UNIVERSITY OF OSNABRÜCK

MASTER THESIS

# Theoretical considerations towards deep binary convolutional neural networks from Fourier analysis

*Supervisor:*
Prof. Dr. phil.KAI-UWE KÜHNBERGER

*SubSupervisor:*
ULF KRUMNACK, M.A.

*Author:*
Hiroshi SAWADA

*A thesis submitted in fulfillment of the requirements for the degree of Master of Cognitive Science*

*in the*

Artificial Intelligence Group
Department of Cognitive Science

September 26, 2016

# Declaration of Authorship

I, Hiroshi SAWADA, declare that this thesis titled, "Theoretical considerations towards deep binary convolutional neural networks from Fourier analysis" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

*"Whether gods exist or not, there is no way to get absolute certainty about ethics. Without absolute certainty, what do we do? We do the best we can. "*

Richard Stallman

UNIVERSITY OF OSNABRÜCK

# *Abstract*

Faculty Name
Department of Cognitive Science

Master of Cognitive Science

**Theoretical considerations towards deep binary convolutional neural networks from Fourier analysis**

by Hiroshi SAWADA

Deep convolutional neural network exhibits great power to capture nonlinear mapping having dominant position for image processing [1], speech processing [7], and natural language processing [8] in recent years. However, it has not yet well revealed the reasons the algorithm holds extraordinary capabilities to solve these problems. Thereby, this thesis aims to describe intrinsic properties that this algorithm has from an aspect.

Besides, the network of deep learning needs a lot of computation to be trained and its memory for formed weights has considerable amount of size. Slow convergence and requirement of large amount of memory makes this algorithm difficult to be trained as online-learning.

One of the approach to overcome this problem is having the network only binary values, either +1 or -1 [5]. The target object which turns to be binary means only weights or both weights and values of nodes. The way of computation which reveals state-of-art outcome is introduced in [6]. I came up with slightly different computation of convolution from the approach of this paper, and this way of computation is displayed on chapter2.

Chapter1 describes basic knowledge of back propagation which is most fundamental idea of convolutional neural networks.
Chapter2 pays attention of convolutional neural networks itself, and new ways of computation for binary neural networks is introduced here.
Chapter3 states the relevance of dimensional reduction and compression. Binary neural network can be seen as sort of quantized figure of normal convolutional neural networks which should have some relevance to lossy compression.
Chapter4 contains description of Fourier analysis to understand binary input data and binary weights better.
Chapter5 shows you analysis of weights which come from training of binary neural networks, and conclude all chapters.

This paper holds many exemplifications of displayed ideas with codes.
It partially because this thesis is not a paper which notes only results, but a thing that I shall show what I have studied.
From a different point of view, it is because I suppose making things clear and deliberated to the detail is needed particularly about algorithms named "deep learning". The reason is what is occurring in CNN tends to be black boxed for many developers. This comes partially from natures of CNN but driven by usefulness of pre-released library.

Code is written by Haskell.
I acknowledge this makes rather difficult to be read for many readers.
Nevertheless, it gives ones very well structured representation of code that fit a thesis. I believe, also practically, Haskell is going to be introduced for machine learning tasks due to its support of parallel computing.

I declare I have written every lines of codes with my understanding. Comprehensive figures of code can be seen in [4]

I wish that reading this thesis guides readers more profound considerations of this area.

# *Acknowledgements*

Thanks to my solid academic training of cognitive science for my two years master in Osnabrueck, Germany.

Stepping back to myself just after entering this institute, I have realized that this master course of cognitive science could provide most of the answers towards my questions about cognition which I have wondered in my early life. Also, a lot of challenging tasks which should be pursued during the rest of my life came out from different perspectives which I haven't focused ever. This period in Osnabrueck literally turned to be my turning point in my life.

The two major I picked up, Neuro-informatics and artificial intelligence are one of the most exciting fields where you could apply what I have studied to the various challenging problems in industries. It was pretty fortunate that I could have studied these fields from great professors with those who holds curiosity which is more or less shared with me.

Specifically, I would like to express my sincere gratitude to four professors in Neuro-informatics group and Artificial intelligence group.

Prof. Dr.-Ing. Gunther Heidemann gave me solid training of machine learning and computer vision, where some part of my thesis is addressed to what I have learned then.

Prof. Dr. rer. nat. Gordon Pipa provided us a study project corporating with IBM where we could develop some softwares with Bluemix and Watson without any cost.

Priv. Doz. Dr.-Ing. Helmar Gust gave me his diverse knowledge of classical artificial intelligence with actual programming skills. My prolog skills which I have acquired in his lecture is absolutely contributing to my implementation by Haskell in this thesis.

Prof. Dr. phil. Kai-Uwe Kühnberger not only offered several interesting seminars but supervised this thesis.

Also, I would like to thank Ph.D Ulf Krumnack who also supported this thesis and offered me some chances to work on this theme.

Lastly, I would like to sincerely thanks to my father and mother who financially supported me. I hope I can show you my achievement in an explicit way after my graduation.

# Contents

**Declaration of Authorship** iii

**Abstract** viii

**Acknowledgements** ix

**1 Back propagation** 1
- 1.1 Overview . . . 1
- 1.2 Feed-forward process . . . 1
  - 1.2.1 Formalization . . . 1
  - 1.2.2 Code representation . . . 3
  - 1.2.3 Notation with function application . . . 4
- 1.3 Calculation of error derivatives . . . 5
  - 1.3.1 Error derivatives for the weights on second layer . . . 5
  - 1.3.2 Error derivatives for the weights on the first layer . . . 6
  - 1.3.3 Interpretation of error derivative calculation . . . 7
  - 1.3.4 Derivative of sigmoid function . . . 8
  - 1.3.5 Code representation . . . 9

**2 Convolutional Neural Network** 11
- 2.1 Prelude . . . 11
- 2.2 Drawback of training in more than 4 layers perceptron . . . 11
- 2.3 Feed-forward process of convolutional neural network . . . 12
  - 2.3.1 1D Feed-forward summation of convolution . . . 12
  - 2.3.2 Code representation . . . 13
  - 2.3.3 Inner product based feed-forward . . . 15
  - 2.3.4 Two dimensional linear summation of feed-forward . . . 20
  - 2.3.5 code representation of two dimensional linear summation of convolution . . . 20
  - 2.3.6 Activation function . . . 22
  - 2.3.7 Pooling layer . . . 23
- 2.4 Calculation of error derivatives . . . 24
  - 2.4.1 Difference between multilayer perceptron and back-propagation . . . 24
  - 2.4.2 Error derivatives of element-wise summation after convolution . . . 24
  - 2.4.3 Derivatives of activation function . . . 25
  - 2.4.4 Derivatives of pooling . . . 25
  - 2.4.5 Distribution of overall error derivatives with respect to each errors . . . 25
  - 2.4.6 Code representation of error propagation . . . 26
  - 2.4.7 Code representation of updating weights . . . 27
- 2.5 Interpretation of Inner product based feed-forward / feed-back . . . 29
- 2.6 Computation for binary input and weight with inner product based feed-forward / feed-back . . . 31
- 2.7 Further discussion . . . 33

**3 Compression for dimensional reduction** 35
- 3.1 Overview . . . 35
- 3.2 dimensional reduction and compression . . . 35
  - 3.2.1 Data Number and Dimension . . . 35
  - 3.2.2 Independent and sequential dimension . . . 35
  - 3.2.3 Compression and dimensional reduction . . . 36
- 3.3 Dimensional reduction for independent data . . . 36

3.3.1 run length encoding . . . 36
3.3.2 Huffman coding . . . 37
3.4 Dimensional reduction for context dependent data . . . 40
3.4.1 Compression for symbolical data (Prediction by partial matching ) . . . 40
3.4.2 Compression for non-symbolical data ( JPEG algorithm ) . . . 40
3.4.3 Consideration towards Jpeg encoding algorithm . . . 48
3.4.4 Summery . . . 48

**4 Fourier analysis of CNN** **51**
4.1 Prelude . . . 51
4.2 Fourier Transformation . . . 51
4.3 Fast Fourier transformation . . . 54
4.4 Convolutional theory . . . 55
4.5 Representation of signals after Fourier transformation . . . 57
4.5.1 Delta function . . . 57
4.5.2 Sinc function . . . 57
4.6 Frequency domain of filters . . . 61
4.6.1 Filter size . . . 61
4.6.2 Fourier transformation of filters on which binary weights are put . . . 63
4.6.3 Fourier transformation of binary input data . . . 63
4.7 Binary character representation . . . 63

**5 Analysis of Binary neural network** **65**

**Bibliography** **67**

# Chapter 1

# Back propagation

## 1.1 Overview

In this section, I will introduce most basic algorithm for training neural networks which is back propagation [3] particularly for two layers perceptron. This algorithm is necessary to introduce CNN in the following chapter. However, back propagation is not only for CNN, and most generally explained with multilayer perceptron. Thereby, I will as well explain it with multilayer perceptron.

Back propagation algorithm consists of two steps mathematically

1) feed-forward
2) derivative of Error with respect to each weight

result of second calculation tells you as an implementation, should be divided into two distinct operation, which consists of three

1) feed-forward
2) error-propagation
3) weights update

## 1.2 Feed-forward process

### 1.2.1 Formalization

Feed Forward process is regarded as nonlinear mapping from dimension of data to teacher class. This is denoted as following.

$$n_{d2j} = \phi_1((w_{1ij} + b_j) \cdot n_{d1i}) \tag{1.1}$$

$$n_{d3k} = \phi_2((w_{2jk} + b_k) \cdot n_{d2j}) \tag{1.2}$$

where $i \in I, j \in J, k \in Kd \in D$ ,
I = number of input neurons,
J = number of hidden neurons,
K = number of output neurons,
b = bias ,
D = number of training data,
$n_{dai}$ = activation of each neurons,
a = number of layer,
$w_{abc}$ = activation of weight,
b = index of node in previous layer which forms a vertex of weight,
c = index of node in subsequent layer which forms another vertex of weight,
$\phi_a$ = activation function ,

Weights can be included into the neurons of front layer which is denoted as following.

$$h_j = \phi_1(w_{1ij} \cdot (g_i + b_0)) \tag{1.3}$$

$$o_k = \phi_2(w_{2jk} \cdot (h_j + b_0)) \tag{1.4}$$

This states following.

Input Node ; $n_{d1i}$

- Input node except bias term is equivalent to training data. But, Input dimension have to be vector even given data has more than two Rank. For instance, image can be treated as a vector as well as signal data.
- Each data can be either serially (Online learning) or parallely (Batch learning) passed to input node. Combination of both online and batch learning is called mini batch learning. For online learning and batch learning, one does not need to prepare all of data before training, but training process is slower than batch learning since there is a dependency in the order of feeding data in.

Hidden Node ; $n_{d2j}$

- Hidden layer is an only layer whose number of node can be set as parameter by ones to implement.
- Number of hidden neuron is considered with the number of data, the number of teacher class, intrinsic dimension of data, and complexity of task. There is no deterministic estimation to get most proper N, thereby determined by inductively. Complexity of task is derived from the combination of training data and teacher data. For instance, discrimination of XandY data is regarded as an easier task than XOR data given Euclidean distance between pairs of data which is supposed to be in the same class.

Output Node ; $n_{d3k}$

- Number of output neurons have to be equal to the number of class which you would like to classify if the task is classification.
- Note that number of input node is determined by descriptive dimension + bias, whereas, the number of output node is absolutely no relevant to the input dimension, since this comes from teacher data which is generally less than number of data. For instance, if the dimension of data is 1, and the number of output class is 100, Ninput, hidden, output is N 2,X,100

Weight ; $w_{1ij}, w_{2ij}$

- Weight is often initially randomized, and its values are floating values between -1 and 1. Algorithm itself is deterministic, however, initial condition ends up with different result.
- One of weights have to be independent from activation of previous layer's neurons so that the network could represent the item which is not attributed from input neurons.

Activation function $\phi_a$

- There are some well known activation function such as sigmoid, tanh , softmax, and rectfied linear unit.
- One of the properties of activation function(sigmoid,hyperbolic tangent,softmax) in feed-forward process is preventing overshooting of fires of neurons, and squashing inner dot of previous weights and nodes to certain values.
- activation function of hidden layers and output layers can be different.

### 1.2.2 Code representation

In this section, I will put python and Haskell code because comparing both gives us small insights about parallel computing.

Following is the feed-forward process in python without using matrix library.

```
for i in range(hiddenNum):
    for j in range(inputNum):
        hidden_value[i] += activationF(w1[j][i] * input_value[j])
```

Since the process is inner product with vector and matrix whose rank is two, it can be written as 2 loop.

Next, is Haskell code.

```
data Weight = Weight [[Double]] deriving (Show)
data Node   = Node     [Double]  deriving (Show)

--weight initialize
wInit :: (Int,Int) -> Weight
--vector*vector
vv  :: [Double] ->  [Double]  -> [Double]
--vector*matrix
vm  :: [Double] -> [[Double]] -> [Double]
--sigmoid function
sigmoid :: Double -> Double
--feed-forward integration function
feedForward :: (Double -> Double) -> Node -> Weight -> Node

wInit (a,b) = Weight [[0.1,0.7],[-0.4,0.1]]
vv a b = foldr (\(x,y) z -> x * y : z) [] (zip a b)
vm a b  = map (\x ->foldl (+) 0 $ vv a x ) b
feedForward f (Node a) (Weight b) = Node $ map f $ vm a b
sigmoid x = 1 / (1+exp(-x))

let w1 = wInit (2,2)
let w2 = wInit (2,2)

let hidden = feedForward sigmoid input  w1
let output = feedForward sigmoid hidden w2
```

Inner product is written as combination of map, zip, and reduce(foldl and foldr in Haskell).
In skeletal parallel computing [2] , it is indispensable to distinct generalized loop into higher order functions especially map and reduce.

Map is applying a function from a list to another list whose element is 1:1 relation which does not affect other mapping at all. This is a loop which enables us completely separate on different processors or different computers.
Zip is receiving two lists and create two dimensional tuple appending two elements of the list.
Reduce is applying a function to a list and generates results which caused from elements from an original list.

Coming back to the Haskell code of back propagation, "feedForward" is defined by multiplication of matrix and vector, and it is defined by multiplication of vector and vector.
Haskell type of both is defined as

```
map :: (a -> b) -> [a] -> [b]
zip :: [a] -> [b] -> [(a, b)]
reduce (foldr) :: (a -> b -> b) -> b -> t a -> b
```

Each computational cost can be estimated by

Given
number of processors = p
number of list elements = n
cost of applying function = t

$map\ f = (\frac{n}{p})t_f$
$zip \quad = \frac{n}{p}t$
$reduce(\otimes) = ((\frac{n}{p}) + log_2 p)t(\otimes)$

The computation quantity of "reduce" follows $log_2 p$ because it is assumed that data structure is going to be a balanced binary tree as list element is mutually associative. For instance, summation of element of list does not affect the order of its calculation.

By the way, summation of inner product can be written by using another interesting higher order function which is $scan$. Scan is a $reduce$ which stores all of elements on its calculation process. For instance,

```
scan (+) 0 [3,2,1,4,5] == [3,5,6,10,15]
```

This is pretty useful when you calculate summation of partial element of array over and over again. For instance, if you want to calculate sums from 2nd element to 4th which is 2+1+4 you can calculate subtract 1st element from 4th element on integral array which is 10 - 3.

This is called integral image in computer vision but useful in parallel programming.

### 1.2.3 Notation with function application

Codes above in Haskell can be translated to more simple way which means not just succession of function which is succeded "$" but nested function application. Result after translation is following.

```
type Weight = [[Double]]
type Node   =  [Double]

--vv a b = foldr (\(x,y) z -> x * y : z) [] $ zip a b
vv = (foldr (\(x,y) z -> x * y : z) [] .) . zip

--vm a b = map (\x ->foldl (+) 0 $ vv a x ) b
vm = map . ((foldl (+) 0 .) . vv)

--feedForward f (Node a) (Weight b) = Node $ map f $ vm a b
feedForward f = (map f .) . vm
```

First, I replaced all of data constructor to the form without any data constructor by "type" which allows me to erase arguments.
Next, I have removed all of arguments of $vv, vm, feedForward$ . How could you assure transformation is correct?
Here is the operation of transformation for $vv$.

```
vv a b = foldr (\(x,y) z -> x * y : z) [] $ zip a b
vv a b = (foldr (\(x,y) z -> x * y : z) [] . zip a) b
```

```
vv a = (foldr (\(x,y) z -> x * y : z) [] . zip a)
vv a = ((foldr (\(x,y) z -> x * y : z) [] .) $ zip a)
vv a = ((foldr (\(x,y) z -> x * y : z) [] .) $ zip a)
vv = ((foldr (\(x,y) z -> x * y : z) [] .) . zip )
vv = (foldr (\(x,y) z -> x * y : z) [] .) . zip
```

Principle idea is an operator ( $ ) can be translated in a following way.

$a \$ b\, c\, d == (a\ .\ b\, c)\, d$

## 1.3 Calculation of error derivatives

### 1.3.1 Error derivatives for the weights on second layer

Error propagation is denoted as following.

Objective function of back propagation is following error minimization function

$$E_{total} = \frac{1}{D}\sum_{d=1}^{D} E_d \tag{1.5}$$

$E_d$ is defined as

$$E_d = \frac{1}{2}\sum_{k=0}^{K}(n_{d2k} - t_{dk})^2 \tag{1.6}$$

where $t_{dk}$ = 1 of k vectors which are orthonormal
The derivatives of each weights are

$$w'_{lij} = w_{lij} - \rho\frac{\partial E_d}{\partial w_{lij}} \tag{1.7}$$

where $\rho$ = learning rate.
$l$ = index of the layer

$\frac{\partial E_d}{\partial w_{lij}}$ means distribution of overall error to every single weights in any layers.

$w_{lij}$ consists of two which is $w_{1ij}$ and $w_{2jk}$ given two layers in the network.

$w_{2jk}$ is able to be unfolded in a following way using chain rule.

$$\frac{\partial E_d}{\partial w_{2ij}} = \frac{\partial E_d}{\partial n_{d2k}}\frac{\partial n_{d2k}}{\partial w_{2jk}} \tag{1.8}$$

$\frac{\partial E_d}{\partial n_{d2k}}$ can be calculated as

$$\frac{\partial E_d}{\partial n_{d2k}} = \frac{\partial \frac{1}{2}\sum_l (n_{d2k} - t_{dk})^2}{\partial n_{d2k}} = n_{d2k} - t_{dk} \tag{1.9}$$

however, $n_{d2k} = \phi y_{d2k}$ needs to be composed to one step smaller elements for calculation of its derivative.

