ConvNets and Babysitting the Learning Process

Arjun Jain | 10 March 2017

Agenda

- CNN building blocks: ReLU, MaxPool, Convolution
- Weight Initialization
- Baby sitting the Learning Process
- Hyperparameter Optimization
- Apply all these to a real world example – Classifying CIFAR-10

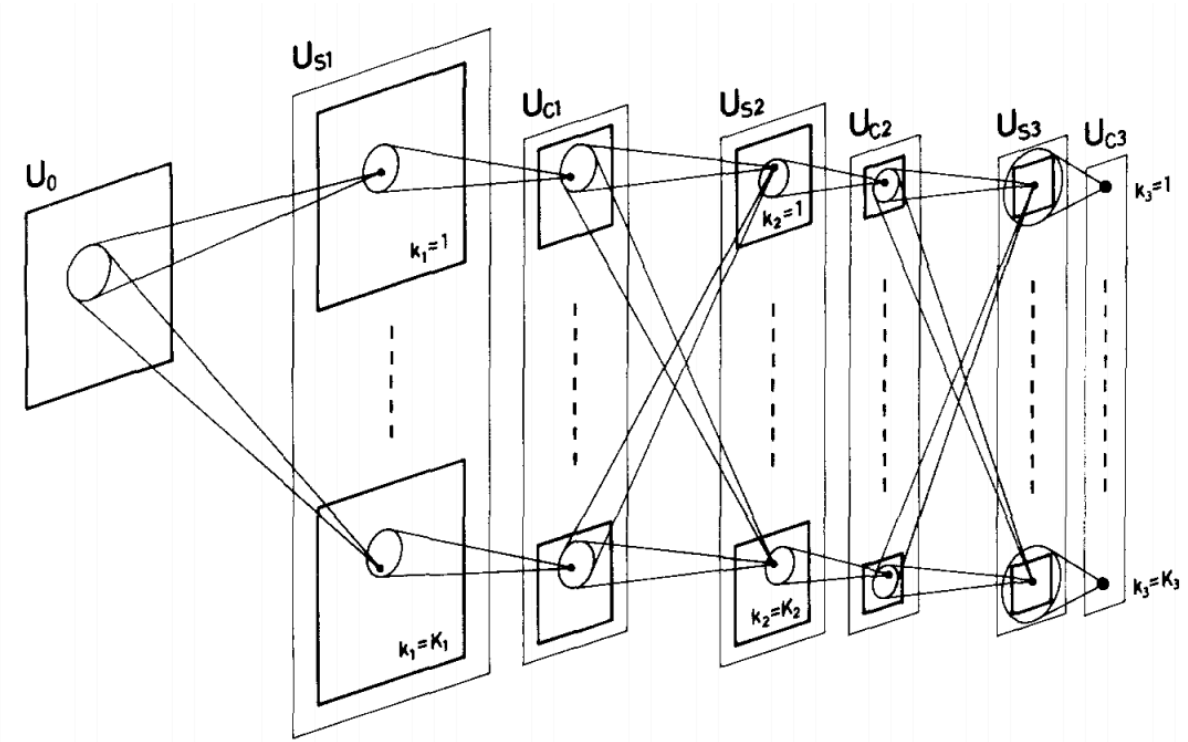
Sources

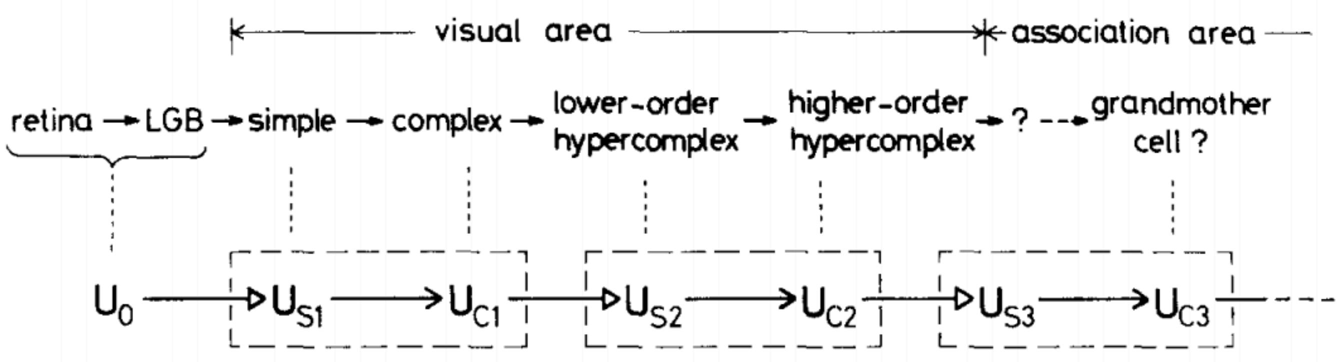
A lot of the material has been shamelessly and gratefully collected from:

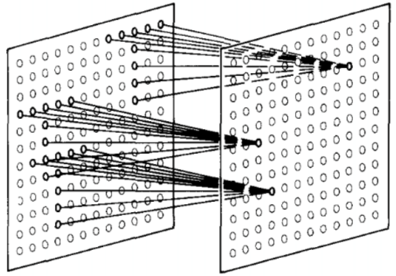
- **http://cs231n.stanford.edu/**
- https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-history-training/
- https://adeshpande3.github.io/adeshpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-About.html
- https://research.fb.com/learning-to-segment/
- https://research.fb.com/deep-learning-tutorial-at-cvpr-2014/
- http://code.madbits.com/wiki/doku.php?id=tutorial_morestuff
- https://www.cs.ox.ac.uk/people/nando.defreitas/machinelearning/practicals/practical4.pdf
- http://torch.ch/docs/developer-docs.html
- https://github.com/torch/nn/blob/31d7d2bc86a914e2a9e6b3874c497c60517dc853/doc/module.md
- https://web.stanford.edu/group/pdplab/pdphandbook/handbookch6.html
- http://neuralnetworksanddeeplearning.com/chap2.html

Brief History – The First ConvNet

- Neocognitron: multiple convolutional and pooling layers similar to modern networks, but the network was trained by using a reinforcement scheme
- Did not still use backpropagation
- Translational invariant

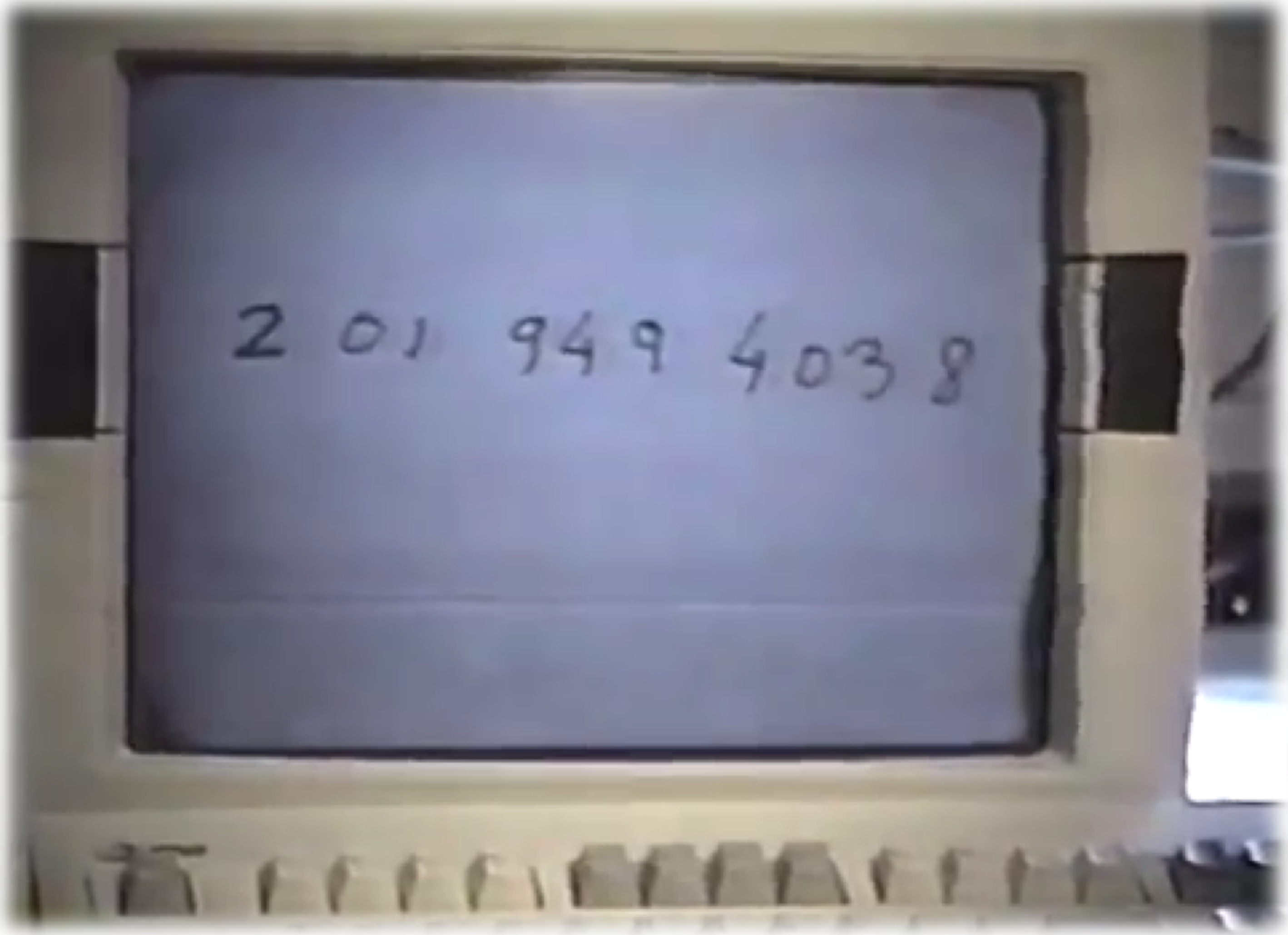




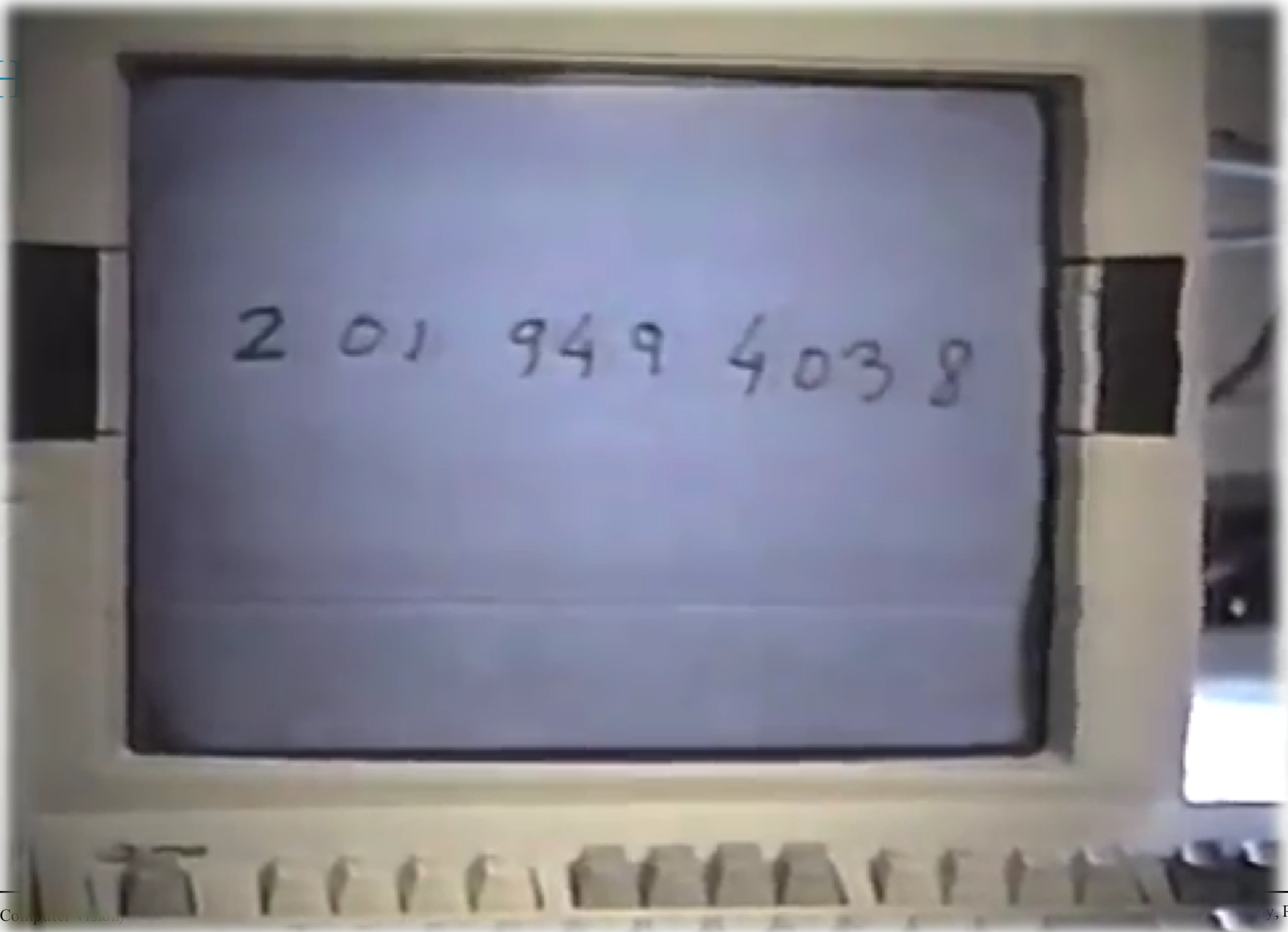


Kunihiko Fukushima

Brief History – LeNet-5 In Action

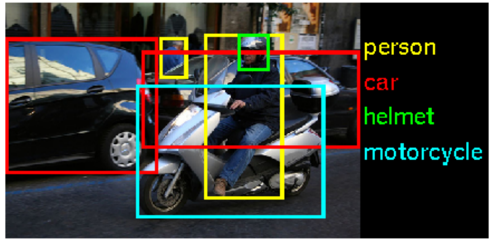


Brief H



Brief History – The Tipping Point

- 2012 ILSVRC: ImageNet Large-Scale Visual Recognition Challenge – Annual World Cup of Computer Vision
- More than a million training images and 1000 categories



ImageNet Classification with Deep Convolutional Neural Networks

Alex Krizhevsky
University of Toronto
kriz@cs.utoronto.ca

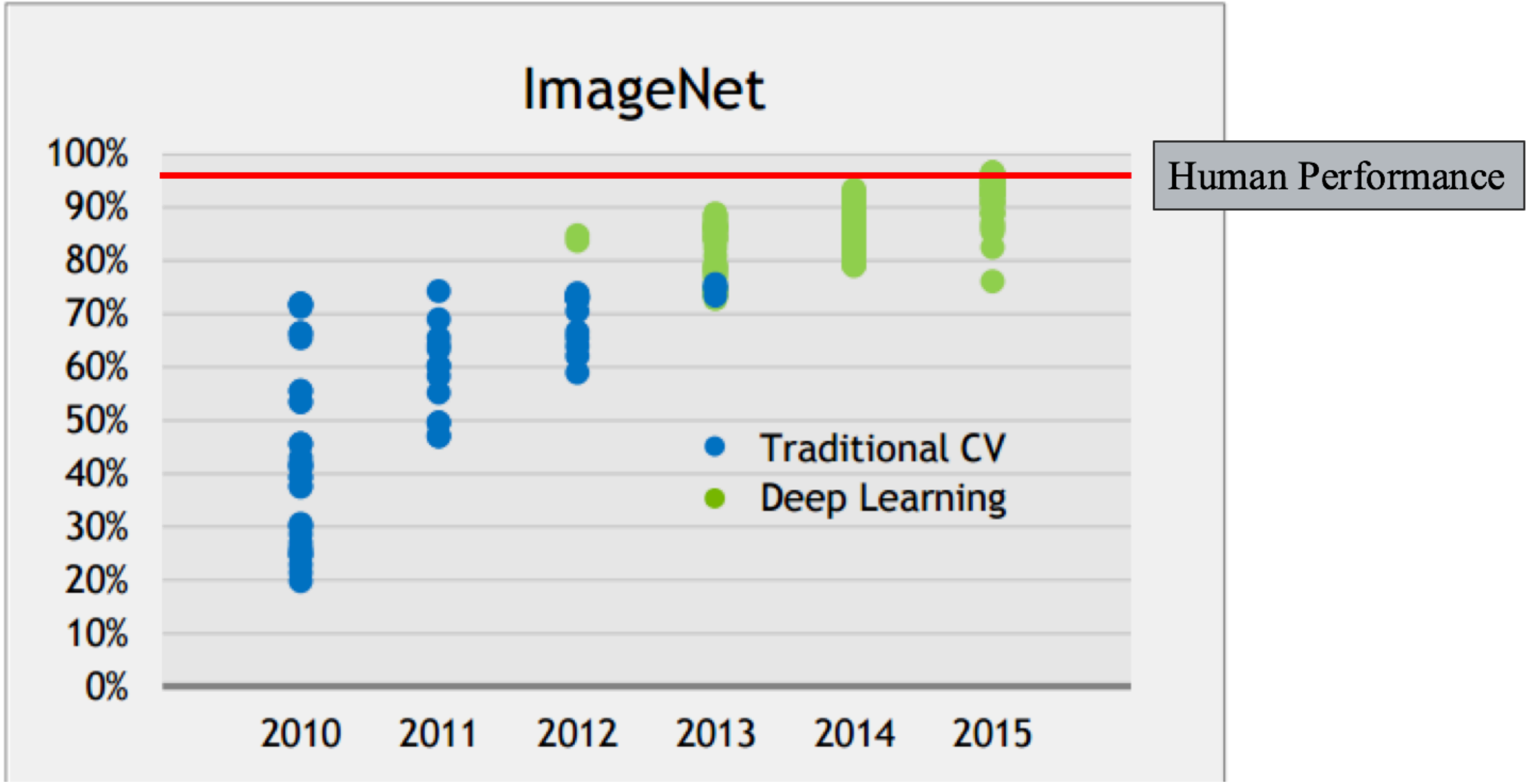
Ilya Sutskever
University of Toronto
ilya@cs.utoronto.ca

Geoffrey E. Hinton
University of Toronto
hinton@cs.utoronto.ca

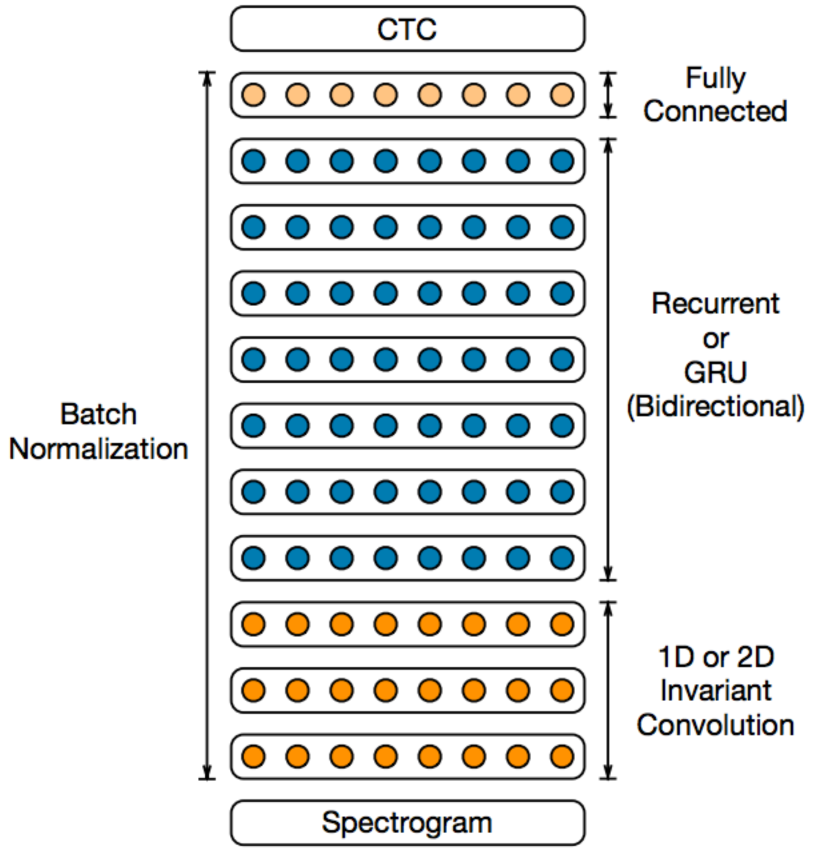
Brief History – The Tipping Point

- Reported 15.4% Top 5 error rate. The next best entry achieved an error of 26.2%
- > 8000 citations
- The coming out party for CNNs in the computer vision community
- Shocked the computer vision community. Trained end-to-end on raw pixels, without using any feature engineering methods
- From here it was apparent that deep learning would take over computer vision and that other methods would not be able to catch up

Why ConvNets?



Used in Speech too!



Deep Speech 2: End-to-End Speech Recognition in English and Mandarin
Amodei et al., Baidu Research

Acoustic modelling from the signal domain using CNNs
Pegah Ghahremani, Vimal Manohar, Daniel Povey, Sanjeev Khudanpur

Deep Convolutional Neural Networks for LVCSR
Tara N. Sainath, Abdel-rahman Mohamed, Brian Kingsbury, Bhuvana Ramabhadran

Brief History – So What Changed (since the 1970s)?

- Three things:
    - Availability of large amounts of labeled data - 15 million annotated images from a total of over 22,000 categories
    - Compute power – A single NVidia TITAN X card churns of 11 TFLOPS with ~3500 cores
    - Algorithms:
        - ReLU - Found to decrease training time
        - Dropout – prevent overfitting to the training data

Deep Learning – Today – Human Computer Interaction

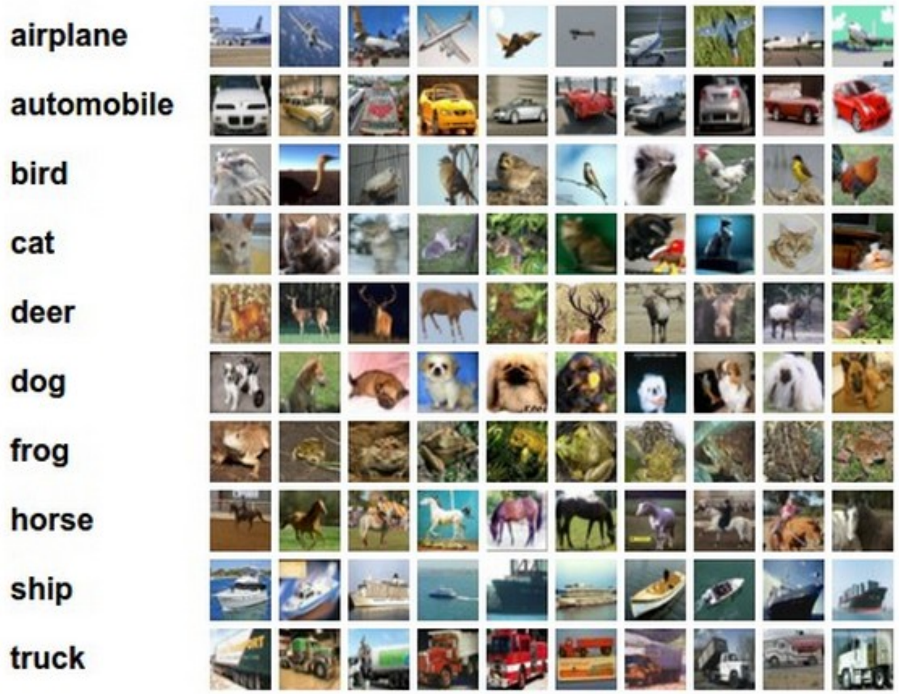

Welcome,
inside the 750i

Deep Learning – Today – Lip Reading

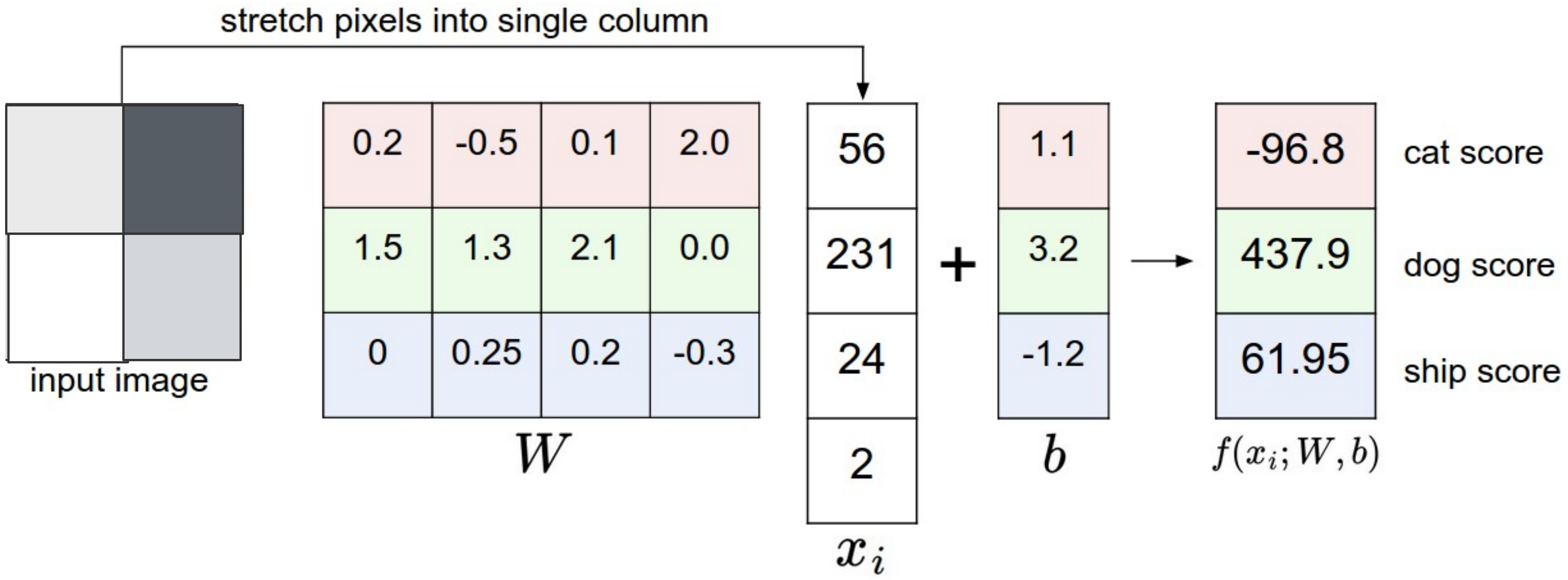

BBC One HD
07-Apr-2016 22:51:37
THE GOVERNMENT WILL PAY FOR BOTH SIDES

Linear Classification: CIFAR-10

10 labels
50,000 training images
10,000 test images
each image is an array of size **32 x 32 x 3 = 3072** numbers total



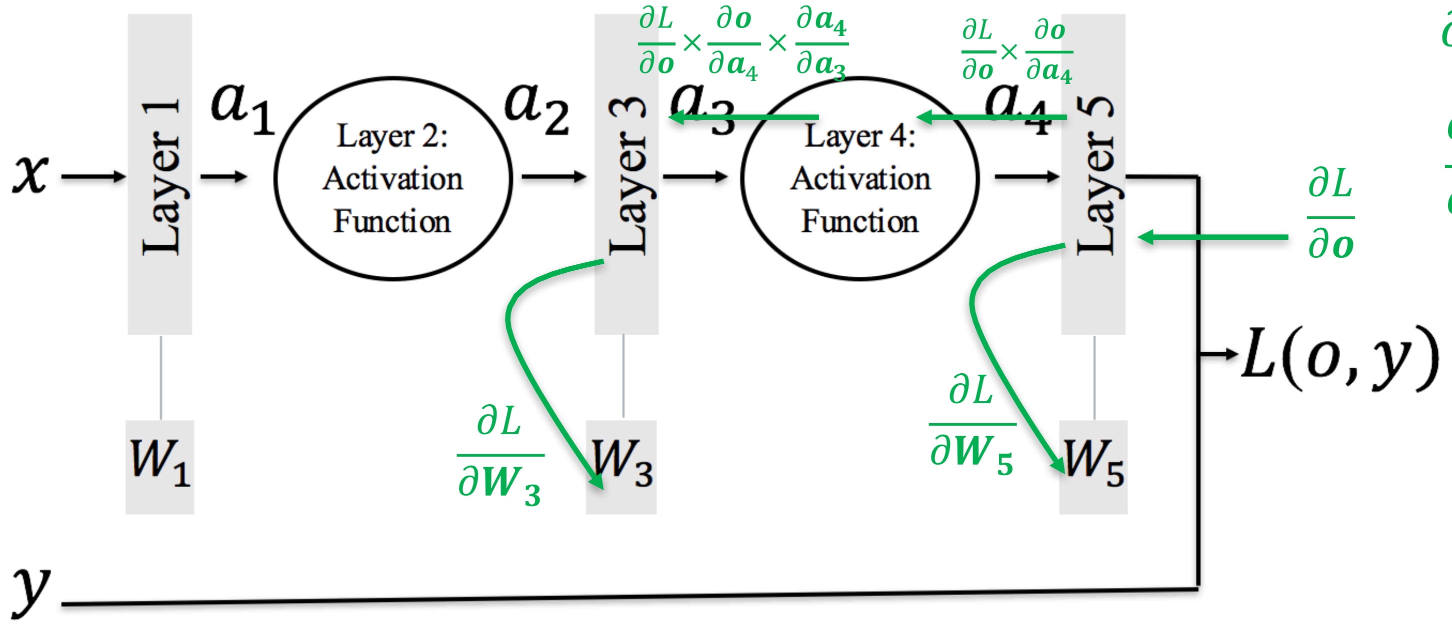
Example with an Image with 4 Pixels, and 3 Classes (cat/dog/ship)



Multiple Layers – Back Prop: Chain Rule

$\frac{\partial \boldsymbol{o}}{\partial \boldsymbol{a_4}}$ *is the Jacobian* $\epsilon\ \mathbb{R}^{\dim(o)\times\dim(a_4)}$

$\frac{\partial L}{\partial \boldsymbol{o}}$ *is the gradient* $\epsilon\ \mathbb{R}^{1\times n}$



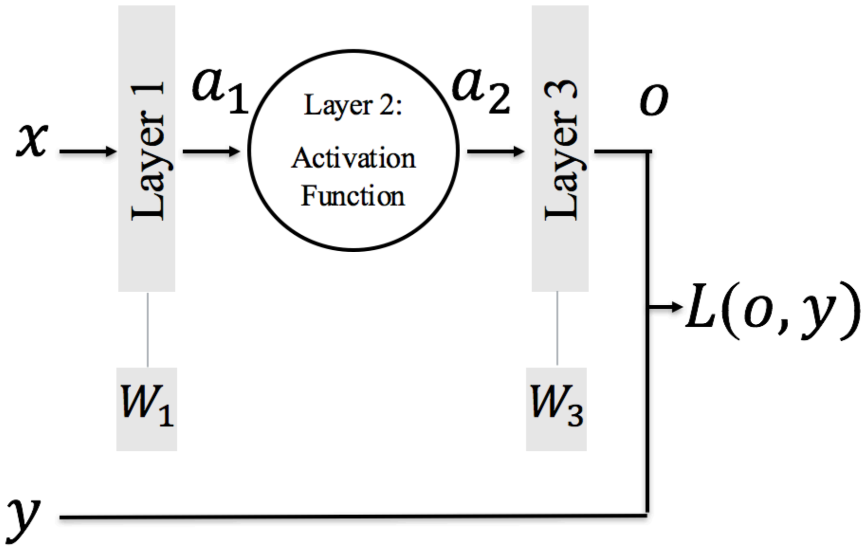
We want:

$$\frac{\partial L}{\partial \boldsymbol{W_1}}, \frac{\partial L}{\partial \boldsymbol{W_3}}, \frac{\partial L}{\partial \boldsymbol{W_5}}$$

Now we can compute:

$$\frac{\partial L}{\partial \boldsymbol{W_3}} = \frac{\partial L}{\partial \boldsymbol{o}} \times \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{a_4}} \times \frac{\partial \boldsymbol{a_4}}{\partial \boldsymbol{a_3}} \times \frac{\partial \boldsymbol{a_3}}{\partial \boldsymbol{W_3}}$$

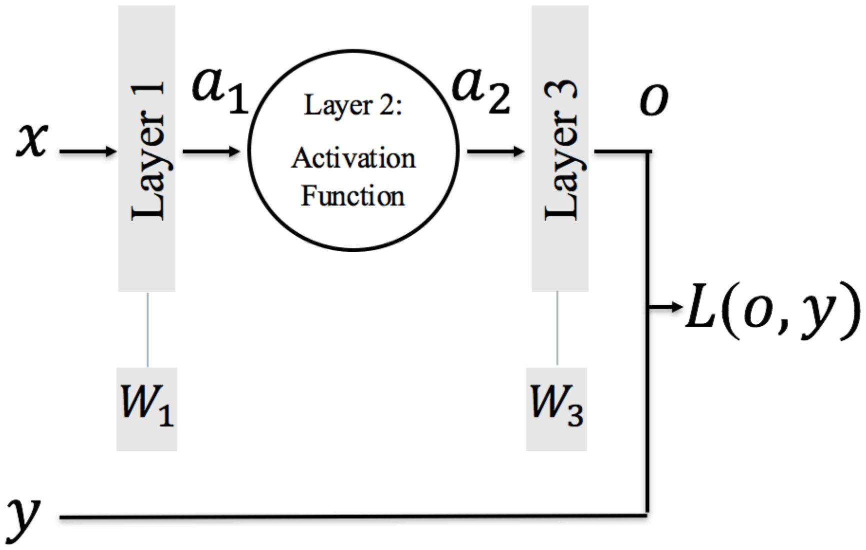
Multiple Layers – Feed Forward – In Torch7



- Example: 3 modules layer1, layer2, layer3
- By hand:
  - `a1 = layer1:forward(x)`
  - `a2 = layer2:forward(a1)`
  - `o = layer3:forward(a2)`
- Using `nn.Sequential`:
  - `model = nn.Sequential()`
  - `model:add(layer1)`
  - `model:add(layer2)`
  - `model:add(layer3)`
  - `o = model:forward(x)`

  (output is returned, but also stored internally)

Multiple Layers – Feed Forward – In Torch7



- `criterion = nn.SomeCriterion()`
- `loss = criterion:forward(o, y)`
- `dl_do = criterion:backward(o, y)`
- Gradient with respect to input is returned
- Arguments are input and gradient with respect to outpu
- By hand:
  - `l3_grad = layer3:backward(a2, dl_do)`
  - `l2_grad = layer2:backward(a1, l3_grad)`
  - `l1_grad = layer1:backward (x, l2_grad)`
- Using nn.Sequential:
  - `l1_grad = model:backward(x, dl_do)`

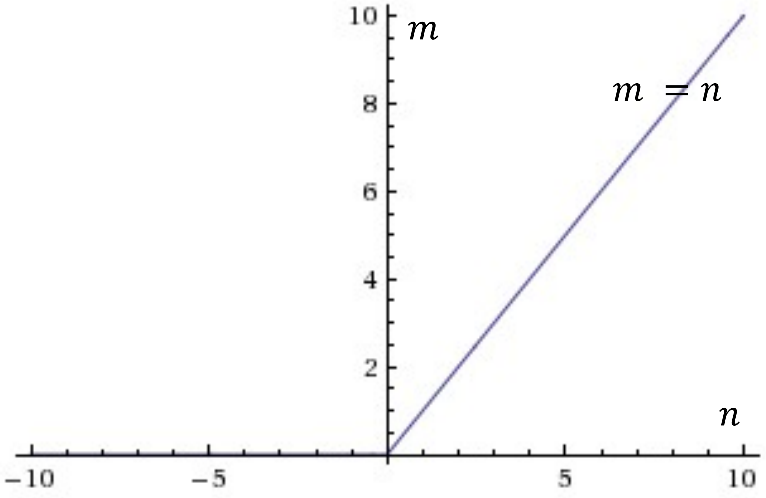
Building Blocks: **Activation Functions (ReLU)**

https://github.com/stencilman/CS763_Spring2017/blob/master/Notebooks/ReLU.ipynb

Building Blocks – ReLU – Activation Function

$\boldsymbol{n} \in \mathbb{R}^d$ → nn.ReLU → $\boldsymbol{m} \in \mathbb{R}^d$

$$m_i = \max(0, n_i)$$



$$m_i = \begin{cases} 0 & if \;\; n_i < 0 \\ n_i & if \;\; n_i > 0 \end{cases}$$

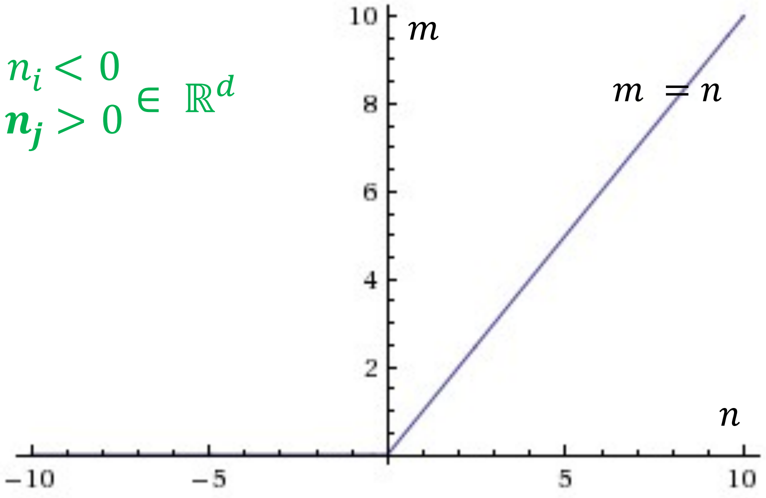
Building Blocks – ReLU

$$\frac{\partial L}{\partial \boldsymbol{m}} \cdot \frac{\partial \boldsymbol{m}}{\partial \boldsymbol{n}} \in \mathbb{R}^{1\times \dim(n)} \qquad \frac{\partial \boldsymbol{m}}{\partial \boldsymbol{n}} \in \mathbb{R}^{\dim(m)\times \dim(n)} \qquad \frac{\partial L}{\partial \boldsymbol{m}} \in \mathbb{R}^{1\times \dim(m)}$$

$$\boldsymbol{n} \in \mathbb{R}^d \longrightarrow \text{nn.ReLU} \longrightarrow \boldsymbol{m} \in \mathbb{R}^d$$

$$m_i = \max(0, n_i)$$

$$\frac{\partial m_i}{\partial n_j} = \frac{\partial max(0,n_i)}{\partial n_j} = \begin{cases} 0 & if \;\; n_i < 0 \\ 1 & if \;\; \boldsymbol{n_j} > 0 \end{cases} \in \mathbb{R}^d$$



$$m_i = \begin{cases} 0 & if \;\; n_i < 0 \\ n_i & if \;\; n_i > 0 \end{cases}$$

Building Blocks – ReLU

$$\frac{\partial L}{\partial \boldsymbol{m}} \cdot \frac{\partial \boldsymbol{m}}{\partial \boldsymbol{n}} \in \mathbb{R}^{1\times \dim(n)} \qquad \frac{\partial \boldsymbol{m}}{\partial \boldsymbol{n}} \in \mathbb{R}^{\dim(m)\times\dim(n)} \qquad \frac{\partial L}{\partial \boldsymbol{m}} \in \mathbb{R}^{1\times \dim(m)}$$

$$\boldsymbol{n} \in \mathbb{R}^d \longrightarrow \boxed{\text{nn.ReLU}} \longrightarrow \boldsymbol{m} \in \mathbb{R}^d$$

Input

`torch.rand` gives us random numbers unformly in the range $[0, 1]$. We subtract 0.5 to bring it to the range $[-0.5, 0.5]$

```
In [1]: require 'nn';
        n = torch.rand(5) - 0.5
        print(n)

Out[1]: -0.0044
        -0.1521
         0.4794
        -0.1014
         0.4201
        [torch.DoubleTensor of size 5]
```

Output

```
In [2]: relu = nn.ReLU()
        m = relu:forward(n)
        print(m)

Out[2]:  0.0000
         0.0000
         0.4794
         0.0000
         0.4201
        [torch.DoubleTensor of size 5]
```

Building Blocks – ReLU

$$\frac{\partial L}{\partial \boldsymbol{m}} \cdot \frac{\partial \boldsymbol{m}}{\partial \boldsymbol{n}} \in \mathbb{R}^{1\times \dim(n)} \qquad \frac{\partial \boldsymbol{m}}{\partial \boldsymbol{n}} \in \mathbb{R}^{\dim(m)\times \dim(n)} \qquad \frac{\partial L}{\partial \boldsymbol{m}} \in \mathbb{R}^{1\times \dim(m)}$$

$$\boldsymbol{n} \in \mathbb{R}^{d} \longrightarrow \boxed{\text{nn.ReLU}} \longrightarrow \boldsymbol{m} \in \mathbb{R}^{d}$$

So simplicity, we start by setting the gradient of the loss with respect to the output of this layer (flowing in through the next layer) $\frac{\partial L}{\partial O^l}$ to be all ones. Next, we see that gradient of the lost with respect to the input of this layer $\frac{\partial L}{\partial I^l}$ is one where $n > 0$ and zero otherwise.

