

# **AN OVERVIEW OF DEEP LEARNING (AND ITS CHALLENGES FOR TECHNICAL COMPUTING)**

GRAHAM TAYLOR

SCHOOL OF ENGINEERING  
UNIVERSITY OF GUELPH

TCMM 2014  
Leuven, Belgium

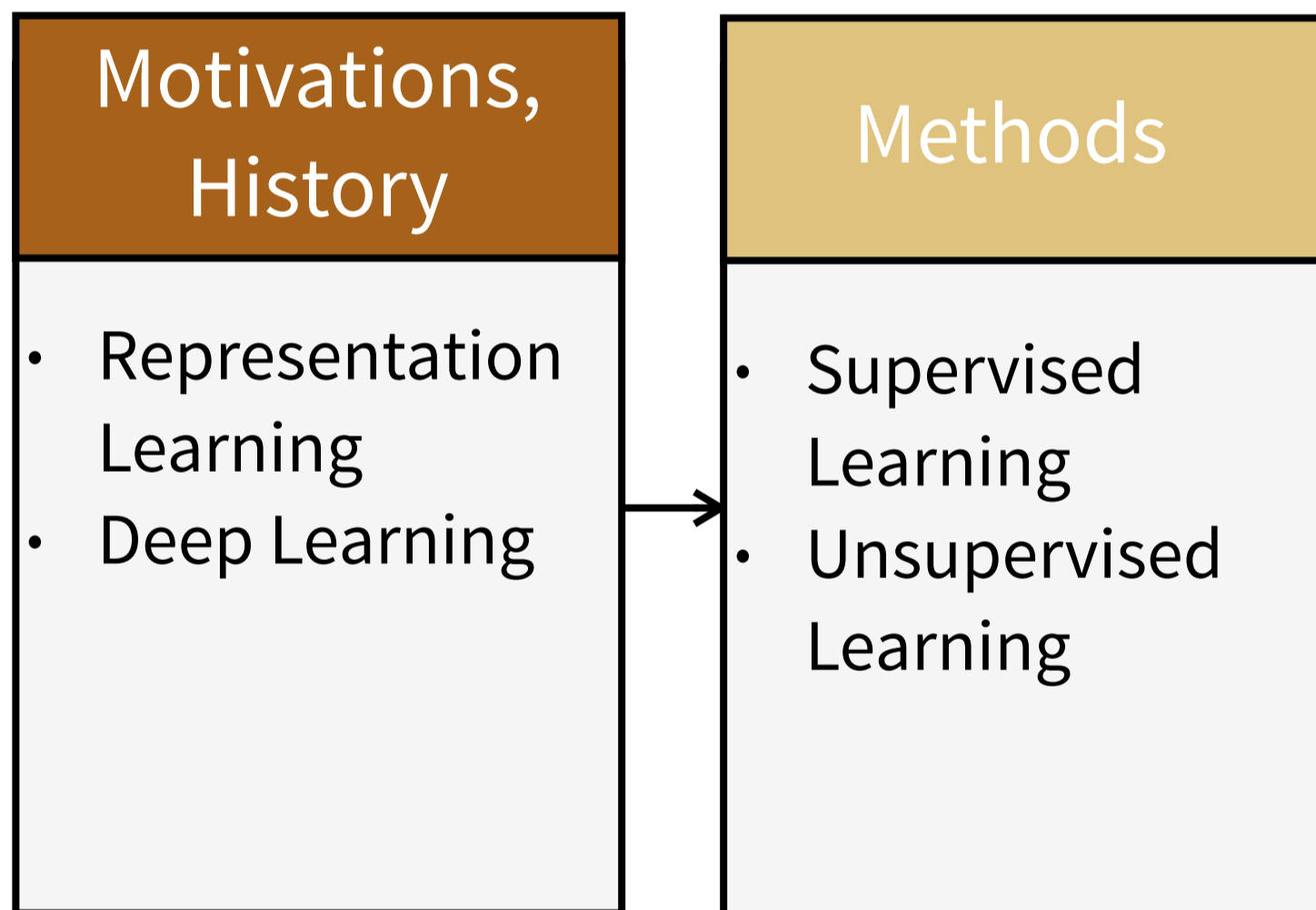
# Roadmap

# Roadmap

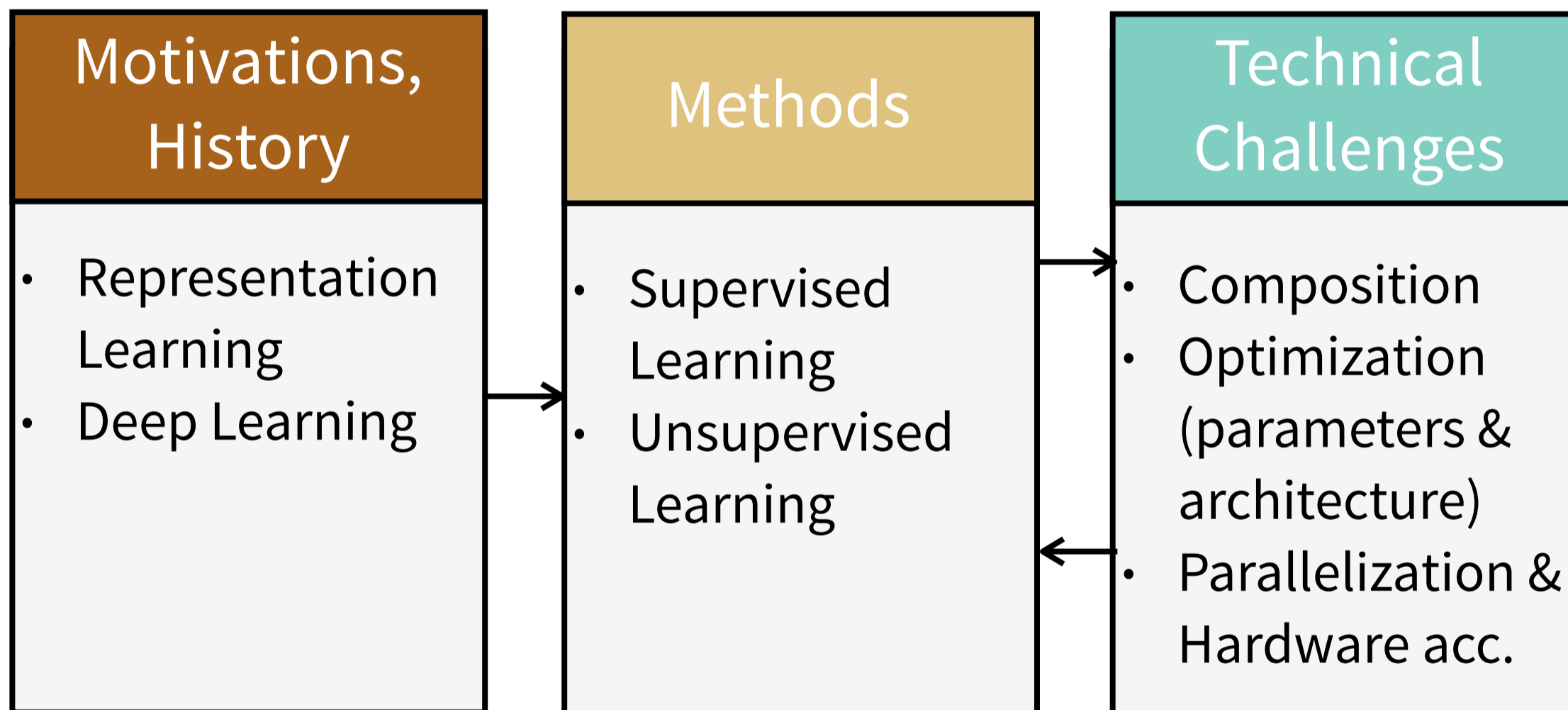
## Motivations, History

- Representation Learning
- Deep Learning

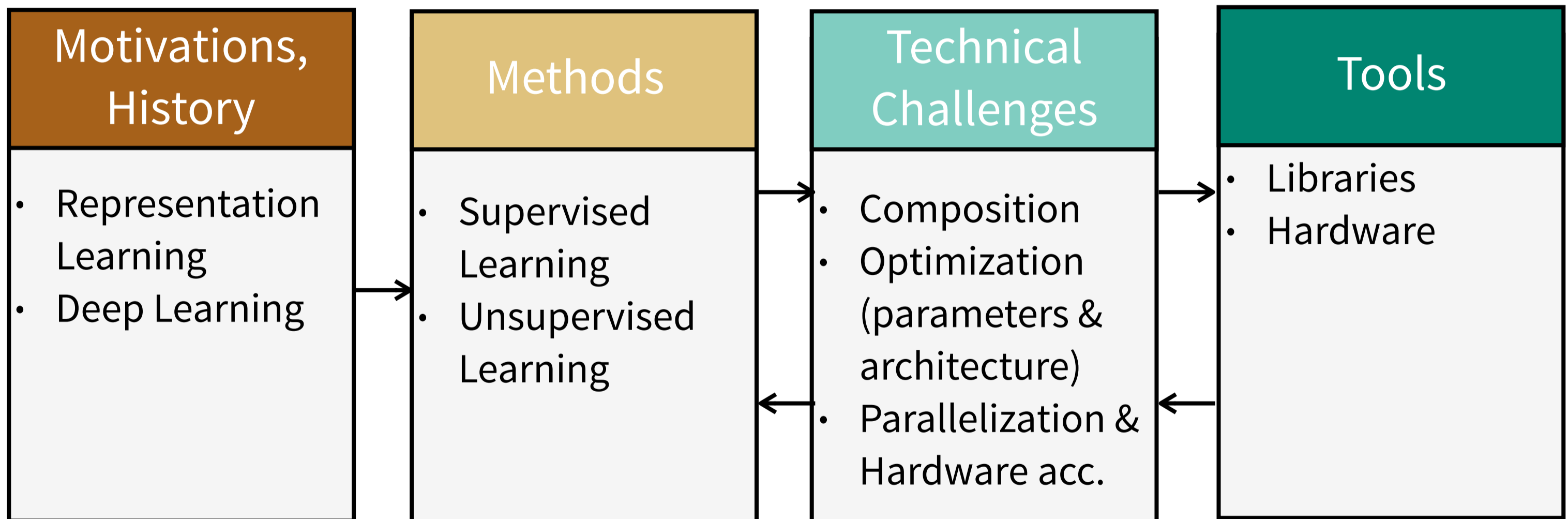
# Roadmap



# Roadmap



# Roadmap

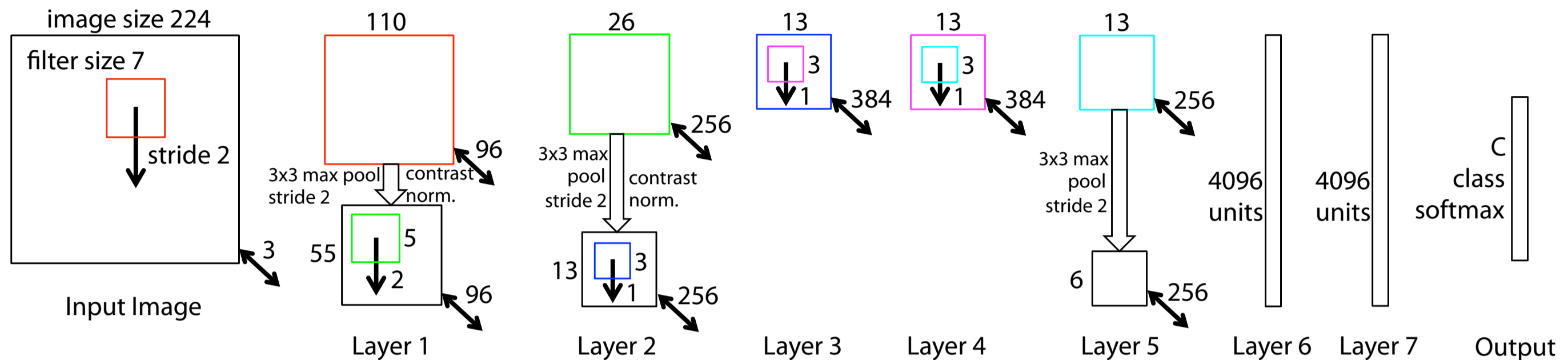


# Learning Representations

- Representations are “Concepts” or “Abstractions” that help us make sense of the **variability in data**
- Often **hand-designed** to have desirable properties: e.g. sensitive to variables we want to predict, less sensitive to other factors explaining variability
- Representation learning algorithms are machine learning algorithms which involve the **learning of features** or explanatory factors

# Deep Learning

Example: A Convolutional Net  
“An Astounding Baseline for Recognition”

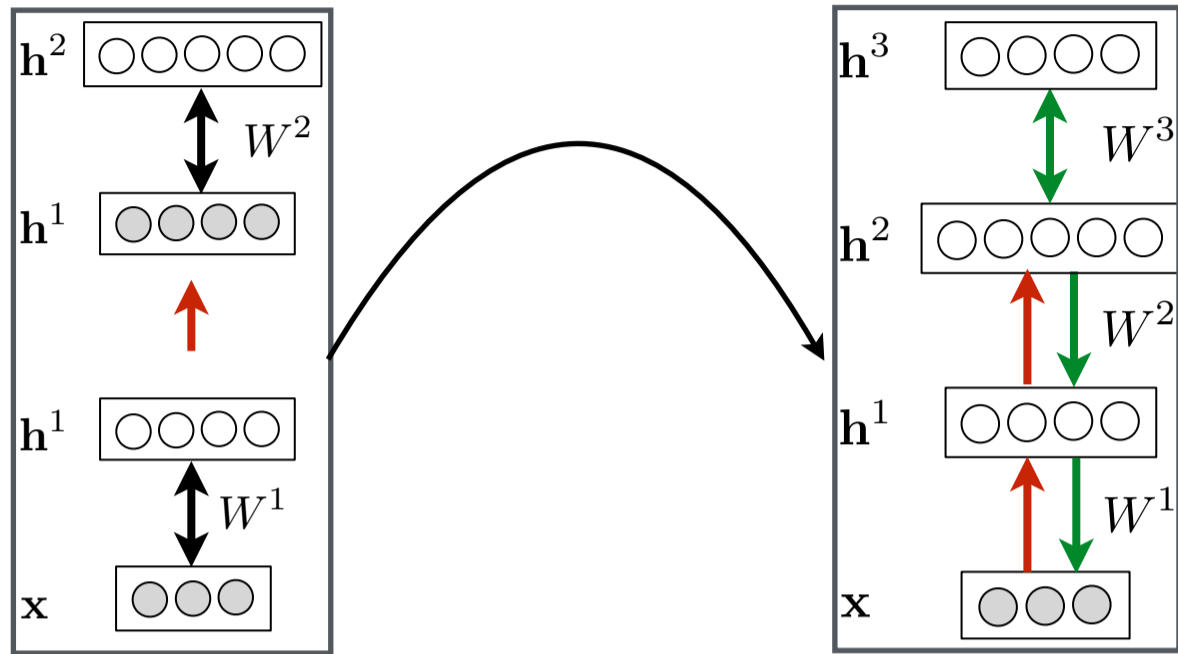


- Learn multiple layers of representation, corresponding to different levels of abstraction
  - **theory** on the advantage of depth (Hasted et al 1986 & 1991), (Bengio et al. 2007), (Bengio and Delalleau 2011), (Braverman 2011)
  - exploiting **composition** gives an exponential gain in representational power (humans organize ideas and concepts hierarchically!)
  - biologically inspired learning - the brain is deep!



# Deep Learning Timeline

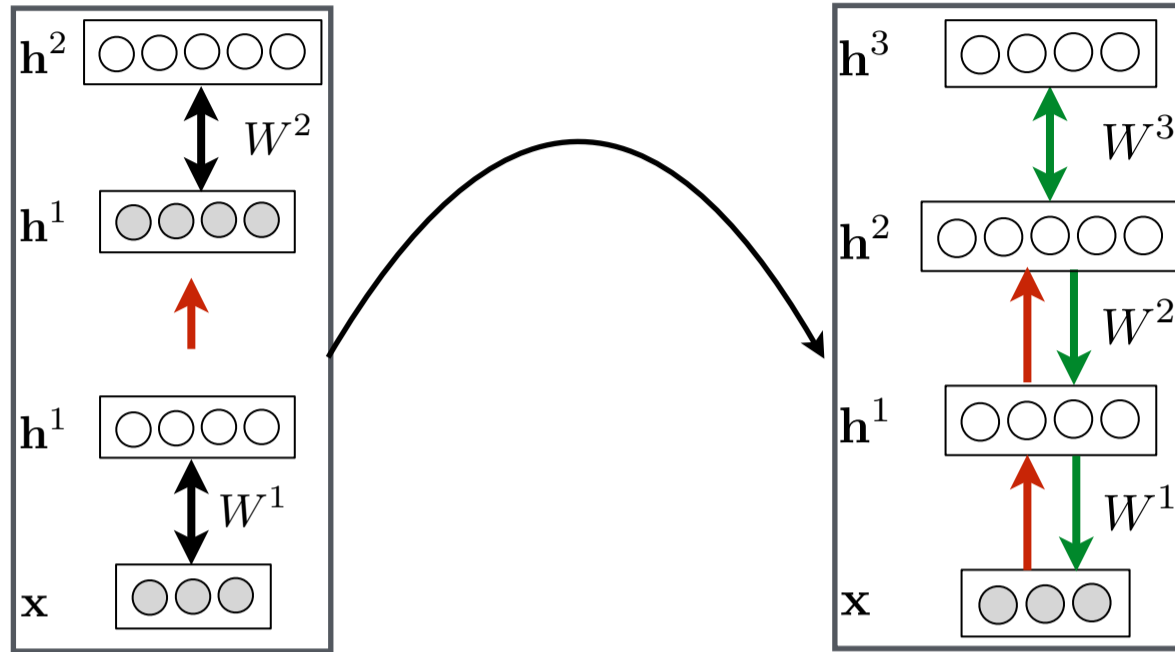
# Deep Learning Timeline



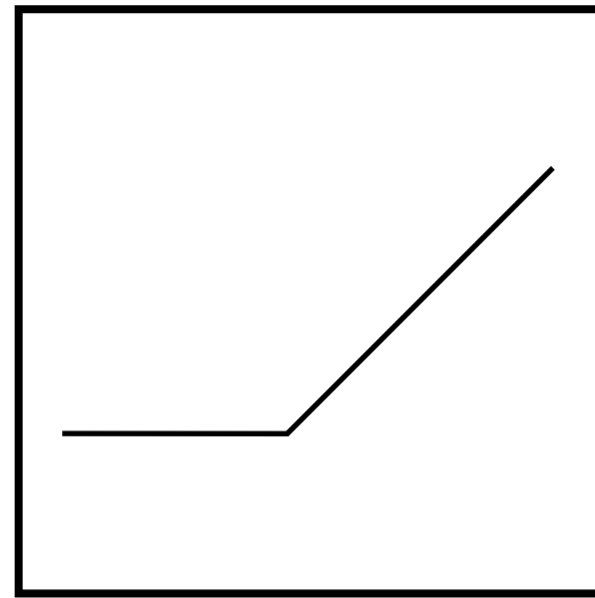
2006

Pre-training & stacking

# Deep Learning Timeline

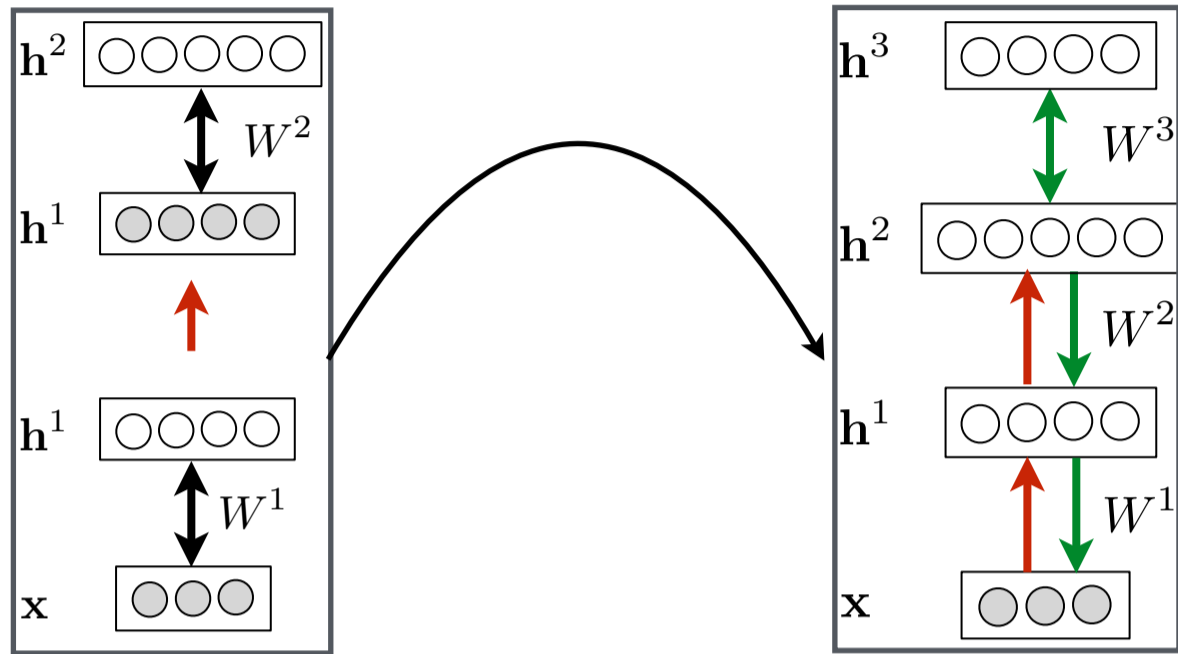


2006  
Pre-training & stacking

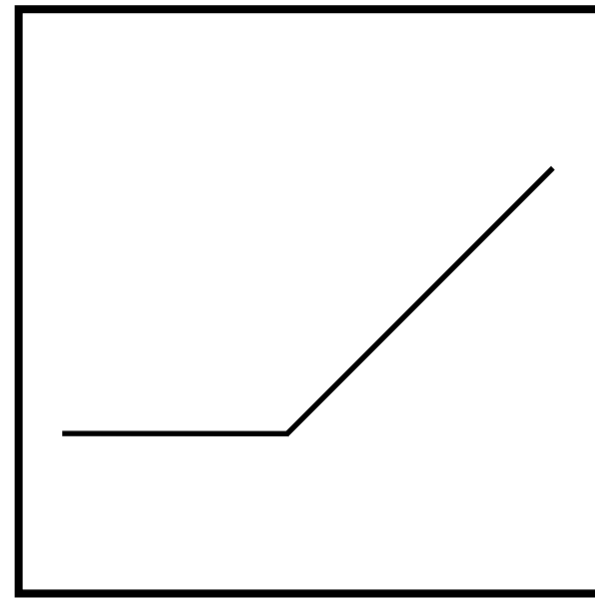


2010  
Rectifier units explored

# Deep Learning Timeline



2006  
Pre-training & stacking

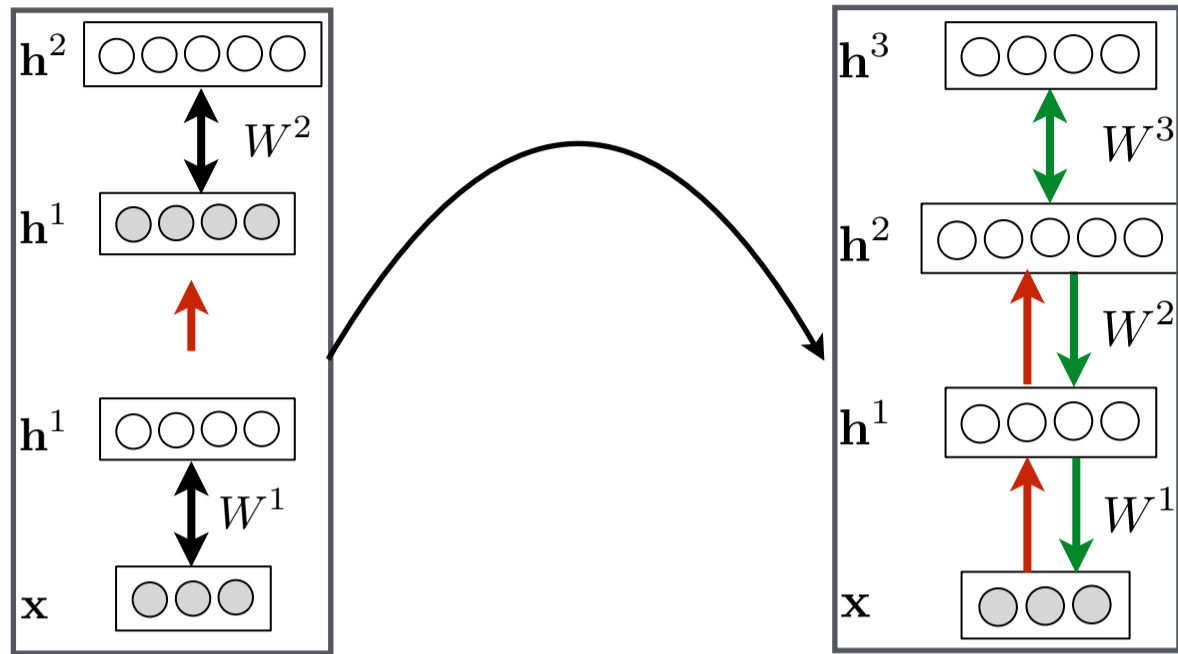


2010  
Rectifier units explored

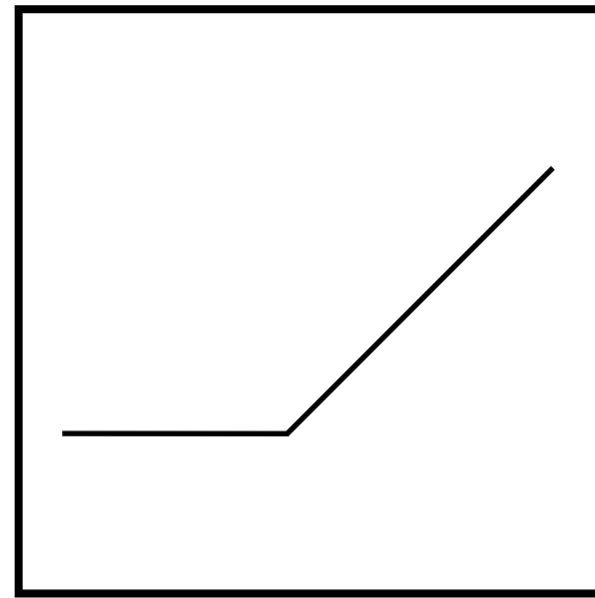


2010-2012  
Most major speech systems  
incorporate DL

# Deep Learning Timeline



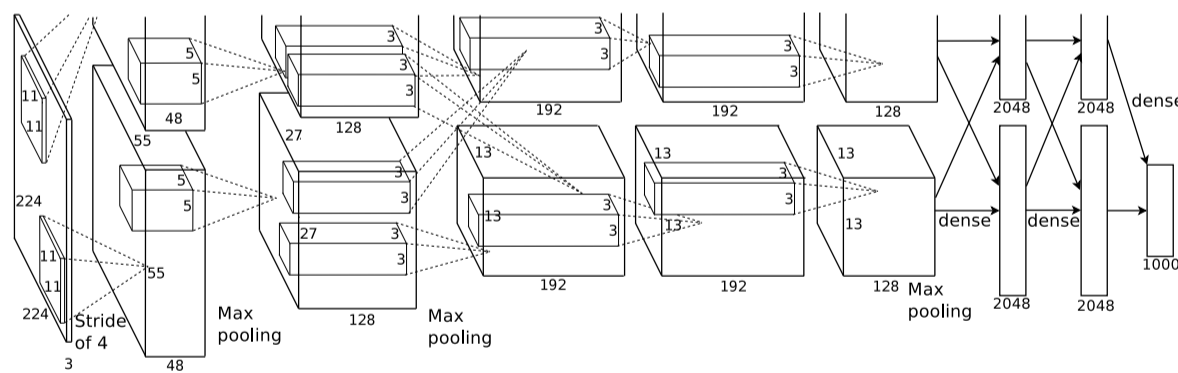
2006  
Pre-training & stacking



2010  
Rectifier units explored

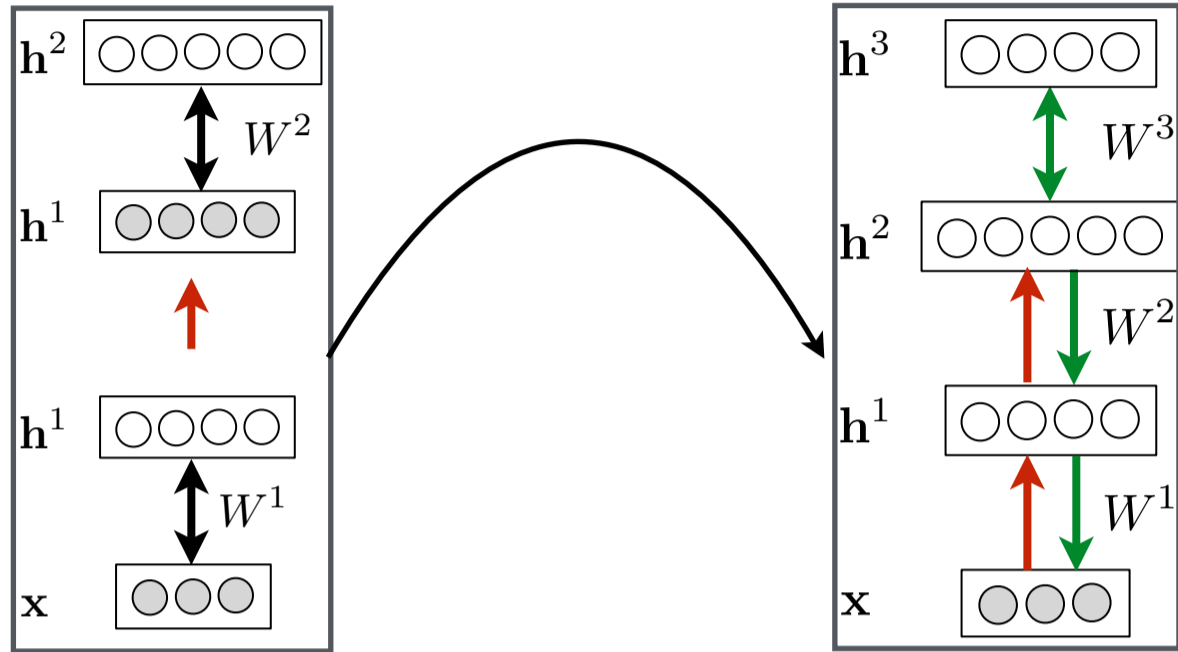


2010-2012  
Most major speech systems  
incorporate DL

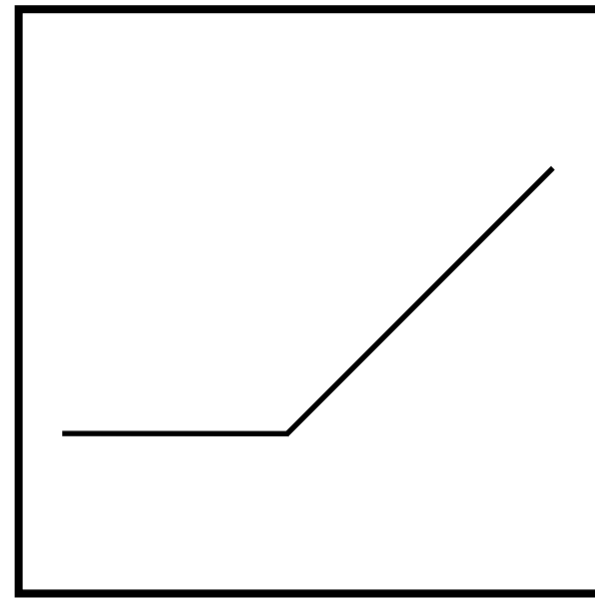


2012  
University of Toronto wins  
ImageNet with convnet

# Deep Learning Timeline



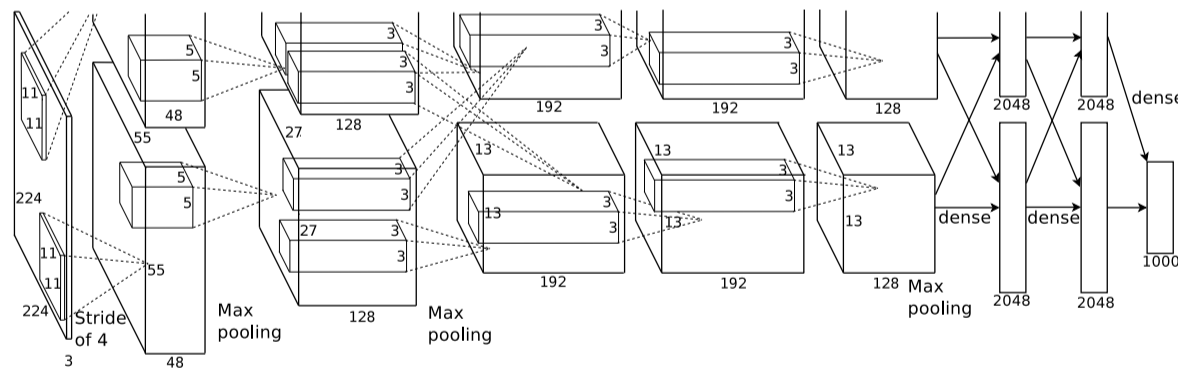
2006  
Pre-training & stacking



2010  
Rectifier units explored



2010-2012  
Most major speech systems  
incorporate DL

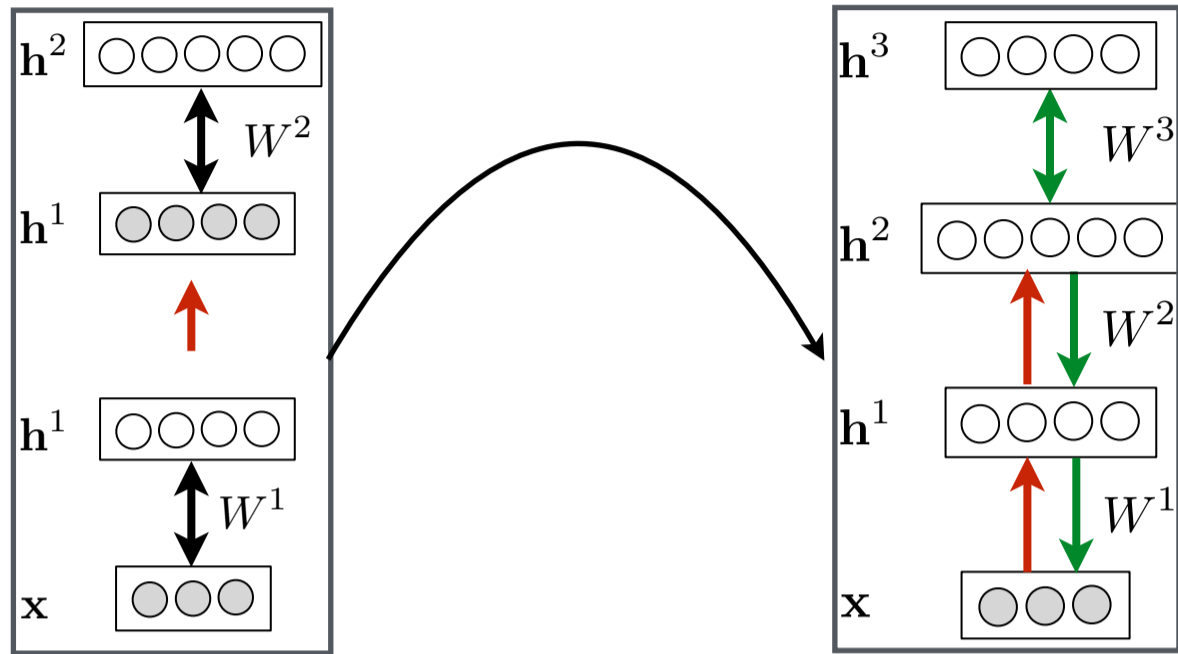


2012  
University of Toronto wins  
ImageNet with convnet

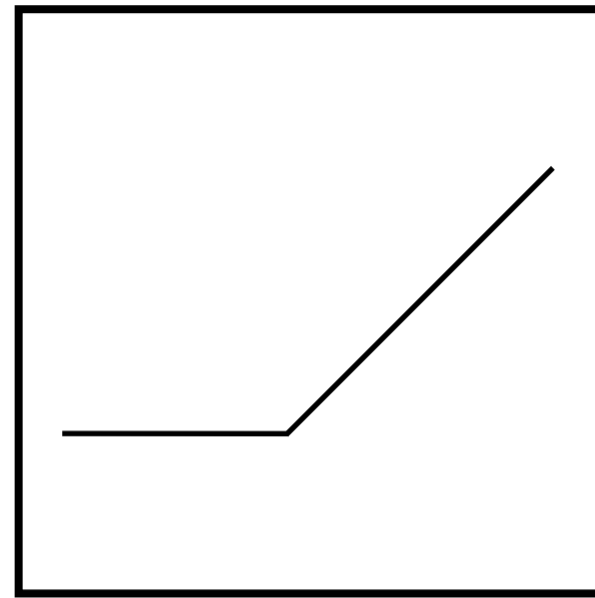


2012-2013  
Google, MS, Facebook, IBM, NEC, Baidu, etc.  
build DL products and accelerate research

# Deep Learning Timeline



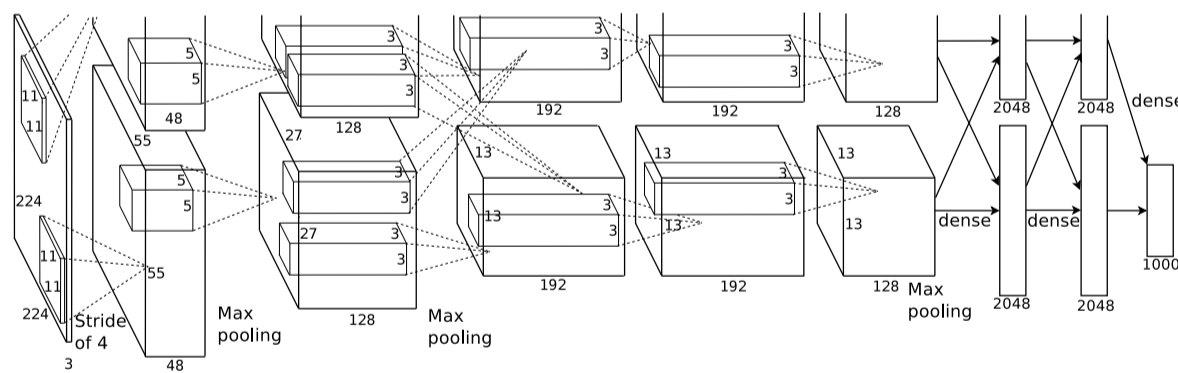
2006  
Pre-training & stacking



2010  
Rectifier units explored



2010-2012  
Most major speech systems  
incorporate DL



2012  
University of Toronto wins  
ImageNet with convnet



2012-2013  
Google, MS, Facebook, IBM, NEC, Baidu, etc.  
build DL products and accelerate research



2014  
DL Mania!  
Specialized hardware avail

# Example: Visual recognition

Traditional recognition typically goes like this...



# Example: Visual recognition

Traditional recognition typically goes like this...

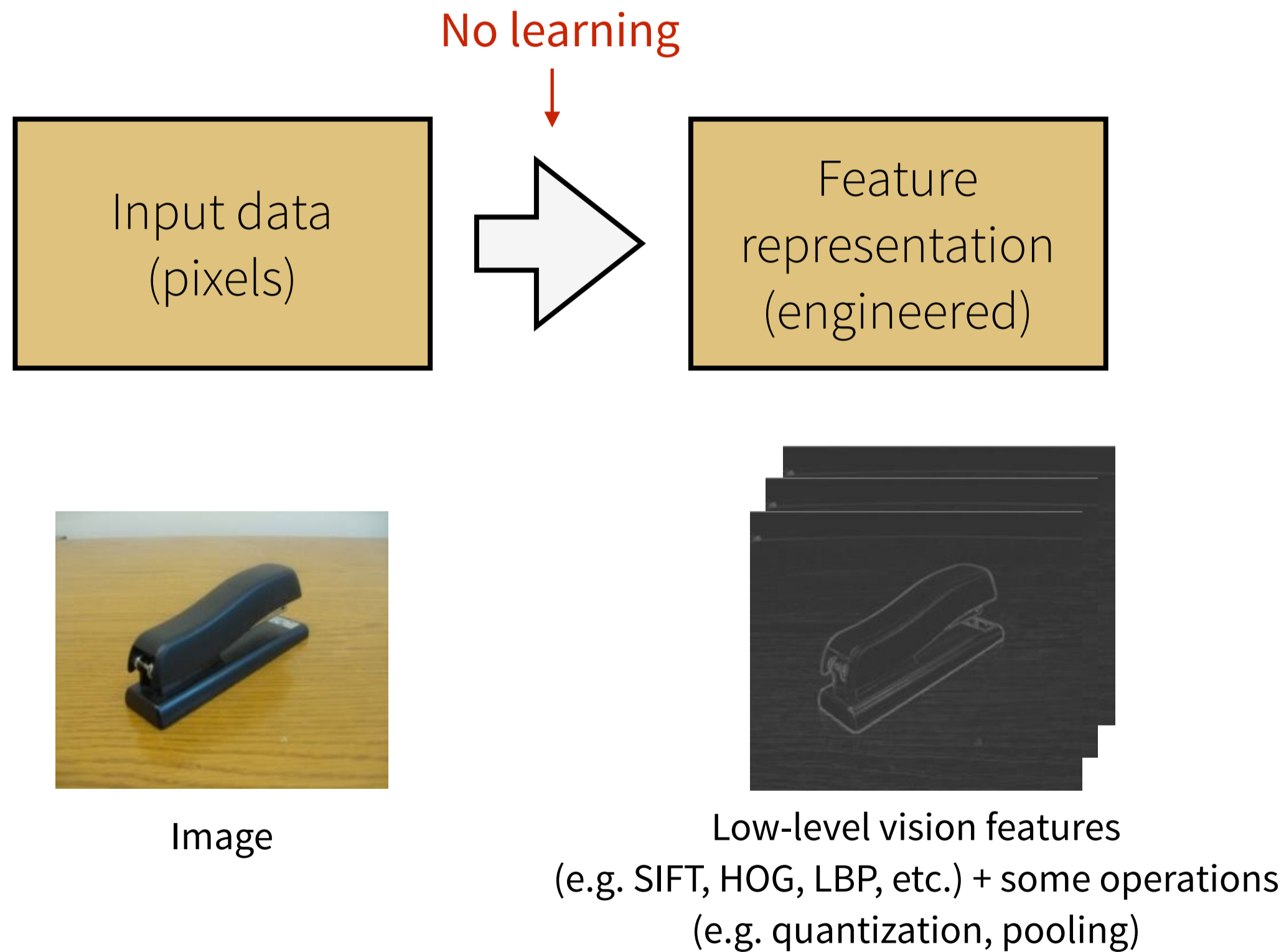
Input data  
(pixels)



Image

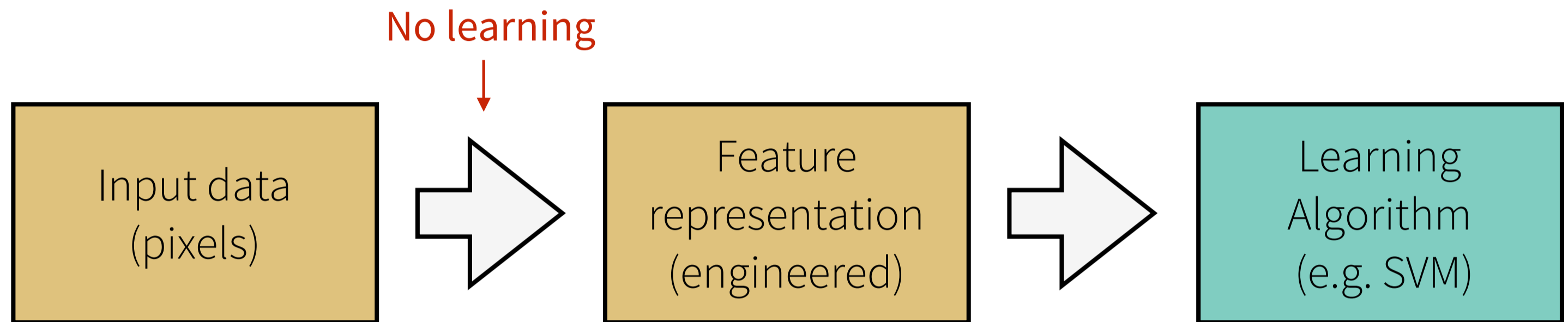
# Example: Visual recognition

Traditional recognition typically goes like this...

