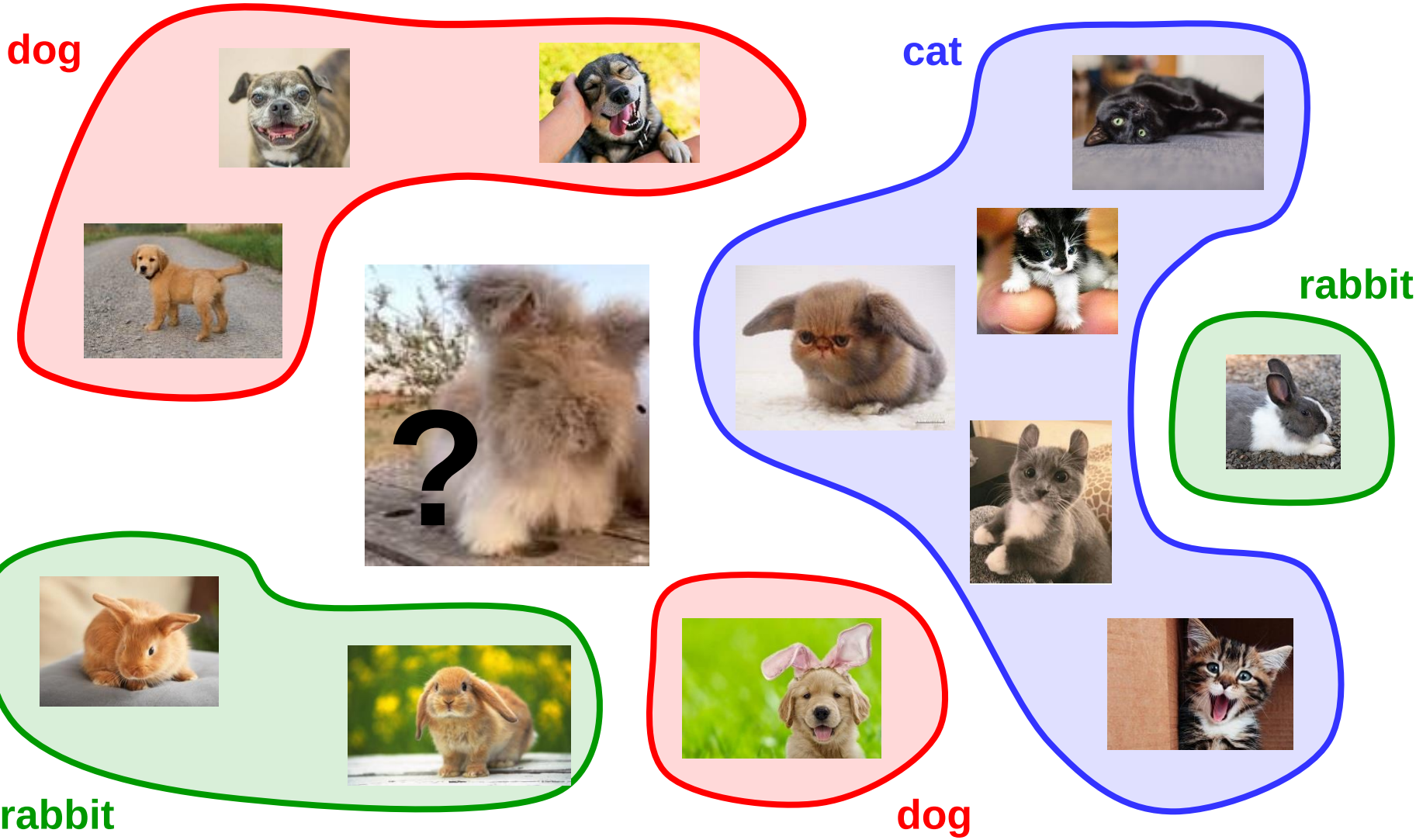
Supervised Classification



Many slides are from Olga Veksler (UW) and Y. LeCun (Facebook)

Supervised Classification (outline)

- Intro to Machine Learning (ML)

for simplicity, discussed in the context of **linear classification**

  - ML types: **supervised**, unsupervised, reinforcement learning
  - Learning quality: overfitting, underfitting, generalization
  - Training and Testing
  - Loss functions: quadratic, cross-entropy
  - Optimization by gradient descent, learning rate, SGD, batches
  - Towards **non-linear classification**
    - kernel methods, multi-layered neural networks (NN)

- Convolutional Neural Networks (CNNs)

  - Convolutional and pooling layers
  - ReLU, drop-out, normalization, batch-normalization, etc
  - Weights regularization
  - Optimization by backpropagation

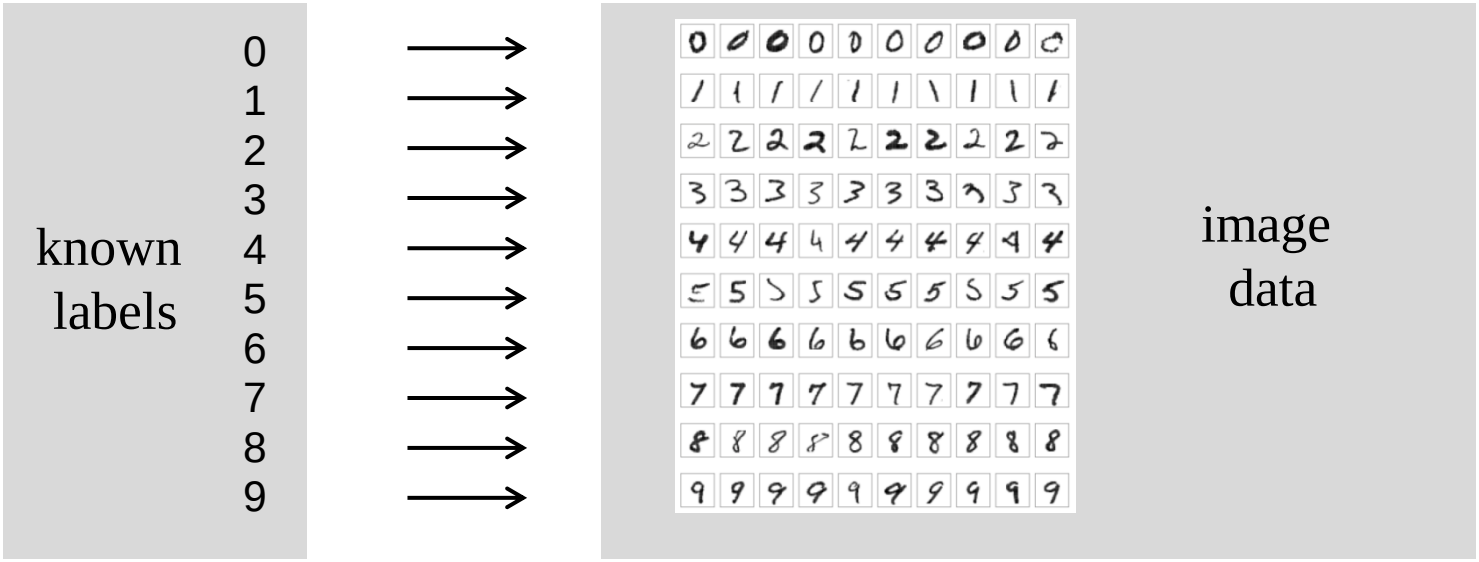
Intro to Machine Learning (ML)

- supervised **linear classification**
  - perceptron, single layer NNs

- towards **non-linear classification**
  - multi-layer NNs

Example: supervised digit recognition

- Easy to collect images of digits with their correct labels



- **ML algorithm** can use collected data to produce a program for recognizing previously unseen images of digits


new image

→ 0

automatically produced label


new image

→ 4

automatically produced label

Types of Machine Learning

focus of this topic

Supervised Learning

our digit recognition example

- given training examples with corresponding correct outputs, also called *label*, *target*, *class*, *answer*, etc.
- learn to produces correct output for a new example

Unsupervised Learning

many of our examples in topic 9

- given unlabeled training examples
- discover good data representation
  - e.g. “natural” clusters
- K-means is the most widely known example
- weak-supervision allows partially labeled training examples

Reinforcement Learning

- learn to select action that maximizes payoff

Subtypes of supervised ML:

focus of this topic

- **Classification** our digit recognition example
  - output belongs to a finite set
  - example: age $\in$ {baby, child, adult, elder}
  - output is also called *class* or *label*

- **Regression**
  - output is continuous
  - example: age $\in$ [0,130]

- Difference mostly in design of loss functions

Supervised ML

- Have training examples with corresponding outputs/labels
- For example: fish classification - *salmon* or *sea bass*?

feature vector → $\mathbf{x}^1 = \begin{bmatrix} 3.3 \\ 5.7 \end{bmatrix}$ $\mathbf{x}^2 = \begin{bmatrix} 6.3 \\ 8.7 \end{bmatrix}$ $\mathbf{x}^3 = \begin{bmatrix} 2.3 \\ 1.7 \end{bmatrix}$ $\mathbf{x}^4 = \begin{bmatrix} 6.4 \\ 7.0 \end{bmatrix}$

individual features $\mathbf{x}_k^i$
e.g. fish length and brightness


salmon
$\mathbf{y}^1=0$


sea bass
$\mathbf{y}^2=1$


salmon
$\mathbf{y}^3 = 0$


sea bass
$\mathbf{y}^4=1$

- Each example should be represented by *feature vector* $\mathbf{x}^i$
  - data may be given in vector form from the start
  - if not, for each example **i**, extract useful features, put them as a vector
  - fish classification example:
    - extract two features, *fish length* and *average fish brightness*
      (can extract any number of other features)
    - for images, can use raw pixel intensity or color as features

- $\mathbf{y}^i$ is the output (label or target) for example $\mathbf{x}^i$

Supervised ML

- We are given
  1. Training examples $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^n$
  2. Target output for each sample $\mathbf{y}^1, \mathbf{y}^2, \ldots, \mathbf{y}^n$

  } *labeled data*

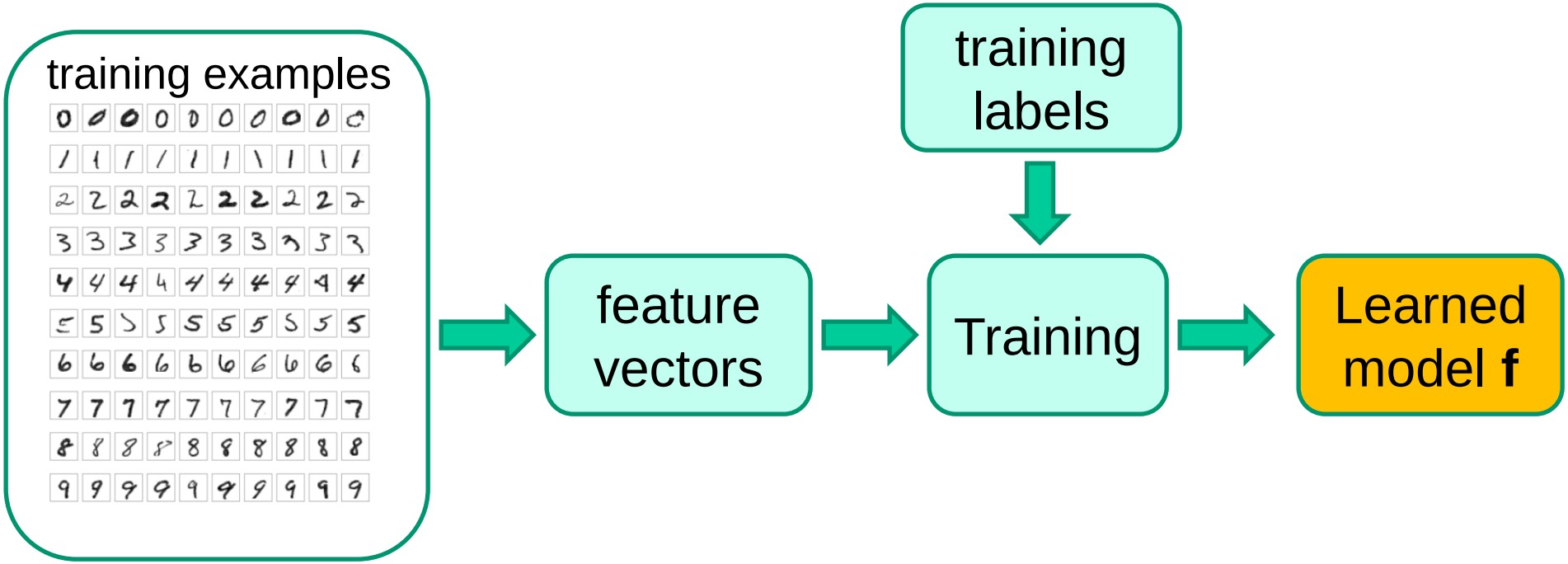
- **Training phase**
  - estimate function $\mathbf{y} = \mathbf{f}(\mathbf{x})$ from labeled data

    where $\mathbf{f}(\mathbf{x})$ is called *classifier*, *learning machine*, *prediction function*, etc.

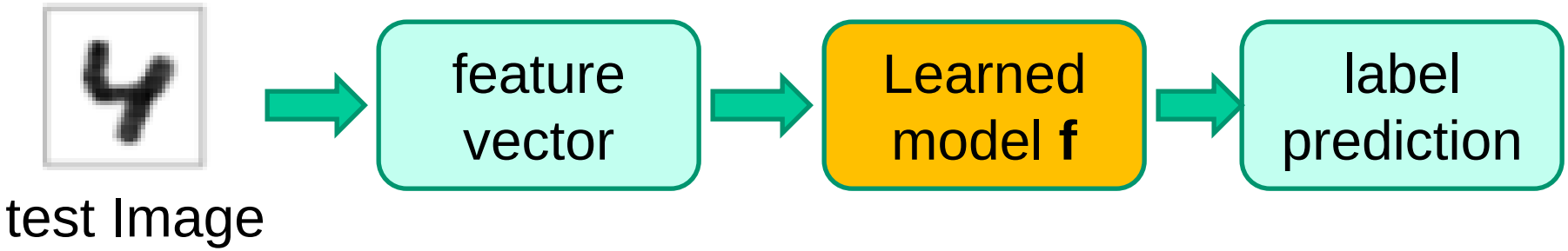
- **Testing phase** (deployment)
  - predict output $\mathbf{f}(\mathbf{x})$ for a new (unseen) sample $\mathbf{x}$

Training/Testing Phases Illustrated

Training



Testing



Training phase as parameter estimation

Estimate prediction function $\mathbf{y} = \mathbf{f}(\mathbf{x})$ from labeled data

Typically, search for **f** is limited to some type/group of classifiers ("*hypothesis space*") parameterized by *weights* **w** that must be estimated

$\mathbf{f_w(x)}$ or $\mathbf{f(w,x)}$ **w = ?**

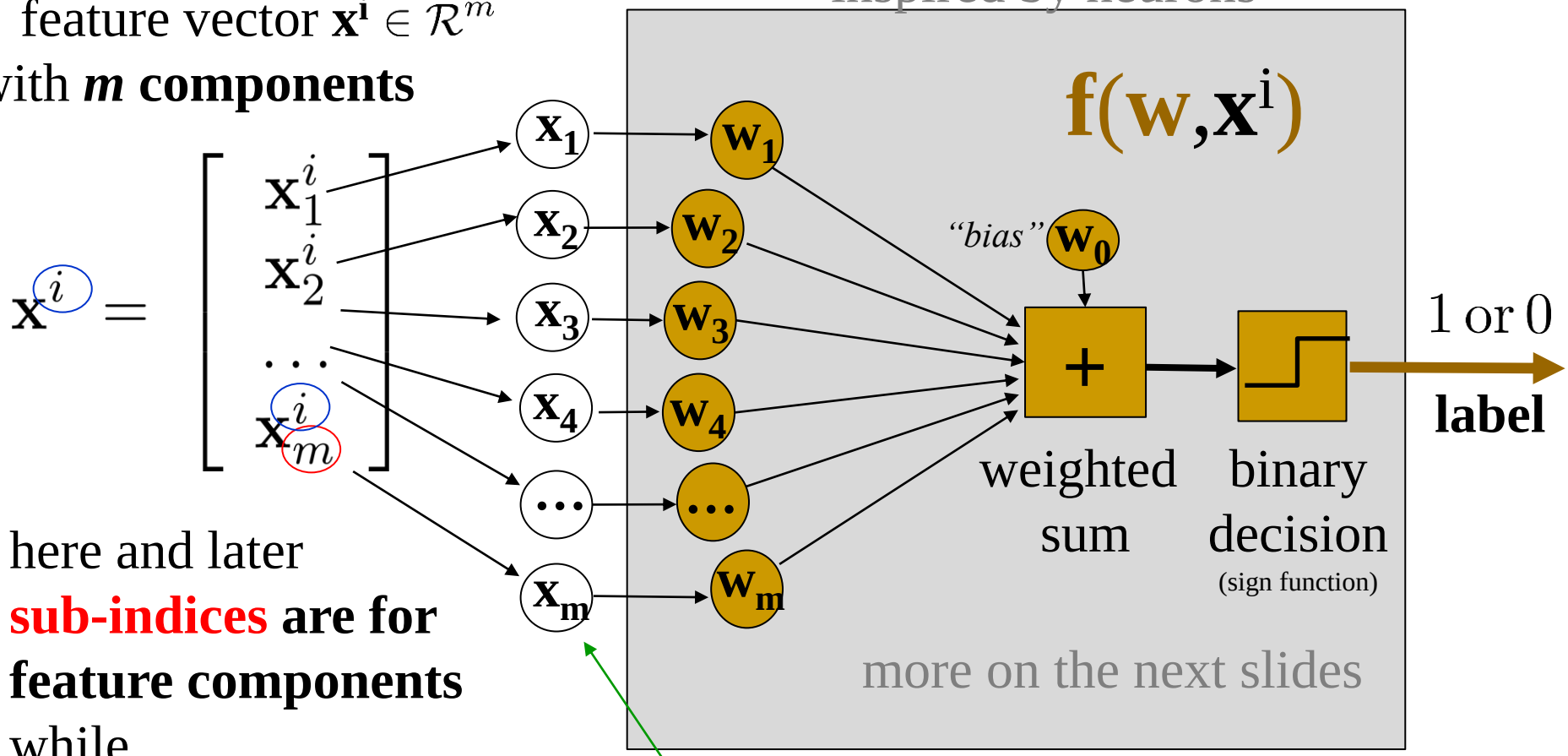
Goal: find classifier parameters (weights) **w** so that $\mathbf{f(w,x^i) = y^i}$ "as much as possible" for all training examples, where "as much as possible" is defined by a *loss function* $\boldsymbol{L(y,f)}$ penalizing $\mathbf{f(w,x^i) \neq y^i}$

$$\mathbf{w^* = \arg\min_w \sum_i L(y^i, f(w,x^i))}$$

Linear classifier example: *perceptron*

m-dimensional feature vector $\mathbf{x}^i \in \mathcal{R}^m$ with ***m* components**

Frank Rosenblatt, 1958
inspired by neurons

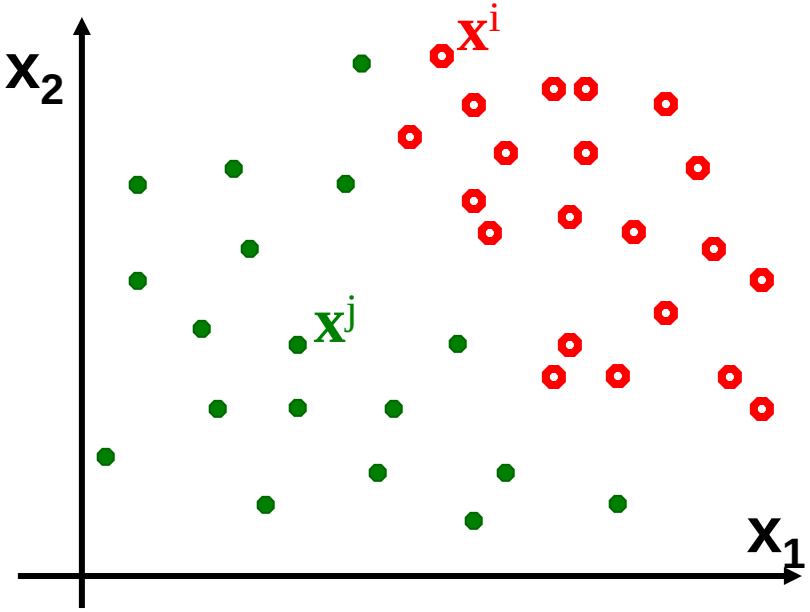


here and later
sub-indices are for feature components
while
super-indices are for data points (feature vectors)

NOTE: for simplicity, we omit **super-indices** (or **sub-indices**) assuming the context is "clear"

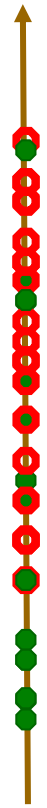
Linear classifier example: *perceptron*

For two class problem and 2-dimensional data (feature vectors)



consider some
linear transformation
from 2D space to 1D

$$w_0 + w_1 x_1 + w_2 x_2$$



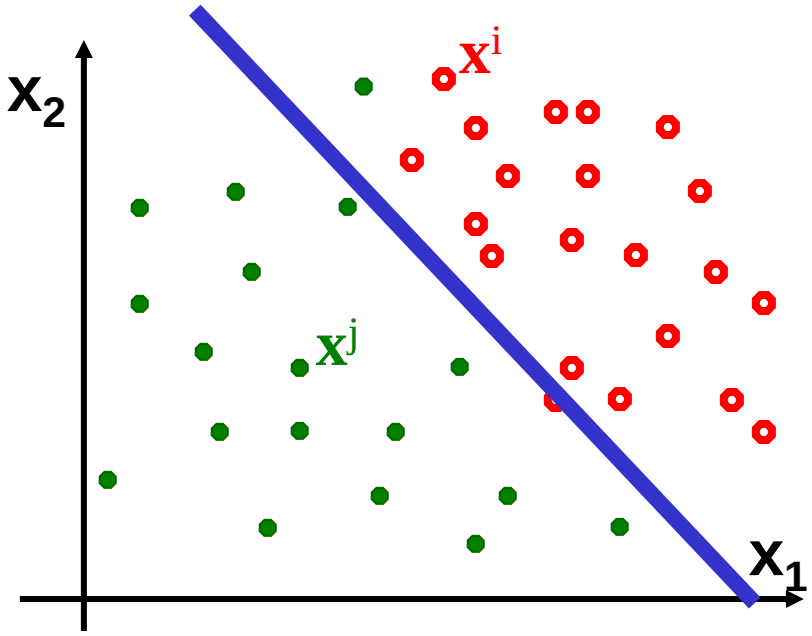
points of two classes can be completely mixed

Question:

Is it possible to find a linear transformation onto 1D so that transformed 1D points can be separated (by a ***threshold***)?

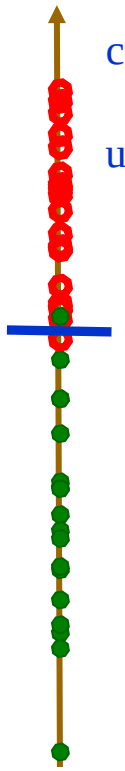
Linear classifier example: *perceptron*

For two class problem and 2-dimensional data (feature vectors)



"good"
linear transformation
from 2D space to 1D

$$w_0^* + w_1^* x_1 + w_2^* x_2$$



NOTE
can always shift threshold to 0 using weight w_0

0

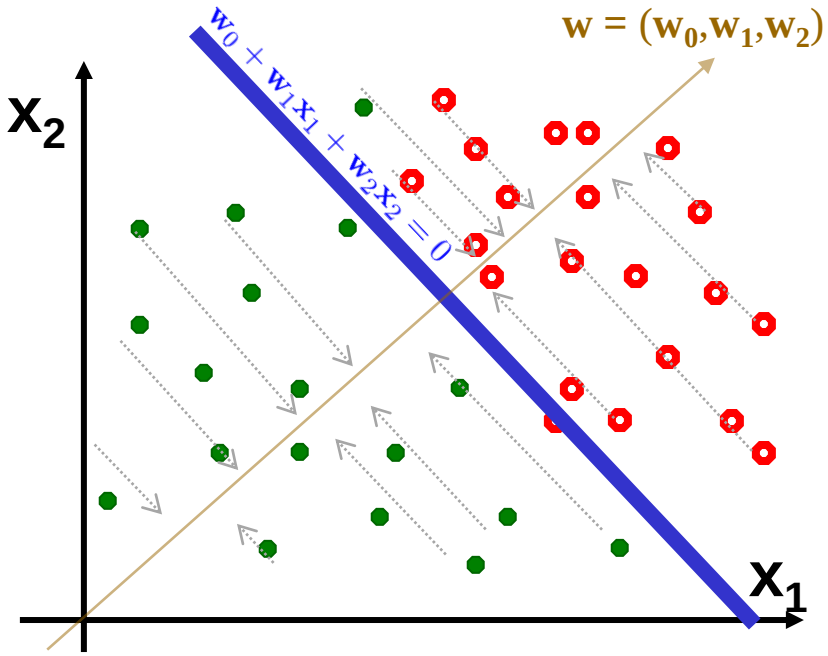
good separation by simple threshold

Answer:

In this case, YES, because the data is linearly separable in the original feature space. So, what is the transformation?

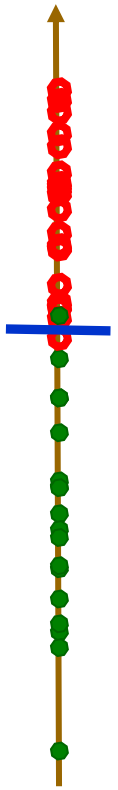
Linear classifier example: *perceptron*

For two class problem and 2-dimensional data (feature vectors)



"good"
linear transformation
from 2D space to 1D

$$w_0^* + w_1^* x_1 + w_2^* x_2$$



good separation by simple threshold

(w_1, w_2) – are the coordinates of the **normal**

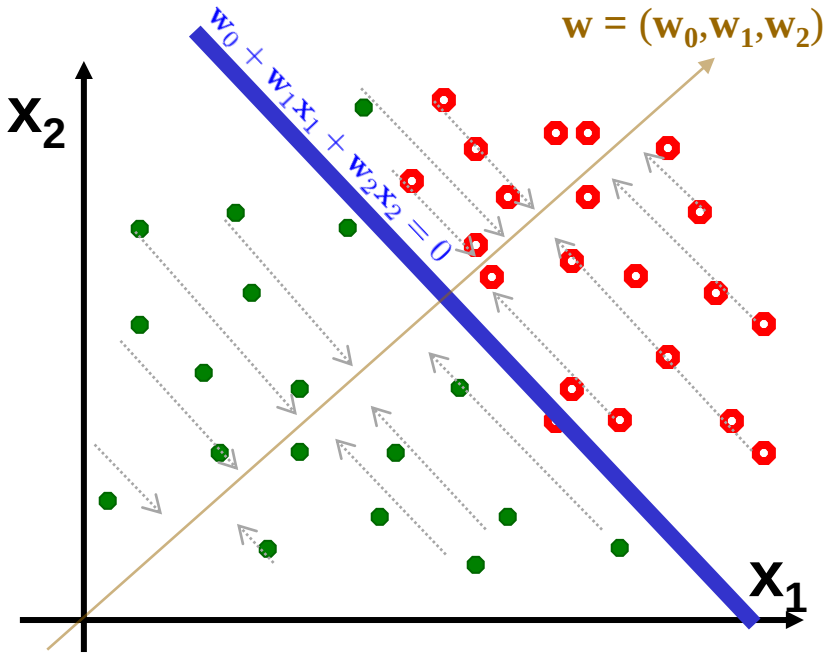
w_0 – is "bias" (shifting threshold to 0)

Answer:

This 2D →1D linear transformation is a projection onto the **normal** of the separating **hyper-plane**.

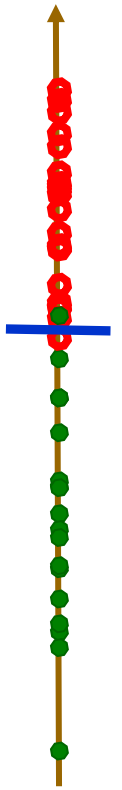
Linear classifier example: *perceptron*

For two class problem and 2-dimensional data (feature vectors)



"good"
linear transformation
from 2D space to 1D

$$w_0^* + w_1^* x_1 + w_2^* x_2$$



good separation by simple threshold

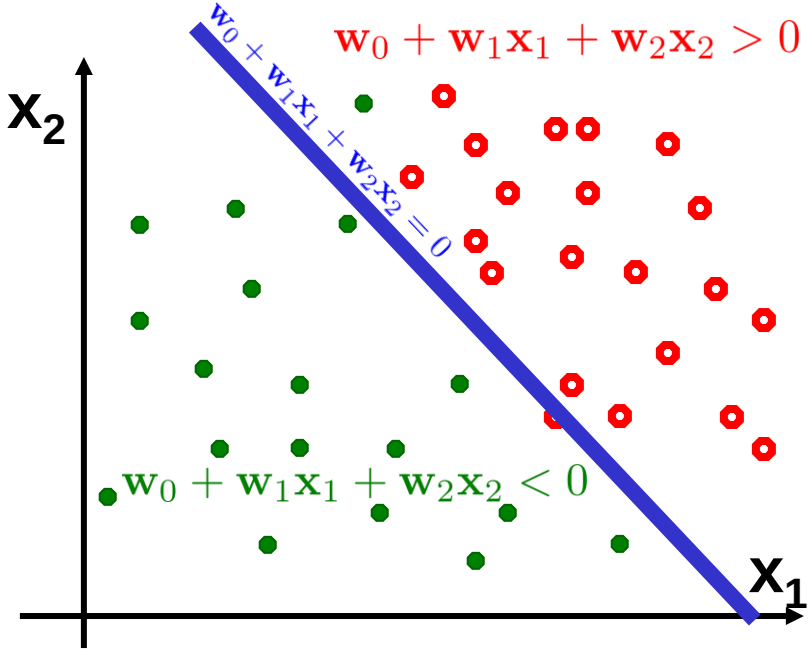
(w_1,w_2) – are the coordinates of the **normal**

w_0 – is "bias" (shifting threshold to 0)

In fact, **any** 2D →1D linear transformation $\mathbf{w} = (w_0,w_1,w_2)$ is a **projection onto normal of some hyper-plane**. So, original question really asks if there is a hyper-plane separating data.

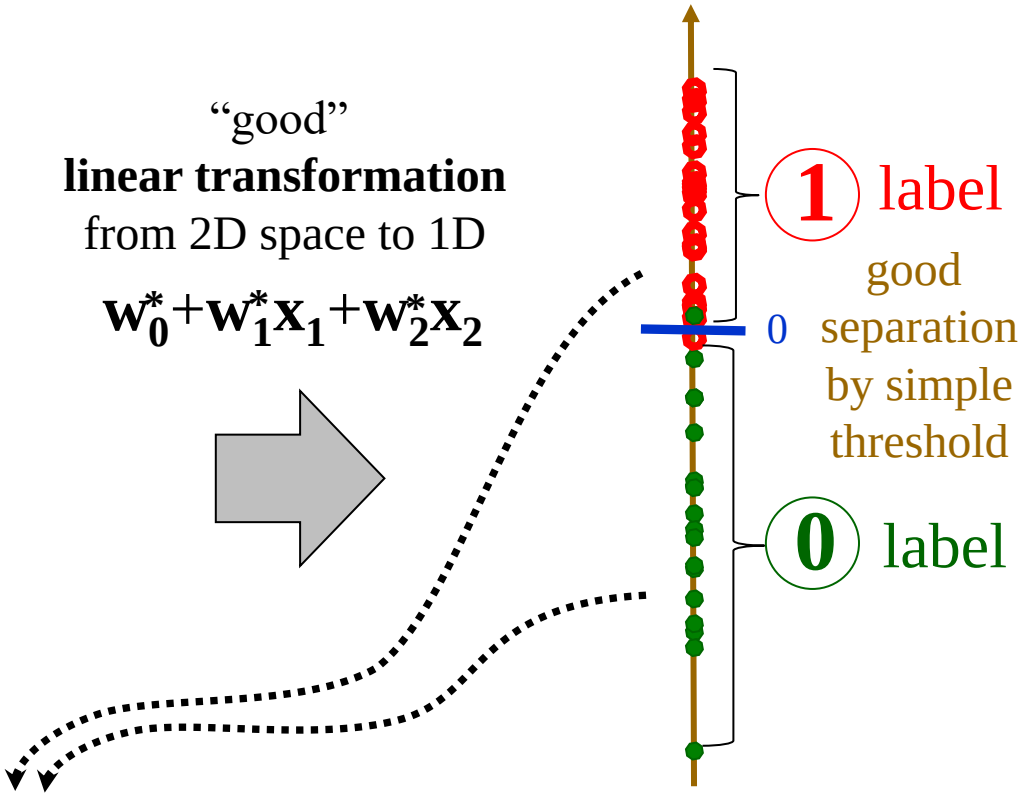
Linear classifier example: *perceptron*

For two class problem and 2-dimensional data (feature vectors)



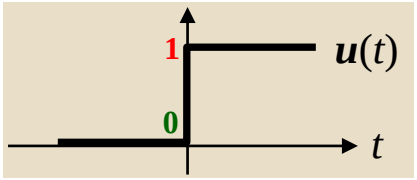
"good"
linear transformation
from 2D space to 1D

$\mathbf{w_0^*+w_1^*x_1+w_2^*x_2}$



thresholding can be formally represented by this prediction function

$$\mathbf{f(w,x)} = u\,(\mathbf{w_0+w_1x_1+w_2x_2}) \qquad \mathbf{f(w,x)} \in \{0,1\}$$

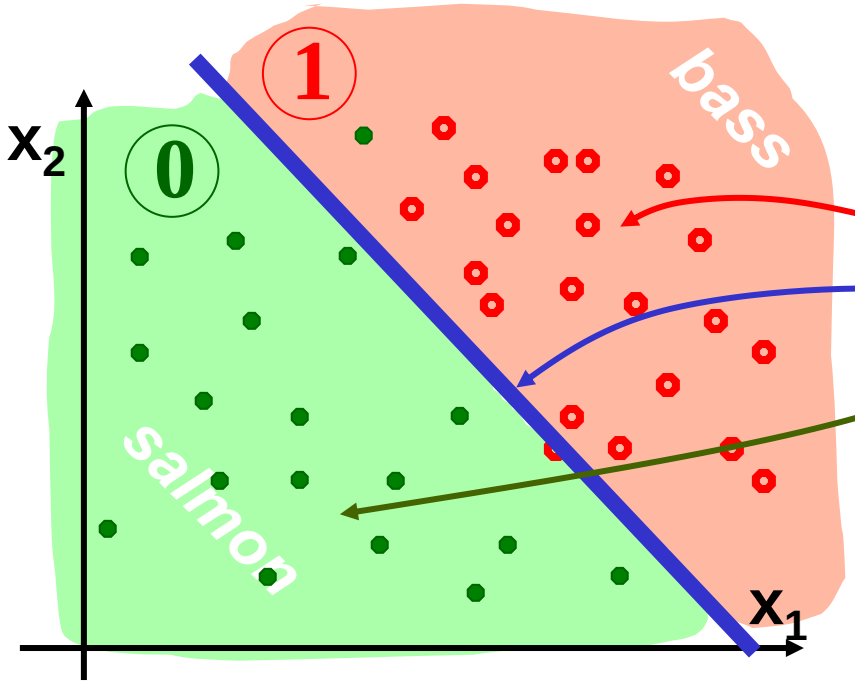


unit step function
(a.k.a. *Heaviside* function)

$$u(t) := \begin{cases} 1 & \text{if } t > 0 \\ 0 & O.W. \end{cases}$$

Linear classifier example: *perceptron*

For two class problem and 2-dimensional data (feature vectors)



$$\mathbf{f}(\mathbf{w},\mathbf{x}) = u\,(\mathbf{w}_0+\mathbf{w}_1\mathbf{x}_1+\mathbf{w}_2\mathbf{x}_2)$$

decision regions

decision boundary

Can use this function to classify any (new) point.

