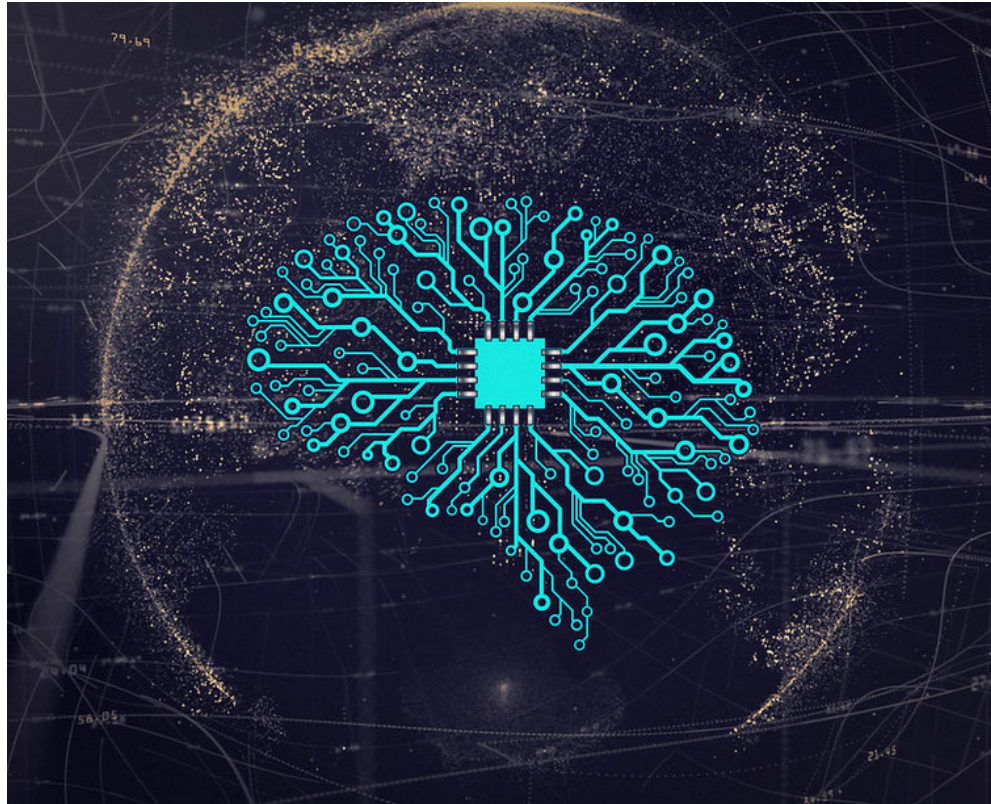


Part I: Intro to Learning Basics



Intelligent Visual Computing
Evangelos Kalogerakis

Learning basics: Classification

Suppose you want to predict **mug** or **no mug** for a shape.

Output: $y = 1$ [*coffee mug*], $y = 0$ [*no coffee mug*]

Input: $\mathbf{x} = \{x_1, x_2, \dots\}$ [*curvature, shape histograms etc*]



Mug?



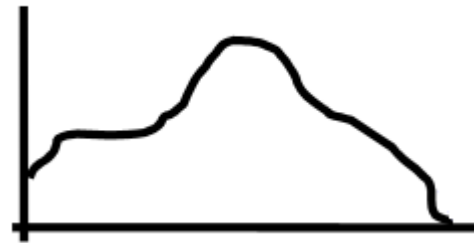
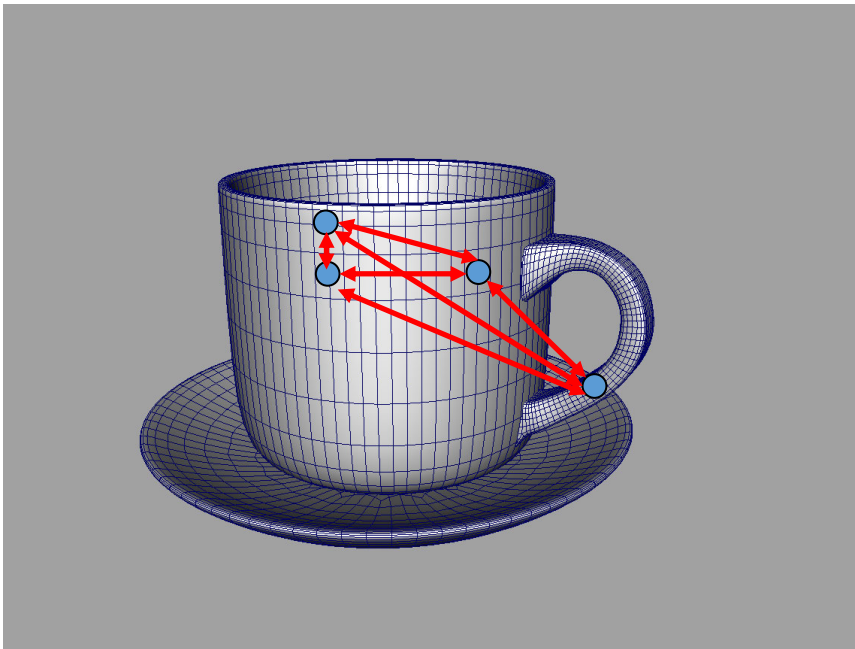
Mug?



Mug?

“Hand-engineered” descriptors

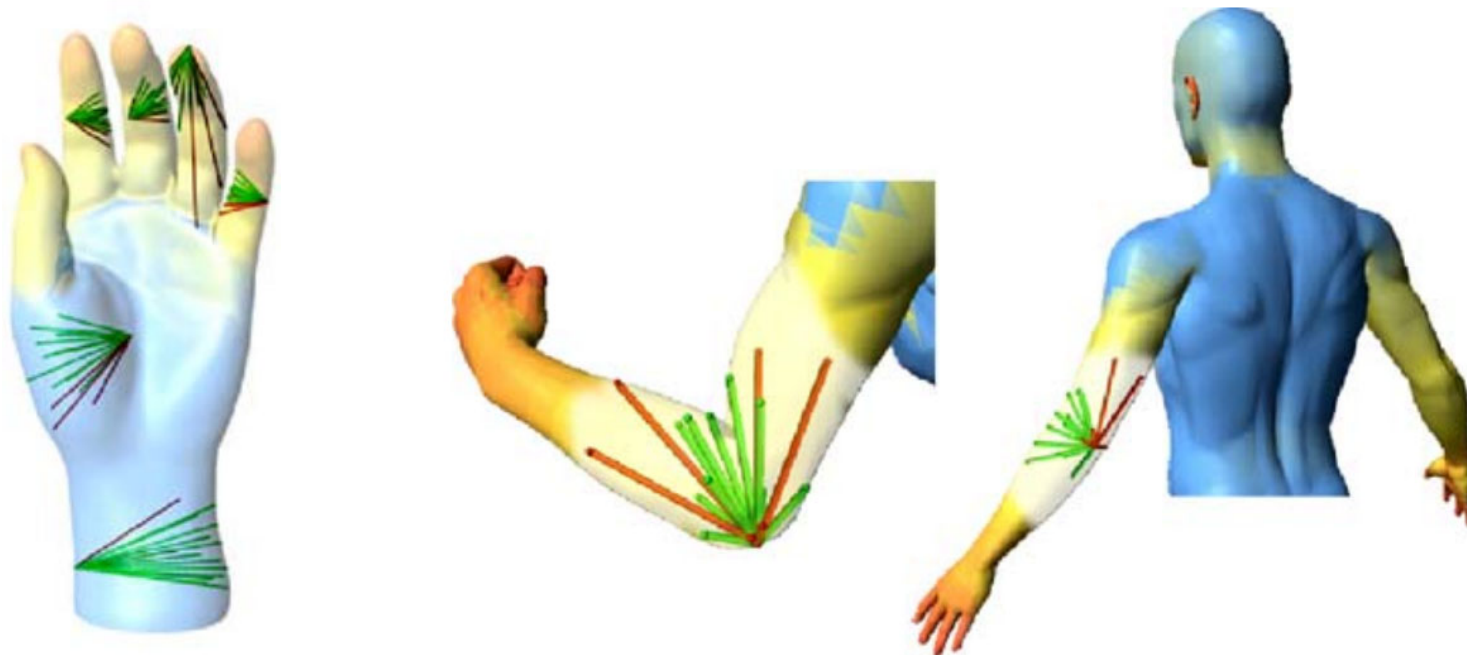
Shape histograms: measure distance between pairs of sample points, then create a histogram of these distances



Robert Osada, Thomas Funkhouser, Bernard Chazelle, and David Dobkin.
"Shape Distributions." ACM Transactions on Graphics 21(4):807-832, 2002

“Hand-engineered” descriptors

Local shape diameter: for each sample point, throw rays from the point opposite to its normal, check where they intersect, measure ray lengths, and average them.



Contextual Part Analogies in 3D Objects / Lior Shapira, Shy Shalom, Ariel Shamir, Richard H. Zhang, Daniel Cohen-Or, International Journal of Computer Vision (IJCV), 2009

Learning basics: Classification

Suppose you want to predict **mug** or **no mug** for a shape.

Output: $y = 1$ [*coffee mug*], $y = 0$ [*no coffee mug*]

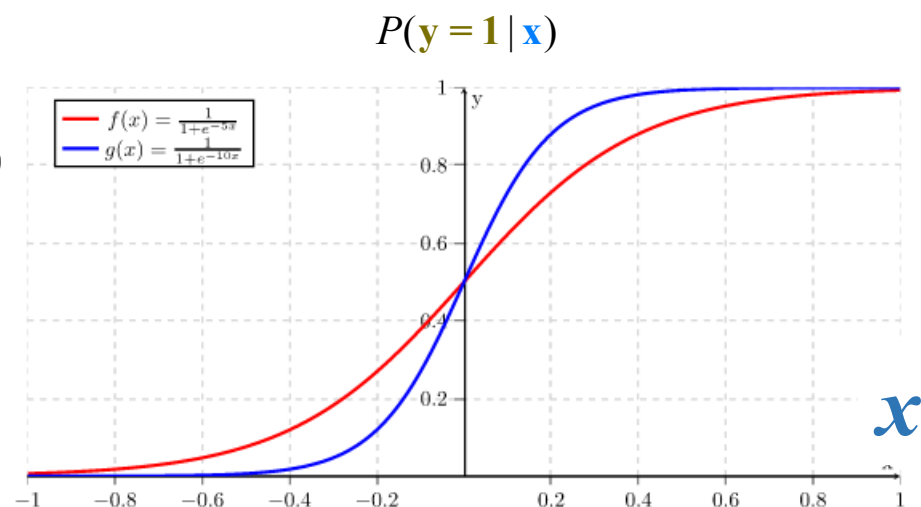
Input: $\mathbf{x} = \{x_1, x_2, \dots\}$ [*curvature, shape histograms etc*]

Classification function:

$$P(\mathbf{y} = \mathbf{1} \mid \mathbf{x}) = \mathbf{f}(\mathbf{x}) = \sigma(\mathbf{x}^T \cdot \mathbf{w})$$

where \mathbf{w} is a **weight vector**

$$\sigma(\mathbf{x}^T \cdot \mathbf{w}) = \frac{1}{1 + \exp(-\mathbf{x}^T \cdot \mathbf{w})}$$



Learning basics: Classification

Suppose you want to predict **mug** or **no mug** for a shape.

Output: $y = 1$ [*coffee mug*], $y = 0$ [*no coffee mug*]

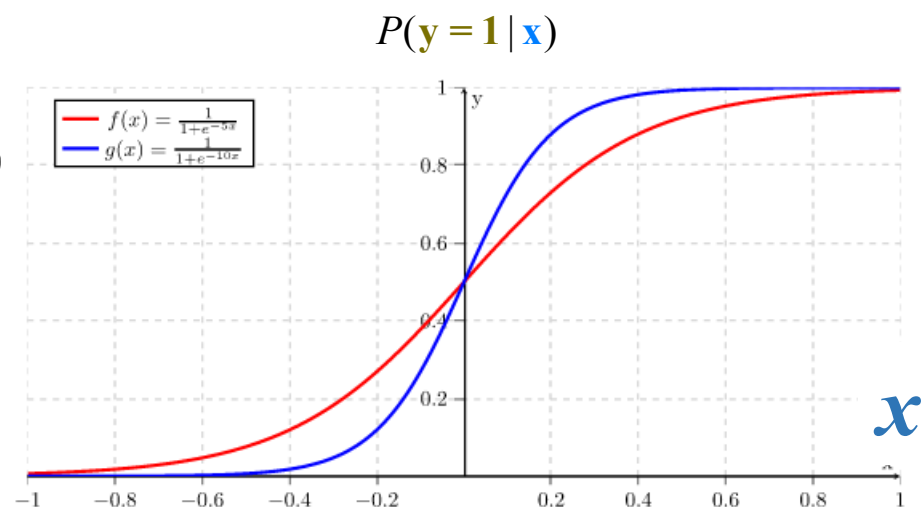
Input: $\mathbf{x} = \{x_1, x_2, \dots\}$ [*curvature, shape histograms etc*]

Classification function:

$$P(\mathbf{y} = \mathbf{1} \mid \mathbf{x}) = \mathbf{f}(\mathbf{x}) = \sigma(\mathbf{x}^T \cdot \mathbf{w})$$

where \mathbf{w} is a **weight vector**

$$\sigma(\mathbf{x}^T \cdot \mathbf{w}) = \frac{1}{1 + \exp(-\mathbf{x}^T \cdot \mathbf{w})}$$



Learning basics: Classification

Suppose you want to predict **mug** or **no mug** for a shape.

Output: $y = 1$ [*coffee mug*], $y = 0$ [*no coffee mug*]

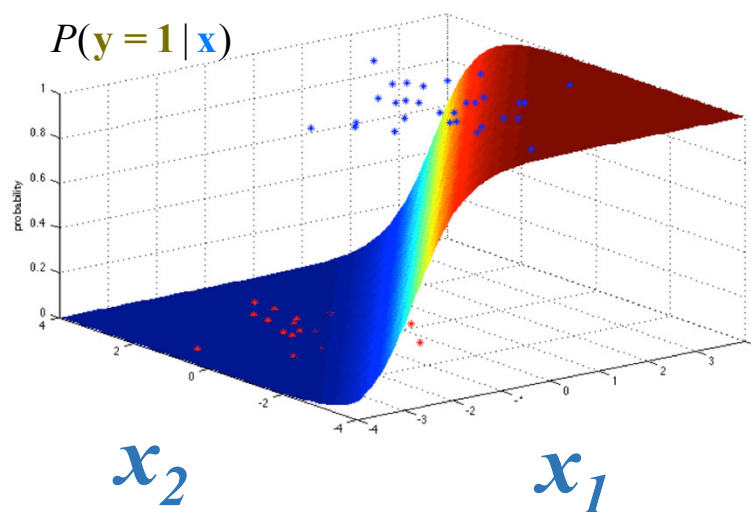
Input: $\mathbf{x} = \{x_1, x_2, \dots\}$ [*curvature, shape histograms etc*]

Classification function:

$$P(\mathbf{y} = \mathbf{1} \mid \mathbf{x}) = \mathbf{f}(\mathbf{x}) = \sigma(\mathbf{x}^T \cdot \mathbf{w})$$

where \mathbf{w} is a **weight vector**

$$\sigma(\mathbf{x}^T \cdot \mathbf{w}) = \frac{1}{1 + \exp(-\mathbf{x}^T \cdot \mathbf{w})}$$



Training

Need to estimate parameters \mathbf{w} from training data e.g., shapes of objects \mathbf{x}_i and given labels \mathbf{y}_i (mugs/no mugs) ($i=1 \dots K$ training shapes)

Find parameters that **maximize probability of training data**

$$\max_{\mathbf{w}} \prod_{i=1}^K P(\mathbf{y} = 1 \mid \mathbf{x}_i)^{[y_i^{(gt)} == 1]} [1 - P(\mathbf{y} = 1 \mid \mathbf{x}_i)]^{[y_i^{(gt)} == 0]}$$

Training

Need to estimate parameters \mathbf{w} from training data e.g.,
shapes of objects \mathbf{x}_i and given labels y_i (mugs/no mugs)
($i=1 \dots K$ training shapes)

Find parameters that **maximize probability of training data**

$$\max_{\mathbf{w}} \prod_{i=1}^K \sigma(\mathbf{x}_i^T \cdot \mathbf{w})^{[y_i^{(gt)}==1]} [1 - \sigma(\mathbf{x}_i^T \cdot \mathbf{w})]^{[y_i^{(gt)}==0]}$$

Training

Need to estimate parameters \mathbf{w} from training data e.g.,
shapes of objects \mathbf{x}_i and given labels y_i (mugs/no mugs)
($i=1 \dots K$ training shapes)

Find parameters that **maximize the log prob. of training data**

$$\max_{\mathbf{w}} \log \left\{ \prod_{i=1}^K \sigma(\mathbf{x}_i^T \cdot \mathbf{w})^{[y_i^{(gt)}==1]} [1 - \sigma(\mathbf{x}_i^T \cdot \mathbf{w})]^{[y_i^{(gt)}==0]} \right\}$$

Training

Need to estimate parameters \mathbf{w} from training data e.g.,
shapes of objects \mathbf{x}_i and given labels y_i (mugs/no mugs)
($i=1 \dots K$ training shapes)

Find parameters that **maximize the log prob. of training data**

$$\max_{\mathbf{w}} \sum_{i=1}^K [y_i^{(gt)} == 1] \log \sigma(\mathbf{x}_i^T \cdot \mathbf{w}) + [y_i^{(gt)} == 0] \log(1 - \sigma(\mathbf{x}_i^T \cdot \mathbf{w}))$$

Training

Need to estimate parameters \mathbf{w} from training data e.g., shapes of objects \mathbf{x}_i and given labels \mathbf{y}_i (mugs/no mugs) ($i=1 \dots K$ training shapes)

This is called **log-likelihood**

$$\max_{\mathbf{w}} \sum_{i=1}^K [y_i^{(gt)} == 1] \log \sigma(\mathbf{x}_i^T \cdot \mathbf{w}) + [y_i^{(gt)} == 0] \log(1 - \sigma(\mathbf{x}_i^T \cdot \mathbf{w}))$$

$L(\mathbf{w})$

Training

Need to estimate parameters \mathbf{w} from training data e.g., shapes of objects \mathbf{x}_i and given labels \mathbf{y}_i (mugs/no mugs) ($i=1 \dots K$ training shapes)

We have an **optimization problem**.

