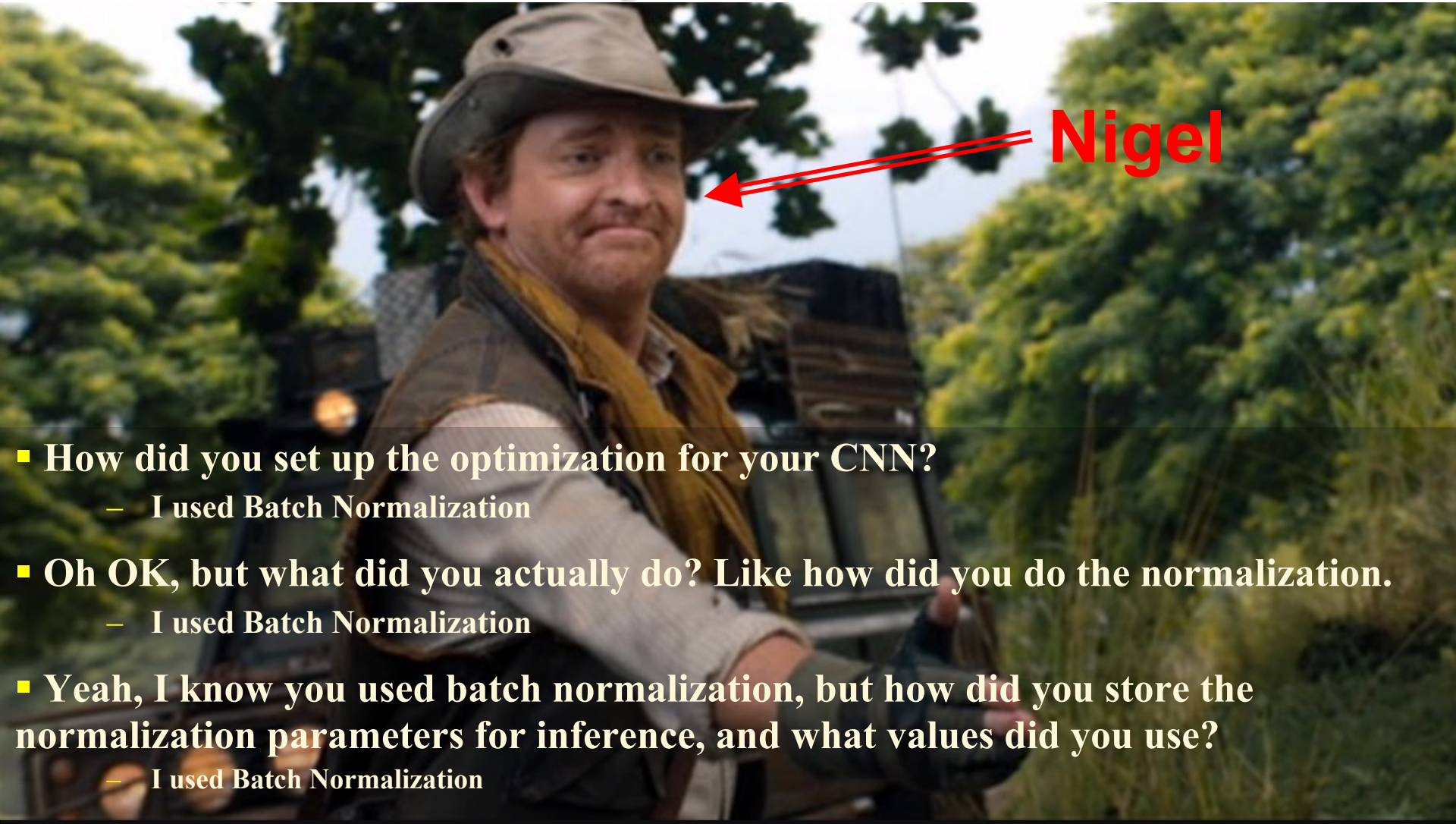
Convolutional Neural Networks

- 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.

Don’t be a Jumanji NPC*



- How did you set up the optimization for your CNN?
  - I used Batch Normalization
- Oh OK, but what did you actually do? Like how did you do the normalization.
  - I used Batch Normalization
- Yeah, I know you used batch normalization, but how did you store the normalization parameters for inference, and what values did you use?
  - I used Batch Normalization
- Can you say anything other than, “I used batch normalization”
  - I used Batch Normalization

*Warning: I don’t understand popular culture, and I didn’t check with my kids before writing this.

What is Convolution?

- Discrete-time convolution is defined as*

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

$$= w(j) * z(j) = \sum_{k=-\infty}^{\infty} w(j-k)\, z(k)$$

In 2D, convolution is given by

$$x(j_1, j_2) = w(j_1, j_2) * z(j_1, j_2) = z(j_1, j_2) * w(j_1, j_2)$$

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

*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 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 $p \times p$