- Can be generalized to feature vectors $\mathbf{x}$ of any dimension m :

$\mathbf{f}(W,X) = \mathbf{u}\,(W^T X)$ for $W^T = [\mathbf{w}_0, \mathbf{w}_1, ..., \mathbf{w}_m]$ and $X^T = [1, \mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_m]$

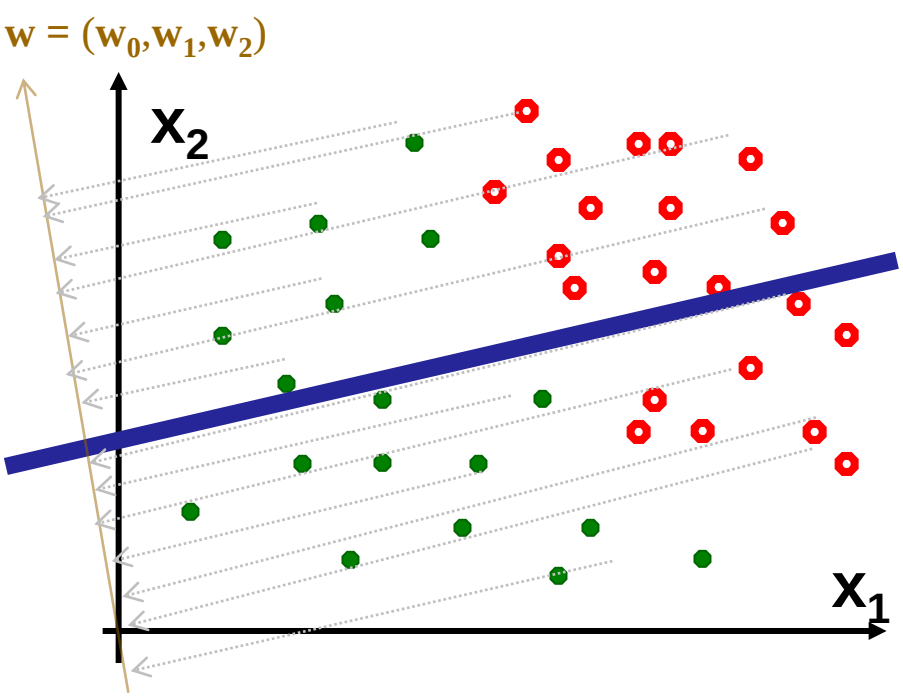
"bias"

homogeneous **representation of feature vector x**

- Classifier that makes decisions based on linear combination of features is called a **linear classifier**

Linear Classifiers

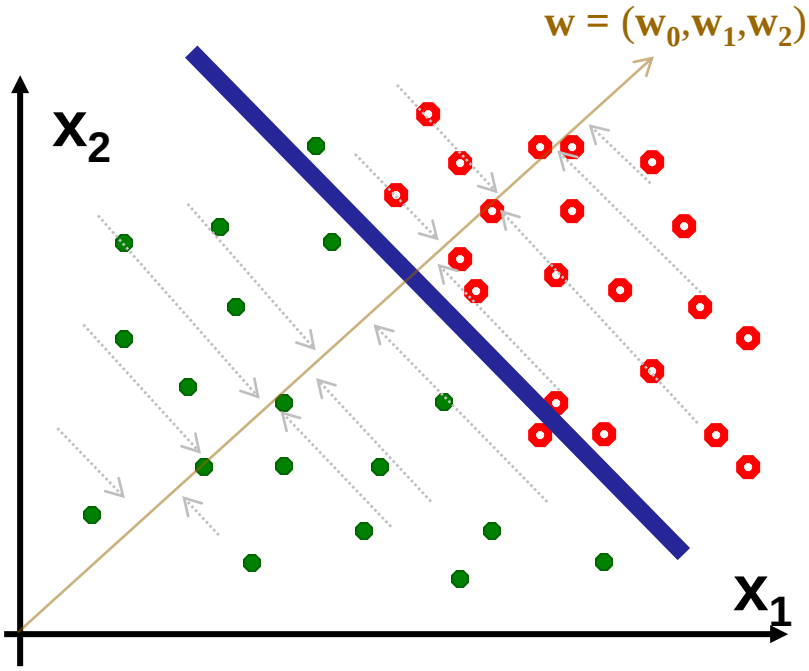
bad **w**



classification error 38%

projected points onto
normal line are all mixed-up

better **w**

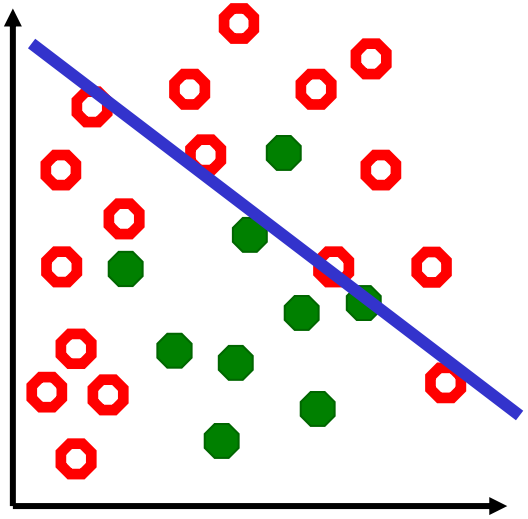


classification error 4%

projected points onto
normal line are well separated

Underfitting

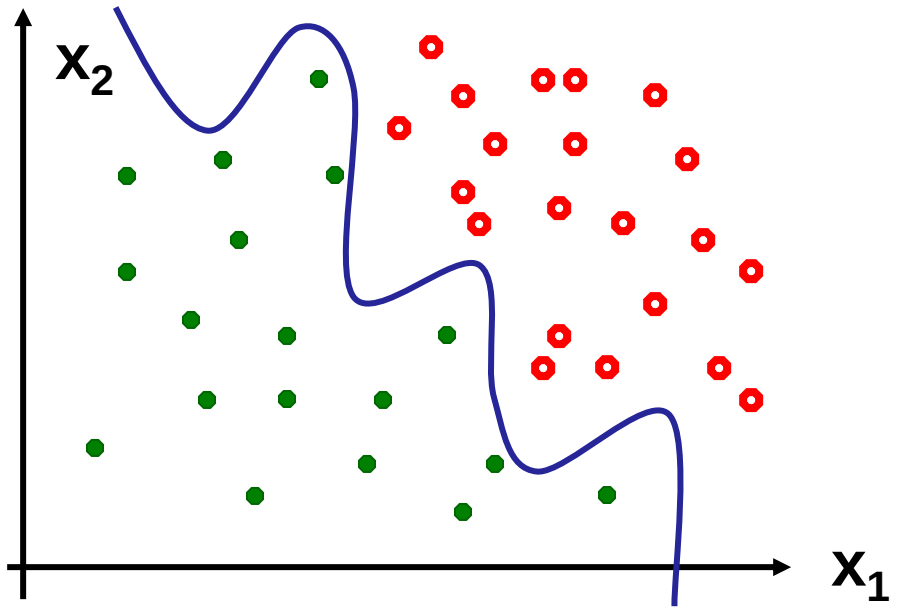
For some types of data
no linear decision boundary
can separate the samples well



- Classifier underfits the data if it can produce decision boundaries that are too simple for this type of data

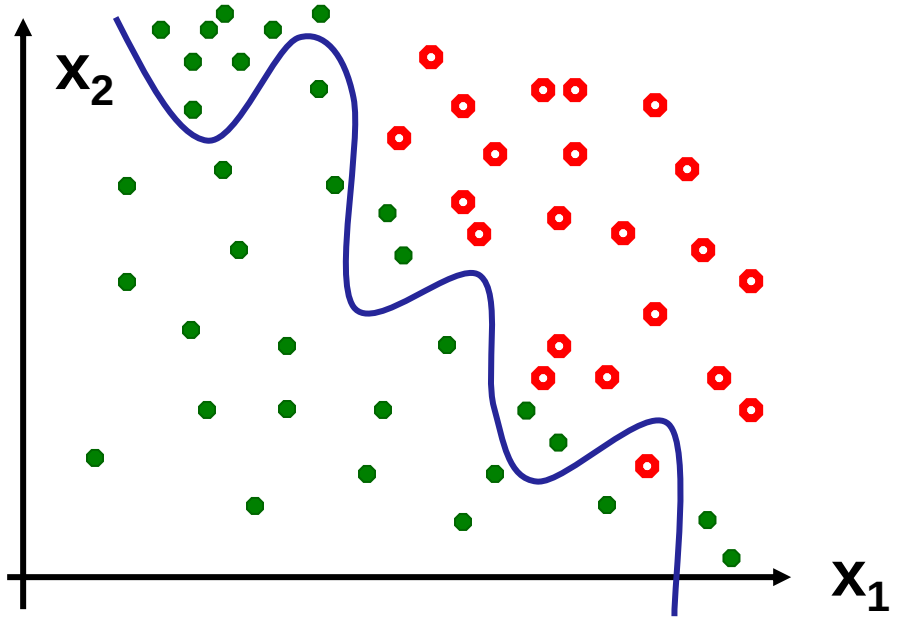
  - chosen classifier type (hypothesis space) is not expressive enough

More Complex (non-linear) Classifiers



for example, if **f**(**w**,**x**) is a polynomial of high degree can achieve 0% classification error

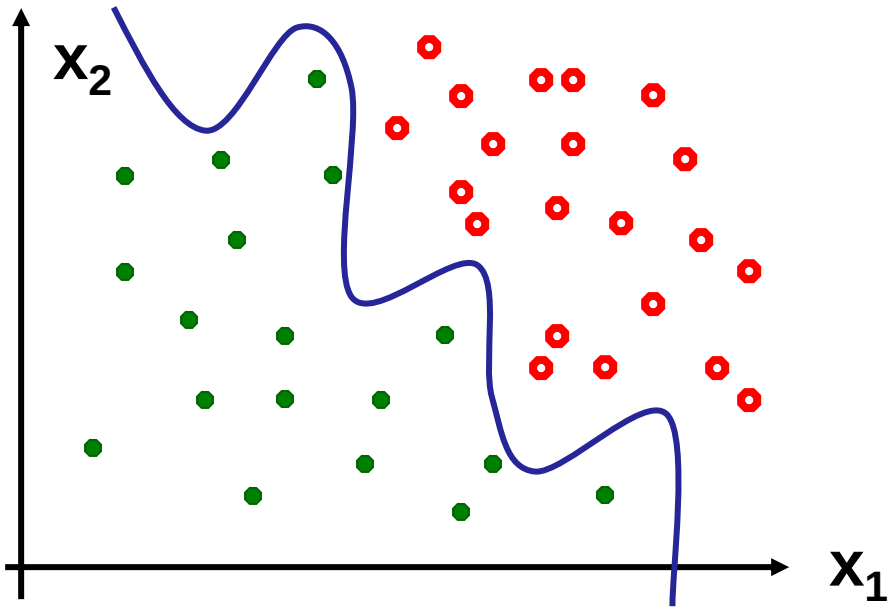
More Complex (non-linear) Classifiers



The goal is to classify well on **new data**

Test “wiggly” classifier on new data: 25% error

Overfitting



- Amount of data for training is always limited
- Complex model often has too many parameters to fit reliably to limited data
- Complex model may adapt too closely to “random noise” in training data, rather than look at a “big picture”

Overfitting: Extreme Example

- Two class problem: *face* and *non-face* images
- Memorize (i.e. store) all the “face” images
- For a new image, see if it is one of the stored faces
  - if yes, output “face” as the classification result
  - If no, output “non-face”

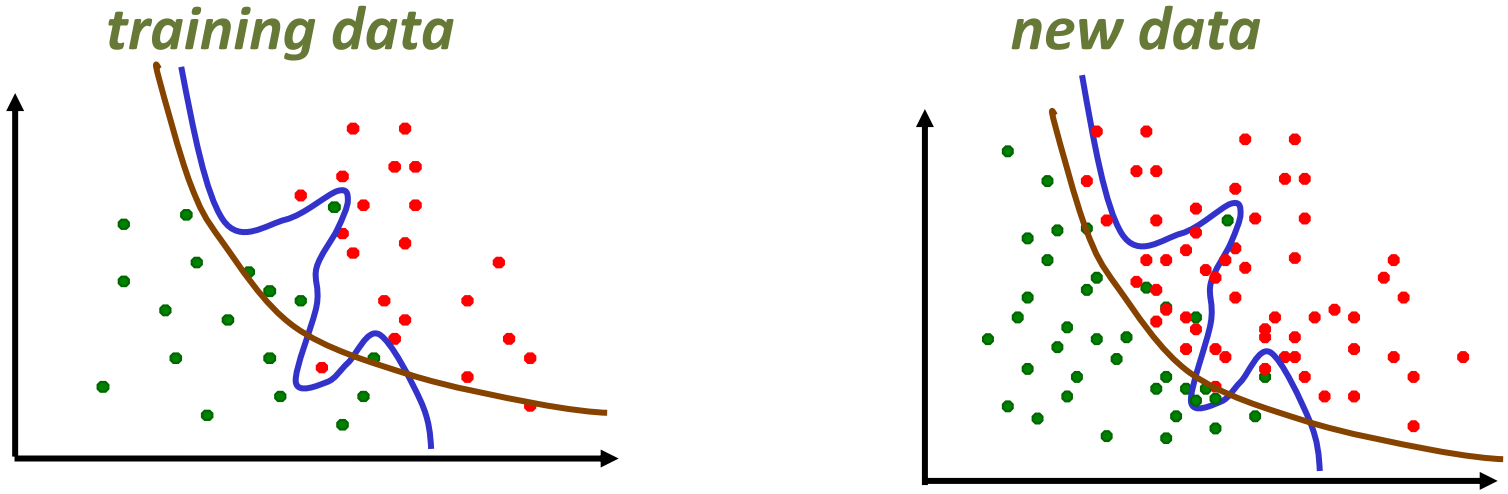
problem:

- zero error on stored data, 50% error on test (new) data
- decision boundary is very irregular

Such learning is **memorization without generalization**

slide is modified from Y. LeCun

Generalization



- Ability to produce correct outputs on previously unseen examples is called **generalization**
- Big question of learning theory: how to get good generalization with a limited number of examples
- Intuitive idea: **favor simpler classifiers**
  - William of Occam (1284-1347): “entities are not to be multiplied without necessity”
- Simpler decision boundary may not fit ideally to training data but tends to generalize better to new data

Training and Testing

How to diagnose overfitting?

Divide all labeled samples $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^n$ into ***training* set** and ***test* set**

- Use training set (training samples) to tune weights **w**
- Use test set (test samples) to check how well classifier with tuned weights **w** work on unseen examples

Thus, two main phases in classifier design are:

1. **training**
2. **testing**

Training Phase

Find best weights $\mathbf{w}^*$ such that $f(\mathbf{w},\mathbf{x}^i) = \mathbf{y}^i$
"as much as possible" for *training* samples $\mathbf{x}^i$

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \sum_{i \in \text{train}} L(\mathbf{y}^i, \mathbf{f}(\mathbf{w}, \mathbf{x}^i))$$

optimization problem

loss function $L(\mathbf{y}, \mathbf{f})$ penalizes whenever $\mathbf{y}^i \neq \mathbf{f}(\mathbf{w}, \mathbf{x}^i)$

Iverson brackets

- e.g. if $L(\mathbf{y}, \mathbf{f}) = [\mathbf{y} \neq \mathbf{f}]$ then the loss counts *classification errors*

- classification error on training data is called ***training error***

Testing Phase

- The goal is good performance on unseen examples

- Evaluate performance of the trained classifier **f**(**w**,**x**) on the test samples (unseen labeled samples)

- Testing on unseen labeled examples is an approximation of how well classifier will perform in practice

- If testing results are poor, may have to go back to the training phase and redesign **f**(**w**,**x**)

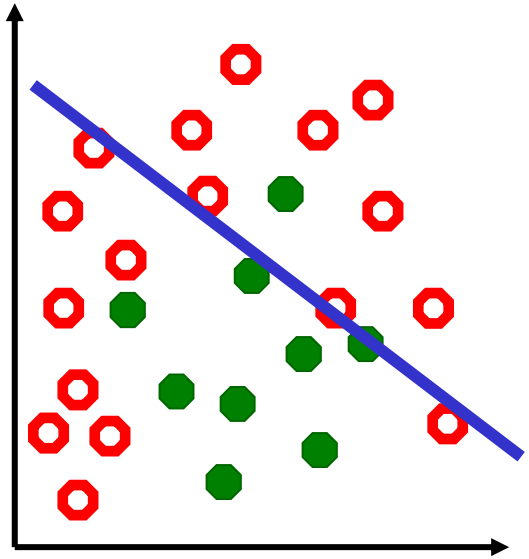
- Classification error on test data is called ***test error***

- Side note

  "deploying" the final classifier **f**(**w**,**x**) in practice is also called testing

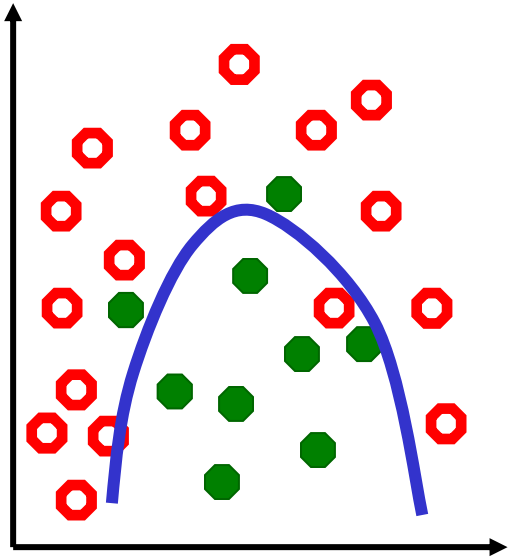
Underfitting → Overfitting

underfitting



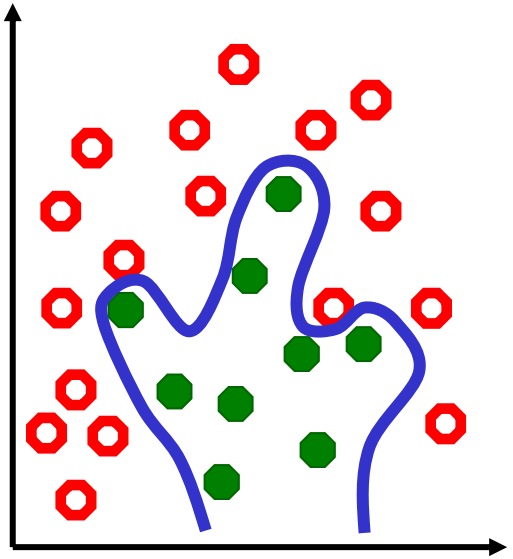
- high training error
- high test error

"just right"



- low training error
- low test error

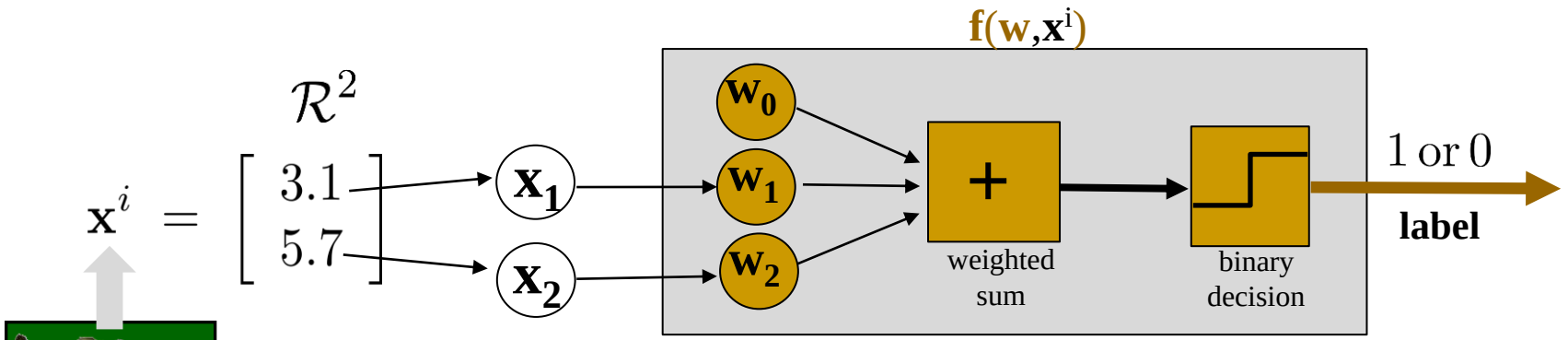
overfitting



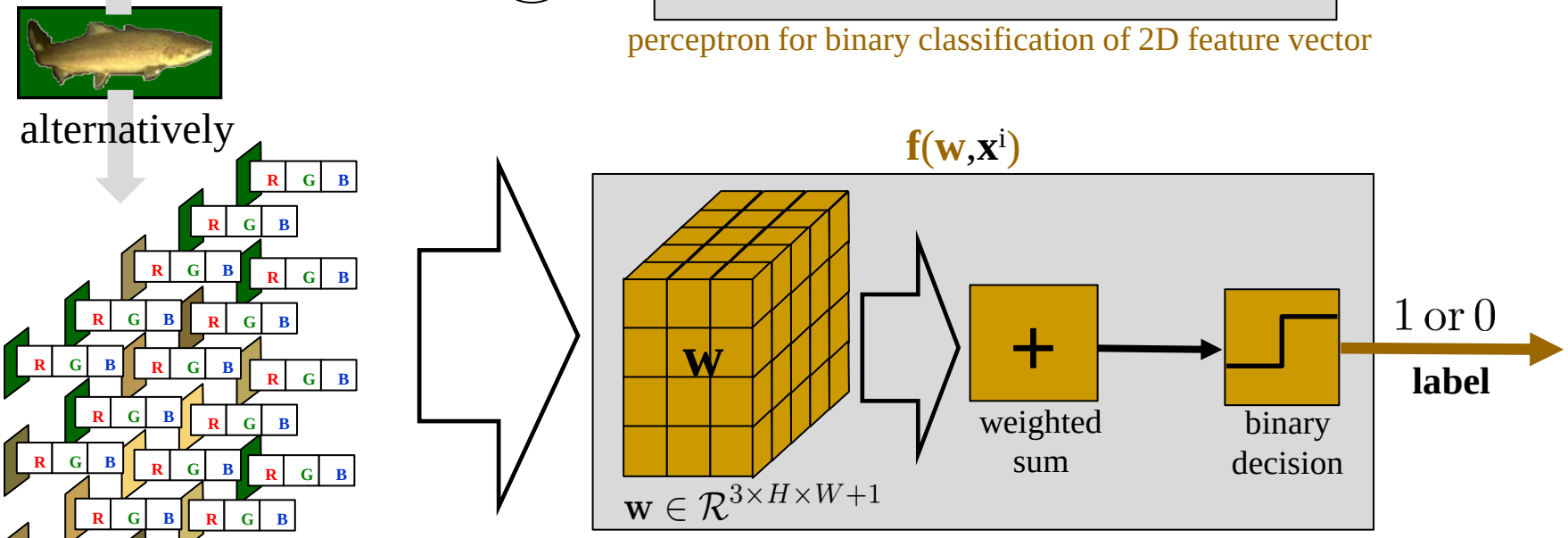
- low training error
- high test error

One can have **more-complex** or **less-complex** linear classification methods

Examples:



perceptron for binary classification of 2D feature vector



perceptron for binary classification of
feature vector representing whole RGB image

can be interpreted as "**fully connected" one-layer binary NN**

$\mathbf{x}^i \in \mathcal{R}^{3\times H\times W}$

NOTE: both **perceptrons are** still **linear classifiers**

Training requires optimization of **Loss Function**

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \underbrace{\sum_{i \in \text{train}} L(\mathbf{y}^i, \mathbf{f}(\mathbf{w}, \mathbf{x}^i))}_{L(\mathbf{w})}$$

single example loss

prediction on example $\mathbf{x}^i$

$L(\mathbf{w})$

total loss

NOTE: our loses are **multi-variate** functions

$\mathbf{w} = (\mathbf{w}^0, \mathbf{w}^1, \mathbf{w}^2, \ldots, \mathbf{w}^n)$

quick overview:

optimization of multi-variate functions
via
Gradient Descent

Optimization of continuous differentiable functions

How to minimize a function of a single variable

$$f(x) = (x-5)^2$$

- From calculus: take derivative and set it to 0

$$\frac{d}{dx} f(x) = 0$$

- May find a closed form solution, as in the simple example above

$$\frac{d}{dx} f(x) \;=\; 2(x-5) \;=\; 0 \qquad \Rightarrow \qquad x = 5$$

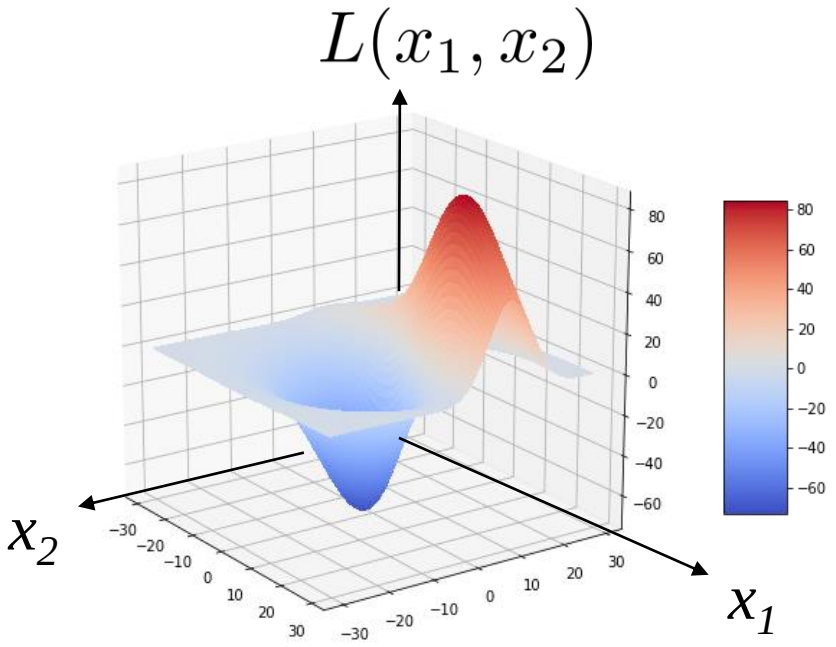
In practice, more often cannot find a closed form solution and need to solve numerically.

Particularly true for complex multi-variate functions.

Differentiation

Remember some slides from topic 3

3D plot

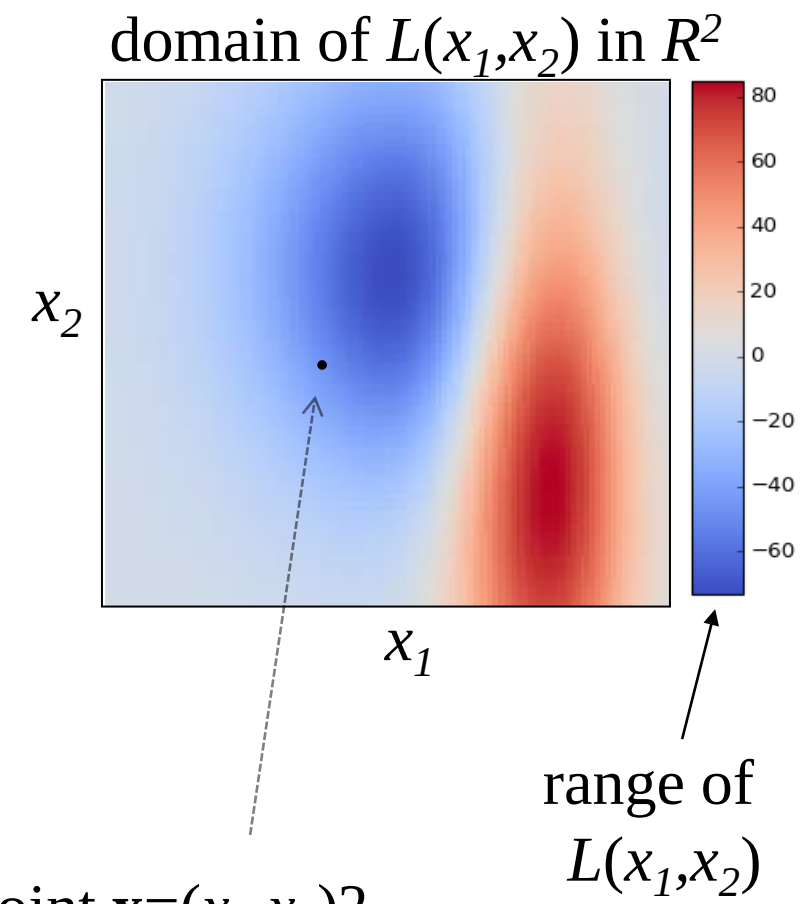


What is "**slope**" of $L(x_1,x_2)$ at a given point $\mathbf{x}=(x_1,x_2)$?

Differentiation

Remember some slides from topic 3

"heat-map" visualization of L



What is "**slope**" of $L(x_1,x_2)$ at a given point $\mathbf{x}=(x_1,x_2)$?

Differentiation

Remember some slides from topic 3

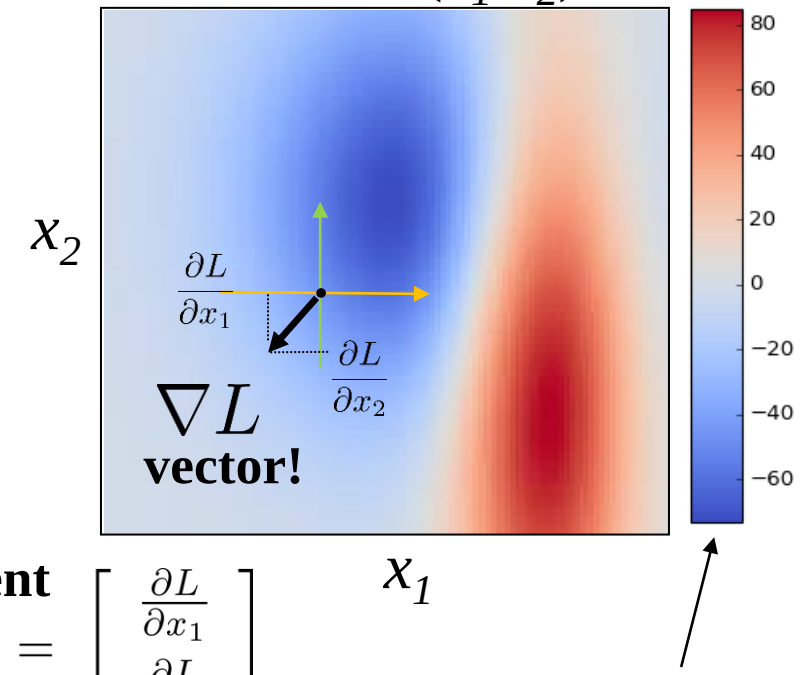
"heat-map" visualization of L

"partial" derivatives

$$\frac{\partial L}{\partial x_1} = \lim_{\epsilon \to 0} \left(\frac{L(x_1 + \epsilon, x_2) - L(x_1, x_2)}{\epsilon} \right)$$

$$\frac{\partial L}{\partial x_2} = \lim_{\epsilon \to 0} \left(\frac{L(x_1, x_2 + \epsilon) - L(x_1, x_2)}{\epsilon} \right)$$

domain of $L(x_1,x_2)$ in R^2



gradient

$$\nabla L = \begin{bmatrix} \frac{\partial L}{\partial x_1} \\ \frac{\partial L}{\partial x_2} \end{bmatrix}$$

range of $L(x_1,x_2)$

direction of the steepest ascent at point x=(x_1,x_2)

Differentiation

The most common optimization method for continuous differentiable (multi-variate) functions:

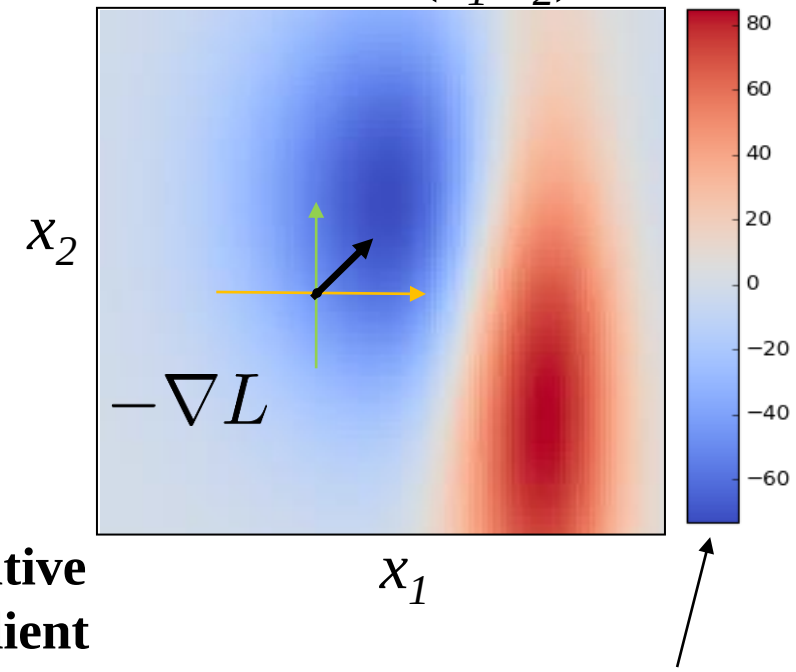
gradient descent

take a step $\mathbf{x}' = \mathbf{x} - \alpha \nabla L$

towards lower values of the function

"heat-map" visualization of L

domain of $L(x_1,x_2)$ in R^2



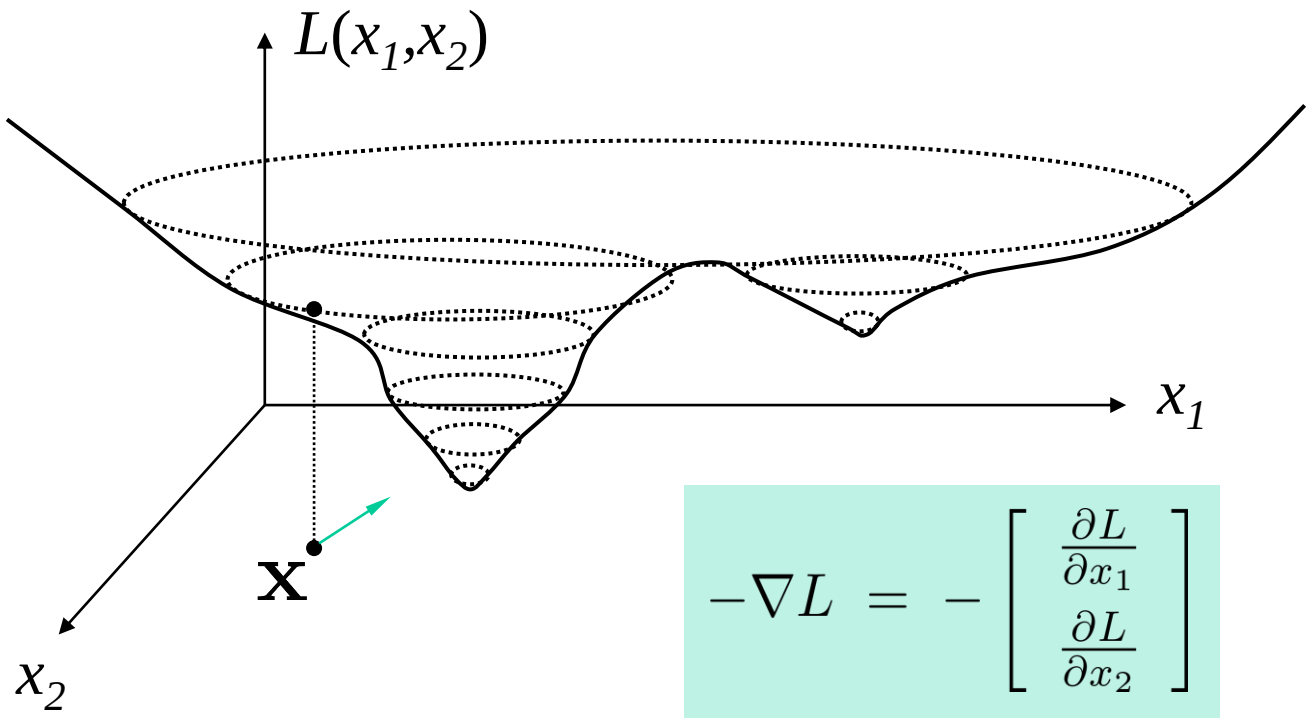
negative gradient

range of $L(x_1,x_2)$

direction of the steepest descent at point $\mathbf{x}=(x_1,x_2)$

Gradient Descent

Example: for a function of two variables

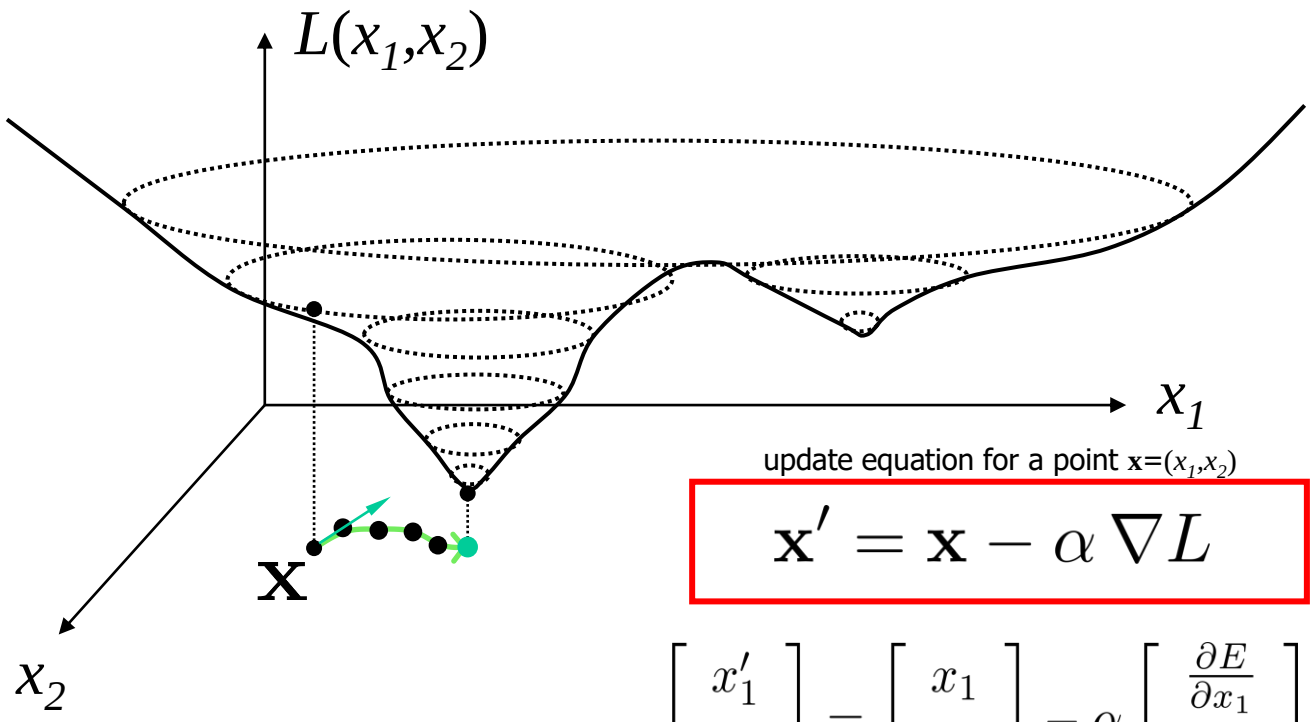


$$-\nabla L = -\begin{bmatrix} \frac{\partial L}{\partial x_1} \\ \frac{\partial L}{\partial x_2} \end{bmatrix}$$

- direction of (negative) gradient at point $\mathbf{x}=(x_1,x_2)$ is direction of the steepest descent towards lower values of function L
- magnitude of gradient at $\mathbf{x}=(x_1,x_2)$ gives the value of the slope

Gradient Descent

Example: for a function of two variables



update equation for a point $\mathbf{x}=(x_1,x_2)$

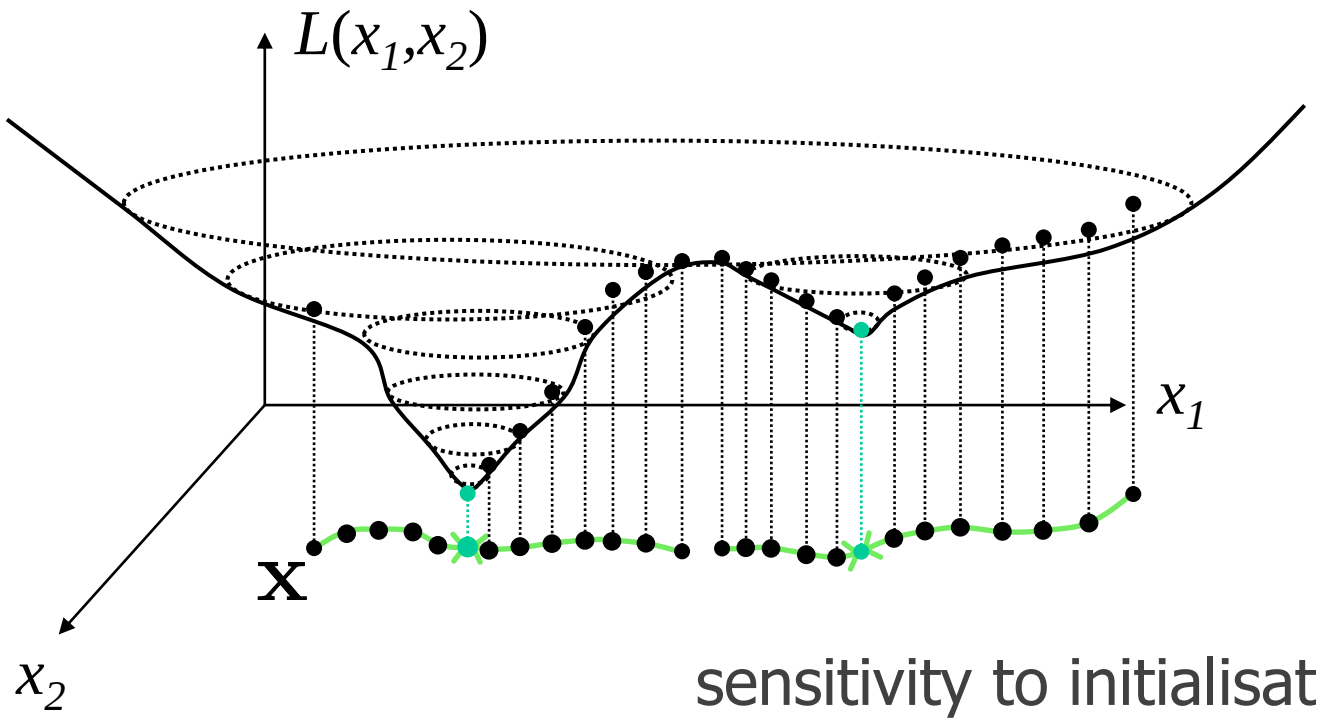
$$\mathbf{x}' = \mathbf{x} - \alpha \nabla L$$

$$\begin{bmatrix} x_1' \\ x_2' \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} - \alpha \begin{bmatrix} \frac{\partial E}{\partial x_1} \\ \frac{\partial E}{\partial x_2} \end{bmatrix}$$

Stop at a local minima where $\nabla L = \vec{0}$

Gradient Descent

Example: for a function of two variables

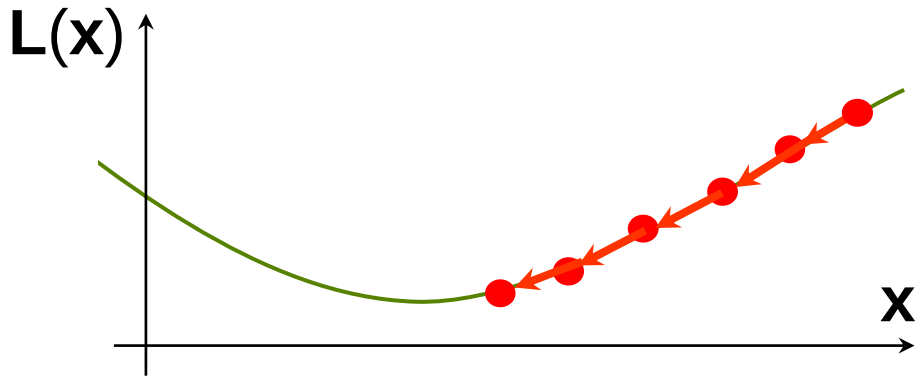


sensitivity to initialisation !!

