Generalized Convolutional Neural Network

Mikael Hedberg

July 7, 2015

1 Introduction

The purpose of this paper is to derive the equations for a generalized convolutional neural network in such in depth that one may easily implement it in C++ or CUDA / OpenCL. View this document as the theory and reference to the implementation. The big picture is to combine and write several state-of-the-art Machine Learning algorithms and see what kind of cool applications one could write. This is a small step in that direction.

2 Generalized CNN

Before jumping into the in depth calculations, let us start with a picture describing how different layers of the network could look like.

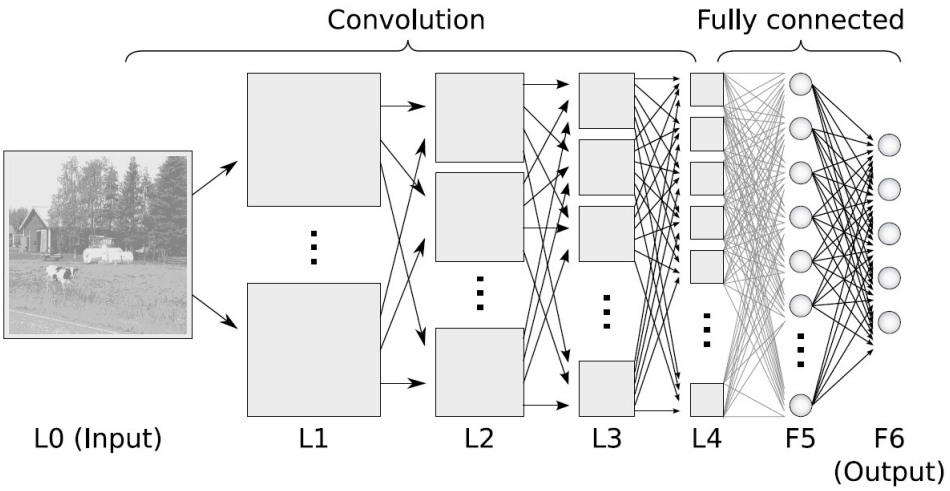

Figure 1: An example of a convolutional neural network [1]. L stands for layer, the boxes corresponds to two dimensional images (maps) and the circles are perceptrons.

With reference to Figure 1 let us define some useful notations:

3 Fundamental Equations

Layer index	l	$0 \leq l \leq L, \quad l, L \in \mathbb{N}$
The number of images in layer l	N^l	$N^l \in \mathbb{N}$
The i:th image in layer l	$\boldsymbol{Y}_i^{(l)}$	$1 \leq i \leq N^l$
Activation function of layer l	$\phi^{(l)}$	$\phi^{(l)} : \mathbb{R} \to \mathbb{R}$
Set of parameters in layer l	$\Omega^{(l)}$	$\{\omega_i \in \mathbb{R}\}$
Combinator function of image i at coordinate (x, y) in layer l	$z_{i,xy}^{(l)}(\boldsymbol{Y}_{1:N^{l-1}}^{(l-1)}, \Omega^{(l)})$	$z_{i,xy}^{(l)} : (\boldsymbol{Y}_{1:N^{l-1}}^{(l-1)}, \Omega^{(l)}) \to \mathbb{R}$
Image coordinates of image i in layer l	S_i^l	$S_i^l \subset \mathbb{N}$
Connected images in layer $l+1$ to map i in layer l	C_i^l	$C_i^l \subset \mathbb{N}$
Connected images in layer l-1 to map i in layer l	D_i^l	$D_i^l \subset \mathbb{N}$

Table 1: Definitions and notations

The network that we will work with in this paper is given by:

$$[\boldsymbol{Y}_i^{(l)}]_{xy} = \phi^{(l)}([\boldsymbol{Z}_i^{(l)}(\boldsymbol{Y}_{1:N^{l-1}}^{(l-1)}, \Omega^{(l)})]_{xy}) \tag{1}$$

Eq. 1 is very abstract and it supports different image sizes, arbitrary connections, shared parameters between units and even images that has random indexing. By deriving the gradient of this equation using back-propagation, we get a very general expression for all kinds of feed-forward networks. Observe that it can be easily expanded to indexing in any dimension, just by replacing (x, y).

Let us now consider a set of training data $T = \{(\boldsymbol{Y}_{1:N^0}^{(0)}, \boldsymbol{T}_{1:N^L})_t, 1 \leq t \leq N^T\}$. By defining an appropriate error function $r_t : (\boldsymbol{Y}_{1:N^L}^{(L)}, \boldsymbol{T}_{1:N^L})_t \to \mathbb{R}$ we may now try to minimize the total error by adjusting the parameters of the layers $\Omega^{(l)}$

$$E = \sum_{t=1}^{N^T} r_t(\boldsymbol{Y}_{1:N^L}^{(L)}, \boldsymbol{T}_{1:N^L})_t \tag{2}$$

From here on, we will omit the subscript t since it will only complicate the notation. For batches one may just add t onto all of the below variables and sum it. Observe that the notation $\Omega_m^{(k)}$ means the variable m in the set of parameters $\Omega^{(k)}$.

$$\frac{\partial r}{\partial \Omega_m^{(k)}} = \sum_{i=1}^{N^L} \sum_{(x,y)\in S_i^L} \frac{\partial r}{\partial [\boldsymbol{Y}_i^{(L)}]_{xy}} \frac{\partial [\boldsymbol{Y}_i^{(L)}]_{xy}}{\partial \Omega_m^{(k)}} \tag{3}$$

$$\frac{\partial [\boldsymbol{Y}_i^{(L)}]_{xy}}{\partial \Omega_m^{(k)}} = \phi^{(L)'}([\boldsymbol{Z}_i^{(L)}]_{xy}) \sum_{j=1}^{N^{L-1}} \sum_{(x_1,y_1)\in S_j^{L-1}} \frac{\partial [\boldsymbol{Z}_i^{(L)}]_{xy}}{\partial [\boldsymbol{Y}_j^{(L-1)}]_{x_1y_1}} \frac{\partial [\boldsymbol{Y}_i^{(L-1)}]_{x_1y_1}}{\partial \Omega_m^{(k)}} \tag{4}$$

In general we have the following relation between any two layers $\neq L$:

$$\frac{\partial [\boldsymbol{Y}_i^{(l)}]_{xy}}{\partial \Omega_m^{(k)}} = \phi^{(l)'}([\boldsymbol{Z}_i^{(l)}]_{xy}) \sum_{j=1}^{N^{l-1}} \sum_{(x_1,y_1)\in S_j^{l-1}} \frac{\partial [\boldsymbol{Z}_i^{(l)}]_{xy}}{\partial [\boldsymbol{Y}_j^{(l-1)}]_{x_1y_1}} \frac{\partial [\boldsymbol{Y}_j^{(l-1)}]_{x_1y_1}}{\partial \Omega_m^{(k)}} \tag{5}$$

From this we can define deltas that will simplify the gradient calculation.

$$[\boldsymbol{\delta}_i^{(L)}]_{xy} = \frac{\partial r}{\partial [\boldsymbol{Y}_i^{(L)}]_{xy}} \phi^{(L)'}([\boldsymbol{Z}_i^{(L)}]_{xy}) \tag{6}$$

$$[\boldsymbol{\delta}_j^{(l)}]_{xy} = \sum_{i=1}^{N^{l+1}} \sum_{(x_1,y_1)\in S_i^{l+1}} [\boldsymbol{\delta}_i^{(l+1)}]_{x_1y_1} \frac{\partial [\boldsymbol{Z}_i^{(l+1)}]_{x_1y_1}}{\partial [\boldsymbol{Y}_j^{(l)}]_{xy}} \phi^{(l)'}([\boldsymbol{Z}_j^{(l)}]_{xy}) \tag{7}$$

Finally, the gradient is given by:

$$\frac{\partial r}{\partial \Omega_m^{(k)}} = \sum_{i=1}^{N^k} \sum_{(x,y)\in S_i^k} [\boldsymbol{\delta}_i^{(k)}]_{xy} \frac{\partial [\boldsymbol{Z}_i^{(k)}]_{xy}}{\partial \Omega_m^{(k)}} \tag{8}$$

I call Eq. 1, 6, 7, 8 the fundamental feed-forward network equations since it defines the gradient and feed-forward relations of any known feed-forward network (known to the Author).

3.1 MLP

Let us recap the fundamental equations:

$$[\boldsymbol{Y}_i^{(l)}]_{xy} = \phi^{(l)}([\boldsymbol{Z}_i^{(l)}(\boldsymbol{Y}_{1:N^{l-1}}^{(l-1)}, \Omega^{(l)})]_{xy}) \tag{9}$$

$$[\boldsymbol{\delta}_i^{(L)}]_{xy} = \frac{\partial r}{\partial [\boldsymbol{Y}_i^{(L)}]_{xy}} \phi^{(L)'}([\boldsymbol{Z}_i^{(L)}]_{xy}) \tag{10}$$

$$[\boldsymbol{\delta}_j^{(l)}]_{xy} = \sum_{i=1}^{N^{l+1}} \sum_{(x_1,y_1)\in S_i^{l+1}} [\boldsymbol{\delta}_i^{(l+1)}]_{x_1y_1} \frac{\partial [\boldsymbol{Z}_i^{(l+1)}]_{x_1y_1}}{\partial [\boldsymbol{Y}_j^{(l)}]_{xy}} \phi^{(l)'}([\boldsymbol{Z}_j^{(l)}]_{xy}) \tag{11}$$

$$\frac{\partial r}{\partial \Omega_m^{(k)}} = \sum_{i=1}^{N^k} \sum_{(x,y)\in S_i^k} [\boldsymbol{\delta}_i^{(k)}]_{xy} \frac{\partial [\boldsymbol{Z}_i^{(k)}]_{xy}}{\partial \Omega_m^{(k)}} \tag{12}$$

For a fully connected layer such as MLP, we always have the possibility to reshape the images $\boldsymbol{Y}_i^{(l)}, \boldsymbol{Y}_j^{(l-1)}$ so that they are equivalent to vectors $\boldsymbol{y}^{(l)}, \boldsymbol{y}^{(l-1)}$.

Hence, the coordinate sets S_i^l, S_i^{l-1} contains only one coordinate and we may skip the (x, y) notation. We also have $\phi^{(l)}$ acting component-wise on the input. We may therefore simply the fundamental equations to:

$$y_i^{(l)} = \phi^{(l)}(z_i^{(l)}(y_{1:N^{l-1}}^{(l-1)}, \Omega^{(l)})) \tag{13}$$

$$\delta_i^{(L)} = \frac{\partial r}{\partial y_i^{(L)}} \phi^{(L)'}(z_i^{(L)}) \tag{14}$$

$$\delta_j^{(l)} = \sum_{i=1}^{N^{l+1}} \delta_i^{(l+1)} \frac{\partial z_i^{(l+1)}}{\partial y_j^{(l)}} \phi^{(l)'}(z_j^{(l)}) \tag{15}$$

$$\frac{\partial r}{\partial \Omega_m^{(k)}} = \sum_{i=1}^{N^k} \delta_i^{(k)} \frac{\partial z_i^{(k)}}{\partial \Omega_m^{(k)}} \tag{16}$$

Let us define $z_i^{(l)}$ by it's vector form:

$$\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)} \boldsymbol{y}^{(l-1)} + \boldsymbol{b}^{(l)} \tag{17}$$

Where $\boldsymbol{Y}_{1:N^{l-1}}^{(l-1)} = \boldsymbol{y}^{(l-1)}$ and $\Omega^{(l)} = \{\boldsymbol{W}^{(l)}, \boldsymbol{b}^{(l)}\}$. By looking at Eq. 17 component-wise we obtain the familiar form:

$$y_i^{(l)} = \phi^{(l)}(\sum_{j=1}^{N^{l-1}} \omega_{ij}^{(l)} y_j^{(l-1)} + b_i^{(l)}) \tag{18}$$

Which does indeed to look very similar to the component-wise Eq. 1.
In vector form we have

$$\boldsymbol{y}^{(l)} = \phi^{(l)}(\boldsymbol{W}^{(l)} \boldsymbol{y}^{(l-1)} + \boldsymbol{b}^{(l)}) \tag{19}$$

Let us now look at the remaining fundamental equations. We will look at two different output activations and derive the corresponding fundamental equations for this case. First with MSE and linear output which is useful when training a network for regression.

$$\phi^{(L)}(x) = x \tag{20}$$

$$r(\boldsymbol{t}, \boldsymbol{y}^{(L)}) = \frac{1}{2} \|\boldsymbol{t} - \boldsymbol{y}^{(L)}\|^2 \tag{21}$$

The fundamental equation for the output now transforms into

$$\delta_i^{(L)} = \frac{1}{2} \frac{\partial \|\boldsymbol{t} - \boldsymbol{y}^{(L)}\|^2}{\partial y_i^{(L)}} = -(t_i - y_i^{(L)}) \tag{22}$$

In vector notation

$$\boldsymbol{\delta}^{(L)} = -(\boldsymbol{t} - \boldsymbol{y}^{(L)}) \tag{23}$$

We will now consider a cross-entropy error function with soft-max activation. The problem is this case is that the activation function $\phi^{(l)}$ doesnt depend on the other inputs which is necessary for softmax activation. We will therefore set

$\phi^{(L)}$ to the identity transform and write the output in terms of $\boldsymbol{Z}_i^L(\boldsymbol{Y}_{1:N^L}^L, \Omega^{(L)})$ which can take any type of shape needed.

$$\phi^{(L)}(x) = x \tag{24}$$

$$z_i^L = \frac{\exp\{\sum_{j=1}^{N^{L-1}} \omega_{ij}^{(L)} y_j^{(L-1)} + b_i^{(L)}\}}{\sum_{k=1}^{N^L} \exp\{\sum_{j=1}^{N^{L-1}} \omega_{kj}^{(L)} y_j^{(L-1)} + b_k^{(L)}\}} \tag{25}$$

For a single binary classification task, the Cross-entropy function takes the form:

$$r(\boldsymbol{t}, \boldsymbol{y}^{(L)}) = -(t \log(y^{(L)}) + (1-t)\log(1-y^{(L)})) \tag{26}$$

Which also implies that we need to use a normal Sigmoid activation instead of the Softmax activation function. For multiple classes the CE function is given by

$$r(\boldsymbol{t}, \boldsymbol{y}^{(L)}) = -\sum_{i=0}^{N^L} t_i \log y_i^{(L)} \tag{27}$$

Let us now calculate the delta at the output layer for this case.

$$\delta_i^{(L)} = -\frac{\partial \sum_{j=0}^{N^L} t_j \log y_j^{(L)}}{\partial y_i^L} = -\frac{t_i}{y_i^{(L)}} \tag{28}$$

$$\tag{29}$$

However, with this change in $z_i^{(L)}$ extra care has to be taken to the derivative $\frac{\partial z_i^{(L)}}{\partial y_j^{(L-1)}}$. It can be shown that

$$\delta_i^{(L)} \frac{\partial z_i^{(L)}}{\partial y_j^{(L-1)}} = -(t_i - y_i^{(L)})\omega_{ij}^{(L)} \tag{30}$$

In any case, with one dimensional sigmoid and CE, softmax with CE and regression with linear equations. We obtain exactly the same update equations.