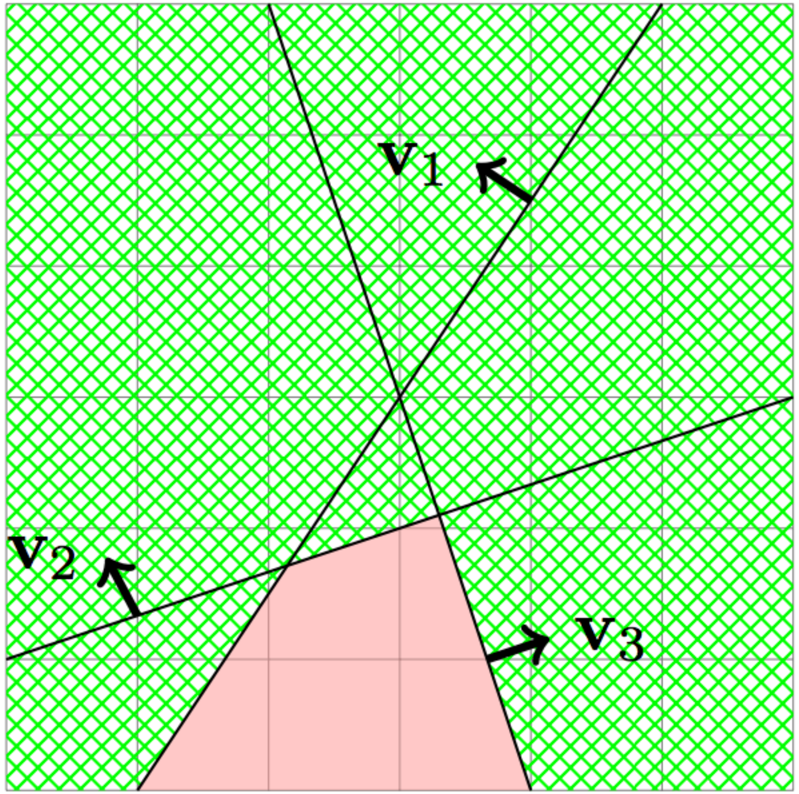
In [3]:
```
nextgrad=torch.ones(5)
relu:backward(n, nextgrad)
print(relu.gradInput)
```

Out[3]:
```
 0
 0
 1
 0
 1
[torch.DoubleTensor of size 5]
```

In [4]:
```
print(nextgrad)
```

Out[4]:
```
 1
 1
 1
 1
 1
[torch.DoubleTensor of size 5]
```

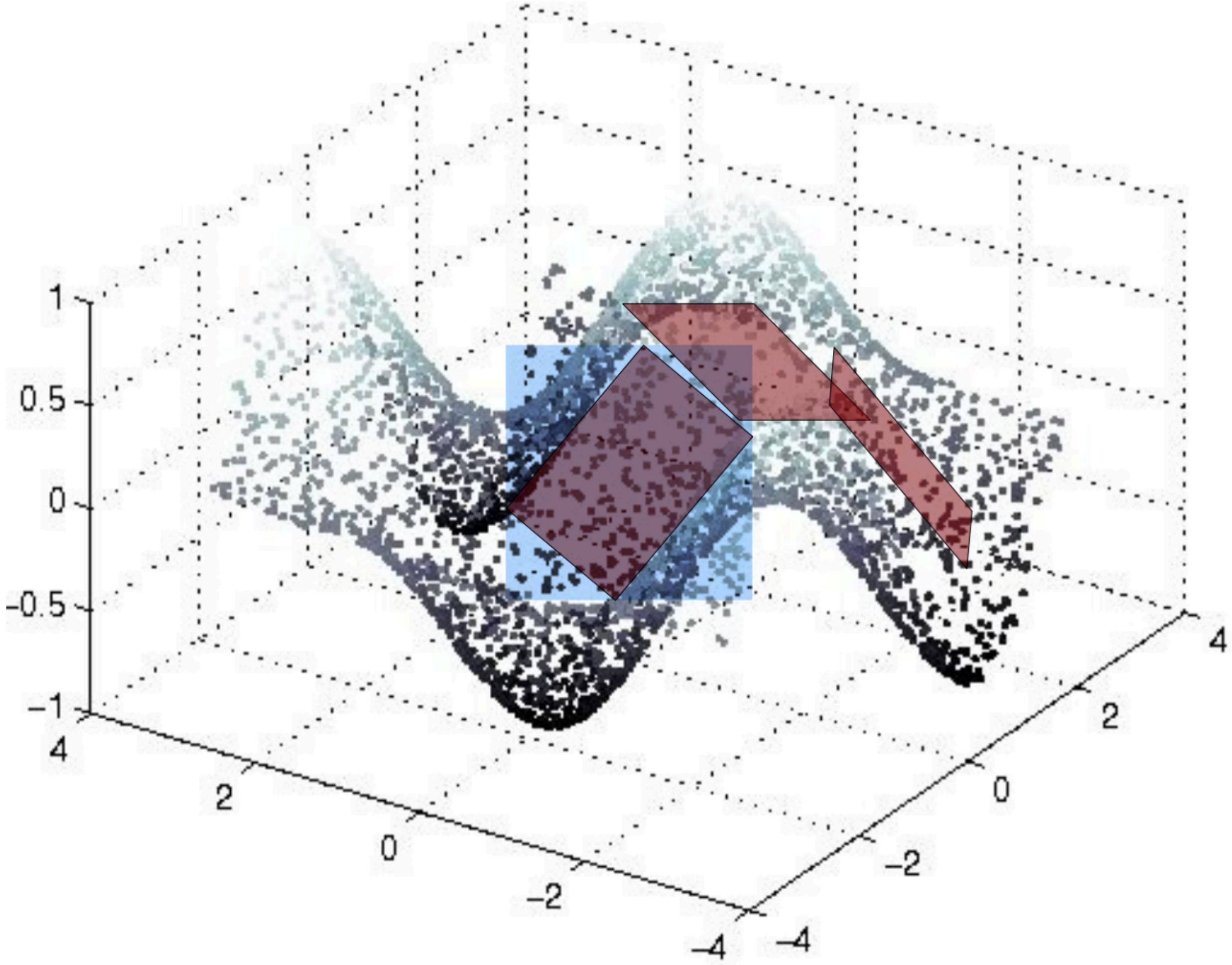
Building Blocks – ReLU



- Each hidden unit represents one hyperplane (parameterized by weight and bias) that bisects the input space into two half spaces.
- By choosing different weights in the hidden layer we can obtain arbitrary arrangement of n hyperplanes.
- The theory of hyperplane arrangement (Zaslavsky, 1975) tells us that for a general arrangement of n hyperplanes in d dimensions, the space is divided into $\sum_{s=0}^{d}\binom{n}{s}$ regions.

Expressiveness of Rectifier Networks
Xingyuan Pan, Vivek Srikumar

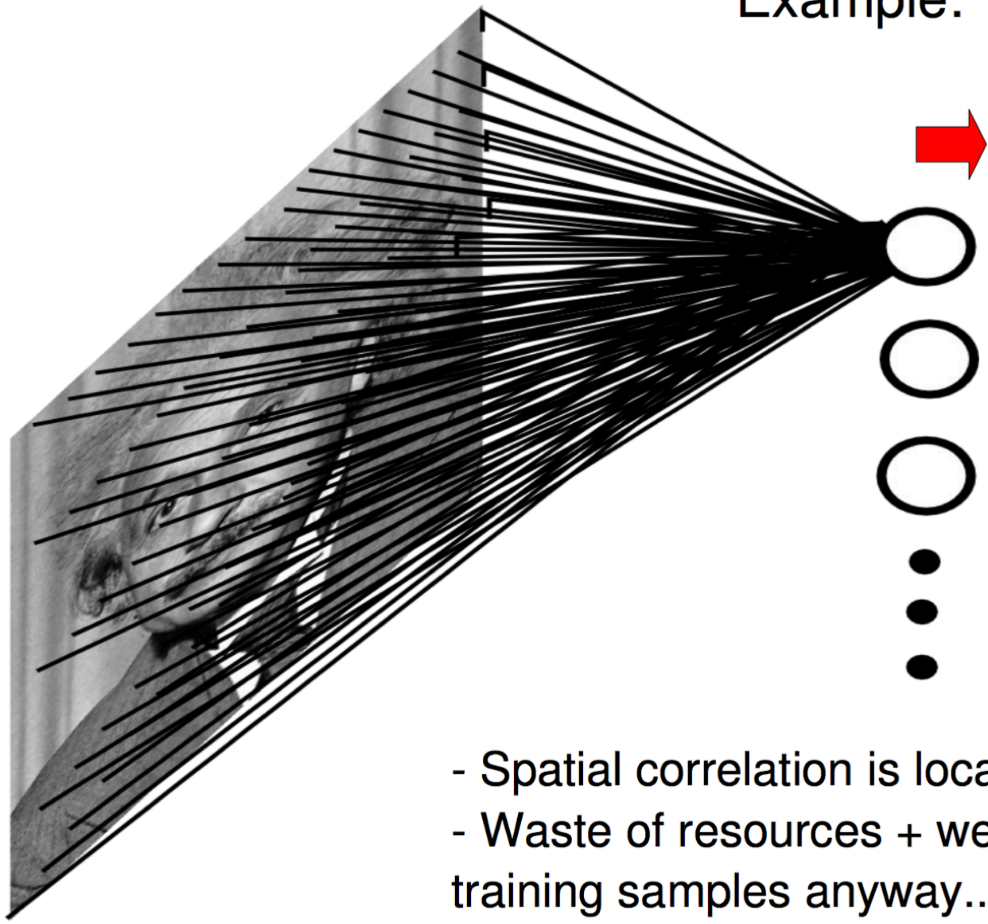
Building Blocks – ReLU



Building Blocks: **Convolution**

https://github.com/stencilman/CS763_Spring2017/blob/master/Notebooks/Convolution.ipynb

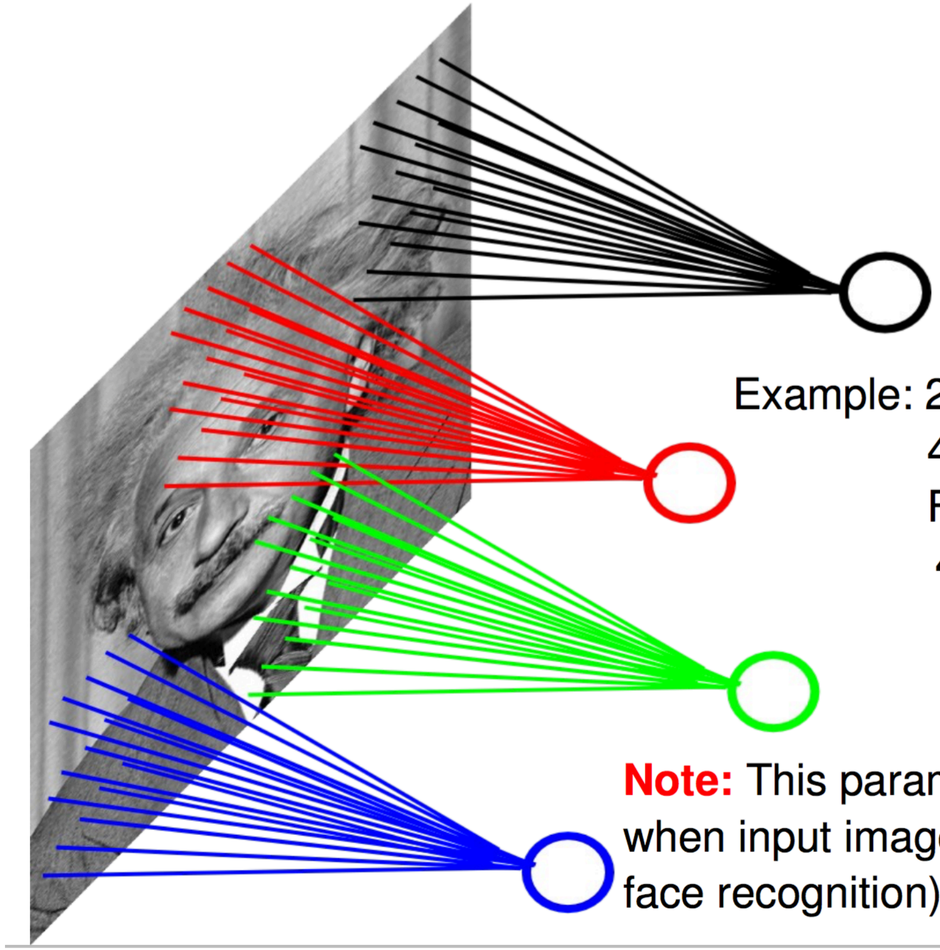
Building Blocks - Convolution



Example: 200x200 image
40K hidden units
~2B parameters!!!

- Spatial correlation is local
- Waste of resources + we have not enough training samples anyway..

Building Blocks - Convolution



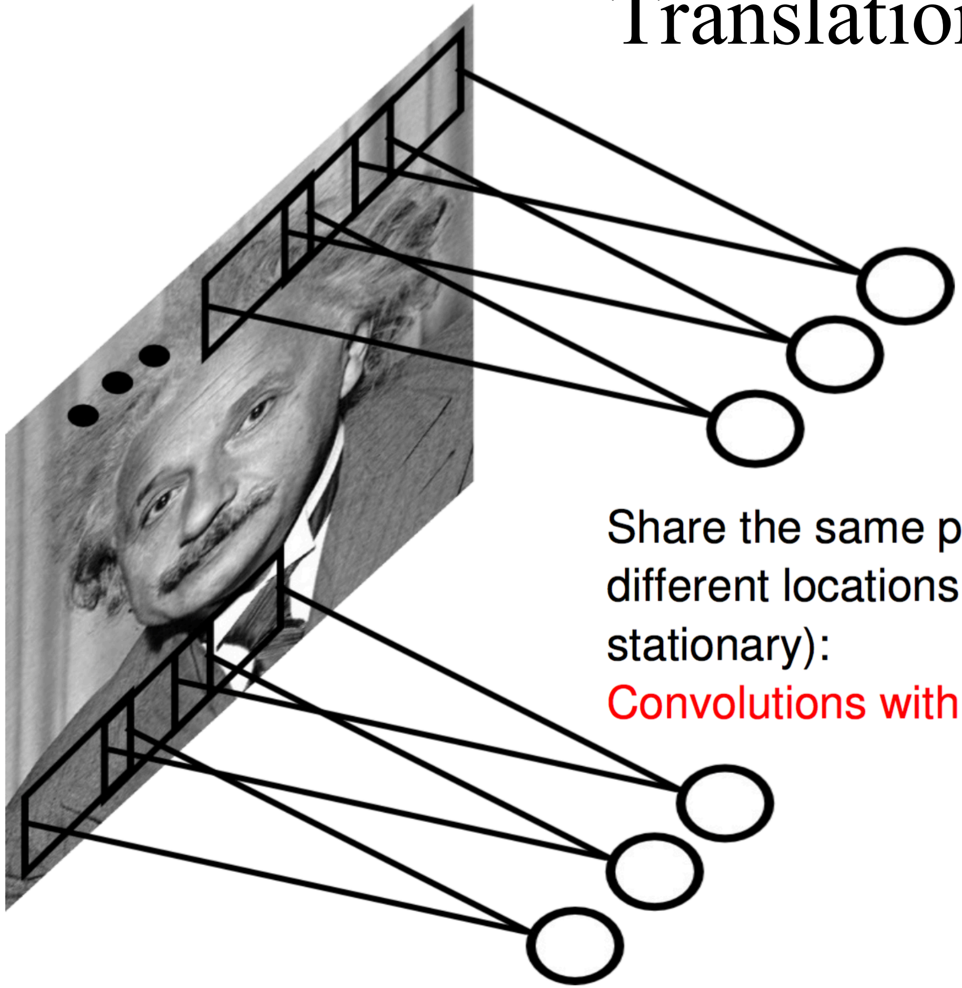
Example: 200x200 image
40K hidden units
Filter size: 10x10
4M parameters

Note: This parameterization is good when input image is registered (e.g., face recognition).

34

Building Blocks - Convolution

Translational Invariance

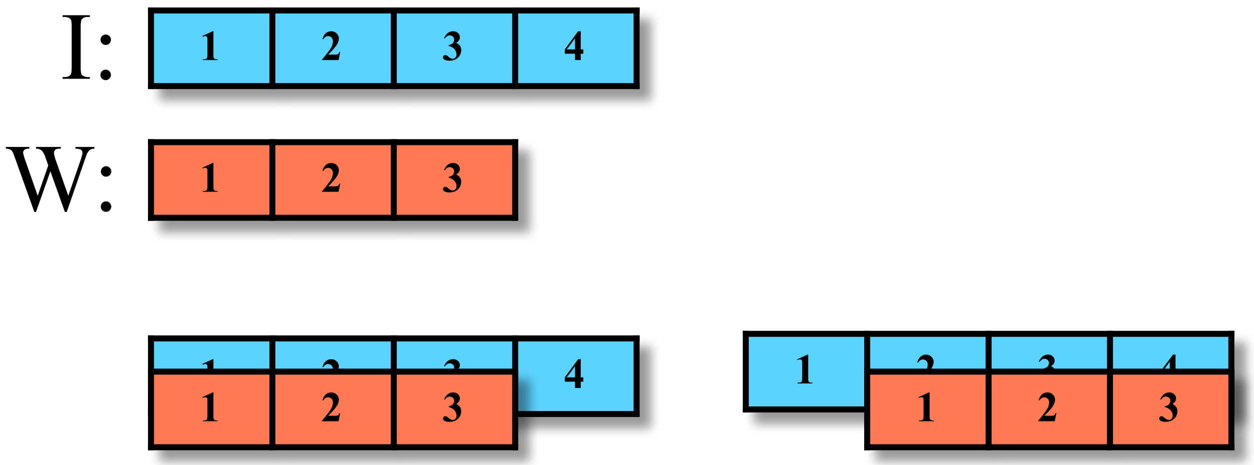


Share the same parameters across different locations (assuming input is stationary):
Convolutions with learned kernels

Building Blocks – Convolution (Discrete 1D)

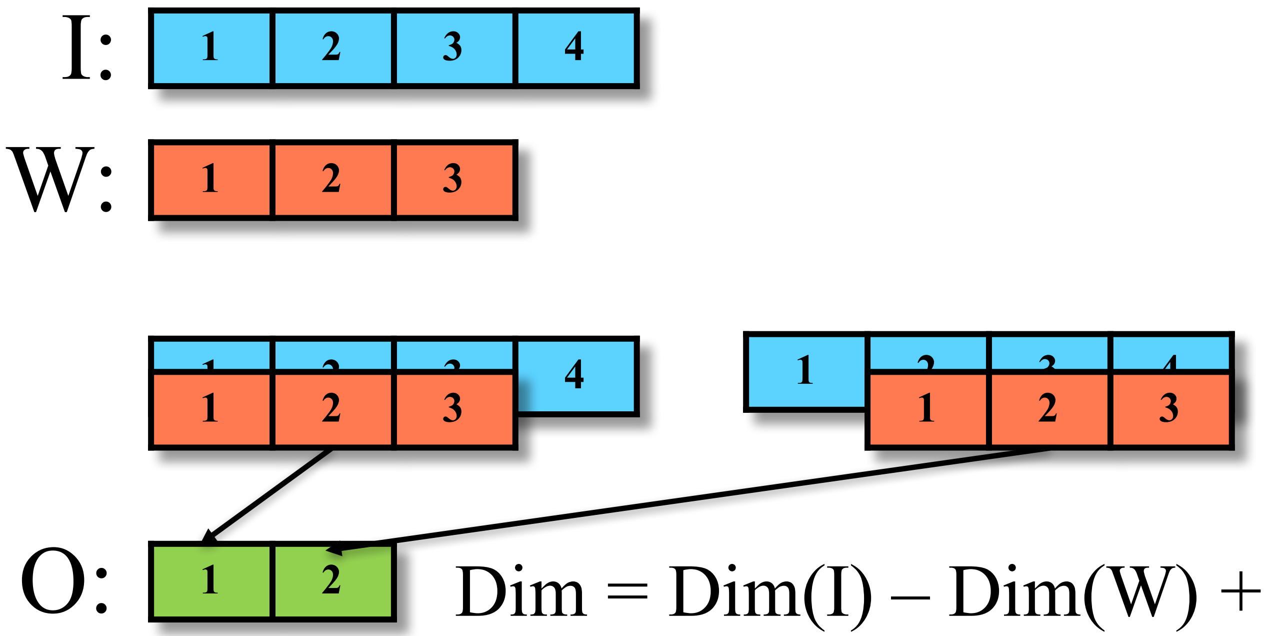
I:	1	2	3	4
W:	1	2	3	

Building Blocks – Convolution (Discrete 1D)



Slide

Building Blocks – Convolution (Discrete 1D)



Slide

Dim = Dim(I) – Dim(W) + 1

Correlation

$$O_1 = I_1 W_1 + I_2 W_2 + I_3 W_3$$

$$O_2 = I_2 W_1 + I_3 W_2 + I_4 W_3$$

$$O_i = \sum_{j=1}^{Dim(W)} I_{j+i-1} W_j$$

Building Blocks – Convolution (Discrete 1D)

I: | 1 | 2 | 3 | 4 |

W: | 1 | 2 | 3 |

W^{Flip}: | 1 | 2 | 3 | $=$ W: | 3 | 2 | 1 |

Slide

O′: | 1 | 2 |

Dim = Dim(I) – Dim(W) + 1

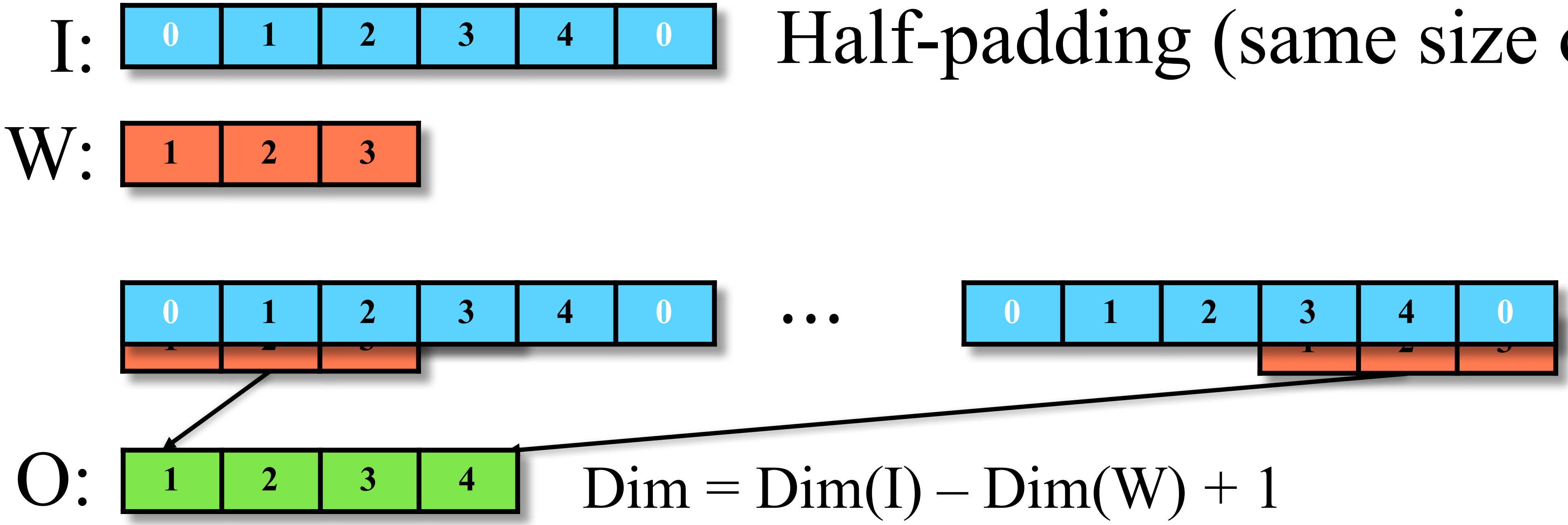
$$O'_1 = I_1 W_1^{Flip} + I_2 W_2^{Flip} + I_3 W_3^{Flip}$$

$$O'_2 = I_2 W_1^{Flip} + I_3 W_2^{Flip} + I_4 W_3^{Flip}$$

True Convolution

$$O_i = \sum_{j=1}^{Dim(W)} I_{j+i-1} W_{Dim(W)-j+1}$$

Building Blocks – Convolution (Discrete 1D)



Half-padding (same size output)

Dim = Dim(I) – Dim(W) + 1

Building Blocks – Convolution (Discrete 1D)

I: | 0 | 1 | 2 | 3 | 4 | 0 |

W: | 1 | 2 | 3 |

Half-padding, Stride = 2

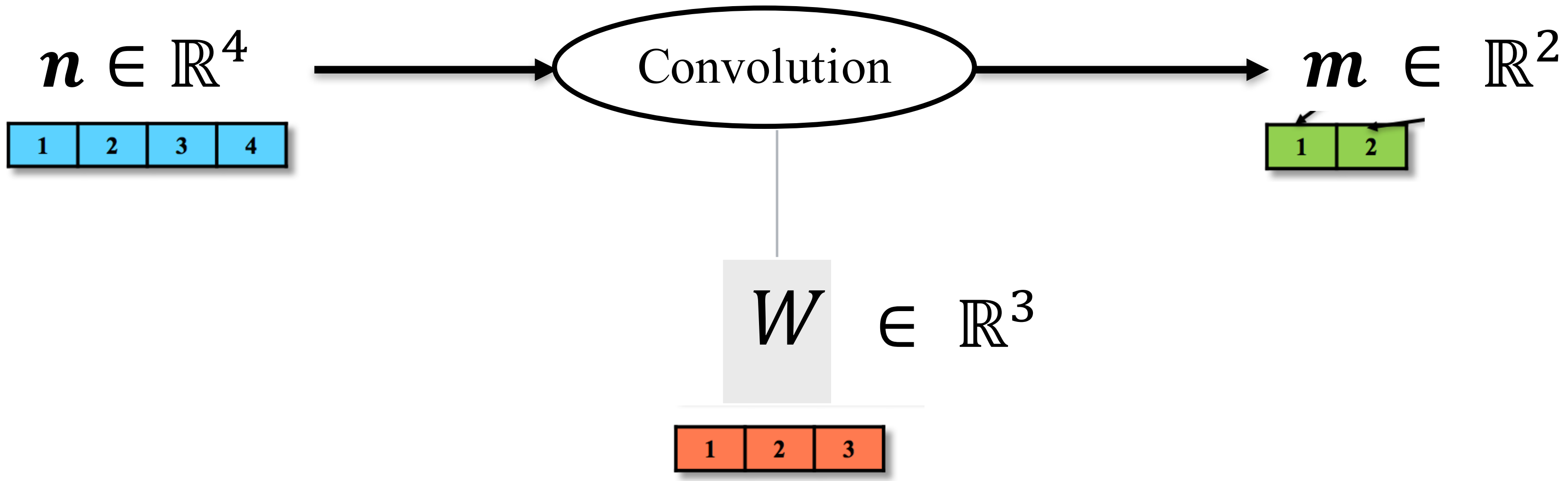
0	1	2	3	4	0
1	2	3			

0	1	2	3	4	0
		1	2	3	

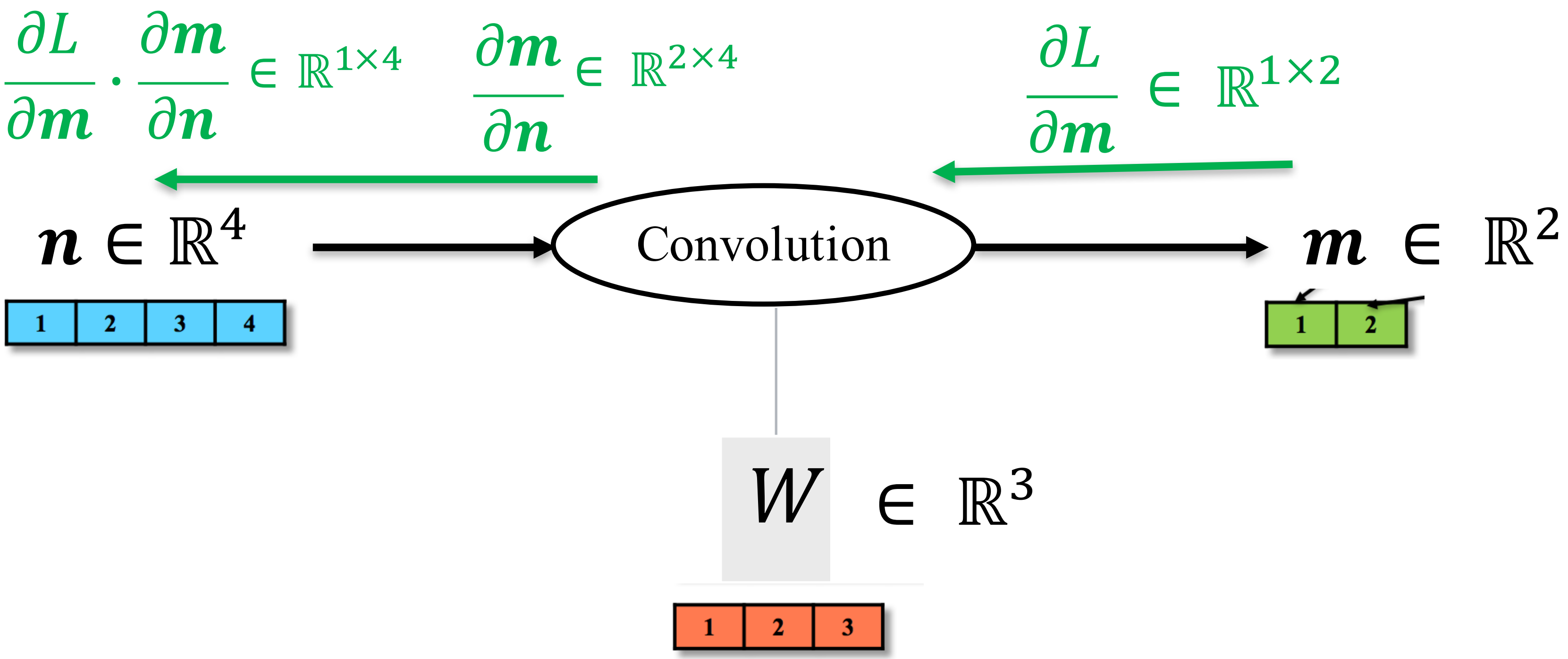
O: | 1 | 2 |

$$\text{Dim} = \left\lfloor \frac{\text{Dim(I)} - \text{Dim(W)}}{\text{Stride=2}} \right\rfloor + 1$$

Building Blocks – Convolution – Feed Forward

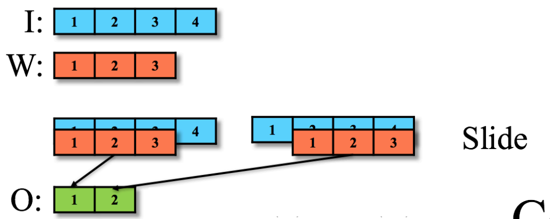


Building Blocks – Convolution – Backward



Building Blocks – Convolution – Backward

$$\frac{\partial \boldsymbol{O}}{\partial \boldsymbol{I}} = \begin{bmatrix} W_1 & W_2 & W_3 & 0 \\ 0 & W_1 & W_2 & W_3 \end{bmatrix}$$



Correlation

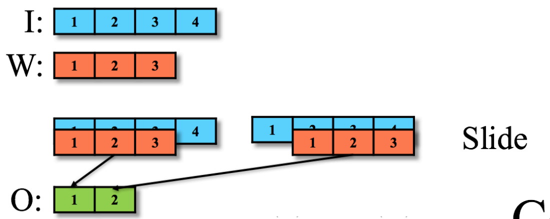
$$O_1 = I_1 W_1 + I_2 W_2 + I_3 W_3$$

$$O_2 = I_2 W_1 + I_3 W_2 + I_4 W_3$$

Building Blocks – Convolution – Backward

$$\frac{\partial \boldsymbol{O}}{\partial \boldsymbol{I}} = \begin{bmatrix} W_1 & W_2 & W_3 & 0 \\ 0 & W_1 & W_2 & W_3 \end{bmatrix}$$

$$\frac{\partial \boldsymbol{O}}{\partial \boldsymbol{W}} = \begin{bmatrix} I_1 & I_2 & I_3 \\ I_2 & I_3 & I_4 \end{bmatrix}$$



