CS9840

Learning and Computer Vision

Prof. Olga Veksler

Lecture 10

Neural Networks

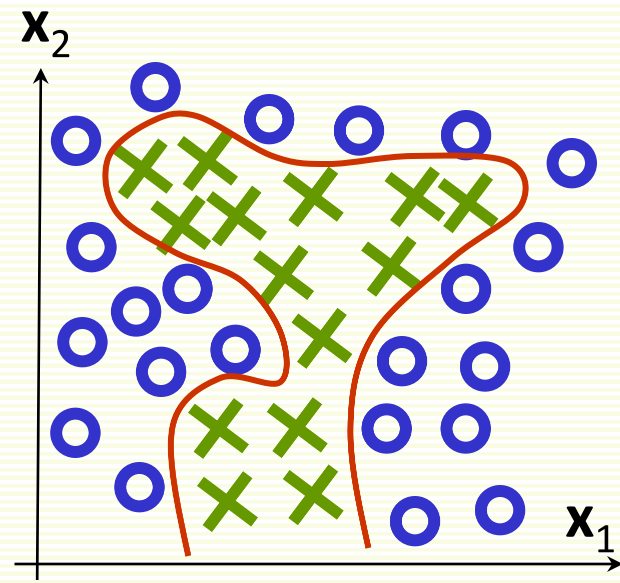
Many slides are from A. Ng, Y. LeCun, G. Hinton, A. Ranzato

Outline

- Intro/History
- Perceptron (1 layer NN)
- Multilayer Perceptron (MLP)
- Deep Networks (DNN)
  - convolutional Network
- Training Deep Network
  - stacked autoencoders

Neural Networks

- Neural Networks correspond to some classifier function $\mathbf{f}_{\mathbf{NN}}(\mathbf{x})$
- Can carve out arbitrarily complex decision boundaries without requiring as many terms as polynomial functions
- Originally inspired by research in how human brain works
  - but cannot claim that this is how the brain actually works
- Now very successful in practice, but took a while to get there



ANN History: First Successes

- 1958, F. Rosenblatt, Cornell University
    - perceptron, oldest neural network still in use today
        - that's what we studied in lecture on linear classifiers
    - Algorithm to train the perceptron network
    - Built in hardware
    - Proved convergence in linearly separable case
    - Early success lead to a lot of claims which were not fulfilled
    - New York Times reports that perceptron is "*the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence*."

ANN History: Stagnation

- 1969, M. Minsky and S. Pappert
  - Book “Perceptrons”
  - Proved that perceptrons can learn only linearly separable classes
  - In particular cannot learn very simple XOR function
  - Conjectured that multilayer neural networks also limited by linearly separable functions
- No funding and almost no research (at least in North America) in 1970’s as the result of 2 things above

ANN History: Revival & Stagnation (Again)

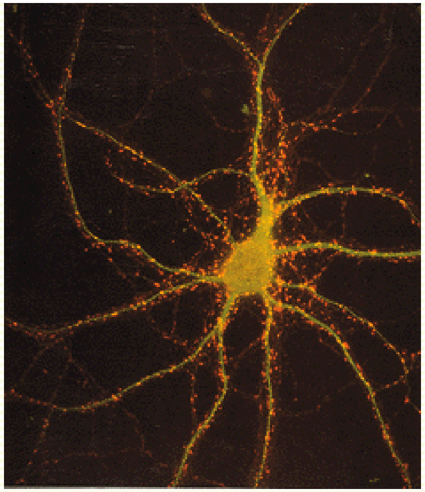
- Revival of ANN in early 1980
- 1986, (re)discovery of backpropagation algorithm by Werbos, Rumelhart, Hinton and Ronald Williams
  - Allows training a MLP
- Many examples of mulitlayer Neural Networks appear
- 1998, Convolutional network (convnet) by Y. Lecun for digit recognition, very successful
- 1990's: research in NN move slowly again
  - Networks with multiple layers are hard to train well (except convnet for digit recognition)
  - SVM becomes popular, works better

ANN History: Deep Learning Age

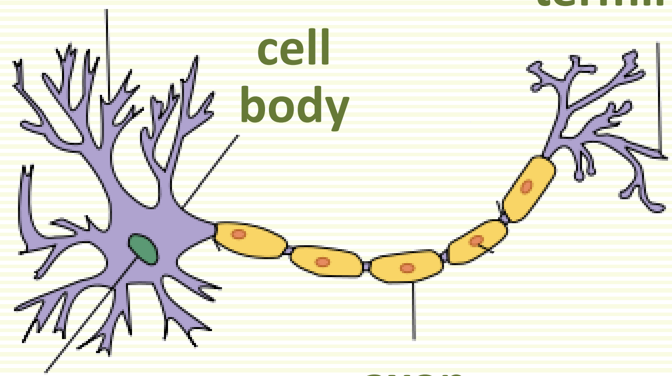
- Deep networks are inspired by brain architecture
- Until now, no success at training them, except convnet
- 2006-now: deep networks are trained successfully
  - massive training data becomes available
  - better hardware: fast training on GPU
  - better training algorithms for network training when there are many hidden layers
    - unsupervised learning of features, helps when training data is limited
- Break through papers
  - Hinton, G. E, Osindero, S., and Teh, Y. W. (2006). A fast learning algorithm for deep belief nets. Neural Computation, 18:1527-1554.
  - Bengio, Y., Lamblin, P., Popovici, P., Larochelle, H. (2007). Greedy Layer-Wise Training of Deep Networks, Advances in Neural Information Processing Systems 19
- Industry: Facebook, Google, Microsoft, etc.

Neuron: Basic Brain Processor

- Neurons (or nerve cells) are special cells that process and transmit information by electrical signaling
  - in brain and also spinal cord
- Human brain has around 10^{11} neurons
- A neuron connects to other neurons to form a network
- Each neuron cell communicates to anywhere from 1000 to 10,000 other neurons

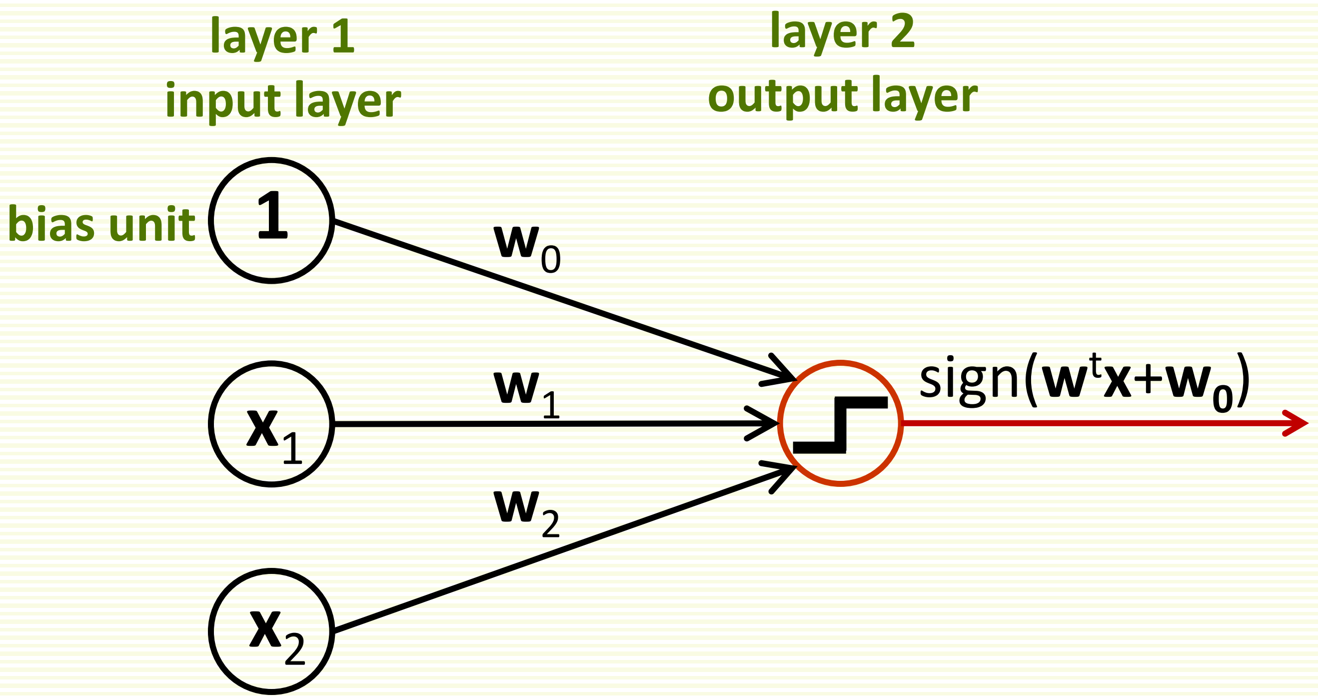


Neuron: Main Components



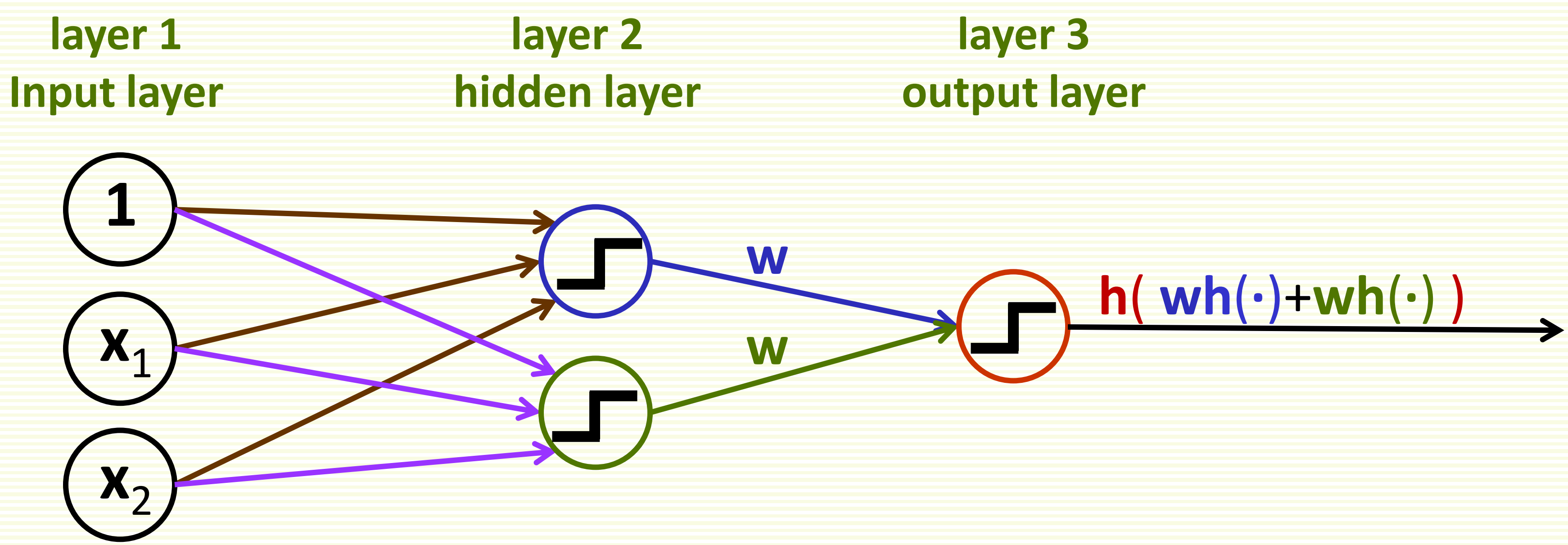
- **cell body**
  - computational unit
- **dendrites**
  - “input wires”, receive inputs from other neurons
  - a neuron may have thousands of dendrites, usually short
- **axon**
  - “output wire”, sends signal to other neurons
  - single long structure (up to 1 meter)
  - splits in possibly thousands branches at the end, “axon terminals”

Perceptron: 1 Layer Neural Network (NN)



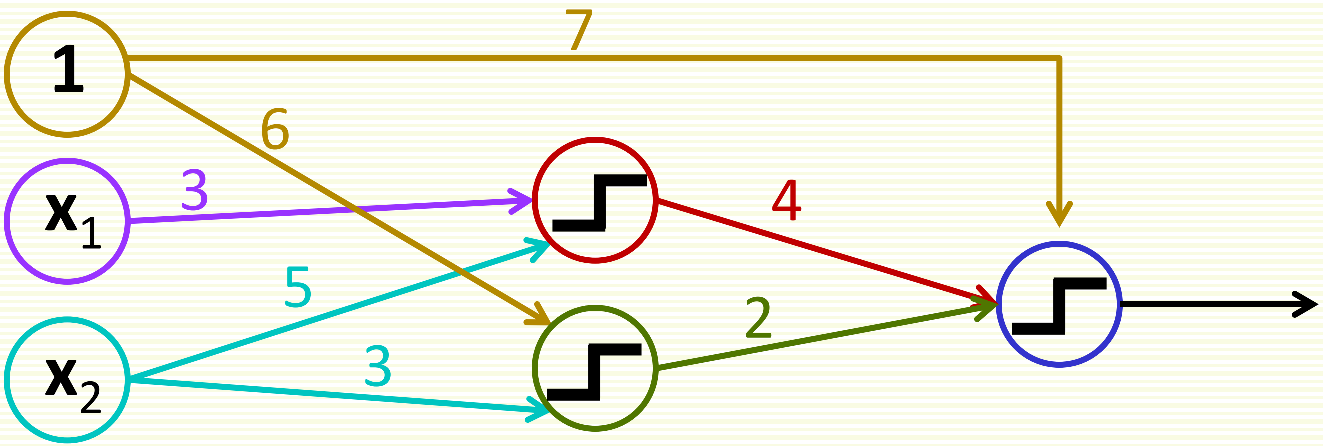
- Linear classifier $\mathbf{f}(\mathbf{x})$ = sign($\mathbf{w}^t\mathbf{x}+\mathbf{w_0}$) is a single neuron "net"
- Input layer units emits features, except bias emits "1"
- Output layer unit applies **h**(t) = **sign**(t)
- **h**(t) is also called an *activation function*

Multilayer Perceptron (MLP)



- First hidden unit outputs $h(w_0 + w_1 x_1 + w_2 x_2)$
- Second hidden unit outputs $h(w_0 + w_1 x_1 + w_2 x_2)$
- Network implements classifier $f(\mathbf{x}) = h(wh(\cdot) + wh(\cdot))$
- More complex boundaries than Perceptron

MLP Small Example



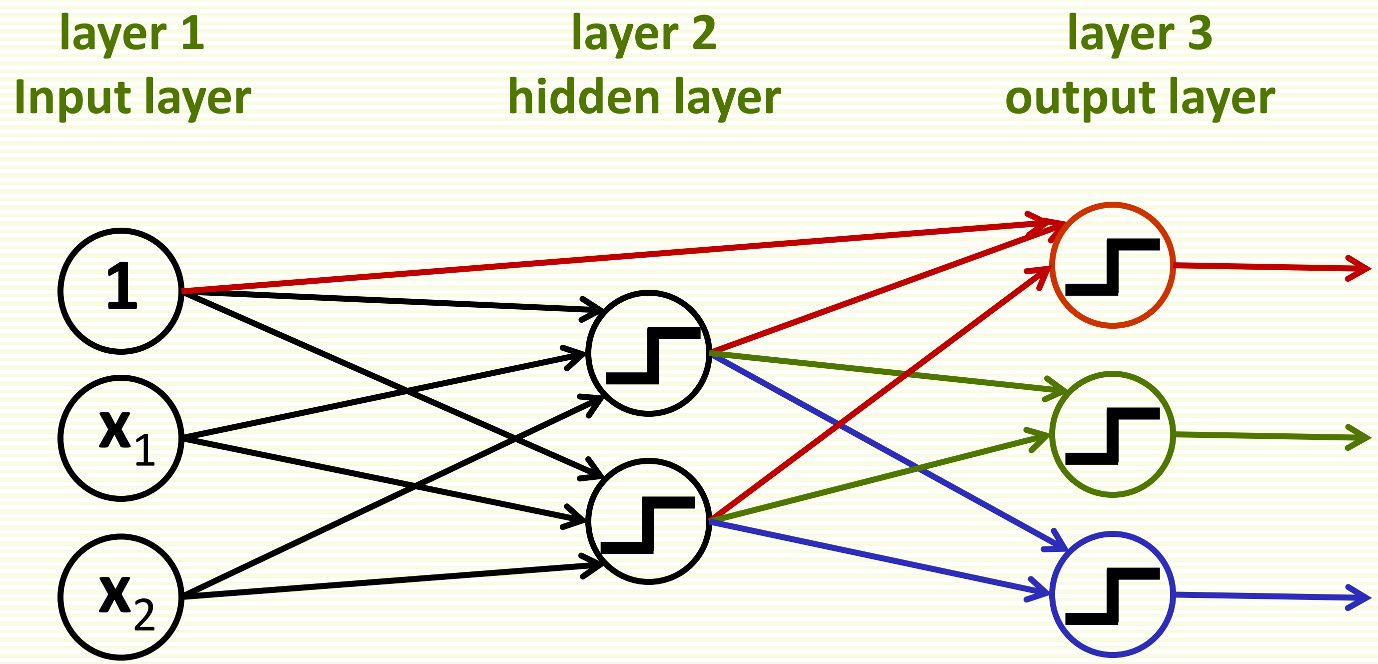
- Implements classifier

$$\mathbf{f}(\mathbf{x}) = \text{sign}(\ 4\mathbf{h}(\cdot)+2\mathbf{h}(\cdot) + 7\ )$$

$$= \text{sign}(4\ \text{sign}(3\mathbf{x}_1+5\mathbf{x}_2)+2\ \text{sign}(6+3\mathbf{x}_2) + 7)$$

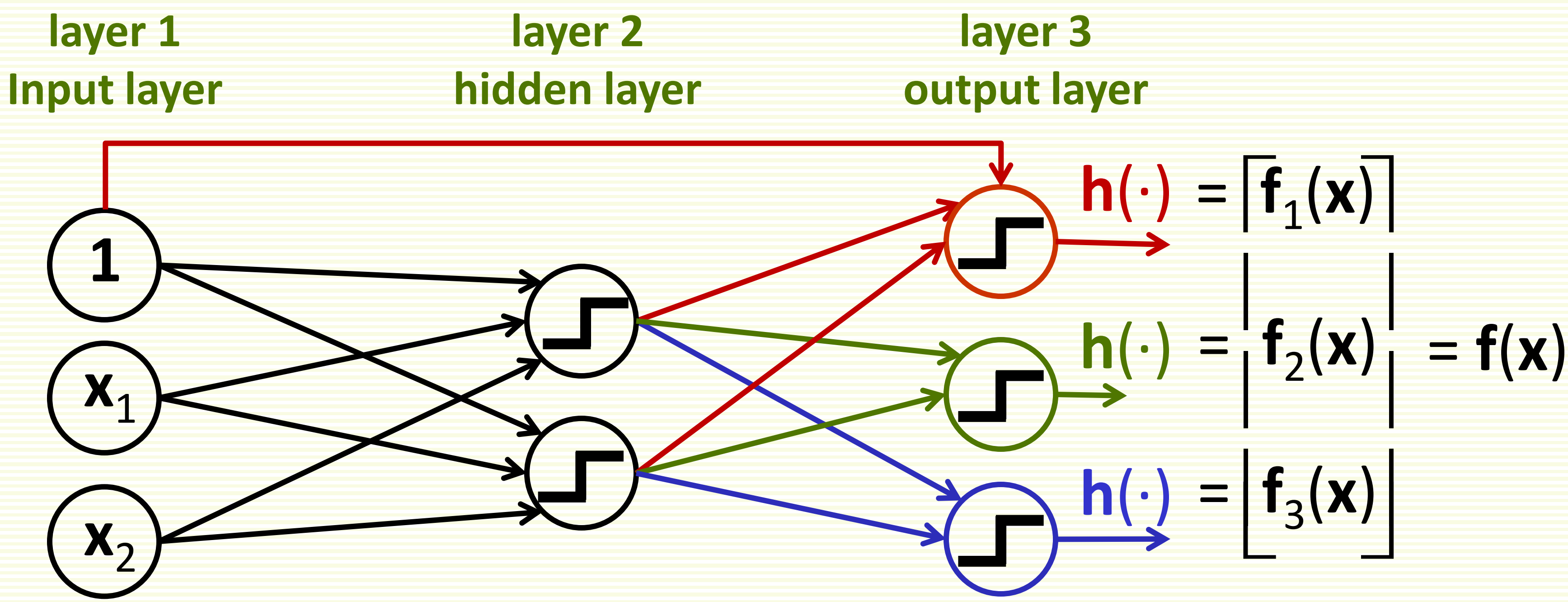
- Computing **f**(**x**) is called *feed forward operation*
  - graphically, function is computed from left to right
- Edge weights are learned through training

MLP: Multiple Classes



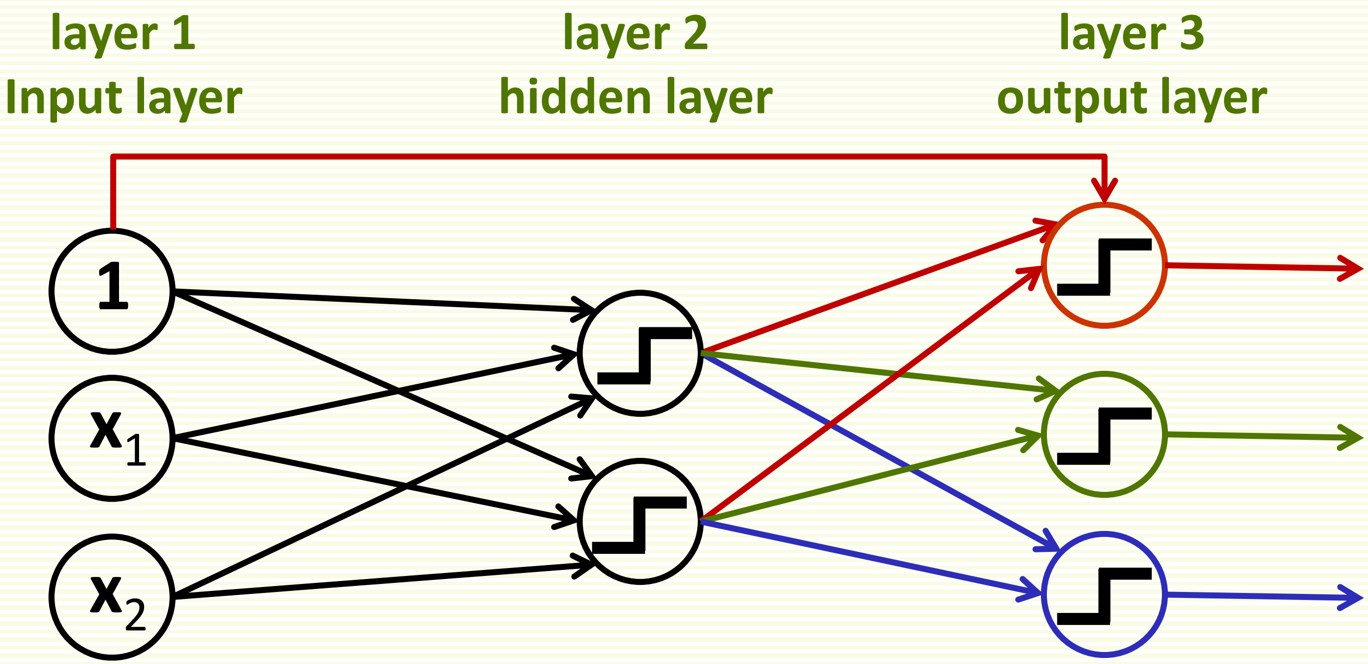
- 3 classes, 2 features, 1 hidden layer
  - 3 input units, one for each feature
  - 3 output units, one for each class
  - 2 hidden units
  - 1 bias unit, can draw in layer 1, or each layer has one

MLP: General Structure



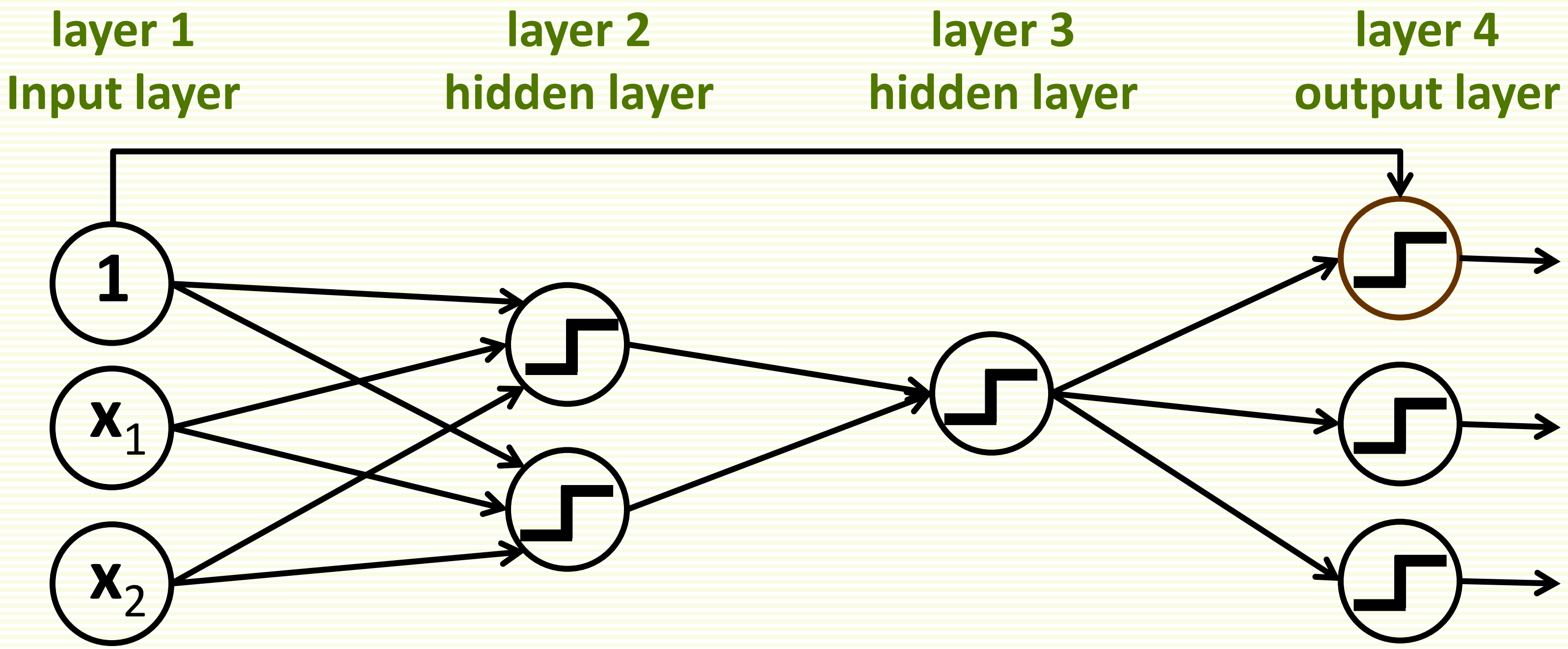
- $\mathbf{f}(\mathbf{x})$ is multi-dimensional
- Classification:
  - If $\mathbf{f}_1(\mathbf{x})$ is largest, decide class 1
  - If $\mathbf{f}_2(\mathbf{x})$ is largest, decide class 2
  - If $\mathbf{f}_3(\mathbf{x})$ is largest, decide class 3

MLP: General Structure



- Input layer: **d** features, **d** input units
- Output layer: **m** classes, **m** output units
- Hidden layer: how many units?
  - more units correspond to more complex classifiers

MLP: General Structure



- Can have many hidden layers
- *Feed forward* structure
  - **i**th layer connects to (**i+1**)th layer
  - except bias unit can connect to any layer
  - or, alternatively each layer can have its own bias unit

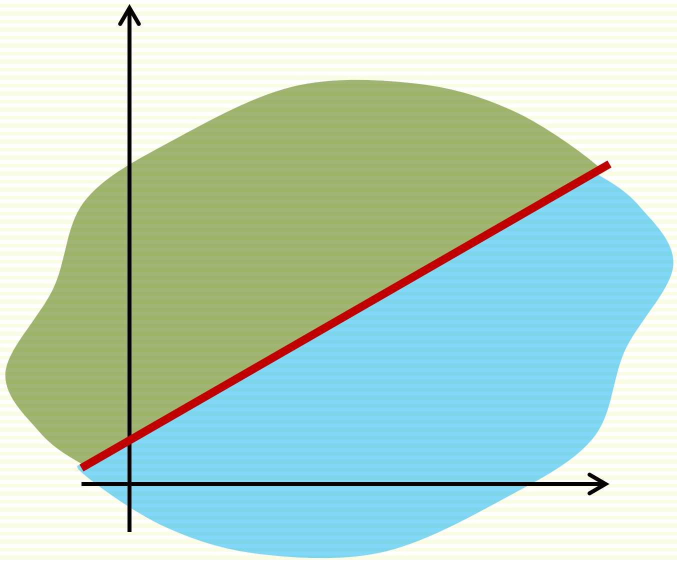
MLP: Overview

- MLP corresponds to rather complex classifier **f**(**x**,**w**)
  - complexity depends on the number of hidden layers/units
  - f(x,w) is a composition of many functions
    - easier to visualize as a network
    - notation gets ugly
- To train MLP, just as before
  - formulate an objective or *loss* function **L**(**w**)
  - optimize it with gradient descent
    - lots of notation due to gradient complexity
    - lots of tricks to get gradient descent work reasonably well

