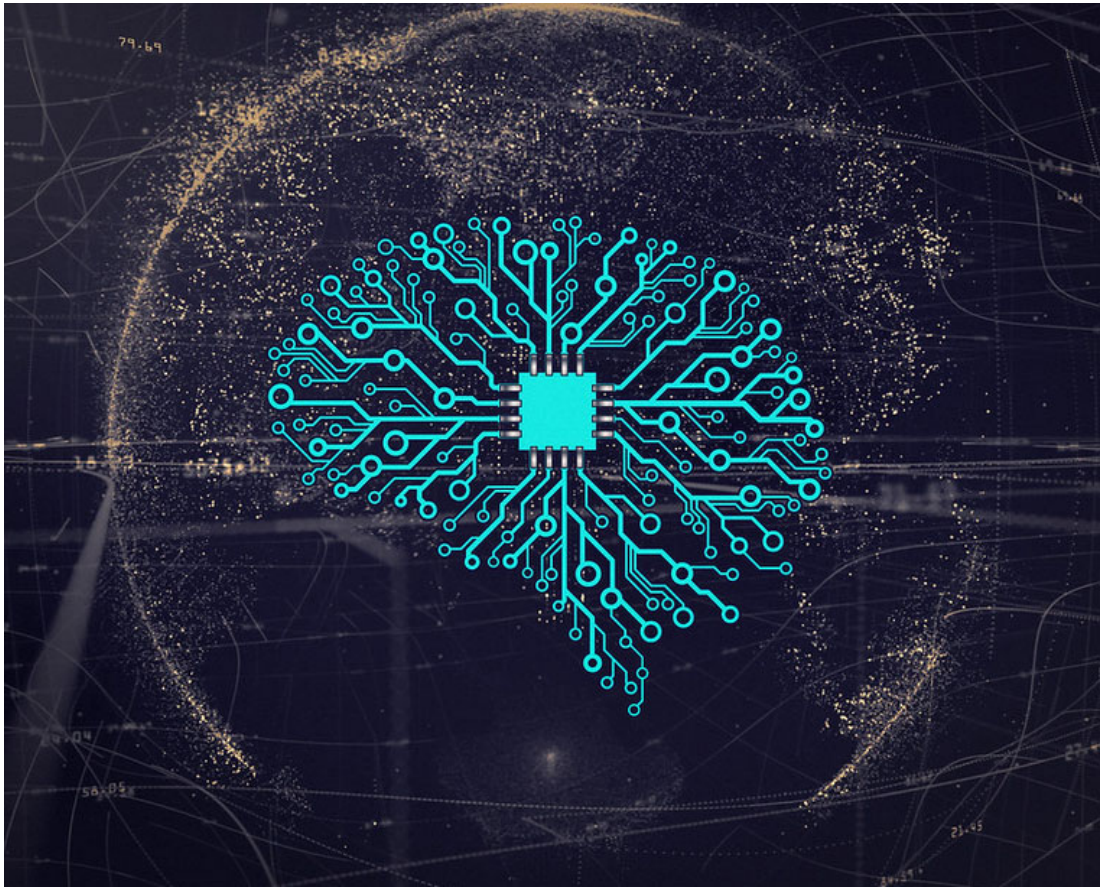
Part I: Intro to NN

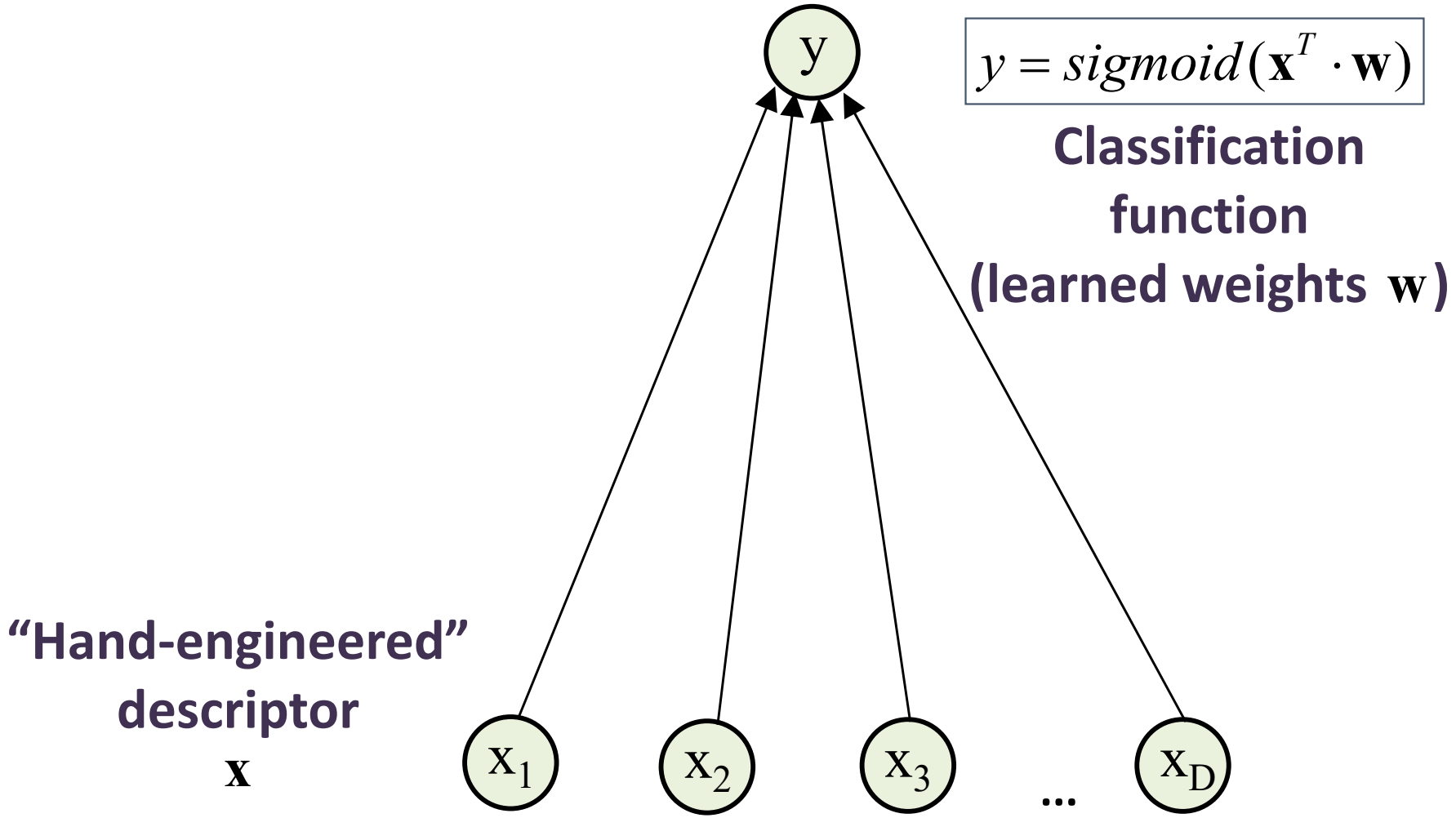


Intelligent Visual Computing

Evangelos Kalogerakis

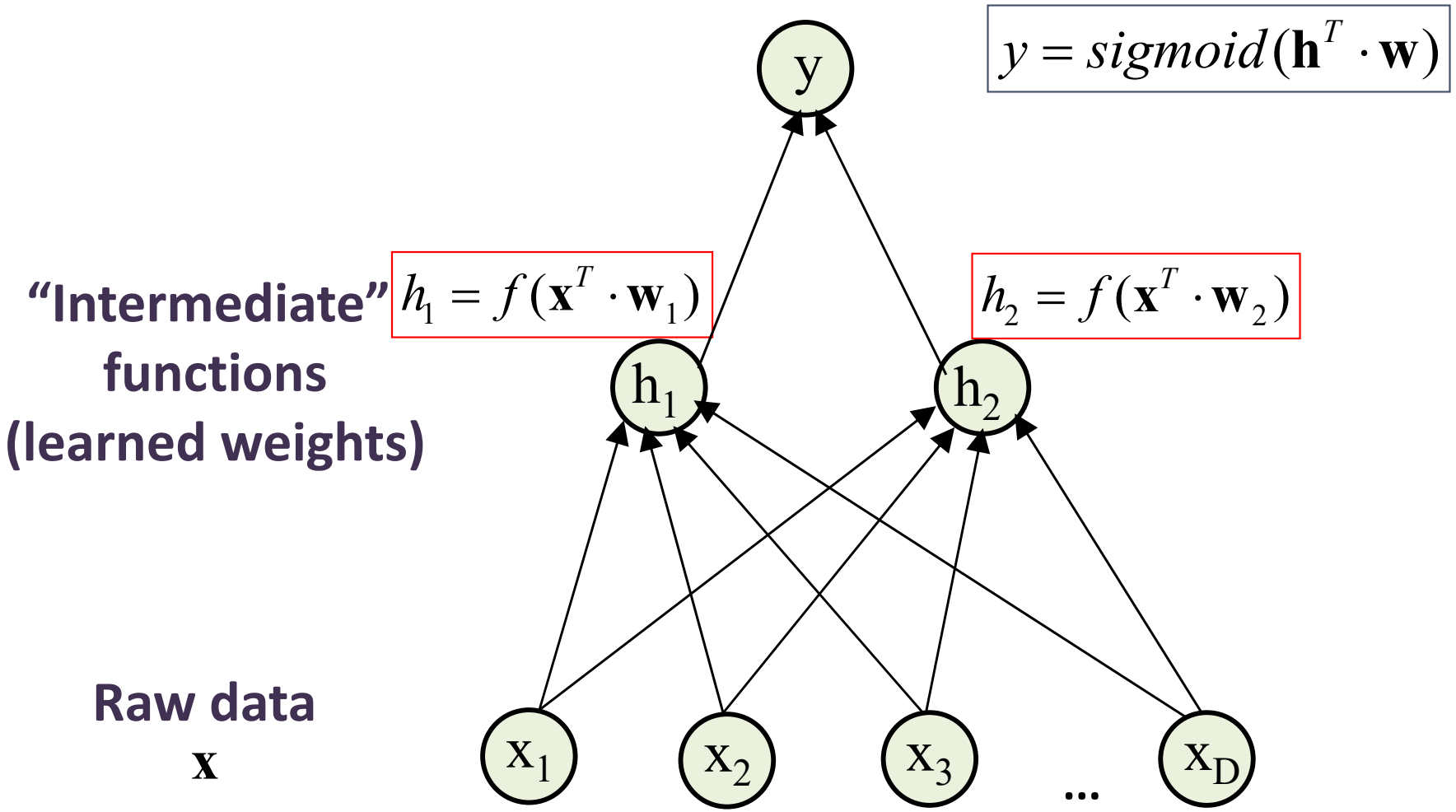
From “shallow” mappings…

Old-style approach: output is a **direct function** of hand-engineered shape descriptors



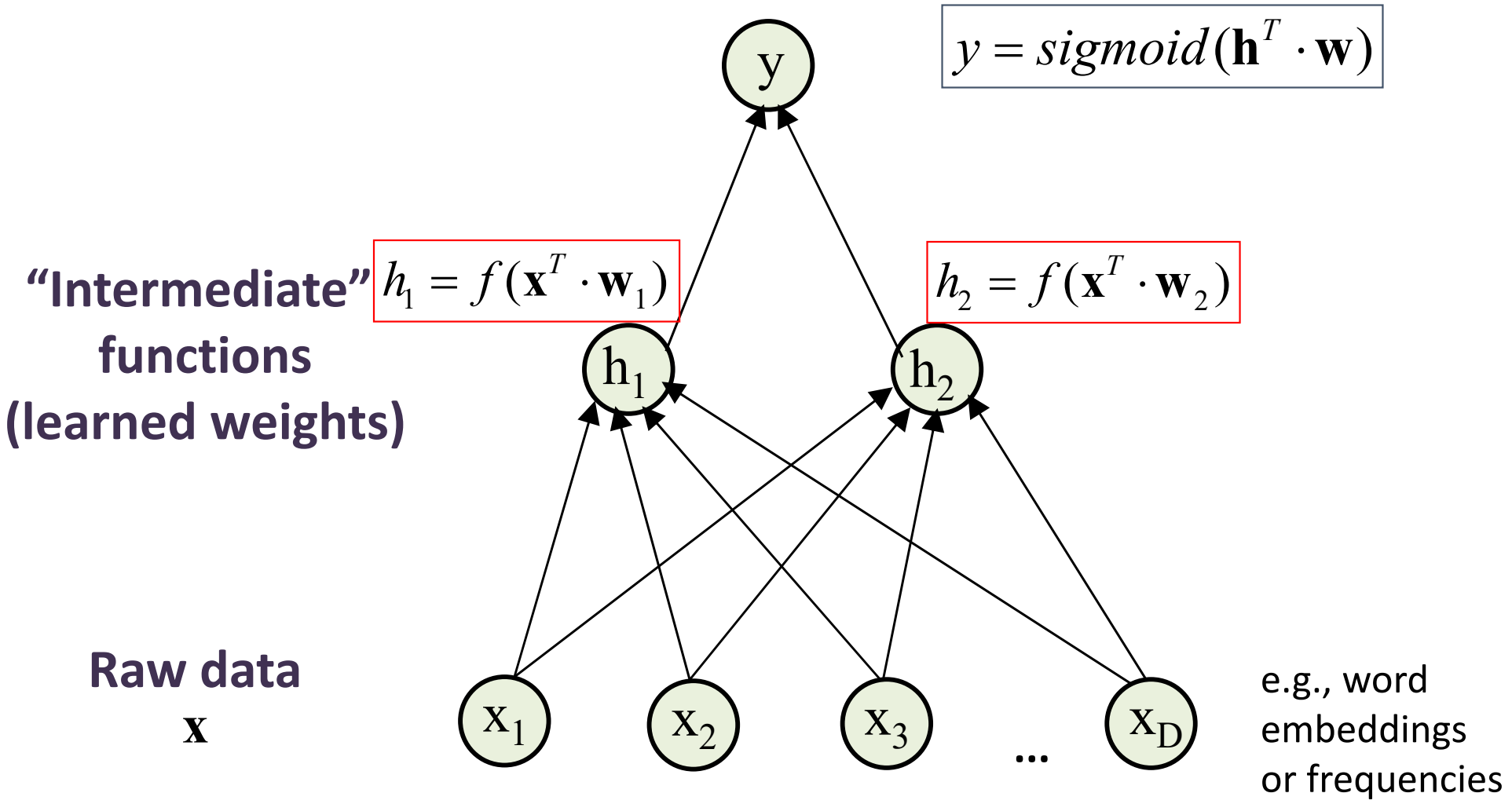
… to neural nets

Introduce **intermediate learned functions** that yield optimized descriptors.



… to neural nets

Introduce **intermediate learned functions** that yield optimized descriptors.



… to neural nets

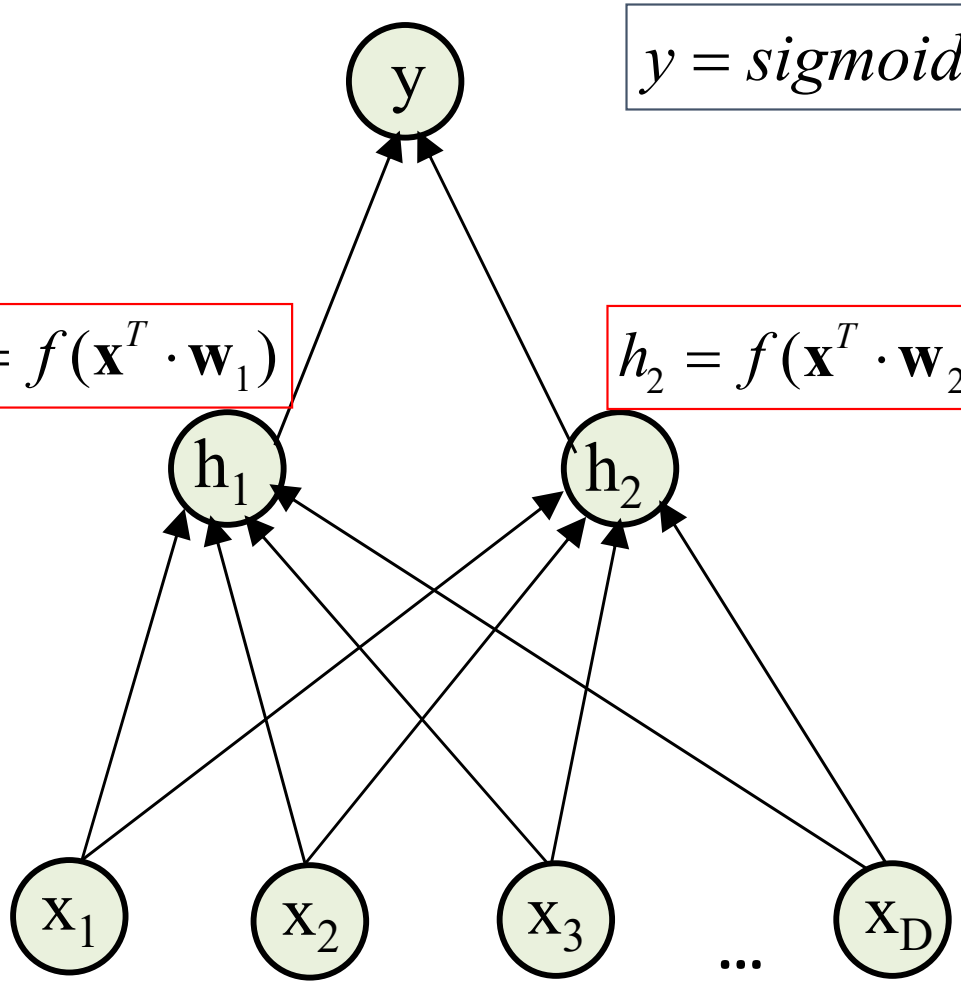
Introduce **intermediate learned functions** that yield optimized descriptors.

$$y = sigmoid(\mathbf{h}^T \cdot \mathbf{w})$$

"Intermediate" functions (learned weights)

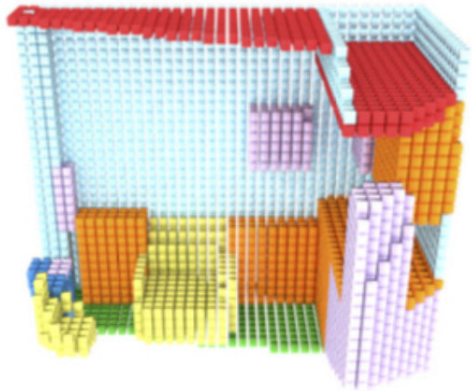
$$h_1 = f(\mathbf{x}^T \cdot \mathbf{w}_1)$$

$$h_2 = f(\mathbf{x}^T \cdot \mathbf{w}_2)$$



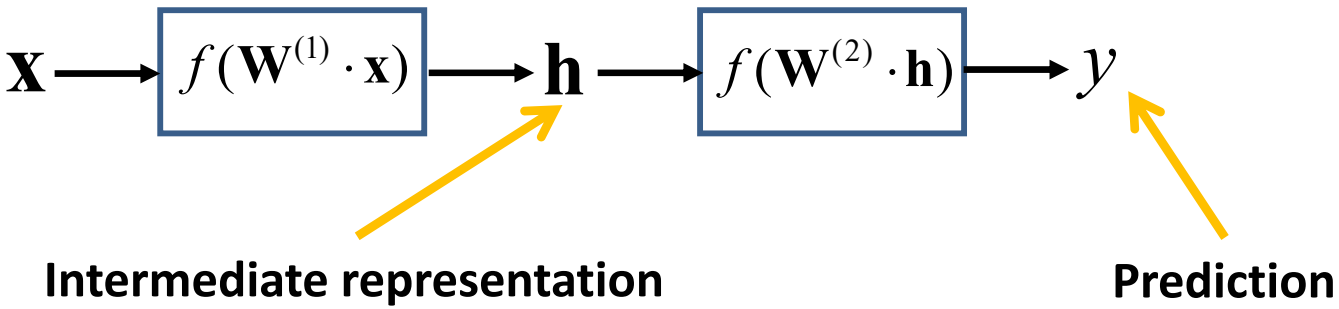
Raw data
x

e.g., voxels occupied or not



Neural network

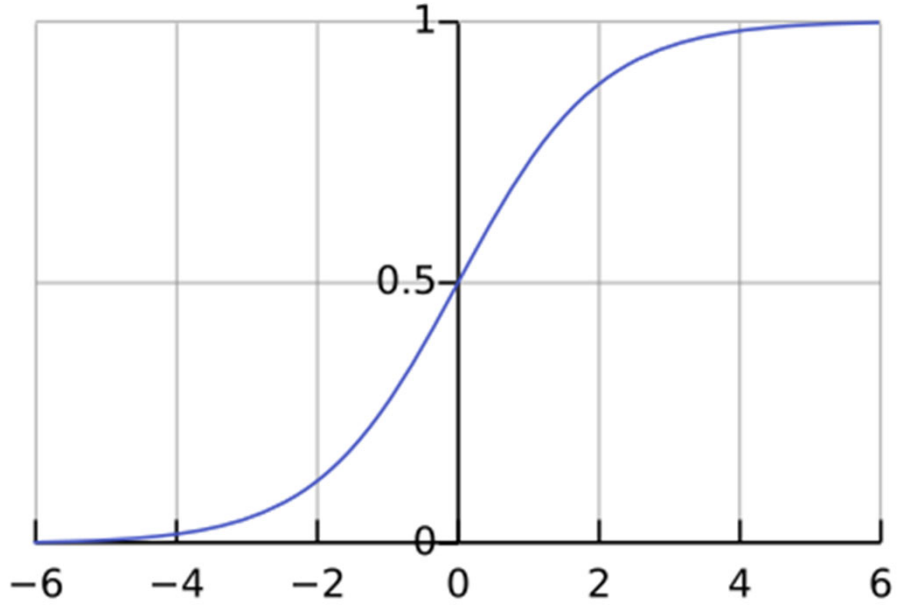
Our output function has **multiple stages** ("layers").



$$where \quad \mathbf{W}^{(\cdot)} = \begin{bmatrix} \mathbf{w}_1^{(\cdot)} \\ \mathbf{w}_2^{(\cdot)} \\ \cdots \\ \mathbf{w}_M^{(\cdot)} \end{bmatrix}$$

modified slides originally by Adam Coates

Activation functions



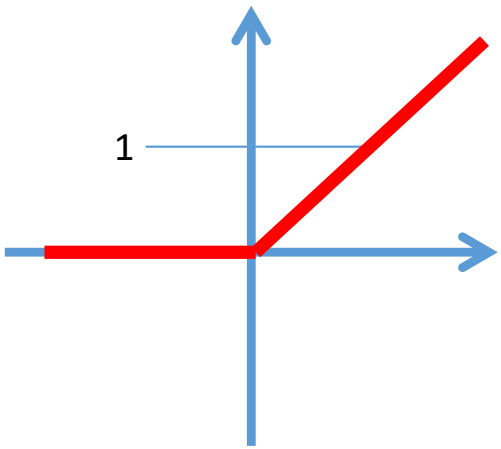
$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$$

Derivative: $\sigma'(x) = \sigma(x)[1 - \sigma(x)]$

Activation functions

("*f*" in the previous slides)

ReLu

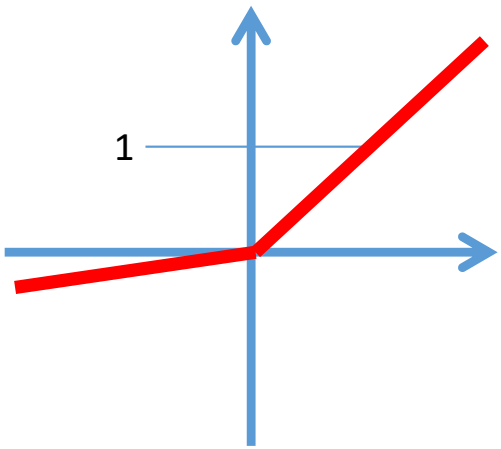


("dies" at negative input)

Derivative:

$$ReLu'(x) = [\, x > 0 \,]$$

Leaky ReLu



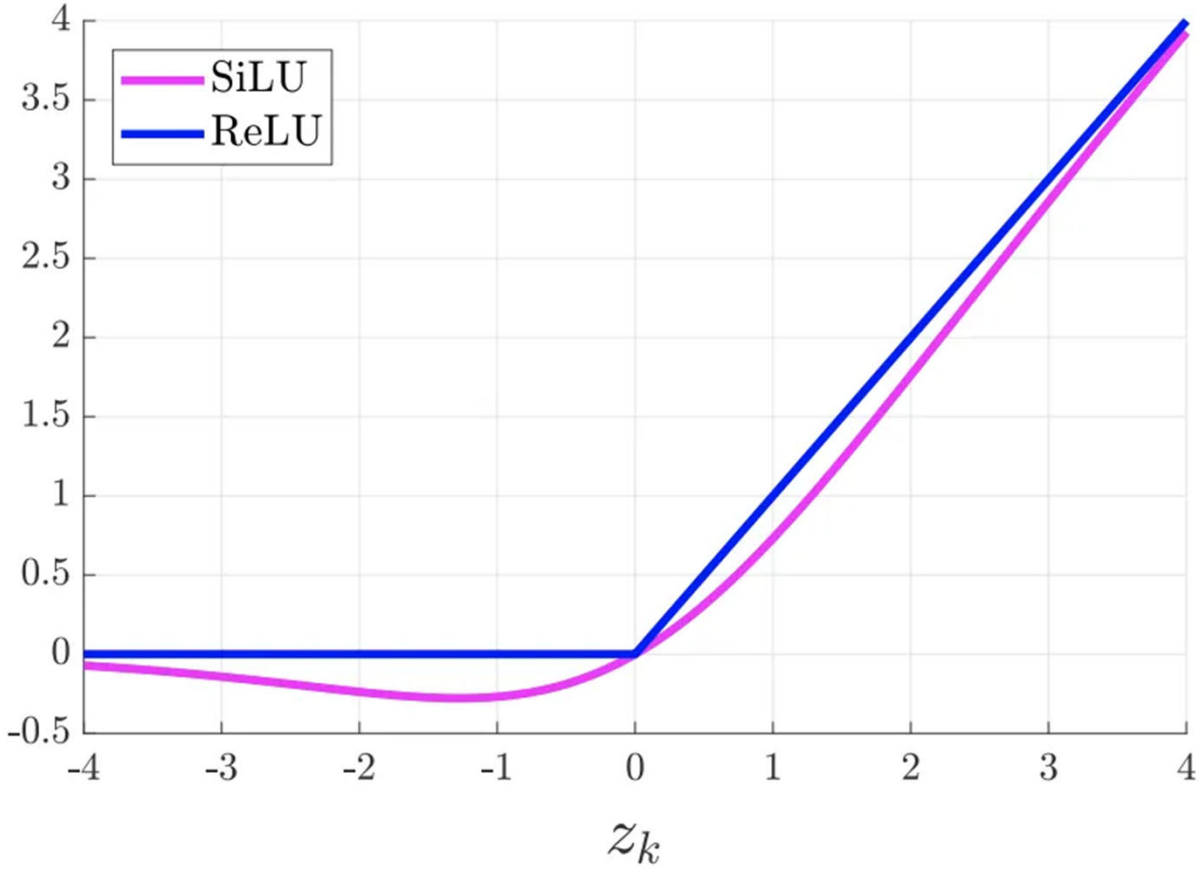
Derivative:

$$LReLu'(x) = a[\, x < 0 \,] + [\, x > 0 \,]$$

Activation functions

("*f*" in the previous slides)

SiLu



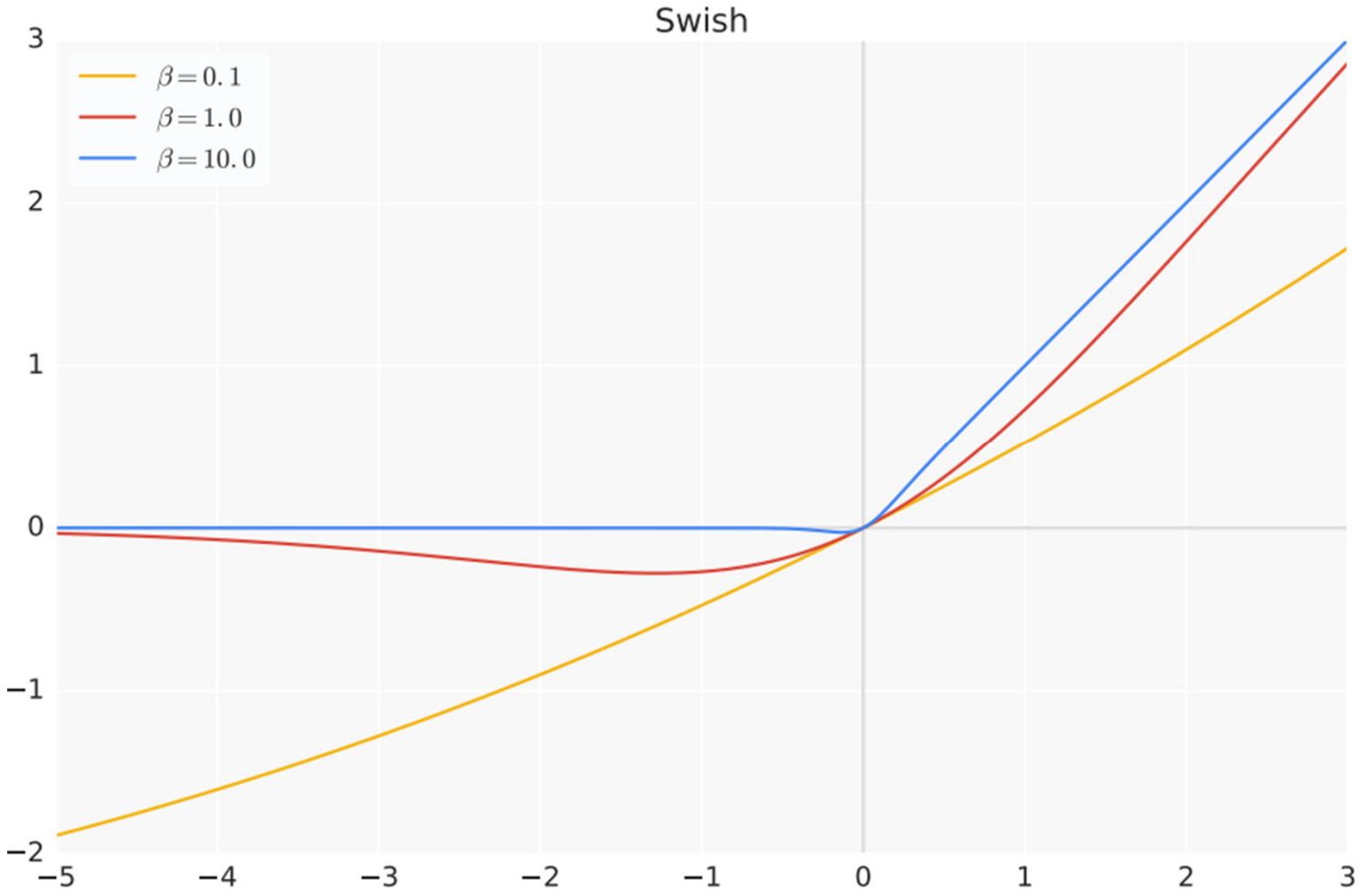
$$f(x) = \frac{x}{1+e^{-x}}$$

$$f'(x) = ?$$

Activation functions

("*f*" in the previous slides)

"swish"

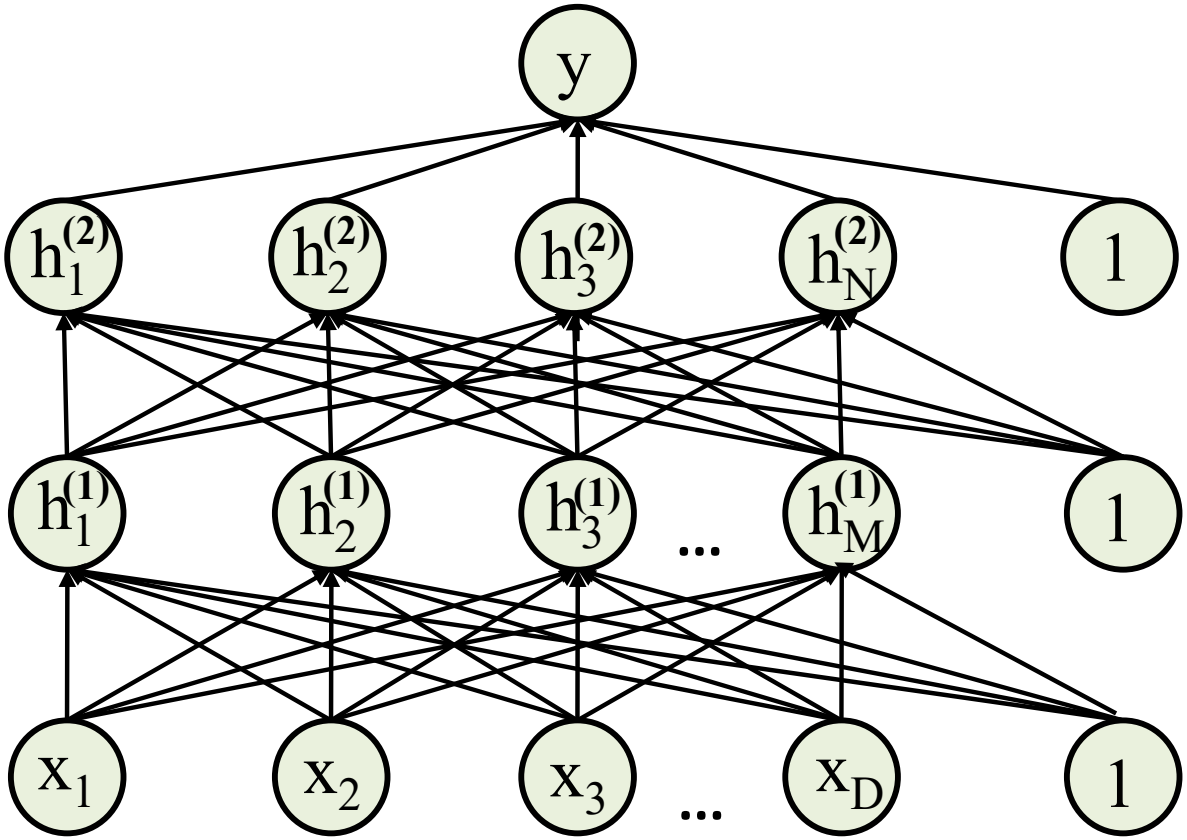


$$f(x,b) = \frac{x}{1+e^{-bx}}$$

https://en.wikipedia.org/wiki/Activation_function

Deep Neural Network (fully connected)

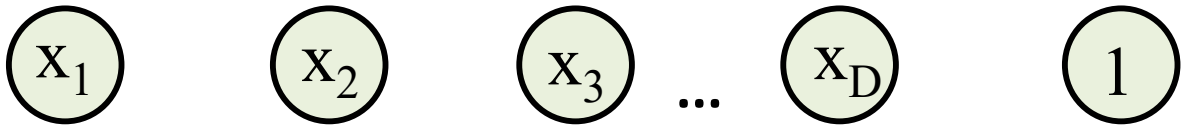
Stack the intermediate functions **in many layers.**



Fully Connected Layers

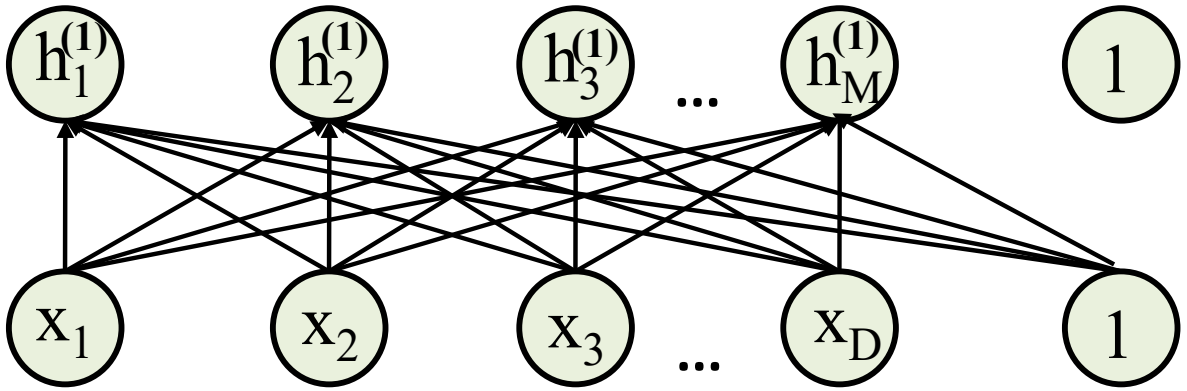
Forward propagation

Process to compute output:



Forward propagation

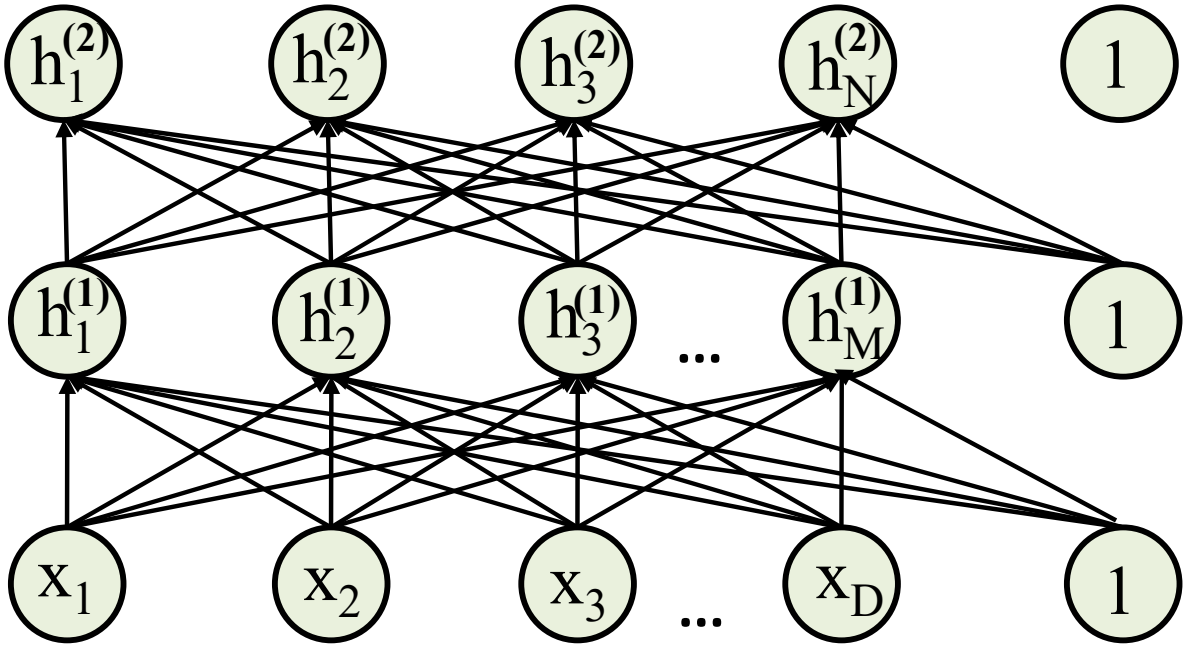
Process to compute output:



$$\mathbf{x} \longrightarrow \boxed{f(\mathbf{W}^{(1)} \cdot \mathbf{x})} \longrightarrow \mathbf{h}^{(1)}$$

Forward propagation

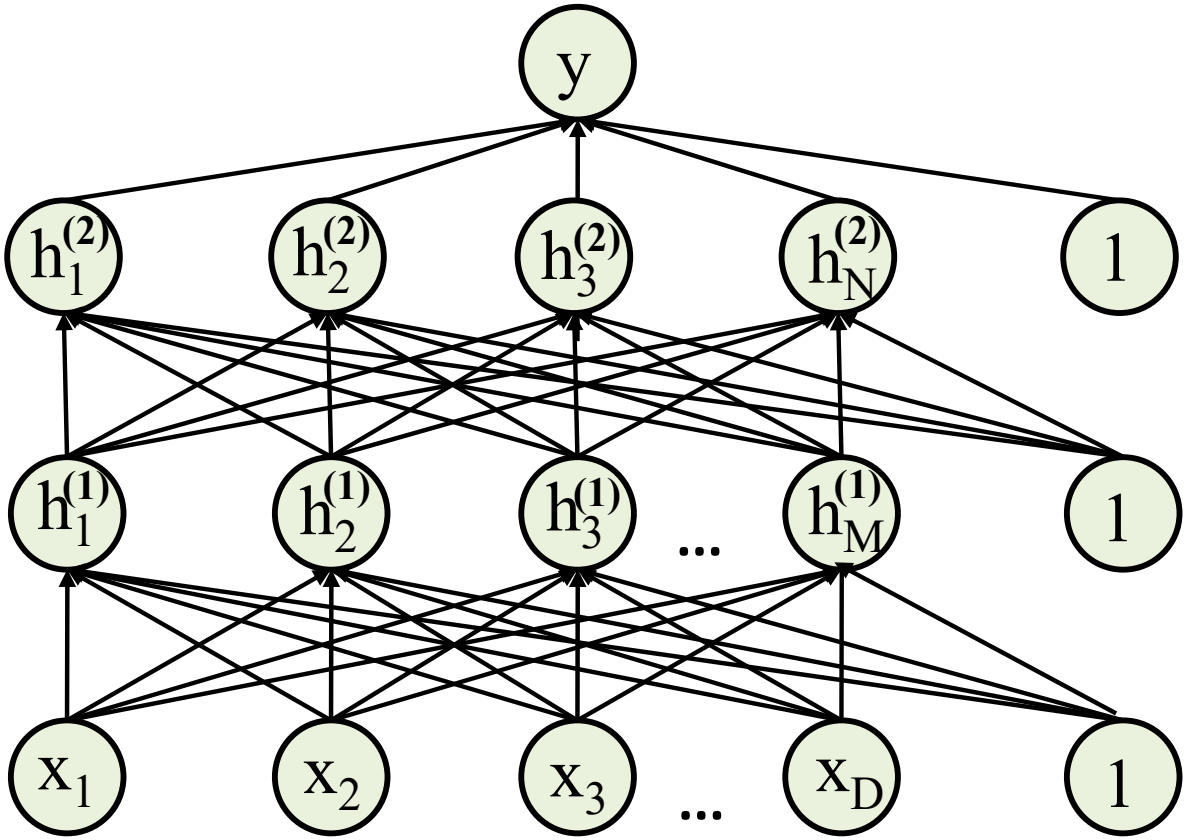
Process to compute output:



$$\mathbf{x} \longrightarrow f(\mathbf{W}^{(1)} \cdot \mathbf{x}) \longrightarrow \mathbf{h}^{(1)} \longrightarrow f(\mathbf{W}^{(2)} \cdot \mathbf{h}^{(1)}) \longrightarrow \mathbf{h}^{(2)}$$

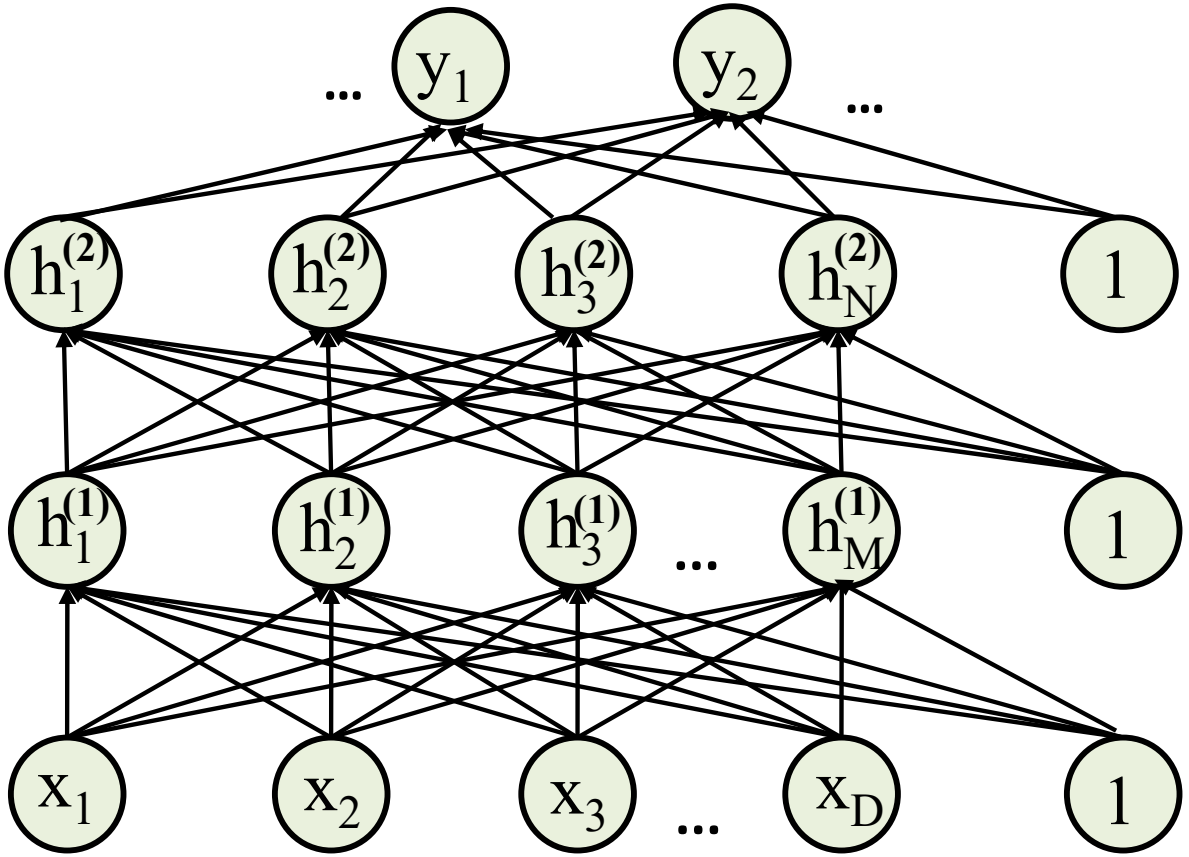
Forward propagation

Process to compute output:



$$\mathbf{x} \rightarrow f(\mathbf{W}^{(1)} \cdot \mathbf{x}) \rightarrow \mathbf{h}^{(1)} \rightarrow f(\mathbf{W}^{(2)} \cdot \mathbf{h}^{(1)}) \rightarrow \mathbf{h}^{(2)} \rightarrow f(\mathbf{W}^{(3)} \cdot \mathbf{h}^{(2)}) \rightarrow y$$

Multiple outputs



$$\mathbf{x} \rightarrow f(\mathbf{W}^{(1)} \cdot \mathbf{x}) \rightarrow \mathbf{h}^{(1)} \rightarrow f(\mathbf{W}^{(2)} \cdot \mathbf{h}^{(1)}) \rightarrow \mathbf{h}^{(2)} \rightarrow f(\mathbf{W}^{(3)} \cdot \mathbf{h}^{(2)}) \rightarrow \mathbf{y}$$

How to learn the parameters?

Use a loss function e.g., for binary classification:

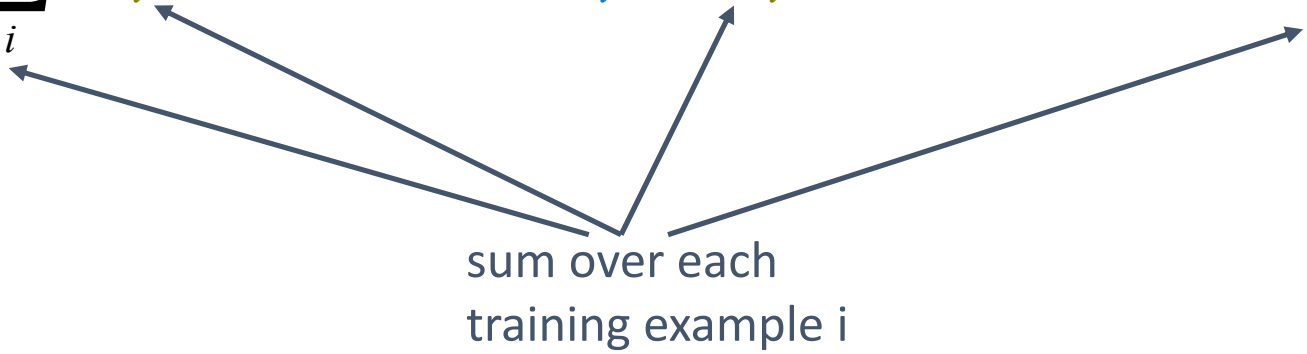
$$L(\mathbf{W}) = -\sum_i [y_i^{(gt)} == 1] \log f(\mathbf{x}_i) + [y_i^{(gt)} == 0] \log(1 - f(\mathbf{x}_i))$$

(note: I use **negative** log-likelihood here)

How to learn the parameters?

Use a loss function e.g., for binary classification:

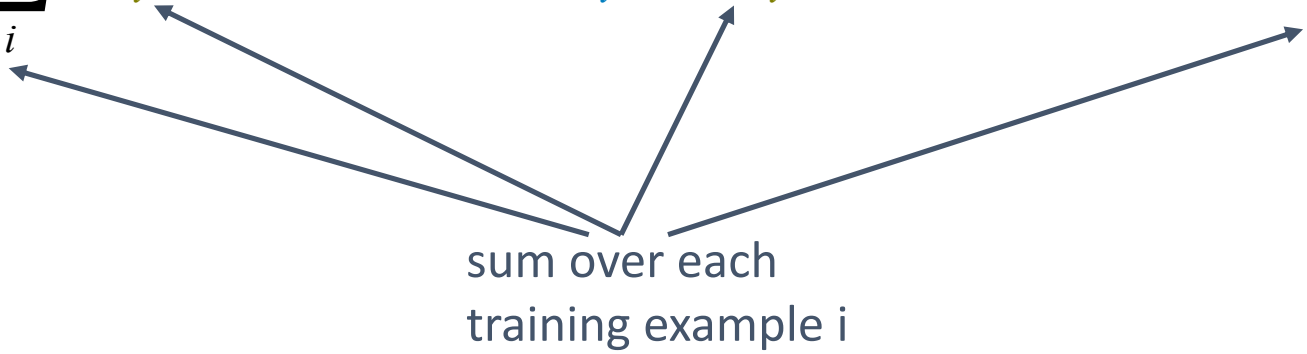
$$L(\mathbf{W}) = -\sum_i [y_i^{(gt)} == 1] \log f(\mathbf{x}_i) + [y_i^{(gt)} == 0] \log(1 - f(\mathbf{x}_i))$$



How to learn the parameters?

Use a loss function e.g., for binary classification:

$$L(\mathbf{W}) = -\sum_i [y_i^{(gt)} == 1] \log f(\mathbf{x}_i) + [y_i^{(gt)} == 0] \log(1 - f(\mathbf{x}_i))$$



How to learn the parameters?

Use a loss function e.g., for binary classification:

$$L(\mathbf{W}) = -\sum_i [y_i^{(gt)} == 1] \log f(\mathbf{x}_i) + [y_i^{(gt)} == 0] \log(1 - f(\mathbf{x}_i))$$

Need gradient wrt param. **$\mathbf{W}=\{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}...\}$** of all layers!

How to learn the parameters?

Use a loss function e.g., for binary classification:

$$L(\mathbf{W}) = -\sum_i [y_i^{(gt)} == 1] \log f(\mathbf{x}_i) + [y_i^{(gt)} == 0] \log(1 - f(\mathbf{x}_i))$$

Need gradient wrt param. **$\mathbf{W}=\{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}...\}$** of all layers!

Our neural network implements a **composite function**:

How to learn the parameters?

Use a loss function e.g., for binary classification:

$$L(\mathbf{W}) = -\sum_i [y_i^{(gt)} == 1] \log f(\mathbf{x}_i) + [y_i^{(gt)} == 0] \log(1 - f(\mathbf{x}_i))$$

Need gradient wrt param. **$\mathbf{W}=\{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}...\}$** of all layers!

Our neural network implements a **composite function**:

$$f^{(1)}(\mathbf{x}_i, \mathbf{W}^{(1)})$$

first layer transforms input with parameters **$\mathbf{W}^{(1)}$**

How to learn the parameters?

Use a loss function e.g., for binary classification:

$$L(\mathbf{W}) = -\sum_i [y_i^{(gt)} == 1] \log f(\mathbf{x}_i) + [y_i^{(gt)} == 0] \log(1 - f(\mathbf{x}_i))$$

Need gradient wrt param. **$\mathbf{W}=\{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}...\}$** of all layers!

Our neural network implements a **composite function**:

$$f^{(2)}(\quad f^{(1)}(\mathbf{x}_i, \mathbf{W}^{(1)}), \quad \mathbf{W}^{(2)} \quad)$$

second layer transforms the result with parameters $\mathbf{W}^{(2)}$

How to learn the parameters?

Use a loss function e.g., for binary classification:

$$L(\mathbf{W}) = -\sum_i [y_i^{(gt)} == 1] \log f(\mathbf{x}_i) + [y_i^{(gt)} == 0] \log(1 - f(\mathbf{x}_i))$$

Need gradient wrt param. **$\mathbf{W}=\{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}...\}$** of all layers!

Our neural network implements a **composite function**:

$$f^{(3)}(\quad f^{(2)}(\quad f^{(1)}(\mathbf{x}_i, \mathbf{W}^{(1)}), \quad \mathbf{W}^{(2)} \quad), \quad \mathbf{W}^{(3)} \quad)$$

third layer transforms the result with parameters **$\mathbf{W}^{(3)}$**

How to learn the parameters?

Use a loss function e.g., for binary classification:

$$L(\mathbf{W}) = -\sum_i [y_i^{(gt)} == 1] \log f(\mathbf{x}_i) + [y_i^{(gt)} == 0] \log(1 - f(\mathbf{x}_i))$$

Need gradient wrt param. **$\mathbf{W}=\{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}...\}$** of all layers!

Our neural network implements a **composite function**:

$$L(\ f^{(3)}(\ \ f^{(2)}(\ \ f^{(1)}(\mathbf{x}_i, \mathbf{W}^{(1)}),\ \ \mathbf{W}^{(2)}\ \ ),\ \ \mathbf{W}^{(3)}\ \ )\ \ )$$

loss compares output to "ground-truth" signal -
this is another function "sitting on top of" the output

How to learn the parameters?

Use a loss function e.g., for binary classification:

$$L(\mathbf{W}) = -\sum_i [y_i^{(gt)} == 1] \log f(\mathbf{x}_i) + [y_i^{(gt)} == 0] \log(1 - f(\mathbf{x}_i))$$

Need gradient wrt param. **$\mathbf{W}=\{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}...\}$** of all layers!

Our neural network implements a **composite function**:

$$L(\ f^{(3)}(\ \ f^{(2)}(\ \ f^{(1)}(\mathbf{x})\ )\ )\ )$$

... omitting notation clutter ....

How to learn the parameters?

Use a loss function e.g., for binary classification:

$$L(\mathbf{W}) = -\sum_i [y_i^{(gt)} == 1] \log f(\mathbf{x}_i) + [y_i^{(gt)} == 0] \log(1 - f(\mathbf{x}_i))$$

Need gradient wrt param. **$\mathbf{W}=\{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}...\}$** of all layers!

Anybody remembering the **chain rule**?

$$\frac{\partial L}{\partial \mathbf{x}} = ...$$

... omitting notation clutter ....

How to learn the parameters?

Use a loss function e.g., for binary classification:

$$L(\mathbf{W}) = -\sum_i [y_i^{(gt)} == 1] \log f(\mathbf{x}_i) + [y_i^{(gt)} == 0] \log(1 - f(\mathbf{x}_i))$$

Need gradient wrt param. **$\mathbf{W}=\{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}...\}$** of all layers!

Anybody remembering the **chain rule**?

$$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial f^{(3)}} \cdot \frac{\partial f^{(3)}}{\partial f^{(2)}} \cdot \frac{\partial f^{(2)}}{\partial f^{(1)}} \cdot \frac{\partial f^{(1)}}{\partial \mathbf{x}}$$

How to learn the parameters?

Use a loss function e.g., for binary classification:

$$L(\mathbf{W}) = -\sum_i [y_i^{(gt)} == 1] \log f(\mathbf{x}_i) + [y_i^{(gt)} == 0] \log(1 - f(\mathbf{x}_i))$$

Need gradient wrt param. **$\mathbf{W}=\{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}...\}$** of all layers!

In our case, we need gradients wrt weights...

$$\frac{\partial L}{\partial \mathbf{W}^{(1)}} = \frac{\partial L}{\partial f^{(3)}} \cdot \frac{\partial f^{(3)}}{\partial f^{(2)}} \cdot \frac{\partial f^{(2)}}{\partial f^{(1)}} \cdot \frac{\partial f^{(1)}}{\partial \mathbf{W}^{(1)}}$$

How to learn the parameters?

Use a loss function e.g., for binary classification:

$$L(\mathbf{W}) = -\sum_i [y_i^{(gt)} == 1] \log f(\mathbf{x}_i) + [y_i^{(gt)} == 0] \log(1 - f(\mathbf{x}_i))$$

Need gradient wrt param. **$\mathbf{W}=\{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}...\}$** of all layers!

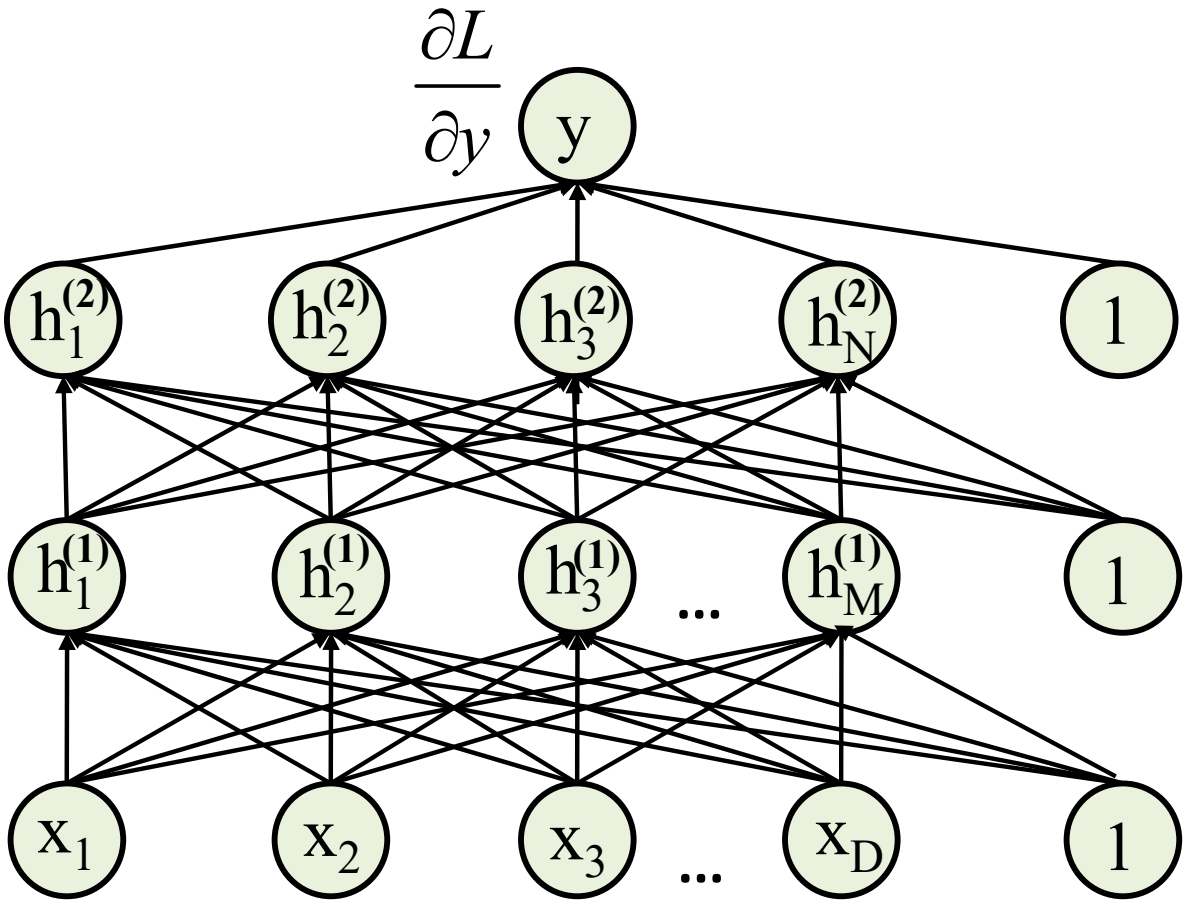
In our case, we need gradients wrt weights...

$$\frac{\partial L}{\partial \mathbf{W}^{(2)}} = \frac{\partial L}{\partial f^{(3)}} \cdot \frac{\partial f^{(3)}}{\partial f^{(2)}} \cdot \frac{\partial f^{(2)}}{\partial \mathbf{W}^{(2)}}$$

... and so on ....

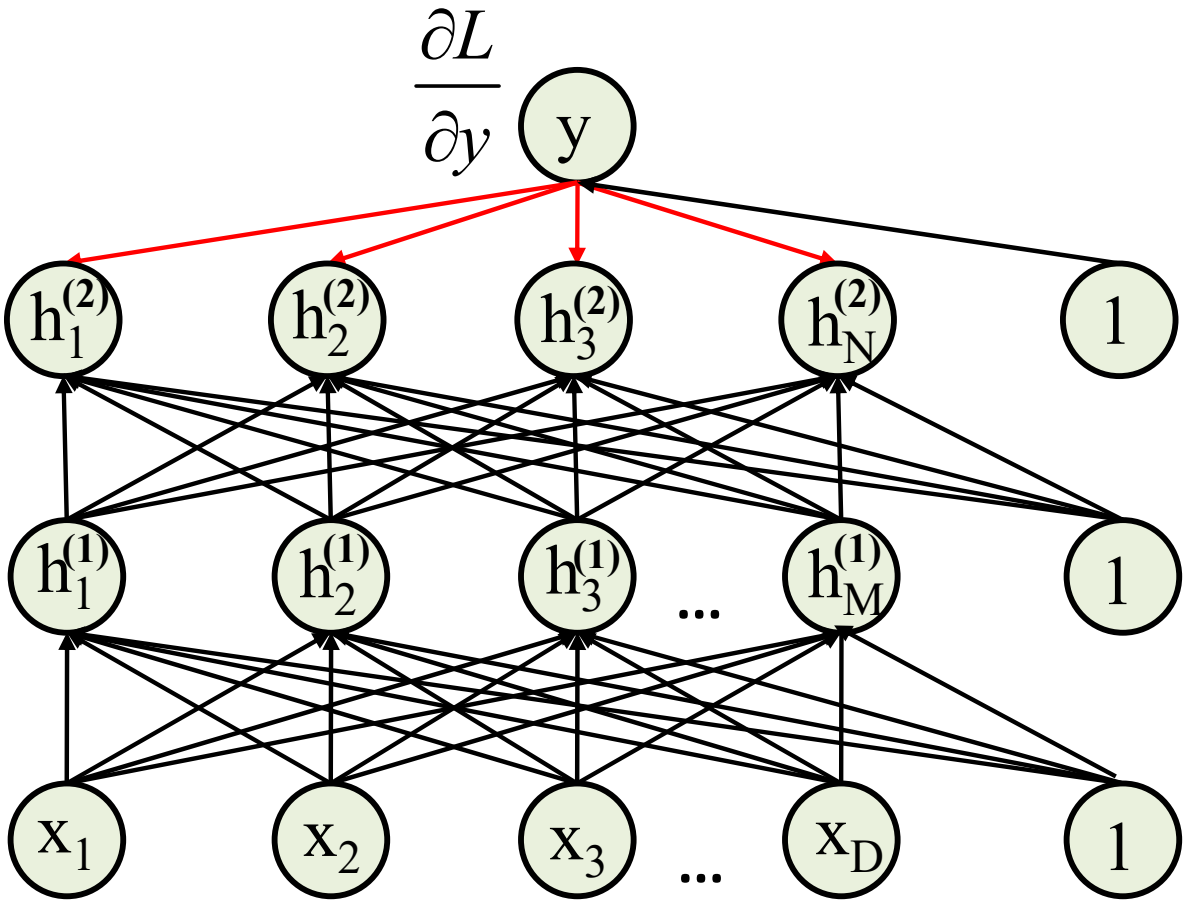
Backpropagation

For each training example i *(in the followings eq., I omit i for clarity)*:



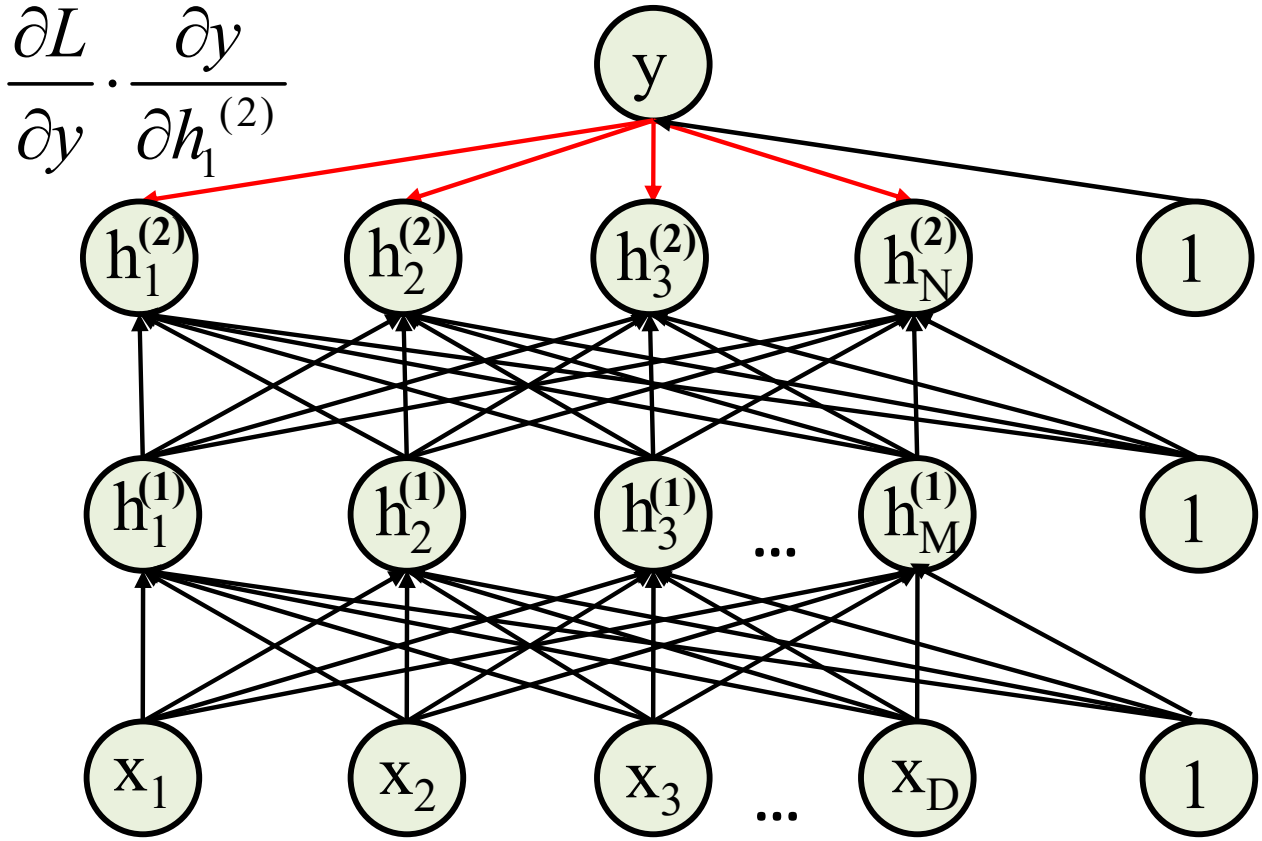
Backpropagation

For each training example i *(in the followings eq., I omit i for clarity)*:



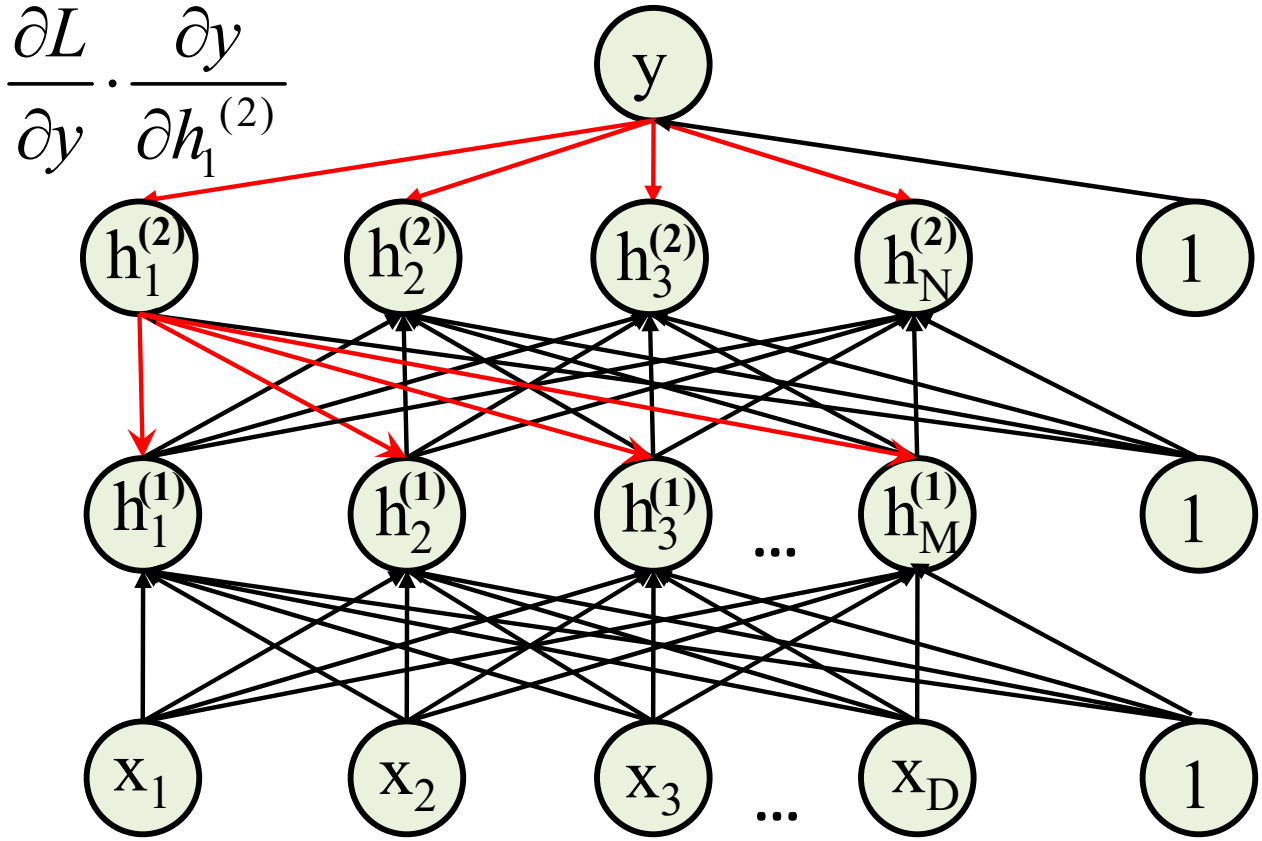
Backpropagation

For each training example i *(in the followings eq., I omit i for clarity)*:



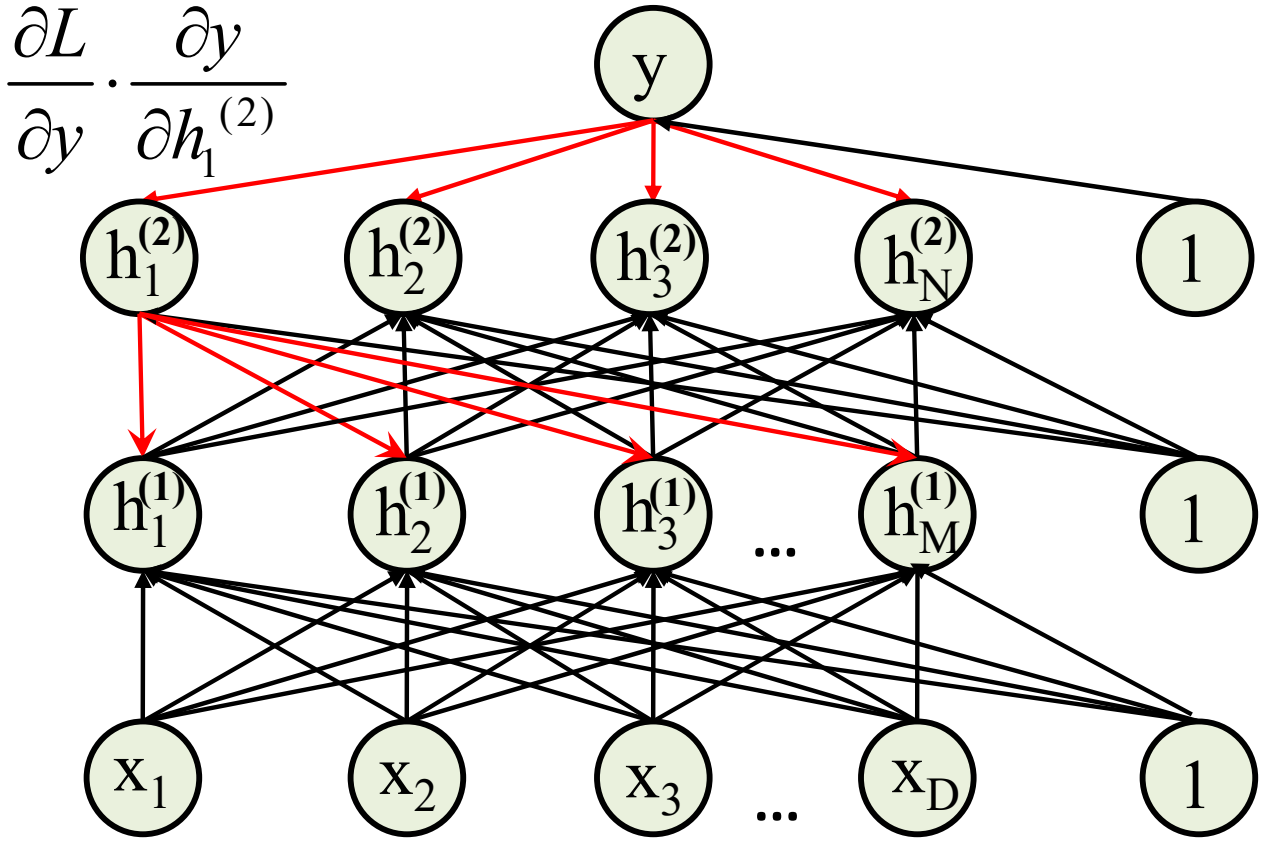
Backpropagation

For each training example i *(in the followings eq., I omit i for clarity)*:



Backpropagation

For each training example i *(in the followings eq., I omit i for clarity)*:



Is this magic?

