

Convolutional Neural Networks

- Convolution with Vector Input and Output
- Single and Multi-Layer CNNs
- Upsampling, Pooling, and Stride

Limitations of Dense Neural Networks

- Too many parameters:
 - Number of parameters scale roughly as N_x^2 for N -dimensional inputs.
 - Processing a 1024×1024 image, requires $\sim 10^{12}$ parameters per layer.
 - Both training and inference is slow.
- The solution to many problems should be space-invariant:
 - Object recognition: Recognition shouldn't depend on object position in image.
 - Image denoising: Noise removal should depend on content, not absolute location in the image.

What is Convolution?

- Discrete-time convolution is defined as*

$$\begin{aligned}x(j) &= z(j) * w(j) = \sum_{k=-\infty}^{\infty} z(j-k) w(k) \\ &= w(j) * z(j) = \sum_{k=-\infty}^{\infty} w(j-k) z(k)\end{aligned}$$

In 2D, convolution is given by

$$\begin{aligned}x(j_1, j_2) &= w(j_1, j_2) * z(j_1, j_2) = z(j_1, j_2) * w(j_1, j_2) \\ &= \sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} z(j_1 - k_1, j_2 - k_2) w(k_1, k_2)\end{aligned}$$

*CNN sometimes use a non-standard definition of convolution, but you can just time reverse w to convert. However, the non-standard definition leads to issues. So, we will adopt the standard definition.

“Convolution” in DNN Software

- DNN software implements a non-standard “convolution”.
- Non-standard DNN convolution is actually correlation:

$$\tilde{x}(j) = w(j) \underset{\text{DNN}}{*} z(j) = \sum_{k=-\infty}^{\infty} w(k-j) z(k) = w(-j) * z(j)$$

- The problem is that this “convolution” is not commutative:

$$\begin{aligned} \tilde{x}(j) &= w(j) \underset{\text{DNN}}{*} z(j) = \sum_{k=-\infty}^{\infty} w(k-j) z(k) \\ &= \sum_{k=-\infty}^{\infty} w(k) z(k+j) = z(-j) \underset{\text{DNN}}{*} w(j) = \tilde{x}(-j) \end{aligned}$$

- We will adopt the standard definition of convolution for our analysis, but remember that the software is different.

Convolution with Finite Kernels

- Discrete-time convolution is defined as

$$x(j) = w(j) * z(j) = \sum_{k=-p}^p z(j - k) w(k)$$

where w is of length $2p + 1$

- In 2D, convolution is given by

$$x(j_1, j_2) = \sum_{k_1=-p}^p \sum_{k_2=-p}^p z(j_1 - k_1, j_2 - k_2) w(k_1, k_2)$$

where w is of size $(2p + 1)(2p + 1)$

2D Convolution with "Same" boundary

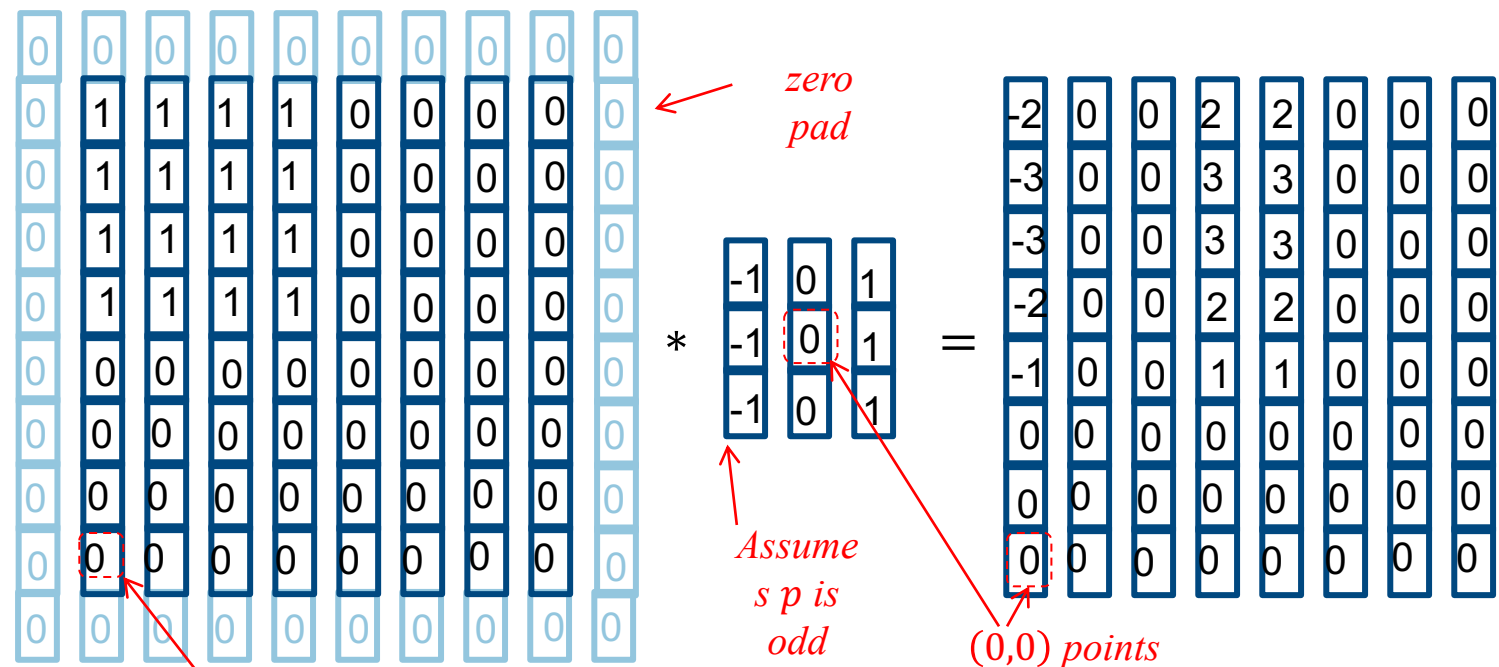
- For 3x3 filter is given by

$$x(j_1, j_2) = \sum_{k_1=-1}^1 \sum_{k_2=-1}^1 z(j_1 - k_1, j_2 - k_2) w(k_1, k_2)$$