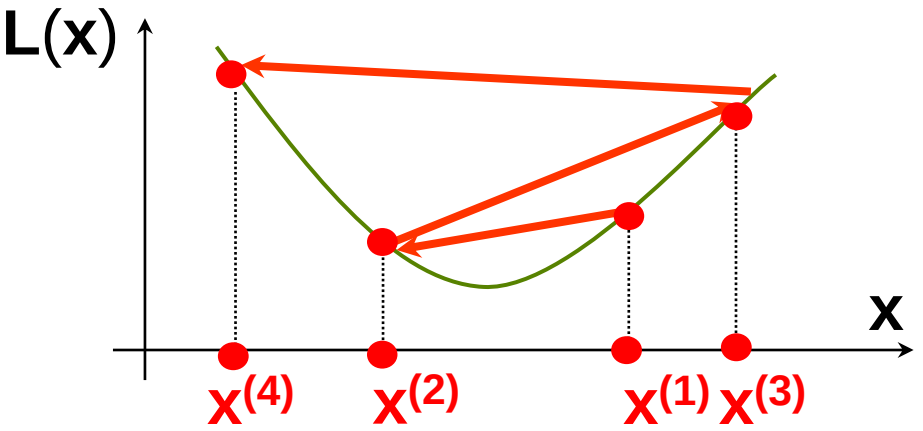
How to Set Learning Rate α?

$$\mathbf{x}' = \mathbf{x} - \alpha \nabla L$$

- If α too small, too many iterations to converge



- If α too large, may overshoot the local minimum and possibly never even converge



Variable Learning Rate

If desired, can change learning rate α at each iteration

$\mathbf{k} = 1$

$\mathbf{x}^{(1)}$ = any initial guess

choose α, ε

while $\alpha \|\nabla L(\mathbf{x}^{(k)})\| > \varepsilon$

$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha \nabla L(\mathbf{x}^{(k)})$

$\mathbf{k} = \mathbf{k} + 1$

fixed α
gradient descent

$\mathbf{k} = 1$

$\mathbf{x}^{(1)}$ = any initial guess

choose ε

while $\alpha \|\nabla L(\mathbf{x}^{(k)})\| > \varepsilon$

choose $\alpha^{(k)}$

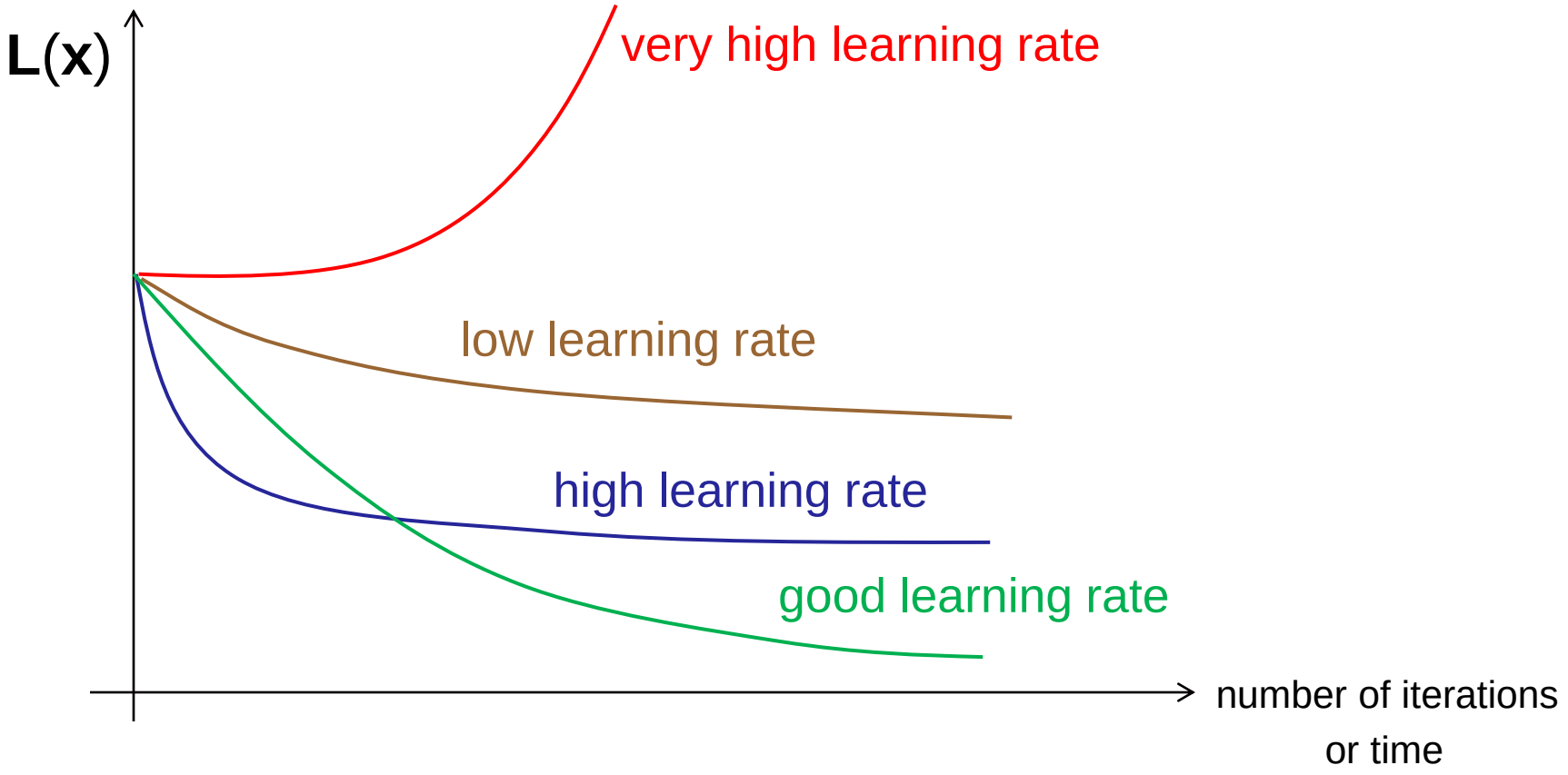
$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha^{(k)} \nabla L(\mathbf{x}^{(k)})$

$\mathbf{k} = \mathbf{k} + 1$

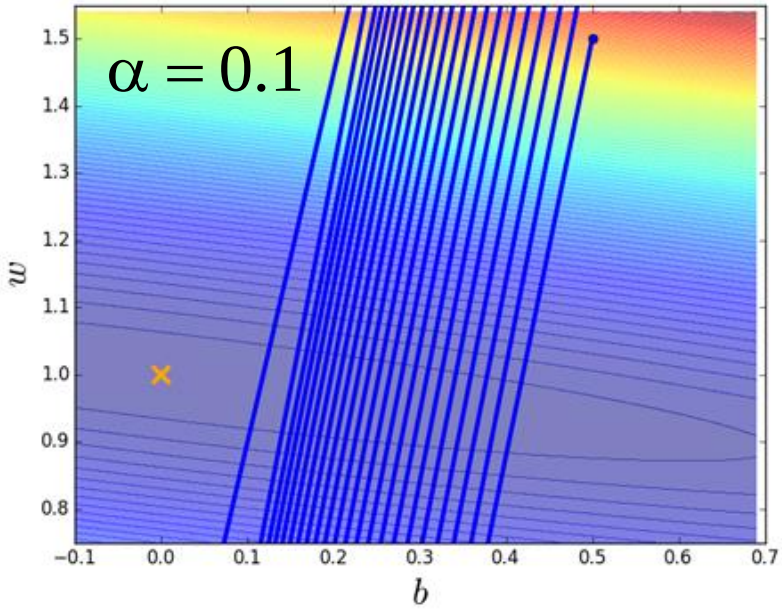
variable α
gradient descent

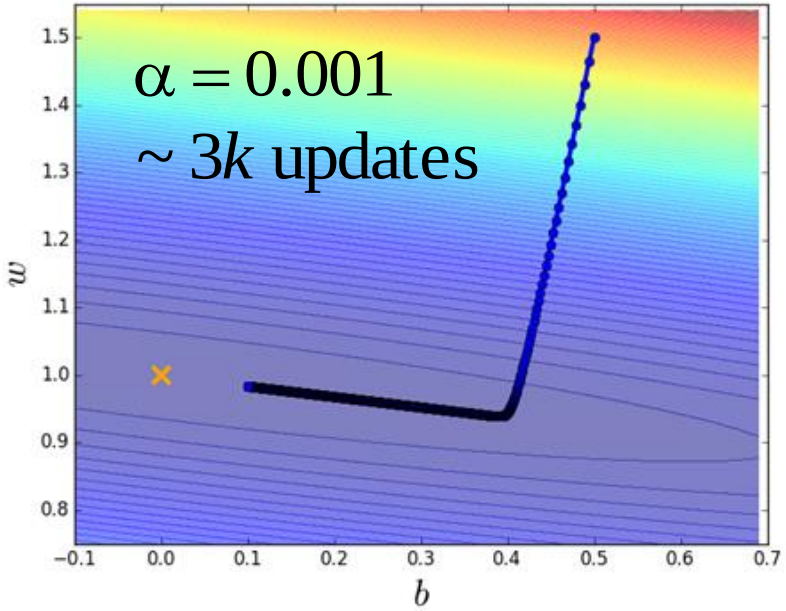
Learning Rate

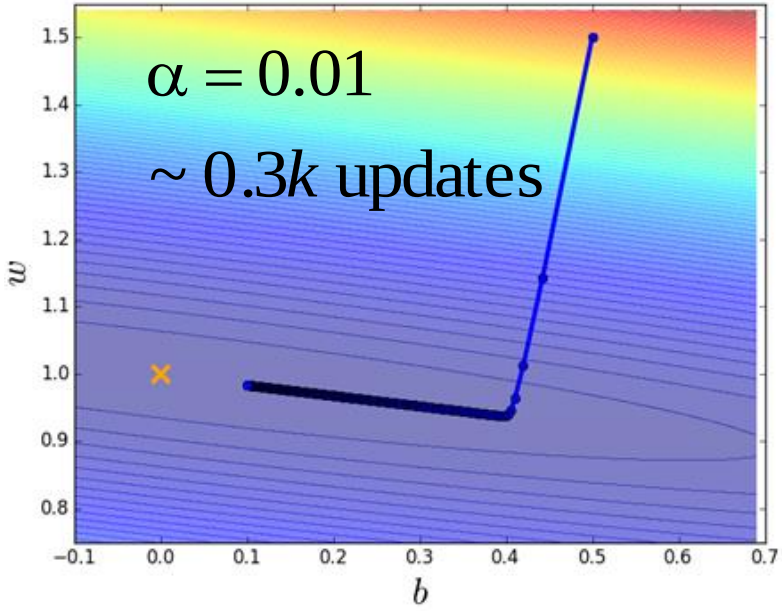
- Monitor learning rate by looking at how fast the objective function decreases



Learning Rate: Loss Surface Illustration







Back to

Loss Functions
and
Loss Optimization

Training Perceptron - First Attempt

single example loss

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \underbrace{\sum_{i \in \text{train}} L(\mathbf{y}^i, \mathbf{f}(\mathbf{w}, \mathbf{x}^i))}_{L(\mathbf{w})}$$

prediction on example $\mathbf{x}^i$

total loss

Consider **perceptron**: $\mathbf{f}(\mathbf{w}, \mathbf{x}) = u(W^T X)$

vector representation of $\mathbf{w}$

$$W^T = [\mathbf{w}_0, \mathbf{w}_1, ..., \mathbf{w}_m]$$

$$X^T = [1, \mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_m]$$

homogeneous representation of $\mathbf{x}$

Classification error loss: $L(\mathbf{y}, \mathbf{f}) = [\mathbf{y} \neq \mathbf{f}]$

Iverson brackets

perceptron's prediction on example $\mathbf{x}^i$

$$\Rightarrow \quad L(W) = \underbrace{\sum_{i \in \text{train}} [\mathbf{y}^i \neq u(W^T X^i)]}$$

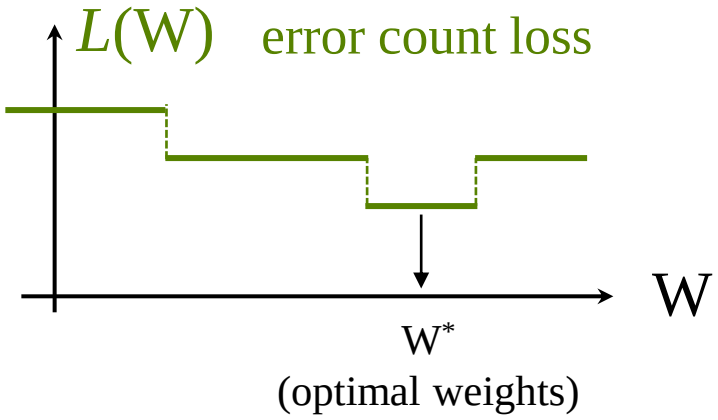
classification error <u>counts</u>

since both $\mathbf{y}^i$, $u \in \{0,1\}$

UNIVERSITY OF WATERLOO

Zero Gradients Problem

Classification error loss function $L(\mathrm{W})$ is **piecewise constant**:



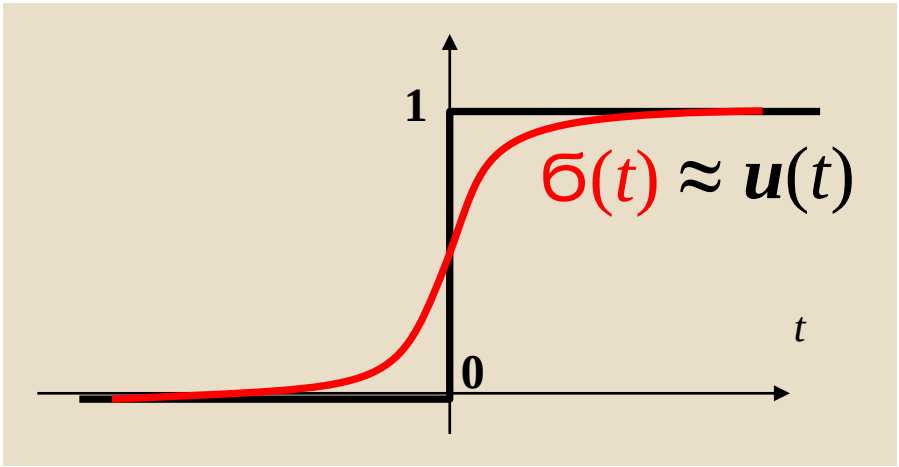
NOTE: in this case gradient ∇L is always either zero or does not exist

"error count" loss function cannot be optimized via *gradient descent*

Work-around for Zero Gradients

Perceptron: $\mathbf{f}(\mathbf{w}, \mathbf{x}^i) = u(W^T X^i) \approx \sigma(W^T X^i)$

approximate decision function *u* using its **softer** version (relaxation)



$u(t)$ - ***unit step*** function
(a.k.a. *Heaviside* function)

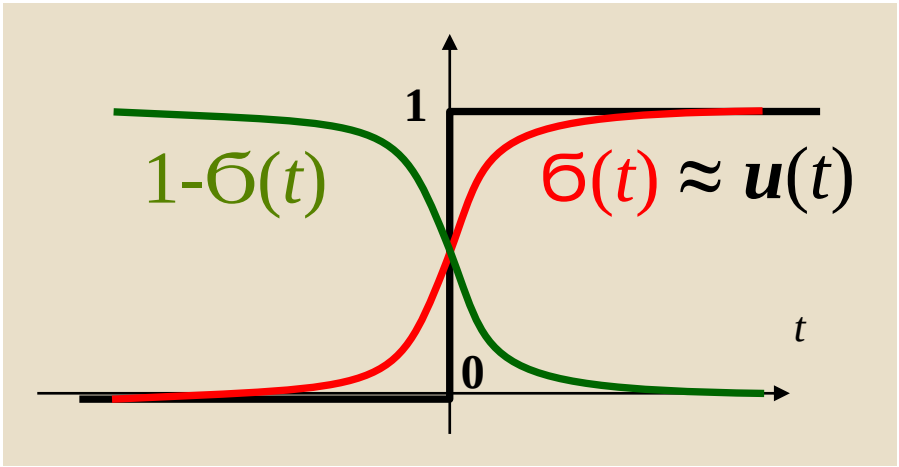
$\sigma(t)$ - ***sigmoid*** function

$$\sigma(t) := \frac{1}{1 + \exp(-t)}$$

Work-around for Zero Gradients

Perceptron: $\mathbf{f}(\mathbf{w}, \mathbf{x}^i) = u(W^T X^i) \approx \sigma(W^T X^i)$

approximate decision function *u* using its **softer** version (relaxation)



$u(t)$ - ***unit step*** function
(a.k.a. *Heaviside* function)

$\sigma(t)$ - ***sigmoid*** function

$$\sigma(t) := \frac{1}{1 + \exp(-t)}$$

Relaxed predictions are often interpreted as prediction **"probabilities"**

$$\Pr(\mathbf{x}^i \in \text{Class1} \mid W) = \sigma(W^T X^i)$$

$$\Pr(\mathbf{x}^i \in \text{Class0} \mid W) = 1 - \sigma(W^T X^i) \equiv \sigma(-W^T X^i)$$

Training Perceptron - Second Attempt

Perceptron approximation: $\mathbf{f}(\mathbf{w}, \mathbf{x}^i) = u(W^T X^i) \approx \sigma(W^T X^i)$

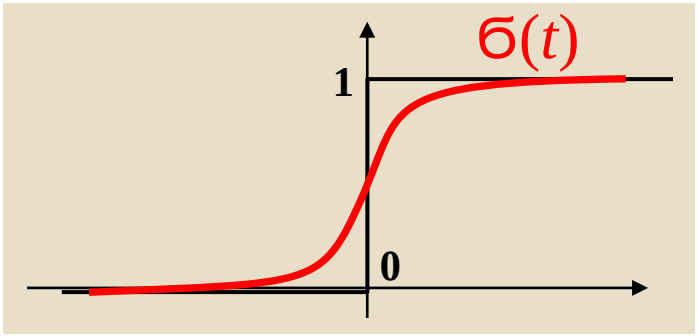
Classification error loss: $L(\mathbf{y}, \sigma) = [\mathbf{y} \neq \sigma]$ where $\mathbf{y} \in \{0, 1\}$

now makes no sense at all

relaxed decision function (sigmoid)

never returns exactly 0 or 1

NOTE:
To be able to use **gradient descent** we need to "**soften**" both the **decision function** and the **loss function**



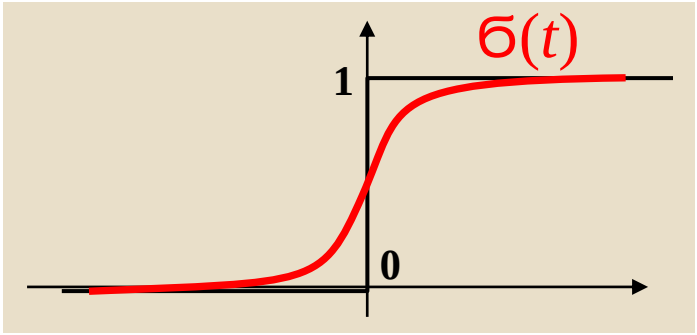
$$0 < \quad \sigma(t) \equiv \frac{1}{1 + \exp(-t)} \quad < 1$$

Quadratic Loss

Perceptron approximation: $\mathbf{f}(\mathbf{w}, \mathbf{x}^i) = u(W^T X^i) \approx \sigma(W^T X^i)$

Consider **quadratic loss**: $L(\mathbf{y}, \sigma) = (\mathbf{y} - \sigma)^2$, $\mathbf{y} \in \{0, 1\}$

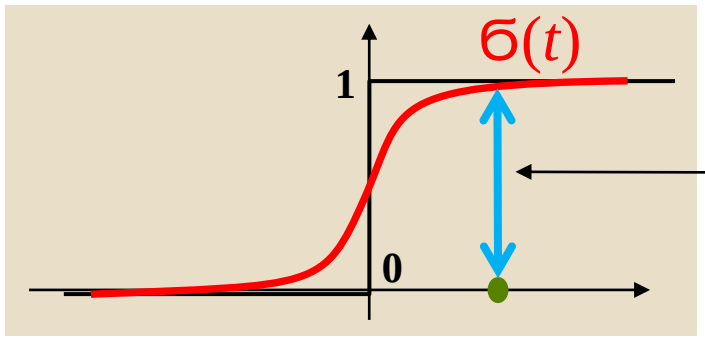
NOTE:
Loss $L(\mathbf{y}, \sigma(W^T X))$ is now differentiable with respect to W because $L(\mathbf{y}, \sigma)$ is differentiable w.r.t. σ



Quadratic Loss

Perceptron approximation: $\mathbf{f}(\mathbf{w}, \mathbf{x}^i) = u(W^T X^i) \approx \sigma(W^T X^i)$

Consider **quadratic loss**: $L(\mathbf{y}, \sigma) = (\mathbf{y} - \sigma)^2$, $\mathbf{y} \in \{0, 1\}$



$(\mathbf{y}^i - \sigma(W^T X^i))$

$W^T X^i > 0$

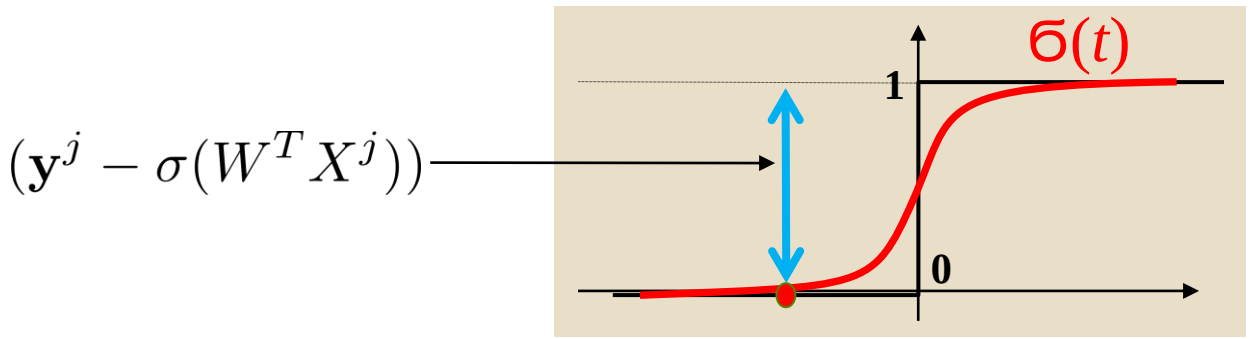
$\mathbf{y}^i = 0$

misclassified example

Quadratic Loss

Perceptron approximation: $\mathbf{f}(\mathbf{w}, \mathbf{x}^i) = u(W^T X^i) \approx \sigma(W^T X^i)$

Consider **quadratic loss**: $L(\mathbf{y}, \sigma) = (\mathbf{y} - \sigma)^2$, $\mathbf{y} \in \{0, 1\}$



$W^T X^j < 0$

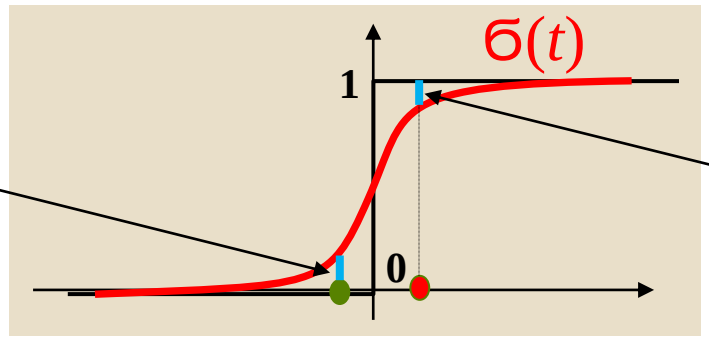
$\mathbf{y}^j = 1$

another misclassified example

Quadratic Loss

Perceptron approximation: $\mathbf{f}(\mathbf{w}, \mathbf{x}^i) = u(W^T X^i) \approx \sigma(W^T X^i)$

Consider **quadratic loss**: $L(\mathbf{y}, \sigma) = (\mathbf{y} - \sigma)^2$, $\mathbf{y} \in \{0, 1\}$



$(\mathbf{y}^j - \sigma(W^T X^j))$

$(\mathbf{y}^i - \sigma(W^T X^i))$

$W^T X^j$ $W^T X^i$

$\mathbf{y}^j = 0$ $\mathbf{y}^i = 1$

correctly classified examples

NOTE:

loss function encourages W s.t. correctly classified points are moved further from the decision boundary, i.e. $W^T X^i \gg 0$ and $W^T X^j \ll 0$.

Quadratic Loss

Perceptron approximation: $\mathbf{f}(\mathbf{w}, \mathbf{x}^i) = u(W^T X^i) \approx \sigma(W^T X^i)$

Consider **quadratic loss**: $L(\mathbf{y}, \sigma) = (\mathbf{y} - \sigma)^2$

Total loss $\Rightarrow$ $L(W) = \sum_{i \in \text{train}} (\mathbf{y}^i - \sigma(W^T X^i))^2$

approximation for perceptron's prediction on example $\mathbf{x}^i$

Sum of Squared Differences (SSD)

(binary case)

Cross-Entropy Loss (related to *logistic regression* loss)

Perceptron approximation: $\mathbf{f}(\mathbf{w}, \mathbf{x}^i) = u(W^T X^i) \approx \sigma(W^T X^i)$

Consider two probability distributions over two classes (e.g. bass or salmon) : $(\mathbf{y}, 1-\mathbf{y})$ and $(\sigma, 1-\sigma)$

$$\Pr(\mathbf{x}^i \in \text{Class1} \mid W) = \sigma(W^T X^i)$$

$$\Pr(\mathbf{x}^i \in \text{Class0} \mid W) = 1 - \sigma(W^T X^i)$$

Distance between two distributions can be evaluated via **cross-entropy**.

Remember from topic 9:

$$H(p^S | p^\theta) := -\sum_k p_k^S \ln p_k^\theta$$

(binary case)

Cross-Entropy Loss (related to *logistic regression* loss)

Perceptron approximation: $\mathbf{f}(\mathbf{w}, \mathbf{x}^i) = u(W^T X^i) \approx \sigma(W^T X^i)$

Consider two probability distributions
over two classes (e.g. bass or salmon) : $(\mathbf{y}, 1-\mathbf{y})$ and $(\sigma, 1-\sigma)$

(binary)
Cross-entropy loss:

$$L(\mathbf{y}, \sigma) = -\mathbf{y} \ln \sigma - (1-\mathbf{y}) \ln(1-\sigma)$$

Distance between two distributions can be evaluated via **cross-entropy**.
Remember from topic 9:

$$H(p^S | p^\theta) := -\sum_k p_k^S \ln p_k^\theta$$

(binary case)

Cross-Entropy Loss (related to *logistic regression* loss)

Perceptron approximation: $\mathbf{f}(\mathbf{w}, \mathbf{x}^i) = u(W^T X^i) \approx \sigma(W^T X^i)$

Consider two probability distributions over two classes (e.g. bass or salmon) : $(\mathbf{y}, 1-\mathbf{y})$ and $(\sigma, 1-\sigma)$

(binary)

Cross-entropy loss:

$$L(\mathbf{y}, \sigma) = -\mathbf{y} \ln \sigma - (1-\mathbf{y}) \ln(1-\sigma)$$

Each data label $\mathbf{y}$ provides "deterministic" distribution $(\mathbf{y}, 1-\mathbf{y})$ that is either (1,0) or (0,1). Thus, we get an equivalent alternative expression:

$$L(\mathbf{y}, \sigma) = \begin{cases} -\ln \sigma & \text{if } \mathbf{y} = 1 \\ -\ln(1-\sigma) & \text{if } \mathbf{y} = 0 \end{cases}$$

(binary case)

Cross-Entropy Loss (related to *logistic regression* loss)

Perceptron approximation: $\mathbf{f}(\mathbf{w}, \mathbf{x}^i) = u(W^T X^i) \approx \sigma(W^T X^i)$

Consider two probability distributions
over two classes (e.g. bass or salmon) : $(\mathbf{y}, 1-\mathbf{y})$ and $(\sigma, 1-\sigma)$

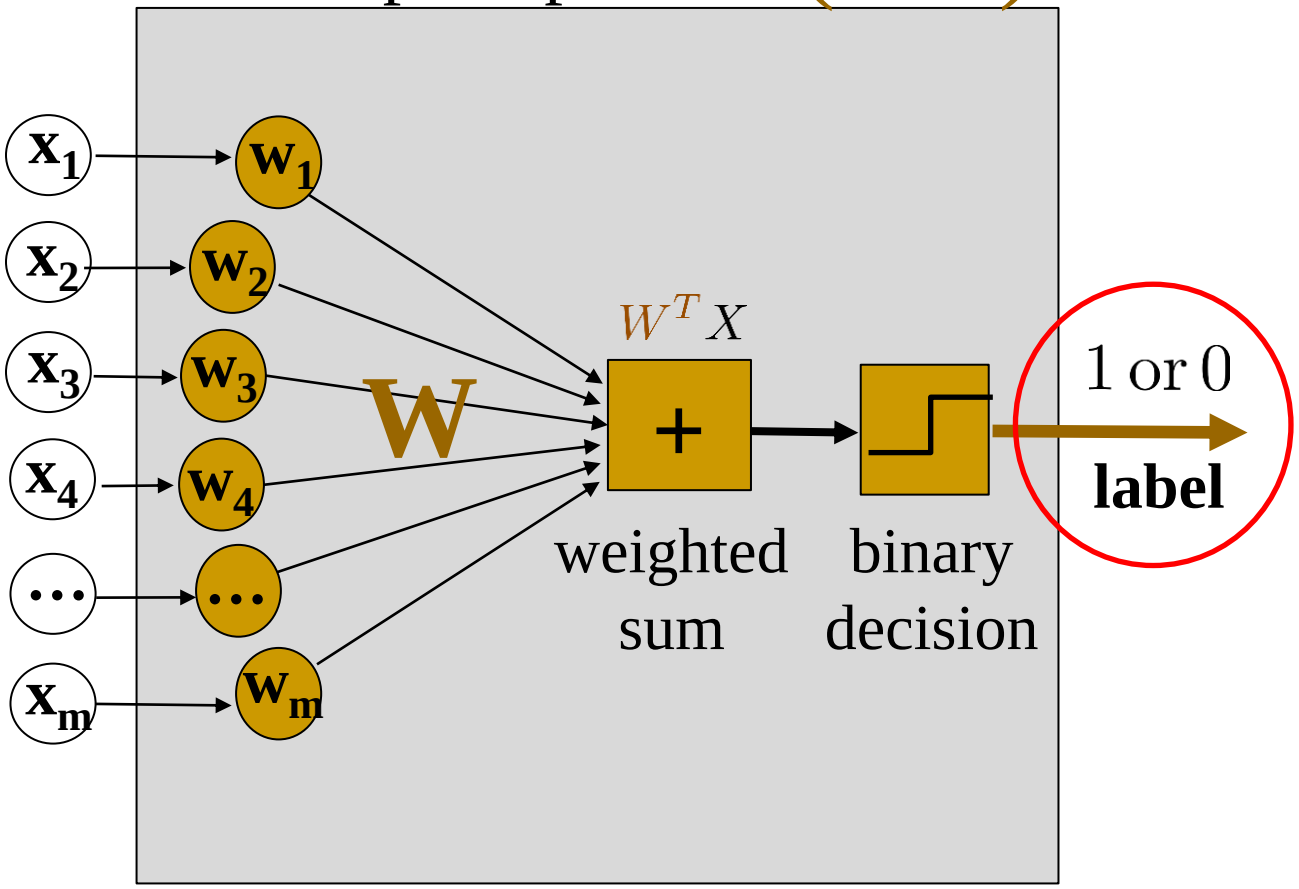
Total loss: $\sum_{i \in \text{train}} \left(-\mathbf{y}^i \ln \sigma(W^T X^i) - (1-\mathbf{y}^i) \ln(1-\sigma(W^T X^i))\right)$

$\Rightarrow$ $L(W) = -\sum_{\substack{i \in \text{train} \\ \mathbf{y}^i=1}} \ln \sigma(W^T X^i) - \sum_{\substack{i \in \text{train} \\ \mathbf{y}^i=0}} \ln(1-\sigma(W^T X^i))$

sum of **Negative Log-Likelihoods (NLL)**

Towards Multi-label Classification

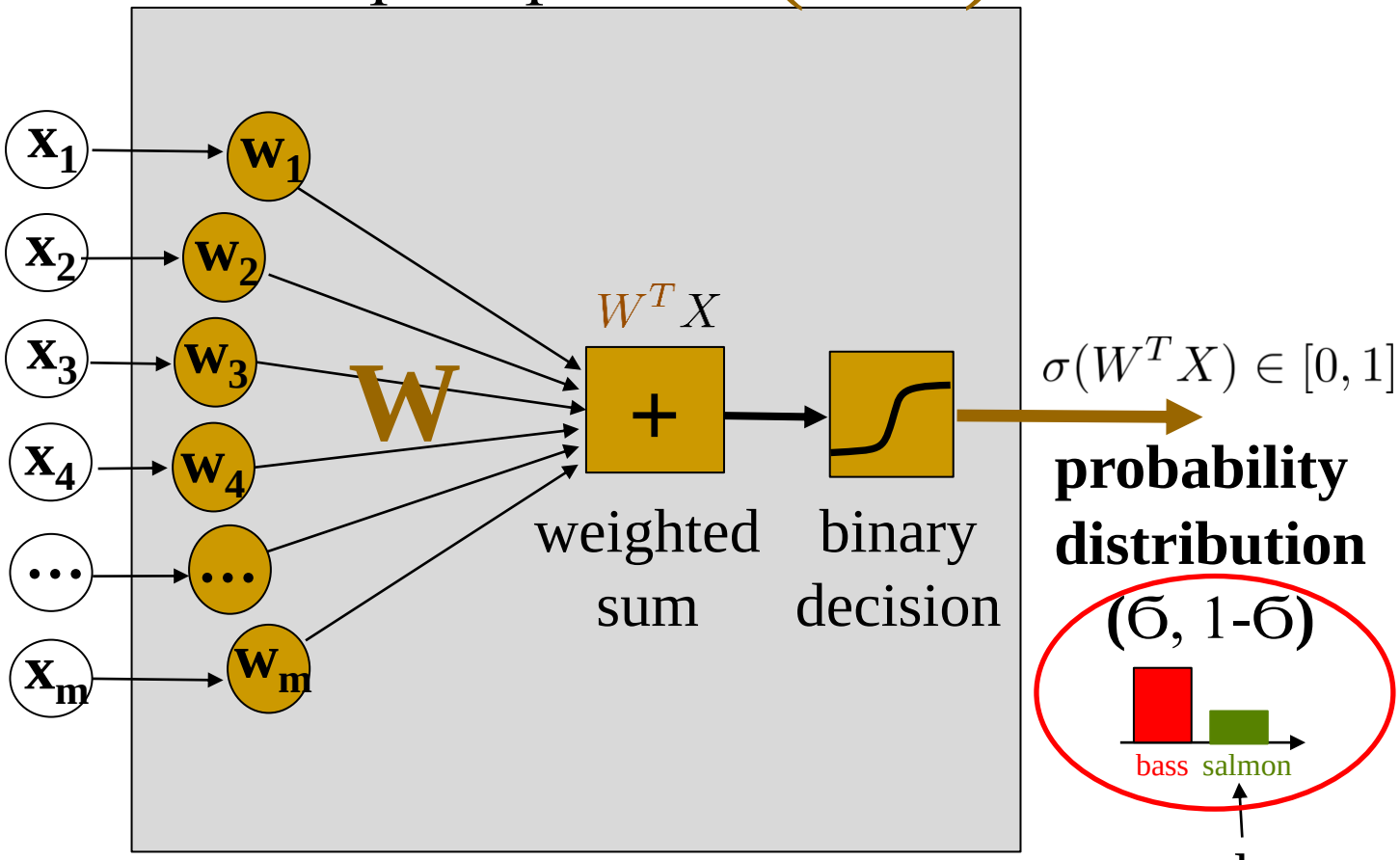
Remember: basic perceptron $\mathbf{u(w^T x)}$



binary classification

Towards Multi-label Classification

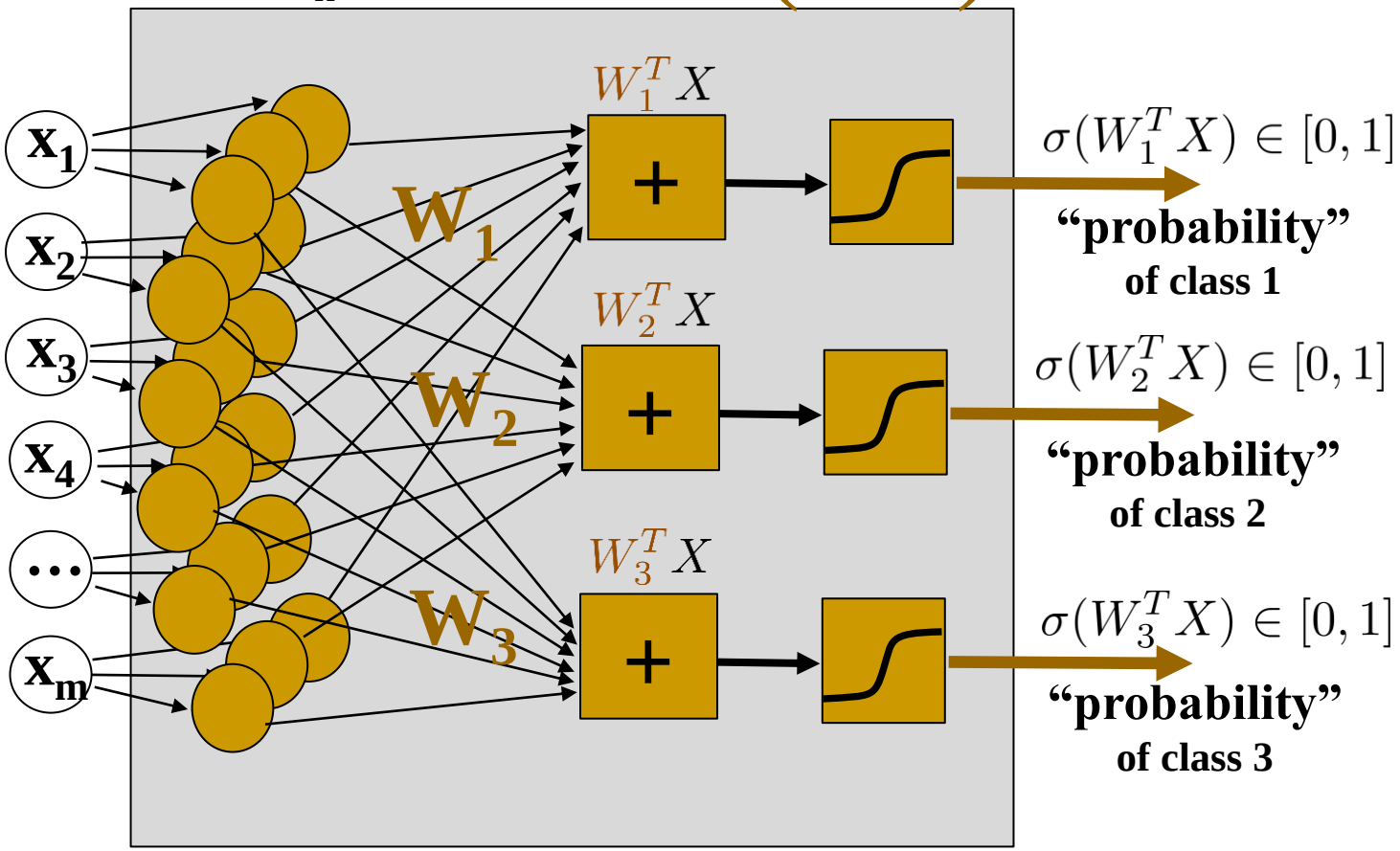
Remember: "relaxed" perceptron $\sigma(\mathbf{w}^T\mathbf{x})$



binary classification

Towards Multi-label Classification

use K linear transforms W_k and sigmoids $\sigma(\mathbf{w}^T\mathbf{x})$

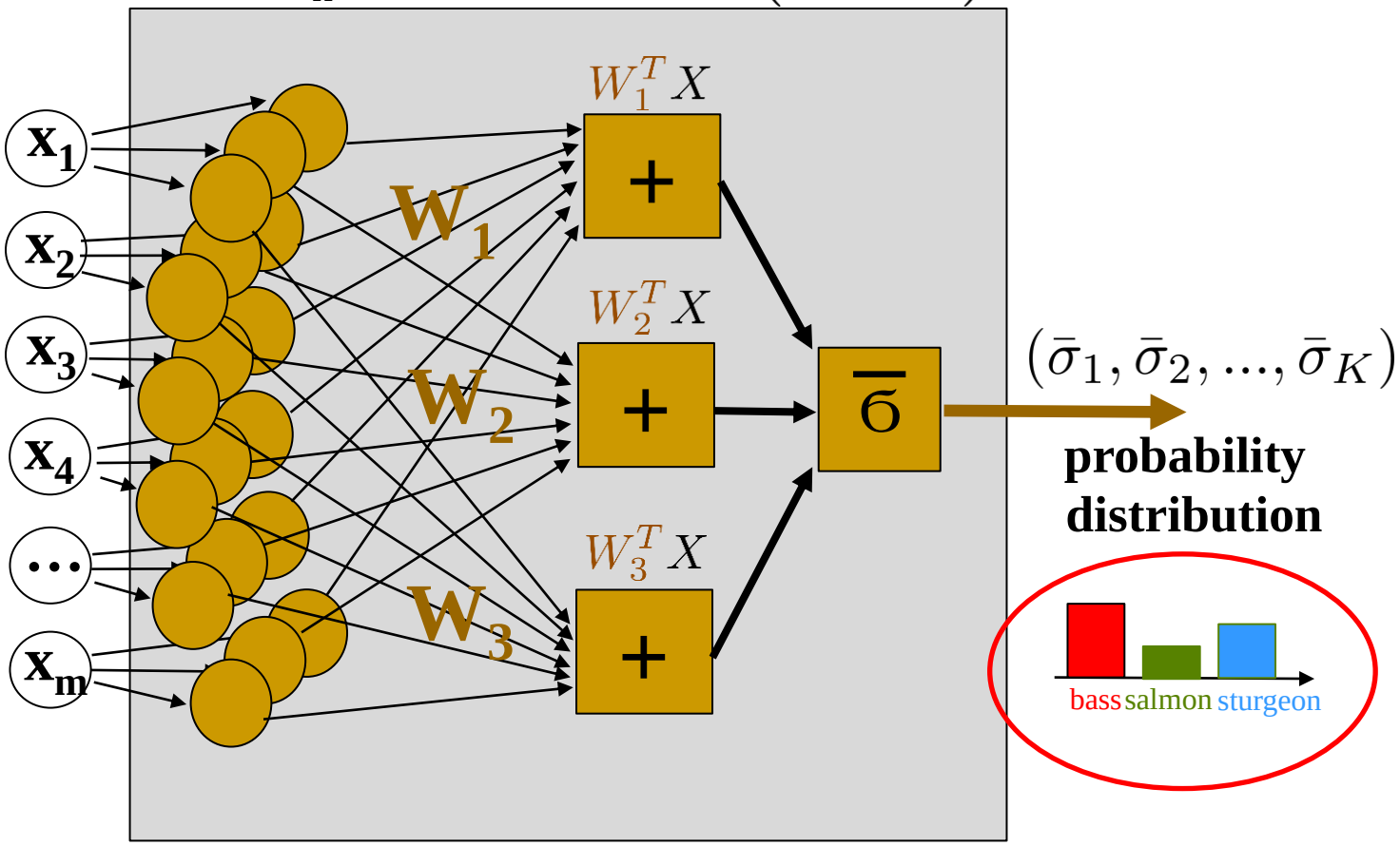


multi-label classification

We get "probability scores" ($\sigma_1,\sigma_2,\ldots,\sigma_K$) over K classes which may not add up to 1 and **must be normalized**

