

Artificial Intelligence

CE-417, Group 2

Computer Eng. Department

Sharif University of Technology

Fall 2020

By Mohammad Hossein Rohban, Ph.D.

Courtesy: Most slides are adopted from CSE-573 (Washington U.), original
slides for the textbook, and CS-188 (UC. Berkeley).

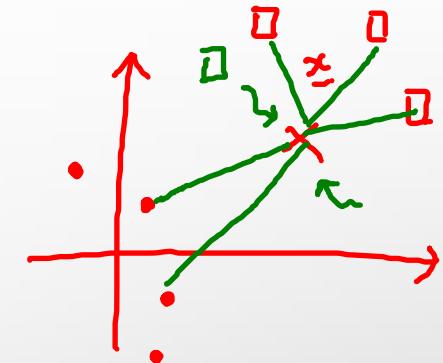
Neural Networks

$$P(y | \underline{x}) = \frac{1}{1 + e^{y \langle \underline{w}, \underline{x} \rangle}}$$

$\sum_{i=1}^d w_i x_i$

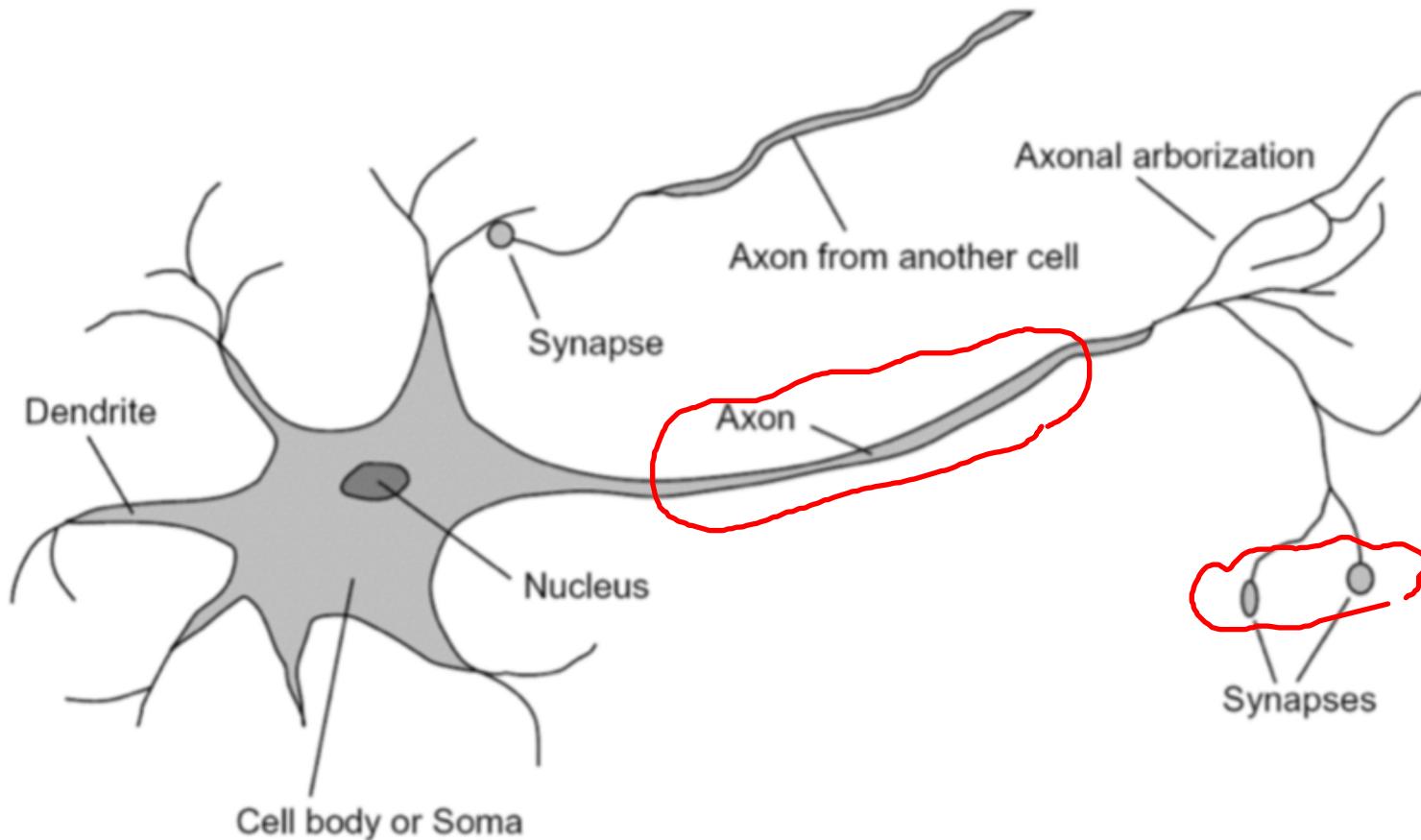
Linear classifiers: Perceptron

$$e^{-\frac{\|\underline{x}^{(i)} - \underline{x}\|^2}{2\sigma^2}}$$



- Decision trees
 - Inductive bias: use a combination of small number of features
- Nearest neighbor classifier (= estimate the label as majority votes of k-NNs of input in the training data in the feature space)
 - Inductive bias: all features are equally good $\rightarrow \|\underline{x} - \underline{z}\|_2^2 = \sum_{i=1}^d (x_i - z_i)^2$
- Logistic Regression, and Perceptron
 - Inductive bias: use all features, but some more than others
 - learning weights for features

A neuron



→ inspire
for a
new
computation

$$\left\{ \begin{array}{l} \underline{w^T x_0 + b = 0} \\ w^T \underline{x_1} + b = 0 \\ w^T (\underline{x - x_0}) = 0 \end{array} \right.$$

$\underline{x_0}$ ←
Perceptron

$$\underline{w^T x + b = 0} \\ \langle \underline{(w, b)}, (\underline{x}, 1) \rangle = 0$$

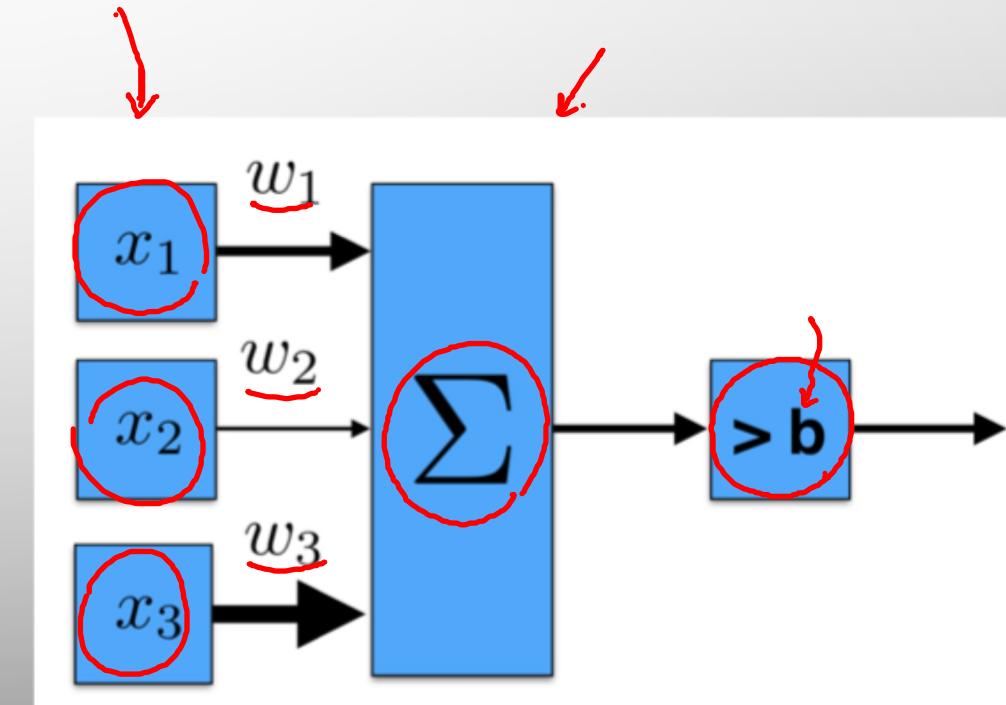
- Input are feature values
- Each feature has a weight
- Sum in the activation
- If the activation is:
 - $> b$, output class 1
 - otherwise, output class 2

$$x \rightarrow (x, 1)$$

$$\underline{w^T x} + \underline{b} \rightarrow \underline{(w, b)^T (x, 1)}$$

↓ sign ↓
↓ -1 ↓

$$\text{activation}(w, x) = \sum_i \underline{w_i x_i} = w^T x$$



Example: spam

- Imagine 3 features (spam is “positive” class):

- Free (number of occurrences of “free”)
- Money (number of occurrences of “money”)
- BIAS (intercept, always has value 1)

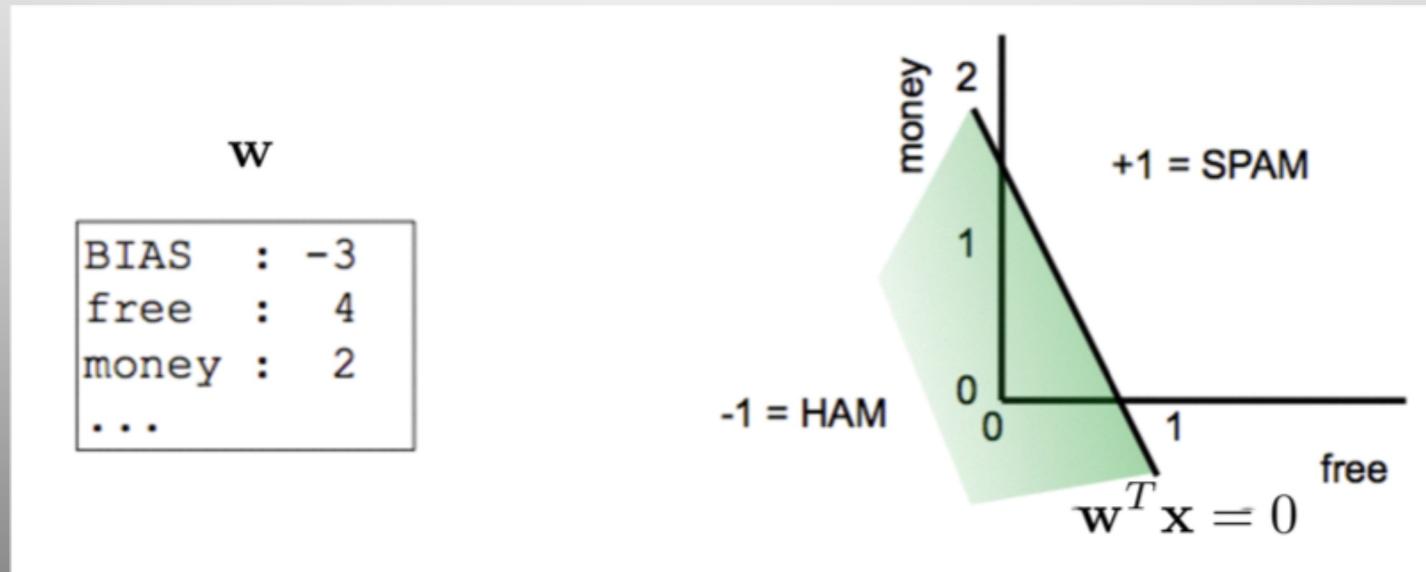
email	<u>x</u>	<u>w</u>	$w^T x$								
“free money”	<table border="1"><tr><td>BIAS : 1</td></tr><tr><td>free : 1</td></tr><tr><td>money : 1</td></tr><tr><td>...</td></tr></table>	BIAS : 1	free : 1	money : 1	...	<table border="1"><tr><td>BIAS : -3</td></tr><tr><td>free : 4</td></tr><tr><td>money : 2</td></tr><tr><td>...</td></tr></table>	BIAS : -3	free : 4	money : 2	...	$(1)(-3) + (1)(4) + (1)(2) + \dots = 3$
BIAS : 1											
free : 1											
money : 1											
...											
BIAS : -3											
free : 4											
money : 2											
...											