Tensor notation

$$x^{j_1, j_2} = w_{(j_1, j_2)} * z^{(j_1, j_2)}$$

Parenthesizes denote convolution

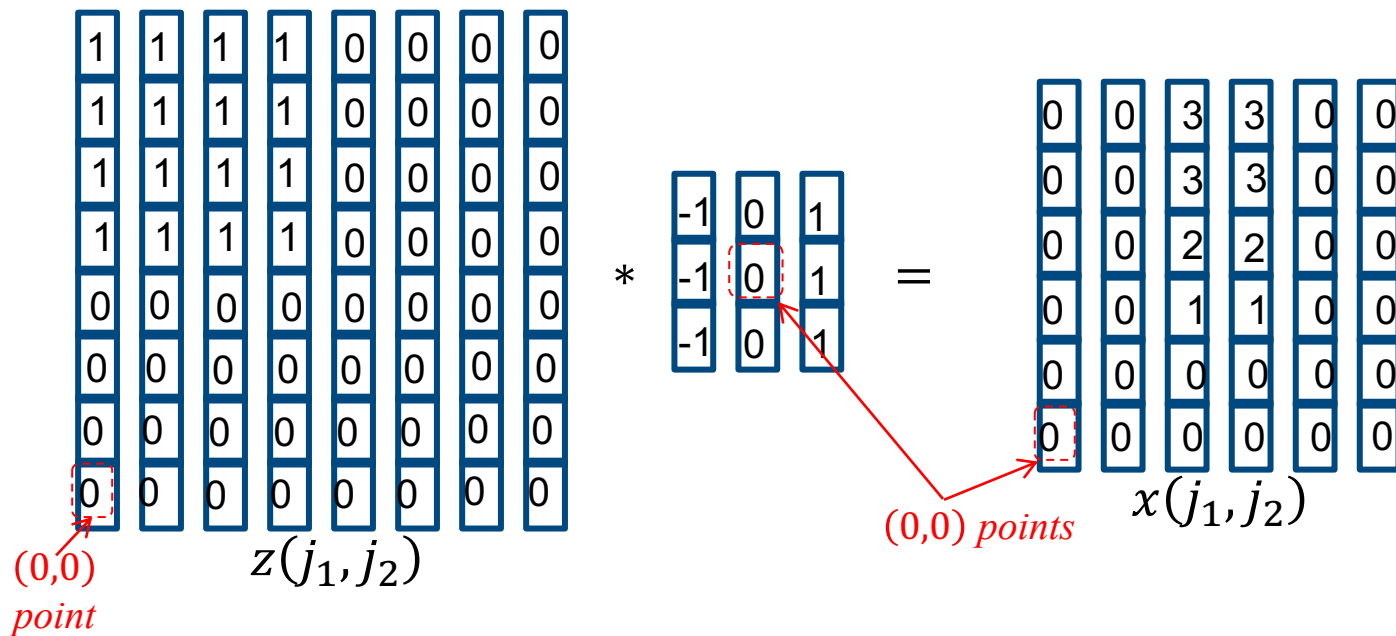


2D Convolution with “Valid” Boundary

- For 3×3 filter is given by

$$x(j_1, j_2) = \sum_{k_1=-1}^1 \sum_{k_2=-1}^1 z(j_1 + 1 - k_1, j_2 + 1 - k_2) w(k_1, k_2)$$

Tensor notation $x^{j_1, j_2} = w_{(j_1, j_2)} * z^{(j_1, j_2)}$

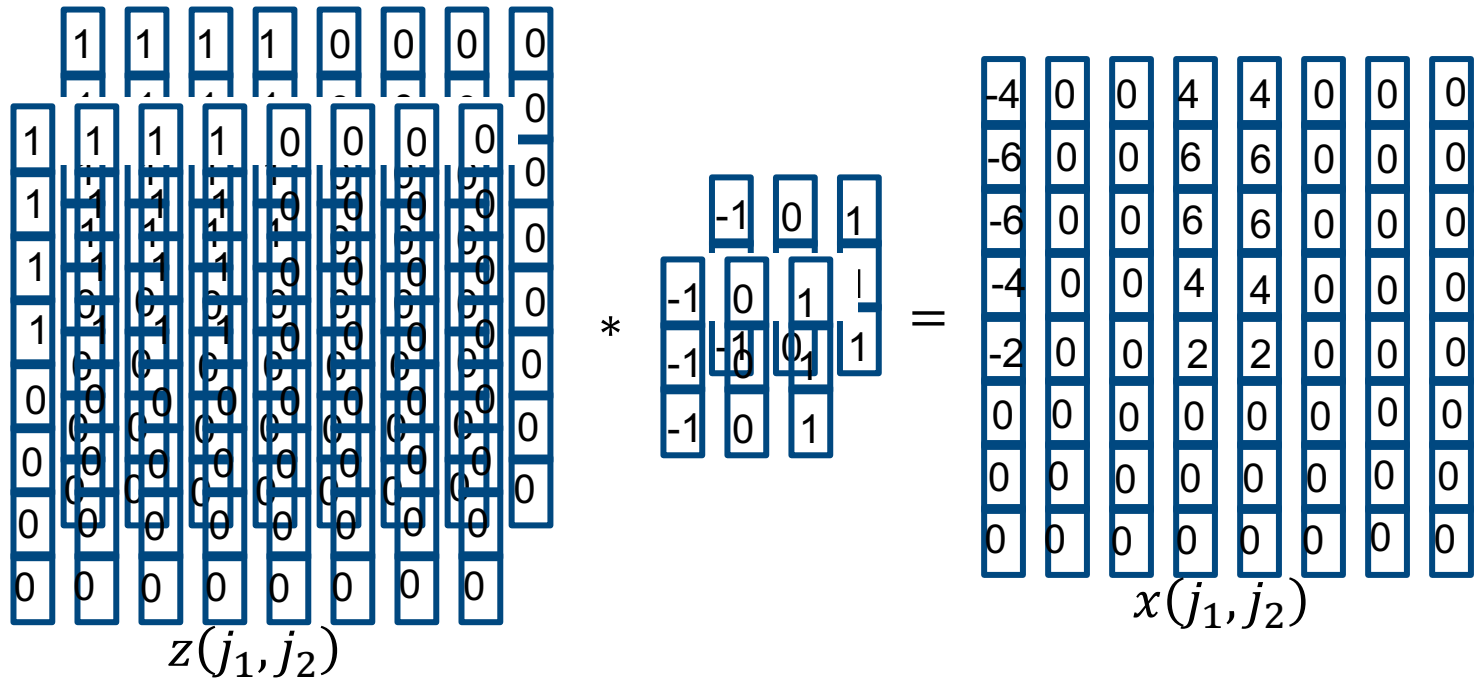


2D Convolution with Vector Input

- Vector 2D convolution of depth $d_i = 2$ and $d_o = 1$

$$x(j_1, j_2) = \sum_{i=0}^{d-1} \sum_{k_1=-p}^p \sum_{k_2=-p}^p z(j_1 - k_1, j_2 - k_2, i) w(k_1, k_2, i)$$