Common Approach: Soft-Max

use K linear transforms W_k and **soft-max** $\bar{\sigma}(\mathbf{W}X)$



multi-label classification

Notation: K rows of matrix **W** are vectors W_k so that vector **W**X has elements $W_k^T X$

Soft-Max Function $\bar{\sigma} : \mathbb{R}^K \to \Delta_K$

$$\begin{pmatrix} a^1 \\ a^2 \\ \cdots \\ a^K \end{pmatrix} \xrightarrow{\text{softmax}} \begin{pmatrix} \frac{\exp a^1}{\sum_k \exp a^k} \\ \frac{\exp a^2}{\sum_k \exp a^k} \\ \cdots \\ \frac{\exp a^K}{\sum_k \exp a^k} \end{pmatrix}$$

$\mathbf{a} \in \mathbb{R}^K$

$\bar{\sigma}(\mathbf{a}) \in \Delta_K$

Example:

$$\begin{bmatrix} -3 \\ 2 \\ 1 \end{bmatrix} \xrightarrow{\text{softmax}} \begin{bmatrix} \frac{\exp(-3)}{\exp(-3)+\exp(2)+\exp(1)} \\ \frac{\exp(2)}{\exp(-3)+\exp(2)+\exp(1)} \\ \frac{\exp(1)}{\exp(-3)+\exp(2)+\exp(1)} \end{bmatrix} = \begin{bmatrix} 0.005 \\ 0.7275 \\ 0.2676 \end{bmatrix}$$

remember soft-max question in HW4

$$\bar{\sigma}(\mathbf{a}) = \arg\min_{\mathbf{S} \in \Delta_K} -\sum_{k=1}^{K} S^k a^k - T\, H(\mathbf{S})$$

probability simplex for K classes

typically, soft max in NN uses T = 1

Soft-max normalizes logits vector **a** converting it to distribution over classes

Soft-Max Function $\bar{\sigma} : \mathbb{R}^K \to \Delta_K$

$$\begin{pmatrix} a^1 \\ a^2 \\ \dots \\ a^K \end{pmatrix} \xrightarrow{\text{softmax}} \begin{pmatrix} \frac{\exp a^1}{\sum_k \exp a^k} \\ \frac{\exp a^2}{\sum_k \exp a^k} \\ \dots \\ \frac{\exp a^K}{\sum_k \exp a^k} \end{pmatrix}$$

$\mathbf{a} \in \mathbb{R}^K$

$\bar{\sigma}(\mathbf{a}) \in \Delta_K$

NN Example:

$$\begin{pmatrix} W_1^T X \\ W_2^T X \\ \dots \\ W_K^T X \end{pmatrix} \xrightarrow{\text{softmax}} \begin{pmatrix} \frac{\exp W_1^T X}{\sum_k \exp W_k^T X} \\ \frac{\exp W_2^T X}{\sum_k \exp W_k^T X} \\ \dots \\ \frac{\exp W_K^T X}{\sum_k \exp W_k^T X} \end{pmatrix}$$

$\mathbf{W}X$

$\bar{\sigma}(\mathbf{W}X)$

$(\bar{\sigma}_1, \bar{\sigma}_2, \dots, \bar{\sigma}_K)$

Soft-max normalizes logits vector **a** converting it to distribution over classes

Soft-Max Function $\bar{\sigma} : \mathbb{R}^K \to \Delta_K$

NOTE:

soft-max generalizes sigmoid

to multi-class predictions. Indeed, consider binary perceptron with scalar linear discriminator $W^T X$ (e.g. for class 1)

$$\underset{\text{sigmoid}}{\sigma(W^T X)} = \frac{1}{1+e^{-W^T X}}$$

$$\equiv \frac{e^{\frac{1}{2}W^T X}}{e^{\frac{1}{2}W^T X} + e^{-\frac{1}{2}W^T X}} = \bar{\sigma}_1 \begin{pmatrix} \frac{1}{2}W^T X \\ -\frac{1}{2}W^T X \end{pmatrix}$$

class 1 output of **soft-max** for a combination of two linear predictors: $\frac{1}{2}W^T X$ for class 1 and $-\frac{1}{2}W^T X$ for class ¬1 (class 0)

NN Example:

$$\begin{pmatrix} W_1^T X \\ W_2^T X \\ \cdots \\ W_K^T X \end{pmatrix}$$

softmax

$$\begin{pmatrix} \frac{\exp W_1^T X}{\sum_k \exp W_k^T X} \\ \frac{\exp W_2^T X}{\sum_k \exp W_k^T X} \\ \cdots \\ \frac{\exp W_K^T X}{\sum_k \exp W_k^T X} \end{pmatrix}$$

$\mathbf{W}X$

$\bar{\sigma}(\mathbf{W}X)$

$(\bar{\sigma}_1, \bar{\sigma}_2, \ldots, \bar{\sigma}_K)$

Soft-max normalizes logits vector **a** converting it to distribution over classes

(general multi-class case)

Cross-Entropy Loss

K-label perceptron's output: $\bar{\sigma}(\mathbf{W}X^i)$ for example X^i *k*-th index

Multi-valued label $\mathbf{y}^i = k$ gives **one-hot** distribution $\bar{\mathbf{y}}^i = (0, 0, 1, 0, \ldots, 0)$

Consider two probability distributions over K classes (e.g. bass, salmon, sturgeon) : $\bar{\mathbf{y}}^i$ and $(\bar{\sigma}_1, \bar{\sigma}_2, \bar{\sigma}_3, \ldots, \bar{\sigma}_K)$

$$\Pr(\mathbf{x}^i \in \text{Class } k \mid W) = \bar{\sigma}_k(WX^i)$$

Total loss:

$$L(W) = \sum_{i \in \text{train}} \overbrace{\sum_k -\bar{\mathbf{y}}_k^i \ln \bar{\sigma}_k(WX^i)}^{\textbf{cross entropy}}$$

$$\Rightarrow \quad L(W) = -\sum_{i \in \text{train}} \ln \bar{\sigma}_{\mathbf{y}^i}(WX^i)$$

NOTE: same as **Seed Loss** in interactive segmentation (Topic 9, slide 107)

sum of **Negative Log-Likelihoods (NLL)**

Multi-label Classification so-far

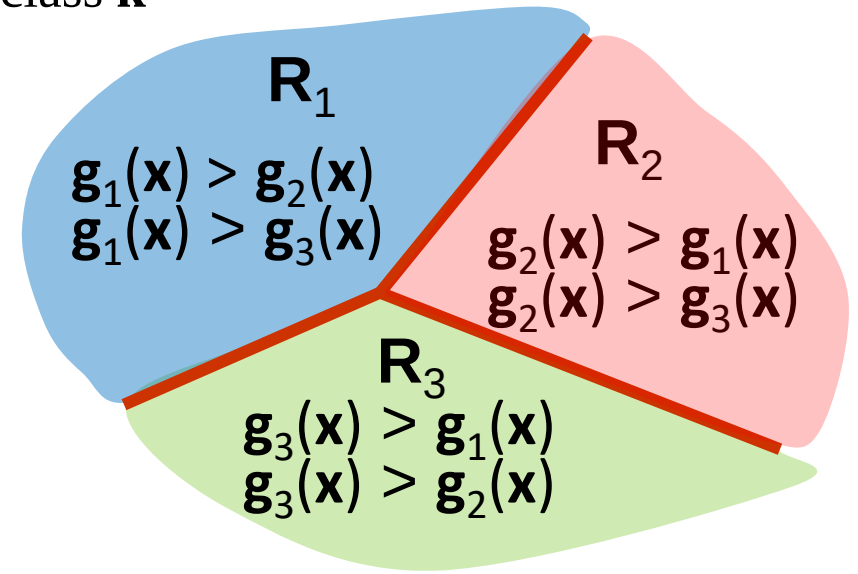
Define **K** linear transforms (functions)

$$\mathbf{g}_k(\mathbf{x}) = W_k^T X \quad \text{for } \mathbf{k} = 1, 2, \ldots \mathbf{K}$$

- Soft-max $\bar{\sigma}$ assign **x** to class **k** if

$$\mathbf{g}_k(\mathbf{x}) > \mathbf{g}_m(\mathbf{x}) \text{ for all } \mathbf{m} \neq \mathbf{k}$$

- Let $\mathbf{R}_k$ be decision region for class **k**
  - all points in $\mathbf{R}_k$ assigned to class **k**

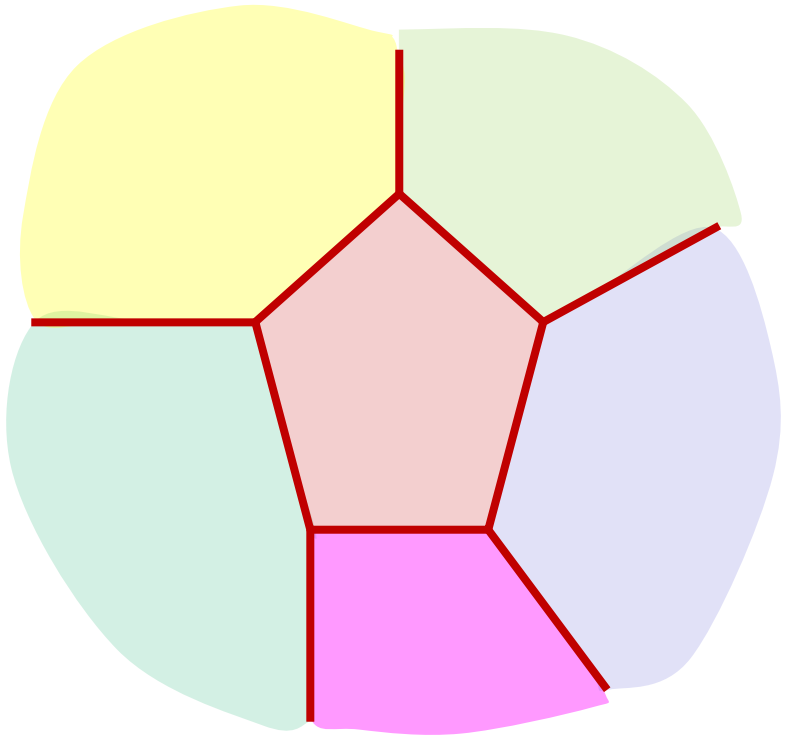


slide credit: O.Veksler

Multi-label Classification so-far

Can be shown that decision regions are convex, spatially contiguous, with **linear boundaries** between any two classes

Limitation of single layer NN
(perceptron)



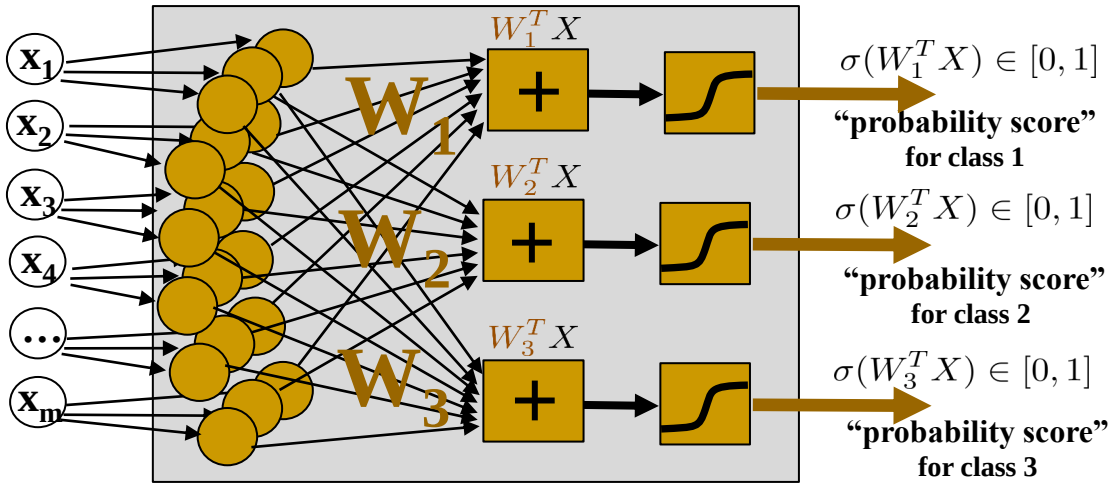
slide credit: O.Veksler

Towards

Multilayer Neural Networks

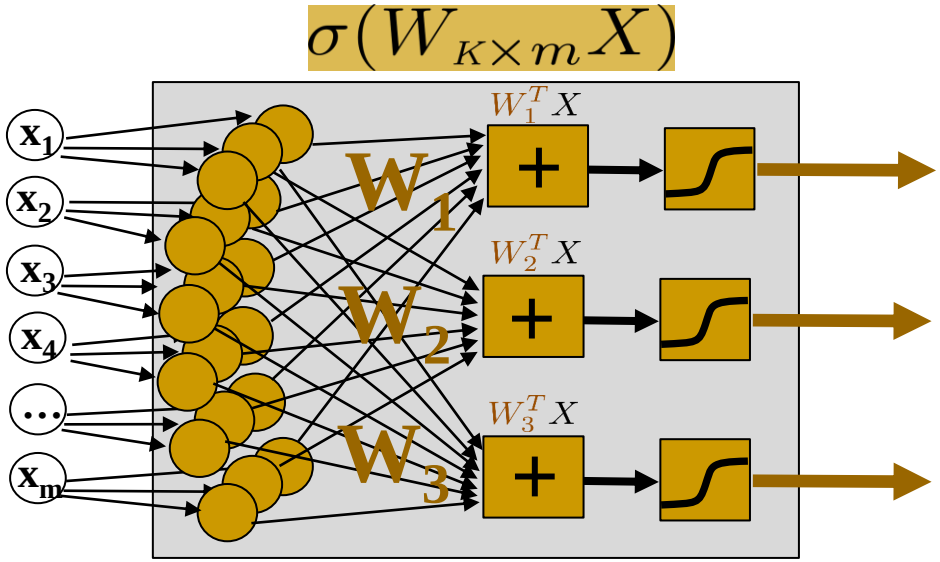
Remember:

Single layer multi-class NNs



Remember:

Single layer multi-class NNs



$\mathbf{x} \in \mathcal{R}^m$

$(\sigma_1, \sigma_2, ..., \sigma_K) \in [0,1]^K$

"probability scores"
(no normalization)
computed by *K* perceptrons

Notation: K x $(m+1)$ matrix $W = \begin{bmatrix} W_1^T \\ W_2^T \\ \cdots \\ W_K^T \end{bmatrix}$

we will use notation $W_{K\times m}$

emphasizing
size of output and input
while implicitly **assuming**
one extra dimension for *bias*
and 1 in *homogeneous* input

where rows are $W_k^T = [\mathbf{w}_0^k, \mathbf{w}_1^k, ..., \mathbf{w}_m^k]$

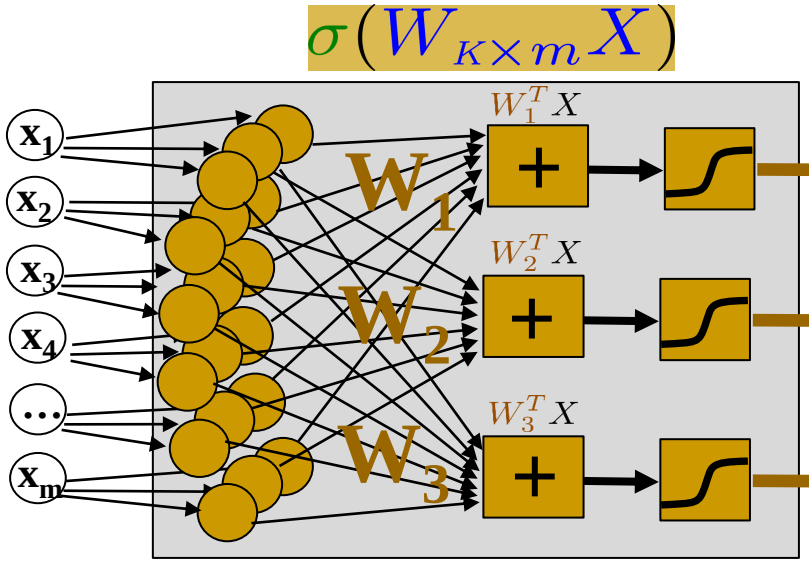
"bias"

feature vector (input) $X^T = [1, \mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_m]$

***homogeneous* representation of *m*-dimensional feature vector x**

Remember:

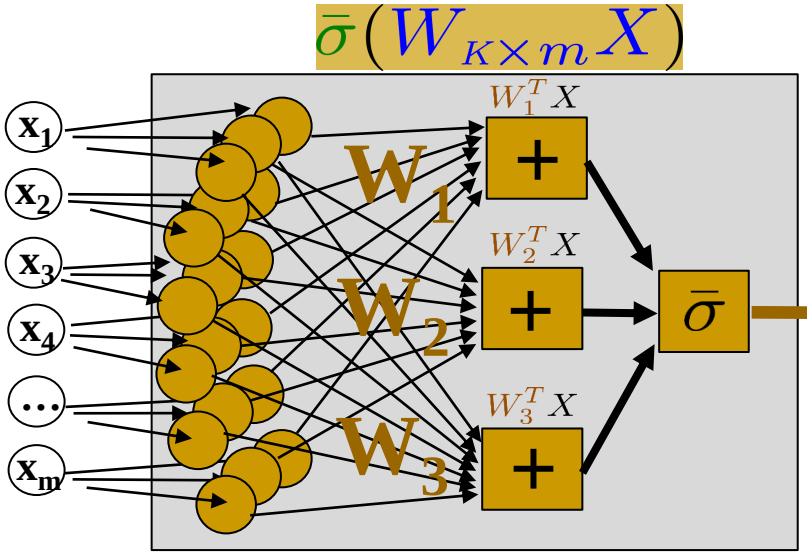
Single layer multi-class NNs



$\mathbf{x} \in \mathcal{R}^m$

$(\sigma_1, \sigma_2, ..., \sigma_K) \in [0,1]^K$

"probability scores"
(no normalization)
computed by *K* perceptrons,
that is, **sigmoid** function
applied to **linear transform** $W_{K\times m}X$



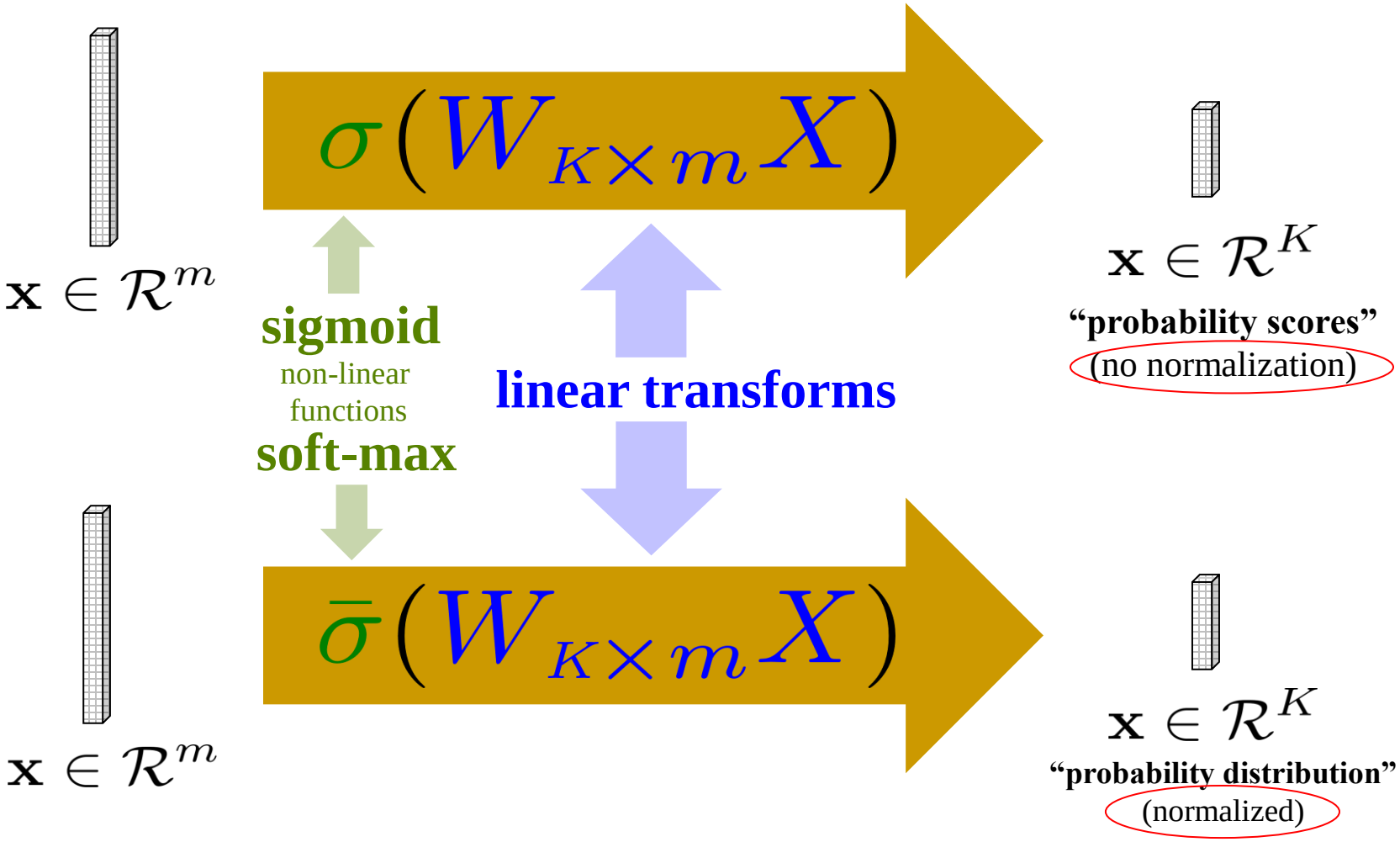
$\mathbf{x} \in \mathcal{R}^m$

$(\bar{\sigma}_1, \bar{\sigma}_2, ..., \bar{\sigma}_K) \in [0,1]^K$

"probability distribution"
(normalized)
computed by **soft max** function
applied to **linear transform** $W_{K\times m}X$

Remember:

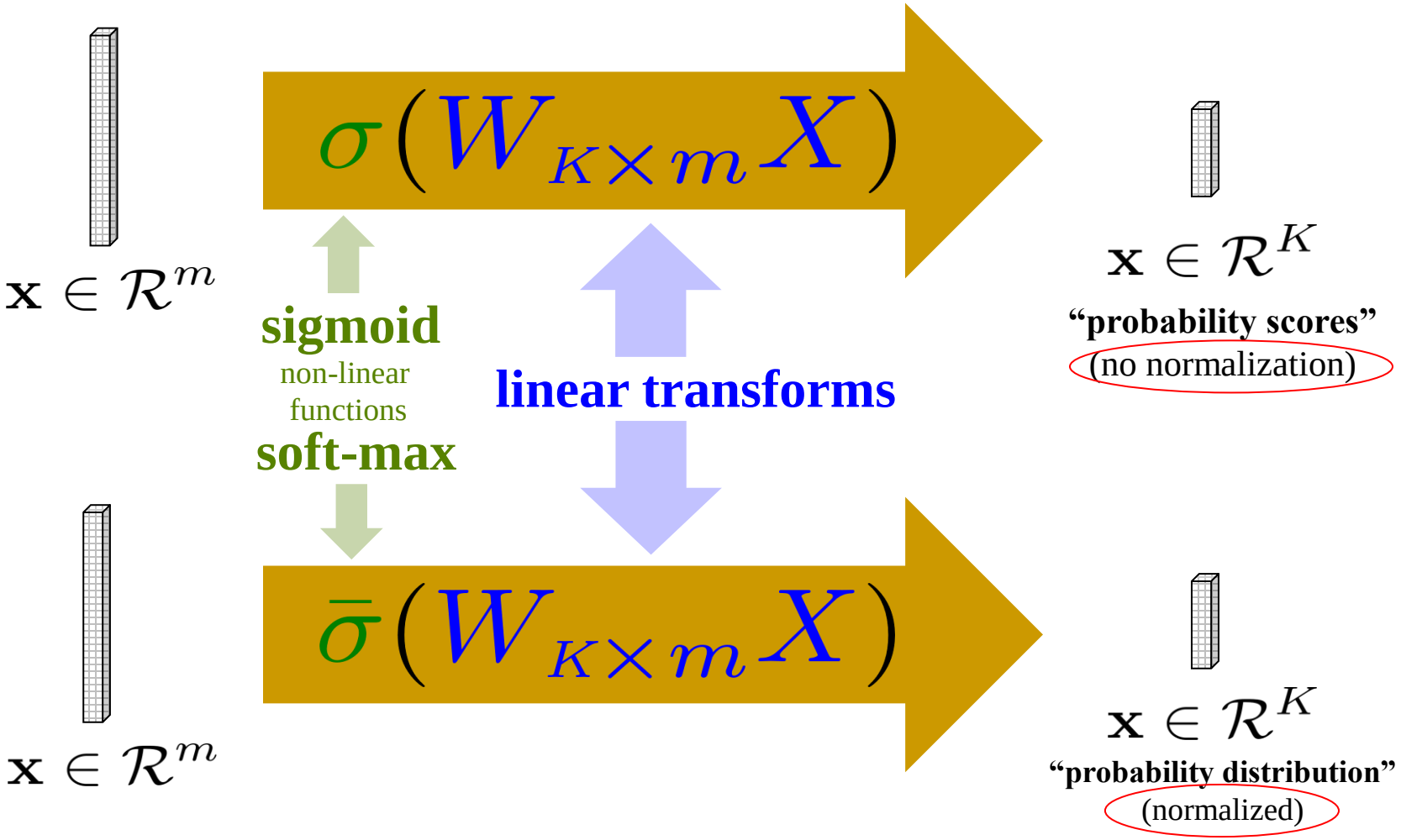
Single layer multi-class NNs



Side Question: what losses can be used to train these (linear) single-layer NN classifiers?

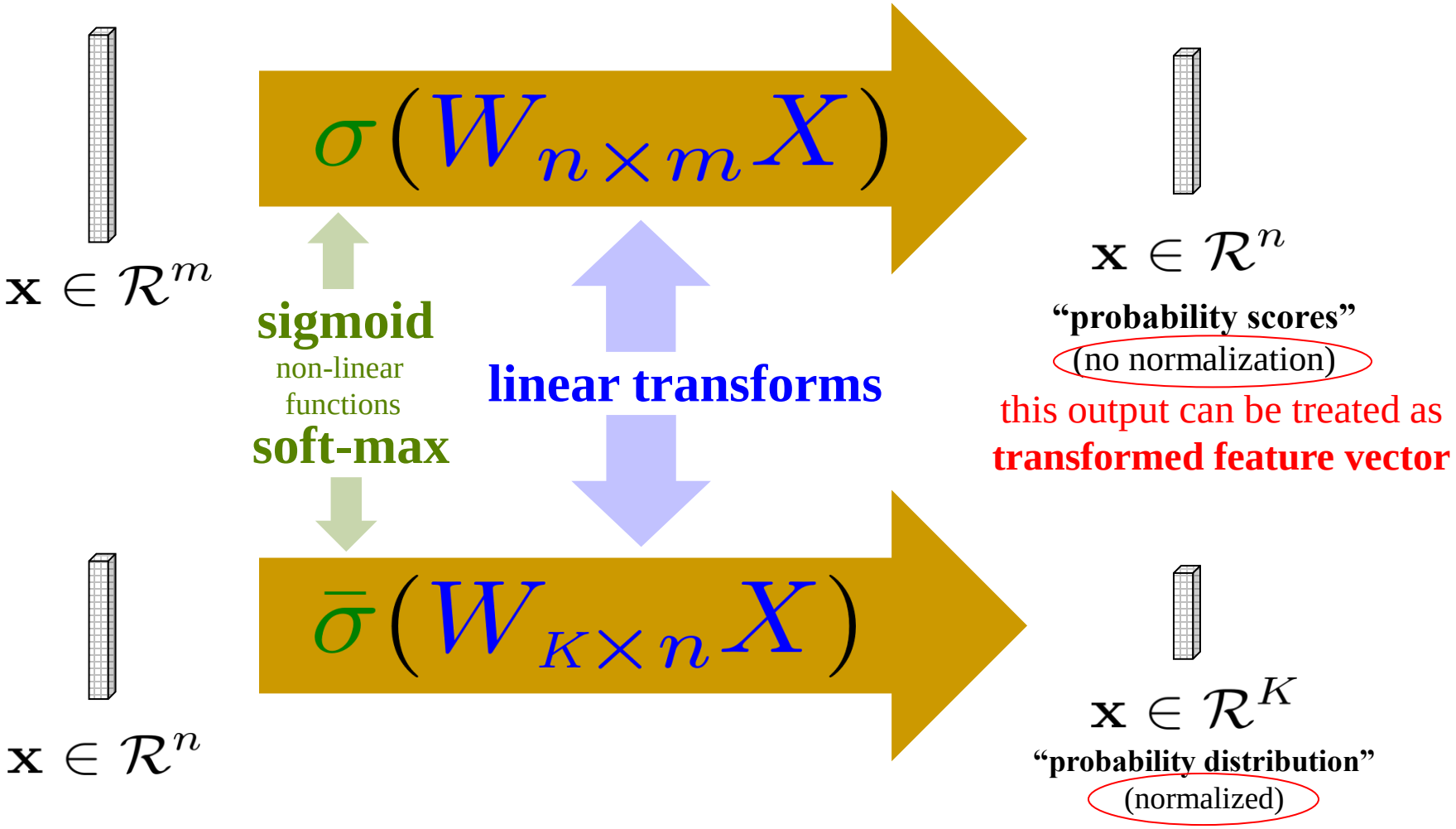
Multi-layer NNs ?

Motivated by neurons, can we **link perceptrons**?



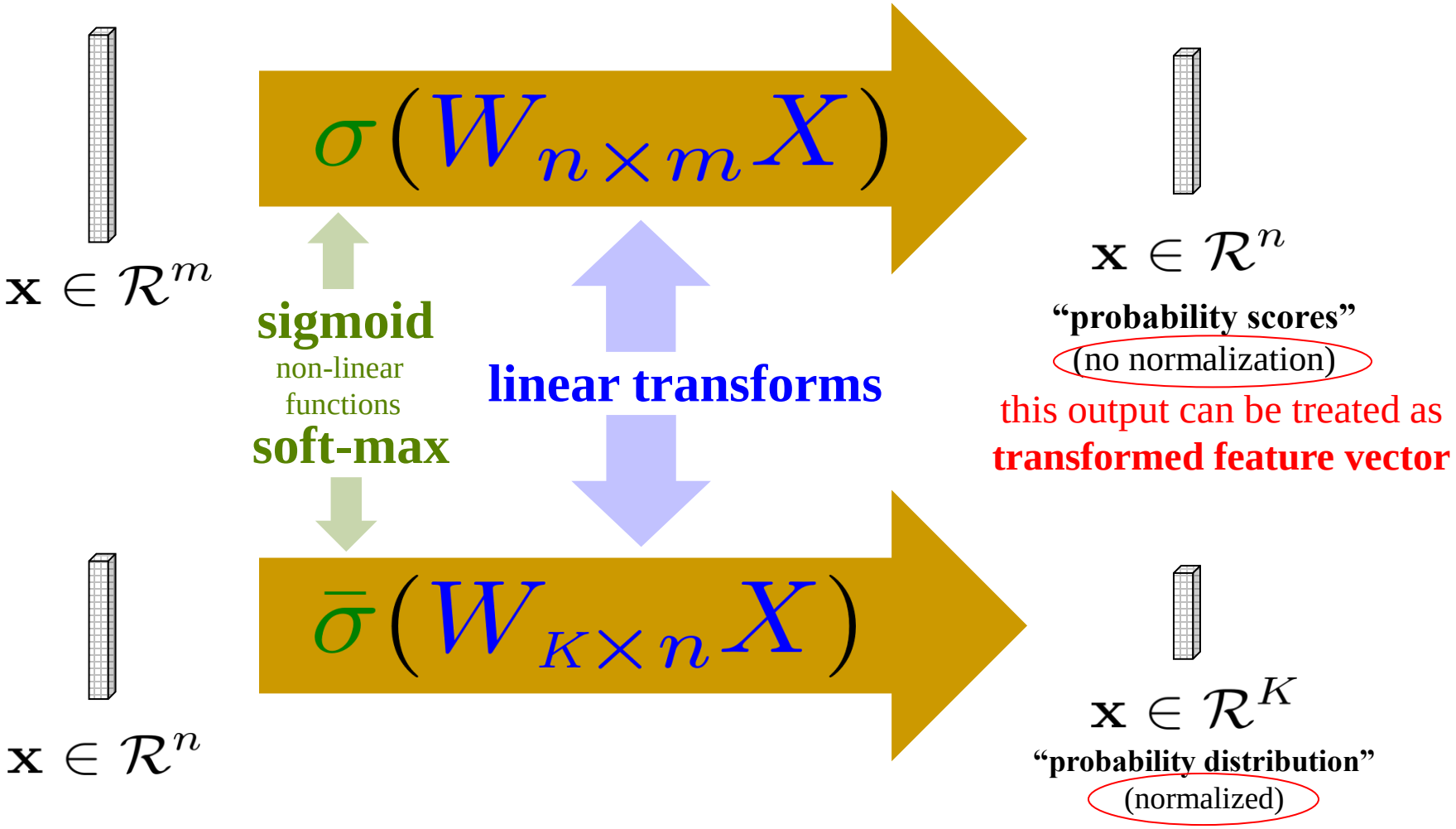
Multi-layer NNs ?

Motivated by neurons, can we **link perceptrons**?



Multi-layer NNs ?

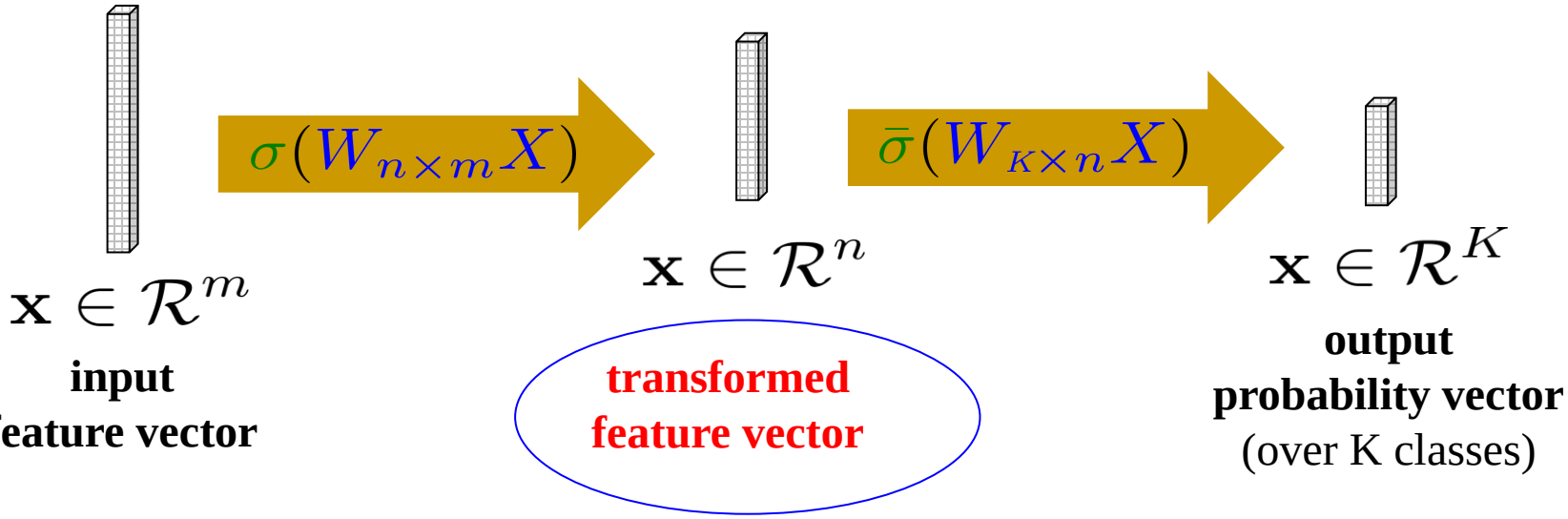
Motivated by neurons, can we **link perceptrons**?



Multi-layer NNs ?

Motivated by neurons, can we **link perceptrons**?

NOTE: in the 60's the idea of multi-layer NN was discredited by one (now notoriously famous) book based on their **conjecture** that a **combination of linear classifiers can not produce a non-linear one**.

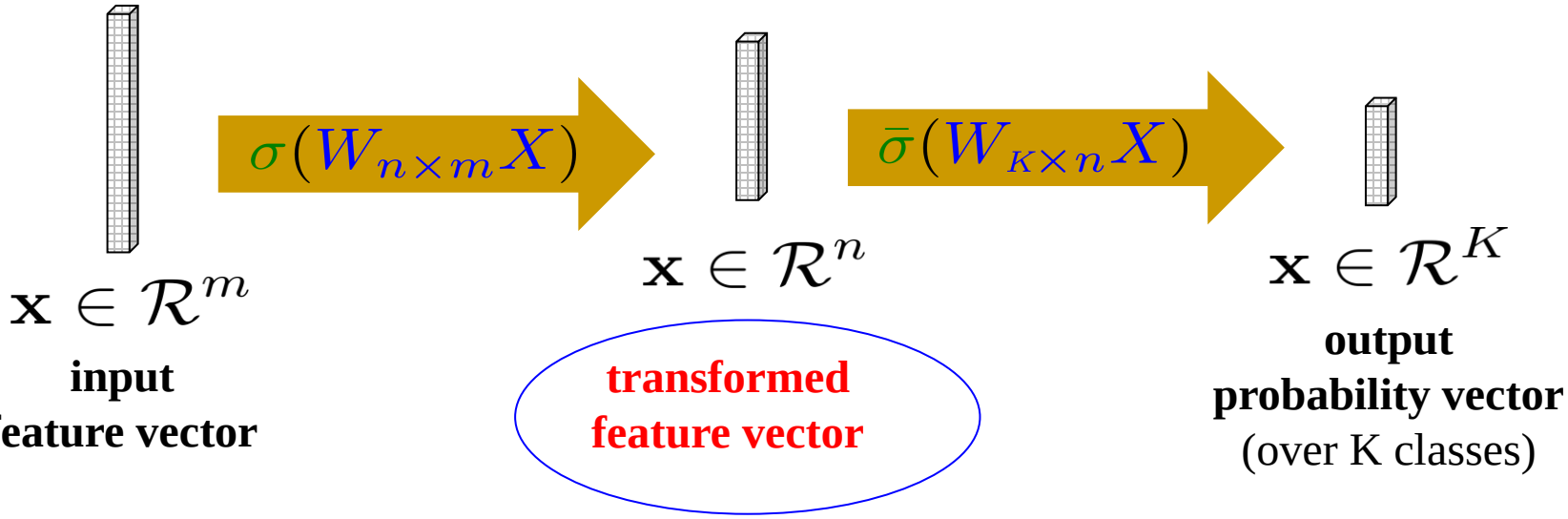


due to non-linear decision function (e.g. б)
this **feature transformation is non-linear**

Multi-layer NNs ?

Motivated by neurons, can we **link perceptrons**?

