APPENDIX A

BackPropagation in Deep Neural Networks

The backpropagation, is a widely used method in training artificial feed forward neural networks (FFN). Training ANN has two steps : error backward propagation and weight update. When an input vector is presented to the network, it is propagated forward through the network, layer by layer, until it reaches the output layer. The output of the network is then compared to the desired output, using a loss function, and an error value is calculated for each of the neurons in the output layer. The error values are then propagated backwards (hence the name backpropagation), starting from the output, until each neuron has an associated error value which roughly represents its contribution to the original output. Once the error at each neuron is determined, the weights are updated using an optimization method like gradient descent so as to minimize the loss function. we show that the error back propagation can be conveniently done in matrix notion in almost similar manner from layer to layer. Further, it will be shown that the weight update term in gradient descent can be easily computed based on the back propagated error at the rear end of the connection (carrying the weight) and the input of it. This process thus gives a simpler interpretation and derivation for weight update in deep networks. In this section, we discuss a generalized formulation of deep neural network training using back propagation.

A.1 Notations and Meaning

T_z	Target or desired output at node z in the final layer
O_z	Output produced at some arbitrary node z in some layer
ΔE_L	Error computed or backpropagated at layer L
ΔEF_L	Error weighted by the derivative of activation function at layer L
US_2	The operation up-sampling by 2
'*'	The convolution operator

$\odot$	The correlation operator
'.*'	Element by element multiplication

A.2 Training FFN using backpropagation

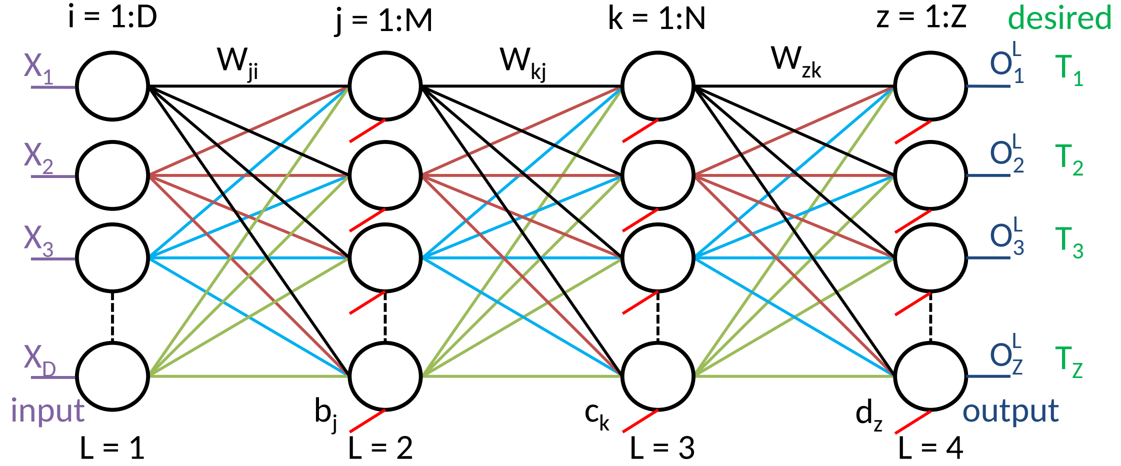

Figure A.1: Typical feed forward 4 layer neural network

Figure A.1 shows an arbitrary 4 layer feed forward neural network. There are D neurons in the input layer, M neurons in 2nd layer, N neurons in 3rd layer and Z neurons in the output layer. There is a weighted connection between each neuron in a layer to every other neurons in the layer immediately following it. Also there is a bias to every neurons in all layers except at the input layer. During training these parameters (weights and biases) need to be updated so as to minimise the loss function. A typical loss function could be the half of the sum of squared difference defined by

$$E = \sum_z \frac{1}{2}(T_z - O_z^L)^2 \tag{A.1}$$

Here T_z and O_z^L are the desired output and actual output produced at node z of the output layer L for any specific input $X = \{X_i\}_{i=1}^D$. Note that the notion of input to the normal feed forward neural network is a column vector. If the input is an image, it has to be vectorised to form the input vector. $\{X_i\}_{i=1}^D$ forms the

column vector X with D elements as X_i.