Note that each layers are calculated as sequence of inner product and multiplication of activation function.

$$n_{d2k} = \phi(y_{d2k}) \tag{1.10}$$

where $y_{d2k} = (w_{2jk} + b_j) \cdot n_{d1j}$

thereby, above can be written as

$$\frac{\partial E_d}{\partial w_{2ij}} = \frac{\partial E_d}{\partial n_{d2k}} \frac{\partial n_{d2k}}{\partial \phi y_{d2k}} \frac{\partial \phi y_{d2k}}{\partial w_{2jk}} \tag{1.11}$$

In this condition, $\frac{\partial n_{d2k}}{\partial \phi y_{d2k}}$ and $\frac{\partial \phi y_{d2k}}{\partial w_{2jk}}$ can be calculated as

$$\frac{\partial n_{d2k}}{\partial \phi y_{d2k}} = \frac{\partial \phi_k(y)}{\partial y}|_{y=y_{d2k}} = \phi' \tag{1.12}$$

$$\frac{\partial y_{d2k}}{\partial w_{2jk}} = n_{d1j} \tag{1.13}$$

Above two means derivative of activation function and activation of nodes which is in previous layer.

Substituting (1.9), (1.12), (1.13) to (1.11),

$$\frac{\partial E_d}{\partial w_{2ij}} = (n_{d2k} - t_{dk}) {\phi'}_2 n_{d1j} \tag{1.14}$$

This means weight matrix on the second layer should be updated reflected activation of nodes in previous layer which is connected to the weight and difference between teacher signal and node value and derivative of adopted activation function.

### 1.3.2 Error derivatives for the weights on the first layer

$w1$ is not directly interconnected to output node thereby needs to be propagated to the more forward in a following way.

$$\frac{\partial E_d}{\partial w_{1ij}} = \frac{\partial E_d}{\partial n_{d2k}} \frac{\partial n_{d2k}}{\partial y_{d2k}} \frac{\partial y_{d2k}}{\partial n_{d1j}} \frac{\partial n_{d1j}}{\partial y_{d1j}} \frac{\partial y_{d1j}}{\partial w_{1ij}} \tag{1.15}$$

The first derivative ($\frac{\partial E_d}{\partial n_{d2k}}$) and second derivative ($\frac{\partial n_{d2k}}{\partial y_{d2k}}$) is derivative which have already been calculated by (1.9) and (1.12).

third derivative ($\frac{\partial y_{d2k}}{\partial n_{d1j}}$) is

$$\frac{\partial y_{d2k}}{\partial n_{d1j}} = w_{2jk} \tag{1.16}$$

because

$$y_{d2k} = w_{2jk} \cdot n_{d1j}$$

fourth ($\frac{\partial n_{d1j}}{\partial y_{d1j}}$) is

$$\frac{\partial n_{d1j}}{\partial y_{d1j}} = \phi'_1 \tag{1.17}$$

because

$$n_{d1j} = \phi'_1(y_{d1j})$$

and fifth ($\frac{\partial y_{d1j}}{\partial w_{1ij}}$) is

$$\frac{\partial y_{d1j}}{\partial w_{1ij}} = n_{d0i} \tag{1.18}$$

because

$$y_{d1j} = w_{1ij} \cdot n_{d0i}$$

To feed (1.9),(1.12),(1.16),(1.17),(1.18) into (1.15)

$$\begin{aligned} \frac{\partial E_d}{\partial w_{1ij}} &= (n_{d2k} - t_{dk})\phi'_2 w_{2jk}\phi'_1 n_{d0i} \\ &= ((n_{d2k} - t_{dk})\phi'_2 \cdot w_{2jk})\phi'_1 n_{d0i} \end{aligned} \tag{1.19}$$

$(n_{d2k} - t_{dk})\phi'_2 \cdot w_{2jk}$ in (1.19) means propagating error to the backward passing through the same weight which have been made use of in forward pass.

### 1.3.3 Interpretation of error derivative calculation

(1,14) and (1,19) tells that derivative of error with respect to each weights is composed by following.

- derivative of activation function given firings of its neurons in feed-forward process
- one step forward values of neurons
- accumulated errors which has propagated from backwards layer.

These deduce following statements.

Letting forward node of $w_{lfb}$ $N_{df}$ , backward node $N_{db}$

**Punishment towards the excitement of input** $N_{df}$
if more excited the activation of the input value to the weight is ( absolute value of neuron is larger ), more error would be propagated.

**Punishment towards the amount of error propagation** $N_{db}$
if the amount of error propagated from backward is larger, updating values would be larger.

### 1.3.4 Derivative of sigmoid function

By the way, so as for network weights to hold every information for nonlinear mapping between data and its teacher label, they have to have a functionality which saves patterns which have already learned when new data is fed in. I call this anti-plasticity because it prevents violating acquired mapping from another training data. This functionality appears in back propagation algorithm specifically derivative of activation function.

Suppose activation function is sigmoid function, let us take a look at its character.

$$S(u) = \frac{1}{1 + \exp(-u)} \tag{1.20}$$

Its derivative is calculated as following.

$$\begin{aligned}
S'(u) &= \frac{dS}{du} \\
&= \frac{d}{du}(1 + \mathrm{e}^{-u})^{-1} \\
&= -1(1 + \mathrm{e}^{-u})^{-2}\frac{d\mathrm{e}^{-u}}{du} \\
&= -1(1 + \mathrm{e}^{-u})^{-2}(-\mathrm{e}^{-u}) \\
&= \frac{\mathrm{e}^{-u}}{(1 + \mathrm{e}^{-u})^2} \\
&= \frac{1 + \mathrm{e}^{-u} - 1}{(1 + \mathrm{e}^{-u})^2} \\
&= \frac{1}{1 + \mathrm{e}^{-u}}(1 - \frac{1}{1 + \mathrm{e}^{-u}}) \\
&= S(u)(1 - S(u))
\end{aligned}$$

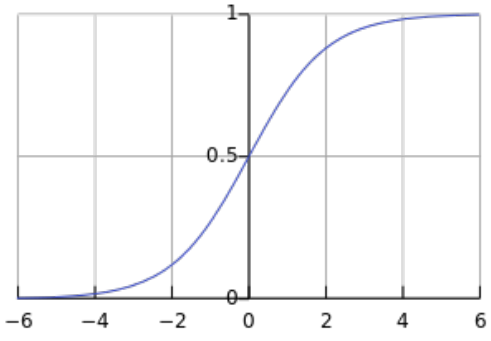


FIGURE 1.1: sigmoid function

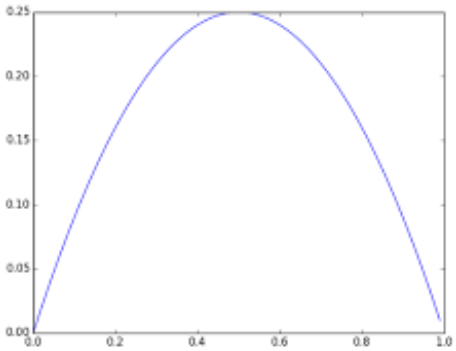


FIGURE 1.2: derivative of sigmoid

Figure 1 represents mapping by sigmoid function. Figure 2 represents mapping via derivative of sigmoid function.

It tells you when you let activation function sigmoid, all of the values of node have values from 0 to 1, and its derivatives are maximized when the value is 0.5. If you note that derivative of activation is calculated feeding into the values of neurons in feed-forward step, it says exactly about anti-plasticity. If one firing of neuron is close to 0 or 1, the error is less likely to be propagated to the node than a node whose value is close to 0.5. In other words, if the activation of neurons are already fixed by past training regardless

of current input values, errors are not be propagated to the node, rather flowed to the another node whose value have never been fixed.hyperbolic tangent has similar property that sigmoid has, the firing of neurons can be negative in this case, though.

To summarise them, Back propagation algorithm can be divided into two parts, which is

**Plasticity** $N_{df}$
: if more excited the activation of the input value to the weight is ( absolute value of neuron is larger ), more error would be propagated.

**Anti-plasticity** $N_{db}$
: if the amount of error propagated from backward is larger, updating values would be larger.

### 1.3.5 Code representation

```
--activation function
sigmoid :: Double -> Double
devSig  :: Double -> Double

--calcErrors input : (Derivative of activation function,
--Teacher node , Output node, Output error)
calcErrors :: (Double -> Double) -> Node -> Node -> Node

--sum of square errors
calcErrorSigma :: Node ->  Double

--backpropagate error (Derivative of activation function, Node,
--Node, Weight between 1st & 2nd layer, hiddenError)
hiddenError  :: (Double -> Double) -> Node -> Node -> Weight -> Node

--weight update (learning rate, input node, output node,
--weight before update, weight after update)
--I let the node two dimensional to multiply element wise multiplication
adjustWeight :: Double -> [Node] -> [Node] -> Weight -> Weight

--square mean of error in final layer
calcErrorSigma = (foldl (+) 0 . map (\x->x*x))

--backpropagation to hidden layer
hiddenError f h = ((foldr (\(x,y) z-> x*y:z) [] . zip (map f h)) . ) . vm

--weight adjustment in two layer
adjustWeight lr = (map2 (map2 (+)) . ) . dot'

sigmoid x = 1 / (1+exp(-x))
devSig x  = x * (1 - x)

--map2
map2 :: (a -> b -> c) -> [a] -> [b] -> [c]
map2 _  a []= []
map2 _ [] a = []
map2 f (h1:t1) (h2:t2) = f h1 h2 : map2 f t1 t2

-- matrix inner product.
-- more proper than vv and vm which is defined above
-- because vector is regarded as two dimensional matrix.
-- This is necessary to calculate (vector.T dot vector -> matrix)
dot' :: (Num a) => [[a]] -> [[a]] -> [[a]]
dot' a b = group (length a) ( map ( foldl (+) 0 )
  [ (map2 (*) x y)  | x <- a, y <- t])
```

```
  where t = t' b

--below is functions which is called from above.

--transposition of matrix
t' :: [[a]] -> [[a]]
t' ([]:_) = []
t' x = (map head x) : t' (map tail x)

--take some elements before index
take' :: Int -> [a] -> [a]
take' a [] = []
take' 0 a  = []
take' i (h:t) = h : (take' (i-1) t)

take'' :: Int -> [a] -> [a]
take'' a l = reverse
$ foldl (\x y -> if length x < 2 then y : x else x) [] l

--drop some elements after index
drop' :: Int -> [a] -> [a]
drop' 0 a  = a
drop' i (h:t) = drop' (i-1) t

--group is combination of take & drop
group :: Int -> [a] -> [[a]]
group  _ [] = []
group i t = take' i t : group i (drop' i t)
```

```
dot' a b = group (length a) ( map ( foldl (+) 0 ) [ (map2 (*) x y)  | x <- a, y <- t])

 where t = t' b
```

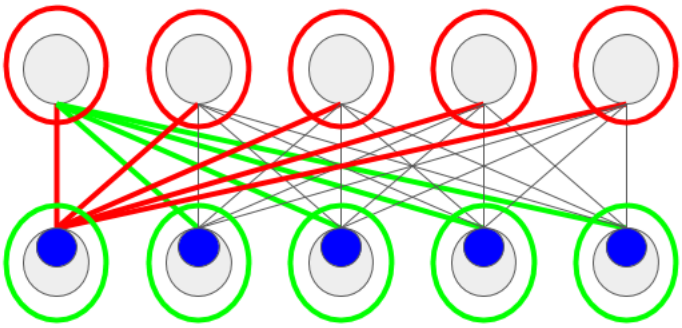

FIGURE 1.3: dot product and feed-forward

# Chapter 2

# Convolutional Neural Network

## 2.1 Prelude

In this section, I will introduce convolutional neural network. As we had a look at in the last chapter, every nodes of multi-layer perceptron is interconnecting to the every nodes of next layer. It states that each dimension of multi-layer perceptron should not have any local dependencies in it, in other words, asked to be mutually independent. This is why unsupervised learning algorithm for dimensional reduction is adopted before feeding data into multi-layer perceptron.

CNN is different from multi-layer perceptron in a sense that it is in charge of dimensional reduction as well as classification. Concretely, CNN sets a constraint towards connection between values on a dimension in a data and another dimension in the same data. If the distance between two dimensions are longer than certain values, the connection in between them are out of scope that network takes into its consideration. The maximum distance is determined by filter size which is one of its parameters that one can pick up by themselves. For instance, if the filter size is just 1, it means there is no interconnection between each dimensions. If the number of dimensions are as many as length of input dimension, it means all of dimensions interconnects to dimensions on a subsequent layer just as the case of multi-layer perceptron.

In the explanation of CNN, I will introduce another type of convolution operation, which is so called inner-product based convolution.

Last but not least in this chapter, I will introduce another way of calculation to compute binary neural networks based on inner-product based convolution. Binary neural networks is the way to train neural network classifier aiming both reduction of training computational cost and retaining good classification. One of the algorithm to calculate binary neural net is XNORNet. My approach is slightly different from the computation which is presented in this paper.

## 2.2 Drawback of training in more than 4 layers perceptron

For some data-set, CNN which has more than 2 hidden layers shows better performance than the one whose layer is just 1. However from following reason, it is difficult to propagate every weights as it should be by back-propagation if activation function is sigmoid. This is called vanishing gradient problem on error propagation step. Namely, error derivative is going to be smaller if the node is closer to forward. This is explained as following.

Error propagation step to the first weight $w_{1ij}$ had be written as following.

$$\frac{\partial E_d}{\partial w_{1ij}} = (n_{d2k} - t_{dk}){\phi'}_2 w_{2jk} {\phi'}_1 n_{d0i} \tag{2.1}$$

If the network has more than 4 layers, error propagation to the 1st weight can be written as

$$\frac{\partial E_d}{\partial w_{1ij}} = (n_{d3l} - t_{dl}){\phi'}_3 w_{3kl} {\phi'}_2 w_{2jk} {\phi'}_1 n_{d0i} \tag{2.2}$$

This tells you backward process is completely linear with respect to the error of the last layer. Besides,

$$\phi = \frac{1}{1 + \exp(-u)}$$
$$\phi'(u) = \phi(u)(1 - \phi(u))$$
$$0 < u < 1$$
$$0 < \phi'(u) < 0.25$$

When derivative of error in last layer come back to the forward layer, error is going to be reduced by passing through multiplication of derivative of multiplication.

Next graph shows you how weights are updated in four layer perceptron in each layer.

To summarise it, normal back-propagation with more than 4 layers is trained, forward weight cannot be well tuned.

This gives you a motivation to introduce different activation function which is rectified linear unit which is introduced later on.

## 2.3 Feed-forward process of convolutional neural network

In this section, I will explain feed-forward process of CNN. Main architecture of Feed-forward process of CNN consists of 4, linear combination of convolution, activation function, pooling, and fully connected layers. There are many types of CNN which is modified this mainstream. In this chapter, I will have a look at the ones except fully connected layer because it is identical with multilayer perceptron in the previous chapter.

### 2.3.1 1D Feed-forward summation of convolution

CNN has different architecture based on dimension of input data contrary to multilayer perceptron. Here I will have a look at 1 dimensional and 2 dimensional CNN respectively.

Feed-forward process of convolution network operation is defined by following which is 1st convolution and 2nd, summing up all of corresponding elements.

$$b_{djn} = \sum_{i=0}^{I-1} \sum_{s=0}^{S-1} a_{di(s+n)} w_{ijs} \tag{2.3}$$

where $S$ = kernel size
$s \in S$
$n \in N$
$N$ = dimension of data
$I$ = number of input feature
$i \in I$
$j \in J$
$J$ = number of hidden feature
$a_{di(s+n)}$ = input data set
$b_{djn}$ = a feature in hidden layer

$w_{ijs}$ = weight

First of all, dimensions of nodes and weights are incremented just one up than the ones for multilayer perceptron, namely, two dimension for nodes, three dimension for weights.

Specifically, regarding nodes, one hidden node consists of smaller unit of nodes whose dimension is the same of original data or dimension - 2 (assuming no boundary compensation) and there are multiple hidden nodes. Here, I call larger unit of hidden node as feature, and smaller unit of node an individual hidden node. For instance, $b_{1,5,3}$ of $b_{djn}$ means a hidden node in the 1st data of data-set, in the 5th feature, and 3rd value on the data.

Weights between one layer and subsequent one has also three distinct index. $w_{1,6,3}$ of $w_{ijs}$ means weights which connects 1st feature in previous layer and 6th feature in subsequent layer, and the 3rd value on the filter.

Input feature in a first layer is mostly just single unless it has multiple dimension originally such as RGB value on image data, thereby first hidden layer does not sum up multiple firing of neurons but just supply multiple features as they are.

### 2.3.2 Code representation

If you implement linear combination of convolution straightforwardly in Haskell, the code is going to be like following.

```
--argument
  --1st : activation function
  --2nd : previous node
  --3rd : weights in between
  --return : Node on next layers
  feedForward1D :: (Double->Double) -> Node1D -> Weight1D -> Node1D

  --second map from left will iterate different features on same nodes,
  --then, rightest map2 puts pairs of input and weights on convolution calculation,
  --then, the matrix is transposed so it can be added by element-wise.
  feedForward1D f n = map $ map g . t' . map2 convolve1D n
     where g = f . foldl (+) 0
```

This operation is overall scheme of summation of convolution which have passed through activation function.
"map2 convolve1D n" calculates convolution with input node and weights on each filter. They are going to be transposed because we would like to calculate element wise addition in each dimension. After the transposition, each dimension is going to be added together. Most outside "map" operation is the distribution to the next feature from one of the features on input node.

convolve1D is defined as following, and its explanation is described by figure 2.1

```
--1 dimensional convolution
  convolve1D :: [Double] -> Kernel1D -> [Double]

  --apply boundry compensation to the top of list and the bottom of list
  --iterate 1d list
  boundry1D :: Int -> [Double] -> [Double]

  --take first element and insert on top of the list
  --iterate 1d list
  filling1D  :: Int -> [Double] -> [Double]
```

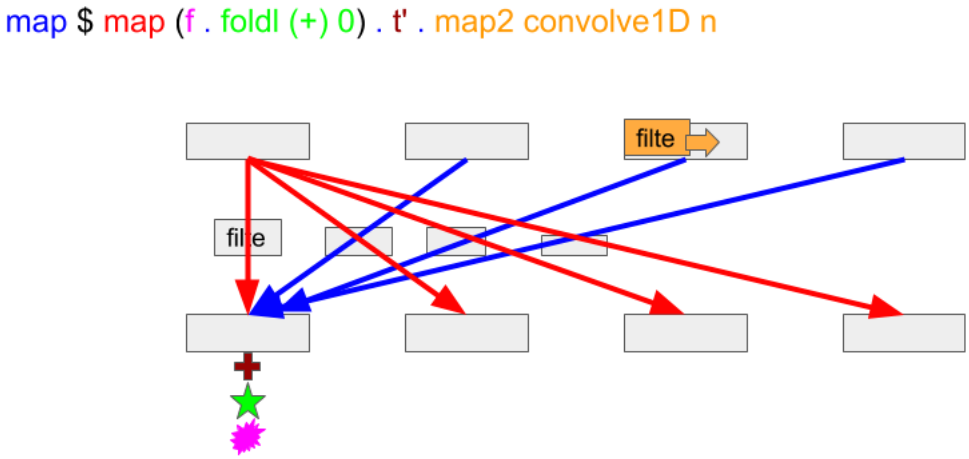


FIGURE 2.1: One dimensional CNN feed-forward process

```
--this function is just insertion N value on top of list
-- N is first argument
filling'    :: Int -> Double -> [Double]

--actual operation of convolution
convolve1D' :: [Double] -> Kernel1D -> [Double]

convolve1D d k = convolve1D' dd k
-- get the half of kernel size to insert boundry compensation
-- for convolution beforehand
  where s = (length k) `div` 2
-- apply boundry compensation
        dd = boundry1D s d


boundry1D s a = dt
--boundry compensation here is taking nearest elements on the list,
--and insert it to the top and bottom of the list.
  where r  = (reverse . filling1D s . reverse) a
        dt = filling1D s r


filling1D s a = b
--take first element and insert on top of given list
  where rh = filling' s (head a)
        b  = (++) rh a

convolve1D' d k
--if the size of remained vector is less than kernel size, stop convolution.
  | lk > ld   = []
--convolution is done by taking first N element from the list
--and multiply with kernel and then sum up
--first N element is kernel size
  | otherwise = foldl (+) 0 ( map2 (*) k (take' lk d)) : (convolve1D' (tail d) k)
  where ld = length d
        lk = length k
```

That is, input nodes are going to be scaled bigger according to the size of kernel. If the kernel size is 3,

```
boundary1D sh st a = filling1D sh ((reverse . filling1D st . reverse) a)

filling1D s a = (++) (filling' s (head a)) a

filling' i a = a : filling' (i-1) a

convolve1D' = foldl (+) 0 ( map2 (*) k (take' lk d)) : (convolve1D' (tail d) k)

convolve1D d k = convolve1D' (boundary1D s s d) k  ///  where s = (length k) `div` 2
```

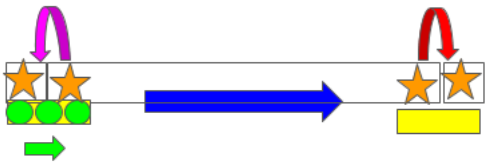

FIGURE 2.2: One dimensional convolution (boundary compensation)

you need to fill a value on the top and the bottom of the list.