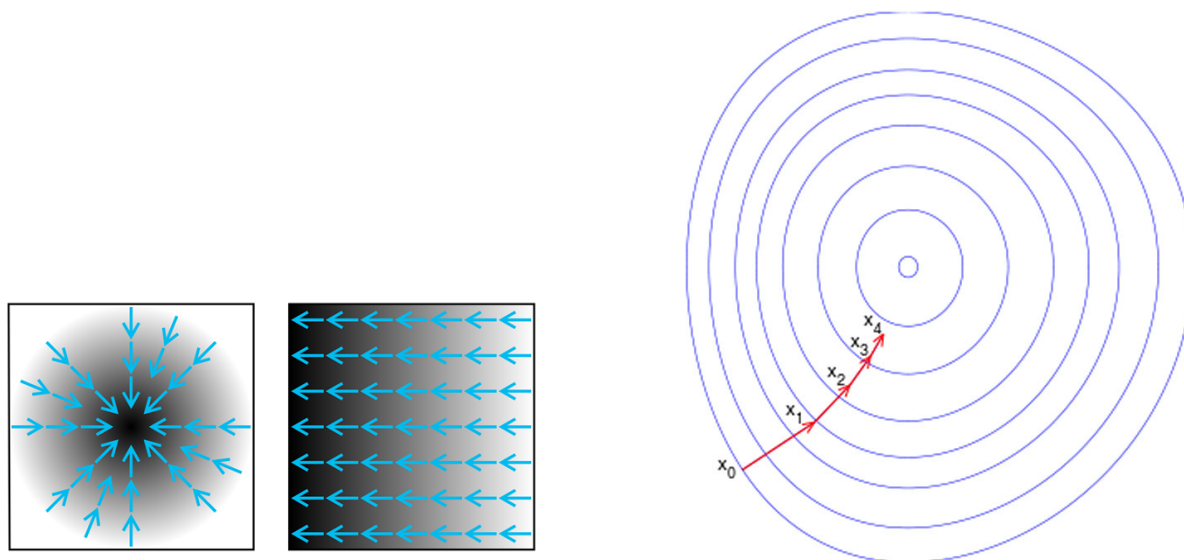
$$\max_{\mathbf{w}} \sum_{i=1}^K [y_i^{(gt)} == 1] \log \sigma(\mathbf{x}_i^T \cdot \mathbf{w}) + [y_i^{(gt)} == 0] \log(1 - \sigma(\mathbf{x}_i^T \cdot \mathbf{w}))$$

$L(\mathbf{w})$

$$\frac{\partial L(\mathbf{w})}{\partial w_d} = \sum_i x_{i,d} [y_i^{(gt)} - \sigma(\mathbf{x}_i^T \cdot \mathbf{w})]$$

(partial derivative for d^{th} parameter)

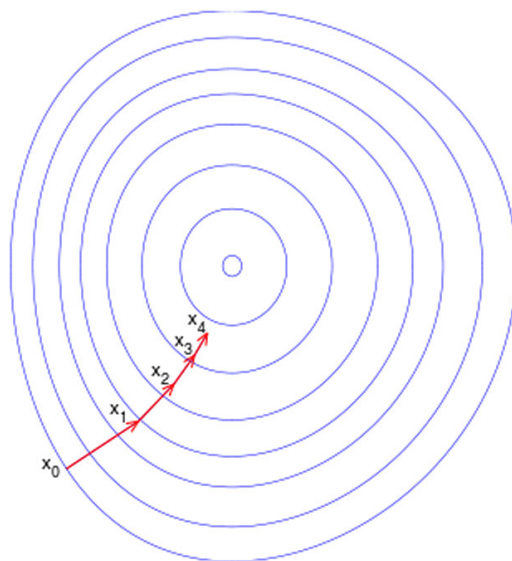
How can we maximize a function?



Follow the gradient! Given a random initialization of parameters and a step rate η , update them according to:

$$\mathbf{w}_{new} = \mathbf{w}_{old} + \eta \nabla L(\mathbf{w})$$

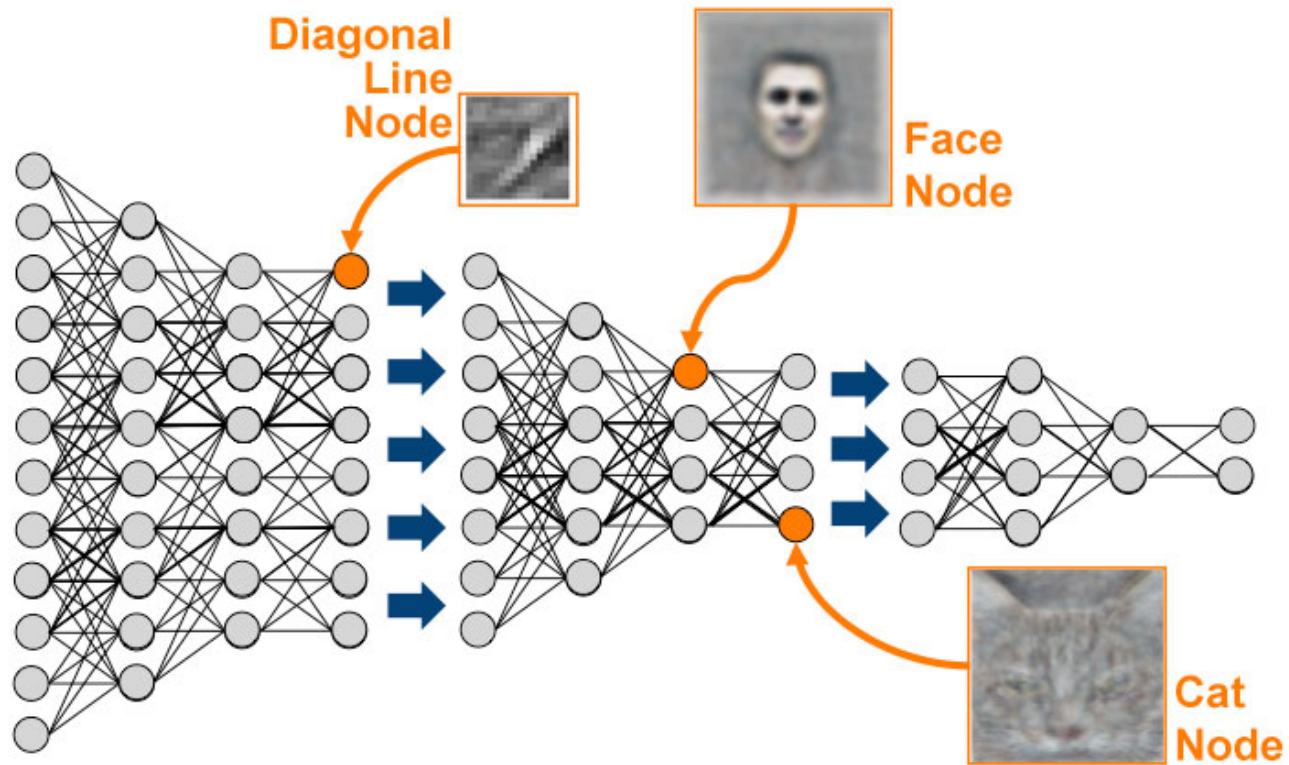
How can we minimize a function?



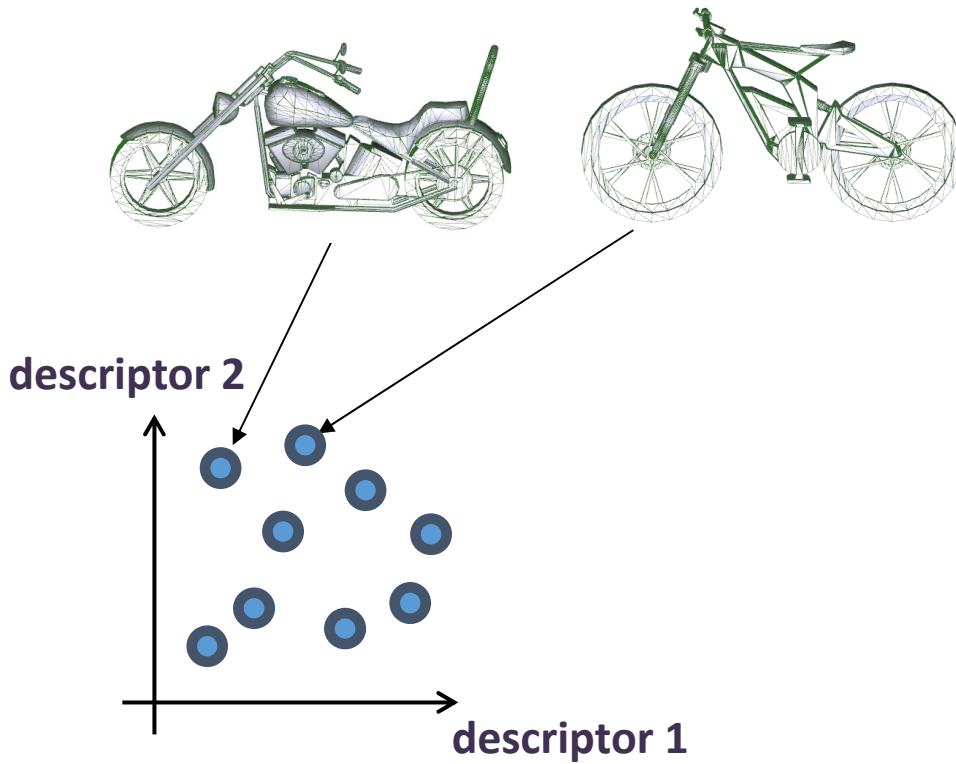
Gradient descent: Given a random initialization of parameters and a step rate η , update them according to:

$$\mathbf{w}_{new} = \mathbf{w}_{old} - \eta \nabla L(\mathbf{w})$$

Part II: Neural Network Intro



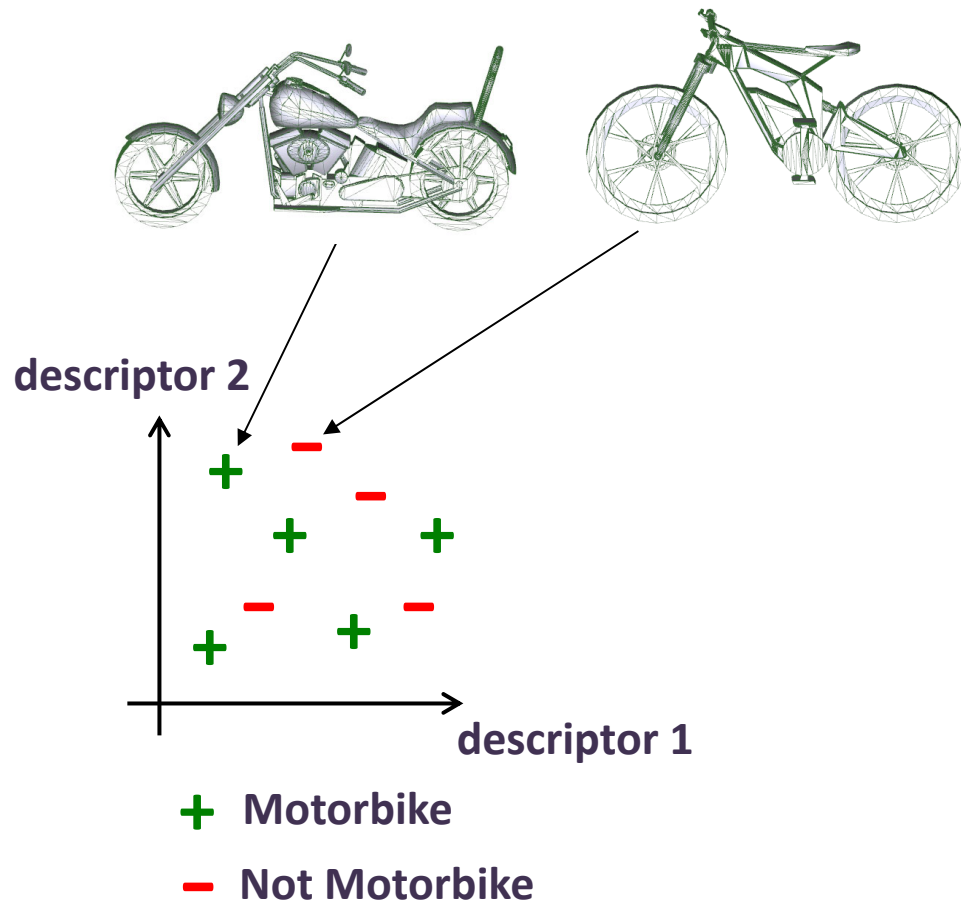
The importance of good descriptors



“Traditional approach”:

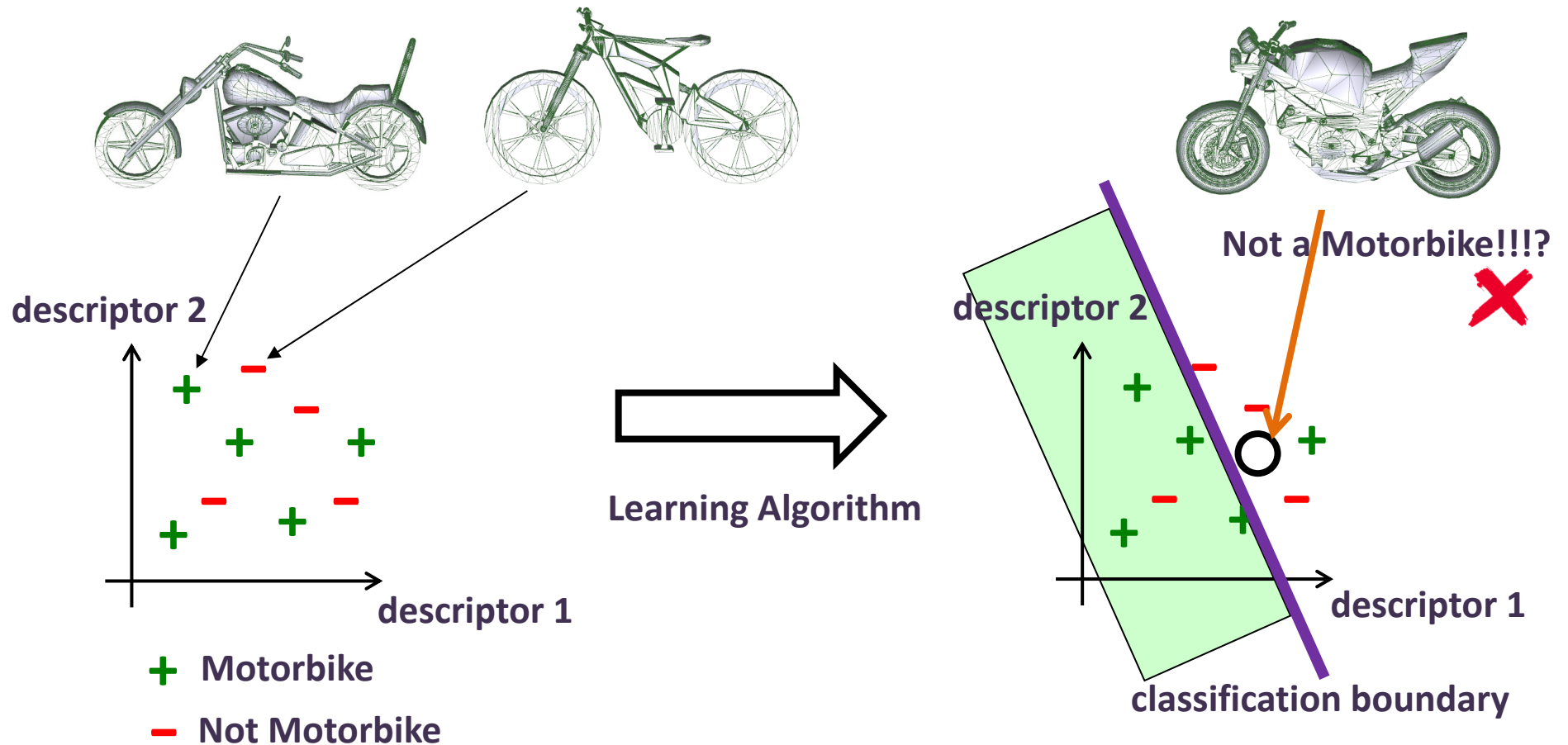
Engineer descriptors: shape histograms, local shape diameter