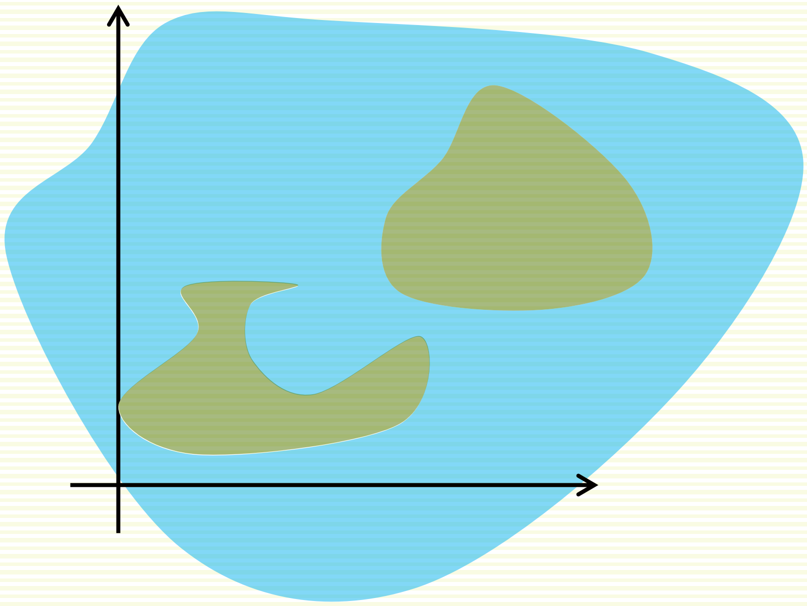
Expressive Power of MLP

- Every continuous function from input to output can be implemented with enough hidden units, 1 hidden layer, and proper *nonlinear* activation functions
  - easy to show that with linear activation function, multilayer neural network is equivalent to perceptron
- This is more of theoretical than practical interest
  - proof is not constructive (does not tell how construct MLP)
  - even if constructive, would be of no use, we do not know the desired function, our goal is to learn it through the samples
  - but this result gives confidence that we are on the right track
    - MLP is general (expressive) enough to construct any required decision boundaries, unlike the Perceptron

Decision Boundaries



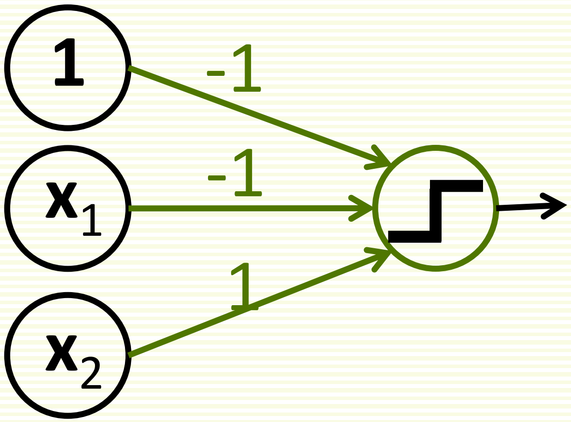
- Perceptron (single layer neural net)

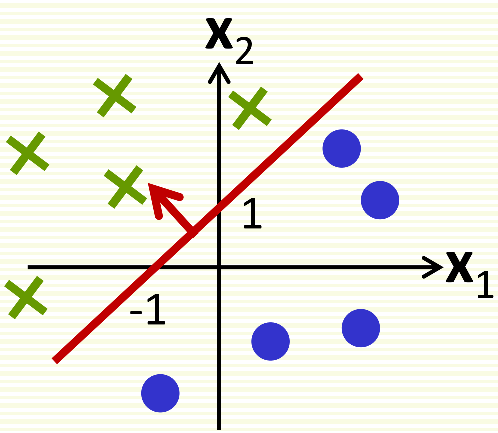


- Arbitrarily complex decision regions
- Even not contiguous

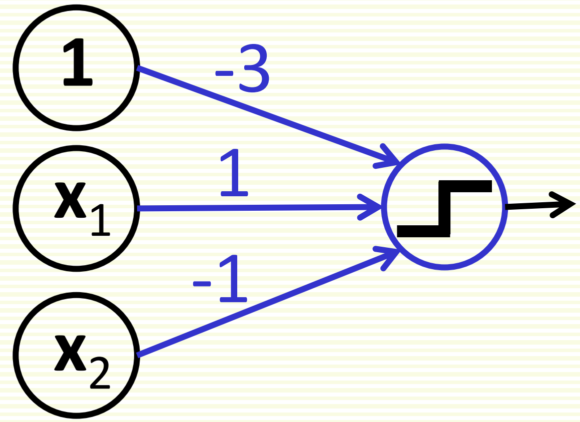
Nonlinear Decision Boundary: Example

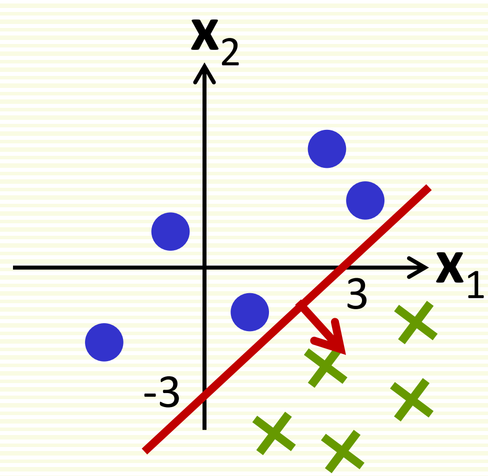
$-\mathbf{x}_1 + \mathbf{x}_2 - 1 > 0 \Rightarrow \text{class } 1$





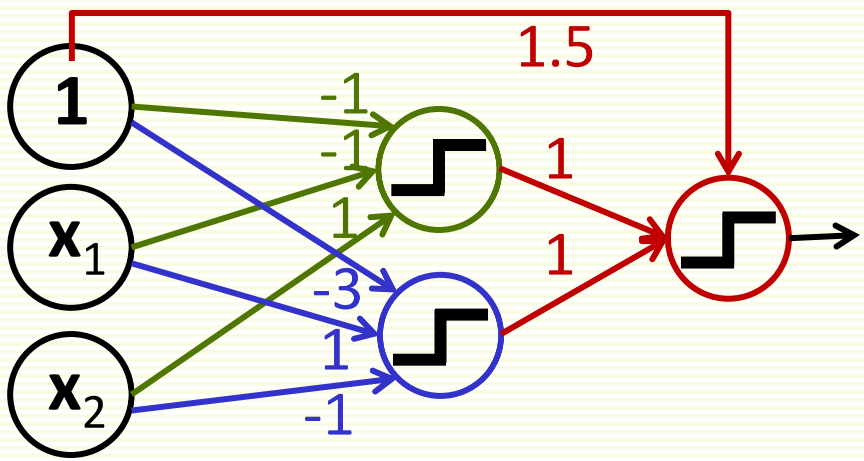
$\mathbf{x}_1 - \mathbf{x}_2 - 3 > 0 \Rightarrow \text{class } 1$

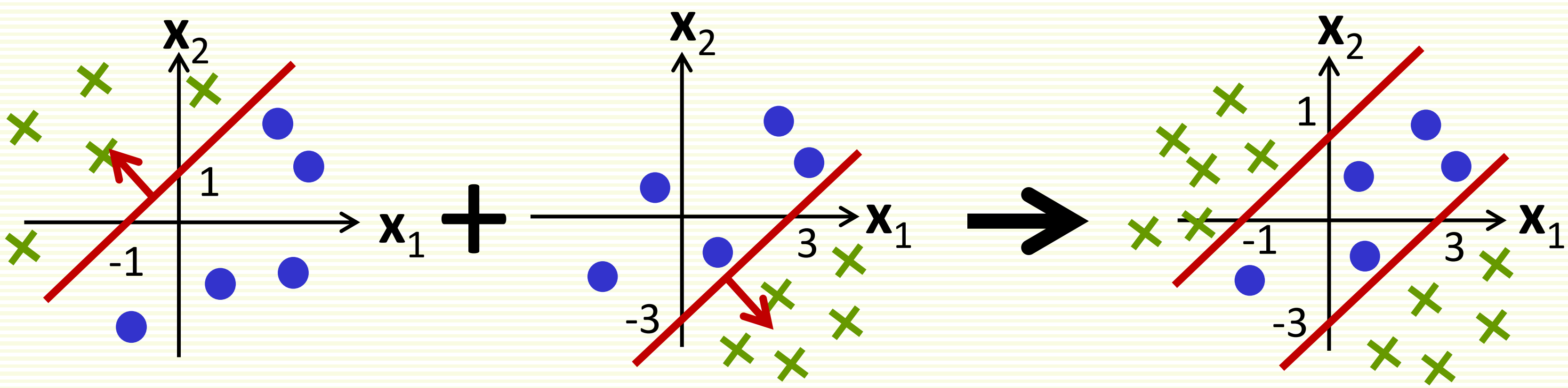




Nonlinear Decision Boundary: Example

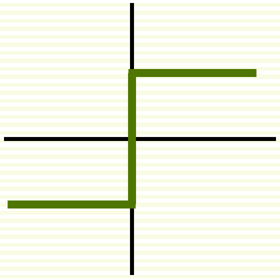
- Combine two Perceptrons into a 3 layer NN



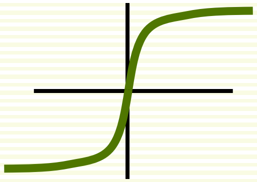


Multi-Layer Neural Networks: Activation Function

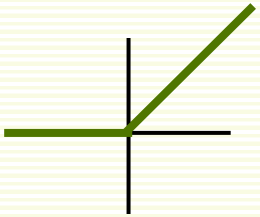
- **h**() = **sign**() does not work for gradient descent

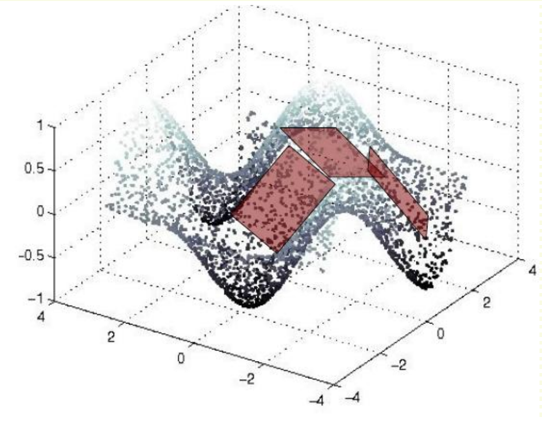


- Can use sigmoid function



- Rectified Linear (RuLu) popular recently
  - constructs locally linear function





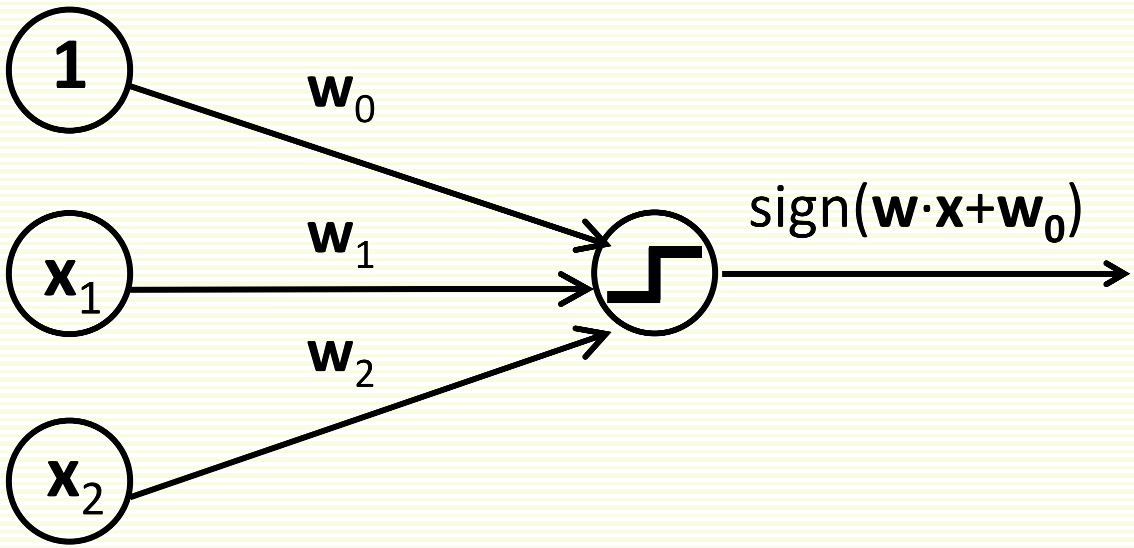
NN: Modes of Operation

- Due to historical reasons, training and testing stages have special names
  - **Backpropagation (or training)**

    Minimize objective function with gradient descent
  - **Feedforward (or testing)**

NN: Vector Notation

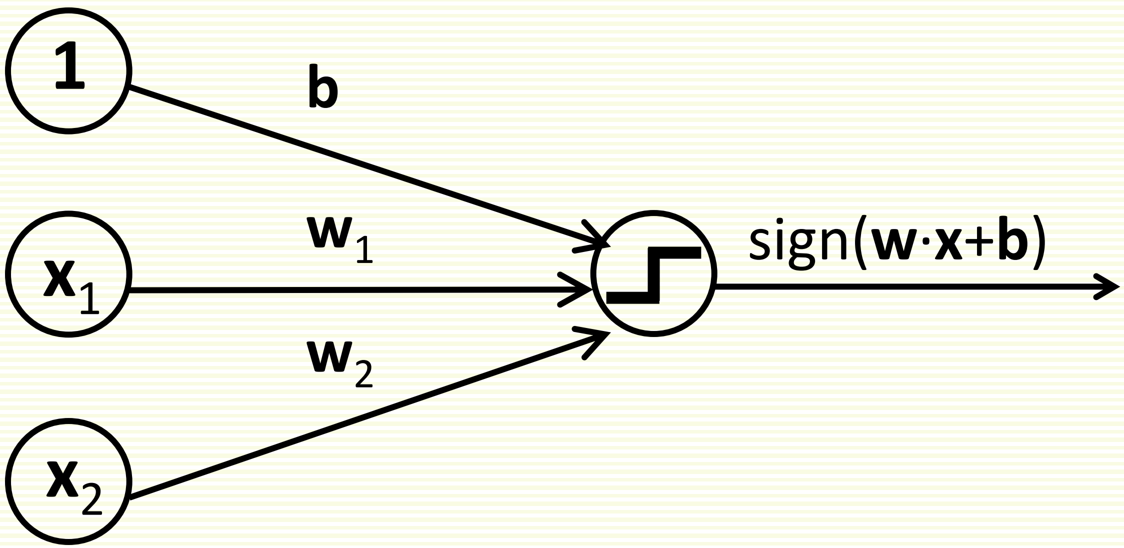
- Want more compact (vector) notation
- Compact notation for Perceptron



$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \qquad \mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$$

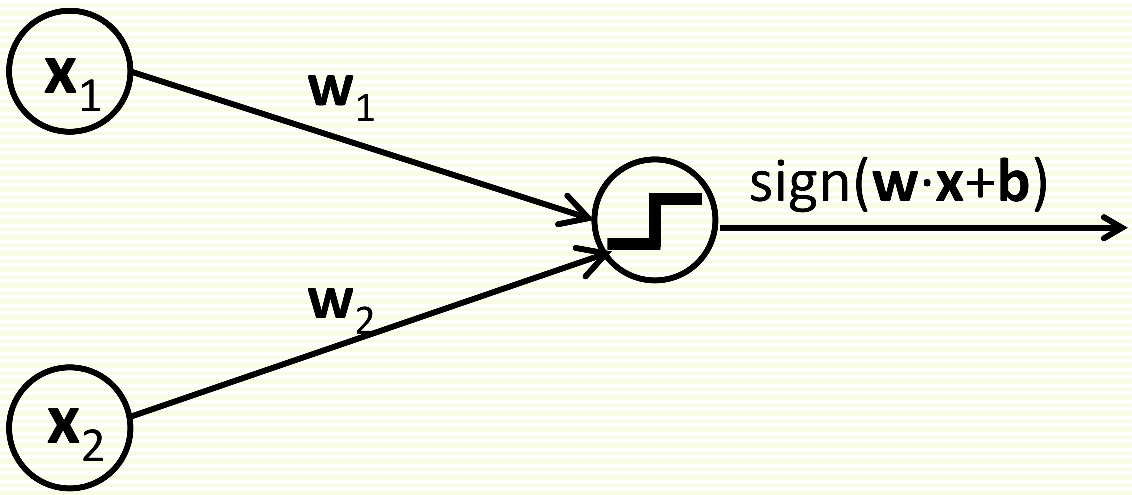
NN: Vector Notation

- Change notation a bit

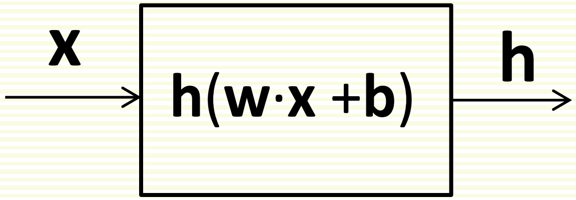


NN: Vector Notation

- Do not draw bias unit

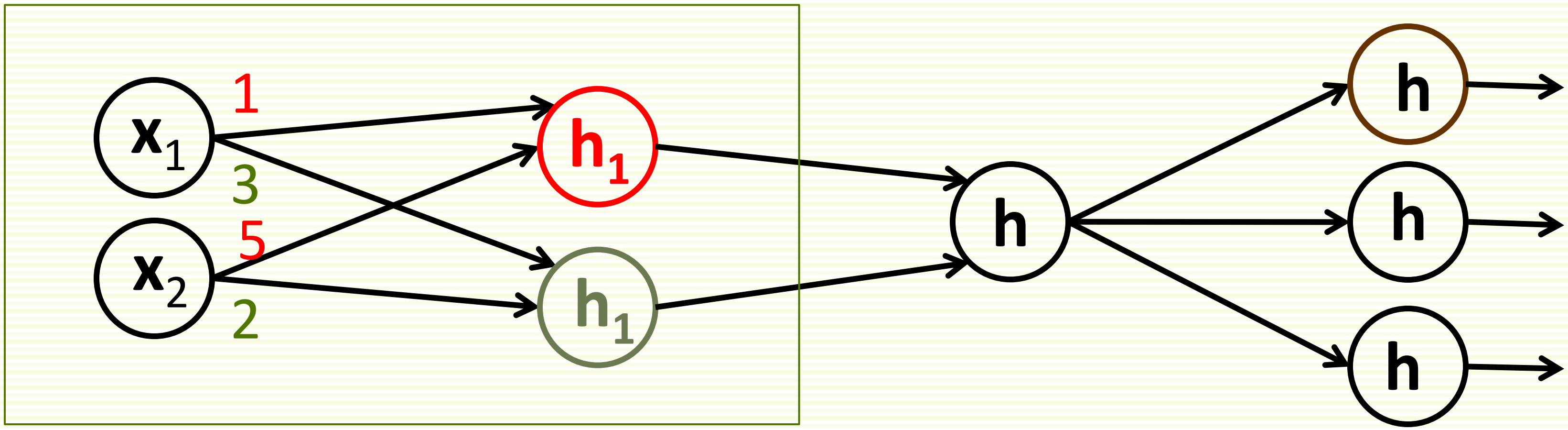


- Compact picture
  - $\mathbf{h}$(t) = sign(t)

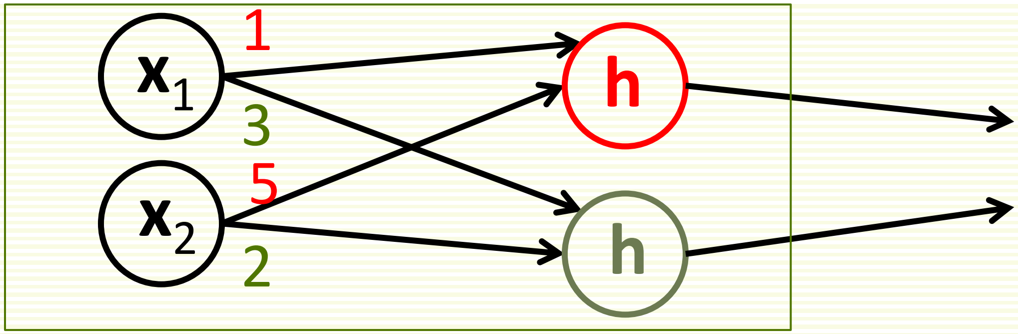


NN: Vector Notation

- For now, look just at the first layer (2 perceptrons)



NN: Vector Notation, First Layer

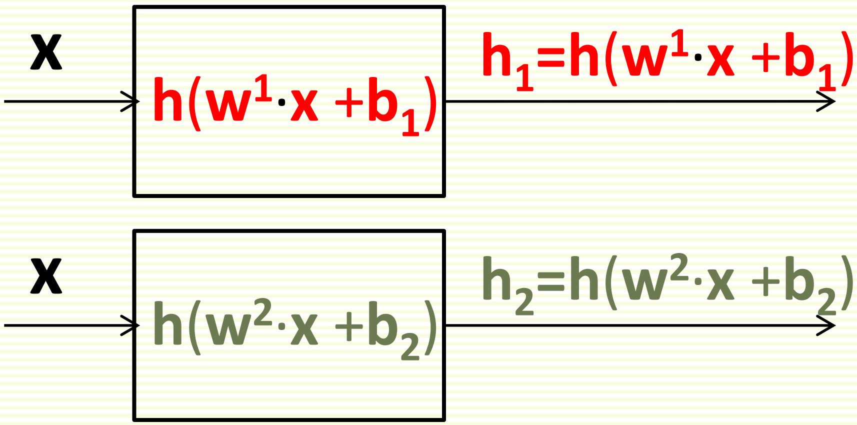


- Red perceptron has weights w^1 and bias b_1
- Green perceptron has weights w^2 and bias b_2

$$\mathbf{x} \longrightarrow \boxed{h(\mathbf{w}^1 \cdot \mathbf{x} + \mathbf{b}_1)} \xrightarrow{h_1 = h(\mathbf{w}^1 \cdot \mathbf{x} + \mathbf{b}_1)} \qquad \mathbf{w}^1 = \begin{bmatrix} 1 \\ 5 \end{bmatrix}$$

$$\mathbf{x} \longrightarrow \boxed{h(\mathbf{w}^2 \cdot \mathbf{x} + \mathbf{b}_2)} \xrightarrow{h_2 = h(\mathbf{w}^2 \cdot \mathbf{x} + \mathbf{b}_2)} \qquad \mathbf{w}^2 = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

NN: Vector Notation , First Layer

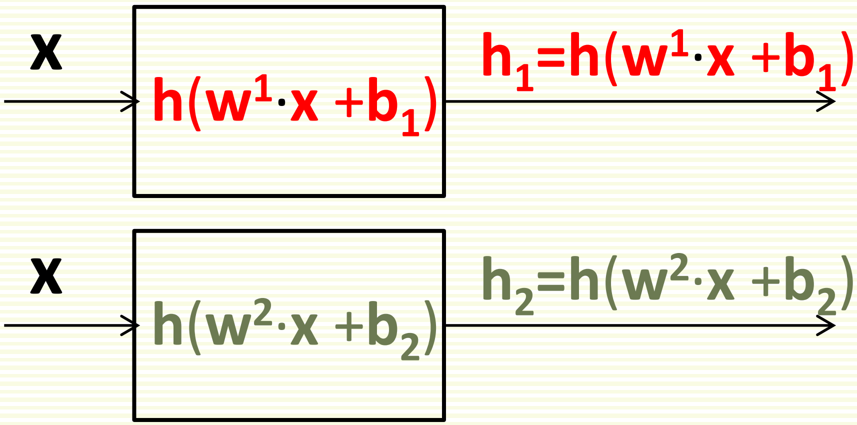


$$\begin{bmatrix} 1 & 5 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} w^1 \cdot x \\ w^2 \cdot x \end{bmatrix}$$

$$\mathbf{W}^1 \cdot \mathbf{x}$$

$$\mathbf{w}^1 = \begin{bmatrix} 1 \\ 5 \end{bmatrix} \quad \mathbf{w}^2 = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

NN: Vector Notation , First Layer

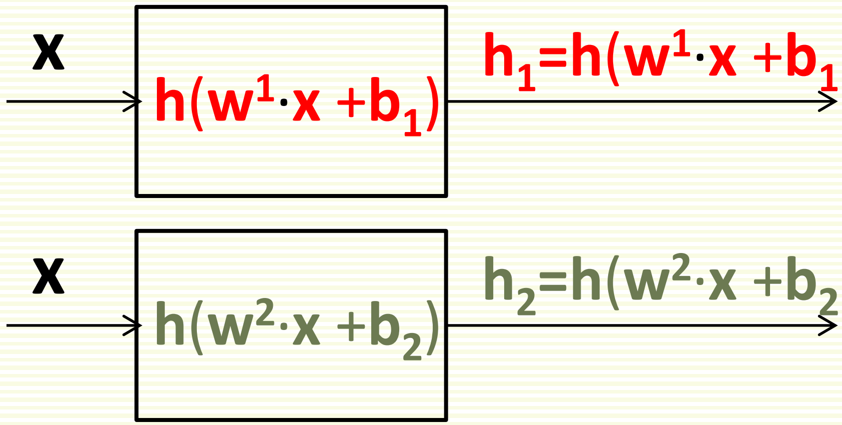


$$\begin{bmatrix}1 & 5\\ 3 & 2\end{bmatrix}\begin{bmatrix}x_1\\ x_2\end{bmatrix}+\begin{bmatrix}b_1\\ b_2\end{bmatrix}=\begin{bmatrix}w^1\cdot x+b_1\\ w^2\cdot x+b_2\end{bmatrix}$$

$$\mathbf{W}^1 \cdot \mathbf{x} + \mathbf{b}^1$$

$$\mathbf{w}^1 = \begin{bmatrix}1\\ 5\end{bmatrix} \quad \mathbf{w}^2 = \begin{bmatrix}3\\ 2\end{bmatrix}$$

NN: Vector Notation , First Layer



$$\mathbf{h}\left(\begin{bmatrix} 1 & 5 \\ 3 & 2 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}\right) = \begin{bmatrix} h_1 \\ h_2 \end{bmatrix}$$

$$\mathbf{h}(\mathbf{W}^1 \cdot \mathbf{x} + \mathbf{b}^1)$$

$$\mathbf{w}^1 = \begin{bmatrix} 1 \\ 5 \end{bmatrix} \quad \mathbf{w}^2 = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

- **h**(**v**) for vector **v** – apply **h** to each component of **v**

NN: Vector Notation , First Layer

$\mathbf{x} \rightarrow \boxed{h(w^1 \cdot x + b_1)} \rightarrow h_1 = h(w^1 \cdot x + b_1)$

$\mathbf{x} \rightarrow \boxed{h(w^2 \cdot x + b_2)} \rightarrow h_2 = h(w^2 \cdot x + b_2)$

more compact $\Rightarrow$

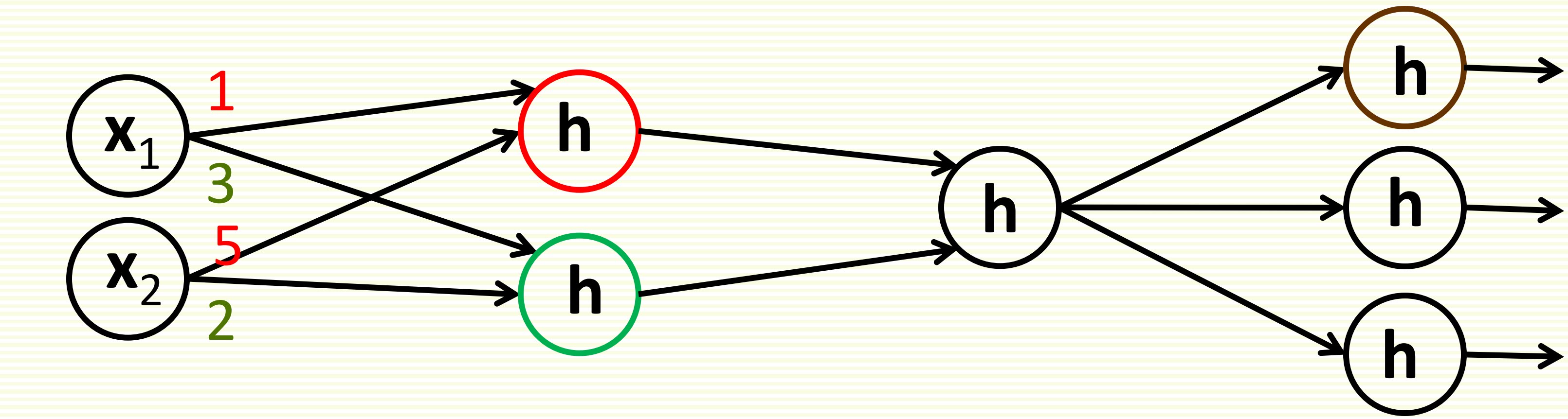
$\mathbf{x} \rightarrow \boxed{\mathbf{h}(\mathbf{W}^1 \mathbf{x} + \mathbf{b}^1)} \rightarrow \mathbf{h}^1$

