CEng 796 – Spring 2020

R. Gökberk Cinbiş, Emre Akbaş

Week 2: Background Review

Part I: Deep learning review

Part II: Probability, Random Variables, Bayes Nets (Probabilistic Directed Acyclic Graphical Models)

METU Computer Engineering

CEng 796 – Spring 2020

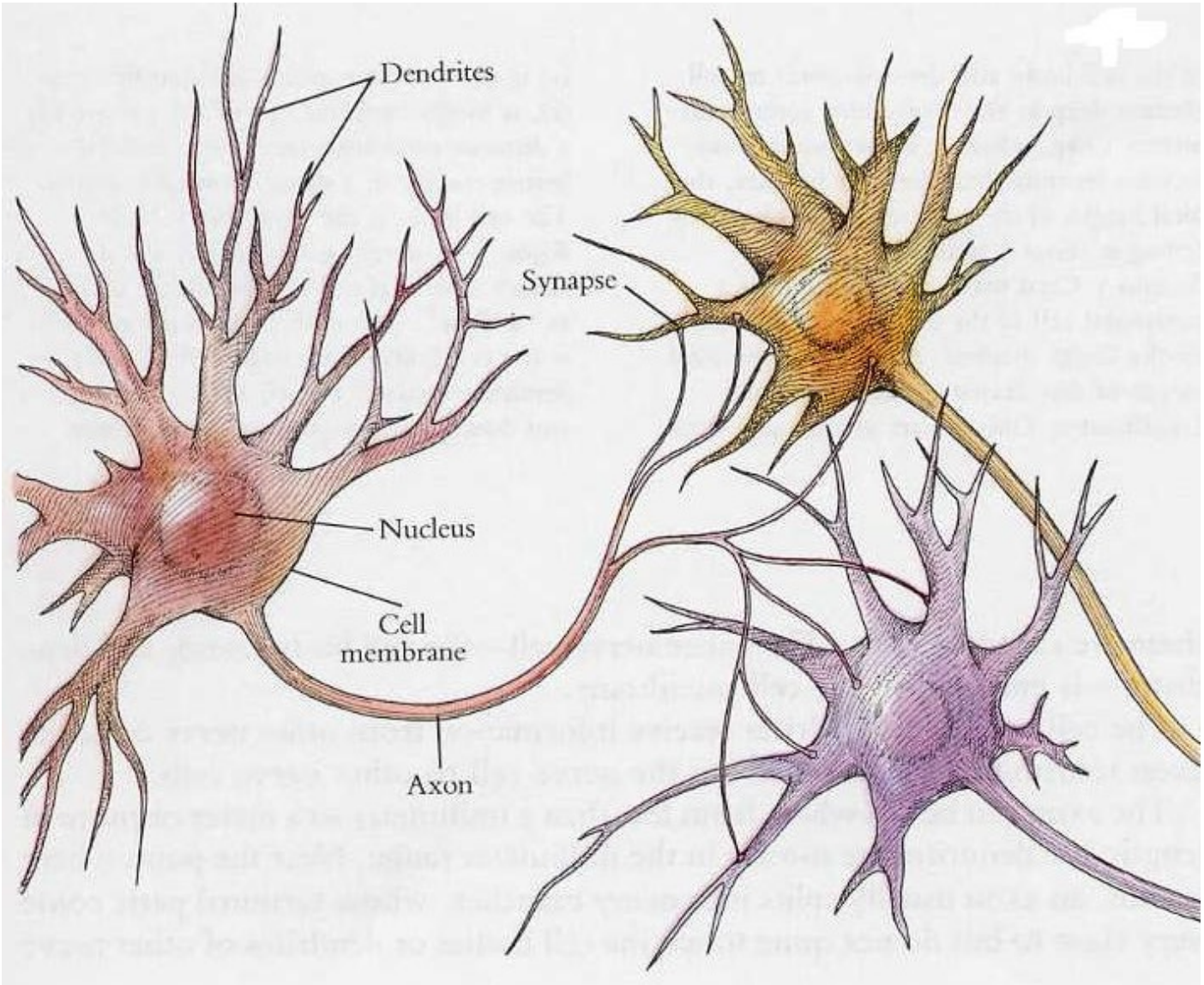
R. Gökberk Cinbiş, Emre Akbaş

Week 2: Background Review

Part I: Deep learning review

Part II: Probability, Random Variables, Bayes Nets (Probabilistic Directed Acyclic Graphical Models)

Biological Neuron



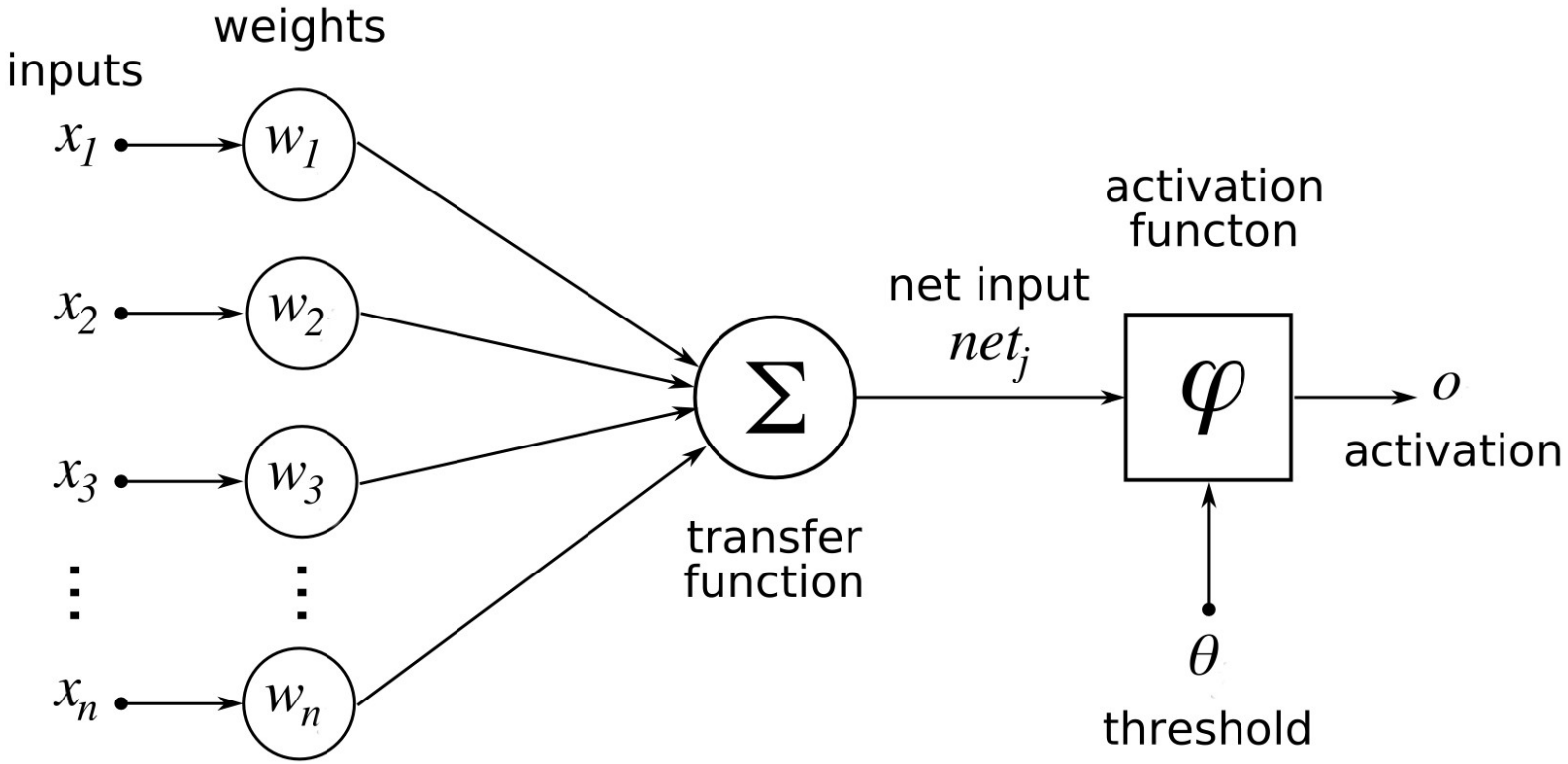
[Figure from Hubel (1995)]

Dendrites receive impulses from other cells.

Signal travels through the **axon** and is delivered to other cells via **axon terminals** through inter-cell regions called **synapses**.

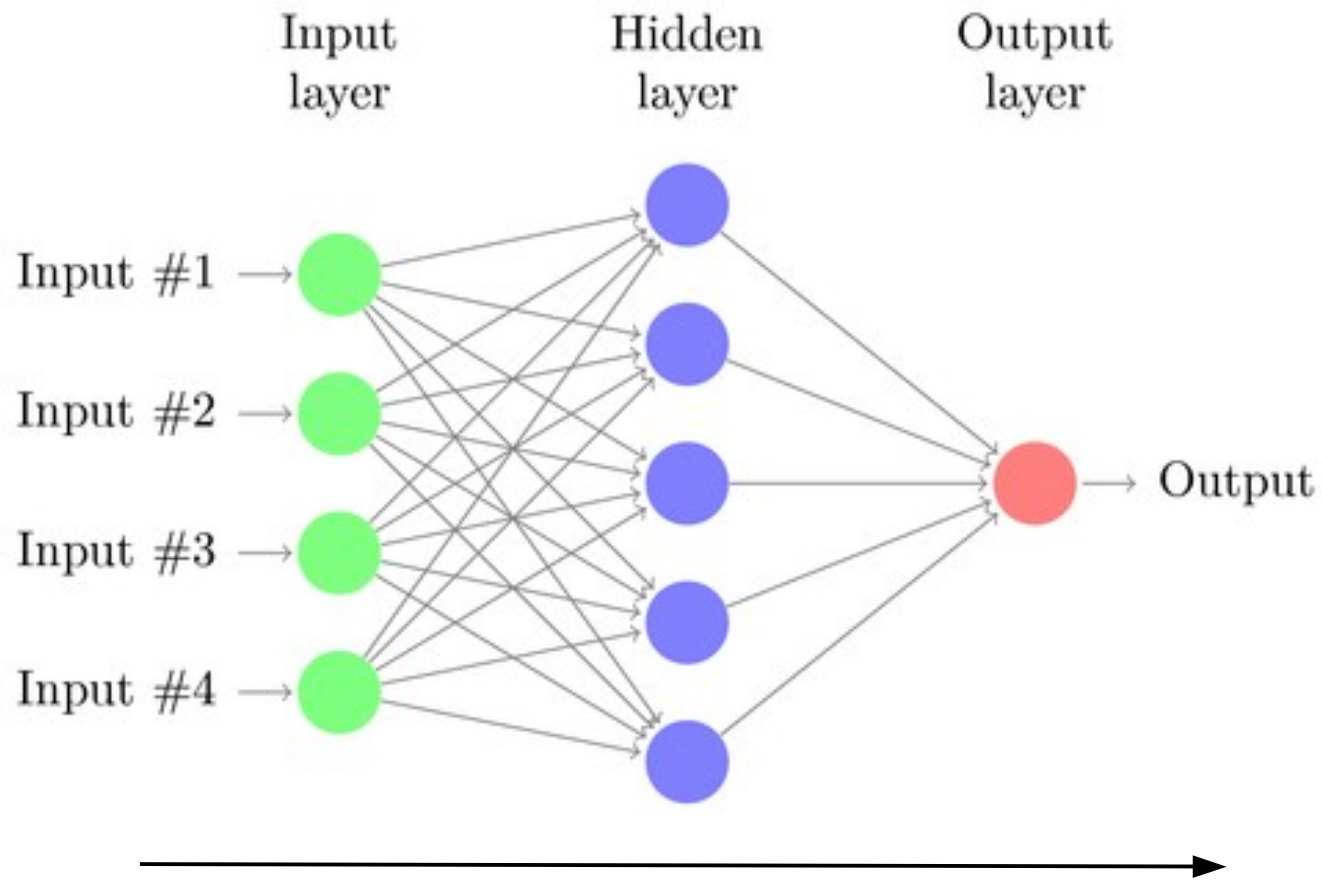
Artificial Neuron

$$o = \varphi\left(\sum_{i=1}^{D} w_i x_i ; \theta\right) \text{ where } x \in \mathbb{R}^D, o \in \mathbb{R}$$



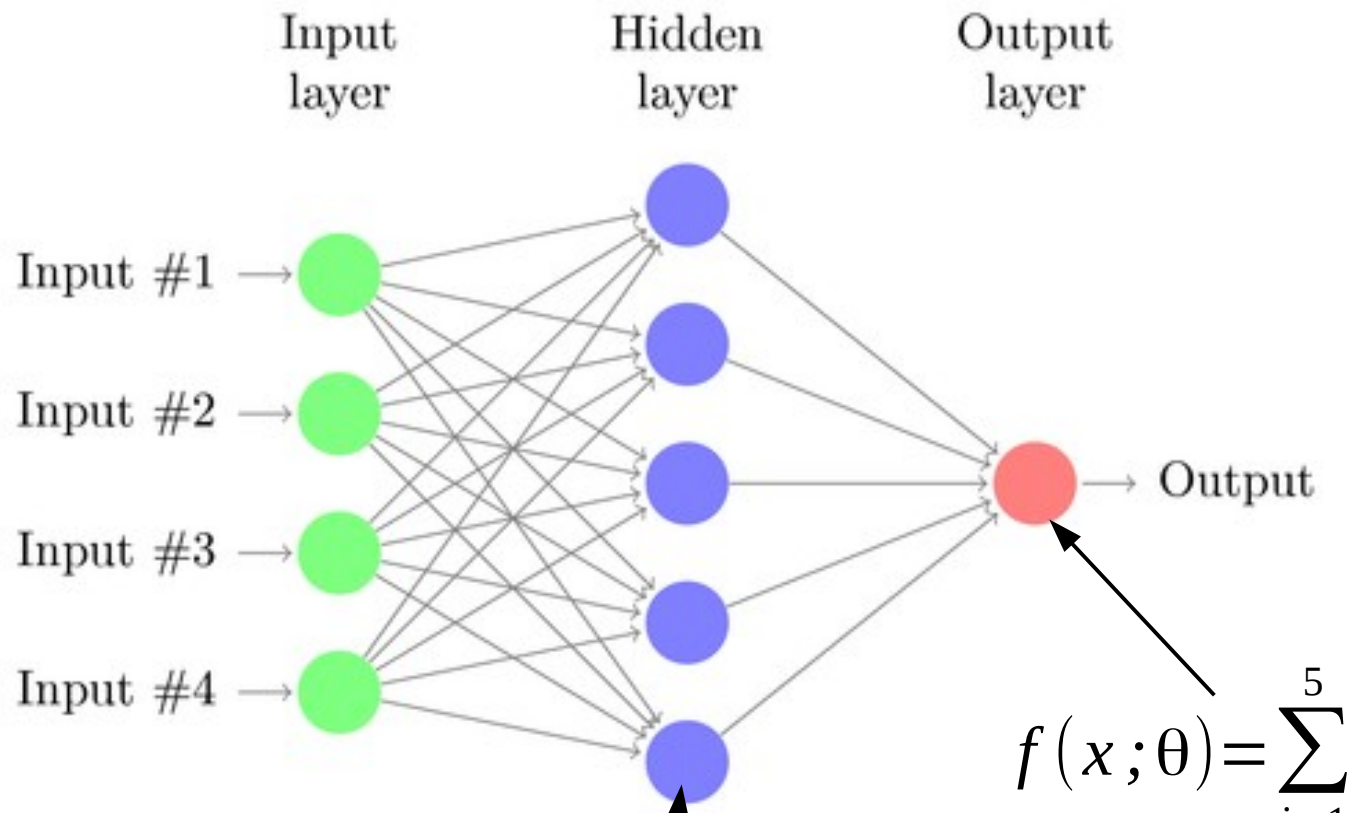
F. Rosenblat's Perceptron (1958)

