Rheinische
Friedrich-Wilhelms-
Universität Bonn

Institute for Computer Science
Department VI
Autonomous Intelligent Systems

Rheinische Friedrich-Wilhelms-Universität Bonn

Master Thesis

Signal Decomposition in the Frequency Domain

Author:
Julian Max Schröteler

First Examiner:
Prof. Dr. Sven Behnke

Second Examiner:
Prof. Dr. Joachim K. Anlauf

Advisor:
Max Schwarz

Date: August 28, 2019

Declaration

I hereby declare that I am the sole author of this thesis and that none other than the specified sources and aids have been used. Passages and figures quoted from other works have been marked with appropriate mention of the source.

Place, Date

Signature

Abstract

Convolutional Neural Networks (CNNs) are the main building block in state-of-the-art approaches for numerous computer vision task. They are good at extracting features on all levels of granularity, but they are bad at understanding the relation between an image and a spatially transformed version of that same image.

In this thesis, we try to develop methods that have a better understanding of that exact relation. Rather than using the spatial domain representation, which is critically deformed by spatial transformations, we make use of the frequency domain representation. This representation behaves in a completely different manner when a spatial transformation is applied, and we investigate whether we can take advantage of this. For instance, a spatial translation is mainly encoded in the phase spectrum of the frequency domain representation, so an idea would be to use the phase spectrum in order to recover the translation parameters.

We limit our investigation to one-dimensional signals and try to extract semantic information from those signals, i.e. we decompose the signals into a localization, which tells us where in the signal some interesting pattern occurs, and a normalization, which tells us what kind of interesting pattern occurs. In other words, the localization is a spatial translation that characterizes the relation between the signal and the normalized pattern, and finding an estimate of the localization is equivalent to understanding the relation between the signal and the normalized pattern.

We suggest numerous solutions for these signal decomposition tasks, ranging from very simple methods, like multiplying two arrays in the frequency domain, to more sophisticated methods, such as building and training CNNs that work with the frequency domain representation. We also combine two of those methods in an iterative manner, which gives us the possibility to accurately simultaneously extract both localization and normalization at once, even for two different patterns at the same time.

In order to evaluate our methods, we construct our own Morse signal dataset. Since different Morse letters can look very similar to each other, it is not trivial to decompose Morse signals. We find that methods which combine frequency domain computations with CNNs reach high accuracies, showing that the frequency domain can indeed be used to deal with spatial translations on the input data.

Contents

1 Introduction

1.1 Motivation

Convolutional Neural Networks (CNNs) sit at the core of state-of-the-art approaches to numerous computer vision tasks such as object detection, image classification, and semantic segmentation, where their ability to learn features on all levels of granularity makes them an extremely powerful tool.

Convolutional layers in neural networks maintain a number of small *kernels* (also called *filters* or *feature detectors*); during inference, these kernels are convolved with the input (e.g. an image) to produce a feature map. The feature map contains information about whether and where the feature encoded by the kernel is present in the input or not. During learning, the values in the kernels (also called *weights*) are usually adjusted using a backpropagation optimization scheme, enabling the network to learn on its own which exact features are important and how it wants to represent them in the kernels.

If several of these convolutional layers are connected in series, the features become more complex the further away they are from the original input. Just like a single convolutional layer locally combines pixel values of an input image to get rudimentary features, a subsequent convolutional layer can combine these feature map values to get more sophisticated patterns. Consider, for example, a CNN that recognizes faces: The features in the first layers would be as simple as straight edges; in the next layer, the features may be circles or curved lines; further layers would combine these simple structures to eyes and ears, and the final layer would have the entire face as feature.

Even though contemporary deep CNNs are far more elaborate than just executing many simple convolutional layers one after another, they can still suffer from one problem rooted within their basic building block, the convolution layer itself: Missing comprehension of viewpoint changes. If the object in the image is transformed in an affine manner, e.g. translated, rotated, or scaled, then the network will not be able to make use of the features it finds in the original image to find the features in the transformed image.

For instance, using the face recognition example again, we could have a look at two images of the same face, where the second image is simply a rotated version of

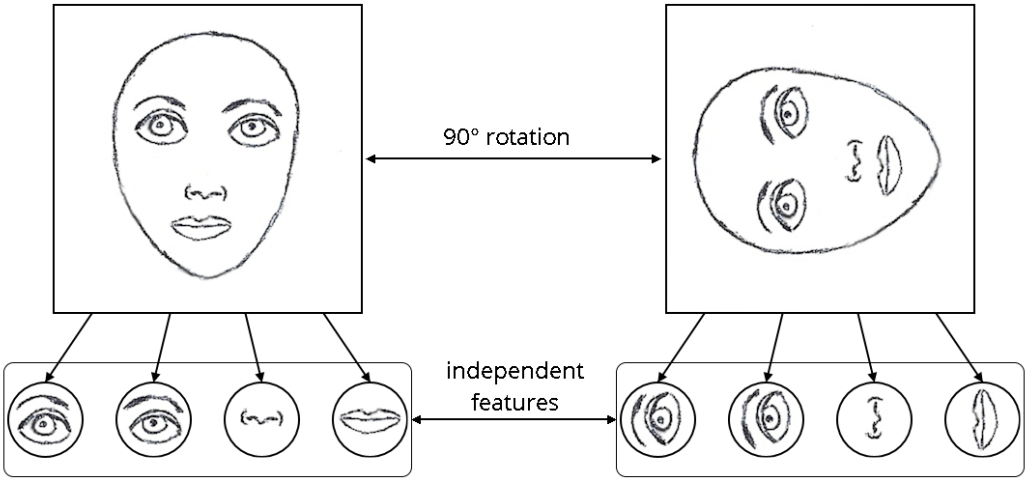


Figure 1.1: Example of facial features. The two images show the same face, only rotated differently. Accordingly, the facial features are the same in both images, only rotated differently. A CNN cannot make use of that correlation and treats the two sets of features as completely independent from each other.

the first one (see Figure 1.1). In the first image, a simple CNN that has never seen rotated images of faces during training would find mid-level features such as eyes, the nose, and the mouth, but for the second image, it would not find these features, because it is unable to see the simplification that the eyes, nose and mouth in the second image are the same as in the first, only rotated. Instead, an additional set of feature detectors is required, where each detector corresponds to a rotated feature. And, of course, the network needs to be trained with additional data samples, namely rotated faces, to learn the weights for those new kernels. This is extremely inefficient, because it means that in order to be able to generalize to unseen examples, the CNN needs a large number of individual kernels: it requires, for each feature, roughly one kernel for each possible rotation of the in fact semantically same object, and it needs a large training set that contains samples for each possible rotation. Analogously, introducing other affine transformations to the input images would increase required network size and training dataset size even more. Today's state-of-the-art CNN architectures incorporate methods to keep the network to a moderate size, but they still need large datasets for effective training.

1.2 Approach

Spatial deformations change the structure of an input image in a way that makes it difficult for CNNs to recognise the correlation between the features in the original image and the features in the deformed image. Therefore, the basic idea we investigate in this thesis is to transfer the input to another representation before passing it on to any convolutional layer. Specifically, we apply a Fast Fourier Transform (FFT) to get the frequency domain representation of the input. In the spatial domain, information is manifested in a mapping from pixel locations to intensity or RGB values; contrary to this, the FFT decomposes images into a sum of sinusoids, i.e. in the frequency domain, the image is represented as a mapping from frequency values to the characteristics of those sinusoids within the given image. This means that unlike in the spatial domain, where the features which CNNs detect are mostly coherent visual structures, in the frequency domain, a CNN could detect abstract patterns in the interaction of sinusoids of neighbouring frequencies. We hope to exploit this difference, premising that spatial deformations do not alter frequency domain features as critically as they do alter spatial domain features. However, our goal is not to simply insert a preprocessing step, which applies the FFT, into an existing state-of-the-art CNN architecture. Rather, we want to find methods that make use of the specific characteristics of the frequency domain representation, and combine them with more simplistic CNNs.

1.3 Problem Statement

In order to work out whether this approach is feasible at all, we simplify the problem by considering one-dimensional signals instead of two-dimensional images. Since any one-dimensional array can be interpreted as an image with height (or, alternatively, width) equal to one, this simplification does not disqualify our methods from being generalized to images with height (or width, respectively) greater than one. However, this simplification does restrict the number of spatial deformations we can choose from; for example, rotations do not exist for one-dimensional signals. In this thesis, we will focus on one specific transformation which is equally important for both 1D and 2D: the translation.

At this point, it is important to note that it is often said that CNNs are, in fact, already invariant to translations in the input. And indeed, if an object is shifted by an integer amount of pixels within an image, then the CNN's feature detectors will find the same features in both the orignal image and the image containing the shifted object, only at the shifted locations. So, in this case, the translation commutes with the convolutions (this property of convolutions is also called *translation*

equivariance). Most CNNs then use pooling layers which subsample the feature maps by combining (e.g. take the maximum, or the average) small regions (usually of size 2×2) to a single output value (effectively halving the size of the feature map in the 2×2-case). This aims at making the exact location of the feature within that (2×2-)region irrelevant, and the main information that is forwarded by the pooling layer is that somewhere inside that small region, the feature was found. If the initial shift was small enough, then the ouput feature map is approximately the same for both the original image and the translated input image. Due to the cascading effect of many consecutive of such convolution and pooling operations, a CNN's output can be made mostly invariant to even larger shifts.

There are two problems with that strategy: First, the pooling operation simply discards the precise location of the feature. This important information could otherwise be used to make the network more efficient or even more accurate. The second problem is that this strategy only works well if the shift is equal to an integer amount of pixels. If the shift has a sub-pixel value, then, in general, it cannot be expected that the feature detectors will find the same features as before, because the shift changes the actual pixel intensity values. The conclusion is that arbitrary translations still pose a challenge to CNNs.

The task that we will deal with in this thesis is the task of signal decomposition: Many one-dimensional signals in the real world contain time-limited patterns that are the main interest of analyzing those signals. For example, in an ECG signal, which records the electrical activity of a heart over time, the individual heart beats are the time-limited patterns of major importance. In seismograms, which record ground motions, earthquakes create time-limited patterns. A **signal** that contains such a pattern can be decomposed into a **localization**, which transports the information at which point in time this pattern has occurred, and a **normalized pattern**, which transports the information what kind of pattern has occurred. This decomposition task can be split up into four individual sub-tasks:

1.) Reconstruction: Given the localization and the normalized pattern, we want to reconstruct the original signal.

2.) Localization: Given the signal and the normalized pattern, we want to find where exactly in the signal the normalized pattern can be found.

3.) Normalization: Given the signal and the localization, we want to extract a, in some sense, canonical representation of the pattern found at that location.

4.) **Simultaneous Localization and Normalization**: Given only the signal, we want to extract both the localization of the interesting pattern and, at the same time, a canonical representation of that pattern.

We will suggest solutions to all four of those problems.
Since we want to investigate methods that work in the frequency domain, we will use FFT to transfer the input(s) from the spatial domain into the frequency domain, and only there will we use CNNs (see also Figure 1.2). Usually, the normalized pattern array will contain the important pattern at the beginning of the array (i.e. the pattern is "normalized" in the sense that it starts at time 0), and the localization is a translation that characterizes the relation between the normalized pattern and the signal (i.e. how much the normalized pattern needs to be shifted in time in order to align with the pattern in the original signal). This means that the localization contains critical information, thus it is pivotal for the accuracy of the decomposition that the solutions do not have weaknesses with respect to sub-pixel translations. The type of signals we will be considering during these tasks is Morse code. Obviously, the individual Morse letters are time-limited patterns, and since different letters can look very similar to each other, it is not trivial to decompose a Morse signal using convolutions and feature detectors.

1.4 Structure of the Thesis

The rest of the thesis is structured the following way:

- In Chapter 2, we will discuss related work, including ideas that tackle the problem CNNs have with spatial deformations, and approaches that use the frequency domain for finding solutions to computer vision tasks.
- Chapter 3 describes the most relevant aspects from Fourier Analysis, such as the DFT, the FFT and some elemental properties of the frequency domain.
- Chapter 4 specifies the Morse signal dataset that we will use for testing and evaluating our suggested solutions.
- Chapter 5 represents the core of this thesis, as it contains the description and evaluation of our suggested signal decomposition modules.
- Chapter 6 gives a small example of how to extend one of our frequency domain modules to the 2D case.
- Chapter 7 summarizes the insights and results of this thesis.

All figures in this thesis, including the drawing of a face in Figure 1.1, were created by the author.

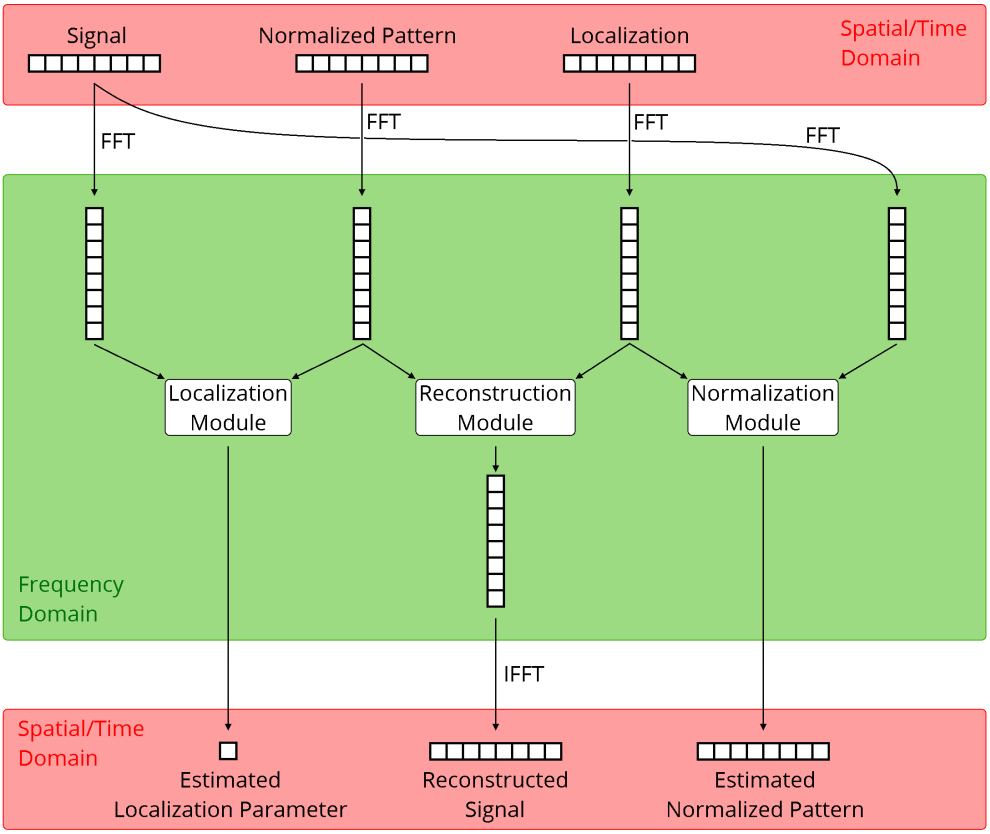


Figure 1.2: Illustration of the signal decomposition tasks: In the spatial domain, we are given two of the following three arrays: a signal, a normalized version of an interesting pattern in that signal, and the localization of that pattern within that signal. We apply an FFT to the given arrays and need to find a way in the frequency domain that enables us to retrieve an accurate approximation of the missing array, i.e. we need to find out how to implement the Localization module, the Reconstruction module and the Normalization module. Note that the Localization module does not need to output a full array that approximates the localization, it suffices for this module to output a single number that characterizes the point in time where the interesting pattern occurred. Also, notice that the Reconstruction module is the only one whose output needs to be manually transferred back into the spatial domain by an inverse FFT (IFFT). The two other modules directly output spatial domain objects. Not drawn in the figure is a fourth module, which gets as input only the signal and needs to output both an estimated localization parameter and an estimated normalized pattern. This module will be a combination of the Localization module and the Normalization module.

2 Related Work

In this chapter, we give a short overview about the related work. Particularly, methods that attempt to resolve the problem CNNs have when dealing with spatial deformations and frequency domain methods in computer vision.

2.1 Convolutional Neural Networks versus Spatial Deformations

Convolutional Neural Networks have set the state-of-the-art for numerous computer vision tasks, e.g. DenseNet [16] for image classification, YOLO [30] for object detection and Mask R-CNN [14] for object instance segmentation. All of these CNN architectures have one thing in common: during training, they all rely on extensive data augmentation in order to increase their capability to model spatial deformations. Data augmentation means that for each training sample, a spatial transformation is randomly sampled (e.g. translation, scaling, rotation, mirroring) and applied to the input data before it gets passed to the CNN, thereby increasing the variety of possible spatial deformations in the training data.

Other, more elegant solutions to improve the ability of CNNs to handle spatial deformations include Spatial Transformer Networks [17], Deformable Convolutional Networks [7, 48], Capsule Networks [15, 33], and Scattering Networks [3]. Spatial Transformer Networks [17] try to transform the input data into a somewhat canonical representation. A so-called *localisation net* is trained to compute transformation parameters from an input feature map. The parameters are then used to generate a grid, and finally, the input feature map is sampled at the grid points, producing the warped feature map. This largely eliminates the variety of possible spatial deformations in the data which a subsequent CNN would encounter otherwise.

Deformable Convolutional Networks, introduced by Dai et al. [7] and improved by Zhu et al. [48], try to break up the geometric limitations introduced by rectangular feature detectors. For each convolution, they additionally train a predictor for offsets, and these offsets are then used to effectively deform the receptive field of the individual entries in the output feature map. This leads to the output feature

maps being, to a certain level, independent from spatial deformations in the input feature map.
Both Spatial Transformer Networks and Deformable Convolutional Networks require a large number of random accesses in the input data, which we try to avoid in this thesis because they are expensive to implement in hardware and there is no evidence that the visual cortex of the human brain does something similar [18].
In Capsule Networks [15], the feature maps do not only contain, for each pixel, a confidence value which tells us that a certain feature is present there, rather they are built out of so-called *capsules* which contain both the information that a certain feature is present and, additionally, some information about the inner state of the feature. This inner state includes parameters about the spatial transformation of the feature, meaning that Capsule Networks explicitly try to understand spatial transformations, and use this information in a process called *dynamic routing* [33] to decide which higher-level capsule should have access to the information of which lower-level capsule. Capsule Networks do not exclude the use of the frequency domain, in fact, it might be possible to combine frequency domain methods with a Capsule Network architecture.
Scattering Networks [3] use a CNN with fixed kernels to perform wavelet scattering. The result is a representation of the input that is invariant to some transformations e.g. translations and rotations. So, a CNN that works on top of this representation does not need to model spatial deformations. While this is useful for some tasks, like image classification, other tasks, like object localization, cannot profit from translation invariance.

2.2 Computer Vision and the Frequency Domain

While the frequency domain is widely used in image processing (e.g. filtering [32], denoising [4], and compression [41]), only few attempts have been made to use of the characteristics of the frequency domain representation in order to solve computer vision tasks. Some approaches use hand-crafted frequency domain features for classification or detection tasks [9, 20, 26], but in more recent times, the frequency domain is almost exclusively used, in the context of computer vision, as a method to accelerate the calculation of convolutions [2, 23, 24] using the convolution theorem (which we will elaborate in Section 3.3.5).
However, there is one computer vision task where using the frequency domain is very common: image registration. This is very interesting for this thesis, because our localization task is closely related to image registration. While there are many different possibilities to perform image registration [49], there exists one

class of methods that is based on the phase correlation [21], which is a frequency-domain-based approach for extracting the translational offset between two images. However, in its most basic version, phase correlation cannot give sub-pixel accuracy, and can only determine translations. Reddy et al. [29] and Chen et al. [5] use a log-polar transform on the frequency domain representation of the input images which converts scaling and rotation into translations, thereby enabling them to also determine scaling and rotation parameters using phase correlation. Foroosh et al. [10] derive an analytical method that improves phase correlation to be capable of sub-pixel registration. Takita et al. [38] realize sub-pixel registration by using the *Phase-Only Correlation* and fitting a peak model to it. Stone et al. [37] achieve sub-pixel accuracy by fitting a hyperplane through the phase spectrum of the phase correlation. This last idea does not require any inverse FFTs and serves as inspiration for one of our localization modules.

There are also some works that use the frequency domain in combination with CNNs, for example:
Wang et al. [42] use the frequency domain for compressing CNNs in order to increase efficiency, not only in terms of time performance but also in terms of required storage space.
Yao et al. [45] use the frequency domain representation of sensor data as input for their combined CNN/RNN architecture because they assume that this representation contains more useful patterns than the spatial domain representatation.
Rippel et al. [31] study the frequency domain representation of convolutional kernels and use this *spectral parametrization* to learn the filters directly in the frequency domain. Additionally, they introduce a *spectral pooling* layer which crops the frequency domain representation, thereby reducing the feature map size, but without throwing away as much information as spatial pooling operations usually do. Still, this approach maintains the standard CNN structure without trying to customize it for the special characteristics of the frequency domain representation of the input data.
A recent work by Yao et al. [46] also adapts CNN layers using the frequency domain, with the goal of making them more compatible with the structure of the frequency domain representation of the input data. Additionally, they use the short-time Fourier transform and employ a procedure called *hologram interleaving* that helps to maintain both a high time and a high frequency resolution.
However, all of these works still rely on standard CNN architectures. The approach that is closest to the idea of this thesis is the one by Farazi et al. [13]: In the context of video prediction, the authors use phase correlation combined with a small neural network to predict a transformation which, when applied to the

current frame, generates an estimate of the next frame. In contrast, our method focuses on decomposing a signal in a single-shot manner.

2.3 Signal Decomposition

In most literature, the term *signal decomposition* refers to either decomposing a signal into its spectral components (which is usually done by e.g. Fast Fourier Transform, Discrete Cosine Transform, or Wavelet Transform), or it refers to separating different, superimposed signals from each other. In this thesis, however, we decompose signals into semantic components: localizations, which point at interesting regions in the signal, and normalized patterns, which tell us exactly what is so interesting about those regions.

As we have already mentioned above, the task of extracting the localization is similar to image registration, which is a well-studied challenge with numerous approaches to solve the task [49].

The task of extracting the normalized pattern, on the other hand, is not as straightforward, because it requires us to extract a somehow canonical representation of the pattern, while getting rid of any other pattern that might be present in the signal. Spatial Transformer Networks [17] try to perform some sort of normalization, but they can only predict global parametric transformations, thus not necessarily removing additional patterns, and use an expensive warping method. Considering the difference between the pattern in the original signal and the normalized pattern as noise, the normalization task becomes a very special denoising task. The normalized pattern can then be recovered using Denoising Autoencoders [39], a method that finds, in a first step called *encoding*, a very compact representation of the input, and then, in a second step called *decoding*, unfolds that compact representation again to get an output of the same size as the input. The idea is that the compact representation is noise-invariant, s.t. the model outputs a noise-free version of the input. Vincent et al. [40] and Xie et al. [44] use deep neural networks where, during the encoding step, the layers become progressively smaller, eventually reaching a bottleneck (i.e. the compact representation of the input), and then, during decoding, the layers become larger again until they reach the same size as the input. Zhu et al. [47] and Gondara [12] also use deep neural networks, but in a convolutional architecture, which is an idea that we will also use for one of our normalization modules.

To our knowledge, no method exists that is capable of doing something similar to extracting both localization and normalized pattern simultaneously.

3 Elemental Fourier Analysis

Transforming an one-dimensional input signal from the spatial domain to the frequency domain is a central idea of this thesis. To perform this operation, we will be using the **Fast Fourier Transform**, which itself is an efficient way of computing the **Discrete Fourier Transform** of a signal. In this chapter, we will describe these operations as well as the most relevant characteristics of the frequency domain representation. Smith [35] gives a good overview about these aspects of elemental Fourier Analysis.

3.1 Discrete Fourier Transform

The Discrete Fourier Transform (DFT) takes as input a finite sequence of complex numbers and maps it to another complex sequence of the same length.

Definition 3.1 (Discrete Fourier Transform):
Let $A = \{a_j\}_{j=0,\dots,N-1} \subset \mathbb{C}$ be a complex sequence of length $N \in \mathbb{N}$. The **Discrete Fourier Transform** of A, written as $\mathrm{DFT}(A)$, is the complex sequence $\mathrm{DFT}(A) = \{\widehat{a}_n\}_{n=0,\dots,N-1} \subset \mathbb{C}$ given by the formula

$$\widehat{a}_n = \sum_{k=0}^{N-1} a_k \cdot \exp\left(\frac{-i2\pi nk}{N}\right) \qquad \text{for } n = 0, \dots, N-1. \tag{3.1}$$

Using above notation, for each $n = 0, \dots, N-1$, the coefficients

$$\exp\left(\frac{-i2\pi nk}{N}\right) = \cos\left(\frac{2\pi nk}{N}\right) - i\sin\left(\frac{2\pi nk}{N}\right)$$

correspond to complex sinusoids of the form $\omega_n : x \mapsto \cos(2\pi nx) - i\sin(2\pi nx)$. More precisely, ω_n is a complex sinusoid with frequency n, and for each $k = 0, \dots, N-1$, the coefficient $\exp(-i2\pi nk/N)$ is obtained by evaluating $\omega_n(x)$ at the point $x = k/N$.

This leads to the intuition that the n-th component of $\mathrm{DFT}(A)$,

$$\widehat{a}_n = \sum_{k=0}^{N-1} a_k \cdot \omega_n(k/N),$$

contains information about how the sinusoid ω_n contributes to the input array A. This intuition becomes even more obvious when looking at the inverse DFT:

Definition 3.2 (Inverse Discrete Fourier Transform):
Let $\widehat{A} = \{\widehat{a}_n\}_{n=0,\dots,N-1} \subset \mathbb{C}$ be a complex sequence of length $N \in \mathbb{N}$. The **Inverse Discrete Fourier Transform** of $\widehat{A}$, written as $\mathrm{IDFT}(\widehat{A})$, is the complex sequence $\mathrm{IDFT}(\widehat{A}) = \{a_j\}_{j=0,\dots,N-1} \subset \mathbb{C}$ given by the formula

$$a_j = \frac{1}{N} \sum_{k=0}^{N-1} \widehat{a}_k \cdot \exp\left(\frac{i2\pi jk}{N}\right) \qquad \text{for } j = 0, \dots, N-1. \tag{3.2}$$

If, in Definition 3.2, the sequence $\widehat{A}$ has previously been obtained by applying a DFT to an input sequence A, then Equation (3.2) tells us that the j-th component of A can be reconstruced as the dot product of $N^{-1} \cdot \mathrm{DFT}(A)$ and the vector $\Omega = (\overline{\omega}_0(j/N), \overline{\omega}_1(j/N), \dots, \overline{\omega}_{N-1}(j/N))^T$, where $\overline{\omega}_k(j/N)$ are the complex conjugates of the sinusoids ω_k evaluated at the points j/N. This again illustrates that $\mathrm{DFT}(A)$ tells us how to combine the sinusoids ω_n in order to recover the input array A. It also illustrates why we use term *frequency domain* to refer to the space in which the output $\mathrm{DFT}(A) = \{\widehat{a}_n\}_{n=0,\dots,N-1}$ of a DFT lives: For a frequency value f, the component $\widehat{a}_f$ gives information about the characteristics of the sinusoid ω_f within the input sequence A. While, theoretically, the frequency value f can be any real number, we will only come across integer frequency values in our signal decomposition computations.

