8-2018

Understanding 1D Convolutional Neural Networks Using Multiclass Time-Varying Signals

Ravisutha Sakrepatna Srinivasamurthy
Clemson University, ravisutha.s.s@gmail.com

Understanding 1D Convolutional Neural Networks Using Multiclass Time-Varying Signals

---

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Engineering

---

by
Ravisutha Sakrepatna Srinivasamurthy
August 2018

Accepted by:
Dr. Robert J. Schalkoff, Committee Chair
Dr. Harlan B. Russell
Dr. Ilya Safro

Abstract

In recent times, we have seen a surge in usage of Convolutional Neural Networks to solve all kinds of problems - from handwriting recognition to object recognition and from natural language processing to detecting exoplanets. Though the technology has been around for quite some time, there is still a lot of scope to do research on what's really happening 'under the hood' in a CNN model.

CNNs are considered to be black boxes which learn something from complex data and provides desired results. In this thesis, an effort has been made to explain what exactly CNNs are learning by training the network with carefully selected input data. The data considered here are one dimensional time varying signals and hence the 1-D convolutional neural networks are used to train, test and to analyze the learned weights.

The field of digital signal processing (DSP) gives a lot of insight into understanding the seemingly random weights learned by CNN. In particular, the concepts of Fourier transform, Savitzky-Golay filters, Guassian filters and FIR filter design lights up seeming dark alley of CNNs. As a result of this study, a few interesting inferences can be made regarding dropout regularization, optimal kernel length and optimal number of convolution layers.

Acknowledgments

I would like to thank my advisor, Dr. Robert J Schalkoff for his expert advice and for the encouragement without which this work would have never seen a light of day. His courses ECE 8650 and ECE 8720 have greatly helped me in understanding machine learning and artificial neural network concepts in depth and has inspired me to explore uncharted territories. I would also like to thank Dr. Russell and Dr. Safro for being a part of my committee and for their timely suggestions. Finally, I am grateful to all the professors who have taught me so much in the span of past two years.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Objective

The objective of this thesis is to understand the underlying principles and working of convolutional neural networks (CNNs) as applied to selected 1-D time-varying signals. This involves training the CNNs (of different settings) with carefully selected dataset to explain why the CNN has learned certain weights. By doing so, it hopefully gives insights to the reason why CNNs with certain settings works better than the others.

1.2 Evolution of Convolutional Neural Networks

A standard artificial neural network (ANN) is a structure composed of number of interconnected neurons [26]. Each unit performs a (usually non-linear) computation on the input which can also be output of neurons from the previous layer. The most important algorithm which powers ANN training is backpropagation [24]. The layers of a 'standard' ANN model are input layer, hidden layer(s) and an output layer. An example ANN model used for a classification problem is shown in Figure 1.1 where input units are represented by x, hidden units are represented by h and the output units are represented by o. The input to the neural network is mapped to the output during forward propagation based on loss function and optimization algorithm. The weights of the ANN are updated so as to reduce

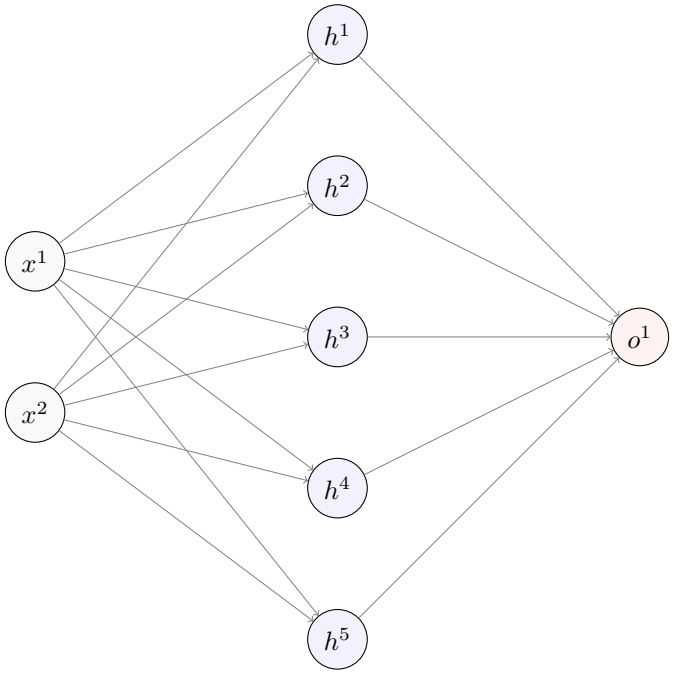


Figure 1.1: A simple ANN model consisting of an input layer, a hidden layer and an output layer

the loss function. The performance of the ANN are measured in the validation phase and a common form of validation is k-fold validation which is employed throughout the thesis.

Although ANN's are pretty good at solving some problems, they suffer from two main challenges. First, *shallow* NNs - neural networks with few hidden layers, are insufficient to solve complex problems and hence you need more hidden layers to get better results. But, *deep* NNs are difficult to train as a result of vanishing gradients. The second problem arises while selecting best features from a high dimensional input (ex: Image dataset). Without extracting the best features, the ANN is prone to overfit the data. It is difficult to extract best features for a particular task and moreover these features have to be hand-picked.

Convolutional neural networks solve the second problem easily as they takes care of feature selection [15] and it generally requires lesser number of neurons as it shares the weight with number of convolution outputs. An example CNN model used for image classification is shown in Figure 1.2. In this CNN model, there area total of 8 layers - 5 convolution layers

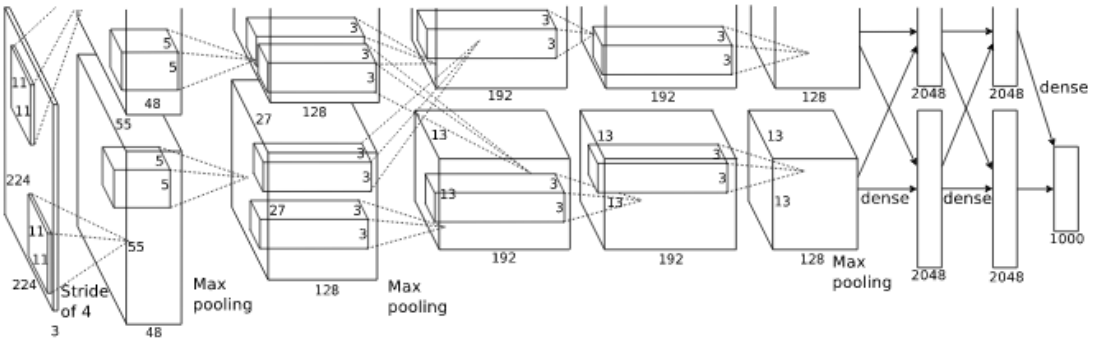


Figure 1.2: ILSVRC-2012 winning CNN model [14]

and 3 dense/fully-connected layers. The outputs from first, second and fifth convolution layers are fed to max pooling layer. The output layer classifies the input data to one of 1000 possible classes.

1.3 Analysis Using DSP Tools

In this thesis, CNNs are trained with the carefully selected time varying signals. These are one-dimensional signals and DSP provides both time domain and frequency domain tools to analyze the weights learned by CNNs.

1.3.1 Time domain analysis

Weights learned by convolution kernels are observed and compared with known filter patterns. A neat explanation for the pattern can be provided for certain problems and these concepts are further discussed in Chapter 3.

1.3.2 Frequency domain analysis

Time domain analysis is insufficient for analyzing most signal processing problems; that is where frequency domain analysis comes in. It provides intuitive explanation for some concepts such as padding and regularizations. Tools like FFT, FIR filter design etc. are used for the analysis and these concepts are introduced in Chapter 4 and are applied in

Chapter 5.

1.4 Related Works

It is been a while ([15]) since the advent of CNN models and yet only a handful of research has gone into understanding CNNs learned behaviors. Having said that, there are several papers which offer insight into the working of CNNs. It is interesting to note that almost all papers base their research on 2D CNNs and their premise for understanding CNNs is usually using the field of information theory.

In [8], a method for analyzing hidden layers by inserting linear classifiers (probes) is discussed. These classifiers use outputs from hidden layers. By analyzing these probes (using concepts from information theory) potential problems can be discovered. Mathematical frameworks for CNN are proposed by [19] and [31] papers using information theory. The paper [31] uses matrix-based Renyi's α -entropy function to analyze dynamics of layers using information flow. Multiscale contractions, linearization of hierarchical symmetries, and sparse separations in convolution layers are analyzed in [19]. Another paper [16] gives a *physics* perspective on CNNs and tries to relate deep learning with statistical mechanics. This paper tries to answer whether neural networks can approximate functions well when set of possible functions are exponentially larger than the set of practically possible networks. This thesis is different from the above papers as it goes all the way to the basic principles. That is, 1D convolutions are explored using tools from well established field of digital signal processing.

1.5 Contributions

In this thesis, an effort has been made to understand the underlying principles of a CNN model. To really understand what CNNs are capable of learning, one has to carefully craft the dataset and observe the weights of a trained CNN model. For this thesis, multiple datasets were created such as sinusoids sampled at different frequencies,

sinusoids of different frequencies, triangular, rectangular waveforms, overlapped triangular and rectangular waveforms, input data containing uniformly distributed noise and much more. The weights are observed in both time and frequency domains. A explanation as to why dropout regularization works will be provided in this thesis. The inferences from time and frequency domain studies lead to the understanding of what happens when you increase the kernel length, relationship between kernel length and number of convolution layers and a guide to choose a better kernel length based on the input data.

This thesis is arranged as follows. The next chapter explains basics of 1D CNNs and also introduces number of hyperparameters which will be later explored in this thesis. Going further, CNNs are analyzed in both time and frequency domain in Chapters 3 and 5 respectively. Inferences based on time and frequency domain studies are made in Chapter 6. Finally, conclusion and possible future works are discussed in Chapter 7.

Chapter 2

Basics of 1D CNN

Convolutional Neural Networks (CNNs) come under the umbrella of Deep Learning, a subset of machine learning that uses multi-layered artificial neural networks to deliver state-of-the-art accuracy in tasks such as object detection, speech recognition, language translation and others [11]. The CNN rose to fame when a CNN model won LSVRC-2012 (Imagenet) competition[14]. Typically, a CNN maps 2D input (images) either to categorical outputs (in case of classification problems) or to a real value (in case of regression based problems). Most of the literature [10, 12, 17, 27, 23, 30] exists for 2D CNNs and in this chapter, 1D CNNs are explored. Typically, a CNN model involves two kinds of layers as shown in Figure 2.1:

1. Convolution layers; and
2. Multilayer Feed Forward (MLFF) layers.

The first section discusses forward propagation, back propagation for both convolution and MLFF layers will be discussed later.

2.1 Forward Propagation

The convolution layer is typically situated just after the input layer. The output from the convolution layer is fed to MLFF network. Convolution layer performs convolution

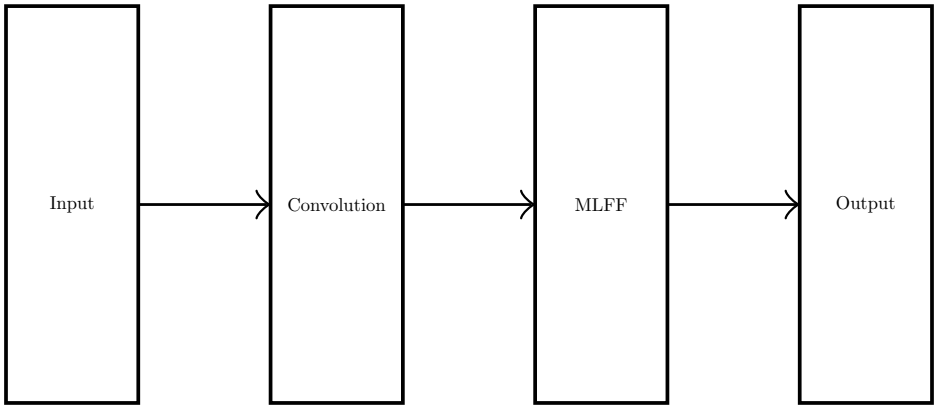


Figure 2.1: A typical CNN structure

(or correlation) along with other operations on the input which can be thought of as feature extractor (discussed in section 2.2) and MLFF can be thought of as decision block which decides either class of the input or predicts a value based on the features extracted by the previous convolution layer. The following sections explore each operation in detail.

CNN models for this thesis are built and tested using a python package called *keras*. This chapter also includes *keras* code along with explanations for certain modules of CNN model. In depth description of the package is provided in Appendix A.

2.2 Convolution Layer

Convolution involves sliding the kernel over the input signal which is also known as shift-compute procedure. This can be done in two ways:

1. Non causal convolution - Cross-correlation (Used in typical CNNs); or
2. Causal convolution.

The characteristics of both types of convolutional layers are discussed in the following subsections along with other important operations such as pooling, activation functions etc..

2.2.1 Non causal convolution - correlation

Non causal convolution is nothing but cross-correlation as described by any standard DSP text book. It is non causal because the output y is dependent on the future input. For example, if the output *y(0)* of a system is dependent on future input *x(1)* then such systems are called non causal systems.

Let the input to convolution layer of length n be represented by x and let the kernel of length k be represented by h. Let the kernel window be shifted s positions (number of strides) after each convolution operation. Then non-causal convolution between x and h for stride s is defined as

$$y(n) = \begin{cases} \sum_{i=0}^{k} x(n+i)h(i), & \text{if } n = 0. \\ \sum_{i=0}^{k} x(n+i+(s-1))h(i), & \text{otherwise.} \end{cases} \tag{2.1}$$

For example, if n = 5, k = 3 and s = 1 then

$$y(0) = x(0)h(0) + x(1)h(1) + x(2)h(2)$$
$$y(1) = x(1)h(0) + x(2)h(1) + x(3)h(2)$$
$$y(2) = x(2)h(0) + x(3)h(1) + x(4)h(2)$$

Notice that the output length is not equal to length of the input and this mode is called "valid" mode in *keras*. If needed, the output length can be made equal to the input length using padding and this mode is called "same" padding convolution in *keras*.

$$o = n \tag{2.2}$$

Without padding and in "valid" mode, the length of the output o for stride s is given by

$$o = \lfloor \frac{(n-k)}{s} \rfloor + 1 \tag{2.3}$$

Keras Code
`model.add (Conv1D (no_kernels, length_of_kernel, activation='relu', padding="same", input_shape=(n_i, 1)))`
`model.add (Conv1D (no_kernels, length_of_kernel, activation='relu', padding="valid", input_shape=(n_i, 1)))`

Table 2.1: Keras non causal convolution code

With padding, the length of output for input of length n, kernel of length k, padding of length p is given by

$$o = \lfloor \frac{(n + 2p - k)}{s} \rfloor + 1 \tag{2.4}$$

2.2.2 Causal Convolution

In causal convolution, the output is not dependent on future inputs. Let the input to convolution layer of length n be represented by x and let the kernel of length k be represented by h. Let the kernel window be shifted s positions (number of strides) after each convolution operation. Then causal convolution between x and h for stride s is defined as

$$y(n) = \begin{cases} \sum_{i=0}^{k} x(n-i)h(i), & \text{if } n = k-1. \\ \sum_{i=0}^{k} x(n-i+(s-1))h(i), & \text{otherwise.} \end{cases} \tag{2.5}$$

As in non causal convolution, we have with and without padding modes and the output length is same as before. Lets consider a without padding example - n = 5, k = 3

Keras Code
`model.add (Conv1D (no_kernels, length_of_kernel, activation='relu', padding="causal", input_shape=(n_i, 1)))`

Table 2.2: Keras causal convolution code

and s = 1 then

$$y(2) = x(2)h(0) + x(1)h(1) + x(0)h(0)$$

$$y(3) = x(3)h(0) + x(2)h(1) + x(1)h(0)$$

$$y(4) = x(4)h(0) + x(3)h(1) + x(2)h(0)$$

Clearly, y is independent of future inputs and hence causal. Because of the causality, the kernel weights can be thought of as impulse response (It is conventional to represent impulse response as h and hence the representation). That is,

$$y(n) = x(n) \circledast h(n) \tag{2.6}$$

The output of the system can be found for any input given the impulse response h(n). In fact, convolution in time domain corresponds to multiplication in frequency domain and this property is extensively used in Chapter 5. The frequency domain counterpart of Equation 2.6 is shown in Equation 2.7.

$$Y(f) = X(f)H(f) \tag{2.7}$$

2.2.3 Relation between causal and non-causal convolutions

Consider Equation 2.1 for $s = 1$ which can be written in terms of Equation 2.5 as follows.

$$\begin{aligned} y(n) &= \sum_{i=0}^{k} x(n+i)h(i) \\ &= \sum_{i=0}^{k} x(n-(-i))h(-i) \end{aligned}$$

Let $j = -i$.

$$y(n) = \sum_{j=0}^{-k} x(n-j)h(j)$$

This can be written in terms of convolution as

$$y(n) = x(n) \circledast h(-n) \tag{2.8}$$

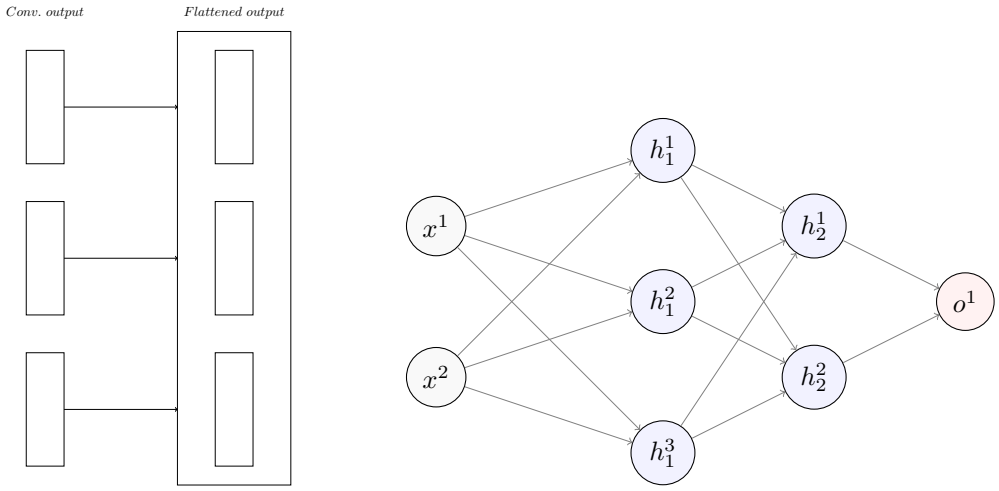
The Fourier Transform for Equation 2.8

$$Y(f) = X(f)H^*(f) \tag{2.9}$$

Form Equations 2.9 and 2.7, it can be shown that magnitude of frequency responses of causal and non-causal convolution output are equal and only differ in phase response.

$$|Y(f)| = |X(f)H(f)| = |X(f)H^*(f)| \tag{2.10}$$

While analyzing the filters, it turns out that the phase component doesn't play a significant role and hence it really doesn't matter whether causal or non-causal filter is chosen. Finally, it can be seen that if the kernel weights $y(n) = y(-n)$ then the convolution operation is same as the correlation.



(a) Flatten output (b) Fully connected network

Figure 2.2: Flatten output and fully connected network

2.2.4 Pooling

Pooling is used to reduce the dimensionality of a given mapping while highlighting the prominent features. Generally, pooling is employed after the convolution layer to reduce the dimension of the convolution output. It also helps to reduce overfitting. The most famous pooling technique is max pooling. Max pooling refers to picking up max value in a window of size f and this window is slid over the input with a stride of length s after each pooling operation.

2.2.5 Flatten

The output of convolution layers may have a depth greater than one. Flatten concatenates the output from the convolution layers to form a flat structure which can then be fed as input to MLFF network (Figure 2.2a).

2.3 Multilayer Feed Forward Network

MLFF or fully connected layer is a structure where neurons from a layer is connected to all the neurons in the next layer [26]. MLFF accepts flattened output from convolution

layers and maps it to the output as shown in Figure 2.2b.

2.4 Output Layer - One Hot Encoding

The main focus of this thesis is on classification problems. There are multiple ways to express the desired output. Most used format to represent a classification output is *One-hot encoding* format. For a N-class problem, the output is a vector of dimension N. Each element of this output vector can only attain a value of either 0 (ON) or 1 (OFF). And only one element of the vector can be "ON" at a time. For example, consider a two class problem. The two classes are represented by

Class 1: [1 0]

Class 2: [0 1]

2.5 Activation Functions

Activation functions can be either linear or non-linear. If the inner product of the input to a neuron and it's weight set is denoted by *net* then output of the neuron is a some function of *net* (denoted by y). Non-linear activations enable the network to learn complex mappings and there exists multiple non-linear functions to choose from and hence a decision has to be made while designing an ANN or in this case a CNN.

$$net = \underline{w}^T \underline{x} \tag{2.11}$$

$$y = f(net) \tag{2.12}$$

A popular activation function is ReLU (Rectified Linear Unit) and it's input-output relation is shown in Figure 2.3. The activation function for all neurons except the last layer

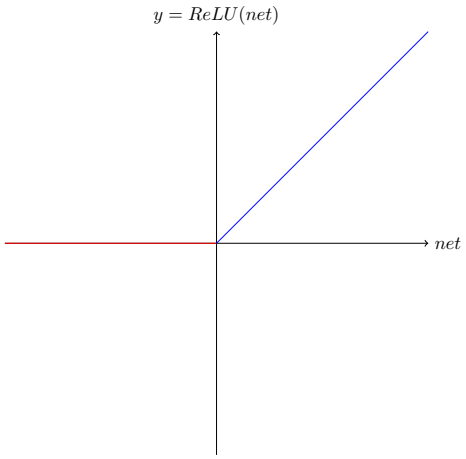


Figure 2.3: Plot of ReLU activation function

will be assumed to be ReLU through out the document unless otherwise specified.

$$y = max(0, net) \tag{2.13}$$

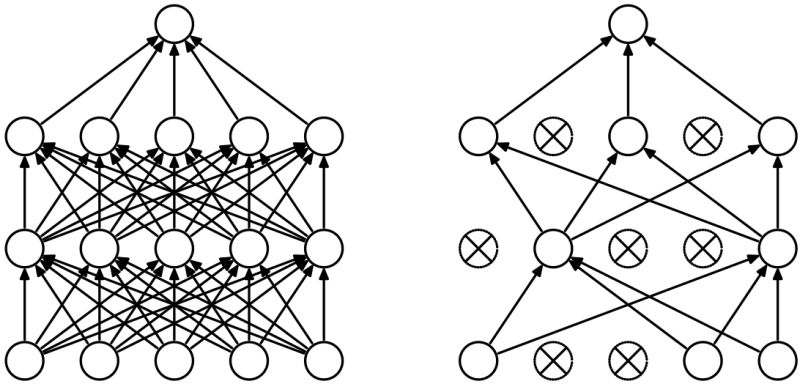
Before ReLU became famous, sigmoid activation $\sigma(net)$ was widely popular. But ReLU is found to converge faster than sigmoid activations and ReLU units have been mathematically proven to be the average of cascaded sigmoids [21]. Sigmoid activation is given by the following equation.

$$\sigma(net) = \frac{1}{1 + e^{-net}} \tag{2.14}$$

For the output layer, the choice of activation function depends on the type of output. For classification problems, SoftMax activations are preferred and for predictive/regression problems, ReLU is preferred.

