3	64	0	0.209	72.27	72.27
3	128	0	0.217	72.23	72.23
3	16	0.0001	0.19	73.0	73.06
3	32	0.0001	0.194	72.45	72.6
3	64	0.0001	0.209	72.08	72.13
3	128	0.0001	0.217	72.23	72.35
3	16	0.0003	0.19	73.05	73.07
3	32	0.0003	0.194	72.59	72.69
3	64	0.0003	0.209	72.14	72.19
3	128	0.0003	0.217	72.26	72.29
3	16	0.001	0.19	24.9	24.92
3	32	0.001	0.194	56.94	72.96
3	64	0.001	0.209	24.95	25.02
3	128	0.001	0.217	25.01	25.07
3	16	0.003	0.19	25.03	25.1
3	32	0.003	0.194	25.03	25.05
3	64	0.003	0.209	25.0	25.05
3	128	0.003	0.217	25.06	25.12

Seq2Seq QPSK Demodulation Architecture Sweep Training Graphs

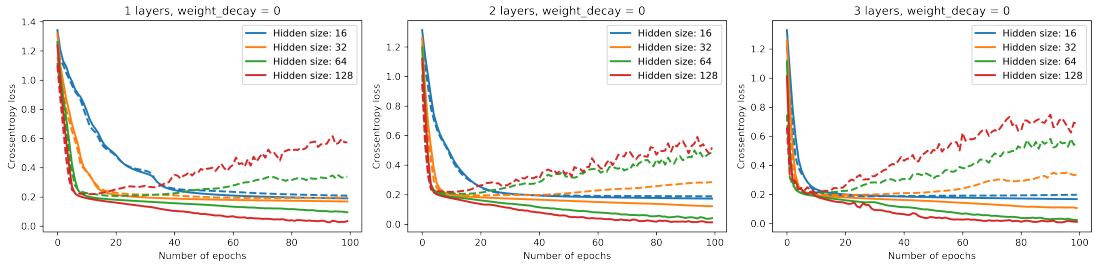


Figure A.1: No regularization

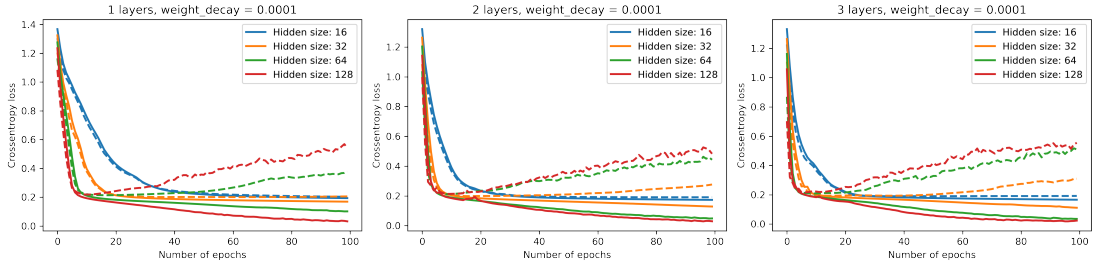


Figure A.2: Weight decay 0.0001

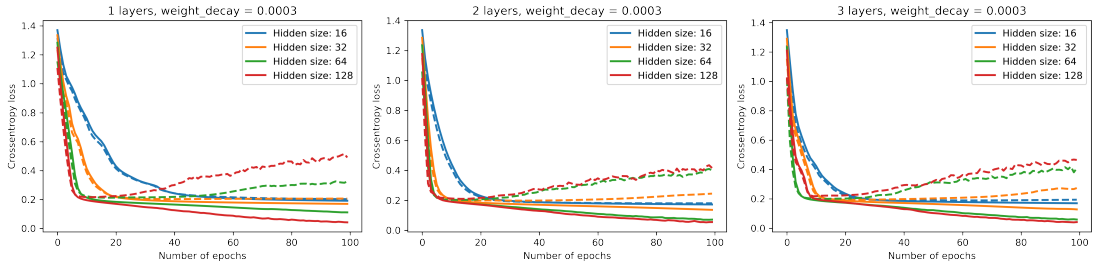


Figure A.3: Weight decay 0.0003

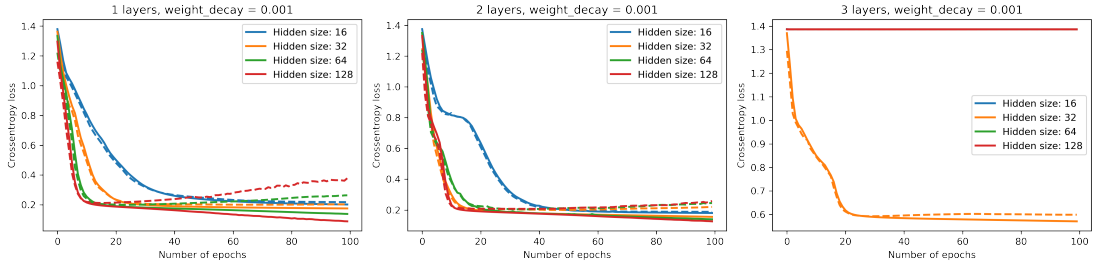
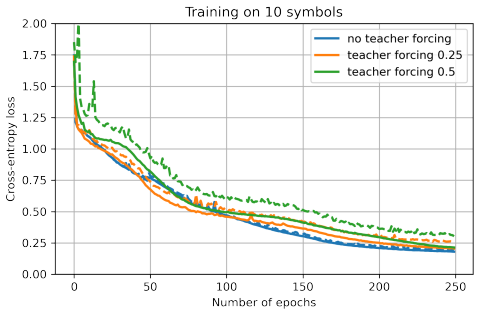


Figure A.4: Weight decay 0.001

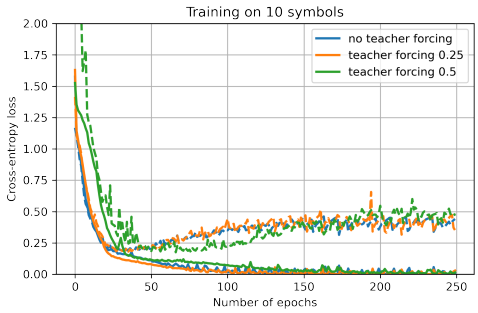
A.1.2 Simultaneous AMC and Demodulation

Teacher Forcing

Figures A.5-A7 summarize the training results for different input lengths and cell sizes.

