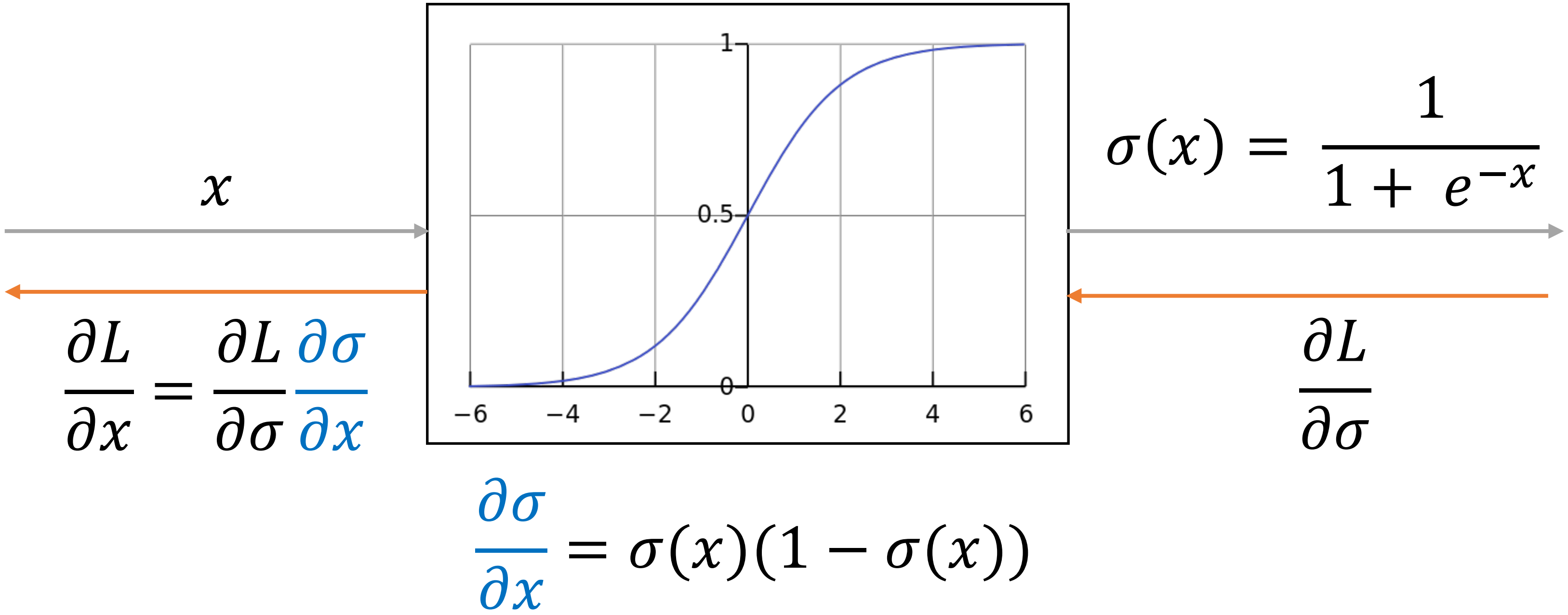
MML minor #5

Свёрточные нейронные сети

Sigmoid активация



x

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial \sigma} \frac{\partial \sigma}{\partial x}$$

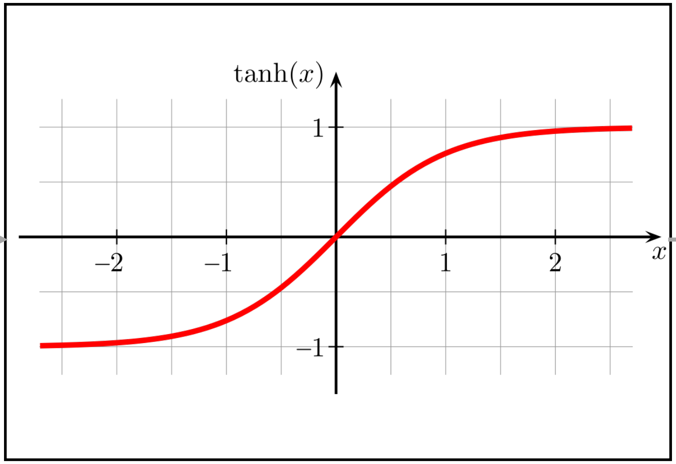
$$\frac{\partial L}{\partial \sigma}$$

$$\frac{\partial \sigma}{\partial x} = \sigma(x)(1 - \sigma(x))$$

- Нейроны с сигмоидой могут насыщаться и приводить к **угасающим градиентам**.
- Не центрированы в нуле.
- e^x дорого вычислять.

Tanh активация

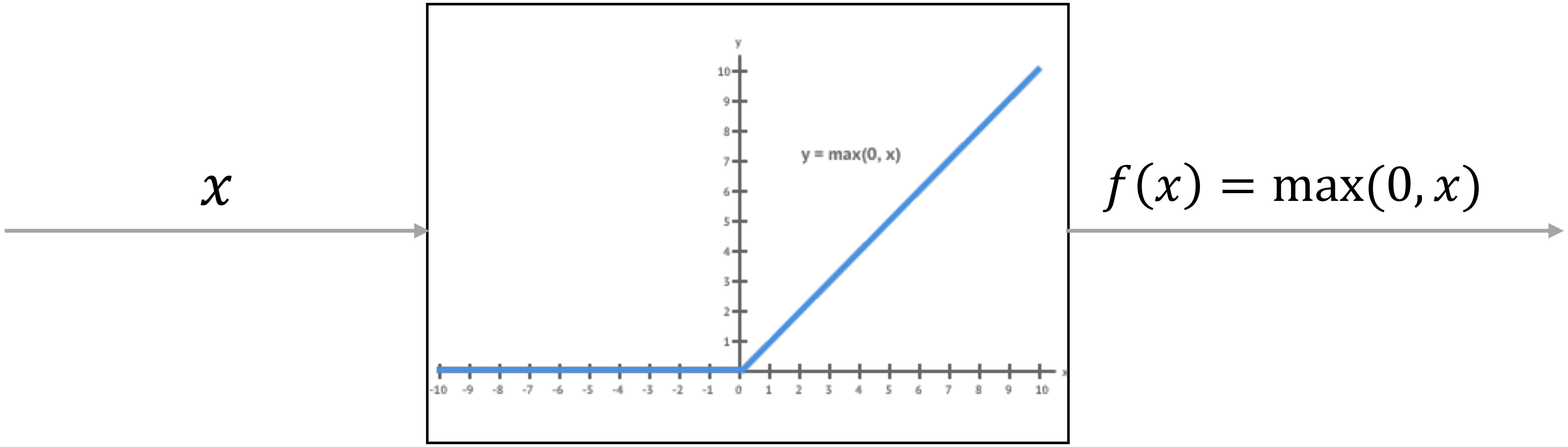
x



$$\tanh(x) = \frac{2}{1 + e^{-2x}} - 1$$

- Центрирован в нуле.
- Но все еще как сигмоида.

ReLU активация

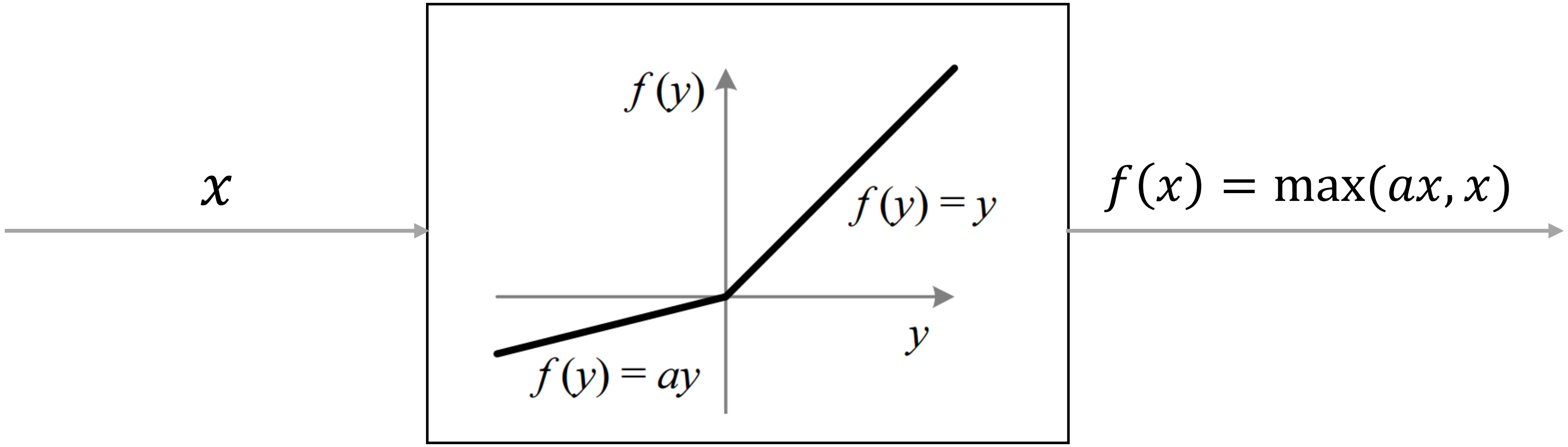


x

$f(x) = \max(0, x)$

- Быстро считается.
- Градиенты не угасают при $x > 0$.
- На практике ускоряет сходимость!
- Не центрирован в нуле.
- Могут умереть: если не было активации - не будет обновления!