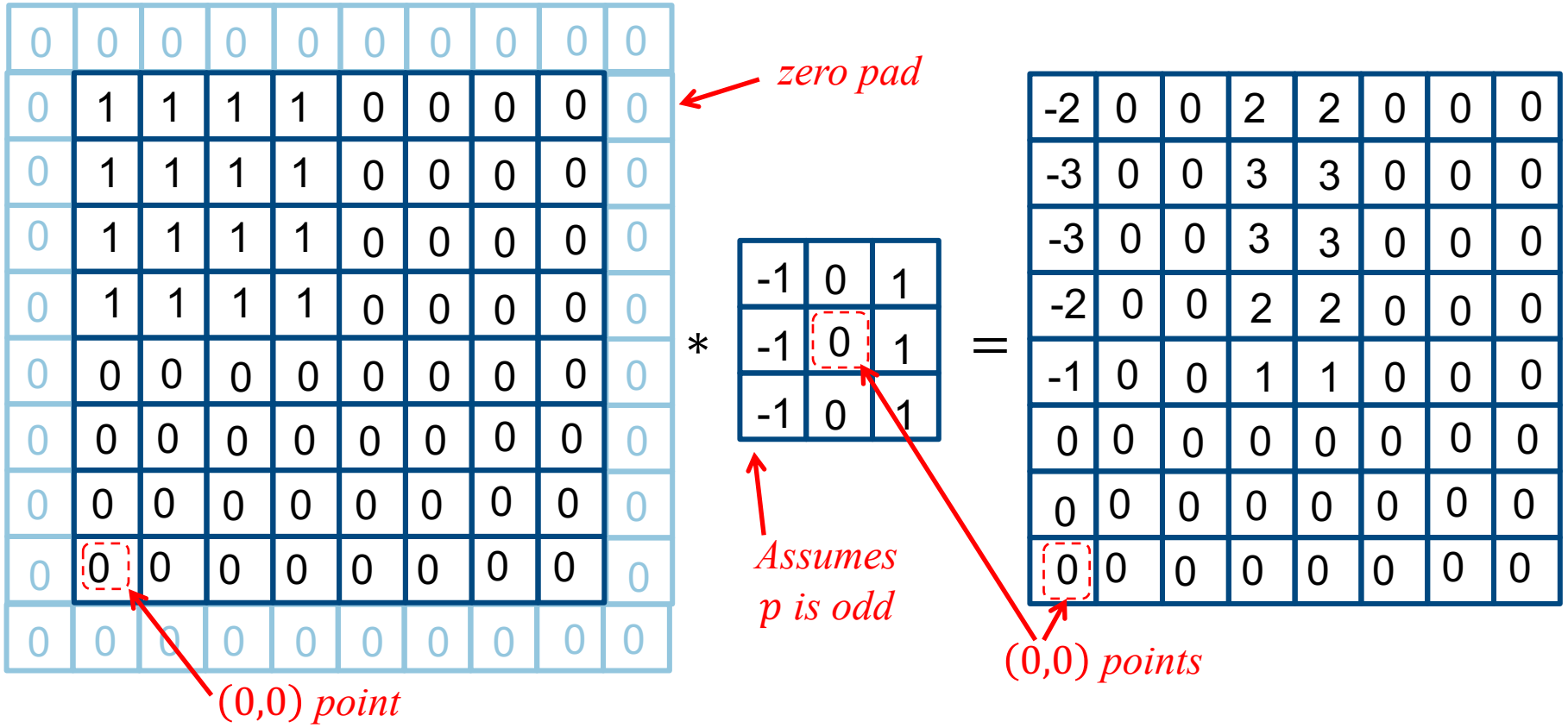
2D Convolution with "Same" boundary

- For 3×3 filter is given by

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

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

Parenthesizes denote convolution

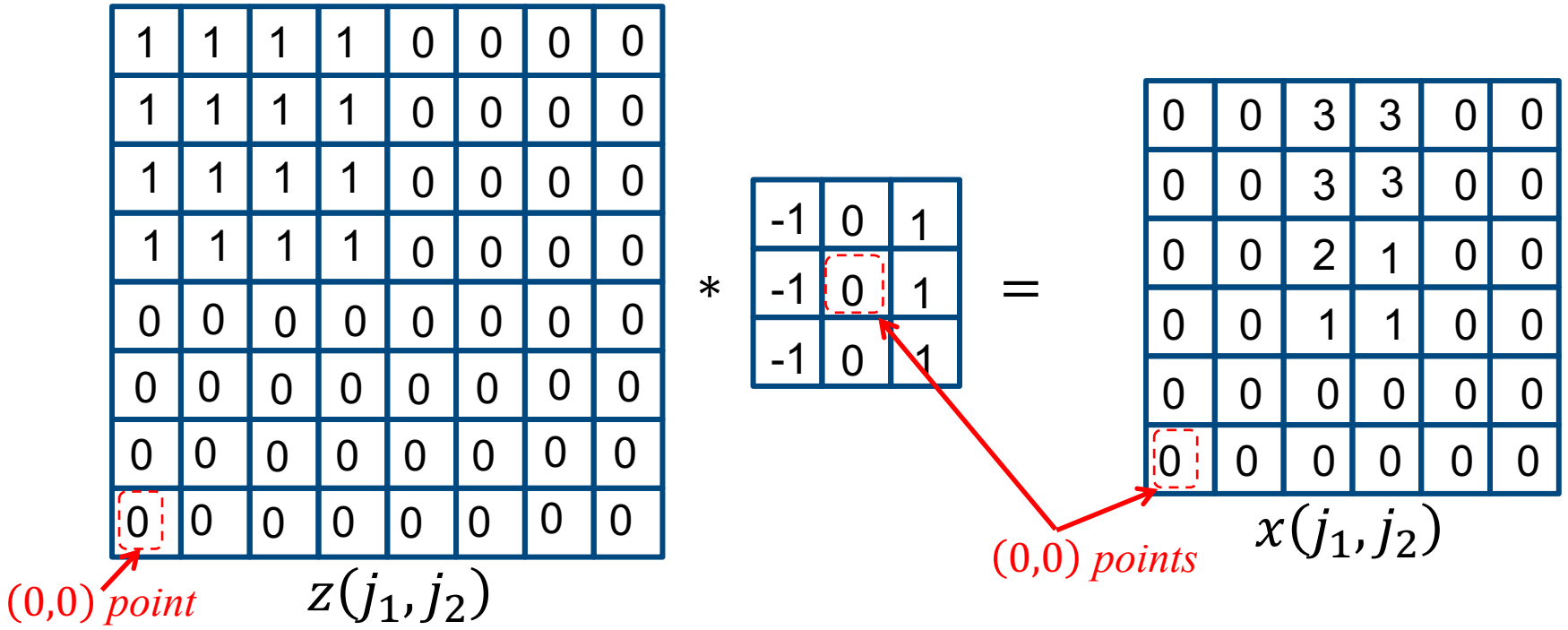


2D Convolution with "Valid" Boundary

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

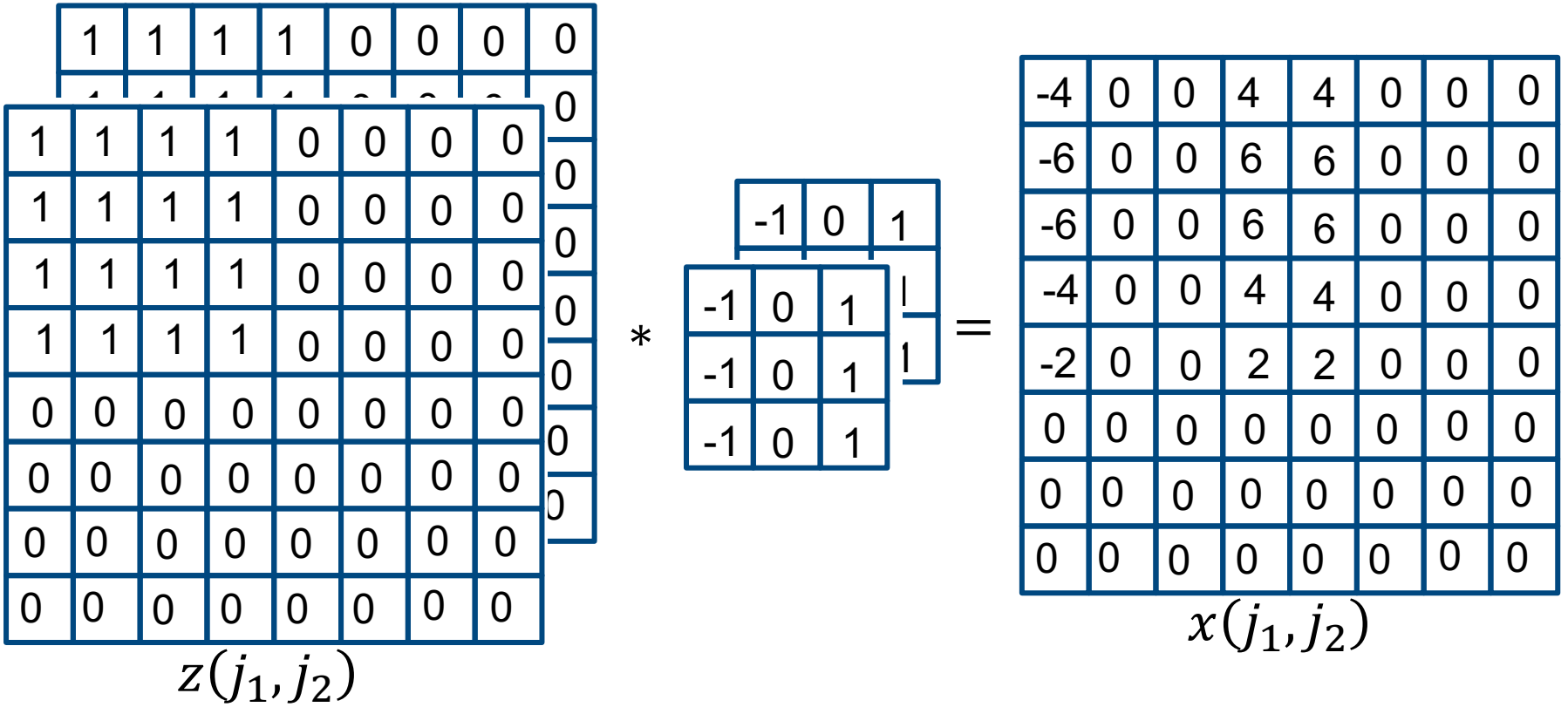


2D Convolution with Vector Input

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

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

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