All nodes, except at the input layer, compute the weighted sum of inputs from the previous layer to produce an intermediate output Net. The activation function is applied to each of this Net to produce the final output at each neuron. We will consider the popular Sigmoid activation function defined by

$$O_z^L = f(Net_z^L) = \frac{1}{1 + \exp^{-Net_z^L}} \tag{A.2}$$

where Net_z^L is the intermediate output produced by node z in layer L and is defined by

$$Net_z^L = \sum_k W_{zk} O_k^{L-1} + d_z \tag{A.3}$$

Here O_k^{L-1} is the output at node k in $L - 1^{th}$ layer and W_{zk} is the weights of the connections from node k in layer $L - 1$ to node z in layer L.

We use gradient descent to update the parameters. Each parameter is updated in the negative direction of the gradient of the loss function (Eq. A.1) computed with respect to the parameter to be updated. The procedure is going to be explained in two steps. In the first step we will compute the contribution of the error to the over all loss function at each node. The error at each node in the output layer is first computed. Then the contribution by each node of the layers lower in the hierarchy is computed by backpropagating the error. In the second step, we will compute the gradient and update the parameters. Subsection A.2.1 discusses the error back propagation procedure while subsection A.2.2 discusses the parameter updating procedure. In all these discussion, we refer the final layer as L, pre-final layer as $L - 1$, the next lower layer as $L - 2$ and so on. The main theme of the presented formulation is that either error back propagation or the weight update based on it can be extended to any number of layers (on similar steps).

A.2.1 Backpropagating the Error across the layers

We will compute the error at each node in the output layer and is backpropgated to the layers lower in the hierarchy to determine the contribution of this error by each node in the network.

Finding Error at Output Layer L

Each node in the output layer contribute to the total error E. The contribution of Error by a node z at Layer L is due to its output O_z^L. Let ΔE_z^L denote this quantity and can be measured by computing $\frac{\partial E}{\partial O_z^L}$. By the definition of the error function (Eq. A.1), this turns out to be

$$\Delta E_z^L = \frac{\partial E}{\partial O_z^L} = O_z^L - T_z \tag{A.4}$$

Propagating the error from Layer L to layer $L-1$

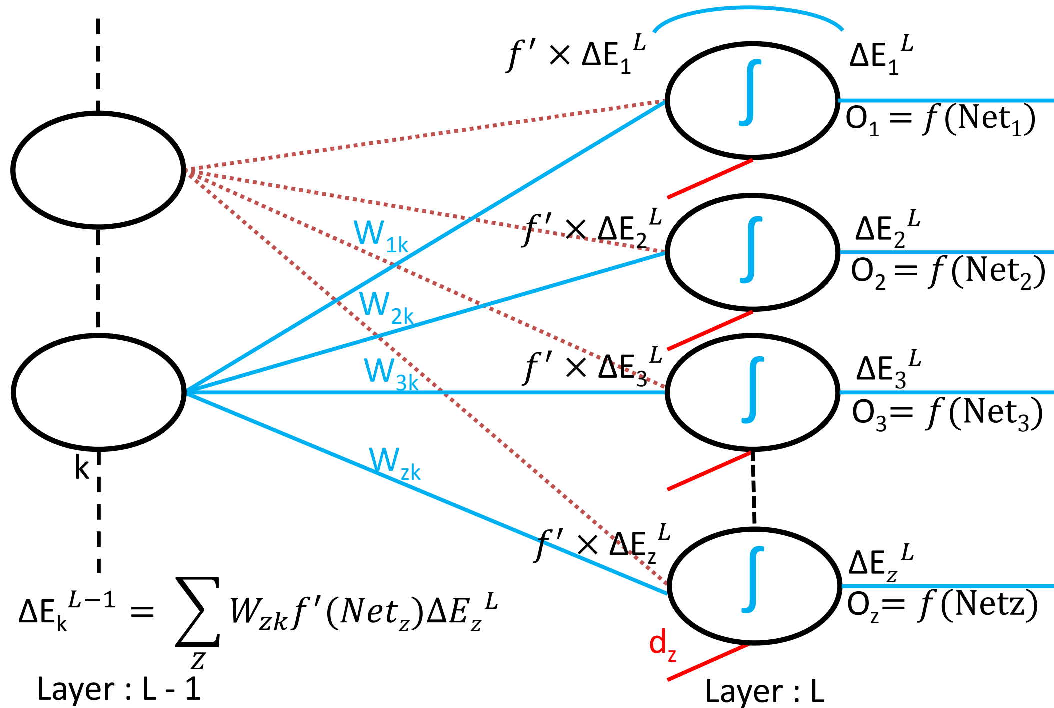

Figure A.2: Error backpropagation from nodes in layer L to node k in layer $L-1$.