Leaky ReLU активация

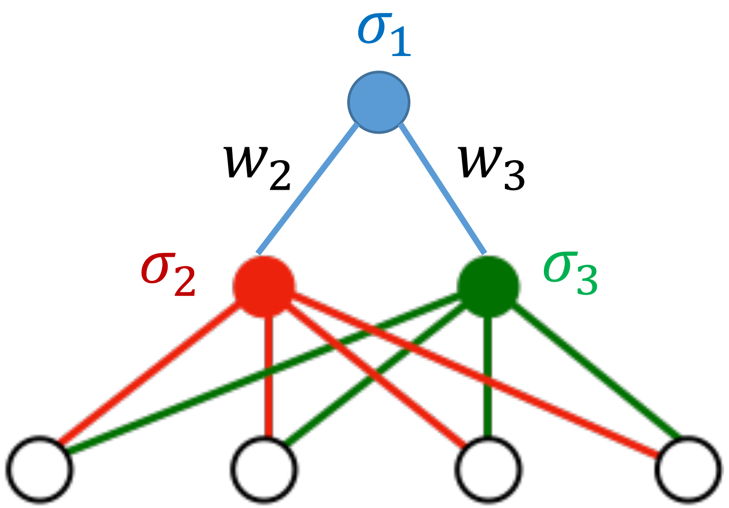


$f(x) = \max(ax, x)$

- Всегда будут обновления!
- $a \neq 1$

Инициализация весов

Давайте начнем с нулей?



Входы:

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial \sigma_1}\sigma_1(1-\sigma_1)\sigma_2$$

$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial \sigma_1}\sigma_1(1-\sigma_1)\sigma_3$$

σ_2 и σ_3 обновляются одинаково!

- Нужно сломать симметрию!
- Может случайным шумом?
- Но насколько большим? $0.03 \cdot \mathcal{N}(0,1)$?

Инициализация весов

- Линейные модели любят когда входы нормализованы.
- Нейрон это линейная комбинация входов + активация.
- Выход нейрона будет использован следующими слоями.

Weights initializations

- Let's look at the neuron output before activation: $\sum_{i=1}^{n} x_i w_i$.

- If $E(x_i) = E(w_i) = 0$ and we generate weights independently from inputs, then $E(\sum_{i=1}^{n} x_i w_i) = 0$.

- But variance can grow with consecutive layers.

- Empirically this hurts convergence for deep networks!

Weights initializations

- Let’s look at the variance of $\sum_{i=1}^{n} x_i w_i$:

Weights initializations

- Let’s look at the variance of $\sum_{i=1}^{n} x_i w_i$:

$$Var(\sum_{i=1}^{n} x_i w_i) =$$

i.i.d. w_i and mostly uncorrelated x_i

$$= \sum_{i=1}^{n} Var(x_i w_i) =$$

Weights initializations

- Let’s look at the variance of $\sum_{i=1}^{n} x_i w_i$:

$$Var(\sum_{i=1}^{n} x_i w_i) =$$

i.i.d. w_i and mostly uncorrelated x_i

$$= \sum_{i=1}^{n} Var(x_i w_i) =$$

independent factors w_i and x_i

$$= \sum_{i=1}^{n} \begin{pmatrix} [E(x_i)]^2 Var(w_i) \\ +[E(w_i)]^2 Var(x_i) \\ +Var(x_i) Var(w_i) \end{pmatrix} =$$

Weights initializations

- Let's look at the variance of $\sum_{i=1}^{n} x_i w_i$:

$$Var(\sum_{i=1}^{n} x_i w_i) =$$

i.i.d. w_i and mostly uncorrelated x_i

$$= \sum_{i=1}^{n} Var(x_i w_i) =$$

independent factors w_i and x_i

$$= \sum_{i=1}^{n} \begin{pmatrix} [E(x_i)]^2 Var(w_i) \\ +[E(w_i)]^2 Var(x_i) \\ +Var(x_i)Var(w_i) \end{pmatrix} =$$

w_i and x_i have 0 mean

$$= \sum_{i=1}^{n} Var(x_i)Var(w_i) = Var(x)[n\,Var(w)]$$

Weights initializations

- Let’s look at the variance of $\sum_{i=1}^{n} x_i w_i$:

$$Var\left(\sum_{i=1}^{n} x_i w_i\right) =$$

i.i.d. w_i and mostly uncorrelated x_i

$$= \sum_{i=1}^{n} Var(x_i w_i) =$$

independent factors w_i and x_i

$$= \sum_{i=1}^{n} \begin{pmatrix} [E(x_i)]^2 Var(w_i) \\ +[E(w_i)]^2 Var(x_i) \\ +Var(x_i)Var(w_i) \end{pmatrix} =$$

w_i and x_i have 0 mean

$$= \sum_{i=1}^{n} Var(x_i)Var(w_i) = Var(x)[n\, Var(w)]$$

↑

We want this to be 1

Weights initializations

- Let’s use the fact that $Var(aw) = a^2 Var(w)$.

- For $[n\,Var(aw)]$ to be 1
  we need to multiply $\mathcal{N}(0{,}1)$ weights ($Var(w) = 1$)
  by $a = 1/\sqrt{n}$.

- Xavier initialization (Glorot et al.)
  multiplies weights by $\sqrt{2}/\sqrt{n_{in} + n_{out}}$.

- Initialization for ReLU neurons (He et al.)
  uses multiplication by $\sqrt{2}/\sqrt{n_{in}}$.

Batch normalization

- We know how to initialize our network to constrain variance.
- But what if it grows during backpropagation?
- Batch normalization controls mean and variance of outputs **before activations**.

Batch normalization

- Let’s normalize h_i – neuron output before activation:

$$h_i = \gamma_i \boxed{\frac{h_i - \mu_i}{\sqrt{\sigma_i^2}}} + \beta_i$$

→ 0 mean, unit variance

Batch normalization

- Let’s normalize h_i – neuron output before activation:

$$h_i = \gamma_i \boxed{\frac{h_i - \mu_i}{\sqrt{\sigma_i^2}}} + \beta_i$$

→ 0 mean, unit variance

- Where do μ_i and σ_i^2 come from? We can estimate them having a **current training batch**!

Batch normalization

- Let’s normalize h_i – neuron output before activation:

$$h_i = \gamma_i \boxed{\frac{h_i - \mu_i}{\sqrt{\sigma_i^2}}} + \beta_i$$

→ 0 mean, unit variance

- Where do μ_i and σ_i^2 come from? We can estimate them having a **current training batch**!
- During testing we will use an exponential moving average over train batches:

$$0 < \alpha < 1 \qquad \begin{aligned} \mu_i &= \alpha \cdot \mathbf{mean_{batch}} + (1 - \alpha) \cdot \mu_i \\ \sigma_i^2 &= \alpha \cdot \mathbf{variance_{batch}} + (1 - \alpha) \cdot \sigma_i^2 \end{aligned}$$

Batch normalization

- Let’s normalize h_i – neuron output before activation:

$$h_i = \gamma_i \boxed{\frac{h_i - \mu_i}{\sqrt{\sigma_i^2}}} + \beta_i \xrightarrow{} \text{0 mean, unit variance}$$

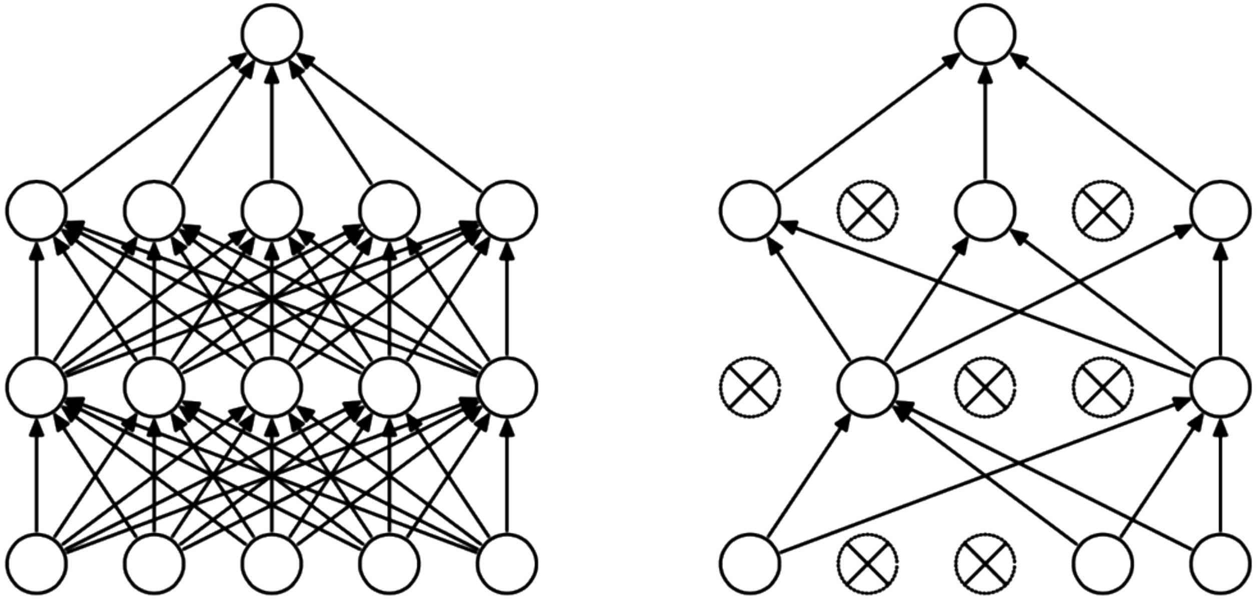
- Where do μ_i and σ_i^2 come from? We can estimate them having a **current training batch**!
- During testing we will use an exponential moving average over train batches:

$$0 < \alpha < 1 \qquad \begin{aligned} \mu_i &= \alpha \cdot \mathbf{mean_{batch}} + (1 - \alpha) \cdot \mu_i \\ \sigma_i^2 &= \alpha \cdot \mathbf{variance_{batch}} + (1 - \alpha) \cdot \sigma_i^2 \end{aligned}$$

- What about γ_i and β_i? Normalization is a differentiable operation and we can apply **backpropagation**!

Dropout

- Regularization technique to reduce overfitting.
- We keep neurons active (non-zero) with probability p.
- This way we sample the network during training and change only a subset of its parameters on every iteration.