No, this is nothing more than the **chain rule**!

Suppose a node n in layer l applies a function e.g., ReLU to its input $\boldsymbol{h^{(l-1)}}$:

$$h_n^{(l)} = \mathrm{ReLU}\left(\sum_m w_{n,m}^{(l)} h_m^{(l-1)}\right)$$

Is this magic?

No, this is nothing more than the **chain rule**!

Suppose a node n in layer l applies a function e.g., ReLU to its input $\boldsymbol{h}^{(l-1)}$:

$$h_n^{(l)} = \mathrm{ReLU}\left(\sum_m w_{n,m}^{(l)} h_m^{(l-1)}\right)$$

The derivatives wrt its weights are:

$$\frac{\partial L(\mathbf{w})}{\partial w_{n,m}^{(l)}} = \frac{\partial L(\mathbf{w})}{\partial h_n^{(l)}} \frac{\partial h_n^{(l)}}{\partial w_{n,m}^{(l)}}$$

Is this magic?

No, this is nothing more than the **chain rule**!

Suppose a node n in layer l applies a function e.g., ReLU to its input $\boldsymbol{h}^{(l-1)}$:

$$h_n^{(l)} = \mathrm{ReLU}\left(\sum_m w_{n,m}^{(l)} h_m^{(l-1)}\right)$$

The derivatives wrt its weights are:

$$\frac{\partial L(\mathbf{w})}{\partial w_{n,m}^{(l)}} = \boxed{\frac{\partial L(\mathbf{w})}{\partial h_n^{(l)}}} \frac{\partial h_n^{(l)}}{\partial w_{n,m}^{(l)}}$$

Received "message"

Is this magic?

No, this is nothing more than the **chain rule**!

Suppose a node n in layer l applies a function e.g., ReLU to its input $\boldsymbol{h}^{(l-1)}$:

$$h_n^{(l)} = \mathrm{ReLU}\left(\sum_m w_{n,m}^{(l)} h_m^{(l-1)}\right)$$

The derivatives wrt its weights are:

$$\frac{\partial L(\mathbf{w})}{\partial w_{n,m}^{(l)}} = \frac{\partial L(\mathbf{w})}{\partial h_n^{(l)}} [h_n^{(l)} > 0] h_m^{(l-1)}$$

Received "message"

Remember ReLU derivative!

Is this magic?

No, this is nothing more than the **chain rule**!

Suppose a node n in layer l applies a function e.g., ReLU to its input $\boldsymbol{h^{(l-1)}}$:

$$h_n^{(l)} = \mathrm{ReLU}\left(\sum_m w_{n,m}^{(l)} h_m^{(l-1)}\right)$$

The derivatives wrt its weights are:

$$\frac{\partial L(\mathbf{w})}{\partial w_{n,m}^{(l)}} = \frac{\partial L(\mathbf{w})}{\partial h_n^{(l)}} [h_n^{(l)} > 0] h_m^{(l-1)}$$

Remember ReLU derivative!

Received "message":

$$\frac{\partial L(\mathbf{w})}{\partial h_n^{(l)}} = \sum_t \frac{\partial L(\mathbf{w})}{\partial h_t^{(l+1)}} \frac{\partial h_t^{(l+1)}}{\partial h_n^{(l)}}$$

Training algorithm

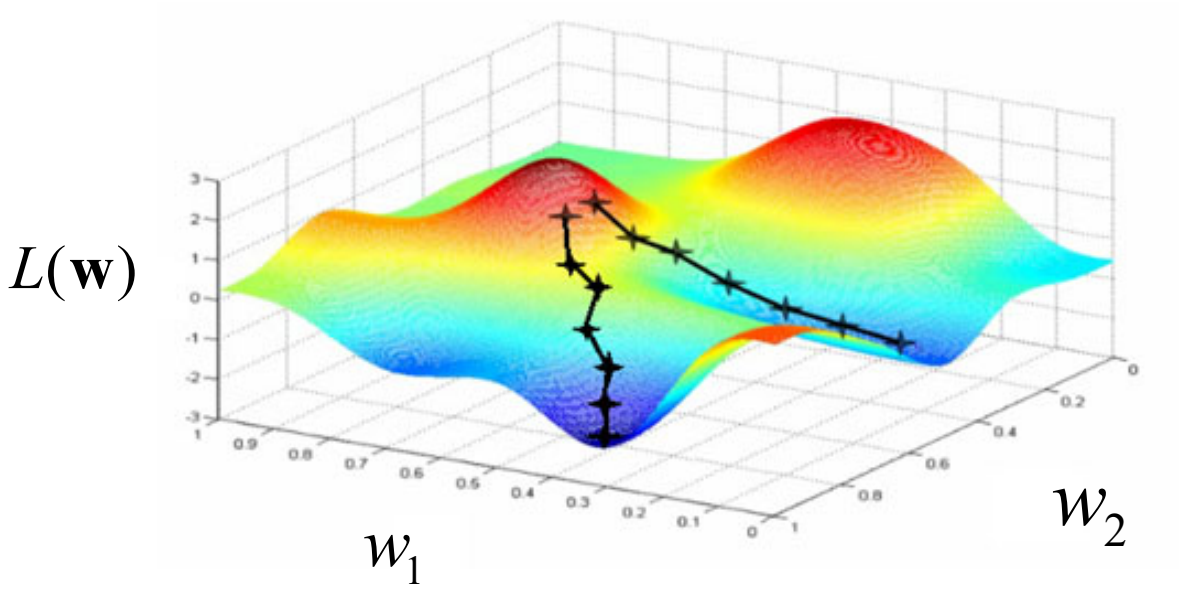
For each training example

1. **Forward propagation** to compute outputs per layer
2. **Back propagate** messages from top to bottom layer
3. Compute **derivatives wrt layer parameters**
4. **Accumulate the derivatives** from that training example

Apply the gradient descent rule

Gradient Descent

$$\mathbf{w}_{new} = \mathbf{w}_{old} - \eta \nabla_{\mathbf{w}} L(\mathbf{w}) =$$
$$= \mathbf{w}_{old} - \eta \sum_i \nabla_{\mathbf{w}} L_i(\mathbf{w})$$



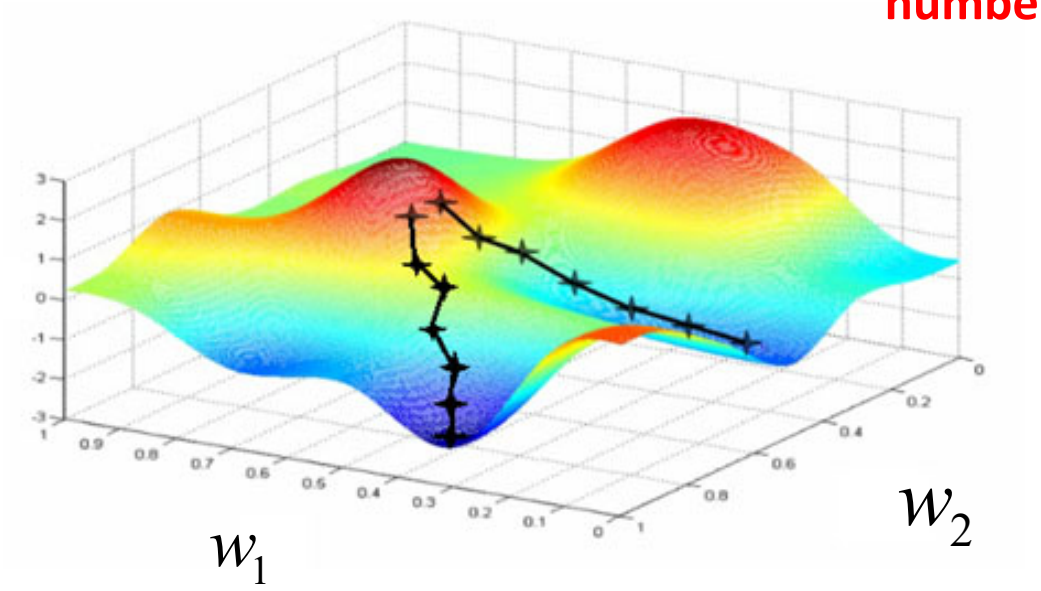
Lots and lots of local minima in neural networks!

Gradient Descent

$$\mathbf{w}_{new} = \mathbf{w}_{old} - \frac{\eta}{K}\sum_i \nabla_{\mathbf{w}} L_i(\mathbf{w})$$

(we often divide the loss with the #training examples K so that the cost function doesn't depend on the number of training examples)

$L(\mathbf{w})$



Lots and lots of local minima in neural networks!

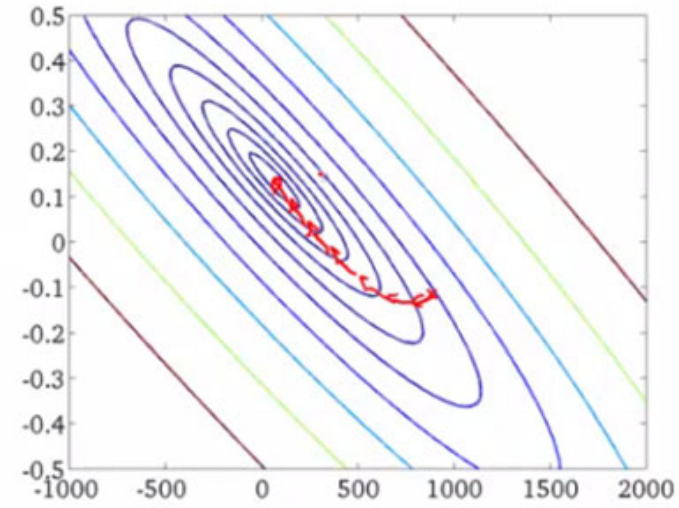
Stochastic Gradient Descent

At each weight update, don't compute gradient from all training examples, but **update it from one example** e.g.

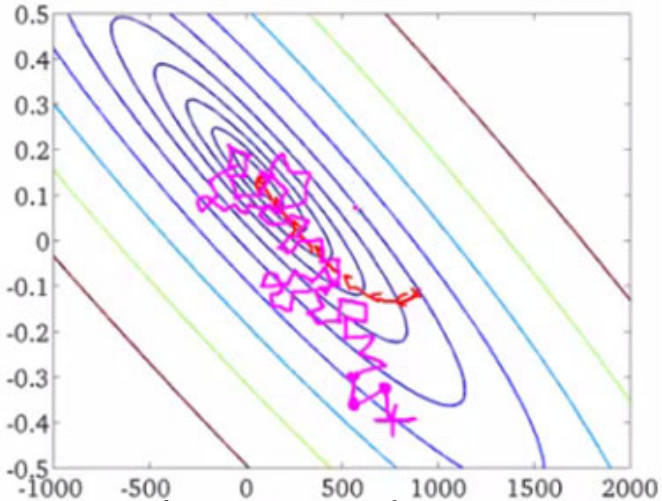
$$\mathbf{w}_{new} = \mathbf{w}_{old} - \eta \nabla_{\mathbf{w}} L_i(\mathbf{w})$$

Advantages:

- memory (do not need to keep the dataset in the mem)
- noisier gradient "jerk" the model out of local minima



Standard Gradient Descent



Stochastic Gradient Descent

Minibatch Gradient Descent

Minibatch gradient descent (computationally more efficient):

- split dataset into minibatches
- at each iteration update weights from one minibatch (average gradient over the examples of the minibatch)

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = \frac{1}{|R|} \sum_{i \in R} \nabla_{\mathbf{w}} L_i(\mathbf{w})$$

R is a random subset of training examples

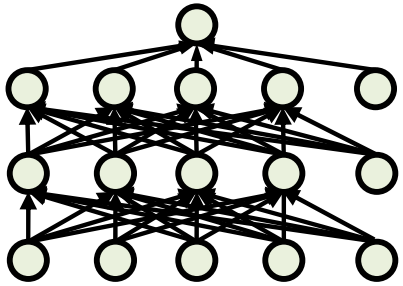
Minibatch Gradient Descent

Minibatch gradient descent (computationally more efficient):

- split dataset into minibatches
- at each iteration update weights from one minibatch (average gradient over the examples of the minibatch)

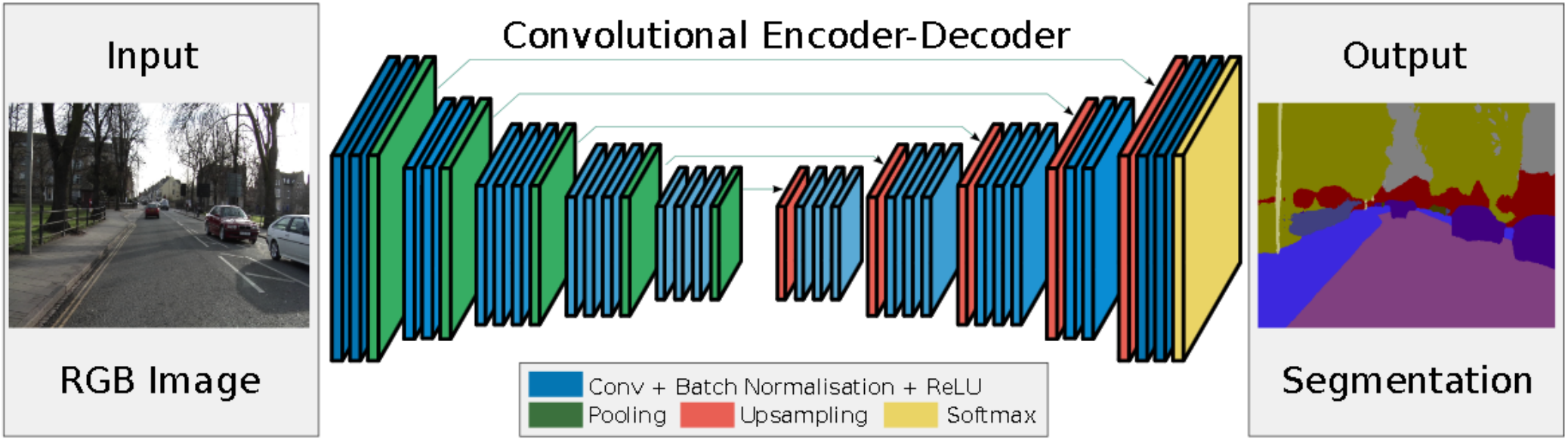
$$\nabla_{\mathbf{w}} L(\mathbf{w}) = \frac{1}{|R|} \sum_{i \in R} \nabla_{\mathbf{w}} L_i(\mathbf{w})$$

R is a random subset of training examples

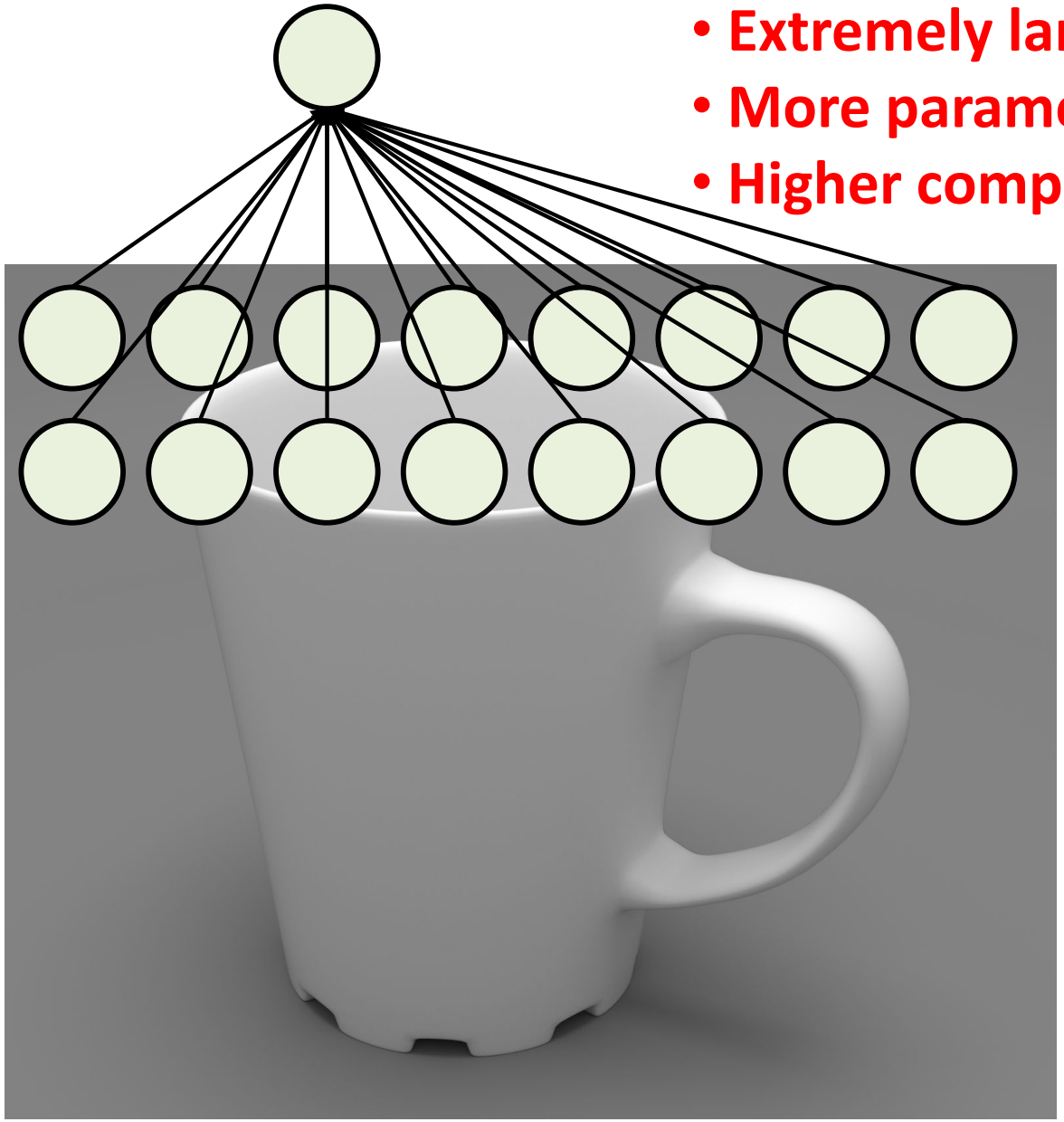


Still, fully connected networks can have a terrible number of parameters to optimize!

Part III: ConvNets

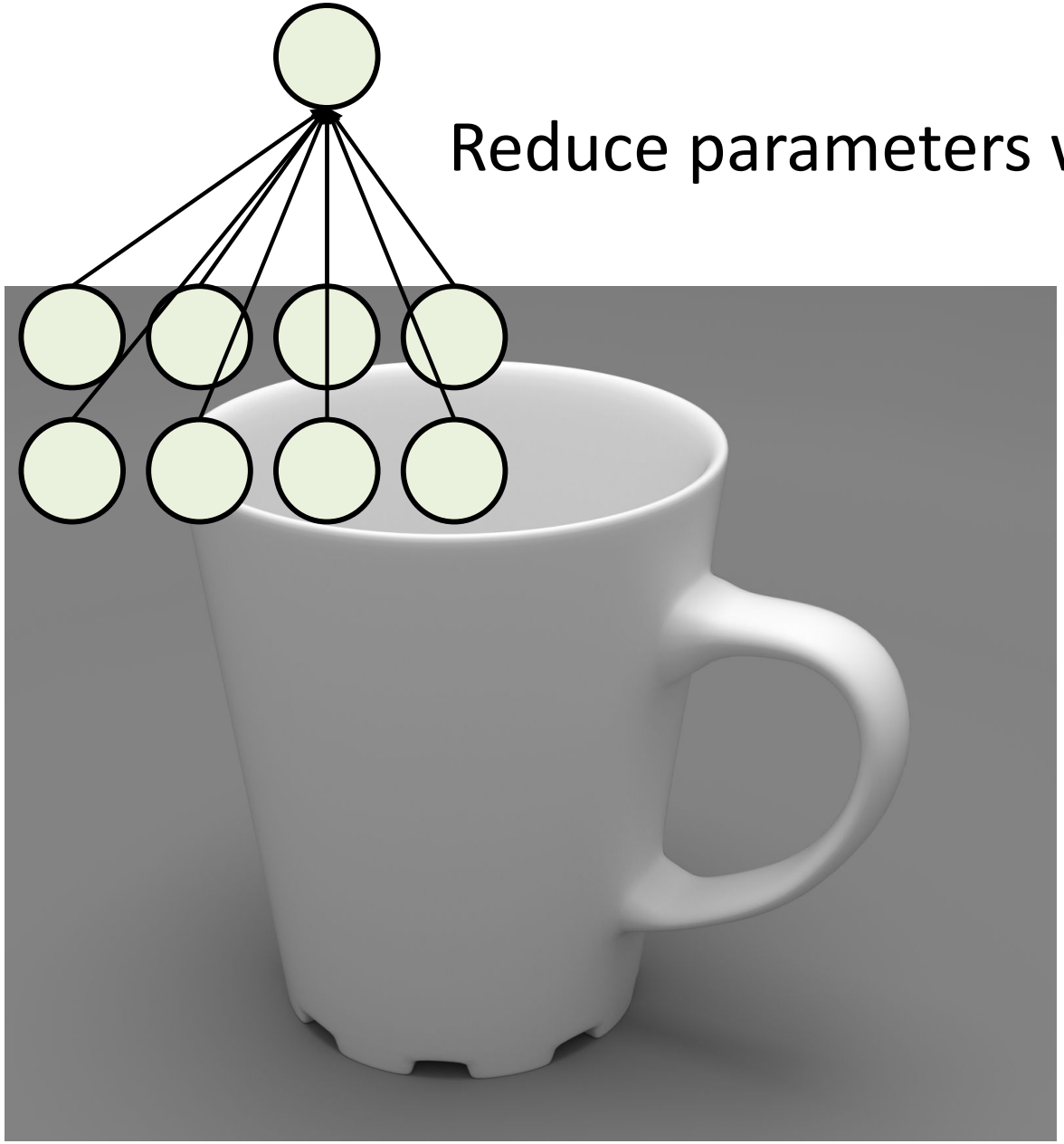


Main problem



- **Extremely large number of connections.**
- **More parameters to train.**
- **Higher computational expense.**

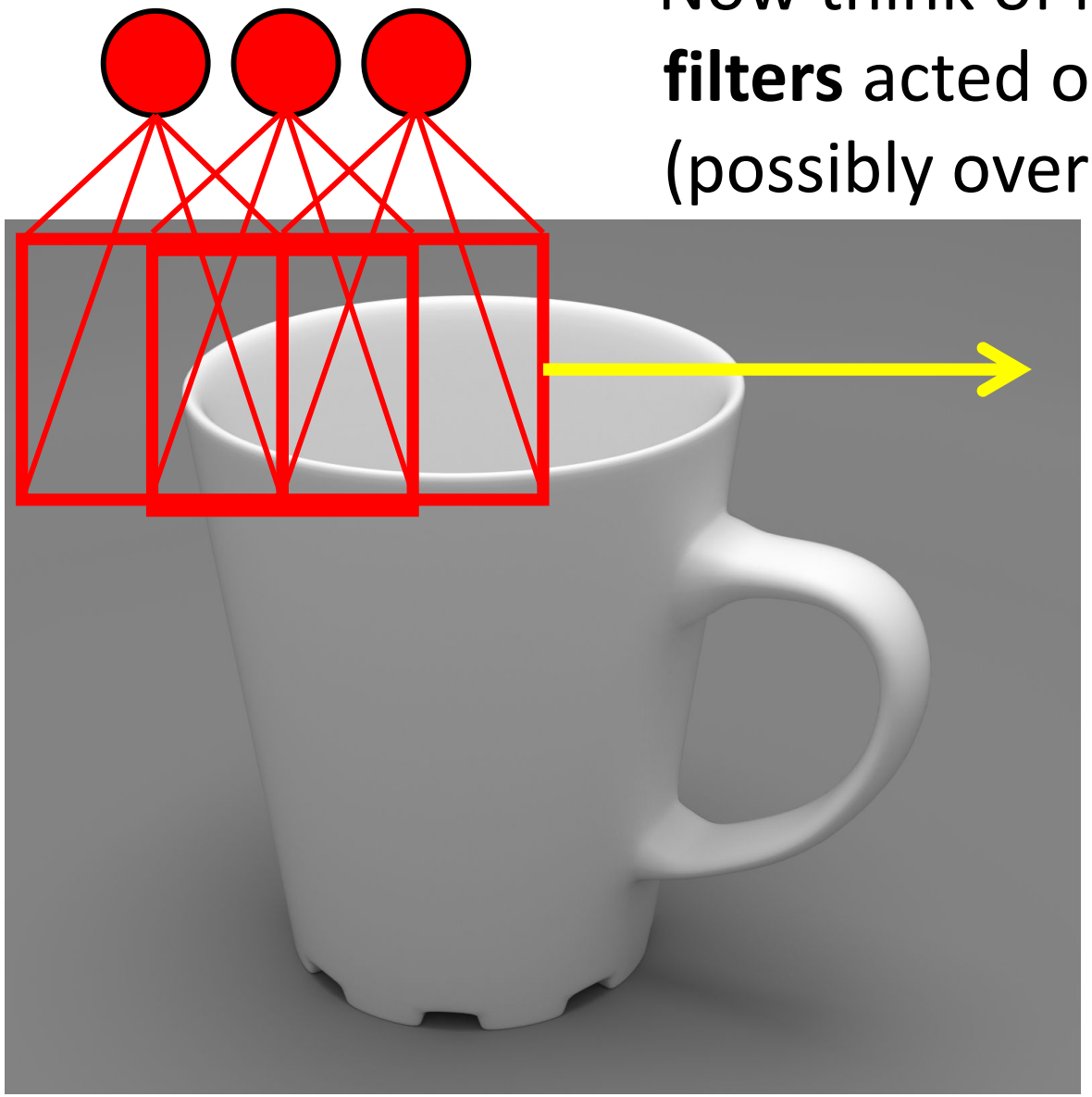
Local connectivity



Reduce parameters with **local connections!**

Neurons as convolution filters

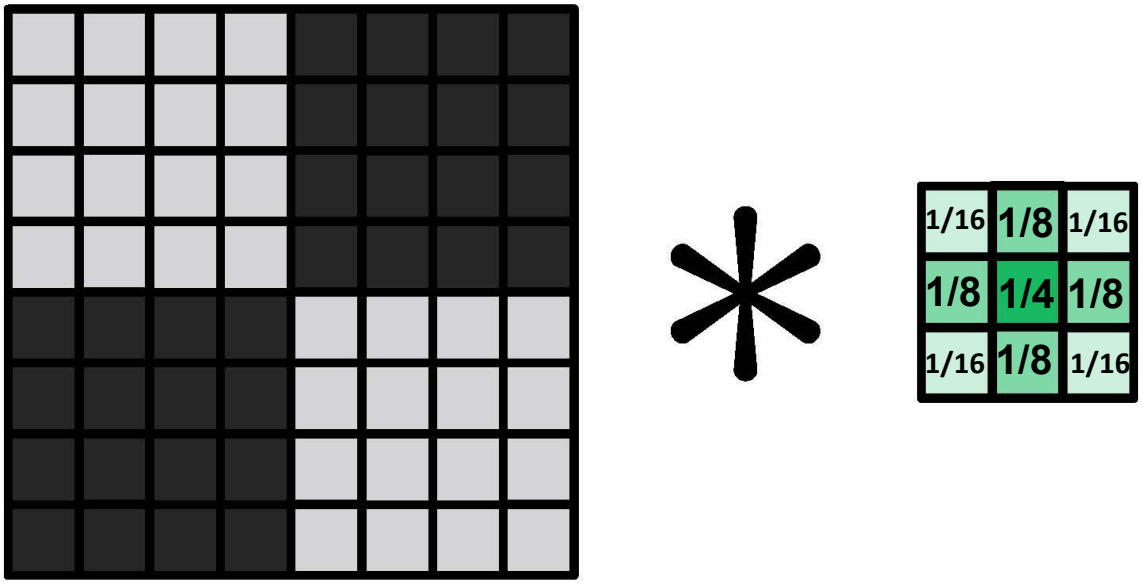
Now think of neurons as convolutional **filters** acted on small adjacent (possibly overlapping) windows



2D Convolution Review

$$O(i,j) = \sum_{k=-n}^{k=n} \sum_{l=-n}^{l=n} w(k,l) I(i+k, j+l)$$

Note: formally, this is called cross-correlation.

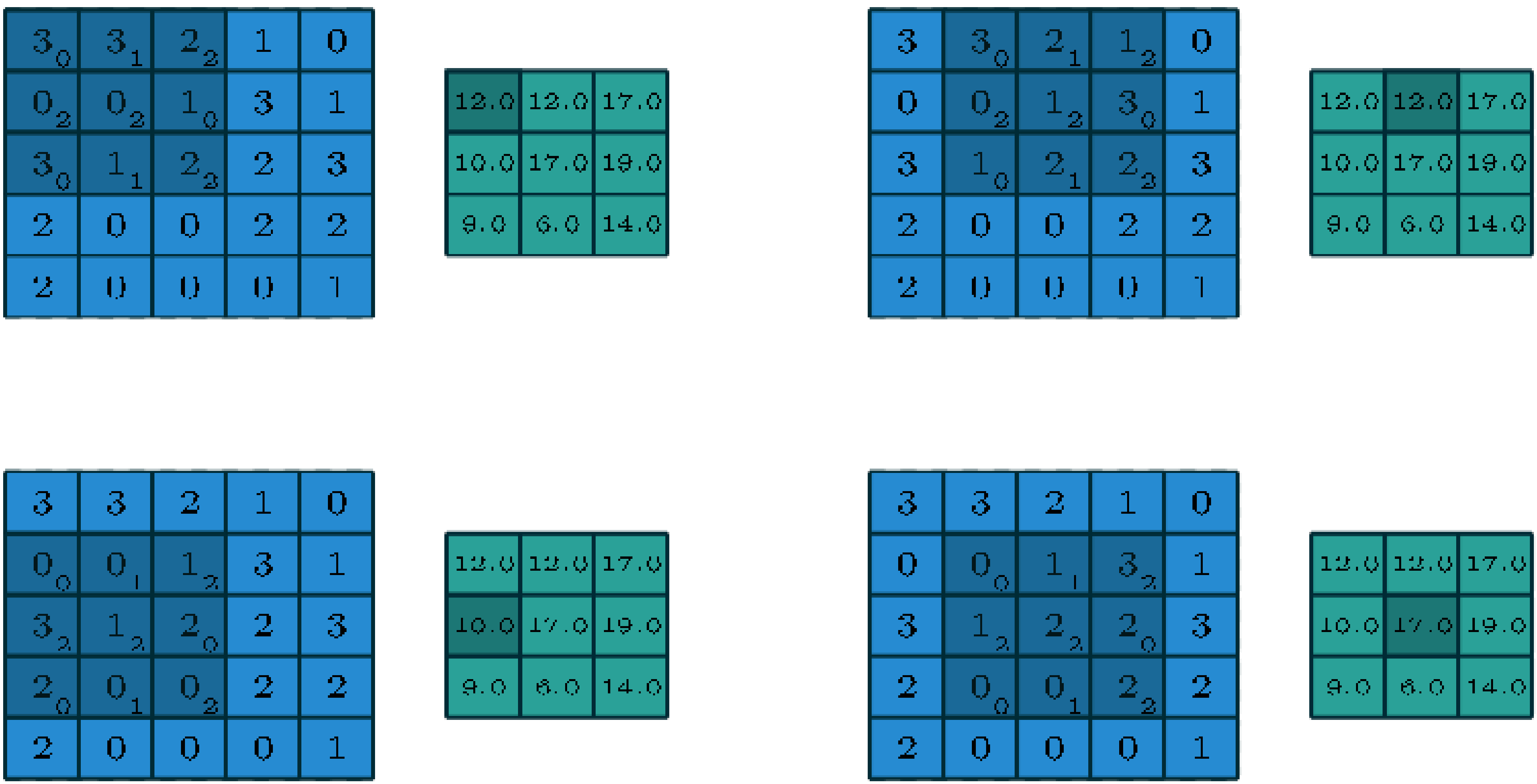


$$O(i,j) = \sum_{k=-n}^{k=n} \sum_{l=-n}^{l=n} w(k,l) I(i-k, j-l)$$

Note: formally, this is convolution. The minus difference (i.e., flipping) is not important for neural networks since the filters are learned.

2D Convolution Example

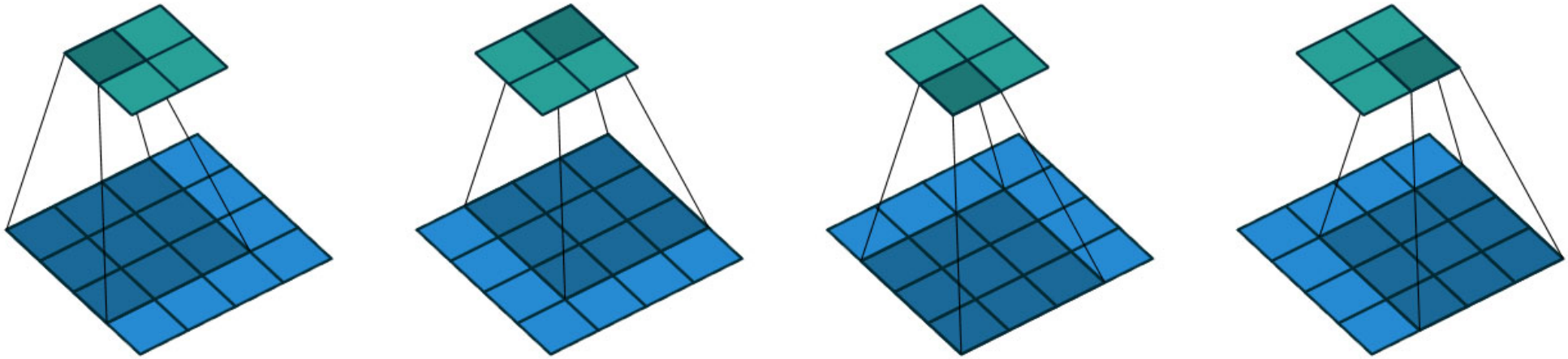
Blue is the input image, **filter weights** are on the bottom corner of the pixels, **green-ish** is the output.