Tensor notation $x^{j_1, j_2} = w_{(j_1, j_2), i} * z^{(j_1, j_2), i}$

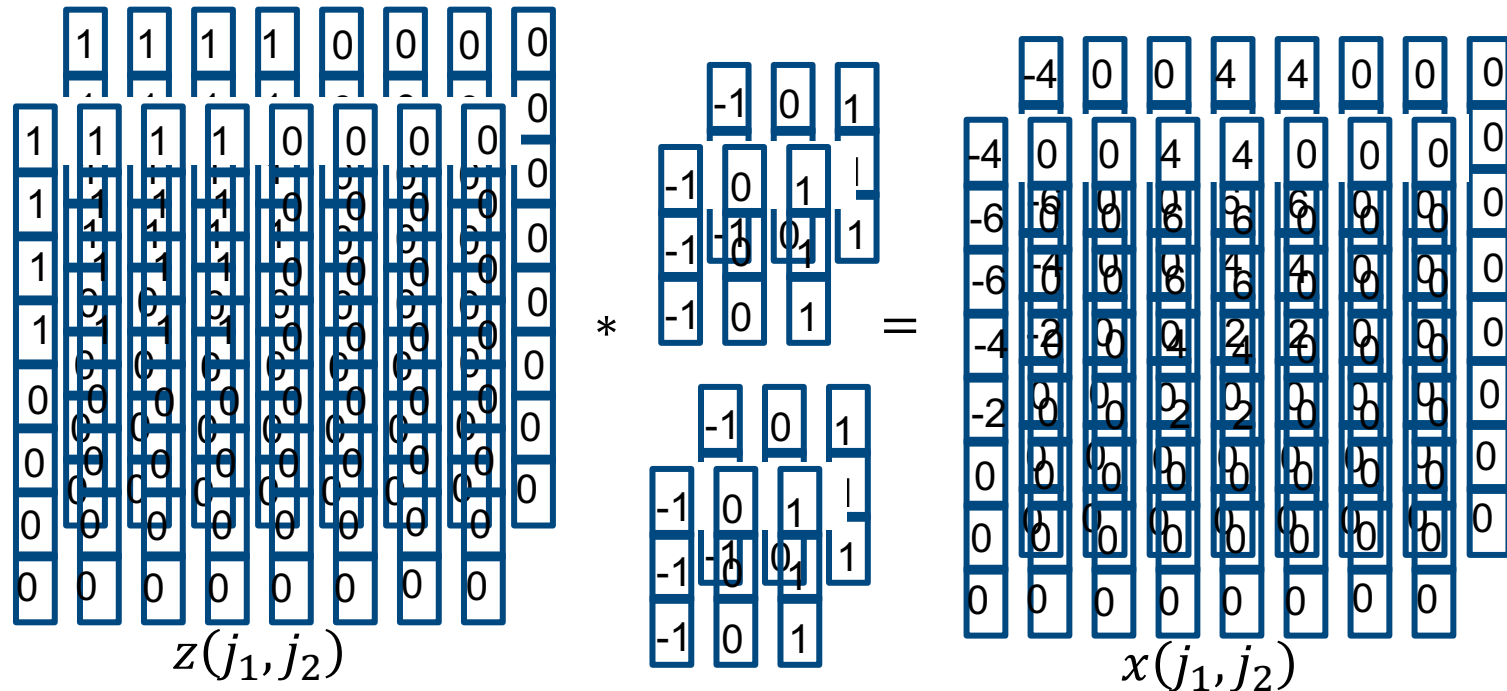


2D Convolution with Vector Input/Output

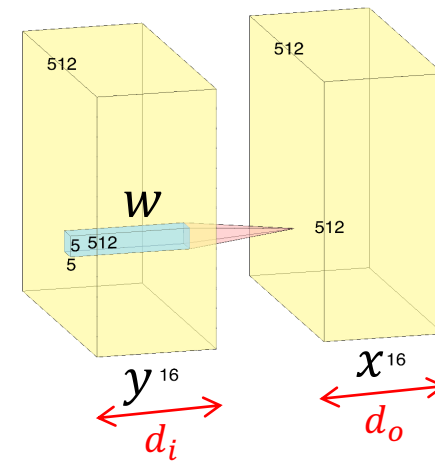
- Vector 2D convolution of depth $d_i = 2$ and $d_o = 2$

$$x^{j_1, j_2, j_3} = W_{(j_1, j_2), i}^{j_3} * z^{(j_1, j_2), i}$$

- Beautiful animations at <https://pathmind.com/wiki/convolutional-network>



Convolution Diagram



- Sum notation

$$x(j_1, j_2, j_3) = \sum_{i=0}^{d_i-1} \sum_{k_1=-p}^p \sum_{k_2=-p}^p y(j_1 - k_1, j_2 - k_2, i) w(k_1, k_2, i, j_3)$$

- Tensor notation

$$x^{j_1, j_2, j_3} = w_{(j_1, j_2), i}^{j_3} * y^{(j_1, j_2), i}$$

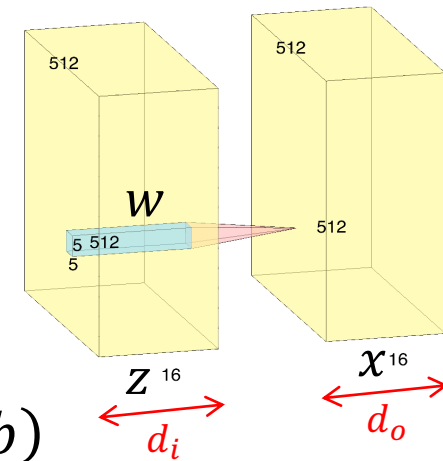
where

w is $5 \times 5 \times 16 \times 16 = 6,400$

y is $512 \times 512 \times 16 = 4,194,304$

x is $512 \times 512 \times 16 = 4,194,304$

Single Layer CNN



- Single layer 2D CNN example:

$$x = f_{\theta}(z) = \sigma(w * z + b)$$

$$x^{j_1, j_2, j_3} = \sigma(w_{(j_1, j_2), i}^{j_3} * z^{(j_1, j_2), i} + b^{j_3})$$

