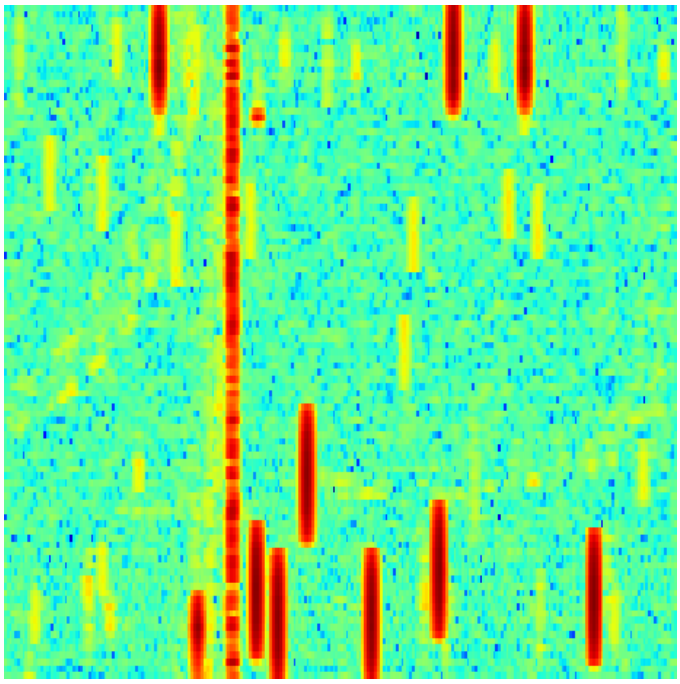

Detecting Jamming and Interference in Airborne Radar Using Convolutional Neural Networks

Master's thesis in Complex Adaptive Systems

ERIK WALLIN

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2019

Master's thesis

Detecting Jamming and Interference in Airborne Radar Using Convolutional Neural Networks

ERIK WALLIN

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Signal processing research group
Chalmers University of Technology
Gothenburg, Sweden 2019

Detecting Jamming and Interference in Airborne Radar Using Convolutional Neural Networks
ERIK WALLIN

Supervisors: Albert Nummelin and Dennis Sångberg, Saab Surveillance
Examiner: Thomas Rylander, Department of Electrical Engineering

Master's Thesis
Department of Electrical Engineering
Signal processing research group
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: DRFM jamming in a range-doppler map from an airborne radar.

Typeset in LaTeX
Gothenburg, Sweden 2019

Detecting Jamming and Interference in Airborne Radar Using Convolutional Neural Networks
ERIK WALLIN
Department of Electrical Engineering
Chalmers University of Technology

Abstract

Dealing with different forms of radar jamming is a central challenge in air-to-air combat. Jamming is a collection of techniques used for obstructing desired target-detection in an enemy radar. This thesis studies the possibilites of detecting jamming in radars by training a convolutional neural network with synthetic radar-data generated by software. This particular convolutional neural network is designed to classify range-doppler maps from airborne radars into four different classes: no jamming, noise jamming, DRFM jamming, and interference. A suitable network architecture and values for hyperparameters are found by iterative experimental studies. The final classification accuracy on a test set is 96.75%. This achieved accuracy suggests that convolutional neural-networks can be used to detect radar jamming with good results.

Keywords: airborne radar, radar jamming, ECCM, DRFM, machine learning, convolutional neural networks.

Acknowledgements

I would like to thank my supervisors, Albert Nummelin and Dennis Sångberg, for support and guidance throughout this thesis work. I would also like to thank my examiner, Thomas Rylander, for valuable comments and feedback. Finally, I would like to thank Magnus Enger and Saab Survaillance for giving me the opportunity to write this thesis and to conduct my work at their facilities.

Erik Wallin, Gothenburg, July 2019

Contents

List of Abbreviations

DRFM	Digital radio frequency memory
CNN	Convolutional neural network
PRI	Pulse-repetition interval
PRF	Pulse-repetition frequency
RCS	Radar cross-section
ECM	Electronic countermeasures
EW	Electronic warfare
JNR	Jamming-to-noise ratio
ECCM	Electronic counter-countermeasures
ReLU	Rectified linear unit
SGD	Stochastic gradient-descent

1

Introduction

This introduction initially provides some background and motivation for the work conducted in this thesis. This is followed by sections on radar jamming and machine learning to further explain the problem statement and why there is a demand for the work in this thesis. The last sections provide limitations of this thesis and give an overview of its different chapters.

1.1 Background

A central aspect of modern air-combat is the fight between radars on both sides. Both detecting the enemy with the own radar, and remaining undetected in the enemy radar are key aspects for success in air combat.

One way to obstruct the enemy from making valuable radar-measurements is to employ *radar jamming*. Jamming is a collection of methods with the purpose of injecting noise or false information into the enemy radar. In large-scale battles there is also a risk of unintentional jamming between friendly aircraft in the form of *interference*. Interference can happen when radars operate too close to each other at the same radio frequency.

To counter jamming, the radar both needs to identify the presence of jamming and what kind of jamming it is being exposed to. There are some traditional methods for handling jamming but these are often limited to the most basic types of jamming. At the same time, jamming methods are quickly becoming more advanced which creates a demand for more developed and accurate methods to counter jamming [1].

The objective for this thesis is to investigate the possibilities of developing techniques for countering radar jamming (including advanced jamming-techniques), using modern methods from the field of machine learning. The use of machine learning in radar contexts has not been thoroughly studied but is considered by some radar experts to have promising potential.

1.2 Jamming in range-doppler maps

An essential step of signal processing in radars is the construction of range-doppler maps. These image-like maps show received signal strength with respect to range

and range rate. A peak in a range-doppler map can indicate the presence of a target at the given range and range rate. The signal strength is typically given in decibel. An example of a range-doppler map which includes a target is given in figure 1.1. This figure also presents the colorbar that is used throughout this thesis (but is not shown in all subsequent figures).

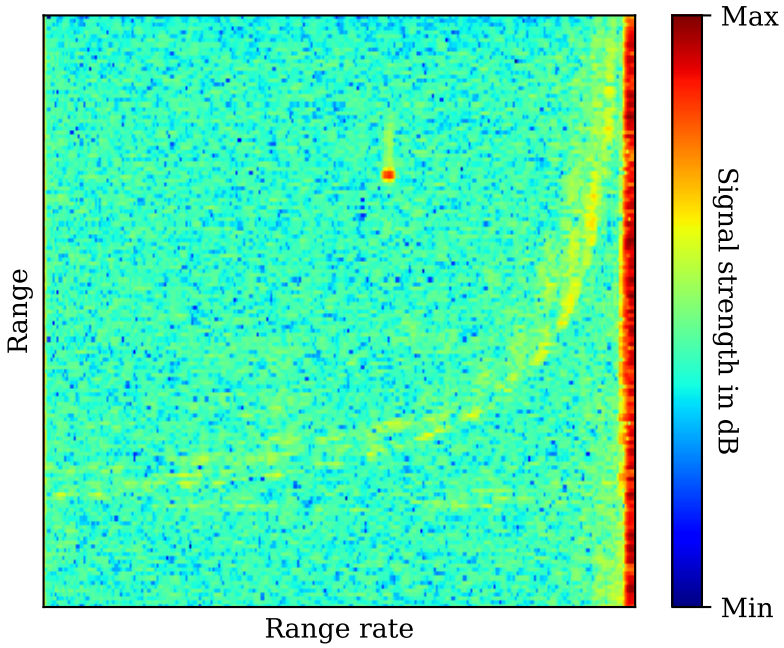

Figure 1.1: A typical range-doppler map. The sharp peak of signal strength in the upper middle part of the range-doppler map indicates the presence of a target at that given range and range rate. The vertical line at the right edge of the map is *ground clutter*. The signal strength is given in dB as shown by the colorbar.

Jamming either masks targets in range-doppler maps or creates deceptive patterns and targets that make it difficult to detect true targets. Target masking is generally achieved through *noise jamming* and deceptive patterns or targets are achieved through jamming with *digital radio frequency memory* (DRFM). An example of noise jamming in a range-doppler map is shown in 1.2 and an example of DRFM jamming can be seen in figure 1.3.

There exists traditional model-based methods used for detecting and countering noise jamming, but it is difficult to create model-based methods to handle DRFM jamming. It is however often possible for a trained radar-expert to identify DRFM jamming by looking at the range-doppler map.

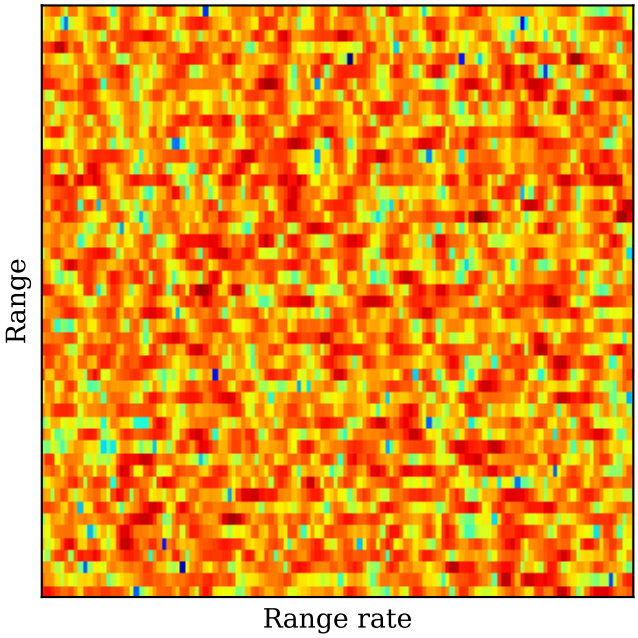

Figure 1.2: Noise jamming in a range-doppler map. The radar receiver is jammed by strong noise-signals that obstructs target detection.

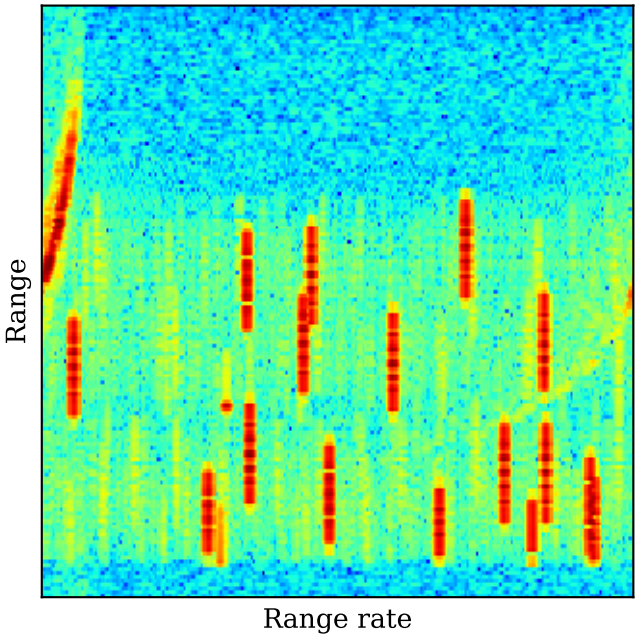

Figure 1.3: DRFM jamming in a range-doppler map. The radar receiver is jammed by deceptive signals that create false targets or misleading patterns in the range-doppler map.

1.3 Machine learning on radar data

Machine learning is the collective name for statistical methods that use a set of training data to create a model for making predictions on unseen data. Machine learning has historically been successful in situations where it is unrealistic or impossible to create explicit algorithms for automating tasks that traditionally are carried out by humans. Some examples are natural-language processing, speech recognition, and image analysis [2, 3, 4].

Classification of radar data is an example of a task that often is successfully accomplished by humans, but is difficult to automate with explicit instructions. This makes the classification problem of radar data a good candidate for machine learning. In particular methods designed for image-like data should be especially applicable to this problem, knowing that radar data can be presented as image-like range-doppler maps.

The current most popular and successful method for image classifications is *convolutional neural networks* (CNNs) [5]. CNNs have also been used to classify vehicles based on data from a frequency-modulated continuous-wave radar [6].

1.4 Ethical and societal aspects

Development of new technology is often accompanied by issues related to ethics and sustainability. This thesis deals with defense technology: a field where these matters are especially present. While defense technology is instrumental in ensuring the safety of nations, it is also subject to the risk of being used incorrectly for the wrong reasons. In order to minimize this risk, it is important to consider ethical and societal implications while conducting studies in this field.

This thesis in particular is focused on technology related to surveillance and survivability for aircraft. Improving these qualities leads to greater capabilities for accomplishing the main objective of defense technology: protecting nations against acts of aggression. To prevent undesirable use of the results in this thesis, any classified data or information are excluded. Furthermore, any work building on this thesis which might include classified data and information will follow strict regulations with respect to security and export control.

1.5 Aim

The purpose of this thesis is to develop a model to identify and classify jamming in radar data. The approach is to use machine learning based on convolutional neural networks to build a categorical classifier. Such a classifier should make important improvements to the possibilities of countering radar jamming.

1.6 Scope and limitations

The work in this thesis is limited to processing of digital data generated by an airborne radar. The thesis does not engage in any aspects of the hardware concerning radars or analog signal-processing methods. The scope of this work is also limited to processing of *synthetic data*, i.e. data generated by software.

1.7 Thesis overview

This thesis is composed of seven chapters. Following this introductory chapter comes two theory-themed chapters: the first on airborne radars and the second on machine learning with neural networks. These chapters provide the necessary theoretical backgrounds for the reader to understand the methods and results of the remaining chapters.

The fourth chapter describes the methods used for generating synthetic radar-data with focus on parameter selections for different data classes. The fifth chapter deals with the design of the machine-learning model: finding a good network architecture, setting hyperparameters, and selecting an optimizer. The following sixth chapter presents the classification results of the final model on a set of test data together with some learning history. The final seventh chapter gives conclusions from the obtained results and discusses possibilities of further work following this thesis.

2

Airborne radar and electronic countermeasures

This chapter provides an introduction to some radar concepts that are important for this thesis. We begin by covering some radar fundamentals: properties of pulse-doppler radars, ground clutter, and pulse compression. We then move on to describe the concept of electronic countermeasures and a few methods in this field that are central throughout the thesis. The last section explains interference in radars.

The discussions in this chapter are inspired by George W. Stimson's *Introduction to Airborne Radar* [1]. This book is recommended for further reading on the topics in this chapter.

2.1 Fundamentals of the pulse-doppler radar

Combat aircraft are in general equipped with a radar system of pulse-doppler type in order to detect enemies in air-to-air combat. The radar emits pulses of electromagnetic signals at a certain carrier frequency. An aircraft that is hit by these signals cause reflections that return to the radar receiver. By measuring these reflections, enemy targets can be detected by the radar system.

The pulse-doppler technique allows for measurements of both the range and the range rate of the target. The range to the target is determined by measuring the time delay of the radar echo. The range rate of the target is determined by measuring frequency shifts of echoes from a coherent train of pulses. Measurements of both range and range rates makes it possible to differentiate between moving targets and returns from the ground or other stationary objects.

2.1.1 Pulse-repetition interval

Each pulse train emitted by a pulse-doppler radar has a specified pulse rate. The time between each pulse transmission is called the *pulse-repetition interval* (PRI). The PRI is an important parameter which influences many characteristics of the pulse-doppler radar.

A radar pulse travels with the speed of light. A reflected signal travels both to the reflecting surface of the target, and back to the radar receiver. The round-trip time

for a reflected signal is thus

$$\tau = \frac{2r}{c_0} \tag{2.1}$$

where r is the distance to the target and c_0 is the speed of light. If however the round-trip time for a reflected signal is larger than the PRI, it is difficult to say whether the echo comes from the first pulse, or one of the subsequent pulses. The maximum range for which we have unambiguous range measurements, r_u, is thus the maximum range for which the echo from one pulse reaches the receiver before the next pulse is transmitted:

$$r_u = \frac{c_0 T_{\text{PRI}}}{2}. \tag{2.2}$$

Figure 2.1 gives a visualization of how range ambiguities arise in pulsed radars.

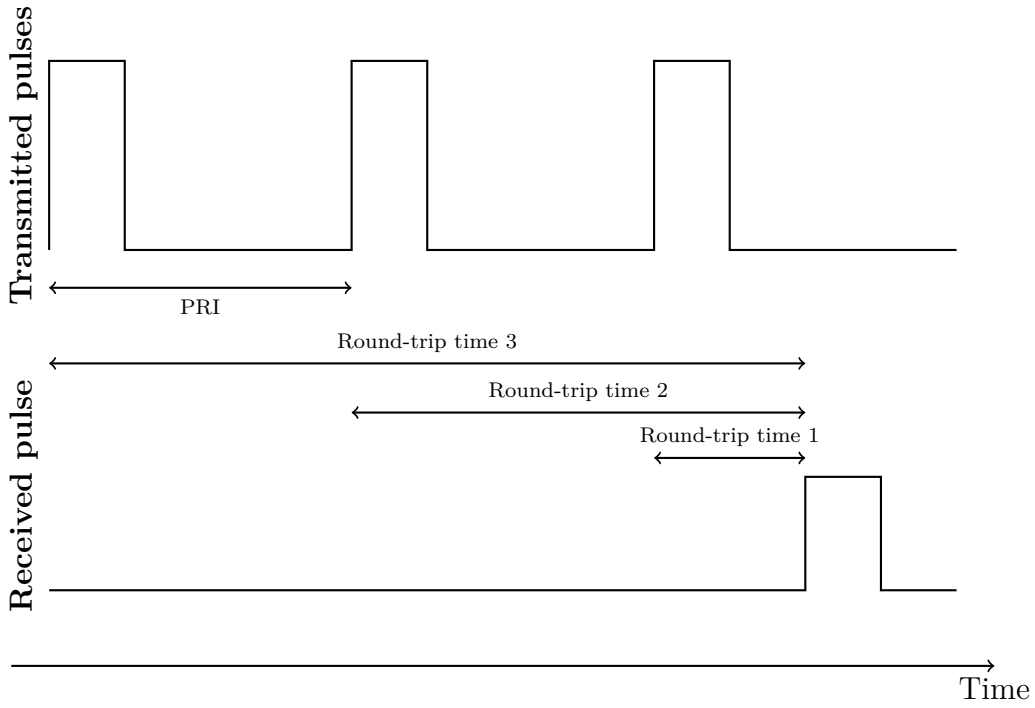

Figure 2.1: If the range to the target is larger than the maximum unambiguous range r_u, it is difficult to determine which of the transmitted pulses is causing the reflection. Here we see an example where the received echo can be a reflection from three different transmitted pulses.

To measure the range rate of detected targets, the radar makes use of the *doppler effect*. The doppler effect causes frequency shifts in the reflected signal if the relative velocity between the radar and target is not zero. The shift in frequency of the echo follows

$$\Delta f = -\frac{2\dot{r}f}{c_0}, \tag{2.3}$$

where $\dot{r}$ is the rate of change of the distance between the radar and the target (a positive $\dot{r}$ means the target is moving away from the radar), f is the radar carrier-frequency, and c_0 is the speed of light.

In order to measure frequency shifts in the reflected signal, a Fourier transform is applied to the train of pulse echoes. The Fourier transform converts the radar echoes from the time domain to a frequency spectrum. A central property of the pulse-doppler radar is the *coherency* of the emitted pulse train: the pulses have a constant phase difference. This allows for the Fourier transform to be applied across the full pulse train, making it possible to detect frequency shifts that otherwise would be too small to measure in a single pulse.

The frequency spectrum obtained from the Fourier transform shows the frequency content of the received signal. A peak in this spectrum indicates that the transformed time signal contains a periodic component of the given frequency. The spectrum obtained from a pulse-doppler radar has a peak corresponding to the frequency shift caused by the velocity of the target aircraft, $f + \Delta f$, but also peaks shifted by multiples of the *pulse repetition frequency* (PRF, the inverse of the PRI). These peaks are often referred to as *sidelobes* and have frequencies given by:

$$f + \Delta f + n f_{\text{PRF}} \text{ where } n = \pm 1, \pm 2, \pm 3, \ldots. \tag{2.4}$$

These sidelobes are caused by the discontinuities in the pulse train, which add frequency content to the signal that differs from the actual radio frequency. To avoid unambiguous range-rate measurements caused by the sidelobes, the PRF needs to be high enough so that the frequency sidelobes are placed outside of what is realistic for the operating conditions.