3.2 Fast Fourier Transform

The straightforward way to implement the DFT is to use a matrix multiplication. If the input sequence is written as N-dimensional vector $A = (a_0, a_1, \dots, a_{N-1})^T$, then the output vector $\mathrm{DFT}(A) = (\widehat{a}_0, \widehat{a}_1, \dots, \widehat{a}_{N-1})^T$ can be calculated as $\mathrm{DFT}(A) = WA$, where the DFT matrix W is given by

$$W = \left(\exp\left(\frac{-i2\pi jk}{N}\right)\right)_{\substack{j=0,\dots,N-1 \\ k=0,\dots,N-1}}.$$

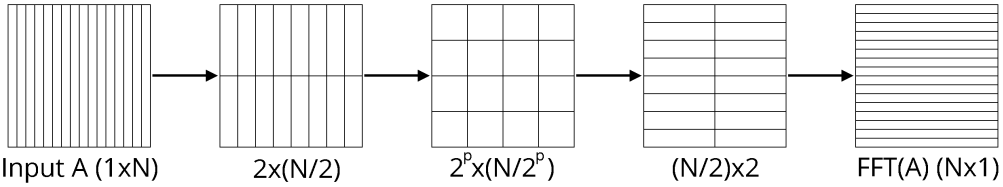


Figure 3.1: Illustration of the FFT. For an input A of length $N = 2^m$ for some $m \in \mathbb{N}$, the algorithm computes two-dimensional arrays, where the output of the p-th FFT step, $1 \leq p \leq m$, has size $2^p \times N/2^p$. The last array, which is the output $\mathrm{FFT}(A)$, has size $N \times 1$, and is equivalent to $\mathrm{DFT}(A)$.

The matrix multiplication requires N times N multiplications and N times $N-1$ additions, thus the complexity of this matrix multiplication is $\mathcal{O}(N^2)$.
Several approaches exist to speed up the calculation of the DFT, the most common being the method developed by Cooley and Tukey [6]; this method is what we will refer to as Fast Fourier Transform (FFT).
Given a complex input sequence $A = \{a_j\}_{j=0,\ldots,N-1} \subset \mathbb{C}$ of length $N = 2^m$ for some $m \in \mathbb{N}$, the way the FFT works is usually described the following way: the FFT splits A up into a subsequence of even indices, $A_{even} = \{a_{2j}\}_{j=0,\ldots,N/2-1}$ and a subsequence of odd indices, $A_{odd} = \{a_{2j+1}\}_{j=0,\ldots,N/2-1}$. These subsequences both have length $N/2$. The algorithm then calculates the DFT of both A_{even} and A_{odd}, and then combines these two (using factors of the form $\pm\exp(-i2\pi k/N)$) to get $\mathrm{FFT}(A)$. The DFTs of A_{even} and A_{odd} are computed recursively using the FFT approach again. Since it is trivial to calculate the DFT of an array of length two, the recursion is guaranteed to finish after $m = \log_2(N)$ levels of recursion.

Alternatively, we illustrate the FFT process the other way around (see also Figure 3.1): Starting with a complex input sequence $A = \{a_j\}_{j=0,\ldots,N-1} \subset \mathbb{C}$ of length $N = 2^m$ for some $m \in \mathbb{N}$, we write it as a $1 \times N$-matrix. The first step of the FFT then produces a $2 \times N/2$ -matrix A_1, where for the computation of each entry of A_1, only two entries of A are used. The second step then produces a $4 \times N/4$ -matrix A_2, where, again, for each entry of A_2, only two entries of A_1 are accessed. This happens m times, where the p-th step (for $1 \leq p \leq m$) produces a $2^p \times N/2^p$ -matrix, until the m-th step outputs a $N \times 1$-matrix which is equivalent to $\mathrm{DFT}(A)$.
The exact computation works as follows: For an arbitrary p, $1 \leq p \leq m$, the input matrix of the p-th FFT step has size $2^{p-1} \times N/2^{p-1}$:

$$\text{Input} = \underbrace{\left(\begin{array}{c} \ldots \end{array}\right)}_{N/2^{p-1} \text{ columns}} \Bigg\} 2^{p-1} \text{ rows}$$

The input matrix is then cut in half:

$$\text{Input}_{left} = \underbrace{\left.\begin{pmatrix} | & | & & | \\ \text{Inp}_1 & \text{Inp}_2 & \dots & \text{Inp}_{N/2^p} \\ | & | & & | \end{pmatrix}\right\}}_{N/2^p \text{ columns}} 2^{p-1} \text{ rows}$$

and

$$\text{Input}_{right} = \underbrace{\left.\begin{pmatrix} | & | & & | \\ \text{Inp}_{1+N/2^p} & \text{Inp}_{2+N/2^p} & \dots & \text{Inp}_{N/2^{p-1}} \\ | & | & & | \end{pmatrix}\right\}}_{N/2^p \text{ columns}} 2^{p-1} \text{ rows}$$

where Inp_k is the k-th column of the input matrix. The parameter matrix P for the p-th FFT step is defined as:

$$P = \underbrace{\left.\begin{pmatrix} | & | & & | \\ \Omega & \Omega & \dots & \Omega \\ | & | & & | \end{pmatrix}\right\}}_{N/2^p \text{ columns}} 2^{p-1} \text{ rows}, \qquad \text{with} \qquad \Omega = \begin{pmatrix} \exp(-i2\pi\frac{0}{2^p}) \\ \exp(-i2\pi\frac{1}{2^p}) \\ \vdots \\ \exp(-i2\pi\frac{2^{p-1}-1}{2^p}) \end{pmatrix}.$$

Finally, the output of the p-th FFT step is computed:

$$\text{Output} = \underbrace{\left.\begin{pmatrix} \text{Input}_{left} + P \circ \text{Input}_{right} \\ \hdashline \\ \text{Input}_{left} - P \circ \text{Input}_{right} \end{pmatrix}\right\}}_{N/2^p \text{ columns}} 2^p \text{ rows}$$

where $P \circ \text{Input}_{right}$ is the componentwise complex product of the two matrices. The whole computation consists of $2^{p-1} \cdot N/2^p = N/2$ multiplications and $2^p \cdot N/2^p = N$ additions, meaning that the p-th FFT step requires $\mathcal{O}(N)$ operations. Since there are $m = \log_2(N)$ of these FFT steps necessary to reach the final output, the FFT has an overall complexity of $\mathcal{O}(N \log(N))$.
The inverse FFT (IFFT) works almost the same way, with the exception that the

parameter matrices P_{IFFT} are the complex conjugates of the parameter matrices P used in the FFT, and the output matrix at the end of each IFFT step needs to be scaled by $\frac{1}{2}$.

3.3 Characteristics of the Frequency Domain Representation

3.3.1 Inclusion of negative Frequencies

As we have seen, the DFT (and thereby also the FFT) maps an input array $A = \{a_j\}_{j=0,\dots,N-1} \subset \mathbb{C}$, with $N = 2^m$ for some $m \in \mathbb{N}$, to complex coefficients $\{\widehat{a}_n\}_{n=0,\dots,N-1} \subset \mathbb{C}$, where $\widehat{a}_n$ contains information about how the complex sinusoid $\omega_n : x \mapsto \cos(2\pi n x) - i \sin(2\pi n x)$, which has frequency n, contributes to A. It does this by sampling the ω_n at the points k/N for $k = 0, \dots, N-1$ and using these values as coefficients in the DFT formula Equation (3.1). However, due to aliasing effects, the values $\omega_n(k/N)$ are equal to $\omega_{n-N}(k/N)$:

$$\begin{aligned}
\omega_{n-N}(k/N) &= \exp(-i2\pi(n-N)k/N) & &= \frac{\exp(-i2\pi nk/N)}{\exp(-i2\pi Nk/N)} \\
&= \frac{\exp(-i2\pi nk/N)}{\exp(-i2\pi k)} & &= \frac{\exp(-i2\pi nk/N)}{1^k} \\
&= \exp(-i2\pi nk/N) & &= \omega_n(k/N),
\end{aligned}$$

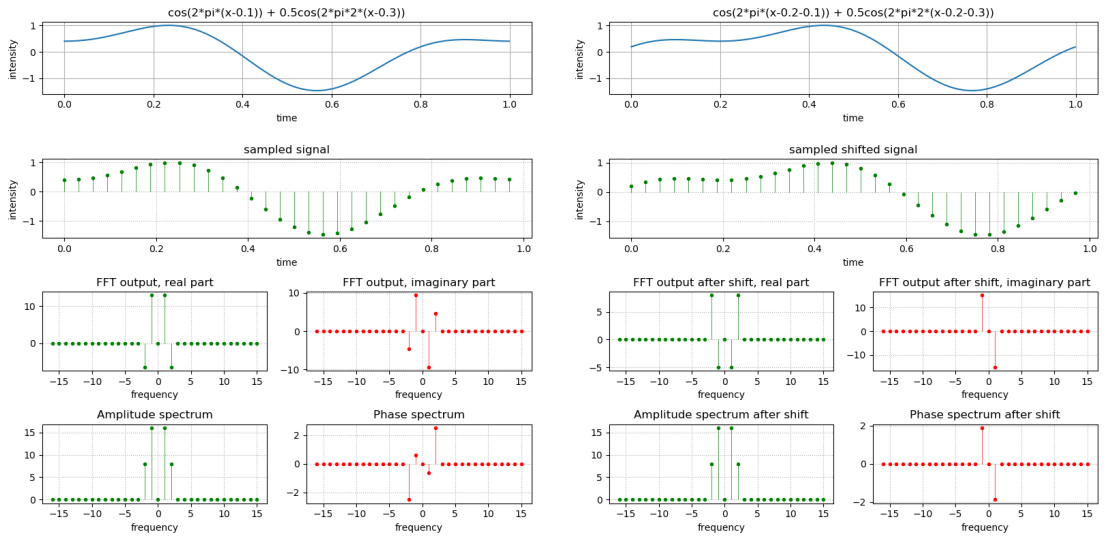
which is why the second half of the resulting FFT values $\{\widehat{a}_n\}_{n=N/2,\dots,N-1}$ are often interpreted as belonging to the sinusoids ω_{n-N}, covering the negative frequencies from $-N/2$ up to -1. So, $\text{FFT}(A) = \{\widehat{a}_n\}_{n=0,\dots,N-1}$ is often reordered to $\text{FFT}_{\text{reorder}}(A) = \{\widehat{a}_{g(n)}\}_{n=0,\dots,N-1}$ with

$$g(n) = \begin{cases} n + N/2, & \text{if } n < N/2 \\ n - N/2, & \text{else,} \end{cases}$$

s.t. $\text{FFT}_{\text{reorder}}(A)$ is correctly ordered from frequency $-N/2$ up to frequency $N/2 - 1$. We call this reordering operation *fftshift*, in the style of NumPy[1] and MATLAB[2].

[1] `https://docs.scipy.org/doc/numpy/reference/generated/numpy.fft.fftshift.html`
[2] `https://www.mathworks.com/help/matlab/ref/fftshift.html`

3.3.2 Interpretation of the FFT Result



(a) FFT result for example signal s.

(b) FFT result for s_{shifted}.

Figure 3.2: Simple example of the FFT output. In (a), the continuous input signal is $s(t) = \cos(2\pi x - 0.2\pi) + 0.5\cos(2\pi 2x - 1.2\pi)$. The signal is sampled at 32 points and the FFT is applied to the discrete signal. The figure shows in the third row the real and imaginary part of the output of the FFT, reordered by *fftshift*; in the bottom row, the amplitude spectrum and the phase spectrum of the input signal, also reordered by *fftshift*, are shown. When the signal s is shifted by 0.2 to the right, i.e. $s_{\text{shifted}}(t) = s(t - 0.2)$, then the arrays shown in (b) are obtained.

The input array A for the FFT is usually a signal that is located in the spatial/time domain, and has been equidistantly sampled over some finite time interval. The complex output values of the FFT, $\{\widehat{a}_n\}_{n=0,\dots,N-1}$, that are located in the frequency domain, provide two important pieces of information:

1.) For each $n = 0, \dots, N-1$, the absolute value $|\widehat{a}_n| = \sqrt{\text{Re}(\widehat{a}_n)^2 + \text{Im}(\widehat{a}_n)^2}$ tells us, how strongly the sinusoid ω_n contributes to the input signal. This value is also called the *amplitude* of the frequency n, and plotting the amplitudes against the frequency values yields the **amplitude spectrum** of A.

2.) The *phase* value of $\widehat{a}_n$ is computed as $\arctan2\left(\text{Im}(\widehat{a}_n), \text{Re}(\widehat{a}_n)\right)$. It tells us how to shift the sinusoid ω_n along the time-axis in order to align ω_n with the input signal. Plotting the phases against the frequency values yields the **phase spectrum** of A.

Together, the amplitude spectrum and the phase spectrum make up the frequency domain representation of the signal A.

Figure 3.2a shows a simple example: The continuous input signal is $s(t) = \cos(2\pi x - 0.2\pi) + 0.5\cos(2\pi 2x - 1.2\pi)$, i.e. it is the sum of a cosine wave with frequency 1, amplitude 1, and phase -0.2π, and a second cosine wave with frequency 2, amplitude 0.5 and phase -1.2π. After sampling it at 32 points, the array A is obtained, which is visualized in the second row of Figure 3.2a. The computation of FFT(A) yields complex numbers, visualized in the third row of Figure 3.2a. In general, simply looking at the real and imaginary part of of the output coefficients of FFT(A) does not give much insight into the characteristic properties of s; however, after calculating the amplitude spectrum and phase spectrum (visualized in the last row of Figure 3.2a), the characteristics of s are revealed: The amplitude spectrum shows the value 16 for frequency 1, and the value 8 for frequency 2; this relates to the input signal, where the cosine of frequency 1 had double the amplitude of the cosine of frequency 2. The phase spectrum shows the value -0.2π for frequency 1, and for frequency 2, it show the value 0.8π, which is, due to the periodicity of sine and cosine, equivalent to a phase value of $-2\pi + 0.8\pi = -1.2\pi$. So, for frequency 1 and 2, the phase spectrum shows the exact phase values of the respective cosines in the input signal. The only other frequencies that do not give 0 amplitude and phase values are the frequencies -1 and -2. Recall that the FFT computes coefficients for the complex sinusoids $\omega_n : x \mapsto \cos(2\pi nx) - i\sin(2\pi nx)$. For an entirely real-valued signal, like s, the imaginary parts need to add up to 0; this can be done by assigning, to the negative frequencies, the complex conjugates of the FFT coefficients of the respective positive frequencies, and this leads to negative frequencies having the same amplitude values as the respective positive frequencies, and to negative frequencies having the negative values of the phase values of the respective positive frequencies.

3.3.3 Periodicity

Since all the complex sinusoids ω_n are periodic, the DFT implicitly sees the input signal as periodic as well, specifically it assumes that the discretized signal $A = \{a_j\}_{j=0,\dots,N-1}$ can be periodically continued to $a_{j+qN} = a_j$ for all $q \in \mathbb{Z}$. This

periodic extension is also present in the output $\mathrm{FFT}(A) = \{\widehat{a}_j\}_{j=0,\dots,N-1}$:

$$\begin{aligned}
\widehat{a}_{n+qN} &= \sum_{k=0}^{N-1} a_k \cdot \exp\left(\frac{-i2\pi(n+qN)k}{N}\right) \\
&= \sum_{k=0}^{N-1} a_k \cdot \exp\left(\frac{-i2\pi nk}{N}\right) \exp\left(\frac{-i2\pi qNk}{N}\right) \\
&= \sum_{k=0}^{N-1} a_k \cdot \exp\left(\frac{-i2\pi nk}{N}\right) \exp(-i2\pi qk) \\
&= \sum_{k=0}^{N-1} a_k \cdot \exp\left(\frac{-i2\pi nk}{N}\right) \cdot 1^{qk} \\
&= \widehat{a}_n \qquad\qquad \text{for } n = 0, \dots, N-1 \text{ and } q \in \mathbb{Z}.
\end{aligned}$$

3.3.4 Behaviour with respect to Translations

Since we are particularly interested in translations, we now investigate how the frequency domain representation of a signal changes when it is spatially shifted. First off, for pixel-wise shifts, the Shift theorem is applicable:

Theorem 3.3 (Shift Theorem):

Let $A = \{a_j\}_{j=0,\dots,N-1} \subset \mathbb{C}$ be a discretized input signal and $\widehat{A} = \{\widehat{a}_n\}_{n=0,\dots,N-1} \subset \mathbb{C}$ its FFT. Let $q \in \mathbb{Z}$. Then, for the shifted signal $B = \{b_j\}_{j=0,\dots,N-1} := \{a_{j+q}\}_{j=0,\dots,N-1}$, the FFT $\widehat{B} = \{\widehat{b}_n\}_{n=0,\dots,N-1}$ is given as

$$\widehat{b}_n = \exp\left(\frac{-i2\pi nq}{N}\right)\widehat{a}_n \qquad \text{for } n = 0, \dots, N-1.$$

Proof:

A proof can be found in most books that deal with DFT, e.g. [35], chapter 7. □

Note that the Shift Theorem makes use of the periodic extension discussed in Section 3.3.3 s.t. $b_j = a_{j+q}$ is well-defined for all $j = 0, \dots, N-1$. Since $\exp(-i2\pi nq/N)$ lies on the unit circle, this theorem implies that shifting an input signal pixel-wise in the spatial domain does only change the phase spectrum and leaves the amplitude spectrum untouched, i.e. the amlitude spectrum is invariant to pixel-wise shifts and all the translational information is encoded in the phase spectrum.

If the input signal is very simple (i.e. if it can be expressed as a combination of sines and cosines with integer frequencies smaller than half the sampling

frequency), like the signal s in Figure 3.2, then sub-pixel shifts still affect the frequency domain representation in a very well-defined manner: If the input signal is shifted by some value $\delta \in \mathbb{R}$, i.e. $s_{\text{shifted}}(t) := s(t - \delta)$, then all the sines and cosines in the input are shifted according to their frequency, i.e. if the phase value for frequency n is φ_n before the shift, then the phase value for frequency n after the shift is $\varphi_n - 2\pi n\delta$. In our example in Figure 3.2b, we chose $\delta = 0.2$. φ_1 was equal to -0.2π before the shift, and the phase spectrum shows the value $-0.2\pi - 2\pi \cdot 1 \cdot 0.2 = -0.6\pi$ afterwards. Likewise, for frequency 2, the phase changed from -1.2π to $-1.2\pi - 2\pi \cdot 2 \cdot 0.2 = -2\pi$, which is equivalent to 0 due to the periodicity of sine and cosine, which is also the value computed in the phase spectrum after the shift. For the negative frequencies -1 and -2, the same considerations as before apply, so they still show the negative values of the phase values of the respective positive frequencies. The amplitude spectrum is not altered in any way.

For more complex signals, we expect sub-pixel shifts to introduce more serious effects, like aliasing, on both the amplitude spectrum and the phase spectrum, but a goal of this thesis is exactly to investigate whether these effects are as fatal as the effects a sub-pixel shift has on the spatial representation.

3.3.5 Convolution Theorem

The one-dimensional discrete convolution is defined the following way:

Definition 3.4 (Convolution):

Let $A = \{a_j\}_{j=0,\dots,N-1}$ be a discretized input signal and $K = \{k_j\}_{j=0,\dots,N-1}$ be a kernel. The n-th component of the convolution $A * K$ is then defined as

$$(A * K)_n = \sum_{m=0}^{N-1} a_{n-m} k_m. \tag{3.3}$$

For indices $l < 0$ or $l > N - 1$, the values a_l have to be manually defined, depending on the application. For Convolutional Neural Networks (CNNs) , these values are taken from a user-defined padding function, e.g. each a_l is assigned zero, or $a_l = a_0$ for $l < 0$ and $a_l = a_{N-1}$ for $l > N - 1$. Usually, the kernels are much smaller in size than the input array; in that case, in Equation (3.3), k_m can be treated as zero if m is larger than the actual kernel size.
CNNs perform a large number of convolutions, and calculating the sum in Equation (3.3) for every single output value can become very computationally expensive. So, some implementations make use of the frequency domain in order to accelerate the computation of the convolution, making use of the convolution theorem:

Theorem 3.5 (Convolution Theorem):

Let $A = \{a_j\}_{j=0,\dots,N-1}$ be a discretized input signal and $K = \{k_j\}_{j=0,\dots,N-1}$ be a kernel. The convolution $A * K$ is equivalent to a componentwise multiplication in the frequency domain:

$$A * K = \text{IFFT}(\text{FFT}(\text{A}) \circ \text{FFT}(\text{K})),$$

where $\circ$ denotes the componentwise complex multiplication.

Proof:

A proof can be found in most books that deal with DFT, e.g. [35], chapter 7. □

If the number of convolutions is large and/or the input arrays A are very long, then the simple pointwise multiplication will outperform the naive sliding-window calculation of the sums in Equation (3.3), even if the cost for the FFTs and IFFTs is included.

In order to simplify this part, we only considered one-dimensional convolutions, but analogous statements also apply to two-dimensional convolutions.

4 Morse Signal Dataset

In order to measure the success (or failure) of our solutions, we will be using a Morse signal dataset. An already existing Morse code dataset would have been the one made by Dey et al. [8], but their implementation does not support sub-pixel shifts. So, we implemented our own Morse signal dataset. In this section, we will explain the construction of our dataset.

4.1 Base Dataset

Our implementation follows the standards of the International Telecommunication Union [28]: A morse letter is a combination of dots • and dashes —; Table 4.1 shows the individual letter encodings. A dash is three times the length of a dot, and each pause inside of a single letter has the same length as a dot. The length of the pause inbetween two letters is equal to three dots. The length of the pause inbetween two words is seven dots.

a	• —	n	— •
b	— • • •	o	— — —
c	— • — •	p	• — — •
d	— • •	q	— — • —
e	•	r	• — •
f	• • — •	s	• • •
g	— — •	t	—
h	• • • •	u	• • —
i	• •	v	• • • —
j	• — — —	w	• — —
k	— • —	x	— • • —
l	• — • •	y	— • — —
m	— —	z	— — • •

Table 4.1: Morse codes for the English alphabet.

In our implementation, the Morse signals are located on a timeline that starts at 0 and ends at 1. The length of a dot is 0.03, and therefore, the length of a pause

inside a letter is also 0.03, while the length of a dash is $3 \cdot 0.03 = 0.09$.

Each sample is constructed the following way: First, it is uniformly randomly chosen which of the 26 letters of the English alphabet is used for that sample. Then, the translation τ is uniformly randomly chosen from the interval $[0, 1 - l]$, where l is the length of the chosen letter (e.g. the letter "a" is equal to "• —", which is a dot (length: 0.03), an inner-letter pause (length: 0.03) and a dash (length: 0.09); so, the letter "a" has length $l = 0.03 + 0.03 + 0.09 = 0.15$). This ensures that the entire Morse letter is always inside the interval [0,1], as τ is set to be the starting point of the letter. Finally, dots and dashes are given the intensity value 1, and everything else 0. Using all these pieces of information, a continuous-time function of the Morse signal, $\Psi : [0, 1] \to \{0, 1\}$, is constructed (see Figure 4.2, the topmost row on the left). This function maps time values from the interval $[0, 1]$ to intensity values from the binary set $\{0, 1\}$.

The **signal** array is then generated by sampling that function at N equidistant points (see Figure 4.2, the second row on the left), where $N \in \mathbb{N}$ is the resolution of the discretization, and the distance between two adjacent sampling points is N^{-1}. This means that the i-th entry of the discrete signal array is given by $\Psi(j \cdot N^{-1})$ for $j = 0, \ldots, N - 1$. We use $N = 2^j$ for some $j \in \mathbb{N}$, because powers of two are the optimal resolution for the FFT.

The **localization** (see Figure 4.2, the third row on the left) has two possible representations: first, a single decimal number that is equal to the translation τ. Second, it can be represented as a Gaussian curve with mean equal to τ; we chose the standard deviation 0.004 in order to get a thin Gaussian curve. Due to this Gaussian curve definition, the localization is able to model uncertainties and the presence of more than one hypotheses for the ground truth, similar to *activity blobs* [1]. To get a discrete array, the Gaussian curve is also sampled at N equidistant points in $[0, 1]$.

Finally, the **normalized pattern** is generated. It is simply how the signal array would look like in the case of $\tau = 0$ and in the absence of noise and any additional Morse letters (see Figure 4.2, the fourth row on the left).

The advantage of setting up a continuous-time function of the already shifted letter, and then sampling a discretization from it, is that it avoids the necessity of an interpolation. The alternative would be to generate the discrete normalized pattern first, and then use the translation τ to shift the normalized pattern to its target position in order to get the signal. However, this would require interpolating the intensity values if the shift τ has a sub-pixel value (in this context, τ having sub-pixel value means that τ is not an integer multiple of the sampling interval N^{-1}), which is why we chose the former approach.

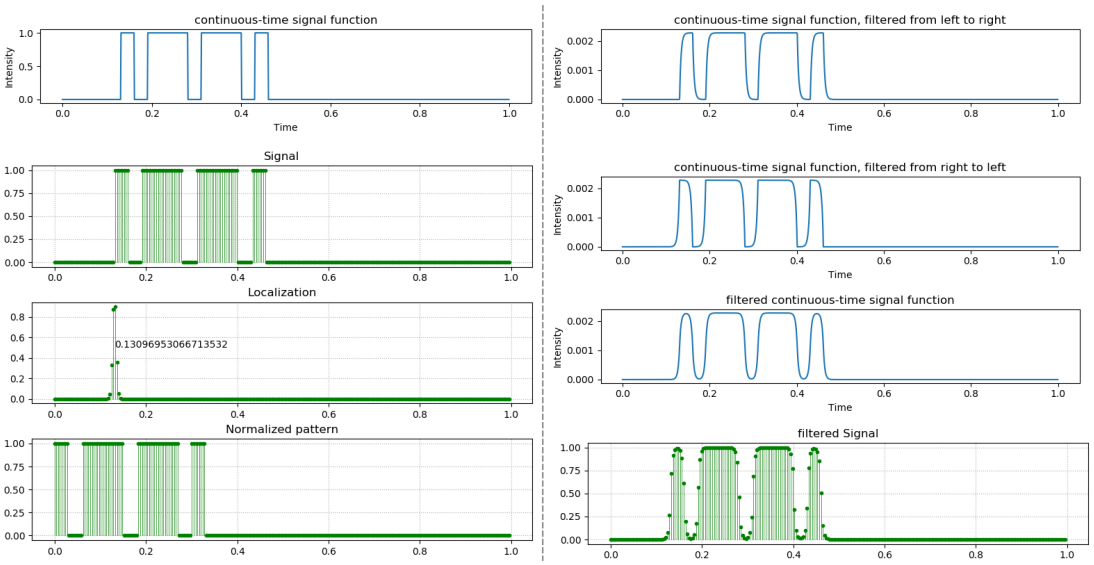


Figure 4.2: Morse code sample of the letter "p" (• — — •), with resolution $N = 256$. Left column: first row: continuous-time signal Ψ, the vertical lines are only added for visualization; second row: discretized signal; third row: localization represented as thin Gaussian, with the numerical shift value τ written next to the peak; fourth row: normalized pattern. Right column: first row: Ψ convolved with the left-to-right filter; second row: Ψ convolved with the right-to-left filter; third row: Ψ_{filtered}; fourth row: discretized and normalized Ψ_{filtered}.

4.2 Extensions

4.2.1 Smoothing

The Morse code dataset we described so far has a major drawback, namely that there are discontinuities at the locations where the intensity value changes from 0 to 1 and vice versa. This causes the Fourier decomposition of the underlying continuous-time signal to contain sinusoids of (possibly infinitely) large frequencies, which makes it impossible to meet the requirements of the Nyquist-Shannon Sampling Theorem [34], which enables aliasing effects to come into play. If the input signal is shifted by a sub-pixel value, then it might become difficult to distinguish between these aliasing effects and the modifications to the frequency domain representation introduced by the translation. So, we emulate an analog low-pass filter that smoothes the continuous-time Morse signal function. This will remove high frequencies, thereby acting as an anti-aliasing filter.
We do this by convolving the continuous-time Morse signal function $\Psi : [0, 1] \to \{0, 1\}$ with two filters and then computing the average of the two results. The

first filter, $K_{l2r} : \mathbb{R} \to \mathbb{R}$, smoothes Ψ from left to right, and the second filter, $K_{r2l} : \mathbb{R} \to \mathbb{R}$, smoothes Ψ from right to left. The two filters are defined the following way (see also Figure 4.3):

$$K_{l2r}(x) = \begin{cases} \alpha \left((1-\alpha)^N\right)^x & , \text{ if } x \geq 0 \\ 0 & , \text{ else.} \end{cases}$$

$$K_{r2l}(x) = \begin{cases} \alpha \left((1-\alpha)^N\right)^{-x} & , \text{ if } x \leq 0 \\ 0 & , \text{ else.} \end{cases}$$

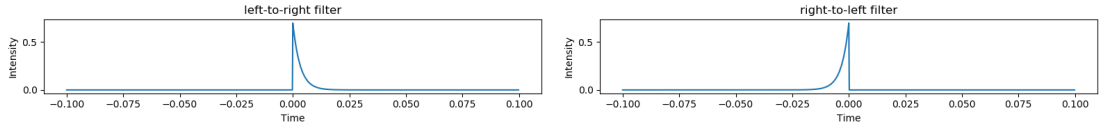


Figure 4.3: Exponential smoothing filters. Left is K_{l2r}, right is K_{r2l}. Note that the convolution flips the filters horizontally, so K_{l2r} assigns nonzero weights to the function values on the left of t (where t is the time value for which the convolution is evaluated), and K_{r2l} assigns nonzero weights to the function values on the right.