2D Convolution: boundaries

What about the pixels along the image boundary?

a) don’t perform convolution when filter goes outside the image

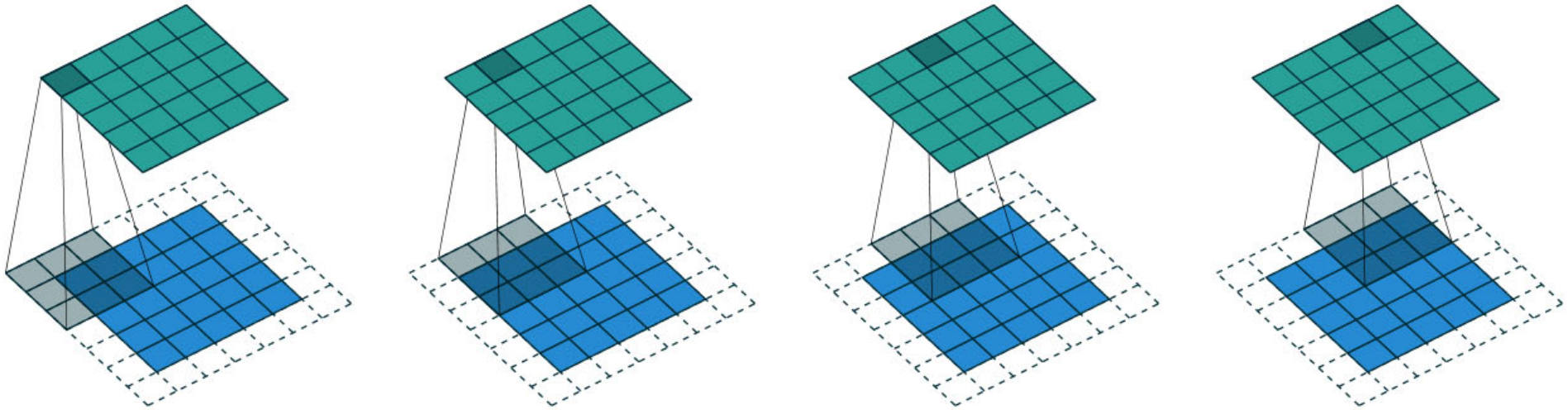


(3x3 kernel, stride 1 pixel, output has smaller size)

2D Convolution: boundaries

What about the pixels along the image boundary?

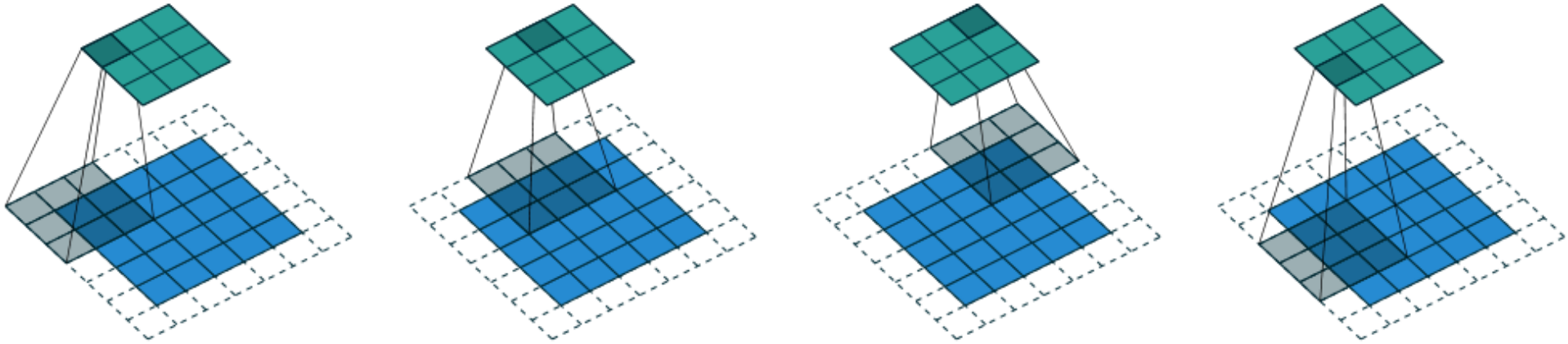
b) extend image boundary using constant intensity pixels (padding) or reflect pixels along the border



(3x3 kernel, stride 1 pixel, padding 1 pixel, output has same size)

2D convolution: stride

The filter can shift horizontally/vertically more than one pixels (**stride**: amount of shifting)



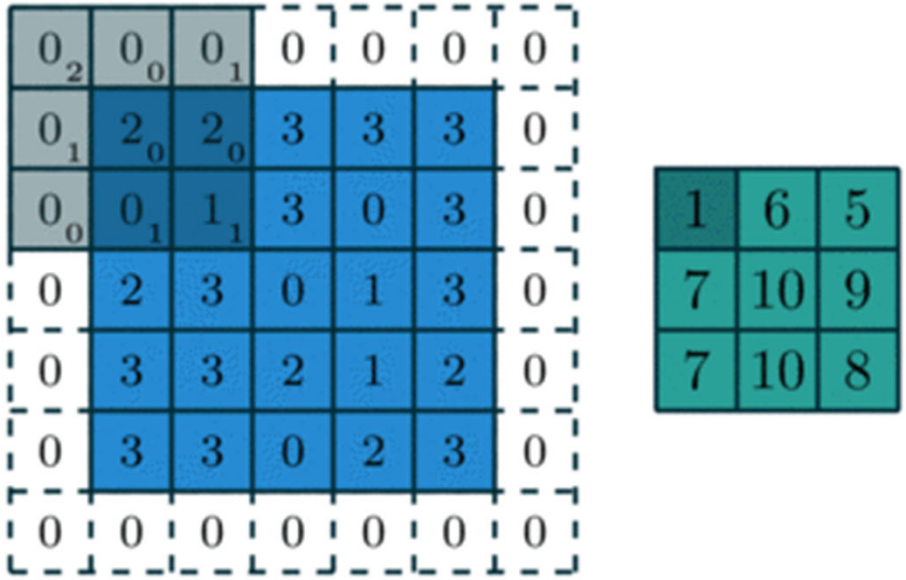
(3x3 kernel, stride 2 pixels, padding 1 pixel, output has smaller size)

2D Convolution: Output Size

How do we compute output size given convolution setting?

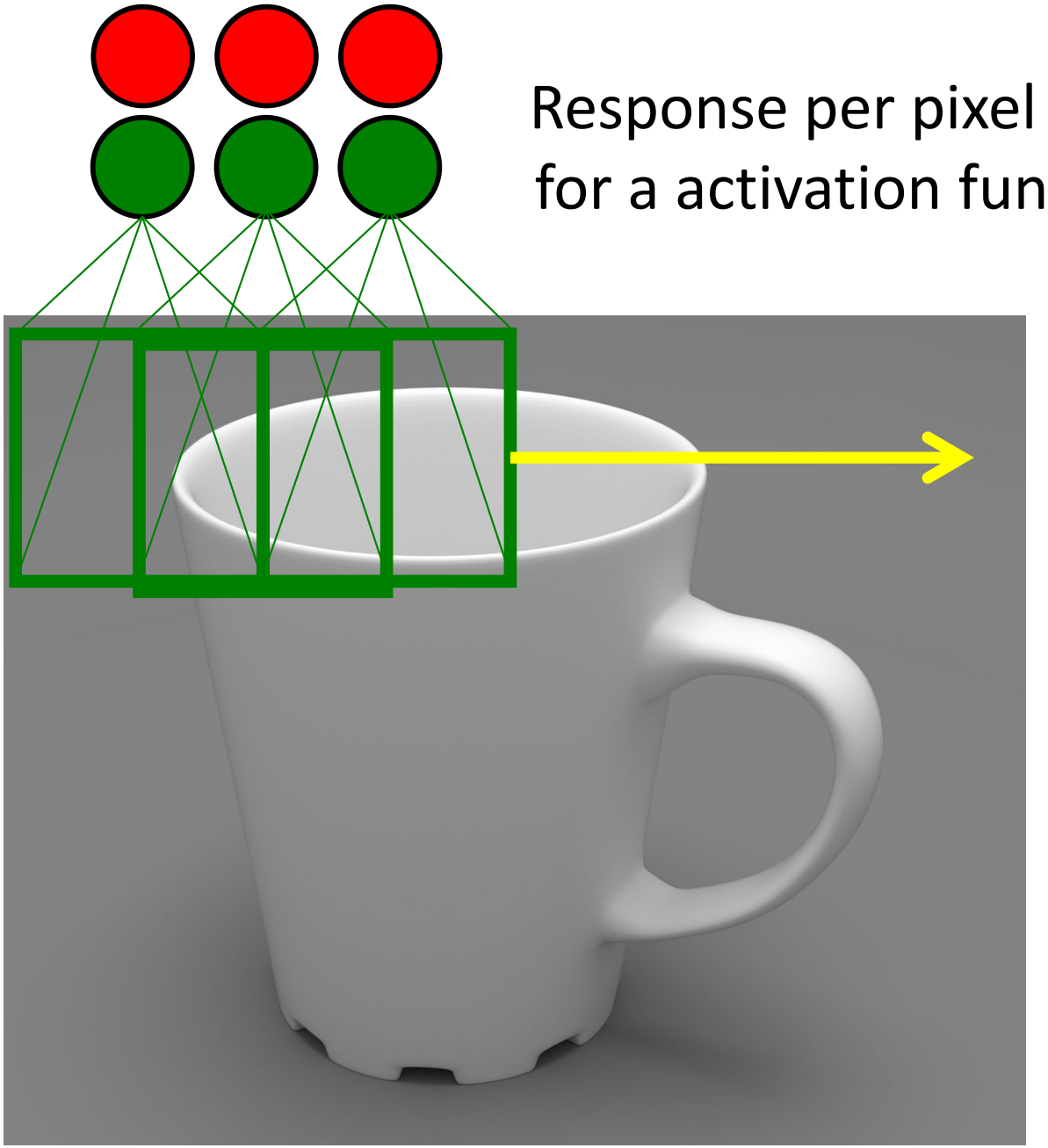
output size is:

[(input size) – (receptive field size) + 2 * (padding size) **]** / stride + 1



(5x5 input, 3x3 kernel, stride 2 pixels, padding 1 pixel, output is 3x3)

Can have many filters!

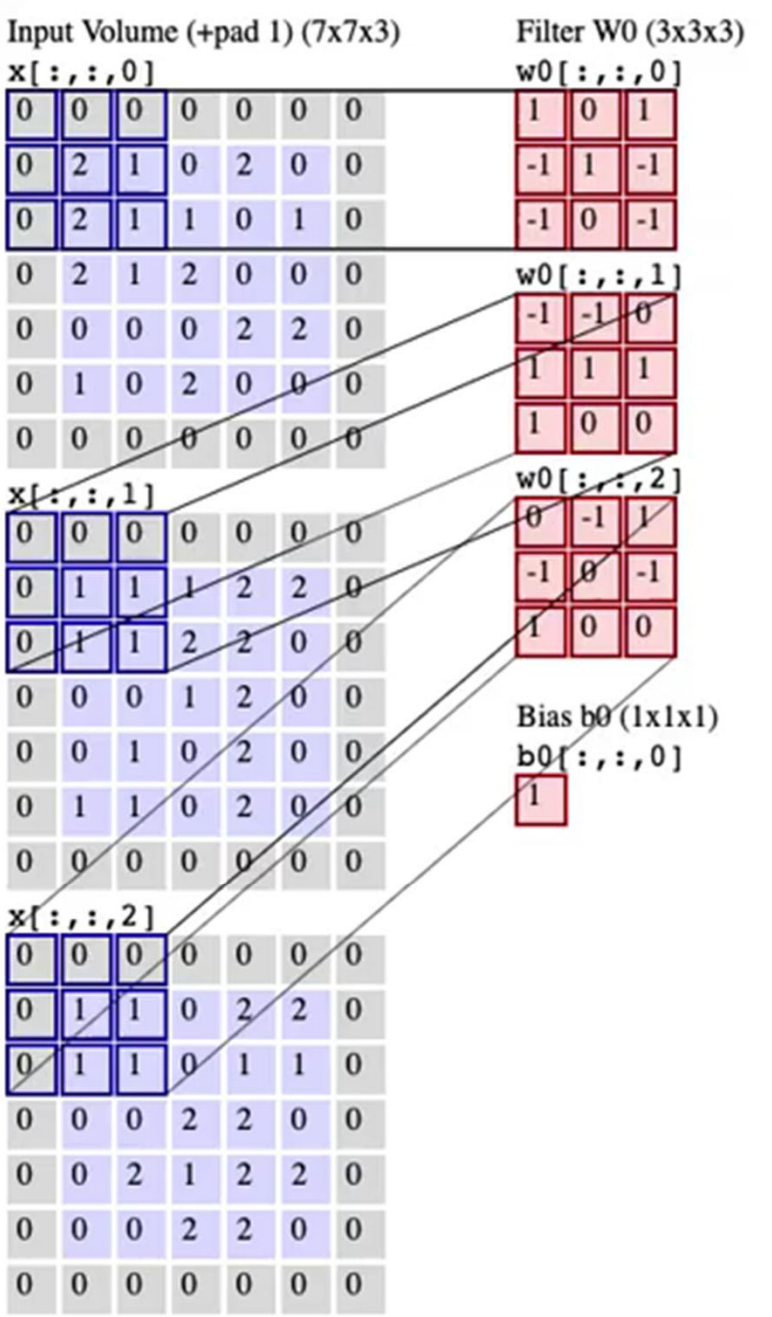


Response per pixel (i,j), per filter f

for a activation function g: $h_{i,j,f} = g(\boldsymbol{w_f} \cdot \boldsymbol{x_{i,j}})$

ReLu, LReLu etc

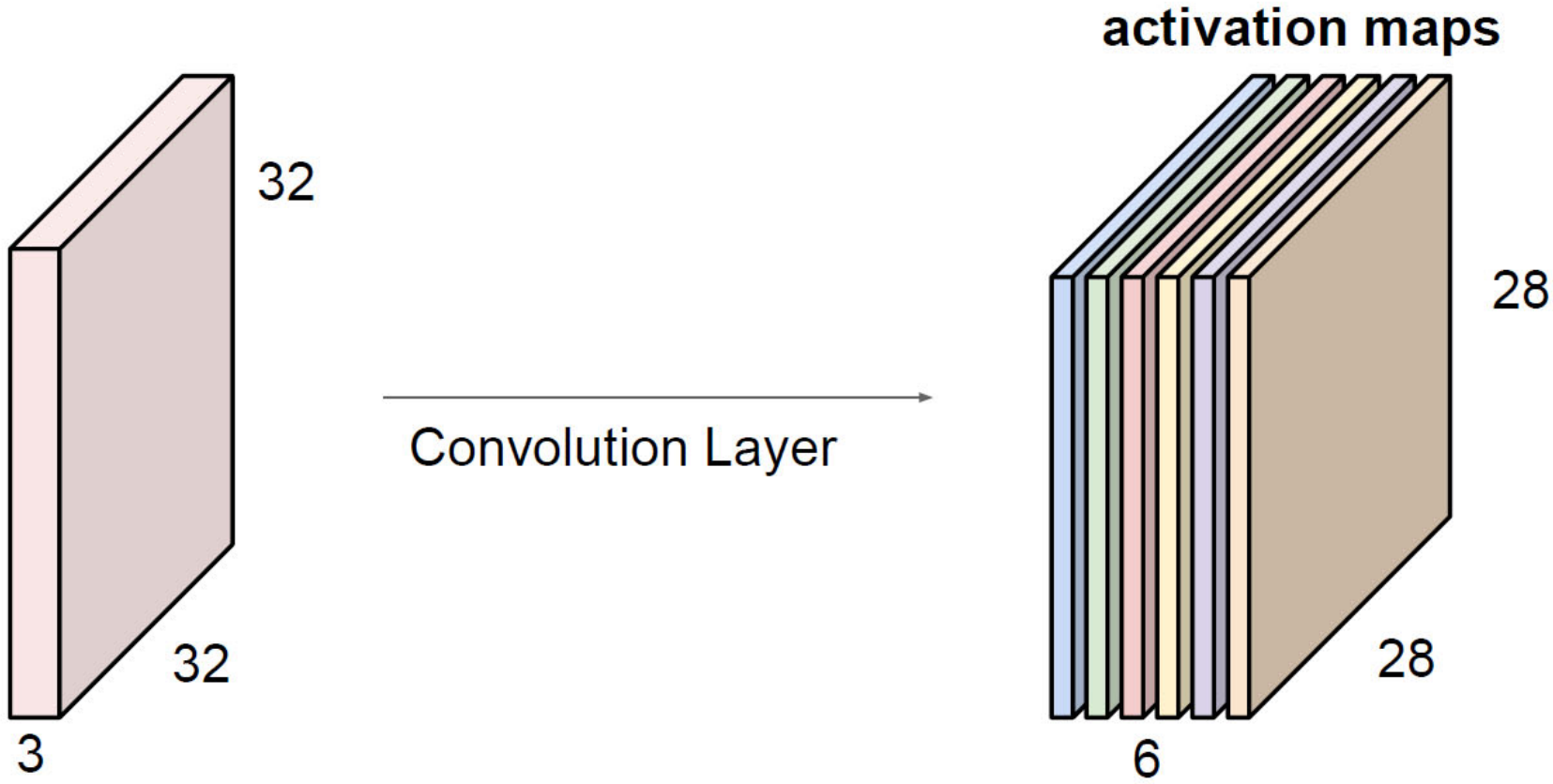
Convolution when input has multiple channels



$$O(i,j,q) = \sum_{k=-n}^{k=n} \sum_{l=-n}^{l=n} \sum_{channel\ c} w_q(k,l,c) I(i+k, j+l, c) + b_q$$

Convolution layer

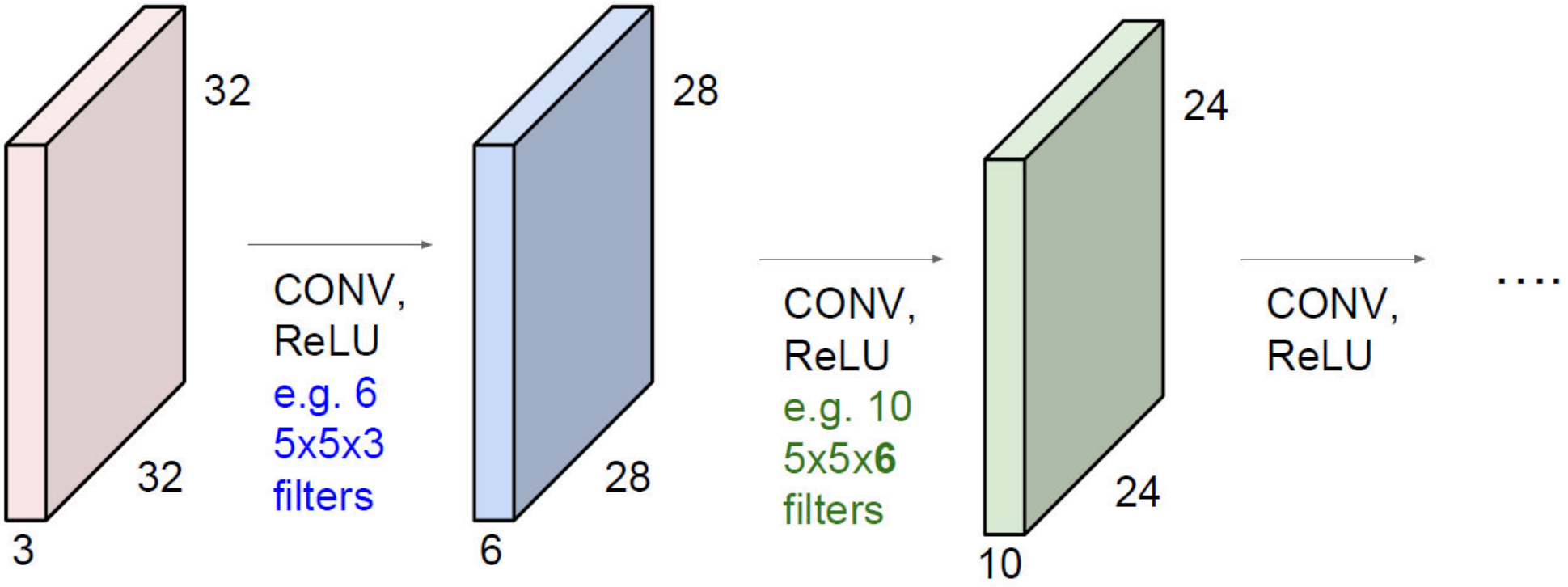
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



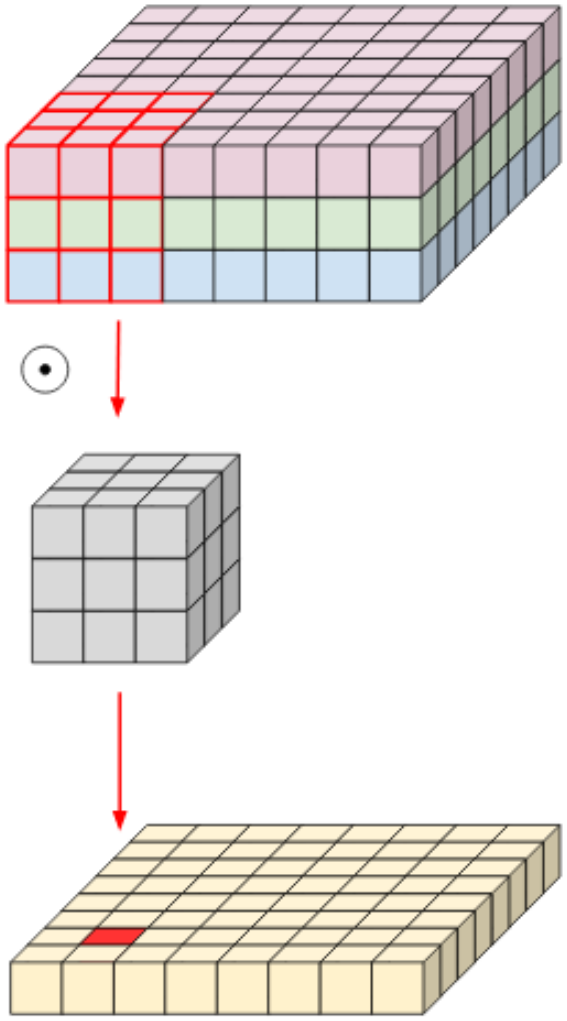
We stack these up to get a "new image" of size 28x28x6!

Convolution layers

Stack several convolutional layers together!

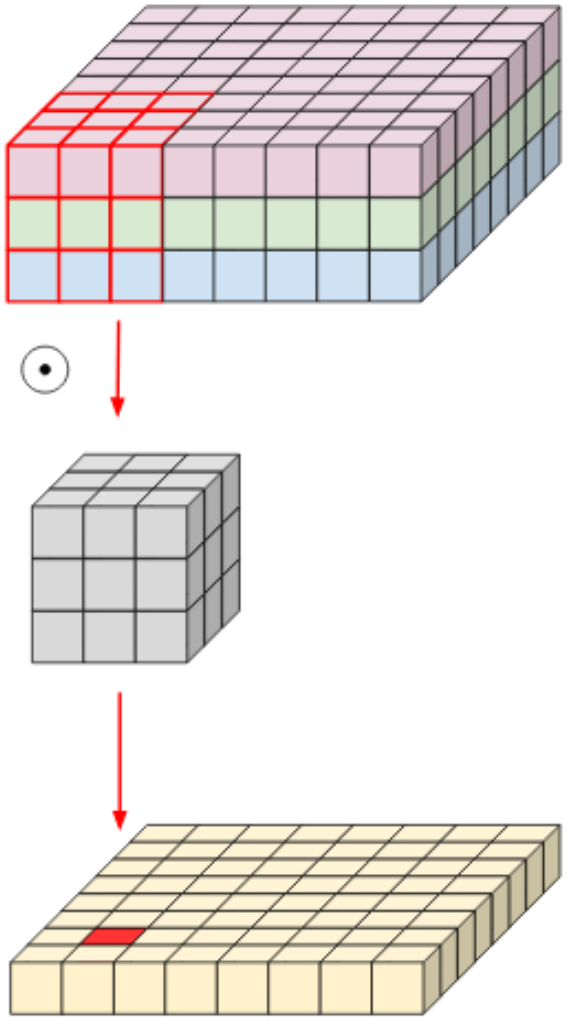


“Standard” convolution

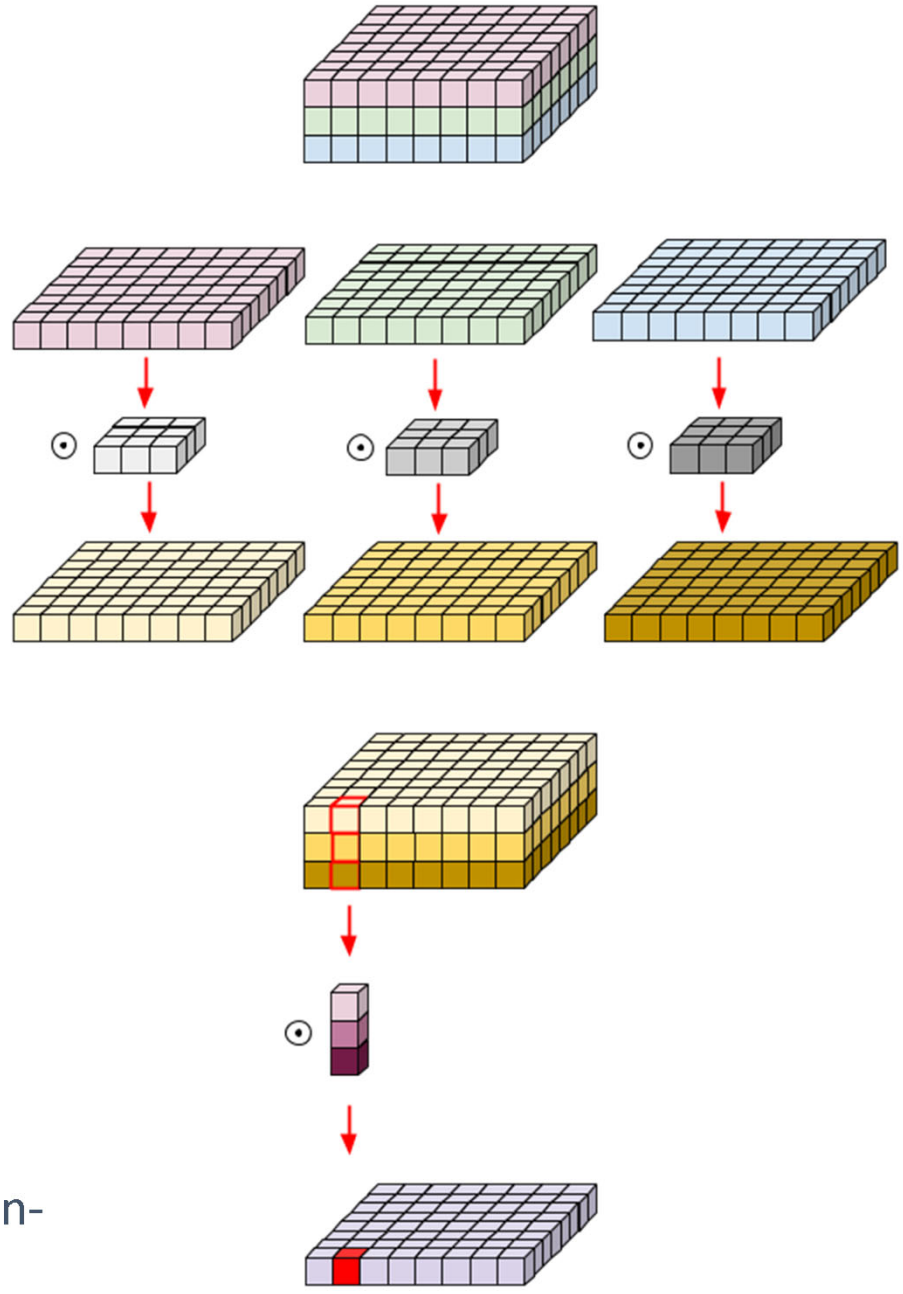


https://medium.com/@zurister/depth-wise-convolution-and-depth-wise-separable-convolution-37346565d4ec

“Standard” convolution



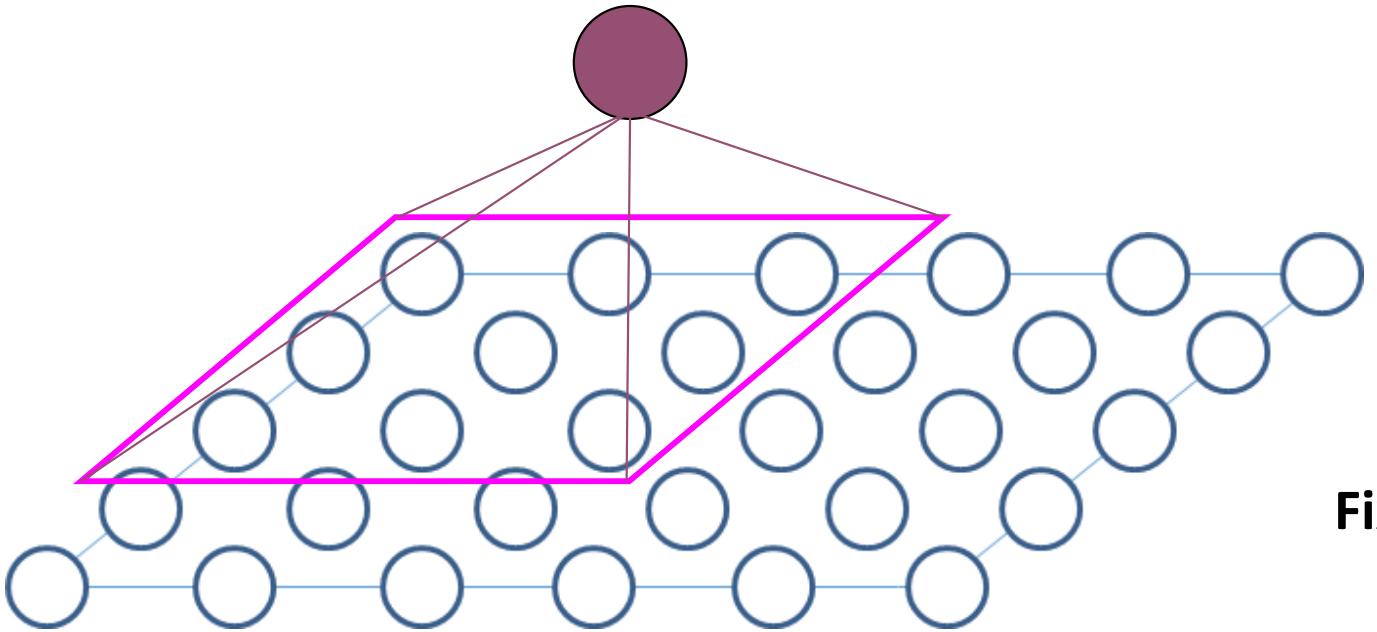
“Depthwise separable” convolution



https://medium.com/@zurister/depth-wise-convolution-and-depth-wise-separable-convolution-37346565d4ec

Pooling layer

Apart from hidden layers dedicated to convolution, we can have layers that perform subsampling



Max pooling:

$$h_{p',f} = \max_{p}(\mathbf{x_p})$$

Mean pooling:

$$h_{p',f} = \underset{p}{avg}(\mathbf{x_p})$$

Fixed filter (e.g., Gaussian):

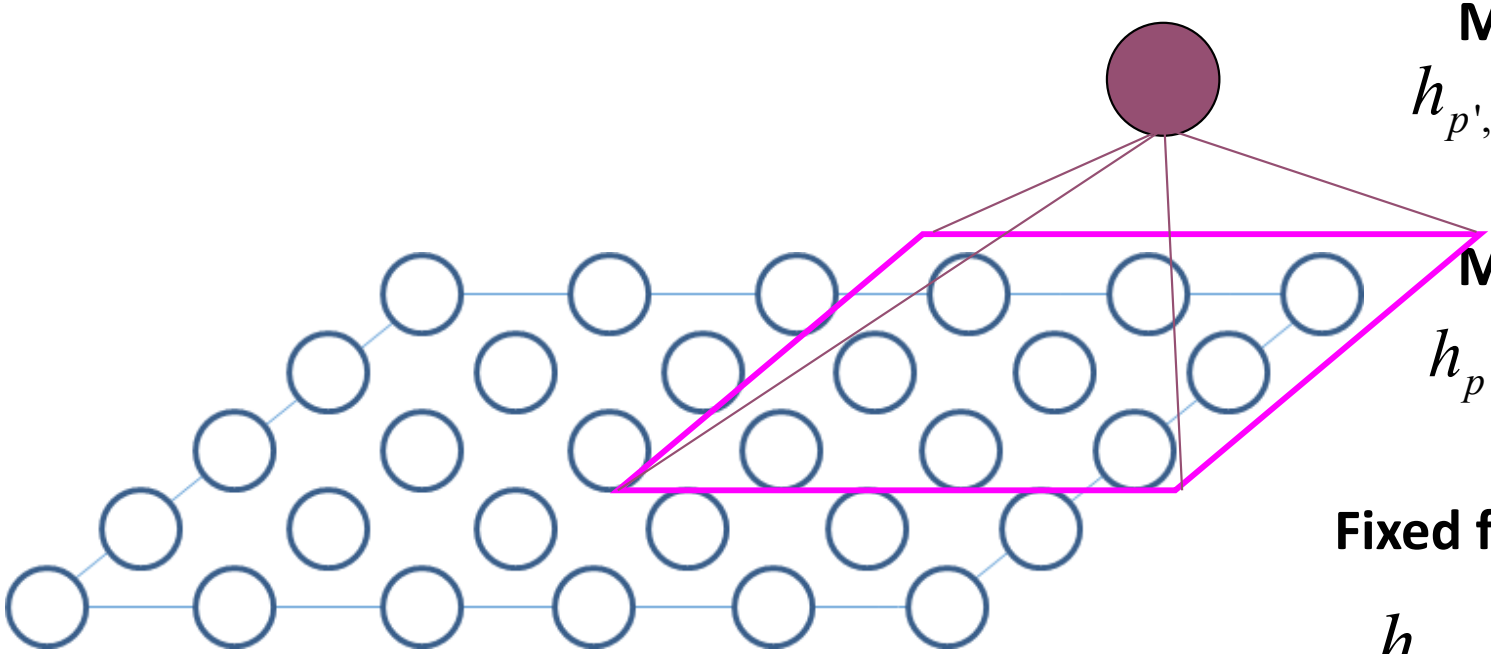
$$h_{p',f} = w_{gaussian} \cdot \mathbf{x_p}$$

Progressively reduce the resolution of the image, so that the next convolutional filters are applied on larger scales

[Scherer et al., ICANN 2010]
[Boureau et al., ICML 2010]

Pooling layer

Apart from hidden layers dedicated to convolution, we can have layers that perform subsampling



Max pooling:

$$h_{p',f} = \max_{p}(\mathbf{x_p})$$

Mean pooling:

$$h_{p',f} = \underset{p}{avg}(\mathbf{x_p})$$

Fixed filter (e.g., Gaussian):

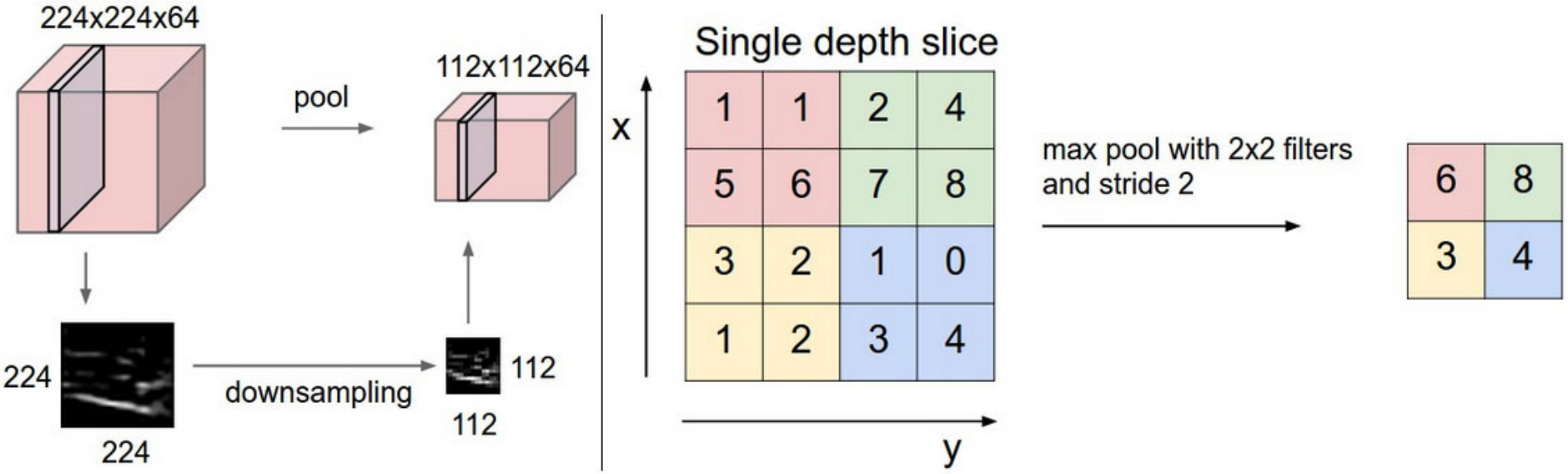
$$h_{p',f} = w_{gaussian} \cdot \mathbf{x_p}$$

Progressively reduce the resolution of the image, so that the next convolutional filters are applied on larger scales

[Scherer et al., ICANN 2010]
[Boureau et al., ICML 2010]

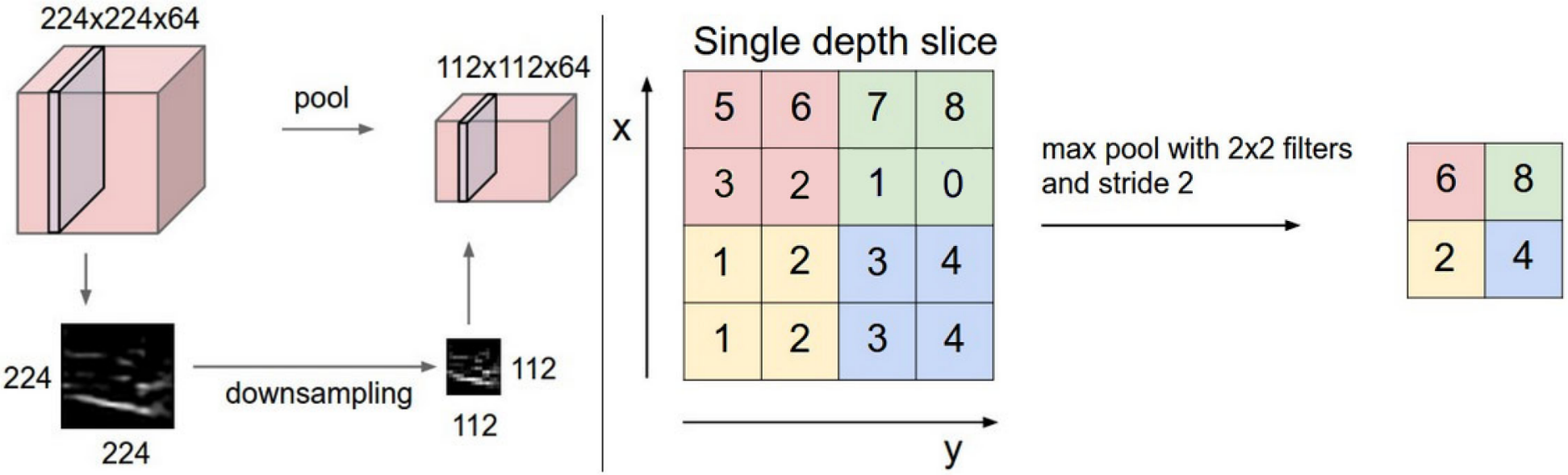
Pooling layer

Max-pooling promotes some **invariance** to input noise, or perturbations



Pooling layer

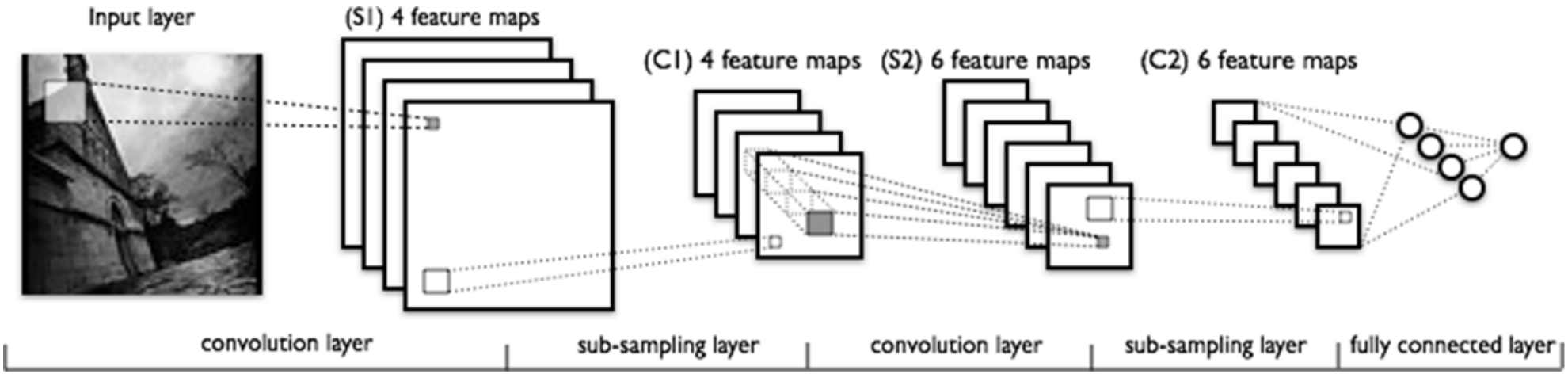
Max-pooling promotes some **invariance** to input noise, or perturbations



A mini convolutional neural network

Interchange convolutional and pooling (subsampling) layers.