SoftMax function for n class problem (representing n probabilities of input belonging to each of n classes) is

$$P(class(\underline{i}) = j|\underline{net}) = \frac{e^{net_j}}{\sum_{k=1}^{n} e^{net_k}} \tag{2.15}$$



(a) Standard Neural Net (b) After applying dropout.

Figure 2.4: Dropout

2.6 Dropouts

It is very difficult to find a CNN model without dropout regularization which hints at the popularity of this powerful yet simple regularization algorithm. Dropouts are used to prevent overfitting which can be loosely stated as the phenomenon of memorizing the inputs in contrast to learning general traits of the inputs.

Dropout refers to dropping the output of a neuron which implies zero input to the next layer. In a layer, there can be several neurons and whether a neuron drops the output or not is decided by the dropout rate P(drop). If P(drop) = 0.8, each output randomly choses a number from 0 to 1. If the chosen number is lesser than 0.8, that output will be dropped. Pictorial representation of this concept is provided by the Figure 2.4.

It may not be obvious as to why dropout works but a rather simple explanation awaits in Chapter 5.

2.7 Back Propagation

Until now, only forward propagation has been discussed. In the following subsections, back propagation of both MLFF and convolution layers will be discussed.

2.7.1 Loss Function

Loss function represents the deviation of the predicted output, $\underline{o}$, from the target output, $\underline{t}$. Some of the simplest loss functions are TSS (total sum of squares) and it's cousins MSE (Mean Squared Error) and RMSE (Root MSE). TSS is expressed as follows.

$$\begin{aligned} L &= \frac{1}{2}(\underline{t} - \underline{o})^T(\underline{t} - \underline{o}) \\ &= \frac{1}{2}\sum(t - o)^2 \end{aligned} \tag{2.16}$$

MSE and RMSE are modification of Equation 2.16 and these loss functions are mostly used for regression problems. For classification problems, categorical cross entropies (also called Maximum Likelihood Estimation) are used.

$$L = -\frac{1}{N}\sum_{n=1}^{N}[y_n log(\hat{y}_n) + (1 - y_n) log(1 - \hat{y}_n)] \tag{2.17}$$

Our objective is to minimize loss function, Gradient descent optimization can be used to reduce the loss function.

2.7.2 Gradient Descent (GDR) Optimization

The basic idea is that loss functions are generally convex functions and if weights are updated in the opposite direction of the gradients, hopefully weights reach *global* minima. In order to update the weights, the gradient of loss with respect to the output needs to be back propagated. The parameter which back propagates this loss information from the output layer j to the previous layer i is called *sensitivity* δ_j. Updating the weights differs slightly for MLFF layer and convolution layers and these are discussed in following sections.

2.7.2.1 MLFF weight update

Let the layer j be fully connected with layer i. If $f(net)$ represents activation function then the weights update for each neuron j is given by

$$\Delta w_{ji} = \epsilon \delta_j o_i \tag{2.18}$$

where

o_i : output from previous layer i

$$\delta_j = (t_j - o_j) f_j^{'}(net_j); \text{ if output unit} \tag{2.19}$$

$$\delta_j = f_j^{'}(net_j) \sum_n \delta_n w_{nj}; \text{ if hidden units} \tag{2.20}$$

2.7.3 Convolution

The weight update for convolution layers are slightly different and it is better explored by taking an example. Let's go back to forward propagation of convolution layers and consider an example for non-causal convolution (Equation 2.1) with n = 5, k = 3 and s = 1. Then convolution outputs are as follows.

$$y(0) = x(0)h(0) + x(1)h(1) + x(2)h(2)$$

$$y(1) = x(1)h(0) + x(2)h(1) + x(3)h(2)$$

$$y(2) = x(2)h(0) + x(3)h(1) + x(4)h(2)$$

Let's assume that sensitivity δ_j is already found out by back propagating gradients through MLFF. The difference between a standard neuron weight and a convolution neuron weight is that standard neuron weight affects output of just one neuron whereas a convolution weight affects multiple outputs. In this example, $h(0)$ affects $y(0)$, $y(1)$ and $y(2)$. Hence the weights are updated by summing up products of sensitivity from the output and the input. It is mathematically expressed as follows.

$$\Delta h(j) = \epsilon \sum_{n} \delta_j(n) x(n) \tag{2.21}$$

$$\Delta h(j) = \epsilon(\delta(n) \circledast x(n)) \tag{2.22}$$

Instead of constant learning rate ϵ, one can apply what is know as adaptive learning rate wherein learning rate decays with iterations. There are better optimizers such as RMSprop[20], momentum[29], adam[13] etc which normally tend to converge faster than GDR. In this thesis, adam optimizer is used although the output of the experiments are quite independent of the choice of optimizer.

Chapter 3

Time Domain Analysis

In this chapter, the weights learned by different CNN models for carefully crafted datasets will be explored. The weights learned by the convolution layer are especially interesting and some weight patterns can be explained just be looking at it. And some are very complex and requires frequency domain analysis which are explored in the next chapter. The dataset of signals used in this chapter includes sinusoids, triangle and rectangular waveforms.

3.1 Classification of Sinusoids

3.1.1 Problem Description

One of the basic classes of time varying signals are sinusoids - all possible phase shifted sine waveforms. The first experiment is to distinguish between two sinusoids of different frequencies, 5 Hz and 10 Hz. The sampling frequency is 128 Hz and time period of the observation is 1 second. Hence, there are 128 samples as shown in Figure 3.1.

3.1.2 CNN Model

For this experiment and for this entire chapter, a very simple CNN model is considered. It will have one convolution layer with two kernels of length 32. The convolution

Dataset Sampling Frequency : 128 Hz		
Classes	**Type**	**Parameter**
Class 1	Sinusoids	Frequency 5 Hz
Class 2	Sinusoids	Frequency 10 Hz
Description	Both class includes 200 randomly phase shifted version of sine waves. The first class is hot encoded and is represented by [1 0]. Similarly the second class is represented by [0 1].	

Table 3.1: Dataset 1

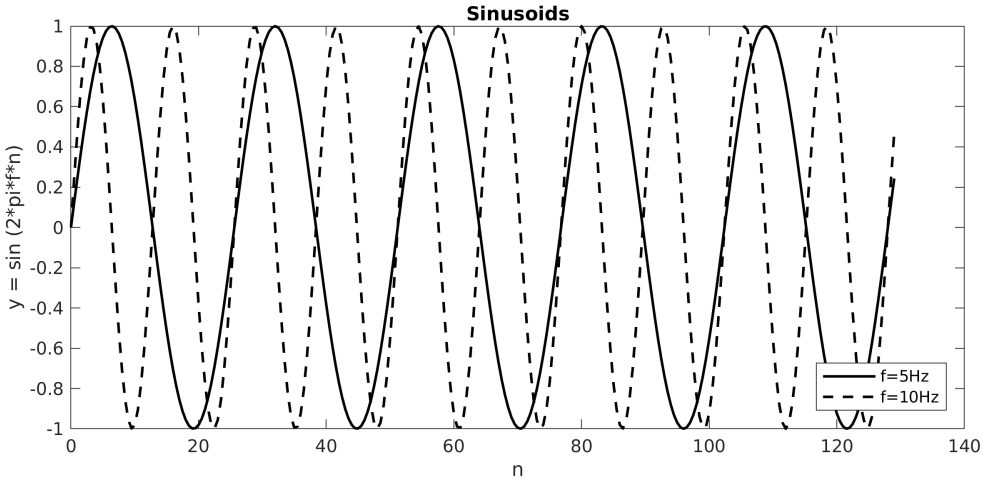


Figure 3.1: Two classes - sinusoids of 5 and 10Hz

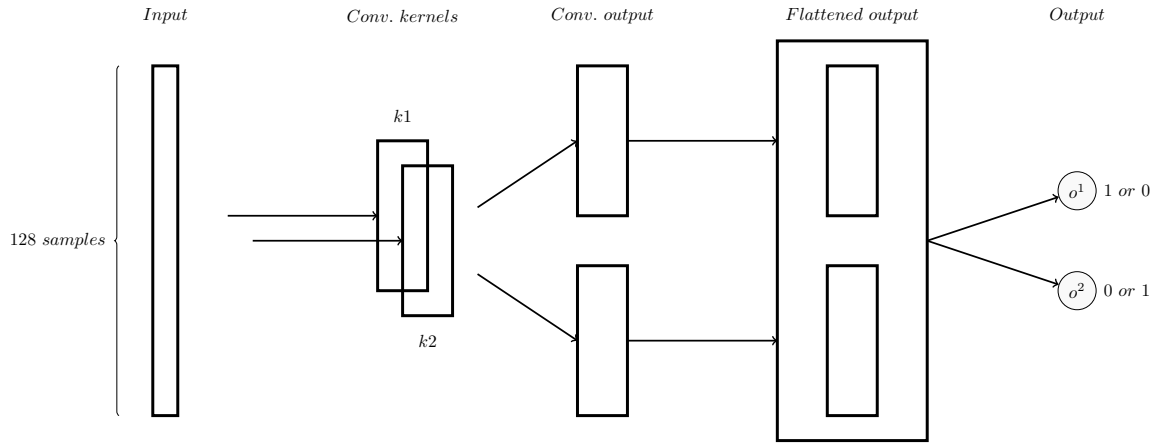


Figure 3.2: CNN Model 1

CNN Model 1	
Layers	**Settings**
Input	128 samples
Conv. Layers 2	2 Kernels of length 32
Fully connected	-
Output	Categorical output 2 categories

Table 3.2: CNN Model 1

layer is followed by two neurons representing categorical output - [1 0] for class 1 and [0 1] for class 2. A pictorial representation of the CNN model is shown in the Figure 3.2.

3.1.3 Results

The CNN model is trained for 100 epochs and the model is validated using 10-fold validation. Minimum, average and maximum accuracies are shown below.

The plot of loss vs epoch is shown in Figure 3.3. The weights obtained from the two kernels after training are shown in Figures 3.4 and 3.5.

10 Fold Validation		
Min. Accuracy	**Avg. Accuracy**	**Max. Accuracy**
100	100	100

Table 3.3: CNN model 1 validation

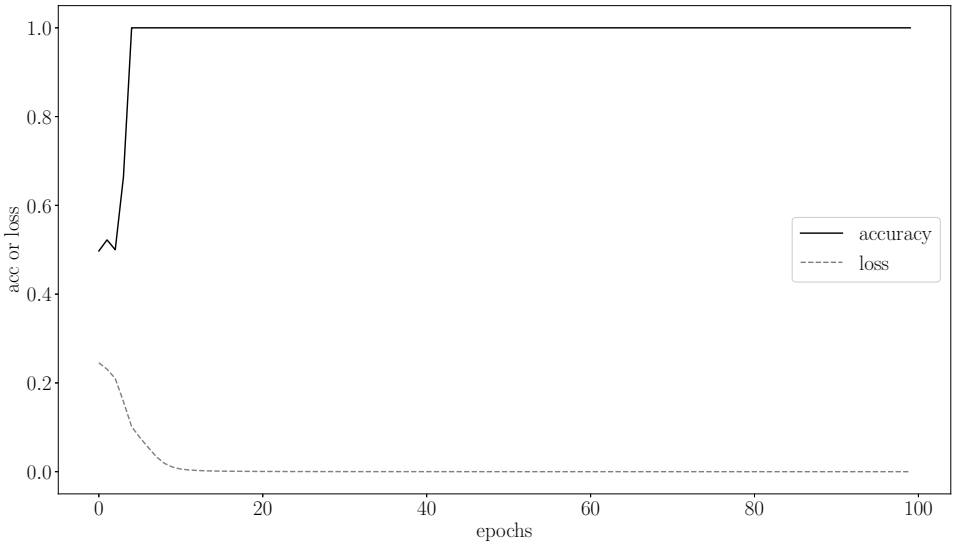


Figure 3.3: Loss plot vs epoch

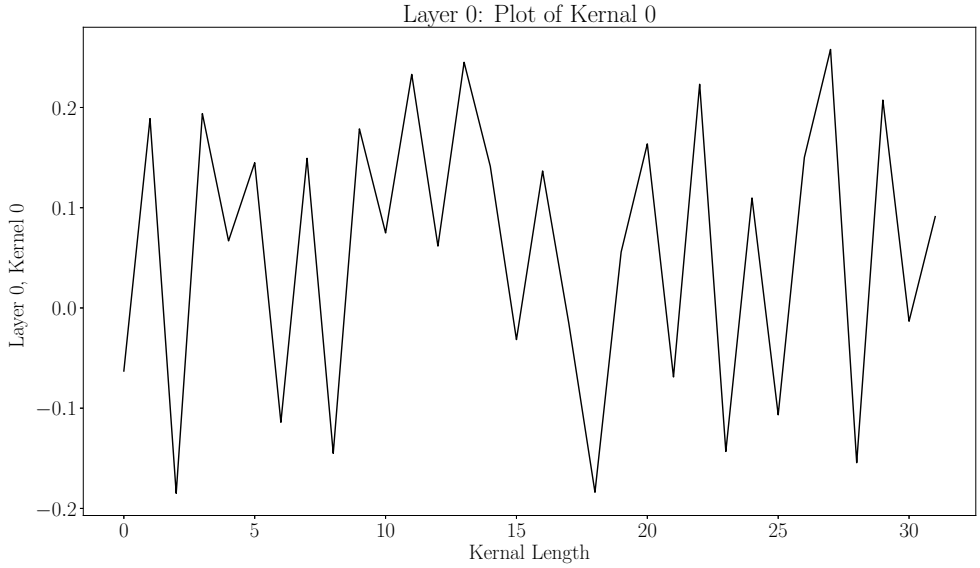


Figure 3.4: Kernel 1

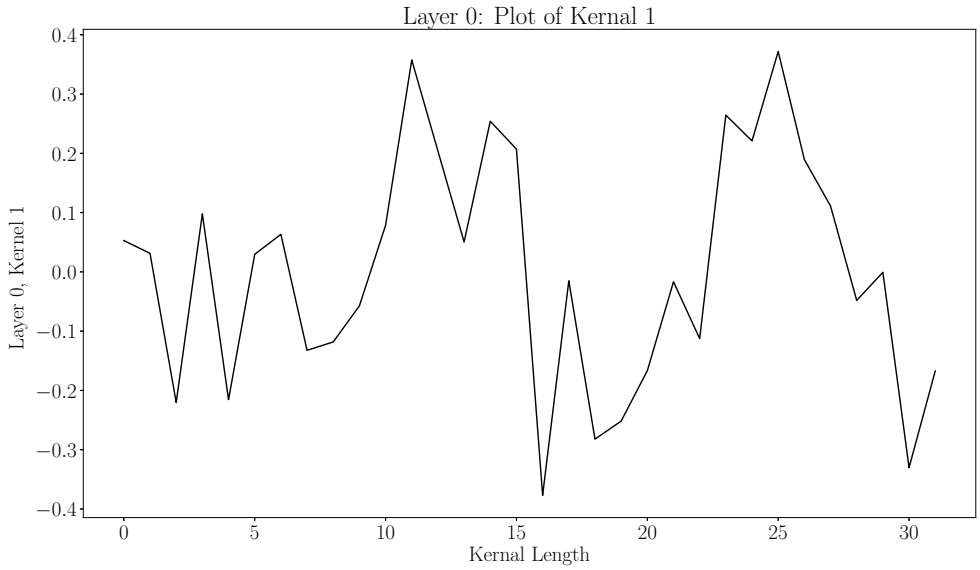


Figure 3.5: Kernel 2

3.1.4 Explanation

Looking at plots 3.4 and 3.5 doesn't reveal any obvious insights. But a surprisingly simple explanation lies ahead in Chapter 5. This simple experiment was set to show how difficult it is to analyze weights without proper tools. But few interesting observations can be made when a CNN is trained with dataset involving different slopes.

3.2 A More Challenging Dataset

3.2.1 Problem Description

The basic idea is to generate overlapping triangle and rectangular signals. There are two ways to over lap these two signals as shown in Figures 3.6a and 3.6b. To make things interesting, the overlapped triangle-rectangle blocks are placed randomly anywhere in 128 sample window as shown in Figures 3.7a and 3.7b. The remainder of the input samples contains uniformly distributed noise with values between 0 and 1. CNN is trained to classify these two types of signals. The weights learned by the convolutional kernel forms

Dataset Sampling Frequency : 128 Hz		
Classes	**Type**	**Parameter**
Class 1	Triangle-Rectangle overlap block	40 % triangle and 60 % rectangle signal
Class 2	Rectangle-Triangle overlap block	40 % triangle and 60 % rectangle signal
Description	The overlap block length is 32 and is embedded somewhere in between random signals within the 128 sample window. Both classes have 1000 such samples.	

Table 3.4: Dataset

10 Fold Validation		
Min. Accuracy	**Avg. Accuracy**	**Max. Accuracy**
74.5	90.7	99

Table 3.5: Experiment 2 validation

an interesting and known pattern which is discussed in subsection 3.2.3.

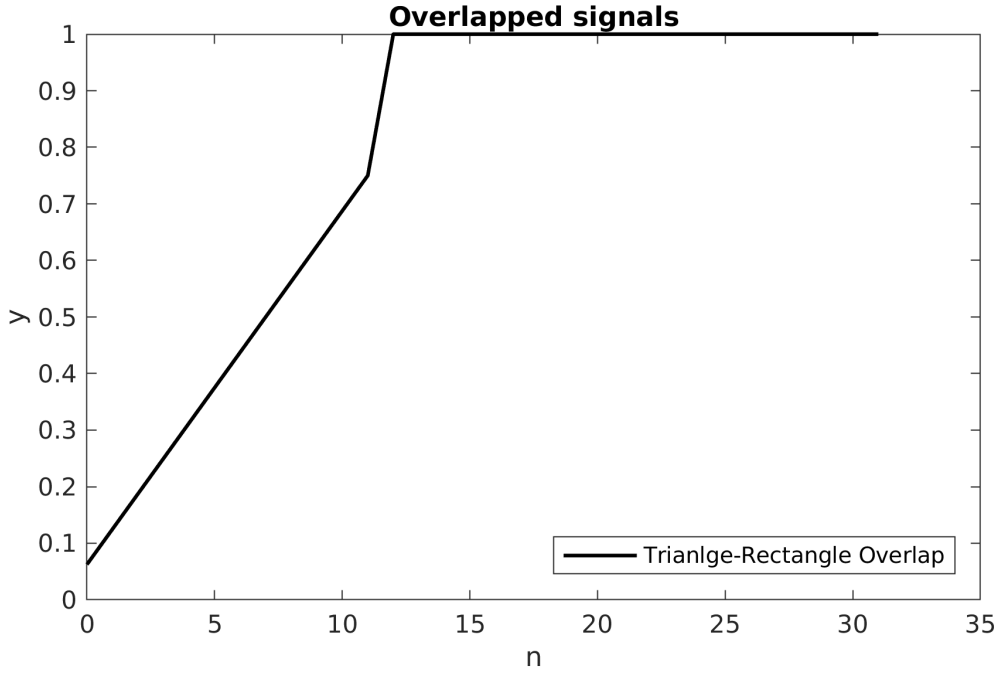
3.2.2 Results

The 10 fold validation result is shown in Table 3.5. The kernel weights learned by CNN when the accuracy was 99 % is shown by Figures 3.9a and 3.9b.

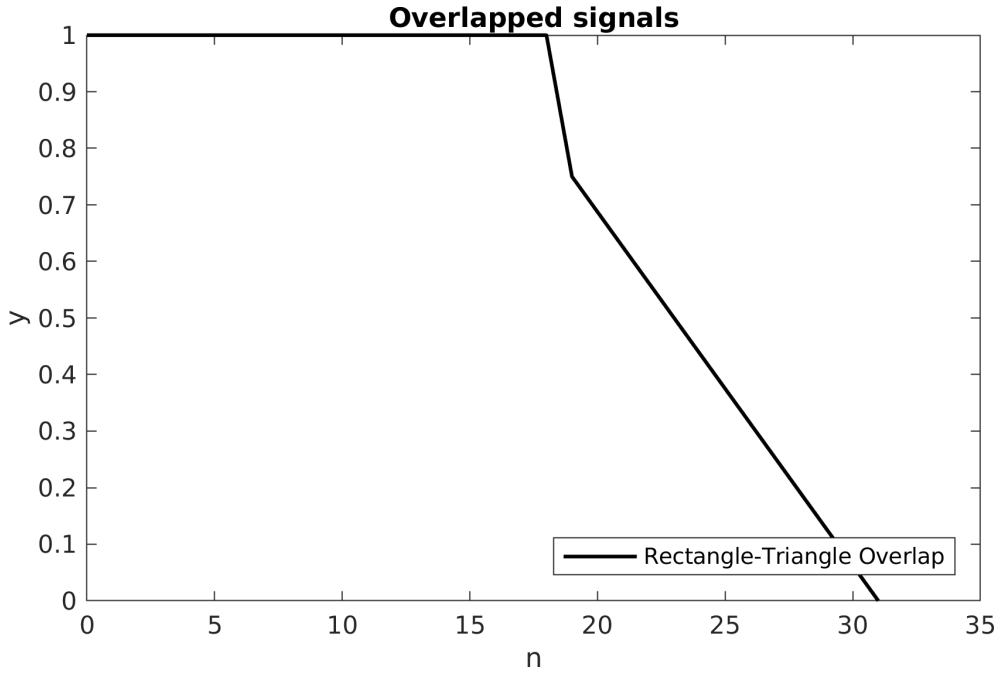
3.2.3 Explanation

Let's see how we can solve this problem manually. To distinguish between Class 1 and Class 2, smoothing is necessary to remove random noise and then differentiation of the signal yields zero for rectangle and differentiation of triangle will yield positive or negative slope. In this case, Class 1 has positive ramp and Class 2 has negative ramp due to overlapping and differentiation yields positive and negative values respectively. This would have helped to distinguish between the two classes. Second derivative can be substituted in place of first derivative.