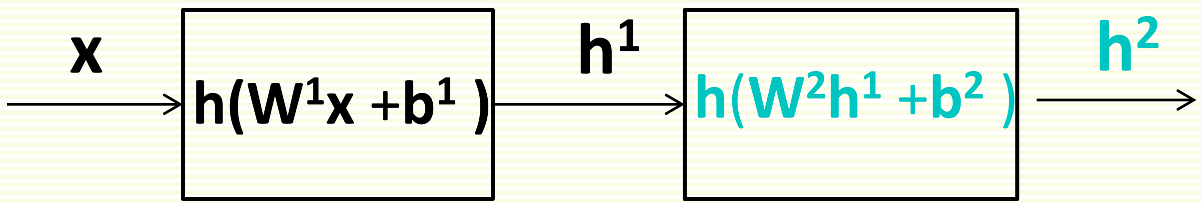
$$\mathbf{W}^1 = \begin{bmatrix} 1 & 5 \\ 3 & 2 \end{bmatrix} \quad \mathbf{b}^1 = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

$$\mathbf{h}^1 = \mathbf{h}(\mathbf{W}^1 \mathbf{x} + \mathbf{b}^1) = \begin{bmatrix} h_1 \\ h_2 \end{bmatrix}$$

- **h**(**v**) for vector **v** – apply **h** to each component of **v**

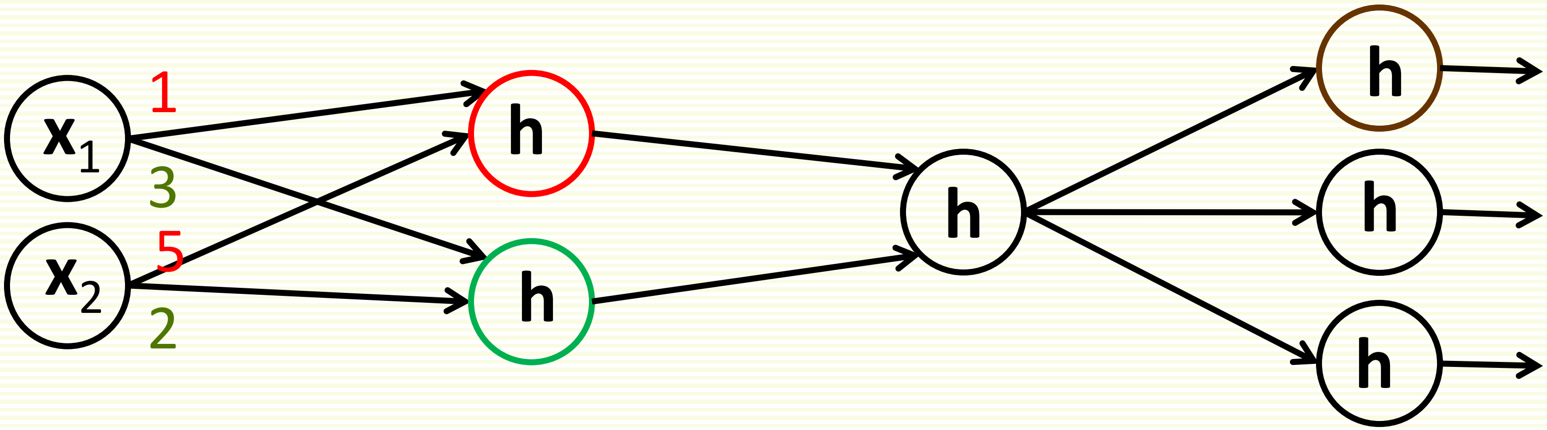
NN: Vector Notation, Next Layer



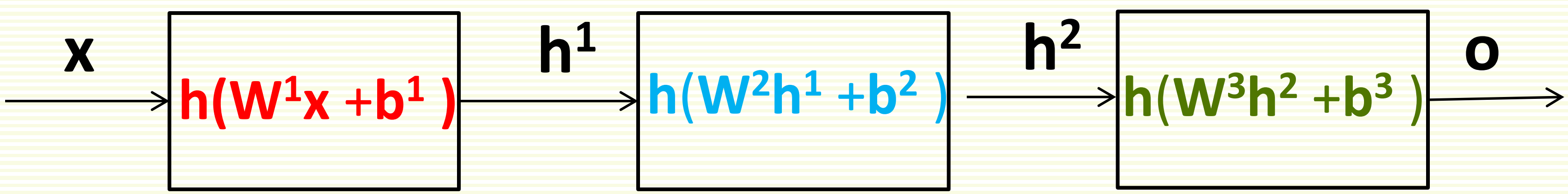


- $\mathbf{W}^2$ is a matrix of weights between hidden layer 1 and 2
  - $\mathbf{W}^2(\mathbf{r},\mathbf{c})$ is weight from unit **c** to unit **r**
- $\mathbf{b}^2$ is a vector of bias weights for second hidden layer
  - b^2_r is bias weight of unit **r** in second layer
- $\mathbf{h}^2$ is a vector of second layer outputs
  - h^2_r is output of unit **r** in second layer

NN: Vector Notation, all Layers



- Complete depiction



- **o** vector from the output layer
- $\mathbf{o} = \mathbf{h(W^3h^2 + b^3)}$

  $= \mathbf{h(W^3h(W^2h^1 + b^2) + b^3)}$

  $= \mathbf{h(W^3h(W^2h(W^1x + b^1) + b^2) + b^3)}$

NN: Output Representation

- Output of NN is a vector
- So label $\mathbf{y}^i$ of sample $\mathbf{x}^i$ should also be a vector
- Let $\mathbf{x}^i$ be sample of class $\mathbf{k}$

$$\mathbf{y}^i = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \begin{array}{l} \\ \\ \longleftarrow \text{row } \mathbf{k} \\ \\ \\ \end{array}$$

- Want output unit $\mathbf{o}_k = 1$
- Want other output units zero

$$\mathbf{f}(\mathbf{x}^i) = \mathbf{o} = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \begin{array}{l} \\ \\ \longleftarrow \text{row } \mathbf{k} \\ \\ \\ \end{array}$$

Training NN: Loss Function

- Want to minimize difference between $\mathbf{y}^i$ and $\mathbf{f}(\mathbf{x}^i)$
- Let **W** be all edge weights
- With squared difference **loss** (error)
- Loss on one example $\mathbf{x}^i$:

$$L\left(\mathbf{x}^i, \mathbf{y}^i; \mathbf{W}\right) = \left\| \mathbf{f}\left(\mathbf{x}^i\right) - \mathbf{y}^i \right\|^2 = \sum_{j=1}^{m} \left(\mathbf{f}_j\left(\mathbf{x}^i\right) - \mathbf{y}_j^i \right)^2$$

$$\mathbf{f}(\mathbf{x}) = \mathbf{o} = \begin{bmatrix} 0.5 \\ \vdots \\ 0.9 \\ \vdots \\ 0.3 \end{bmatrix} \qquad \mathbf{y}^i = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \begin{array}{l} \\ \\ \longleftarrow \text{row } \mathbf{k} \\ \\ \\ \end{array}$$

- **f** depends on **W**, but too cumbersome to write **f**(**x**,**W**) everywhere

Training NN: Loss Function

- Let $\mathbf{X} = \mathbf{x}^1, \ldots, \mathbf{x}^n$

  $\mathbf{Y} = \mathbf{y}^1, \ldots, \mathbf{y}^n$

- Loss on all examples: $\mathbf{L}(\mathbf{X}, \mathbf{Y}; \mathbf{W}) = \sum_{\mathbf{i}=1}^{\mathbf{n}} \left\| \mathbf{f}(\mathbf{x}^{\mathbf{i}}) - \mathbf{y}^{\mathbf{i}} \right\|^2$

- Gradient descent

initialize **w** to random
choose **ε**, α
while $\alpha\|\nabla \mathbf{L}(\mathbf{X},\mathbf{Y};\mathbf{W})\| > \boldsymbol{\varepsilon}$
    $\mathbf{w} = \mathbf{w} - \alpha \nabla \mathbf{L}(\mathbf{X},\mathbf{Y};\mathbf{W})$

Training NN: Cross Entropy Loss Function

- Cross entropy loss works well for classification

- First put the output **o** through soft-max

$$\mathbf{f}_k(\mathbf{x}) = \frac{\exp(\mathbf{o}_k)}{\sum_{j=1}^{m} \exp(\mathbf{o}_j)}$$

$$\mathbf{o} = \begin{bmatrix} 0.6 \\ -1 \\ 5 \\ 8 \\ 4 \end{bmatrix} \Rightarrow \begin{bmatrix} 0.006 \\ 0.0001 \\ 0.047 \\ 0.94 \\ 0.17 \end{bmatrix} = \mathbf{f}(\mathbf{x}) = \text{sofmax}(\mathbf{o})$$

- Interpret $\mathbf{f}_k(\mathbf{x})$ as probability of class **k**

Training NN: Cross Entropy Loss Function

- One sample cross entropy loss, dropping superscripts from $\mathbf{x}^i, \mathbf{y}^i$:

$$\mathbf{L}(\mathbf{x}, \mathbf{y}; \mathbf{W}) = -\sum_j \mathbf{y}_j \log \mathbf{f}_j(\mathbf{x})$$

- If sample $\mathbf{x}$ is of class k, then the above is equivalent to

$$\mathbf{L}(\mathbf{x}, \mathbf{y}; \mathbf{W}) = -\log \mathbf{f}_k(\mathbf{x})$$

  - minimizing **–log** is equivalent to maximizing probability

- Loss on all samples

$$\mathbf{L}(\mathbf{X}, \mathbf{Y}; \mathbf{W}) = \sum \mathbf{L}(\mathbf{x}, \mathbf{y}; \mathbf{W})$$

Training NN: -Log Loss Function

- Need to find derivative of **L** wrt every network weight $\mathbf{w}_i$

$$\frac{\partial \mathbf{L}}{\partial \mathbf{w}_i}$$

- After derivative found, according to gradient descent, weight update is:

$$\Delta \mathbf{w}_i = -\alpha \frac{\partial \mathbf{L}}{\partial \mathbf{w}_i}$$

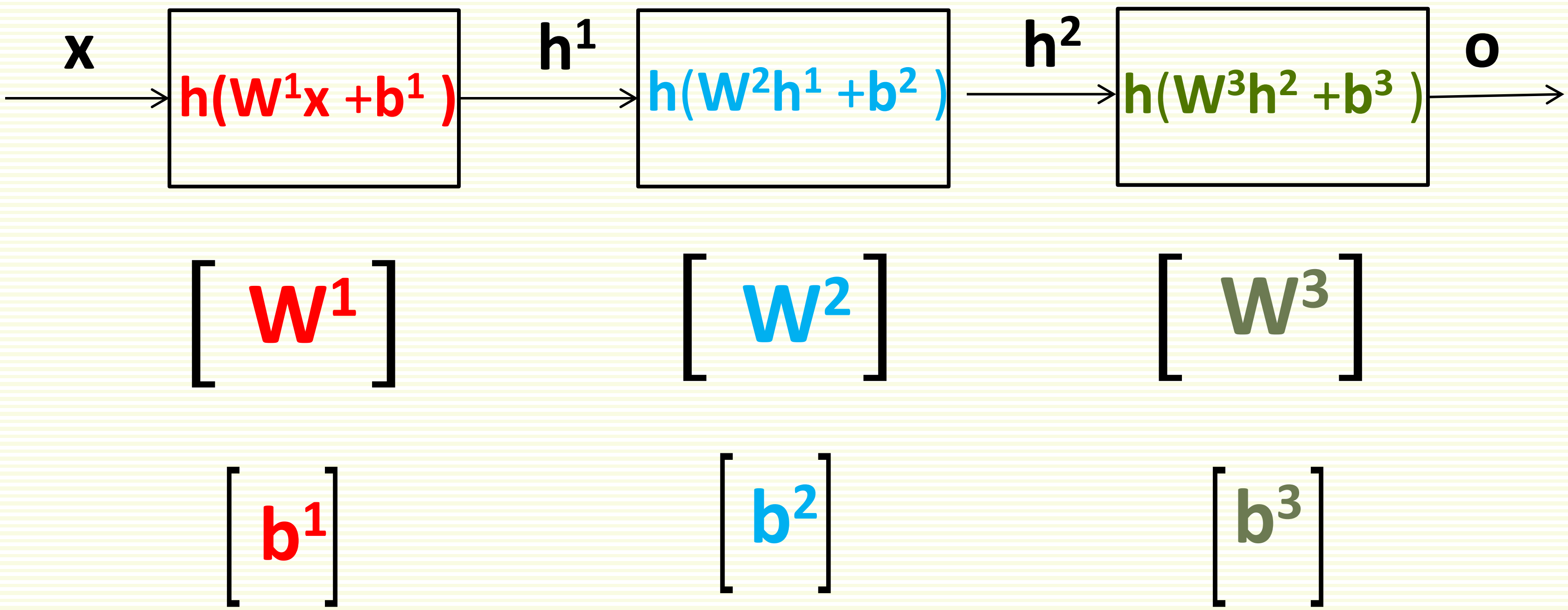
  - where α is the learning rate

- Update weight:

$$\mathbf{w}_i = \mathbf{w}_i + \Delta \mathbf{w}_i$$

Training NN: -Log Loss Function

- How many weights do we have in our network?



- Weights are in matrices $\mathbf{W}^1,\mathbf{W}^2,\ldots,\mathbf{W}^l$
- And are in matrices $\mathbf{b}^1,\mathbf{b}^2,\ldots,\mathbf{b}^l$

Training NN: -Log Loss Function

- Consider matrix $\mathbf{W}^1$

$$\mathbf{W}^1 = \begin{bmatrix} \mathbf{w}^1_{11} & \cdots & \mathbf{w}^1_{1\mathbf{k}} \\ \vdots & \cdots & \vdots \\ \mathbf{w}^1_{\mathbf{d}1} & \cdots & \mathbf{w}^1_{\mathbf{dk}} \end{bmatrix}$$

- Need to compute derivative wrt every $\mathbf{w}^1_{\mathbf{js}}$
- Organize derivatives in matrix

$$\frac{\partial \mathbf{L}}{\partial \mathbf{W}^1} = \begin{bmatrix} \frac{\partial \mathbf{L}}{\partial \mathbf{w}^1_{11}} & \cdots & \frac{\partial \mathbf{L}}{\partial \mathbf{w}^1_{1\mathbf{k}}} \\ \vdots & \cdots & \vdots \\ \frac{\partial \mathbf{L}}{\partial \mathbf{w}^1_{\mathbf{d}1}} & \cdots & \frac{\partial \mathbf{L}}{\partial \mathbf{w}^1_{\mathbf{dk}}} \end{bmatrix}$$

Training NN: -Log Loss Function

- Chain rule for derivatives of composed functions:

$$\frac{\partial \mathbf{f}\left(\mathbf{h}(\mathbf{w})\right)}{\partial \mathbf{w}} = \frac{\partial \mathbf{f}}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \mathbf{x}}$$

- NN is a composition of compositions … of compositions of functions **h**(**h**(**h** ()))
- Have to apply the chain rule a lot

Training NN: -Log Loss Function

$$\mathbf{f}_k(\mathbf{x}) = \frac{\exp(\mathbf{o}_k)}{\sum_{j=1}^{m} \exp(\mathbf{o}_j)} \qquad \mathbf{L}(\mathbf{x}, \mathbf{y}; \mathbf{w}) = -\sum_j \mathbf{y}_j \log \mathbf{f}_j(\mathbf{x})$$

- **f (x,W) = f(g(o(W)))**
- So first take derivatives wrt $\mathbf{o}_j$

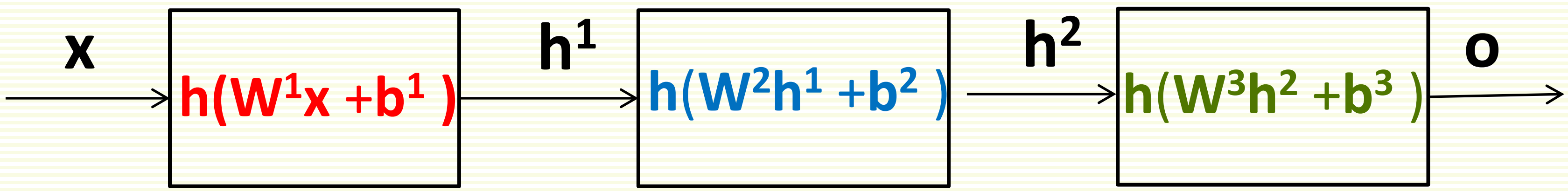
$$\frac{\partial \mathbf{L}}{\partial \mathbf{o}_j} = \mathbf{f}_j(\mathbf{x}) - \mathbf{y}_j$$

- Vector of derivatives wrt **o**

$$\frac{\partial \mathbf{L}}{\partial \mathbf{o}} = \mathbf{f}(\mathbf{x}) - \mathbf{y}$$

$$\frac{\partial \mathbf{L}}{\partial \mathbf{o}} = \begin{bmatrix} \frac{\partial \mathbf{L}}{\partial \mathbf{o}_1} \\ \cdots \\ \frac{\partial \mathbf{L}}{\partial \mathbf{o}_m} \end{bmatrix} = \begin{bmatrix} \mathbf{f}_1(\mathbf{x}) - \mathbf{y}_1 \\ \cdots \\ \mathbf{f}_m(\mathbf{x}) - \mathbf{y}_m \end{bmatrix}$$

Training NN: -Log Loss Function



$$\frac{\partial \mathbf{L}}{\partial \mathbf{o}} = \mathbf{f}(\mathbf{x}) - \mathbf{y}$$

- Assume ReLu $\mathbf{h}(\mathbf{z}) = \max(\mathbf{z},\mathbf{0})$

- Compute derivatives “backwards”

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

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

Training NN: -Log Loss Function

$$\mathbf{x} \longrightarrow \boxed{h(W^1x + b^1)} \xrightarrow{\mathbf{h}^1} \boxed{h(W^2h^1 + b^2)} \xrightarrow{\mathbf{h}^2} \boxed{h(W^3h^2 + b^3)} \xrightarrow{\mathbf{o}}$$

$$\frac{\partial \mathbf{L}}{\partial \mathbf{o}} = \mathbf{f}(\mathbf{x}) - \mathbf{y}$$

- Assume ReLu **h**(**z**) = max(**z**,**0**)

- Compute derivatives “backwards”

$$\frac{\partial \mathbf{L}}{\partial \mathbf{W}^3} = \frac{\partial \mathbf{L}}{\partial \mathbf{o}} \frac{\partial \mathbf{o}}{\partial \mathbf{W}^3} = (\mathbf{f}(\mathbf{x}) - \mathbf{y})(\mathbf{h}^2)^{\mathsf{T}}$$

$$\frac{\partial \mathbf{L}}{\partial \mathbf{h}^2} = \frac{\partial \mathbf{L}}{\partial \mathbf{o}} \frac{\partial \mathbf{o}}{\partial \mathbf{h}^2} = (\mathbf{W}^3)^{\mathsf{T}} (\mathbf{f}(\mathbf{x}) - \mathbf{y})$$

Training NN: -Log Loss Function

- Sketch of derivation for $\dfrac{\partial \mathbf{L}}{\partial \mathbf{W}^{3}}$

$$\mathbf{W}^{3} = \begin{bmatrix} \mathbf{w}_{11}^{3} & \mathbf{w}_{12}^{3} \\ \mathbf{w}_{21}^{3} & \mathbf{w}_{22}^{3} \end{bmatrix}$$

$$\frac{\partial \mathbf{L}}{\partial \mathbf{W}^{3}} = \begin{bmatrix} \dfrac{\partial \mathbf{L}}{\partial \mathbf{w}_{11}^{3}} & \dfrac{\partial \mathbf{L}}{\partial \mathbf{w}_{12}^{3}} \\ \dfrac{\partial \mathbf{L}}{\partial \mathbf{w}_{21}^{3}} & \dfrac{\partial \mathbf{L}}{\partial \mathbf{w}_{22}^{3}} \end{bmatrix}$$

Training NN: -Log Loss Function

- Recall

$$\mathbf{W}^3 = \begin{bmatrix} \mathbf{w}_{11}^3 & \mathbf{w}_{12}^3 \\ \mathbf{w}_{21}^3 & \mathbf{w}_{22}^3 \end{bmatrix} \quad \mathbf{h}^2 = \begin{bmatrix} \mathbf{h}_1^2 \\ \mathbf{h}_2^2 \end{bmatrix} \quad \mathbf{b}^2 = \begin{bmatrix} \mathbf{b}_1^2 \\ \mathbf{b}_2^2 \end{bmatrix}$$

- Thus

$$\mathbf{W}^3\mathbf{h}^2 = \begin{bmatrix} \mathbf{h}_1^2\mathbf{w}_{11}^3 + \mathbf{h}_2^2\mathbf{w}_{12}^3 \\ \mathbf{h}_1^2\mathbf{w}_{21}^3 + \mathbf{h}_2^2\mathbf{w}_{22}^3 \end{bmatrix}$$

$$\mathbf{W}^3\mathbf{h}^2 + \mathbf{b}^3 = \begin{bmatrix} \mathbf{h}_1^2\mathbf{w}_{11}^3 + \mathbf{h}_2^2\mathbf{w}_{12}^3 + \mathbf{b}_1^3 \\ \mathbf{h}_1^2\mathbf{w}_{21}^3 + \mathbf{h}_2^2\mathbf{w}_{22}^3 + \mathbf{b}_2^3 \end{bmatrix}$$

$$\mathbf{o} = \mathbf{h}\left(\mathbf{W}^3\mathbf{h}^2 + \mathbf{b}^3\right) = \begin{bmatrix} \mathbf{h}\left(\mathbf{h}_1^2\mathbf{w}_{11}^3 + \mathbf{h}_2^2\mathbf{w}_{12}^3 + \mathbf{b}_1^3\right) \\ \mathbf{h}\left(\mathbf{h}_1^2\mathbf{w}_{21}^3 + \mathbf{h}_2^2\mathbf{w}_{22}^3 + \mathbf{b}_2^3\right) \end{bmatrix}$$

Training NN: -Log Loss Function

$$\mathbf{o} = \begin{bmatrix} \mathbf{o}_1 \\ \mathbf{o}_2 \end{bmatrix} = \mathbf{h}\left(\mathbf{W}^3\mathbf{h}^2 + \mathbf{b}^3\right) = \begin{bmatrix} \mathbf{h}_1^2\mathbf{w}_{11}^3 + \mathbf{h}_2^2\mathbf{w}_{12}^3 + \mathbf{b}_1^3 \\ \mathbf{h}_1^2\mathbf{w}_{21}^3 + \mathbf{h}_2^2\mathbf{w}_{22}^3 + \mathbf{b}_2^3 \end{bmatrix}$$

- Using chain rule

$$\frac{\partial \mathbf{L}}{\partial \mathbf{W}^3} = \begin{bmatrix} \dfrac{\partial \mathbf{L}}{\partial \mathbf{w}_{11}^3} & \dfrac{\partial \mathbf{L}}{\partial \mathbf{w}_{12}^3} \\[2ex] \dfrac{\partial \mathbf{L}}{\partial \mathbf{w}_{21}^3} & \dfrac{\partial \mathbf{L}}{\partial \mathbf{w}_{22}^3} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial \mathbf{L}}{\partial \mathbf{o}_1}\boxed{\dfrac{\partial \mathbf{o}_1}{\partial \mathbf{w}_{11}^3}} & \dfrac{\partial \mathbf{L}}{\partial \mathbf{o}_1}\boxed{\dfrac{\partial \mathbf{o}_1}{\partial \mathbf{w}_{12}^3}} \\[2ex] \dfrac{\partial \mathbf{L}}{\partial \mathbf{o}_2}\boxed{\dfrac{\partial \mathbf{o}_2}{\partial \mathbf{w}_{21}^3}} & \dfrac{\partial \mathbf{L}}{\partial \mathbf{o}_2}\boxed{\dfrac{\partial \mathbf{o}_2}{\partial \mathbf{w}_{22}^3}} \end{bmatrix}$$

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

Training NN: -Log Loss Function

- Need $$\frac{\partial \mathbf{o}}{\partial \mathbf{W}^3}$$

$$\mathbf{o} = \mathbf{h}\left(\mathbf{W}^3\mathbf{h}^2 + \mathbf{b}^3\right) = \begin{bmatrix} \mathbf{h}\left(\mathbf{h}_1^2\mathbf{w}_{11}^3 + \mathbf{h}_2^2\mathbf{w}_{12}^3 + \mathbf{b}_1^3\right) \\ \mathbf{h}\left(\mathbf{h}_1^2\mathbf{w}_{21}^3 + \mathbf{h}_2^2\mathbf{w}_{22}^3 + \mathbf{b}_2^3\right) \end{bmatrix}$$

- Assume ReLu **h**(**z**) = max(**z**,**0**)

- Assuming non-negativity of input to function **h**

$$\mathbf{o} = \mathbf{h}\left(\mathbf{W}^3\mathbf{h}^2 + \mathbf{b}^3\right) = \begin{bmatrix} \mathbf{h}_1^2\mathbf{w}_{11}^3 + \mathbf{h}_2^2\mathbf{w}_{12}^3 + \mathbf{b}_1^3 \\ \mathbf{h}_1^2\mathbf{w}_{21}^3 + \mathbf{h}_2^2\mathbf{w}_{22}^3 + \mathbf{b}_2^3 \end{bmatrix}$$

- If there are negative components, replace by 0

Training NN: -Log Loss Function

- Continue

$$\mathbf{o} = \begin{bmatrix} \mathbf{h}_1^2 \mathbf{w}_{11}^3 + \mathbf{h}_2^2 \mathbf{w}_{12}^3 + \mathbf{b}_1^3 \\ \mathbf{h}_1^2 \mathbf{w}_{21}^3 + \mathbf{h}_2^2 \mathbf{w}_{22}^3 + \mathbf{b}_2^3 \end{bmatrix}$$

$$\frac{\partial \mathbf{o}}{\partial \mathbf{W}^3} = \begin{bmatrix} \frac{\partial \mathbf{o}}{\partial \mathbf{w}_{11}^3} & \frac{\partial \mathbf{o}}{\partial \mathbf{w}_{12}^3} \\ \frac{\partial \mathbf{o}}{\partial \mathbf{w}_{21}^3} & \frac{\partial \mathbf{o}}{\partial \mathbf{w}_{22}^3} \end{bmatrix} = \begin{bmatrix} \mathbf{h}_1^2 & \mathbf{h}_2^2 \\ \mathbf{h}_1^2 & \mathbf{h}_2^2 \end{bmatrix}$$

- Plug into

$$\frac{\partial \mathbf{L}}{\partial \mathbf{W}^3} = \begin{bmatrix} \frac{\partial \mathbf{L}}{\partial \mathbf{o}_1} \frac{\partial \mathbf{o}_1}{\partial \mathbf{w}_{11}^3} & \frac{\partial \mathbf{L}}{\partial \mathbf{o}_1} \frac{\partial \mathbf{o}_1}{\partial \mathbf{w}_{12}^3} \\ \frac{\partial \mathbf{L}}{\partial \mathbf{o}_2} \frac{\partial \mathbf{o}_2}{\partial \mathbf{w}_{21}^3} & \frac{\partial \mathbf{L}}{\partial \mathbf{o}_2} \frac{\partial \mathbf{o}_2}{\partial \mathbf{w}_{22}^3} \end{bmatrix} = \begin{bmatrix} \frac{\partial \mathbf{L}}{\partial \mathbf{o}_1} \mathbf{h}_1^2 & \frac{\partial \mathbf{L}}{\partial \mathbf{o}_1} \mathbf{h}_2^2 \\ \frac{\partial \mathbf{L}}{\partial \mathbf{o}_2} \mathbf{h}_1^2 & \frac{\partial \mathbf{L}}{\partial \mathbf{o}_2} \mathbf{h}_2^2 \end{bmatrix}$$