Another parameter that turns out to be important for the properties of the frequency spectrum is the number of pulses in the pulse train, N. The resolution of the frequency spectrum is defined by the width of the frequency peaks. The null-to-null width of the frequency peaks is

$$f_{\text{peak width}} = \frac{2 f_{\text{PRF}}}{N}. \tag{2.5}$$

So in order to achieve high resolution range-rate measurements, N needs to be high. The effects of the PRF and N on the frequency spectrum are summarized in figure 2.2.

We can conclude that the choice of PRI is of great importance for both range and velocity measurements. If we choose a low PRI the spectral sidelobes in the frequency spectrum have high separation, but the maximum unambiguous range measurement is low. If we instead choose a high PRI we can measure range at greater distances, but the spectral sidelobes are close to the true frequency peak. This causes ambiguities in the target range-rate measurement.

The choice of PRI is always a trade-off between accuracy in range measurements or velocity measurements and air-to-air operation generally requires switching between multiple PRI values to resolve ambiguities in range and range-rate measurements. Choosing the number of pulses, N, for each PRI is thus a trade-off between resolution in the frequency spectrum and how quickly we can resolve ambiguities.

A typical PRI for air-to-air combat is in the order of magnitude 100 µs and the number of pulses for one PRI is typically a few hundred.

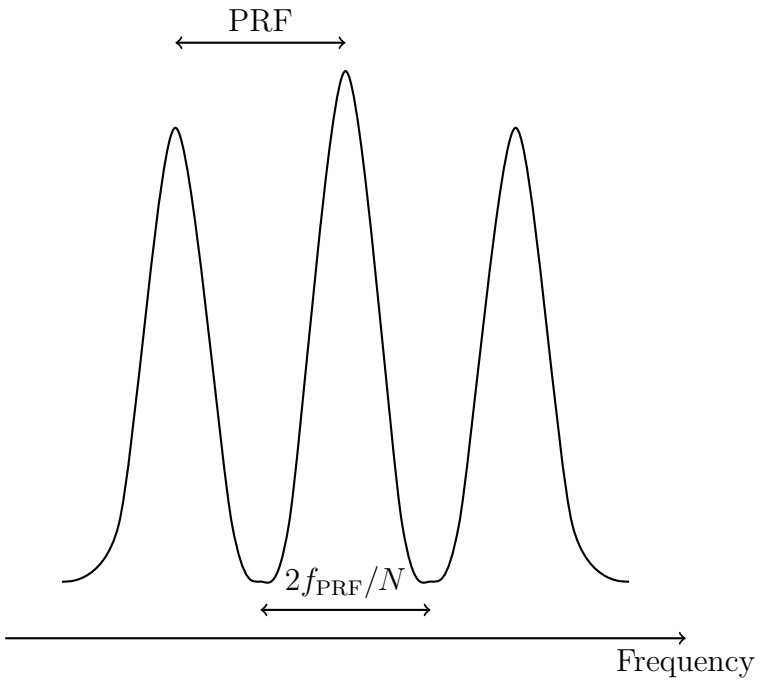

Figure 2.2: The frequency spectrum obtained by Fourier transforming the radar returns from a pulse-doppler radar. The spectrum has sidelobes at both sides of the true frequency peak. These sidelobes are separated by the PRF. The null-to-null bandwidth of the frequency peaks is $2f_{\mathrm{PRF}}/N$ and is what defines the resolution of the frequency spectrum.

2.1.2 Windowing

One way to reduce sidelobes in the frequency spectrum is to apply *windowing*. This is done by multiplying the radar signal with some window function before applying the Fourier transform [7].

The window function is zero outside a specified interval and is typically "bell shaped" with a maximum in the middle of this interval. If the input signal is N samples long, it is common to make the window function non-zero for N samples. This way the windowing can be applied to the entire input signal.

One common window function is the Gaussian window:

$$\begin{aligned} w(n) &= \exp\left[-\frac{1}{2}\left(\frac{n-N/2}{\sigma N/2}\right)^2\right] \\ \text{for } 0 \leq & n \leq N, \text{ zero elsewhere} \\ \sigma &\leq 0.5, \end{aligned} \tag{2.6}$$

where n is the sample number and σ is a scaling parameter. The Gaussian window with $\sigma = 0.4$ is shown in figure 2.3.

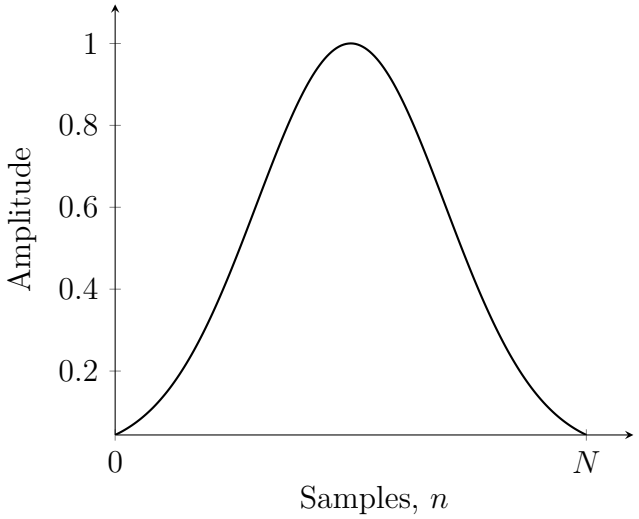

Figure 2.3: A Gaussian window-function with $\sigma = 0.4$.

If we have a discrete input signal $s(n)$, $0 \leq n \leq N$, we multiply this element wise with $w(n)$ before applying the Fourier transform. This reduces the frequency sidelobes that arise from Fourier transforming a finite time and discrete signal.

2.1.3 Pulse compression

The possibilities of detecting targets with a radar system are greatly related to the signal strength of the target echoes. As a radar pulse propagates to the target and back, its power is attenuated with a factor proportional to r^{-4}. If the received echo is too weak, we are not able to detect it in the presence of noise and ground

clutter. To achieve large detection ranges we must therefore use large transmission powers.

There are two obvious ways to increase the power output of the radar. The first way is to increase the *duty factor* (the ratio of the pulse length and PRI) by making the pulse length longer. The second way is to increase the peak power of each pulse. However increasing the pulse length leads to a lower resolution in range measurements: the range resolution is approximately half of the pulse length. Increasing the peak power to very high powers might also be practically impossible because of the high demands it puts on the radar transmitter.

The way we keep high detection ranges combined with fine range resolution is instead through *pulse compression*. Pulse compression is achieved through first applying some form of encoding to the transmitted pulse. Received echoes are then passed through a filter that compresses pulses that match the transmission encoding into narrow pulses of high power. This way we can transmit longer pulses to achieve high transmitted power and long detection ranges, without limiting the resolution of range measurements. The concept of pulse compression is shown in figure 2.4.

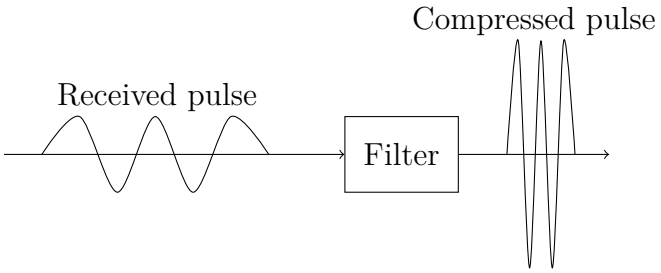

Figure 2.4: An illustration of pulse compression. The radar echo is passed through a filter in the receiver. If the echo matches the pulse-compression encoding, it is compressed to a shorter pulse of higher peak power.

The most basic application of pulse compression is through linear frequency-modulation. In linear frequency-modulation, the transmitted pulse has a linearly increasing or decreasing carrier frequency. The echoes that reach the radar are then passed through a filter that introduces a delay proportional to the carrier frequency. This way the beginning of the pulse is delayed longer than the end of the pulse and the pulse is compressed into a shorter pulse of higher peak power. There also exists non-linear pulse encodings that can be used for pulse compression.

2.1.4 Processing signals into a range-doppler map

To obtain meaningful information from the radar echoes, we need to process the received signals into a useful format. An often instrumental part of this processing is the construction of range-doppler maps. These maps show the strength of the received echoes with respect to range and range rate in a readable form. In

actual airborne radar applications, the range-doppler maps are typically used as input for some target-detection process. The target-detection process in turn outputs detection data that are presented to the radar operator. The construction of range-doppler maps involves a combination of the techniques that are already briefly covered in this chapter.

A pulse-doppler radar emits trains of pulses and listens for echoes between pulse transmissions. Each pulse train has a specified number of pulses, N. This means that each pulse train also has N listening intervals. Every listening interval is divided into discrete time steps. These discrete time steps can be directly translated to travel distance of the radar pulses, and are thus often referred to as *range gates*. A typical size of the range gates is around 100 m. The number of range gates, M, in each listening interval depends on the radar PRI and pulse length.

The radar stores the received signal in each range gate as a complex number (after applying pulse compression), whose real and imaginary parts represent two orthogonal components of the raw signal. These components are often referred to as *I/Q data*. The radar stores the IQ values for all range gates in all listening intervals, resulting in a $M \times N$-array of complex numbers. Each row of this array represents one range gate, and each column represents one listening interval (or pulse). To transform the columns of this array from the time domain to the velocity domain, we apply the Fourier transform to every *row* (scaled by some window function).

The output of the Fourier transform is complex: the magnitude represents the signal strength, and the argument represents the signal phase. Range-doppler maps of the type included in this thesis show the magnitude of the complex Fourier spectrum. Additionally, the magnitude is typically transformed into a decibel scale in order to bring out features of the data in a large range of values. For example, the decibel scale can allow a weak target echo to be visible in the presence of strong ground clutter.

Figure 2.5 shows how an array of I/Q data is transformed to a range-doppler map by applying the Fourier transform.

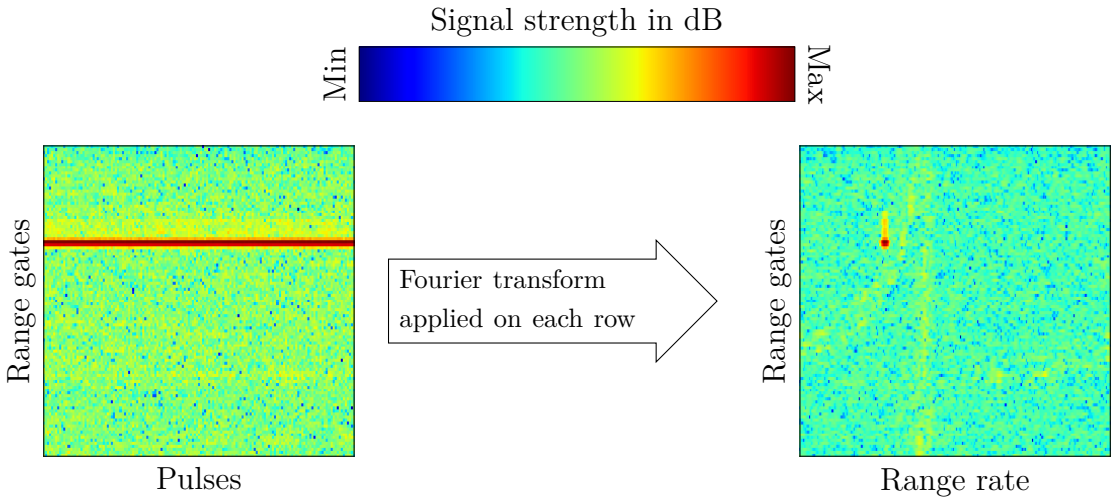

Figure 2.5: The left panel shows the magnitude of the recorded I/Q data in all range gates for every pulse of a pulse train. This particular example shows data from a pulse train with 151 pulses and 105 range gates. The right panel shows the result of applying the Fourier transform to every row of the data in the left panel. The Fourier transform is applied on the complex data, not on the magnitude which is shown in the figure. The Fourier transform translates the time dependent data from each listening interval to the velocity domain. We see that a horizontal line of strong radar returns in the left panel is transformed into a single peak with an observable range rate in the right panel. Both panels use a decibel scale, as shown by the colorbar.

2.1.5 Radar cross-section

The strength of target returns are determined by many factors. The power of the radar emitter and the distance to the target are two important aspects that are already mentioned. Another important factor is the *radar cross-section* (RCS) of the target. This factor is usually expressed in m^2 and is a measure of how visible a target is to radars. A larger RCS means the target is more easily detected by radars.

Many aspects of the target influence its RCS, some of these are:

- *The geometric area of the target*
- *The geometric shape of the target*
- *The surface material of the target*
- *The radar wavelength*

Airborne fighter-radars typically operate in the *X band*, which means using carrier frequencies around 10 GHz. Typical RCS values for radars in the X band are 2-3 m^2 for small fighter-planes and up to 100 m^2 for cargo aircraft [8].

2.1.6 Ground clutter

The radar not only receives echoes from enemy targets, but also from the ground. The echoes from the ground are often called *ground clutter*. Part of this clutter is echoes from the direction in which the radar antenna is pointing, *mainlobe clutter*. However all antennas also have *sidelobes* that receive returns from directions other than the mainlobe direction. The sidelobes generally receive ground return from large areas of the ground beneath the aircraft. The mainlobe and sidelobe clutters are illustrated in figure 2.6. The sidelobe return from the ground directly beneath the aircraft is called *altitude return* and is stronger than the other sidelobe clutter.

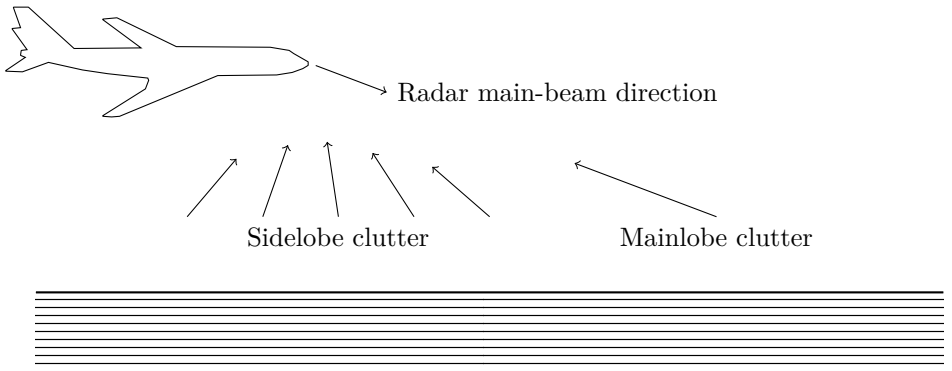

Figure 2.6: The radar receives undesired echoes from the ground. These returns are often called ground clutter. Ground echoes from the mainlobe direction are called mainlobe clutter. Clutter from other directions are sidelobe clutter.

The profile of the ground clutter depends on multiple factors: e.g. the antenna direction, the altitude of the aircraft, the velocity of the aircraft, and the PRI of the radar:

- The antenna direction impacts the strength and the doppler shift of the mainlobe clutter. If the antenna is pointing up there might be no mainlobe clutter. If the antenna is pointing toward the horizon, the mainlobe clutter covers a wide span of ranges but has a concentrated doppler shift. If the antenna is pointing down at a steep angle, the mainlobe clutter has a greater spread of doppler shifts but is more concentrated in range.
- The altitude of the aircraft determines the strength of the altitude return.
- The velocity of the aircraft determines the doppler shifts of the clutter: the ground is stationary so all doppler shifts of the ground clutter are due to the velocity of the aircraft. The doppler shift of the ground echoes are determined by the radial component of the relative velocity between the ground and the aircraft.
- The PRI of the radar sets the span for unambiguous range and range-rate measurements and thus determines how the ground that falls outside these spans is interpreted by the radar.

Examples of ground clutter with different antenna directions are shown in figure 2.7, 2.8 and 2.9.

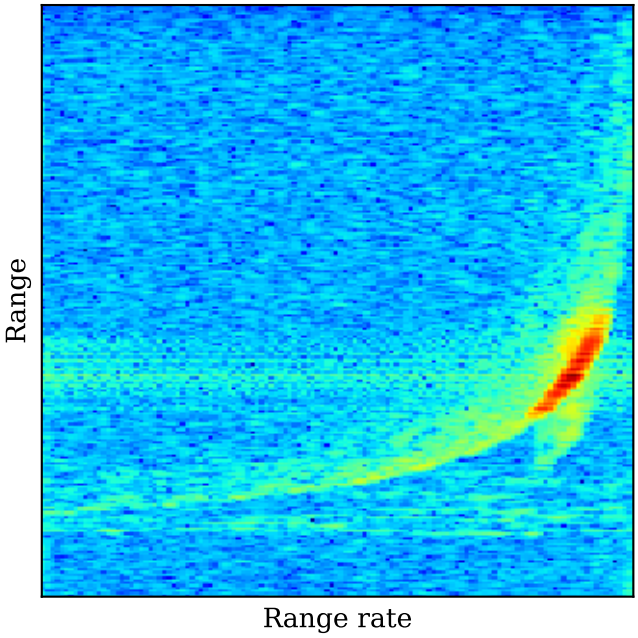

Figure 2.7: Ground clutter when the radar antenna is pointed at a slight downward-angle. The clutter creates a blob that has some spread in both range and range-rate.

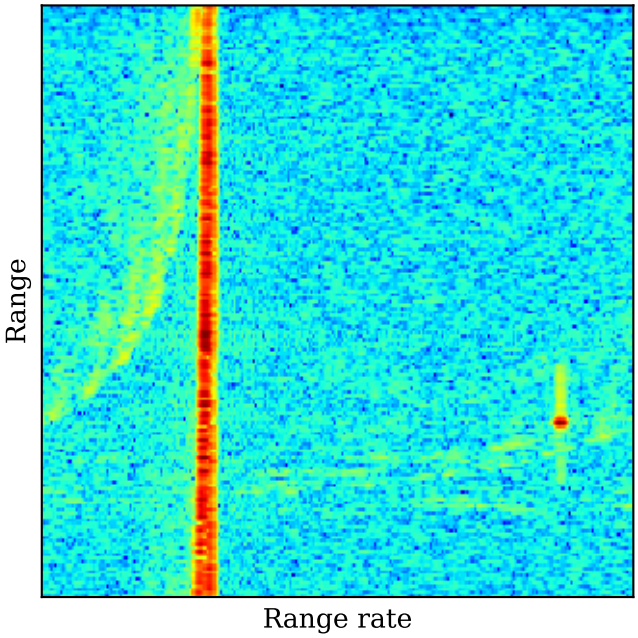

Figure 2.8: Ground clutter when the radar antenna is pointed toward the horizon. The clutter is concentrated to a narrow span of range rates but is spread across the full span of ranges. This creates a vertical line in the range-doppler map.

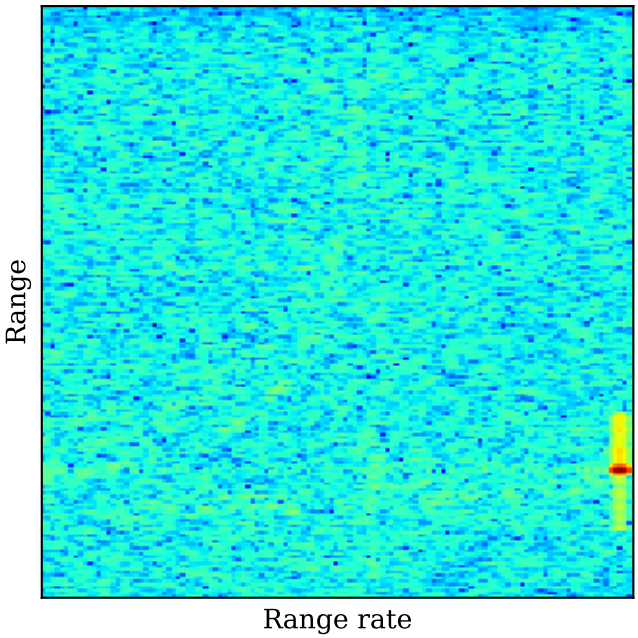

Figure 2.9: A range-doppler map from a radar with the antenna pointed up toward the sky. No strong returns from the ground are received.