The importance of good shape descriptors



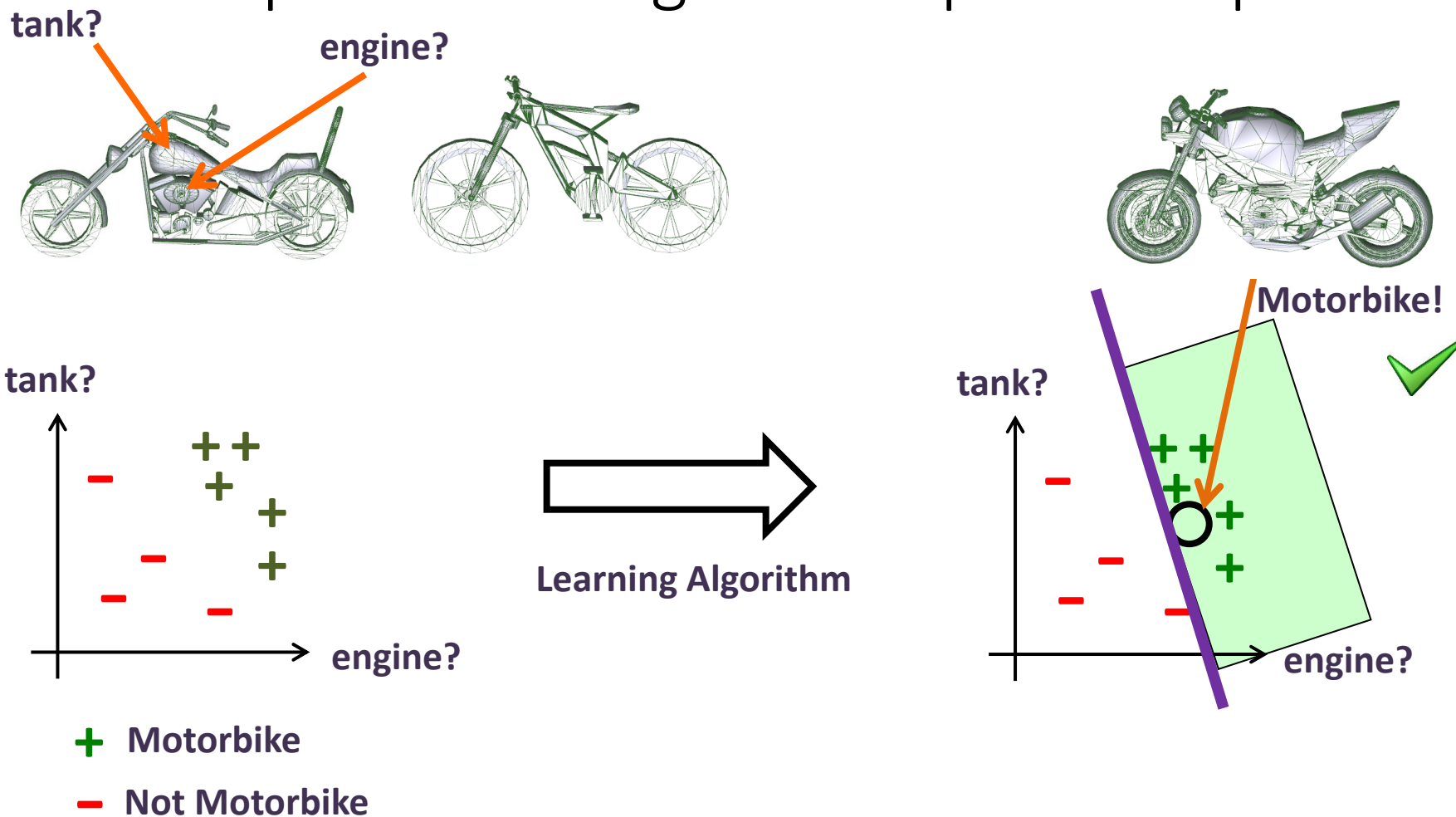
Gather training labels

The importance of good shape descriptors



Train a classifier.... easily fails to generalize

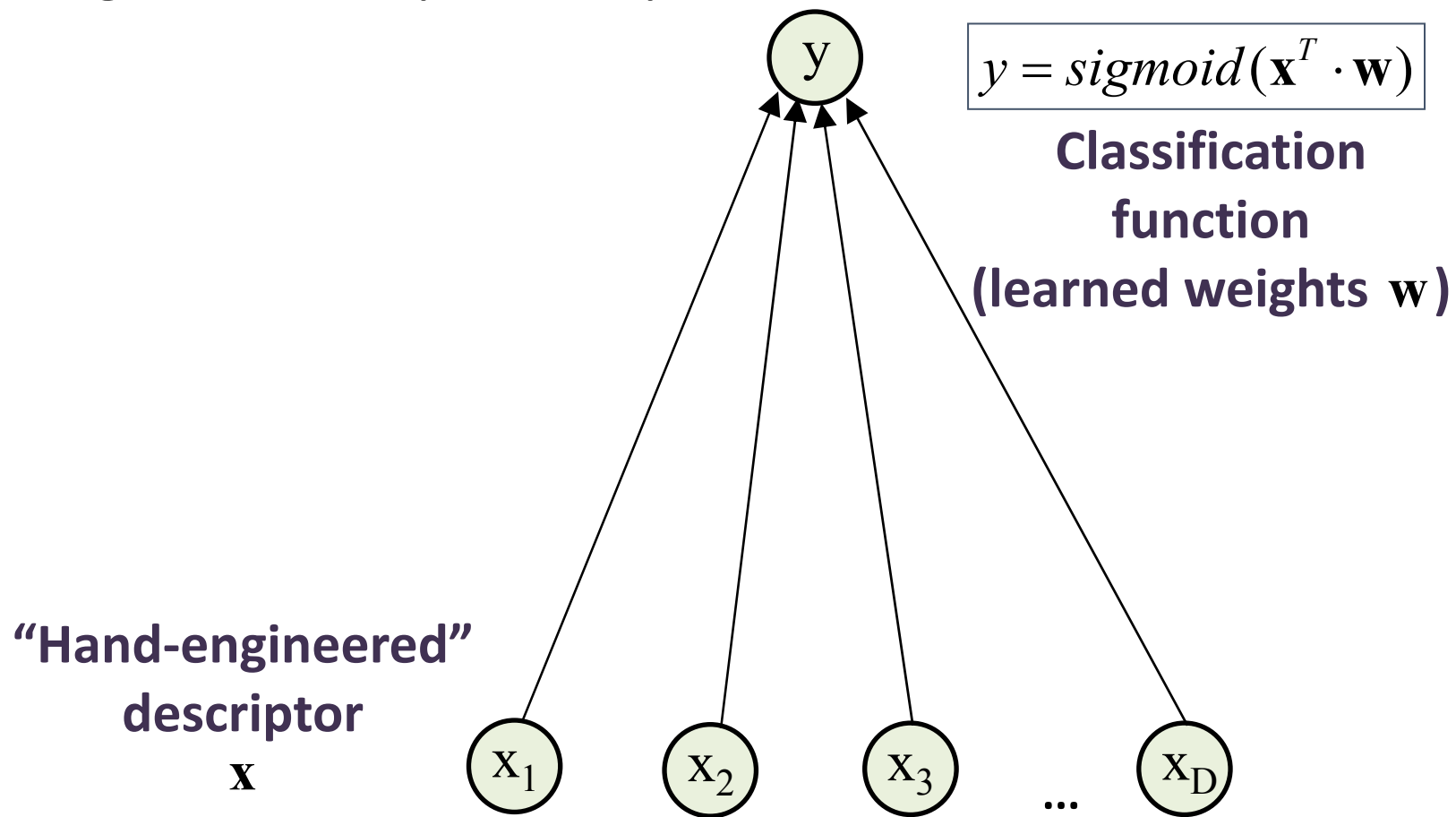
The importance of good shape descriptors



Need descriptors that capture semantics, function...

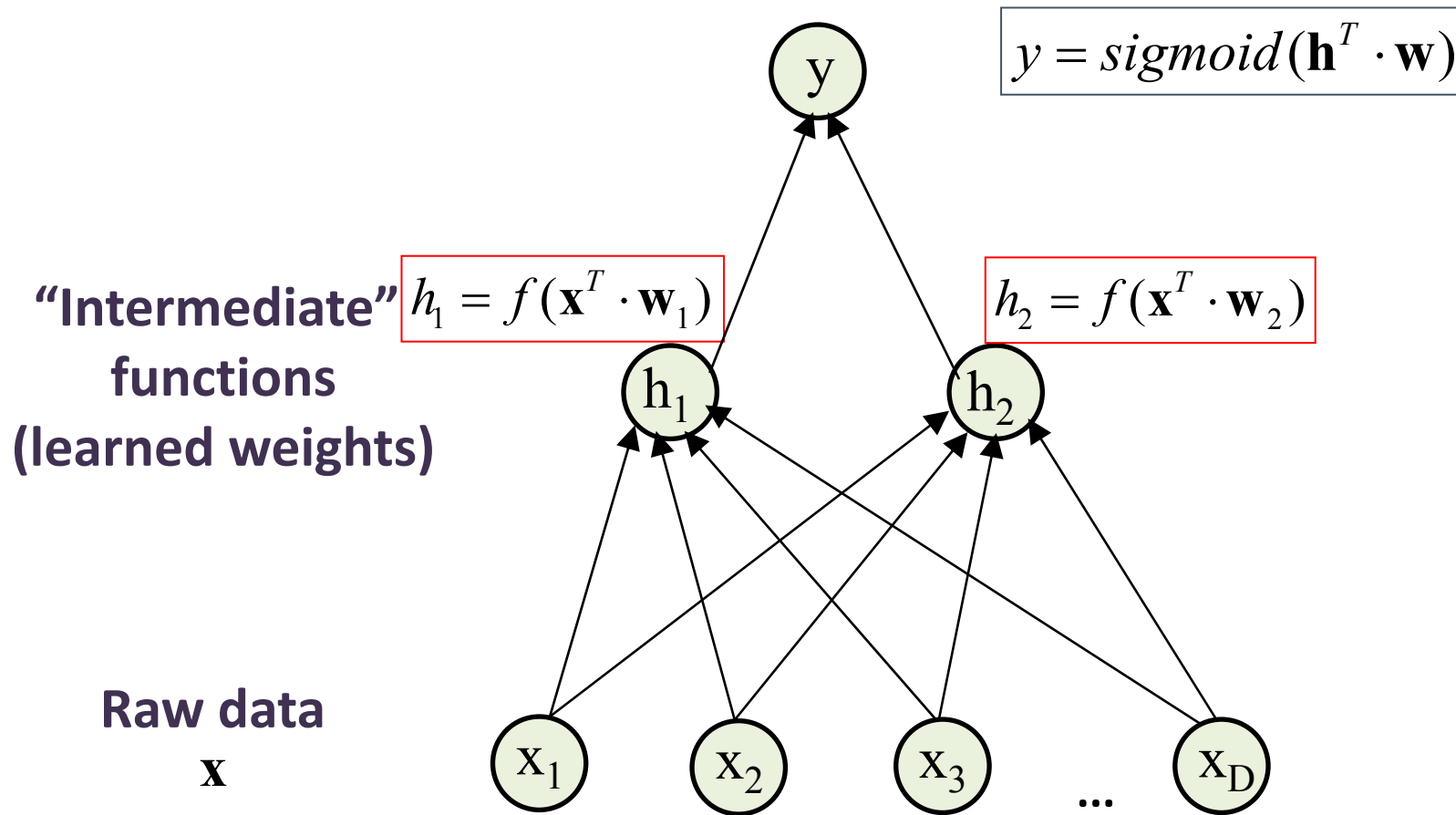
From “shallow” mappings...

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



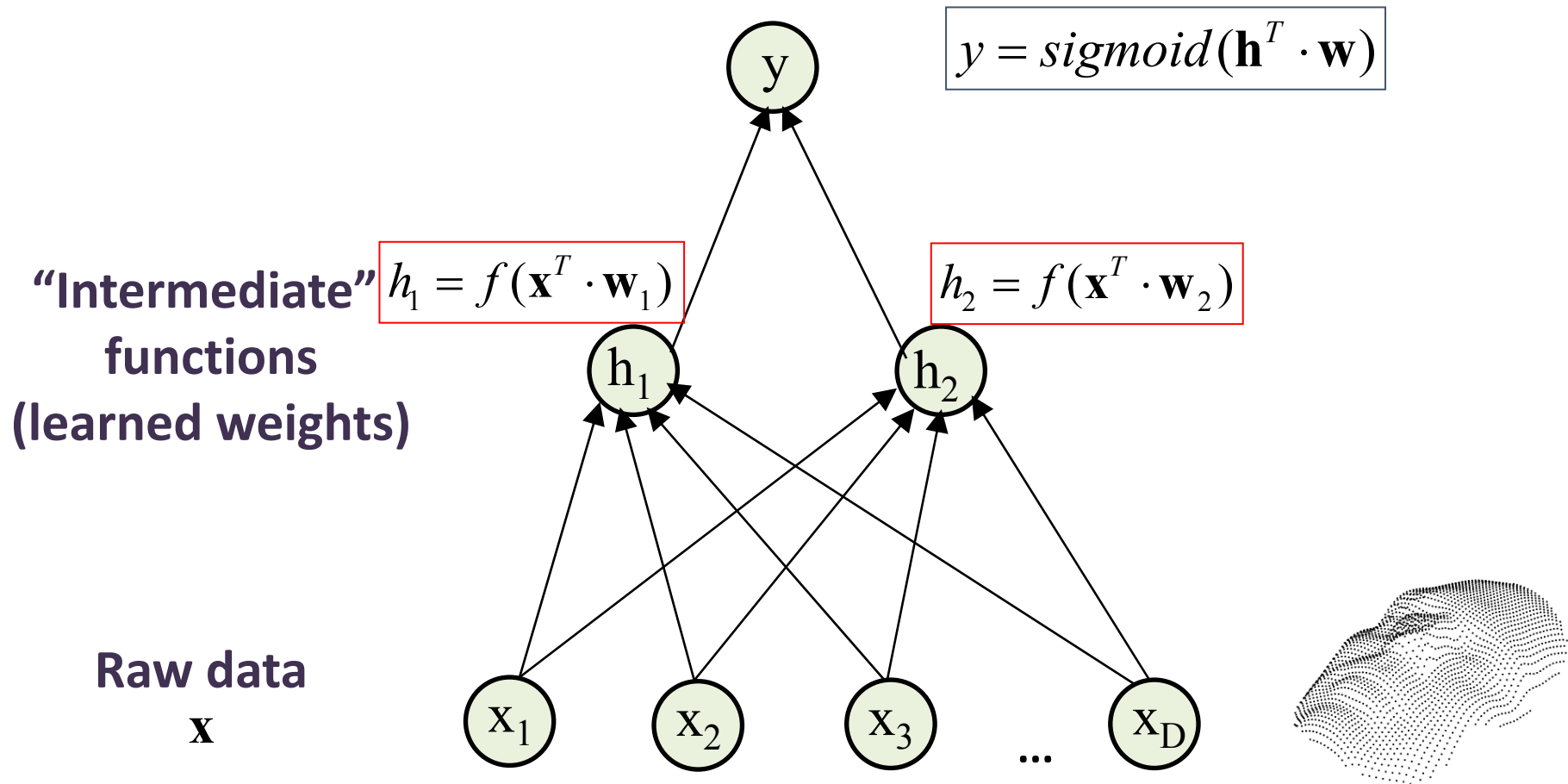
... to neural nets

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



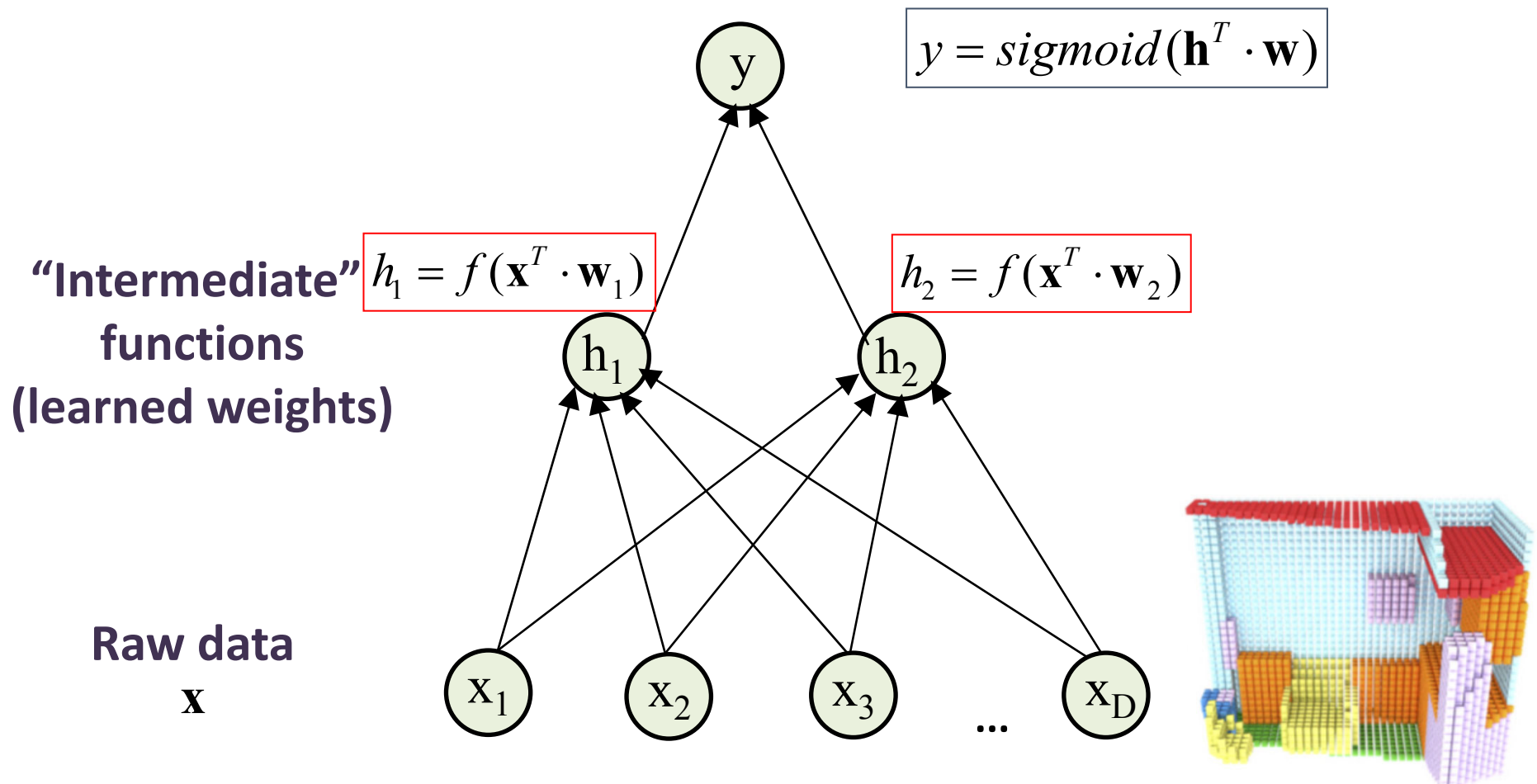
... to neural nets

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



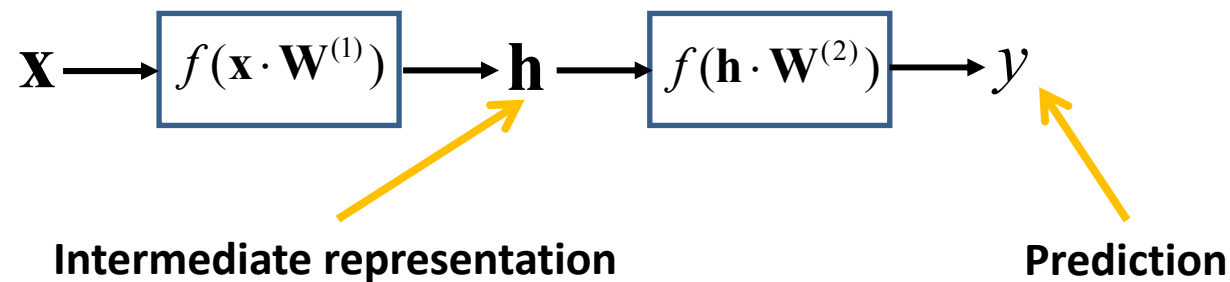
... to neural nets

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



Neural network

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

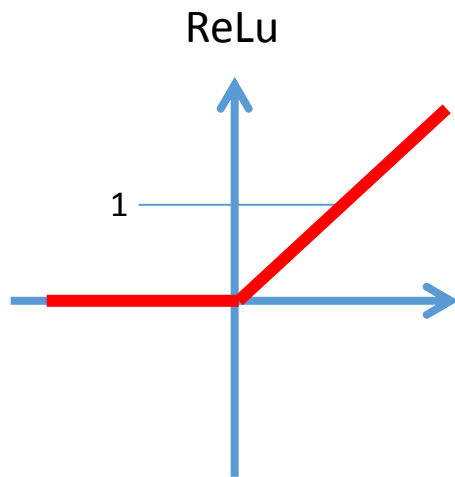


$$\text{where } \mathbf{W}^{(\cdot)} = \begin{bmatrix} \mathbf{w}_1^{(\cdot)} & \mathbf{w}_2^{(\cdot)} & \dots & \mathbf{w}_M^{(\cdot)} \end{bmatrix}$$

*modified slides originally
by Adam Coates*

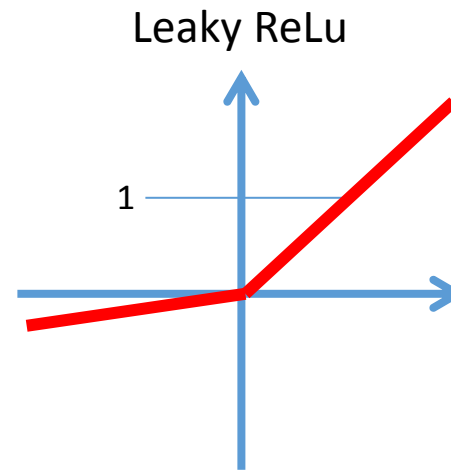
Activation functions

("f" in the previous slides)



$$\text{ReLU}'(x) = [x > 0]$$

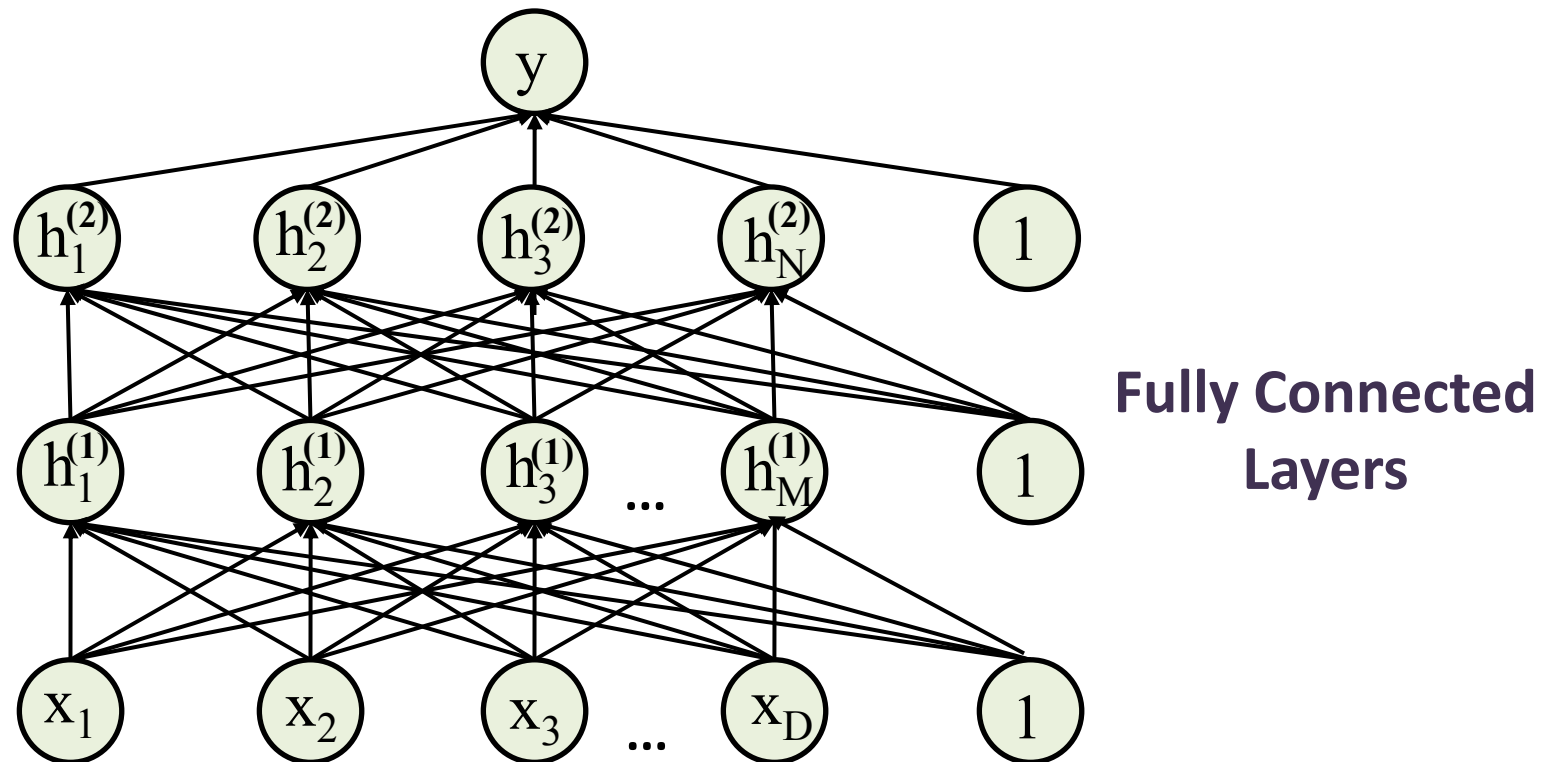
(dies at too negative input)



$$\text{LReLU}'(x) = a[x < 0] + [x > 0]$$

Deep Neural Network (fully connected)