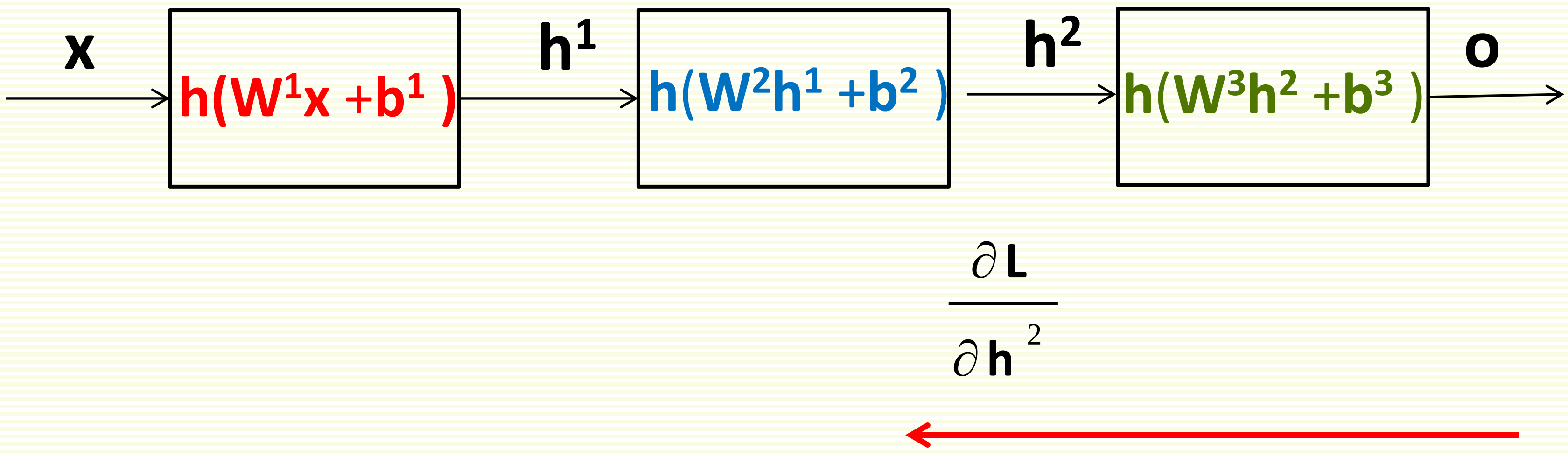
Training NN: -Log Loss Function

- Rewrite $$\frac{\partial \mathbf{L}}{\partial \mathbf{W}^3} = \begin{bmatrix} \frac{\partial \mathbf{L}}{\partial \mathbf{o}_1}\mathbf{h}_1^2 & \frac{\partial \mathbf{L}}{\partial \mathbf{o}_1}\mathbf{h}_2^2 \\ \frac{\partial \mathbf{L}}{\partial \mathbf{o}_2}\mathbf{h}_1^2 & \frac{\partial \mathbf{L}}{\partial \mathbf{o}_2}\mathbf{h}_2^2 \end{bmatrix} = \begin{bmatrix} \frac{\partial \mathbf{L}}{\partial \mathbf{o}_1} \\ \frac{\partial \mathbf{L}}{\partial \mathbf{o}_2} \end{bmatrix} \begin{bmatrix} \mathbf{h}_1^2 & \mathbf{h}_2^2 \end{bmatrix}$$

- Recall $$\frac{\partial \mathbf{L}}{\partial \mathbf{o}} = \mathbf{f}(\mathbf{x}) - \mathbf{y} = \begin{bmatrix} \mathbf{f}_1(\mathbf{x}) - \mathbf{y}_1 \\ \vdots \\ \mathbf{f}_\mathbf{m}(\mathbf{x}) - \mathbf{y}_\mathbf{m} \end{bmatrix}$$

- So, finally $$\frac{\partial \mathbf{L}}{\partial \mathbf{W}^3} = (\mathbf{f}(\mathbf{x}) - \mathbf{y})(\mathbf{h}^2)^\mathbf{T}$$

Training NN: -Log Loss Function

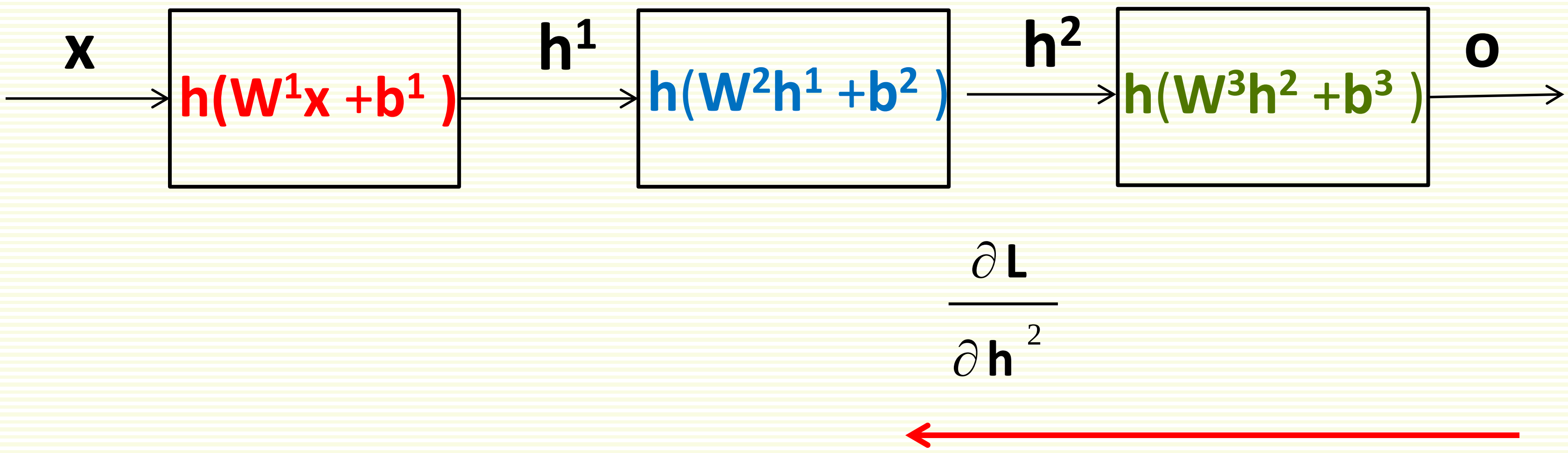


- Continue compute derivatives “backwards”

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

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

Training NN: -Log Loss Function

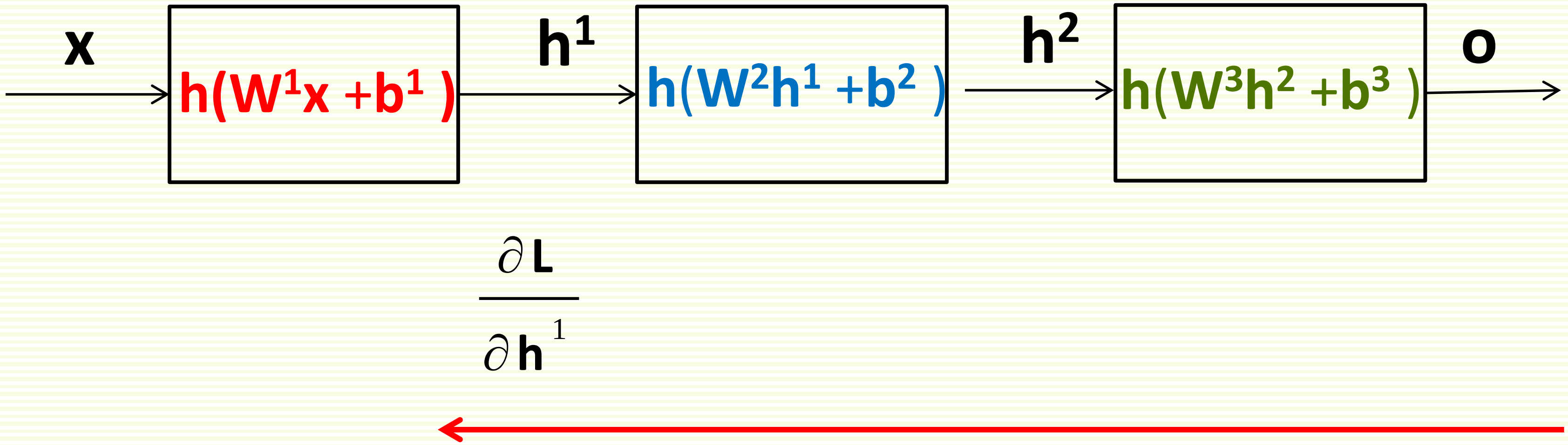


- Continue computing derivatives "backwards"

$$\frac{\partial \mathbf{L}}{\partial \mathbf{W}^2} = \frac{\partial \mathbf{L}}{\partial \mathbf{h}^2} \frac{\partial \mathbf{h}^2}{\partial \mathbf{W}^2} = \frac{\partial \mathbf{L}}{\partial \mathbf{h}^2} \left(\mathbf{h}^1\right)^{\mathsf{T}}$$

$$\frac{\partial \mathbf{L}}{\partial \mathbf{h}^1} = \frac{\partial \mathbf{L}}{\partial \mathbf{h}^2} \frac{\partial \mathbf{h}^2}{\partial \mathbf{h}^1} = \left(\mathbf{W}^2\right)^{\mathsf{T}} \frac{\partial \mathbf{L}}{\partial \mathbf{h}^2}$$

Training NN: -Log Loss Function



- Continue computing derivatives “backwards”

$$\frac{\partial \mathbf{L}}{\partial \mathbf{W}^{1}} = \frac{\partial \mathbf{L}}{\partial \mathbf{h}^{1}} \frac{\partial \mathbf{h}^{1}}{\partial \mathbf{W}^{1}} = \frac{\partial \mathbf{L}}{\partial \mathbf{h}^{1}} (\mathbf{x})^{\mathbf{T}}$$

Training Protocols

- Batch Protocol
  - full gradient descent
  - weights are updated only after all examples are processed
  - might be very slow to train
- Single Sample Protocol
  - examples are chosen randomly from the training set
  - weights are updated after every example
  - weighs get changed faster than batch, less stable
  - One iteration over all samples (in random order) is called an **epoch**
- Mini Batch
  - Divide data in batches, and update weights after processing each batch
  - Middle ground between single sample and batch protocols
  - Helps to prevent over-fitting in practice, think of it as "noisy" gradient
  - allows CPU/GPU memory hierarchy to be exploited so that it trains much faster than single-sample in terms of wall-clock time
  - One iteration over all mini-batches is called an **epoch**

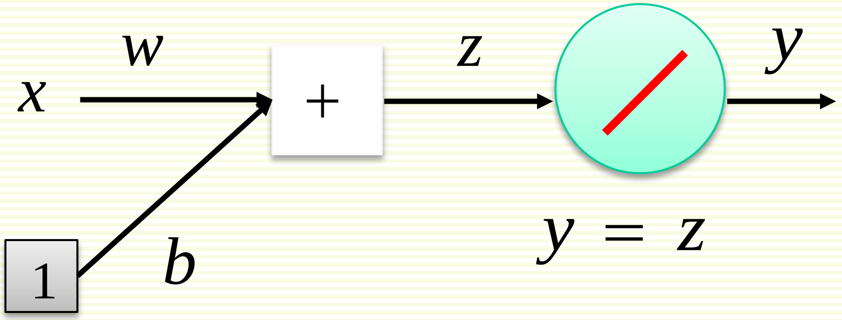
Training DNN: Initialization

- For gradient descent, need to pick initialization parameters $\mathbf{w}^0$
  - do not set all the parameters $\mathbf{w}^0$ equal
  - set the parameters in $\mathbf{w}^0$ randomly

Training DNN: Learning Rate

- Set the learning rate carefully

- Toy example



- Optimal weights: w = 1, b = 0
- Gradient descent

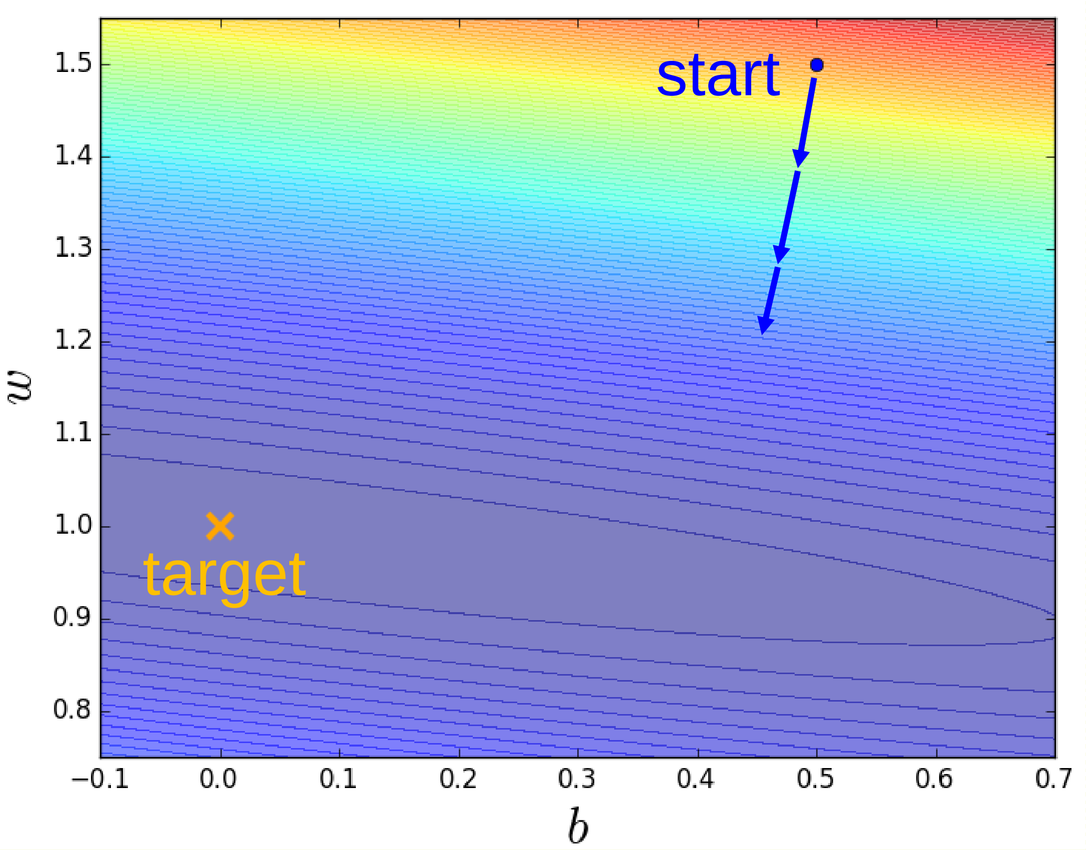
$$\mathbf{w}^{\mathbf{t}} = \mathbf{w}^{\mathbf{t}-1} - \alpha \nabla \mathbf{L}\left(\mathbf{w}^{\mathbf{t}-1}\right)$$

- Training Data (20 examples)

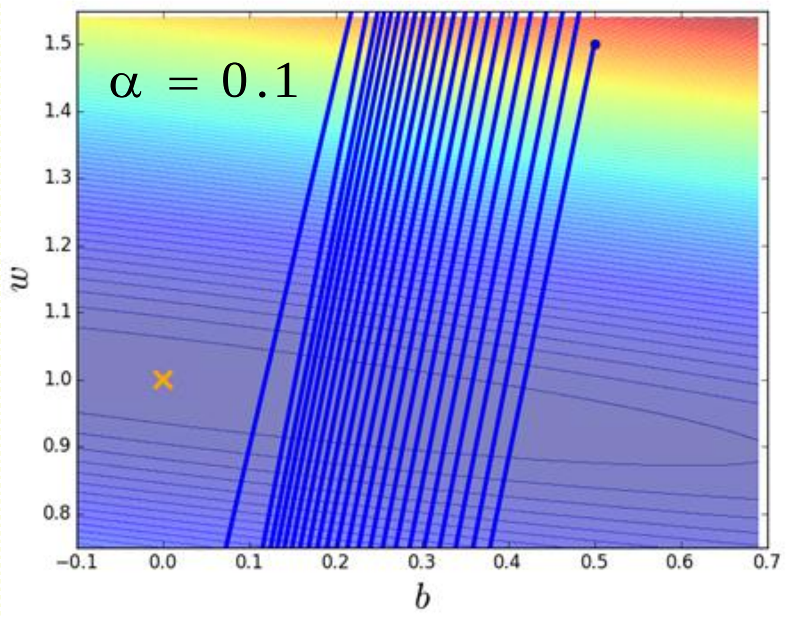
x = [0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5]
y = [0.1, 0.4, 0.9, 1.6, 2.2, 2.5, 2.8, 3.5, 3.9, 4.7, 5.1, 5.3, 6.3, 6.5, 6.7, 7.5, 8.1, 8.5, 8.9, 9.5]

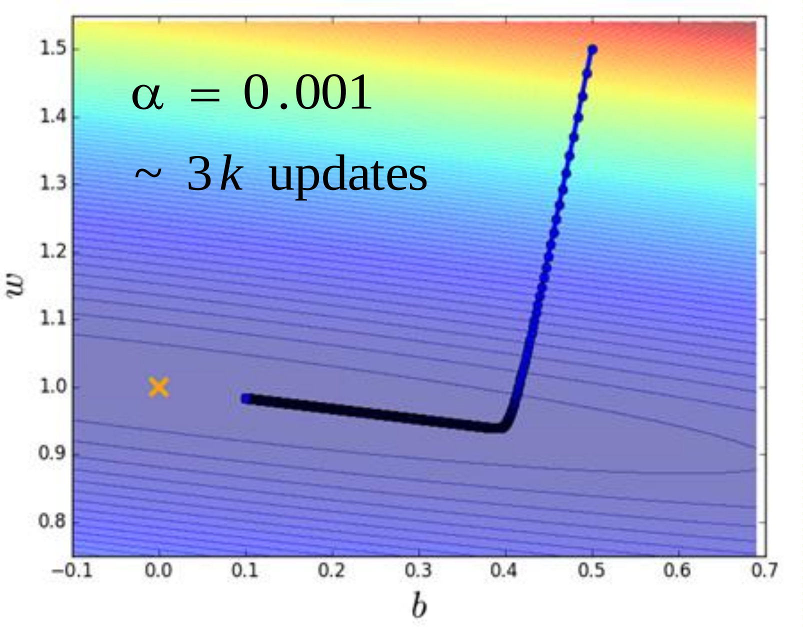
Training DNN: Learning Rate

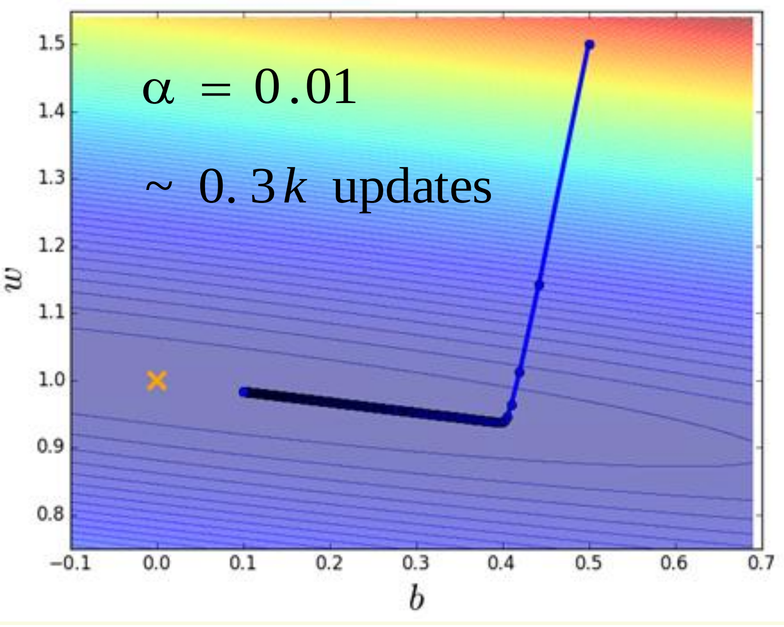
- Surface of the loss function **L(w,b)**



Training DNN: Learning Rate



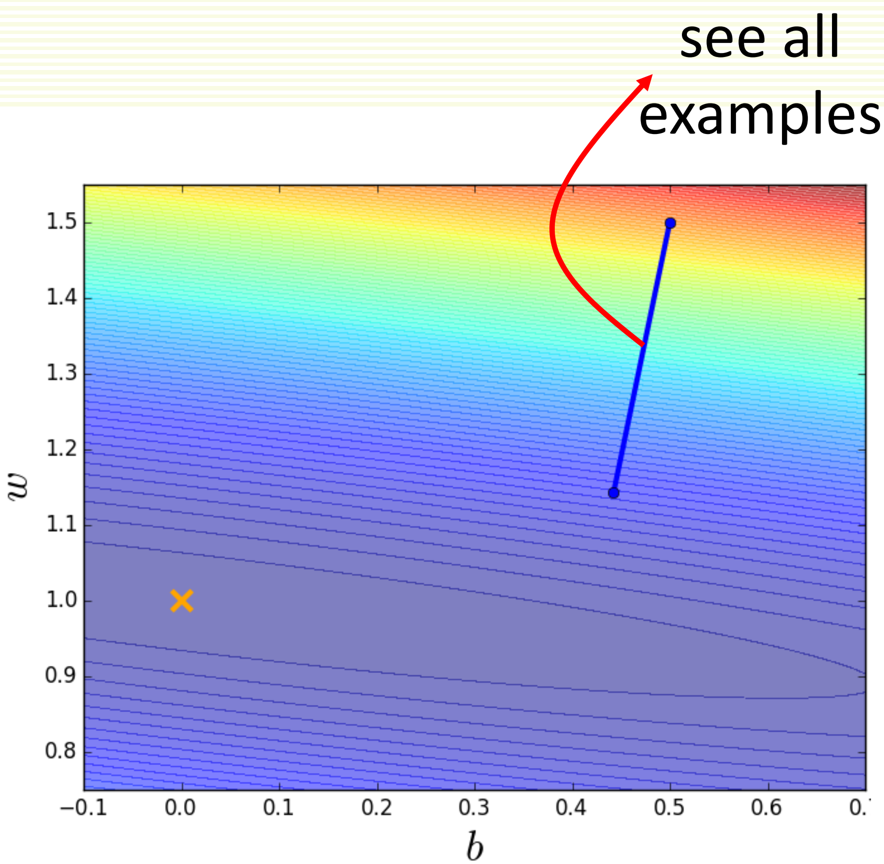




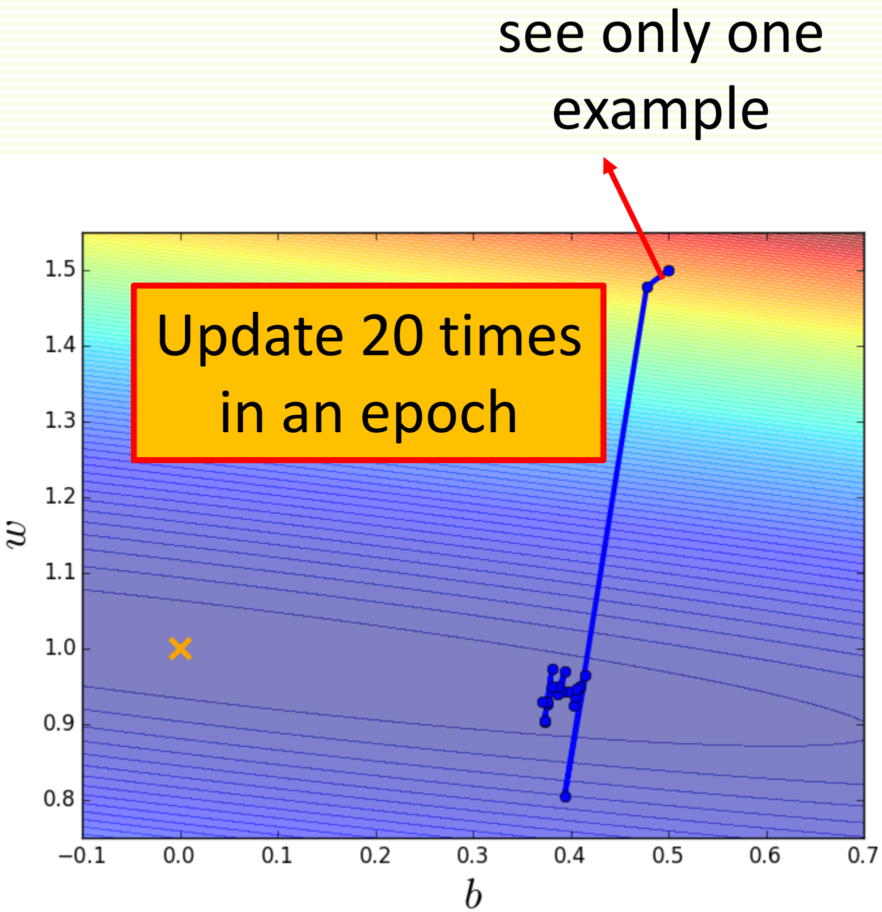
Training DNN: Learning Rate

- Can adjust α at the training time
- The loss function $\mathbf{L}(\mathbf{w})$ should decrease during gradient descent
  - If $\mathbf{L}(\mathbf{w})$ oscillates, α is too large, decrease it
  - If $\mathbf{L}(\mathbf{w})$ goes down but very slowly, α is too small, increase it

Training DNN: Gradient descent



Gradient descent



Stochastic gradient descent,
1 epoch

Training DNN: Gradient descent

- Real Example: Handwriting Digit Classification

Gradient descent

Training DNN: Momentum

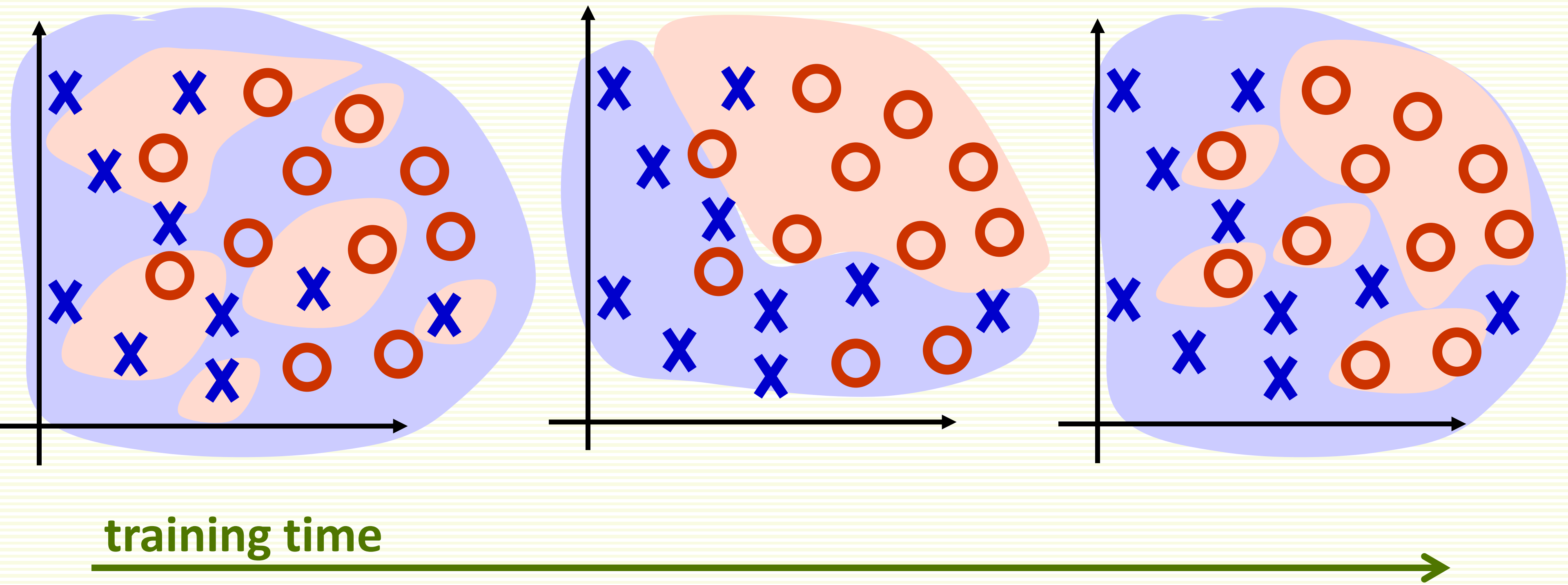
- Gradient descent finds only a local minima
- Momentum: popular method to avoid local minima and speed up descent in flat (plateau) regions
- Add temporal average direction in which weights have been moving recently
- Previous direction: $\Delta\mathbf{w}^{t}=\mathbf{w}^{t}-\mathbf{w}^{t-1}$
- Weight update rule with momentum:

$$\mathbf{w}^{t+1} = \mathbf{w}^{t} + \underbrace{(1-\beta)\nabla \mathbf{L}\left(\mathbf{w}^{t}\right)}_{\text{steepest descent direction}} + \underbrace{\beta\,\Delta\mathbf{w}^{t-1}}_{\text{previous direction}}$$

Training DNN: Normalization

- Features should be normalized for faster convergence
- Suppose fish length is in meters and weight in grams
    - typical sample [length = 0.5, weight = 3000]
    - feature length will be almost ignored
    - If length is in fact important, learning will be very slow
- Any normalization we looked at before will do
    - test samples should be normalized exactly as training samples

Trainind DNN: How Many Epochs?