$O_1 = I_1W_1 + I_2W_2 + I_3W_3$

$O_2 = I_2W_1 + I_3W_2 + I_4W_3$

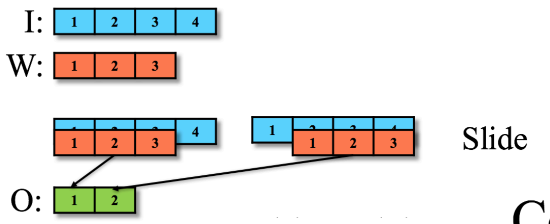
Correlation

Building Blocks – Convolution – Backward

$$\frac{\partial \boldsymbol{O}}{\partial \boldsymbol{I}} = \begin{bmatrix} W_1 & W_2 & W_3 & 0 \\ 0 & W_1 & W_2 & W_3 \end{bmatrix}$$

$$\frac{\partial \boldsymbol{O}}{\partial \boldsymbol{W}} = \begin{bmatrix} I_1 & I_2 & I_3 \\ I_2 & I_3 & I_4 \end{bmatrix}$$

$$\frac{\partial L}{\partial \boldsymbol{O}} = \begin{bmatrix} \partial LO_1 & \partial LO_2 \end{bmatrix}$$

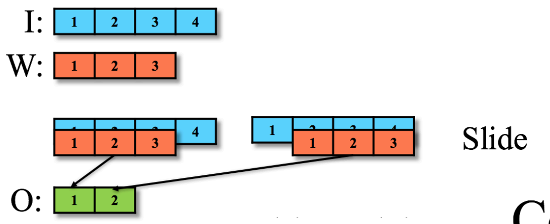


$O_1 = I_1W_1 + I_2W_2 + I_3W_3$

$O_2 = I_2W_1 + I_3W_2 + I_4W_3$

Correlation

Building Blocks – Convolution – Backward



$O_1 = I_1W_1 + I_2W_2 + I_3W_3$

$O_2 = I_2W_1 + I_3W_2 + I_4W_3$

Correlation

$$\frac{\partial \boldsymbol{O}}{\partial \boldsymbol{I}} = \begin{bmatrix} W_1 & W_2 & W_3 & 0 \\ 0 & W_1 & W_2 & W_3 \end{bmatrix}$$

$$\frac{\partial \boldsymbol{O}}{\partial \boldsymbol{W}} = \begin{bmatrix} I_1 & I_2 & I_3 \\ I_2 & I_3 & I_4 \end{bmatrix}$$

$$\frac{\partial L}{\partial \boldsymbol{O}} = [\partial LO_1 \quad \partial LO_2]$$

$$\frac{\partial L}{\partial \boldsymbol{W}} = \frac{\partial L}{\partial \boldsymbol{O}} \times \frac{\partial \boldsymbol{O}}{\partial \boldsymbol{W}} = [\partial LO_1 \times I_1 + \partial LO_2 \times I_2 \quad \partial LO_1 \times I_2 + \partial LO_2 \times I_3 \quad \partial LO_1 \times I_3 + \partial LO_2 \times I_4]$$

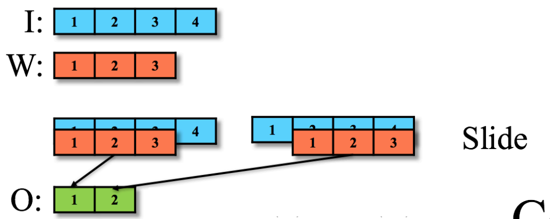
Building Blocks – Convolution – Backward

$$\frac{\partial \boldsymbol{O}}{\partial \boldsymbol{I}} = \begin{bmatrix} W_1 & W_2 & W_3 & 0 \\ 0 & W_1 & W_2 & W_3 \end{bmatrix}$$

$$\frac{\partial \boldsymbol{O}}{\partial \boldsymbol{W}} = \begin{bmatrix} I_1 & I_2 & I_3 \\ I_2 & I_3 & I_4 \end{bmatrix}$$

$$\frac{\partial L}{\partial \boldsymbol{O}} = \begin{bmatrix} \partial LO_1 & \partial LO_2 \end{bmatrix}$$

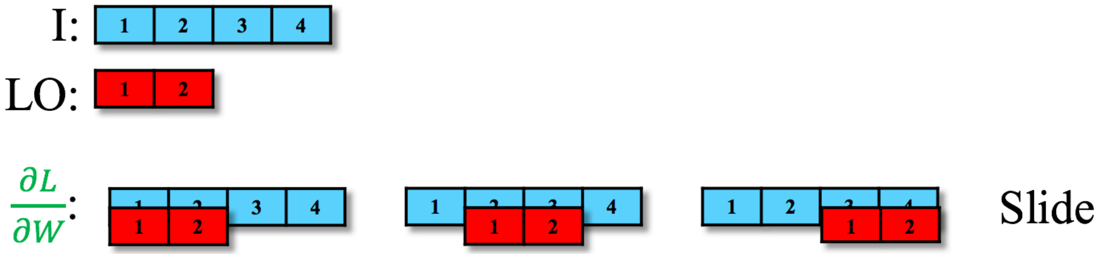
$$\frac{\partial L}{\partial \boldsymbol{W}} = \frac{\partial L}{\partial \boldsymbol{O}} \times \frac{\partial \boldsymbol{O}}{\partial \boldsymbol{W}} = \begin{bmatrix} \partial LO_1 \times I_1 + \partial LO_2 \times I_2 & \partial LO_1 \times I_2 + \partial LO_2 \times I_3 & \partial LO_1 \times I_3 + \partial LO_2 \times I_4 \end{bmatrix}$$



$$O_1 = I_1 W_1 + I_2 W_2 + I_3 W_3$$

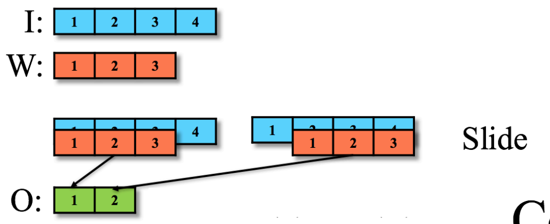
$$O_2 = I_2 W_1 + I_3 W_2 + I_4 W_3$$

Correlation



Building Blocks – Convolution – Backward

$$\frac{\partial \boldsymbol{O}}{\partial \boldsymbol{I}} = \begin{bmatrix} W_1 & W_2 & W_3 & 0 \\ 0 & W_1 & W_2 & W_3 \end{bmatrix}$$



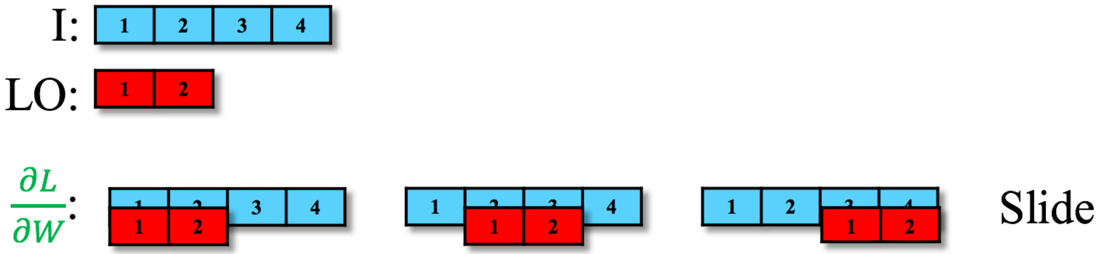
Correlation

$$O_1 = I_1 W_1 + I_2 W_2 + I_3 W_3$$
$$O_2 = I_2 W_1 + I_3 W_2 + I_4 W_3$$

$$\frac{\partial \boldsymbol{O}}{\partial \boldsymbol{W}} = \begin{bmatrix} I_1 & I_2 & I_3 \\ I_2 & I_3 & I_4 \end{bmatrix}$$

$$\frac{\partial L}{\partial \boldsymbol{O}} = [\partial LO_1 \quad \partial LO_2]$$

$$\frac{\partial L}{\partial \boldsymbol{W}} = \frac{\partial L}{\partial \boldsymbol{O}} \times \frac{\partial \boldsymbol{O}}{\partial \boldsymbol{W}} = [\partial LO_1 \times I_1 + \partial LO_2 \times I_2 \quad \partial LO_1 \times I_2 + \partial LO_2 \times I_3 \quad \partial LO_1 \times I_3 + \partial LO_2 \times I_4]$$



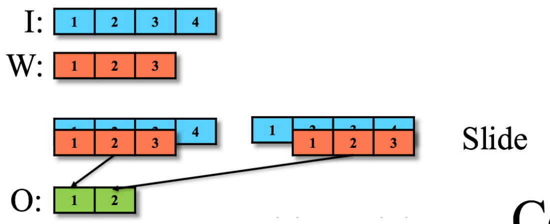
$$\frac{\partial L}{\partial \boldsymbol{W}} = \text{Correlation(I, LO)}$$

Building Blocks – Convolution – Backward

$$\frac{\partial \boldsymbol{O}}{\partial \boldsymbol{I}} = \begin{bmatrix} W_1 & W_2 & W_3 & 0 \\ 0 & W_1 & W_2 & W_3 \end{bmatrix}$$

$$\frac{\partial \boldsymbol{O}}{\partial \boldsymbol{W}} = \begin{bmatrix} I_1 & I_2 & I_3 \\ I_2 & I_3 & I_4 \end{bmatrix}$$

$$\frac{\partial L}{\partial \boldsymbol{O}} = \begin{bmatrix} \partial LO_1 & \partial LO_2 \end{bmatrix}$$

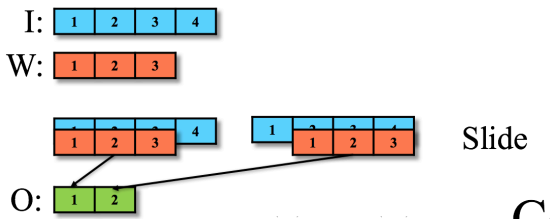


$O_1 = I_1W_1 + I_2W_2 + I_3W_3$

$O_2 = I_2W_1 + I_3W_2 + I_4W_3$

Correlation

Building Blocks – Convolution – Backward



Correlation

$$O_1 = I_1 W_1 + I_2 W_2 + I_3 W_3$$

$$O_2 = I_2 W_1 + I_3 W_2 + I_4 W_3$$

$$\frac{\partial \boldsymbol{O}}{\partial \boldsymbol{I}} = \begin{bmatrix} W_1 & W_2 & W_3 & 0 \\ 0 & W_1 & W_2 & W_3 \end{bmatrix}$$

$$\frac{\partial \boldsymbol{O}}{\partial \boldsymbol{W}} = \begin{bmatrix} I_1 & I_2 & I_3 \\ I_2 & I_3 & I_4 \end{bmatrix}$$

$$\frac{\partial L}{\partial \boldsymbol{O}} = \begin{bmatrix} \partial LO_1 & \partial LO_2 \end{bmatrix}$$

$$\frac{\partial L}{\partial \boldsymbol{I}} = \frac{\partial L}{\partial \boldsymbol{O}} \times \frac{\partial \boldsymbol{O}}{\partial \boldsymbol{I}} = \begin{bmatrix} \partial LO_1 \times W_1 & \partial LO_1 \times W_2 + \partial LO_2 \times W_1 & \partial LO_1 \times W_3 + \partial LO_2 \times W_2 & \partial LO_2 \times W_3 \end{bmatrix}$$

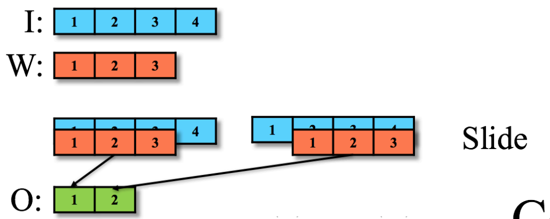
Building Blocks – Convolution – Backward

$$\frac{\partial \boldsymbol{O}}{\partial \boldsymbol{I}} = \begin{bmatrix} W_1 & W_2 & W_3 & 0 \\ 0 & W_1 & W_2 & W_3 \end{bmatrix}$$

$$\frac{\partial \boldsymbol{O}}{\partial \boldsymbol{W}} = \begin{bmatrix} I_1 & I_2 & I_3 \\ I_2 & I_3 & I_4 \end{bmatrix}$$

$$\frac{\partial L}{\partial \boldsymbol{O}} = [\partial LO_1 \quad \partial LO_2]$$

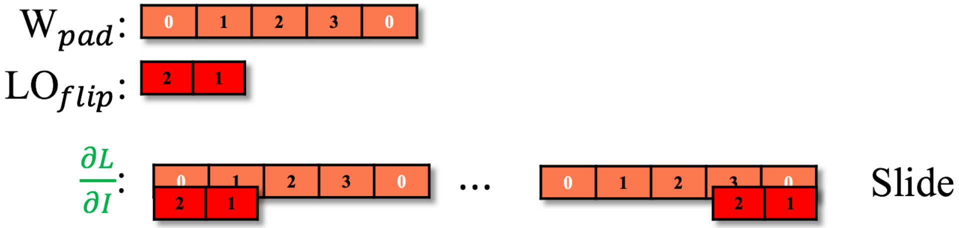
$$\frac{\partial L}{\partial \boldsymbol{I}} = \frac{\partial L}{\partial \boldsymbol{O}} \times \frac{\partial \boldsymbol{O}}{\partial \boldsymbol{I}} = [\partial LO_1 \times W_1 \quad \partial LO_1 \times W_2 + \partial LO_2 \times W_1 \quad \partial LO_1 \times W_3 + \partial LO_2 \times W_2 \quad \partial LO_2 \times W_3]$$



$O_1 = I_1W_1 + I_2W_2 + I_3W_3$

$O_2 = I_2W_1 + I_3W_2 + I_4W_3$

Correlation

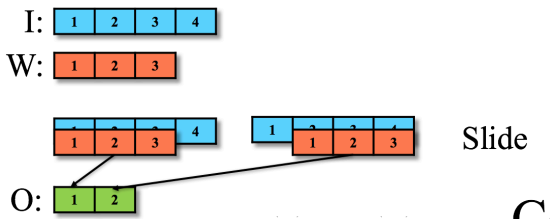


Building Blocks – Convolution – Backward

$$\frac{\partial \boldsymbol{O}}{\partial \boldsymbol{I}} = \begin{bmatrix} W_1 & W_2 & W_3 & 0 \\ 0 & W_1 & W_2 & W_3 \end{bmatrix}$$

$$\frac{\partial \boldsymbol{O}}{\partial \boldsymbol{W}} = \begin{bmatrix} I_1 & I_2 & I_3 \\ I_2 & I_3 & I_4 \end{bmatrix}$$

$$\frac{\partial L}{\partial \boldsymbol{O}} = \begin{bmatrix} \partial LO_1 & \partial LO_2 \end{bmatrix}$$

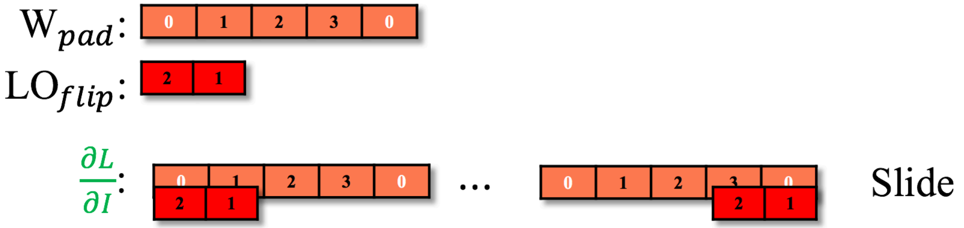


$$O_1 = I_1W_1 + I_2W_2 + I_3W_3$$

$$O_2 = I_2W_1 + I_3W_2 + I_4W_3$$

Correlation

$$\frac{\partial L}{\partial \boldsymbol{I}} = \frac{\partial L}{\partial \boldsymbol{O}} \times \frac{\partial \boldsymbol{O}}{\partial \boldsymbol{I}} = \begin{bmatrix} \partial LO_1 \times W_1 & \partial LO_1 \times W_2 + \partial LO_2 \times W_1 & \partial LO_1 \times W_3 + \partial LO_2 \times W_2 & \partial LO_2 \times W_3 \end{bmatrix}$$



$$\frac{\partial L}{\partial \boldsymbol{I}} = \text{Correlation}(W_{pad}, \text{LO}_{flip})$$

Building Blocks – Convolution – in Torch7

Convolution Layer

In this notebook, we will look into the forward and the backward the the `nn.SpatialConvolution` layer. We will also see how to compute the gradient of the loss with respect to the weights $\frac{\partial L}{\partial W}$ for this layer. I leave gradient of the loss with respect to the input $\frac{\partial L}{\partial I}$ as an excercise.

Input

Similar to the example we used in the class, let us have the input n of size 1×4.

In [11]:
```
require 'nn';
n = torch.rand(4):reshape(1,1,4)
print(n)
```

Out[11]:
```
(1,.,.) =
  0.3347  0.5901  0.7132  0.3187
[torch.DoubleTensor of size 1x1x4]
```

Output using Convolution Block

Also similar to the example we used in the class, let us create a convolution block with a weights w of size 1×3 and obtain the output $m = Convolution(n, w)$ of size 1×2.

In [12]:
```
conv = nn.SpatialConvolutionMM(1,1,3,1)
conv.bias:fill(0)
m = conv:forward(n)
print(m)
```

Out[12]:
```
(1,.,.) =
  0.1897  0.1130
[torch.DoubleTensor of size 1x1x2]
```

Building Blocks – Convolution – in Torch7

Doing backward and calculating the gradinent of the loss with respect to the weights

Let us set the gradient of the loss with respect to the input of next layer (flowing in through the next layer) $\frac{\partial L}{\partial I^{l+1}}$ to be random numbers. After the `backward()` call, let us print the $\frac{\partial L}{\partial W}$ as calcuated by torch.

In [13]:
```
nextgrad=torch.rand(2):reshape(1,1,2)
conv:backward(n, nextgrad)
print(conv.gradWeight)
```

Out[13]:
```
 0.6464  0.8428  0.5257
[torch.DoubleTensor of size 1x3]
```

Expression for calcuating the gradinent of the loss with respect to the weights

Again, as we learnt in class, the $\frac{\partial L}{\partial W} = Convolution(I, \frac{\partial L}{\partial I^{l+1}})$. We varify that this is indeed exactly equal to $\frac{\partial L}{\partial W}$ computed by torch. I leave the calculation of $\frac{\partial L}{\partial I}$ as an exercise. Remember: $\frac{\partial L}{\partial I} = Convolution(W^{padded}, {\frac{\partial L}{\partial I^{l+1}}}^{Flip})$

In [20]:
```
convback = nn.SpatialConvolutionMM(1,1,2,1)
convback.bias:fill(0)
convback.weight:copy(nextgrad:reshape(1,2))
gradWeight = convback:forward(n)
print(gradWeight)
```

Out[20]:
```
(1,.,.) =
  0.6464  0.8428  0.5257
[torch.DoubleTensor of size 1x1x3]
```

Building Blocks – Convolution – in Torch7

Convolution Layer

In this notebook, we will look into the forward and the backward the the `nn.SpatialConvolution` layer. We will also see how to compute the gradient of the loss with respect to the weights $\frac{\partial L}{\partial W}$ for this layer. I leave gradient of the loss with respect to the input $\frac{\partial L}{\partial I}$ as an excercise.

Input

Similar to the example we used in the class, let us have the input n of size 1×4.

In [11]:
```
require 'nn';
n = torch.rand(4):reshape(1,1,4)
print(n)
```

Out[11]:
```
(1,.,.) =
  0.3347  0.5901  0.7132  0.3187
[torch.DoubleTensor of size 1x1x4]
```

Output using Convolution Block

Also similar to the example we used in the class, let us create a convolution block with a weights w of size 1×3 and obtain the output $m = Convolution(n, w)$ of size 1×2.

In [12]:
```
conv = nn.SpatialConvolutionMM(1,1,3,1)
conv.bias:fill(0)
m = conv:forward(n)
print(m)
```

Out[12]:
```
(1,.,.) =
  0.1897  0.1130
[torch.DoubleTensor of size 1x1x2]
```

Building Blocks – Convolution – in Torch7

Doing backward and calculating the gradinent of the loss with respect to the weights

Let us set the gradient of the loss with respect to the input of next layer (flowing in through the next layer) $\frac{\partial L}{\partial I^{l+1}}$ to be random numbers. After the `backward()` call, let us print the $\frac{\partial L}{\partial W}$ as calcuated by torch.

In [13]:
```
nextgrad=torch.rand(2):reshape(1,1,2)
conv:backward(n, nextgrad)
print(conv.gradWeight)
```

Out[13]:
```
(1,.,.) =
  0.0781  0.1019 -0.2249  0.2193
[torch.DoubleTensor of size 1x1x4]

 0.6464  0.8428  0.5257
[torch.DoubleTensor of size 1x3]
```

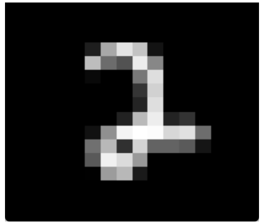
Expression for calcuating the gradinent of the loss with respect to the weights

Again, as we learnt in class, the $\frac{\partial L}{\partial W} = Convolution(I, \frac{\partial L}{\partial I^{l+1}})$. We varify that this is indeed exactly equal to $\frac{\partial L}{\partial W}$ computed by torch. I leave the calculation of $\frac{\partial L}{\partial I}$ as an exercise. Remember: $\frac{\partial L}{\partial I} = Convolution(W^{padded}, \frac{\partial L}{\partial I^{l+1}}^{Flip})$

In [20]:
```
convback = nn.SpatialConvolutionMM(1,1,2,1)
convback.bias:fill(0)
convback.weight:copy(nextgrad:reshape(1,2))
gradWeight = convback:forward(n)
print(gradWeight)
```

Out[20]:
```
(1,.,.) =
  0.6464  0.8428  0.5257
[torch.DoubleTensor of size 1x1x3]
```

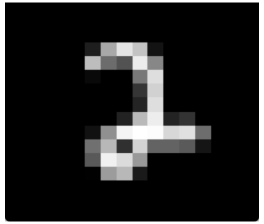
Building Blocks - Convolution



$\boldsymbol{n} \in \mathbb{R}^{16\times16}$ → Convolution → $\boldsymbol{m} \in \mathbb{R}^{8\times12\times12}$

$W \in \mathbb{R}^{1\times8\times5\times5}$

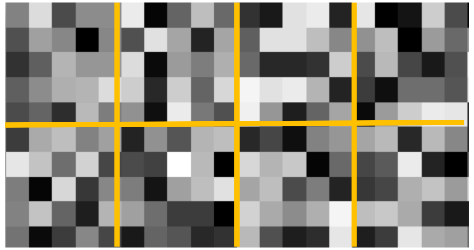
Building Blocks - Convolution



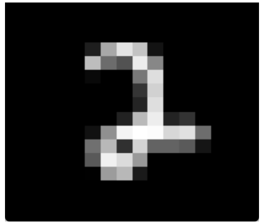
$\boldsymbol{n} \in \mathbb{R}^{16\times16} \rightarrow$ Convolution $\rightarrow \boldsymbol{m} \in \mathbb{R}^{8\times12\times12}$

$W \in \mathbb{R}^{1\times8\times5\times5}$

$=$



Building Blocks - Convolution



$\boldsymbol{n} \in \mathbb{R}^{16\times16} \rightarrow$ Convolution $\rightarrow \boldsymbol{m} \in \mathbb{R}^{8\times12\times12}$

W

```
> conv = nn.SpatialConvolutionMM(1, 8, 5, 5)
> m = conv:forward(n)
> =n:size()
  1
 16
 16
[torch.LongStorage of size 3]

> =m:size()
  8
 12
 12
[torch.LongStorage of size 3]
```

```
> =conv.weight:size()
  8
 25
[torch.LongStorage of size 2]

> =conv.bias:size()
 8
[torch.LongStorage of size 1]
```

Building Blocks - Convolution

1	-2	2	2
2	1	3	-2
-2	3	-3	1
-1	2	-4	2

$\boldsymbol{n} \in \mathbb{R}^{2\times4\times4}$ → Convolution → $\boldsymbol{m} \in \mathbb{R}^{2\times3\times3}$

-3.1	-3.9	-10.9
4.1	-12.9	16.1
-3.9	.1	9.1

$W \in \mathbb{R}^{2\times2\times2\times2}$

Building Blocks - Convolution

Image I = 2 x 4 x 4

I[1, :, :]

1	-2	2	2
2	1	3	-2
-2	3	-3	1
-1	2	-4	2

I[2, :, :]

3	0	0	0
-2	-2	1	-1
2	-1	3	1
5	-2	0	1

Weights W = 2 x 2 x 2 x 2
(nOutputPlane x nInputPlane x kH x kW)

W[1, 1, :, :]

1	-2
-2	1

W[2, 1, :, :]

3	1
2	2

W[1, 2, :, :]

1	0
0	1

W[2, 2, :, :]

0	0
0	4

0.1
0.2

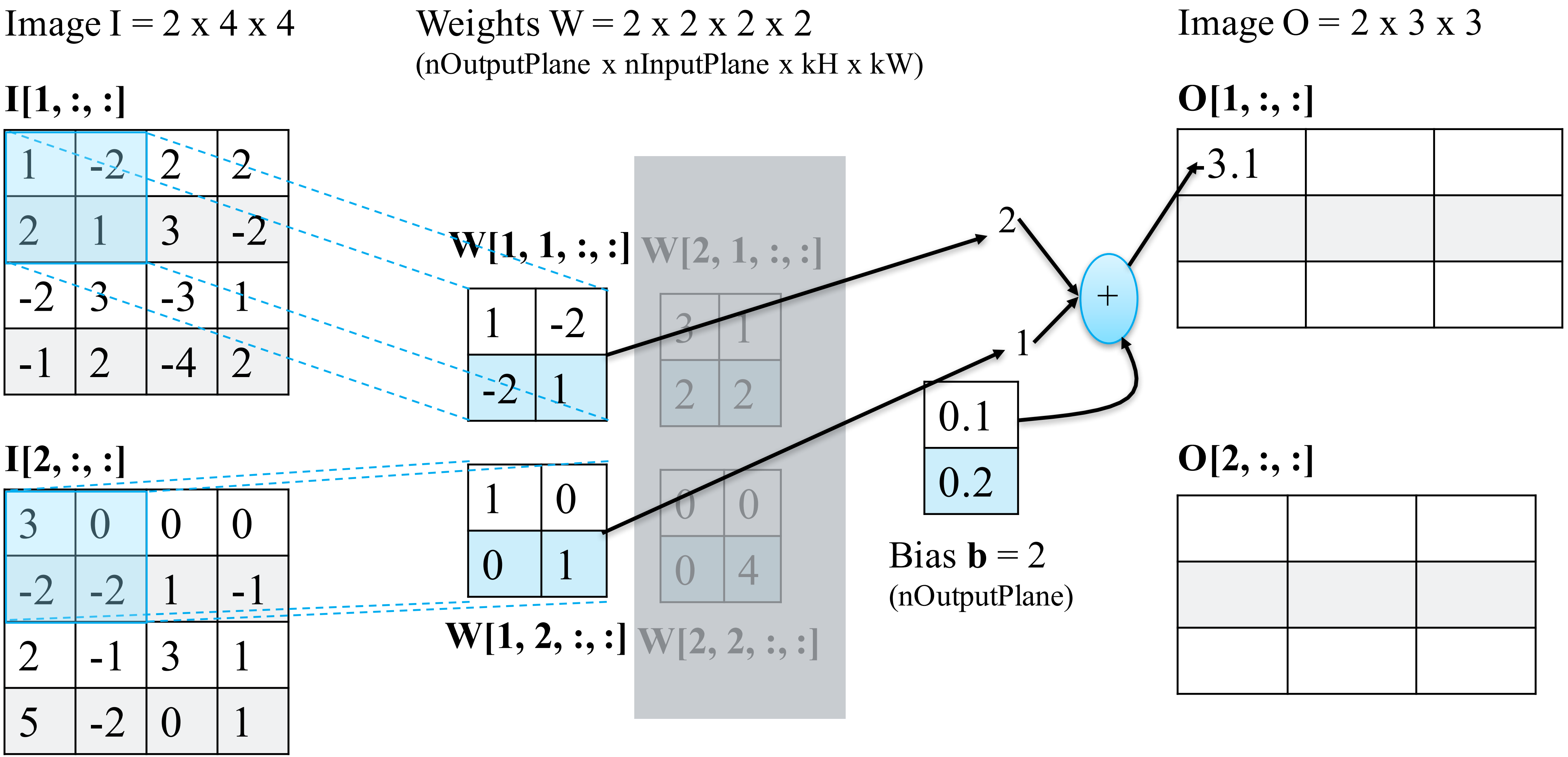
Bias **b** = 2
(nOutputPlane)

Image O = 2 x 3 x 3

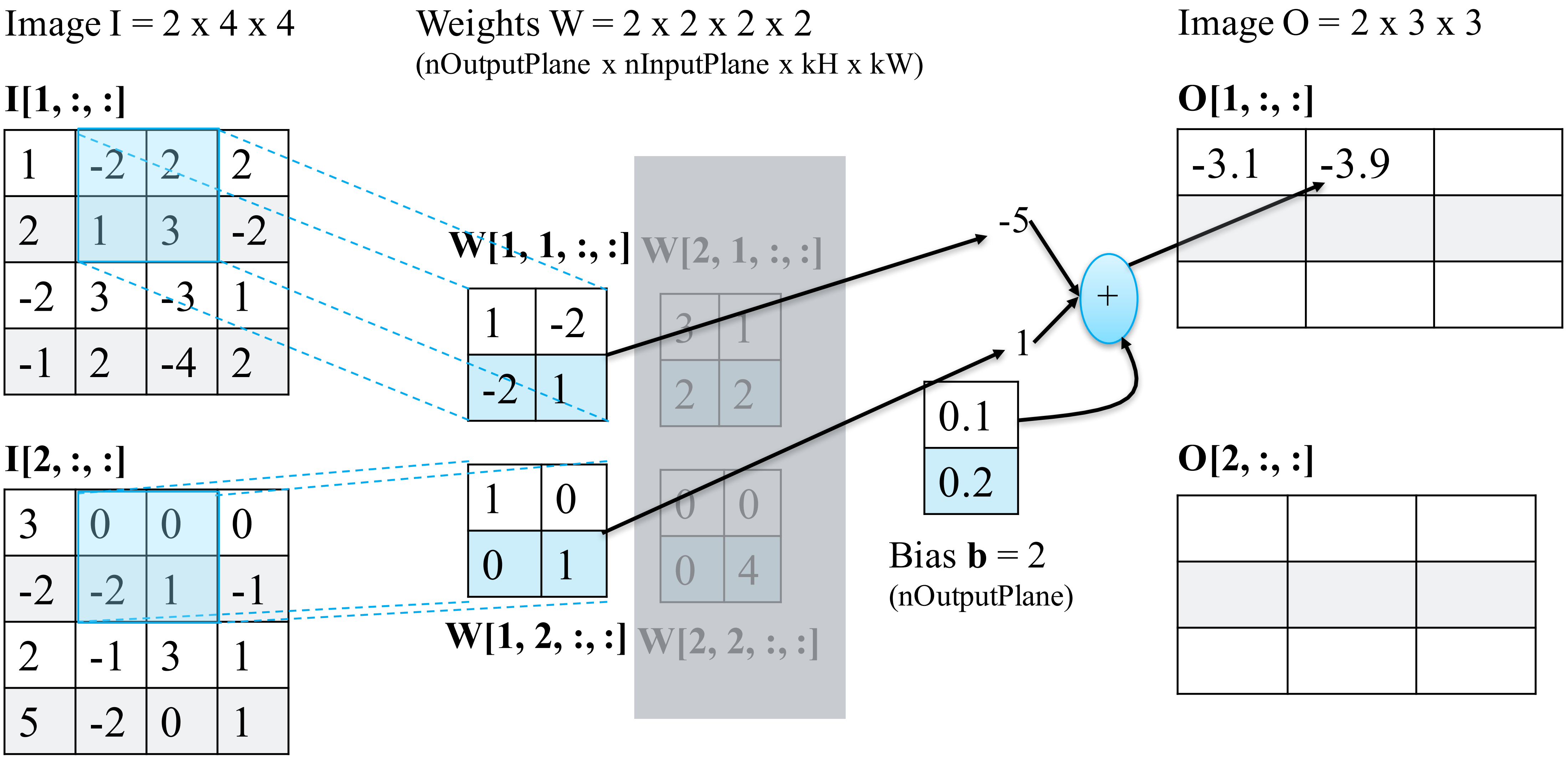
O[1, :, :]

O[2, :, :]