To perform second order differentiation via convolution, a well known method is to convolve the input signal with second order Gaussian filter. This algorithm especially used in the field of digital image processing to perform edge detection on an image. Expression

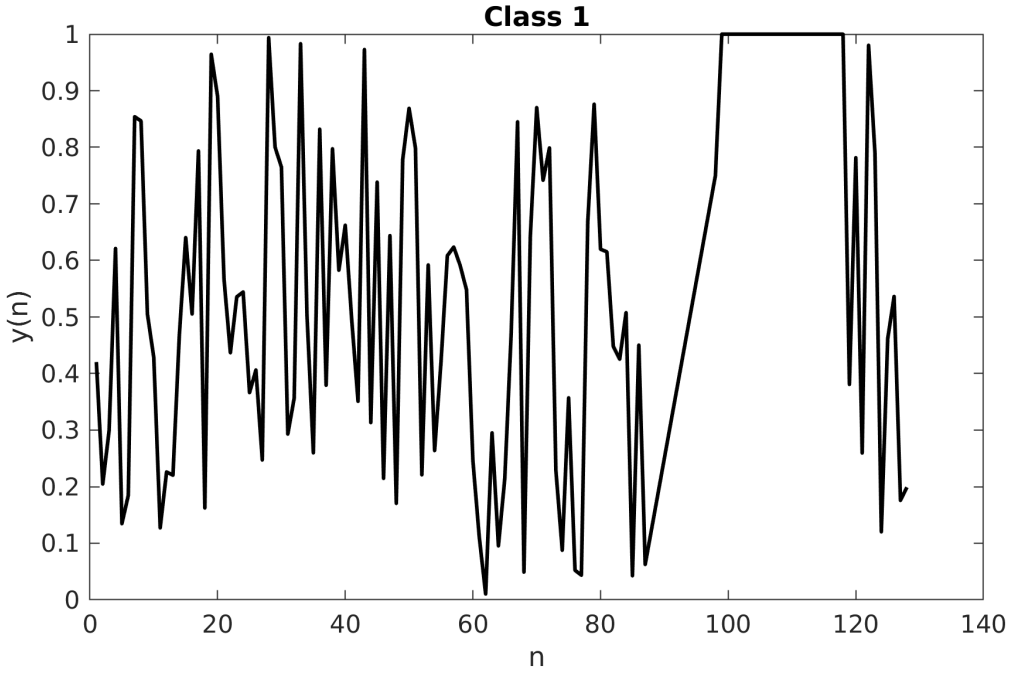


(a) Triangle-Rectangle Overlap

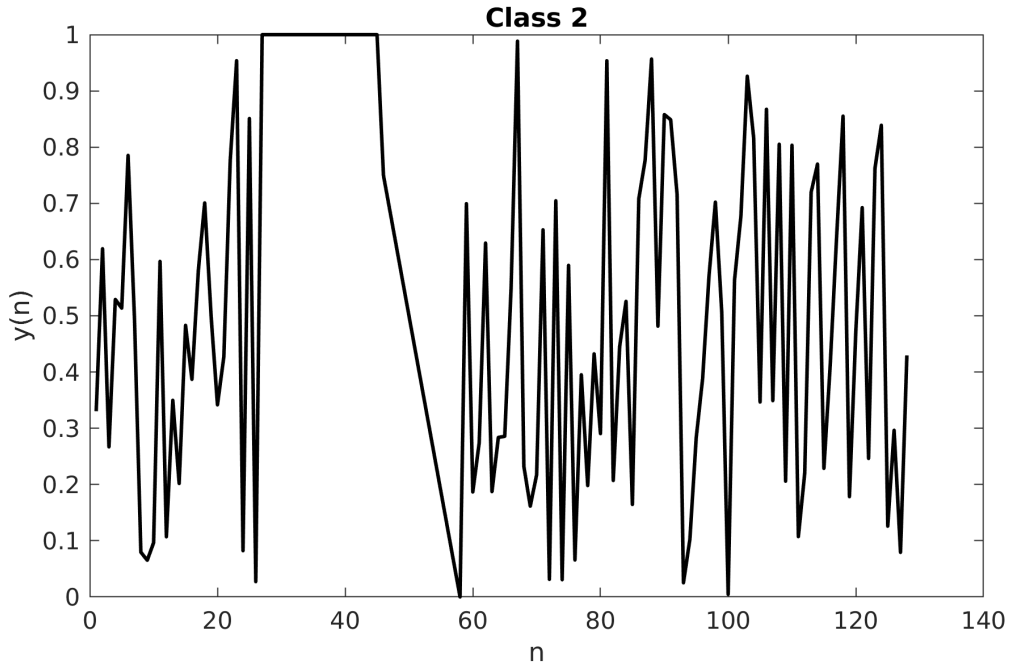


(b) Rectangle Triangle Overlap

Figure 3.6: Two ways of overlapping



(a) Class 1



(b) Class 2

Figure 3.7: Dataset 2 - Overlapped triangle and rectangular waveforms

for Gaussian distribution with mean 0 and variance σ^2 is as follows.

$$g(x) = \frac{1}{\sqrt{2\pi}\sigma}\ e^{-\frac{x^2}{2\sigma^2}} \tag{3.1}$$

First order differentiation of $g(x)$ is given by

$$g^{'}(x) = \frac{-x}{\sqrt{2\pi}\sigma^3}\ e^{-\frac{x^2}{2\sigma^2}} \tag{3.2}$$

Second order differentiation (inverse mexican hat) of $g(x)$ is given by

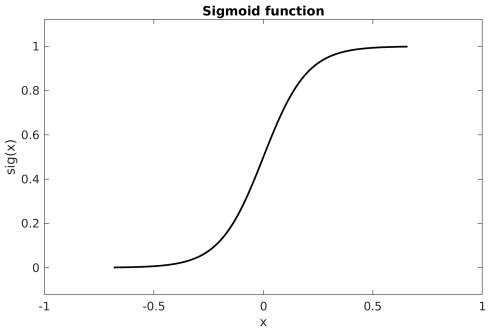
$$g^{''}(x) = \frac{(x^2-\sigma^2)}{\sqrt{2\pi}\sigma^5}\ e^{-\frac{x^2}{2\sigma^2}} \tag{3.3}$$

Figure 3.10 compares the weights learned by CNN to scaled second order Gaussian filter weights ($mean \approx 23$ and $variance \approx 6.5$). **Those two weights are pretty close and whenever the accuracy is high ($>= 90\%$), learned kernel weights always looks like a second order Gaussian filter.**

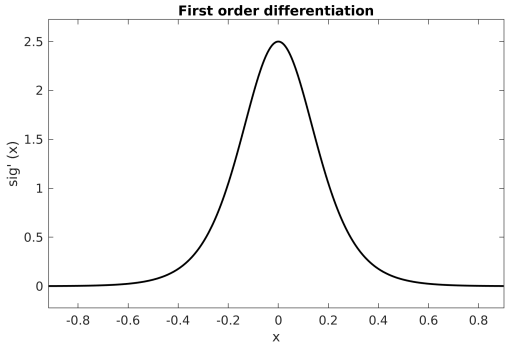
The bounded ramps due to overlapping between triangle and rectangle can be modeled by a sigmoid function as show below along with its first and second derivative shown in the Figure 3.8. Now compare second derivative shown in Figure 3.8c with Box A of Figure 3.8d. Both looks pretty much the same (ignoring the scales) which suggests second order differentiation via convolution.

To further bolster this point, the actual convolution output is compared with convolution output obtained from convolving input with second order Gaussian (simulated output). The simulated output follows the actual output pretty closely and thus confirming that CNN is trying to mimic a Gaussian filter.

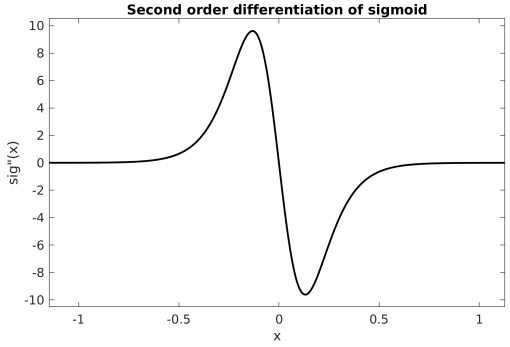
To conclude, the process adopted by CNN is very similar to the process described above if we were to design a classification algorithm using conventional signal processing techniques.



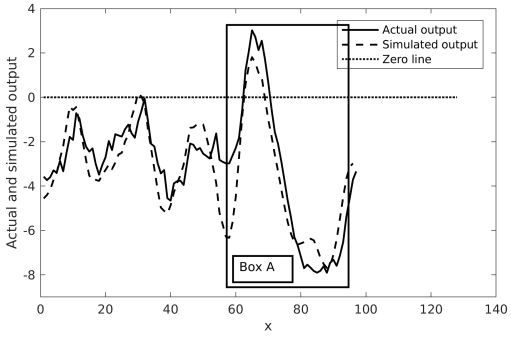
(a) Sigmoid



(b) First derivative of sigmoid

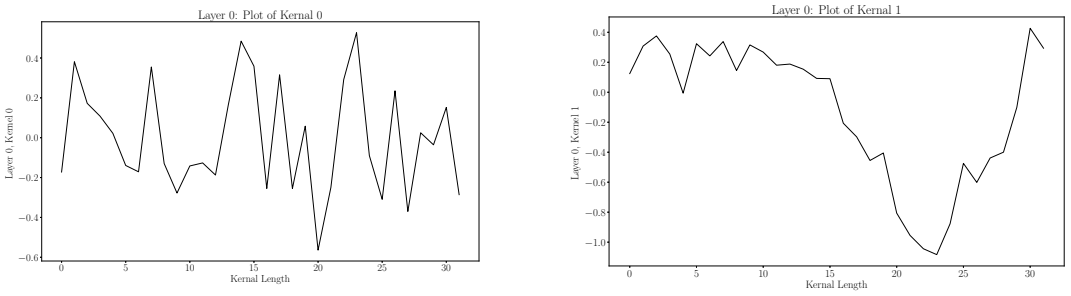


(c) Second derivative of sigmoid



(d) Actual output

Figure 3.8: Differentiation of sigmoids



(a) Kernel 1

(b) Kernel 2

Figure 3.9: Learned Kernels

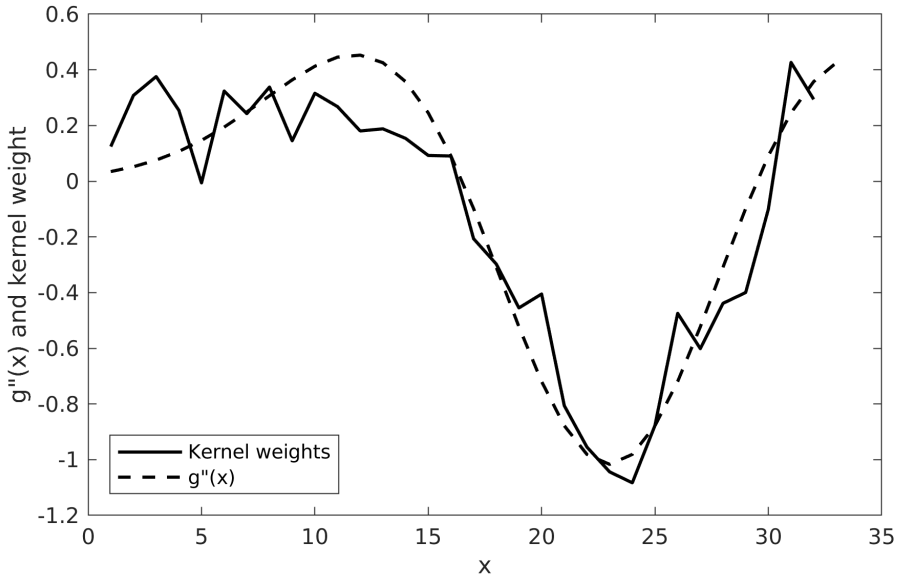
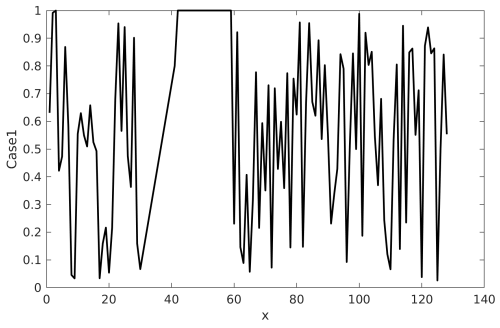
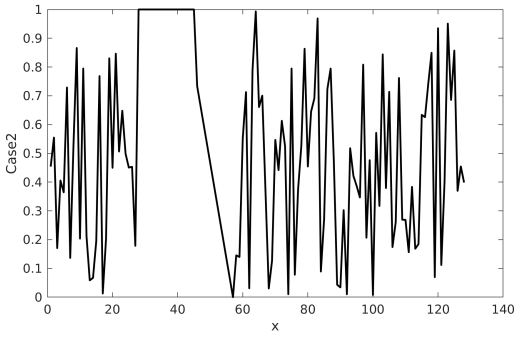


Figure 3.10: Comparing learned kernel weights and artificially generated Gaussian filter weights

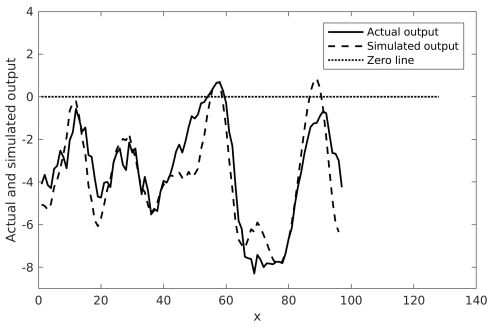


(a) Example 1

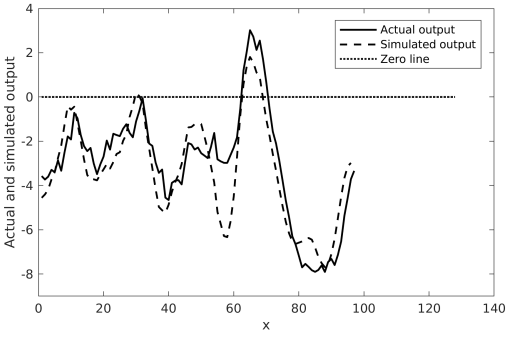


(b) Example 2

Figure 3.11: Example inputs



(a) Example 1 convolution output



(b) Example 2 convolution output

Figure 3.12: Example outputs of convolution

Dataset Sampling Frequency : 128 Hz		
Classes	**Type**	**Parameter**
Class 1	Triangle block	Width: 40
Class 2	Triangle block	Width: 20
Description	The triangle block lengths are 40 and 20 for class 1 and class 2 respectively. Triangle block is embedded somewhere in between random signals within the 128 sample window. Both classes have 300 such waveforms.	

Table 3.6: Dataset

10 Fold Validation		
Min. Accuracy	**Avg. Accuracy**	**Max. Accuracy**
90	96	100

Table 3.7: Experiment 3 validation

3.3 Classification of Triangular Waveforms

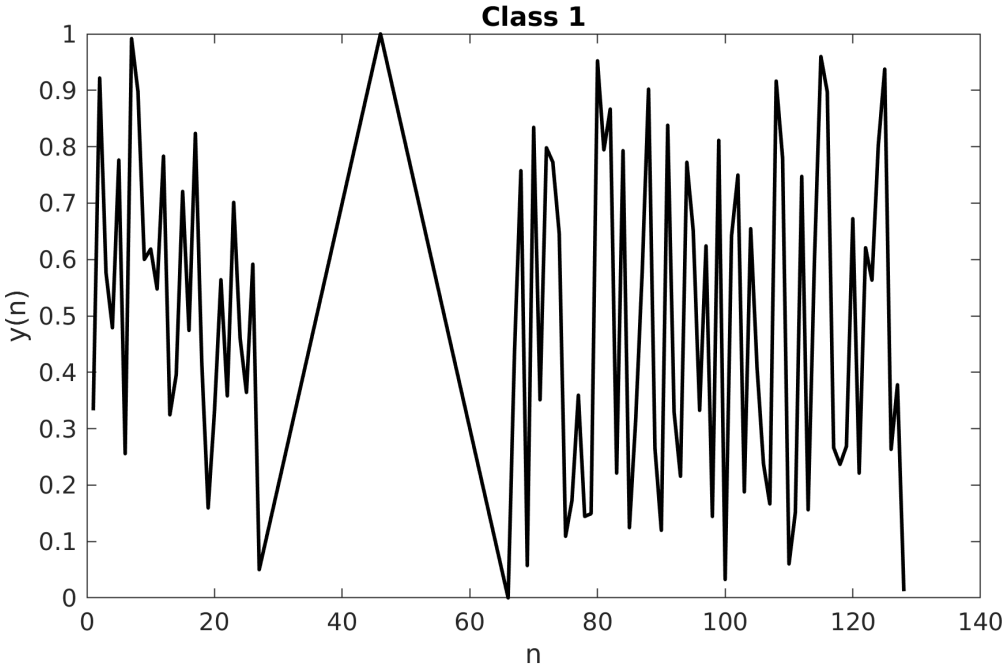
Continuing with the theme of slope detection, let's explore classification of triangular waveforms. To classify triangles of different widths, it is important to figure out slope of the triangle waveform. This experiment can also be viewed as a test of the explanation provided in the previous section.

3.3.1 Problem Description

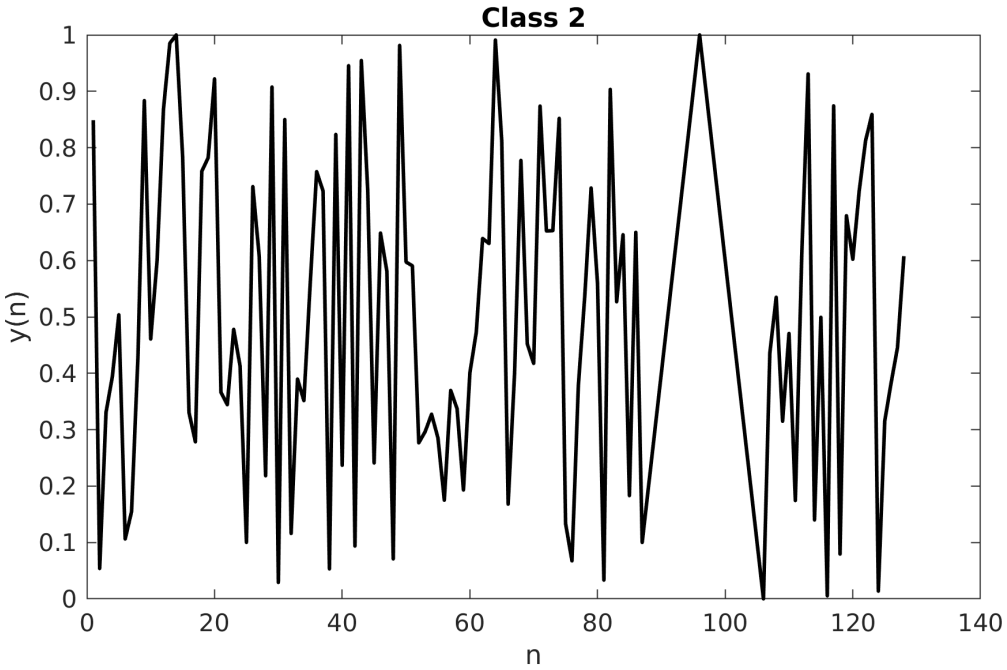
Once again a triangle will be placed in between random noise somewhere within the 128 sample window. The first class will contain a triangle of width 40 samples and class two signals will contain a triangle of width 20 as shown in Figure 3.13.

3.3.2 Result

After training, the 10 fold validation of the model for this problem is as shown in Table 3.7. Weights learned for two separate runs for kernel lengths 32 and 50 are shown in Figures 3.14 and 3.15.

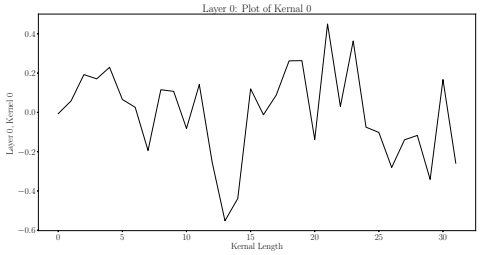


(a) Class 1

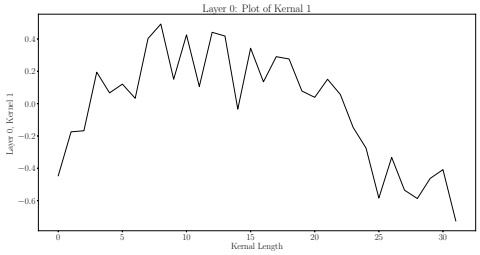


(b) Class 2

Figure 3.13: Dataset



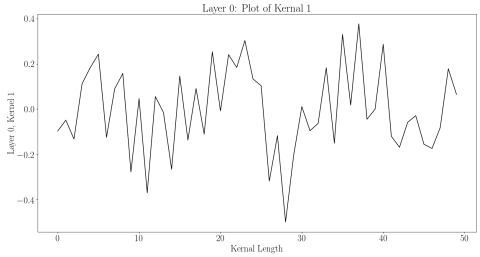
(a) Kernel 1



(b) Kernel 2

Figure 3.14: First run

(a) Kernel 1



(b) Kernel 2

Figure 3.15: Second run

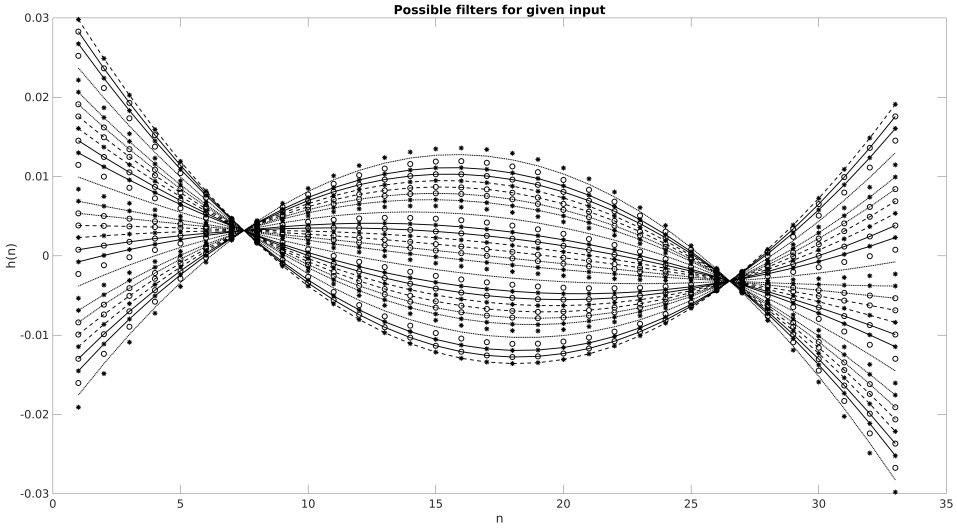


Figure 3.16: Possible filter coefficients to achieve second order differentiation via convolution.

3.3.3 Explanation

Figures 3.14a and 3.15b is kind of similar to second order Gaussian seen in the previous section. Clearly, plots 3.14b and 3.15a don't look anything close to a Gaussian filter. Is the theory falling apart? Fortunately, no. To achieve 2nd order differentiation, Gaussian filters are not the only option. There exists a family of filters which can do second order differentiation and these filters can be extracted using Savitzky-Golay algorithm.

