VIBOT 2018

Deep Learning with Convolutional Neural Networks

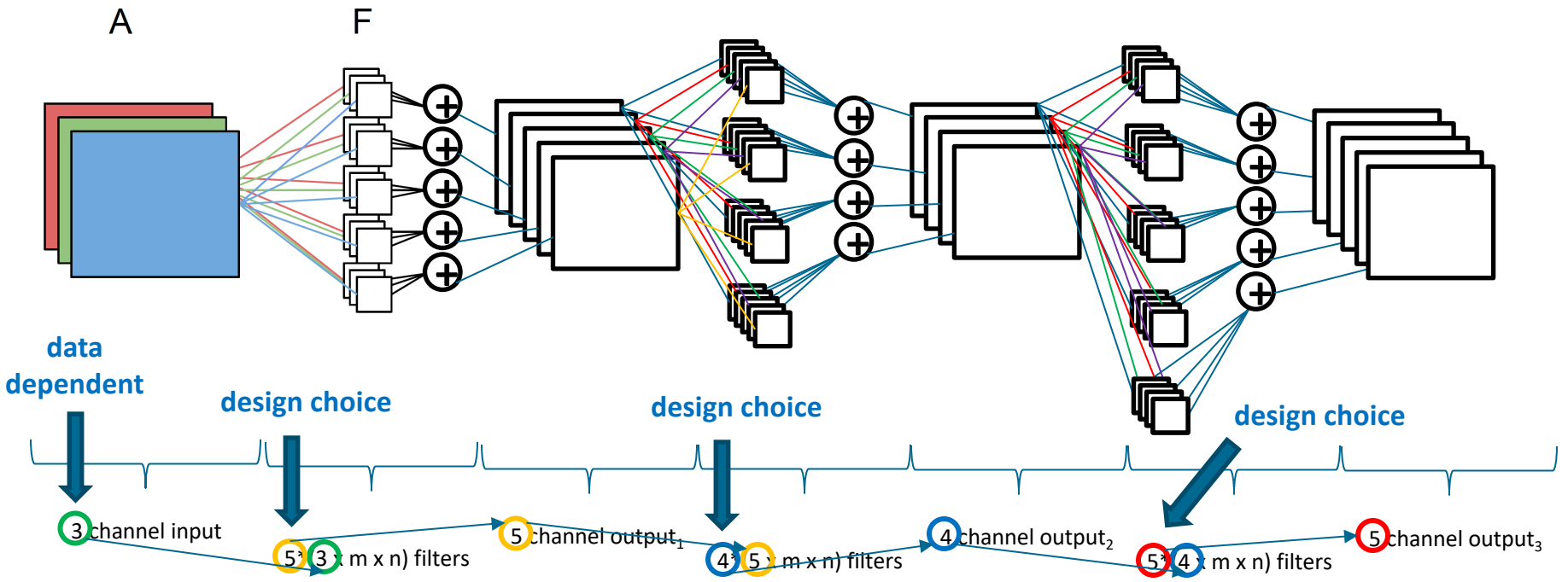
Francesco Ciompi

francesco.ciompi@radboudumc.nl

Radboudumc

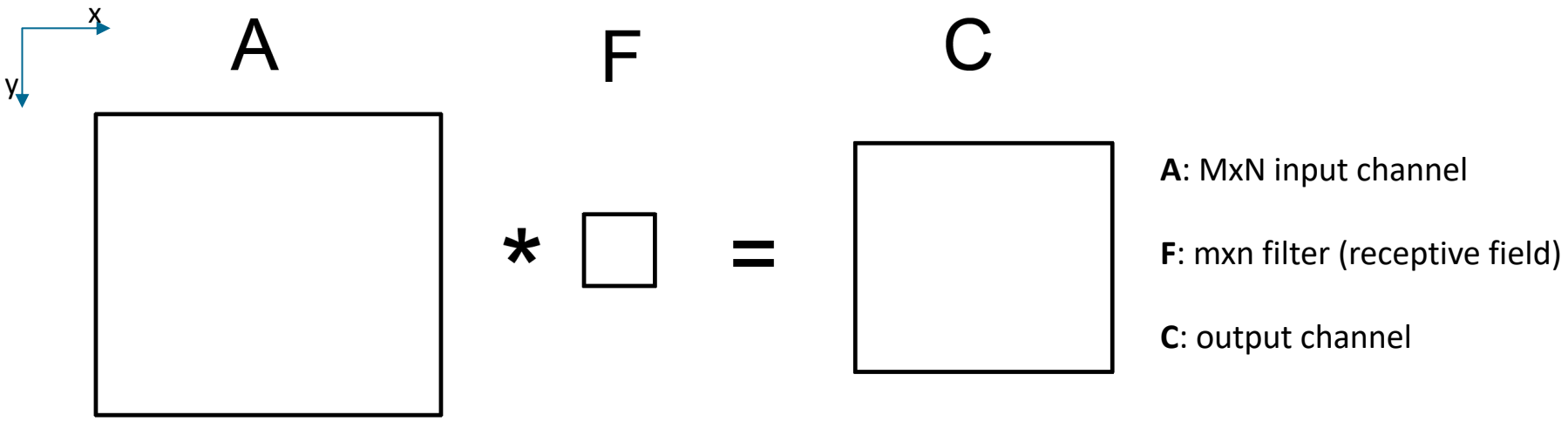
Convolutional Neural Network

- A convolutional network is (basically) a stack of convolutional layers



Convolutional Neural Networks

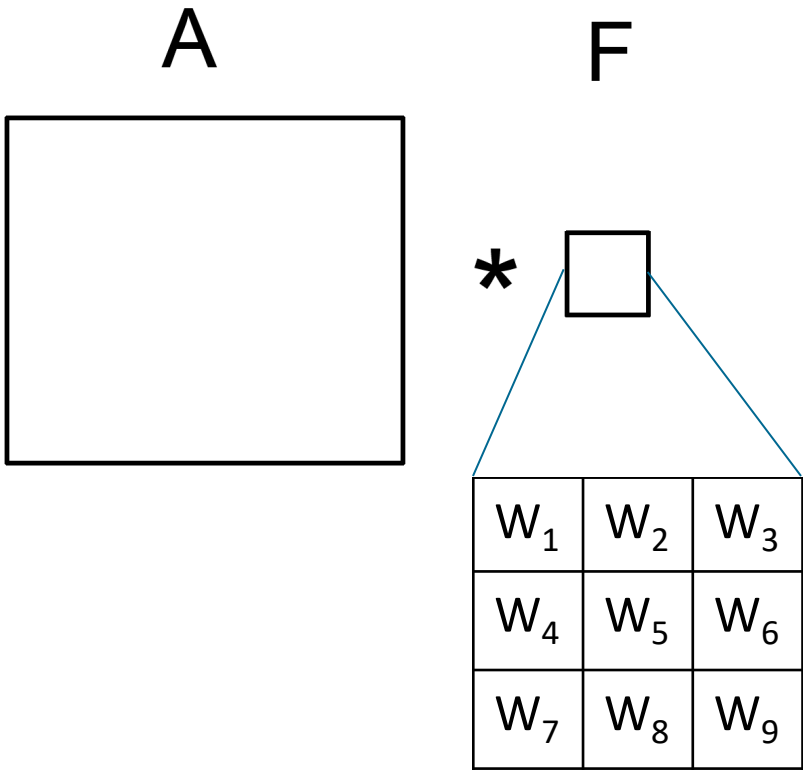
Convolutional Neural Network

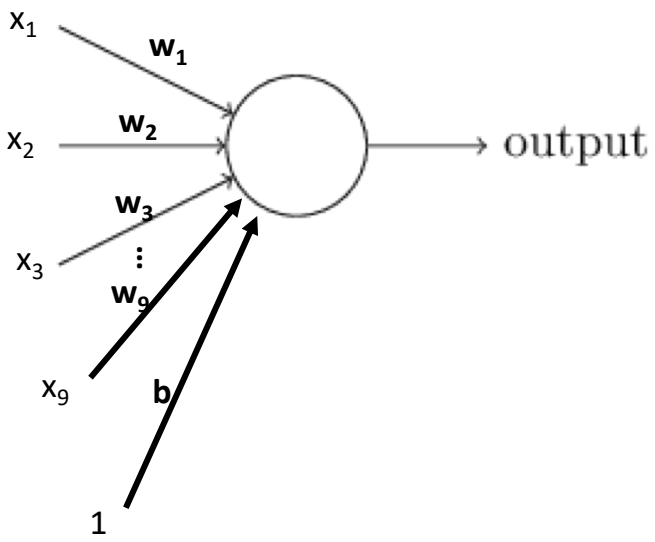


Convolution

$$C(i,j) = \sum_{x=-m/2}^{m/2} \sum_{y=-n/2}^{n/2} A(i-x, j-y)F(x,y)$$

Convolutional Neural Network

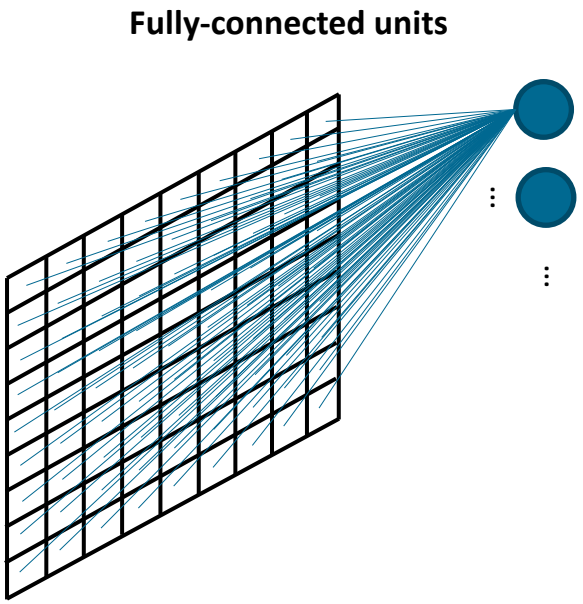




Training a convolutional network means learning these parameters!

Convolutional Neural Network

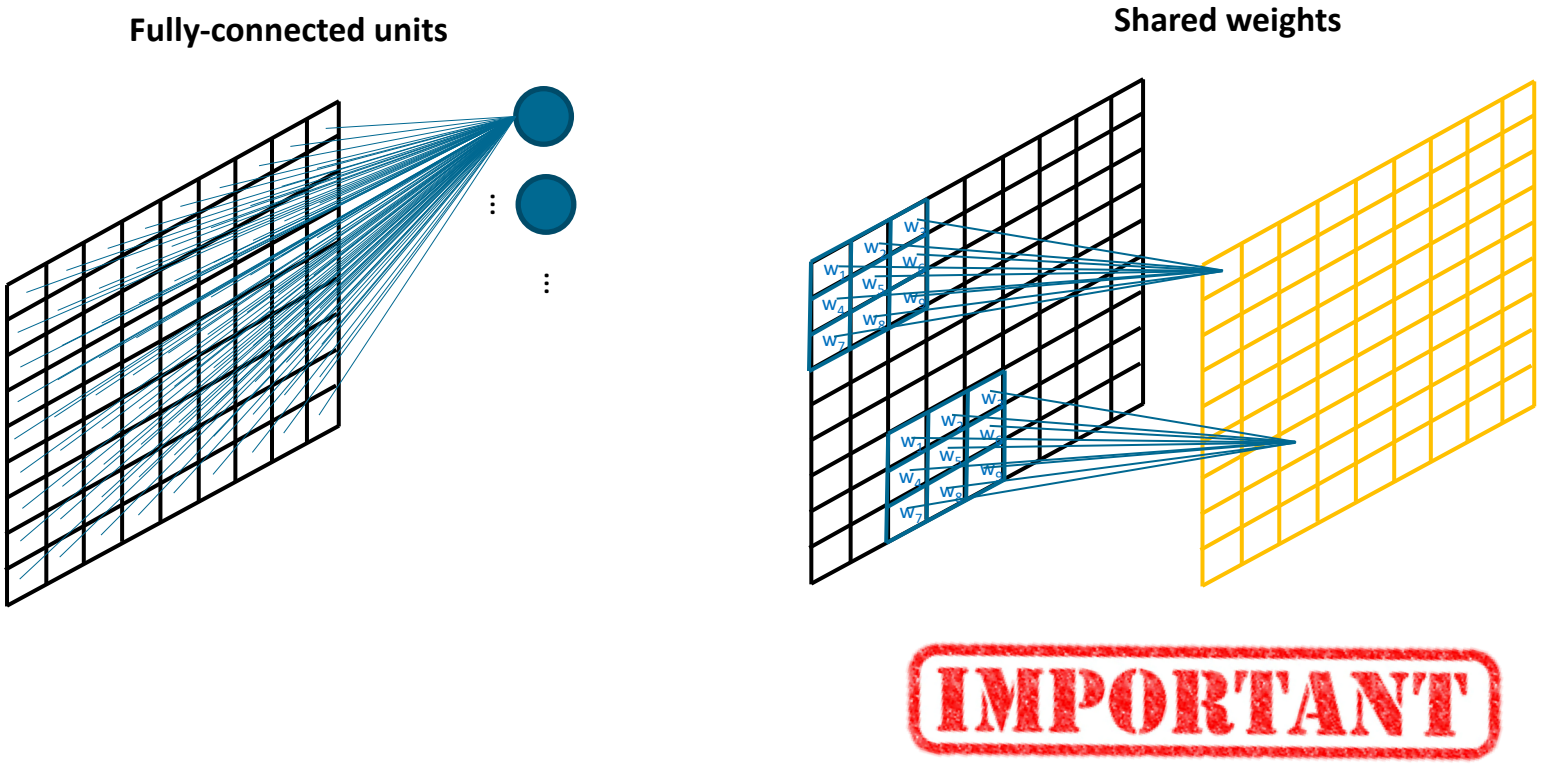
- Why not a multi layer perceptron (fully-connected layer)?



- **Huge first layer**
  - Lots of weights
  - Increase the capacity
  - Lots of training data required
  - Huge memory requirement
- **No robustness to distortions or shift of the input**
- **In MLP, input variables can be presented in any (fixed) order**
  - This does not take into account for topology in data
  - Images have a strong 2D structure

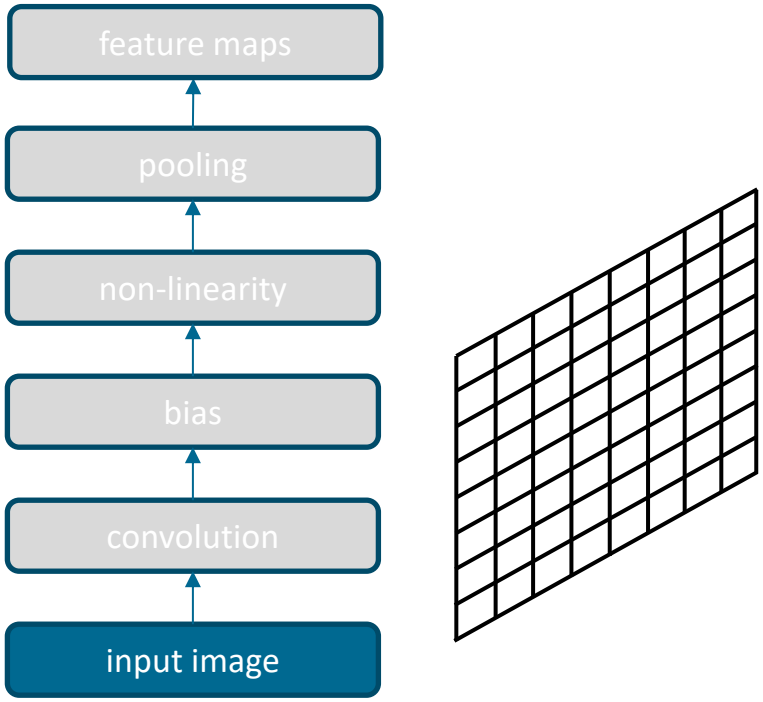
Convolutional Neural Network

- Why not a multi layer perceptron?

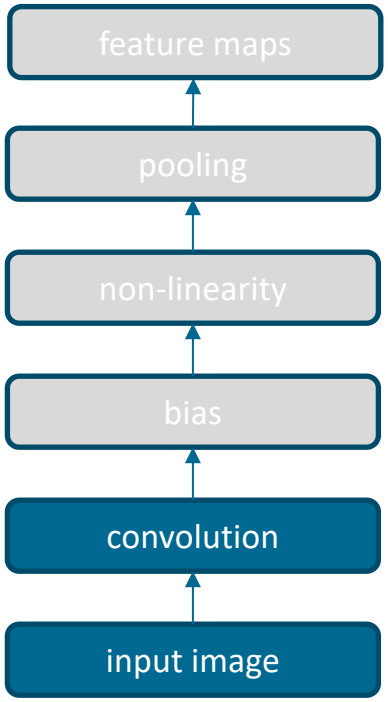


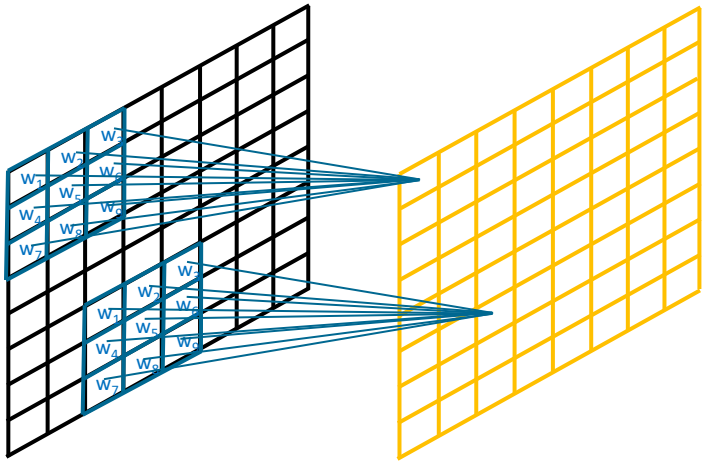
Convolutional Neural Networks

Convolutional Neural Network

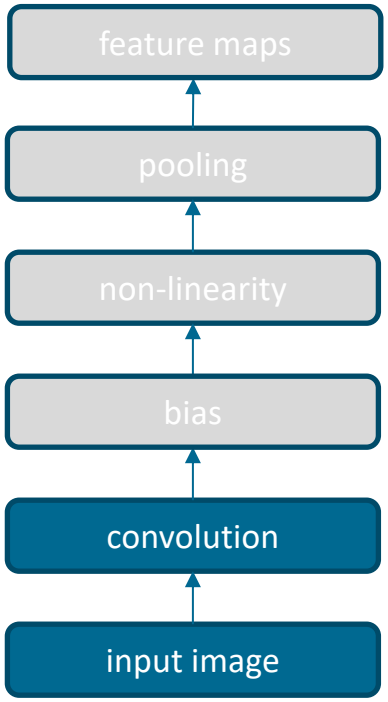


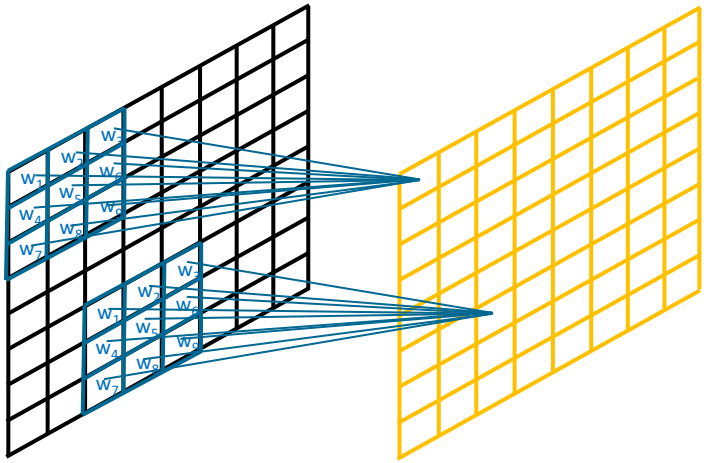
Convolutional Neural Network

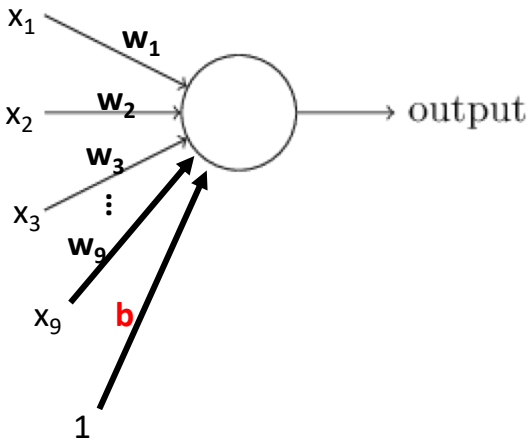




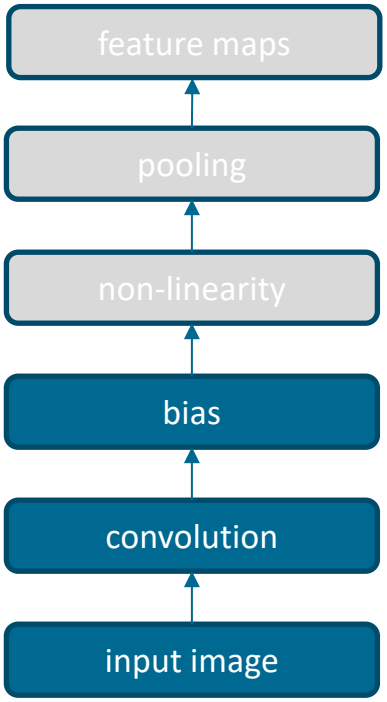
Convolutional Neural Networks

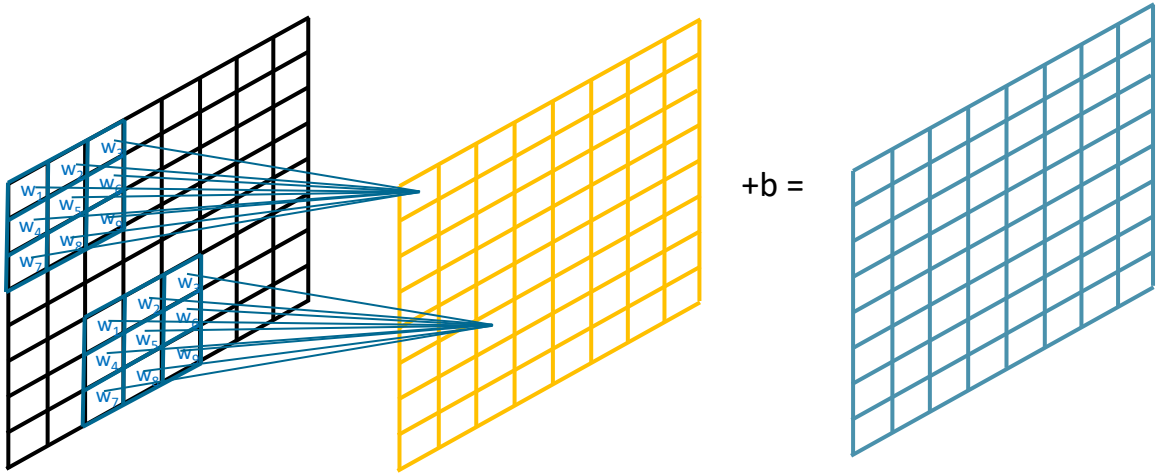






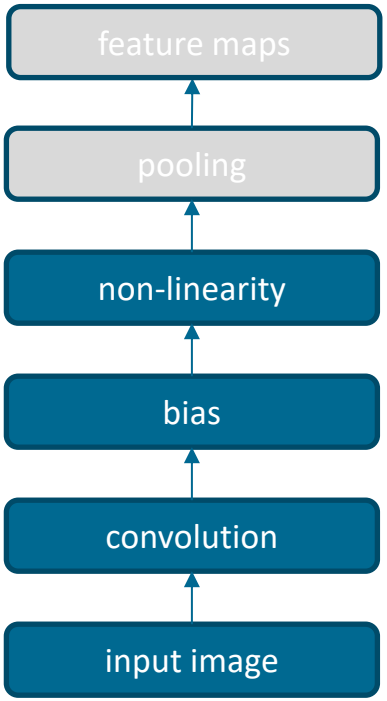
Convolutional Neural Networks

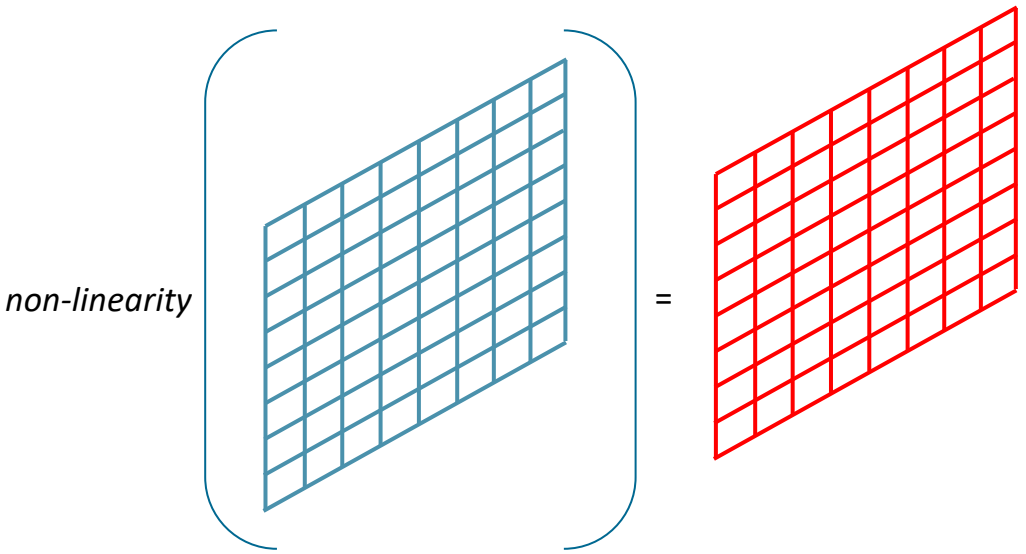




We add the trained value b to each pixel.

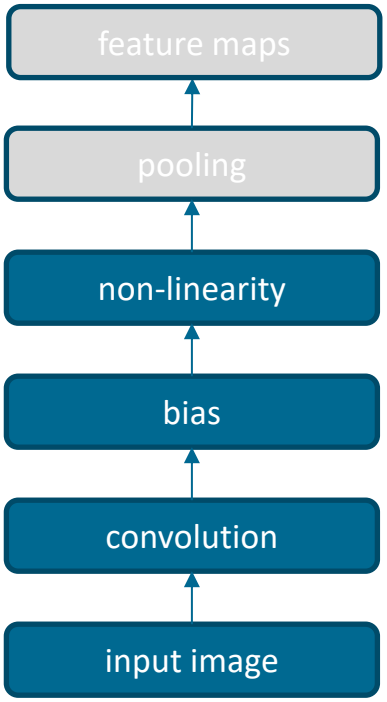
Convolutional Neural Networks





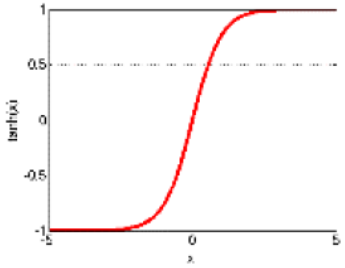
We apply a non-linear function to each pixel. The result is also called **activation**

Convolutional Neural Network

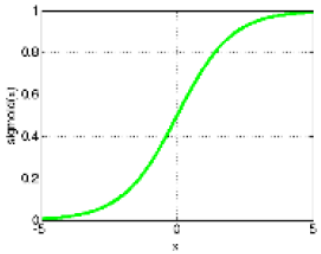


Non-linearity function: continue and differentiable (almost) everywhere

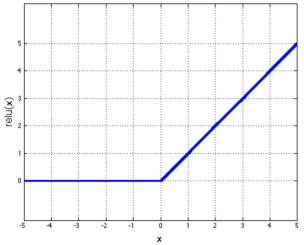
Tanh



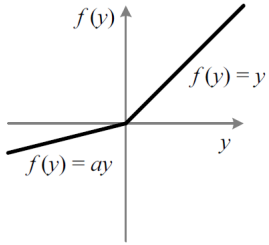
Sigmoid



Rectified linear (ReLU)



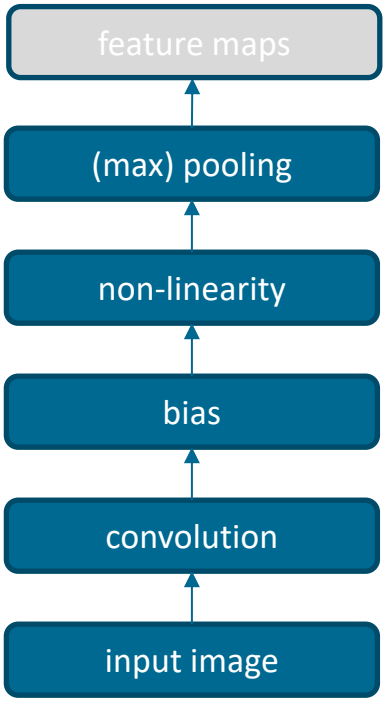
Parametric Rectified linear (PReLU)
[ArXiv:1502.01852]



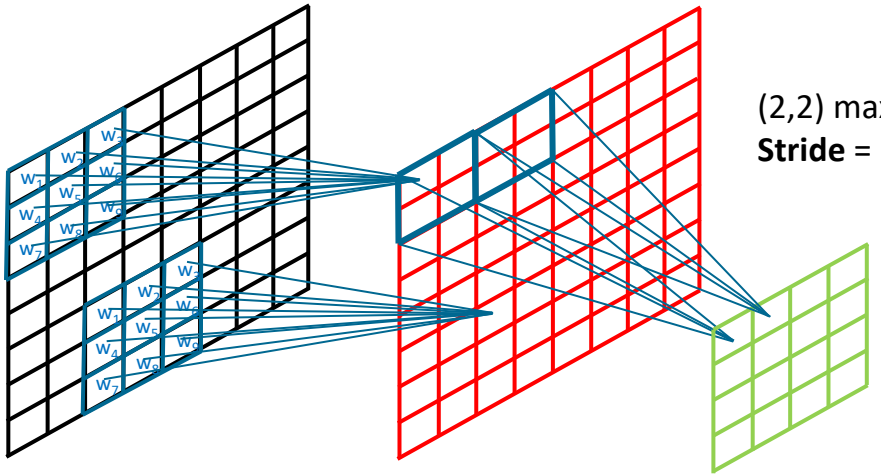
ReLU

- makes learning faster
- sparse representation
- avoid saturation
- avoid vanishing gradient

Convolutional Neural Network

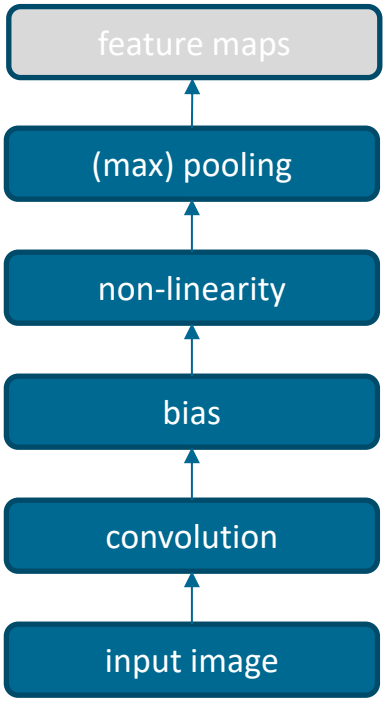


- **Pooling**
  - Encodes a degree of invariance with respect to translations
  - Reduces the size of the layers

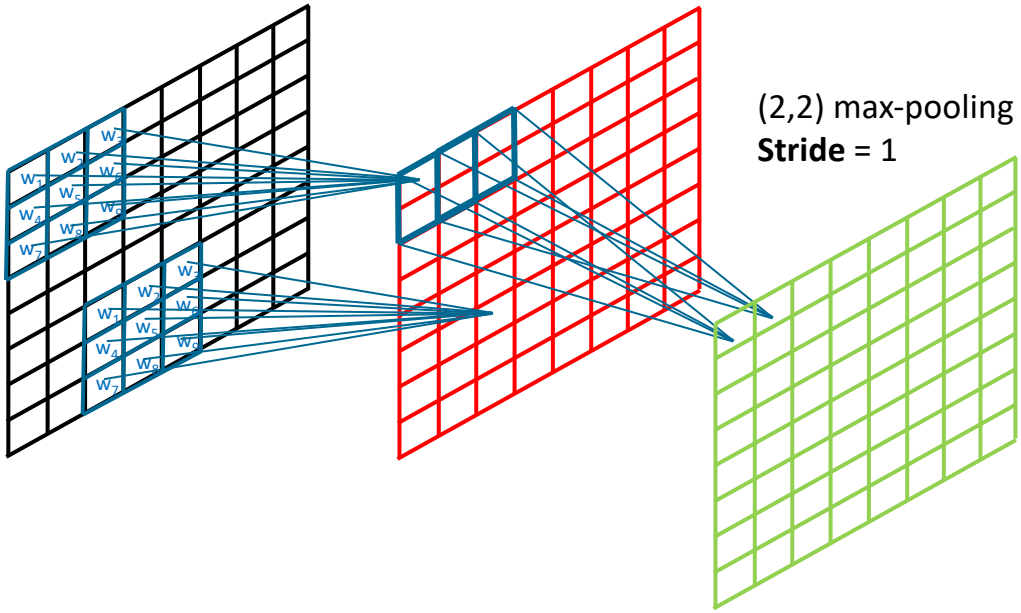


(2,2) max-pooling
Stride = 2

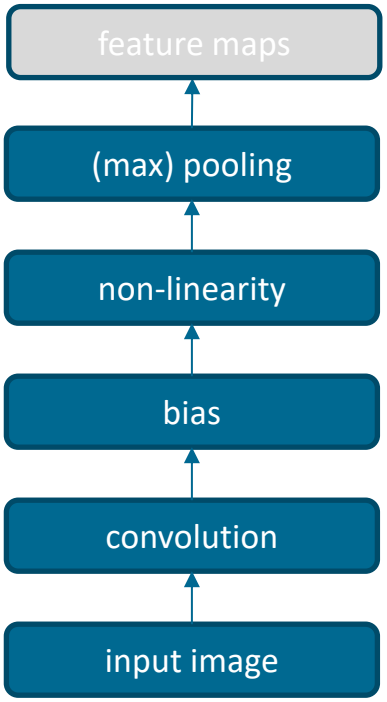
Convolutional Neural Network



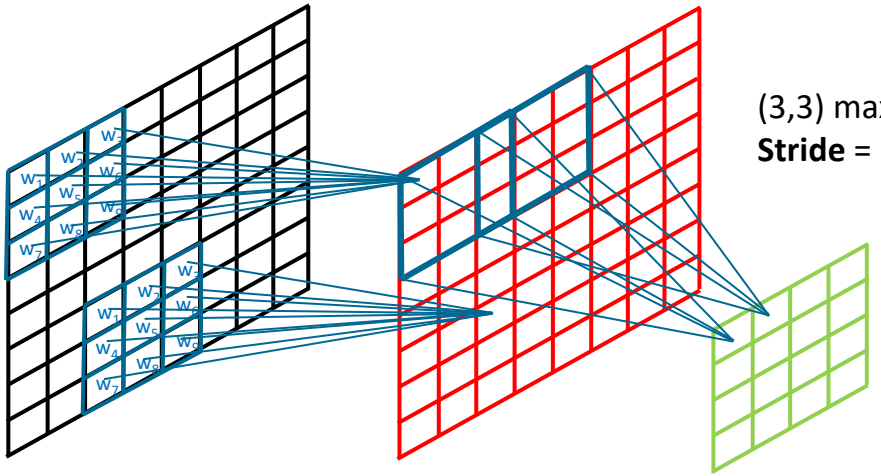
- **Pooling**
  - Encodes a degree of invariance with respect to translations
  - Reduces the size of the layers



Convolutional Neural Network

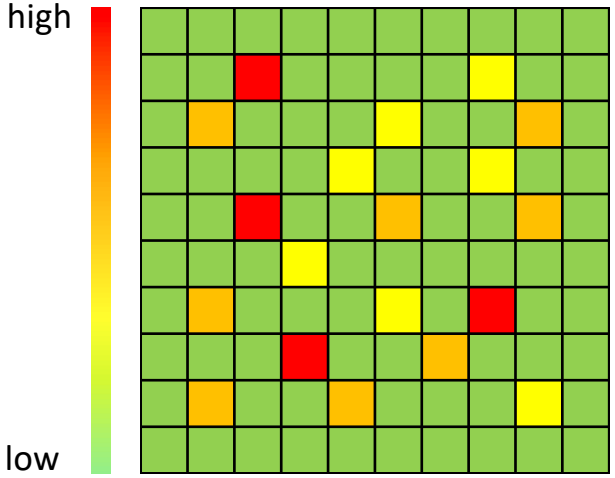


- **Pooling**
  - Encodes a degree of invariance with respect to translations
  - Reduces the size of the layers

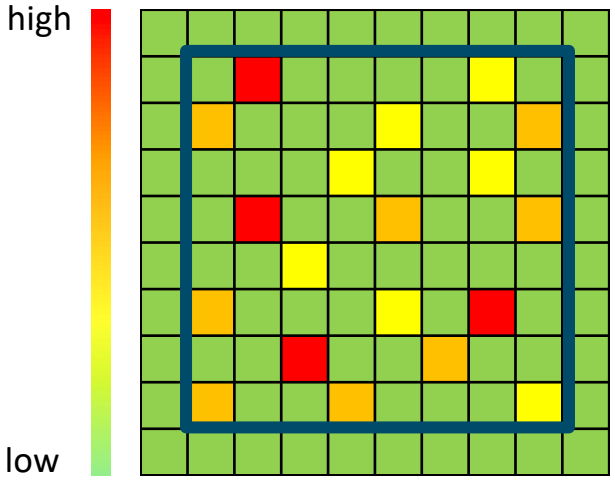


(3,3) max-pooling
Stride = 2

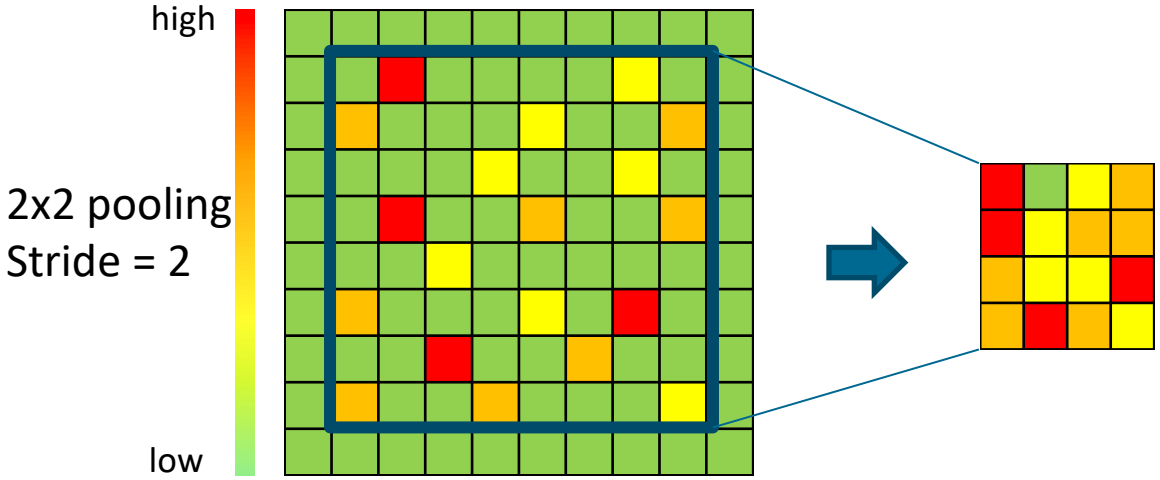
Why max pooling?



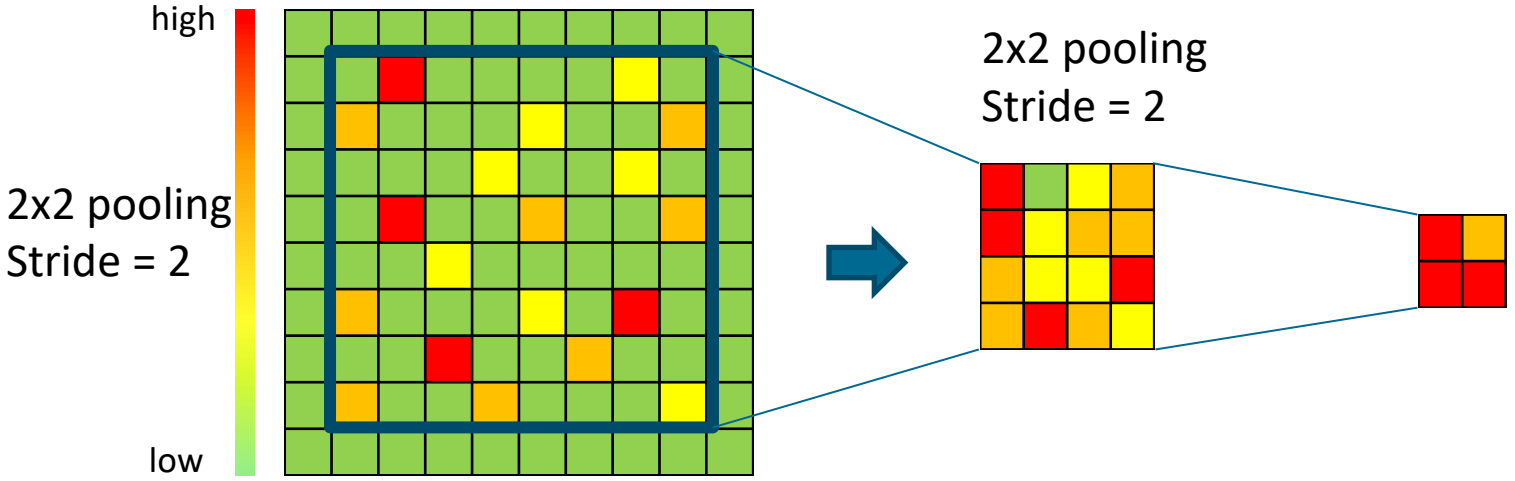
Why max pooling?



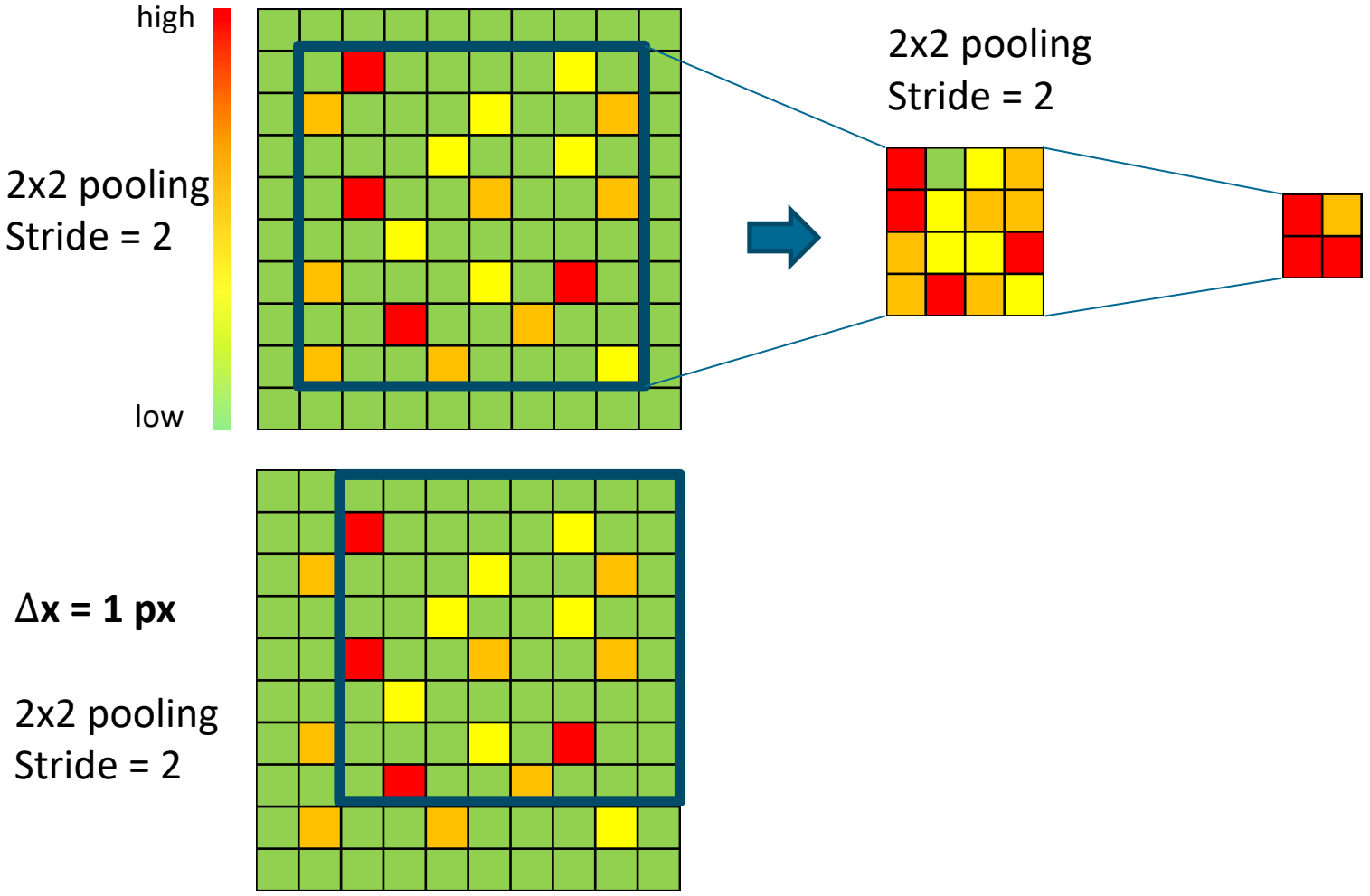
Why max pooling?