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



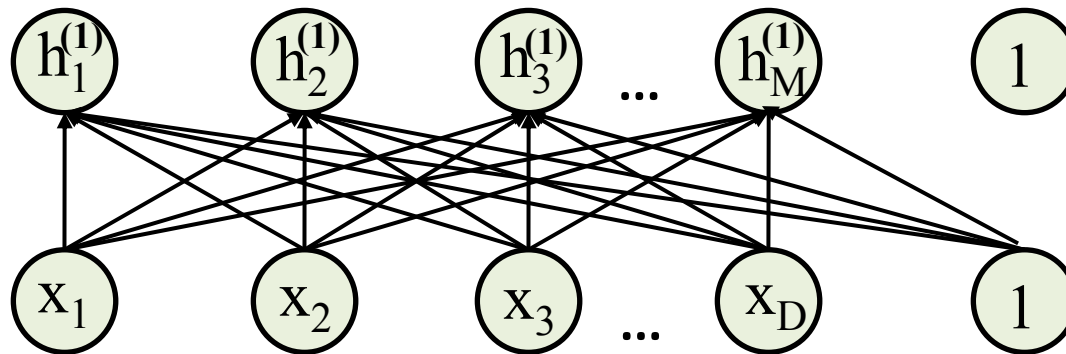
Forward propagation

Process to compute output:



Forward propagation

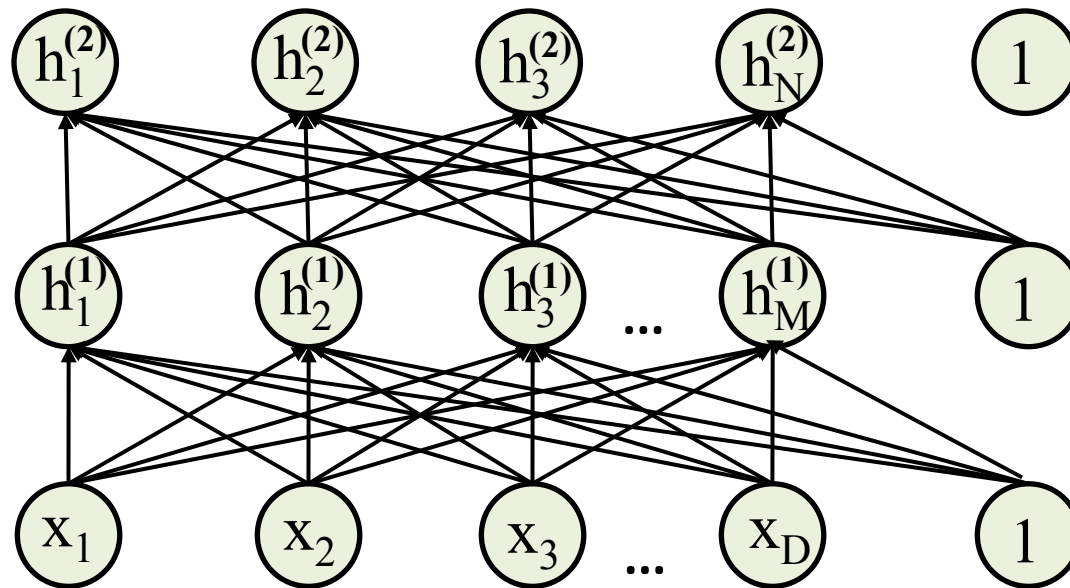
Process to compute output:



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

Forward propagation

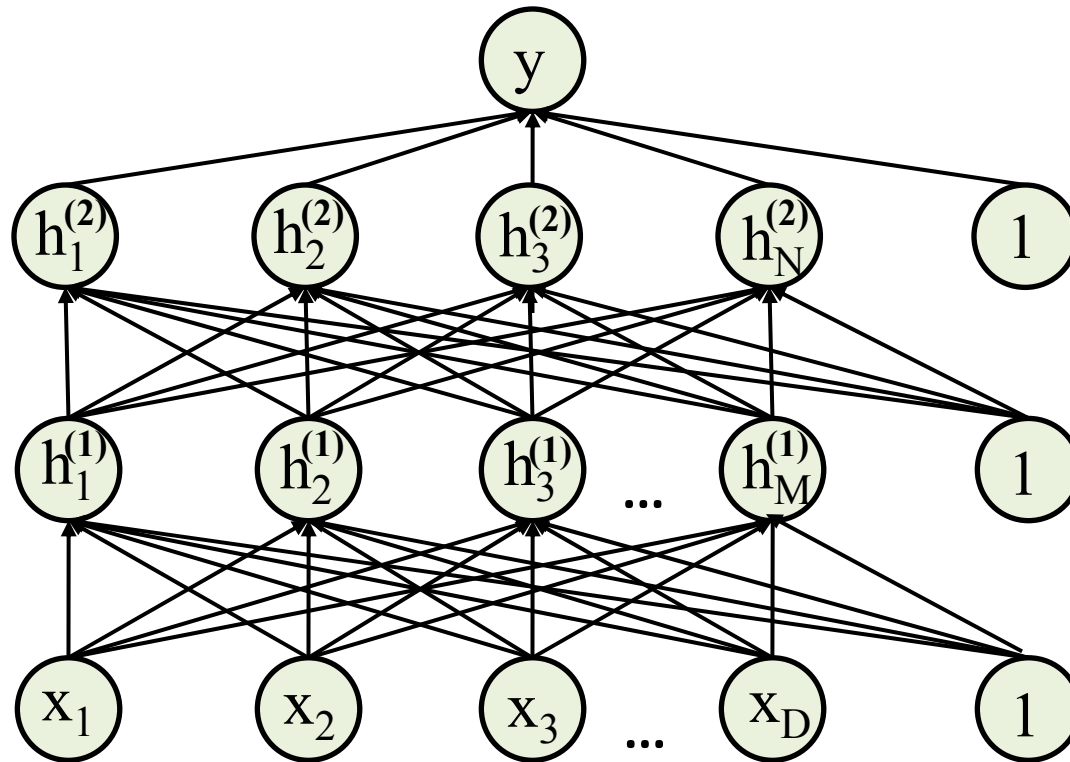
Process to compute output:



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

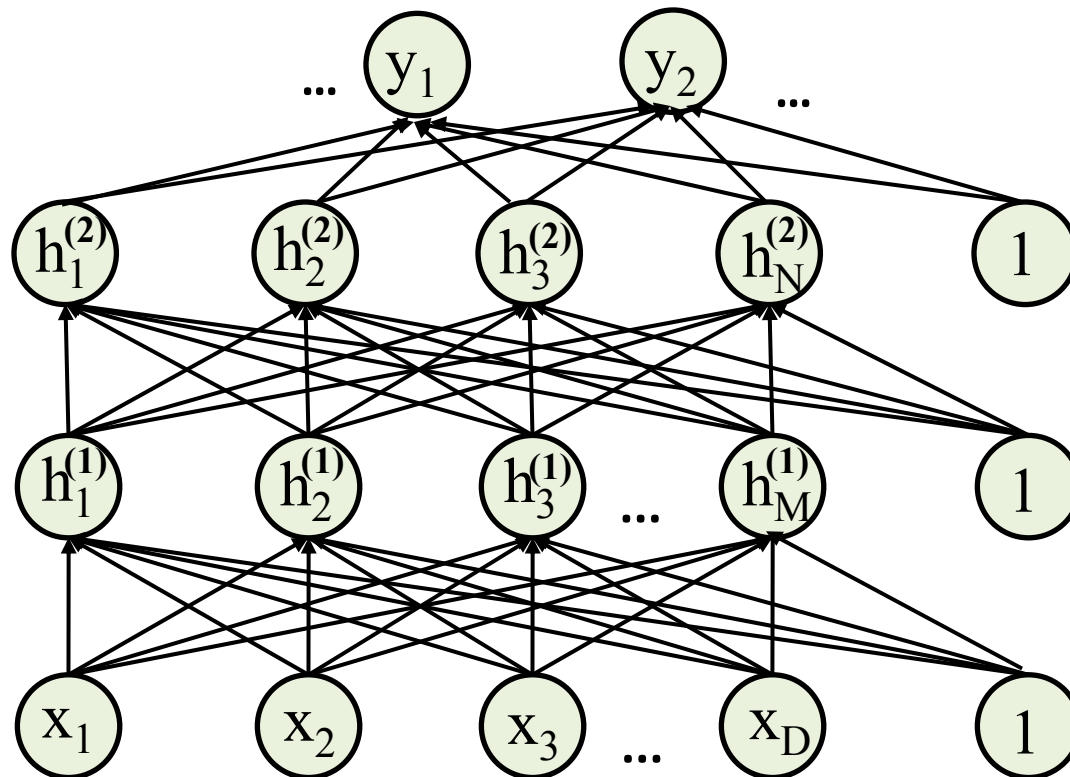
Forward propagation

Process to compute output:



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

Multiple outputs



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

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:

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

sum over each
training example i


A diagram consisting of four arrows pointing from a central point below the text to the components of the equation above. Two arrows point to the summation symbol \sum and the index i in the term $-\sum_i$. Two other arrows point to the $y_i^{(gt)}$ terms in the two summands of the equation.

How to learn the parameters?

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

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

sum over each
training example i

A diagram consisting of four arrows pointing from a central point below the text to the components of the equation above. Two arrows point to the summation symbol \sum and the index i in the term \sum_i . Two other arrows point to the terms $y_i^{(gt)}$ and x_i in the expression $[y_i^{(gt)} == 1] \log f(\mathbf{x}_i)$.

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. $\mathbf{W} = \{\mathbf{W}^{(1)}, \mathbf{W}^{(2)} \dots\}$ 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. $\mathbf{W} = \{\mathbf{W}^{(1)}, \mathbf{W}^{(2)} \dots\}$ of all layers!

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


How to learn the parameters?

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

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

Need gradient wrt param. $\mathbf{W} = \{\mathbf{W}^{(1)}, \mathbf{W}^{(2)} \dots\}$ 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:

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

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

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

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


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. $\mathbf{W} = \{\mathbf{W}^{(1)}, \mathbf{W}^{(2)} \dots\}$ of all layers!

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

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

 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. $\mathbf{W} = \{\mathbf{W}^{(1)}, \mathbf{W}^{(2)} \dots\}$ 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. $\mathbf{W} = \{\mathbf{W}^{(1)}, \mathbf{W}^{(2)} \dots\}$ 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:

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

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

Anybody remembering the **chain rule**?

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

... 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. $\mathbf{W} = \{\mathbf{W}^{(1)}, \mathbf{W}^{(2)} \dots\}$ 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. $\mathbf{W} = \{\mathbf{W}^{(1)}, \mathbf{W}^{(2)} \dots\}$ 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. $\mathbf{W} = \{\mathbf{W}^{(1)}, \mathbf{W}^{(2)} \dots\}$ of all layers!

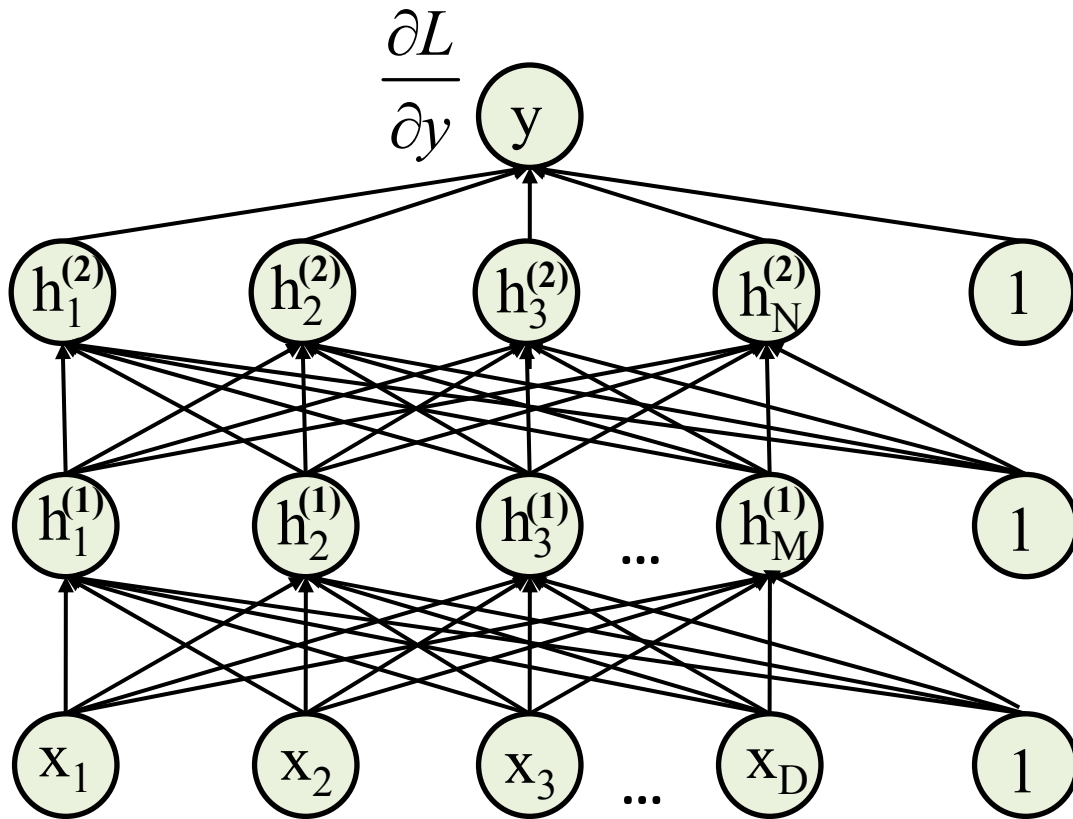
In our case, we need gradients wrt weights...

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

... and so on

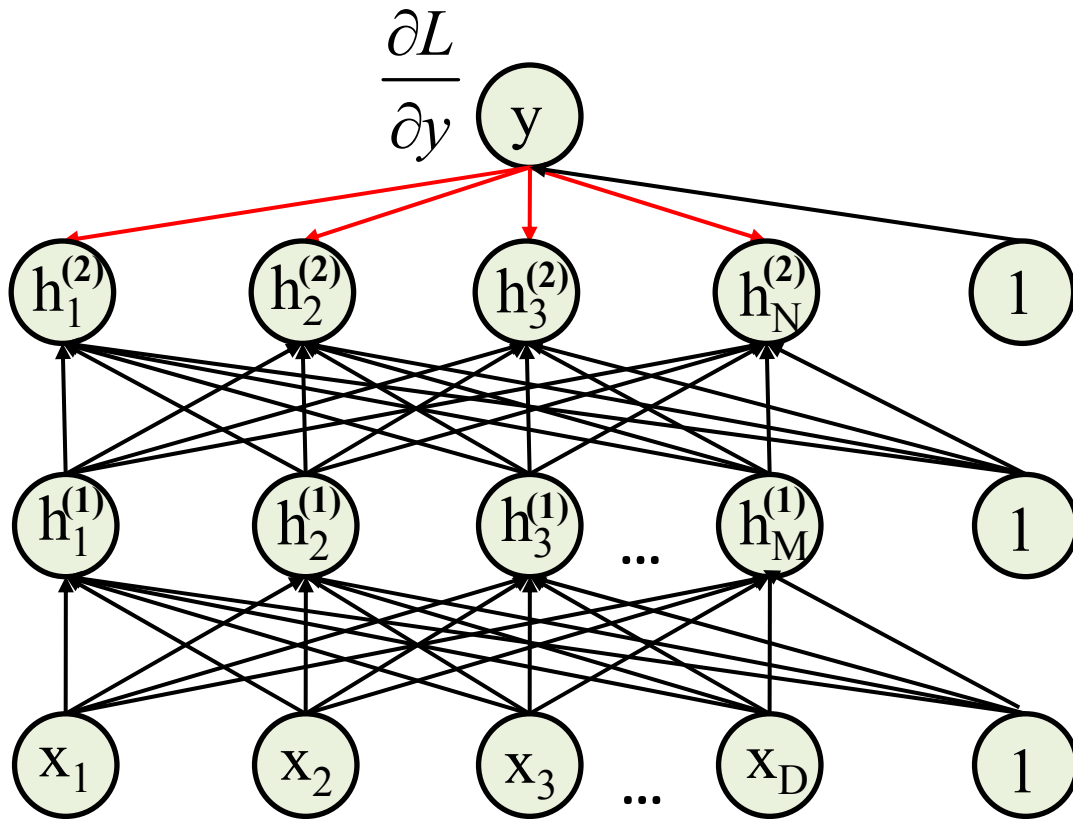
Backpropagation

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



Backpropagation

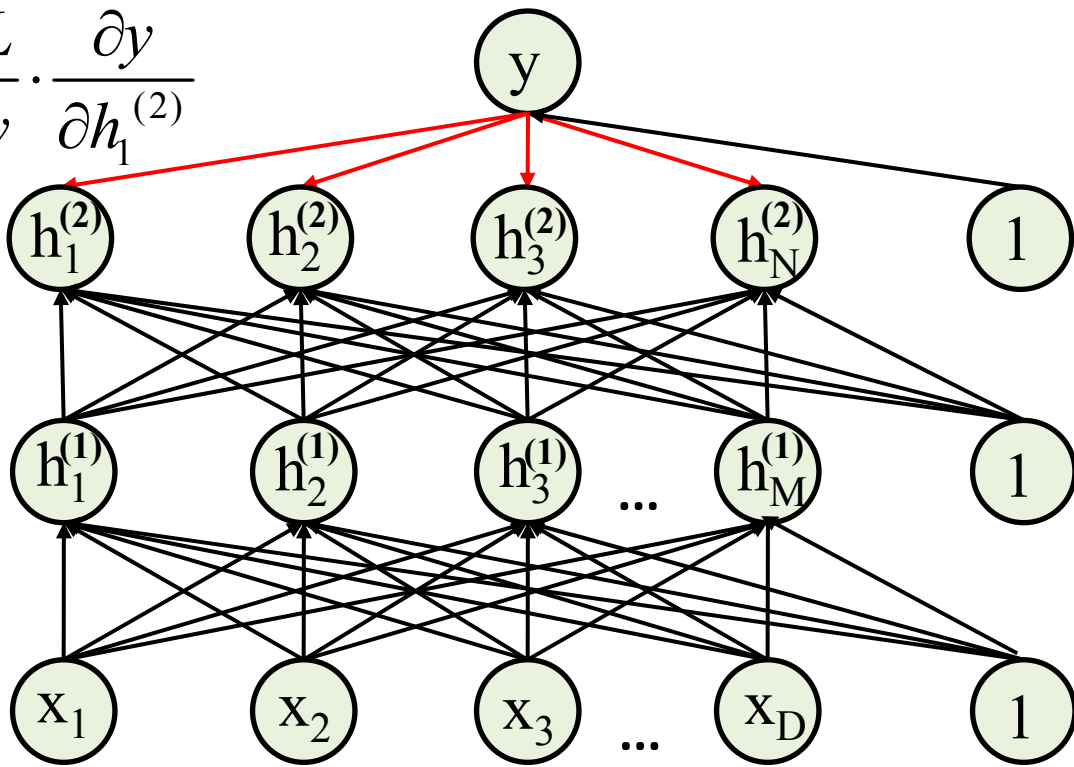
For each training example i (in the followings eq., I omit i for clarity):