Remark 1. Note that $\delta_i^{(L)}$ in the case of a one dimensional variable with sigmoid $\phi^{(L)}$ we obtain the same equations as in the regression case. I.e. $\delta^{(L)} = -(t - y^{(L)})$.

A case that will not give the same update equations is if we use the MLE norm together with a non-trivial activation function at layer L.

$$\delta_i^{(L)} = \frac{1}{2}\frac{\partial \|\boldsymbol{t} - \boldsymbol{y}^{(L)}\|^2}{\partial y_i^{(L)}} = -(t_i - y_i^{(L)})\phi'^{(L)}(z_i^{(L)}) \tag{31}$$

This may also be expanded to the CE function if you carefully choose the output activation function $\phi^{(L)}(z_i^{(L)})$ so that it has positive or zero values.

$$\delta_i^{(L)} = -\frac{t_i - y_i^{(L)}}{y_i^{(L)}(1 - y_i^{(L)})}\phi'^{(L)}(z_i^{(L)}) \tag{32}$$

For a layer not equal to the output layer, we have the following expression (where the activation function is normally a tanh or a sigmoid.)

$$z_i^{(l)} = \sum_{j=1}^{N^{l-1}} \omega_{ij}^{(l)} y_j^{(l-1)} + b_i^{(l)} \tag{33}$$

$$\delta_i^{(l)} = \sum_{j=1}^{N^{l+1}} \omega_{ji}^{(l+1)} \delta_j^{(l+1)} \phi'^{(l)}(z_i^{(l)}) \tag{34}$$

Or in vector notation

$$\boldsymbol{\delta}^{(l)} = \boldsymbol{W}^{(l+1)T} \boldsymbol{\delta}^{(l+1)} \circ \phi'^{(l)}(\boldsymbol{z}^{(l)}) \tag{35}$$

Before ending this section, let us finish by calculating the gradient of this layer

$$\frac{\partial r}{\partial \Omega_m^{(l)}} = \sum_{i=1}^{N^l} \delta_j^{(l)} \frac{\partial z_i^{(l)}}{\partial \Omega_m^{(l)}} \tag{36}$$

$$\frac{\partial r}{\partial \omega_{kj}^{(l)}} = \sum_{i=1}^{N^l} \delta_j^{(l)} \frac{\partial \sum_{m=1}^{N^{l-1}} \omega_{im}^{(l)} y_m^{(l-1)} + b_i^{(l)}}{\partial \omega_{kj}^{(l)}} \tag{37}$$

$$\frac{\partial r}{\partial \omega_{kj}^{(l)}} = \delta_k^{(l)} y_j^{(l-1)} \tag{38}$$

$$\frac{\partial r}{\partial b_k^{(l)}} = \delta_k^{(l)} \tag{39}$$

In vector form

$$\Delta \boldsymbol{W}^{(l)} = \boldsymbol{\delta}^{(l)} \boldsymbol{y}^{(l-1)T} \tag{40}$$

$$\Delta \boldsymbol{b}^{(l)} = \boldsymbol{\delta}^{(l)} \tag{41}$$

The important equations are summarized in the below tables:

Error function	$r(\boldsymbol{t}, \boldsymbol{y}^{(L)})$
CE, binary	$-(t \log(y^{(L)}) + (1-t) \log(1-y^{(L)}))$
CE, multiple classes	$-\sum_{i=0}^{N^L} t_i \log y_i^{(L)}$
MSE	$\frac{1}{2} \|\boldsymbol{t} - \boldsymbol{y}^{(L)}\|^2$

Table 2: Common error functions.

(Error, Activation)	$\boldsymbol{\delta}^{(L)}$
(CE, Softmax)	$-(\boldsymbol{t} - \boldsymbol{y}^{(L)})$
(CE - binary, Sigmoid)	$-(t - y^{(L)})$
(MSE, Linear)	$-(\boldsymbol{t} - \boldsymbol{y}^{(L)})$
(MSE, Any)	$-(\boldsymbol{t}_i - \boldsymbol{y}_i^{(L)}) \circ \phi'^{(L)}(\boldsymbol{z}_i^{(L)})$
(CE, Any)	$-\boldsymbol{t} \circ \frac{1}{\boldsymbol{y}^{(L)}} \circ \phi'^{(L)}(\boldsymbol{z}_i^{(L)})$
(CE - binary, Any)	$-\frac{t-y^{(L)}}{y^{(L)}(1-y^{(L)})} \phi'^{(L)}(z^{(L)})$

Table 3: Output deltas for different activations and error functions.

Feed forward	$\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)}\boldsymbol{y}^{(l-1)} + \boldsymbol{b}^{(l)}$
Feed forward	$\boldsymbol{y}^{(l)} = \phi^{(l)}(\boldsymbol{z}^{(l)})$
Back propagation	$\boldsymbol{\delta}^{(L)}$ by Table 3.1
Back propagation	$\boldsymbol{\delta}^{(l)} = \boldsymbol{W}^{(l+1)T}\boldsymbol{\delta}^{(l+1)} \circ \phi^{'(l)}(\boldsymbol{z}^{(l)})$
Weight update	$\Delta\boldsymbol{W}^{(l)} = \boldsymbol{\delta}^{(l)}\boldsymbol{y}^{(l-1)T}$
Bias update	$\Delta\boldsymbol{b}^{(l)} = \boldsymbol{\delta}^{(l)}$

Table 4: Fundamental MLP equations.

3.2 Radial basis layer

Let use once again recall the fundamental equations.

$$[\boldsymbol{Y}_i^{(l)}]_{xy} = \phi^{(l)}([\boldsymbol{Z}_i^{(l)}(\boldsymbol{Y}_{1:N^{l-1}}^{(l-1)}, \Omega^{(l)})]_{xy}) \tag{42}$$

$$[\boldsymbol{\delta}_i^{(L)}]_{xy} = \frac{\partial r}{\partial [\boldsymbol{Y}_i^{(L)}]_{xy}} \phi^{(L)'}([\boldsymbol{Z}_i^{(L)}]_{xy}) \tag{43}$$

$$[\boldsymbol{\delta}_j^{(l)}]_{xy} = \sum_{i=1}^{N^{l+1}} \sum_{(x_1,y_1)\in S_i^{l+1}} [\boldsymbol{\delta}_i^{(l+1)}]_{x_1y_1} \frac{\partial [\boldsymbol{Z}_i^{(l+1)}]_{x_1y_1}}{\partial [\boldsymbol{Y}_j^{(l)}]_{xy}} \phi^{(l)'}([\boldsymbol{Z}_j^{(l)}]_{xy}) \tag{44}$$

$$\frac{\partial r}{\partial \Omega_m^{(k)}} = \sum_{i=1}^{N^k} \sum_{(x,y)\in S_i^k} [\boldsymbol{\delta}_i^{(k)}]_{xy} \frac{\partial [\boldsymbol{Z}_i^{(k)}]_{xy}}{\partial \Omega_m^{(k)}} \tag{45}$$

Once again as for the MLP, if we have image inputs we may always reshape the maps into vectors. We may therefore omit the subscript (x, y). Yielding the below fundamental equations:

$$y_i^{(l)} = \phi^{(l)}(z_i^{(l)}(y_{1:N^{l-1}}^{(l-1)}, \Omega^{(l)})) \tag{46}$$

$$\delta_i^{(L)} = \frac{\partial r}{\partial y_i^{(L)}} \phi^{(L)'}(z_i^{(L)}) \tag{47}$$

$$\delta_j^{(l)} = \sum_{i=1}^{N^{l+1}} \delta_i^{(l+1)} \frac{\partial z_i^{(l+1)}}{\partial y_j^{(l)}} \phi^{(l)'}(z_j^{(l)}) \tag{48}$$

$$\frac{\partial r}{\partial \Omega_m^{(k)}} = \sum_{i=1}^{N^k} \delta_i^{(k)} \frac{\partial z_i^{(k)}}{\partial \Omega_m^{(k)}} \tag{49}$$

For a radial basis function, the activation doesn't really make sense to use the separation between activation and combinator function. We will thus always regard $\phi^{(l)}(x) = x$ as the identity map. We may therefore simplify the fundamental equations further.

$$y_i^{(l)} = z_i^{(l)}(y_{1:N^{l-1}}^{(l-1)}, \Omega^{(l)}) \tag{50}$$

$$\delta_i^{(L)} = \frac{\partial r}{\partial y_i^{(L)}} \tag{51}$$

$$\delta_j^{(l)} = \sum_{i=1}^{N^{l+1}} \delta_i^{(l+1)} \frac{\partial z_i^{(l+1)}}{\partial y_j^{(l)}} \tag{52}$$

$$\frac{\partial r}{\partial \Omega_m^{(k)}} = \sum_{i=1}^{N^k} \delta_i^{(k)} \frac{\partial z_i^{(k)}}{\partial \Omega_m^{(k)}} \tag{53}$$

For a general radial basis layer, we usually set $z_i^{(l)}(y_{1:N^{l-1}}^{(l-1)}, \Omega^{(l)}) = \rho(\|y_{1:N^{l-1}}^{(l-1)} - \boldsymbol{c}_i^{(l)}\|)$. Which in practice normally looks like:

$$z_i^{(l)}(y_{1:N^{l-1}}^{(l-1)}, \Omega^{(l)}) = \exp\{-\beta_i \|y_{1:N^{l-1}}^{(l-1)} - \boldsymbol{c}_i^{(l)}\|\} \tag{54}$$

The parameter set thus consists of $\Omega^{(l)} = \{(\boldsymbol{c}_i^{(l)}, \beta_i)\}$. If one is interested in using RBF:s as outputs, a common approach is to linearly combine the RBF:s at the output layer. This in turn may be viewed as

$$z_i^{(L)}(y_{1:N^{L-1}}^{(L-1)}, \Omega^{(L)}) = \sum_{j \in D_i^L} a_{ij} \rho(\|y_{1:N^{l-1}}^{(l-1)} - \boldsymbol{c}_i^{(l)}\|) \tag{55}$$

Where D_i^L is as indicated in the notation's table. The connected images to the node i. Normally, it's fully connected. The parameter set of the last layer is thus $\Omega^{(l)} = \{a_{ij}\}$.

3.3 Convolution Layer

As usual, always start with the fundamental equations:

$$[\boldsymbol{Y}_i^{(l)}]_{xy} = \phi^{(l)}([\boldsymbol{Z}_i^{(l)}(\boldsymbol{Y}_{1:N^{l-1}}^{(l-1)}, \Omega^{(l)})]_{xy}) \tag{56}$$

$$[\boldsymbol{\delta}_i^{(L)}]_{xy} = \frac{\partial r}{\partial [\boldsymbol{Y}_i^{(L)}]_{xy}} \phi^{(L)'}([\boldsymbol{Z}_i^{(L)}]_{xy}) \tag{57}$$

$$[\boldsymbol{\delta}_j^{(l)}]_{xy} = \sum_{i=1}^{N^{l+1}} \sum_{(x_1,y_1) \in S_i^{l+1}} [\boldsymbol{\delta}_i^{(l+1)}]_{x_1 y_1} \frac{\partial [\boldsymbol{Z}_i^{(l+1)}]_{x_1 y_1}}{\partial [\boldsymbol{Y}_j^{(l)}]_{xy}} \phi^{(l)'}([\boldsymbol{Z}_j^{(l)}]_{xy}) \tag{58}$$

$$\frac{\partial r}{\partial \Omega_m^{(k)}} = \sum_{i=1}^{N^k} \sum_{(x,y) \in S_i^k} [\boldsymbol{\delta}_i^{(k)}]_{xy} \frac{\partial [\boldsymbol{Z}_i^{(k)}]_{xy}}{\partial \Omega_m^{(k)}} \tag{59}$$

This is the first time that the coordinates will come in handy. Observe however that we will omit the residual equations (i.e. $[\boldsymbol{\delta}_i^{(L)}]_{xy}$) for a convolutional layer since it will not be used immediately. However, it is very straight-forward to derive for an interested reader.

Let us start off by defining

$$[\boldsymbol{Z}_i^{(l)}(\boldsymbol{Y}_{1:N^{l-1}}^{(l-1)}, \Omega^{(l)})]_{xy} = \sum_{j \in D_i^l} (\boldsymbol{Y}_j^{(l-1)} \star \boldsymbol{\omega}_{ij}^{(l)})(x, y) + b_i^{(l)} \tag{60}$$

$$(\boldsymbol{Y}_j^{(l-1)} \star \boldsymbol{\omega}_{ij}^{(l)})(x, y) = \sum_{u=0}^{M_u} \sum_{v=0}^{M_v} [\boldsymbol{Y}_j^{(l-1)}]_{x+u, y+v} \cdot [\boldsymbol{\omega}_{ij}^{(l)}]_{u,v} \tag{61}$$

Where M_u, M_v is the width and height of the kernel $\boldsymbol{\omega}_{ij}^{(l)}$. Furthermore, we have the parameters $\Omega^{(l)} = \{\boldsymbol{\omega}_{ij}^{(l)}, \boldsymbol{b}^{(l)}\}$.

Remark 2. Observe that in most cases $\boldsymbol{\omega}_{ij}^{(l)} = \boldsymbol{\omega}_{i}^{(l)}$

Remark 3. Observe that there's nothing limiting us at the moment to use different sizes of the kernels in the same layer. However, practically this is rarely used.