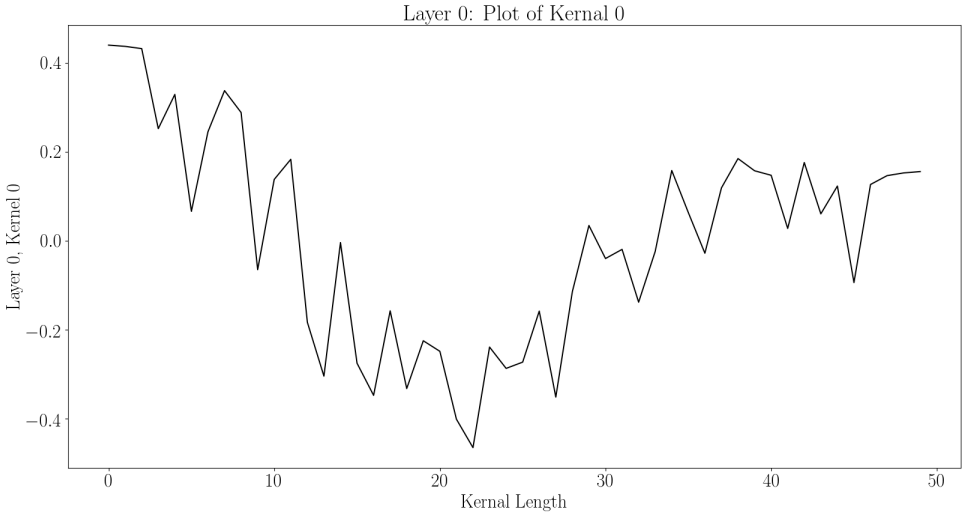
Savitzky-Golay Algorithm The primary aim of this algorithm is to smoothen the input data in other words to remove noise. This is done by convolving a smoothing window over input signal. The coefficients of the smoothing window is found by fitting adjacent data points with low-degree polynomial using linear least squares (via a pseudo inverse) [25]. If you arrange the co-efficients in certain order, you can get first derivative, second derivative etc. of the input signal. A matlab implementation of this algorithm gives out all possible filters which can perform second order differentiation and the filters are shown in Figure 3.16.

Now let's compare filters from Sav-Gol algorithm with CNN learned filters (Figure 3.18). Plots 3.17b and 3.17a are quite similar in structure and the same observation can be made for Figures 3.18b and 3.18a. You may notice the difference in scale (y-axis). Because convolution is a linear operation, the scale difference just results in equally scaled outputs and hence the filters obtained by Sav-gol and learned kernels are similar[1].

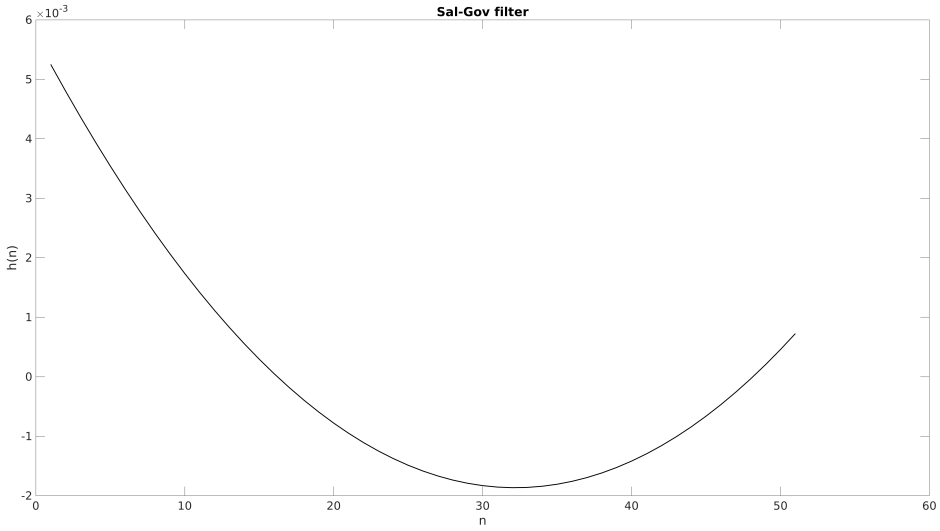
3.4 Weights Learned by Fully Connected Network

For CNN Model 1, there were two convolution kernels in the first layer and all the problems that we've dealt so far was two class problems. Ideally, each layer has to learn something about one of the two classes. Output from the flatten layer is fed to the output neurons. There are two neurons for each case. If the first neuron is ON (> 0.5) then the output belongs to class 1 and vice versa. Intuitively, the output weights of first neuron should be of opposite polarity to output weights of second neuron. And this theory holds true and weights of last layer is depicted in the Figure 3.19. **Clearly, the weights of output neuron 0 is a mirror image of weights of output neuron 1 where mirror is placed at the zero crossing.**

[1]Sav-gol algorithm only takes in odd filter length so the Sal-gov filter lengths in this examples are 33 instead of 32 and 51 instead of 50.

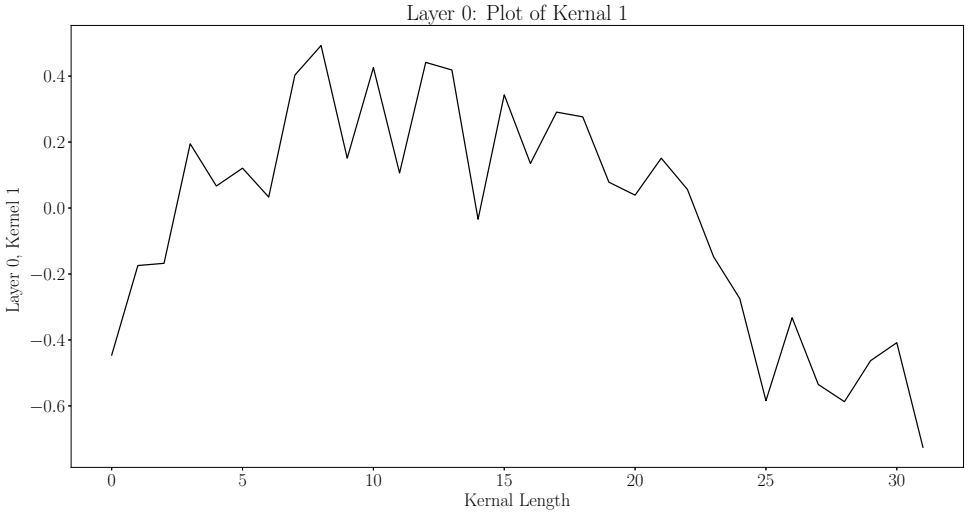


(a) Original Kernel

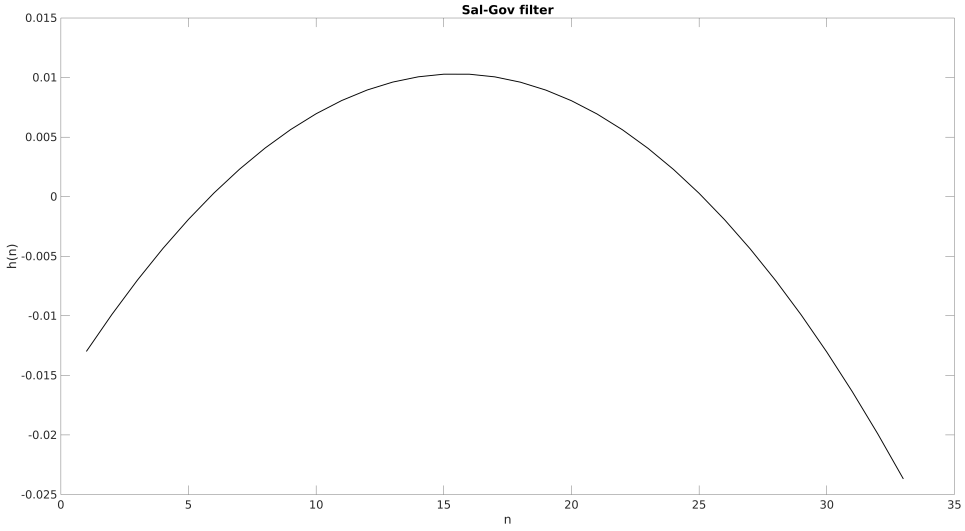


(b) Sav-gol filter

Figure 3.17: Second run - Comparing Sal-gov filter output with original kernel



(a) Original Kernel



(b) Sav-gol filter

Figure 3.18: Second run

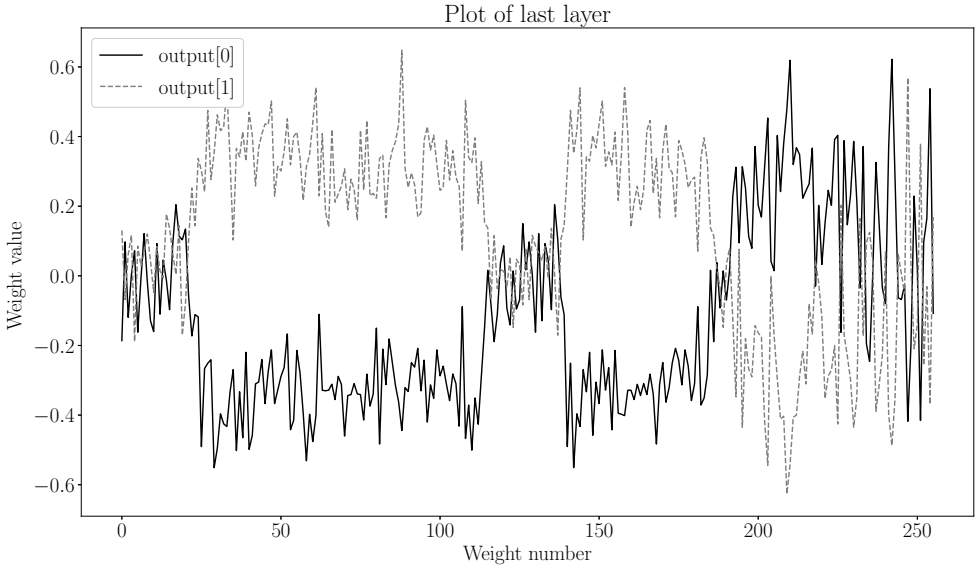


Figure 3.19: Last layer weights

Chapter 4

Frequency Domain Tools

In this chapter, frequency domain tools are explored which will be extensively used in the next chapter to understand the weights learned by the convolution layers. To convert a discrete signal from time domain to frequency domain, we use DFT (Discrete Fourier Transform) which is explained in the section 4.1.

4.1 Discrete Fourier Transform

DFT is a mathematical procedure used to determine harmonic or frequency content of a discrete signal sequence [18]. The core concept of DFT is that (loosely) any signal can be expressed as sum of sinusoids of different frequencies. These sinusoids form the basis vectors and hence DFT can be thought of as change of basis operation. DFT's origin is in continuous Fourier Transform (FT) and is defined as follows.

$$X(f) = \int_{-\infty}^{\infty} x(t) e^{-2\pi f t} dt \tag{4.1}$$

It's discrete counterpart is defined as

$$X(m) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi nm/N} \tag{4.2}$$

Using Euler expansion,

$$X(m) = \sum_{n=0}^{N-1} x(n)[cos(2\pi nm/N) - jsin(2\pi nm/N)] \tag{4.3}$$

where,

$$\begin{aligned} m &= \text{frequency domain indices,} \\ n &= \text{time domain indices - 0 ... N-1} \\ x(n) &= \text{sequence of input samples} \\ j &= \sqrt{-1} \\ N &= \text{number of samples in input sequence} \end{aligned}$$

The value of N is very important as you get back N points, hence the name N point DFT. If $N = 4$, its called 4 point DFT. The frequency represented by each N seperate points $(0, ..., N-1)$ are

$$f(m) = \frac{mf_s}{N} \tag{4.4}$$

where f_s is the sampling frequency of $x(n)$. For example, if $f_s = 128$ and $N = 128$ then each point obtained by DFT analysis represents frequency response for frequencies $f = 0, 1, 2, ...127$.

4.1.1 DFT Symmetry

If the input sequence is real (which is the case throughout this thesis), then X(m) is the complex conjugate of $X(N-m)$. That is,

$$X(m) = X^*(N-m) \tag{4.5}$$

This implies that magnitude of DFT is symmetric and phase of DFT response is anti-

symmetry at the point $(N/2) - 1$. Hence,

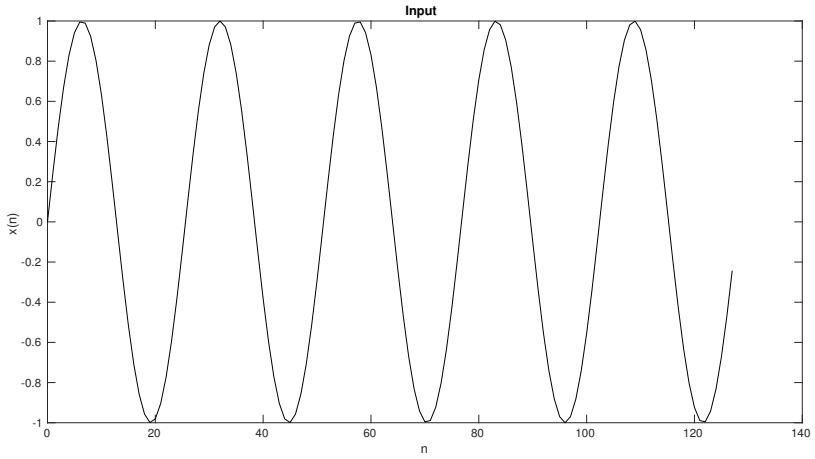
$$\begin{aligned} |X(0)| &= |X(N-1)| \\ |X(1)| &= |X(N-2)| \\ &\vdots \\ |X(N/2)-1)| &= |X(N/2)| \end{aligned}$$

and,

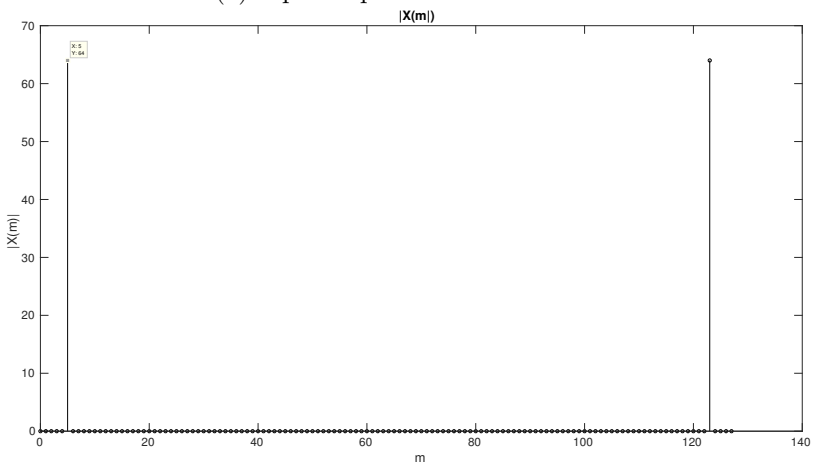
$$\begin{aligned} \underline{/X(0)} &= -\underline{/X(N-1)} \\ \underline{/X(1)} &= -\underline{/X(N-2)} \\ &\vdots \\ \underline{/X(N/2)-1)} &= -\underline{/X(N/2)} \end{aligned}$$

4.1.2 FFT

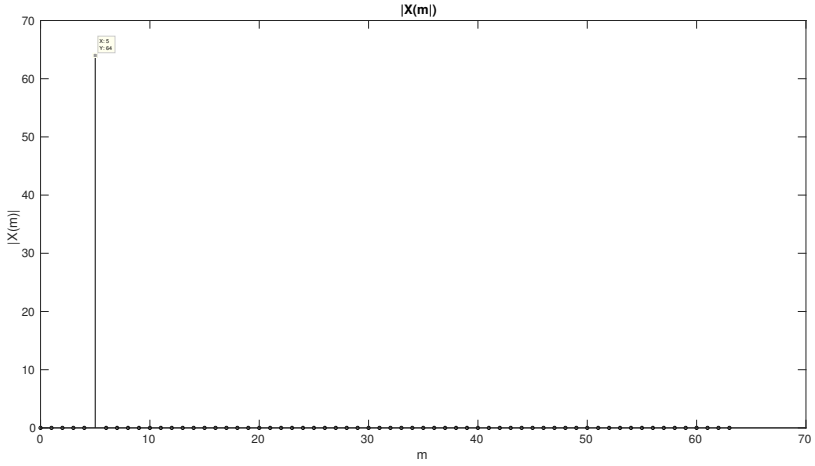
Fast Fourier Transform is an algorithm to compute DFT rapidly. Matlab and Python-scipy provide libraries to compute FFTs. A FFT example of a sinusoid is shown in the Figure 4.1. In this thesis, the emphasis is on magnitude response and as it is symmetric, studying frequency response until $f_s/2$ is sufficient. From here on, the frequency response will be similar to the plot shown in Figure 4.1c.



(a) Input sequence - 5Hz sinusoid



(b) Magnitude response



(c) Magnitude response without second half

Figure 4.1: Magnitude FFT response of the input sequence

4.2 DFT of Triangular and Rectangular Waveforms

The rectangular wave rect (t) with width T is shown in Figure 4.2a. It's DFT is as follows and is plotted in Figure 4.2b.

$$F\{rect(t)\} = AT\ sinc(fT) \tag{4.6}$$

The triangular wave tri (t) with width 2T is shown in Figure 4.3a. It's DFT is given by the equation 4.7 and is plotted in the Figure 4.3b.

$$F\{tri(t)\} = AT\ sinc^2(fT) \tag{4.7}$$

where

$$sinc(x) = \frac{sin(x)}{x} \tag{4.8}$$

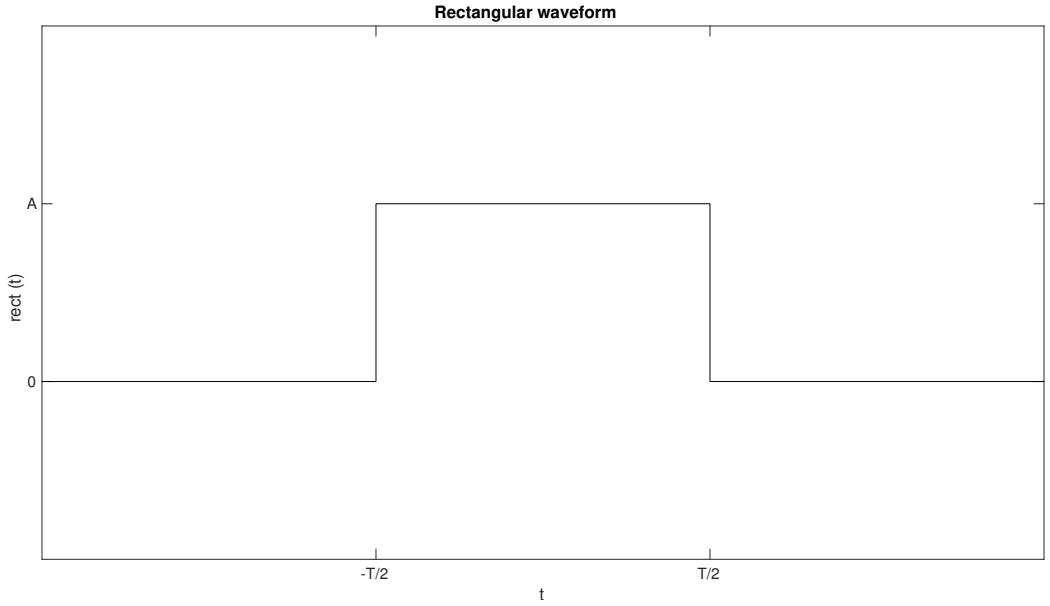
4.3 FIR Filters

Finite Impulse Response (FIR) filters (also known as non-recursive filters) are used for filtering the input signal. If the filter is represented by h(k), input is represented by x(n) then the output represented by y(n) is given by

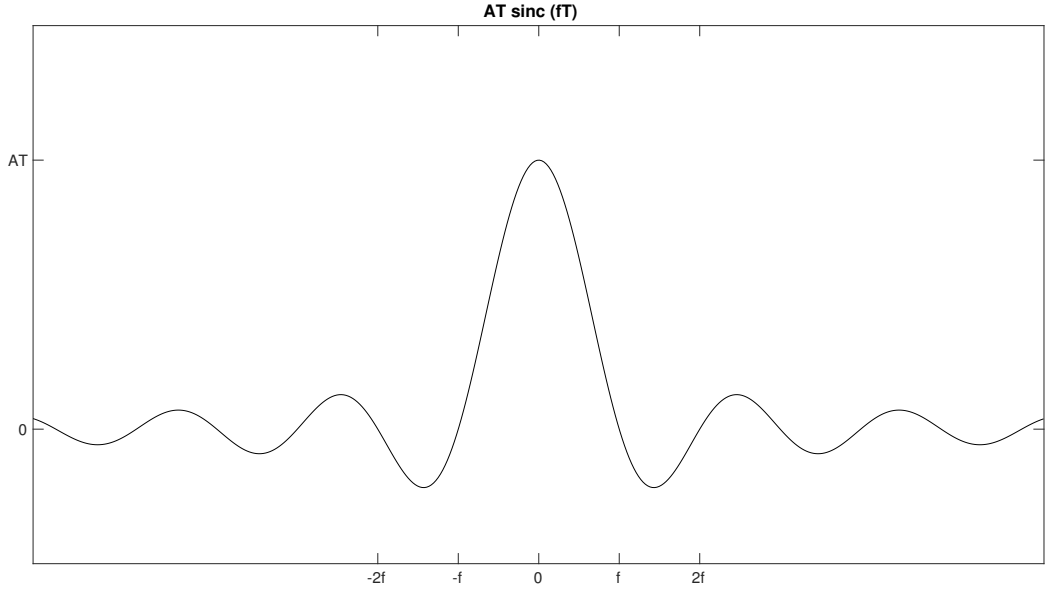
$$y(n) = h(k) \circledast x(n) \tag{4.9}$$

This is similar to causal filter discussed in the Chapter 2. We know that convolution in time domain corresponds to multiplication in frequency domain. Using this property, the filter weights can be found as follows (given the desired output frequency response $Y(f)$). This concept will be extensively used in coming chapters.

$$Y(f) = H(f)X(f) \tag{4.10}$$

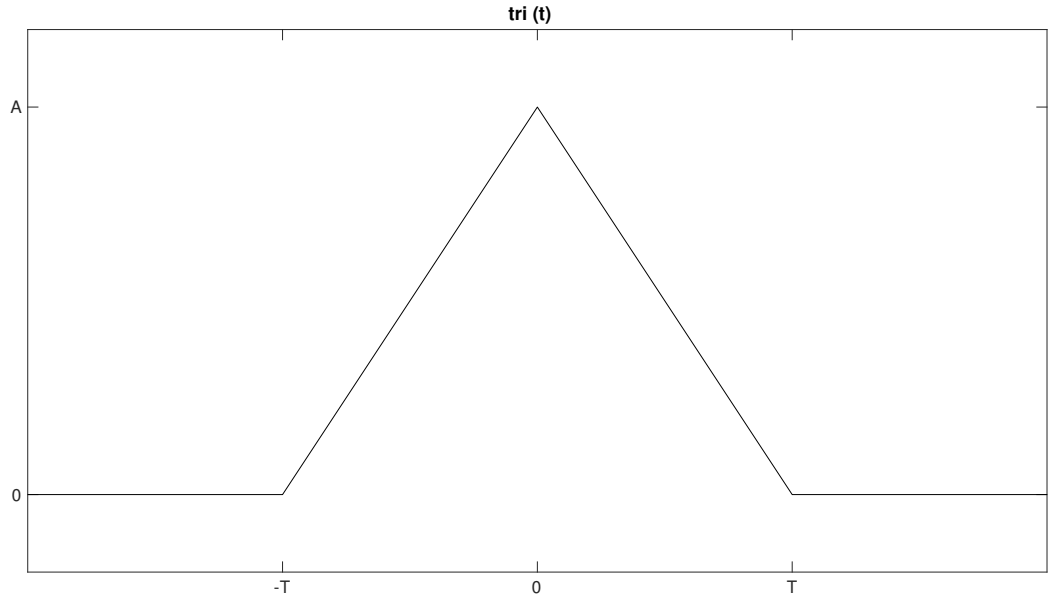


(a) $rect(t)$

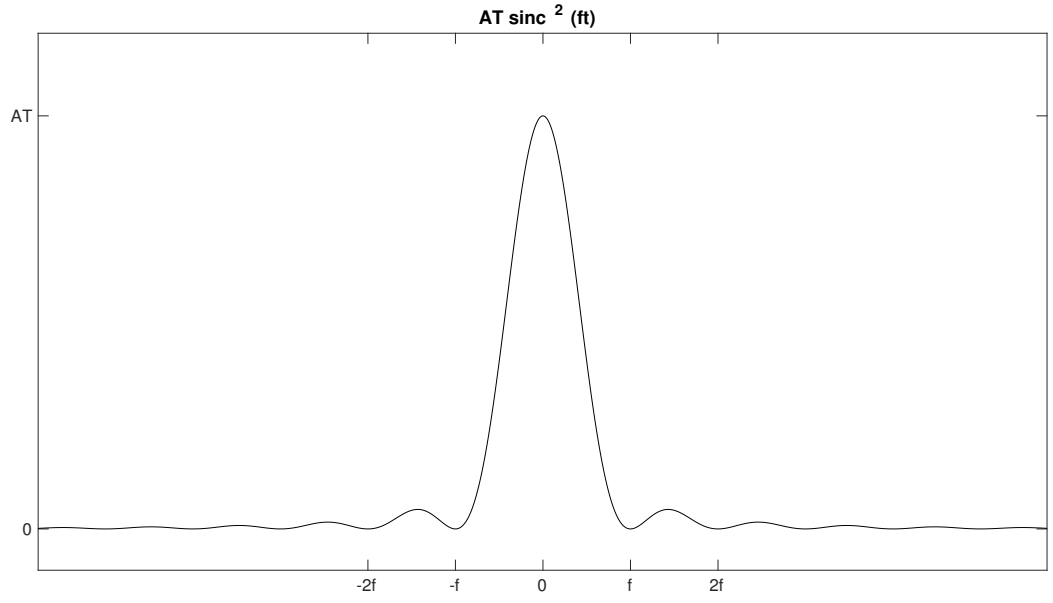


(b) DFT of $rect(t)$

Figure 4.2: Rectangular waveform and its DFT



(a) $tri(t)$



(b) DFT of $tri(t)$

Figure 4.3: Triangular waveform and its DFT

$$H(f) = \frac{Y(f)}{X(f)} \tag{4.11}$$

$$h(k) = F^{-1}\{H(f)\} \tag{4.12}$$

4.4 Analyzing FFT of a Filter

Unlike analyzing FFT of input or output sequence, analyzing the FFT of a filter or a kernel is slightly different. For input and output sequences, the FFT output represents frequency component for $f(m) = mf_s/N$ frequencies. For input sequence, the f_s is just sampling frequency of the input. The *same* sampling frequency has to be considered while analyzing the FFT output of a filter.