NOTE: in the 60's the idea of multi-layer NN was discredited by one (now notoriously famous) book based on their **conjecture** that a **combination of linear classifiers can not produce a non-linear one**.



$$\bar{\sigma}(\underbrace{W_{K\times n}\sigma(W_{n\times m}X)}) \not\equiv \bar{\sigma}(\underbrace{W'_{K\times m}X})$$

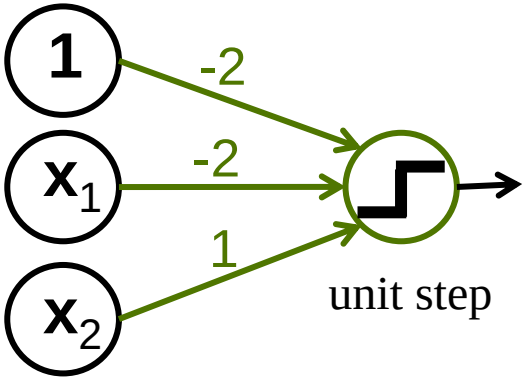
this **feature transformation is non-linear**

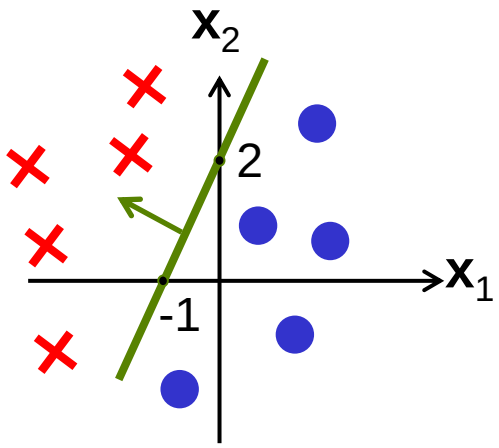
there is no equivalent linear transformation

Multi-layer NN: Nonlinear Boundary Example

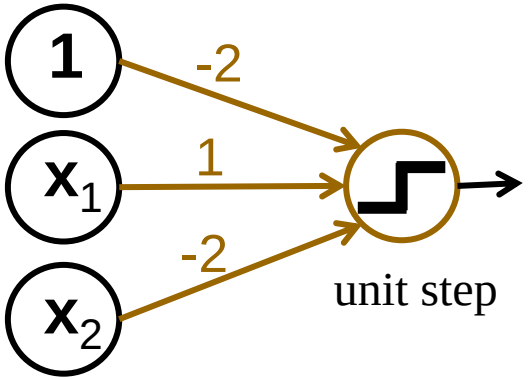
First, consider two single-layer perceptrons (each is a linear classifier) :

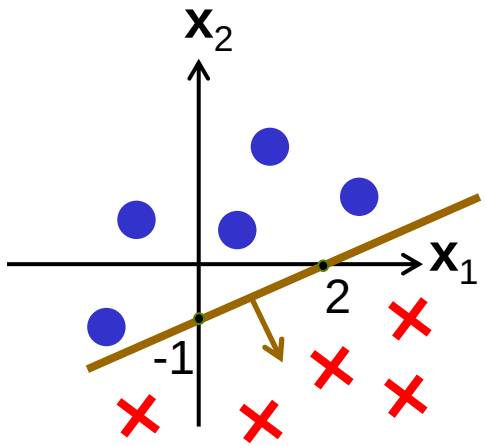
$-2x_1 + x_2 - 2 > 0 \Rightarrow$ class 1





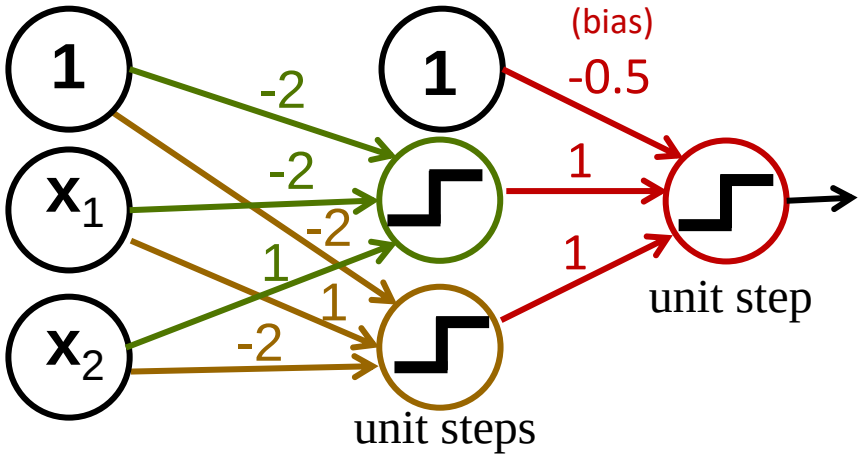
$x_1 - 2x_2 - 2 > 0 \Rightarrow$ class 1



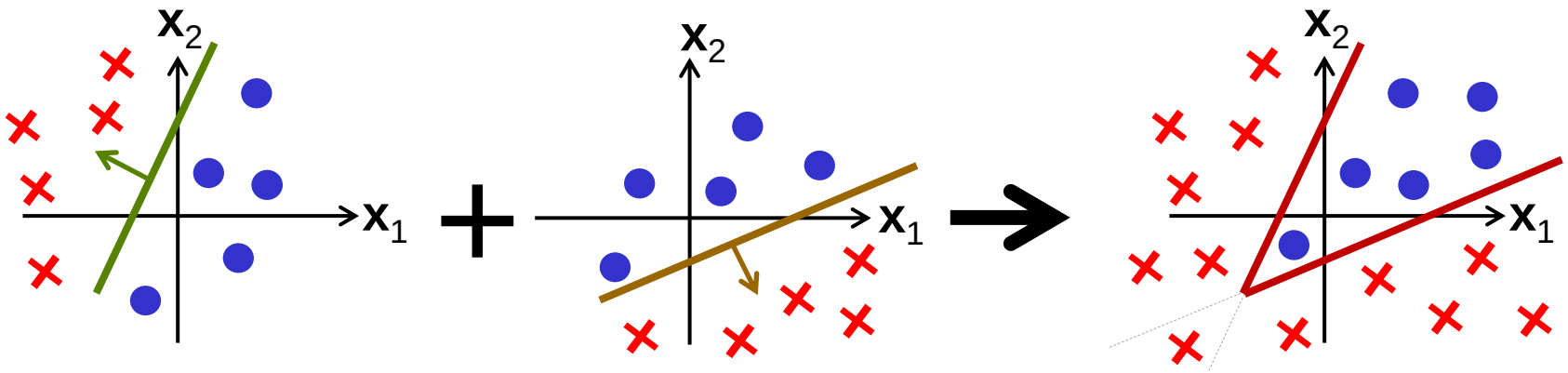


Multi-layer NN: Nonlinear Boundary Example

Combine the same two perceptrons inside 2-layer NN

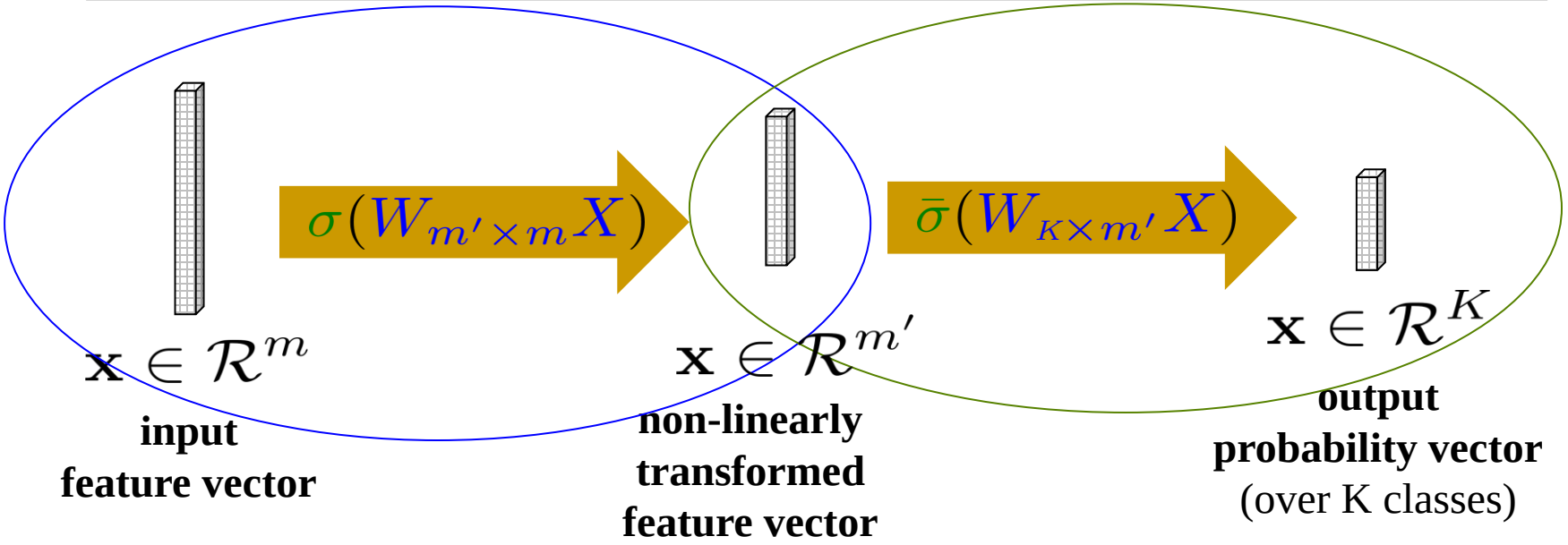


Question: what does **layer 2** do?



non-linear boundary between two classes

Multi-layer NN: Non-Linear Feature Embedding



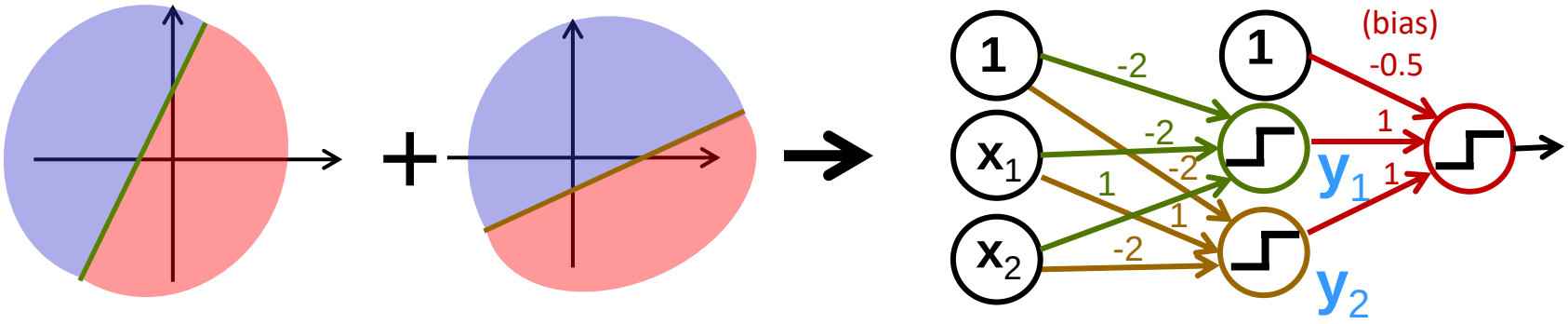
new "embedding"
of the features

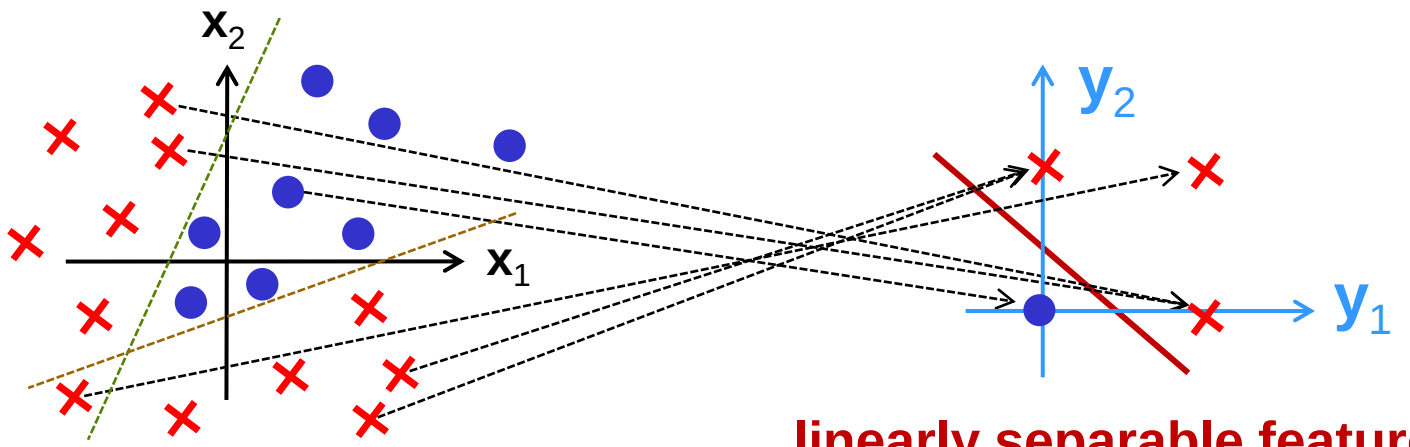
Interpretation:

- layer 1 maps input features to new (transformed) features
- layer 2 applies linear classifier to the new features

Multi-layer NN: Non-Linear Feature Embedding

consider our earlier two-layer NN example:





can't separate linearly
in the original feature space

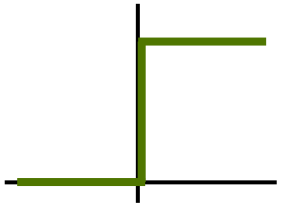
linearly separable features
in the new "**embedding space**"

NOTE: unlike our kernel approach in topic 9
now we might learn such embeddings!

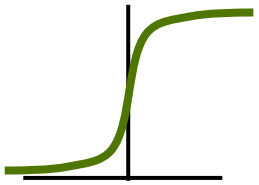
Multi-layer NN: Activation Functions

In the interior (hidden) layer, non-linear decision function is now called **activation function** (representing neuron "activation")

- $u()$ – step function does not work for gradient descent

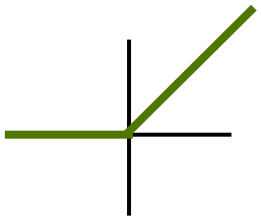


- $\sigma()$ - sigmoid function allows gradient descent

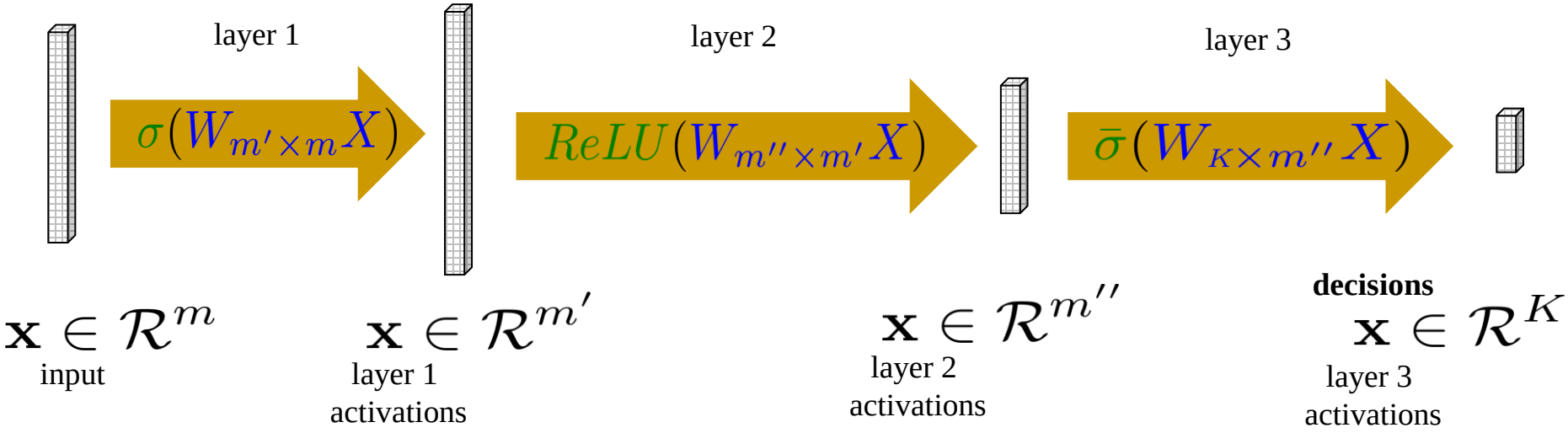


NOTE: activation functions do not need to make 0-1 decisions

- ReLU() - Rectified Linear Unit is popular
  - the simplest kind of non-linear function (2-piecewise linear)
  - gradients do not saturate for positive half-interval
  - must be careful with learning rate, otherwise many units can become "dead", i.e. always output 0



Multi-layer NNs



- non-linear classification
- non-linear feature embedding (new features)
- optimization w.r.t. weights *W* by ***backpropagation***

(gradient w.r.t. different layers is computed via ***chain rule***)

[Rumelhart, Hinton, Williams 1986]

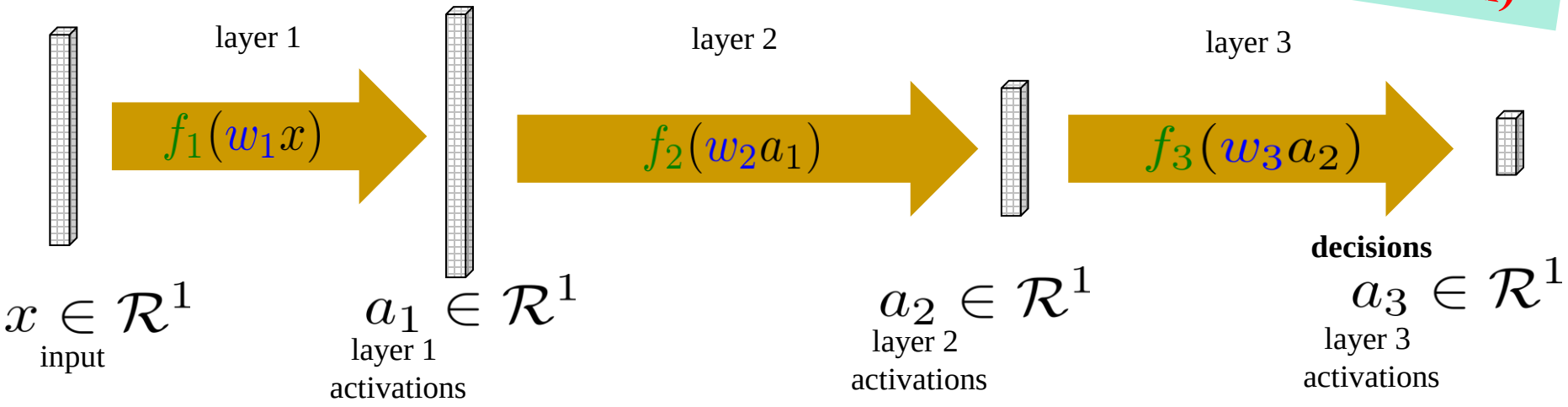
- automatic differentiation ☺

Toy (scalar) illustration of **backpropagation**

i.e. chain rule for gradient descent updates of NN weights

UNIVERSITY OF WATERLOO

OPTIONAL SLIDE (can skip 12 min)



Assume: scalar weights, $w_1,\ w_2,\ w_3$ scalar activation functions, $f_1,\ f_2,\ f_3$ scalar loss $L(y, f) \equiv L_y(f)$

Scalar NN model (composition function) $f_3(w_3 f_2(w_2 f_1(w_1 x)))$ with a_3, a_2, a_1

Optimization of loss $l(w_1, w_2, w_3) := L_y(f_3(w_3 f_2(w_2 f_1(w_1 x))))$

chain rule

$$\frac{\partial l}{\partial w_3} = L'_y\, f'_3\, a_2$$

$$\frac{\partial l}{\partial w_2} = L'_y\, f'_3\, w_3\, f'_2\, a_1$$

$$\frac{\partial l}{\partial w_1} = L'_y\, f'_3\, w_3\, f'_2\, w_2\, f'_1\, x$$

Backward pass updates weights

using part. derivatives $\frac{\partial l}{\partial w_i} = \frac{\partial l}{\partial w_{i+1}} \frac{w_{i+1}\, f'_i}{a_i}$ implied by the chain rule

Forward pass updates activations

$$a_i = f_i(w_i\, a_{i-1})$$

Training/Optimization Protocols

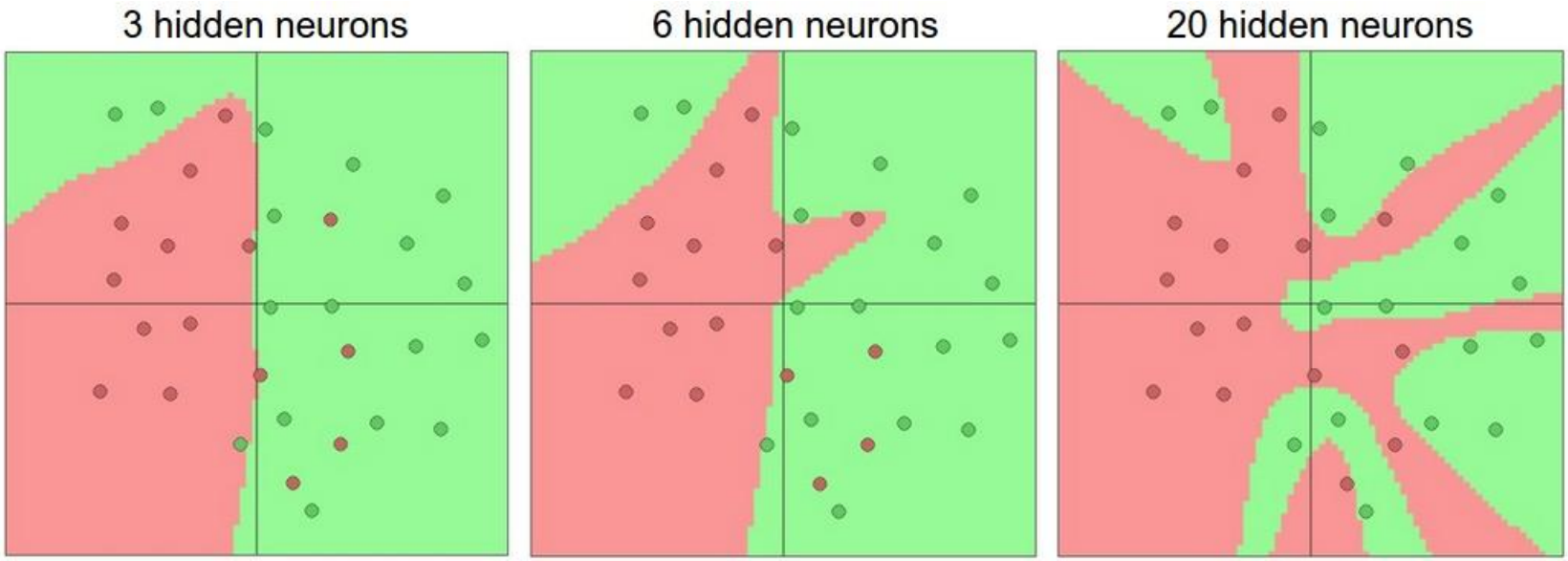
$$L(\mathbf{w}) = \sum_{i \in \text{train}} L(\mathbf{y}^i, \mathbf{f}(\mathbf{w}, \mathbf{x}^i))$$

- "Full" Gradient Descent Protocol
  - weights are updated only after all training examples are processed (**epoch**)
  - now days training sets are typically so large that this is not even practical

- Batch Protocol
  - Divide data in batches, and update weights after processing each batch
  - Batches are chosen randomly (Stochastic Gradient Descent)
    - empirically, known to work better than fixed ordering
  - Weighs get changed more frequently than "full" gradient
    - may be less stable, often requires smaller learning rate
    - helps to prevent over-fitting in practice, think of it as "noisy" gradient
  - One iteration over all batches is called an **epoch**

Network Size

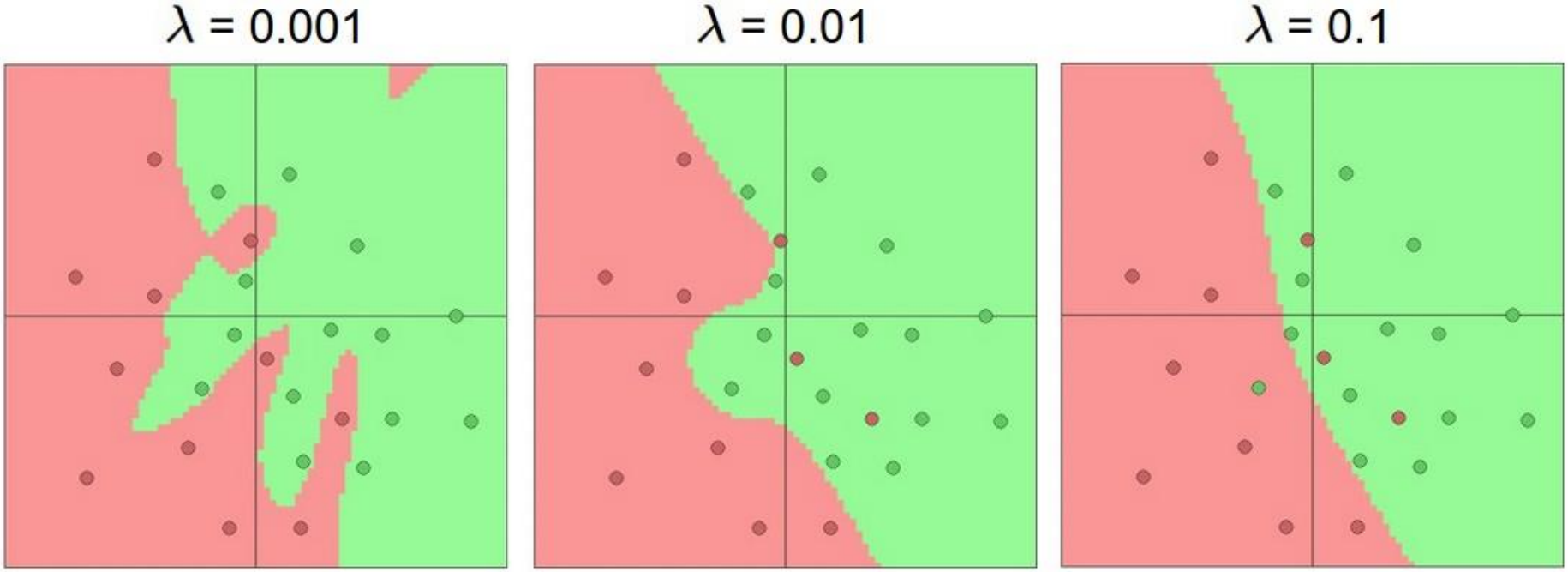
- Larger networks are more prone to overfitting



slide credits: O. Veksler

Regularization

- Less overfitting for sparser/simpler network with less units
- Works better by adding weight regularization $\frac{\lambda}{2}\|\mathbf{w}\|^2$ to the loss



- During gradient descent, subtract $\lambda\mathbf{w}$ from each weight $\mathbf{w}$
  - intuitively, implements **weight decay**

slide credits: O. Veksler

Convolutional Neural Networks (CNNs)

Towards Convolutional Neural Networks

Remember (slide from Topic 3):

…… **convolution** operation is defined ……

$$g[i,j] = \sum_{u=-k}^{k} \sum_{v=-k}^{k} h[u,v] \cdot f[i-u, j-v]$$

It is written as:

$$g = h * f$$

$$= \sum_{u=-k}^{k} \sum_{v=-k}^{k} h[-u,-v] \cdot f[i+u, j+v]$$

- You should also remember that convolution is a **linear operation**
  Thus, it can be written as $g = \mathbf{W} f$
- CNNs use **convolutions as very sparse linear transformations**.
- In the context of (large) images, such NN design is motivated by **efficiency** and **neighborhood processing - we will learn filters**

Early Work on CNNs

Fukushima (1980) – Neo-Cognitron

LeCun (1998) – Convolutional Networks (ConvNets)

- similarities to Neo-Cognitron
- success on character recognition

Other attempts at deeply layered Networks trained with backpropagation

- not much success (e.g. very slow, diffusion of gradient)

Recent work has shown significant training improvements with various tricks (drop-out, unsupervised learning of early layers, etc.)

ConvNets: Use Domain Knowledge for NN design

- Convnets exploit prior knowledge about image recognition tasks into network architecture design
  - local spatial connectivity, grid structure, geometry
  - weight constraints (translational invariance)

- Prejudices the network towards the particular way of solving the problem that we had in mind

Domain specific way of enforcing network regularization (network sparsity/simplicity)

Convolutional Network: Motivation

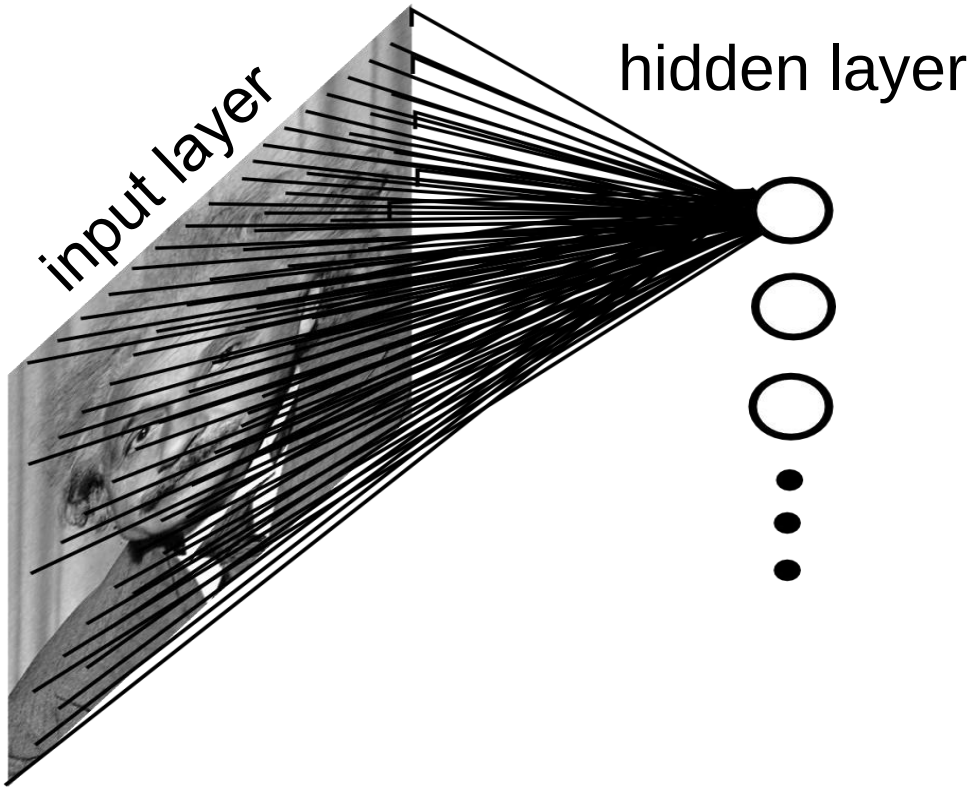
Consider a **fully connected network** (most weights W[i,j] ≠ 0)

Example: 200 by 200 image, $4x10^4$ connections to one hidden unit

For 10^5 hidden units → $4x10^9$ connections

But distant pixels are unrelated (correlations are mostly local)

Do not waste resources by connecting unrelated pixels



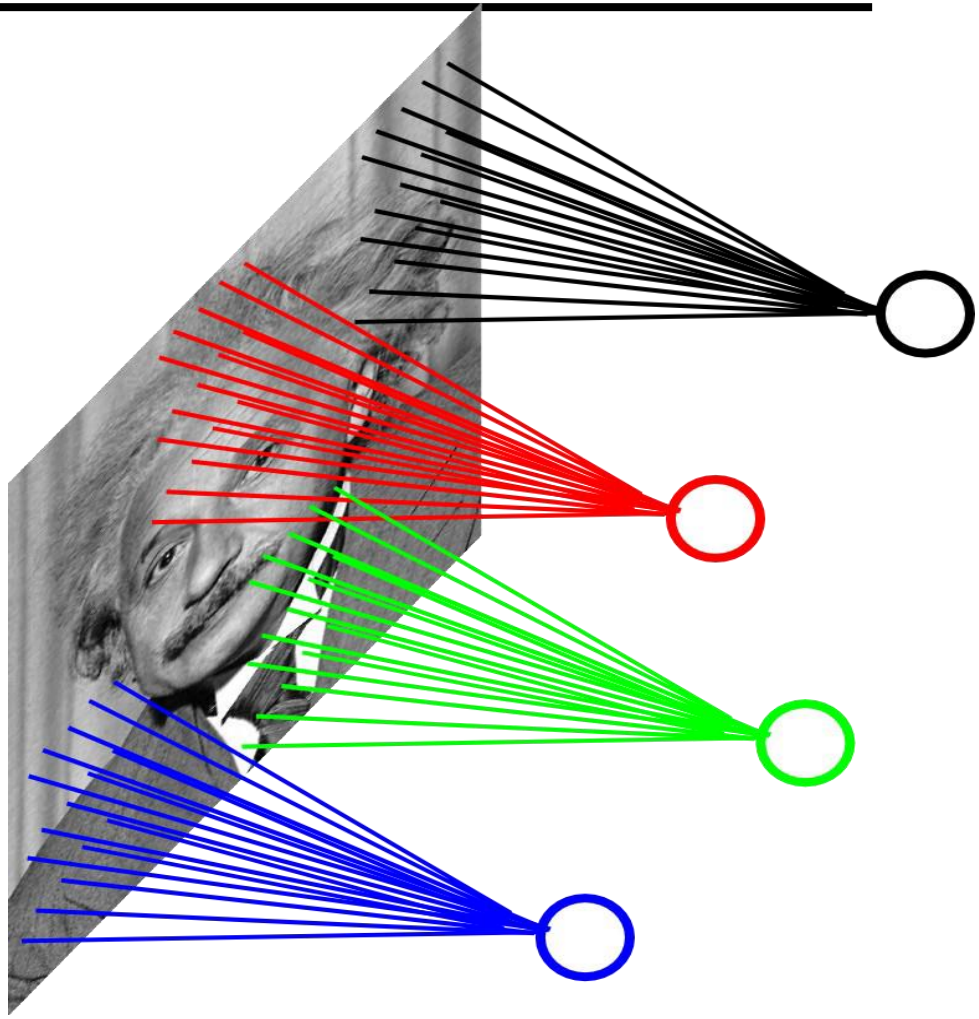
Convolutional Network: Motivation

Connect only pixels in a local patch, say 10x10

For 200 by 200 image, 10^2 connections to one hidden unit

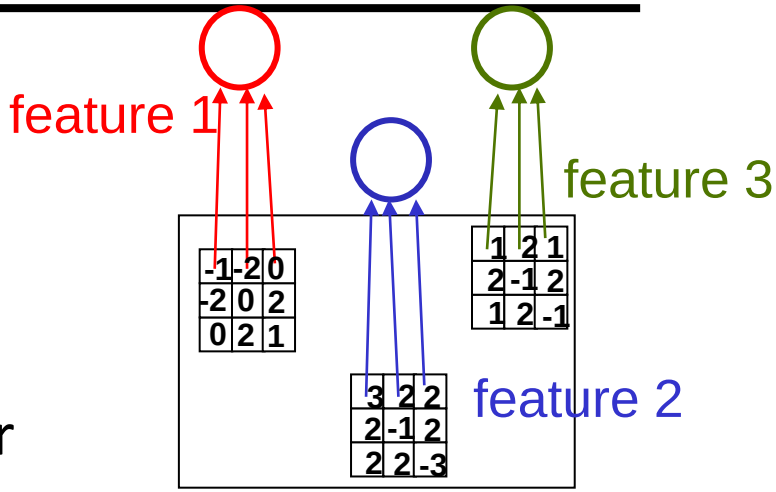
For 10^5 hidden units $\rightarrow$ 10^7 connections

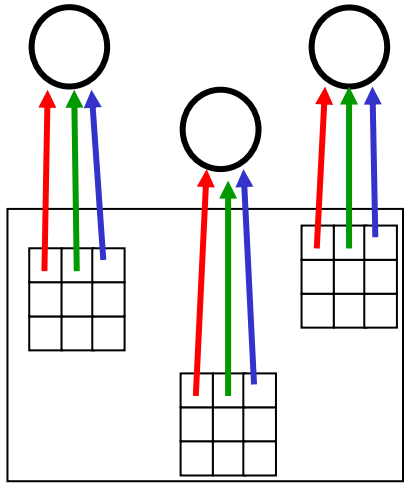
- contrast with $4x10^9$ for fully connected layer
- factor of 400 decrease



Convolutional Network: Motivation

- Intuitively, each neuron learns a good feature (a filter) in one particular location
- If a feature is useful in one image location, it should be useful in all other locations
  - *stationarity*: statistics is similar at different locations
- Idea: make all neurons detect the **same feature at different positions**
  - i.e. **share parameters** (network weights) across different locations
  - greatly reduces the number of tunable parameters to learn





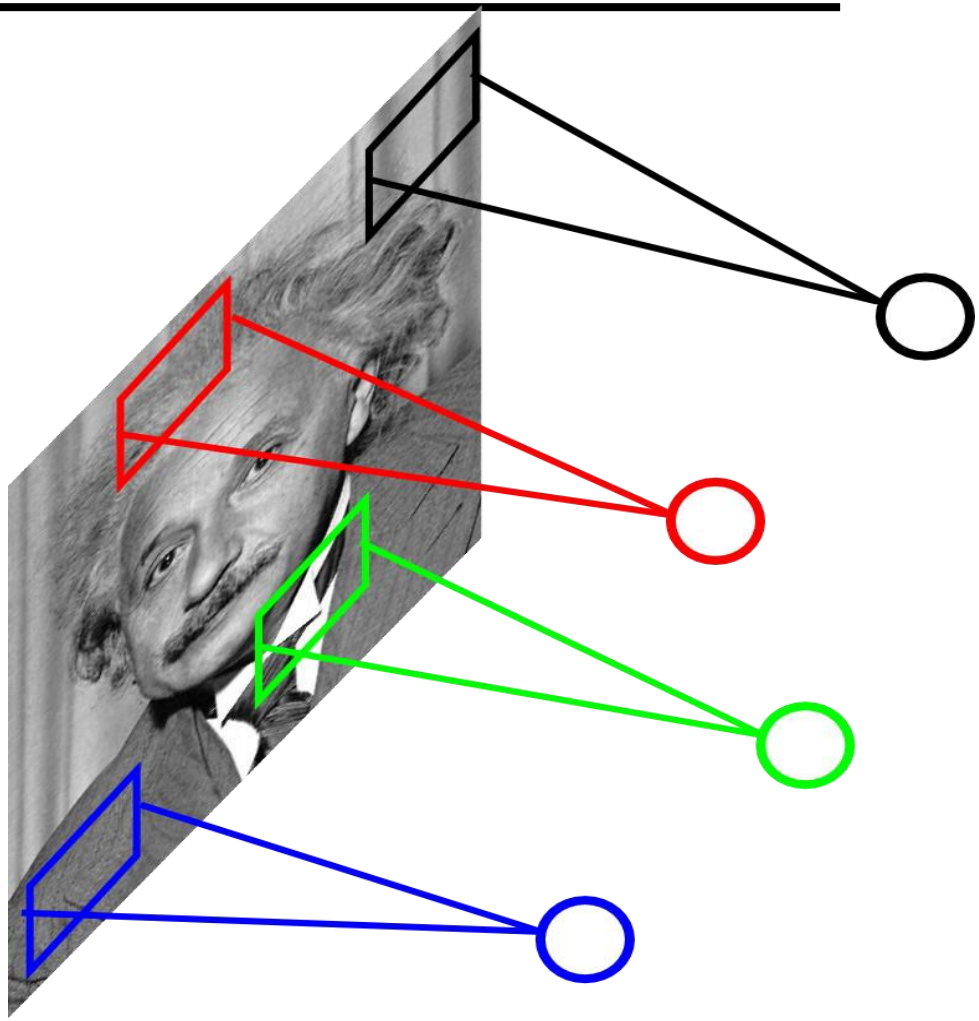
red connections have equal weight
green connections have equal weight
blue connections have equal weight

ConvNets: Weight Sharing

Much fewer parameters to learn

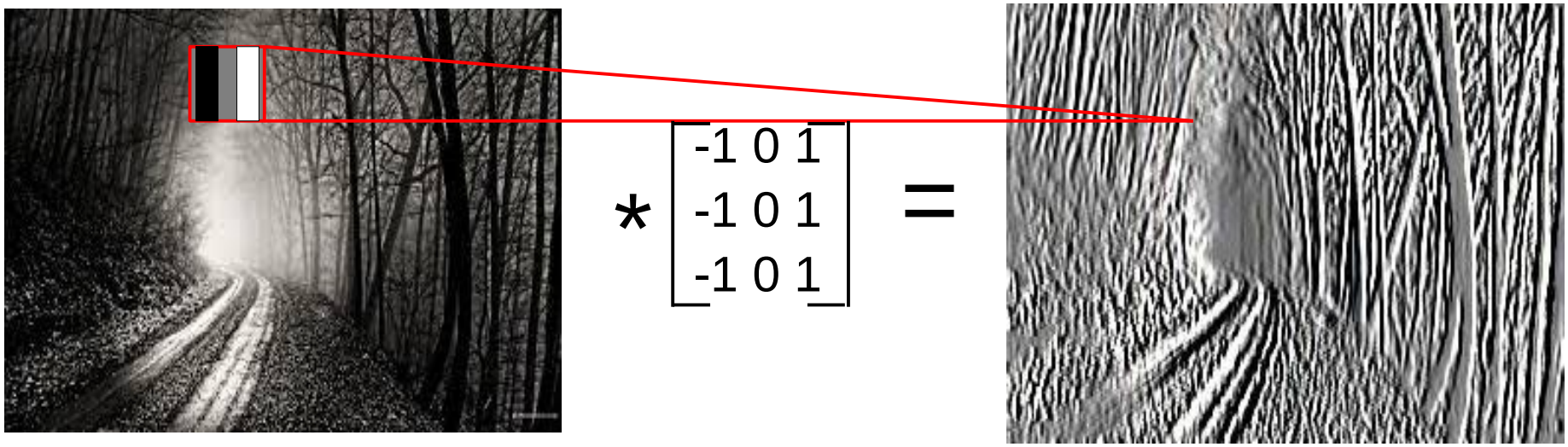
For 10^5 hidden units and 10x10 patch

- 10^7 parameters to learn without sharing
- 10^2 parameters to learn with sharing