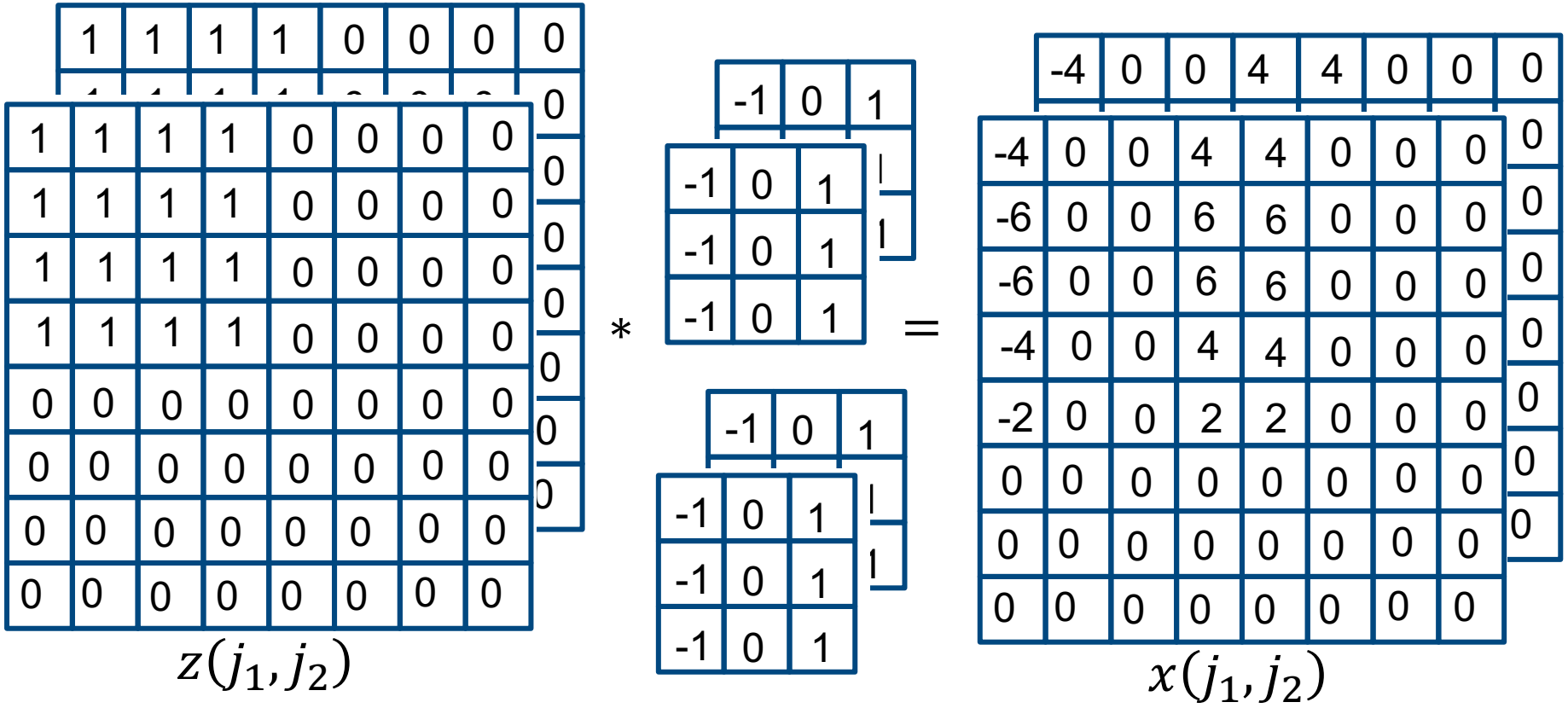


2D Convolution with Vector Input/Output

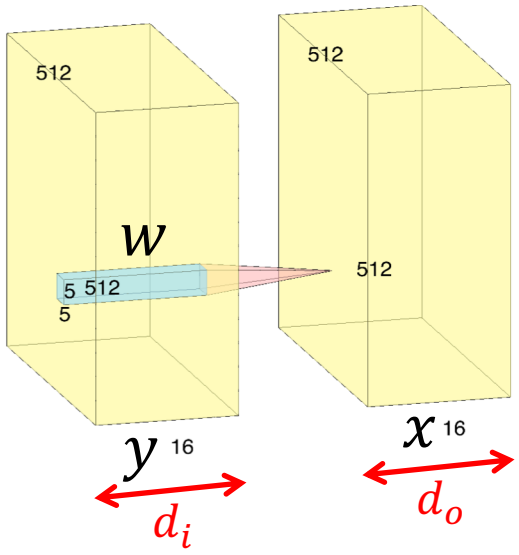
- Vector 2D convolution of depth $d_i = 2$ and $d_o = 2$

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

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



Convolution Diagram



- Sum notation

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

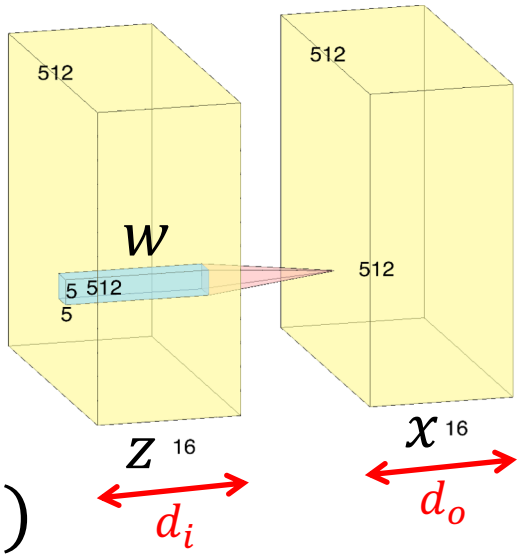
  - Tensor notation

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

where

w is $5\times5\times16\times16 = 6{,}400$
y is $512\times512\times16 = 4{,}194{,}304$
x is $512\times512\times16 = 4{,}194{,}304$