### 2.3.3 Inner product based feed-forward

If the kernel size is 1, it can be intuitively understood that convolution operation can be regarded as inner product in each dimension, and the number of the procedure is equivalent to number of dimension. This is because summation of convolution can be omitted because kernel is single. This is useful to compare normal multilayer perceptron and CNN because feed-forward process of multilayer perceptron is successive application of inner product and nonlinear activation function. Different point over them is multi-layer perceptron has inter connection over different dimensions. However, convolution whose kernel is just 1 does not have any interconnection over multiple dimension at all.

If the kernel size is more than 2, convolution operation can be seen as summation of inner product over different features. The reason is following. Convolution of more than 2 kernel size can be seen as linear combination of convolution operation whose kernel size is just 1. As I explained before, convolution operation for single kernel can be seen as inner product. In addition, summation for convolution and summation for over multi-features after each convolution can be commutative. Therefore, convolution operation in this case is linear combination of inner product. Mathematically, the logic is described as following

$$
\begin{aligned}
b_{djn} &= \sum_{i=0}^{I-1} \sum_{s=0}^{S-1} a_{di(s+n)} w_{ijs} \\
&= \sum_{s=0}^{S-1} \sum_{i=0}^{I-1} a_{di(s+n)} w_{ijs} \\
&= \sum_{s=0}^{S-1} (a_{di(s+n)} \cdot w_{ijs})
\end{aligned}
$$

(2.5)

That means, instead of convolution with filters, different input features ($a_{di}$) is added together after letting them pass through multiplication of different weights, then, they are summed up according to the size of kernel. I name this operation inner product based feed-forward.

Algorithmically, however, the operation is not going to be combinations of simple inner products because this is an operation for two dimensional matrix and three dimensional matrix. The process can be interpreted as following figure (2) and (3).

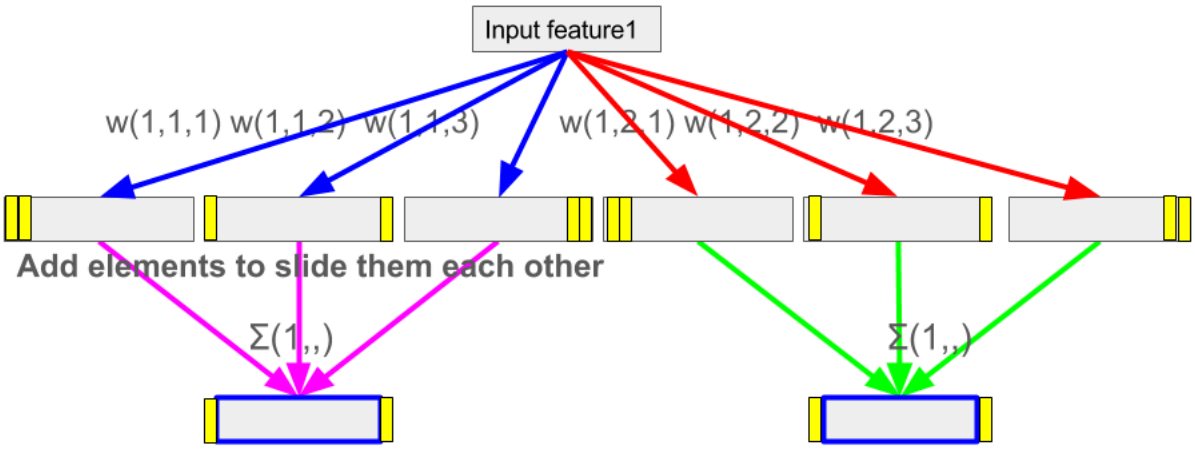


FIGURE 2.3: one dimensional CNN feed-forward process

Figure 2 explains the process of distribution from one feature to multiple features on the next layer. Figure 3 explains flow of computation in case there are multiple input features.

Inner based feed-forward operation can be distinct with two operation. First is multiplication and summation over multi-dimensions. Second is another summation enlarging input features according to the size of filters, and let the size as it was in the end, namely cutting out center of it.

First operation consists of four in terms of its necessary iterations.
First operation is iterating one input feature with one of weights on a filter. Then, second process is repeating first operation setting the weights different on the same filter. Third operation is iterating different input features over previous operation. And fourth is distributing features for the next features over every operations till 3rd step. To summarize it,

1. iterate on single feature and multiply with given single weight.
2. iterate weights on a filter
3. iterate multiple input features with different weights
4. iterate output features ( or distribute features for the next layer)

Note that every step has own finite loop and certain conditions to go to the next step. This operation is not sequential rather nested.

After the end of third step, every corresponding elements of list which acquired from 1st step should be added with corresponding elements of list in third dimension.

For instance,

given

[[ [1,2,3,4,5] [2,3,4,5,6]], [[-1,-2,-3,-4,-5] [1,2,2,2,1] ]]

you should get

[1-1,2-2,3-3,4-4,5-5],[2+1,3+2,4+2,5+2,6+1]

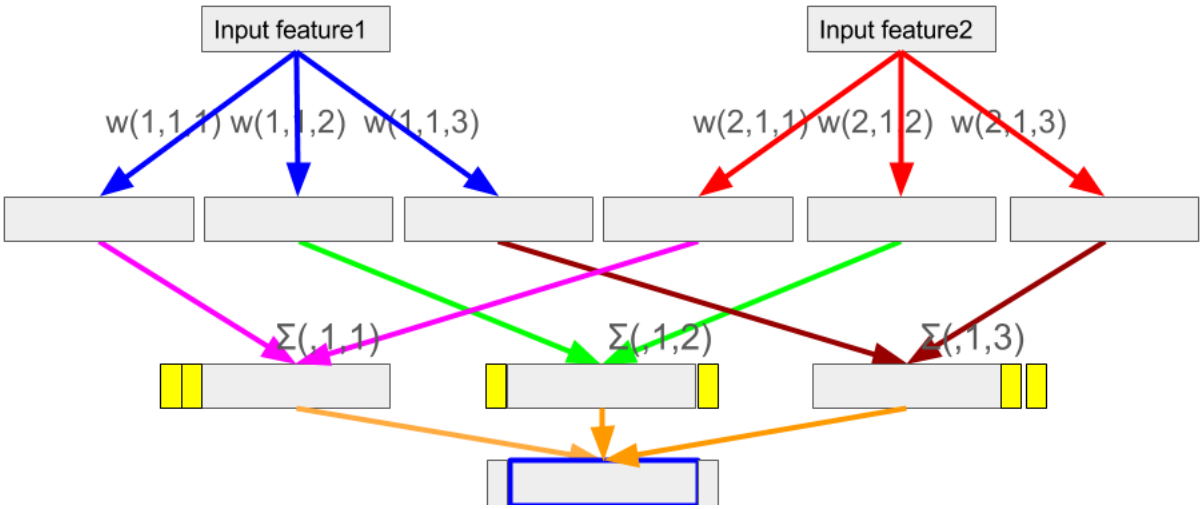


FIGURE 2.4: one dimensional CNN feed-forward process

Algorithmically, first operation is represented as following.

```
--feedForward in a different way,
  feedForward1D' :: Node1D -> Weight1D -> Node1D

--sub function of
  feedForward1D''  :: Node1D -> Weight1D -> [Node1D]

  feedForward1D' n w = feedForward1D''' (feedForward1D'' n w) k
     where k = (length . head . head) w

  feedForward1D'' n2 w4 = map (\w3 -> group k
     . map (foldl (+) 0) . t' . map (concat)
     $ map2 (\w2 n1 -> map (\w1 -> map (*w1) n1) w2) w3 n2) w4
     where k = (length . head) n2
```

Figure(4) explains correspodence of nested structure.

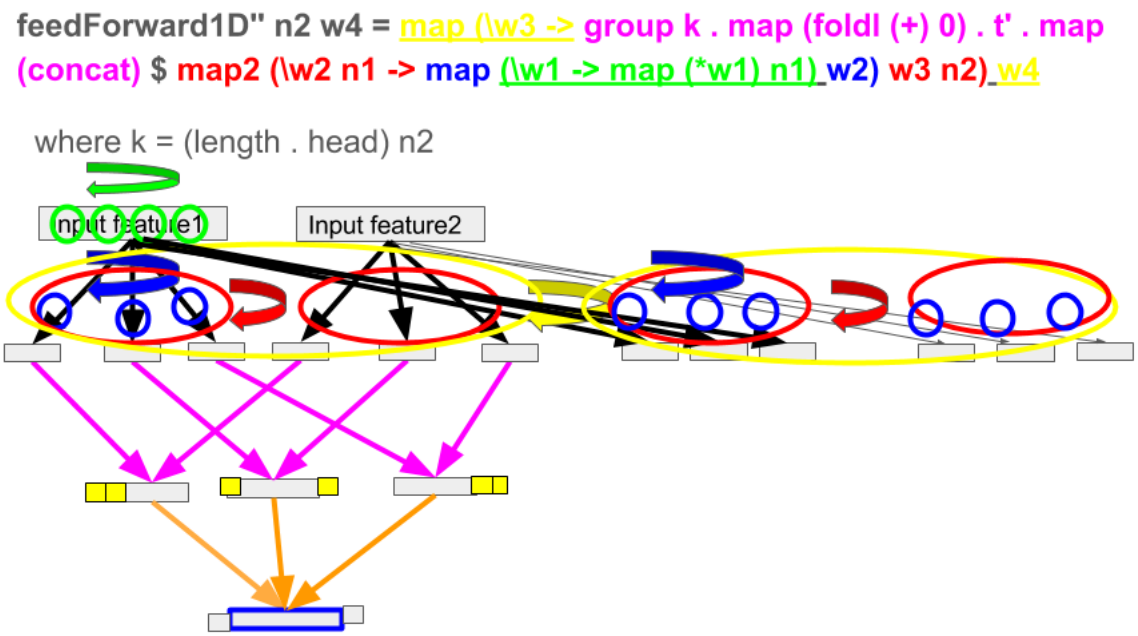


FIGURE 2.5: first part of CNN feed-forward (altered version)

One thing which is not self-explanatory is summation over different features.

Element-wise summation can be written as combination of "foldl (+) 0 " and "t'"(transposition). In this case, it is element-wise multiplication, but elements of 1st dimension and 3rd dimension, not the adjacent dimension. Thereby, you need to insert concatenation of list before transposition and grouping after summation.

```
feedForward1D''' :: [Node1D] -> KernelSize -> Node1D


feedForward1D''' n k = map (\x-> pickUp x kk kk) s
  where kk = k `div` 2
        s  = map (\x -> map (foldl (+) 0) . t'
        $ map2 (\(h,t) xx-> boundry1D t h xx)
        (zip [0..2*kk] (reverse [0..2*kk])) x ) n
```

Second step of inner product based feed-forward is summation over output of first operation according to the number of weights on single filter. All of values which multiplied with weights on single filter should be added together.
For instance, given you have output from first operation.

$$\begin{pmatrix} 0.1 & -0.2 & -0.2 & 0.3 & 0.1 & 0.3 & 0.1 & -0.3 \\ 0.2 & -0.3 & -1 & -0.4 & 0.5 & 0.0 & 0.1 & -0.1 \\ -0.2 & -0.2 & -0.4 & 0.2 & 0.3 & -0.1 & 0.2 & 0.4 \end{pmatrix}$$
$$\begin{pmatrix} -0.2 & -0.4 & -0.5 & 0.1 & 0.6 & 0.2 & -0.1 & 0.3 \\ 0.4 & -0.1 & -0.2 & -0.4 & 0.5 & 0.4 & 0.1 & 0.2 \\ -0.6 & -0.6 & -0.1 & 0.2 & 0.3 & 0.1 & -0.2 & 0.1 \end{pmatrix}$$

and filter is

[ [0.2,-0.1,0.3],[-0.1,0.2,-0.1] ],[ [-0.2,0.3,0.1],[0.2,-0.1,0.4] ]

Column of matrix corresponds dimensions of data.
Row of matrix represents Nth of weights on a filter which are added.
And, third dimension is the result of Nth filters.(N is numbers of filter.)

Column of the matrix is enlarged by following.

$$\begin{pmatrix} 0.1 & 0.1 & 0.1 & -0.2 & -0.2 & 0.3 & 0.1 & 0.3 & 0.1 & -0.3 \\ 0.2 & 0.2 & -0.3 & -1 & -0.4 & 0.5 & 0.0 & 0.1 & -0.1 & -0.1 \\ -0.2 & -0.2 & -0.4 & 0.2 & 0.3 & -0.1 & 0.2 & 0.4 & 0.4 & 0.4 \end{pmatrix}$$

2 dimensions are added in such a manner. 2 reflects the fact that filter size is 3. And, element whose value is equal to adjacent element is added either top of the list or bottom of the list. This compensation can be done in other ways such as filling "0" and so on. The position of inserting elements is determined by the row of matrix. If the list is in first row, it means these values are outcomes after multiplying Nth weights on a filter. Come to think of convolution process, multiplication between last element of a data and first weights on a kernel is discarded or simply not calculated. By contrast, elements of "-1"th element and first weights on a filter is used. It suggests that same calculation can be done by sliding elements of list, sums them up, and retrieve central part.For instance, the result of multiplication between last element means values of (row,column) = (1,10) which does not need to be calculated. Instead, -1 nth values on the first list can be represented as the values of (row, column) = (1,1) which should be included as final result.

Concretely, edge of above matrix is cut leaving center

$$\begin{pmatrix} 0.1 & 0.1 & -0.2 & -0.2 & 0.3 & 0.1 & 0.3 & 0.1 \\ 0.2 & -0.3 & -1 & -0.4 & 0.5 & 0.0 & 0.1 & -0.1 \\ -0.2 & -0.4 & 0.2 & 0.3 & -0.1 & 0.2 & 0.4 & 0.4 \end{pmatrix}$$

and, added by element-wisely.

$$\begin{pmatrix} 0.1 & -0.6 & -0.3 & -1.0 & 0.7 & 0.3 & 0.8 & 0.4 \end{pmatrix}$$

```
feedForward1D''' n k = map (\x-> pickUp x kk kk) s

where s  = map (\x -> map (foldl (+) 0) . t' $ map2 (\(h,t) xx-> boundry1D t h
xx) (zip [0..2*kk] (reverse [0..2*kk])) x ) n
```

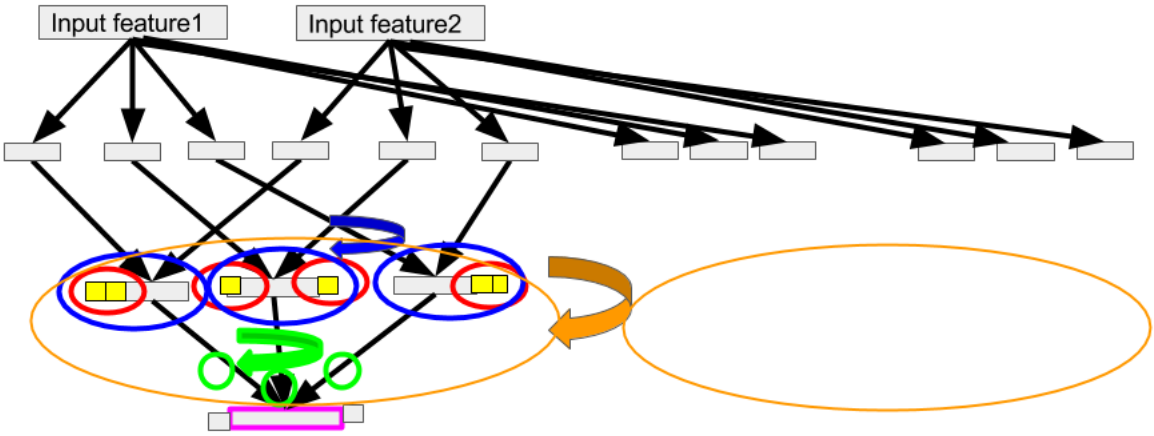


FIGURE 2.6: second part of CNN feed-forward (altered version)

### 2.3.4 Two dimensional linear summation of feed-forward

Two dimensional feed-forward process of convolution can be written as following.

$$b_{djmn} = \sum_{i=0}^{I-1} \sum_{v=0}^{V-1} \sum_{h=0}^{H-1} a_{di(m+v)(n+h)} w_{ijvh} \tag{2.6}$$

where $V$ = kernel vertical size
$v \in V$
$H$ = kernel vertical size
$h \in H$
$m \in M$
$M$ = vertical dimension of data
$n \in N$
$N$ = horizontal dimension of data
$I$ = number of input feature
$i \in I$
$j \in J$
$J$ = number of hidden feature
$a_{di(m+v)(n+h)}$ = input data set
$b_{djmn}$ = a feature in hidden layer
$w_{ijvh}$ = weight

In 2D convolutional neural network, dimension of node and weight is incremented from 1D convolutional neural network.

Node is 4 dimensional whose dimension is position of data-set, position of feature, vertical position of data, horizontal position of data .