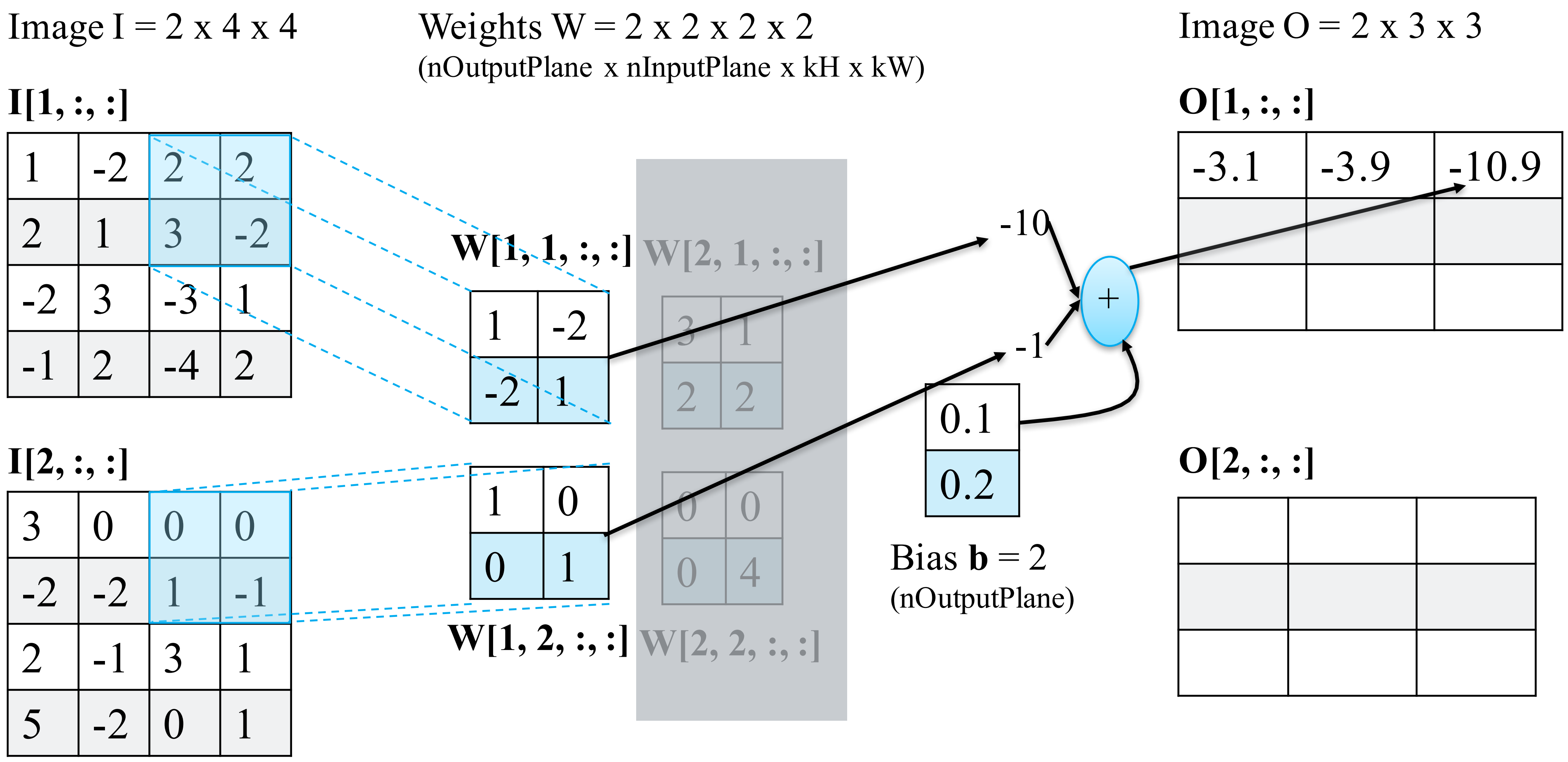
Building Blocks - Convolution



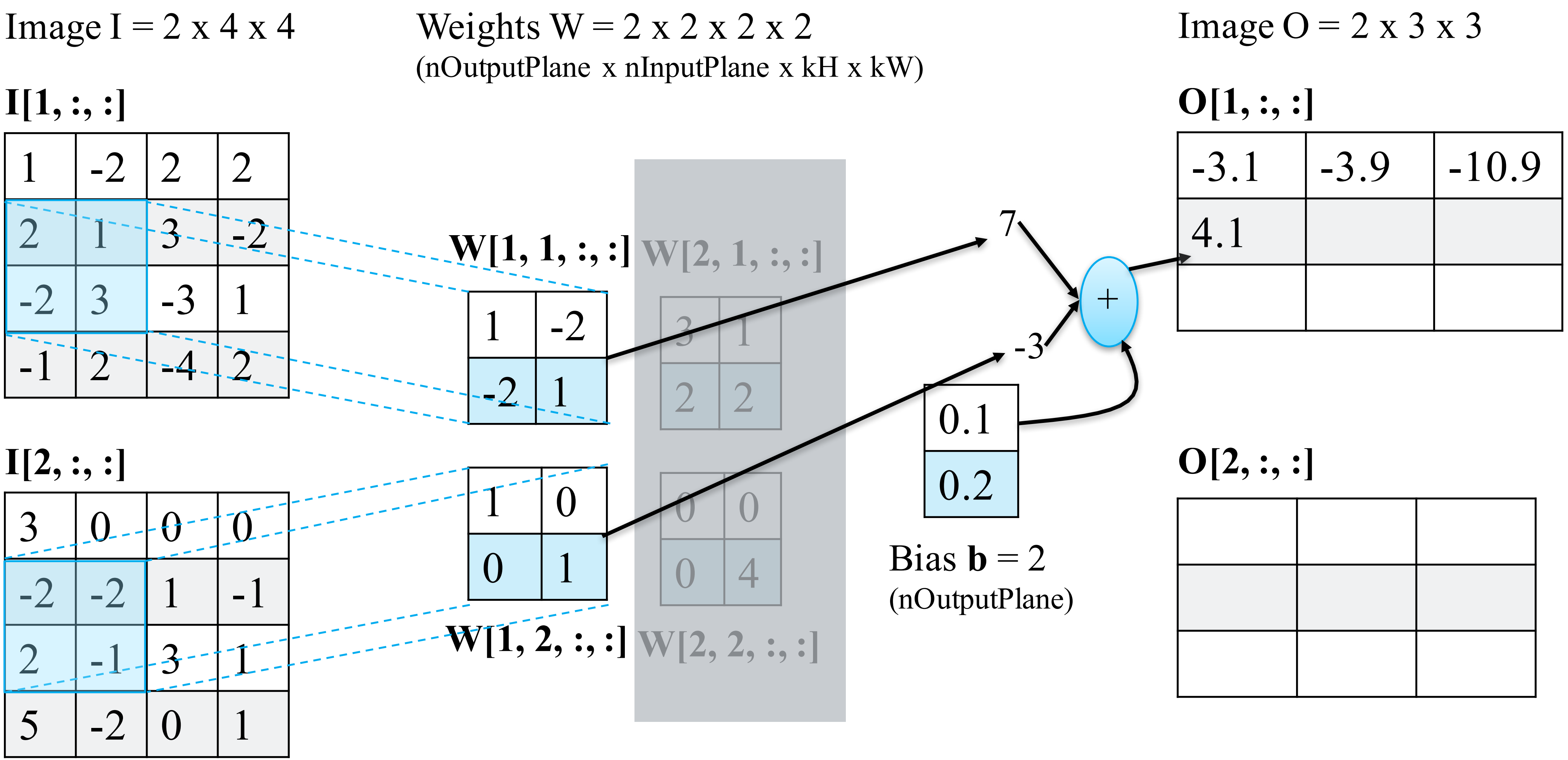
Building Blocks - Convolution



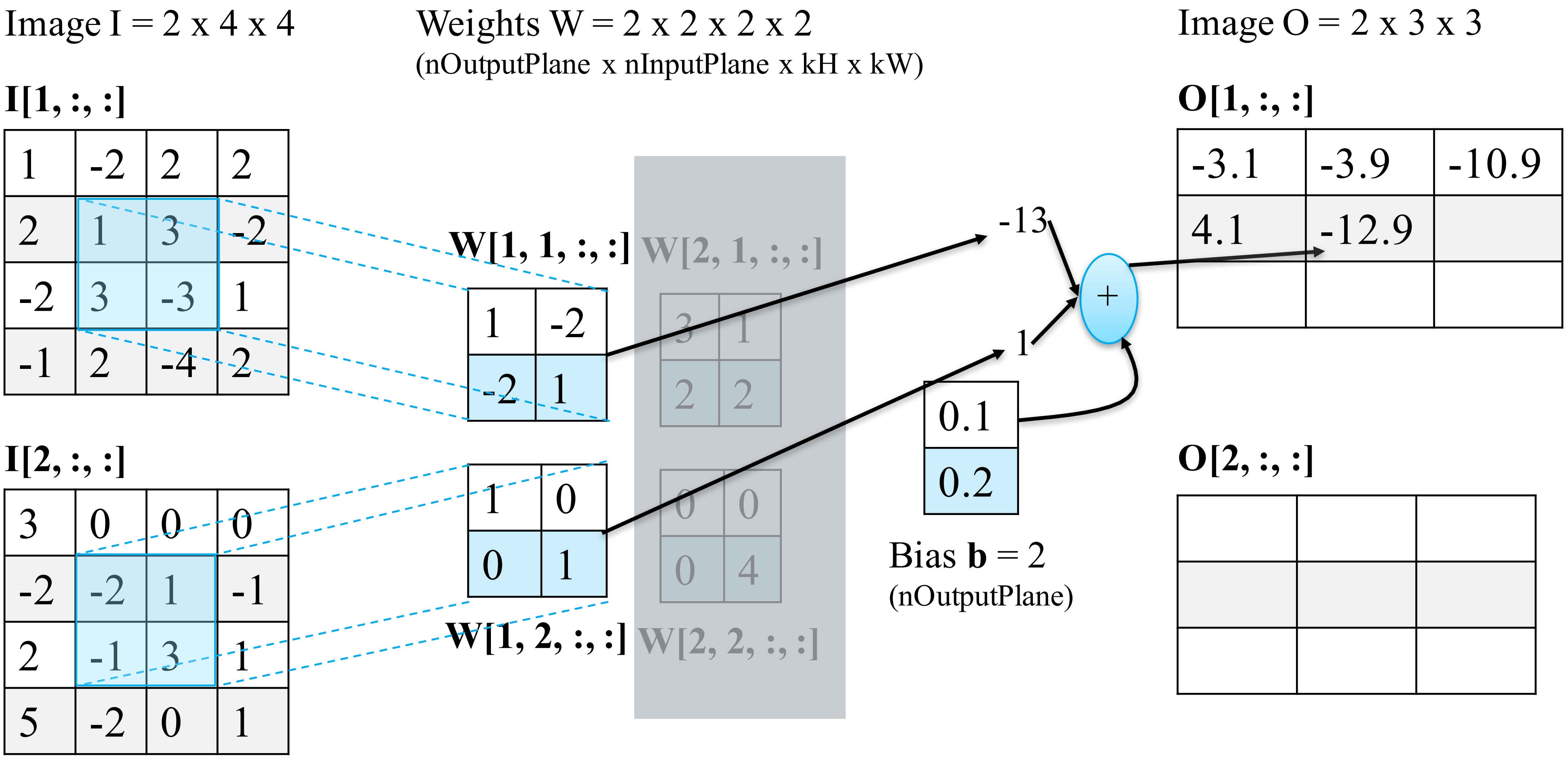
Building Blocks - Convolution



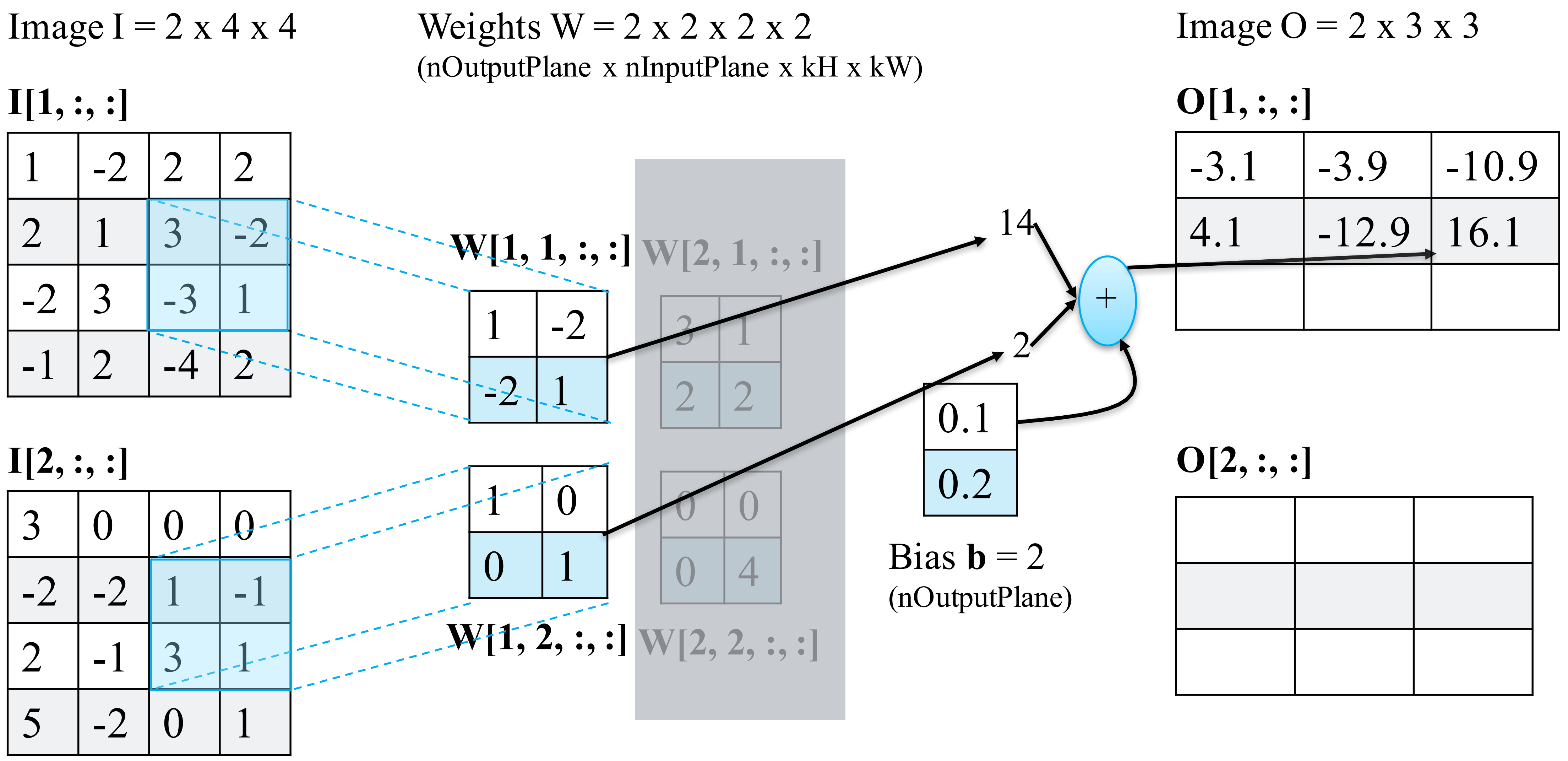
Building Blocks - Convolution



Building Blocks - Convolution



Building Blocks - Convolution



Building Blocks - Convolution

Image I = 2 x 4 x 4

I[1, :, :]

1	-2	2	2
2	1	3	-2
-2	3	-3	1
-1	2	-4	2

I[2, :, :]

3	0	0	0
-2	-2	1	-1
2	-1	3	1
5	-2	0	1

Weights W = 2 x 2 x 2 x 2
(nOutputPlane x nInputPlane x kH x kW)

W[1, 1, :, :]

1	-2
-2	1

W[2, 1, :, :]

3	1
2	2

W[1, 2, :, :]

1	0
0	1

W[2, 2, :, :]

0	0
0	4

-4

0

+

0.1
0.2

Bias **b** = 2
(nOutputPlane)

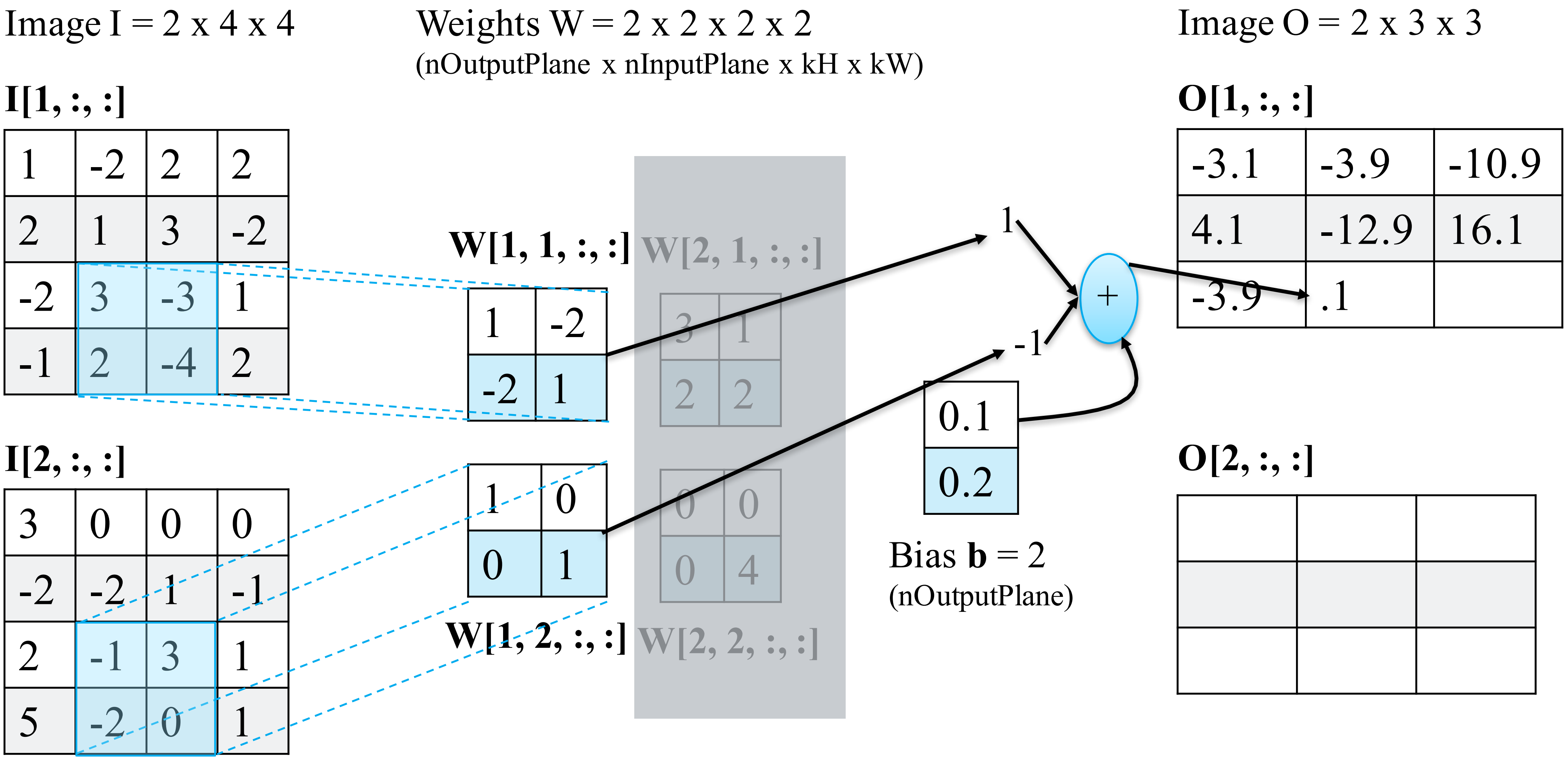
Image O = 2 x 3 x 3

O[1, :, :]

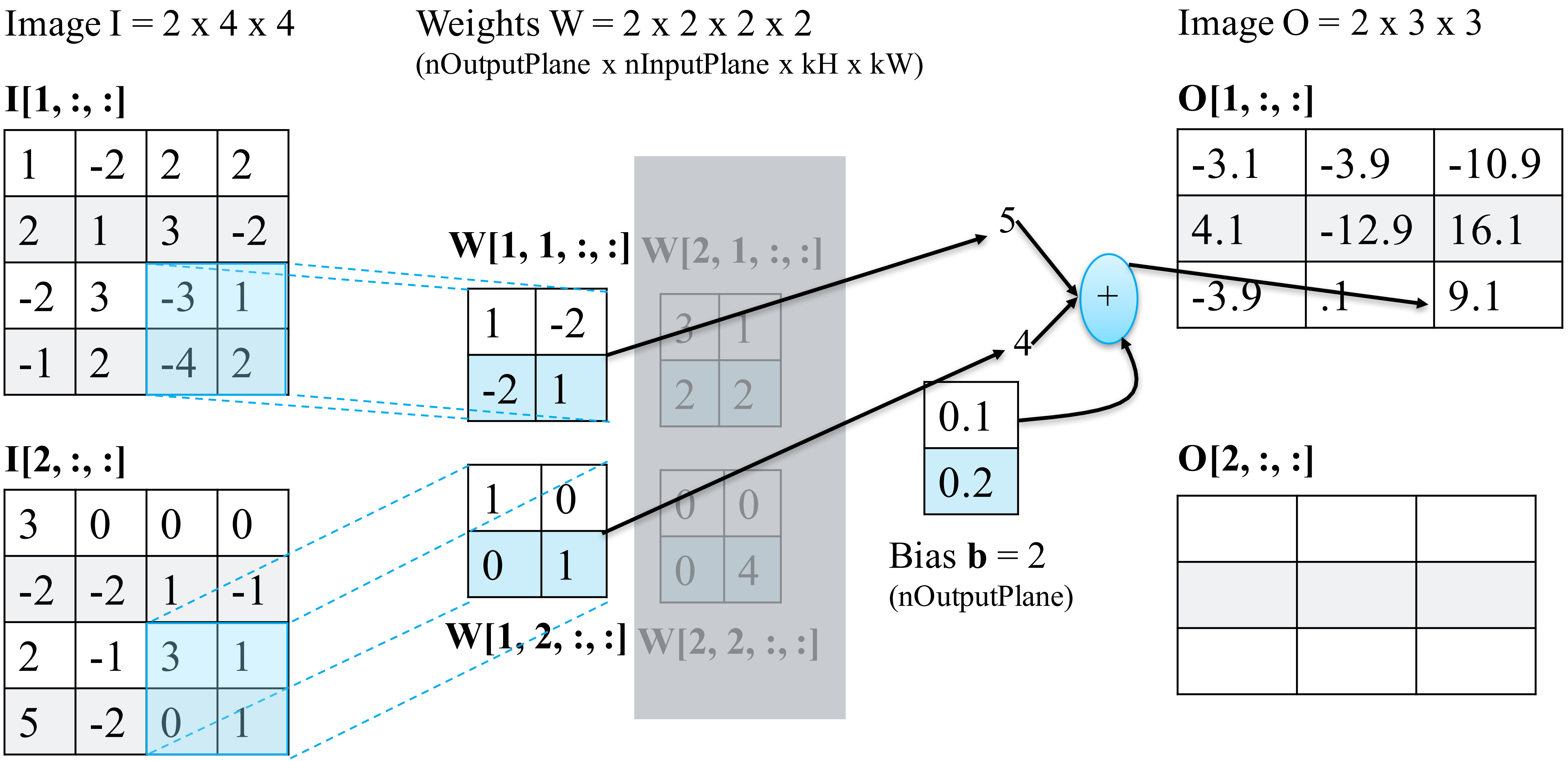
-3.1	-3.9	-10.9
4.1	-12.9	16.1
-3.9		

O[2, :, :]

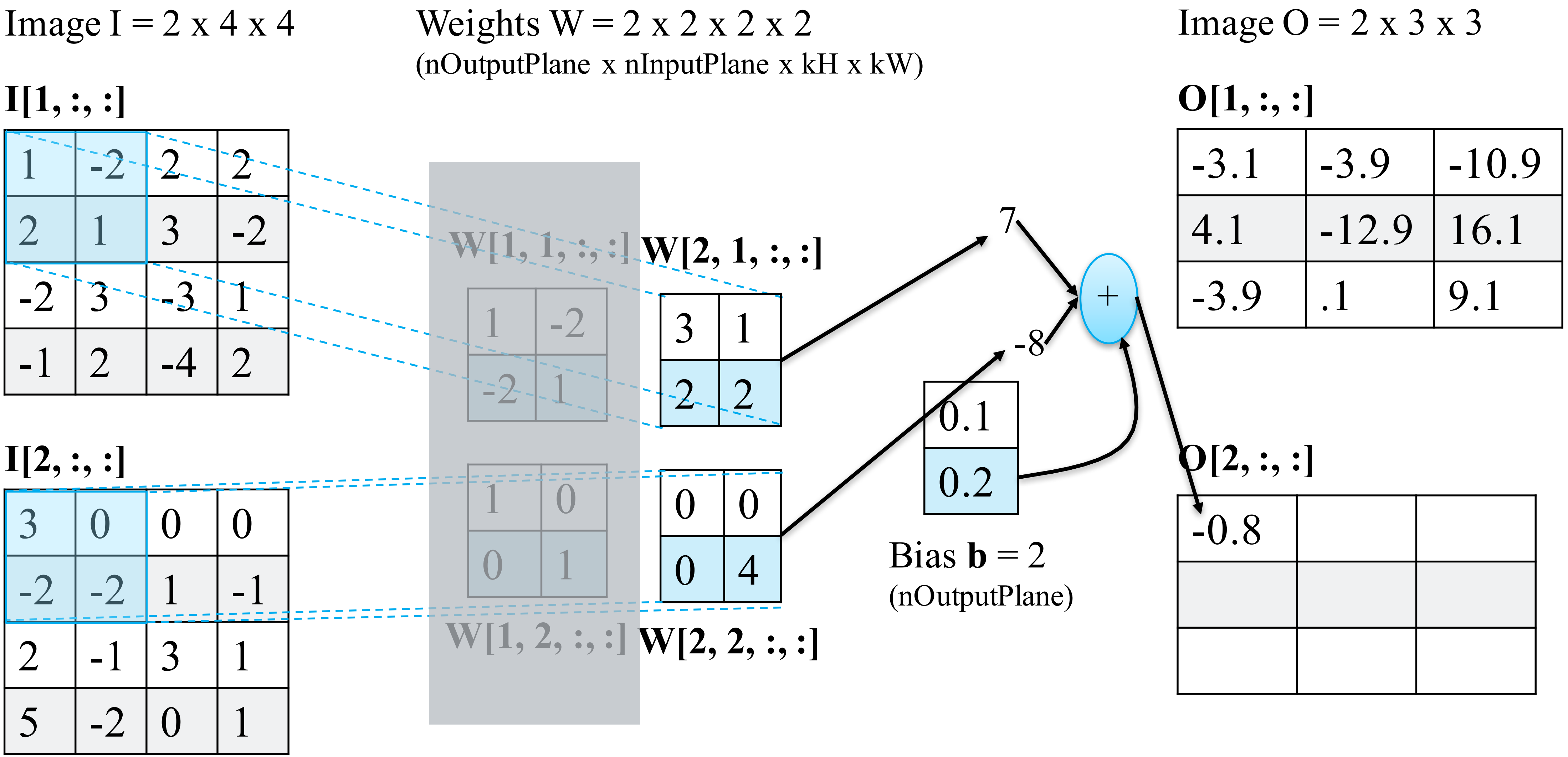
Building Blocks - Convolution



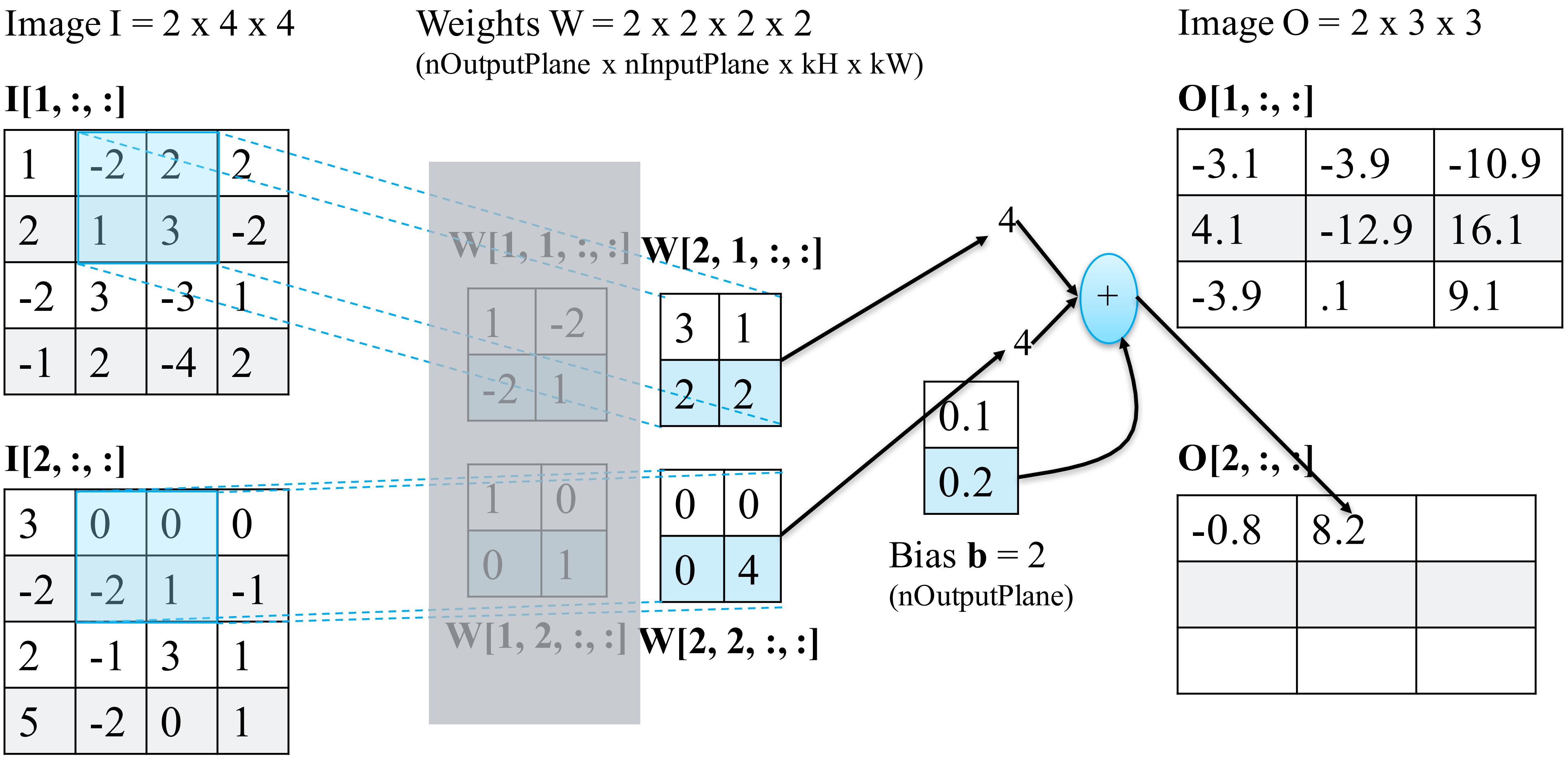
Building Blocks - Convolution



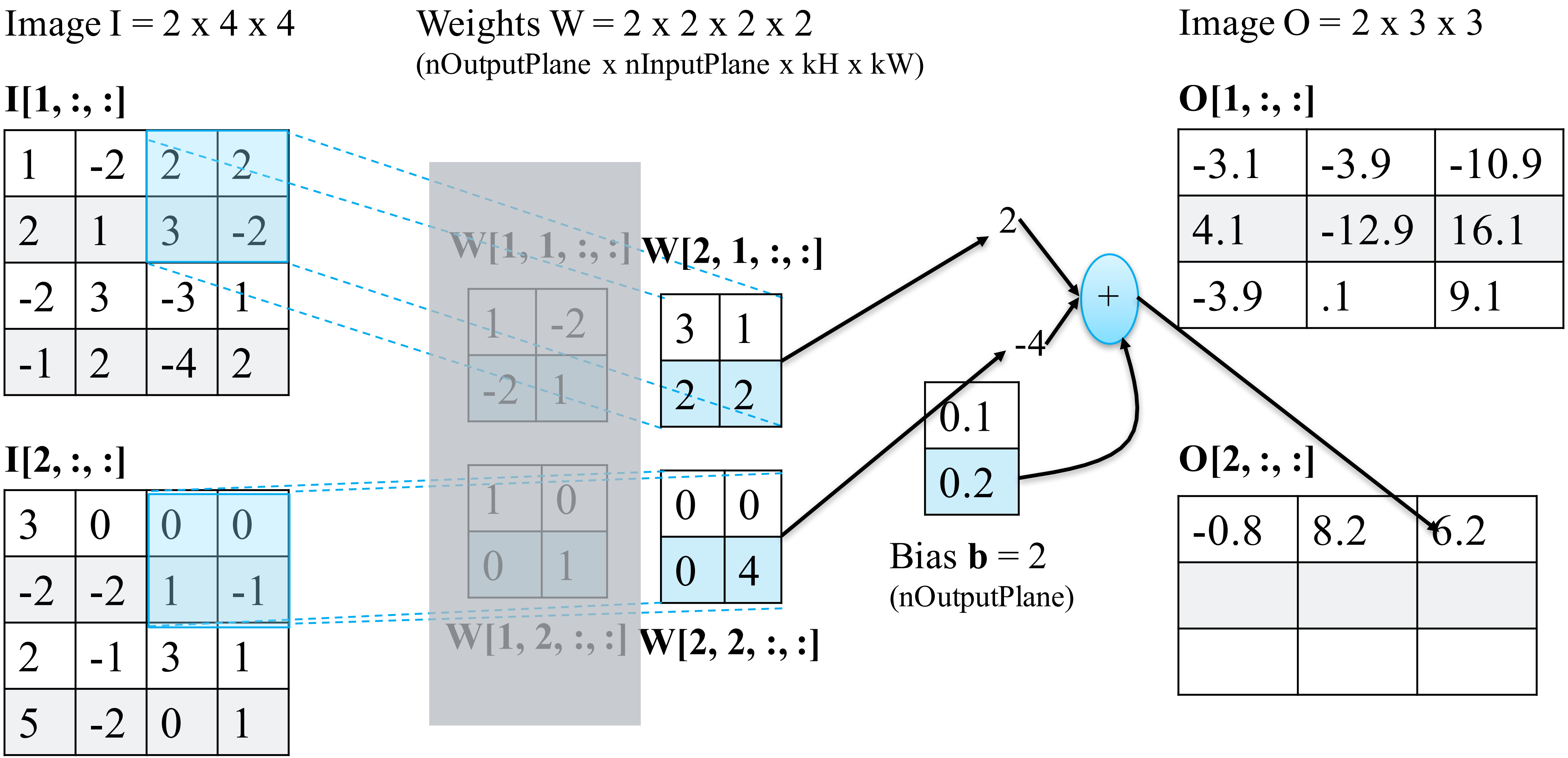
Building Blocks - Convolution



Building Blocks - Convolution



Building Blocks - Convolution



Building Blocks - Convolution

Image I = 2 x 4 x 4

I[1, :, :]

1	-2	2	2
2	1	3	-2
-2	3	-3	1
-1	2	-4	2

I[2, :, :]

3	0	0	0
-2	-2	1	-1
2	-1	3	1
5	-2	0	1

Weights W = 2 x 2 x 2 x 2
(nOutputPlane x nInputPlane x kH x kW)

W[1, 1, :, :]

1	-2
-2	1

W[1, 2, :, :]

1	0
0	1

W[2, 1, :, :]

3	1
2	2

W[2, 2, :, :]

0	0
0	4

9

-4

+

0.1
0.2

Bias **b** = 2
(nOutputPlane)

Image O = 2 x 3 x 3

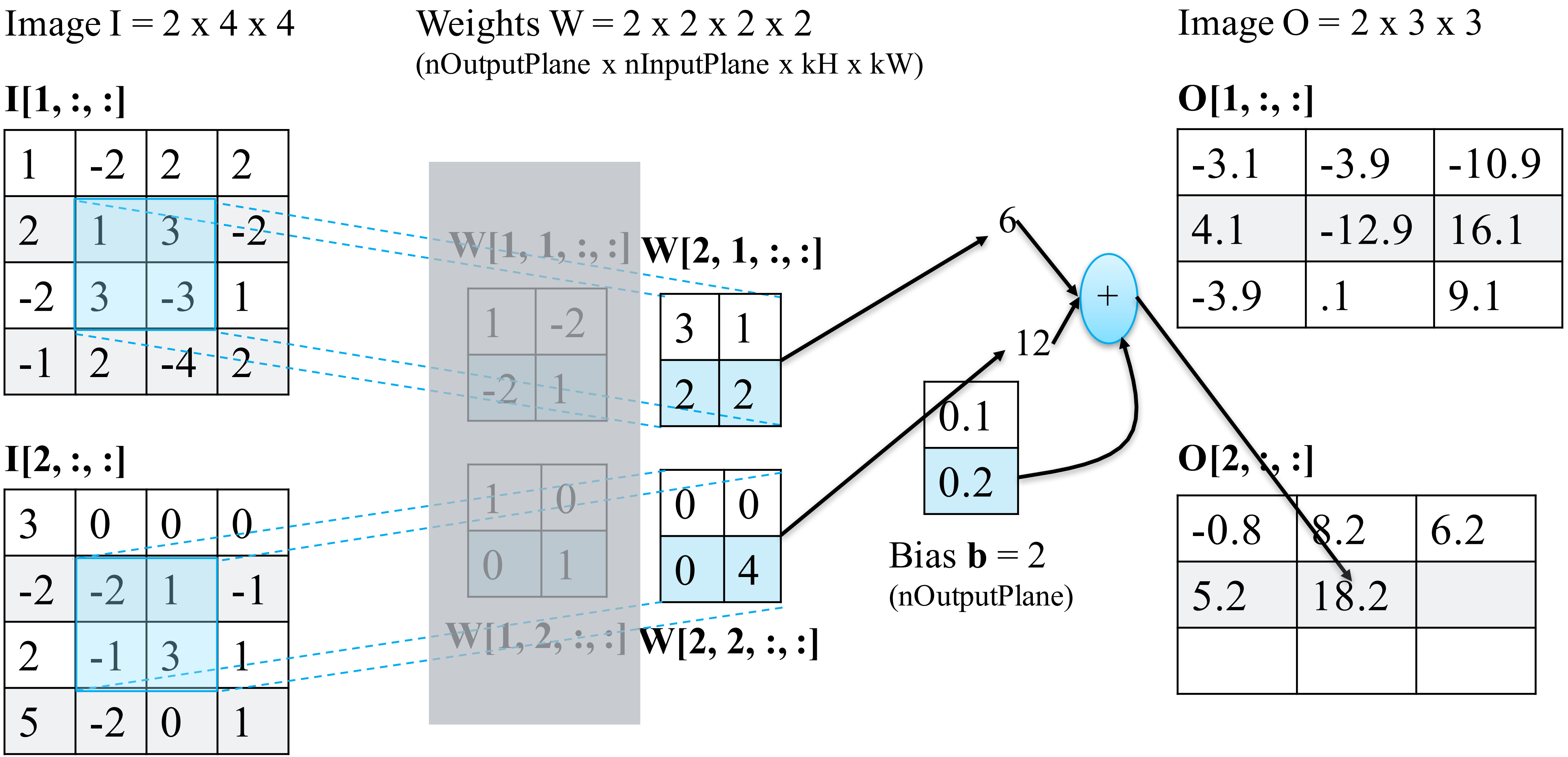
O[1, :, :]

-3.1	-3.9	-10.9
4.1	-12.9	16.1
-3.9	.1	9.1

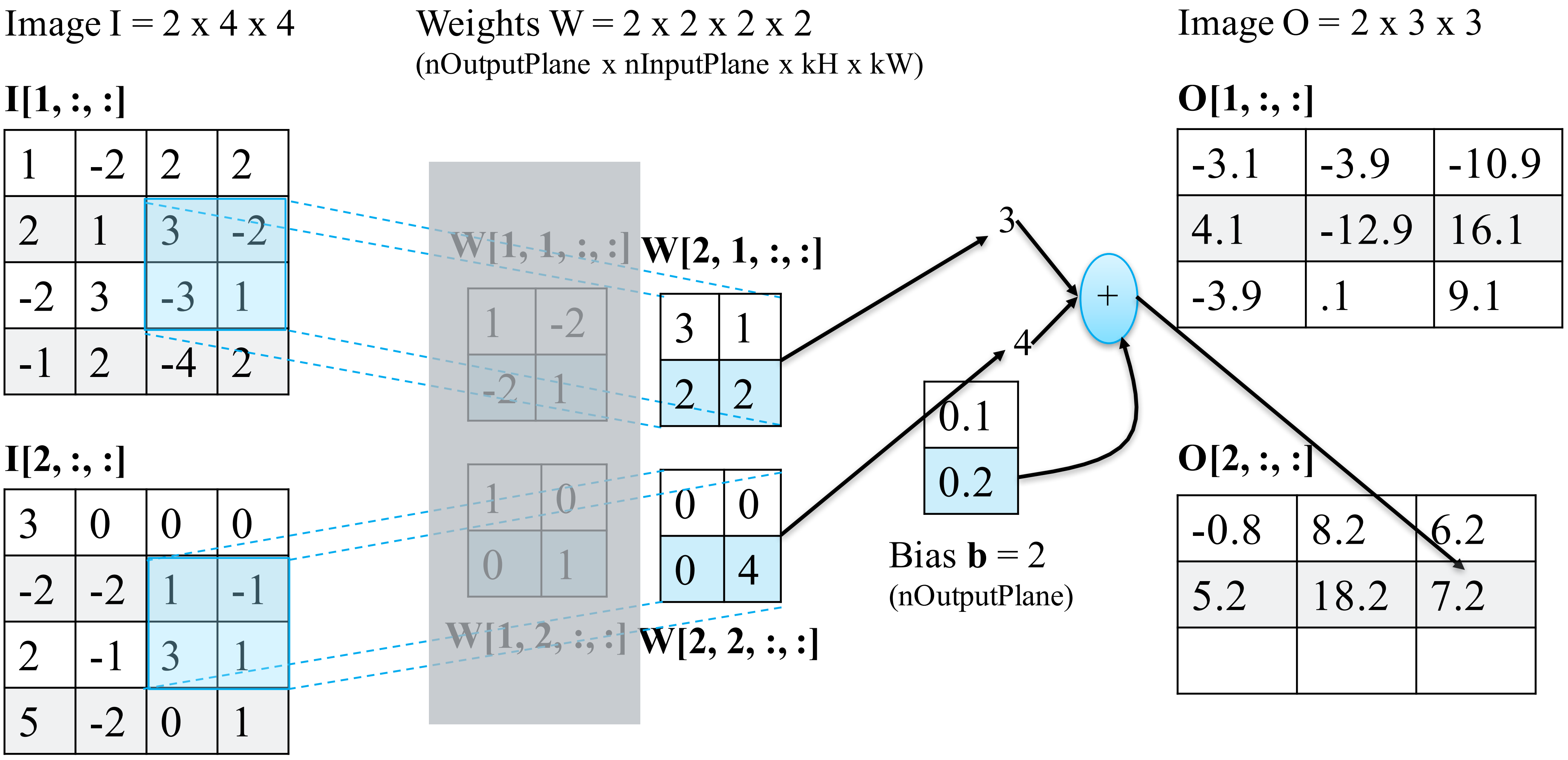
O[2, :, :]