The parameter $\alpha \in (0, 1)$ influences, how strong the smoothing is; the smaller α is, the stronger the smoothing becomes. In our case, $\alpha = 0.7$ was chosen. N, as always, denotes the number of sampling points that will be used for discretization. The idea of these filters is similar to that of *exponential smoothing* (see [25], section 6.4.3.1.), i.e. for each input time t, the left-to-right filter computes a weighted average of the function value at t and the function values at the time values on its left, where the weight is smaller the further away the time value is from t. Similarly, the right-to-left filter uses the function values on the right of t for the weighted average.
The filtered continuous-time Morse signal function, which will be used for sampling, is then defined as

$$\begin{aligned} \Psi_{\text{filtered}}(t) &:= \frac{1}{2}((\Psi * K_{l2r})(t) + (\Psi * K_{r2l})(t)) \\ &= \frac{1}{2}\left(\int_0^1 \Psi(x) K_{l2r}(t-x)\mathrm{d}x + \int_0^1 \Psi(x) K_{r2l}(t-x)\mathrm{d}x\right) \\ &= \frac{1}{2}\left(\int_0^t \Psi(x) K_{l2r}(t-x)\mathrm{d}x + \int_t^1 \Psi(x) K_{r2l}(t-x)\mathrm{d}x\right), \end{aligned} \tag{4.1}$$

where, in the last equation, the integration limits have changed because $K_{l2r}(x) = 0$ for $x < 0$ and $K_{r2l}(x) = 0$ for $x > 0$. Since Ψ is equal to 1 only in some intervals, and 0 everywhere else, we can use the function $\Psi_{c,d}(x) = \begin{cases} \Psi(x) \ , \text{ if } x \in [c,d] \\ 0 \quad\quad , \text{ else} \end{cases}$, where $0 \leq c \leq d \leq 1$, to define

$$\mathcal{I}_{c,d} := \{[a,b] \subset [c,d] \mid \forall x \in [a,b] : \Psi_{c,d}(x) = 1 \text{ and } \exists \varepsilon > 0 : \forall x \in [a-\varepsilon, a) \cup (b, b+\varepsilon] : \Psi_{c,d}(x) = 0\}$$

which is the set of disjoint, maximum size intervals where $\Psi_{c,d}$ is equal to 1. Using this definition, we can write Equation (4.1) as:

$$\Psi_{\text{filtered}}(t) = \frac{1}{2}\left(\sum_{[a,b]\in\mathcal{I}_{0,t}} \int_a^b K_{l2r}(t-x)\mathrm{d}x + \sum_{[a,b]\in\mathcal{I}_{t,1}} \int_a^b K_{r2l}(t-x)\mathrm{d}x\right). \quad (4.2)$$

It is possible to analytically calculate the indefinite integrals of K_{l2r} and K_{r2l}, so Equation (4.2) can be computed via

$$\Psi_{\text{filtered}}(t) = \frac{1}{2}\left(\sum_{[a,b]\in\mathcal{I}_{0,t}} \left[\frac{\alpha\left((1-\alpha)^N\right)^{t-x}}{-N\log(1-\alpha)}\right]_{x=a}^{b} + \sum_{[a,b]\in\mathcal{I}_{t,1}} \left[\frac{\alpha\left((1-\alpha)^N\right)^{x-t}}{N\log(1-\alpha)}\right]_{x=a}^{b}\right).$$

For better consistency, after discretization, we normalize the discretized signal s.t. the maximum value of the array is 1. Figure 4.2, right column, shows an example of the filtered continuous-time signals.

4.2.2 Two Morse Signal Dataset

The second important extension is the **Two Morse signal dataset**. It simply adds a second Morse letter to each dataset sample. This dataset follows the rule that both letters need to be fully inside the $[0, 1]$ time interval, and the gap between the two Morse letters needs to be at least three dots (i.e. 0.09) wide, s.t. the two letters cannot be confused to be one large letter. Additionally, we disallow the same letter to appear twice in a sample, because this ambiguity could possibly lead to unwanted behaviour in the localization task (remember that the localization task consists of taking a signal and a normalized pattern as input; if the normalized pattern appears twice in the signal, then any solution for that task would have to deal with this ambiguity).
In the spatial domain, there is a clear separation between the two letters, but in the

frequency domain, both letters influence the amplitude spectrum and the phase spectrum, so this extension is designed to make it more challenging to use the frequency domain for the signal decomposition task. Note that this dataset also maintains, for each sample, a random number called LeftOrRight, which is either 0 or 1. When requesting a sample from a Two Morse signal datasest, it will give out the signal that contains the two letters, and then give out the localization and the normalized pattern of the left letter, if LeftOrRight = 0; if LeftOrRight = 1, then it will give out the normalized pattern and the localization that belong to the right letter. This means that any module that wants to perform well on such a dataset must be able to work well on both the left and the right letter. Figure 4.4 shows an example from a Two Morse signal dataset.

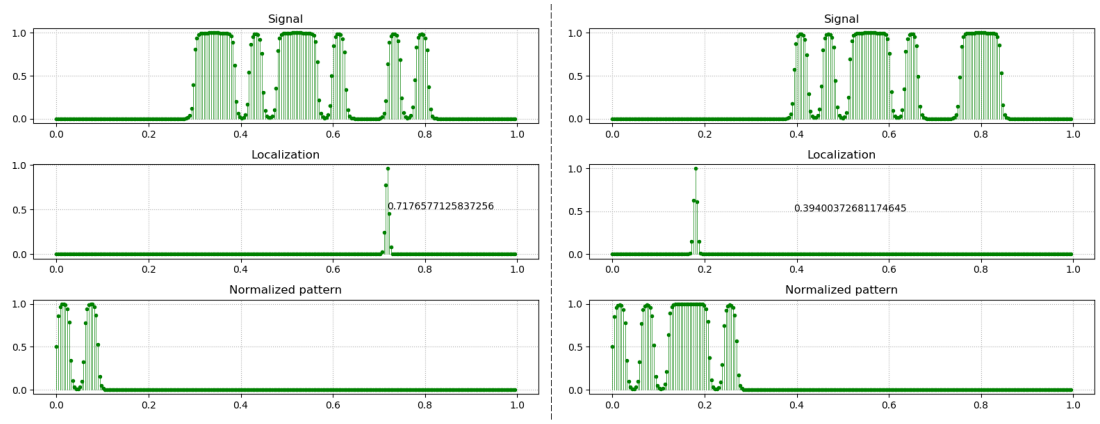


Figure 4.4: Two Morse signal dataset example. It can be seen that LeftOrRight=1 in the example on the left, and LeftOrRight=0 in the example on the right. Additionally, the example on the right is injected with localization noise, i.e. the peak of the localization is far away from the numerical ground truth.

4.2.3 Noisy Localization

For the normalization task (i.e. given a signal and a localization, the task is to find the normalized pattern at that location), it is useful to achieve robustness towards lateral localization noise. When the localization does only point at a rough estimate of the point in time where a Morse code letter in the signal starts, rather than at the exact position, then the normalization task becomes more difficult, the ground truth might even be ambiguous. We will make use of this extension in order to get a robust solution for the normalization task. In our implementation, we sample a random number from a normal distribution with mean zero and standard deviation 0.1. Due to the 68-95-99.7 rule of thumb for Gaussian distributions, this means that in roughly 99% of the cases, the localization noise will be inside the

interval $(-0.3, 0.3)$, which is not trivial given that all our signals are located in the time interval $[0, 1]$. If the noise would send the localization out of the interval $[0, 1]$, then we make use of the periodicity implied by the FFT (recall Section 3.3.3), and we cyclically project it back into the interval $[0, 1]$ (e.g. the noisy localization $-1 < \tau_{outside} < 0$ is projected to $1 + \tau_{outside}$, and $1 < \tau_{outside} < 2$ is projected to $\tau_{outside} - 1$). The signal array and the normalized pattern array are not affected by this extension, because we only inject noise to the localization after constructing the continuous-time signal function Ψ. Only the localization array is changed, i.e. the peak in the localization is shifted along the time-axis, away from the ground truth. The right column of Figure 4.4 shows an example with localization noise.

4.3 Evaluation Metrics

4.3.1 Loss Functions

Recall the tasks defined in Section 1.3. The first task was a reconstruction task, so we need to output an array that is approximately the same as the ground truth signal. Similarly, for the third task, the normalization task, we need to output an array that is approximately the same as the ground truth normalized pattern. For both of these tasks, we will measure the difference between our output and the ground truth using the **MSEloss** (Mean Squared Error loss): For two arrays $a, b \in \mathbb{R}^N$, the MSE loss is defined as

$$\text{MSEloss}(a, b) := \sum_{i=0}^{N-1} \frac{(a_i - b_i)^2}{N}.$$

The second task was the localization task. To solve this, we need to output a single number ρ that should be close to the ground truth translational value τ (note that we do not use the discretized thin Gaussian representation of the localization, because it is too inaccurate). However, we cannot simply use the absolute value $|\rho - \tau|$, because, due to the periodicity in the input signal implied by the FFT, $\rho = -0.1$ and $\tau = 0.9$ describe the same translation for the frequency domain representation. For this reason, we use a loss function that does not penalize combinations such as $\rho = -0.1$ and $\tau = 0.9$:

$$\text{Closs}(\rho, \tau) := \text{MSEloss}\left(\begin{pmatrix} \cos(2\pi\rho) \\ \sin(2\pi\rho) \end{pmatrix}, \begin{pmatrix} \cos(2\pi\tau) \\ \sin(2\pi\tau) \end{pmatrix}\right),$$

where **Closs** stands for Circular loss, because it uses ρ and τ as angles to produce two points on the unit circle and then measures the difference between those two 2D-points. Obviously, if e.g. $\rho = -0.1$ and $\tau = 0.9$, then both are mapped to the same point on the unit circle, and the loss is zero.
The fourth task was a combination of the normalization and the localization task, so in that task, we use both MSEloss and Closs to measure the difference between our output and the ground truth.

4.3.2 Explicit Datasets

At this point, we define some datasets that we will use throughout the evaluations in Chapter 5.

Testing datasets:

- $\mathbf{MSD}^{1000}$: A Morse signal dataset as defined in Section 4.1, containing 1000 samples and making use of the smoothing described in Section 4.2.1. The resolution of the discretization is 256, i.e. the discretized signal, the discretized localization and the discretized normalized pattern are arrays of length 256.

- $\mathbf{MSDL}^{1000}$: A Morse signal dataset as defined in Section 4.1, containing 1000 samples and making use of the smoothing described in Section 4.2.1. Additionally, the localization array is distorted as specified in Section 4.2.3. The resolution of the discretization is 256, i.e. the discretized signal, the discretized localization and the discretized normalized pattern are arrays of length 256.

- $\mathbf{TMSD}^{1000}$: A Two Morse signal dataset as defined in Section 4.2.2, containing 1000 samples and making use of the smoothing described in Section 4.2.1. The resolution of the discretization is 256, i.e. the discretized signal, the discretized localization and the discretized normalized pattern are arrays of length 256.

- $\mathbf{TMSDL}^{1000}$: A Two Morse signal dataset as defined in Section 4.2.2, containing 1000 samples and making use of the smoothing described in Section 4.2.1. Additionally, the localization array is distorted as specified in Section 4.2.3. The resolution of the discretization is 256, i.e. the discretized signal, the discretized localization and the discretized normalized pattern are arrays of length 256.

Training datasets:

- $\mathbf{MSD}^{\infty}$: Same as $\mathbf{MSD}^{1000}$, just with 10000 samples instead of 1000. Additionally, this dataset has a *randomize* functionality, which randomly assigns new letters and new translations τ to each sample. When training a neural network with this dataset, the *randomize* function will always be invoked after each training epoch, s.t. each epoch gets different samples than the previous epochs. This makes this dataset practically infinitely large for training purposes. Repititions are extremely improbable because the translations τ are uniformly randomly sampled real numbers.
- $\mathbf{TMSD}^{\infty}$: Same as $\mathbf{TMSD}^{1000}$, just with 10000 samples instead of 1000, and equipped with the same *randomize* functionality as $\mathbf{MSD}^{\infty}$. Additionally, the *randomize* function also randomizes the numbers LeftOrRight that are present in each Two Morse signal dataset.
- $\mathbf{TMSDL}^{\infty}$: Same as $\mathbf{TMSDL}^{1000}$, just with 10000 samples instead of 1000, and equipped with the same *randomize* functionality as $\mathbf{TMSD}^{\infty}$. Additionally, the *randomize* function also assigns new noise values to the localization noise.

5 Signal Decomposition Modules

In this chapter, we will present our approaches to solve the individual signal decomposition tasks described in Section 1.3. For each task, we will discuss the idea behind our suggested solutions, describe some implementation details if necessary, and evaluate our solutions using the Morse Signal dataset. The entire implementation, including the Morse signal dataset and all experiments, was done using the PyTorch deep learning library [27]. The tests were run on a 16-thread Intel® Core™ i9-9900K CPU @ 3.60 GHz with 64 GiB of RAM. Note that the convolution operation in PyTorch is highly optimized, likely more optimized than the FFT/IFFT routines, meaning that it might be difficult to implement a frequency domain method that is faster than a spatial CNN method.

5.1 Reconstruction Module

The **Reconstruction task** is defined the following way:

Given the localization and the normalized pattern, we want to reconstruct the original signal.

Since the original signal is not an input, this might be the least interesting task, because the localization and the normalized pattern already contain the most important semantic information about the signal.

5.1.1 Idea

This is the easiest of the four tasks: It is a well-known fact that convolving a signal with a delta peak delays the signal, i.e. the output of the convolution shows the input signal shifted to the time/location where the delta peak is. In our case, the localization is a discretized Gaussian curve instead of a delta peak, but this can be split up into several delta peaks: If the localization array is given as $L = \{l_i\}_{i=0,\dots,N-1}$, then L is simply a weighted sum of delta peaks:

$$L = \sum_{i=0}^{N-1} l_i \cdot \delta_i,$$

where δ_i is a sequence of length N whose i-th component is 1 and all other components are 0. Due to the linear properties of the convolution, the convolution between the normalized pattern P and the localization L can then be written as

$$P * L = \sum_{i=0}^{N-1} l_i \cdot (P * \delta_i)$$

So, $P * L$ is equal to the weighted sum of arrays, each of which contains a shifted version of P. Since we want the weights l_i to add up to 1, we first divide L by the sum of its components before convolving. Thus, our **Reconstruction Module** produces the following output:

$$\text{Output} = P * \frac{L}{\sum_{i=0}^{N-1} l_i}$$

The output is an approximation of the original signal. Figure 5.1a shows an example of this approach.

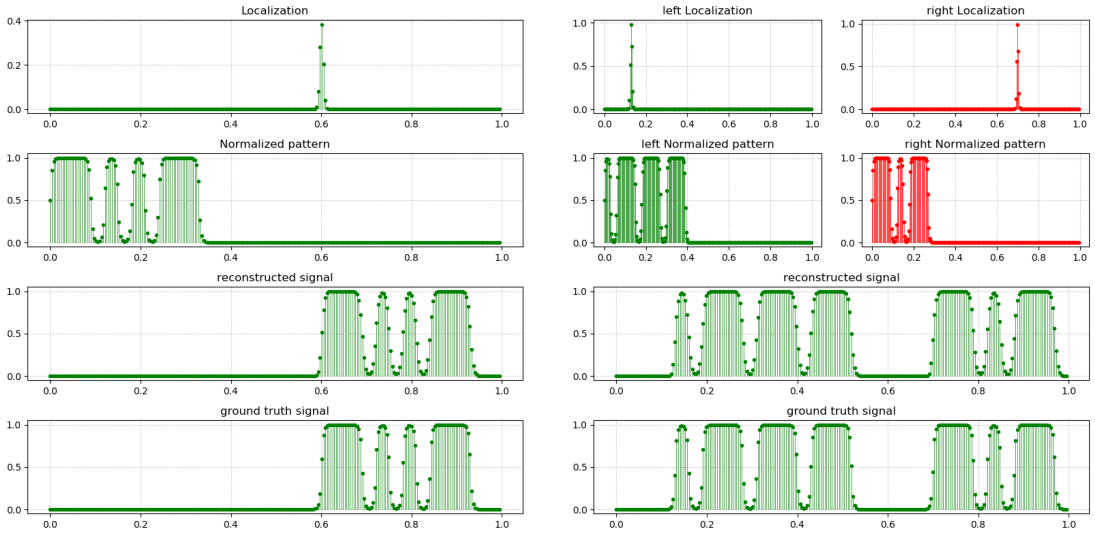
5.1.2 Implementation

As for the implementation, the only real choice here is how to perform the convolution. The first possibility to do this is to directly compute the convolution in the spatial domain, using Equation (3.3). However, using this formula makes it necessary to define a padding function: We simply pad the array P with zeroes. The second possibility is to use the detour through the frequency domain, where the convolution breaks down to a simple componentwise multiplication:

$$\text{Output} = \text{IFFT}\left(\text{FFT}(P) \circ \text{FFT}\left(\frac{L}{\sum_{i=0}^{N-1} l_i}\right)\right).$$

5.1.3 Evaluation

We tested the reconstruction module on two datasets: $\mathbf{MSD}^{1000}$ and $\mathbf{TMSD}^{1000}$, as defined in Section 4.3.2. The reconstruction error is measured using the MSELoss described in Section 4.3.1.
In order to reconstruct signals in the Two Morse signal dataset, we get as inputs both pairs of (localization, normalized pattern). Both pairs are sent individually through the reconstruction module, and the outputs are added together; see also Figure 5.1b for an example.



(a) Morse signal dataset example.

(b) Two Morse signal dataset example.

Figure 5.1: Reconstruction examples. (a) is a Morse signal dataset example. Shown are the localization L, the normalized pattern P, the output of the reconstruction module (i.e. $P * L$) and, in the last row, the ground truth signal. Note that the reconstructed signal is a bit blurrier than the ground truth, which is caused by the uncertainty introduced by using a Gaussian instead of a delta peak as localization. For the Two Morse signal dataset example in (b), the two normalized patterns are individually convolved with their respective localization, and then simply added componentwise to obtain the reconstruction.

Table 5.2 shows the result of our experiment. All of the reconstruction losses are in the magnitude of 10^{-4}, which means, by definition of the MSELoss, that for each tested sample, the average component-wise absolute difference between the output array of the module and the ground truth signal array lies around $\sqrt{10^{-4}} = 10^{-2}$. So, on average, each array element of the reconstructed signal differs for about 0.01 from the ground truth, which means that the reconstruction is quite accurate. Unsurprisingly, for the reconstruction error, it does not matter whether the Reconstruction Module uses the spatial domain or the frequency domain. The efficiency, however, is strongly affected by that choice: The module using the frequency domain is more than twice as fast as the other module, showing the speed advantage of the frequency domain convolution. Since our method of reconstructing one $\mathbf{TMSD}^{1000}$ sample is equivalent to reconstructing two $\mathbf{MSD}^{1000}$ samples individually, it is also not surprising that both Reconstruction Modules need twice the time for the $\mathbf{TMSD}^{1000}$ dataset as compared to the $\mathbf{MSD}^{1000}$ dataset, and are double as inaccurate for the former as compared to the latter.

	Average Reconstruction error $\mathbf{MSD}^{1000}$	Average Time needed $\mathbf{MSD}^{1000}$	Average Reconstruction error $\mathbf{TMSD}^{1000}$	Average Time needed $\mathbf{TMSD}^{1000}$
RM	$3.041 \cdot 10^{-4}$	0.375ms	$5.970 \cdot 10^{-4}$	0.742ms
RM_fd	$3.070 \cdot 10^{-4}$	0.149ms	$6.031 \cdot 10^{-4}$	0.294ms

Table 5.2: Evaluation of the Reconstruction Module. **RM** denotes the reconstruction module performing the convolution in the spatial domain, while **RM_fd** denotes the reconstruction module performing the convolution in the frequency domain. The reconstruction errors show the average MSELoss the respective module produces on a sample of the respective dataset. The time measurements show the average time needed by the modules to perform a single forward pass on the respective dataset.

5.2 Localization Module

The **Localization task** is defined the following way:

Given the signal and the normalized pattern, we want to find where exactly in the signal the normalized pattern can be found.

This is probably the most interesting task, because it requires us to estimate the translational parameter itself. The task is closely related to image registration, where the goal is to find a transformation that aligns two given images.

5.2.1 Different Approaches

We found several solutions for constructing the **Localization Module**. The input to this module is the signal S and the normalized pattern P, and the output is a number ρ that is an estimate of the ground truth translational value τ.

Division in the Frequency Domain

The first idea we had was to again make use of the assumption that the signal $S = \{s_i\}_{i=0,\dots,N-1}$ is obtained by convolving the normalized pattern $P = \{p_i\}_{i=0,\dots,N-1}$ with the localization $L = \{l_i\}_{i=0,\dots,N-1}$:

$$S = P * L,$$

or, equivalently, according to the convolution theorem (Section 3.3.5):

$$\mathrm{FFT}(S) = \mathrm{FFT}(P) \circ \mathrm{FFT}(L).$$

This means that, given the signal S and the normalized pattern P, the localization L can be recovered by dividing S by P in the frequency domain:

$$L^{predicted} = \text{IFFT}\left(\frac{\text{FFT}(S)}{\text{FFT}(P) + \varepsilon}\right), \tag{5.1}$$

where $\varepsilon > 0$ is some small constant that prevents divisions by zero. However, we want our localization module to output a single number ρ, which can be directly compared to the ground truth translational value τ. So we need to read ρ off the predicted L. To do this, we construct an $xvals$ vector that contains the time values that were used to discretize the signal: $xvals_i = i \cdot N^{-1}$ for $i = 0, \ldots, N-1$. We can then set $\rho = xvals_{i^*}$ with $i^* = \text{argmax}(L^{predicted})$.
This ρ is the output of our first localization module, which we will call $\mathbf{LM}^{\text{div-fd}}$.

Phase Correlation

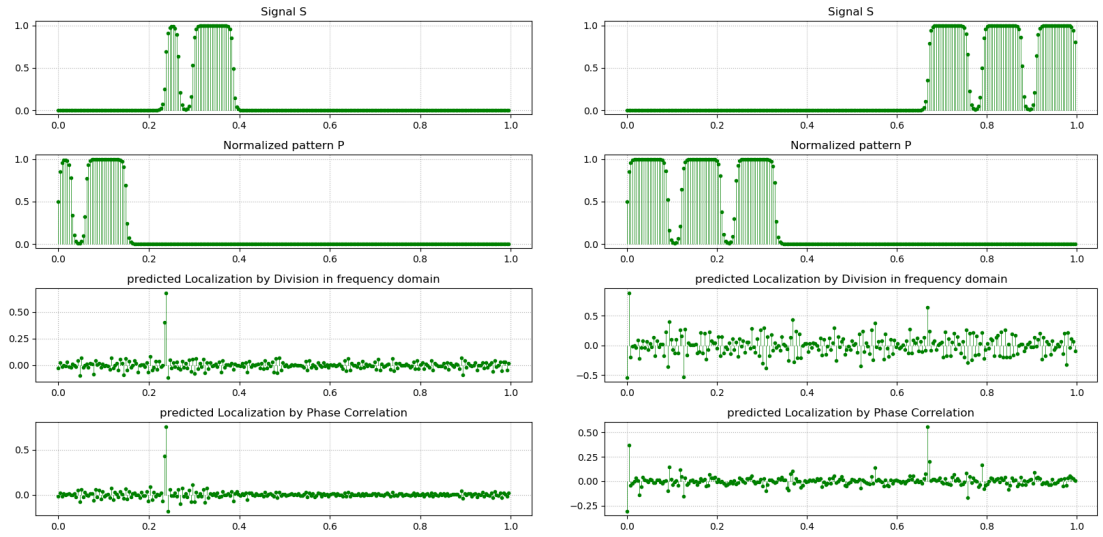
As it turns out, that first idea is very similar to an other, very well-known approach for image registration: Phase Correlation [21]. The phase correlation method computes the so-called *cross-power spectrum* between the signal S and the normalized pattern P in order to recover the frequency domain representation of L:

$$\text{FFT}(L^{predicted}) = \frac{\text{FFT}(S) \circ \overline{\text{FFT}(P)}}{|\text{FFT}(S) \circ \overline{\text{FFT}(P)}|}, \tag{5.2}$$

where $\overline{\text{FFT}(P)}$ is the complex conjugate of $\text{FFT}(P)$ and $|\bullet|$ denotes the amplitude spectrum of its argument.
It can now be shown that, under certain circumstances, Equation (5.1) and Equation (5.2) are the same (note that the =-sign denotes componentwise equality, as all objects in the following equations are arrays and all operations are componentwise):

$$\begin{aligned}
\frac{\text{FFT}(S) \circ \overline{\text{FFT}(P)}}{|\text{FFT}(S) \circ \overline{\text{FFT}(P)}|} &= \frac{\text{FFT}(S) \circ \overline{\text{FFT}(P)} \circ \text{FFT}(P)}{\text{FFT}(P) \circ |\text{FFT}(S)| \circ |\overline{\text{FFT}(P)}|} && (5.3)\\
&= \frac{\text{FFT}(S)}{\text{FFT}(P)} \circ \frac{|\text{FFT}(P)|^2}{|\text{FFT}(S)| \circ |\text{FFT}(P)|} && (5.4)\\
&= \frac{\text{FFT}(S)}{\text{FFT}(P)} \circ \frac{|\text{FFT}(P)|}{|\text{FFT}(S)|}\\
&\overset{?}{=} \frac{\text{FFT}(S)}{\text{FFT}(P)}. && (5.5)
\end{aligned}$$

In order to obtain Equation (5.3), we simply multiplied FFT(P) on both numerator and denominator, and we make use of the fact that the amplitude spectrum of the product of two arrays is equal to the product of the two individual amplitude spectra, due to elementary properties of complex numbers. Equation (5.4) also follows from elementary properties of complex numbers. Equation (5.5) is the critial statement; in only holds if the amplitude spectra are equal, i.e. if $|\text{FFT}(S)| = |\text{FFT}(P)|$. As we have seen in Section 3.3.4, this condition is met for very simple signals or if the translation between S and P has a value that is an integer multiple of the discretization interval N^{-1}, but in general, we have to assume that aliasing effects have distorted the amplitude spectrum s.t. $|\text{FFT}(S)| \neq |\text{FFT}(P)|$. Figure 5.3 shows an example what can happen if the amplitude spectra are not equal.