2.2 Electronic countermeasures

Since radar operation is such an important element of air-to-air combat, it is natural that that methods for preventing valuable radar measurements have emerged. The purpose of these methods might be to prevent the own aircraft from being detected, or to provide deceptive signals that in other ways confuse the enemy radar. The collective name for these methods is *electronic countermeasures* (ECM) or simply jamming. A system which implements ECM techniques is usually referred to as an *electronic warfare-system* (EW system).

2.2.1 Noise jamming

One of the more simple ECM techniques is noise jamming. Noise jamming raises the levels of background noise against which target echoes has to compete, which might leave weak target-echoes undetected. The power levels of noise jamming is often specified by a *jamming-to-noise ratio* (JNR) given in dB. This value is the ratio of the signal strengths of the jamming noise and the background noise. A higher JNR means higher jamming power.

The noise can be distributed uniformly both in the time and frequency domain. It can be also be concentrated to a certain frequency band or certain time intervals. Jamming with continuous white noise raises the background over the whole range-doppler map. This type of jamming is effective and easy to implement, but the transmitted energy from the EW system needs to be distributed over all frequencies which limits the peak power, making target returns "burn through" the noise at a longer range.

To raise the noise power-levels, the frequency of the transmitted noise can instead be limited to a more narrow frequency band. We expect the target returns to lie in a frequency band centered around the enemy radar carrier frequency, so to maximize the power we can to limit the noise to these frequencies. This technique is called spot noise jamming. The noise might still cover the entire range-doppler map of the enemy radar but the noise levels are higher.

To further concentrate the noise power we can limit the noise to certain time intervals, so that the noise covers a range span we expect the target to be in. This technique is called range-bin masking.

Jamming with continuous noise is shown in figure 2.10 and jamming with range-bin masking is shown in figure 2.11.

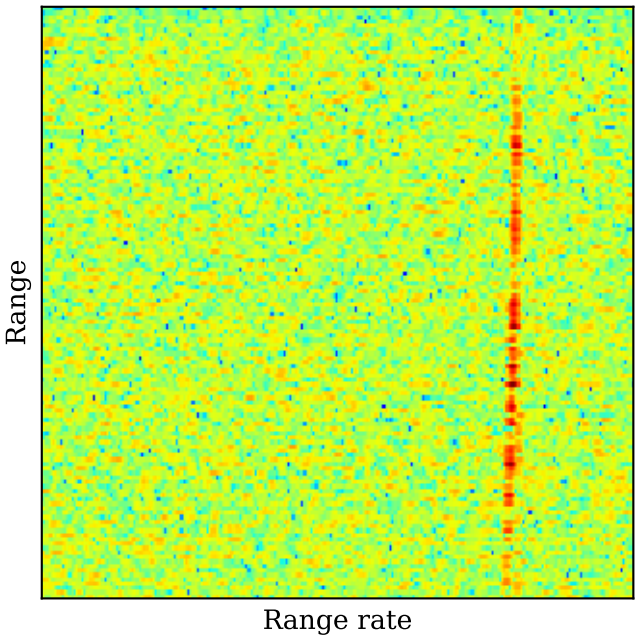

Figure 2.10: Noise jamming with continuous noise. The noise covers the whole range-doppler map. The noise in this range-doppler map is not strong enough to mask the ground clutter, but it likely covers weak target echoes.

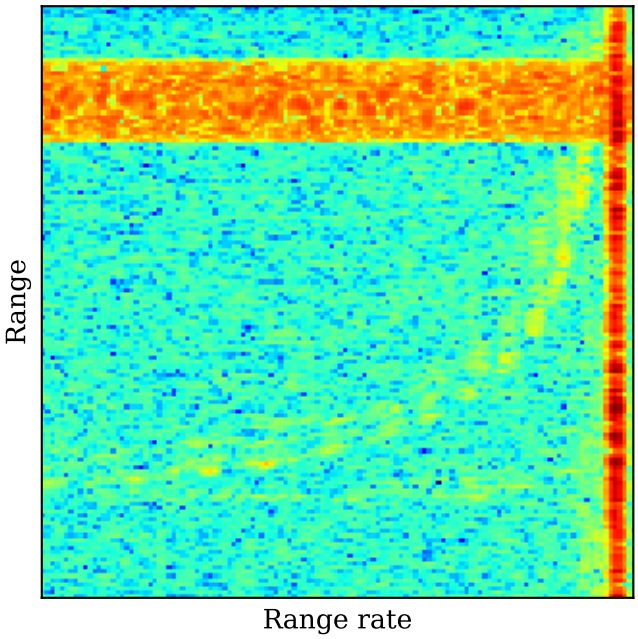

Figure 2.11: Noise jamming with range-bin masking. To increase the noise levels, the jamming is restricted to a limited range interval. The jamming can mask target echoes within this range span.

2.2.2 Digital radio frequency memory

Digital radio frequency memory, or DRFM, is a jamming technique in which the received pulse is recorded and stored digitally. The stored pulse can then be modified with frequency shifts, modulations and time delays before it is transmitted back one or multiple times to the enemy radar. DRFM can be used to create false targets in the enemy radar, or to create deceptive patterns in radar data that confuse the enemy's signal-processing methods. Because of its digital elements, the DRFM jammer is flexible and can be used in many ways. The DRFM techniques that are used in this thesis are covered in the following sections.

2.2.2.1 False targets

A DRFM jammer can be used to create false targets in the enemy radar. To do this, the EW system stores the received pulse and sends it back with some frequency shift and time delay. This creates an echo that looks like the true target but is located at a false location in the range-doppler map. This technique can be used to create multiple false targets in the enemy radar, and it is common to add some gain to the false targets to make them stronger than the true target. An example of false targets in a range-doppler map is shown in figure 2.12.

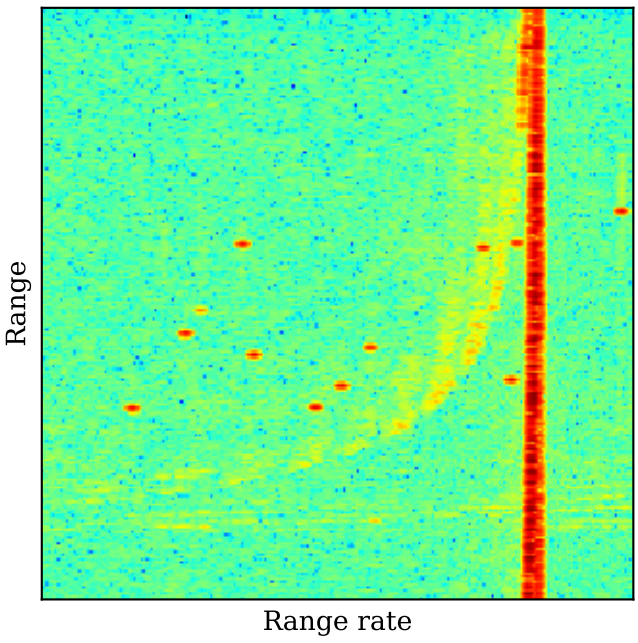

Figure 2.12: DRFM jamming with false targets. The enemy EW system transmits signals that look similar to true targets but at false ranges and range rates.

2.2.2.2 Reverse pulses and fast sweep

In addition to shifting the frequency and adding time delay, we can modify the DRFM pulse in further ways. One example is to reverse the transmitted pulse in

time domain. This makes the pulse not match the filter for pulse compression when it reaches the enemy radar receiver. So instead of appearing as a concentrated peak, it is extended in range.

Another way to modify the pulse is through *fast sweep*. With this technique the DRFM system speeds up the recorded pulse and repeats it an integer number of times (the *fast-sweep factor*) in the same time as the recorded pulse length. This technique is shown schematically in figure 2.13. Depending on the pulse compression of the enemy radar-receiver, the fast-sweep pulse appears as some pattern in the enemy radar-doppler map.

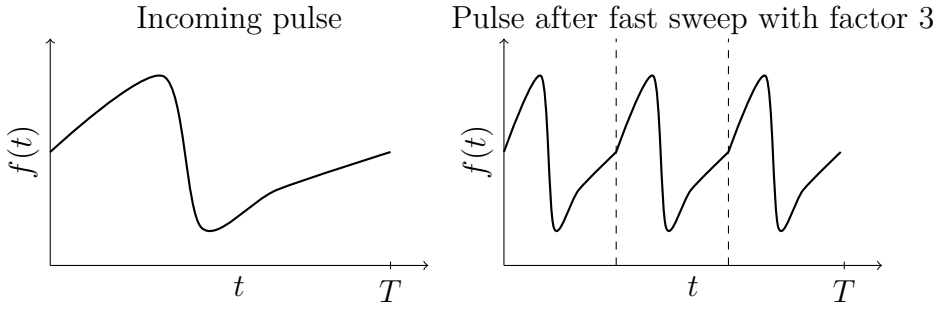

Figure 2.13: Schematic visualization of the fast-sweep technique. The incoming pulse has length T. The pulse is then squeezed in time so that it can be repeated an integer number of times during time T when it is transmitted by the EW system. The number of times the pulse is repeated is called the fast-sweep factor. In the figure the fast-sweep factor is 3.

Examples of DRFM jamming with reverse pulses and fast sweep in range-doppler maps are shown in figure 2.14 and 2.15.

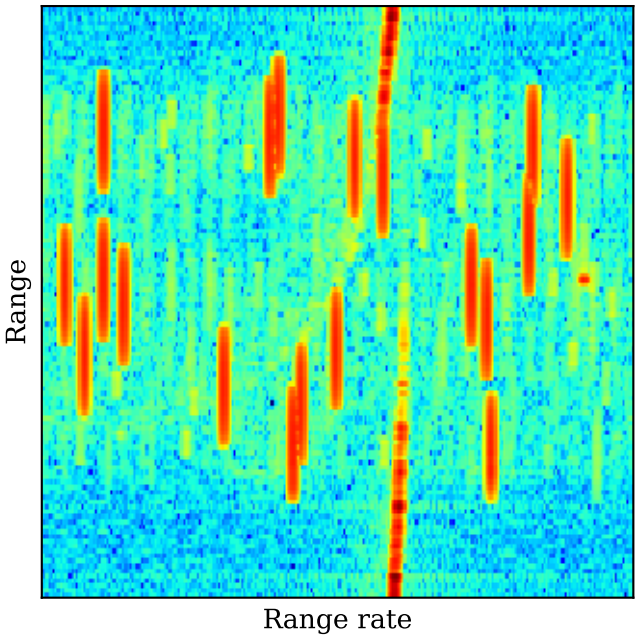

Figure 2.14: DRFM jamming with reverse pulses. The reverse pulses create false targets that are stretched in range.

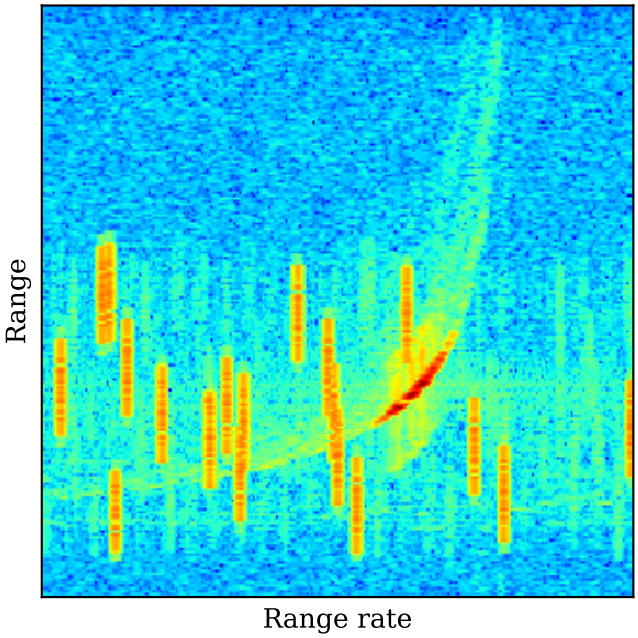

Figure 2.15: DRFM jamming with fast sweep. The fast-sweep pulses appear as vertical patterns in the range-doppler map.

2.2.2.3 Pulse repetition

A more simple form of DRFM jamming is to simply record the received pulse and repeat it a number of times with or without time delay. This creates a train of false targets at the same doppler frequency as the true target but at different ranges. The EW system can repeat the full pulse, which creates false targets that look similar to the true target. It can also repeat a fraction of the recorded pulse, which creates targets with some deformations because these pulses do not fully match the pulse compression of the receiving radar.

Examples of how pulse repetitions look in range-doppler maps are shown in figure 2.16 and 2.17.

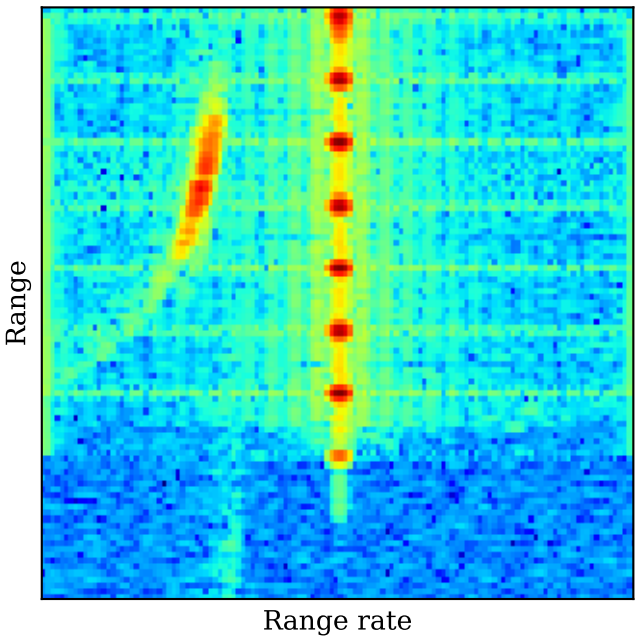

Figure 2.16: DRFM jamming with pulse repetition. Here, the full pulse is repeated, creating false targets at the same range rate as the true target. The false targets are separated uniformly in range with a distance determined by the time between transmitted jamming pulses.

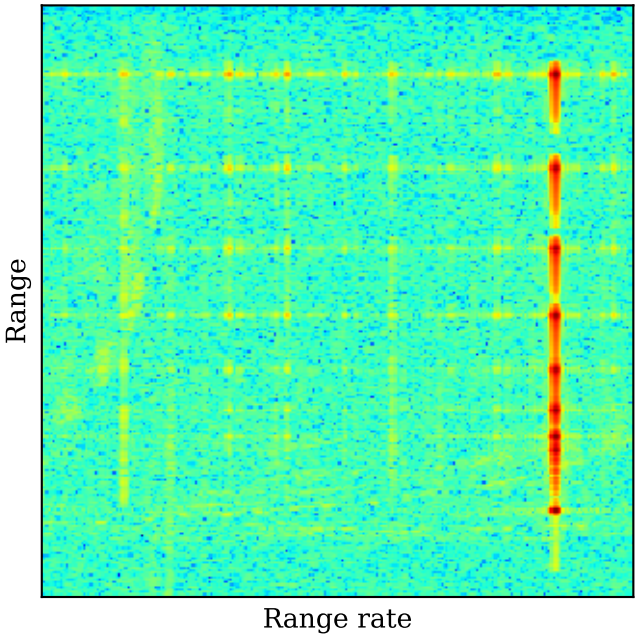

Figure 2.17: DRFM jamming with pulse repetition. Only a fraction of the received pulse is repeated by the EW system for this range-doppler map. The partial pulses do not fully match the pulse compression of the radar, creating some range distortions in the jamming.

2.2.2.4 Non-coherent pulse-repetition

Another simple form of DRFM jamming is a *non-coherent pulse-repetition*. In this technique, an incoming pulse is recorded and repeated with some time delay. The same pulse is then repeated with the same time delay after each of some specified number of subsequent pulses. After these repetitions, a new incoming pulse is recorded and repeated in the same way as before with the same time delay.

Because the same pulse is reused multiple times, the DRFM pulses are not fully coherent with the receiving radar. This creates a frequency shift in the DRFM pulses relative the true target. The result is that this DRFM technique creates a number of false targets with different frequency shifts, but with the same range shift. An example of DRFM jamming with non-coherent pulse-repetition is shown in figure 2.18.

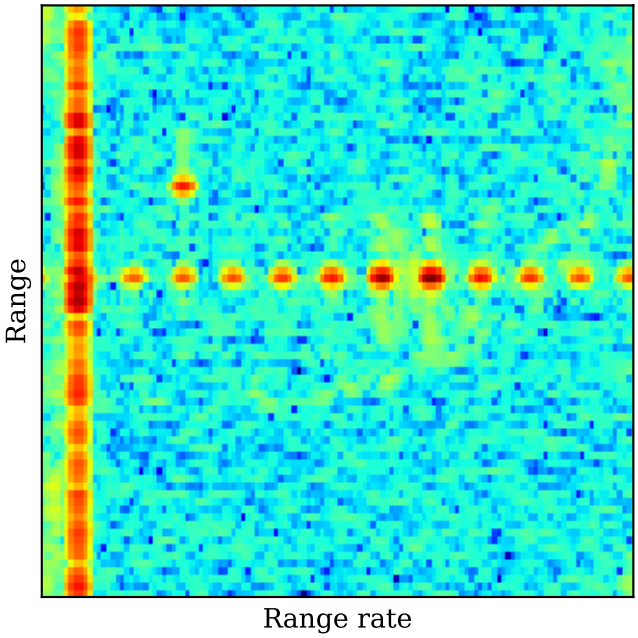

Figure 2.18: DRFM jamming with non-coherent pulse-repetitions. The DRFM pulses create a horizontal line of false targets. These targets have different frequency shifts as a result of the non-coherency with the receiving radar. The range shift is determined by the time delay of the jamming pulses.

2.3 Interference

Interference is phenomenon which can happen when two or more radars operate close to each other at the same radio frequency. Pulses from one radar are received by another radar and might clutter the range-doppler map. Interference is not strictly jamming because it generally happens undesirably between friendly sources, but it creates similar problems in the way that it obstructs desired target detection.

The interference patterns in the resulting range-doppler maps look different depending on the PRIs and pulse encodings of the interfering radars. A typical result from interference is that a number of vertical lines are created in the range-doppler map; an example of this is shown in figure 2.19.

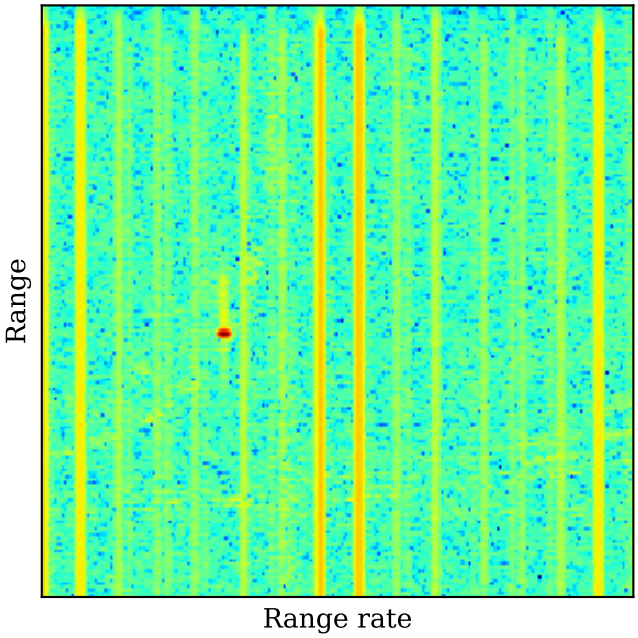

Figure 2.19: Interference in a range-doppler map. Interference can be interpreted as a type of unintentional jamming that can happen when friendly radars operate close to each other. Interference typically appears as vertical lines in the range-doppler map.