Filtering via Convolution Recap

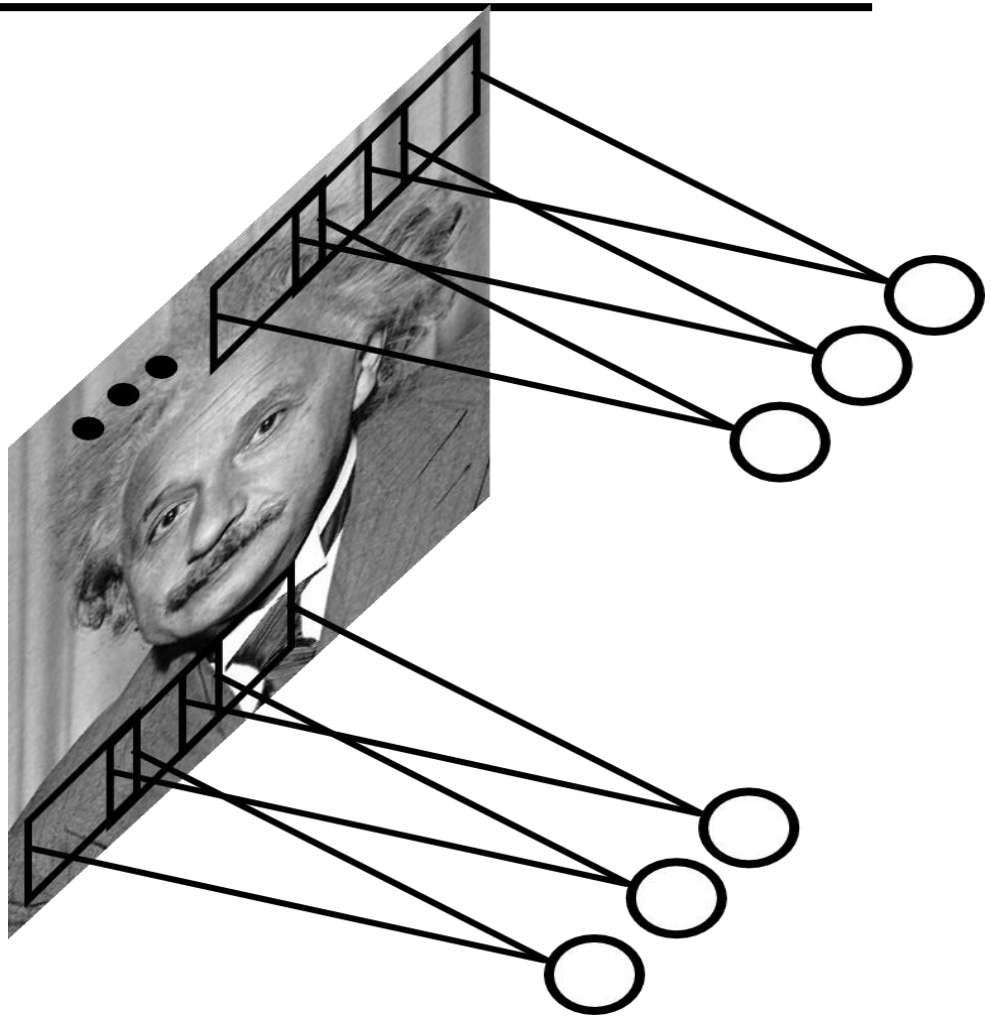
Recall filtering with convolution for feature extraction