In the end, **unwrap all feature maps into a single feature vector** and pass it through the classical (**fully connected**) neural network.

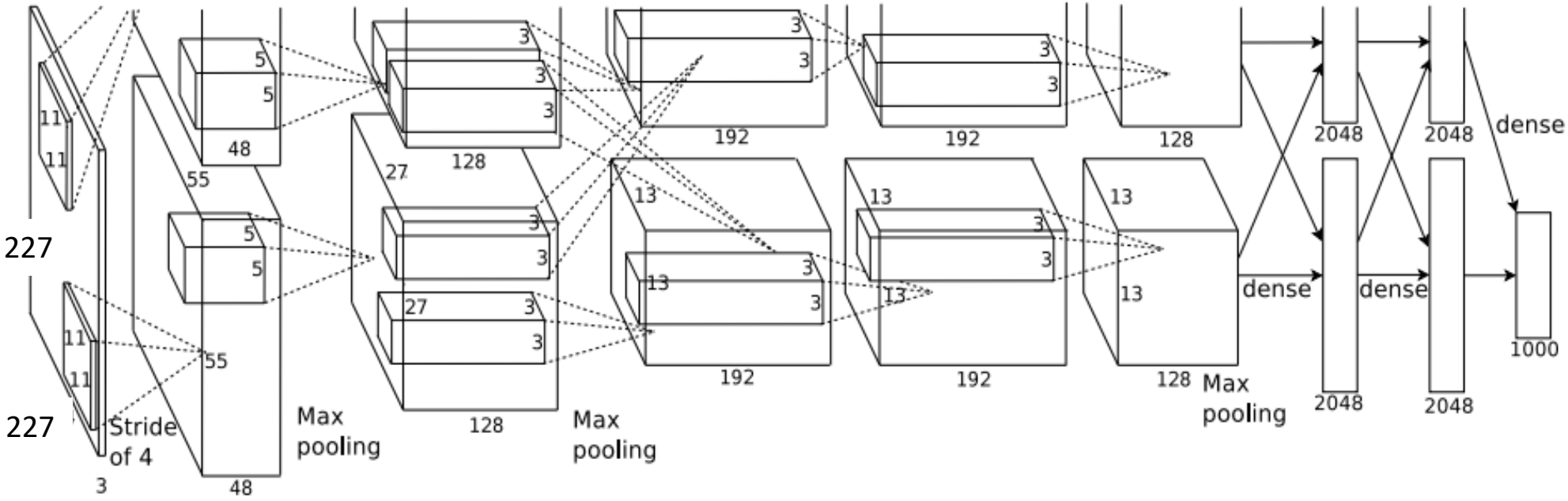


Source: http://deeplearning.net/tutorial/lenet.html

AlexNet

Proposed architecture from Krizhevsky et al., NIPS 2012:

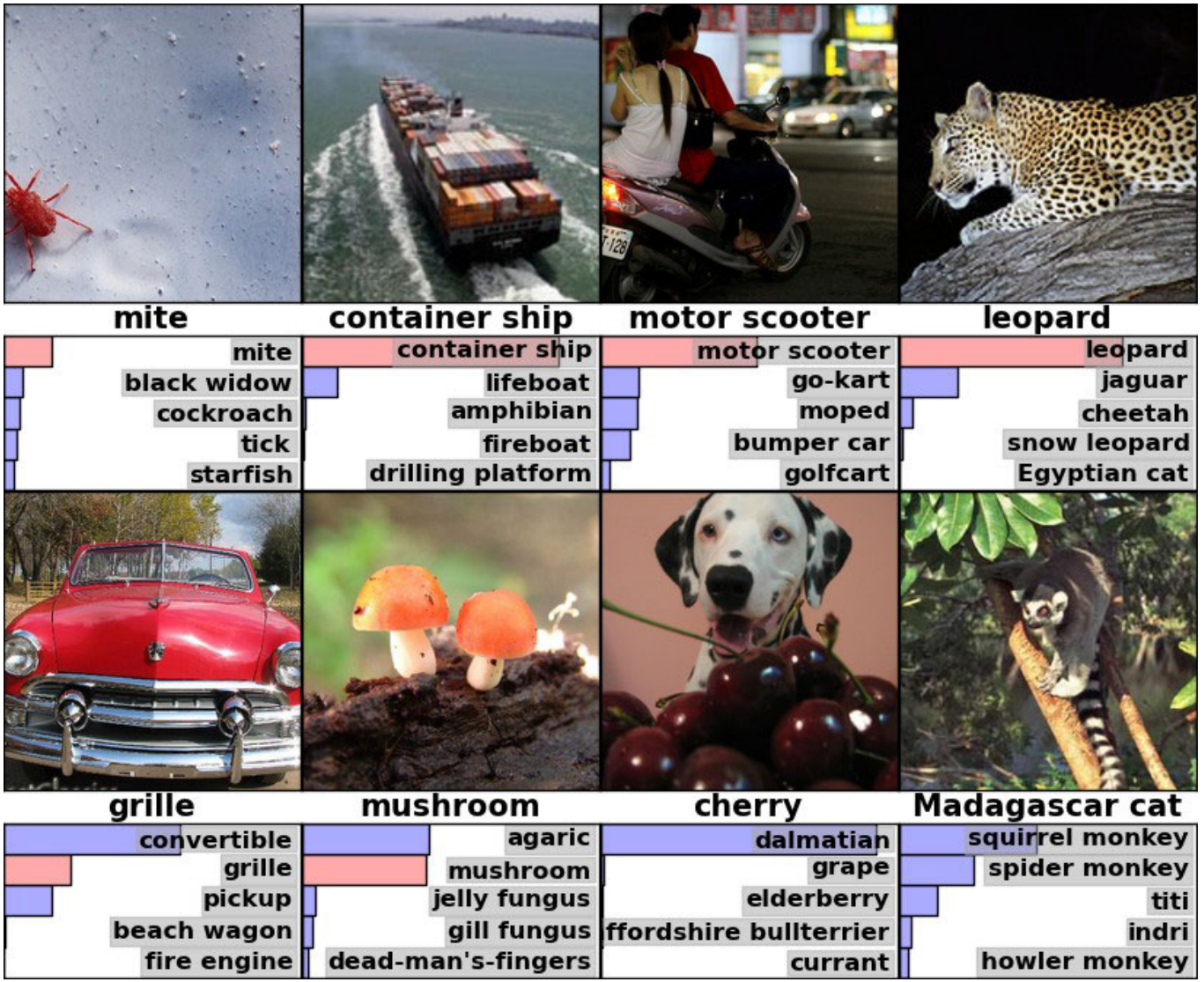
- Convolutional layers with ReLus
- Max-pooling layers
- SGD on GPU with momentum, L2 reg., dropout
- Applied to image classification (ImageNet competition – top runner & game changer)
- **Trained on a large dataset (1M images)!**



Krizhevsky et al., AlexNet, NIPS 2012

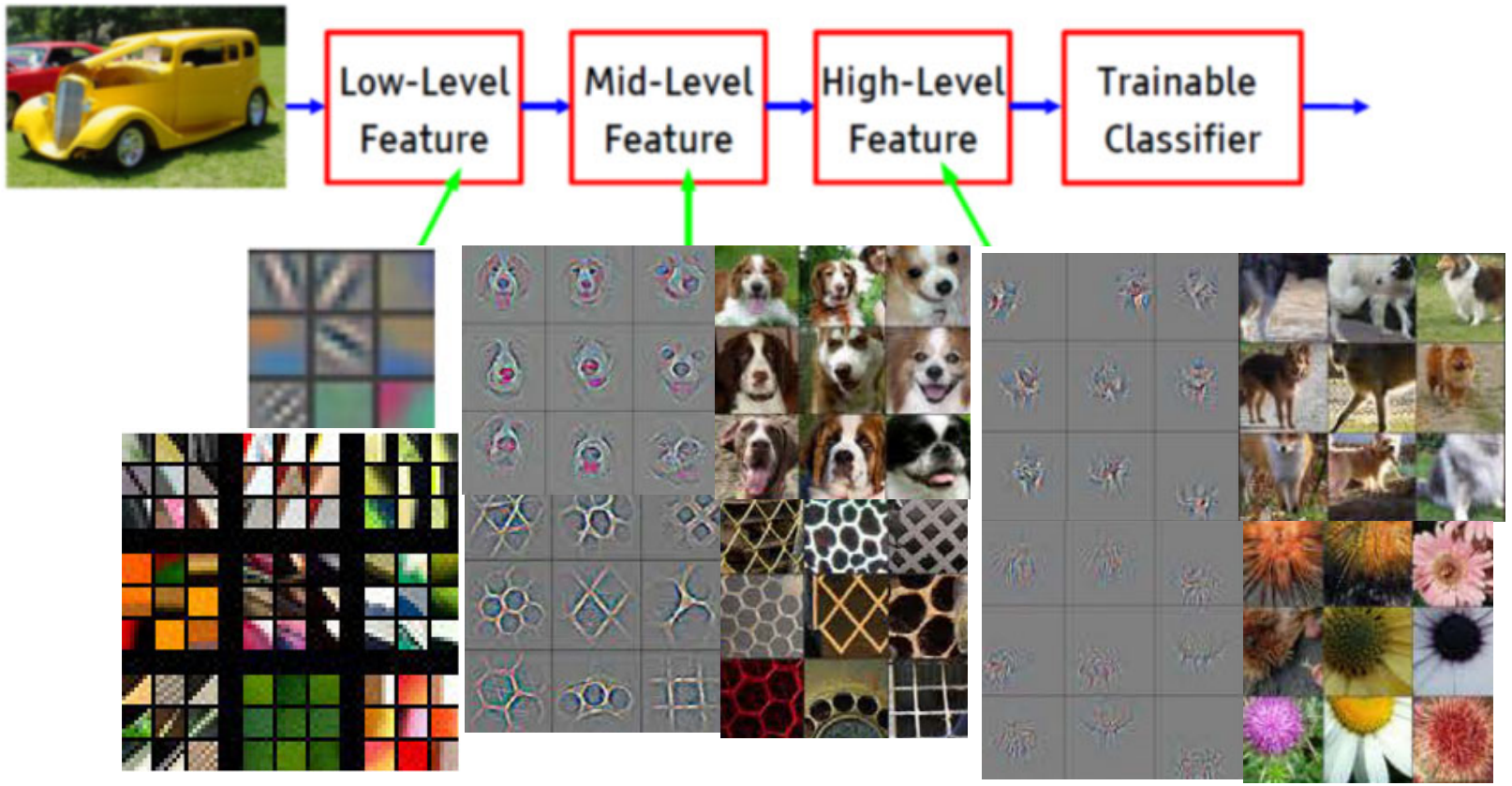
Application: Image-Net

Top result in LSVRC 2012: ~85%, Top-5 accuracy.



Learned filters

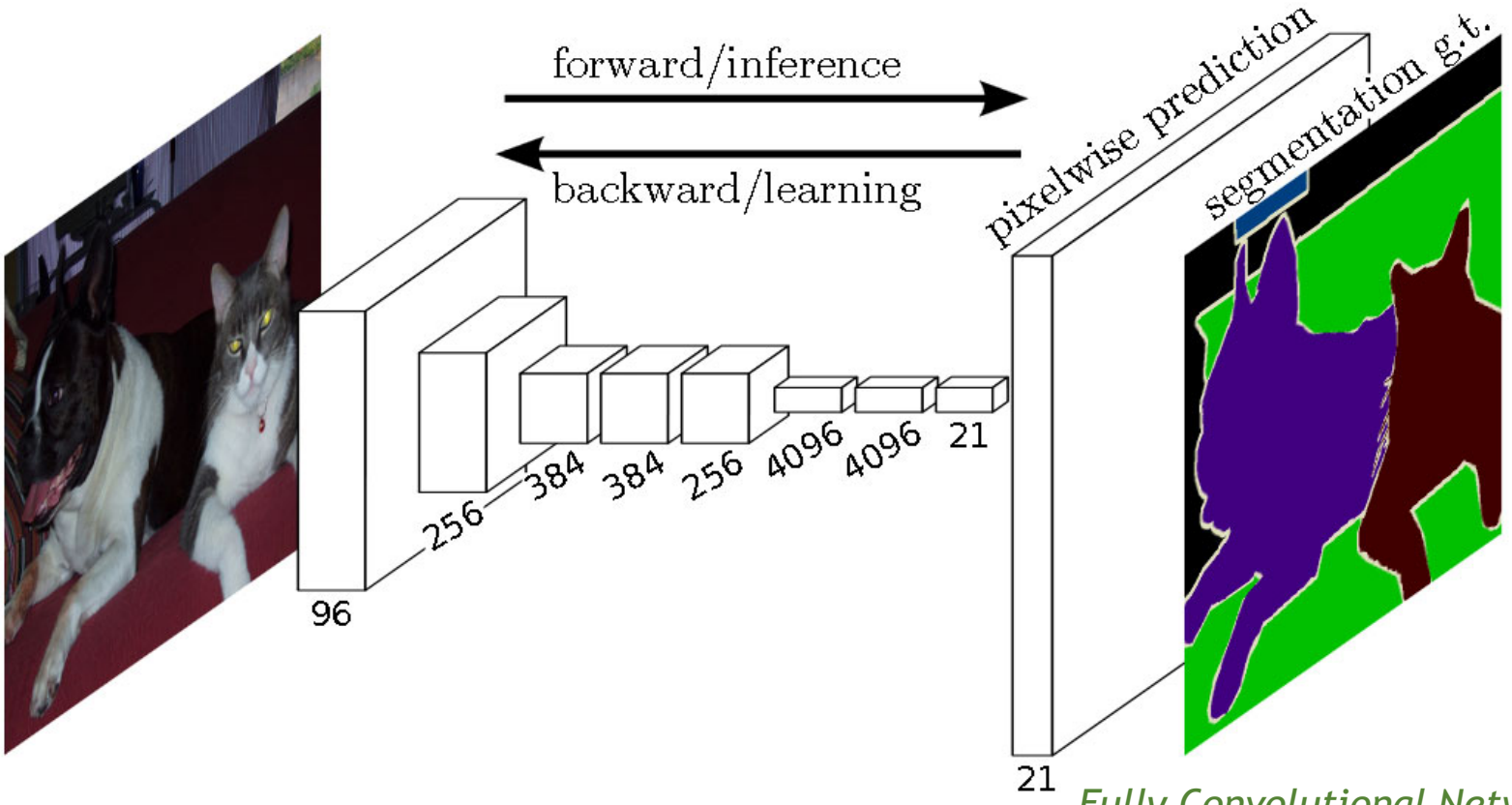
After learning, convolution filters tend to capture various **hierarchical patterns** (edges, local structures, parts…)



see Matthew D. Zeiler and Rob Fergus, Visualizing and Understanding Convolutional Networks, 2014

Application: Image Segmentation

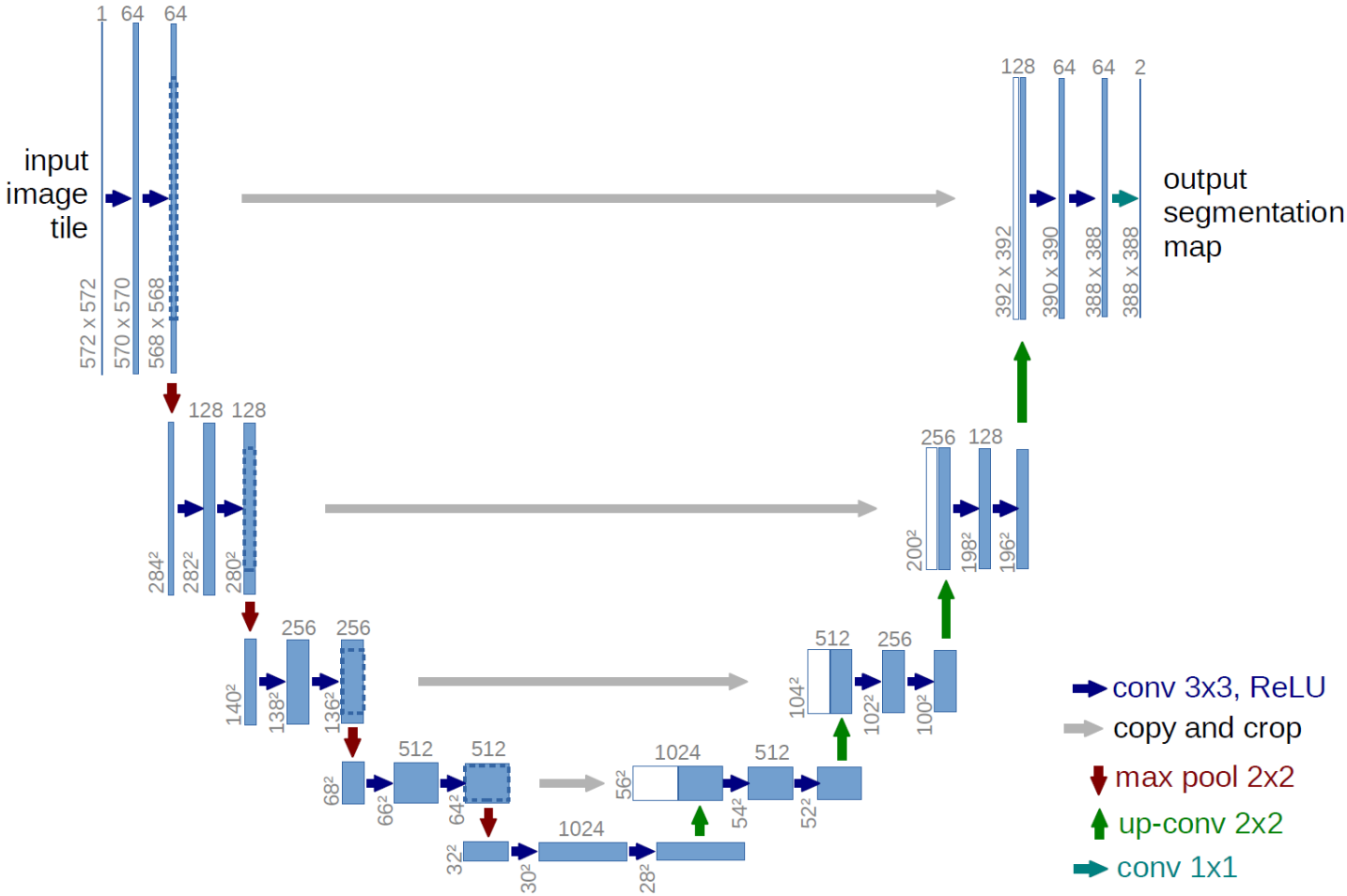
Instead of predicting a property for the whole image, output **dense predictions** e.g., probabilities of part labels per pixel **without fully connected layers**



Fully Convolutional Networks for Semantic Segmentation, Long et al. 2015

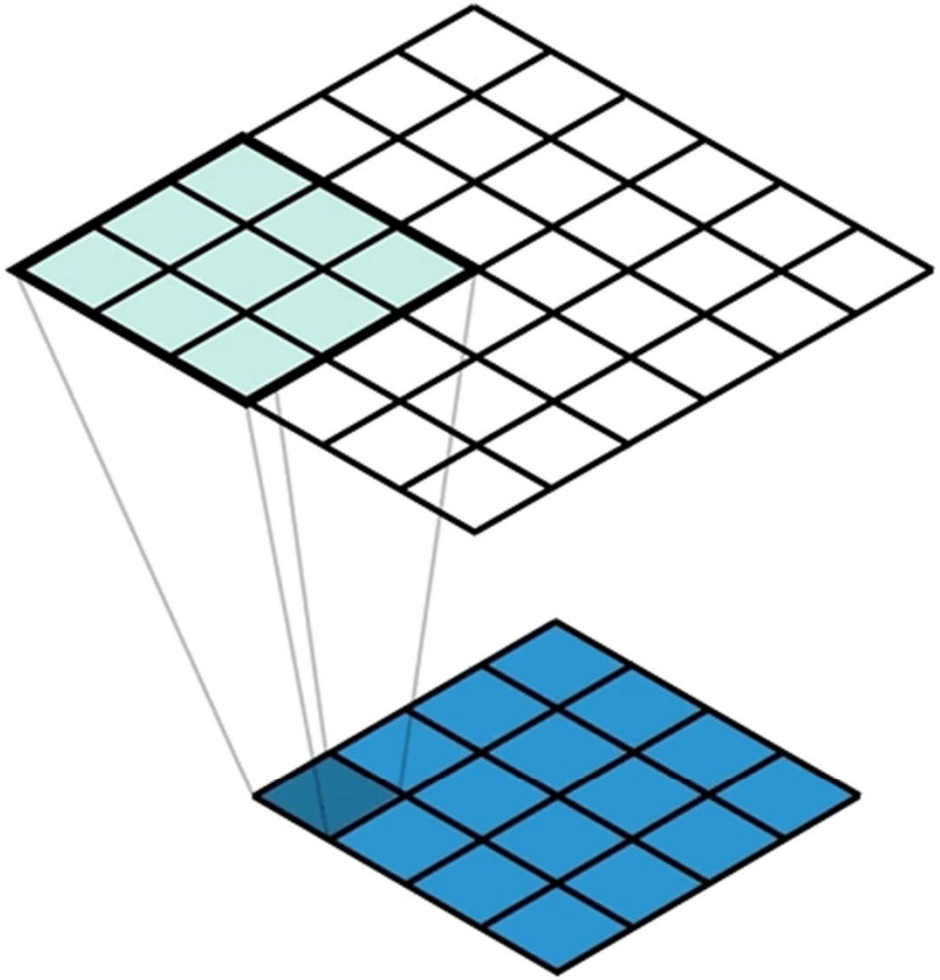
“U-Net” architecture

If the decoder relies exclusively on the last feature map of the encoder, then it can easily fail to produce fine-grained details. Concatenate feature maps of the encoder in the decoder (“U-Net”)



U-Net: Convolutional Networks for Biomedical Image Segmentation, Ronneberger et al., 2015

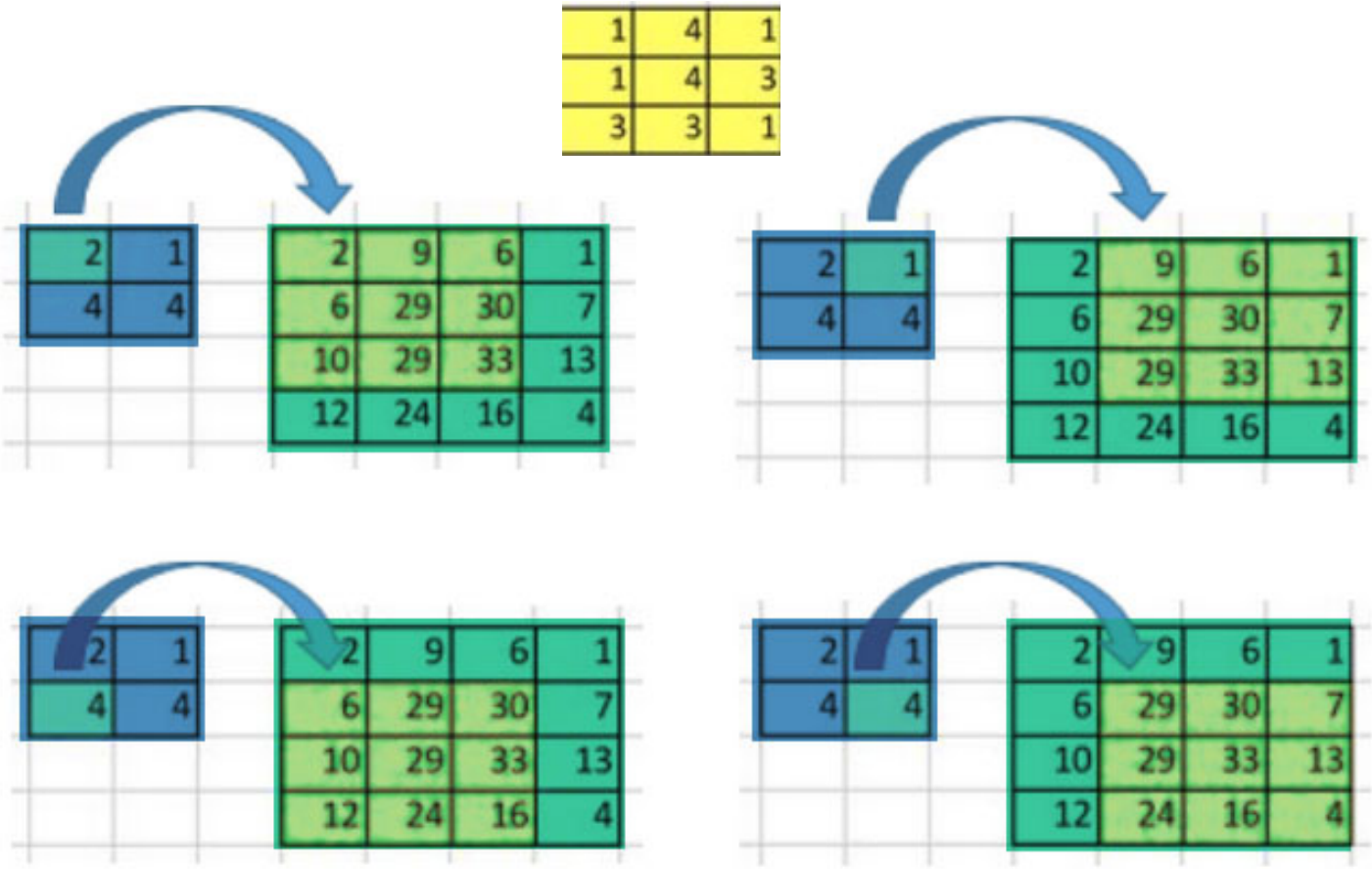
Transpose Convolution



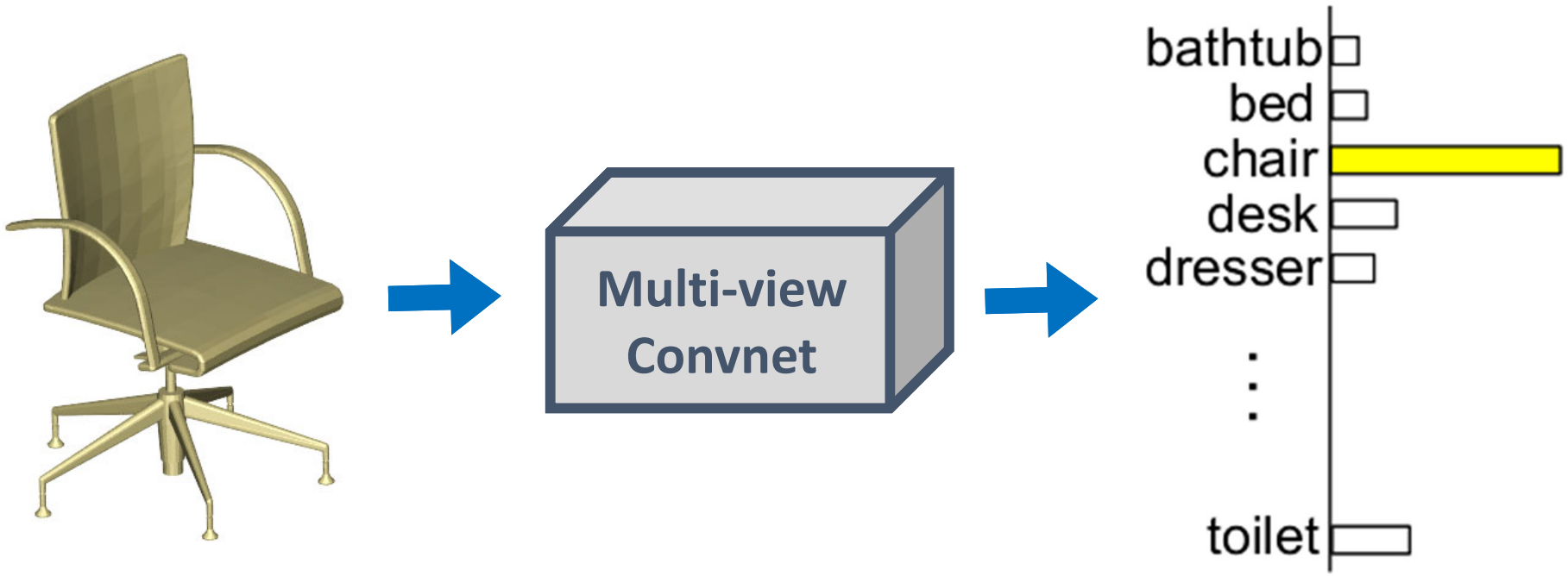
https://medium.com/apache-mxnet/transposed-convolutions-explained-with-ms-excel-52d13030c7e8

Transpose Convolution

Like convolution, but “reversed”. **Blue-ish** is the input image (2x2), filter weights are yellow (3x3), output is green-ish (4x4)

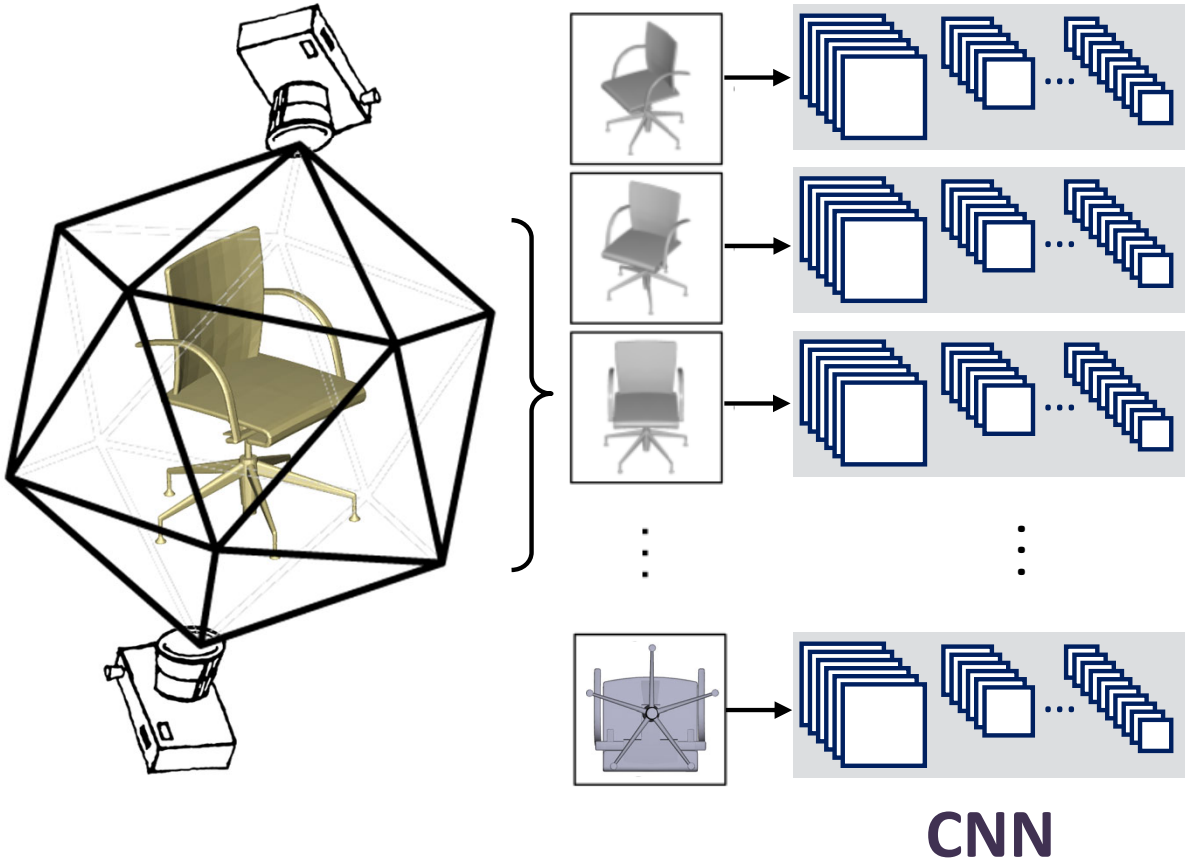


Application: 3D Shape Recognition

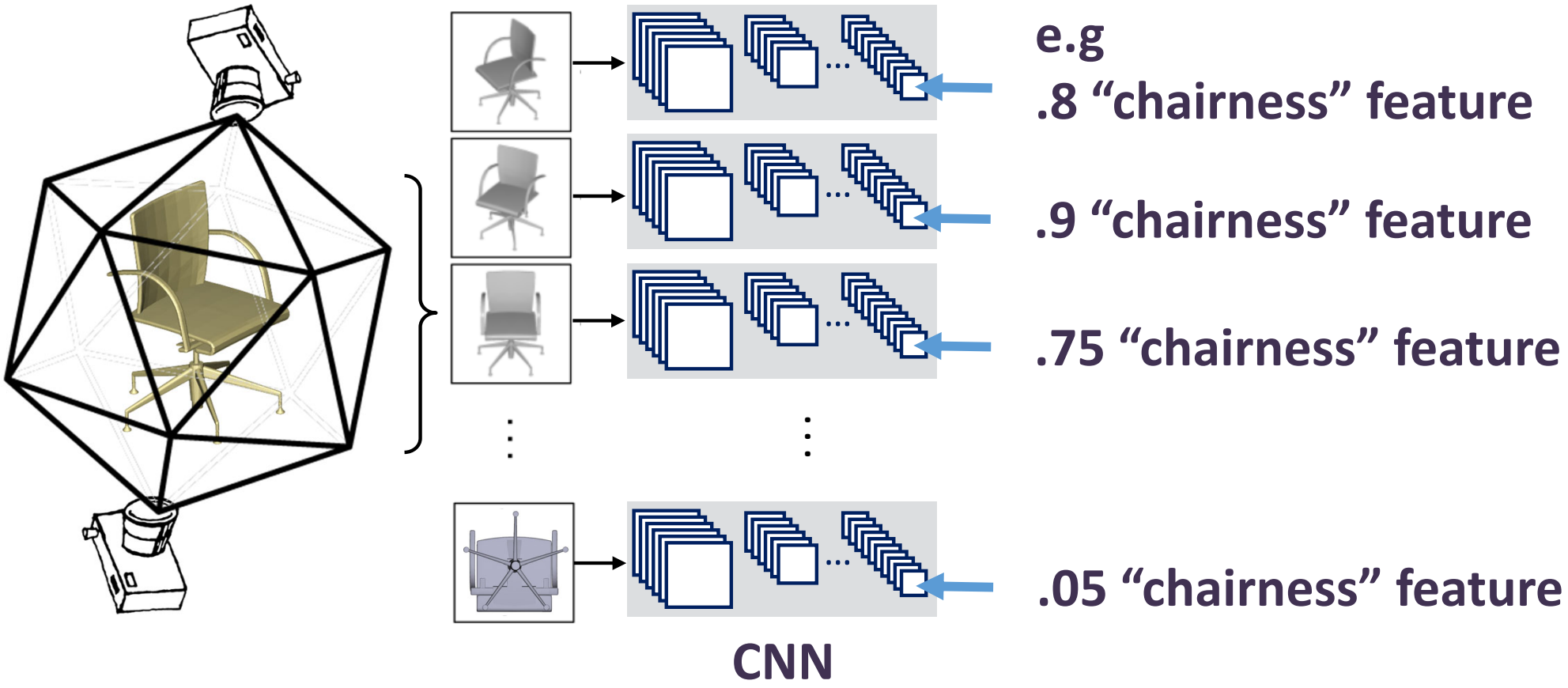


Su, Maji, Kalogerakis, Learned-Miller, Multi-view CNN for 3D Shape Recognition, ICCV 2015