2.4 Electronic counter countermeasures

There is a natural demand for methods to counter ECM or to exploit its weaknesses. These methods are logically called *electronic counter-countermeasures* (ECCM).

Noise jamming in particular has a few traditional exploitation points. While it often obstructs measurements of range and velocity, it also acts as a beacon that exposes the direction to the jamming aircraft. In this way the jamming can be exploited to track the angle of the enemy aircraft.

By observing the direction of the noise jamming, there are also ways to passively approximate the range to the jamming aircraft. This can be done by e.g. measuring the angular rate of the jamming and comparing this to changes in the own-radar's velocity, or by employing triangulation with angle measurements of the jamming from different positions.

Another way to reduce the vulnerability to noise jamming is to use a *guard antenna* with low antenna-directivity. If the signal received by the guard antenna is stronger than the signal from the main antenna, we can assume that the radar is being jammed in one of its sidelobes.

There are no traditional ECCM techniques to counter DRFM jamming. However a trained radar-operator can often distinguish between true targets and false targets or deceptions from DRFM jamming because the DRFM signals often look unnatural or very strong compared to a true target.

Interference between allied aircraft can often be avoided by using transponder systems to shift the carrier frequencies of the interfering radars. There also exists techniques to remove the interference pulses from the received signals by measuring the PRI and pulse width of the interfering pulses.

An important first step for successful ECCM is to identify what kind of jamming the radar is exposed to. Because the techniques to counter jamming differ for different jamming techniques or interference, it is crucial to know what kind of jamming or interference that is being received by the radar. When the incoming radar signals are correctly classified as a certain type of jamming or interference, it becomes easier to decide any further actions to take.

The demand for correctly classified radar data in the field of ECCM is the motivation for this thesis. The next chapter covers theory on machine learning with neural networks, in particular image classifications with convolutional neural networks. These techniques are in subsequent chapters used to classify radar data.

3

Neural networks

This chapter presents theory on machine learning and neural networks, in particular convolutional neural networks. The chapter begins by giving some background on machine learning and providing fundamental theory on neural networks. It then goes on to cover elements of convolutional neural networks in more detail, together with central concepts regarding neural networks such as back propagation, optimizers, data splits, and regularization.

3.1 Machine learning and supervised learning

Machine learning is a group of statistical methods that are used to automate some task based on a set of data, rather than by explicit instructions. Common tasks for machine-learning methods are regression and classification. These both fall under the category of *supervised learning*, where data has both input and output.

The common procedure for a supervised learning-algorithm is to build some form of function for mapping new input to corresponding output based on a set of *training data* where both the input and output are known. Data for which both the input and output are known are often referred to as *labeled data* [9].

3.1.1 Model parameters and hyperparameters

The process of training usually involves setting the internal *model parameters* using some statistical or iterative technique. The model parameters are what determine the "skill" of the model: the model makes predictions based on the values of its model parameters. The model parameters might need to be initialized manually but are generally not further modified by the practitioner.

In contrast to the model parameters we have the *hyperparameters*. The hyperparameters are set prior to training and are part of the model design. The values of the hyperparameters are not modified by the training process and optimal values for the hyperparameters are generally not known. The hyperparameters are thus typically set by the practitioner based on prior experience, trial and error, or by copying values from research [9].

3.1.2 Overfitting

Supervised learning is in various degrees subject to the risk of *overfitting*. Overfitting can happen when the algorithm tries to learn a function which is too complex or when there is noise in the output labels. What often happens in the case of overfitting is that the algorithm memorizes the training examples instead of generalizing and identifying patterns and trends, leading to errors when producing output labels for new unseen input examples. Techniques that aim to avoid or reduce overfitting are often referred to as *regularization* [9].

3.2 Basic elements of neural networks

One of the current most popular methods for supervised learning is *artificial neural networks*, which will be referred to throughout this thesis as simply *neural networks*.

Neural networks consist of a set of nodes or neurons, divided between a set of layers. The layers are ordered and the neurons between two adjacent layers are connected by a set of scalar *weights*. The first layer is the *input layer* and the last layer is the *output layer*. The layers in between are called *hidden layers*. The number of neurons in the different layers and the number of layers are hyperparameters that are set by the practitioner. Figure 3.1 shows a schematic visualization of a neural network with one hidden layer.

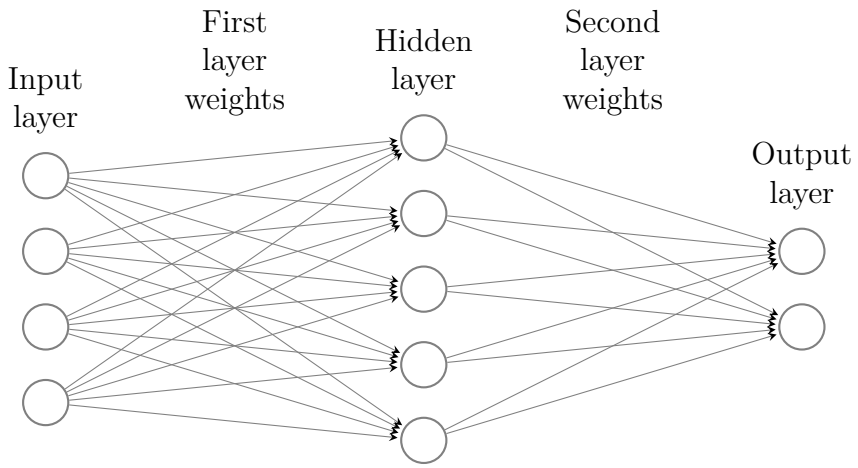

Figure 3.1: A neural network with four input neurons, one hidden layer with five neurons and two output neurons. Each neuron in one layer is connected to every neuron in the next layer by a set of weights: one weight for each connection.

Each neuron has an input and an output. The input is the weighted sum of the output from the neurons in the previous layer. The output is this weighted sum fed through some non-linear activation function $g(x)$. The forward propagation from layer i with n neurons to layer j with m neurons can be defined by

$$\mathbf{o}_j = g\left(\mathbf{w}_{ij}\mathbf{o}_i + \mathbf{b}_{ij}\right). \tag{3.1}$$

$\mathbf{o}_j$ is a $1 \times m$-vector with the outputs of the neurons in layer j. Likewise, $\mathbf{o}_i$ is containing the outputs of layer i in a $1 \times n$-vector. The matrix $\mathbf{w}_{ij}$ is of size $m \times n$ and contains the weights between layer i and j. $\mathbf{b}_{ij}$ is of size $1 \times n$ and contains a set of *biases* that are used to shift the activation function g which is applied element wise to the resulting $1 \times m$-vector. The computational step consisting of forward propagation from one layer of neurons to the following layer of neurons is often referred to as a *fully-connected layer*.

The number of neurons in the input layer is generally the same as the number of features of the input data. Likewise, the number of output neurons is often the same as the number of output features, or the number of classes in a classification problem.

The features of the input data often have different ranges or scales, leading to some features being dominant in the weighted sums of the forward propagation. It is thus standard procedure to normalize the input in some way before feeding it to the network [9].

3.2.1 Activation function

Using a non-linear activation-function is what allows a neural network to map non-linear relationships between the input and the output. Historically, functions like the sigmoid or the hyperbolic tangent have been common in neural networks. However in modern applications, these are often replaced by the *rectified linear unit* (ReLU) [9]:

$$f(x) = \max(0, x). \tag{3.2}$$

The ReLU activation offers great computational simplicity which is an important aspect for the performance of large neural-networks. The ReLU function is shown in figure 3.2.

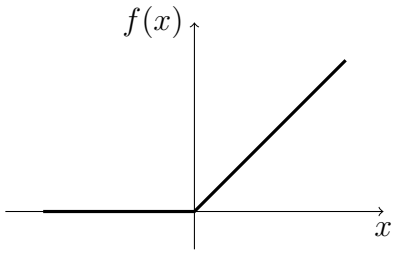

Figure 3.2: The rectified linear unit (ReLU): $f(x) = \max(0, x)$. This is the most common activation function in neural networks today. The main reason is the function's computational simplicity, which is important to reduce the computational heaviness of large-scale neural-network applications.

The activation function for the output neurons is in general different from the activation function for the other layers. In classification problems where the number of output neurons correspond to the number of classes, it is common to use *softmax* as the final activation. The softmax function takes a vector of k real numbers and normalizes it to a pseudo probability-distribution of k different probabilities in the

interval (0,1) that together sum to unity. If i_l are the input values of the output neurons where $l = 1, \ldots, k$, the softmax activation is given by

$$\text{softmax}(i_l) = \frac{\exp i_l}{\sum \exp i_m}. \tag{3.3}$$

The class corresponding to the neuron with the highest output is normally chosen as the output class [9].

3.3 Convolutional neural networks

Convolutional neural networks is a type of neural network designed for spatially dependent data. Typical types of data suited for CNNs are sound, text, and images. Image classification in particular have made great improvements in recent years with the help of CNNs; CNNs have been the dominant technique for top scores in the benchmark image classification challenge ILSVRC [5].

The main component of a CNN is the convolutional layer. There is at least one convolutional layer in a CNN but typically more. The convolutional layer is followed by an activation function. In addition to the convolutional layer, a CNN can contain e.g. pooling layers and fully-connected layers. In the case of image classification, the input to the CNN is an image and the output is a class.

3.3.1 Convolutional layer

The convolutional layer is the central component of a CNN. The convolutional layer takes an input tensor with a width, height and depth and convolves it with one or more filters. The depth of the filters is the same as the depth of the input tensor but the width and height of the filter are hyperparameters and determine the *kernel size*. The elements of the filters are real numbers and are the weights of a CNN.

The convolutional part consists of sliding the filters over the input tensor, taking the dot product of the filter with the different segments of the input tensor. The result of the dot product is placed at the corresponding position in the output tensor. A bias term specific to the filter is typically also added to the dot product:

$$\text{segment of input tensor} \cdot \text{filter} + \text{bias term}. \tag{3.4}$$

The output of a convolutional layer is sometimes referred to as a *feature map*. The convolving operation is illustrated in figure 3.3, however here without including any bias term.

The depth of the output tensor equals the number of filters in the convolutional layer. The width and height are determined by the kernel size of the filters, the amount of zero padding to the input tensor, and the *stride*. Zero padding can be used by adding zeros to the edges of the input tensor. This is done in order to enable the filters to be applied further out on the edges and corners of the input tensor. The stride on the other hand defines how far the filter is moved in each sliding step, both in horizontal and vertical direction. The use of zero padding and stride are shown in figure 3.4 [9].

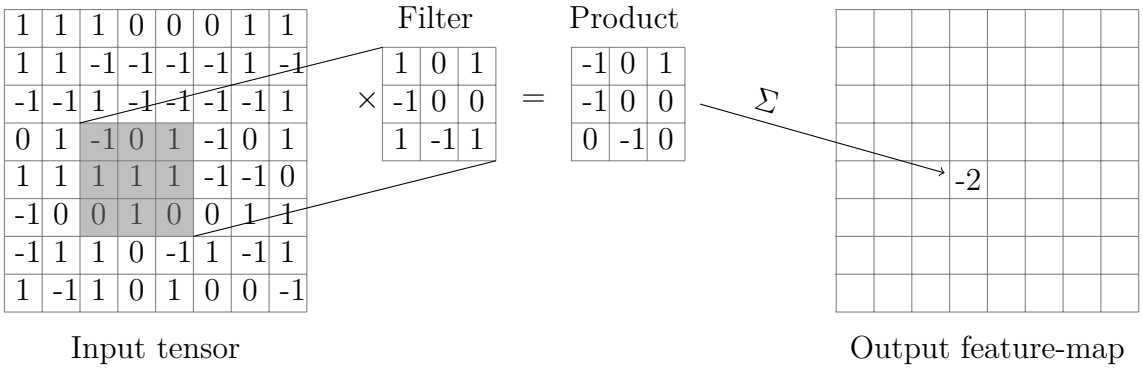

Figure 3.3: The convolutional step of a convolutional neural network. We take a filter of some size (3×3 in the figure). We "slide" the filter over the whole input tensor and take the dot product between the filter and each segment of the input tensor. The result of the dot product at each position is placed at the corresponding position in the output tensor.

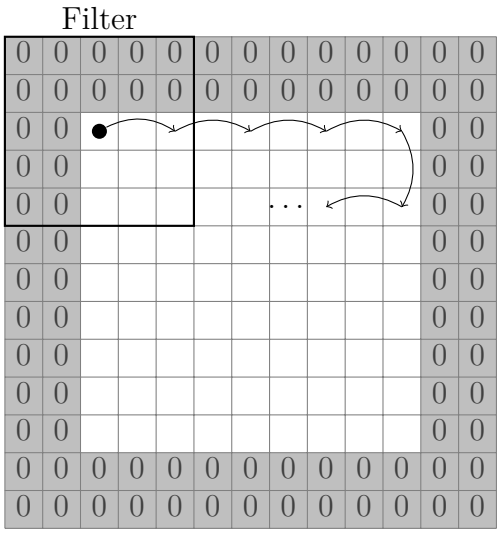

Figure 3.4: An input tensor to a convolutional layer of width and height 9×9. The input is padded with two layers of zeroes. The filter has size 5×5. The arrows denote how the filter is moved in each sliding step. The step distance is a parameter called *stride* and has a horizontal and a vertical component. In the figure the stride is 2 in both directions. The size of the filter, the amount of zero padding and the stride determine how the width and height of the tensor are changed when passed through the convolutional layer. In the figure the width and height of the output are 5×5.

3.3.2 Pooling layers

Pooling layers are common in CNNs. Their main function is to reduce dimensionality of the tensors passed through the network, making the computational loads lighter.

The most common pooling layer is the *max-pooling layer*. This layer applies a max filter to regions of the input tensor and places the results at the corresponding positions in the output tensor. The intuition behind the max-pooling layer is that it extracts the "extreme" features of the input tensor, which are likely to play an important role in the classification of the data instance.

Common practice is to use max-pooling layers with non-overlapping filters of size 2×2. This reduces the width and height of the input tensor by half. The max-pooling layer is illustrated in figure 3.5. Note that this kind of pooling layer is only applied in the width- and height dimensions; the depth of the input tensor is preserved.

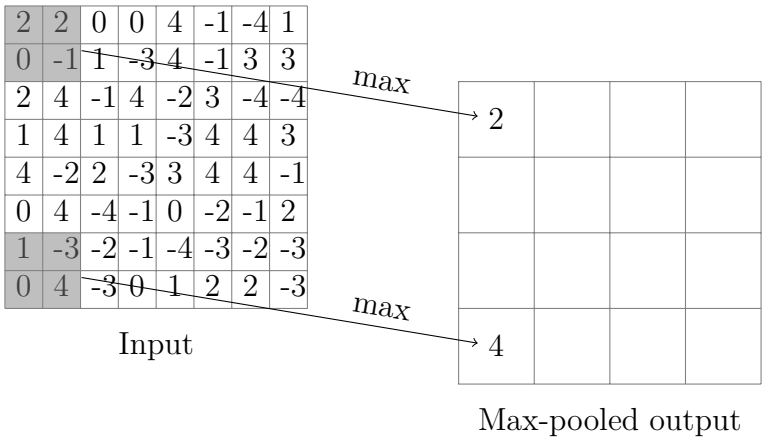

Figure 3.5: The max-pooling layer of a convolutional neural network. A max filter is applied to regions of the input tensor. These regions are usually non-overlapping but cover the whole input. The result of the max filter is placed at the corresponding position in the output tensor. In the figure, the pooling regions are of size 2×2 and the regions are non-overlapping. The result is that an input of size 8×8 is down sampled to size 4×4.

The pooling layer has no trainable parameters. How many pooling layers that are used and where they are placed can vary from different applications. It is however common to use max-pooling after every two or three convolutional layers [10].

3.3.3 Classification layer

At the end of a CNN there needs to be a layer that makes the final classification. This can be done in many ways, but a current popular way is to use *global average pooling* on the output from the final convolutional layer into a single final fully-connected layer [11].

The global-average-pooling layer takes the average of every width-height plane in the input tensor, reducing an input size of $h \times w \times d$ to $1 \times 1 \times d$. The idea behind the global-average-pooling layer is that it detects the presence of features regardless of their position in the width-height plane. The motivation for this is that image classification typically is a translation-invariant problem: the objects or features that make an image belong to a certain class can be at any position in the image. Furthermore, the global-average-pooling has the advantage of reducing the computational complexity of the network and has no trainable parameters or hyperparameters [12].

The global-average-pooling layer is often followed by a fully-connected layer with the number of output neurons corresponding to the number of classes in the classification problem. The output-neurons of this layer are fed into a softmax activation-function that gives the final output-scores. The argmax of these scores are typically selected as the output class [11].

The structure of the classification layer described above is shown in figure 3.6.

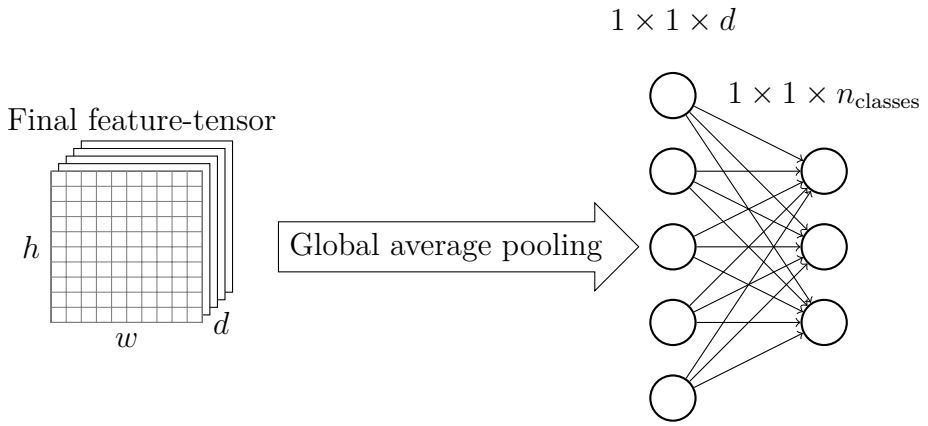

Figure 3.6: A common structure for a classification layer in a CNN. Global average pooling is applied to the final feature-tensor of size $w \times h \times d$. This is done by taking the average of each $w \times h$-matrix making the result a one-dimensional vector of size d. The elements of this one-dimensional vector are then used as nodes in a final fully connected layer where the number of output nodes correspond to the number of classes in the classification problem.

3.3.4 Architecture summary

A CNN often consists of a series of convolutional layers, activation functions, and pooling layers, followed by a classification layer. An example CNN is given in figure 3.7. This is a CNN for image classification that takes input images of size $64 \times 64 \times 1$ (gray scale). It has three convolutional layers each with 16 filters of size 5×5. All the convolutional layers are followed by ReLU activations. It has one max-pooling layer with filter size 2×2. The final ReLU activation is followed by a global-average-pooling layer into a fully-connected layer with four output nodes corresponding to four different classes. The final activation function is softmax.