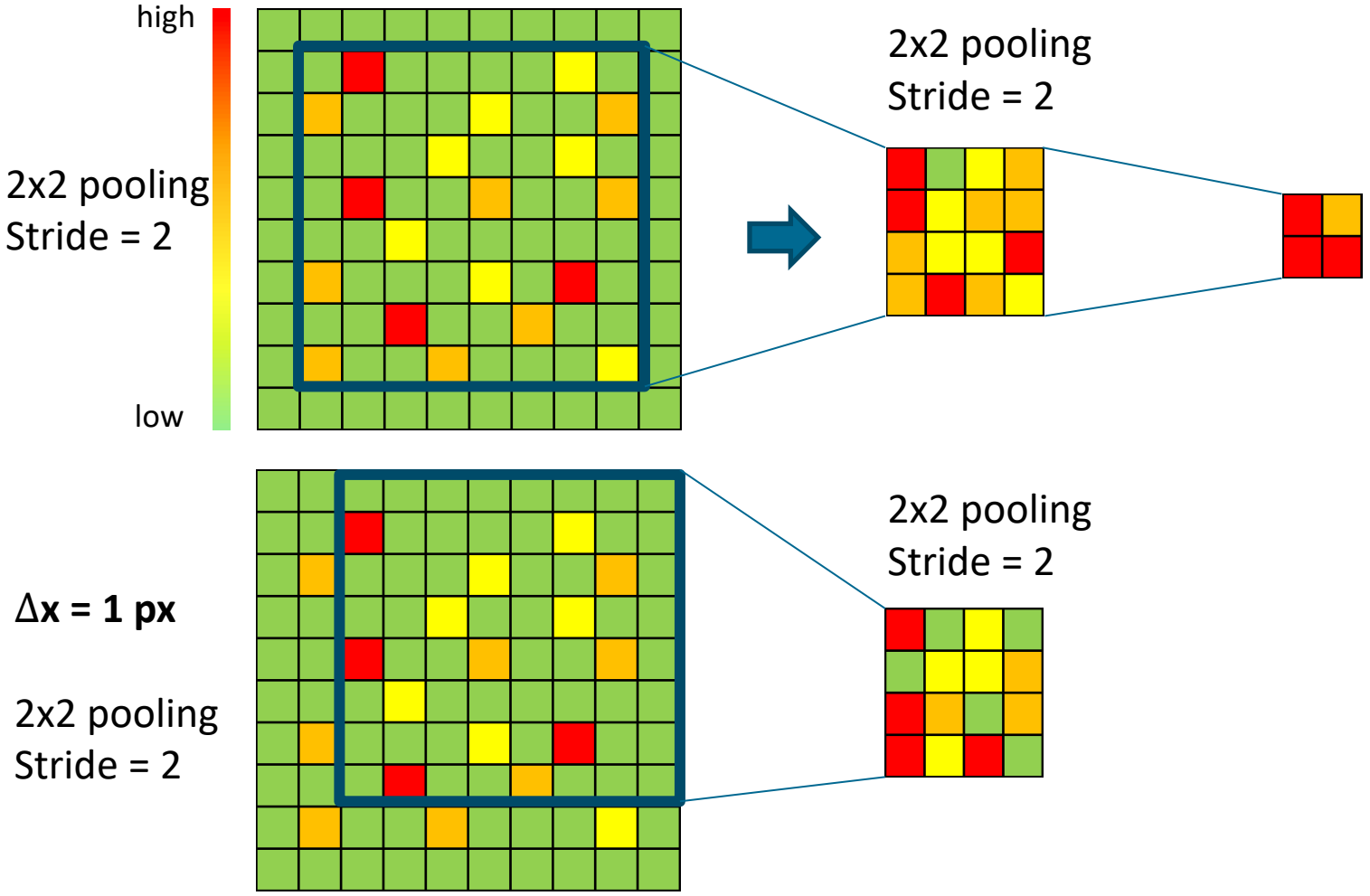
Why max pooling?



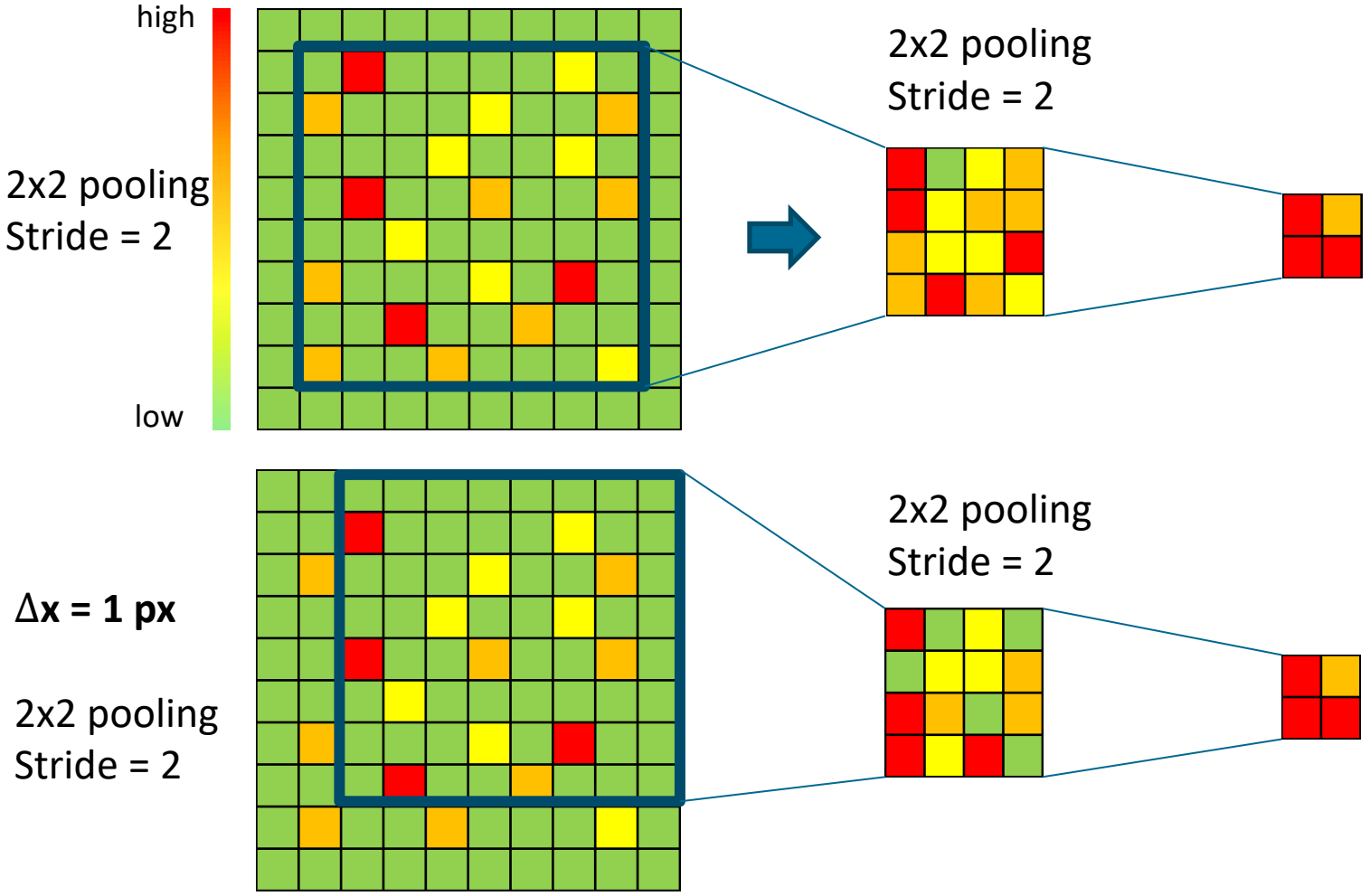
Why max pooling?



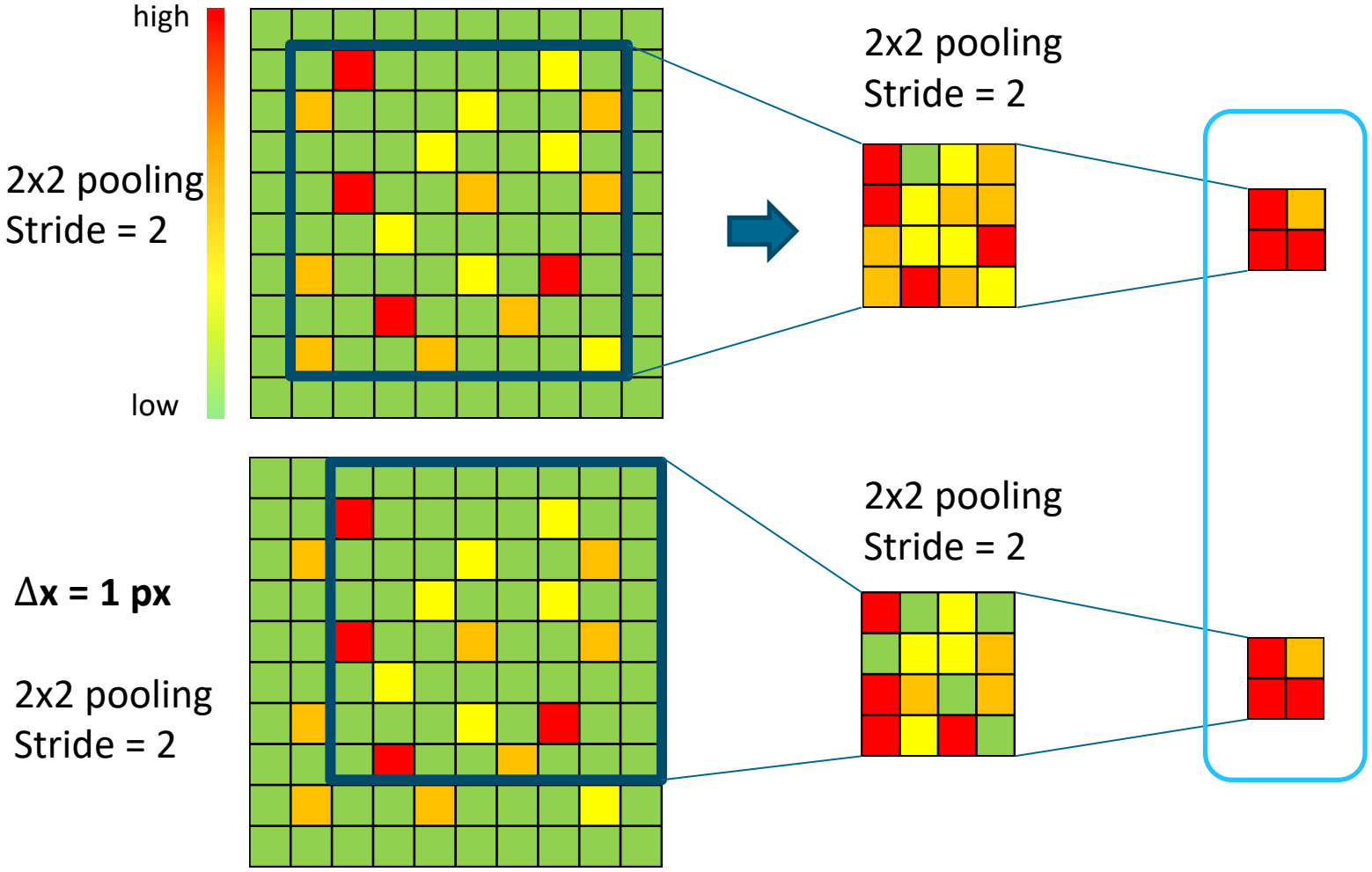
Why max pooling?



Why max pooling?



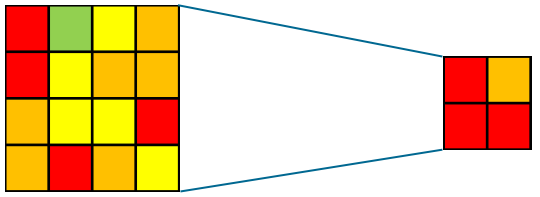
Why max pooling?



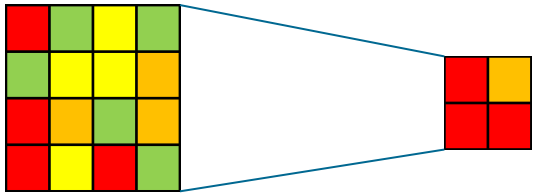
Why max pooling

- Pooling **reduces** the size of **feature maps** while keeping the highest activations
- Smaller feature maps are **good** for **memory footprint**
- Pooling introduces some kind of **translation invariance** in the network

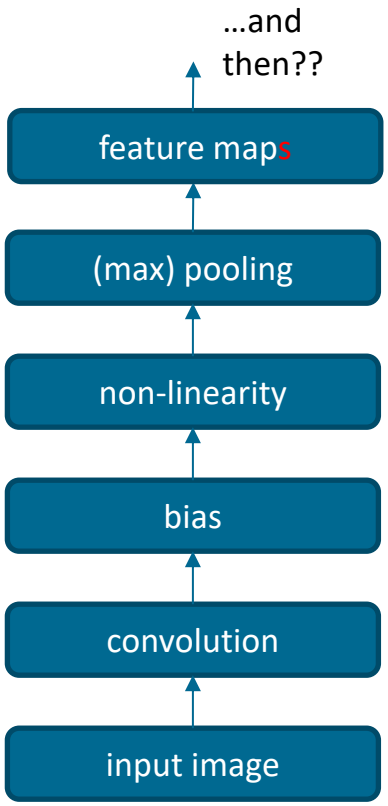
2x2 pooling
Stride = 2

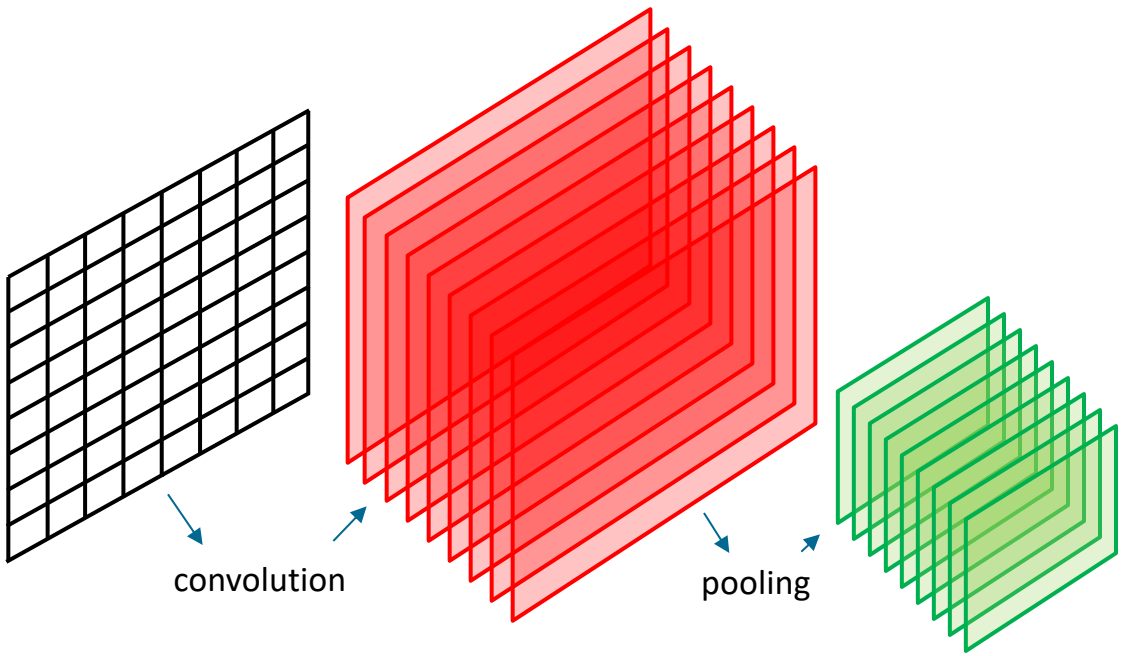


2x2 pooling
Stride = 2

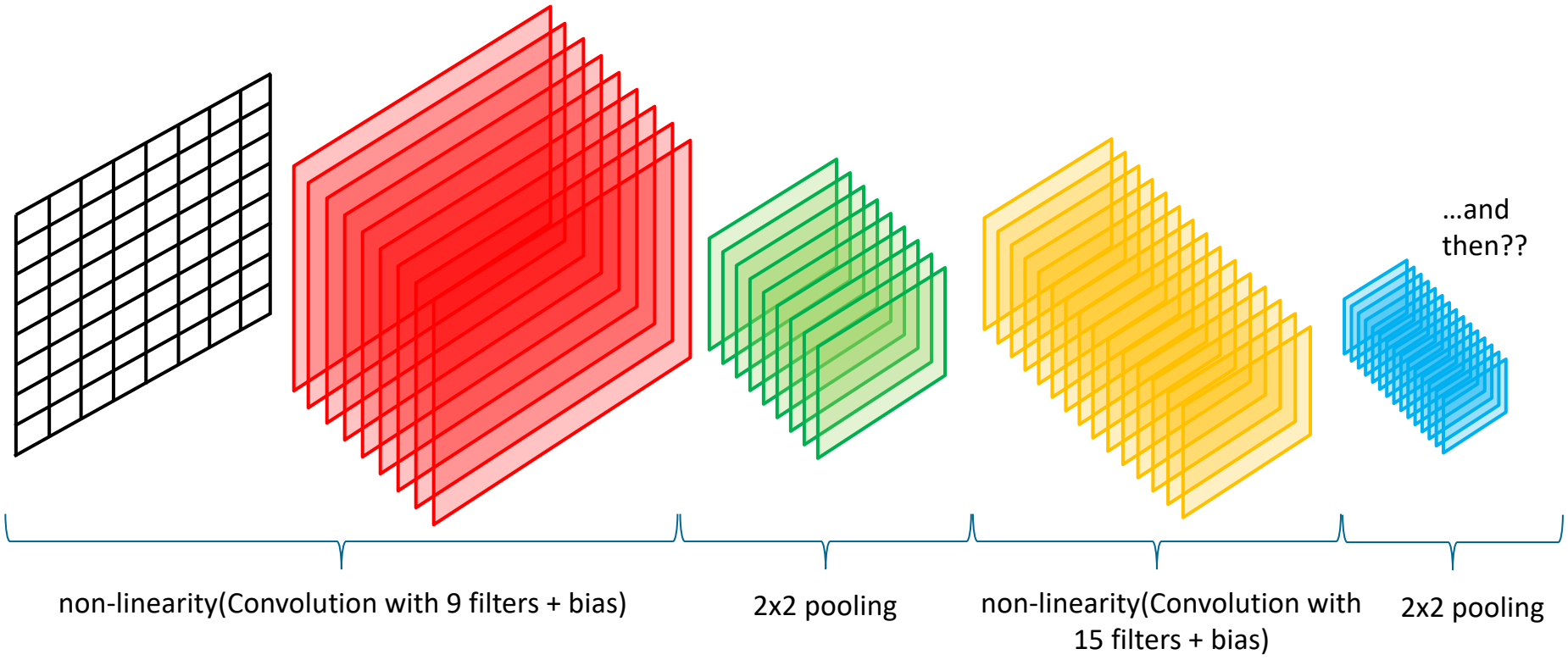


Convolutional Neural Network

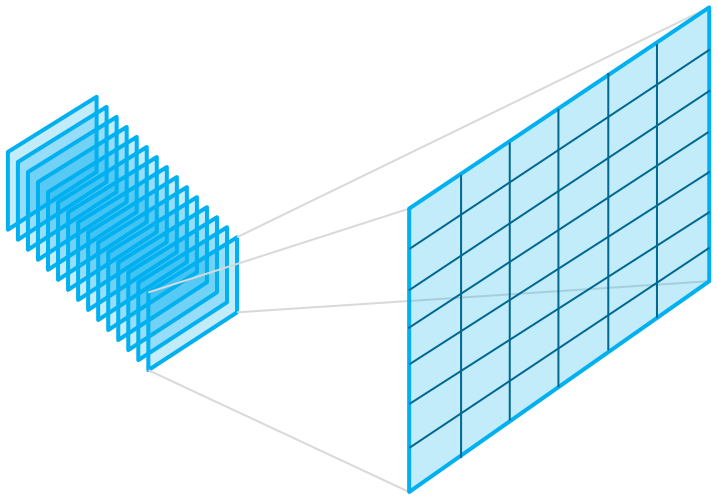




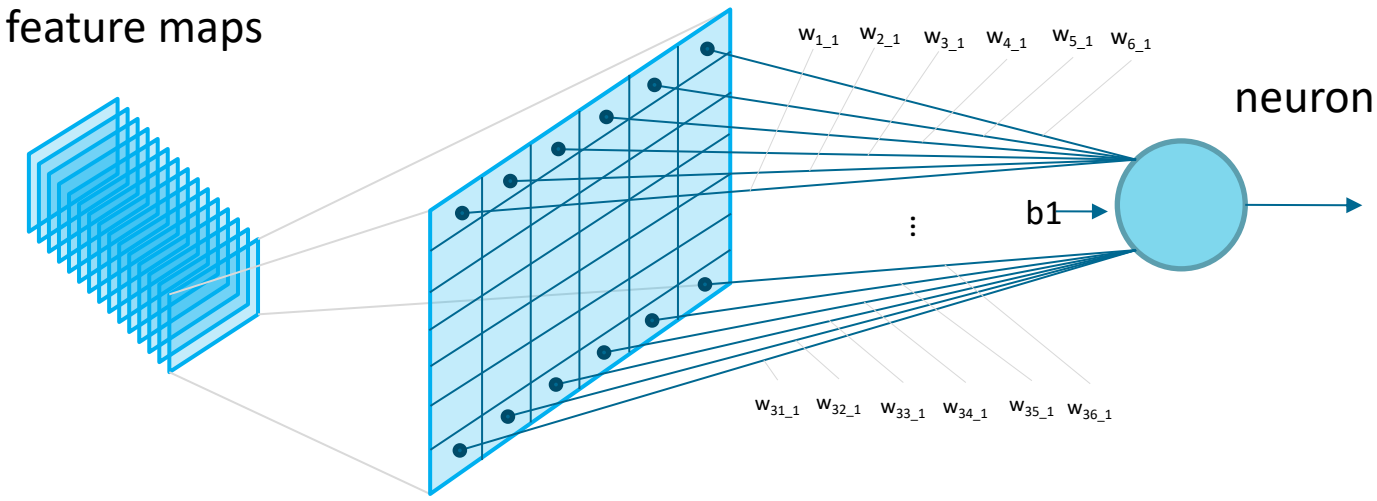
Convolutional Neural Network



Fully-connected layer

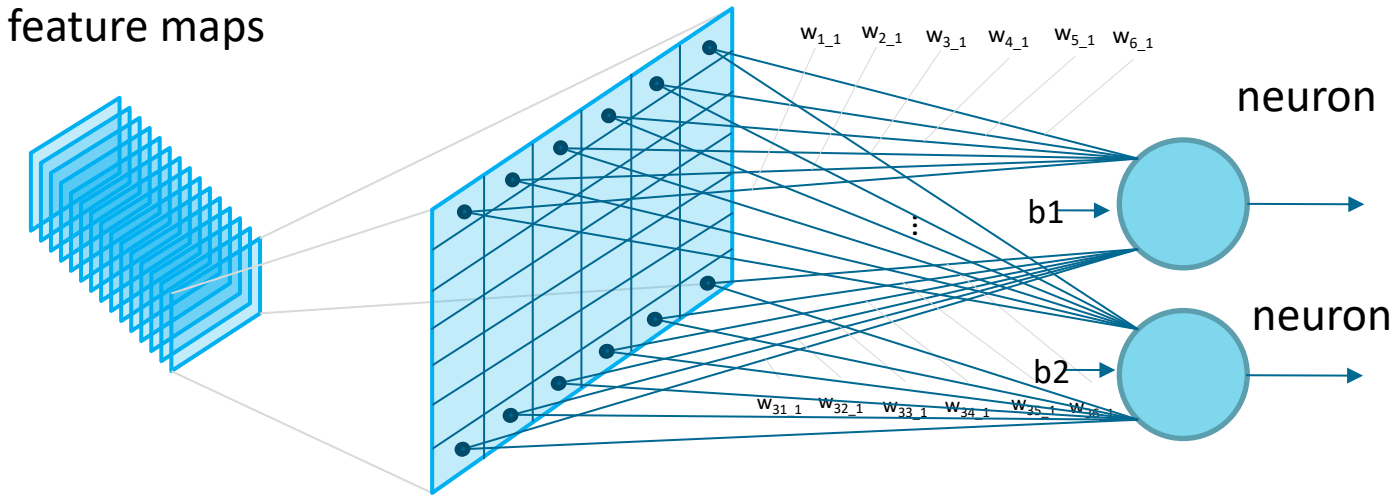


Fully-connected layer



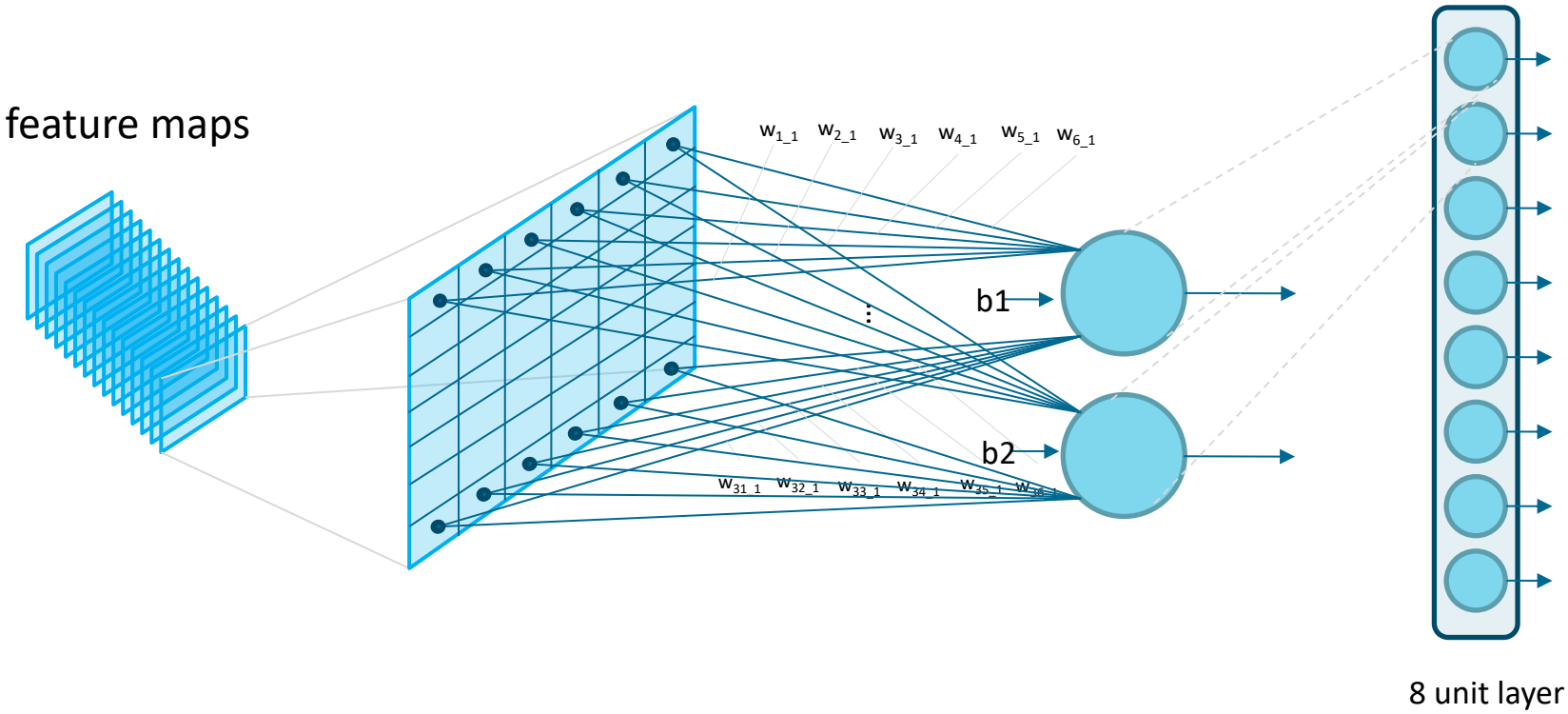
Fully-connected layer

- Each neuron is connected to **all activations** of **all feature maps** generated by the previous layer



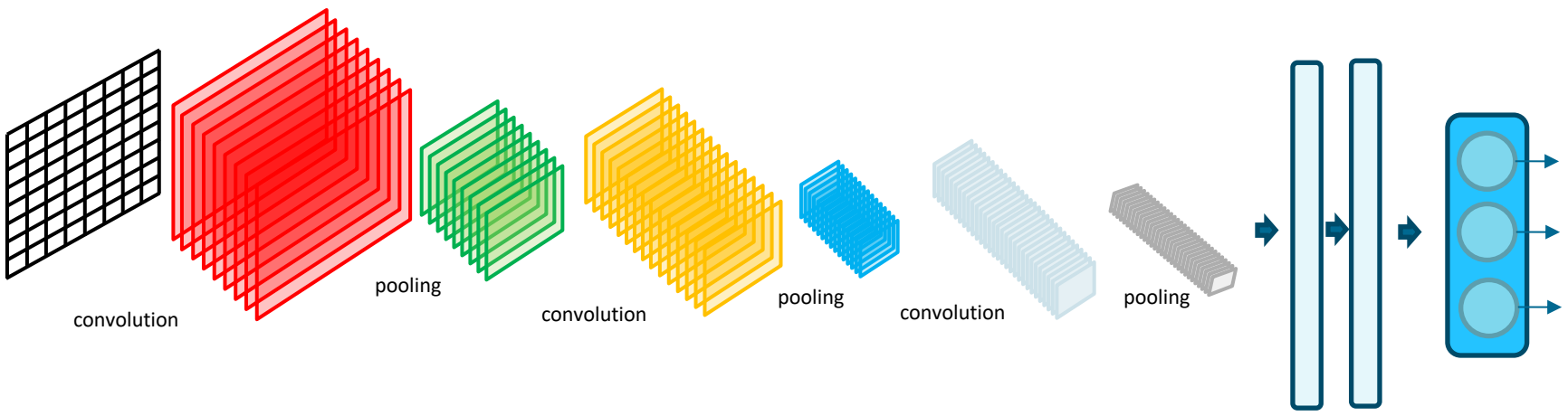
Fully-connected layer

- A layer of such a neuron type is called fully-connected layer

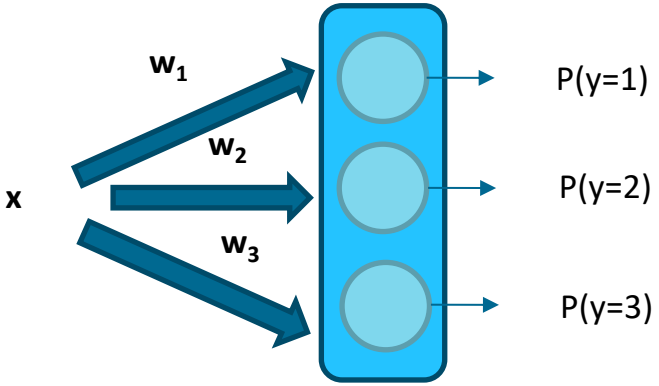


Convolutional Neural Network

Suppose we have a **three-class** problem



Soft-max layer (aka
- Multinomial Logistic Regression
- Polytomous Logistic Regression
- Multiclass Logistic Regression)
- Multinomial logit
- …



P(y=1)+P(y=2)+P(y=3) = 1

$$P(y=1) = \frac{\exp(\boldsymbol{w_1} \cdot \boldsymbol{x})}{\sum_{k=1}^{3} \exp(\boldsymbol{w_k} \cdot \boldsymbol{x})}$$

$$P(y=2) = \frac{\exp(\boldsymbol{w_2} \cdot \boldsymbol{x})}{\sum_{k=1}^{3} \exp(\boldsymbol{w_k} \cdot \boldsymbol{x})}$$

$$P(y=3) = \frac{\exp(\boldsymbol{w_3} \cdot \boldsymbol{x})}{\sum_{k=1}^{3} \exp(\boldsymbol{w_k} \cdot \boldsymbol{x})}$$

How many parameters?

- Input image size: 32x32x3, 10 classes (CIFAR10)
- We use **valid** convolutions

Layer type	Filter size	# filters (neurons)	# parameters	Output shape
Convolutional	3x3x3	32	3x3x3x32 + 32 = 896	30x30x32

How many parameters?

- Input image size: 32x32x3, 10 classes (CIFAR10)
- We use **valid** convolutions

Layer type	Filter size	# filters (neurons)	# parameters	Output shape
Convolutional	3x3x3	32	3x3x3x32 + 32 = 896	30x30x32
Pooling	**2x2**	**--**	**0**	**15x15x32**

How many parameters?

- Input image size: 32x32x3, 10 classes (CIFAR10)
- We use **valid** convolutions

Layer type	Filter size	# filters (neurons)	# parameters	Output shape
Convolutional	3x3x3	32	3x3x3x32 + 32 = 896	30x30x32
Pooling	2x2	--	0	15x15x32
Convolutional	**3x3x32**	**64**	**3x3x32x64 + 64 = 18,496**	**13x13x64**

How many parameters?

- Input image size: 32x32x3, 10 classes (CIFAR10)
- We use **valid** convolutions

Layer type	Filter size	# filters (neurons)	# parameters	Output shape
Convolutional	3x3x3	32	3x3x3x32 + 32 = 896	30x30x32
Pooling	2x2	--	0	15x15x32
Convolutional	3x3x32	64	3x3x32x64 + 64 = 18,496	13x13x64
Pooling	**2x2**		**0**	**6x6x64**

How many parameters?

- Input image size: 32x32x3, 10 classes (CIFAR10)
- We use **valid** convolutions

Layer type	Filter size	# filters (neurons)	# parameters	Output shape
Convolutional	3x3x3	32	3x3x3x32 + 32 = 896	30x30x32
Pooling	2x2	--	0	15x15x32
Convolutional	3x3x32	64	3x3x32x64 + 64 = 18,496	13x13x64
Pooling	2x2		0	6x6x64
Convolutional	**3x3x64**	**128**	**3x3x64x128 + 128 = 73,856**	**4x4x128**

How many parameters?

- Input image size: 32x32x3, 10 classes (CIFAR10)
- We use **valid** convolutions

Layer type	Filter size	# filters (neurons)	# parameters	Output shape
Convolutional	3x3x3	32	3x3x3x32 + 32 = 896	30x30x32
Pooling	2x2	--	0	15x15x32
Convolutional	3x3x32	64	3x3x32x64 + 64 = 18,496	13x13x64
Pooling	2x2		0	6x6x64
Convolutional	3x3x64	128	3x3x64x128 + 128 = 73,856	4x4x128
Pooling	**2x2**		**0**	**2x2x128**

How many parameters?

- Input image size: 32x32x3, 10 classes (CIFAR10)
- We use **valid** convolutions

Layer type	Filter size	# filters (neurons)	# parameters	Output shape
Convolutional	3x3x3	32	3x3x3x32 + 32 = 896	30x30x32
Pooling	2x2	--	0	15x15x32
Convolutional	3x3x32	64	3x3x32x64 + 64 = 18,496	13x13x64
Pooling	2x2		0	6x6x64
Convolutional	3x3x64	128	3x3x64x128 + 128 = 73,856	4x4x128
Pooling	2x2		0	2x2x128
Fully-connected	**--**	**256**	**2x2x128x256 + 256 = 131,328**	**1x256**

How many parameters?

- Input image size: 32x32x3, 10 classes (CIFAR10)
- We use **valid** convolutions

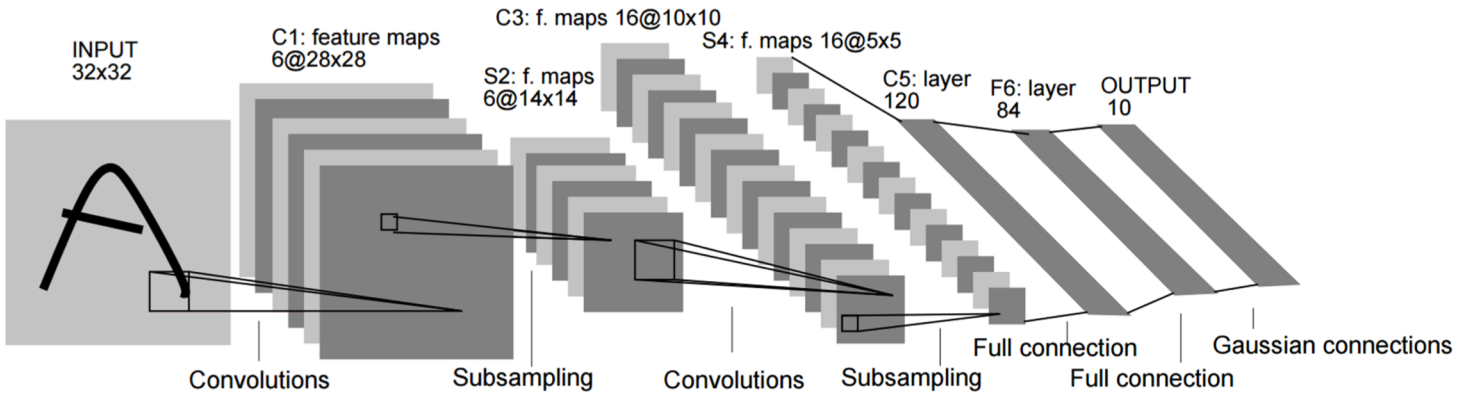
Layer type	Filter size	# filters (neurons)	# parameters	Output shape
Convolutional	3x3x3	32	3x3x3x32 + 32 = 896	30x30x32
Pooling	2x2	--	0	15x15x32
Convolutional	3x3x32	64	3x3x32x64 + 64 = 18,496	13x13x64
Pooling	2x2		0	6x6x64
Convolutional	3x3x64	128	3x3x64x128 + 128 = 73,856	4x4x128
Pooling	2x2		0	2x2x128
Fully-connected	--	256	2x2x128x256 + 256 = 131,328	1x256
Softmax	**--**	**10**	**10 x 256+ 10 = 2,570**	**1x10**

How many parameters?

- Input image size: 32x32x3, 10 classes (CIFAR10)
- We use **valid** convolutions

Layer type	Filter size	# filters (neurons)	# parameters	Output shape
Convolutional	3x3x3	32	3x3x3x32 + 32 = 896	30x30x32
Pooling	2x2	--	0	15x15x32
Convolutional	3x3x32	64	3x3x32x64 + 64 = 18,496	13x13x64
Pooling	2x2		0	6x6x64
Convolutional	3x3x64	128	3x3x64x128 + 128 = 73,856	4x4x128
Pooling	2x2		0	2x2x128
Fully-connected	--	256	2x2x128x256 + 256 = 131,328	1x256
Softmax	--	10	10 x 256+ 10 = 2,570	1x10
TOTAL			**227,246 parameters**	

Parameters: LeNet (1998)



Total: 60,000

Parameters: AlexNet (2012)

C1: 2*48*(11*11*3+1)=34944

S2: 0

C3: 2*128*(5*5*48+1)=307456

S4: 0

C5: 2*192*(3*3*128+1)=442752

C6: 2*192*(3*3*192+1)=663936

C7: 2*128*(3*3*192+1)=442624

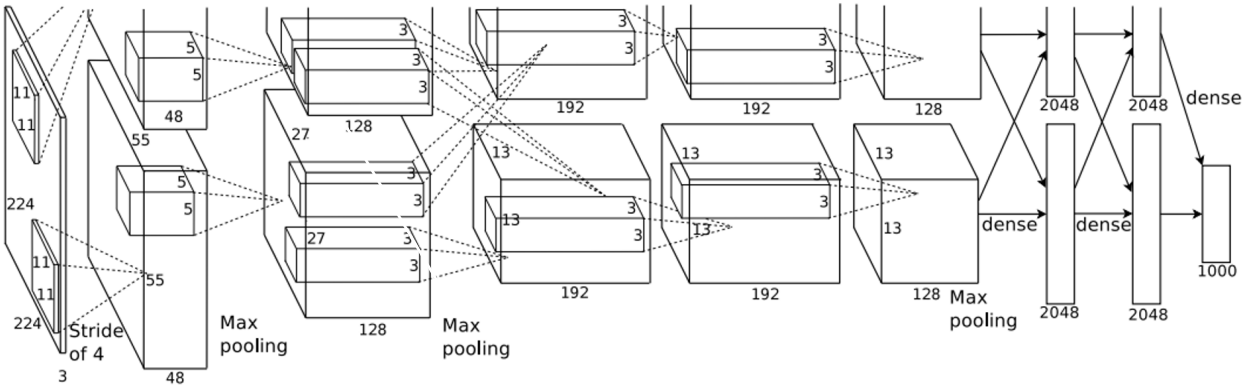
S8: 0

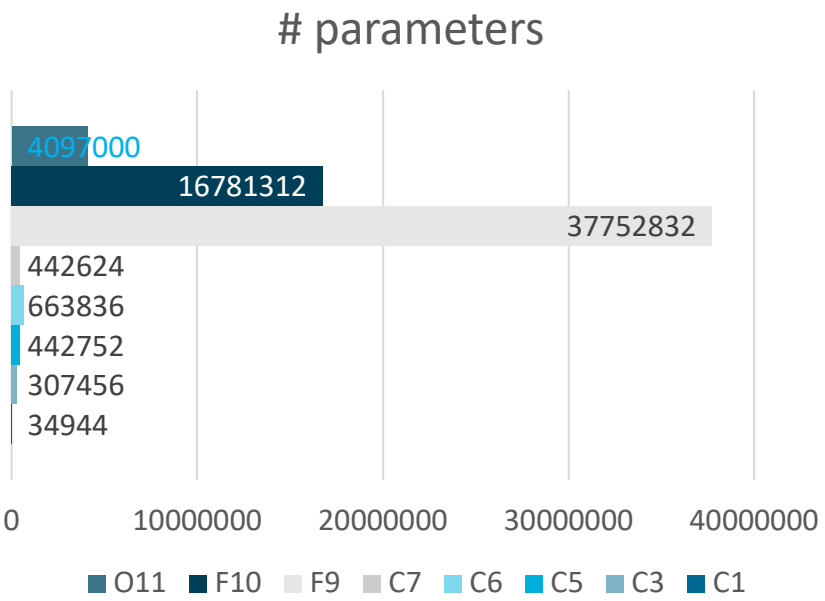
F9: 4096*(2*6*6*128+1)=**37752832**

F10: 4096*(4096+1)=16781312

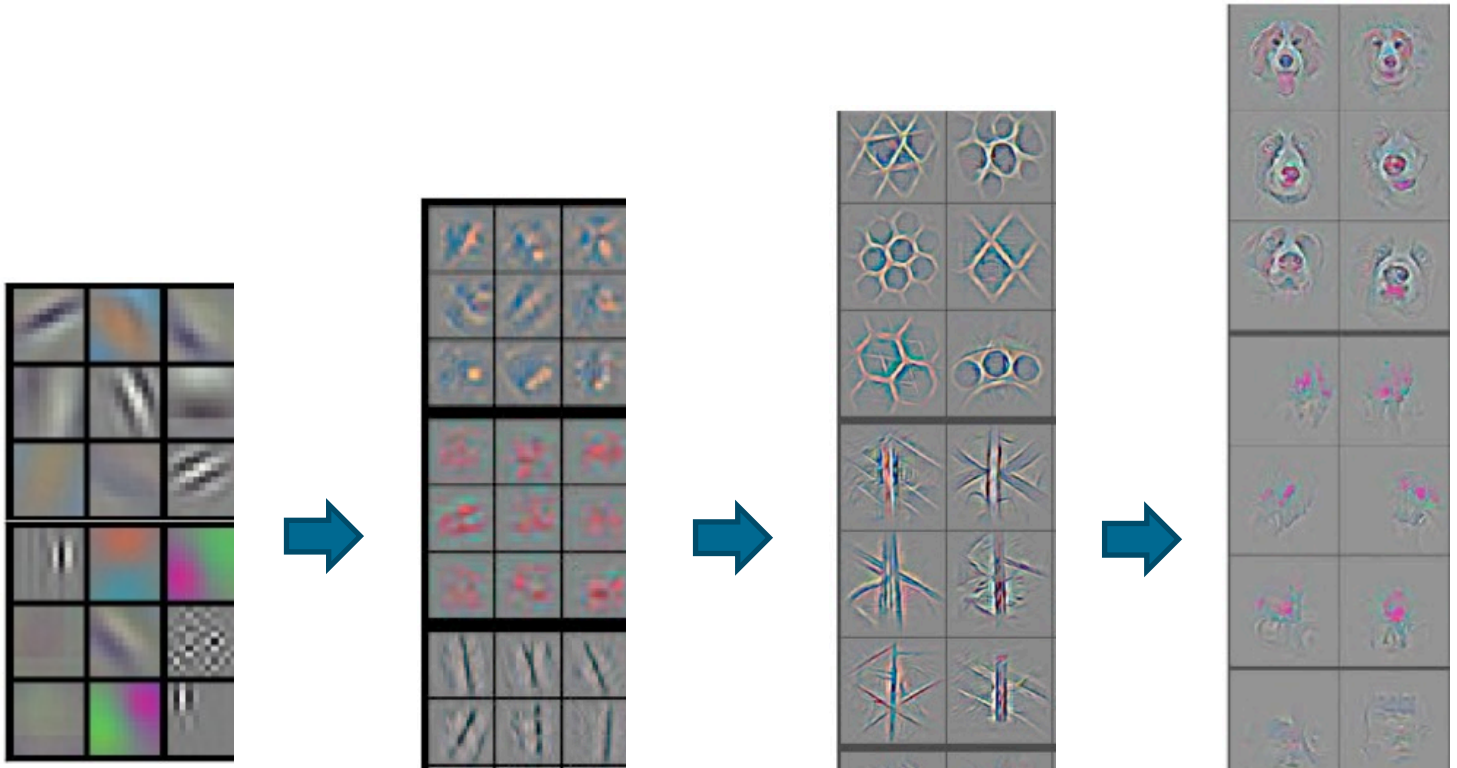
O11: 1000*(4096+1)=4097000

Total: 60,522,856 (1000 times more than LeNet!)