-0.8	8.2	6.2
5.2		

Building Blocks - Convolution



Building Blocks - Convolution



Building Blocks - Convolution

Image I = 2 x 4 x 4

I[1, :, :]

1	-2	2	2
2	1	3	-2
-2	3	-3	1
-1	2	-4	2

I[2, :, :]

3	0	0	0
-2	-2	1	-1
2	-1	3	1
5	-2	0	1

Weights W = 2 x 2 x 2 x 2

(nOutputPlane x nInputPlane x kH x kW)

W[1, 1, :, :]

1	-2
-2	1

W[1, 2, :, :]

1	0
0	1

W[2, 1, :, :]

3	1
2	2

W[2, 2, :, :]

0	0
0	4

-1

-8

+

0.1
0.2

Bias **b** = 2

(nOutputPlane)

Image O = 2 x 3 x 3

O[1, :, :]

-3.1	-3.9	-10.9
4.1	-12.9	16.1
-3.9	.1	9.1

O[2, :, :]

-0.8	8.2	6.2
5.2	18.2	7.2
-8.8		

Building Blocks - Convolution

Image I = 2 x 4 x 4

I[1, :, :]

1	-2	2	2
2	1	3	-2
-2	3	-3	1
-1	2	-4	2

I[2, :, :]

3	0	0	0
-2	-2	1	-1
2	-1	3	1
5	-2	0	1

Weights W = 2 x 2 x 2 x 2
(nOutputPlane x nInputPlane x kH x kW)

W[1, 1, :, :]

1	-2
-2	1

W[1, 2, :, :]

1	0
0	1

W[2, 1, :, :]

3	1
2	2

W[2, 2, :, :]

0	0
0	4

2

0

+

0.1
0.2

Bias **b** = 2
(nOutputPlane)

Image O = 2 x 3 x 3

O[1, :, :]

-3.1	-3.9	-10.9
4.1	-12.9	16.1
-3.9	.1	9.1

O[2, :, :]

-0.8	8.2	6.2
5.2	18.2	7.2
-8.8	2.2	

Building Blocks - Convolution

Image I = 2 x 4 x 4

I[1, :, :]

1	-2	2	2
2	1	3	-2
-2	3	-3	1
-1	2	-4	2

I[2, :, :]

3	0	0	0
-2	-2	1	-1
2	-1	3	1
5	-2	0	1

Weights W = 2 x 2 x 2 x 2
(nOutputPlane x nInputPlane x kH x kW)

W[1, 1, :, :]

1	-2
-2	1

W[1, 2, :, :]

1	0
0	1

W[2, 1, :, :]

3	1
2	2

W[2, 2, :, :]

0	0
0	4

-12

4

+

0.1
0.2

Bias **b** = 2
(nOutputPlane)

Image O = 2 x 3 x 3

O[1, :, :]

-3.1	-3.9	-10.9
4.1	-12.9	16.1
-3.9	.1	9.1

O[2, :, :]

-0.8	8.2	6.2
5.2	18.2	7.2
-8.8	2.2	-7.8

Building Blocks – Convolution – in Torch7

```
> I = torch.DoubleTensor({{{1,-2,2,2},{2,1,3,-2},{-2,3,-3,1},{-1,2,-4,2}}, {{3,0,0,0},{-2,-2,1,-1},{2,-1,3,1},{5,-2,0,1}}})
> conv = nn.SpatialConvolutionMM(2,2,2,2)
> conv.bias = torch.DoubleTensor({0.1, 0.2})
> conv.weight = torch.DoubleTensor({{1,-2,-2,1,1,0,0,1},{3,1,2,2,0,0,0,4}})
> O = conv:forward(I)
> =I
(1,.,.) =
  1 -2  2  2
  2  1  3 -2
 -2  3 -3  1
 -1  2 -4  2

(2,.,.) =
  3  0  0  0
 -2 -2  1 -1
  2 -1  3  1
  5 -2  0  1
[torch.DoubleTensor of size 2x4x4]

> =O
(1,.,.) =
   3.1000  -3.9000 -10.9000
   4.1000 -12.9000  16.1000
  -3.9000   0.1000   9.1000

(2,.,.) =
  -0.8000   8.2000   6.2000
   5.2000  18.2000   7.2000
  -8.8000   2.2000  -7.8000
[torch.DoubleTensor of size 2x3x3]
```

Building Blocks – Convolution – in Torch7

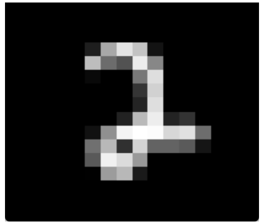
```
function SpatialConvolutionMM:__init(nInputPlane, nOutputPlane, kW, kH, dW, dH, padW, padH)
   parent.__init(self)

   dW = dW or 1
   dH = dH or 1

   self.nInputPlane = nInputPlane
   self.nOutputPlane = nOutputPlane
   self.kW = kW
   self.kH = kH

   self.dW = dW
   self.dH = dH
   self.padW = padW or 0
   self.padH = padH or self.padW

   self.weight = torch.Tensor(nOutputPlane, nInputPlane*kH*kW)
   self.bias = torch.Tensor(nOutputPlane)
   self.gradWeight = torch.Tensor(nOutputPlane, nInputPlane*kH*kW)
   self.gradBias = torch.Tensor(nOutputPlane)

   self:reset()
end
```

https://github.com/torch/nn/blob/master/SpatialConvolutionMM.lua

Building Blocks - Convolution

$$\frac{\partial \boldsymbol{L}}{\partial \boldsymbol{n}} = \frac{\partial L}{\partial \boldsymbol{m}} \times \frac{\partial \boldsymbol{m}}{\partial \boldsymbol{n}}^{(8\times12\times12)\times(16\times16)} \in \mathbb{R}^{1\times(16\times16)}$$

$$\frac{\partial L}{\partial \boldsymbol{m}} \in \mathbb{R}^{1\times(8\times12\times12)}$$



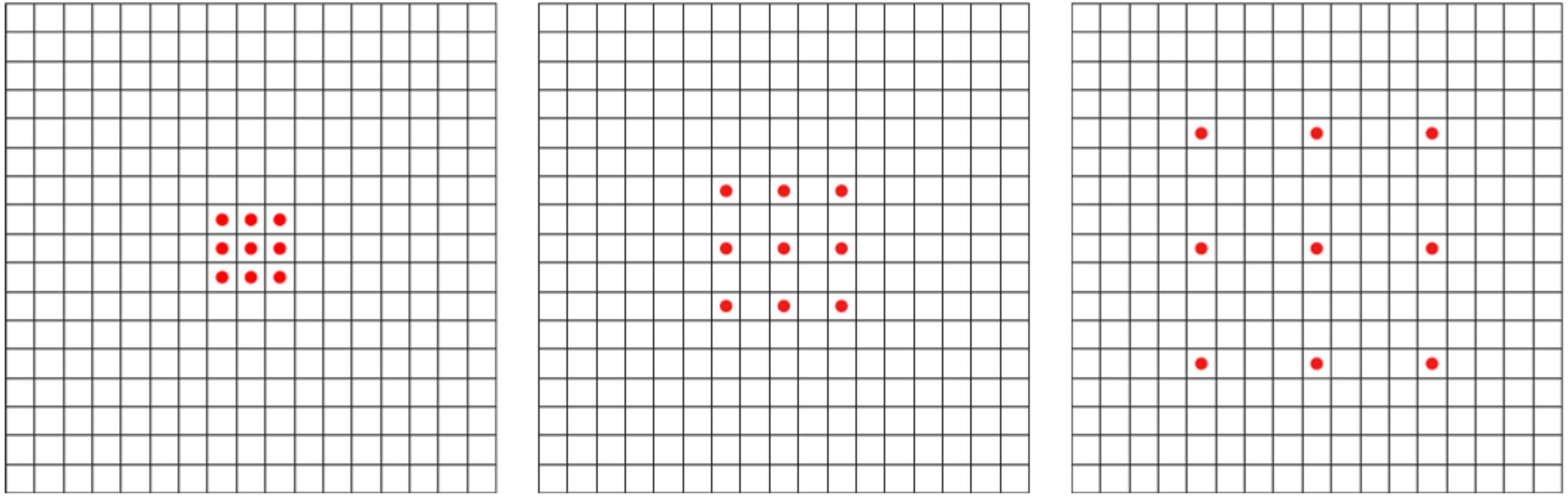
$$\boldsymbol{n} \in \mathbb{R}^{16\times16} \rightarrow \text{Convolution} \rightarrow \boldsymbol{m} \in \mathbb{R}^{8\times12\times12}$$

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \boldsymbol{m}} \times \frac{\partial \boldsymbol{m}}{\partial W} \in \mathbb{R}^{1\times8\times5\times5}$$

W

```
> nextgrad = torch.ones(8,12,12)
> conv:backward(n, nextgrad)
> =conv.gradInput:size()
  1
 16
 16
[torch.LongStorage of size 3]

> return conv.gradWeight:size()
  8
 25
[torch.LongStorage of size 2]
```

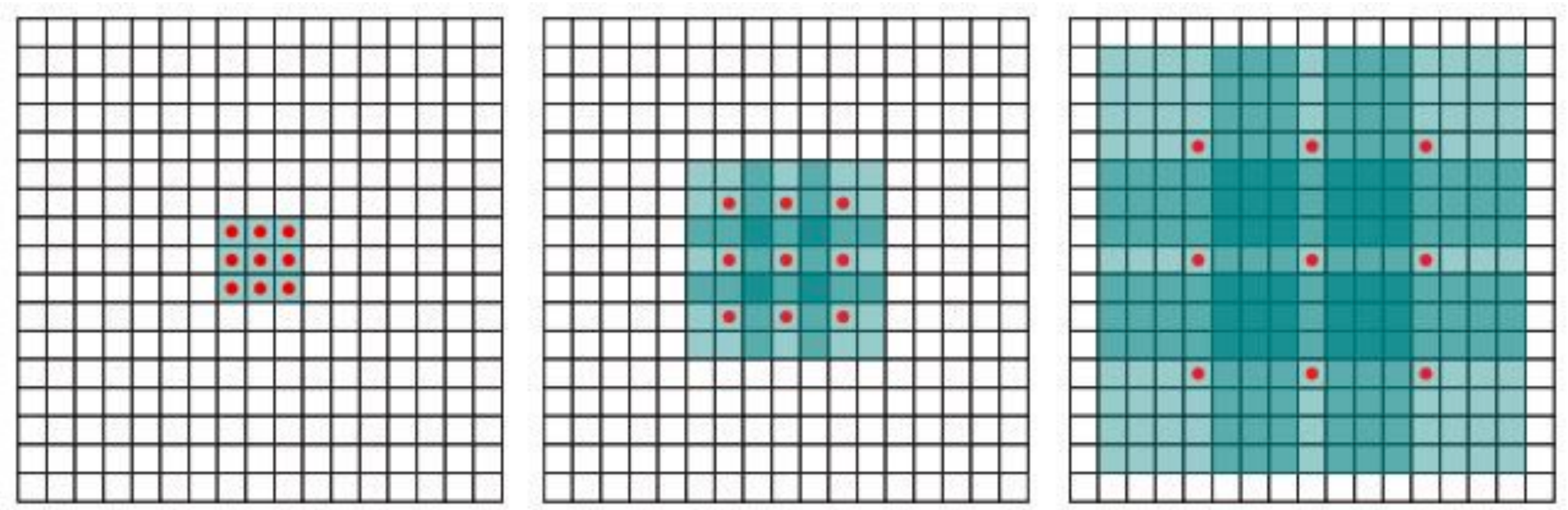
Aside: Dilated Convolution



(a) (b) (c)

Aside: Dilated Convolution

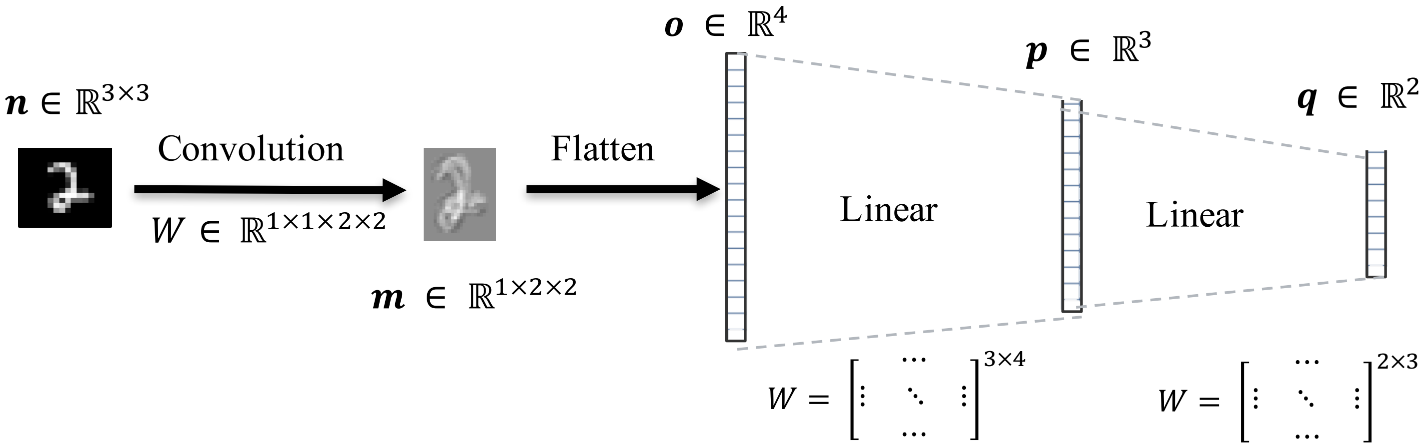
module = nn.SpatialDilatedConvolution(nInputPlane, nOutputPlane, kW, kH, [dW], [dH], [padW], [padH], [dilationW], [dilationH])



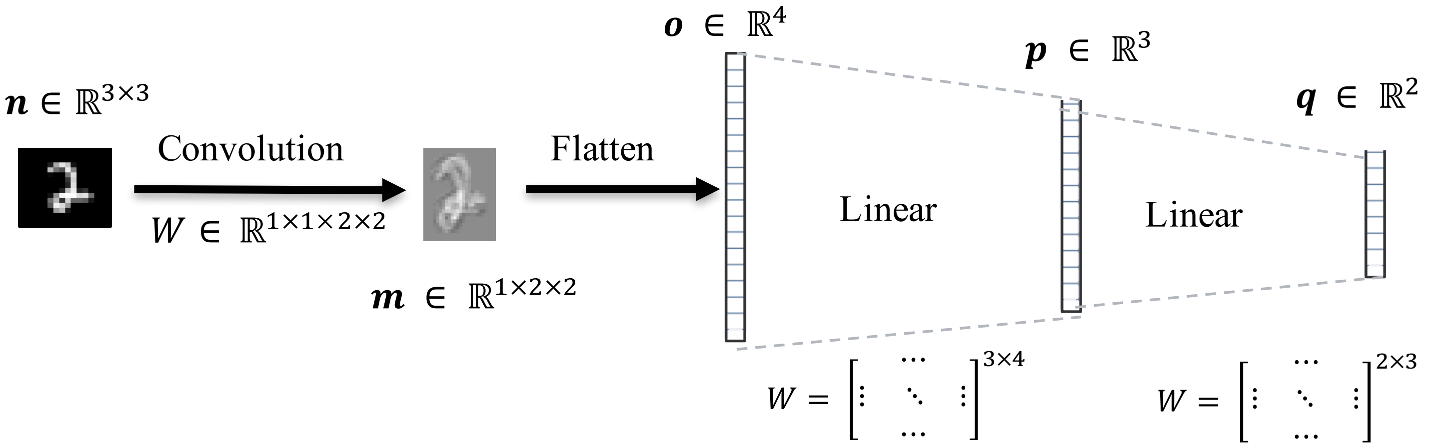
(a) (b) (c)

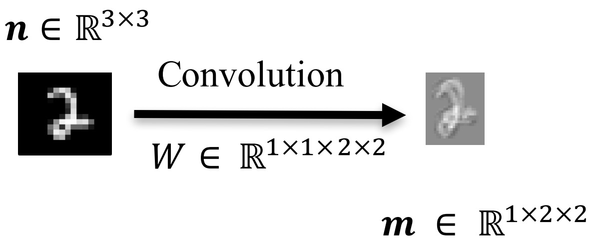
Multi-Scale Context Aggregation by Dilated Convolutions
[Fisher Yu](), [Vladlen Koltun]()

Everything is a Convolution!

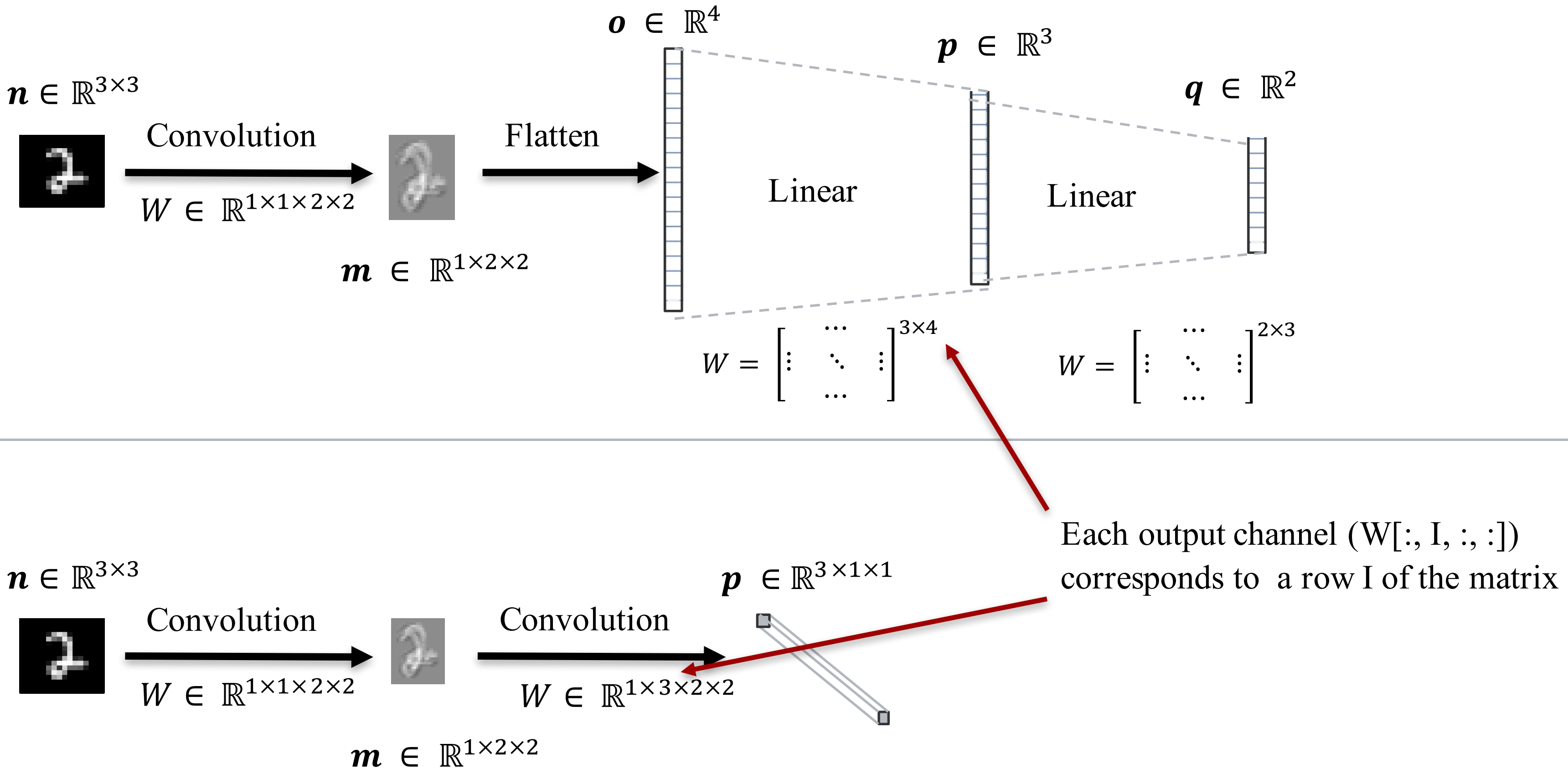


Everything is a Convolution!

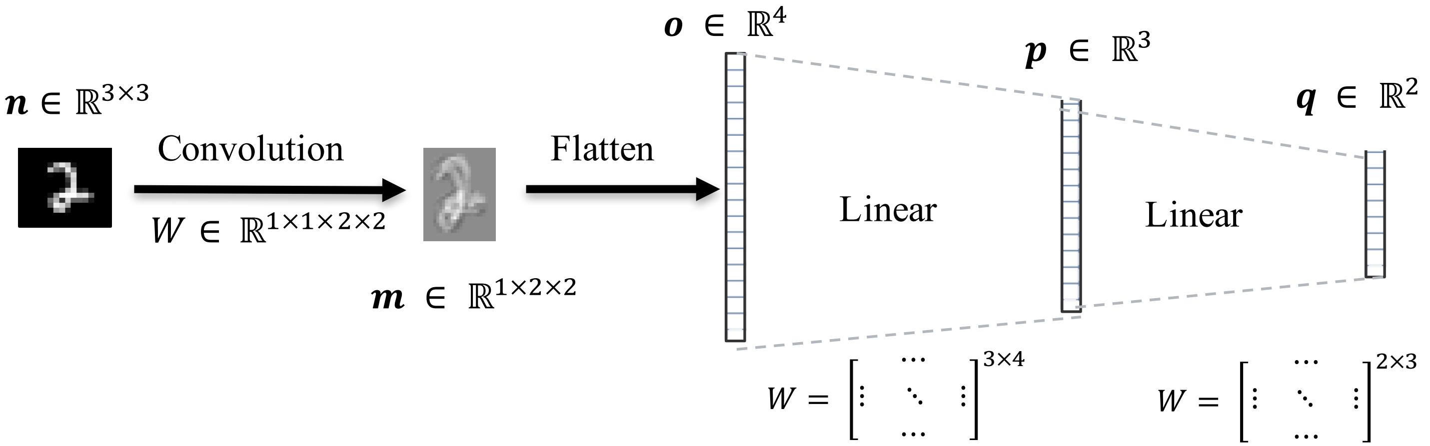


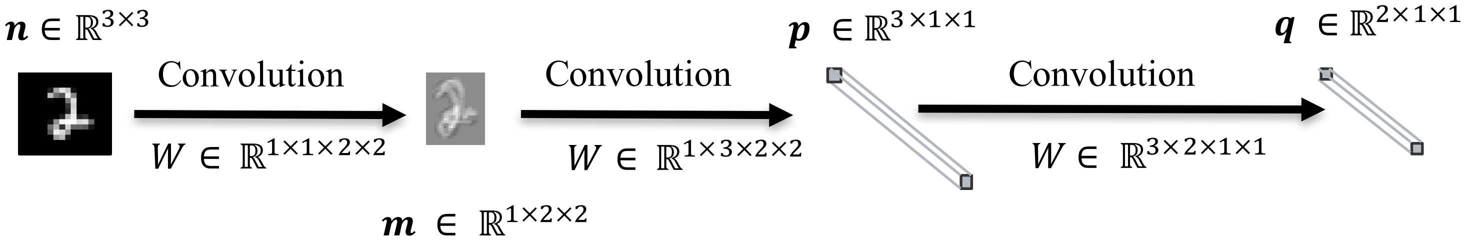


Everything is a Convolution!



Everything is a Convolution!

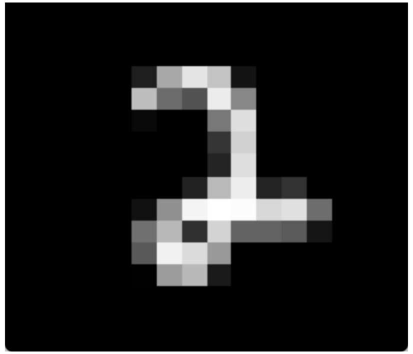




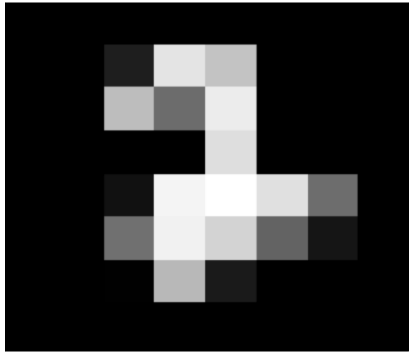
Building Blocks:
Max Pooling

https://github.com/stencilman/CS763_Spring2017/blob/master/Notebooks/Max-Pool.ipynb

Building Blocks – Pooling (Max Pooling)

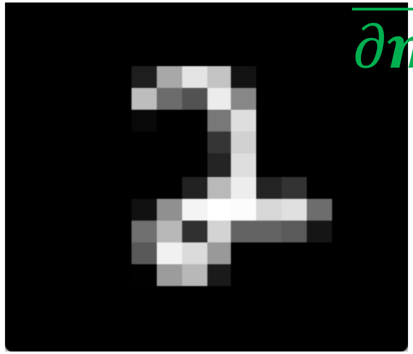


$\boldsymbol{n} \in \mathbb{R}^{4\times 4}$ → Max Pooling → $\boldsymbol{m} \in \mathbb{R}^{2\times 2}$

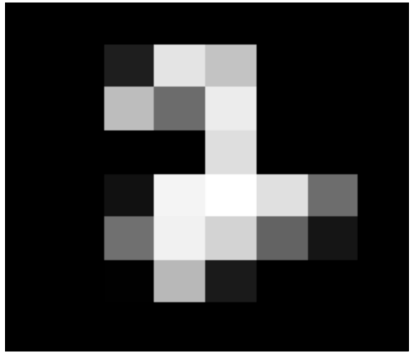


Building Blocks – Pooling (Max Pooling)

$$\frac{\partial L}{\partial \boldsymbol{m}} \cdot \frac{\partial \boldsymbol{m}}{\partial \boldsymbol{n}} \in \mathbb{R}^{1\times(4\times4)} \qquad \frac{\partial \boldsymbol{m}}{\partial \boldsymbol{n}} \in \mathbb{R}^{(2\times2)\times(4\times4)} \qquad \frac{\partial L}{\partial \boldsymbol{m}} \in \mathbb{R}^{1\times(2\times2)}$$

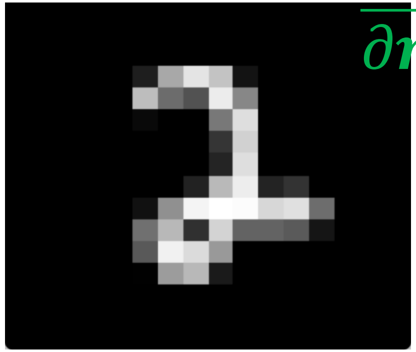


$$\boldsymbol{n} \in \mathbb{R}^{4\times4} \rightarrow \text{Max Pooling} \rightarrow \boldsymbol{m} \in \mathbb{R}^{2\times2}$$

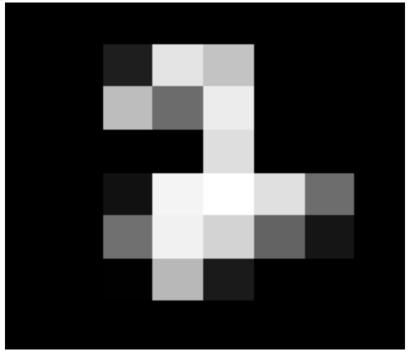


Building Blocks – Pooling (Max Pooling) – in Torch7

$\frac{\partial L}{\partial \boldsymbol{m}} \cdot \frac{\partial \boldsymbol{m}}{\partial \boldsymbol{n}} \in \mathbb{R}^{1\times(4\times4)}$ $\qquad$ $\frac{\partial \boldsymbol{m}}{\partial \boldsymbol{n}} \in \mathbb{R}^{(2\times2)\times(4\times4)}$ $\qquad$ $\frac{\partial L}{\partial \boldsymbol{m}} \in \mathbb{R}^{1\times(2\times2)}$



$\boldsymbol{n} \in \mathbb{R}^{4\times4} \longrightarrow$ Max Pooling $\longrightarrow \boldsymbol{m} \in \mathbb{R}^{2\times2}$



```
> n = torch.rand(1,4,4)
> pool = nn.SpatialMaxPooling(2, 2)
> m = pool:forward(n)
> =n
(1,.,.) =
  0.2692  0.4190  0.2095  0.9163
  0.2778  0.9199  0.5555  0.1638
  0.6936  0.2328  0.0553  0.1798
  0.3611  0.3225  0.9032  0.5106
[torch.DoubleTensor of size 1x4x4]

> =m
(1,.,.) =
  0.9199  0.9163
  0.6936  0.9032
[torch.DoubleTensor of size 1x2x2]
```

```
> nextgrad = torch.ones(1,2,2)
> pool:backward(n, nextgrad)
> =pool.gradInput
(1,.,.) =
  0  0  0  1
  0  1  0  0
  1  0  0  0
  0  0  1  0
[torch.DoubleTensor of size 1x4x4]
```

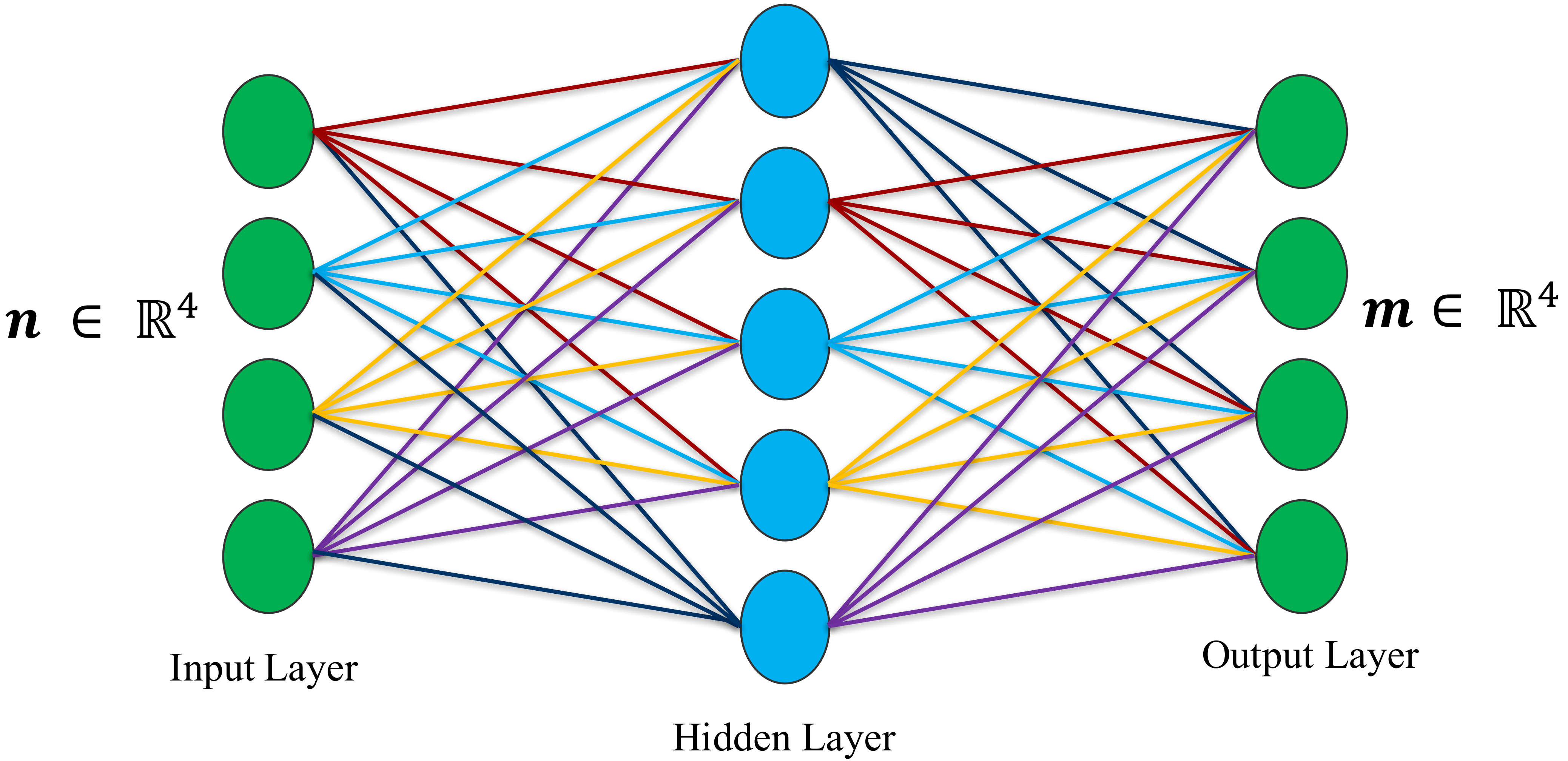
Other Pooling Layers

- Average Pooling
- No Pooling? **Striving for Simplicity: The All Convolutional Net**
  Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, Martin Riedmiller

Weight Initialization

https://github.com/stencilman/CS763_Spring2017/blob/master/Notebooks/Weight-init.ipynb

What happens when W=0 init is used?



- First idea: **Small random numbers** (normal distribution between -0.01 to 0.01)

```
self.W = torch.randn(fan_out, fan_in) * 0.01
self.b = torch.randn(fan_out)* 0.01
```

- First idea: **Small random numbers** (normal distribution between -0.01 to 0.01)

```
self.W = torch.randn(fan_out, fan_in) * 0.01
self.b = torch.randn(fan_out)* 0.01
```

Works ~okay for small networks (like our one layer cifar-10 classifier), but can lead to non-homogeneous distributions of activations across the layers of a network.