Remark 4. In practical applications where sub-sampling is used, one may in some cases simply sub-sample by taking every k:th coordinate in the x,y directions. Instead of convolving the entire image and then sub-sample every k:th pixel, we may put the sub-sampling operation directly in the convolution operation. Thus yielding:

$$(\boldsymbol{Y}_j^{(l-1)} \star \boldsymbol{\omega}_{ij}^{(l)})(x,y) = \sum_{u=0}^{M_u}\sum_{v=0}^{M_v}[\boldsymbol{Y}_j^{(l-1)}]_{kx+u,ky+v} \cdot [\boldsymbol{\omega}_{ij}^{(l)}]_{u,v} \tag{62}$$

We will now turn to the interesting part of the convolutional layer. That is, how do we calculate the deltas?

$$\frac{\partial[\boldsymbol{Z}_i^{(l+1)}(\boldsymbol{Y}_{1:N^{l-1}}^{(l)}, \Omega^{(l+1)})]_{xy}}{\partial[\boldsymbol{Y}_j^{(l)}]_{x_1y_1}} = \frac{\partial \sum_{k\in D_i^{l+1}}(\boldsymbol{Y}_k^{(l)} \star \boldsymbol{\omega}_{ik}^{(l+1)})(x,y) + b_i^{(l)}}{\partial[\boldsymbol{Y}_j^{(l)}]_{x_1y_1}} = \tag{63}$$

$$\frac{\partial \sum_{u=0}^{M_u}\sum_{v=0}^{M_v}[\boldsymbol{Y}_j^{(l)}]_{x+u,y+v} \cdot [\boldsymbol{\omega}_{ij}^{(l+1)}]_{u,v}}{\partial[\boldsymbol{Y}_j^{(l)}]_{x_1y_1}}\mathbb{1}_{\{j\in D_i^{l+1}\}} = \tag{64}$$

$$\sum_{u=0}^{M_u}\sum_{v=0}^{M_v}\mathbb{1}_{\{x_1=x+u,y_1=y+v\}} \cdot [\boldsymbol{\omega}_{ij}^{(l+1)}]_{u,v}\mathbb{1}_{\{j\in D_i^{l+1}\}} \tag{65}$$

This latest function is pretty hard to get an intuition from on its own. Therefore, let us put it into the right context.

$$[\boldsymbol{\delta}_j^{(l)}]_{xy} = \sum_{i=1}^{N^{l+1}}\sum_{(x_1,y_1)\in S_i^{l+1}}[\boldsymbol{\delta}_i^{(l+1)}]_{x_1y_1}\frac{\partial[\boldsymbol{Z}_i^{(l+1)}]_{x_1y_1}}{\partial[\boldsymbol{Y}_j^{(l)}]_{xy}}\phi^{(l)'}([\boldsymbol{Z}_j^{(l)}]_{xy}) \tag{66}$$

$$[\boldsymbol{\delta}_j^{(l)}]_{xy} = \tag{67}$$

$$\sum_{i=1}^{N^{l+1}}\sum_{(x_1,y_1)\in S_i^{l+1}}[\boldsymbol{\delta}_i^{(l+1)}]_{x_1y_1}\sum_{u=0}^{M_u}\sum_{v=0}^{M_v}\mathbb{1}_{\{x=x_1+u,y=y_1+v\}} \cdot [\boldsymbol{\omega}_{ij}^{(l+1)}]_{u,v}\mathbb{1}_{\{j\in D_i^{l+1}\}}\phi^{(l)'}([\boldsymbol{Z}_j^{(l)}]_{xy}) \tag{68}$$

The above expression looks horrible, so let us take some time to analyze it.

$$\sum_{(x_1,y_1)\in S_i^{l+1}}[\boldsymbol{\delta}_i^{(l+1)}]_{x_1y_1}\sum_{u=0}^{M_u}\sum_{v=0}^{M_v}\mathbb{1}_{\{x=x_1+u,y=y_1+v\}} \cdot [\boldsymbol{\omega}_{ij}^{(l+1)}]_{u,v} \tag{69}$$

What does this equation really mean? Imagine that we have performed the summation over all (x_1, y_1). The only terms that will survive are the one who's given by $(x = x_1 + u, y = y_1 + v)$. This is true, but one should be careful to perform the variable substitution directly, since we have to observe that (x, y) is defined on S_j^l whereas (x_1, y_1) is defined on S_i^{l+1}. By performing the summation and substitute $(x_1 = x - u, y_1 = y - v)$, we may have for some combination of (x, y, u, v) that $(x_1 = x-u, y_1 = y-v) \notin S_i^{l+1}$. Therefore, when performing the summation we have to take this into account. Eq. 69 is therefore transformed into:

$$\sum_{u=0}^{M_u}\sum_{v=0}^{M_v}[\boldsymbol{\delta}_i^{(l+1)}]_{x-u,y-v}[\boldsymbol{\omega}_{ij}^{(l+1)}]_{u,v}\mathbb{1}_{\{(x-u,y-v)\in S_i^{l+1}\}} \tag{70}$$

If you were a mathematician you would probably already scream about all the omitted details. But above is actually incorrect in the sense that $[\boldsymbol{\delta}_i^{(l+1)}]_{x-u,y-v}$ may not even be defined. But on those values, we kind of indicate that with the $\mathbb{1}$ function. So imagine that you expand the size of $\boldsymbol{\delta}_i^{(l+1)}$ if you want to be rigorous.

Let us now turn to the implementor's perspective. A very simple method of not hassling with the boundaries is to allow expansion of $\boldsymbol{\delta}_i^{(l+1)}$ so that we have zeros located on the coordinates where $\mathbb{1}_{\{(x-u,y-v)\in S_i^{l+1}\}}$ doesn't hold. Let us denote this delta by $\tilde{\boldsymbol{\delta}}_i^{(l+1)}$. And we obtain

$$\sum_{u=0}^{M_u}\sum_{v=0}^{M_v}[\tilde{\boldsymbol{\delta}}_i^{(l+1)}]_{x-u,y-v}[\boldsymbol{\omega}_{ij}^{(l+1)}]_{u,v} \tag{71}$$

Furthermore, we observe that convolution defined by 71 is actually equivalent to convolution previously defined by simply rotating the kernel $\boldsymbol{\omega}_{ij}^{(l+1)}$ by 180 degrees. Let us denote this kernel by $\tilde{\boldsymbol{\omega}}_{ij}^{(l+1)}$.

$$\sum_{u=0}^{M_u}\sum_{v=0}^{M_v}[\tilde{\boldsymbol{\delta}}_i^{(l+1)}]_{x-u,y-v}[\boldsymbol{\omega}_{ij}^{(l+1)}]_{u,v} = (\tilde{\boldsymbol{\delta}}_i^{(l+1)} \star \tilde{\boldsymbol{\omega}}_{ik}^{(l+1)})(x, y) \tag{72}$$

Eq. 68 is now simplified to:

$$[\boldsymbol{\delta}_j^{(l)}]_{xy} = \sum_{i=1}^{N^{l+1}}(\tilde{\boldsymbol{\delta}}_i^{(l+1)} \star \tilde{\boldsymbol{\omega}}_{ij}^{(l+1)})(x, y)\mathbb{1}_{\{j\in D_i^{l+1}\}}\phi^{(l)'}([\boldsymbol{Z}_j^{(l)}]_{xy}) \tag{73}$$

By noting that $\sum_{i=1}^{N^{l+1}}\mathbb{1}_{\{j\in D_i^{l+1}\}}$ is actually equivalent to all connected images from j in l to $l+1$. Eq. 73 is therefore transformed into:

$$[\boldsymbol{\delta}_j^{(l)}]_{xy} = \sum_{i\in C_j^l}(\tilde{\boldsymbol{\delta}}_i^{(l+1)} \star \tilde{\boldsymbol{\omega}}_{ij}^{(l+1)})(x, y)\phi^{(l)'}([\boldsymbol{Z}_j^{(l)}]_{xy}) \tag{74}$$

Remark 5. Let us continue on Remark 4 which optimizes a combined convolution and sub-sampling operation by doing it simultaneously. In this case Eq. 69 takes the form

$$\sum_{(x_1,y_1)\in S_i^{l+1}}[\boldsymbol{\delta}_i^{(l+1)}]_{x_1y_1}\sum_{u=0}^{M_u}\sum_{v=0}^{M_v}\mathbb{1}_{\{x=kx_1+u,y=ky_1+v\}}\cdot[\boldsymbol{\omega}_{ij}^{(l+1)}]_{u,v} \tag{75}$$

The coordinate change is not as straight forward in this case since we are dealing with integer equations and they may not be directly divided. However, this case can be handled rather neatly just by using vanilla sub-sampling and discard calculating unnecessary coordinates.

Lastly, let us now derive the equations for weight updates.

$$\frac{\partial r}{\partial [\boldsymbol{\omega}_{pq}^{(l)}]_{u,v}} = \sum_{i=1}^{N^l} \sum_{(x,y)\in S_i^l} [\boldsymbol{\delta}_i^{(l)}]_{xy} \frac{\partial [\boldsymbol{Z}_i^{(l)}]_{xy}}{\partial [\boldsymbol{\omega}_{pq}^{(l)}]_{u,v}} \tag{76}$$

$$[\boldsymbol{Z}_i^{(k)}]_{xy} = \sum_{j\in D_i^l} \sum_{m=0}^{M_u} \sum_{n=0}^{M_v} [\boldsymbol{Y}_j^{(l-1)}]_{x+m,y+n} \cdot [\boldsymbol{\omega}_{ij}^{(l)}]_{m,n} + b_i^{(l)} \tag{77}$$

By inserting Eq. 77 in Eq. 76. We obtain

$$\frac{\partial r}{\partial [\boldsymbol{\omega}_{pq}^{(l)}]_{u,v}} = \sum_{i=1}^{N^l} \sum_{(x,y)\in S_i^l} [\boldsymbol{\delta}_i^{(l)}]_{xy} \sum_{j\in D_i^l} \sum_{m=0}^{M_u} \sum_{n=0}^{M_v} [\boldsymbol{Y}_j^{(l-1)}]_{x+m,y+n} \cdot \frac{\partial [\boldsymbol{\omega}_{ij}^{(l)}]_{m,n}}{\partial [\boldsymbol{\omega}_{pq}^{(l)}]_{u,v}} \tag{78}$$

As before, $\sum_{i=1}^{N^{l+1}} \mathbb{1}_{\{j\in D_i^{l+1}\}}$ is equivalent to all connected images from j in l to $l+1$.

$$\frac{\partial r}{\partial [\boldsymbol{\omega}_{pq}^{(l)}]_{u,v}} = \sum_{(x,y)\in S_p^l} [\boldsymbol{\delta}_p^{(l)}]_{xy} [\boldsymbol{Y}_q^{(l-1)}]_{x+u,y+v} \tag{79}$$

And for the more common case where we don't distinguish the kernel between the connections, we obtain

$$\frac{\partial r}{\partial [\boldsymbol{\omega}_{i}^{(l)}]_{u,v}} = \sum_{(x,y)\in S_i^l} \sum_{j\in C_i^l} [\boldsymbol{\delta}_i^{(l)}]_{xy} [\boldsymbol{Y}_j^{(l-1)}]_{x+u,y+v} \tag{80}$$

For the bias we obtain

$$\frac{\partial r}{\partial b_i^{(l)}} = \sum_{(x,y)\in S_i^l} [\boldsymbol{\delta}_i^{(l)}]_{xy} \tag{81}$$

Let us now recap all the important equations.

Convolution Definition	$(\boldsymbol{Y}_j^{(l-1)} \star \boldsymbol{\omega}_{ij}^{(l)})(x,y) = \sum_{u=0}^{M_u}\sum_{v=0}^{M_v}[\boldsymbol{Y}_j^{(l-1)}]_{x+u,y+v} \cdot [\boldsymbol{\omega}_{ij}^{(l)}]_{u,v}$
Feed forward	$[\boldsymbol{Y}_i^{(l)}]_{xy} = \phi^{(l)}(\sum_{j\in D_i^l}(\boldsymbol{Y}_j^{(l-1)} \star \boldsymbol{\omega}_{ij}^{(l)})(x,y))$
Delta padding	$\tilde{\boldsymbol{\delta}}_i^{(l+1)}$ by padding $\mathbb{1}_{\{(x-u,y-v)\in S_i^{l+1}\}}$, which corresponds to a padding of size M_u, M_v around $\boldsymbol{\delta}_i^{(l+1)}$.
Kernel rotation	$\tilde{\boldsymbol{\omega}}_{ij}^{(l+1)} = rot180(\boldsymbol{\omega}_{ij}^{(l+1)})$
Back propagation	$[\boldsymbol{\delta}_j^{(l)}]_{xy} = \sum_{i\in C_j^l}(\tilde{\boldsymbol{\delta}}_i^{(l+1)} \star \tilde{\boldsymbol{\omega}}_{ij}^{(l+1)})(x,y)\phi^{(l)'}([\boldsymbol{Z}_j^{(l)}]_{xy})$
Kernel update with multiple kernels per map	$\frac{\partial r}{\partial[\boldsymbol{\omega}_{pq}^{(l)}]_{u,v}} = \sum_{(x,y)\in S_p^l}[\boldsymbol{\delta}_p^{(l)}]_{xy}[\boldsymbol{Y}_q^{(l-1)}]_{x+u,y+v}$
Kernel update with one kernel per map	$\frac{\partial r}{\partial[\boldsymbol{\omega}_i^{(l)}]_{u,v}} = \sum_{(x,y)\in S_i^l}\sum_{j\in C_i^l}[\boldsymbol{\delta}_i^{(l)}]_{xy}[\boldsymbol{Y}_j^{(l-1)}]_{x+u,y+v}$
Bias update	$\frac{\partial r}{\partial b_i^{(l)}} = \sum_{(x,y)\in S_i^l}[\boldsymbol{\delta}_i^{(l)}]_{xy}$

Table 5: Fundamental convolution equations.

3.4 Pooling and Sub Sampling

The purpose of performing pooling or sub-sampling is to reduce the dimensionality of the problem. As usual, always start with the fundamental equations:

$$[\boldsymbol{Y}_i^{(l)}]_{xy} = \phi^{(l)}([\boldsymbol{Z}_i^{(l)}(\boldsymbol{Y}_{1:N^{l-1}}^{(l-1)}, \Omega^{(l)})]_{xy}) \tag{82}$$

$$[\boldsymbol{\delta}_i^{(L)}]_{xy} = \frac{\partial r}{\partial[\boldsymbol{Y}_i^{(L)}]_{xy}}\phi^{(L)'}([\boldsymbol{Z}_i^{(L)}]_{xy}) \tag{83}$$

$$[\boldsymbol{\delta}_j^{(l)}]_{xy} = \sum_{i=1}^{N^{l+1}}\sum_{(x_1,y_1)\in S_i^{l+1}}[\boldsymbol{\delta}_i^{(l+1)}]_{x_1y_1}\frac{\partial[\boldsymbol{Z}_i^{(l+1)}]_{x_1y_1}}{\partial[\boldsymbol{Y}_j^{(l)}]_{xy}}\phi^{(l)'}([\boldsymbol{Z}_j^{(l)}]_{xy}) \tag{84}$$

$$\frac{\partial r}{\partial\Omega_m^{(k)}} = \sum_{i=1}^{N^k}\sum_{(x,y)\in S_i^k}[\boldsymbol{\delta}_i^{(k)}]_{xy}\frac{\partial[\boldsymbol{Z}_i^{(k)}]_{xy}}{\partial\Omega_m^{(k)}} \tag{85}$$

In this section we will consider three types of sub-sampling and pooling layers. Let us start with sub-sampling as defined by LeCun. Also observe that we will not consider the $\boldsymbol{\delta}_i^{(L)}$ since pooling will very rarely be located on the output layer.