Hierarchical representation



Layer 1

Layer 2

Layer 3

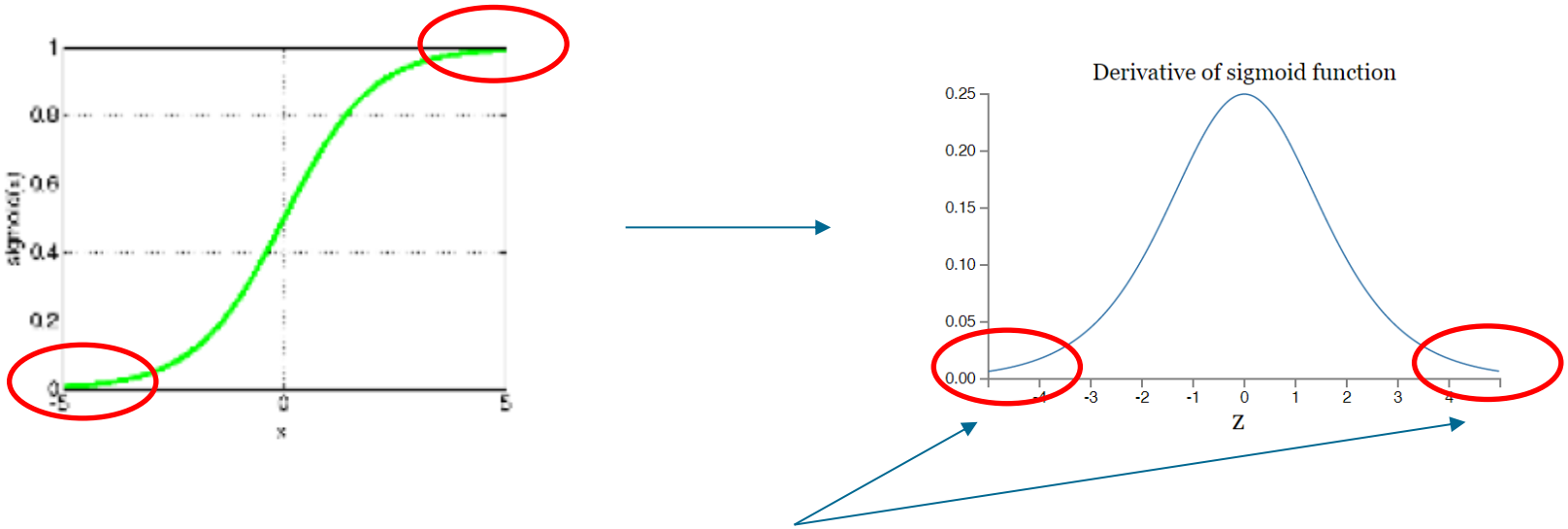
Layer 4

Zeiler and Fergus, 2013

ReLU

- What's the **problem** of **sigmoid** activation functions?

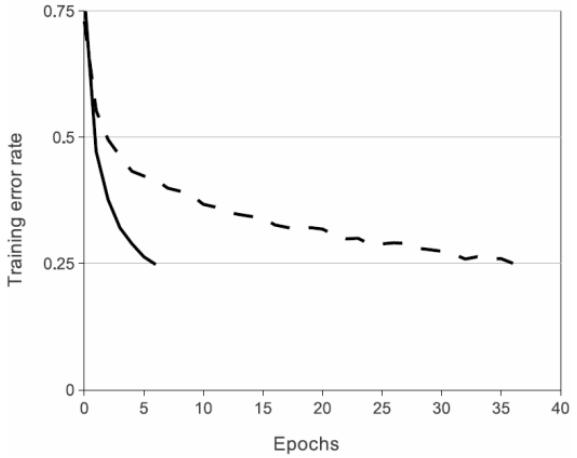
- Remember that we train with **gradient descent** (derivatives) and **backpropagation** (chain rule)

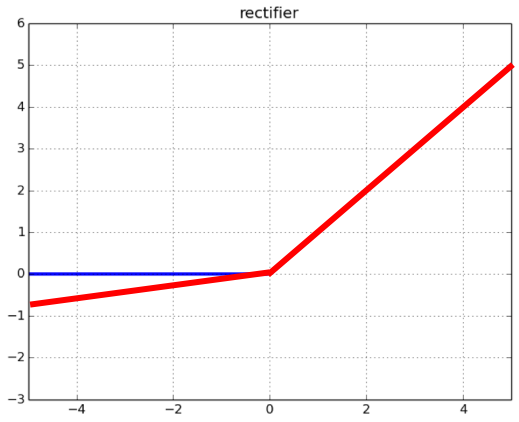


The derivative here is 0! => **kills the backpropagation of gradient**

ReLU

- __/ computes the function $f(x) = \max(0, x)$
- Greatly **accelerates convergence of SGD** compared to sigmoid/tanh function.
- Very inexpensive
- Doesn't face **vanishing gradient problem**

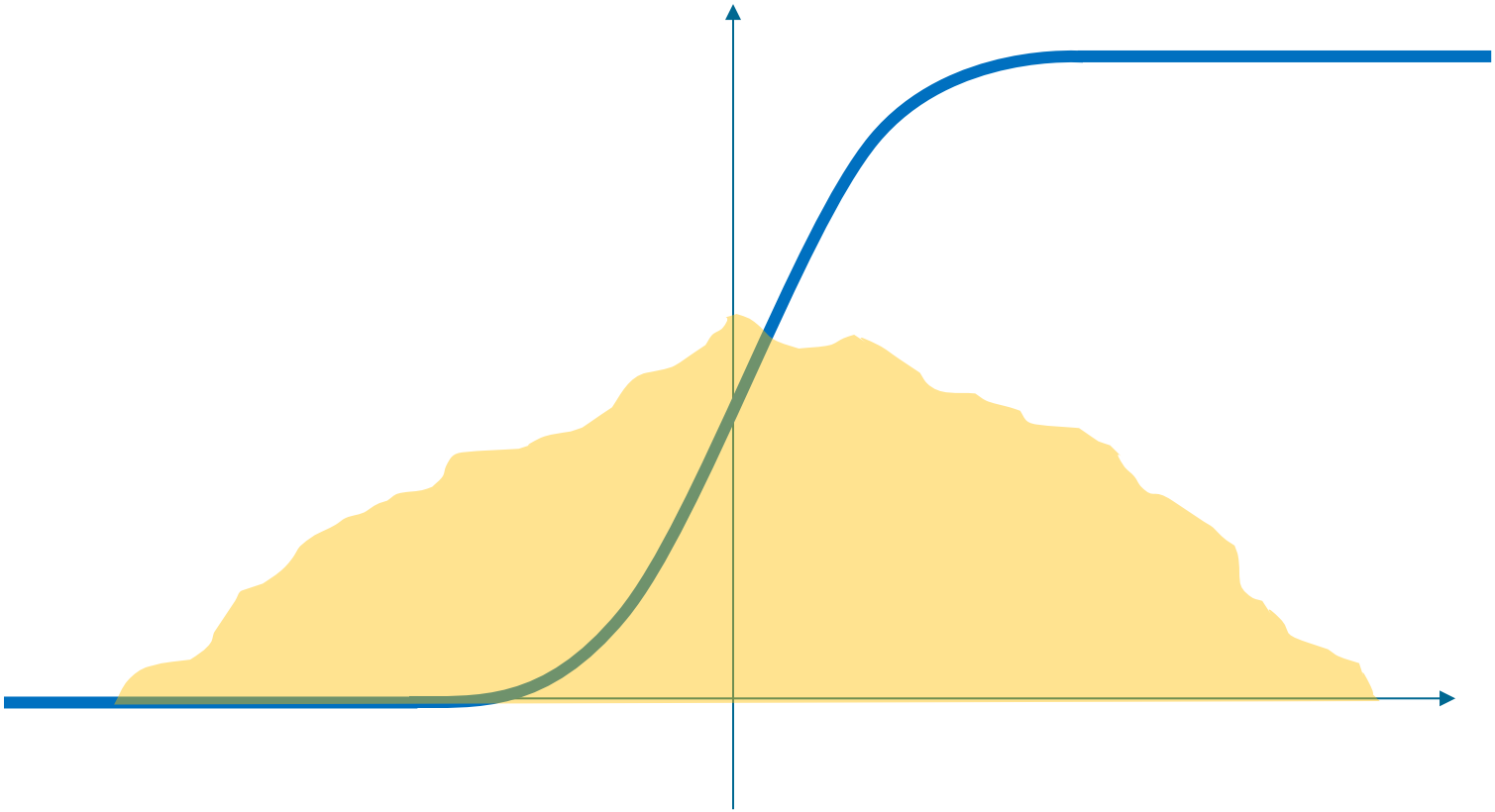




- Leaky ReLUs are one attempt to fix the "dying ReLU" problem.

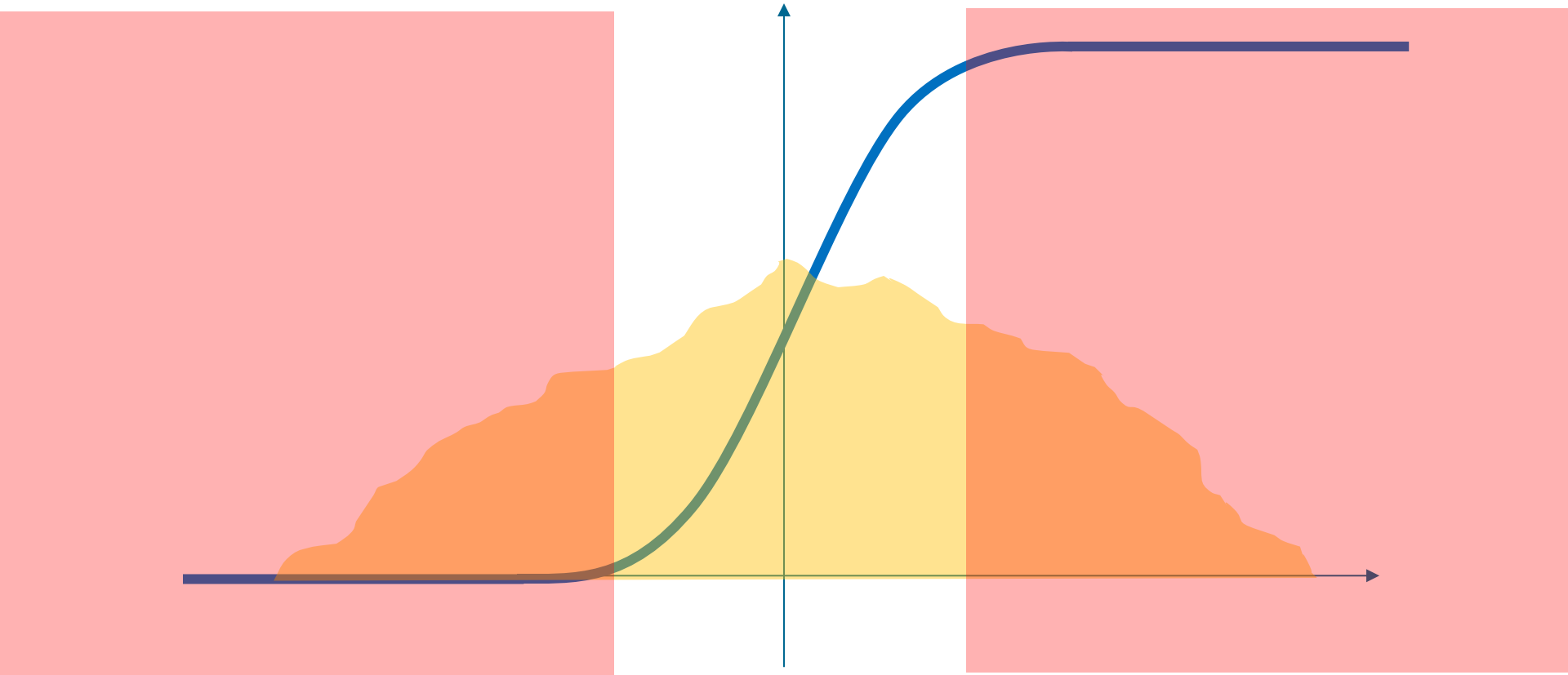
Initialization

- The **distribution of initial activations** is important to ensure a good learning procedure



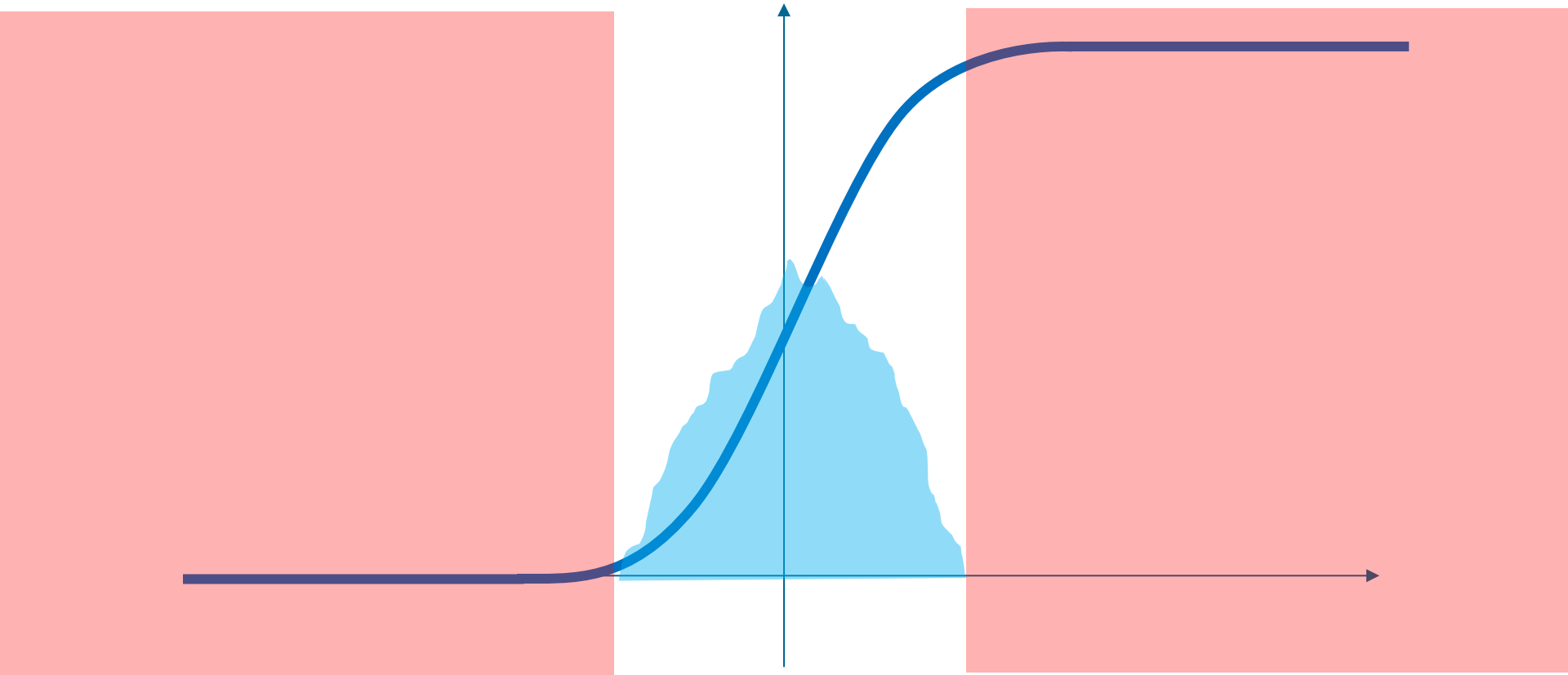
Initialization

- The **distribution of initial activations** is important to ensure a good learning procedure



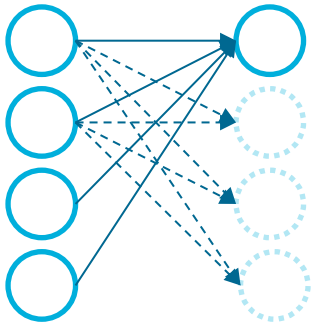
Initialization

- The **distribution of initial activati**ons is important to ensure a good learning procedure



Initialization

Glorot, 2010



100 neurons

- In the previous approach, as the **number of inputs** to a neuron **grows**, so too will its **output's variance**.
- This can be **calibrated** by scaling its weight vector by the square root of its "*fan-in*"
- Initialize the weights input to the neuron with a **Gaussian** with $\mathrm{N}(0, \sqrt{\frac{1}{\mathrm{n_{in}}}})$
- All neurons in the network initially have approximately the same output distribution
- Empirically improves the rate of convergence.

Initialization

He, 2015

- Glorot does not focus on specific non linearities
- He: specific for ReLU non-linearities
- ReLU weights should be sampled from zero-mean Gaussian distribution with standard deviation of $\sqrt{\frac{2}{n_{in}}}$
- Recommendation: use **ReLU units** and the initialization strategy by **HE et al**.

We let the initialized elements in W_l be mutually independent and share the same distribution. As in [7], we assume that the elements in $\mathbf{x}_l$ are also mutually independent and share the same distribution, and $\mathbf{x}_l$ and W_l are independent of each other. Then we have:

$$Var[y_l] = n_l Var[w_l x_l], \tag{6}$$

where now y_l, x_l, and w_l represent the random variables of each element in $\mathbf{y}_l$, W_l, and $\mathbf{x}_l$ respectively. We let w_l have zero mean. Then the variance of the product of independent variables gives us:

$$Var[y_l] = n_l Var[w_l] E[x_l^2]. \tag{7}$$

Here $E[x_l^2]$ is the expectation of the square of x_l. It is worth noticing that $E[x_l^2] \neq Var[x_l]$ unless x_l has zero mean. For the ReLU activation, $x_l = max(0, y_{l-1})$ and thus it does not have zero mean. This will lead to a conclusion different from [7].

If we let w_{l-1} have a symmetric distribution around zero and $b_{l-1} = 0$, then y_{l-1} has zero mean and has a symmetric distribution around zero. This leads to $E[x_l^2] = \frac{1}{2} Var[y_{l-1}]$ when f is ReLU. Putting this into Eqn.(7), we obtain:

$$Var[y_l] = \frac{1}{2} n_l Var[w_l] Var[y_{l-1}]. \tag{8}$$

With L layers put together, we have:

$$Var[y_L] = Var[y_1] \left(\prod_{l=2}^{L} \frac{1}{2} n_l Var[w_l] \right). \tag{9}$$

This product is the key to the initialization design. A proper initialization method should avoid reducing or magnifying the magnitudes of input signals exponentially. So we expect the above product to take a proper scalar (*e.g.*, 1). A sufficient condition is:

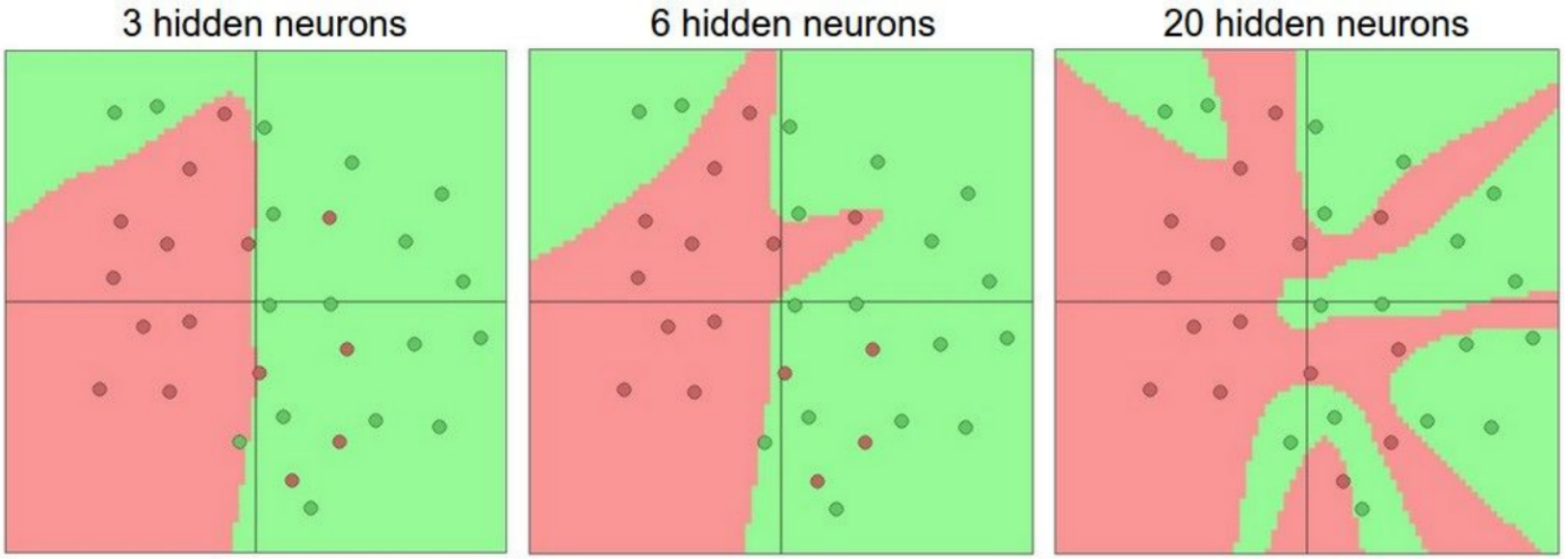
$$\frac{1}{2} n_l Var[w_l] = 1, \quad \forall l. \tag{10}$$

This leads to a zero-mean Gaussian distribution whose standard deviation (std) is $\sqrt{2/n_l}$. This is our way of initialization. We also initialize $\mathbf{b} = 0$.

For the first layer ($l = 1$), we should have $n_1 Var[w_1] = 1$ because there is no ReLU applied on the input signal. But the factor $1/2$ does not matter if it just exists on one layer. So we also adopt Eqn.(10) in the first layer for simplicity.

Neurons in hidden layer

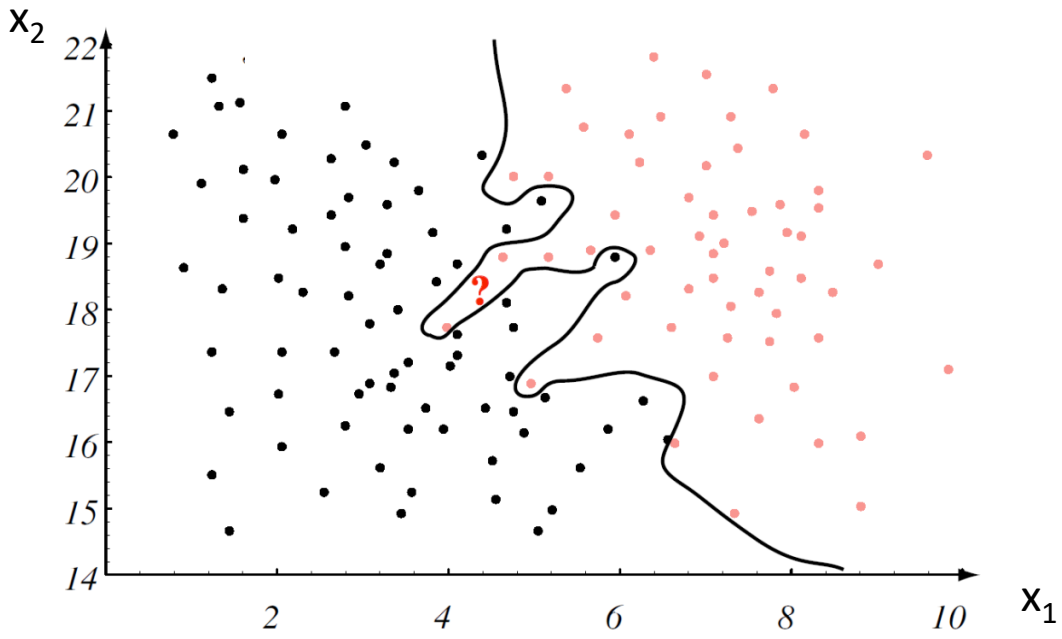
Demo = http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html



more neurons = more capacity

Overfitting?

- If we have many parameters in our network, we can **learn the training set by heart! OVERFITTING**
- But many parameters are also needed to learn complex function!



We want to learn from training data and do a good job on unseen data!

Regularization

- You can think of overfitting as due to *"too much freedom"* to the parameters of our model

- How can we "limit" this freedom?
  - **WEIGHT REGULARIZATION**

- We can **modify the loss function** by adding a term that discourages large weights:

$$L = L_{data}(x, W) + \lambda R(\boldsymbol{W})$$

What we defined already

Regularization loss

Regularization strength

Hyper-parameter defined by the user

Regularization

- Two types of regularization are typically used in the loss function:
  - **L1 regularization**: $R(W) = \sum_i |w_i|$
  - **L2 regularization**: $R(W) = \sum_i w_i^2$

- `kernel_regularizer`: instance of `keras.regularizers.Regularizer`
- `bias_regularizer`: instance of `keras.regularizers.Regularizer`
- `activity_regularizer`: instance of `keras.regularizers.Regularizer`

Example

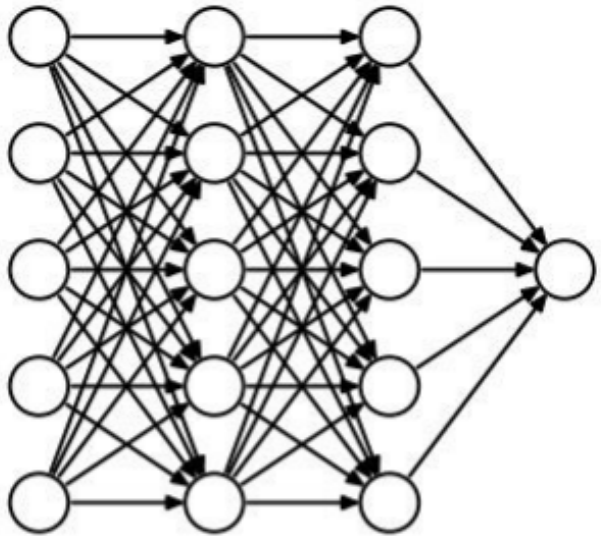
```
from keras import regularizers
model.add(Dense(64, input_dim=64,
                kernel_regularizer=regularizers.l2(0.01),
                activity_regularizer=regularizers.l1(0.01)))
```

Available penalties

```
keras.regularizers.l1(0.)
keras.regularizers.l2(0.)
keras.regularizers.l1_l2(0.)
```

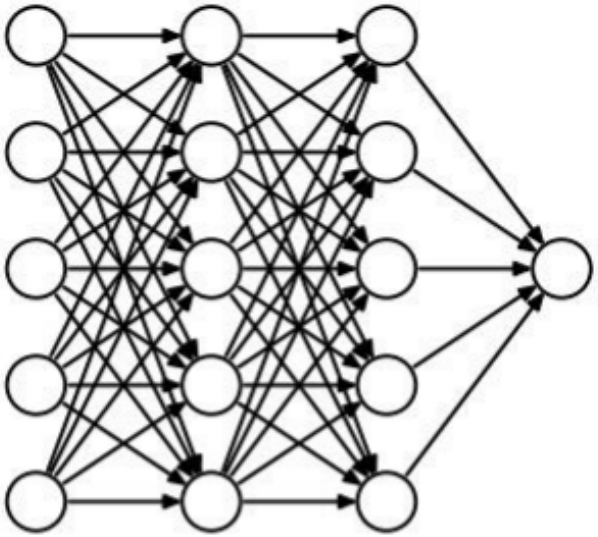
Dropout

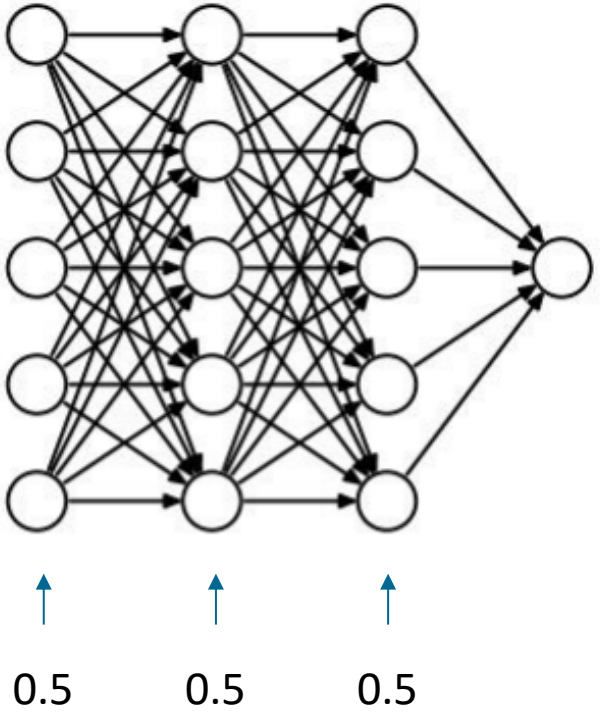
- Let’s consider this network



Dropout

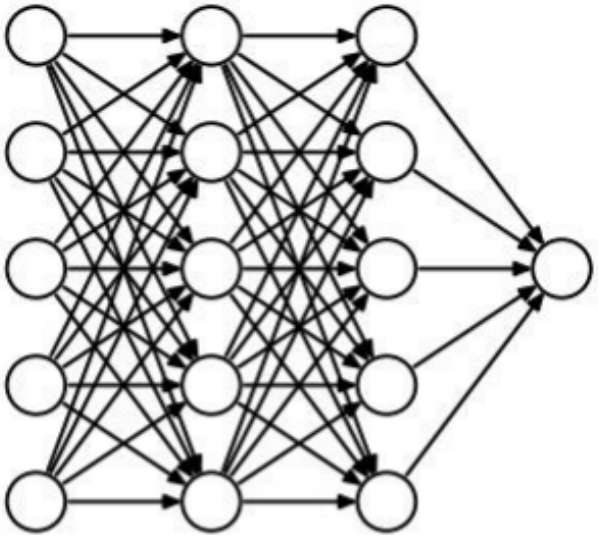
- Let's **toss a coin** (p=0.5) for each neuron and **remove it** accordingly

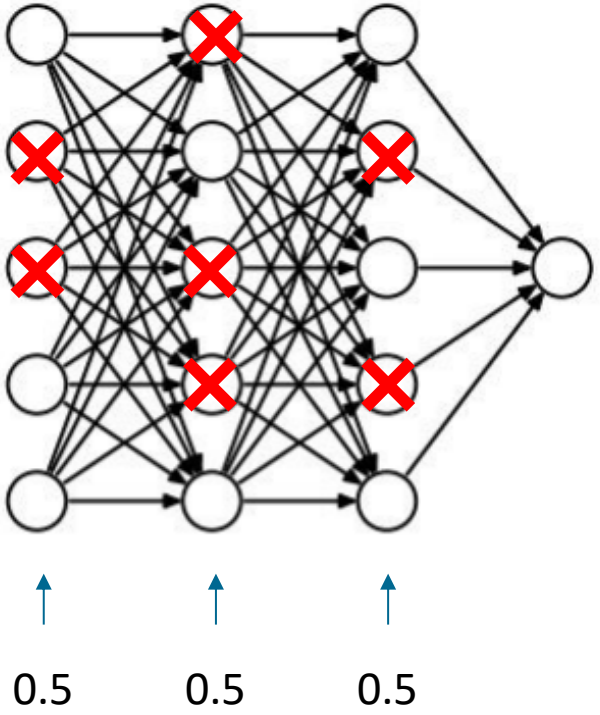




Dropout

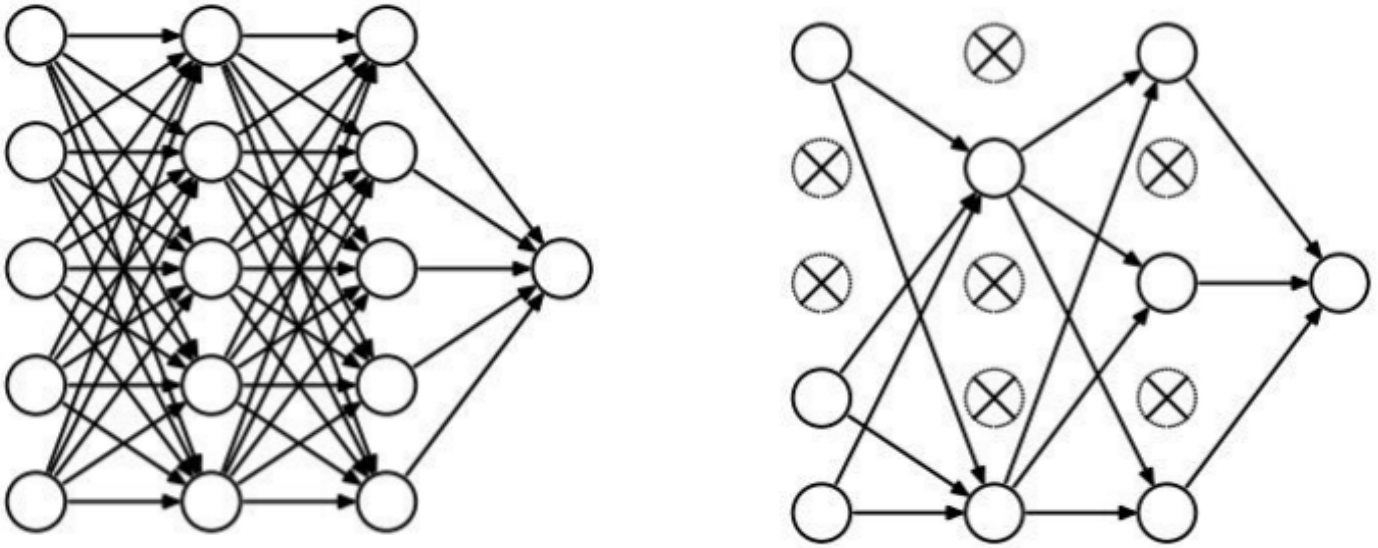
- We do this **for each training sample**





Dropout

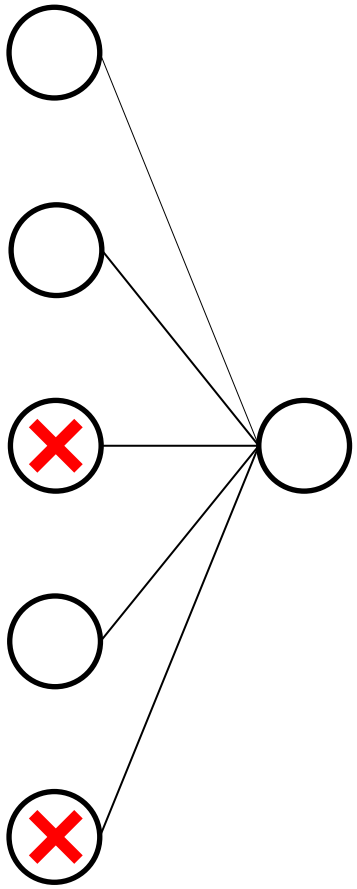
- We are left with a much smaller network



- **Each sample** is processed by a **different network**
  - Less parameters = less risk of overfitting

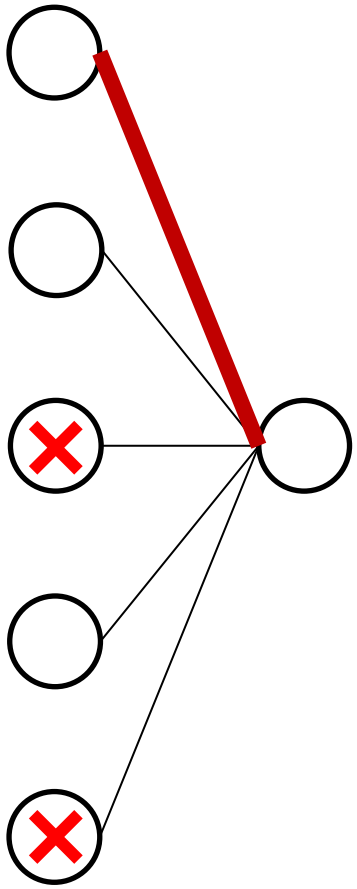
Dropout

- Each neuron **cannot rely on any one feature**



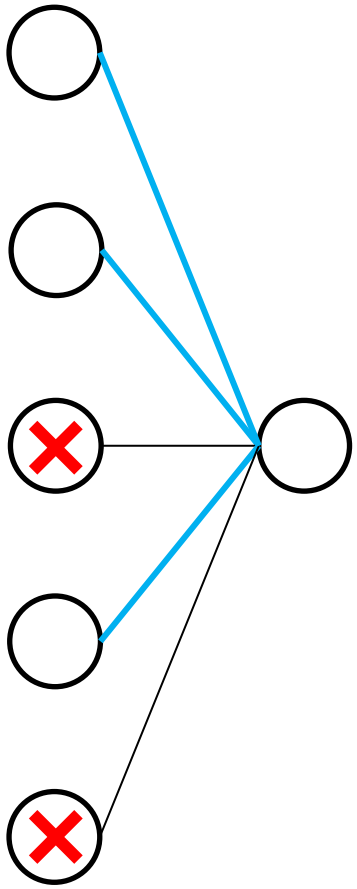
Dropout

- Each neuron cannot rely on any one feature
  - **Does not "bet"** on only one specific input



Dropout

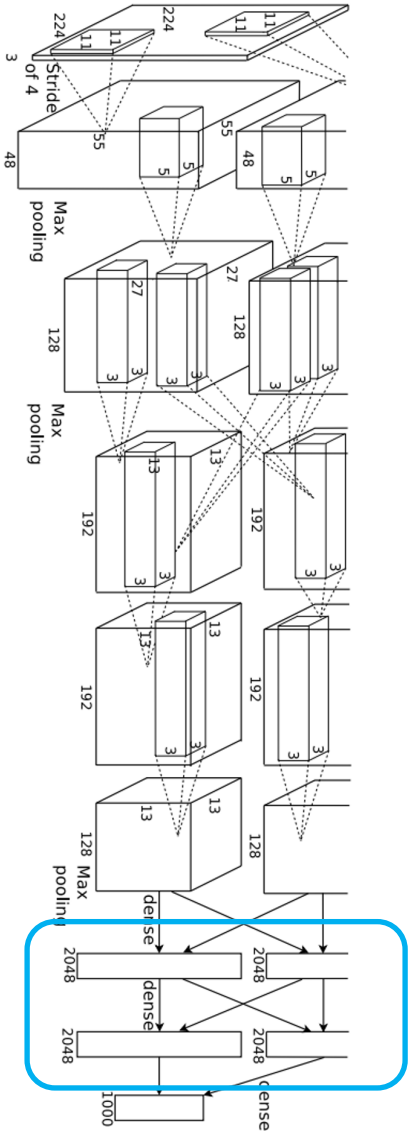
- Each neuron cannot rely on any one feature
  - Does not “bet” on only one specific input
  - **Spread out weights** -> shrinks the norm of weights ~ L2 regularization



Dropout

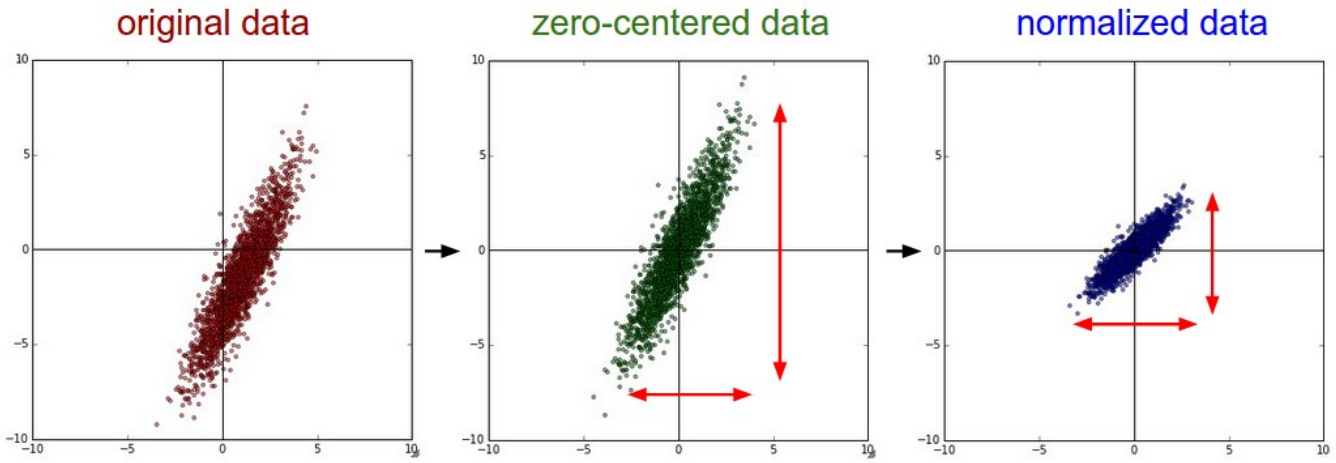
- Can be seen as training many neural networks at once
  - Reminds of **ensemble** learning
  - Typically improves (slightly) performance

- Typically done in **fully-connected layers**
  - But people use it also for convolutional layers



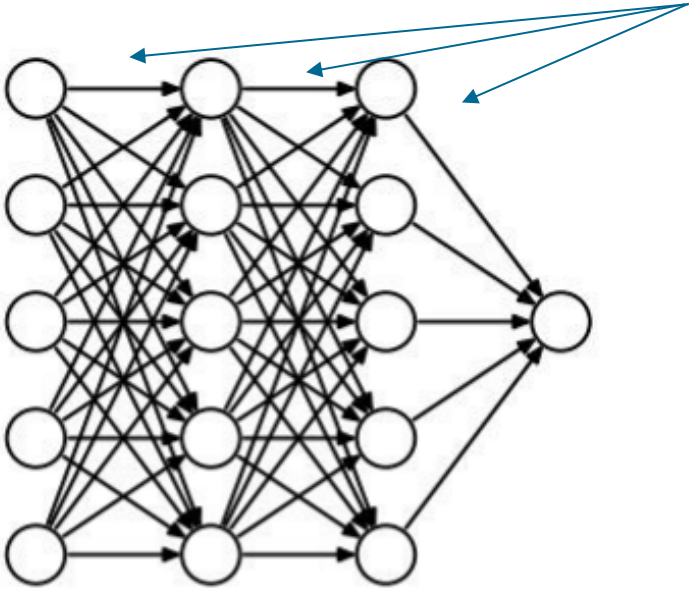
Batch normalization

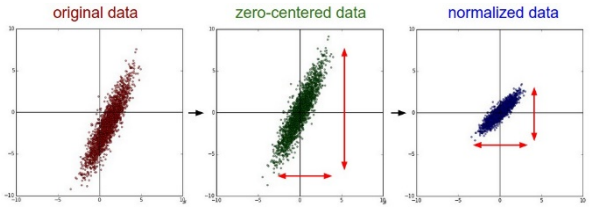
- We know that **normalizing inputs helps** training
  - In kNN, it makes features more “comparable”
  - In SGD, it helps the learning algorithm find the best parameters: makes training quicker!



Batch normalization

- In a **neural networks**, **each layer** takes activations of **previous layers** as input
- It would be nice to have them **normalized as well**, to make overall **learning more efficient**, faster



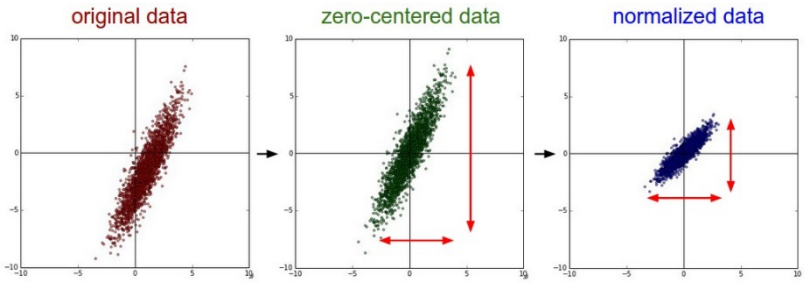


Batch normalization

- All the efforts on **initialization** was to achieve **unit Gaussian activations**.
  - Why don't we explicitly **force them to have unit Gaussians**?

- This is possible because **normalization** is a simple **differentiable** operation.

$$\hat{x}^{(k)} = \frac{x^{(k)} - E\left[x^{(k)}\right]}{\sqrt{Var\left[x^{(k)}\right]}}$$



Batch normalization

- Normalizes activations = makes **learning faster**

- Makes **weights** of a deep layer **more robust to changes** of weights in previous layers
  - Makes sure that at least the **mean** value and the **variance don't change**

- Has some **regularization** effect
  - Mean and variance are computed per mini-batch
  - Mini-batches are randomly sampled
  - This adds randomness to the normalized values
  - Similar to dropout, if we use small mini-batches…

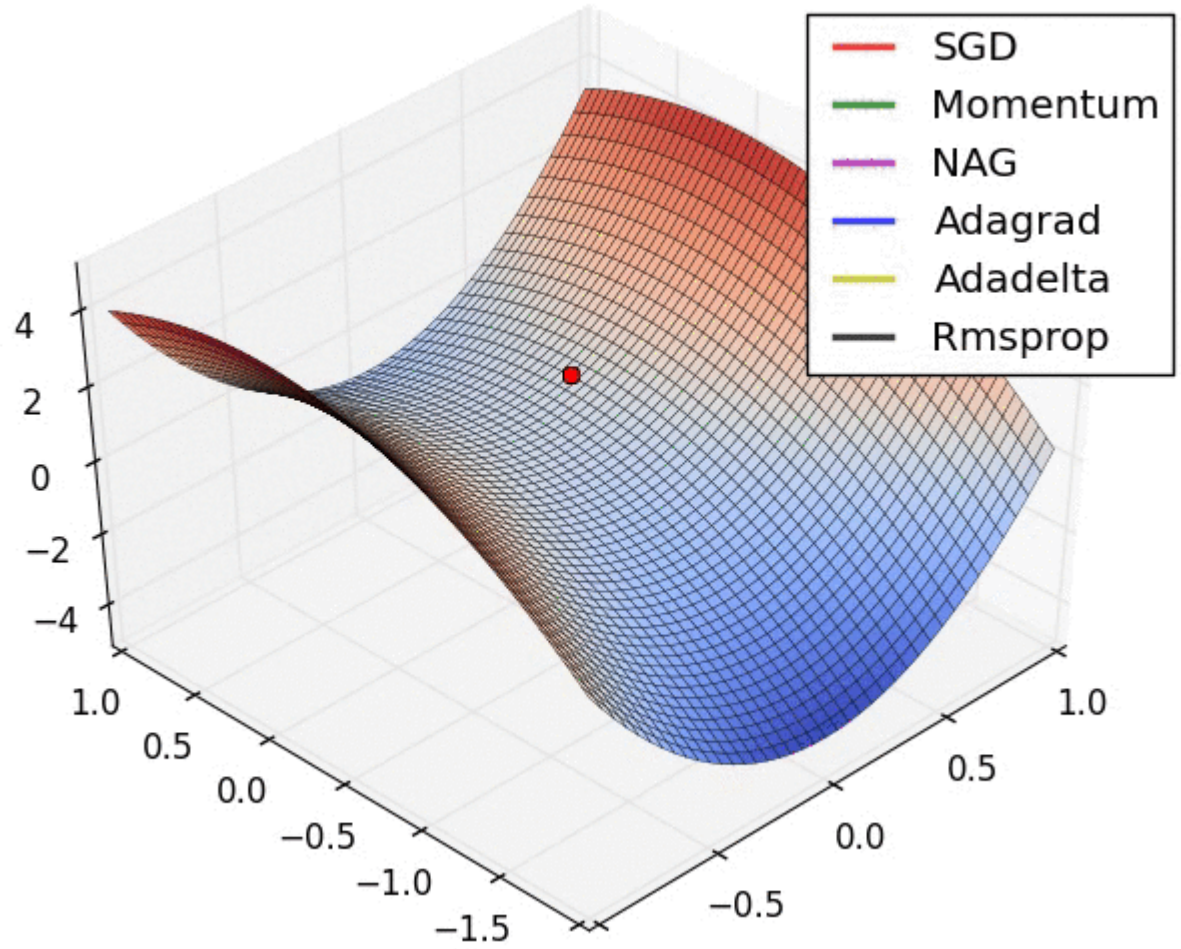
Batch normalization and Dropout

Where do you have to put these layers?