Weight is also defined by 4 dimension which is position of feature in previous layer for one vertex,position of feature in subsequent layer for another vertex, vertical position on kernel, horizontal position on kernel

### 2.3.5 code representation of two dimensional linear summation of convolution

```
--dimension is one higher than 1D
feedForward2D :: Node2D -> Weight2D -> Node2D

--only difference between 1D and 2D is two.
--One is convolving by 2D and another is covering over it
--with one more "map"
feedForward2D n = map $ map g . t' . map2 convolve2D n
   where g = map (foldl (+) 0)

--the way not to use signature "$" is following
--feedForward2D f n = map (map (map (f . foldl (+) 0))
  . t' . map2 (convolve2D) n)
```

Convolution for two dimension can be written as following.

```
--2 dimensional convolution
convolve2D  :: [[Double]] -> Kernel2D -> [[Double]]

--apply boundry compensation to the top of list and the bottom of list
--iterate 2d list
boundry2D :: Int -> Int -> [[Double]] -> [[Double]]

--take first element and insert on top of the list
--iterate 2d list
```

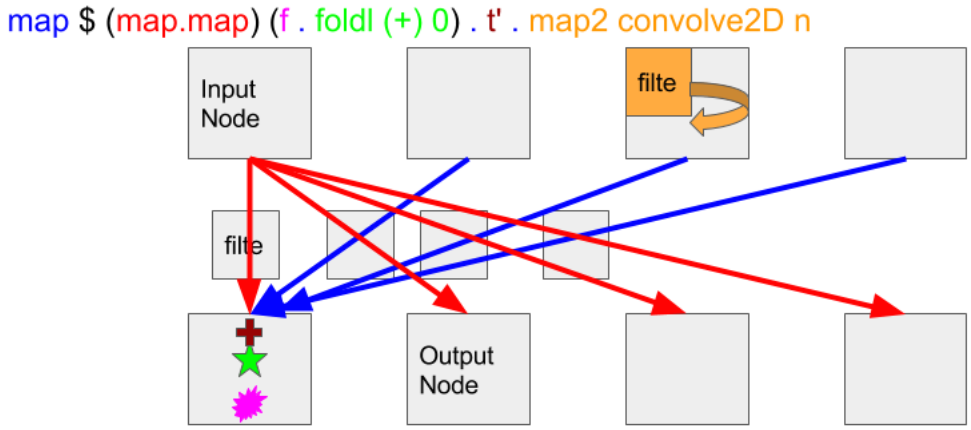


FIGURE 2.7: two dimensional CNN feed-forward

```
filling2D  :: Int -> [[Double]] -> [[Double]]

--actual operation of 2d convolution
convolve2D'  :: [[Double]] -> Kernel2D -> [[Double]]
convolve2D'' :: [[Double]] -> Kernel2D ->  [Double]

convolve2D d k = convolve2D' dd k
--for 2d convolution, boundry compensation have to be done
--twice for rows and columns.
  where s  = (length k) `div` 2
        dd = (boundry2D s s . boundry2D s s) d


boundry2D hs ts a = dt
--boundry2d is identical to boundry 1d except input list is 2d
  where r  = (map (reverse) . filling2D ts . map (reverse)) a
        dt = (t' . filling2D hs) r


filling2D s a = b
--filling1d is identical to filling 1d except input list is 2d
  where rh = map (\x -> filling' s (head x)) a
        b  = map2 (++) rh a

--according to the kernel size, you fill up nearest values as window.
--filling' 3 [2,3,4,5,6] -> [2,2,2]
filling' 0 _ = []
filling' i a = a : filling' (i-1) a

--this operation is to iterate different rows
convolve2D' d k
--if number of rest element is less than kernel size, stop
  | lk > ld   = []
--apply column convolution to the each list and append them
  | otherwise = (convolve2D'' d k) : (convolve2D' (tail d) k)
  where lk = length k
        ld = length d

--this operation is to iterate different columns
```

```
convolve2D'' d k
--if number of rest element is less than kernel size, stop
  | lk > ld    = []
--actual convolution operation.
--two dimensional list is concatenated to 1d and then,
--elementwise multiplication is done.
--finally, sums up everything.
  | otherwise = foldl (+) 0 ( map2 (*) (concat k)
  (concatMap (take' lk) d) ) :(convolve2D'' (map (tail) d) k)
  where lk = length $ head k
        ld = length $ head d
```

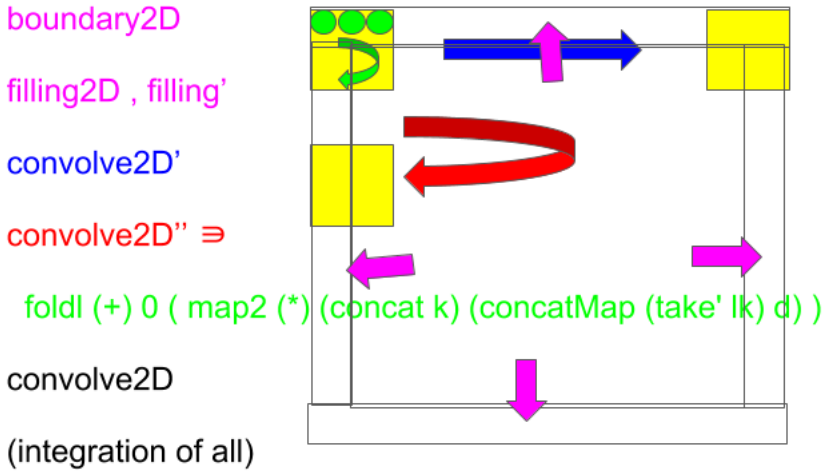


FIGURE 2.8: Two dimensional convolution (boundary compensation)

### 2.3.6 Activation function

Activation function for one dimensional case can be written as

$$c_{djn} = \phi b_{djn} \tag{2.7}$$

Note that activation function is applied after the summation of convolved values which is exactly identical to the case for multi-layer perceptron.

Most often used activation function for CNN is rectified linear unit ( Relu ) which is defined as follows.

$$Relu(x) = 0 \ \ (x \leq 0)$$

$$Relu(x) = x \ \ (x > 0) \tag{2.8}$$

Note that this activation function does not have any squashing output values under "1" unlike sigmoid or hyperbolic tangent.
The good feature of Relu is explained in the section of its derivative which comes later on.

### 2.3.7 Pooling layer

Convolution step and activation nonlinear function can be followed by pooling layer more than often not. Pooling layer is defined by following. There are some way to calculate pooling. Most frequently used pooling is max pooling, which is defined as following.

$$d_p = \max\left(c_{(lp+s)}\right) \tag{2.9}$$

where $s \in [0,\, l]$
$p$ output dimension
$l$ reduction scale

The reason that max pooling is an indispensable tool for convolutional neural network is acquiring different scaling of the data.

```
--argument
  --1st : node
  --return : node
  maxPooling1D  :: Node1D -> Node1D

  --argument
  --1st : pooling size (default 2)
  --2nd : node
  --return : node
  maxPooling1D' :: Int -> Node1D -> Node1D

  --same with 1D
  maxPooling2D  :: Node2D -> Node2D

  --same with 1D
  maxPooling2D' :: Int -> Node2D -> Node2D

  --assume pooling size is 2
  maxPooling1D = maxPooling1D' 2

  --pooling is done by following.
  --first group list with every N elements such as (N==2) , [2,3,4,5]-> [[2,3],[4,5]]
  --then, calculating max value among them
  maxPooling1D' s = map (map (foldr max 0) . group s)

  --assume pooling size is 2
  maxPooling2D = maxPooling2D' 2

  --only difference between 1D and 2D pooling is again two.
  --One is grouping by 2D and another is covering over it with one more "map"
  maxPooling2D' s = map (map (map (foldr max 0)) . group2D s)
```

After successive operation of multiple convolution layer and pooling layer, there is a case that full connected layer is stack in the end of the network. Formalization of this process is omitted since it is identical

to the process of normal multi layer perceptron forward process. Its necessity and role is suppose to be discussed in further chapter.

## 2.4 Calculation of error derivatives

### 2.4.1 Difference between multilayer perceptron and back-propagation

What is different between normal multilayer perceptron and CNN is dimension of its weight. The former has 2 dimensions at most. Latter has either 3 (in case for 1 dimensional CNN) or 4 (in case for 2 dimensional CNN). Why is the dimension of CNN incremented ? That is because CNN sets a constraint towards connection of different dimension, on the other hand, normal multilayer perceptron does not have any constraints about inter-connectivity between different dimension.

In the chapter 1, objective function of BP for normal multilayer perceptron was described as following.

$$\frac{\partial E_d}{\partial w_{lij}} \tag{2.10}$$

where $l$ the number of layer
$i$ the index of hidden(input) neurons
$j$ the index of output(hidden) neurons

Practically, weights are two dimensional if you separate them layer by layer,
Objective function of BP for 1D convolutional neural network is

$$\frac{\partial E_d}{\partial w_{lijn}} \tag{2.11}$$

where $l$ = the number of layer
$i$ = the index on hidden(input) neurons
$j$ = the index on output(hidden) neurons
$n$ = the index on the kernel

### 2.4.2 Error derivatives of element-wise summation after convolution

As I explained in the feed-forward process, there are 4 types of operation in CNN, convolution followed by summation, activation function, pooling, and fully connected inner product operation. I will explain those of three except fully connected inner product which is showed in the last chapter.

Convolution was written as following.

$$b_{djn} = \sum_{i=0}^{I-1} \sum_{s=0}^{S-1} a_{di(s+n)} w_{ijs} \tag{3}$$

Derivative of element-wise summation after convolution can be written as following.

$$\frac{\partial b_d}{\partial a_{di(s+n)}} = \sum_{i=0}^{I-1} \sum_{s=0}^{S-1} b_{dj(s+n)} w_{ijs} \tag{2.12}$$

Which means, to put it simply, derivative of an element-wise summation after convolution is an element-wise summation of convolution.
With more details, product between input node and weights have replace to the product between output node and weights. The reason that derivative of convolution is going to be so simple and symmetric is the fact that convolution operation is composed by linear combination of weights and neurons of forward layer as well as inner product. That makes sense that its derivative is going to be identical to feed-forward layer.

### 2.4.3 Derivatives of activation function

Derivatives of activation function is up to adopted nonlinear function.

As of most frequently used activation function, rectified linear unit, its derivative can be as following.

$$\frac{dRelu(x)}{dx} = \begin{cases} 0 & (x \leq 0) \\ 1 & (x > 0) \end{cases} \tag{2.13}$$

Which means, there is no error propagation if an input value is negative. If an input value is positive, just let it pass through after multiplying with value of firing of nodes.

Applying Relu overcomes vanishing gradient problems in comparison with sigmoid function because it just let input pass through when the derivative is positive.

### 2.4.4 Derivatives of pooling

Derivative of pooling derives from an adopted pooling operation. If it is max-pooling, it is described as following.

$$\frac{\partial Pooling}{\partial x_{(li+s)(lj+t)}} = \begin{cases} \frac{\partial Pooling}{\partial a_{ij}} & if\ a_{ij} = x_{(li+s)(lj+t)} \\ 0 & otherwise \end{cases} \tag{2.14}$$

That means, simply you propagate errors to the node which only has max value and neglect rest of it.

### 2.4.5 Distribution of overall error derivatives with respect to each errors

To sum them up, what is going to be the distribution of errors to the each weights on different layers and layers ?

The overall scheme of back propagation is similar to the propagation of multilayer perceptron. From chapter 1, assignment of overall errors was written as following.

$$\frac{\partial E_d}{\partial w_{1ij}} = \frac{\partial E_d}{\partial n_{d2k}} \frac{\partial n_{d2k}}{\partial y_{d2k}} \frac{\partial y_{d2k}}{\partial n_{d1j}} \frac{\partial n_{d1j}}{\partial y_{d1j}} \frac{\partial y_{d1j}}{\partial w_{1ij}} \tag{4}$$

Given that, let us assume we have simple CNN whose composition operations are, in order, convolution with element-wise summation, activation function, pooling, fully-connected layer. Distribution of error to a weight on the first layer can be written as following.

$$\frac{\partial E_d}{\partial w_{1ijs}} = \underline{\frac{\partial E_d}{\partial y_{df}} \frac{\partial y_{df}}{\partial p_{dfs}} \frac{\partial p_{dfs}}{\partial a_{dfs}} \frac{\partial a_{dfs}}{\partial c_{dfs}}} \frac{\partial c_{dfs}}{\partial w} \tag{2.15}$$

where $y$ = Node just before fully connected layer
$p$ = Node just before pooling layer
$a$ = Node just before passing through activation function
$c$ = Node just before convolution with element-wise summation
$f$ = Index of features
$s$ = Index on a kernel
$d$ = Index on data-set

Derivatives except rightest one of (15) is clear. Leftest term is derived from chapter (1). Second left one which is derivative of pooling operation is derived from (14). Third left one which is derivatives of activation function is derived from (13). Fourth left one which is derivatives of summation after convolution is derived from (12). Actual calculation depends on adopted activation function and way of pooling. But generally can be written as following.

$$\frac{\partial E_d}{\partial c_{dfs}} = \phi'_2 \sum_{i=0}^{I-1} \sum_{s=0}^{S-1} b_{dj(s+n)} w_{ijs} \phi'_1 \sum_{i=0}^{I-1} \sum_{s=0}^{S-1} b_{dj(s+n)} w_{ijs} \tag{2.16}$$

To summarise them, the operation which is marked as underline is regarded as almost identical operation of feed-forward process . Only difference of them is activation function is refer to the values of nodes on feed-forward process not the back-propagated error. That means, error back-propagation itself is again linear, and negative errors can be flowed as opposed to feed-forward process.

What is still not clear is operation to update weights after propagating errors to the node.

This can be derived from following.

$$\frac{\partial c_d}{\partial w_{lijn}} = a_{di(s+n)} \cdot c_{dj(s+n)} \tag{2.17}$$

where $i$ = index on input features to connect this weight
$j$ = index on features of propagated errors to connect this weight
$n$ = index on dimension

The result is almost identical with updates for multilayer perceptron. This is because inner product and convolution is both linear combination of each weight. Difference between them is for multi-layer perceptron, a weight is used only one time, but for CNN weights are multiplied with multiple values on different dimensions. In either case, updates function can be written as inner product between previous values of nodes and propagated errors in the next layer.

As we have realized in the calculation of error derivatives with regards to each weights, this process can be divided into 2, which is error propagation, and updating weights after propagation. Of course, you can calculate error derivatives integrating both operation. However, calculating error derivative for each weights individually is not computationally efficient because computation for error propagation is always identical over calculations for error derivatives towards different weights.

Therefore, error propagation step and updating weights should be distinct, and latter step should wait until you get the propagated values for a weight.

### 2.4.6 Code representation of error propagation

Error propagation code can be written as following.

```
--argument
  --1st : derivative of activation function
  --2nd : Node on previous layer
  --3rd : Error on subsequent layer
  --4th : weights in between
  backPropagateError1D :: (Double->Double) -> Node1D
  -> Error1D -> Weight1D -> Error1D

  --same with 1D
  backPropagateError2D :: (Double->Double) -> Node2D
  -> Error2D -> Weight2D -> Error2D

  --back propagation step of CNN is very similar to forward step.
  --however, there is a difference in terms of multiplying activation function.
  --In backward process, input for activation function is previous values of
    nodes not the propagated error.
  --that is why (*df xx)) is used.
  --Then, propagated error is multiplied by it
  backPropagateError1D df n =
     map2 (\x -> map2 (\xx -> (*(df xx)) . foldl (+) 0)
     x . t' . map2 (convolve1D) n)


  --same with 1D except one more map2 and convolution by 2D
  backPropagateError2D df n =
    map2 (\x -> map2 (\xx -> map2 (\xxx -> (*(df xxx)) . foldl (+) 0) xx )
    x . t' . map2 (convolve2D) n)
```

Propagation of error is element-wise summation after convolution which is exactly same of forward process. There is another way of calculation which have been introduced in the section for inner product based Feed-forward. Of course, it can be applied to error-propagation step. I will consider the meaning in a following section.

### 2.4.7 Code representation of updating weights

From (17), updating process can be written as inner product between input nodes and output nodes. But, we have to note that in convolution process, same weights are used as many times as the input nodes dimension. Therefore, input nodes does not need to be transposed unlike multilayer perceptron so as to keep its length of column as dimension of data.

In order to compute different weights on a filter respectively, dimension of input nodes and errors on subsequent layer requires to be slided according with the position of weights on a filter. That process is written as sliding input nodes according to kernel size before feeding them into dot product.

As piece of code, it can be realized by following in case for one dimensional CNN, and figure() represents basic idea of computation.

```
  --argument
  --1st : Node on previous layer
  --2nd : Error on subsequent layer
  --3rd : Kernel size (this is necessary for sliding method which I adopt)
  --return : derivative of weight
  weightDerivative1D  :: Node1D -> Error1D -> KernelSize -> Weight1D

  slideNode1D :: KernelSize -> Node1D -> [Node1D]

  -- process to calculate derivatives of weights in each layer
```

```
-- weights are updated by values of previous node
-- and accumulated error in the next layer
-- since forward process is convolution, weights needs
-- to be calculated by iterating all of
-- pairs of node and errors. This operation can be done by dot product.
weightDerivative1D n e fs = map (\x -> dot' x (t' e) ) m
    where s  = fs `div` 2
          m  = slideNode1D s n


--first let size of list longer so as to reflect
--kernel size of the convolution.
--then, pick up all of possible combinations of filters
slideNode1D k = map (\x-> map2 (pickUp x) [0..2*k]
(reverse [0..2*k])) . map (boundry1D k k)

pickUp  :: [a] -> Int -> Int -> [a]

--this function pick up part of list dropping first I element,
--and last j element.
pickUp a i j = (reverse . drop j . reverse . drop i) a
```

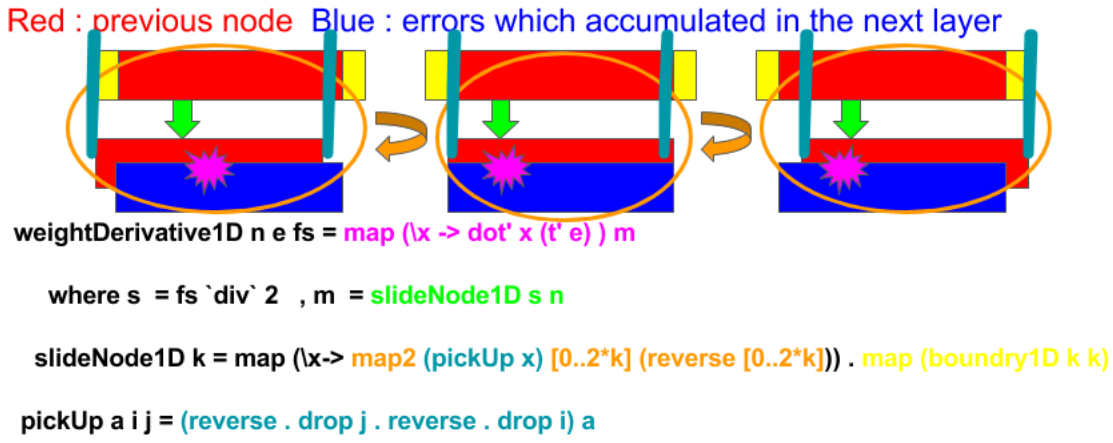


FIGURE 2.9: Updating weights for 1D CNN

For two dimensional CNN, weights are updated as following. Figure () provides visualization of codes, and Figure () describes visual combination of slided input nodes and errors on subsequent layer.

```
--same with 1D
weightDerivative2D  :: Node2D -> Error2D -> KernelSize -> Weight2D

slideNode2D  :: KernelSize -> Node2D -> [[Node2D]]
slideNode2D' :: KernelSize -> Node2D ->  [Node2D]
```

```
  --basic idea is same with corresponding 1D version,
  --note m is going to be five dimension,
  --and most inner word dimension should be concatenated
  --to be fed into the dot product.
  weightDerivative2D n e fs = (map.map)
      (\x -> dot' (map (concat) x) (map (concat) e) ) m
      where s  = fs `div` 2
            m  = slideNode2D s n

  --size of twoD node is going to be larger in any directions,
  --reflecting convolution kernel,
  --then, cut out all of combinations of the candidate list,
  --for instance, if the kernel size is three,
  --nice rectangular nodes should be cut.
  --this step can be done by applying pickUp function
  --both vertically and horizontically.
  slideNode2D  k = map (\x -> slideNode2D' k (t' x))
      . slideNode2D' k

  --2D version of slideNode1D
  slideNode2D' k = map (\x-> map2 (pickUp x) [0..2*k]
      (reverse [0..2*k])) . map (boundry2D k k)

--argument
  --1st : learning rate
  --2nd : derivative of weight
  --3rd : weight before update
  --4th : updated weight
  updateWeight1D :: LearningRate -> Weight1D
    -> Weight1D -> Weight1D

  --same with 1D
  updateWeight2D :: LearningRate -> Weight2D
    -> Weight2D -> Weight2D


--update weight can be done
  updateWeight1D lr = (map2.map2.map2) (\x y-> x*y*lr)
  updateWeight2D lr = (map2.map2.map2.map2) (\x y-> x*y*lr)
```

## 2.5 Interpretation of Inner product based feed-forward / feed-back

I have introduced another way of feed-forward and feed-back calculation which I name inner based product feed-forward and feed-back. What is the incentive to consider making use of this method ? In terms of saving computation, it is unlikely to reduce it significantly because computation numbers of multiplication and addition for both methods are same. If it could reduce, it attributes whether it could let the network more parallel.