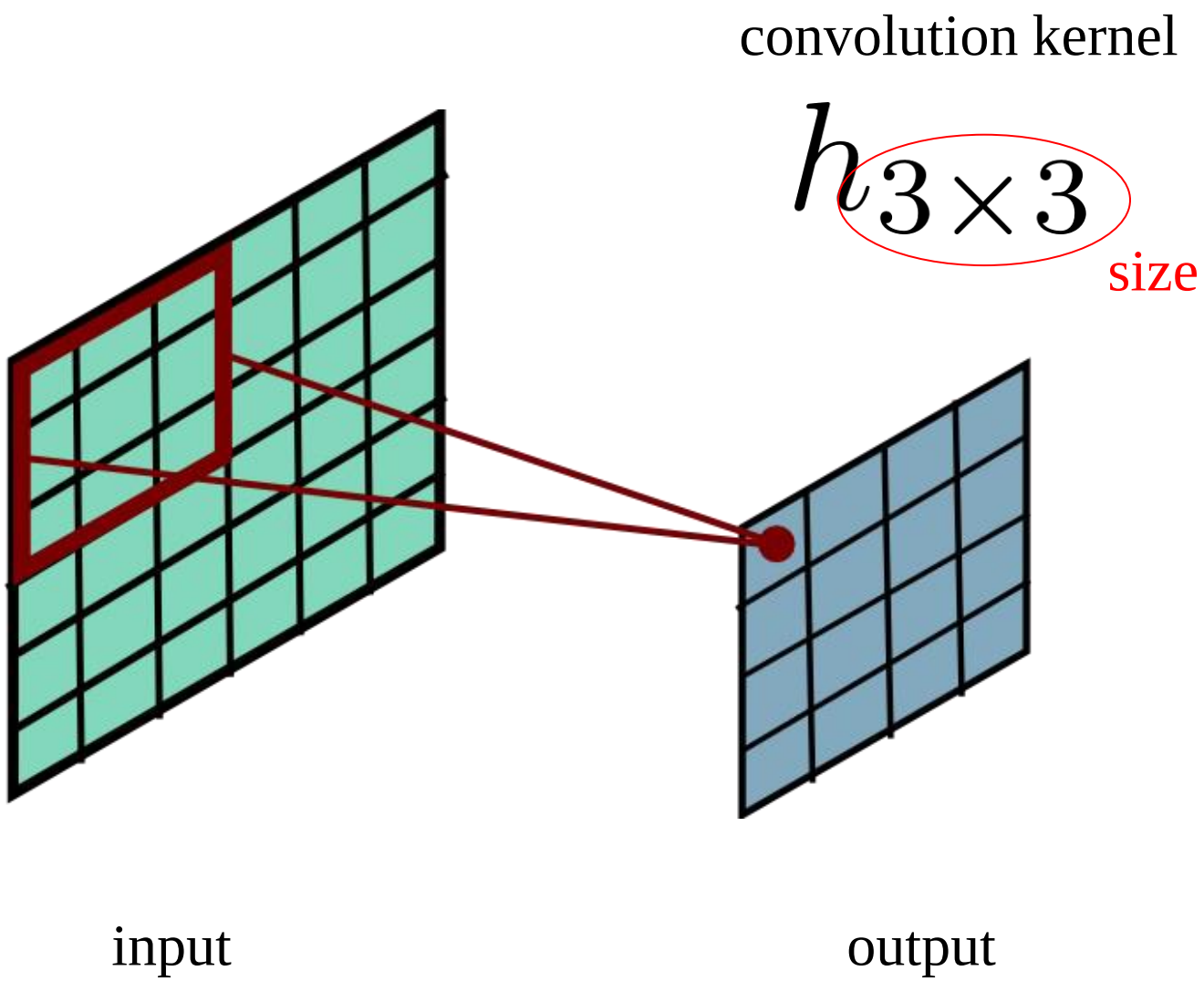
Convolutional Layer

Note similarity to convolution with some fixed filter

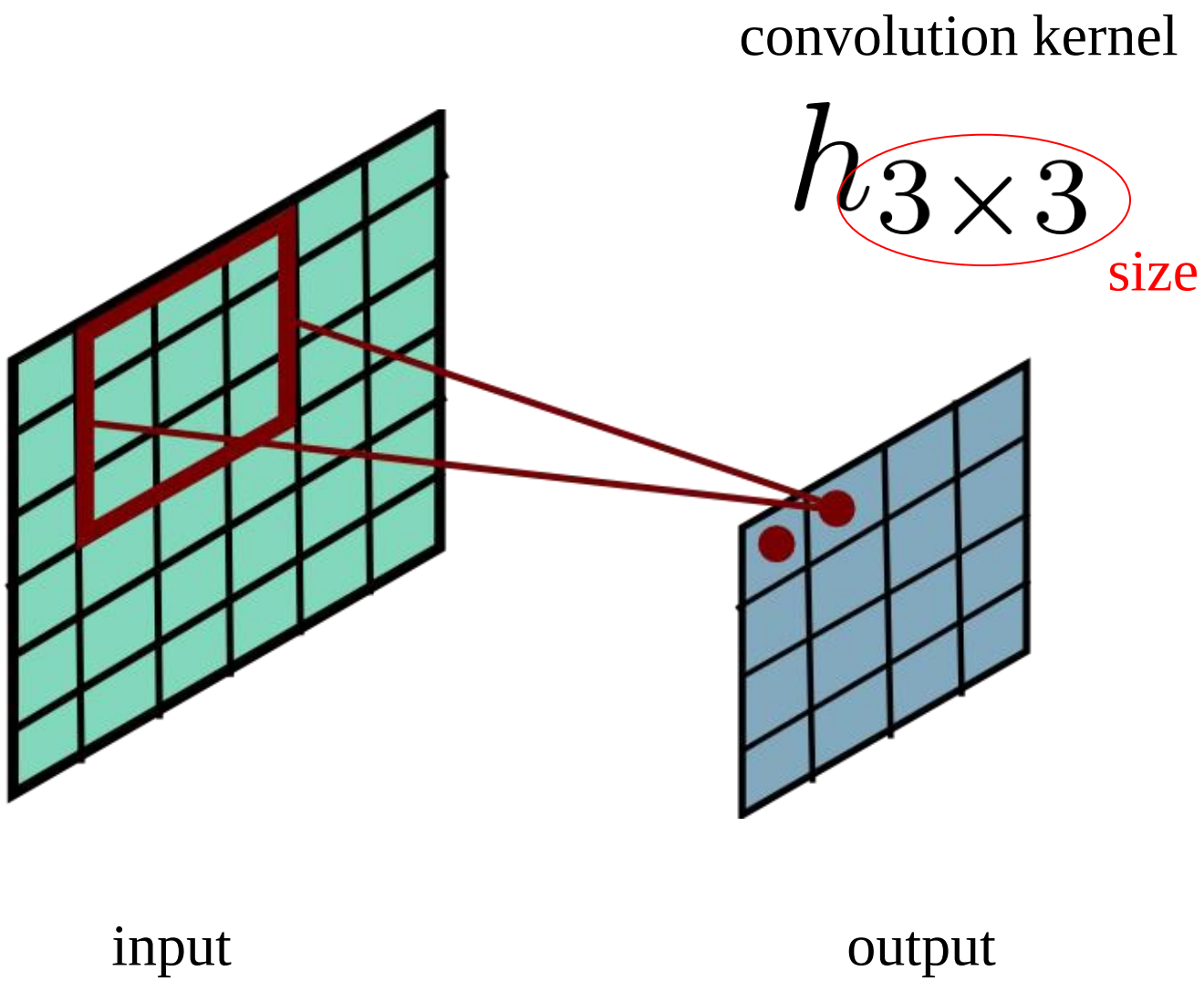
But here the filter is learned



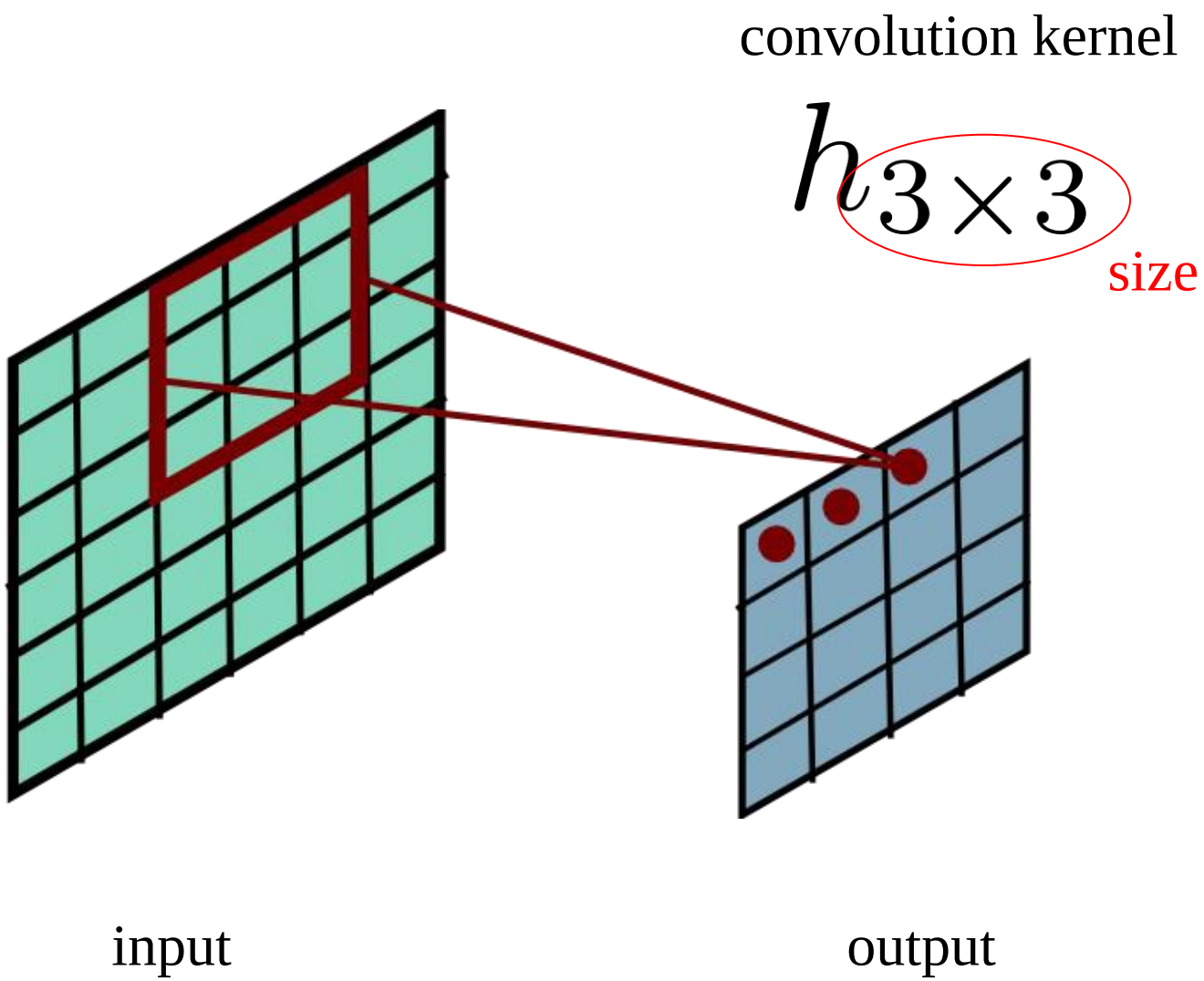
Convolutional Layer



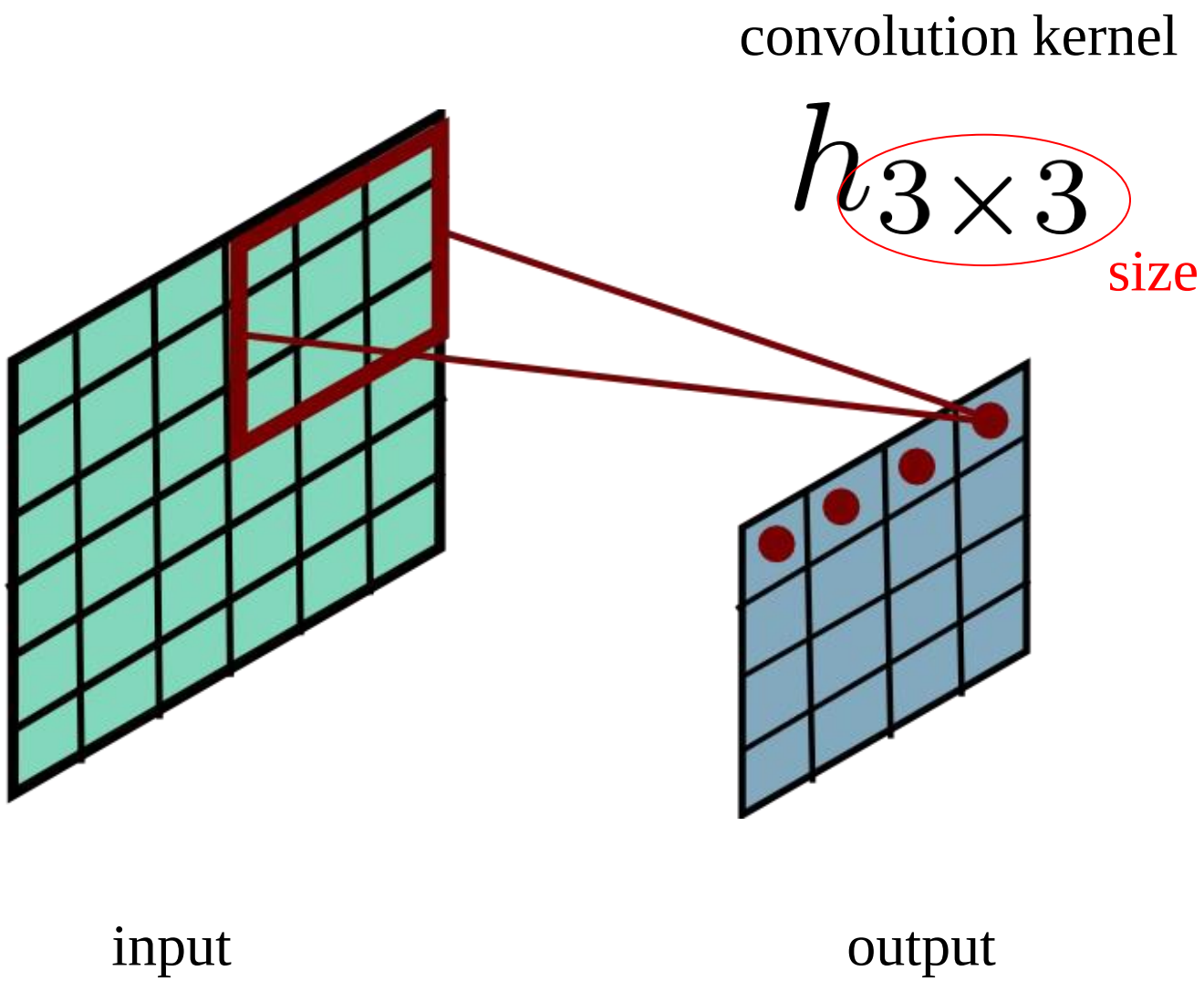
Convolutional Layer



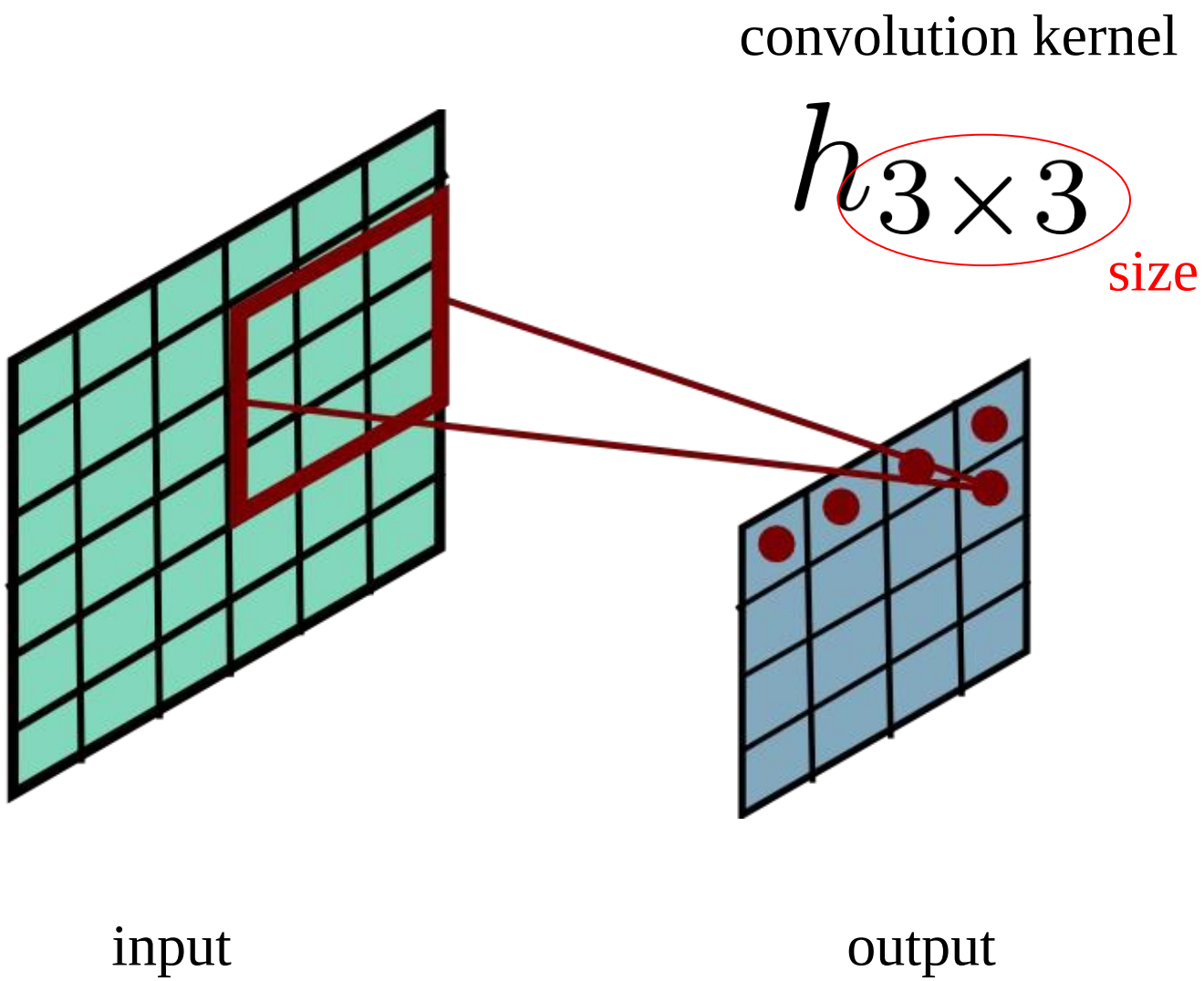
Convolutional Layer



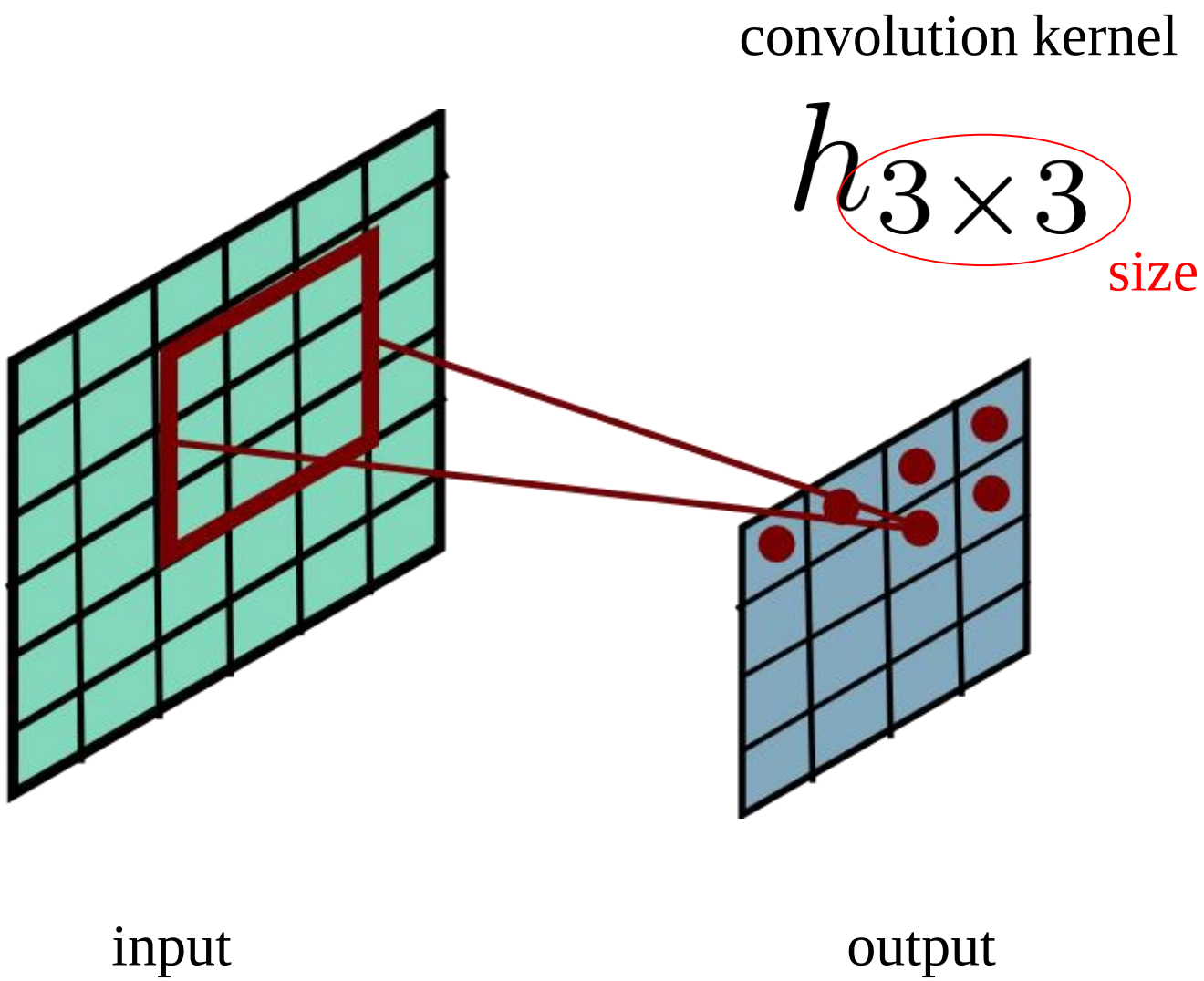
Convolutional Layer



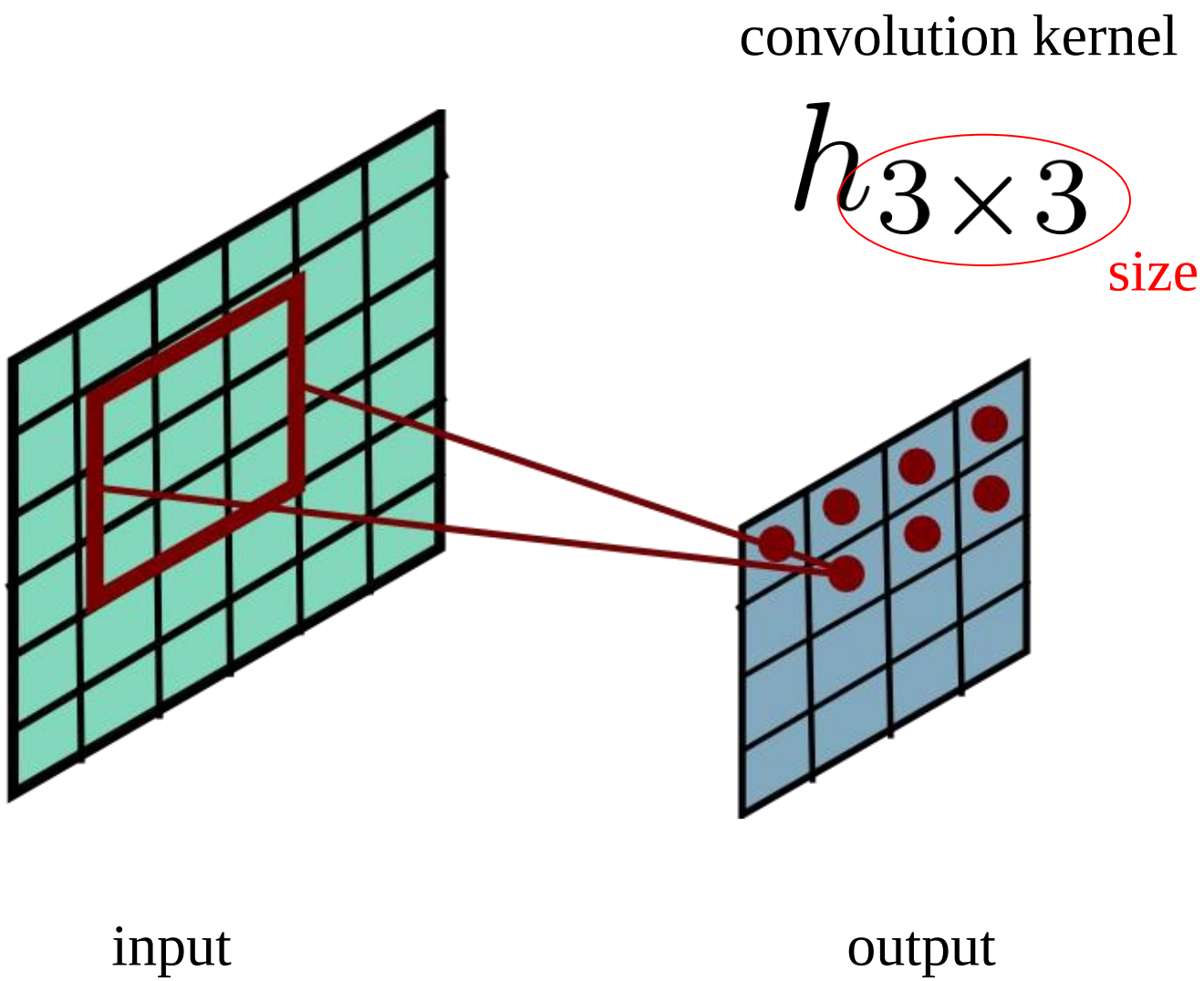
Convolutional Layer



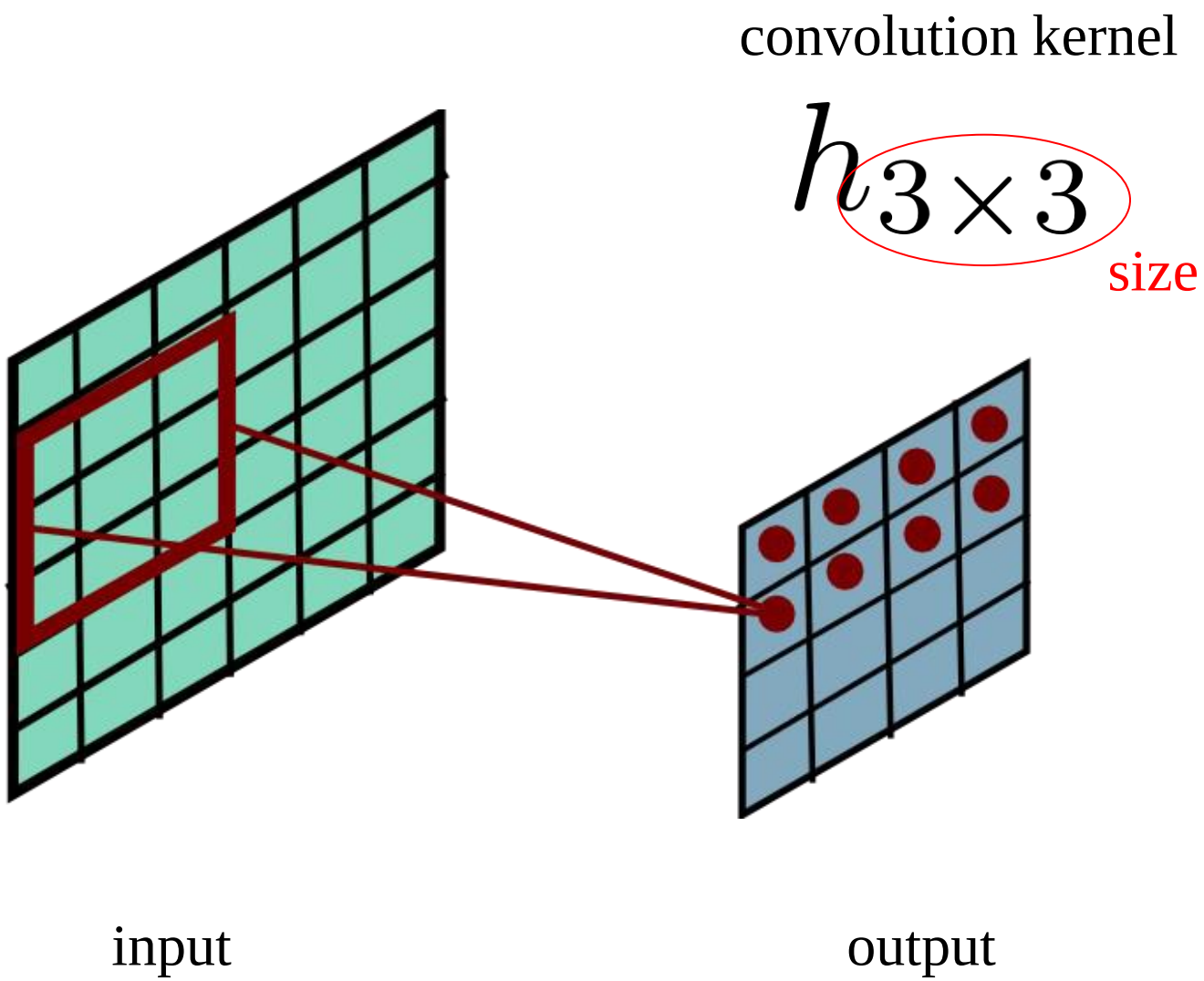
Convolutional Layer



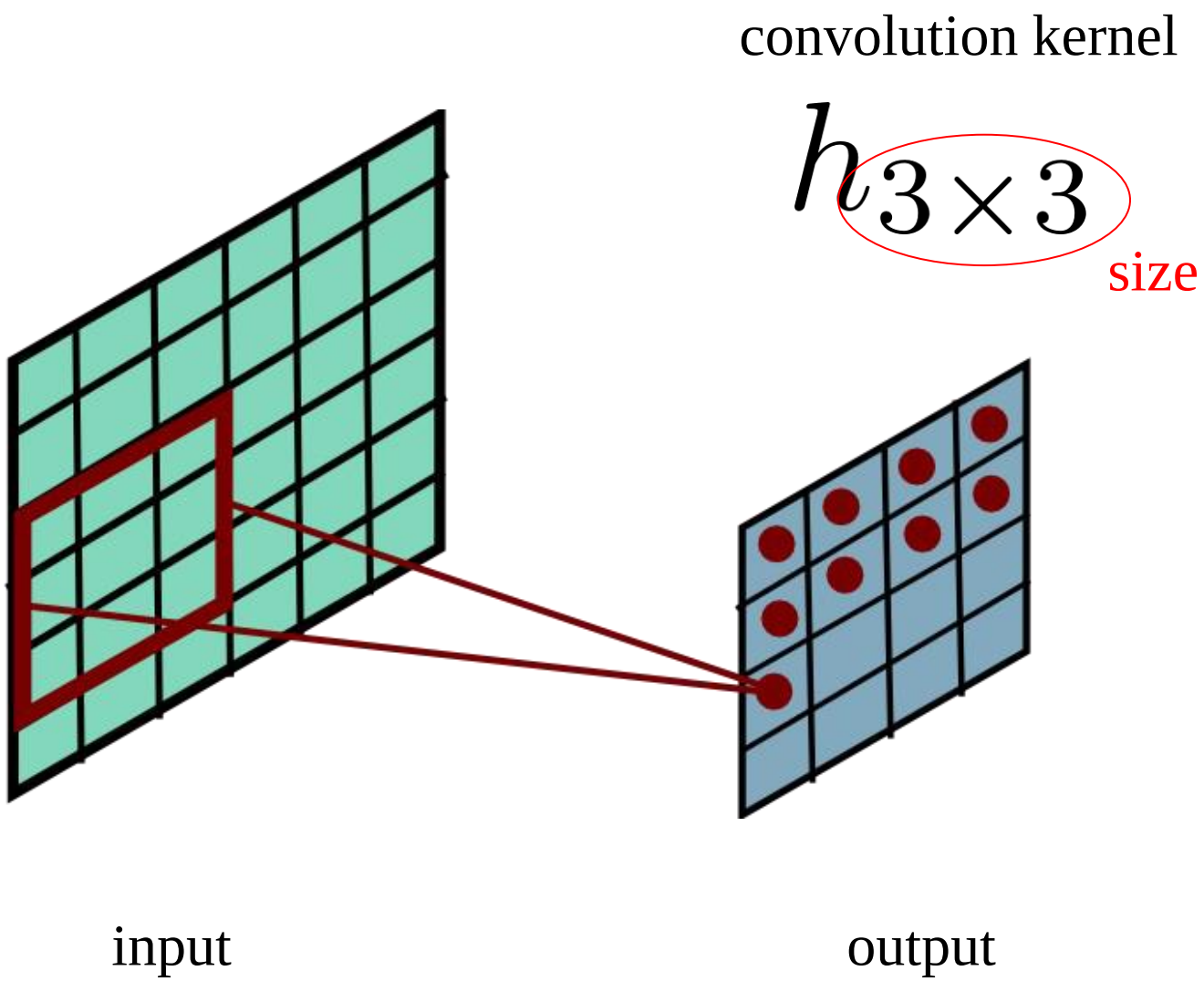
Convolutional Layer



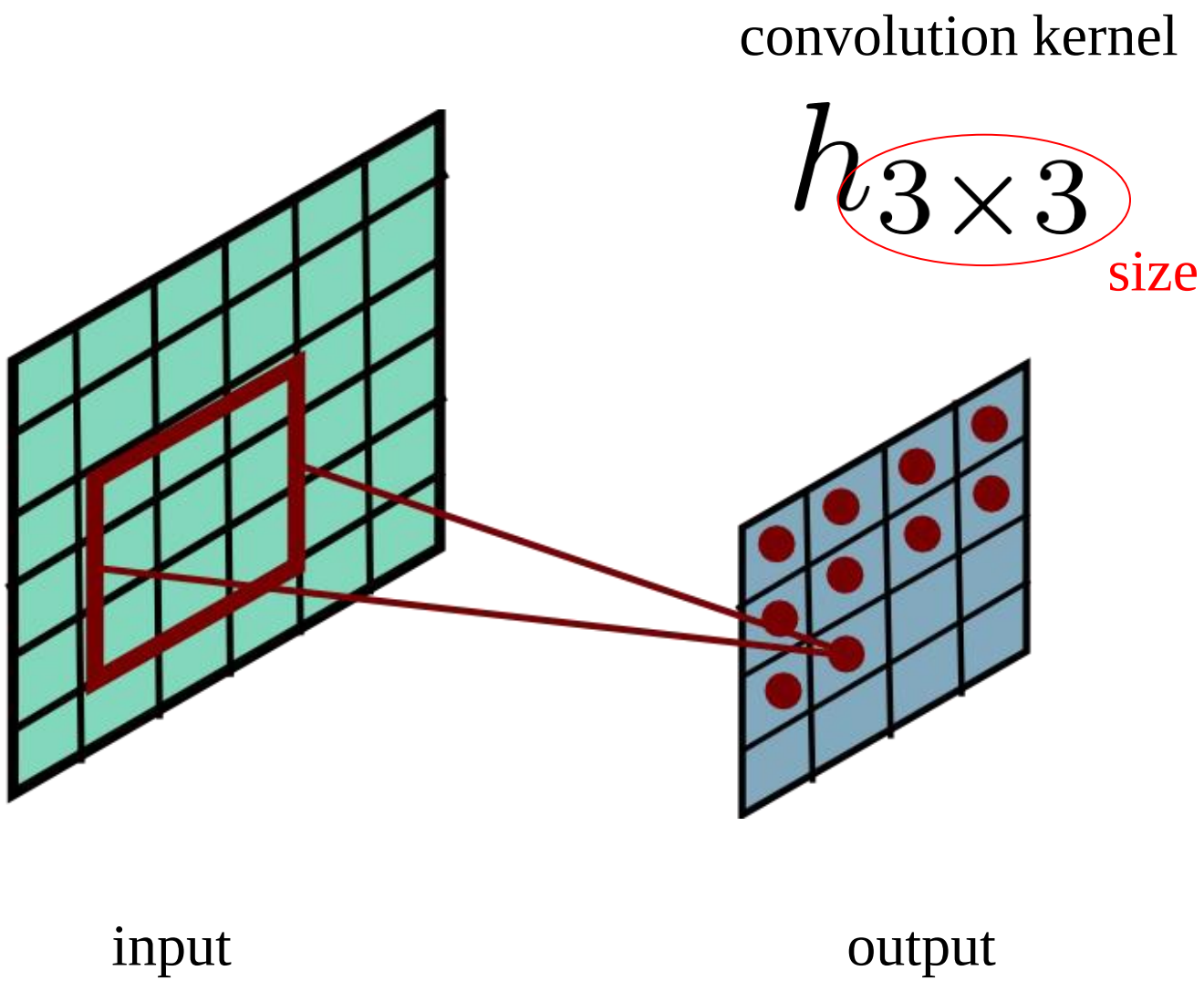
Convolutional Layer



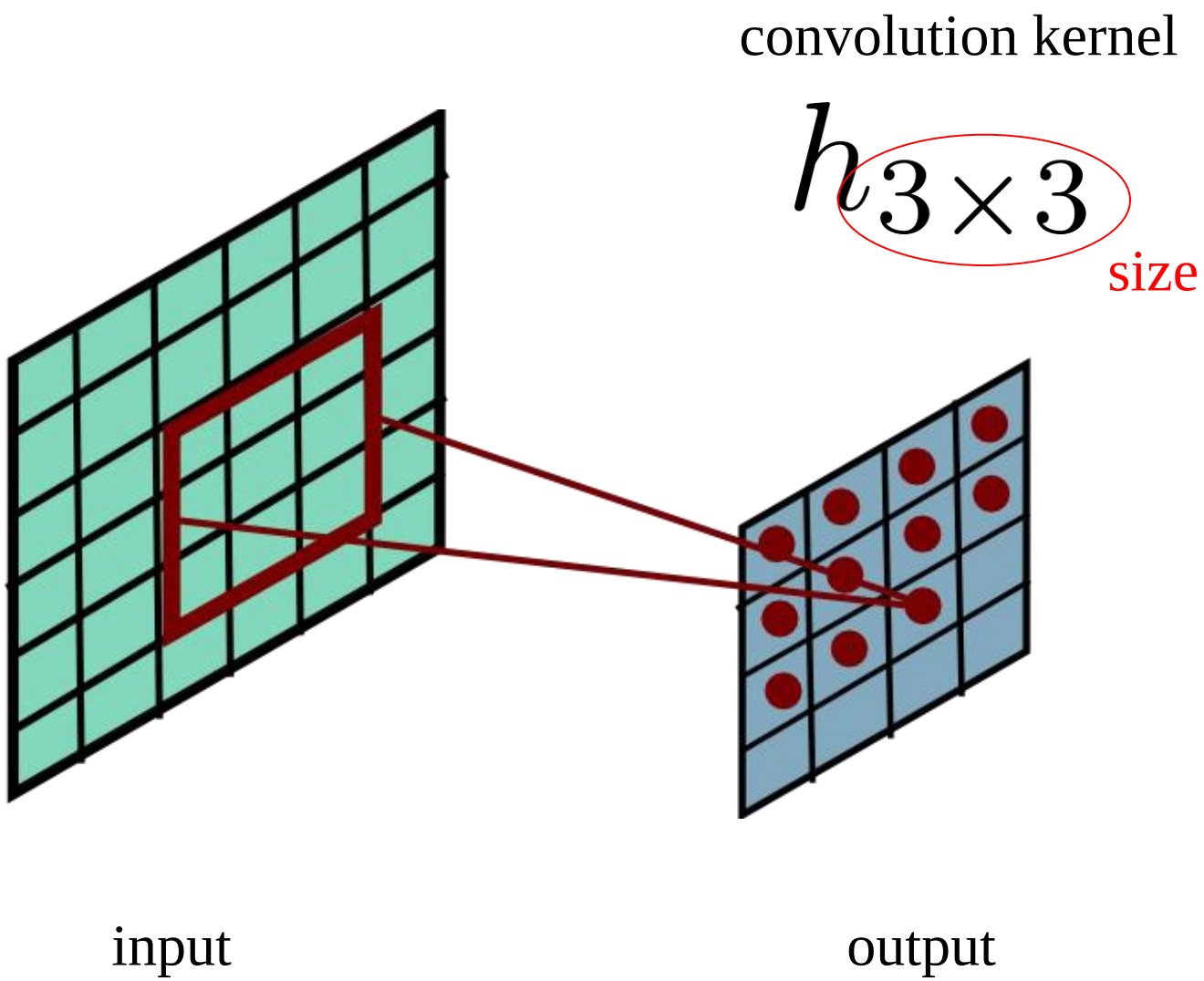
Convolutional Layer



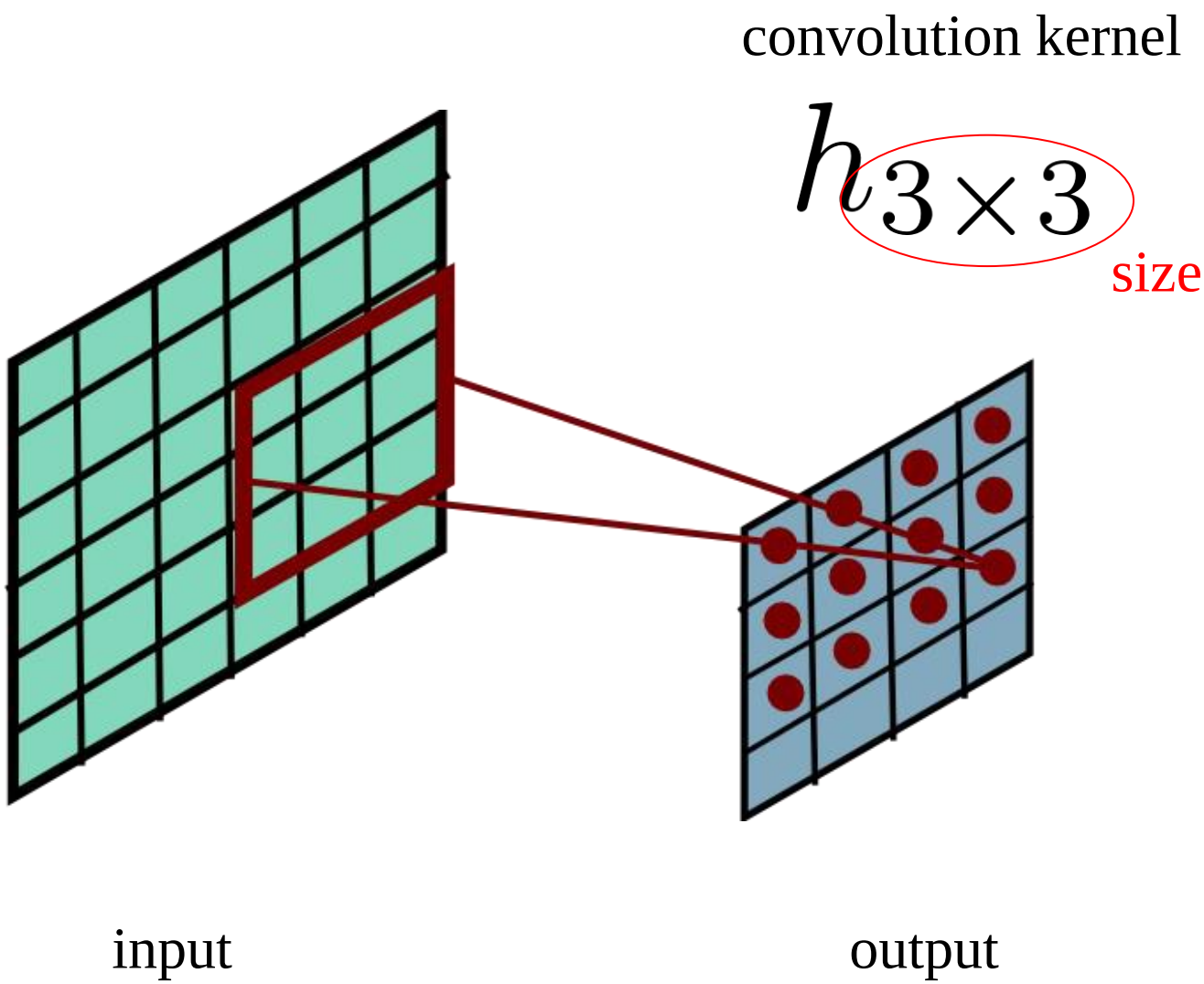
Convolutional Layer



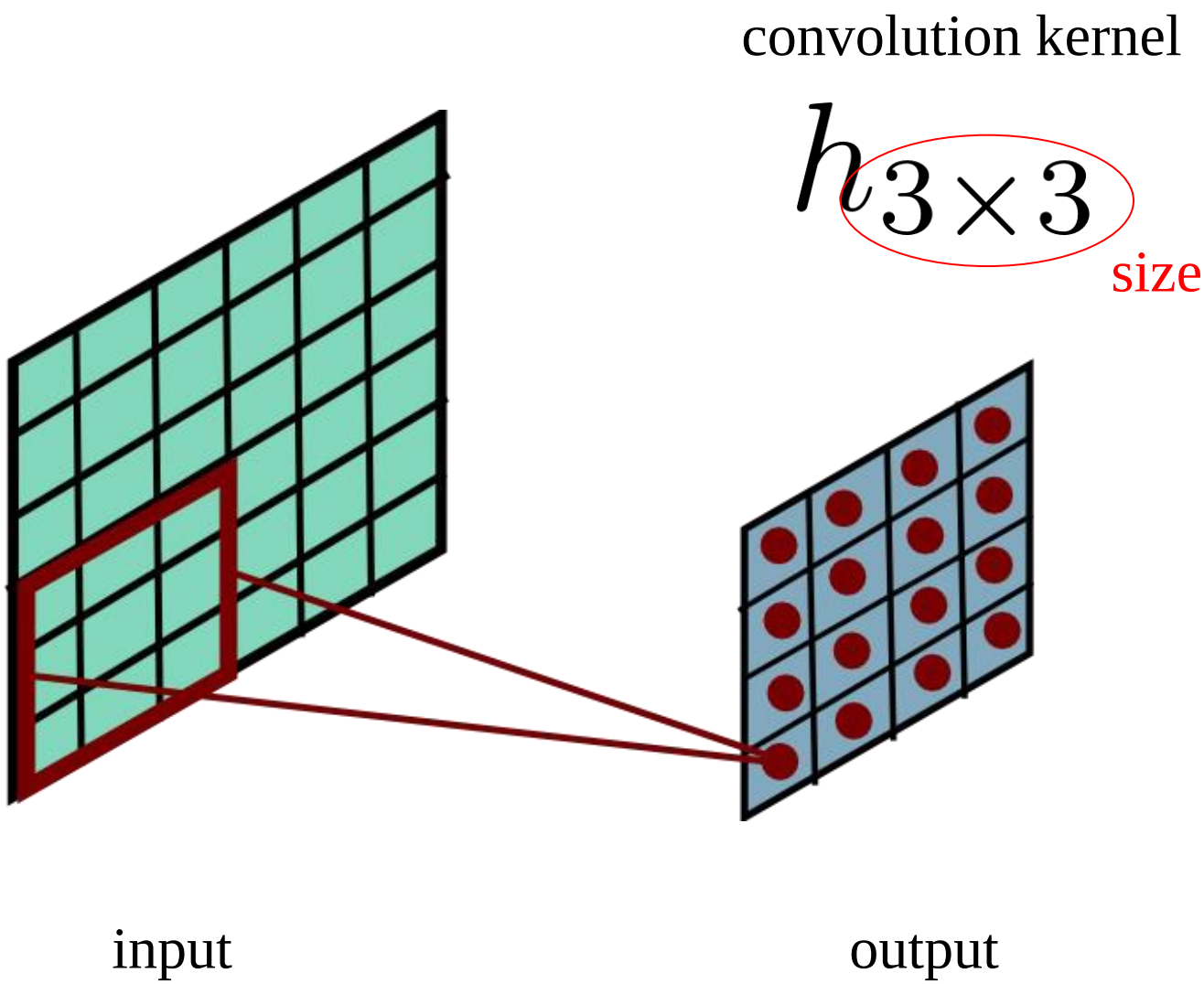
Convolutional Layer



Convolutional Layer

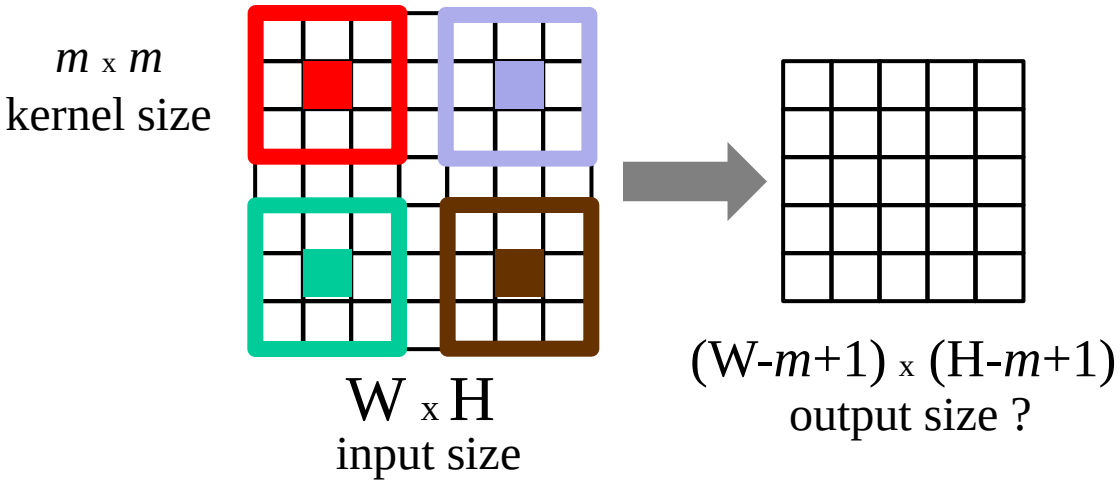


Convolutional Layer

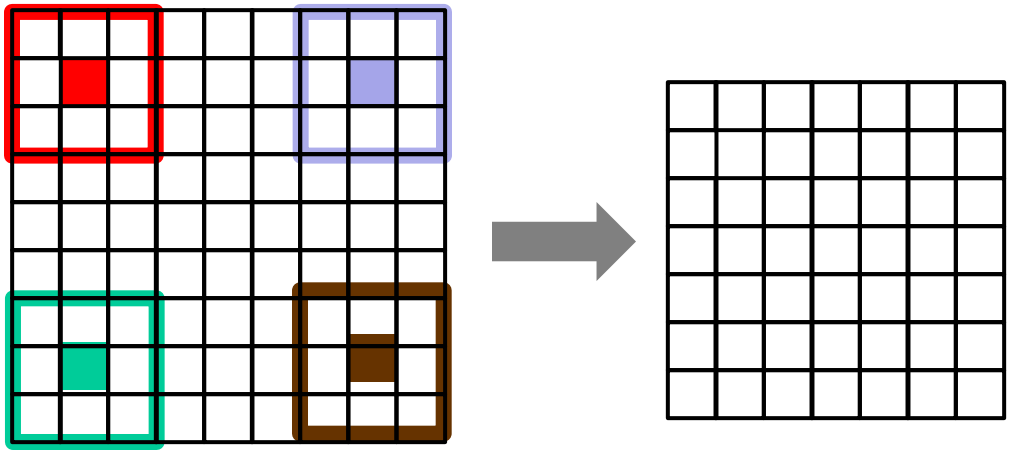


Convolutional Layer - Size Change

Output is usually slightly smaller because the borders of the image are left out



If want output to be the same size, zero-pad the input



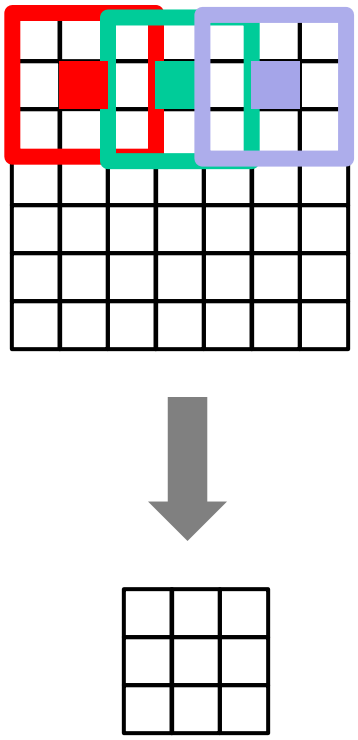
Convolutional Layer - Stride

Can apply convolution only to some pixels (say every second)

- output layer is smaller

Example

- stride = 2 means apply convolution every second pixel
- makes output image approximately **twice smaller** in each dimension
  - image not zero-padded in this example



strided convolution

minimizes information sharing/duplication

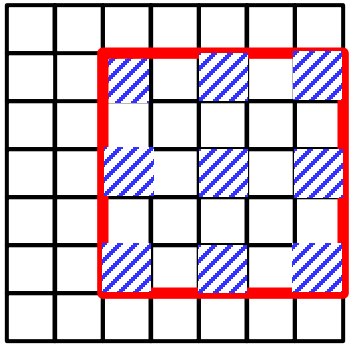
(overlap of kernel windows in the input)

but also reduces spatial resolution of the output

Convolutional Layer - Dilation

It maybe helpful to increase kernel size
to enlarge "*receptive field*"
for each element of the output

But larger kernels could be expensive…



Use only subset of points within the kernel's window

atrous convolution (Fr. *à trous* – hole)

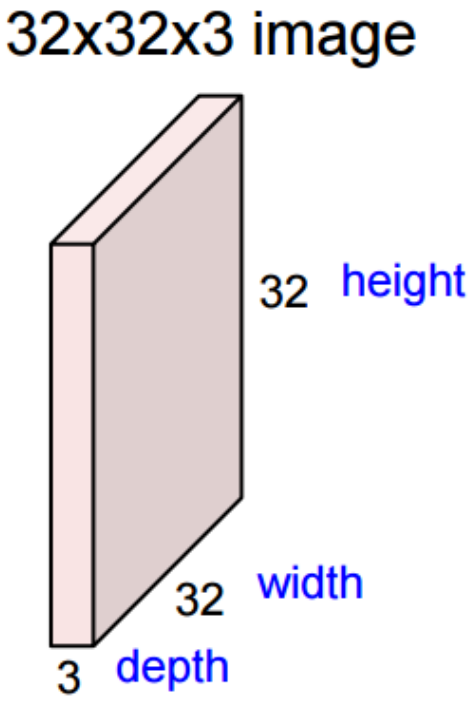
a.k.a. ***dilated convolution***

larger *receptive field* (5x5) for output elements
while effectively using smaller kernels (3x3)

It often makes sense to combine atrous convolution with stride

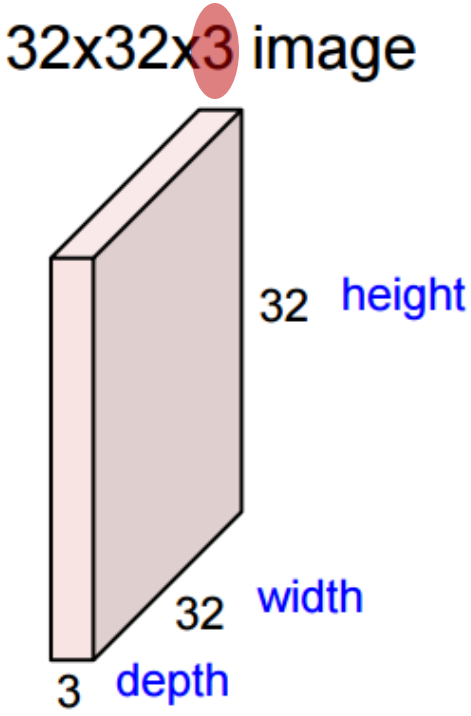
Convolutional Layer – Feature Depth

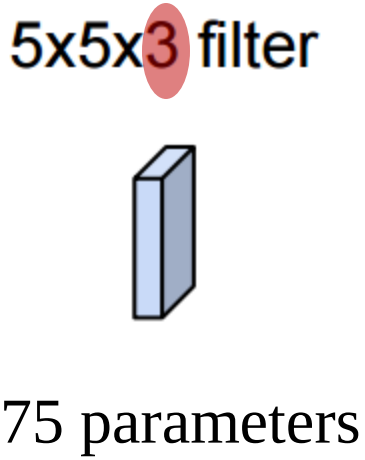
Input image is usually color, has 3 channels or depth 3



Convolutional Layer – Feature Depth

Convolve 3D image with 3D filter





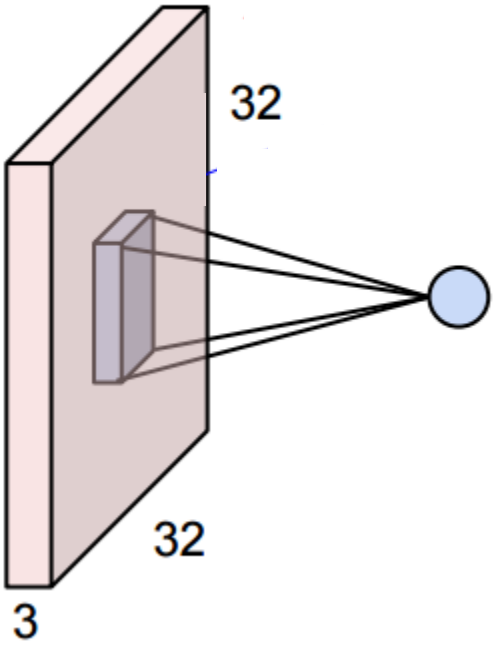
75 parameters

Convolutional Layer – Feature Depth

Each convolution step is a 75 dimensional dot product between the 5x5x3 filter and a piece of image of size 5x5x3

Can be expressed as $\mathbf{w}^t\mathbf{x}$, 75 parameters to learn ($\mathbf{w}$)

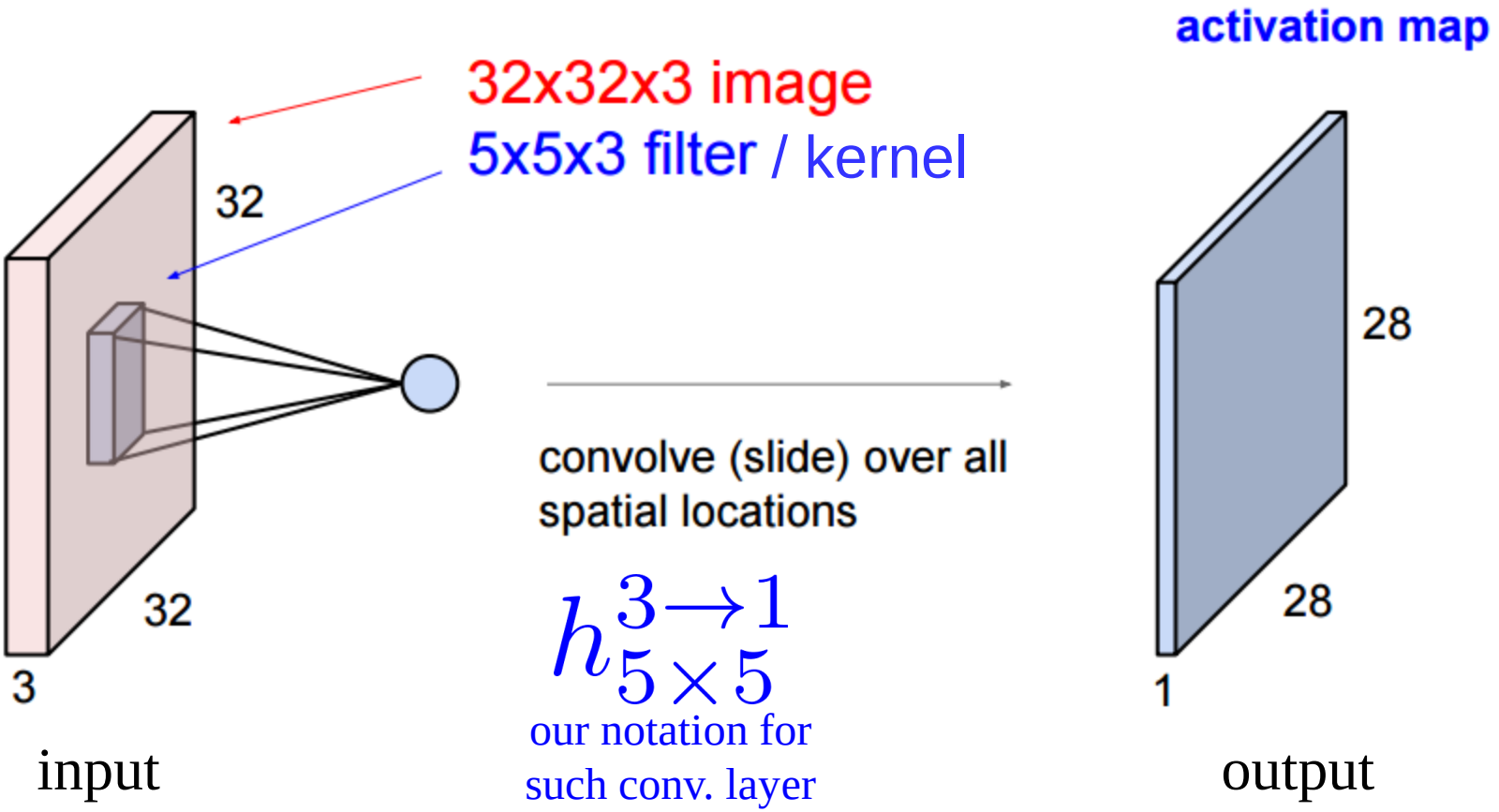
Can add bias $\mathbf{w}^t\mathbf{x} + b$, 76 parameters to learn ($\mathbf{w}$,b)



Convolutional Layer

Convolve 3D image with 3D filter

- result is a 28x28x1 activation map, no zero padding used



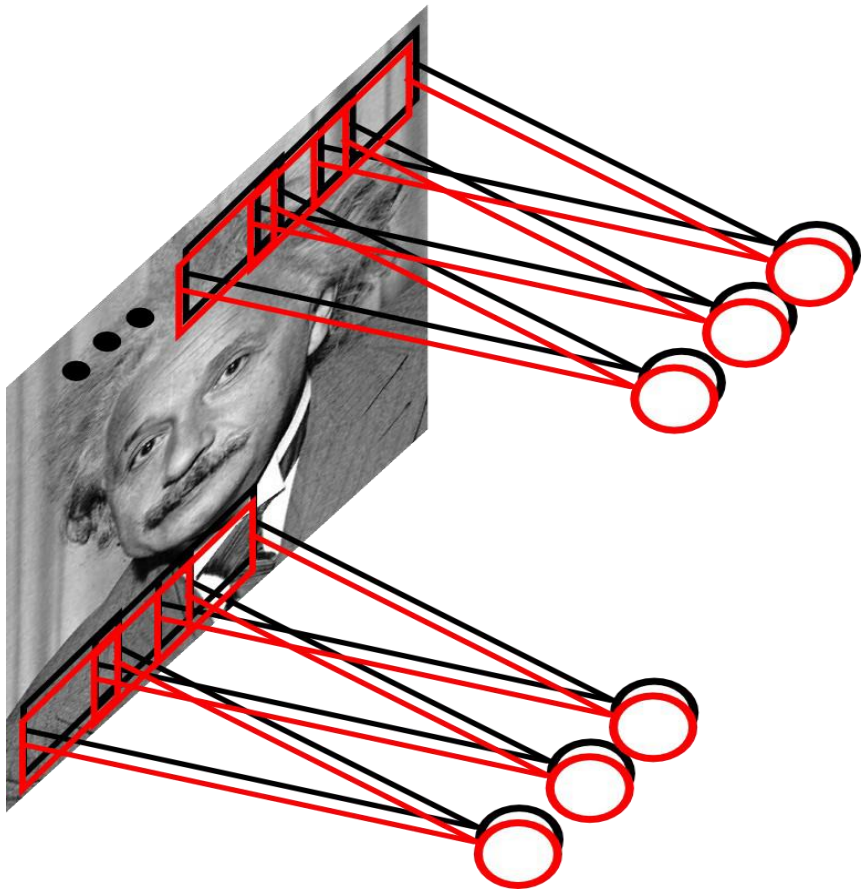
Convolutional Layer

One filter is responsible for one feature type

Learn multiple filters

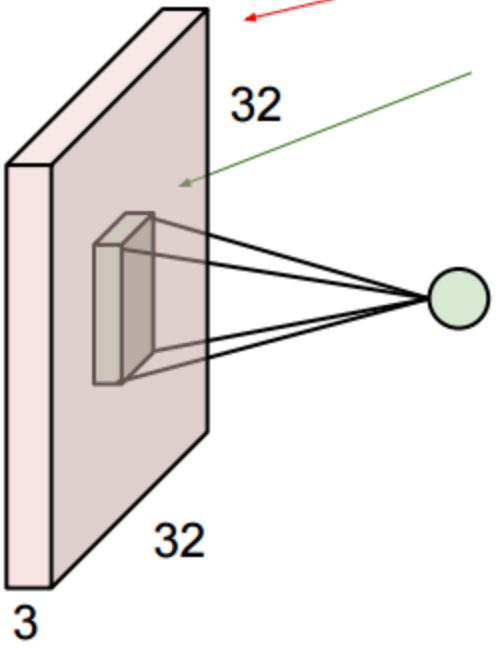
Example:

- 10x10 patch
- 100 filters
- only 10^4 parameters to learn



Convolutional Layer

Consider one extra filter

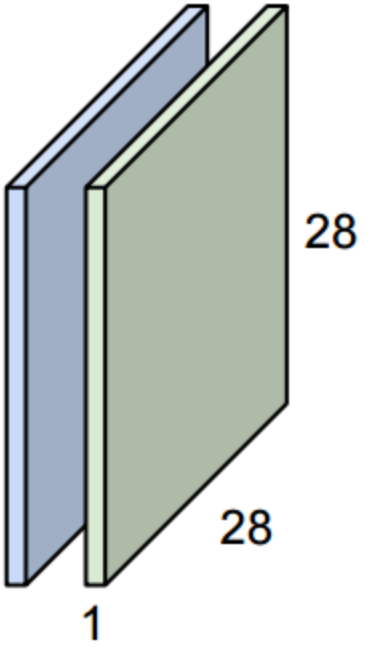


32x32x3 image

5x5x3 filter / kernel

Output from 2 kernels of shape 5x5x3

convolve (slide) over all spatial locations



$$h_{5\times 5}^{3\to 2}$$

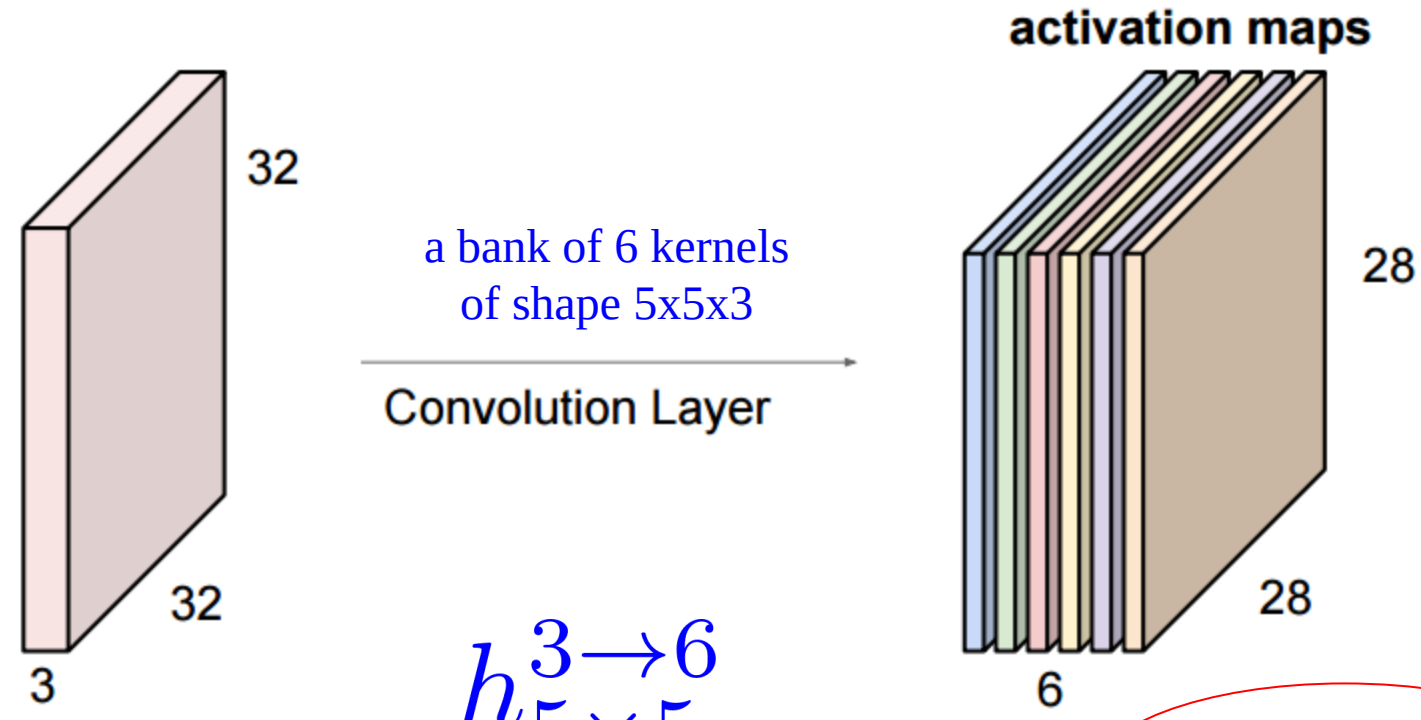
our notation for such conv. layer **with two filters**

input

output

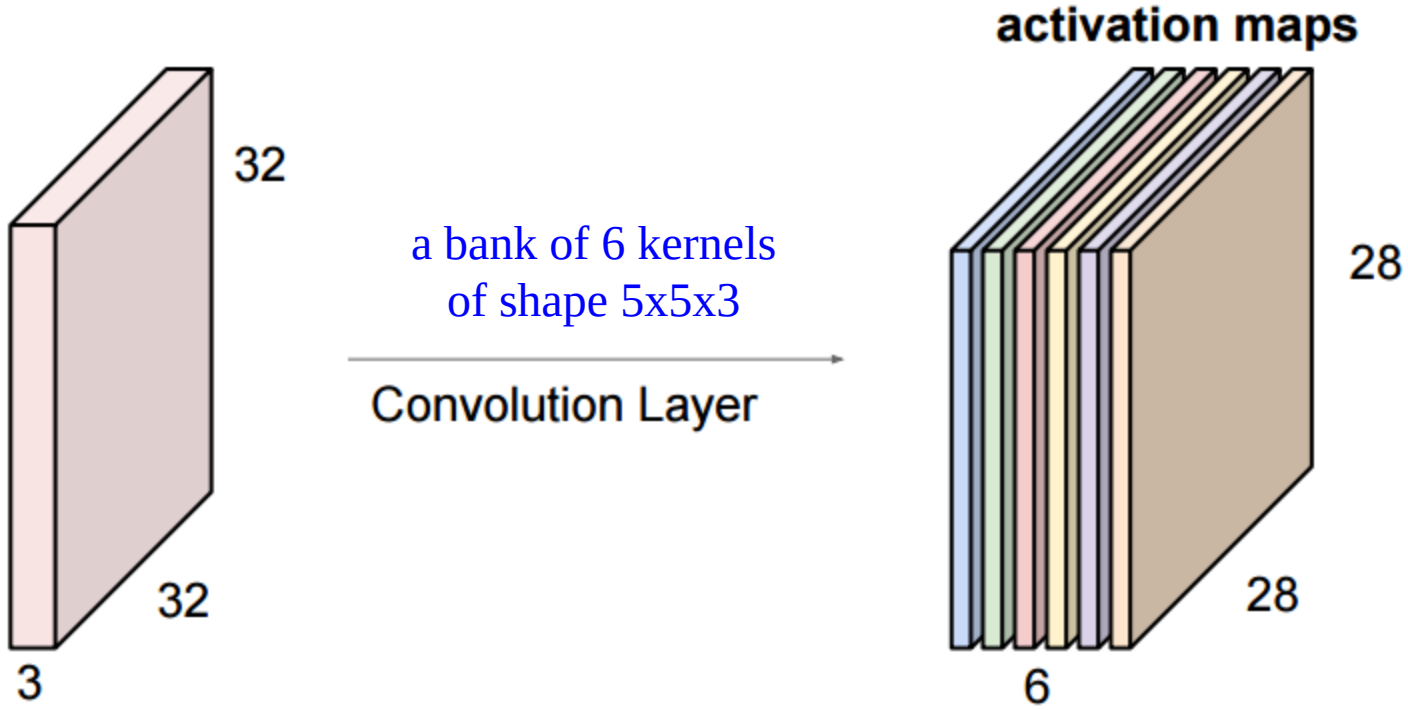
Convolutional Layer

- If have 6 filters (each of size 5x5x3) get 6 activation maps, 28x28 each
- Stack them to get new 28x28x6 “image”



Convolutional Layer

Apply activation function (say ReLu) to the activation map

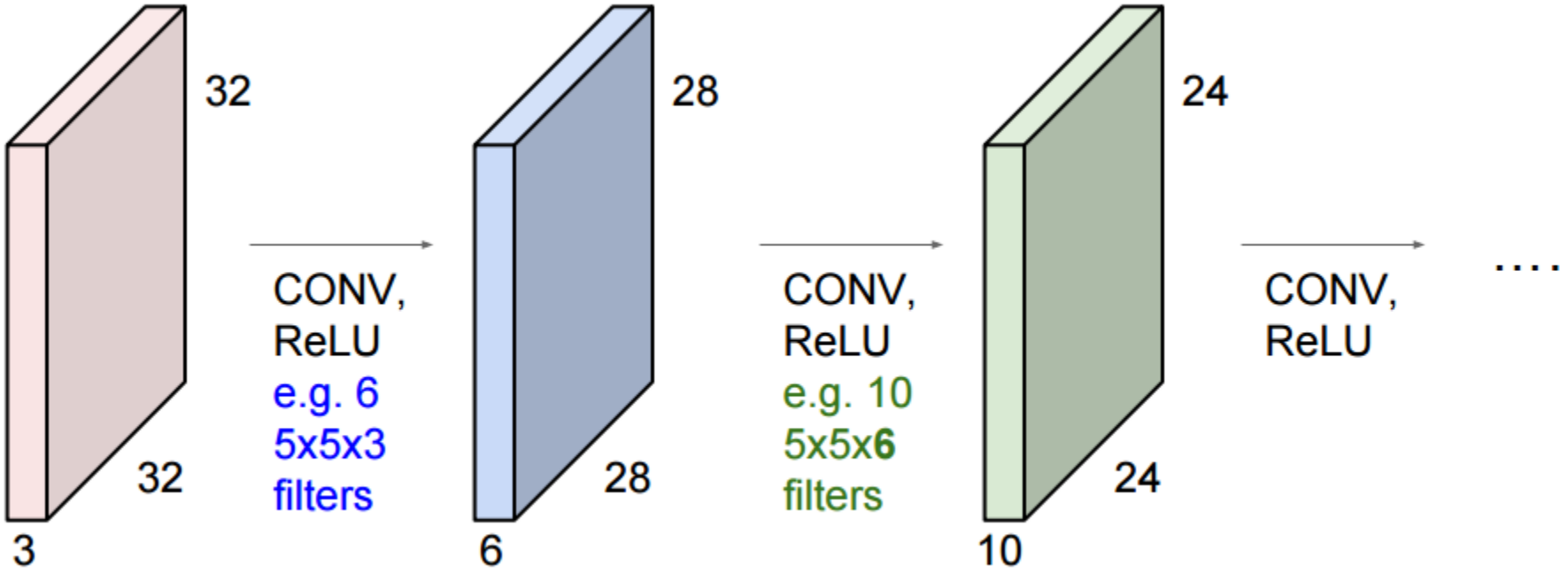


X

$$ReLU(h_{5\times5}^{3\to6} * X)$$

Several Convolution Layers

Construct a sequence of convolution layers interspersed with activation functions



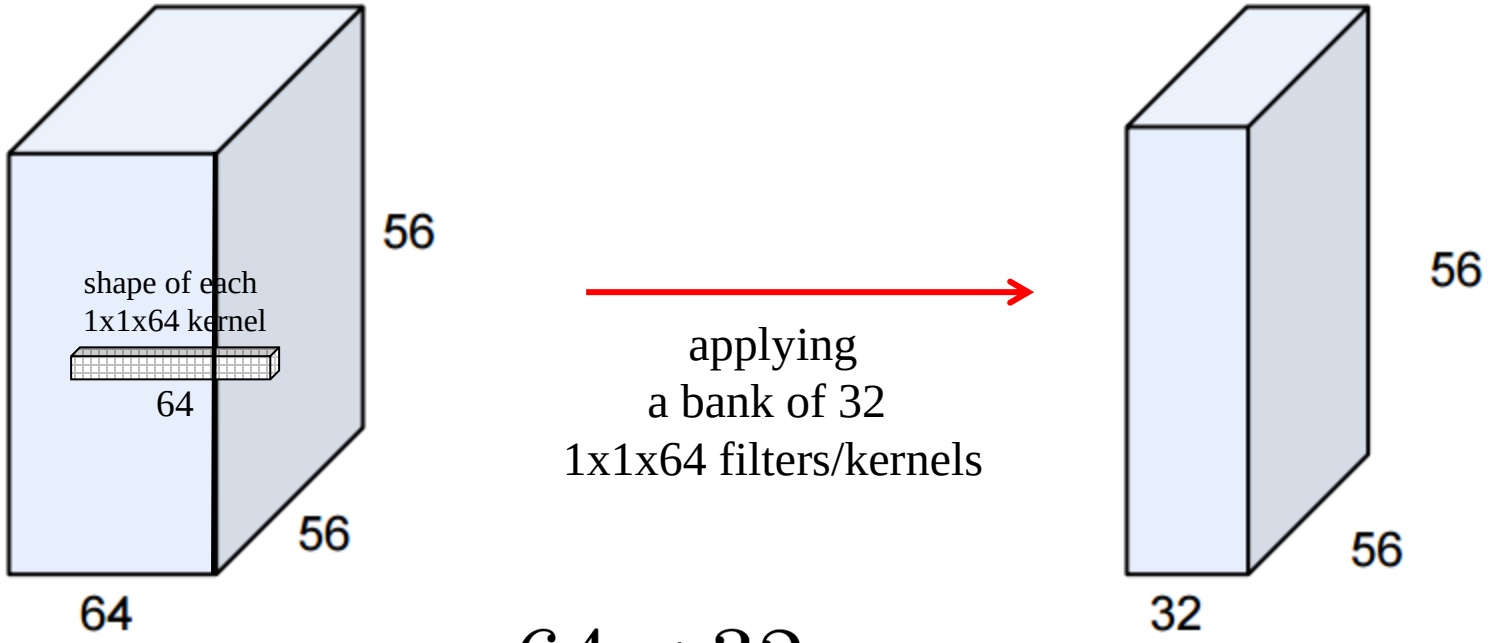
$$ReLU(h_{5\times5}^{3\to6} * X) \qquad ReLU(h_{5\times5}^{6\to10} * X)$$

Convolutional Layer

1x1 convolutions make perfect sense

Example

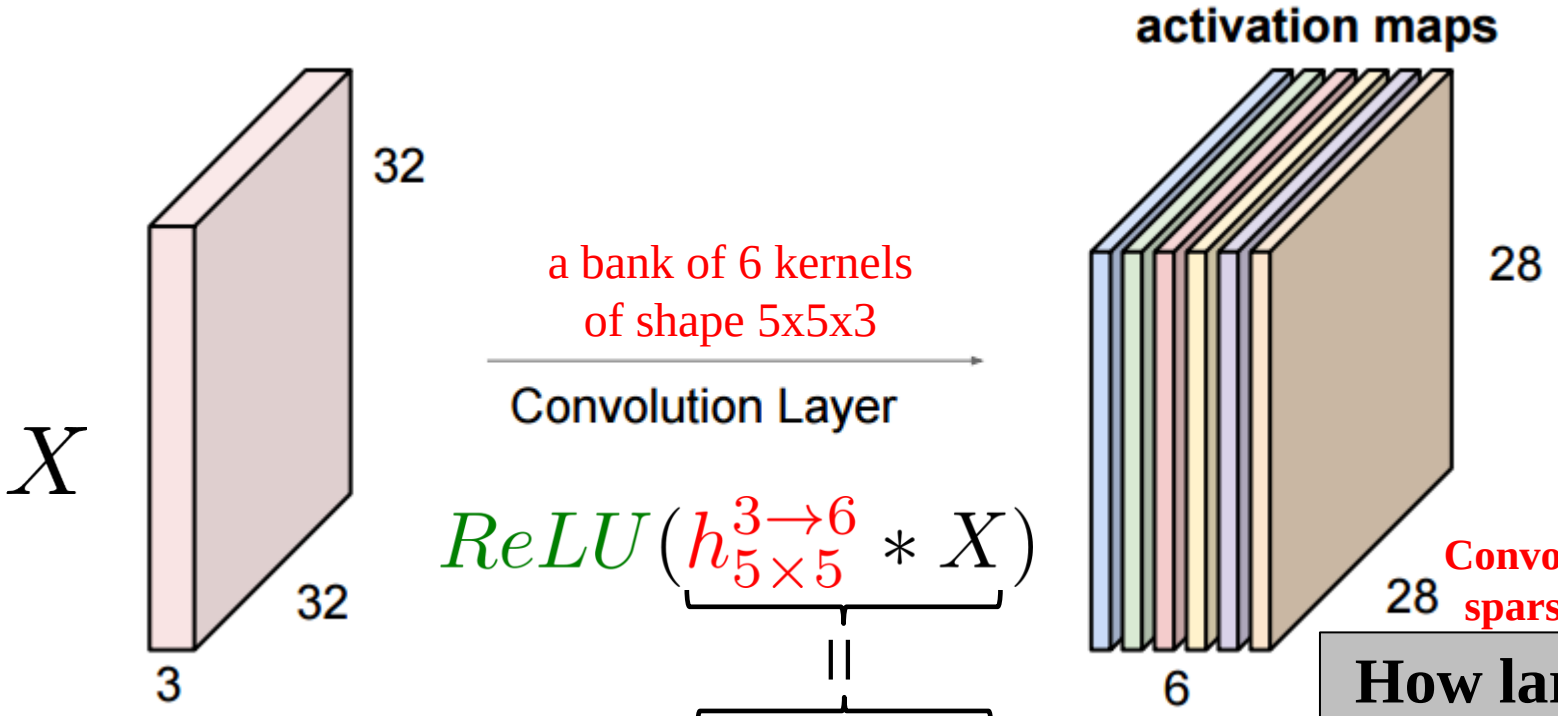
- Input image of size 56x56x64
- Convolve with 32 filters, each of size 1x1x64



$$h_{1\times1}^{64\rightarrow32} * X$$

Convolutional Layer vs Fully Connected

For example, assume that we applied ReLU to the activation maps



Convolution is highly sparse special case.