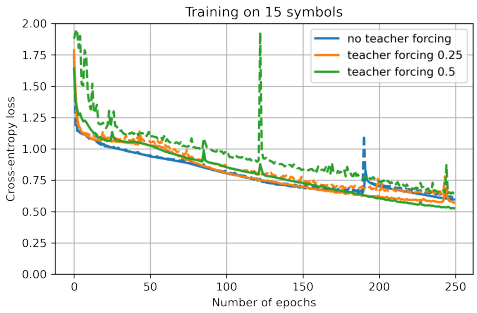


(a) Hidden size 16

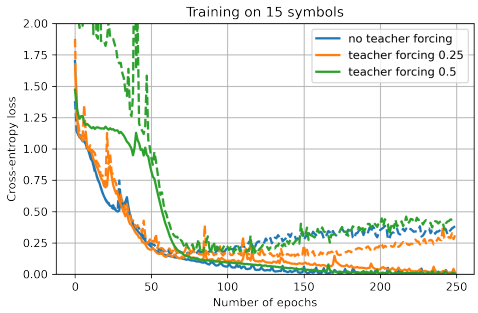


(b) Hidden size 128

Figure A.5: Training with teacher forcing on 10 symbol inputs

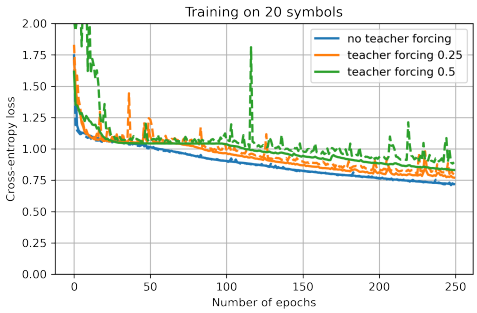


(a) Hidden size 16

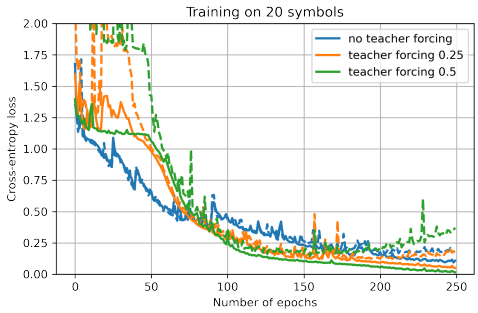


(b) Hidden size 128

Figure A.6: Training with teacher forcing on 15 symbol inputs



(a) Hidden size 16



(b) Hidden size 128

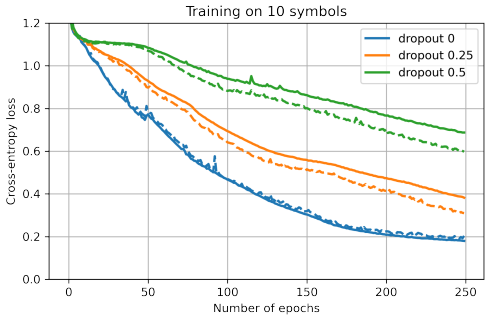
Figure A.7: Training with teacher forcing on 20 symbol inputs

Table A.2: Seq2Seq for BPSK/QPSK with teacher forcing

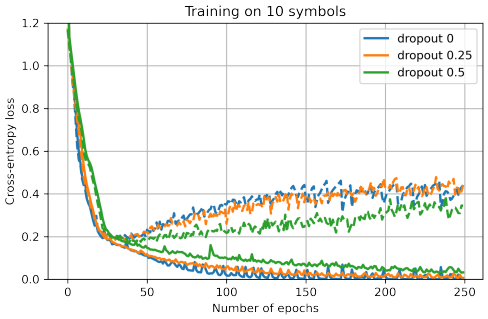
Sequence length	**Hidden size**	**Forcing rate**	**Best loss**	**Best val. loss**	**Mean acc**
10	16	0.00	0.18	0.191	0.762
10	16	0.25	0.205	0.259	0.748
10	16	0.50	0.213	0.303	0.703
10	128	0.00	0.001	0.191	0.69
10	128	0.25	0.001	0.181	0.687
10	128	0.50	0.004	0.175	0.672
15	16	0.00	0.596	0.616	0.633
15	16	0.25	0.57	0.634	0.652
15	16	0.50	0.524	0.643	0.693
15	128	0.00	0.001	0.159	0.676
15	128	0.25	0.013	0.14	0.686
15	128	0.50	0.002	0.171	0.674
20	16	0.00	0.719	0.711	0.648
20	16	0.25	0.771	0.798	0.554
20	16	0.50	0.832	0.882	0.54
20	128	0.00	0.094	0.165	0.68
20	128	0.25	0.05	0.123	0.68
20	128	0.50	0.016	0.145	0.687

Training With Dropout

Figures A.8-A10 summarize the training results for all input lengths and cell sizes with with dropout for regularization.

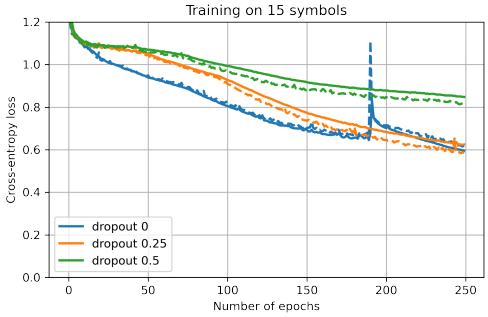


(a) Hidden size 16

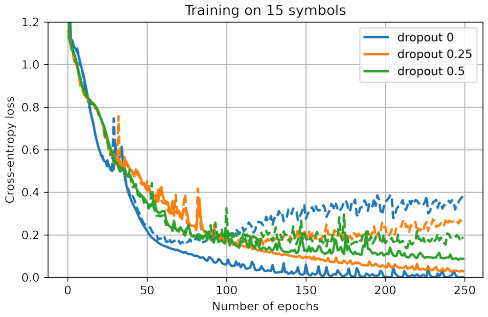


(b) Hidden size 128

Figure A.8: Training with dropout on 10 symbol inputs

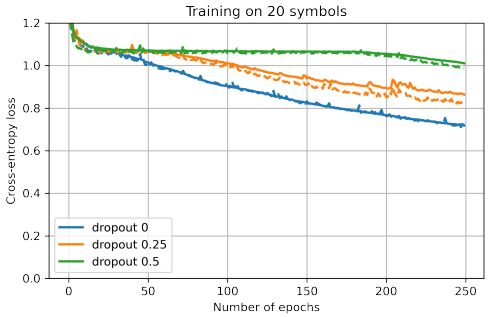


(a) Hidden size 16

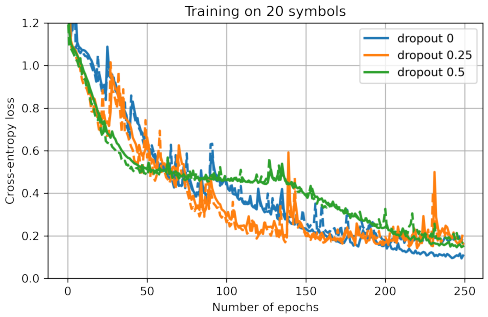


(b) Hidden size 128

Figure A.9: Training with dropout on 15 symbol inputs



(a) Hidden size 16



(b) Hidden size 128

Figure A.10: Training with dropout on 20 symbol inputs

Table A.3: Seq2Seq for BPSK/QPSK with dropout

Sequence length	**Hidden size**	**Dropout**	**Best loss**	**Best val. loss**	**Mean acc**
10	16	0.00	0.18	0.191	0.762
10	16	0.25	0.382	0.31	0.743
10	16	0.50	0.687	0.598	0.662
10	128	0.00	0.001	0.191	0.69
10	128	0.25	0.007	0.183	0.701
10	128	0.50	0.031	0.171	0.691
15	16	0.00	0.596	0.616	0.633
15	16	0.25	0.623	0.584	0.676
15	16	0.50	0.848	0.813	0.461
15	128	0.00	0.001	0.159	0.676
15	128	0.25	0.027	0.154	0.689
15	128	0.50	0.082	0.138	0.701
20	16	0.00	0.719	0.711	0.648
20	16	0.25	0.864	0.817	0.536
20	16	0.50	1.01	0.986	0.511
20	128	0.00	0.094	0.165	0.68
20	128	0.25	0.152	0.142	0.653
20	128	0.50	0.147	0.165	0.683