- **Dropout**: after the layer with parameters
- **Batch normalization**: after the layer with parameters but **before** the non linearity

```
model.add(Dense(64, input_dim=14, init='uniform'))
model.add(BatchNormalization())
model.add(Activation('tanh'))
model.add(Dropout(0.5))
```

Update rules



Credits: Alec Radford

Update rules

- Last week, we saw the basic rule to update parameters using stochastic gradient descent:

$$\boldsymbol{W}_t = \boldsymbol{W}_{t-1} - \eta \nabla_{\boldsymbol{W}} \, \mathrm{L}(\mathbf{W})\Big|_{\boldsymbol{W}=\boldsymbol{W}_{t-1}}$$

SGD

```
keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)
```

Stochastic gradient descent optimizer.

Includes support for momentum, learning rate decay, and Nesterov momentum.

Arguments

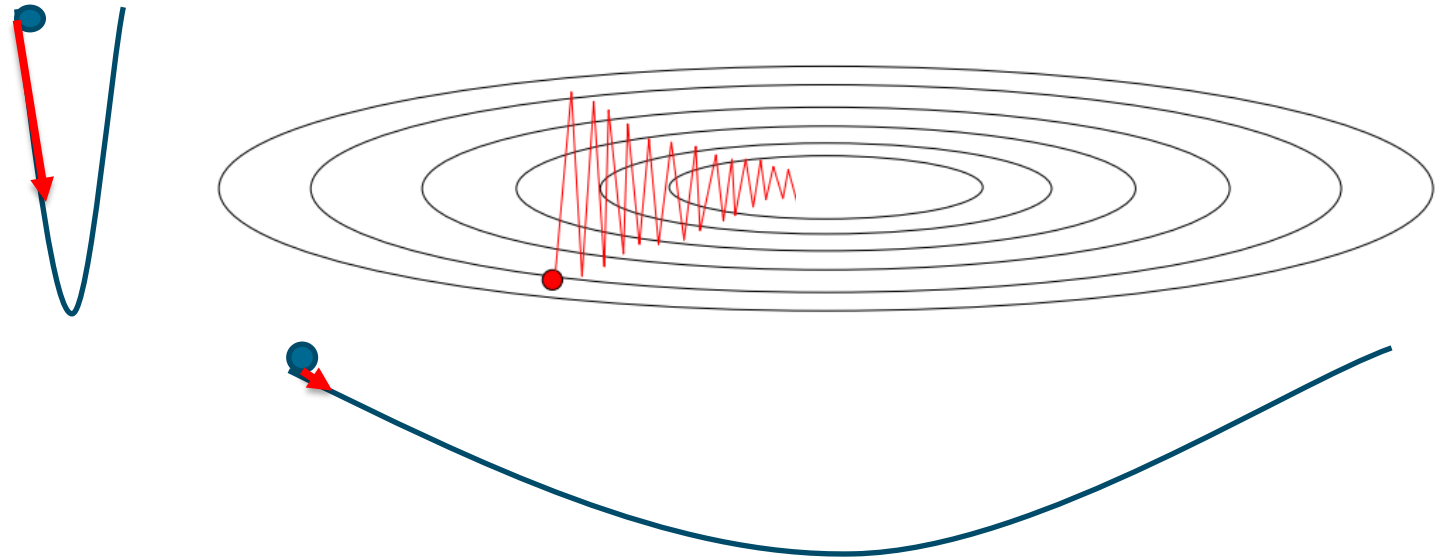
- **lr**: float >= 0. Learning rate.
- **momentum**: float >= 0. Parameter that accelerates SGD in the relevant direction and dampens oscillations.
- **decay**: float >= 0. Learning rate decay over each update.
- **nesterov**: boolean. Whether to apply Nesterov momentum.

Update rules

- Last week, we saw the basic rule to update parameters using stochastic gradient descent:

$$\boldsymbol{W}_t = \boldsymbol{W}_{t-1} - \eta \nabla_{\boldsymbol{W}} \, \mathrm{L}(\mathbf{W})\Big|_{\boldsymbol{W}=\boldsymbol{W}_{t-1}}$$

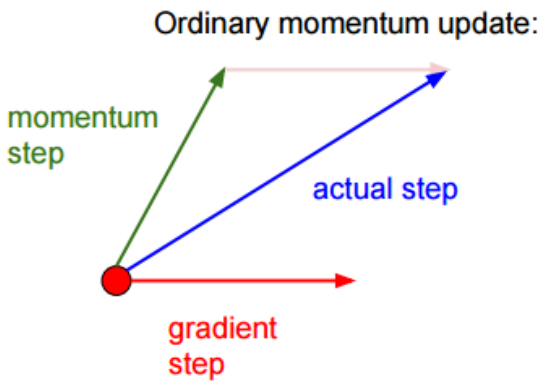
- What happens if the loss function is like this:



Momentum

Physical analogy

- A ball rolling down the loss function
  - Gradient is like the force (or acceleration) at each moment.
  - The force influences the velocity not the directly the position



SGD

```
keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)
```

Stochastic gradient descent optimizer.

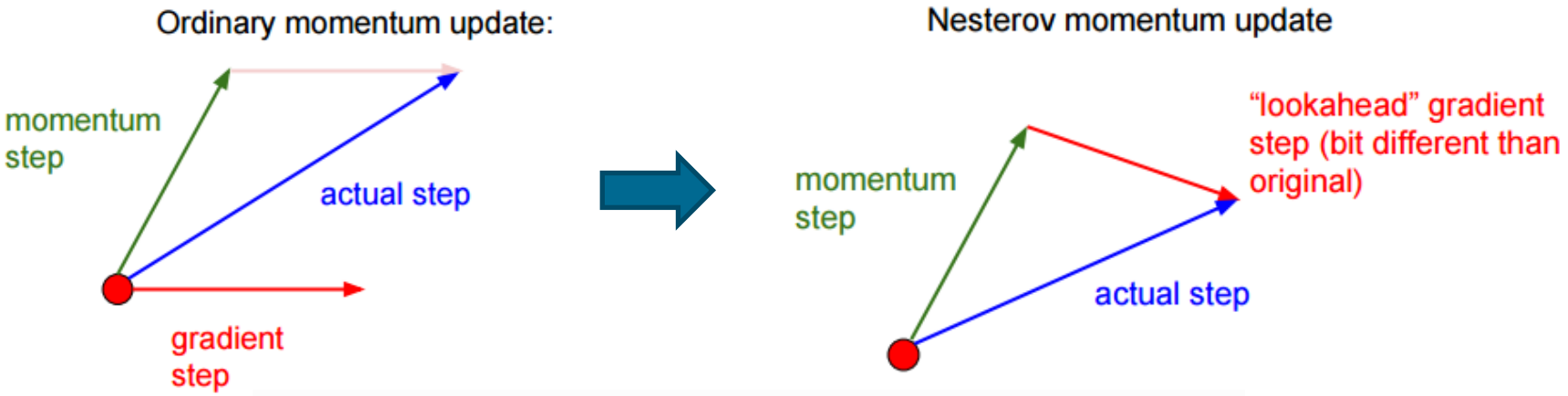
Includes support for momentum, learning rate decay, and Nesterov momentum.

Arguments

- **lr**: float >= 0. Learning rate.
- **momentum**: float >= 0. Parameter that accelerates SGD in the relevant direction and dampens oscillations.
- **decay**: float >= 0. Learning rate decay over each update.
- **nesterov**: boolean. Whether to apply Nesterov momentum.

- Does not let it fluctuate too much resulting in faster convergence.

Nesterov Momentum



SGD

```
keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)
```

Stochastic gradient descent optimizer.

Includes support for momentum, learning rate decay, and Nesterov momentum.

Arguments

- **lr**: float >= 0. Learning rate.
- **momentum**: float >= 0. Parameter that accelerates SGD in the relevant direction and dampens oscillations.
- **decay**: float >= 0. Learning rate decay over each update.
- **nesterov**: boolean. Whether to apply Nesterov momentum.

RMSProp

avoid “killing” the learning rate

```
# RMSprop
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += -learning_rate * dx / (np.sqrt(cache) + eps)
```

“personalize” the update for each parameter

RMSprop [source]

```
keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=None, decay=0.0)
```

RMSProp optimizer.

It is recommended to leave the parameters of this optimizer at their default values (except the learning rate, which can be freely tuned).

This optimizer is usually a good choice for recurrent neural networks.

Arguments

- **lr**: float >= 0. Learning rate.
- **rho**: float >= 0.
- **epsilon**: float >= 0. Fuzz factor. If `None`, defaults to `K.epsilon()`.
- **decay**: float >= 0. Learning rate decay over each update.

ADAM

keeps track of your “velocity” (like momentum)

“personalize” update per parameter (like RMSProp)

```
# ADAM
m = beta1*m + (1-beta1)*dx
v = beta2*v + (1-beta2)*(dx**2)
x += -learning_rate * m / (np.sqrt(v) + eps)
```

Adam [source]

```
keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0, amsgrad=False)
```

Adam optimizer.

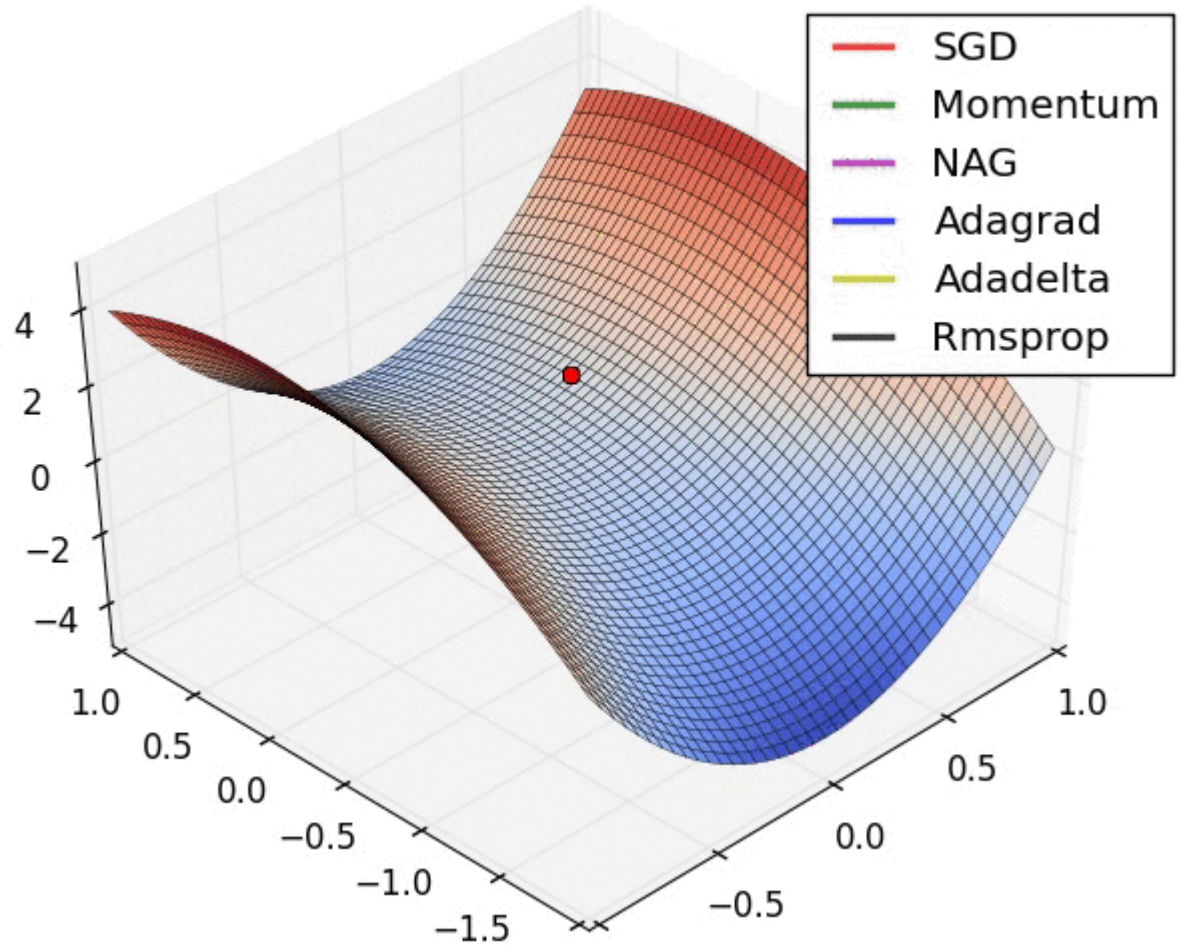
Default parameters follow those provided in the original paper.

Arguments

- **lr**: float >= 0. Learning rate.
- **beta_1**: float, 0 < beta < 1. Generally close to 1.
- **beta_2**: float, 0 < beta < 1. Generally close to 1.
- **epsilon**: float >= 0. Fuzz factor. If `None`, defaults to `K.epsilon()`.
- **decay**: float >= 0. Learning rate decay over each update.
- **amsgrad**: boolean. Whether to apply the AMSGrad variant of this algorithm from the paper "On the Convergence of Adam and Beyond".

K

Update rules

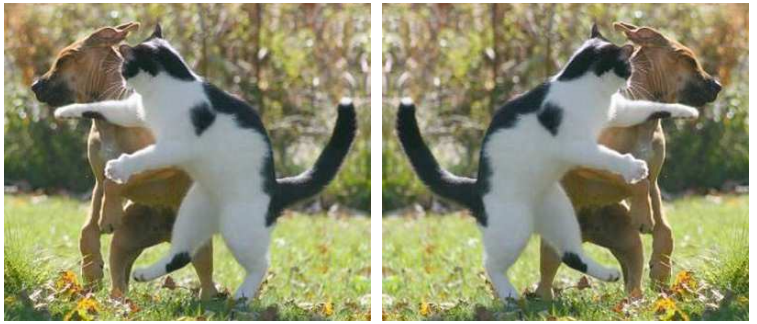


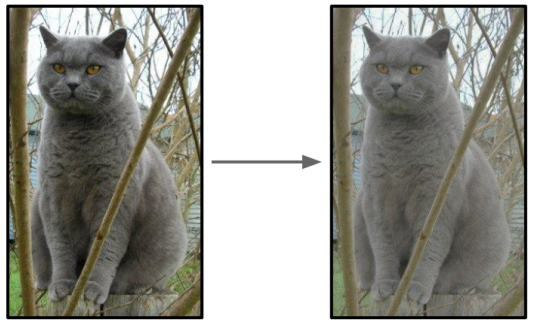
ConvNets in practice

Data augmentation

If your dataset is small:

- Mirror images
- Distorted/blurred
- Rotations
- Color changes
- **Be creative!**



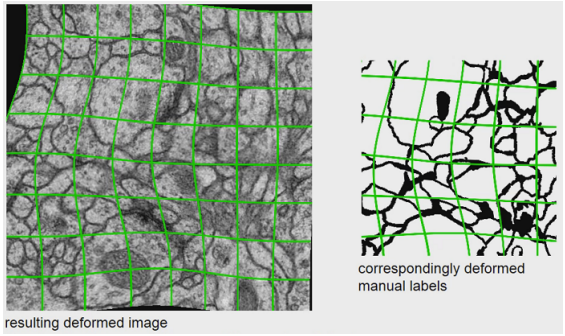


Augmentor

Augmentor is an image augmentation library in Python for machine learning. It aims to be a platform and framework independent library, which is more convenient, allows for finer grained control over augmentation, and implements as many augmentation procedures as possible. It employs a stochastic approach using building blocks that allow for operations to be pieced together in a pipeline.

Augmentor v0.2.0 | docs passing | build error | license MIT | repo status WIP | python 2.7, 3.3-3.6

https://github.com/mdbloice/Augmentor



Convolutional layer

Conv2D [source]

```
keras.layers.Conv2D(filters, kernel_size, strides=(1, 1), padding='valid', data_format=None, dilati
```

2D convolution layer (e.g. spatial convolution over images).

This layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs. If `use_bias` is True, a bias vector is created and added to the outputs. Finally, if `activation` is not `None`, it is applied to the outputs as well.

When using this layer as the first layer in a model, provide the keyword argument `input_shape` (tuple of integers, does not include the sample axis), e.g. `input_shape=(128, 128, 3)` for 128x128 RGB pictures in `data_format="channels_last"`.

Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).
- **kernel_size**: An integer or tuple/list of 2 integers, specifying the width and height of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
- **strides**: An integer or tuple/list of 2 integers, specifying the strides of the convolution along the width and height. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
- **padding**: one of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".
- **dilation_rate**: an integer or tuple/list of 2 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any `dilation_rate` value != 1 is incompatible with specifying any stride value != 1.
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the kernel matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).

Max pooling layer

MaxPooling2D [source]

```
keras.layers.MaxPooling2D(pool_size=(2, 2), strides=None, padding='valid', data_format=None)
```

Max pooling operation for spatial data.

Arguments

- **pool_size**: integer or tuple of 2 integers, factors by which to downscale (vertical, horizontal). (2, 2) will halve the input in both spatial dimension. If only one integer is specified, the same window length will be used for both dimensions.
- **strides**: Integer, tuple of 2 integers, or None. Strides values. If None, it will default to `pool_size`.
- **padding**: One of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

- If `data_format='channels_last'`: 4D tensor with shape: `(batch_size, rows, cols, channels)`
- If `data_format='channels_first'`: 4D tensor with shape: `(batch_size, channels, rows, cols)`

Output shape

- If `data_format='channels_last'`: 4D tensor with shape: `(batch_size, pooled_rows, pooled_cols, channels)`
- If `data_format='channels_first'`: 4D tensor with shape: `(batch_size, channels, pooled_rows, pooled_cols)`

Dropout layer

GaussianDropout [source]

```
keras.layers.GaussianDropout(rate)
```

Apply multiplicative 1-centered Gaussian noise.

As it is a regularization layer, it is only active at training time.

Arguments

- **rate**: float, drop probability (as with `Dropout`). The multiplicative noise will have standard deviation `sqrt(rate / (1 - rate))`.

Input shape

Arbitrary. Use the keyword argument `input_shape` (tuple of integers, does not include the samples axis) when using this layer as the first layer in a model.

Output shape

Same shape as input.

References

- Dropout: A Simple Way to Prevent Neural Networks from Overfitting Srivastava, Hinton, et al. 2014

Batch normalization layer

BatchNormalization [source]

```
keras.layers.BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001, center=True, scale=True, be
```

Batch normalization layer (Ioffe and Szegedy, 2014).

Normalize the activations of the previous layer at each batch, i.e. applies a transformation that maintains the mean activation close to 0 and the activation standard deviation close to 1.

Arguments

- **axis**: Integer, the axis that should be normalized (typically the features axis). For instance, after a `Conv2D` layer with `data_format="channels_first"`, set `axis=1` in `BatchNormalization`.
- **momentum**: Momentum for the moving mean and the moving variance.
- **epsilon**: Small float added to variance to avoid dividing by zero.
- **center**: If True, add offset of `beta` to normalized tensor. If False, `beta` is ignored.
- **scale**: If True, multiply by `gamma`. If False, `gamma` is not used. When the next layer is linear (also e.g. `nn.relu`), this can be disabled since the scaling will be done by the next layer.
- **beta_initializer**: Initializer for the beta weight.
- **gamma_initializer**: Initializer for the gamma weight.
- **moving_mean_initializer**: Initializer for the moving mean.
- **moving_variance_initializer**: Initializer for the moving variance.
- **beta_regularizer**: Optional regularizer for the beta weight.
- **gamma_regularizer**: Optional regularizer for the gamma weight.
- **beta_constraint**: Optional constraint for the beta weight.
- **gamma_constraint**: Optional constraint for the gamma weight.

Input shape

Arbitrary. Use the keyword argument `input_shape` (tuple of integers, does not include the samples axis) when using this layer as the first layer in a model.

Output shape

Same shape as input.

References

- Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Training ConvNets in practice

You have to define:

- **Network architecture**
  - Dropout layers (optional)
- **Initialization** strategy
- **Loss** function
  - Regularization (optional)
- **Learning rate**
  - Update rule
  - Adaptive learning rate strategy
- **Training set** (often given)
  - Validation set
  - Data augmentation strategy
- **Mini-batch** size

You tune things using the **VALIDATION** set

When you are happy with the performance, you classify the test set (and submit to challenger)

Hands-on tricks

- Normalize input data: makes training easier
- Use dropout
- Use batch normalization
- Initialize with He15
- Use ReLU (or some of its variation variation)
- Use ADAM (or RMSProp) optimizer
- Use adaptive learning rate

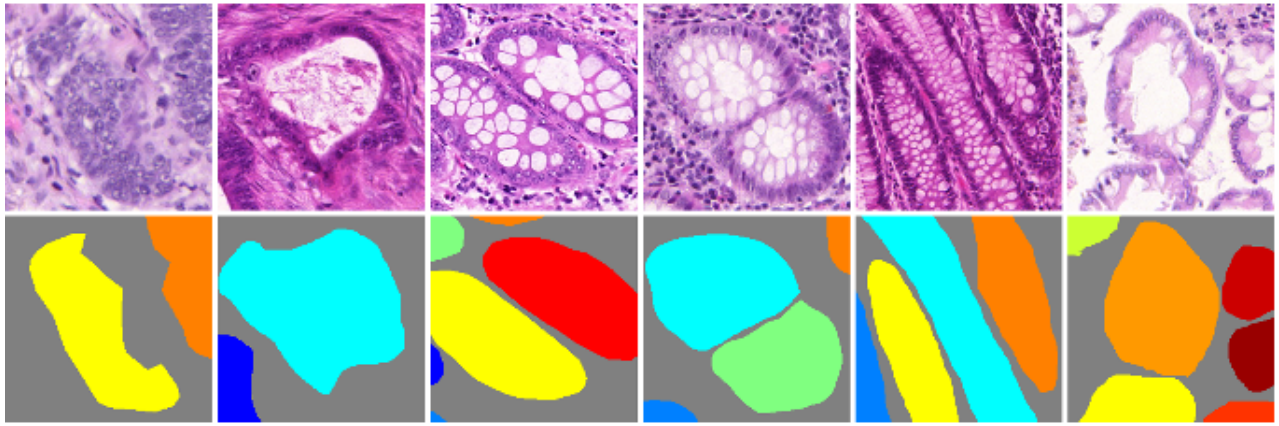
- Overfit on your training set
- Compute accuracy on Validation set before you train
- Monitor training learning curves

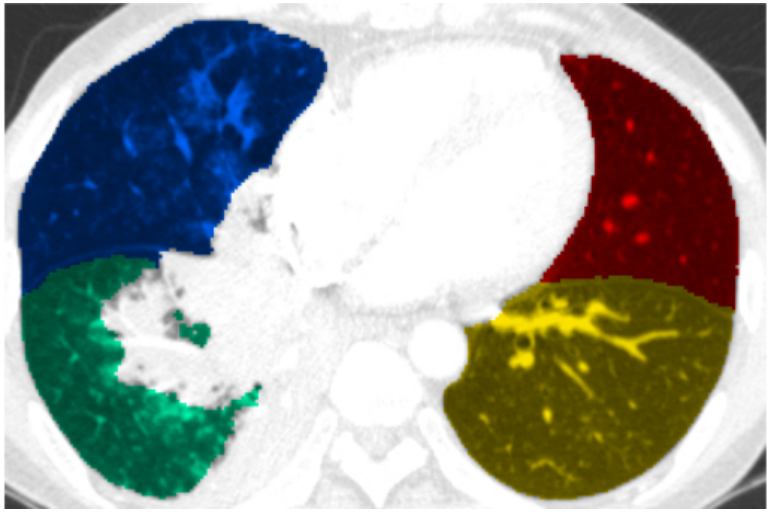
Break?

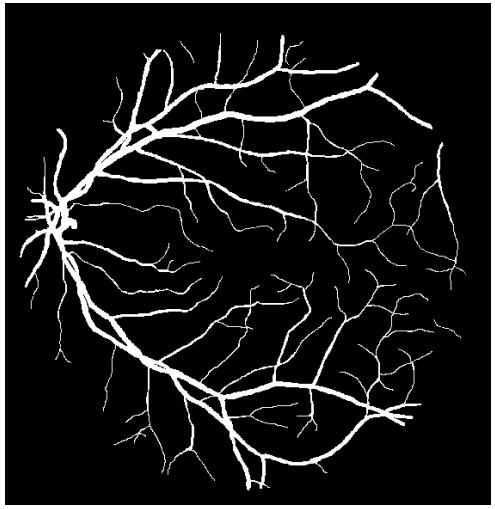
Image segmentation using Convolutional Networks

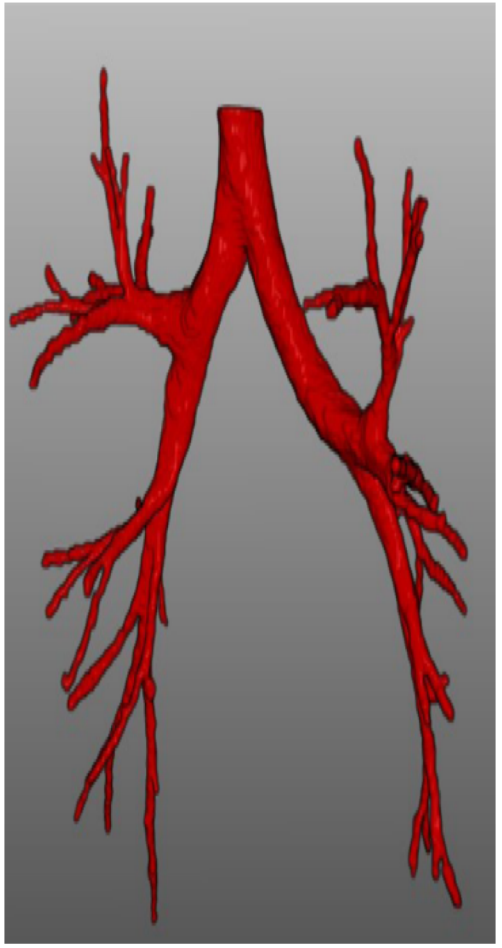
Radboudumc

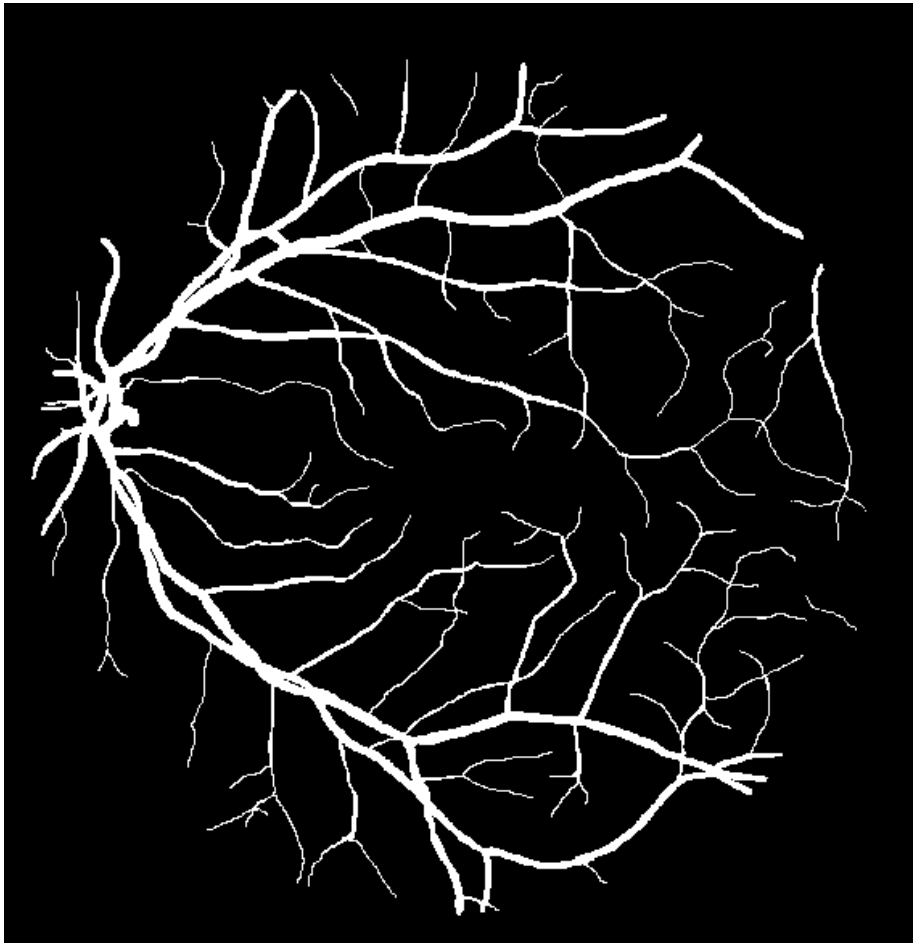
Segmentation in medical imaging







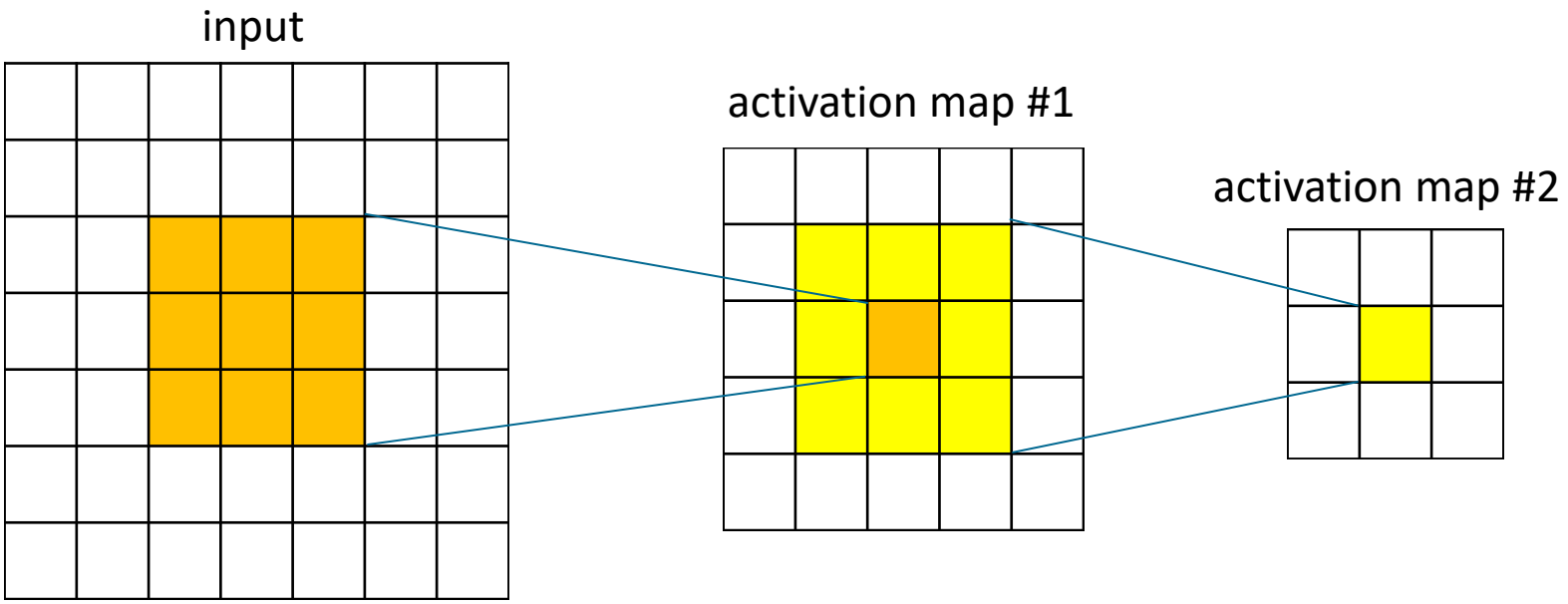




Receptive field

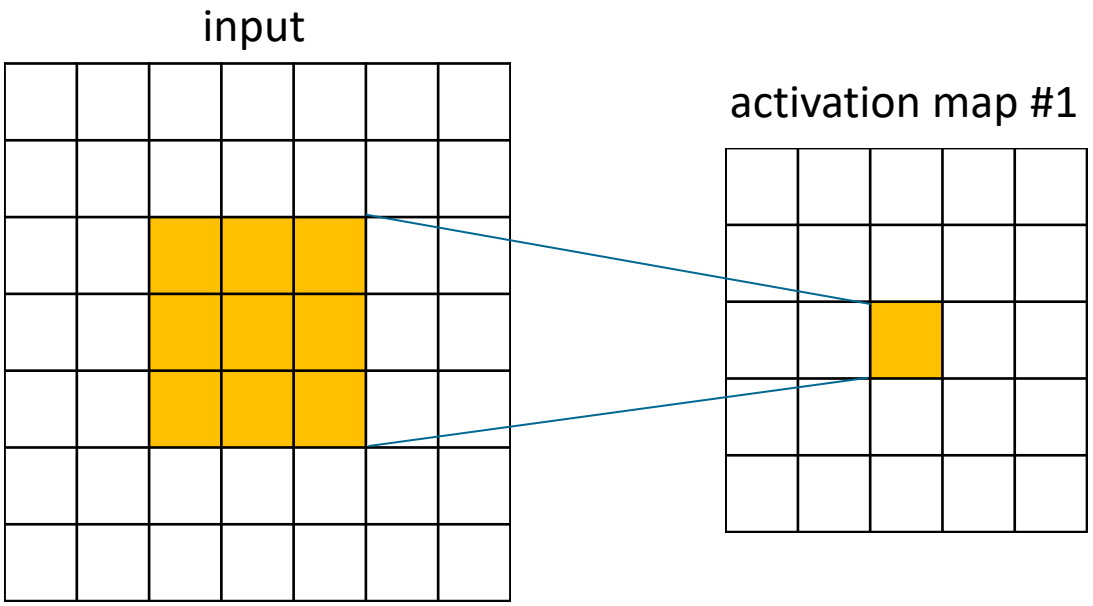
- The receptive field is the **area of the input image** that is seen by single **neurons** in any layer of the network

- Example: 2 convolutional layers with 3x3 filters



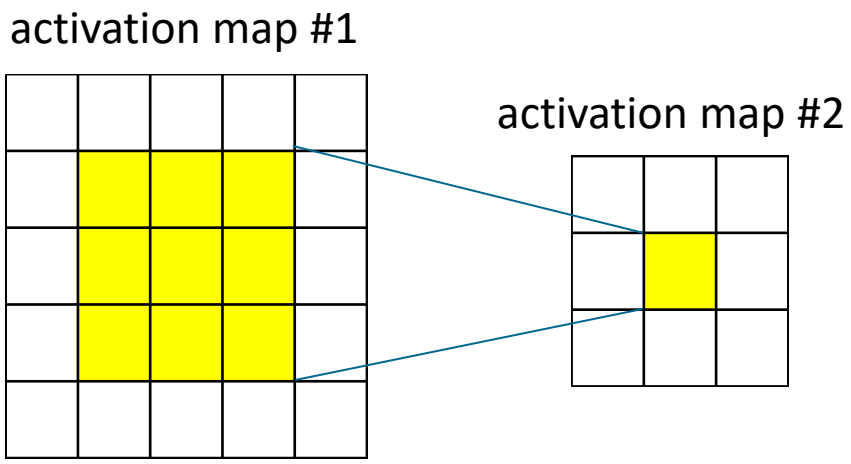
Receptive field

- Each activation map #1 “sees” an area of 3x3 in the input



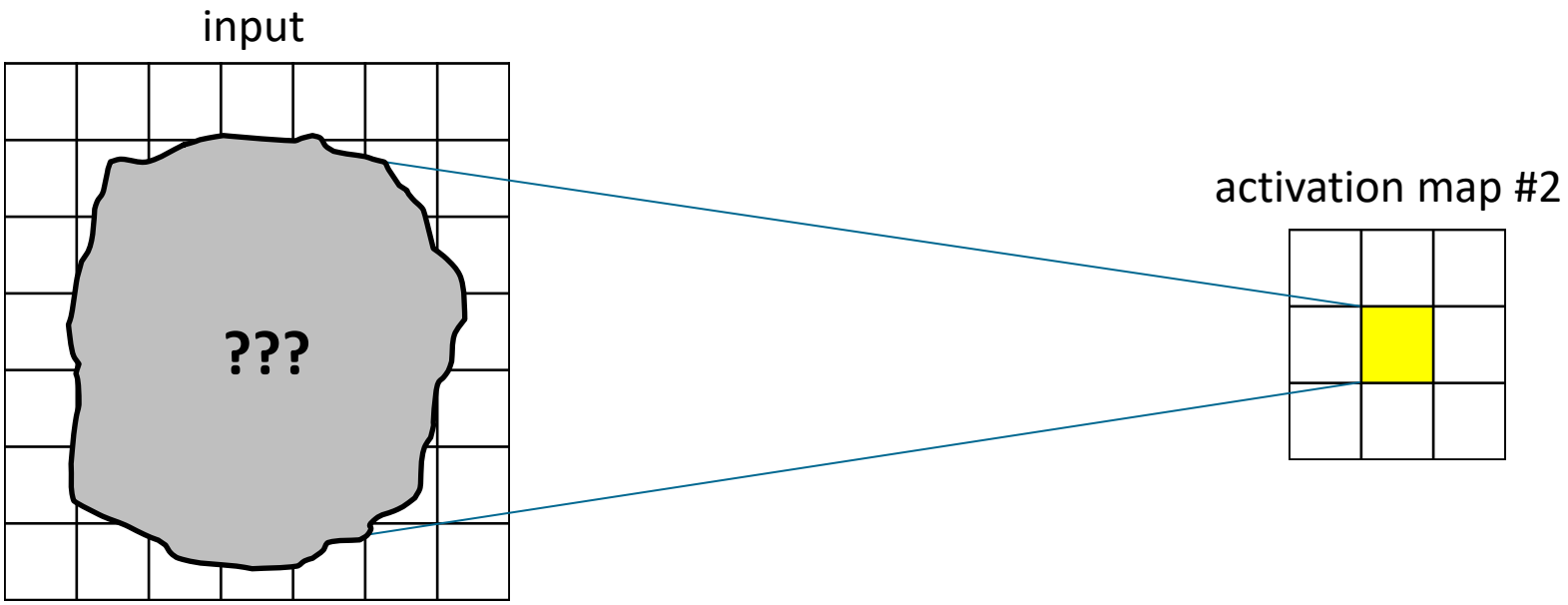
Receptive field

- Each activation map #2 “sees” an area of 3x3 in map #1



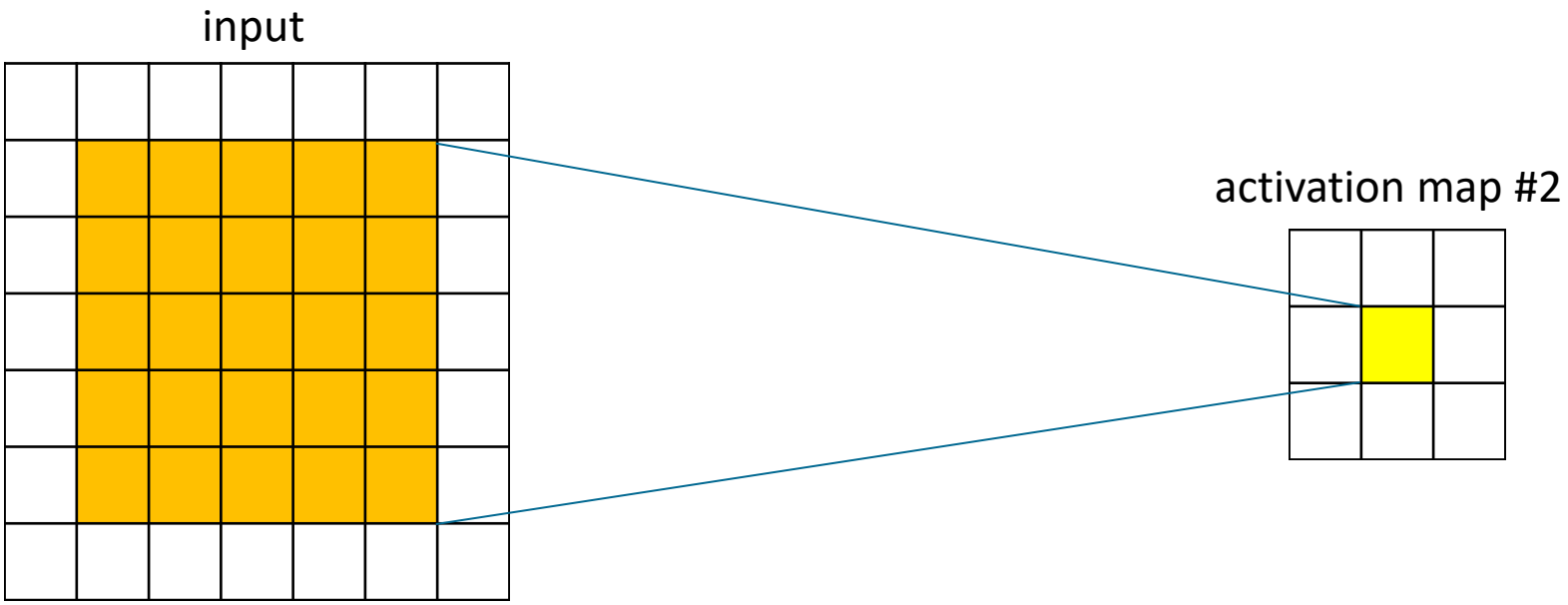
Receptive field

- How much of the input does an activation of map #2 see?