How large is *W* ?

m = 32x32x3

n = 28x28x6

1.4m parameters

vs. 450 parameters

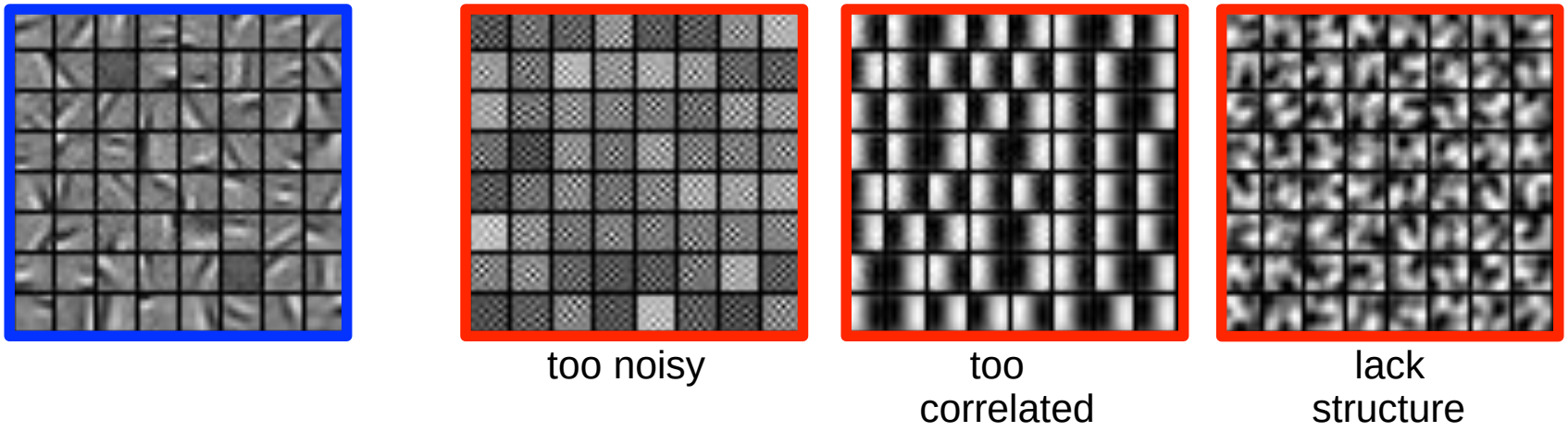
for 6 kernels 5x5x3

The convolution is a linear transform.

So, we can equivalently express it via matrix multiplication for some matrix *W*.

Check Learned Convolutions

- Good training: learned filters exhibit structure and are uncorrelated



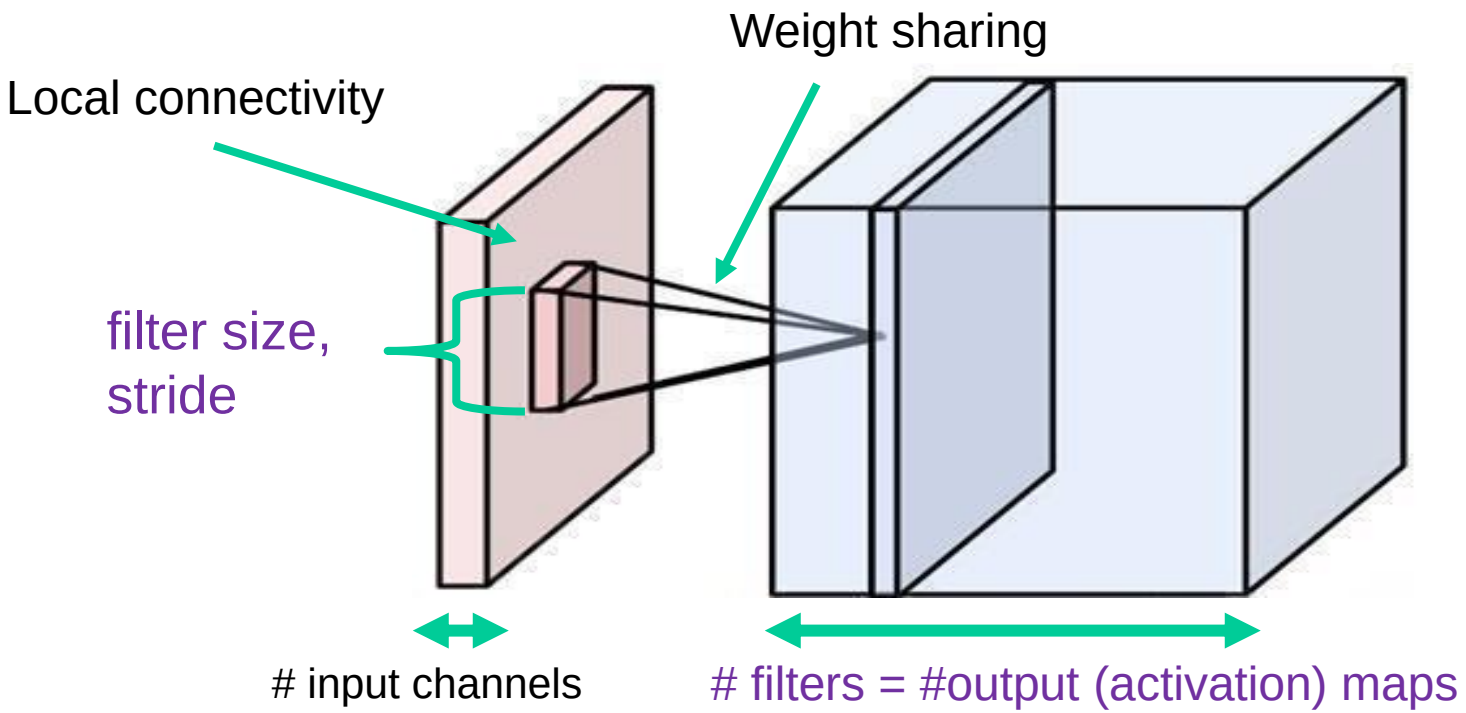
Convolutional Layer Summary

Local connectivity

Weight sharing

Handling multiple input/output channels

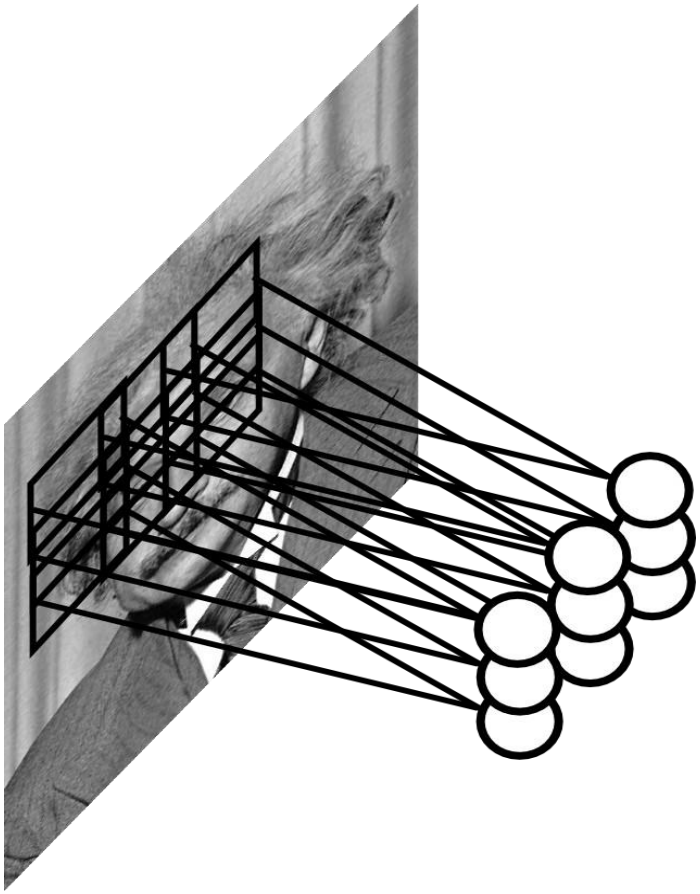
Retains location associations



Pooling Layer

Say a filter is an *eye* detector

Want detection to be robust to precise *eye* location

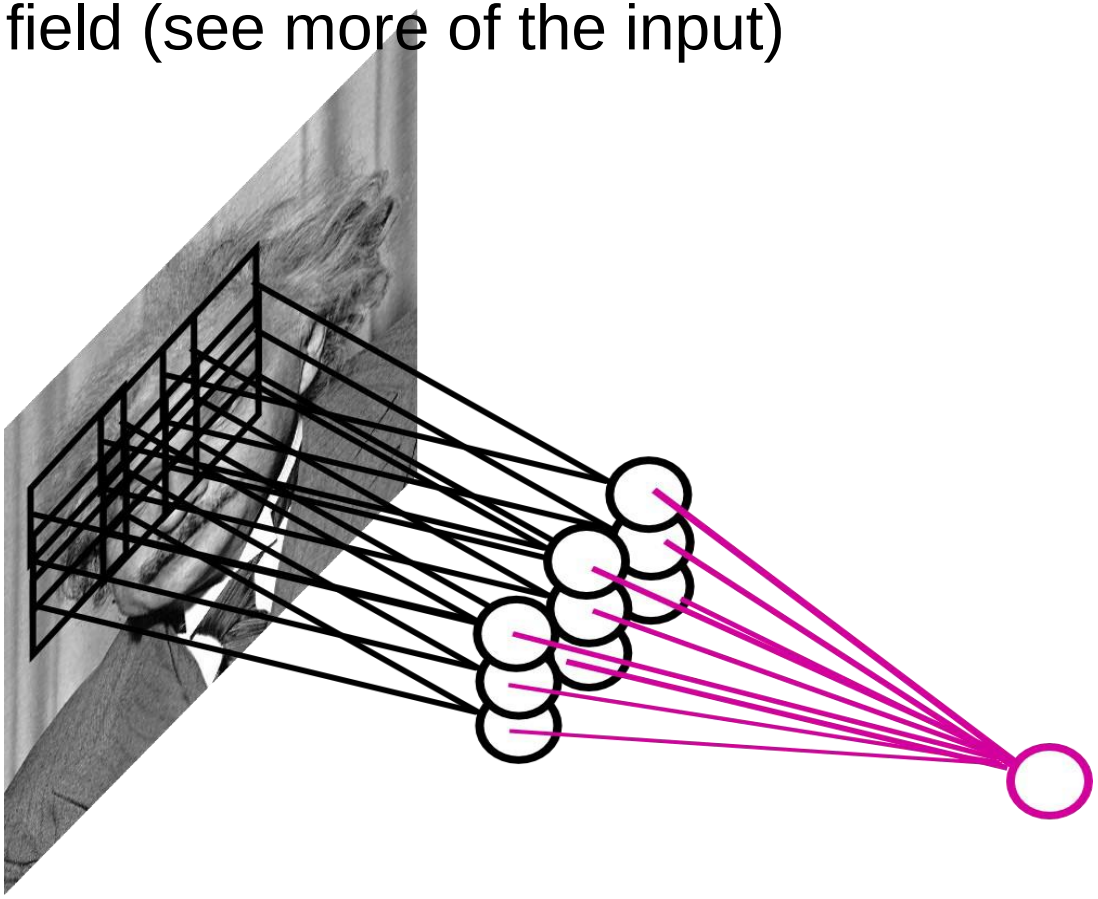


Pooling Layer

Pool responses at different locations

- by taking max, average, etc.
- robustness to exact spatial location
- also larger receptive field (see more of the input)

- Usually pooling applied with stride > 1
- This reduces resolution of output map
- But we already lost resolution (precision) by pooling



Pooling Layer: Max Pooling Example

Single depth slice

1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

max pool with 2x2 filters
and stride 2

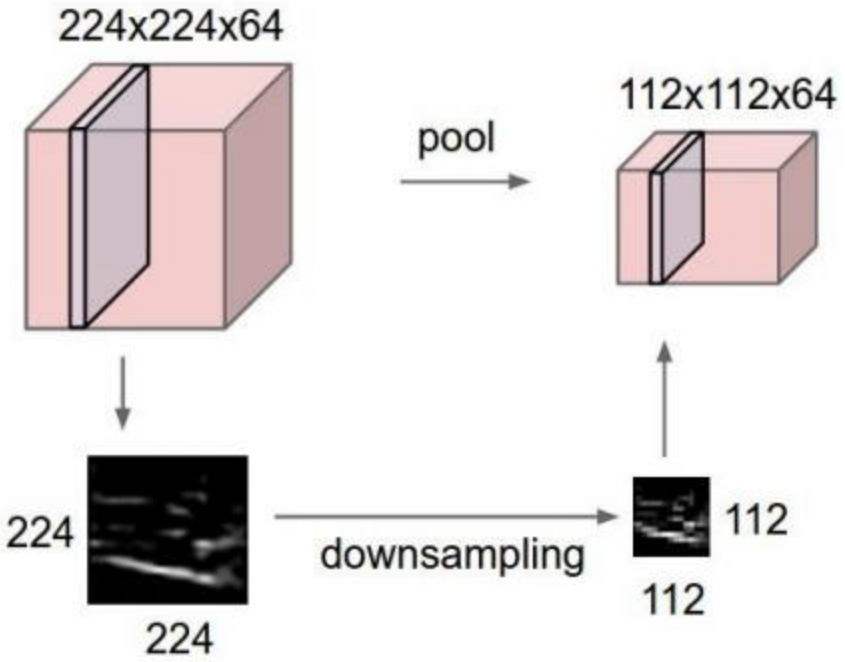
6	8
3	4

$$\text{Pool}_{2\times 2}^{st2}(X)$$

our notation for
2 by 2 pooling layer
with stride 2

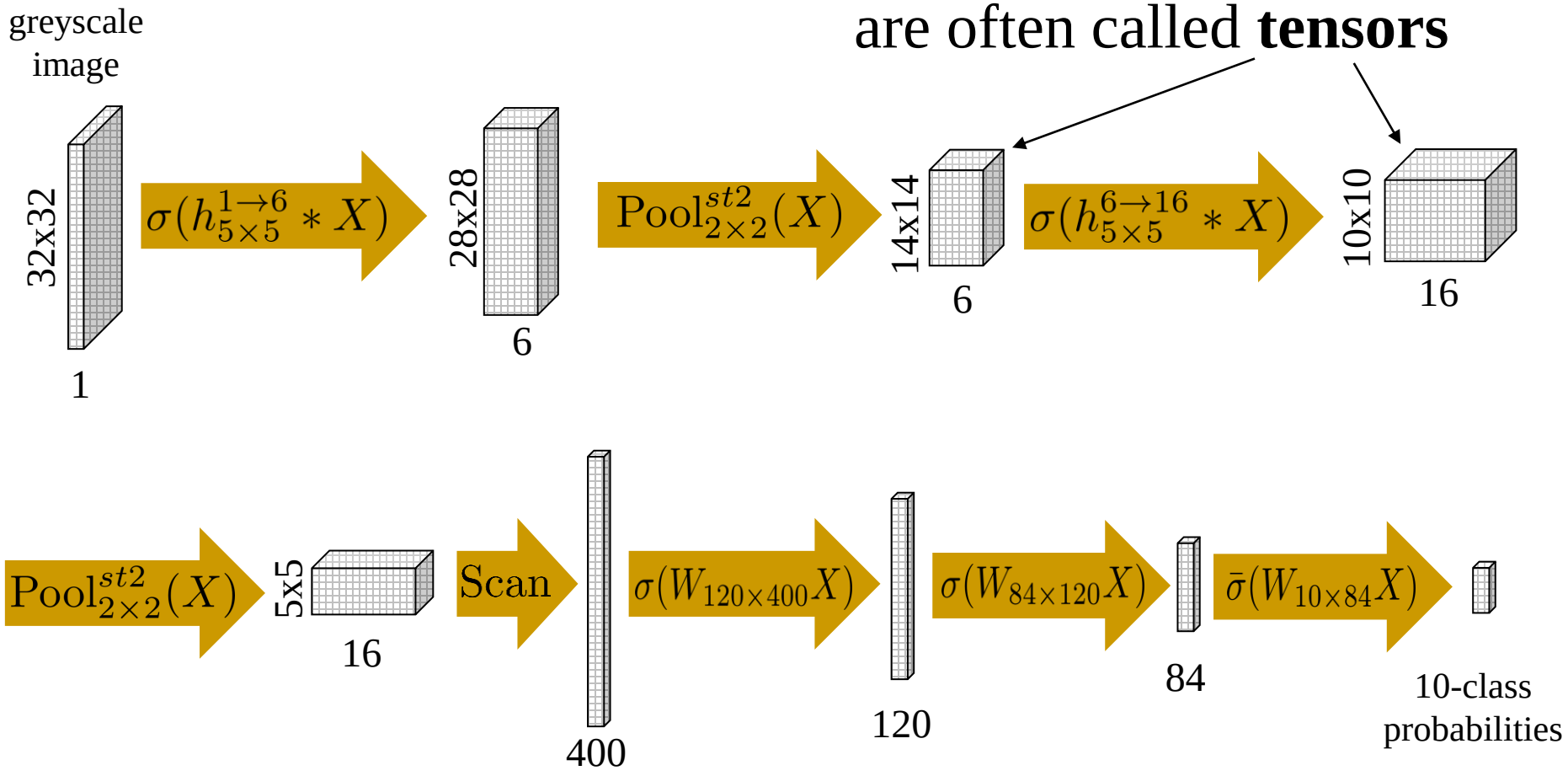
Pooling Layer

Pooling usually applied to each activation map separately



Basic CNN example (à la ***LeNet*** -1998)

NOTE: such multi-dimensional arrays are often called **tensors**



First CNN architectures for classification

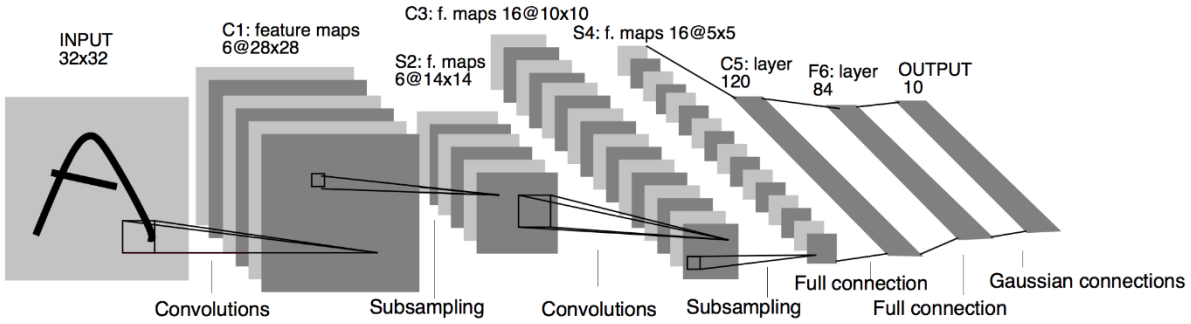
- **first CNNs** (1982-89)
  (a.k.a. *convNets*)

Neocognitron: A new algorithm for pattern recognition tolerant of deformations and shifts in position
K. Fukushima, S. Miyake - Pattern Recognition 1982

Handwritten digit recognition with a back-propagation network
Y. LeCun et al - NIPS 1989

- **LeNet** (1998)

Handwritten digit recognition with a back-propagation network
Y. LeCun, L. Bottou, Y. Bengio, P. Haffner - Proc.of IEEE 1998



First CNN architectures for classification

- **first CNNs** (1982-89)
  (a.k.a. *convNets*)

Neocognitron: A new algorithm for pattern recognition tolerant of deformations and shifts in position
K. Fukushima, S. Miyake - Pattern Recognition 1982

Handwritten digit recognition with a back-propagation network
Y. LeCun et al - NIPS 1989

- **LeNet** (1998)

https://youtu.be/FwFduRA_L6Q

Deep CNN architectures for classification

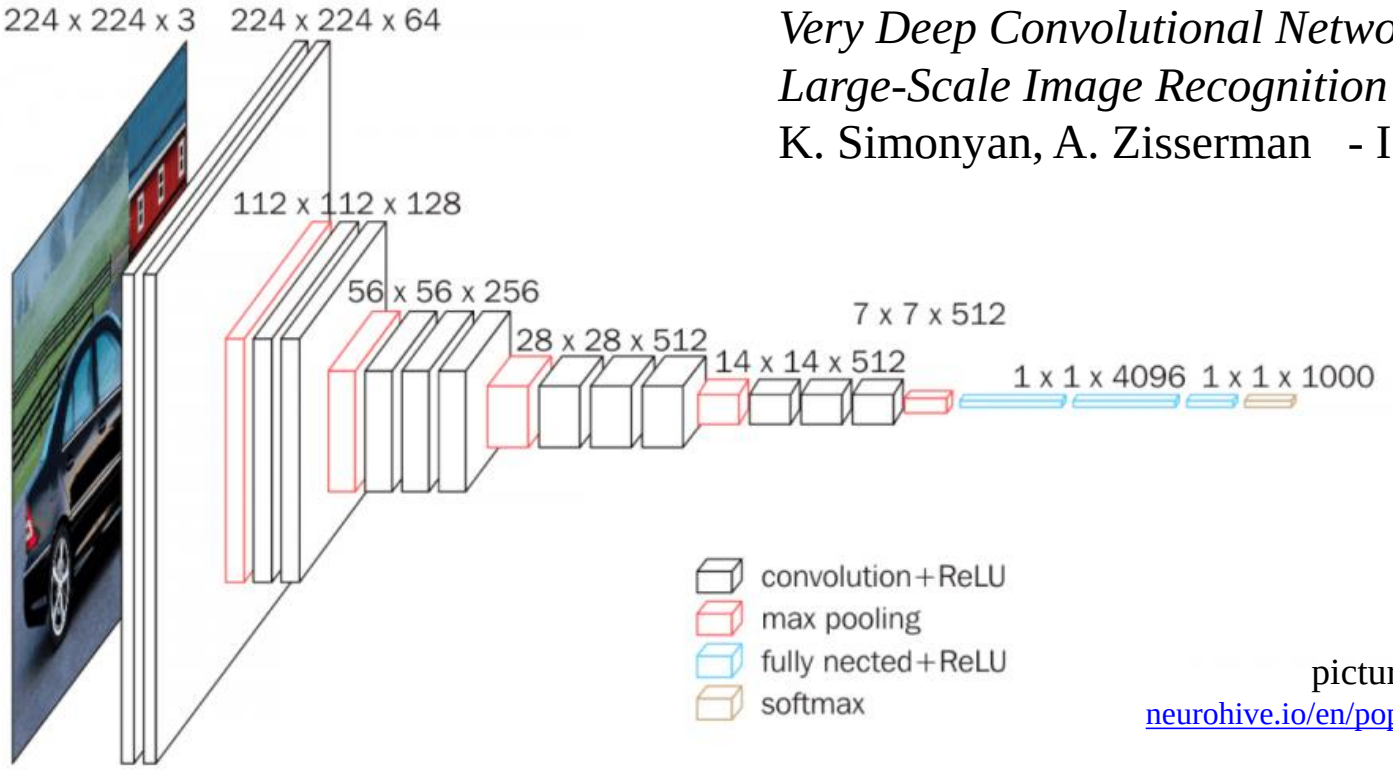
- **AlexNet** (2012) *ImageNet classification with deep convolutional neural networks*
  Alex Krizhevsky, Ilya Sutskever, Geoffrey Hinton - NIPS 2012.

- **VGG** (2014) *Very Deep Convolutional Networks for Large-Scale Image Recognition*
  K. Simonyan, A. Zisserman - ICLR 2015

  http://www.robots.ox.ac.uk/~vgg/practicals/cnn/index.html

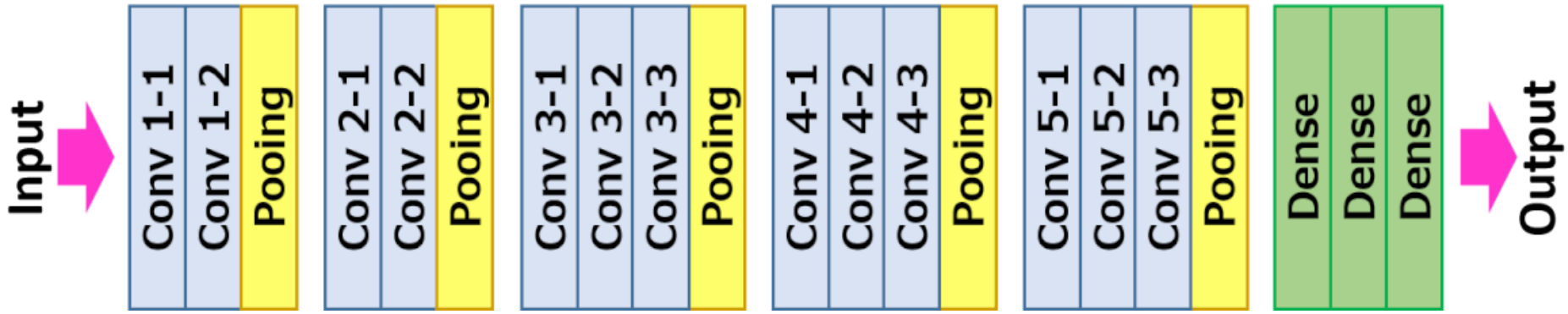
- **ResNet** (2016) *Deep residual learning for image recognition*
  K. He, X. Zhang, S. Ren, J. Sun. - CVPR 2016

VGG -16



Very Deep Convolutional Networks for Large-Scale Image Recognition
K. Simonyan, A. Zisserman - ICLR 2015

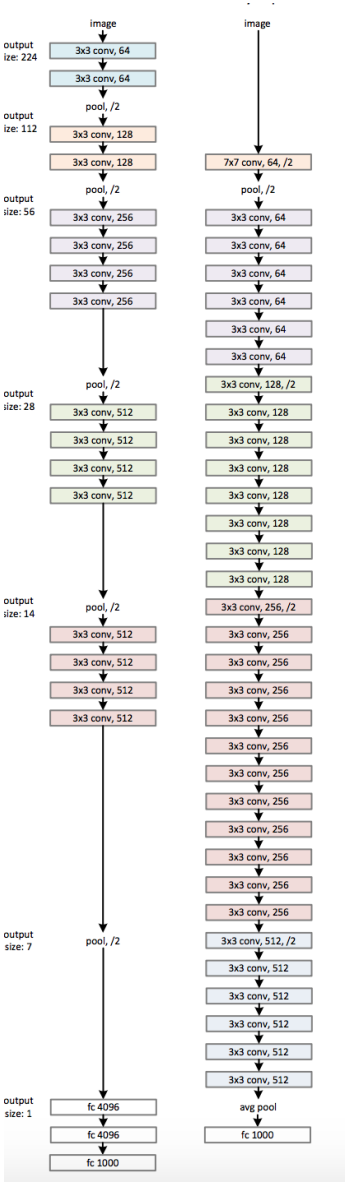
picture credits
neurohive.io/en/popular-networks/vgg16/



ResNet

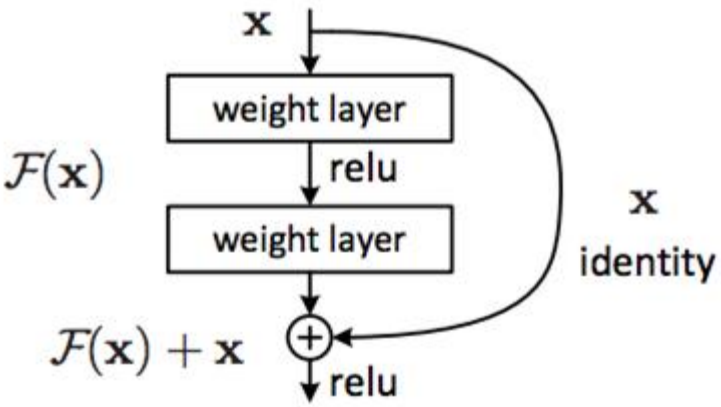
very deep ☺

one of the
state of the art
on ***image net***
www.image-net.org
- very large dataset
of labeled images
>14,000,000



Deep residual learning for image recognition. K. He, X. Zhang, S. Ren, and J. Sun. CVPR 2016

key technical trick

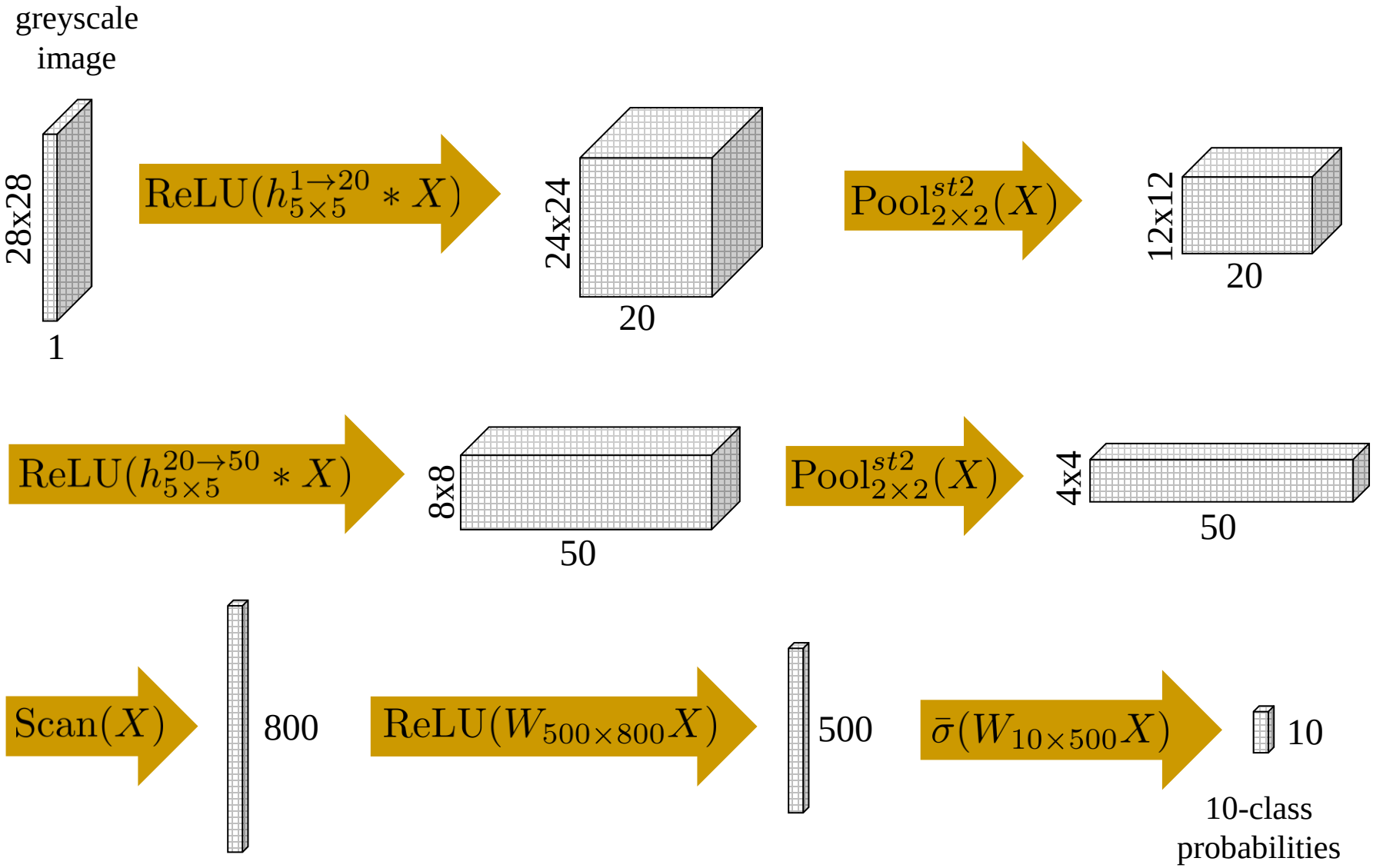


resnet block

(residual link helps gradient descent)

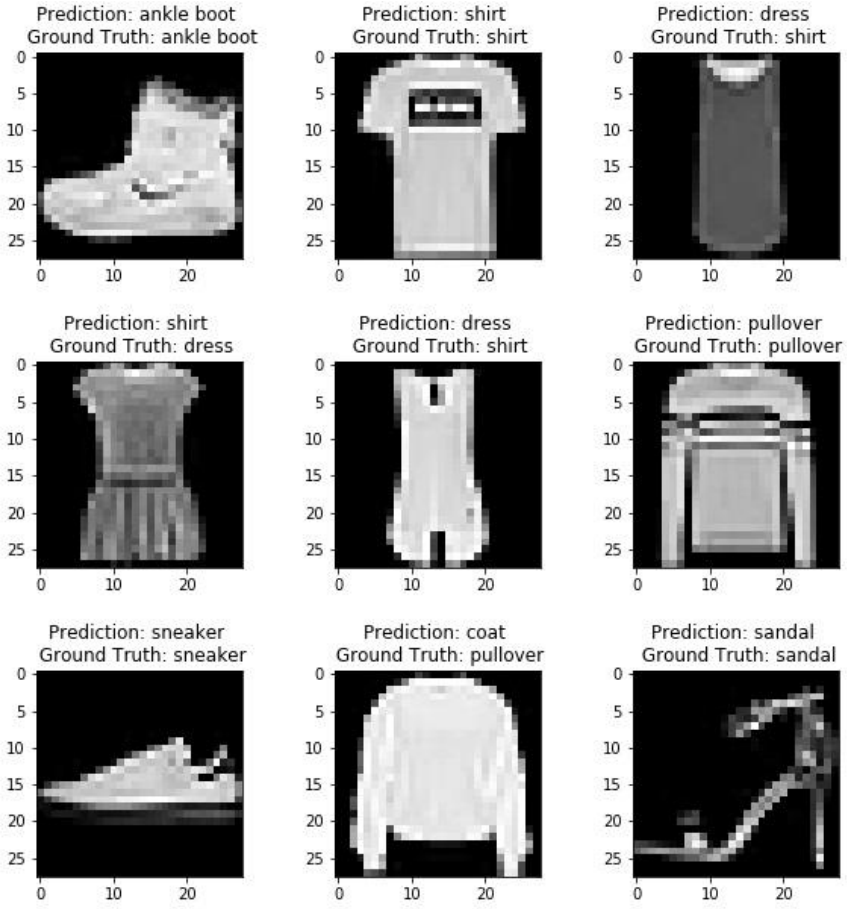
FashionMNIST classification example

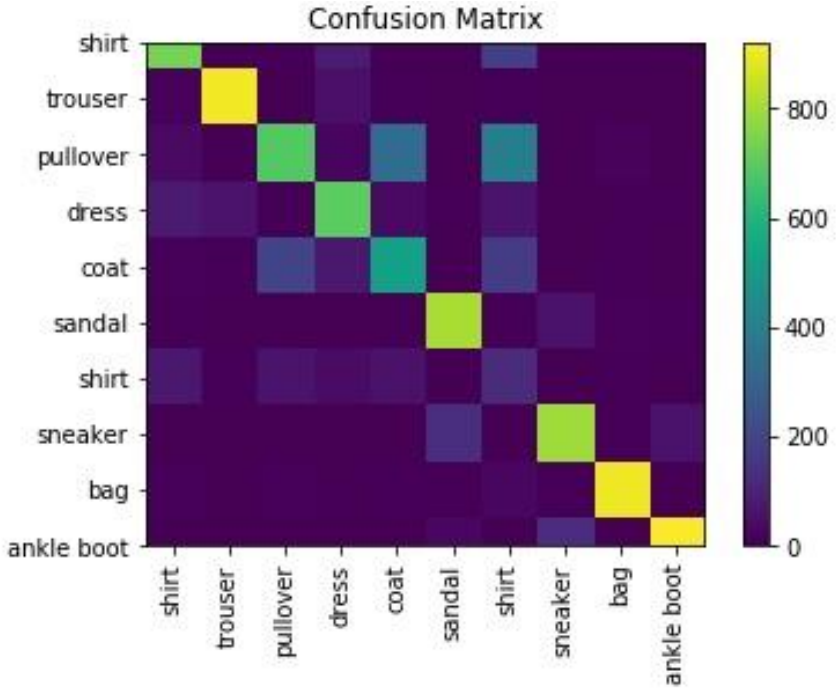
see class MyNet in Classification_Notebook_CS484_UW.ipynb



FashionMNIST classification example

see class MyNet in Classification_Notebook_CS484_UW.ipynb





Practical Issues for NN training

- data augmentation
  - addresses limited training data
  - e.g. transform available labeled data (domain and range transformations, deformations, cropping, etc.)
- stochastic gradient descent (SGD)
  - batch size selection
  - batch normalization (subtract batch *mean* and divide by batch *st.d.*)
- hyper-parameter tuning (e.g. learning rate)
  - break test data into "*validation*" data + (real) "*testing*" data
  - real testing data is often hidden, and one must use validation data for internal testing purposes, as in assignment 4
- debugging