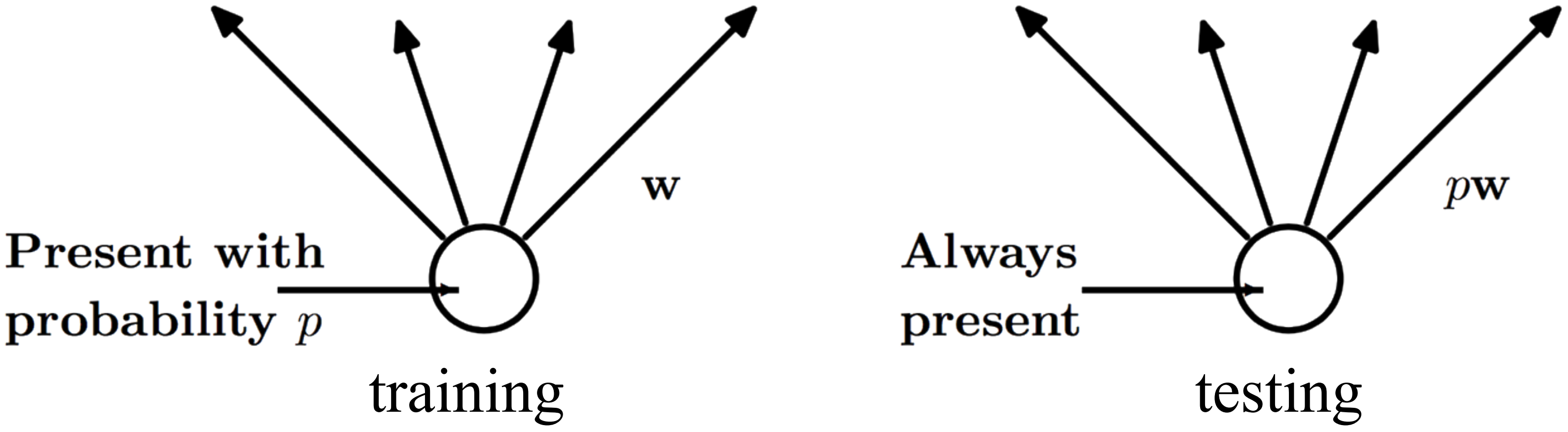
(a) Standard Neural Net (b) After applying dropout.

http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf

Dropout

- During testing all neurons are present but their outputs are multiplied by p to maintain the scale of inputs:

Expected input weight: $p \cdot w + (1 - p) \cdot 0$

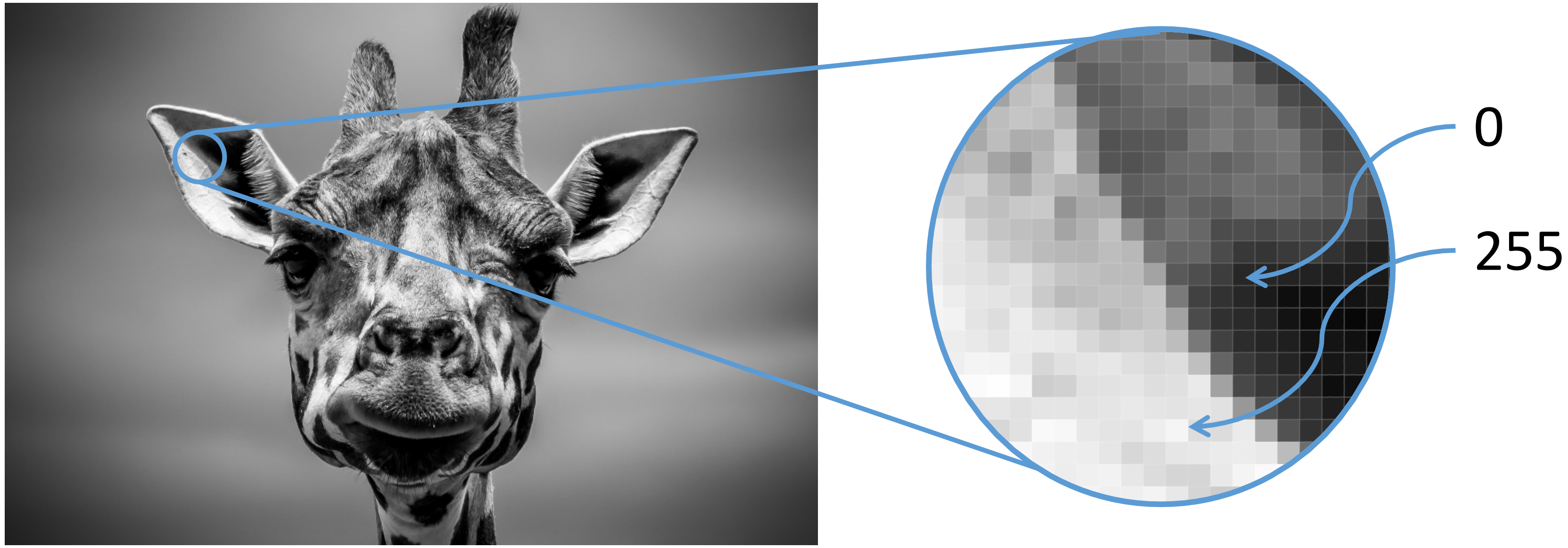


http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf

- The authors of dropout say it’s similar to having an ensemble of exponentially large number of smaller networks.

Digital representation of an image

- Grayscale image is a matrix of pixels (**pic**ture **el**ements)
- Dimensions of this matrix are called image resolution (e.g. 300 x 300)
- Each pixel stores its brightness (or **intensity**) ranging from 0 to 255, 0 intensity corresponds to black color:



- Color images store pixel intensities for 3 channels: red, green and blue

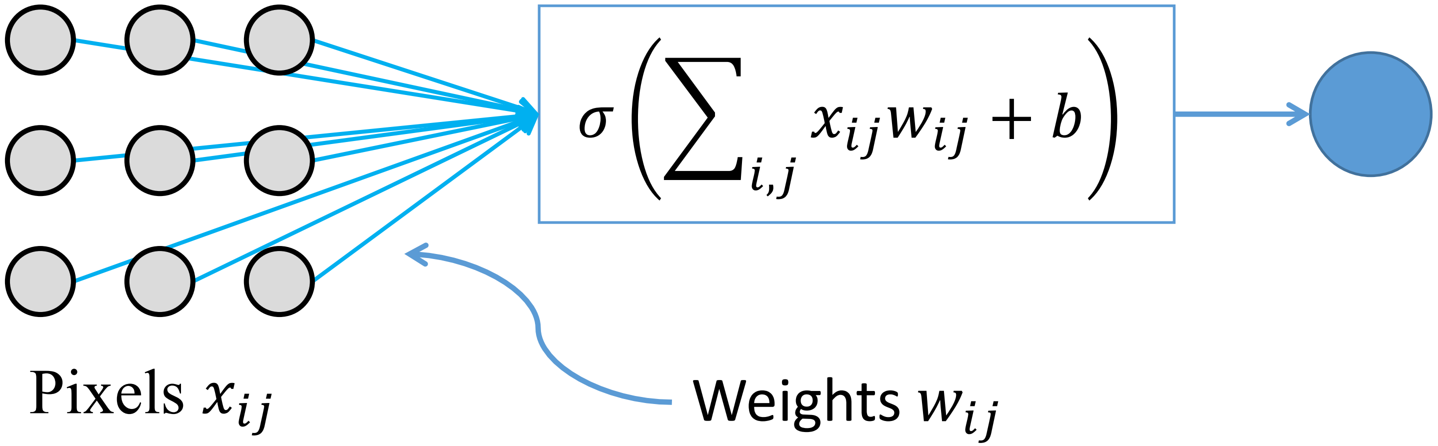
Image as a neural network input

- Normalize input pixels: $x_{norm} = \frac{x}{255} - 0.5$

Image as a neural network input

- Normalize input pixels: $x_{norm} = \frac{x}{255} - 0.5$

- Maybe MLP will work?



$$\sigma\left(\sum_{i,j} x_{ij} w_{ij} + b\right)$$

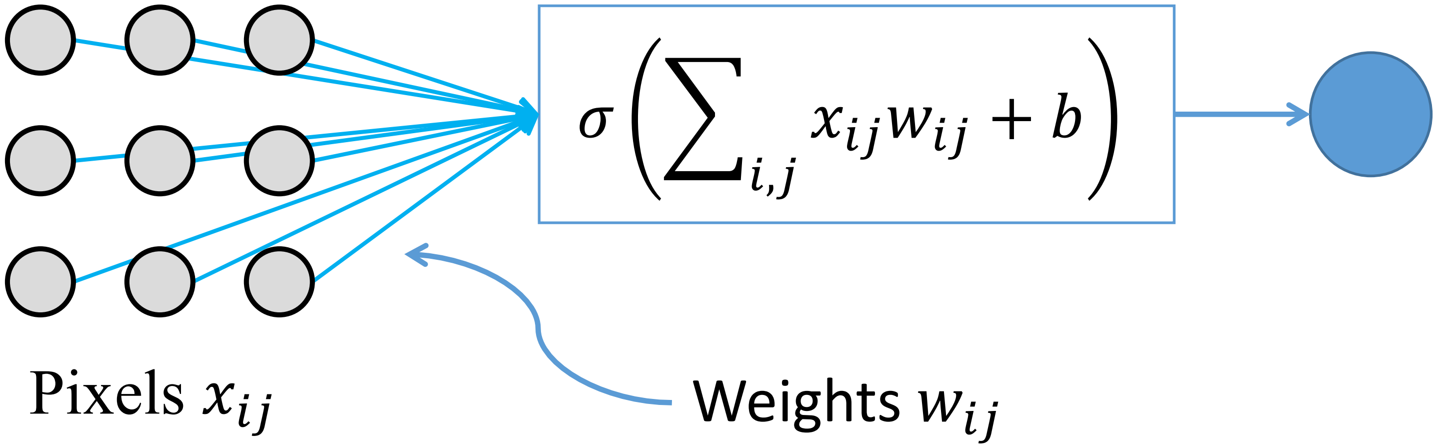
Pixels x_{ij}

Weights w_{ij}

Image as a neural network input

- Normalize input pixels: $x_{norm} = \frac{x}{255} - 0.5$

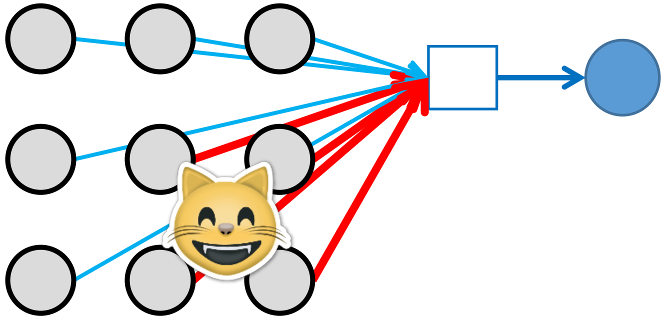
- Maybe MLP will work?