3.4.1 Weighed sub-sampling

Let us begin by defining the sub-sampling regime.

$$[\boldsymbol{Z}_i^{(l)}(\boldsymbol{Y}_{1:N^{l-1}}^{(l-1)}, \Omega^{(l)})]_{xy} = \sum_{j\in D_i^l}\alpha_{ij}\sum_{u=0}^{K_u-1}\sum_{v=0}^{K_v-1}[\boldsymbol{Y}_j^{(l-1)}]_{xK_u+u,yK_v+v} + b_i^{(l)} \tag{86}$$

Where (K_u, K_v) defines the sub-sampling factors in the respective dimensions. Normally they are all equal, but we distinguish them here for generality. One could also allow different sub-sampling sizes between interconnected maps, but

the notation becomes rather tedious. Furthermore, we also allow interconnections in the sub-sampling layer. However, in a practical application $j = i$ and the entire scheme gets reduced to

$$[\boldsymbol{Z}_i^{(l)}(\boldsymbol{Y}_{1:N^{l-1}}^{(l-1)}, \Omega^{(l)})]_{xy} = \alpha_i \sum_{u=0}^{K-1} \sum_{v=0}^{K-1} [\boldsymbol{Y}_i^{(l-1)}]_{xK+u,yK+v} + b_i^{(l)} \tag{87}$$

The big question now is how to we propagate the delta through this sub-sampling layer. Let us study

$$\frac{\partial [\boldsymbol{Z}_i^{(l)}]_{x_1y_1}}{\partial [\boldsymbol{Y}_k^{(l-1)}]_{xy}} = \frac{\partial \sum_{j \in D_i^l} \alpha_{ij} \sum_{u=0}^{K_u-1} \sum_{v=0}^{K_v-1} [\boldsymbol{Y}_j^{(l-1)}]_{x_1K_u+u,y_1K_v+v} + b_i^{(l)}}{\partial [\boldsymbol{Y}_k^{(l-1)}]_{xy}} \tag{88}$$

$$= \alpha_{ik} \sum_{u=0}^{K_u-1} \sum_{v=0}^{K_v-1} \mathbb{1}_{\{x=x_1K_u+u,y=y_1K_v+v\}} \mathbb{1}_{\{k \in D_i^l\}} \tag{89}$$

Let us now put this into the fundamental equation governing delta, in order to simplify it further.

$$[\boldsymbol{\delta}_j^{(l)}]_{xy} = \sum_{i=1}^{N^{l+1}} \sum_{(x_1,y_1) \in S_i^{l+1}} [\boldsymbol{\delta}_i^{(l+1)}]_{x_1y_1} \alpha_{ij} \sum_{u=0}^{K_u-1} \sum_{v=0}^{K_v-1} \mathbb{1}_{\{x=x_1K_u+u,y=y_1K_v+v\}} \mathbb{1}_{\{j \in D_i^{l+1}\}} \phi^{(l)'}([\boldsymbol{Z}_j^{(l)}]_{xy}) \tag{90}$$

As noted earlier in the convolution section. The sum $\sum_{i=1}^{N^{l+1}} \mathbb{1}_{\{j \in D_i^{l+1}\}}$, is equivalent to summing i over C_j^l instead. Which transforms the expression into

$$[\boldsymbol{\delta}_j^{(l)}]_{xy} = \sum_{i \in C_j^l} \sum_{(x_1,y_1) \in S^{l+1}} [\boldsymbol{\delta}_i^{(l+1)}]_{x_1y_1} \alpha_{ij} \sum_{u=0}^{K_u-1} \sum_{v=0}^{K_v-1} \mathbb{1}_{\{x=x_1K_u+u,y=y_1K_v+v\}} \phi^{(l)'}([\boldsymbol{Z}_j^{(l)}]_{xy}) \tag{91}$$

As usual the expressions looks rather horrible, but fear not. For simplicity, we assume the same sizes in the layers because otherwise a lot more subscripts has to be added to the notation. Instead of trying a variable substitution as in the convolution case, we notice here that the summation $\sum_{(x_1,y_1) \in S^{l+1}} \sum_{u=0}^{K_u-1} \sum_{v=0}^{K_v-1} \mathbb{1}_{\{x=x_1K_u+u,y=y_1K_v+v\}}$ will correspond to the entire image of the previous layer since we don't allow overlapping. Concretely this means that we will have the value $[\boldsymbol{\delta}_i^{(l+1)}]_{x_1y_1}$ on sub rectangles defined on $\sum_{u=0}^{K_u-1} \sum_{v=0}^{K_v-1} \mathbb{1}_{\{x_1K_u+u,y_1K_v+v\}}$. This important insight can simplify Eq. 91 into

$$[\boldsymbol{\delta}_j^{(l)}]_{xy} = \sum_{i \in C_j^l} \alpha_{ij} [\boldsymbol{\delta}_i^{(l+1)}]_{\lfloor \frac{x}{K_u} \rfloor, \lfloor \frac{y}{K_v} \rfloor} \phi^{(l)'}([\boldsymbol{Z}_j^{(l)}]_{xy}) \tag{92}$$

And of course, for the practical situations the delta transforms into

$$[\boldsymbol{\delta}_j^{(l)}]_{xy} = \alpha_j [\boldsymbol{\delta}_j^{(l+1)}]_{\lfloor \frac{x}{K_u} \rfloor, \lfloor \frac{y}{K_v} \rfloor} \phi^{(l)'}([\boldsymbol{Z}_j^{(l)}]_{xy}) \tag{93}$$

Let us now derive the gradients for the weights.

$$\frac{\partial r}{\partial \alpha_{mn}} = \sum_{i=1}^{N^l} \sum_{(x,y) \in S_i^l} [\boldsymbol{\delta}_i^{(l)}]_{xy} \frac{\partial \sum_{j \in D_i^l} \alpha_{ij} \sum_{u=0}^{K_u-1} \sum_{v=0}^{K_v-1} [\boldsymbol{Y}_j^{(l-1)}]_{xK_u+u,yK_v+v} + b_i^{(l)}}{\partial \alpha_{mn}} \tag{94}$$

We see directly that the sum is zero for all $(i,j) \neq (m,n)$. From this, we conclude that

$$\frac{\partial r}{\partial \alpha_{mn}} = \sum_{(x,y)\in S_m^l} \sum_{u=0}^{K_u-1} \sum_{v=0}^{K_v-1} [\boldsymbol{Y}_n^{(l-1)}]_{xK_u+u,yK_v+v} [\boldsymbol{\delta}_m^{(l)}]_{xy} \tag{95}$$

For most practical cases we end up with

$$\frac{\partial r}{\partial \alpha_{m}} = \sum_{(x,y)\in S_m^l} \sum_{n\in D_m^l} \sum_{u=0}^{K_u-1} \sum_{v=0}^{K_v-1} [\boldsymbol{Y}_n^{(l-1)}]_{xK_u+u,yK_v+v} [\boldsymbol{\delta}_m^{(l)}]_{xy} \tag{96}$$

The bias is simply given by

$$\frac{\partial r}{\partial b_i^l} = \sum_{(x,y)\in S_i^l} [\boldsymbol{\delta}_i^{(l)}]_{xy} \tag{97}$$

The equations to remember are summarized in the below table

Feed forward, cross weights	$[\boldsymbol{Y}_i^{(l)}]_{xy} = \phi^{(l)}(\sum_{j\in D_i^l} \alpha_{ij} \sum_{u=0}^{K_u-1} \sum_{v=0}^{K_v-1} [\boldsymbol{Y}_j^{(l-1)}]_{xK_u+u,yK_v+v} + b_i^{(l)})$
Feed forward, single weights	$[\boldsymbol{Y}_i^{(l)}]_{xy} = \phi^{(l)}(\alpha_i \sum_{j\in D_i^l} \sum_{u=0}^{K_u-1} \sum_{v=0}^{K_v-1} [\boldsymbol{Y}_j^{(l-1)}]_{xK_u+u,yK_v+v} + b_i^{(l)})$
Back propagation	$[\boldsymbol{\delta}_j^{(l)}]_{xy} = \sum_{i\in C_j^l} \alpha_{ij} [\boldsymbol{\delta}_i^{(l+1)}]_{\lfloor \frac{x}{K_u} \rfloor, \lfloor \frac{y}{K_v} \rfloor} \phi^{(l)'}([\boldsymbol{Z}_j^{(l)}]_{xy})$
Weight update, cross weights	$\frac{\partial r}{\partial \alpha_{mn}} = \sum_{(x,y)\in S_m^l} \sum_{u=0}^{K_u-1} \sum_{v=0}^{K_v-1} [\boldsymbol{Y}_n^{(l-1)}]_{xK_u+u,yK_v+v} [\boldsymbol{\delta}_m^{(l)}]_{xy}$
Weight update, single weights	$\frac{\partial r}{\partial \alpha_{m}} = \sum_{(x,y)\in S_m^l} \sum_{n\in D_m^l} \sum_{u=0}^{K_u-1} \sum_{v=0}^{K_v-1} [\boldsymbol{Y}_n^{(l-1)}]_{xK_u+u,yK_v+v} [\boldsymbol{\delta}_m^{(l)}]_{xy}$
Bias update	$\frac{\partial r}{\partial b_i^l} = \sum_{(x,y)\in S_i^l} [\boldsymbol{\delta}_i^{(l)}]_{xy}$

Table 6: Fundamental weighed sub-sampling equations.

3.4.2 Max pooling

For a max-pooling layer we set the activation function to the identity transform. Furthermore, max pooling from several simultaneous layers are omitted. Thus yielding:

$$[\boldsymbol{Y}_i^{(l)}]_{xy} = [\boldsymbol{Z}_i^{(l)}(\boldsymbol{Y}_{1:N^{l-1}}^{(l-1)}, \Omega^{(l)})]_{xy} = \max\{[\boldsymbol{Y}_i^{(l-1)}]_{K_u x+u, K_v y+v}, 0 \leq u \leq K_u-1, 0 \leq v \leq K_v-1\} \tag{98}$$

The derivative of Eq. 98 is actually much simpler than it looks.

$$\frac{\partial [\boldsymbol{Z}_i^{(l+1)}]_{x_1 y_1}}{\partial [\boldsymbol{Y}_j^{(l)}]_{xy}} = \frac{\max\{[\boldsymbol{Y}_i^{(l)}]_{K_u x_1+u, K_v y_1+v}, 0 \leq u \leq K_u-1, 0 \leq v \leq K_v-1\}}{\partial [\boldsymbol{Y}_j^{(l)}]_{xy}} \tag{99}$$

We can quickly deduce that the above derivative is zero for $i \neq j$. Furthermore we can see that the only value that will survive is if $(x,y) = (x_1 K_u +$

$u_{max}, y_1 K_v + v_{max}$) inside the pooling region and this value is equal to 1. These two conditions transforms the fundamental equation over the deltas to:

$$[\boldsymbol{\delta}_j^{(l)}]_{xy} = \mathbb{1}_{\{[\boldsymbol{Y}_j^{(l)}]_{xy} = \max\{[\boldsymbol{Y}_j^{(l)}]_{K_u \lfloor \frac{x}{K_u} \rfloor + u, K_v \lfloor \frac{y}{K_v} \rfloor + v}\}} [\boldsymbol{\delta}_j^{(l+1)}]_{\lfloor \frac{x}{K_u} \rfloor, \lfloor \frac{y}{K_v} \rfloor} \phi^{(l)'}([\boldsymbol{Z}_j^{(l)}]_{xy}) \tag{100}$$

In this case, the table becomes quite small

Feed forward	$[\boldsymbol{Y}_i^{(l)}]_{xy} = \max\{[\boldsymbol{Y}_i^{(l-1)}]_{K_u x + u, K_v y + v}, 0 \leq u \leq K_u - 1, 0 \leq v \leq K_v - 1\}$
Back propagation	$[\boldsymbol{\delta}_j^{(l)}]_{xy} = \mathbb{1}_{\{[\boldsymbol{Y}_j^{(l)}]_{xy} = \max\{[\boldsymbol{Y}_j^{(l)}]_{K_u \lfloor \frac{x}{K_u} \rfloor + u, K_v \lfloor \frac{y}{K_v} \rfloor + v}\}} [\boldsymbol{\delta}_j^{(l+1)}]_{\lfloor \frac{x}{K_u} \rfloor, \lfloor \frac{y}{K_v} \rfloor} \phi^{(l)'}([\boldsymbol{Z}_j^{(l)}]_{xy})$

Table 7: Fundamental max-pooling equations.

3.4.3 Vanilla sub sampling

This section aims at describing a very trivial sub sampling operation which simply takes every k:th coordinate to the second layer. As in the previous sub section we will not allow connections between several maps, since it doesn't make sense.

$$[\boldsymbol{Y}_i^{(l)}]_{xy} = [\boldsymbol{Z}_i^{(l)}(\boldsymbol{Y}_{1:N^{l-1}}^{(l-1)}, \Omega^{(l)})]_{xy} = [\boldsymbol{Y}_i^{(l-1)}]_{xK_u, yK_v} \tag{101}$$

The derivative of Eq. 101 will be very simple

$$\frac{\partial [\boldsymbol{Z}_i^{(l+1)}]_{x_1 y_1}}{\partial [\boldsymbol{Y}_j^{(l)}]_{xy}} = \frac{\partial [\boldsymbol{Y}_i^{(l)}]_{x_1 K_u, y_1 K_v}}{\partial [\boldsymbol{Y}_j^{(l)}]_{xy}} \tag{102}$$

As in the max pooling case we note that the above expression is zero if $i \neq j$. Secondly, we note that $(x, y) = (x_1 K_u, y_1 K_v)$ are the only values for which the derivative is non zero. The fundamental equation takes the following form

$$[\boldsymbol{\delta}_j^{(l)}]_{xy} = [\boldsymbol{\delta}_j^{(l+1)}]_{\lfloor \frac{x}{K_u} \rfloor, \lfloor \frac{y}{K_v} \rfloor} \phi^{(l)'}([\boldsymbol{Z}_j^{(l)}]_{xy}) \mathbb{1}_{\{0 \equiv x \pmod{K_u}, 0 \equiv y \pmod{K_v}\}} \tag{103}$$

As in the max-pooling case, the table becomes very simple

Feed forward	$[\boldsymbol{Y}_i^{(l)}]_{xy} = [\boldsymbol{Y}_i^{(l-1)}]_{xK_u, yK_v}$
Back propagation	$[\boldsymbol{\delta}_j^{(l)}]_{xy} = [\boldsymbol{\delta}_j^{(l+1)}]_{\lfloor \frac{x}{K_u} \rfloor, \lfloor \frac{y}{K_v} \rfloor} \phi^{(l)'}([\boldsymbol{Z}_j^{(l)}]_{xy}) \mathbb{1}_{\{0 \equiv x \pmod{K_u}, 0 \equiv y \pmod{K_v}\}}$

Table 8: Fundamental vanilla pooling equations.

4 LM Optimization

4.1 Multilayer Perceptron

For smaller networks it's known that Levenberg Marquardt optimization outperforms gradient descent both in training time and generalization. It's thus interesting to derive the equations for a LM trained network.