A.2 FCN

Table A.4: Initial architecture sweep training results: preamble length 8

Preamble length	Num. layers	Num. filters	Filter width	Best loss	Mean acc	Best acc
8	3	16	3	3.5e-03	0.235	0.289
8	3	16	9	3.6e-04	0.557	0.566
8	3	16	15	9.1e-04	0.540	0.575
8	3	16	35	1.1e-04	0.545	0.553
8	3	32	3	2.7e-03	0.277	0.281
8	3	32	9	1.7e-04	0.574	0.580
8	3	32	15	1.1e-04	0.580	0.586
8	3	32	35	8.9e-05	0.561	0.575
8	4	16	3	1.6e-03	0.154	0.174
8	4	16	9	8.7e-05	0.471	0.538
8	4	16	15	4.7e-04	0.429	0.499
8	4	16	35	8.3e-05	0.471	0.553
8	4	32	3	1.0e-03	0.100	0.117
8	4	32	9	2.3e-05	0.411	0.562
8	4	32	15	7.0e-06	0.415	0.531
8	4	32	35	2.9e-04	0.464	0.551
8	5	16	3	1.3e-03	0.163	0.187
8	5	16	9	1.8e-05	0.388	0.484
8	5	16	15	1.4e-05	0.397	0.481
8	5	16	35	9.9e-04	0.422	0.518
8	5	32	3	5.7e-04	0.113	0.138
8	5	32	9	7.8e-06	0.393	0.517
8	5	32	15	4.2e-06	0.428	0.471
8	5	32	35	1.6e-03	0.344	0.543

Table A.5: Initial architecture sweep training results: preamble length 16

Preamble length	**Num. layers**	**Num. filters**	**Filter width**	**Best loss**	**Mean acc**	**Best acc**
16	3	16	3	3.6e-03	0.219	0.279
16	3	16	9	3.6e-04	0.571	0.583
16	3	16	15	7.0e-04	0.663	0.680
16	3	16	35	4.2e-05	0.689	0.705
16	3	32	3	2.9e-03	0.275	0.281
16	3	32	9	1.8e-04	0.580	0.588
16	3	32	15	5.3e-05	0.695	0.716
16	3	32	35	5.2e-05	0.716	0.722
16	4	16	3	1.8e-03	0.144	0.161
16	4	16	9	3.9e-05	0.531	0.565
16	4	16	15	3.5e-04	0.608	0.717
16	4	16	35	2.1e-05	0.610	0.680
16	4	32	3	1.2e-03	0.122	0.135
16	4	32	9	1.6e-05	0.561	0.618
16	4	32	15	4.8e-06	0.570	0.655
16	4	32	35	2.5e-04	0.632	0.706
16	5	16	3	1.5e-03	0.132	0.153
16	5	16	9	1.8e-04	0.600	0.653
16	5	16	15	6.1e-06	0.566	0.627
16	5	16	35	5.8e-04	0.544	0.708
16	5	32	3	6.3e-04	0.115	0.131
16	5	32	9	8.1e-06	0.570	0.630
16	5	32	15	4.1e-06	0.648	0.691
16	5	32	35	6.5e-04	0.527	0.658

Table A.6: Initial architecture sweep training results: preamble length 32

Preamble length	**Num. layers**	**Num. filters**	**Filter width**	**Best loss**	**Mean acc**	**Best acc**
32	3	16	3	3.9e-03	0.187	0.256
32	3	16	9	4.1e-04	0.552	0.567
32	3	16	15	7.4e-04	0.652	0.683
32	3	16	35	2.2e-05	0.761	0.794
32	3	32	3	3.2e-03	0.251	0.257
32	3	32	9	2.2e-04	0.558	0.565
32	3	32	15	5.5e-05	0.674	0.693
32	3	32	35	3.0e-05	0.810	0.827
32	4	16	3	1.9e-03	0.164	0.185
32	4	16	9	6.3e-05	0.537	0.574
32	4	16	15	3.3e-04	0.630	0.725
32	4	16	35	2.8e-05	0.724	0.809
32	4	32	3	1.3e-03	0.115	0.128
32	4	32	9	1.3e-05	0.541	0.625
32	4	32	15	9.9e-06	0.611	0.678
32	4	32	35	3.0e-04	0.758	0.826
32	5	16	3	1.5e-03	0.155	0.182
32	5	16	9	2.1e-04	0.582	0.610
32	5	16	15	9.2e-06	0.599	0.701
32	5	16	35	7.8e-04	0.662	0.782
32	5	32	3	6.1e-04	0.126	0.154
32	5	32	9	9.4e-06	0.556	0.648
32	5	32	15	5.5e-06	0.580	0.678
32	5	32	35	4.9e-04	0.636	0.754

A.3 MTL

A.3.1 AMC+SNR MTL Models

Table A.7: AMC-MTL model training results, case 0: linear SNR estimator head

w_{FS}	w_{SNR}	Best loss	Best val. loss	Mean acc	Best acc
1	0.9	0.589	0.484	60.8	75.54
1	0.8	0.612	0.492	59.14	74.44
1	0.7	0.583	0.495	61.83	73.7
1	0.6	0.573	0.497	62.59	73.88
1	0.5	0.577	0.461	74.38	76.57
1	0.4	0.528	0.474	63.54	76.25
1	0.3	0.526	0.453	76.05	76.66
1	0.2	0.481	0.439	76.26	77.48
1	0.1	0.43	0.429	77.2	78.13

Table A.8: AMC-MTL model training results, case 1: dB SNR estimator head

w_{AMC}	w_{SNR}	Best loss	Best val. loss	Mean acc	Best acc
1	0.9	0.697	0.427	77.0	77.71
1	0.8	0.657	0.428	76.98	77.72
1	0.7	0.621	0.421	77.52	78.27
1	0.6	0.615	0.427	77.08	78.06
1	0.5	0.588	0.428	77.43	78.0
1	0.4	0.512	0.425	77.61	78.09
1	0.3	0.456	0.419	76.58	77.9
1	0.2	0.405	0.422	77.62	78.28
1	0.1	0.32	0.418	77.4	77.75

Table A.9: AMC-MTL model training results: classifcation SNR estimator head

w_{AMC}	w_{SNR}	Best loss	Best val. loss	Mean acc	Best acc
1	0.9	0.624	0.429	77.46	77.93
1	0.8	0.578	0.426	77.97	78.59
1	0.7	0.562	0.428	77.65	78.44
1	0.6	0.537	0.427	77.74	78.68
1	0.5	0.486	0.424	77.89	78.54
1	0.4	0.443	0.422	77.87	78.15
1	0.3	0.406	0.426	78.14	78.4
1	0.2	0.351	0.426	77.5	78.57
1	0.1	0.285	0.417	77.59	78.12

A.3.2 FS+CFO MTL Models

Table A.10: FS-MTL model training results with CFO estimator: using fixed FS loss weighting

w_{FS}	w_{CFO}	Best loss	Best val. loss	Mean acc (10kHz)	Mean acc (30kHz)
1	0.9	0.00476	6.03e-05	79.71	78.39
1	0.8	0.00459	6.46e-05	79.85	78.37
1	0.7	0.00418	6.63e-05	79.81	78.36
1	0.6	0.0038	6.2e-05	80.25	78.98
1	0.5	0.0034	6.08e-05	80.2	79.46
1	0.4	0.00294	6.68e-05	80.15	78.51
1	0.3	0.00242	6.11e-05	80.09	79.06
1	0.2	0.00211	5.87e-05	80.95	79.74
1	0.1	0.00158	5.54e-06	81.18	79.76