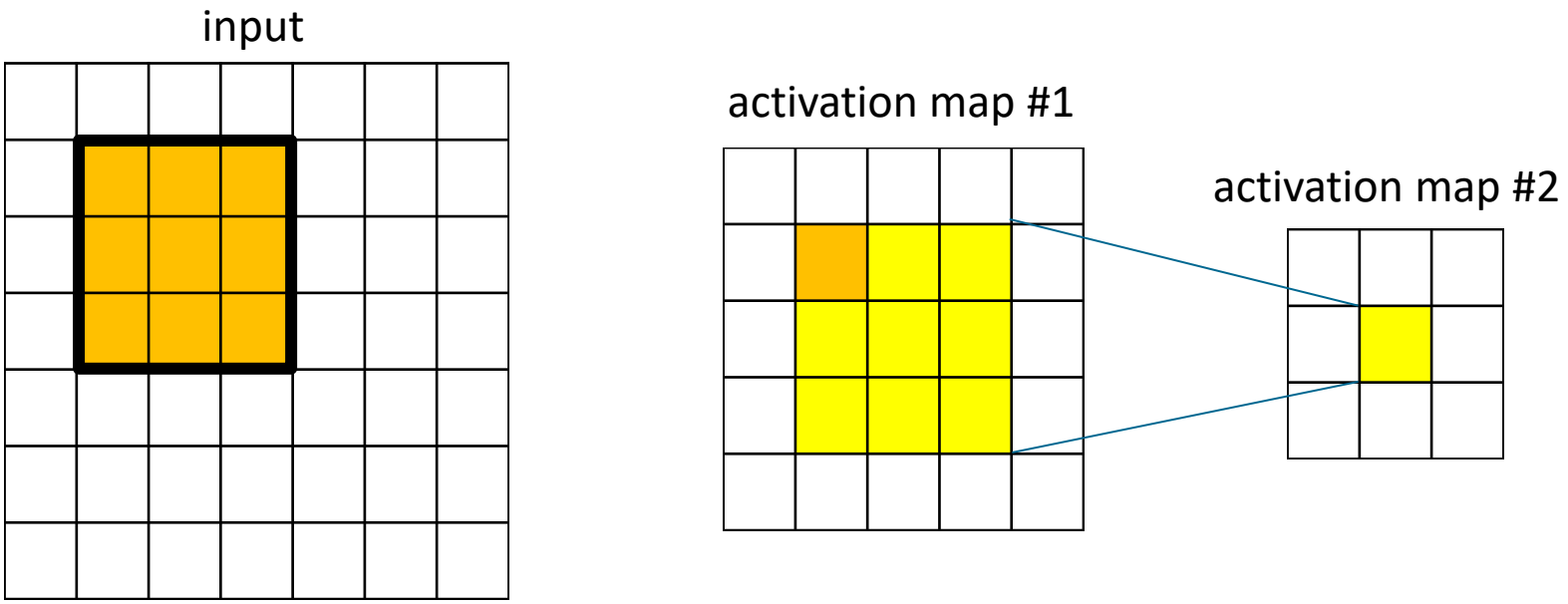
Receptive field

- How much of the input does an activation of map #2 see?
- It sees a 5x5 region! **WHY?**



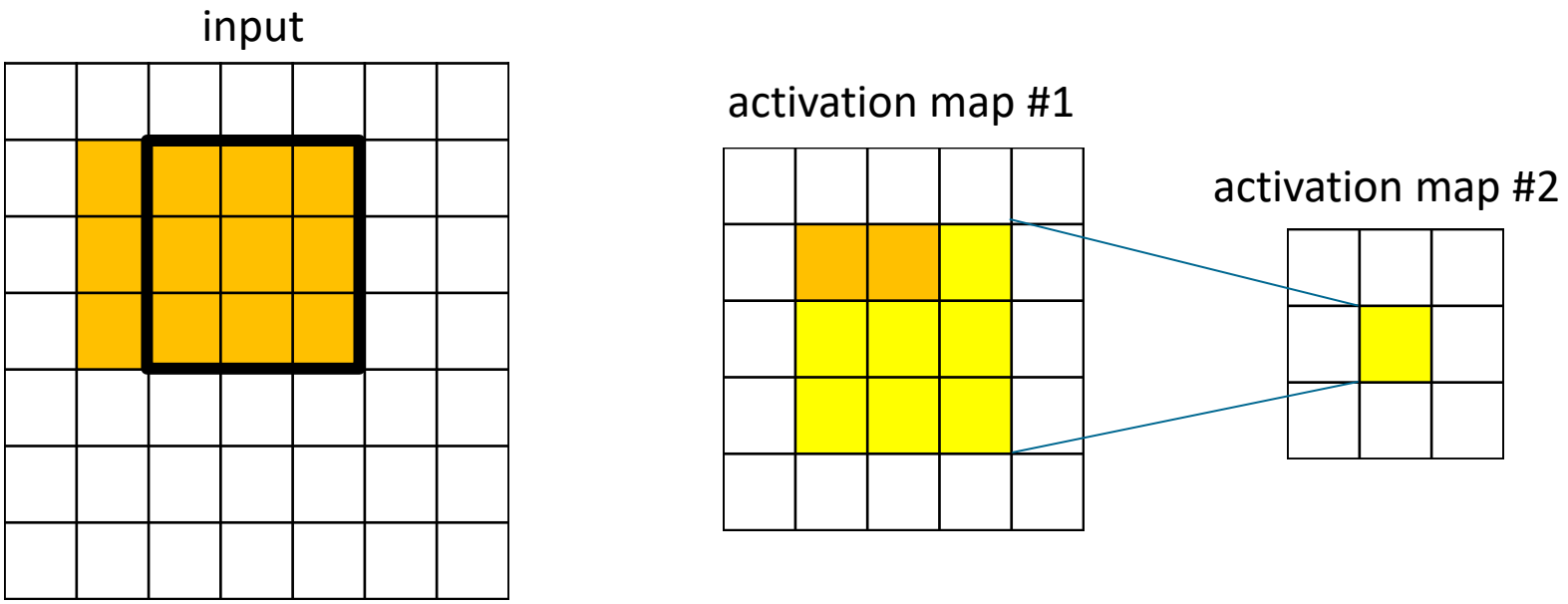
Receptive field

- How much of the input does an activation of map #2 see?
- It sees a 5x5 region! **WHY?**



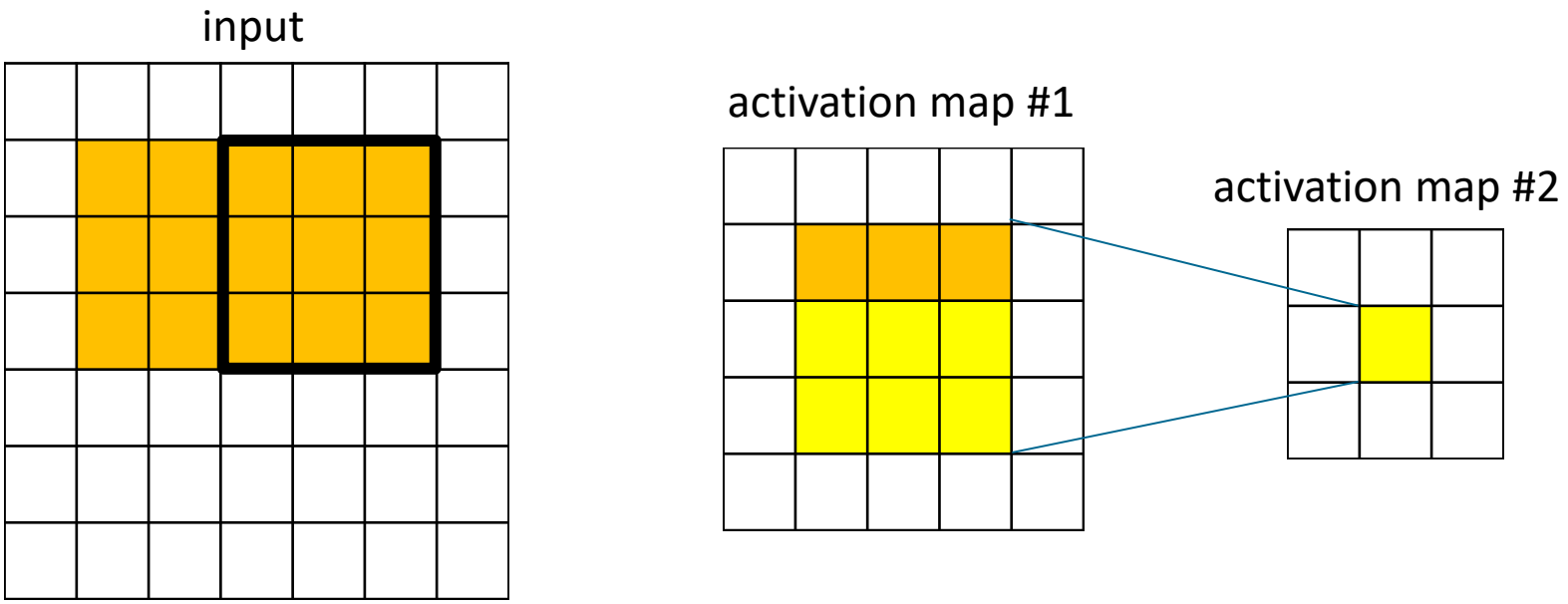
Receptive field

- How much of the input does an activation of map #2 see?
- It sees a 5x5 region! **WHY?**



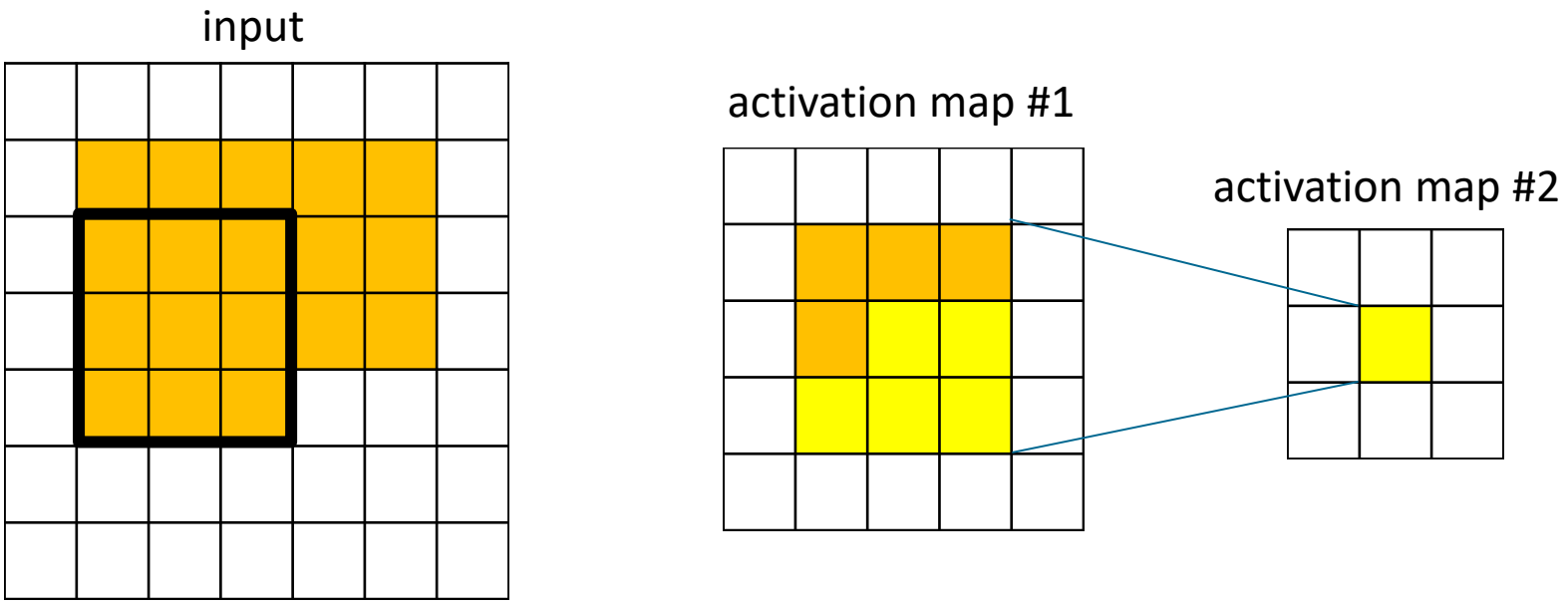
Receptive field

- How much of the input does an activation of map #2 see?
- It sees a 5x5 region! **WHY?**



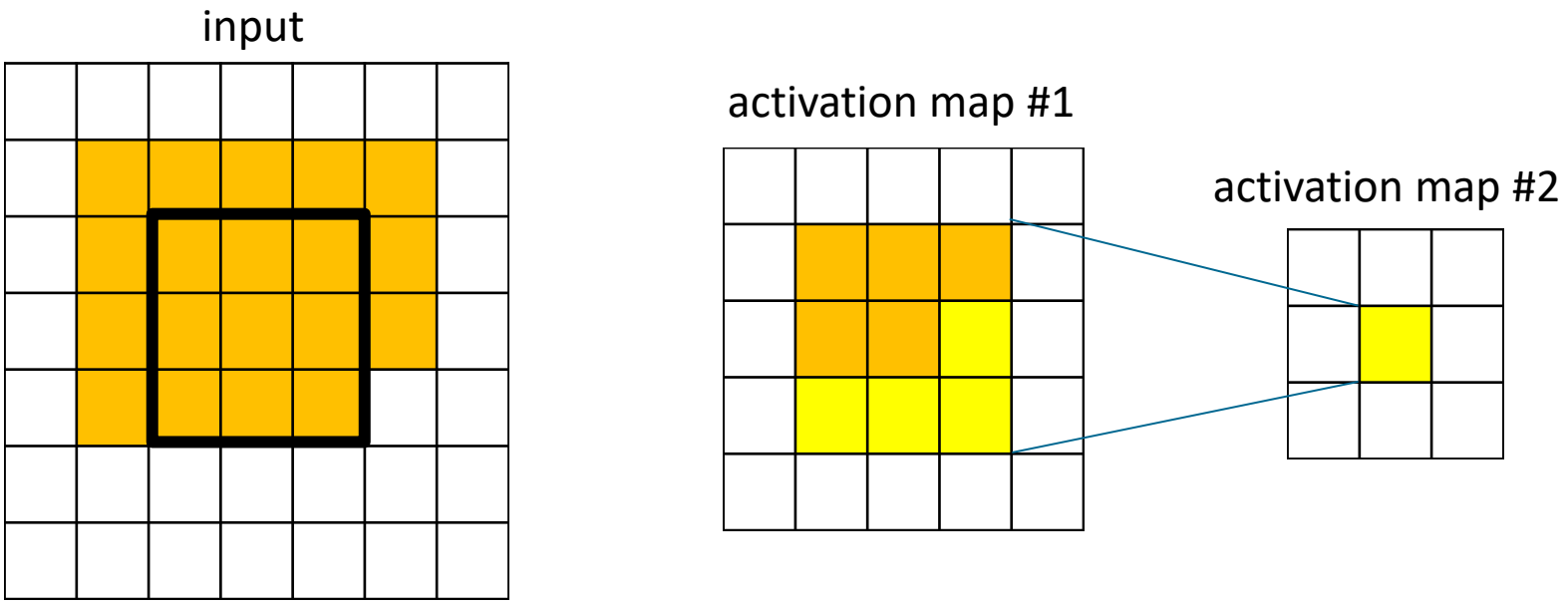
Receptive field

- How much of the input does an activation of map #2 see?
- It sees a 5x5 region! **WHY?**



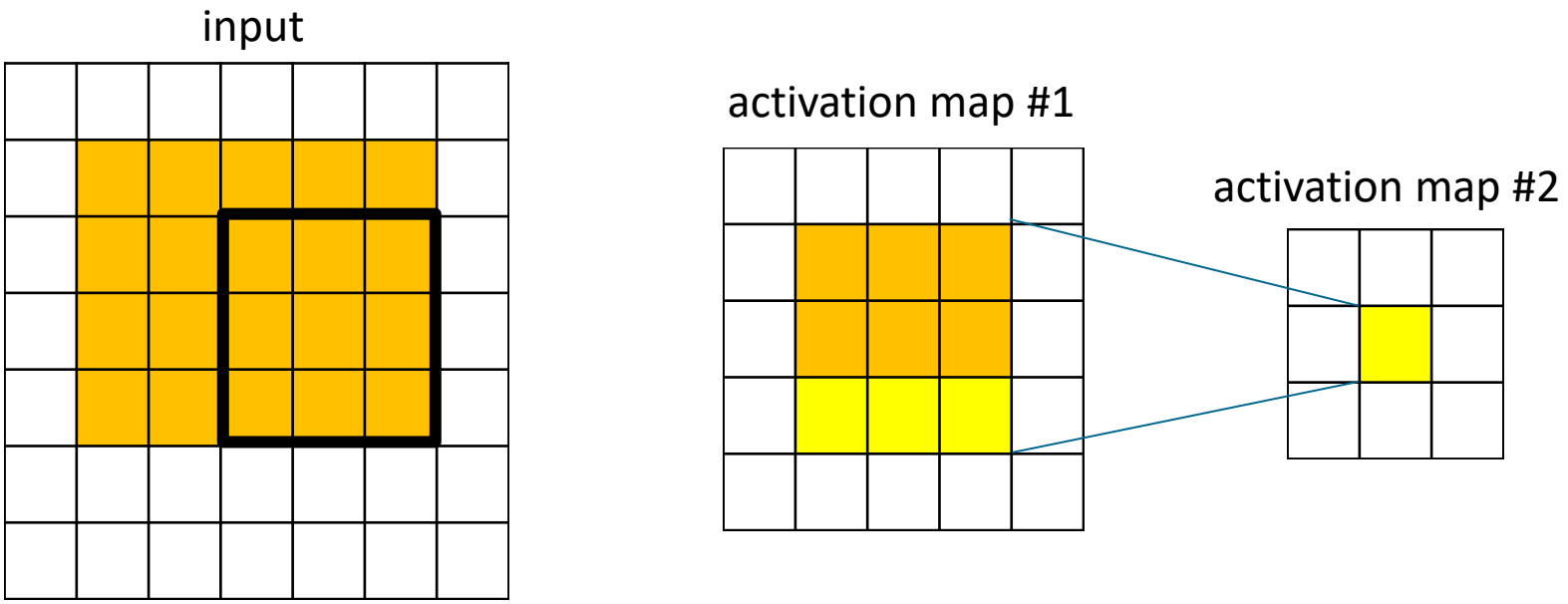
Receptive field

- How much of the input does an activation of map #2 see?
- It sees a 5x5 region! **WHY?**



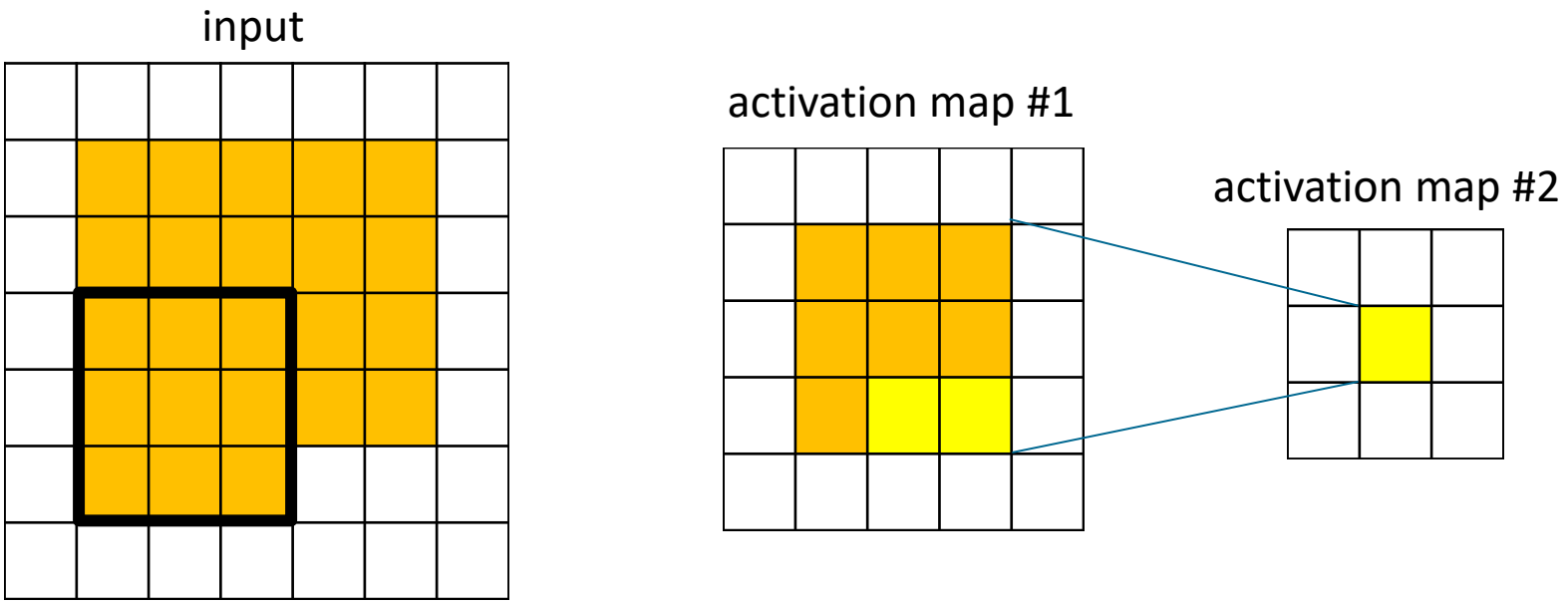
Receptive field

- How much of the input does an activation of map #2 see?
- It sees a 5x5 region! **WHY?**



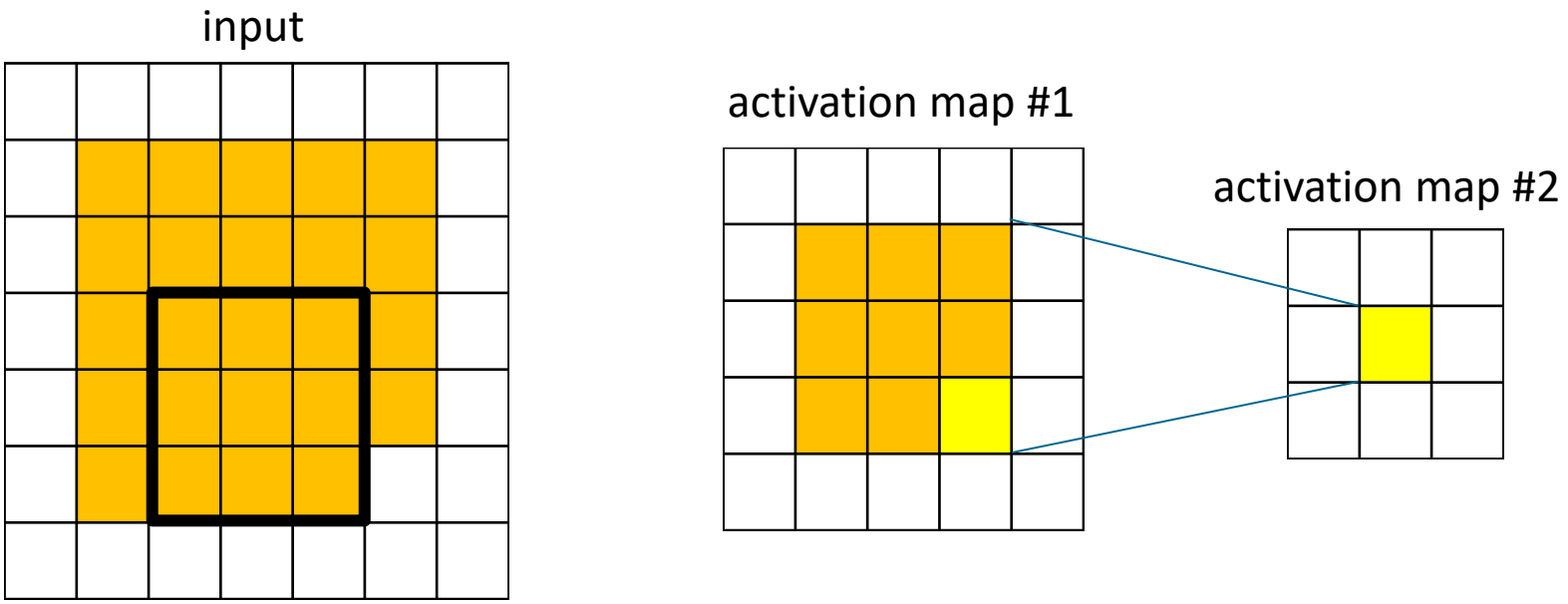
Receptive field

- How much of the input does an activation of map #2 see?
- It sees a 5x5 region! **WHY?**



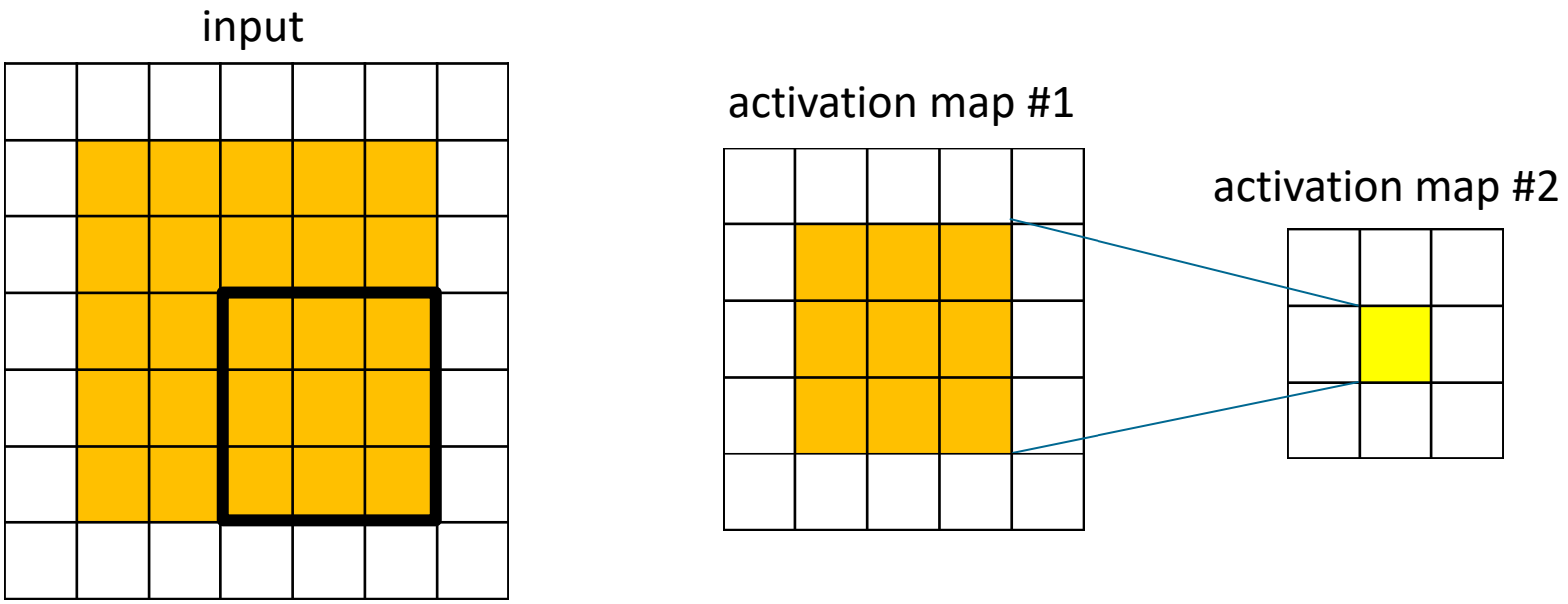
Receptive field

- How much of the input does an activation of map #2 see?
- It sees a 5x5 region! **WHY?**



Receptive field

- How much of the input does an activation of map #2 see?
- It sees a 5x5 region! **WHY?**

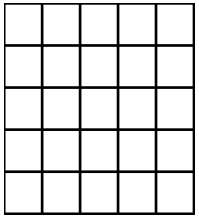


Receptive field

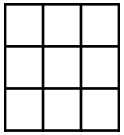
Factors that affect the receptive field:

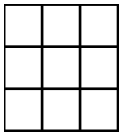
- **Number** of layers
- Filter **size**

Two layers of 3x3 filters give the same receptive field of one layer with a 5x5 filter



5x5 + 1 = 31





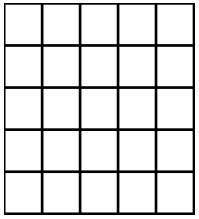
2 x (3x3 + 1) = 20

Receptive field

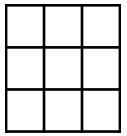
Factors that affect the receptive field:

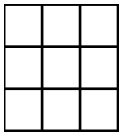
- **Number** of layers
- Filter **size**
- Presence of **max-pooling**
  - Increase receptive field
  - Decrease resolution
  - Spatial information is lost

Two layers of 3x3 filters give the same receptive field of one layer with a 5x5 filter



5x5 + 1 = 31



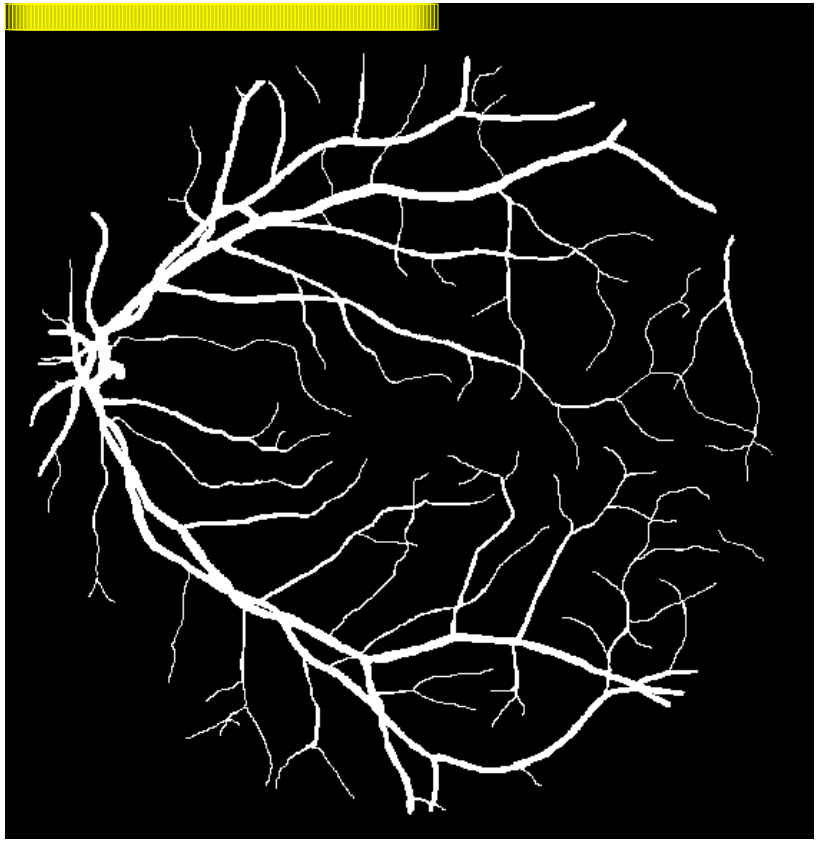


2 x (3x3 + 1) = 20

ConvNets for segmentation

How to approach this problem?

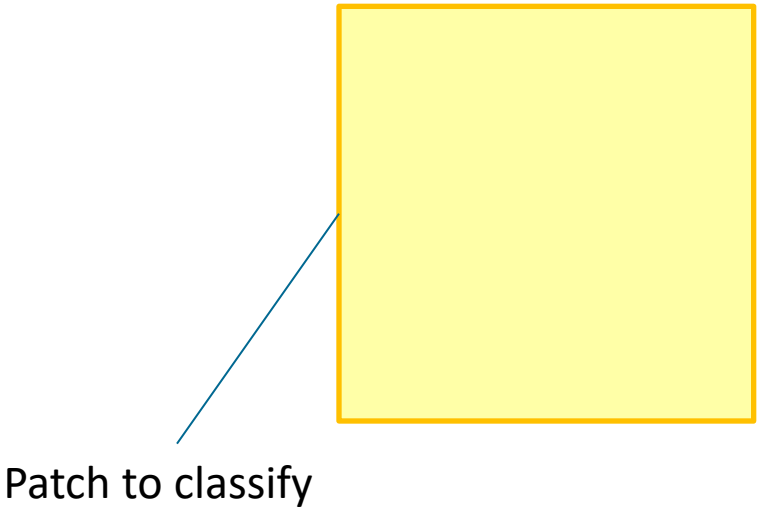
- ConvNets can classify patches and label the central pixel
- Slide all patches and classify each of them?
- Not very efficient solution
- SLOW!!!



ConvNets for segmentation

Why not efficient?

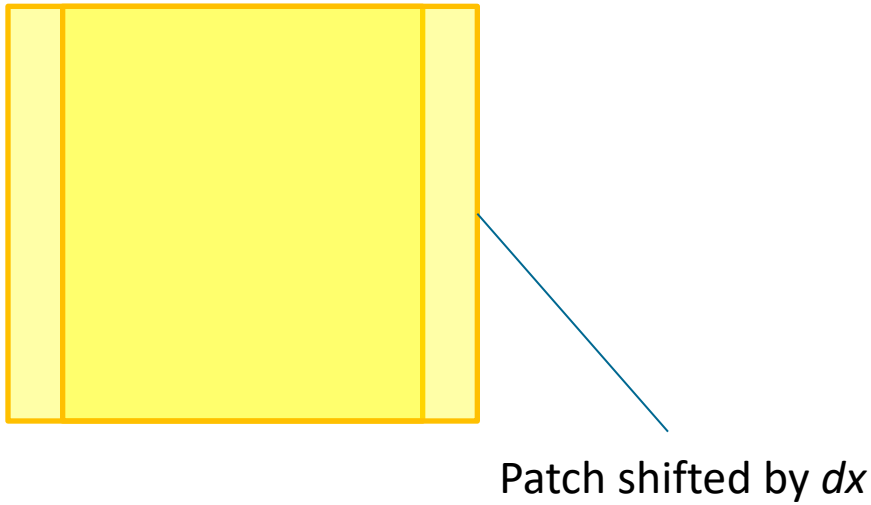
- When we classify one patch, we do convolutions
- If we shift the patch by one pixel, we repeat a lot of convolutions



ConvNets for segmentation

Why not efficient?

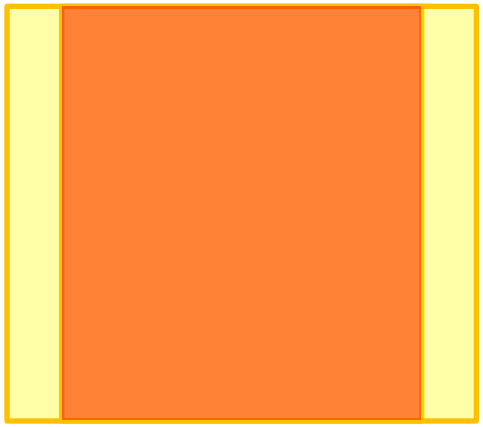
- When we classify one patch, we do convolutions
- If we shift the patch by one pixel, we repeat a lot of convolutions



ConvNets for segmentation

Why not efficient?

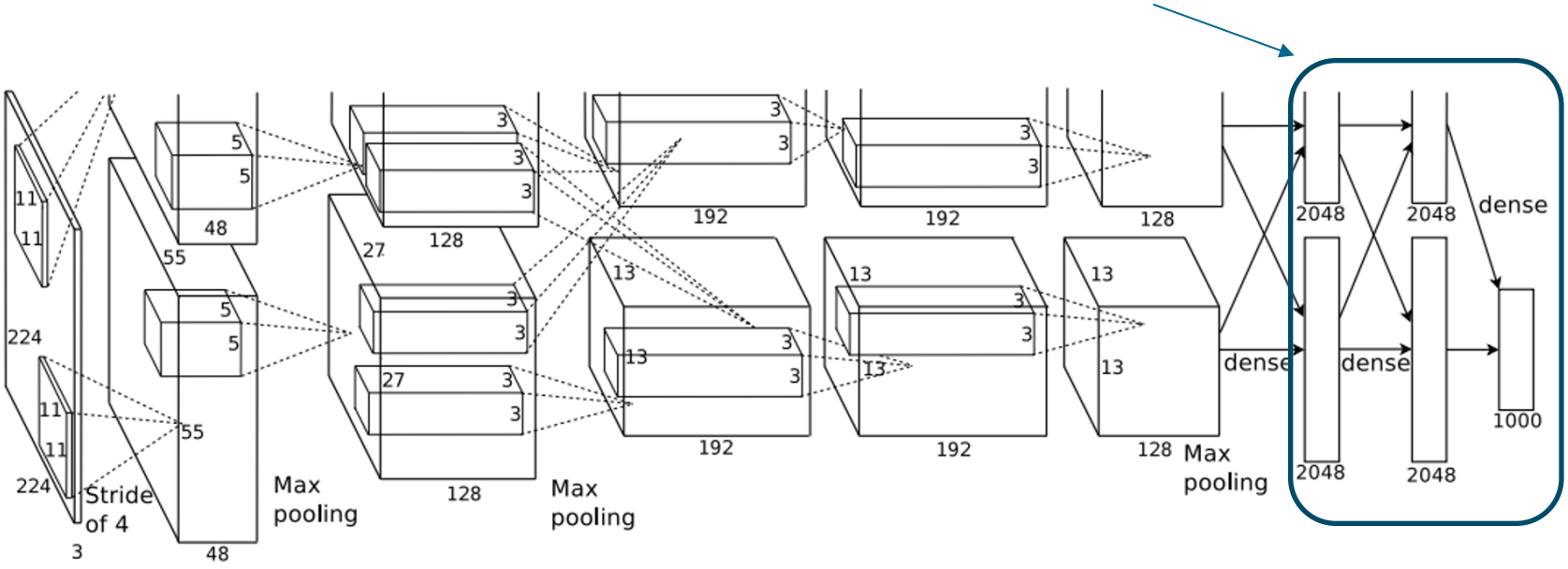
- When we classify one patch, we do convolutions
- If we shift the patch by one pixel, we repeat a lot of convolutions



Operations repeated!!!

ConvNets for segmentation

- ConvNets can be used **efficiently** for **segmentation**
- They can be used (almost) *out-of-the-box*
- We just need to “re-think” about **fully connected layers**



About fully-connected layers

- ConvNets are built on *translation invariance*
  - Convolution
  - Pooling
  - Activation functions
- A “standard” ConvNet computes a *function*
- A network with only these 3 elements computes a “*filter*”
- We can call it a *Deep filter* or a ***Fully-convolutional network***

About fully-connected layers

- ConvNets are built on *translation invariance*

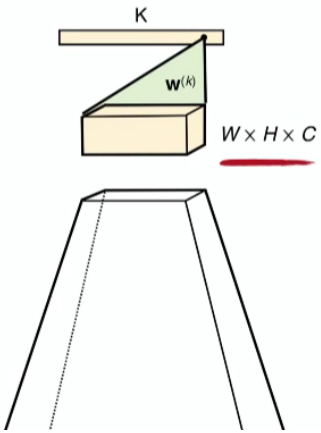
  - Convolution
  - Pooling
  - Activation functions

  These elements do not depend on the size of the input!

- A “standard” ConvNet computes a *function*
- A network with only these 3 elements computes a “*filter*”
- We can call it a *Deep filter* or a ***Fully-convolutional network***

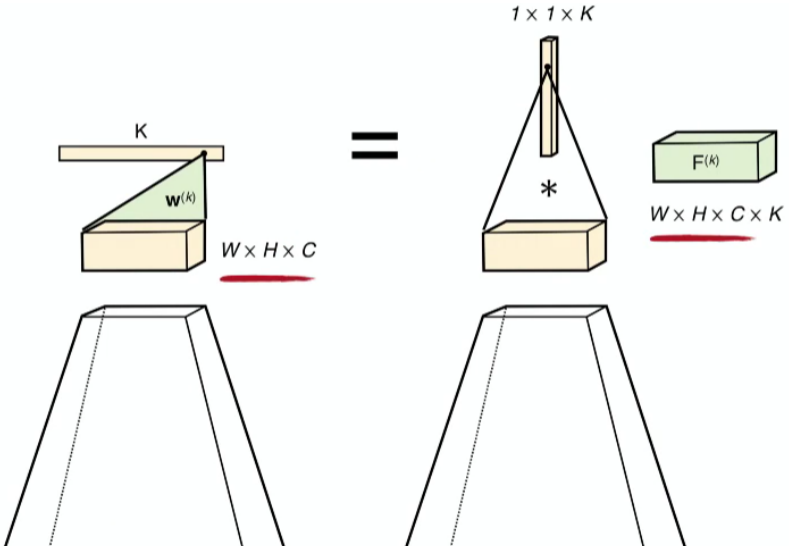
About fully-connected layers

- A FC layer has fixed dimensions and throw away spatial coordinates (it flattens everything and makes a vector)



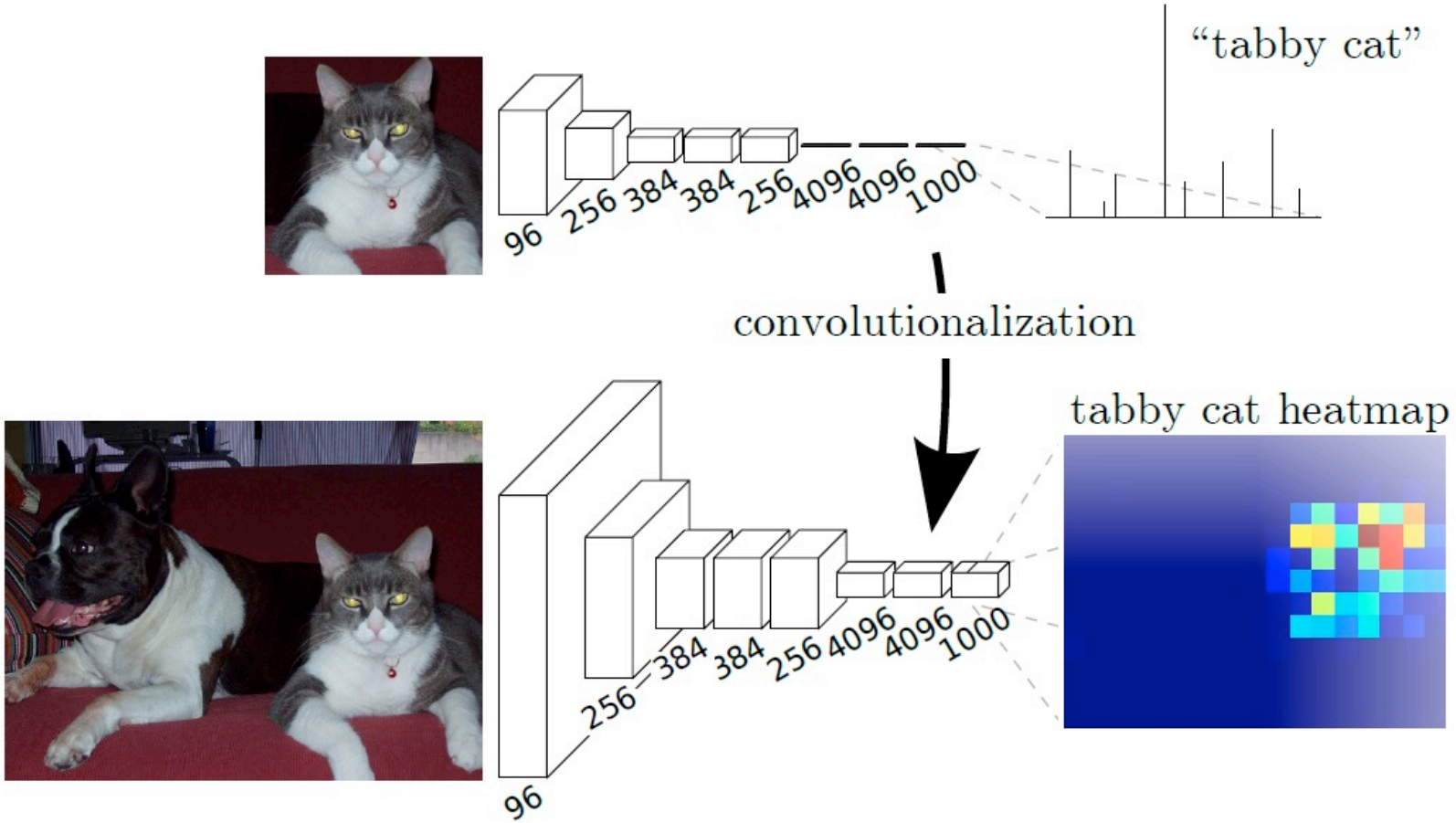
About fully-connected layers

- A FC layer has fixed dimensions and throw away spatial coordinates (it flattens everything and makes a vector)



- However, FC layers can also be viewed as **convolutions** with **kernels that cover their entire input regions**

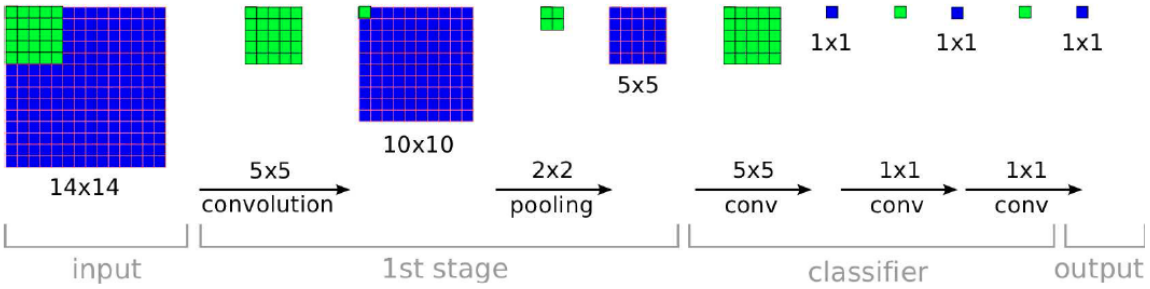
Fully Convolutional Network



Long et al., 2015

Fully convolutional network

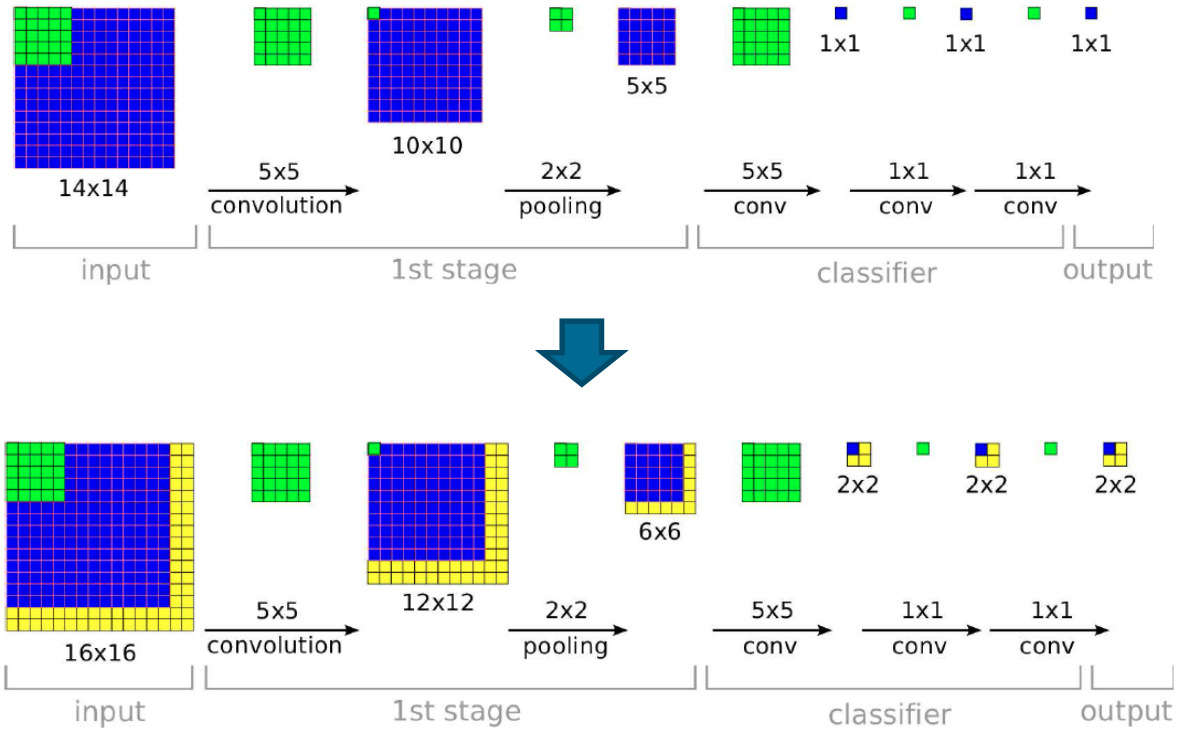
- A convolutional network becomes a big convolutional filter
- We train it with patches of size *PxP* to predict one value



Trained with 14x14 patches to predict one output value

Fully convolutional network

- A convolutional network becomes a big convolutional filter
- We train it with patches of size *PxP* to predict one value