First, we will give an intuitive idea behind the backpropagation of error and

then we will derive explicit expressions. In order to find the contribution of the final error E at layer L due to the output of a node k in layer $L-1$, the error that we have found out at each node in L has to be backpropagated. Once the error ΔE_z^L at all nodes z in an arbitrary layer L is computed, it has to cross these neurons during backpropagation of the error to compute the error at all nodes in the lower layer $L-1$. This is done by weighing ΔE_z^L by the corresponding derivative of the sigmoid function $f^{'}(Net_z^L)$, and then accumulating the shares through the weighted connections at each node k in layer $L-1$. The procedure can be explained better by referring to Fig A.2. It shows the relationship to each of ΔE_z^L from the node k in layer $L-1$. It can be seen that node k in layer $L-1$ influences the output at all nodes in L through the respective weight connections. Thus the error computed at each neuron in layer L contributes through the respective weight connection to each node k in layer $L-1$ and it has to cross each neuron z in layer L. Thus the Error at a node k in layer $L-1$ can be computed as

$$\Delta E_k^{L-1} = \frac{\partial E_z^L}{\partial O_k^{L-1}} = \sum_z \frac{\partial E_z^L}{\partial O_z^L} f^{'}(Net_z^L) W_{zk} \tag{A.5}$$

Where $f^{'}(Net_z^L$ represents the derivative of the activation function with respect to Net value at node z in layer L. By its definition (Eq A.2), this turns out to be $O_z^L(1-O_z^L)$. In matrix form, this can be written as

$$\Delta E^{L-1} = \mathbf{W^T} * \left[\Delta E^L . * f^{'}(Net^L)\right] \tag{A.6}$$

Where '.*' represents the element by element multiplication, '*' represents the matrix multiplication, and $\mathbf{T}$ represents the matrix transpose. The Eq. A.6 remains same for back propagation of error through any number of layers. The weight matrix $\mathbf{W}$ changes from layer to layer. The $f^{'}$ of the corresponding activation function (with associated outputs) and error at current layer are used to backpropagate the error to its previous layer.

The expression in Eq. A.5 can also be explicitly computed using the normal chain rule for gradient computation and is shown below.

$$\frac{\partial E_z^L}{\partial O_k^{L-1}} = \sum_z \frac{\partial E_z^L}{\partial O_z^L} \frac{\partial O_z^L}{\partial Net_z^L} \frac{\partial Net_z^L}{\partial O_k^{L-1}} \tag{A.7}$$

Where $\frac{\partial Net_z^L}{\partial O_k^{L-1}}$ can be computed as

$$\frac{\partial Net_z^L}{\partial O_k^{L-1}} = \frac{\partial}{\partial O_k^{L-1}} \sum_k W_{zk} O_k^{L-1} = W_{zk} \tag{A.8}$$

When substituted Eq. A.8 in Eq. A.7, it turns out that

$$\frac{\partial E_z^L}{\partial O_k^{L-1}} = \sum_z \frac{\partial E_z^L}{\partial O_z^L} \frac{\partial O_z^L}{\partial Net_z^L} W_{zk} \tag{A.9}$$

which is exactly the same equation provided in Eq. A.5. The consequence of Eq. A.6 is that the error backpropagation can be done independent of gradient update, and can be propagated back in each layers lower in the hierarchy one by one by simple matrix multiplication.

A.2.2 Updating the parameters

As noted earlier, the optimisation method used to minimise the loss function defined in Eq. A.1 could be gradient descent. The parameters are updated in the negative direction of the gradient of the error function computed with respect to the parameter to be updated.