- Tensor structures

w is $5 \times 5 \times 16 \times 16 = 6,400$

b is $16 = 16$

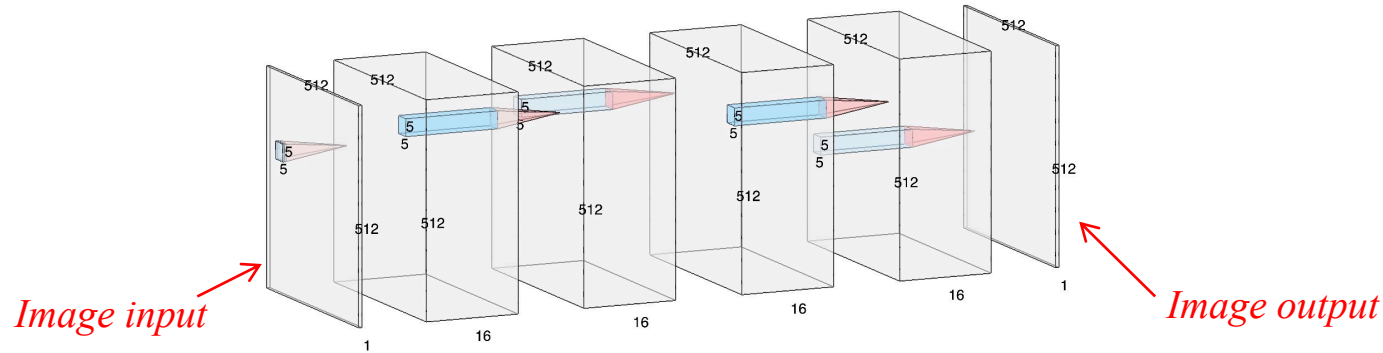
} *Parameter
Tensors*

z is $512 \times 512 \times 16 = 4,194,304$

x is $512 \times 512 \times 16 = 4,194,304$

} *Feature or
State Tensors*

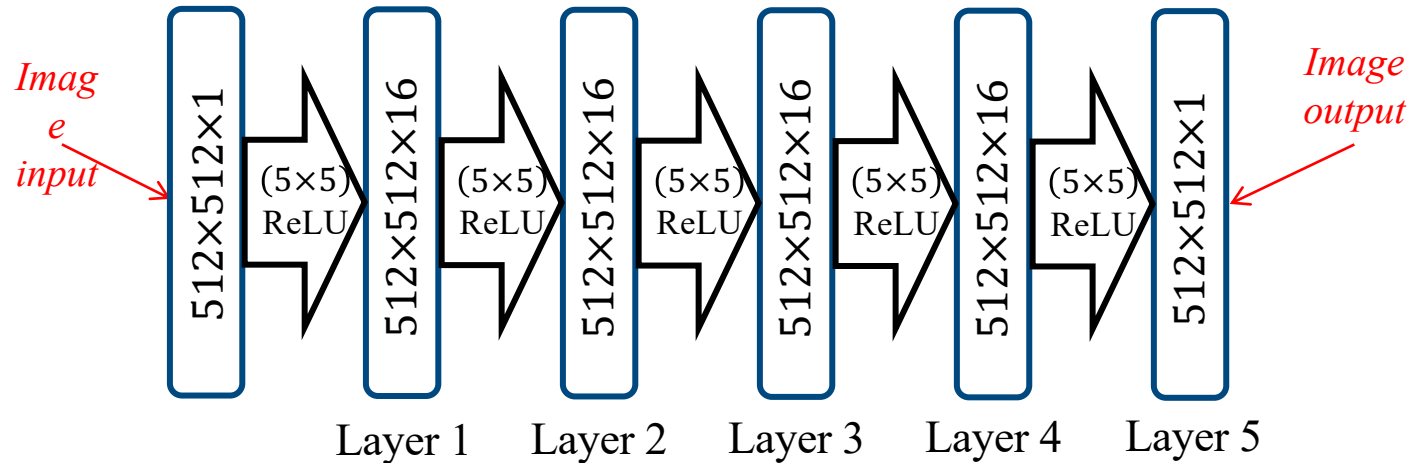
5 Layer CNN



■ Parameters:

- Layer 1: filter $5 \times 5 \times 1 \times 16$; offset 16; ReLU; #params 416
- Layer 2: filter $5 \times 5 \times 16 \times 16$; offset 16; ReLU; #params 6,416
- Layer 3: filter $5 \times 5 \times 16 \times 16$; offset 16; ReLU; #params 6,416
- Layer 4: filter $5 \times 5 \times 16 \times 16$; offset 16; ReLU; #params 6,416
- Layer 5: filter $5 \times 5 \times 16 \times 1$; offset 1; ReLU; #params 401
- Total # params = 20,065

Simpler Diagram for 5 Layer CNN



- Good news:

- Dramatic reduction in number of parameters as compared to dense network
- Spatially invariant behavior (except for boundaries)
- For interior layers, each pixel is formed by a 16-dimensional feature vector
- Intuition: CNN can design its own features!

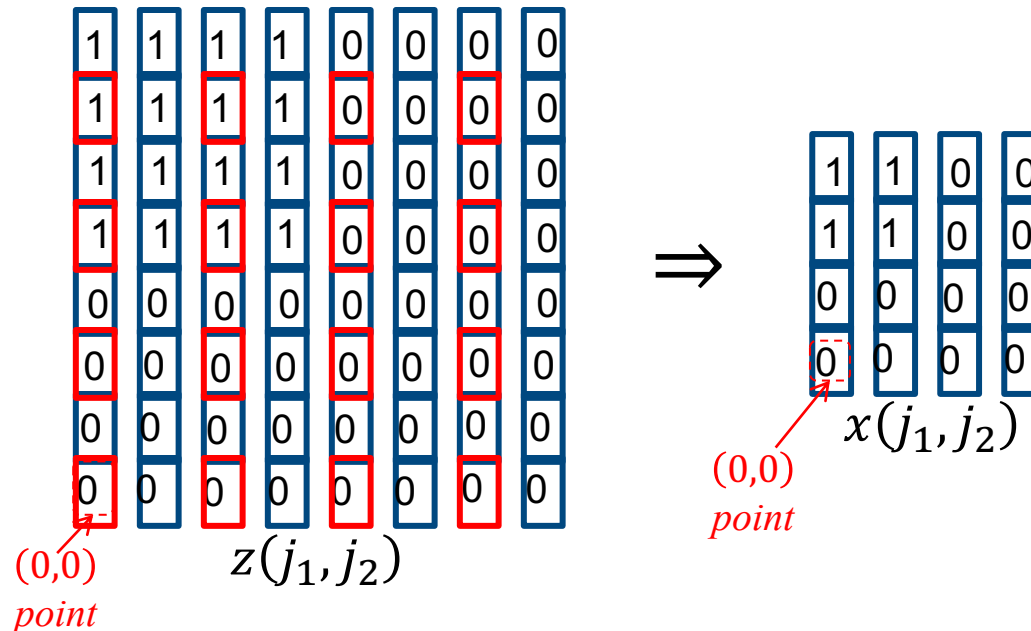
- Bad news:

- Not very useful
- ReLUs are fast but can be difficult to train
- Spatial resolution of each layer is the same
- Output isn't appropriate for classification