Backpropagation

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

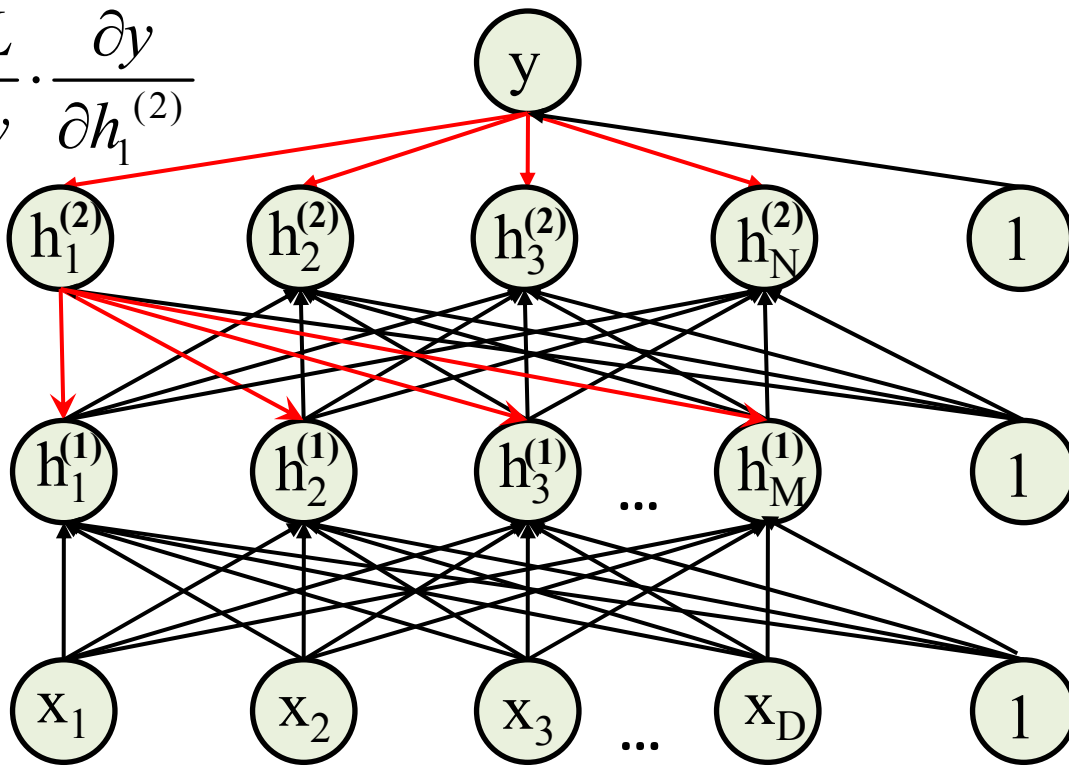
$$\frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial h_1^{(2)}}$$



Backpropagation

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

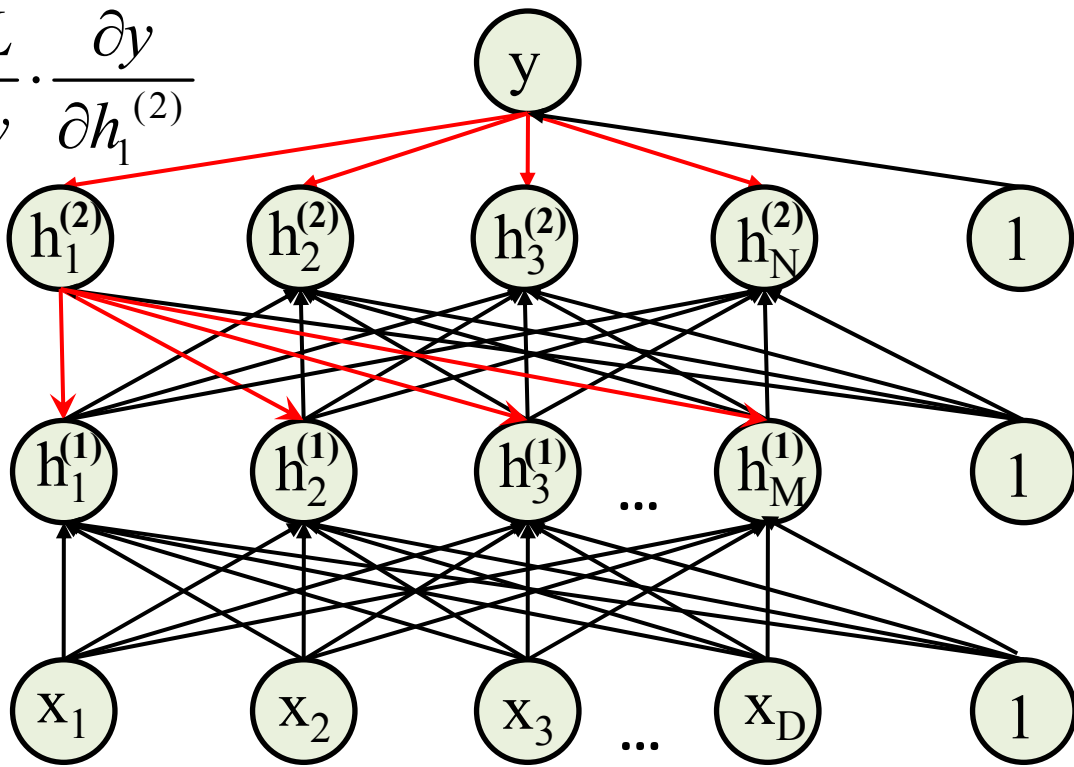
$$\frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial h_1^{(2)}}$$



Backpropagation

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

$$\frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial h_1^{(2)}}$$



Is this magic?

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

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

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

Is this magic?

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

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

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

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., sigmoid to its input $\mathbf{h}^{(l-1)}$:

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

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., sigmoid to its input $\mathbf{h}^{(l-1)}$:

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

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., sigmoid to its input $\mathbf{h}^{(l-1)}$:

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

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”

:

$$\frac{\partial L(\mathbf{w})}{\partial h_n^{(l)}} = \sum_t \frac{\partial L(\mathbf{w})}{\partial h_t^{(l+1)}} \frac{\partial h_t^{(l+1)}}{\partial h_n^{(l)}} = \sum_t \delta_t^{(l+1)} [h_t^{(l+1)} > 0] w_{t,n}^{(l+1)}$$

Training algorithm

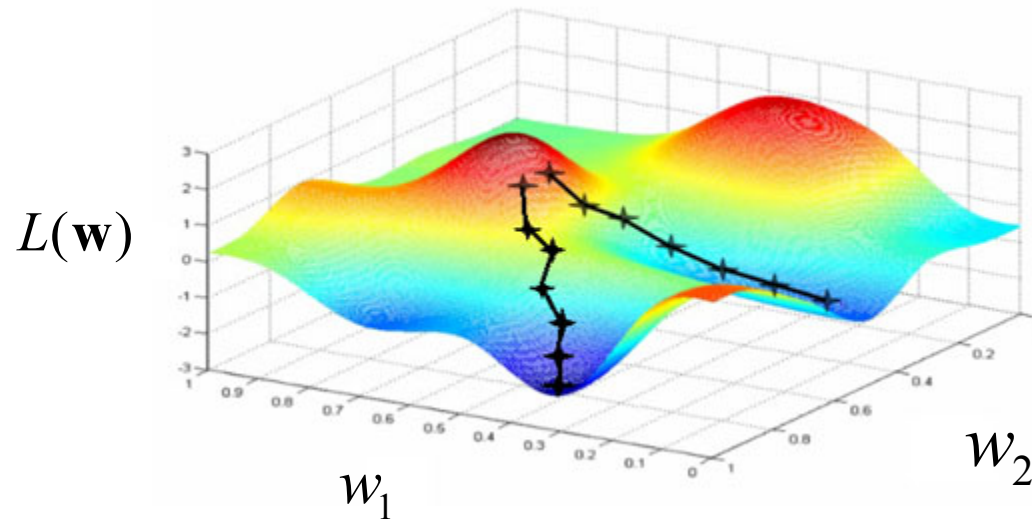
For each training example

1. **Forward propagation** to compute outputs per layer
2. **Back propagate** messages δ from top to bottom layer
3. Multiply messages δ with inputs to compute **derivatives per layer**
4. **Accumulate the derivatives** from that training example

Apply the gradient descent rule

Gradient Descent

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

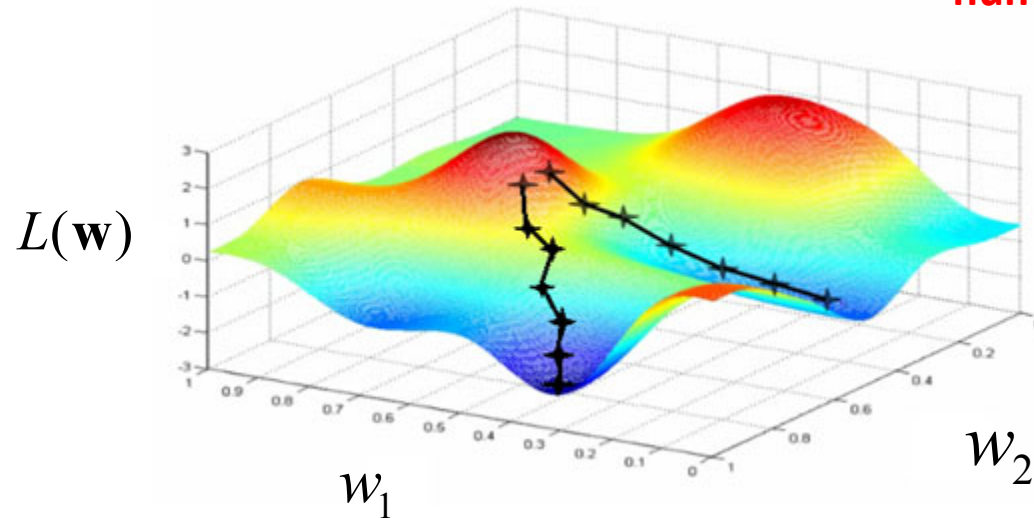


Lots and lots of local minima in neural networks!

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)



Lots and lots of local minima in neural networks!

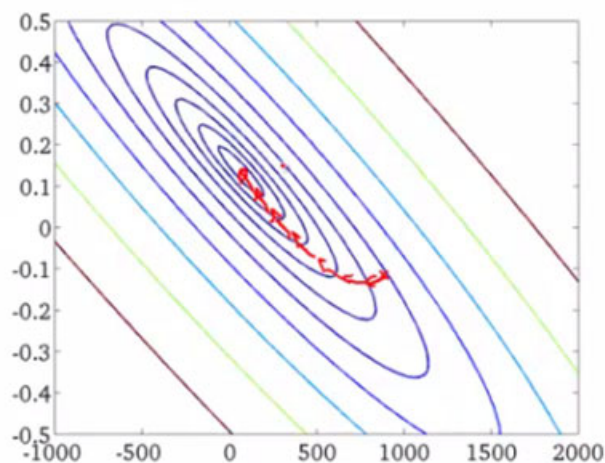
Stochastic Gradient Descent

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

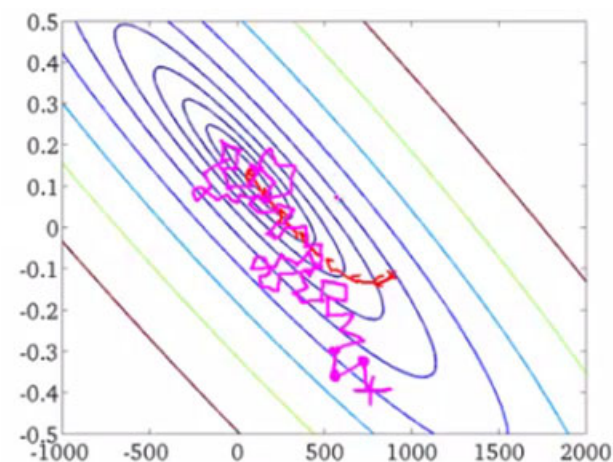
$$\mathbf{w}_{new} = \mathbf{w}_{old} - \eta \nabla_{\mathbf{w}} L_i(\mathbf{w})$$

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

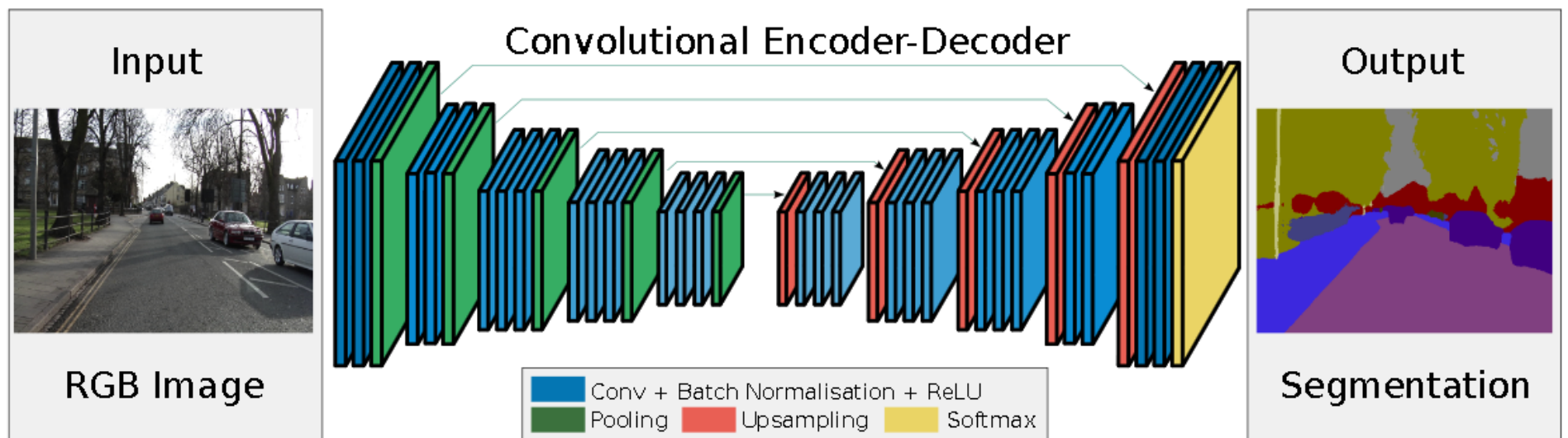
Minibatch gradient descent (computationally more efficient):

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

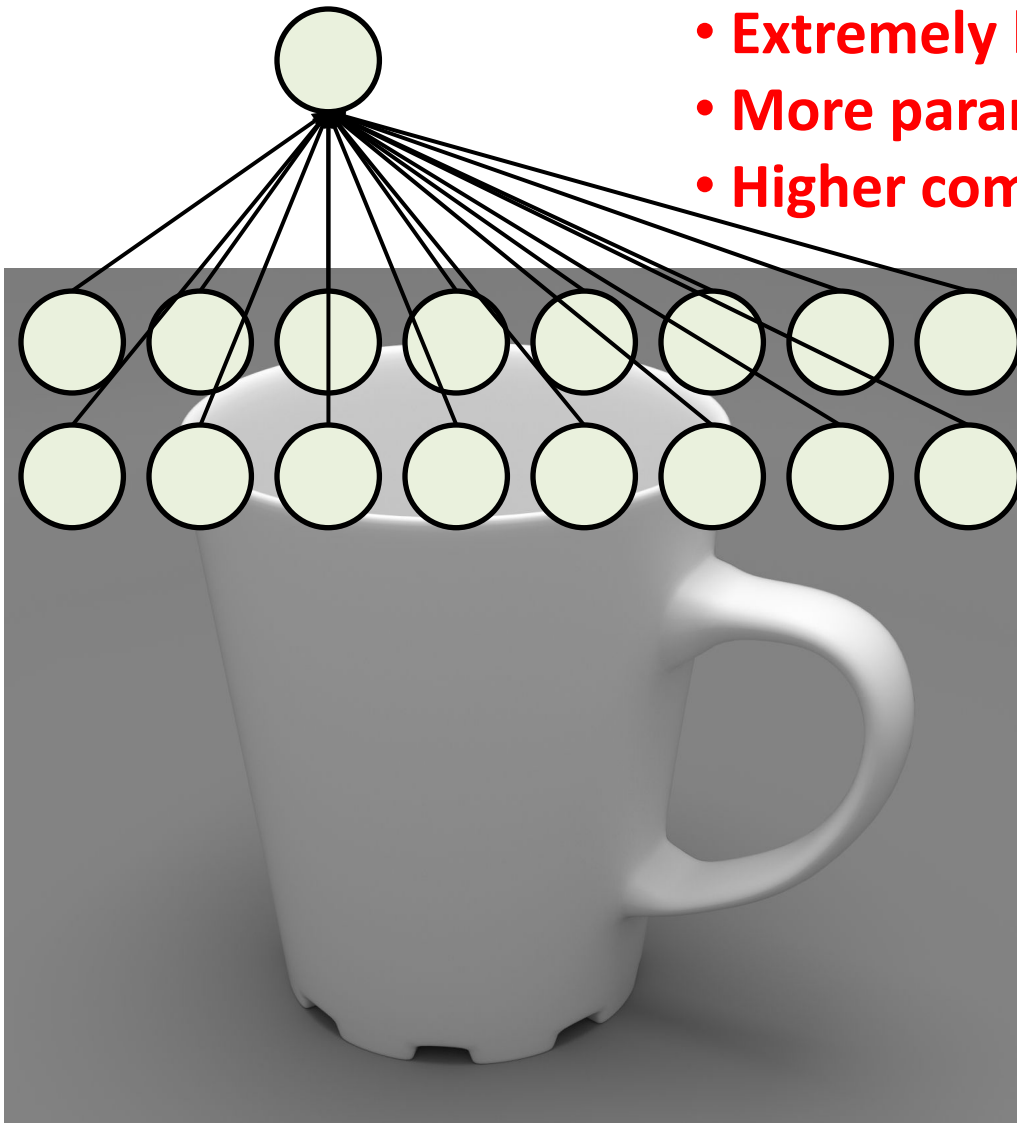
$$\nabla_{\mathbf{w}} L(\mathbf{w}) = \frac{1}{|R|} \sum_{i \in R} \nabla_{\mathbf{w}} L_i(\mathbf{w})$$