- Actually, no!

Why not MLP?

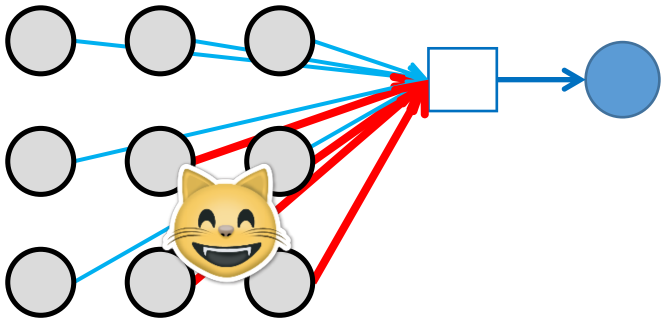
- Let’s say we want to train a “cat detector”



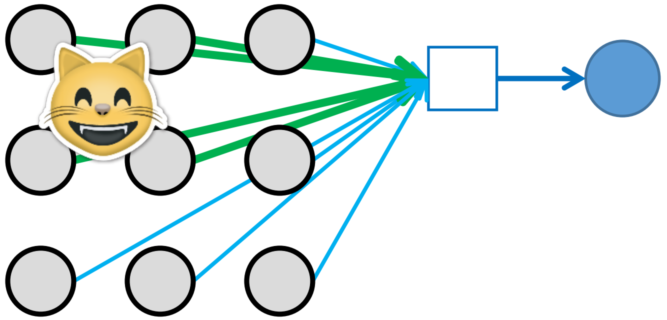
On this training image red weights w_{ij} will change a little bit to better detect a cat

Why not MLP?

- Let’s say we want to train a “cat detector”



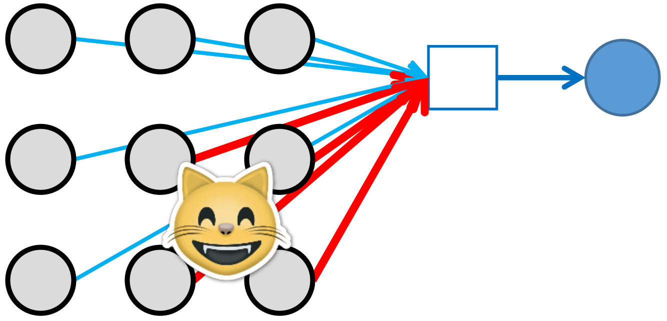
On this training image red weights w_{ij} will change a little bit to better detect a cat



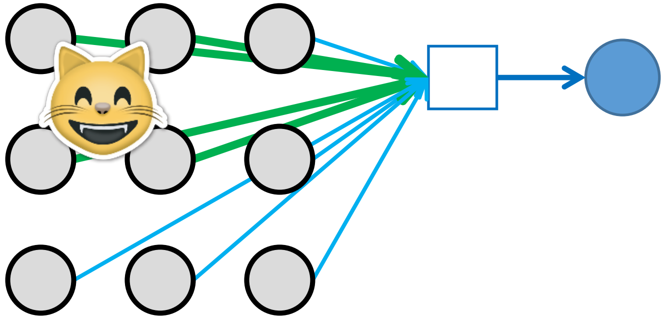
On this training image green weights w_{ij} will change…

Why not MLP?

- Let’s say we want to train a “cat detector”



On this training image red weights w_{ij} will change a little bit to better detect a cat

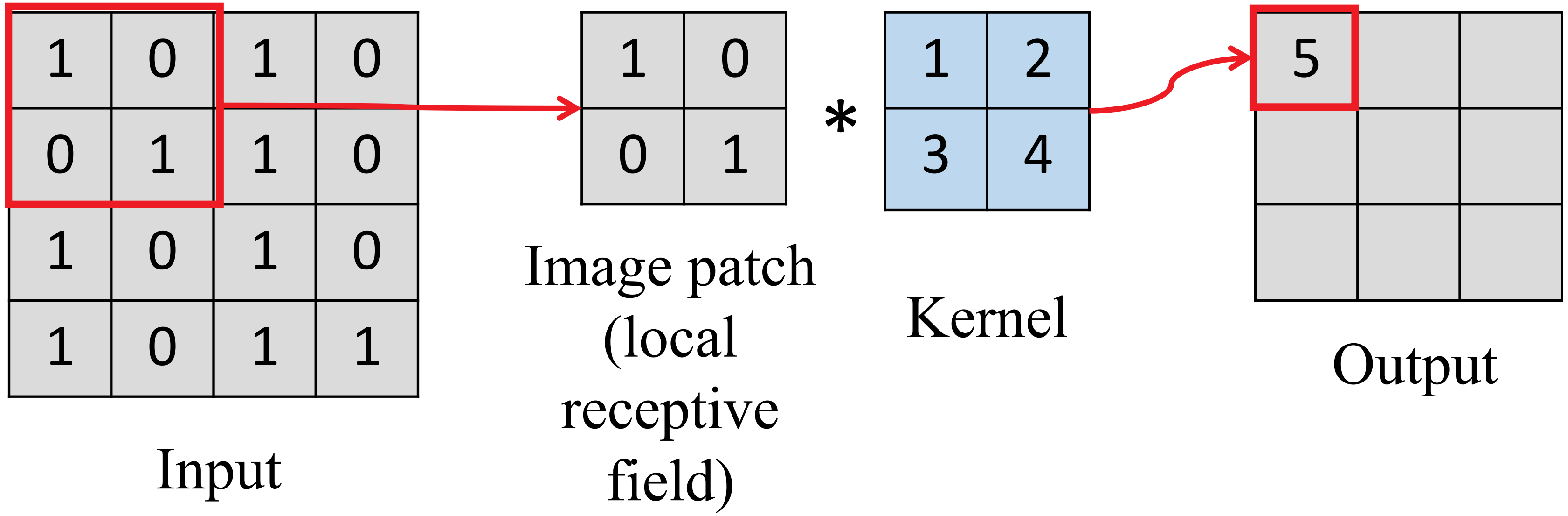


On this training image green weights w_{ij} will change…

- We learn the same “cat features” in different areas and don’t fully utilize the training set!
- What if cats in the test set appear in different places?

Convolutions will help!

Convolution is a dot product of a **kernel** (or filter)
and a patch of an image (**local receptive field**) of the same size



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

Input

1	0
0	1

Image patch (local receptive field)

*

1	2
3	4

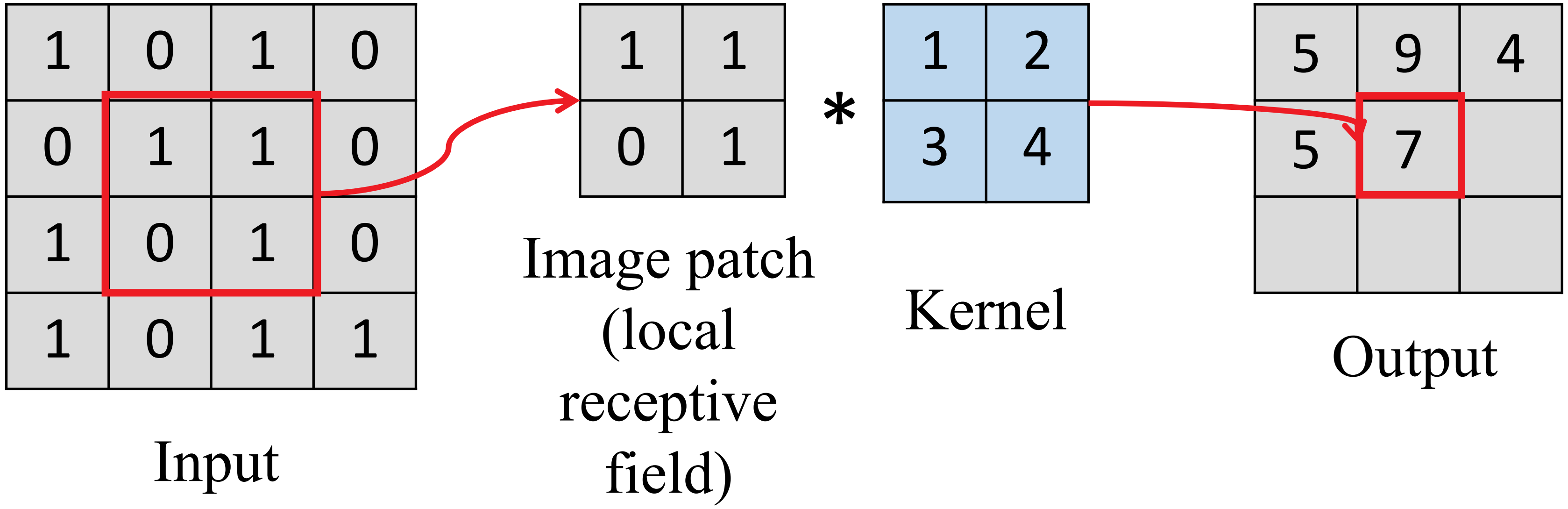
Kernel

5		

Output

Convolutions will help!