$w^T x > 0 \rightarrow \text{SPAM!!}$

Geometry of the perceptron

$$\textcircled{1} \quad \begin{cases} \underline{w}^T \underline{x}_0 + b = 0 \\ \underline{w}^T \underline{x} + b = 0 \\ \underline{w}^T (\underline{x} - \underline{x}_0) = 0 \end{cases}$$

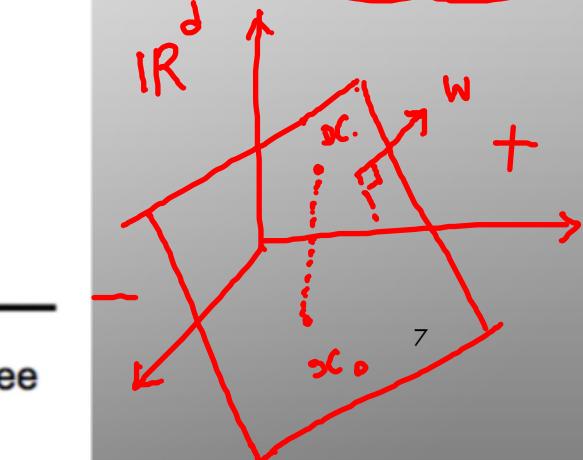
- In the space of feature vectors
 - Examples are points (in D dimensions)
 - A weight vector is a hyperplane (a D-1 dimensional object)
 - One side corresponds to $y=+1$
 - Other side corresponds to $y=-1$
- Perceptrons are also called as linear classifiers



Decision
Boundary ?

$$\underline{w}^T \underline{x} + b > 0$$

$$\textcircled{2} \quad \underline{w}^T \underline{x} + b = 0$$



Learning a perceptron

Input: training data $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)$

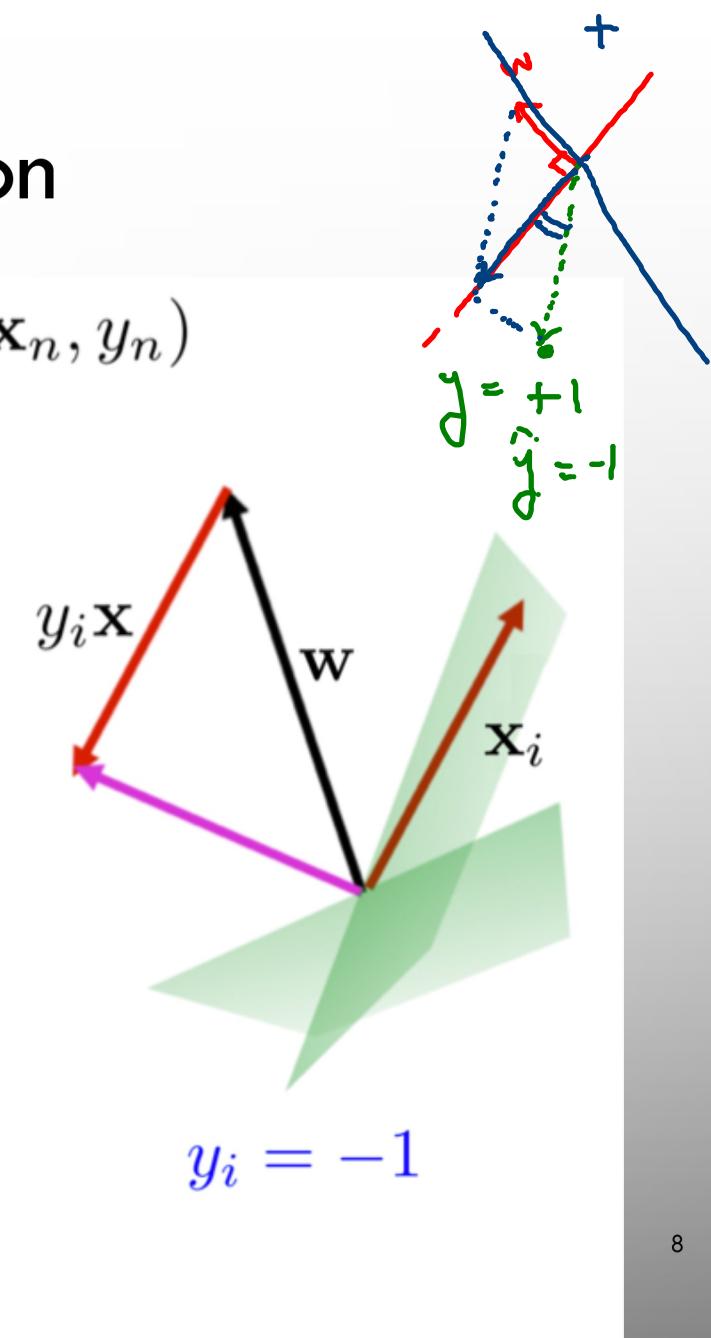
→ Perceptron training algorithm [Rosenblatt 57]

- ◆ Initialize $\mathbf{w} \leftarrow [0, \dots, 0]$
- ◆ for iter = 1, ..., T → epoch
- ▶ for i = 1, ..., n
 - predict according to the current model

true label → $\hat{y}_i = \begin{cases} +1 & \text{if } \mathbf{w}^T \mathbf{x}_i > 0 \\ -1 & \text{if } \mathbf{w}^T \mathbf{x}_i \leq 0 \end{cases}$

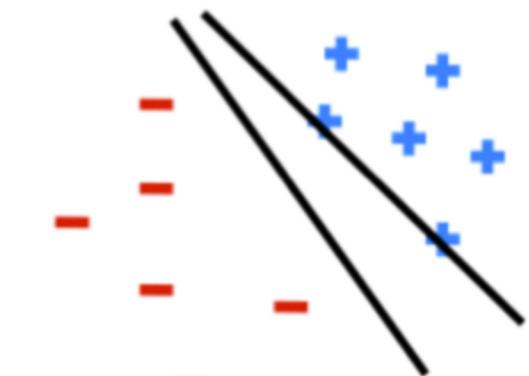
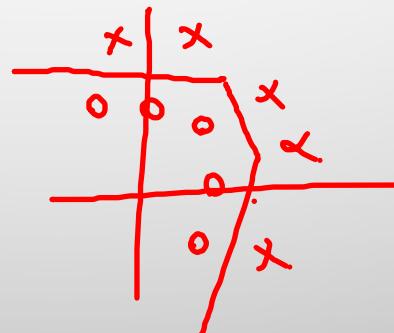
- if $y_i = \hat{y}_i$, no change
 - else, $\mathbf{w} \leftarrow \mathbf{w} + y_i \mathbf{x}_i$
- $K \leftarrow K+1$

$$\|\mathbf{x}_i\| \downarrow$$

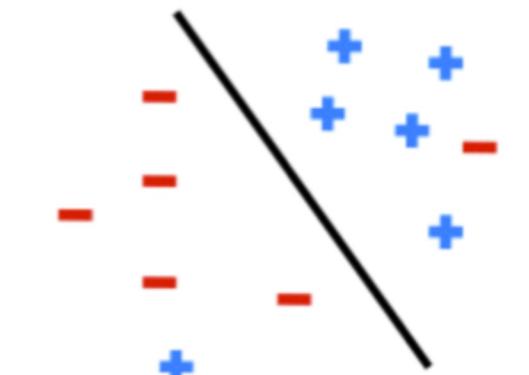


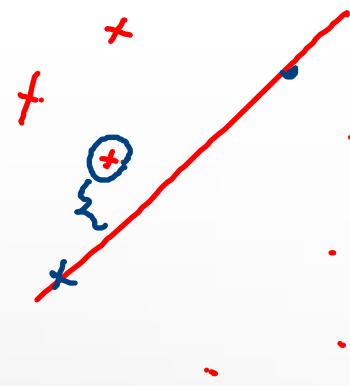
Properties of perceptrons linearly - Separable

- **Convergence:** if the training data is separable then the perceptron training will eventually converge [block 62, novikoff 62]

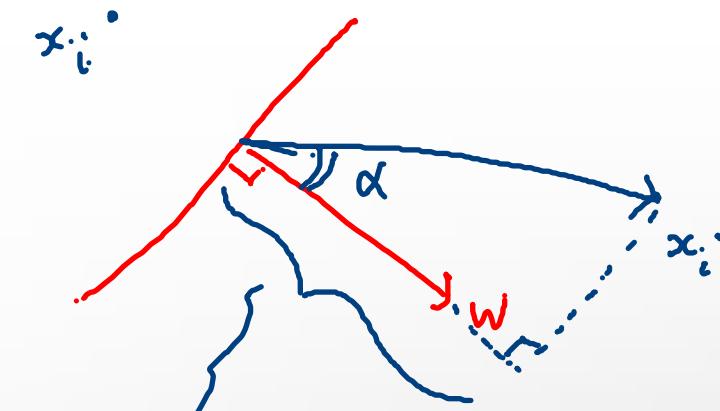


Non-Separable





Maximum margin



- Assuming that the data is linearly separable, we define margin as

$$\delta = \max_{\mathbf{w}} \min_{(\mathbf{x}_i, y_i)} |y_i \mathbf{w}^T \mathbf{x}_i|$$

such that, $\|\mathbf{w}\| = 1$

W orgmat

If the trn. set is

linearly separable $\Rightarrow \delta > 0$

intrinsic measure
separability

distance of x_i
to the separating
hyperplane.

$$\begin{aligned} \langle \hat{w}, w^{(k.)} \rangle &\gg \langle \hat{w}, w^{(k-1)} \rangle + \delta \\ &> \langle \hat{w}, w^{(k-2)} \rangle + \cdot \delta + \delta \end{aligned}$$

Proof of convergence :

Assumption : $\|\hat{w}\| = 1 \geq \langle \hat{w}, w^{(.)} \rangle + k\delta$

Val. of w
after the k -th
update.