Multilayer Perceptron (MLP)



Fully connected, feedforward network

Multilayer Perceptron (MLP)



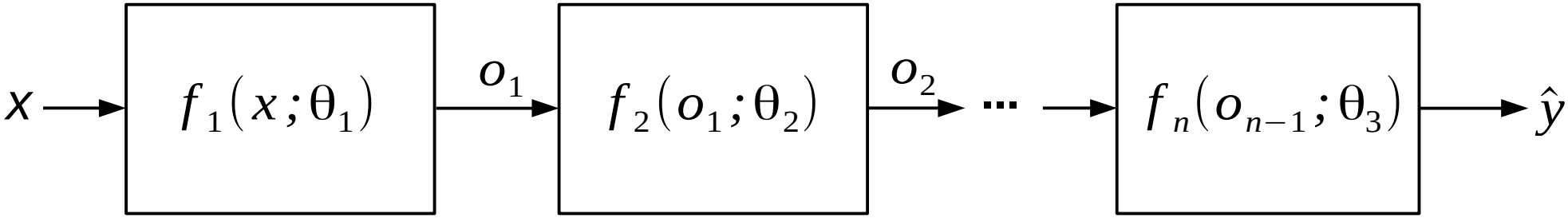
$$f(x;\theta)=\sum_{i=1}^{5}\theta_i\varphi(w_5^T x)$$

Input $x\in\mathbb{R}^4$

$$g(x;w_5)=\varphi(w_5^T x)$$

Activation function

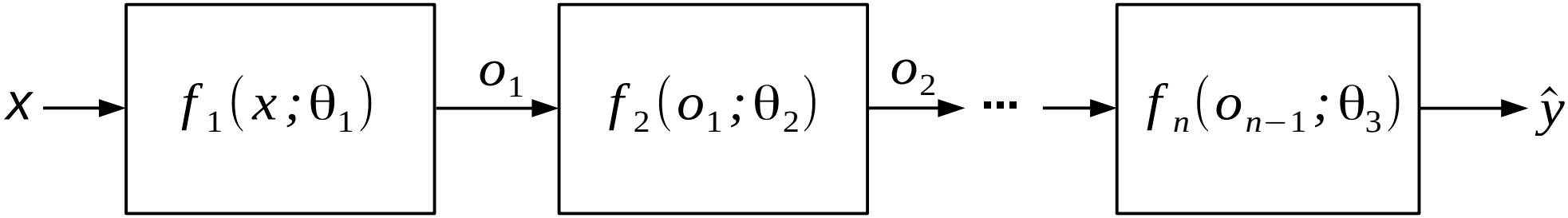
Artificial Neural Networks



$$\hat{y} = f_n(f_{n-1}(\dots f_2(f_1(x;\theta_1);\theta_2)\dots;\theta_{n-1});\theta_n)$$

Minimize loss: $\Theta^{best} = \underset{\Theta}{argmin}\, L(\hat{y},\dots,\Theta)$

Artificial Neural Networks

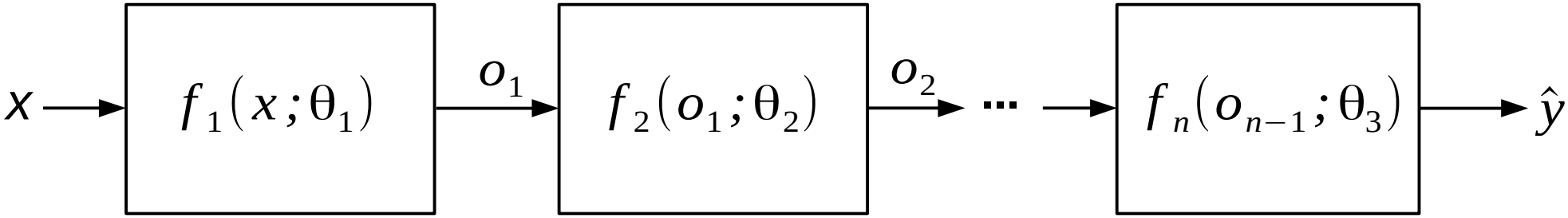


$$\hat{y} = f_n(f_{n-1}(\dots f_2(f_1(x;\theta_1);\theta_2)\dots;\theta_{n-1});\theta_n)$$

Minimize loss: $\Theta^{best} = \underset{\Theta}{argmin}\, L(\hat{y},\dots,\Theta)$

Differentiable *L( )* and *f( )*'s

Artificial Neural Networks



$$\hat{y} = f_n(f_{n-1}(\dots f_2(f_1(x;\theta_1);\theta_2)\dots;\theta_{n-1});\theta_n)$$

Minimize loss: $\Theta^{best} = \underset{\Theta}{argmin}\, L(\hat{y},\dots,\Theta)$

Differentiable *L( )* and *f( )*'s

f( ) is typically

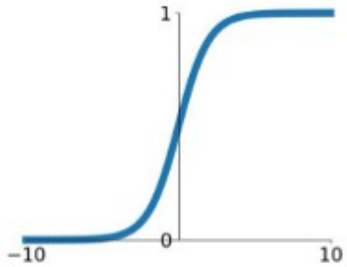
- Linear operation (dot-product, matrix mult.,convolution,...)
- Non-linear operation (activation, pooling, normalization,...)

Activation function

Introduces non-linearity

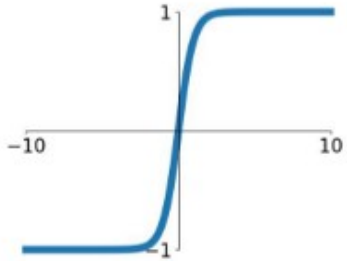
Sigmoid

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



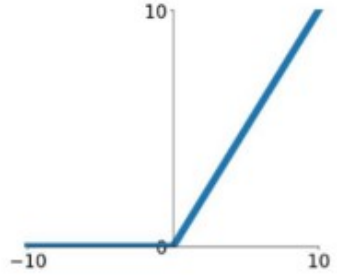
tanh

$$\tanh(x)$$



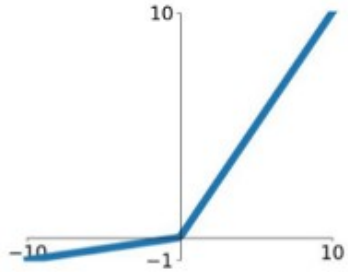
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

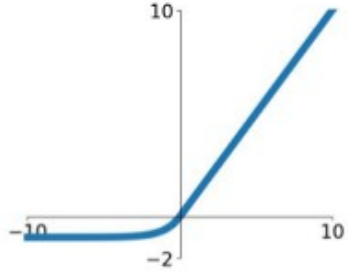


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



[Figure by P. Jain]

MLPs are universal function approximators

Cybenko 1989:

Approximation by Superpositions of a Sigmoidal Function*

G. Cybenko†

Abstract. In this paper we demonstrate that finite linear combinations of compositions of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of n real variables with support in the unit hypercube; only mild conditions are imposed on the univariate function. Our results settle an open question about representability in the class of single hidden layer neural networks. In particular, we show that arbitrary decision regions can be arbitrarily well approximated by continuous feedforward neural networks with only a single internal, hidden layer and any continuous sigmoidal nonlinearity. The paper discusses approximation properties of other possible types of nonlinearities that might be implemented by artificial neural networks.

Key words. Neural networks, Approximation, Completeness.

1. Introduction

A number of diverse application areas are concerned with the representation of general functions of an n-dimensional real variable, $x \in \mathbb{R}^n$, by finite linear combinations of the form

$$\sum_{j=1}^{N} \alpha_j \sigma(y_j^{\mathrm{T}} x + \theta_j), \qquad (1)$$

where $y_j \in \mathbb{R}^n$ and $\alpha_j, \theta \in \mathbb{R}$ are fixed. (y^{T} is the transpose of y so that $y^{\mathrm{T}}x$ is the inner product of y and x.) Here the univariate function σ depends heavily on the context of the application. Our major concern is with so-called sigmoidal σ's:

$$\sigma(t) \to \begin{cases} 1 & \text{as} \quad t \to +\infty, \\ 0 & \text{as} \quad t \to -\infty. \end{cases}$$

MLPs are universal function approximators

Cybenko 1989:

Approximation by Superpositions of a Sigmoidal Function*

G. Cybenko†

Abstract. In this paper we demonstrate that finite linear combinations of compositions of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of n real variables with support in the unit hypercube; only mild conditions are imposed on the univariate function. Our results settle an open question about representability in the class of single hidden layer neural networks. In particular, we show that arbitrary decision regions can be arbitrarily well approximated by continuous feedforward neural networks with only a single internal, hidden layer and any continuous sigmoidal nonlinearity. The paper discusses approximation properties of other possible types of nonlinearities that might be implemented by artificial neural networks.

Key words. Neural networks, Approximation, Completeness.

1. Introduction

A number of diverse application areas are concerned with the representation of general functions of an n-dimensional real variable, $x \in \mathbb{R}^n$, by finite linear combinations of the form

$$\sum_{j=1}^{N} \alpha_j \sigma(y_j^T x + \theta_j), \qquad (1)$$

where $y_j \in \mathbb{R}^n$ and $\alpha_j, \theta \in \mathbb{R}$ are fixed. (y^T is the transpose of y so that $y^T x$ is the inner product of y and x.) Here the univariate function σ depends heavily on the context of the application. Our major concern is with so-called sigmoidal σ's:

$$\sigma(t) \to \begin{cases} 1 & \text{as} \quad t \to +\infty, \\ 0 & \text{as} \quad t \to -\infty. \end{cases}$$

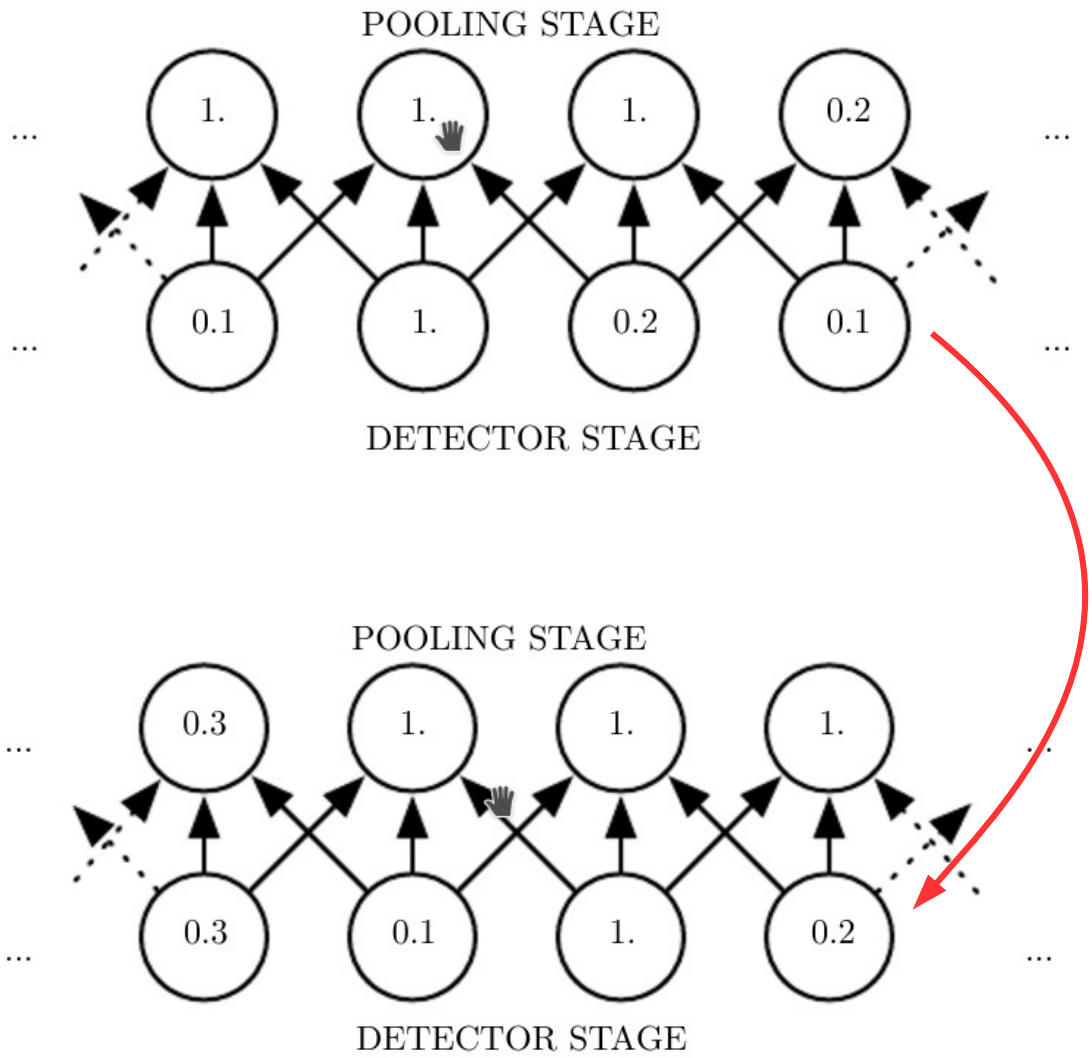
These theorems are about the representation power of NNs not about their learnability or practical feasibility.