Let us concern ourselves the the following network:

$$y_i^{(l)} = \phi^{(l)}(\sum_{j=1}^{N^{(l-1)}} y_j^{(l-1)} \times \omega_{ji}^{(l)} + b_i^{(l)}) \tag{104}$$

The error function is the standard SSE function:

$$E = \sum_{i=1}^{N^t} \parallel \boldsymbol{t}_i - \boldsymbol{y}_i^{(L)} \parallel^2 \tag{105}$$

The LM optimization problem can be applied to the following problem:

$$E(\boldsymbol{\omega}) = \frac{1}{2}\sum_{i=1}^{N^t} r_i(\boldsymbol{\omega})^2 \tag{106}$$

Where $r_i(\boldsymbol{\omega}) = \parallel \boldsymbol{t}_i - \boldsymbol{y}_i^{(L)} \parallel$. We note here that the gradient of the square sum is obtained by:

$$\frac{\partial E(\boldsymbol{\omega})}{\partial \omega_j} = \sum_{i=1}^{N^t} r_i(\boldsymbol{\omega}) \frac{\partial r_i(\boldsymbol{\omega})}{\partial \omega_j} \tag{107}$$

By using Levenberg & Marquard the solution is given iteratively by:

$$(\boldsymbol{J}^T\boldsymbol{J} + \lambda \boldsymbol{diag}(\boldsymbol{J}^T\boldsymbol{J}))\boldsymbol{\Delta} = -\boldsymbol{J}^T\boldsymbol{r}(\boldsymbol{\omega}) \tag{108}$$

Where $(\boldsymbol{J})_{ij} = \frac{\partial r_i(\boldsymbol{\omega})}{\partial \omega_j}$. By using the previous definitions the gradient of $E(\boldsymbol{\omega})$ is obtained simply by using the Jacobian and the residual vector.

$$\nabla E(\boldsymbol{\omega}) = \boldsymbol{J}^T\boldsymbol{r} \tag{109}$$

A remark on the notation is that:

$$\boldsymbol{\omega} = (\omega_{11}^{(1)}, \omega_{21}^{(1)} \ldots, \omega_{N^0 1}^{(1)}, \omega_{12}^{(1)} \ldots, \omega_{N^0 2}^{(1)}, \ldots, \omega_{N^0 N^1}^{(1)}, b_1^{(1)}, b_2^{(1)} \ldots, b_{N^1}^{(1)}, \omega_{11}^{(2)}, \ldots, b_{N^L}^{(L)})$$

The goal is now to make use of the standard back-propagation equations and put them into a LM form. Let us begin by calculating the derivative $\frac{\partial r_i(\boldsymbol{\omega})}{\partial \omega_j}$. For readability, we hereon remove the subscript i from the training set.

$$\frac{\partial r(\boldsymbol{\omega})}{\partial \omega_i} = \frac{\partial}{\partial \omega_i} \parallel \boldsymbol{t} - \boldsymbol{y}^{(L)} \parallel = -\frac{1}{\parallel \boldsymbol{t} - \boldsymbol{y}^{(L)} \parallel} \sum_{k=1}^{N^L} (t_k - y_k^{(L)}) \frac{\partial y_k^{(L)}}{\partial \omega_i} \tag{110}$$

$$\frac{\partial y_k^{(L)}}{\partial \omega_i} = \phi^{(L)'}(z_k^{(L)}) \sum_{j=1}^{N^{(L-1)}} \frac{\partial y_j^{(L-1)}}{\partial \omega_i} \times \omega_{jk}^{(L)}$$

$$\frac{\partial y_j^{(L-1)}}{\partial \omega_i} = \phi^{(L-1)'}(z_j^{(L-1)}) \sum_{m=1}^{N^{(L-2)}} \frac{\partial y_m^{(L-2)}}{\partial \omega_i} \times \omega_{mj}^{(L-1)}$$

$$\vdots$$

$$\frac{\partial y_n^{(l)}}{\partial \omega_i} = \phi^{(l)'}(z_n^{(l)}) y_p^{(l-1)}$$

The last line holds true if $\omega_i = \omega_{pn}^{(l)}$.

The next step is now to derive the delta notation for the LM case. In order to do this, we put the equations together and identify the deltas.

$$\frac{\partial r(\boldsymbol{\omega})}{\partial \omega_i} = \frac{\partial}{\partial \omega_i} \parallel \boldsymbol{t} - \boldsymbol{y}^{(L)} \parallel = -\frac{1}{\parallel \boldsymbol{t} - \boldsymbol{y}^{(L)} \parallel} \sum_{k=1}^{N^L} (t_k - y_k^{(L)})$$
$$\phi^{(L)'}(z_k^{(L)}) \sum_{j=1}^{N^{(L-1)}} \omega_{jk}^{(L)} \times \phi^{(L-1)'}(z_j^{(L-1)}) \sum_{m=1}^{N^{(L-2)}} \omega_{mj}^{(L-1)} \times$$
$$\vdots$$
$$\phi^{(l)'}(z_n^{(l)}) y_p^{(l-1)}$$

By defining deltas accordingly, we may simply the above form.

$$\delta_k^{(L)} = (t_k - y_k^{(L)}) \tag{111}$$

$$\delta_j^{(l)} = \sum_{k=1}^{N^{l+1}} \delta_k^{(l+1)} \times \phi^{(l+1)'}(z_k^{(l+1)}) \times \omega_{jk}^{(l+1)} \tag{112}$$

This yields the following gradient equation:

$$\frac{\partial r(\boldsymbol{\omega})}{\partial \omega_{pn}^{(l)}} = -\frac{1}{\parallel \boldsymbol{t} - \boldsymbol{y}^{(L)} \parallel} \phi^{(l)'}(z_n^{(l)}) \times y_p^{(l-1)} \times \delta_n^{(l)} \tag{113}$$

We now have everything to construct the LM equations.

$$\boldsymbol{r}(\boldsymbol{\omega}) = (r_1(\boldsymbol{\omega}), r_2(\boldsymbol{\omega}) ..., r_{N^t}(\boldsymbol{\omega})) \tag{114}$$

$$\boldsymbol{J} = \frac{\partial r_i(\boldsymbol{\omega})}{\partial \omega_{pn}^{(l)}} = -\frac{1}{\parallel \boldsymbol{t} - \boldsymbol{y}^{(L)} \parallel} \phi^{(l)'}(z_{n,i}^{(l)}) \times y_{p,i}^{(l-1)} \times \delta_{n,i}^{(l)} \tag{115}$$

4.2 Implementation Considerations

A difficult problem when implementing this approach for big neural networks is, a part from the memory requirements, the fact that the matrix $(\boldsymbol{J}^T\boldsymbol{J} + \lambda \boldsymbol{diag}(\boldsymbol{J}^T\boldsymbol{J}))$ is very often badly conditioned. One could try to multiply the left and right side of the equation in order to improve the condition of the linear system. However, this is not guaranteed to work in every situation. A solution to this problem is to consider implementing SVD decomposition and solve the linear system from the pseudo-inverse. The problem may arise when the training data doesn't distinguish very well.

The SVD decomposition of a matrix is given by:

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{W}\boldsymbol{V}^T \tag{116}$$

Where $\boldsymbol{A}$ is any matrix of dimension $M \times N$. $\boldsymbol{U}$ is is a column orthogonal matrix of size $M \times N$. $\boldsymbol{W}$ is a diagonal matrix with positive or zero elements. $\boldsymbol{V}$ is a $N \times N$ orthogonal matrix.

There are some important properties to the SVD decomposition. First of all the columns of $\boldsymbol{U}$ whose same numbered elements w_j in the diagonal matrix $\boldsymbol{W}$ are non zero are an orthonormal base that span the range of $\boldsymbol{A}$. Secondly, the columns of $\boldsymbol{V}$ whose elements w_j are zero are a orthonormal base for the nullspace of $\boldsymbol{A}$. In order to understand why we may use this method to solve a linear equation system, let us consider the inverse of $\boldsymbol{A}$ using its SVD form.

$$\boldsymbol{A}^{-1} = \boldsymbol{V}[diag(1/w_i)]\boldsymbol{U}^T \tag{117}$$

From the above equation we directly that we may encounter problem when some singular values are close to zero. The formal definition of saying that the matrix $\boldsymbol{A}$ is ill-conditioned is by the condition number which is simply the ratio between the greatest and the smallest singular value. In this way we can say whether or not the inverse will be susceptible to round-off errors during computation. For floating precision, we'll have problems when the inverse condition number is approaching the machine's floating point precision - which is around 10^{-7}. By using double floats, we may boost this number to 10^{-15}. The big question is now how we fix this problem so that a stable solution may be obtained. The fix is very simple and clever, we set the $1/w_i$ to zero for w_i small. What small is considered is relative, but if the number is 10^7 smaller than the greatest number for float precision and 10^{15} smaller for double precision, then we set the inverse to zero.

Another technique can increase the stability of the method is to use preconditioning on $(\boldsymbol{J}^T\boldsymbol{J} + \lambda \boldsymbol{diag}(\boldsymbol{J}^T\boldsymbol{J}))$. An effective technique here is to multiply the matrix by the inverse of its diagonal. However, it must be checked that none of the diagonal elements are zero.

5 Gradient Descent

Given a lower bounded continuously differentiable function $F(\boldsymbol{x})$, then we know that $F(\boldsymbol{x})$ decreases fastest in the negative gradient direction $-\nabla F(\boldsymbol{x})$. Thus we can construct the below iteration schema where $\mu_i \in \mathbb{R}$.

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - \mu_i \nabla F(\boldsymbol{x}_i) \tag{118}$$

If μ_i are chosen small enough so that $F(\boldsymbol{x}_0) \geq F(\boldsymbol{x}_1) \geq F(\boldsymbol{x}_2) \geq \dots$ then convergence to a local minimum is guaranteed. There is an alternative to the gradient descent algorithm called Stochastic Gradient Descent which is often applied in training neural networks. This method implies that the data that we are fitting on is changed for every iteration. The convergence guarantee becomes more complicated but it can be proved that the method converges almost surely to a local minimum in this case (See Robbins-Seigmund Theorem).

5.1 Choosing μ_i

The ideal μ_i for every iteration would be the one that minimizes $F(\boldsymbol{x}_i - \mu \nabla F(\boldsymbol{x}_i))$. In order to find this optimal μ_i we need to solve the equation:

$$\frac{d}{d\mu_i} F(\boldsymbol{x}_i - \mu_i \nabla F(\boldsymbol{x}_i)) = 0 \tag{119}$$

Observe that for a neural net with many nested layers, it will be extremely difficult to extract the exact analytical expression for μ_i (someone courageous could check this out to see if it's true). However, we'll get back to Eq. 119 in the Conjugate Gradient chapter, which aims at actually solving the equation.

Very often when looking at implementations of different neural networks, the step size is often taken in a heuristic way. For example, decreasing as a function of the training epoch. This has to be defined a tested from case to case to make sure its performance is satisfactory.

In the remaining part of this section, we'll look at other methods of finding μ_i. The methods that we'll consider are Golden Section Search and Brent's method for finding μ_i. However, there are other inexact methods based on the Armijo-Goldstein condition as well as the Wolfe condition.

In order to use Golden Section Search or Brent's method, we need to bracket the solution. This is done by searching for a bracket, in the gradient direction from $\boldsymbol{x}_i$ until the bracket has been obtained. Afterwards, either Brent's or the Golden Search may be applied. Note however, that this will obtain the exact value of μ_i and is rather costly.

5.2 Momentum

Let us recapitulate the gradient descent method.

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - \mu_i \nabla F(\boldsymbol{x}_i) \tag{120}$$

It is well known that the above learning method may be very slow. In many literature an additional term known as the momentum has been proposed.

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - \mu_i \nabla F(\boldsymbol{x}_i) + p\Delta\boldsymbol{x}_{i-1} \tag{121}$$

Where p is known as the momentum parameter. Intuitively, the moment term averages out the learning path. This may be particularly useful for long narrow valleys where the vanilla descent method usually oscillate.

6 Conjugate Gradient

6.1 Conjugate Directions

As we've seen previously in the gradient descent method, if we find the optimal μ_i, it's always orthogonal to the previous gradient. This will render the method ineffective in narrow valleys since it will oscillate between the edges of the valley (it depends on the starting position how severe). A cure to this problem would be to find a set of orthogonal directions $\boldsymbol{d}_0, \boldsymbol{d}_1 \dots \boldsymbol{d}_{n-1}$ so that taking a step in each direction will remove the corresponding component of the error $\boldsymbol{e} = \tilde{\boldsymbol{x}} - \boldsymbol{x}$. This would imply that after n steps, we are done. Mathematically, this means that we will update $\boldsymbol{x}_i$ by

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \alpha_i \boldsymbol{d}_i \tag{122}$$

Since we require that the error $\boldsymbol{e}_i = \tilde{\boldsymbol{x}} - \boldsymbol{x}_i$ has its $\boldsymbol{d}_i, \boldsymbol{d}_{i-1} \dots \boldsymbol{d}_0$ components removed after every iteration, we require that $\boldsymbol{e}_{i+1}$ is orthogonal to $\boldsymbol{d}_i$. By using

this condition together with Eq. 122 we obtain the following value on α_i, which is in fact useless since we don't know $\boldsymbol{e}_i$.

$$\alpha_i = -\frac{\boldsymbol{d}_i^T \boldsymbol{e}_i}{\boldsymbol{d}_i^T \boldsymbol{d}_i} \tag{123}$$

Let us start exposing a potential solution on a quadratic form.

$$F(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T \boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}^T \boldsymbol{x} + c \tag{124}$$

$$\nabla F(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x} - \boldsymbol{b} \tag{125}$$

[2] now suggest that we require the errors and the directions are $\boldsymbol{A}$-orthogonal instead of simply orthogonal.

$$\boldsymbol{d}_i^T \boldsymbol{A}\boldsymbol{d}_j = 0, i \neq j \tag{126}$$

$$\boldsymbol{e}_{i+1}^T \boldsymbol{A}\boldsymbol{d}_i = 0 \tag{127}$$

Not surprisingly, this new condition is equivalent to finding the minimum of $F(\boldsymbol{x})$ along the search direction $\boldsymbol{d}_i$.

$$\frac{dF(\boldsymbol{x}_{i+1)}}{d\alpha_i} = 0 \tag{128}$$

$$\nabla F(\boldsymbol{x}_{i+1})^T \frac{d\boldsymbol{x}_{i+1}}{\alpha_i} = 0 \tag{129}$$

$$-\boldsymbol{r}_{i+1}^T \boldsymbol{d}_i = 0 \tag{130}$$

$$\boldsymbol{d}_i^T \boldsymbol{A}\boldsymbol{e}_{i+1} = 0 \tag{131}$$

Where we have defined the residual $\boldsymbol{r}_i = -\nabla F(\boldsymbol{x}_i)$. Since we required $\boldsymbol{A}$-orthogonality between $\boldsymbol{e}_{i+1}$ and $\boldsymbol{d}_i$ we obtain the following expression for $\boldsymbol{e}_{i+1}$:

$$\boldsymbol{r}_{i+1} = -\boldsymbol{A}\boldsymbol{e}_{i+1} \tag{132}$$

Let us now transform Eq. 122 by using the residual instead.

$$\boldsymbol{b} - A\boldsymbol{x}_{i+1} = \boldsymbol{b} - A\boldsymbol{x}_i - \alpha_i \boldsymbol{A}\boldsymbol{d}_i \tag{133}$$

$$\boldsymbol{r}_{i+1} = \boldsymbol{r}_i - \alpha_i \boldsymbol{A}\boldsymbol{d}_i \tag{134}$$

By using Eq. 134 together with Eq. 132 we can obtain an update equation on the error as well.

$$\boldsymbol{e}_{i+1} = \boldsymbol{e}_i + \alpha_i \boldsymbol{d}_i \tag{135}$$

So, instead of the useless Eq. 123 we can now calculate α_i by expressing it in terms of directions and residuals. This is a direct consequence of the requirement that the directions are orthogonal to the proceeding error.

$$\boldsymbol{d}_i^T \boldsymbol{A}\boldsymbol{e}_{i+1} = 0 \tag{136}$$

$$\boldsymbol{d}_i^T \boldsymbol{A}(\boldsymbol{e}_i + \alpha_i \boldsymbol{d}_i) = 0 \tag{137}$$

$$\alpha_i = -\frac{\boldsymbol{d}_i^T \boldsymbol{A}\boldsymbol{e}_i}{\boldsymbol{d}_i^T \boldsymbol{A}\boldsymbol{d}_i} \tag{138}$$

$$\alpha_i = \frac{\boldsymbol{d}_i^T \boldsymbol{r}_i}{\boldsymbol{d}_i^T \boldsymbol{A}\boldsymbol{d}_i} \tag{139}$$

Up until now we have just required that $\boldsymbol{e}_i$ and the $\boldsymbol{d}_j$ to be $\boldsymbol{A}$-orthogonal. We now need to verify by using this condition, we can converge to the correct solution in n iterations (n is the dimension of $\boldsymbol{x}$).