Let, \hat{w} be the separating hyperplane with margin δ

$$\begin{aligned} \langle \hat{w}, w^{(k)} \rangle &= \hat{w}^T (\underbrace{w^{(k-1)}}_{\|\hat{w}\|} + \underbrace{y_i x_i}_{\text{update rule}}) \\ &= \hat{w}^T w^{(k-1)} + \underbrace{\hat{w}^T y_i x_i}_{\text{algebra}} \\ &\geq \hat{w}^T w^{(k-1)} + \delta \quad \text{definition of margin} \end{aligned}$$

w is getting closer



ideal
direction

- ① \hat{w} argmax of the for previous opt.
- ② For each i , $\hat{w}^T y_i x_i \geq \delta$

$$\|\vec{a}\|_2^2 = \langle \vec{a}, \vec{a} \rangle$$

Proof of convergence (cont.)

bound the norm

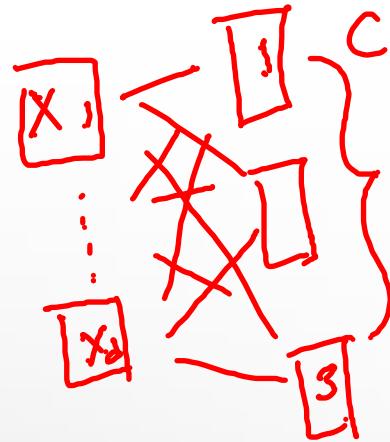
$$\begin{aligned}
 \|\mathbf{w}^{(k)}\|^2 &= \|\mathbf{w}^{(k-1)} + y_i \mathbf{x}_i\|^2 \\
 &\leq \|\mathbf{w}^{(k-1)}\|^2 + \|y_i \mathbf{x}_i\|^2 \\
 &\leq \|\mathbf{w}^{(k-1)}\|^2 + 1 \\
 &\leq k
 \end{aligned}$$

update rule
 $y_i \mathbf{w}^{(k-1)T} \mathbf{x}_i \leq 0$
 norm
 $\|\mathbf{w}^{(k)}\| \leq \sqrt{k}$

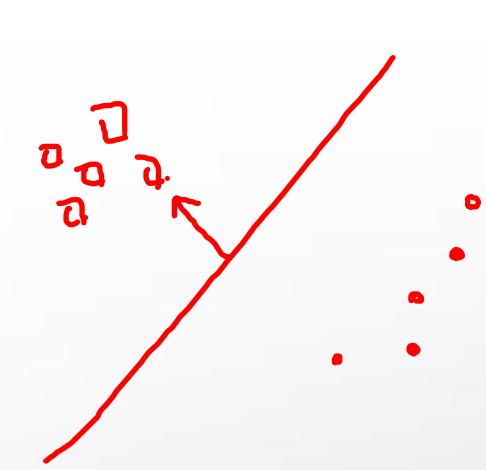
all your
 training data
 lie inside the
 unit hyper ball.

$$\begin{aligned}
 k\delta \leq \|\mathbf{w}^{(k)}\| &\leq \sqrt{k} \\
 \frac{1}{\delta} &\leq \sqrt{k} \\
 k &\leq \frac{1}{\delta^2}
 \end{aligned}$$

k is bounded



Limitations of perceptrons



- **Convergence:** if the data isn't separable, the training algorithm may not terminate. → Pocket Alg. → Trn. error
- **Overtraining:** test/validation accuracy rises and then falls.

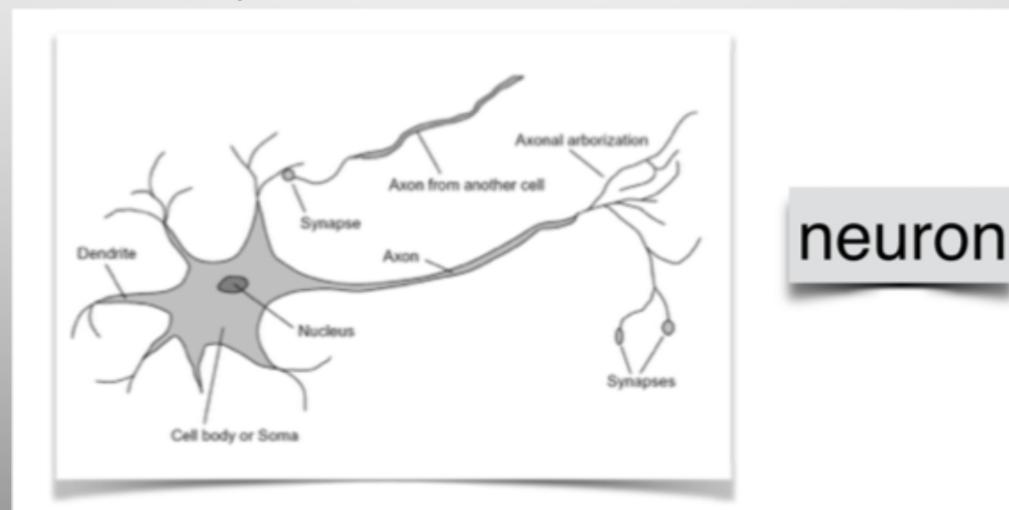
MLP

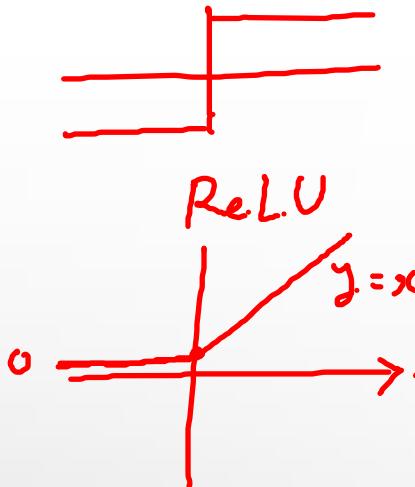
a series of

Connected
perceptrons.

Multi-Layered Perceptron : Motivation

- One of the main weakness of linear models is that they are **linear**.
- Decision trees and **k-NN classifiers** can model **non-linear** boundaries.
- Multi-layer neural networks are yet another **non-linear** classifier.
- Take the biological inspiration further by chaining together **perceptrons**.
- Allows us to use what we learned about linear models:
 - Loss functions, regularization, optimization





activation
function

Non-linearity is important

link function

$$h_i = f(\mathbf{w}_i^T \mathbf{x})$$

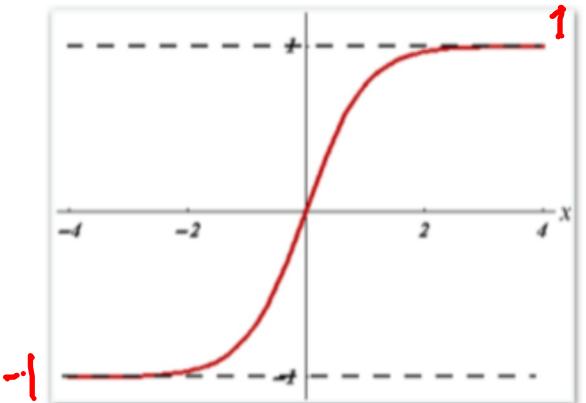
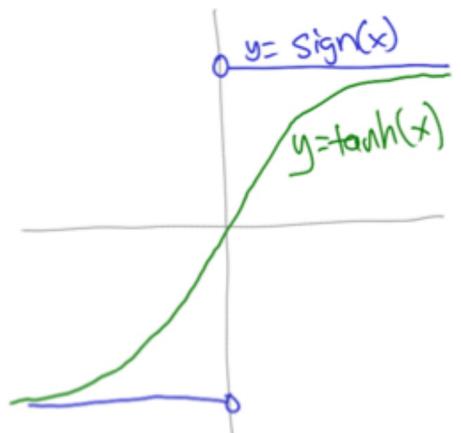
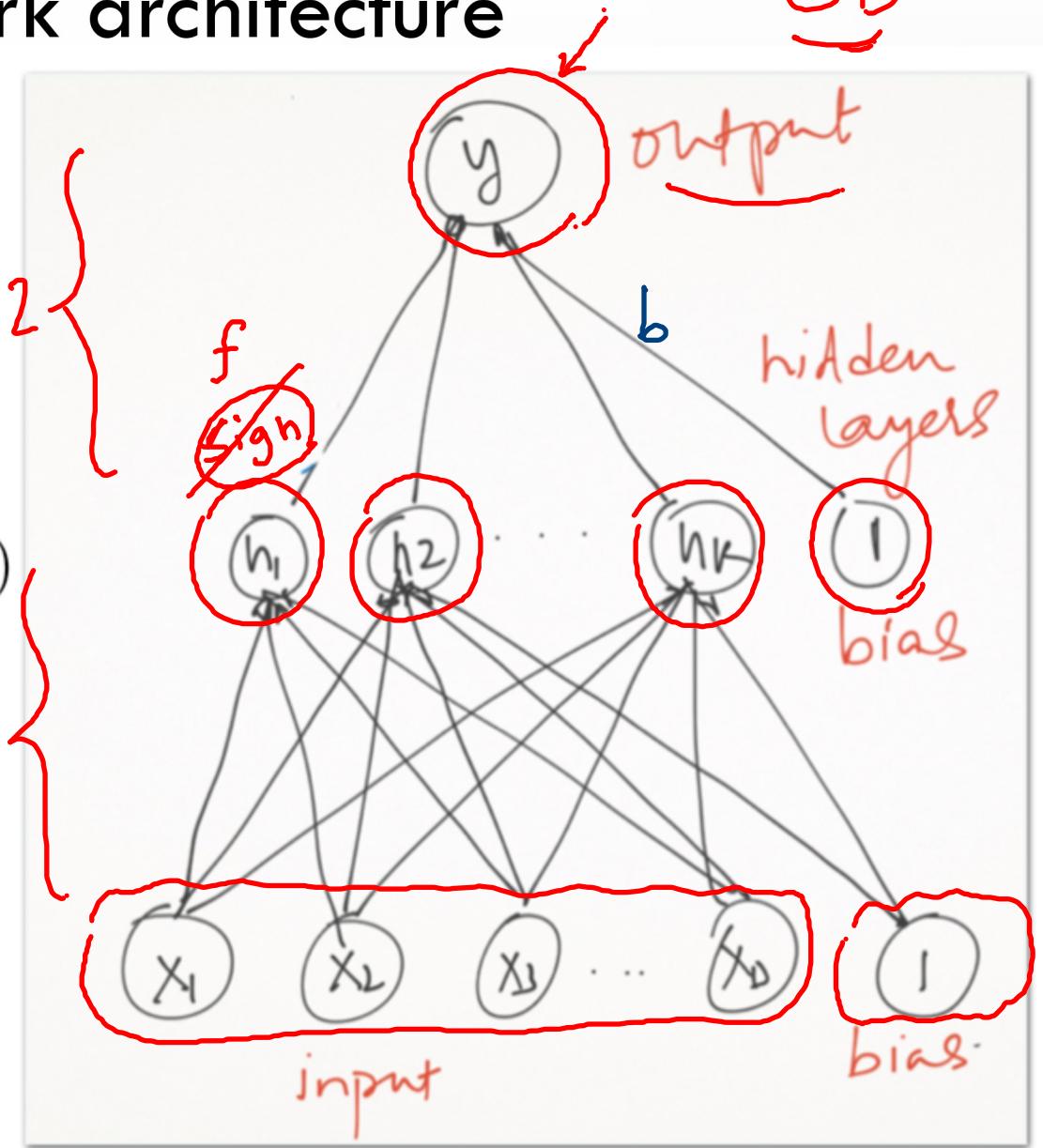


