2 Convolution Neural Network Overview

In this section I will introduce convolutional filters and convolutional layers, activation function, normalization in neural network and optimizer for neural network. Since we are dealing with a regression problem in this project, the dimension of output vector of neural network is identical to input vector, pooling and fully connected layer will not be used in the network (Mixed-scale dense neural network) applied for the main problem in this project and I will not discuss them.

2.1 Convolutional filter and Convolution layer

2.1.1 Conventional Convolutional Filter

The convolutional filter is normally used in convolution neural networks, such as Alex-net, Res-Net networks from CNN family.

Actually the so-called convolution operation applied in convolution neural network is cross-correlation when strictly speaking, since the filer is combined with an input window without reversing the filter. The conventional convolutional filter is applied to image patches of the same size as the filter and strided according to stride value.

Convolutional operation is based on three key ideas that improve deep neural neural network, they are sparse interactions, parameter sharing and equivariant representations. [5]

Traditional neural network's fully-connected layers describe the interaction between each input and output unit using matrix multiplication by a matrix of parameters. Every output unit interacts with every input unit. Convolutional networks and convolutional layers, however, typically have sparse interactions. This is achieved by using smaller kernel compared to the input. For example, when processing an image, the input image might have thousands or millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels. It means that computing the output requires fewer operations. And this causes a local receptive field, meaning the neuron in next layer has limited connections to neurons in last layer instead of being connected to all neurons in last layer.

Parameter sharing refers to repeat use of the same parameters in a model. For example, in a traditional neural net, when computing the output of a layer, each element of the weight matrix was used exactly once. It is visited in the multiplication with one element of input, but never be revisited. The parameter sharing used by the convolution operation means that rather than learning diverse sets of parameters for different locations, it will learn only one set of kernel parameters.

The property of parameter sharing causes the layer to have a property called invariance to translation, which is similar to linear space invariance property, that is to say with formal convolutional computation by a kernel in image. The property of invariance to translation is the following: for example, let I be a function describing image intensity at integer coordinates. Let g be a function mapping one image function to another image , such that $I^{'} = g(I)$, with $I^{'}(x, y) = I(x - 1, y)$. This shifts every pixel of I one unit to the right. If we apply this transformation to I , and then apply convolution, the result will be identical to the convolution applied to I.

2.1.2 Conventional Convolutional Layer

We have a batch of image channels as input and the processing by conventional convolutional layer can be expressed by:

$$output(N_i, C_{out_j}) = bias(C_{out_j}) + \sum_{k=0}^{C_{in}-1} weight(C_{out_j}, k) \star input(N_i, k) \tag{2.1}$$

where $\star$ represents cross-correlation operation, N stands for batch, C stands for channel. C_{in} is the number of input channels, C_{out} is the number of output channels.

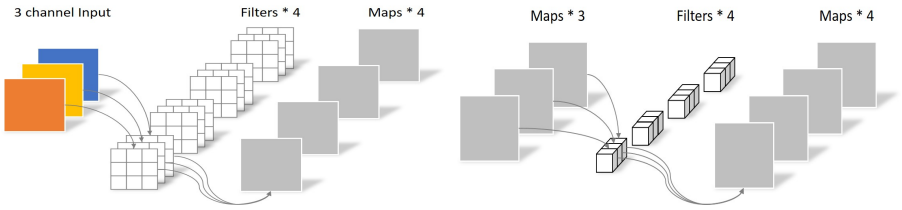
And the height of the output image channel can be expressed by:

$$H_{out} = \frac{H_{in} + 2 \times padding - dilation \times (kernel \quad size - 1) - 1}{stride} + 1 \tag{2.2}$$

where stride is the stride number, dilation is the dilation rate of the convolutional filter. The dilation is equal to 1 for conventional convolutional filter. And the expression for the output channel width is similar.

2.1.3 Pointwise Convolutional Layer

The pointwise convolution is very similar to the conventional convolution operation. The difference is that the size of the convolution kernel for pointwise convolution is $1\times1\times$M, where M is the depth(channel number) of the previous layer. For each filter, the pointwise convolution operation will compute a linear combination of channels of previous layer, generating a new feature map based on their weights. Finally, the number of newly generated feature maps is the same as the number of pointwise convolutional filters. It will be used in mixed-scale dense neural network.