For example, let the input's f_s be 128 Hz and it is observed for 1 second resulting in an input of length 128 samples. Let's consider a filter of length 32 and the sampling frequency of the filter should be considered to be same as input sampling frequency i.e 128 Hz. Hence, 32 point FFT of the filter represents component for the $f(m) = 128m/32 = 4m$ frequencies. The output from a 32 point FFT represents

$$H(f) = \{H(0), H(4), H(8), ..., H(128)\} \tag{4.13}$$

As discussed earlier, the points $(N/2)$ to $(N-1)$ are redundant. Hence, the non-redundant frequencies stop at $m = (N/2) - 1 = 15$ or $f = 4 * 15 = 60$.

$$H(f) = \{H(0), H(4), ..., H(60)\} \tag{4.14}$$

In the next chapter, all the fundamental DSP tools discussed in this chapter will be used to analyze the weights learned by the convolution layer.

Chapter 5

Frequency Domain Analysis

In this chapter, the weights learned by CNNs are analyzed in frequency domain. Dataset for this analysis includes sinusoids, triangular and rectangular waveforms. This chapter is mainly dependent on Fourier Transforms (FT) and all basic concepts covered in the previous chapter (Chapter 4). We will also look into the effects of padding and dropout regularization while doing analysis in frequency domain.

5.1 Classification of Sinusoids

5.1.1 Problem Description

This is same problem encountered in the Chapter 3. The brief description of the dataset is summarized in the Table 5.1.

Dataset Sampling Frequency : 128 Hz		
Classes	**Type**	**Parameter**
Class 1	Sinusoids	Frequency 5 Hz
Class 2	Sinusoids	Frequency 10 Hz
Description	Both class includes 200 randomly phase shifted version of sine waves. The first class is hot encoded and is represented by [1 0]. Similarly the second class is represented by [0 1].	

Table 5.1: Dataset

CNN Model 1	
Layers	**Settings**
Input	128 samples
Conv. Layers 2	2 Kernels of length 32
Fully connected	-
Output	Categorical output 2 categories

Table 5.2: CNN Model 1

10 Fold Validation		
Min. Accuracy	**Avg. Accuracy**	**Max. Accuracy**
100	100	100

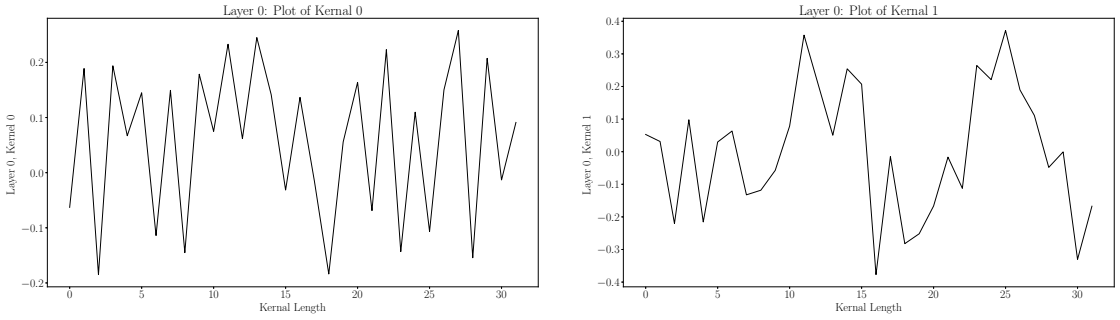
Table 5.3: CNN model 1 validation

5.1.2 CNN Model 1

The same CNN model is used as it is simple to analyze but at the same time very capable of handling this dataset. A brief characterization of the model is given in the Table 5.2.

5.1.3 Result

Validation of the model is shown in the Table 5.3. The two kernel weights after training are plotted in the Figure 5.1.



(a) Kernel 1

(b) Kernel 2

Figure 5.1: Weights

5.1.4 Analysis

The FFT of both kernels are shown in Figures 5.2a and 5.2b. Clearly, kernel 2 is a bandpass filter with center frequency at around 5Hz and kernel 1 is (almost) a bandpass filter with center frequency around at 10 Hz. But, you can also observe that it is not a clean bandpass filter; frequency response at frequencies other than 5 and 10Hz is non-zero with significant response. Subsection 5.3 discusses this phenomenon.

Let's confirm the observed frequency response by using two examples. In the first example, a class 1 input is considered. Convolution of class 1 (5Hz sinusoid) input with both kernels are shown in Figure 5.3.

5.1.4.1 Example 1

Let's begin by listing the properties of a class 1 signal.

- $avg\_value = 0.5$
- $amplitude = 0.5$
- $max\_value = avg\_value + amplitude = 1$
- $min\_value = avg\_value - amplitude = 0$

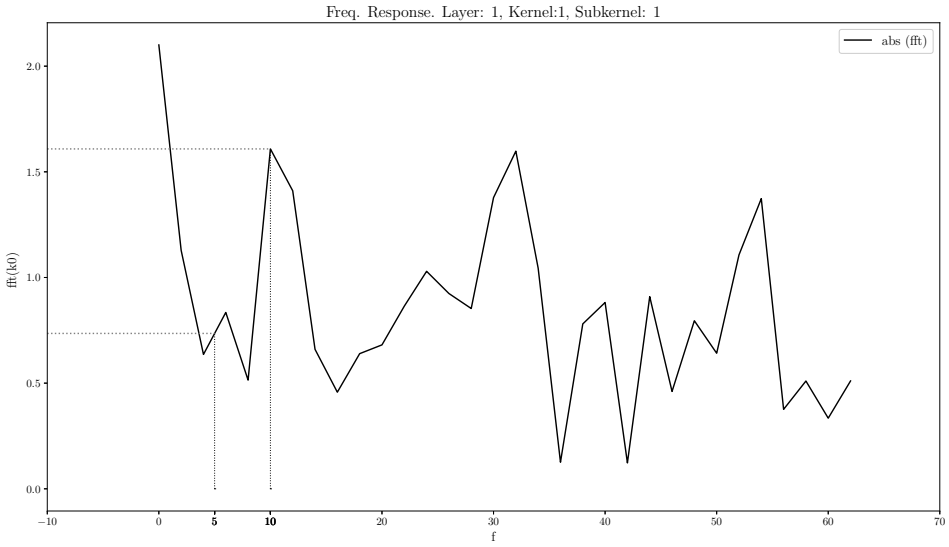
According to the frequency response of Kernel 1, following properties can be listed.

- Frequency response at 5 Hz, $f_5 = 0.75$
- Frequency response at 0 Hz, $f_0 = 2.2$

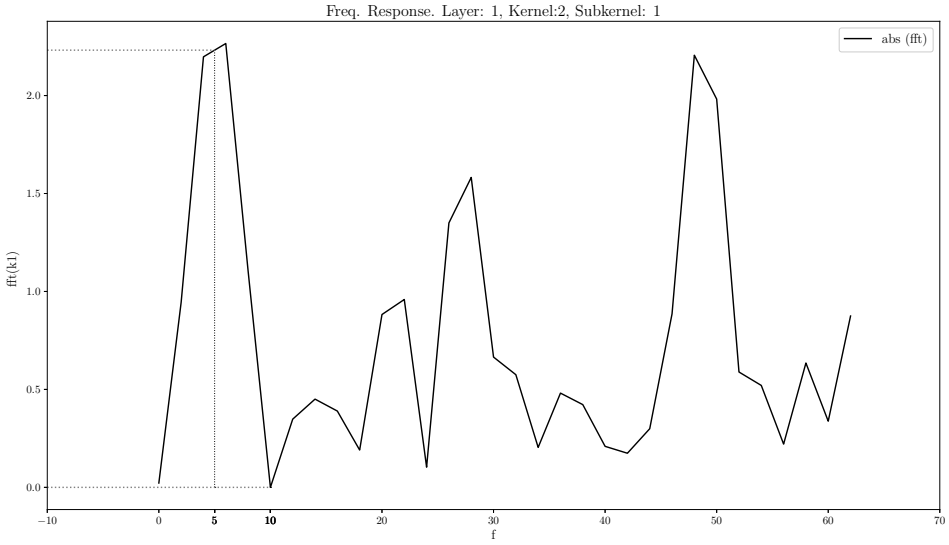
We know that convolution in time is equivalent to multiplication in frequency domain. Hence the output of the convolution should be as follows.

For a 5Hz sinusoid the output is,

- $Avg\_output = avg\_value * f_0 = 0.5 * 2.2 = 1.1$



(a) Frequency response of Kernel 1



(b) Frequency response of Kernel 2

Figure 5.2: FFT of Kernels 1 and 2

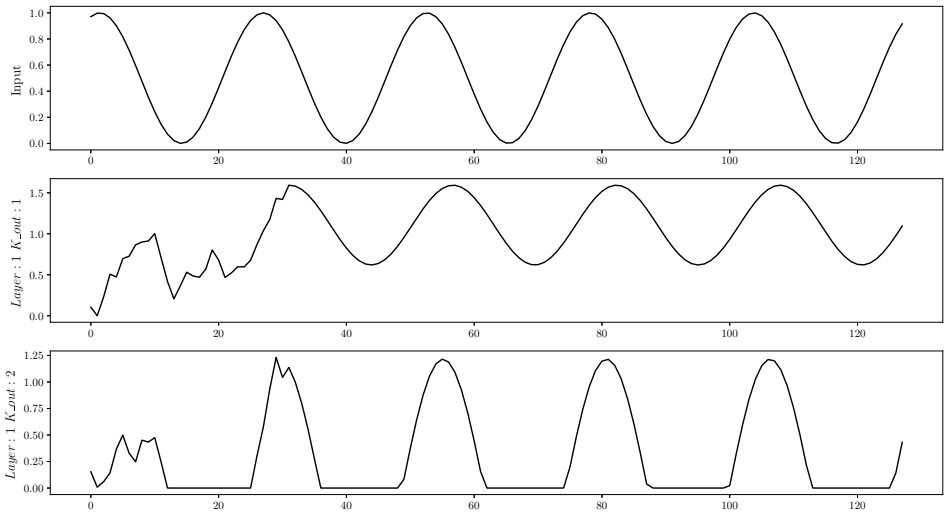


Figure 5.3: Example 1: 5 Hz input

- $Amplitude\_output = f_5 * amplitude = 0.5 * 0.75 = 0.375$
- $Max\_output = Avg\_output + Amplitude\_output = 1.475$
- $Min\_output = Avg\_output - Amplitude\_output = 0.375$

Clearly, the output from Kernel 1 agrees with the above calculation. A similar case can be made for kernel 2. Frequency response at 5Hz is 2.2 units and clearly, the amplitude from convolving kernel 2 with the class 1 input is 1.1 ($0.5 * 2.2 = 1.1$) for most part (except for the glitches). The glitches in the output are discussed in Section 5.4.

5.1.4.2 Example 2: Class 2 input

Let's begin by listing the properties of a Class 2 input.

- $avg\_value = 0.5$
- $amplitude = 0.5$
- $max\_value = avg\_value + amplitude = 1$

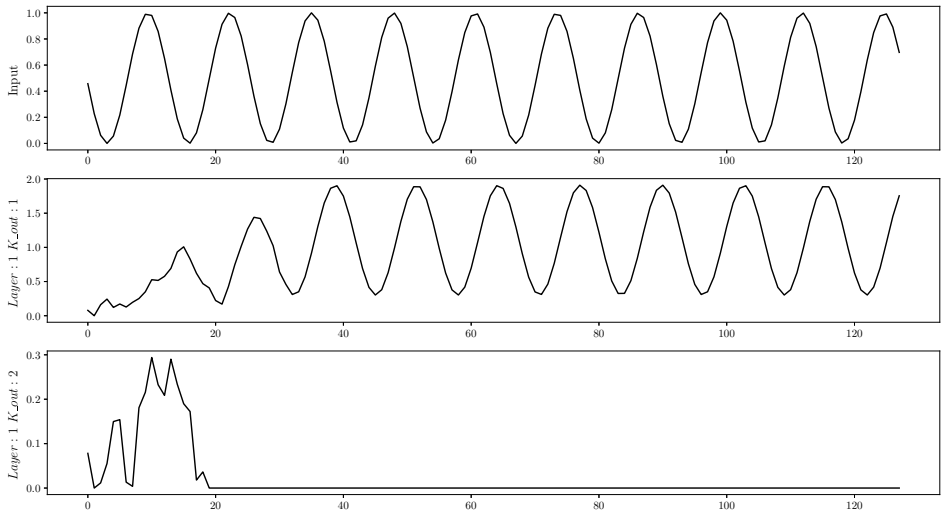


Figure 5.4: Example 2: 10 Hz input

- $min\_value = avg\_value - amplitude = 0$

Let's consider the properties of Kernel 1.

- Frequency response at 10 Hz, $f_{10} = 1.6$
- Frequency response at 0 Hz, $f_0 = 2.2$

Output of the convolution for a 10Hz sinusoid with Kernel 1 should be as follows.

- $Avg\_output = avg\_value * f_0 = 0.5 * 2.2 = 1.1$
- $Amplitude\_output = amplitude * f_{10} = 0.5 * 1.6 = 0.8$
- $Max\_output = Avg\_output + Amplitude\_output = 1.9$
- $Min\_output = Avg\_output - Amplitude\_output = 0.3$

Clearly, the output values and the calculated values match for the most part (except glitches). Now, let's consider the properties of Kernel 2.

- Frequency response at 10 Hz, $f_{10} = 0$
- Frequency response at 0 Hz, $f_0 \approx 0$

Output of the convolution for a 10Hz sinusoid with Kernel 2 should be as follows.

- $Avg\_output = avg\_value * f_0 = 0.5 * 0 = 0$
- $Amplitude\_output = amplitude * f_{10} = 0.5 * 0 = 0$
- $Max\_output = Avg\_output + Amplitude\_output = 0$
- $Min\_output = Avg\_output - Amplitude\_output = 0$

Clearly, Kernel 2 blocks 10 Hz sinusoids which can be confirmed by observing the output of convolution (Figure 5.4).

The dataset studied in this section was very simple and obvious - two sinusoids of two different frequencies. A slightly more complex dataset is considered in the next section to check whether the kernels can pick up frequencies in a non-obvious dataset.

5.2 Classification of Sinusoids in Presence of Noise

5.2.1 Problem Description

Dataset consists of burst of sinusoids in between random noise as shown in Figure 5.5. The noises are uniformly distributed between 0 and 1. The convolution layers in the CNN model are expected to filter out noise before feeding into the output layer.

5.2.2 Results

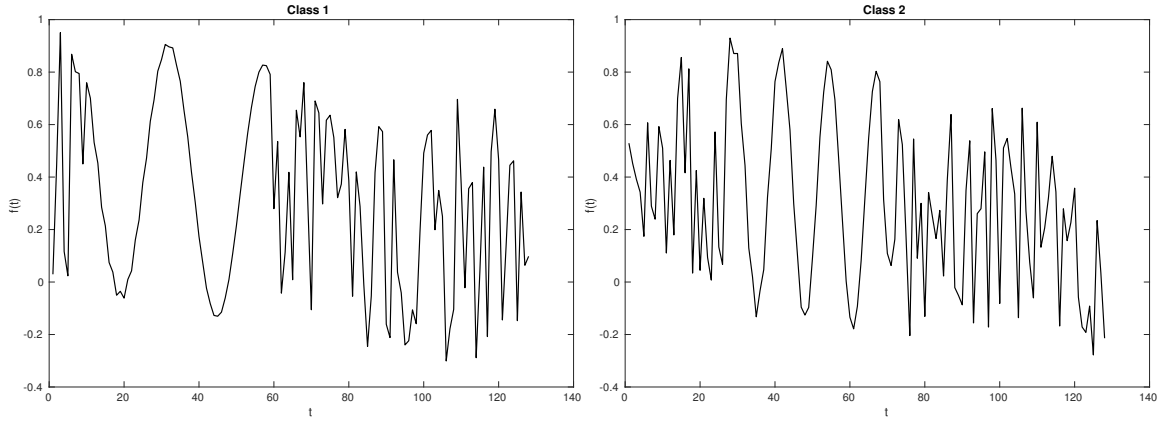
To validated the model, k-fold validation is used with $k = 10$. The 10 fold validation result is as shown in Table 5.5.

Dataset Sampling Frequency : 128 Hz		
Classes	**Type**	**Parameter**
Class 1	Sinusoids + random noise	Frequency 5 Hz
Class 2	Sinusoids + random noise	Frequency 10 Hz
Description	Both class includes 200 random bursts of sine waves of random block sizes. The first class is hot encoded and is represented by [1 0]. Similarly the second class is represented by [0 1].	

Table 5.4: Dataset description

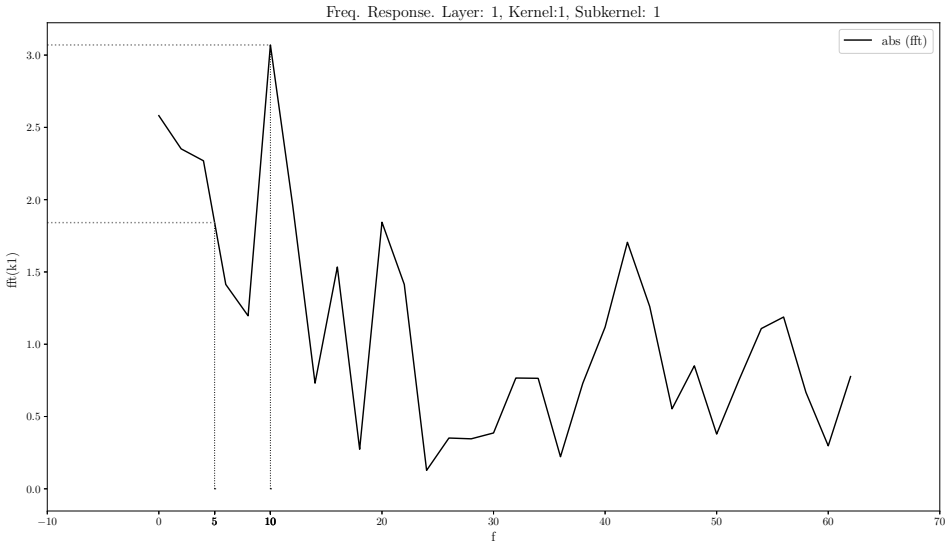
10 Fold Validation		
Min. Accuracy	**Avg. Accuracy**	**Max. Accuracy**
98.75	99.75	100