Pooling

A pooling function takes the output of the previous layer at a certain location *L* and computes a "summary" of the neighborhood around *L*.

E.g. max-pooling

Max-pooling



Max-pooling introduces **translation invariance**.

Input layer has shifted to the right 1-pixel.

But only half of the values in the output layer have changed.

Figure 9.8 from Goodfellow et al. (2016).

Normalization

Softmax:

$$q_{ic} = \frac{e^{f_c(x_i)}}{\sum_k e^{f_k(x_i)}}$$

Batch-normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

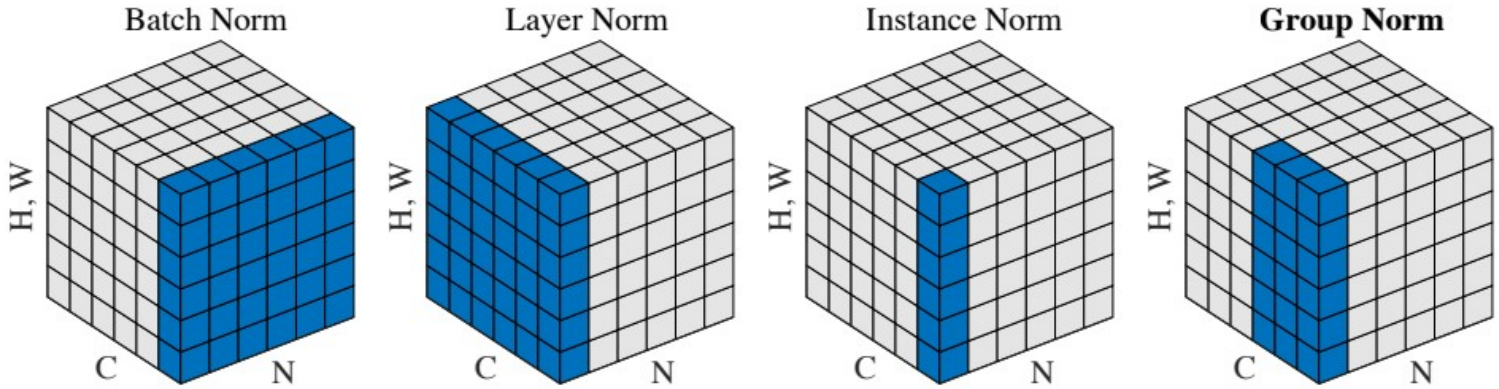


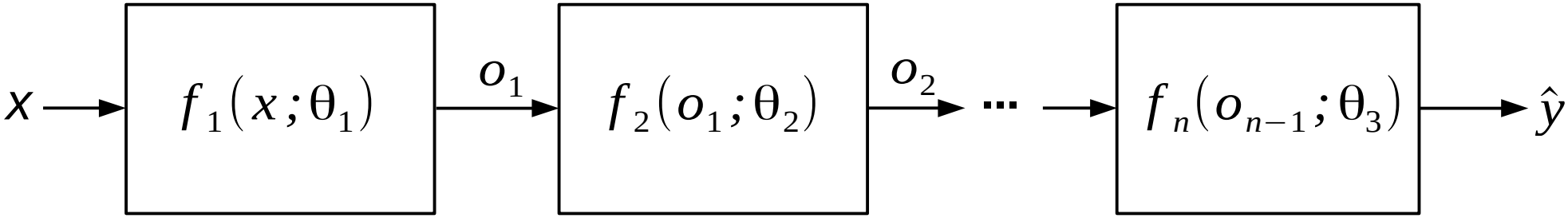
Figure 2. **Normalization methods**. Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

```
def GroupNorm(x, gamma, beta, G, eps=1e−5):
    # x: input features with shape [N,C,H,W]
    # gamma, beta: scale and offset, with shape [1,C,1,1]
    # G: number of groups for GN

    N, C, H, W = x.shape
    x = tf.reshape(x, [N, G, C // G, H, W])

    mean, var = tf.nn.moments(x, [2, 3, 4], keep_dims=True)
    x = (x − mean) / tf.sqrt(var + eps)

    x = tf.reshape(x, [N, C, H, W])

    return x ∗ gamma + beta
```

Figure 3. Python code of Group Norm based on TensorFlow.

[From “Group normalization”, Wu et al. ECCV 2018]

Artificial Neural Networks



$$\hat{y} = f_n(f_{n-1}(\dots f_2(f_1(x;\theta_1);\theta_2)\dots;\theta_{n-1});\theta_n)$$

L( ) and its optimization procedure?

Minimize loss: $\Theta^{best} = \underset{\Theta}{argmin}\, L(\hat{y},\dots,\Theta)$

e.g. Cross-entropy

A widely used cost function for multi-class classification.

Given a dataset $S=\{(x_i, y_i)\}_{i=1}^{N}$ where

$x_i \in \mathbb{R}^D$ and y_i is a C-dimensional one-hot vector

$$L=-\sum_{i=1}^{N}\sum_{c=1}^{C} y_{ic}\log q_{ic}$$

q_i is the softmax of $f(x)$

$$q_{ic}=\frac{e^{f^{(c)}(x_i)}}{\sum_k e^{f^{(k)}(x_i)}}$$

How do we optimize?

Stochastic Gradient Descent (SGD)

Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

Require: Learning rate ϵ_k.
Require: Initial parameter $\boldsymbol{\theta}$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$
  end while

[From Goodfellow et al. (2016)]

Stochastic Gradient Descent (SGD)

It is necessary to decrease the learning rate over time.

This is because the SGD gradient estimator introduces a source of noise (the random sampling of *m* training examples) that does not vanish even when we arrive at a minimum.

By comparison, the true gradient of the total cost function becomes small and then 0 when we approach and reach a minimum using batch gradient descent, so batch gradient descent can use a fixed learning rate.

Sufficient condition for convergence:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty, \quad \text{and} \quad \sum_{k=1}^{\infty} \epsilon_k^2 < \infty.$$

(ie. diverges) *(ie. converges)*

[From Goodfellow et al. (2016)]

Setting the learning rate is not trivial. You (almost) always have to search for its optimal value and decay schedule.

So, researchers have developed **adaptive learning rate** methods

Before introducing adaptive learning rate methods

Momentum:

$$v = mv - \alpha \nabla_w f(w)$$

Update: $w = w + v$

Typical value for momentum is *m=0.9*

Per parameter adaptive learning rate methods

- So far, learning rate was global (equally applied to all parameters in the model)
- Methods have been proposed to adapt learning rates per parameter.
  - Motivation: the cost function is highly sensitive in some directions and insensitive in others, in the parameter space
  - So, it might make sense to use different learning rates per parameter.

Per parameter adaptive learning rate methods

- E.g. “delta-bar-delta” algorithm [Jacobs (1988)]:

  “A heuristic method.

  If the partial w.r.t. to a given model parameter, remains the same sign, then the learning rate should increase.

  If it changes sign,then the learning rate should decrease. “

[Goodfellow et al. (2016)]