Firstly, we need to prove that the error $\boldsymbol{e}_0$ can be expressed as a linear combination of $\{\boldsymbol{d}_i\}$. Assume that $\boldsymbol{A}$ is a positive definite matrix and assume that $\{\boldsymbol{d}_i\}$ doesn't span $\mathbb{R}^n$. This implies that:

$$\boldsymbol{d}_i = \sum_{j \neq i} \beta_j \boldsymbol{d}_j \tag{140}$$

$$\boldsymbol{d}_i^T \boldsymbol{A} \boldsymbol{d}_i > \sum_{j \neq i} \beta_j \boldsymbol{d}_i^T \boldsymbol{A} \boldsymbol{d}_j \tag{141}$$

This is a contradiction since $\boldsymbol{d}_i^T \boldsymbol{A} \boldsymbol{d}_j = 0$ for $i \neq j$ by definition. We must therefore have that $\{\boldsymbol{d}_i\}$ span $\mathbb{R}^n$. This in turn implies that the error $\boldsymbol{e}_0$ can be written as a linear combination of $\{\boldsymbol{d}_i\}$.

$$\boldsymbol{e}_0 = \sum_{j=0}^{n-1} \gamma_j \boldsymbol{d}_j \tag{142}$$

We can now easily see the connection between γ_i and α_i

$$\boldsymbol{d}_i^T \boldsymbol{A} \boldsymbol{e}_0 = \sum_{j=0}^{n-1} \gamma_j \boldsymbol{d}_i^T \boldsymbol{A} \boldsymbol{d}_j \tag{143}$$

$$\gamma_i = \frac{\boldsymbol{d}_i^T \boldsymbol{A} \boldsymbol{e}_0}{\boldsymbol{d}_i^T \boldsymbol{A} \boldsymbol{d}_i} \tag{144}$$

$$\gamma_i = \frac{\boldsymbol{d}_i^T \boldsymbol{A} (\boldsymbol{e}_0 + \sum_{j=0}^{i-1} \alpha_j \boldsymbol{d}_j)}{\boldsymbol{d}_i^T \boldsymbol{A} \boldsymbol{d}_i} \tag{145}$$

$$\gamma_i = \frac{\boldsymbol{d}_i^T \boldsymbol{A} \boldsymbol{e}_i}{\boldsymbol{d}_i^T \boldsymbol{A} \boldsymbol{d}_i} \tag{146}$$

$$\gamma_i = -\alpha_i \tag{147}$$

What Eq. 147 means is that when we perform the update equation 135 we remove a component of $\boldsymbol{e}_i$ until the error is finally zero. This also proves that the algorithm converges in n steps.

In order to derive an algorithm for using Conjugated Directions, we need a way of choosing $\{\boldsymbol{d}_i\}$. This can effectively be done by Conjugated Gram-Schmidt. In short the process consists of first choosing a base $\{\boldsymbol{u_i}\}$ for $\mathbb{R}^n$ and express a conjugated direction as

$$\boldsymbol{d}_i = \boldsymbol{u}_i + \sum_{j=0}^{i-1} \beta_{ij} \boldsymbol{d}_j \tag{148}$$

$$\beta_{ij} = -\frac{\boldsymbol{u}_i^T \boldsymbol{A} \boldsymbol{d}_j}{\boldsymbol{d}_j^T \boldsymbol{A} \boldsymbol{d}_j} \tag{149}$$

The reason to expose is not actual implement this as an algorithm since it will be equivalent to a Gaussian elimination, but to understand the Conjugate

Gradient method. Another method that will calculate conjugated directions is Powel's method that is vastly more effective.

6.2 Conjugated Gradient

By understanding conjugated directions, it becomes much easier to understand conjugated gradients since this is just a special case with $\boldsymbol{u}_i = \boldsymbol{r}_i$. There's many reasons why this choice is interesting. Firstly, the residuals has a nice property that they are orthogonal to previous search directions (which will become clear soon). The most important reason is actually that the gradients will reduce the complexity of β_{ij} so that we don't need to store old search vectors. Because of this, the algorithm will have a nice $\mathcal{O}(n)$ complexity instead of $\mathcal{O}(n^2)$.

Some results of this choice is that

$$\mathcal{D}_i = span\{\boldsymbol{d}_0, \boldsymbol{A}\boldsymbol{d}_0 \dots \boldsymbol{A}^{i-1}\boldsymbol{d}_0\} = span\{\boldsymbol{r}_0, \boldsymbol{A}\boldsymbol{r}_0 \dots \boldsymbol{A}^{i-1}\boldsymbol{r}_0\} \tag{150}$$

$$\boldsymbol{r}_i^T \boldsymbol{r}_j = 0, \qquad i \neq j \tag{151}$$

This can easily be seen from Eq. 148 and Eq. 134. Eq. 150 is called a Krylov subspace and is constructed by consecutive multiplications of a given matrix. By using this fact we can deduce that $\boldsymbol{r}_{i+1} \perp \mathcal{D}_{i+1}$. Since $\boldsymbol{A}\mathcal{D}_i \subset \mathcal{D}_{i+1}$, we conclude that $\boldsymbol{r}_{i+1}$ is $\boldsymbol{A}$-orthogonal to $\mathcal{D}_i$. What this means is that $\boldsymbol{r}_{i+1}$ is already $\boldsymbol{A}$-orthogonal to all of the previous directions in Eq. 148 - which makes the Gram-Schmidt construction easy.

By using Eq. 134 we know the two following facts:

$$\boldsymbol{r}_i^T \boldsymbol{r}_{j+1} = \boldsymbol{r}_i^T \boldsymbol{r}_j - \alpha_j \boldsymbol{r}_i^T \boldsymbol{A}\boldsymbol{d}_j \tag{152}$$

$$\alpha_j \boldsymbol{r}_i^T \boldsymbol{A}\boldsymbol{d}_j = \boldsymbol{r}_i^T \boldsymbol{r}_j - \boldsymbol{r}_i^T \boldsymbol{r}_{j+1} \tag{153}$$

By using Eq. 149 we therefore obtain the following form of β_{ij}.

$$\beta_{ij} = \frac{1}{\alpha_{i-1}} \frac{\boldsymbol{r}_i^T \boldsymbol{r}_i}{\boldsymbol{d}_{i-1}^T \boldsymbol{A}\boldsymbol{d}_{i-1}}, \qquad i = j+1 \tag{154}$$

For values $i = j + 1$, else $\beta_{ij} = 0$. By plugging Eq.139 and the fact that $\boldsymbol{d}_i^T \boldsymbol{r}_i = \boldsymbol{u}_i^T \boldsymbol{r}_i = \boldsymbol{r}_i^T \boldsymbol{r}_i$, we obtain the below equation:

$$\beta_i = \frac{\boldsymbol{r}_i^T \boldsymbol{r}_i}{\boldsymbol{r}_{i-1}^T \boldsymbol{r}_{i-1}} \tag{155}$$

By putting it all together we obtain the Conjugate Gradient method for a quadratic form.

$$\boldsymbol{d}_0 = \boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0 \tag{156}$$

$$\alpha_i = \frac{\boldsymbol{r}_i^T \boldsymbol{r}_i}{\boldsymbol{d}_i^T \boldsymbol{A}\boldsymbol{d}_i} \tag{157}$$

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \alpha_i \boldsymbol{d}_i \tag{158}$$

$$\boldsymbol{r}_{i+1} = \boldsymbol{r}_i - \alpha_i \boldsymbol{A}\boldsymbol{d}_i \tag{159}$$

$$\beta_{i+1} = \frac{\boldsymbol{r}_{i+1}^T \boldsymbol{r}_{i+1}}{\boldsymbol{r}_i^T \boldsymbol{r}_i} \tag{160}$$

$$\boldsymbol{d}_{i+1} = \boldsymbol{r}_{i+1} + \beta_{i+1} \boldsymbol{d}_i \tag{161}$$

Up until now, we've only considered a very special case of a quadratic function. Except for understanding the Conjugate Gradient method, it's pretty useless. There's also some alternatives for solving linear systems called Biconjugate Gradient Method which doesn't assume anything of the linear system. However, now we are going to expose the Conjugate Gradient method for a general continuously differentiable function $f(\boldsymbol{x})$.

Recall that Eq. 161 and Eq. 159 define vectors that satisfy the orthogonality condition:

$$\boldsymbol{r}_i^T \boldsymbol{r}_j = 0 \qquad \boldsymbol{d}_i^T \boldsymbol{A} \boldsymbol{d}_j = 0 \qquad \boldsymbol{r}_i^T \boldsymbol{d}_j = 0 \qquad j < i \tag{162}$$

The problem now is that we don't have access to the Hessian matrix $\boldsymbol{A}$. However, in proximity of a point $\boldsymbol{x_i}$ we can use the Taylor expansion

$$f(\boldsymbol{x}) \approx \frac{1}{2} \boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x} - \boldsymbol{b}^T \boldsymbol{x} + c \tag{163}$$

However, we now that this is true for some values of $\boldsymbol{A}$ and $\boldsymbol{b}$ but we still don't know them. What we do is the following: Take a point $\boldsymbol{x_i}$ and construct $\boldsymbol{r}_i = -\nabla f(\boldsymbol{x}_i)$ where $f(\boldsymbol{x}_i)$ is of form Eq. 163. Suppose we now define the next point $\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \alpha_i \boldsymbol{d}_i$ where $\alpha_i = min_\alpha[f(\boldsymbol{x}_i + \alpha \boldsymbol{d}_i)]$, then $\boldsymbol{r}_{i+1} = -\nabla f(\boldsymbol{x}_{i+1})$ is the same vector as would have been constructed by Eq. 159.

Now when we have a basic understanding of the algorithm let us simply write out the general non-linear algorithm.

$$\boldsymbol{d}_0 = \boldsymbol{r}_0 = -\nabla f(\boldsymbol{x}_0) \tag{164}$$

$$\alpha_i = min_\alpha[f(\boldsymbol{x}_i + \alpha \boldsymbol{d}_i)] \tag{165}$$

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \alpha_i \boldsymbol{d}_i \tag{166}$$

$$\boldsymbol{r}_{i+1} = -\nabla f(\boldsymbol{x}_{i+1}) \tag{167}$$

$$\beta_{i+1} = max\Big\{ \frac{\boldsymbol{r}_{i+1}^T (\boldsymbol{r}_{i+1} - \boldsymbol{r}_i)}{\boldsymbol{r}_i^T \boldsymbol{r}_i}, 0 \Big\} \tag{168}$$

$$\boldsymbol{d}_{i+1} = \boldsymbol{r}_{i+1} + \beta_{i+1} \boldsymbol{d}_i \tag{169}$$

6.3 Implementation Consideration

This section mainly deals with the non-linear CG method. Remarks like preconditioning still holds for the standard CG method where we have a matrix $\boldsymbol{A}$. However, the formulas may change a bit when using this.

Given that we re-evaluate the derivative at every iteration in the non-linear CG, we shouldn't expect to have any particular round-off errors and the method is thus very stable. Of course, care has to be taken when dividing by $\boldsymbol{r}_i^T \boldsymbol{r}_i$. However, if this value is very close to zero, then we usually stop the algorithm before.

For a neural network, the most expensive part to evaluate is $\alpha_i = min_\alpha[f(\boldsymbol{x}_i + \alpha \boldsymbol{d}_i)]$ and different methods should be tried in order to find α_i. For example, Brent's Method, Golden Section Search and Brent's Method incorporating derivatives. One could possibly explore approximate methods for α_i such as the Wolfe condition.

7 BFGS

BFGS is known as a quasi Newton method in the sense that it tries to use an approximate Hessian matrix to minimize the objective function.

$$f(\boldsymbol{x}) \approx f(\boldsymbol{p}) + \nabla f(\boldsymbol{p})^T(\boldsymbol{x} - \boldsymbol{p}) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{p})^T \boldsymbol{B}_k(\boldsymbol{x} - \boldsymbol{p}) \tag{170}$$

Instead of having $\boldsymbol{B}_k = \boldsymbol{H}$, we will set $\boldsymbol{B}_k$ as an approximation to $\boldsymbol{H}$ that will converge towards the Hessian $\boldsymbol{H}$ with increasing k. Let us rewrite 170 into a more convenient for where f_k indicates the evaluation of the function f at $\boldsymbol{x}_k$. Furthermore, we remove the difference by a simple coordinate transform.

$$\tilde{f}(\boldsymbol{x})_k \approx f_k + \nabla f_k^T \boldsymbol{x} + \frac{1}{2}\boldsymbol{x}^T \boldsymbol{B}_k \boldsymbol{x} \tag{171}$$

By searching a minimum to Eq. 171, we obtain the following solution, which is simply the newton update equation:

$$\boldsymbol{d}_k = -\boldsymbol{B}_k^{-1} \nabla f_k \tag{172}$$

Remember that for a perfect quadratic function, the $\boldsymbol{d}_k$ will take you directly to the minimum. However, in the real world, we only use $\boldsymbol{d}_k$ as an approximation and we are not sure that it will yield the best solution. That's why we set the update criteria as

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k \tag{173}$$

Where α_k is given by a line search through the direction $\boldsymbol{d}_k$ (starting at $\alpha_k = 1$). The problem that BFGS now tries to solve is how we approximate the Hessian that we use in Eq. 172. A very reasonable way to tackle this problem is to require that $\nabla \tilde{f}(\boldsymbol{x}_{k+1})_{k+1} = \nabla f_{k+1}$ and $\nabla \tilde{f}(\boldsymbol{x}_k)_{k+1} = \nabla f_k$. The first condition is trivial since it's given by the definition of a Taylor expansion (170). The second condition will imply

$$\alpha_k \boldsymbol{B}_{k+1} \boldsymbol{d}_k = \nabla f_{k+1} - \nabla f_k \tag{174}$$

Heron, let us denote

$$\boldsymbol{s}_k = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k, \qquad \boldsymbol{y}_k = \nabla f_{k+1} - \nabla f_k \tag{175}$$

$$\boldsymbol{B}_{k+1} \boldsymbol{s}_k = \boldsymbol{y}_k \tag{176}$$

Eq. 176 is referred to as the secant equation. Since we will require that $\boldsymbol{B}_{k+1}$ is symmetric positive definite, we also obtain the so called curvature condition

$$\boldsymbol{s}_k^T \boldsymbol{y}_k > 0 \tag{177}$$

Note that if we impose the Wolfe condition or strong Wolfe condition on the line search Eq. 177 is guaranteed to hold. Furthermore, if the curvature condition

Eq. 177 holds, then we are guaranteed the existence of a symmetric positive definite matrix $\boldsymbol{B}_{k+1}$ of Eq. 176. However, in this case may choose between an infinite amount of solutions, so we therefore need to impose an additional condition that will guarantee that $\boldsymbol{B}_{k+1}$ is the closest matrix to its predecessor. The norm that defines closest in this since is usually the Frobenius norm with a wisely chosen weight matrix. let us define the inverse

$$\boldsymbol{H}_k = \boldsymbol{B}_k^{-1} \tag{178}$$

Instead of imposing conditions on the Hessian approximation, we can impose conditions on its inverse $\boldsymbol{H}_k$ and by doing this we obtain the BFGS (Broyden, Fletcher, Goldfarb and Shanno) algorithm.

$$\boldsymbol{s}_k = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k \tag{179}$$

$$\boldsymbol{y}_k = \nabla f_{k+1} - \nabla f_k \tag{180}$$

$$\boldsymbol{u}_k = \frac{\boldsymbol{s}_k}{\boldsymbol{s}_k^T \boldsymbol{y}_k} - \frac{\boldsymbol{H}_k \boldsymbol{y}_k}{\boldsymbol{y}_k^T \boldsymbol{H}_k \boldsymbol{y}_k} \tag{181}$$

$$\boldsymbol{H}_{k+1} = \boldsymbol{H}_k + \frac{\boldsymbol{s}_k \otimes \boldsymbol{s}_k}{\boldsymbol{s}_k^T \boldsymbol{y}_k} - \frac{\boldsymbol{H}_k \boldsymbol{y}_k \otimes \boldsymbol{H}_k \boldsymbol{y}_k}{\boldsymbol{y}_k^T \boldsymbol{H}_k \boldsymbol{y}_k} + [\boldsymbol{y}_k^T \boldsymbol{H}_k \boldsymbol{y}_k] \boldsymbol{u}_k \otimes \boldsymbol{u}_k \tag{182}$$

Where $\rho_k = \frac{1}{\boldsymbol{y}_k^T \boldsymbol{s}_k}$. One usually initialize $\boldsymbol{H}_0$ by either setting it to the identity or by calculating a finite difference of $f(\boldsymbol{x}_0)$.