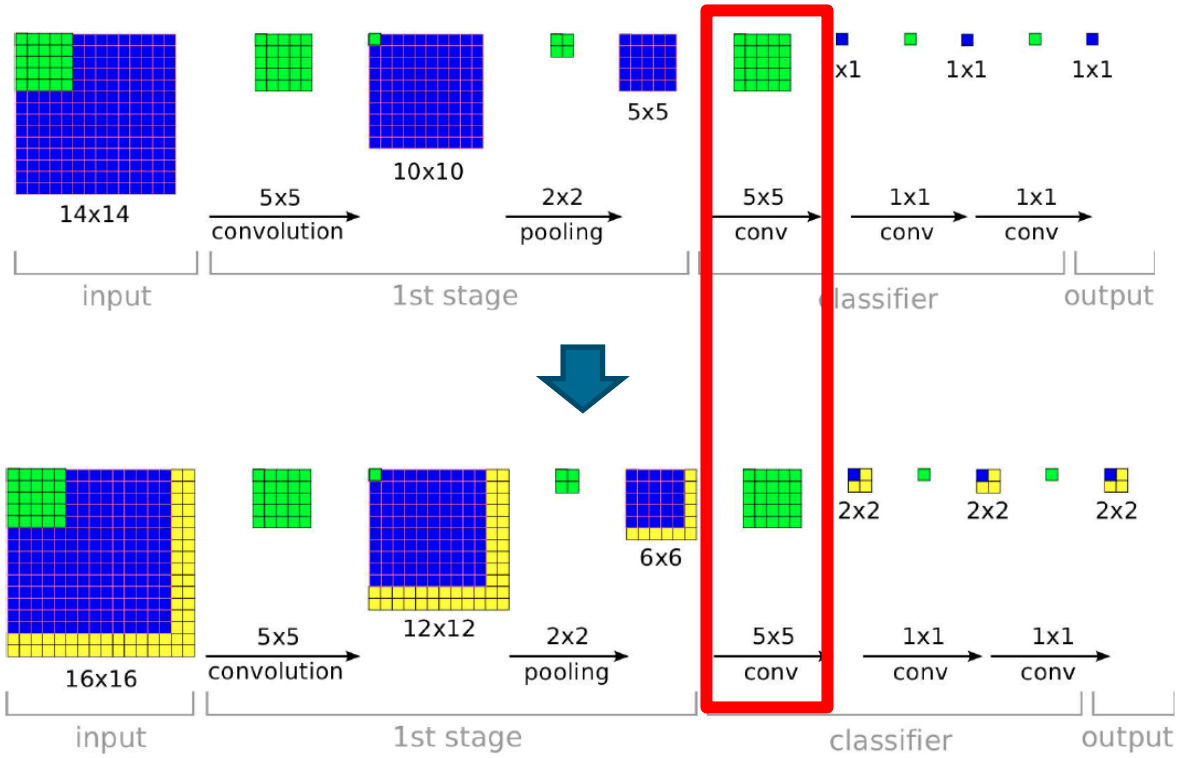


Trained with 14x14 patches to predict one output value

Applied to input **larger** than 14x14 to get a **2D output**

Fully convolutional network

- A convolutional network becomes a big convolutional filter
- We train it with patches of size *PxP* to predict one value

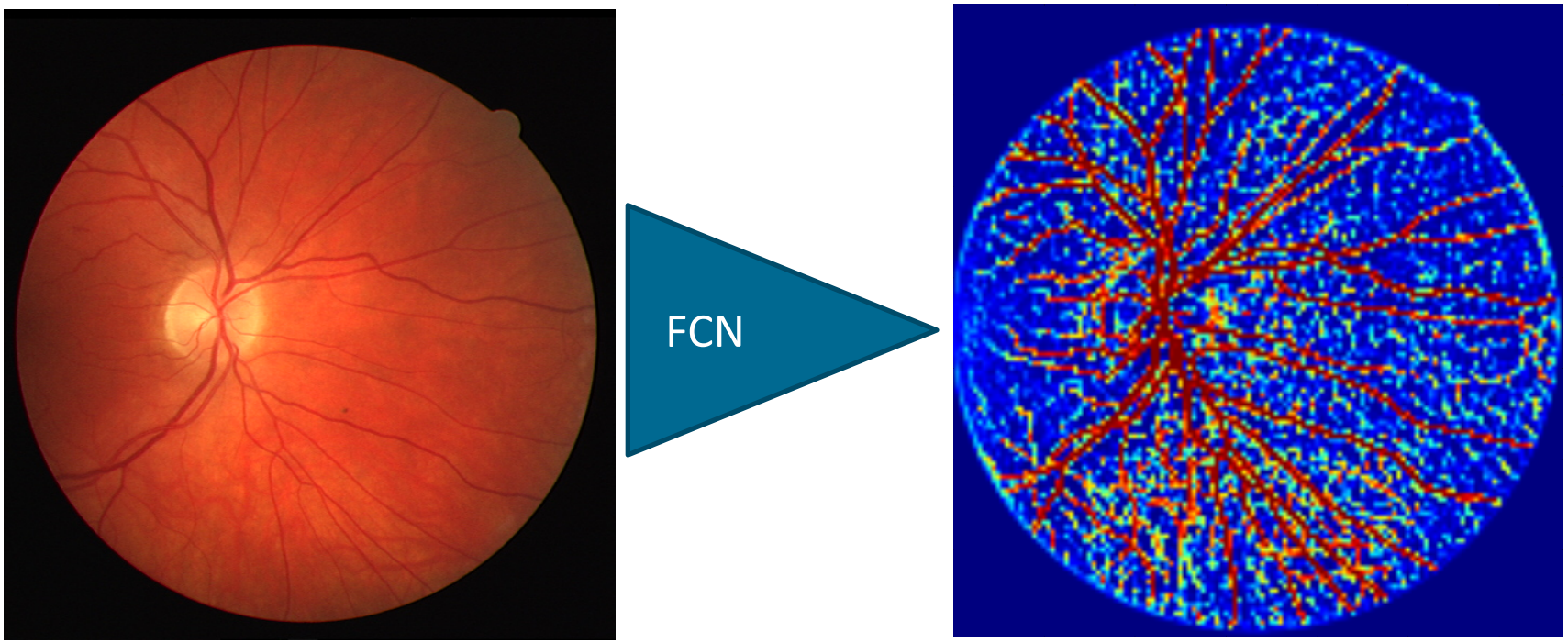


Trained with 14x14 patches to predict one output value

Applied to input **larger** than 14x14 to get a **2D output**

Fully convolutional network

- DRIVE data



Low resolution to high resolution

Given a low-res output of our network, we can think of getting its full resolution version by:

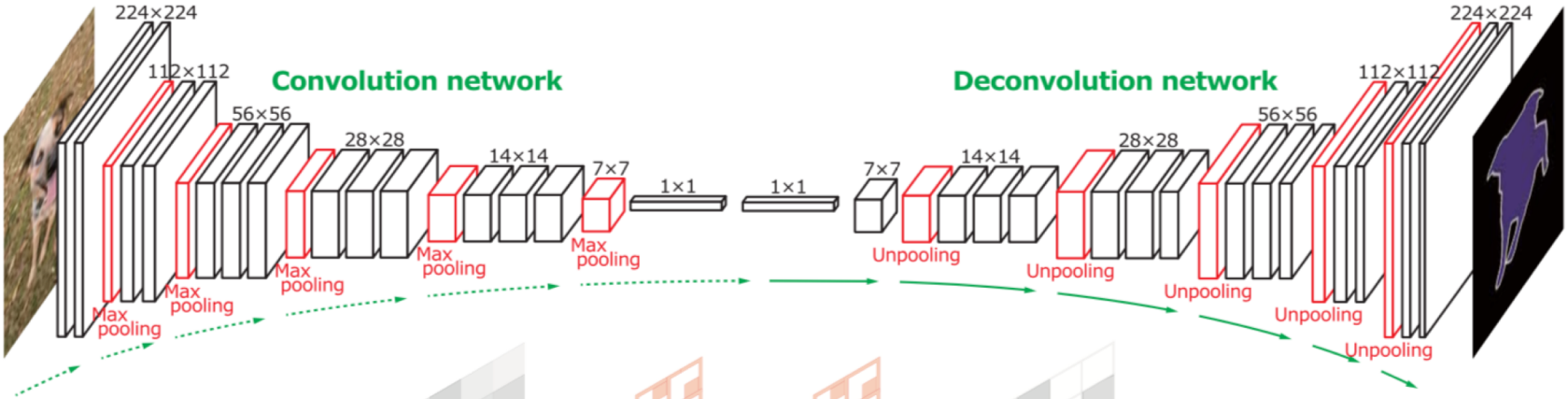
1. **Upscaling**
   - Interpolation (nearest neighbor, linear, etc…)
   - Learning **de-convolution** filters [Noh15, Fischer15]

2. **Combining** multiple low-res results: “shift and stitch”

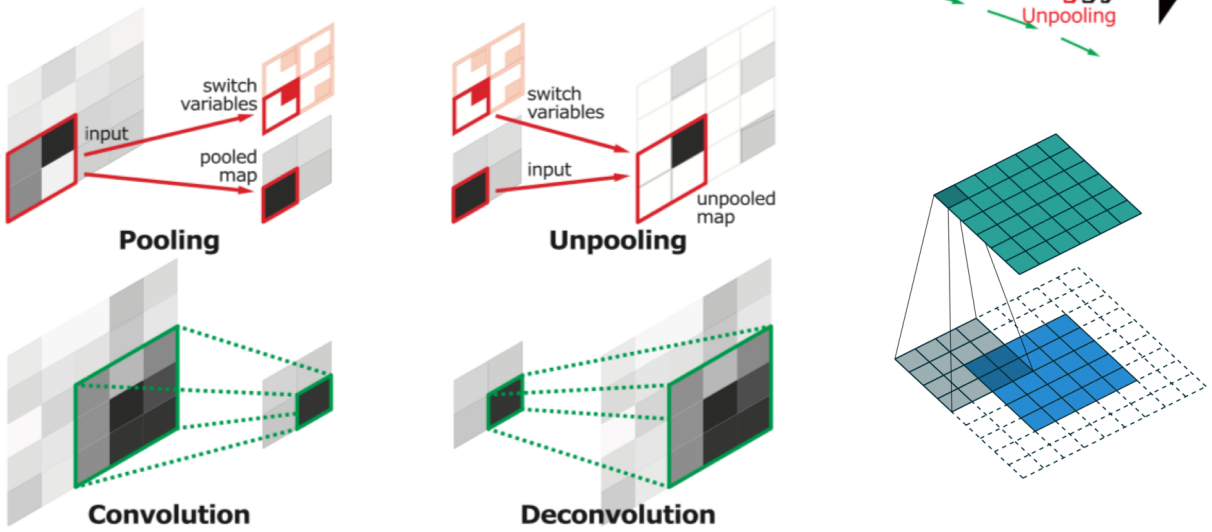
3. **Avoiding** low resolution ☺

De- / Up- / Transposed Convolution

The idea is to **learn filters** to do up-sampling



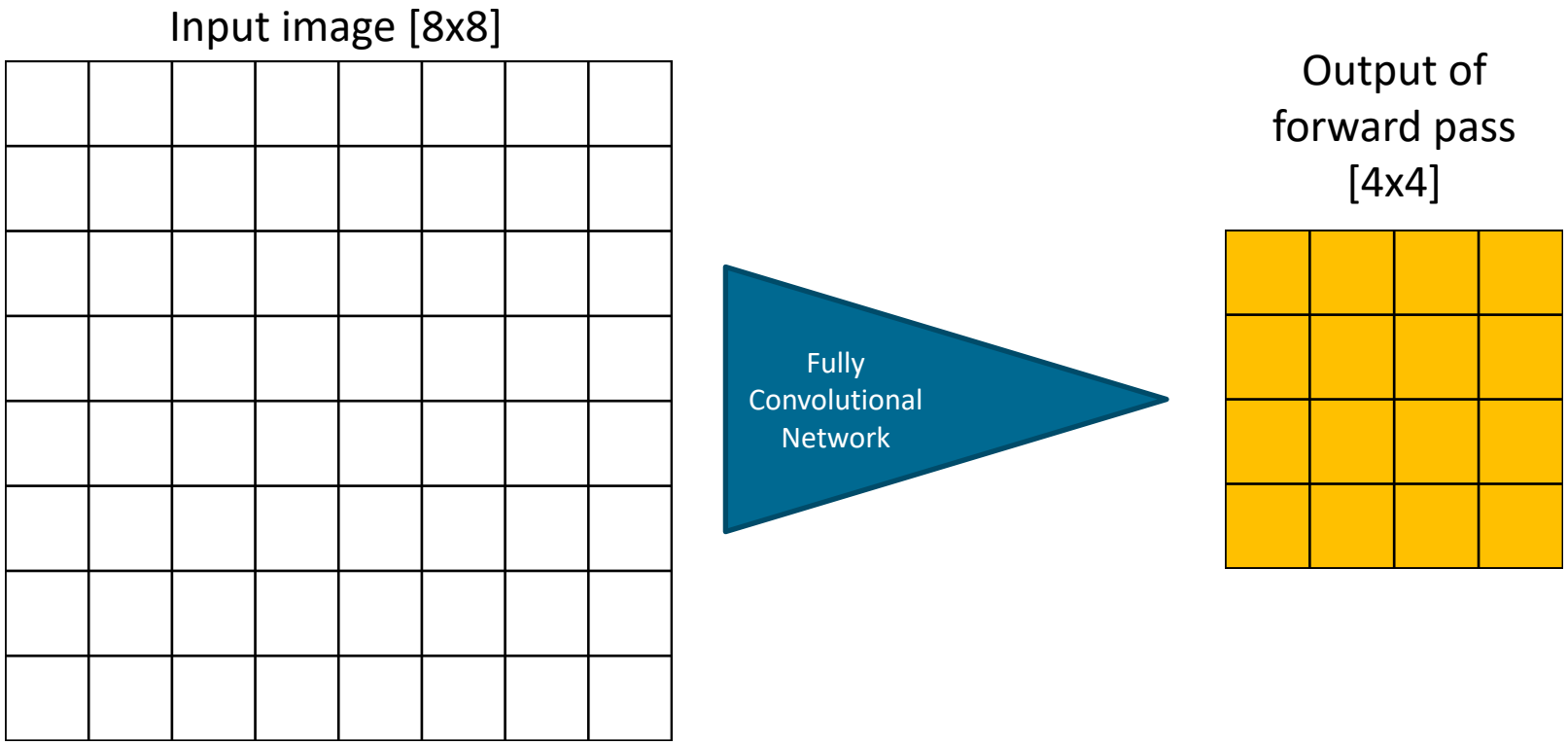
Unpooling generates a **sparse** map



H. Noh et al., "Learning Deconvolution Network for Semantic Segmentation", ICCV 2015

Fully convolutional network

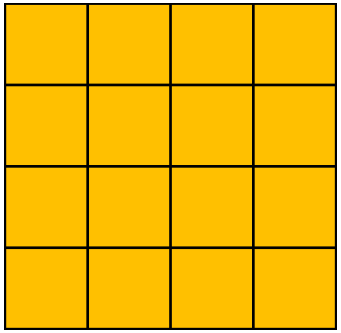
- If we have a network with one **2x2 pooling layer (stride 2)**
- A 8x8 input gives a 4x4 output:



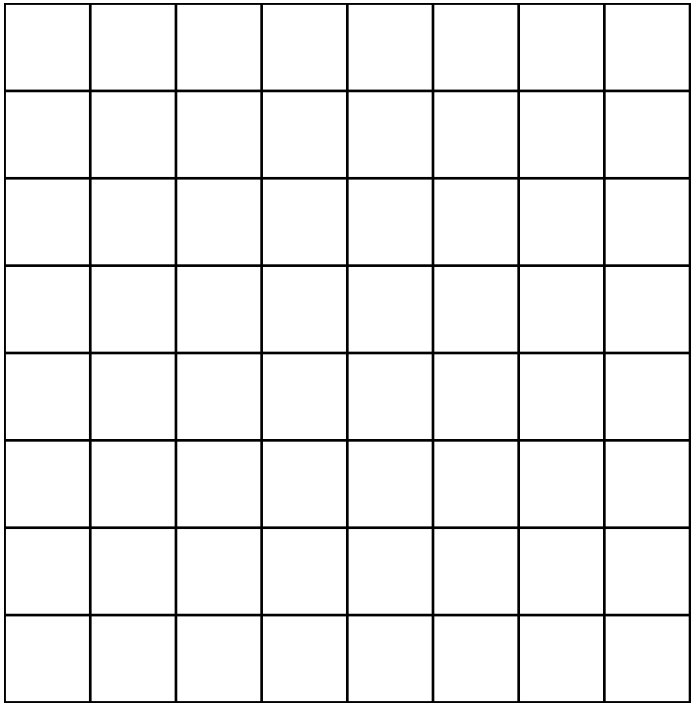
2. Shift and stitch

- What does this 4x4 correspond to?

Output of forward pass [4x4]

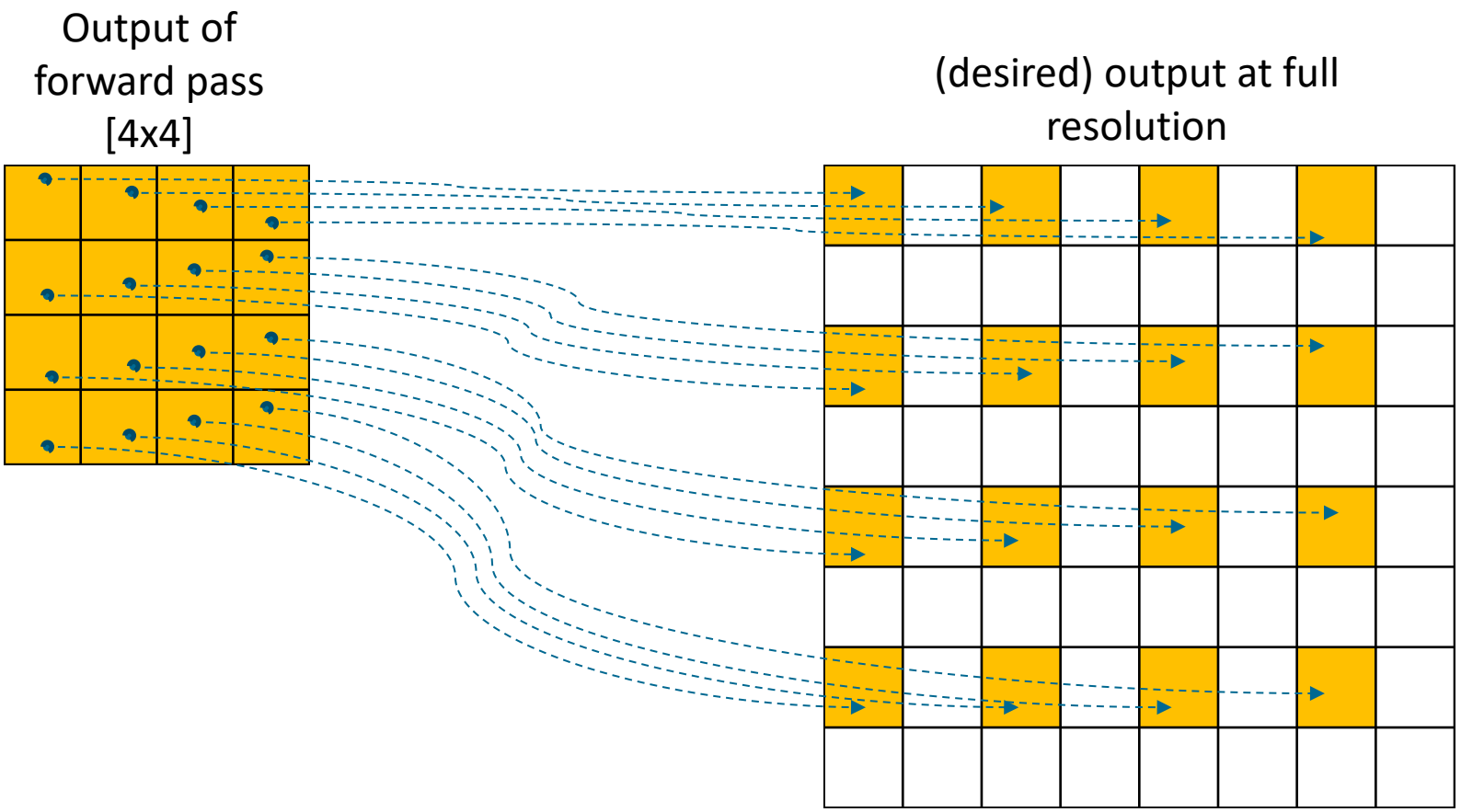


(desired) output at full resolution



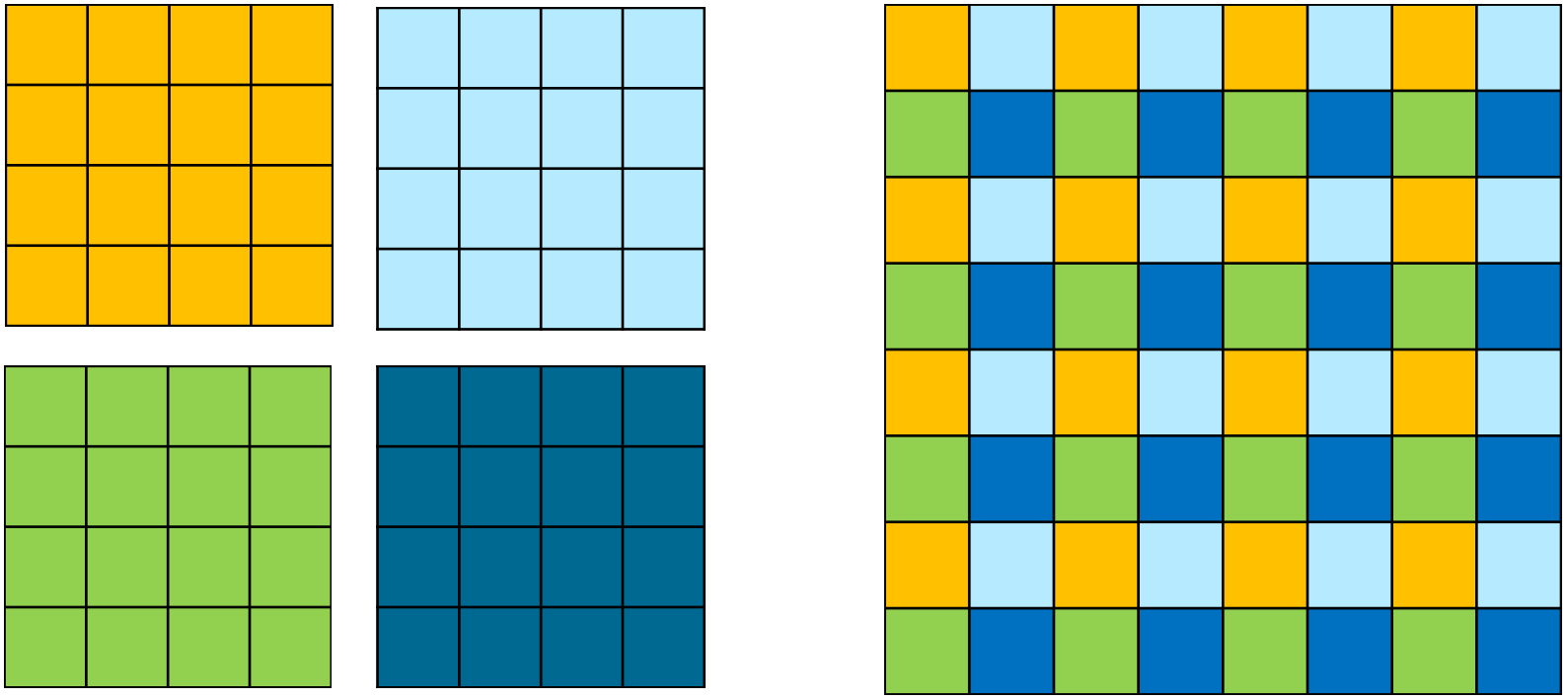
Fully convolutional network

- What does this 4x4 correspond to?



Fully convolutional network

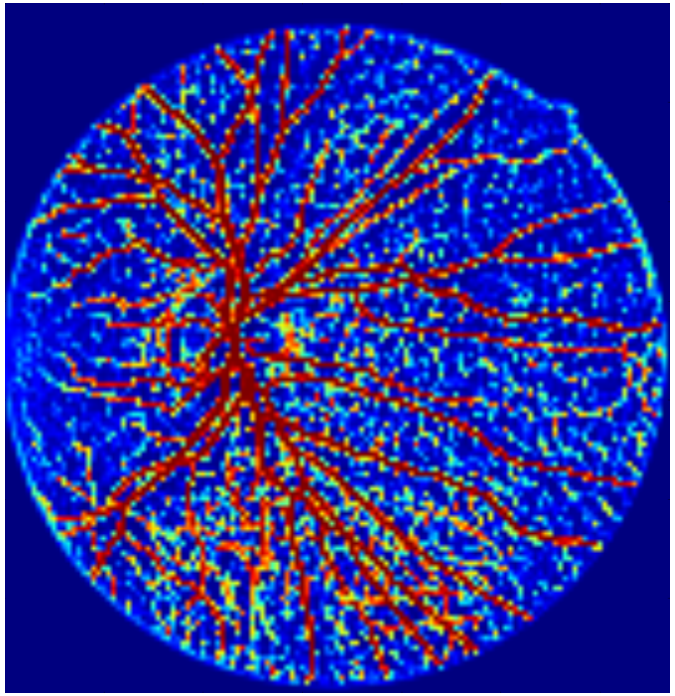
- How to get output at full resolution (intuition)



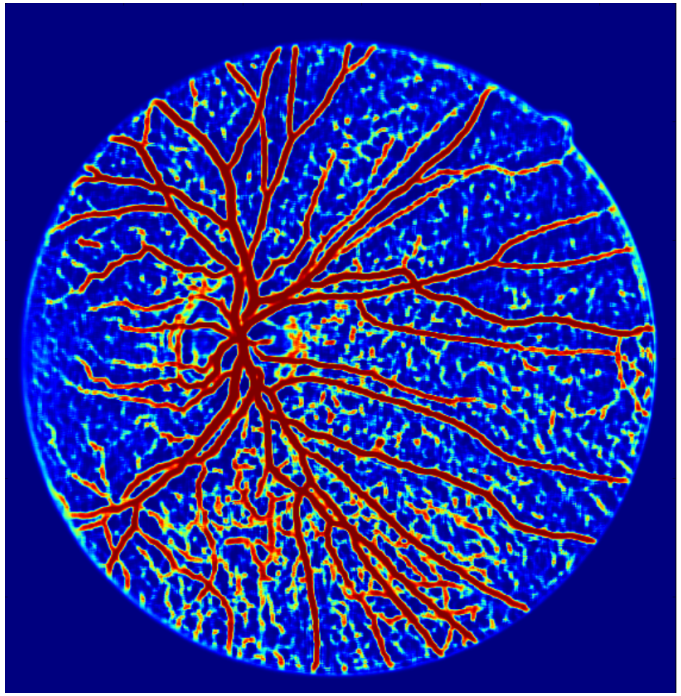
Fully convolutional network

- DRIVE data

Low resolution

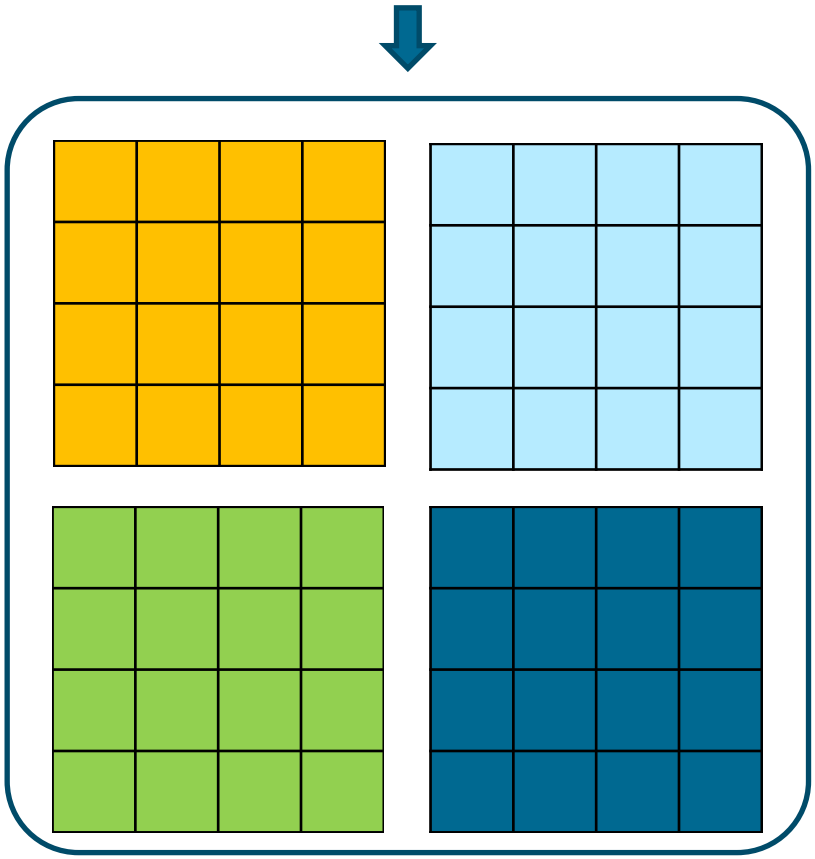


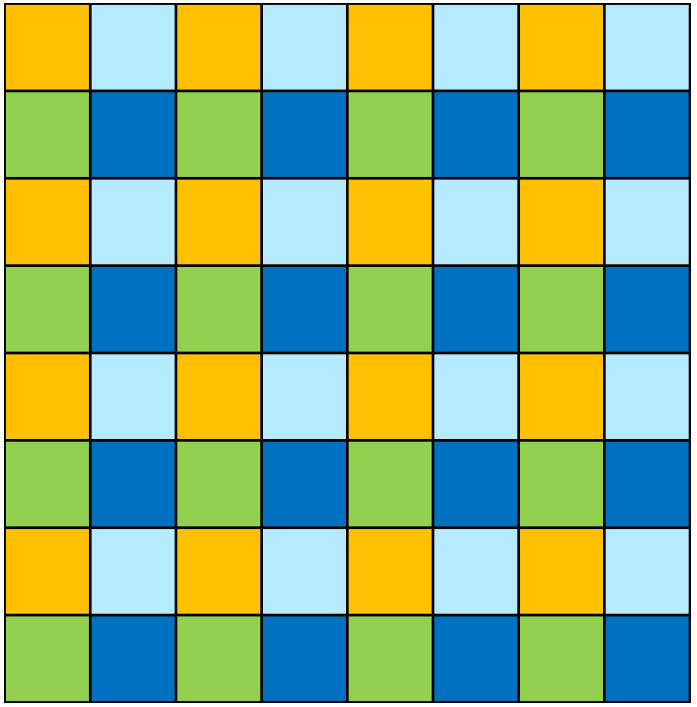
Full resolution



Shift and stitch

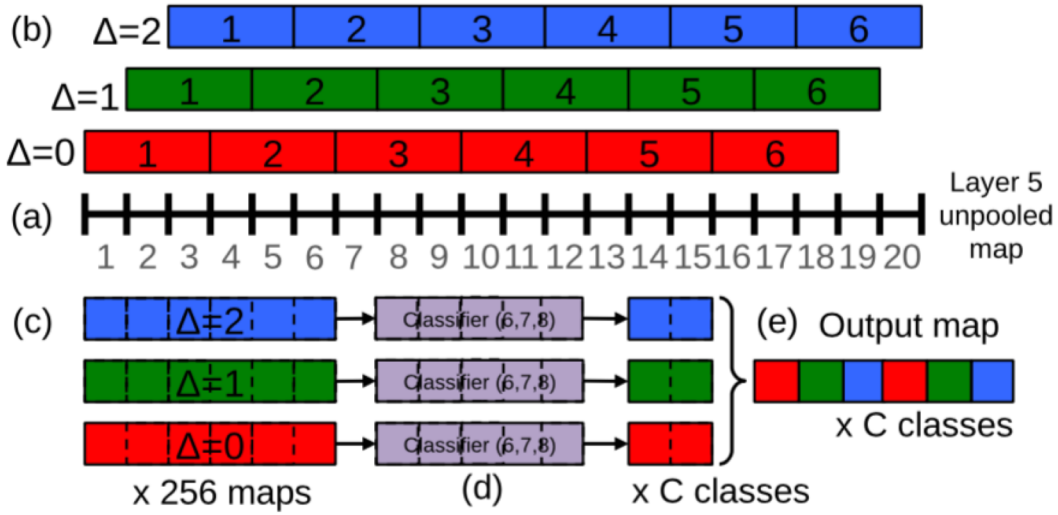
- How to get output at full resolution (intuition)
- How can we get these four 4x4 outputs?





Shift and stitch

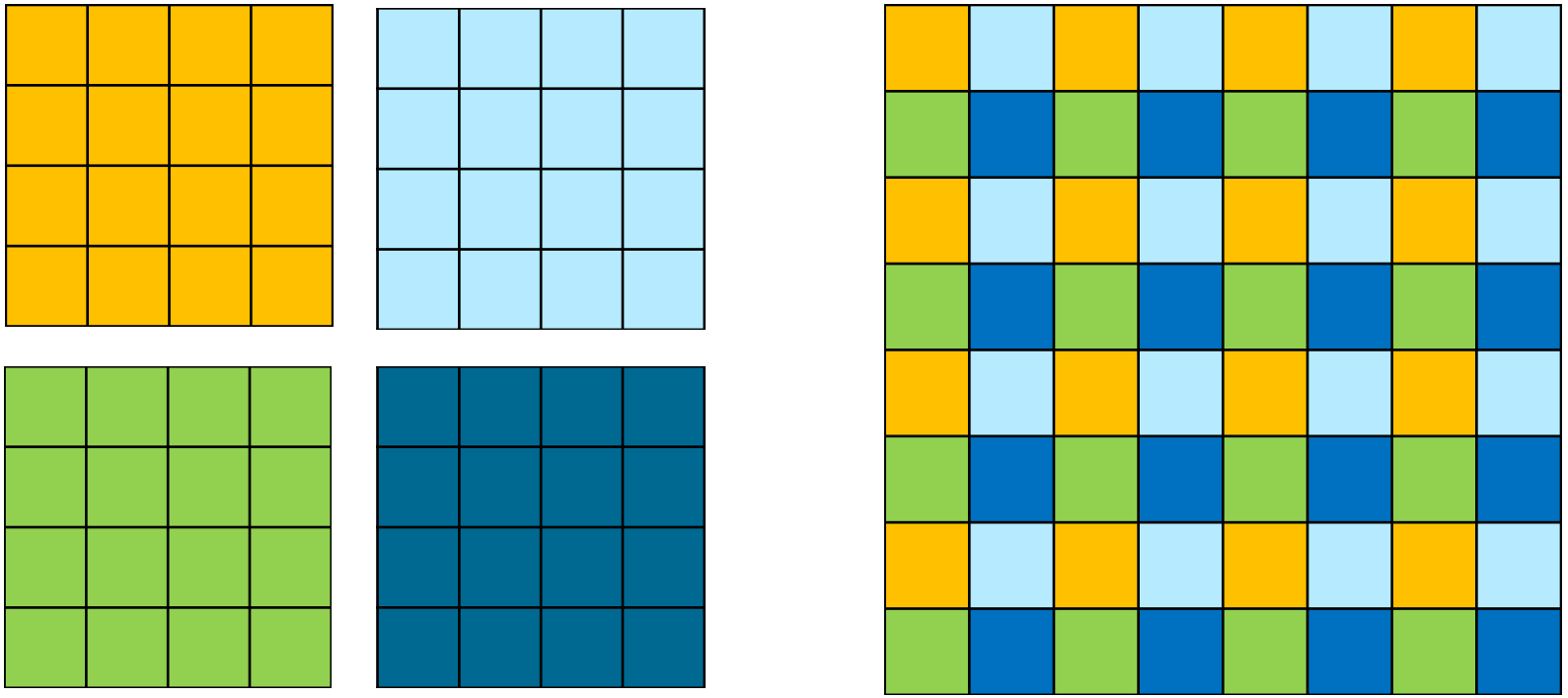
- The idea is to
  1. Shift the input
  2. Classify
  3. Interlace the results



P. Sermanet et al., 2013 [OverFeat]

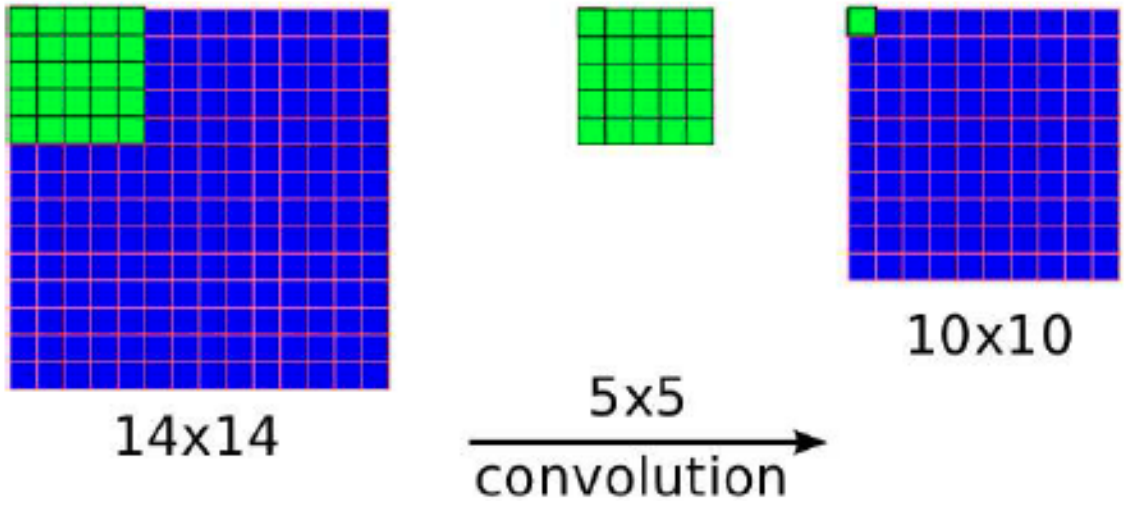
Shift and stitch

- **How many times** do we shift?
- It depends **how much** the network **down-samples** the input
- If we have M max-pooling layers, it's (roughly) 2^M times
- In this example, we shift 2 times along *x* and 2 times along *y*



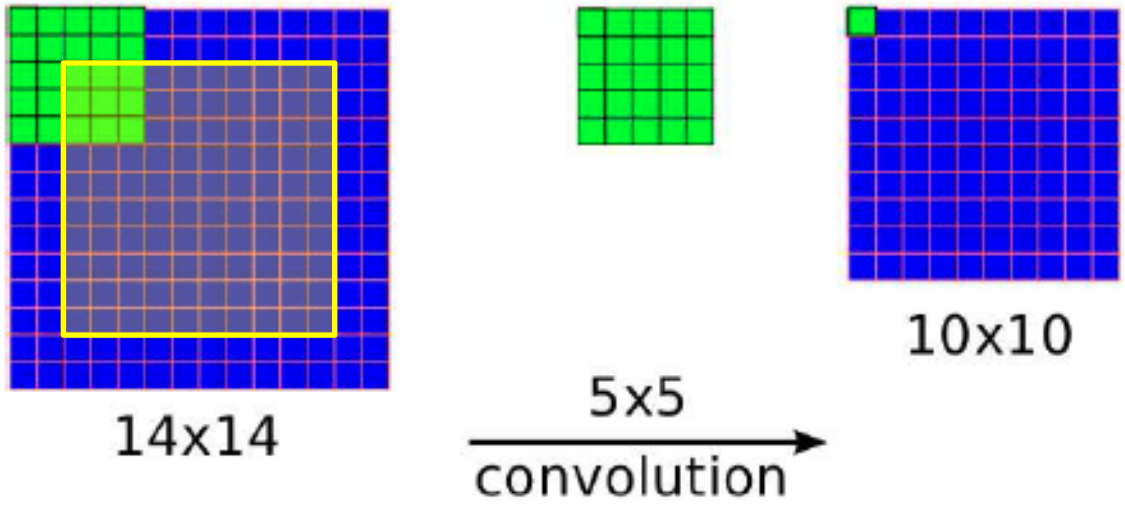
Shift and stitch

- When we do a forward pass, which input pixel does the output pixel correspond to?



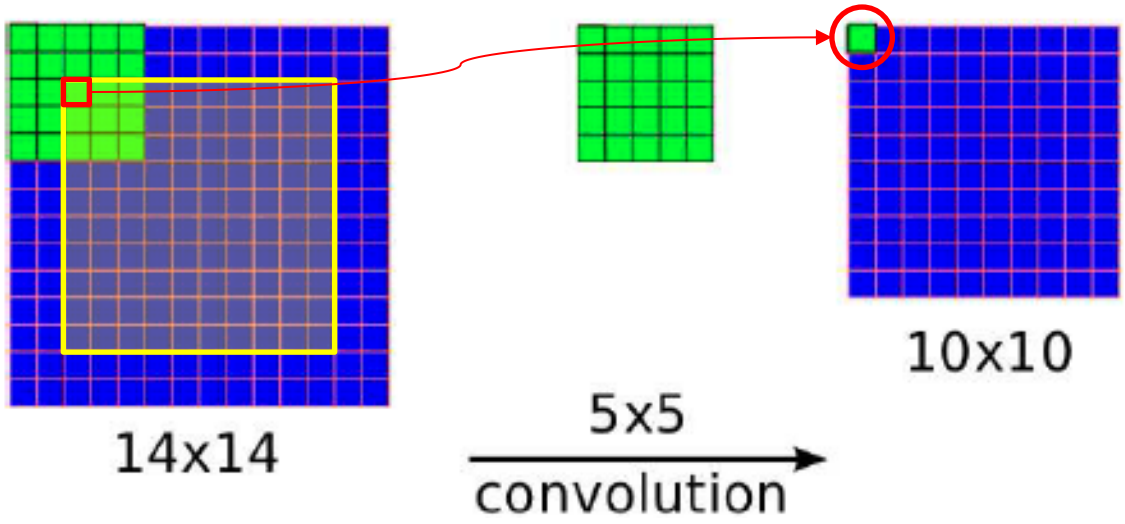
Shift and stitch

- When we do a forward pass, which input pixel does the output pixel correspond to?



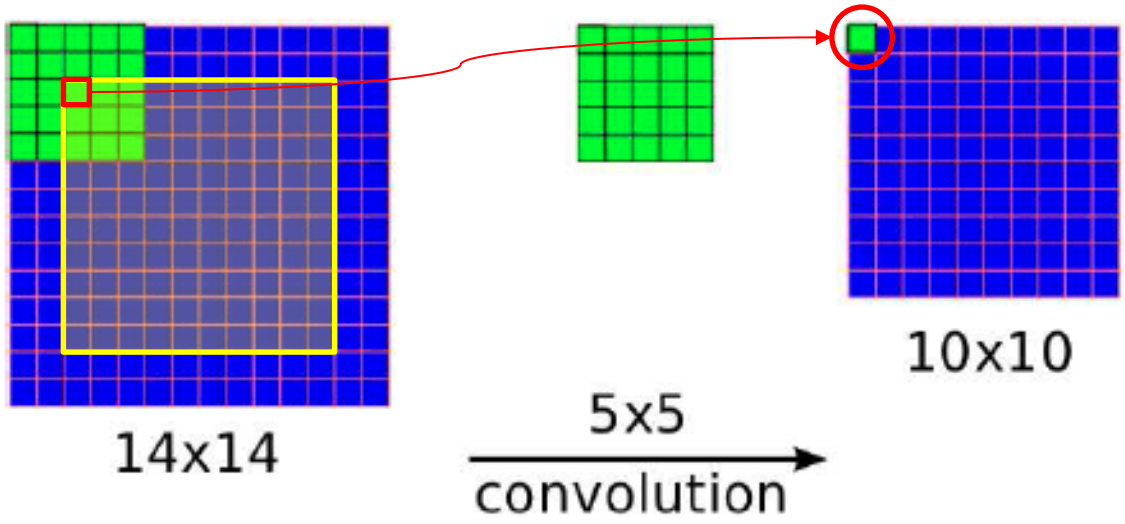
Shift and stitch

- When we do a forward pass, which input pixel does the output pixel correspond to?