Table 5.5: CNN model 1 validation

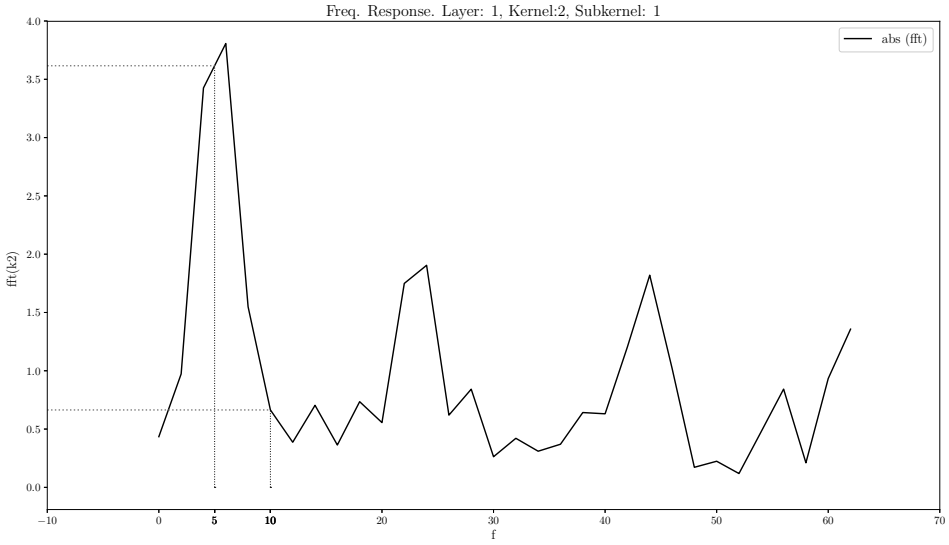


(a) Class 1 (b) Class 2

Figure 5.5: Dataset

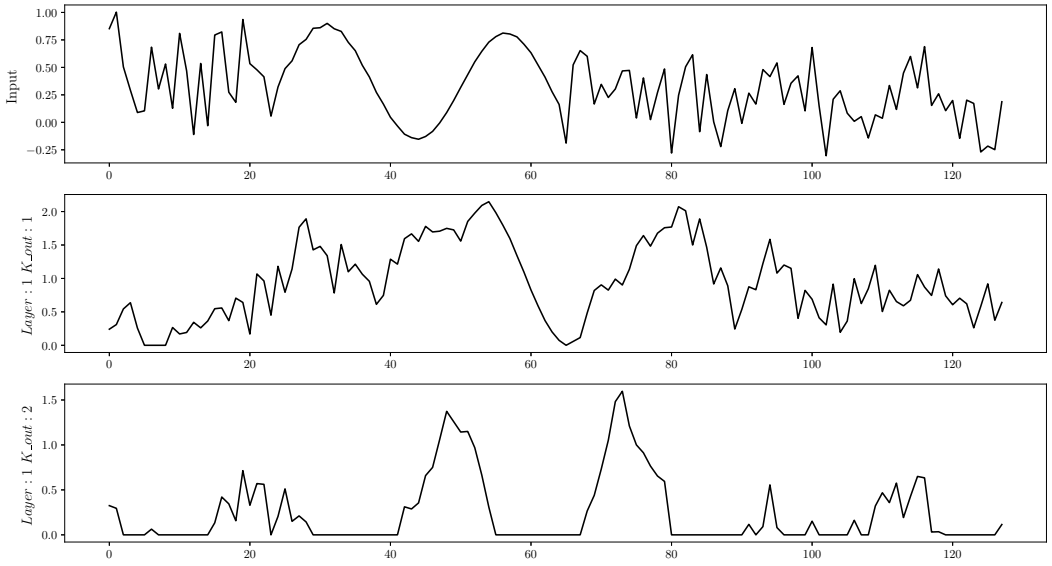


(a) Frequency response of Kernel 1

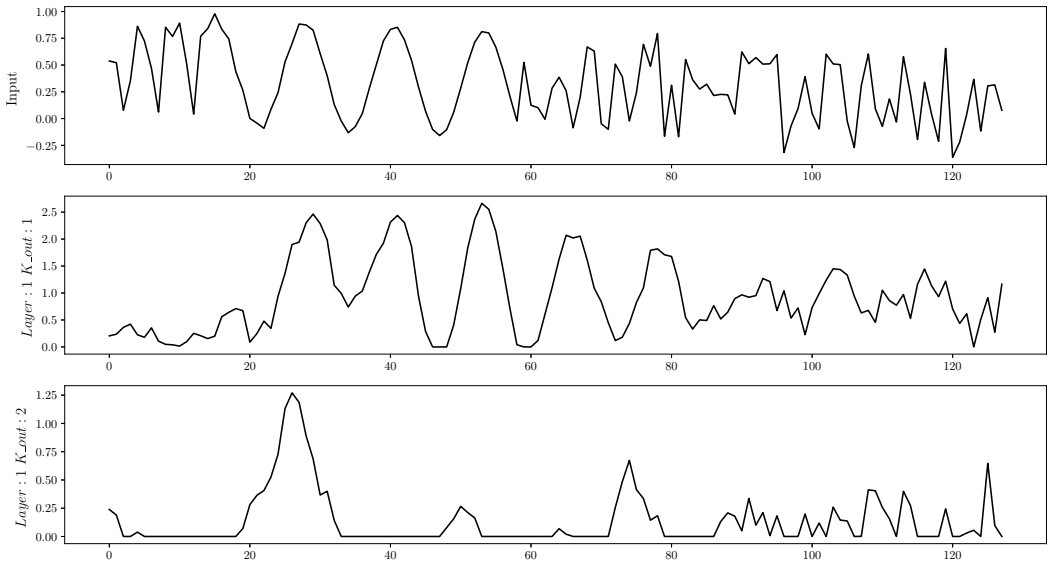


(b) Frequency response of Kernel 2

Figure 5.6: Frequency Response



(a) Example 1



(b) Example 2

Figure 5.7: Examples

5.2.3 Analysis

From the frequency response of the filters, it is clear that the convolution kernels were able to separate sinusoids from the noise. The Kernel 1 is a bandpass filter with center frequency 10Hz and Kernel 2 acts as a bandpass filter with center frequency 5 Hz. Noise suppression in Kernel 1 is not good (as seen in Figure 5.7) because the DC response of Kernel 1 is non-zero whereas the noise suppression is better in Kernel 2 as DC response is close to 0. In the next section, we will study the effect of Dropout regularization and its impact on frequency response.

5.3 Effect of Dropout Regularization

Dropout is one of the famous methods to minimize overfitting. This method is shown to outperform models without dropouts [28] and it is suggested that optimal value of dropout rate is 0.5. In this section effect of dropouts are studied using fft to better understand the effect of dropout regularization.

Let's consider the dataset from the previous section - sinusoids with noise. The CNN model with and without dropouts are explored to understand the effects of dropout.

5.3.1 Without Dropout

Let's consider the output of convolution layer shown in Figure 5.7. The convolution kernels does implement bandpass filters at 5 Hz and 10 Hz but it fails to filter out undesired noise signals. This is because of the undesired response at frequencies other than 5 and 10 Hz. Now, let's modify the CNN Model 1 to accommodate dropout.

5.3.2 With Dropout

Dropouts are normally introduced after a convolution layer and from here onwards this model will be referred to as CNN Model 2. A pictorial depiction of this model is shown in Figure 5.8.

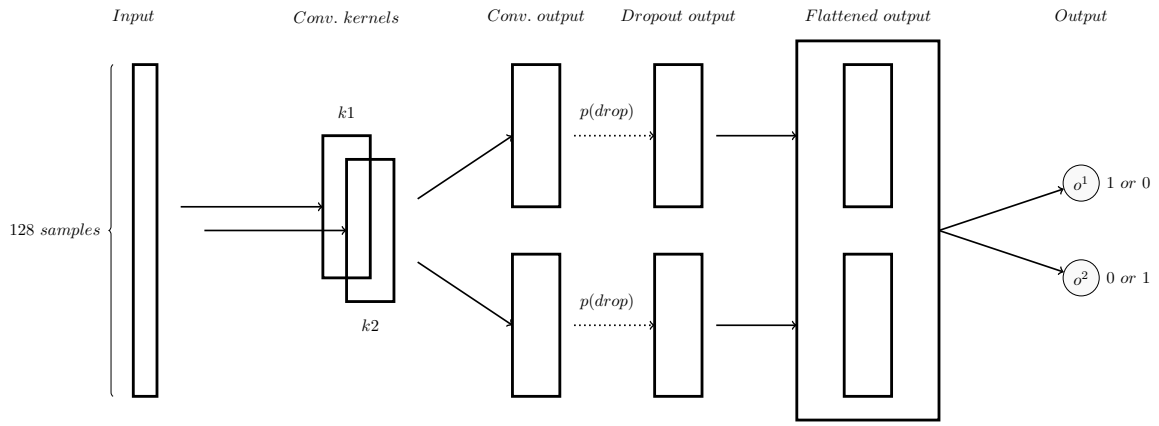


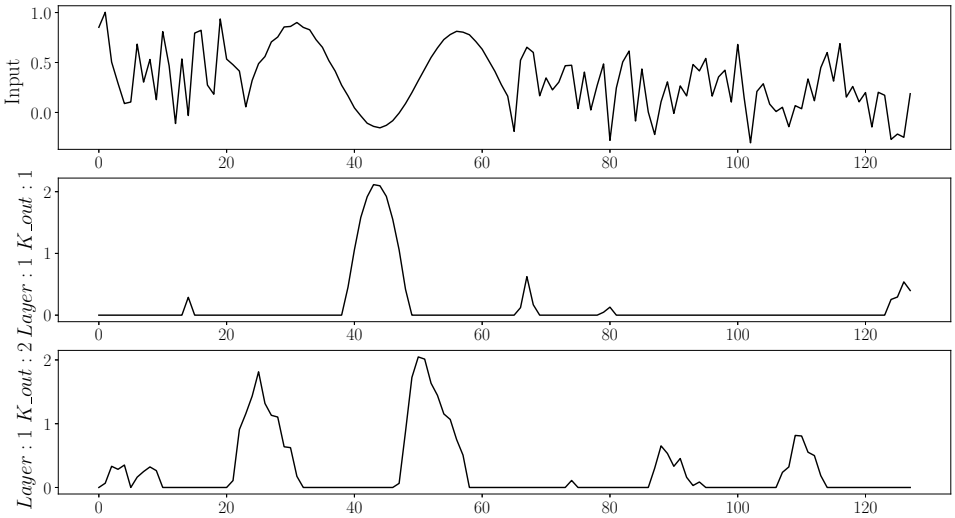
Figure 5.8: CNN Model 2

The convolution output for class 1 and class 2 are shown in the Figure 5.10. Clearly, the noise suppression is much better. Let's analyze the filter in frequency domain (Figure 5.9). Both kernels form a band pass filter at frequencies 5 Hz and 10 Hz and frequency response at undesired frequencies are close to zero.

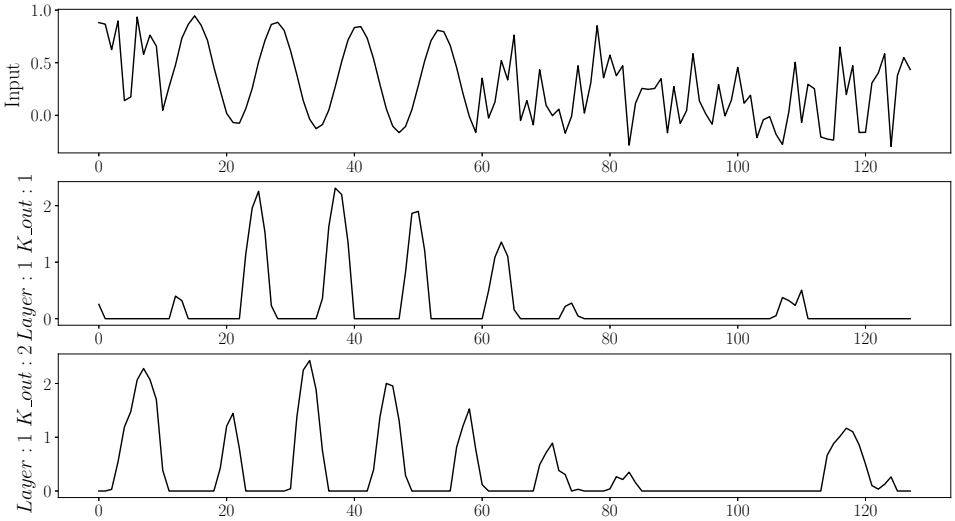
5.3.3 Explanation

It is clear that the kernel picks up most relevant information and filters out noise when a dropout layer is added. Let's say the dropout rate is 0.5 which means half of the output from convolution layer are dropped.

Initially, weights are set randomly. Let's assume that the convolution kernel passes both actual sinusoids as well as noise. There are four possible cases after drop out. In the first case, the majority of the input passed on to the output layer is made up of sinusoids and the output layer classifies the output correctly. This results in no error backpropagation. In the second case, the sinusoids are passed on to the next layer (output layer in this case) and the network misclassifies the output. The errors will be high and the gradients of the error will be backpropagated to the convolution layer. Because of dropout, the error gradients due to random noise will be zero. But the error gradient due to sinusoids will be non-zero. The weights of the convolution layer will now be adjusted in the direction of sinusoids. This is a kind of *filtering* in back propagation phase.

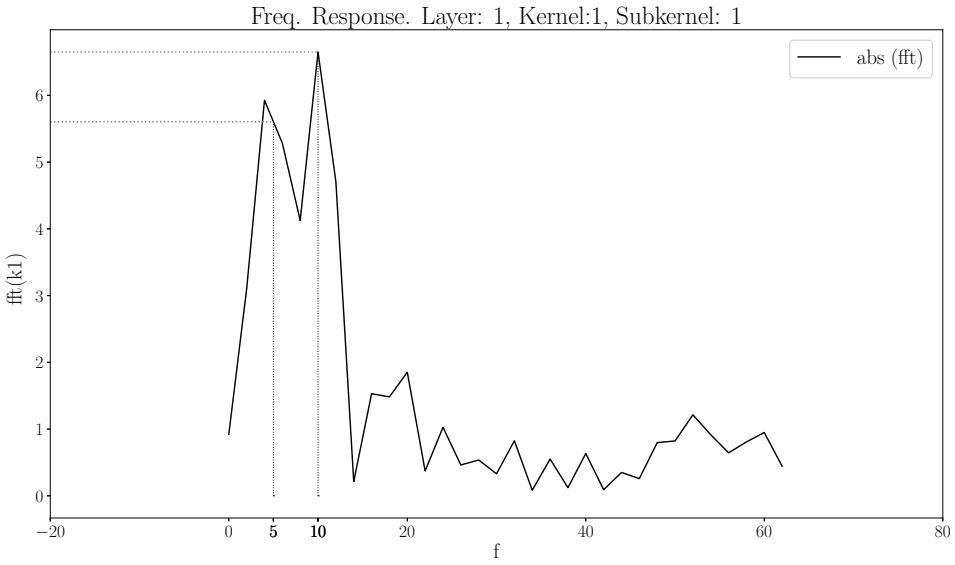


(a) Output of convolution for class 1 input

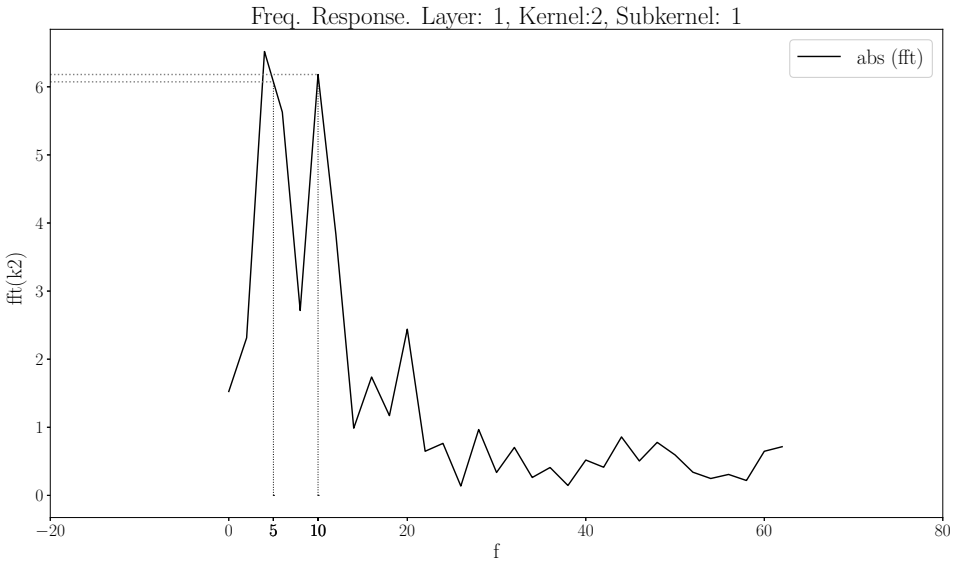


(b) Output of convolution for class 2 input

Figure 5.9: Example outputs with dropout - p(drop) = 0.8



(a) Frequency response of Kernel 1



(b) Frequency response of Kernel 2

Figure 5.10: Frequency response of the convolution kernels for p(drop) = 0.8

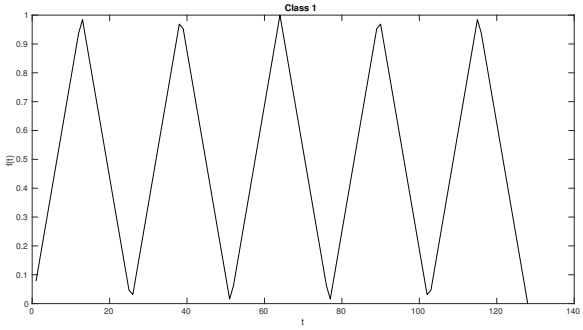
In the third case, majority of the input passed on the next layer will be random noise (carrying no information). If the output layer classifies correctly (by chance) then the convolution weights will not be updated. But if the output layer classifies incorrectly then the convolution weights will be moved further away from the random noises. Backpropagation because of the sinusoids will be zero and hence the convolution weight will still remain to be in the direction of the sinusoids. To conclude, dropout regularization can be thought of as a filter in the backpropagation phase.

5.4 Effect of Different Types of Padding

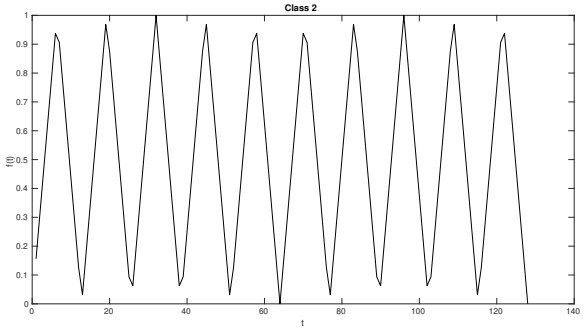
Through out this chapter, zero padding was used to make the convolution output length the *same* as input length. But this introduces undesired outputs at the start and end of the convolution as observed in Figures 5.4 and 5.3. There are two ways to mitigate this problem. The obvious way is not to use padding (No advantages are seen from padding). If the first solution is not acceptable for some reason, dropout can be used which seems to mitigate this problems quite well.

5.5 Triangle or Rectangular Waveform Classifications of Different Frequencies

The classification of two triangle waveforms of two different frequencies (5 and 10 Hz for example) or two rectangular waveforms of different frequencies works similar to classification of sinusoids. As discussed in the previous chapter, FFT of triangle and rectangular waveforms are scaled form of $sinc^2(f)$ and $sinc(f)$ function respectively. Hence, we can expect bandpass filter formations at center frequencies thereby filtering out undesired frequencies.

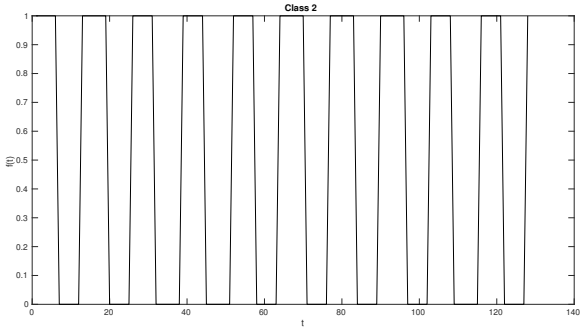


(a) Triangular dataset - Class 1



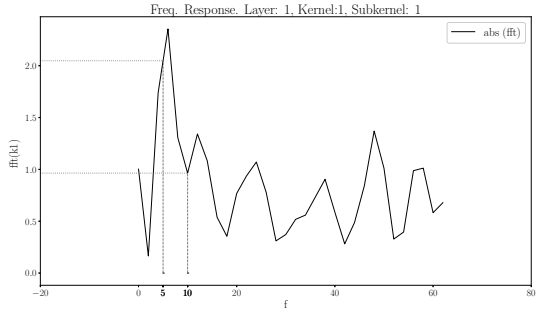
(b) Triangular dataset - Class 2

(c) Rectangular dataset - Class 1

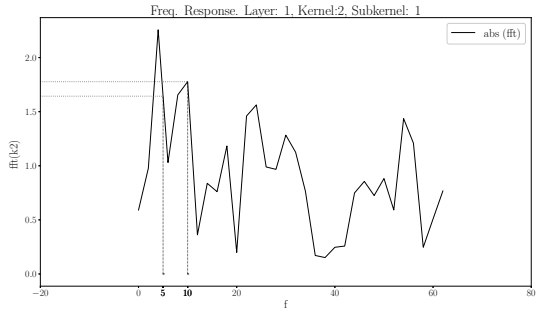


(d) Rectangular dataset - Class 2

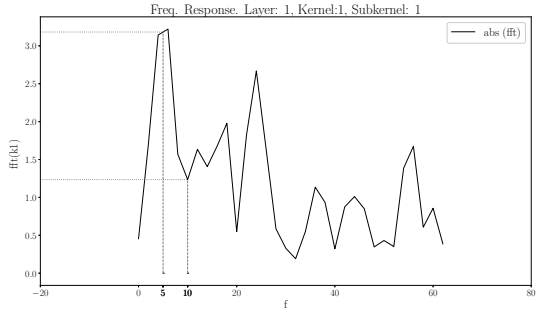
Figure 5.11: Triangular and Rectangular datasets



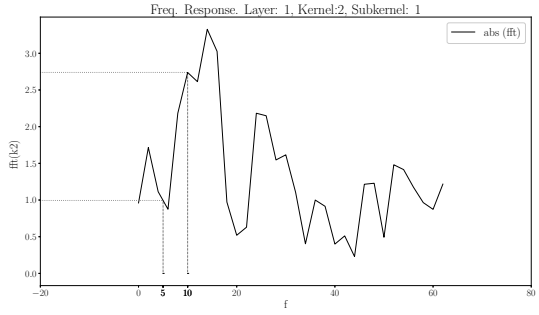
(a) Triangular dataset - Kernel 1



(b) Triangular dataset - Kernel 2



(c) Rectangular dataset - Kernel 1



(d) Rectangular dataset - Kernel 2

Figure 5.12: Triangular and Rectangular datasets

Triangular waveform dataset Sampling Frequency : 128 Hz		
Classes	**Type**	**Parameter**
Class 1	Triangular	Frequency 5 Hz
Class 2	Triangular	Frequency 10 Hz
Description	Both class includes 200 triangular waveforms. The first class is hot encoded and is represented by [1 0]. Similarly the second class is represented by [0 1].	
Triangular waveform dataset Sampling Frequency : 128 Hz		
Classes	**Type**	**Parameter**
Class 1	Rectangular	Frequency 5 Hz
Class 2	Rectangular	Frequency 10 Hz
Description	Both class includes 200 triangular waveforms. The first class is hot encoded and is represented by [1 0]. Similarly the second class is represented by [0 1].	

Table 5.6: Dataset description

5.6 Conclusion

Let's recap few important observations and results from this chapter. In this chapter, classification problems based on sinusoids, triangular and rectangular waveforms were explored and FFT of learned kernels were carefully studied. Kernel weights learned by the convolution layer mimics a FIR filter wherein pass-bands vary based on the problem. Passband were also seen at undesired frequencies without regularization. Dropout helps in mitigating this shortcoming by highlighting the appropriate frequencies for the given problem. Also, a neat explanation for this phenomenon is articulated. Finally, we observed the undesired effect of padding.

Chapter 6

Implications and Extensions

In this Chapter, implications of the observations made so far will be explored along with some extensional experiments corresponding to the learned kernels. This includes implications of time domain analysis, predicting optimal kernel length and the sequencing of multiple target classes in the input signal.

6.1 Time Domain Implications

From Chapter 3, it is clear that convolution kernels learn some kind of smoothing filter (ex: Gaussian filter) for each class. If this is the case, then filter has to obey certain characteristics. In the next subsection, effect of increasing the kernel length is studied.