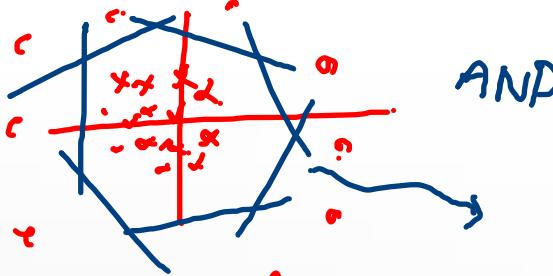
Figure 10.2: picture of sign versus tanh
¹ Its derivative is just $1 - \tanh^2(x)$.

$$\tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

Two-layer network architecture

Opt. \rightarrow General purpose Solver
GD

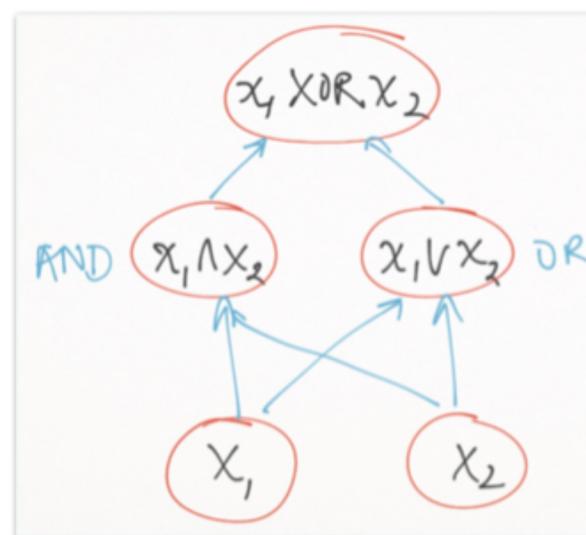




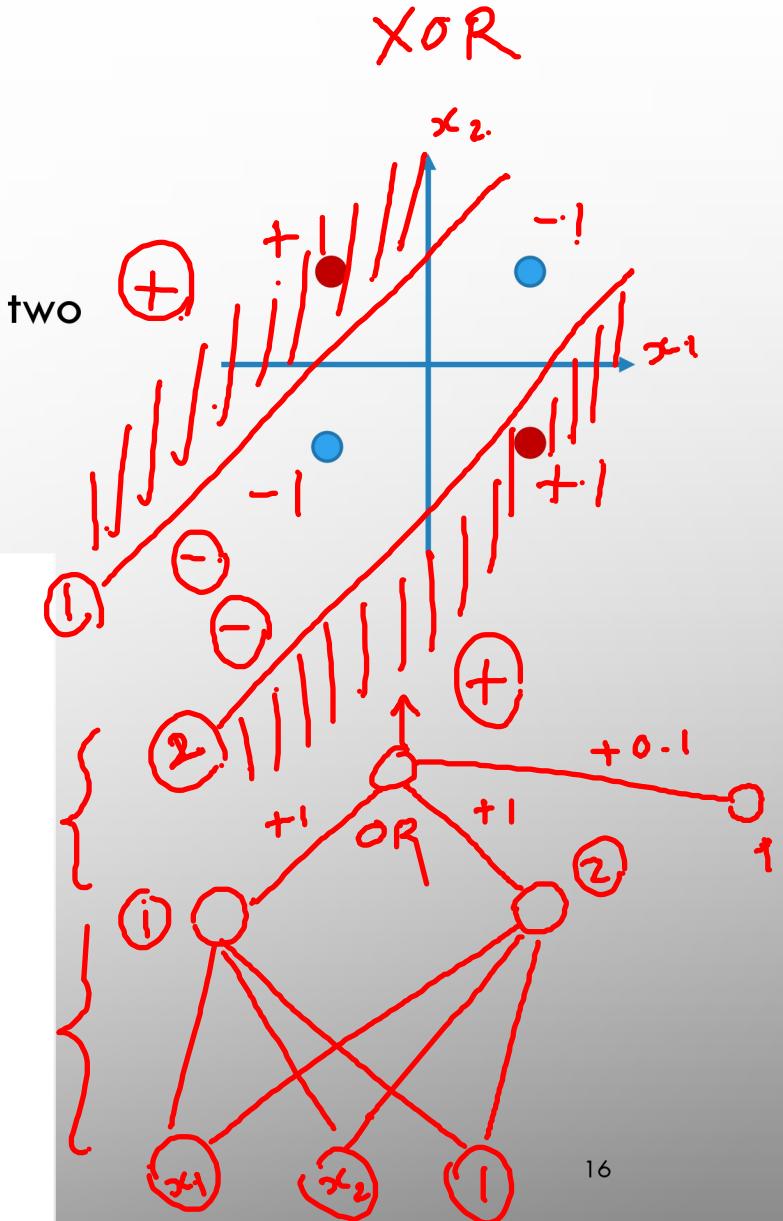
The XOR function

- Note that a perceptron cannot learn the **XOR** function:
- Exercise:** come up with the parameters of a two layer network with two hidden units that computes the **XOR** function.
- Here is a table with a bias feature for XOR

y	x_0	x_1	x_2
+1	+1	+1	+1
+1	+1	-1	-1
-1	+1	+1	-1
-1	+1	-1	+1



Do we gain anything beyond two layers?



EXPRESSIVE POWER OF A TWO-LAYER NETWORK

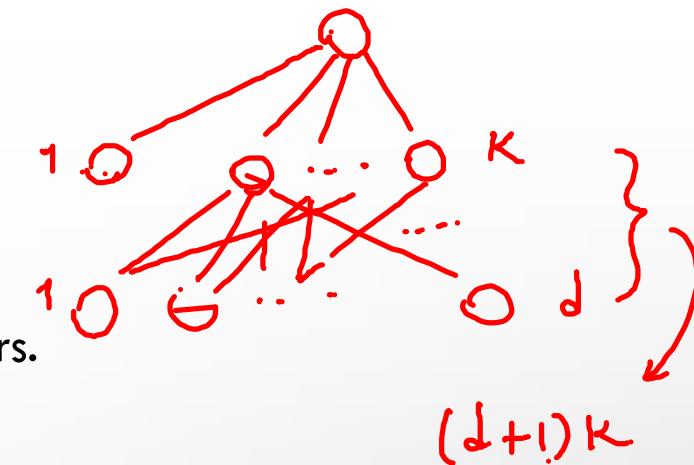
Theorem 10 (Two-Layer Networks are Universal Function Approximators). Let F be a continuous function on a bounded subset of D -dimensional space. Then there exists a two-layer neural network \hat{F} with a finite number of hidden units that approximate F arbitrarily well. Namely, for all x in the domain of F , $|F(x) - \hat{F}(x)| < \epsilon$.

Is it really a good news ?!

- Colloquially “a two-layer network can **approximate** any function”.
- Going from one to two layers dramatically improves the representation power of the network.

How many hidden units?

- d dimensional data with k hidden units has $(d+2)k + 1$ parameters.
 - $(d+1)k$ in the first layer (1 for the bias) and $k+1$ in the second layer
- With n training examples, set the number of hidden units $k \sim n/d$ to keep the number of parameters comparable to size of training data.
- k is both a form of **regularization** and **inductive bias**
- Training and test error vs. k



$$\begin{aligned} & \xrightarrow{10} \\ & \rightarrow (d+2)k+1 \\ & \quad \underbrace{\qquad\qquad\qquad}_{1000} \end{aligned}$$

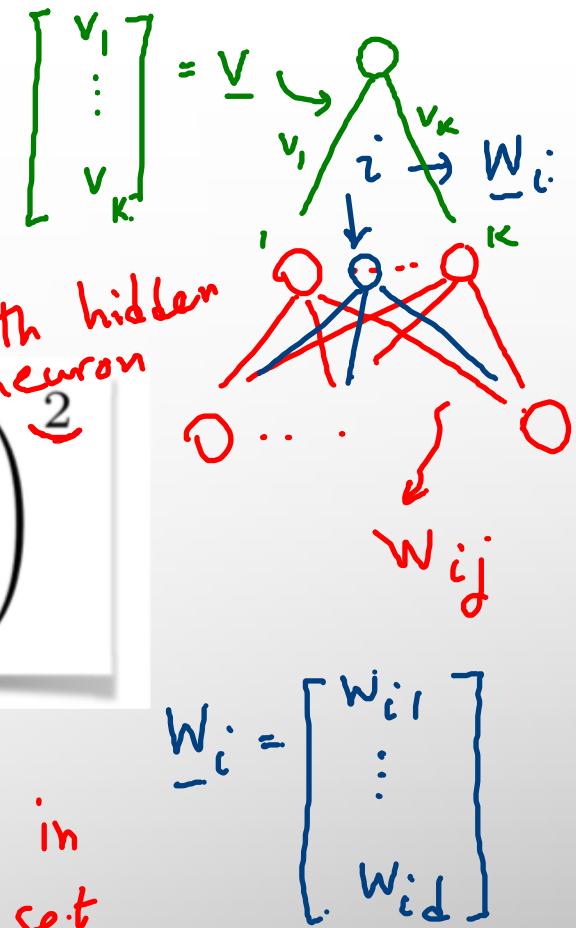
* The complexity of the model (k) should be determined by the amount of ¹⁸ trn. samples, not by complexity of the target.

Training a two-layer network

- Optimization framework:

$$\min_{W, v} \sum_n \frac{1}{2} \left(\underbrace{y_n}_{\text{ideal output for } x_n \text{ in trn. set.}} - \sum_i \underbrace{v_i f(\mathbf{w}_i^T \mathbf{x}_n)}_{\substack{\text{output of the last neuron} \\ \text{output of the } i\text{-th hidden neuron}}} \right)^2$$

MSE



- Loss minimization: replace squared-loss with any other.
- Regularization:
 - Traditionally NN are not regularized (early stopping instead)
 - But you can add a regularization (e.g. L₂-norm of the weights)
- Optimization by gradient descent
 - Highly non-convex problem so no guarantees about optimality. → GD

Training a two-layer network (cont.)

- Optimization framework:

$$\min_{W,v} \sum_n \frac{1}{2} \left(y_n - \sum_i \mathbf{v}_i f(\mathbf{w}_i^T \mathbf{x}_n) \right)^2$$

- Or equivalently

$$\min_{W,v} \sum_n \frac{1}{2} (y_n - \mathbf{v}^T \mathbf{h}_n)^2$$

$$\mathbf{h}_{i,n} = f(\mathbf{w}_i^T \mathbf{x}_n)$$

- Computing gradients: second layer

$$\mathbf{v}^{(t+1)} \leftarrow \mathbf{v}^{(t)} - \alpha \sum_{n=1}^N g_n$$

\mathbf{g}_n activation values for the n -th trn. sample.

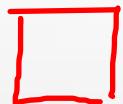
$$\mathbf{g}_n \in \mathbb{R}^K$$

$$\frac{dL_n}{d\mathbf{v}} = - (y_n - \mathbf{v}^T \mathbf{h}_n) \mathbf{h}_n$$

scalar vec.

$$\underline{w}_i^{(t+1)} \leftarrow \underline{w}_i^{(t)} - \alpha \sum_n g_i^{(n)}$$

$\underline{x}_1 \dots \underline{x}_N$



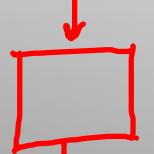
act. val.



update weights

- Computing gradients: first layer
 - Chain rule of derivatives

$\underline{x}_1 \dots \underline{x}_N$



act. val.