However a big problem with using this method is the same as Levenberg-Marquardt: i.e. memory consumption for big problems! And especially for a CNN it can be too much to store all the variables n^2. That's we the proceeding subsection will deal with a limited memory model of the BFGS algorithm. Before proceeding, let us summarize this section by writing out the BFGS algorithm.

Algorithm 1: BFGS

Initialize $\boldsymbol{x}_0$, $\boldsymbol{H}_0 = \boldsymbol{I}$
Calculate $\alpha_0 : f(\boldsymbol{x}_0 - \alpha_0 \nabla f_0)$ satisfies the Wolfe condition.
$\boldsymbol{x}_1 = \boldsymbol{x}_0 - \alpha_0 \nabla f_0$
$\boldsymbol{s}_0 = \boldsymbol{x}_1 - \boldsymbol{x}_0$
$\boldsymbol{y}_0 = \nabla f_1 - \nabla f_0$
$\boldsymbol{H}_0 = \frac{\boldsymbol{y}_0^T \boldsymbol{s}_0}{\boldsymbol{y}_0^T \boldsymbol{y}_0} \boldsymbol{I}$
for $k = 0$ *to* N **do**
  Calculate $\alpha_k : f(\boldsymbol{x}_k - \alpha_k \boldsymbol{H}_k \nabla f_k)$ satisfies the Wolfe condition.
  $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \alpha_k \boldsymbol{H}_k \nabla f_k$
  $\boldsymbol{s}_k = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k$
  $\boldsymbol{y}_k = \nabla f_{k+1} - \nabla f_k$
  $\boldsymbol{u}_k = \frac{\boldsymbol{s}_k}{\boldsymbol{s}_k^T \boldsymbol{y}_k} - \frac{\boldsymbol{H}_k \boldsymbol{y}_k}{\boldsymbol{y}_k^T \boldsymbol{H}_k \boldsymbol{y}_k}$
  $\boldsymbol{H}_{k+1} = \boldsymbol{H}_k + \frac{\boldsymbol{s}_k \otimes \boldsymbol{s}_k}{\boldsymbol{s}_k^T \boldsymbol{y}_k} - \frac{\boldsymbol{H}_k \boldsymbol{y}_k \otimes \boldsymbol{H}_k \boldsymbol{y}_k}{\boldsymbol{y}_k^T \boldsymbol{H}_k \boldsymbol{y}_k} + [\boldsymbol{y}_k^T \boldsymbol{H}_k \boldsymbol{y}_k] \boldsymbol{u}_k \otimes \boldsymbol{u}_k$

$\boldsymbol{x}_{sol} = \boldsymbol{x}_N$

7.1 L-BFGS

Quasi newton methods, such as LM and BFGS are not very suited for large scale problems since they usually require too much memory to operate. For big data problems with 100 million parameters, they would require 400 MB of memory if we store them with floating point precision and 800 MB with double precision. Understandably, we cannot store the square of this number, not even in a distributed environment. This is the reason why Low Memory BFGS is interesting.

L-BFGS stores a set of of m number of vectors of size n instead of a matrix of size n^2. These vectors are the curvature information from previous iterations that get discarded as we move along. The idea is that information from previous iterations are less likely to be relevant for the behavior of the current Hessian. Let us recall the update equations of the BFGS algorithm.

$$\boldsymbol{s}_k = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k \tag{183}$$

$$\boldsymbol{y}_k = \nabla f_{k+1} - \nabla f_k \tag{184}$$

$$\boldsymbol{u}_k = \frac{\boldsymbol{s}_k}{\boldsymbol{s}_k^T \boldsymbol{y}_k} - \frac{\boldsymbol{H}_k \boldsymbol{y}_k}{\boldsymbol{y}_k^T \boldsymbol{H}_k \boldsymbol{y}_k} \tag{185}$$

$$\boldsymbol{H}_{k+1} = \boldsymbol{H}_k + \frac{\boldsymbol{s}_k \otimes \boldsymbol{s}_k}{\boldsymbol{s}_k^T \boldsymbol{y}_k} - \frac{\boldsymbol{H}_k \boldsymbol{y}_k \otimes \boldsymbol{H}_k \boldsymbol{y}_k}{\boldsymbol{y}_k^T \boldsymbol{H}_k \boldsymbol{y}_k} + [\boldsymbol{y}_k^T \boldsymbol{H}_k \boldsymbol{y}_k] \boldsymbol{u}_k \otimes \boldsymbol{u}_k \tag{186}$$

So instead of storing the entire matrix $\boldsymbol{H}_k$, we now store m number of vector pairs $\{\boldsymbol{s}_k, \boldsymbol{y}_k\}$. By performing the Eq. 186 m number of times for the for the pairs, we obtain the below algorithm for computing $\boldsymbol{H}_k \nabla f_k$.

Algorithm 2: L-BFGS (Two-loop)

$\boldsymbol{g} = \nabla f_k$
for $i = k - 1$ *to* $k - m$ **do**
  $\rho_i = \frac{1}{\boldsymbol{y}_i^T \boldsymbol{s}_i}$
  $\gamma_i = \rho_i \boldsymbol{s}_i^T \boldsymbol{g}$
  $\boldsymbol{g} = \boldsymbol{g} - \gamma_i \boldsymbol{y}_i$
$\boldsymbol{H}_k^0 = \frac{\boldsymbol{y}_{k-1}^T \boldsymbol{s}_{k-1}}{\boldsymbol{y}_{k-1}^T \boldsymbol{y}_{k-1}} \boldsymbol{I}$
$\boldsymbol{r} = \boldsymbol{H}_k^0 \boldsymbol{g}$
for $i = k - m$ *to* $k - 1$ **do**
  $\lambda = \rho_i \boldsymbol{y}_i^T \boldsymbol{r}$
  $\boldsymbol{r} = \boldsymbol{r} + \boldsymbol{s}_i (\gamma_i - \lambda)$
$\boldsymbol{H}_k \nabla f_k = \boldsymbol{r}$

Algorithm 3: L-BFGS

Initialize $\boldsymbol{x}_0$, m
Calculate $\alpha_0 : f(\boldsymbol{x}_0 - \alpha_0 \nabla f_0)$ satisfies the Wolfe condition.
$\boldsymbol{x}_1 = \boldsymbol{x}_0 - \alpha_0 \nabla f_0$
$\boldsymbol{s}_0 = \boldsymbol{x}_1 - \boldsymbol{x}_0$
$\boldsymbol{y}_0 = \nabla f_1 - \nabla f_0$
for $k = 1$ *to* N **do**
  $\boldsymbol{p}_k = -\boldsymbol{H}_k \nabla f_k$ by Alg. 2
  Calculate $\alpha_k : f(\boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k)$ satisfies the Wolfe condition.
  $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k$
  if $k > m$ **then**
    Discard $(\boldsymbol{s}_{k-m}, \boldsymbol{y}_{k-m})$ from storage.
  $\boldsymbol{s}_k = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k$
  $\boldsymbol{y}_k = \nabla f_{k+1} - \nabla f_k$

7.2 Implementation Considerations

It is known that the BFGS approach has very powerful self-correcting properties of its Hessian. It is known that these properties only holds when an adequate line search is performed. I.e. a line search conforming to the Wolfe conditions. Furthermore, the line search should always try $\alpha_k = 1$ before proceeding to other values. The Wolfe conditions are given below.

$$f(\boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k) \leq f_k + c_1 \alpha_k \nabla f_k^T \boldsymbol{p}_k \tag{187}$$

$$\nabla f(\boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k)^T \boldsymbol{p}_k \geq c_2 \nabla f_k^T \boldsymbol{p}_k \tag{188}$$

The strong Wolfe conditions are given by

$$f(\boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k) \leq f_k + c_1 \alpha_k \nabla f_k^T \boldsymbol{p}_k \tag{189}$$

$$|\nabla f(\boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k)^T \boldsymbol{p}_k| \leq |c_2 \nabla f_k^T \boldsymbol{p}_k| \tag{190}$$

Typical values for (c_1, c_2) are $(10^{-4}, 0.9)$.

Another problem that we have both for L-BFGS and BFGS is the initial value(s) of $\boldsymbol{H}_0$. A heuristic that has proven itself to be very powerful here is to start calculating a steepest descent and with the new values set

$$\boldsymbol{H}_0 = \frac{\boldsymbol{y}_k^T \boldsymbol{s}_k}{\boldsymbol{y}_k^T \boldsymbol{y}_k} \boldsymbol{I} \tag{191}$$

before the BFGS update step. This method aims at estimating a matrix with similar eigenvalues to the true Hessian. Observe that this method still holds for L-BFGS but with the exception that we've previously stored the vectors $(\boldsymbol{y}_k, \boldsymbol{s}_k)$.

One practical problem is that the value $\boldsymbol{y}_k^T \boldsymbol{s}$ (the curvature condition) may be close to zero and therefore introduce singularities. For example, if we are in an area with very small variations (i.e. a long slow valley) we may have $\boldsymbol{y}_k^T \boldsymbol{s}$ dangerously close to zero.

8 Ideas

One idea is to evaluate a very accurate line search of the steepest descent in periods so that we can re-define the step size. Another idea is somehow evaluate descent directions as a floating mean so that we could get through valleys more quickly.

Another idea would be switch between different optimization methods where one is weak and the other is strong. A bit like LM but with less memory requirements.

Something that should be analyzed is the error curve along the descent direction for a neural net where convergence is slow.

For a neural network, we could analyze the change of the weights and check whether or not we only have a subset of them are important for a particular optimization task. If this is the case, could we reduce the dimension of the problem and apply more expensive optimization methods on these?

9 Software Implementation

We will concentrate our efforts on implementing the neural network using OpenCL. The reason is that we would like to have state-of-the-art performance on inhomogeneous platforms and truly utilize all the available resources to speed up the learning / classification / regression process. To truly squeeze out everything possible from the target device, one would probably need to dive down into the device's native programming language such as CUDA, Assembler or C. However, the effort in doing this is probably not worth it (consider, buying all the target hardware and manually tune the algorithms for each one). Therefore, we will use an empirical optimization method to create dynamic kernels that are the best for the given test parameters. In order to do this efficiently, we need to understand the differences between the GPU and the CPU.

By glancing at the multiplication tests as well as the convolution tests, we can see dramatic differences in performances between the different implementations. For example, local memory when using an Intel CPU, there's no local configuration that will speed up the convolution kernel. Thus, considering local memory when constructing a kernel for CPU can be discarded. Furthermore, the Intel OpenCL 1.2 Platform compiling for a Intel Core i7 4700MQ supports implicit vectorization as long as SoA are used instead of AoS. This means that explicit vectorization is often not needed.

- The CPU is relatively insensitive to local work group sizes. One could let the runtime determine the local work size for the CPU. To have the global size dividable by 2 will let the runtime to perform better choices. However, for work group sizes of 1 with very small work loads in every iteration can in some cases be extremely bad for performance.

- Do not use explicit local memory. The CPU is very good at putting relevant memory into the L1 cache implicitly. It's a waste of resources trying to explicitly allocate local memory as can be seen in the convolution test.
- Vectorization is often implicit, but there's usually no performance penalty to use OpenCL vectors if it makes the code more readable. One could potentially try with / without the vectorization attribute in this case to check whether or not the implicit vectorization performs better. This is mostly to safeguard against compiler without implicit vectorization.
- Do not bother unrolling loops. Loops can even be helpful with the Intel compiler since it can make implicit vectorization easier.

The GPU results look very similar between the Nvidia GPU and the integrated Intel HD GPU with the difference of how effective local memory optimization are. Normally, the GPU is divided into stream processors which in turn contains one instruction stream over multiple ALUs together with a shared memory over all the ALUs. Assuming this standard architecture, the local work groups should be mapped to the SMs with the local work size spread over the ALUs in the SM. If we map a single work item into every SM the performance should be slow since we are only using a fraction of the capacity of the SM. Evidence of this can be found in the test data when a local work group of size 1 is used. However, this is contradictory when doing multiple calculations inside the work-item with the same work size. Because, in this case we don't have the same performance bottleneck and the algorithm is able to perform slightly worse than the best result. It could be that the compiler / hardware in some cases have a hard time redistributing the work load into the other ALUs / SMs. Furthermore, trying to put large chunks of work into every work-item always yield worse performance in the vector multiplication example. Whereas for the CPU, the best result was obtained when accumulating a work size of 1000 inside a single work-item. One significant performance improvement can be obtained by explicitly using local memory and reducing the amount of global memory fetches. What's interesting to note here is that using local memory does not automatically make your algorithm faster for a specific GPU. For convolution, the improvements start to apply for filters greater than 2 for the Nvidia GPU. However, for the Intel HD GPU, we don't see any significant improvements before the filter size is 20 or greater. Before rounding up with the summation, let's look at how the performance is affected by multiple kernel launches compared to a single launch for a convolution layer of 4 inputs.