Shift and stitch

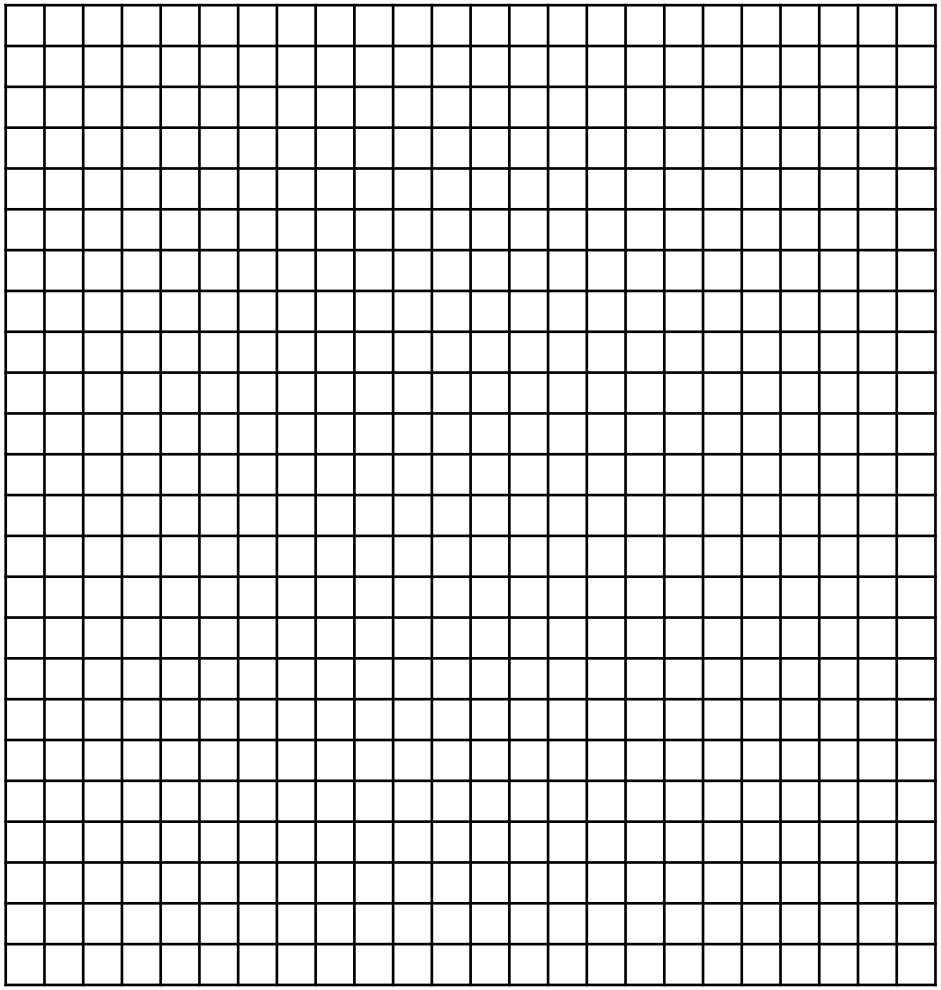
- Our network is now a big filter with **receptive field** PxP
- To make the top-left pixel in the output correspond to the top-left pixel in the input, we have to do some **padding**

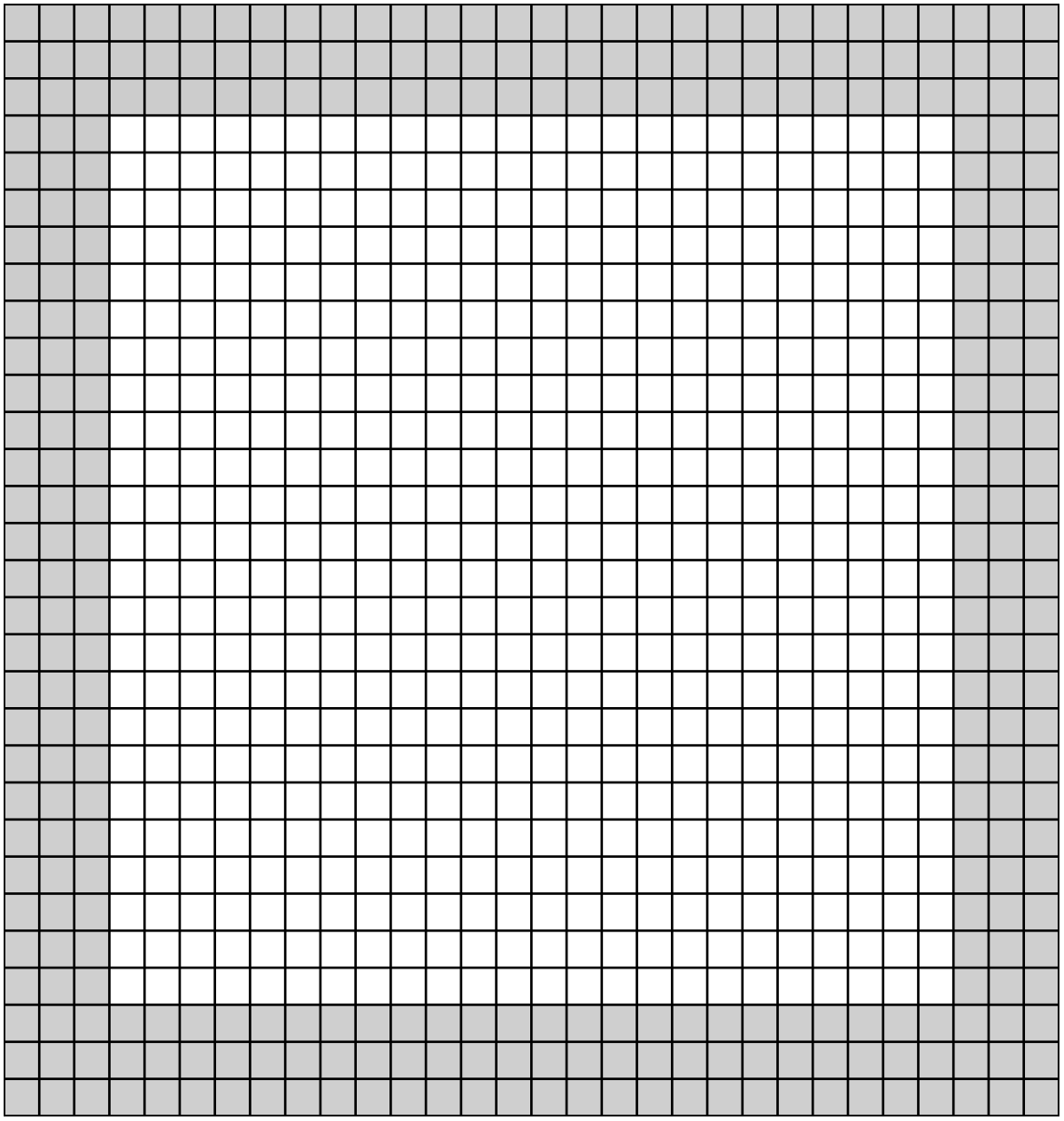


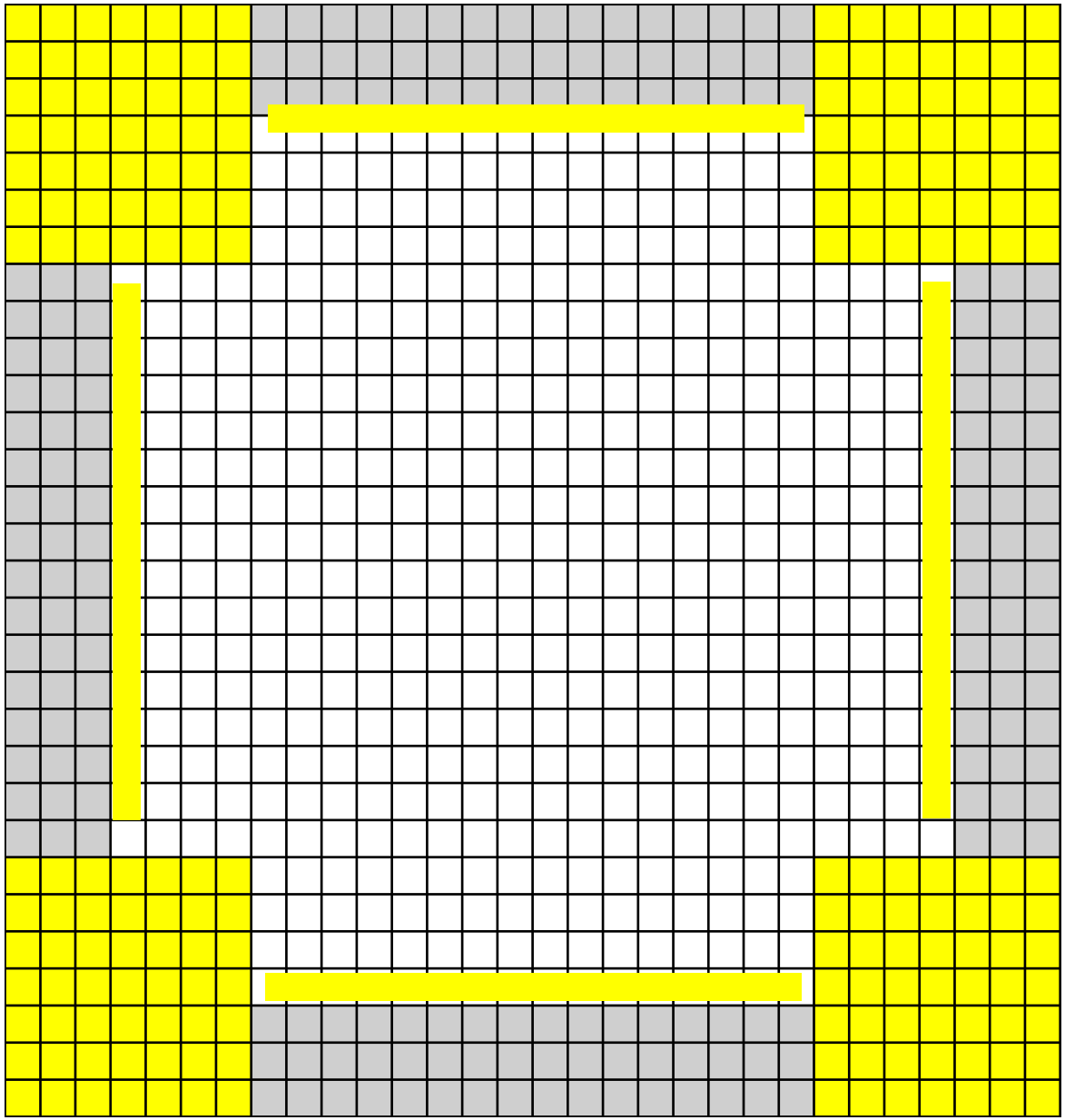
Shift and stitch

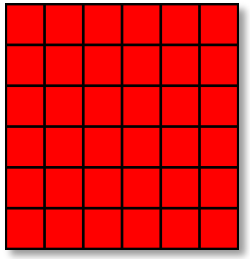
Example

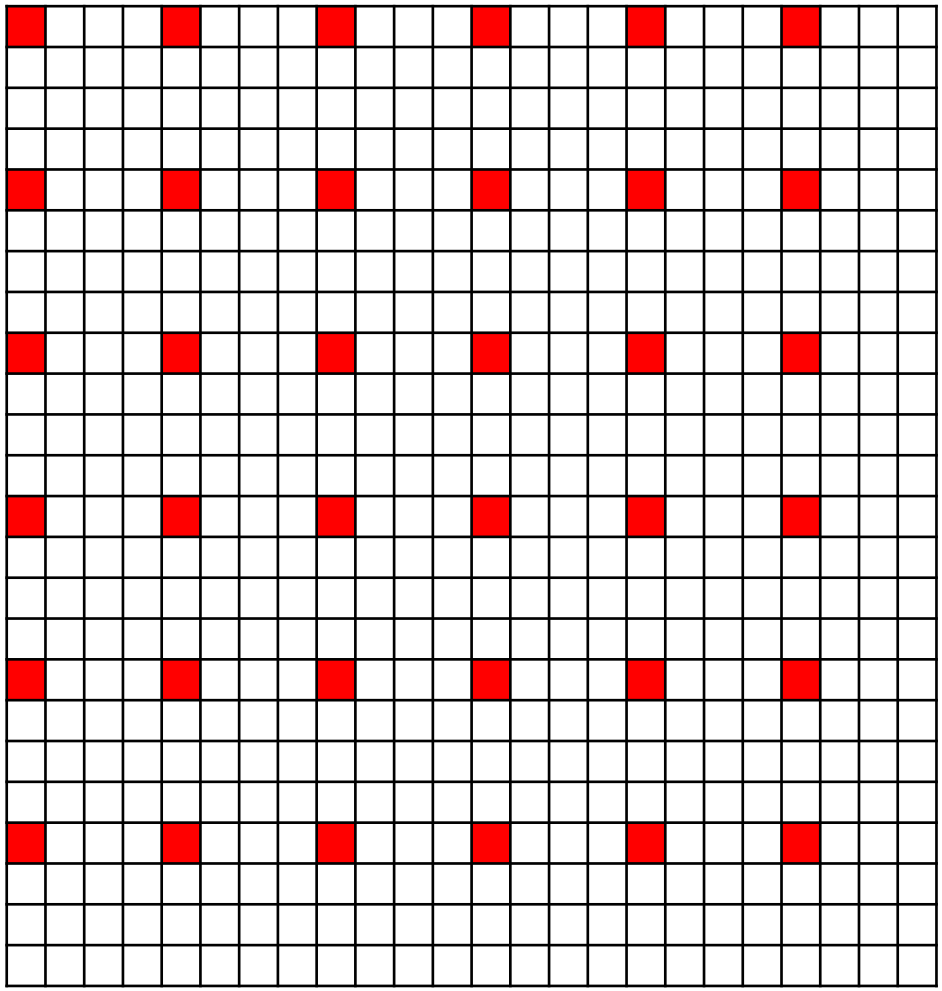
- Network with receptive field (7, 7)
- 2 pooling layers
- Padded convolutions (‘same’)
- Input image size = (24, 24)

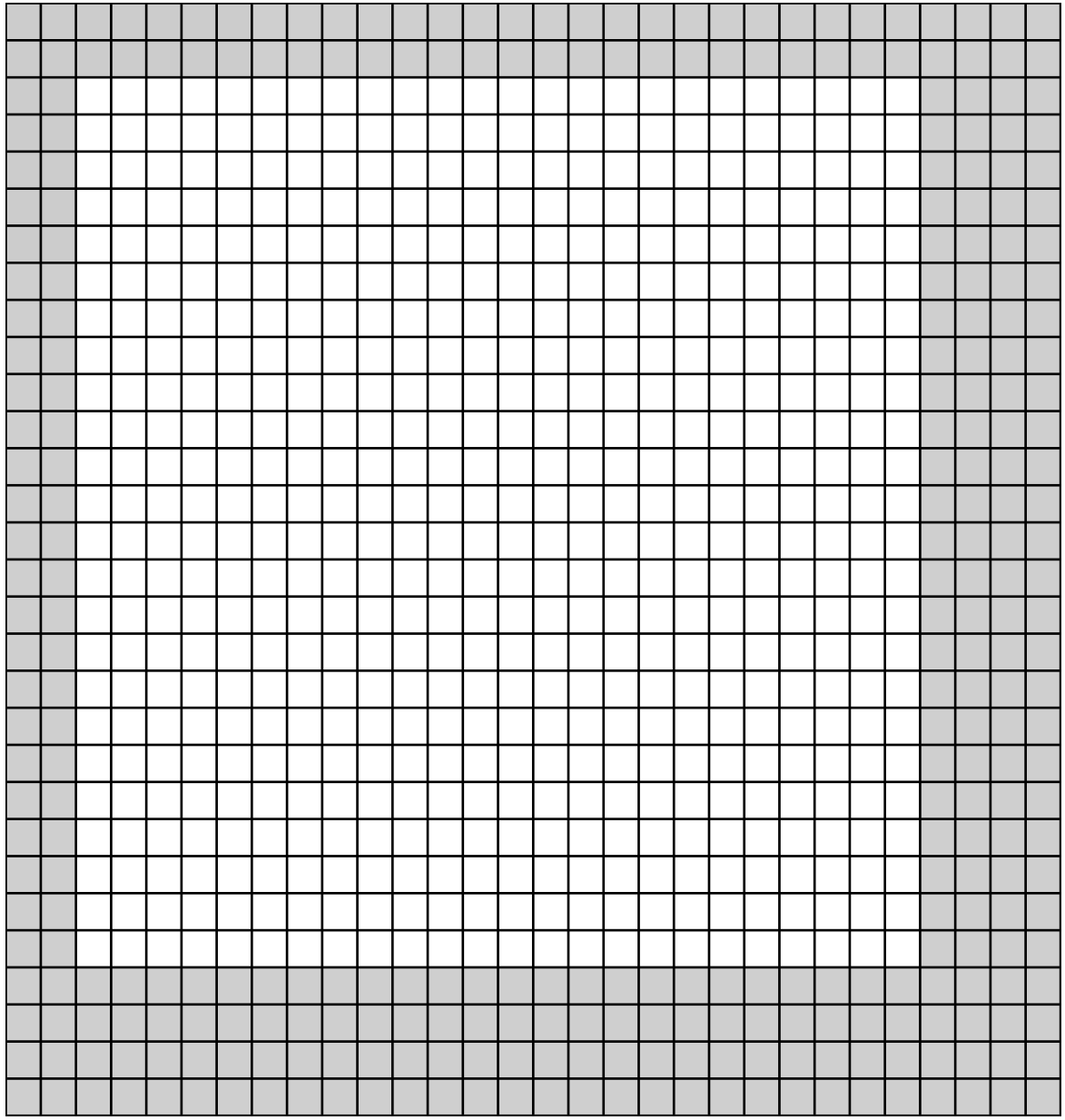


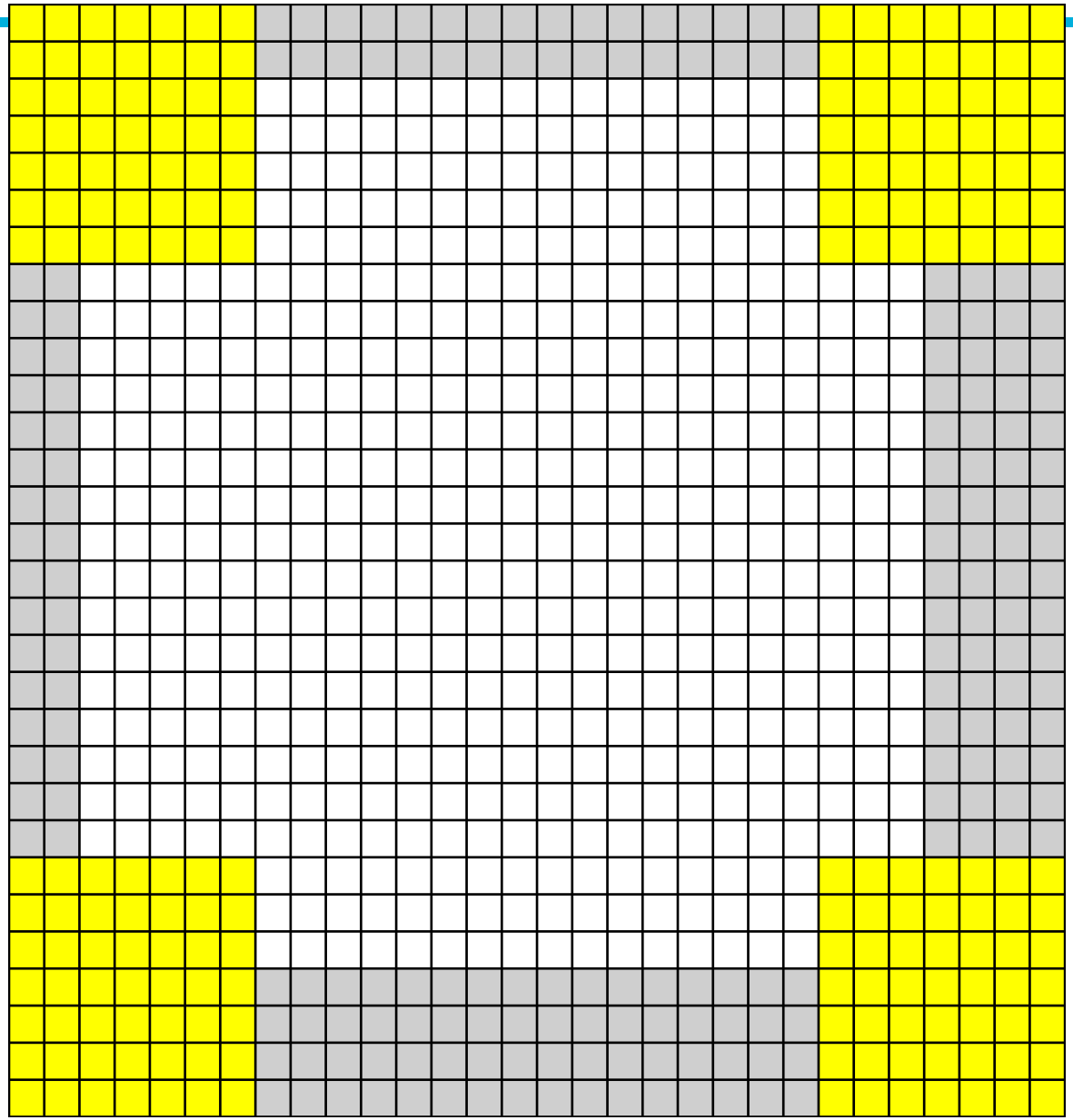


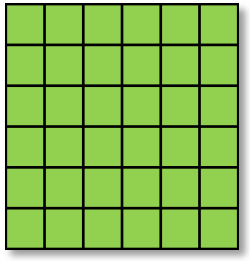


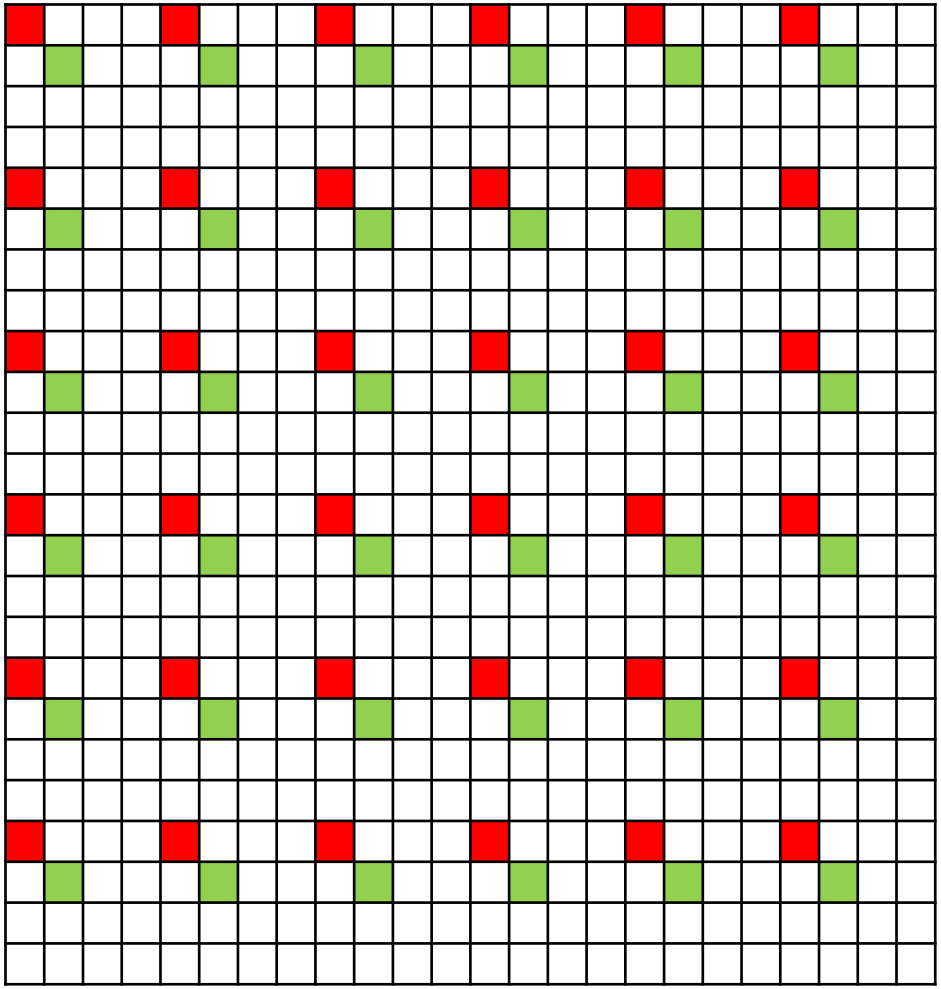












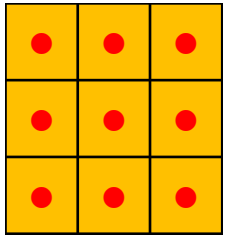
Training patch size?

- Fully-convolutional networks are trained using patches
- The patch size defines a trade-off between accuracy and the use of context:
  - **Larger patches** require more max-pooling layers, which reduce localization accuracy
  - **Small patches** allow the network to see only little context

Dilated convolutions (2016)

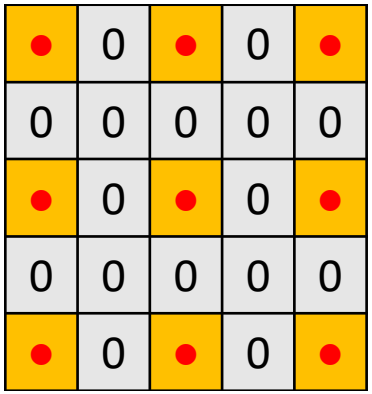
- The idea is to keep the **same amount of parameters** and **increase the receptive field**

● = filter weight



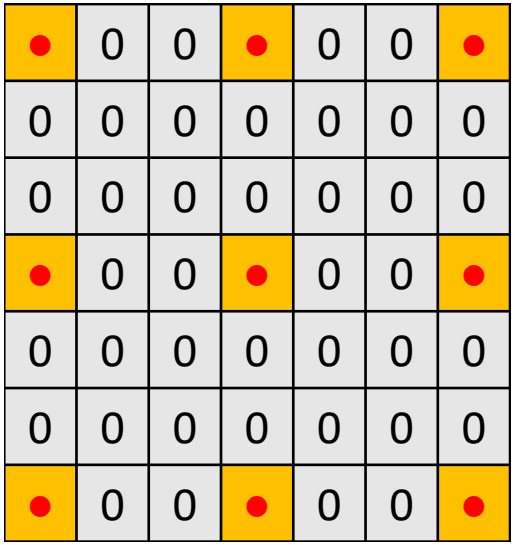
3x3

dilation rate = 1



5x5

dilation rate = 2

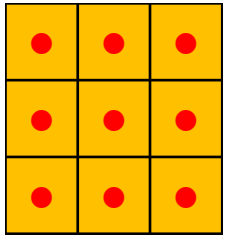


7x7

dilation rate = 3

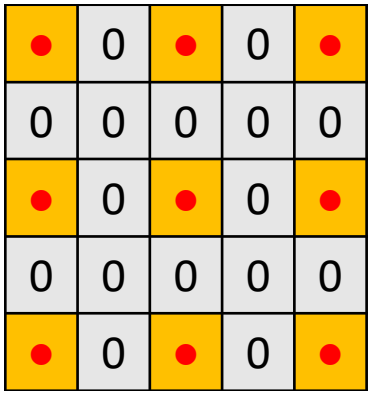
Dilated convolutions (2016)

- The idea is to keep the **same amount of parameters** and **increase the receptive field**



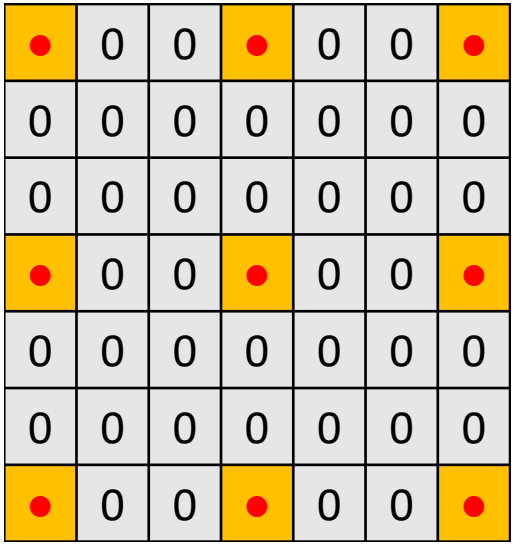
3x3

dilation rate = 1



5x5

dilation rate = 2



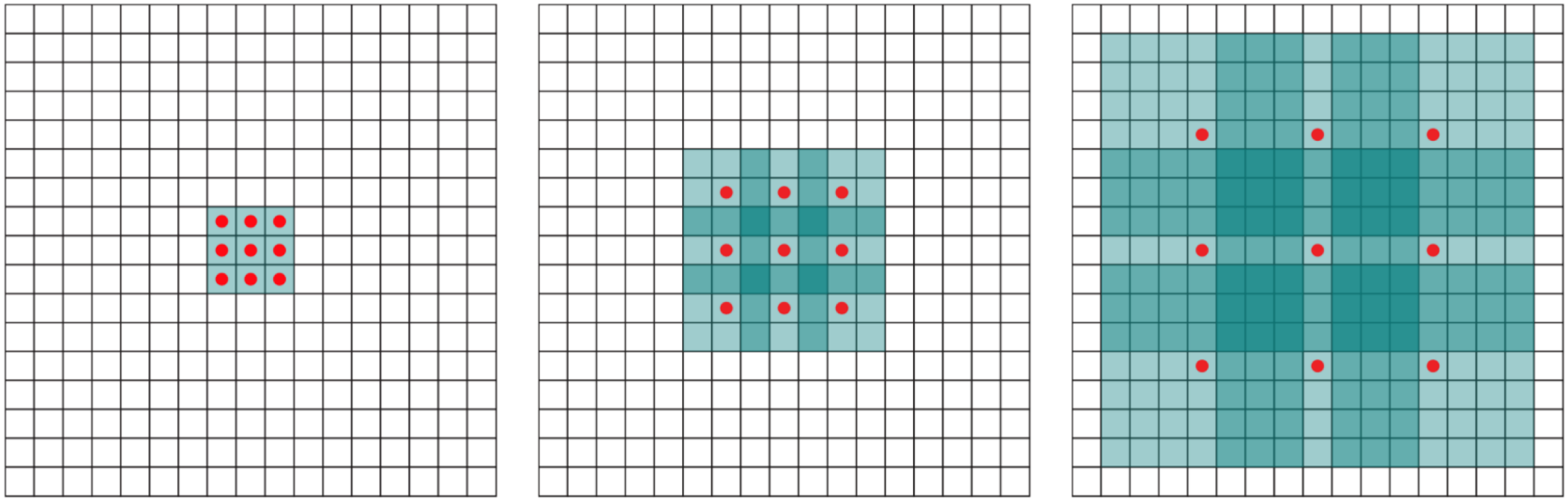
7x7

dilation rate = 3

Conv2D

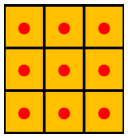
```
rides=(1, 1), padding='valid', data_format=None, dilation_rate=(1, 1), activation=None,
```