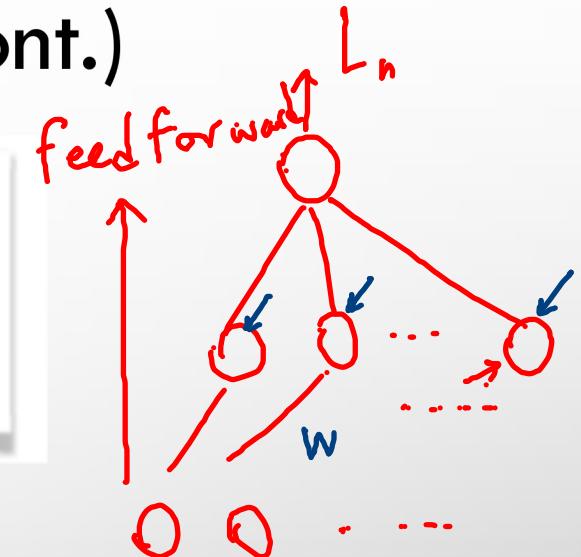


act. val. j -th

update weights, hidden neuron

Training a two-layer network (cont.)

$$\min_{W,v} \sum_n \frac{1}{2} \left(y_n - \sum_i \mathbf{v}_i f(\mathbf{w}_i^T \mathbf{x}_n) \right)^2$$



Chain rule:

$$\frac{dL_n}{dh_j}$$

$$\frac{dh_i}{dW_i} \rightarrow \text{Grad}$$

$$\frac{dL_n}{d\mathbf{w}_i} = \sum_j \frac{dL_n}{dh_j} \frac{dh_j}{d\mathbf{w}_i}$$

O if $i \neq j$

$$\frac{dL_n}{d\mathbf{w}_i} = -(y_n - v^T h_n) v_i f'(\mathbf{w}_i^T \mathbf{x}_n) \mathbf{x}_n = g'_i$$

vec. of also called as back-propagation
the weights

that are connected to the i -th hidden
neuron.

Sample from the trn. set.

← mini-batch

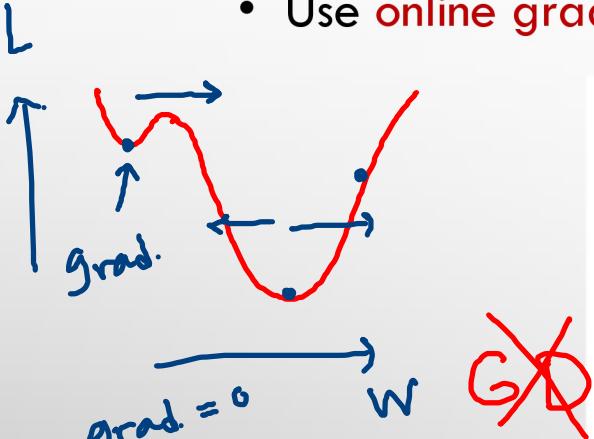


CPU

GPU
CUDA

Practical issues: gradient descent

- Use online gradients (or stochastic gradients)



$$w \leftarrow w - \eta \frac{dL_n}{dw}$$

SGD → computationally efficient

$\sum_{n \in \text{mini-batch}(j)} L_n$

$$\frac{dL}{dw} = \sum_n \frac{dL_n}{dw}$$

mean → sample better

online



- Learning rate η : start with a high value and reduce it when the validation error stops decreasing.
- Momentum: move out small local minima

Schedule

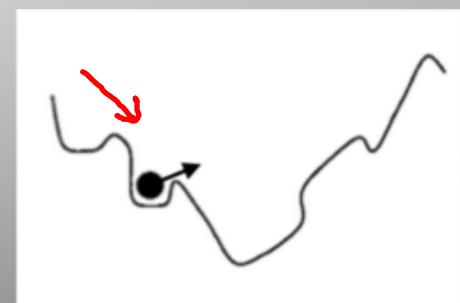


Usually set to a high value: $\beta = 0.9$

$$\Delta w^{(t)} = \beta \Delta w^{(t-1)} + (1 - \beta) \left(-\eta \frac{dL_n}{dw^{(t)}} \right)$$

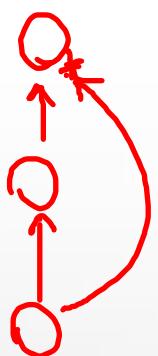
$$w^{(t+1)} \leftarrow w^t + \Delta w^{(t)}$$

prev. grad. upd.



Adam
:

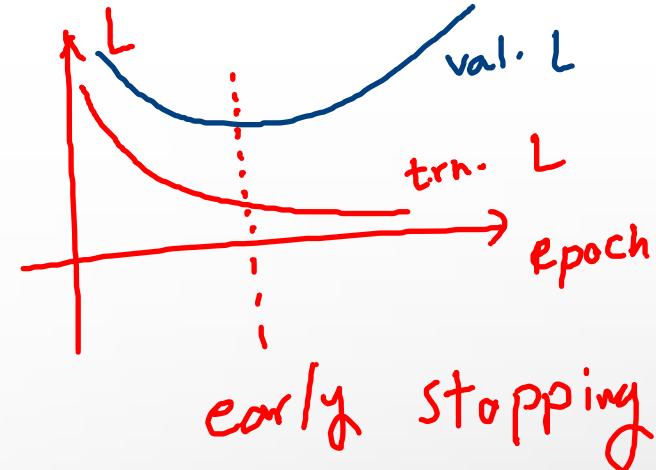
22



- ① Batch Normalization → mitigate Vanishing Grad.
- ② Res Net (2016)

Practical issues: initialization

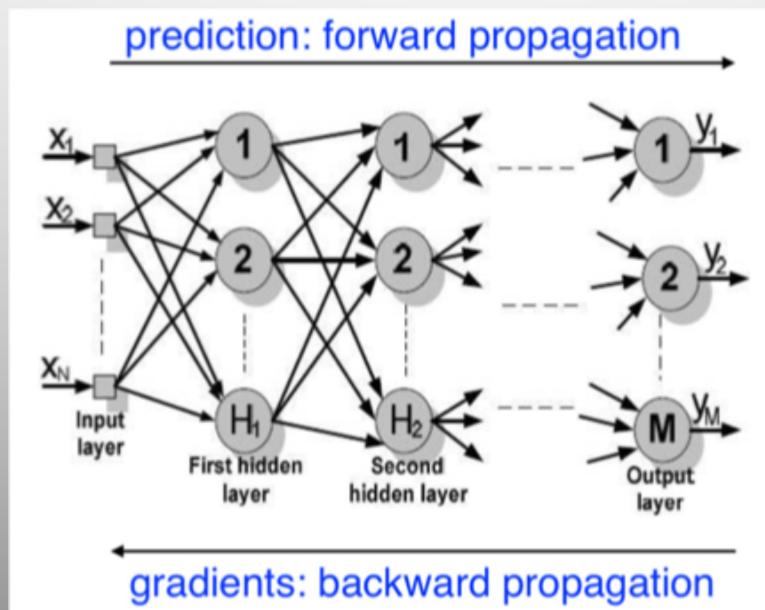
Xavier $\xrightarrow{\text{Vanishing Gradient}} \frac{\partial L}{\partial w}$



- Initialization didn't matter for linear models
 - Guaranteed convergence to global minima as long as step size is suitably chosen since the **objective is convex**
- Neural networks are **sensitive to initialization**
 - Many local minima
 - **Symmetries:** reorder the hidden units and change the weights accordingly to get another network that produces identical outputs
 - Train multiple networks with randomly initialized weights, and pick the one that yields best validation accuracy.

Beyond two layers

- The architecture generalizes to any directed acyclic graph (DAG)
 - For example a multi-layer network
 - One can order the vertices in a DAG such that all edges go from left to right (topological sorting)
 - Gradient can be computed **recursively** using the **chain rule**.



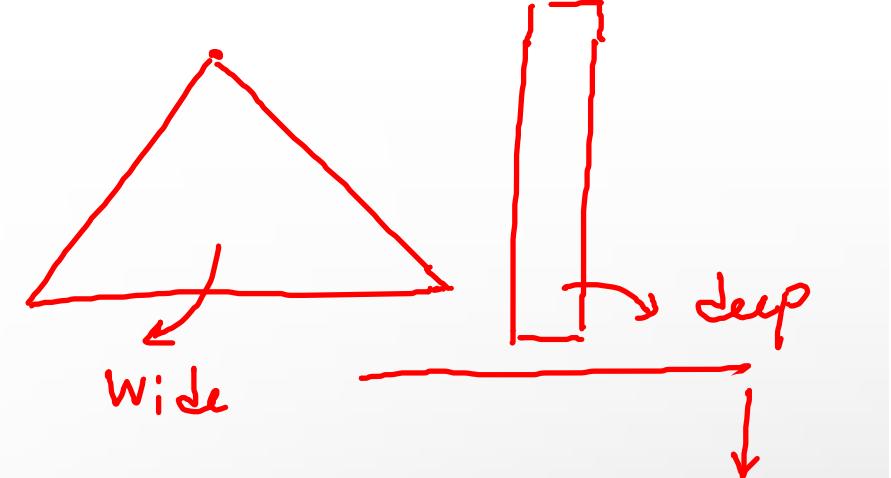
Breadth vs. Depth

- Why train deeper networks?
- We will borrow ideas from theoretical computer science:

- A **boolean circuit** is a DAG where each node is either an **input**, an **AND gate**, an **OR gate**, or a **NOT gate**. One of these is designated as an **output gate**.
- **Circuit complexity** of a **boolean function** f is the size of the smallest circuit (i.e., with the fewest nodes) that can compute f .
- **The parity function**: the number of 1s is even or odd

$$\text{parity}(\mathbf{x}) = \left(\sum_d x_d \right) \bmod 2$$

[Håstad, 1987] A depth- k circuit requires $\exp\left(n^{\frac{1}{k-1}}\right)$ to compute the parity function of n inputs



Over param.

Sanjeev

Arora.