Adagrad [Duchi et al. (2011)]

```
# Assume the gradient dx and parameter vector x
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

[From http://cs231n.github.io/neural-networks-3/]

Cache keeps track of per parameter sum of squared gradients.

Weights with high gradients → decrease learning rate
weights that receive small or infrequent updates → increase learning rates.

AdaGrad performs well for some but not all deep learning models.

RMSprop

[Slide 29, Lecture 6, Hinton's Coursera class]

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

[From http://cs231n.github.io/neural-networks-3/]

Modifies Adagrad to use a moving average of squared gradients (instead of the complete sum over history).

Typically, *decay_rate = 0.9*

Performs well.

Adam [Kingma and Ba (2014)]

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

[From http://cs231n.github.io/neural-networks-3/]

Uses a “smoothed” gradient *m*.

Recommended values for hyper-parameters: eps=1e-8, beta1=0.9, beta2=0.999

In practice, Adam is currently recommended as the default method to use.

[From http://cs231n.github.io/neural-networks-3/]

How do we compute the gradient?

Backpropagation

A neural network is nothing but a composition of several linear and non-linear functions:

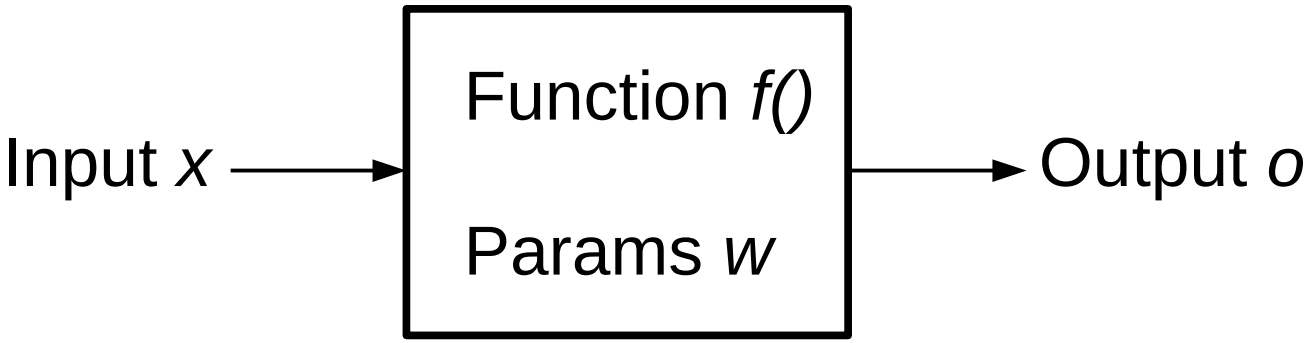
$$y = f_k(f_{k-1}(\dots f_1(x;\theta_1);\theta_{k-1});\theta_k)$$

Given a specific architecture, i.e. composition, one can easily write the gradient w.r.t. parameters.

But a modular approach is desirable so that we don't have to derive the gradient again and again.

We can “compose” new architectures by simply connecting computing blocks.

A computing block:

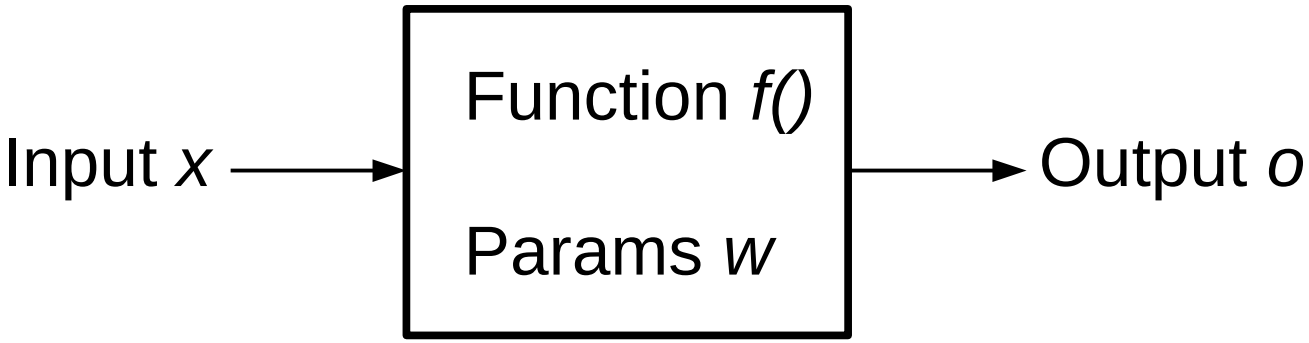


Forward pass: $o = f(x; w)$

Derivative of output w.r.t. input: $\frac{\partial o}{\partial x} = \frac{\partial f(x; w)}{\partial x}$

Derivative of output w.r.t. parameters: $\frac{\partial o}{\partial w} = \frac{\partial f(x; w)}{\partial w}$

A computing block:



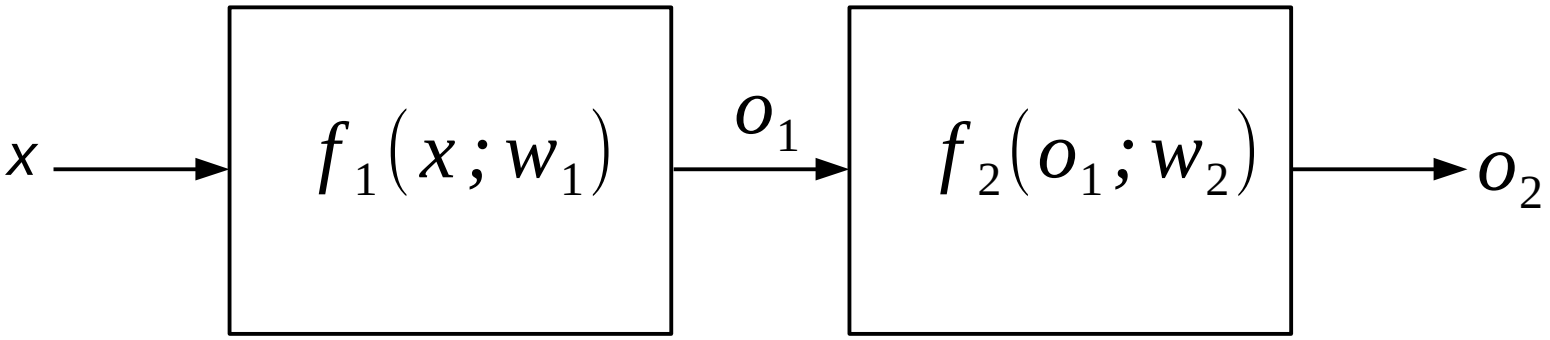
Typically, X, o and w are vectors or matrices. Care has to be taken in computing the derivatives.

Forward pass: $o = f(x; w)$

Derivative of output w.r.t. input: $\frac{\partial o}{\partial x} = \frac{\partial f(x; w)}{\partial x}$

Derivative of output w.r.t. parameters: $\frac{\partial o}{\partial w} = \frac{\partial f(x; w)}{\partial w}$

Multiple blocks



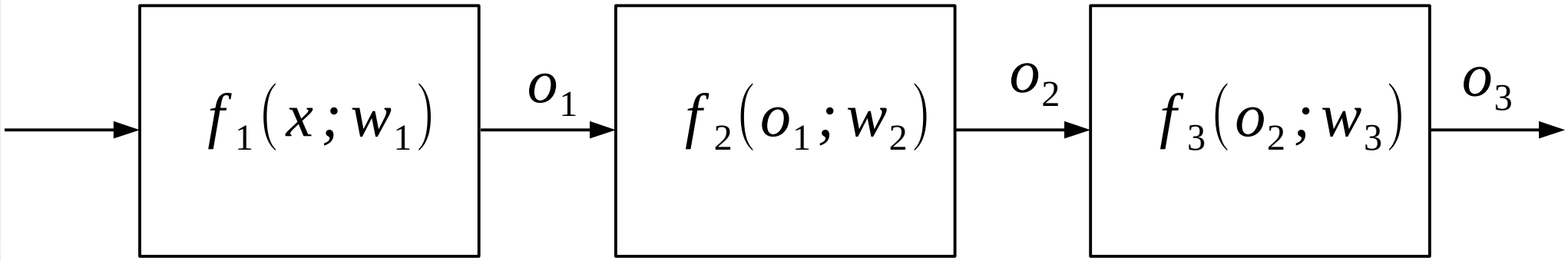
To update w_2 $$\frac{\partial o_2}{\partial w_2}$$

To update w_1 $$\frac{\partial o_2}{\partial w_1} = \frac{\partial o_2}{\partial o_1}\frac{\partial o_1}{\partial w_1}$$

Each block has its own:
- Derivative w.r.t. input
- Derivative w.r.t. parameters.