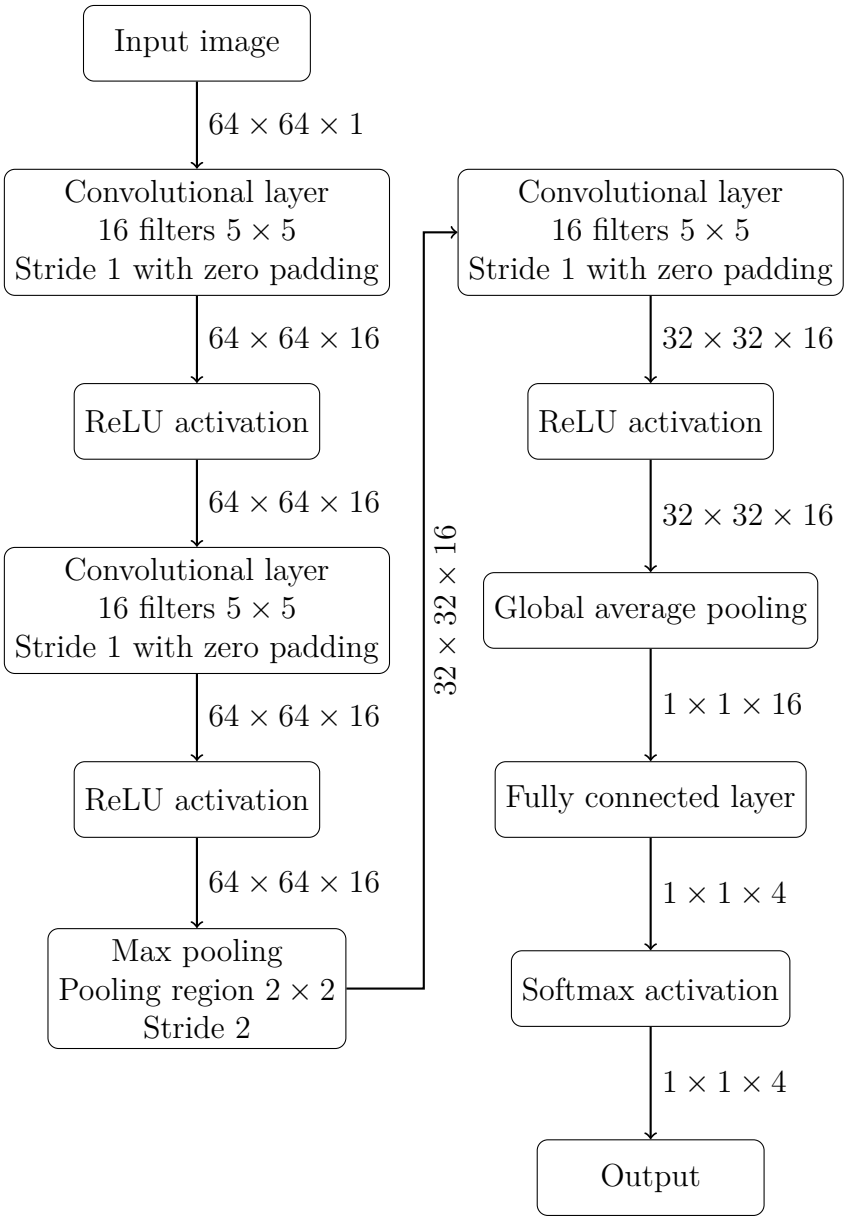

Figure 3.7: An example of a CNN architecture. The dimensions of the inputs and outputs of each layer are denoted on the arrows. The input to the network is a gray-scale image of size 64×64. The network has three convolutional layers each with 16 filters of size 5×5. The convolutional layers use unity strides and zero padding so that the width and height of the input is preserved. Each convolutional layer is followed by a ReLU activation layer. The second convolutional layer is followed by a max-pooling layer that down samples the width and height from 64×64 to 32×32. The final ReLU activation is followed by a global-average-pooling layer. The output of this layer is fed to a fully-connected layer that outputs the final output neurons. In this figure we have four output neurons each representing one class. The output of the fully connected layer is fed to a softmax activation-function that gives us the final output-scores for each class.

3.4 Setting the network parameters

The output of a neural network is a function of its internal parameters: primarily by the weights and biases. This section covers the training process that adjusts these parameters so that the network makes correct predictions.

The most common way to train a neural network is to make the training process into an optimization problem. This is done by introducing a *loss function*. The loss function acts as measure of the network performance: a lower loss indicates a better performance. The loss is a function of some labeled data set x containing training examples, and the network parameters θ:

$$L(\theta, x). \tag{3.5}$$

The optimization problem thus becomes the problem of finding the set of network parameters θ that minimizes $L(\theta, x)$:

$$\underset{\theta}{\operatorname{argmin}}\ L(\theta, x). \tag{3.6}$$

This optimization problem can be solved by some iterative optimization algorithm. Common techniques are various methods based on gradient-descent optimization [9].

3.4.1 Loss function: categorical cross-entropy

There are different loss functions suitable for different types of problems. This thesis focuses on categorical classifications and the most common loss function for categorical classification-problems is the *categorical cross-entropy*. Categorical cross-entropy can be used to measure the dissimilarity between a true probability distribution and an estimated probability distribution. Greater cross entropy means larger differences between the two distributions.

With a discrete true distribution $p(x)$ and an estimation of this distribution $q(x)$, the cross entropy L_{ce} between these distributions is

$$L_{\text{ce}} = -\sum_{\text{all } x} p(x) \log q(x). \tag{3.7}$$

In a context of supervised classification, p is zero for all classes except the true class for which it is one. If we denote the network output for the true class q_{true} the cross entropy becomes

$$L_{\text{ce}} = -\log q_{\text{true}}. \tag{3.8}$$

We can see that the cross entropy is zero if the output for the true class is one, and it goes toward infinity if the output approaches zero.

However, we generally have more than one sample in supervised learning. The loss is typically calculated as an average of the cross entropies across all the training samples:

$$L_{\text{ce}} = -\frac{1}{N} \sum_{i=1}^{N} \log q_{\text{true},i} \tag{3.9}$$

where N is the number of training samples and $q_{\text{true},i}$ is the output for the true class for training sample i [9].

3.4.2 Training using back propagation

One of the key techniques for efficient training of a neural network is *back propagation*. Back propagation provides a computationally efficient way to calculate the partial derivatives of the loss with respect to each network parameter. The main idea behind back propagation is to apply the chain rule step wise, starting at the loss function propagating backwards through the network. The back-propagation algorithm is not covered in detail in this thesis; an extensive description of back propagation is provided by Goodfellow, Bengio and Courville in their book *Deep Learning* [9].

To obtain the gradients of the network parameters, we feed some of the training data through the network to evaluate the loss. It is generally not possible to feed all training samples through the network simultaneously because of memory limitations. Instead, the training data is divided into *batches* with some *batch size* which denotes the number of training samples that are fed through the network for one set of parameter updates.

When one batch is fed through the network, the loss is evaluated on this batch and the gradients are calculated with back propagation. The network parameters are then updated according to some optimizer to lower the loss. After one update, the next batch is fed through the network and the parameters are updated once more. This procedure is repeated until some stopping condition is reached. It is common to set a total number of *epochs* as stopping condition. This number denotes how many times *all* training samples are fed through the network.

Prior to the iterative update process, the weights and biases of the network need to be initialized. This is typically done by sampling from some random distribution, e.g. a Gaussian or uniform distribution, or by setting parameters to zeros. The training process is summarized in the list below:

1. *Initialize the network parameters with some random distribution.*
2. *Feed one batch of training data through the network.*
3. *Evaluate the loss on the batch based on the network output.*
4. *Calculate the gradients of the loss with back propagation.*
5. *Update the network parameters according to an update rule from some optimizer.*
6. *Repeat step 2 to 5 until a set number of epochs is reached, or when some other stopping condition is satisfied.*

3.4.2.1 Stochastic gradient descent

The most fundamental optimization-method for neural network is *stochastic gradient-descent* (SGD). Stochastic gradient-descent works just like normal gradient descent with the exception that is uses an approximation of the gradient, based on the training batches that are described above. When the batch size equals the full training set, stochastic gradient descent is the same as regular gradient descent. The update

step given by SGD is

$$\theta_t \leftarrow \theta_{t-1} - \eta \nabla_{\theta_{t-1}} L(\theta, x_{\text{batch}}) \tag{3.10}$$

where θ are the network parameters, η is the scalar learning rate, $\nabla_{\theta_{t-1}} L(\theta, x_{\text{batch}})$ are the gradients of the loss (evaluated at θ_{t-1}), and x_{batch} is the current training batch. The learning rate η is a hyperparameter.

There are many variants of SGD and other optimizers that are inspired or based on SGD. One variant that is recommended in the Stanford course *CS231n: Convolutional Neural Networks for Visual Recognition* by Andrej Karpathy [13] is SGD + Nesterov momentum. The update steps in this optimizer are:

$$\begin{aligned} \tilde{\theta} &\leftarrow \theta_{t-1} + \alpha v \\ g &\leftarrow \nabla_{\tilde{\theta}} L(\theta, x_{\text{batch}}) \\ v &\leftarrow \alpha v - \eta g \\ \theta_t &\leftarrow \theta_{t-1} + v. \end{aligned} \tag{3.11}$$

This optimizer uses a moment vector v and a scalar momentum-parameter α that help the optimizer avoid getting stuck in local minima. It also employs an intermediate update $\tilde{\theta}$ to evaluate the gradient after the velocity is applied. The velocity v needs to be initialized and α is a hyperparameter [9].

It is common to use SGD optimizers with some form of *decay* applied to the learning rate. This means that the learning rate is lowered by some decay parameter γ as a function of the training iteration t. One way to do this is though stepwise decay:

$$\eta = \frac{\eta_0}{1 + \gamma t} \tag{3.12}$$

where η_0 is the initial learning rate [14]. The motivation is that a larger learning rate at the beginning of the training process quickly gives a rough estimation of a good parameter-set. A smaller learning rate later in the training process can then fine tune these parameters with smaller update-steps.

3.4.2.2 Adam optimizer

One of the current most popular and optimizers for large neural-networks is the *Adam optimizer*. Adam is different from SGD in that it maintains different learning rates for different parameters. These learning rates are in addition adaptive and are changed as the learning progresses depending on the results. The update rules in

the Adam optimizer are

$$\begin{aligned} g &\leftarrow \nabla_{\theta_{t-1}} L(\theta, x_{\text{batch}}) \\ m &\leftarrow \beta_1 m + (1-\beta_1) g \\ v &\leftarrow \beta_2 v + (1-\beta_2) g^2 \\ \hat{m} &\leftarrow \frac{m}{(1-\beta_1^t)} \\ \hat{v} &\leftarrow \frac{v}{(1-\beta_2^t)} \\ \theta_t &\leftarrow \theta_{t-1} - \frac{\eta \hat{m}}{\sqrt{\hat{v}} + \epsilon}. \end{aligned} \tag{3.13}$$

This optimizer has the scalar hyperparameters η, β_1, β_2 and ϵ. η is the learning rate, β_1 and β_2 are exponential-decay parameters in $[0, 1)$ for the moment vectors. ϵ is a small positive number to avoid division by zero. It uses moment vectors m and v that are initialized to zero [15].

The original paper that introduces the Adam optimizer [15] recommends the default values for the hyperparameters

$$\begin{aligned} \eta &= 0.001 \\ \beta_1 &= 0.9 \\ \beta_2 &= 0.999 \\ \epsilon &= 10^{-8}. \end{aligned} \tag{3.14}$$

In studies where different optimizers are compared, Adam with its default parameters has been shown to be a good choice for most large neural-network applications [16, 13].

3.5 Training data, validation data, and test data

To properly evaluate the performance of a supervised learning-algorithm, we generally need to divide the available data into different sets. Just by looking at the performance on the training data, it is impossible to know if there has been any overfitting during training.

To get around this problem, we introduce a *validation set* and a *training set*. The training set contains the data samples that are used by the training algorithm to update the model parameters. The validation set is in contrast not seen by the training algorithm; it is only used for monitoring the performance of the model. If the performance is high on the training set but not on the validation set, there is a high chance that there has been overfitting during training. On the other hand, if the performance is high on both sets, it is a good indication that the model has reached a good generalization that works on unseen data.

Performance on the validation set is a good measure for setting hyperparameters and determining the model architecture. However, if all hyperparameter selections

and the model architecture are based on the same validation set, there is a risk that the model overfits also on the validation data. Thus we need to introduce a third data set: the *test set*.

The test set is used for a final performance evaluation when all hyperparameters have been set and the model architecture is finalized. The test set should not be used in any way by the training algorithm or to make any decisions regarding hyperparameters or model architecture. It should only be used to give an honest performance measure of the final model.

How the available data are divided into these three different data sets can vary, and depends on both the amount of available data and the nature of the data. The training set is however generally the largest of the three. Typical ranges for percentages of the total data allocated for validation- and test sets seem to be around 10-20%, while the rest of the data are put in the training set [9].

3.6 Regularization in a convolutional neural network

Techniques with the aim to reduce or prevent overfitting are known as regularization. A CNN has, just as any supervised learning-algorithm, the risk to overfit on training data and considerations need to be put in how to reduce and avoid this risk.

A common cause of overfitting is that the model is too complex for the data it is being used for. In a very deep network it might be easier for the model to memorize the training examples instead of mapping simple relationships between the input and output. This can also happen if the training set is too small.

A first step to reduce overfitting is thus to use as much training data as possible, while keeping the model as simple as possible. The model complexity can then be increased by e.g. adding more convolutional layers until the performance on the validation set is no longer increasing [9].

3.6.1 Weight decay

Weight decay is a common technique for regularization in neural networks. Weight decay puts a penalty on large weights by adding the L^2-norm of the weights (and sometimes biases) to the loss function, scaled by a weight-decay parameter λ:

$$\tilde{L} = L + \frac{\lambda}{2}|\theta|^2. \tag{3.15}$$

This reduces the risk of some parameters becoming very large, making the remaining parameters underutilized [9].

Weight decay in CNNs is used with success both by Krizhevsky in [4] and by Simonyan and Zisserman in [10]. Both papers reported the use of weight-decay parameter $\lambda = 5 \cdot 10^{-4}$.

Typical values for λ in fully-connected layers range between 0 and 0.1 and are often a negative power of 10, e.g. $\lambda = 0.01$ or $\lambda = 0.001$ [17].

3.6.2 Monitoring validation loss

The most important way to detect and prevent overfitting is to monitor the performance on the validation set. The performance on the validation set is typically monitored during the entire training process. A sign of overfitting is that the loss on the training set is decreasing but the loss on the validation set is increasing. This scenario is shown in figure 3.8. If this happens, we need to introduce more regularization to the model e.g. by adding weight decay, increasing the amount of training data, or making the network architecture more simple.

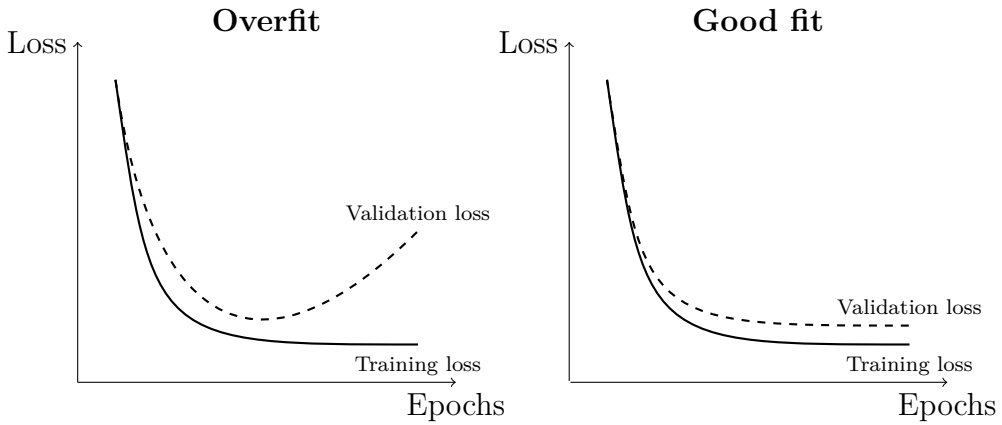

Figure 3.8: The left panel shows a case of overfitting during training. The training loss is decreasing but the validation loss has a minimum. After the minimum in the validation loss the model is no longer generalizing but instead memorizing the training data. The right panel shows a case of a good fit. Both the training and validation loss are decreasing until they stabilize around the same point. It is normal that the validation loss is slightly higher than the training loss even in a good fit.

It is possible that the model has reached a good generalization before it starts overfitting. In that case we simply want to ignore the training progress after the point of overfitting. This can be done by continuously saving the model parameters each time a new minimum for the validation loss is reached during training. The final set of parameters is then the one that gave the lowest validation loss. By employing this technique, sometimes referred to as *early stopping*, we can allow some degree overfitting as long as a good generalization is found at some point during training [9].

3.7 Convolutional neural networks for radar data

This chapter has reviewed the fundamentals of machine learning, neural networks and in particular convolutional neural networks. The goal of this thesis is to create a

machine-learning model that can identify and classify jamming in radar data. This is done by using range-doppler maps as "image inputs" to a CNN. The CNN is trained to classify the range-doppler maps into classes that correspond to different types of jamming, in addition to a class which represents non-jammed radar data.

Most of the techniques described in this chapter are used to some degree in the remaining chapters to create this classification model. Especially chapter 5 describes how hyperparameters and CNN architecture are chosen for the model used in this thesis. The next chapter covers how synthetic radar data are generated with the purpose of creating a labeled data-set for this classification task.

4

Generating synthetic radar-data

This chapter covers the process of providing necessary labeled radar-data in order to train a CNN. The first two sections discusses general aspects regarding real vs. simulated data, and the amount of data needed. The chapter then continues with a brief description of the software used to generate radar data in this thesis. It then goes on to cover the different data classes that are generated by the software, and how parameters for these data classes are set. The theory behind many of the radar concepts in this chapter is described in chapter 2.

4.1 Obtaining training data

Having sufficient labeled data of high quality is a crucial element of any supervised learning-algorithm. The training data are what makes it possible for the model to make predictions on unseen data. The amount of data is generally important to reduce overfitting and to cover the full distribution of possible data instances. It is also important to use data with little noise: if many of the labels in the training set are incorrect, the model will learn errors.

Obtaining labeled data for machine-learning tasks is however not a trivial task. A common situation is that there are much real data available, but these data are *unlabeled*. To use these data they must first be labeled in some way. Typically this needs to be done manually, which quickly becomes time consuming.

An alternative is to generate labeled synthetic data that are supposed to resemble real data as much as possible. When compared to manually labeling a large data set, this is often a quick and easy way to obtain large amounts of labeled data. A risk when using synthetic data is that the synthetic data might not fully represent all features of their real counterpart, as a result of using simplified models. This can lead to errors if a model trained on simulated data is used to make predictions on real data.

Real and synthetic data can also be used in combination. One way to do this is to first train the model on a large set on synthetic data. The resulting model is then *fine tuned* on a smaller set of labeled real data. The fine-tuning part is basically a second training phase with another data set, possibly with a smaller learning rate. The intuition behind this method is that the first training phase can teach the model the general features of the data that are possible to represent with simulation

models. The fine-tuning phase can then make the model learn features of the real data that are not represented in the synthetic data [9].

The work in this thesis uses synthetic data exclusively. The reason for this is mainly the security classifications covering real radar-data from combat aircraft.

4.2 The amount of training data

The amount of labeled data needed to obtain good results from a classification model varies from different problems. We generally need the same amount of training instances in all different classes. If the training set is heavily unbalanced, the model might learn to always predict the class with the most training instances because this is an easy way to achieve a low loss.

How many instances that are needed for each class depends on the data distributions for the different classes. More complex distributions require more training examples to cover all features and properties belonging to the different classes. It is generally never a disadvantage to have more training data than needed, other than some increased computational complexity. The benchmark data set for image classifications, ImageNet, contains several hundred instances per class [5].

For this thesis, 1200 data instances per class are generated. This number should be large enough to give a good representation of each class, without requiring unmanageable computational times to be generated.

All range-doppler maps in the figures of this thesis are instances from the data set generated by the methods described in this chapter.

4.3 Simulation software

The software used to generate synthetic data for this thesis is a MATLAB-program written at Saab Surveillance. This software is designed to simulate radar data from an airborne radar. The software takes user-defined scenarios and generates radar data based on a set of given parameters. The software has features to generate target returns, ground clutter, noise jamming, interference and DRFM jamming. However, the models used in this software do not include effects from e.g. rain and engine-induced vibrations.

For this thesis, the software is used to generate range-doppler maps of different classes. All of these classes correspond to different types of jamming except one class which contains non-jammed range-doppler maps. Each class has its own generative model and a set of parameters. Some of these parameters are randomized for each data instance. This way, a data set can be created where all data instances look different, but all instances of a certain class share some basic features that are unique for its class.