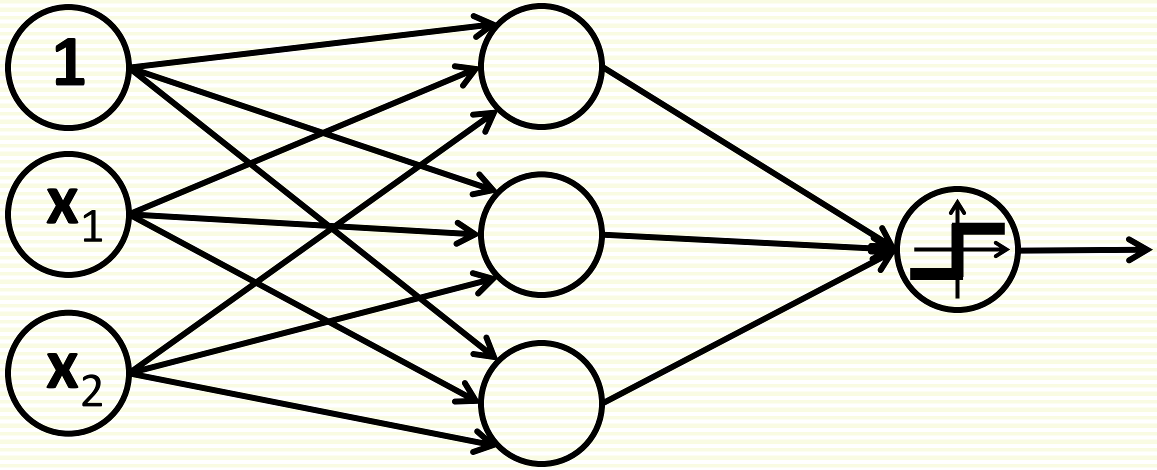
Large training error: random decision regions in the beginning - underfit

Small training error: decision regions improve with time

Zero training error: decision regions fit training data perfectly - overfit

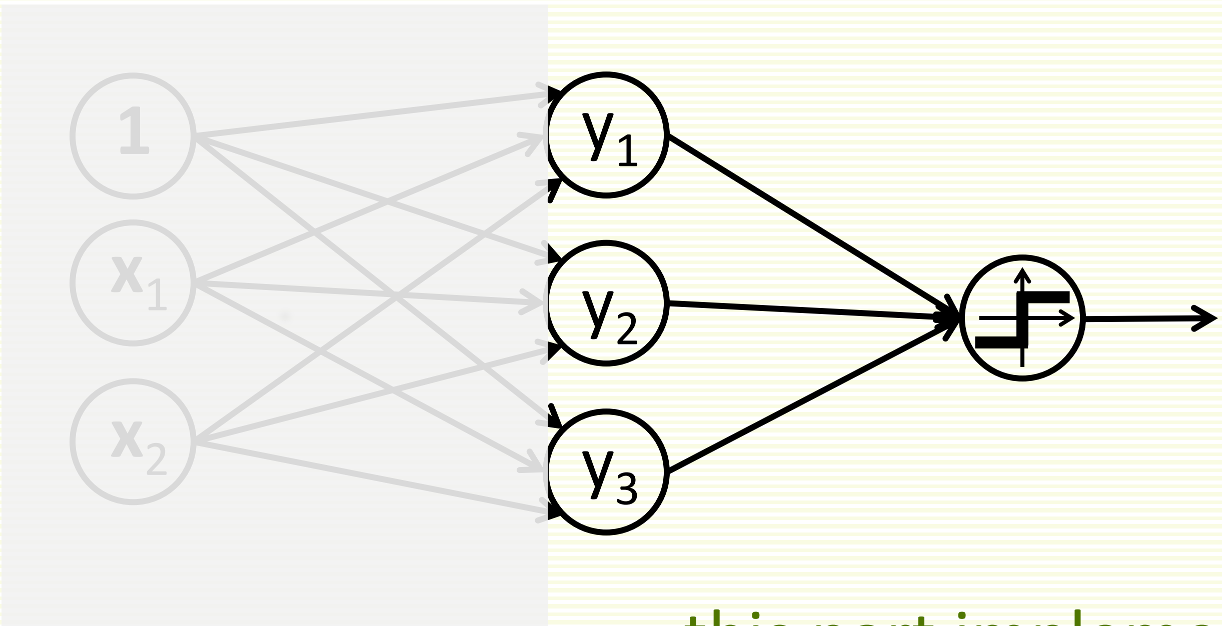
can learn when to stop training through validation

NN as Non-Linear Feature Mapping



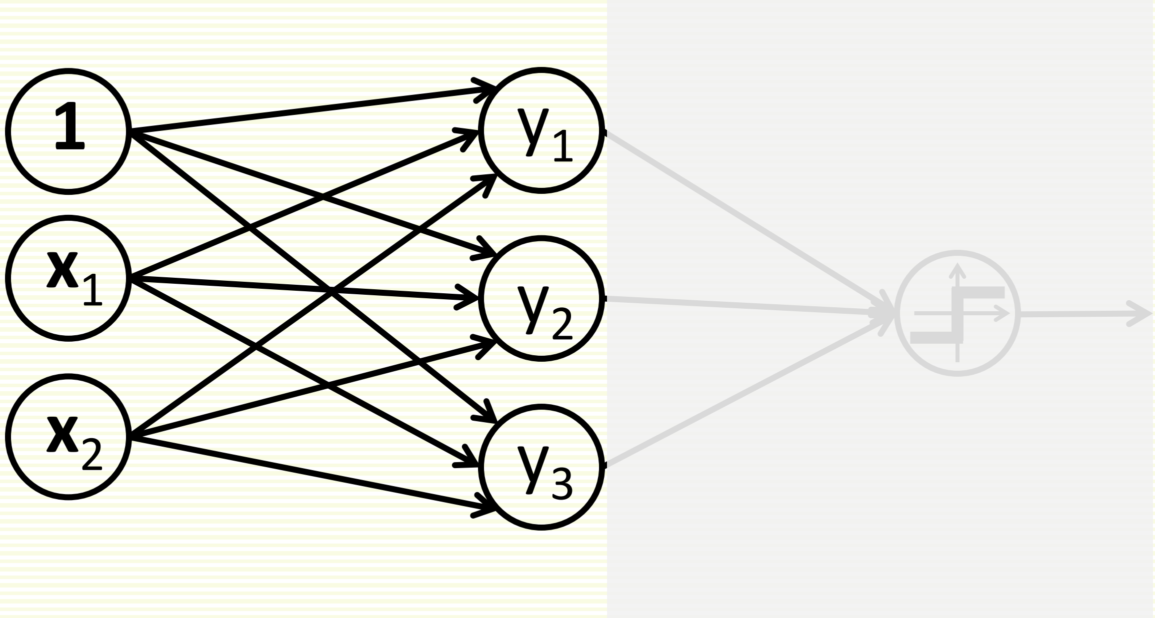
- 1 hidden layer NN can be interpreted as first mapping input features to new features
- Then applying (linear classifier) to the new features

NN as Non-Linear Feature Mapping



this part implements
Perceptron (liner classifier)

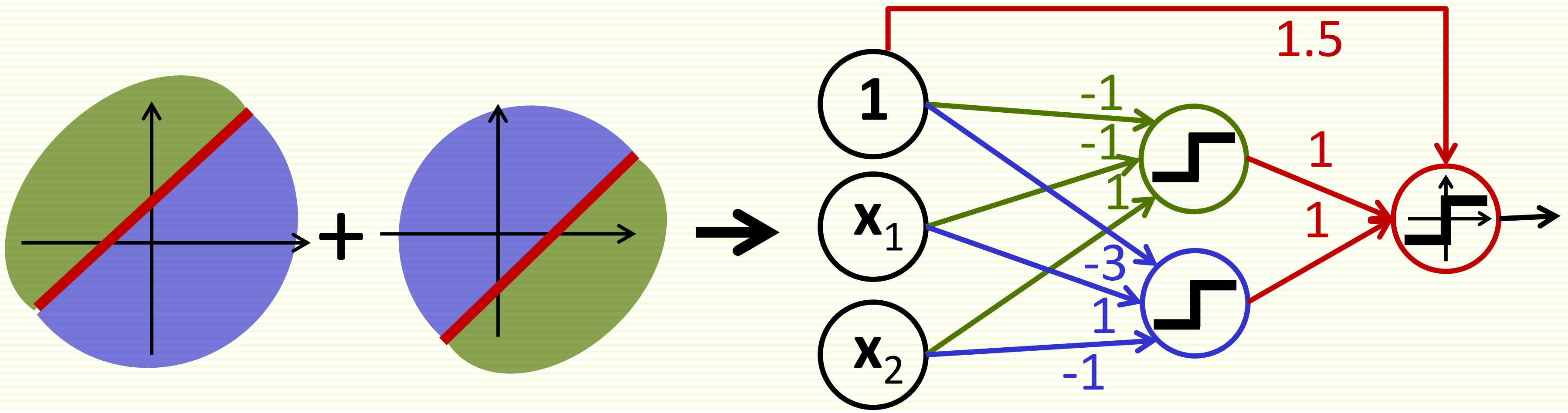
NN as Non-Linear Feature Mapping

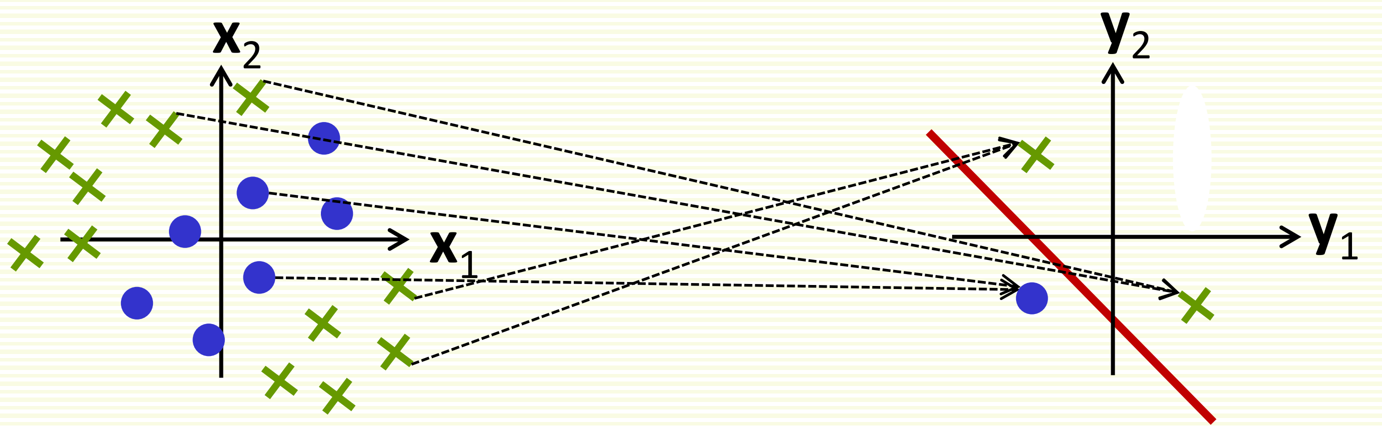


this part implements mapping to new features **y**

NN as Nonlinear Feature Mapping

- Consider 3 layer NN example we saw previously:





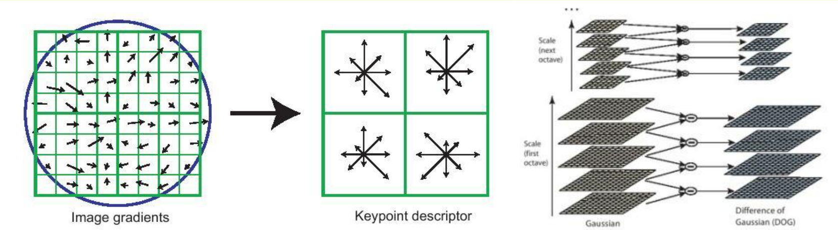
non linearly separable in the original feature space

linearly separable in the new feature space

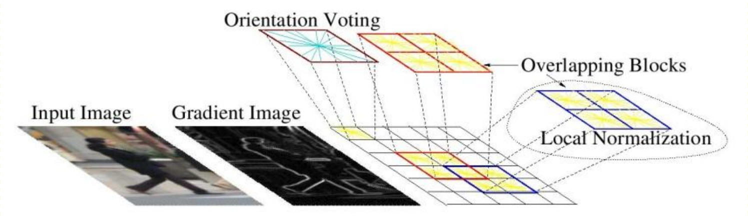
NN as Nonlinear Feature Mapping

- Features are key to recent success in object recognition
- Multitude of hand-crafted features, time consuming

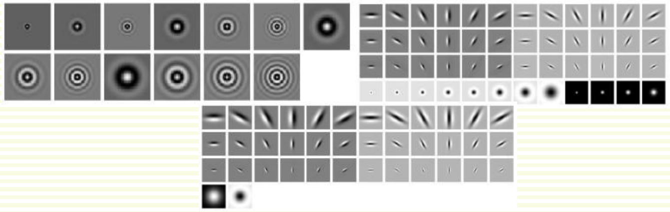
SIFT



HOG



Textons



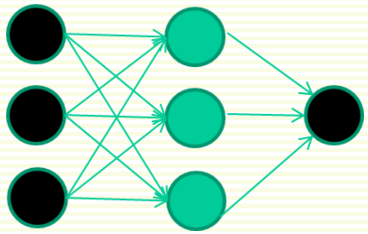
Patches

- With NN, change in paradigm: instead of hand-crafting , learn features automatically from data

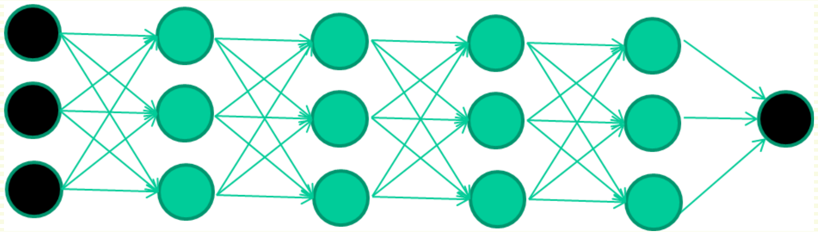
Shallow vs. Deep Architecture

- How many layers should we choose?

Shallow network



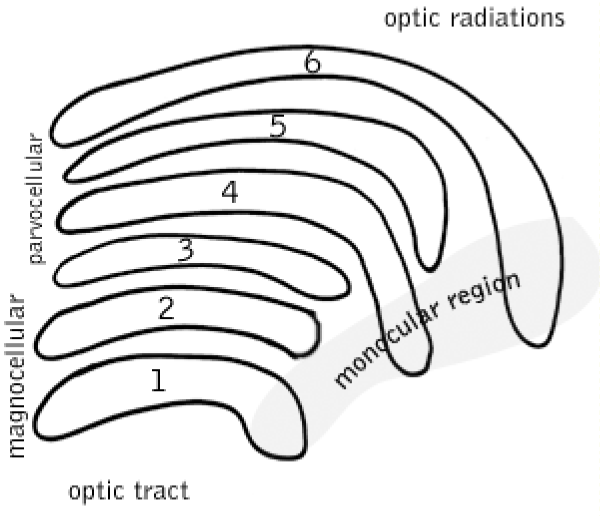
Deep network

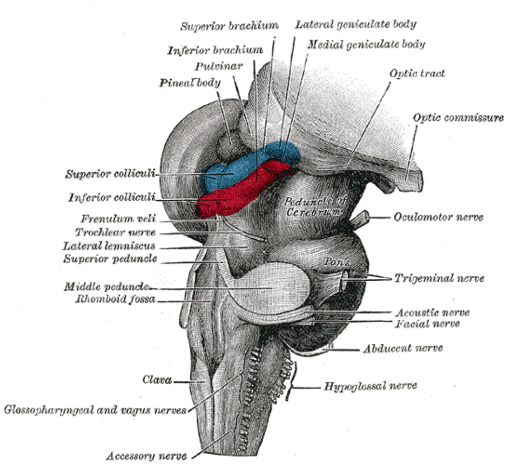


- Deep network lead to many successful applications recently

Why Deep Networks

- Evidence from biology



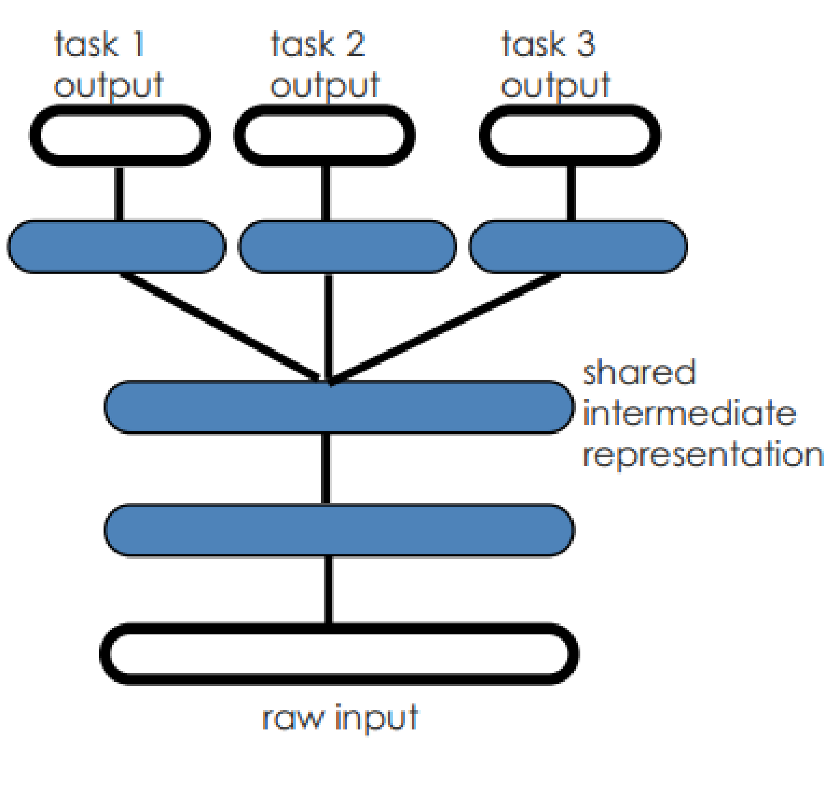


Why Deep Networks

- 2 layer networks can represent any function
- But deep architectures are more efficient for representing some functions
  - problems that can be represented with a polynomial number of nodes with *k* layers, may require an exponential number of nodes with *k*-1 layers
  - thus with deep architecture, less units might be needed overall
    - less weights, less parameter updates, more efficient

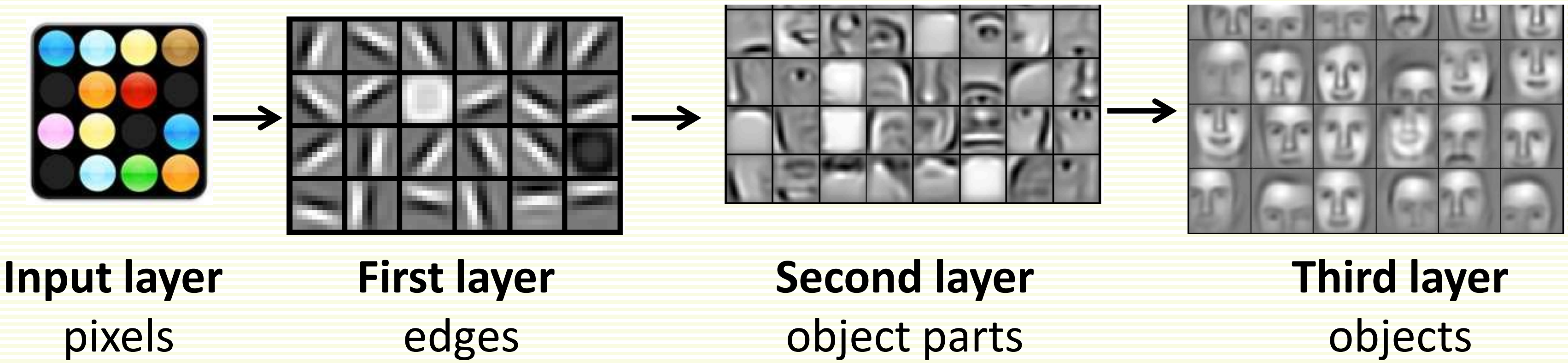
Why Deep Networks

- Sub-features created in deep architecture can potentially be shared between multiple tasks



Why Deep Networks: Hierarchical Feature Extraction

- Deep architecture works well for hierarchical feature extraction
  - hierarchies features are especially natural in vision
- Each stage is a trainable feature transform
- Level of abstraction increases up the hierarchy



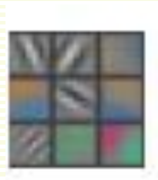
Why Deep Networks: Hierarchical Feature Extraction

- Another example (from M. Zeiler’2013)

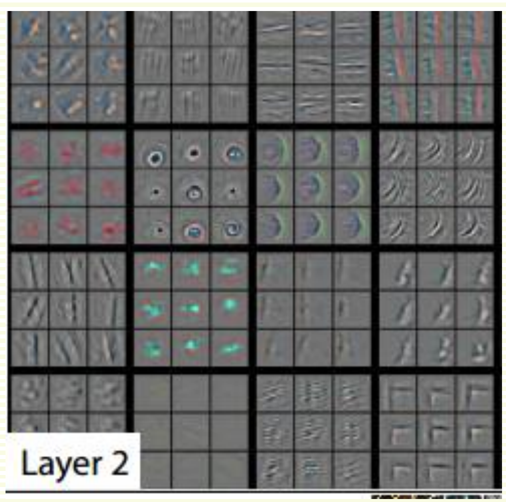
visualization of learned features

Patches that result in high response

Layer 1



Layer 2

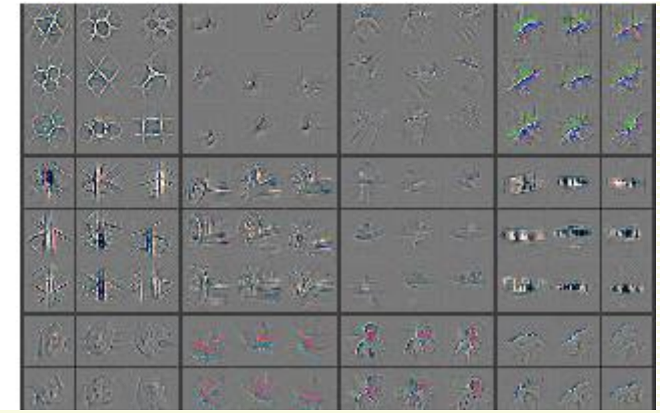


Why Deep Networks: Hierarchical Feature Extraction

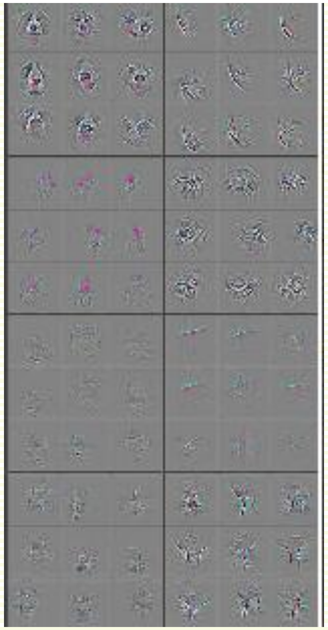
visualization of learned features

Patches that result in high response

Layer 3



Layer 4



Early Work on Deep Networks

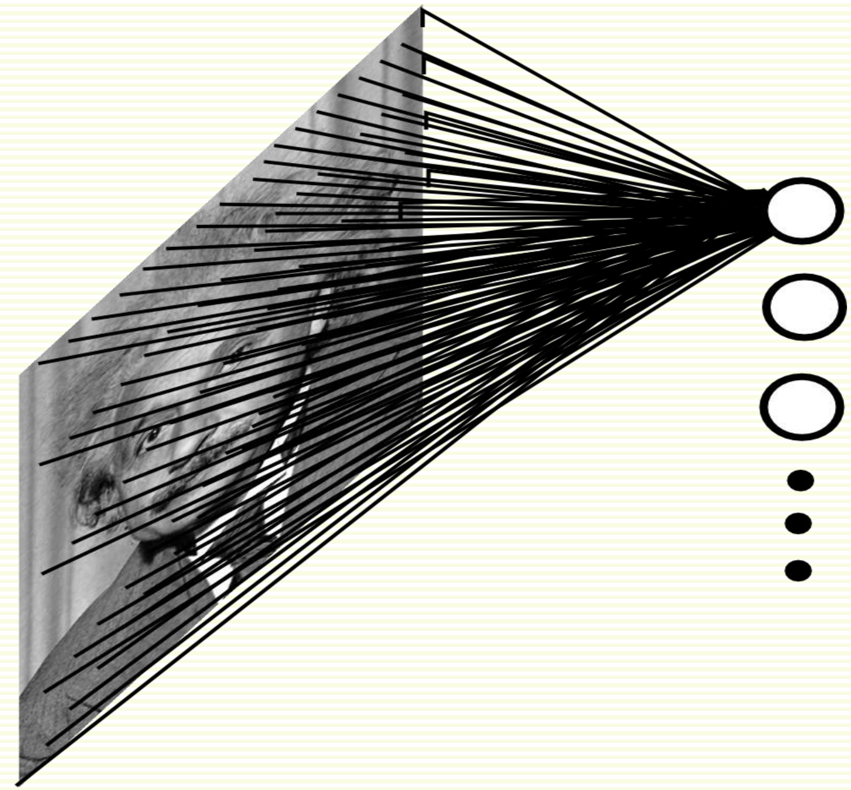
- Fukushima (1980) – Neo-Cognitron
- LeCun (1998) – Convolutional Networks (convnets)
  - Similarities to Neo-Cognitron
- Other attempts at deeply layered Networks trained with backpropagation
  - not much success
    - very slow
    - diffusion of gradient
  - recent work has shown significant training improvements with various tricks (drop-out, unsupervised learning of early layers, etc.)