The biggest pros to consider this computation is that it gives you another representation which is unlike convolution.
By exchanging the order of computation of two summation, each weights are respectively multiplied with every values on the data. For instance, if the input data is image, this operation means the degree which is how much extent the original image would be emphasized or weaken. Then, each emphasized or weaken channel would be integrated together after being slided to certain direction.

```
Red : previous node  Blue : errors which accumulated in the next layer

weightDerivative2D n e fs = (map.map) (\x -> dot' (map (concat) x) (map (concat) e) ) m

    where s  = fs `div` 2 , m  = slideNode2D s n

slideNode2D  k = map (\x -> slideNode2D' k (t' x)) . slideNode2D' k

slideNode2D' k = map (\x-> map2 (pickUp x) [0..2*k] (reverse [0..2*k])) . map (boundry2D k k)
```

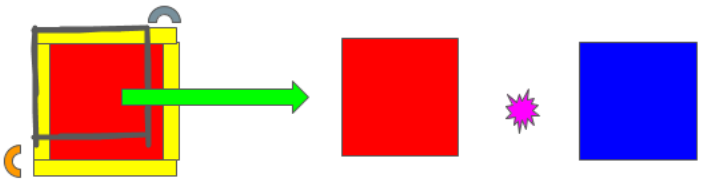

FIGURE 2.10: Updating weights for 2D CNN

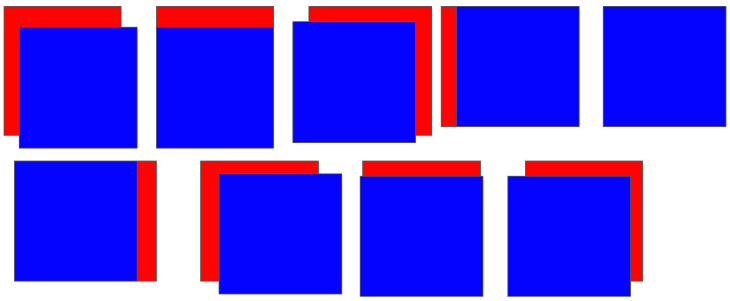

FIGURE 2.11: Updating weights for 2D CNN (all kernel)

Recently, 1 by 1 convolution convolution is often applied mixed with 3 by 3 filters such as Google net[] or Residual net[]. 1 by 1 convolution means, from inner product based feed-forward's point of view, multiplying same weights to the input features and adding up element-wisely. Second addition after sliding each windows are omitted because there is just no addition for convolution. Thereby, 1 by 1 convolution never allows different dimensions in data-set to interact each other, but lets correspond dimensions in each features mix with.

Weights value can be negative. If it is negative, it contributes to pull integrated values to negative. If all weights are negative, it does not pass any information to the next layer if the activation function is rectified linear unit. If they are all positive, it neither reduces information nor retrieves features. Therefore, positive and negative weights needs to be mixed in a degree that could capture respective features.

Average or summation of all of weights on a filter lets you estimate how much information is going to be reduced by applying these weights. If average is positive, it is more likely that information quantity is not so much reduced by the filtering. However, negative average does not necessarily attribute all information is rubbed out. Figure () shows convolution after applying certain weights. Average of weights are negative in case for left-top one, as all weights are [[-1,0,0],[0,0.8,0],[0,0,0]] . However, it retrieves some edges. Same things can be applied to different filters.

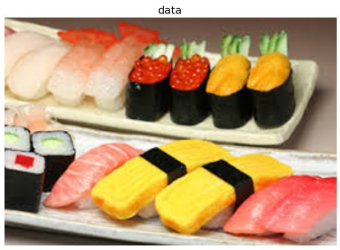


FIGURE 2.12: original data

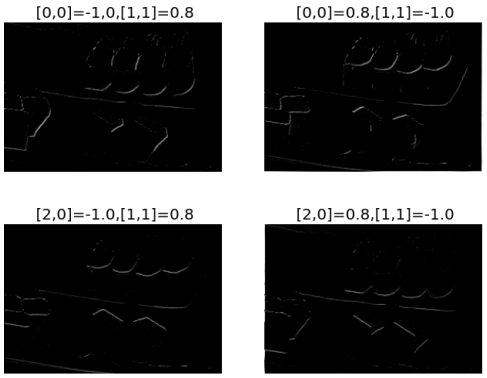


FIGURE 2.13: representation after convolution

Also recently, binary neural network which let values of weights either +1 or -1 has remarkable accuracy of classification in spite of its simplicity. It suggests following. First, in many cases, there are multiple combinations of weights to leave as best performance as the other weights could achieve. Second, pairs of adjacent weights which enables it to maximally retrieve features could be done by the combinations of weights whose value is either -1 or 1, assuming maximum boundary of absolute weights is 1. For instance, edge detection for image classification, edge is maximally retrieved by 1 by 1 adjacent filters whose values is large and signs are different. In binary network, there is no ways that weights could be "0". It might be better in terms of computation efficiency because weights whose value is "0" has no influence towards different dimensions, meaning that it does not at all help to retrieve features for classification inside of data. If you would let a feature retrieval to the more backward layer, you can set the weight pairs as [+1,+1]. If you would discard them, you could set [-1,-1], which will turn out that activation function can delete its influence.

What can we say about sliding windows for achieving summation of convolution in the last step? It means, dimension in the data is only mixed with some certain local area which is specified by the size of kernel. If CNN is two dimensional and the size of filter is 9, dimensions which are referred each other are the ones aligned with less than one intervals in between. That locality preference does take into account of relationship over more than filter size stacking multiple convolution.

## 2.6 Computation for binary input and weight with inner product based feed-forward / feed-back

Binary neural network is quite promising approach not only in terms of computation efficiency but understanding of CNN. It sets weights as binary or both inputs and weights binary. Some ways to compute convolution of binary input data with binary weights has been come up with recently[]. Inner product based feed-forward which I proposed in this chapter allows its computation to turn to binary operation in a pretty elegant way.

Let us assume network size and network weights are both binary, with precisely, input data is represented as "0" or "1", and weights are either "1" or "-1" as a functionality, but "-1" is represented as "0" and "1" as it is so that it can be binary data.

Operation of first multiplication can be covering a bit-mask. However, since weight "-1" turns to "0" which erases result of operation in either input data is "0" or "1", it should be converted to "1" in a way. To realize this, multiplication can be divided into two sub function which one of them is "AND" operation without flipping and another is "AND" operation after flipping operation. By this step, first multiplication

step of binary step had been done.

Second, the value in the same dimension but in a different dimension needs to be summed up. This step is done by "Bit count" operation. It just calculates either which bit either "0" or "1" exceeds the other one in terms of its number. If the number of "1" exceeds "0", the overall value turns out to be "1". Otherwise, the overall value become "0".

After above step, all of nodes which is covered by bit-mask in a filter needs to be added by each element after sliding to certain direction. This sliding operation can be done by "bit-shift" operation. After bit-shift operation, again "bit-count" operation should be applied so that it along with summation of bit.

Replacing addition by "bit counting" contains quantization. But, in order to let every information be 1 bit, that should be a most proper way of computation.

To summarize it, overall algorithm can be written as following.

1. Bit mask operation by weights on each filter.
2. Bit count operation for getting together distribution from different input features.
3. Bit shift operation (to the direction which is determined by weights).
4. Bit count operation for combining distribution from weights on a filter.

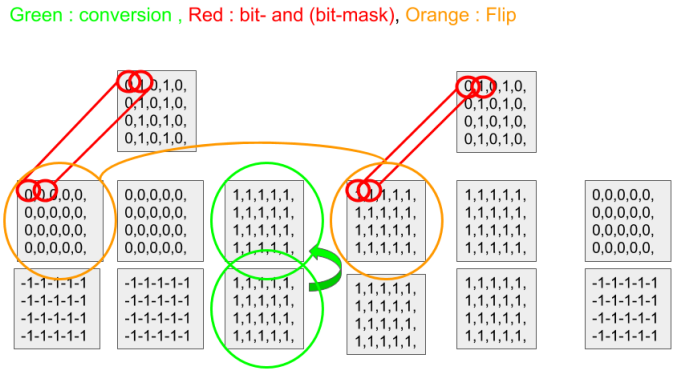


FIGURE 2.14: binary convolution 1

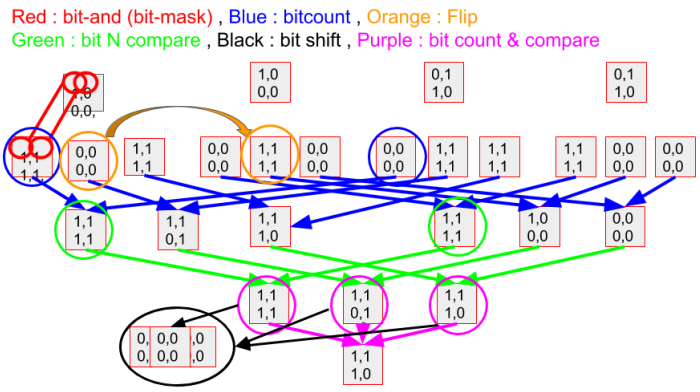


FIGURE 2.15: binary convolution 2

For binary neural networks, passing through activation function change nothing towards network because entry to the input is , from its beginning, either "0" or "1". In other words, you can omit the step of

activation function, but it does not mean functionality of non-linear is omitted. In fact, the process after bit-counting operation which is done inside of convolution step has sort of similar function of passing through activation function.

In the feed-back process, convolution step for binary input and weight is perfectly identical with feed-forward step. What is unique is operation of multiplying between back-propagated errors and derivative of activation function. Derivative of activation function can be replaced by the value after convolution step.

## 2.7 Further discussion

In this section, I introduced representation of CNN with Haskell and proposed another way of calculation with it. This calculation is not for computational cost reduction, but analysis of weights for convolution step and calculation for binary input features and weights.

However, still there are very few points to reveal superiority of CNN which I can make after suggesting this way of computation. Then, is there any other approaches to give new perspective or consideration?

In order to investigate further, in the next chapter, I will step back to the basic questions which are, what is dimension of data, what is dimensional reduction, having look at adjacent area, namely compression,

# Chapter 3

# Compression for dimensional reduction

## 3.1 Overview

In this chapter, I will discuss what is meant by reducing dimension of data because the process in CNN layer by layer can be regarded as successive reduction of information quantity while retrieving features.

First, I will show the shared points between dimensional reduction and compression.

Second, I will introduce some compression algorithms in a range that they can contribute to explain dimensional reduction which is happening in CNN.

Overall objective of this chapter is to provide general points of view to investigate CNN better, and it is not directly connecting to binary neural networks.

## 3.2 dimensional reduction and compression

### 3.2.1 Data Number and Dimension

Data for machine learning has two different axis to be analyzed.

One is number of data-set. Another is dimension. If you take both as a vector. Data is represented as two dimensional matrix where you can use tools for linear algebra. If you let each data as a function, you can regard the dimension as terms of polynomial function, and see its combination as overall sequential equation.

As I discussed in the previous chapter, tasks for machine learning can be categorized with two.

One is reducing data's dimension. Another is categorization ( classification ) of data set.

Both can be called dimensional reduction. Note that, in this thesis, "dimension" means numbers of properties that a data has.

Categorization of data-set needs dimensional reduction if original data set has more or less redundant dimension. Mathematically, categorization requires data's dimension to be linear independent.

If there is a data-set just has 1 dimension, classification is pretty easy by adopting some metrics over data. However, if data has multiple dimension which is far from being independent each other, descriptive dimension unvails intrinsic features, thereby counting on original data dimension will deteriorate dimension

### 3.2.2 Independent and sequential dimension

There are two distinct types of data in terms of its dimension.

First type of data is order of dimension can be arbitrary. It is data which is like user profile information at customer service department in one of companies. The order of column does not affect anything about data itself. The data source is called memoryless source because the information in between data has 0 which means every dimension is independent.

Second type of data is the order of each dimension is indispensable information and cannot be commutative each other. This is data such as natural language, speech,and image data. The source of information has some memory in its context, and thereby called finite or infinite context model. Furthermore, there are two models of information source. One is Markov information source which is often used for data whose

type is character. Another is Fourier transformation based which is used for data for double type.

If the data has some context between each dimension, it have to be captured, and redundant dimension should be integrated. In the end, Second type of data which has dependency inside of it should ideally reduce to former types of data which each dimension is independent.

### 3.2.3 Compression and dimensional reduction

When reducing dimension of sequential data, there are two distinct case. First case is data is just one. In that case, the way to compress is called compression algorithm. Second case is data is more than 2. In this case, algorithm is categorized as one of machine learning method. First case is independent from individual difference of data and measured by self-information quantity assuming dimension is mutually independent. On the other hand, second case is example dependent, and two data's similarity is measured by mutual information quantity. Note mutual information is also assumed independency of dimension. Data whose dimension is independent is difficult to measure its information quantity and minimum length of code which is deduced from it.
Even though both method is reducing dimension, both aims and applied algorithm is so different. Since we are interested in dimensional reduction which is beneficial for classification, compression algorithm seems to have few relevancy in our topic. However, it is required to extract common points inside of dimensions of one data before comparing multiple data if dimension of data is dependent. That is because you cannot compare multiple values of same dimension over multiple data unless you make sure that its dimension is independent. For instance, comparing single pixel whose row and column is same does not make sense if two picture is pre-aligned by human laboring.

Then, how could we scoop up orders of dimension as well as dimension together in order to find intrinsic feature?
To answer the question, I will devote this chapter for compression algorithm in terms of capturing context between data.

To summarise this section, supervised learning task ; for instance, classification task which is solved by CNN, needs three steps which is

1. sequential information inference (compression)

2. dimensional reduction between multiple data

3. classification

However, Each process does not need to be stepped sequentially, but can be simultaneously.

## 3.3 Dimensional reduction for independent data

### 3.3.1 run length encoding

Given that we have signal data = [0,0,0,1,1,1,1,0,0,0].
how could I reduce dimension, in this case information quantity?

One of the primitive algorithm is run length encoding which just putting same and sequential symbols together.

```
class RunLength where
    runLength :: (Eq a) => [a] -> [(a,Int)]

instance RunLength where
  --runLength h = map (head &&& length) $ group' h
```

```
  runLength = map (\x -> (head x,length x)) . group'


take'  :: Int -> [a] -> [a]
take'' :: Int -> [a] -> [a]

drop' :: Int -> [a] -> [a]


--this group needs one argument, let successive symbols integrate.
--e.g.."sssaaaavvvxxx"-> " "sss","aaaa","vvv","xxx" "
group' :: (Eq a) => [a] -> [[a]]

take'' a l = reverse $ foldl (\x y -> if length x < 2 then y : x else x) [] l

drop' 0 a  = a
drop' i (h:t) = drop' (i-1) t

group' [] = []
group' h = (take l h) : group' (drop l h)
    where s = accord' h
          l = length s
```

### 3.3.2 Huffman coding

However, if the data is not binary but has very few sequential symbols such as [0,1,2,1,2,1,2,1,2,1,2], run length encoding rather increase the size of data. For such data, Huffman coding is more useful.

The concept of Huffman coding is assigning length of bits according to the frequency of symbol. Since frequently appeared symbol will get shorter bits, on the other hand, hardly seen symbol will get longer bits, to sum them up, it is expected that the size of bits after coding is going to be smaller.

For functional language, hash is implemented by a binary tree preventing side effects, and reduce computation resource for search $(O(N^2) -> O(logN))$.

Since data structure which entails encoding information as well can be treated as a binary tree which is biased to either rightest side or leftest side for making them prefix code representing figure(1), it is well suited with the way for functional programming language to treat hash.