CNN Stride

- Stride of $(d_v, d_h) = (2,2)$, results in

$$x(j_1, j_2) = z(2j_1, 2j_2)$$



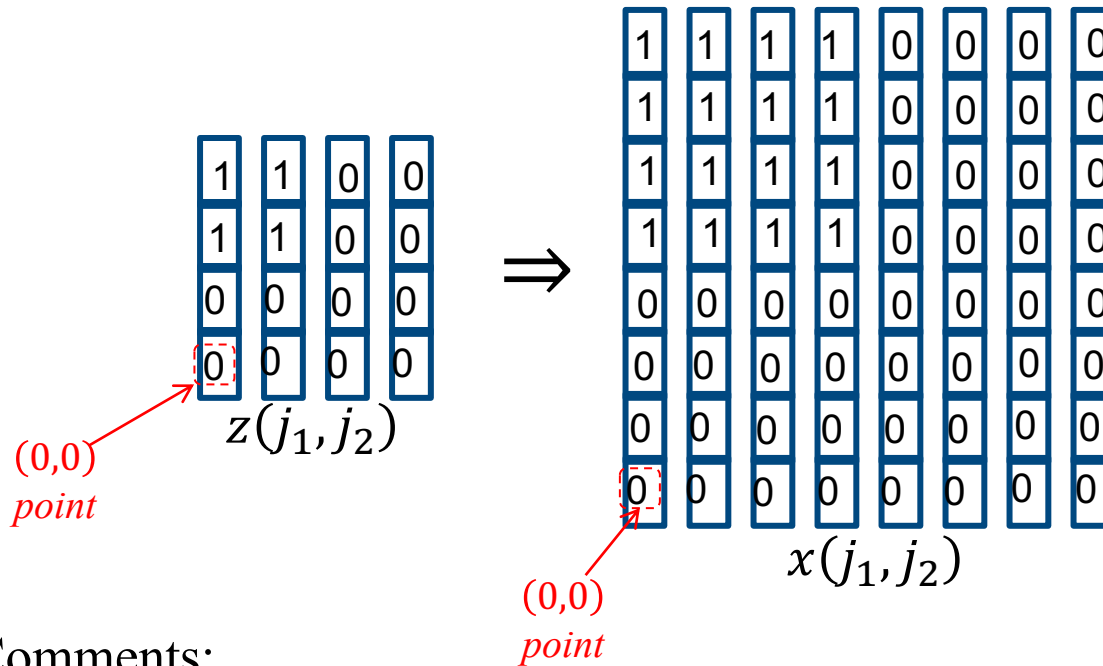
- Comments:

- Output has reduced dimensions of $(\lceil N_v/d_v \rceil, \lceil N_h/d_h \rceil)$
- Also known as decimation
- Can be used with most operators

CNN Upsampling

- Upsampling of $(L_v, L_h) = (2,2)$, results in

$$x(j_1, j_2) = z(\lfloor j_1/2 \rfloor, \lfloor j_2/2 \rfloor)$$



- Comments:

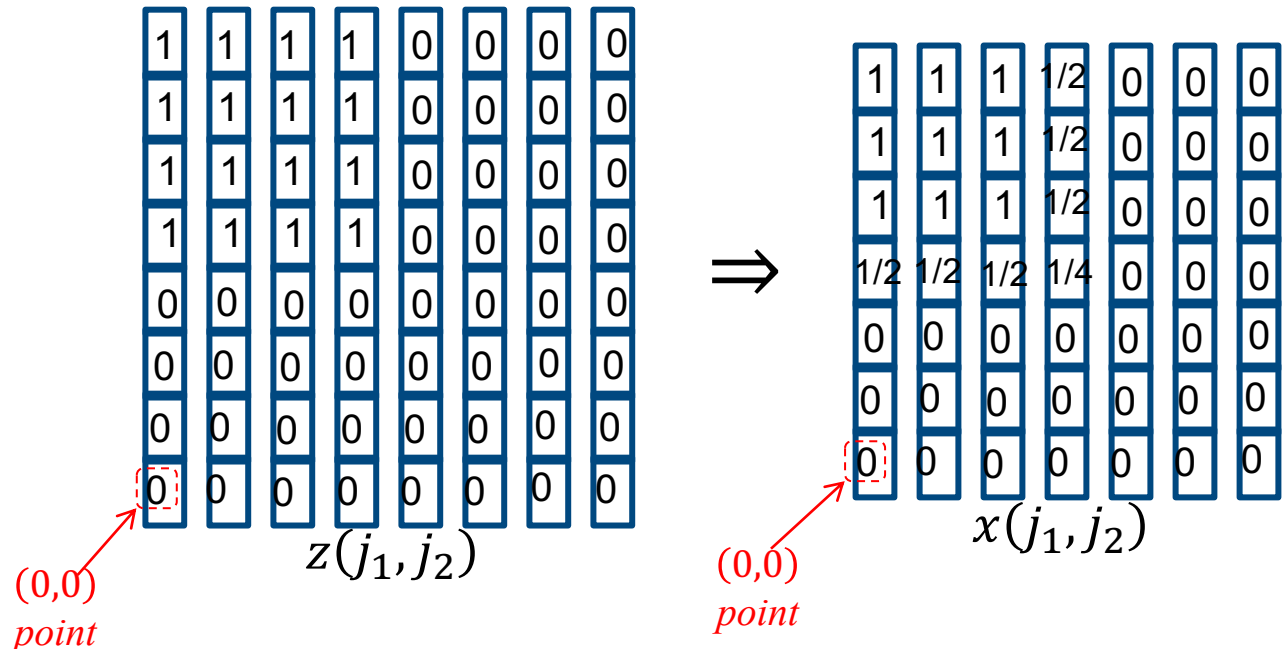
- Also known as pixel replication
- Output has dimensions of $(L_v N_v, L_h N_h)$

CNN Average Pooling

- 2D average pooling equation

$$x(j_1, j_2) = \sum_{k_1=0}^{p-1} \sum_{k_2=0}^{p-1} \frac{1}{p^2} z(j_1 + k_1, j_2 + k_2)$$

For $p = 2$, and stride of (1,1), we have

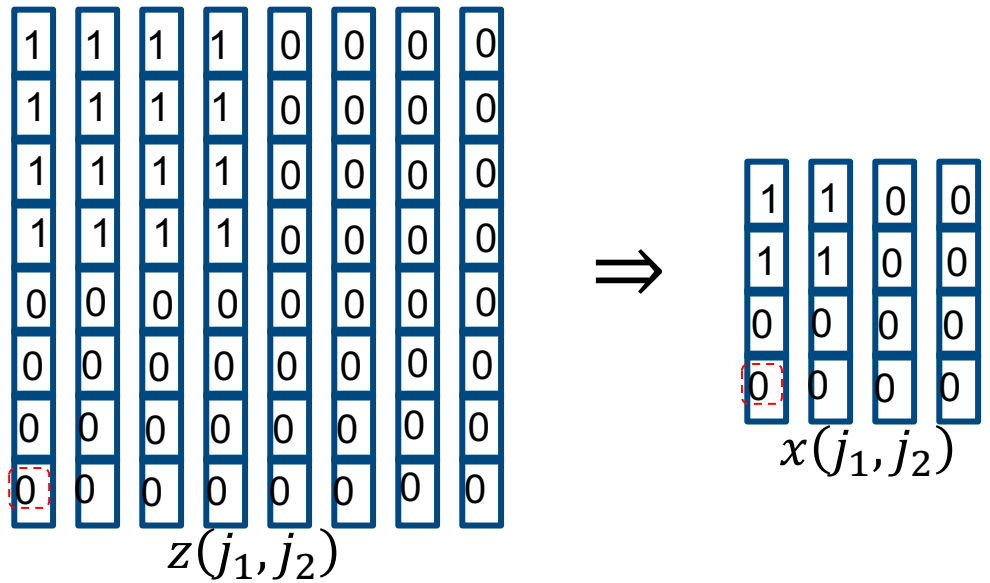


CNN Average Pooling with Stride

- 2D average pooling equation

$$x(j_1, j_2) = \sum_{k_1=0}^{p-1} \sum_{k_2=0}^{p-1} \frac{1}{p^2} z(d_1 j_1 + k_1, d_2 j_2 + k_2)$$