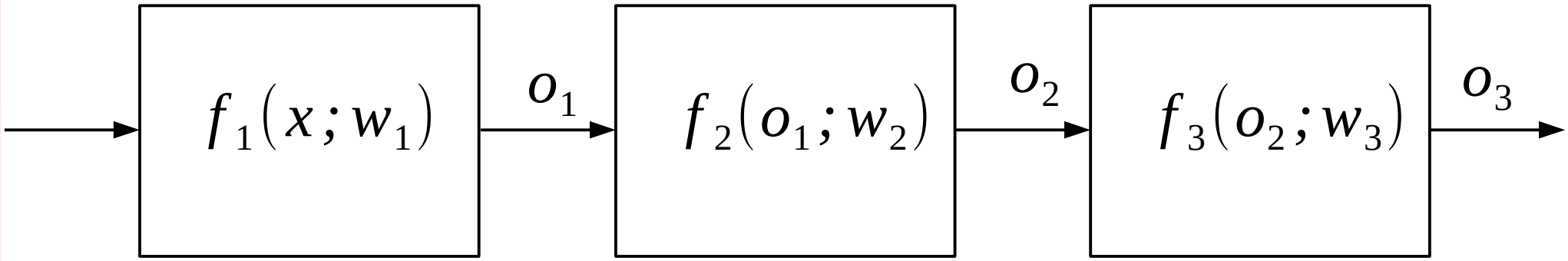
When you are back-propagating, be careful which one to use.

Multiple blocks



$$\frac{\partial o_3}{\partial w_1} = \frac{\partial o_3}{\partial o_2} \frac{\partial o_2}{\partial o_1} \frac{\partial o_1}{\partial w_1}$$

Multiple blocks



$$\frac{\partial o_3}{\partial w_1} = \frac{\partial o_3}{\partial o_2}\frac{\partial o_2}{\partial o_1}\frac{\partial o_1}{\partial w_1}$$

Last step: multiply with derivative w.r.t. parameters

Chain the “derivatives w.r.t. to input”

MLP vs Convolution Neural Network (CNN)

- MLP → fully connected.
  - Uses matrix multiplication to compute the next layer.
- CNN → sparse connections.
  - Uses convolution to compute the next layer.
- Everything else stays the same
  - Activation functions
  - Cost functions
  - Training (back-propagation)
  - ...

In a supervised learning problem,

ConvNets learn both:

- Hierarchical representations of the data, and
- Decision boundaries on these representations

at the same time.

Deep Learning = Learning Hierarchical Representations

Y LeCun

Traditional Pattern Recognition: Fixed/Handcrafted Feature Extractor

Feature Extractor → Trainable Classifier

Mainstream Modern Pattern Recognition: Unsupervised mid-level features

Feature Extractor → Mid-Level Features → Trainable Classifier

Deep Learning: Representations are hierarchical and trained

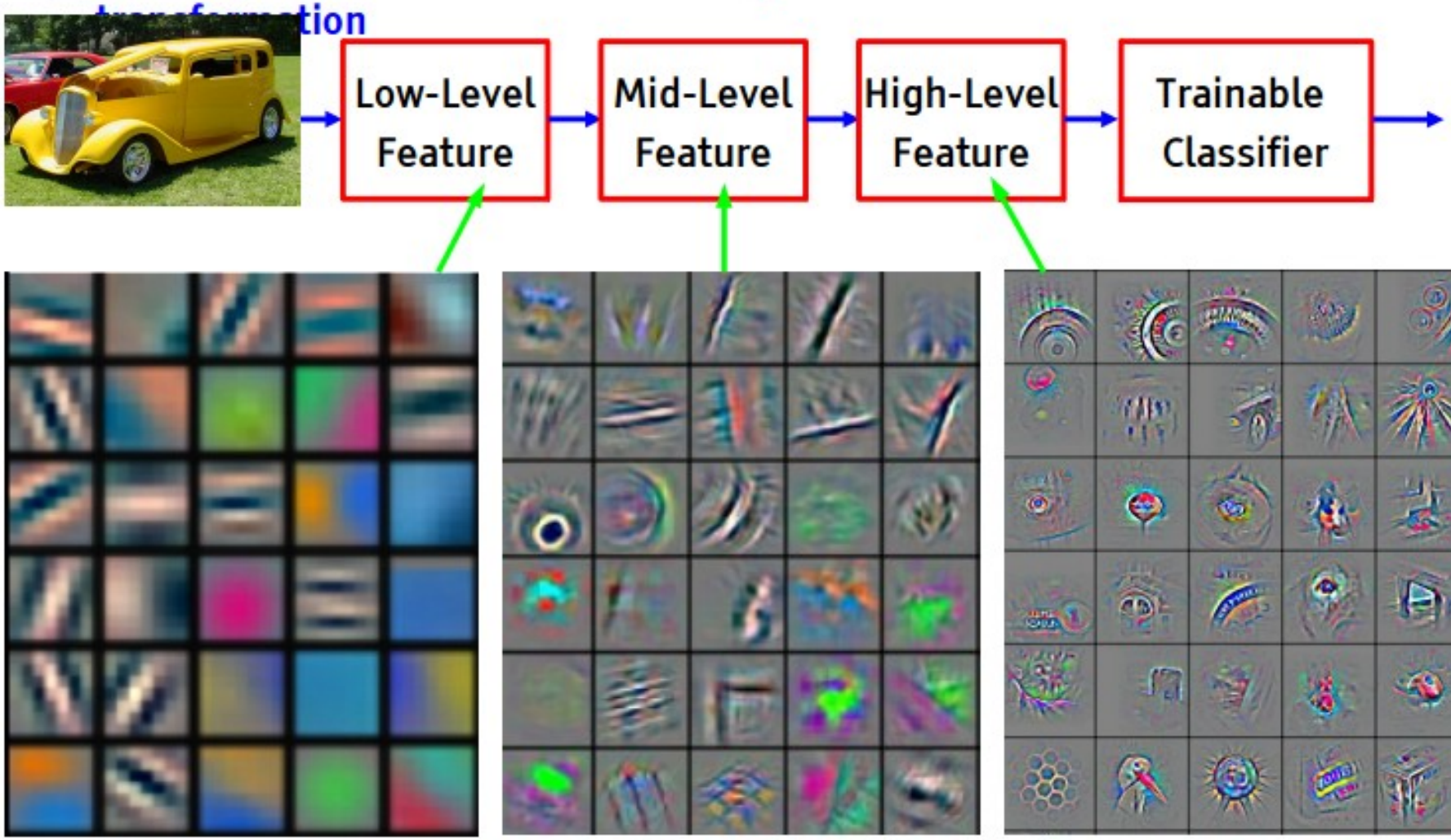
Low-Level Features → Mid-Level Features → High-Level Features → Trainable Classifier

[Slide by Yann LeCun (http://www.slideshare.net/embeddedvision/01-am-keynotelecun)]

Deep Learning = Learning Hierarchical Representations

Y LeCun
MA Ranzato

- It's deep if it has more than one stage of non-linear feature transformation



Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

[Slide by Yann LeCun http://www.cs.nyu.edu/~yann/talks/lecun-ranzato-icml2013.pdf]

Convolution

Closely related to **correlation (a.k.a. cross-correlation)**.

We use these operations **to extract information** from a signal.

$$s(t) = (x \star w)(t) = \int x(a)\, w(a+t)\, da$$

$$s[t] = (x \star w)[t] = \sum_{a=-\infty}^{a=\infty} x[a]\, w[a+t]$$

Sliding dot-product

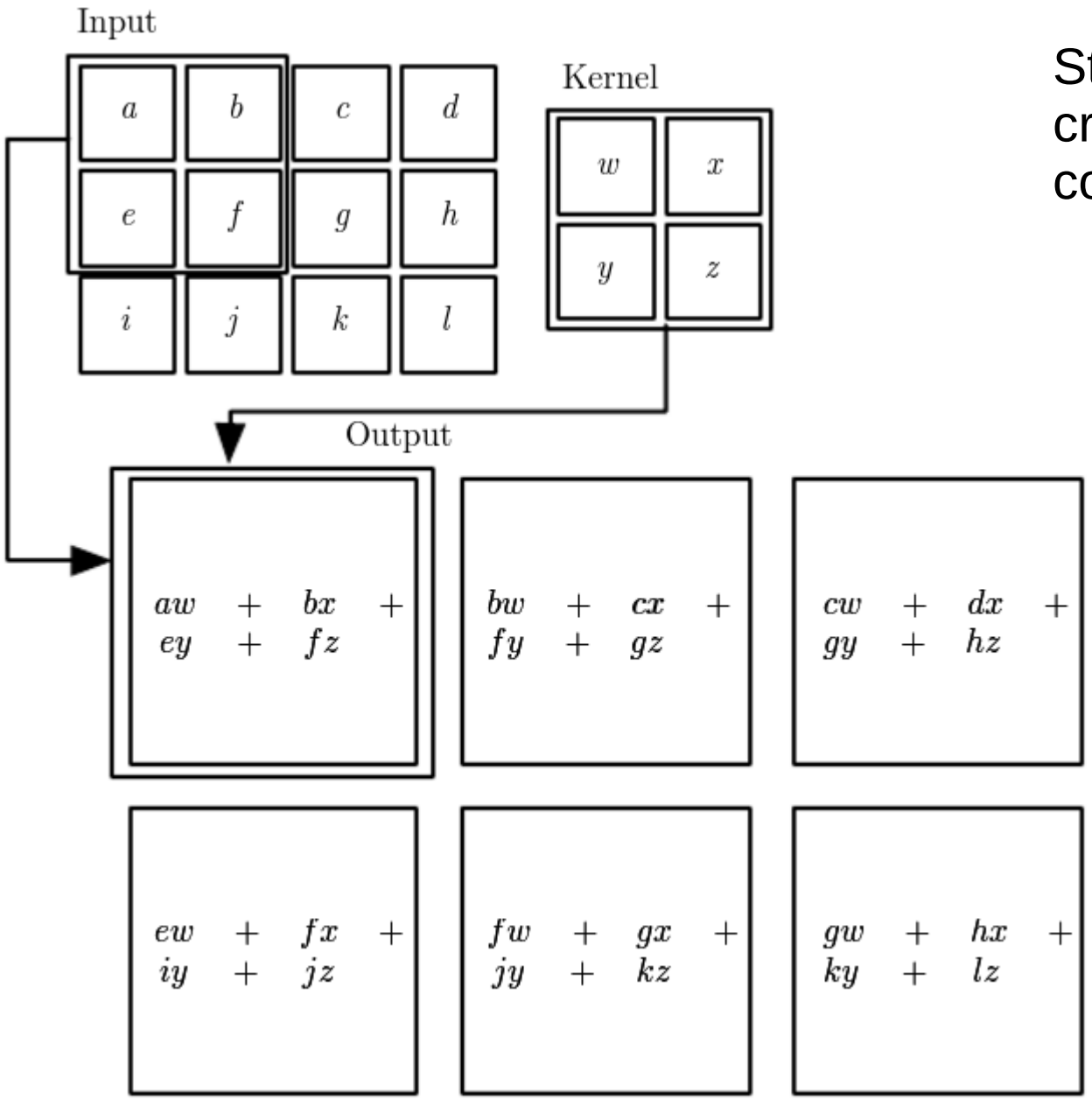
In 2D

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n)$$

This is the formula for cross-correlation in 2D.

Many machine learning libraries implement cross-correlation but call it convolution.

Convolution example

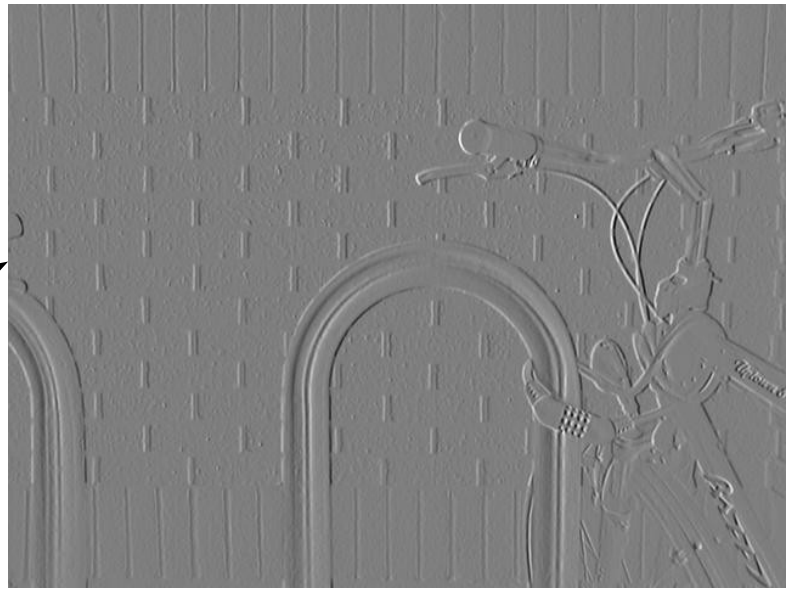


Strictly speaking, this is a cross-correlation, not convolution.

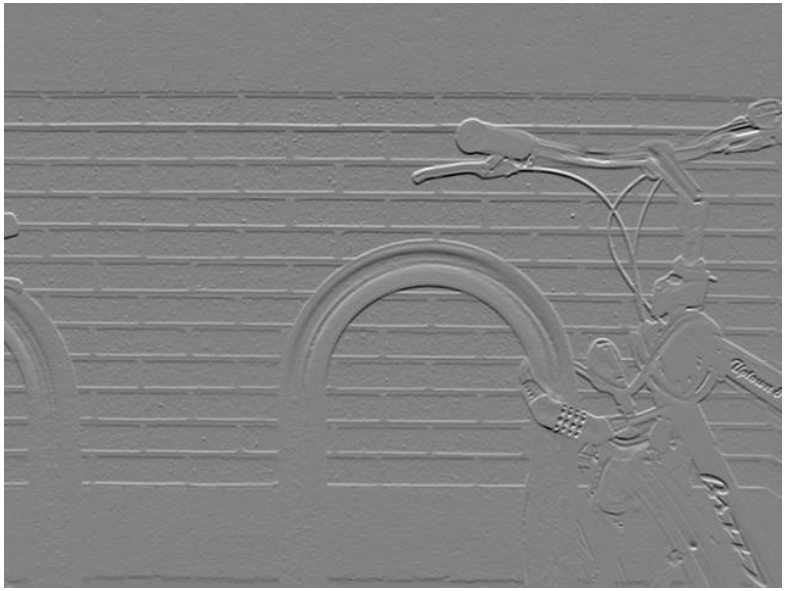
Figure 9.1 from Goodfellow et al. (2016).

Convolution example

$$\begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$