(a) Conventional Convolutional Layer (b) Pointwise Convolutional Layer

Figure 2.1: Graphical Representation of Convolutional Layer

2.2 Activation Functions

In deep neural network, what happens if we use a linear activation function for each hidden layer? In this case, the deep neural network is equivalent to a network with only 1 hidden layer:

$$a = w^{'}(w^{T}x + b) = w^{'}w^{T}x + w^{'}b = w_2x + c \tag{2.3}$$

where $w^{'}$ stands for linear activation function. The above equation proves that the output a will still be a linear transformation of the input x if a linear activation function is applied.

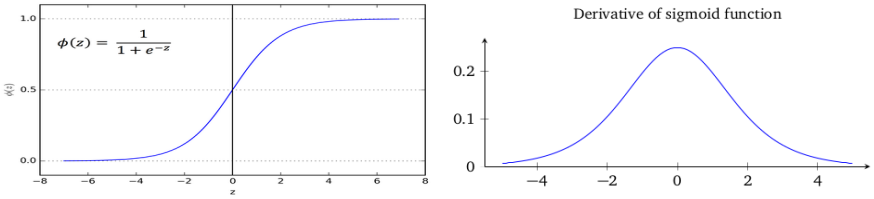
2.2.1 Sigmoid Function

The sigmoid activation function is non-linear and is defined as:

$$a = g(z) = \frac{1}{1+e^{-z}} \tag{2.4}$$

And its derivative form is given by:

$$g^{'}(z) = g(z)(1-g(z)) \tag{2.5}$$



(a) Sigmoid Function (b) Derivative of Sigmoid Function

Figure 2.2: Sigmoid Activation and Derivative

2.2.2 The Gradient Vanishing and Exploding Problem

Take the example of a network with three hidden layers.

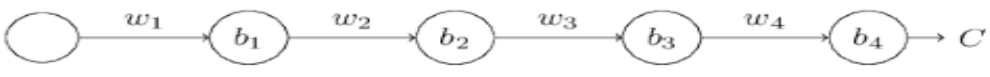


Figure 2.3: Example of 3 Hidden Layer

$$\frac{\partial C}{\partial b_1} = \sigma^{'}(z_1)w_2\sigma^{'}(z_2)w_3\sigma^{'}(z_3)w_4\sigma^{'}(z_4)\frac{\partial C}{\partial a_4} \tag{2.6}$$

Here, w_1, w_2,...are the weights, b_1,b_2,... are the biases added, and C is the cost function. The output a_j from the j-th neuron is $\sigma(z_j)$, where σ is sigmoid activation function, and $z_j = w_j a_{j-1} + b_j$ is the weighted input to the neuron.

As shown by Fig 2.2(b), the peak value of the derivative of the sigmoid is around 0.25, and usually $|w_j| < 1$ if we use our standard approach with initializing the weights in the network to be gaussian distributed, and thus $|w_j \sigma^{'}(z_j)| < \frac{1}{4}$. When taking a product of many such terms, the product will tend to decrease exponentially: the more terms, the smaller the product will be. This is a possible explanation for the vanishing gradient problem.

When $|w_j \sigma^{'}(z_j)| > 1$, we get an exploding gradient with non-careful weights initialization. However, gradient exploding is very less likely to happen than gradient vanishing since the possibility of $|w| > 4$ is very small unless we initialize the weights with very large values instead of standard weight parameter initialization method[6].

In fact, when using sigmoid, the gradient will usually vanish. To avoid the vanishing gradient problem, we need $|w\sigma^{'}(z)| \geq 1$. It seems like this could be reached with larger w, however, $\sigma(z)$ also depends on w since $\sigma^{'}(z) = \sigma^{'}(wa+b)$, where a is the input activation. When w is getting large, it is necessary to prevent $\sigma^{'}(wa + b)$ from being small. That turns out to be a considerable constraint because large w tend to cause large $wa + b$. The only way to avoid it is to make the values of input activation fall within a fairly narrow range. It means whether that gradient vanishing or exploding depends on values of activation neurons. I will introduce some normalization methods in the following part.