Single Layer CNN



- Single layer 2D CNN example:

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

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

J3 is output channel number, i is input channel number

- Tensor structures

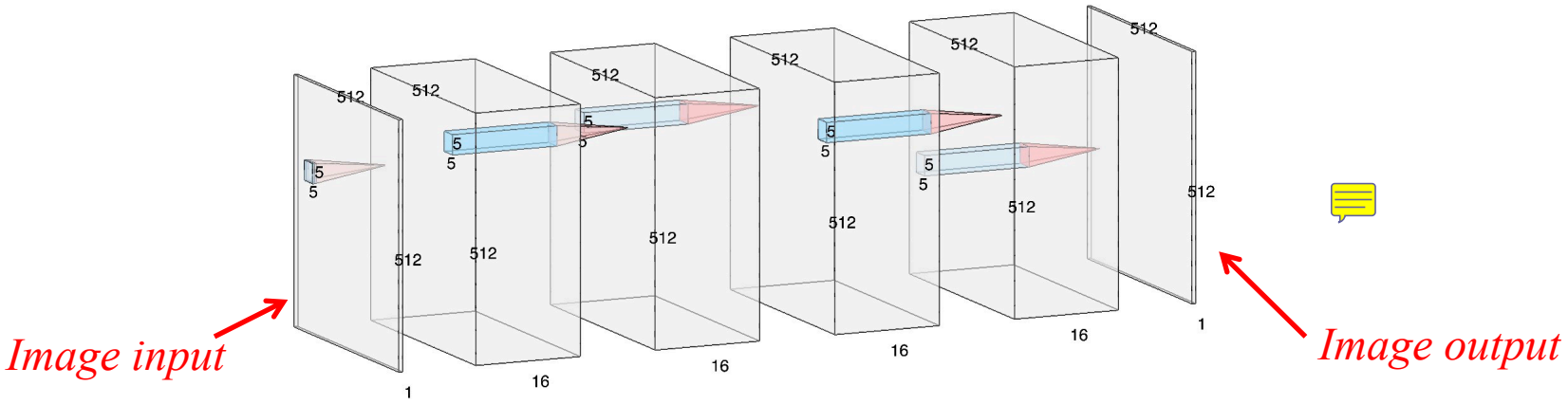
w is $5\times5\times16\times16 = 6{,}400$
b is $16 = 16$

Parameter Tensors

z is $512\times512\times16 = 4{,}194{,}304$
x is $512\times512\times16 = 4{,}194{,}304$

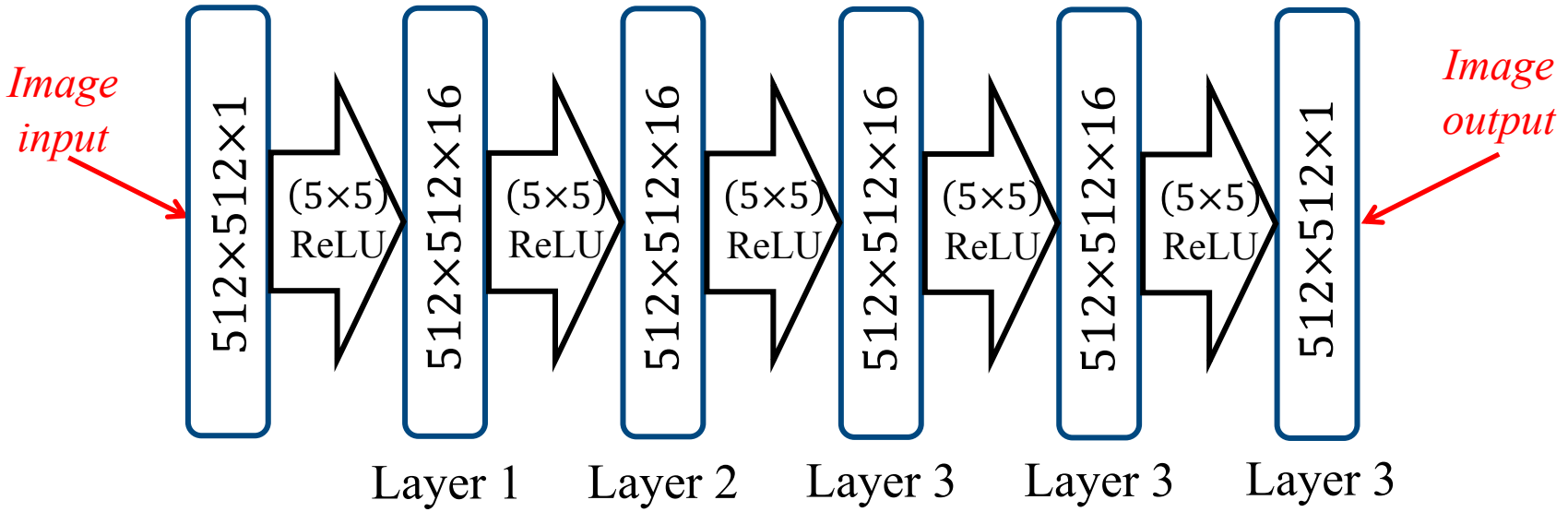
Feature or State Tensors

5 Layer CNN



- Parameters:
  - Layer 1: filter 5×5×1×16; offset 16; ReLU; #params 416
  - Layer 2: filter 5×5×16×16; offset 16; ReLU; #params 6,416
  - Layer 3: filter 5×5×16×16; offset 16; ReLU; #params 6,416
  - Layer 4: filter 5×5×16×16; offset 16; ReLU; #params 6,416
  - Layer 5: filter 5×5×16×1; offset 16; ReLU; #params 416
  - Total # params = 20,080