$$\begin{bmatrix} -1 & -2 & +1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$



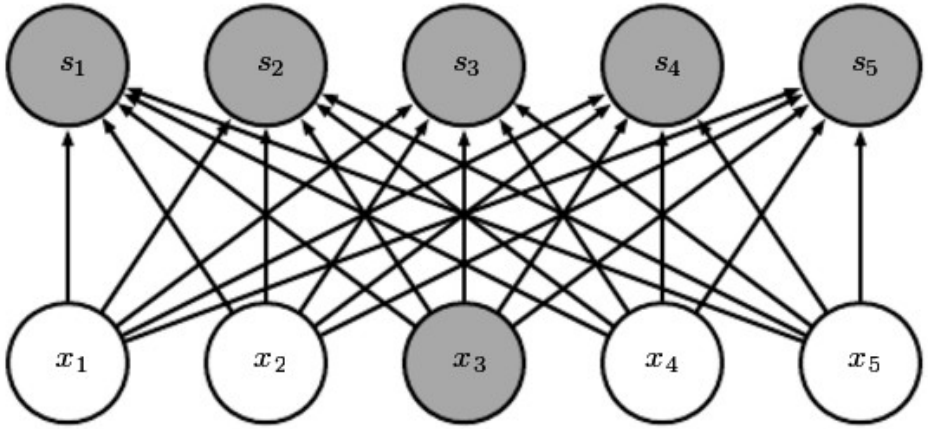
[Figures from https://en.wikipedia.org/wiki/Sobel_operator]

Motivation behind ConvNets

1) Sparse interactions
2) Parameter sharing
3) Equivariant representations ***(& PARTIAL SPATIAL INVARIANCES)***
4) Ability to process inputs of variable sizes

1) Sparse interactions

In a usual ANN, nodes are fully-connected



In CNN, sparse connections:

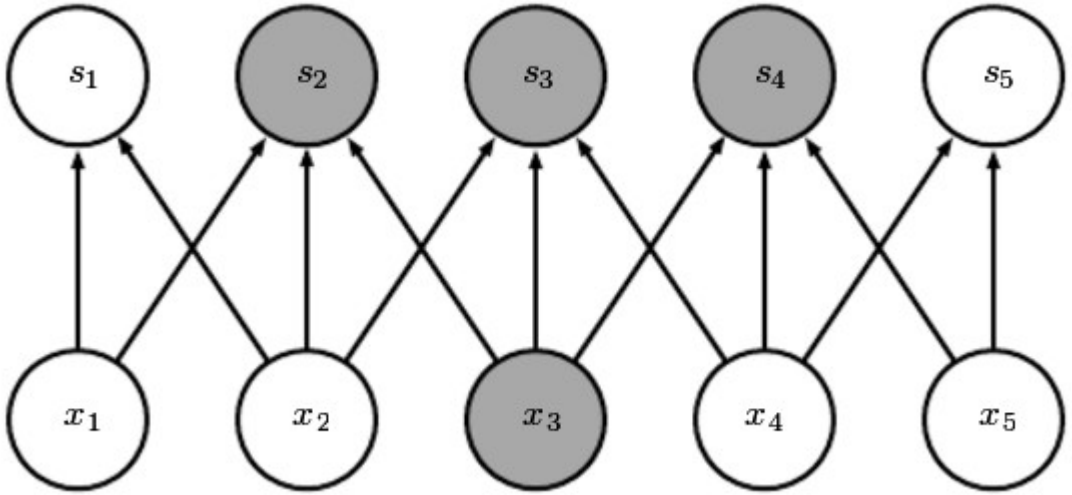
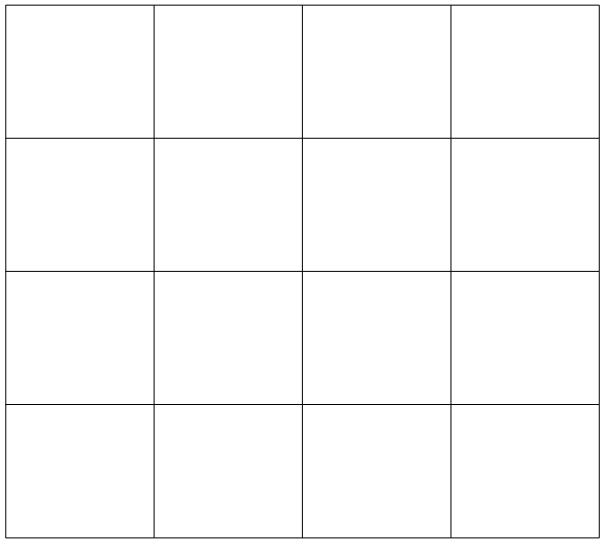
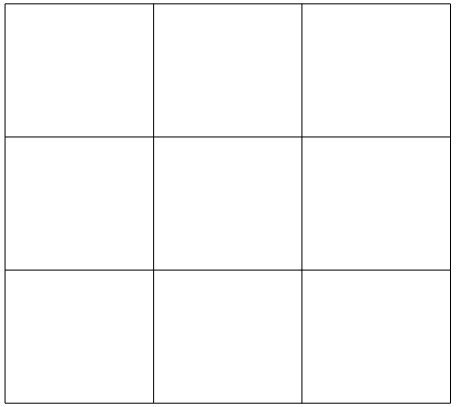


Figure 9.2 from Goodfellow et al. (2016).

Sparse interactions



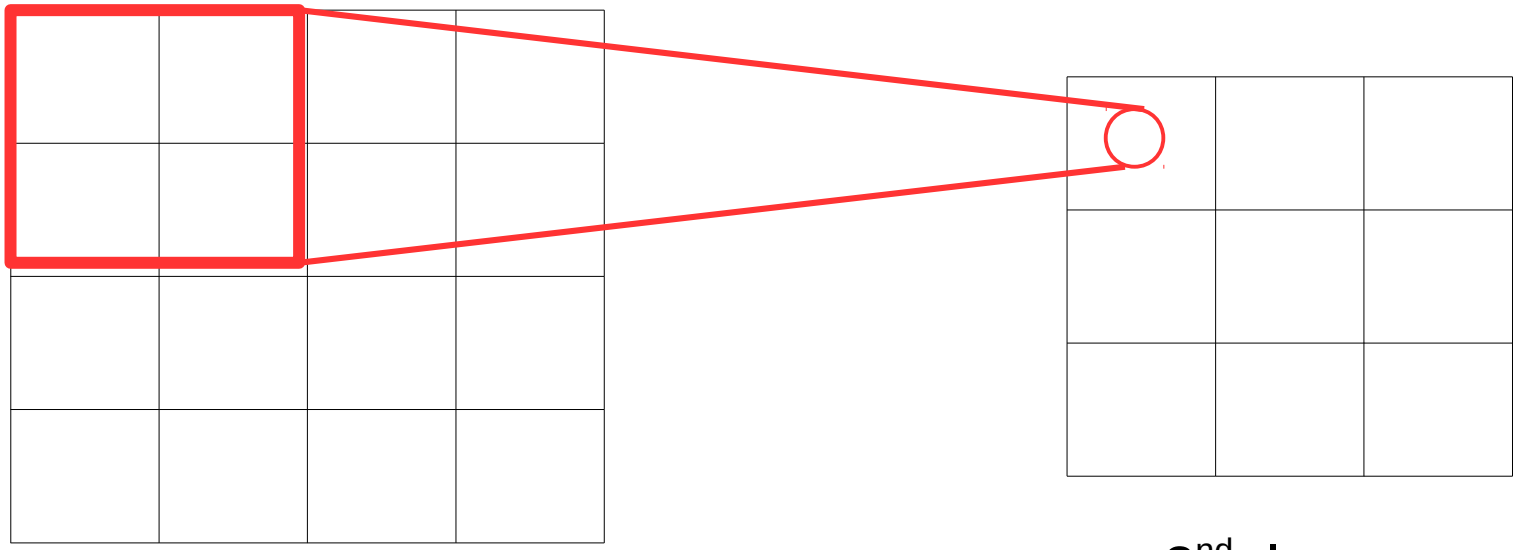
1st (input) layer: 4x4 image



2nd layer

2x2 filter

Sparse interactions

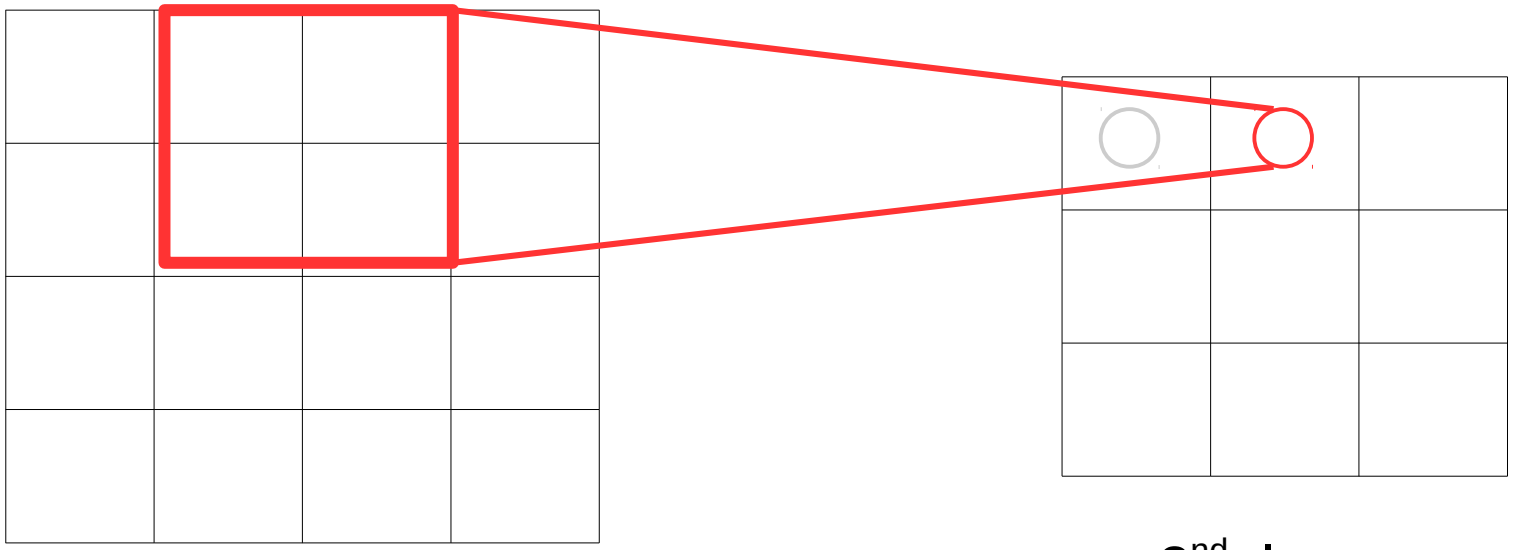


2^{nd} layer

1^{st} (input) layer: 4x4 image

Node in the 2^{nd} layer is not fully-connected to the nodes in the 1^{st} layer.

Sparse interactions

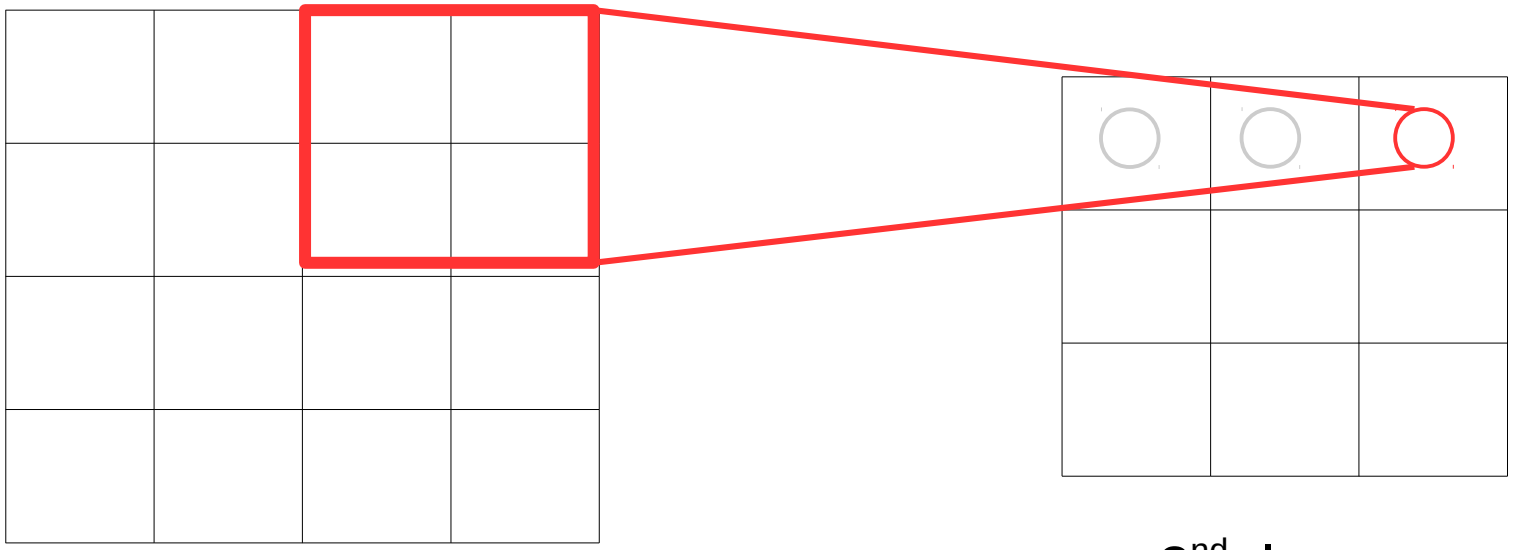


2^{nd} layer

1^{st} (input) layer: 4x4 image

: computed

Sparse interactions

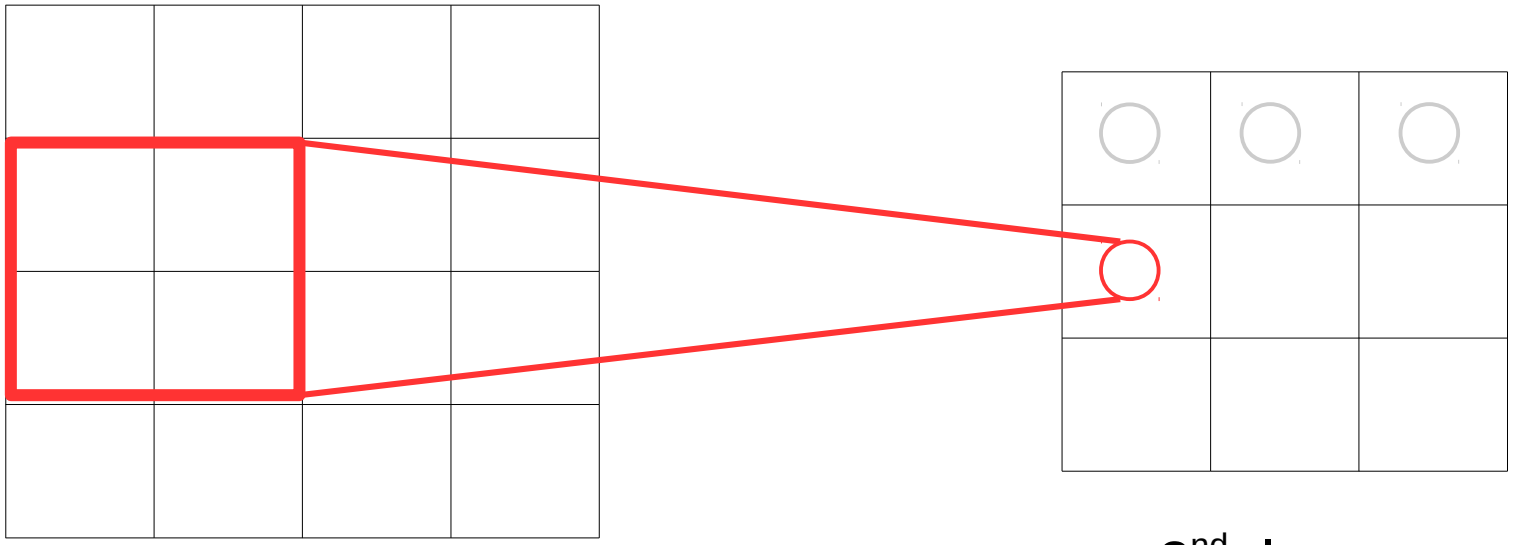


2^{nd} layer

1^{st} (input) layer: 4x4 image

: computed

Sparse interactions

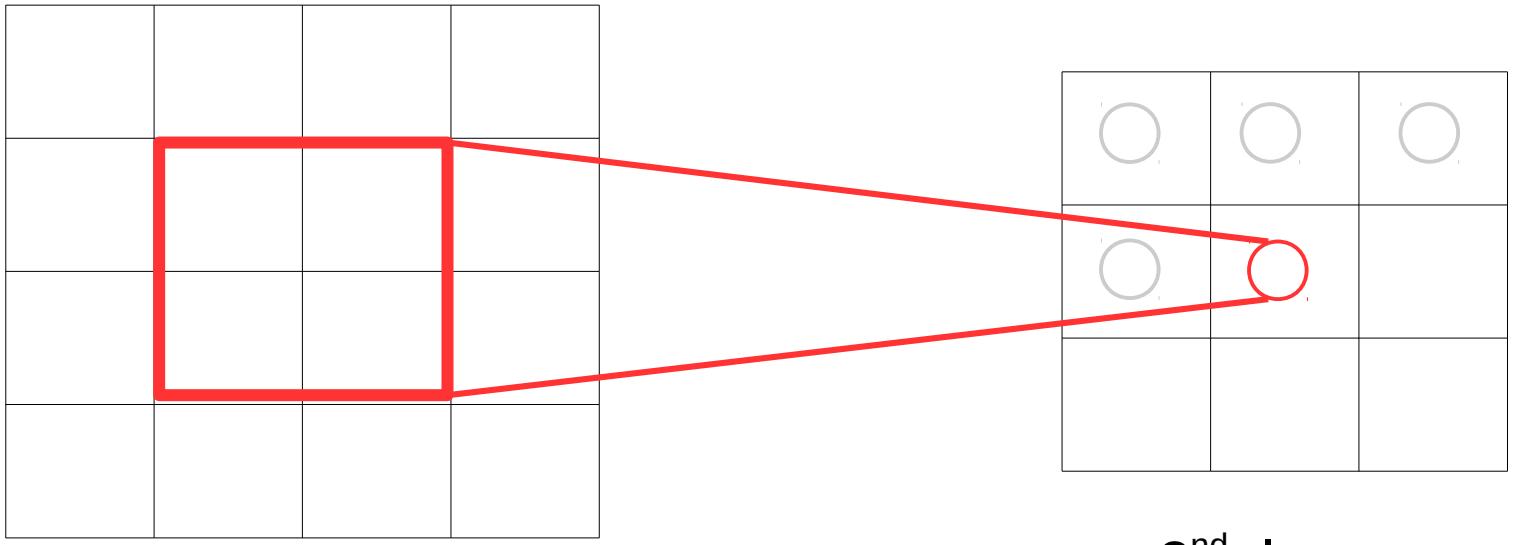


2^{nd} layer

1^{st} (input) layer: 4x4 image

○: computed

Sparse interactions

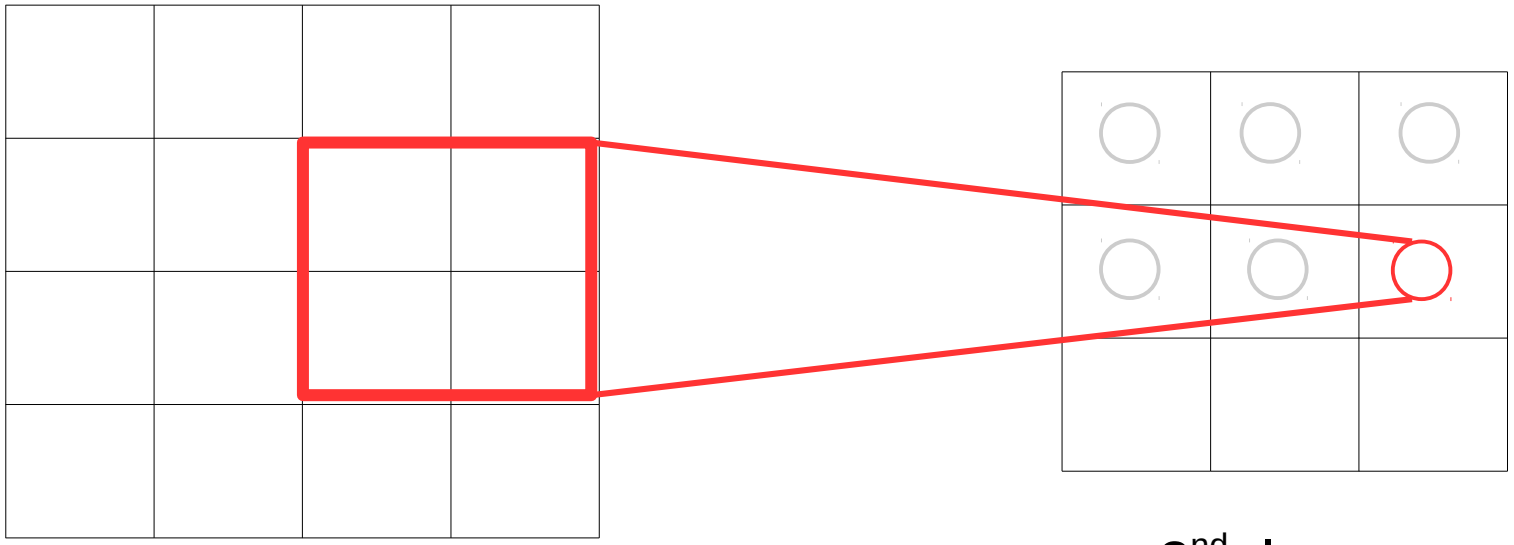


2^{nd} layer

1^{st} (input) layer: 4x4 image

: computed

Sparse interactions



2nd layer

1st (input) layer: 4x4 image

: computed

Sparse interactions

Complexity of fully-connected vs sparse:

m: # of nodes in the 1st layer
n: # of nodes in the 2nd layer
k: # of elements in the filter

Fully-connected: ?
Sparse: ?

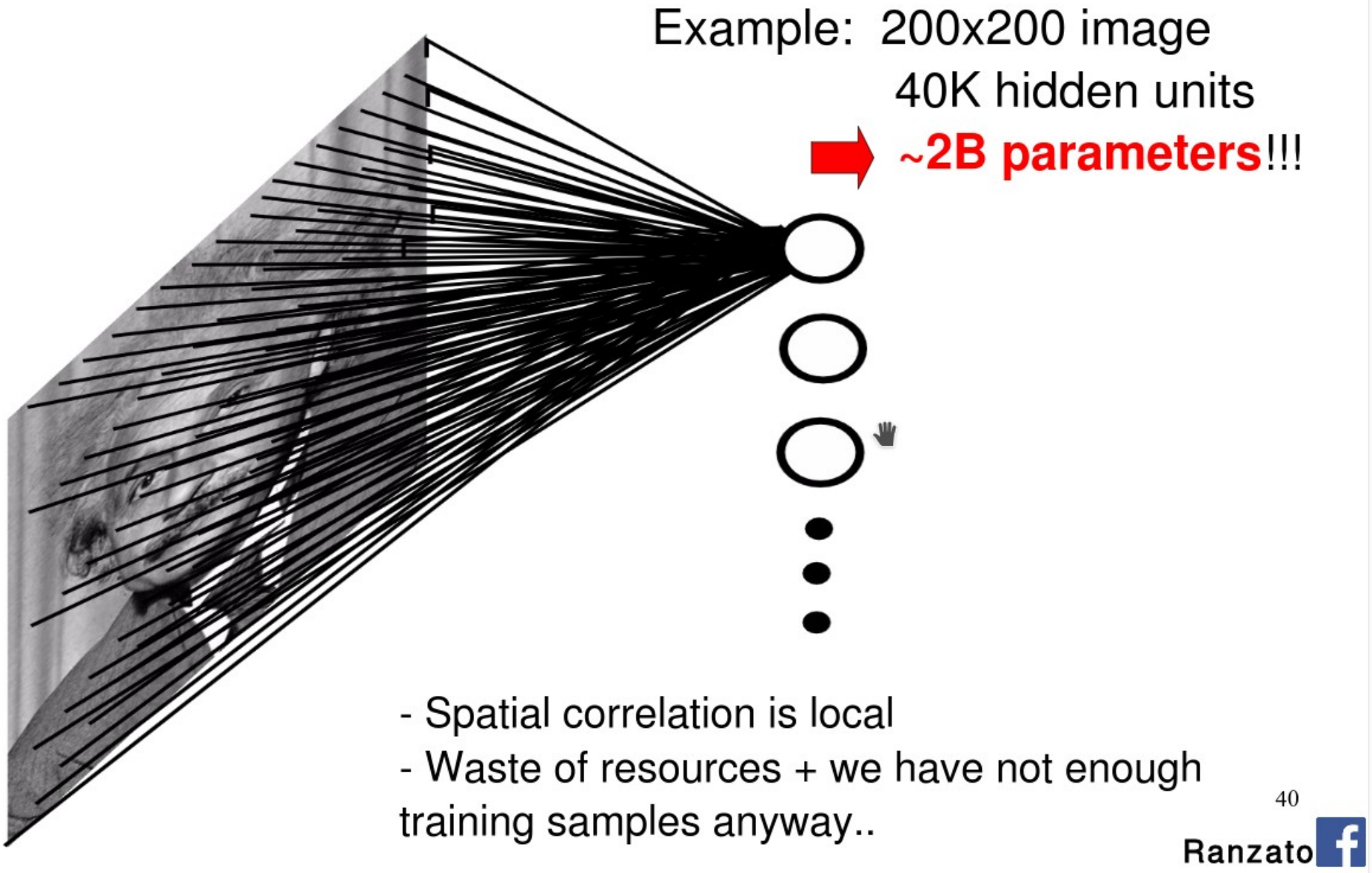
Sparse interactions

Complexity of fully-connected vs sparse:

m: # of nodes in the 1st layer
n: # of nodes in the 2nd layer
k: # of elements in the filter

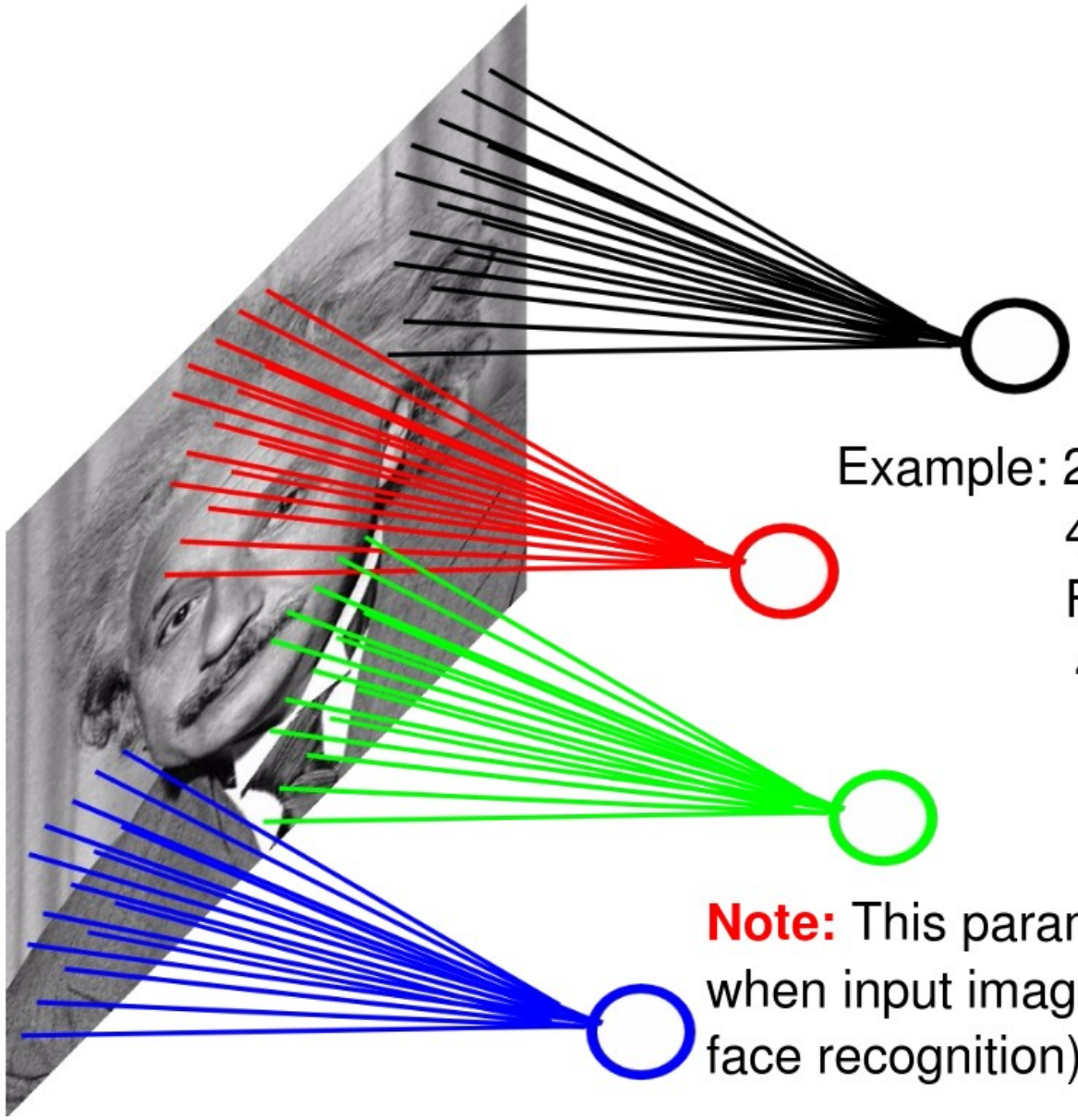
Fully-connected: O(*mn*)
Sparse: O(*nk*) where, typically, $k \ll m$

Fully Connected Layer



[Slide by Marc'Aurelio Ranzato from his Deep Learning Tutorial at CVPR 2014 link]

Locally Connected Layer



Example: 200x200 image
40K hidden units
Filter size: 10x10
4M parameters

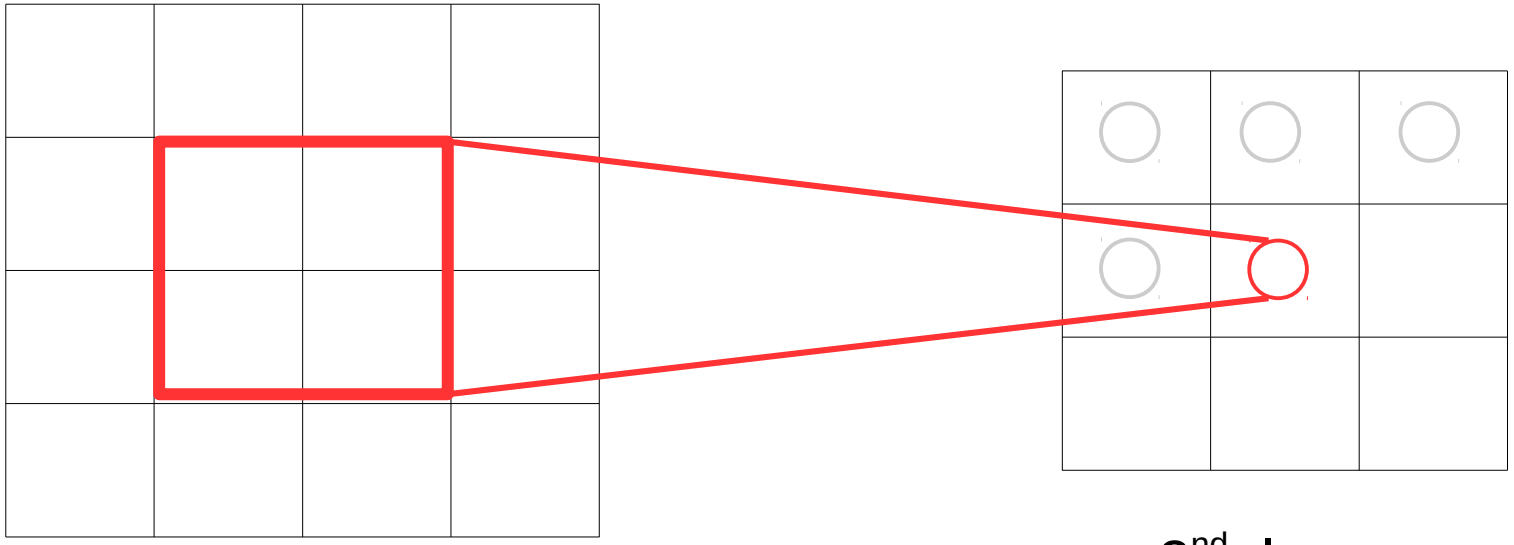
Note: This parameterization is good when input image is registered (e.g., face recognition).

41

Ranzato

[Slide by Marc'Aurelio Ranzato from his Deep Learning Tutorial at CVPR 2014 link]

2) Parameter Sharing



1st (input) layer: 4x4 image

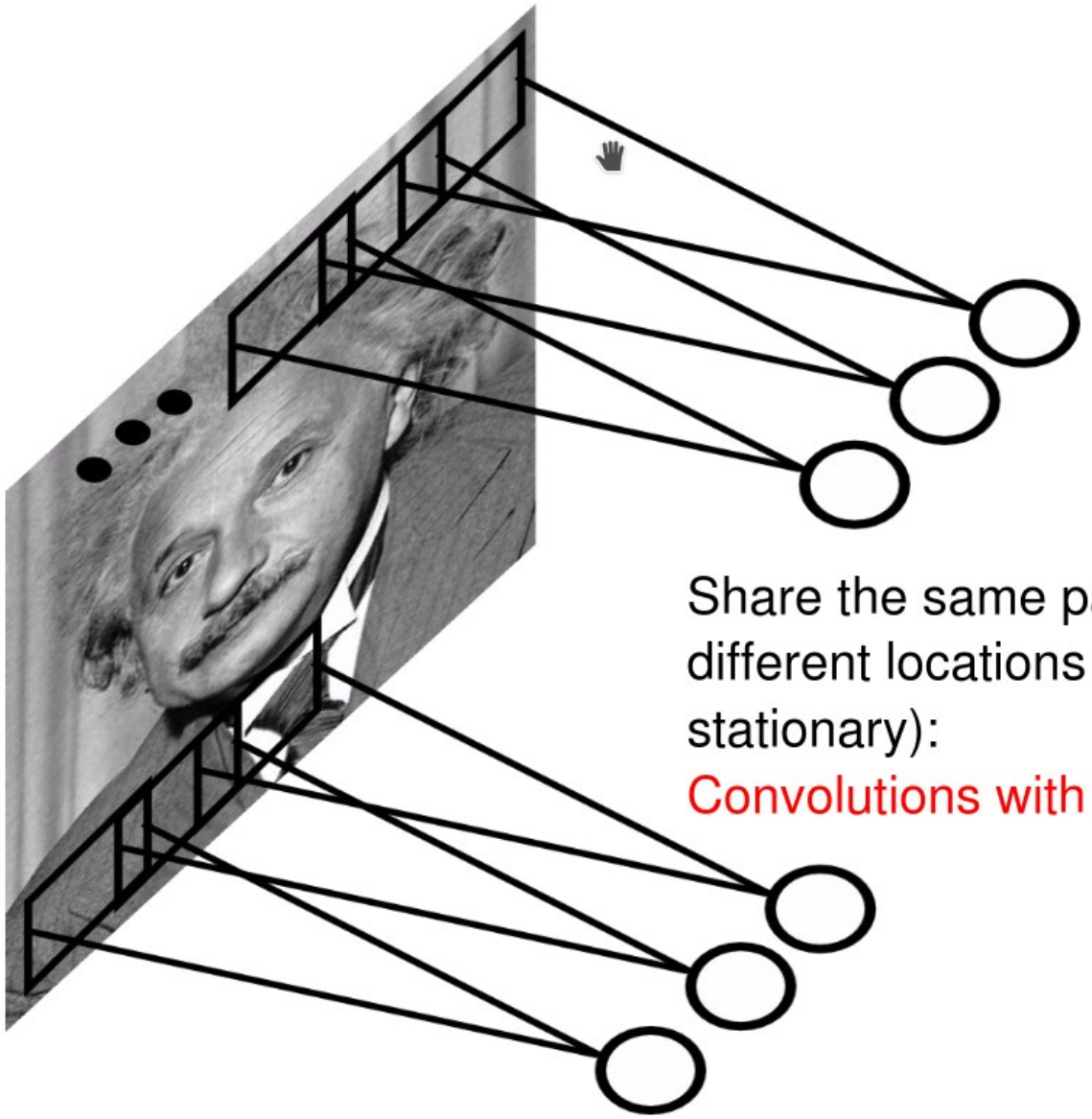
2nd layer

Same kernel/filter (the red window) is applied everywhere on a layer.