Updating the weights

The update rule based on the gradient descent is

$$W_{zk}^{new} = W_{zk}^{old} - \eta \frac{\partial E}{W_{zk}} \tag{A.10}$$

Here η is the learning parameter and $\frac{\partial E}{W_{zk}}$ is the gradient of the loss function E with respect to the parameter W_{zk}. The $\frac{\partial E}{W_{zk}}$ can be computed using chain rule

$$\frac{\partial E}{\partial W_{zk}} = \frac{\partial E}{\partial O_z} \frac{\partial O_z}{\partial W_{zk}} \tag{A.11}$$

Here, the first term $\frac{\partial E}{\partial O_z}$ for all nodes in all layers is already computed by error backpropagation as discussed in the last section. The second term can be com-

puted by chain rule as

$$\frac{\partial O_z}{\partial W_{zk}} = \frac{\partial O_z}{\partial Net_z}\frac{\partial Net_z}{\partial W_{zk}} \tag{A.12}$$

Where $\frac{\partial O_z}{\partial Net_z}$ is the derivative of sigmoid function and is defined by

$$\frac{\partial O_z}{\partial Net_z} = O_z(1 - O_z) \tag{A.13}$$

$$\frac{\partial Net_z}{\partial W_{zk}} = \frac{\partial}{\partial W_{zk}} \sum_k W_{zk} O_k^{L-1} = O_k^{L-1} \tag{A.14}$$

W_{zk} is the weight connecting node k in layer $L - 1$ to node z in layer L, $\frac{\partial Net_z}{\partial W_{zk}}$ turns out to be the output at the node k at layer $L - 1$. Thus $\frac{\partial O_z^L}{\partial W_{zk}}$ turns out to be the output of the neuron k at layer $L - 1$ weighted by the derivative of the sigmoid with respect to Net_z^L at node z in layer L.

Updating the weights for the network shown in Fig. A.1

Let $\mathbf{W}_{H_1}$, $\mathbf{W}_{H_2}$ and $\mathbf{W}_{H_3}$ be the weight matrices of the network holding respectively the weights between layer 1 & 2, 2 & 3 and 3 & 4.

$$W_{H_1} = \begin{bmatrix} W_{11} & W_{12} & W_{13} & . & . & . & W_{1D} \\ . & . & . & . & . & . & . \\ . & . & . & W_{ji} & . & . & . \\ . & . & . & . & . & . & . \\ W_{M1} & W_{M2} & W_{M3} & . & . & . & W_{MD} \end{bmatrix}$$

$$W_{H_2} = \begin{bmatrix} W_{11} & W_{12} & W_{13} & . & . & . & W_{1M} \\ . & . & . & . & . & . & . \\ . & . & . & W_{kj} & . & . & . \\ . & . & . & . & . & . & . \\ W_{N1} & W_{N2} & W_{N3} & . & . & . & W_{NM} \end{bmatrix}$$

$$W_{H_3} = \begin{bmatrix} W_{11} & W_{12} & W_{13} & . & . & . & W_{1N} \\ . & . & . & . & . & . & . \\ . & . & . & W_{zk} & . & . & . \\ . & . & . & . & . & . & . \\ W_{Z1} & W_{Z2} & W_{Z3} & . & . & . & W_{ZN} \end{bmatrix}$$

Let $O^4 = \{O_z\}_{z=1}^{Z}$, $O^3 = \{O_k\}_{k=1}^{N}$ and $O^2 = \{O_j\}_{j=1}^{M}$ be the output produced at each node in layers 4th, 3rd and 2nd respectively when an input $X = \{X_i\}_{i=1}^{D}$ is fed to the network. The output at layer 1 is same as the input and hence $O^1 = \{O_i\}_{i=1}^{D} = \{X_i\}_{i=1}^{D}$. Also let $T = \{T_z\}_{z=1}^{Z}$ is the target or desired output at each node in the output layer. Now the error vector at layer l, Δ_l can be computed by error backpropagation.

$$\begin{aligned} \Delta_4 &= \left[O^4 - T\right]_{[z\times 1]} && \text{(A.15)} \\ \Delta_3 &= \left[\mathbf{W}_{H_3}^{\mathbf{T}}\right]_{N\times Z} * \left[O^4 . * (1 - O^4) . * \Delta_4\right]_{Z\times 1} && \text{(A.16)} \\ \Delta_2 &= \left[\mathbf{W}_{H_2}^{\mathbf{T}}\right]_{M\times N} * \left[O^3 . * (1 - O^3) . * \Delta_3\right]_{N\times 1} && \text{(A.17)} \end{aligned}$$

Note that Eq. A.16 and Eq. A.17 follows from the Eq. A.6.