R is a random subset of training examples

Part III: ConvNets

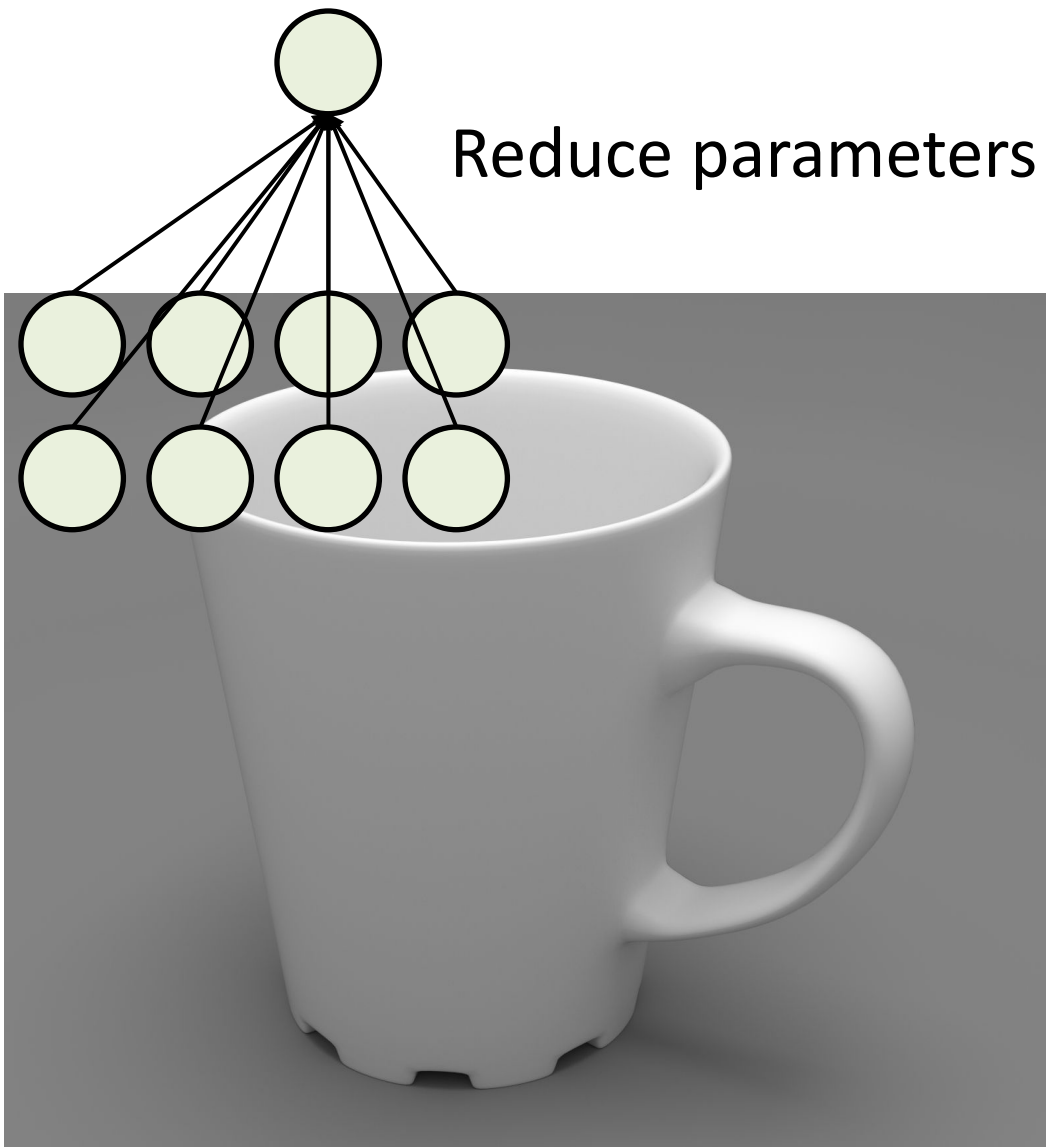


Main problem



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

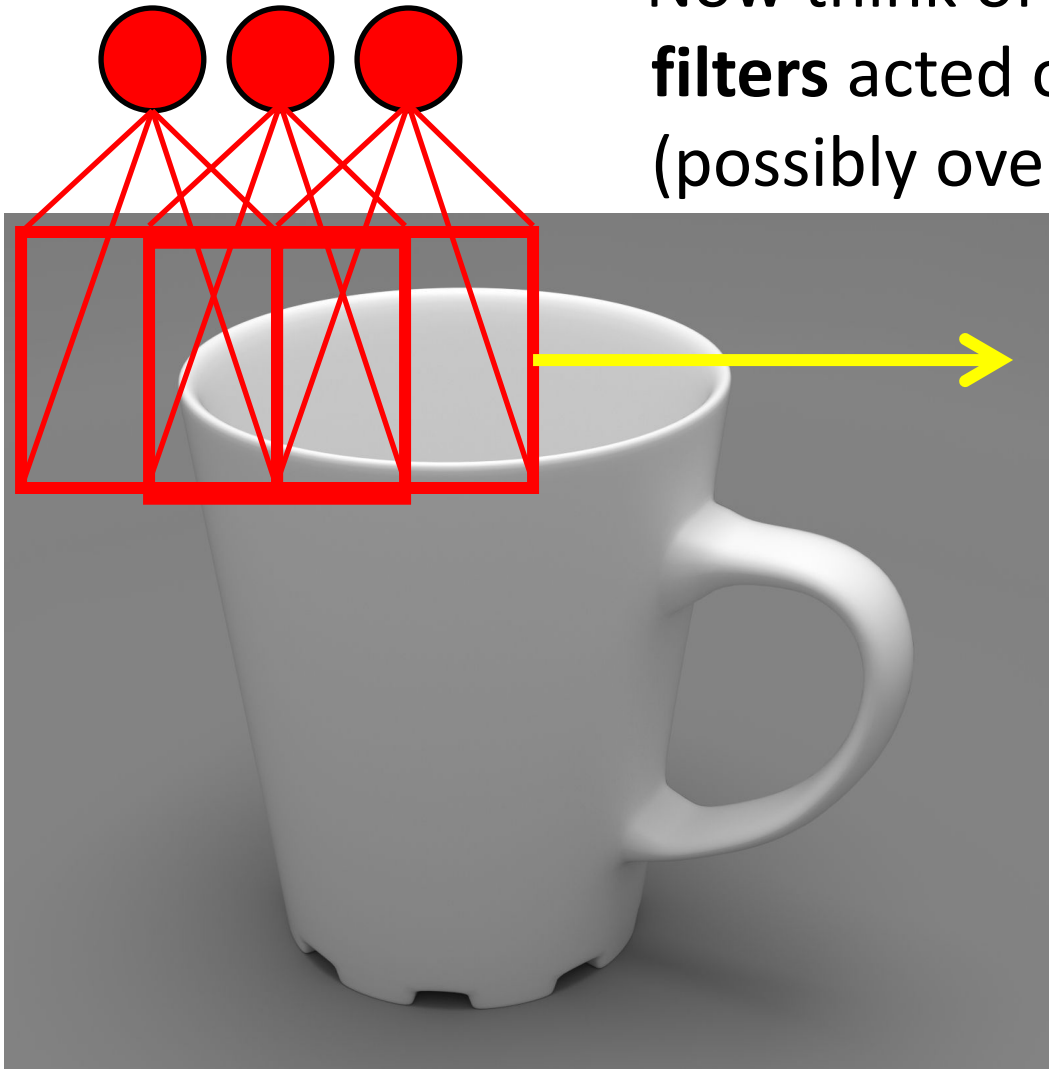
Local connectivity



Reduce parameters with **local connections!**

Neurons as convolution filters

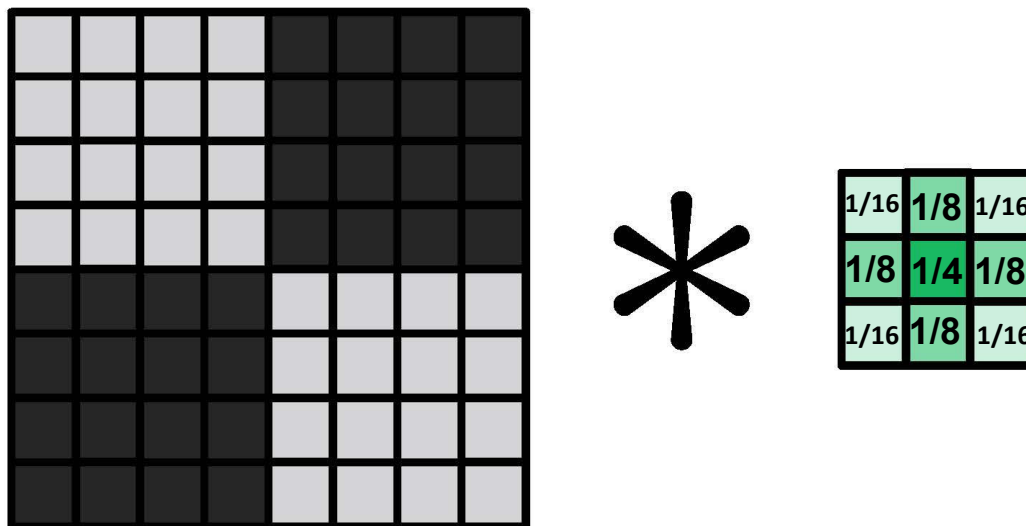
Now think of neurons as convolutional **filters** acted on small adjacent (possibly overlapping) windows



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.

3 ₀	3 ₁	2 ₂	1	0
0 ₂	0 ₂	1 ₀	3	1
3 ₀	1 ₁	2 ₂	2	3
2	0	0	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

3	3 ₀	2 ₁	1 ₂	0
0	0 ₂	1 ₂	3 ₀	1
3	1 ₀	2 ₁	2 ₂	3
2	0	0	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

3	3	2	1	0
0 ₀	0 ₁	1 ₂	3	1
3 ₂	1 ₂	2 ₀	2	3
2 ₀	0 ₁	0 ₂	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

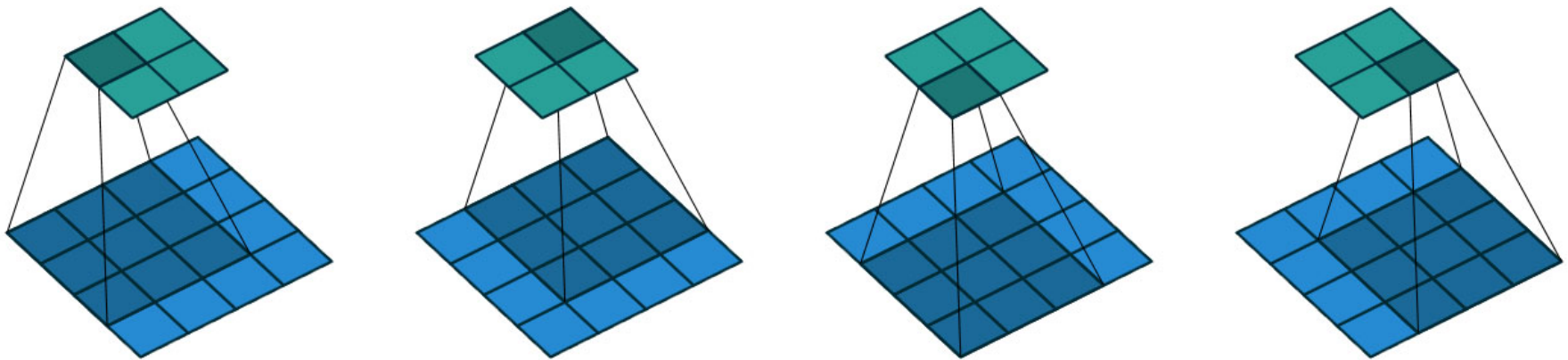
3	3	2	1	0
0	0 ₀	1 ₁	3 ₂	1
3	1 ₂	2 ₂	2 ₀	3
2	0 ₀	0 ₁	2 ₂	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

2D Convolution: boundaries

What about the pixels along the image boundary?

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

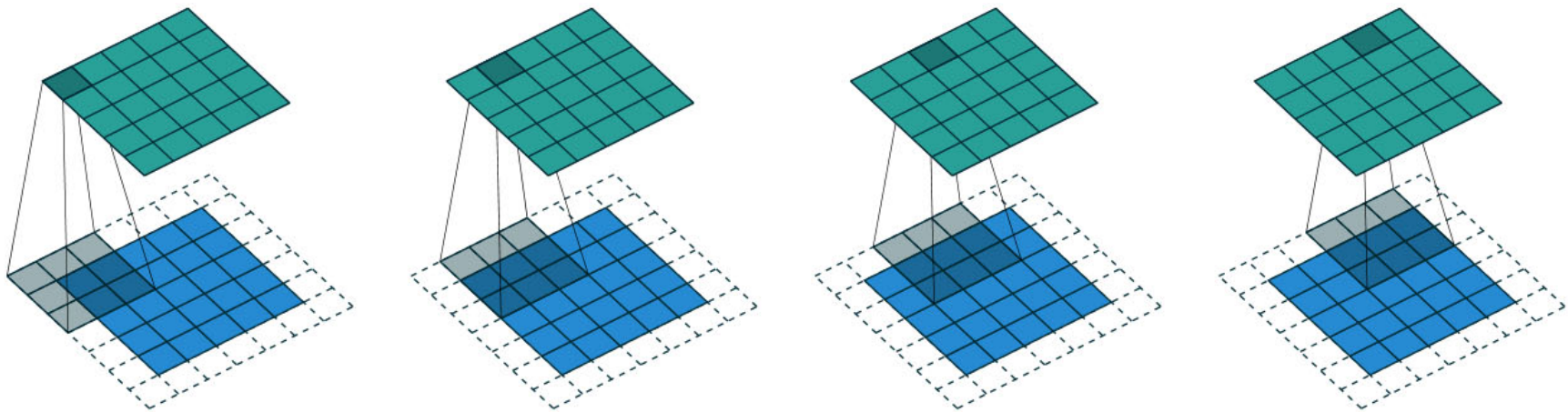


(3x3 kernel, stride 1 pixel, output has smaller size)

2D Convolution: boundaries

What about the pixels along the image boundary?

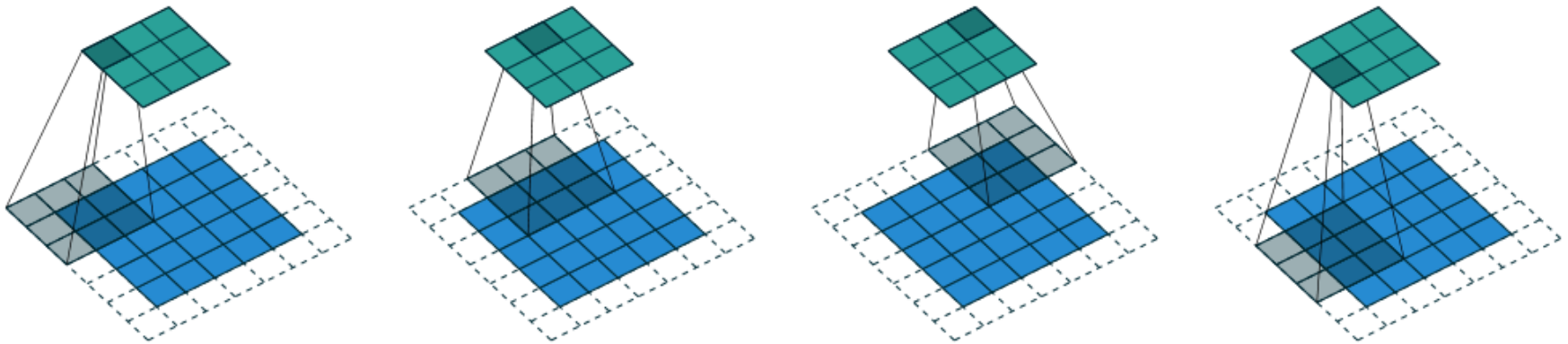
- b) extend image boundary using constant intensity pixels (padding)
or reflect pixels along the border



(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)



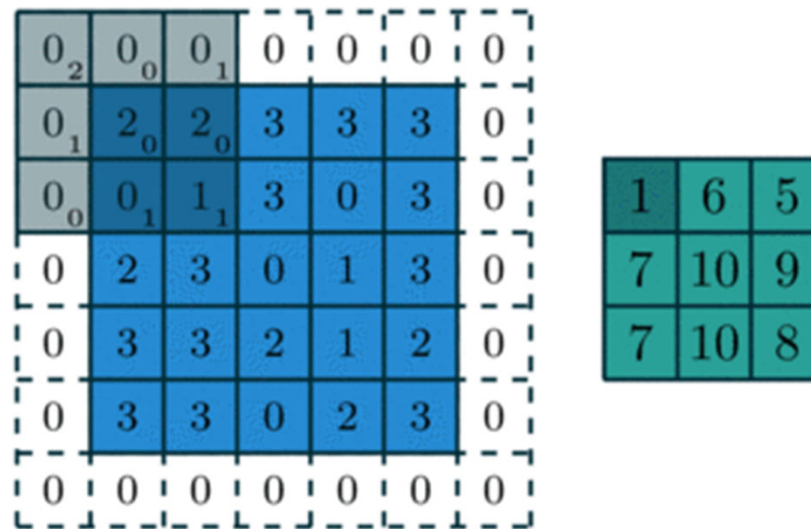
(3x3 kernel, stride 2 pixels, padding 1 pixel, output has smaller size)

2D Convolution Boundaries

How do we compute output size given convolution setting?