# of total parameters to be learned and storage requirements dramatically reduced.

Note *m* and *n* are roughly the same, but *k* is much less than *m*.

Convolutional Layer



[Slide by Marc'Aurelio Ranzato from his Deep Learning Tutorial at CVPR 2014 link]

3) Equivariance

This is a direct consequence of parameter sharing.

A function *f* is equivariant to function *g* if *f(g(x)) = g(f(x)).*

Convolution on images creates a 2D map of where certain features appear in the input. If we move the object in the input, its representation will move the same amount in the output.

Useful when detecting structures that are common in the input. E.g. edges in an image.

4) Ability to process arbitrary sized inputs

Fully-connected networks accept fixed-size input vector.

Recurrent Neural Networks (RNN)

- So far, we have seen MLPs and CNNS
- CNNs are typically used for grid data
- RNNs are for processing ***sequential data***
- Central idea: parameter sharing (through time)
  - If separate parameters for different time indices:
    - Cannot generalize to sequence lengths not seen during training
  - So, parameters are shared across several time steps
- Major difference from MLP and CNNs: RNNs have **cycles** (i.e. feedback/recurrent connections)

$$h^{(3)} = f\left(f\left(f\left(h^{(0)}, x^{(1)}; \theta\right), x^{(2)}; \theta\right), x^{(3)}; \theta\right)$$

h: hidden state
theta: (shared) parameters

t: time
x: input

The network maps the whole input $x^{(1)}, x^{(2)}, \ldots, x^{(t)}$

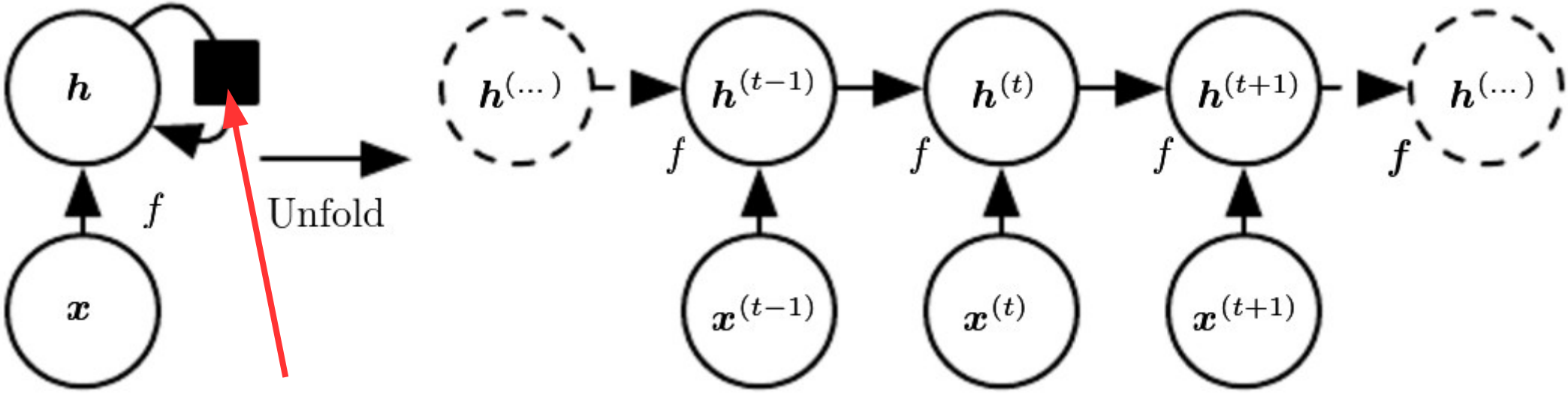
to $h^{(t)}$.

$$\boldsymbol{h}^{(t)} = f\left(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}\right)$$

Folded representation and unfolding

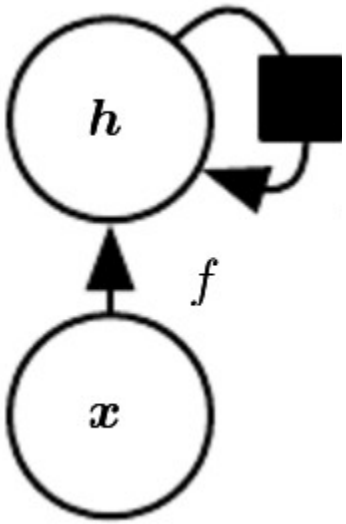
$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta})$$

Two different ways of drawing above equation:



This means a single time-step

[Fig. 10.2 from Goodfellow et al. (2016)]



[Fig. 10.2 from Goodfellow et al. (2016)]

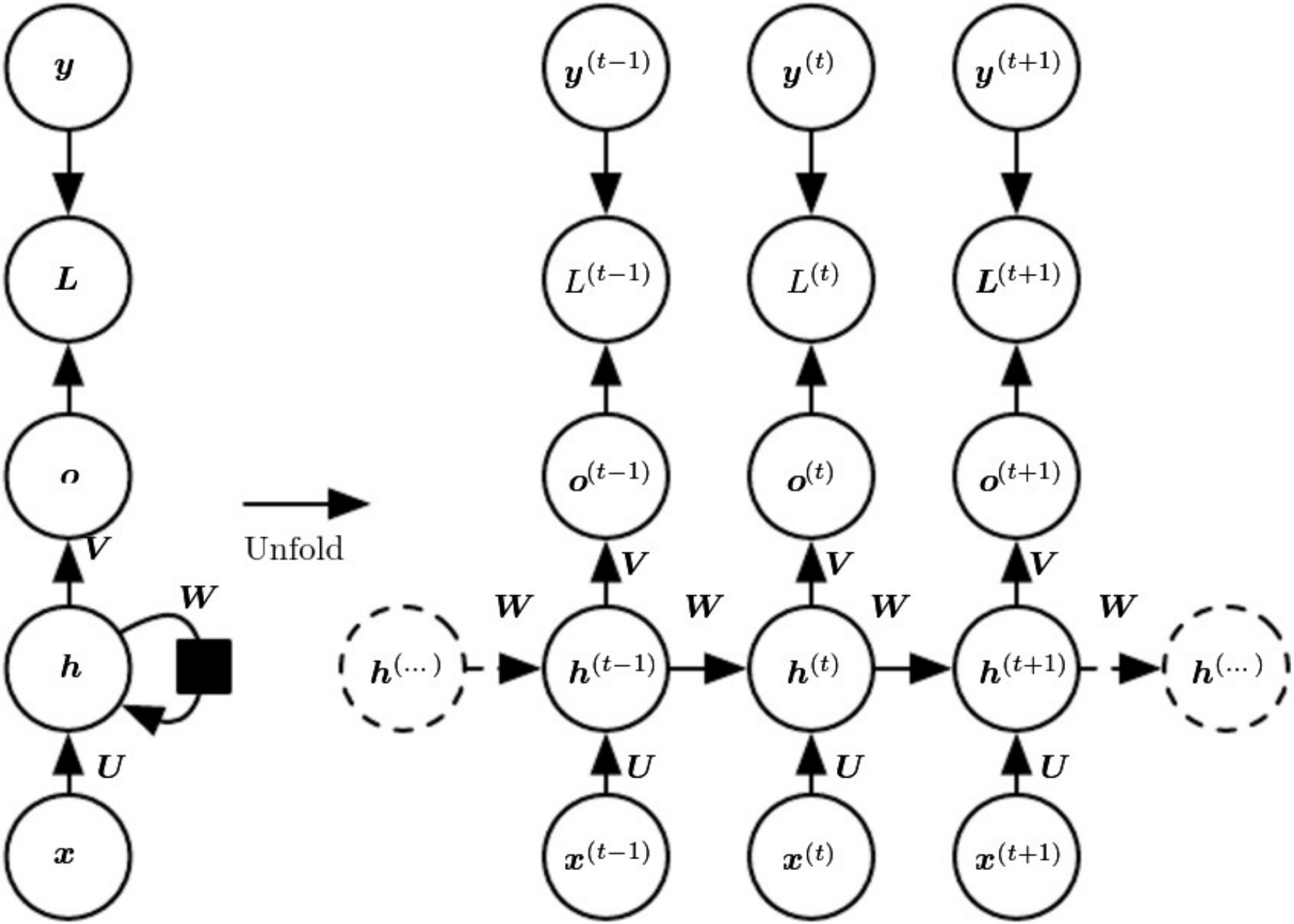
$$h^{(t)} = f\left(h^{(t-1)}, x^{(t)}; \theta\right)$$

$$= g^{(t)}\left(x^{(1)}, x^{(2)}, \ldots, x^{(t)}; \theta\right)$$

The whole input, $x^{(1)}$ to $x^{(t)}$, is of arbitrary length but $h^{(t)}$ is fixed length.

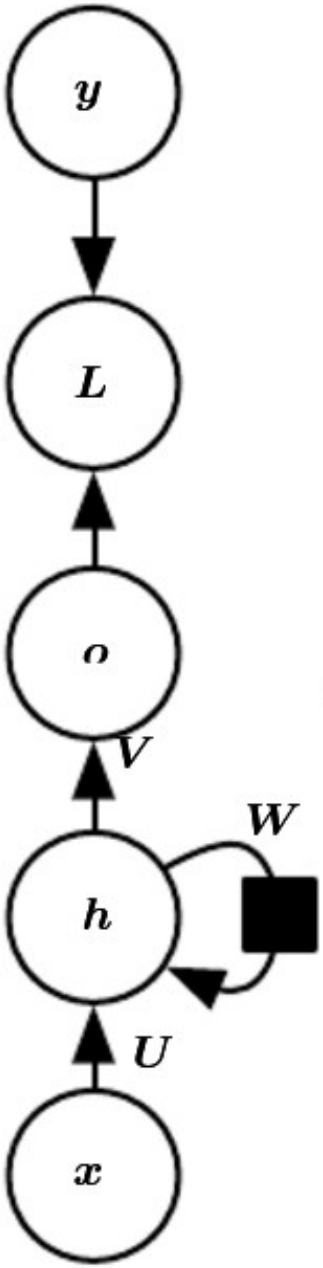
So, $\boldsymbol{h^{(t)}}$ **is a lossy summary of** the task-relevant aspects of $\boldsymbol{x^{(1)}}$ **to** $\boldsymbol{x^{(t)}}$.

A recurrent network that maps input sequence ***x*** to output sequence ***o***, using a loss function *L* and label sequence ***y***.



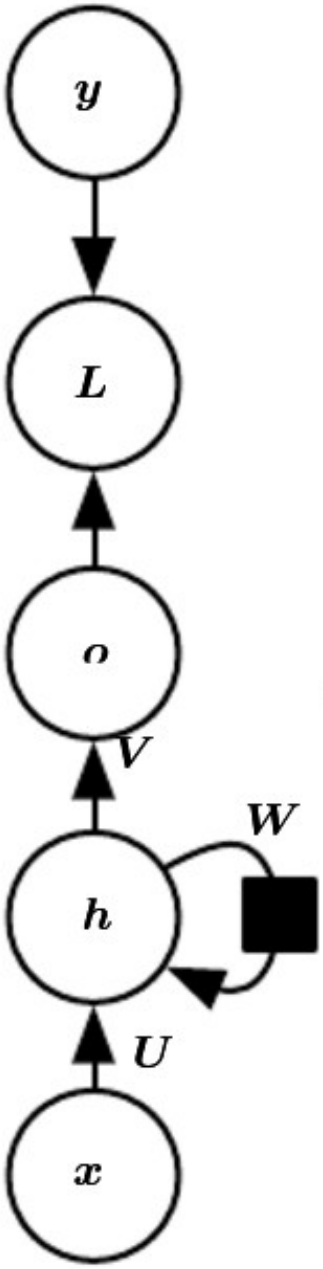
[Fig. 10.3 from Goodfellow et al. (2016)]

A recurrent network that maps input sequence ***x*** to output sequence ***o***, using a loss function *L* and label sequence ***y***.