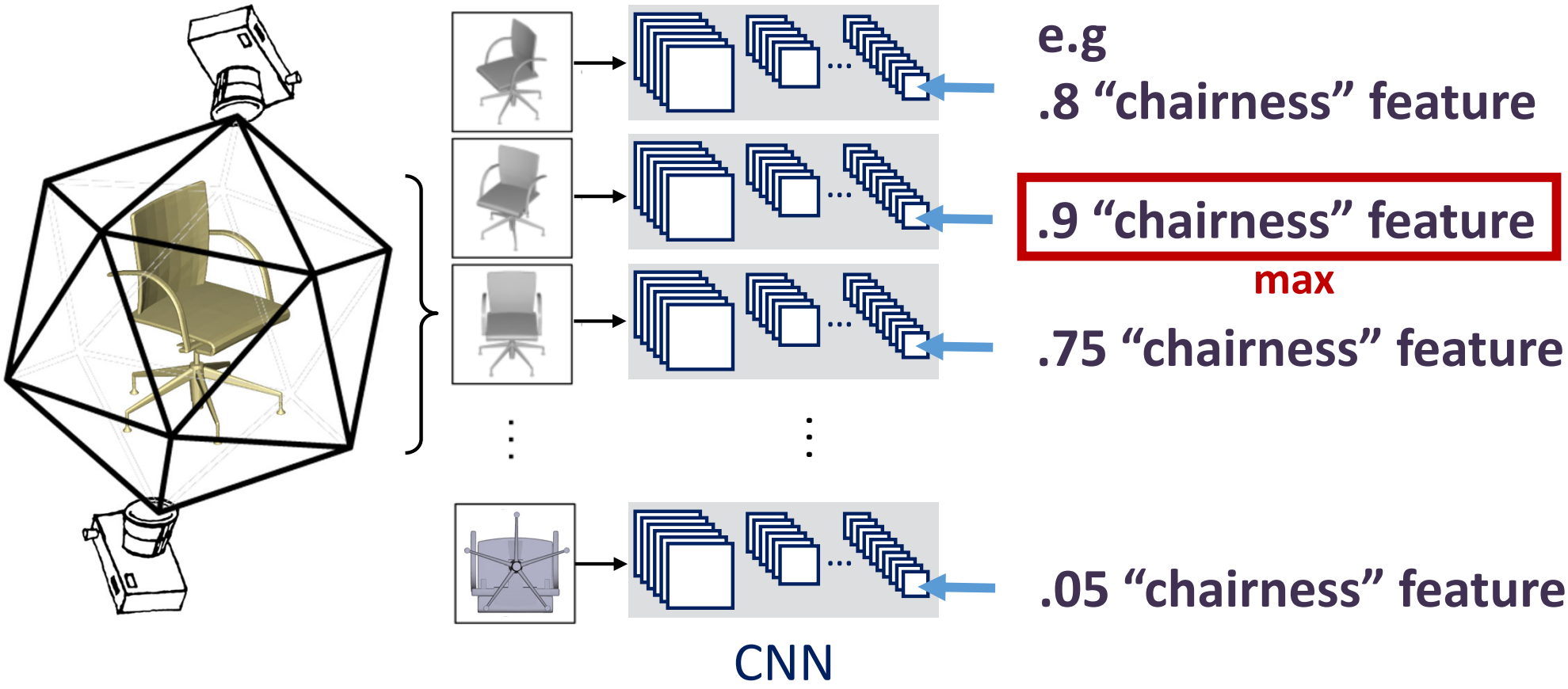
Shape recognition with multi-view CNNs



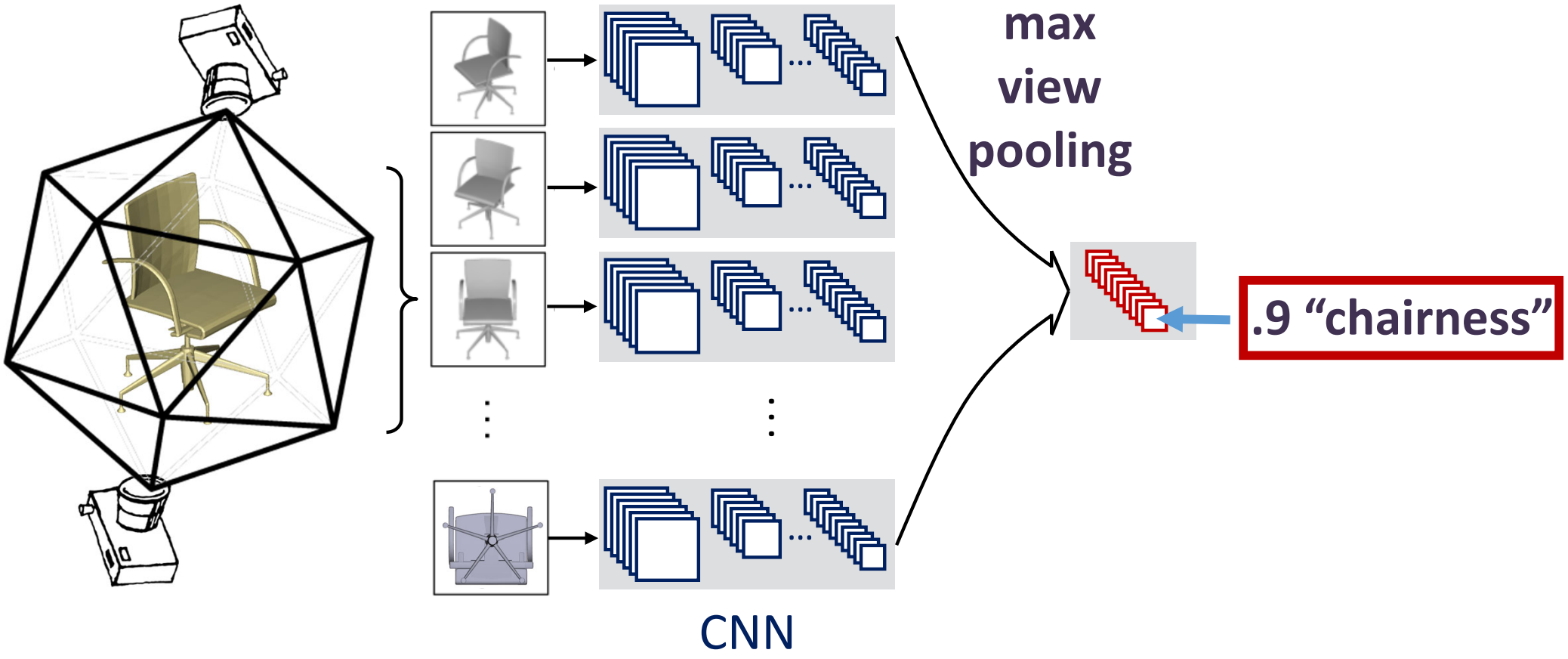
View Pooling



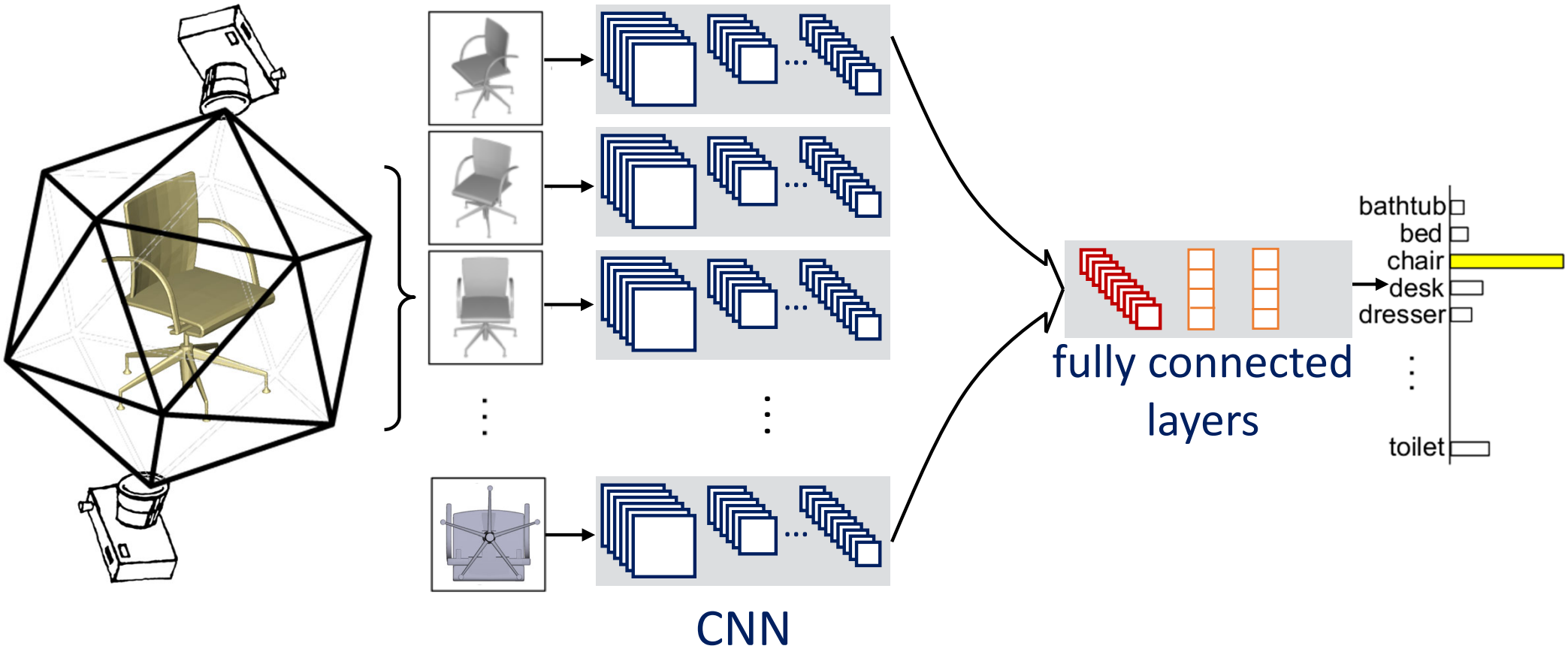
View Pooling



View Pooling

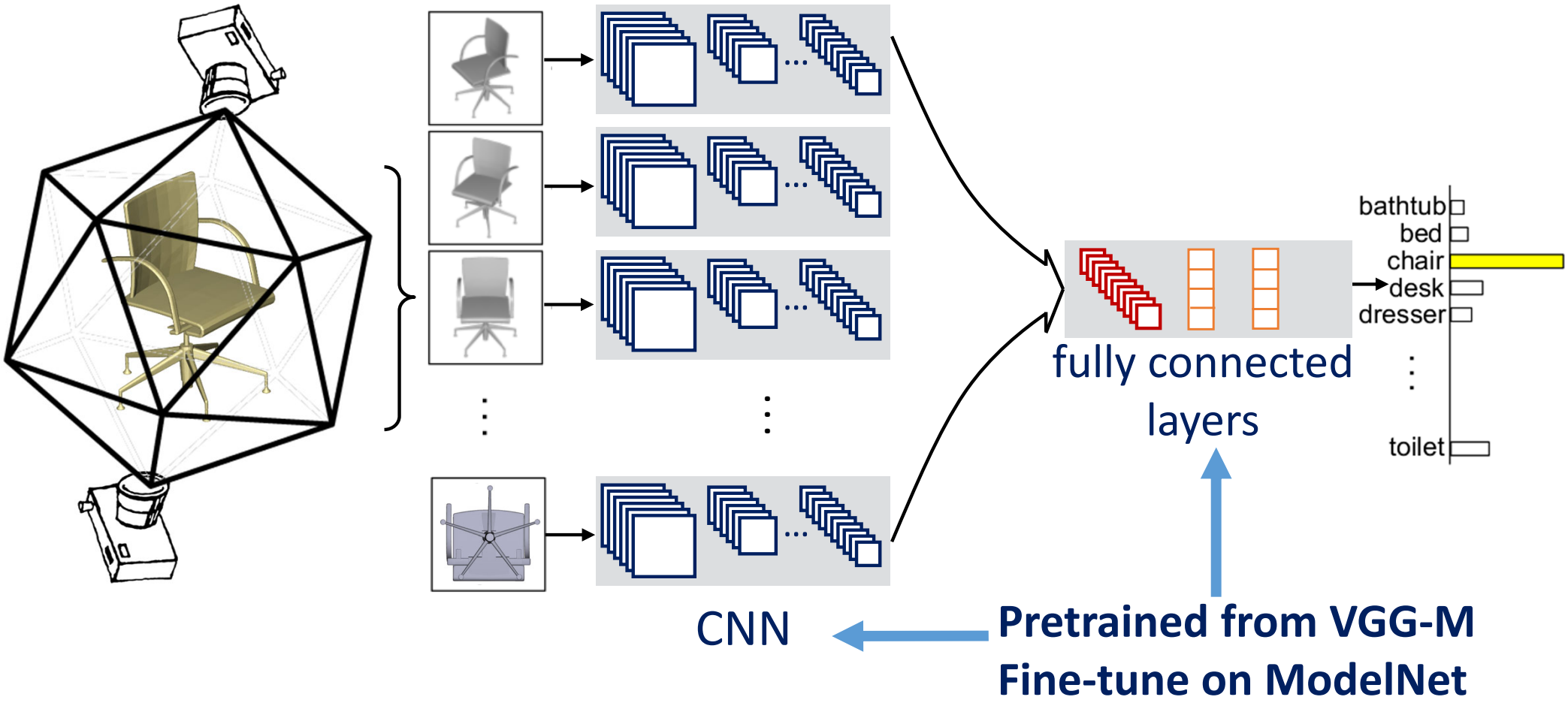


MVCNN architecture

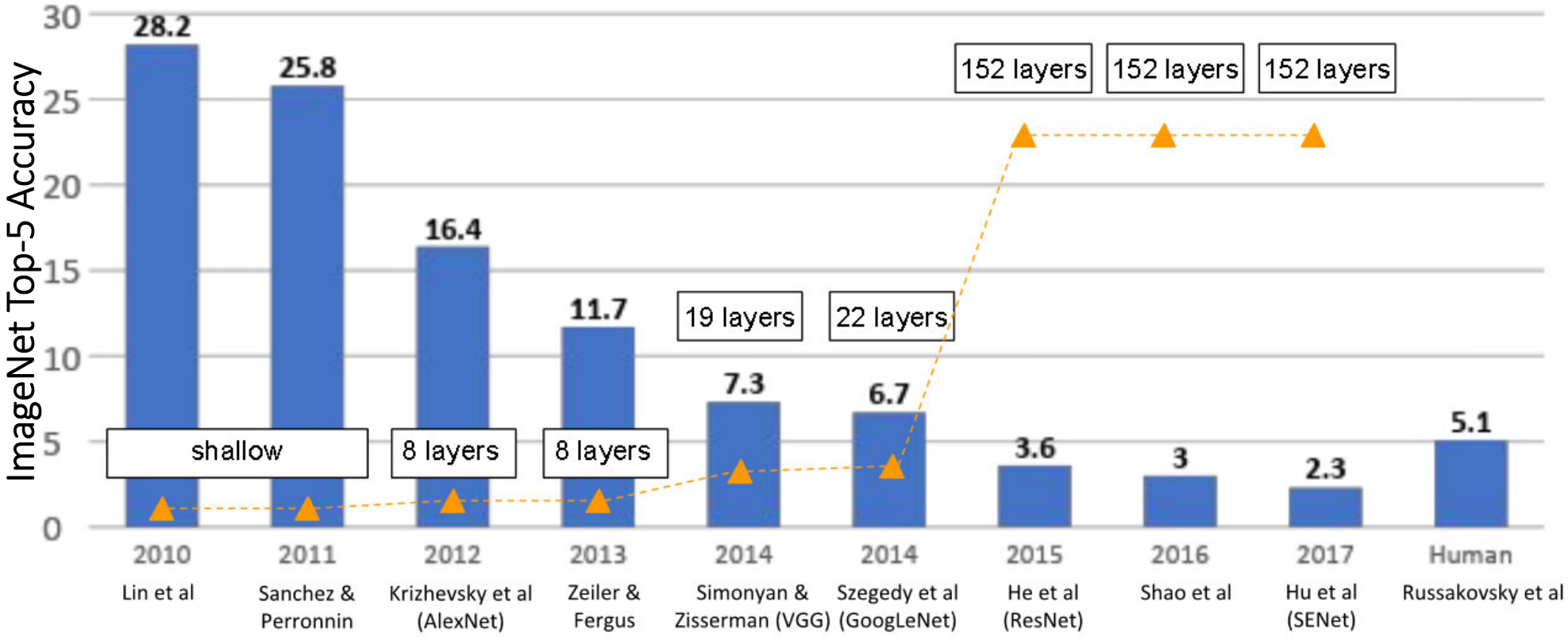


Su, Maji, Kalogerakis, Learned-Miller, ICCV 2015

Training


bathtub
bed
chair
desk
dresser
toilet
fully connected layers
CNN
Pretrained from VGG-M
Fine-tune on ModelNet

Since 2012, huge progress in 2D/3D recognition!

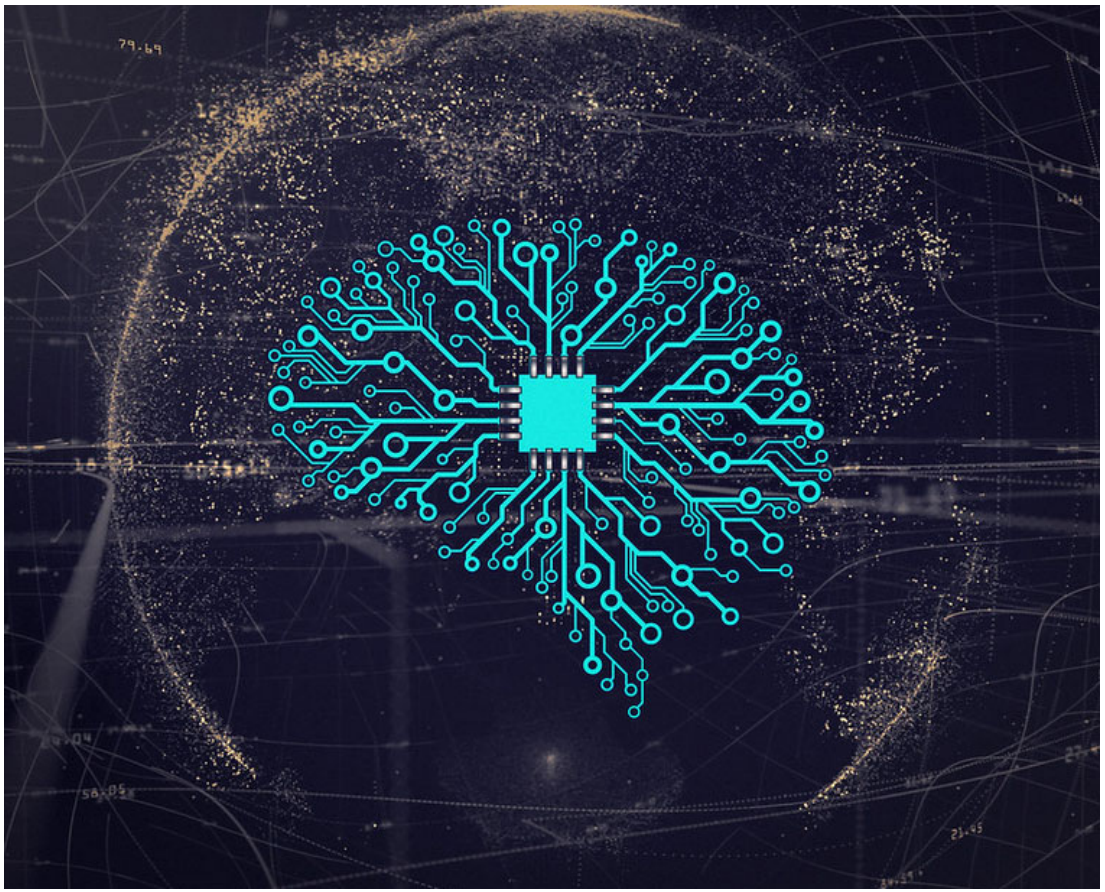


To achieve this progress...

Many things turned out to matter a lot:

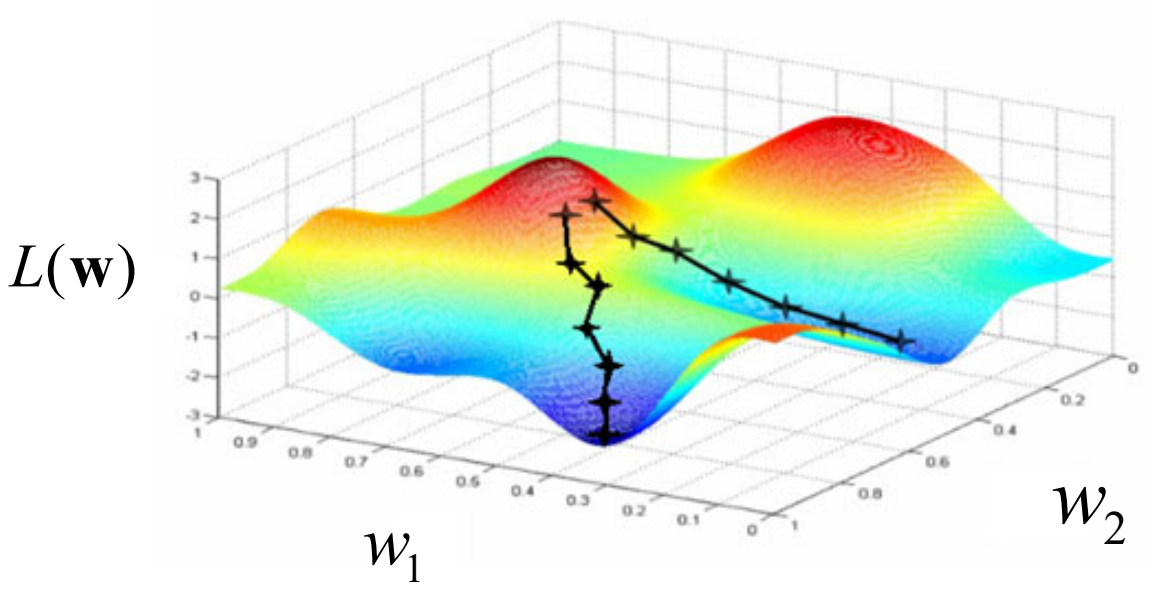
- How to initialize its parameters for training?
- How to train with many layers (e.g., hundreds of them)?
- How to prevent overfitting / underfitting?
- What should the network structure be?
- Should features be computed using local pixels only?

Part IV: Neural Networks Tricks and Tips



Intelligent Visual Computing
Evangelos Kalogerakis

Gradient Descent



Lots and lots of local minima in neural networks!

Yet, this does not work so easily...

- Optimization becomes difficult with **many layers**.

- Hard to diagnose and **debug malfunctions**.

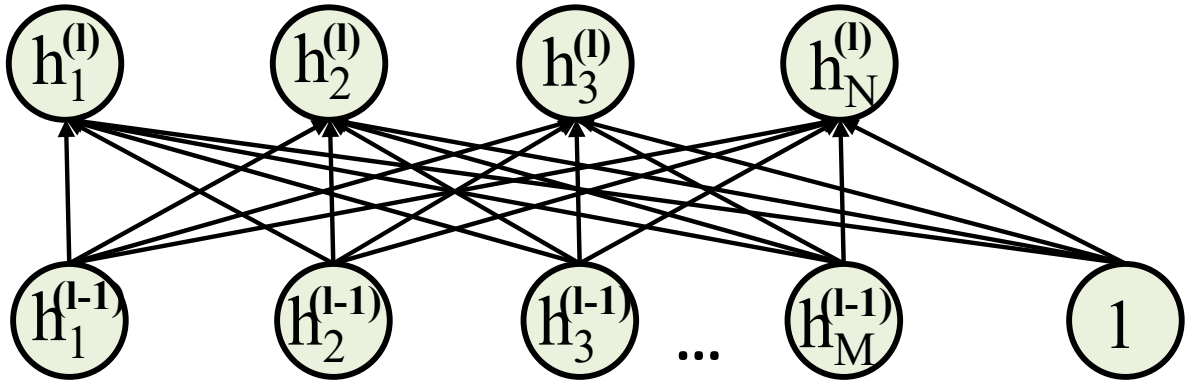
- **Many things turn out to matter**:
  - Initialization of parameters
  - Optimization procedure and hyper-parameters (step size)
  - Network structure

Initialization

Initialize filters/weights to small values. How "small" should these be?

Assume one linear layer with output: $h_n^{(l)} = \mathbf{w}_n \bullet \mathbf{h}^{(\mathbf{l-1})}$

$$Var[h_n^{(l)}] = M \cdot Var[w_{n,m}^{(l)}] Var[h_m^{(l-1)}]$$



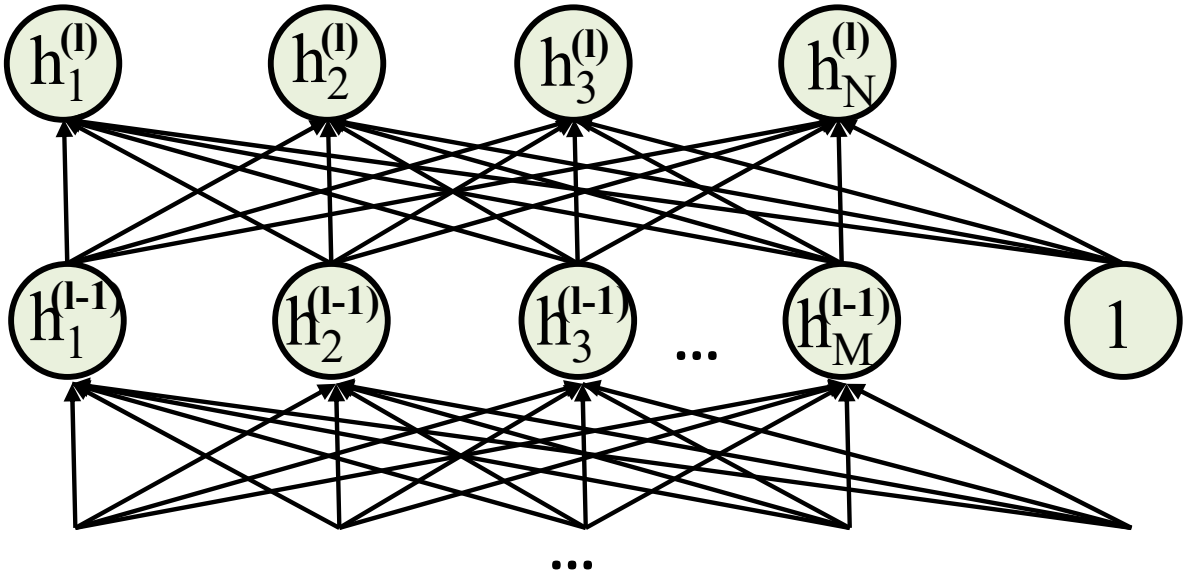
Sketch of a proof:
http://andyljones.tumblr.com/post/110998971763/an-explanation-of-xavier-initialization

Initialization

OK, but how “small” these random values should be?

The input to this layer depends on many previous layers…

$$Var[h_n^{(l)}] = \left(\prod_{\substack{layer \\ b=1\ldots l-1}} M^{(b)} \cdot Var[w_{n,m}^{(b)}]\right) Var[input]$$



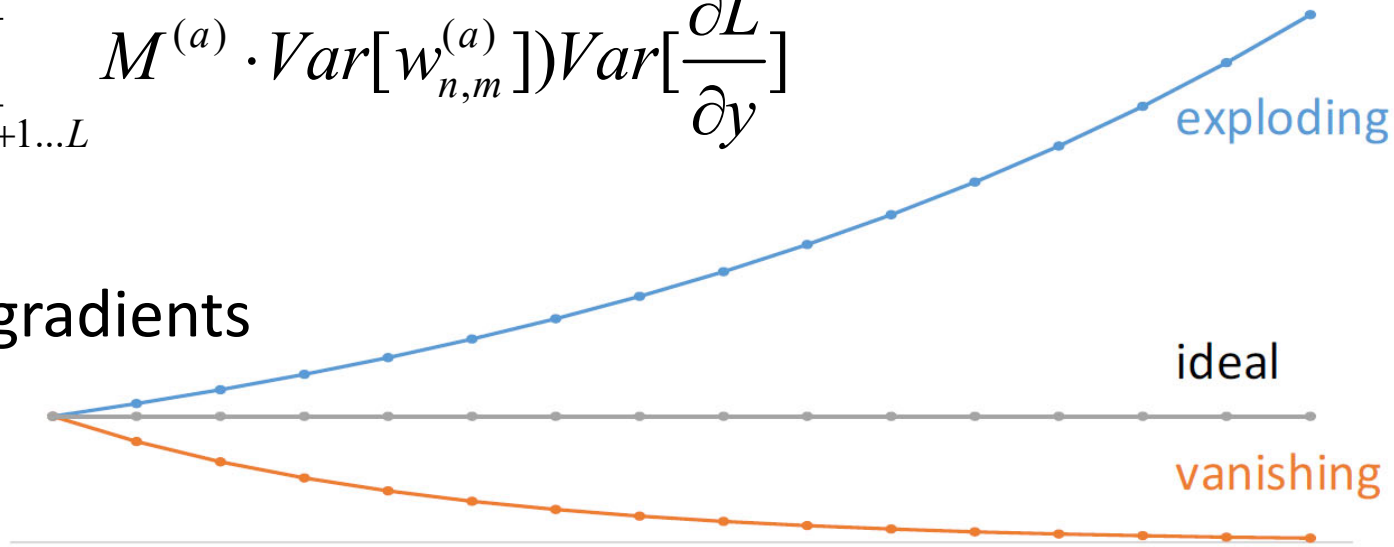
Initialization

Similar behavior for gradients!

$$Var[h_n^{(l)}] = \left(\prod_{b=1\ldots l-1}^{layer} M^{(b)} \cdot Var[w_{n,m}^{(b)}]\right) Var[input]$$

$$Var\left[\frac{\partial L}{h_n^{(l)}}\right] = \left(\prod_{a=l+1\ldots L}^{layer} M^{(a)} \cdot Var[w_{n,m}^{(a)}]\right) Var\left[\frac{\partial L}{\partial y}\right]$$

Both outputs & gradients can easily explode or vanish!!!



Initialization

Want: $M \cdot Var[w_{n,m}] = 1$

$N \cdot Var[w_{n,m}] = 1$

(M input nodes, N output nodes)

A trade-off: $Var[w_{n,m}] = \frac{2}{N+M}$

Gaussian dist. Initialization: $N(0,r^2)$ $r = \sqrt{\frac{2}{N+M}}$

Uniform dist. Initialization: [-r, r] $r = \sqrt{\frac{6}{N+M}}$

[Understanding the difficulty of training deep feedforward neural networks, Glorot & Bengio 2010]

Initialization (for ReLUs)

Biases are often set to 0 or small positive numbers e.g., 0.01 (to prevent ReLUs to get stuck at their negative part).

For the rest of the weights:

Gaussian dist. Initialization: $N(0,r^2)$

$$Var[w_{n,m}] = \frac{2}{N} \; or \; \frac{2}{M} \; or \; \frac{4}{N+M}$$

Uniform dist. initialization [-r, r]

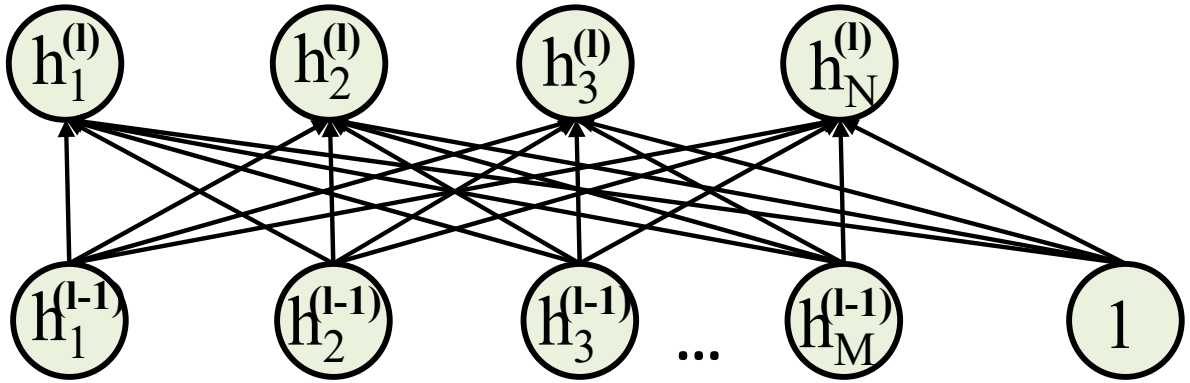
$$r = \sqrt{\frac{6}{N}} \; or \; \sqrt{\frac{6}{M}} \; or \; \sqrt{\frac{12}{N+M}}$$

Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification,He et al., 2015

Batch Normalization

During training, weights will change, variances will change, distributions of layer outputs can vary wildly!

$$Var[h_n^{(l)}] = M \cdot Var[w_{n,m}^{(l)}] Var[h_m^{(l-1)}]$$



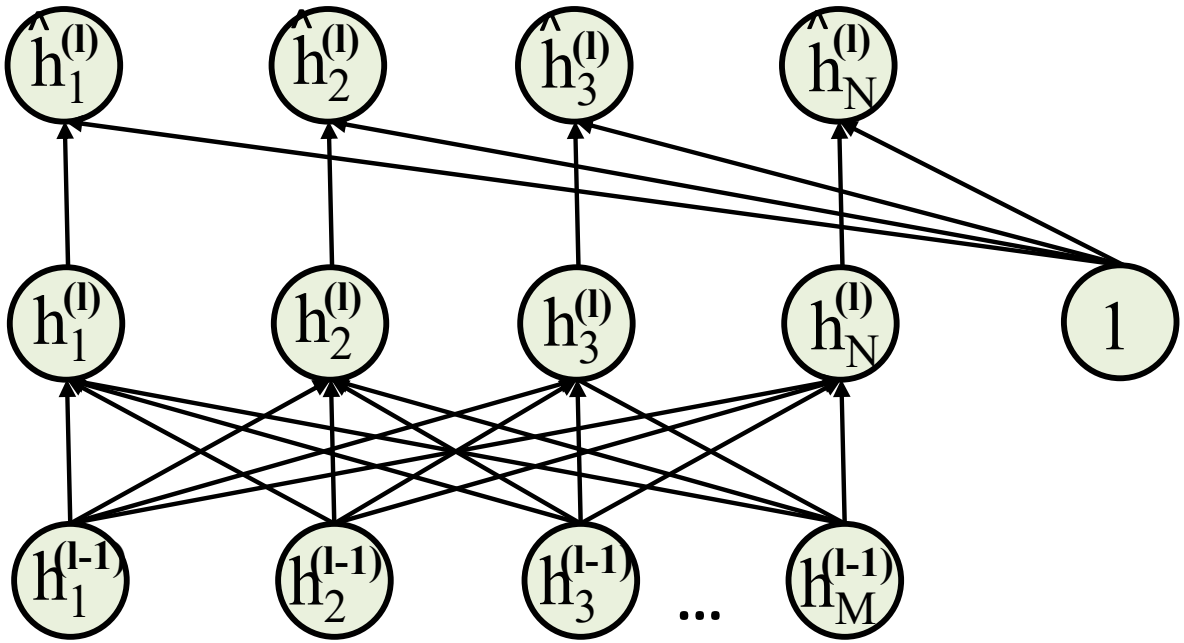
Let's explicitly fix the distributions of our nodes => Batch normalization

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, Ioffe & Szegedy, ICML 2015

Batch Normalization

Standardize outputs within each **batch**:

$$\hat{h}_n^{(l)} = \frac{h_n^{(l)} - E_{batch}[h_n^{(l)}]}{\sqrt{Var_{batch}[h_n^{(l)}] + \varepsilon}}$$



Practically implemented as another layer!