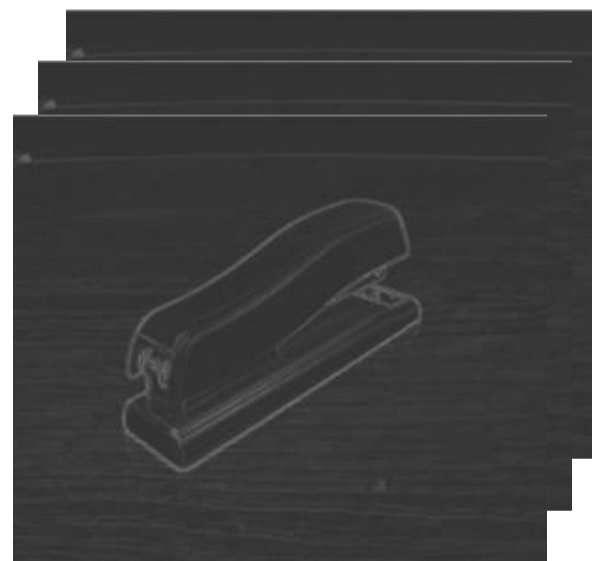


# Example: Visual recognition

Traditional recognition typically goes like this...



Image



Low-level vision features  
(e.g. SIFT, HOG, LBP, etc.) + some operations  
(e.g. quantization, pooling)

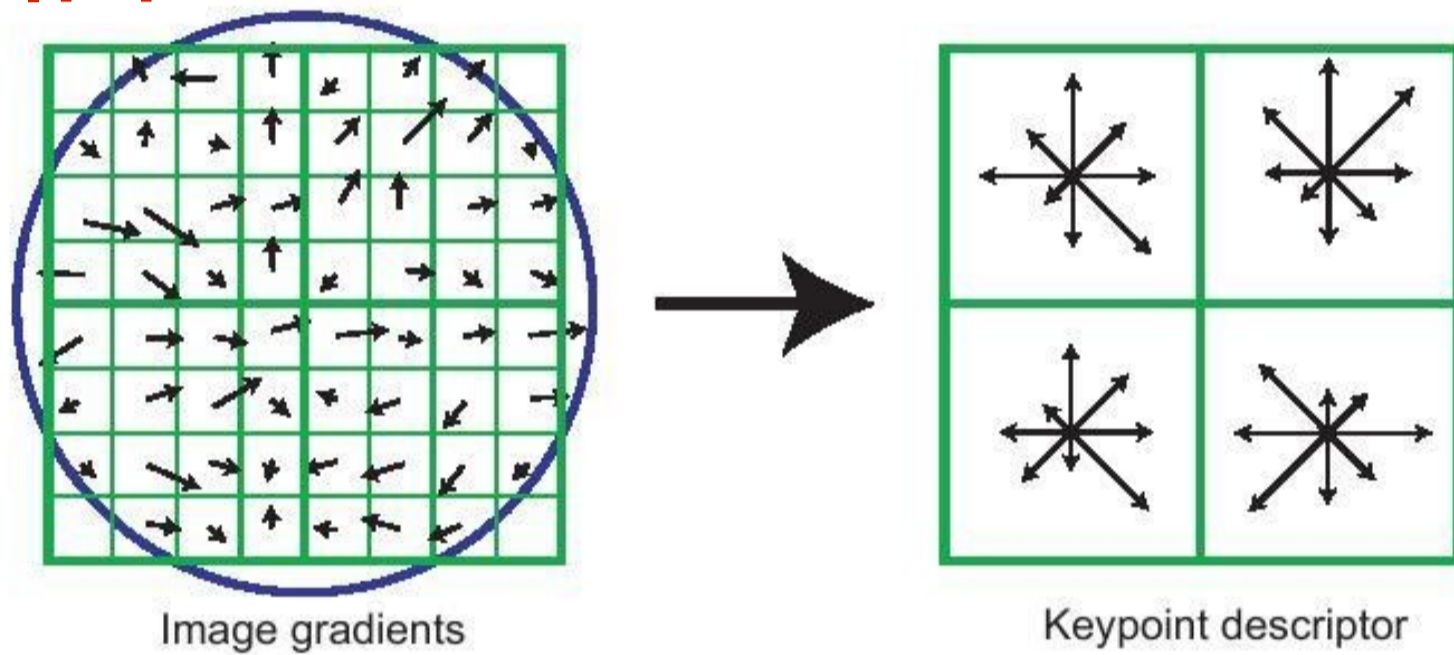


Recognition or detection

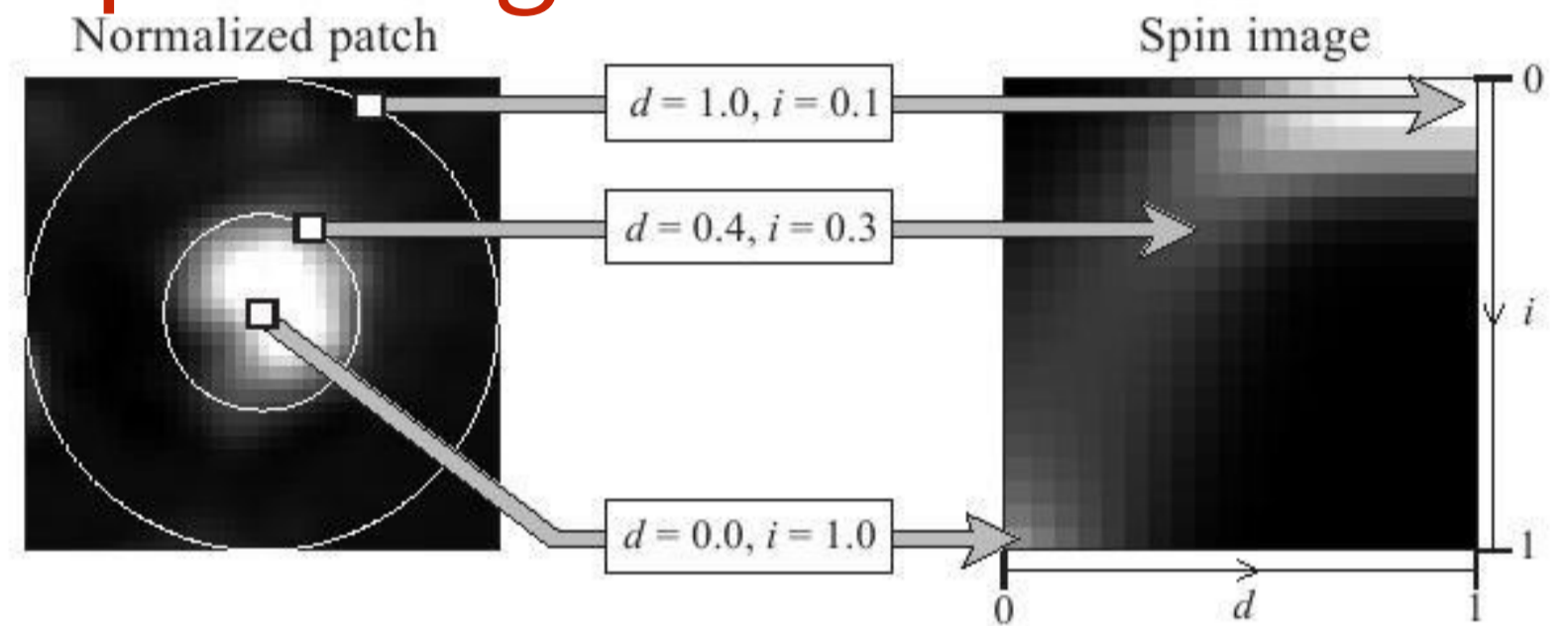
# Feature Engineering

CV community, much effort engineering features...

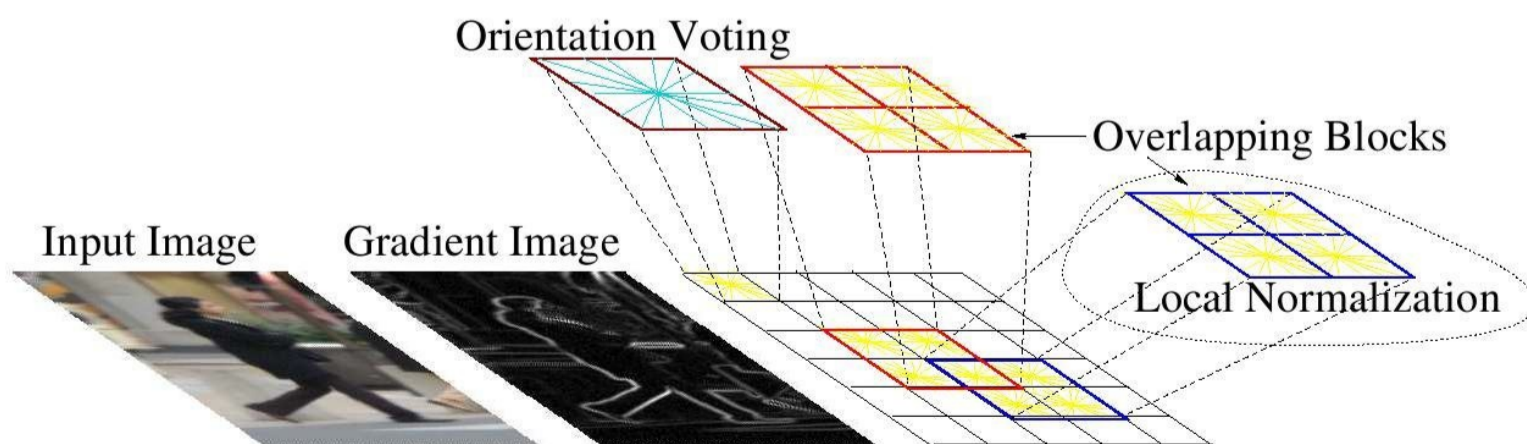
## SIFT



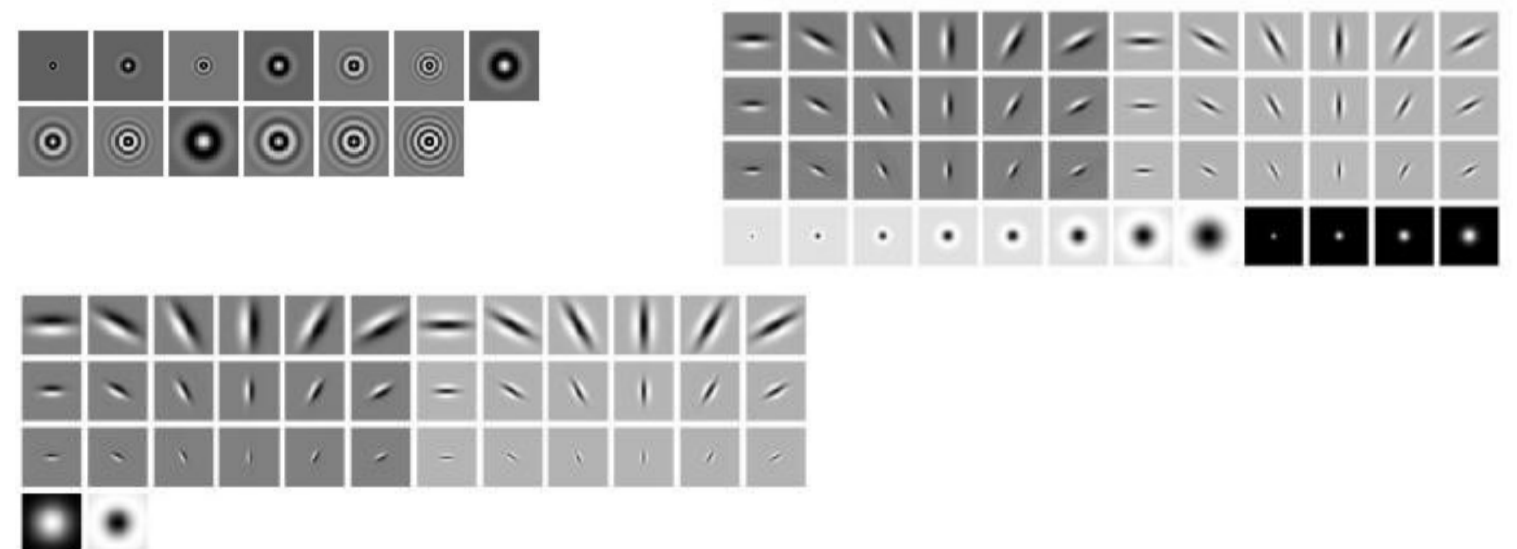
## Spin Image



## HoG

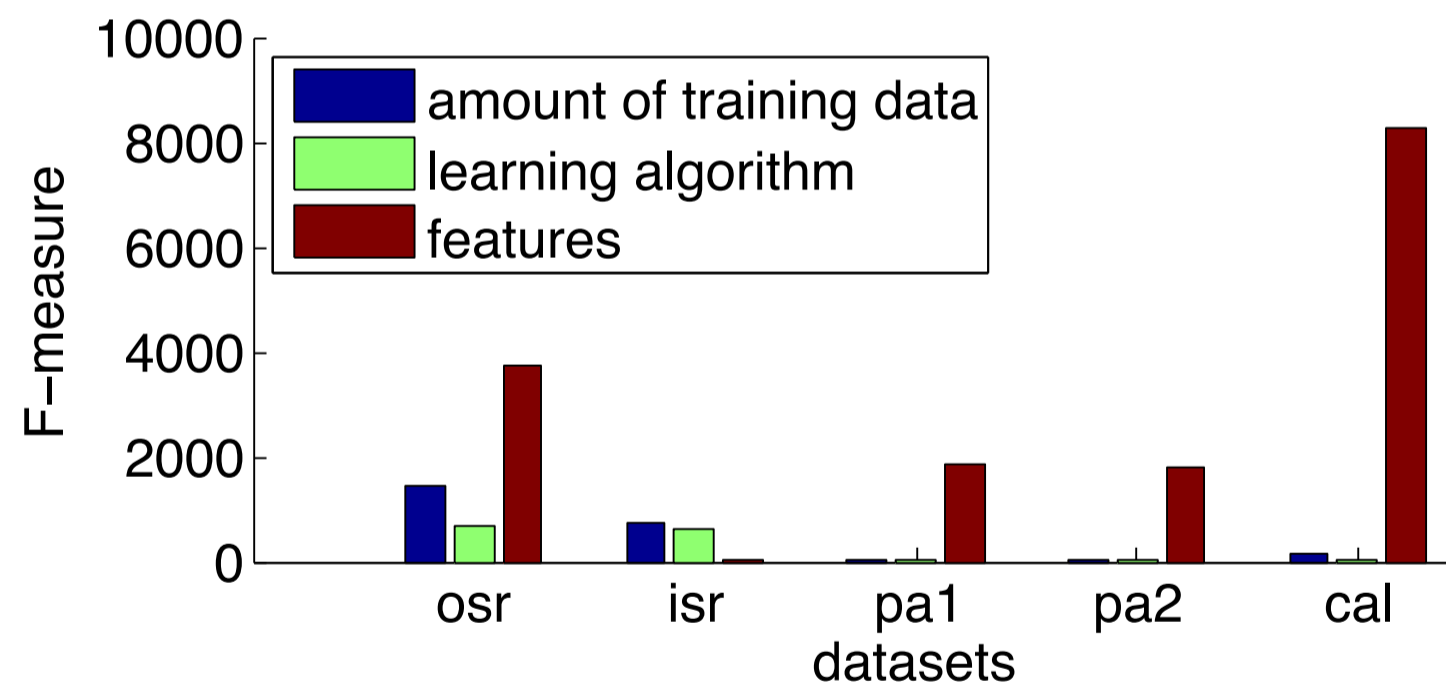


## Textons

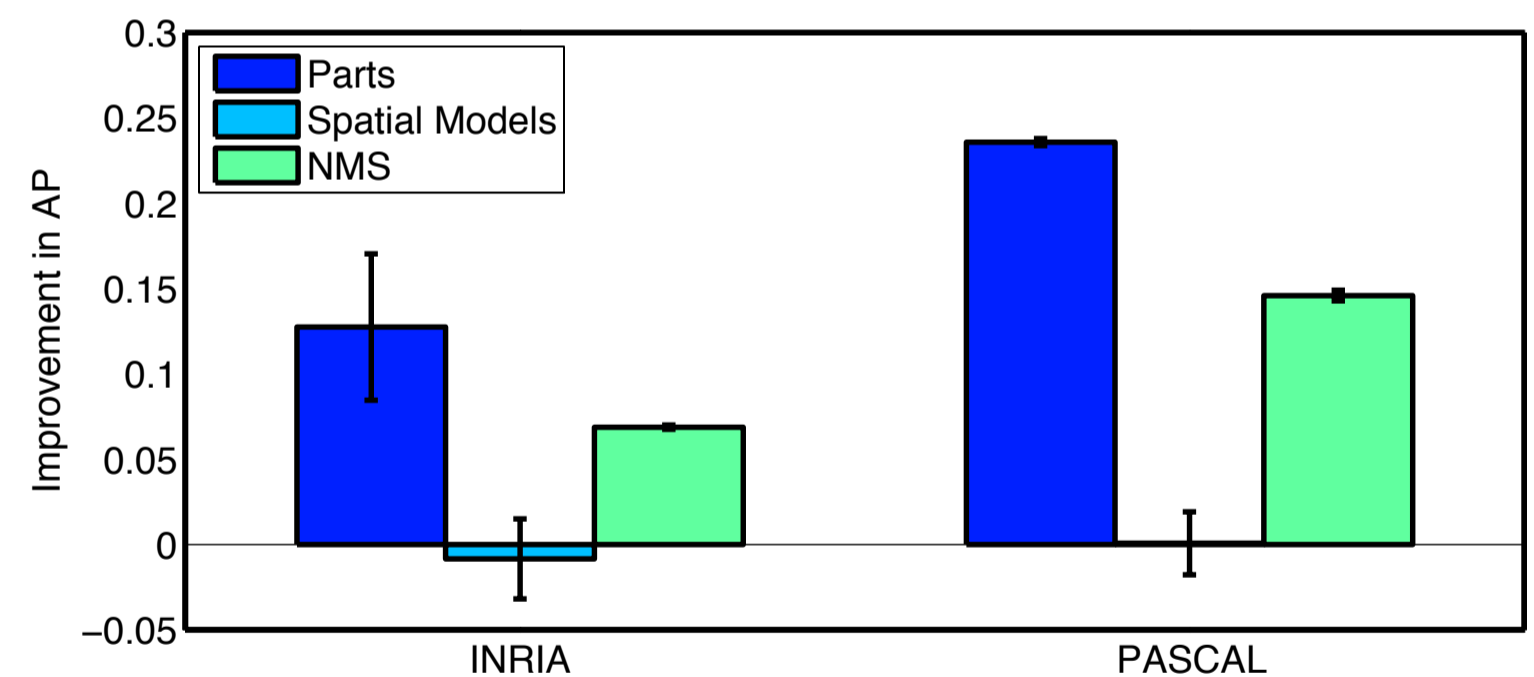


# What Limits Performance?

- Ablation studies on Deformable Parts Model (Felzenszwalb et al. 2010)
- Replace each component with humans (AMT)



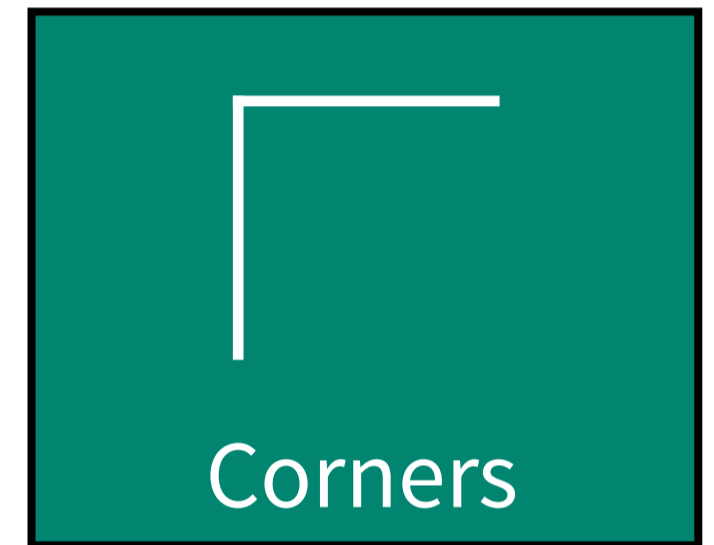
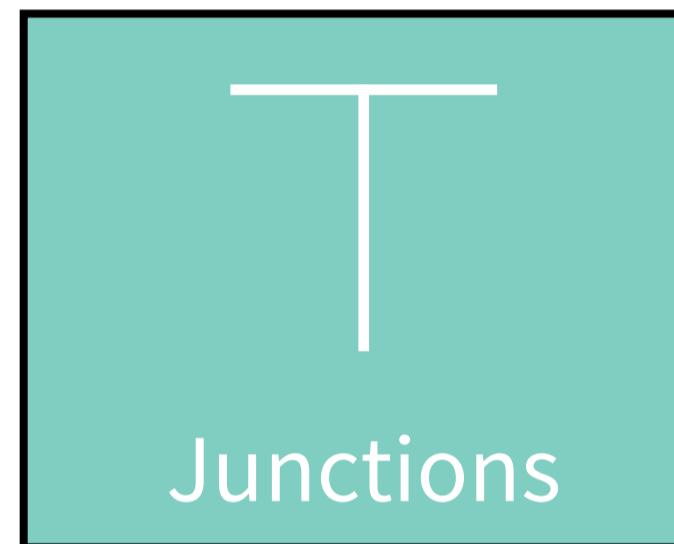
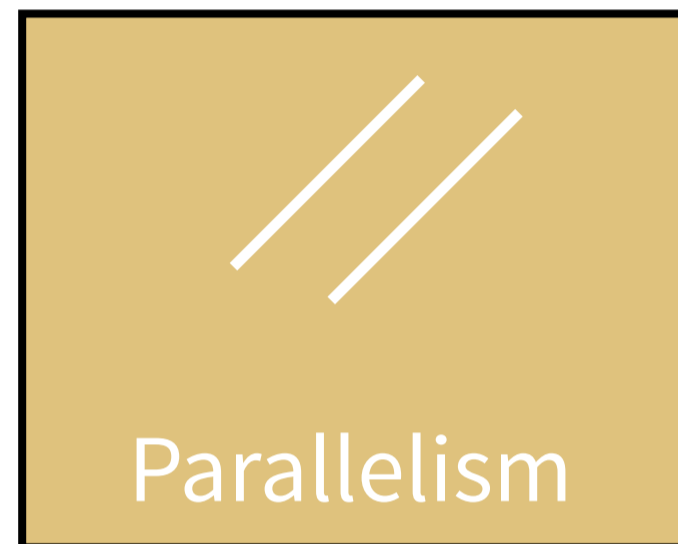
Parikh & Zitnick (CVPR 2010)  
Multiple recognition tasks



Parikh & Zitnick (CVPR 2011)  
Person detectors

# Mid-level Representations

- Mid-level cues



David Marr's "Tokens" (1982)



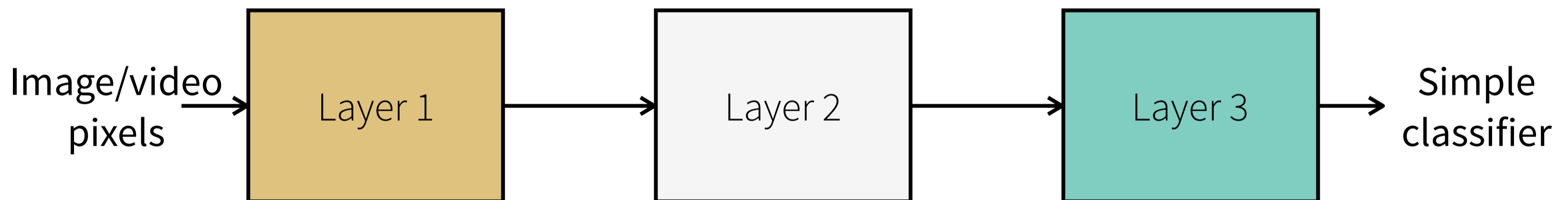
- Object parts



- Difficult to hand-engineer (What about learning them?)

# Learning a Feature Hierarchy

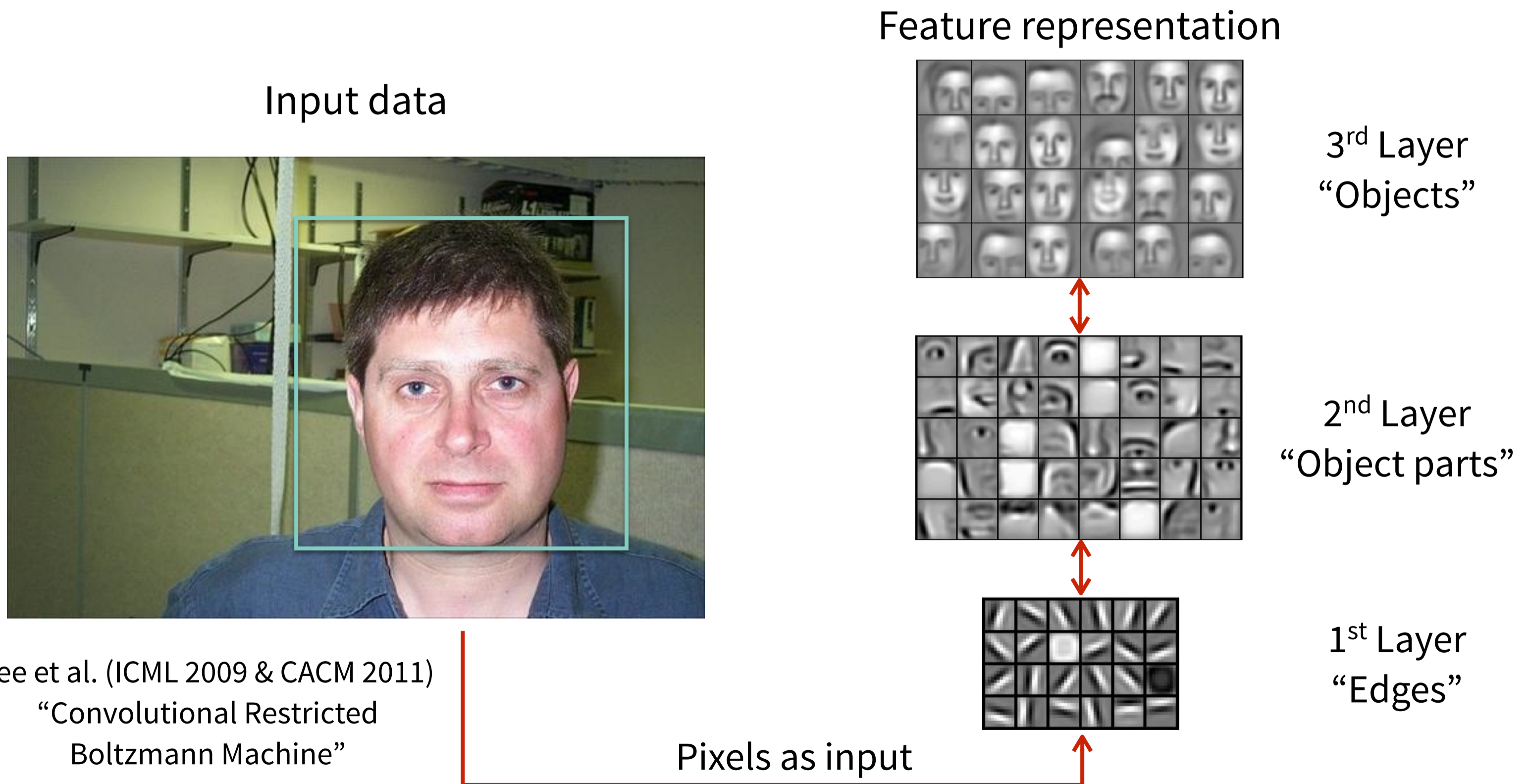
- Learn a hierarchy
- All the way from pixels to output (e.g. classifier)
- Each layer extracts features from output of prev. layer



- Train all layers jointly

# Learning a Feature Hierarchy (2)

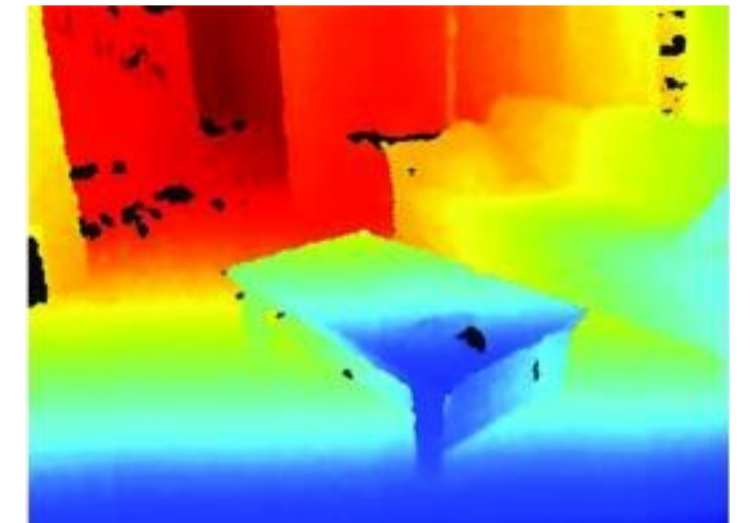
## Learning useful mid-level features from images





# Feature Hierarchies. So what?

- Better performance on discriminative tasks
- Extension to other domains:
  - Kinect (RGB + D)
  - Video
  - Multi-spectral
- Feature computation time
  - Dozens of features now regularly used
  - Prohibitive for large datasets (10's sec/image)



# Feature Learning Paradigms

## Supervised Learning

- End-to-end learning of deep architectures (e.g. deep neural networks) using back-propagation
- Works well when there is a lot of labeled data
- Structure of the model is important (e.g. convnet)

## Unsupervised Learning

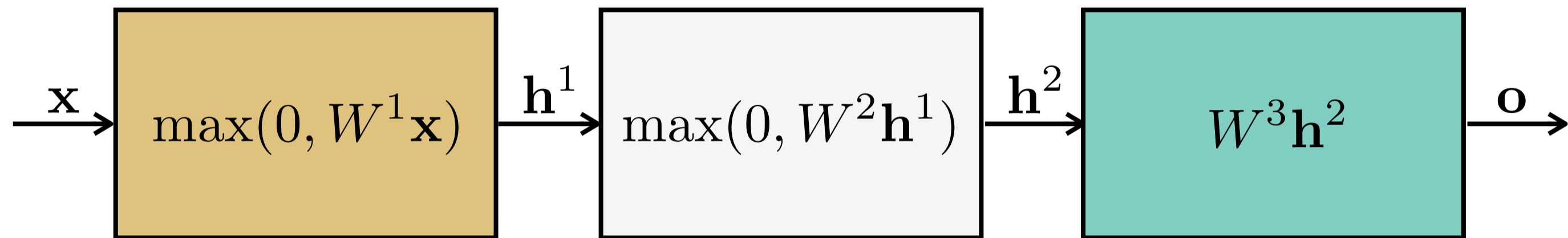
- Learn statistical structure or dependencies in the data from unlabelled data
- Usually a layer-by-layer training strategy is employed
- Useful when there is little or no labeled data

# Neural Networks (Introduction)

- Approximate a complicated function by a **composition of simpler functions**
  - Each simple function will have parameters subject to training
  - The composition is a **highly non-linear** function
- Assume the input is a vector
  - For images, this means we ignore spatial layout of the pixels

# Neural Networks for Supervised Learning

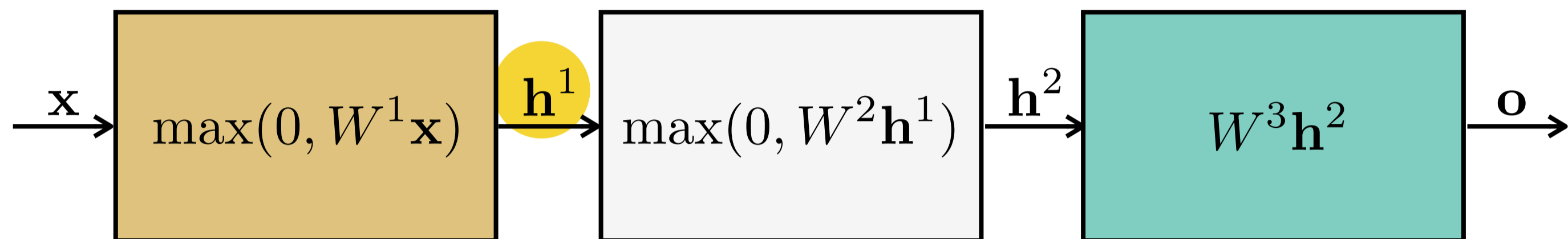
Example of a 2 hidden layer (or 4 layer) network ...



$x$	input
$h^1$	1 <sup>st</sup> layer hidden units
$h^2$	2 <sup>nd</sup> layer hidden units
$o$	output (prediction)

# Forward Propagation

Forward propagation is the process of computing the output of the network given its input



$$\mathbf{x} \in \mathbb{R}^D$$

$$W^1 \in \mathbb{R}^{N_1 \times D}$$

$$\mathbf{b}^1 \in \mathbb{R}^{N_1}$$

$$\mathbf{h}^1 \in \mathbb{R}^{N_1}$$

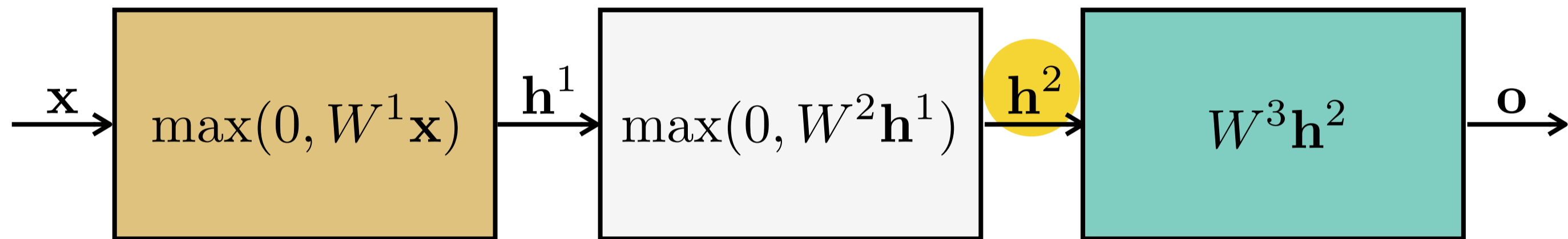
$$\mathbf{h}^1 = \max(0, W^1 \mathbf{x} + \mathbf{b}^1)$$

$W^1$       1<sup>st</sup> layer weights

$\mathbf{b}^1$       1<sup>st</sup> layer biases

The non-linearity  $u = \max(0, v)$  is called a **ReLU** in the DL literature. Each output hidden unit takes as input all the units at the previous layer: each such layer is called “**fully connected**”.

# Forward Propagation



$$\mathbf{h}^1 \in \mathbb{R}^{N_1}$$

$$W^2 \in \mathbb{R}^{N_2 \times N_1}$$

$$\mathbf{b}^2 \in \mathbb{R}^{N_2}$$

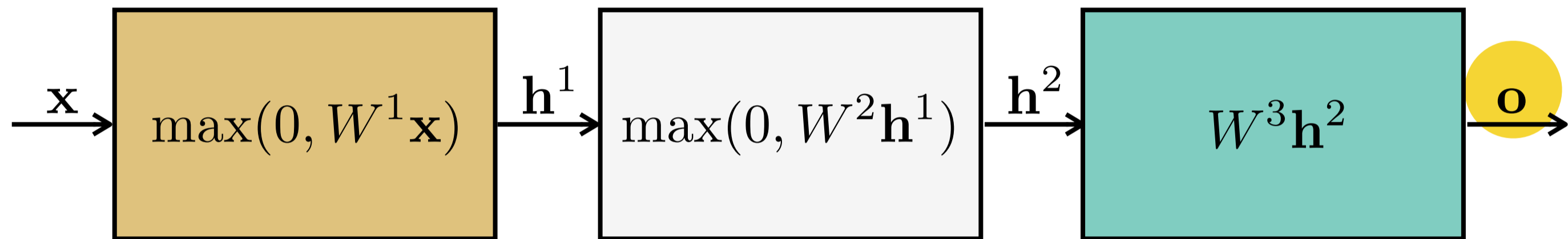
$$\mathbf{h}^2 \in \mathbb{R}^{N_2}$$

$$\mathbf{h}^2 = \max(0, W^2 \mathbf{h}^1 + \mathbf{b}^2)$$

$W^2$       2<sup>nd</sup> layer weights

$\mathbf{b}^2$       2<sup>nd</sup> layer biases

# Forward Propagation



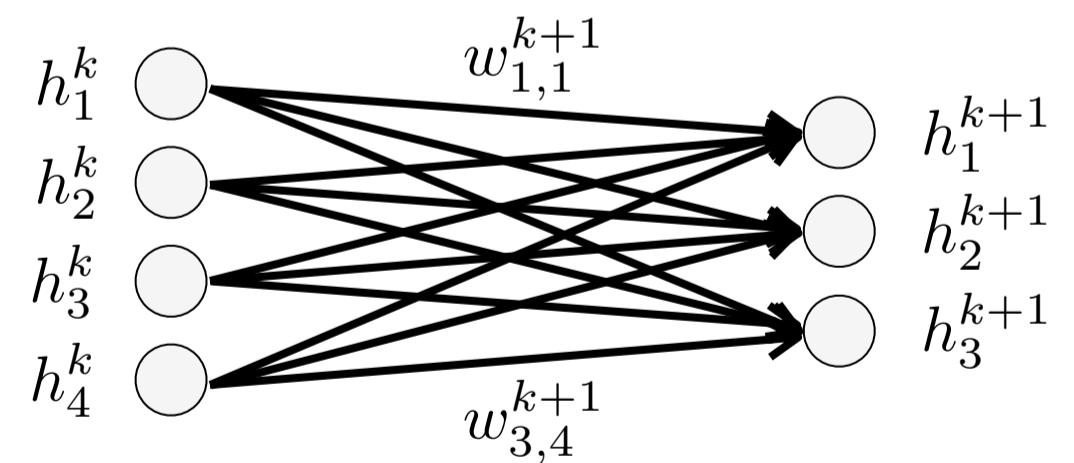
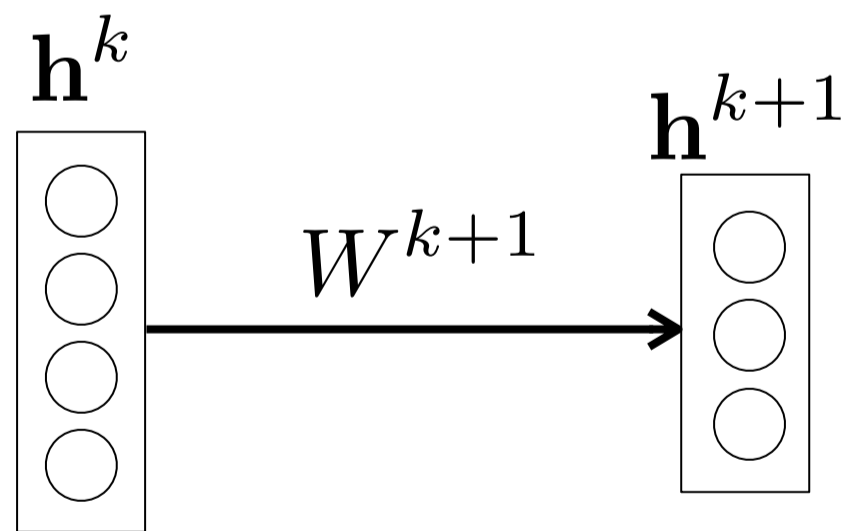
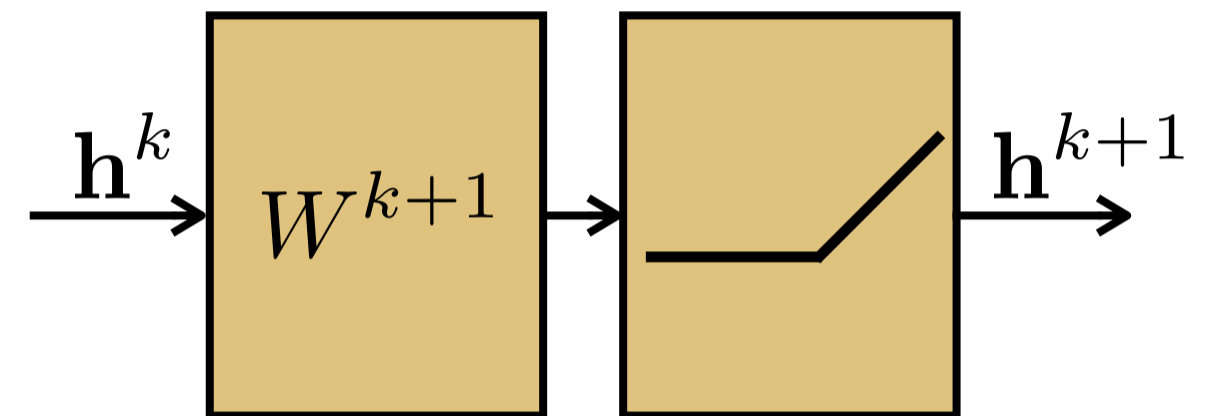
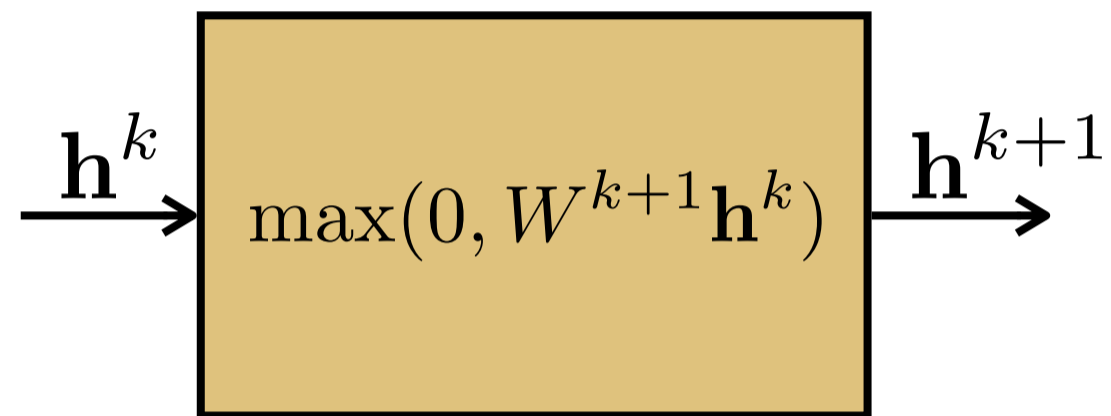
$$\mathbf{h}^2 \in \mathbb{R}^{N_2} \quad W^3 \in \mathbb{R}^{N_3 \times N_2} \quad \mathbf{b}^3 \in \mathbb{R}^{N_3} \quad \mathbf{o} \in \mathbb{R}^{N_3}$$

$$\mathbf{o} = \max(0, W^3 \mathbf{h}^2 + \mathbf{b}^3)$$

$W^3$       3<sup>rd</sup> layer weights

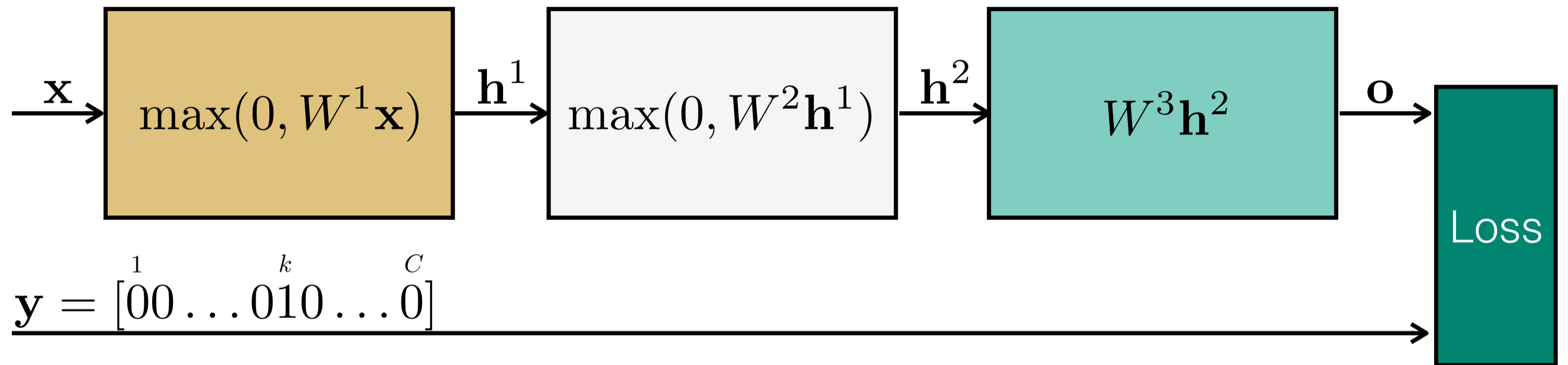
$\mathbf{b}^3$       3<sup>rd</sup> layer biases

# Alternative Graphical Representations





# How Good is a Network?



Probability of class  $k$  given input (softmax):

$$p(c_k = 1 | \mathbf{x}) = \frac{e^{\mathbf{o}_k}}{\sum_{j=1}^C e^{\mathbf{o}_j}}$$

(Per-sample) **Loss**; e.g. negative log-likelihood  
(good for classification of a small number of classes):

$$L(\mathbf{x}, \mathbf{y}; \theta) = - \sum_j y_j \log p(c_j | \mathbf{x})$$

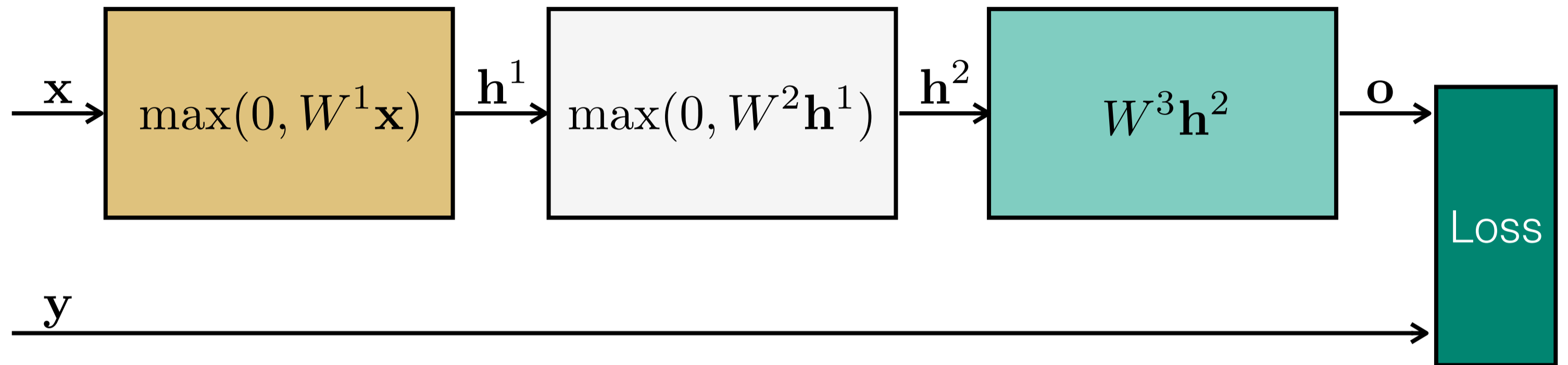
# Training

- Learning consists of minimizing the loss (plus some regularization term) w.r.t. parameters over the whole training set

$$\theta^* = \arg \min_{\theta} \sum_{n=1}^P L(\mathbf{x}^n, \mathbf{y}^n; \theta)$$

- **Question:** How to minimize a complicated function of the parameters?
- **Answer:** Chain rule, a.k.a. **Backpropagation!** This is the procedure to compute gradients of the loss w.r.t. parameters in a multi-layer neural network.