This RNN is universal in the sense that any function computable by a Turing machine can be computed by such a recurrent network of a finite size.

A recurrent network that maps input sequence ***x*** to output sequence ***o***, using a loss function *L* and label sequence ***y***.

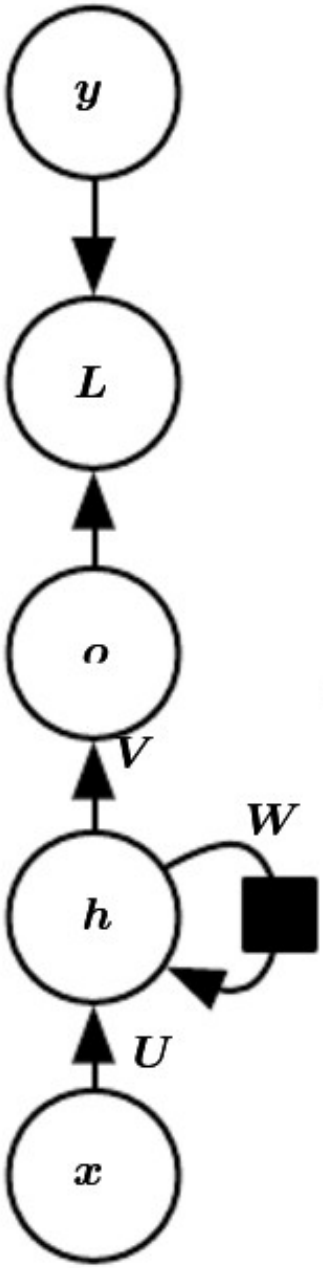


Update equations:

$$
\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \mathrm{softmax}(\boldsymbol{o}^{(t)})
\end{aligned}
$$

Note: *tanh( ), softmax( )* are just example choices.

A recurrent network that maps input sequence ***x*** to output sequence ***o***, using a loss function *L* and label sequence ***y***.

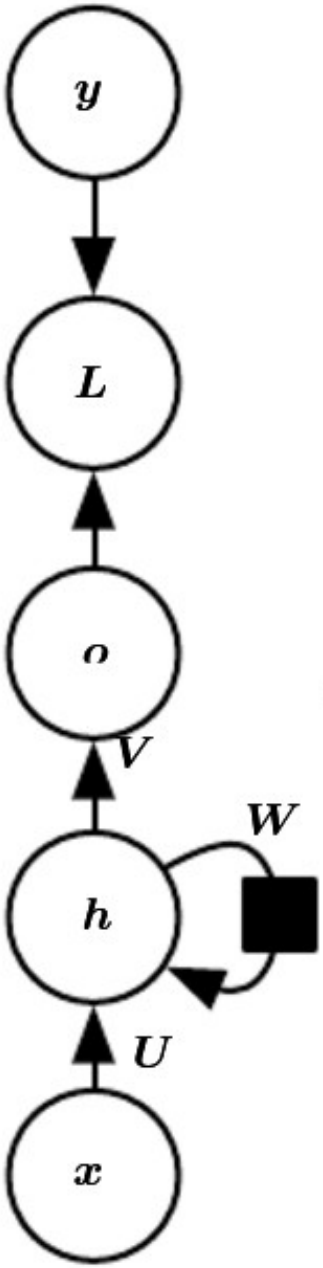


Total loss:

$$L\left(\{\boldsymbol{x}^{(1)},\ldots,\boldsymbol{x}^{(\tau)}\},\{\boldsymbol{y}^{(1)},\ldots,\boldsymbol{y}^{(\tau)}\}\right)$$
$$=\sum_t L^{(t)}$$
$$=-\sum_t \log p_{\text{model}}\left(y^{(t)}\mid\{\boldsymbol{x}^{(1)},\ldots,\boldsymbol{x}^{(t)}\}\right),$$

Negative log-likelihood loss, just as an example

A recurrent network that maps input sequence ***x*** to output sequence ***o***, using a loss function *L* and label sequence ***y***.



Total loss:

$$L\left(\{\boldsymbol{x}^{(1)},\ldots,\boldsymbol{x}^{(\tau)}\},\{\boldsymbol{y}^{(1)},\ldots,\boldsymbol{y}^{(\tau)}\}\right)$$
$$=\sum_t L^{(t)}$$
$$=-\sum_t \log p_{\text{model}}\left(y^{(t)}\mid\{\boldsymbol{x}^{(1)},\ldots,\boldsymbol{x}^{(t)}\}\right),$$

Negative log-likelihood loss, just as an example

Gradient of *L* w.r.t. model parameters?
Backpropagation through time.

References

- Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. Mathematics of control, signals and systems, 2(4), 303-314.
- Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., and Bengio, Y. (2013a).Maxout networks. In S. Dasgupta and D. McAllester, editors, ICML'13 , pages 1319–1327
- Hubel, D. H. (1995). Eye, brain, and vision. Scientific American Library/Scientific American Books. [Available online: http://hubel.med.harvard.edu/book/bcontex.htm]
- Jacobs, R. A. (1988). Increased rates of convergence through learning rate adaptation.Neural networks, 1(4), 295–307. Mohamed, A. R., Dahl, G., & Hinton, G. (2009, December).
- Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. Psychological review, 65(6), 386.
- Wu, Y. and He, K., (2018). Group normalization. In Proceedings of the European Conference on Computer Vision (ECCV) (pp. 3-19).

METU Computer Engineering

CEng 796 – Spring 2020

R. Gökberk Cinbiş, Emre Akbaş

Week 2: Background Review

Part I: Deep learning review

Part II: Probability, Random Variables, Bayes Nets (Probabilistic Directed Acyclic Graphical Models)