(a) (b)

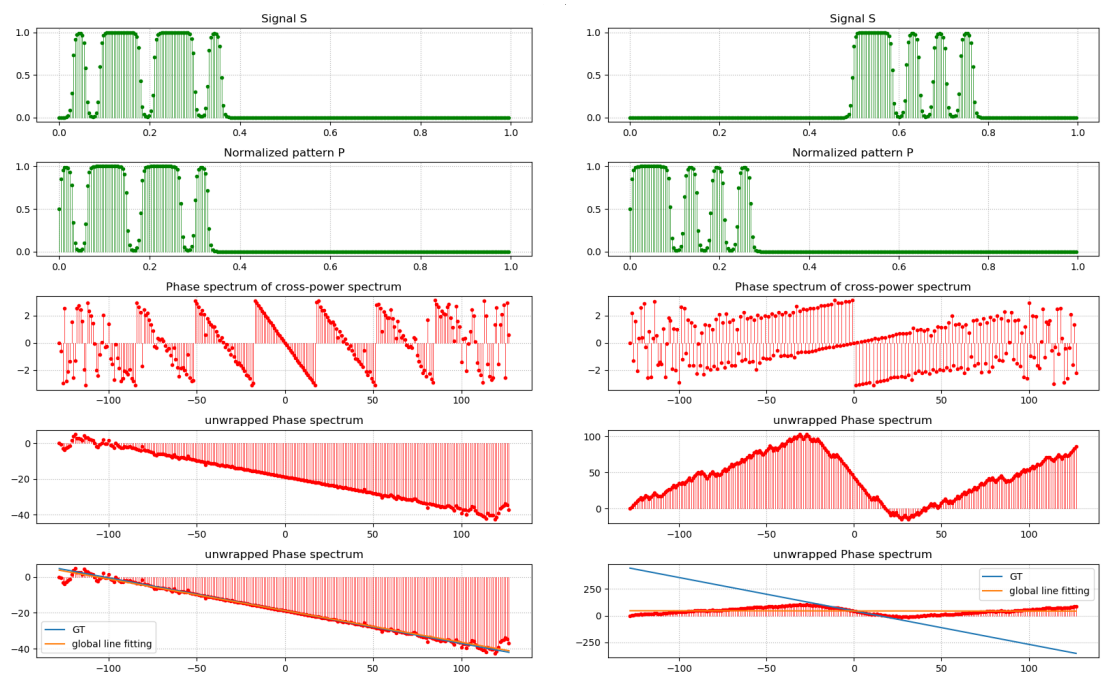
Figure 5.3: Comparison between Phase Correlation and Division in frequency domain. In (a), both methods compute very similar $L^{predicted}$ and thus output the same ρ. In (b), the difference between the amplitude spectra of S and P causes the two methods to compute very dissimilar $L^{predicted}$, the $\mathbf{LM}^{\text{div-fd}}$ module even outputs a completely wrong value for ρ.

Finally, we define the second localization module, $\mathbf{LM}^{\text{pc}}$, which calculates $L^{predicted}$ according to the phase correlation formula Equation (5.2) and then outputs $\rho = xvals_{i^*}$ with $i^* = \text{argmax}(L^{predicted})$.

Global Line Fitting

Both approaches so far have a major drawback: They do not have sub-pixel resolution, they still output an integer multiple of the discretization interval N^{-1}.

We tried to achieve sub-pixel accuracy by using $L^{predicted}$ as weights for the $xvals$ vector, e.g. $\rho = \sum_{i=0}^{N-1} l_i^{predicted} \cdot i \cdot N^{-1}$, but this had very little success because $L^{predicted}$ is too noisy (which can also be seen in Figure 5.3). They also both apply an IFFT operation and leave the frequency domain in order to compute the final value for ρ, which might not be in the spirit of a true frequency domain method.



(a) (b)

Figure 5.4: Global Line Fitting example. Calculating the cross-power spectrum between S and P and taking the phase spectrum from it yields the truncated phase spectrum in the third row. Unwrapping the phase spectrum using the method in [11] gives the unwrapped phase spectrum in the fourth row. Fitting a line through the unwrapped phase spectrum using the idea from [37] works well for example (a), but not for example (b), because the unwrapped phase spectrum in (b) does not show a unique line structure.

So, investigating the phase part of the cross-power spectrum between S and P, one can often find line structures, such as the ones in Figure 5.4, third row. These line structures give the idea that it is possible to fit a hyperplane (i.e. a line) through the phase spectrum in order to recover the localization parameter. Note that for the rest of this section, we will consider the frequency domain representation that uses the negative frequencies as described in Section 3.3.1.
A problem is that the phase spectrum is truncated s.t. all phase values are inside the interval $(-\pi, \pi]$, so fitting a line through that spectrum will almost certainly always lead to a line that is close to the constant zero-line. We first need to unwrap

the phases in order to be able to fit a line through the spectrum; we do this by using the very simple phase unwrapping method found in [11]; an example can be seen in Figure 5.4, fourth row.
From here, we can apply the line fitting idea, which is inspired by Stone et al. [37], who also recover a translational parameter from the phase spectrum of the cross-power spectrum: By definition of the cross-power spectrum between S and P (Equation (5.2)), the phase part of the cross-power spectrum is equal to the difference between the phase spectrum of S and the phase spectrum of P, i.e.

$$\text{phase}_f\left(\frac{\text{FFT}(S) \circ \overline{\text{FFT}(P)}}{|\text{FFT}(S) \circ \overline{\text{FFT}(P)}|}\right) = \text{phase}_f(\text{FFT}(S)) - \text{phase}_f(\text{FFT}(P)) \tag{5.6}$$

for each frequency $f = -N/2, \ldots, N/2 - 1$. This, again, follows from elemental properties of complex numbers. Assuming that both S and P are very simple signals that are only combinations of sinusoids with frequencies smaller than $N/2$, then we know from Section 3.3.4 that for each frequency f, $f = -N/2, \ldots, N/2-1$, the corresponding entries of the phase spectra satisfy the following equation:

$$\text{phase}_f(\text{FFT}(S)) = \text{phase}_f(\text{FFT}(P)) - 2\pi f\tau, \tag{5.7}$$

where τ is the ground truth translational value that we want to find out. For ease of notation, we will denote the cross-power spectrum between S and P as $\text{CPS}(S, P)$. Then, Equation (5.6) and Equation (5.7) can be combined to

$$\text{phase}_f(\text{CPS}(S, P)) = -2\pi f\tau. \tag{5.8}$$

In other words, the phases of the cross-power spectrum are a linear function of the frequency values, with slope equal to $-2\pi\tau$. Therefore, we will use a simple linear regression on the phase part of the cross-power spectrum to calculate the slope of the line that fits through the phase spectrum, and then divide that slope by -2π in order to get an approximation for τ. The final formula is the following:

$$\rho = -\frac{1}{2\pi}\frac{\sum_{f=-N/2}^{N/2-1}(\text{phase}_f - \overline{\text{phase}})(f - \overline{f})}{\sum_{f=-N/2}^{N/2-1}(f - \overline{f})^2}, \tag{5.9}$$

where we use the abbreviations

$$\begin{aligned} \text{phase}_f &:= \text{phase}_f(\text{CPS}(S, P)), \\ \overline{\text{phase}} &:= \frac{1}{N} \sum_{f=-N/2}^{N/2-1} \text{phase}_f, \text{ and} \\ \overline{f} &:= \frac{1}{N} \sum_{f=-N/2}^{N/2-1} f. \end{aligned}$$

This ρ in Equation (5.9) will be the output of our global line fitting localization module $\mathbf{LM}^{\text{global-lf}}$.

Local Line Fitting

For samples such as the one in Figure 5.4b, the global line fitting module is not very accurate, because the unwrapped phase spectrum simply does not have a line structure. However, upon closer inspection, we can see that the ground truth line does fit through the middle part of the phase spectrum, i.e. if we only consider the frequencies in a small interval centered around 0, then the line fitting could produce a more precise approximation of the localization. Additionally, there are also other regions of the phase spectrum that seem to have a line structure that is parallel to the one in the middle (see also Figure 5.5).

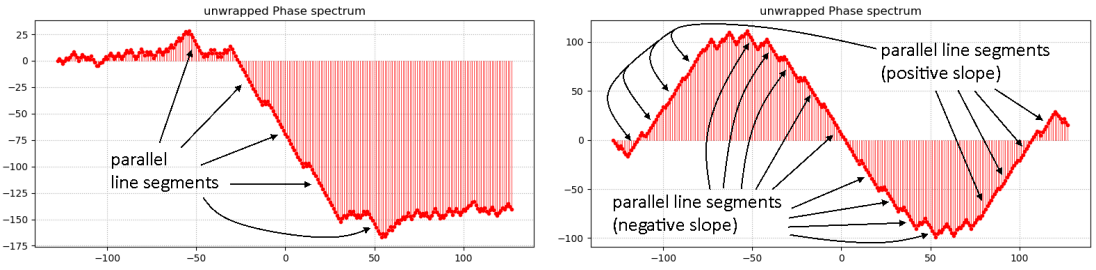


Figure 5.5: Parallel line segments in the phase spectrum. Note that the right example seems to have two dominant gradients, one positive and one negative, and there are many line segments that belong to the negative gradient as well as other line segments that belong to the positive gradient.

So, instead of calculating the slope of a line that fits through the entire phase spectrum, we calculate the slopes of lines that fit through small parts of the spectrum. To be more precise, we calculate the local gradients of the phases between pairs

of neighbouring frequencies in the cross-power spectrum:

$$\text{grad}_f = \frac{\text{phase}_{f+1} - \text{phase}_f}{f+1-f}. \tag{5.10}$$

Dividing the gradient by $-(2\pi)$, like in Equation (5.9), converts the slope value to an estimate of the translation τ:

$$\rho_f = -\frac{1}{2\pi}\text{grad}_f. \tag{5.11}$$

In order to decide which ρ_f are important and which ρ_f are not, we make use of the amplitude spectra of S and P. For each estimate ρ_f, we calculate the weight

$$w_f = \text{amp}_{f+1} + \text{amp}_f, \tag{5.12}$$

where

$$\text{amp}_f = \frac{\text{amplitude}_f(\text{FFT}(S)) \cdot \text{amplitude}_f(\text{FFT}(P))}{\left(\sum_{k=-N/2}^{N/2+1} \text{amplitude}_k(\text{FFT}(S))\right) \cdot \left(\sum_{k=-N/2}^{N/2+1} \text{amplitude}_k(\text{FFT}(P))\right)}.$$

As we have seen in Section 3.3.2, the amplitude spectrum tells us, how strongly the frequency f contributes to the input signal. So, we normalize the amplitude spectra of S and P s.t. each of them sums up to 1, and then amp_f is simply the product of the normalized amplitude spectra of S and P at frequency f. So, w_f should tell us how important the gradient grad_f (and thereby ρ_f) is, because it contains the information how important the frequencies f and $f+1$ are for S and P. Note that we cannot use the amplitude part of the cross-power spectrum, because due to its definition Equation (5.2), the amplitude part of the cross-power spectrum is always equal to one for all frequencies.
Finally, we normalize the weights w_f s.t. they sum up to 1, i.e.

$$\widehat{w}_f = \frac{w_f}{\sum_{k=-N/2}^{N/2-1} w_k} \tag{5.13}$$

and calculate

$$\rho = \sum_{f=-N/2}^{N/2-1} \widehat{w}_f \rho_f. \tag{5.14}$$

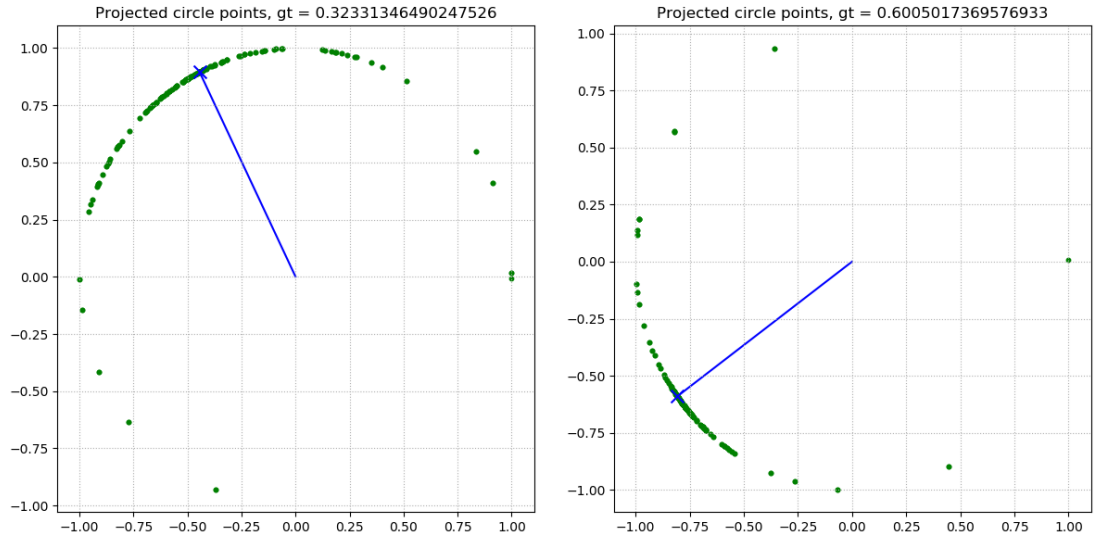


Figure 5.6: Two examples of gradients projected to the unit circle. We can see that the 2D-points $\widehat{\rho}_f$ are clustered around the projection of the ground truth τ, $(\cos(2\pi\tau), \sin(2\pi\tau))^T$, which is marked as blue cross.

However, this approach has a problem: due to the periodicity induced by the FFT (as we have seen in Section 3.3.3), the ground truth translation $\tau \in [0, 1]$ is equivalent to $-1 + \tau \in [-1, 0]$ in the frequency domain representation. This causes some ρ_f to approximate the positive translation τ, and some other ρ_f to approximate the negative translation $-1 + \tau$. This can also be seen in the right example of Figure 5.5, where there seem to be two gradients that dominate the spectrum, one positive and one negative. When some ρ_f pull the weighted sum in Equation (5.14) towards τ and some other ρ_f pull the sum towards $-1 + \tau$, then ρ often ends up somewhere in the middle. So, we use the same trick we used to define the circular loss Closs in Section 4.3.1: we project the estimates ρ_f to the unit circle, where it does not matter if ρ_f approximates the positive τ or the negative $-1 + \tau$, and define

$$\widehat{\rho}_f = \begin{pmatrix} \cos(2\pi\rho_f) \\ \sin(2\pi\rho_f) \end{pmatrix}. \tag{5.15}$$

Note that this means that it is not necessary anymore to unwrap the phase part of the cross-power spectrum beforehand, because the phase unwrapping described in [11] only adds integer multiples of 2π to each phase. So, due to the 2π-periodicity of sin and cos, Equation (5.15) would be the same with or without phase unwrapping. Figure 5.6 shows an example of ρ_f projected to the unit circle. We now compute

the weighted sum of these circle-points $\widehat{\rho}_f$:

$$\widehat{\rho} = \sum_{f=-N/2}^{N/2-1} \widehat{w}_f \widehat{\rho}_f, \tag{5.16}$$

extract its angle $\text{angle}(\widehat{\rho}) \in [0, 2\pi)$ (e.g. by calculating $\text{atan2}(\text{Im}(\widehat{\rho}), \text{Re}(\widehat{\rho})) \in (-\pi, \pi]$ and then mapping it to the interval $[0, 2\pi)$) and finally divide it by 2π in order to get a value ρ in the interval $[0, 1)$ that approximates the ground truth translational parameter τ:

$$\rho = \frac{\text{angle}(\widehat{\rho})}{2\pi}, \tag{5.17}$$

which will be the output of our local line fitting localization module $\mathbf{LM}^{\text{local-lf}}$.

Local Line Fitting CNN

The question that now arises is how to further improve the line fitting procedure. Recall that both line fitting modules so far assume that the signal S and the normalized pattern P satisfy the relation $\text{phase}_f(\text{FFT}(S)) = \text{phase}_f(\text{FFT}(P)) - 2\pi f\tau$ (Equation (5.7)), which might be too strong of an assumption for more complex signals. We try to alleviate the effects which could arise if that condition is not satisfied by inserting a convolutional neural network into the procedure. Specifically, this new module $\mathbf{LM}^{\text{local-lf-CNN}}$ will do exactly the same as the previous $\mathbf{LM}^{\text{local-lf}}$ module, with one exception: the weights w_f are not computed manually, as in Equation (5.12), rather they are calculated by a CNN that gets as input the amplitude spectra of both S and P. That way, the new module will learn which frequencies are the most important, based on features in the amplitude spectra. In our CNN architecture (see Figure 5.7), each convolution has kernel width 5, zero padding of length 2 on both sides to ensure that all feature maps have the same length as the input array, and is followed by a non-linear activation function

$$\text{ReLU}(x) := \max\{0, x\},$$

except the last convolutional layer, which is followed by a Softmax layer

$$\text{Softmax}(x_i) := \frac{\exp(x_i)}{\sum_j \exp(x_j)},$$

s.t. the output weights are all in the interval $[0, 1]$ and sum up to 1.

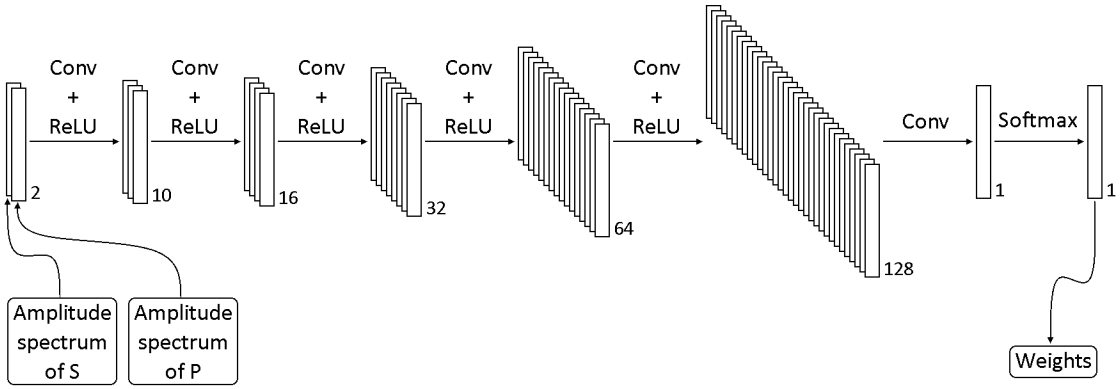


Figure 5.7: Architecture of the CNN that is used by the $\mathbf{LM}^{\text{local-lf-CNN}}$ module. Its input are the phase spectra of the signal S and the normalized pattern P, and the output is a weight vector. All 1D-convolutions have kernel length 5. The small number next to the feature maps denotes the number of feature map channels.

Spatial CNN

Finally, we take a look at a spatial domian method, because we want to investigate what the advantage of using the frequency domain over the spatial domain is. To do this, we leave the signal S and the normalized pattern P in the spatial domain and calculate the cross-correlation between the two. The cross-correlation can be characterised as "convolution without flipping the kernel", i.e.

$$(S \star P)_n := \sum_{m=0}^{N-1} S_{n+m} \overline{P}_m.$$

Notice that the complex conjugation of P has no effect in our case, as P only has real values. Figure 5.8 shows, how the result of a typical cross-correlation looks like.

For each location, the cross-correlation shows how well the normalized pattern matches the structure in the signal at that location. So, to find an approximation of the ground truth translation τ, we need to find out the global maximum of $S \star P$. Again, we cannot simply take the argmax of the cross-correlation, because we want to achieve sub-pixel accuracy. So, we send $S \star P$ through a CNN which calculates weights w_i, that tell us which of the discrete time points are most relevant. Recall that the i-th element of S belongs to the time value $i \cdot N^{-1}$, i.e. the output will be

$$\rho = \sum_{i=0}^{N-1} w_i \cdot i \cdot N^{-1}. \tag{5.18}$$

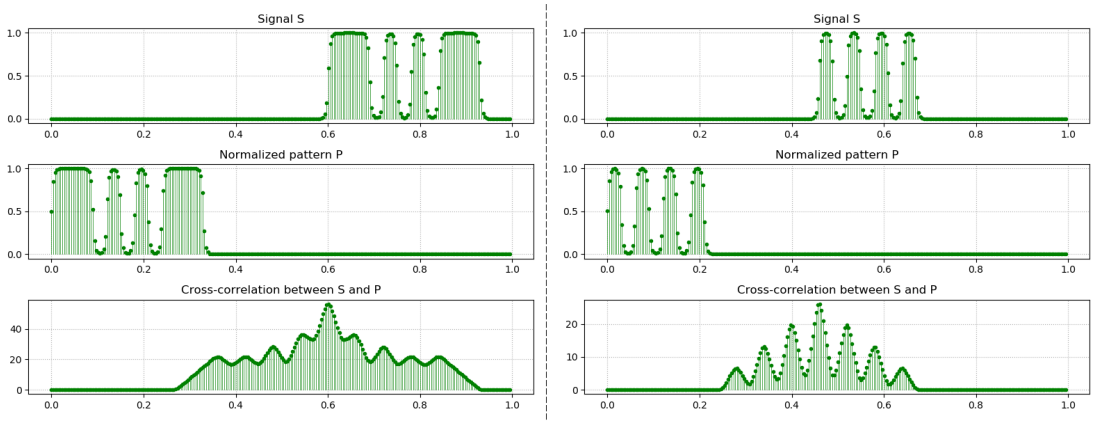


Figure 5.8: Examples for the cross-correlation between S and P.

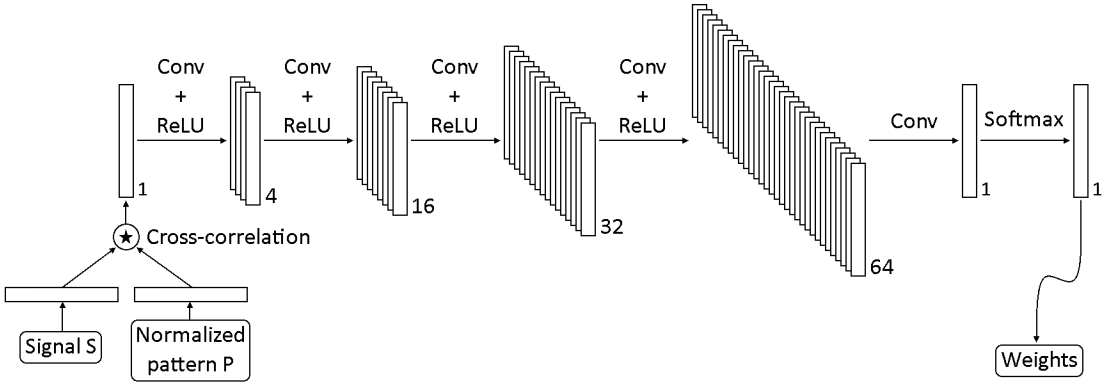


Figure 5.9: Architecture of the spatial CNN. First, the cross-correlation between the signal S and the normalized pattern P is computed. The result is passed through a CNN which eventually outputs a weight vector.

Figure 5.9 shows the architecture of that CNN. Again, each convolution has kernel length 5 and zero padding of size 2 on both sides, and all but the last convolutional layers are followed by a ReLU-layer, and the last convolutional layer is followed by a Softmax layer in order to get weights that are in the interval $[0, 1]$ and sum up to 1. The ρ in Equation (5.18) is the output of our spatial CNN localization module, $\mathbf{LM}^{\text{spatial-CNN}}$.

5.2.2 Evaluation

We will now evaluate how well our localization modules perform. First, a small summary of the different modules. All of them get as input a discretized signal S

and a discrete normalized pattern P, both of which have length N, and output a single number ρ that approximates the ground truth localization parameter τ.

- $\mathbf{LM}^{\mathrm{div-fd}}$: Divides FFT($S$) by FFT($P$), transfers the result back to the spatial domain, calculates the argmax and outputs $\rho = \mathrm{argmax} \cdot N^{-1}$.
- $\mathbf{LM}^{\mathrm{pc}}$: Calculates the cross-power spectrum between S and P, transfers it back to the spatial domain, calculates the argmax and outputs $\rho = \mathrm{argmax} \cdot N^{-1}$.
- $\mathbf{LM}^{\mathrm{global-lf}}$: Calculates the cross-power spectrum between S and P, unwraps its phase spectrum, fits a line through these unwrapped phase, takes the slope of that line, and converts in into an estimate ρ of τ.
- $\mathbf{LM}^{\mathrm{local-lf}}$: Calculates the cross-power spectrum between S and P, takes the local gradients in its phase spectrum without unwrapping it, projects these gradients to the unit circle, and computes the weighted sum of these circle points, where the weights are calculated from the amplitude spectra of S and P, to get an estimate ρ of τ.
- $\mathbf{LM}^{\mathrm{local-lf-CNN}}$: Same as $\mathbf{LM}^{\mathrm{local-lf}}$, except that the weights are calculated by a CNN that gets as input the amplitude spectra of S and P.
- $\mathbf{LM}^{\mathrm{spatial-CNN}}$: Calculates the cross-correlation between S and P, and sends it into a CNN which outputs weights w_i. The module then outputs $\rho = \sum_i w_i \cdot i \cdot N^{-1}$.

Notice that the last two modules contain CNNs, which means that they require to be trained. We trained both of these using the Adam optimizer [19] with learning rate $1 \cdot 10^{-4}$, and all other parameters were the default parameters from PyTorch (they are equal to the "good default settings" suggested in Algorithm 1 of [19]). The loss function we used for training is the Closs, as defined in Section 4.3.1. We trained separately on two datasets, once on the dataset $\mathbf{MSD}^{\infty}$ and once on the dataset $\mathbf{TMSD}^{\infty}$ (as defined in Section 4.3.2); the subscript MSD or TMSD will denote whether the module has been trained on the former or the latter dataset, and a number behind it will denote the number of training epochs, e.g. $\mathbf{LM}^{\mathrm{local-lf-CNN}}_{\mathrm{MSD10}}$ is the module $\mathbf{LM}^{\mathrm{local-lf-CNN}}$ trained on the $\mathbf{MSD}^{\infty}$ dataset for 10 epochs, and $\mathbf{LM}^{\mathrm{spatial-CNN}}_{\mathrm{TMSD100}}$ is the $\mathbf{LM}^{\mathrm{spatial-CNN}}$ module trained on the $\mathbf{TMSD}^{\infty}$ dataset for 100 epochs.

For testing, we used both the $\mathbf{MSD}^{1000}$ and the $\mathbf{TMSD}^{1000}$ dataset, as defined in Section 4.3.2. We expect the first four modules, i.e. the modules that do

not train a CNN, to perform badly on the $\mathbf{TMSD}^{1000}$ dataset, because when constructing those modules, we did not consider large sources of noise such as the second Morse letter that is present in each sample of the $\mathbf{TMSD}^{1000}$ dataset. For this evaluation, the localization modules need only to localize one Morse letter in the $\mathbf{TMSD}^{1000}$ samples, not both (also, recall that the dataset contains a fixed vector LeftOrRight$\in \{0,1\}^{1000}$ that decides for each sample whether the left or the right letter shall be the letter of interest, so this test is not biased towards either the left or the right letter). For measuring the accuracy, we used the Closs as defined in Section 4.3.1.

$\lvert\rho-\tau\rvert$	$\mathrm{Closs}(\rho,\tau)$
0.5	2
$1\cdot 10^{-1}$	$\sim 2\cdot 10^{-1}$
$1\cdot 10^{-2}$	$\sim 2\cdot 10^{-3}$
$1\cdot 10^{-3}$	$\sim 2\cdot 10^{-5}$
$1\cdot 10^{-4}$	$\sim 2\cdot 10^{-7}$
$1\cdot 10^{-5}$	$\sim 2\cdot 10^{-9}$

Table 5.10: Conversion between $|\rho-\tau|$ and $\mathrm{Closs}(\rho,\tau)$. E.g., if $\mathrm{Closs}(\rho,\tau)$ is in the order of 10^{-5}, then the absolute distance between ρ and τ is approximately in the order of 10^{-3}. Due to periodicity reasons, $\mathrm{Closs}(\rho,\tau)$ cannot become larger than 2 for $\rho,\tau\in[0,1]$.

Table 5.11 shows the results of our experiments. It now becomes obvious why we tried out so many approaches: While most of the ideas work well for signals that only contain one Morse letter, only two modules (namely the two that use a CNN) have a satisfying accuracy for signals that contain two Morse letters.
For better clearness, Table 5.10 shows how to convert between $|\rho-\tau|$ and $\mathrm{Closs}(\rho,\tau)$. Our datasets have resolution 256, so the pixel width is equal to $N^{-1}=256^{-1}\approx 3.9\cdot 10^{-3}$. This means that sub-pixel accuracy is achieved if the Closs error is in the order of at most 10^{-6}.

Evaluation on $\mathbf{MSD}^{1000}$

The two modules $\mathbf{LM}^{\mathrm{div-fd}}$ and $\mathbf{LM}^{\mathrm{pc}}$ were not constructed to have sub-pixel accuracy, and this is confirmed by Table 5.11. The $\mathbf{LM}^{\mathrm{global-lf}}$ module has the worst accuracy, the reason being that the unwrapped phase part of the cross-power spectrum contains line segments of both positive and negative slopes, causing the global line fitting module to output values around zero most of the time.