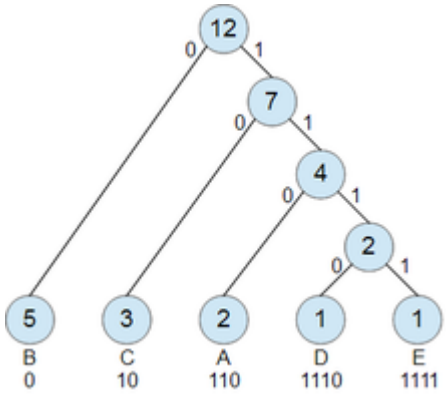


FIGURE 3.1: Huffman Tree

```
import MyLib.LeafTree
import MyLib.TreeHash
```

```
class Huffman where

  --this is integration of all code.
  huffman :: (Ord a) => [a] -> (String,[(a,String)])

  --make dictionary for encoding
  makeDict  :: LeafTree k v -> [(k,String)]
  makeDict' :: String -> LeafTree k v -> [(k,String)]

  --1st arugment: signal
  --2nd arugment: tree which has encode information
  encode  :: (Ord k) => [k] -> LeafTree k Int -> String
  encode' :: (Ord k) => String -> LeafTree k v -> k -> String


instance Huffman where

  --huhman coding integration code.
  --1 : cumulativeFromList ( TreeMap )
  --get cumulative frequency given input list in a tree structure,
  --2 : treeToList
  --change the tree into list,
  --3 : insertsVL ( LeafTree )
  --again change to tree but in this time, leaf tree
  -- leaf tree are allowed us to put anything on their intermidiate node.
  --4. create dictionary
  --4. encode given created leaf tree

  huffman l = (encode l a, makeDict a)
    where a = ((insertsVL . treeToList) . cumulativeFromList) l


  makeDict = makeDict' ""
  makeDict' _ None = []
  makeDict' s (LNode (Leaf k v) r) = (k,t) : makeDict' u r
    where t = s ++ "0"
          u = s ++ "1"


  --encode' _ None = []
  encode l tree = foldr (\x y -> (encode' "" tree x) ++ y) "" l
  encode' s (LNode (Leaf k v) r) e
    | e == k    = t
    | otherwise = encode' u r e
    where t = s ++ "0"
          u = s ++ "1"
```

Encoding and making hash table for decoding can be done by traversing the Huffman tree which is represented as figure(1).
However, since tree as hash and tree for encoding is different in a sense that node except edge of the tree is able to hold information or not, tree as hash is converted list, and then insert again along with the values to tree of figure(1).

Tree hash, specifically, the inside of function " freqTableFromList " is described by following.

```
data TreeMap k v  = Nil | Node k v (TreeMap k v) (TreeMap k v) deriving (Show)

class TreeHash where

  -- cumurative frequency
  -- iterating list
  cumulativeFromList :: Ord k => [k] -> TreeMap k Int

  --adding elements given one query
  cumulative :: Ord k => k -> TreeMap k Int -> TreeMap k Int

  --tree To list
  treeToList :: TreeMap k v -> [(k, v)]

  --traverse
  --compareL :: TreeMap k v -> (k,v)

instance TreeHash where

  -- input  : list
  -- output : Tree
  freqTableFromList = foldl (\x y -> cumulative y x) Nil

  -- if there are no elements at the point you come to a leaf, add new node
  cumulative a  Nil = Node a 1 Nil Nil
  -- if there are same elements in the list, add +1 on top of its index
  cumulative a (Node k v l r)
    | a == k    = Node k (v+1) l r
    | a < k     = Node k v (cumulative a l) r
    | otherwise = Node k v l (cumulative a r)
    -- search by key, go depth guided by key

  --this is search function
  search _ Nil = Nothing
  search x (Node k v l r)
    | x == k    = Just v
    | x < k     = search x l
    | otherwise = search x r

  --change tree to list
  treeToList tree = traverse tree [] where
    traverse Nil xs = xs
    traverse (Node k v l r) xs = traverse l ((k, v) : traverse r xs)
```

Another tree ( Huffman Tree ) which has the coding information as the path to the each node is described by following.

```
--LeafTree is only leaves are able to hold pairs of key and value.
--Note LNode has just two argument unlike Node
--which is defined inside of TreeMap
data LeafTree k v = None | Leaf k v
   | LNode (LeafTree k v) (LeafTree k v) deriving (Show)

--This is class for functions which is for LeafTree
class FuncForLeafTree where
  --insert elements sorting by values
  insertsVL :: Ord v => [(k,v)] -> LeafTree k v
  insertVL  :: Ord v => k -> v -> LeafTree k v -> LeafTree k v
```

```
instance FuncForLeafTree where

  --this is just looping list of elements
  insertsVL = foldl (\a (k, v) -> insertVL k v a) None

  --this creates left-shifted tree
  insertVL k v None = LNode (Leaf k v) None
  insertVL k v (LNode (Leaf lk lv) r)
    | v > lv     = LNode (Leaf k v) (LNode (Leaf lk lv) r)
    | otherwise = (LNode (Leaf lk lv) (insertVL k v r))
```

Average coding length which is provided by Huffman coding cannot be reduced under self entropy. In other words, minimum coding length is determined by self entropy.

$$H(X) = -\sum_{x \in z\chi} p(x) log_2 p(x) \tag{3.1}$$

However, the compression limitation assumes memoryless source which every single data that come from is independent from each other.

If data-set does not hold independency assumption, there is a case that average coding length can be smaller than entropy in case of memoryless information source.

Nevertheless, the notion is still valuable because many compression algorithm adopts Huffman coding in the last process assuming redundant dimension can be reduced to independent base dimension where one can see its source as a memoryless information source.

## 3.4 Dimensional reduction for context dependent data

### 3.4.1 Compression for symbolical data (Prediction by partial matching )

When the source of the data is not memoryless information source, and data has some information in between each data, there are some approaches to capture dependency between data. One of them is called finite context model whose source is Markov information source.

Here is an example of analysis of data assumed Markov information source.

Given data
Arithmetic coding

### 3.4.2 Compression for non-symbolical data ( JPEG algorithm )

In the previous section, we took a look at an algorithm for data which is generated by Markov source. However, a problem of this model of source is when the data is represented as numerical values, the difference between multiple values are not going to be taken into consideration. Concretely, every distance between each byte sequence of ASCII code can be treated equal, this is proper. Whereas, difference of pixel values cannot be seen as having equal distance by this model. That is, Markov model assumes simplest unit can be treated as mutually independent, but it is applied only for symbolical data.Then, we need to have another context dependent model for compressing non-symbolical data.

In this section, I will take a look at one example algorithm; JPEG compression algorithm, to consider what means by compressing non-symbolical data.

There are some different ways of encoding, but Most dominantly used JPEG algorithm(JFIF encoding) consists of

1. Color space transformation (RGB to YRB)
2. Down-sampling (Resolution reduction)
3. Block splitting
4. Discrete Cosine transformation
5. Quantization (Pixel values reduction)
6. Jigzag scan
7. Separation of DC / AC component
8. Encoding difference of DC component
9. Zero-length Encoding for DC component
10. ( Huffman coding )
11. Conversion to bit sequence

As you can see, it holds many steps but most crucial step is 4. Discrete Cosine transformation. I will have a brief look at each step.

**1. Color space transformation**

Input image is converted from RGB into a different color space called YCBCR (or, informally, YCbCr). It has three components Y′, $C_B$ and $C_R$ : the Y′ component represents the brightness of a pixel, and the CB and CR components represent the chrominance (split into blue and red components).

```
data Image = RGB [[RGBPixel]] | YRB [[YRBPixel]]

data RGBPixel = RGBPixel
  {
    red   :: Double,
    green :: Double,
    blue  :: Double
  } deriving (Show)


data YRBPixel = YRBPixel
  {
    intensity :: Double,
    hueRed    :: Double,
    hueBlue   :: Double
  } deriving (Show)


  --First step is changing from rgb to yrb
  --image data type has three channels
  rgbToyrb :: Image -> Image

  --pixel has just three double values
  --data type is going to be changed from RGB to YRB
  rgbToyrb'':: RGBPixel -> YRBPixel

  --each 2D picture is converted to YRB
  rgbToyrb (RGB a) = YRB (map (map (rgbToyrb')) a)

  --actual operation as a pixel level is done here.
  --this transformation is defined by Jpeg.
  rgbToyrb' a = YRBPixel {intensity = i,hueRed = hr,hueBlue = hb}
     where i  =   0.299 *(red a) + 0.587 *(green a) + 0.114 *(blue a) - 128
```

```
          hr = - 0.1687*(red a) - 0.3313*(green a) + 0.5  * (blue a) + 128
          hb =   0.5   *(red a) - 0.4187*(green a) - 0.0813*(blue a) + 128

--each 2D picture is converted to YRB
  rgbToyrb (RGB a) = YRB (map (map (rgbToyrb')) a)

  --actual operation as a pixel level is done here.
  --this transformation is defined by Jpeg.
  rgbToyrb' a = YRBPixel {intensity = i,hueRed = hr,hueBlue = hb}
     where i  =   0.299 *(red a) + 0.587 *(green a) + 0.114 *(blue a) - 128
          hr = - 0.1687*(red a) - 0.3313*(green a) + 0.5  * (blue a) + 128
          hb =   0.5   *(red a) - 0.4187*(green a) - 0.0813*(blue a) + 128
```

Motivation to convert into YRB is this three channel represent more characteristics of data. The reason that hue of green is eliminated is it can be represented as middle of red and blue.

**2.Down-sampling**

Due to the densities of color- and brightness-sensitive receptors in the human eye, humans can see considerably more fine detail in the brightness of an image (the Y′ component) than in the hue and color saturation of an image (the Cb and Cr components). Using this knowledge, encoders can be designed to compress images more efficiently.

```
  --downsampling is reducing resolution.
  --Normally ration of resolution -> y(intensity) : hueRed : hueBlue
  --is going to be 4:2:2, but in this case,
  --4:4:4 which is often used indigital camera
  downSampling :: Image -> [[[Double]]]

  --In the process of downsampliing, data type is changed to list of double
  yrbToDouble :: YRBPixel -> [Double]

--scaling down resolution of Hue-red and Hue-green with respect to intensity
  --in this case, just chaging data type from my own "YRB" to Double
  downSampling (YRB a) = map (map (yrbToDouble)) a

  --this is type conversion
  yrbToDouble a = [i,hr,hb]
     where i  = intensity a
          hr = hueRed  a
          hb = hueBlue a
```

In most case, down-sampling is done by pooling resolution of hue red channel and hue blue channel. But, in this code, I will let it be because there are some cases that omit down-sampling.

**3.Block splitting**

After subsampling, each channel must be split into 88 blocks. Depending on chroma subsampling.

```
  --next step of down-sampling is block splitting
  --which each channel is truncated with 8 by 8
  blockSpliting :: [[[Double]]] -> [[[[Double]]]]

  --block splitting is splitting whole resolutions
  --into 8 by 8 blocks
  blockSpliting a = map (map (group 8)) a
```

Block splitting is better because without splitting, spectrum of coefficients tends to have very steep curve, which makes quantization step difficult.
The size of block ;8 may be derived from statistical experiments and partially because it it even.

**4.Discrete cosine transform**

Before computing the DCT of the 8 by 8 block, its values are shifted from a positive range to one centered on zero. For an 8-bit image, each entry in the original block falls in the range [ 0 , 255 ] [0,255] [0,255]. The midpoint of the range (in this case, the value 128) is subtracted from each entry to produce a data range that is centered on zero, so that the modified range is [ 128 , 127 ] [-128,127] [-128,127]. This step reduces the dynamic range requirements in the DCT processing stage that follows.

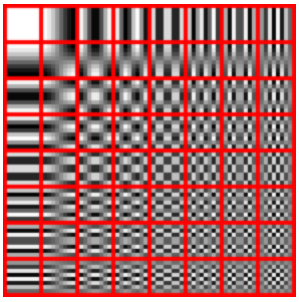

FIGURE 3.2: base representation of DCT

```
class DCT where

  dct1D :: [Double] -> [[Double]]

  dct2D :: [[Double]] -> [[Double]]

  dct2D'  :: [[Double]] -> [[Double]]
  dct2D'' :: [[Double]] -> [[Double]]

  base' :: (Double -> Double) -> Double -> [[Double]]

instance DCT where

  dct1D a = dot' b c
     where b = atLeast2d a
           l = fromIntegral (length a) / 1.0
           c = base' cos l

  dct2D = dct2D'' . dct2D'

  dct2D' a = dot' a c
     where l = fromIntegral (length a) / 1.0
           c = base' cos l


  dct2D'' a  = dot' b c
     where l = fromIntegral (length a) / 1.0
           b = t' a
           c = base' cos l


  --base function generation
  --sub function of base generator
```

```
base' f a = map (\y -> map (\x -> f ( 2 * pi * x * y / a ))
  [0..b] ) [0..b] where b = a - 1
```

1d and 2d discrete cosine transformation can be written as inner product ( dot' ), which is also used by neural network. Base function is two dimensional matrix and its coefficients are symmetrical. Its details are explained with Fourier transformation in the next chapter.

**5. Quantization**

The human eye is good at seeing small differences in brightness over a relatively large area, but not so good at distinguishing the exact strength of a high frequency brightness variation. This allows one to greatly reduce the amount of information in the high frequency components. This is done by simply dividing each component in the frequency domain by a constant for that component, and then rounding to the nearest integer. This rounding operation is the only lossy operation in the whole process (other than chroma subsampling) if the DCT computation is performed with sufficiently high precision. As a result of this, it is typically the case that many of the higher frequency components are rounded to zero, and many of the rest become small positive or negative numbers, which take many fewer bits to represent.

```
--next step after dct is quantization in each pixel.
--As a matter of fact, its process is just
--divided by values of quantization tables
--in a floating manner.
--values of elements on quantization table is set by Jpeg standard.

quantization :: [[Double]] -> [[Double]] -> [[Int]]
quantizationTable :: [[Double]]

--quantization is dividing by values of quantization tables and converting integer.
quantization = map2 (map2 (((round .) . (/)) ))

--quantization is defined by Jpeg standard
quantizationTable =
  [
    [16,11,10,16,24,40,51,61],
    [12,12,14,19,26,58,60,55],
    [14,13,16,24,40,57,60,56],
    [14,17,22,29,51,87,80,62],
    [18,22,37,56,68,109,103,77],
    [24,35,55,64,81,104,113,92],
    [49,64,78,87,103,121,120,101],
    [72,92,95,98,112,100,103,99]
  ]
```

The value of quantization table might also be derived from accumulated experiments to consider both information loss in this process and reduction of information quantity in the end.

**6. Jigzag scan**

This step is represented as figure (3).

```
--next step after quantization is jigzag scan
--which traverse two dimensional tables in a jigzag way.
--It will let 2D tables 1D
jigzag  :: [[Int]] -> [Int]
jigzag' :: Int -> Int -> [[Int]] -> [Int]

--two dimensional blocks are converted into 1d 64 elements list
```

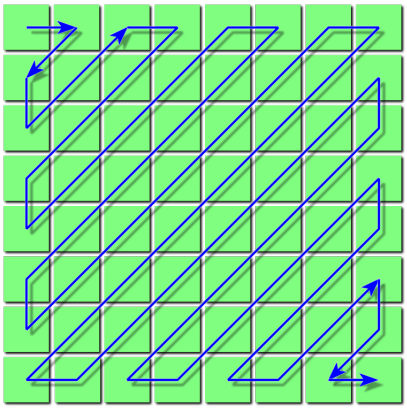

FIGURE 3.3: Jig-zag scan

```
jigzag a = jigzag' ((length a)*2) 0 a

--jigzag process has 3 arugments
--argument is (depth of jigzag(8by8->16),current depth,2D list).
jigzag' m a b
-- current depth is the end of jigzag, prepare empty list
  | m == a    = []
-- if the current depth is even, append the first element
--of extracted list as usual
  | even a    = map (head) c ++ r
-- if the current depth is odd, append as reversed
--to let the traversal jigzag
  | otherwise = (reverse $ map (head) c) ++ r
-- after half step of jigzag, it has empty list which needs to be excluded
  where b' = (zip [0..] . filter (/=[])) b
-- according to the depth, lists are extracted
        c = (snd . unzip . filter (\(x,y)->x<a))  b'
-- none-extracted lists is going to brought as next arguments
        d = (snd . unzip . filter (\(x,y)->x>=a))  b'
        r = jigzag' m (a+1) ((map (tail) c) ++ d)
```

This scan aims to make as many alignment of zero as possible. This is because, diagonally neighboring elements has alike coefficients.

**7.Separation of DC / AC**

Direct component represents overall brightness of image and is represented as an independent coefficients from others. which is explained in also next chapter.

Alternated component represent how many split patterns is hidden of this image.

Since direct component tends to have by far bigger coefficients.

```
--next step is separating DC component and AC component.
  -- in 64 elements of block, a first value denotes DC component.
  -- and rest of them is AC component
  dcComponent :: [[[Int]]] ->  [[Int]]
  acComponent :: [[[Int]]] -> [[[Int]]]

--after jigzag scan, DC & AC component is divided.
  dcComponent = map (map (head))
  acComponent = map (map (tail))
```

**8. Encoding difference of direct component**

Since direct component is average intensity of its block, neighboring direct component might have similar values. To decode correctly, first direct component have to be encoded as it is.

```
--For DC component, the difference of previous values
--over multiple blocks are stored,
--except values of first block in each channel.
takeDifference  :: [Int] ->  [Int]
takeDifference' :: [Int] ->  [Int]

--DC component is encoded by the value of its difference.
takeDifference (h:t) = h : takeDifference' t
takeDifference' [h]   = []
takeDifference' (h:t) = h - (head t) : takeDifference' t
```

**9. Zero-length Encoding for Alternated component**

Elements of alternated component after jig-zag scan is expected to have many zero especially in the end of list which means high frequency coefficients. To reduce information maximally, each element of zero should not be coded as it is. Plus, to decode correctly, each length of running code, and element itself have to have fixed code length. However, coefficients in frequency domain are likely to have very big values in some cases even after quantization step. Thereby, JFIF standard lets numbers of bit to represent each coefficient variable code length.

Concretely, zero length yield following data format.

–[((A,B),C)]
–A = number of zero before this value (4 bit fixed)
–B = minimum bits enough to express coefficient value (4 bit fixed)
–C = its value

```
--For AC component, things are more complicated..
--All of non-zero code will be coded.
--[((A,B),C)]
--A = number of zero before this value (4 bit fixed)
--B = minimum bits enough to express coefficient value (4 bit fixed)
--C = its value
zeroLength  :: [Int] -> [((Int,Int),Int)]
zeroLength' :: [Int] -> Int -> [((Int,Int),Int)]

--AC component is only non-zero component is stored unless it has
--sequence more than 15 zero.

zeroLength a = zeroLength' a 0

--
zeroLength' [] _ = []
--1st argument is encoded list, second argument is variables
--to store previous successive number of 0
zeroLength' (h:t) s
--if it comes zero, you will jump it storing number of zero
  | h == 0    = zeroLength' t (s+1)
--if it comes non zero, you will store it with numbers of
--minimum bits for this value
  | otherwise = ((s,bit+1),h) : zeroLength' t 0
--minimum bits is calculated as following.
  where bit = round $ logBase 2 $ fromIntegral (h::Int)
```

**10. ( Huffman coding )**

Normally, after doing zero length encoding, Huffman coding follows. But, I will omit this code since I described already.