2.2.3 Rectified Linear Unit Function

Another activation function is rectified linear unit, it has the form:

$$g(z) = max[0, z] \tag{2.7}$$

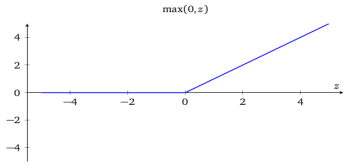


Figure 2.4: Rectified Linear Unit

It has some advantages over the sigmoid function. Basically the training of neural network is an optimization process for a cost function defined at the output of the neural network and optimization is normally gradient based, with an iterative update of the parameters.

The derivative of ReLU is existing and equal to 1 everywhere except when input z is zero where the derivative doesn't exist. Training convergence is easier with ReLU and avoid either the gradient vanishing or gradient exploding. It also has lower computation complexity by its simple form.

2.2.3.1 Dead ReLU Problem

However, Relu has unavoidable drawbacks. The output of ReLU is not zero mean and dead ReLU problem may happen [7]. As implied by Eq.2.6, the gradient with respect to weight parameters in successive layers can be equal to zero.This phenomenon is mainly caused by bad parameter initialization and large learning rate. To overcome it, we can apply the Xavier initialization method mentioned by [6] or use adaptive optimizer that can modulate learning rate adaptively during training.

Let us consider the equations for backpropagation [8]:

$$\delta^l = \bigtriangledown_a C \odot \sigma^{'}(z^l) \tag{2.8}$$

where the vector δ stands for the change of activation values of each neuron in a layer l. So δ^l is a vector representing the change on the activation value of a neuron in the output layer. C stands for the cost function, so $\bigtriangledown_a$ is gradient taken with respect to activation value of neuron in the output layer. z^l is the weighted sum for neuron in the output layer before activation, and σ stands for activation function applied on the neuron.

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma^{'}(z^l) \tag{2.9}$$

where w stands for the weight matrix between neurons in l-th and $(l+1)$-th layers.

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{2.10}$$

where b_j^l stands for the bias of j-th neuron in l-th layer.

$$\frac{\partial C}{\partial w^l_{j,k}} = a^{l-1}_k \delta^l_j \tag{2.11}$$

where a^{l-1}_k stands for activation value of k-th neuron in $(l-1)$-th layer , $w^l_{j,k}$ stands for the weight value of k-th neuron in $(l-1)$-th layer to j-th neuron in l-th layer .

Considering a situation where a certain neuron in a layer gets its weighted sum to be less than 0 in forward propagation, meaning that the input to ReLU falls in its negative input domain, the activation values would be 0. As implied by Eq.2.11, $\frac{\partial C}{\partial w^l_{j,k}}$ would be equal to 0 since a^{l-1}_k is equal to 0. It means the weight values to all the neurons in next layer would be equal to zero. Therefore, it is not possible to update the weight parameter value in backpropagation, unless the weighted sum returns a value larger than 0 by linear affine transformation. In further forward propagation rounds which depends on the distribution of activation values of neurons in the previous layer, this neuron could not be recovered. If the weighted sum of neurons in a layer are mostly negative values, most neurons are not able to update their weight parameter values in further back propagation steps. If this happens in early layers, even the first hidden layer, most neurons couldn't be recovered , at this moment, the dead ReLU problem becomes serious. And the learning rate in parameter update should not to be too large, otherwise, there would be more possibility that the distribution of weighted sum of neuron falls in the negative input domain of ReLU. I will introduce adaptive optimizer in next part.

I will introduce the z-scored normalization, layer normalization and batch normalization that prevent serious dead ReLU problem from happening in following part. However, dead ReLU problem is not always bad, I will also introduce its positive effect in Dense net in section 3.

Further, there are some variants of ReLU, like maxout and leaky ReLU. In mixed-scale dense network, ReLU was applied.

2.3 Optimizers

Gradient descent is so far the most common and popular way to optimize neural networks. Gradient descent is a way to minimize the objective function $J(\theta)$ parameterized by a model's parameters $\theta \in R$ d by updating the parameters in the opposite direction of the gradient of the objective function $\bigtriangledown_\theta J(\theta)$ w.r.t. to the parameters [9]. The learning rate η determines the size of the steps we take to find the local minimum. In other words, we follow the direction of the slope of the surface created by the objective function downhill until we reach a valley [9].