Table A.11: FS-MTL model training results with CFO estimator: using mixed loss weighting

w_{FS}	w_{CFO}	Best loss	Best val. loss	Mean acc (10kHz)	Mean acc (30kHz)
0.1	0.9	0.00438	0.000895	78.65	76.97
0.2	0.8	0.00418	0.000367	78.88	77.71
0.3	0.7	0.00377	0.000271	79.9	78.1
0.4	0.6	0.00358	0.000192	79.36	79.09
0.5	0.5	0.00321	0.000149	80.0	78.21
0.6	0.4	0.00274	0.000119	80.4	78.77
0.7	0.3	0.00236	0.000101	80.58	79.03
0.8	0.2	0.00208	7.04e-05	80.78	79.48
0.9	0.1	0.00897	5.66e-05	80.77	79.46

A.3.3 FS+SNR MTL Models

Table A.12: FS-MTL model training results with SNR estimator: using fixed loss weighting

Preamble length	w_{FS}	w_{CFO}	Best loss	Best val. loss	Mean acc	Best acc
16	1	0.9	0.00594	0.000409	50.9	54.09
16	1	0.8	0.00527	0.000407	52.68	53.75
16	1	0.7	0.00482	0.000397	52.02	53.92
16	1	0.6	0.00445	0.000406	53.29	54.12
16	1	0.5	0.00407	0.000403	53.5	54.21
16	1	0.4	0.00351	0.000405	53.69	54.28
16	1	0.3	0.00314	0.000397	54.24	54.48
16	1	0.2	0.00274	0.000395	54.29	54.6
16	1	0.1	0.00234	0.000397	54.19	54.51
32	1	0.9	0.00481	0.000111	71.12	71.48

32	1	0.8	0.00431	0.000109	71.19	71.38
32	1	0.7	0.0039	0.000109	71.42	71.51
32	1	0.6	0.00349	0.00011	71.22	71.44
32	1	0.5	0.00307	0.000108	71.36	71.62
32	1	0.4	0.0027	0.000111	71.18	71.42
32	1	0.3	0.00228	0.000108	71.32	71.64
32	1	0.2	0.00189	0.000108	71.46	71.73
32	1	0.1	0.00152	0.000106	71.32	71.66
64	1	0.9	0.00421	0.000139	81.4	81.78
64	1	0.8	0.00389	0.000136	81.3	81.97
64	1	0.7	0.0035	0.000137	81.49	81.66
64	1	0.6	0.00308	0.000136	81.47	81.68
64	1	0.5	0.00267	0.000138	81.56	81.74
64	1	0.4	0.00232	0.000141	81.51	81.71
64	1	0.3	0.00189	0.000139	81.34	81.54
64	1	0.2	0.00152	0.000136	81.27	81.36
64	1	0.1	0.00116	0.000136	81.25	81.7

Table A.13: FS-MTL model training results with SNR estimator: using fixed loss weighting

Preamble length	w_{FS}	w_{CFO}	**Best loss**	**Best val. loss**	**Mean acc**	**Best acc**
16	0.1	0.9	0.00432	0.00062	0.2723	0.3615
16	0.2	0.8	0.00423	0.000613	12.46	41.82
16	0.3	0.7	0.00391	0.000612	33.07	45.96
16	0.4	0.6	0.00368	0.000601	37.71	48.75
16	0.5	0.5	0.00355	0.000546	41.39	52.21
16	0.6	0.4	0.00315	0.000493	51.88	52.82
16	0.7	0.3	0.00289	0.000465	52.8	53.42

16	0.8	0.2	0.00255	0.000434	53.53	53.99
16	0.9	0.1	0.00223	0.000408	54.15	54.66
32	0.1	0.9	0.004	0.000609	1.682	5.992
32	0.2	0.8	0.00393	0.000595	52.01	67.03
32	0.3	0.7	0.00353	0.000307	70.3	70.77
32	0.4	0.6	0.00313	0.000204	70.75	71.17
32	0.5	0.5	0.00285	0.000168	71.0	71.32
32	0.6	0.4	0.00248	0.000143	71.13	71.45
32	0.7	0.3	0.00211	0.000128	71.37	71.57
32	0.8	0.2	0.00177	0.000119	71.4	71.71
32	0.9	0.1	0.00144	0.000112	71.15	71.57
64	0.1	0.9	0.00418	0.000602	13.36	57.63
64	0.2	0.8	0.00369	0.000299	60.77	81.13
64	0.3	0.7	0.00325	0.000208	64.24	81.13
64	0.4	0.6	0.00294	0.000178	77.49	81.39
64	0.5	0.5	0.00255	0.000161	81.14	81.53
64	0.6	0.4	0.00217	0.00015	81.32	81.7
64	0.7	0.3	0.00185	0.000142	81.53	82.03
64	0.8	0.2	0.00144	0.00014	81.52	81.77
64	0.9	0.1	0.00112	0.000138	81.6	81.85

Appendix B

Methodology for Training on Simulated Wireless Data

Throughout the work presented in this thesis, a great deal of models have been trained and evaluated. After much experimentation, a collection of high level advice has been constructed as a guide for anyone attempting to delve into DL in this area. This 'recipe' is heavily influenced by Andrej Karpathy's "A Recipe for Training Neural Networks" [140] and adapted for wireless communications, the original is a highly recommended read before proceeding. The methodology for training DNNs on (simulated) wireless communications datasets can be summarized as follows:

1. Data is the most important – use known solutions to reproduce baselines, these will be a temporary goal to aim for, an accuracy 'ceiling'. Spend time understanding the dataset, visualize it in multiple domains: time, frequency, spectrograms, etc. Visually inspecting the dataset can give valuable intuition and insights how to preprocess the data, or what layer types might work better.

2. Start with simple, minimal training dataset generated with high SNR. Establish a basic training and testing pipeline. The goal is to overfit a simple shallow DNN. Use a simple default optimizer like ADAM, with default learning rate and batch size (most DL libraries will have reasonable initial values that work for most easy problems). At this stage the model test accuracy will likely be very low – this

establishes the accuracy 'floor'.

3. Once the groundwork is set (training loop, evaluation functions), it is time to reduce SNR, make the dataset slightly more challenging, and push the initial model to the point where it is no longer overfitting. If a baseline exists, that can be an excellent reference for setting the training SNR – refer to Figure 5.8 (page 97).

4. At this stage the model prediction accuracies on the test set will hopefully be higher than previously established 'floor', but still lower than the 'ceiling'. Begin model architecture exploration, change the number of layers, neurons, experiment with layer and activation function types. The goal of this model is to outperform, or at least match, the baseline accuracy at the training SNR.

5. Add regularization. Weight decay or dropout are common choices, experiment with different parameters for these. Make sure your training loop has validation loss monitoring and saves the best model.

6. If the performance still does not match the baseline, you might want to generate more data (if training loss is high), add more regularization (if validation loss is high relative to training loss). If generating more data returns diminishing returns (refer to result in Figure 4.13 on page 79), revisit the model architecture. A flowchart for helping decide which stage to pay attention to based on the performance metrics is provided in Figure B.1.

7. Once the limit of iterating over architectures, optimization and dataset parameters is reached, try more advanced methods, like MTL.

A similar methodology would apply for non-simulated data, i.e. real data collected from over-the-air transmissions using an SDR. Of course it would be much more difficult to apply MTL in such a case, and, when it comes to real data, the best way to improve performance is to collect more of it.