The main limitation of this software when used for this thesis are its features for DRFM jamming. Only a few basic DRFM techniques were implemented in the latest

version of this software. Some additional DRFM techniques had to be implemented specifically for this thesis work and some had to be modified.

4.4 Data classes and parameter selections

The data set used in this thesis contains four different classes. These classes are also the classes that we want the machine-learning model to identify. These classes are:

- *No jamming*
- *Interference*
- *Noise jamming*
- *DRFM jamming*

All of these classes have parameters that are randomized for each data instance, and parameters that are constant for all data instances in the class. Some parameters are class specific and some are common for all classes. The parameters common for all data classes are given in table 4.1. The values in this table are chosen to be reasonably realistic for typical conditions of air combat.

Table 4.1: Parameters that are common for all data classes. Parameters that are constant have a value in the "constant" column. The parameters that are randomized have values in the columns for min- and max value. The randomized parameters are uniformly randomized within the specified ranges for each data instance.

Parameter	**Min value**	**Max value**	**Constant value**
Own altitude [m]			3000
Own velocity [m/s] (parallel to the ground)			250
PRI [μs]	40	200	
Number of pulses	100	200	
Elevation angle [degrees] (radar main-beam)	-10	20	
Probability of target presence			50%
Distance to target [m]	300	40 000	
Range rate of target [m/s] (relative own aircraft)	-1000	100	
RCS of target [m^2]	3	20	
Linear frequency modulation [MHz]			0.04
Circular frequency modulation [MHz]			1.1
Duty factor			0.1
Radio frequency [GHz]			9.7

The presence of a target in each data instance is random with the probability 50% (except for in the class DRFM). A maximum of *one* true target is placed in each range-doppler map. Any target is placed in the direction of the own-radar main-beam with randomized, range, range rate and RCS.

The elevation angle defines the direction in which the radar main-lobe is pointing. This parameter is the angle between the horizontal plane of the aircraft and the radar main-lobe, as shown in figure 4.1. The radar is however never angled to the left or right.

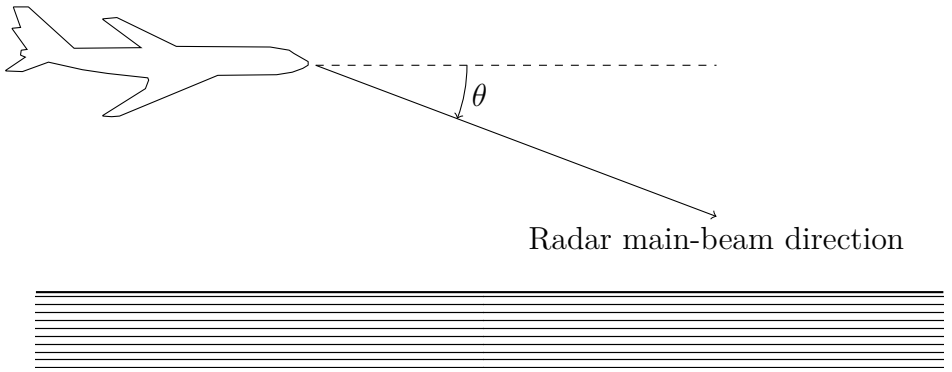

Figure 4.1: The elevation angle θ is the angle between the horizontal plane of the aircraft and the radar main-beam direction. It is positive if the radar is looking down at the ground and negative if the radar is looking up.

The window function used for the entire data set is the *Taylor window*, which is a popular window function for radar applications. We use a Taylor window with sidelobe suppression SL = -60 dB. A detailed description of the Taylor window is given in *Spotlight Synthetic Aperture Radar: Signal Processing Algorithms* by Carrara, Goodman and Majewski [7].

The pulse compression used by the simulated radar is through a combination of linear frequency-modulation and circular frequency-modulation. This encoding is defined by a linear modulation-frequency and a circular modulation-frequency. This pulse compression technique is covered in the book *Modern Radar Systems* by Meikle [18].

4.4.1 Class: No jamming

The class *no jamming* contains range-doppler maps with only ground clutter and targets. All the parameters are selected as in table 4.1. The ground clutter in these range-doppler maps look different depending on the randomized elevation angle and PRI. The targets are placed at random ranges and range rates in each range-doppler map.

4.4.2 Class: Interference

In the class *interference*, we add interference from another radar. The PRI of the interference is random but the pulse encoding of the interference, the gain of the interference (JNR), and the duty factor of the interference are constant. The parameters unique for the interference class are given in table 4.2.

Table 4.2: Parameters specific for the class interference.

Parameter	Min value	Max value	Constant value
Interference PRI [μs]	4	200	
Interference duty factor			0.3
Interference linear frequency modulation [MHz]			0.1
Interference circular frequency modulation [MHz]			2.0
JNR (interference) [dB]			30

4.4.3 Class: Digital radio frequency memory

In the DRFM class, the own radar is being jammed with some form of DRFM jamming. A true target is always present in this class. The DRFM class has six different sub categories that are described in the sections below. Each instance of the DRFM class is randomly placed in one of the sub categories with equal probabilities. These sub classes are however not visible to the learning algorithm; we want the CNN to classify all of these sub categories as simply DRFM. More detailed descriptions of the different DRFM types are given in chapter 2.

The only parameter that is common to all DRFM instances is the random linear gain of the DRFM signals (relative the target returns). This parameter is given in table 4.3.

Table 4.3: The range for the random linear gain which is common for all DRFM types.

Parameter	Min value	Max value	Constant value
DRFM linear gain	1	30	

4.4.3.1 False targets

The DRFM technique *false targets* places false targets that look like true targets in the receiving radar. The false targets are shifted in range and range-rate relative to

the true target. The shifts are randomized for each false target. Also the number of false targets is randomized. The parameters for this DRFM type are given in table 4.4.

Table 4.4: Parameters for the DRFM technique: false targets.

Parameter	Min value	Max value	Constant value
Number of false targets	10	20	
Range delay [m]	-5000	5000	
Range-rate shift [m/s]	-300	300	

4.4.3.2 Reverse pulse

The reverse-pulse technique flips the incoming pulse and sends it back a number of times with shifts in range and range rate. This technique works much like false targets with the addition of the pulse flip. The parameters for this technique are given in table 4.5.

Table 4.5: Parameters for the DRFM technique: reverse pulse.

Parameter	Min value	Max value	Constant value
Number of false targets	10	20	
Range delay [m]	-5000	5000	
Range-rate shift [m/s]	-300	300	

4.4.3.3 Fast sweep

The fast-sweep technique speeds up the incoming pulse and creates a new pulse with repeated copies of the incoming pulse. The number of repetitions is specified by the fast-sweep factor. The fast-sweep pulse is then sent back as false targets much like in the false-targets technique. The parameters for this technique are given in table 4.6.

Table 4.6: Parameters for the DRFM technique: fast sweep.

Parameter	Min value	Max value	Constant value
Number of false targets	10	20	
Range delay [m]	-5000	5000	
Range-rate shift [m/s]	-300	300	
Fast sweep factor	2	4	

4.4.3.4 Pulse repetition

The pulse-repetition technique records the incoming pulse and repeats it a number of times without any frequency shifts. The only parameter for this technique is the number of repetitions, which is given in table 4.7.

Table 4.7: Parameters for the DRFM technique: pulse repetition.

Parameter	Min value	Max value	Constant value
Number of pulse repetitions	5	20	

4.4.3.5 Partial pulse-repetition

The technique partial pulse-repetition works just like the pulse-repetition technique with the exception that only a fraction of the incoming pulse is recorded (starting from the beginning of the pulse). This fraction is an additional randomized parameter for this DRFM type. The parameters for this technique are given in table 4.8.

Table 4.8: Parameters for the DRFM technique: partial pulse-repetition.

Parameter	Min value	Max value	Constant value
Number of pulse repetitions	5	20	
Fraction of pulse recorded	0.2	0.5	

4.4.3.6 Non-coherent pulse-repetition

In non-coherent pulse-repetition, one pulse is recorded and repeated with some defined time delay after some number of subsequent pulses. Then a new pulse is

recorded and the process is repeated. This technique creates false targets with different frequency shifts but with the same range shift. The only parameter for this technique is the number of times one pulse is "reused". The range for this parameter is given in table 4.9. The time delay of the jamming pulses are uniformly randomized from zero to the PRI of the incoming pulses (but the same for all jamming pulses).

Table 4.9: Parameters for the DRFM technique: non-coherent pulse-repetition.

Parameter	Min value	Max value	Constant value
Reuse number	5	15	

4.4.4 Class: Noise jamming

In the noise-jamming class, the own radar is being jammed by noise jamming. This class includes both jamming with continuous noise and range-bin masking. The two types are selected for each data instance with equal probability. The parameters in this class are the jamming-to-noise ratio (JNR) and the length of the range interval that is jammed for range-bin masking. These parameters are given in table 4.10.

Table 4.10: Parameters for the noise-jamming class. Continuous noise and range-bin masking are selected with equal probability for each data instance.

Parameter	Min value	Max value	Constant value
JNR [dB]	30	50	
Jamming interval [m] (for range-bin masking)	200	1000	

4.5 Using synthetic data to train convolutional neural networks

The radar data generated by the methods described in this chapter are used as training data for a CNN in the following chapters; the range-doppler maps of the data set are used as input instances for the CNN. The next chapter covers the process of selecting network architecture and tuning hyperparameters. The subsequent chapter gives the classification results from the final model.

Note that the parameter values given in this chapter are chosen to somewhat resemble real-world scenarios, but the values can probably be adjusted further to make the data even more realistic. The classification results on this data set are still likely to give a good indication of how a CNN performs on data of similar nature (real data or synthetic data with other parameter values).

5

Model design

This chapter covers the design process of the CNN model used to classify radar data. The chapter begins with covering general considerations when designing a CNN classifier. The chapter continues with a description of the method used for finding a suitable model-design in this thesis. It then presents results from the model design-process which eventually lead to a final model-design. This chapter largely acts as an application of the theory on neural networks which is presented in chapter 3, on the radar data which are generated in chapter 4.

5.1 General considerations and method

Model design is a central challenge in supervised learning. There is no single model that works well for all tasks. Every distinct supervised learning-task requires thought and work with regard to model design and tuning of hyperparameters to make the model suitable for the particular problem.

The design of a CNN classifier is no exception. A too large network yields over-fitting or leads to unmanageable computational heaviness. A too small network leads to poor classification performance. Additionally there is a large set of other hyperparameters that all need to be set to achieve good results.

The brute-force method for finding a good model-design is to try every different possible model and look for the one that gives the best results. We however quickly realize that there exists an infinite number of possible network architectures and hyperparameter combinations. Some restrictions need to be put on how we vary the different hyperparameters.

The model-design process in this thesis mainly focuses on three aspects of the model, which all are expected to be of importance for the model performance. These are:

- *CNN architecture:*
  - *The number of convolutional layers*
  - *Kernel size for the convolutional layers*
  - *The number of filters in the convolutional layers*
  - *Whether to use max-pooling or not*
- *Batch size*

- *Optimizer settings: which optimizer to use and hyperparameters for this optimizer*

Other hyperparameters that are not investigated, such as choice of activation function, parameter initializations, convolutional stride etc. are set to values that are commonly cited for well-performing models in other research.

The method used in this thesis is to first test combinations of different layers in the CNN, without changing optimizer settings or batch size. By observing the validation loss during training of the different models we get a rough idea of what combinations of layers that yield good results. In the next step we make small tweaks to the models in order to increase the performance. The changes that are made to the models are based on the training characteristics from the previous results. The changes might be e.g. adding a layer, adding some regularization, or adding training epochs.

When a good set of layers has been found, we try this model with different batch sizes to see if the results can be improved. In the final step we use the best batch size and try different optimizer settings to see if we can improve the results even further.

The motivation behind this described method is that we expect the CNN architecture to be the most sensitive property of the model, with regards to achieved loss. The architecture is also likely the property of the model that is most specific to the particular problem of this thesis. For these reasons, we make the selection of architecture the first step of this method. The batch size and choice of optimizer are also expected to affect the results and are therefore studied in the next steps. These hyperparameters are however likely less problem specific and can thus be set to commonly-cited values while we study the more problem specific architecture in the first step. Remaining hyperparameters might also be important for the results, but can not all be studied because of time restrictions. Instead, we use commonly-cited values that have been used with good results for other applications.

The data set of 1200 instances per class generated in the previous chapter is used to create the training, validation and test sets. The test set contains 100 instances of each class. The validation set is 15% of the remaining data randomly selected. The training set is the other 85%.

5.2 Image resizing and normalizing

CNNs generally need input of a fixed size. However, the size of the range-doppler maps differ based on different PRIs and different number of pulses.

To make all range-doppler maps the same size, we employ image resizing through bilinear interpolation. The range-doppler maps in the data set of this thesis have widths and heights of roughly 100-200 “pixels”. All maps are resized to size 64×64 before they are fed to the CNN. This size seems large enough to preserve key features of the original range-doppler maps. A larger size could have been used but that would lead to increased computational complexity. An example of a range-doppler map before and after resizing is shown in figure 5.1.

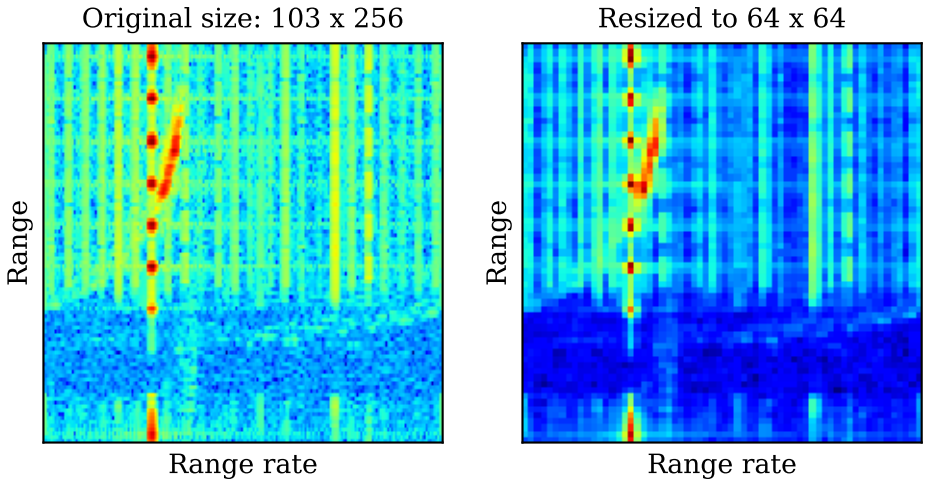

Figure 5.1: The left panel shows a range-doppler map of size 103×256. The right panel shows the same range-doppler map after it has been resized to size 64×64. We see that some details are lost in the resizing, but the main features of the range-doppler map are preserved.

In addition to the requirement of equally-sized input, we also want the data values to be of similar scales and ranges. This is achieved through first translating all elements of the range-doppler maps so that the mean value of each map is zero. Then through scaling of all values so that the standard deviation of the elements in each range-doppler map is unity.

5.3 Smoothing the validation loss

The most important quantity to observe during training is the validation loss. The validation loss is the best indication of how the model performs on unseen data so we generally look for the models that achieve the lowest validation loss during training.

The validation loss as a function of training progress is however sometimes noisy to various degrees. This can be a result of stochastic errors in the batch approximation of the loss function when performing optimization iterations. It can also be a result of the validation set being smaller than the training set, not giving an as accurate representation of the data distribution as the larger training set on which we perform the optimization.

We generally want to ignore the noise and instead look for the trends in the validation loss. This is achieved through smoothing the validation loss with a cubic Savitzky-Golay filter. The window size used in the filter is 31. More information on the Savitzky-Golay filter can be found in *Numerical Recipes in C: The Art of Scientific Computing* by Press, Teukolsky, Vellerling and Flannery [19].

En example of a noisy validation-loss curve which is smoothed by the Savitzky-Golay filter is shown in figure 5.2.

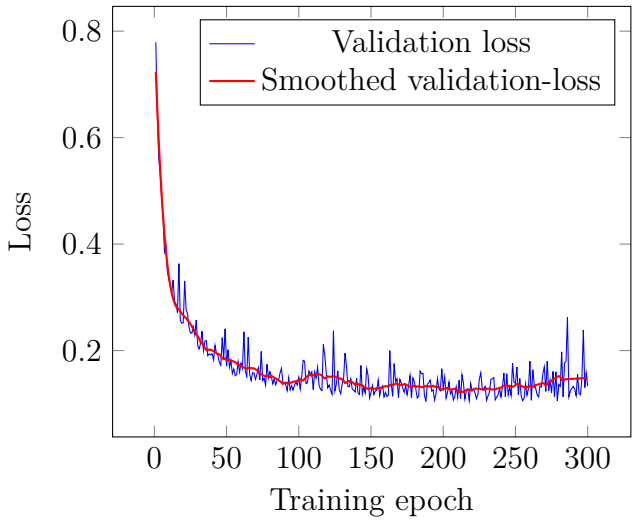

Figure 5.2: Validation loss as a function of training epoch, together with the validation loss smoothed by a Savitzky-Golay filter. The validation loss is sometimes noisy. The smoothing is done to more easily analyze the general trends of the training progress.

5.4 Finding a set of layers

The first step in the model design-process for this thesis is to find a good set of layers for the CNN. This is done by testing combinations of different layer-types and different number of layers while keeping the batch size and optimizer settings constant. Parameters that are varied in this first step with corresponding values that are tested are:

- *The number of convolutional layers:*
  $1, 2$ *and* 3
- *The number of filters in each convolutional layer:*
  16 *and* 32
- *The kernel size of the filters:*
  3×3, 5×5 *and* 7×7
- *With or without max-pooling layers*

The values for these parameters are selected so that they are similar to values used in famous CNN architectures [13]. Using only up to three convolutional layers is however shallow compared to many other CNN applications. The motivation for this is that we rather start with a too simple network of low computational loads to later increase the complexity if needed, instead of the other way around. The models which include max-pooling layers have max-pooling layers between all convolutional layers. However the models with only one convolutional layer do not use any max-pooling layer, because that would be redundant.

All of the models tested use a final classification-layer of the type described in chapter 3: the output from the last convolutional layer is fed into a global-average-

pooling layer which in turn provides the input for the final fully-connected layer. The convolutional layers of the models all use stride 1 with zero padding so that the width and height of the inputs are preserved. The activation for all convolutional layers is ReLU. All the initial weights in the models throughout this thesis are sampled from the *Glorot uniform-distribution* and all biases are initialized to zeros [20].

The performance of the different models are analyzed based on some measures calculated mainly from the smoothed validation loss. These measures are:

- *The minimum value for the smoothed validation loss for the entire training progress.*
- *The noise in the validation loss: calculated as the standard deviation of the difference between the validation loss and the smoothed validation loss.*
- *Minimum epoch: at which training epoch the minimum value for the smoothed loss is reached.*
- *The overfit difference: the difference between the minimum smoothed validation loss and the maximum smoothed validation loss reached* after *the minimum.*
- *The training time to complete the set number of training epochs.*

We basically look for models which reach a low validation loss quickly with low noise. The overfit difference is a measure of how much the model is overfitting after the minimum is reached. It does not have to be a problem that the model overfits if a low minimum is reached before the point of overfitting, but the overfit measure can help with identifying the models which might need added regularization.

The first set of training runs is done with a constant batch size of 32 and with the Adam optimizer using the default hyperparameter-values. Every model is trained for a total of 300 epochs. The results are given in table 5.1.

Table 5.1: Results from the first architecture search. The table is sorted by ascending minimum smoothed validation-loss from top to bottom. The kernel size is denoted by a single number because the filters have square sizes with sides of this length. We see that the lowest losses are achieved with models using 3 convolutional layers and no max-pooling layers. The Adam optimizer with the default parameters and a batch size of 32 is used for all models. Each model is trained for 300 epochs. The time it takes to complete the training for each model is given in the column *Training time*.