# Learning by Perturbing Weights



Let's say we want to decrease the loss by adjusting  $W_{i,j}^1$ . We could consider a very small  $\epsilon = 1e^{-6}$  and compute:

$$L(\mathbf{x}, \mathbf{y}; \theta)$$

$$L(\mathbf{x}, \mathbf{y}; \theta \setminus W_{i,j}^1, W_{i,j}^1 + \epsilon)$$

Then update:

$$W_{i,j}^1 \leftarrow W_{i,j}^1 + \epsilon \operatorname{sgn} (L(\mathbf{x}, \mathbf{y}; \theta) - L(\mathbf{x}, \mathbf{y}; \theta \setminus W_{i,j}^1, W_{i,j}^1 + \epsilon))$$

# The Idea behind Backpropagation

- A better idea: randomly perturb the **activities** of the hidden units
- We don't know what the hidden units ought to do, but we can compute how fast the error changes when we change a hidden activity
  - instead of using desired activities to train hidden units, use error derivatives w.r.t. activities
  - each hidden activity can affect many output units and therefore have many separate effects on error (sum them)
- We can compute error derivatives for all hidden units efficiently at the same time
  - Once we have the error derivatives for the hidden activities, it's easy to get the error derivatives for the weights going into a hidden unit

# Derivative w.r.t. Input of Softmax

First convert the discrepancy between each output and its target value into an error derivative

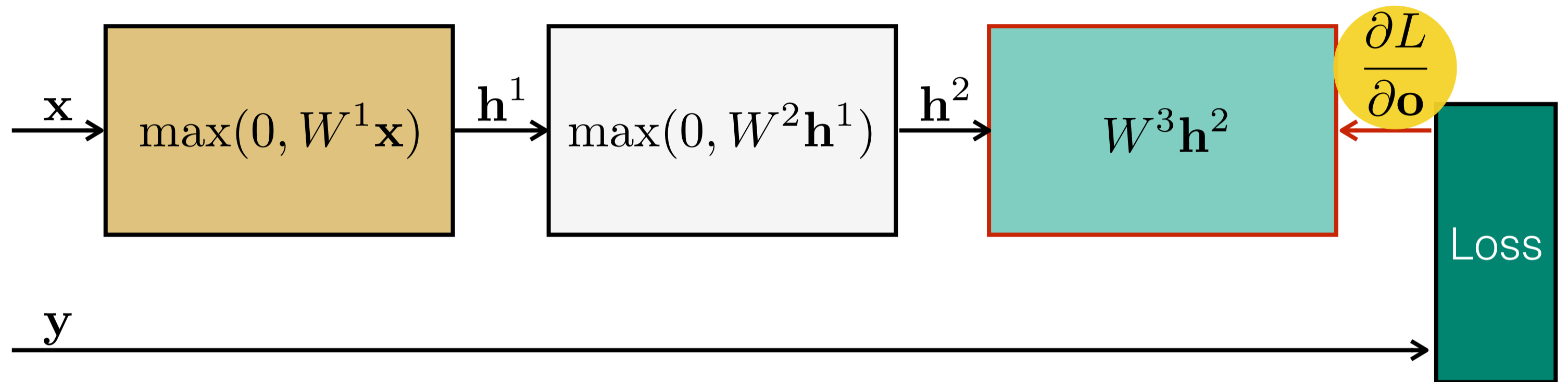
$$p(c_k = 1 | \mathbf{x}) = \frac{e^{o_k}}{\sum_{j=1}^C e^{o_j}}$$

$$L(\mathbf{x}, \mathbf{y}; \theta) = - \sum_j y_j \log p(c_j | \mathbf{x}) \quad \mathbf{y} = [00 \dots 010 \dots 0]$$

By substituting the first formula into the second, and taking the derivative w.r.t.  $o$  we get:

$$\frac{\partial L}{\partial o_j} = p(c_j | \mathbf{x}) - y_j$$

# Backward Propagation



Given  $\frac{\partial L}{\partial \mathbf{o}}$  and assuming we can easily compute the Jacobian of each module, we have:

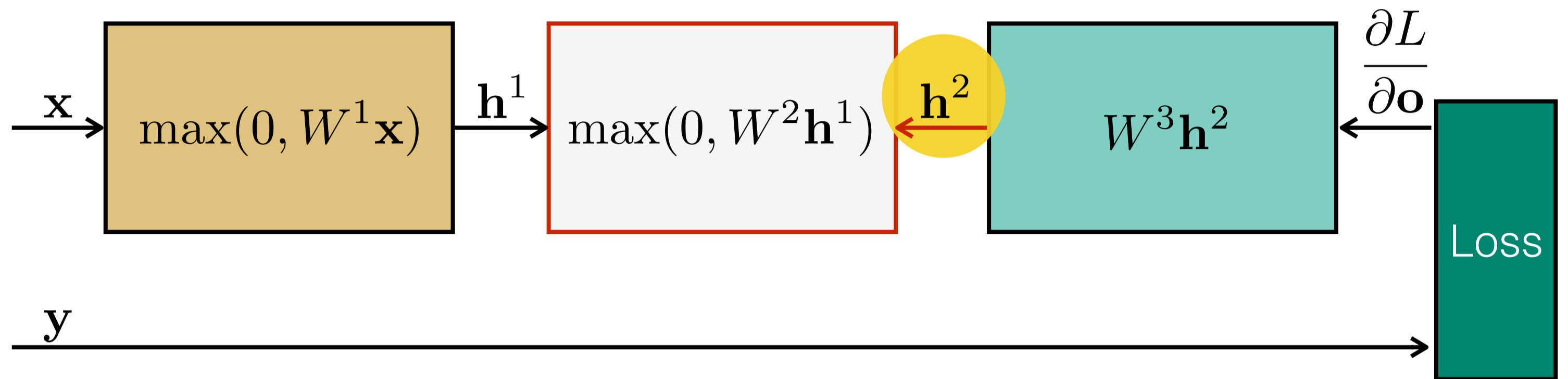
$$\frac{\partial L}{\partial W^3} = \frac{\partial L}{\partial \mathbf{o}} \frac{\partial \mathbf{o}}{\partial W^3}$$

$$= (p(\mathbf{c}|\mathbf{x}) - \mathbf{y}) \mathbf{h}^{2T}$$

$$\frac{\partial L}{\partial \mathbf{h}^2} = \frac{\partial L}{\partial \mathbf{o}} \frac{\partial \mathbf{o}}{\partial \mathbf{h}^2}$$

$$= W^{3T} (p(\mathbf{c}|\mathbf{x}) - \mathbf{y})$$

# Backward Propagation

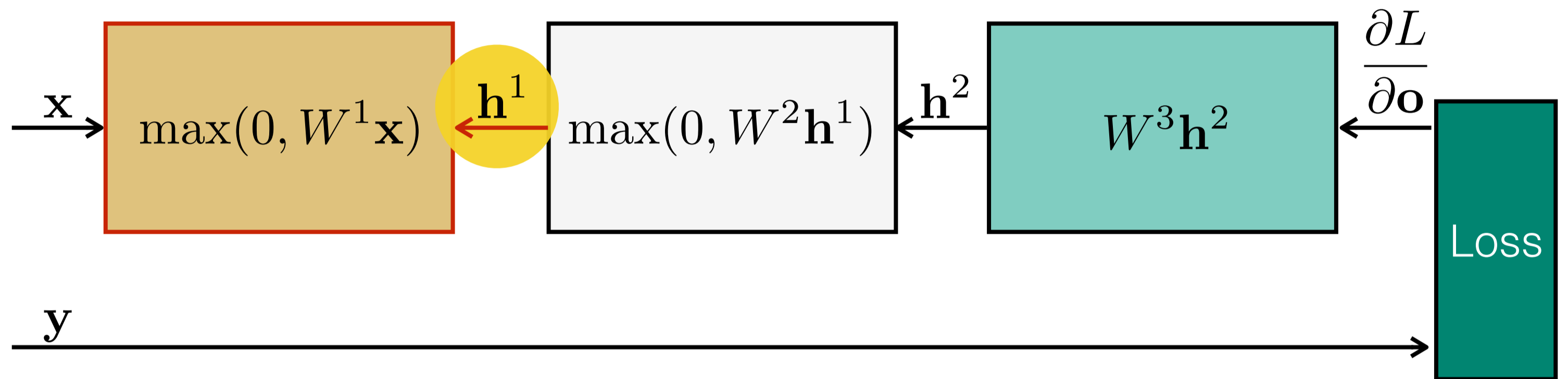


Given  $\frac{\partial L}{\partial \mathbf{h}^2}$  we can compute now:

$$\frac{\partial L}{\partial W^2} = \frac{\partial L}{\partial \mathbf{h}^2} \frac{\partial \mathbf{h}^2}{\partial W^2}$$

$$\frac{\partial L}{\partial \mathbf{h}^1} = \frac{\partial L}{\partial \mathbf{h}^2} \frac{\partial \mathbf{h}^2}{\partial \mathbf{h}^1}$$

# Backward Propagation



Given  $\frac{\partial L}{\partial \mathbf{h}^1}$  we can compute now:

$$\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial \mathbf{h}^1} \frac{\partial \mathbf{h}^1}{\partial W^1}$$



# Technical Challenge: Composition

- Neural networks are modular architectures
  - Often use repeated blocks
- Forward pass and backward pass must be defined for each module

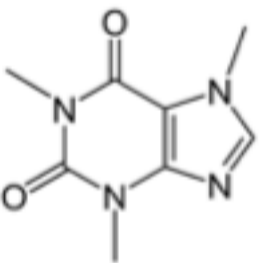
# Tools for Building Neural Networks

- Modern scientific computing tools exploit modularity



- **Torch7** (LuaJIT + C)

- <http://torch.ch/>



- **Caffe** (C++ w/ Python & Matlab wrappers)

- <http://caffe.berkeleyvision.org/>

- **Theano/Pylearn2** (Python) — See Pascal Lamblin's talk!

- [http://deeplearning.net/software\\_links/](http://deeplearning.net/software_links/)

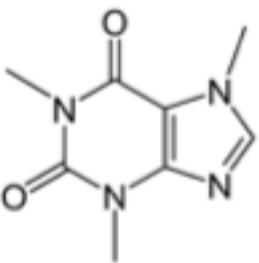
# Tools for Building Neural Networks

- Modern scientific computing tools exploit modularity



- **Torch7** (LuaJIT + C)

- <http://torch.ch/>



- **Caffe** (C++ w/ Python & Matlab wrappers)

- <http://caffe.berkeleyvision.org/>

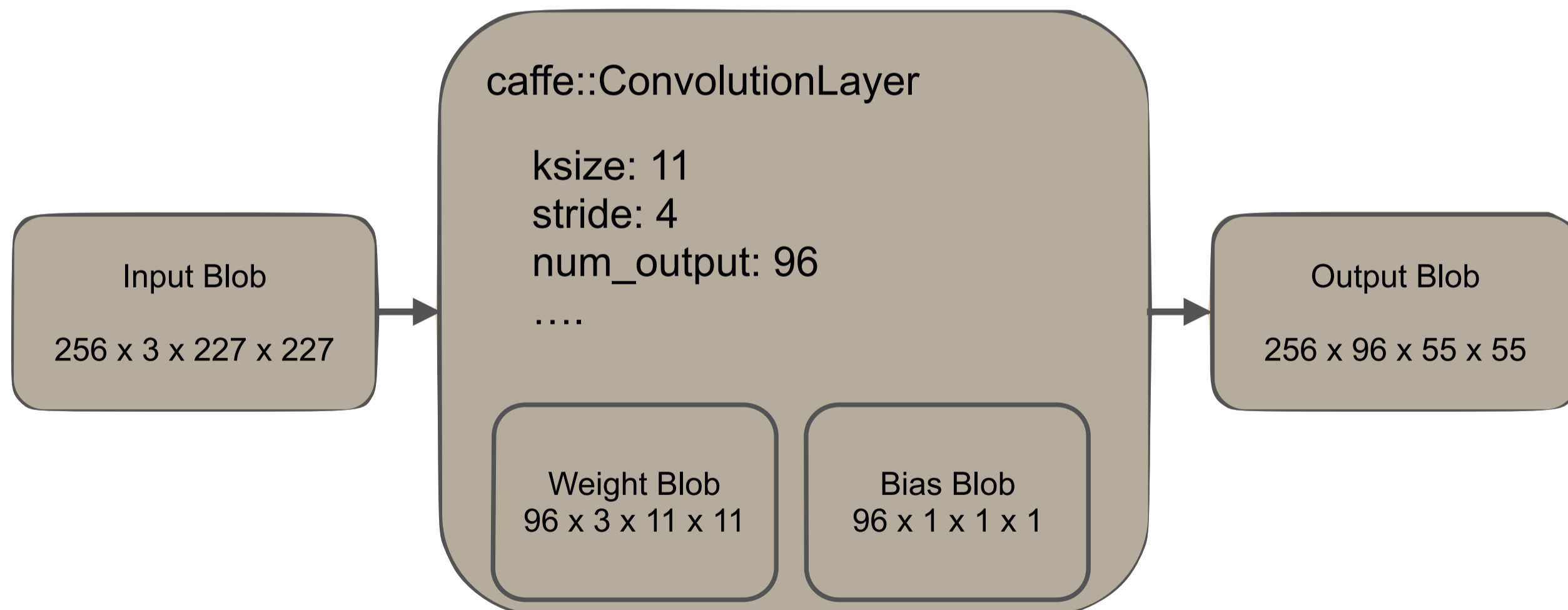
- **Theano/Pylearn2** (Python) — See Pascal Lamblin's talk!

- [http://deeplearning.net/software\\_links/](http://deeplearning.net/software_links/)

See (<https://sites.google.com/site/deeplearningcvpr2014/>)  
for a short presentation of each.

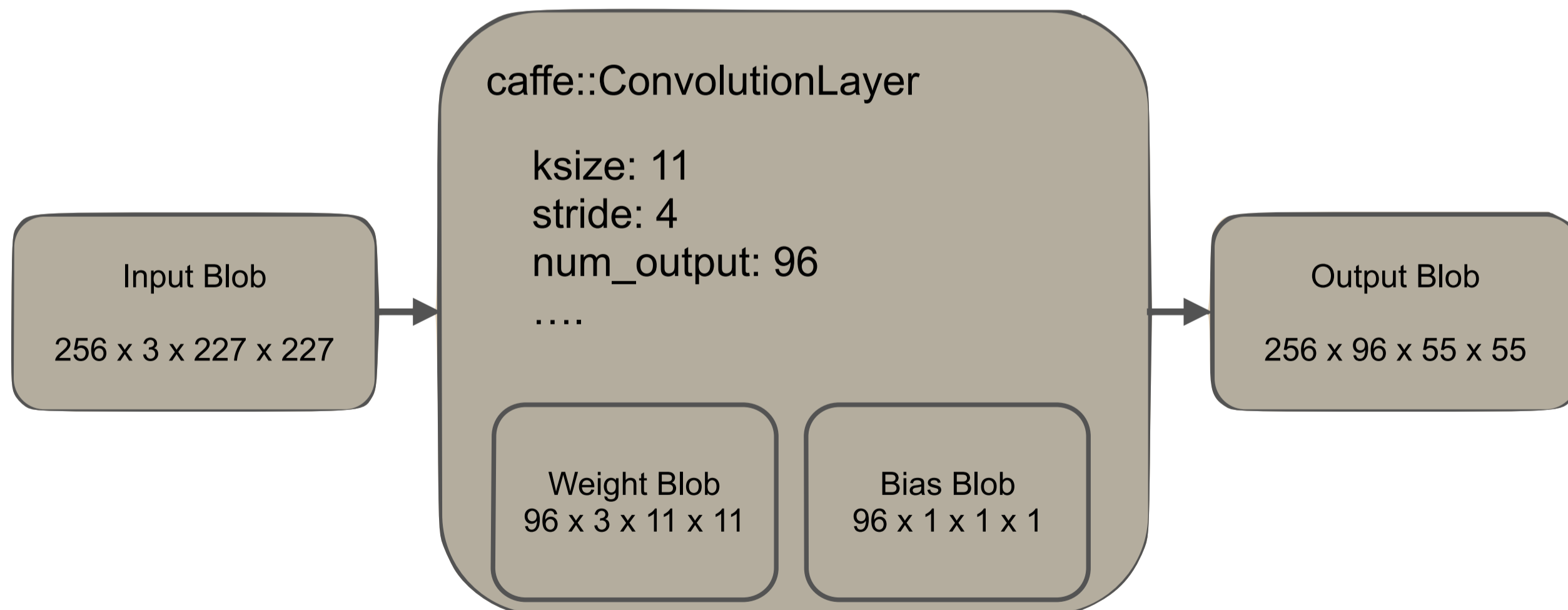
# Caffe Example

Caffe nets are composed of layers as defined in a model schema



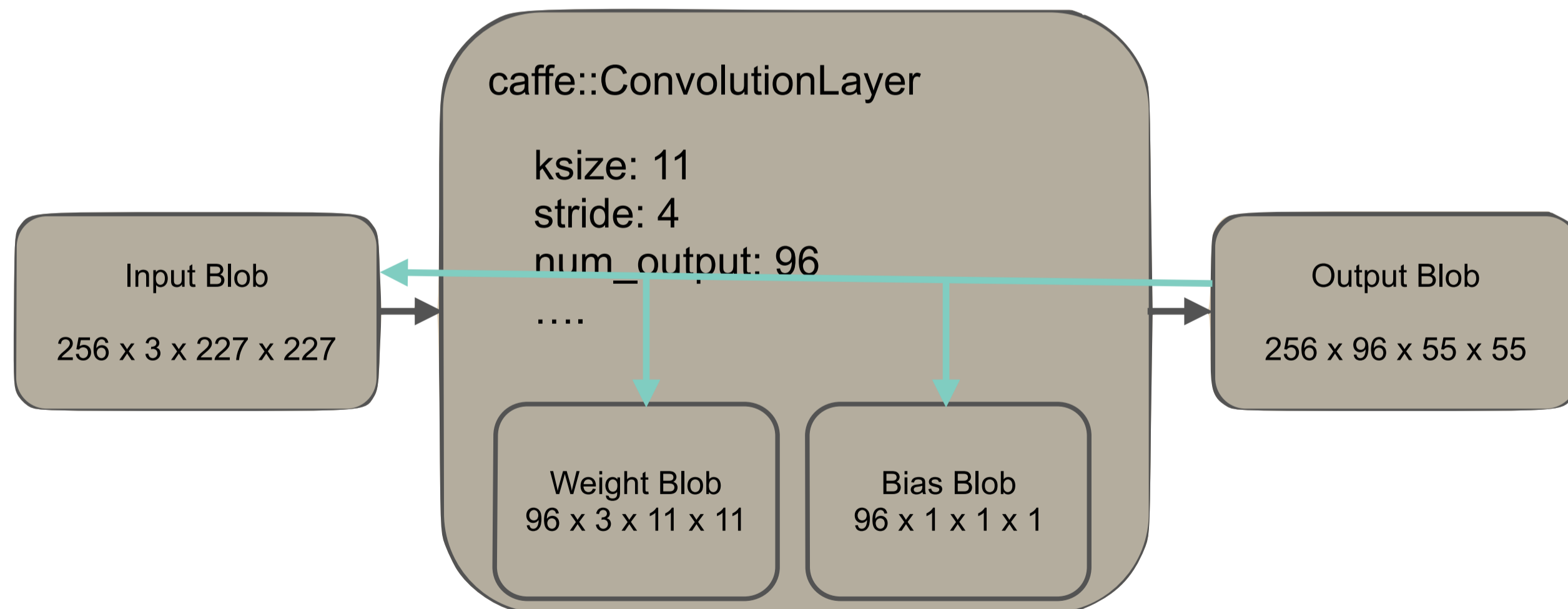
# Caffe: Each layer defines...

- Forward: given input, compute the output →
- Backward: given the gradient w.r.t. the output, compute the gradient w.r.t. the input and its internal parameters →
- Setup: how to initialize the layer



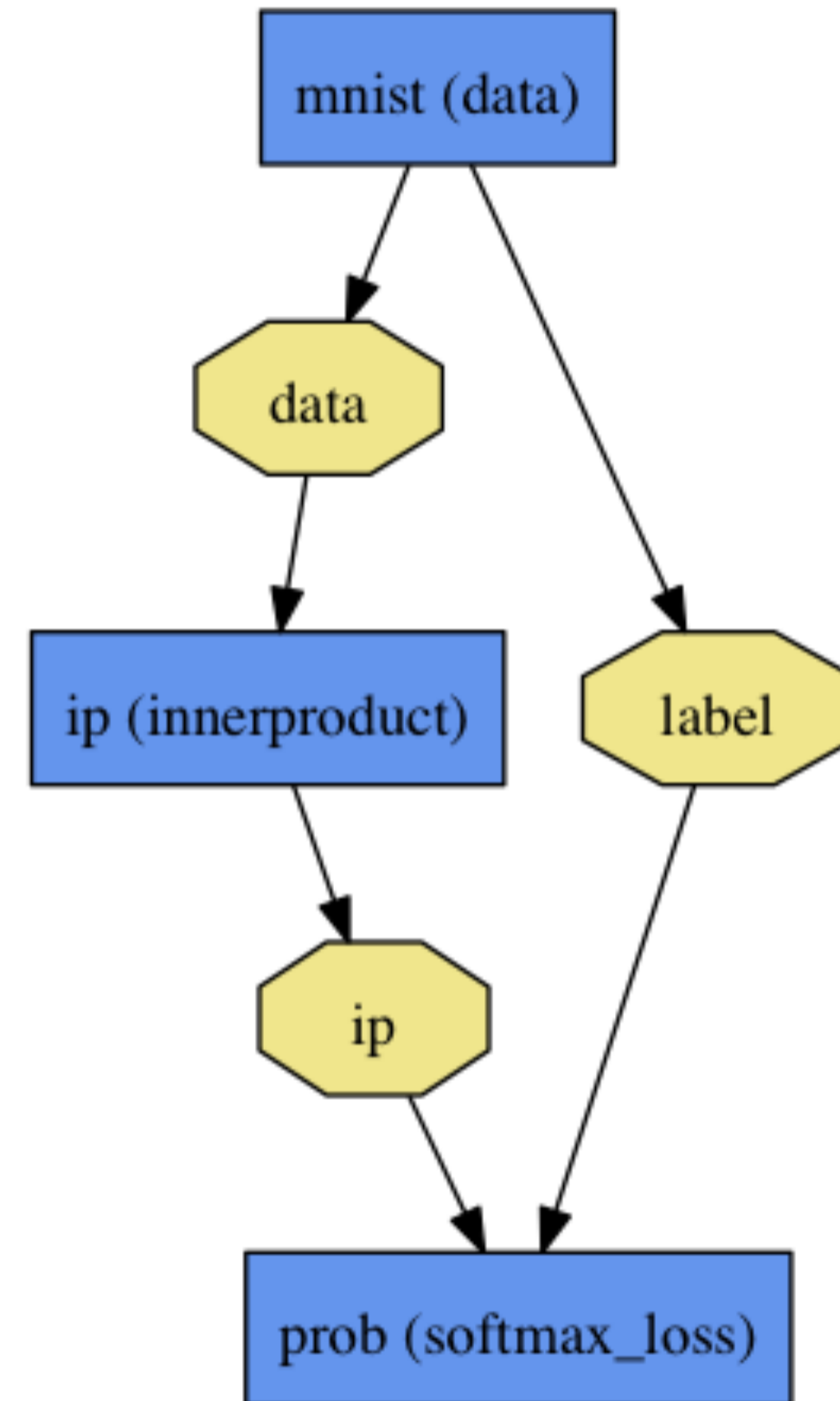
# Caffe: Each layer defines...

- Forward: given input, compute the output →
- Backward: given the gradient w.r.t. the output, compute the gradient w.r.t. the input and its internal parameters →
- Setup: how to initialize the layer



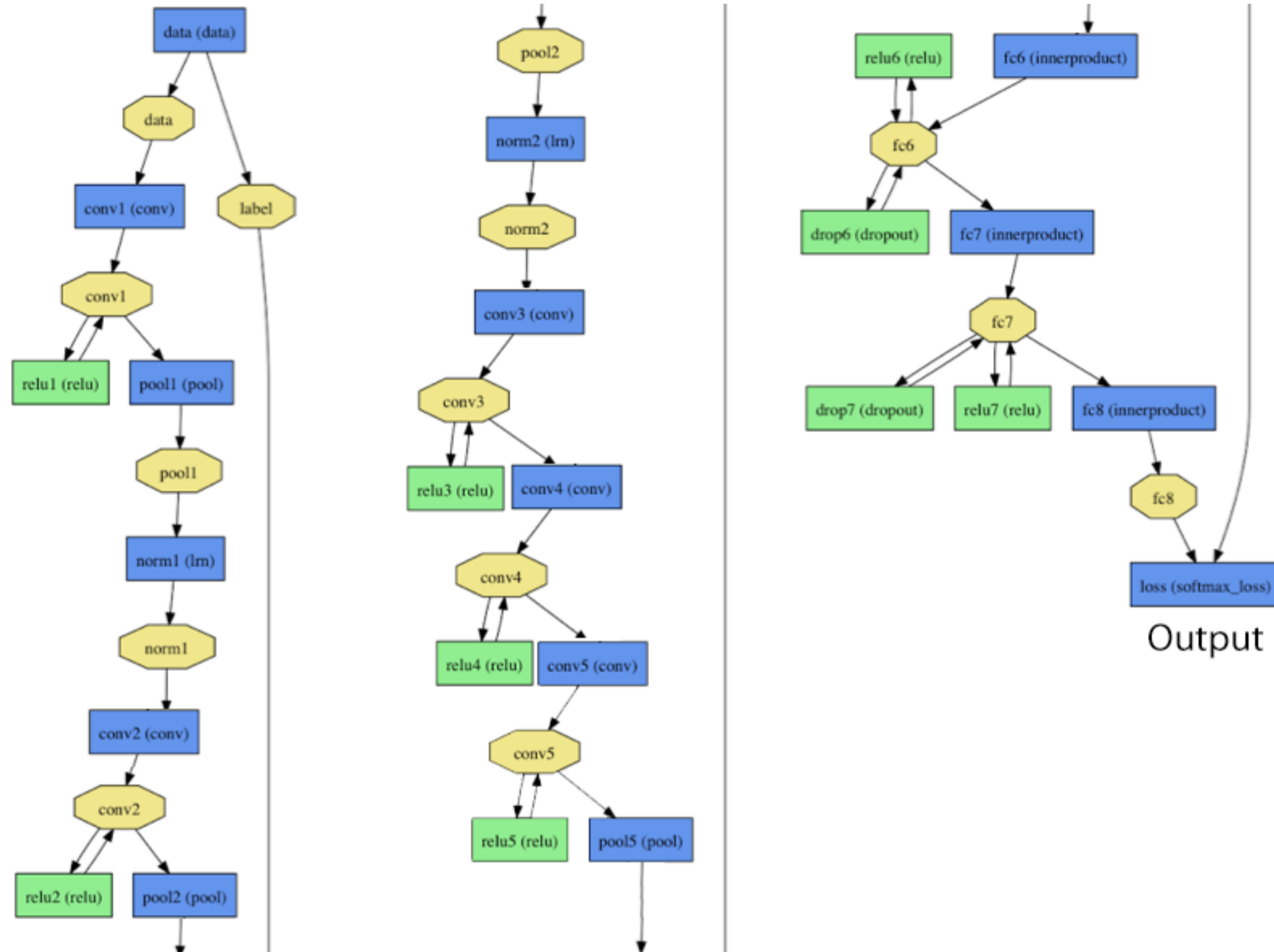
# Caffe: Definition of a Net

```
name: "mnist-small"
# data layer for input
layers {
  layer {
    name: "mnist"
    type: "data"
    source: "data/mnist-train-leveldb"
    batchsize: 64
    scale: 0.00390625
  }
  top: "data"
  top: "label"
}
# linear classifier by inner product
layers {
  layer {
    name: "ip"
    type: "innerproduct" num_output: 10 weight_filler {
      type: "xavier" }
  }
  bottom: "data"
  top: "ip"
}
# softmax loss for training
# takes classifier output and labels
layers {
  layer {
    name: "prob"
    type: "softmax_loss"
  }
  bottom: "ip"
  bottom: "label"
}
```



# What about big nets?

Input





# Technical Challenge: Computing Gradients

- **Computing gradients by hand** is tedious and error-prone
  - Checking numerically by finite differences is a must!
- Theano, by way of **symbolic differentiation** will compute your derivatives for you, as long as you assemble a symbolic graph

# Theano: teaser

Create a function which computes the derivative of some expression  $y$  w.r.t. its parameter  $x$ . For example, we can compute the gradient of  $x^2$  with respect to  $x$ .

```
from theano import pp
import theano.tensor as T
x = T.dscalar('x')
y = x ** 2
gy = T.grad(y, x)
pp(gy) # print out the gradient prior to
optimization

'((fill((x ** TensorConstant{2}),
TensorConstant{1.0}) * TensorConstant{2}) *
(x ** (TensorConstant{2} -
```

```
f = function([x], gy)
print f(4)
print f(94.2)

8.0
188.4
```

# Technical Challenge: Optimization

By and far, Stochastic Gradient Descent (on mini-batches) is most popular:

$$\theta \leftarrow \theta - \eta \frac{\partial L}{\partial \theta}, \eta \in (0, 1)$$

Usually with one or more “tricks”, e.g. momentum:

$$\begin{aligned}\theta &\leftarrow \theta - \eta \Delta \\ \Delta &\leftarrow 0.9\Delta + \frac{\partial L}{\partial \theta}\end{aligned}$$

Other tricks include Nesterov momentum, adaptive learning rates, rmsprop, etc...

# Technical Challenge: Hyperparameter Optimization

- Neural networks have many associated architectural and learning settings, we call these “hyperparameters”, e.g.
  - Number of layers
  - Number of hidden units in each layer
  - Learning rate
  - Regularization (e.g. weight decay)
  - When to stop training (overstopping)
- How to set these?

# Hyperparameter Optimization

- Traditionally, hyperparameters have been set by:
  - expert knowledge\* (experience)
  - cross-validation
- A number of approaches have been proposed recently for auto-tuning models based on Sequential Model-Based Global Optimization strategies, e.g. Bayesian Optimization

Bergstra et al. Making a science of model search: hyperparameter optimization in hundred of dimensions for vision architectures, ICML 2013

# Hyperopt

<http://jaberg.github.io/hyperopt/>

Hyperopt (James Bergstra and collaborators) is a Python library for optimizing over awkward search spaces with real-valued, discrete, and conditional dimensions

```
# define an objective function
def objective(args):
    case, val = args
    if case == 'case 1':
        return val
    else:
        return val ** 2

# define a search space
from hyperopt import hp
space = hp.choice('a',
    [
        ('case 1', 1 + hp.lognormal('c1', 0, 1)),
        ('case 2', hp.uniform('c2', -10, 10))
    ])

# minimize the objective over the space
from hyperopt import fmin, tpe
best = fmin(objective, space, algo=tpe.suggest, max_evals=100)

print best
# -> {'a': 1, 'c2': 0.01420615366247227}
print hyperopt.space_eval(space, best)
# -> ('case 2', 0.01420615366247227)
```

# Spearmint

<https://github.com/JasperSnoek/spearmint>

Spearmint (Jasper Snoek and collaborators) is a Python-based software package to perform Bayesian optimization

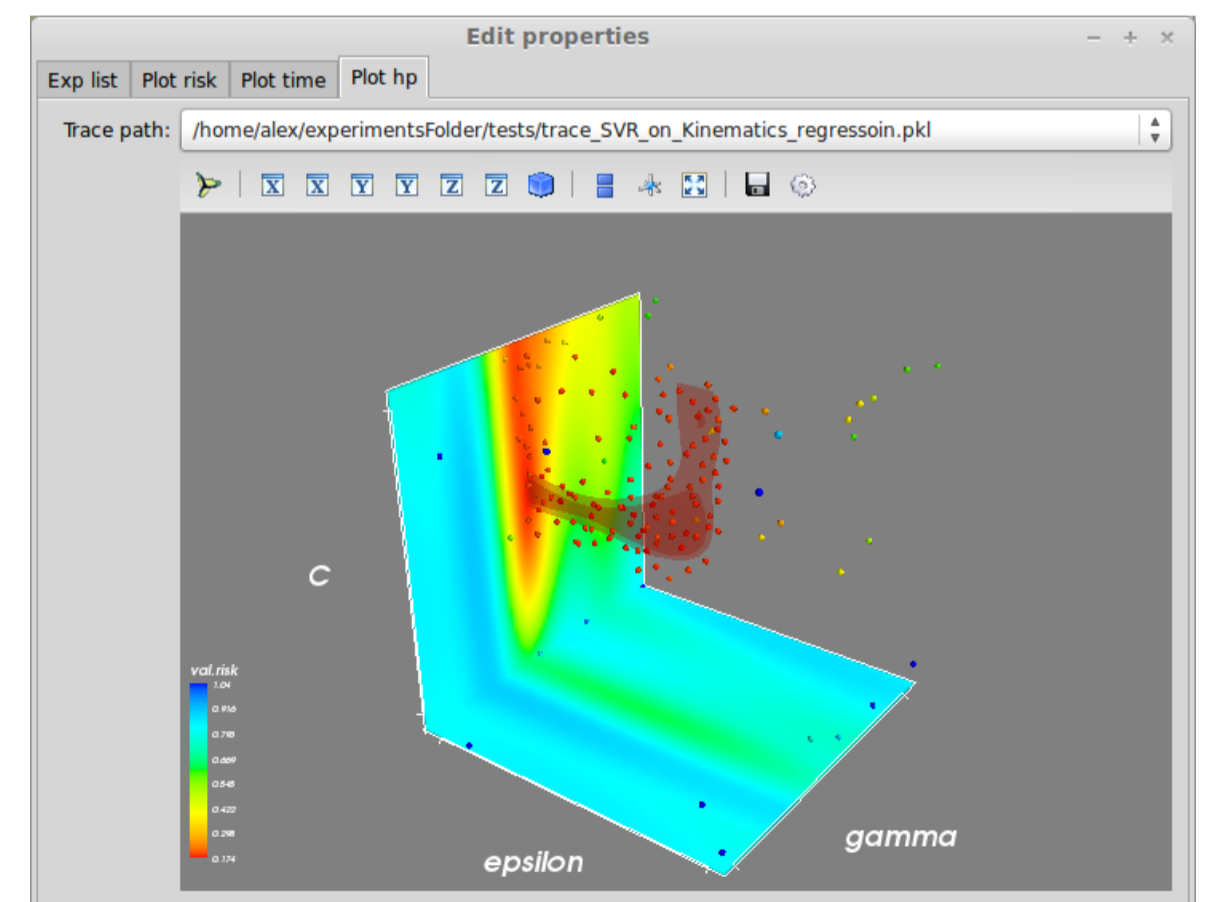
```
from spearmint_salad import hp
from sklearn.svm import SVR

# Encapsulate the class into a hp.Obj to be able to instantiate using variable parameters
# or constant parameters if necessary.
hp_space = hp.Obj(SVR)(
    C = hp.Float( min_val=0.01, max_val=1000, hp.log_scale ), # variable
    kernel = 'rbf', # constant
    gamma = hp.Float( min_val=10**-5, max_val=1000, hp.log_scale ), # variable
    epsilon = hp.Float(min_val=0.01, max_val=1, hp.log_scale), # variable
)
```

```
from spearmint_salad import metric

metric = metric.SquareDiffLoss()
make_salad( hp_space, metric, dataset_path)
```

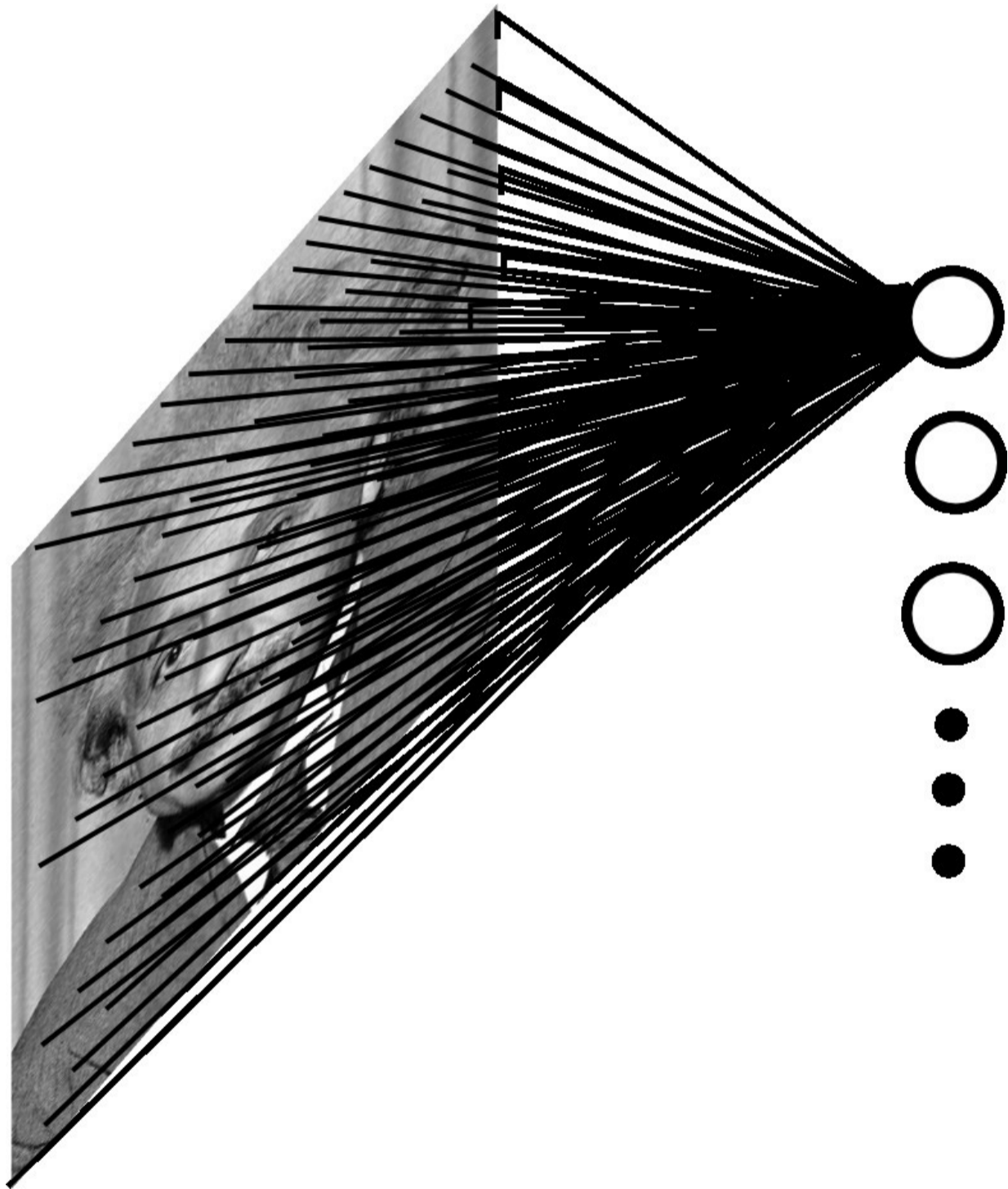
```
python vis.py
```



# Fully-Connected Layer

Example: 200 × 200 image  
40k hidden units

➔ ~2B parameters!