For $p = 2$ with stride of $(d_1, d_2) = (2, 2)$, we have

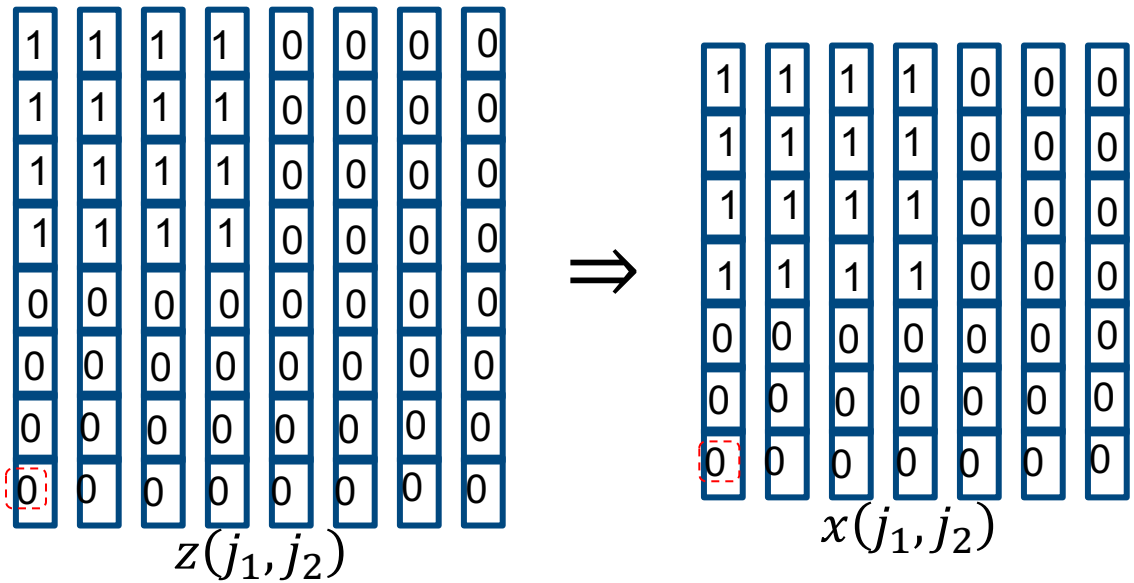


CNN Max Pooling

- 2D max pooling equation

$$x(j_1, j_2) = \max_{0 \leq k_1 < p} \max_{0 \leq k_2 < p} z(j_1 + k_1, j_2 + k_2)$$

For $p = 2$ and stride of $(d_1, d_2) = (1, 1)$, we have

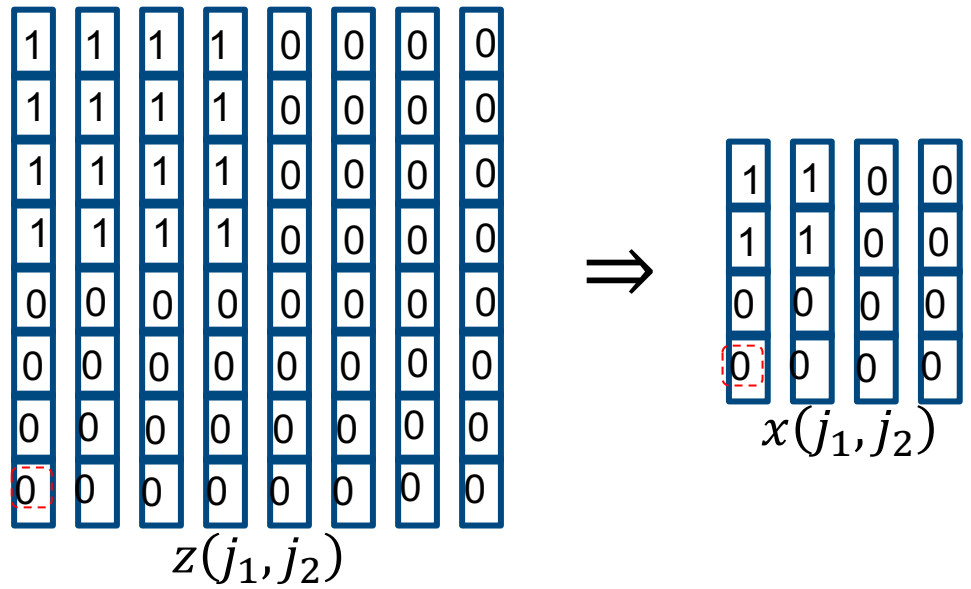


CNN Max Pooling with Stride

- 2D max pooling with a stride equation

$$x(j_1, j_2) = \max_{0 \leq k_1 < p} \max_{0 \leq k_2 < p} z(d_1 j_1 + k_1, d_2 j_2 + k_2)$$

For $p = 2$ and stride of $(d_1, d_2) = (2, 2)$, we have

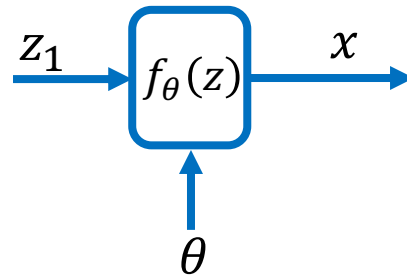


Adjoint Gradients for CNNs

- Implementation of a CNN node
- The fast adjoint gradient
- Computing the adjoint gradient for CNN nodes

Backpropagation for CNN Nodes

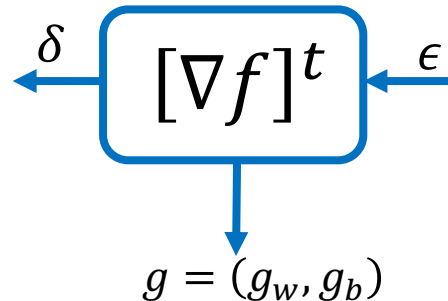
- For each node you need two functions:
 - Forward propagation function:



$$x \leftarrow f(z_1, \theta)$$

You need a software implementation of the forward function.