Kernel size	Nr conv layers	Nr filters per layer	Max Pooling	Min val loss smoothed	Min epoch	Overfit	Noise	Training time
7	3	16	No	0.113	165	0.050	0.033	4h 31m
7	3	32	No	0.115	129	0.270	0.127	8h 26m
5	3	32	No	0.115	149	0.150	0.052	4h 46m
5	3	16	No	0.116	204	0.030	0.024	1h 55m
3	3	32	No	0.122	214	0.030	0.023	1h 46m
5	3	16	Yes	0.127	122	0.139	0.038	23m
7	2	32	No	0.130	189	0.032	0.027	4h 29m
5	3	32	Yes	0.131	66	0.240	0.032	49m
5	2	32	No	0.133	241	0.019	0.021	2h 19m
7	2	32	Yes	0.140	187	0.080	0.040	1h 59m
5	2	32	Yes	0.143	296	0.005	0.026	44m
3	3	32	Yes	0.147	86	0.198	0.051	20m
3	3	16	Yes	0.148	243	0.034	0.032	9m
7	3	32	Yes	0.149	48	0.272	0.041	2h 4m
7	2	16	Yes	0.151	210	0.040	0.029	55m
5	2	16	Yes	0.152	260	0.025	0.023	22m
7	3	16	Yes	0.159	64	0.257	0.063	1h 4m
3	2	32	Yes	0.160	291	0.059	0.026	18m
3	3	16	No	0.161	255	0.016	0.029	48m
7	2	16	No	0.162	300	0.000	0.028	2h 25m
5	2	16	No	0.170	300	0.000	0.023	1h 6m
3	2	32	No	0.174	300	0.000	0.021	58m
3	2	16	No	0.176	300	0.000	0.017	31m
3	2	16	Yes	0.188	283	0.012	0.023	8m
7	1	32		0.272	294	0.004	0.010	53m
7	1	16		0.306	300	0.000	0.006	42m
5	1	32		0.325	292	0.006	0.010	19m
5	1	16		0.358	300	0.000	0.007	12m
3	1	32		0.442	300	0.000	0.006	9m
3	1	16		0.458	300	0.000	0.005	6m

We see that more layers generally seem to give a lower minimum smoothed validation loss. Max-pooling layers reduce the training times significantly but most of the best performing models are without max-pooling layers. Many models overfit to various degrees. Overfitting seems to happen more in the deeper models.

The noise is larger in the deeper models. Most of the one-layer models have low noise. The one-layer models also reach their minimum at the end of the training progress, which indicates that a lower validation loss could be reached with more training epochs.

As a next step we try to improve some of the models by increasing the number of training epochs, adding weight decay, or adding convolutional layers.

The models for which we add weight decay are some of the well-performing models that have some degree of overfitting. These are:

- *Kernel size:* 7×7
  Nr. convolutional layers: 3
  Nr. filters: 16
  No max pooling
- *Kernel size:* 5×5
  Nr. convolutional layers: 3
  Nr. filters: 32
  No max pooling
- *Kernel size:* 5×5
  Nr. convolutional layers: 3
  Nr. filters: 16
  With max pooling
- *Kernel size:* 5×5
  Nr. convolutional layers: 3
  Nr. filters: 32
  With max pooling

Weight decay is added to all weights and biases with weight-decay parameter $\lambda = 5 \cdot 10^{-4}$ for the convolutional layers and $\lambda = 10^{-3}$ for the final fully-connected layer.

More convolutional layers are added to 3-layer models that have a low degree of overfitting; if no overfitting is happening, the model could possibly benefit from increased complexity. The 3-layer models with low overfitting are:

- *Kernel size:* 5×5
  Nr. filters: 16
  No max pooling
- *Kernel size:* 3×3
  Nr. filters: 32
  No max pooling

These settings are tested in the next step with 4 and 5 convolutional layers instead.

The models that reach their minimum validation loss at the end of the training progress can likely benefit from being run for a larger number of training epochs. We therefore run some of these models again with a larger number of training epochs. These models are:

- *Kernel size:* 5×5
  Nr. convolutional layers: 2
  Nr. filters: 32
  With max pooling
- *Kernel size:* 7×7
  Nr. convolutional layers: 2
  Nr. filters: 16
  No max pooling
- *Kernel size:* 5×5
  Nr. convolutional layers: 2
  Nr. filters: 16
  No max pooling
- *Kernel size:* 7×7
  Nr. convolutional layers: 1
  Nr. filters: 32

The results from the second run are shown in table 5.2. We quickly note that added weight decay did not increase the performance of any model, adding weight decay instead made the minimum loss larger. We see that the model which gives the lowest smoothed validation loss is:

- *Kernel size:* 3×3
  Nr. convolutional layers: 4
  Nr. filters: 32
  No max pooling

This model also has a relatively low training time compared to the other models that give low losses. Adding a fifth layer with these settings does not improve the performance of the model, so 4 convolutional layers seems to be a good choice when using these parameters. None of the models for which we increased the number of training epochs reached low enough losses to compete with the best performing models.

Table 5.2: Results from the second architecture search. The rows are sorted by ascending minimal smoothed validation-loss from top to bottom. In this set of runs, some models have added convolutional layers, some models have added weight decay, and some models are run for more training epochs. Adding weight decay did not improve the models. The lowest smoothed validation loss is achieved with kernel size 3×3, 4 convolutional layers, 32 filter per layer, no max-pooling layers, and no weight decay. All models use the Adam optimizer with default hyperparameters and batch size 32.

Kernel size	Nr conv layers	Nr filters per layer	Max Pooling	Weight decay	Min val loss smoothed	Epochs	Min epoch	Overfit	Noise	Training time
3	4	32	No	No	0.098	300	129	0.125	0.036	2h 33m
5	4	16	No	No	0.111	300	80	0.127	0.046	2h 51m
7	3	16	No	No	0.113	300	165	0.050	0.033	4h 31m
5	3	32	No	No	0.115	300	149	0.150	0.052	4h 46m
5	3	16	No	No	0.116	300	204	0.030	0.024	1h 55m
3	5	32	No	No	0.120	300	84	0.200	0.033	3h 20m
3	3	32	No	No	0.122	300	214	0.030	0.023	1h 46m
5	2	16	No	No	0.123	1200	998	0.038	0.020	4h 26m
5	3	16	Yes	No	0.127	300	122	0.139	0.038	23m
5	3	32	Yes	No	0.131	300	66	0.240	0.032	49m
7	2	16	No	No	0.132	600	411	0.028	0.027	4h 37m
5	5	16	No	No	0.135	300	65	0.298	0.040	3h 44m
5	2	32	Yes	No	0.142	600	199	0.077	0.028	1h 29m
7	1	32		No	0.177	2400	2377	0.003	0.011	6h 35m
5	3	32	Yes	Yes	0.182	300	88	0.042	0.026	48m
5	3	16	Yes	Yes	0.189	300	253	0.027	0.033	23m
7	3	16	No	Yes	0.189	300	245	0.024	0.032	4h 20m
5	3	32	No	Yes	0.196	300	136	0.029	0.031	4h 48m

5.5 Finding batch size

The next hyperparameter to study is the batch size. In the previous section we found a well-performing architecture using kernel size 3×3, 4 convolutional layers, 32 filters in each layer without any max-pooling layers. In this section we take this architecture and try different batch sizes to see if we can improve the results even further. The optimizer we use is still Adam with the default hyperparameters.

The previously used batch size is 32. We now try some batch sizes smaller and larger than this number. It is common for batch sizes as powers of 2 offer better run times [9]. We therefore try the batch sizes: 8, 16, 32, 64, 128, and 256. The number of training epochs for each batch size is set so that the validation loss seems to reach its minimum value before the end of training.

The results from using different batch sizes are shown in table 5.3. This table also includes the unsmoothed minimum validation-loss. We see that increasing batch size also increases training times. This is expected because a larger batch size requires more training examples to be fed through the CNN in order to make one update step. We also see that a larger batch size seems to make the validation loss less noisy. This is likely because a larger batch size gives a better approximation of the loss function, leading to less randomness in the update steps. The lowest unsmoothed validation-loss is reached with the largest batch-size.

Table 5.3: Results from models using different batch sizes. All models use kernel size 3×3, 4 convolutional layers, 32 filters per layer, and no max-pooling layers. The Adam optimizer with default hyperparameters is used for all models. The lowest unsmoothed validation loss is reached with batch size 256. The noise in the validation loss seems to decrease with increased batch size.

Batch size	Min val loss	Min val loss smoothed	Epochs	Min epoch	Overfit	Noise	Training time
256	0.085	0.100	1200	890	0.044	0.023	10h 5m
64	0.093	0.102	300	207	0.041	0.029	2h 52m
32	0.094	0.098	300	129	0.125	0.036	2h 33m
8	0.101	0.142	300	53	0.238	0.044	2h 34m
128	0.103	0.127	600	414	0.080	0.028	5h 30m
16	0.105	0.132	300	68	0.190	0.029	2h 38m

We see no clear correlation between achieved losses and batch size, other than that the lowest loss is achieved with the largest batch size. On the other hand, the decrease in noise from using a larger batch size makes it seem that it is better to use larger batch-sizes for more predicable results. Increasing the batch size even further could possibly lead to even lower losses and less noise. This would however lead to large computer memory requirements and even longer training times. We use batch size 256 from this point forward because of the low achieved loss and low noise while still keeping training times manageable.

5.6 Finding optimizer settings

The final step in the model design-process is to try different optimizer settings. We have until now only used the Adam optimizer with the default hyperparameters. We now try to adjust some of these hyperparameters. We also try the optimizer SGD + Nesterov momentum. The different optimizer-settings are tried on a CNN with 4 convolutional layers, 32 filters in each layer, 3×3 filter size, no max-pooling layers, using batch size 256. These are the best settings from the previous sections.

The default hyperparameters for the Adam optimizer are

$$\begin{aligned} \eta &= 0.001 \\ \beta_1 &= 0.9 \\ \beta_2 &= 0.999 \\ \epsilon &= 10^{-8}. \end{aligned} \tag{5.1}$$

We try both a larger learning rate $\eta = 0.01$ and a smaller learning rate $\eta = 0.0001$ (with the other parameters as default). The documentation for the machine-learning framework *Tensorflow* suggests some tuning of ϵ and proposes the values $\epsilon = 0.1$ and $\epsilon = 1.0$ [21]. These values are also tested (with the other parameters default).

We also run one model with the optimizer SGD + Nesterov momentum. The hyperparameters used for this optimizer are learning rate $\eta = 0.1$, momentum $\alpha = 0.8$, learning rate decay $\gamma = 0.1/1200$ (calculated as learning rate divided by number of training epochs). These values are selected because they lie in common ranges of values cited in other research [9].

The number of training epochs is 1200 for all different optimizer settings, except for Adam with $\eta = 0.0001$ which is run for 2400 epochs because we expect a smaller learning rate to make the training slower.

The results from the different optimizer settings are shown in figure 5.3. Because of the different characteristics of the validation-loss curves, the data is now plotted instead of presented in tables.

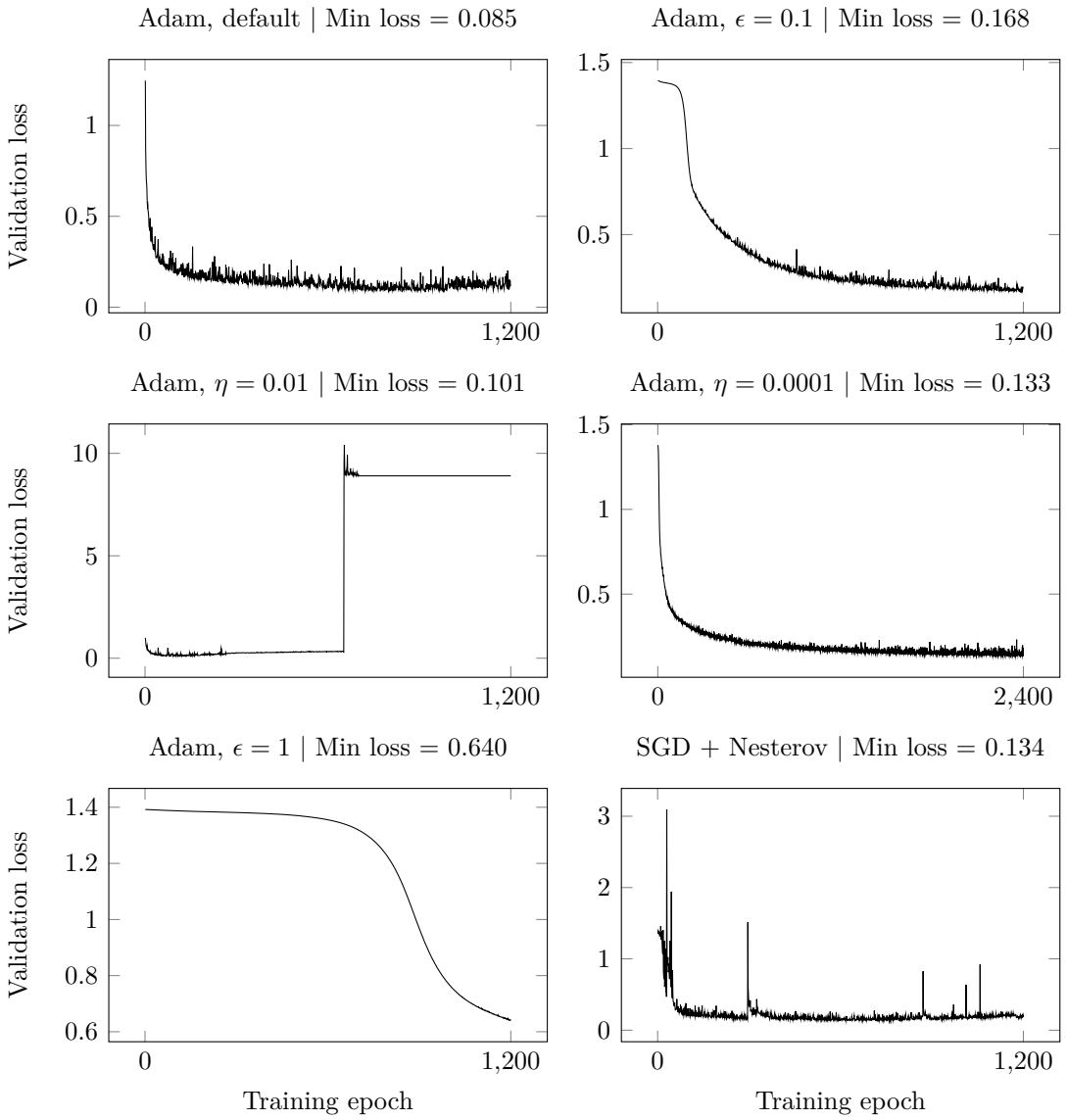

Figure 5.3: The validation loss during training when using different optimizers. The values denoted *min loss* are the minimum losses reached during the full training progress for the respective optimizers. The training times are ∼11 hours for the sessions with 1200 epochs and ∼22 hours for Adam with $\eta = 0.0001$ that is run for 2400 epochs.

The minimum validation loss is still reached with the Adam default settings. A larger ϵ makes the training progression much slower than with the default ϵ. Using $\eta = 0.01$ makes the validation loss descend quickly but it later jumps to large values.

Using $\eta = 0.0001$ makes the validation loss smooth but the validation loss never reaches very low values. It is however not clear that this curve yet has reached its minimum at the end of training. Maybe using even more training epochs for $\eta = 0.0001$ could decrease the loss further, but this would lead to large training times.

SGD + Nesterov momentum gives a noisy validation loss with many sharp peaks. The minimum validation loss is also not very low. Maybe this optimizer can be improved by tweaking the hyperparameters.

We can conclude that the Adam optimizer with its default parameter values seems to be a good optimizer for the classification problem in this thesis. It provides reasonable training times and reaches low losses.

5.7 Final model

The experiments in this chapter has led to a final CNN architecture, optimizer setting, and set of hyperparameters that seems suitable for the radar data used in this thesis. These are:

- *CNN architecture:*
  - 4 *convolutional layers*
  - 3×3 *sized filters*
  - 32 *filters in each layer*
  - *No max-pooling layers and no weight decay*
- *Batch size:* 256
- *Optimizer: Adam optimizer with the default hyperparameters*

The final CNN-architecture is shown in figure 5.4. ReLU is used as activation for all convolutional layers. All convolutional layers use stride 1 with zero padding so that the width and height of the input is preserved. The CNN ends with a global average pooling layer into a fully connected layer with a softmax activation, as described in chapter 3.

The next step is to train this model and use the early stopping technique to save the model parameters that gives the minimum validation loss during training. The models are trained five times because the random starting conditions might lead to different minima. The saved model with the lowest validation loss is then used to predict the data samples in the test set to give a final score of the model accuracy.

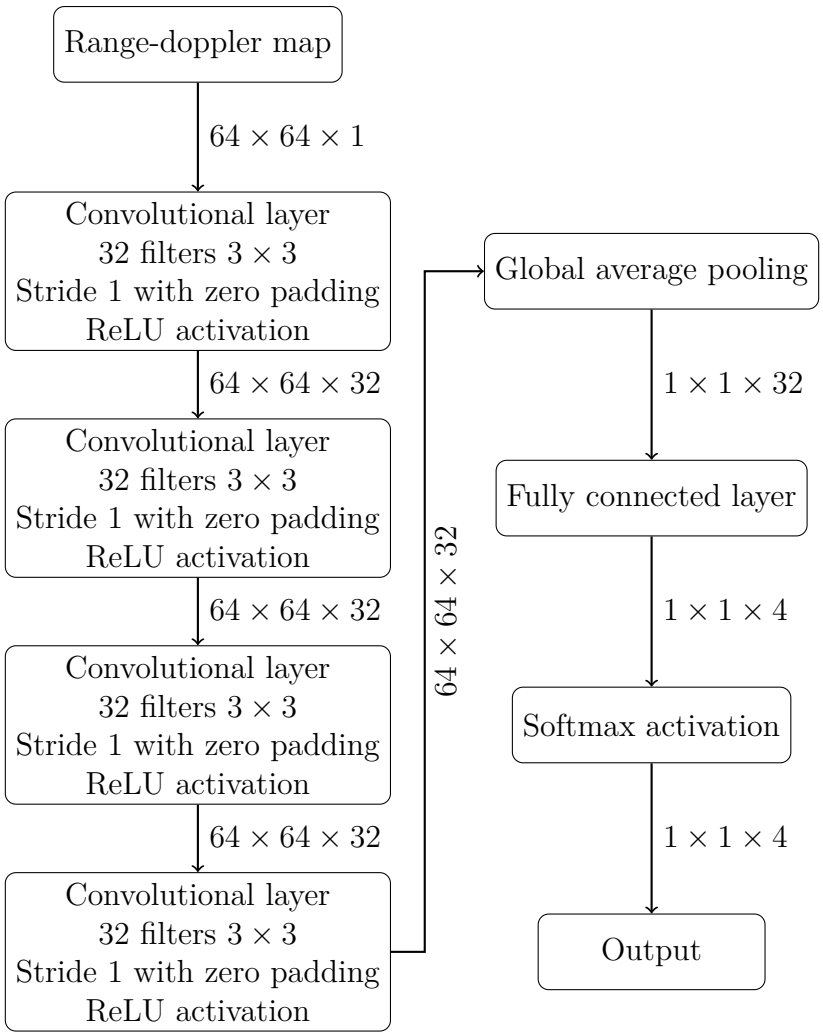

Figure 5.4: The final CNN architecture resulting from the model design-process. This architecture seems suitable for the radar data in this thesis. The dimensions of the inputs and outputs for each layer are denoted on the arrows. The CNN has 4 convolutional layers, each with 32 filters of size 3×3. It has no max-pooling layers. The output of the final convolutional layer is fed into a global average pooling layer which provides the input for a fully connected layer with the final softmax activation giving the output scores.