Convolution is a dot product of a **kernel** (or filter)
and a patch of an image (**local receptive field**) of the same size



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

Input

1	1
0	1

Image patch (local receptive field)

*

1	2
3	4

Kernel

5	9	4
5	7	

Output

Convolutions have been used for a while

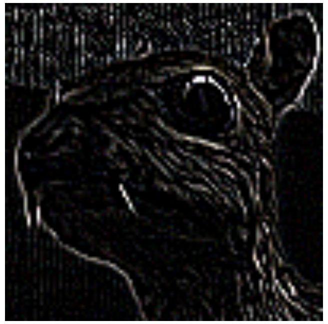
Original image

*

Kernel

-1	-1	-1
-1	8	-1
-1	-1	-1

=



Edge detection

Sums up to 0 (black color) when the patch is a solid fill

Convolutions have been used for a while

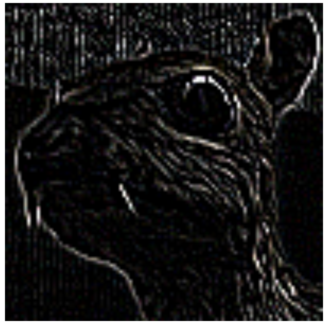
Original image

Kernel

*

-1	-1	-1
-1	8	-1
-1	-1	-1

=

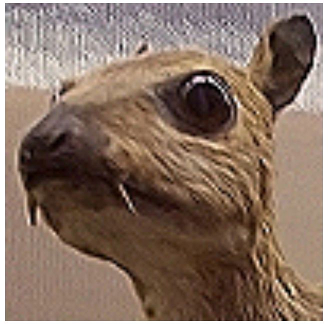


Edge detection

*

0	-1	0
-1	5	-1
0	-1	0

=

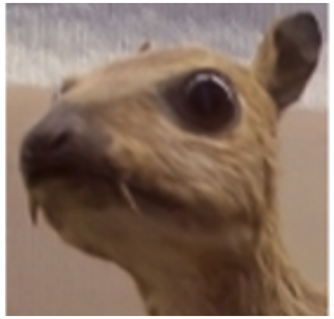


Sharpening

Doesn’t change an image for solid fills

Adds a little intensity on the edges

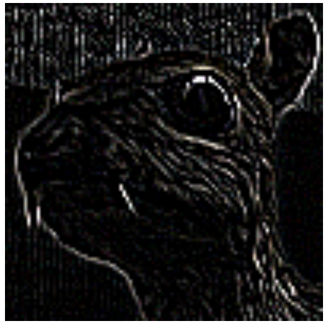
Convolutions have been used for a while



Original image

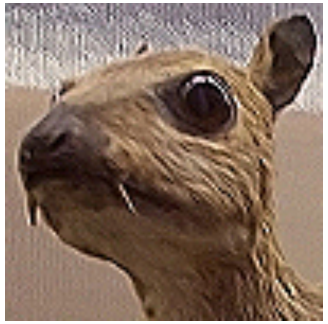
Kernel

$$* \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} =$$



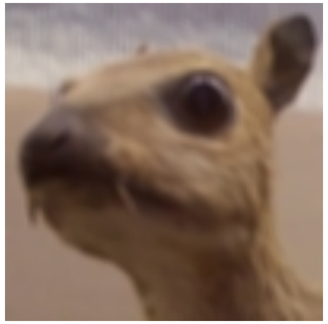
Edge detection

$$* \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} =$$



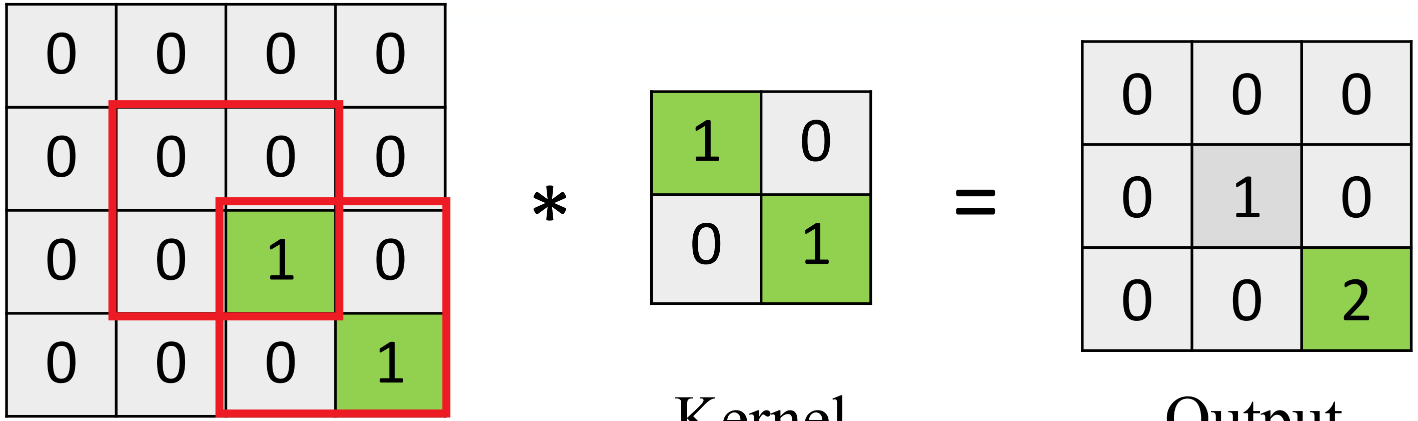
Sharpening

$$* \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} =$$



Blurring

Convolution is similar to correlation



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

Input

*

1	0
0	1

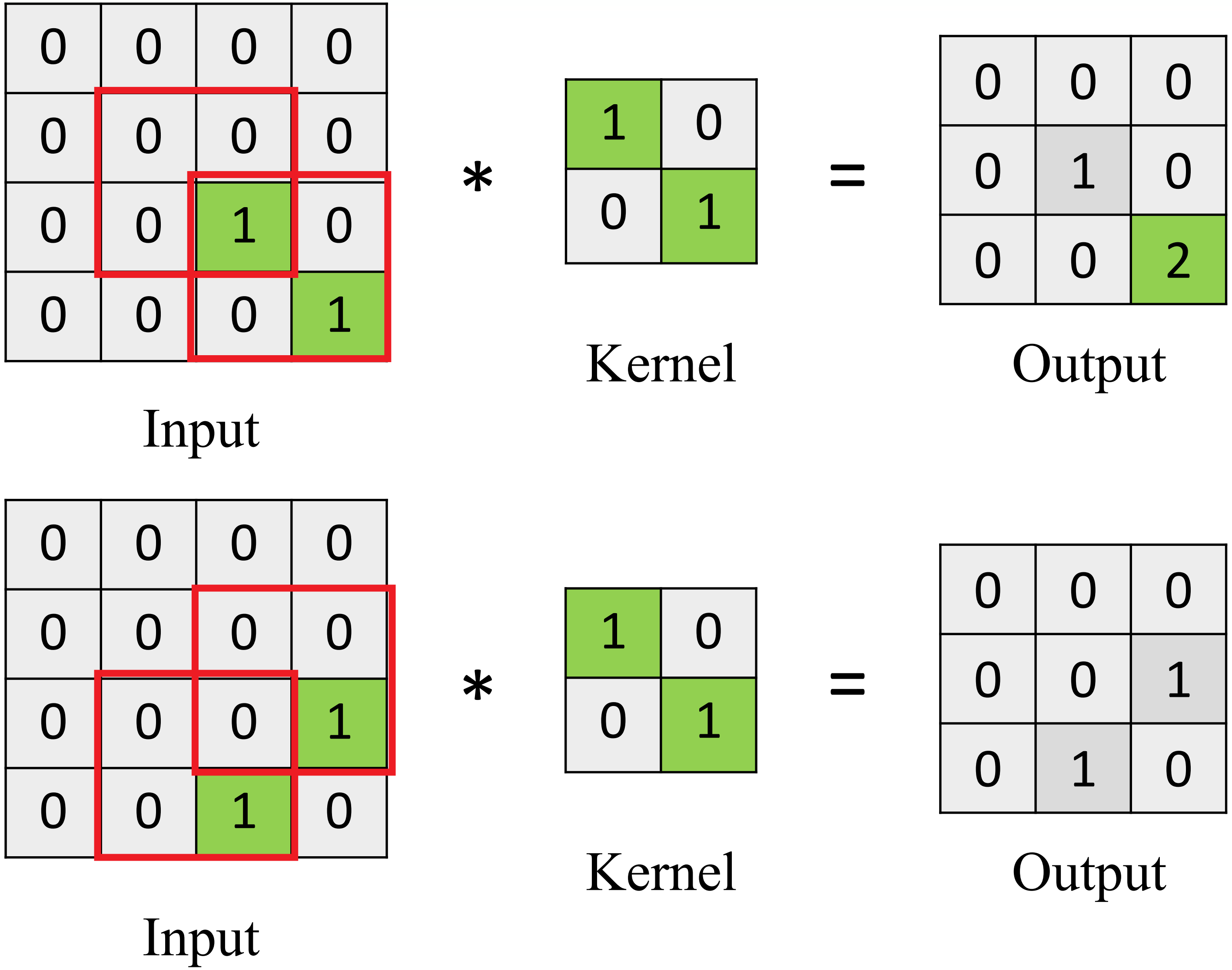
Kernel

=

0	0	0
0	1	0
0	0	2

Output

Convolution is similar to correlation



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

Input

*

1	0
0	1

Kernel

=

0	0	0
0	1	0
0	0	2

Output

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

Input

*

1	0
0	1

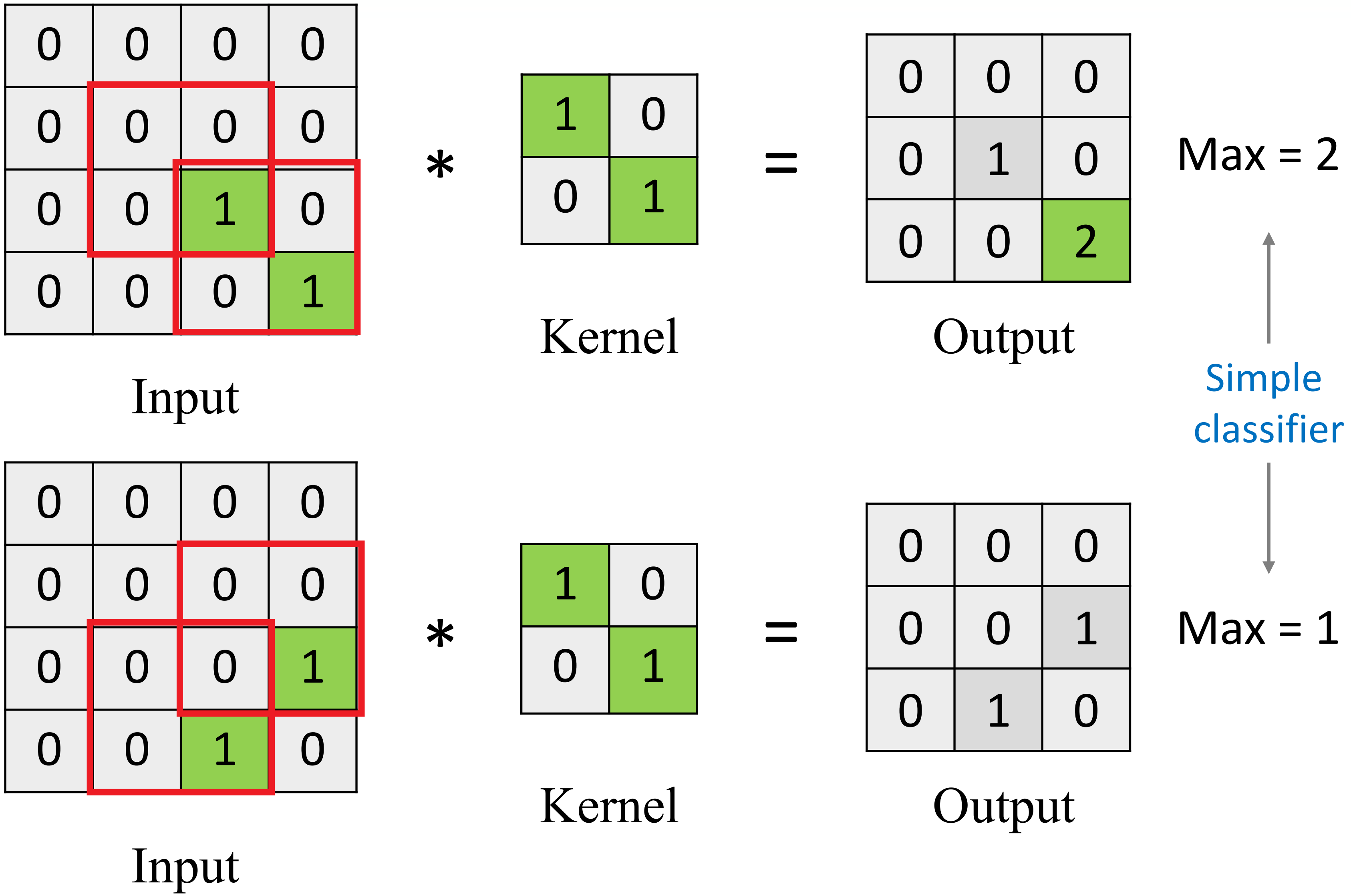
Kernel

=

0	0	0
0	0	1
0	1	0

Output

Convolution is similar to correlation



$$\begin{bmatrix}0&0&0&0\\0&0&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix} * \begin{bmatrix}1&0\\0&1\end{bmatrix} = \begin{bmatrix}0&0&0\\0&1&0\\0&0&2\end{bmatrix}$$

Input

Kernel

Output

Max = 2

Simple classifier

$$\begin{bmatrix}0&0&0&0\\0&0&0&0\\0&0&0&1\\0&0&1&0\end{bmatrix} * \begin{bmatrix}1&0\\0&1\end{bmatrix} = \begin{bmatrix}0&0&0\\0&0&1\\0&1&0\end{bmatrix}$$