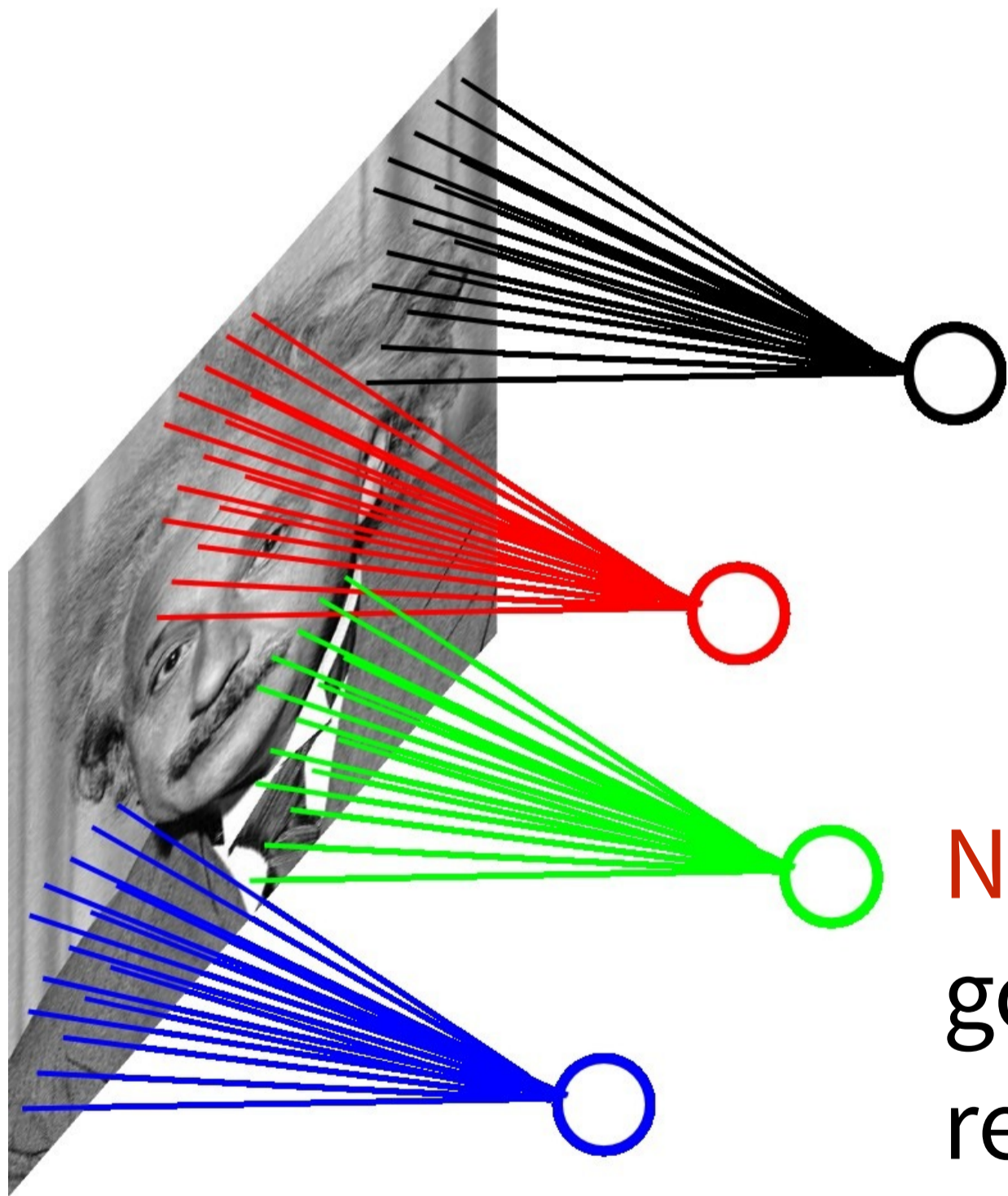


- Spatial correlation is local
- Waste of resources
- We don't have enough training examples to fit!



# Locally-Connected Layer

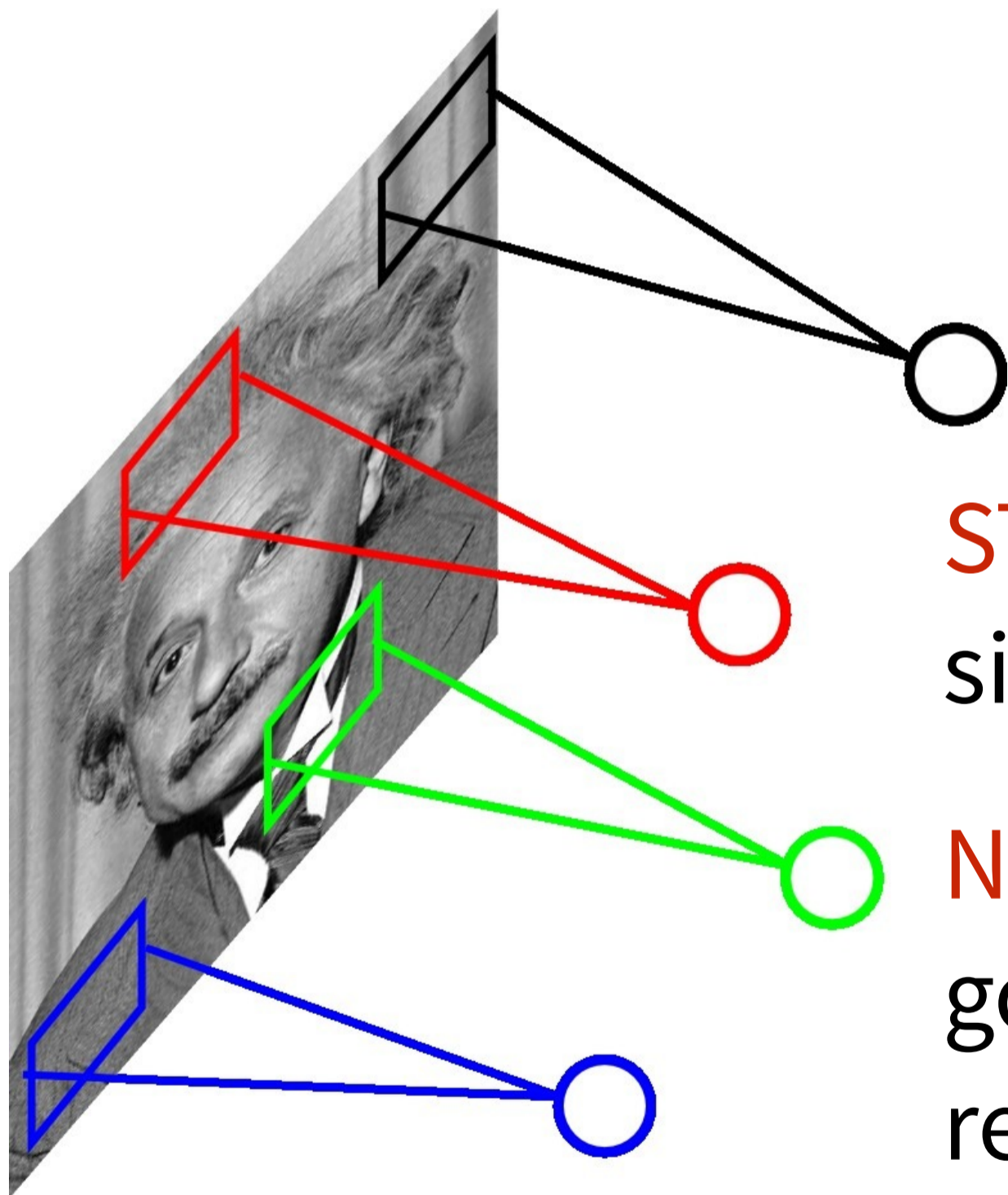
Example: 200 × 200 image  
40k hidden units  
Filter size: 10 × 10  
4M parameters



**Note:** this parameterization is good when the input images are registered (e.g. face recognition)

# Locally-Connected Layer

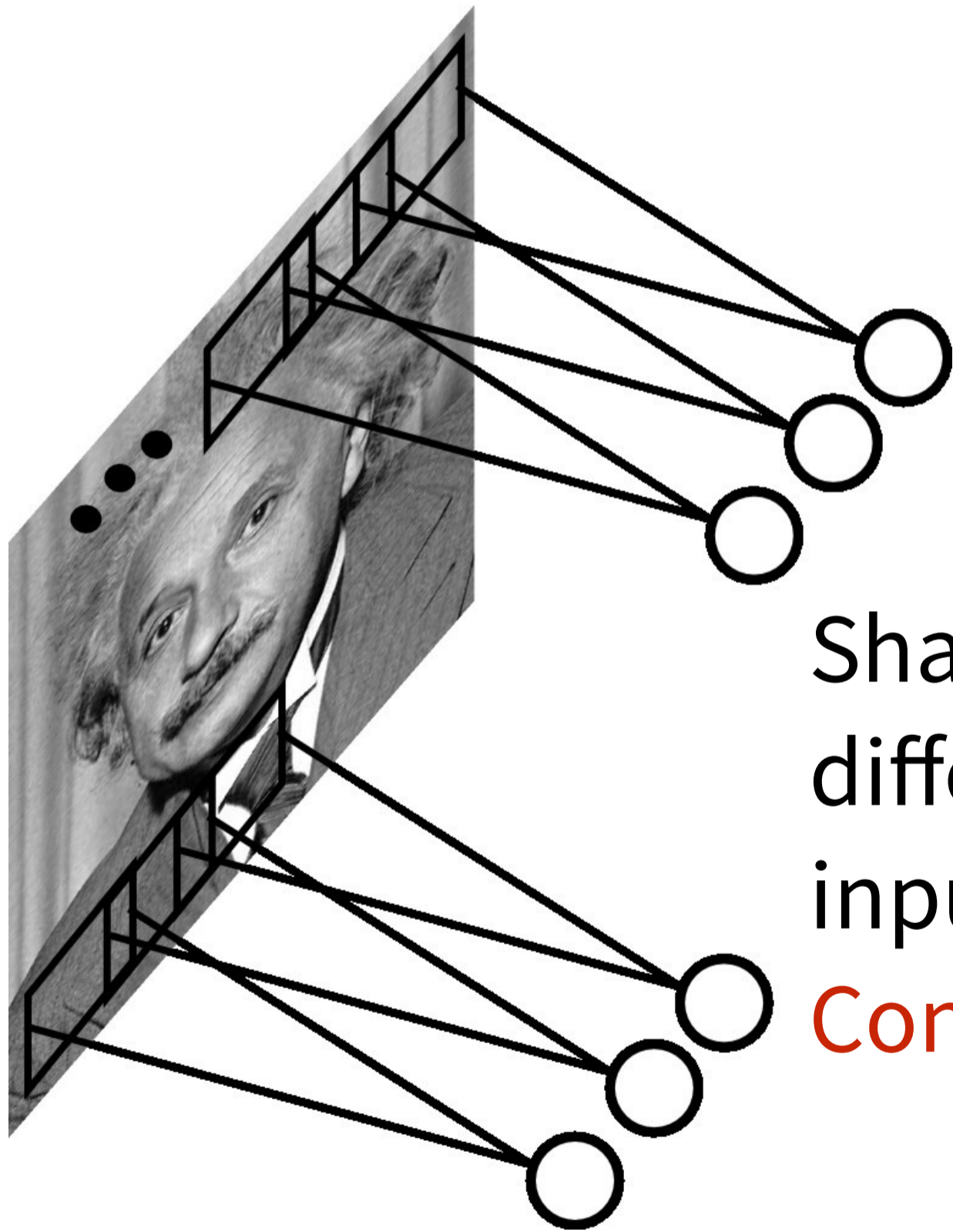
Example: 200 × 200 image  
40k hidden units  
Filter size: 10 × 10  
4M parameters



**STATIONARITY?** Statistics are similar at different locations.

**Note:** this parameterization is good when the input images are registered (e.g. face recognition)

# Convolutional Layer



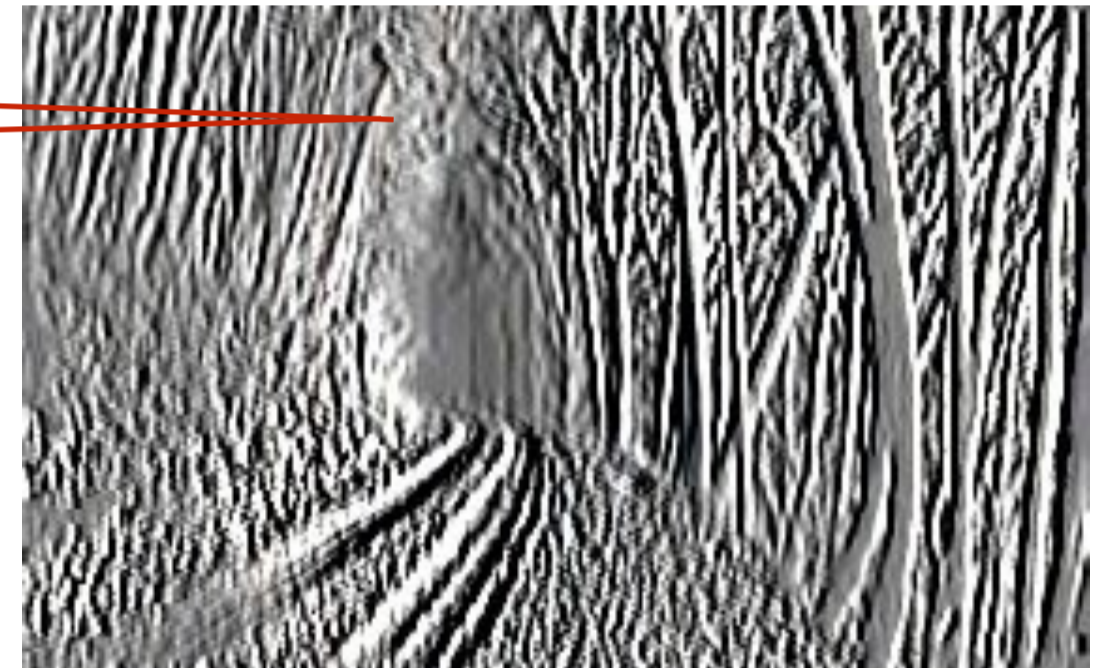
Share the same parameters across different locations (assuming input is stationary)

**Convolutions with learned kernels**

# Convolutional Layer

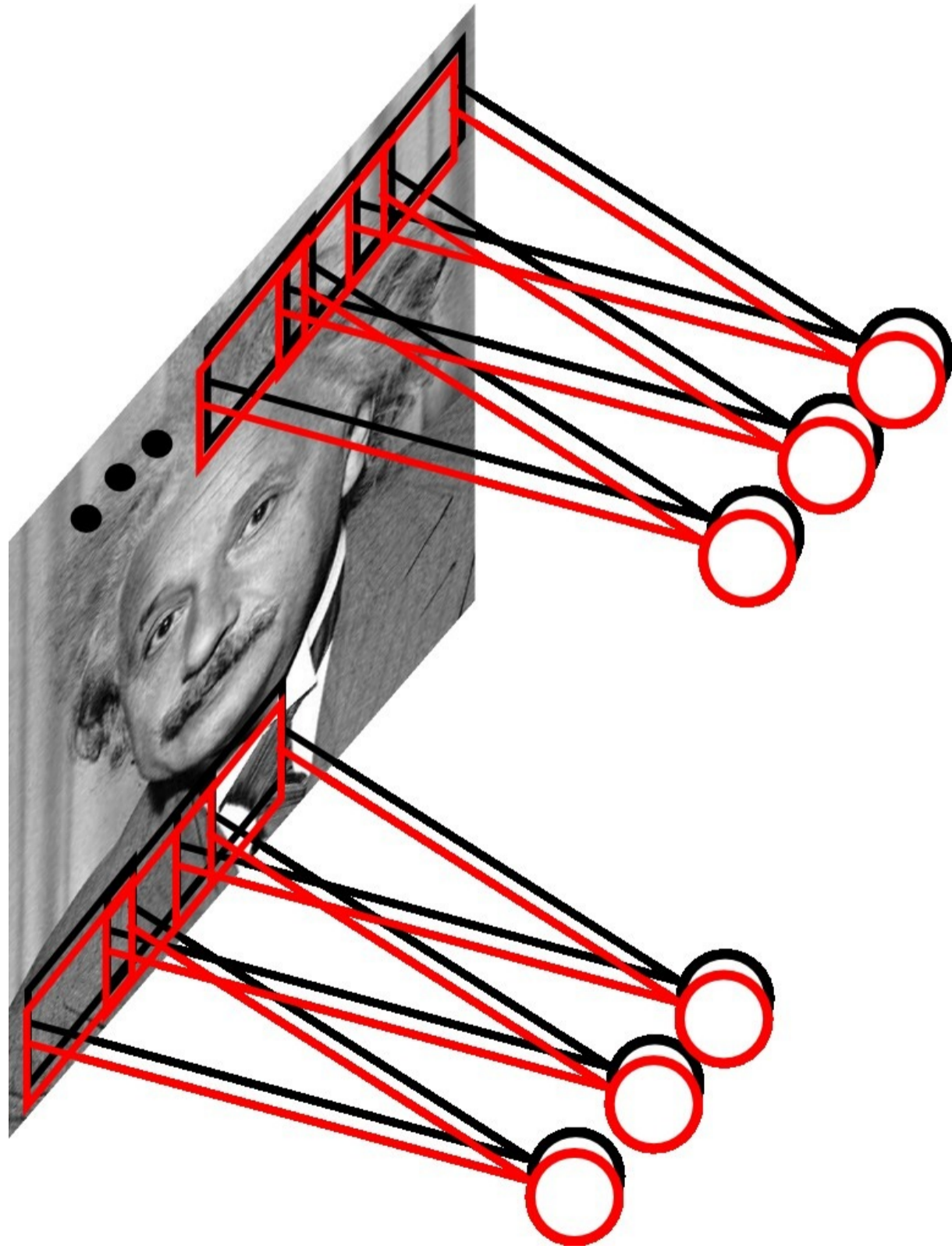


$$* \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} =$$



# Convolutional Layer

Learn **multiple filters**.

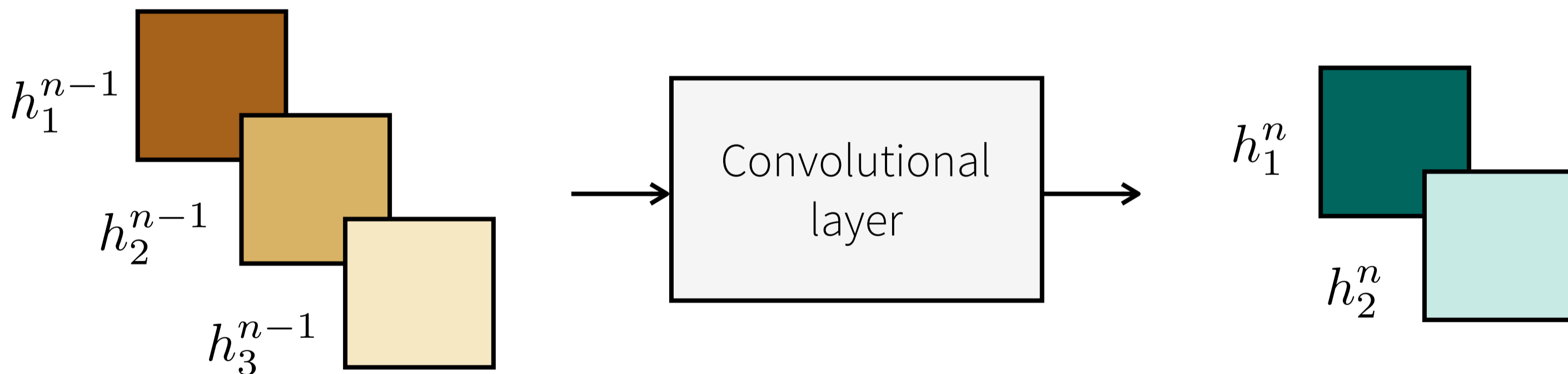


Example: 200 × 200 image  
100 filters  
Filter size: 10 × 10  
10k parameters

# Convolutional Layer

$$h_j^n = \max \left( 0, \sum_{k=1}^K h_k^{n-1} * w_{kj}^n \right)$$

output feature map      input feature map      kernel

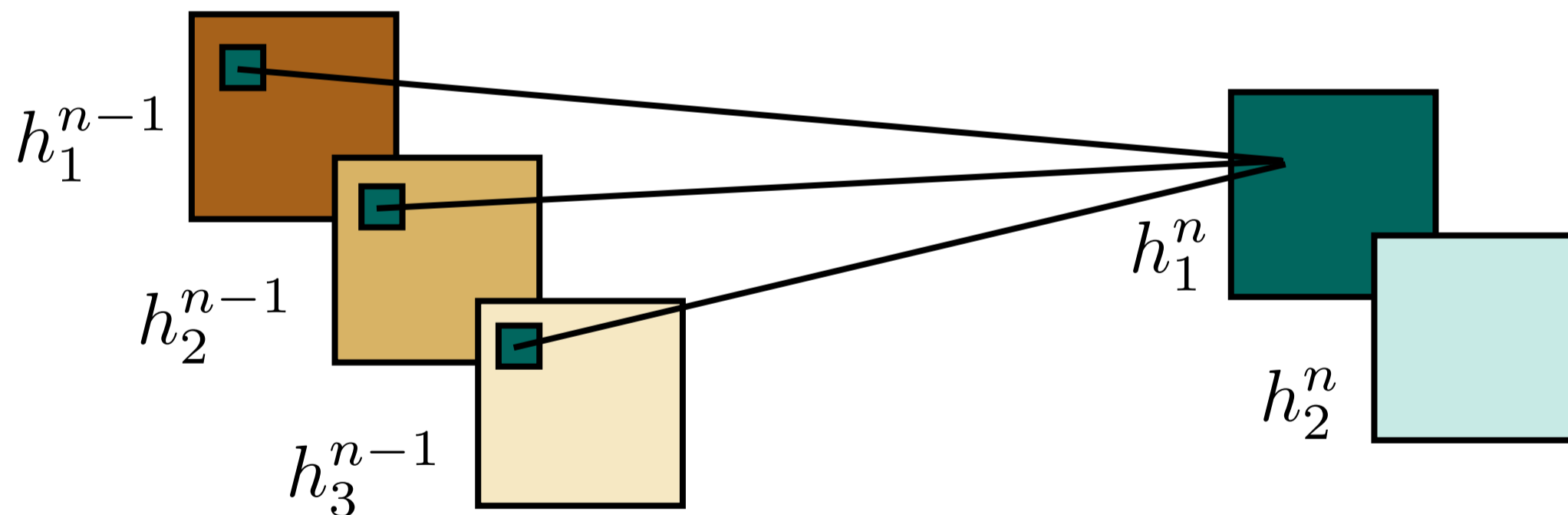


$n$  indexes layers  
 $j$  indexes maps (features)

# Convolutional Layer

$$h_j^n = \max \left( 0, \sum_{k=1}^K h_k^{n-1} * w_{kj}^n \right)$$

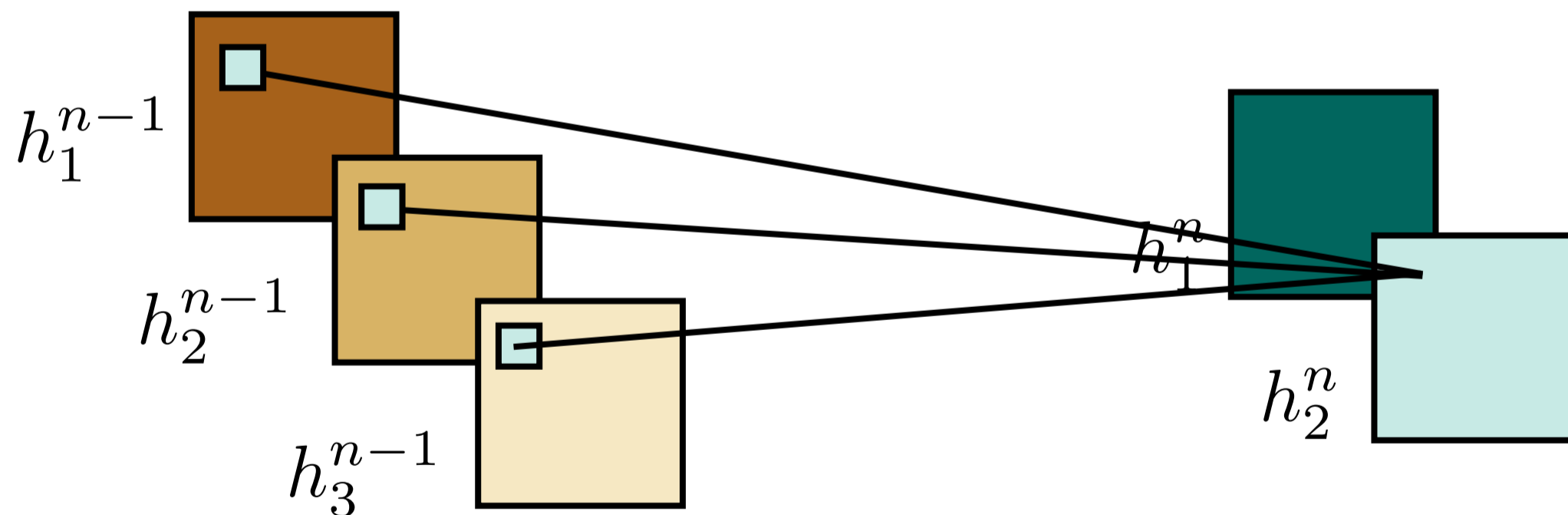
output feature map      input feature map      kernel



# Convolutional Layer

$$h_j^n = \max \left( 0, \sum_{k=1}^K h_k^{n-1} * w_{kj}^n \right)$$

output feature map      input feature map      kernel



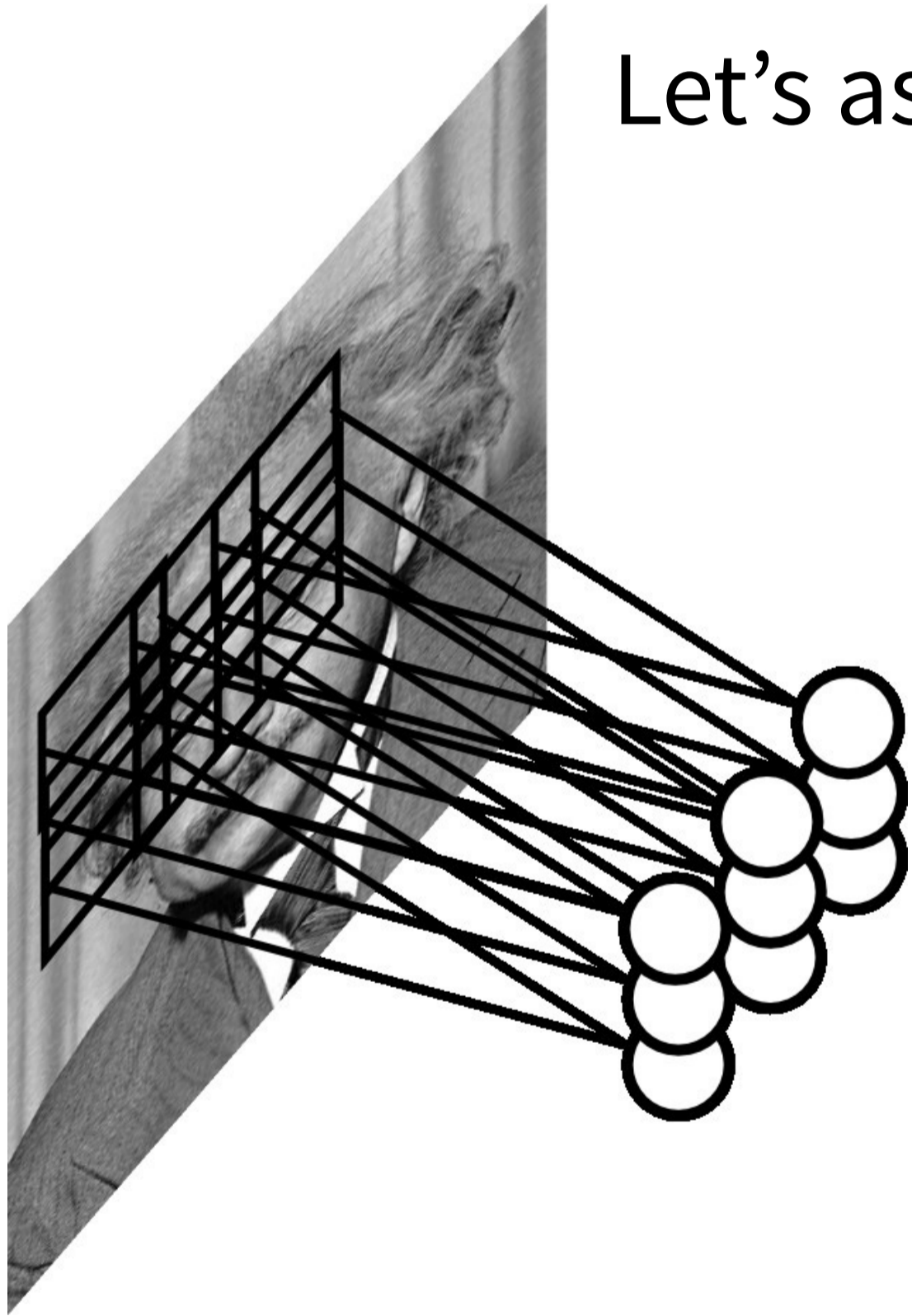


# Convolutional Net - Recap

- A **fully-connected** neural network applied to images
  - scales quadratically with the size of the input
  - does not leverage stationarity
- Solution
  - connect each hidden to a small patch of input
  - share the weights across space
- This is called a **convolutional layer**
- A network with convolutional layers is called a **convolutional net**

# Pooling Layer

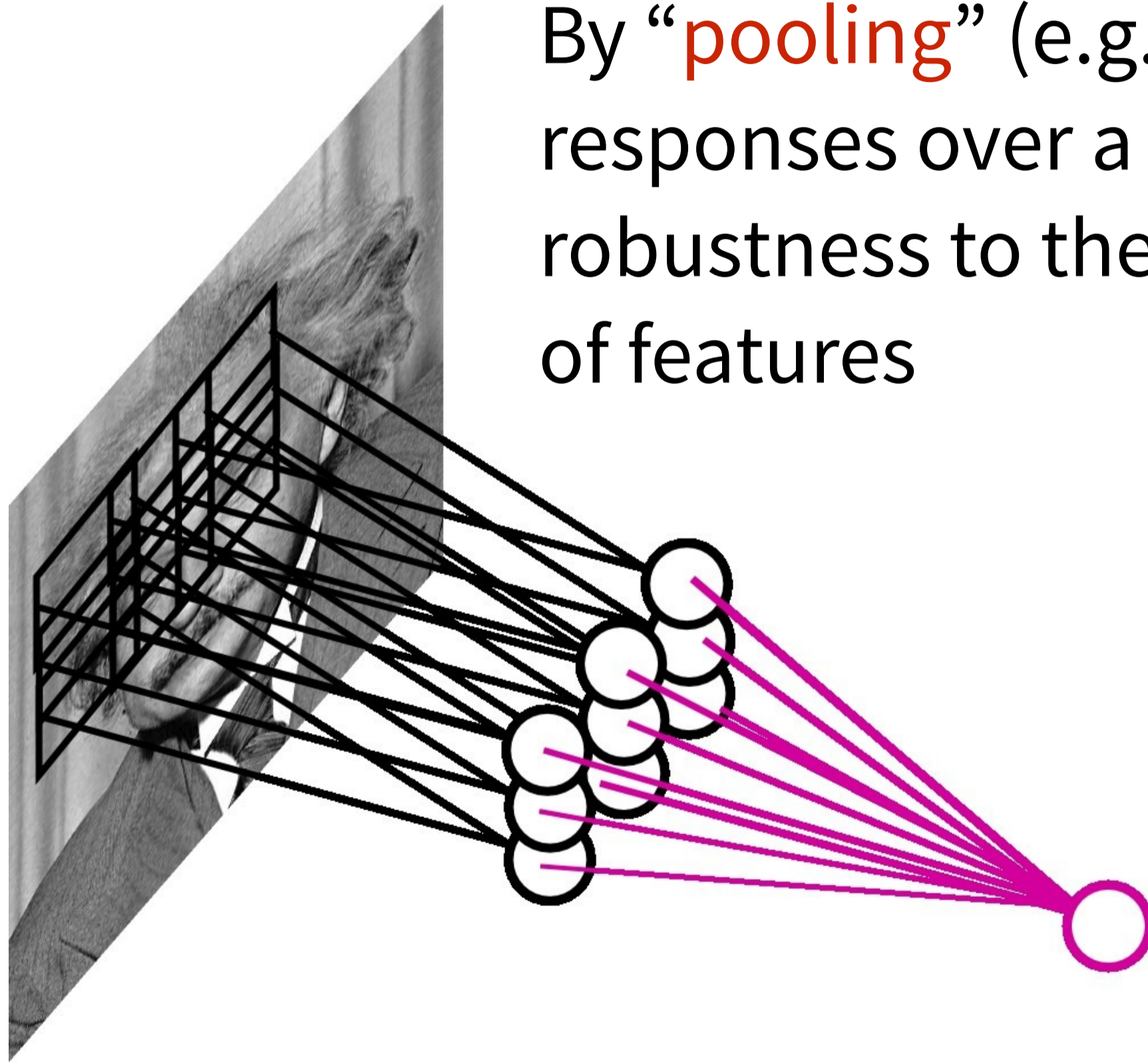
Let's assume the filter is an "eye" detector



How can we make the detection robust to the exact location of the eye?

# Pooling Layer

By “pooling” (e.g. taking max) the filter responses over a local region we gain robustness to the exact spatial location of features

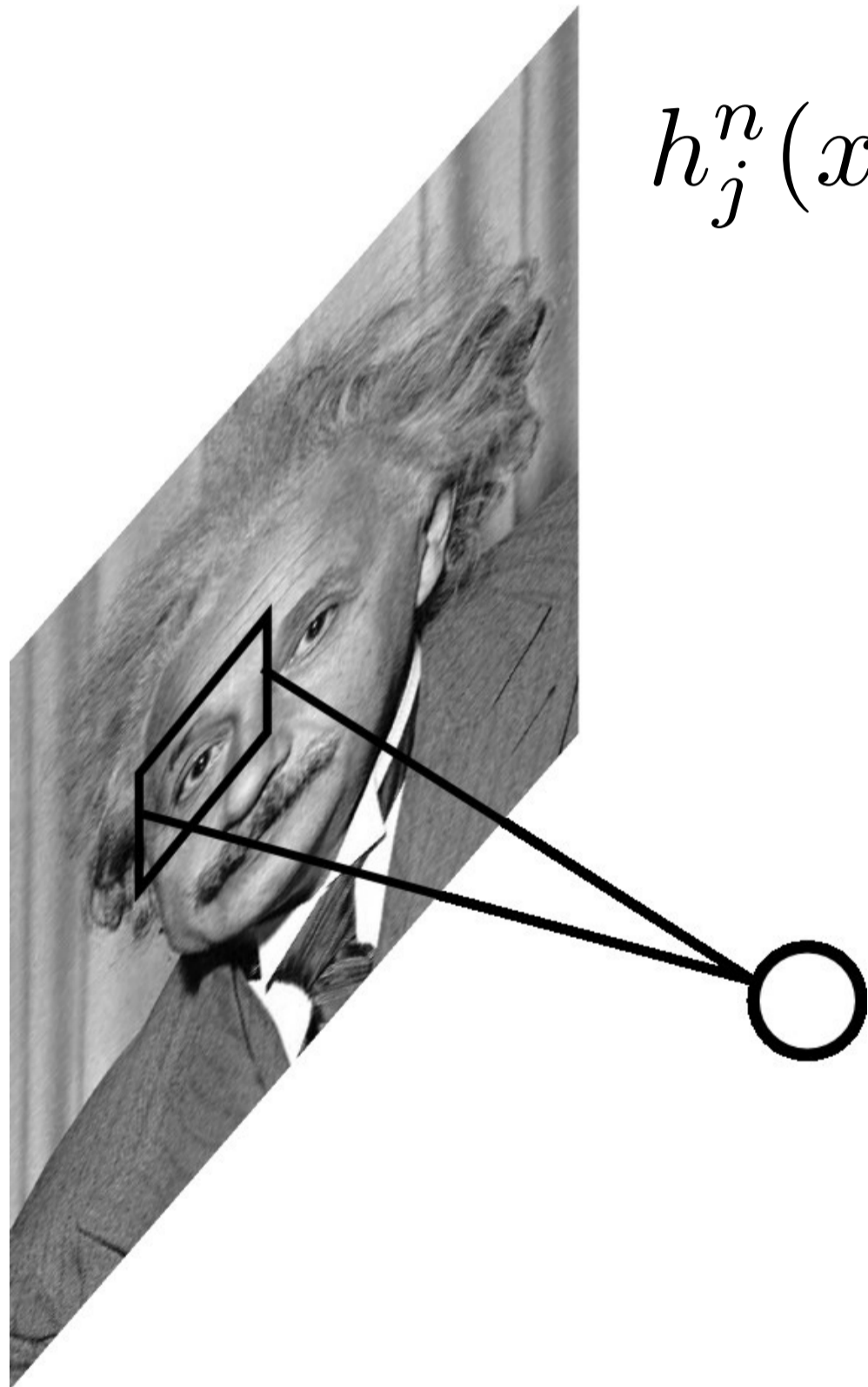


“Shift invariance”

# Types of Pooling

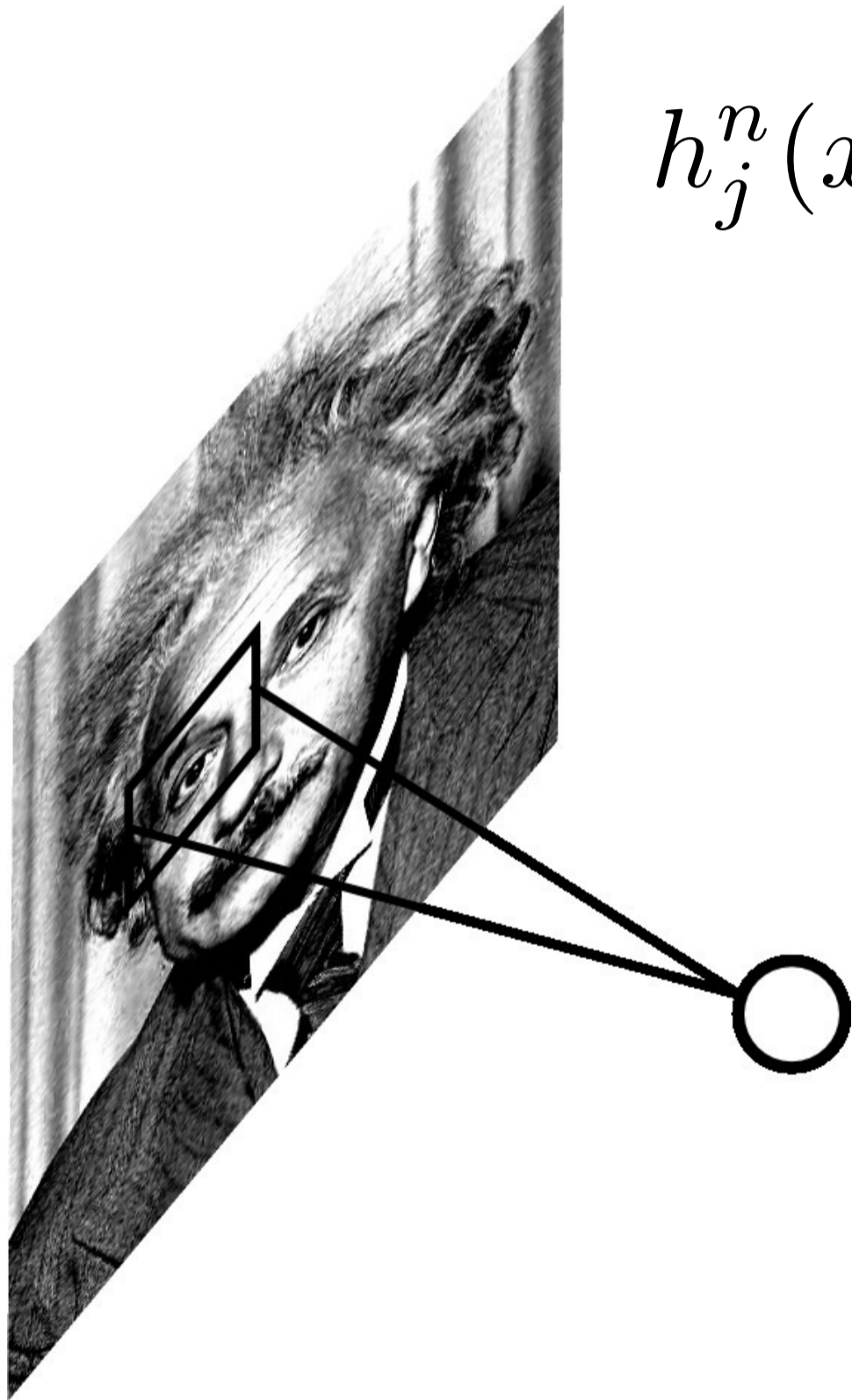
Max-Pooling	$h_j^n(x, y) = \max_{x' \in N(x), y' \in N(y)} h_j^{n-1}(x', y')$
Average-Pooling	$h_j^n(x, y) = \frac{1}{K} \sum_{x' \in N(x), y' \in N(y)} h_j^{n-1}(x', y')$
L2-Pooling	$h_j^n(x, y) = \sqrt{\sum_{x' \in N(x), y' \in N(y)} h_j^{n-1}(x', y')^2}$
L2-Pooling Over Features	$h_j^n(x, y) = \sqrt{\sum_{k \in N(j)} h_k^{n-1}(x, y)^2}$

# Local Contrast Normalization



$$h_j^n(x, y) = \frac{h^{n-1}(x, y) - m_j^{n-1}(N(x, y))}{\sigma_j^{n-1}(N(x, y))}$$

# Local Contrast Normalization

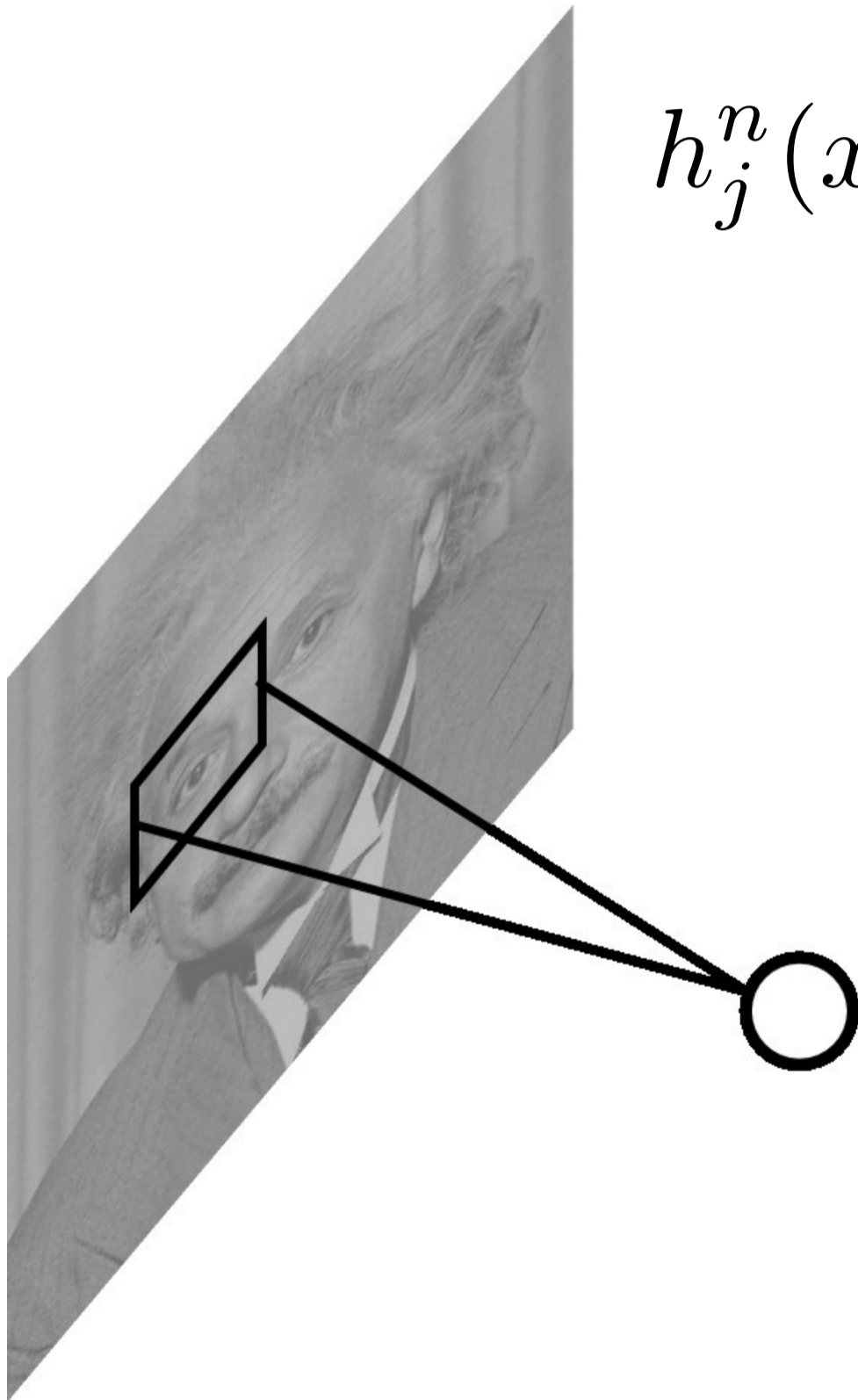


$$h_j^n(x, y) = \frac{h^{n-1}(x, y) - m_j^{n-1}(N(x, y))}{\sigma_j^{n-1}(N(x, y))}$$

Control the dynamic range  
of each feature map.