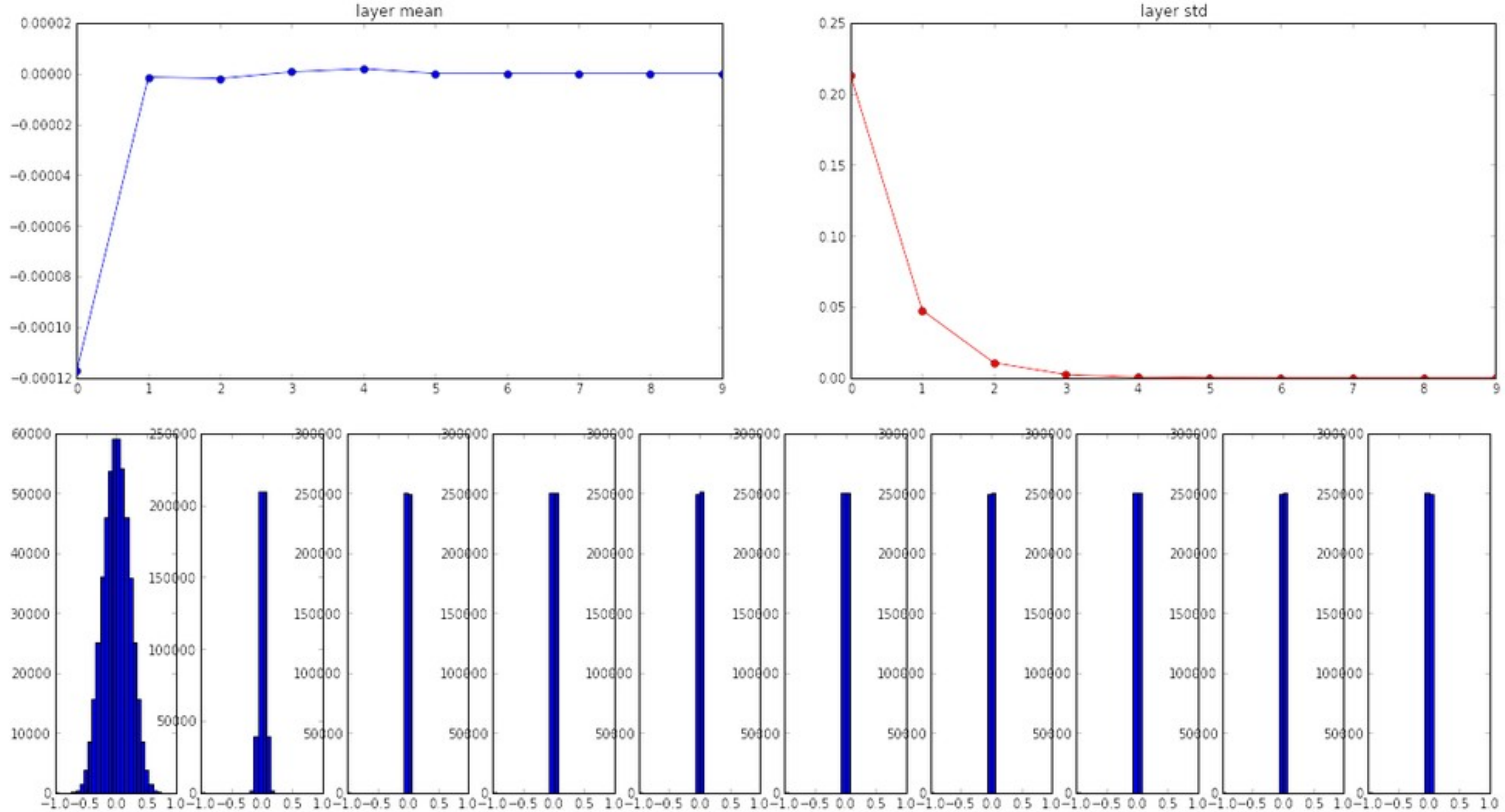
Lets look at some activation statistics

E.g. 10-layer net with 500 neurons on each layer, using tanh non-linearities, and initializing as described in last slide.

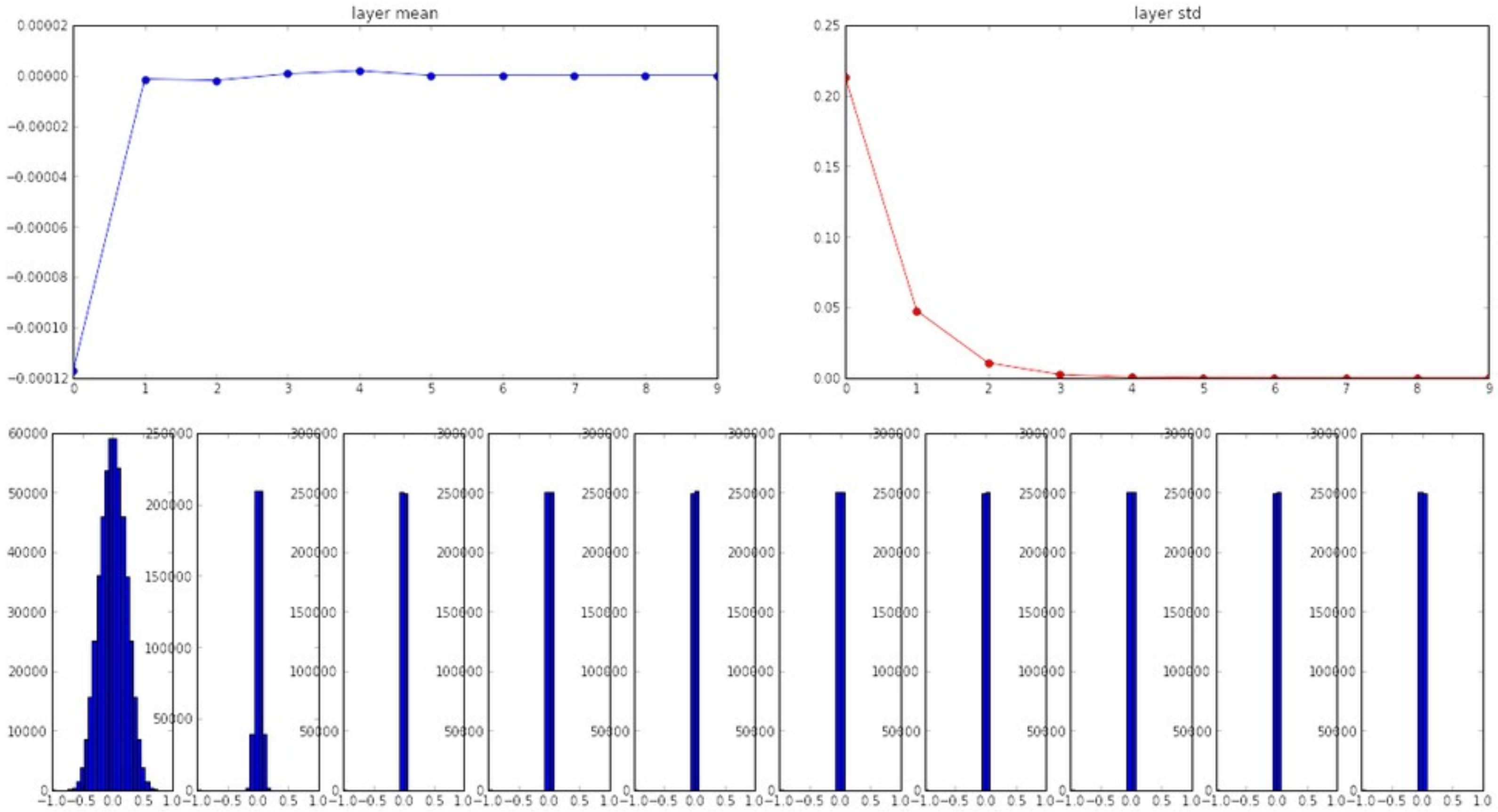
```
local Linear = torch.class("Linear")
function Linear:__init(fan_in, fan_out)
    self.output = nil
    self.W = torch.randn(fan_out, fan_in) * 0.01
    self.b = torch.randn(fan_out)* 0.01
end
-- Wx + b
function Linear:forward(xi)
    self.output = self.W * xi:reshape(x:size(1),1) + self.b
    return self.output
end

x = torch.randn(1000)
plot = Plot():histogram(x, 100, -1, 1):title('input'):draw()
l = Linear.new(1000, 1000)
input = x
means = {}
stds = {}
xaxis = {}
inputstr = string.format('Input  mean:  %f std: %f', x:mean(1)[1], x:std(1)[1])
for i = 1, 10 do
    h = l:forward(input)
    h = nn.Tanh():forward(h)
    mean  = h:mean(1)[1][1]
    std = h:std(1)[1][1]
    table.insert(means, mean)
    table.insert(stds, std)
    table.insert(xaxis, i)
    plot = Plot():histogram(h, 100,-1,1):title('layer '..i):draw();
    input = h
end
plot = Plot():line(xaxis,means):title('mean'):draw()
plot = Plot():line(xaxis,stds):title('std'):draw()
print(inputstr)
for i=1,10 do
    print(string.format('Layer  %d mean:  %f std: %f',i, means[i], stds[i]))
end
```

Lets look at some activation statistics

```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```



```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```



All activations become zero!

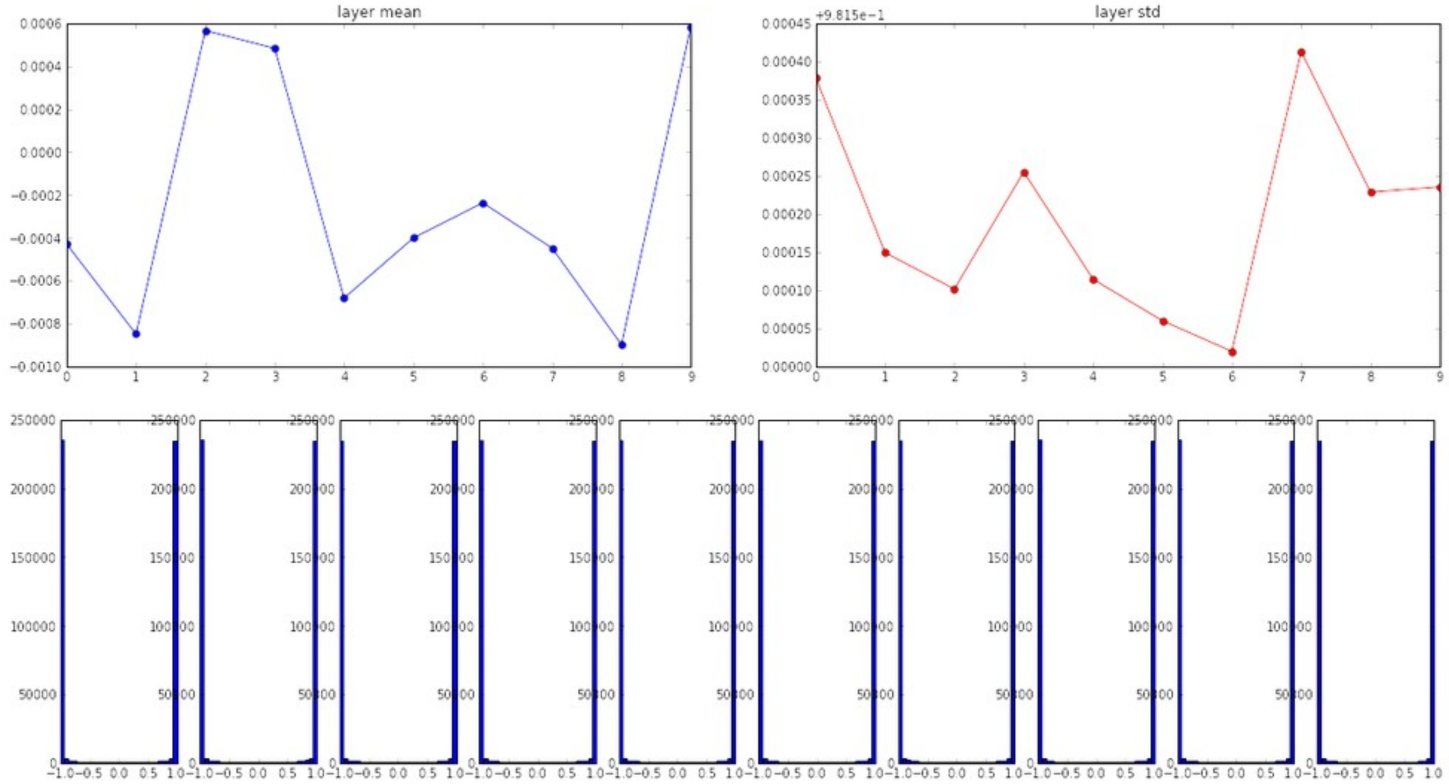
Q: think about the backward pass. What do the gradients look like?

Hint: think about backward pass, the W update.

```
self.W = torch.randn(fan_out, fan_in) * 1.0
self.b = torch.randn(fan_out)* 1.0
```

*1.0 instead of *0.01

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean -0.000430 and std 0.981879
hidden layer 2 had mean -0.000849 and std 0.981649
hidden layer 3 had mean 0.000566 and std 0.981601
hidden layer 4 had mean 0.000483 and std 0.981755
hidden layer 5 had mean -0.000682 and std 0.981614
hidden layer 6 had mean -0.000401 and std 0.981560
hidden layer 7 had mean -0.000237 and std 0.981520
hidden layer 8 had mean -0.000448 and std 0.981913
hidden layer 9 had mean -0.000899 and std 0.981728
hidden layer 10 had mean 0.000584 and std 0.981736
```

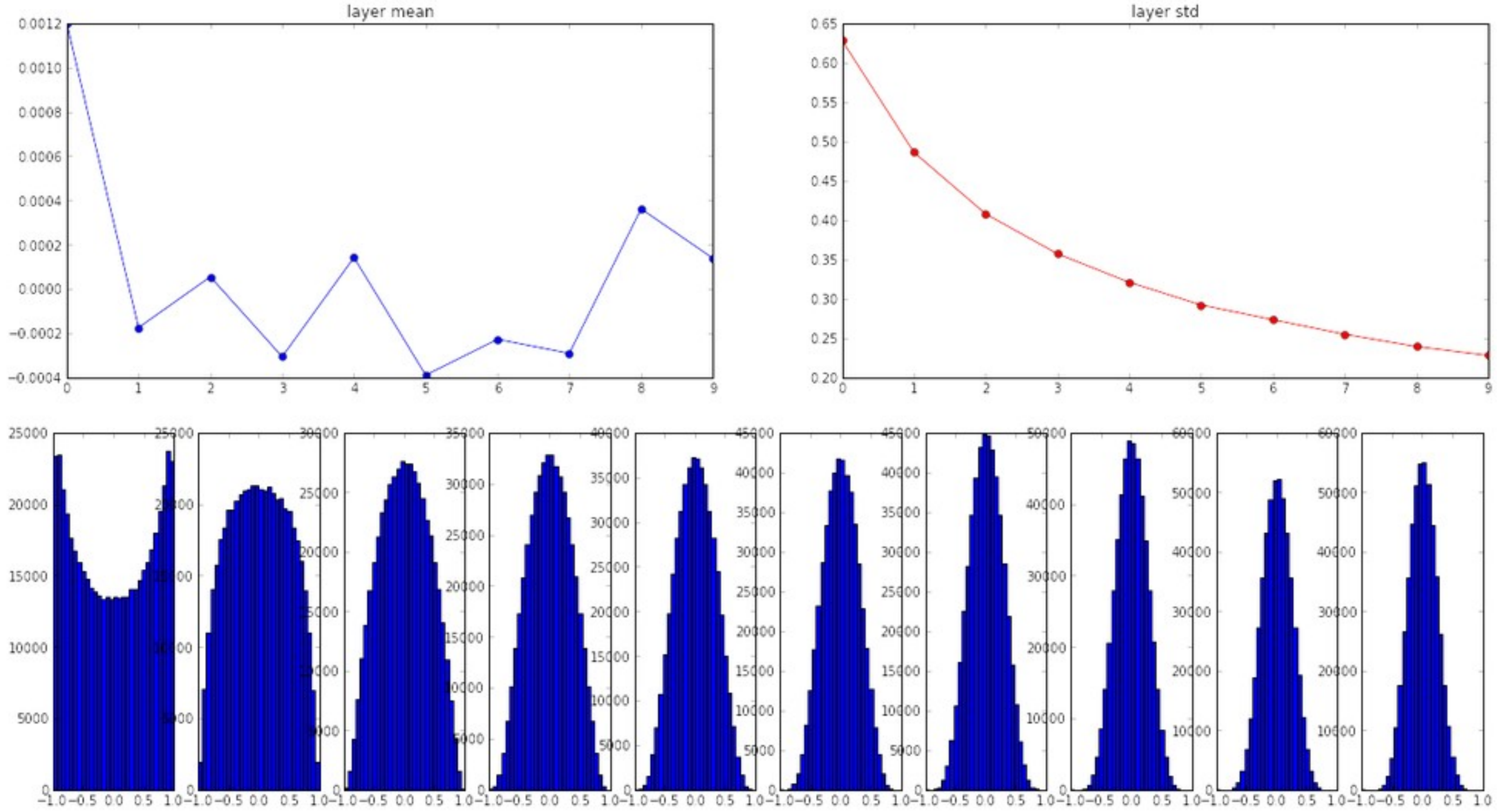


Almost all neurons completely saturated, either -1 and 1. Gradients will be all zero.

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean 0.001198 and std 0.627953
hidden layer 2 had mean -0.000175 and std 0.486051
hidden layer 3 had mean 0.000055 and std 0.407723
hidden layer 4 had mean -0.000306 and std 0.357108
hidden layer 5 had mean 0.000142 and std 0.320917
hidden layer 6 had mean -0.000389 and std 0.292116
hidden layer 7 had mean -0.000228 and std 0.273387
hidden layer 8 had mean -0.000291 and std 0.254935
hidden layer 9 had mean 0.000361 and std 0.239266
hidden layer 10 had mean 0.000139 and std 0.228008
```

```python
self.W = torch.randn(fan_out, fan_in) / math.sqrt(fan_in)
self.b = torch.randn(fan_out) / math.sqrt(fan_in)
```

“Xavier initialization”
[Glorot et al., 2010]

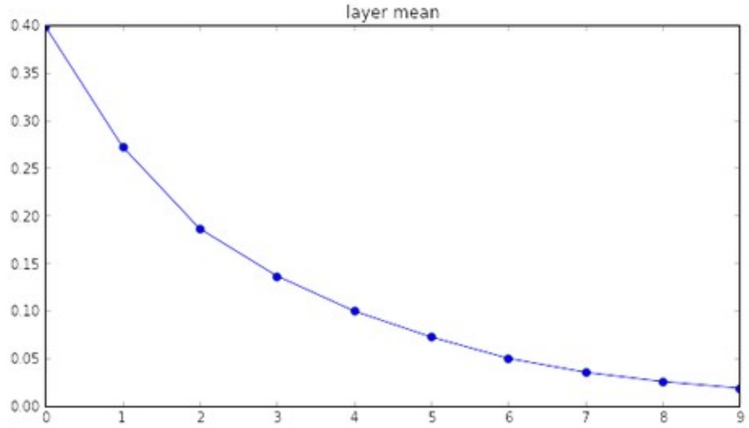


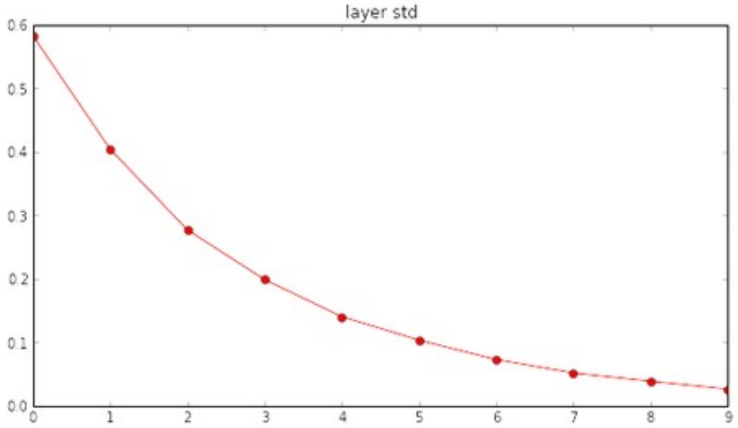
Reasonable initialization.
(Mathematical derivation assumes linear activations)

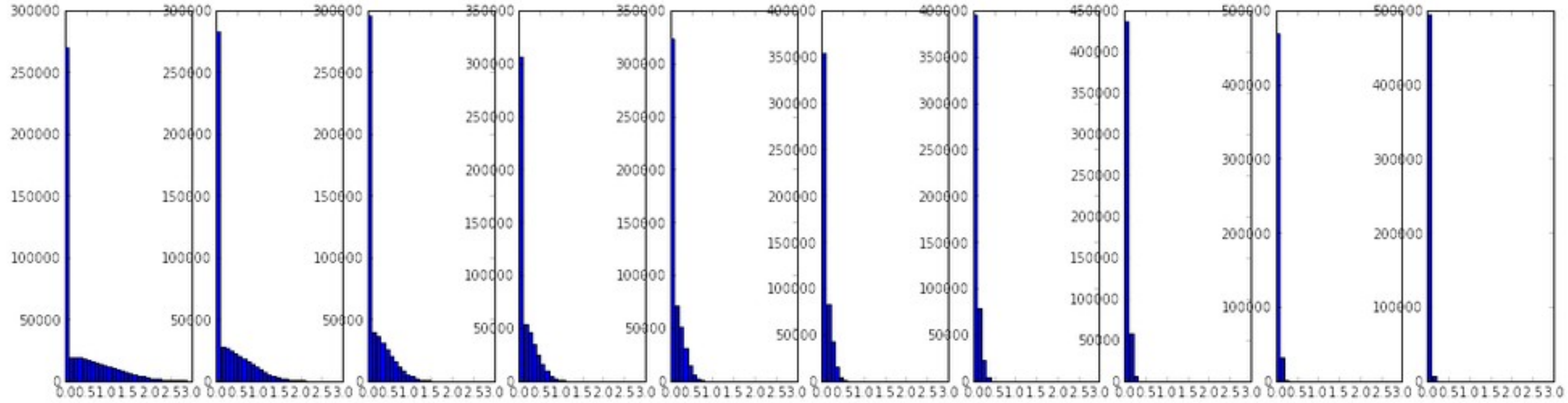
```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.398623 and std 0.582273
hidden layer 2 had mean 0.272352 and std 0.403795
hidden layer 3 had mean 0.186076 and std 0.276912
hidden layer 4 had mean 0.136442 and std 0.198685
hidden layer 5 had mean 0.099568 and std 0.140299
hidden layer 6 had mean 0.072234 and std 0.103280
hidden layer 7 had mean 0.049775 and std 0.072748
hidden layer 8 had mean 0.035138 and std 0.051572
hidden layer 9 had mean 0.025404 and std 0.038583
hidden layer 10 had mean 0.018408 and std 0.026076
```

```
self.W = torch.randn(fan_out, fan_in) / math.sqrt(fan_in)
self.b = torch.randn(fan_out) / math.sqrt(fan_in)
```

but when using the ReLU nonlinearity it breaks.



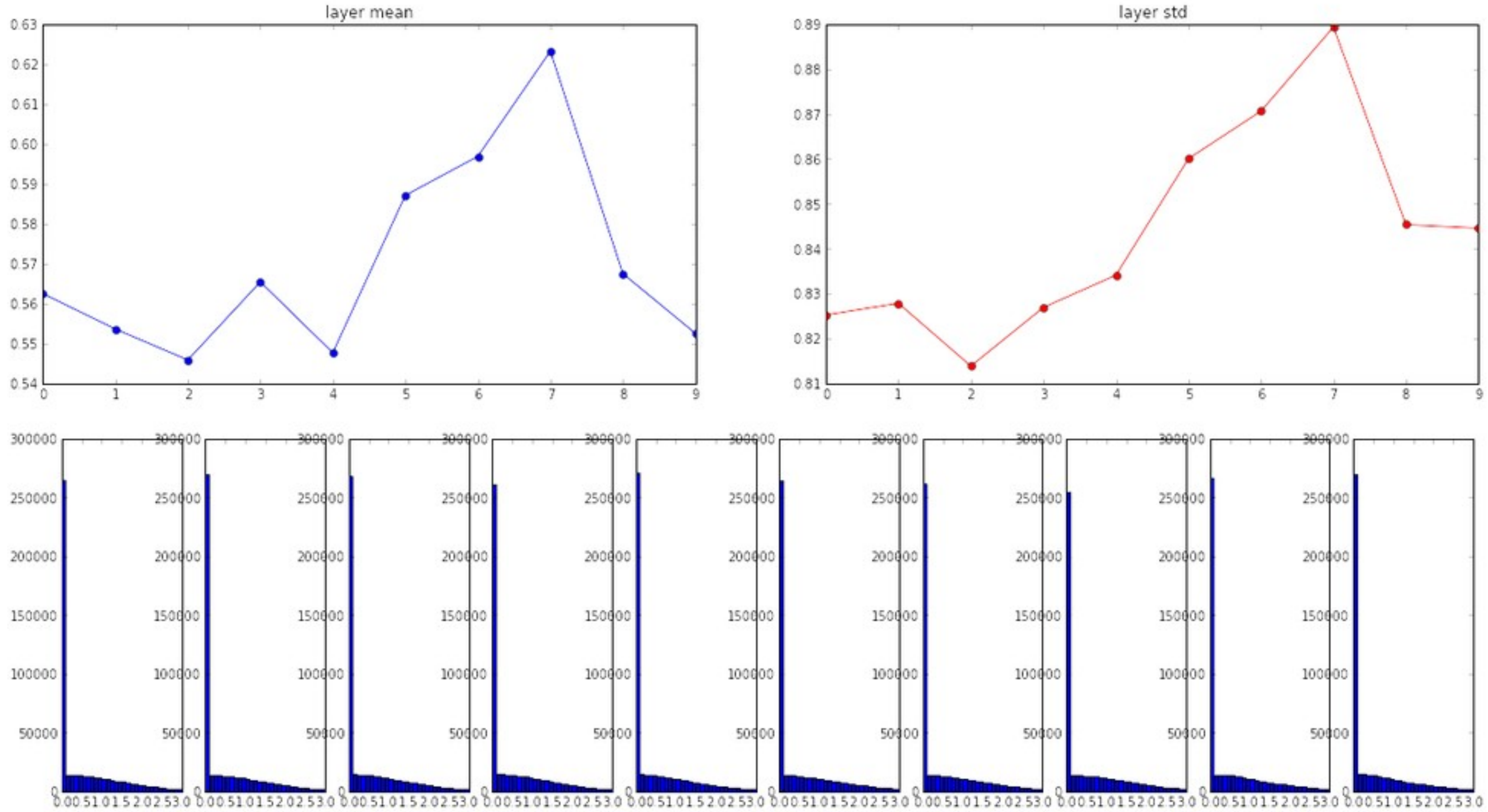




```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
hidden layer 6 had mean 0.587103 and std 0.860035
hidden layer 7 had mean 0.596867 and std 0.870610
hidden layer 8 had mean 0.623214 and std 0.889348
hidden layer 9 had mean 0.567498 and std 0.845357
hidden layer 10 had mean 0.552531 and std 0.844523
```

```python
self.W = torch.randn(fan_out, fan_in) / math.sqrt(fan_in/2)
self.b = torch.randn(fan_out) / math.sqrt(fan_in/2)
```

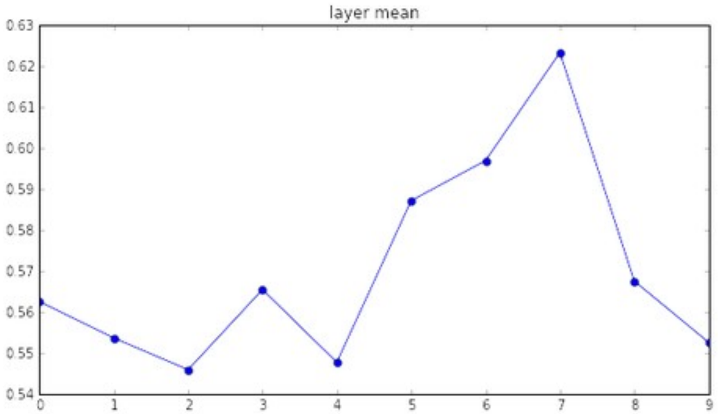
He et al., 2015
(note additional /2)

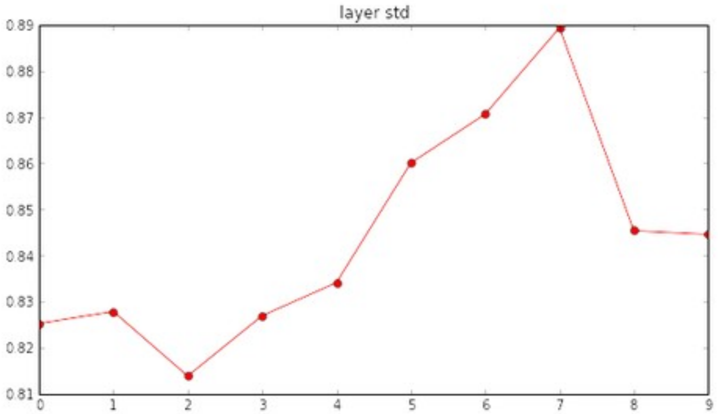


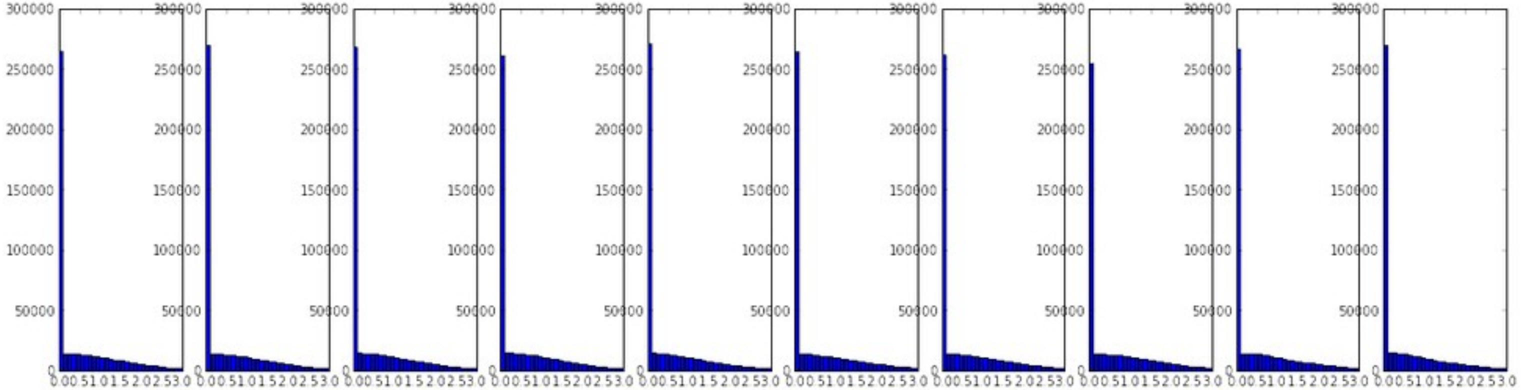
```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
hidden layer 6 had mean 0.587103 and std 0.860035
hidden layer 7 had mean 0.596867 and std 0.870610
hidden layer 8 had mean 0.623214 and std 0.889348
hidden layer 9 had mean 0.567498 and std 0.845357
hidden layer 10 had mean 0.552531 and std 0.844523
```

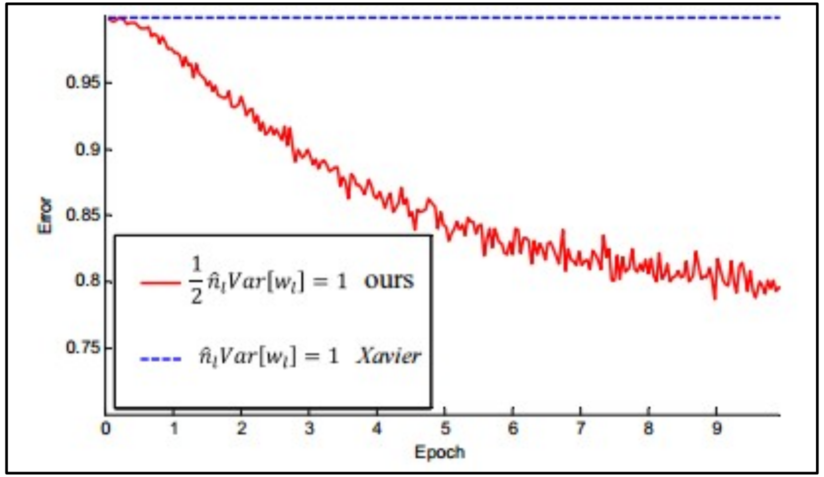
```
self.W = torch.randn(fan_out, fan_in) / math.sqrt(fan_in/2)
self.b = torch.randn(fan_out) / math.sqrt(fan_in/2)
```

He et al., 2015
(note additional /2)









```lua
-- "Efficient backprop"
-- Yann Lecun, 1998
local function w_init_heuristic(fan_in, fan_out)
   return math.sqrt(1/(3*fan_in))
end


-- "Understanding the difficulty of training deep feedforward neural networks"
-- Xavier Glorot, 2010
local function w_init_xavier(fan_in, fan_out)
   return math.sqrt(2/(fan_in + fan_out))
end


-- "Understanding the difficulty of training deep feedforward neural networks"
-- Xavier Glorot, 2010
local function w_init_xavier_caffe(fan_in, fan_out)
   return math.sqrt(1/fan_in)
end


-- "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification"
-- Kaiming He, 2015
local function w_init_kaiming(fan_in, fan_out)
   return math.sqrt(4/(fan_in + fan_out))
end
```

https://github.com/e-lab/torch-toolbox/blob/master/Weight-init/weight-init.lua

Proper initialization is an active area of research…

Understanding the difficulty of training deep feedforward neural networks
by Glorot and Bengio, 2010

Exact solutions to the nonlinear dynamics of learning in deep linear neural networks
by Saxe et al, 2013

Random walk initialization for training very deep feedforward networks
by Sussillo and Abbott, 2014

Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification
by He et al., 2015

Data-dependent Initializations of Convolutional Neural Networks
by Krähenbühl et al., 2015

All you need is a good init
by Mishkin and Matas, 2015

…

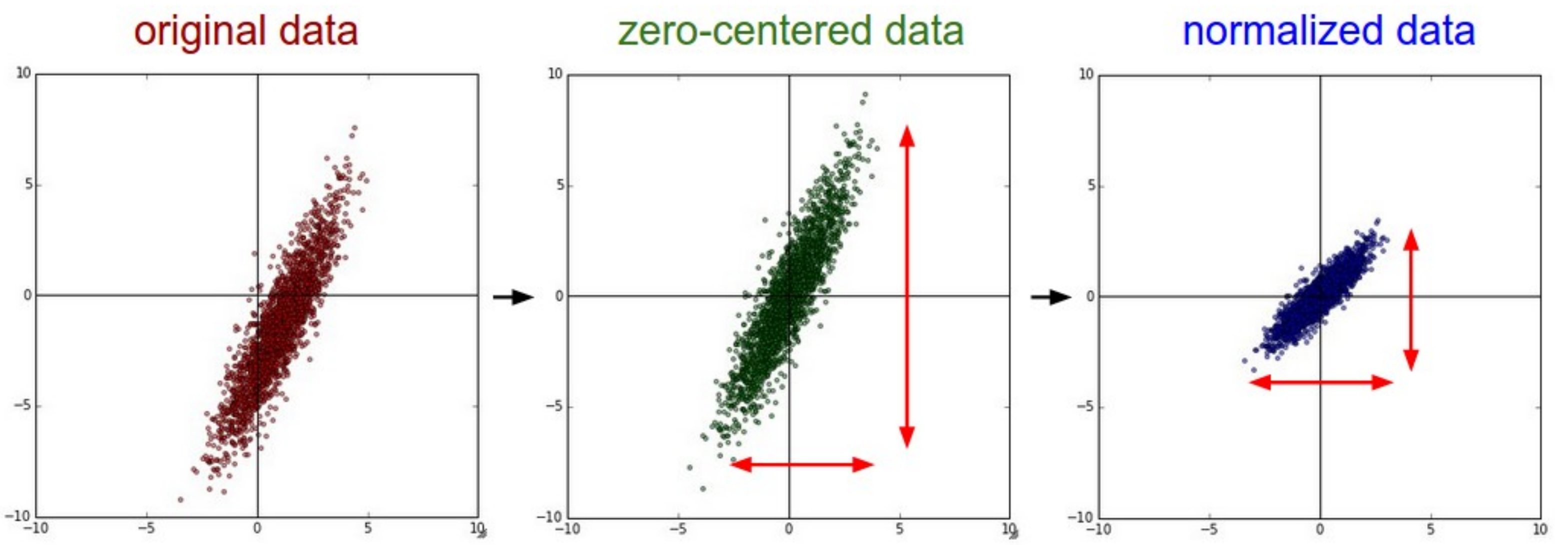
Batch Normalization

"you want unit gaussian activations? just make them so."