ConvNets: Prior Knowledge for Network Architecture

- Convnets use prior knowledge about recognition task into network architecture design
  - connectivity structure
  - weight constraints
  - neuron activation functions
- This is less intrusive than hand-designing the features
  - but it still prejudices the network towards the particular way of solving the problem that we had in mind

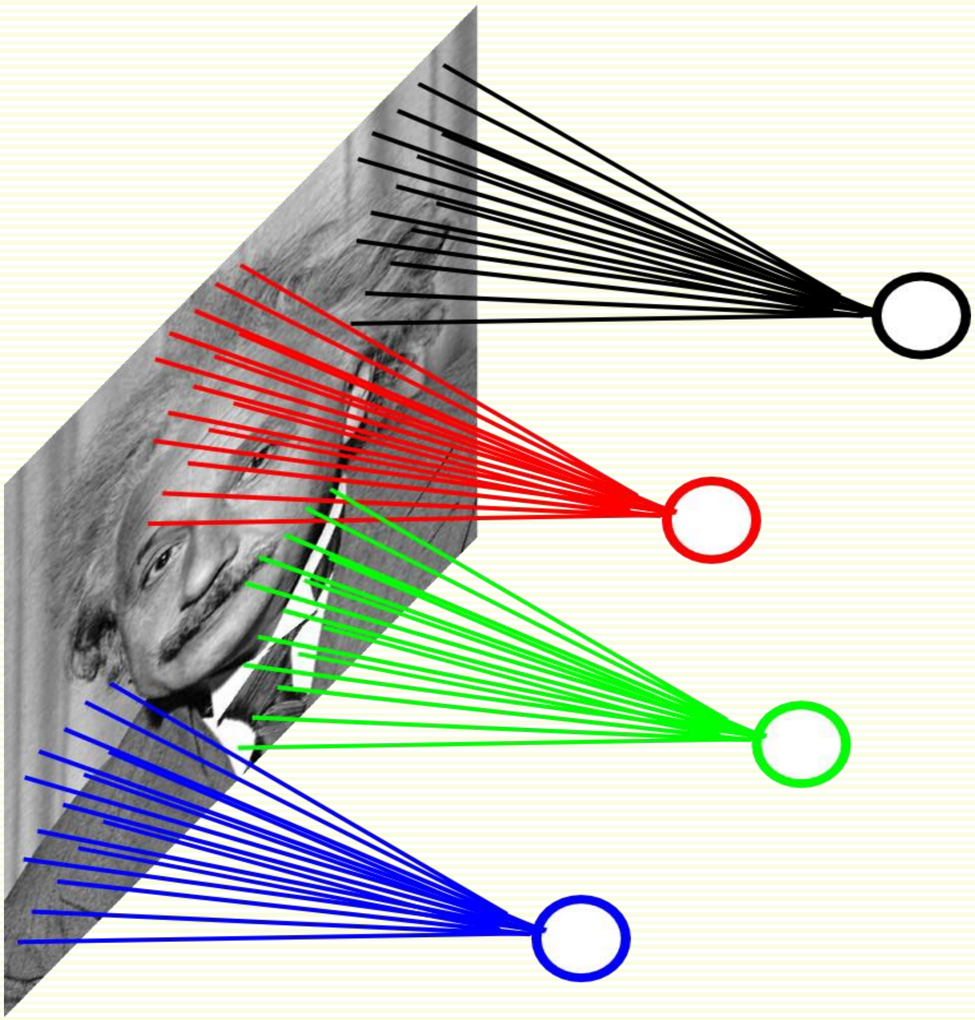
Convolutional Network: Motivation

- Consider a fully connected network
- Example: 200 by 200 image, 4×10^4 connections to one hidden unit
- For 10^5 hidden units $\rightarrow$ 4×10^9 connections
- But spatial correlations are mostly local
- Should not waste resources by connecting unrelated pixels



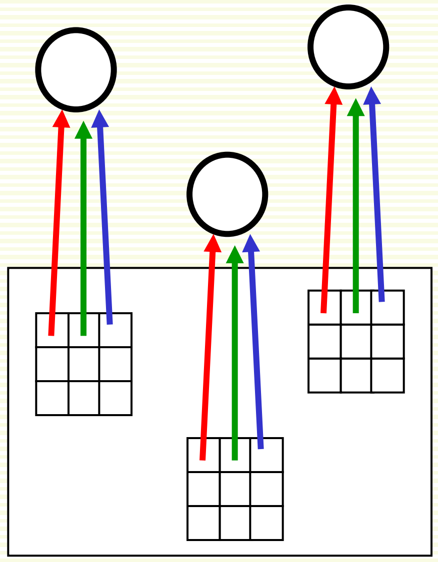
Convolutional Network: Motivation

- 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 → 10^7 connections
- factor of 400 decrease



Convolutional Network: Motivation

- If a feature is useful in one image location, it should be useful in all other locations
    - *Stationarity*: statistics is similar at different locations
- All neurons detect the same feature at different positions in the input image
    - i.e. share parameters (network weights) across different locations
    - bias is usually not shared
    - also greatly reduces the number of tunable parameters



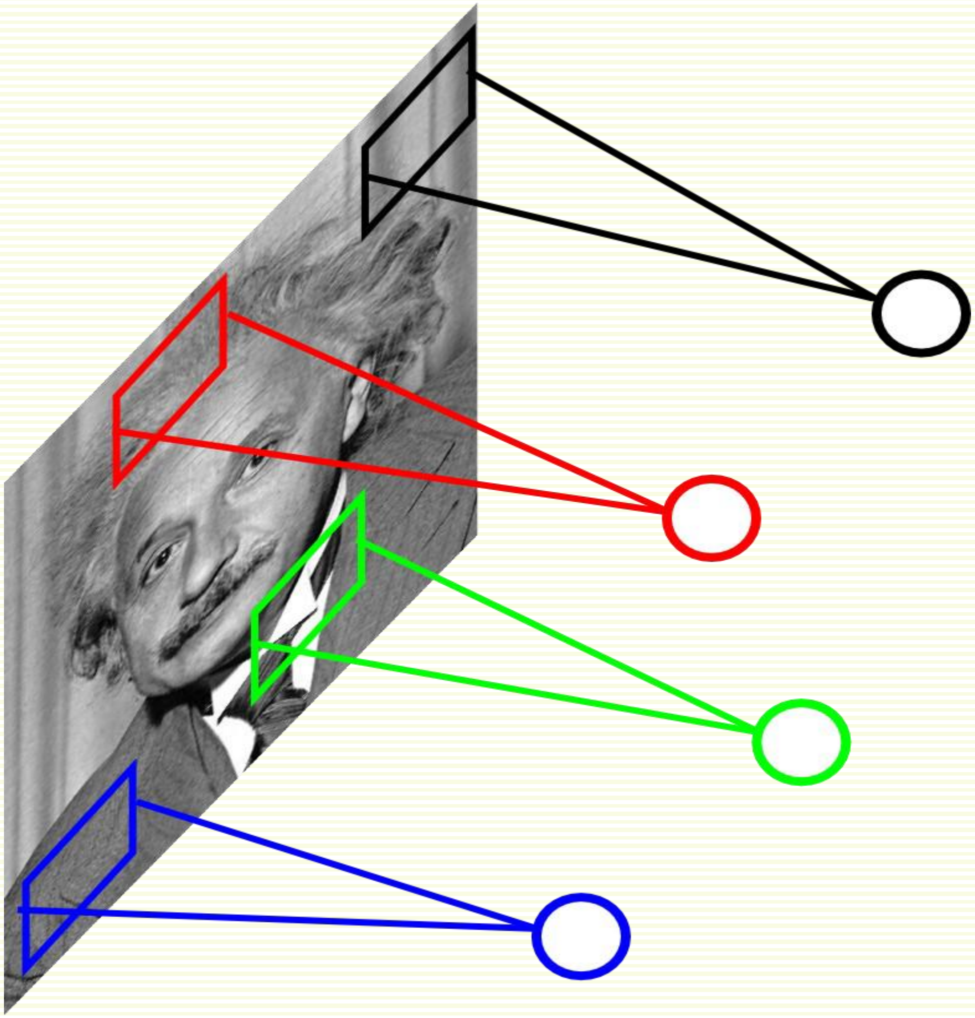
all red connections have the same weight

all green connections have the same weight

all blue connections have the same 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

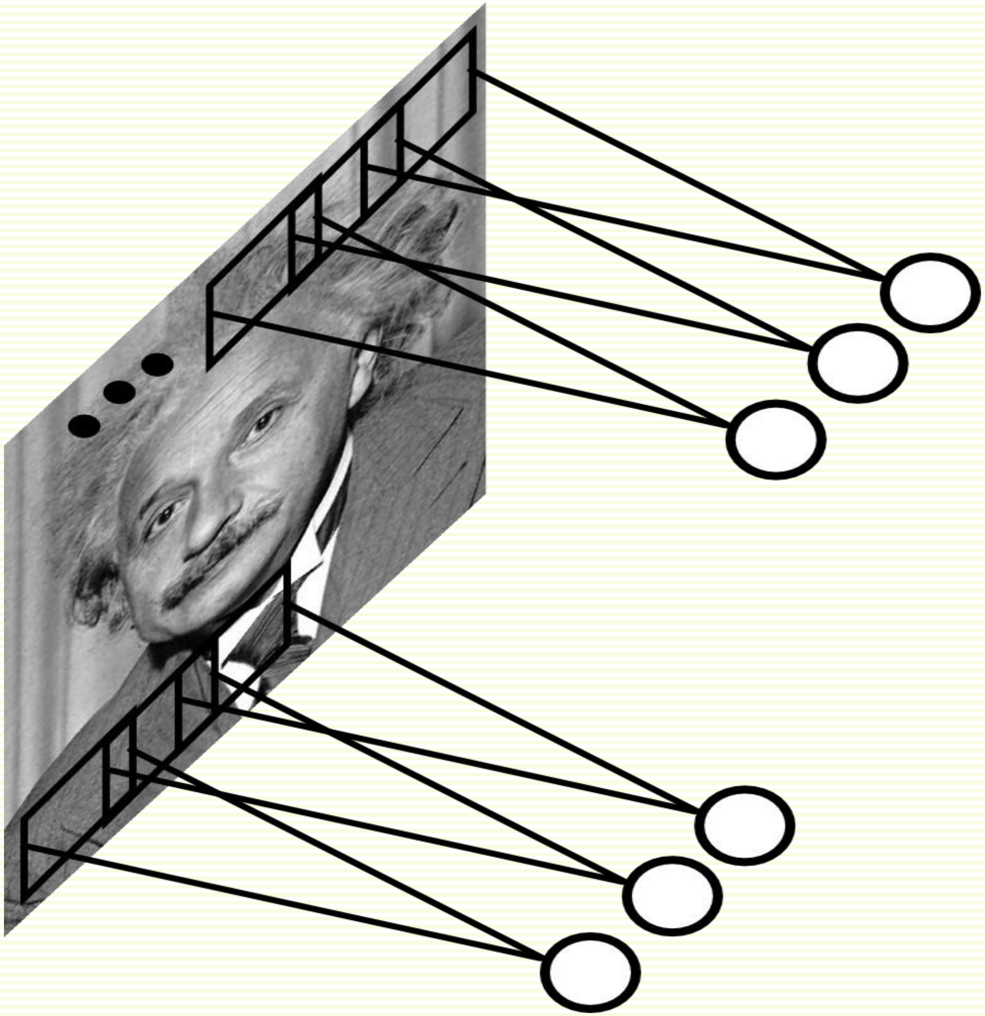


Weight Sharing Constraints

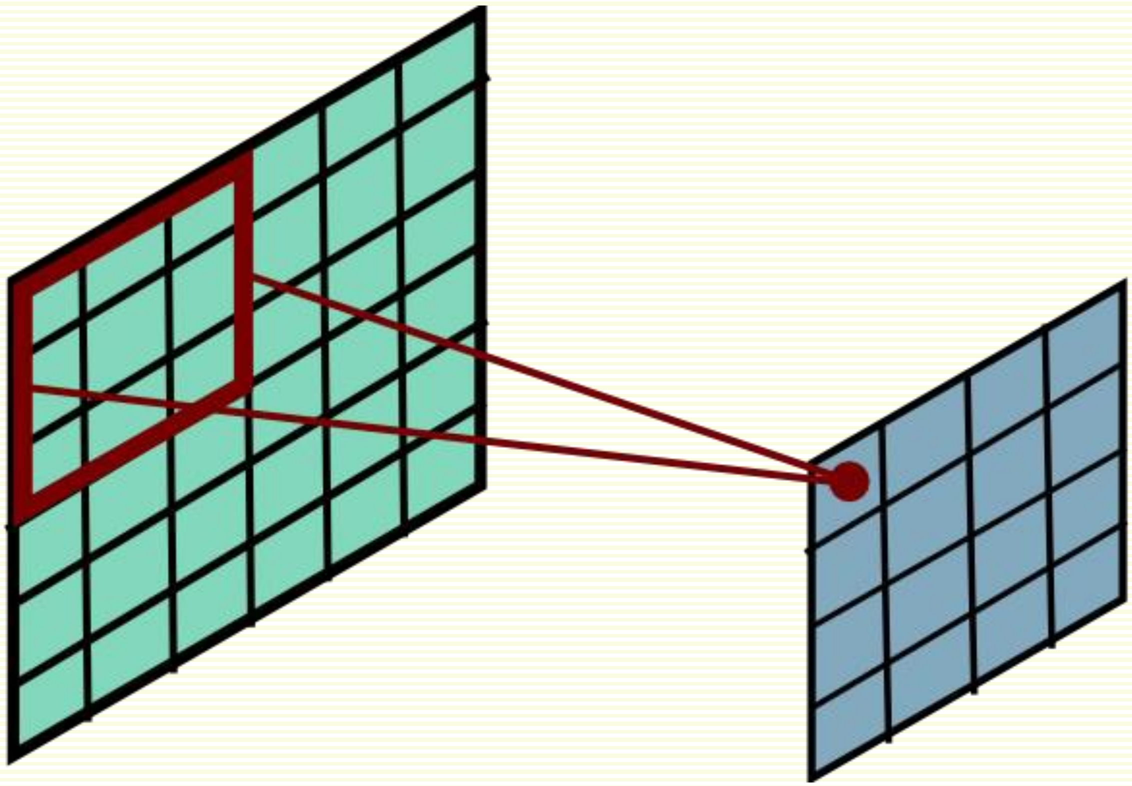
- Easy to modify backpropagation algorithm to incorporate weight sharing
- Compute the gradients as usual, and then modify the gradients so that they satisfy the constraints.
  - if the weights started off satisfying the constraints, they will continue to satisfy them
- To constrain $\mathbf{w_1} = \mathbf{w_2}$, we need $\Delta\mathbf{w_1} = \Delta\mathbf{w_2}$
- Before we used $\dfrac{\partial \mathbf{L}}{\partial \mathbf{w}_1}$ to update $\mathbf{w_1}$ and $\dfrac{\partial \mathbf{L}}{\partial \mathbf{w}_2}$ to update $\mathbf{w_2}$
- Now use $\dfrac{\partial \mathbf{E}}{\partial \mathbf{w}_1} + \dfrac{\partial \mathbf{E}}{\partial \mathbf{w}_2}$ to update $\mathbf{w_1}$ and $\mathbf{w_2}$, use

Convolutional Layer

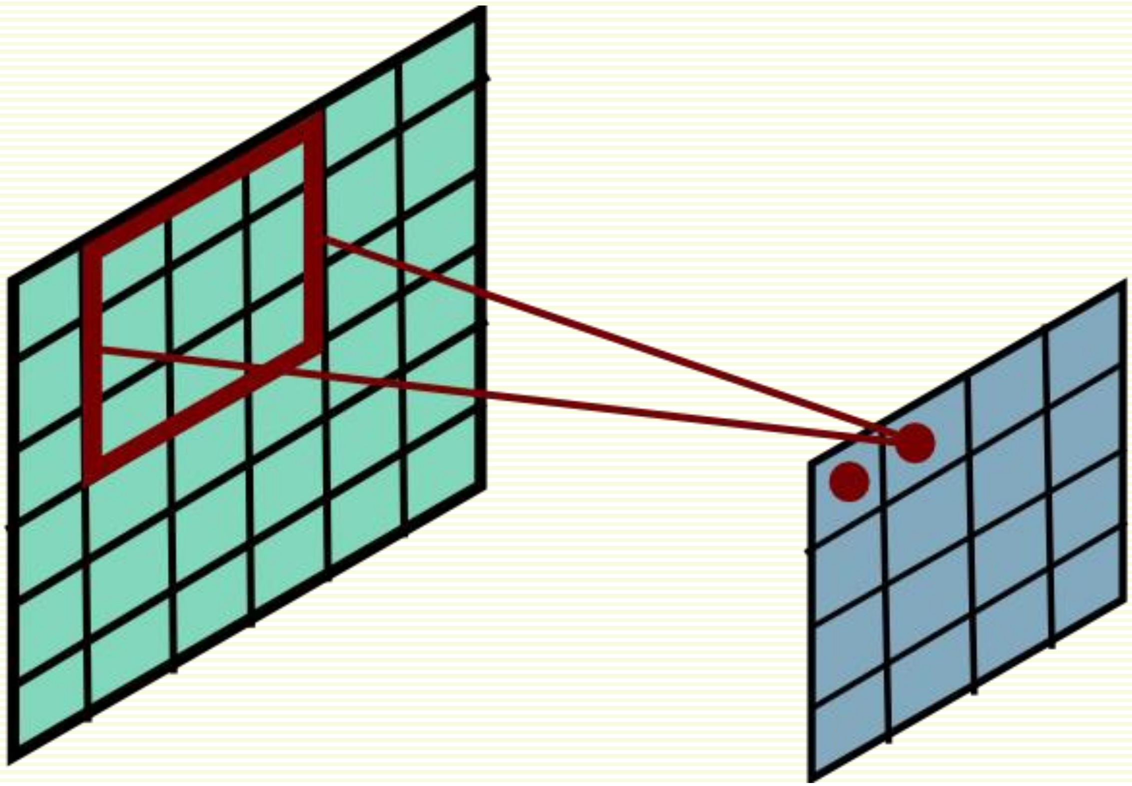
- Share parameters (network weights) across different locations
- Note similarity to convolution with some fixed filter
- But here the filter is learned