# Local Contrast Normalization

$$h_j^n(x, y) = \frac{h^{n-1}(x, y) - m_j^{n-1}(N(x, y))}{\sigma_j^{n-1}(N(x, y))}$$



Performed also across features  
and in the higher layers...

Effects:

- improves invariance
- improves optimization
- increases sparsity

**Note:** computational cost is negligible  
compared to convolutional layer.

# Convnets: Single Stage

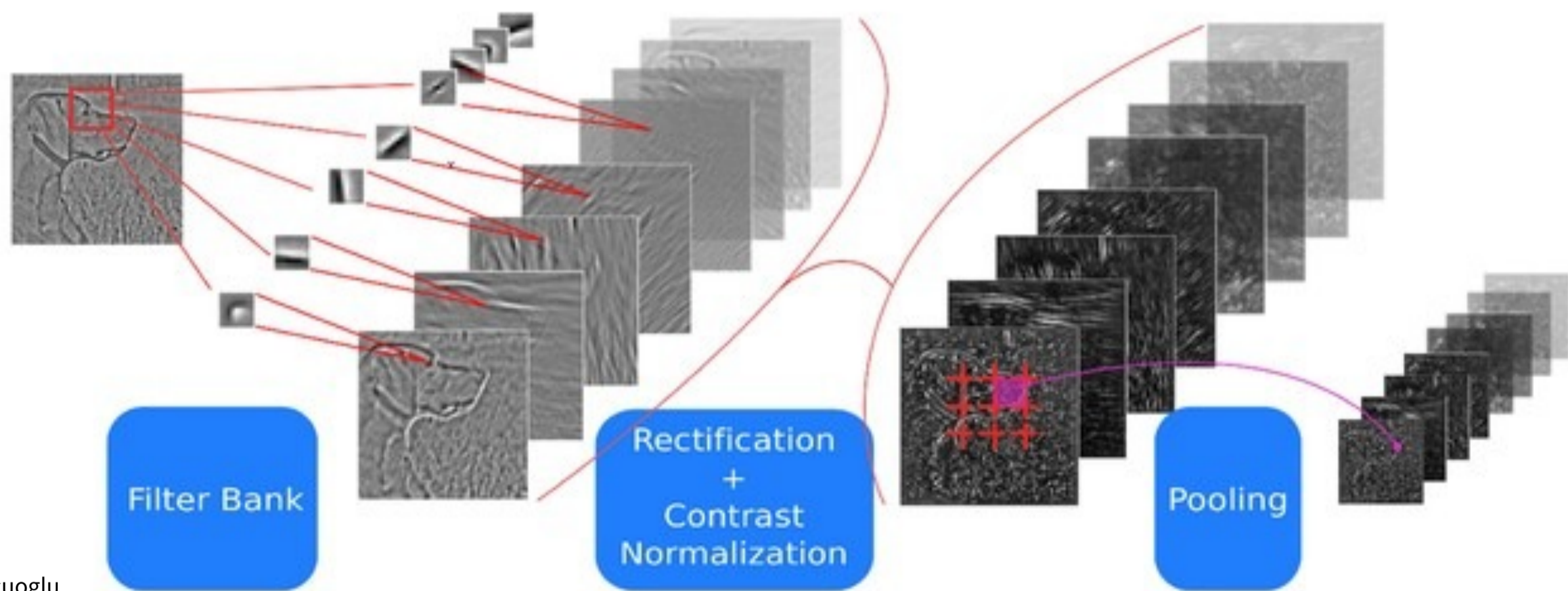


Image credit: Koray Kavukcuoglu

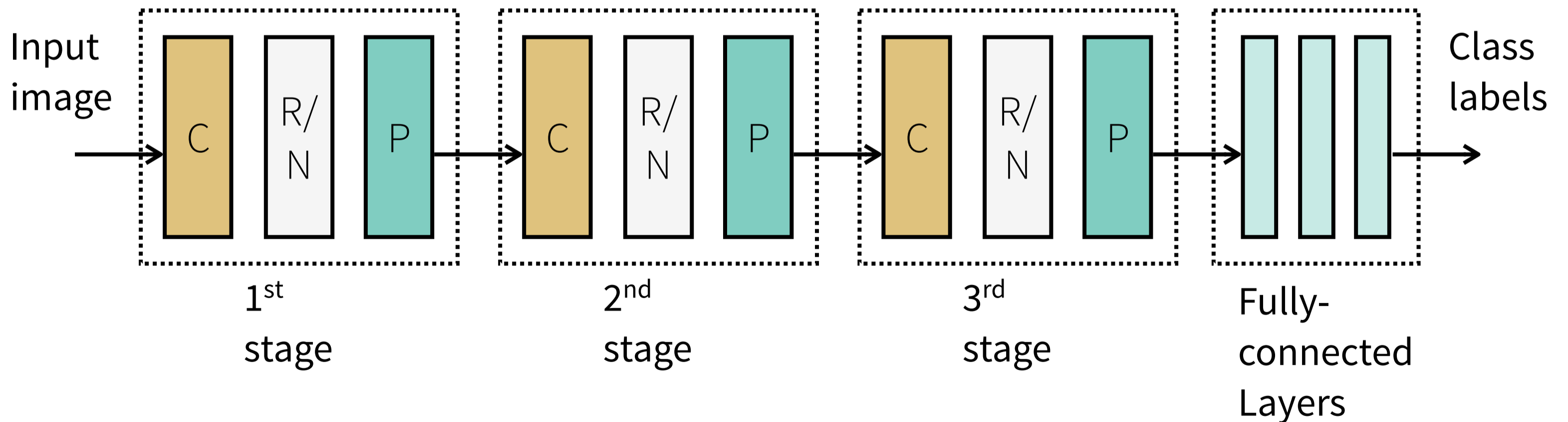


# Convnets: Typical Architecture

## Single stage



## Whole system

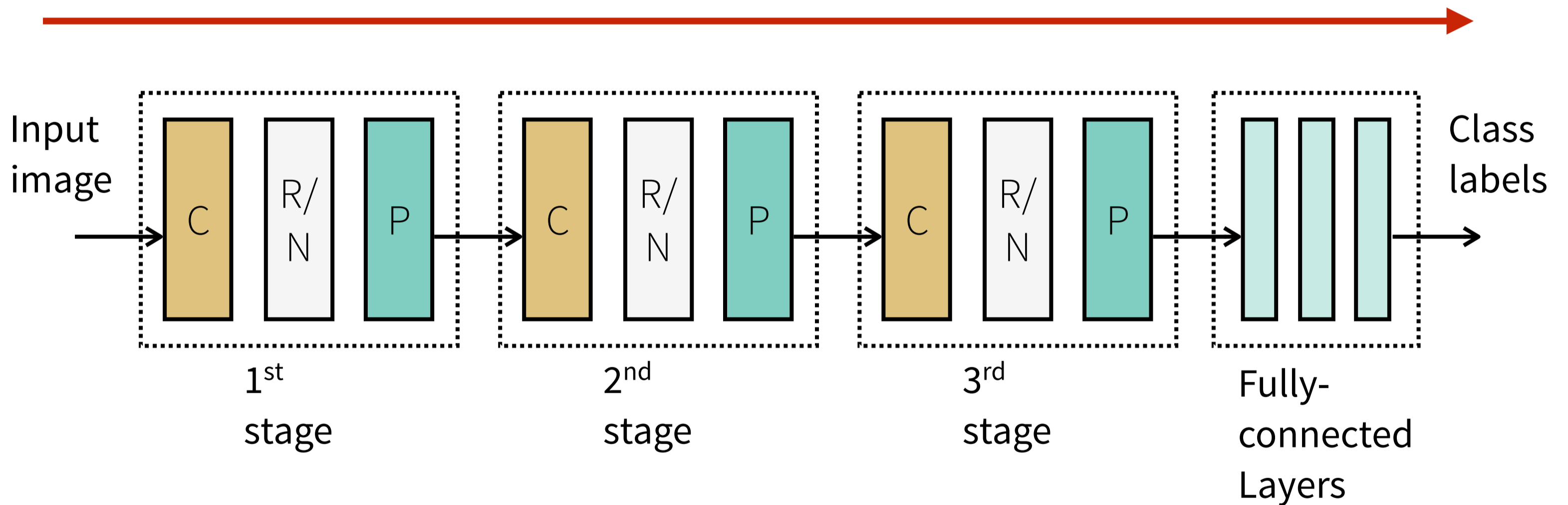


# Convnets: Training

- All layers are differentiable
- We can use standard back-propagation
- Algorithm:
  - Given a small mini-batch:
    - F-Prop
    - B-Prop
    - Parameter updates

# Convnets: Testing

At test time, only run forward propagation



Convnets can naturally process larger images at little cost.  
Traditional methods use inefficient sliding windows.

# Convnets: today

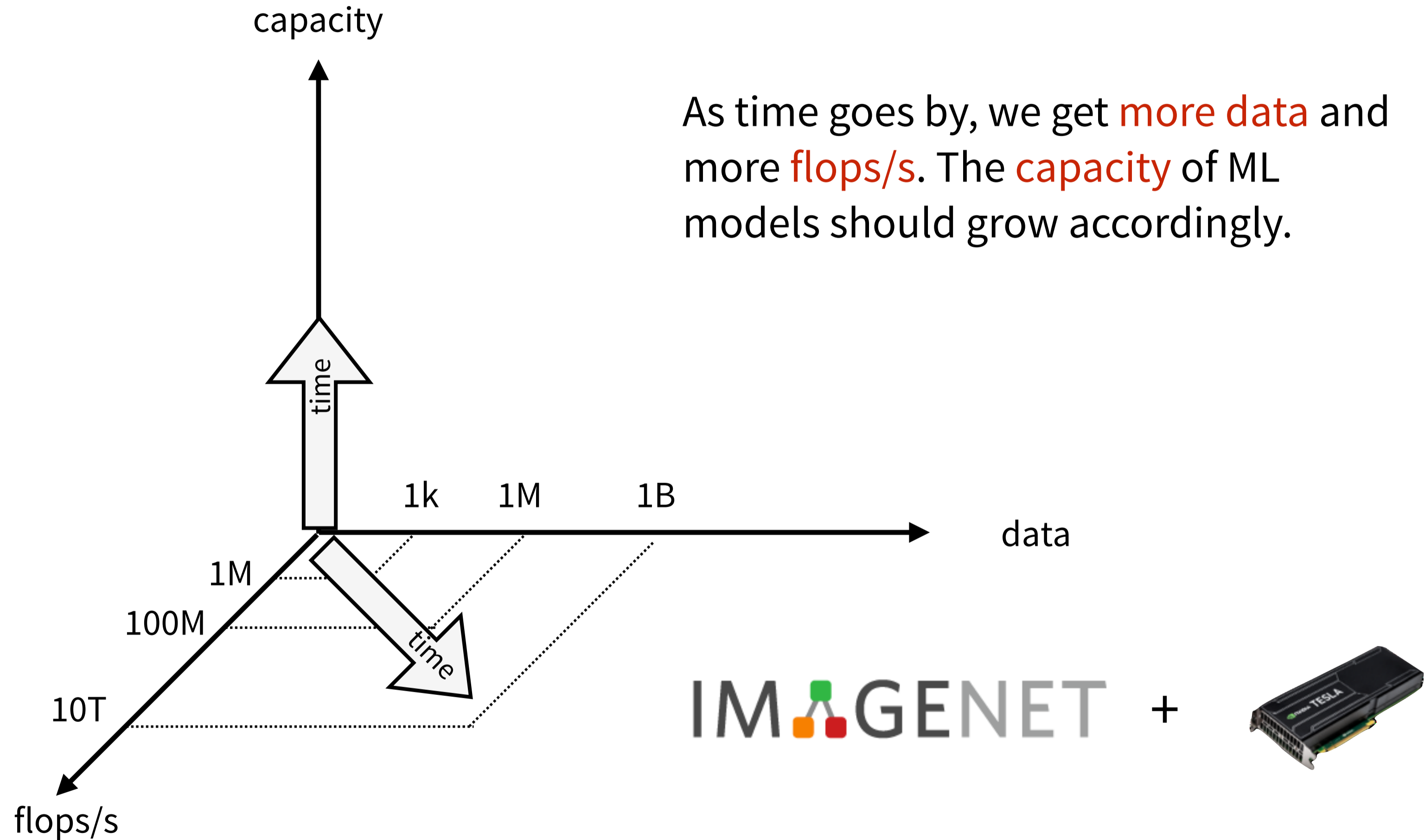
- Until 2012, the common wisdom was that training didn't work because we would “get stuck in local minima”
- Local minima are all similar, there are long plateaus, and it can take a long time to break symmetries
- Optimization is not the real problem, when:
  - the dataset is large
  - units do not saturate much
  - we use normalization layers

# Technical Challenge: Scalability

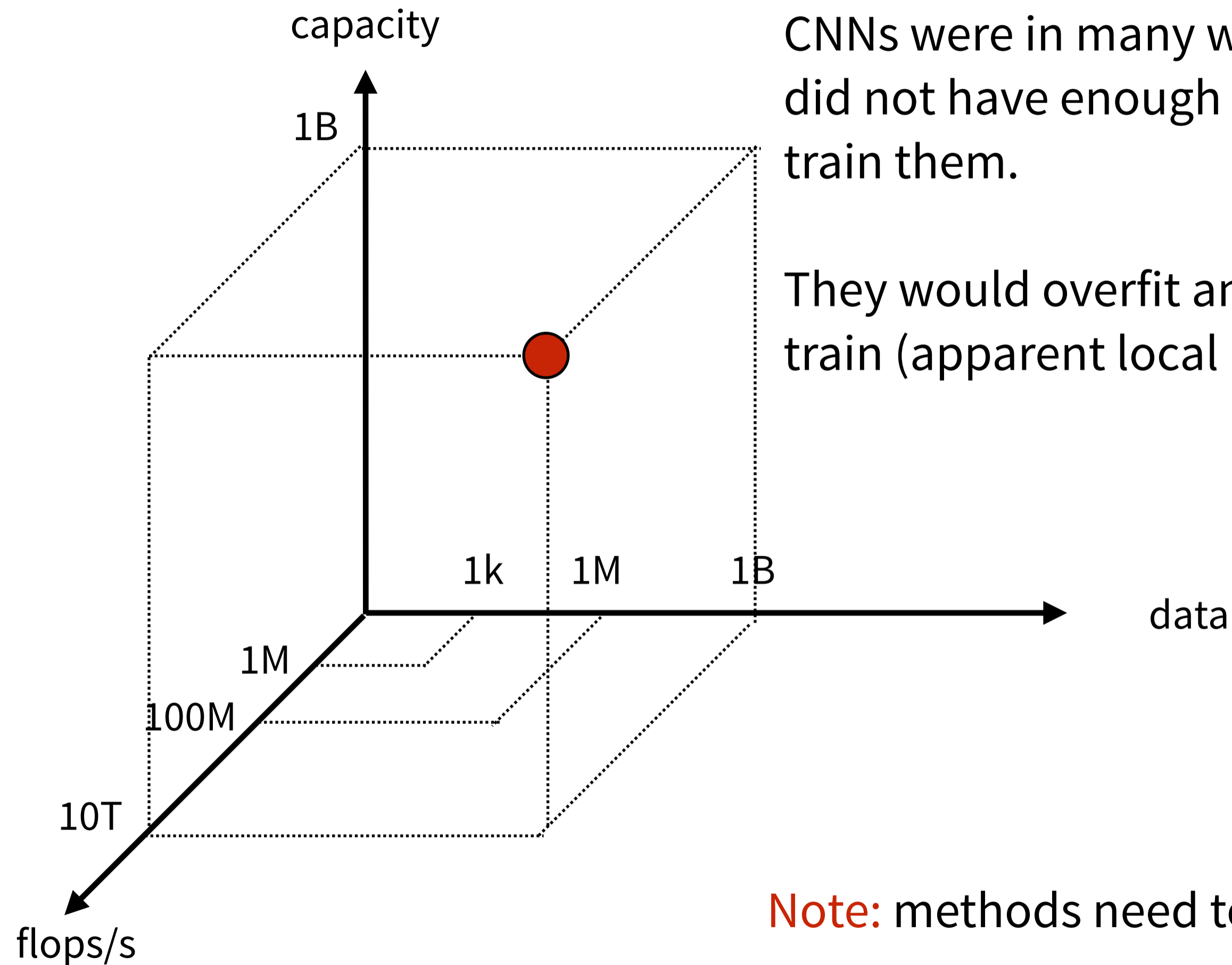
- The real challenges are:
  - **Generalization**
    - How many training examples to fit 1B params?
    - How many parameters/samples to model spaces with 1M dimensions?
  - **Scalability**

# Convnets: why so successful now?

As time goes by, we get **more data** and more **flops/s**. The **capacity** of ML models should grow accordingly.



# Convnets: why so successful now?




CNNs were in many ways **premature**: we did not have enough data and flops/s to train them.

They would overfit and be too slow to train (apparent local minima).

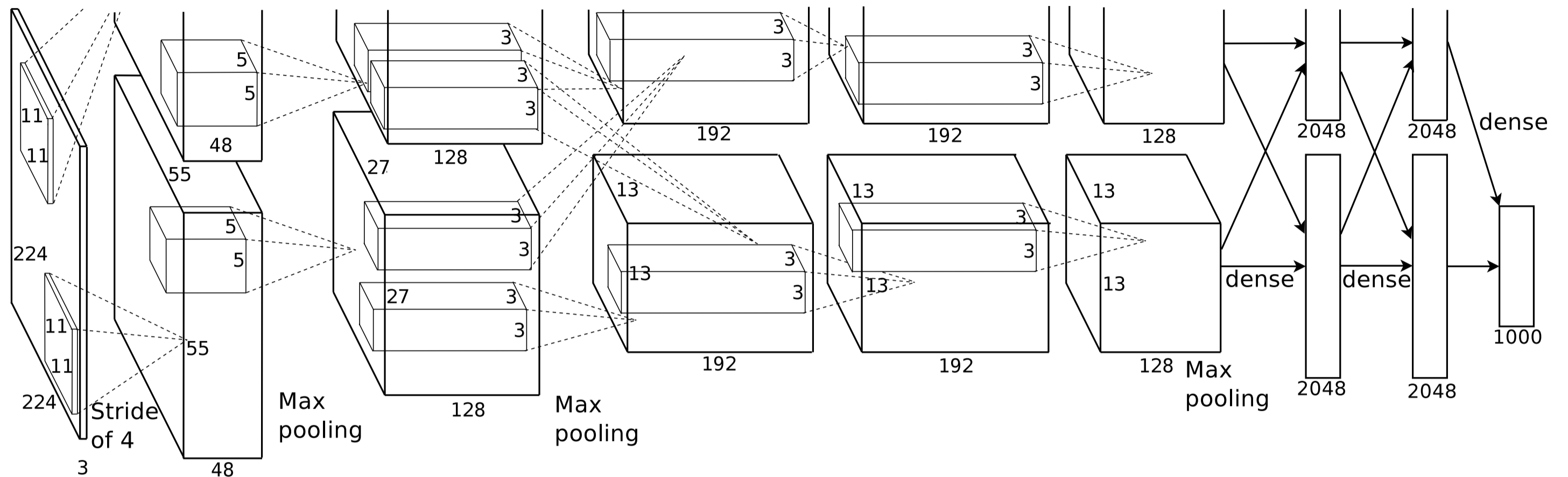
**Note:** methods need to be easily scalable!

# Tools: Scalability

- DL particularly well-suited to parallelization:
  - Data parallelism inherent in pixel-based inputs (e.g. images and videos)
  - Task parallelism inherent in redundant processing units (neurons)
- Hardware accelerators (e.g. GPUs) 
  - Torch 7, Theano, Caffe all provide GPU support by way of CUDA
- Distributed frameworks (e.g. Google, Microsoft)



# Motivation

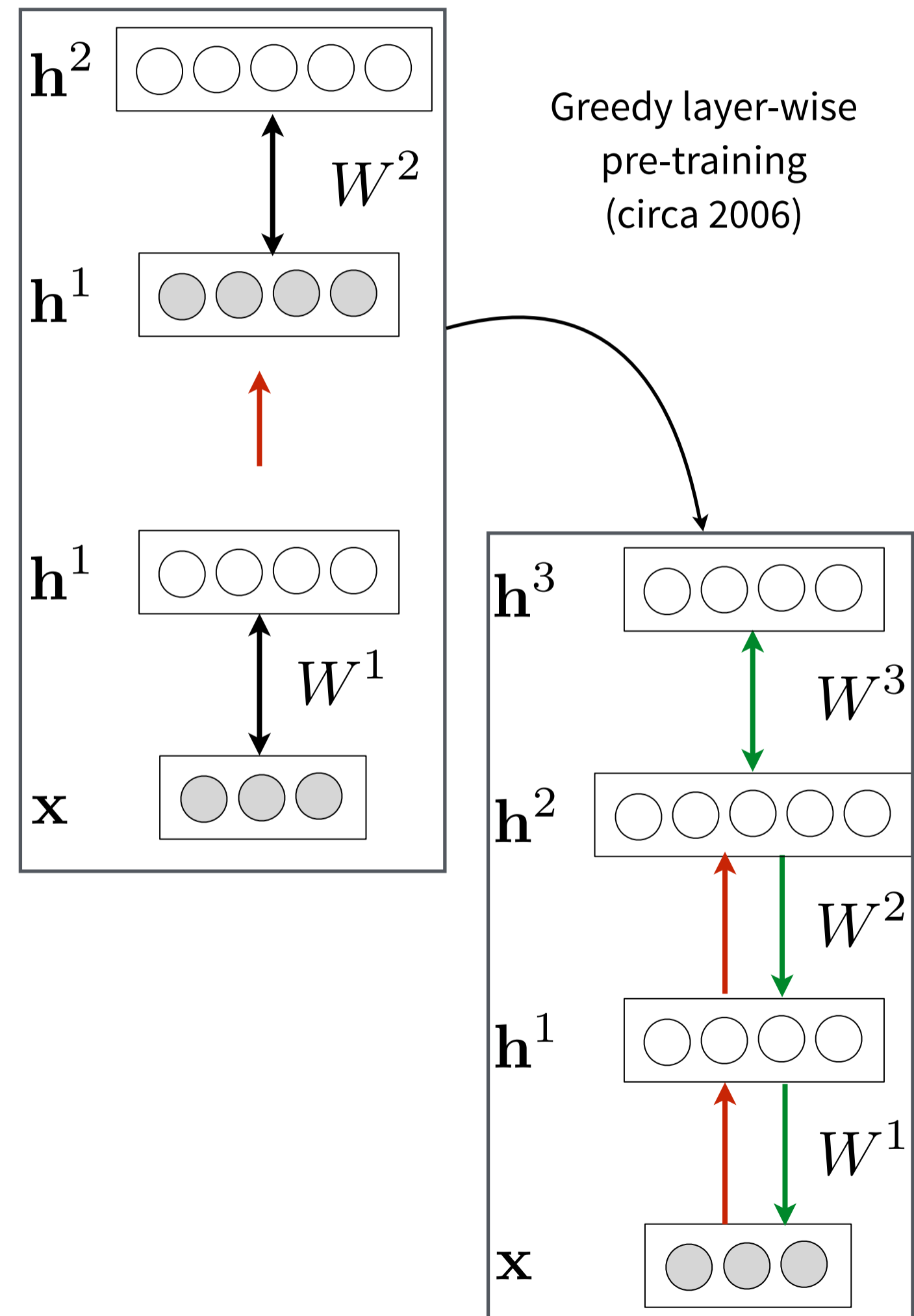


- Most impressive results in deep learning have been obtained with **purely supervised learning methods**
- In vision, typically classification (e.g. object recognition)
- Though progress has been slower, it is likely that **unsupervised learning will be important to future advances** in DL

Image: Krizhevsky (2012) - AlexNet, the “hammer” of DL

# An Interesting Historical Fact

- Unsupervised learning was the catalyst for the present DL revolution that started around 2006
- Now we can train deep supervised neural nets without “pre-training”, thanks to
  - Algorithms (nonlinearities, regularization)
  - More data
  - Better computers (e.g. GPUs)
- Should we still care about unsupervised learning?



# Why Unsupervised Learning?

## Reason 1:

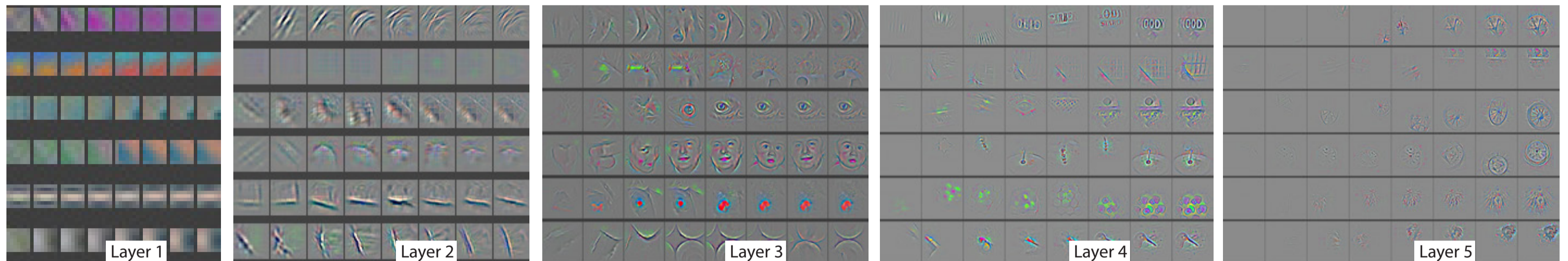
We can exploit unlabelled data; much more readily available and often free.



# Why Unsupervised Learning?

## Reason 2:

We can capture enough information about the observed variables so as to ask new questions about them; questions that were not anticipated at training time.



# Why Unsupervised Learning?

## Reason 3:

Unsupervised learning has been shown to be a good regularizer for supervised learning; it helps generalize.

This advantage shows up in practical applications:

- transfer learning, domain adaptation
- unbalanced classes
- zero-shot, one-shot learning

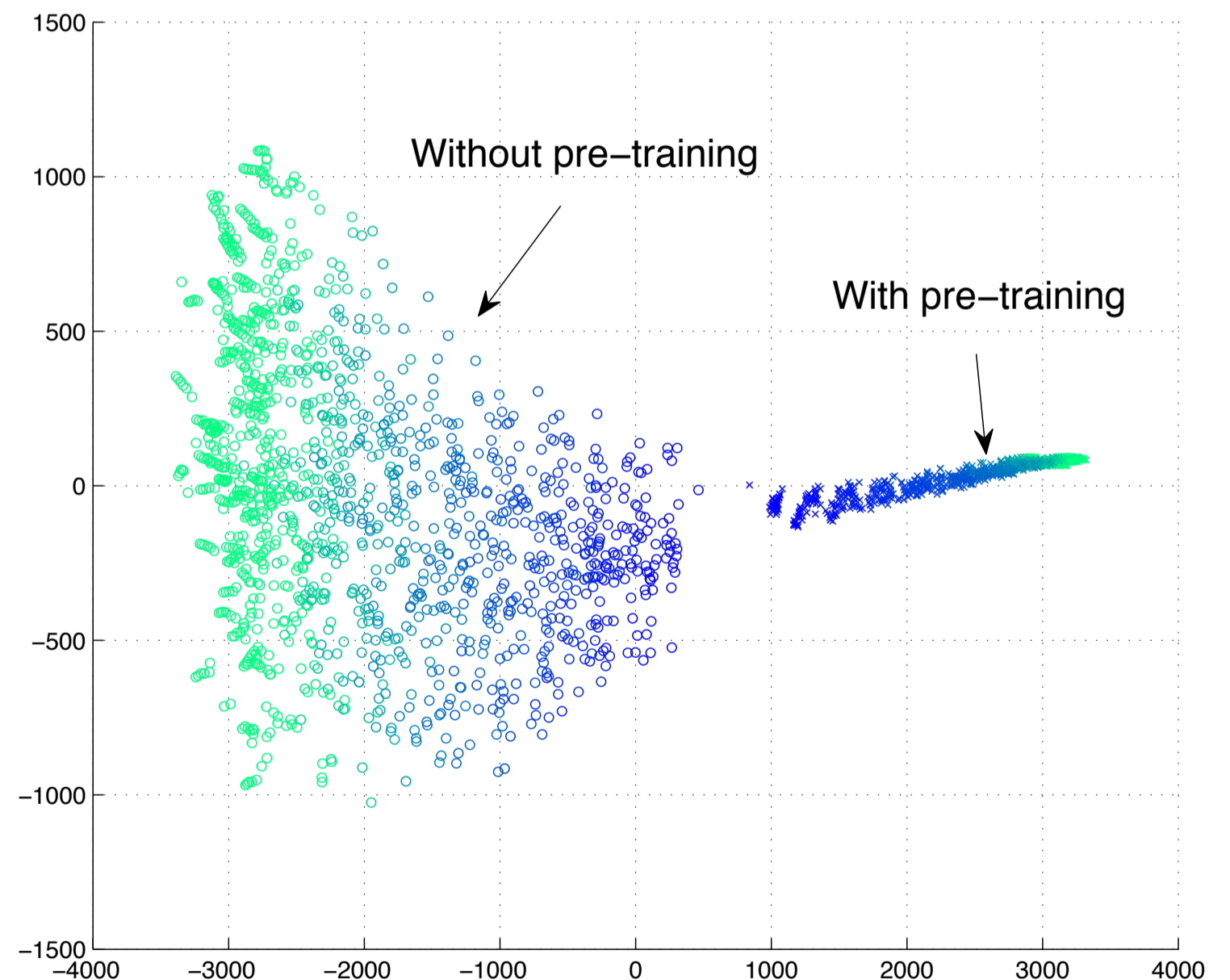
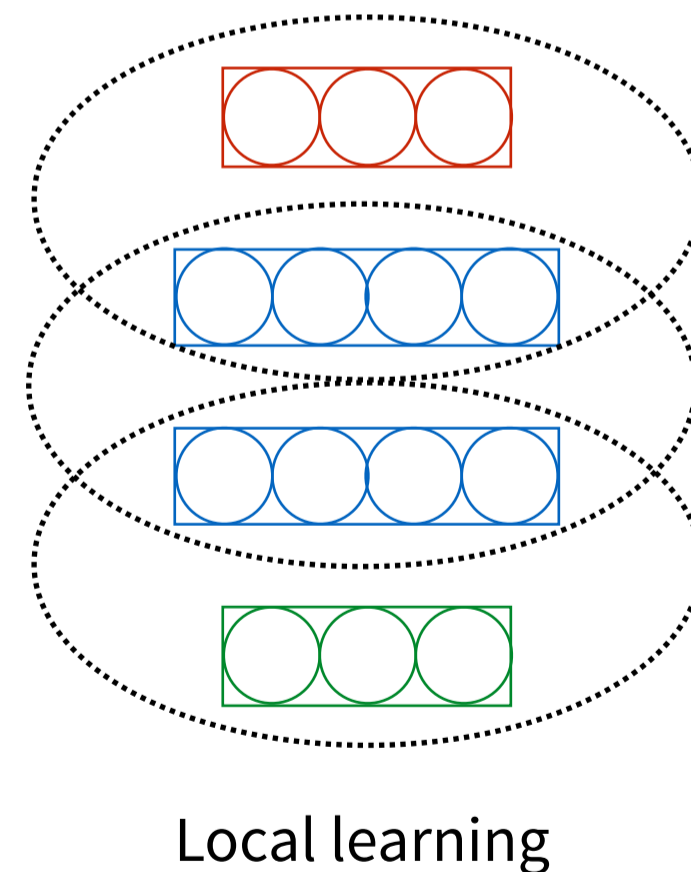
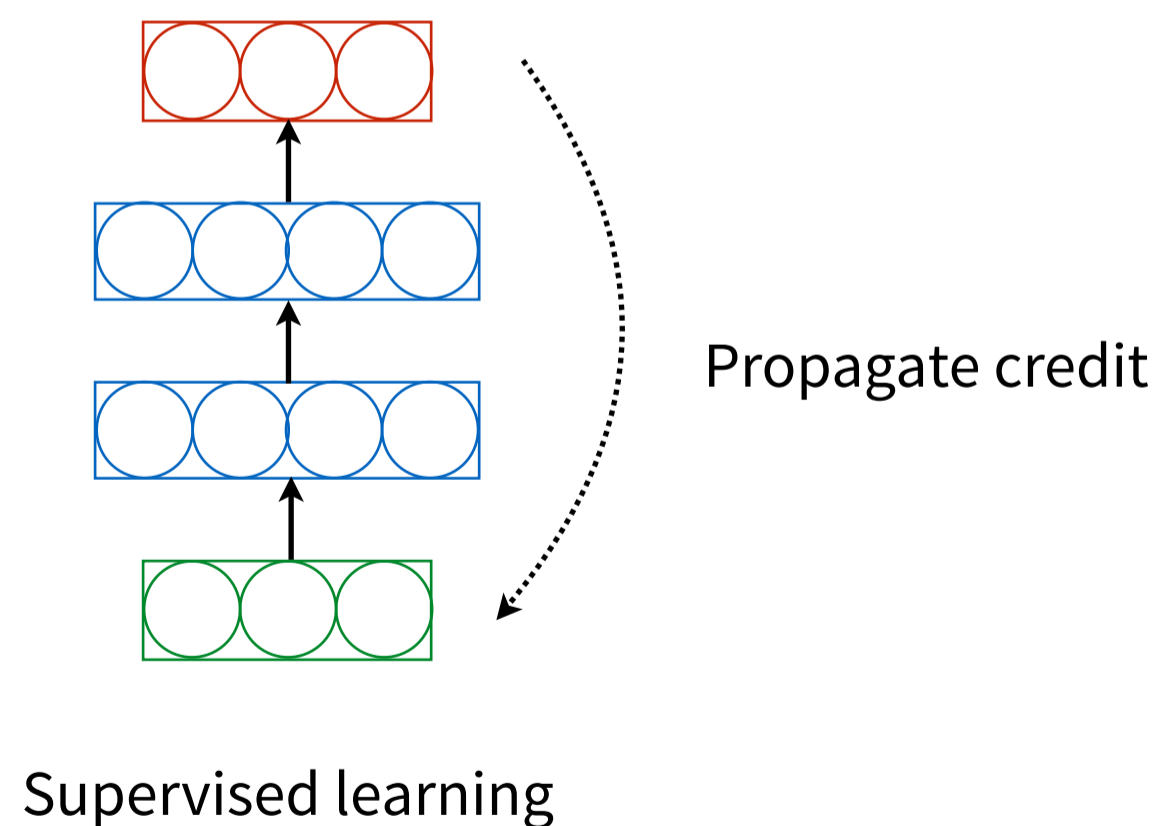


Image: ISOMAP embedding of functions represented by 50 networks  $w$  and  $w/o$  pre training (Erhan et al., 2010)

# Why Unsupervised Learning?

## Reason 4:

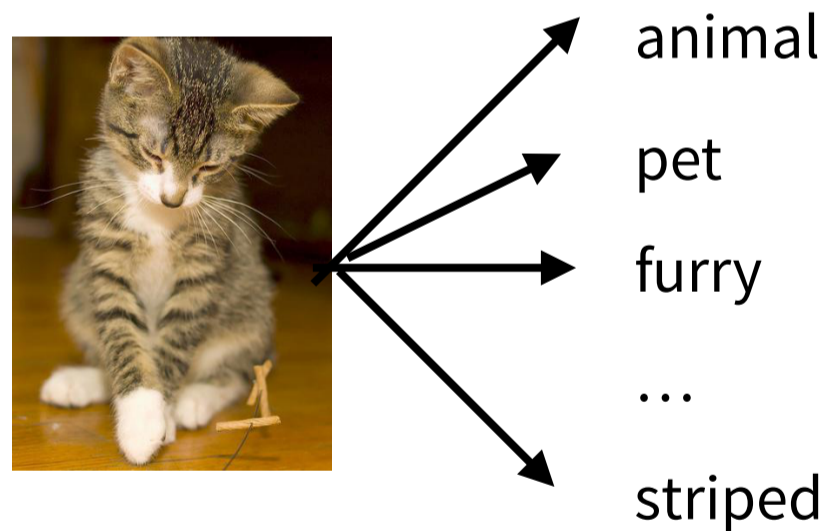
There is evidence that unsupervised learning can be achieved mainly through a level-local training signal; compare this to supervised learning where the only signal driving parameter updates is available at the output and gets backpropagated.



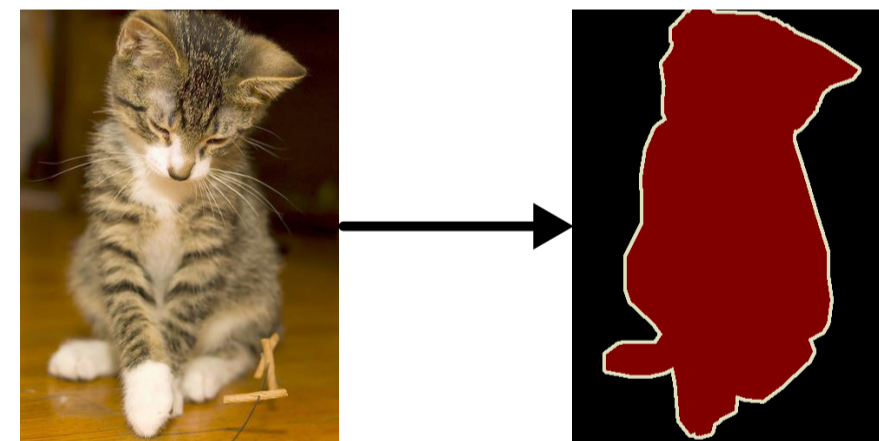
# Why Unsupervised Learning?

## Reason 5:

A recent trend in machine learning is to consider problems where the output is high-dimensional and has a complex, possibly multi-modal joint distribution. Unsupervised learning can be used in these “**structured output**” problems.