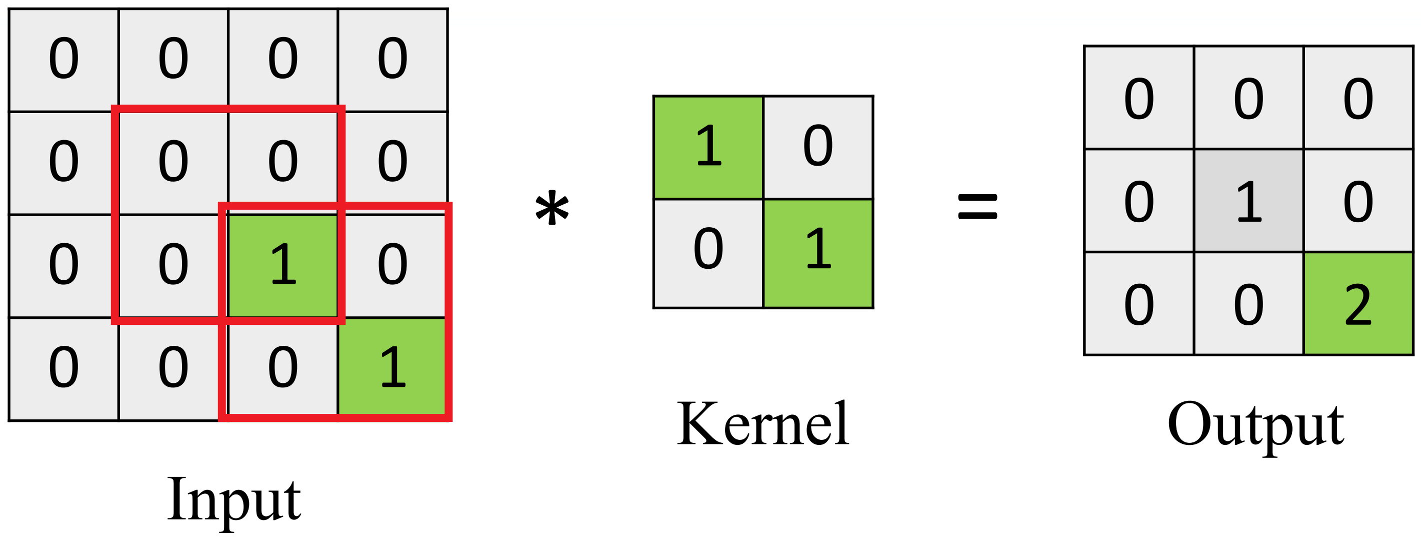
Input

Kernel

Output

Max = 1

Convolution is translation equivariant



Input

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

*

Kernel

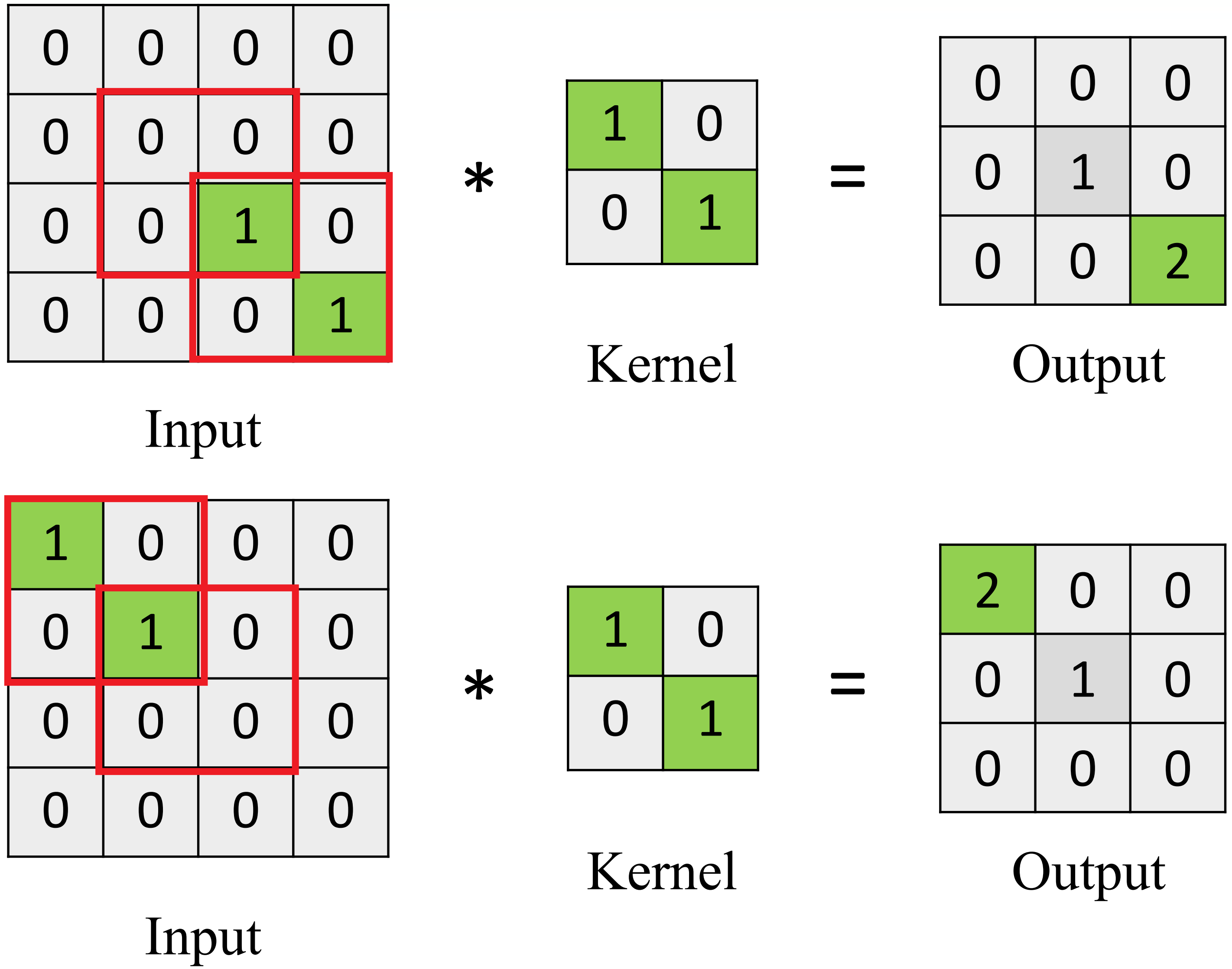
1	0
0	1

=

Output

0	0	0
0	1	0
0	0	2

Convolution is translation equivariant



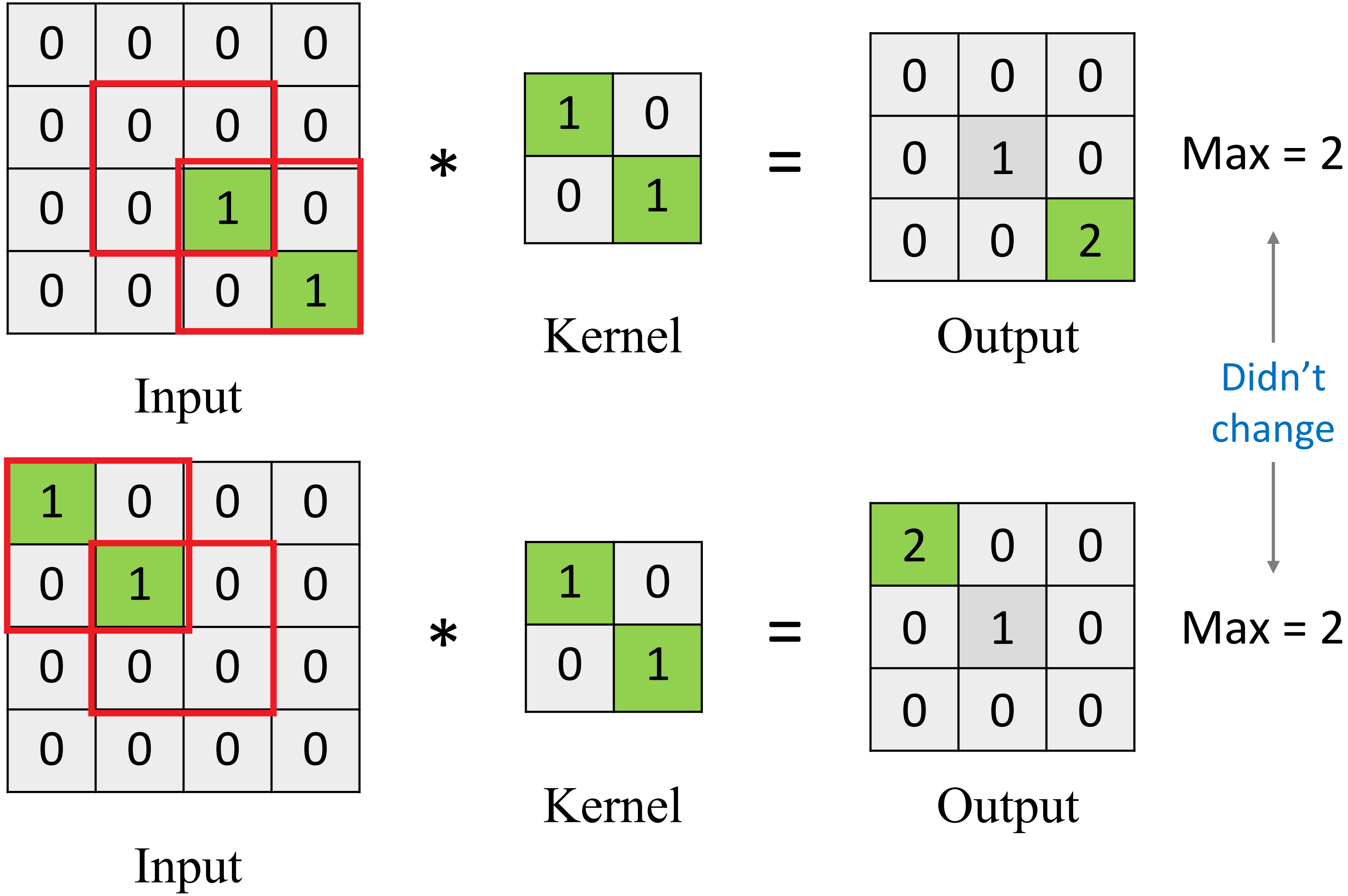
$$\begin{bmatrix}0&0&0&0\\0&0&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix} * \begin{bmatrix}1&0\\0&1\end{bmatrix} = \begin{bmatrix}0&0&0\\0&1&0\\0&0&2\end{bmatrix}$$

Input Kernel Output

$$\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&0\\0&0&0&0\end{bmatrix} * \begin{bmatrix}1&0\\0&1\end{bmatrix} = \begin{bmatrix}2&0&0\\0&1&0\\0&0&0\end{bmatrix}$$