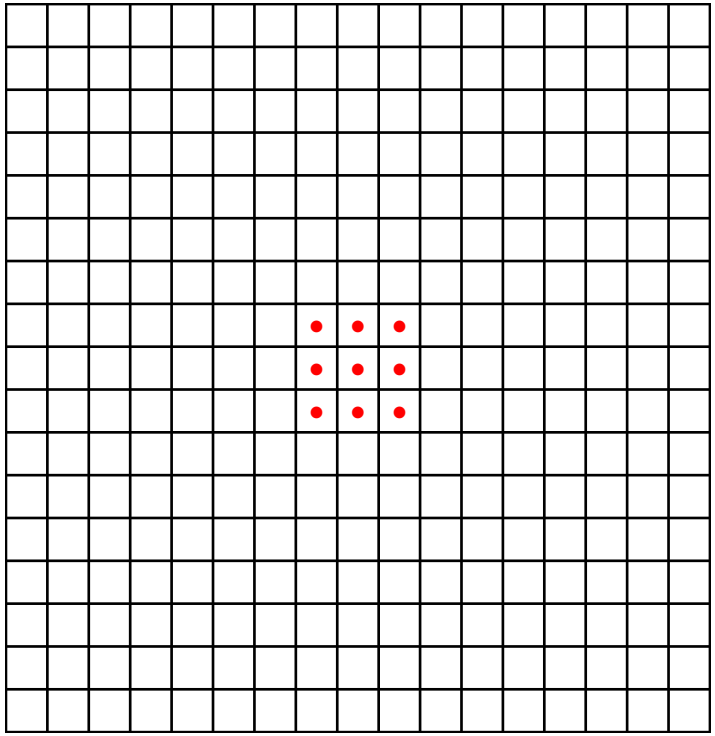
Dilated convolutions



Dilated convolutions

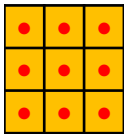
- 3x3 filter
- Dilation rate = 1



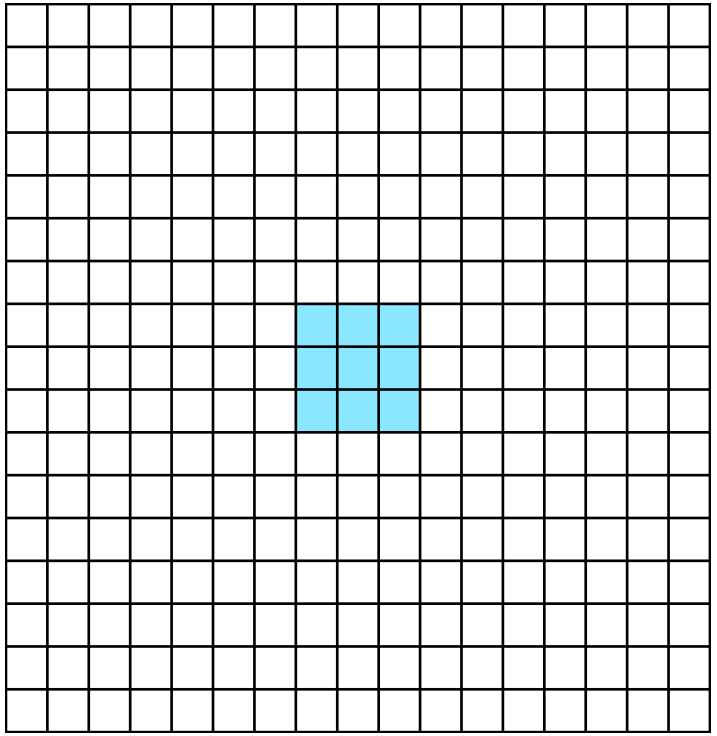


Dilated convolutions

- 3x3 filter
- Dilation rate = 1

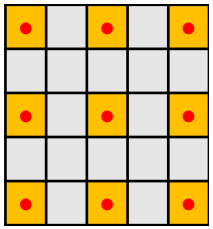


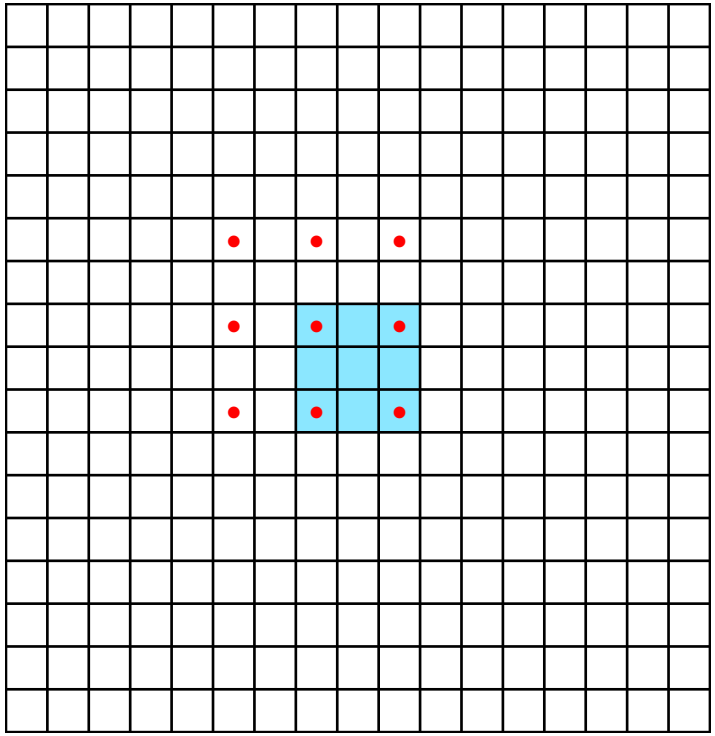
receptive field = 3x3



Dilated convolutions

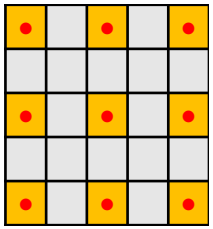
- 3x3 filter
- Dilation rate = 2

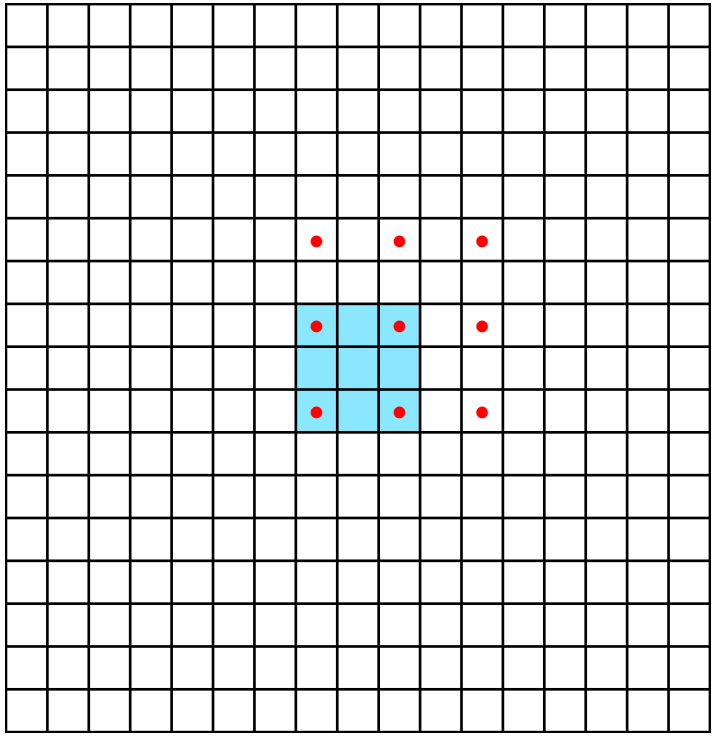




Dilated convolutions

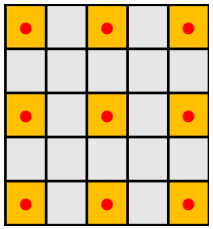
- 3x3 filter
- Dilation rate = 2

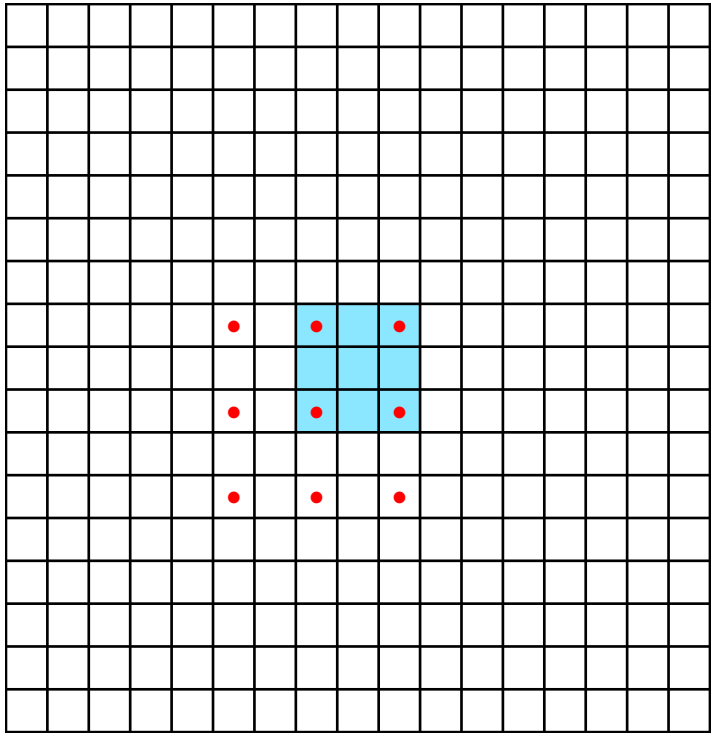




Dilated convolutions

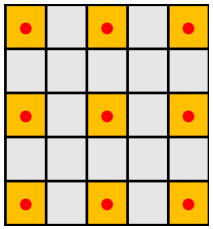
- 3x3 filter
- Dilation rate = 2

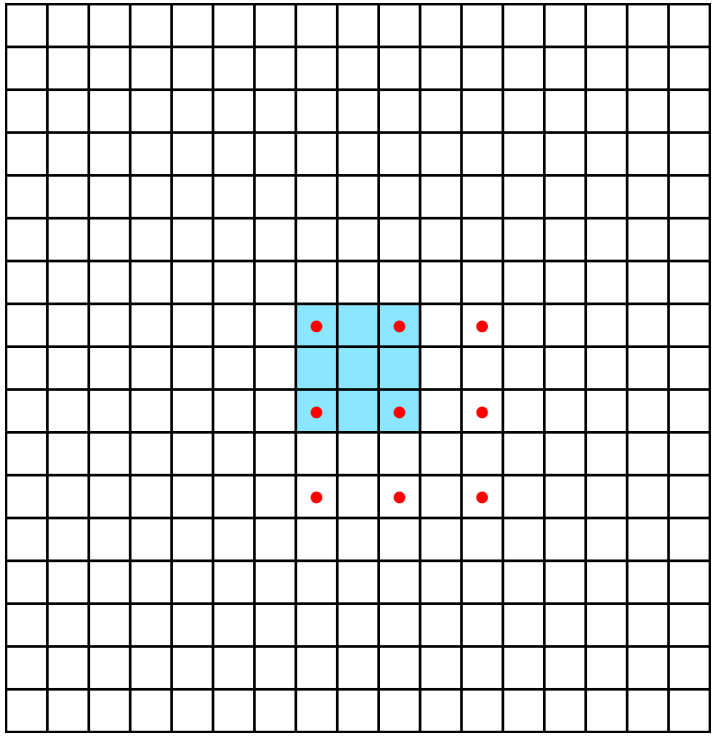




Dilated convolutions

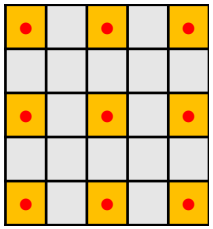
- 3x3 filter
- Dilation rate = 2



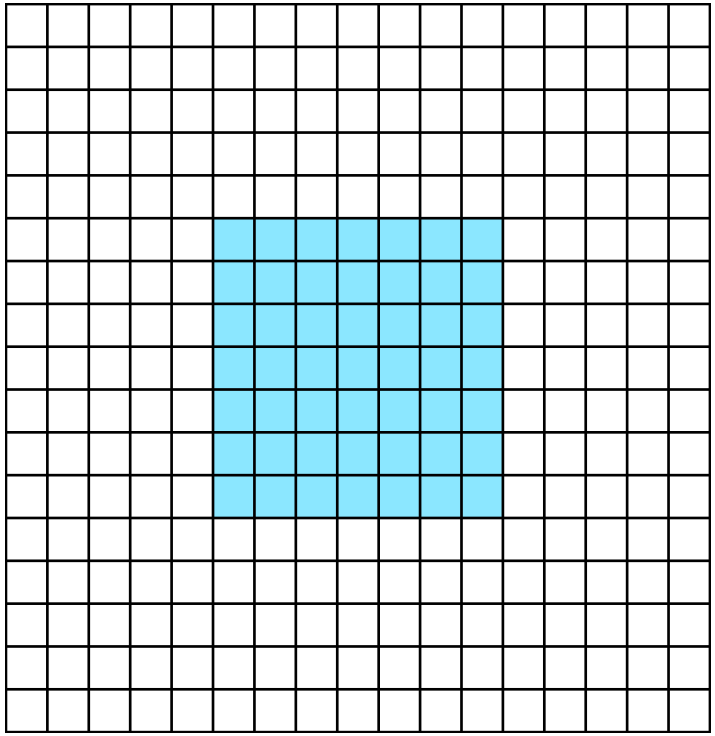


Dilated convolutions

- 3x3 filter
- Dilation rate = 2

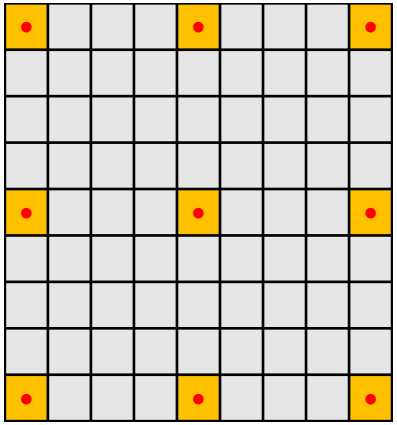


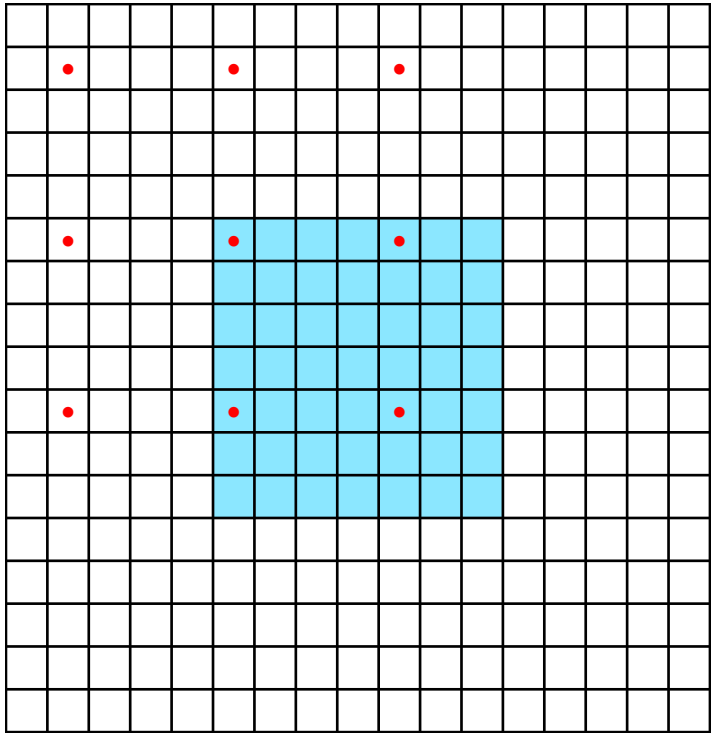
receptive field = 7x7



Dilated convolutions

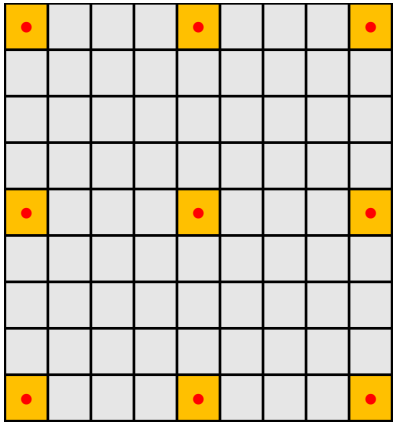
- 3x3 filter
- Dilation rate = 4

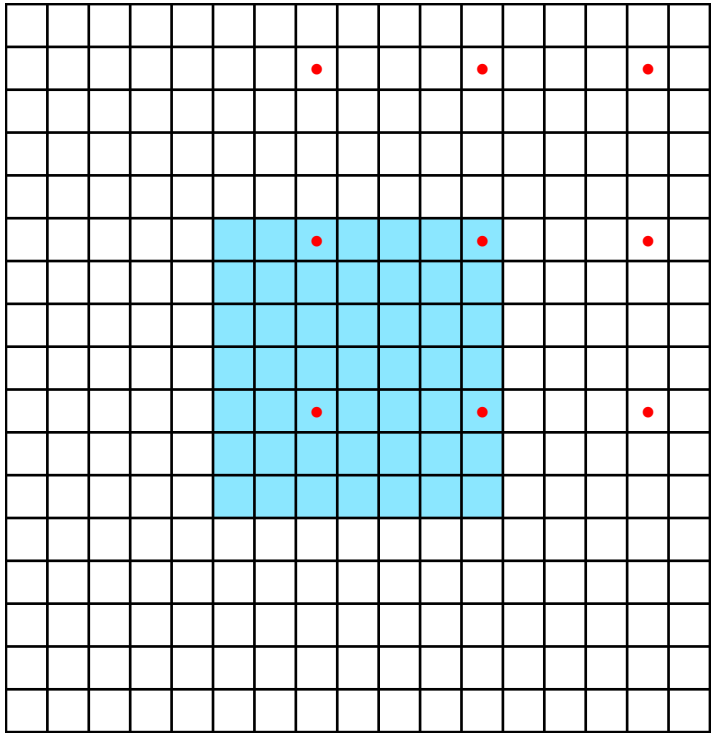




Dilated convolutions

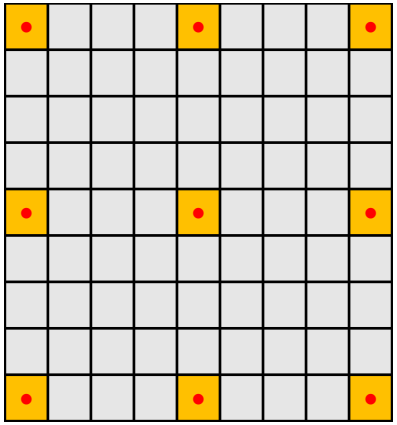
- 3x3 filter
- Dilation rate = 4

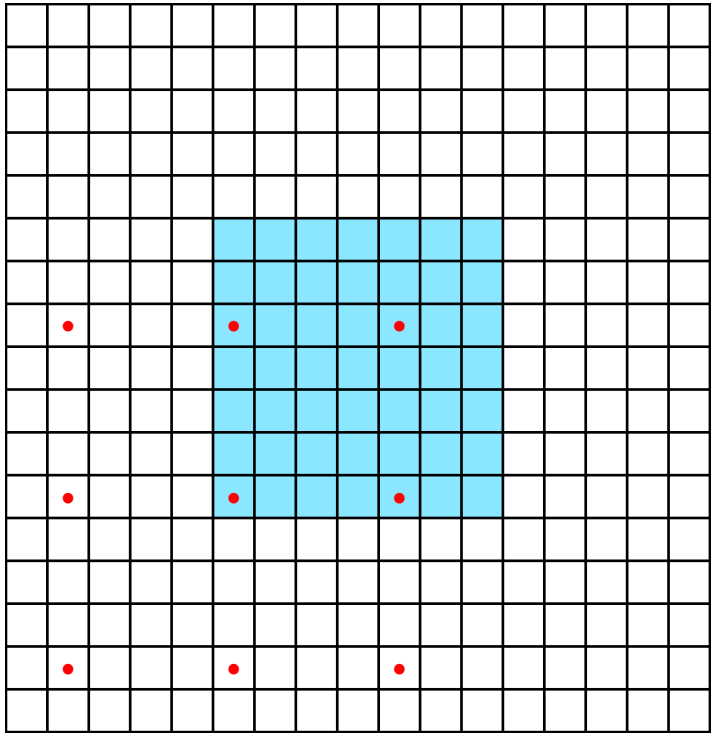




Dilated convolutions

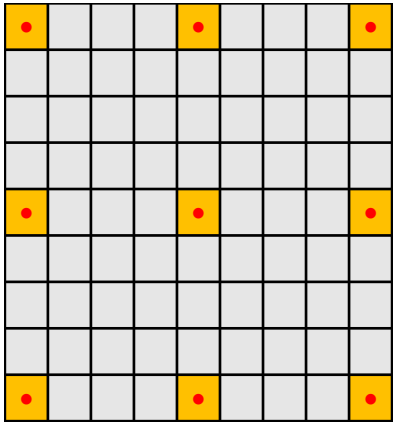
- 3x3 filter
- Dilation rate = 4

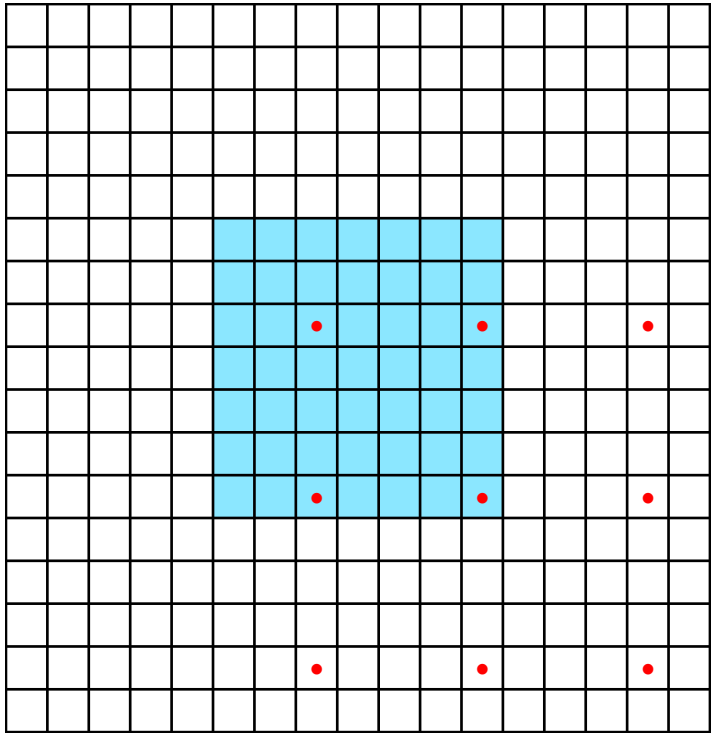




Dilated convolutions

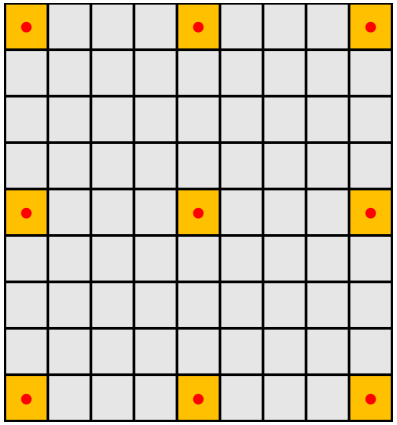
- 3x3 filter
- Dilation rate = 4



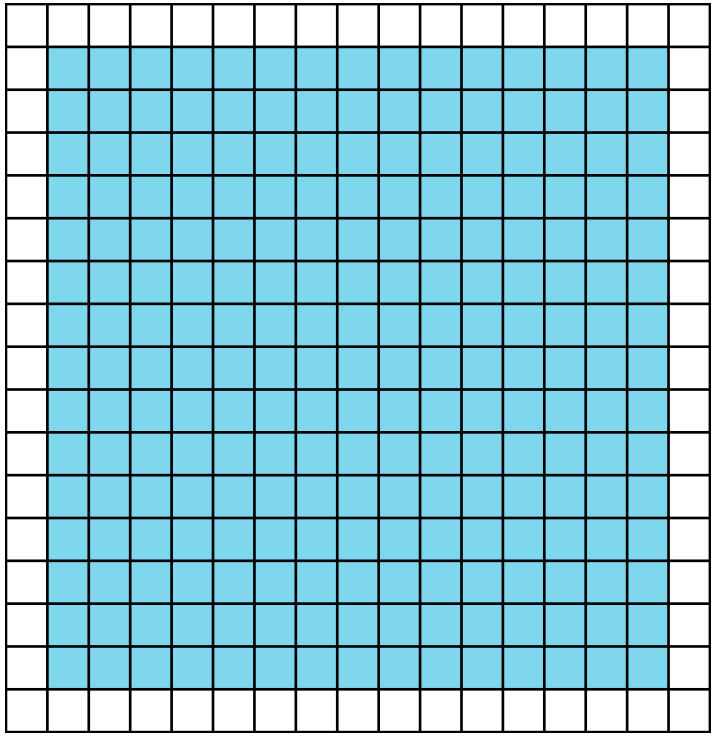


Dilated convolutions

- 3x3 filter
- Dilation rate = 4



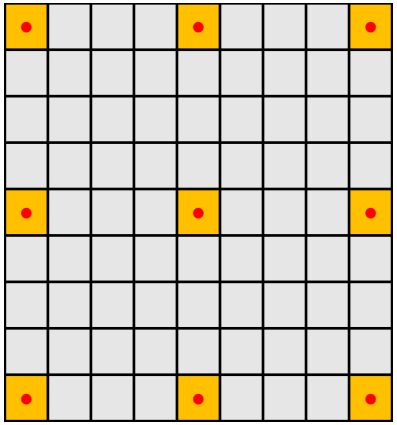
receptive field = 15x15

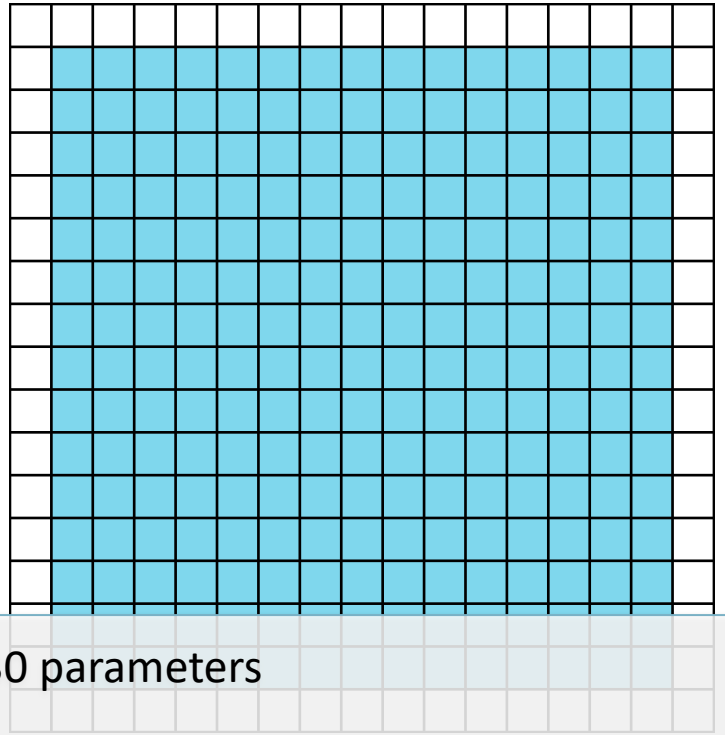


Dilated convolutions

- 3x3 filter
- Dilation rate = 4

receptive field = 15x15





- Three layers with 3x3 filters: 3 x (3x3 + 1) = 30 parameters
- No pooling layers
- Receptive field quickly increases to 15x15

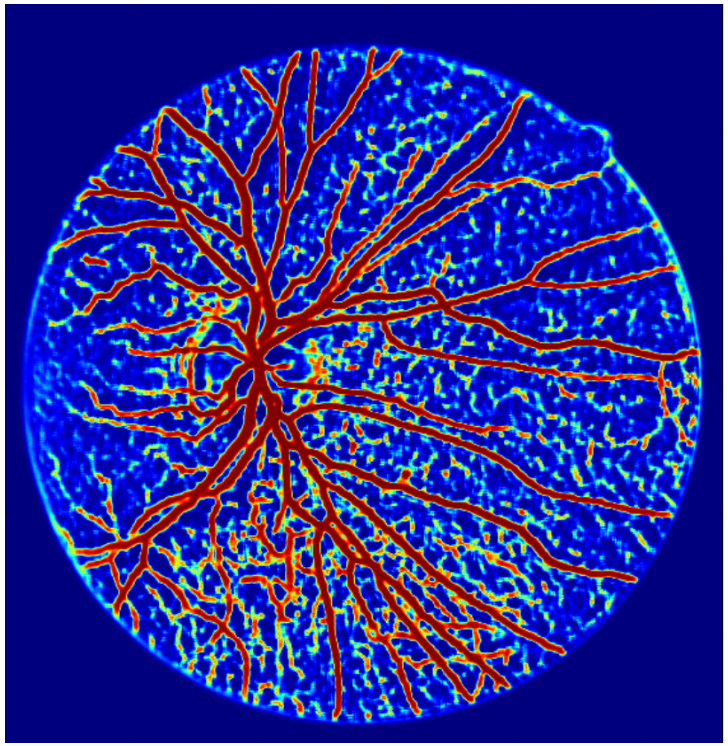
Dilated convolutions

- Exponentially **expanding** receptive field
- Can be plugged into **existing** architectures
- No pooling, no subsampling
- Allows dense prediction at **full resolution**
- **No new filter is made**, only convolution is done in a different way!

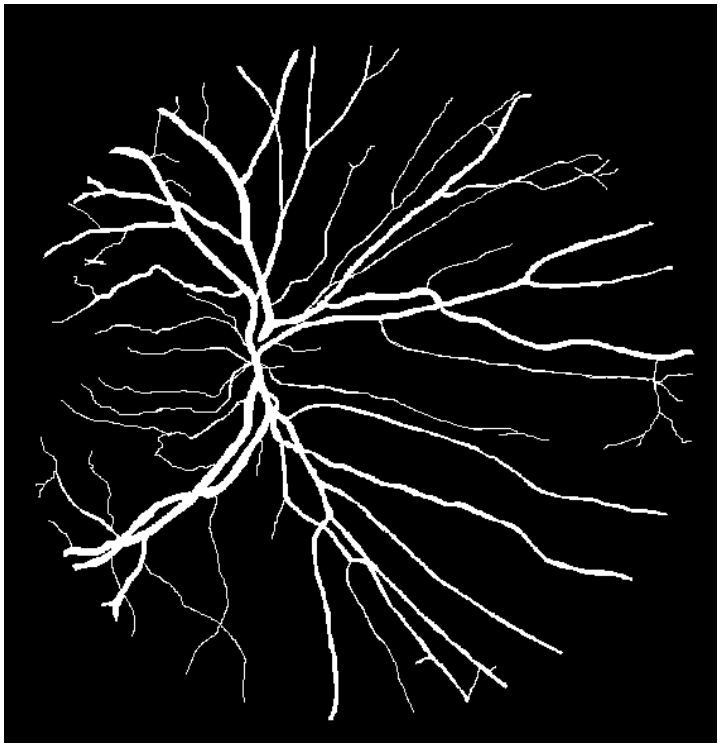
Loss function for segmentation

- When we train to predict a full resolution map, what's the loss function?

Network output



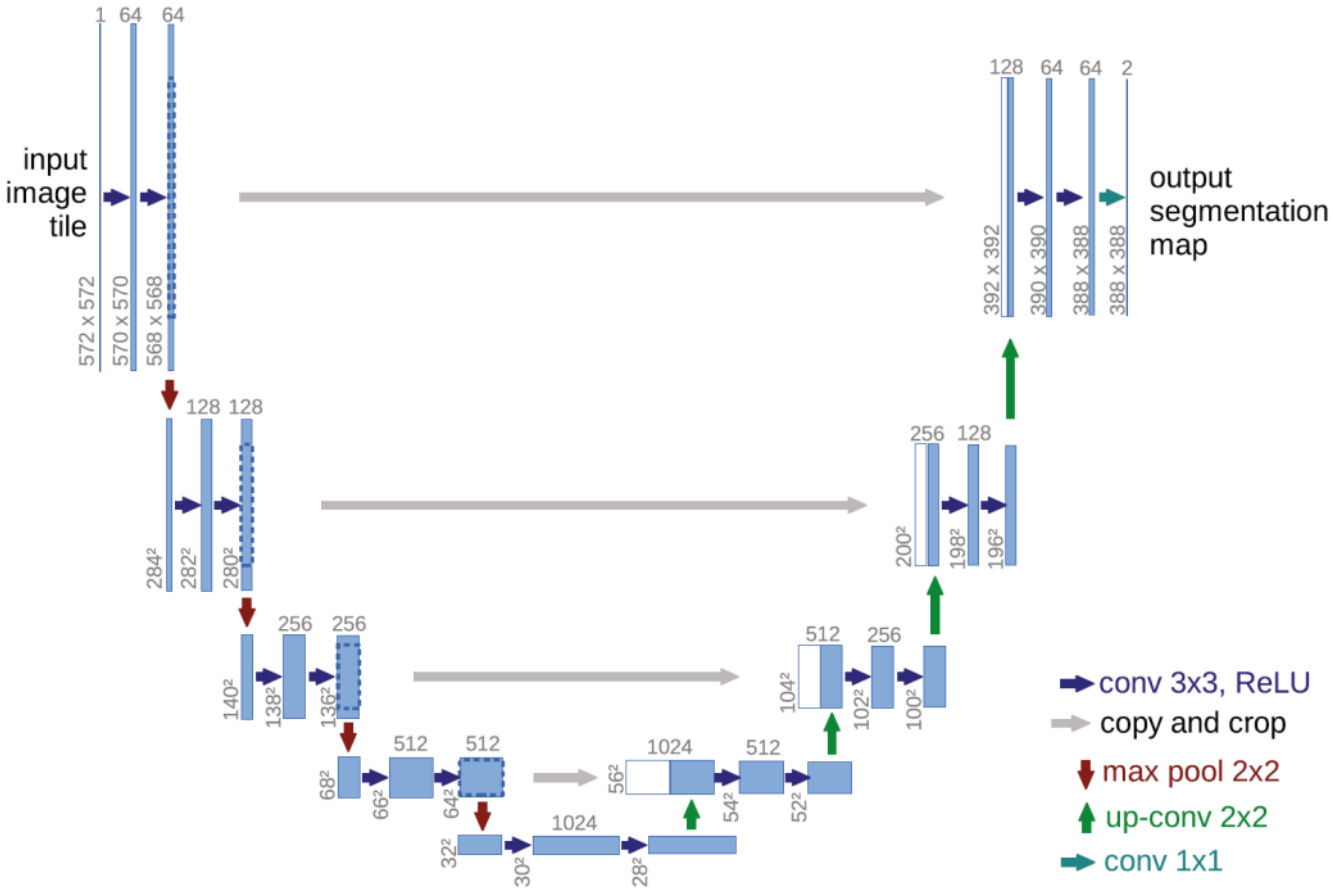
Target



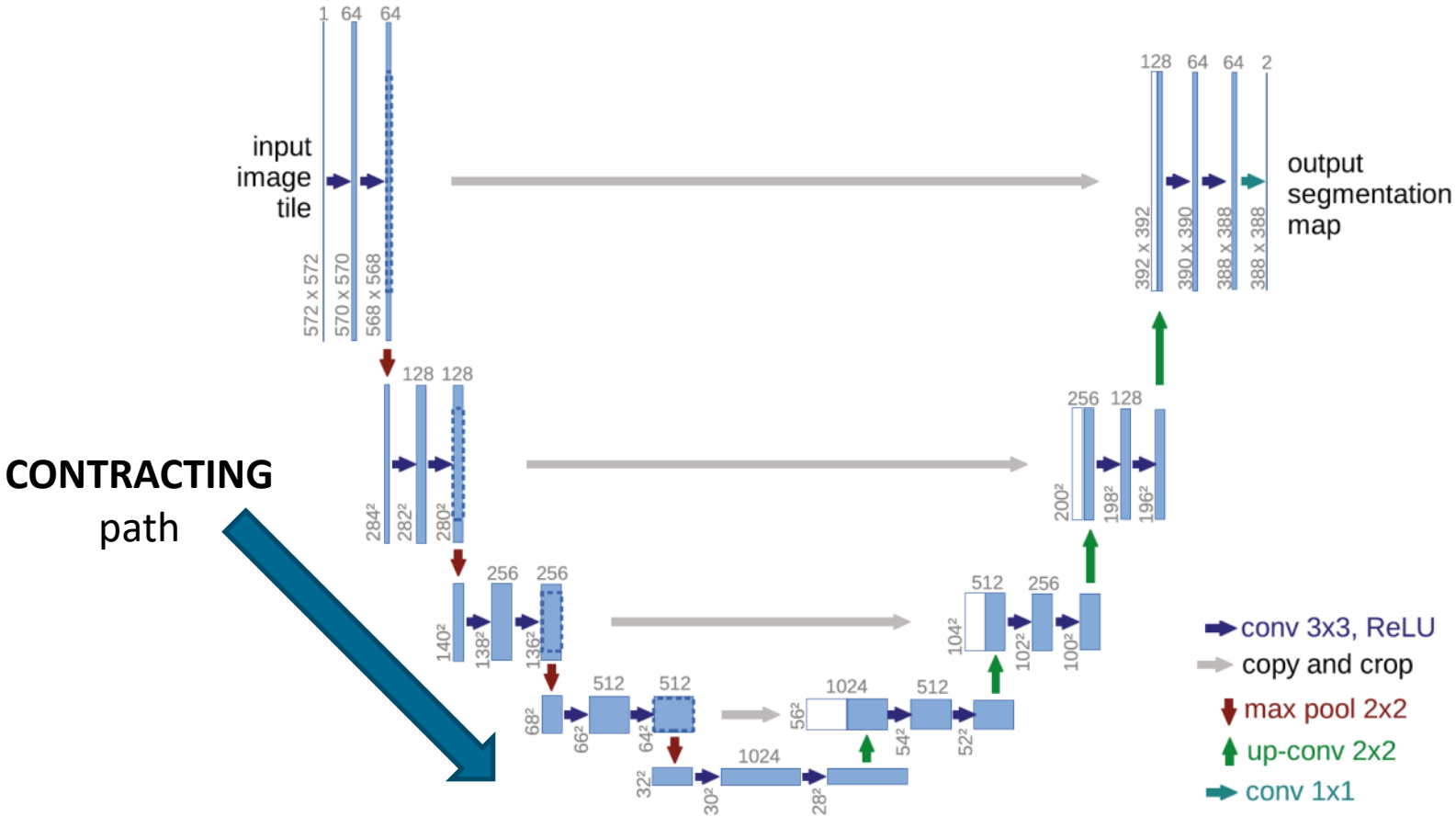
Loss function for segmentation

- The loss is now computed **per pixel**!
- Each pixel counts as one sample during training
- The cost is computed by averaging the losses over the full-res output
- You can use:
  - Cross-entropy loss
  - Log-loss
  - Dice score
  - …

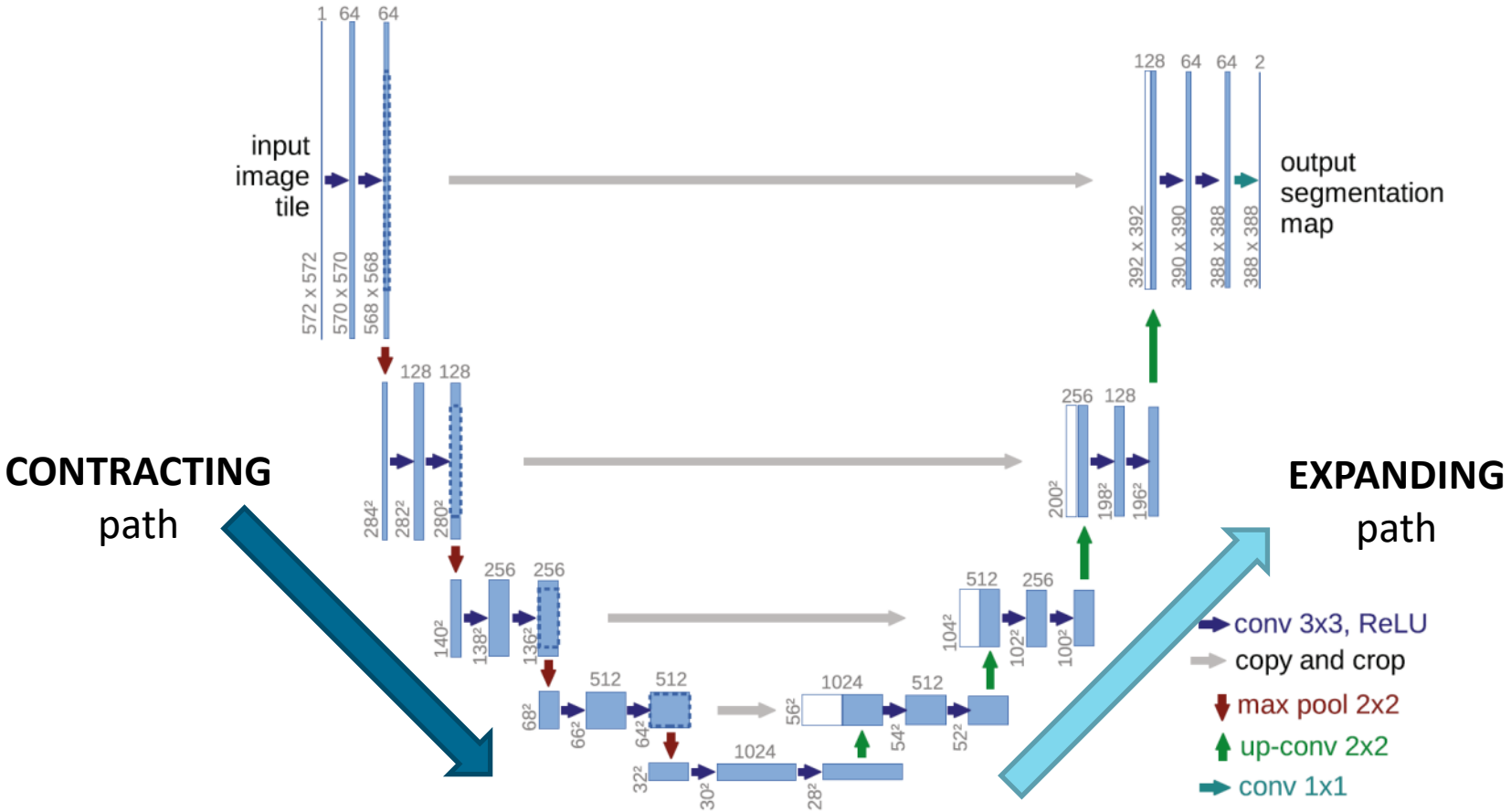
U-Net, 2015


input image tile
output segmentation map
conv 3x3, ReLU
copy and crop
max pool 2x2
up-conv 2x2
conv 1x1

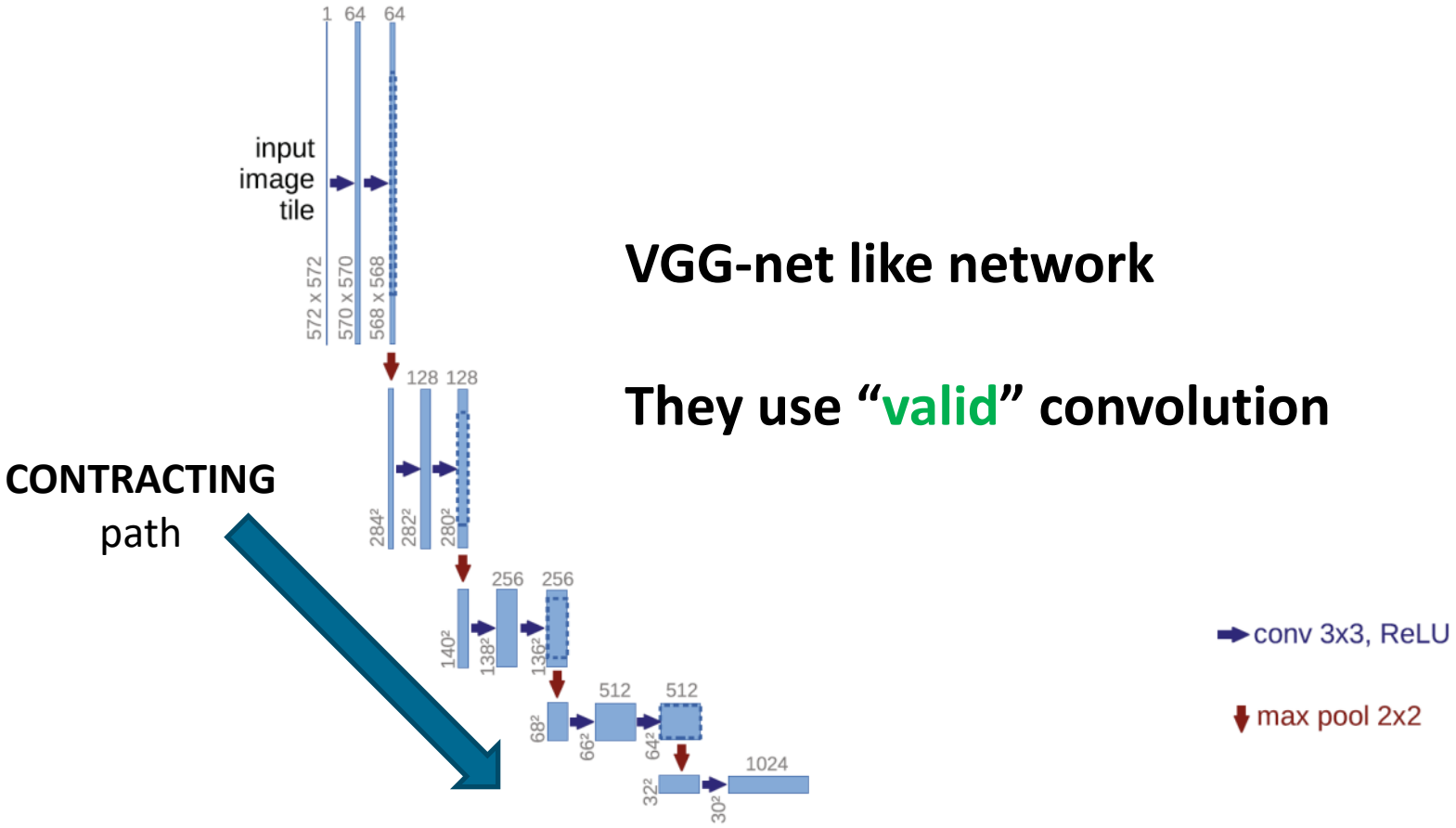
U-Net


CONTRACTING
path
input
image
tile
output
segmentation
map
conv 3x3, ReLU
copy and crop
max pool 2x2
up-conv 2x2
conv 1x1

U-Net


CONTRACTING
path
EXPANDING
path
input
image
tile
output
segmentation
map
conv 3x3, ReLU
copy and crop
max pool 2x2
up-conv 2x2
conv 1x1

U-Net

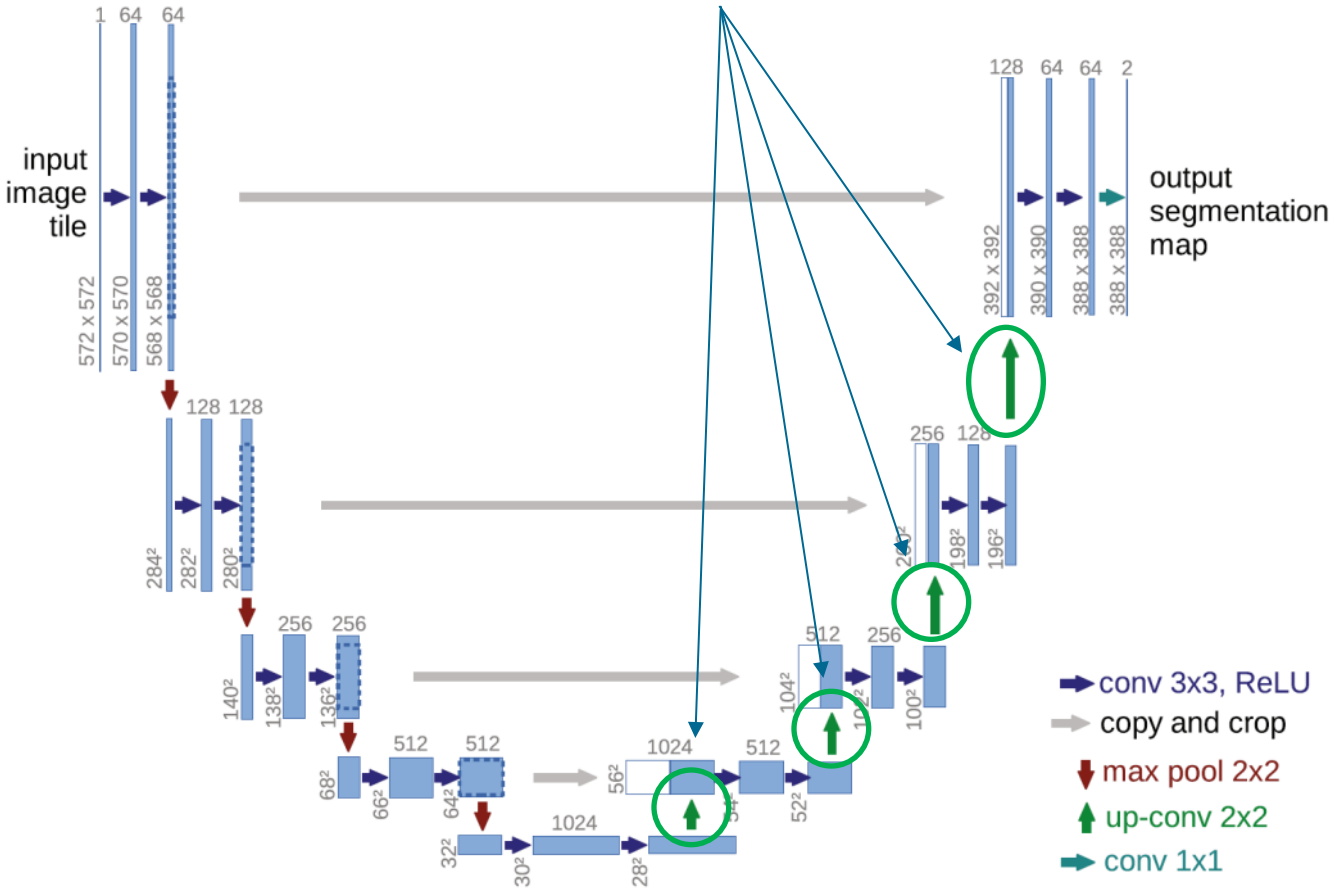


VGG-net like network

They use “valid” convolution

U-Net

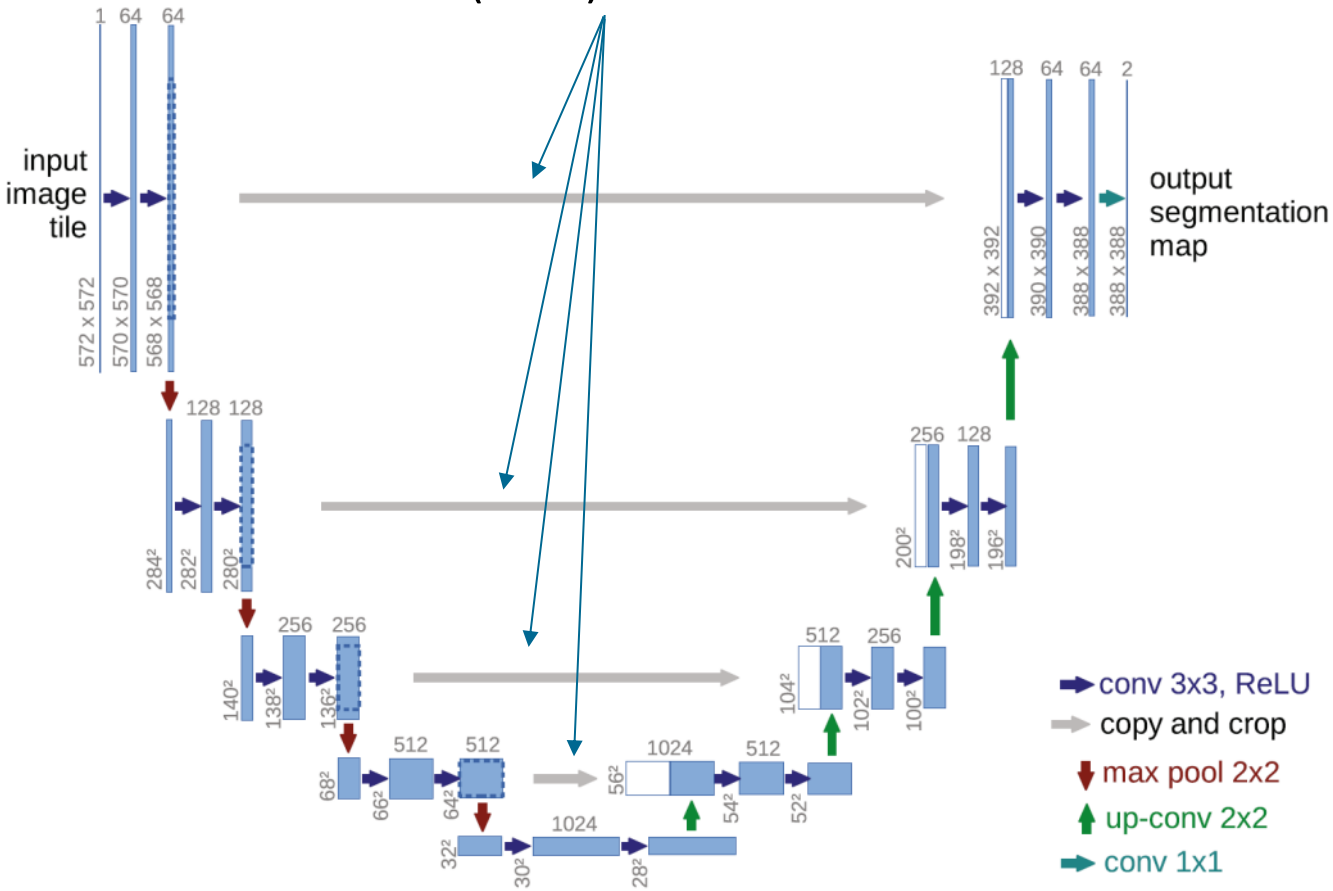
1. **Up-sampling**
2. **2x2 convolution**
   - Halves the number of feature channels



U-Net

Concatenation

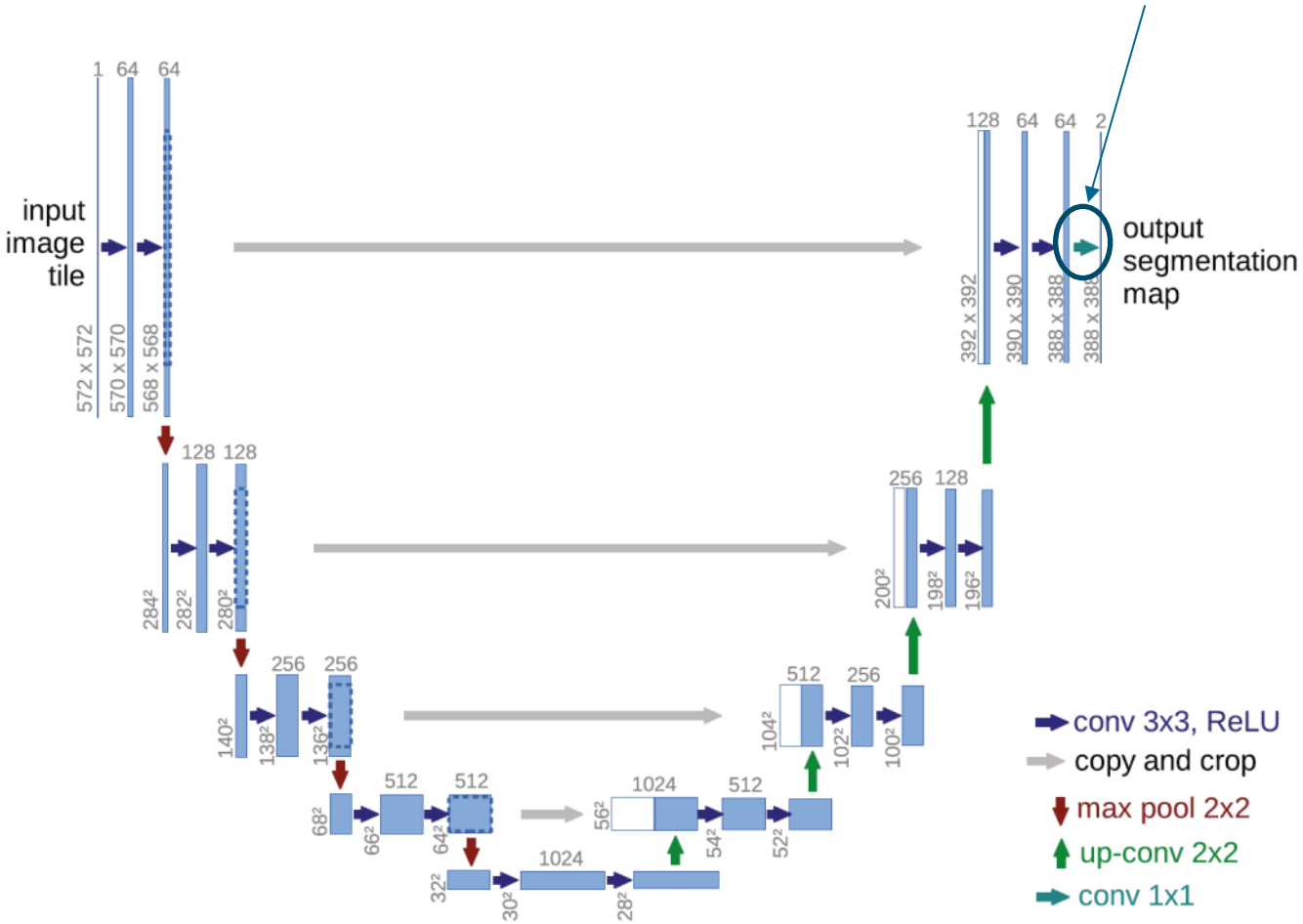
- Adds (back) *details to context*



U-Net

1x1 convolution

- From 64 features to 2 (output)



U-Net

- Examples of results

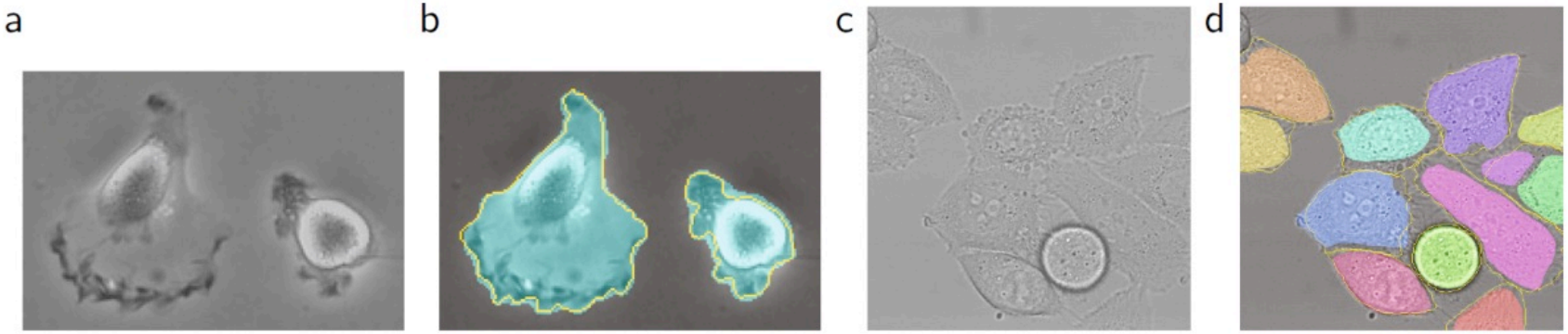


Fig. 4. Result on the ISBI cell tracking challenge. (**a**) part of an input image of the "PhC-U373" data set. (**b**) Segmentation result (cyan mask) with manual ground truth (yellow border) (**c**) input image of the "DIC-HeLa" data set. (**d**) Segmentation result (random colored masks) with manual ground truth (yellow border).

U-Net

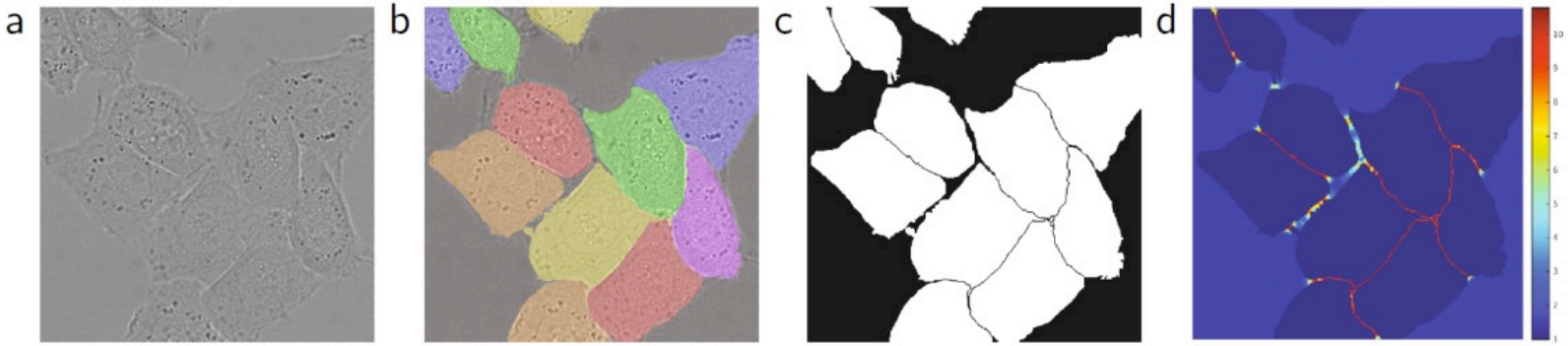


Fig. 3. HeLa cells on glass recorded with DIC (differential interference contrast) microscopy. (**a**) raw image. (**b**) overlay with ground truth segmentation. Different colors indicate different instances of the HeLa cells. (**c**) generated segmentation mask (white: foreground, black: background). (**d**) map with a pixel-wise loss weight to force the network to learn the border pixels.

They used a "trick"

Loss function

- They use cross-entropy loss at pixel level

$$E = \sum_{\mathbf{x} \in \Omega} w(\mathbf{x}) \log(p_{\ell(\mathbf{x})}(\mathbf{x}))$$

- They also use a trick (loss weight) to enforce good segmentation at object borders

$$w(\mathbf{x}) = w_c(\mathbf{x}) + w_0 \cdot \exp\left(-\frac{(d_1(\mathbf{x}) + d_2(\mathbf{x}))^2}{2\sigma^2}\right)$$

U-Net

- It can be trained with little data
- Fast!
- Segmentation of a 512x512 image takes < 1 sec (on GPU)
- In the paper they used heavy **data augmentation**!
  - *Elastic deformation*

VIBOT 2018

Deep Learning with Convolutional Neural Networks

Francesco Ciompi

francesco.ciompi@radboudumc.nl

Radboudumc