Simpler Diagram for 5 Layer CNN

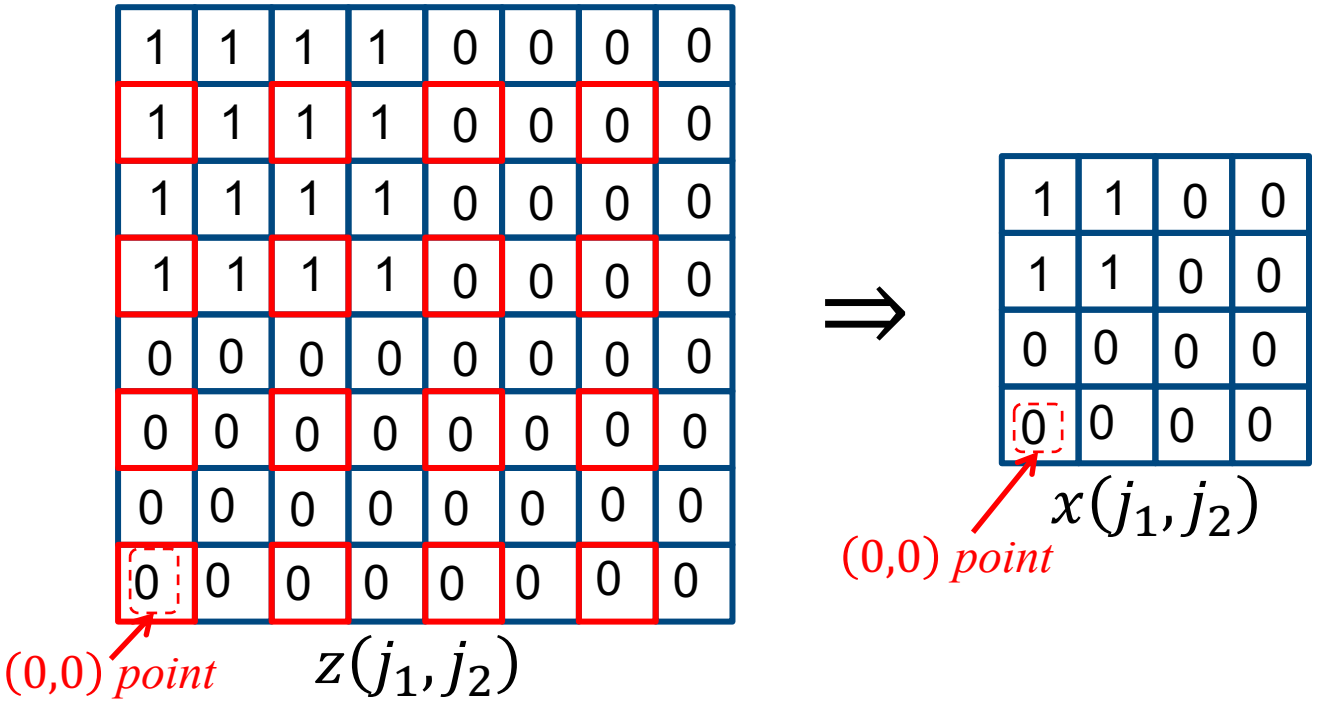


- Good news:
  - Dramatic reduction in number of parameters as compare to dense network
  - Spatially invariant behavior (except for boundaries)
  - For interior layers, each pixel is formed by a 16 dimensional feature vector
  - Intuition: CNN can design its own features!
- Bad news:
  - Not very useful
  - 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

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



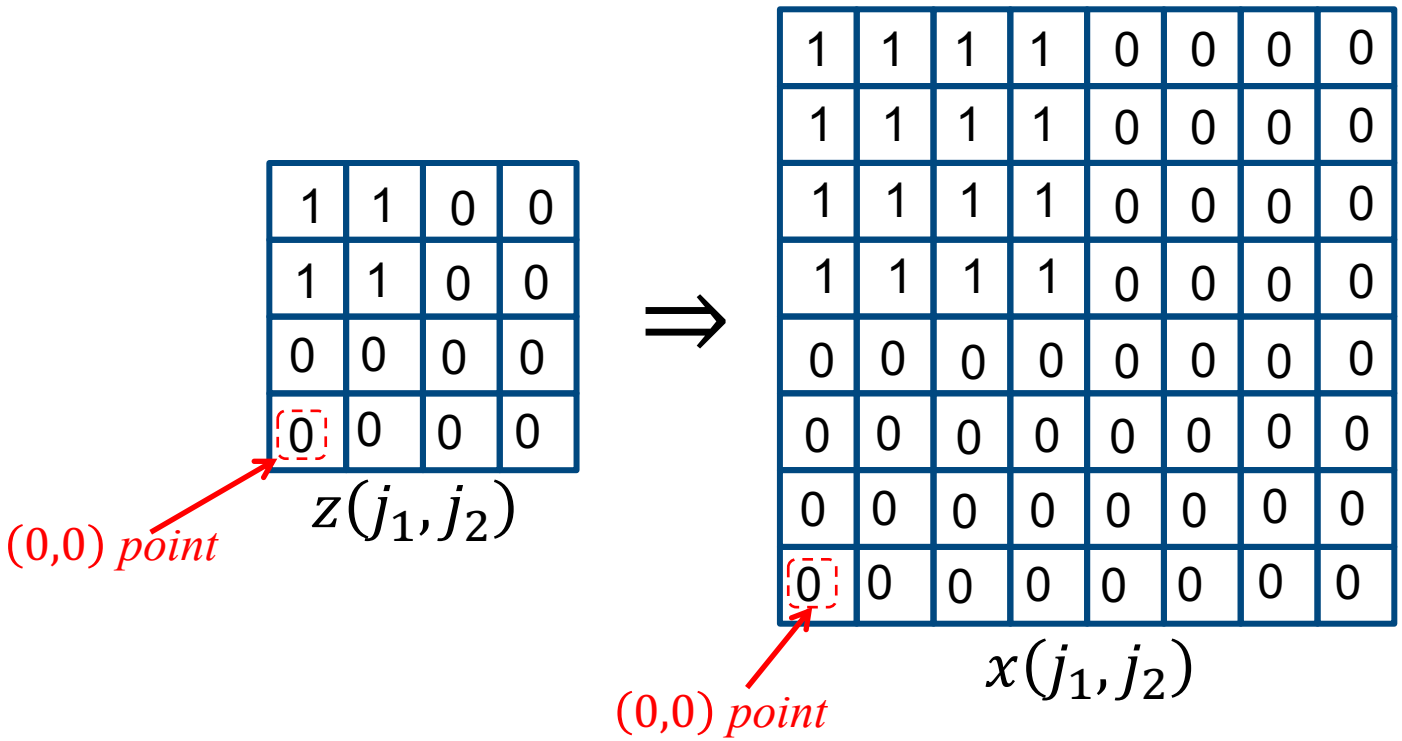
- Comments:
  - Output has reduced dimensions of $(\lfloor N_v/d_v \rfloor, \lfloor N_h/d_h \rfloor)$
  - Also known as decimation
  - Can be used with most operators