Input Kernel Output

Convolution is translation equivariant



$$\begin{bmatrix}0&0&0&0\\0&0&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix} * \begin{bmatrix}1&0\\0&1\end{bmatrix} = \begin{bmatrix}0&0&0\\0&1&0\\0&0&2\end{bmatrix}$$

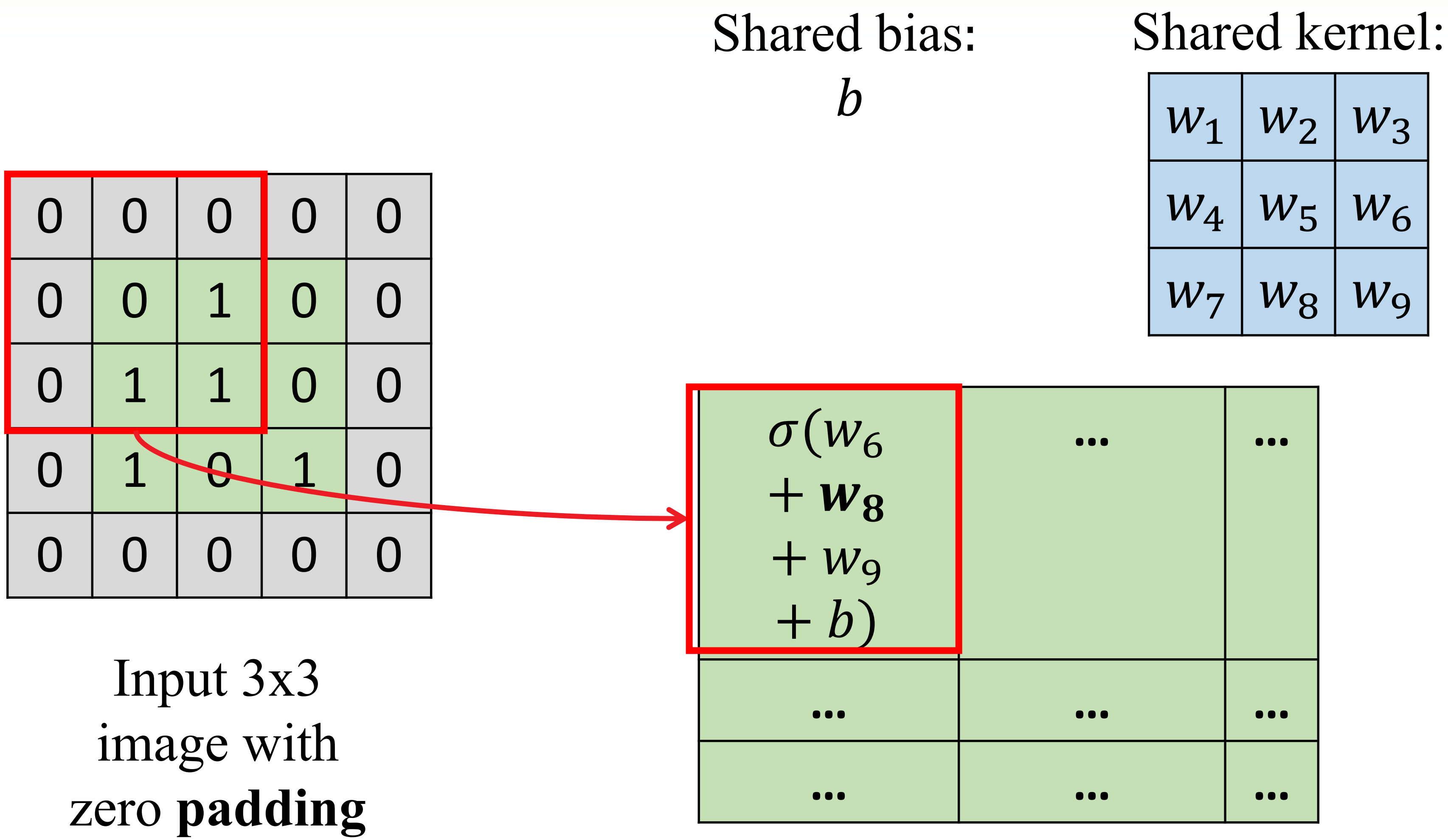
Max = 2

$$\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&0\\0&0&0&0\end{bmatrix} * \begin{bmatrix}1&0\\0&1\end{bmatrix} = \begin{bmatrix}2&0&0\\0&1&0\\0&0&0\end{bmatrix}$$

Max = 2

Didn’t change

Convolutional layer in neural network



Shared bias:
b

Shared kernel:

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

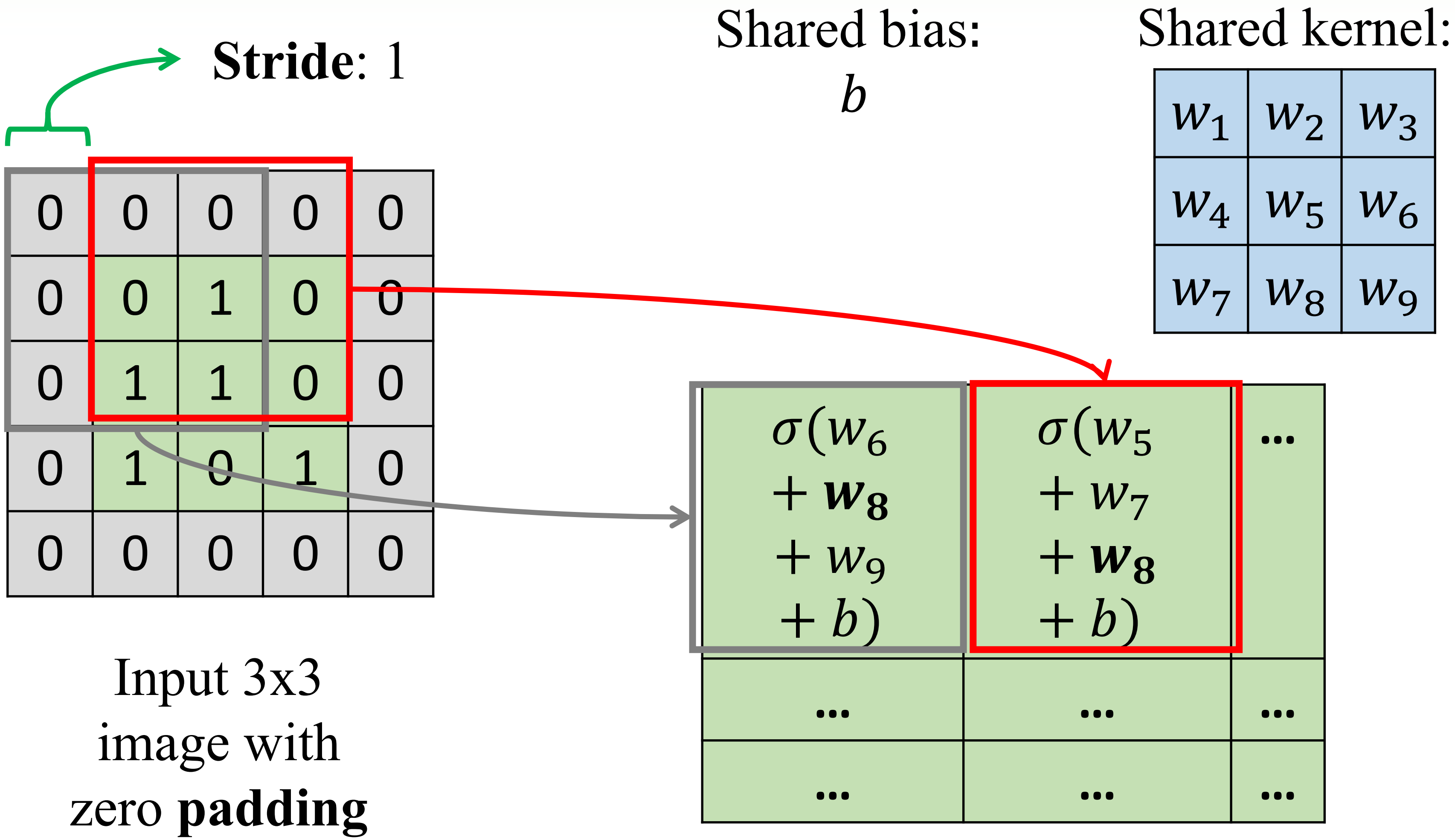
0	0	0	0	0
0	0	1	0	0
0	1	1	0	0
0	1	0	1	0
0	0	0	0	0

$\sigma(w_6 + \boldsymbol{w_8} + w_9 + b)$	…	…
…	…	…
…	…	…

Input 3x3
image with
zero **padding**
(grey area)

9 output neurons (**feature map**) with
only 10 parameters

Convolutional layer in neural network



Stride: 1

Shared bias:
b

Shared kernel:

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

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

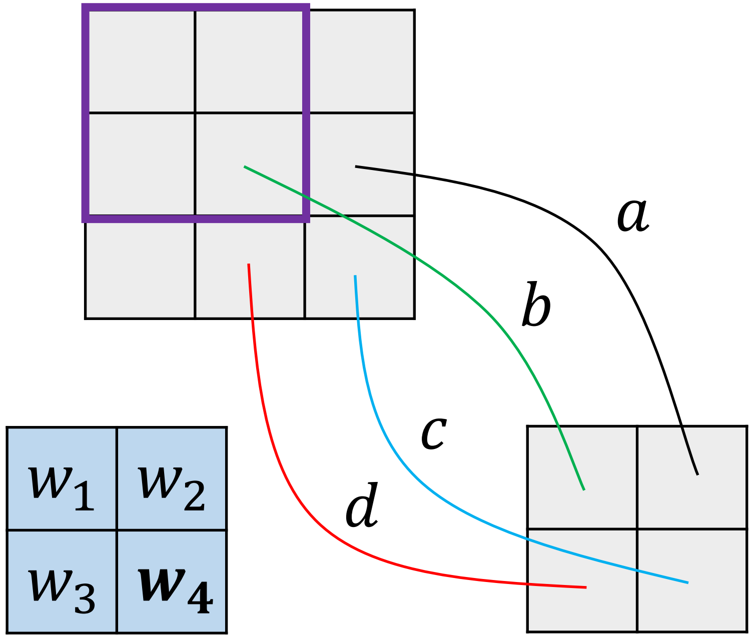
$\sigma(w_6 + \boldsymbol{w_8} + w_9 + b)$	$\sigma(w_5 + w_7 + \boldsymbol{w_8} + b)$	…
…	…	…
…	…	…

Input 3x3
image with
zero **padding**
(grey area)

9 output neurons (**feature map**) with
only 10 parameters

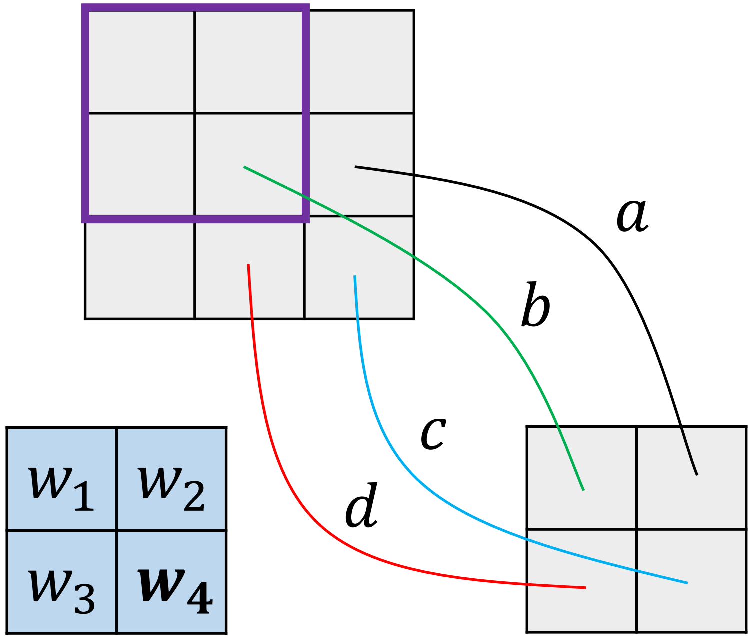
Backpropagation for CNN

Gradients are first calculated as if the kernel weights were not shared:



Backpropagation for CNN

Gradients are first calculated as if the kernel weights were not shared:

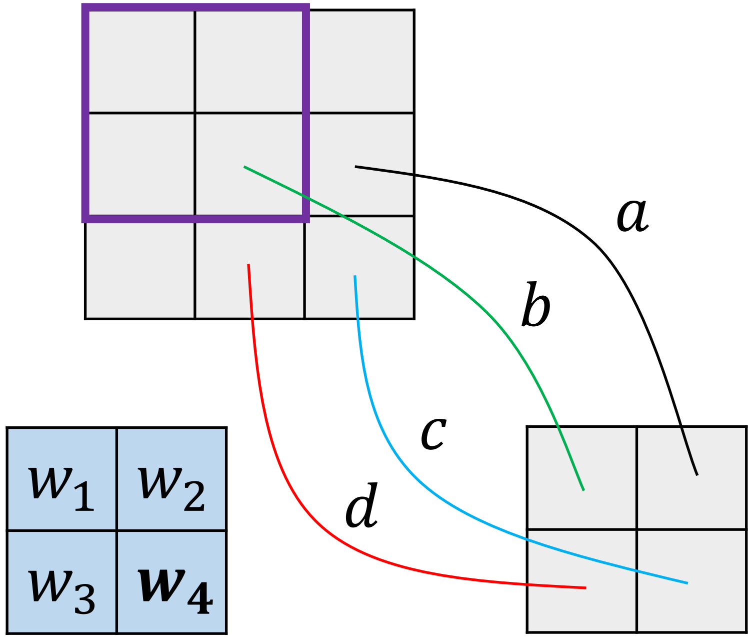


$$a = a - \gamma \frac{\partial L}{\partial a} \qquad b = b - \gamma \frac{\partial L}{\partial b}$$

$$c = c - \gamma \frac{\partial L}{\partial c} \qquad d = d - \gamma \frac{\partial L}{\partial d}$$

Backpropagation for CNN

Gradients are first calculated as if the kernel weights were not shared:



$$a = a - \gamma \frac{\partial L}{\partial a} \qquad b = b - \gamma \frac{\partial L}{\partial b}$$

$$c = c - \gamma \frac{\partial L}{\partial c} \qquad d = d - \gamma \frac{\partial L}{\partial d}$$

$$w_4 = w_4 - \gamma \left(\frac{\partial L}{\partial a} + \frac{\partial L}{\partial b} + \frac{\partial L}{\partial c} + \frac{\partial L}{\partial d} \right)$$

Gradients of the same shared weight are summed up!

Convolutional vs fully connected layer

- In convolutional layer the same kernel is used for every output neuron, this way we share parameters of the network and train a better model

Convolutional vs fully connected layer

- In convolutional layer the same kernel is used for every output neuron, this way we share parameters of the network and train a better model

- 300x300 input, 300x300 output, 5x5 kernel – **26** parameters in convolutional layer and $\mathbf{8.1 \times 10^{9}}$ parameters in fully connected layer (each output is a perceptron)

Convolutional vs fully connected layer

- In convolutional layer the same kernel is used for every output neuron, this way we share parameters of the network and train a better model

- 300x300 input, 300x300 output, 5x5 kernel – **26** parameters in convolutional layer and $\mathbf{8.1 \times 10^9}$ parameters in fully connected layer (each output is a perceptron)

- Convolutional layer can be viewed as a special case of a fully connected layer when all the weights outside
  the **local receptive field** of each neuron equal 0
  and kernel parameters are shared between neurons

Ссылки

- http://cs231n.stanford.edu/
- http://cs231n.github.io/convolutional-networks/
- https://brohrer.github.io/how_convolutional_neural_networks_work.html
- https://blog.keras.io/how-convolutional-neural-networks-see-the-world.html