6.1.1 Kernel Length and Noise

One of the properties of a smoothening filter is, as the length of the filter increases, noise suppressing capability of that filter increases [22]. Let's see if this property holds good for convolution kernels. We revisit the problem of classification of triangular waveforms with different widths. The CNN model for this experiment will just have one kernel in convolution layer and the output of the kernel will be fed to output layer (Table 6.2). The triangle signals are placed amidst random noise. It is crucial that convolution kernels suppress this noise

Dataset Sampling Frequency : 128 Hz		
Classes	**Type**	**Parameter**
Class 1	Triangle block	Width: 40
Class 2	Triangle block	Width: 20
Description	The triangle block lengths are 40 and 20 for class 1 and class 2 respectively. Triangle block is embedded somewhere in between random signals within the 128 sample window. Both classes have 300 such waveforms.	

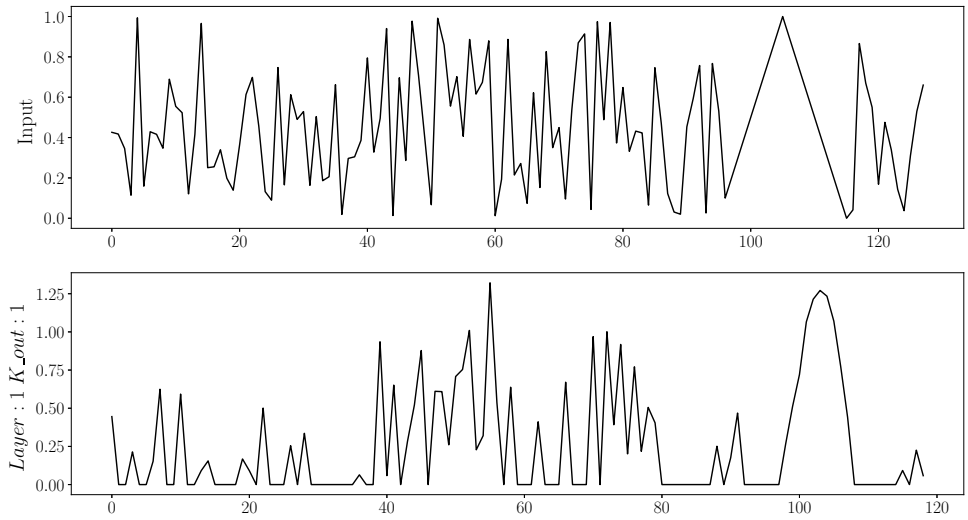
Table 6.1: Dataset

CNN Model 2	
Layers	**Settings**
Input	128 samples
Conv. Layers 2	1 Kernels of varied length
Fully connected	-
Output	Categorical output 2 categories

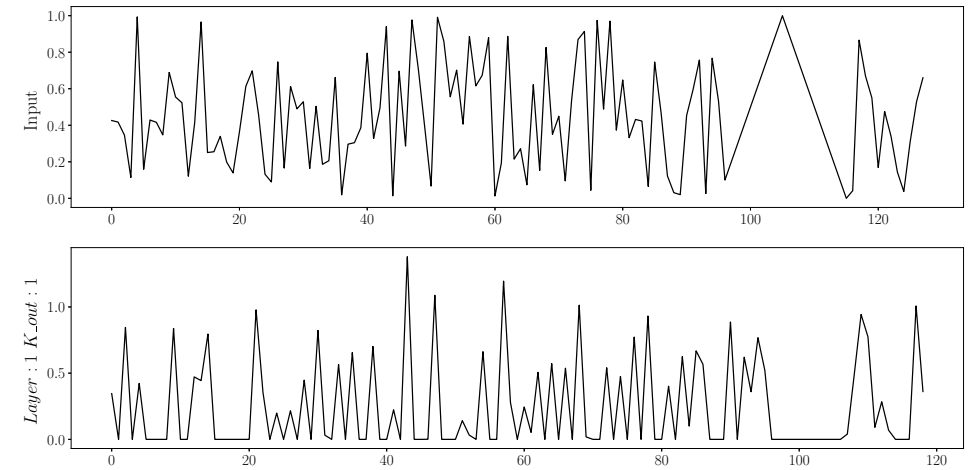
Table 6.2: CNN Model 2

to achieve better results. This experiment is conducted for different kernel lengths and the outputs from convolution layer are plotted in Figures 6.1 and 6.2.

Observe the noise suppression for CNNs with different kernel lengths - 5, 10, 15 and 20. Noise suppression observed in the convolution output for a kernel with length 10 is slightly better than 5. Much better noise suppression is observed when a kernel of length 15 is used instead of 10. By further increasing the kernel size to 20, we see no noise in the output of the convolution. Hence, the noise suppression capability of a CNN improves with the increase in kernel length. In other words, **the *feature selection* capability of a CNN increases with the increase in kernel length** which is the main purpose of this layer. All experiments until this point dealt with CNN having just one convolution layer. In the next subsection, the questions regarding the purpose of multi-layered convolutions are explored.

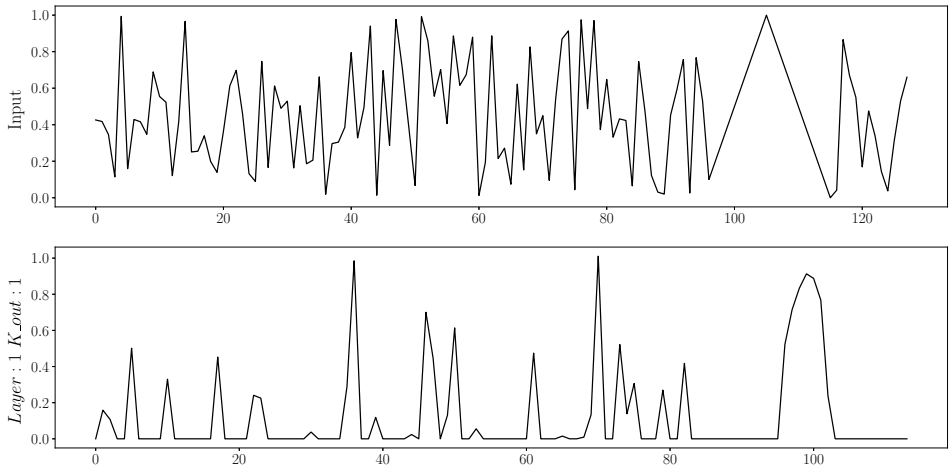


(a) Kernel length = 5

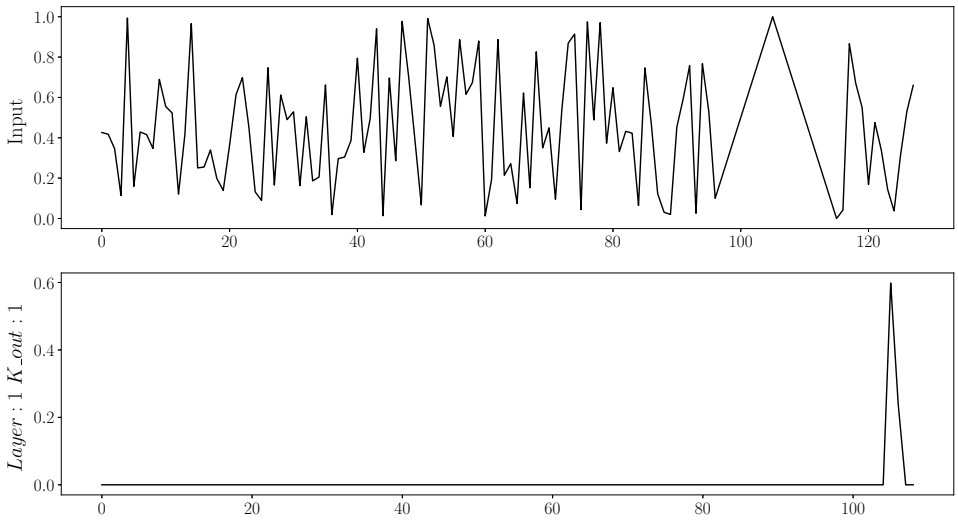


(b) Kernel length = 10

Figure 6.1: Kernels lengths 5 and 10



(a) Kernel length = 15



(b) Kernel length = 20

Figure 6.2: Kernels lengths 15 and 20

6.1.2 Multi-layer Convolutions

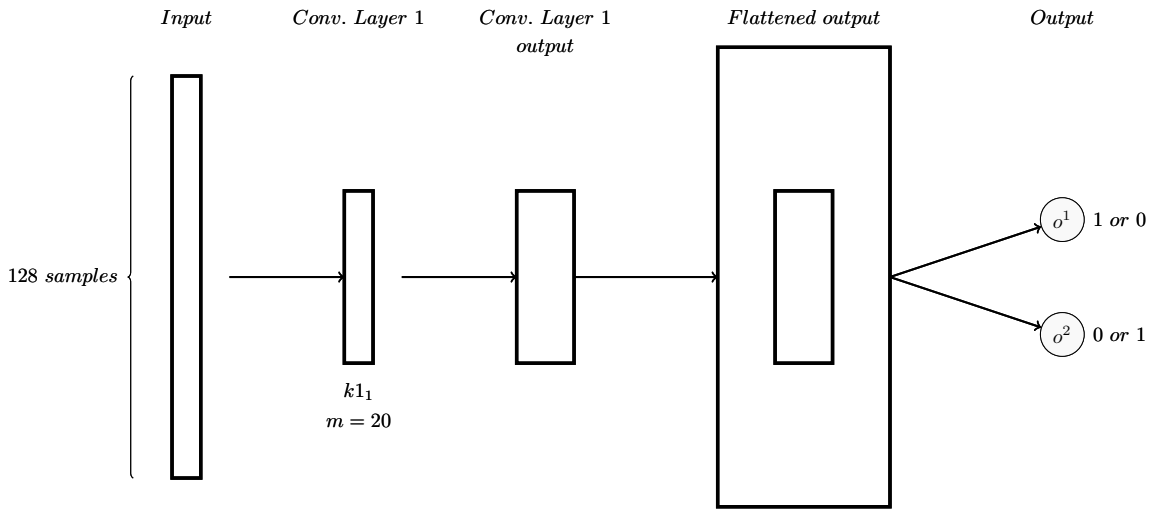
It's a common practice to use CNN with multiple convolution layers to maximize the accuracy especially for object detection and image classification problems. Generally, multiple layer implies better accuracy. One possible explanation for such a behavior is explored in this section.

It is observed that the three passes of 3-point rectangular smooth is equivalent to 7-point pseudo-Gaussian smooth [22]. In general, p passes of a m-length smoothening is equivalent to smoothening using a kernel of length $pm - p + 1$. If convolution layers of CNN were to satisfy this property, then the noise suppression observed for a kernel with length 20 (total noise suppression shown in Figure 6.2) should also be observed for a CNN model with 2 layers and a kernel in each layer of length 10 (as depicted in Figure 6.3b). That is, the model with $m = 10$ and $p = 2$ equivalent to model with $m = 10 * 2 - 2 + 1 = 19 \approx 20$.

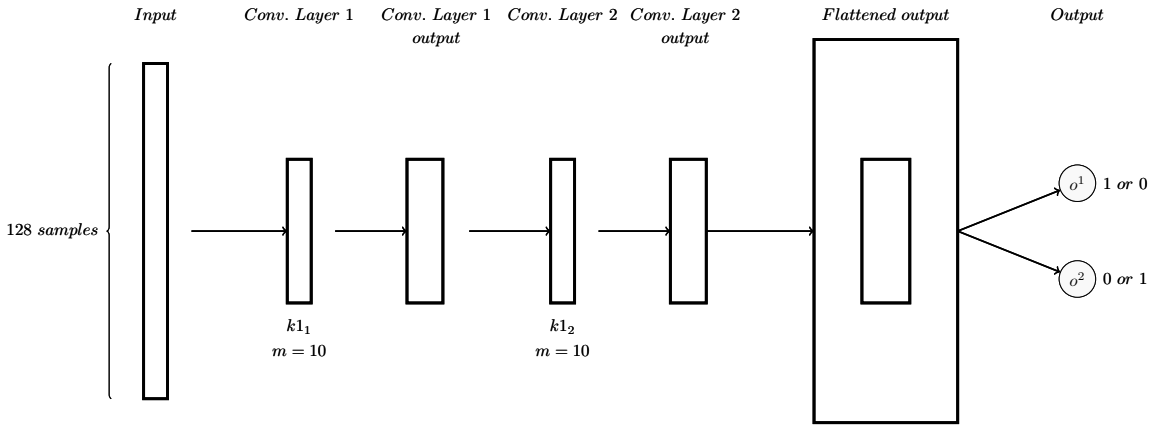
The architecture for CNN with two layers is shown in Figure 6.3b. Two layers in the CNN model have just one kernel each. The intermediate outputs of these kernels are shown in Figure 6.4. It can be observed that the output of first layer still contains lot of noise. But in the second pass, the noise is further suppressed and the noise suppression is comparable to the noise suppression observed in the convolution output of a CNN with just one kernel of length 20. Hence, **multi-layer convolution layers improve the features extraction capability of a CNN**. Until now, time domain implications were considered. In the next section, implications of frequency domain analysis will be explored.

6.2 Frequency Domain Implications

From Chapter 5, it is clear that a convolution kernel can act as a *bandpass* filter. This is a well studied concept in DSP and hence the convolution kernel is expected to exhibit certain properties. And one such property of a FIR filter describes the relationship between kernel length, sampling frequency and the desired transition band. This relation

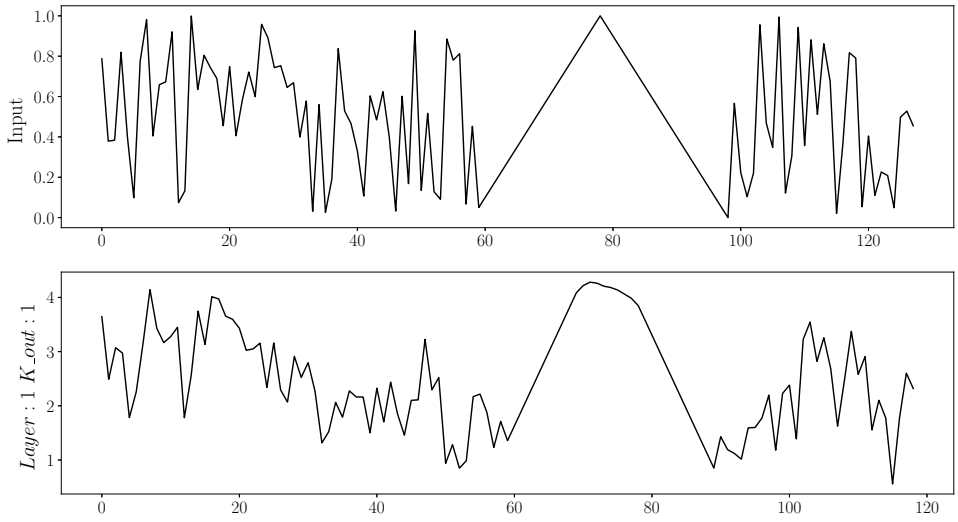


(a) CNN model with just one convolution layer

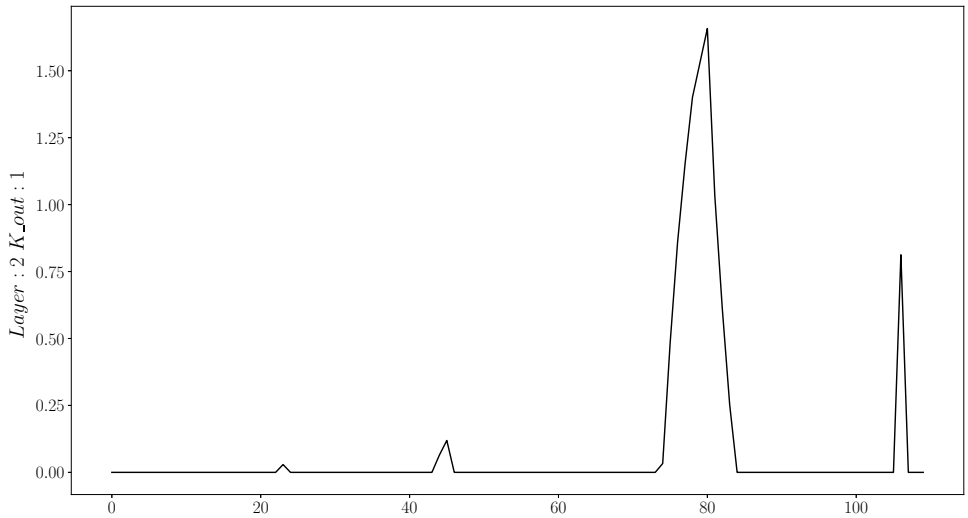


(b) CNN model with two convolution layers

Figure 6.3: CNN model with one and two layers



(a) Input and convolution layer 1 output (first pass)



(b) Convolution layer 2 output (second pass)

Figure 6.4: Output of convolution layers 1 and 2

is tested for convolution layers in the next subsection.

6.2.1 Kernel Length, Sampling Frequency and Bandwidth

Referring to literature [9], there exists an *empirical* relationship between *filter length* (k), *sampling frequency* (f_s) and *transition band* (Δf). An estimate of kernel length k is given by

$$k = (\frac{2}{3})\ (\frac{f_s}{\Delta f})\ log(\frac{1}{10\delta_1\delta_2}) \tag{6.1}$$

where, δ_1 and δ_2 are desired pass band and stop band ripples respectively. The transition band (Δf) is given by

$$\Delta f = f_1 - f_2 \tag{6.2}$$

where f_1 is the pass band edge and f_2 is the stop band edge. After numerous experiments, it is observed that the convolution kernel doesn't exactly follow Equation 6.1 but the kernel length required to achieve certain accuracy is directly proportional to the sampling frequency (f_s) and inversely proportional to difference between frequencies belonging to different classes (Δf). To understand the kernel length dynamics with respect to f_s and Δf, consider the following experiment.

Experiment Set 1 In this experimental setup, the transition band Δf is held constant and f_s is increased from 128 to 256 Hz, then to 512, 1024 and 2048 Hz. According to the earlier hypothesis, the kernel length required to achieve certain accuracy increases as f_s increases. The dataset contains sinusoids of frequencies 15, 16 and 17 Hz. Because the CNN has to distinguish between these frequencies, the transition band $\Delta f = 1$ Hz. The CNN with input size 128 is used to classify the dataset into three classes. Different kernel sizes are used and the size of first kernel to achieve an average accuracy of 90% and an accuracy of 100% is recorded. The same experiment is repeated for datasets with 256, 512

Dataset Sampling Frequency : 128, 256, 512 or 1024 Hz		
Classes	**Type**	**Parameter**
Class 1	Sinusoid	Frequency: 15 Hz
Class 2	Sinusoid	Frequency: 16 Hz
Class 3	Sinusoid	Frequency: 17 Hz
Description	Pure sinusoids of random phase shifts with frequencies 15, 16 or 17 Hz. Each class contains 200 such waveforms.	

Table 6.3: Dataset

CNN Model	
Layers	**Settings**
Input	128, 256, 512 or 1024 samples
Conv. Layers 2	1 Kernel of various lengths
Fully connected	-
Output	Categorical output 3 categories

Table 6.4: CNN Model

and 1024 Hz sampling frequencies. The dataset and CNN model used for this experiment is summarized in Tables 6.3 and 6.4 respectively. The recorded kernel sizes are shown in the Table 6.5

From the table, it's clear that kernel length required to achieve certain accuracy increases as sampling frequency increase. Thus, **the kernel length is directly proportion to f_s**. In the next experiment, Δf is varied with f_s fixed at 128 Hz. As the Δf increases, the kernel length required to achieve certain accuracy should decrease.

Result		
Sampling Frequency	***k* to achieve an avg. accuracy > 90%**	***k* to achieve an accuracy = 100 %**
128	15	15
256	25	15
512	45	25
1024	50	25
2048	55	35

Table 6.5: Results

Result		
Δf	**Actual Frequencies**	**Minimum Kernel Length to achieve an average accuracy > 90%**
0.1	(15, 15.1, 15.2)	45
0.2	(15, 15.2, 15.4)	25
0.3	(15, 15.3, 15.6)	25
0.4	(15, 15.4, 15.8)	25
0.5	(15, 15.5, 16.0)	20
0.6	(15, 15.6, 16.2)	20
0.7	(15, 15.7, 16.4)	15
0.8	(15, 15.8, 16.6)	10
0.9	(15, 15.9, 16.8)	10
1.0	(15, 16.0, 17.0)	10
2.0	(15, 17.0, 19.0)	10
3.0	(15, 18.0, 21.0)	10
4.0	(15, 19.0, 23.0)	10
5.0	(15, 20.0, 25.0)	10

Table 6.6: Result

Experiment Set 2 In this experiment, the f_s is fixed at 128 Hz. Dataset consists of three classes with different frequencies. The difference between these three frequencies are varied and the kernel length required to achieve an average accuracy of 90 % is recorded (as shown in Table 6.6). The CNN model used for this experiment is similar to the previous experiment except, two kernels are used instead of one kernel.

Clearly, the kernel length required to distinguish three sinusoids of frequencies 15, 15.1 and 15.2 is very high compared to kernel length required to distinguish sinusoids of frequencies 15, 17 and 18. That is, as the Δf increases, the kernel length required to achieve certain performance decreases. Hence, **kernel size required to achieve certain performance is inversely proportional to the transition band Δf.**

Combining the results from the above experiments, **the desired kernel length to achieve certain performance is directly proportional to the sampling frequency, f_s and inversely proportional to the transition band, Δf.** In the next section, the role of multiple kernels is explored by experimenting on classification of sequences of signal

Result	
Number of kernels	**Accuracy**
1	87
2	93
3	95
4	96
5	96

Table 6.7: Result - Dataset With Noise

classes.

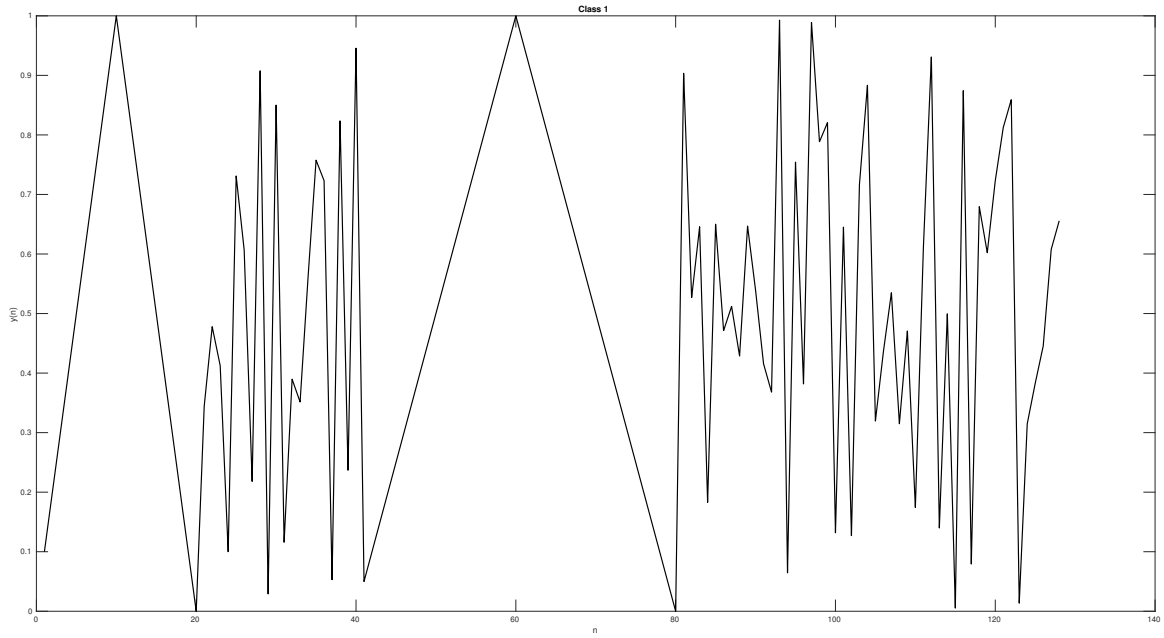
6.3 Distinguishing Sequences

Until now, all problems that we dealt involved detection of some kind of waveform - sinusoids of different frequencies, triangles of different widths etc. Next logical step is to test whether CNN can classify based on the location of the waveform. One of the simplest problem which involves location is the classification problem based on the location of a signal relative to another signal.