Now the gradient at each neuron with respect to the parameter to be updated can be found and the final update rule in matrix form will be

$$\begin{aligned} \mathbf{W}_{H_3}^{new} &= \mathbf{W}_{H_3}^{old} - \eta \left[O^4 . * (1 - O^4) . * \Delta_4\right]_{Z\times 1} * \left[O^3\right]_{1\times N}^{\mathbf{T}} && \text{(A.18)} \\ \mathbf{W}_{H_2}^{new} &= \mathbf{W}_{H_2}^{old} - \eta \left[O^3 . * (1 - O^3) . * \Delta_3\right]_{N\times 1} * \left[O^2\right]_{1\times M}^{\mathbf{T}} && \text{(A.19)} \\ \mathbf{W}_{H_1}^{new} &= \mathbf{W}_{H_1}^{old} - \eta \left[O^2 . * (1 - O^2) . * \Delta_2\right]_{M\times 1} * \left[O^1\right]_{1\times D}^{\mathbf{T}} && \text{(A.20)} \end{aligned}$$

Thus to update the weight (W_{kj}) between neuron j in layer $L-2$ and neuron k in layer $L-1$, the gradient of the error function at the final layer with respect to W_{kj} need to be computed. It can be seen from the above set of equations that this gradient can be interpreted as the product of two terms. The first term is the error computed at node k in layer $L-1$ multiplied by the derivative of the sigmoid activation function with respect to the output at node k. This can be treated as

the error at the rear end of the connection holding the weight W_{kj}. The second term is the output at the node j in layer $L-2$, and can be considered as the input to the connection holding the weight W_{kj}. Thus, the change for updating the weight for a connection between any layers can be simply obtained by computing the product of error at rear end of the connection and input at front end of the connection. This is then weighted by the learning parameter η and the weight of the connection is updated by gradient descent.

A.3 Training CNN using backpropagation

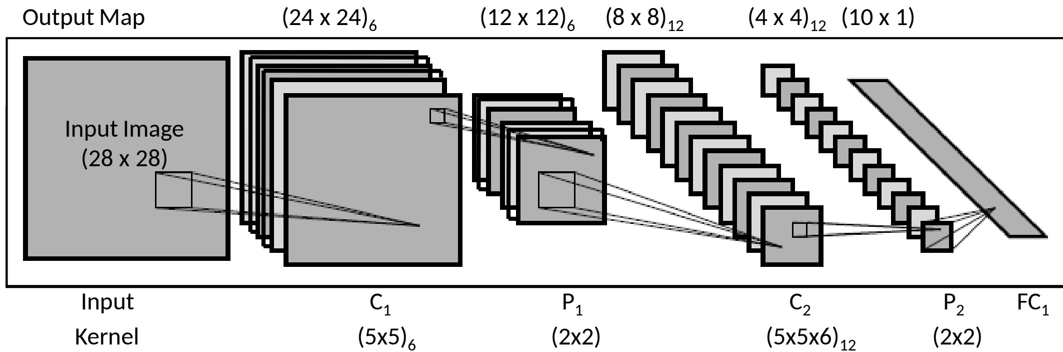

Figure A.3: LeNet architecture for digit recognition

The same backpropagation algorithm that we had discussed in last section is used to train convolutional neural networks as well. In this section, we will discuss this procedure taking the popular LeNet CNN architecture as an example. The LeNet architecture for digit recognition is shown in Fig. A.3. It has two convolution layers (C_1, C_2), two average pooling layers (P_1, P_2) and one fully connected layer (FC_1). There are 6 kernels in C_1 each of size 5×5 and 12 kernels in C_2 each of size $5 \times 5 \times 6$. The average pooling does a 2×2 pooling (thus the kernel weights are fixed as 0.25). The architecture was originally designed for digit recognition for input images of dimension 28×28. The corresponding output maps generated at each layer is shown above the individual blocks. Note that the output dimension is 10×1 since each input has to be mapped to one of the 10 digits. Note that the parameters to be learned in this CNN are 1) the weights of the kernels and their biases at the convolutional layers ($5 \times 5 \times 6 + 6 = 156$ in C_1