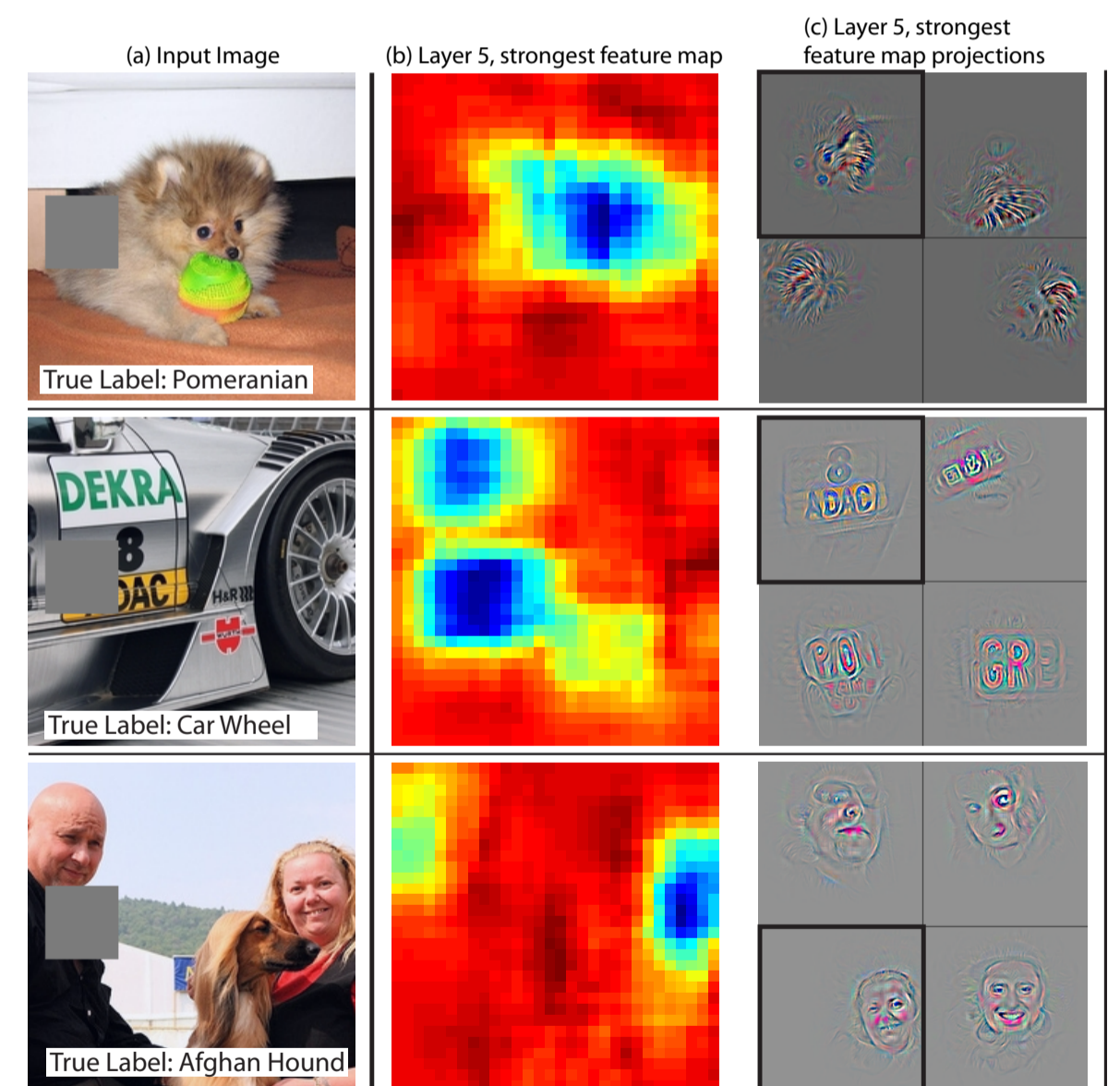
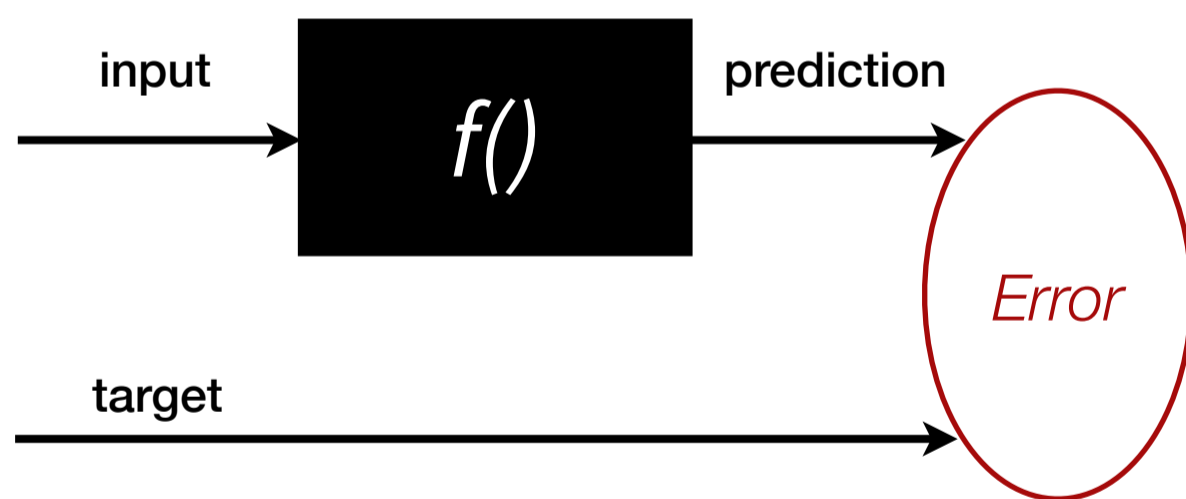
Attribute  
Prediction



Segmentation

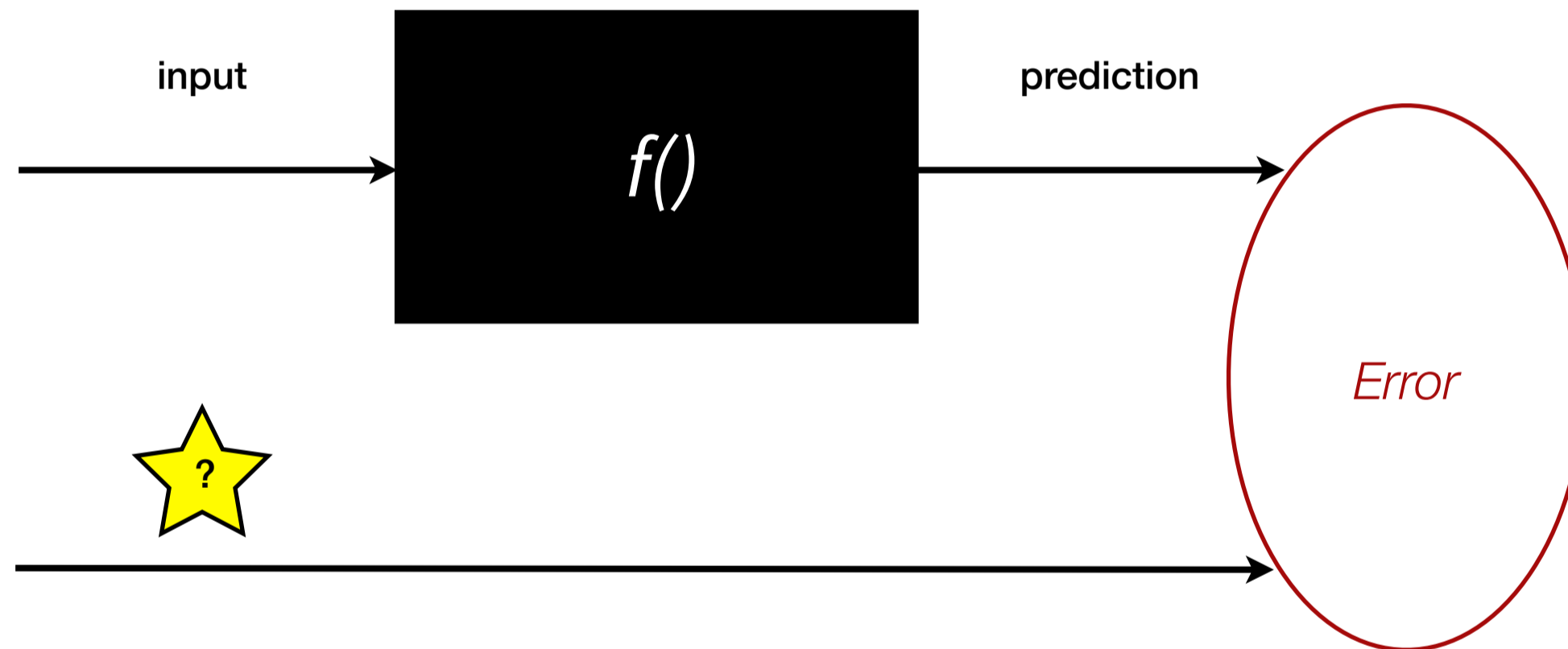
# Supervised Learning of Representations

- Learn a representation with the objective of selecting one that is best suited for predicting targets given input



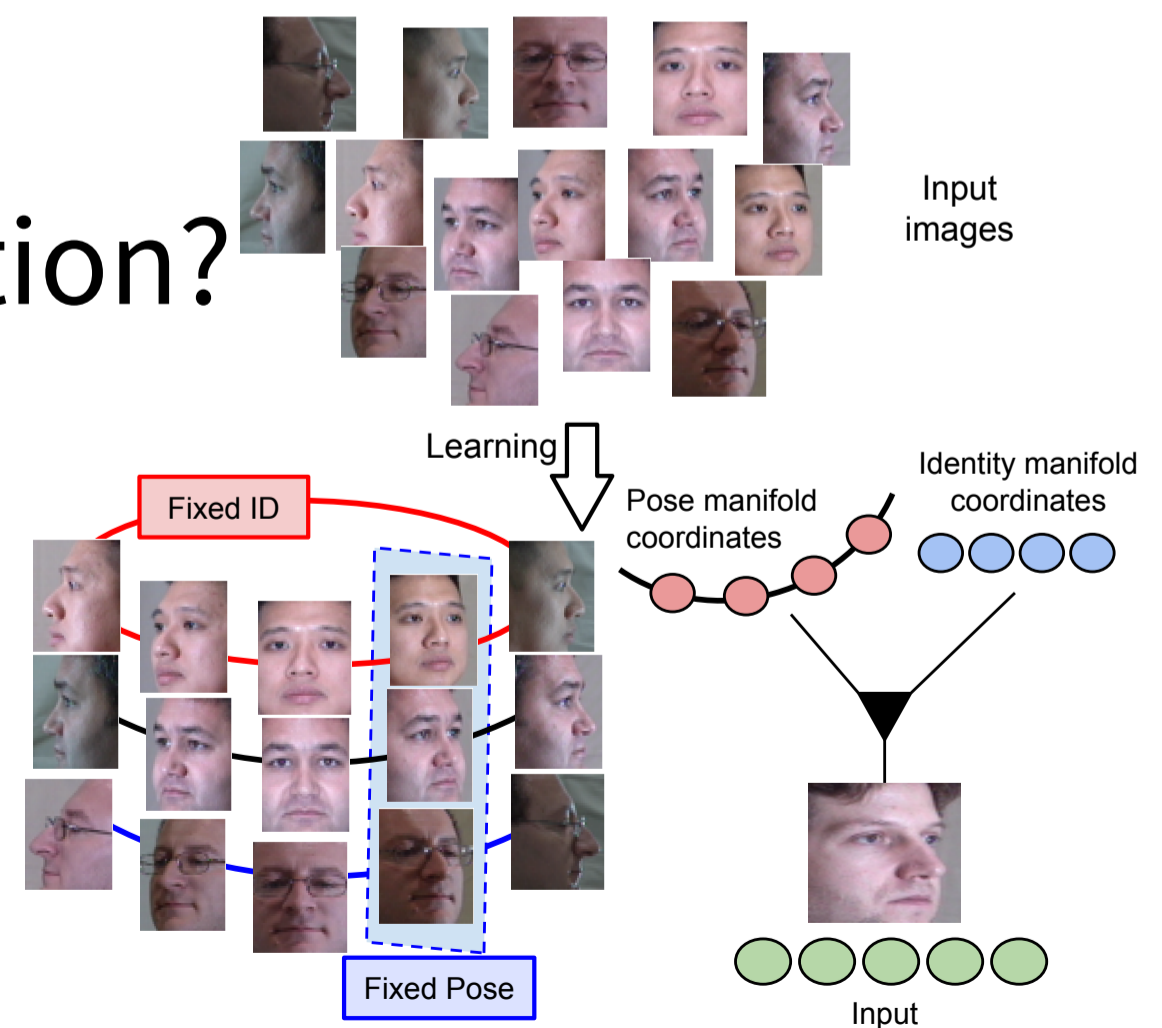
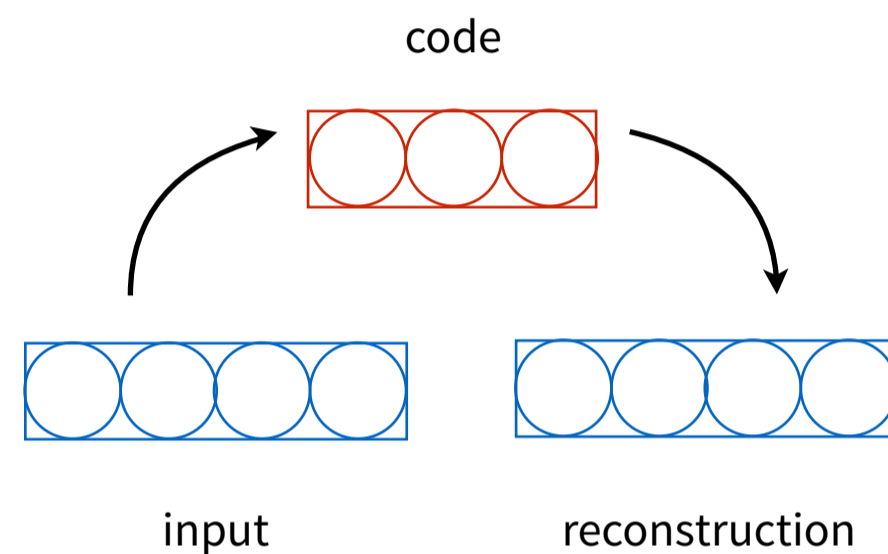


# Unsupervised Learning of Representations



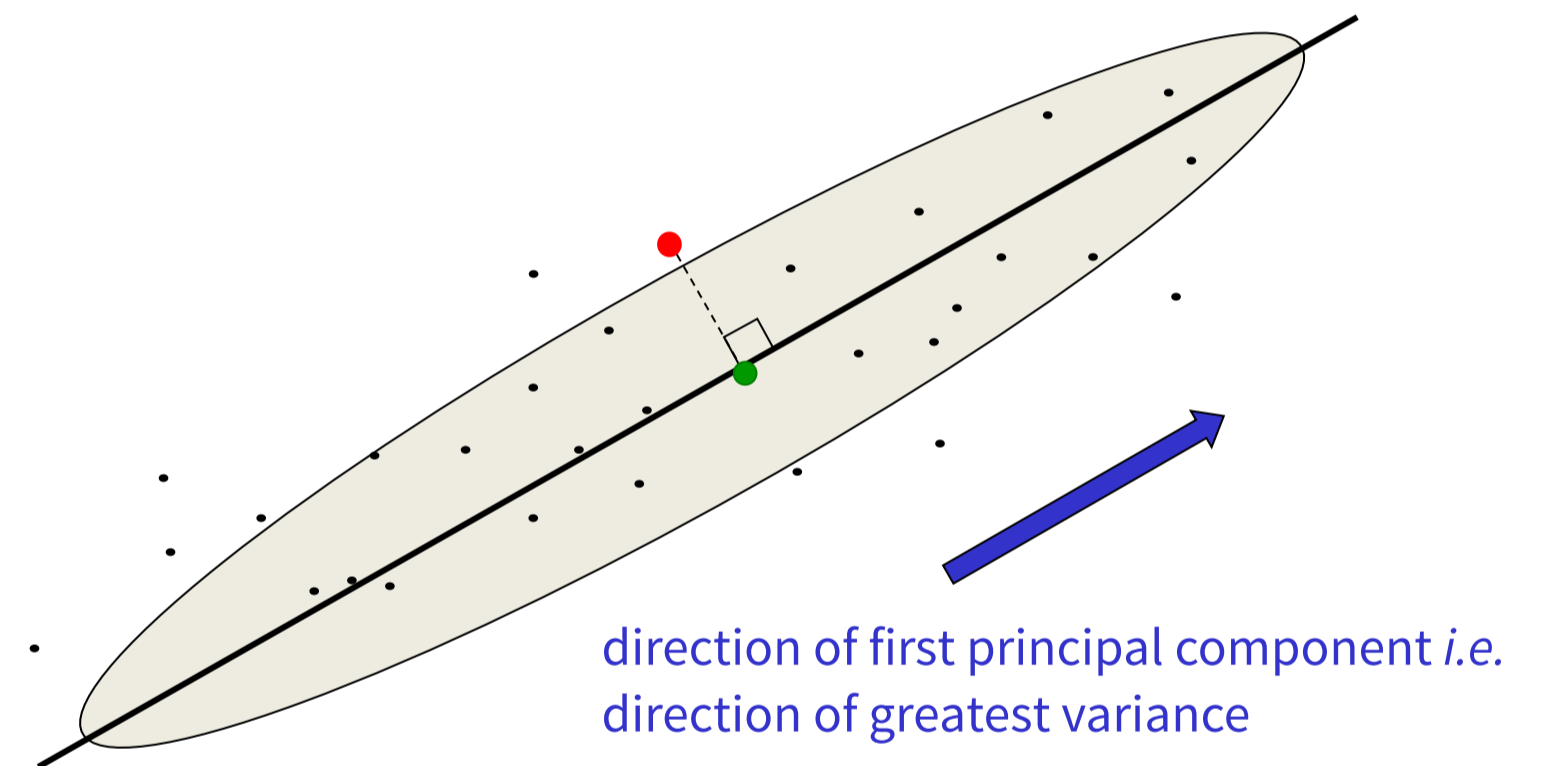
# Unsupervised learning of representations

- What is the objective?
  - reconstruction error?
  - maximum likelihood?
  - disentangle factors of variation?



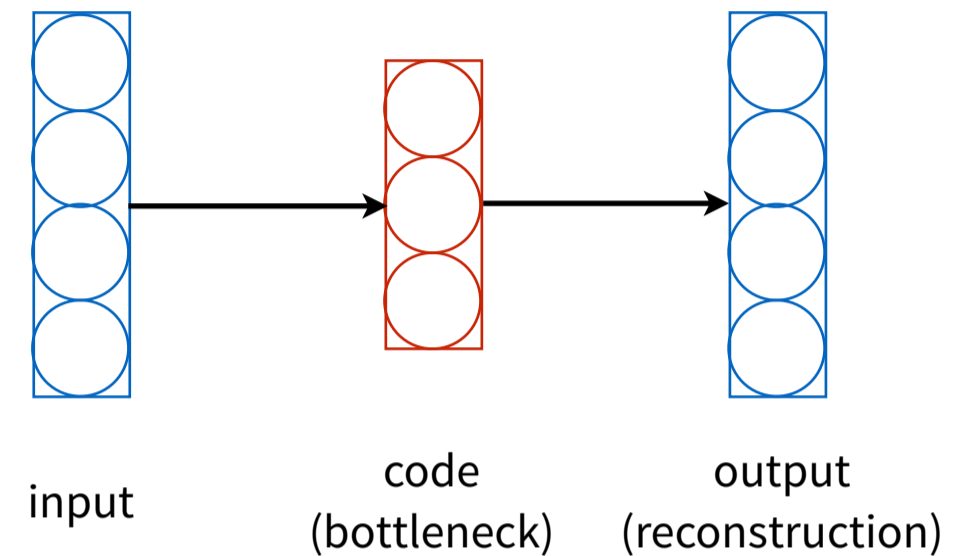
# Principal Components Analysis

- PCA works well when the data is near a linear manifold in high-dimensional space
- Project the data onto this subspace spanned by principal components
- In dimensions orthogonal to the subspace the data has low variance



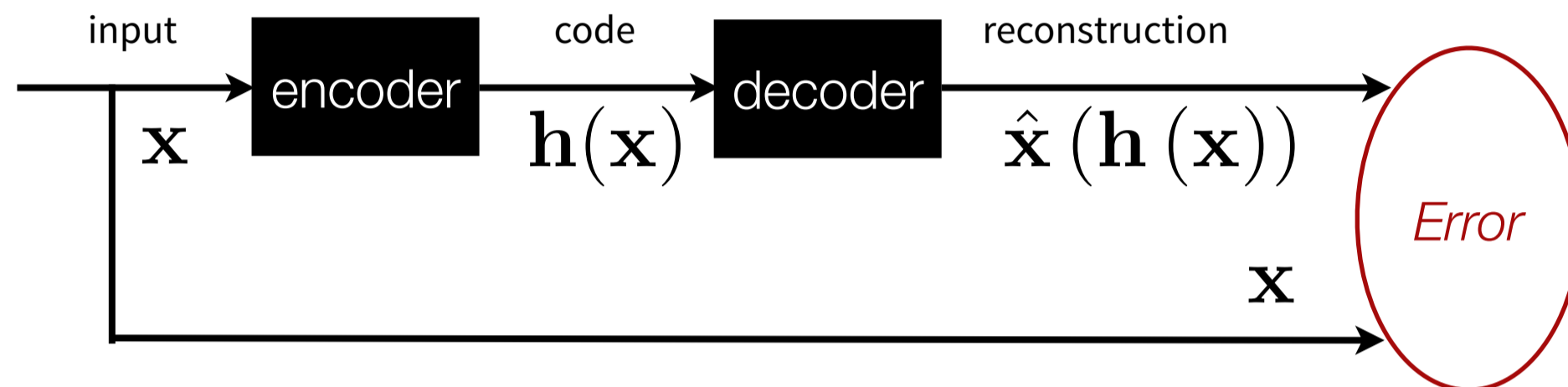
# An inefficient way to fit PCA

- Train a neural network with a “bottleneck” hidden layer
- Try to make the output the same as the input



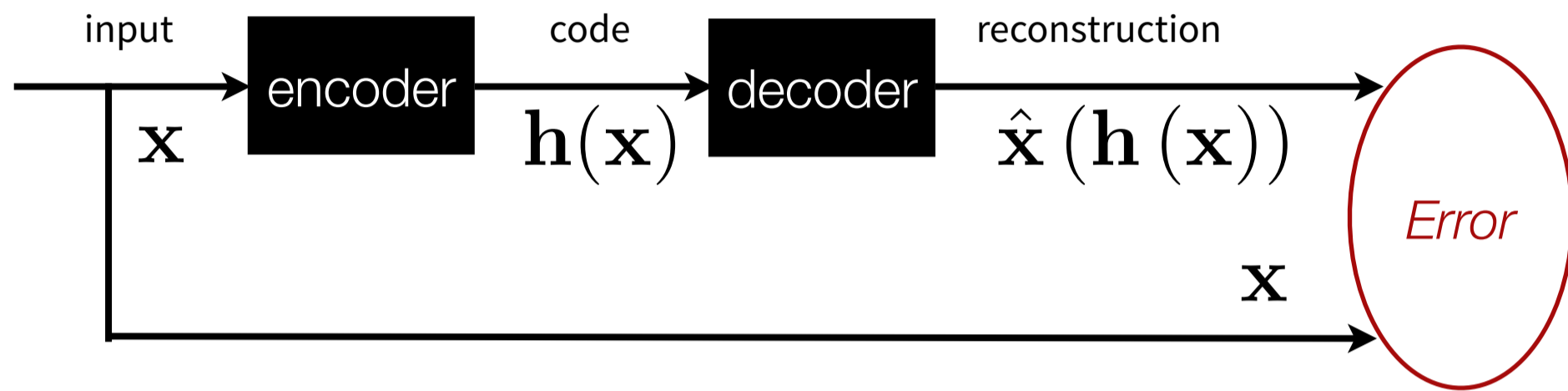
- If the hidden and output layers are linear, and we minimize squared reconstruction error:
  - The  $M$  hidden units will span the same space as the first  $M$  principal components
  - But their weight vectors will not be orthogonal
  - And they will have approximately equal variance

# Why fit PCA inefficiently?



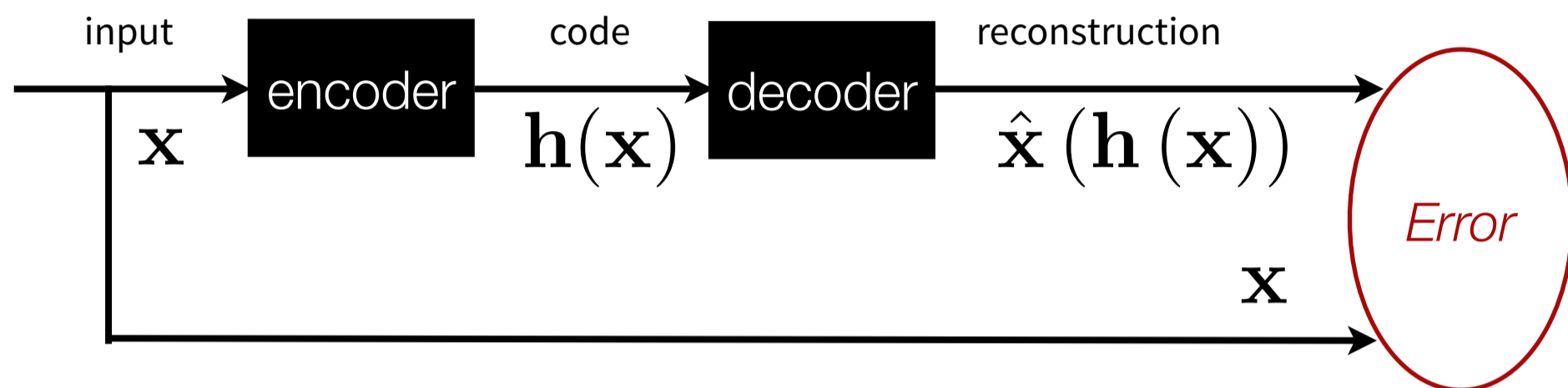
- With nonlinear layers before and after the code, it should be possible to represent data that lies on or near a nonlinear manifold
  - the encoder maps from data space to co-ordinates on the manifold
  - the decoder does the inverse transformation
- The encoder/decoder can be rich, multi-layer functions

# Auto-encoder



- Feed-forward architecture
- Trained to minimize reconstruction error
- bottleneck or regularization essential

# Auto-encoder



- Feed-forward architecture
- Trained to minimize reconstruction error
  - bottleneck or regularization essential

Example: real-valued data

Encoder

$$h_j(\mathbf{x}) = \sigma \left( \sum_i w_{ji} x_i \right)$$

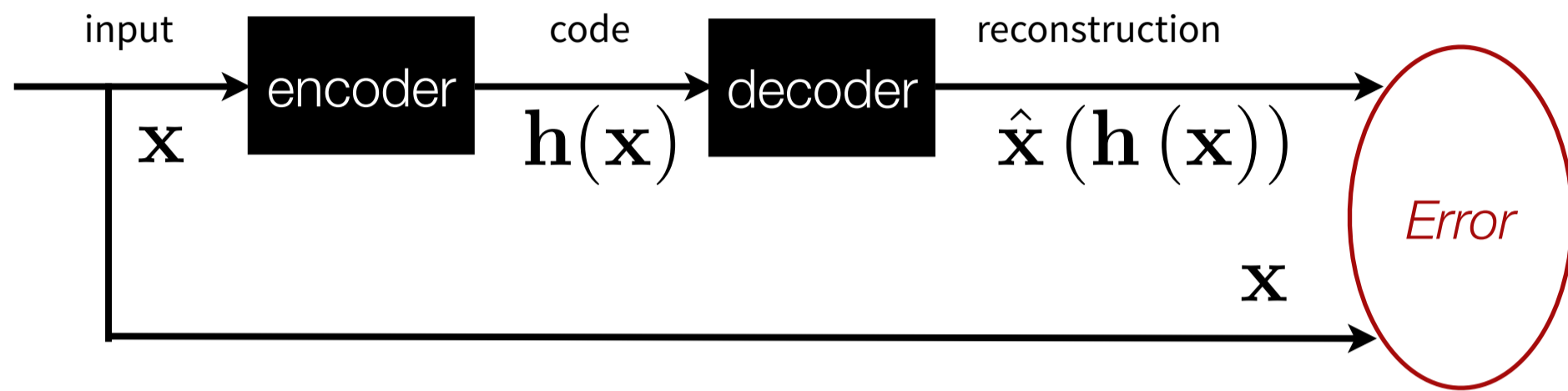
Decoder

$$\hat{x}_i(\mathbf{h}(\mathbf{x})) = \sum_j w_{ji} h_j(\mathbf{x})$$

Error

$$E = \sum_{\alpha} (\hat{\mathbf{x}}(\mathbf{h}(\mathbf{x}^{\alpha})) - \mathbf{x}^{\alpha})^2$$

# Regularized Auto-encoders



- Permit code to be higher-dimensional than the input
- Capture structure of the training distribution due to predictive opposition b/w reconstruction distribution and regularizer
- Regularizer tries to make enc/dec as simple as possible



# Simple?

# Simple?

- Reconstruct the input from the code and make the code **compact**  
(PCA, auto-encoder with bottleneck)

# Simple?

- Reconstruct the input from the code and make the code **compact**  
(PCA, auto-encoder with bottleneck)
- Reconstruct the input from the code and make the code **sparse**  
(sparse auto-encoders)

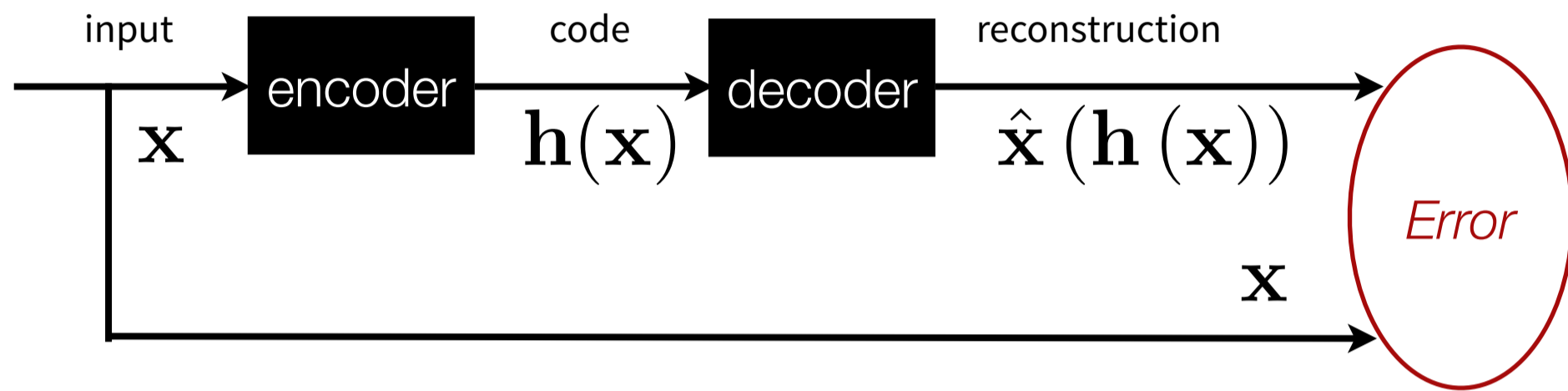
# Simple?

- Reconstruct the input from the code and make the code **compact**  
(PCA, auto-encoder with bottleneck)
- Reconstruct the input from the code and make the code **sparse**  
(sparse auto-encoders)
- **Add noise** to the input or code and reconstruct the cleaned-up version  
(denoising auto-encoders)

# Simple?

- Reconstruct the input from the code and make the code **compact**  
(PCA, auto-encoder with bottleneck)
- Reconstruct the input from the code and make the code **sparse**  
(sparse auto-encoders)
- **Add noise** to the input or code and reconstruct the cleaned-up version  
(denoising auto-encoders)
- Reconstruct the input from the code and make the code **insensitive to the input** (contractive auto-encoders)

# Sparse Auto-encoders



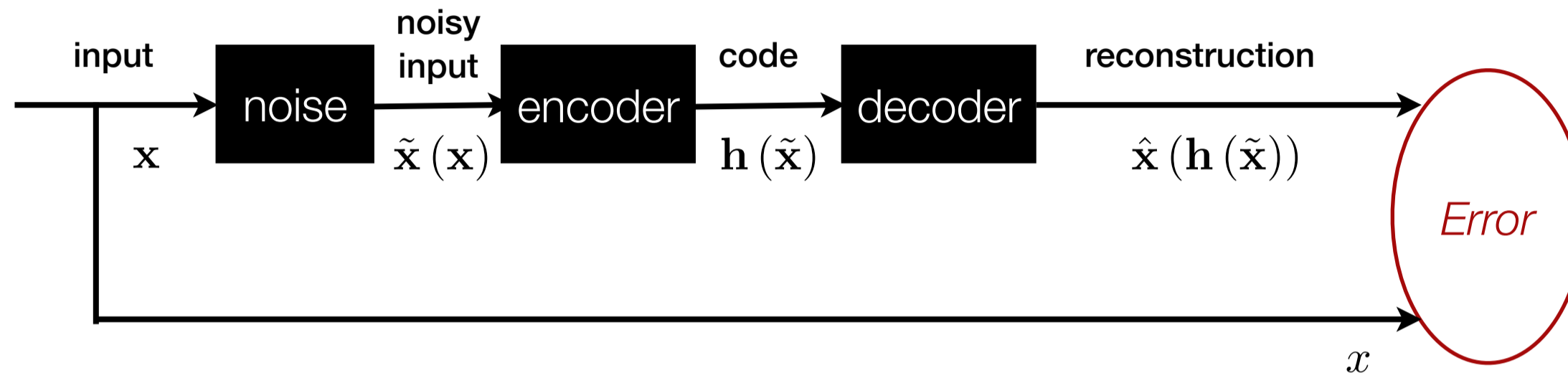
$$\mathcal{L}_{\text{SAE}} = \mathbb{E}[l(\mathbf{x}, \hat{\mathbf{x}}(\mathbf{h}(\mathbf{x})))] + \beta \sum_j \text{KL}(\rho || \hat{\rho}_j)$$

$$\hat{\rho}_j = \frac{1}{N} \sum_i^N h_j(\mathbf{x}_i) : \text{mean activation}$$

$\rho$  : target activation (small)

- Apply a sparsity penalty to the hidden activations
- Also see Predictive Sparse Decomposition (Kavukcuoglu et al. 2008)

# Denoising Auto-encoders



$$\mathcal{L}_{\text{DAE}} = \mathbb{E} [l(\mathbf{x}, \hat{\mathbf{x}}(\mathbf{h}(\tilde{\mathbf{x}})))]$$

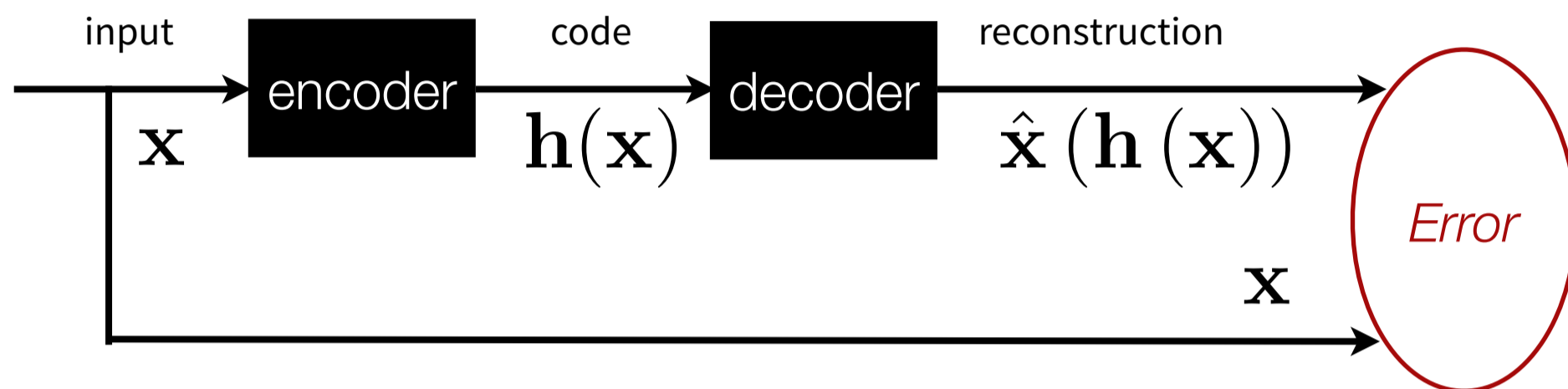
$$\tilde{\mathbf{x}}(\mathbf{x}) = \mathbf{x} + \epsilon$$

$$\epsilon \sim \mathcal{N}(\mathbf{0}, \sigma^2 I)$$

only one possible choice  
of noise model

- The code can be viewed as a lossy compression of the input
- Learning drives it to be a good compressor for training examples (and hopefully others as well) but not arbitrary inputs

# Contractive Auto-encoders



$$\mathcal{L}_{\text{CAE}} = \mathbb{E} \left[ l(\mathbf{x}, \hat{\mathbf{x}}(\mathbf{h}(\mathbf{x}))) + \lambda \left\| \frac{\partial \mathbf{h}(\mathbf{x})}{\partial \mathbf{x}} \right\|^2 \right]$$

$$\mathbf{h}(\mathbf{x}) = \text{sigmoid}(W\mathbf{x} + b)$$

$$\hat{\mathbf{x}}(\mathbf{h}(\mathbf{x})) = \text{sigmoid}(W^T\mathbf{h} + c)$$

- Learn good models of high-dimensional data (Bengio et al. 2013)
- Can obtain good representations for classification
- Can produce good quality samples by a random walk near the manifold of high density (Rifai et al. 2012)



# What do Denoising Auto-encoders Learn?

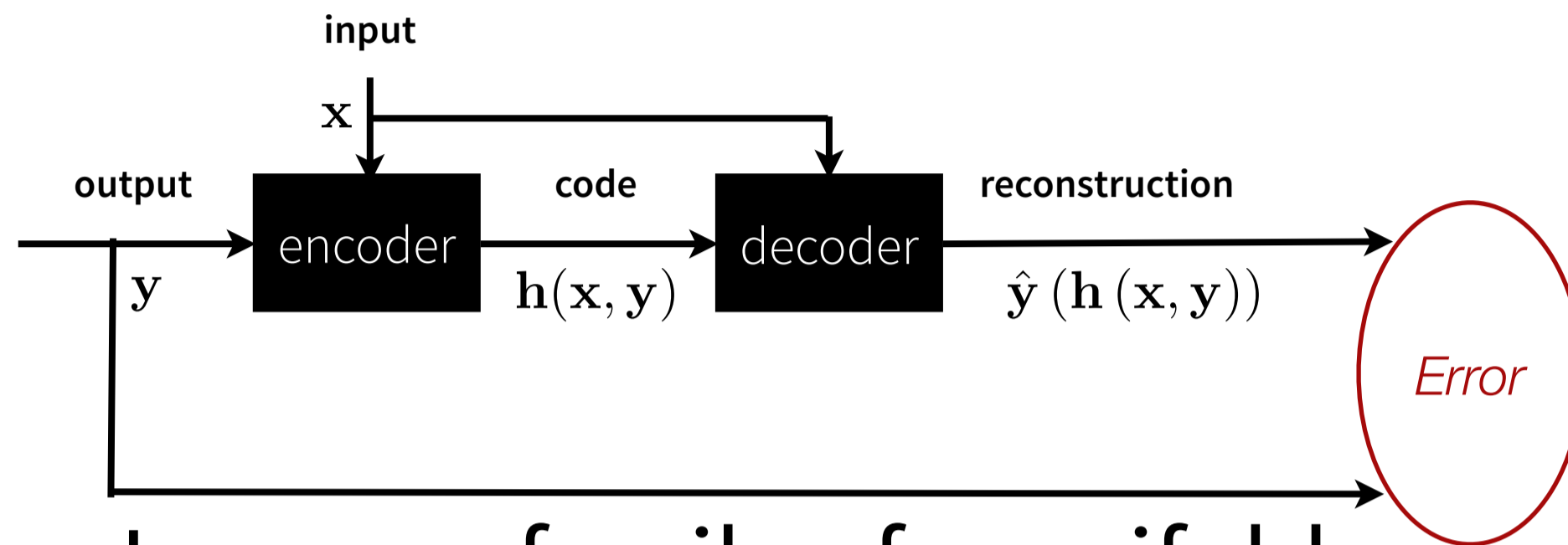
# What do Denoising Auto-encoders Learn?

- The reconstruction function locally characterizes the data generating density (Alain and Bengio 2013)
  - derivative of the log-density (score) with respect to the input
  - second derivative of the density
  - other local properties

# What do Denoising Auto-encoders Learn?

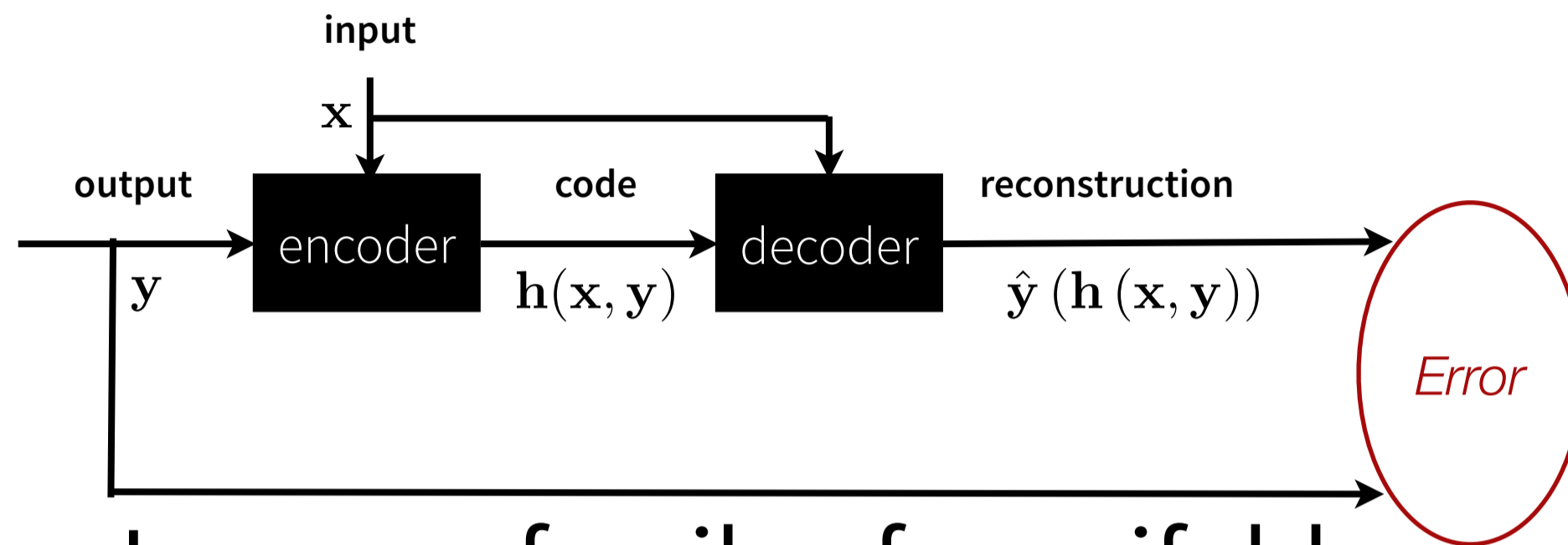
- The reconstruction function locally characterizes the data generating density (Alain and Bengio 2013)
  - derivative of the log-density (score) with respect to the input
  - second derivative of the density
  - other local properties
- Bengio et al. (2013) generalized this result to arbitrary variables (discrete, continuous, or both), arbitrary corruption, arbitrary loss function

# Relational Autoencoders



- Learns a family of manifolds (Memisevic 2011)
- Can be viewed as AE whose weights are modulated by input vector
$$w_{kj}(\mathbf{x}) = \sum_i \hat{w}_{kj}^i x_i$$
- Used for modelling image transformations, extracting spatio-temporal features

# Relational Autoencoders



Example: real-valued data

Encoder

$$h_k(\mathbf{x}; \mathbf{y}) = \sigma \left( \sum_{ij} \hat{w}_{kj}^i x_i y_j \right)$$

Decoder

$$\hat{y}_j(h(\mathbf{x}; \mathbf{y})) = \sum_{ki} \hat{w}_{kj}^i x_i h_k(\mathbf{x}; \mathbf{y})$$

- Learns a family of manifolds (Memisevic 2011)

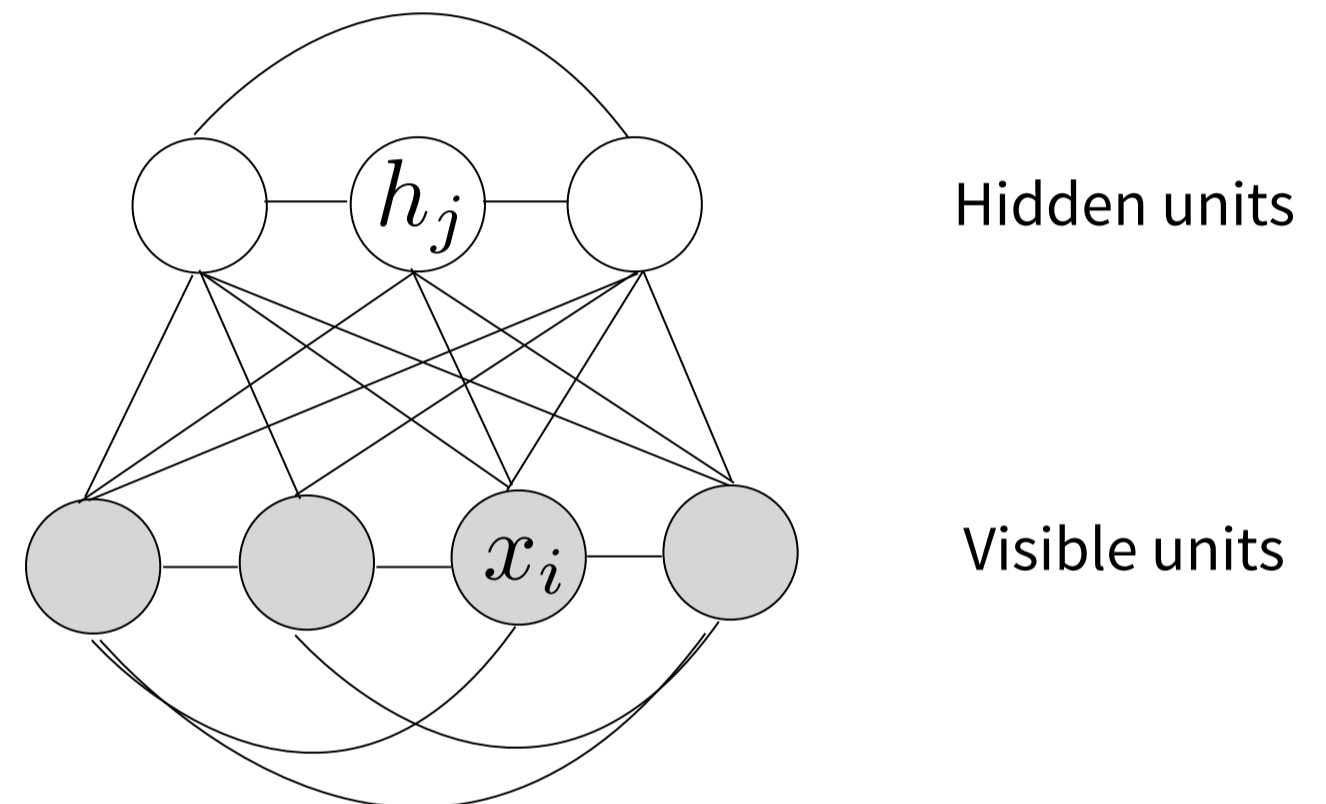
- Can be viewed as AE whose weights are modulated by input vector

$$w_{kj}(\mathbf{x}) = \sum_i \hat{w}_{kj}^i x_i$$

- Used for modelling image transformations, extracting spatio-temporal features

# Boltzmann Machines

- Stochastic Hopfield Networks with hidden units
- Both visible and hidden units are **binary**
- **Energy-based** model
- Needs MCMC to sample from the posterior; this makes inference and learning extremely slow

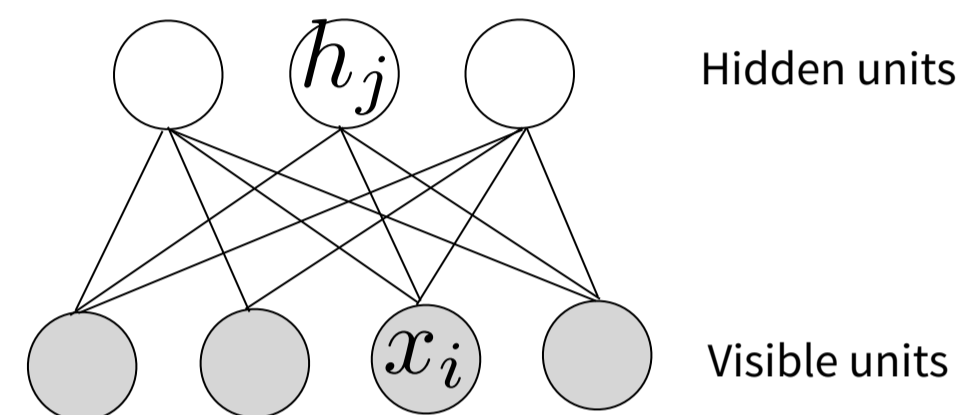


$$p(h_k = 1 | \mathbf{x}, \{h_l\} \forall l \neq k) = \frac{1}{1 + e^{-\Delta E_k}}$$

$$\Delta E_k = E(h_k = 0) - E(h_k = 1) = b_k + \sum_i x_i w_{ik} + \sum_l h_l w_{kl}$$

# Restricted Boltzmann Machines

- We restrict the connectivity to make inference and learning easier.
  - Only one layer of hidden units.
  - No connections between hidden units.
- In an RBM it only takes one step to reach thermal equilibrium when the visible units are clamped
  - So we can quickly get the exact value of  $\langle x_i h_j \rangle_{\mathbf{x}}$



$$p(h_j = 1 | \mathbf{x}) = \frac{1}{1 + e^{-(b_j + \sum_{i \in \text{vis}} x_i w_{ij})}}$$