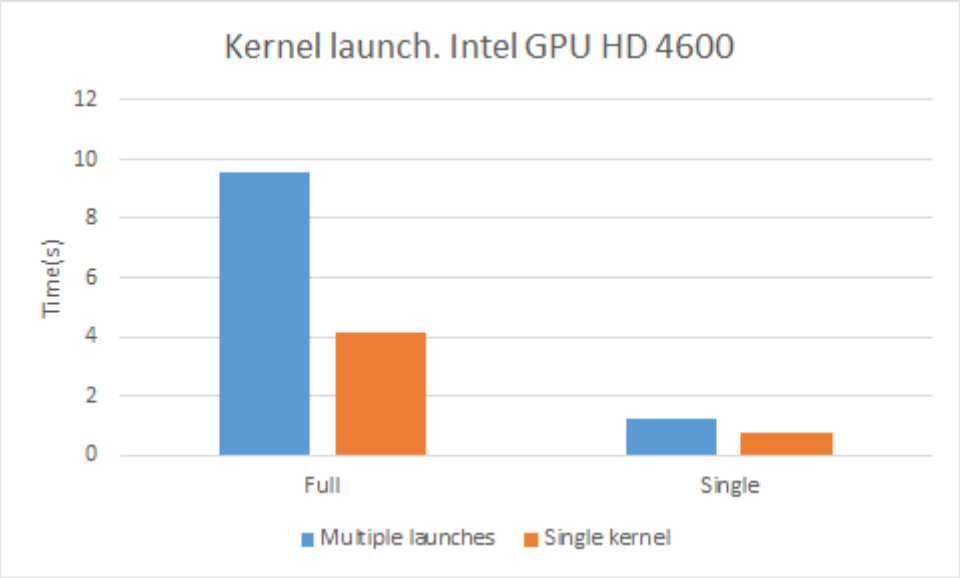

Figure 2: Showing the execution time of a convolutional layer using a single kernel and multiple kernel launches. Full means a fully connected convolutional layer with 4 inputs and 8 outputs. Single means single connections between 4 inputs and 4 outputs

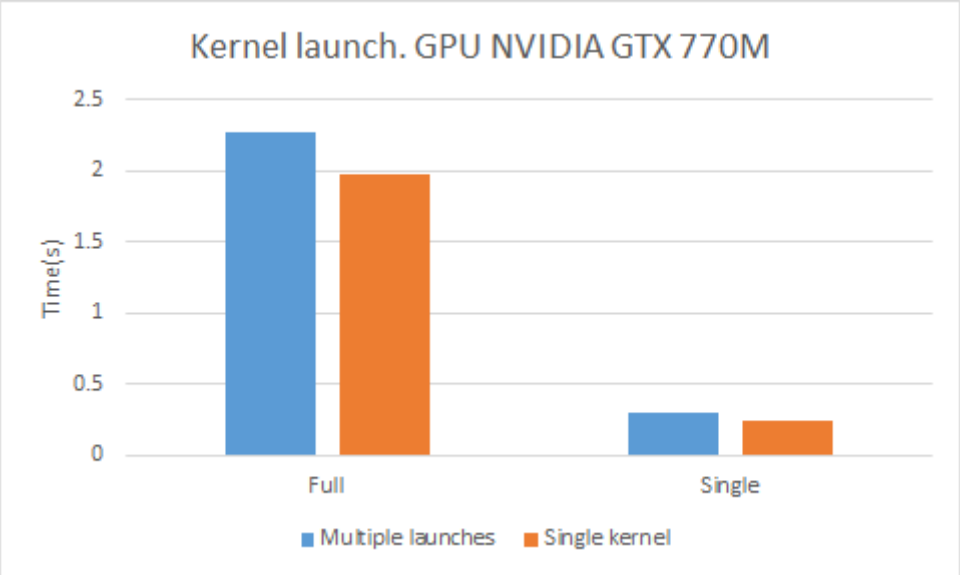

Figure 3: Showing the execution time of a convolutional layer using a single kernel and multiple kernel launches. Full means a fully connected convolutional layer with 4 inputs and 8 outputs. Single means single connections between 4 inputs and 4 outputs

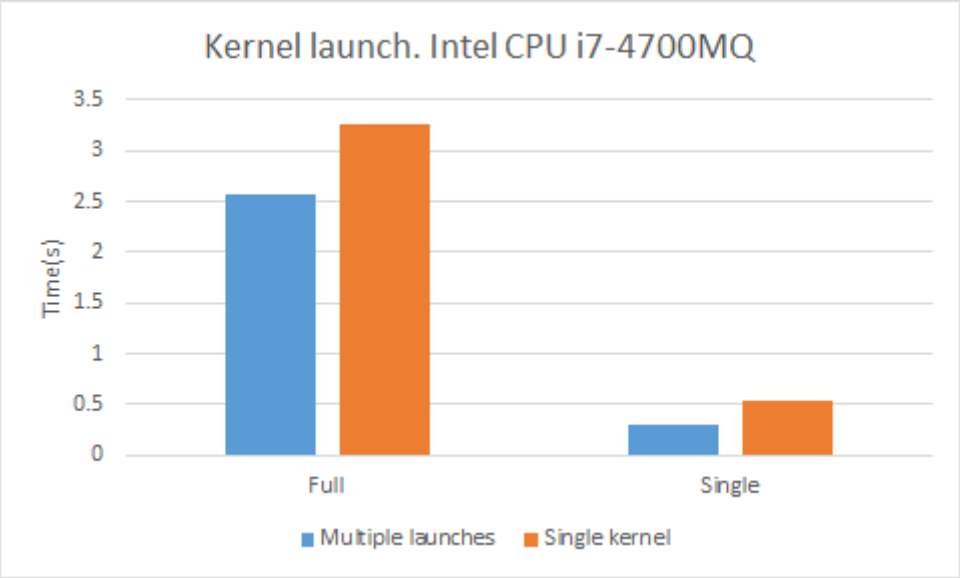

Figure 4: Showing the execution time of a convolutional layer using a single kernel and multiple kernel launches. Full means a fully connected convolutional layer with 4 inputs and 8 outputs. Single means single connections between 4 inputs and 4 outputs

Figure 2 and 3 looks natural since our intuition says that a kernel launch is expensive. It's important to note here that we don't sum the outputs from the multiple launches which is done in the single full kernel launch. So the full single kernel is actually more expensive than the multiple kernel launches. However, we can see something interesting when looking at Figure 4. Here multiple kernel launches seems actually more efficient than performing a single one. It could be argued that the kernel launch overhead is almost insignificant when using the CPU and the reduced amount of work becomes significant. Since the same kernel with the same arguments and memory is launched several times, it could also be that the OpenCL runtime notice this and can save significant initialization costs.

- The GPU does not have fancy branch prediction and smart caches so this must be handled manually.
- Small loops can degrade performance. Unrolling them can in some cases improve performance.
- Local memory does not automatically improve the performance of the GPU. The overhead of putting global memory into local memory has to be small in comparison with the performance gains.
- The local work size can have a significant impact. Especially in the case where local memory is used since it will be bound to the work size.
- The GPU is more sensitive to kernel launches than the CPU.

There are also other pitfalls one have to take into account when writing fast OpenCL code. For example, if you run OpenCL on the host (the CPU), then use calls to clEnqueueMapBuffer and clEnqueueUnmapBuffer instead of calls to clEnqueueReadBuffer or clEnqueueWriteBuffer. This will avoid unnecessary copying within the framework.

10 Automatically tuned OpenCL kernels

For a CPU, we don't care about local memory. For a GPU, we don't try batch computations in the work items since the scheduling is efficiently done. The general approach to all tuning is to define a default kernel for the CPU and GPU that will represent the benchmark for the problem. If automatic tuning is chosen, the algorithm will try to choose the fastest kernel among the modifiable parameters for the problem. This is typically testing the native / preferred vectorization width of the device and modifying the local work size together with the L1 cache size. Let us start with some notations for this chapter.

U	Number of parallel processing units.
S_x, S_y, S_z	The problem size dimensions
T_{max}	Maximum amounts of threads $T_{max} \geq U$
$S_{l,x}, S_{l,y}, S_{l,z}$	The local work group dimensions
$H_{image}, W_{image}, N_{image}$	Image height, width and number of images
F_x, F_y, N_{filter}	Filter height and width and the number of filters

Table 9: Fundamental weighed sub-sampling equations.

10.1 Convolutional layer

10.1.1 CPU

10.1.2 GPU

There's a big difference between a benchmark CPU kernel and a GPU kernel. In this case, we don't want to have thread affinity for the amount of processing units, which in turn contains a considerable amount of ALUs sharing the same decoder. If we were to have thread affinity for the GPU, we would only use a fraction of the power of the GPU (actually 1 divided by the amount of ALUs in every stream processor, assuming a naive architecture and compiler of course). Instead, since we don't have any scheduling overhead (for what I've detected so far), we simply have one work item for every pixel in the input. However, the focus here will lie in using local memory to speed up memory access. What we know at the moment is:

- Every pixel from the input must be read at least once in order for the convolution to work.
- But, the local memory is limited to a single local work group. The amount of memory that we can store in the cache is thus dependent on the local work group.
- We have boundaries that needs to be treated separately when performing convolution.

- The global memory accesses of the output map cannot be smaller than the amount of pixels in the output.
- The filters can be loaded into every cache on every streaming processor. However, this will reduce the amount of available local memory for the actual convolution and should be taken into account.

To formula for the number of global memory accesses needed for a naive convolution is given by:

$$S_z(S_x - R_x)(S_y - R_y)F_xF_y$$
$$R_x = F_x + 1$$
$$R_y = F_y + 1$$

And if we load the work group size together with the boundaries into local memory, we have:

$$S_z(S_x - R_x)(S_y - R_y) + S_{l,z}(S_x - R_x)(S_y - R_y)(\frac{R_x}{S_{l,x}} + \frac{R_y}{S_{l,y}} + \frac{R_xR_y}{S_{l,x}S_{l,y}} + 1)$$

We can directly see that the amount of global memory accesses decreases dramatically for a big local size in the x and y directions. However, we need to be careful with the conclusions here since we have additional complexity when choosing the local work group size.

- The local work group executes in parallel on a stream processor. Meaning that, if our local work group is as big as the problem itself, we cannot benefit from the parallel execution of several streaming processors.
- The filters will have as many memory accesses as the naive equation. A very simple way to make sure we don't access global memory for the filters is to put them onto the constant memory by using the constant modifier.
- The local work size should therefore be considered as the maximum local work size that can be distributed on all the streaming processors (as long as the filter is not equal to one). The second constraint is that the amount of local memory cannot be exceeded and this must be queried on the kernel.

We have additional constrains when defining the local memory size and using the constant memory.

- The local cache size cannot exceed the value returned by CL_DEVICE_LOCAL_MEM_SIZE - CL_KERNEL_LOCAL_MEM_SIZE before setting the cache size.
- The local work size cannot exceed CL_KERNEL_WORK_GROUP_SIZE.
- The filter parameters used for the convolution cannot exceed CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE if we are to use the constant memory space for the parameters. The minimum value is 64KB, which corresponds to 16 000 parameters. In turn, 16 000 parameters would correspond to about 55 21x21 filters in a single layer. This would only be a problem for extremely large problems. In this case, it would be acceptable to split the problem into two kernel calls (or more).

Since a greater value on $S_{l,z}$ than 1 will always increase the amount of global memory accesses. We will always assume that $S_{l,z} = 1$. Furthermore, we will always assume that the filter is in constant memory space. This leads to the following discrete constrained optimization problem for reducing the amount of global memory accesses:

$$(S_{l,x}, S_{l,y})_{min} = \min_{(S_{l,x},S_{l,y})} S_z(S_x - R_x)(S_y - R_y) +$$
$$\frac{(S_x - R_x)(S_y - R_y)}{S_{l,x}S_{l,y}}(R_xS_{l,y} + R_yS_{l,x} + R_xR_y + S_{l,x}S_{l,y})$$
$$(S_x - R_x) \mod S_{l,x} \equiv 0$$
$$(S_y - R_y) \mod S_{l,y} \equiv 0$$
$$S_{l,x} \leq S_{l,x,max}$$
$$S_{l,y} \leq S_{l,y,max}$$
$$\frac{(S_x - R_x)(S_y - R_y)}{S_{l,x}S_{l,y}} \geq U$$
$$S_{l,y}S_{l,x} \leq CL\_KERNEL\_WORK\_GROUP\_SIZE$$
$$R_xS_{l,y} + R_yS_{l,x} + R_xR_y + S_{l,y}S_{l,x} \leq M$$
$$M = CL\_DEVICE\_LOCAL\_MEM\_SIZE - CL\_KERNEL\_LOCAL\_MEM\_SIZE$$

However, as you can see, it's possible that the conditions that the local work group must be dividable with the global work group will not hold. In this case we need to transform the above optimization problem with padding to make sure that it may be solved.

$$(S_{l,x}, S_{l,y}, P_x, P_y)_{min} = \min_{(S_{l,x},S_{l,y},P_x,P_y)} S_z(S_x + P_x - R_x)(S_y + P_y - R_y) +$$
$$\frac{(S_x + P_x - R_x)(S_y + P_y - R_y)}{S_{l,x}S_{l,y}}(R_xS_{l,y} + R_yS_{l,x} + R_xR_y + S_{l,x}S_{l,y})$$
$$(S_x + P_x - R_x) \mod S_{l,x} \equiv 0$$
$$(S_y + P_y - R_y) \mod S_{l,y} \equiv 0$$
$$S_{l,x} \leq S_{l,x,max}$$
$$S_{l,y} \leq S_{l,y,max}$$
$$\frac{(S_x + P_x - R_x)(S_y + P_y - R_y)}{S_{l,x}S_{l,y}} \geq U$$
$$S_{l,y}S_{l,x} \leq CL\_KERNEL\_WORK\_GROUP\_SIZE$$
$$R_xS_{l,y} + R_yS_{l,x} + R_xR_y + S_{l,y}S_{l,x} \leq M$$
$$M = CL\_DEVICE\_LOCAL\_MEM\_SIZE - CL\_KERNEL\_LOCAL\_MEM\_SIZE$$

The solution to the above optimization problem is given by:

Algorithm 4: Calculates the optimal work size and padding with minimal global memory access.

```
OptimalLocalSize
Data: Kernel, Device
Result: (S_{l,x}, S_{l,y}, P_x, P_y)_min
Hypotheses
for P_x : from 0 to S_{l,x,max} do
    for P_y : from 0 to S_{l,y,max} do
        for S_{l,x} : from 0 to S_{l,x,max} do
            for S_{l,y} : from 0 to S_{l,y,max} do
                if (S_x + P_x - R_x) mod S_{l,x} ≢ 0 or
                (S_y + P_y - R_y) mod S_{l,y} ≢ 0 or
                S_{l,y}S_{l,x} > CL_KERNEL_WORK_GROUP_SIZE or
                ((S_x+P_x-R_x)(S_y+P_y-R_y)) / (S_{l,x}S_{l,y}) < U or
                R_xS_{l,y} + R_yS_{l,x} + R_xR_y + S_{l,y}S_{l,x} > M
                then
                    continue
                else
                    Hypotheses.add((S_{l,x}, S_{l,y}, P_x, P_y));
min = ∞
for Hypothesis in Hypotheses do
    accesses = S_z(S_x + P_x - R_x)(S_y + P_y - R_y) +
    ((S_x+P_x-R_x)(S_y+P_y-R_y)) / (S_{l,x}S_{l,y}) (R_xS_{l,y} + R_yS_{l,x} + R_xR_y + S_{l,x}S_{l,y})
    if min < accesses then
        (S_{l,x}, S_{l,y}, P_x, P_y)_min = (S_{l,x}, S_{l,y}, P_x, P_y)
        min = accesses
```

The above implementation is very naive and one should probably limit the padding to a fixed value of say 10 in order not to explode the computations. A common max value on the work dimension is around 1000 and by using this naive approach it could take way too long to calculate the benchmark kernel.

ADD TUNING

10.2 Pooling layer

10.2.1 CPU

10.2.2 GPU

10.3 Perceptron layer

10.3.1 CPU

10.3.2 GPU

References

[1] Convolutional neural network, . URL http://4myhappiness.info/cuda-implementation-of-convolutional-neural/.

[2] Conjugated gradients, . URL http://www.cs.cmu.edu/ quake-papers/painless-conjugate-gradient.pdf.