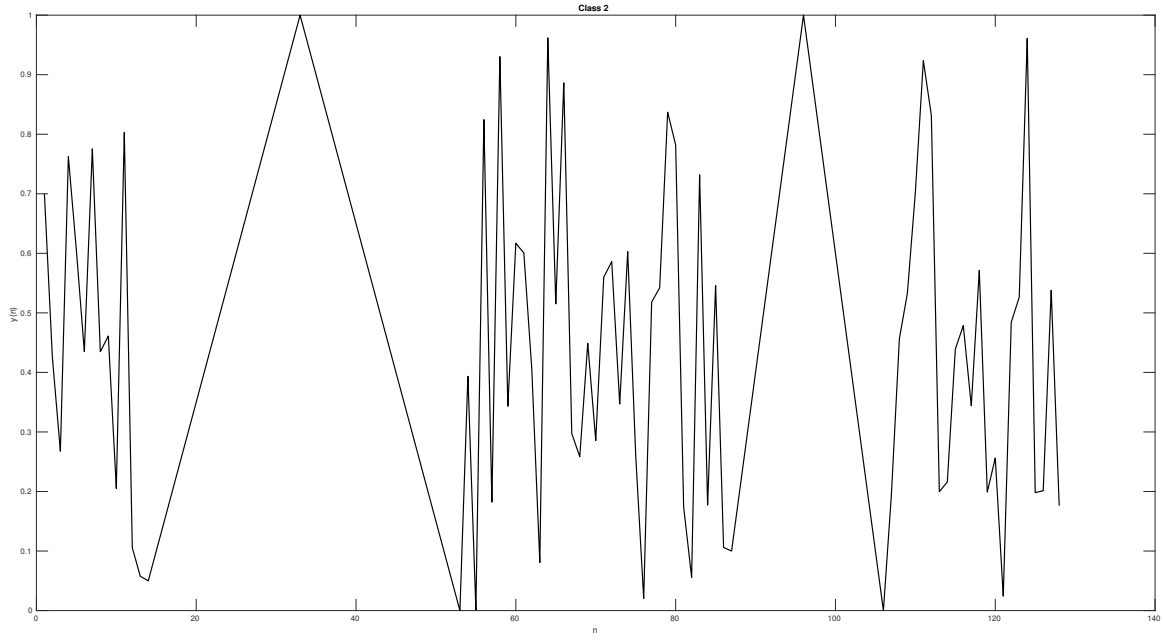
Let's consider two triangular waveforms of widths 20 and 40. If triangle of width 20 precedes the triangle of width 40 then it belongs to Class 1 and if the triangle of width 40 precedes the triangle of width 20, it belongs to Class 2. In the first experiment, triangular waveforms are embedded in between random noise and in the second experiment, random noises are removed and hence the dataset just contains triangular waveforms.

6.3.1 Experiment 1

In this experiment, triangles are embedded somewhere in between the random noise as shown in Figure 6.5. Now, the CNN model with just *one layer* is used to classify the dataset. Number of kernels in the first layer is varied and the corresponding 10-fold accuracies are noted in the Table 6.7. As the number of kernels are increased, the accuracy increase up to a point.



(a) Class 1



(b) Class 2

Figure 6.5: Sequence Dataset

Result	
Number of kernels in second layer	**Accuracy**
1	89
2	96
3	94
4	96
5	96

Table 6.8: Result - CNN with Layer 2

Result	
Number of kernels in second layer	**Accuracy**
1	96
2	100
3	100
4	100
5	100

Table 6.9: Results - Dataset Without Noise

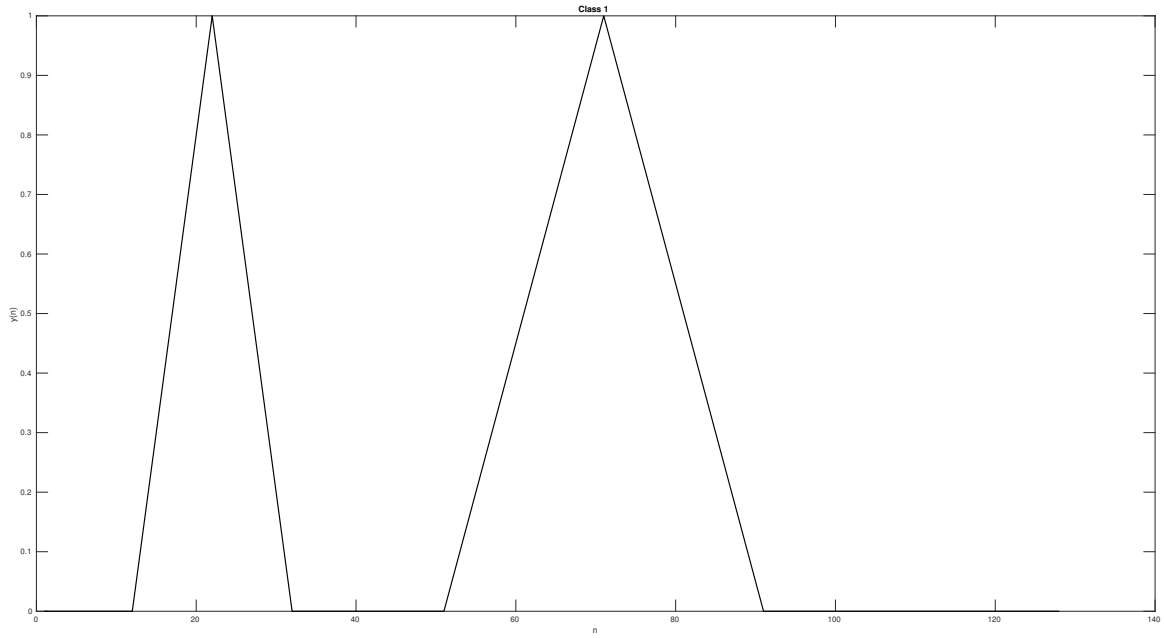
6.3.2 Experiment 2

In the last experiment, number of kernels in the first layer was varied and there was no second convolution layer. Now, we will include the second convolution layer. The number of kernels in the first convolution layer will be fixed at 2 and the number of kernels in the second convolution layer will be increased and the corresponding accuracies are noted down in the Table 6.8.

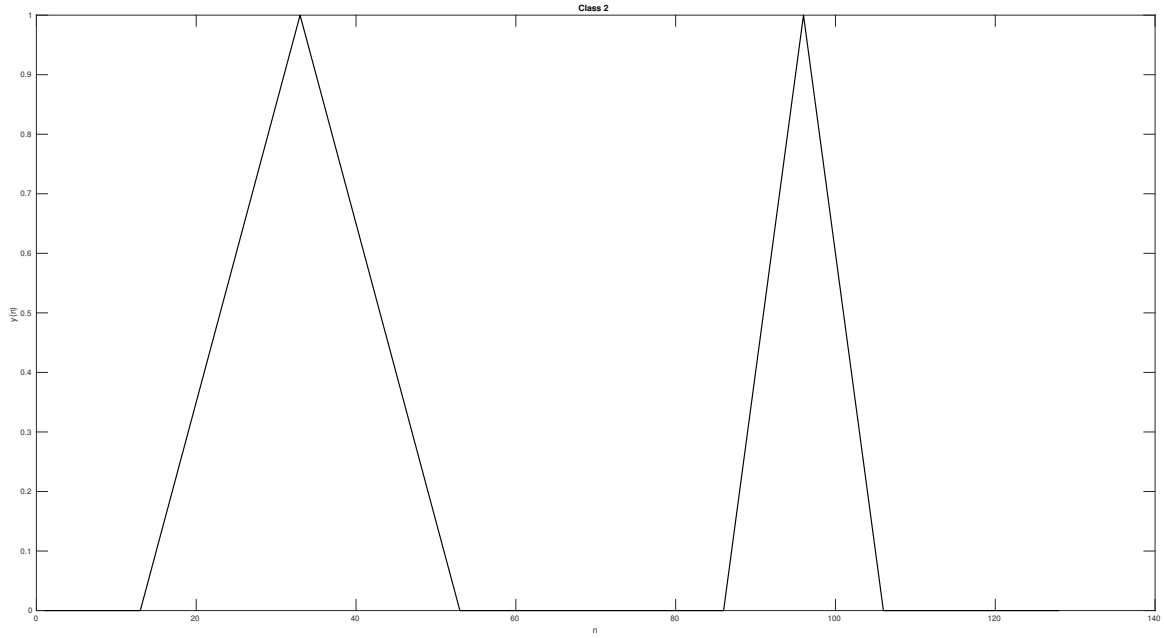
Table 6.8 looks a lot like Table 6.7 which suggests that layer 2 doesn't have much of an impact on the classification problem. As we know from the time domain implications, the second layer helps in reducing the noise present in the input. In the next experiment, dataset will not contain any noise and if CNN with just one layer can provide better results, then the noise was limiting the performance of CNN model.

6.3.3 Experiment 3

The dataset for this experiment is shown in Figure 6.6. The CNN model with just one layer is used to classify the dataset and the results are tabulated in the Table 6.9.



(a) Class 1



(b) Class 2

Figure 6.6: Sequence Dataset Without Noise

The average accuracy for this problem by using CNN with more than one kernel is 100%. For the classification of the noise embedded sequence dataset, it is clear that the noise in the dataset was the limiting factor. Hence, increasing the number of layers doesn't have too much of a role in the classification except to reduce the noise in the dataset. But, increasing the number of kernels does increase the accuracy. This is because, each kernel gives a different perceptive of the input which is analogous to looking at the problem through different colored shades. The output of different kernels are concatenated in the flatten layer and are fed to fully-connected layer or the output layer. This results in repeated input data but each set of input is filtered differently by different kernels. Hence, it gives the output layer much more information to classify the dataset.

Chapter 7

Conclusions and Future Work

In this thesis, an effort was made to understand 1D Convolutional Neural Networks with respect to time varying signals. CNNs are generally thought of as a trainable black box which yields state of the art performance for many intricate problems such as object detection. To understand what a CNN learn under different settings for carefully crafted inputs, two approaches were employed - time domain and frequency domain analysis. In the time domain analysis, the learned weights were compared to the well known filters. We observed CNN learning second order Gaussian filter and Sav-Gol filters to distinguish datasets with different slopes. These learned weights are kind of a smoothening filter and it's implications were studied in Chapter 6. One of the possible reasons why a multi-layered CNN works better than the single layer CNN was proposed. Time domain analysis cannot be used to justify the learned weights for all kinds of problems. Hence, the learned weights were studied in frequency domain in Chapter 5.

In Chapter 5, we observed CNNs learning bandpass filters around different center frequencies and with different bandwidths based on the given problem. Also in that chapter, the effects of dropout regularization were studied and a possible reason as to why it works was proposed. Because of the frequency domain analysis, effects of sampling frequency (f_s) and transition band (Δf) on the kernel length was inferred in Chapter 6.

Future Work There are numerous aspects which were not considered in this thesis. One of them is regarding the multi-layered convolutions. Although this thesis touched this aspect, a further analysis could reveal even more interesting properties of a CNN model. Generally, a CNN model employs some kind of pooling with different strides. A further research on this topic can be beneficial in fine tuning of CNN models. While doing convolution, one can employ different strides. This thesis employed a stride of 1 but other strides could also be used. Analyzing the kernel weights for such a setting is an interesting topic in itself. Ensemble learning is another new technique in the field of deep learning. A study on this technique could result in achieving better performance. Finally, an effort to classify sequences were made in Chapter 6 but as the number of signals in the input increases, the complexity of the problem grows exponentially. Analysis of CNN learned weights for sequencing problems may yield further insights into the working of CNNs.

Appendices

Appendix A Introduction to Keras

Keras is high level neural network package intended to enable fast experimenting. High level implies, you don't need to write code for backpropogation, convolution, passing output from one layer to another, etc. All you need to know is the neural network architecture and input data with which you want to train the network. Hence, keras is suitable for research where prototype can be rapidly built and tested.

You may have heard about other tools such as Tensor Flow[1][6], Theano[2][7] etc. These are also high level neural network tools. But keras is built on top of these tools and hence it is at an even higher level in the hierarchy.

A Installation

Keras can be installed easily with either pip3, conda, apt or with almost any other package manager.

A.1 Starting with Keras

There are two ways to implement a neural network using Keras.

1. Sequential Model[5] - Stack of layers
2. Keras functional API[2] - For complex architectures

For our purposes (and for most applications), sequential model is more than sufficient.

A.2 Sample Architecture

Let's consider a simple convolutional neural network (CNN) with following architecture. This architecture is pictorially represented in Figure 1.

[1]Tensor Flow is an open-source symbolic tensor manipulation framework developed by Google.
[2]Theano is an open-source symbolic tensor manipulation framework developed by LISA Lab at Universit de Montral.

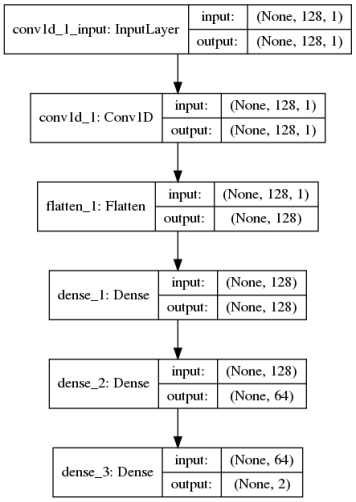


Figure 1: CNN example architecture

1. Input Layer
   (a) 1D data of size 128.
2. Convolutional Layer
   (a) 1D kernel and hence accepts just 1D data.
   (b) Single convolutional layer.
   (c) Two kernels of sizes 32.
3. Fully connected layers / Hidden layers
   (a) Two hidden layers.
   (b) First layer with 128 neurons and second layer with 64 neurons.
4. Output Layer
   (a) Hot encoded outputs - 2 outputs.

A.3 Sequential Model

Now let's build this architecture in keras. Start by creating a keras object called Sequential.

```
model = Sequential()
```

This object contains all keras related functionalities. Let's start to build our neural network by adding a convolutional layer. Input to this conv. layer will be treated as input layer[3].

```
model.add (Conv1D (no_kernels, length_of_kernel, activation='relu
    ', padding="same", input_shape=(n_i, 1)))
```

In our architecture, the input is a vector of length 128. And we are using two convolutional kernels with the size 32. Hence,

```
n_i = 128
no_kernels = 2
length_of_kernel = 32
```

And with respect to padding, multiple options are available[1]. If 'same' option is chosen, the output of the convolution will have same length as the input.

Because we are using two kernels, we get two separate output vectors. These output vectors must be concatenated and then fed to fully connected layer. This is accomplished using flatten() method.

```
model.add (Flatten ())
```

Now, it's time to add fully connected layers. We have two fully connected layers - first layer has 128 neurons and the second layer has 64 neurons.

```
model.add (Dense (128, activation='relu'))
model.add (Dense (64, activation='relu'))
```

[3]You can also specify the input explicitly using model.input() method. See appendix A

Finally, the output layer has two neurons for two classes.

```
model.add (Dense (2, activation='softmax'))
```

You might have noticed that nothing much was said about activation. That's because Keras provides so many options to choose from. Here are a few.

1. softmax -typically used as output layer activation for classification problems.
2. relu
3. elu
4. selu
5. softplus
6. softsign
7. tanh - used to be very famous. Not so much these days.
8. sigmoid
9. hardsigmoid
10. linear
11. LeakyReLU - gaining popularity.
12. PReLU etc.

All we have done until now is to "define" our architecture. Once you define the architecture, keras knows the number of layers, number of neurons in each layer, number of convolutional layers and number of kernels in each convolution etc. Next step is to compile - check if something's wrong and then built the necessary tensors for weights, inputs and outputs.

A.4 Compile

Compiling the model is pretty straight forward. You will have to specify some more options which are discussed below.

```
model.compile(loss='mse', optimizer='adam', metrics=['accuracy'])
```

Here, loss function is nothing but error function which you want to minimize. The 'mse' stands for mean squared error. You can refer [3] for other kinds of loss functions. Optimizer in this case refers to the algorithm using which you change weights in backpropogation. If you wish to use gradient descent optimizer instead of adam, you change the optimizer to 'sgd' (which stands for stochastic gradient descent).

Now, if you want to calculate accuracy after each epoch, you can use accuracy metric. And again be sure to check [4] for other types of metrics.

A.5 Train

When you are ready with input vectors along with corresponding target vectors, you can train your neural network.

```
model.fit(x_train, y_train, batch_size=11, epochs=500, verbose=1)
```

Here x_train is a stack of input vector and y_train is its corresponding target. Batch size and epochs are the hyper-parameters that you can choose. If verbose is 1, it will display cost function/error function and accuracy after each epoch.

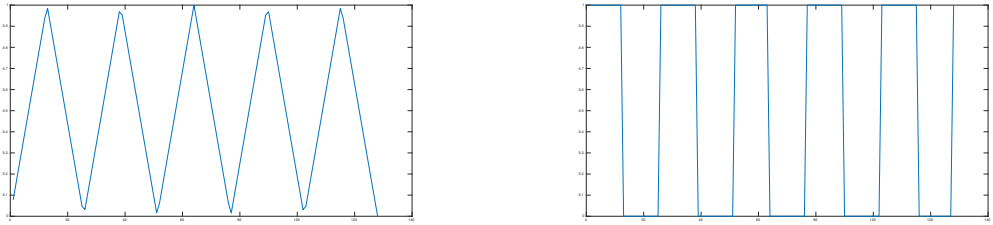
Let's consider a trivial classification problem. Input data contains two class of signals - 1000 samples of triangle and rectangular signals as show in Figure 2.

```
Using Theano backend.

Training output:
Epoch 1/10
2000/2000 [==============================] - 0s 75us/step - loss: 0.0312 - acc: 0.9860
Epoch 2/10
2000/2000 [==============================] - 0s 72us/step - loss: 9.9016e-08 - acc: 1.0000
Epoch 3/10
2000/2000 [==============================] - 0s 72us/step - loss: 8.7489e-08 - acc: 1.0000
Epoch 4/10
2000/2000 [==============================] - 0s 79us/step - loss: 7.6080e-08 - acc: 1.0000
Epoch 5/10
2000/2000 [==============================] - 0s 116us/step - loss: 6.5643e-08 - acc: 1.0000
Epoch 6/10
2000/2000 [==============================] - 0s 109us/step - loss: 5.6482e-08 - acc: 1.0000
Epoch 7/10
2000/2000 [==============================] - 0s 108us/step - loss: 4.8631e-08 - acc: 1.0000
Epoch 8/10
2000/2000 [==============================] - 0s 112us/step - loss: 4.1959e-08 - acc: 1.0000
Epoch 9/10
2000/2000 [==============================] - 0s 110us/step - loss: 3.6284e-08 - acc: 1.0000
Epoch 10/10
2000/2000 [==============================] - 0s 111us/step - loss: 3.0897e-08 - acc: 1.0000


Test output:
2000/2000 [==============================] - 0s 17us/step
Accuracy:100.0
```

Figure 3: CNN example output



(a) Class 1: Triangle wave

(b) Class 2: Rectangle wave

Figure 2: Input Data

A.6 Test

If you want to test your learned network, you use following method which takes input and target as parameters. Figure 3 shows output after training and accuracy obtained for test input.

```
model.evaluate(x_test, y_test, verbose=0)
```

Bibliography

[1] Keras - convolution, 2018. [Online; accessed 17-June-2018].

[2] Keras - functional, 2018. [Online; accessed 17-June-2018].

[3] Keras - losses, 2018. [Online; accessed 17-June-2018].

[4] Keras - metric, 2018. [Online; accessed 17-June-2018].

[5] Keras - sequential, 2018. [Online; accessed 17-June-2018].

[6] Keras - tensorflow, 2018. [Online; accessed 17-June-2018].

[7] Keras - theano, 2018. [Online; accessed 17-June-2018].

[8] Guillaume Alain and Yoshua Bengio. Understanding intermediate layers using linear classifier probes. *CoRR*, abs/1610.01644, 2016.

[9] Maurice G. Bellanger. *Digital Processing of Signals: Theory and Practice*. John Wiley & Sons, Inc., New York, NY, USA, 1984.

[10] Convolutional Neural Networks for Visual Recognition. Cs231n: Convolutional neural networks for visual recognition, 2018. [Online; accessed 17-June-2018].

[11] Deep learning. Deep learning — Wikipedia, the free encyclopedia, 2018. [Online; accessed 17-June-2018].

[12] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning, 2016. cite arxiv:1603.07285.

[13] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[15] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[16] Henry W. Lin and Max Tegmark. Why does deep and cheap learning work so well? *CoRR*, abs/1608.08225, 2016.

[17] Tianyi Liu, Shuangsang Fang, Yuehui Zhao, Peng Wang, and Jun Zhang. Implementation of training convolutional neural networks. *CoRR*, abs/1506.01195, 2015.

[18] Richard G. Lyons. *Understanding Digital Signal Processing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1996.

[19] Stéphane Mallat. Understanding deep convolutional networks. *CoRR*, abs/1601.04920, 2016.

[20] Mahesh Chandra Mukkamala and Matthias Hein. Variants of rmsprop and adagrad with logarithmic regret bounds. *CoRR*, abs/1706.05507, 2017.

[21] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 807–814, USA, 2010. Omnipress.

[22] Tom O'Haver. *A Pragmatic Introduction to Signal Processing*. 12 2017.

[23] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *CoRR*, abs/1511.08458, 2015.

[24] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Neurocomputing: Foundations of research. chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, Cambridge, MA, USA, 1988.

[25] SavitzkyGolay filter. Savitzkygolay filter — Wikipedia, the free encyclopedia, 2018. [Online; accessed 17-June-2018].

[26] Robert J. Schalkoff. *Artificial Neural Networks*. McGraw-Hill Higher Education, 1st edition, 1997.

[27] Wenzhe Shi, Jose Caballero, Lucas Theis, Ferenc Huszar, Andrew P. Aitken, Christian Ledig, and Zehan Wang. Is the deconvolution layer the same as a convolutional layer? *CoRR*, abs/1609.07009, 2016.

[28] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.

[29] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages III–1139–III–1147. JMLR.org, 2013.

[30] Jianxin Wu. Introduction to convolutional neural networks. 2017.

[31] Shujian Yu, Robert Jenssen, and José C. Príncipe. Understanding convolutional neural network training with information theory. *CoRR*, abs/1804.06537, 2018.