Table A.1: Parameters for Weight Initialisation for CNN in A.3

	C_1	C_2	FC_1
Fan_{in}	$1 \times (5 \times 5) = 25$	$6 \times (5 \times 5) = 150$	$12 \times (4 \times 4) = 192$
Fan_{out}	$6 \times (5 \times 5) = 150$	$12 \times (5 \times 5) = 300$	10
ζ	0.0756	0.0471	0.0704

and $5 \times 5 \times 6 \times 12 + 12 = 1812$ in C_2) and 2) the weights and biases at the fully connected layer ($4 \times 4 \times 12 \times 10 + 10 = 1930$).

A.3.1 Parameter Initialization

The parameters for each layer are initialised based on the number of input and output connections at that layer. Specifically these are initialised with random numbers selected from uniform distribution between $U[-\zeta, \zeta]$, where the bound ζ is determined by the fan_{in} and fan_{out} of the layer.

$$\zeta = \sqrt{\frac{1}{fan_{in} + fan_{out}}} \tag{A.21}$$

Here, for convolutional layers fan_{in} is defined as the product of number of input maps and kernel size and fan_{out} is defined as the number of output maps and kernel size. For fully connected layer, fan_{in} is the product of number of input maps and size of the input map while fan_{out} is the number output nodes. Table A.1 shows the parameters for weight initialisation for the CNN architecture shown in Fig. A.3.

A.3.2 Forward Propagation

The respective operations are performed on the input maps at each layer. At convolutional layers, each kernel is used to convolve with the input map, then kernel bias is used to offset the result, and then the sigmoid activation function is applied to produce an output map. Note that as discussed in section 6.4, only

valid part of result of convolution is used to generate the output map.

$$O\left[i\right] = \frac{1}{1 + \exp^{-\left(\sum_{d=1}^{D}[I(:,:,d)*K_i(:,:,d)+B(i)]\right)}} \tag{A.22}$$

In Eq. A.22, $*$ represents the convolution operation and $B(i)$ represents the bias of ith kernel. $O[i]$ is the output map generated for i^{th} kernel when applied on the dth input map. For convolutional layer 1, $D = 1$ and i varies from 1 to 6. For convolutional layer 2, $D = 6$ and i varies from 1 to 12. At the pooling layers, each 2×2 block is averaged to form a single pixel thereby reducing the output map size by 2 along each dimension. At the fully connected layer, the output is computed just like normal feed forward neural network as defined by the set of equations Eq. A.3 and Eq. A.2.

A.3.3 Backward Error Propagation

We use the same loss function defined in Eq. A.1. For any input image, we compute the result at the output layer which will be the 10×1 vector, and now we can compute $\frac{\partial E}{\partial O_k}$ using Eq. A.4. Now this error has to be backpropagated across different layers.

Error backpropagation across fully connected layer FC_1

The derivative of the loss function with respect to the output at any node k in the final layer $\frac{\partial E}{\partial O_k}$ is $O_k - T_k$. Thus the error in fully connected layer FC_1 is a 10×1 vector holding $(O_k - T_k)_{k=1}^{10}$.

$$\Delta E_{FC_1} = O - T \tag{A.23}$$

The error at nodes in the FC_1 has to be backpropagated to the pooling layer 2 (P_2). As per the Eq. A.9, this turns out to be ΔE_{S_2}

$$\Delta E_{P_2} = W. * O. * (1 - O). * \Delta E_{FC_1} \tag{A.24}$$

Here in Eq. A.24, '.*' represents the element by element multiplication. Here W is the weight matrix connecting 192×1 output map at P_2 to the 10×1 output vector. The 192×1 output map at P_2 is actually the vectorised representation of 12 maps each of size 4×4. Thus ΔE_{S_2} is of dimension $([192, 10] \times [10, 1] \rightarrow [192, 1])$. This is reshaped into $(4 \times 4)_{12}$ for representational convenience in propagating the error further down the layers.