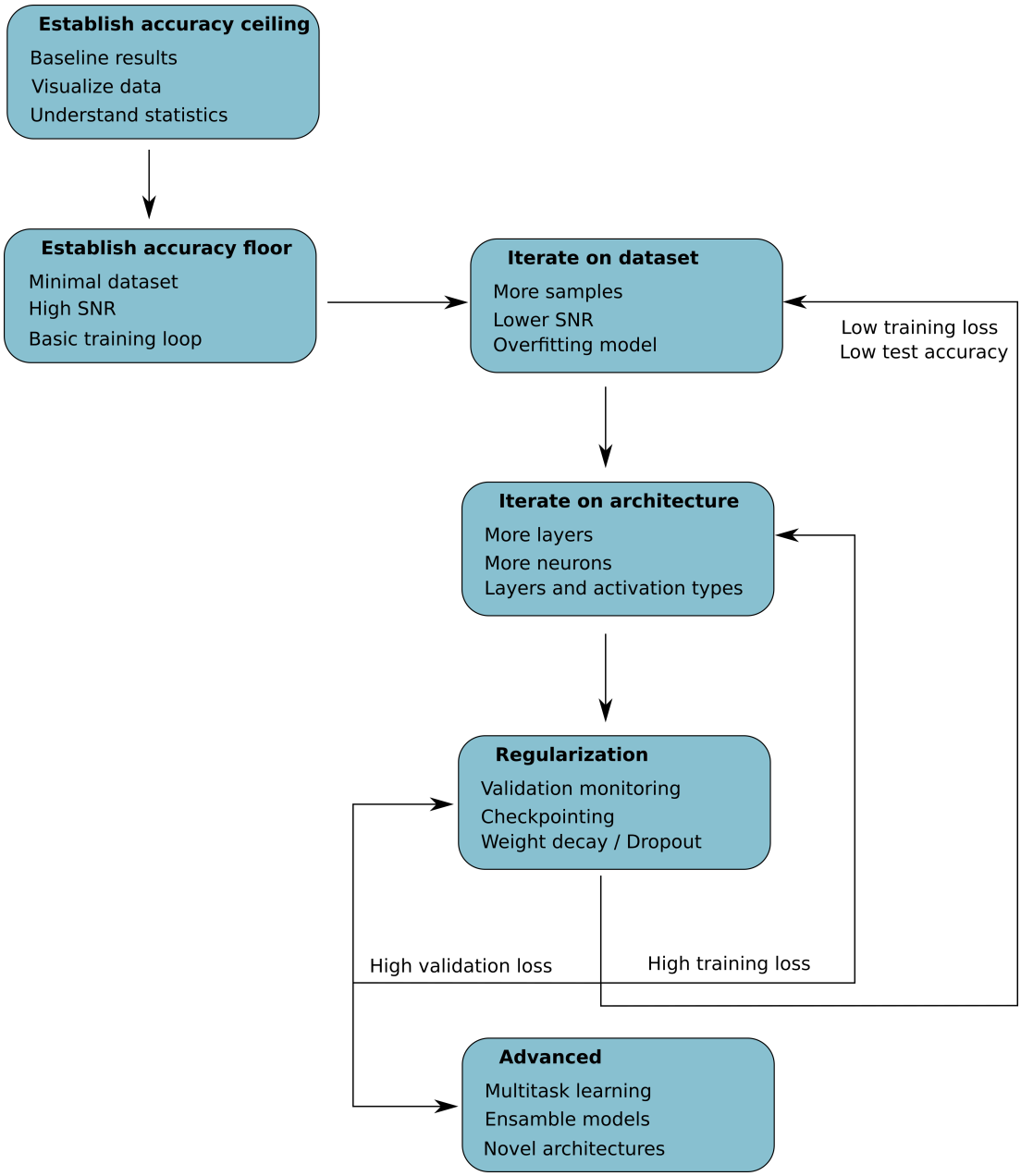


Figure B.1: Training DNNs on simulated wireless data

Appendix C

Jupyter Notebooks

The result for individual chapters can be accessed in the form of Python scripts and Jupyter Notebooks on this github repository `https://github.com/skalade/thesis`. The intention of publishing this code is to make the thesis as reproducible as possible and allow adoption of the findings to be prompt.

Bibliography

[1] H. He, S. Jin, C.-K. Wen, F. Gao, G. Y. Li, and Z. Xu, "Model-driven deep learning for physical layer communications," *IEEE Wireless Communications*, vol. 26, no. 5, pp. 77–83, 2019.

[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] N. Shone, T. N. Ngoc, V. D. Phai, and Q. Shi, "A deep learning approach to network intrusion detection," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 2, no. 1, pp. 41–50, 2018.

[4] T. J. O'Shea, J. Corgan, and T. C. Clancy, "Convolutional radio modulation recognition networks," in *Engineering Applications of Neural Networks*. Springer International Publishing, 2016, pp. 213–226.

[5] S. Cammerer, T. Gruber, J. Hoydis, and S. Ten Brink, "Scaling deep learning-based decoding of polar codes via partitioning," in *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, 2017, pp. 1–6.

[6] 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.

[7] C. Wang, F. Haider, X. Gao, X. You, Y. Yang, D. Yuan, H. M. Aggoune, H. Haas, S. Fletcher, and E. Hepsaydir, "Cellular architecture and key technologies for 5G

wireless communication networks," *IEEE Communications Magazine*, vol. 52, no. 2, pp. 122–130, 2014.

[8] T. Inoue, "5G standards progress and challenges," in *2017 IEEE Radio and Wireless Symposium (RWS)*, 2017, pp. 1–4.

[9] F. Ling, *Synchronization in digital communication systems*. Cambridge University Press, 2017.

[10] G. E. Bottomley, *Channel equalization for wireless communications: from concepts to detailed mathematics*. John Wiley & Sons, 2012.

[11] J. Zacharias, M. Barz, and D. Sonntag, "A survey on deep learning toolkits and libraries for intelligent user interfaces," *arXiv preprint arXiv:1803.04818*, 2018.

[12] R. Vinayakumar, K. P. Soman, and P. Poornachandran, "Applying deep learning approaches for network traffic prediction," in *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, 2017, pp. 2353–2358.

[13] O. Omotere, J. Fuller, L. Qian, and Z. Han, "Spectrum occupancy prediction in coexisting wireless systems using deep learning," in *2018 IEEE 88th Vehicular Technology Conference (VTC-Fall)*, 2018, pp. 1–7.

[14] O. Dobre, A. Abdi, Y. Bar-Ness, and W. Su, "Survey of automatic modulation classification techniques: classical approaches and new trends," *IET Communications*, vol. 1, pp. 137–156(19), April 2007.

[15] W. Xu, X. You, C. Zhang, and Y. Be'ery, "Polar decoding on sparse graphs with deep learning," in *2018 52nd Asilomar Conference on Signals, Systems, and Computers*, 2018, pp. 599–603.

[16] Hiriart-Urruty and Jean-Baptiste, "Conditions for global optimality 2," *Journal of Global Optimization*, vol. 13, pp. 349–367, 1998.

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[18] M. T. Ribeiro, S. Singh, and C. Guestrin, ""why should i trust you?" explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 2016, pp. 1135–1144.

[19] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," *Advances in neural information processing systems*, vol. 30, 2017.

[20] V. Belle and I. Papantonis, "Principles and practice of explainable machine learning," *Frontiers in big Data*, p. 39, 2021.

[21] G. Ras, N. Xie, M. Van Gerven, and D. Doran, "Explainable deep learning: A field guide for the uninitiated," *Journal of Artificial Intelligence Research*, vol. 73, pp. 329–397, 2022.