# Learning in RBMs

- Goal: maximize the product of the probabilities that the RBM assigns to the binary vectors in the training set
- Everything that one weight needs to know about the other weights and the data is contained in the difference of two correlations

$$\frac{\partial \log p(\mathbf{x})}{\partial w_{ij}} = \langle x_i h_j \rangle_{\mathbf{x}} - \langle x_i h_j \rangle_{\text{model}}$$

$$\Delta w_{ij} \propto \langle x_i h_j \rangle_{\mathbf{x}} - \langle x_i h_j \rangle_{\text{model}}$$



# Learning in RBMs

- Goal: maximize the product of the probabilities that the RBM assigns to the binary vectors in the training set
- Everything that one weight needs to know about the other weights and the data is contained in the difference of two correlations

$$\frac{\partial \log p(\mathbf{x})}{\partial w_{ij}} = \langle x_i h_j \rangle_{\mathbf{x}} - \langle x_i h_j \rangle_{\text{model}}$$

Derivative of log prob of  
one training vector,  $\mathbf{x}$ ,  
under the model

$$\Delta w_{ij} \propto \langle x_i h_j \rangle_{\mathbf{x}} - \langle x_i h_j \rangle_{\text{model}}$$

# Learning in RBMs

- Goal: maximize the product of the probabilities that the RBM assigns to the binary vectors in the training set
- Everything that one weight needs to know about the other weights and the data is contained in the difference of two correlations

$$\frac{\partial \log p(\mathbf{x})}{\partial w_{ij}} = \langle x_i h_j \rangle_{\mathbf{x}} - \langle x_i h_j \rangle_{\text{model}}$$

Derivative of log prob of one training vector,  $\mathbf{x}$ , under the model

Expected value of product of states at thermal equilibrium when  $\mathbf{x}$  is clamped on the visible units (positive phase)

$$\Delta w_{ij} \propto \langle x_i h_j \rangle_{\mathbf{x}} - \langle x_i h_j \rangle_{\text{model}}$$

# Learning in RBMs

- Goal: maximize the product of the probabilities that the RBM assigns to the binary vectors in the training set
- Everything that one weight needs to know about the other weights and the data is contained in the difference of two correlations

$$\frac{\partial \log p(\mathbf{x})}{\partial w_{ij}} = \langle x_i h_j \rangle_{\mathbf{x}} - \langle x_i h_j \rangle_{\text{model}}$$

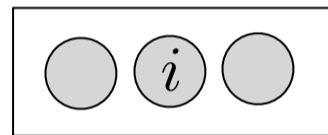
Derivative of log prob of one training vector,  $\mathbf{x}$ , under the model

Expected value of product of states at thermal equilibrium when  $\mathbf{x}$  is clamped on the visible units (positive phase)

Expected value of product of states at thermal equilibrium with no clamping (negative phase)

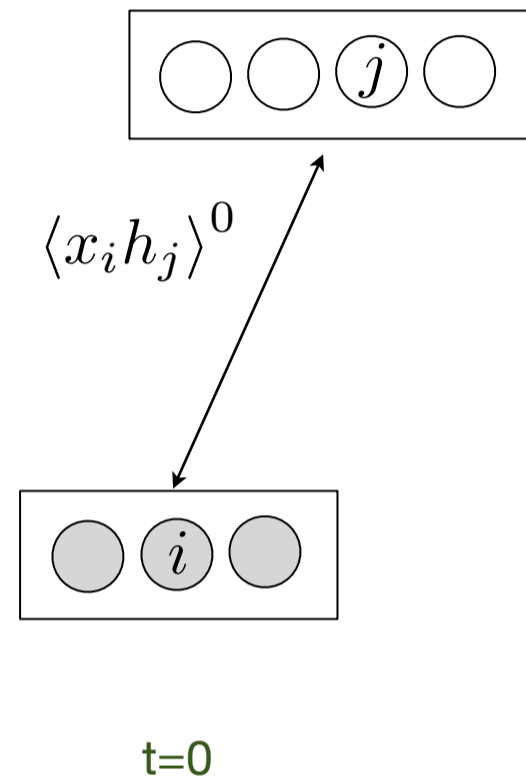
$$\Delta w_{ij} \propto \langle x_i h_j \rangle_{\mathbf{x}} - \langle x_i h_j \rangle_{\text{model}}$$

# The Boltzmann Machine Learning Algorithm - RBMs

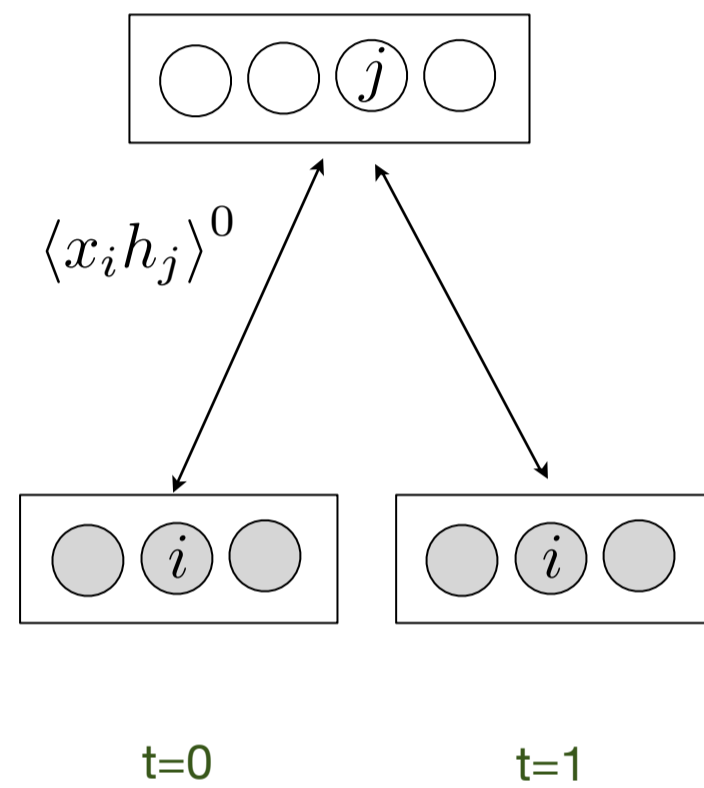


$t=0$

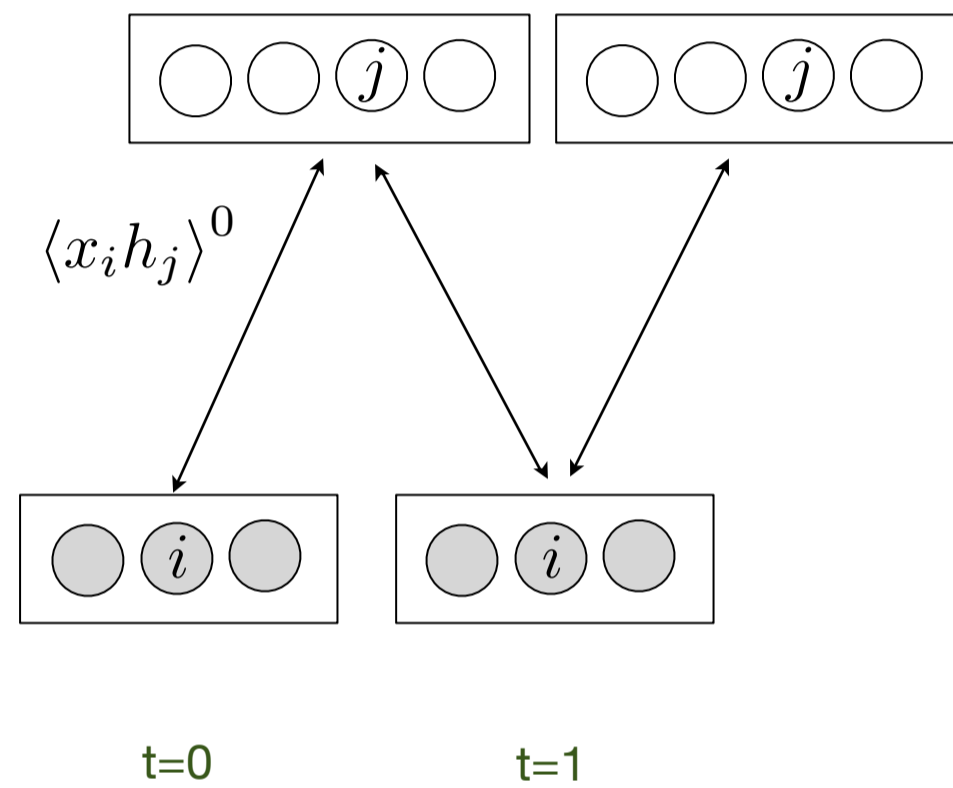
# The Boltzmann Machine Learning Algorithm - RBMs



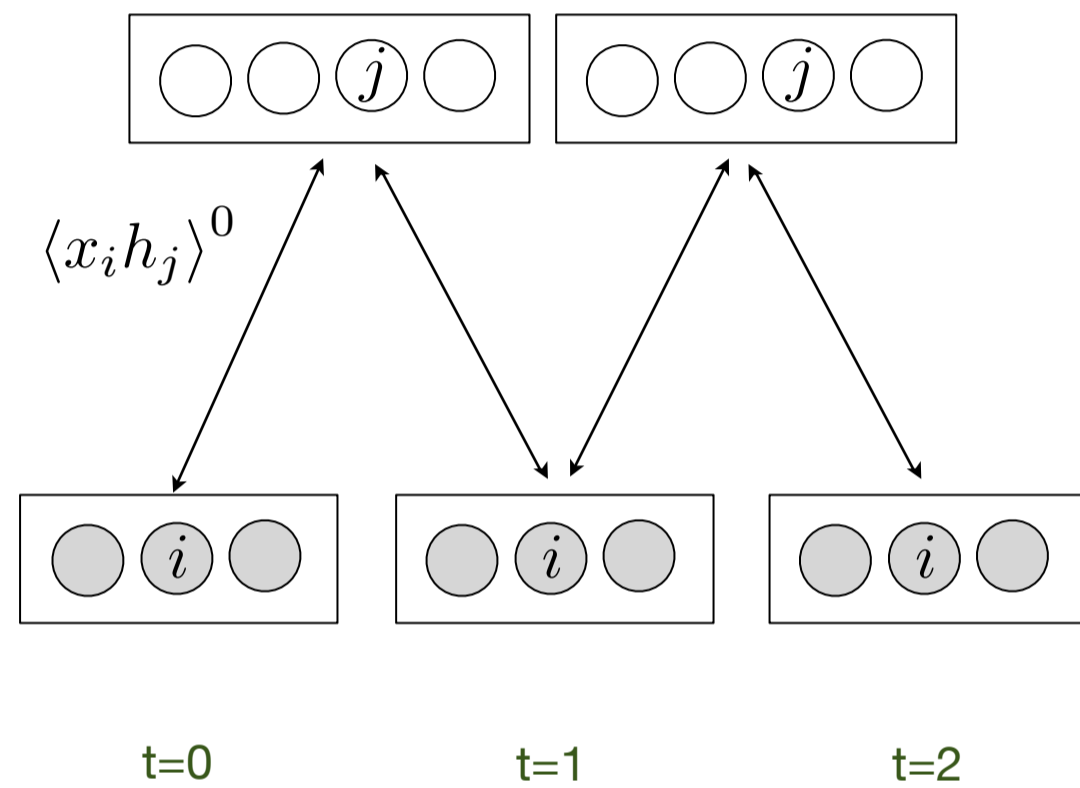
# The Boltzmann Machine Learning Algorithm - RBMs



# The Boltzmann Machine Learning Algorithm - RBMs

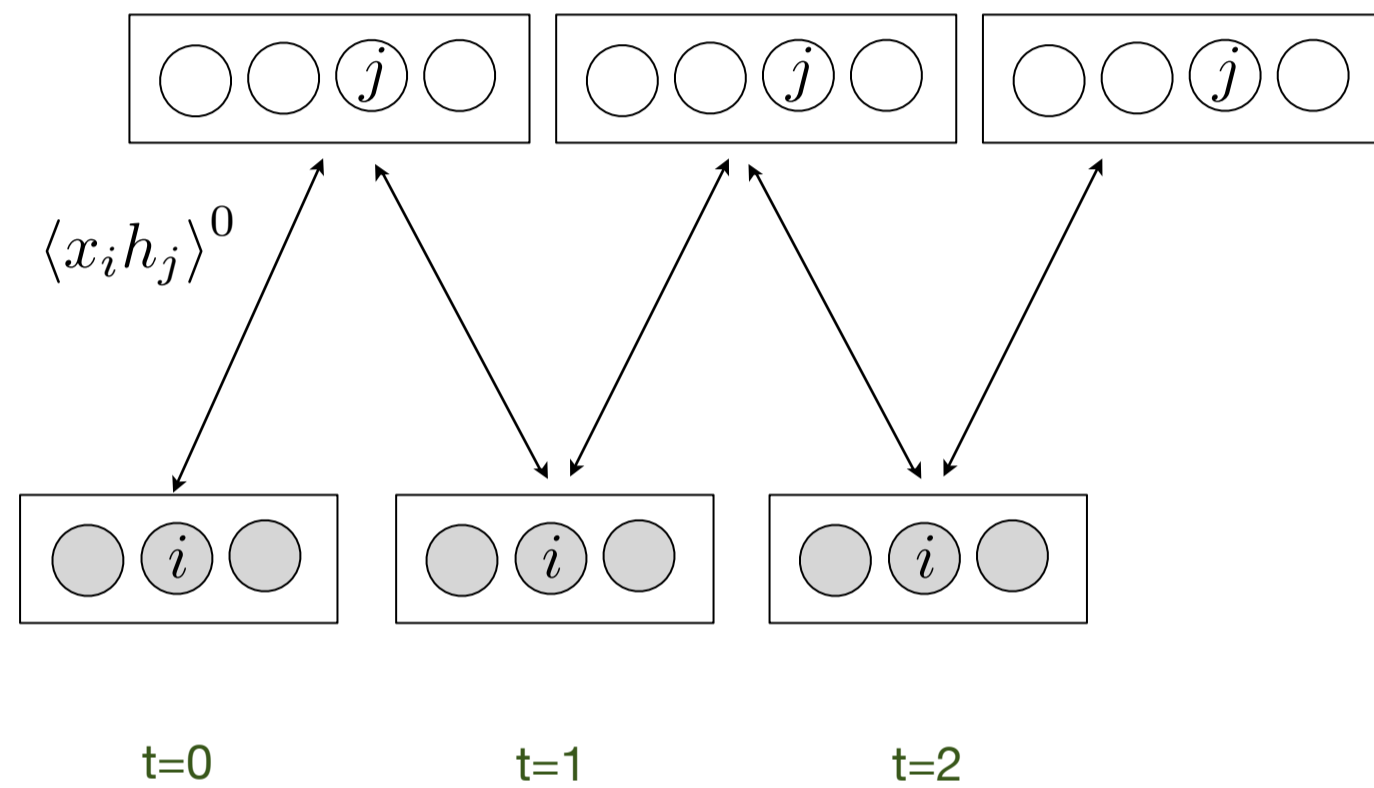


# The Boltzmann Machine Learning Algorithm - RBMs

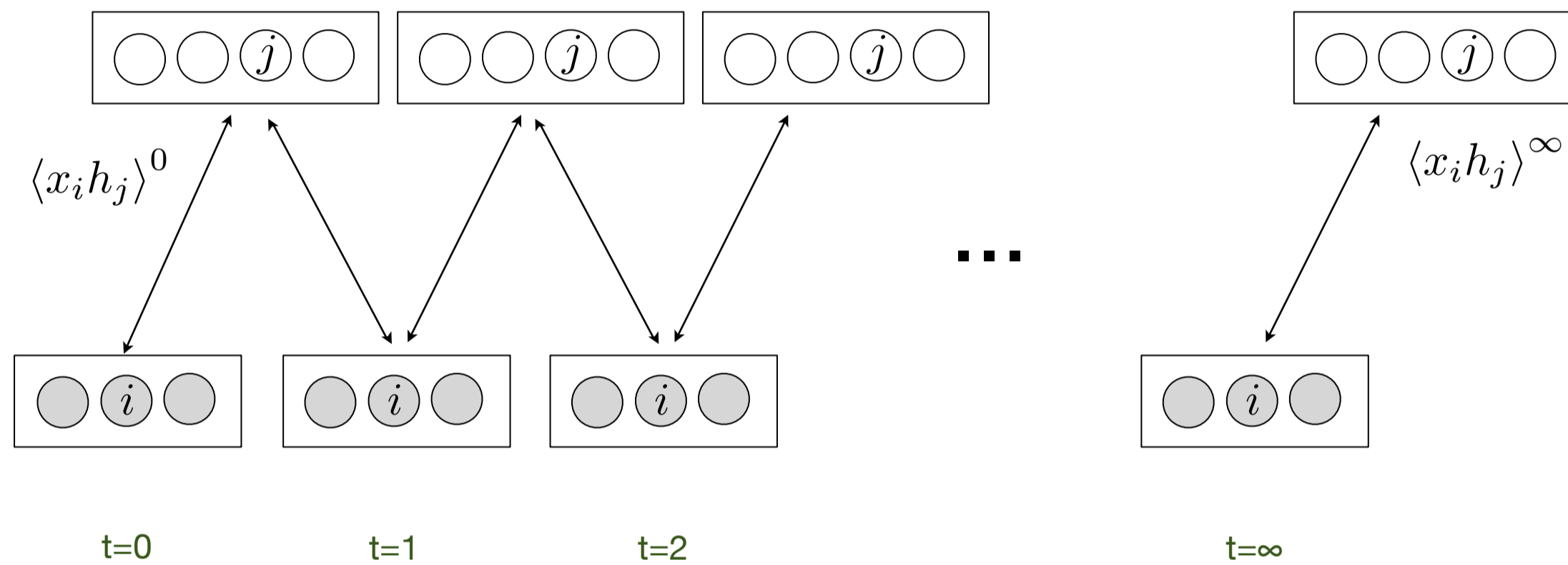




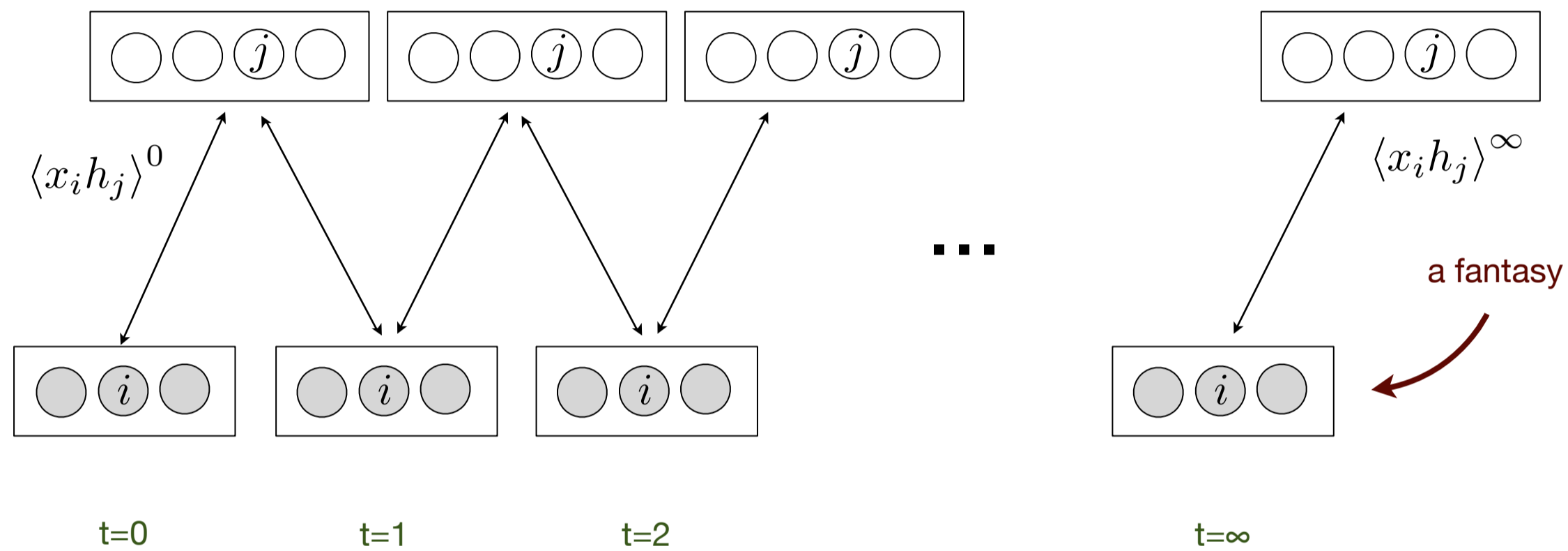
# The Boltzmann Machine Learning Algorithm - RBMs



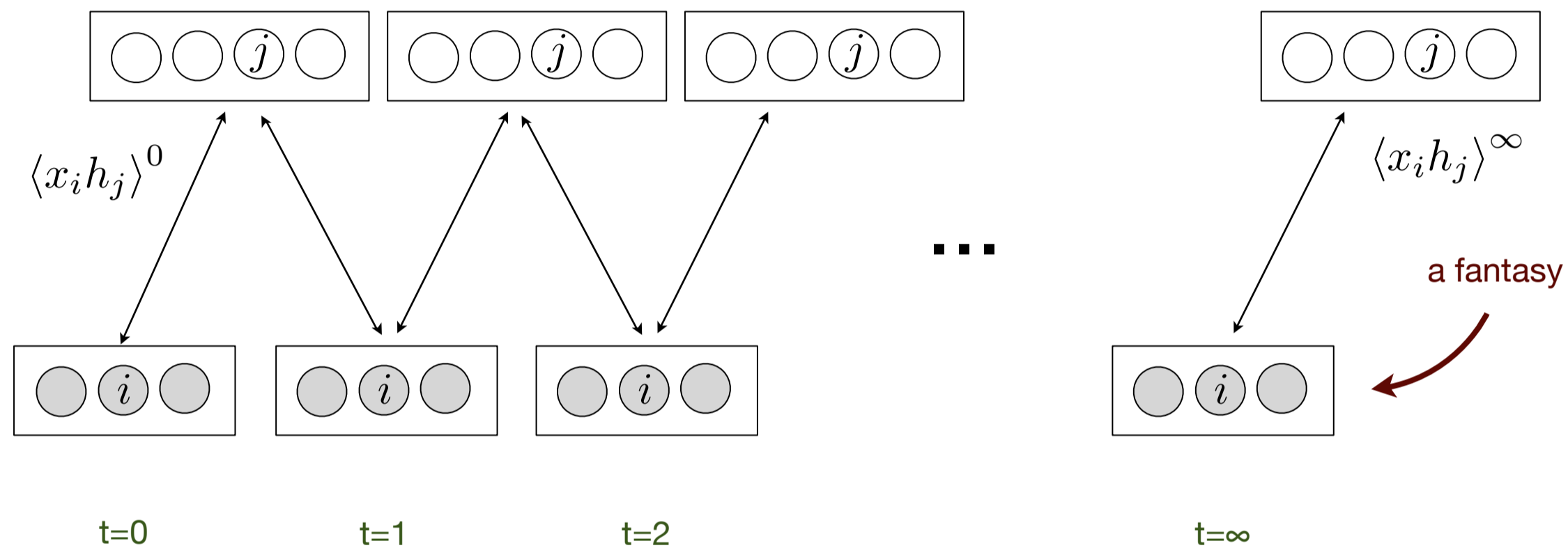
# The Boltzmann Machine Learning Algorithm - RBMs



# The Boltzmann Machine Learning Algorithm - RBMs



# The Boltzmann Machine Learning Algorithm - RBMs



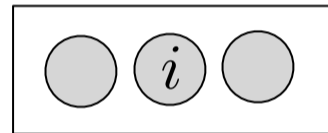
$$\Delta w_{ij} = \epsilon \left( \langle x_i h_j \rangle^0 - \langle x_i h_j \rangle^\infty \right)$$

$$\langle x_i h_j \rangle^0 = \langle x_i h_j \rangle_{\mathbf{x}}$$

$$\langle x_i h_j \rangle^\infty = \langle x_i h_j \rangle_{\text{model}}$$

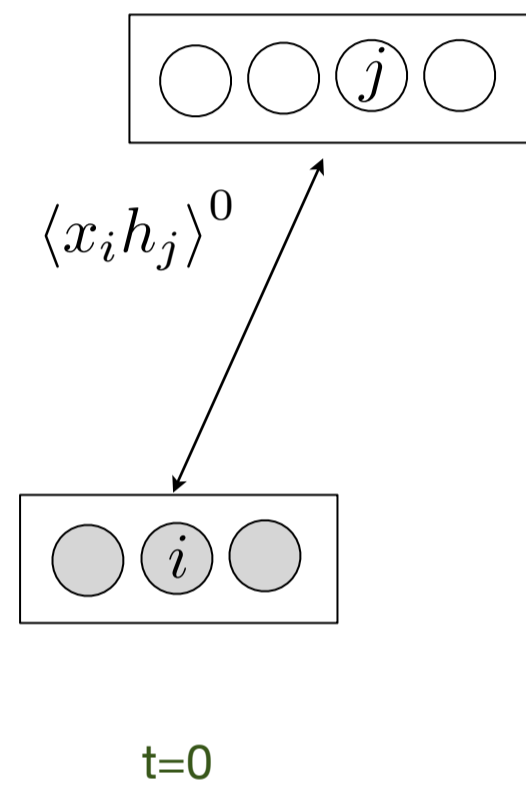
# Contrastive Divergence

Instead of running the Markov chain to equilibrium, run for just one (or a few) steps!



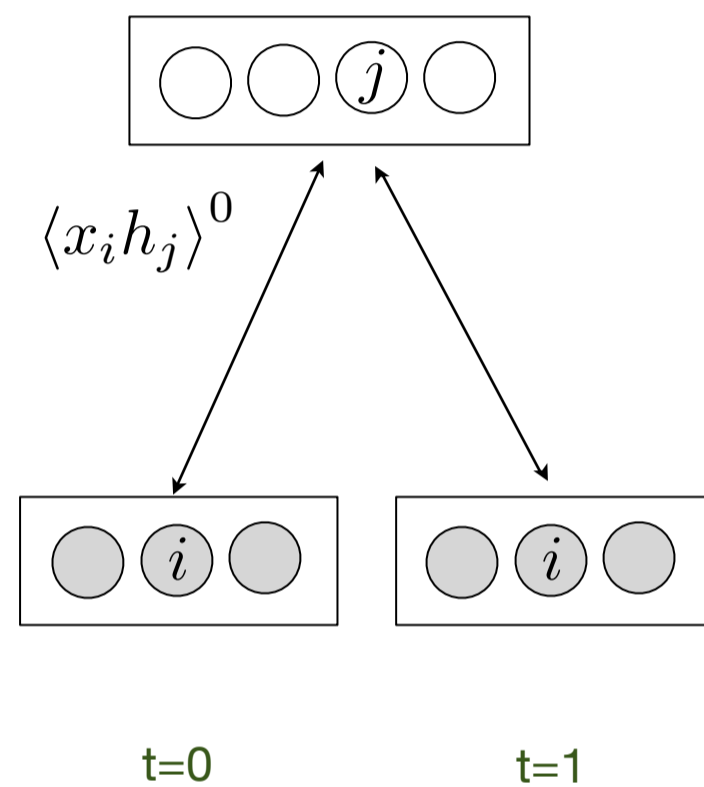
t=0

# Contrastive Divergence



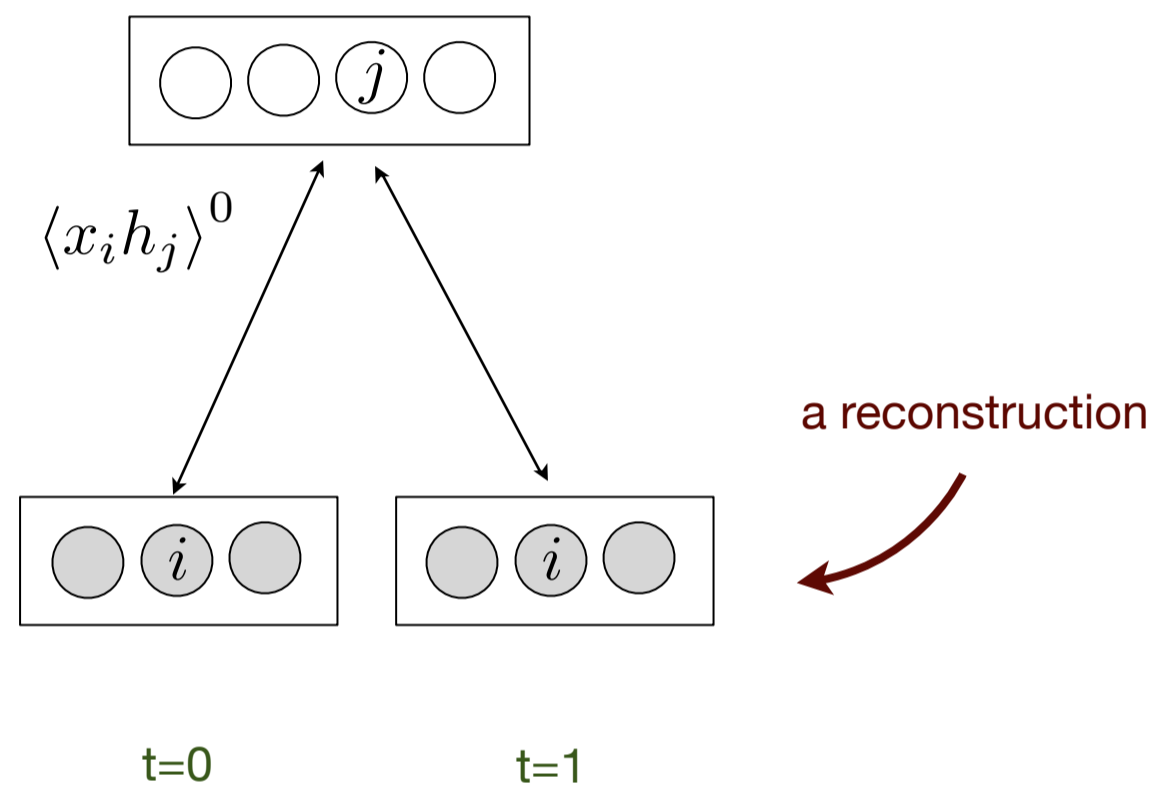
Instead of running the Markov chain to equilibrium, run for just one (or a few) steps!

# Contrastive Divergence



Instead of running the Markov chain to equilibrium, run for just one (or a few) steps!

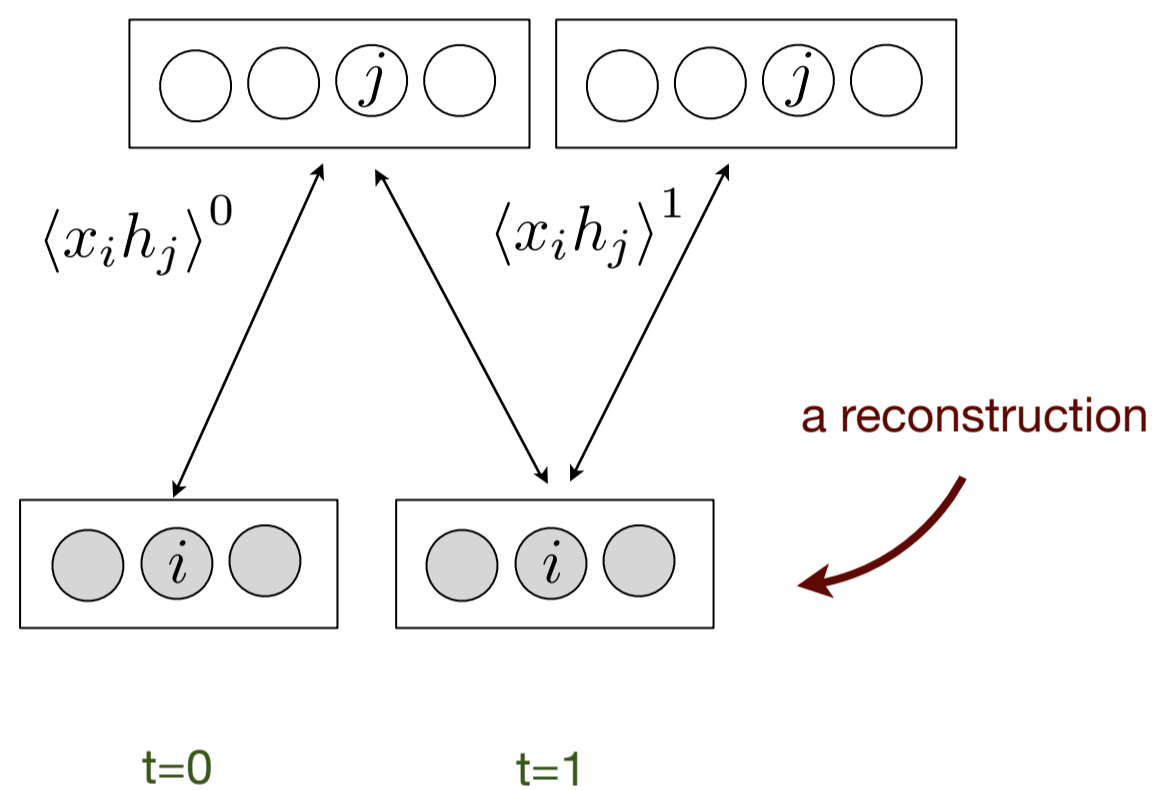
# Contrastive Divergence



Instead of running the Markov chain to equilibrium, run for just one (or a few) steps!

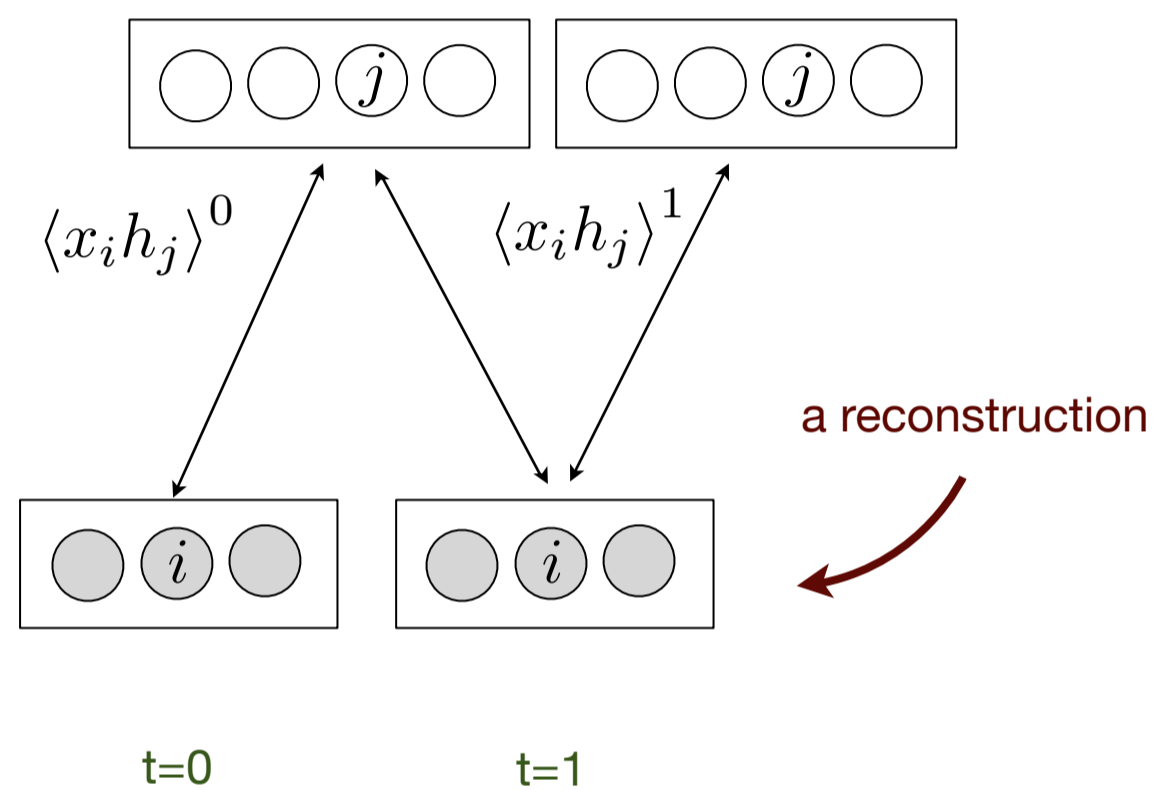


# Contrastive Divergence



Instead of running the Markov chain to equilibrium, run for just one (or a few) steps!

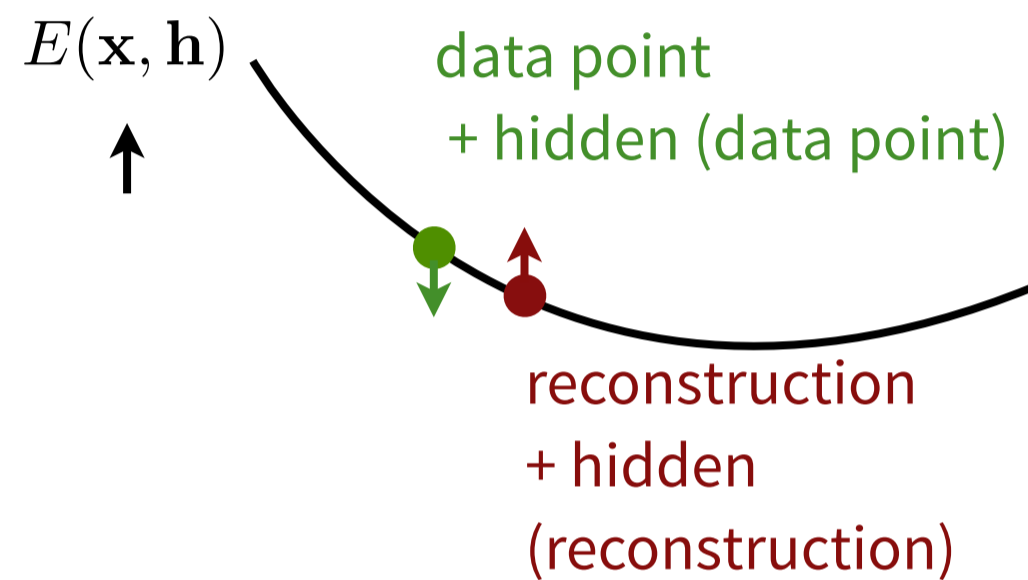
# Contrastive Divergence



Instead of running the Markov chain to equilibrium, run for just one (or a few) steps!

$$\Delta w_{ij} = \epsilon \left( \langle x_i h_j \rangle^0 - \langle x_i h_j \rangle^1 \right)$$

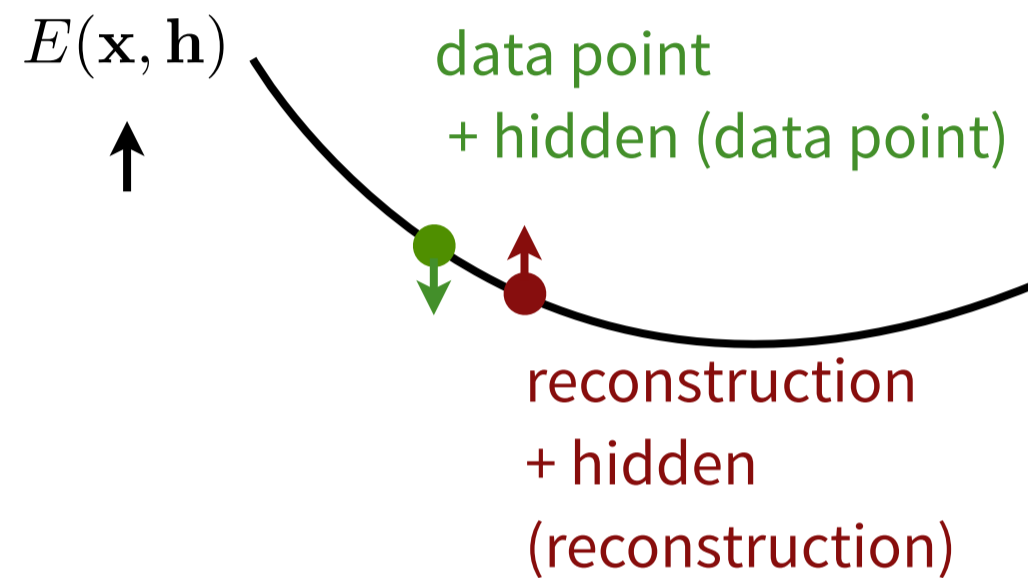
# Contrastive Divergence (A picture)



Change the weights to pull the energy down at the data point

Change the weights to pull the energy up at the reconstruction

# Contrastive Divergence (A picture)



Change the weights to pull the energy down at the data point

Change the weights to pull the energy up at the reconstruction

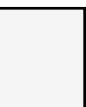
# Alternatives to CD

- Persistent CD a.k.a. Stochastic Maximum Likelihood (Tieleman 2008)
  - don't reset the Markov chain at the data for every point
- Score Matching/Ratio Matching (Hyvarinen 2005, 2007)
  - minimize the expected distance b/w model and data “score function”
- Minimum Probability Flow (Sohl-Dickstein et al. 2011)
  - establish dynamics that would transform the observed data distribution into the model distribution
  - minimize the KL divergence b/w the data distribution and the distribution produced by running the dynamics for an infinitesimal time

For a comparison, see *Inductive Principles for Restricted Boltzmann Machine Learning*, Marlin et al. 2010

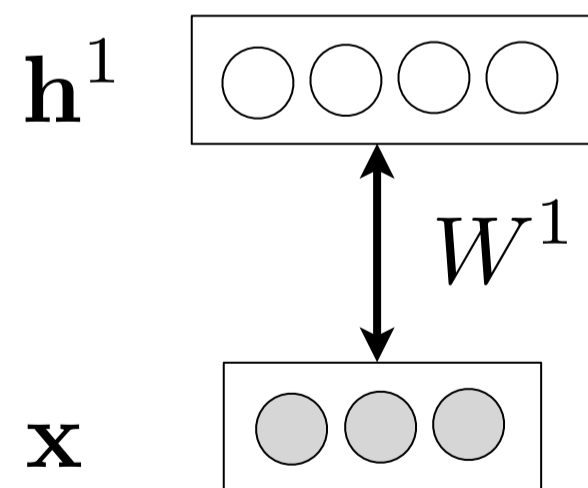
# Stacking to Build Deep Models

- Greedy layer-wise training can be used to build deep models
- It is most popular to use RBMs, but other architectures (regularized autoencoders, ICA, even k-means) can be stacked