6

Classification results

This chapter presents the classification results for the best model design from the previous chapter. A model is trained from scratch five times and the best iteration is selected to be evaluated on the test set. The classification results on the test set are then presented as a confusion matrix. The chapter ends with a discussion of the classification results.

6.1 Final training-runs

The model design-process in the previous chapter gave us the model settings:

- *CNN architecture:*
  - 4 *convolutional layers*
  - 3×3 *sized filters*
  - 32 *filters in each layer*
  - *No max-pooling layers and no weight decay*
- *Batch size:* 256
- *Optimizer: Adam optimizer with the default hyperparameters*

We now run five training runs with these settings (with different randomized initiations of the model parameters). Each run is 1500 training epochs long to make sure that the validation loss has reached its minimum at the end of training. We use the early-stopping technique to save the parameters that gives the lowest validation loss in each run.

The results from the five runs are given in table 6.1. The lowest validation loss obtained in these runs is 0.0874 from run number 1, which is slightly larger than the previous minimum of 0.085 with the same settings (but the parameters that gave this minimum were not saved). We actually observe relatively large differences between the minimum validation-losses from the different runs, which indicates that the starting condition for the optimization has a significant influence over the results.

Table 6.1: Results from the final training-runs. The runs use the same model design, but are initialized with different randomized model-parameters. The lowest validation loss is achieved in run nr. 1. We see that the difference between the minimum losses from different runs are relatively large. This shows that the initial parameter initialization affects the training results.

Run nr.	Minimum validation loss	Training time
1	0.0874	13h 32m
2	0.1003	13h 28m
4	0.1059	13h 26m
5	0.1064	13h 37m
3	0.1132	13h 31m

The validation loss and the training loss as functions of training epoch for run number 1 are plotted in figure 6.1. The training progress behaves much like what we expect from a well-performing model. The training loss is continuously decreasing with some noise. The validation loss is slightly larger than the training loss and decreases until around training epoch 900 where some overfitting starts to happen. The validation loss is slightly more noisy than the training loss. At the final few training epochs, we observe a sharp peak in the validation loss. This peak is accompanied by a smaller peak in the training loss. This peak is after the point of overfitting and does not matter for the saved model, but it shows that small update steps can lead to large jumps in the loss.

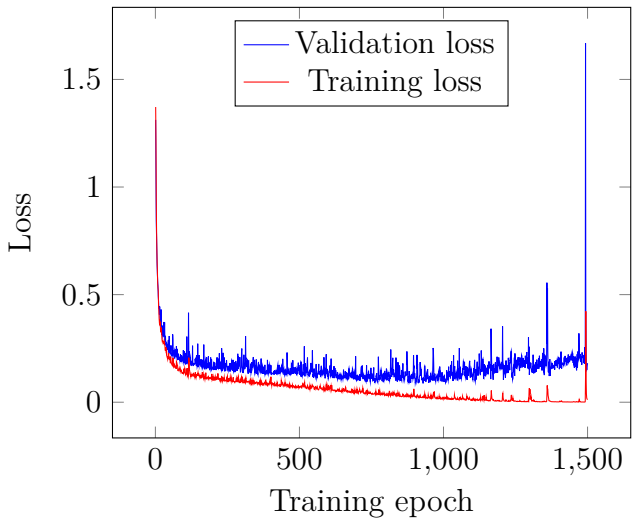

Figure 6.1: The training progression for the best run (run nr. 1). The training loss has an initial quick decrease. After epoch ∼100 it is decreasing steadily with little noise. The validation loss is more noisy than the training loss and is slightly higher than the training loss for the large part of the training progress. A minimum of 0.0874 is reached in the validation loss around epoch ∼900. After this minimum, the model starts to overfit.

6.2 Performance on test set

The final performance-measure for the obtained model is the performance on the test set. The saved model from run number 1 is used to predict all the samples in the test set. These predictions are compared with the true labels to evaluate the final performance-scores.

The loss evaluated on the test set is 0.0989. This is slightly larger than the validation loss, but only ∼13% larger. The accuracy on the test set is 96.75%. The true labels versus the predicted labels are shown in table 6.2 as a *confusion matrix*. We see that the model performs well on the range-doppler maps containing interference and noise jamming, with 99/100 samples for interference and 100/100 samples for noise jamming correctly classified. Most of the errors arise for the no-jamming and the DRFM classes, with 93/100 respectively 95/100 samples correctly classified.

Table 6.2: Confusion matrix for the classification results on the test set of 400 data samples. The confusion matrix shows true labels versus predicted labels. The sum of the diagonal elements is the number of total correct predictions. We see that most incorrect predictions happen by mixing up instances belonging to classes *no jamming* and DRFM.

		Predicted class			
		No jamming	Interference	DRFM	Noise jamming
True class	No jamming	93	0	7	0
	Interference	0	99	1	0
	DRFM	5	0	95	0
	Noise jamming	0	0	0	100

Accuracy: 387/400, 96.75%

6.3 Analysis of classification errors

Most of the incorrectly classified instances belong to the classes *no jamming* and *DRFM*. We can look at some of these incorrectly classified range-doppler maps to get an idea of why the model makes the incorrect classifications.

Figure 6.2 shows an incorrectly classified instance from the *no jamming*-class which is classified as DRFM. The ground clutter in this range-doppler map is relatively weak and has a few "peaks". The peaks in the ground clutter are probably interpreted by the model as false targets from DRFM jamming, which can explain the incorrect prediction. The other incorrectly classified range-doppler maps from the *no jamming*-class look similar to this example.

An incorrectly classified instance from the DRFM class is shown in figure 6.3. This instance is classified as *no jamming*. We see that no DRFM jamming is visible in this range-doppler map. The jamming is likely covered by the strong ground clutter. This makes the incorrect classification of this range-doppler map reasonable.

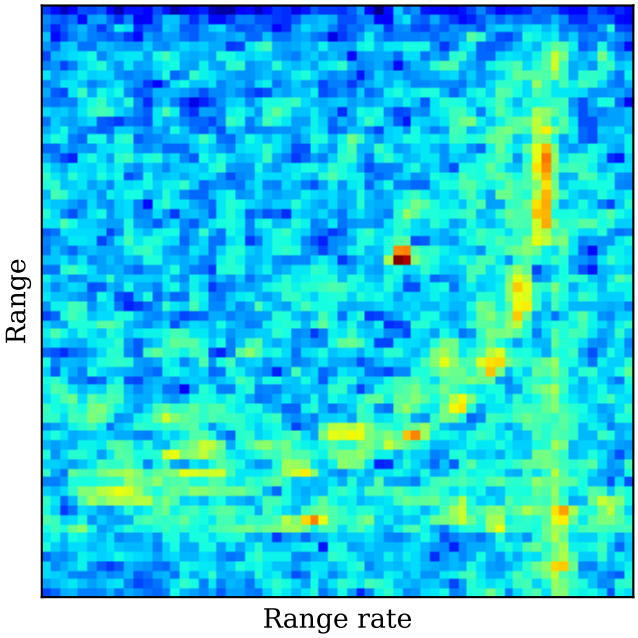

Figure 6.2: A range-doppler map in the test set belonging to the class *no jamming* (resized to 64×64). This instance is incorrectly classified as DRFM. A possible explanation is that the ground clutter has "peaks" that might be interpreted as false targets by the CNN.

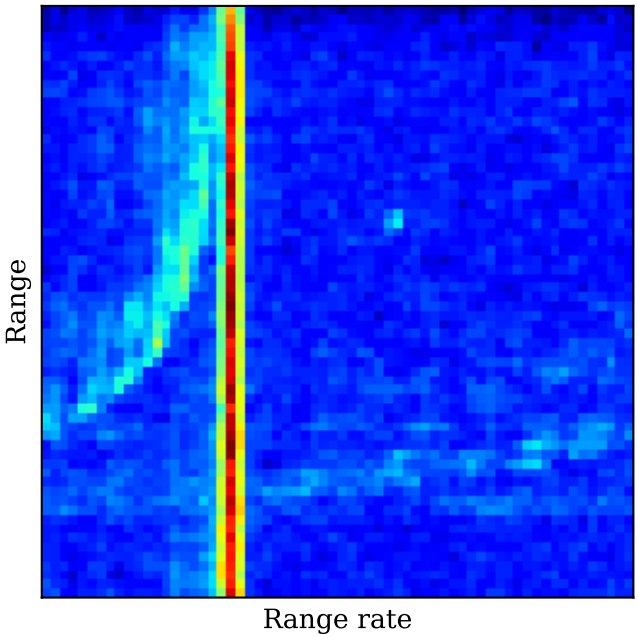

Figure 6.3: A range-doppler map in the test set belonging to the class DRFM (resized to 64×64). This instance is incorrectly classified as *no jamming*. The DRFM jamming is not visible in the range-doppler map. It might be covered by the ground clutter or has disappeared for some other reason.

Another incorrectly classified DRFM instance is shown in figure 6.4. This instance is also classified as *no jamming*. The DRFM jamming is visible in this range-doppler map, but it is weak and located on the border of the map (the lower right part). Most other DRFM instances in the data set have much more obvious jamming. The output scores from the model on this sample are ∼0.7 for no jamming and ∼0.3 for DRFM, which indicates that the model shows some uncertainty. The other incorrectly classified DRFM instances look similar to figure 6.3 and 6.4.

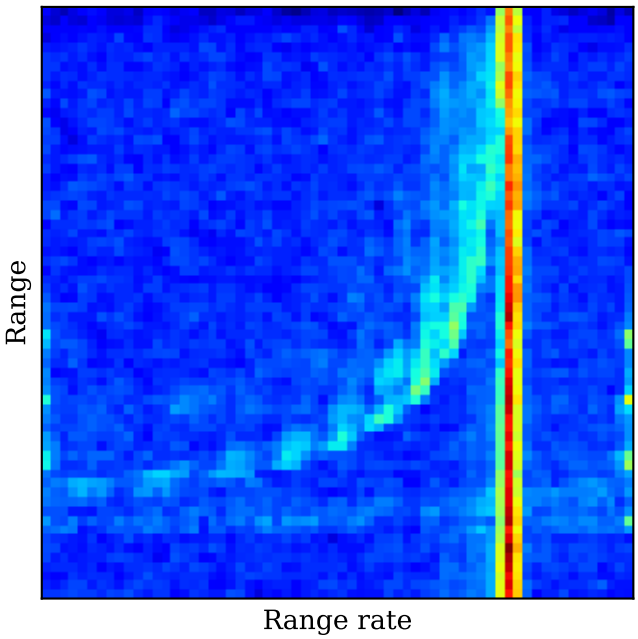

Figure 6.4: A range-doppler map in the test set belonging to the class DRFM (resized to 64×64). This instance is incorrectly classified as *no jamming*. Signals that likely are DRFM jamming are visible in the lower right part of the range-doppler map. These signals are however weak. The weak signals in combination with the fact that the signals lie on the border of the map provide a possible explanation for the incorrect classification.

7
Conclusion

This final chapter summarizes the main conclusions that can be drawn from this thesis work. It also suggests further work that can be done within this field.

7.1 Main findings and discussion

The most important finding of this thesis is the fact that it seems possible to achieve high classification accuracies on radar data using CNNs. In this thesis we reach a classification accuracy of 96.75% on a test set. A jamming classifier with an accuracy this high should be useful in real-world applications, possibly to support already existing ECCM techniques or to provide guidance for radar operators. Further studies are required to investigate if similar accuracies can be reached on non-synthetic data. However, the results in this thesis make the prospects promising.

One interesting finding is that many of the classification errors that the model makes on the test set are errors that also a human likely would make. This implies that the CNN model looks for features in the image similar to how a human does. Even though these particular errors makes the achieved accuracy lower, they give a valuable indication of how we can expect a CNN to behave when used for range-doppler maps. If it is impossible for a human to make a certain classification, it is probably also impossible for a CNN.

The model-design process proves to be a time-consuming, and in some regards difficult task. There are many hyperparameters that can be tweaked and it is difficult to find optimal values for all of them, especially when the models require long training-times. The results are however not very sensitive to all hyperparameters. We note that low losses are achieved with many different types of CNN architectures. The model design that we finally settle for is a CNN with kernel size 3×3, 4 convolutional layers, 32 filters in each layer, and with no max-pooling layers. Although, we can probably reach similar final classification-accuracies with architectures other than this one.

The final training-runs from which we choose the final model are made with the same model design but with different parameter-initializations (sampled from the same distribution). The results from these runs show that the parameter initialization impacts which minimum loss is reached during training. The results on which we base the model design-decisions are also dependent on randomized parameter-initializations. However, during the design process we only run each model once

(mainly because of time limitations). Going through the same model design-process again with new samplings of the initial parameters for each model could therefore maybe lead to settling for a different final model-design. It is probably better to run each model a few times for more robust results.

There are also hyperparameters or model designs that are not at all studied in this thesis, e.g. different distributions for parameter initializations, different activation functions, and combinations of convolutional layers with different kernel sizes and number of filters. There are many ways the model design-process could be been done differently, which possibly could lead to different final results. The results from the model design-process in this thesis should however give a good general indication of what kind of models are well-suited for the radar data that we study. For example using a CNN with only one or two convolutional layers is likely not enough for good results, no matter how well we tune all other hyperparameters. Using many more convolutional layers than three or four probably also does not lead to good results.

Apart from the model design-process, a large part of the work in this thesis is centered around generating labeled data. Machine-learning methods are data driven, so it natural that much time of a machine-learning study is spent on work related to data management. The main challenge with regards to data for this thesis were the parameter selections for the data generation. The parameters need to be chosen both so that different data classes are distinguishable and so that the data is relatively realistic. If the synthetic data is too disparate from real data, the results do not provide a good indication of how a similar model can perform on real data.

7.2 Further work

There is a lot of potential for further work following this thesis. Some examples are given in the sections below.

7.2.1 Transferring a network trained on synthetic data to real-world data

A radar-jamming classifier is only useful in real-world applications if it works on real-world data. Training a model that works on real likely requires a labeled data-set of real-world data. Unless this kind of data set already exists, these labels probably need to be added manually, which can become a time-consuming task.

One way to get away with a smaller data set of labeled real-world data is to first pre-train a model on synthetic data which are generated to resemble the real-world counterpart as much as possible. The model can then be fine-tuned on the real-world data-set. The fine-tuning process can likely benefit from a smaller learning rate because we expect the model parameters reached in the pre-training phase to be somewhat close to a good parameter-set for the real-world data.

7.2.2 Analysis and improvements of computational times

The computational times required for both training and using the model in this thesis are of great interest. Reducing the training times can allow us to train more models and therefore more easily tune hyperparameters. A significant improvement to the training times obtained in this thesis can likely be achieved through parallelization on a GPU. The model training in this thesis is all carried out on a CPU. Training of neural networks has great potential for parallel computing, and can thus benefit from the large number of cores on a GPU [4].

The computational requirements for making predictions are very low compared to training times [9]. An analysis of computational times required for making model predictions in a real-world application can however still be of significance. In a real-world application, predictions need to made in real time on radar hardware, where other signal processing is already taking place. Any further added computational task need to be ensured not to overload the capacity of the hardware.

7.2.3 Using additional input-parameters

The model in this thesis only uses the range-doppler map as input data. One option is to use additional parameters as inputs to the network. Such parameters could be e.g. the PRI and the direction of the radar main-lobe for each range-doppler map. These parameters could provide the network with additional information that could simplify the classifications. Additional parameters can be added to the output features of the CNN as inputs to a final network of fully-connected layers as shown in figure 7.1.

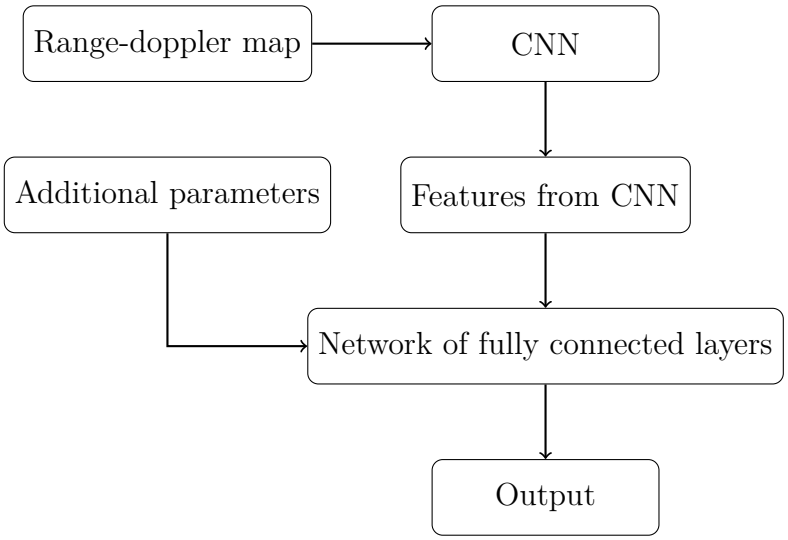

Figure 7.1: Additional parameters can be combined with image features from a CNN. This can be done by feeding both the image features and the additional parameters to a network of fully-connected layers.

7.2.4 Using complex data

When constructing range-doppler maps, only the amplitude data of the Fourier transform is used. The output of a Fourier transform is actually complex with information about both the signal amplitude and phase at given frequencies. The model in this thesis uses only the amplitude data and therefore ignores any phase information. One option is to instead feed complex data to the CNN. The additional phase information can possibly simplify the classifications.

7.2.5 Finding jammed regions in the range-doppler map

A useful addition to classifications of jammed range-doppler maps would be to detect the regions of the map in which jamming occurs. This problem falls under the category of image segmentation. Image segmentation is the problem of assigning labels to each pixel in an image, where the pixels sharing a label also share some characteristics. A modern method that can be used for this is *Mask R-CNN* [22].

7.2.6 Using time-dependent data

In reality, radar data are part of time-dependent events. Information about previous states can likely be used to make predictions about the current state. A future goal can be to use a machine-learning model which includes some form of time-analysis. Methods for this kind of machine learning are however not very far developed. Modern machine-learning techniques for visual recognition are generally limited to analysis of still images.

Bibliography

[1] HD Griffiths, Christopher Baker, and David Adamy. *Stimson's introduction to airborne radar*. Scitech Pub Incorporated, 2014.

[2] Christopher D Manning, Christopher D Manning, and Hinrich Schütze. *Foundations of statistical natural language processing*. MIT press, 1999.

[3] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Brian Kingsbury, et al. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal processing magazine*, 29, 2012.

[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[5] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[6] Samuele Capobianco, Luca Facheris, Fabrizio Cuccoli, and Simone Marinai. *Vehicle Classification Based on Convolutional Networks Applied to FMCW Radar Signals*, pages 115–128. 01 2018.

[7] W Carrara, R Goodman, and R Majewski. *Spotlight Synthetic Aperture Radar: Signal Processing Algorithms,(ser. Artech House remote sensing library)*. Norwood, MA, USA: Artech House, 1995.

[8] Mirabel Cerqueira Rezende, Inácio Malmonge Martin, Roselena Faez, Marcelo Alexandre Souza Miacci, and Evandro Luis Nohara. Radar cross section measurements (8-12 ghz) of magnetic and dielectric microwave absorbing thin sheets. *Revista de Fısica Aplicada e Instrumentaçao*, 15(1):24–29, 2002.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[10] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[12] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[13] Karpathy. Cs231n, convolutional neural networks for visual recognition. `https://github.com/cs231n/cs231n.github.io`, 2019.

[14] Keras - deep learning for humans. `https://github.com/keras-team`, 2019.

[15] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[16] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[17] Max Kuhn and Kjell Johnson. *Applied predictive modeling*, volume 26. Springer, 2013.

[18] Hamish Meikle. *Modern radar systems*. Artech House, 2008.

[19] WH Press, SA Teukolsky, WT Vellerling, and BP Flannery. Numerical recipes in c: The art of scientific computing, 1999.

[20] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[21] Tensorflow - an end-to-end open source machine learning platform. `https://github.com/tensorflow`, 2019.

[22] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.