**11. Conversion to bit sequence**

Last step is letting sequence of integers to sequence of bits.

```
--integer is going to be transformed into bits
--in this case sequence of char type
 --first argument is expected length of bits
 bitsTransform  :: Int -> Int -> String
 bitsTransform' :: Int -> String

 --this is sub component of bits transform
 zeroPrepare :: Int -> String

 --this is a tool for feeding AC component
 --into bitsTransform function.
 acBits :: ((Int,Int),Int) -> String

 --integer turns into bits but as a string format
 --1st argument is length of bits showing discrimination for decoding
 bitsTransform a b = (zeroPrepare (a-l)) ++ c
   where c = (reverse . bitsTransform') b
         l = length c

 --int -> bit are done by dividing by 2 repeatedly.
 --if it comes zero, it will be ended
 bitsTransform' 0 = []
 bitsTransform' b
 --if mode b 2 == 0, then, encode as 0
   | mod b 2 == 0 = "0" ++ (bitsTransform' (b `div` 2))
 --if mode b 2 != 0, then, encode as 1
   | otherwise    = "1" ++ (bitsTransform' (b `div` 2))

 --if there is more space to be filled in, insert "0"
 zeroPrepare 0 = ""
 zeroPrepare a = "0" ++ (zeroPrepare (a-1))

 --AC component is fed into transform and concatenate.
 acBits ((a,b),c) = (bitsTransform 4 a) ++
   (bitsTransform 4 b) ++ (bitsTransform b c)
```

To summerise every step, overall pipe line of Jpeg encoder can be written as following.

```
--this jpegEncoder is all integration of functions listed below.
 --it will change from image to byte string
 --(for visualization of final representation, output is char)
 jpegEncoder :: Image -> String

 --this is all of process when image is encoded as .jpg or .jpeg
 jpegEncoder a = dc ++ ac
    where b  = map (map jigzag)
    $ map (map (quantization quantizationTable)) $ map (map dct2D)
    $ blockSpliting $ downSampling $ rgbToyrb a
```

```
dc = concatMap (concatMap (bitsTransform 8))
     $ map (takeDifference) $ dcComponent b
ac = concatMap (concatMap (concatMap acBits))
     $ map (map zeroLength) $ acComponent b
```

### 3.4.3 Consideration towards Jpeg encoding algorithm

Jpeg is known as one of the best compression algorithm which compress image data especially the ones which has general object, and essential idea is lying on discrete Fourier transform.

The underlying idea of DCT is treating data as a function, and the function can be represented as combination of simple base function like triangle function. This approach is widely used for compression algorithm not only for image but sound because these data can be in some case better made sense in decomposed representation by human.

For instance, human can easily make difference between sound of white noise and sound of pink noise distinct. However, analyzing both waves in real space given entropy provided by memoryless information source model and Markov information source model does not makes sense at all. By contrast, the difference between while noise and pink noise is obvious if you compute both frequency spectrum.

Note here that I do not claim that data which is transformed in frequency have always less information quantity. As a matter of fact, there are as many number of waves that increase information quantity in frequency space as the ones which is decreasing. For instance, in case for 2d waves, image of binary character or digits is more compressed in a way which is operated not in frequency space such as PNG algorithm , with respect to Jpeg algorithm.

What I claim is, statistically speaking, data of sound and general object has less information quantity in frequency domain. This can be translated into that human cognition system is more likely to make sense of them if the data is well represented as combination of base function not as it is. This is because state-of-art compression algorithm consider human cognition process.

By the way, this should be distinct with arguments about if information is lost or not after encoding information by some algorithm. There is a remark that lossy compression cut off cognitively less awared information, to compress data. It is true for explaining about sampling and quantization process. But, what is more important is data itself such as image and sound which human think those are useful to be saved originally exhibit more structures in frequency space regardless of application of compression algorithm.

Contrast to minimum coding length given by entropy assuming infinite information source, compression limitation for non-symbolical data is quite difficult to be set. Being difficult in my sense is not brought by lossy compression, namely a discussion about how much extent human does not aware reduced information from original, but fundamentally, non-uniqueness of the way of capturing context between each dimension makes its estimation hard. Compression algorithm is applied to many different data in a same way. However, there are N different cases to reduce N different data.For instance, sine or cosine wave has least information given MPC algorithm, but square wave might have more information quantity by discrete cosine transform rather than applying wavelet transform.

### 3.4.4 Summery

In this chapter, we took a look at some compression algorithms to analyze dimensional reduction. Compression and dimensional reduction are not often discussed together. However, if the data-set is going to be bigger and get generality, its purpose is close to each other. The reason is following. Compression algorithm are required to have sort of generality which should have high compression rate in any data. By contrast, dimensional reduction can be very specific if the data-set is small and has small variance. But, if data-set is bigger with variety enough to represent generality of its groups, feature extraction for dimensional reduction approaches compression. Conversely, dimensional reduction may be regarded as compression for

specific training data. But, most of dimensional reduction algorithm such as PCA is not assured reversibility to the original input from compressed figure unless possible all of error trajectory is preserved and back traced.

# Chapter 4

# Fourier analysis of CNN

## 4.1 Prelude

In the last chapter, we came to a conclusion that in order to compress data such as natural image, we are able to find more characteristics of the data in frequency domain rather than space domain, thereby it enables us to reduce its information quantity. This means hidden dimension between each elements of one data can be revealed better into the frequency domain in some cases.

In this chapter, we will take a look at representation of weights and input data by Fourier transformation after introducing basic knowledge. But, why is Fourier transformation useful to understand the data and filters not only for data like natural images but binary data or weights? This is because it gives us a new representation of convolution operation from different point of view which I showed in the chapter 2. This is based on convolutional theory which states that convolution operation between two functions can be element-wise multiplication of two functions after Fourier transformation.

Analysis in this chapter will succeed to the next chapters.

## 4.2 Fourier Transformation

Fourier transformation is described as following.

$$F(k) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x)\, e^{-ikx} x \tag{4.1}$$

where x = original index defined such as time / space.
f = frequency
f(x) = original function, data-set
F(k) = signal component in frequency domain

Inverse Fourier transformation is defined by following.

$$f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} F(k)\, e^{ikx} k \tag{4.2}$$

Discrete Fourier transform is defined by following.

$$F(k) = \frac{1}{\sqrt{2\pi}} \sum_{-\infty}^{\infty} f(x)\, e^{-ikx} x \tag{4.3}$$

Code of one dimensional discrete Fourier transformation in Haskell is following.

```
import numpy as np

def dft_1d (self,start, x, N):
```

```
    "1d dft"
    X = [0.0] * N
    for k in range(N):
        "iterate different base function"
        for n in range(N):
            "itereate data itself with same base function "
            real = np.cos(2 * np.pi * k * n / N)
            imag = - np.sin(2 * np.pi * k * n / N)
            X[k] += x[start + n] * complex(real, imag)

    X = np.array(X)
    return X
```

In case for one dimensional Fourier transformation, base function, namely $e^{-ikx}$ in () , or " complex(real,imag) " in python code is 2 dimensional array.

For instance, if you put data-set whose length is 4, its base would be

$$\begin{pmatrix} \cos(0) - i\sin(0) & \cos(0) - i\sin(0) & \cos(0) - i\sin(0) & \cos(0) - i\sin(0) \\ \cos(0) - i\sin(0) & \cos(\frac{1}{4}(2\pi)) - i\sin(\frac{1}{4}(2\pi)) & \cos(\frac{2}{4}(2\pi)) - i\sin(\frac{2}{4}(2\pi)) & \cos(\frac{3}{4}(2\pi)) - i\sin(\frac{3}{4}(2\pi)) \\ \cos(0) - i\sin(0) & \cos(\frac{2}{4}(2\pi)) - i\sin(\frac{2}{4}(2\pi)) & \cos(\frac{4}{4}(2\pi)) - i\sin(\frac{4}{4}(2\pi)) & \cos(\frac{6}{4}(2\pi)) - i\sin(\frac{6}{4}(2\pi)) \\ \cos(0) - i\sin(0) & \cos(\frac{3}{4}(2\pi)) - i\sin(\frac{3}{4}(2\pi)) & \cos(\frac{6}{4}(2\pi)) - i\sin(\frac{6}{4}(2\pi)) & \cos(\frac{9}{4}(2\pi)) - i\sin(\frac{9}{4}(2\pi)) \end{pmatrix} \tag{4.4}$$

To put it simply, Fourier transformation would be inner product between this base matrix and input function. If you consider about that, these codes can be written in Haskell by following.

```
import MyLib.MyComplex

--Fourier transformation
--Input  : transformed function can be defined by real number
--Output : two dimensional complex number
-- (why two dimensional? Answer : matrix can be more easily treated than vector.)
transform :: [Double] -> [[Complex']]

--Fourier transformation
--Fourier transformation is inner product of complex number

--3. Compute inner dot between input and base function
transform a = iDot b c
    -- Let input complex data type
    where b = (atLeast2d . map (toComplex)) a
    -- get length of input, this is sampling frequency
          l = fromIntegral (length a) / 1.0
          c = (map (map (conjuguate)) . base) l
    --  prepare base function according to sampling frequency
    ----(Do not forget to take conjuguate; sin has negative from its definition.)
```

As you can see, Fourier transformation is defined by "iDot" which is dot product for complex number I defined as following and its argument is data itself, and base function.

function " Base " is defined by following.

```
 --base function preparation
  base  :: Double -> [[Complex']]

  --sub function of base generator
  base' :: (Double -> Double) -> Double -> [[Double]]

  --base function preparation
  base  a  = map (\s -> map (\(x,y) -> Complex' (x,y)) s ) d
    where b = base' cos' a
          c = base' sin a
          d = map2 (zip) b c


  --base function generation
  --sub function of base generator
  base' f a = map (\y -> map (\x -> f ( 2 * pi * x * y / a )) [0..b] )
              [0..b] where b = a - 1
```

And, "iDot" and "Conjuaguate" is defined by following.

```
data Complex' = Complex' (Double,Double) deriving (Show)

class MyComplex where

  --basic operation on complex space
  iAdd :: Complex' -> Complex' -> Complex'
  iSub :: Complex' -> Complex' -> Complex'
  iMul :: Complex' -> Complex' -> Complex'
  iDiv :: Complex' -> Complex' -> Complex'

  --inner product for complex number
  iDot :: [[Complex']] -> [[Complex']] -> [[Complex']]


  --change real to complex
  toComplex  :: Double -> Complex'

  --change real to complex
  toReal  :: Complex' -> Double


  --get conjuguate
  conjuguate :: Complex' -> Complex'



instance MyComplex where

  --basic operation on complex space
  iAdd (Complex' (k1,v1)) (Complex' (k2,v2)) = Complex' (k1+k2,v1+v2)
  iSub (Complex' (k1,v1)) (Complex' (k2,v2)) = Complex' (k1-k2,v1-v2)
  iMul (Complex' (k1,v1)) (Complex' (k2,v2)) = Complex' (k1*k2-v1*v2,k1*v2+k2*v1)
  iDiv (Complex' (k1,v1)) (Complex' (k2,v2)) = Complex' (k1/k2,v1/v2)

  --inner product
  iDot a b = group (length a) ( map ( foldl (iAdd) (Complex' (0,0)) )
     [ (map2 (iMul) x y)  | x <- a, y <- t])
     where t = t' a
```

```
--change real to complex
toComplex a = Complex' (a,0)

--change complex to real( just discard imaginary part)
toReal (Complex'(a,b)) = a

--take conjuguate
conjuguate (Complex'(a,b)) = Complex' (a,-b)
```

Since basic operation of complex number is distinct from operation for real number, I described comprehensive figure.

As you can see the definition of iDot, it is defined by just extension to complex number of dot product which I showed in first chapter.

If you note that Fourier transformation is unitary transformation, inverse Fourier transformation can be written in a way that we defined for Fourier transformation.

```
--Inverse Fourier transformation
--Input  : Coefficients of Fourier Series
--Output : one dimensional real number
invTransform :: [[Complex']] -> [Double]

--inverse Fourier transformation
--inverse Fourier transformation is almost identical with Fourier transformation
--Note

-- # Base do not need to take conjuguate from its definition.
-- # Do not forget multiply devide by length of input in the end

invTransform a = concatMap
   (map (\x -> (toReal . iDiv x) (Complex' (l,1)) )) (iDot b c)
   -- iDot again, but do not forget divide by sampling frequency
   --and let them real and flatten

   -- input needs to be transposed after Fourier transformation.
   where b = t' a
         l = fromIntegral (length a) / 1.0 -- get sampling frequency
         c = base l -- get base but do not conjuguate in this time
```

## 4.3 Fast Fourier transformation

Base matrix for Fourier transformation has regularity which came from frequency by its definition.

From matrix (1), considering constant frequency, $a = \cos(2n\pi) + a$ and $a = \sin(2n\pi) + a$

$$\begin{pmatrix} \cos(0) - i\sin(0) & \cos(0) - i\sin(0) & \cos(0) - i\sin(0) & \cos(0) - i\sin(0) \\ \cos(0) - i\sin(0) & \cos(\frac{1}{4}(2\pi)) - i\sin(\frac{1}{4}(2\pi)) & \cos(\frac{2}{4}(2\pi)) - i\sin(\frac{2}{4}(2\pi)) & \cos(\frac{3}{4}(2\pi)) - i\sin(\frac{3}{4}(2\pi)) \\ \cos(0) - i\sin(0) & \cos(\frac{2}{4}(2\pi)) - i\sin(\frac{2}{4}(2\pi)) & -\cos(0) + i\sin(0) & -\cos(\frac{2}{4}(2\pi)) + i\sin(\frac{2}{4}(2\pi)) \\ \cos(0) - i\sin(0) & \cos(\frac{3}{4}(2\pi)) - i\sin(\frac{3}{4}(2\pi)) & -\cos(\frac{2}{4}(2\pi)) + i\sin(\frac{2}{4}(2\pi)) & \cos(\frac{1}{4}(2\pi)) - i\sin(\frac{1}{4}(2\pi)) \end{pmatrix}$$

Given

$$\cos(0) - i\sin(0) = 1$$
$$\cos(\tfrac{1*\pi}{4}) - i\sin(\tfrac{1*\pi}{4}) = \tfrac{\sqrt{2}}{2}(1-i)$$
$$\cos(\tfrac{2*\pi}{4}) - i\sin(\tfrac{2*\pi}{4}) = -i$$
$$\cos(\tfrac{3*\pi}{4}) - i\sin(\tfrac{3*\pi}{4}) = -\tfrac{\sqrt{2}}{2}(1-i)$$

Actual value is

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \frac{\sqrt{2}}{2}(1-i) & -i & -\frac{\sqrt{2}}{2}(1-i) \\ 1 & -i & -1 & i \\ 1 & -\frac{\sqrt{2}}{2}(1-i) & i & \frac{\sqrt{2}}{2}(1-i) \end{pmatrix}$$

As you can see, base matrix has certain symmetrical structure in it.

if you swap 2nd column and 3rd column, the matrix can be divided into following two.

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -\frac{\sqrt{2}}{2}(1-i) \\ 1 & 0 & -i & 0 \\ 0 & 1 & 0 & \frac{\sqrt{2}}{2}(1-i) \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & -i & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -i \end{pmatrix}$$

Matrix are divided into succession of product of matrix whose size is half. By this operation, the number of multiplication of Fourier transformation is going to be reduced from O(N2) to $N \log(N)$. Specifically, if the Rank of base matrix is 4, times of multiplication decrease from $4 \times 4$ to $4 \log 4$ . Logarithm means succession of matrix inner product application forms tree structure, namely, first, multiply most frequent wavelength, and add less frequent wavelength later on. In this example, right matrix is application of coefficients given frequency is 2, and right matrix is application of coefficients given frequency is 4.

division to two matrix

$$\begin{pmatrix} 1 & 0 & \cos(0) - i\sin(0) & 0 \\ 0 & 1 & 0 & \cos(\frac{1*\pi}{4}) - i\sin(\frac{1*\pi}{4}) \\ 1 & 0 & \cos(\frac{2*\pi}{4}) - i\sin(\frac{2*\pi}{4}) & 0 \\ 0 & 1 & 0 & \cos(\frac{3*\pi}{4}) - i\sin(\frac{3*\pi}{4}) \end{pmatrix}$$

.

$$\begin{pmatrix} \cos(0) - i\sin(0) & \cos(0) - i\sin(0) & 0 & 0 \\ \cos(0) - i\sin(0) & \cos(\frac{2*\pi}{4}) - i\sin(\frac{2*\pi}{4}) & 0 & 0 \\ 0 & 0 & \cos(0) - i\sin(0) & \cos(0) - i\sin(0) \\ 0 & 0 & \cos(0) - i\sin(0) & \cos(\frac{2*\pi}{4}) - i\sin(\frac{2*\pi}{4}) \end{pmatrix}$$

## 4.4 Convolutional theory

This is the formalism when size of convolution is non-discrete and infinite.

$$f*g(x) \equiv \int_{-\infty}^{\infty} f(y)\, g(x-y) y \tag{4.5}$$

Convolution operation is translated as element wise multiplication after Fourier transformation of both equation.

$$\mathcal{F}[f*g] = \mathcal{F}[f]\, \mathcal{F}[g] \tag{4.6}$$

Following is the proof.

$$\begin{aligned}
\mathcal{F}[f*g] &= \int_{-\infty}^{\infty} \left( \int_{-\infty}^{\infty} f(y)\, g(x-y) dy \right) e^{-ikx} dx && (4.7)\\
&= \int_{-\infty}^{\infty} f(y) \left( \int_{-\infty}^{\infty} g(x-y)\, e^{-ikx} dx \right) dy && (4.8)\\
&= \int_{-\infty}^{\infty} f(y) \left( \int_{-\infty}^{\infty} g(t)\, e^{-ik(t+y)} dt \right) dy && (4.9)\\
&= \int_{-\infty}^{\infty} f(y) \left( \int_{-\infty}^{\infty} g(t)\, e^{-ikt} dt \right) e^{-iky} dy && (4.10)\\
&= \int_{-\infty}^{\infty} f(y) \mathcal{F}[g]\, e^{-iky} dy && (4.11)\\
&= \int_{-\infty}^{\infty} f(y)\, e^{-iky} dy\, \mathcal{F}[g] && (4.12)\\
&= \mathcal{F}[f]\, \mathcal{F}[g] && (4.13)
\end{aligned}$$

What is done here is
1st : Changing the order over integral of convolution and integral of Fourier transformation
2nd : Letting convolution inner product by transforming variables.
3rd : There you can see multiplication of two inner product.

To put it simply, Fourier transformation can be seen as an inner product between original function and base function.

Inner product between ( convolution A and B ) and base = multiplication ( inner product between input A and base ) ( inner product between B and base )

This transformation () is significantly important when you think about difference between convolution and normal inner product.

What should be noted here is Fourier transform after multiplication of two function is equivalent to convolution of both function after Fourier transformation.

$$\mathcal{F}[f]*\mathcal{F}[g] \;=\; 2\pi\, \mathcal{F}[fg] \tag{4.14}$$

because

$$\begin{aligned}
\mathcal{F}[fg] &= \int_{-\infty}^{\infty} f(x)\, g(x)\, e^{-ikx} dx\\
&= \int_{-\infty}^{\infty} f(x) \left[ \frac{1}{2\pi} \int_{-\infty}^{\infty} G(k') e^{ik'x} k' \right] e^{-ikx} dx\\
&= \frac{1}{2\pi} \int_{-\infty}^{\infty} G(k') \left[ \int_{-\infty}^{\infty} f(x)\, e^{-i(k-k')x} dx \right] dk'\\
&= \frac{1}{2\pi} \int_{-\infty}^{\infty} G(k')\, F(k-k') dk'\\
&= \frac{1}{2\pi} (G*F)(k)\\
&= \frac{1}{2\pi} \mathcal{F}[f]*\mathcal{F}[g] && (4.15)
\end{aligned}$$

The definition described here can be applied to combination of discrete Fourier transformation and discrete convolution which we encounter when considering convolutional neural network.