CNN Upsampling

also known as pixel replication

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

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



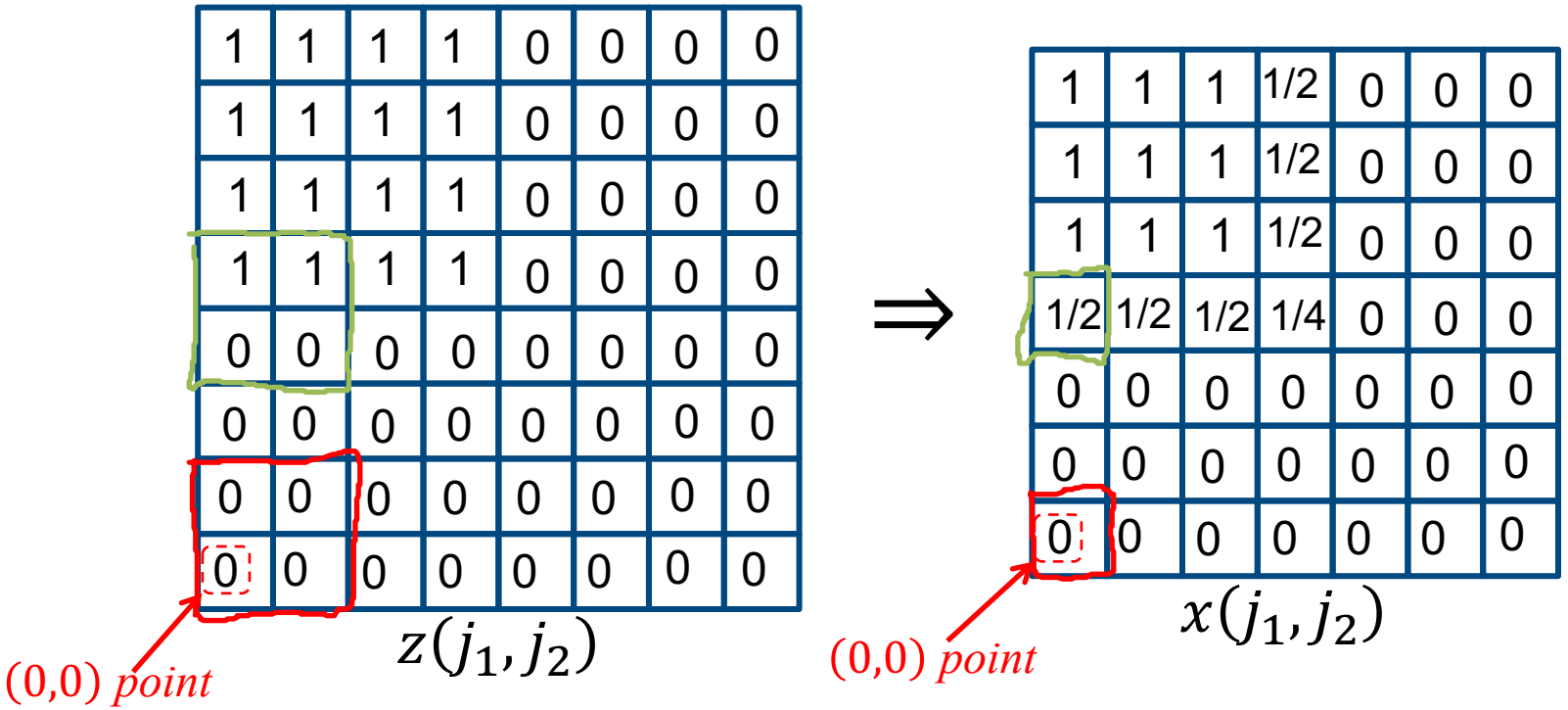
- Comments:
  - Also known as pixel replication
  - Output has dimensions of $(L_v N_v, L_h N_h)$

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

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

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

1	1	1	1	0	0	0	0
1	1	1	1	0	0	0	0
1	1	1	1	0	0	0	0
1	1	1	1	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

$z(j_1, j_2)$

$\Rightarrow$

1	1	0	0
1	1	0	0
0	0	0	0
0	0	0	0

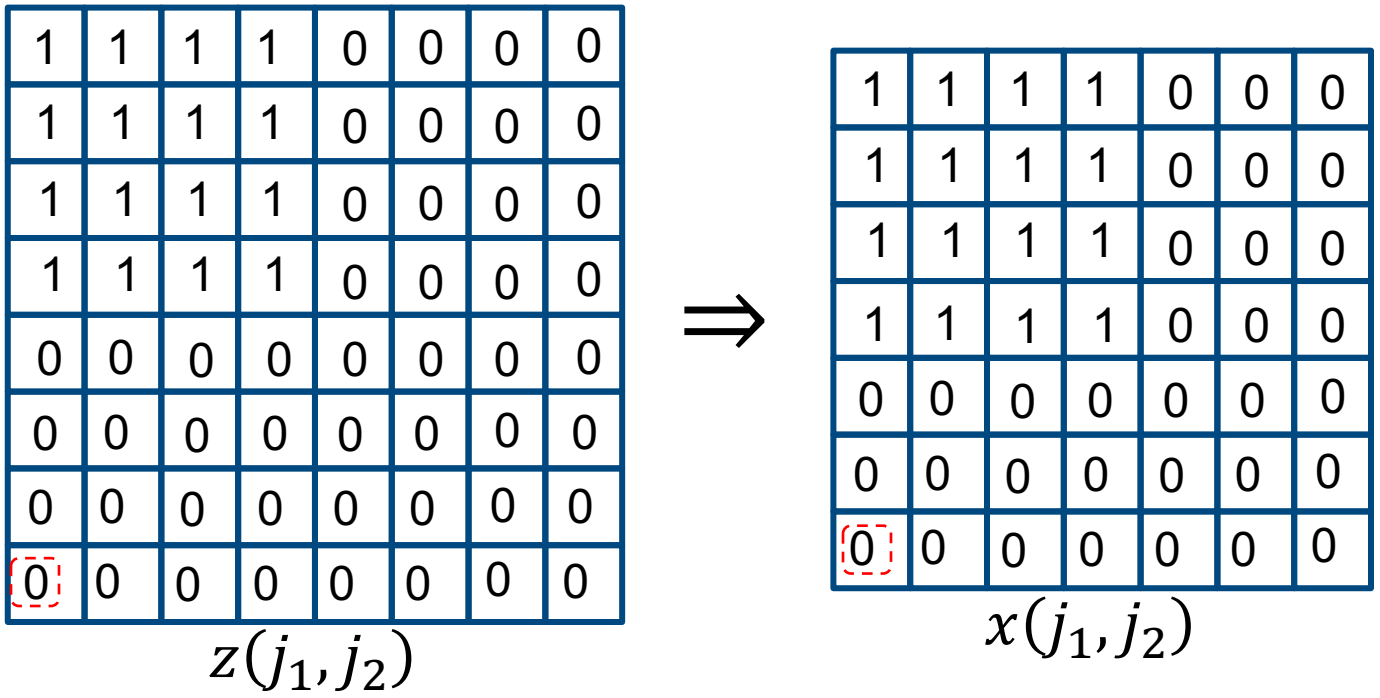
$x(j_1, j_2)$

CNN Max Pooling

- 2D max pooling equation

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

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

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

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

1	1	1	1	0	0	0	0
1	1	1	1	0	0	0	0
1	1	1	1	0	0	0	0
1	1	1	1	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