	Average Localization Error		Average timing per forward pass
	$\mathbf{MSD}^{1000}$	$\mathbf{TMSD}^{1000}$	
$\mathbf{LM}^{\text{div-fd}}$	$2.322 \cdot 10^{-3}$	$3.255 \cdot 10^{-1}$	0.202ms
$\mathbf{LM}^{\text{pc}}$	$3.254 \cdot 10^{-5}$	$1.166 \cdot 10^{-1}$	0.238ms
$\mathbf{LM}^{\text{global-lf}}$	$1.653 \cdot 10^{-1}$	$9.244 \cdot 10^{-1}$	4.741ms
$\mathbf{LM}^{\text{local-lf}}$	$1.350 \cdot 10^{-6}$	$5.552 \cdot 10^{-1}$	0.473ms
$\mathbf{LM}^{\text{local-lf-CNN}}_{\text{MSD40}}$	$\mathbf{3.064 \cdot 10^{-9}}$	$2.357 \cdot 10^{-1}$	1.454ms
$\mathbf{LM}^{\text{local-lf-CNN}}_{\text{TMSD300}}$	$2.766 \cdot 10^{-6}$	$6.325 \cdot 10^{-4}$	1.432ms
$\mathbf{LM}^{\text{spatial-CNN}}_{\text{MSD40}}$	$5.386 \cdot 10^{-7}$	$1.534 \cdot 10^{-1}$	0.664ms
$\mathbf{LM}^{\text{spatial-CNN}}_{\text{TMSD300}}$	$1.125 \cdot 10^{-6}$	$\mathbf{6.312 \cdot 10^{-5}}$	0.657ms

Table 5.11: Evaluation of the Localization Modules. The localization errors are the average Closs the respective module produces on a sample of the respective dataset. The time measurements show the average time the respective module needs to perform a single forward pass, which is the same on both datasets, because both have the same discretization resolution $N = 256$. Marked in bold are the highest accuracies for the two datasets.

Separating the line segments and looking at each local gradient individually, like the $\mathbf{LM}^{\text{local-lf}}$ module does, works very well, as it does reach sub-pixel accuracy. Even better, when removing the manual weight computation and instead inserting a CNN that computes the weights for the local gradients, like the $\mathbf{LM}^{\text{local-lf-CNN}}$ module does, then the best accuracy of all the modules is achieved, if trained on the $\mathbf{MSD}^{\infty}$ dataset for 40 epochs. The $\mathbf{LM}^{\text{spatial-CNN}}$ performed the second best when trained on the $\mathbf{MSD}^{\infty}$ dataset for 40 epochs. If the two CNN modules have instead been trained in the $\mathbf{TMSD}^{\infty}$ dataset for 300 epochs each, then their accuracy is still in the sub-pixel range.

Evaluation on $\mathbf{TMSD}^{1000}$

Only the two modules that contain a CNN were able to produce satisfying accuracy, albeit not in the sub-pixel accuracy range. In addition, this required hundreds of epochs of training on the $\mathbf{TMSD}^{\infty}$ dataset, and training on the $\mathbf{MSD}^{\infty}$ dataset was useless for this challenge. The $\mathbf{LM}^{\text{spatial-CNN}}$ module has slightly higher ac-

curacy than the $\mathbf{LM}^{\text{local-lf-CNN}}$ module, so it seems like using the spatial domain is the best way to treat the localization task for signals that contain two Morse letters. All the other modules failed completely, because the second Morse letter in the signal introduced noise effects that were simply too devastating for those modules.

Speed Analysis

Unsurprisingly, the two modules $\mathbf{LM}^{\text{div-fd}}$ and $\mathbf{LM}^{\text{pc}}$ were the fastest, but the price for this efficiency is the lack of sub-pixel accuracy. The $\mathbf{LM}^{\text{global-lf}}$ module was extremely slow, the reason for this is our inefficient implementation of the phase unwrapping. However, since that module is the most inaccurate of them all anyways, we did not bother to try and make it more efficient. The $\mathbf{LM}^{\text{local-lf}}$ module was not as fast as the first two modules, but it achieves sub-pixel accuracy for the $\mathbf{MSD}^{1000}$ dataset without having to train a CNN, so this module has a good trade-off between accuracy and speed for the $\mathbf{MSD}^{1000}$ dataset. The two CNN modules are slower than all the other modules (except the global line fitting module), but they achieve the best accuracies on both datasets. However, at this point, we can do a more in-depth analysis of the efficiency of the two CNN modules. For this, we constructed several Morse signal datasets, with resolutions ranging from $N = 2^8 = 256$ up to $N = 2^{13} = 8192$. We want to find out if at some point, the spatial CNN becomes considerably slower than the local line fitting CNN, which would justify the use of the frequency domain instead of the spatial domain. Table 5.12 shows the result of this comparison.

	256	512	1024	2048	4096	8192
$\mathbf{LM}^{\text{local-lf-CNN}}$	1.443	2.368	3.783	6.919	14.252	36.723
$\mathbf{LM}^{\text{spatial-CNN}}$	0.661	1.128	2.047	4.946	34.432	123.734

Table 5.12: Speed comparison. For each resolution value $N \in \{256, 512, 1024, 2048, 4096, 8192\}$, we constructed 1000 Morse signal dataset samples and let the two modules do a forward pass on all 1000 samples. The time values in the table show the average time in ms that the respective module needs for performing a single forward pass on a sample of the respective resolution.

Table 5.12 reveals that if the resolution of the discretized signal is at least $2^{12} = 4069$, then the spatial CNN is far slower than the local line fitting CNN. The reason

is the cross-correlation operation; it is very costly in the spatial domain, especially if the input arrays S and P are very long. So, for datasets with resolutions up to 2^{11}, the $\mathbf{LM}^{\text{spatial-CNN}}$ is the preferred alternative, and for resolutions larger than that, the $\mathbf{LM}^{\text{local-lf-CNN}}$ module can be preferred.

5.3 Normalization Module

The **Normalization task** is defined the following way:

Given the signal and the localization, we want to recover the normalized pattern for the structure present in the signal at the given location.

This task is related to a denoising task, where the noise is a lateral shift, and the normalized pattern is how the signal would look like if there were no translation along the timeline, and no other pattern in the signal.

5.3.1 Different Approaches

The **Normalization Module** gets as input the signal S and the localization L and needs to output an estimate Q which should approximate the ground truth normalized pattern P.

Division in the Frequency Domain

As always, the first approach is to make use of the assumption that the signal $S = \{s_i\}_{i=0,\dots,N-1}$, the normalized pattern $P = \{p_i\}_{i=0,\dots,N-1}$ and the localization $L = \{l_i\}_{i=0,\dots,N-1}$ satisfy the following condition:

$$S = P * L \quad \text{or, equivalently,} \quad \text{FFT}(S) = \text{FFT}(P) \circ \text{FFT}(L).$$

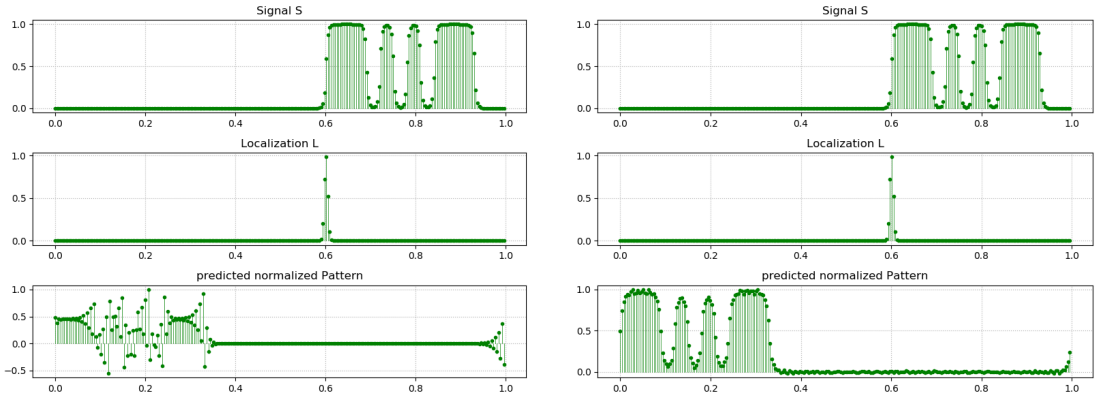
P can then be recovered the following way:

$$P = \text{IFFT}\left(\frac{\text{FFT}(S)}{\text{FFT}(L) + \varepsilon}\right),$$

where $\varepsilon > 0$ is a small constant that prevents divisions by zero.
However, as can be seen in the Figure 5.13a, this produces very bad results; the end result seems to be distorted by aliasing and high-frequency noise.

So, we try to get rid of the aliasing noise by applying a filter. Specifically, we define

$$Q^{\text{fd}} = \{q_k^{\text{fd}}\}_{k=-N/2,\dots,N/2-1} := \frac{\text{FFT}(S)}{\text{FFT}(L) + \varepsilon}$$



(a) Example without filtering.

(b) Example with filtering.

Figure 5.13: Normalization by division in frequency domain. (a) shows an example without filtering, (b) shows an example with filtering.

and calculate the amplitude spectrum

$$\text{amps}(S) := |\text{FFT}(S)|$$

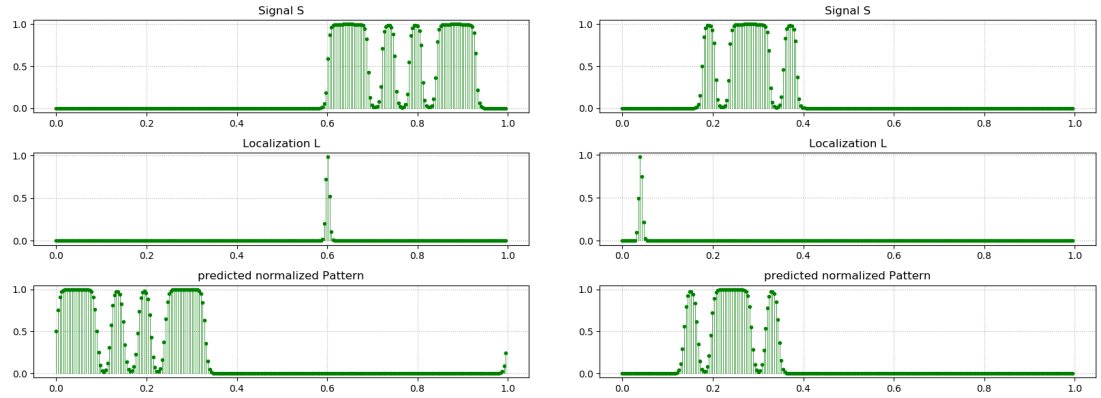
of S. We then set the k-th component of Q^{fd} to 0 if the k-th component of $\text{amps}(S)$ is smaller than $0.01 \cdot \max(\text{amps}(S))$. In other words, if the frequency k is not very important for S, then we also discard this frequency from Q^{fd}. We then calculate $\widehat{Q} = \{\widehat{q}_n\}_{k=0,\dots,N} := \text{IFFT}(Q^{\text{fd}})$ and apply a smoothing filter to it in order to obtain $Q = \{q_n\}_{k=0,\dots,N}$:

$$q_n = \frac{1}{4}\left(\sum_{j=0}^{n} \frac{\widehat{q}_j}{2^{n-j}} + \sum_{j=0}^{N-1-n} \frac{\widehat{q}_{n+j}}{2^j}\right), \tag{5.19}$$

which filters out high frequency noise. This filter is a discrete version of the exponential smoothing [25] that we already used in Section 4.2.1. Finally, we normalize Q s.t. $\max(Q) = 1$. Figure 5.13b shows an example output of this procedure. We will call the normalization module which outputs this Q the $\mathbf{NM}^{\text{div}-\text{fd}}$ module.

Convolution with the Reverse Localization

The second approach is to use a convolution, i.e. we want to recover P by somehow convolving S with L. We already know that the convolution $S * L$ will delay the signal S, i.e. S is shifted to the right. However, we want to shift S to the left. Recall that the interesting pattern (e.g. Morse letter) starts at the ground truth time point τ. If we can shift that pattern to the left, s.t. it starts at time point 0 instead, then this is an approximation for the normalized pattern P. In order to



(a) Simple example. (b) Example with localization noise.

Figure 5.14: Normalization by convolution with flipped localization. It works well for (a), but for (b), the localization is not precise, so the normalization result is unsatisfying.

achieve this, we make use of the periodicity of the frequency domain representation: If we convolve S with the reverse localization in the frequency domain (i.e. convolve S with the flipped localization that has its peak at $1 - \tau$), then the pattern which started at τ, will, after the convolution, start at the localization $\tau + (1 - \tau) = 1$. Thanks to the periodicity of the frequency domain representation, this is equivalent to the pattern starting at the time point 0.
Convolving S with the flipped L in the frequency domain can be realised by the following formula:

$$\mathrm{FFT}(S * L^{\mathrm{flipped}}) = \mathrm{FFT}(S) \circ \overline{\mathrm{FFT}(L)},$$

i.e. all we need to do is use the complex conjugate of $\mathrm{FFT}(L)$ in order to flip L for the convolution. So, the second normalization module, $\mathbf{NM}^{\mathrm{conv}}$, outputs

$$Q = \frac{\mathrm{IFFT}(\mathrm{FFT}(S) \circ \overline{\mathrm{FFT}(L)})}{\max(\mathrm{IFFT}(\mathrm{FFT}(S) \circ \overline{\mathrm{FFT}(L)}))},$$

where the division by the maximum ensures that $\max(Q) = 1$. Figure 5.14 shows how an example output from this module looks like.

Classifier and Storage

Both normalization modules so far assume that the localization L is precise and that the signal S contains only the one pattern P and nothing else. However, we want to find a solution for the normalization task that is more robust: it should

be able to input a signal that contains more than just one interesting structure, and it should be possible to input somewhat misplaced localizations. Figure 5.14b shows an example where the output $\mathbf{NM}^{\text{conv}}$ is not very precise due to localization noise.
From our experience with the localization modules, it seems like we need to make use of CNNs again if we want our normalization module to be able to deal with noisy and more complex signals.
Our idea is to build a CNN classifier, that extracts the semantic information what kind of pattern it encounters in the signal. Then, this class, say c, is sent to a learnable storage, and the storage simply outputs its c-th entry, which we define as the output Q of the classifier-storage normalization module $\mathbf{NM}^{\text{class}-\text{storage}}$. This concept is related to Memory Networks [43], however in our implementation, the storage is exclusively updated through backpropagation, and is not directly updated by the input. Figure 5.15 shows our architecture for this module.

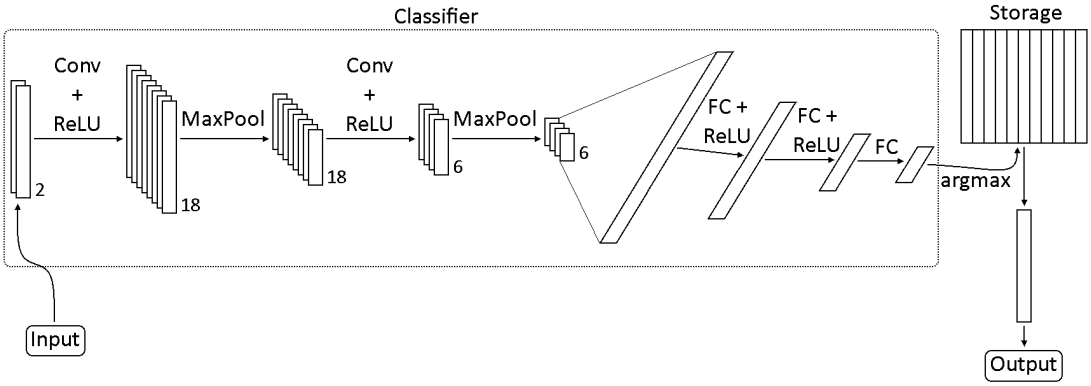


Figure 5.15: Architecture of the classifier-storage normalization module.

The classifier is inspired by the original LeNet-5 architecture [22]. It contains, besides convolutional layers and ReLU activation layers, also MaxPool layers (which subsample the feature map by taking the maximum value from small patches) and fully connected (FC) layers (which apply a learnable, linear transformations). The output of the classifier is a vector whose length is equal to the number of possible classes (26, in the case of our Morse signal datasets), and whose c-th entry is the confidence that the pattern in the input belongs to class c. Our $\mathbf{NM}^{\text{class}-\text{storage}}$ module simply takes the argmax c^* of that vector, and outputs the c^*-th entry of the learnable storage. This means that in the storage, the module maintains estimates of all possible normalized patterns. The question that remains is what the input of the classifier should be. For this, we have two possibilities: We convolve,

as before, the signal with the reverse localization in the frequency domain

$$\mathrm{FFT}(S * L^{\mathrm{flipped}}) = \mathrm{FFT}(S) \circ \overline{\mathrm{FFT}(L)}$$

in order to have the pattern, that belongs to L, start roughly at the beginning of the array, and hand this is as input to the classifier, thereby defining the $\mathbf{NM}^{\mathrm{class-storage-fd}}$ module, or we first apply an inverse FFT on that array and hand that spatial domain representation over to the classifier, thereby defining the $\mathbf{NM}^{\mathrm{class-storage-sd}}$ module. Comparing these two modules directly will tell us whether there is any advantage in passing the frequency domain reprensentation to the classifier instead of the spatial domain representation. Notice that in both cases, the normalized patterns which are maintained and output by the storage are in the spatial domain representation.

Encoder-Decoder

Another approach is to use an Encoder-Decoder CNN, a normalization module which we will call $\mathbf{NM}^{\mathrm{encdec}}$. The encoder produces a vector that contains the most characteristic information about the input, and the decoder unwraps that information and produces an output that is a somehow canonical representation of the input. This idea is similar to that of Denoising Autoencoders [47, 12]. In our implementation, the encoder part will be the same as the classifier part of the $\mathbf{NM}^{\mathrm{class-storage}}$ module. This means that the vector produced by the encoder can maybe contain some class information about the input, but it is also possible that the encoder learns an entire different information representation. In any ways, the decoder will then have to unpack that information and produce an estimate of the normalized pattern. The advantage is that this module has more flexibility than the classifier-storage combination. The encoder part does not necessarily have to learn how to classify the input, rather it learns some abstract representation of the input which it thinks is most useful.
Figure 5.16 shows the architecture of our $\mathbf{NM}^{\mathrm{encdec}}$ module. The encoder part is the same as the classifier part from the $\mathbf{NM}^{\mathrm{class-storage}}$ module, and the decoder part is the almost symmetric counterpart, i.e. it reverses the operations of the encoder in some sense. The "reverse" of the MaxPooling layers in this case are the Upsample layers, which employ a linear interpolation method for upsampling.

As was the case with the $\mathbf{NM}^{\mathrm{class-storage}}$ module, the $\mathbf{NM}^{\mathrm{encdec}}$ module also supports two modes of operation: If its input is the convolution of the signal S with the flipped localization in the frequency domain, i.e.

$$\mathrm{Input} = \mathrm{FFT}(S) \circ \overline{\mathrm{FFT}(L)},$$

then the module will be called $\mathbf{NM}^{\text{encdec}-\text{fd}}$. If the convolved signal is first brought back into the spatial domain before being passed to the encoder, i.e. if

$$\text{Input} = \text{IFFT}(\text{FFT}(S) \circ \overline{\text{FFT}(L)}),$$

then the module will be called $\mathbf{NM}^{\text{encdec}-\text{sd}}$. In both cases, the output of the module will be an estimate of the normalized pattern in its spatial domain representation.

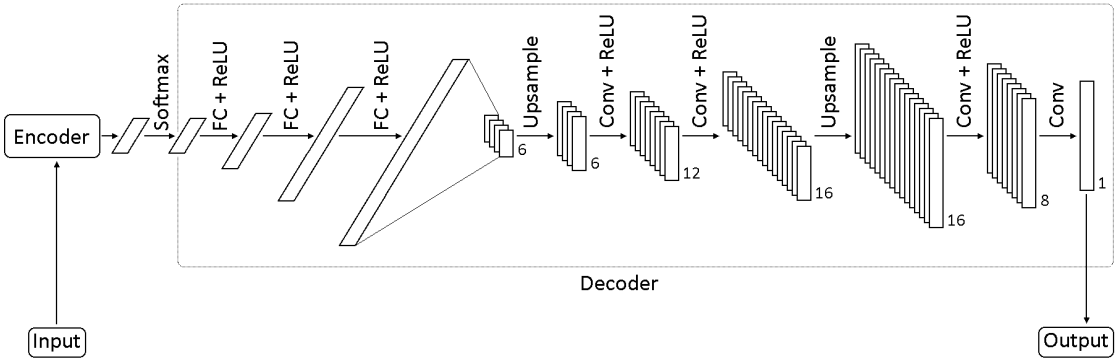


Figure 5.16: Architecture of the encoder-decoder normalization module. The Encoder part is equal to the Classifier part in Figure 5.15.

5.3.2 Training

The four modules $\mathbf{NM}^{\text{class}-\text{storage}-\text{fd}}$, $\mathbf{NM}^{\text{class}-\text{storage}-\text{sd}}$, $\mathbf{NM}^{\text{encdec}-\text{fd}}$, and $\mathbf{NM}^{\text{encdec}-\text{sd}}$ contain neural networks that need training. To do this, we used the Adam optimizer [19] with learning rate $1 \cdot 10^{-4}$, and all other parameters were the default parameters from PyTorch (i.e. the exponential decay rates for the moment estimates are $\beta_1 = 0.9$ and $\beta_2 = 0.999$, and the term to increase numerical stability is $\varepsilon = 10^{-8}$). We used a special dataset for training, namely the $\mathbf{TMSDL}^{\infty}$ dataset, as defined in Section 4.3.2. This dataset poses two big challenges for the normalization modules:

- Each sample signal S of this dataset has two Morse letters, which should make it more challenging for the normalization modules to focus on the one letter that is pointed at by the localization L. We will call this letter *l.o.i.*, which is short for *letter of interest*. We will call the other letter, the one which does not belong to the localization L, the *n.i.l.*, short for *not interesting letter*. Recall that this dataset randomly chooses whether the left or the right letter in the signal is the *l.o.i.*, which is why we have to introduce this terminology.

- In this dataset, the localization L is additionally noisy, meaning that it does not exactly point at the starting location of the *l.o.i.*, which forces the modules to develop robustness towards this kind of noise.

Finally, during training, we also added a third source of variation to this dataset:

- We generate a random window $V = \{v_i\}_{n=0,\dots,N-1}$ and apply it to S (i.e. perform a component-wise multiplication) in order to obtain S^{windowed}. These windows are constructed in a manner s.t. exactly one of the three following cases is true:

  Case 1: S^{windowed} contains only *l.o.i.* and nothing else.

  Case 2: S^{windowed} contains both *l.o.i* and *n.i.l.*

  Case 3: S^{windowed} contains *l.o.i.* and a part of *n.i.l.*

  Figure 5.17 shows some examples of how S^{windowed} can look like. Notice that the window always includes the *l.o.i.* For example, if the *l.o.i.* is the left letter, then the left border of the window is the startpoint of the *l.o.i.*, and the right border of the window is somewhere between the endpoint of the *l.o.i.* and the enpoint of the *n.i.l.*: we sample a normal distributed random number r with mean 0 and standard deviation

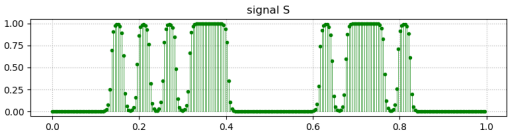
  $$\frac{1}{6}(\text{endpoint}(n.i.l) - \text{endpoint}(l.o.i.));$$

  if $r < 0$, then the right border of the window is set to $\text{endpoint}(n.i.l) + r$, and if $r \geq 0$, the right border of the window is set to $\text{endpoint}(l.o.i.) + r$. This means that Case 1 and Case 2 are approximately equally probable, and Case 3 happens, on average, slightly less often than the other two. In case of *l.o.i.* being the right letter in the signal, then we use analogous mechanisms: The right border of the window is equal to the endpoint of *l.o.i.* and the left border is randomly chosen to be between the startpoint of *n.i.l* and the startpoint *l.o.i.* using a normal distribution with mean 0 and standard deviation

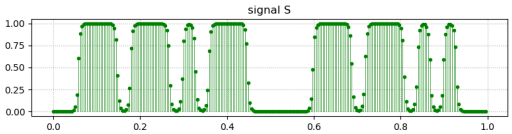
  $$\frac{1}{6}(\text{startpoint}(l.o.i.) - \text{startpoint}(n.i.l.)).$$

  The modules then use S^{windowed} instead of S as input signal.

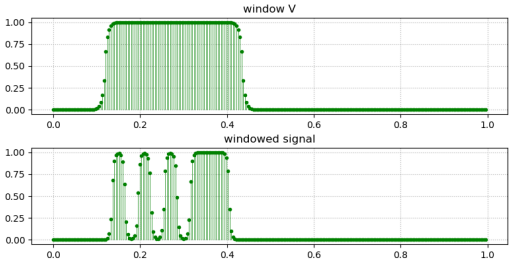
When a normalization module trains on this dataset, then it will learn how to deal with 1.) signals that contain only one Morse letter, 2.) signals that contain two Morse letters and 3.) signals that contain a Morse letter and a part of another Morse letter. It will also learn how to deal with localization noise. The advantage



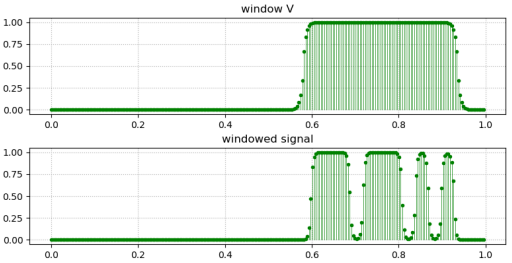
(a) Example 1. The *letter of interest* is the left one.



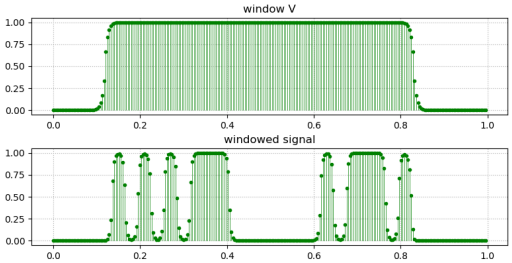
(b) Example 2. The *letter of interest* is the right one.



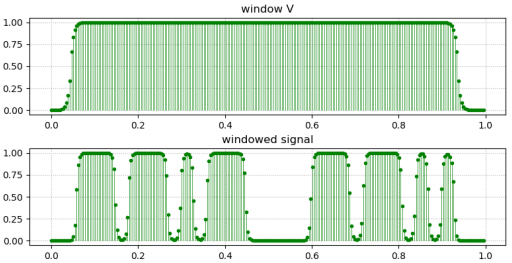
(c) Example 1 Case 1. The window only covers the *l.o.i.*



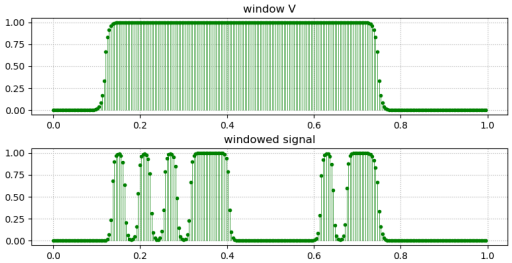
(d) Example 2 Case 1. The window only covers the *l.o.i.*



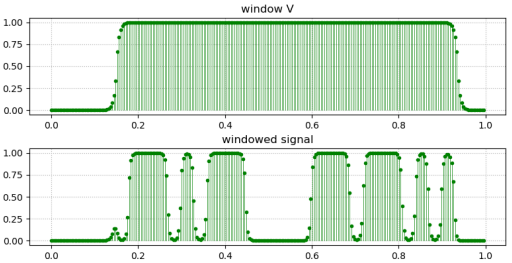
(e) Example 1 Case 2. The window covers both letters entirely.