For functions with multiple inputs, i.e. when $\boldsymbol{\theta}$ is a vector, we make use of partial derivative $\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_\mathbf{i}}$ to measure how $J(\boldsymbol{\theta})$ changes as only the variable $\boldsymbol{\theta}_\mathbf{i}$ changes at point $\boldsymbol{\theta}$. The gradient generalizes the notion of derivative to the case where the derivative is w.r.t vector. The gradient of $J(\boldsymbol{\theta})$ is the vector containing all of the partial derivatives, denoted by $\bigtriangledown J(\boldsymbol{\theta})$ [5].

The directional derivative in direction of unit vector $\mathbf{u}$ is the slope of function $J(\boldsymbol{\theta})$ in direction of $\mathbf{u}$. The directional derivative is the derivative of the function $J(\boldsymbol{\theta} + \alpha\mathbf{u})$ with respect to α evaluated at $\alpha = 0$. By the chain rule, $\frac{\partial J(\boldsymbol{\theta}+\alpha\mathbf{u})}{\partial \alpha} = \mathbf{u}^T \bigtriangledown_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. To minimize $J(\boldsymbol{\theta})$, we can find the direction in which $J(\boldsymbol{\theta})$ decreases with fastest speed. We can do this using the directional derivative: [5]

$$\min_{\mathbf{u},\mathbf{u}^T\mathbf{u}=1} \mathbf{u}^T \bigtriangledown_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \quad = \quad \min_{\mathbf{u},\mathbf{u}^T\mathbf{u}=1} ||\boldsymbol{\theta}||_2 || \bigtriangledown_{\boldsymbol{\theta}} J(\boldsymbol{\theta})||_2 \cos\kappa \tag{2.12}$$

where κ is angle between $\mathbf{u}$ and the gradient. This is minimized when $\mathbf{u}$ points in the opposite direction of the gradient. In other words, the gradient points directly uphill, and the negative gradient points directly downhill. J can be decreased by moving in direction of negative gradient. This is known as gradient descent. [5]

The gradient descent proposes a new point :

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \bigtriangledown_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{2.13}$$

2.3.1 Batch Gradient Descent

Batch gradient descent computes the gradient of the cost function w.r.t. to parameters $\boldsymbol{\theta}$ based on entire training dataset as Eq.2.13. We perform one update by computing gradients based on the whole dataset, it can be very slow. However, batch gradient descent performs redundant calculations based on large datasets, since it recomputes gradients for similar examples before each update of parameter. Therefore, batch gradient descent can basically guarantee convergence to global minimum for convex error surfaces and to a local minimum for non-convex surfaces[9]. We will not use it practically because of large computation cost.

2.3.2 Stochastic Gradient Descent

The stochastic gradient descent performs parameter update based on each training example $x^{(i)}$ and label $y^{(i)}$.

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \cdot \bigtriangledown_{\boldsymbol{\theta}} J(\boldsymbol{\theta}; x^{(i)}; y^{(i)}) \tag{2.14}$$

The stochastic gradient descent can learn online as new data comes and update faster, but performs frequent updates with high variance which causes cost function to fluctuate heavily[9]. However, it has been proved that when slowly decreasing learning

rate, SGD also shows convergence behaviour, almost certainly converging to a local or the global minimum for non-convex and convex optimization respectively.

The first downside of stochastic gradient descent is the difficulty to select the optimal learning rate. The lower learning rate will cause slow convergence rate, while higher learning rate will result in hindered convergence and cause the cost function to fluctuate around the minimum or even to diverge.

Secondly, all parameters to be updated share the same learning rate, this is the drawback for all of BGD, SGD and MBGD. Since different parameters show different behaviours in minimizing the cost function, the same learning rate for them would lead to worse convergence and worse minimum value. As implied by the dead ReLU problem mentioned in last part, the parameter update is better to be carefully performed with a suitable step size, otherwise, there would be more possibility to let more and more neurons fall in dead region, especially for early layers. If the data is sparse and our features have very different frequencies, it's not a good option to update all of them to the same extent, but it is more efficient to perform a greater update for rarely occurring features. [9]

Finally, stochastic gradient descent is easier to get stuck in local minimum region, and sometimes saddle points as well [10].