# Stacking RBMs: Procedure

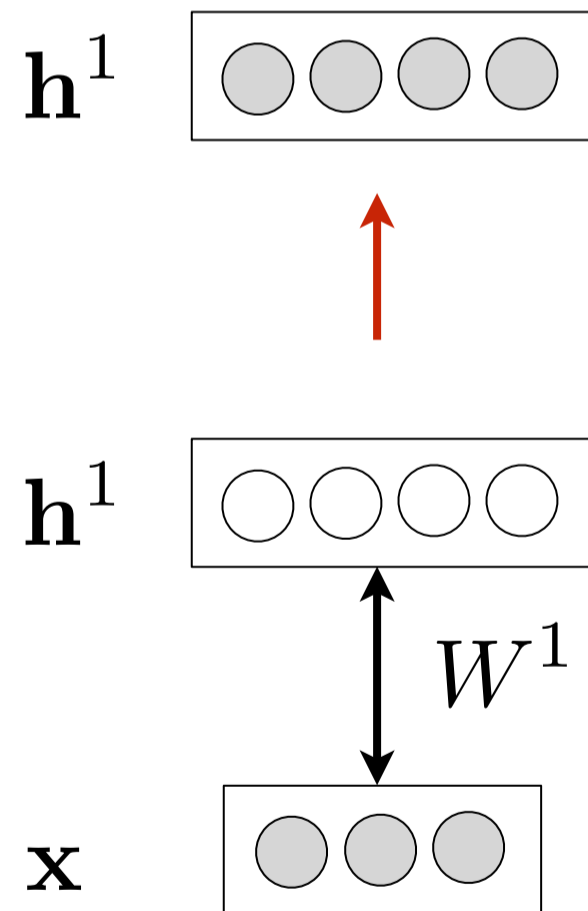
# Stacking RBMs: Procedure



① Train an RBM



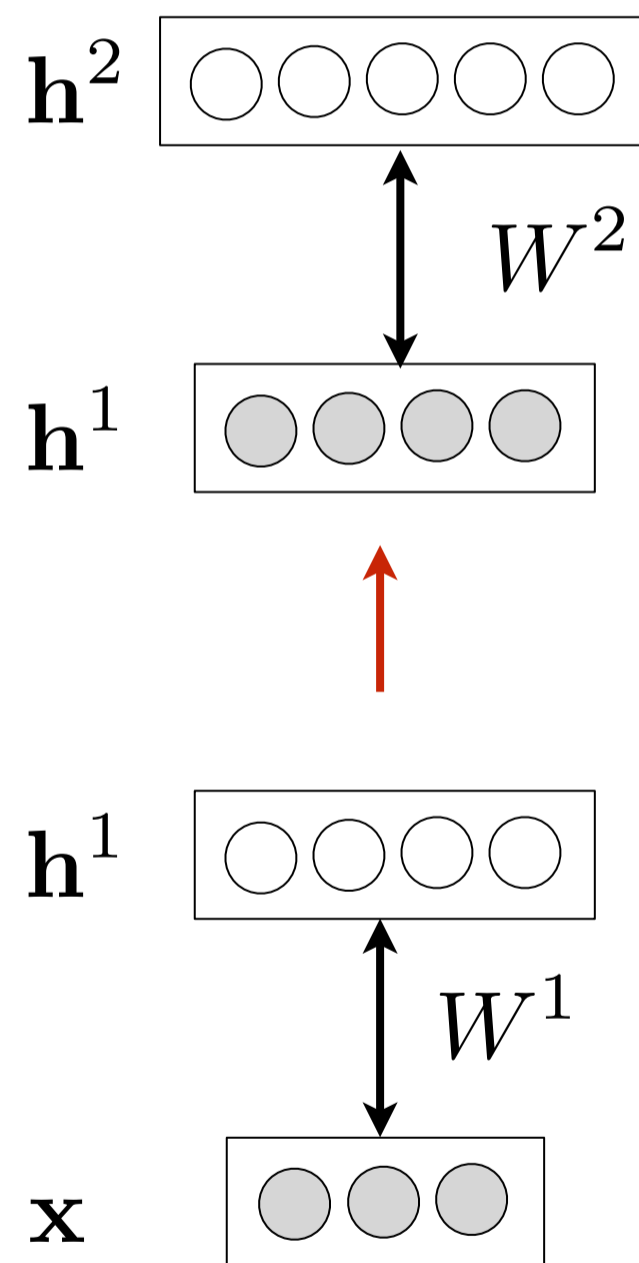
# Stacking RBMs: Procedure



② Run your data through the model to generate a dataset of hidden activations

① Train an RBM

# Stacking RBMs: Procedure



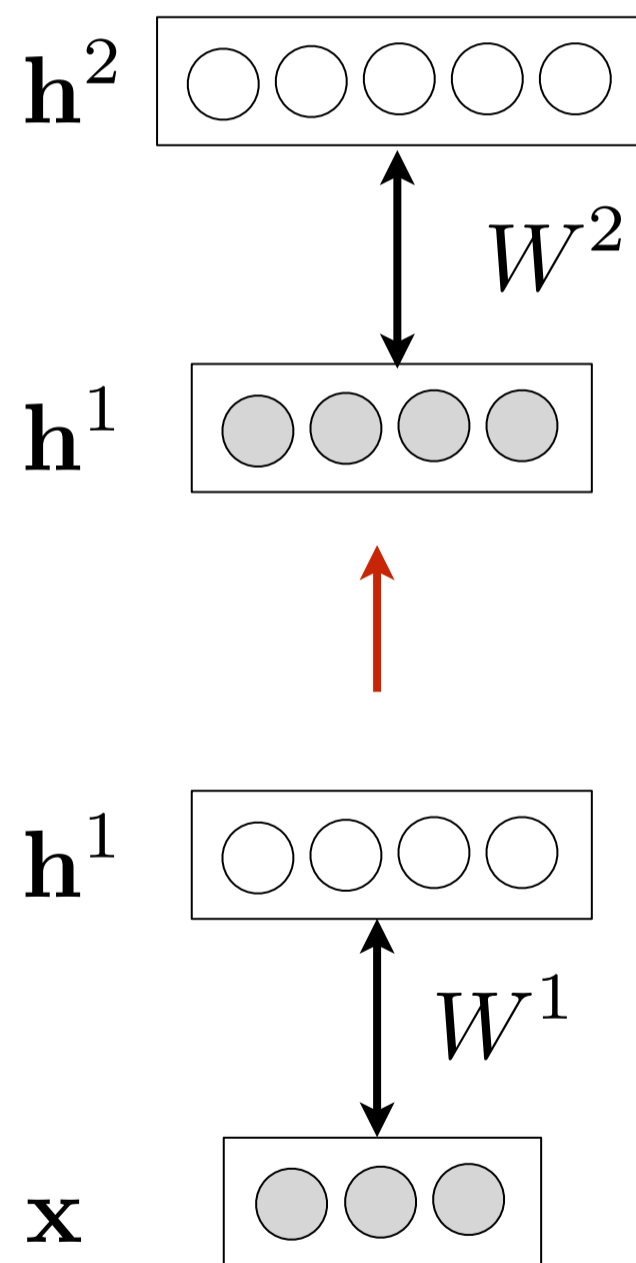
③ Treat the hidden data like data, train another RBM

② Run your data through the model to generate a dataset of hidden activations

① Train an RBM

# Stacking RBMs: Procedure

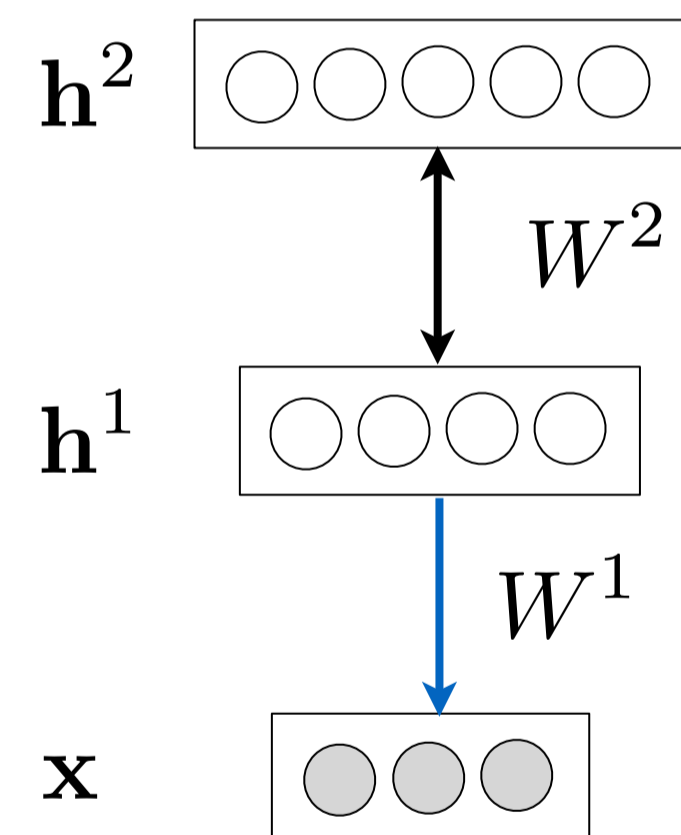
④ Compose the two models



③ Treat the hidden activations like data, train another RBM

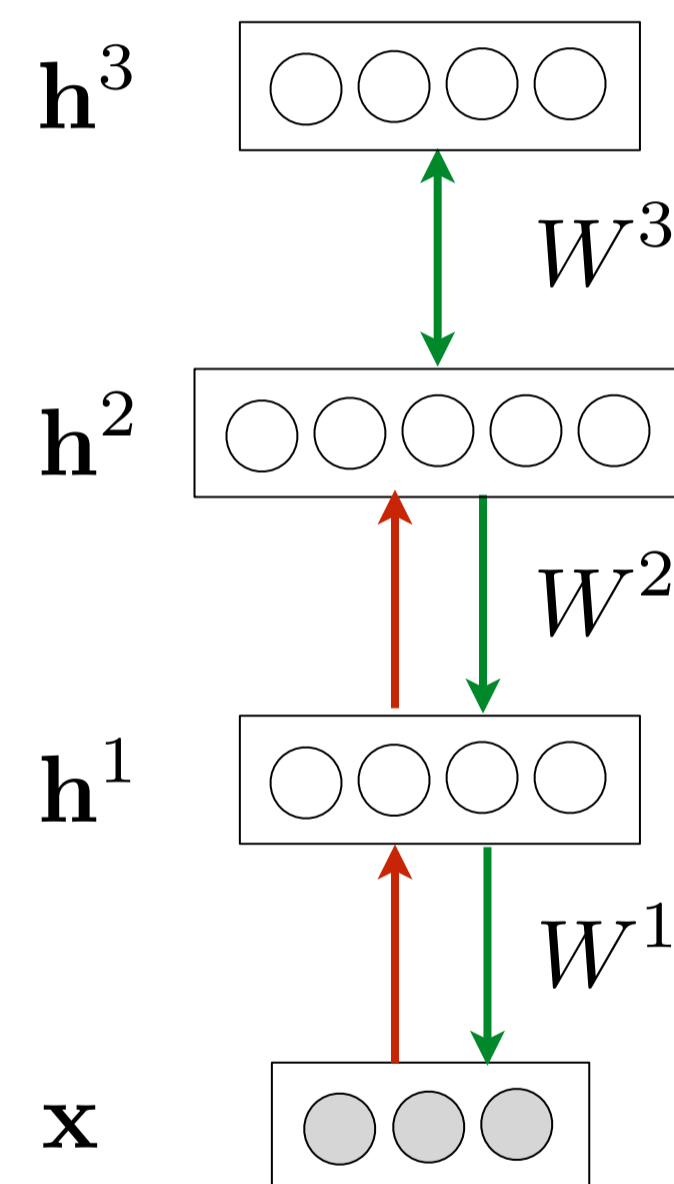
② Run your data through the model to generate a dataset of hidden activations

① Train an RBM



# Deep Belief Networks

- The resulting model is called a Deep Belief Network
- Generate by alternating Gibbs sampling between the top two layers followed by a down-pass
- The lower level bottom-up connections are not part of the generative model, they are used only for inference

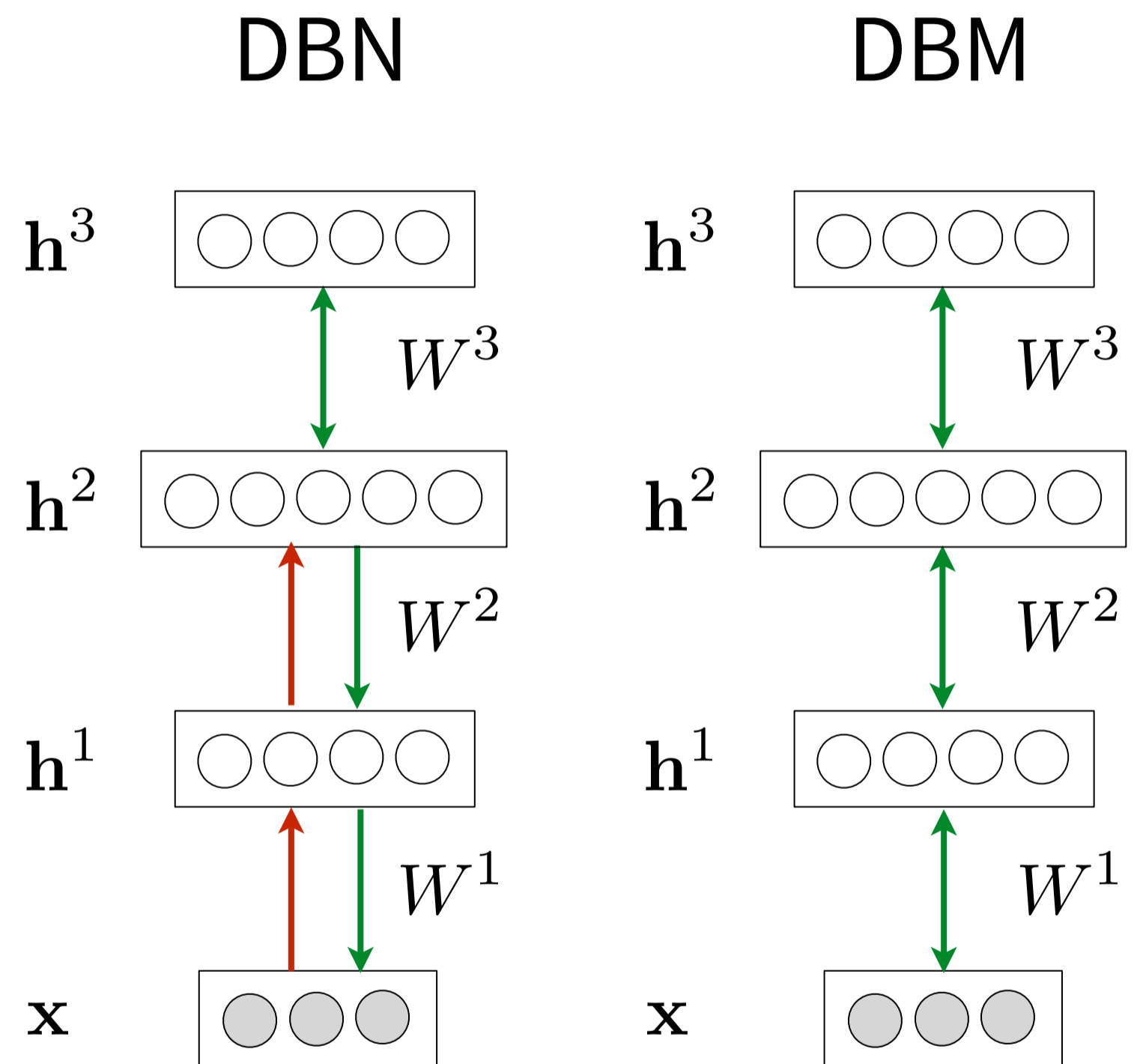


# Stacking RBMs: Intuition

- The weights in the bottom-most RBM define many **different distributions**:  $p(\mathbf{x}, \mathbf{h}), p(\mathbf{x}|\mathbf{h}), p(\mathbf{h}|\mathbf{x}), p(\mathbf{x}), p(\mathbf{h})$
- We can express the RBM as:  $p(\mathbf{x}) = \sum_h p(\mathbf{h})p(\mathbf{x}|\mathbf{h})$
- If we leave  $p(\mathbf{x}|\mathbf{h})$  as-is and improve  $p(\mathbf{h})$ , we improve  $p(\mathbf{x})$
- To improve  $p(\mathbf{h})$  we need it to be **better than**  $p(\mathbf{h}; W^1)$  at modeling the aggregated posterior over hidden vectors produced by applying the RBM to the data

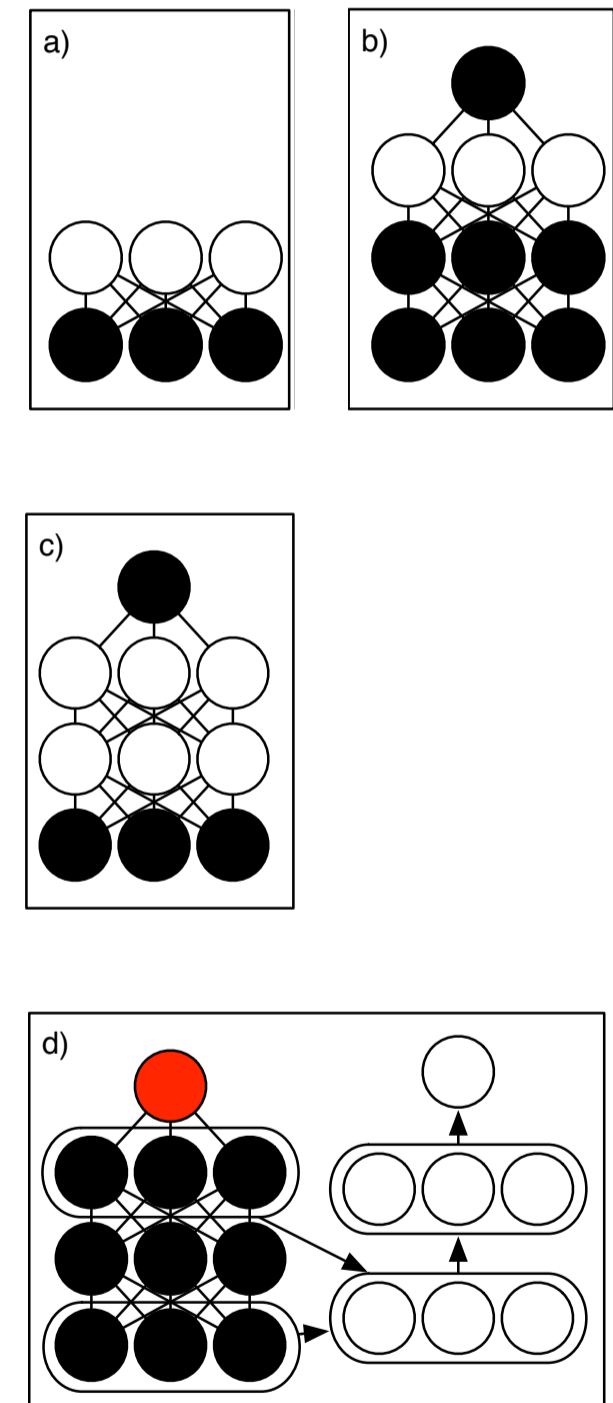
# Deep Boltzmann Machines

- DBN is a hybrid directed graphical model
  - maintains a set of “feed-forward” connections for inference
- DBN is an undirected graphical model
  - **feedback** is important
- Both take different approaches to dealing with intractable  $p(\mathbf{h}|\mathbf{x})$



# Training DBMs

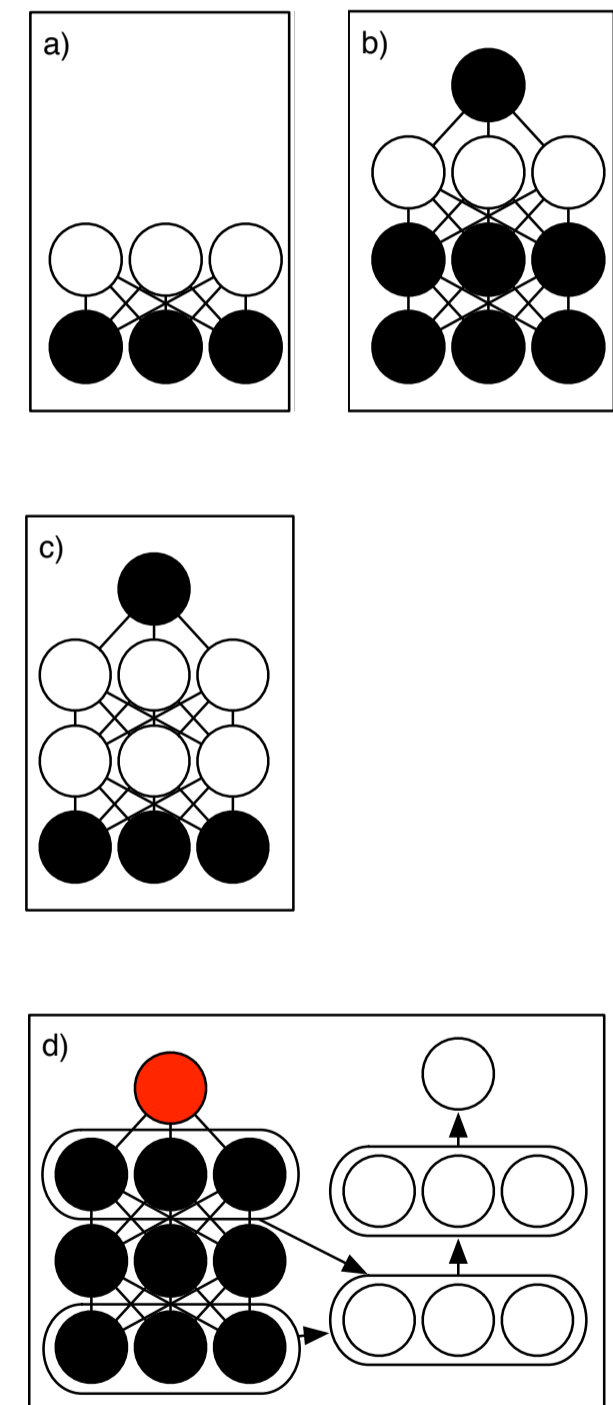
# Training DBMs





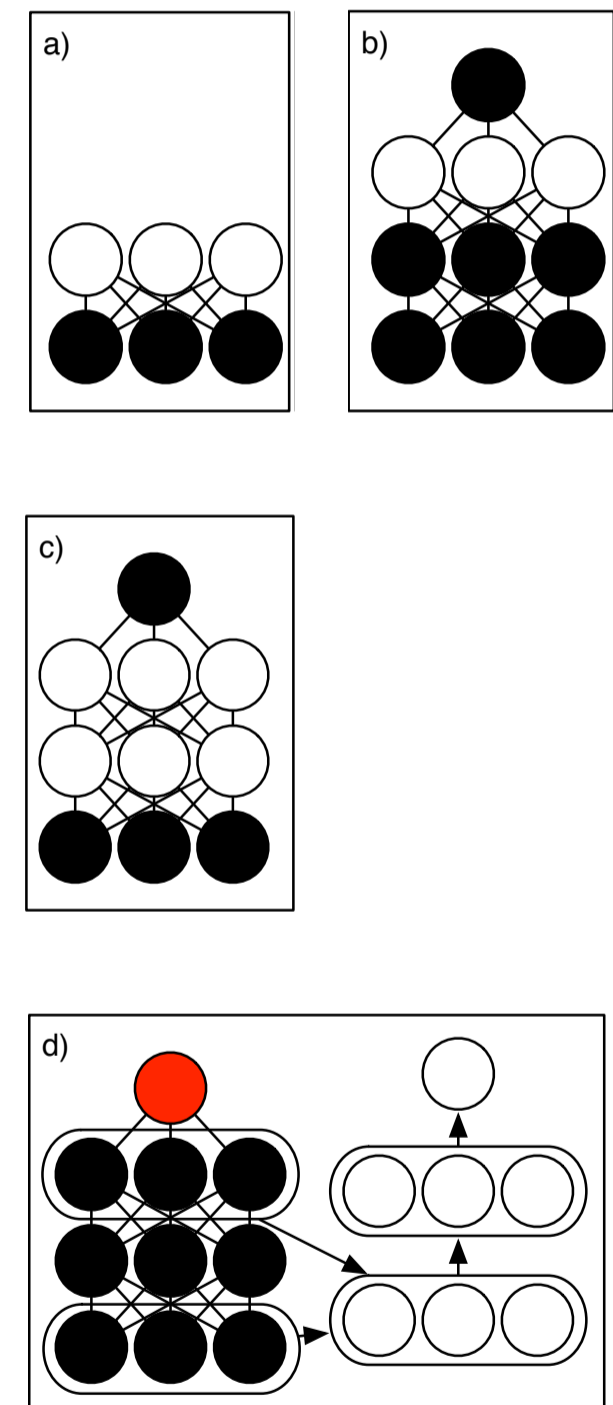
# Training DBMs

- Standard DBM training procedure:



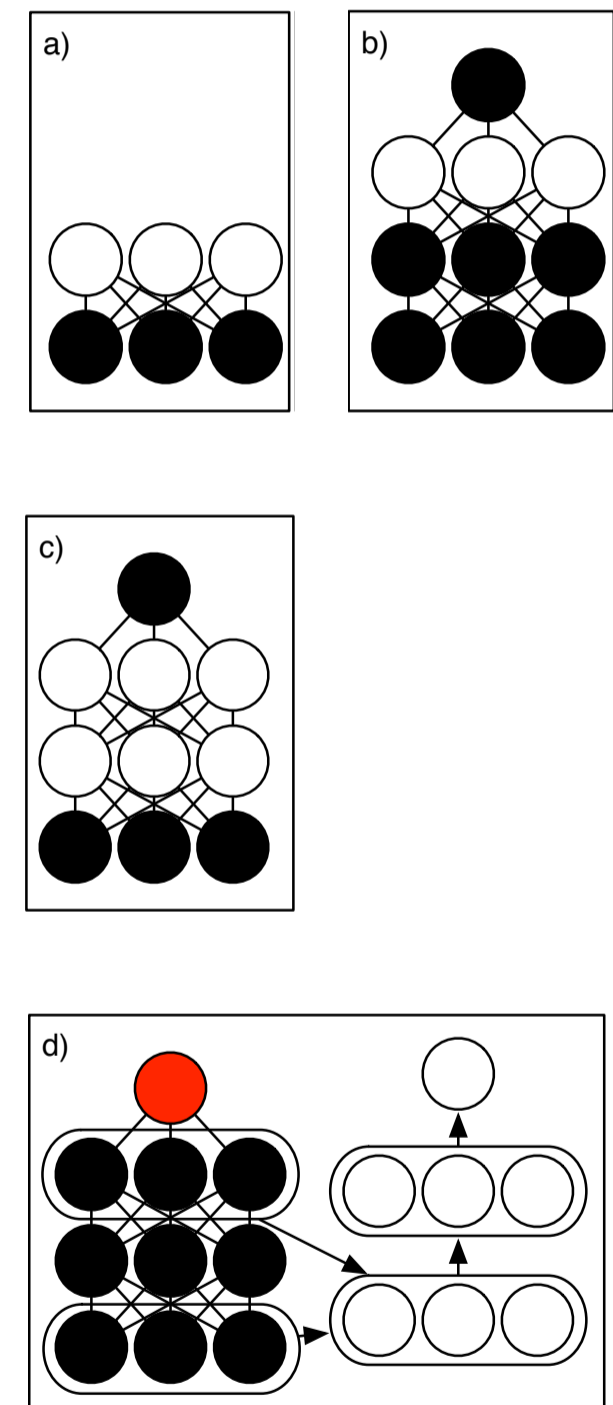
# Training DBMs

- Standard DBM training procedure:
  - Greedy-wise pre-training of RBMs



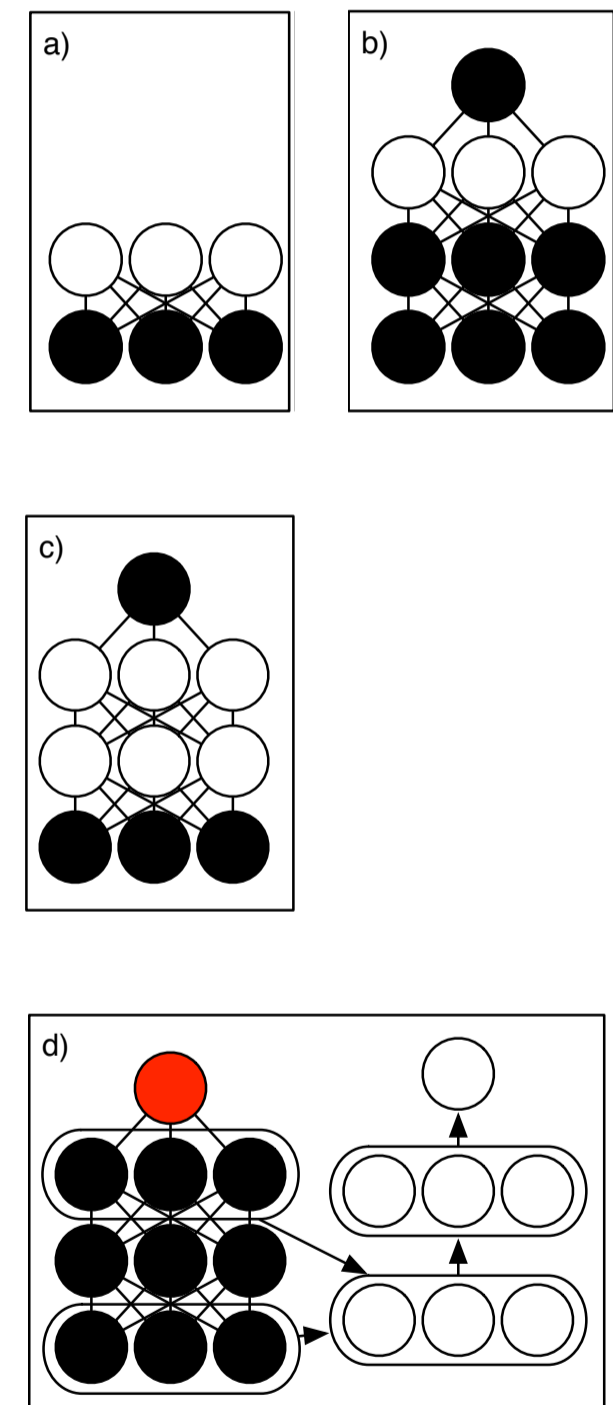
# Training DBMs

- Standard DBM training procedure:
  - Greedy-wise pre-training of RBMs
  - Stitch the RBMs into a DBM and train with variational approximation to log-likelihood



# Training DBMs

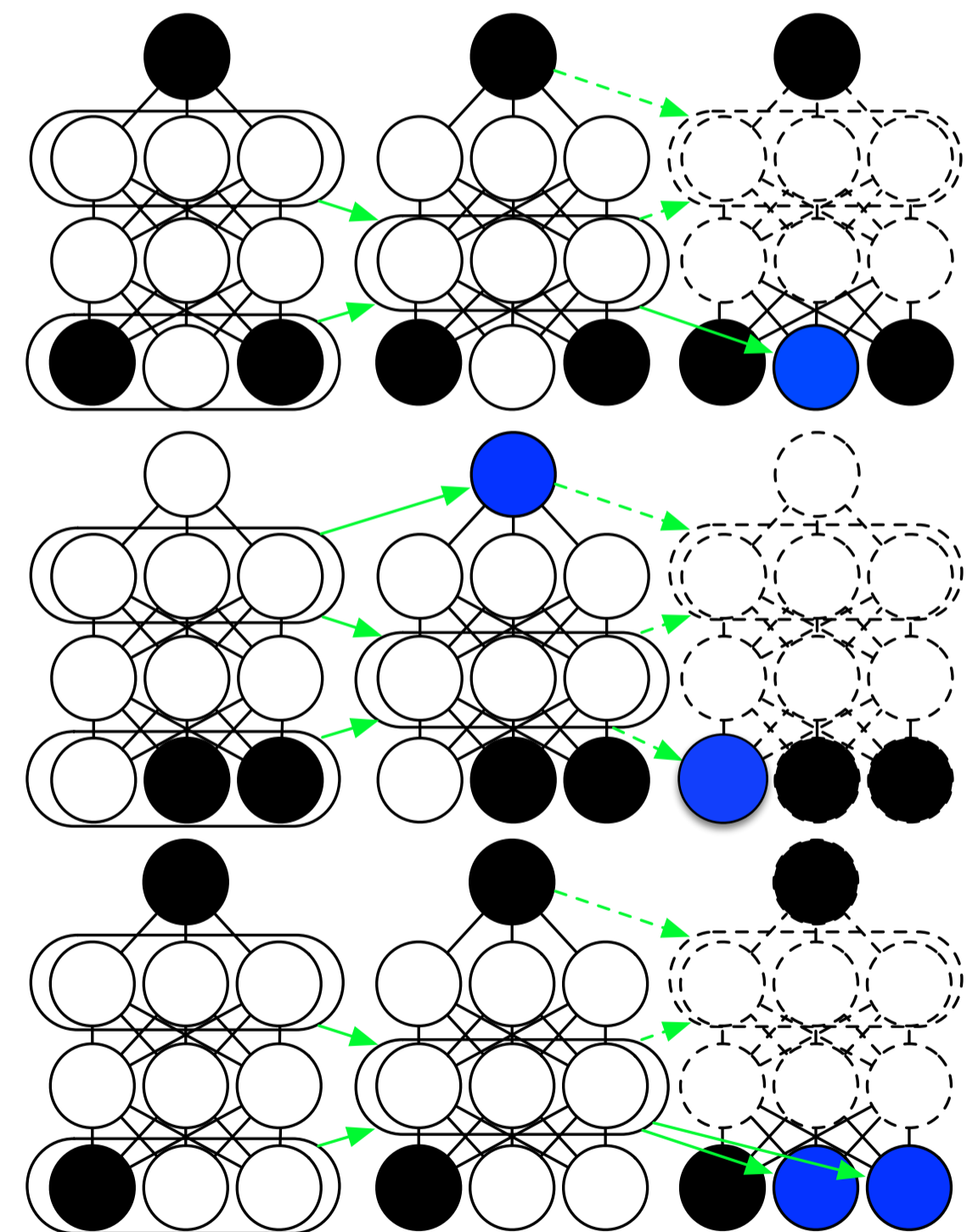
- Standard DBM training procedure:
  - Greedy-wise pre-training of RBMs
  - Stitch the RBMs into a DBM and train with variational approximation to log-likelihood
  - Discriminative fine-tuning (DBM used as feature learner)



# Multi-prediction DBMs

# Multi-prediction DBMs

Multi-prediction training  
for classification

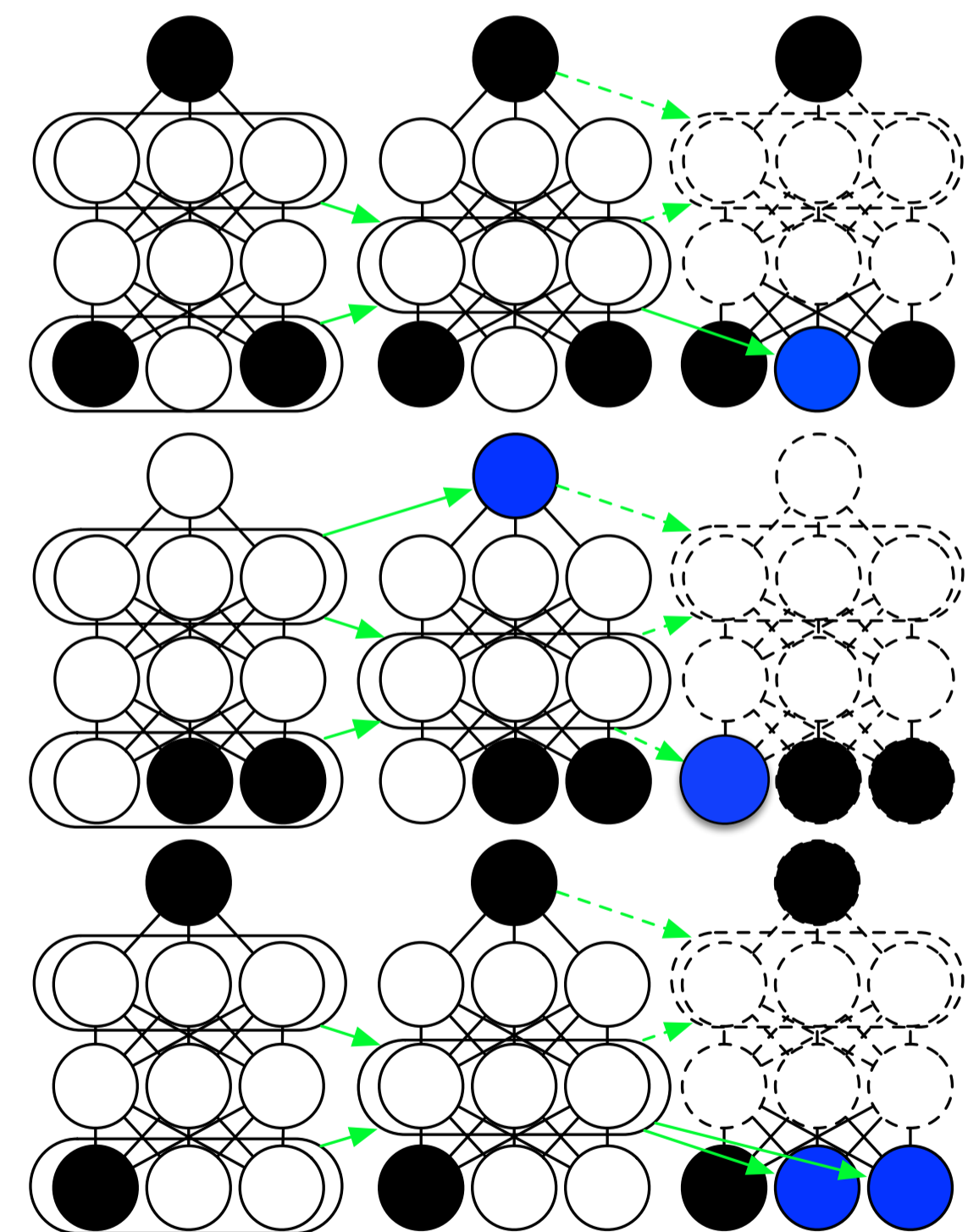


Black - variables net is allowed to observe  
Blue - prediction targets

# Multi-prediction DBMs

- Greedy pre-training is suboptimal
  - training procedure for each layer should account for the influence of deeper layers
  - one model for all tasks can use inference for arbitrary queries
  - needing to implement multiple models and stages makes DBMs cumbersome

## Multi-prediction training for classification

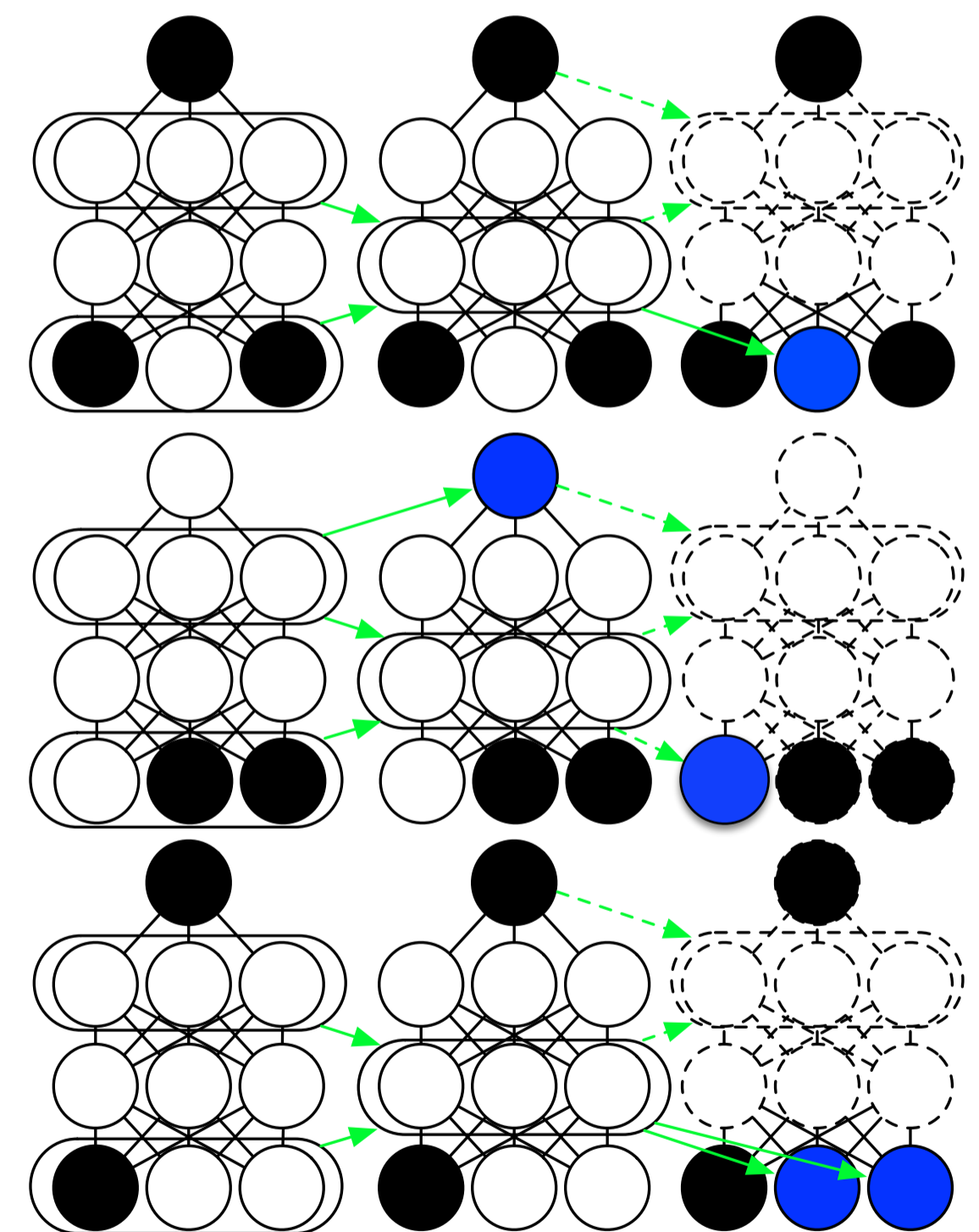


Black - variables net is allowed to observe  
Blue - prediction targets

# Multi-prediction DBMs

- Greedy pre-training is suboptimal
  - training procedure for each layer should account for the influence of deeper layers
  - one model for all tasks can use inference for arbitrary queries
  - needing to implement multiple models and stages makes DBMs cumbersome
- Joint “multi-prediction” training (Goodfellow et al. 2013)
  - Train DBM to predict any subset of vars given the complement of that subset

## Multi-prediction training for classification



Black - variables net is allowed to observe  
Blue - prediction targets



# Conclusions and Challenges

- Most DL success has been achieved by **supervised learning** in the past few years
- All of the **technical challenges** we mentioned for supervised methods apply to **unsupervised learning**
- Single-layer unsupervised learners well developed but joint unsupervised training of deep models remains difficult
- Can we train deep structured output models?

# Thank You!



# Resources

- Online courses
  - Andrew Ng's Machine Learning (Coursera)
  - Geoff Hinton's Neural Networks (Coursera)
- Websites
  - [deeplearning.net](http://deeplearning.net)
  - [http://deeplearning.stanford.edu/wiki/index.php/UFLDL\\_Tutorial](http://deeplearning.stanford.edu/wiki/index.php/UFLDL_Tutorial)

# Surveys and Reviews

Y. Bengio, A. Courville, and P. Vincent. Representation learning: A review and new perspectives. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 35(8):1798–1828, Aug 2013.

Y. Bengio. Deep learning of representations: Looking forward. In *Statistical Language and Speech Processing*, pages 1–37. Springer, 2013.

Y. Bengio, I. Goodfellow, and A. Courville. *Deep Learning*. 2014. Draft available at <http://www.iro.umontreal.ca/~bengioy/dlbook/>

J. Schmidhuber. Deep learning in neural networks: An overview. arXiv preprint arXiv:1404.7828, 2014.

Y. Bengio. Learning deep architectures for ai. *Foundations and trends in Machine Learning*, 2(1):1–127, 2009.

# Papers in this Tutorial

D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio. Why does unsupervised pre-training help deep learning? *The Journal of Machine Learning Research*, 11:625–660, 2010.

K. Kavukcuoglu, M. Ranzato, and Y. LeCun. Fast inference in sparse coding algorithms with applications to object recognition. *arXiv preprint arXiv:1010.3467*, 2010.

P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM, 2008.

S. Rifai, P. Vincent, X. Muller, X. Glorot, and Y. Bengio. Contractive auto-encoders: Explicit invariance during feature extraction. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 833–840, 2011.

G. Alain and Y. Bengio. What regularized auto-encoders learn from the data generating distribution. *arXiv preprint arXiv:1211.4246*, 2012.

Y. Bengio, L. Yao, G. Alain, and P. Vincent. Generalized denoising auto-encoders as generative models. In *Advances in Neural Information Processing Systems*, pages 899–907, 2013.

I.H. Kamyschanska and R. Memisevic. On autoencoder scoring. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 720–728, 2013.

B. M. Marlin, K. Swersky, B. Chen, and N. D. Freitas. Inductive principles for restricted boltzmann machine learning. In *International Conference on Artificial Intelligence and Statistics*, pages 509–516, 2010.

G. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

R. Salakhutdinov and G. E. Hinton. Deep boltzmann machines. In *International Conference on Artificial Intelligence and Statistics*, pages 448–455, 2009.

# Recent Work

M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional neural networks. arXiv preprint arXiv:1311.2901, 2013.

Y. Bengio and E. Thibodeau-Laufer. Deep generative stochastic networks trainable by backprop. arXiv preprint arXiv:1306.1091, 2013.

I. Goodfellow, M. Mirza, A. Courville, and Y. Bengio. Multi-prediction deep boltzmann machines. In Advances in Neural Information Processing Systems, pages 548–556, 2013.

Y. He, K. Kavukcuoglu, Y. Wang, A. Szlam, and Y. Qi. Unsupervised feature learning by deep sparse coding. In ICLR, 2014.

# Practical Tips

Y. Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural Networks: Tricks of the Trade*, pages 437–478. Springer, 2012.

G. E. Hinton. A practical guide to training restricted boltzmann machines. In *Neural Networks: Tricks of the Trade*, pages 599–619. Springer, 2012.