[Ioffe and Szegedy, 2015]

Babysitting the Learning Process

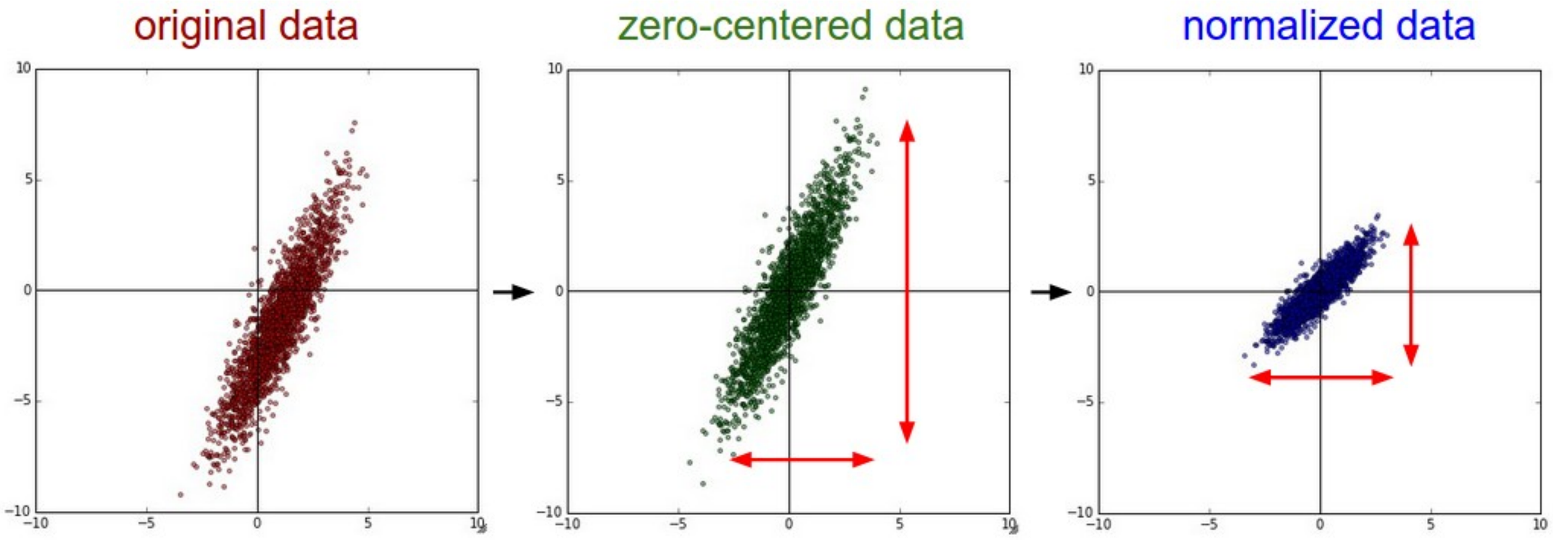
Step 1: Data Preprocessing




```
[> X = torch.rand(10,4)
[> mean = X:mean(1)
[> std = X:std(1)
[> meanrepeated = torch.repeatTensor(mean, 10,1)
[> stdrepeated = torch.repeatTensor(std, 10,1)
[> =X
 0.4822  0.2837  0.6833  0.9955
 0.1640  0.0252  0.2801  0.5854
 0.7129  0.8695  0.2564  0.8241
 0.7037  0.8237  0.2027  0.1058
 0.6759  0.2490  0.0476  0.3699
 0.5486  0.4739  0.2690  0.7482
 0.4623  0.5303  0.5062  0.1874
 0.3652  0.8691  0.6733  0.3607
 0.4448  0.8708  0.8118  0.7951
 0.2013  0.6639  0.5152  0.7776
[torch.DoubleTensor of size 10x4]

[> =mean
 0.4761  0.5659  0.4246  0.5750
[torch.DoubleTensor of size 1x4]

[> =std
 0.1942  0.3047  0.2491  0.3013
[torch.DoubleTensor of size 1x4]
```

- Assume **X** [NxD] is data matrix, each example in a row
- So, in our case we have 10 examples, each 4D

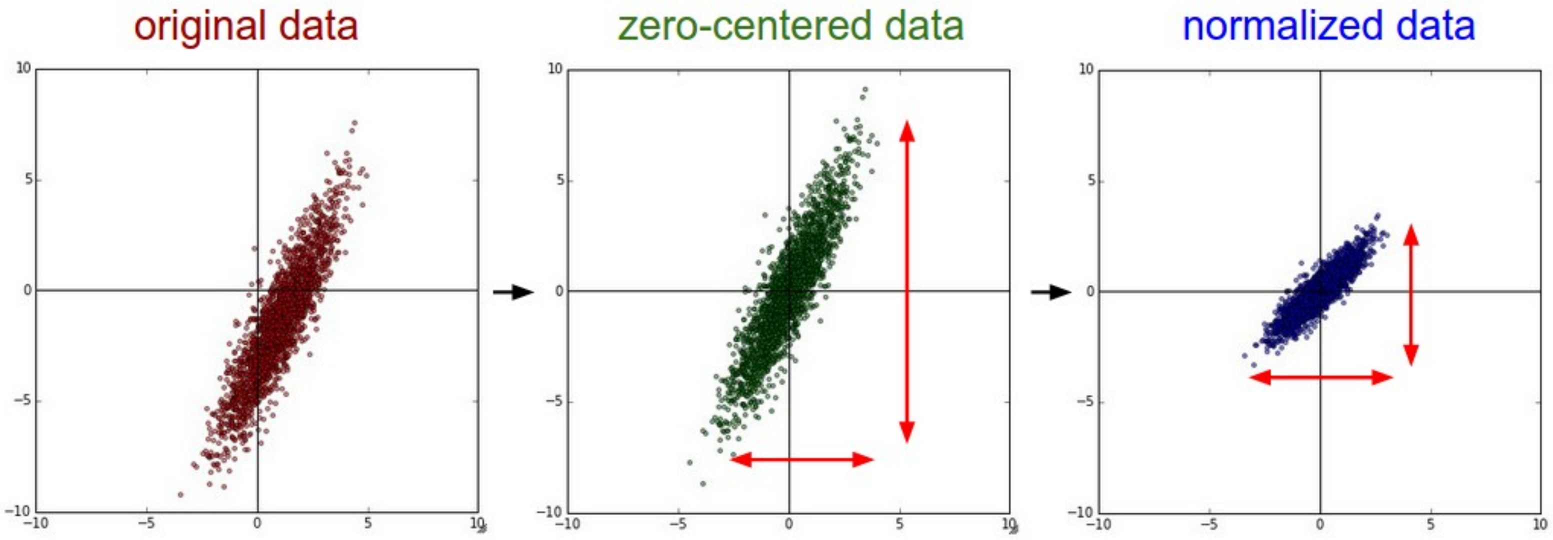
Step 1: Data Preprocessing




```
[> X = torch.rand(10,4)
[> mean = X:mean(1)
[> std = X:std(1)
[> meanrepeated = torch.repeatTensor(mean, 10,1)
[> stdrepeated = torch.repeatTensor(std, 10,1)
[> =meanrepeated
 0.4761  0.5659  0.4246  0.5750
 0.4761  0.5659  0.4246  0.5750
 0.4761  0.5659  0.4246  0.5750
 0.4761  0.5659  0.4246  0.5750
 0.4761  0.5659  0.4246  0.5750
 0.4761  0.5659  0.4246  0.5750
 0.4761  0.5659  0.4246  0.5750
 0.4761  0.5659  0.4246  0.5750
 0.4761  0.5659  0.4246  0.5750
 0.4761  0.5659  0.4246  0.5750
[torch.DoubleTensor of size 10x4]

[> =stdrepeated
 0.1942  0.3047  0.2491  0.3013
 0.1942  0.3047  0.2491  0.3013
 0.1942  0.3047  0.2491  0.3013
 0.1942  0.3047  0.2491  0.3013
 0.1942  0.3047  0.2491  0.3013
 0.1942  0.3047  0.2491  0.3013
 0.1942  0.3047  0.2491  0.3013
 0.1942  0.3047  0.2491  0.3013
 0.1942  0.3047  0.2491  0.3013
 0.1942  0.3047  0.2491  0.3013
[torch.DoubleTensor of size 10x4]
```

- Assume **X** [NxD] is data matrix, each example in a row
- So, in our case we have 10 examples, each 4D

Step 1: Data Preprocessing




```
> X = torch.rand(10,4)
> mean = X:mean(1)
> std = X:std(1)
> meanrepeated = torch.repeatTensor(mean, 10,1)
> stdrepeated = torch.repeatTensor(std, 10,1)
> X = X - meanrepeated
> X=X:cdiv(stdrepeated)
> =X
  0.0315 -0.9264  1.0390  1.3959
 -1.6070 -1.7749 -0.5799  0.0346
  1.2192  0.9964 -0.6751  0.8270
  1.1719  0.8461 -0.8909 -1.5573
  1.0287 -1.0402 -1.5134 -0.6806
  0.3735 -0.3019 -0.6247  0.5749
 -0.0709 -0.1170  0.3280 -1.2866
 -0.5710  0.9953  0.9986 -0.7111
 -0.1611  1.0008  1.5545  0.7308
 -1.4149  0.3216  0.3639  0.6725
[torch.DoubleTensor of size 10x4]
```

- Assume **X** [NxD] is data matrix, each example in a row
- So, in our case we have 10 examples, each 4D

Notebook

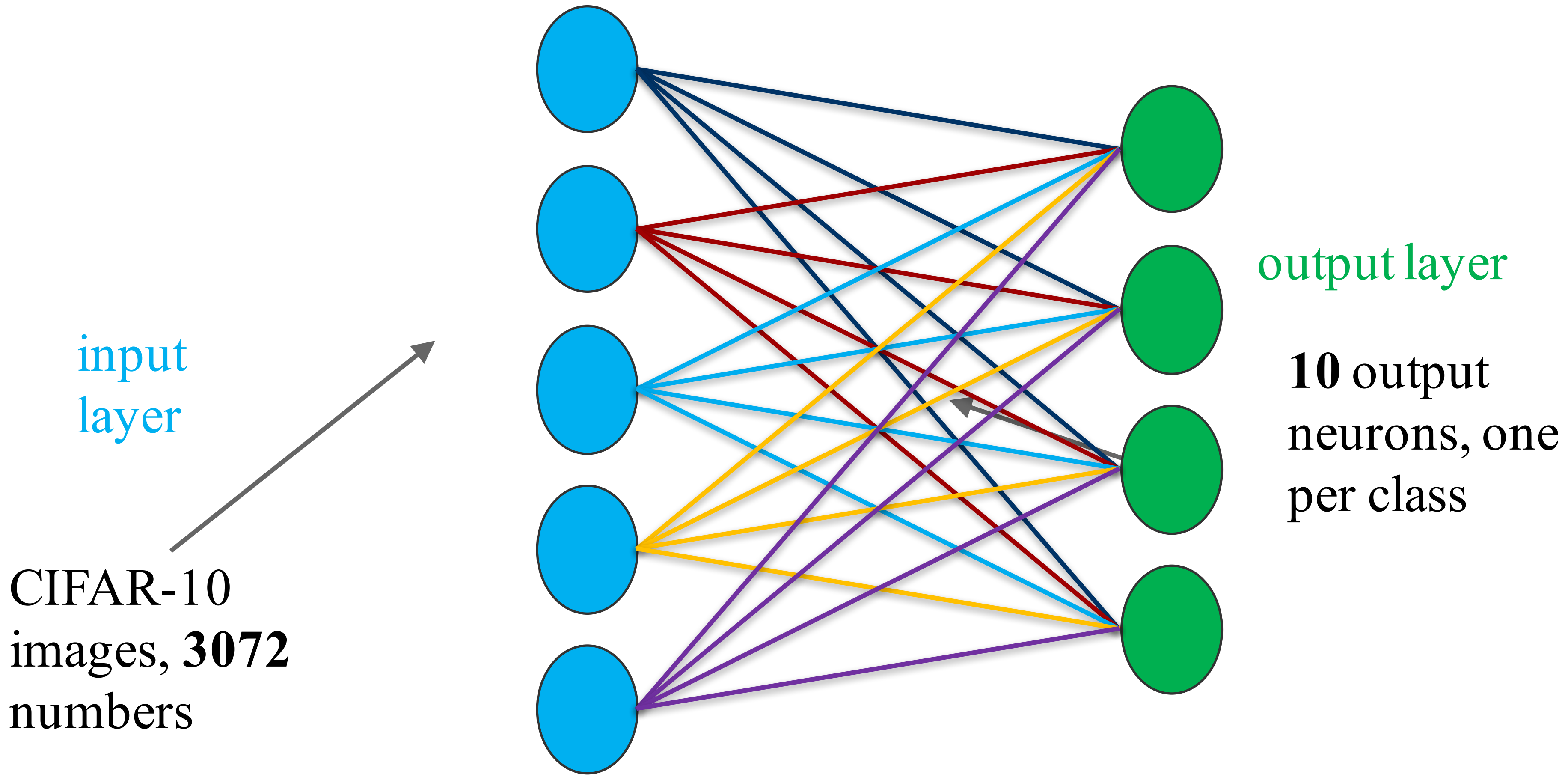
2. Data Preprocessing: We compute the mean and standard deviation 'images' and then subtract and divide by the same respectively (like AlexNet). We also visualize them.

In [3]:
```
x_mean = torch.mean(tr_x:float(), 1)
x_std = torch.std(tr_x:float(), 1)
itorch.image(x_mean)
itorch.image(x_std)
```

In [7]:
```
function get_xi(data_x, idx)
    xi = (data_x[idx]:float() - x_mean)
    xi = xi:cdiv(x_std)
    xi = xi:reshape(3*32*32)
    return xi
end
```

Step 2: Choose the architecture:

say we start with single layer network:



Notebook

1. Data Loading: Let us load the training and the test data and check the size of the tensors. Let us also display the first few images from the training set.

In [1]:

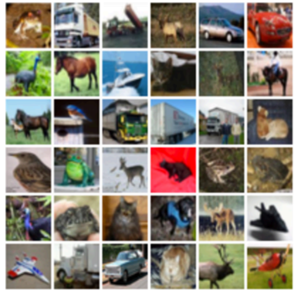
```
-- load trainin images
tr_x = torch.load('cifar10/tr_data.bin')
-- load trainin labels
tr_y = torch.load('cifar10/tr_labels.bin'):double() + 1
-- load test images
te_x = torch.load('cifar10/te_data.bin')
-- load test labels
te_y = torch.load('cifar10/te_labels.bin'):double() + 1
print(tr_x:size())
print(tr_y:size())
```

Out[1]:

```
 50000
     3
    32
    32
[torch.LongStorage of size 4]

 50000
[torch.LongStorage of size 1]
```

In [2]:

```
-- display the first 36 training set images
require 'image';
itorch.image(tr_x[{{1,36},{},{},{}}])
```



Notebook

```
function Linear:__init()
    self.output = nil
    self.gradInput = torch.zeros(10):float()
    self.W = torch.randn(10, 3*32*32):float()*0.01
    self.b = torch.randn(10):float()*0.01
    self.gradW = torch.zeros(10, 3*32*32):float()
    self.gradb = torch.zeros(10):float()
end
```

```
function init_model()
    -- define the model and criterion
    model = Linear.new()
    criterion = CEC.new()
    bestmodel = Linear.new()
end
```

Initialize model, create the state variables for the Linear Layer

Notebook

```
function train_and_test_loop(no_iterations, lr, lambda)
    for i = 0, no_iterations do
        -- trainin input and target
        idx = shuffle[mod(i, tr_x:size(1)) + 1]
        xi = get_xi(tr_x, idx)
        ti = tr_y[idx]

        -- Train
        -- do forward of the model, compute loss
        -- and then do backward of the model
        op = model:forward(xi)
        loss_tr = criterion:forward(op, ti, model, lambda)
        dl_do = criterion:backward(op, ti)
        model:backward(xi, dl_do)
        epochloss_tr = epochloss_tr + loss_tr

        -- Test
        idx = shuffle_te[mod(i, te_x:size(1)) + 1]
        xi = get_xi(te_x, idx)
        ti = te_y[idx]
        -- Compute loss
        op = model:forward(xi)
        loss_te = criterion:forward(op, ti, model, lambda)
        epochloss_te = epochloss_te + loss_te

        -- udapte model weights
        gradient_descent(model, lr)

        if mod(i, 500) == 0 then
            err = evaluate(model, tr_x, tr_y)
            print('iter: '..i.. ', accuracy: '..(1 - err)*100 ..'%  Loss: '..epochloss_tr/100)
            epochloss_te = 0
            epochloss_tr = 0

            if (err < besterr) then
                besterr = err
                bestmodel:copy(model)
                print(' -- best accuracy achieved: '.. (1- besterr)*100 ..'%')
            end
            collectgarbage()
        end
    end
    return (1 - besterr)*100 -- Accuracy
end
```

The training and test function.

It does no_iterations on data based on lr and lambda and returns the accuracy of the classifier.

Double check that the loss is reasonable:

```
-- trainin input and target
idx = shuffle[mod(i, tr_x:size(1)) + 1]
xi = get_xi(tr_x, idx)
ti = tr_y[idx]
-- do forward of the model, compute loss
-- and then do backward of the model
op = model:forward(xi)
loss_tr = criterion:forward(op, ti, model, lambda)
print(loss_tr)
```

Run for single iteration, print loss

```
-- run it
lr = 0.00001
lambda = 0.0
train_and_test_loop(1, lr, lambda)
```

disable regularization

```
Out[11]: 2.2656910718829
```

loss ~2.3. "correct" for 10 classes

Print Loss

Double check that the loss is reasonable:

```
-- Train
-- do forward of the model, compute loss
-- and then do backward of the model
op = model:forward(xi)
loss_tr = criterion:forward(op, ti, model, lambda)
dl_do = criterion:backward(op, ti)
model:backward(xi, dl_do)
epochloss_tr = epochloss_tr + loss_tr

print(loss_tr)
```

Run for single iteration, print loss

```
-- run it
lr = 0.00001
lambda = 1e3
train_and_test_loop(1, lr, lambda)
```

Crank it way up regularization

```
Out[12]: 12.582525307612
```

Print Loss

loss went up, good. (sanity check)

Lets try to train now…

Tip: Make sure that you can overfit very small portion of the training data

```
-- run it
lr = 0.0001
lambda = 0
train_and_test_loop(100000, lr, lambda)
```

```
tr_x = tr_x[{{1,20},{},{},{}}]
te_x = tr_x[{{1,20},{},{},{}}]
tr_y = tr_y[{{1,20}}]
print(tr_x:size())
print(tr_y:size())
```

```
Out[14]:  20
           3
          32
          32
         [torch.LongStorage of size 4]

          20
         [torch.LongStorage of size 1]
```

In the code here:

- take the first 20 examples from CIFAR-10
- turn off regularization (reg = 0.0)
- use vanilla 'sgd'

Lets try to train now…

Tip: Make sure that you can overfit very small portion of the training data

```
-- run it
lr = 0.0001
lambda = 0
train_and_test_loop(100000, lr, lambda)
```

```
Out[54]: iter: 500, accuracy: 20%  Loss: 6.4891701533306
          -- best accuracy achieved: 100%
Out[54]: iter: 1000, accuracy: 100%  Loss: 3.363490690347
Out[54]: iter: 1500, accuracy: 100%  Loss: 2.3995975677242
Out[54]: iter: 2000, accuracy: 100%  Loss: 1.8909617506362
Out[54]: iter: 2500, accuracy: 100%  Loss: 1.5617572159784
Out[54]: iter: 3000, accuracy: 100%  Loss: 1.3375534142717
Out[54]: iter: 3500, accuracy: 100%  Loss: 1.1668484200641
Out[54]: iter: 4000, accuracy: 100%  Loss: 1.0398030826978
```

```
Out[54]: iter: 98000, accuracy: 100%  Loss: 0.075056174474324
Out[54]: iter: 98500, accuracy: 100%  Loss: 0.074695131101785
Out[54]: iter: 99000, accuracy: 100%  Loss: 0.074675841566382
Out[54]: iter: 99500, accuracy: 100%  Loss: 0.074908872365756
Out[54]: iter: 100000, accuracy: 100%  Loss: 0.074439254969025
```

Very small loss, train accuracy 100, nice!

Lets try to train now…

I like to start with small regularization and find learning rate that makes the loss go down.

```
-- run it
lr = 1e-7
lambda = 1e-7
train_and_test_loop(10000, lr, lambda)
```

Lets try to train now…

I like to start with small regularization and find learning rate that makes the loss go down.

```
-- run it
lr = 1e-7
lambda = 1e-7
train_and_test_loop(10000, lr, lambda)
```

```
Out[18]: iter: 0, accuracy: 10%  Loss: 0.023248429529449
          -- best accuracy achieved: 10%

Out[18]: iter: 500, accuracy: 10%  Loss: 11.522416713458

Out[18]: iter: 1000, accuracy: 10%  Loss: 11.517536122735

Out[18]: iter: 1500, accuracy: 11%  Loss: 11.508510566527
          -- best accuracy achieved: 11%

Out[18]: iter: 2000, accuracy: 13%  Loss: 11.510842908524
          -- best accuracy achieved: 13%

Out[18]: iter: 2500, accuracy: 13%  Loss: 11.501224886344

Out[18]: iter: 3000, accuracy: 14%  Loss: 11.49398984774
          -- best accuracy achieved: 14%

Out[18]: iter: 3500, accuracy: 16%  Loss: 11.487628759524
          -- best accuracy achieved: 16%

Out[18]: iter: 4000, accuracy: 17%  Loss: 11.492140238992
          -- best accuracy achieved: 17%
```

Loss barely changing: Learning rate is probably too low

Lets try to train now…

I like to start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low

```
-- run it
lr = 1e-7
lambda = 1e-7
train_and_test_loop(10000, lr, lambda)
```

```
Out[18]: iter: 0, accuracy: 10%  Loss: 0.023248429529449
          -- best accuracy achieved: 10%

Out[18]: iter: 500, accuracy: 10%  Loss: 11.522416713458

Out[18]: iter: 1000, accuracy: 10%  Loss: 11.517536122735

Out[18]: iter: 1500, accuracy: 11%  Loss: 11.508510566527
          -- best accuracy achieved: 11%

Out[18]: iter: 2000, accuracy: 13%  Loss: 11.510842908524
          -- best accuracy achieved: 13%

Out[18]: iter: 2500, accuracy: 13%  Loss: 11.501224886344

Out[18]: iter: 3000, accuracy: 14%  Loss: 11.49398984774
          -- best accuracy achieved: 14%

Out[18]: iter: 3500, accuracy: 16%  Loss: 11.487628759524
          -- best accuracy achieved: 16%

Out[18]: iter: 4000, accuracy: 17%  Loss: 11.492140238992
          -- best accuracy achieved: 17%
```

Loss barely changing: Learning rate is probably too low

Lets try to train now…

I like to start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low

```
-- run it
lr = 1e-7
lambda = 1e-7
train_and_test_loop(10000, lr, lambda)
```

```
Out[18]: iter: 0, accuracy: 10%  Loss: 0.023248429529449
          -- best accuracy achieved: 10%

Out[18]: iter: 500, accuracy: 10%  Loss: 11.522416713458

Out[18]: iter: 1000, accuracy: 10%  Loss: 11.517536122735

Out[18]: iter: 1500, accuracy: 11%  Loss: 11.508510566527
          -- best accuracy achieved: 11%

Out[18]: iter: 2000, accuracy: 13%  Loss: 11.510842908524
          -- best accuracy achieved: 13%

Out[18]: iter: 2500, accuracy: 13%  Loss: 11.501224886344

Out[18]: iter: 3000, accuracy: 14%  Loss: 11.49398984774
          -- best accuracy achieved: 14%

Out[18]: iter: 3500, accuracy: 16%  Loss: 11.487628759524
          -- best accuracy achieved: 16%

Out[18]: iter: 4000, accuracy: 17%  Loss: 11.492140238992
          -- best accuracy achieved: 17%
```

Notice train/val accuracy goes to 17% though, what's up with that? (remember this is softmax)

Loss barely changing: Learning rate is probably too low

Lets try to train now…

```
-- run it
lr = 1e6
lambda = 1e-7
train_and_test_loop(10000, lr, lambda)
```

I like to start with small regularization and find learning rate that makes the loss go down.

Okay now lets try learning rate 1e6. What could possibly go wrong?

loss not going down:
learning rate too low

Lets try to train now…

I like to start with small regularization and find learning rate that makes the loss go down.

loss not going down:

learning rate too low

loss exploding:

learning rate too high

```
-- run it
lr = 1e6
lambda = 1e-7
train_and_test_loop(10000, lr, lambda)
```

```
Out[19]: iter: 0, accuracy: 11%  Loss: 0.023115084740835
          -- best accuracy achieved: 11%

Out[19]: iter: 500, accuracy: 13%  Loss: nan
          -- best accuracy achieved: 13%

Out[19]: iter: 1000, accuracy: 13%  Loss: nan

Out[19]: iter: 1500, accuracy: 13%  Loss: nan

Out[19]: iter: 2000, accuracy: 13%  Loss: nan
```

cost: NaN almost always means high learning rate...

Lets try to train now…

I like to start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low
loss exploding:
learning rate too high

```
-- run it
lr = 1e-3
lambda = 1e-7
train_and_test_loop(3000, lr, lambda)
```

```
Out[29]: iter: 0, accuracy: 20%  Loss: 0.02357119788693
          -- best accuracy achieved: 20%
Out[29]: iter: 500, accuracy: 13%  Loss: nan
Out[29]: iter: 1000, accuracy: 13%  Loss: nan
Out[29]: iter: 1500, accuracy: 13%  Loss: nan
Out[29]: iter: 2000, accuracy: 13%  Loss: nan
Out[29]: iter: 2500, accuracy: 13%  Loss: nan
Out[29]: iter: 3000, accuracy: 13%  Loss: nan
```

3e-3 is still too high. Cost explodes….

=> Rough range for learning rate we should be cross-validating is somewhere [1e-3 … 1e-7]

Hyperparameter Optimization

Cross-validation strategy

I like to do **coarse -> fine** cross-validation in stages

First stage: only a few epochs to get rough idea of what params work
Second stage: longer running time, finer search
… (repeat as necessary)

Tip for detecting explosions in the solver:
If the cost is ever > 3 * original cost, break out early

For example: run coarse search for 2000 iterations

```
for i = 1, 100 do
    init_model()
    lr = math.pow(10, torch.uniform(-7.0, -3.0))
    lambda = math.pow(10, torch.uniform(-5, 5))
    best_acc = train_and_test_loop(2000, lr, lambda)
    print(string.format("Try %d/%d Best val accuracy: %d, lr: %f, lambda: %f",i, 100, best_acc, lr, lambda))
end
```

note it’s best to optimize in log space!

```
Out[10]: Try 1/100 Best val accuracy: 16, lr: 0.000045, lambda: 4996.489302
Out[10]: Try 2/100 Best val accuracy: 31, lr: 0.000003, lambda: 0.001315
Out[10]: Try 3/100 Best val accuracy: 25, lr: 0.000001, lambda: 0.000012
Out[10]: Try 4/100 Best val accuracy: 24, lr: 0.000002, lambda: 216.397129
Out[10]: Try 5/100 Best val accuracy: 26, lr: 0.000007, lambda: 0.000012
Out[10]: Try 6/100 Best val accuracy: 29, lr: 0.000009, lambda: 275.964597
Out[10]: Try 7/100 Best val accuracy: 30, lr: 0.000021, lambda: 0.000253
Out[10]: Try 8/100 Best val accuracy: 13, lr: 0.000809, lambda: 4.339235
Out[10]: Try 9/100 Best val accuracy: 26, lr: 0.000003, lambda: 0.000062
Out[10]: Try 10/100 Best val accuracy: 27, lr: 0.000095, lambda: 18.288190
Out[10]: Try 11/100 Best val accuracy: 14, lr: 0.000000, lambda: 1333.400659
Out[10]: Try 12/100 Best val accuracy: 8, lr: 0.000311, lambda: 0.000020
Out[10]: Try 13/100 Best val accuracy: 8, lr: 0.000617, lambda: 0.000050
Out[10]: Try 14/100 Best val accuracy: 34, lr: 0.000013, lambda: 0.124955
Out[10]: Try 15/100 Best val accuracy: 17, lr: 0.000013, lambda: 5262.631955
```

Nice, 34% with only 2000 iterations

Now run finer search...

```
for i = 1, 100 do
    init_model()
    lr = math.pow(10, torch.uniform(-7.0, -3.0))
    lambda = math.pow(10, torch.uniform(-5, 5))
    best_acc = train_and_test_loop(2000, lr, lambda)
    print(string.format("Try %d/%d Best val accuracy: %d, lr
end
```

adjust range

```
for i = 1, 100 do
    init_model()
    lr = math.pow(10, torch.uniform(-6.0, -4.0))
    lambda = math.pow(10, torch.uniform(-3, 1))
    best_acc = train_and_test_loop(2000, lr, lambda)
    print(string.format("Try %d/%d Best val accuracy: %d,
end
```

```
Out[11]: Try 1/100 Best val accuracy: 35, lr: 0.000055, lambda: 0.002026
Out[11]: Try 2/100 Best val accuracy: 28, lr: 0.000001, lambda: 1.994656
Out[11]: Try 3/100 Best val accuracy: 32, lr: 0.000003, lambda: 0.483409
Out[11]: Try 4/100 Best val accuracy: 37, lr: 0.000032, lambda: 1.981563
Out[11]: Try 5/100 Best val accuracy: 27, lr: 0.000003, lambda: 0.004578
Out[11]: Try 6/100 Best val accuracy: 28, lr: 0.000004, lambda: 0.082862
Out[11]: Try 7/100 Best val accuracy: 34, lr: 0.000020, lambda: 0.003083
Out[11]: Try 8/100 Best val accuracy: 28, lr: 0.000054, lambda: 0.064499
Out[11]: Try 9/100 Best val accuracy: 31, lr: 0.000003, lambda: 0.004361
Out[11]: Try 10/100 Best val accuracy: 32, lr: 0.000004, lambda: 0.001610
Out[11]: Try 11/100 Best val accuracy: 31, lr: 0.000006, lambda: 0.300821
```

37% - relatively good for a 1-layer neural net and only 2000 iterations (we are getting see only 2000/50000 = 4% of our training data!

Now run finer search...

```
for i = 1, 100 do
    init_model()
    lr = math.pow(10, torch.uniform(-7.0, -3.0))
    lambda = math.pow(10, torch.uniform(-5, 5))
    best_acc = train_and_test_loop(2000, lr, lambda)
    print(string.format("Try %d/%d Best val accuracy: %d, lr
end
```

adjust range

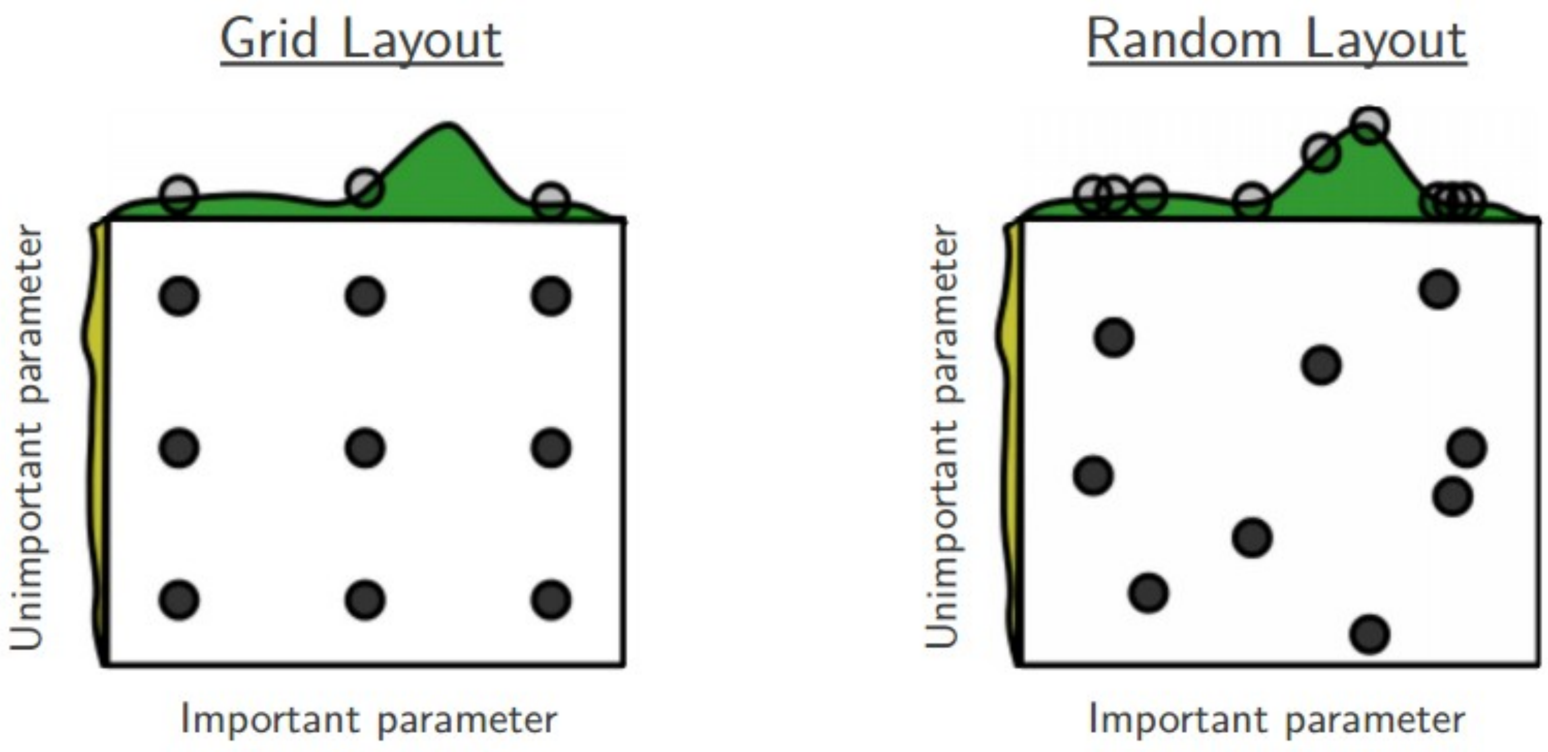
```
for i = 1, 100 do
    init_model()
    lr = math.pow(10, torch.uniform(-6.0, -4.0))
    lambda = math.pow(10, torch.uniform(-3, 1))
    best_acc = train_and_test_loop(2000, lr, lambda)
    print(string.format("Try %d/%d Best val accuracy: %d,
end
```

```
Out[11]: Try 1/100 Best val accuracy: 35, lr: 0.000055, lambda: 0.002026
Out[11]: Try 2/100 Best val accuracy: 28, lr: 0.000001, lambda: 1.994656
Out[11]: Try 3/100 Best val accuracy: 32, lr: 0.000003, lambda: 0.483409
Out[11]: Try 4/100 Best val accuracy: 37, lr: 0.000032, lambda: 1.981563
Out[11]: Try 5/100 Best val accuracy: 27, lr: 0.000003, lambda: 0.004578
Out[11]: Try 6/100 Best val accuracy: 28, lr: 0.000004, lambda: 0.082862
Out[11]: Try 7/100 Best val accuracy: 34, lr: 0.000020, lambda: 0.003083
Out[11]: Try 8/100 Best val accuracy: 28, lr: 0.000054, lambda: 0.064499
Out[11]: Try 9/100 Best val accuracy: 31, lr: 0.000003, lambda: 0.004361
Out[11]: Try 10/100 Best val accuracy: 32, lr: 0.000004, lambda: 0.001610
Out[11]: Try 11/100 Best val accuracy: 31, lr: 0.000006, lambda: 0.300821
```

37% - relatively good for a 1-layer neural net and only 2000 iterations

Make sure the best ones are not on the boundary

Random Search vs. Grid Search



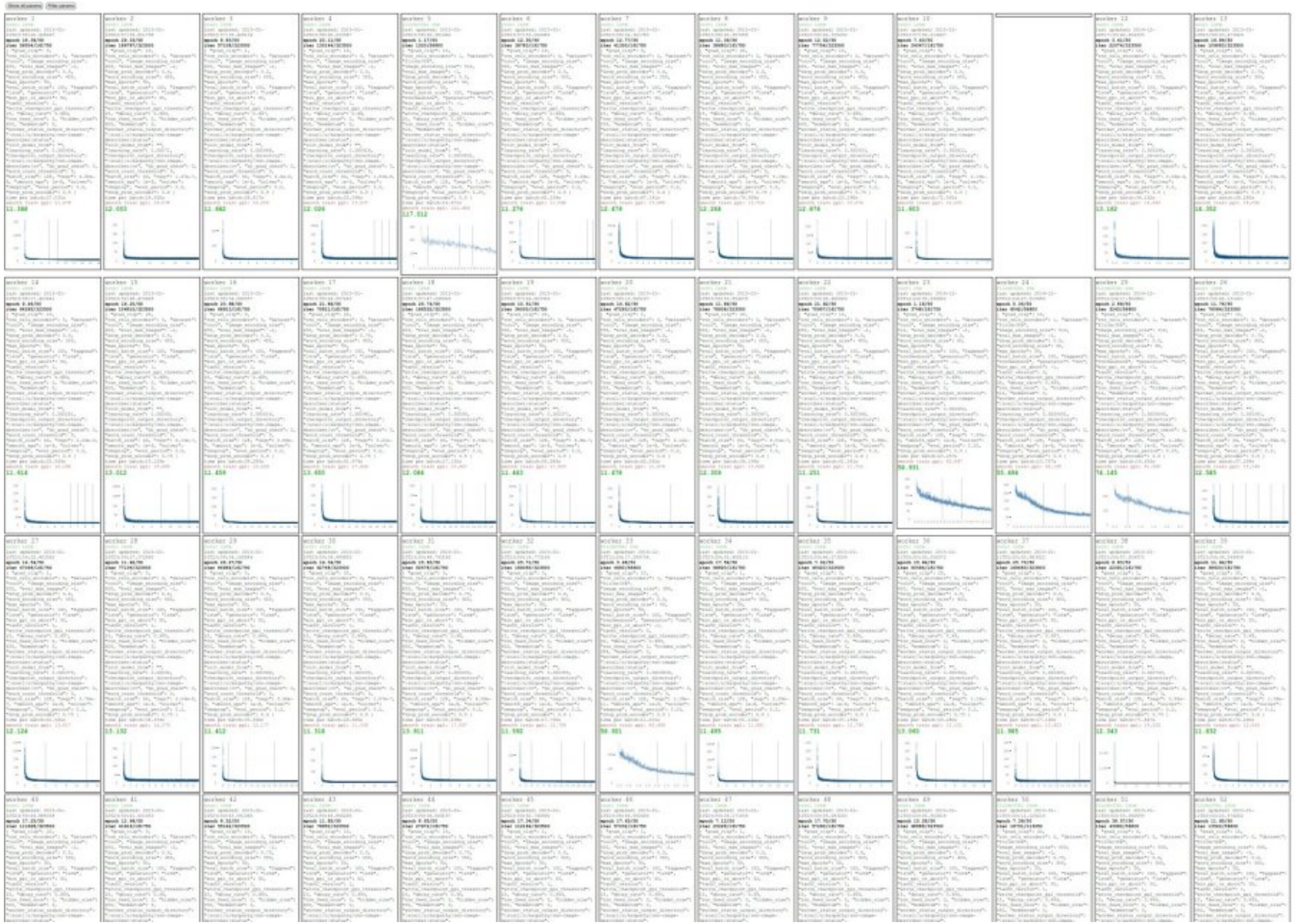
Random Search for Hyper-Parameter Optimization
Bergstra and Bengio, 2012