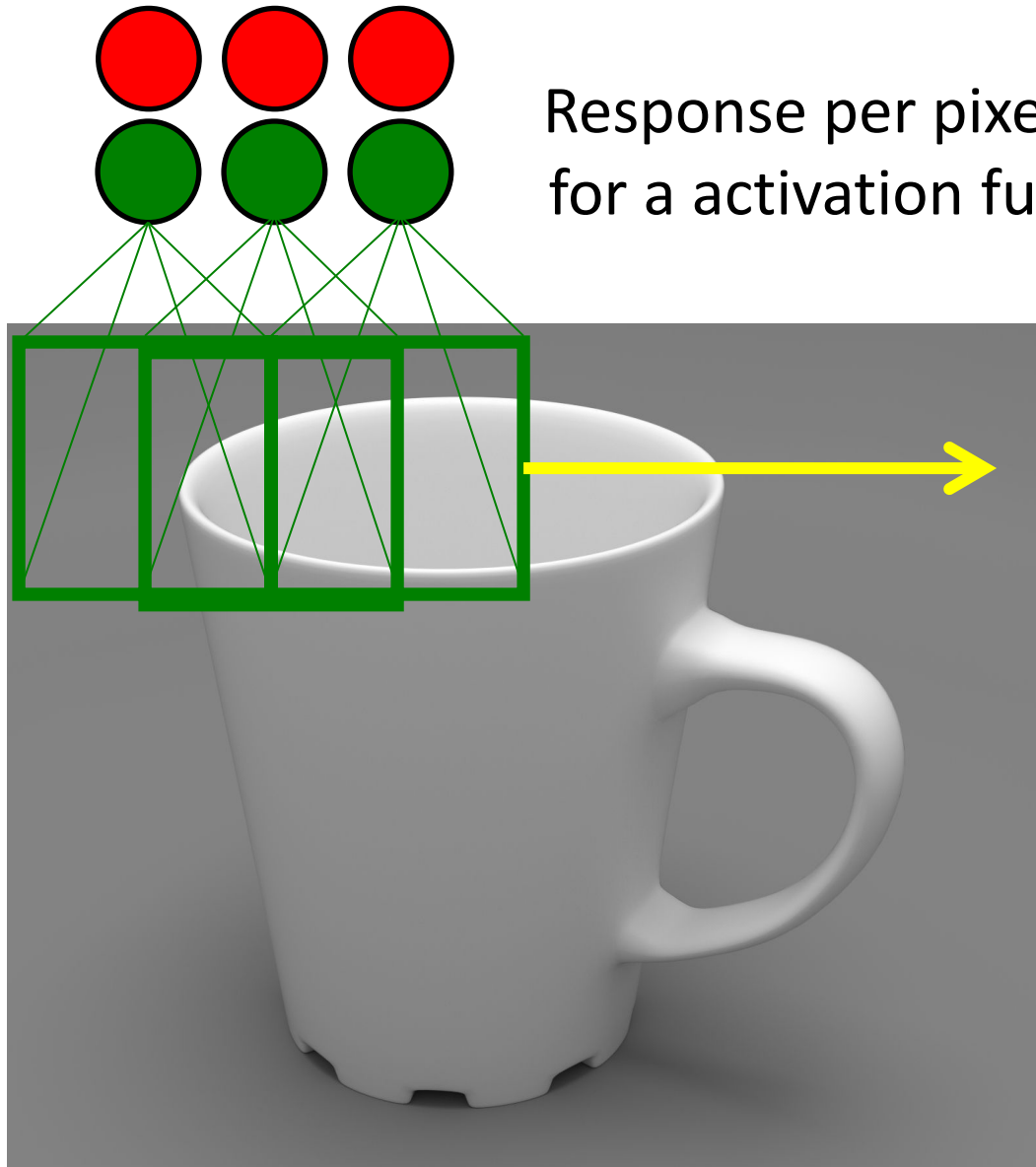
output size is:

$$\lceil (\text{input size}) - (\text{receptive field size}) + 2 * (\text{padding size}) \rceil / \text{stride} + 1$$



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

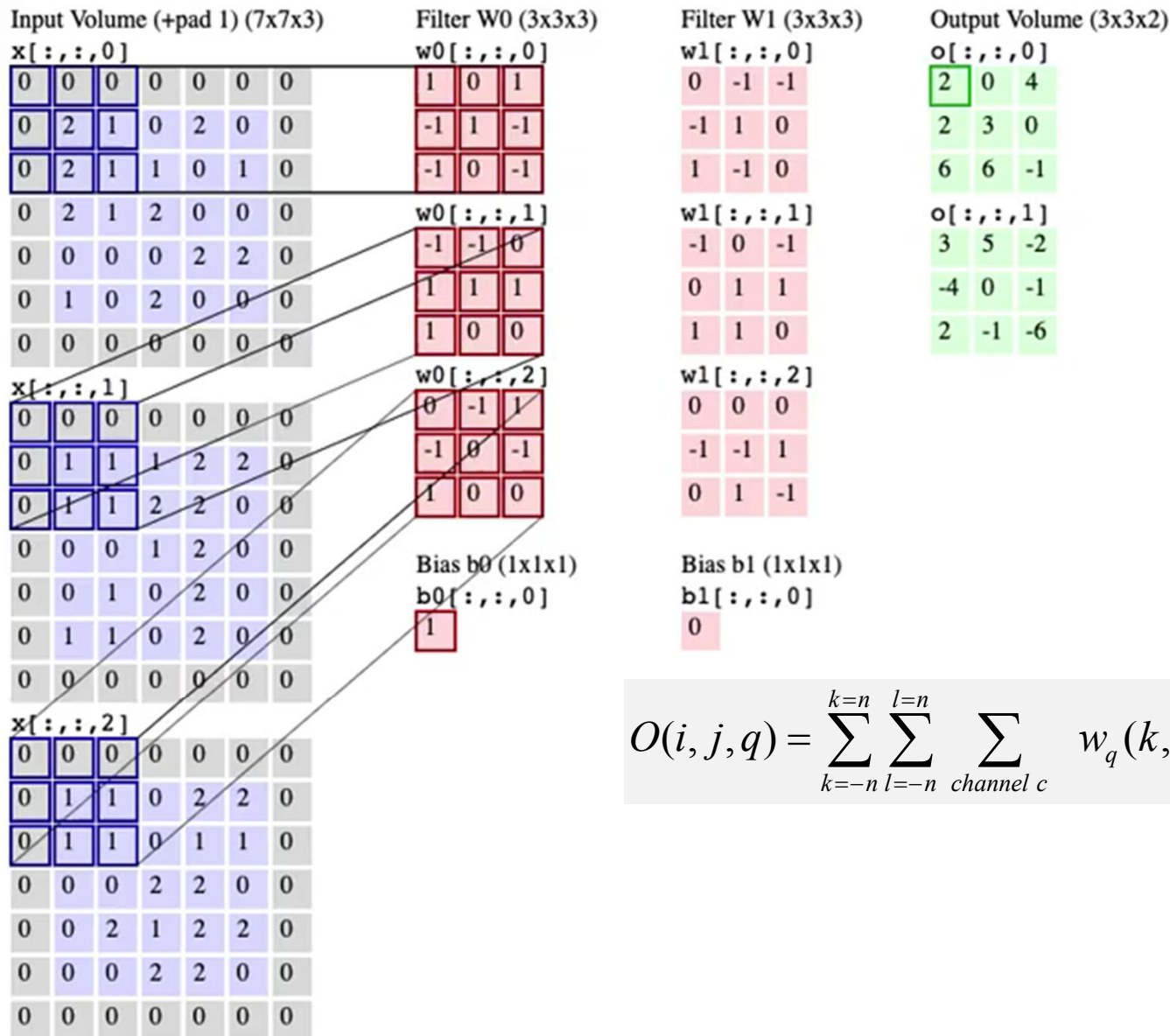
Can have many filters!



Response per pixel (i,j) , per filter f
for a activation function g : $h_{i,j,f} = g(\mathbf{w}_f \cdot \mathbf{x}_{i,j})$

↑
ReLu, LReLu etc

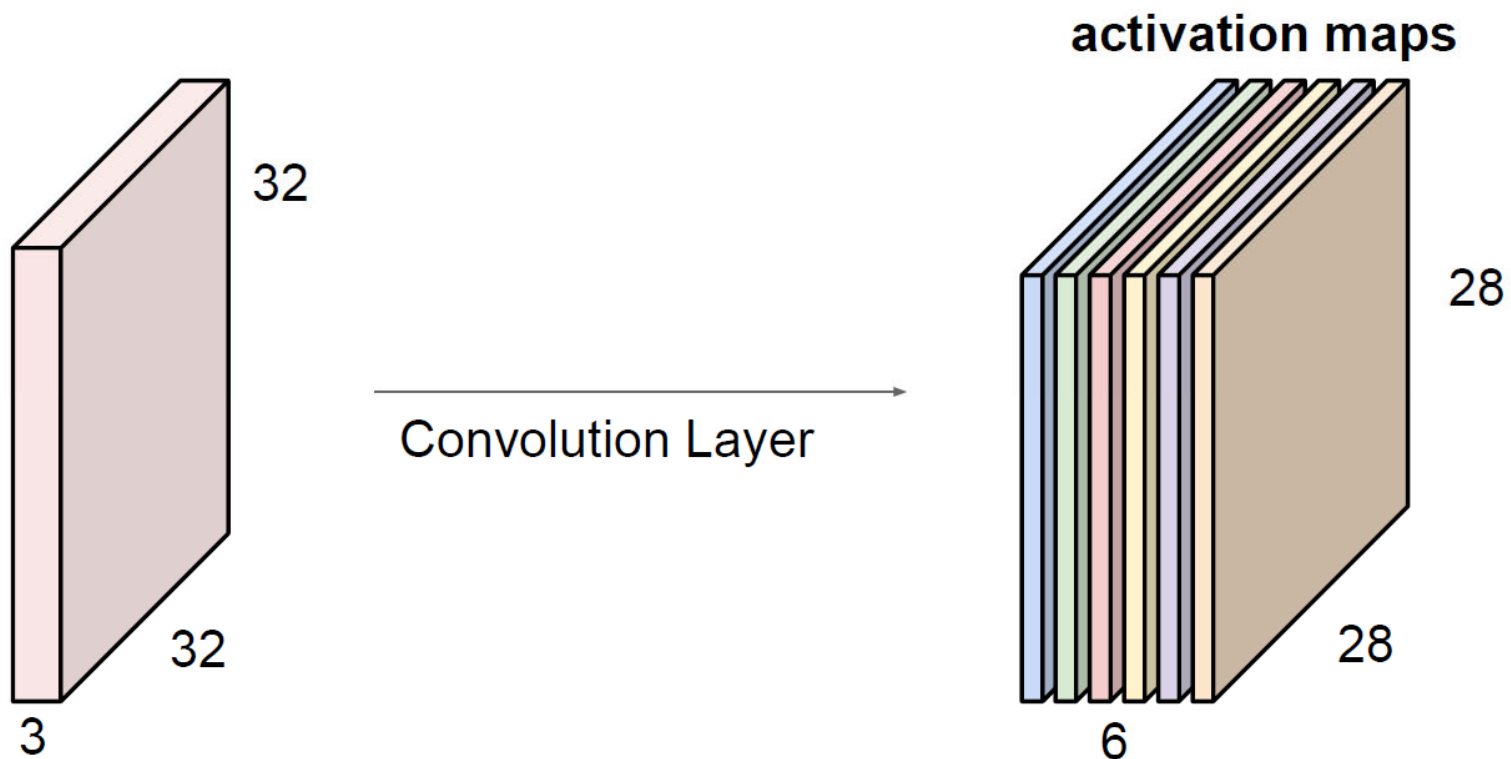
Convolution when input has multiple channels



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

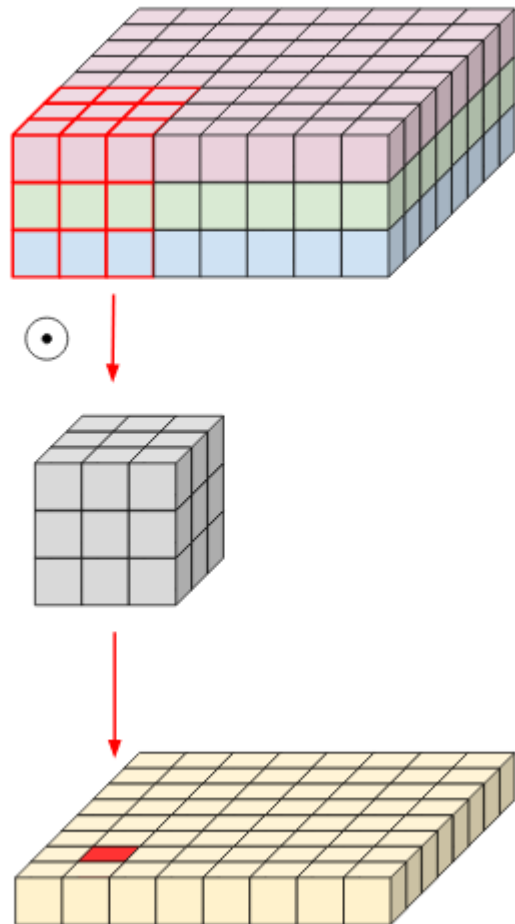
Convolution layer

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

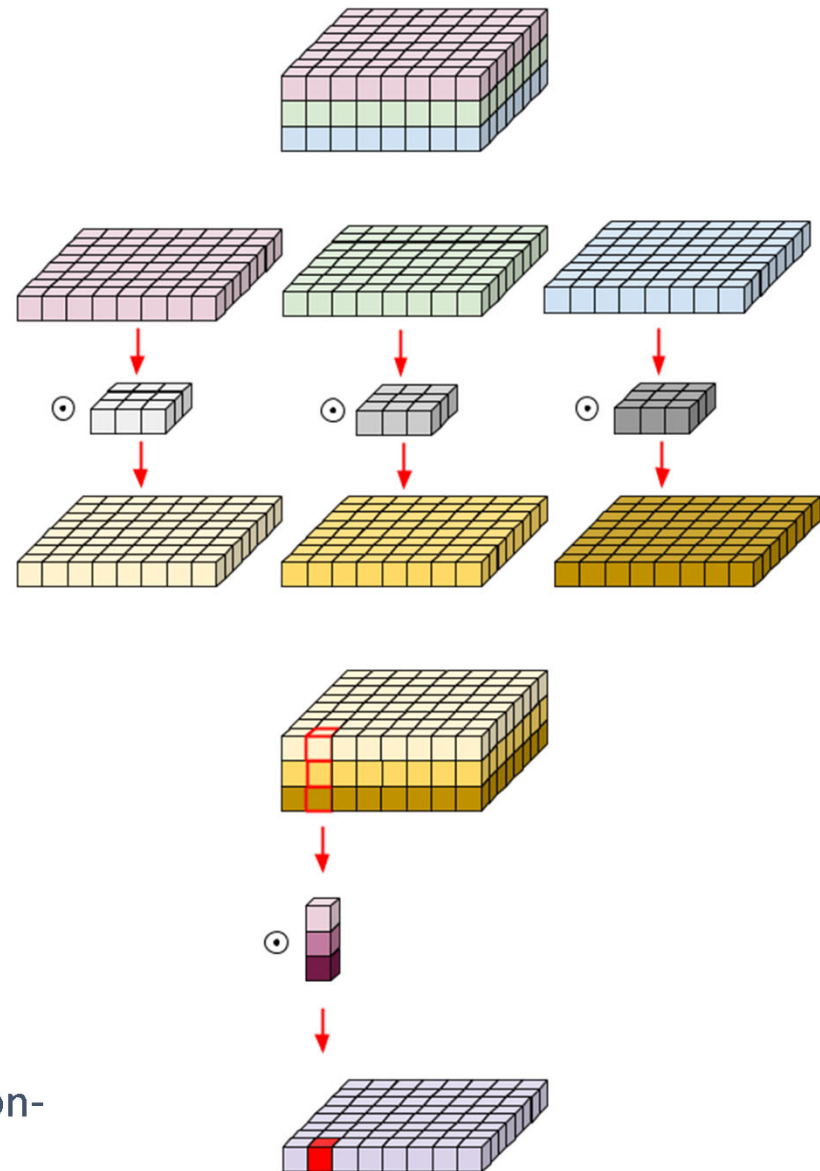


We stack these up to get a “new image” of size 28x28x6!

“Regular” convolution



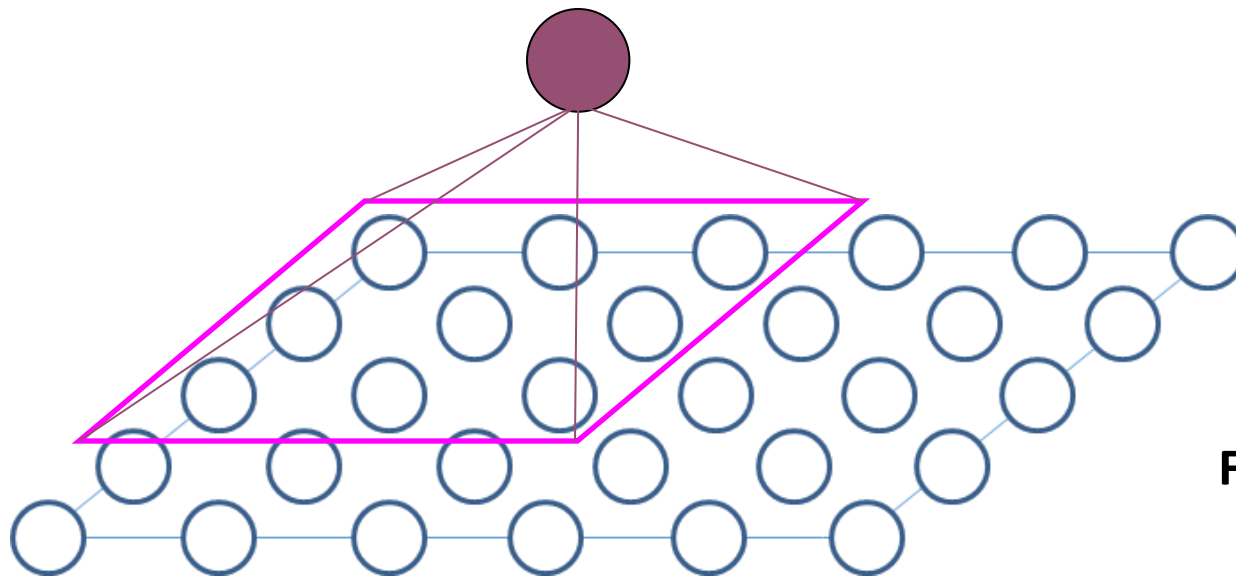
“Depthwise separable” convolution



<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:

$$h_{p',f} = \text{avg}_p(\mathbf{x}_p)$$

Fixed filter (e.g., Gaussian):

$$h_{p',f} = w_{\text{gaussian}} \cdot \mathbf{x}_p$$

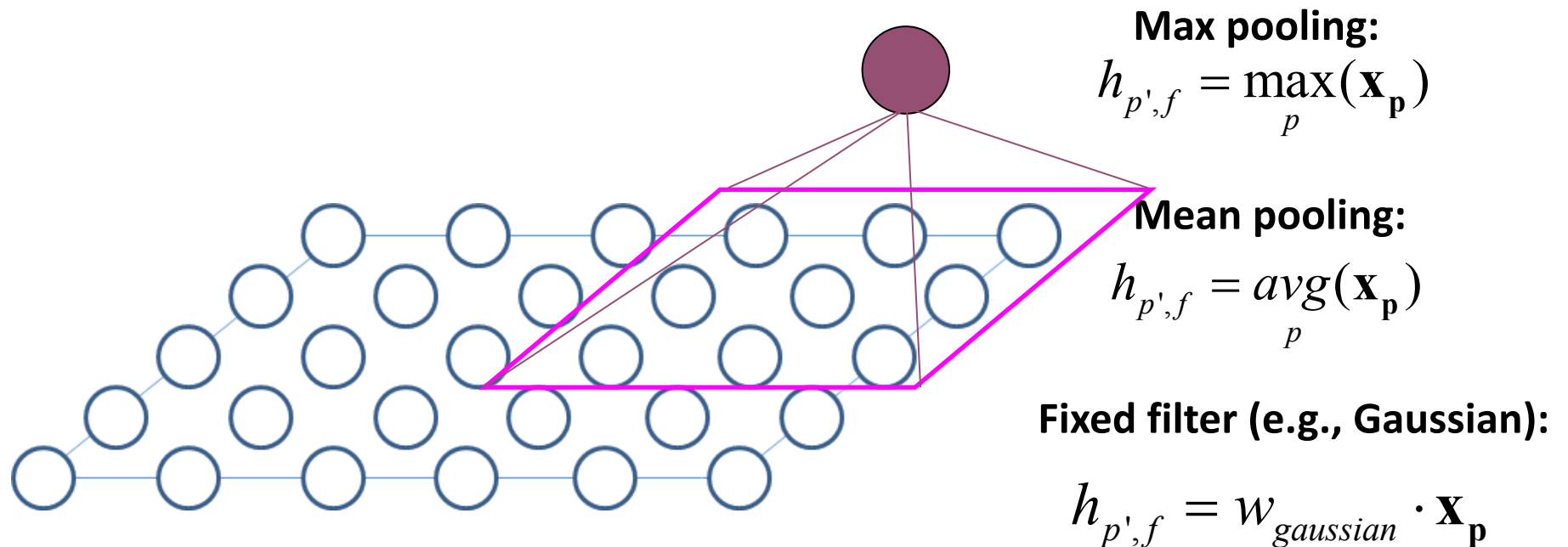
Progressively reduce the resolution of the image, so that the next convolutional filters are applied on larger scales