(f) Example 2 Case 2. The window covers both letters entirely.



(g) Example 1 Case 3. The window fully covers the *l.o.i.* and only a part of the other letter.



(h) Example 2 Case 3. The window fully covers the *l.o.i.* and only a part of the other letter.

Figure 5.17: Examples for the windowing used when training the normalization modules. The window is constructed in a manner such that one of the 3 cases shown above is true. In particular, the *letter of interest* is never cut off by the windowing.

of this is that we do not need to train the normalization modules on multiple datasets.

As for the loss function, we used the MSEloss (as described in Section 4.3.1) for training the encoder-decoder modules.
For the classifier-storage modules, we used a combination of MSEloss and PyTorch's CrossEntropyLoss, which is defined the following way:

$$\text{CrossEntropyLoss}(X, c) := -\log\left(\frac{x_c}{\sum_j \exp(x_j)}\right),$$

where $X = \{x_j\}_{j=0,\dots,C-1}$ is a vector whose j-th entry is the confidence for class j, c is the ground truth class and C is the total number of classes (in our case, $C = 26$).
To be more precise, let Q, X be the output of a $\mathbf{NM}^{\text{class-storage}}$ module, where Q is the estimate of the normalized pattern and X is the intermediate vector produced by the classifier inside the module. Let P be the ground truth normalized pattern and c be the ground truth class of the normalized pattern (i.e. P represents the c-th letter of the alphabet in Morse code). Then, we compare if $\text{argmax}(X) = c$. If that is the case, then the loss that is propagated back through the module will be

$$\text{CrossEntropyLoss}(X, c) + \text{MSELoss}(\text{Q}, \text{P}).$$

If $\text{argmax}(X) \neq c$, then the loss that is propagated back through the module will be

$$\text{CrossEntropyLoss}(X, c).$$

This ensures that the learnable storage inside the $\mathbf{NM}^{\text{class-storage}}$ module is, during training, only updated if the classifier produced a correct prediction. Updating the storage upon a misclassification would not make much sense.

All four modules were trained for 400 epochs.

5.3.3 Evaluation

We will now evaluate the normalization modules. Each of them gets as input the signal S and the localization L, and is supposed to output an array Q which should be an approximation of the ground truth normalized pattern P. First, a small summary of the modules:

- $\mathbf{NM}^{\text{div-fd}}$: Divides FFT($S$) by FFT($L$), performs some filtering based on the amplitude spectrum of S, applies an IFFT on the result, and smoothes it to get the output Q.
- $\mathbf{NM}^{\text{conv}}$: Shifts S to the left by calculating FFT(S) $\circ$ $\overline{\text{FFT}(L)}$, then applies the IFFT to get Q.
- $\mathbf{NM}^{\text{class-storage-fd}}$: Calculates FFT($S$) $\circ$ $\overline{\text{FFT}(L)}$, and passes it on to a classifier CNN, whose output yields an index c. The output Q is the c-th entry of an internal, learnable storage.
- $\mathbf{NM}^{\text{class-storage-sd}}$: Same as $\mathbf{NM}^{\text{class-storage-fd}}$, except that it passes the IFFT of FFT(S) $\circ$ $\overline{\text{FFT}(L)}$ to the classifier.
- $\mathbf{NM}^{\text{encdec-fd}}$: Calculates FFT($S$) $\circ$ $\overline{\text{FFT}(L)}$, and passes it on to a Encoder-Decoder CNN, to get the output Q.
- $\mathbf{NM}^{\text{encdec-sd}}$: Same as $\mathbf{NM}^{\text{encdec-fd}}$, except that it passes the IFFT of FFT(S) $\circ$ $\overline{\text{FFT}(L)}$ to the Encoder-Decoder.

We will test them on four datasets: $\mathbf{MSD}^{1000}$, $\mathbf{MSDL}^{1000}$, $\mathbf{TMSD}^{1000}$, and $\mathbf{TMSDL}^{1000}$, all of which were defined in Section 4.3.2. We will measure the normalization error using the MSEloss. Table 5.18 shows how to convert between the absolute difference and the MSEloss, so it is easier to interpret the error values. For the classifier-storage modules, we will also state the ratio of misclassifications. Note that for the Two Morse signal dataset samples, only one of the two Morse letters will be normalized, namely the one specified by the dataset's internal fixed LeftOrRight $\in \{0,1\}^{1000}$ vector, whose goal it is to prevent the modules to have any bias towards the left or the right letter.

Table 5.19 shows the results of our test. We can see that the two simple modules, $\mathbf{NM}^{\text{div-fd}}$ and $\mathbf{NM}^{\text{conv}}$, only give satisfactory results on the $\mathbf{MSD}^{1000}$ dataset. The reason is obvious: They were not designed to deal with localization noise or a second Morse letter in the signal. However, even on the $\mathbf{MSD}^{1000}$ dataset, the accuracy of these two modules is far worse than the accuracy of the classifier-storage and the encoder-decoder approaches. The four CNN modules profit from their training, they give acceptable results on all four datasets (i.e. the largest normalization error among these modules is $5.374 \cdot 10^{-3}$, which means that, on average, the componentwise difference between output Q and ground truth P is in the order of 10^{-2}). To a certain degree, they acquired robustness towards localization noise and the presence of a second Morse letter in the signal. Figure 5.20

$\lvert a-b\rvert$	MSEloss(a,b)
1	1
$\sim 3\cdot 10^{-1}$	$1\cdot 10^{-1}$
$1\cdot 10^{-1}$	$1\cdot 10^{-2}$
$\sim 3\cdot 10^{-2}$	$1\cdot 10^{-3}$
$1\cdot 10^{-2}$	$1\cdot 10^{-4}$
$\sim 3\cdot 10^{-3}$	$1\cdot 10^{-5}$
$1\cdot 10^{-3}$	$1\cdot 10^{-6}$
$\sim 3\cdot 10^{-4}$	$1\cdot 10^{-7}$
$1\cdot 10^{-4}$	$1\cdot 10^{-8}$

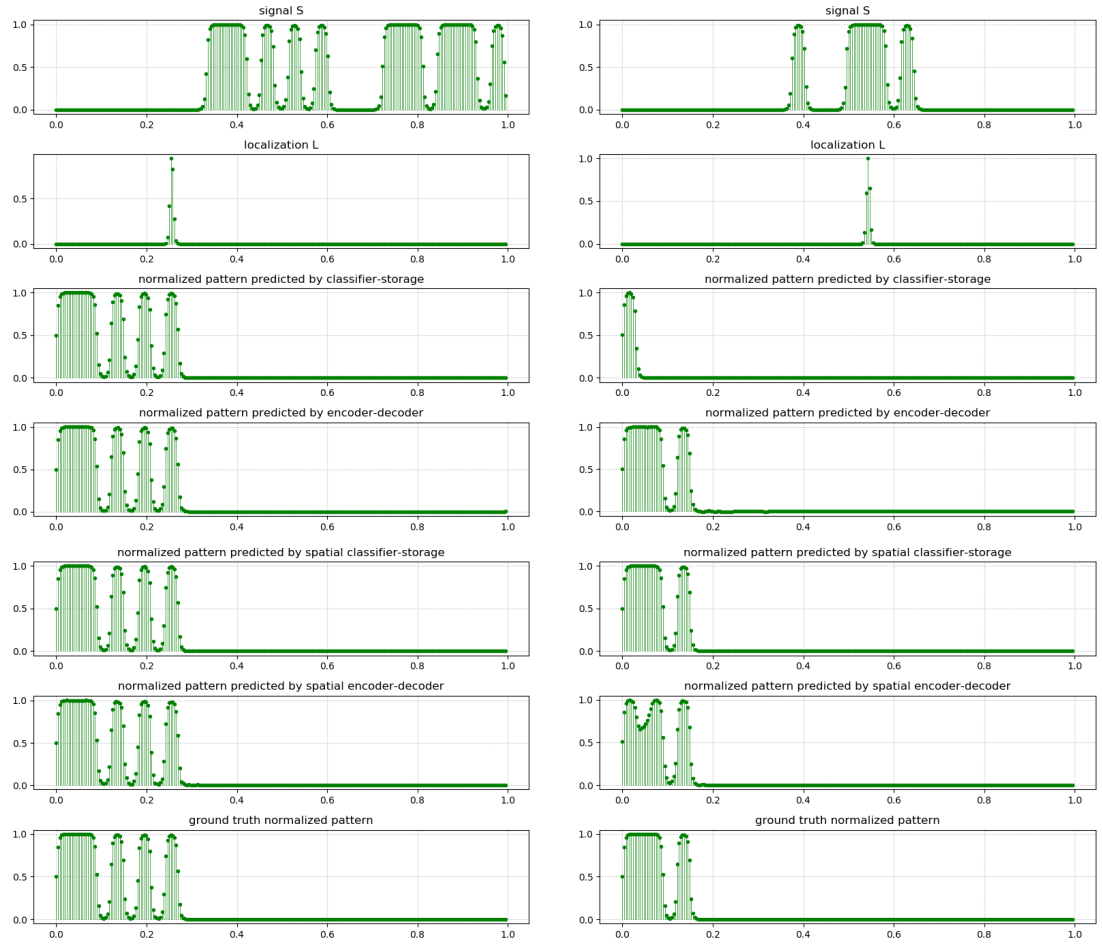
Table 5.18: Conversion between $|a-b|$ and MSEloss(a,b). E.g., if MSEloss(a,b) is in the order of 10^{-5}, then the absolute distance between a and b is approximately in the order of 10^{-3}. If A and B are vectors, and MSEloss(A,B) is in the order of 10^{-5}, then this means that the average componentwise absolute difference between A and B is in the order of 10^{-3}.

shows two examples on the $\mathbf{TMSDL}^{1000}$ dataset; in the first example, the CNN normalization modules do not have any issues with the localization noise or the presence of a second Morse letter, and in the second example, only the classifier-storage architectures makes a gross mistake.
Looking at the accuracy numbers in Table 5.19, the encoder-decoder architecture seems to have a slight edge on the classifier-storage architecture, except for the $\mathbf{MSD}^{1000}$ dataset. The price for this slighlty better accuracy is the loss in efficiency, as the classifier-storage modules work clearly faster than the encoder-decoder modules.
The question whether to use the frequency domain representation or the spatial domain representation as input for the CNNs cannot clearly be answered; in three out of four cases, the $\mathbf{NM}^{\text{class-storage-fd}}$ module has a better accuracy than the $\mathbf{NM}^{\text{class-storage-sd}}$ module, and, also in three out of four cases, the $\mathbf{NM}^{\text{encdec-fd}}$ module has a better accuracy than the $\mathbf{NM}^{\text{encdec-sd}}$ module, so it seems like using the frequency domain representation works better than using the spatial domain representation as input for the CNN. However, the differences are very small. A better argument for the frequency domain representation is that the concept of convolving S with the reverse of L in order to shift S to the left only works in the frequency domain, due to its periodicity properties, and we have now seen that applying an IFFT before handing the convolved signal to the CNN is superfluous.

	Average Normalization Error				Average timing
	$\mathbf{MSD}^{1000}$	$\mathbf{MSDL}^{1000}$	$\mathbf{TMSD}^{1000}$	$\mathbf{TMSDL}^{1000}$	
$\mathbf{NM}^{\text{div-fd}}$	$1.396 \cdot 10^{-3}$	$6.782 \cdot 10^{-2}$	$6.803 \cdot 10^{-2}$	$1.305 \cdot 10^{-1}$	0.474ms
$\mathbf{NM}^{\text{conv}}$	$4.217 \cdot 10^{-4}$	$7.225 \cdot 10^{-2}$	$7.811 \cdot 10^{-2}$	$1.483 \cdot 10^{-1}$	0.139ms
$\mathbf{NM}^{\text{class-storage-fd}}$	$\mathbf{1.456 \cdot 10^{-8}}$	$1.333 \cdot 10^{-4}$	$9.057 \cdot 10^{-4}$	$5.078 \cdot 10^{-3}$	0.454ms
↪ misclassifications	0%	0.5%	2.8%	10.4%	
$\mathbf{NM}^{\text{class-storage-sd}}$	$1.262 \cdot 10^{-5}$	$3.799 \cdot 10^{-5}$	$3.938 \cdot 10^{-3}$	$5.374 \cdot 10^{-3}$	0.464ms
↪ misclassifications	0.1%	0.3%	9.1%	10.7%	
$\mathbf{NM}^{\text{encdec-fd}}$	$7.447 \cdot 10^{-6}$	$7.523 \cdot 10^{-5}$	$\mathbf{2.344 \cdot 10^{-4}}$	$\mathbf{2.971 \cdot 10^{-3}}$	0.774ms
$\mathbf{NM}^{\text{encdec-sd}}$	$8.988 \cdot 10^{-6}$	$\mathbf{3.307 \cdot 10^{-5}}$	$3.415 \cdot 10^{-4}$	$3.085 \cdot 10^{-3}$	0.778ms

Table 5.19: Evaluation of the Normalization Modules. The normalization errors are the average MSEloss the respective module produces on a sample of the respective dataset. The time measurements show the average time the respective module needs to perform a single forward pass, which is the same on all datasets, because all have the same discretization resolution $N = 256$. Marked in bold are the highest accuracies for the four datasets.

In conclusion, using the properties of the frequency domain enabled us to employ relatively simplistic CNN architectures which give good accuracy.



(a) Both the classifier-storage and the encoder-decoder modules make accurate predictions, despite the localization noise and presence of a second Morse letter in the signal.

(b) The frequency-domain-classifier-storage misclassifies the letter, and the spatial-domain-encoder-decoder also makes a mistake. The other two modules make accurate predictions.

Figure 5.20: Example outputs for the classifier-storage and encoder-decoder architectures. The first row shows a signal sample from the $\mathbf{TMSDL}^{1000}$ dataset, the second row shows the imprecise localization. Third row: prediction by the $\mathbf{NM}^{\text{class-storage-fd}}$ module, fourth row: prediction by the $\mathbf{NM}^{\text{encdec-fd}}$ module, fifth row: prediction by the $\mathbf{NM}^{\text{class-storage-sd}}$ module, sixth row: prediction by the $\mathbf{NM}^{\text{encdec-sd}}$ module, last row: ground truth normalized pattern.

5.4 Iterative Module

The final signal decomposition task is the **Simultaneous Localization and Normalization**. It is defined as:

Given only the signal, we want to extract both the localization of the interesting pattern and, at the same time, a canonical representation of that pattern.

In this task, we are only given the input signal S and need to extract both the localization parameter τ and the normalized pattern P for that location.

5.4.1 Alternating Normalization and Localization

Construction

Our approach to this problem is to make use of the localization modules and normalization modules previously discussed. More specific, we alternatingly run a normalization module and a localization module. With each iteration, the normalization module will produce an improved guess of the normalized pattern, which will be used by the localization module to produce an improved guess of the localization, which, in turn, will be used by the normalization module to produce an improved guess of the normalized pattern, which, in turn, will be used by the localization module to produce an improved guess of the localization, and so on. Since we use this iterative method, this solution will be called **Iterative Module** (**IM**). A special advantage of this iterative design is that it allows the module to be used inside a deep CNN, where the individual iterations correspond to consecutive layers. Algorithm 1 describes how we alternatingly run the normalization module and the localization module.

Note that the localization module returns a single number that estimates the ground truth translational parameter τ, rather than an array that estimates the ground truth localization array L, so we need to define a method called *makeGaussian* which converts the single number into a localization array. This is done because the normalization module takes a localization array as input, not a single localization number.

At this point, it becomes obvious why we wanted our normalization module to be robust towards localization noise: The initial localization L_0 is chosen randomly, so in the first iteration, the normalization module gets a possibly completely imprecise localization as input. We want the normalization module to output something sensible even in the first iteration, which can only be achieved if the normalization module knows how to deal with imprecise localizations.

Algorithm 1: First version of the Iterative Module.

Input : **NM** – a normalization module
LM – a localization module
S – input discrete signal
N – length of S
M – number of iterations
Output: ρ – estimate of the ground truth localization parameter τ
Q – estimate of the ground truth normalized pattern P

def `makeGaussian`(μ) :
    $g \leftarrow$ continuous Gaussian curve with mean μ and standard deviation 0.004
    $G \leftarrow \{g(j \cdot N^{-1})\}_{j=0,\dots,N-1}$
    return G

$r \leftarrow$ uniform random number in the interval $[0, 1]$
$L_0 \leftarrow$ `makeGaussian`(r)
for $k = 1, \dots, M$ **do**
    $Q_k \leftarrow \mathbf{NM}(S, L_{k-1})$
    $\rho_k \leftarrow \mathbf{LM}(S, Q_k)$
    $L_k \leftarrow$ `makeGaussian`(ρ_k)
end
$\rho \leftarrow \rho_M$
$Q \leftarrow Q_M$
return ρ, Q

Evaluation

We test the iterative module architecture of Algorithm 1 on the two datasets $\mathbf{MSD}^{1000}$ and $\mathbf{TMSD}^{1000}$. Given the signal S, the **IM** module will produce an output pair (ρ, Q). If the ground truth localization-normalization pair is (τ, P), then the error will be the pair $(\text{Closs}(\rho, \tau), \text{MSEloss}(Q, P))$. Note that for the Two Morse signal dataset, the **IM** module has to extract the localization-normalization pair for only one of the two letters. For calculating the error, we then use the ground truth that belongs to the letter whose ground truth translation τ is closer to the output ρ.
Beforehand, we need to decide which of the normalization modules from Section 5.3 and which of the localization modules from Section 5.2 we want to utilize. For the normalization module, we choose the $\mathbf{NM}^{\text{encdec}-\text{fd}}$ module, because it showed the best overall accuracy across the four datasets listed in Table 5.19. Particularly, it was able to deal with imprecise localizations, which is important for the first iteration where the localization is randomly chosen. Recall that it was trained for 400 epochs using the special $\mathbf{TMSDL}^{\infty}$ dataset, as described in Section 5.3.2. For the localization module, we can choose either the $\mathbf{LM}^{\text{local}-\text{lf}-\text{CNN}}$ module or

the $\mathbf{LM}^{\text{spatial-CNN}}$ because these two are the most accurate localization modules and the only ones that are able to precisely work on the $\mathbf{TMSD}^{1000}$ dataset; we decided to test the **IM** module with both of these localization modules. There are two differently trained versions for each of these two localization modules; the subscript indicates the training parameters (e.g. $\mathbf{LM}^{\text{local-lf-CNN}}_{\text{TMSD500}}$, is the $\mathbf{LM}^{\text{local-lf-CNN}}$ module that has been trained for 500 epochs on the $\mathbf{TMSD}^{\infty}$ dataset).
We also need to pick a number M, i.e. the number of iterations. We experimentally chose $M = 10$.

	$\mathbf{MSD}^{1000}$		$\mathbf{TMSD}^{1000}$	
	Average Localization error	Average Normalization error	Average Localization error	Average Normalization error
$\mathbf{NM}^{\text{encdec-fd}}$ + $\mathbf{LM}^{\text{local-lf-CNN}}_{\text{MSD40}}$	$\mathbf{2.139 \cdot 10^{-7}}$	$7.441 \cdot 10^{-6}$	$8.321 \cdot 10^{-2}$	$5.037 \cdot 10^{-4}$
$\mathbf{NM}^{\text{encdec-fd}}$ + $\mathbf{LM}^{\text{local-lf-CNN}}_{\text{TMSD500}}$	$2.937 \cdot 10^{-6}$	$\mathbf{7.426 \cdot 10^{-6}}$	$9.480 \cdot 10^{-4}$	$7.192 \cdot 10^{-5}$
$\mathbf{NM}^{\text{encdec-fd}}$ + $\mathbf{LM}^{\text{spatial-CNN}}_{\text{MSD40}}$	$5.782 \cdot 10^{-7}$	$7.449 \cdot 10^{-6}$	$2.402 \cdot 10^{-2}$	$\mathbf{6.518 \cdot 10^{-5}}$
$\mathbf{NM}^{\text{encdec-fd}}$ + $\mathbf{LM}^{\text{spatial-CNN}}_{\text{TMSD300}}$	$1.172 \cdot 10^{-6}$	$7.485 \cdot 10^{-6}$	$\mathbf{8.605 \cdot 10^{-5}}$	$9.449 \cdot 10^{-5}$

Table 5.21: Evaluation of the Iterative Module. The localization errors are the average Closs the respective module produces on a sample of the respective dataset, and the normalization errors are the average MSEloss the respective module produces on a sample of the respective dataset. Marked in bold are the highest accuracies for the respective columns.

Table 5.21 shows the result of this test. We can see that the normalization error seems to be almost independent from the choice of the localization module,

the reason for which being the enormous capability of the $\mathbf{NM}^{\text{encdec}-\text{fd}}$ to deal with imprecise localizations. Unsurprisingly, when using the localization modules that have been trained on the $\mathbf{MSD}^{\infty}$, then the highest accuracy for the $\mathbf{MSD}^{1000}$ dataset is achieved, but they also produce the lowest accuracy on the $\mathbf{TMSD}^{1000}$ dataset. Comparing the local line fitting CNN architectures with the spatial CNN architectures, we can see that they are very close in accuracy for the $\mathbf{MSD}^{1000}$ dataset, but in the presence of a second Morse letter in each signal, the $\mathbf{LM}^{\text{spatial}-\text{CNN}}$ modules produce slightly better results than the $\mathbf{LM}^{\text{local}-\text{lf}-\text{CNN}}$ modules.

5.4.2 Shrinking Windows

Improvement

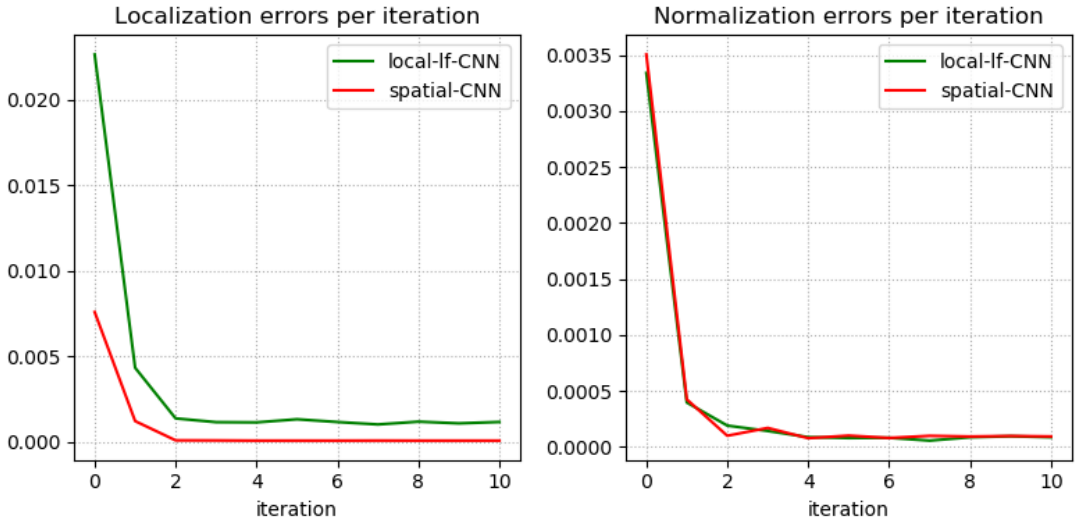


Figure 5.22: Errors per iteration of the Iterative Module. Shown is, for each iteration i, the average error between the output of the i-th iteration and the ground truth. The data comes from a run on the $\mathbf{TMSD}^{1000}$ dataset, where the localization module is either the $\mathbf{LM}^{\text{local}-\text{lf}-\text{CNN}}_{\text{TMSD500}}$ module (green lines) or the $\mathbf{LM}^{\text{spatial}-\text{CNN}}_{\text{TMSD300}}$ module (red lines).

We now try to increase the accuracy of our iterative approach on the Two Morse signal dataset. Figure 5.22 shows how the localization error and the normalization error develops from iteration to iteration when using the **IM** module. It seems like the localization error quickly drops at first but then, it stagnates. However, we want the errors to decrease constantly, so we improve the iterative module by

adding a windowing procedure to each iteration. The goal for our windowing function is to cut away one of the two Morse letters, s.t. the normalization module and the localization module do not have to deal with the second Morse letter. At the beginning of the k-th iteration, the algorithm will construct a window, based on the output localization ρ_{k-1} and output normalized pattern Q_{k-1} of the previous iteration. This window is then applied to the original input signal S, yielding S_k, which is used by the normalization module and the localization module in the k-th iteration instead of S. The window is basically a box function that is 1 inside the interval $[\rho_{k-1}, \rho_{k-1} + w_{k-1}]$, where w_{k-1} is the time length of the normalized Morse letter predicted by Q_{k-1} (e.g. the Morse letter "e" has width 0.03, the morse letter "i" has length 0.09). If the j^*-th component of the array Q_{k-1} is larger than 0.5, and all components with larger indices are smaller than 0.5, then we can set $w_{k-1} = j^*/N$, because it means that the important pattern in Q_{k-1} ends after the j^*-th array index. Now, the windowed signal S_k should only contain the one Morse letter estimated by the pair (ρ_{k-1}, Q_{k-1}). However, this assumes Q_{k-1} and ρ_{k-1} are already very precise approximations of a Morse letter in S. If they are not precise enough, then the windowing cuts off too much, and S_k does not fully contain any of the Morse letters that were present in the original S. So, we add tolerance intervals of length $\beta^k \cdot w_{k-1}$ on both sides of the window, where $\beta \in (0, 1)$ is a window parameter. The interval for the box function now starts at $\rho_{k-1} - \beta^k \cdot w_{k-1}$ and ends at $\rho_{k-1} + w_{k-1} + \beta^k \cdot w_{k-1}$. As the number of iterations increases, i.e. as k becomes larger, β^k will become smaller, so the tolerance intervals will shrink in size, which is why we call this approach **Shrinking Windows Iterative Module** (**SWIM**). It slowly and carefully cuts off more and more parts of S that do not belong to the Morse letter estimated by the pair (ρ_k, Q_k), and the normalization module and localization module cannot constantly jump from one estimation to a completely different estimation. Algorithm 2 describes the implementation of that module.

Note that we account for the periodicity of the frequency domain representation by extending the interval of the windowing function periodically (i.e. if the start of the windowing interval, $\rho_{k-1} - \beta^k \cdot w_{k-1}$, is smaller than 0, then the windowing function additionally assumes 1 on the interval $[1 + \rho_{k-1} - \beta^k \cdot w_{k-1}, 1]$, and if the end of the windowing interval, $\rho_{k-1} + w_{k-1} + \beta^k \cdot w_{k-1}$, is larger than 1, then the windowing function additionally assumes 1 on the interval $[0, \rho_{k-1} + w_{k-1} + \beta^k \cdot w_{k-1} - 1]$). Also, we smoothed the windowing function to prevent discontinuities in the windowed signal (e.g. locations where the signal intensity jumps from 0 to 1). The smoothing is done using the same filter that we have already used in Equation (5.19).
Of course, in the 0-th iteration, which gets a random localization as input, we use

Algorithm 2: Shrinking Windows Iterative Module.