Hyperparameters to play with:

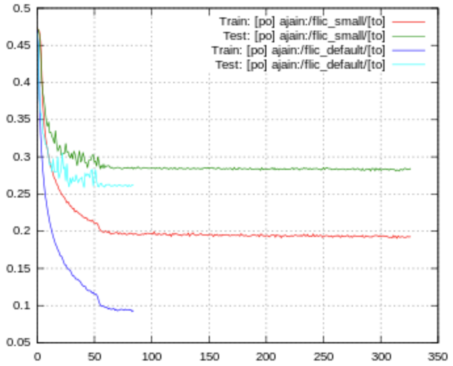
- network architecture
- learning rate, its decay schedule, update type
- regularization (L2/Dropout strength)

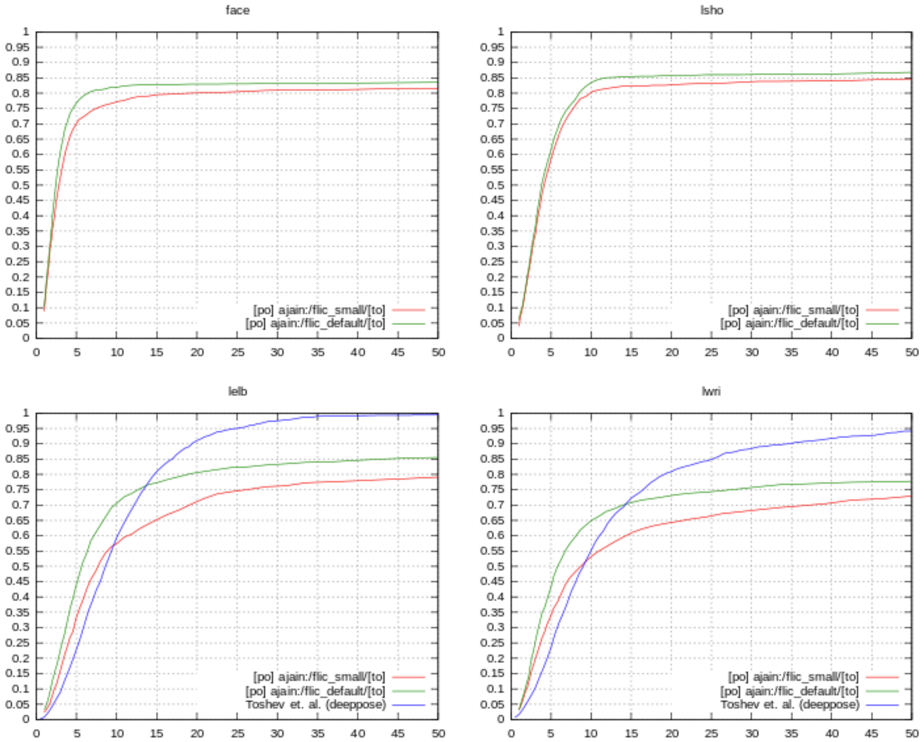
neural networks practitioner
music = loss function

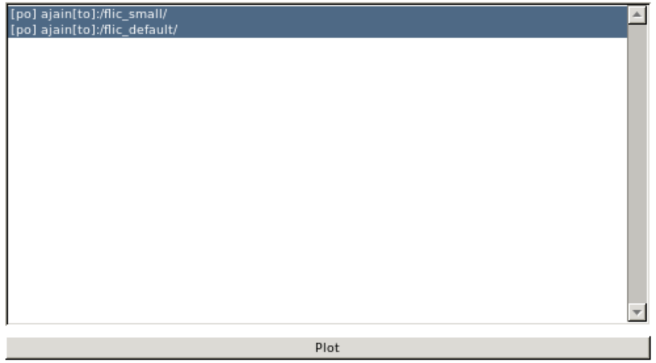
Karpathy's cross-validation "command center"



My cross-validation "command center"







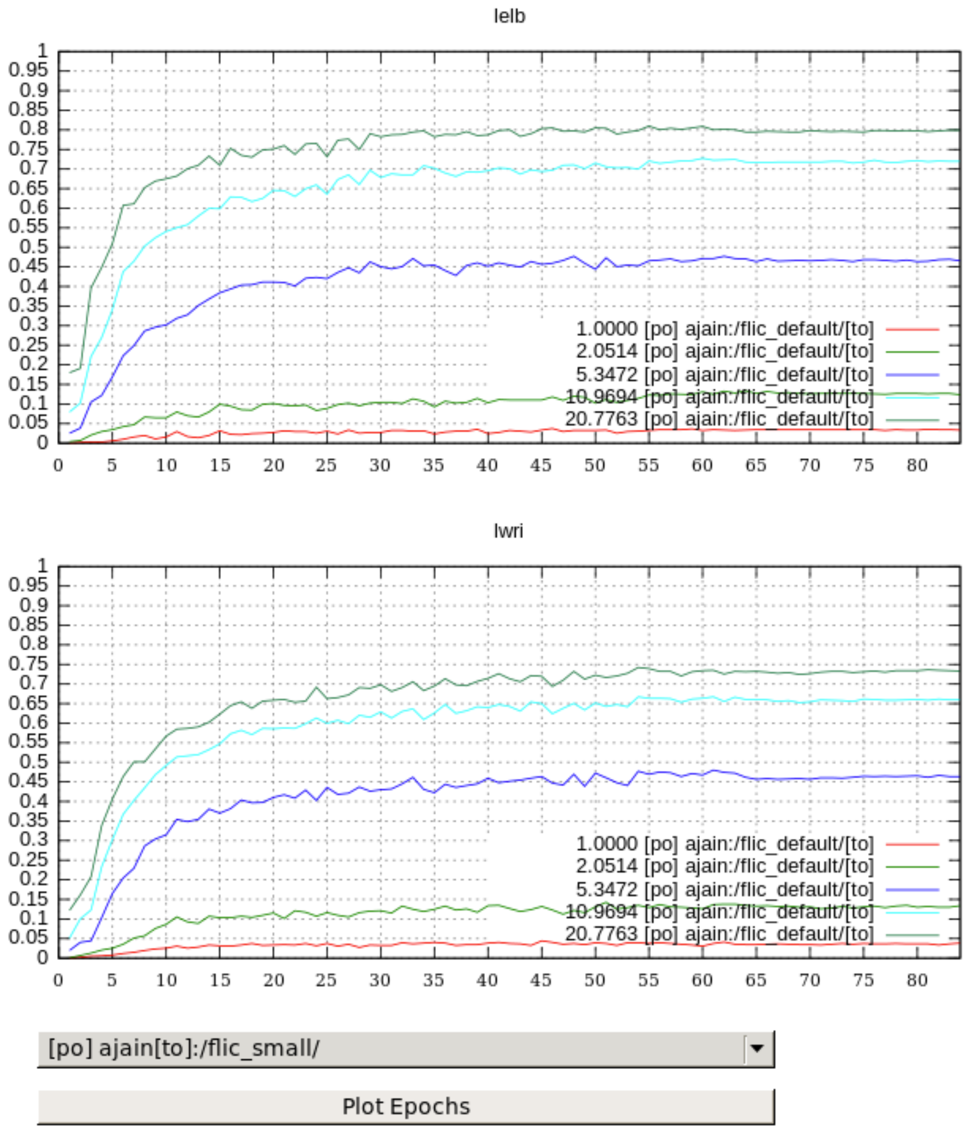
[po] ajain:/flic_small/[to]

```
batch_normalization = false
batch_size = 16
batch_size_per_gpu = 16
big_model = false
body_scale_range = {}
compress_src_model =
conv_1x1_size = 1
conv_nfeats = { 1 = { 1 = 16, 2 = 16, 3 = 16, }, 2 = { 1 = 16, 2 = 16, 3 = 16, }, 3 = { 1 = 16, 2 = 16, 3 = 16, },}
conv_pool = { 1 = { 1 = 2, 2 = 2, 3 = 1, }, 2 = { 1 = 2, 2 = 2, 3 = 1, }, 3 = { 1 = 2, 2 = 2, 3 = 1, },}
conv_size = { 1 = { 1 = 5, 2 = 5, 3 = 5, }, 2 = { 1 = 5, 2 = 5, 3 = 5, }, 3 = { 1 = 5, 2 = 5, 3 = 5, },}
crop_gradOutput = false
```

[po] ajain:/flic_default/[to]

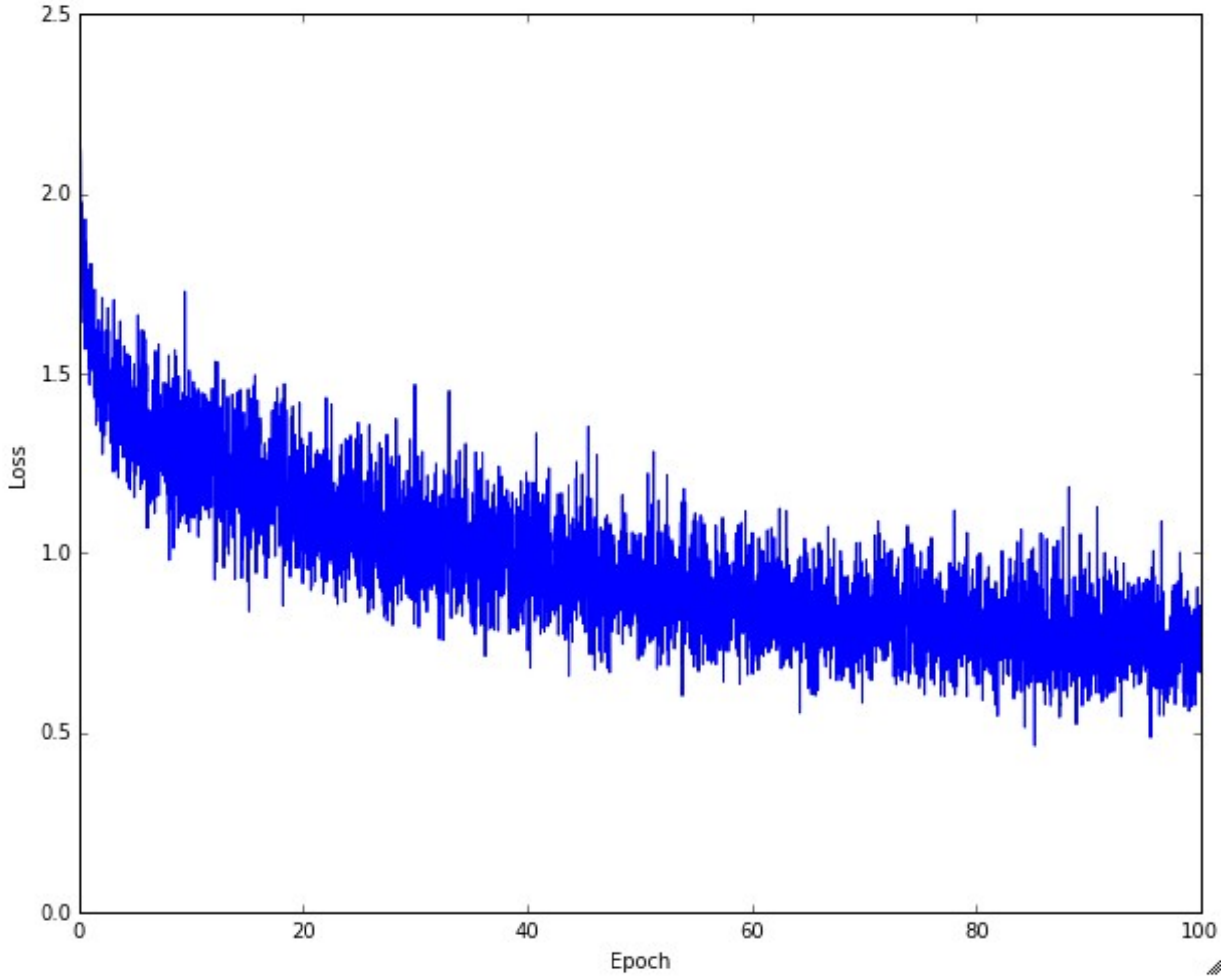
```
batch_normalization = false
batch_size = 16
batch_size_per_gpu = 16
big_model = true
body_scale_range = {}
compress_src_model =
conv_1x1_size = 1
conv_nfeats = { 1 = { 1 = 128, 2 = 128, 3 = 128, }, 2 = { 1 = 128, 2 = 128, 3 = 128, }, 3 = { 1 = 128, 2 = 128, 3 = 128, },}
conv_pool = { 1 = { 1 = 2, 2 = 2, 3 = 1, }, 2 = { 1 = 2, 2 = 2, 3 = 1, }, 3 = { 1 = 2, 2 = 2, 3 = 1, },}
conv_size = { 1 = { 1 = 5, 2 = 5, 3 = 5, }, 2 = { 1 = 5, 2 = 5, 3 = 5, }, 3 = { 1 = 5, 2 = 5, 3 = 5, },}
crop_gradOutput = false
```

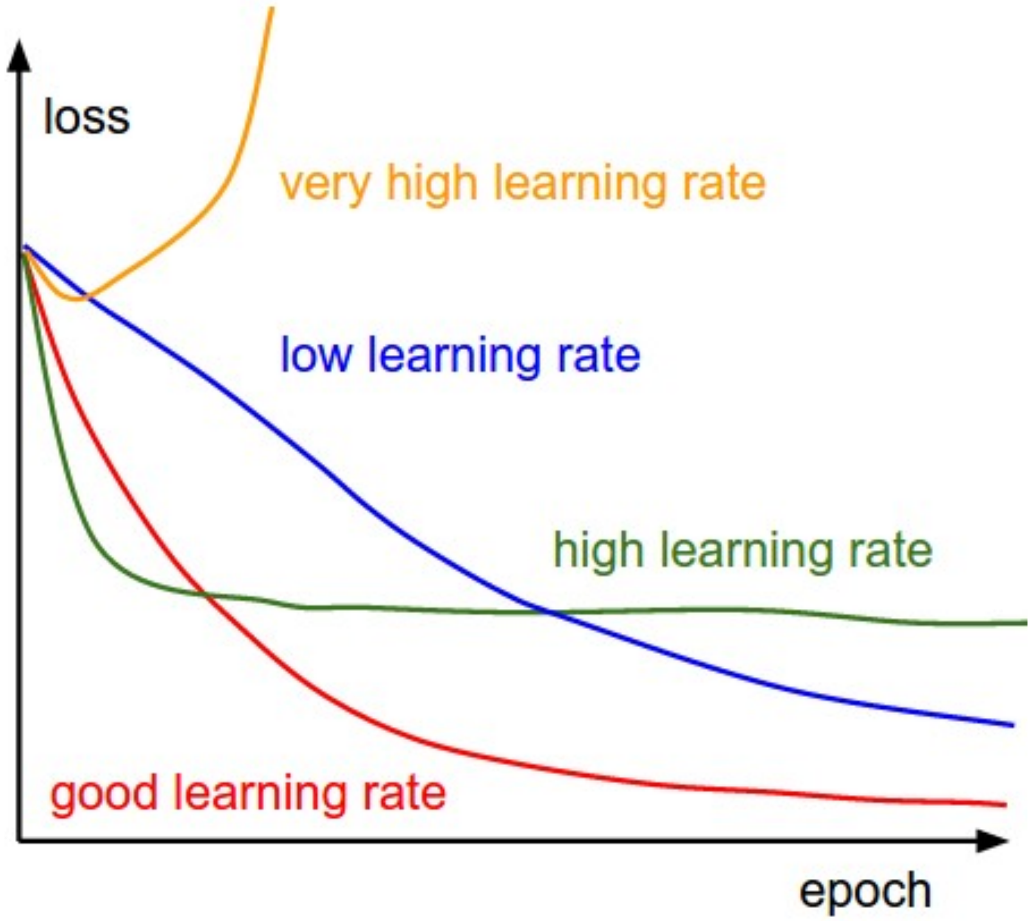
My cross-validation "command center"

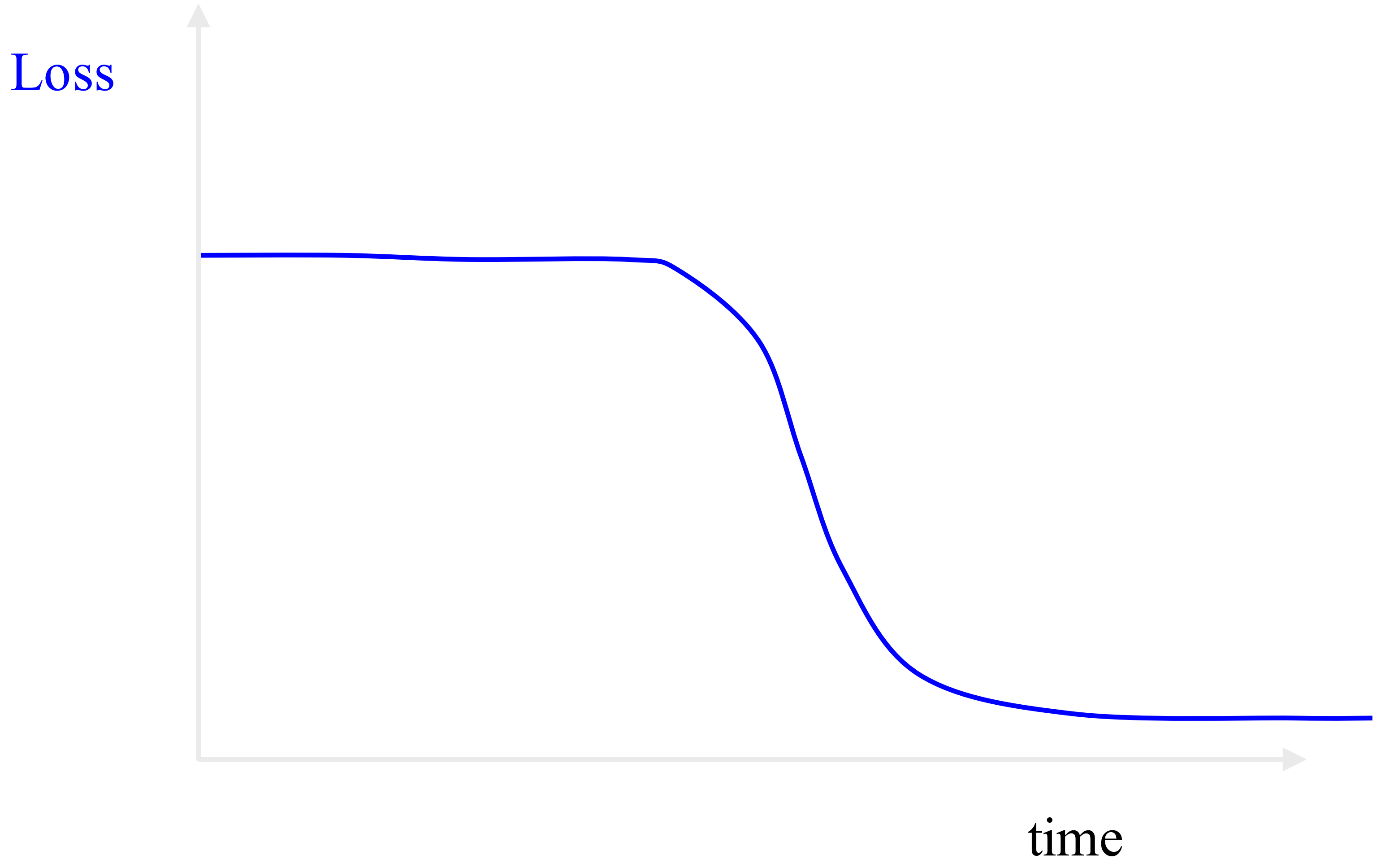


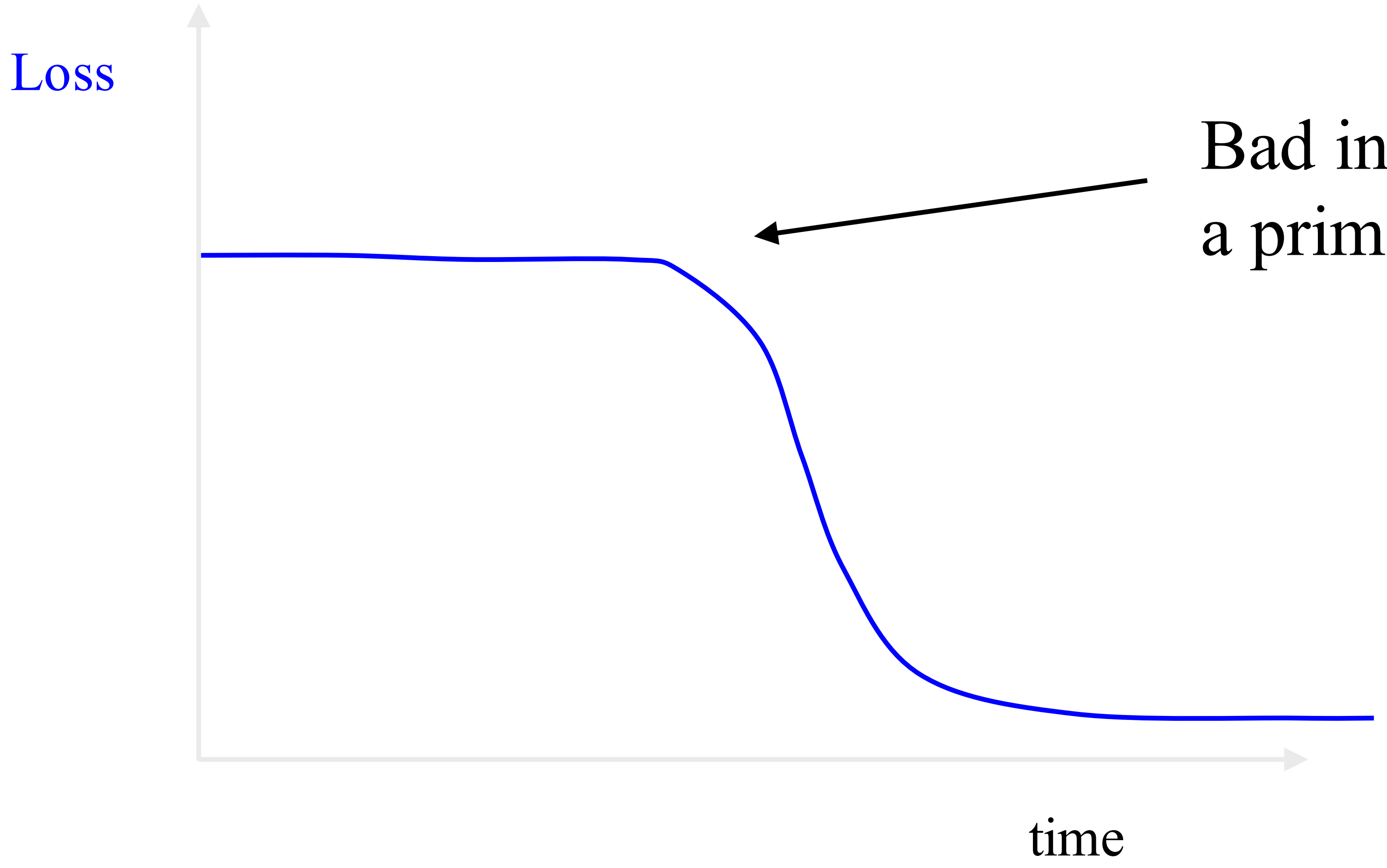
My cross-validation “command center”

Monitor and visualize the loss curve

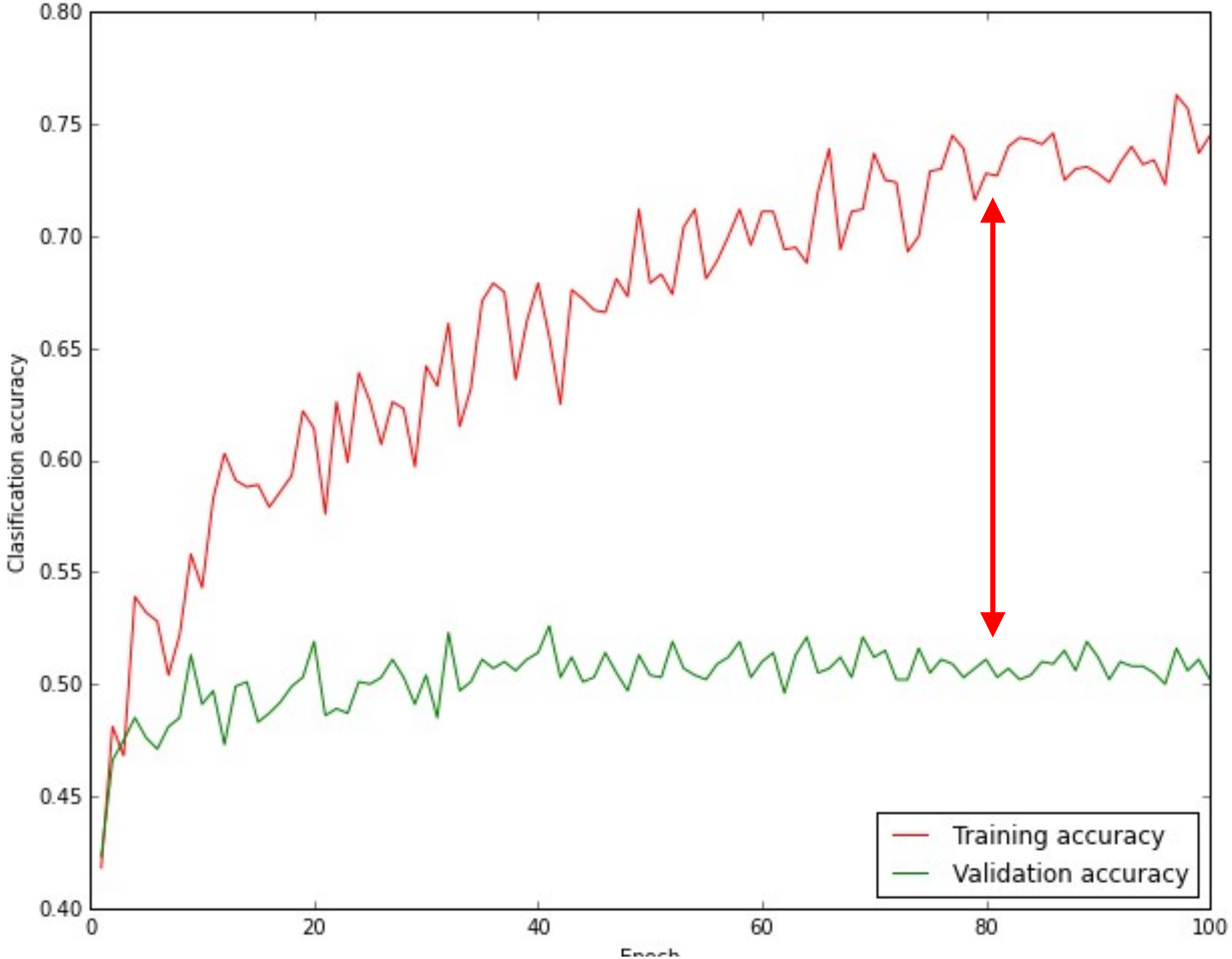





Loss
time


Loss
Bad initialization
a prime suspect
time

Monitor and visualize the accuracy:



big gap = overfitting

=> increase regularization strength?

no gap

=> increase model capacity?

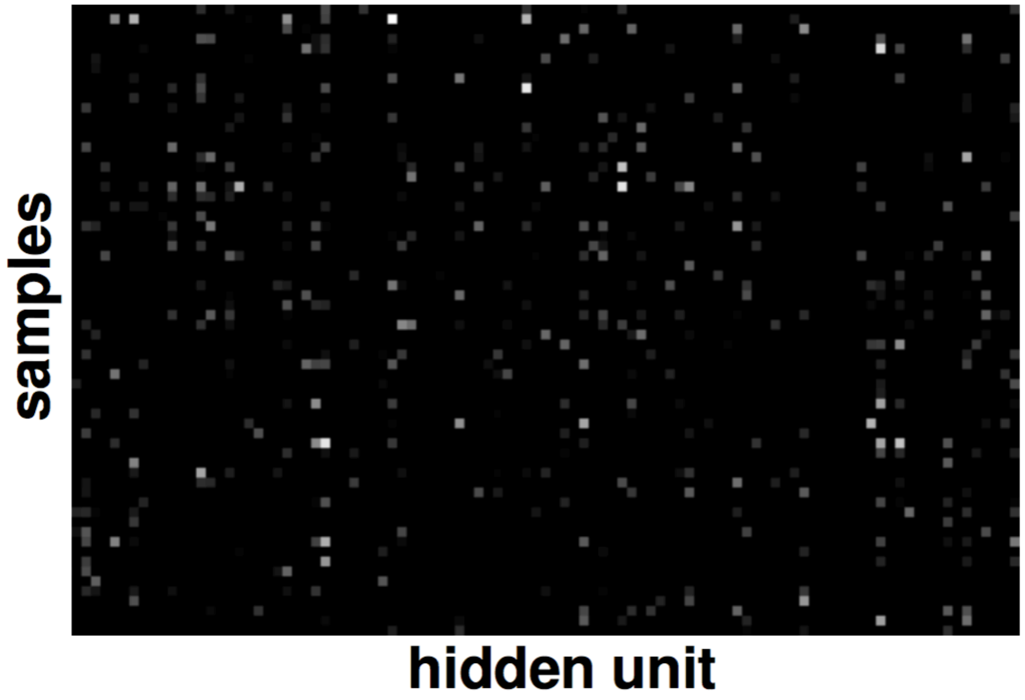
Track the ratio of weight updates / weight magnitudes:

```
function gradient_descent(model, lr)
    w_scale = torch.norm(model.W:view(model.W:nElement()), 2, 1)
    update_scale = torch.norm(lr * model.gradW:view(model.gradW:nElement()), 2, 1)
    model.W = model.W + lr * model.gradW
    model.b = model.b + lr * model.gradb
    print(update_scale/w_scale) -- Want ~1e-3
end
```

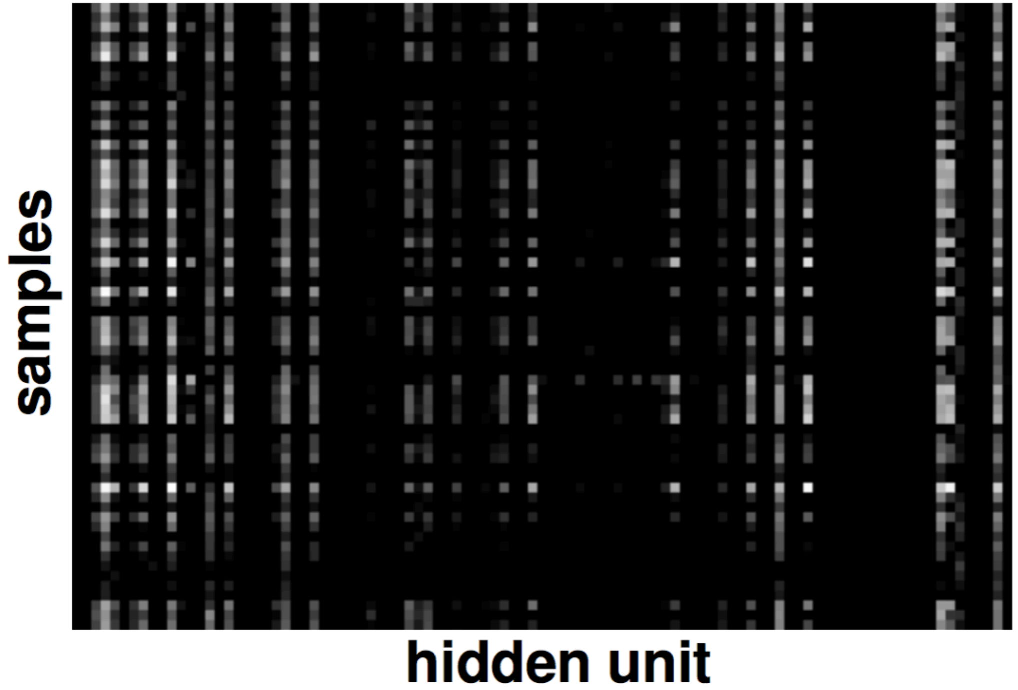
ratio between the values and updates:
want this to be somewhere around 0.001 or so

Visualize Features

- Visualize features (feature maps need to be uncorrelated) and have high variance.



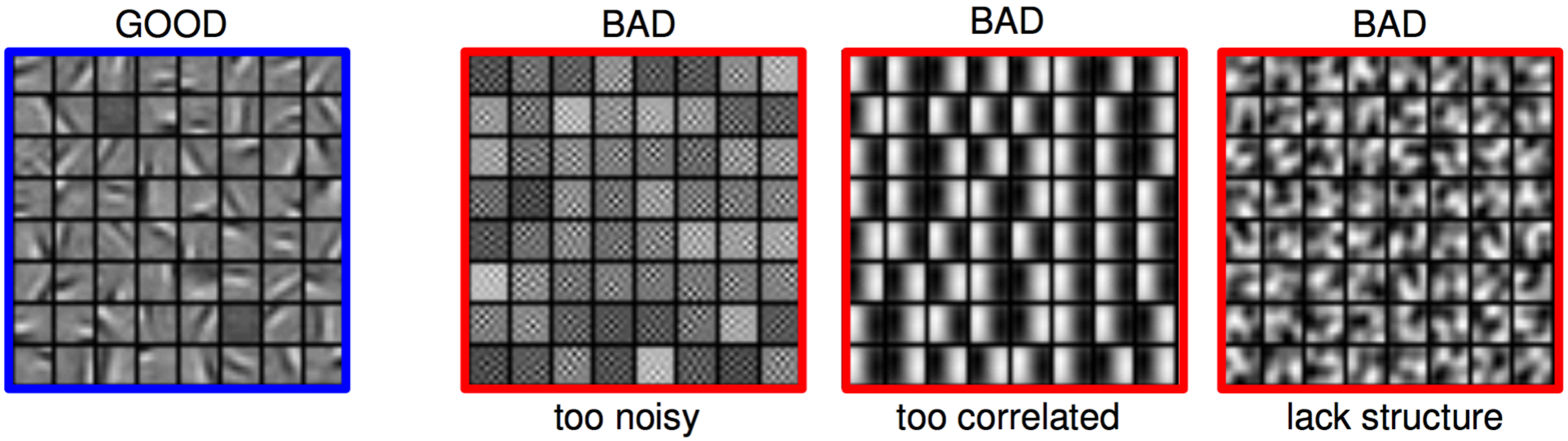
Good training: hidden units are sparse across samples and across features.



Bad training: many hidden units ignore the input and/or exhibit strong correlations.

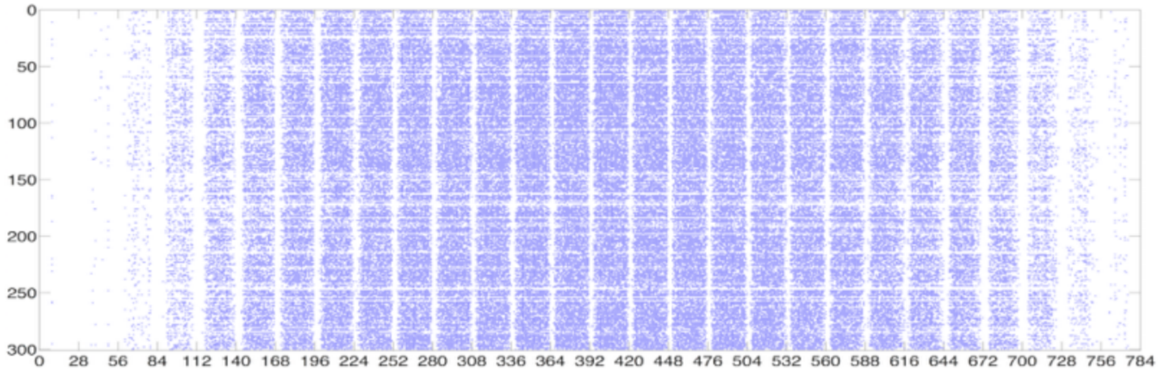
Visualize Weights (Conv Layer)

- Good training: learned filters exhibit structure and are uncorrelated.



Visualize Weights (Fully Connected Layer)

- Sparsity is natural in deep learning
- Visualization of the first FC layer’s sparsity pattern of LeNet
- It has a banded structure repeated 28 times (why? Hint: images are 28x28)



Thank you!