[22] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, "A survey of deep learning techniques for autonomous driving," *Journal of Field Robotics*, vol. 37, no. 3, pp. 362–386, 2020.

[23] S. Documentation, "Simulation and model-based design," 2020. [Online]. Available: https://www.mathworks.com/products/simulink.html

[24] D. Elbaz and M. Zibulevsky, "End to end deep neural network frequency demodulation of speech signals," *arXiv preprint arXiv:1704.02046*, 2017.

[25] D. Hong, Z. Zhang, and X. Xu, "Automatic modulation classification using recurrent neural networks," in *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*, 2017, pp. 695–700.

[26] M. Bkassiny, Y. Li, and S. K. Jayaweera, "A survey on machine-learning techniques in cognitive radios," *IEEE Communications Surveys Tutorials*, vol. 15, no. 3, pp. 1136–1159, 2013.

[27] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in neural information processing systems*, vol. 27, 2014.

[28] Y. Huangfu, J. Wang, R. Li, C. Xu, X. Wang, H. Zhang, and J. Wang, "Predicting the mumble of wireless channel with sequence-to-sequence models," in *2019 IEEE 30th Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*. IEEE, 2019, pp. 1–7.

[29] Z. Chen, Z. Zhang, and Z. Xiao, "Viewing the mimo channel as sequence rather than image: A seq2seq approach for efficient CSI feedback," in *2022 IEEE Wireless Communications and Networking Conference (WCNC)*, 2022, pp. 2292–2297.

[30] H. Sun, X. Zhu, Y. Liu, and W. Liu, "WiFi based fingerprinting positioning based on seq2seq model," *Sensors*, vol. 20, no. 13, p. 3767, 2020.

[31] C. R. Morales, F. R. de Sousa, V. Brusamarello, and N. C. Fernandes, "Multivariate data prediction in a wireless sensor network based on sequence to sequence models," in *2021 IEEE International Instrumentation and Measurement Technology Conference (I2MTC)*, 2021, pp. 1–5.

[32] T. J. O'Shea, K. Karra, and T. C. Clancy, "Learning approximate neural estimators for wireless channel state information," *CoRR*, vol. abs/1707.06260, 2017. [Online]. Available: http://arxiv.org/abs/1707.06260

[33] V. Ninkovic, A. Valka, D. Dumic, and D. Vukobratovic, "Deep learning-based packet detection and carrier frequency offset estimation in IEEE 802.11 ah," *IEEE Access*, vol. 9, pp. 99 853–99 865, 2021.

[34] H. Wu, Z. Sun, and X. Zhou, "Deep learning-based frame and timing synchronization for end-to-end communications," *Journal of Physics: Conference Series*, vol. 1169, no. 1, p. 012060, feb 2019. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/1169/1/012060

[35] J. M. Song, Y.-S. Kil, and S.-H. Kim, "Blind frame syncword detection using deep neural networks with input linear filtering," in *2019 International Conference*

on Information and Communication Technology Convergence (ICTC), 2019, pp. 1039–1041.

[36] E.-R. Jeong, E.-S. Lee, J. Joung, and H. Oh, "Convolutional neural network (CNN)-based frame synchronization method," *Applied Sciences*, vol. 10, no. 20, 2020. [Online]. Available: https://www.mdpi.com/2076-3417/10/20/7267

[37] Y.-S. Kil, J. M. Song, S.-H. Kim, T. Moon, and S.-H. Chang, "Deep learning aided blind synchronization word estimation," *IEEE Access*, vol. 9, pp. 30 321–30 334, 2021.

[38] M. Crawshaw, "Multi-task learning with deep neural networks: A survey," *arXiv preprint arXiv:2009.09796*, 2020.

[39] R. Caruana, "Multitask learning," *Machine Learning*, vol. 28, no. 1, pp. 41–75, 1997.

[40] S.-W. Fu, Y. Tsao, and X. Lu, "SNR-Aware convolutional neural network modeling for speech enhancement." in *Interspeech*, 2016, pp. 3768–3772.

[41] X. Xie, Y. Ni, S. Peng, and Y.-D. Yao, "Deep learning based automatic modulation classification for varying SNR environment," in *2019 28th Wireless and Optical Communications Conference (WOCC)*. IEEE, 2019, pp. 1–5.

[42] L. Weng, Y. He, J. Peng, J. Zheng, and X. Li, "Deep cascading network architecture for robust automatic modulation classification," *Neurocomputing*, vol. 455, pp. 308–324, 2021.

[43] S. Chang, S. Huang, R. Zhang, Z. Feng, and L. Liu, "Multitask-learning-based deep neural network for automatic modulation classification," *IEEE Internet of Things Journal*, vol. 9, no. 3, pp. 2192–2206, 2021.

[44] Y. Wang, G. Gui, T. Ohtsuki, and F. Adachi, "Multi-task learning for generalized automatic modulation classification under non-gaussian noise with varying SNR conditions," *IEEE Transactions on Wireless Communications*, vol. 20, no. 6, pp. 3587–3596, 2021.

[45] J. Qiao, W. Chen, J. Chen, and B. Ai, "Blind modulation classification under uncertain noise conditions: A multitask learning approach," *IEEE Communications Letters*, vol. 26, no. 5, pp. 1027–1031, 2022.

[46] Z. Chen, Z. Liu, X. Geng, Y. Zhao, and H. Wu, "Attention guided multi-task network for joint CFO and channel estimation in OFDM systems," *IEEE Transactions on Wireless Communications*, pp. 1–1, 2023.

[47] S. Liu and S. Wang, "Efficient carrier frequency offset estimation in wireless sensor networks," *IEEE Sensors Letters*, vol. 7, no. 5, pp. 1–4, 2023.

[48] X. Xie, S. Peng, and X. Yang, "Deep learning-based signal-to-noise ratio estimation using constellation diagrams," *Mobile Information Systems*, vol. 2020, pp. 1–9, 2020.

[49] Z. Zhao, B. Deng, F. N. Khan, and H. Fu, "Deep learning-based signal-to-noise ratio estimation for underwater optical wireless communication," in *2022 IEEE 14th International Conference on Advanced Infocomm Technology (ICAIT)*. IEEE, 2022, pp. 120–123.

[50] K. Yang, Z. Huang, X. Wang, and F. Wang, "An SNR estimation technique based on deep learning," *Electronics*, vol. 8, no. 10, p. 1139, 2019.

[51] H. Li, D. Wang, X. Zhang, and G. Gao, "Frame-level signal-to-noise ratio estimation using deep learning." in *INTERSPEECH*, 2020, pp. 4626–4630.

[52] B. Sklar *et al.*, *Digital Communications*. Prentice Hall, Upper Saddle River, NJ, USA, 2001, vol. 2.

[53] 3rd Generation Partnership Project (3GPP), "3gpp ts 38.101-2 version 15.2.0 release 15," European Telecommunications Standards Institute, Technical Specification TS 38.101-2, 2018, release 15.

[54] P. Gavrikov, P. E. Verboket, T. Ungan, M. Müller, M. Lai, C. Schindelhauer, L. M. Reindl, and T. Wendt, "Using Bluetooth low energy to trigger an ultra-low

power FSK wake-up receiver," in *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2018, pp. 781–784.

[55] C. Deng, X. Fang, X. Han, X. Wang, L. Yan, R. He, Y. Long, and Y. Guo, "IEEE 802.11be Wi-Fi 7: New challenges and opportunities," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 4, pp. 2136–2166, 2020.