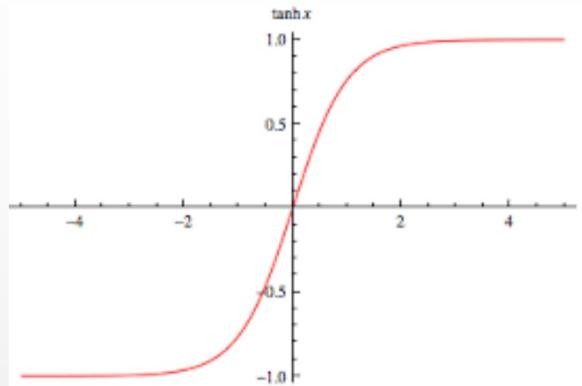


Breadth vs. Depth

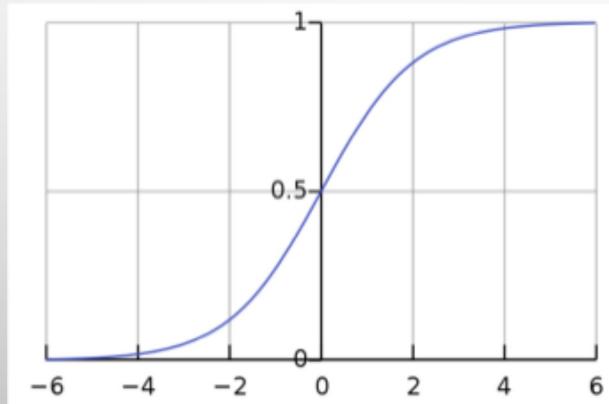
- Why **not** train deeper networks?
- Vanishing gradients
 - Gradients shrink as one moves away from the output layer
 - Convergence is slow
- But:
- Training deep networks is an active area of research.
 - Layer-wise initialization (perhaps using unsupervised data)
 - Engineering: GPUs to train on massive labelled datasets

Choices of link function

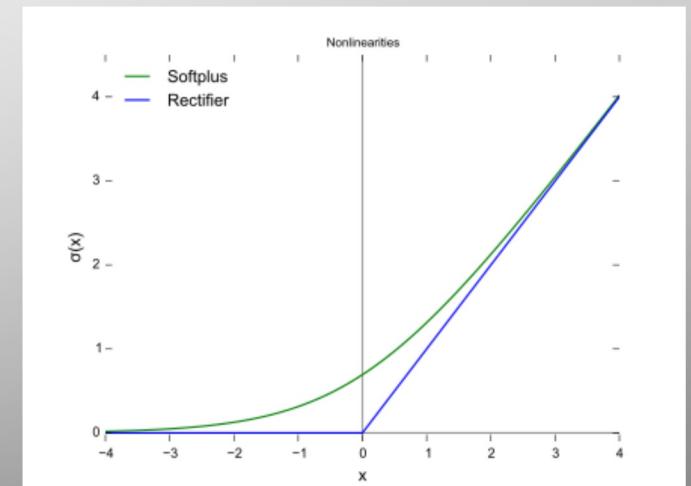
- Tanh

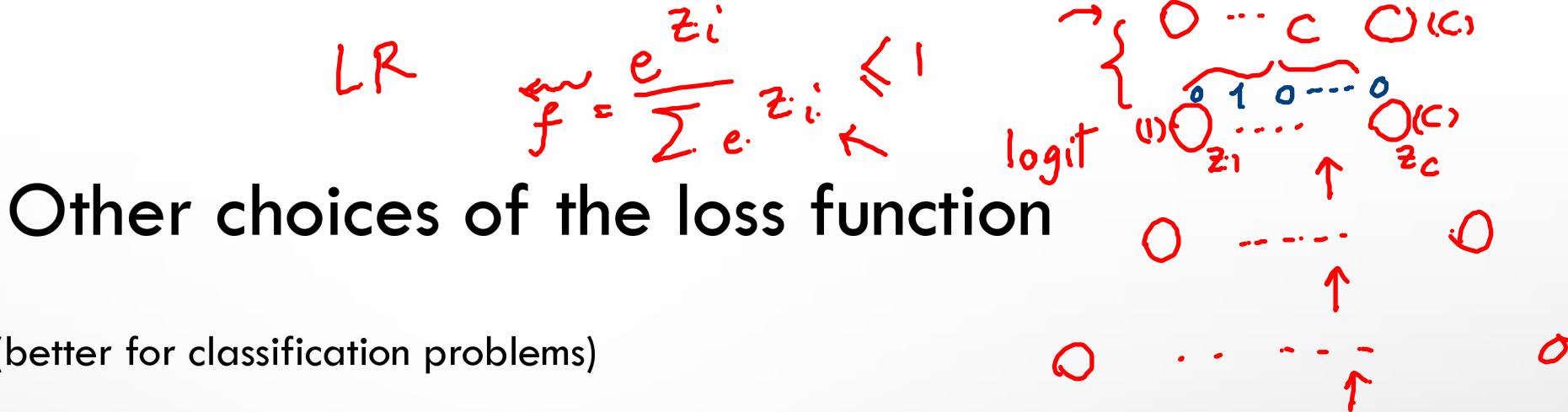


- Sigmoid



- Relu (rectified linear unit) (fewer vanishing gradient problems)





Other choices of the loss function

- Cross entropy (better for classification problems)
- For **binary classification** ($y = 0$ or 1 and f is the network output, designed to be between 0 and 1)

$$l(y_n, f(x_n)) = -(y_n \log f(x_n) + (1 - y_n) \log(1 - f(x_n)))$$

- For **multi-class** problems:

- If there are M classes, we would make M output neurons in the last layer, each designed to be between 0 and 1 .

$$l(y_n, f(x_n)) = -\sum_{c=1}^M \mathbb{I}(y_n = c) \log f_c(x_n)$$

$-\log(f_j(x_i))$

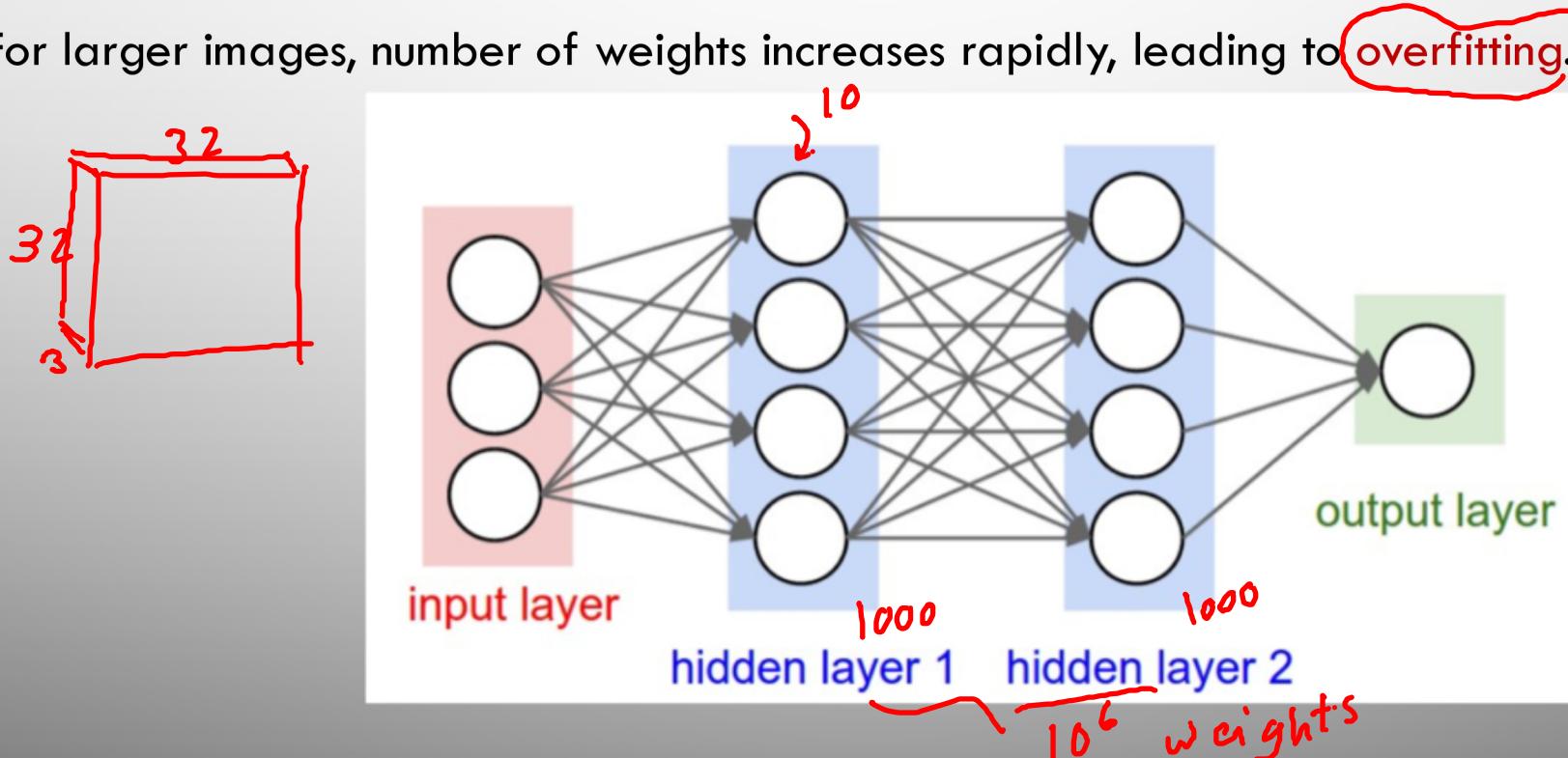
\downarrow

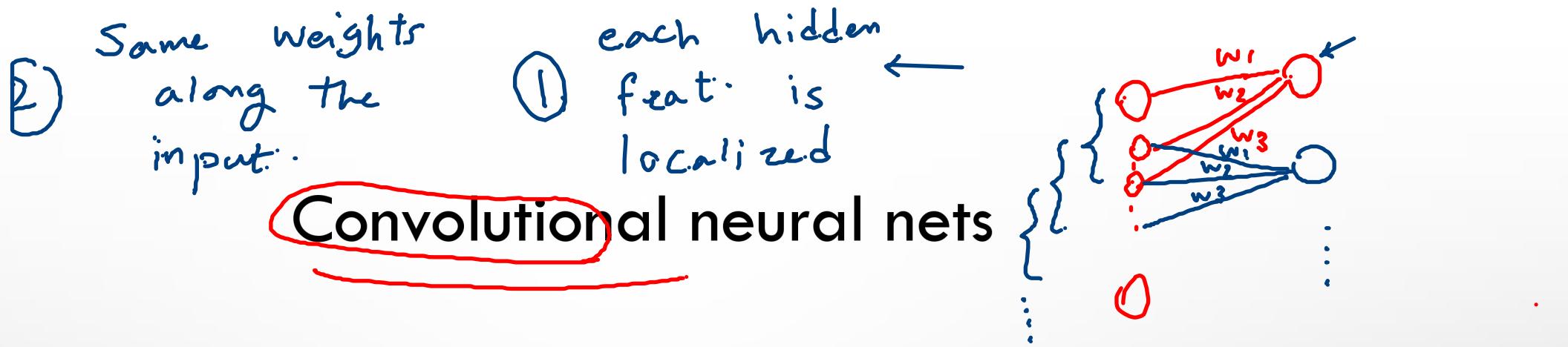
How to make output neurons be between 0 and 1?

- Use softmax:
- $f_j(x_n) = \frac{\exp(\beta h_j)}{\sum_{c=1}^M \exp(\beta h_c)}.$
- If $\beta \rightarrow \infty$, this would be equivalent with the argmax function.