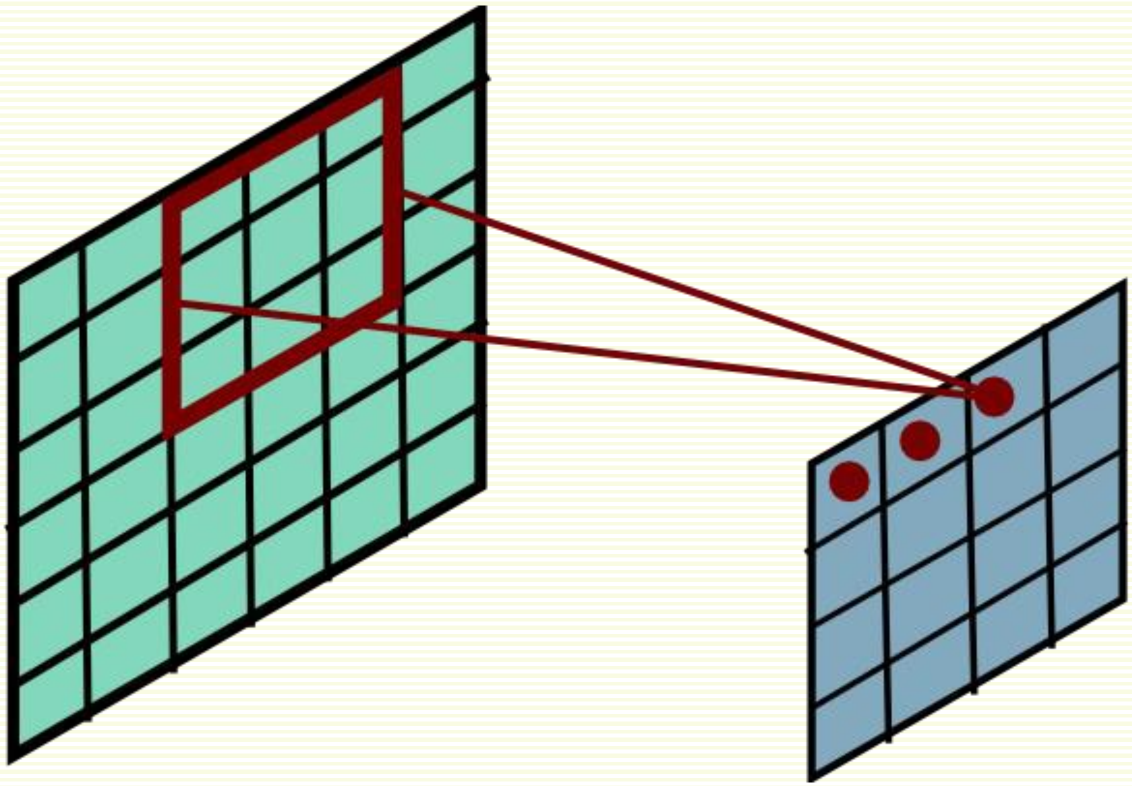
Convolutional Layer



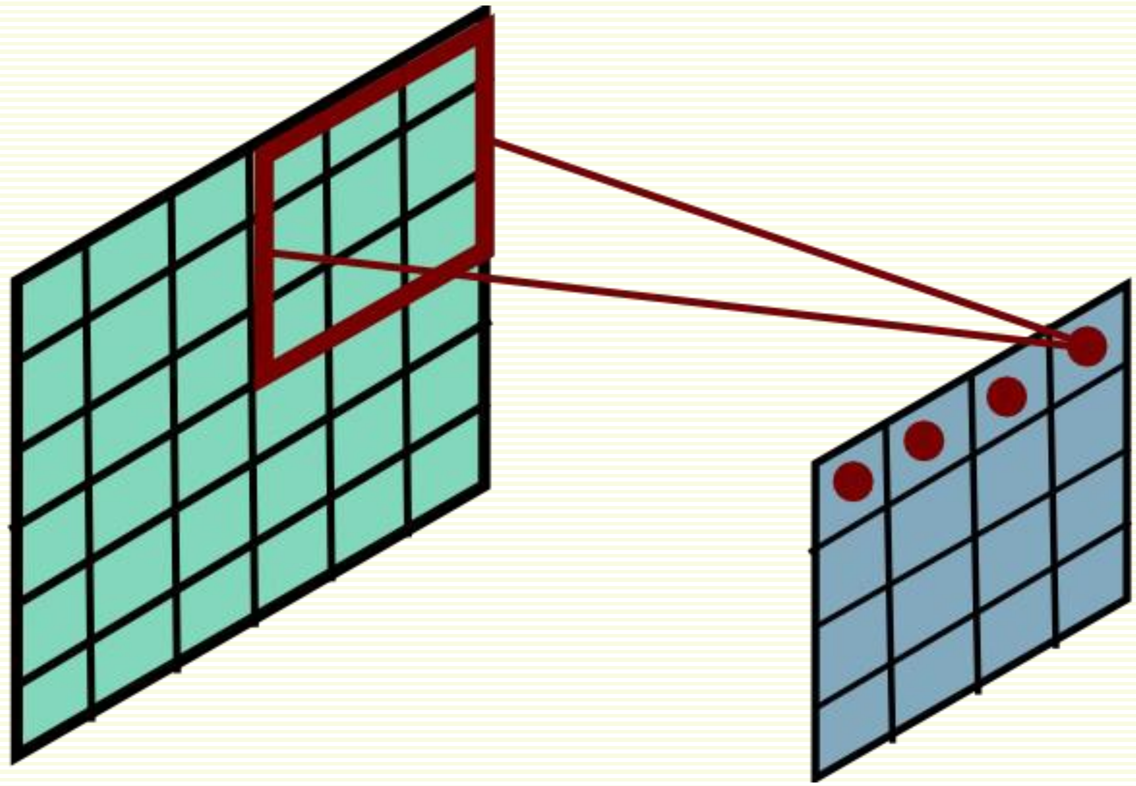
Convolutional Layer



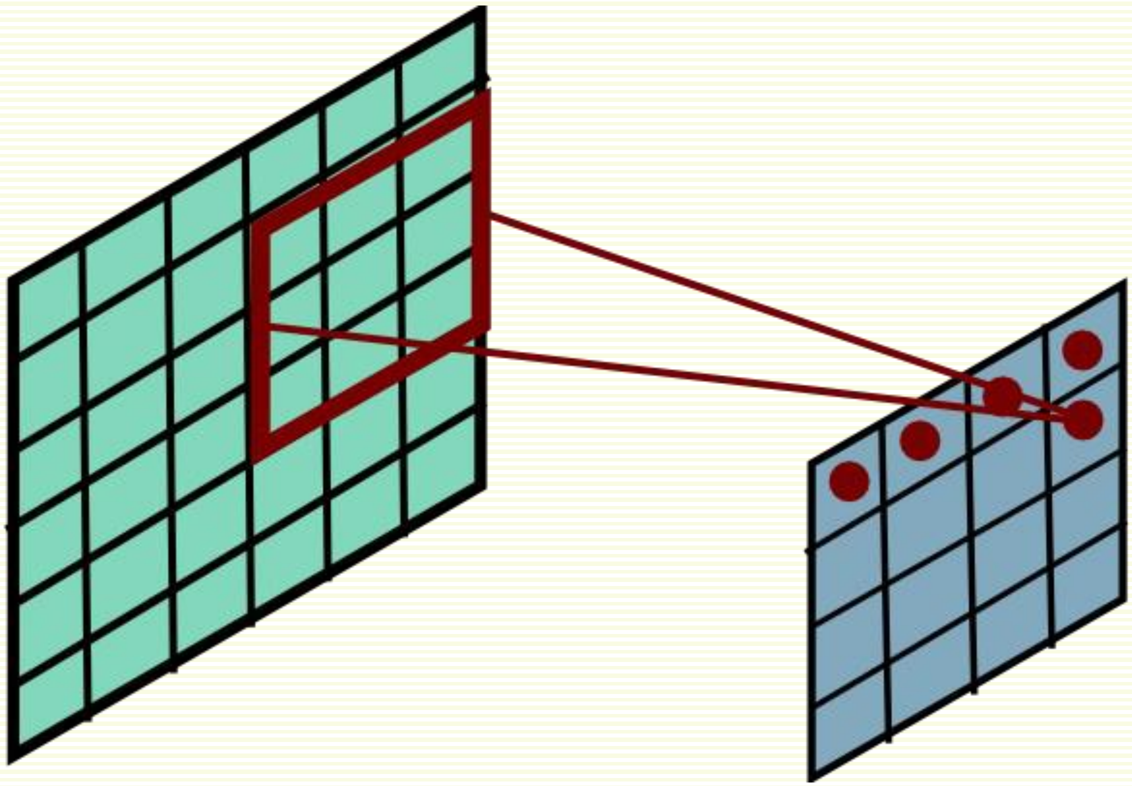
Convolutional Layer



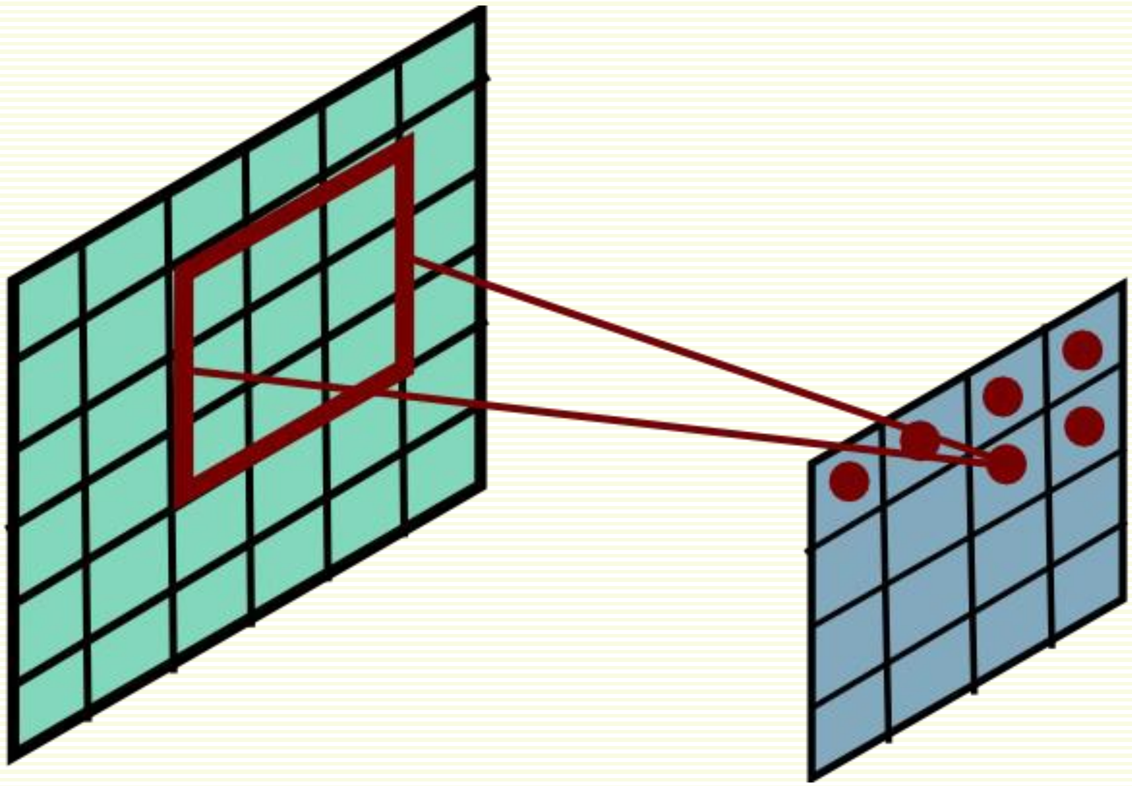
Convolutional Layer



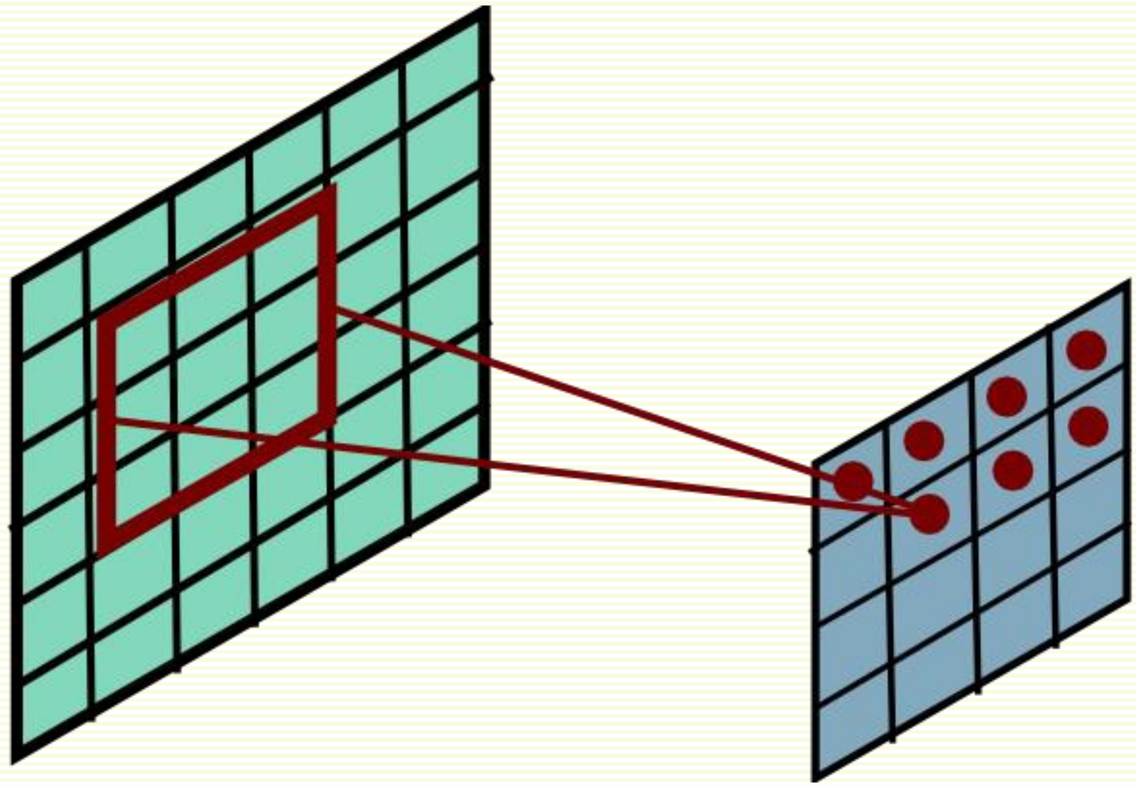
Convolutional Layer



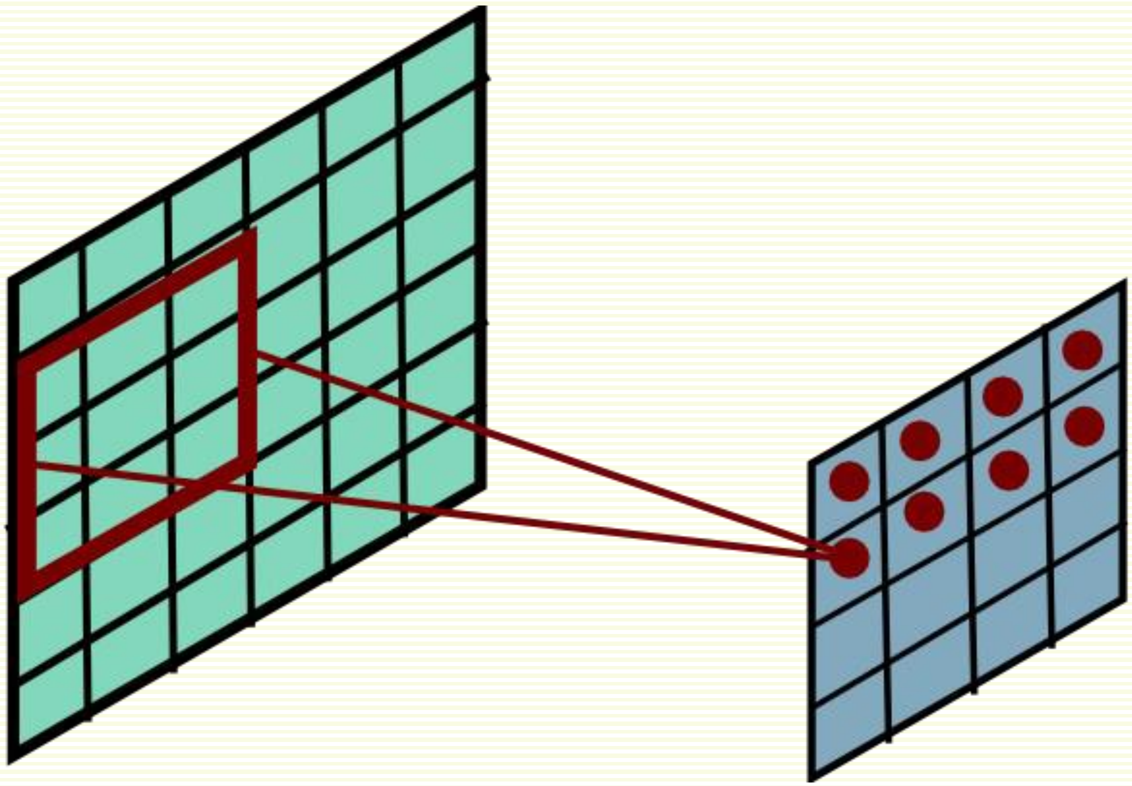
Convolutional Layer



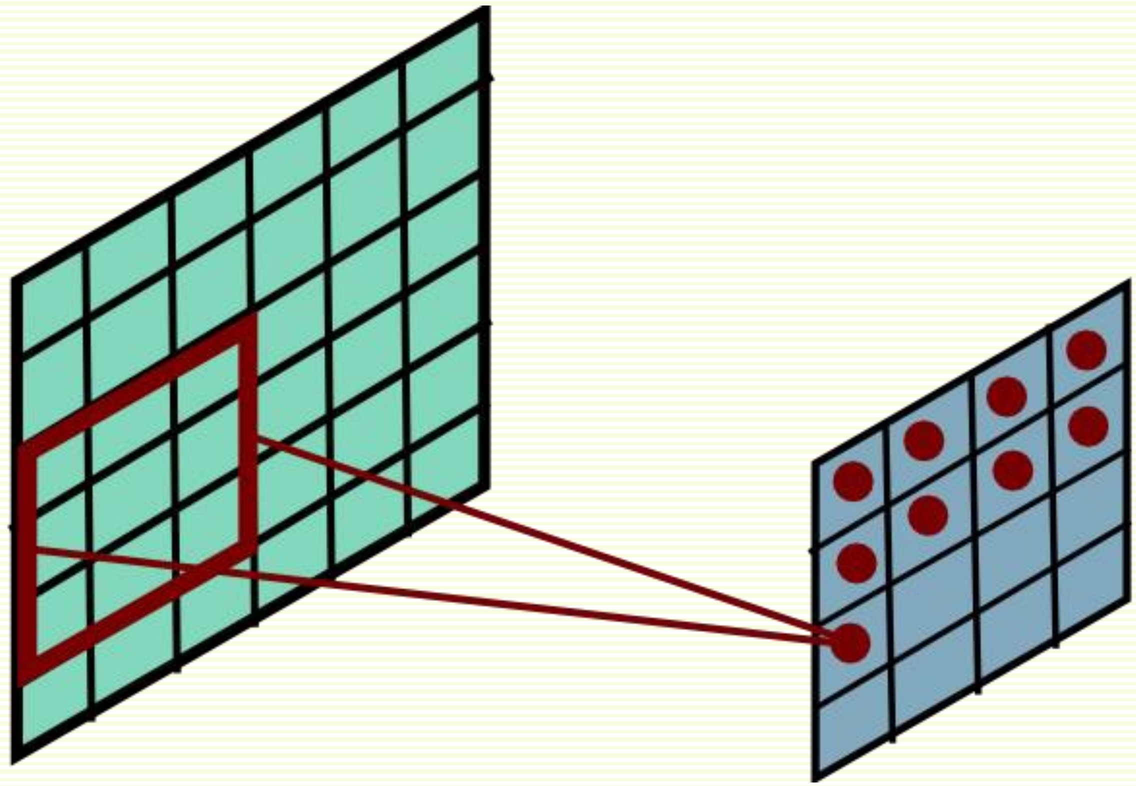
Convolutional Layer



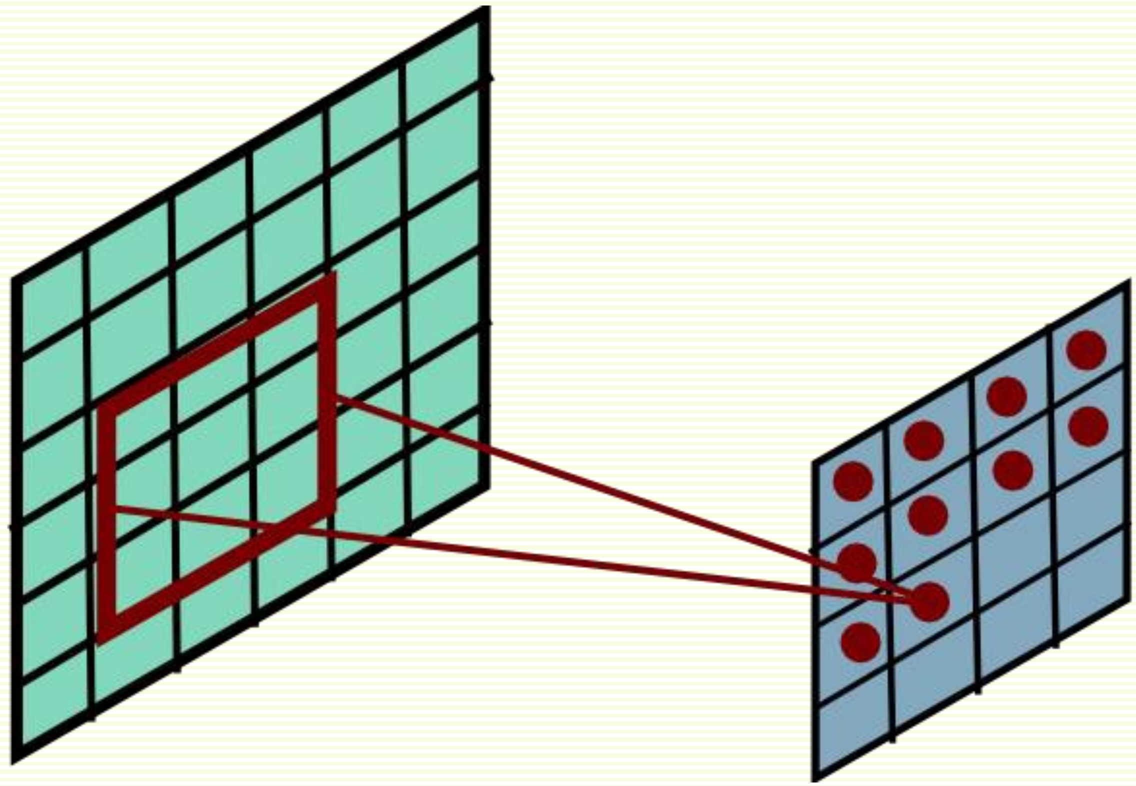
Convolutional Layer



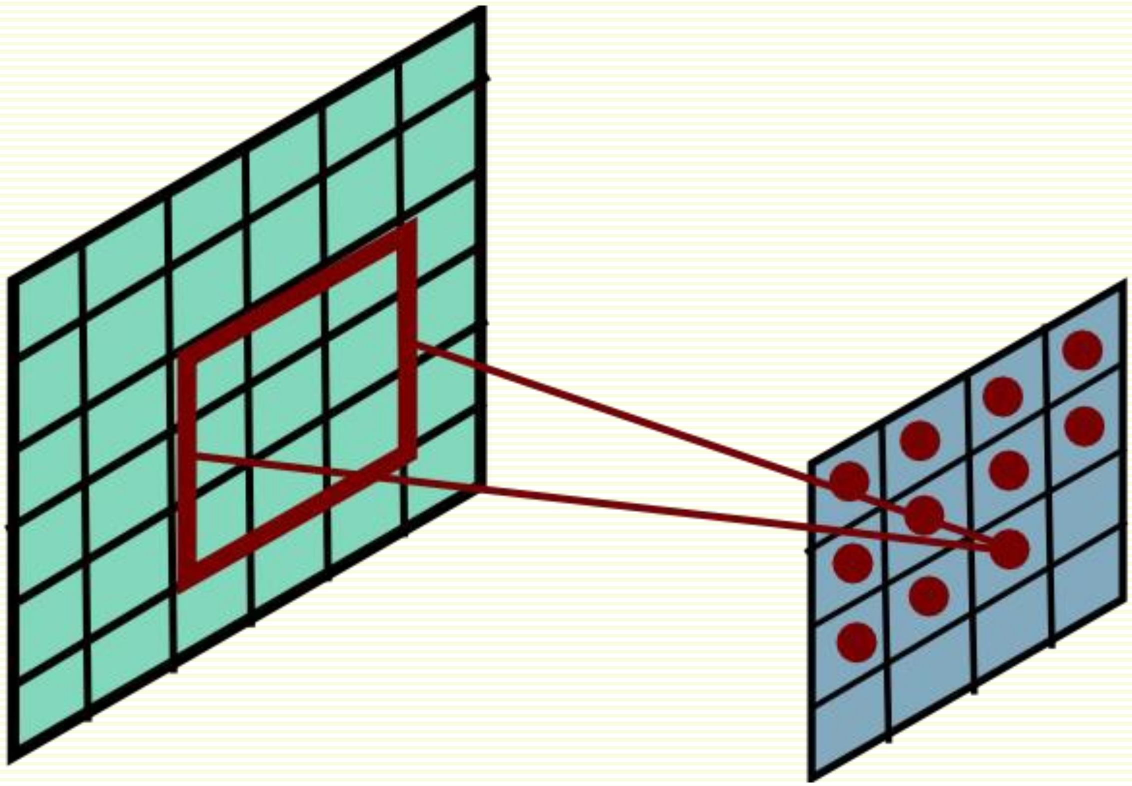
Convolutional Layer



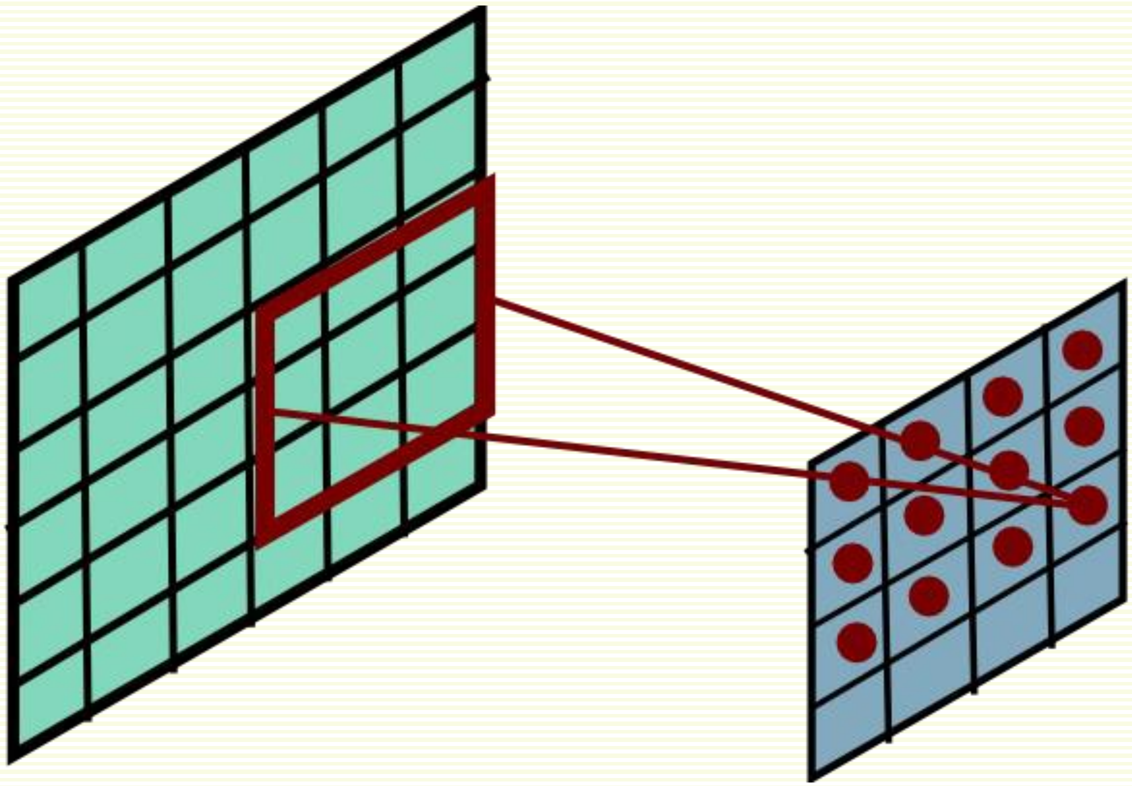
Convolutional Layer



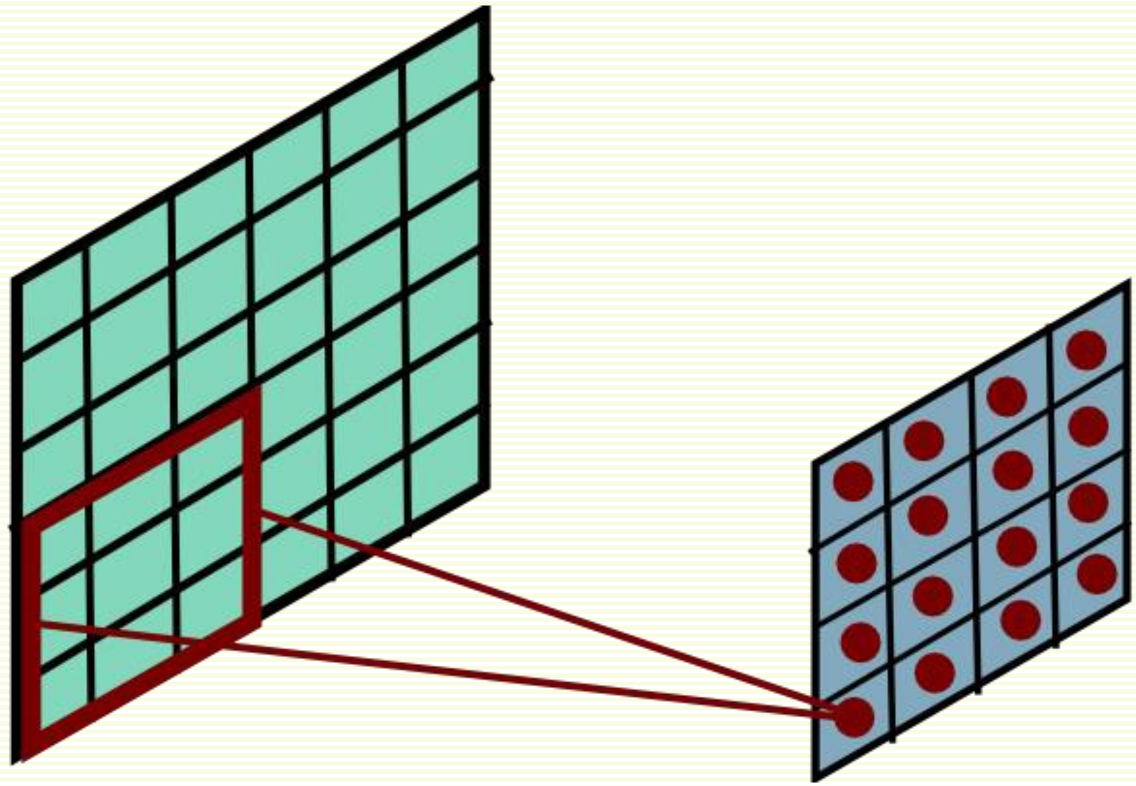
Convolutional Layer



Convolutional Layer

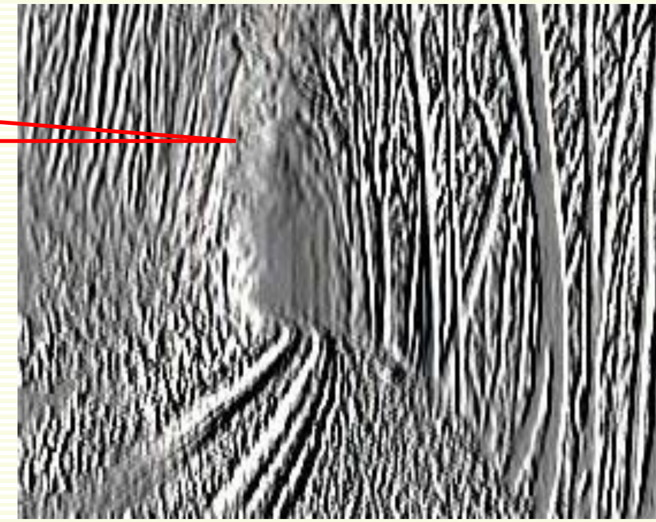


Convolutional Layer



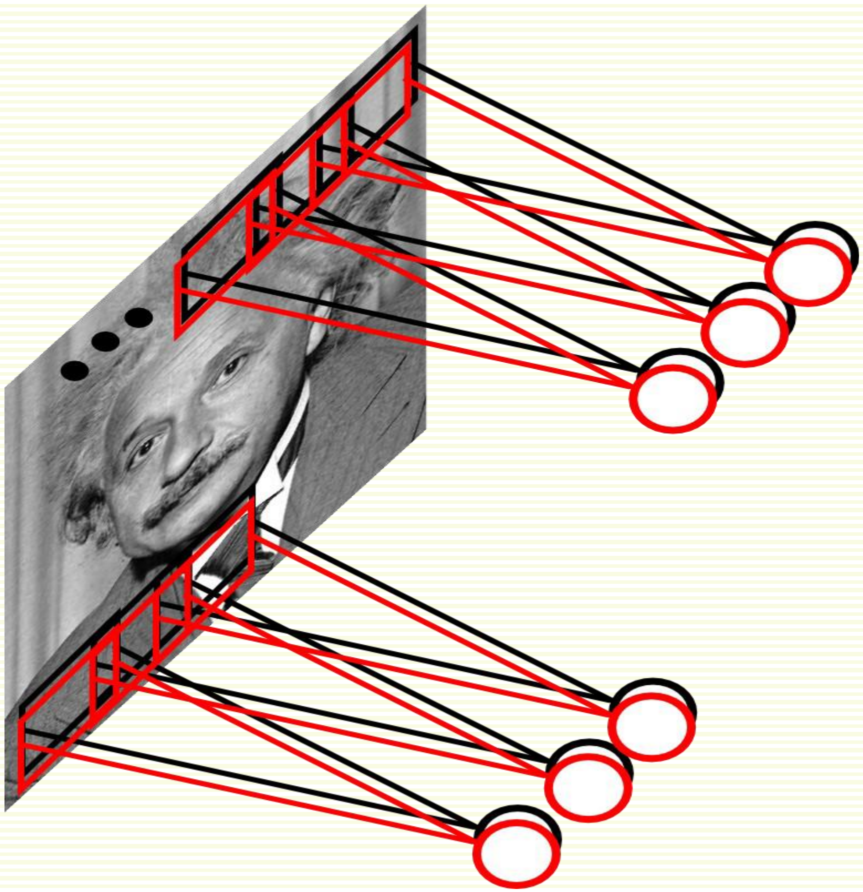
Convolutional Layer

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



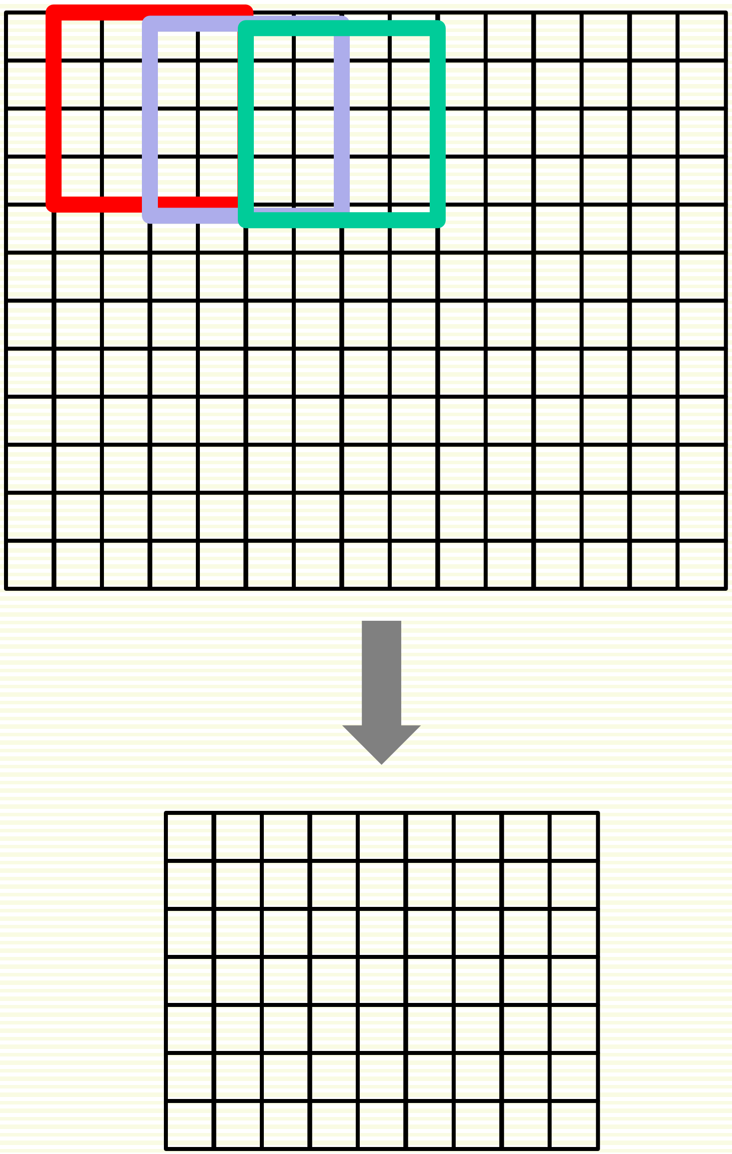
Convolutional Layer

- Each filter is responsible for one feature type
- Learn multiple filters
- Example:
  - 10x10 patch
  - 100 filters
  - only 10^4 parameters to learn
  - because parameters are shared between different locations



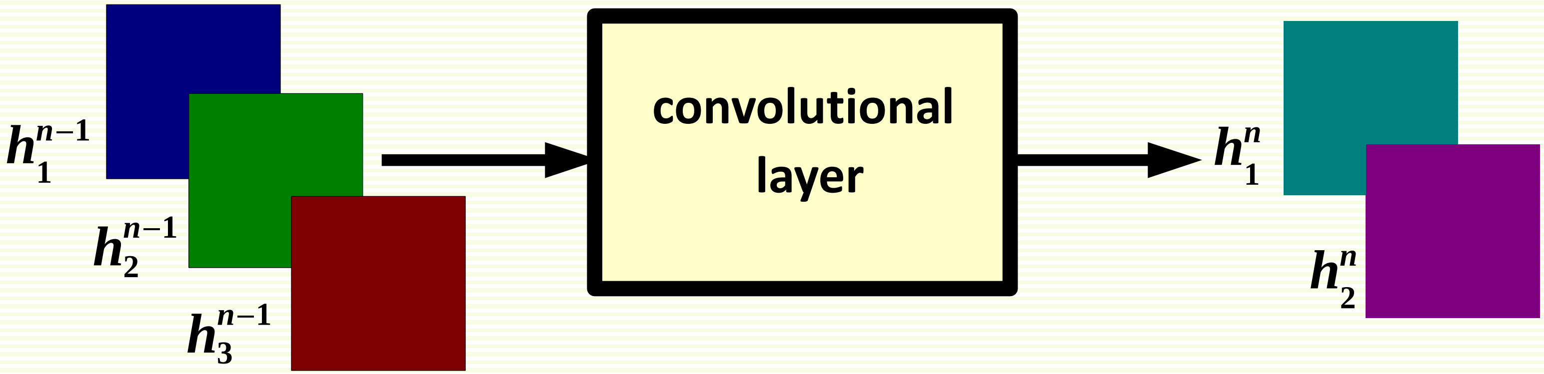
Convolutional Layer

- Can apply convolution to every other pixel, to reduce the number of parameters even further
- Example
  - stride = 2
  - apply convolution every second pixel
  - makes image twice smaller in each dimension



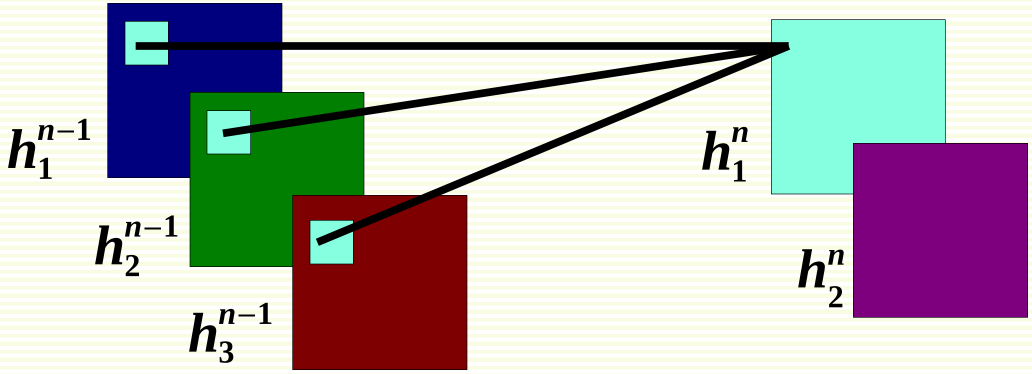
Convolutional Layer

- Each layer **h** is a **d**-dimensional *image* or *map* **r** x **c** x **d**
- Thus perform **d**-dimensional convolution
- If using **d'** filters, next layer is a map of size **r'** x **c'** x **d'**
- Example with **d** = 3 and **d'** = 2 (i.e. 2 filters)
- **r'** and **c'** depend on whether convolution crops image border and the stride of convolution