[56] I. Glover and P. Grant, *Digital Communications*. Pearson Prentice Hall, 2009.

[57] M. Lichtman, "PySDR: A guide to SDR and DSP using Python," *URL: https://pysdr. org/content/sampling. html (visited on 12/12/2021)*, 2021.

[58] M. C. Jeruchim, P. Balaban, and K. S. Shanmugan, *Simulation of communication systems: modeling, methodology and techniques*. Springer Science & Business Media, 2006.

[59] E. Azzouz and A. Nandi, "Procedure for automatic recognition of analogue and digital modulations," *IEE Proceedings-communications*, vol. 143, no. 5, pp. 259–266, 1996.

[60] Z. Zhu and A. K. Nandi, *Automatic modulation classification: principles, algorithms and applications*. John Wiley & Sons, 2015.

[61] A. Nandi and E. Azzouz, "Algorithms for automatic modulation recognition of communication signals," *IEEE Transactions on Communications*, vol. 46, no. 4, pp. 431–436, 1998.

[62] A. Swami and B. Sadler, "Hierarchical digital modulation classification using cumulants," *IEEE Transactions on Communications*, vol. 48, no. 3, pp. 416–429, 2000.

[63] J. M. Mendel, "Tutorial on higher-order statistics (spectra) in signal processing and system theory: theoretical results and some applications," *Proceedings of the IEEE*, vol. 79, no. 3, pp. 278–305, Mar 1991.

[64] E. E. Azzouz and A. K. Nandi, "Automatic identification of digital modulation types," *Signal processing*, vol. 47, no. 1, pp. 55–69, 1995.

[65] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[66] MATLAB, *version 9.11.0 (R2021b)*. Natick, Massachusetts: The MathWorks Inc., 2021.

[67] GNU Radio. (accessed February 2022). [Online]. Available: http://www.gnuradio.org

[68] A. F. Agarap, "Deep learning using rectified linear units (regular)," *arXiv preprint arXiv:1803.08375*, 2018.

[69] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical evaluation of rectified activations in convolutional network," *CoRR*, vol. abs/1505.00853, 2015. [Online]. Available: http://arxiv.org/abs/1505.00853

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

[71] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 11 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[72] A. Graves, "Generating sequences with recurrent neural networks," *CoRR*, vol. abs/1308.0850, 2013. [Online]. Available: http://arxiv.org/abs/1308.0850

[73] A. Karpathy. (2015, May) The unreasonable effectiveness of recurrent neural networks. [Online]. Available: http://karpathy.github.io/2015/05/21/rnn-effectiveness/

[74] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[75] T. Tieleman, G. Hinton *et al.*, "Lecture 6.5-RMSProp: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.

[76] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, p. 1929–1958, Jan 2014.

[77] Q. Wang, Y. Ma, K. Zhao, and Y. Tian, "A comprehensive survey of loss functions in machine learning," *Annals of Data Science*, pp. 1–26, 2020.

[78] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[79] K. Nakayama and K. Imai, "A neural demodulator for amplitude shift keying signals," in *Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94)*, vol. 6, 1994, pp. 3909–3914 vol.6.

[80] K. Ohnishi and K. Nakayama, "A neural demodulator for quadrature amplitude modulation signals," in *Proceedings of International Conference on Neural Networks (ICNN'96)*, vol. 4, 1996, pp. 1933–1938 vol.4.

[81] G. de Veciana and A. Zakhor, "Neural net-based continuous phase modulation receivers," *IEEE Transactions on Communications*, vol. 40, no. 8, pp. 1396–1408, 1992.

[82] M. Onder, A. Akan, and H. Dogan, "Advanced neural network receiver design to combat multiple channel impairments," *Turkish Journal of Electrical Engineering and Computer Sciences*, vol. 24, no. 4, pp. 3066–3077, 2016.

[83] P. Gorday, N. Erdöl, and H. Zhuang, "Flexible FSK learning demodulator," in *2018 9th IEEE Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, 2018, pp. 633–639.

[84] P. Gorday, N. Erdöl, and H. Zhuang, "GFSK demodulation using learned sequence correlation," in *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*, 2020, pp. 0698–0704.

[85] P. E. Gorday, N. Erdöl, and H. Zhuang, "Complex-valued neural networks for noncoherent demodulation," *IEEE Open Journal of the Communications Society*, vol. 1, pp. 217–225, 2020.

[86] A. Ahmad, S. Agarwal, S. Darshi, and S. Chakravarty, "Deepdemod: BPSK demodulation using deep learning over software-defined radio," *IEEE Access*, vol. 10, pp. 115 833–115 848, 2022.

[87] M. Fan and L. Wu, "Demodulator based on deep belief networks in communication system," in *2017 International Conference on Communication, Control, Computing and Electronics Engineering (ICCCCEE)*, 2017, pp. 1–5.

[88] L. Fang and L. Wu, "Deep learning detection method for signal demodulation in short range multipath channel," in *2017 IEEE 2nd International Conference on Opto-Electronic Information Processing (ICOIP)*, 2017, pp. 16–20.

[89] H. Wang, Z. Wu, S. Ma, S. Lu, H. Zhang, G. Ding, and S. Li, "Deep learning for signal demodulation in physical layer wireless communications: Prototype platform, open dataset, and analytics," *IEEE Access*, vol. 7, pp. 30 792–30 801, 2019.

[90] X. Lin, R. Liu, W. Hu, Y. Li, X. Zhou, and X. He, "A deep convolutional network demodulator for mixed signals with different modulation types," in *2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, 2017, pp. 893–896.

[91] A. S. Mohammad, N. Reddy, F. James, and C. Beard, "Demodulation of faded wireless signals using deep convolutional neural networks," in *2018 IEEE 8th An-*

nual Computing and Communication Workshop and Conference (CCWC), 2018, pp. 969–975.

[92] W.-J. Chen, J. Wang, and J.-Q. Li, "End-to-end PSK signals demodulation using convolutional neural network," *IEEE Access*, vol. 10, pp. 58 302–58 310, 2022.

[93] M. Kozlenko, I. Lazarovych, V. Tkachuk, and V. Vialkova, "Software demodulation of weak radio signals using convolutional neural network," in *2020 IEEE 7th International Conference on Energy Smart Systems (ESS)*, 2020, pp. 339–342.

[94] T. Wu, "CNN and RNN-based deep learning methods for digital signal demodulation," in *Proceedings of the 2019 International Conference on Image, Video and Signal Processing*, ser. IVSP '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 122–127.

[95] N. Daldal, A. Sengur, K. Polat, and Z. Cömert, "A novel demodulation system for base band digital modulation signals based on the deep long short-term memory model," *Applied Acoustics*, vol. 166, p. 107346, 2020.

[96] K. Chia, V. M. Baskaran, K. Wong, M. L. Sim, and C. H. Chee, "Recurrent network with attention for symbol detection in communication systems," in *2022 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS)*, 2022, pp. 1–4.

[97] S. Kalade, L. Crockett, and R. W. Stewart, "Using sequence to sequence learning for digital bpsk and qpsk demodulation," in *2018 IEEE 5G World Forum (5GWF)*. IEEE, 2018, pp. 317–320.

[98] W. H. Clark, V. Arndorfer, B. Tamir, D. Kim, C. Vives, H. Morris, L. Wong, and W. C. Headley, "Developing rfml intuition: An automatic modulation classification architecture case study," in *MILCOM 2019 - 2019 IEEE Military Communications Conference (MILCOM)*, 2019, pp. 292–298.