- Adjoint gradient function:

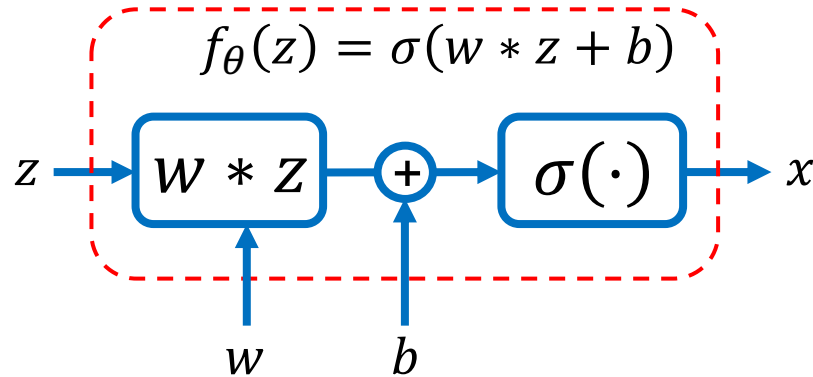


$$[\delta, g_w, g_b] \leftarrow G(\epsilon, z, \theta)$$

You need a software implementation of this function that multiplies ϵ by the adjoint gradient.

Gradient for Single Layer CNN

- Single layer NN:



- We will need the adjoint gradient w.r.t $\theta = (w, b)$

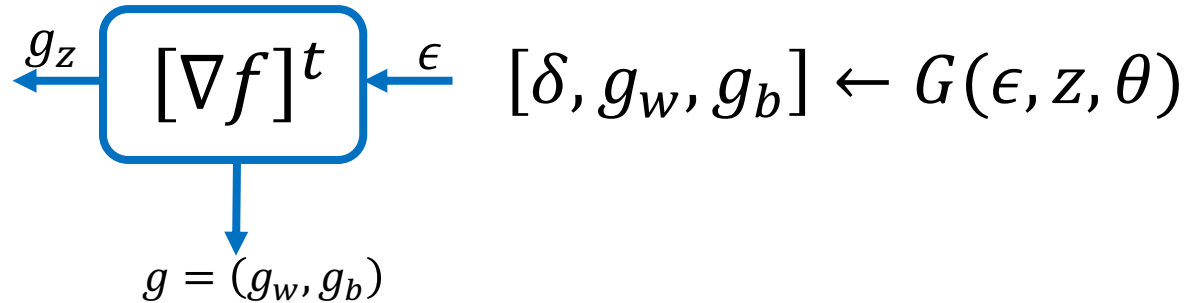
$$\nabla_{\theta} f_{\theta}(z) = [\nabla_w f_{(w,b)}(z), \nabla_b f_{(w,b)}(z)]$$

- And, we will also need:

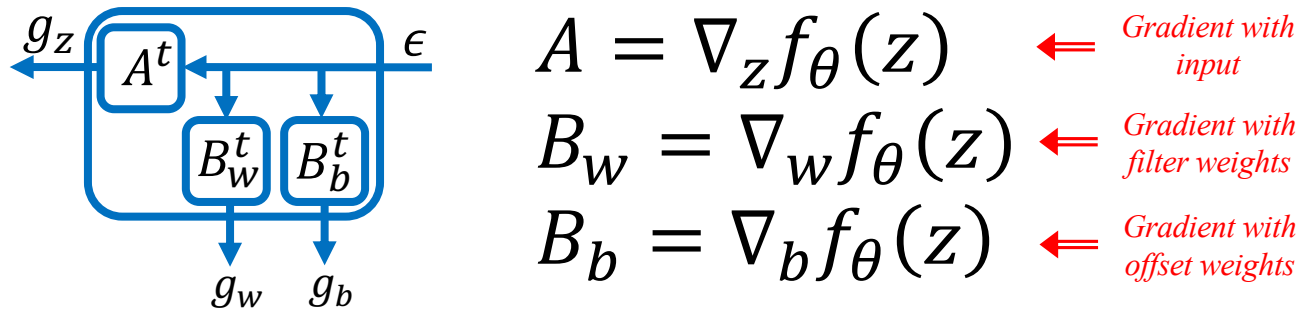
$$\nabla_z f_{\theta}(z)$$

Closer Look at the Adjoint Gradient

- Each node needs a function to compute the adjoint gradient:

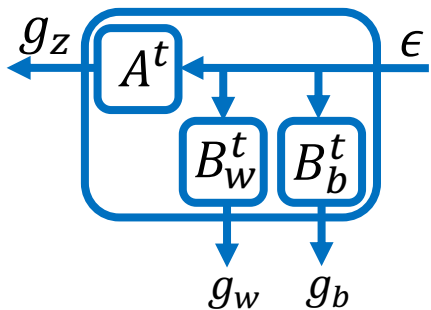


- Conceptually, this is



The Fast Adjoint Gradient

- Conceptually, the adjoint gradient function computes this

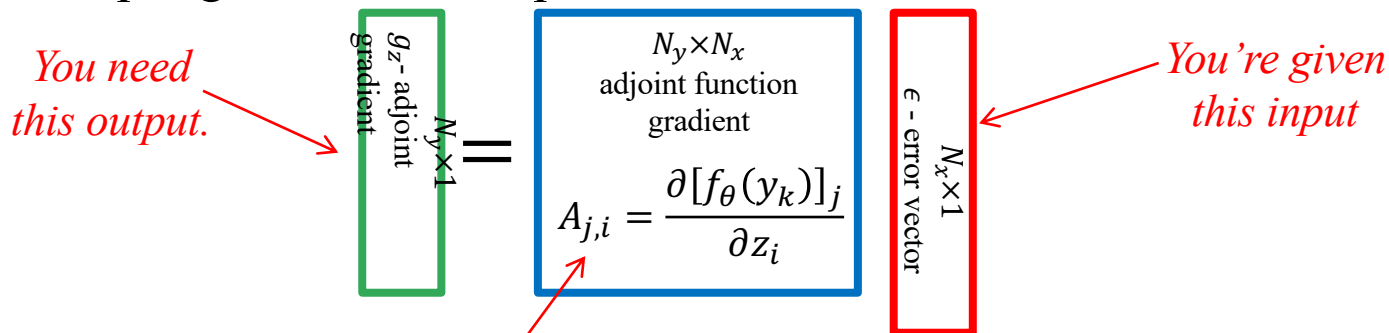


$$A = \nabla_z f_\theta(z)$$

$$B_w = \nabla_w f_\theta(z)$$

$$B_b = \nabla_b f_\theta(z)$$

- For input gradient, it computes



But do you really need to compute this huge matrix? No!

But how??

- Big Idea: Directly compute output without computing gradient!