That is, convolution operation in original domain is expressed as multiplication in frequency domain. Whereas, multiplication operation such as passing through activation function is convolution operation in frequency domain.

## 4.5 Representation of signals after Fourier transformation

From the previous chapter, convolution operation is described as element-wise multiplication between Fourier transformation of data and Fourier transformation of a filter.

Before proceeding to the weight and filter analysis, let me introduce two functions (Delta function and Sinc function) which is useful to analyze weights and filters.

### 4.5.1 Delta function

When you think about mapping between original space and Fourier space, examples tell you a lot.
For instance, when you put data which has same values regardless of original dimension, how is it represented in Fourier space?
That is the case you have an image which has infinite resolution which is utterly black except in the midst which is white.

Given

$$\int_{-\infty}^{\infty} \delta(x) = 1$$

$$\delta(0) = \infty$$

$$\delta(x) = 0 (x \neq 0)$$

$$\int_{-\infty}^{\infty} \delta(x) e^{-ikx} x \ = \ e^0 \ = \ 1 \tag{4.16}$$

In the Fourier domain, Fourier transformation of delta function which has all same coefficient in spite of its terms. That means, it has same values in all of frequency. Inverse Fourier transformation operation of 1 turns out to be $\delta$ function, which I omit the proof.

Likewise, considering Fourier transformation pair, Fourier transformation of signal which has same values in all its domain, such as sheer white image, is going to be $\delta()$, and inverse Fourier transformation of $\delta()$ is 1 in frequency domain.

### 4.5.2 Sinc function

Suppose you let the resolution of previous example finite, what kind of signal is going to be appeared in frequency domain ?
As the image has just two values which is white or black, the image can be represented as combinations of

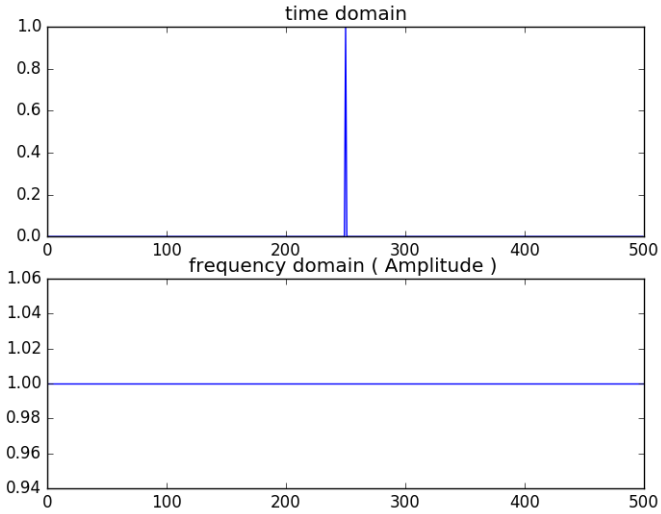


FIGURE 4.1: FFT of $\delta()$

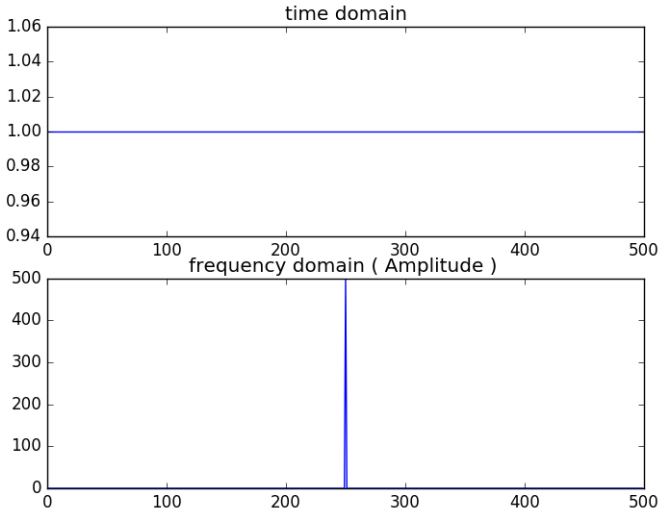


FIGURE 4.2: FFT of $\delta()$

square waves which heads for different directions.

One signal which pass through center of pixel can be described as

$$r_{-a,a}(t) = 1\ (-a \leq t \leq a)$$

$$r_{-a,a}(t) = 0\ (t < -a\ ||\ a < t)$$

which is called rectangular function.

Its Fourier transformation is

$$F[r_{-a,a}](t) = \int_{-\infty}^{\infty} r_{-a,a}(t)\mathrm{e}^{-ikt}dt$$
$$= \int_{-a}^{a} \mathrm{e}^{-ikt}dt$$

Calculating integral of natural logarithm

$$F[r_{-a,a}](t) = \left[\frac{1}{-it}\mathrm{e}^{-ikt}\right]_{-a}^{a}$$
$$= \frac{1}{ik}(\mathrm{e}^{ika} - \mathrm{e}^{-ika})$$

Making use of $sin(a)k = \frac{\mathrm{e}^{ika} - \mathrm{e}^{-ika}}{2i}$

$$F[r_{-a,a}](t) = \frac{2}{k}\frac{\mathrm{e}^{ika} - \mathrm{e}^{-ika}}{2i}$$
$$= \frac{1}{-ik}\sin(a)k \tag{25}$$

(25) tells you spectral of Frequency domain of square wave is multiplication of inverse proportion of frequency and sin function. Which is represented as follows,

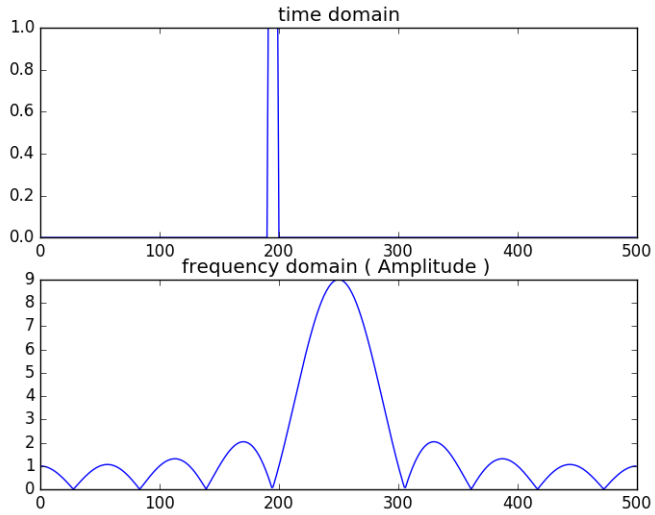


FIGURE 4.3: FFT of square wave

Note that direct component(k = 0) is not infinite unlike delta function, but

$$F[r_{-a,a}](t) \mid (t = 0) = \int_{-a}^{a} \mathrm{e}^{-i0x}dt$$
$$= \int_{-a}^{a} dt$$
$$= 2a$$

Figure (3) is the signal whose value is 1 in the range from 190 to 210, and rest of them is just 0.

Direct component means, for instance in image, average intensity of the pixel. If classification should be robust to general object,the difference of direct component should be out of consideration, in other words, acquire property of intensity invariancy.

By the way, the graph of Figure(3) represents only amplitude in the frequency domain which is provided by following definition.

$$| F(k) | = \sqrt{\cos(x)^2 + \sin(y)^2}$$

When you shift signal to right or left, it does not affect amplitude, but phase, which is defined by following.

$$I(k) = \tan^{-1}(\sin(y)/\cos(x))$$

This is because base function of Fourier series is represented as addition of $\cos$ and $i \sin$ and both can be either positive or negative. Since the derivatives of all four patterns of base functions which are ( $+\cos$ , $+\sin$ ,$-\cos$ , $-\sin$ ) can be circular relationship, difference between original signal and shifted signal whose frequency is same but phase is different would be each values of coefficient and stayed on each frequencies.

Thereby, when we consider only amplitude, translation invariance is assured in frequency domain. Figure4() is a shifted signal of Figure(3), bus shows same spectrum of Figure(4).

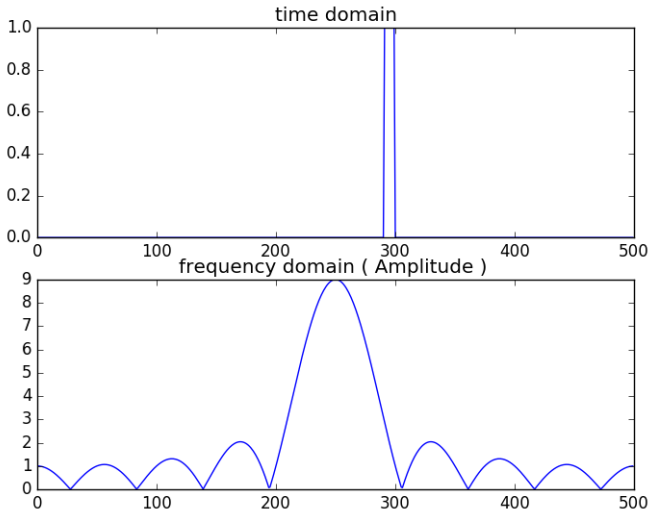


FIGURE 4.4: FFT of shifted square wave

Of course, if you set square wider, low frequency is going to be more dominant, given

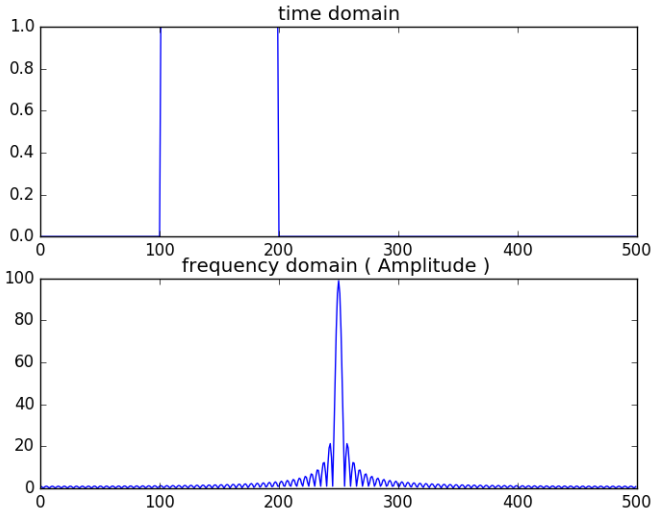


FIGURE 4.5: FFT of $\delta$

## 4.6 Frequency domain of filters

### 4.6.1 Filter size

Signals of speech or general object are pretty difficult to be analyzed due to its complexity. On the other hand, filters which is often used for CNN is by far interpretable because it is rather much smaller. In CNN, network has relatively small size such as 3 or 5 reveal good performance. These filters can be represented as Sinc function in frequency domain.

For instance, Sobel filter which calculates first differential in original domain is represented in frequency domain as

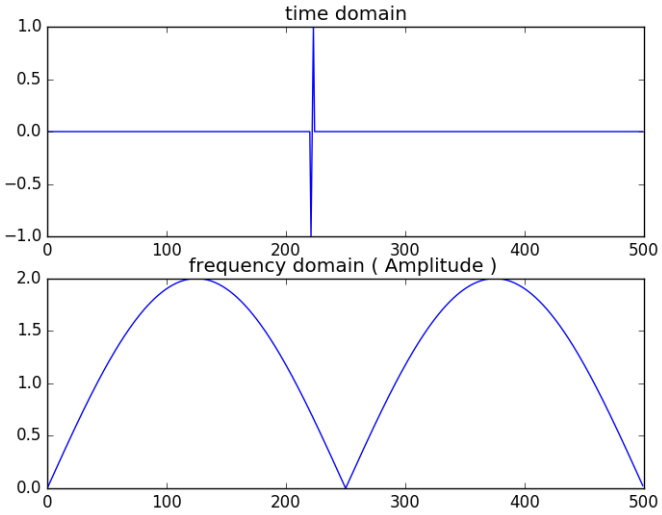


FIGURE 4.6: Sobel : signal (-1,1,-1)

That says, Sobel filter is working as bandpass filter.

Laplacian which is second derivative on original domain is defined as follows.

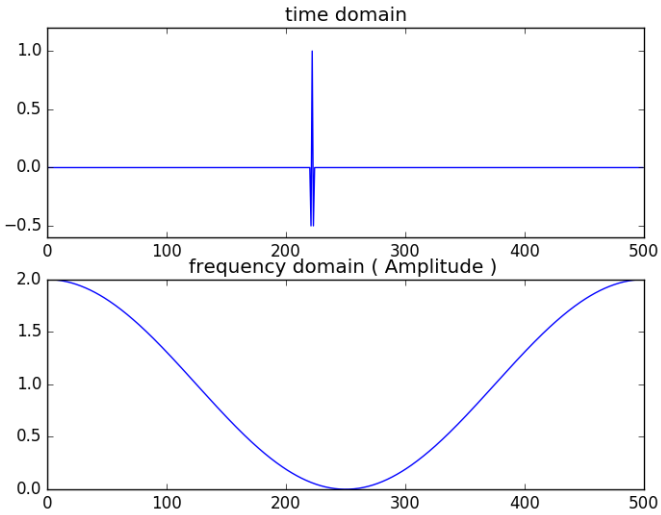


FIGURE 4.7: Laplacian : signal (-0.5,1,-0.5)

which is worked as low block filter or high pass filter.

This Sobel and Laplacian has filter whose size is 3. Sobel has (..,-1,0,1,..) Laplacian has (..,-0.5,1,-0.5,..) .. means rest of component which is 0.

Every filters including these two filters whose filter size is N can be represented as multiplication of coefficient $a$ and combination of rectangular function ( -1 $\leq a \leq$ 1)$whosesizeislessthanN.Formally,$

$$f(N,k,t) = \sigma n = 0 N r_{-n,n}(t) * a \tag{4.17}$$

where

$$r_{-n,n}(t) = 1 \ (-n \leq t \leq n)$$

$$r_{-n,n}(t) = 0 \ (t < -n \ || \ n < t)$$

Fourier transformation of $r_{-n,n}(t)$ is defined by (25), and since Fourier transformation is linear transformation, multiplication of k and addition inside sigma is preserved after transformation, hence,

$$F[r_{-n,n}](t) = \sigma n = 0 N \frac{1}{-i} \sin(n) a \tag{4.18}$$

From (), you will know frequency of this combination of Sinc function ( note "frequency in this context does not mean index of frequency domain after transformation" ) is determined by kernel size, not each values on filters, and its largest frequency is kernel size.

This means longer the filter size is, larger the frequency of Sinc function is. In other words, filter size determines complexity of spectrum in frequency domain. Considering why CNN which reveals best performance ever like [Residual net] has many layers but small filter size, it makes sense because each Sinc function in each layer can be regarded as sort of base function of overall transformation, and each base function does not need to be complex as itself, simple filter rather be able to capture more flexibility of different filters.

### 4.6.2 Fourier transformation of filters on which binary weights are put

Let us consider the representation of filters whose weights are binary [1,-1], 1D, and its size is 3.
Figure() represents Fourier transformation of binary weights. In this case, numbers of possible combinations are

$$2^3 \tag{4.19}$$

. But, some of them shares almost identical representation in frequency domain.

For instance, filters whose weights are [1,-1,1] has just subtle difference with [-1,1,-1] which just flipped positive/negative. Same things can be said about the one between [1,1,1] and [-1,-1,-1]. Non-symmetrical filter based on its middle such as [-1,-1,1] has equivalent representation with mirror-rotated [1,-1,-1].
If you observe its spectrum,

[1,1,1],[-1,-1,-1] low pass filter.
-1,-1
,[-1,-1,1],[1,1,-1],[-1,1,1] bandpass filter
-1,1
,[-1,1,-1] high pass filter

and its frequency of spectrum is defined by filter size as we saw in the formal section.

To conclude it, binary filters have only three roles which is low pass, bandpass, or high pass, and its combinations form more complex classifications.

### 4.6.3 Fourier transformation of binary input data

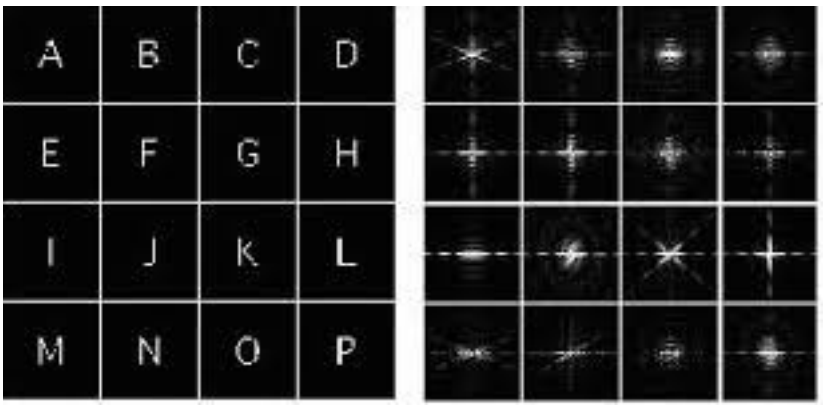


FIGURE 4.8: FFT of binary character

## 4.7 Binary character representation

By last section, we saw that translation and intensity invariance could be captured in frequency domain analyzing direct / alternating component and amplitude / phase component.

In this section, we will take a look at representation of binary character in frequency domain, and add more considerations towards it.

As you see Figure(6), different character has different spectrum in frequency domain. Following is the reasons why such a spectrum is given.

As an assumption, spectrum distribution on frequency domain which is shown on right side of Figure(6) represents amplitude, direct component is centorized, and symmetrical against a horizontal central line.

If there are straight lines in space domain, there might be corresponding line.

# Chapter 5

# Analysis of Binary neural network

# Bibliography


[1] Geoffrey E. Hinton Alex Krizhevsky Ilya Sutskever. “ImageNet Classification with Deep Convolutional Neural Networks”. In: (2012).

[2] M. Cole. “Algorithmic skeletons : A structured approach to the management of parallel computation. Research Monographs in Parallel and Distributed”. In: (1989).

[3] Ronald J. Williams David E. Rumelhart Geoffrey E. Hinton. “Learning representations by back-propagating errors”. In: (1986).

[4] Hiroshi Sawada. *comprehensive codes*. https://github.com/Hiroshi123.

[5] Daniel Soudry Ran El-Yaniv Yoshua Bengio Matthieu Courbariaux Itay Hubara. “Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 or 1”. In: (2016).

[6] Joseph Redmon Ali Farhadi Mohammad Rastegari Vicente Ordonez. “XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks”. In: (2016).

[7] Brian Kingsbury1 Bhuvana Ramabhadran1 Tara N. Sainath1 Abdel-rahman Mohamed2. “DEEP CONVOLUTIONAL NEURAL NETWORKS FOR LVCSR”. In: (2012).

[8] Brian Kingsbury1 Bhuvana Ramabhadran1 Tara N. Sainath1 Abdel-rahman Mohamed2. “Deep Convolutional Neural Networks for Sentiment Analysis of Short Texts”. In: (2014).