[99] T. Linjordet and K. Balog, "Impact of training dataset size on neural answer selection models," in *Advances in Information Retrieval: 41st European Conference*

on IR Research, ECIR 2019, Cologne, Germany, April 14–18, 2019, Proceedings, Part I 41. Springer, 2019, pp. 828–835.

[100] D. Masters and C. Luschi, "Revisiting small batch training for deep neural networks," *arXiv preprint arXiv:1804.07612*, 2018.

[101] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Computation*, vol. 1, no. 2, pp. 270–280, 1989.

[102] PyTorch. "NLP from scratch: Translation with a sequence to sequence network and attention". [Online]. Available: https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html

[103] A. Abdelmutalab, K. Assaleh, and M. El-Tarhuni, "Automatic modulation classification using hierarchical polynomial classifier and stepwise regression," in *2016 IEEE Wireless Communications and Networking Conference*, 2016, pp. 1–5.

[104] A. Orvieto, S. L. Smith, A. Gu, A. Fernando, C. Gulcehre, R. Pascanu, and S. De, "Resurrecting recurrent neural networks for long sequences," *arXiv preprint arXiv:2303.06349*, 2023.

[105] A. X. M. Chang, B. Martini, and E. Culurciello, "Recurrent neural networks hardware implementation on fpga," *arXiv preprint arXiv:1511.05552*, 2015.

[106] A. X. M. Chang and E. Culurciello, "Hardware accelerators for recurrent neural networks on fpga," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2017, pp. 1–4.

[107] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[108] M. Schuster and K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[109] G. Anastasi, M. Conti, M. Di Francesco, and A. Passarella, "Energy conservation in wireless sensor networks: A survey," *Ad hoc networks*, vol. 7, no. 3, pp. 537–568, 2009.

[110] M. Chiani, "Noncoherent frame synchronization," *IEEE Transactions on Communications*, vol. 58, no. 5, pp. 1536–1545, 2010.

[111] J. G. Andrews, "A primer on Zadoff Chu sequences," *arXiv preprint arXiv:2211.05702*, 2022.

[112] E. Dahlman, S. Parkvall, and J. Skold, *5G NR: The next generation wireless access technology*. Academic Press, 2020.

[113] Y. Wang, C. Zhang, Q. Peng, and Z. Wang, "Learning to detect frame synchronization," in *Neural Information Processing: 20th International Conference, ICONIP 2013, Daegu, Korea, November 3-7, 2013. Proceedings, Part II 20*. Springer, 2013, pp. 570–578.

[114] S. Kalade, L. H. Crockett, and R. W. Stewart, "Training deep filters for physical-layer frame synchronization," *IEEE Open Journal of the Communications Society*, vol. 3, pp. 1063–1075, 2022.

[115] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 3431–3440.

[116] J. M. Johnson and T. M. Khoshgoftaar, "Survey on deep learning with class imbalance," *Journal of Big Data*, vol. 6, no. 1, pp. 1–54, 2019.

[117] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[118] H. Baniecki, W. Kretowicz, P. PiÄ, J. WiĹ *et al.*, "Dalex: responsible machine learning with interactive explainability and fairness in python," *Journal of Machine Learning Research*, vol. 22, no. 214, pp. 1–7, 2021.

[119] C. Patel, D. Bhatt, U. Sharma, R. Patel, S. Pandya, K. Modi, N. Cholli, A. Patel, U. Bhatt, M. A. Khan *et al.*, "Dbgc: Dimension-based generic convolution block for object recognition," *Sensors*, vol. 22, no. 5, p. 1780, 2022.

[120] R. Desislavov, F. Martinez-Plumed, and J. Hernandez-Orallo, "Compute and energy consumption trends in deep learning inference," *arXiv preprint arXiv:2109.05472*, 2021.

[121] J. Garland and D. Gregg, "Low complexity multiply accumulate unit for weight-sharing convolutional neural networks," *IEEE Computer Architecture Letters*, vol. 16, no. 2, pp. 132–135, 2017.

[122] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, "Pruning and quantization for deep neural network acceleration: A survey," *Neurocomputing*, vol. 461, pp. 370–403, 2021.

[123] T. Standley, A. Zamir, D. Chen, L. Guibas, J. Malik, and S. Savarese, "Which tasks should be learned together in multi-task learning?" in *International Conference on Machine Learning*. PMLR, 2020, pp. 9120–9132.

[124] D. Xu, W. Ouyang, X. Wang, and N. Sebe, "Pad-net: Multi-tasks guided prediction-and-distillation network for simultaneous depth estimation and scene parsing," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 675–684.

[125] T. Gong, T. Lee, C. Stephenson, V. Renduchintala, S. Padhy, A. Ndirango, G. Keskin, and O. H. Elibol, "A comparison of loss weighting strategies for multi task learning in deep neural networks," *IEEE Access*, vol. 7, pp. 141 627–141 632, 2019.

[126] B. Hilburn, N. West, T. O'Shea, and T. Roy, "SigMF: The signal metadata format," *Proceedings of the GNU Radio Conference*, vol. 3, no. 1, 2018. [Online]. Available: https://pubs.gnuradio.org/index.php/grcon/article/view/52

[127] 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, pp. 611–629, 2018.

[128] T. J. O'Shea, T. Roy, and T. C. Clancy, "Over-the-air deep learning based radio signal classification," *IEEE Journal of Selected Topics in Signal Processing*, vol. 12, no. 1, pp. 168–179, 2018.

[129] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[130] A. Karpathy *et al.* (2016) CS231n convolutional neural networks for visual recognition. [Online]. Available: http://cs231n.stanford.edu/

[131] MathWorks Inc. "NR cell search and MIB and SIB1 recovery". [Online]. Available: https://www.mathworks.com/help/5g/ug/nr-cell-search-and-mib-and-sib1-recovery.html

[132] F. Harris and C. Dick, "SNR estimation techniques for low SNR signals," in *The 15th International Symposium on Wireless Personal Multimedia Communications*, 2012, pp. 276–280.

[133] D. R. Pauluzzi and N. C. Beaulieu, "A comparison of SNR estimation techniques for the AWGN channel," *IEEE Transactions on communications*, vol. 48, no. 10, pp. 1681–1691, 2000.

[134] T. Salman, A. Badawy, T. M. Elfouly, T. Khattab, and A. Mohamed, "Non-data-aided SNR estimation for QPSK modulation in AWGN channel," in *2014 IEEE 10th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. IEEE, 2014, pp. 611–616.

[135] R. M. Dreifuerst, R. W. Heath, M. N. Kulkarni, and J. Charlie, "Deep learning-based carrier frequency offset estimation with one-bit ADCs," in *2020 IEEE 21st International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, 2020, pp. 1–5.

[136] Z. Wang, S. Wei, L. Zou, F. Liao, W. Lang, and Y. Li, "Deep-learning-based carrier frequency offset estimation and its cross-evaluation in multiple-channel models," *Information*, vol. 14, no. 2, p. 98, 2023.

[137] B. Bloessl and F. Dressler, "mSync: Physical layer frame synchronization without preamble symbols," *IEEE Transactions on Mobile Computing*, vol. 17, no. 10, pp. 2321–2333, 2018.

[138] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[139] M. Setzler, E. Coda, J. Rounds, M. Vann, and M. Girard, "Deep learning for spectral filling in radio frequency applications," 2022.

[140] A. Karpathy. (2019) A recipe for training neural networks. [Online]. Available: http://karpathy.github.io/2019/04/25/recipe/