Input : **NM** – a normalization module
LM – a localization module
S – input discrete signal
N – length of S
M – number of iterations
β – window parameter
Output: ρ – estimate of the ground truth localization parameter τ
Q – estimate of the ground truth normalized pattern P

def `makeGaussian`(μ) :
    –defined as in Algorithm 1–

def `smooth`(A) :
    (a_j is the j-th component of A)
    for $n = 0, \dots, N-1$ **do**
        $b_n \leftarrow \frac{1}{4}\left(\sum_{j=0}^{n} 2^{-(n-j)} a_j + \sum_{j=0}^{N-1-n} 2^{-j} a_{n+j}\right)$
    endfor
    $b \leftarrow \max\{b_n \mid n = 0, \dots, N-1\}$
    return $\{b_n / b\}_{n=0,\dots,N-1}$

def `makeWindow`(ρ, Q, k) :
    $j^* \leftarrow \max\{j \in \{0, \dots, N-1\} \mid p_j > 0.5\}$ (p_j is the j-th component of P)
    $w \leftarrow j^*/N$
    start $\leftarrow \rho - \beta^k \cdot w$
    end $\leftarrow \rho + (1 + \beta^k) \cdot w$
    $v \leftarrow$ continuous box function
        that is 1 inside the interval $[\max(0, \text{start}), \min(1, \text{end})]$ and 0 otherwise
    if start < 0 **then**
        v is also 1 in the interval $[1 + \text{start}, 1]$
    if end > 1 **then**
        v is also 1 in the interval $[0, \text{end} - 1]$
    $\widehat{V} \leftarrow \{v(l \cdot N^{-1})\}_{l=0,\dots,N-1}$
    $V \leftarrow$ `smooth`$(\widehat{V})$
    return V

$r \leftarrow$ uniform random number in the interval $[0, 1]$
$L_{\text{init}} \leftarrow$ `makeGaussian`(r)
$Q_0 \leftarrow \mathbf{NM}(S, L_{\text{init}})$, $\rho_0 \leftarrow \mathbf{LM}(S, Q_0)$, $L_0 \leftarrow$ `makeGaussian`(ρ_0) (0-th iteration)
for $k = 1, \dots, M$ **do**
    $V_k \leftarrow$ `makeWindow`(ρ_{k-1}, Q_{k-1}, k)
    $S_k \leftarrow S \circ V_k$ (S_k is the windowed signal in the k-th iteration)
    $Q_k \leftarrow \mathbf{NM}(S_k, L_{k-1})$
    $\rho_k \leftarrow \mathbf{LM}(S_k, Q_k)$
    $L_k \leftarrow$ `makeGaussian`(ρ_k)
end
$\rho \leftarrow \rho_M$
$Q \leftarrow Q_M$
return ρ, Q

the original signal S without the application of a window. In our implementation, for the windowing parameter β, the value 0.85 was experimentally chosen.
Figure 5.25 shows an example of how the windowing does indeed slowly cut away one of the two Morse letters present in the Two Morse signal dataset, and in the end, only one Morse letter is left in the windowed signal. It now becomes clear why we partially trained the normalization modules on signals that contain one Morse letter and only a part of another Morse letter (recall Case 3 in Section 5.3.2): the robustness of the normalization modules in these cases now comes in handy for using them in the **SWIM** module.

Evaluation

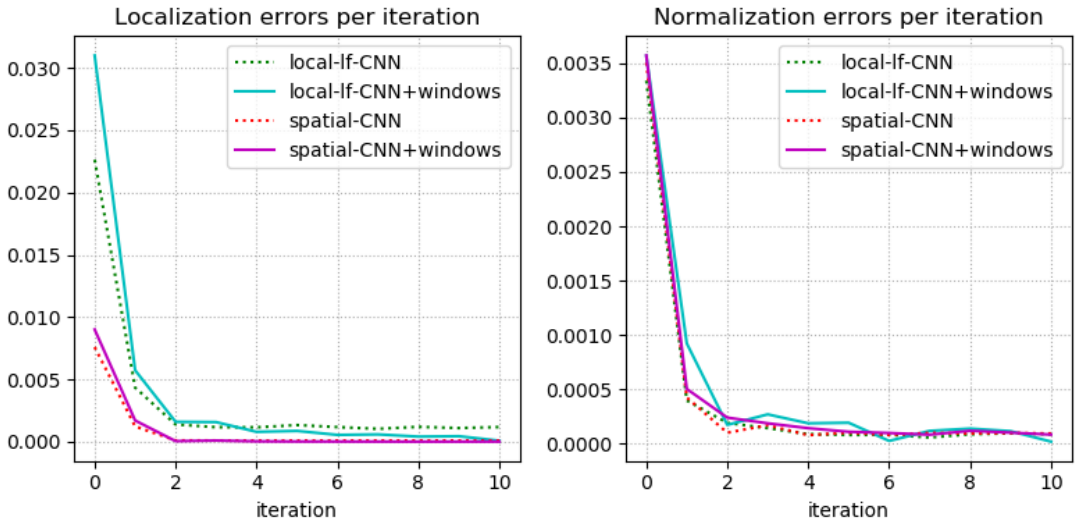


Figure 5.23: Errors per iteration of the Shrinking Windows Iterative Module. Shown is, for each iteration i, the average error between the output of the i-th iteration and the ground truth. The data comes from a run on the $\mathbf{TMSD}^{1000}$ dataset, where the localization module is either the $\mathbf{LM}^{\text{local-lf-CNN}}_{\text{TMSD500}}$ module (cyan lines) or the $\mathbf{LM}^{\text{spatial-CNN}}_{\text{TMSD300}}$ module (magenta lines). The dotted lines show how the errors looked like without the windowing procedure.

Figure 5.23 shows how the error develops from iteration to iteration, this time including the windowing in each iteration. For the $\mathbf{LM}^{\text{local-lf-CNN}}$ module, the localization accuracy does not stagnate anymore, rather it gradually becomes better. The $\mathbf{LM}^{\text{spatial-CNN}}$ module was already very accurate even without the windowing. We tested the **SWIM** module on the $\mathbf{TMSD}^{1000}$ dataset. As before, the normalization module is the $\mathbf{NM}^{\text{encdec-fd}}$ module. For the localization module, we tested the two alternatives $\mathbf{LM}^{\text{local-lf-CNN}}_{\text{TMSD500}}$ and $\mathbf{LM}^{\text{spatial-CNN}}_{\text{TMSD300}}$. Table 5.24 shows the result

	$\mathbf{TMSD}^{1000}$	
	Localization error	Normalization error
$\mathbf{NM}^{\text{encdec}-\text{fd}}$ + $\mathbf{LM}^{\text{local}-\text{lf}-\text{CNN}}_{\text{TMSD500}}$	$8.483 \cdot 10^{-6}$	$\mathbf{8.068 \cdot 10^{-6}}$
$\mathbf{NM}^{\text{encdec}-\text{fd}}$ + $\mathbf{LM}^{\text{spatial}-\text{CNN}}_{\text{TMSD300}}$	$\mathbf{5.021 \cdot 10^{-6}}$	$2.097 \cdot 10^{-5}$

Table 5.24: Evaluation of the Shrinking Windows Iterative Module. The localization errors are the average Closs the respective module produces on a sample of the dataset, and the normalization errors are the average MSEloss the respective module produces on the dataset. Marked in bold are the highest accuracies for the respective columns.

of the test. Comparing it with the results of the **IM** module without windowing (Table 5.21) confirms that the windowing has indeed improved the localization accuracy of both architectures. However, the $\mathbf{LM}^{\text{local}-\text{lf}-\text{CNN}}$ architecture seemingly profited a bit more from the windowing than the $\mathbf{LM}^{\text{spatial}-\text{CNN}}$ architecture, as it was able to improve its localization error from the order of 10^{-4} to 10^{-6}, while the accuracy of the spatial CNN architecture has "only" improved from the order of 10^{-5} to 10^{-6}. However, the local line fitting CNN module seems to have a better synergy with the normalization module: the normalization error is smaller when using the $\mathbf{LM}^{\text{local}-\text{lf}-\text{CNN}}$ module as localizer.

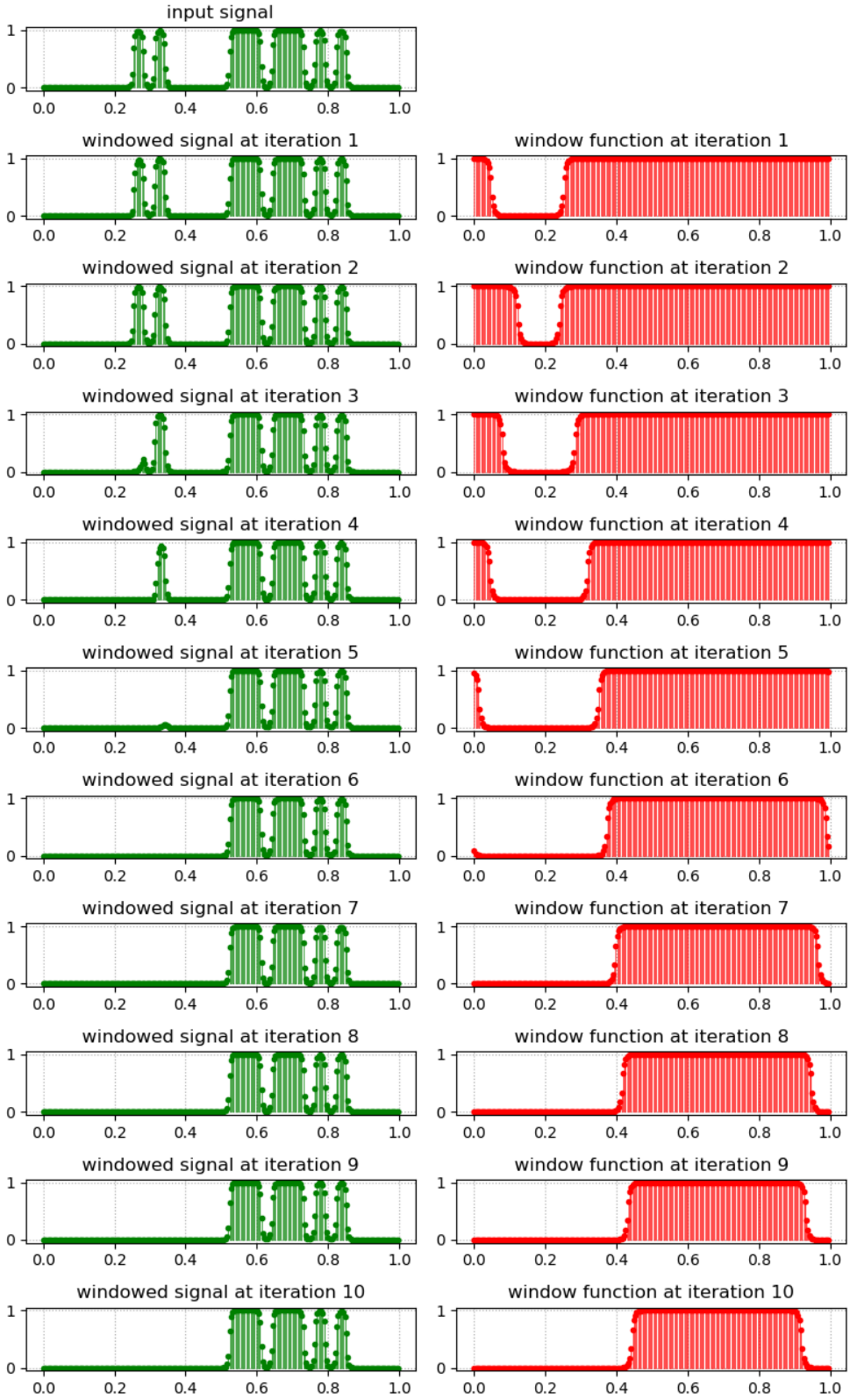


Figure 5.25: Shrinking windows example. It can clearly be seen how the windows (right column) become smaller with each iteration, and how one of the two Morse letters is slowly cut away in the windowed signal (left column).

5.4.3 Simultaneous Extraction of Two Letters

Finally, we try to simultaneously extract both Morse letters from a $\mathbf{TMSD}^{1000}$ sample. An easy solution we found was to make use of the window function: We simultaneously run two copies of the Shrinking Windows Iterative Module, but for each iteration, the window function for the second module is set to be the inverse of the window function calculated by the first module (i.e. if $\{v_j\}_{j=0,\dots,N-1}$ is the window calculated by the first module, then $\{1-v_j\}_{j=0,\dots,N-1}$ is the window used by the second module). This means that the two modules always see different parts of the input signal S, and ideally, in the end, the windowed signal for the first module contains only one Morse letter, and the windowed signal for the second module contains exactly the other Morse letter. So, the output of the first module should be the (localization, normalized pattern) pair for one Morse letter in the signal, and the output of the second module should be the (localization, normalized pattern) pair that belongs to the other Morse letter. Figure 5.27 shows an example output where both letters were successfully extracted by this method.

Evaluation

The testing dataset was the $\mathbf{TMSD}^{1000}$. For each sample, the ground truth is given as two pairs (τ^1, P^1) and (τ^2, P^2), and the module outputs two pairs (ρ^1, Q^1) and (ρ^2, Q^2). The error for the first pair is defined as

$$\text{Error}((\rho^1, Q^1)) = \begin{cases} (\text{Closs}(\rho^1, \tau^1), \text{MSEloss}(Q^1, P^1)) \ , & \text{if Closs}(\rho^1, \tau^1) < \text{Closs}(\rho^1, \tau^2) \\ (\text{Closs}(\rho^1, \tau^2), \text{MSEloss}(Q^1, P^2)) \ , & \text{else.} \end{cases}$$

The error for the second output, $\text{Error}((\rho^2, Q^2))$, is always calculated using the ground truth pair that has not been used by the first output. So, if both output pairs estimate the same Morse letter, then the error cannot be small.
Table 5.26 shows the results for the simultaneous extraction of two Morse letters using the **SWIM** module. It confirms that both localization-normalization pairs can be extracted with high accuracy. Note that the errors are slightly larger than in Table 5.24. The reason for this is that sometimes, the window V does not cut off the entire second Morse letter, s.t. the inverse window $1-V$ does not cover the entire second Morse letter, s.t. the modules that have to deal with the signal windowed by $1-V$ only get one incomplete letter in their signal input. This leads to inaccuracies. The numbers of the normalization errors support the notion that the combination of $\mathbf{NM}^{\text{encdec}-\text{fd}}$ with $\mathbf{LM}^{\text{local}-\text{lf}-\text{CNN}}$ produces better normalized patterns than the combination of $\mathbf{NM}^{\text{encdec}-\text{fd}}$ with $\mathbf{LM}^{\text{spatial}-\text{CNN}}$.

	$\mathbf{TMSD}^{1000}$			
	first letter		second letter	
	Average Localization error	Average Normalization error	Average Localization error	Average Normalization error
$\mathbf{NM}^{\text{encdec-fd}}$ + $\mathbf{LM}^{\text{local-lf-CNN}}_{\text{TMSD500}}$	$1.301 \cdot 10^{-5}$	$\mathbf{8.755 \cdot 10^{-6}}$	$\mathbf{1.706 \cdot 10^{-5}}$	$\mathbf{7.041 \cdot 10^{-5}}$
$\mathbf{NM}^{\text{encdec-fd}}$ + $\mathbf{LM}^{\text{spatial-CNN}}_{\text{TMSD300}}$	$\mathbf{1.054 \cdot 10^{-5}}$	$2.135 \cdot 10^{-5}$	$7.433 \cdot 10^{-5}$	$1.042 \cdot 10^{-4}$

Table 5.26: Evaluation of the simultaneous extraction of two letters. The localization errors are the average Closs the respective module produces on a sample of the respective letter, and the normalization errors are the average MSEloss the respective module produces on a sample of the respective letter. Marked in bold are the highest accuracies for the respective columns.

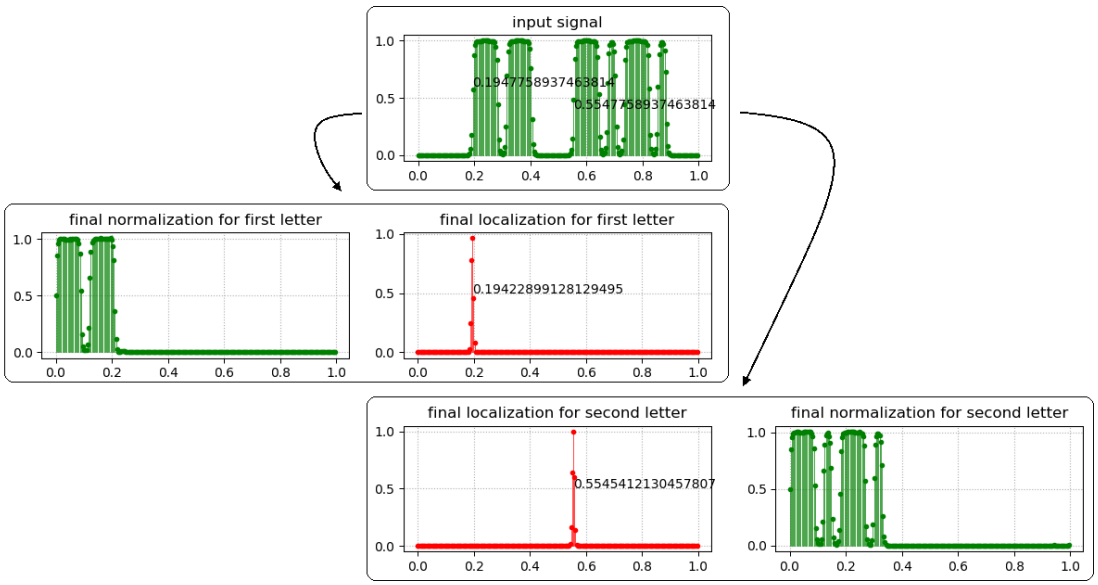


Figure 5.27: Example for extracting two letters at once. The module found accurate predictions for both localization-normalization pairs.

Outlook

If there were more than two Morse letters in one signal, then the basic idea would be to recursively apply the same method, e.g. in the case of three Morse letters, for each iteration, the first **SWIM** module produces a window V^1 as always and uses the signal windowed by V^1. The second module also calculates its own window V^2 according to the usual **SWIM** window making procedure, using its own output from the previous iteration, and uses the signal windowed by $(1 - V^1) \circ V^2$. The third **SWIM** module simply uses the signal windowed by $(1 - V^1) \circ (1 - V^2)$, which only contains the letter that is neither covered by the first module nor by the second. Generalization to more Morse letters is straightforward: if we know the number of Morse letters K, then during each iteration, the k-th module $(k = 1, \ldots, K)$ calculates the window V^k and uses the signal windowed by

$$V^k \circ \prod_{j=1}^{k-1} (1 - V^j)$$

as input for its normalization module and localization module, where the K-th window V^K can simply be set to be 1 everywhere. That way, after a large enough number of iterations, each Morse letter is visible to exactly one module, and every module can only see one letter.
However, this assumes that the normalization module and the localization module can be trained to deal with signals that contain up to K letters (which we have not tested for $K > 2$), and it requires the fine-tuning of both the windowing parameter β and the number of iterations.

6 Extension to 2D

In this chapter, we will briefly demonstrate that our local line fitting approach is also capable of solving the localization task for two-dimensional images.

6.1 Shifted MNIST dataset

Here, we make use of the well-known MNIST[1] dataset of handwritten digits. Each sample is a greyscale image of size 28×28 of one handwritten digit between 0 and 9. Recall that the localization task consists of taking an image and a normalized version of a pattern in that image, and calculating the translation between the two images. Since the original MNIST dataset is constructed to be used for classification rather than image registration, we manipulate it and create the **Shifted MNIST** dataset. Srivastava et al. [36] also implemented a dataset that contains frames with randomly shifted MNIST digits, but they do not consider sub-pixel shifts.
Let S be a 28×28 MNIST sample. A **Shifted MNIST** sample is constructed the following way: First, a resolution $r > 28$ is chosen. The shifted MNIST sample then contains two $r \times r$ frames: The first one is the **normalized pattern** P, with

$$P_{i,j} = \begin{cases} S_{i,j} \ , \text{ if } i,j \in \{0,\dots,27\} \\ 0 \ , \text{ else,} \end{cases} \qquad \text{for all } i,j \in \{0,\dots,r-1\}$$

i.e. the top left 28×28 corner area of P is equal to S and the rest is black.
The second frame is the **translated pattern** X. A real-valued 2D-translation $\Delta = (\Delta_y, \Delta_x)$ is uniformly randomly chosen from the set $[0, r-28] \times [0, r-28]$, and S is shifted inside a $r \times r$ frame by Δ to obtain X. This ensures that the shifted S does not exceed the borders of X and is always fully inside. Since Δ is real-valued, we bilinearly interpolate the pixel values of X:

$$X_{i,j} = \sum_{\substack{k \in \{\Delta_y^{\text{int}}, \Delta_y^{\text{int}}+1\} \\ l \in \{\Delta_x^{\text{int}}, \Delta_x^{\text{int}}+1\}}} (1 - |k - \Delta_y|)(1 - |l - \Delta_x|) S_{i-k,j-l}, \tag{6.1}$$

[1] `http://yann.lecun.com/exdb/mnist/`

where $\Delta_y^{\text{int}}, \Delta_x^{\text{int}} \in \mathbb{N} \cup \{0\}$ are the integer parts of Δ_y and Δ_x, respectively.
Note that if, in Equation (6.1), $i-k, j-l \notin \{0, \dots, 27\}$, then we implicitly set $S_{i-k,j-l} = 0$.
Figure 6.1 shows some examples of the Shifted MNIST dataset.
Given X and P, the task is now to find an estimate of the ground truth translation Δ.

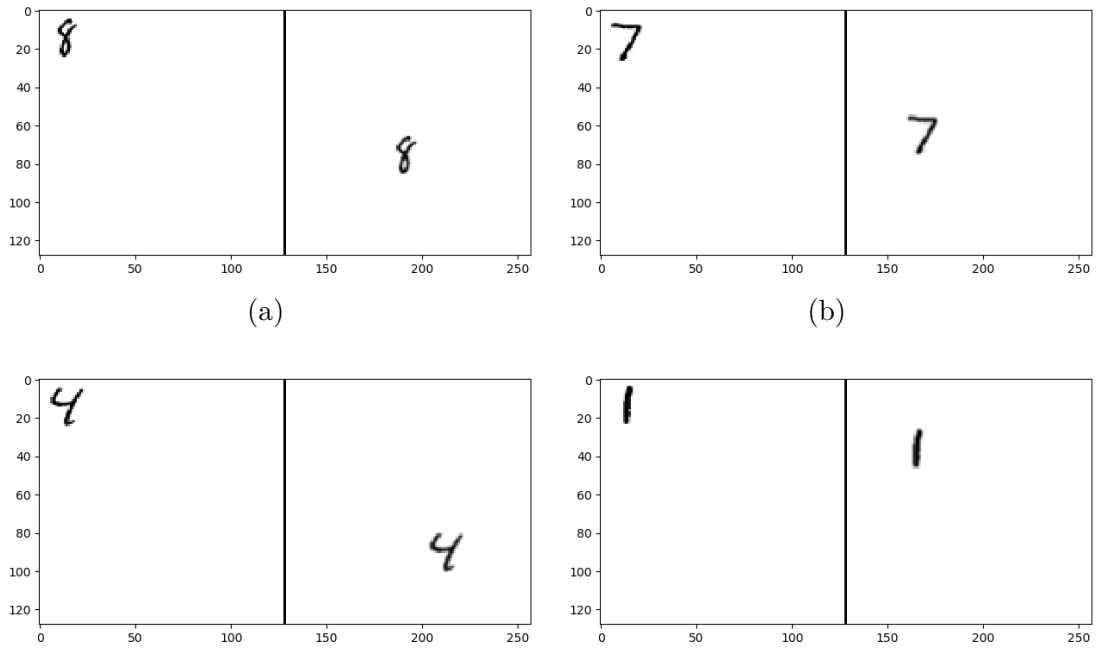


Figure 6.1: Examples for the Shifted MNIST dataset with framesize $r = 128$. For each pair, the left frame is the normalized pattern P and the right frame is the translated pattern X. Note that, for visualization, the images have been inverted, i.e. dark colours belong to high intensity values.

6.2 Localization Modules

For this demonstration, we will take the three modules $\mathbf{LM}^{\text{pc}}$, $\mathbf{LM}^{\text{local-lf}}$, and $\mathbf{LM}^{\text{spatial-CNN}}$, and transform them into 2D modules.

Phase Correlation

The very basic phase correlation image registration serves as a baseline for this experiment. Given the translated pattern X and the normalized pattern P, we

calculate, as before, the cross-power spectrum

$$\mathrm{CPS}(X, P) = \frac{\mathrm{FFT}(X) \circ \overline{\mathrm{FFT}(P)}}{|\mathrm{FFT}(X) \circ \overline{\mathrm{FFT}(P)}|}, \tag{6.2}$$

transfer it back to the spatial domain and take its argmax pixel location

$$\gamma = \mathrm{argmax}(\mathrm{IFFT}(\mathrm{CPS}(X, P))), \tag{6.3}$$

which is the output of the phase correlation module $\mathbf{LM}^{\mathrm{pc-2D}}$. There is no difference to the one-dimensional phase correlation method, because all operations in Equation (6.2) are componentwise, so it does not matter whether its input is 1D or 2D. Note that in the two-dimensional case, it uses the two-dimensional FFT whose entry at position i, j gives information about how a 2D-sinusoid wave of frequency $\sqrt{i^2 + j^2}$, oscillating in direction $(i, j)^T$, contributes to the input of the FFT. The two-dimensional FFT has the same properties as described in Section 3.3, as the one-dimensional FFT. Figure 6.2 shows some examples of how the IFFT of the cross-power spectrum shows one single peak, around the location of Δ.

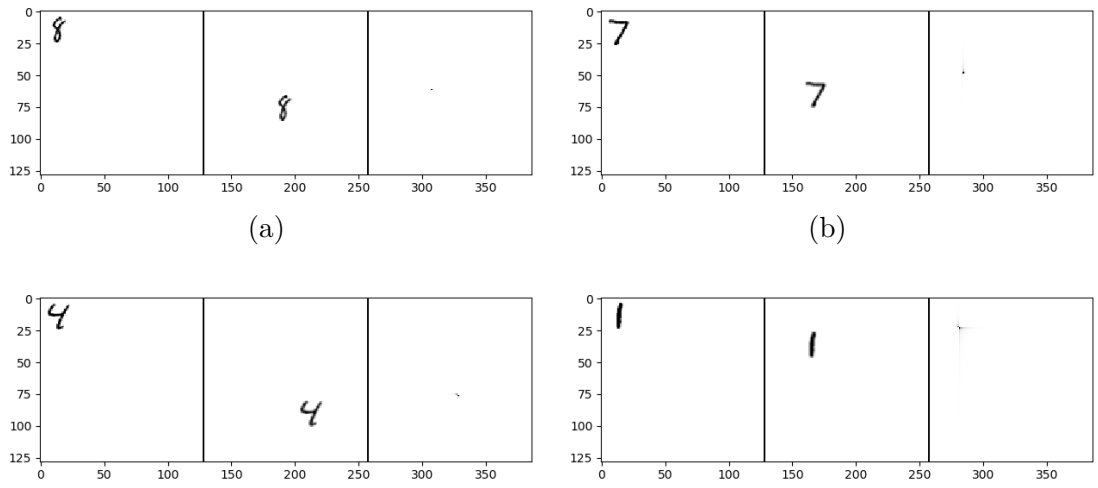


Figure 6.2: Examples for the 2D phase correlation with framesize $r = 128$. For each subfigure, the leftmost frame is the normalized pattern P, the middle frame is the translated pattern X, and the rightmost frame is the phase correlation $\mathrm{IFFT}(\mathrm{CPS}(X, P))$. Note that, for visualization, the images have been inverted, i.e. dark colours belong to high intensity values.