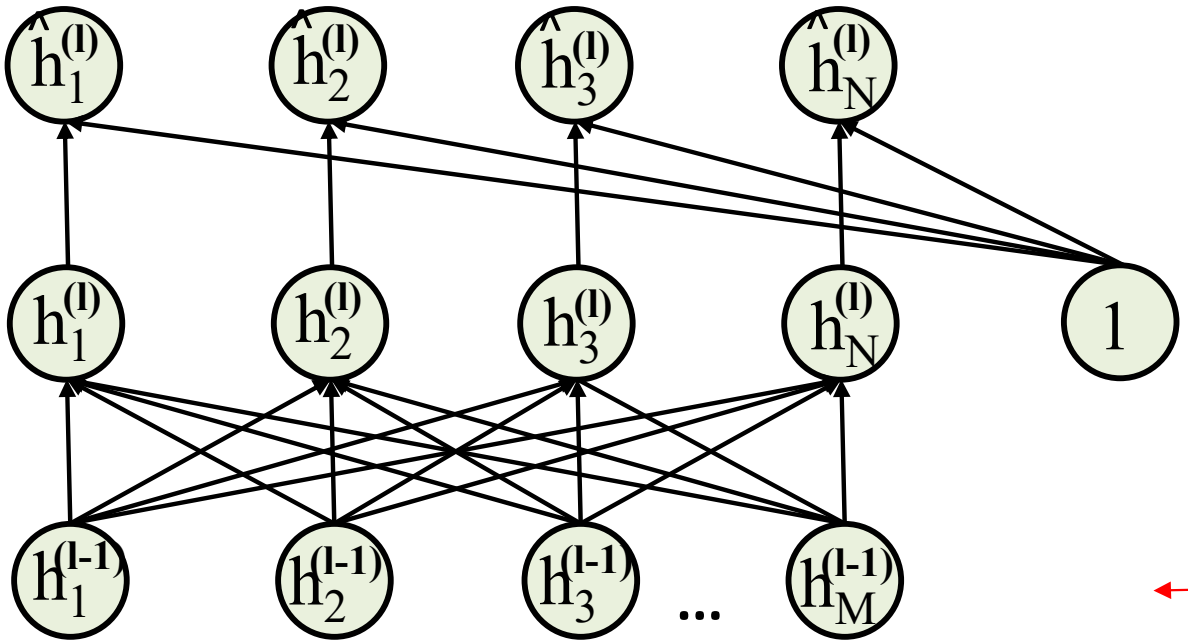
Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, Ioffe & Szegedy, ICML 2015

Batch Normalization

Standardize outputs within each **batch**, then learn to scale & shift them:

$$\hat{h}_n^{(l)} = \frac{h_n^{(l)} - E_{batch}[h_n^{(l)}]}{\sqrt{Var_{batch}[h_n^{(l)}] + \varepsilon}}$$

$$\hat{h}_n^{(l)} = \gamma_n^{(l)} \hat{h}_n^{(l)} + \beta_n^{(l)}$$



Practically implemented as another layer!

Parameters γ, β per node and learned

Bias = β
(no need to use a bias here)

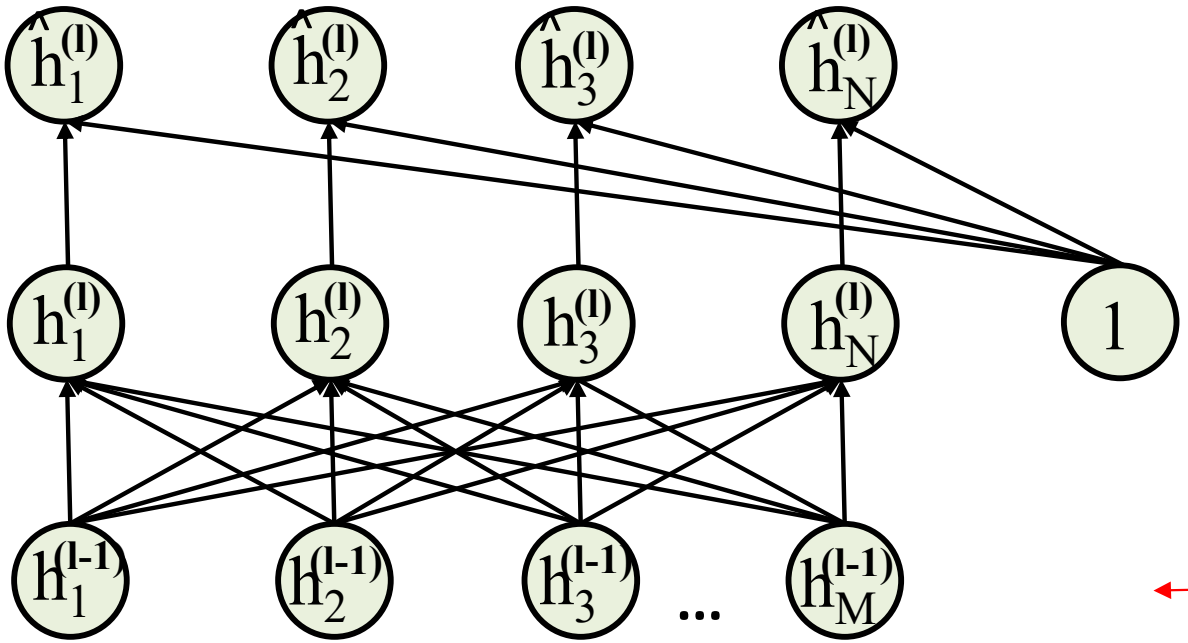
Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, Ioffe & Szegedy, ICML 2015

Batch Normalization

Standardize outputs within each **batch**, then learn to scale & shift them:

$$\hat{h}_n^{(l)} = \frac{h_n^{(l)} - E_{batch}[h_n^{(l)}]}{\sqrt{Var_{batch}[h_n^{(l)}] + \varepsilon}}$$

$$\hat{h}_n^{(l)} = \gamma_n^{(l)} \hat{h}_n^{(l)} + \beta_n^{(l)}$$



Practically implemented as another layer!

Parameters γ, β per node and learned

Bias = β
(no need to use a bias here)

... Yet, when batch size is 1, we can't compute the above statistics (or are unreliable for small batches!)

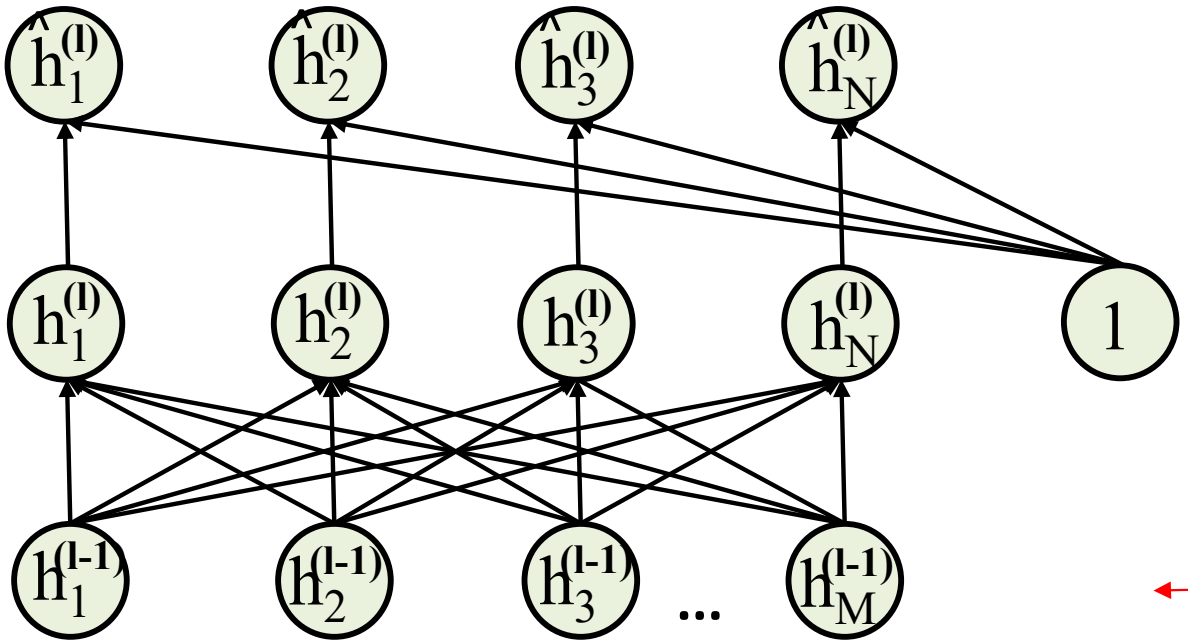
Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, Ioffe & Szegedy, ICML 2015

Layer Normalization

Standardize outputs within each **layer**, then learn to scale & shift them:

$$\hat{h}_n^{(l)} = \frac{h_n^{(l)} - E_{layer}[h^{(l)}]}{\sqrt{Var_{layer}[h^{(l)}] + \varepsilon}}$$

$$\hat{h}_n^{(l)} = \gamma_n^{(l)} \hat{h}_n^{(l)} + \beta_n^{(l)}$$



Practically implemented as another layer!

Parameters γ, β per node and learned

Bias = β
(no need to use a bias here)

Layer Normalization, Ba et al. 2015

Different Normalization Layers

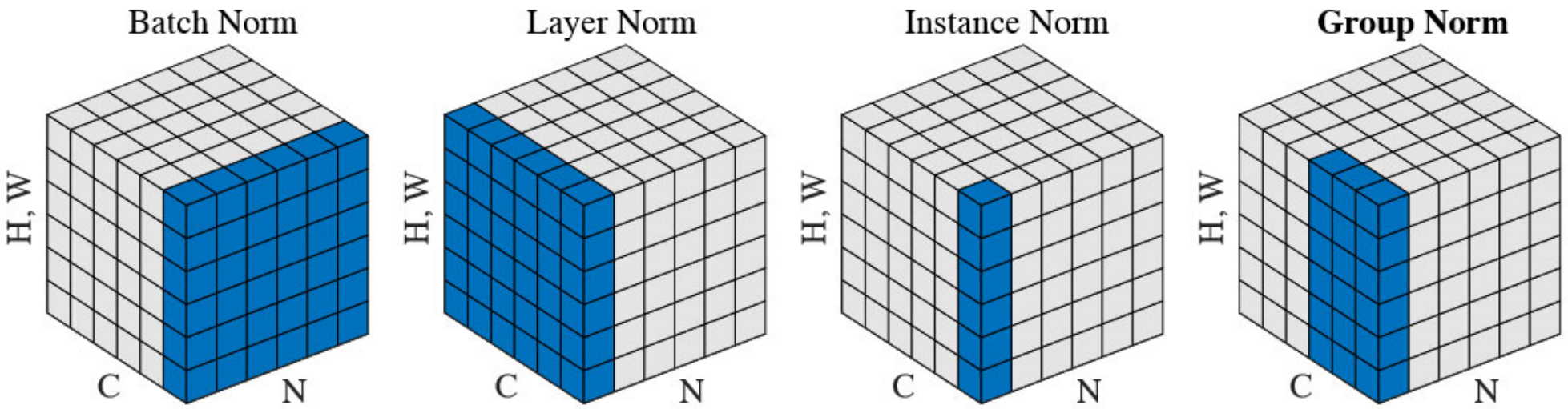


Figure 2. **Normalization methods**. Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

Group Normalization: https://arxiv.org/pdf/1803.08494.pdf

Momentum + regularization

Modify stochastic/batch gradient descent:

$$\text{Before}: \quad \Delta \mathbf{w} = \eta \nabla_{\mathbf{w}} L(\mathbf{w}), \quad \mathbf{w} = \mathbf{w} - \Delta \mathbf{w}$$

$$\text{With momentum}: \Delta \mathbf{w} = \mu \Delta \mathbf{w}_{previous} + \eta \nabla_{\mathbf{w}} L(\mathbf{w}), \quad \mathbf{w} = \mathbf{w} - \Delta \mathbf{w}$$

"Smooth" estimate of gradient from iterations:

- High-curvature directions cancel out, low-curvature directions "add up" & accelerate. Often set to with μ=0.9

Momentum + regularization

Modify stochastic/batch gradient descent:

$$\text{Before}: \quad \Delta \mathbf{w} = \eta \nabla_{\mathbf{w}} L(\mathbf{w}), \quad \mathbf{w} = \mathbf{w} - \Delta \mathbf{w}$$

$$\text{With momentum}: \Delta \mathbf{w} = \mu \Delta \mathbf{w}_{previous} + \eta \nabla_{\mathbf{w}} L(\mathbf{w}), \quad \mathbf{w} = \mathbf{w} - \Delta \mathbf{w}$$

"Smooth" estimate of gradient from iterations:

- High-curvature directions cancel out, low-curvature directions "add up" & accelerate. Often set to with μ=0.9

Use **weight decay** to discourage large weights:

$$\mathbf{w} = \mathbf{w} - \Delta \mathbf{w} - \eta \lambda \mathbf{w}$$

Related to adding a penalty to the loss: $L(\mathbf{w}) + \frac{1}{2} \lambda \, || \, \mathbf{w} \, ||^2$

Momentum + regularization

Modify stochastic/batch gradient descent:

$$\text{Before}: \quad \Delta \mathbf{w} = \eta \nabla_{\mathbf{w}} L(\mathbf{w}), \quad w = w - \Delta \mathbf{w}$$

$$\text{With momentum}: \Delta \mathbf{w} = \mu \Delta \mathbf{w}_{previous} + \eta \nabla_{\mathbf{w}} L(\mathbf{w}), \quad w = w - \Delta \mathbf{w}$$

"Smooth" estimate of gradient from iterations:

- High-curvature directions cancel out, low-curvature directions "add up" & accelerate. Often set to with μ=0.9

- **Other SGD variants:** Adagrad, Adam, AdamW

See also:

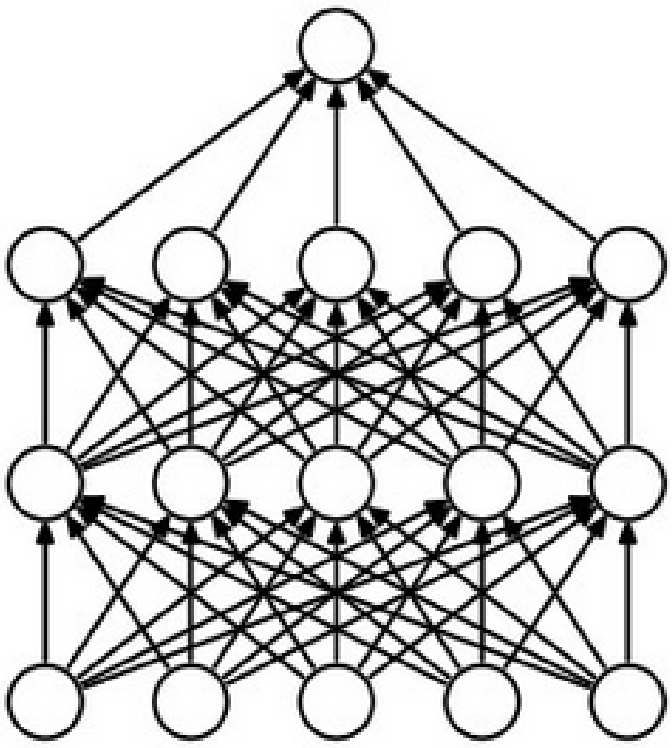
https://en.wikipedia.org/wiki/Stochastic_gradient_descent

https://ruder.io/optimizing-gradient-descent/

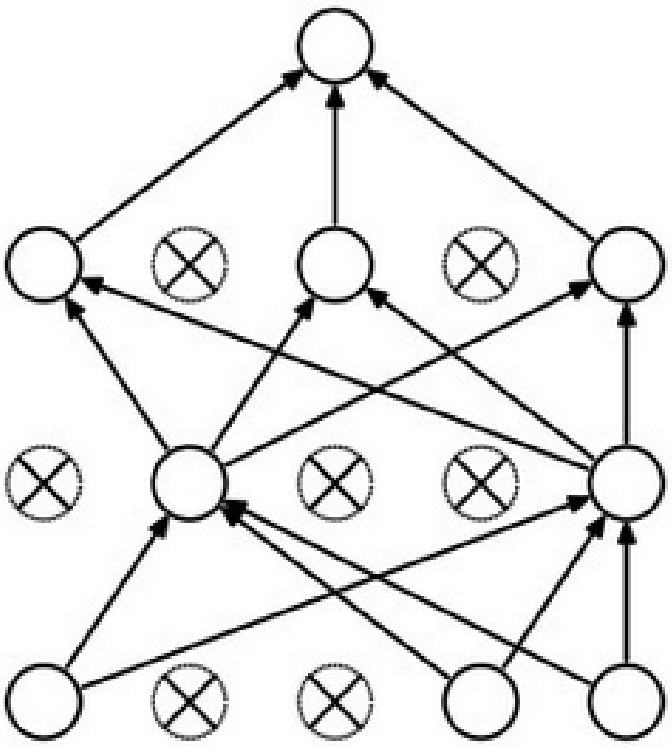
https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3

Dropout for preventing co-adapaption

While **<u>training</u>**, keep a neuron active with some probability (a hyperparameter). Otherwise, set to zero => **removes connections**



(a) Standard Neural Net



(b) After applying dropout.

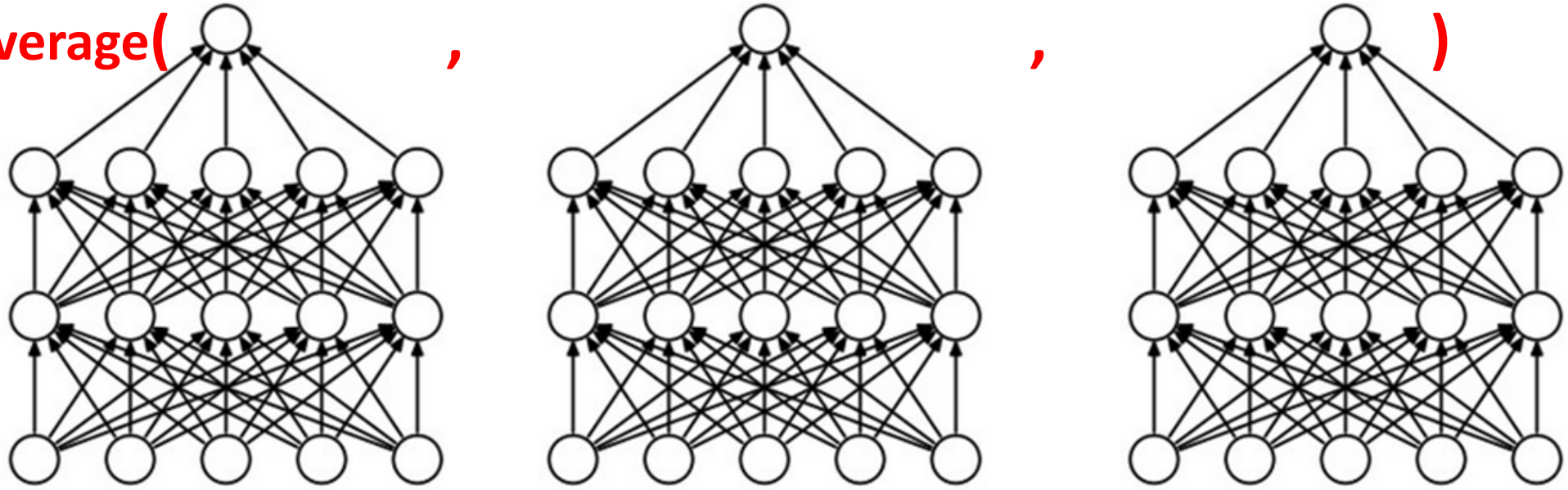
Srivastava et al., Dropout: A Simple Way to Prevent Neural Networks from Overfitting. 2014

Ensemble Averaging

Train multiple nets and average their predictions!

Remember: random initialization, SGD, dropout… yield different parameters (different local minima) per each net trained on the same data

Average(, ,)

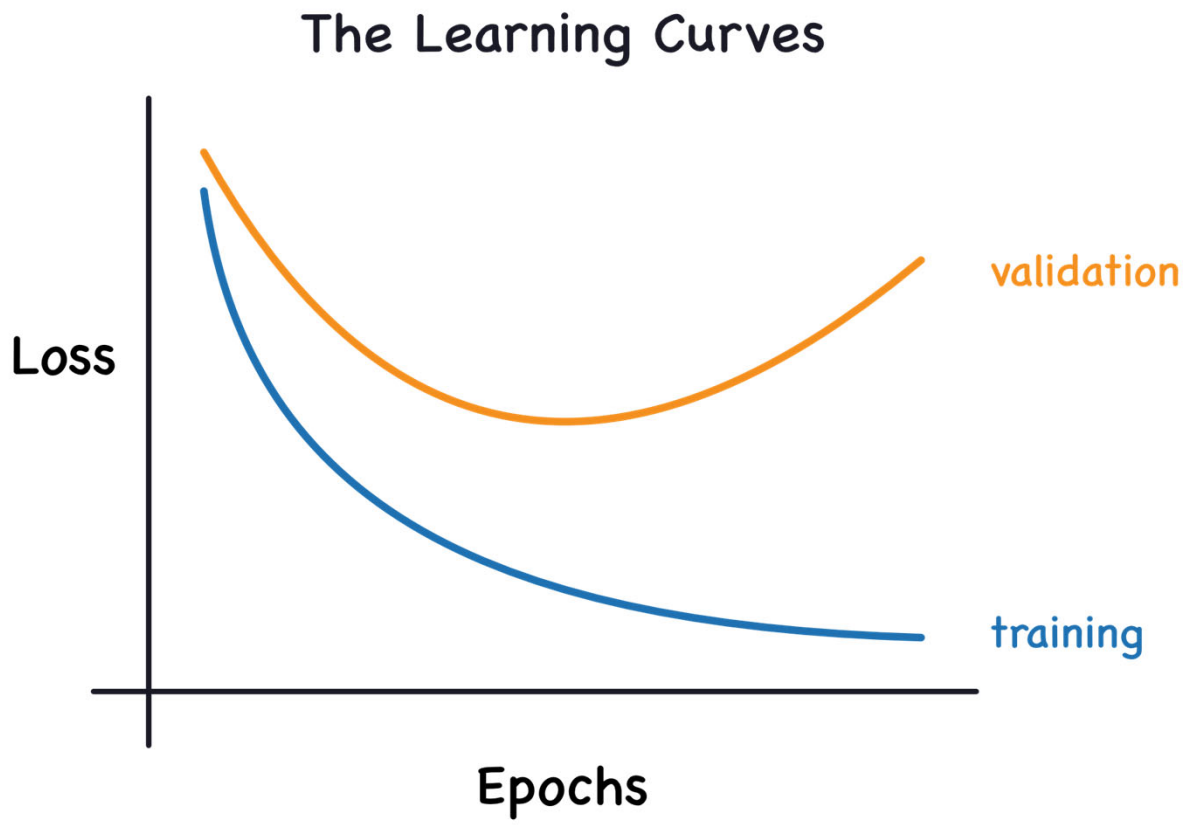


Net 1 **Net 2** **Net 3**

Hashem, S. "Optimal linear combinations of neural networks." Neural Networks 10, no. 4 (1997)

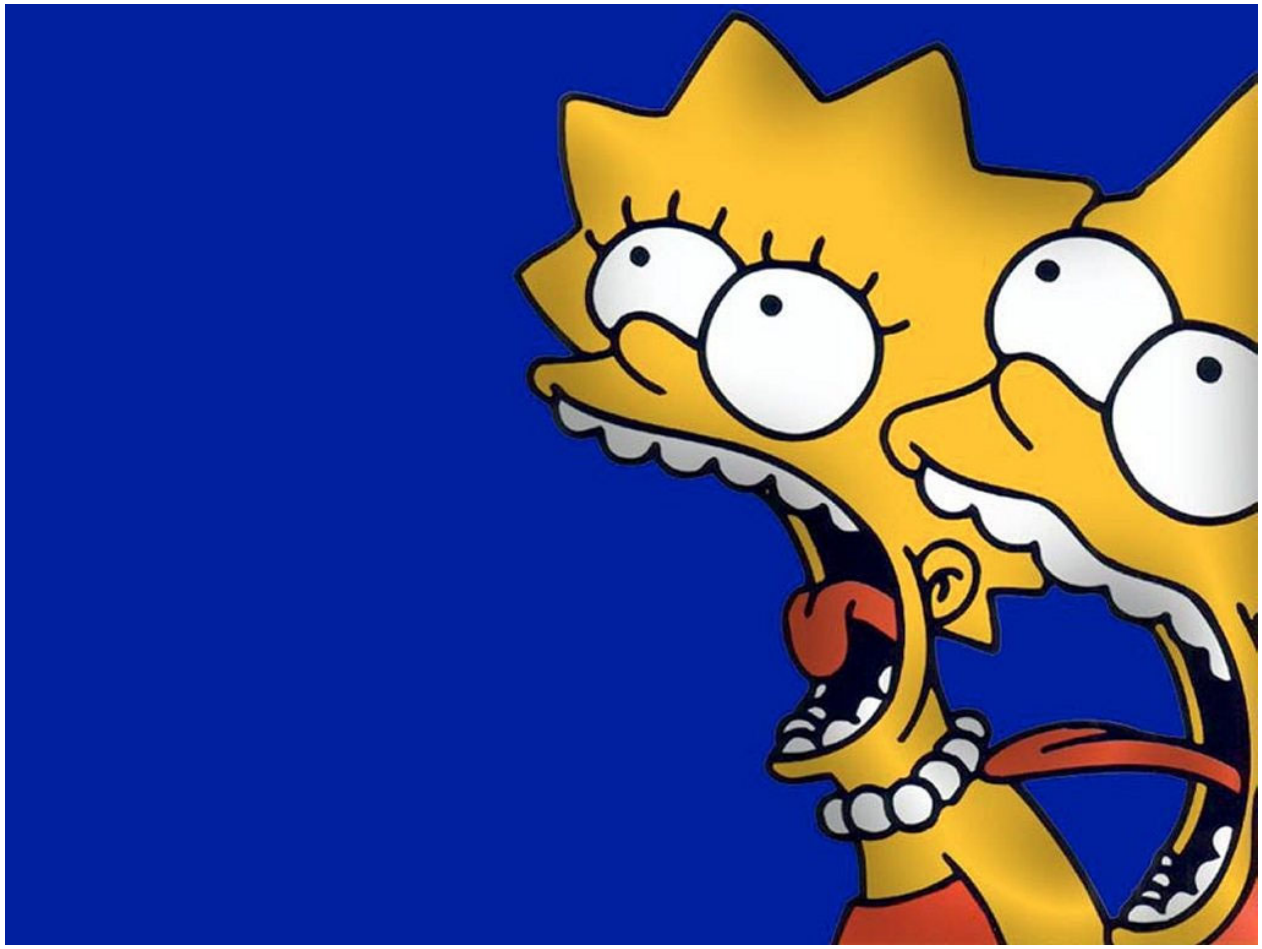
Baby-sitting

Some **baby-sitting** is necessary! Track loss function during training and also check loss / accuracy in the validation set at the same time!

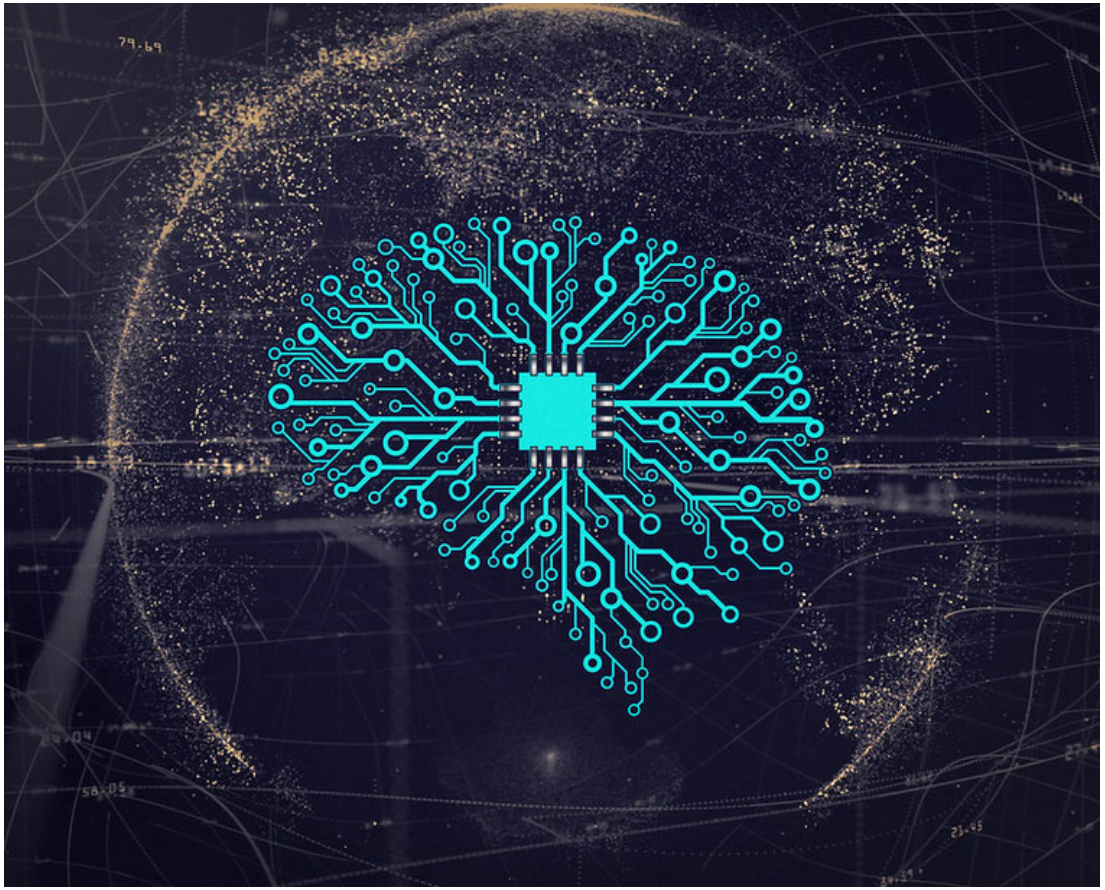


https://www.kaggle.com/code/ryanholbrook/overfitting-and-underfitting/tutorial

Yet, things will not still work well!



Part V: Modern Neural Networks



Intelligent Visual Computing
Evangelos Kalogerakis

Revolution of Depth

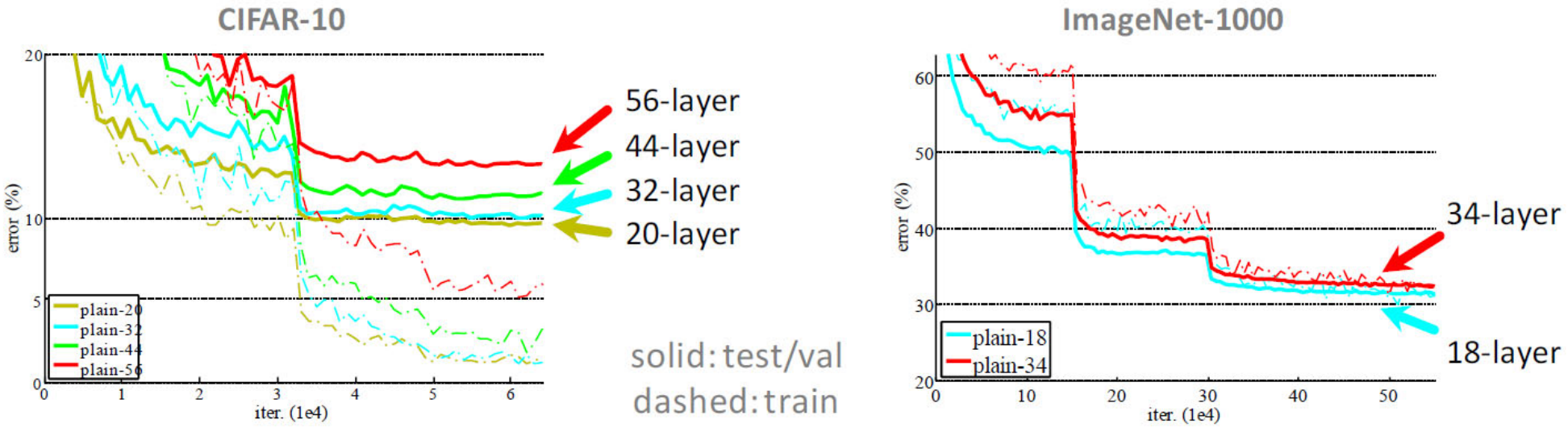
AlexNet, 8 layers
(ILSVRC 2012)

VGG, 19 layers
(ILSVRC 2014)

ResNet, 152 layers
(ILSVRC 2015)

Is learning better networks as simple as stacking more layers?

The deeper, the better?

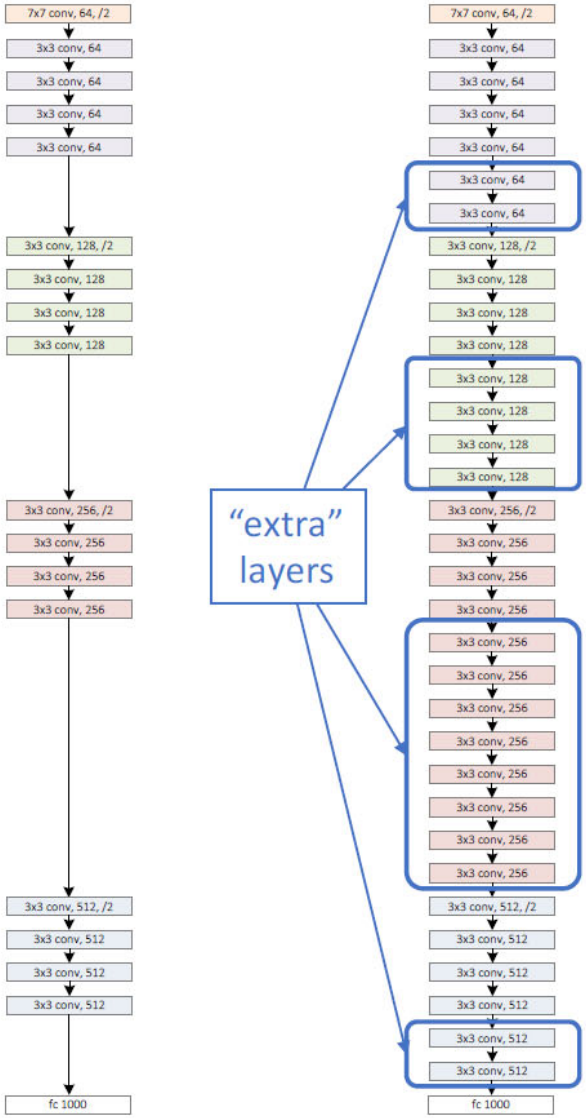


Stacking more layers in “plain” nets results in **higher training error (and test error)**

A general phenomenon, observed in many datasets

ResNets basic idea

a shallower model (18 layers)



a deeper counterpart (34 layers)

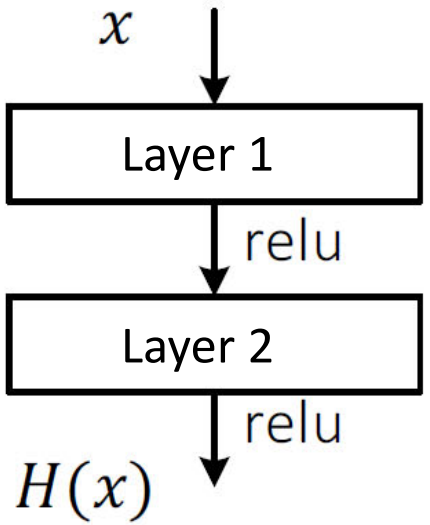
- Richer solution space
- A deeper model should not have **higher training error**
- A solution *by construction*:
  - original layers: copied from a learned shallower model
  - extra layers: set as identity
  - at least the same training error

Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". CVPR 2016.