$z(j_1, j_2)$

$\Rightarrow$

1	1	0	0
1	1	0	0
0	0	0	0
0	0	0	0

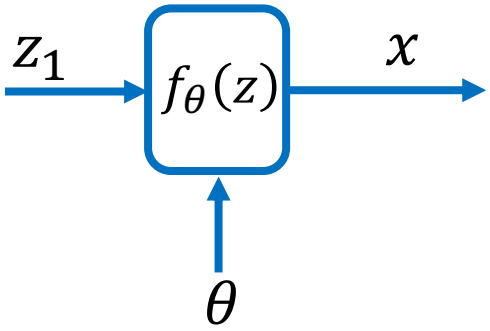
$x(j_1, j_2)$

Adjoint Gradients for CNNs

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

Backpropagation for CNN Nodes

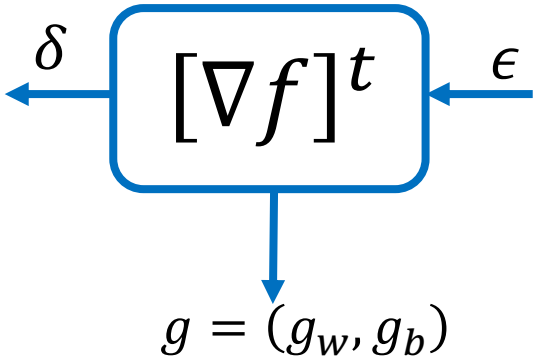
- For each node you need two functions:
  - Forward propagation function:



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

You need a software implementation of the forward function.

  - Adjoint gradient function:

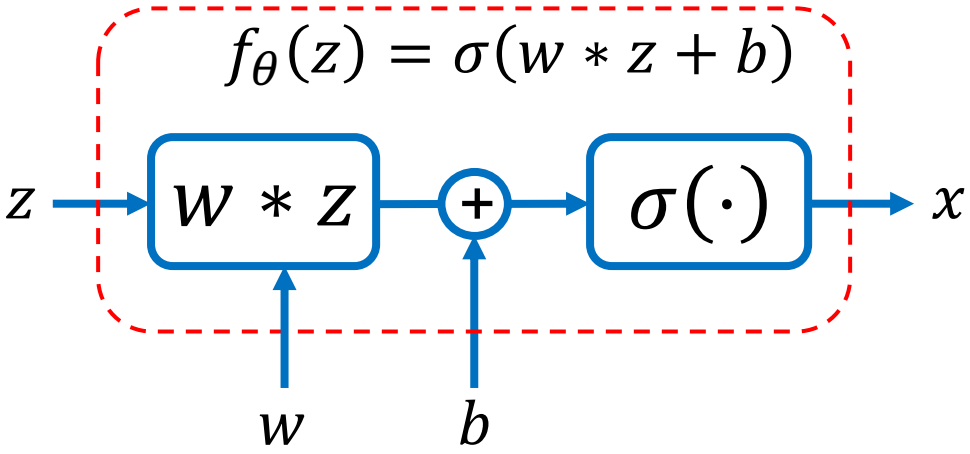


$$[\delta, g_w, g_b] \leftarrow G(\epsilon, z, \theta)$$

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 $\theta = (w, b)$

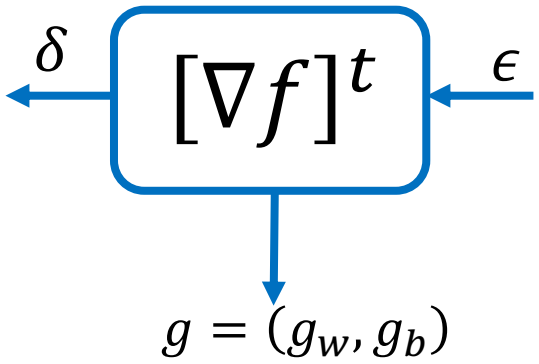
$$\nabla_\theta f_\theta(z) = \left[\nabla_w f_{(w,b)}(z), \nabla_b f_{(w,b)}(z)\right]$$

  - And, we will also need:

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

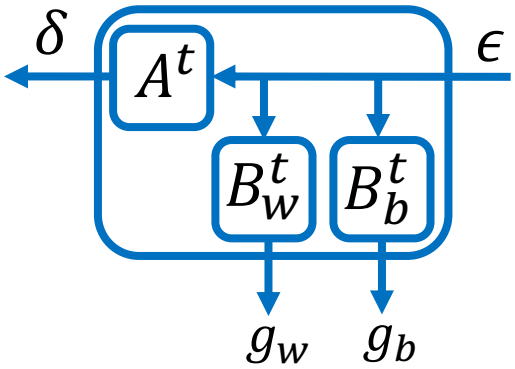
Closer Look at the Adjoint Gradient

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



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

- Conceptually, this is



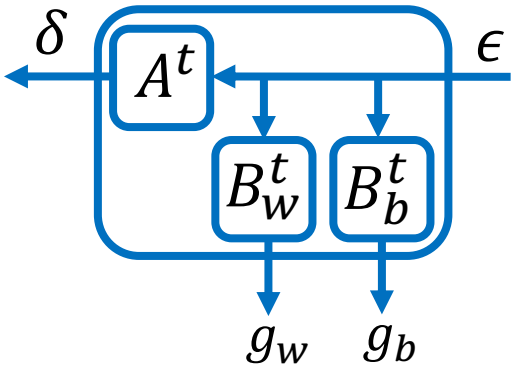
$$A = \nabla_z f_\theta(z) \quad \Longleftarrow \textit{Gradient with input}$$

$$B_w = \nabla_w f_\theta(z) \quad \Longleftarrow \textit{Gradient with filter weights}$$

$$B_b = \nabla_b f_\theta(z) \quad \Longleftarrow \textit{Gradient with offset weights}$$

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

You need this output.

$N_y \times 1$ δ - adjoint gradient $=$ $N_y \times N_x$ adjoint function gradient $A_{j,i} = \frac{\partial [f_\theta(y_k)]_j}{\partial z_i}$ $N_x \times 1$ ϵ - error vector

You're given this input

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

But how??

- Big Idea: Directly compute output without computing gradient!