[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



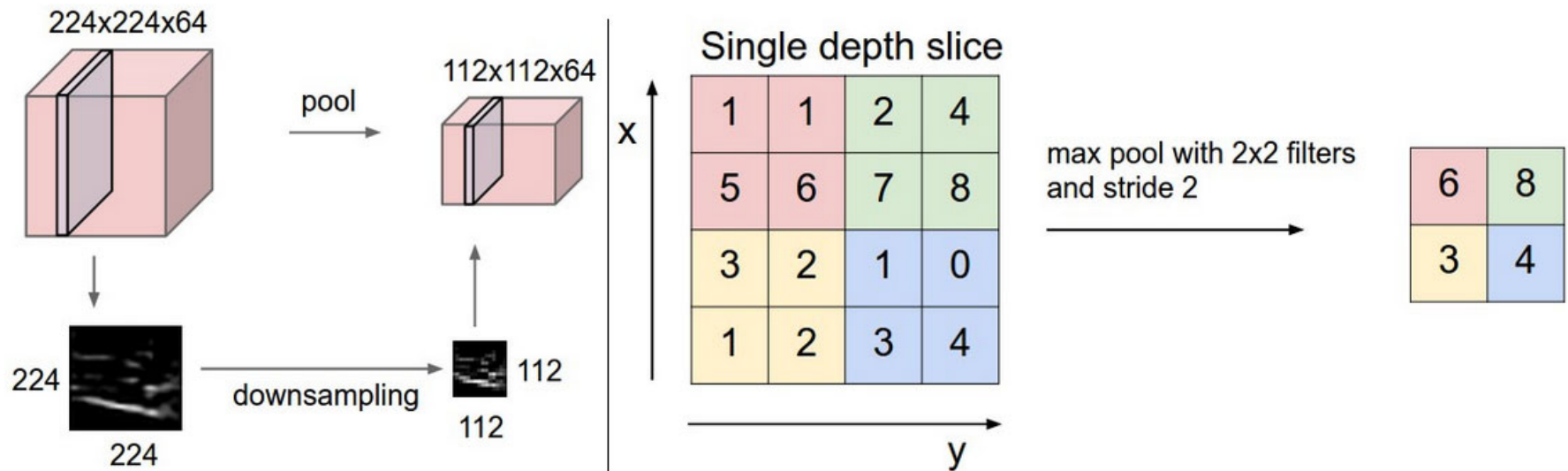
Progressively reduce the resolution of the image, so that the next convolutional filters are applied on larger scales

[Scherer et al., ICANN 2010]

[Boureau et al., ICML 2010]

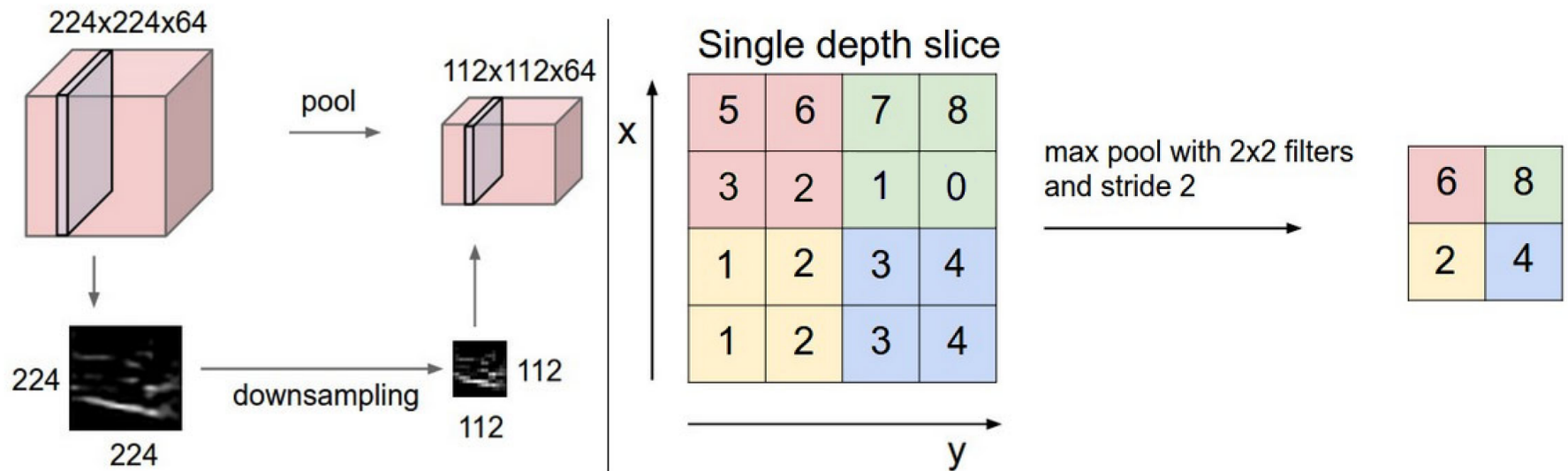
Pooling layer

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



Pooling layer

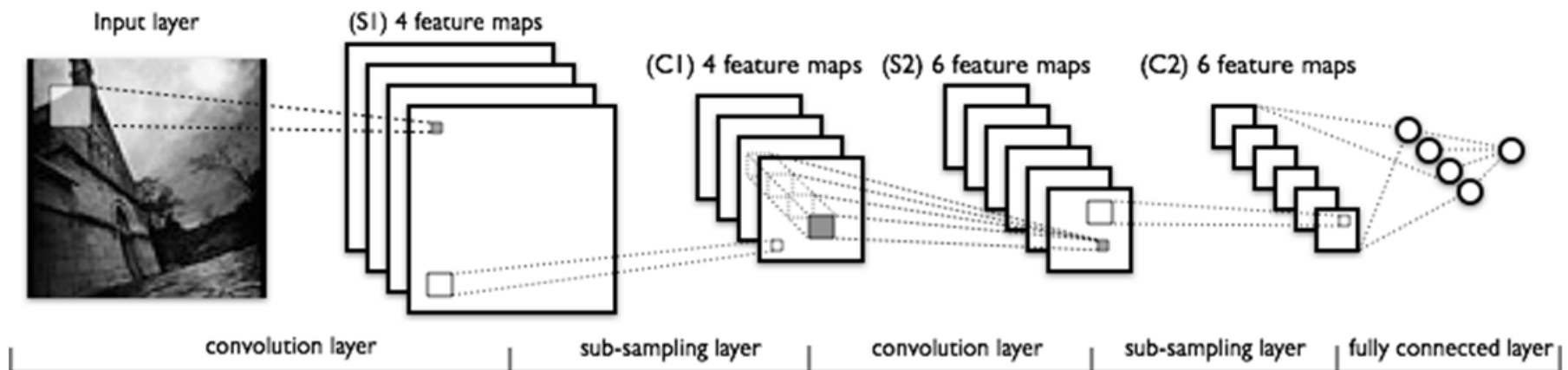
Max-pooling promotes some **invariance** to input noise, or perturbations



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:

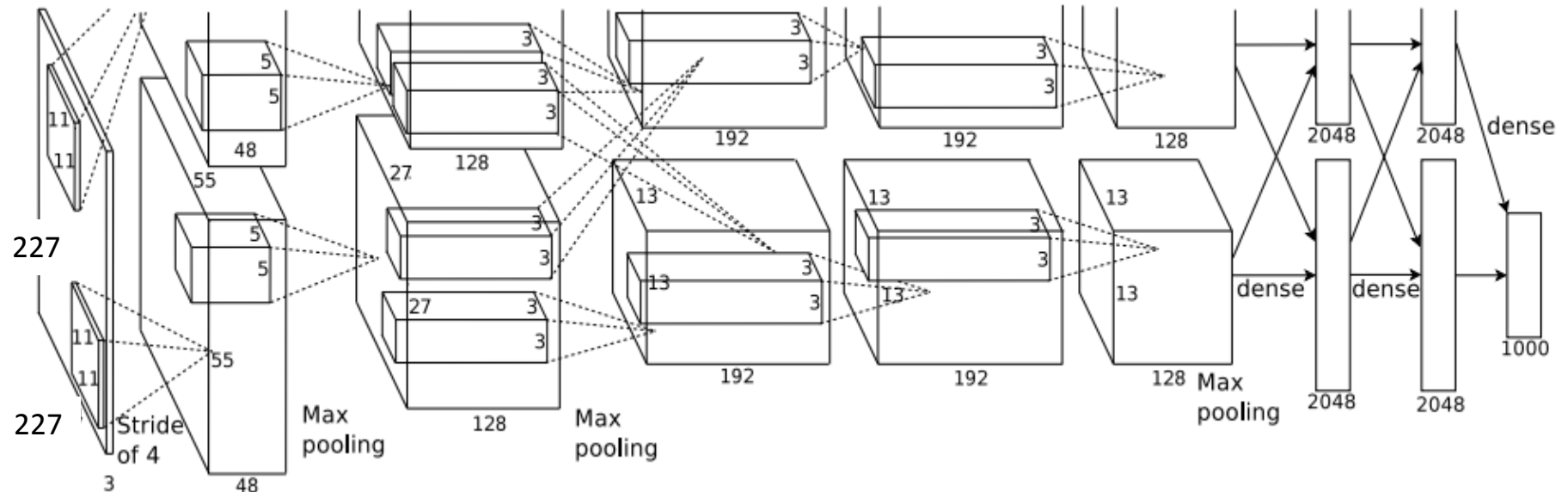
Convolutional layers with ReLus

Max-pooling layers

SGD on GPU with momentum, L2 reg., dropout

Applied to image classification (ImageNet competition – top runner & game changer)

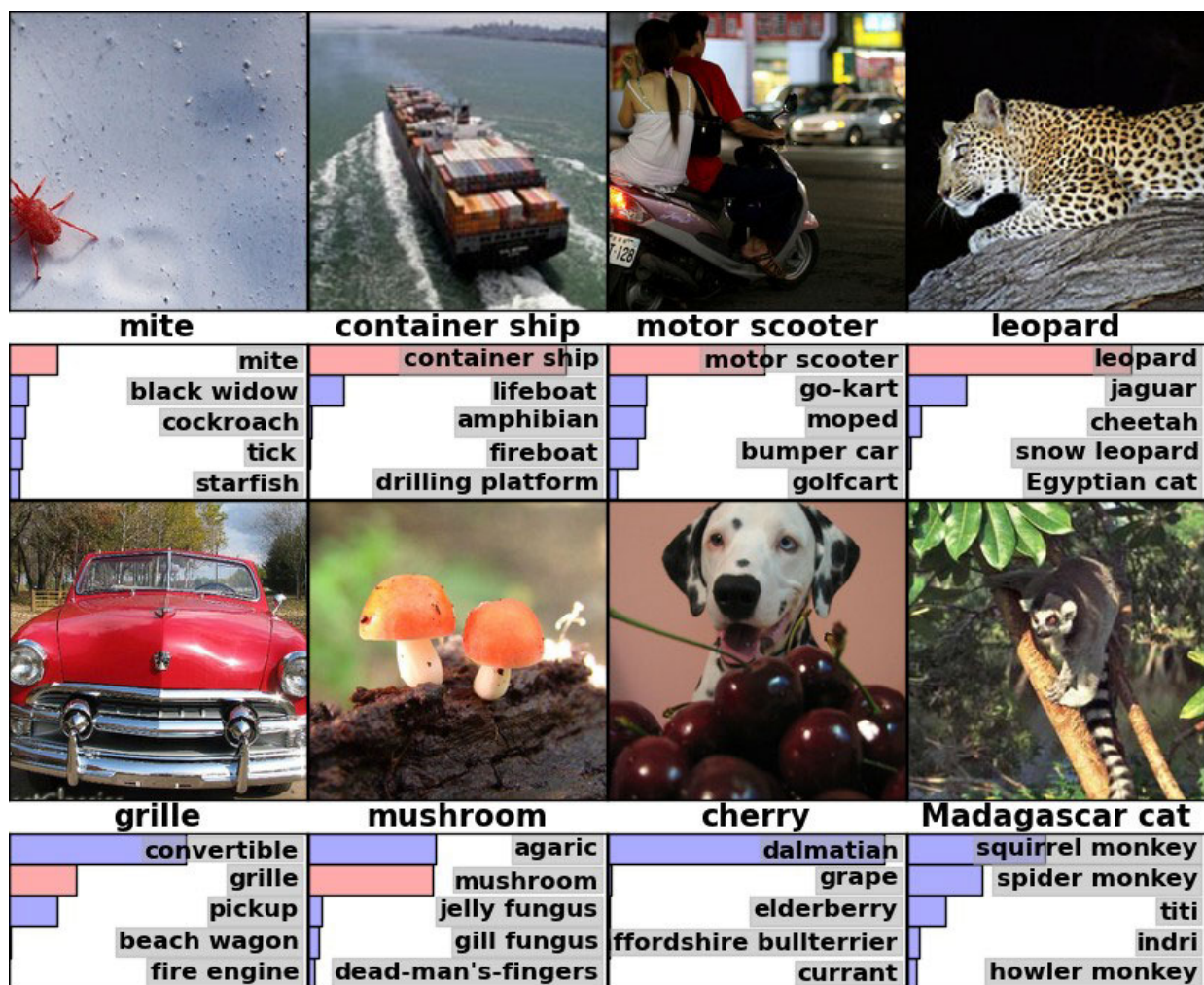
Trained on a large dataset (1M images)!



Krizhevsky et al., NIPS 2012

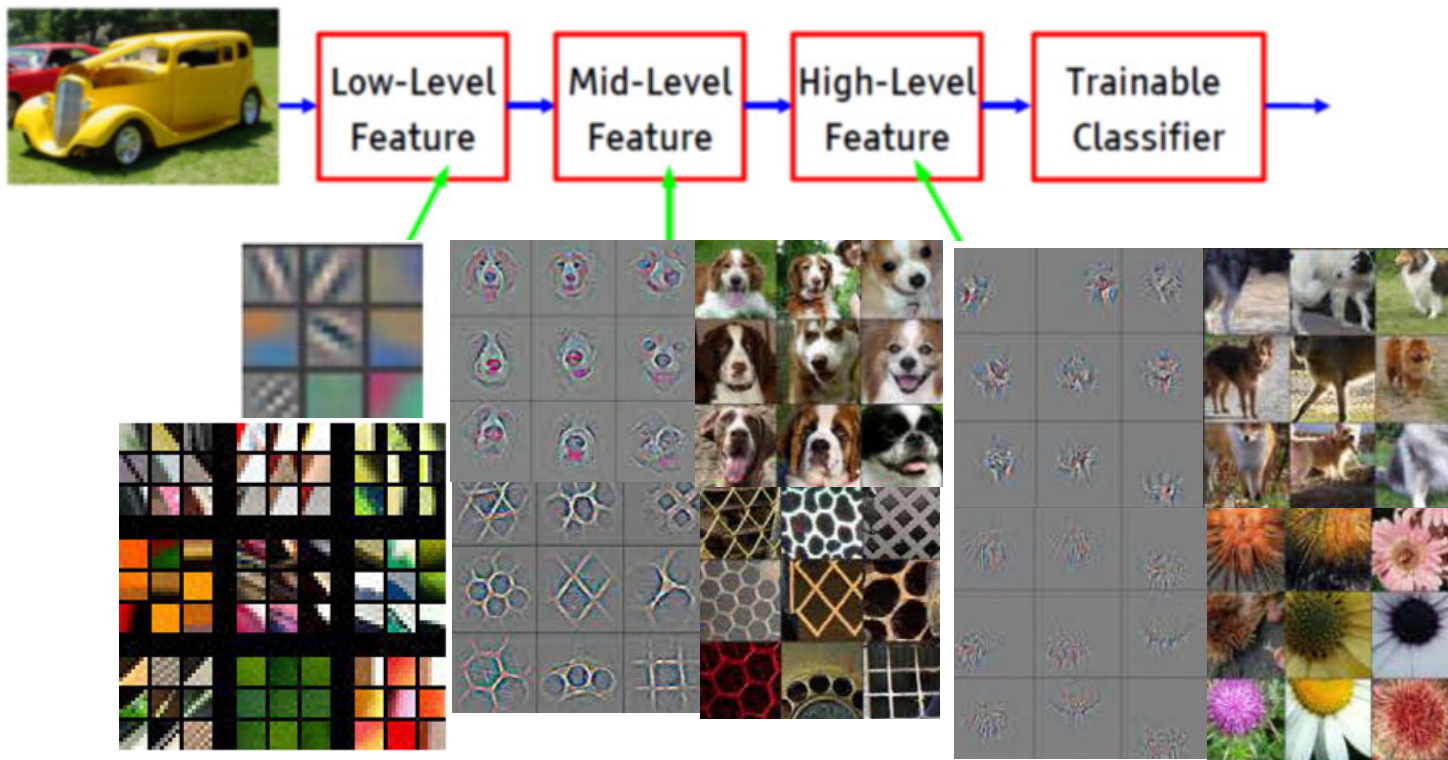
Application: Image-Net

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



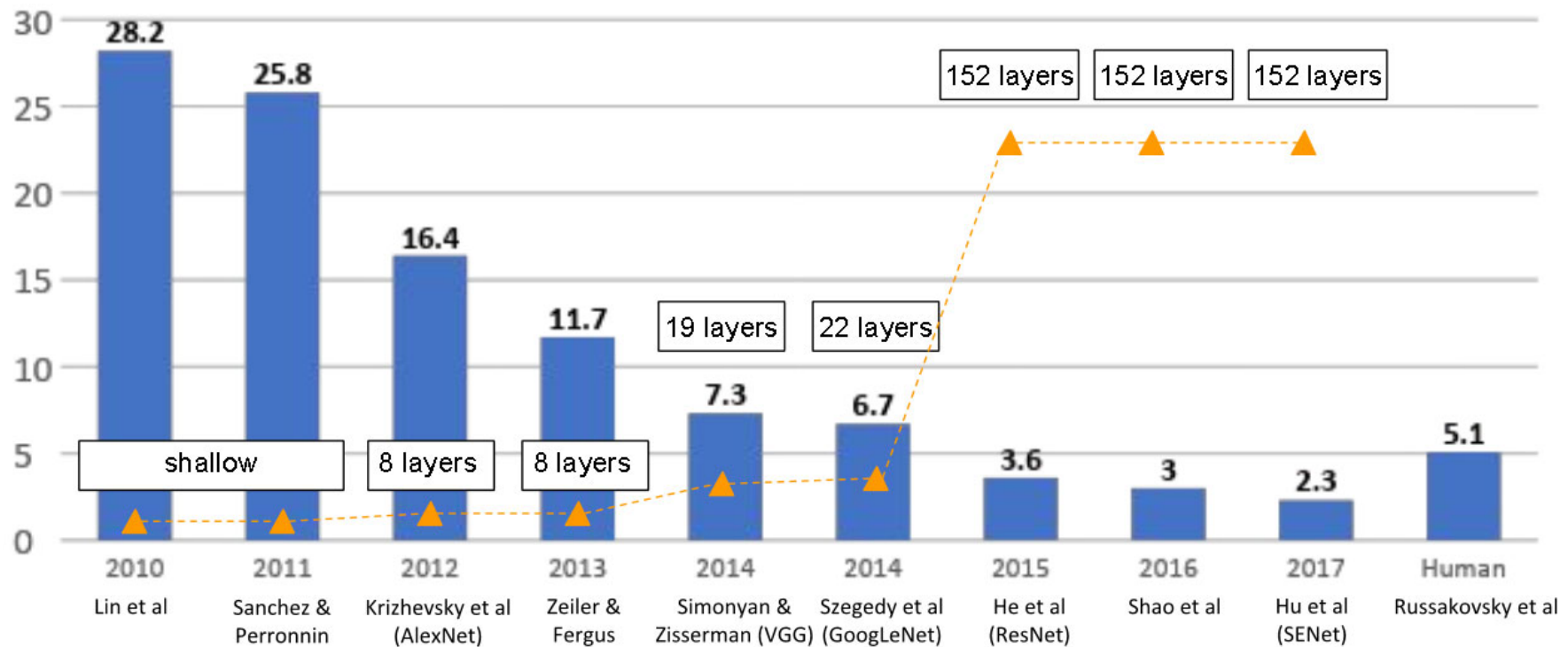
Learned filters

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

Since 2012, huge progress!



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?