2.3.3 Mini-Batch Gradient Descent

Mini-batch gradient descent takes the advantages of BGD and SGD, performing update based on each mini-batches of n training examples:

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \cdot \bigtriangledown_{\boldsymbol{\theta}} J(\boldsymbol{\theta}; x^{(i:i+n)}; y^{(i:i+n)}) \tag{2.15}$$

It reduces the variance of the parameter updates, which can lead to more stable convergence. The mini-batch gradient descent, however, does not guarantee good convergence, but share the same drawbacks as mentioned in stochastic gradient descent.

2.3.4 ADAM Optimizer

Adam is an optimization algorithm based on first-order gradient for stochastic objective functions, it is based on adaptive estimates of lower-order moments. The objective functions are often stochastic. For example, many objective functions are composed of a sum of subfunctions evaluated at different subsamples of data; for such case optimization could be more efficient by taking gradient steps w.r.t. individual subfunctions, i.e. stochastic gradient descent. ADAM method as implied by its name, i.e. adaptive moment estimation, computes individual adaptive learning rates for different parameters by estimating first and second moments of gradients [11].

Basically the stochasticity may come from the evaluation at random subsamples of datapoints, single data or a mini-batch of data, or arise from inherent function noise. Assume the noisy stochastic loss function $J(\boldsymbol{\theta})$ is differentiable w.r.t $\boldsymbol{\theta}$, and let the stochastic functions at subsequent timesteps $1, ..., T$ be $J_1(\boldsymbol{\theta}), J_2(\boldsymbol{\theta}), ..., J_T(\boldsymbol{\theta})$, and $g(\boldsymbol{\theta})_t = \bigtriangledown_{\boldsymbol{\theta} J_t(\boldsymbol{\theta})}$ be gradient at timestep t. We shift our focus to minimize $E[J(\boldsymbol{\theta})]$ w.r.t $\boldsymbol{\theta}$ instead of $J_t(\boldsymbol{\theta})$ for each timestep (each iteration of parameter update). We take the first and second moments of the gradient $g(\boldsymbol{\theta})_t$ at timestep t into account:

$$m_t = \sum_{\tau=1}^{t} g_\tau = E[g(\boldsymbol{\theta})] \qquad v_t = \sum_{\tau=1}^{t} g_\tau^2 = E[g(\boldsymbol{\theta})^2] \tag{2.16}$$

ADAM stores the exponentially decaying average of the past squared gradients v_t, which is an estimate of the second moment (the uncentered variance) of the gradients and an exponentially decaying average of the past gradients m_t, which is an estimate of the first moment(mean) of the gradients.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \qquad v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{2.17}$$

where $\beta_1, \beta_2 \in [0, 1)$ control m_t and v_t.

If m_t and v_t are initialized to be zero vectors, which causes moment estimates to be biased towards zero, such initialization bias can be easily counteracted by choosing [11].

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{2.18}$$

Then, the parameter update follows the rule:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\eta}{\sqrt{\hat{v}_t}} \hat{m}_t \tag{2.19}$$

The ADAM is appropriate for non-stationary objectives and problems with very noisy or sparse gradients.This is quite fitting our case, I will explain the reason why it fits our case in terms of noise and sparse gradient in next section.

However, ADAM has built-in limitations. ADAM may not have convergence for some situations [12]. The detailed theoretical proof was made by [12]. In short, unlike AdaGrad [13], the estimate of second moment v_2 for ADAM is taken within a window of fixed size as time goes by, in contrast, the estimate of second moment v_2 for AdaGrad is taken cumulatively from the very beginning. So, the data used for second moment estimate may undergo considerable change as window moves time by

time. the learning rate of ADAM may not monotonically decreases as in AdaGrad [13]. I attach some training loss evolution graphs in Appendix section, showing optimizer ADAM's non-convergence property for some times.

Even though ADAM has such drawbacks, it has faster convergence speed than normal stochastic gradient descent and mini-batch gradient gradient descent, and since it is quite fitting with stochastic objective(cost) function, I will apply it in the form of SGD and MBGD in experiments.

2.4 Normalization

There are several ways of data preprocessing to process data before feeding into neural network, they are feature scaling, such as min-max normalization that rescales range of features to be $[0, 1]$ or $[-1, 1]$, and standardization (z-score normalization) that put all features on the same scale, converting data distribution to be zero mean and unit variance, but not necessarily to be Gaussian distributed. The z-score based normalization is also applied in hidden layers, i.e.batch normalization[14], layer normalization[15]. The common idea is to convert data distribution to be more smooth to deal with or restore distribution of intermediate images in deep neural network to be like the distribution of the input, i.e.reducing Internal Covariance Shift[14].