Error backpropagation across pooling layer P_2

Since the pooling operation does an average pooling in 2×2 neighbourhood, the error backpropagation across this layer is just an up-sampling by 2.

$$\Delta E_{C_2} = US_2(\Delta E_{P_2}) \tag{A.25}$$

Where US_2 represents the operation, upsampling by 2. This leads to ΔE_{C_2} of size $(8 \times 8)_{12}$.

Error backpropagation across convolutional layer C_2

The error backpropagation across the convolution layer is also governed by the same equation Eq. A.9. But by the special construction of the convolution operation, the weights are related in a special way while propagating the error. It turns out to be the correlation of the kernel with the error to be backpropagated after weighted by the derivative of the activation function. First, we will show the correspondence between the correlation operation and the weighted error propagation by taking an example. Then we will give explicit equation for error backpropgation across C_2. Consider the following convolution operation (only valid part of convolution is considered) in Eq. A.26 and the correlation operation in Eq. A.27 (Note that there is adequate padding by zero to reconstruct the dimension of the input).

$$\begin{bmatrix} x_1 & x_2 & x_3 & x_4 \\ x_5 & x_6 & x_7 & x_8 \\ x_9 & x_{10} & x_{11} & x_{12} \\ x_{13} & x_{14} & x_{15} & x_{16} \end{bmatrix} * \begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{bmatrix} = \begin{bmatrix} y_1 & y_2 \\ y_3 & y_4 \end{bmatrix} \tag{A.26}$$

Here

$$\begin{aligned}
y_1 &= w_9x_1 + w_8x_2 + w_7x_3 + w_6x_5 + w_5x_6 + w_4x_7 + w_3x_9 + w_2x_{10} + w_1x_{11} \\
y_2 &= w_9x_2 + w_8x_3 + w_7x_4 + w_6x_6 + w_5x_7 + w_4x_8 + w_3x_{10} + w_2x_{11} + w_1x_{12} \\
y_3 &= w_9x_5 + w_8x_6 + w_7x_7 + w_6x_9 + w_5x_{10} + w_4x_{11} + w_3x_{13} + w_2x_{14} + w_1x_{15} \\
y_4 &= w_9x_6 + w_8x_7 + w_7x_8 + w_6x_{10} + w_5x_{11} + w_4x_{12} + w_3x_{14} + w_2x_{15} + w_1x_{16}
\end{aligned}$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \Delta y_1 & \Delta y_2 & 0 & 0 \\ 0 & 0 & \Delta y_3 & \Delta y_4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \odot \begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{bmatrix} = \begin{bmatrix} \Delta x_1 & \Delta x_2 & \Delta x_3 & \Delta x_4 \\ \Delta x_5 & \Delta x_6 & \Delta x_7 & \Delta x_8 \\ \Delta x_9 & \Delta x_{10} & \Delta x_{11} & \Delta x_{12} \\ \Delta x_{13} & \Delta x_{14} & \Delta x_{15} & \Delta x_{16} \end{bmatrix} \tag{A.27}$$

Here

$$\begin{aligned}
\Delta x_1 &= w_9\Delta y_1; & \Delta x_2 &= w_9\Delta y_2 + w_8\Delta y_1; & \Delta x_3 &= w_8\Delta y_2 + w_7\Delta y_1; \\
\Delta x_4 &= w_7\Delta y_2; & \Delta x_5 &= w_9\Delta y_3 + w_6\Delta y_1; & \Delta x_8 &= w_7\Delta y_4 + w_4\Delta y_2;
\end{aligned}$$

$$\begin{aligned}
\Delta x_{13} &= w_3\Delta y_3; & \Delta x_9 &= w_6\Delta y_3 + w_3\Delta y_1; & \Delta x_{12} &= w_4\Delta y_4 + w_1\Delta y_2; \\
\Delta x_{16} &= w_1\Delta y_4; & \Delta x_{14} &= w_3\Delta y_4 + w_2\Delta y_3 & \Delta x_{15} &= w_2\Delta y_4 + w_1\Delta y_3;
\end{aligned}$$

$$\begin{aligned}
\Delta x_6 &= w_9\Delta y_4 + w_8\Delta y_3 + w_6\Delta y_2 + w_5\Delta y_1; & \Delta x_7 &= w_8\Delta y_4 + w_7\Delta y_3 + w_5\Delta y_2 + w_4\Delta y_1; \\
\Delta x_{10} &= w_6\Delta y_4 + w_3\Delta y_2 + w_5\Delta y_3 + w_2\Delta y_1; & \Delta x_{11} &= w_5\Delta y_4 + w_2\Delta y_2 + w_4\Delta y_3 + w_1\Delta y_1;
\end{aligned}$$