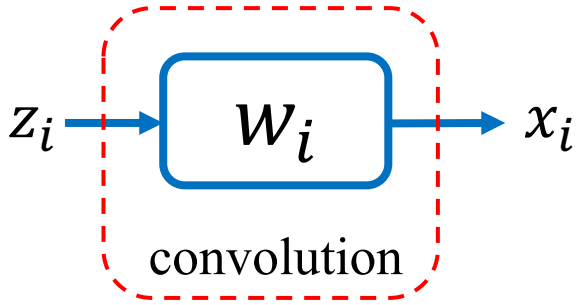
Adjoint Gradient of Convolution w.r.t. Input

- Gradient of output with input:

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

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

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

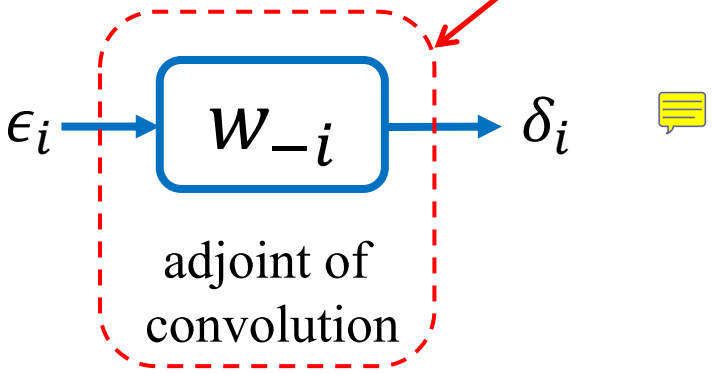


- Adjoint gradient:

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

$$\delta_i = \sum_j w_{j-i}\, \epsilon_j = w_{-i} * \epsilon_i$$

Adjoint uses the time-reversed impulse response



Adjoint Gradient of Convolution w.r.t. Weights

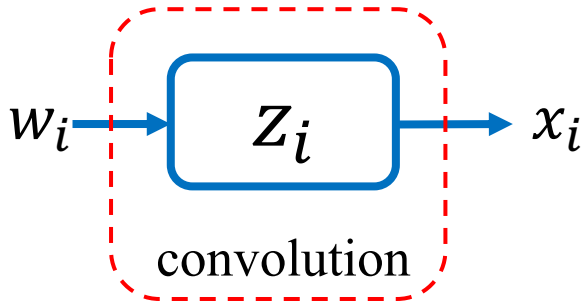
convolution is commutative, but correlation is not commutative

- Gradient of output with weights:

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

$$x = Aw$$

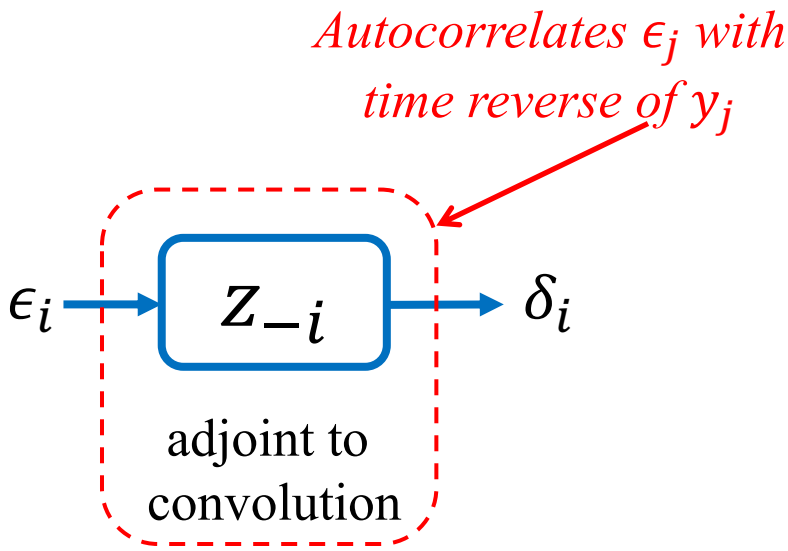
$$\text{where } \frac{\partial x_i}{\partial w_j} = A_{i,j} = z_{i-j}$$



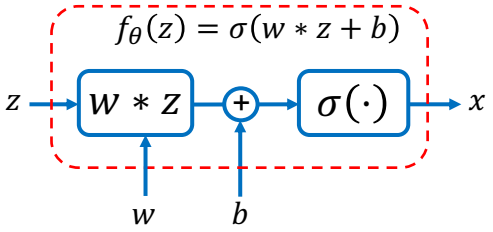
- Adjoint gradient:

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

$$\delta_i = \sum_j z_{j-i}\, \epsilon_j$$



Adjoint Gradient of w.r.t. Input



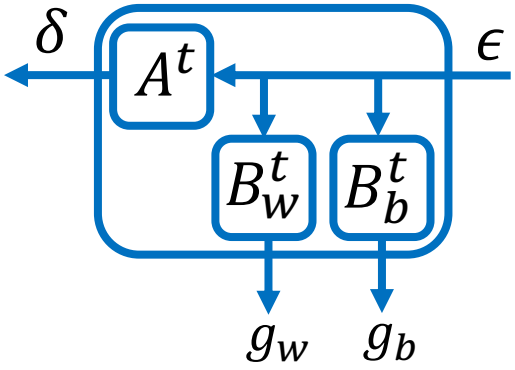
- Forward function:

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

  - The adjoint gradient $\left[\nabla_y f\right]^t \epsilon$ is given by

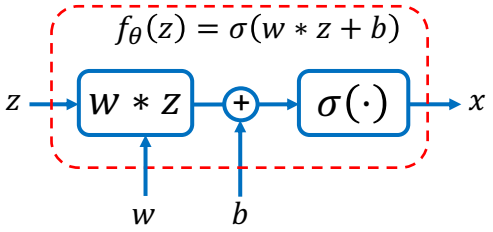
$$\delta_{j_1,j_2,j_3} = w_{(-j_1,-j_2),j_3} \left[\nabla\sigma\right]^{i_1,i_2,i_3}{}_{(j_1,j_2),j_3} \epsilon_{i_1,i_2,i_3}$$

time-reversed = 1d
space-reversed = 2d



Fast because it never computes A!

Adjoint Gradient w.r.t. b

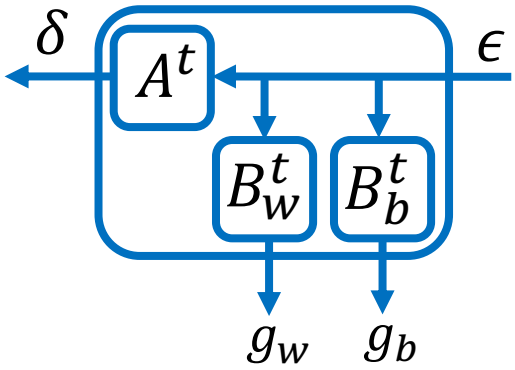


- Forward function:

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

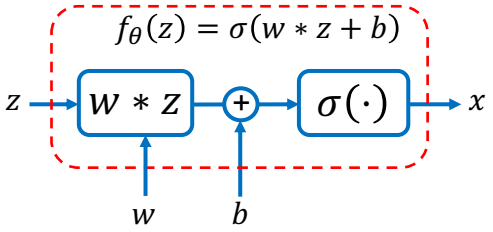
  - The adjoint gradient $[\nabla_b f]^t \epsilon$ is given by

$$[g_b]_{j_3} = \mathbf{1}^{j_1,j_2}[\nabla\sigma]^{i_1,i_2,i_3}{}_{j_1,j_2,j_3}\epsilon_{i_1,i_2,i_3}$$



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

Adjoint Gradient w.r.t. *w*



- Forward function:

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

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

$$[g_w]_{j_1,j_2,j_3} = y^{(-j_1,-j_2)} [\nabla\sigma]^{i_1,i_2,i_3}{}_{(j_1,j_2),j_3} \epsilon_{i_1,i_2,i_3}$$