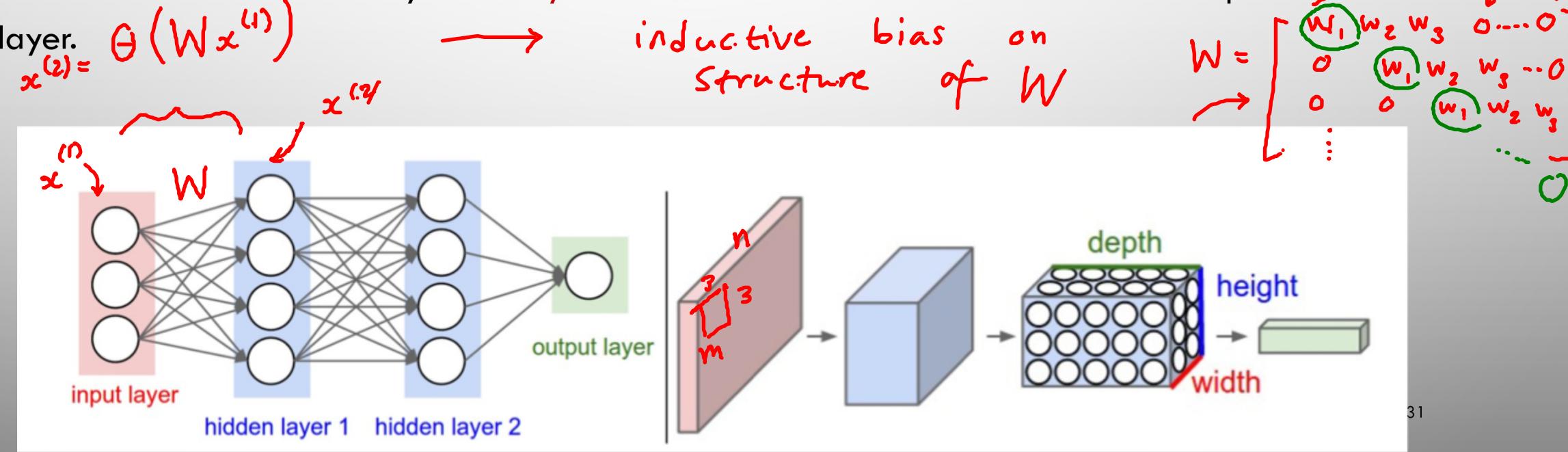
Image as input to a neural network

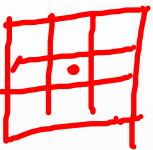
- Regular neural nets don't scale well to full images.
- Requires $32 \times 32 \times 3 = \underline{3072}$ weights just for a single neuron in the 2nd layer.
- For larger images, number of weights increases rapidly, leading to **overfitting**.



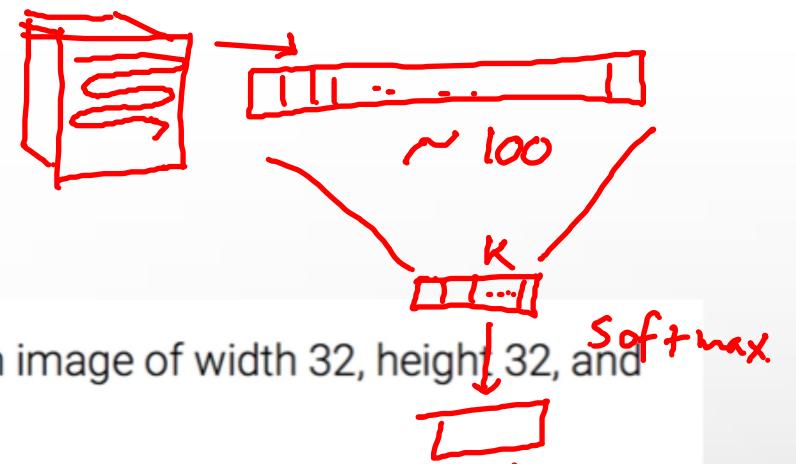


- The layers of a convnet have neurons arranged in 3 dimensions: **width, height, depth**.
- Each neuron in the hidden layers is **only connected to a few number of neurons** in the previous layer.



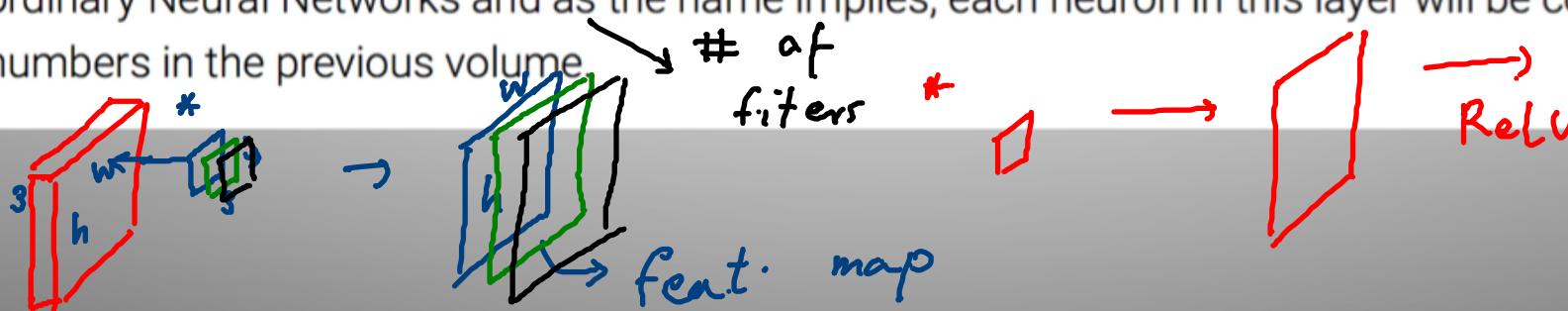
filter → 

Layers used in convnets

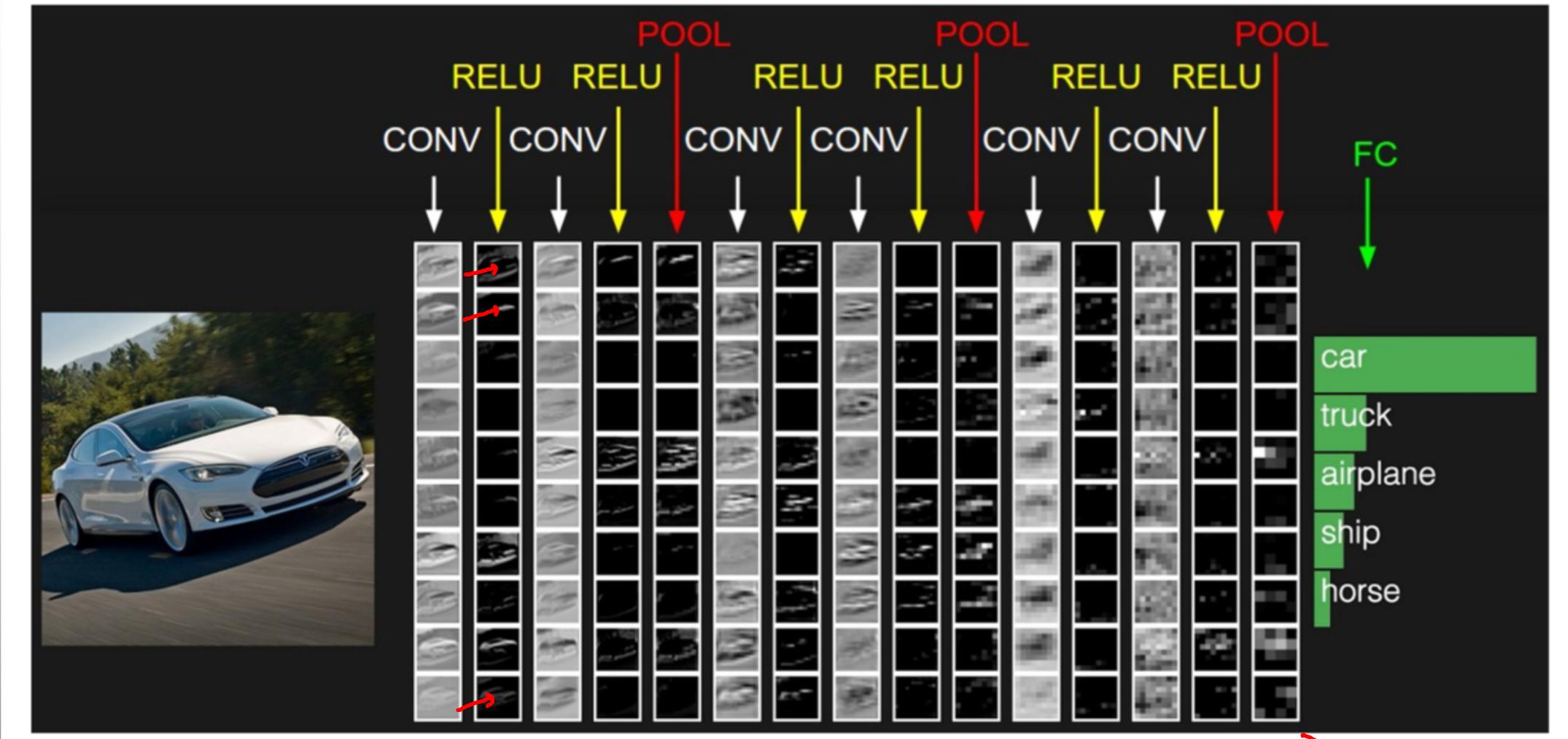


- INPUT [32x32x3] will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R,G,B.
- CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume such as [32x32x12] if we decided to use 12 filters.
- RELU layer will apply an elementwise activation function, such as the $\max(0, x)$ thresholding at zero. This leaves the size of the volume unchanged ([32x32x12]).
- POOL layer will perform a downsampling operation along the spatial dimensions (width, height), resulting in volume such as [16x16x12].
- FC (i.e. fully-connected) layer will compute the class scores, resulting in volume of size [1x1x10], where each of the 10 numbers correspond to a class score, such as among the 10 categories of CIFAR-10. As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume.

Flatten

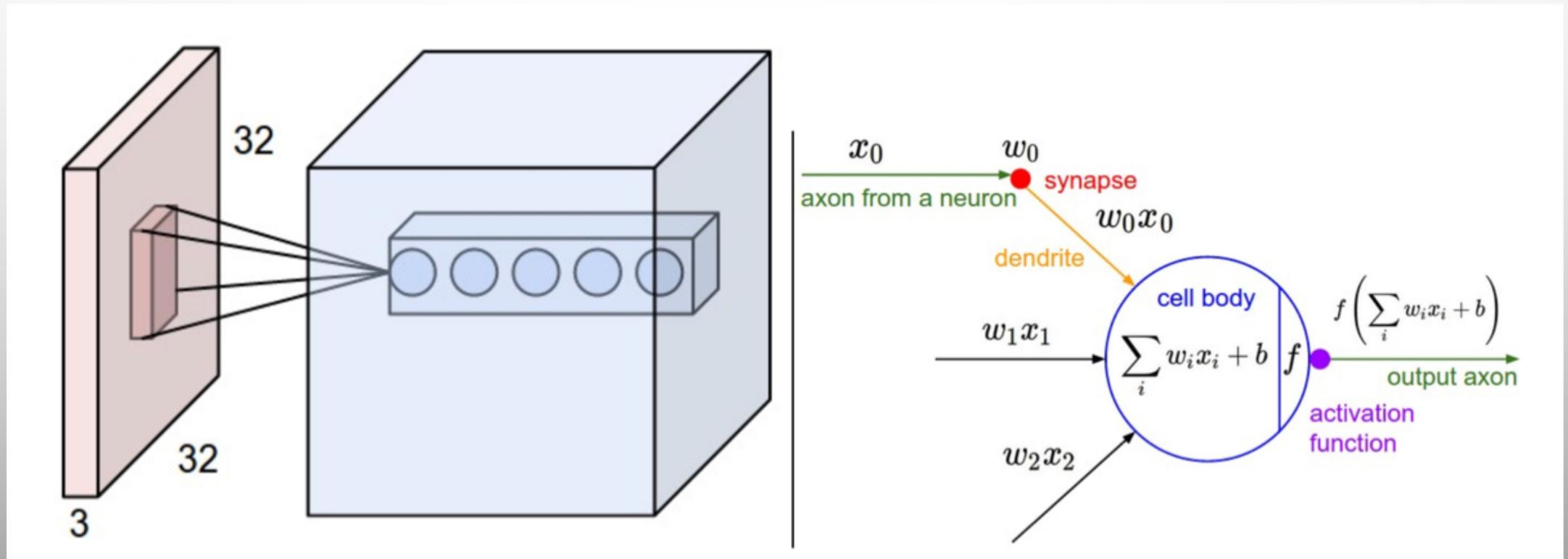


Layers used in convnets (cont.)