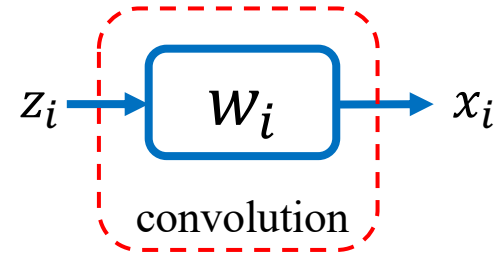
Adjoint Gradient of Convolution w.r.t. Input

- Gradient of output with input:

$$x_i = w_i * z_i = \sum_j w_{i-j} z_j$$

So $x = Az$ where $A_{i,j} = w_{i-j}$

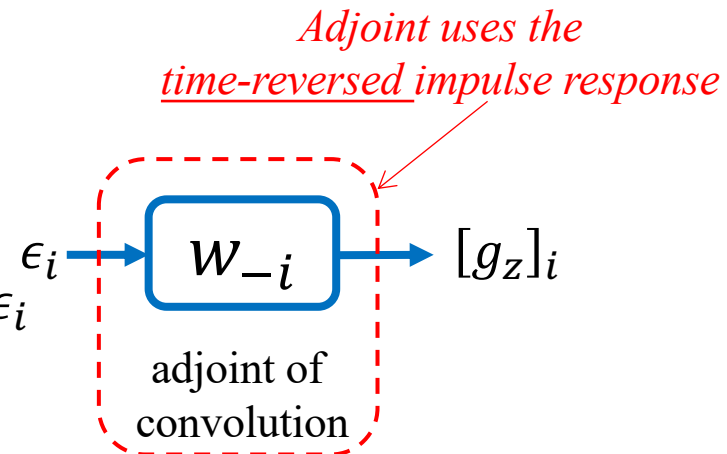
$$\frac{\partial x_i}{\partial z_j} = A_{i,j} = w_{i-j}$$



- Adjoint gradient:

$$[A^t]_{i,j} = A_{j,i} = w_{j-i}$$

$$[g_z]_i = \sum_j w_{j-i} \epsilon_j = w_{-i} * \epsilon_i$$



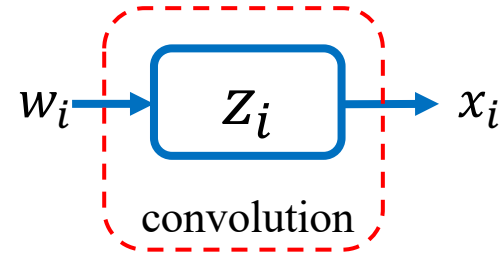
Adjoint Gradient of Convolution w.r.t. Weights

- Gradient of output with weights:

$$x_i = z_i * w_i = \sum_j z_{i-j} w_j$$

$$x = Aw$$

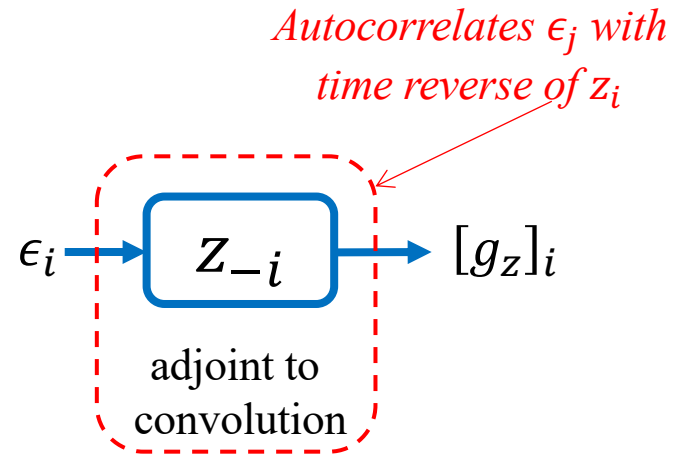
$$\text{where } \frac{\partial x_i}{\partial w_j} = A_{i,j} = z_{i-j}$$



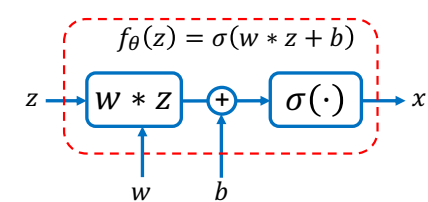
- Adjoint gradient:

$$[A^t]_{i,j} = A_{j,i} = z_{j-i}$$

$$[g_z]_i = \sum_j z_{j-i} \epsilon_j$$



Adjoint Gradient of w.r.t. Input

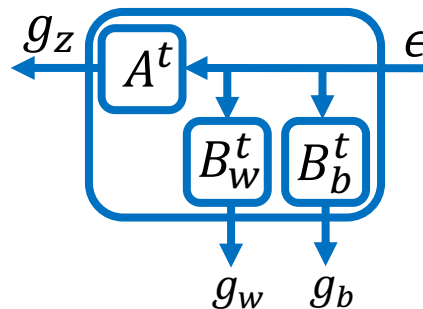


- Forward function:

$$f(y) = \sigma(w_{(j_1, j_2), i}^{j_3} * z^{(j_1, j_2), i} + b^{j_3})$$

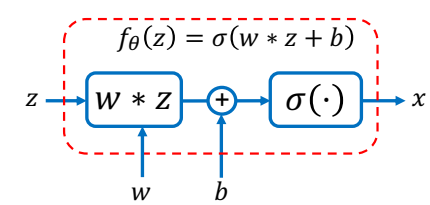
- The adjoint gradient $[\nabla_z f]^t \epsilon$ is given by

$$\delta_{j_1, j_2, i} = w_{(-j_1, -j_2), i}^{j_3} [\nabla \sigma]^{k_1, k_2, k_3}_{(j_1, j_2), j_3} \epsilon_{k_1, k_2, k_3}$$



Fast because it never computes A!

Adjoint Gradient w.r.t. b

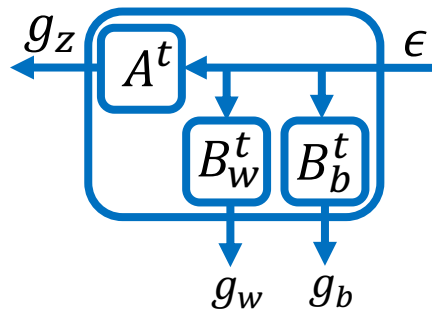


- Forward function:

$$f(y) = \sigma(w_{(j_1, j_2), i}^{j_3} * z^{(j_1, j_2), i} + b^{j_3})$$

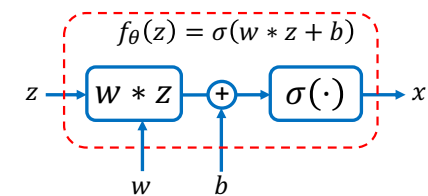
- The adjoint gradient $[\nabla_b f]^t \epsilon$ is given by

$$[g_b]_{j_3} = \mathbf{1}^{j_1, j_2} [\nabla \sigma]^{k_1, k_2, k_3}_{j_1, j_2, j_3} \epsilon_{k_1, k_2, k_3}$$



Note: $\mathbf{1}^{j_1, j_2} = 1$ for all j_1 and j_2

Adjoint Gradient w.r.t. w



- Forward function:

$$f(y) = \sigma(w_{(j_1, j_2), i}^{j_3} * z^{(j_1, j_2), i} + b^{j_3})$$

- The adjoint gradient $[\nabla_w f]^t \epsilon$ is given by

$$[g_w]_{j_1, j_2, i, j_3} = z^{(-j_1, -j_2), i} [\nabla \sigma]^{k_1, k_2, k_3}_{(j_1, j_2), j_3} \epsilon^{k_1, k_2, k_3}$$