Local Line Fitting

Transferring the $\mathbf{LM}^{\mathrm{local-lf}}$ module to 2D is straightforward. We compute $\mathrm{CPS}(X, P)$ according to Equation (6.2), take its phase spectrum $\mathrm{phases}(\mathrm{CPS}(X, P))$, whose entry at position i, j will be called $\mathrm{phase}_{i,j}$, and compute the local gradients in both x-direction and y-direction:

$$\mathrm{grad}^y_{i,j} = \frac{\mathrm{phase}_{i+1,j} - \mathrm{phase}_{i,j}}{i+1-i} \tag{6.4}$$

$$\mathrm{grad}^x_{i,j} = \frac{\mathrm{phase}_{i,j+1} - \mathrm{phase}_{i,j}}{j+1-j}. \tag{6.5}$$

We then divide the gradients by -2π, as we did in the one-dimensional case:

$$\rho^y_{i,j} = -\frac{1}{2\pi}\mathrm{grad}^y_{i,j} \tag{6.6}$$

$$\rho^x_{i,j} = -\frac{1}{2\pi}\mathrm{grad}^x_{i,j} \tag{6.7}$$

and compute weights $w^x_{i,j}$ and $w^y_{i,j}$ from the normalized amplitude spectra of X and P:

$$w^y_{i,j} = \mathrm{amp}_{i+1,j} + \mathrm{amp}_{i,j} \tag{6.8}$$

$$w^x_{i,j} = \mathrm{amp}_{i,j+1} + \mathrm{amp}_{i,j} \tag{6.9}$$

where

$$\mathrm{amp}_{i,j} = \frac{\mathrm{amplitude}_{i,j}(\mathrm{FFT}(X)) \cdot \mathrm{amplitude}_{i,j}(\mathrm{FFT}(P))}{\left(\sum_{k,l=0}^{r-1} \mathrm{amplitude}_{k,l}(\mathrm{FFT}(X))\right) \cdot \left(\sum_{k,l=0}^{r-1} \mathrm{amplitude}_{k,l}(\mathrm{FFT}(P))\right)}.$$

The weights are then also normalized, i.e.

$$\widehat{w^y_{i,j}} = \frac{w^y_{i,j}}{\sum_{k,l=0}^{r-1} w^y_{k,l}} \tag{6.10}$$

$$\widehat{w^x_{i,j}} = \frac{w^x_{i,j}}{\sum_{k,l=0}^{r-1} w^x_{k,l}} \tag{6.11}$$

and the gradients $\rho^y_{i,j}$ and $\rho^x_{i,j}$ are projected to the unit circle

$$\widehat{\rho^y_{i,j}} = (\cos(2\pi\rho^y_{i,j}), \sin(2\pi\rho^y_{i,j}))^T \tag{6.12}$$

$$\widehat{\rho^x_{i,j}} = (\cos(2\pi\rho^x_{i,j}), \sin(2\pi\rho^x_{i,j}))^T. \tag{6.13}$$

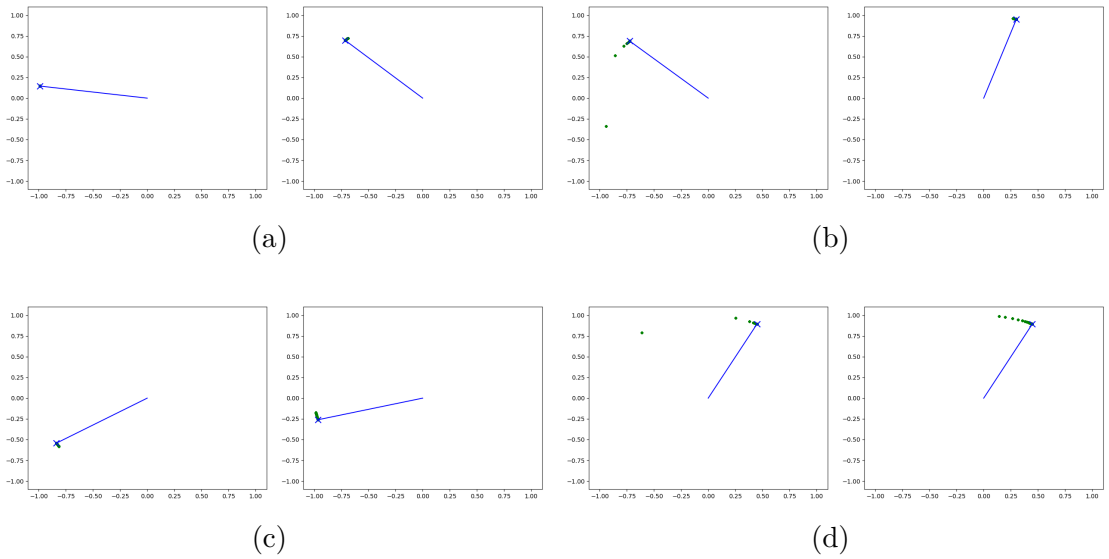


Figure 6.3: Local gradients projected to the unit circle in the 2D case with framesize $r = 128$. In each subfigure, the left graph shows the points $\widehat{\rho^y_{i,j}}$ that estimate Δ_y and the right graph shows the points $\widehat{\rho^x_{i,j}}$ that estimate Δ_x. Marked as a blue cross is the point where the ground truth is located. In some cases, the local gradients are extremely densely clustered around the ground truth.

Figure 6.3 shows how those unit circle points are clustered around the ground truth. The points are clustered very closely around the ground truth, possibly because 2D images contain so much information that the phase part of the cross-power spectrum is hardly disturbed by noise and aliasing. Finally, the unit circle points are weighted:

$$\widehat{\gamma}_y = \sum_{i,j=0}^{r-1} \widehat{w^y_{i,j}}\widehat{\rho^y_{i,j}} \tag{6.14}$$

$$\widehat{\gamma}_x = \sum_{i,j=0}^{r-1} \widehat{w^x_{i,j}}\widehat{\rho^x_{i,j}}. \tag{6.15}$$

Then, the angles $\text{angle}(\widehat{\gamma}_y), \text{angle}(\widehat{\gamma}_x) \in [0.2\pi)$ are extracted, divided by 2π and multiplied by r to get an estimate $\gamma := (\gamma_y, \gamma_x)$ of $\Delta = (\Delta_y, \Delta_x)$:

$$\gamma_y = \frac{r}{2\pi}\text{angle}(\widehat{\gamma}_y) \tag{6.16}$$

$$\gamma_x = \frac{r}{2\pi}\text{angle}(\widehat{\gamma}_x). \tag{6.17}$$

γ is the output of our $\mathbf{LM}^{\text{local-lf-2D}}$ module. The only difference to the one-dimensional module is that we have to keep track of both x-direction and y-direction.

Cross-correlation

Recall that the $\mathbf{LM}^{\text{spatial-CNN}}$ module computed the cross-correlation $X \star P$ between the inputs X and P, handed it to a CNN to compute weights and used those to compute the weighted sum over all pixel locations. For the two-dimensional case, we discard the CNN and use the two-dimensional cross-correlation itself to weight the pixel locations, which saves time:

$$\gamma_y = \frac{\sum_{i,j=0}^{r-1} (X \star P)_{i,j} \cdot i}{\sum_{k,l=0}^{r-1} (X \star P)_{k,l}} - 14 \tag{6.18}$$

$$\gamma_x = \frac{\sum_{i,j=0}^{r-1} (X \star P)_{i,j} \cdot j}{\sum_{k,l=0}^{r-1} (X \star P)_{k,l}} - 14. \tag{6.19}$$

$\gamma = (\gamma_y, \gamma_x)$ is the output of our spatial cross-correlation module $\mathbf{LM}^{\text{cc-2D}}$. Figure 6.4 shows some examples of how the cross-correlation $X \star P$ points at the region of the image where normalized pattern and the translated pattern match best. Notice that due to our implementation, this highlighted region is centered around the center of the digit, which is at position $(\Delta_y + 14, \Delta_x + 14)$, because the digits in the original 28×28 MNIST frames S are centered around $(14, 14)$. This explains why we subtract 14 in Equation (6.18) and Equation (6.19). It is desirable to have the highlighted region being centered around $(\Delta_y + 14, \Delta_x + 14)$ instead of (Δ_y, Δ_x), because otherwise, for small Δ_y or Δ_x, the highlighted region would be cut off at the border of the frame, thereby not being centered around the global maximum peak anymore, which would cause a loss of precision when calculating the weighted sums in Equation (6.18) and Equation (6.19).

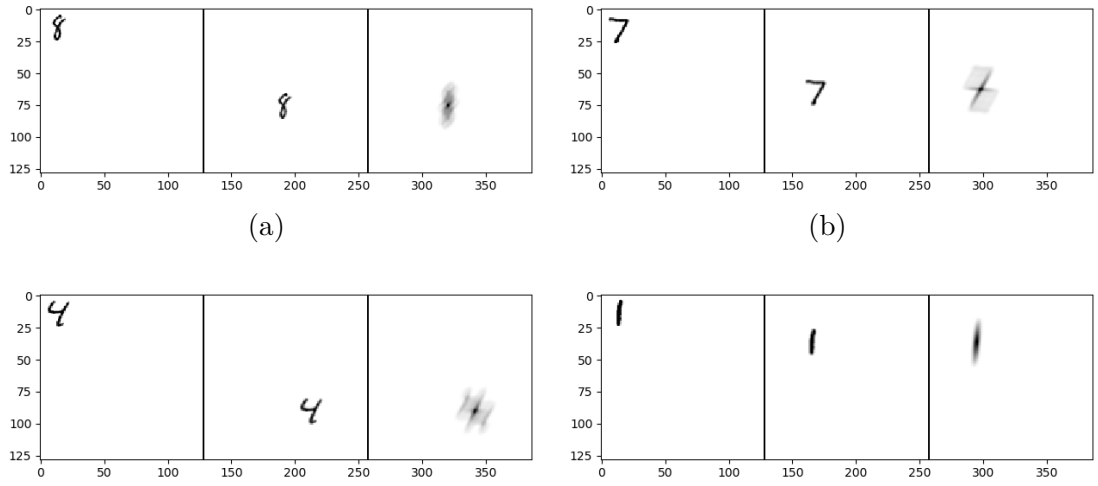


Figure 6.4: Examples for the 2D cross-correlation with framesize $r = 128$. For each subfigure, the left frame is the normalized pattern P, the middle frame is the translated pattern X, and the right frame is the cross-correlation $X \star P$. Note that, for visualization, the images have been inverted, i.e. dark colours belong to high intensity values.

6.3 Evaluation

We tested our three modules on the Shifted MNIST dataset that is obtained when applying the procedure described in Section 6.1 to the testing dataset from MNIST, which results in 10000 samples. Each module gets as input the translated pattern X and the normalized pattern P, both having size $r \times r$ with $r = 256$, and outputs an estimate $\gamma = (\gamma_y, \gamma_x)$ of the ground truth translation $\Delta = (\Delta_y, \Delta_x)$. The error is measured using the Closs:

$$\text{Error}(\gamma, \Delta) = \left(\text{Closs}\left(\frac{\gamma_y}{r}, \frac{\Delta_y}{r} \right), \text{Closs}\left(\frac{\gamma_x}{r}, \frac{\Delta_x}{r} \right) \right), \tag{6.20}$$

where the divisions by r are necessary because the inputs for Closs need to be in the interval $[0, 1]$ in order for it to output a sensible error value.
We also measured the time the modules need, this time we used an NVIDIA GeForce RTX 2080 Ti GPU with 11 GiB of memory for the computations.

Table 6.5 shows the result of our test. In our Shifted MNIST dataset, the pixel width is equal to $1/r = 1/256 \approx 3.9 \cdot 10^{-3}$, so, according to Table 5.10, a module can be considered to have achieved sub-pixel accuracy if the Closs error is in the order of at most 10^{-6}. The baseline phase correlation method $\mathbf{LM}^{\text{pc}-2\text{D}}$ has, unsurprisingly, the worst accuracy because it is not made to achieve sub-

	Average Error between γ_y and Δ_y	Average Error between γ_x and Δ_x	Average timing per forward pass
$\mathbf{LM}^{\mathrm{pc-2D}}$	$5.023 \cdot 10^{-5}$	$4.953 \cdot 10^{-5}$	1.459ms
$\mathbf{LM}^{\mathrm{local-lf-2D}}$	$\mathbf{1.245 \cdot 10^{-7}}$	$\mathbf{9.050 \cdot 10^{-8}}$	1.358ms
$\mathbf{LM}^{\mathrm{cc-2D}}$	$1.618 \cdot 10^{-6}$	$1.563 \cdot 10^{-7}$	7.811ms

Table 6.5: Evaluation of the 2D Localization Modules. The Errors are measured according to Equation (6.20), split up in x- and y-direction. The time measurements are the average time the respective module needs to perform a single forward pass. Marked in bold are the highest accuracies.

pixel accuracy. Both the $\mathbf{LM}^{\mathrm{cc-2D}}$ module and the $\mathbf{LM}^{\mathrm{local-lf-2D}}$ module achieve sub-pixel accuracy, with the latter reaching the highest precision of all. We can compare Table 6.5 with its counterpart in the one-dimensional case (i.e. the first column of Table 5.11), because in both experiments, the pixel width is equal to 1/256. The comparison shows that the phase-correlation method has the same accuracy for both 1D and 2D, which was expected because the $\mathbf{LM}^{\mathrm{pc}}$ module and the $\mathbf{LM}^{\mathrm{pc-2D}}$ module do exactly the same. The $\mathbf{LM}^{\mathrm{cc-2D}}$ module is not as accurate as the $\mathbf{LM}^{\mathrm{spatial-CNN}}$ module was in the one-dimensional case, because it has no CNN that could otherwise increase the precision. Interestingly, the $\mathbf{LM}^{\mathrm{local-lf-2D}}$ module is more accurate than its 1D counterpart $\mathbf{LM}^{\mathrm{local-lf}}$. The reason for this is that in the 2D case, the local gradients are much denser clustered around the ground truth than in the 1D case, which can be seen when comparing Figure 6.3 with Figure 5.6. This difference is probably caused by the different amount of information in 1D Morse signals compared to 2D images of handwritten digits.

Comparing the speed performances, the $\mathbf{LM}^{\mathrm{local-lf-2D}}$ module is the fastest of them all, closely followed by the $\mathbf{LM}^{\mathrm{pc-2D}}$ module. This is interesting because the phase correlation module is seemingly much less complex than the local line fitting module. Unsurprisingly, the $\mathbf{LM}^{\mathrm{cc-2D}}$ module is the slowest because computing a spatial cross-correlation is expensive.

All in all, the $\mathbf{LM}^{\mathrm{local-lf-2D}}$ module is both the fastest and the most accurate localization module on our Shifted MNIST dataset. We did not include any CNN module in this experiment, but our observations in the one-dimensional case suggest that combining our 2D modules with CNNs would improve their accuracy and their robustness if a second digit was present in the translated pattern frame. If training a CNN is undesirable, then the $\mathbf{LM}^{\mathrm{local-lf-2D}}$ module is a good alternative.

7 Conclusion

In this thesis, we developed and tested a variety of methods to extract semantic information from one-dimensional Morse signals. Most notably, our suggested solutions were able to detect patterns inside a signal with sub-pixel accuracy and extract a precise, canoncial representation of that pattern, even if the signal contains more than just the one pattern. The key contribution was the usage of the frequency domain; we did not rely on state-of-the-art CNN architectures, rather we combined more simplistic CNNs with techniques that make use of the special properties of the frequency domain.

In the localization task, if a signal only contains a single Morse letter, then the local line fitting CNN module, which is a frequency domain approach, was more precise than any other method. However, in the presence of a second Morse letter in the signal, a spatial CNN regularly outperformed that module. In the case of very high resolution signals, running the frequency domain method was significantly faster than using the spatial CNN. In the normalization task, our encoder-decoder CNN that runs on the frequency domain representation showed the best overall performance. Combining the encoder-decoder CNN with either the local line fitting CNN or the spatial CNN localization module in an iterative manner and adding a windowing procedure yields the Shrinking Windows Iterative Module, which is capable of precise simultaneous localization and normalization, even for two Morse letters in one signal at the same time.

Regarding our initial motivation, namely to resolve the insufficiency of CNNs to efficiently deal with spatial transformations (in this case sub-pixel translations), this thesis demonstrated how running a CNN on the frequency domain representation can produce results which are comparable to those of running the CNN on the spatial domain representation. It is up to future work to find frequency domain methods that can be applied to more general spatial transformations, and to design them in a manner such that they are clearly more precise or clearly more efficient than spatial domain methods. Our local line fitting CNN module is a first simple example for a frequency domain method which is comparably accurate as a purely spatial method, while being significantly more efficient if the input array is not very small.

It also remains to be seen how well our modules perform on natural signals,

rather than our artificial Morse signal dataset. We believe that we did not include any methodical steps that make use of the unique characteristics of Morse code, but particularly the limited range of signal intensity values in our dataset samples removed an important factor of additional complexity that is present in most natural signals.

Enabling our modules to work on two-dimensional images would be a natural extension. It should be straightforward to implement this because every computational step we used has an equivalent in 2D, and we have already demonstrated this for the local line fitting module. Another point of extension would be the FFT itself; for the CNN modules, we could make the parameters of the FFT learnable. It could be interesting to investigate whether involving the FFT parameters in the training process would result in any improvement by encoding learned statistics of the input dataset directly in the FFT weights.

List of Figures

List of Tables

Bibliography

[1] Sven Behnke. *Hierarchical Neural Networks for Image Interpretation*. Vol. 2766. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Jan. 2003. ISBN: 3-540-40722-7.

[2] T. Brosch and R. Tam. "Efficient Training of Convolutional Deep Belief Networks in the Frequency Domain for Application to High-Resolution 2D and 3D Images". In: *Neural Computation* 27.1 (Jan. 2015), pp. 211–227. ISSN: 0899-7667.

[3] J. Bruna and S. Mallat. "Invariant Scattering Convolution Networks". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.8 (Aug. 2013), pp. 1872–1886. ISSN: 0162-8828.

[4] Antoni Buades, Bartomeu Coll, and Jean-Michel Morel. "A Review of Image Denoising Algorithms, with a New One". In: *SIAM Journal on Multiscale Modeling and Simulation* 4 (Jan. 2005).

[5] Qin-sheng Chen, Michel Defrise, and F. Deconinck. "Symmetric Phase-Only Matched Filtering of Fourier-Mellin Transforms for Image Registration and Recognition". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16.12 (Dec. 1994), pp. 1156–1168. ISSN: 0162-8828.

[6] James W. Cooley and John W. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series". In: *Mathematics of Computation* 19.90 (1965), pp. 297–301. ISSN: 00255718, 10886842. URL: `https://www.jstor.org/stable/2003354`.

[7] Jifeng Dai, Haozhi Qi, Yuwen Xiong, Yi Ci Li, Guodong Zhang, Han Hu, and Yichen Wei. "Deformable Convolutional Networks". In: *2017 IEEE International Conference on Computer Vision (ICCV)* (2017), pp. 764–773.

[8] Sourya Dey, Keith M. Chugg, and Peter A. Beerel. "Morse Code Datasets for Machine Learning". In: *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)* (2018), pp. 1–7.

[9] Russell E. Muzzolini, Yee-Hong Yang, and Roger Pierson. "Texture characterization using robust statistics". In: *Pattern Recognition* 27 (Jan. 1994), pp. 119–134.

[10] H. Foroosh, J. B. Zerubia, and M. Berthod. "Extension of Phase Correlation to Subpixel Registration". In: *IEEE Transactions on Image Processing* 11.3 (Mar. 2002), pp. 188–200. ISSN: 1057-7149.

[11] Munther Gdeisat and Francis Lilley. *One-Dimensional Phase Unwrapping Problem*. Available at `https://www.ljmu.ac.uk/~/media/files/ljmu/about-us/faculties-and-schools/fet/geri/onedimensionalphaseunwrapping_finalpdf.pdf`. Jan. 2011.

[12] L. Gondara. "Medical Image Denoising Using Convolutional Denoising Autoencoders". In: *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*. Dec. 2016, pp. 241–246.

[13] Hafez Farazi and Sven Behnke. "Frequency Domain Transformer Networks for Video Prediction". In: *European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN)*. Bruges, Belgium, 2019.

[14] K. He, G. Gkioxari, P. Dollár, and R. Girshick. "Mask R-CNN". In: *2017 IEEE International Conference on Computer Vision (ICCV)*. Oct. 2017, pp. 2980–2988.

[15] Geoffrey E. Hinton, Alex Krizhevsky, and Sida D. Wang. "Transforming Auto-Encoders". In: *ICANN*. 2011.

[16] G. Huang, Z. Liu, L. v. d. Maaten, and K. Q. Weinberger. "Densely Connected Convolutional Networks". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. July 2017, pp. 2261–2269.

[17] Max Jaderberg, Karen Simonyan, Andrew Zisserman, and Koray Kavukcuoglu. "Spatial Transformer Networks". In: *Advances in neural information processing systems 28*. Ed. by C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett. Curran Associates, Inc., 2015, pp. 2017–2025. URL: `https://papers.nips.cc/paper/5854-spatial-transformer-networks.pdf`.

[18] Eric R Kandel, James H Schwartz, Thomas M Jessell, Department of Biochemistry, Molecular Biophysics Thomas Jessell, Steven Siegelbaum, and AJ Hudspeth. *Principles of Neural Science*. Vol. 4. McGraw-Hill New York, 2000.

[19] Diederik Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *International Conference on Learning Representations* (Dec. 2014).

[20] C. Kreucher and S. Lakshmanan. "LANA: a lane extraction algorithm that uses frequency domain features". In: *IEEE Transactions on Robotics and Automation* 15.2 (Apr. 1999), pp. 343–350. ISSN: 1042-296X.

[21] C.D. Kuglin and D.C. Hines. "The Phase Correlation Image Alignment Method". In: *Proceedings of the IEEE 1975 International Conference on Cybernetics and Society*. 1975, pp. 163–165.

[22] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-Based Learning Applied to Document Recognition". In: *Proceedings of the IEEE* 86.11 (Nov. 1998), pp. 2278–2324. ISSN: 0018-9219.

[23] J. Li, S. You, and A. Robles-Kelly. "A Frequency Domain Neural Network for Fast Image Super-resolution". In: *2018 International Joint Conference on Neural Networks (IJCNN)*. July 2018, pp. 1–8.

[24] Michaël Mathieu, Mikael Henaff, and Yann LeCun. "Fast Training of Convolutional Networks through FFTs". In: *CoRR* abs/1312.5851 (2013). arXiv: 1312.5851. URL: https://arxiv.org/abs/1312.5851.

[25] NIST/SEMATECH. *e-Handbook of Statistical Methods*. Retrieved 11.08.2019. URL: https://www.itl.nist.gov/div898/handbook/.

[26] Ville Ojansivu and Janne Heikkilä. "Object Recognition Using Frequency Domain Blur Invariant Features". In: *Proceedings of the 15th Scandinavian Conference on Image Analysis*. SCIA'07. Aalborg, Denmark: Springer-Verlag, 2007, pp. 243–252. ISBN: 978-3-540-73039-2.

[27] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. "Automatic differentiation in PyTorch". In: *NIPS-W*. 2017.

[28] Radiocommunication Sector of International Telecommunication Union. *International Morse Code*. Available at https://www.itu.int/rec/R-REC-M.1677-1-200910-I. Recommendation M.1677-1. Oct. 2009.

[29] B. S. Reddy and B. N. Chatterji. "An FFT-Based Technique for Translation, Rotation, and Scale-Invariant Image Registration". In: *IEEE Transactions on Image Processing* 5.8 (Aug. 1996), pp. 1266–1271. ISSN: 1057-7149.

[30] Joseph Redmon and Ali Farhadi. "YOLOv3: An Incremental Improvement". In: *CoRR* abs/1804.02767 (2018). arXiv: 1804.02767. URL: https://arxiv.org/abs/1804.02767.

[31] Oren Rippel, Jasper Snoek, and Ryan P. Adams. "Spectral Representations for Convolutional Neural Networks". In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'15. Montreal, Canada: MIT Press, 2015, pp. 2449–2457.

[32] John C. Russ. *The Image Processing Handbook, Fourth Edition*. 4th. Boca Raton, FL, USA: CRC Press, Inc., 2002. ISBN: 084931142X.

[33] Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton. "Dynamic Routing Between Capsules". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon and U. V. Luxburg and S. Bengio and H. Wallach and R. Fergus and S. Vishwanathan and R. Garnett. Curran Associates, Inc., 2017, pp. 3856–3866. URL: https://papers.nips.cc/paper/6975-dynamic-routing-between-capsules.pdf.

[34] C. E. Shannon. "Communication in the Presence of Noise". In: *Proceedings of the IRE* 37.1 (Jan. 1949), pp. 10–21. ISSN: 0096-8390.

[35] Julius O. Smith. *Mathematics of the Discrete Fourier Transform (DFT), with Audio Applications*. W3K Publishing, 2007. ISBN: 9780974560748. URL: https://ccrma.stanford.edu/~jos/mdft/.

[36] Nitish Srivastava, Elman Mansimov, and Ruslan Salakhutdinov. "Unsupervised Learning of Video Representations using LSTMs". In: *Proceedings of the 32nd International Conference on Machine Learning - Volume 37*. ICML'15. Lille, France, 2015, pp. 843–852. URL: https://www.cs.toronto.edu/~nitish/unsupervised_video/.

[37] Harold S. Stone, Michael T. Orchard, Ee-Chien Chang, and Stephen A. Martucci. "A Fast Direct Fourier-Based Algorithm for Subpixel Registration of Images". In: *IEEE Transactions on Geoscience and Remote Sensing* 39.10 (Oct. 2001), pp. 2235–2243. ISSN: 0196-2892.

[38] Kenji Takita, Takafumi Aoki, Yoshifumi Sasaki, Tatsuo Higuchi, and Koji Kobayashi. "High-Accuracy Subpixel Image Registration Based on Phase-Only Correlation". In: *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E86-A (Aug. 2003).

[39] Pascal Vincent, Hugo Larochelle, Y Bengio, and Pierre-Antoine Manzagol. "Extracting and Composing Robust Features with Denoising Autoencoders". In: *Proceedings of the 25th International Conference on Machine Learning* (Jan. 2008), pp. 1096–1103.

[40] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. "Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion". In: *The Journal of Machine Learning Research* 11 (Dec. 2010), pp. 3371–3408. ISSN: 1532-4435.

[41] Gregory K. Wallace. "The JPEG Still Picture Compression Standard". In: *Commununications of the ACM* 34.4 (Apr. 1991), pp. 30–44. ISSN: 0001-0782. URL: http://doi.acm.org/10.1145/103085.103089.

[42] Yunhe Wang, Chang Xu, Shan You, Dacheng Tao, and Chao Xu. "CNNpack: Packing Convolutional Neural Networks in the Frequency Domain". In: *Advances in Neural Information Processing Systems 29*. Ed. by D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett. Curran Associates, Inc., 2016, pp. 253–261. URL: http://papers.nips.cc/paper/6390-cnnpack-packing-convolutional-neural-networks-in-the-frequency-domain.pdf.

[43] Jason Weston, Sumit Chopra, and Antoine Bordes. "Memory Networks". In: *CoRR* abs/1410.3916 (2015). arXiv: 1410.3916. URL: https://arxiv.org/abs/1410.3916.

[44] Junyuan Xie, Linli Xu, and Enhong Chen. "Image Denoising and Inpainting with Deep Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. Curran Associates, Inc., 2012, pp. 341–349. URL: http://papers.nips.cc/paper/4686-image-denoising-and-inpainting-with-deep-neural-networks.pdf.

[45] Shuochao Yao, Shaohan Hu, Yiran Zhao, Aston Zhang, and Tarek Abdelzaher. "DeepSense: A Unified Deep Learning Framework for Time-Series Mobile Sensing Data Processing". In: *Proceedings of the 26th International Conference on World Wide Web*. WWW '17. Perth, Australia: International World Wide Web Conferences Steering Committee, 2017, pp. 351–360. ISBN: 978-1-4503-4913-0. URL: https://doi.org/10.1145/3038912.3052577.

[46] Shuochao Yao, Ailing Piao, Wenjun Jiang, Yiran Zhao, Huajie Shao, Shengzhong Liu, Dongxin Liu, Jinyang Li, Tianshi Wang, Shaohan Hu, Lu Su, Jiawei Han, and Tarek Abdelzaher. "STFNets: Learning Sensing Signals from the Time-Frequency Perspective with Short-Time Fourier Neural Networks". In: *The World Wide Web Conference*. WWW '19. San Francisco, CA, USA: ACM, 2019, pp. 2192–2202. ISBN: 978-1-4503-6674-8. URL: http://doi.acm.org/10.1145/3308558.3313426.

[47] WeiQiang Zhu, Seyyed Mostafa Mousavi, and Gregory C. Beroza. "Seismic Signal Denoising and Decomposition Using Deep Neural Networks". In: vol. abs/1811.02695. 2018. arXiv: 1811.02695. URL: https://arxiv.org/abs/1811.02695.

[48] Xizhou Zhu, Han Hu, Stephen Lin, and Jifeng Dai. "Deformable ConvNets v2: More Deformable, Better Results". In: *CoRR* abs/1811.11168 (2018). arXiv: 1811.11168. URL: https://arxiv.org/abs/1811.11168.

[49] Barbara Zitová and Jan Flusser. "Image Registration Methods: A Survey". In: *Image and Vision Computing* 21 (Oct. 2003), pp. 977–1000.