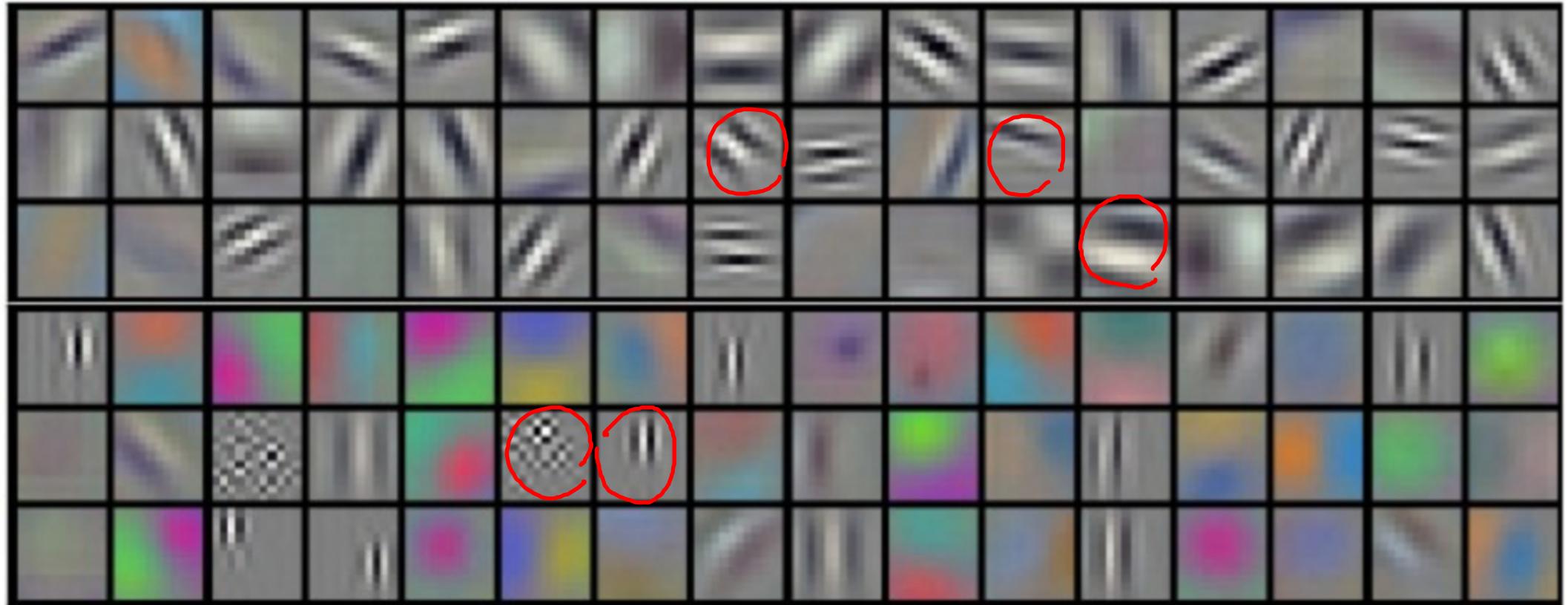


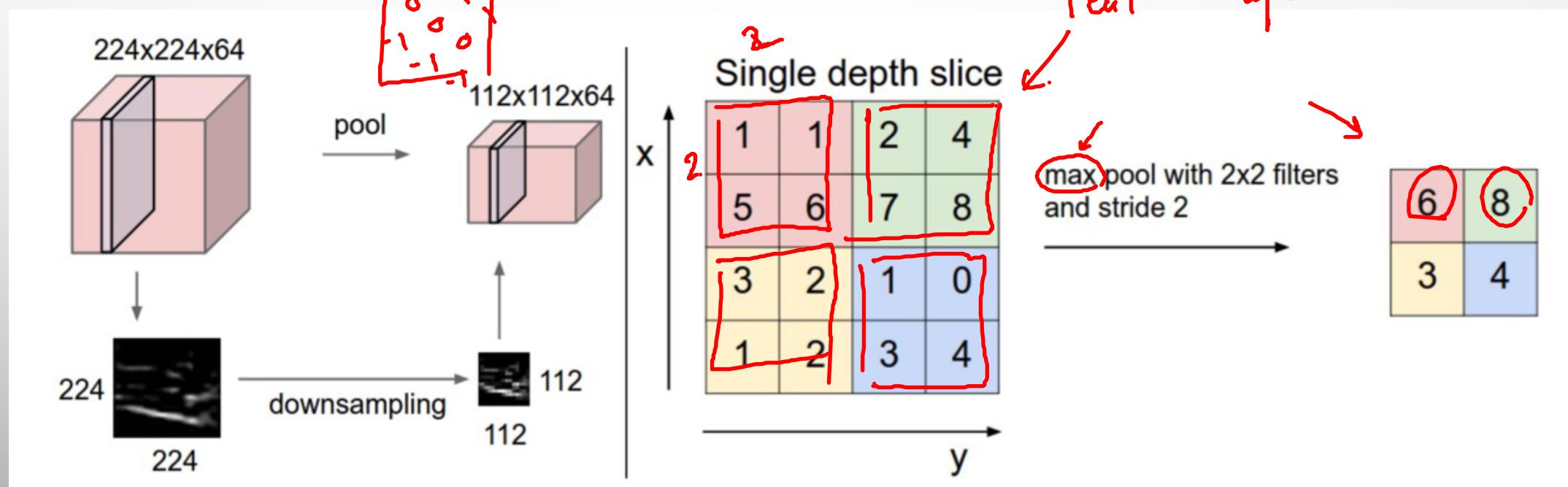
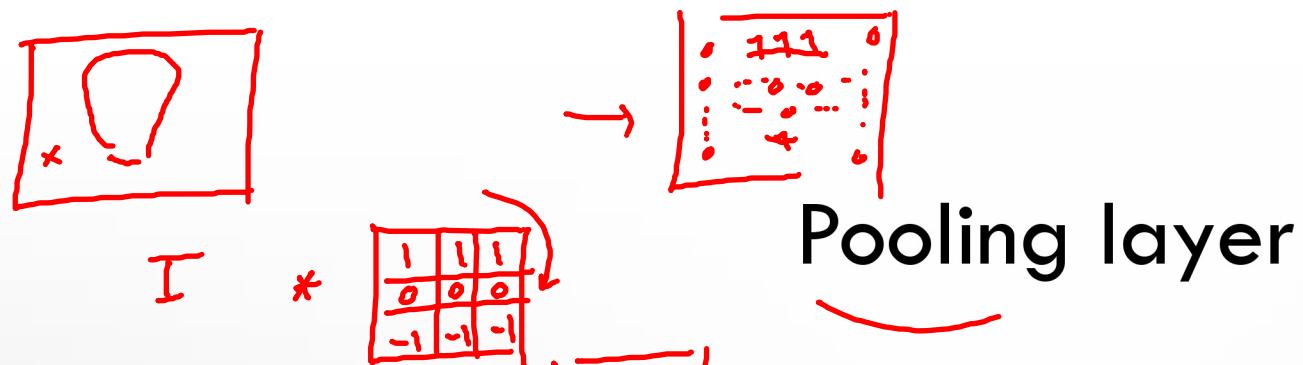
Back prop. ~1000 \xrightarrow{Fc} 5

Convolutional layer



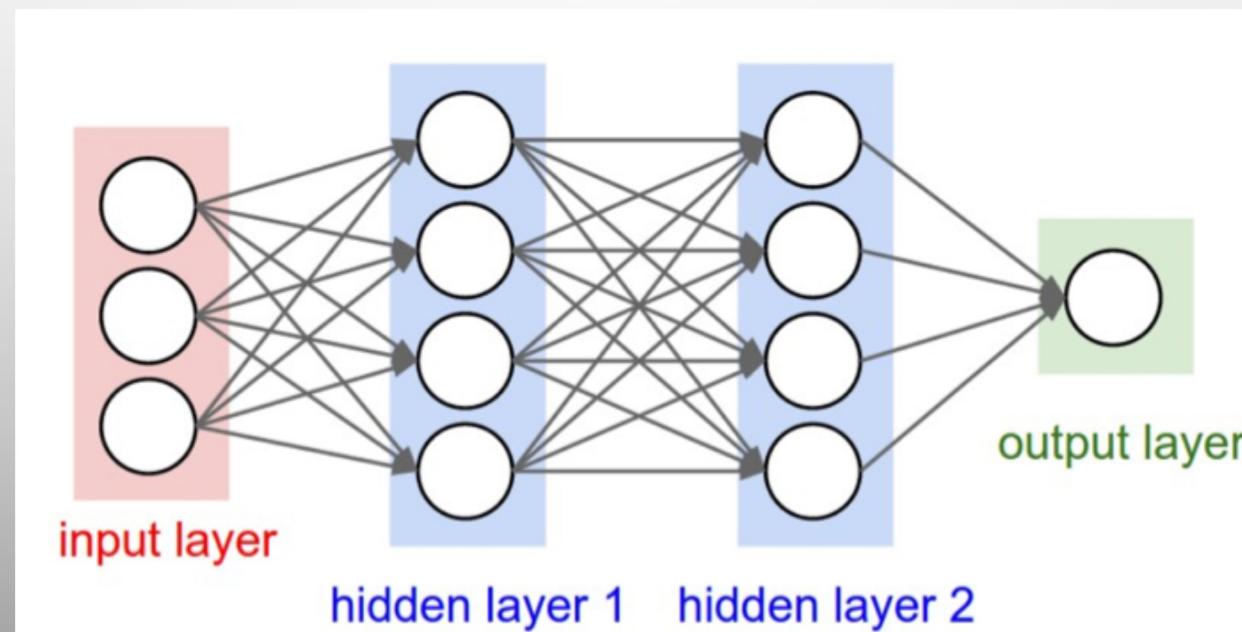
Examples of learned filters





Fully connected layer

- Similar to layers in multilayer perceptrons.



Layer patterns

Layer Patterns

The most common form of a ConvNet architecture stacks a few CONV-RELU layers, follows them with POOL layers, and repeats this pattern until the image has been merged spatially to a small size. At some point, it is common to transition to fully-connected layers. The last fully-connected layer holds the output, such as the class scores. In other words, the most common ConvNet architecture follows the pattern:

```
INPUT -> [[CONV -> RELU]*N -> POOL?] *M -> [FC -> RELU]*K -> FC
```