ResNets basic idea

- Plain net

any two stacked layers



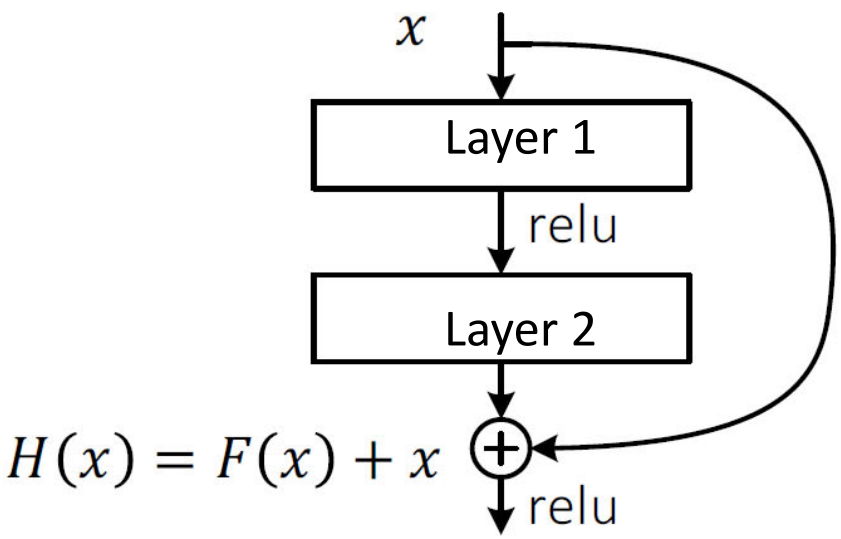
$H(x)$ is any desired mapping,

hope the 2 weight layers fit $H(x)$

Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". CVPR 2016.

ResNets basic idea

- Residual net



$H(x)$ is any desired mapping,

~~hope the 2 weight layers fit $H(x)$~~

hope the 2 weight layers fit $F(x)$

let $H(x) = F(x) + x$

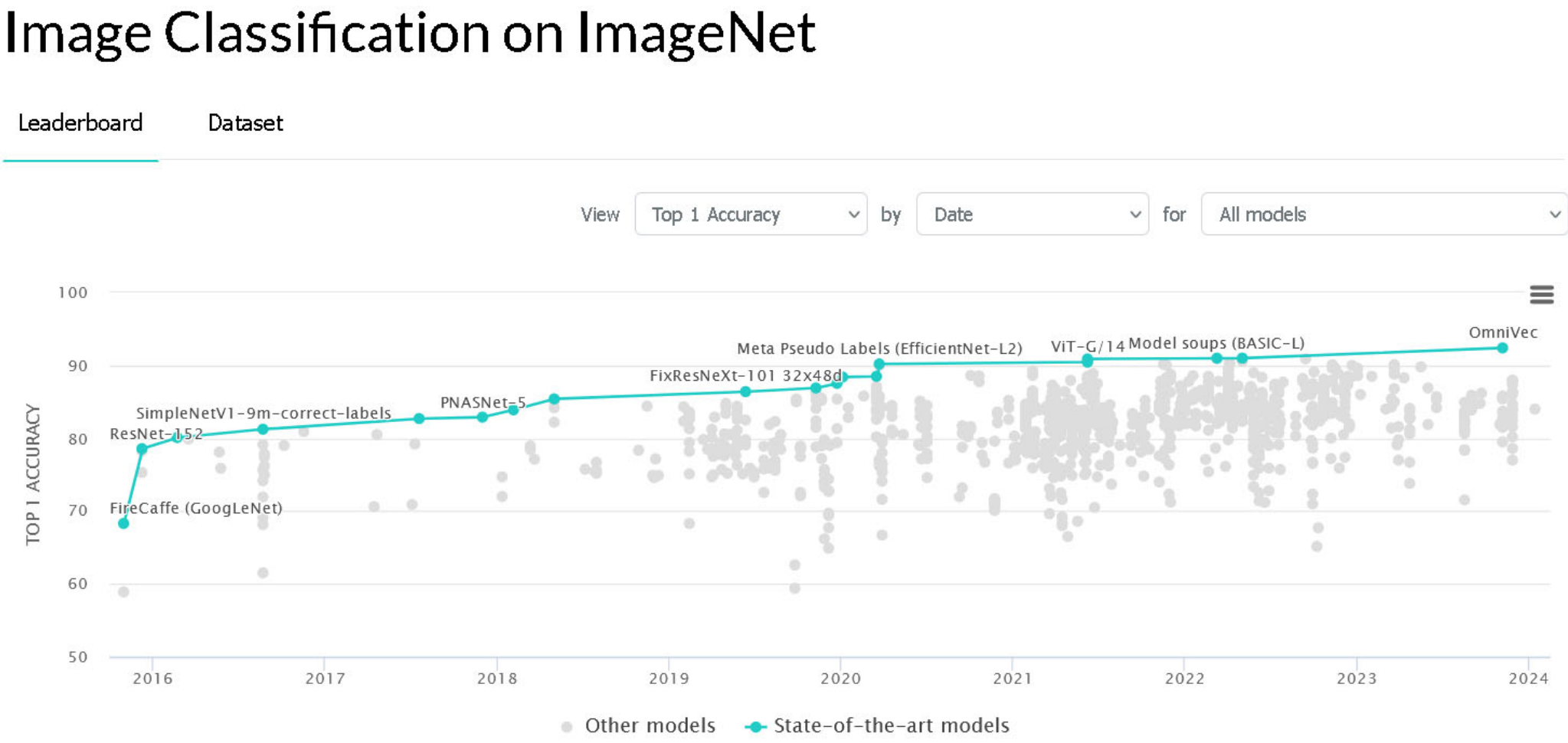
- If identity were optimal, easy to set weights as 0
- If optimal mapping is closer to identity, easier to find small fluctuations

Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". CVPR 2016.

Since 2012, huge progress in 2D/3D recognition!

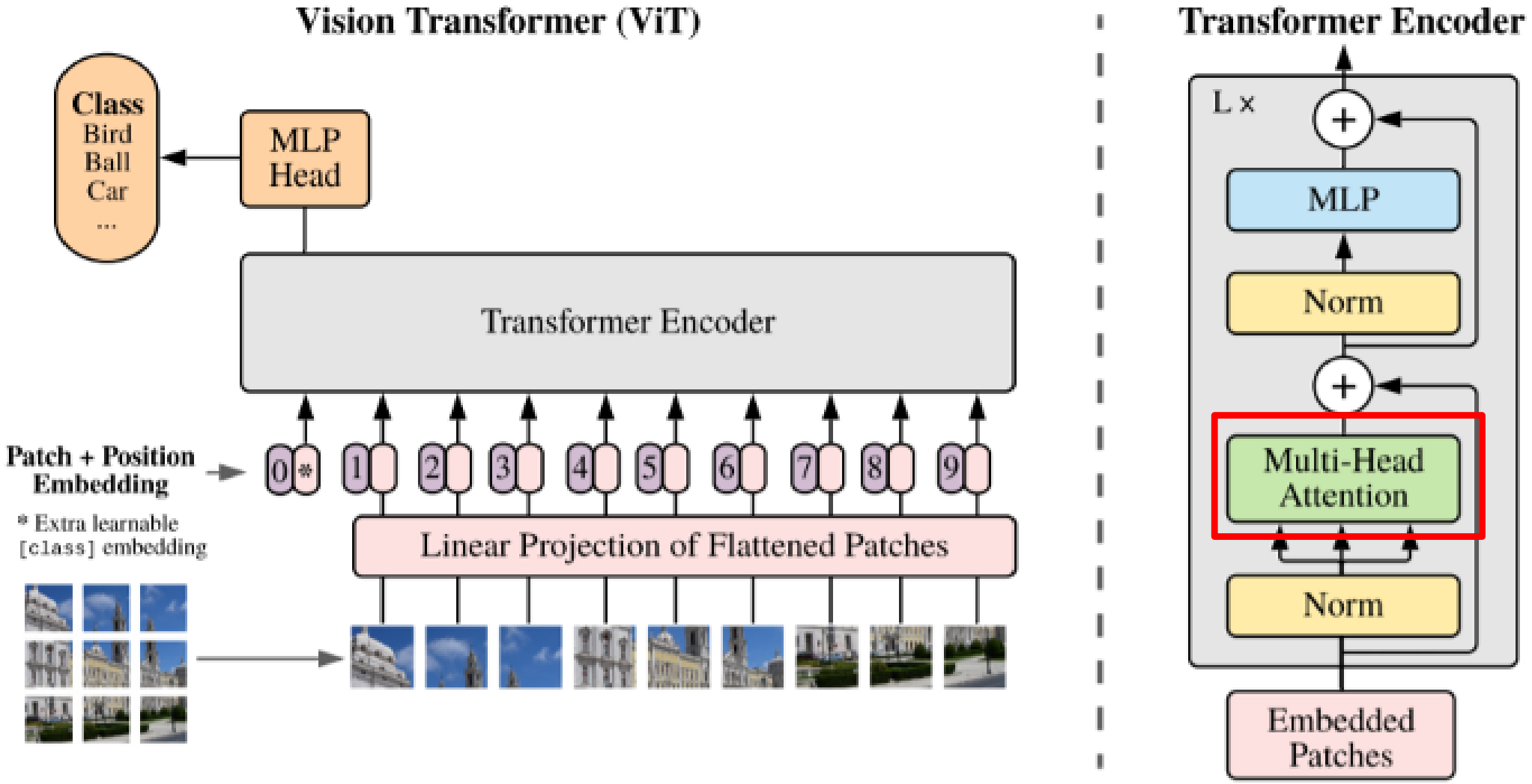
OUTDATED

From ResNets to ViTs



https://paperswithcode.com/sota/image-classification-on-imagenet

Vision Transformers



https://amaarora.github.io/posts/2021-01-18-ViT.html

https://www.youtube.com/watch?v=TrdevFK_am4

Dosovitskiy, ICLR 2021, https://github.com/google-research/vision_transformer

Attention vs Convolution

Convolution has been extremely popular in 2D/3D vision:

- ability to exploit local dependencies in the input data
- highly parallelizable / efficient to compute on GPUs

Capturing long-range interactions between pixels, points, voxels... is not trivial with convolution

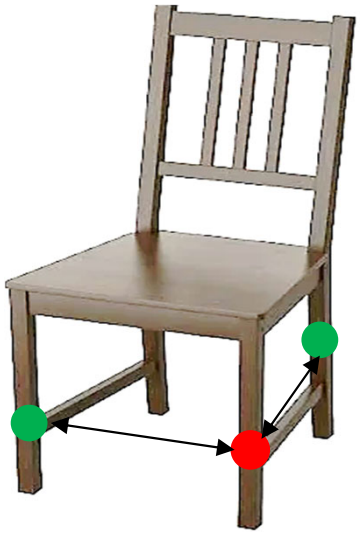
=> Let's see how attention works!

Attention

Help encoder look at other pixels/patches in image while encoding a pixel/patch due to various relations that may exist in objects or scenes in general

Attention

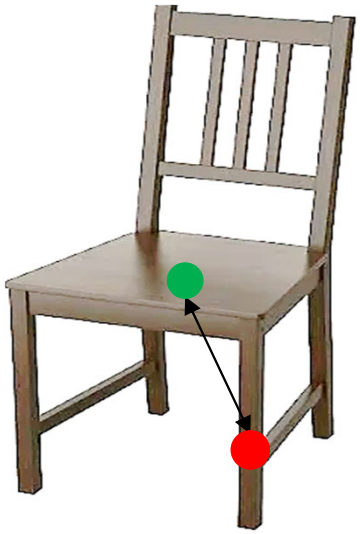
Help encoder look at other pixels/patches in image while encoding a pixel/patch due to various relations that may exist in objects or scenes in general



e.g., symmetries

Attention

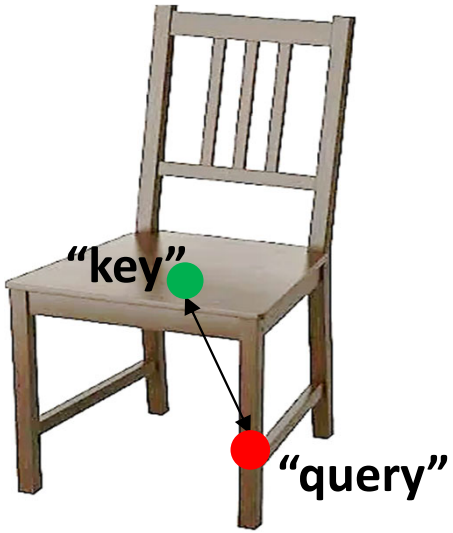
Help encoder look at other pixels/patches in image while encoding a pixel/patch due to various relations that may exist in objects or scenes in general



... or any pixel/patch relations may help recognize objects

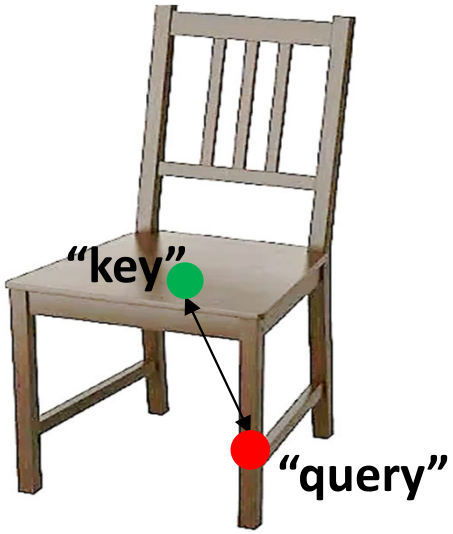
Attention

How much a pixel/patch (query) is related to others (keys)?



Attention

How much a pixel/patch (query) is related to others (keys)?



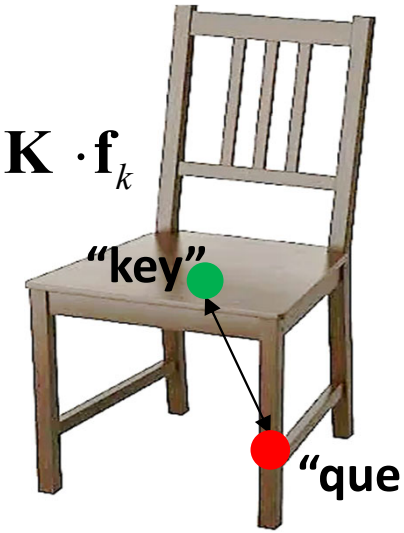
$$q(\mathbf{f}_q) = \mathbf{Q} \cdot \mathbf{f}_q$$

A linear transformation on the input feature vector of the query pixel/patch (e.g., MLP on input point/patch raw features)

Attention

How much a pixel/patch (query) is related to others (keys)?

$$k(\mathbf{f}_k) = \mathbf{K} \cdot \mathbf{f}_k$$



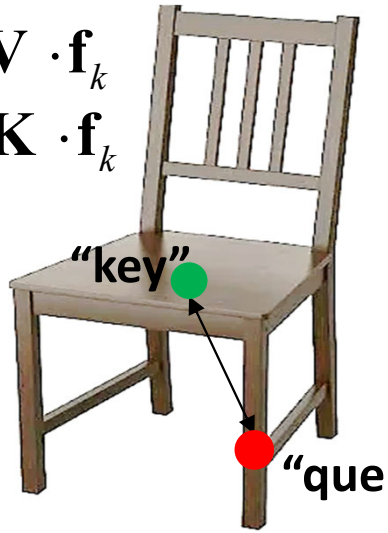
Another transformation on the input feature vector of the key point

$$q(\mathbf{f}_q) = \mathbf{Q} \cdot \mathbf{f}_q$$

Attention

How much a pixel/patch (query) is related to others (keys)?

$$v(\mathbf{f}_k) = \mathbf{V} \cdot \mathbf{f}_k$$
$$k(\mathbf{f}_k) = \mathbf{K} \cdot \mathbf{f}_k$$



One more transformation on feature vector of the key point i.e., we have a "key-value" pair

$$q(\mathbf{f}_q) = \mathbf{Q} \cdot \mathbf{f}_q$$

Attention

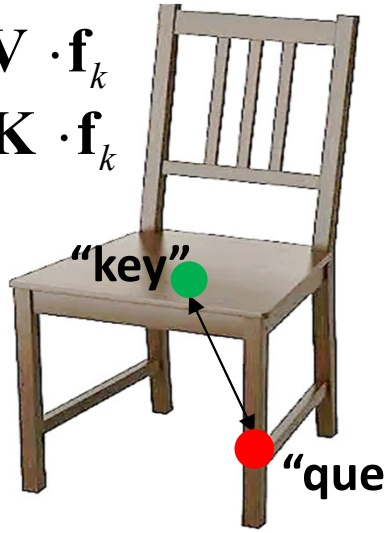
How much a pixel/patch (query) is related to others (keys)?

Attention "score": $a(\mathbf{f}_q, \mathbf{f}_k) = k(\mathbf{f}_k) \cdot q(\mathbf{f}_q)$

(dot product between key-query vectors)

$v(\mathbf{f}_k) = \mathbf{V} \cdot \mathbf{f}_k$

$k(\mathbf{f}_k) = \mathbf{K} \cdot \mathbf{f}_k$



$q(\mathbf{f}_q) = \mathbf{Q} \cdot \mathbf{f}_q$

Attention

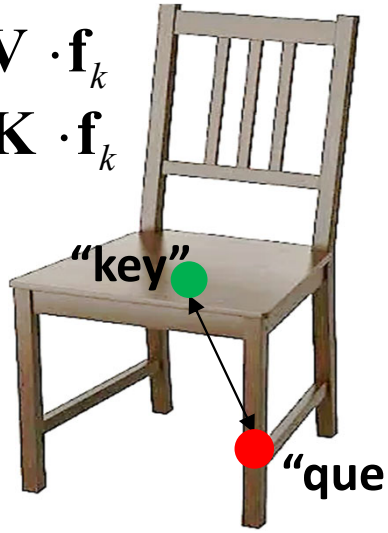
How much a pixel/patch (query) is related to others (keys)?

Attention "score": $\hat{a}(\mathbf{f}_q, \mathbf{f}_k) = \frac{\exp\{ a(\mathbf{f}_q, \mathbf{f}_k) \}}{\sum_{k'} \exp\{ a(\mathbf{f}_q, \mathbf{f}_{k'}) \}}$

(i.e., use softmax)

$v(\mathbf{f}_k) = \mathbf{V} \cdot \mathbf{f}_k$

$k(\mathbf{f}_k) = \mathbf{K} \cdot \mathbf{f}_k$



$q(\mathbf{f}_q) = \mathbf{Q} \cdot \mathbf{f}_q$

Attention

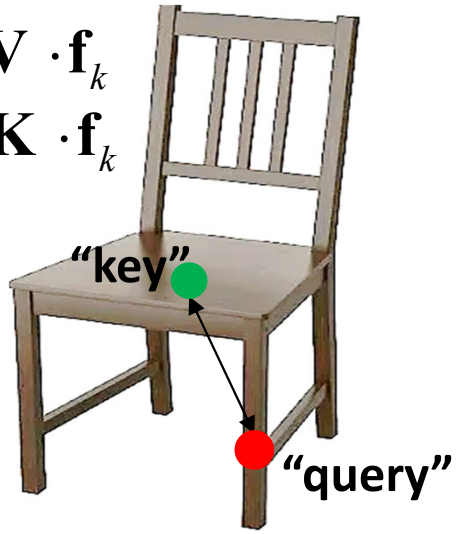
How much a pixel/patch (query) is related to others (keys)?

Attention "score": $\hat{a}(\mathbf{f}_q, \mathbf{f}_k) = \frac{\exp\{ a(\mathbf{f}_q, \mathbf{f}_k) \}}{\sum_{k'} \exp\{ a(\mathbf{f}_q, \mathbf{f}_{k'}) \}}$

(i.e., use softmax)

$v(\mathbf{f}_k) = \mathbf{V} \cdot \mathbf{f}_k$

$k(\mathbf{f}_k) = \mathbf{K} \cdot \mathbf{f}_k$



$q(\mathbf{f}_q) = \mathbf{Q} \cdot \mathbf{f}_q$

Problem: As the number of dimensions D of the input feature vector gets large, the variance of the dot product increased ...the input to softmax gets high values... the softmax is too peaked ...
=> tiny gradients

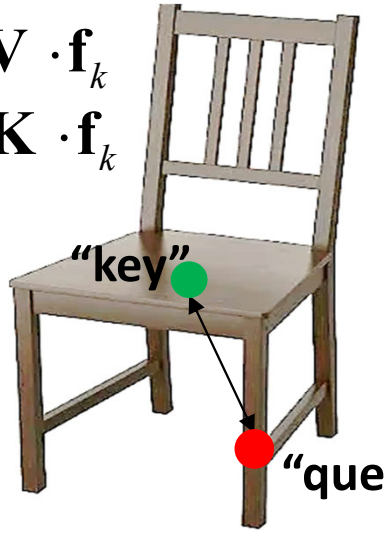
Scaled dot-product attention

How much a pixel/patch (query) is related to others (keys)?

Attention "score": (i.e., use softmax)

$$\hat{a}(\mathbf{f}_q, \mathbf{f}_k) = \frac{\exp\{ a(\mathbf{f}_q, \mathbf{f}_k) / \sqrt{D} \}}{\sum_{k'} \exp\{ a(\mathbf{f}_q, \mathbf{f}_{k'}) / \sqrt{D} \}}$$

$$v(\mathbf{f}_k) = \mathbf{V} \cdot \mathbf{f}_k$$
$$k(\mathbf{f}_k) = \mathbf{K} \cdot \mathbf{f}_k$$



$$q(\mathbf{f}_q) = \mathbf{Q} \cdot \mathbf{f}_q$$

https://ai.stackexchange.com/questions/41861/why-use-a-square-root-in-the-scaled-dot-product

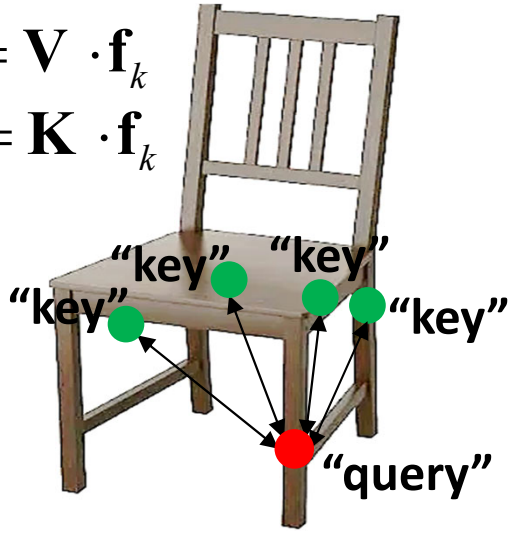
Vaswani, Attention Is All You Need, 2017

New feature representations

Compute a new encoding for query pixel/patch as a **weighted sum of values of key points:**

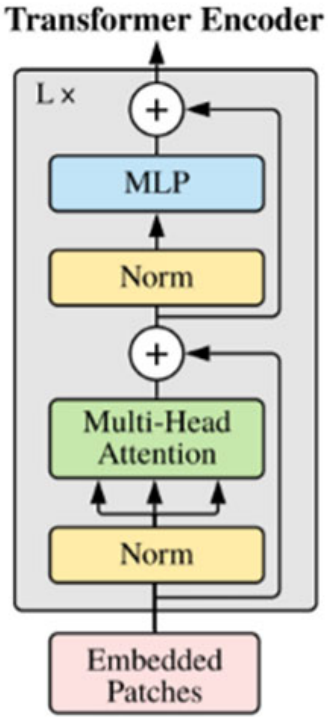
$$New\ features: \mathbf{f}_q' = \sum_k \hat{a}(\mathbf{f}_q, \mathbf{f}_k)\, v(\mathbf{f}_k)$$

$$v(\mathbf{f}_k) = \mathbf{V} \cdot \mathbf{f}_k$$
$$k(\mathbf{f}_k) = \mathbf{K} \cdot \mathbf{f}_k$$



$$q(\mathbf{f}_q) = \mathbf{Q} \cdot \mathbf{f}_q$$

... these can be added back to the input feature vector as a residual and further processed by an MLP (fully connected layer)



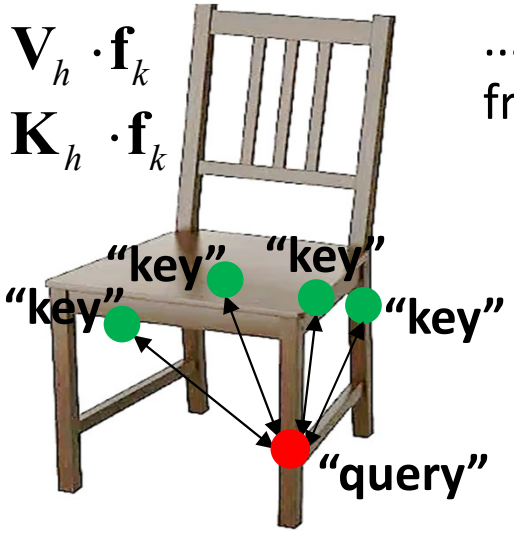
Vaswani, Attention Is All You Need, 2017

Multi-head Attention

Learn different query, key, value transformations for each attention "head".

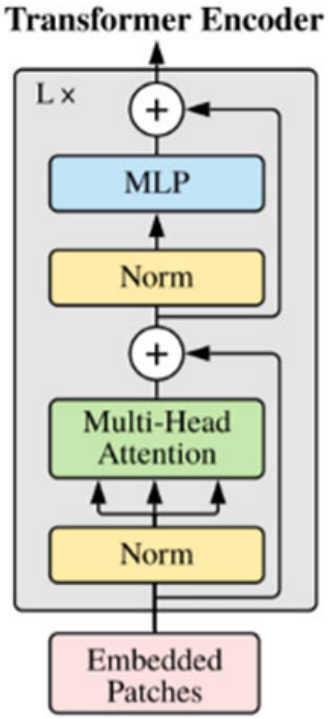
$$Multi-head\ \ attention: \mathbf{f}_{q,h}' = \sum_k \hat{a}_h(\mathbf{f}_q, \mathbf{f}_k)\, v_h(\mathbf{f}_k)$$

$$v(\mathbf{f}_k) = \mathbf{V}_h \cdot \mathbf{f}_k$$
$$k(\mathbf{f}_k) = \mathbf{K}_h \cdot \mathbf{f}_k$$



$$q(\mathbf{f}_q) = \mathbf{Q}_h \cdot \mathbf{f}_q$$

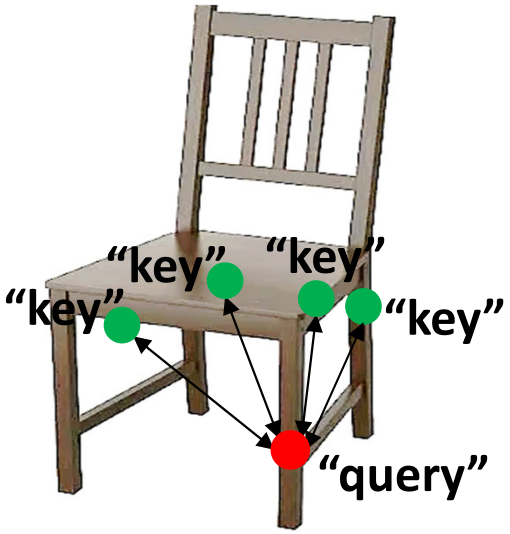
... concatenate the resulting features from all heads, and process them with the MLP



Vaswani, Attention Is All You Need, 2017

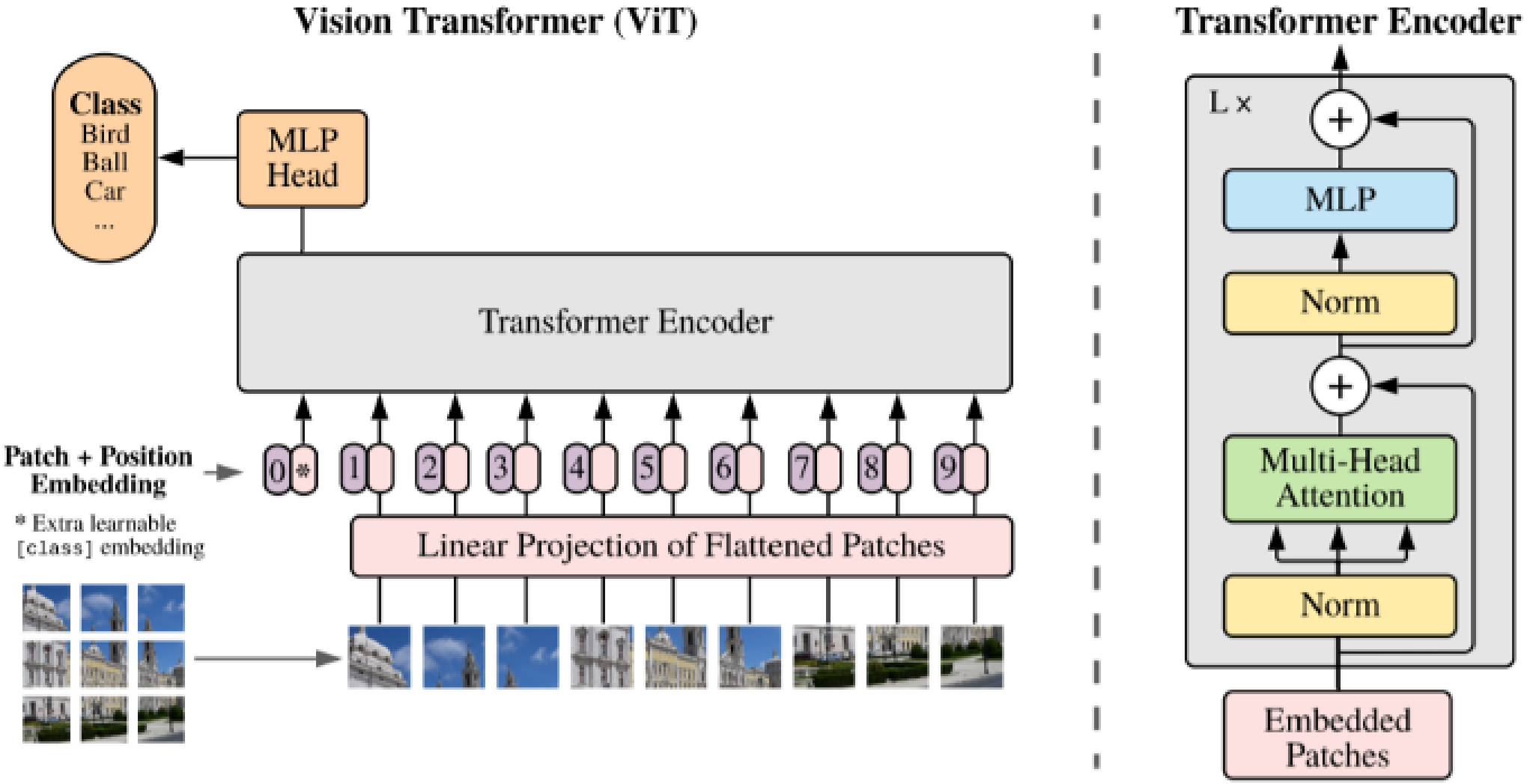
Quadratic complexity!

Comparing each query (N pixels) with every key (N pixels) yields **quadratic complexity**!



Vaswani, Attention Is All You Need, 2017

Compute attention on patches!



Need to also incorporate information about the position of patches:
more recent approaches learn mappings from coordinates (x,y) to positional embeddings
Patch n' Pack: NaViT, a Vision Transformer for any Aspect Ratio and Resolution, Deghani et al. 2023

ViT for segmentation

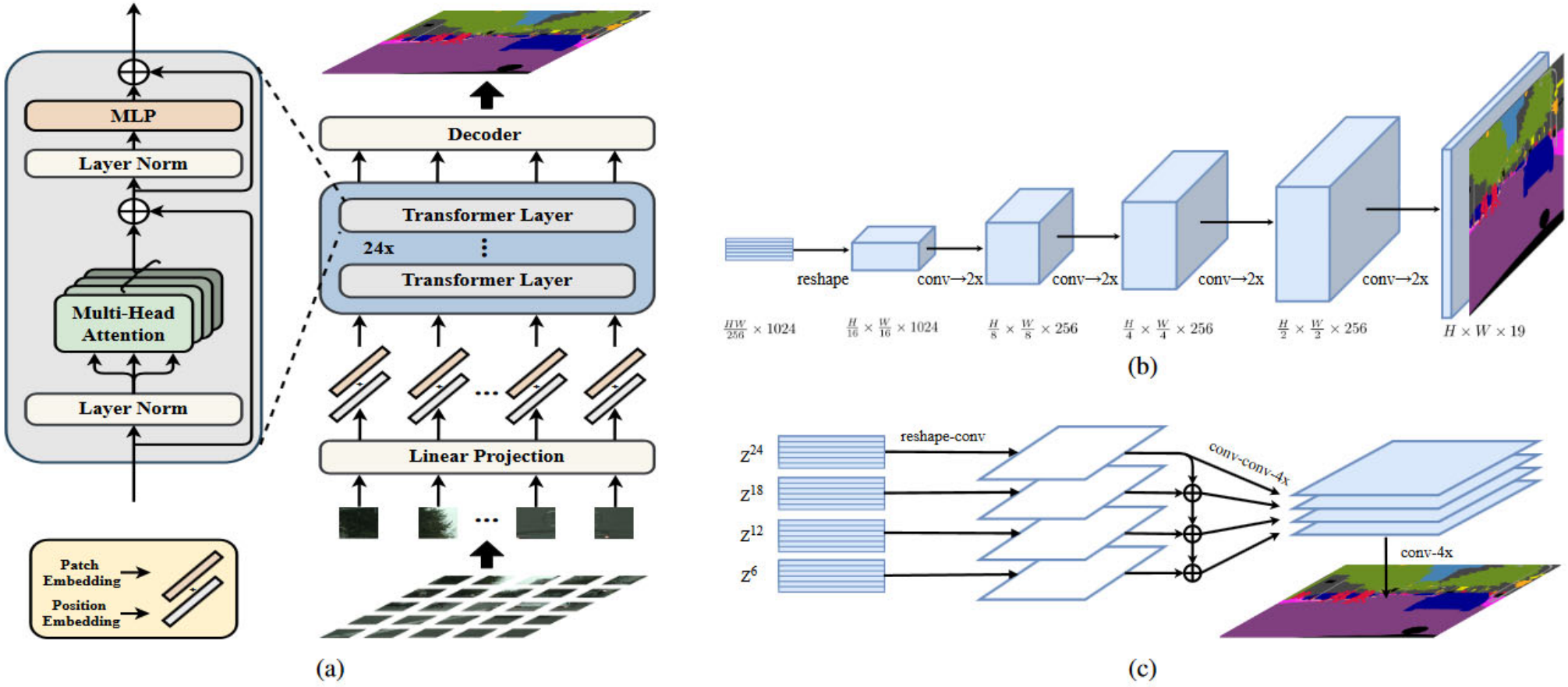


Figure 1. **Schematic illustration of the proposed *SEgmentation TRansformer* (SETR)** (a). We first split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. To perform pixel-wise segmentation, we introduce different decoder designs: (b) progressive upsampling (resulting in a variant called SETR-*PUP*); and (c) multi-level feature aggregation (a variant called SETR-*MLA*).

Rethinking Semantic Segmentation from a Sequence-to-Sequence Perspective with Transformers, https://fudan-zvg.github.io/SETR/