By analysing the above equations, it can be seen that the relationship between the weights and the output during convolution is reproduced during correlation and

hence can be used in backpropagating the error. For example, input x_6 at layer L influences all the output neurons (y_1, y_2, y_3, y_4) at layer $L+1$ during convolution operation through weights w_5, w_6, w_8 and w_9 respectively. Therefore, when back-propagating the error Δy computed at layer $L+1$, the error contribution at node corresponding to x_6 should be the aggregate sum of the error at $\Delta y_1, \Delta y_2, \Delta y_3$ and Δy_4 weighted exactly by the same weights w_5, w_6, w_8 and w_9. By analysing the expression for Δx_6 obtained after the correlation operation, it can be seen that this relationship is preserved. Thus back propagating the error across the convolution layers is equivalent to performing the correlation operation on the weighted error computed for the layer after weighting the derivative of the activation function. Thus ΔE_{P_1} can be computed as

$$\Delta EF_{C_2} = \Delta E_{C_2} .* O_{C_2} .* (1 - O_{C_2}) \quad (A.28)$$

$$\Delta E_{P_1}(:,:,i) = \sum_{l=1}^{L} [\Delta EF_{C_2}(:,:,l) \odot K_l(:,:,i)] \quad (A.29)$$

Where $\odot$ represent the correlation operation, i varies from 1 to 6 and $L = 12$. This will result in backpropagated error dimension as $(12 \times 12)_6$.

Error backpropagation across pooling layer P_1

As discussed earlier, the back propagated error is just an up-sampled version.

$$\Delta E_{C_1} = US_2(\Delta E_{P_1}) \quad (A.30)$$

A.3.4 Learning the Parameters by Gradient Descend

Once the error is propagated for all the neurons, the gradient is computed with respect to the parameter to be updated. The same set of equations provided in Eq. A.11 is used to compute the gradient. The parameters are then updated in the negative direction of the gradient to minimise the loss function.

Derivative of gradient with respect to weights at fully connected layer

$$(\Delta W)_{FC_1} = \Delta E_{FC_1} .* O .* (1 - O) .* I_{FC_1} \quad (A.31)$$

In Eq. A.31, I_{FC_1} represent the vectorized output map at the pooling layer 2 (P_2). Thus the dimension of the gradient is $[10, 1] \times [1, 192] \rightarrow [10, 192]$

Derivative of gradient with respect to kernel weights at convolution layers

$$\Delta EF_{C_l} = \Delta E_{C_l} .* O_{C_l} .* (1 - O_{C_l}) \tag{A.32}$$

$$(\Delta K)_l = \Delta EF_{C_l} * I \tag{A.33}$$

In Eq. A.33, '*' is the convolution operation. $l = 1$ for convolution layer 1 and $l = 2$ for convolution layer 2. I is the corresponding input map to the layer. Since we are taking only valid part of convolution, for C_1, the gradient dimension will be $[5, 5]_6$ ($[24, 24]_6 * [28, 28] \rightarrow [5, 5]_6$). Similarly for convolution layer 2, the gradient dimension will be $[5, 5, 6]_{12}$ ($[8, 8]_{12} * [12, 12]_6 \rightarrow [5, 5, 6]_{12}$)

Updating the kernel weights

Once the gradient is determined with respect to each parameter, we can use gradient descent so as to minimise the loss function.

$$K^{new} = K^{old} - \eta \Delta K \tag{A.34}$$

The gradient for the bias term for any node in the output layer, turns out to be the cumulative weight change computed for all the connections to that neuron. Similarly, the gradient for the bias of the kernels turns out to be the cumulative weight change computed for the weights in the respective kernel.