2.4.1 Batch Normalization, Layer Normalization and Instance Normalization

The Internal Covariance Shift is defined as the change in distribution of network activations caused by change in network parameters during training. The aim of normalization is to reduce the Internal Covariance Shift.

Actually the independent and identically distributed (i.i.d.) random variables can simplify training of machine learning models, increasing the model convergence speed. The whitening such as principle component analysis based whitening is a good preprocessing method to whiten data [16]. However, this kind of method is very computationally consuming and they may not have proper differentiable property for backpropagation for hidden layer. So, some z-score based feature normalization methods such as batch normalization, layer normalization, group normalization [17], instance normalization [3] are proposed. A common framework of them can be formulated as:

$$\mu_i = \frac{1}{m}\sum_{k\in\mathbf{S}_i} x_k \qquad \sigma_i = \sqrt{\frac{1}{m}\sum_{k\in\mathbf{S}_i}(x_k-\mu_i)^2+\epsilon} \qquad \hat{x_i} = \frac{1}{\sigma_i}(x_i-\mu_i) \qquad \hat{y_i} = \gamma\hat{x_i}+\beta \tag{2.20}$$

where x is the feature computed by a layer, and i is the index. In case of 2D images, $i = (i_N, i_C, i_H, i_W)$ is a 4D vector indexing the features in (N, C, H, W) order, where N is batch axis, C is channel axis, H and W are spatial height and width axes.

ϵ is tiny positive constant to avoid zero σ_i. S_i is the set of pixels in which the mean and standard deviation are computed, and m is the size of this set. Many types of feature normalization methods mainly differ in how the set S_i is defined. μ and σ are mean and standard deviation. γ and β are trainable scale and shift. $\hat{y_i}$ is output of feature normalization.

The batch normalization is inspired by image whitening and specifically designed for MBGD whereas layer and instance Normalization is for SGD.The set S_i of BN, S_j of LN and S_q of IN are respectively defined as:

$$\mathbf{S}_i = \{k| \quad k_C = i_C \quad\} \qquad \mathbf{S}_i = \{k| \quad k_N = i_N \quad\} \qquad \mathbf{S}_i = \{k| \quad k_C = i_C \quad k_N = i_N \quad\} \tag{2.21}$$

where i_C (and k_C) denotes the sub-index of i (and k) along the C axis. This means that the pixels sharing the same channel index are normalized together for Batch Normalization, i.e. for each channel, BN computes μ and σ along the (N, H, W) axes.

And i_N (and k_N) denotes the sub-index of i (and k) along the N axis,so LN computes μ and σ along the (C, H, W) axes for each sample.

IN computes μ and σ along the (H, W) axes for each sample and each channel. Graphical comparison is put in Appendix.

2.4.2 Standardization of Mixed-scale Dense Network Input

The z-score normalization is applied to mixed-scale dense network input. Its S_i selection seems taking reference from BN and IN. A total of $N \times C$ means and standard deviation values are computed based on each input channel of all batch of samples, an average is calculated, this becomes the global mean and standard deviation used for following standardization to be applied on all neurons. Different from BN and IN, it doesn't have trained parameters γ and β to be learned:

$$\begin{gathered}\mu_i = \frac{1}{m}\sum_{k\in\mathbf{S}_i} x_k \qquad \sigma_i = \sqrt{\frac{1}{m}\sum_{k\in\mathbf{S}_i}(x_k-\mu_i)^2} \qquad \mathbf{S}_i = \{k| \quad k_C = i_C \quad k_N = i_N \quad\} \\ \mu = \frac{1}{n}\sum_{l=1}^{N\times C}\mu_l \qquad \sigma = \frac{1}{n}\sum_{l=1}^{N\times C}\sigma_l \qquad \hat{x_i} = \frac{1}{\sigma}(x_i-\mu) \qquad \mathbf{S}_i = \{k| \quad k_C = i_C \quad k_N = i_N \quad\}\end{gathered} \tag{2.22}$$