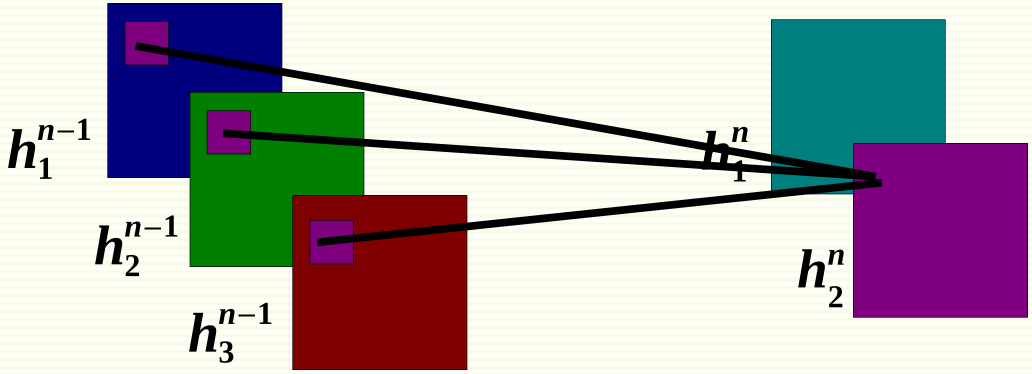
Convolutional Layer

- Example with **d** = 3 and **d’** = 2 (i.e. 2 filters)
- Applying the first filter



Convolutional Layer

- Example with **d** = 3 and **d'** = 2 (i.e. 2 filters)
- Applying the second filter



Convolutional Layer

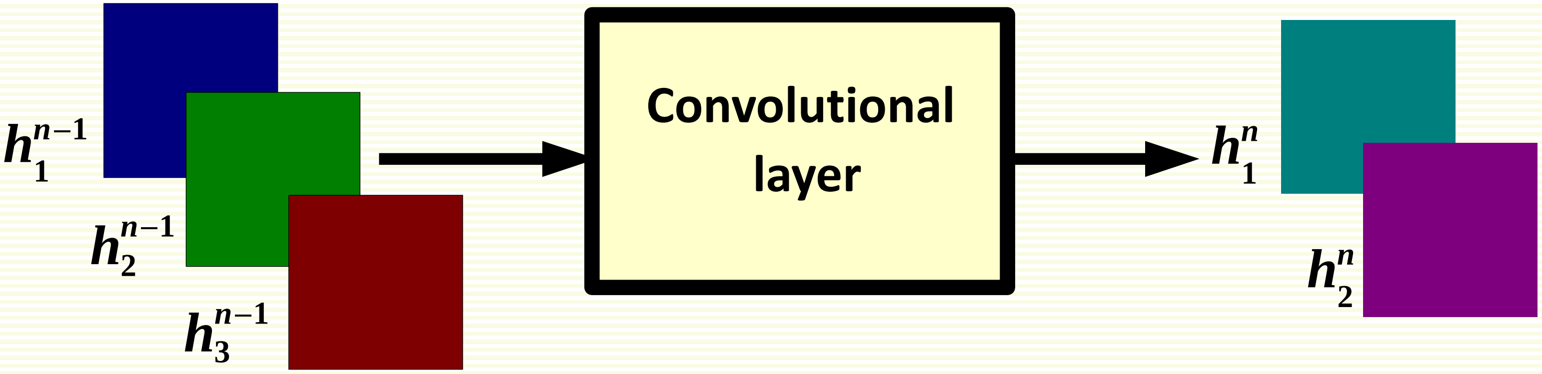
- Formula for convolution application to **K** dimensional layer $\mathbf{h}^{n-1}$
  - Also with application of ReLu activation function

kernel

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

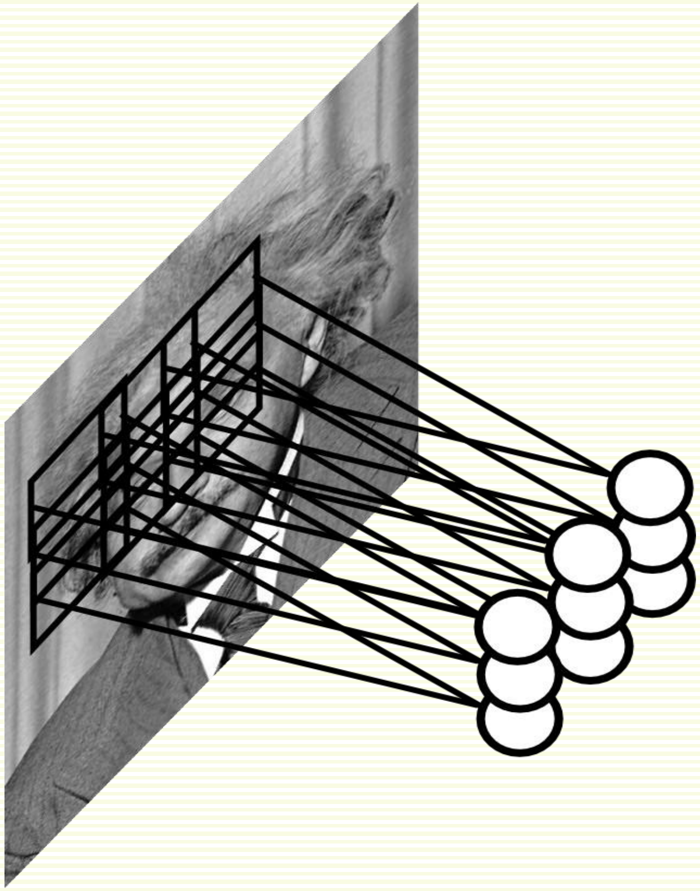
output feature map

input feature map



Pooling Layer

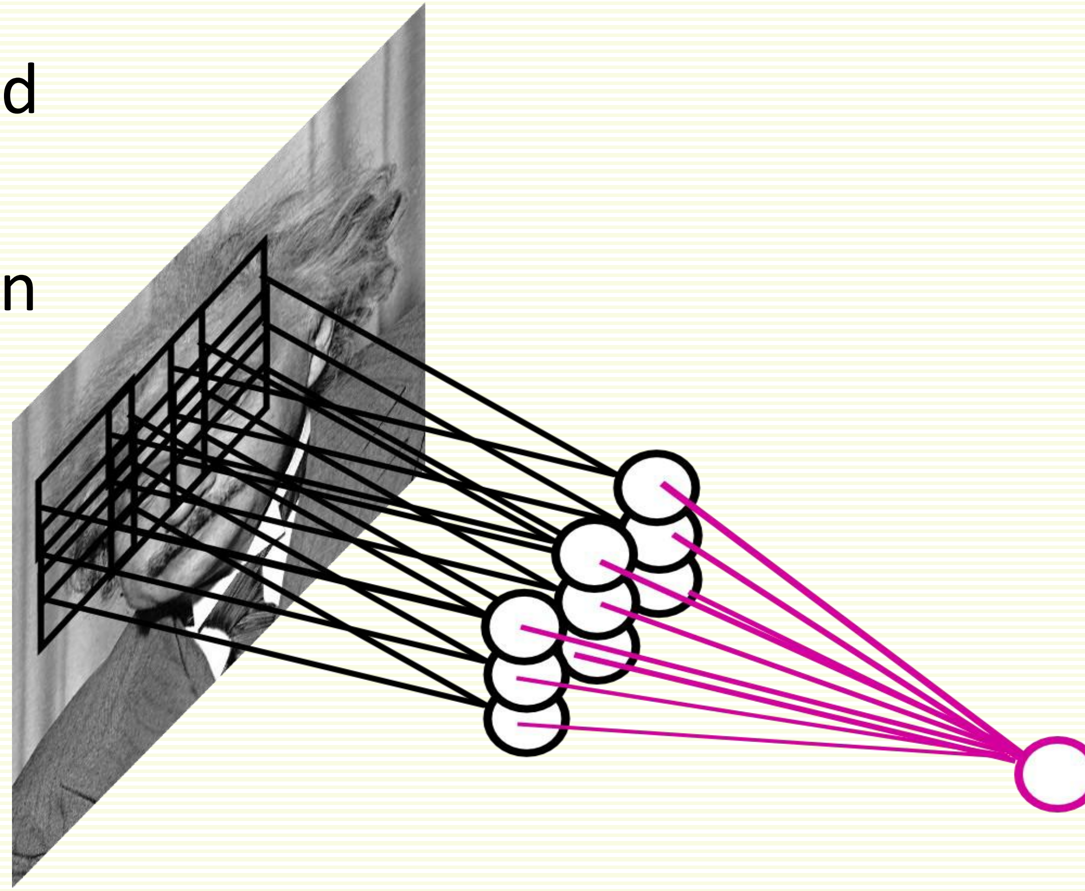
- Say a filter is an eye detector
- Want to detection to be robust to precise eye location



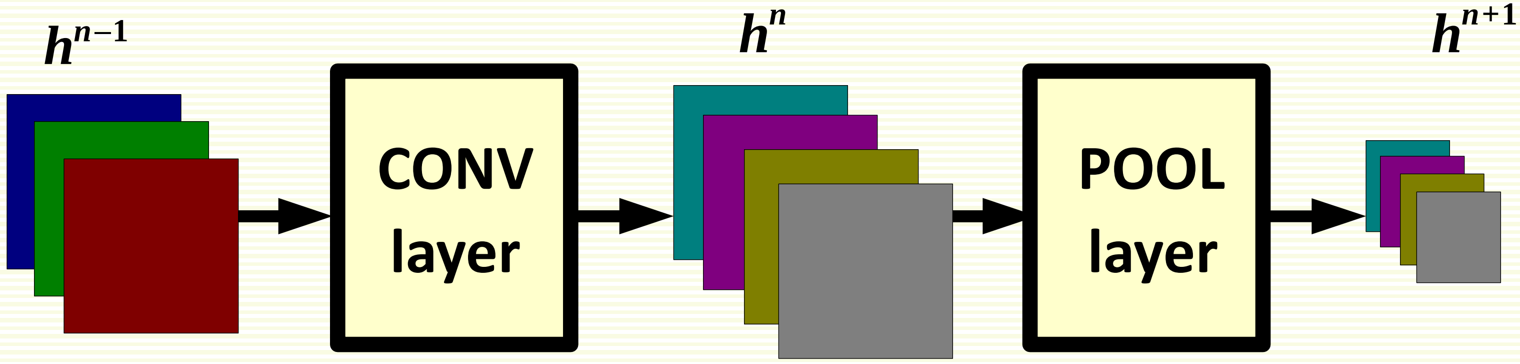
Pooling Layer

- *Pool* filter responses at different locations gain robustness to exact spatial location
  - pooling could be taking max, average, etc.

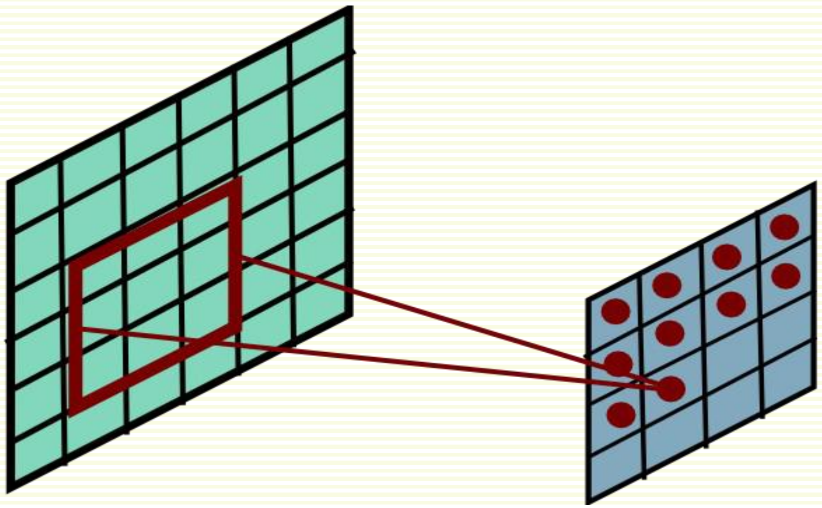
- Usually pooling applied with stride > 1
- This reduces resolution of output map
- But we already lost resolution (precision) by pooling



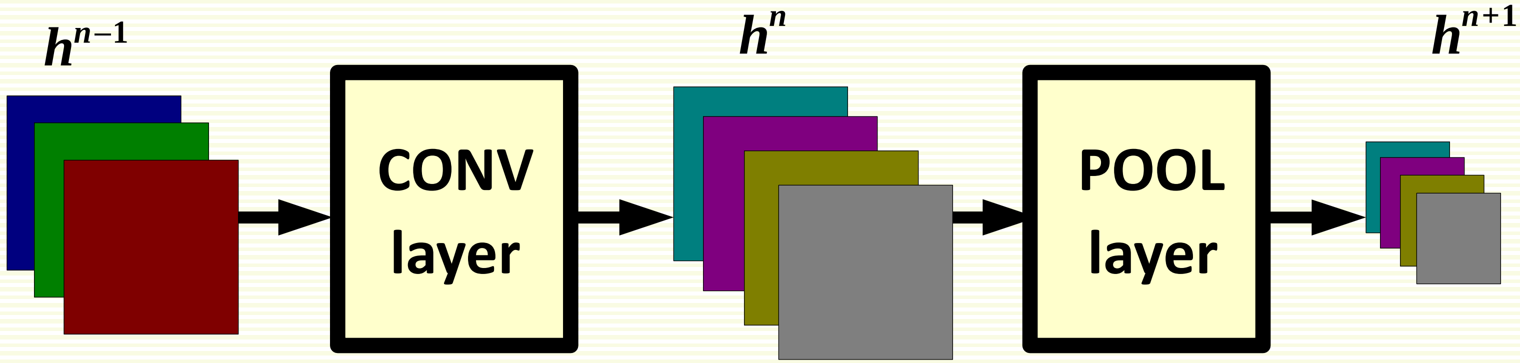
Pooling Layer: Receptive Field Size



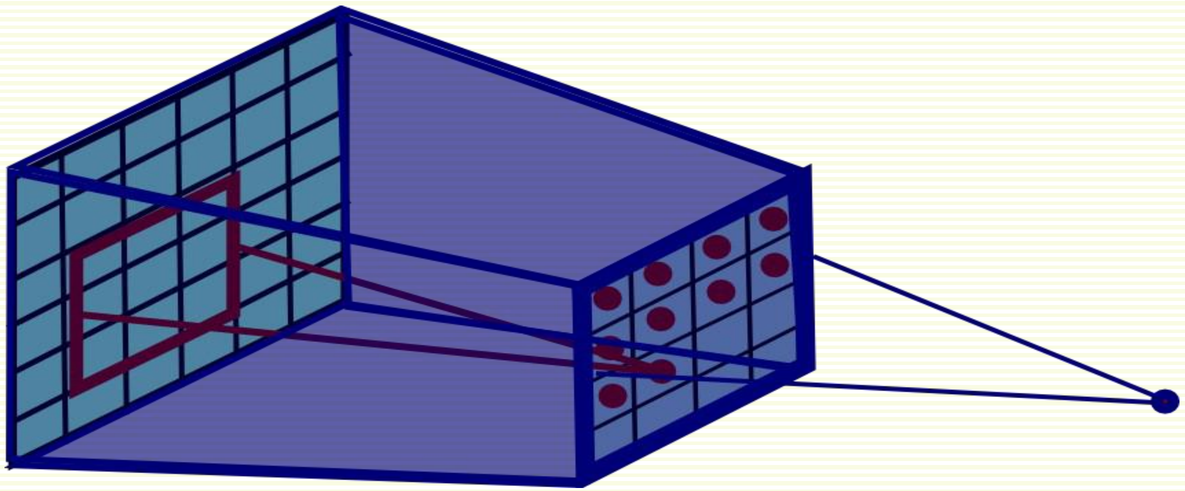
- If convolution filters have size **K** x **K** and stride 1, and pooling layer has pools of size **P** x **P**, then each unit in pooling layer depends on patch (in preceding convolution layer) of size (**P**+**K**-1) x (**P**+**K**-1)



Pooling Layer: Receptive Field Size



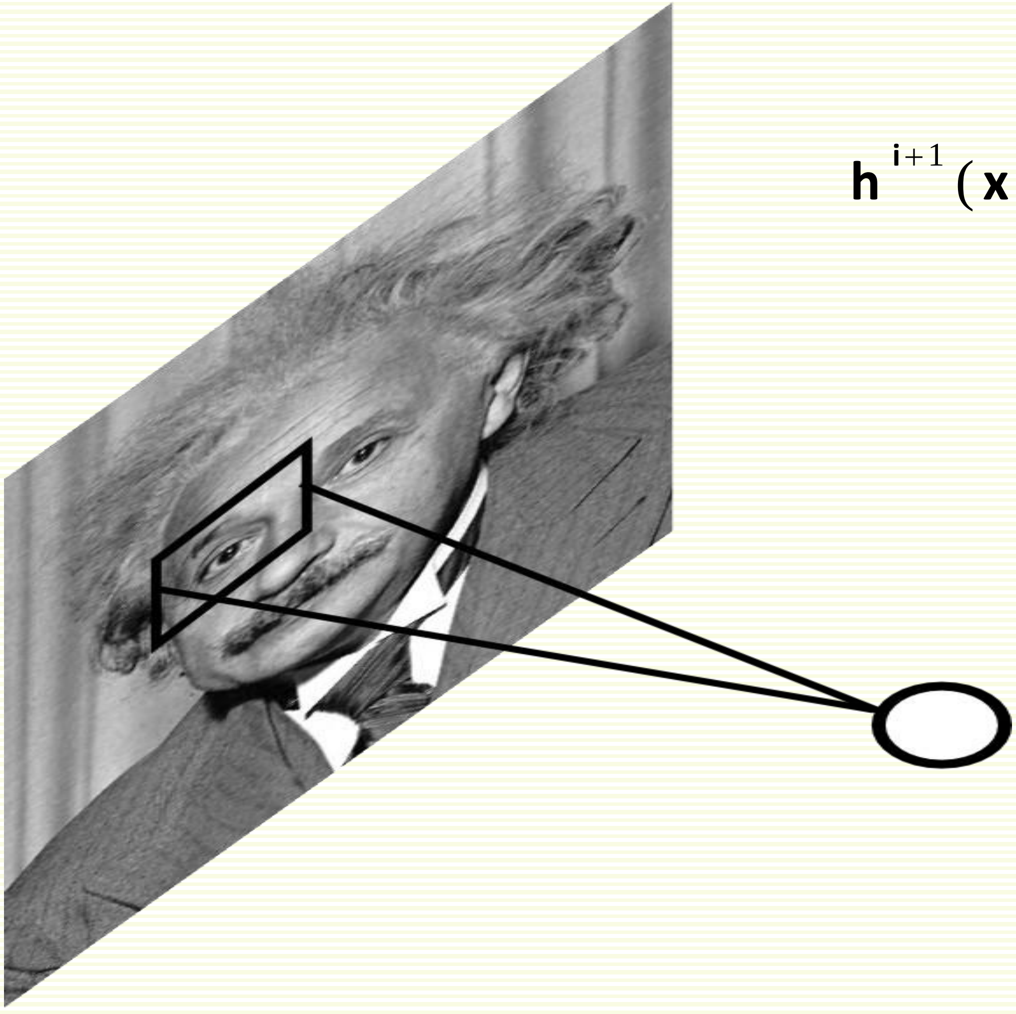
- If convolution filters have size KxK and stride 1, and pooling layer has pools of size PxP, then each unit in the pooling layer depends upon a patch (in the preceding convolution layer) of size (P+K-1)x(P+K-1)



Problem with Pooling

- After several levels of pooling, we have lost information about the precise positions of things
- This makes it impossible to use the precise spatial relationships between high-level parts for recognition.

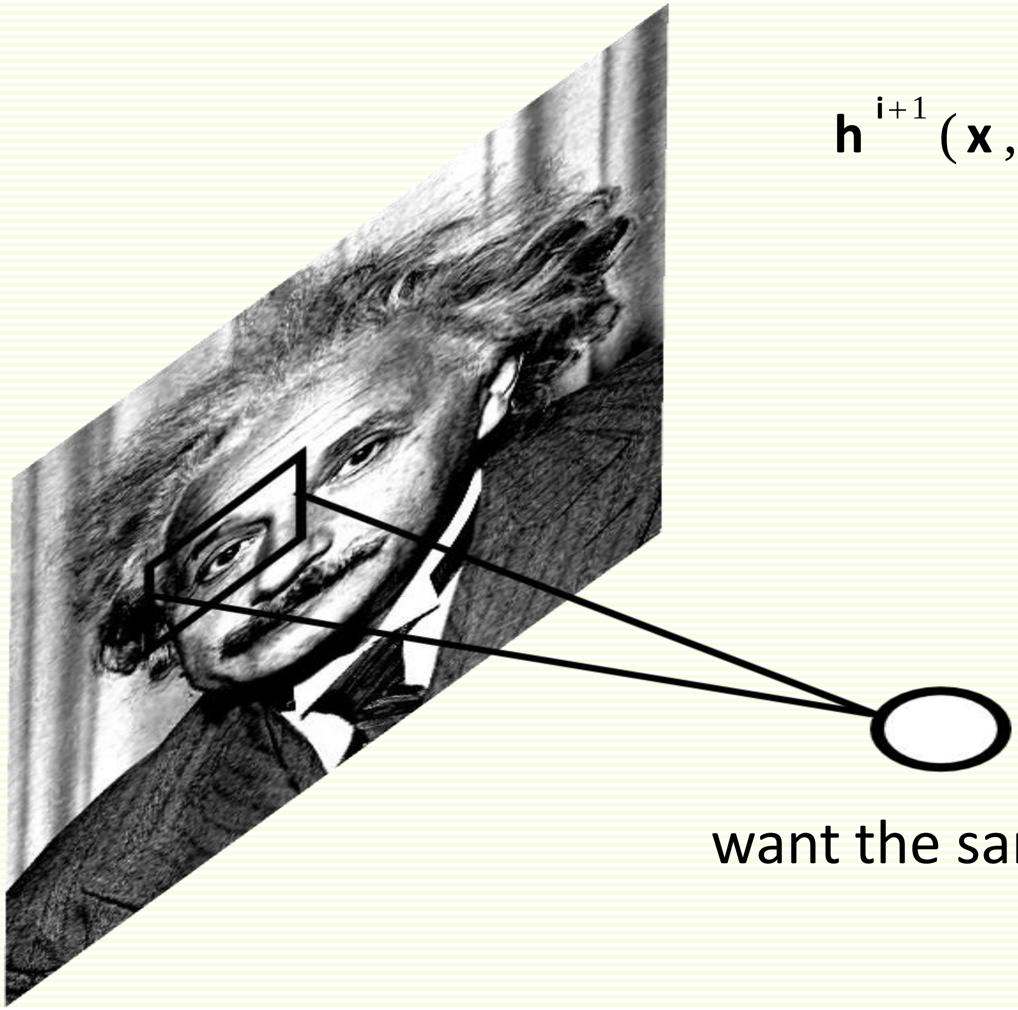
Local Contrast Normalization



$$\mathbf{h}^{\mathbf{i}+1}(\mathbf{x},\mathbf{y}) = \frac{\mathbf{h}^{\mathbf{i}}(\mathbf{x},\mathbf{y}) - \mu^{\mathbf{i}}(\mathbf{N}(\mathbf{x},\mathbf{y}))}{\sigma^{\mathbf{i}}(\mathbf{N}(\mathbf{x},\mathbf{y}))}$$

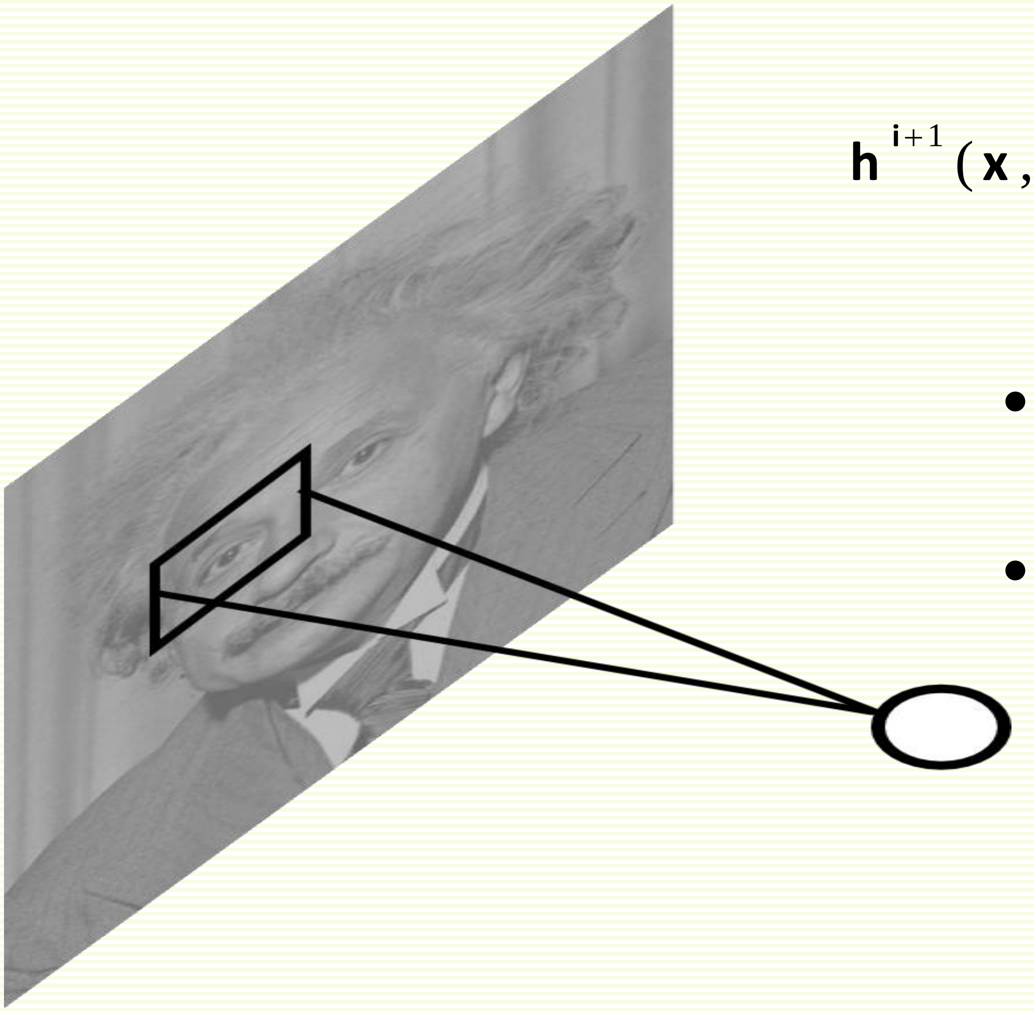
Local Contrast Normalization

$$h^{i+1}(x, y) = \frac{h^{i}(x, y) - \mu^{i}(N(x, y))}{\sigma^{i}(N(x, y))}$$



want the same response

Local Contrast Normalization

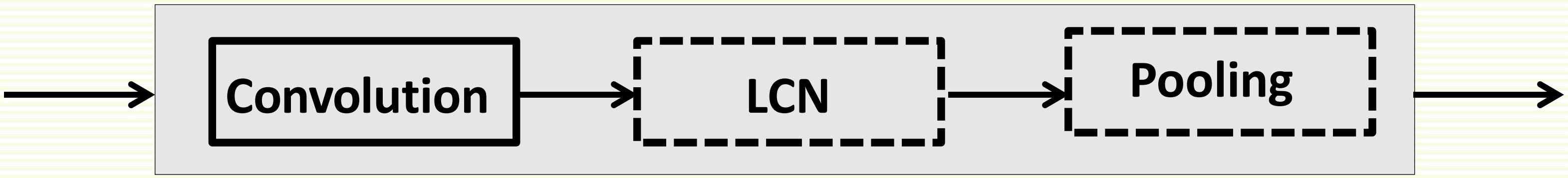


$$\mathbf{h}^{\mathbf{i}+1}(\mathbf{x},\mathbf{y}) = \frac{\mathbf{h}^{\mathbf{i}}(\mathbf{x},\mathbf{y}) - \mu^{\mathbf{i}}(\mathbf{N}(\mathbf{x},\mathbf{y}))}{\sigma^{\mathbf{i}}(\mathbf{N}(\mathbf{x},\mathbf{y}))}$$

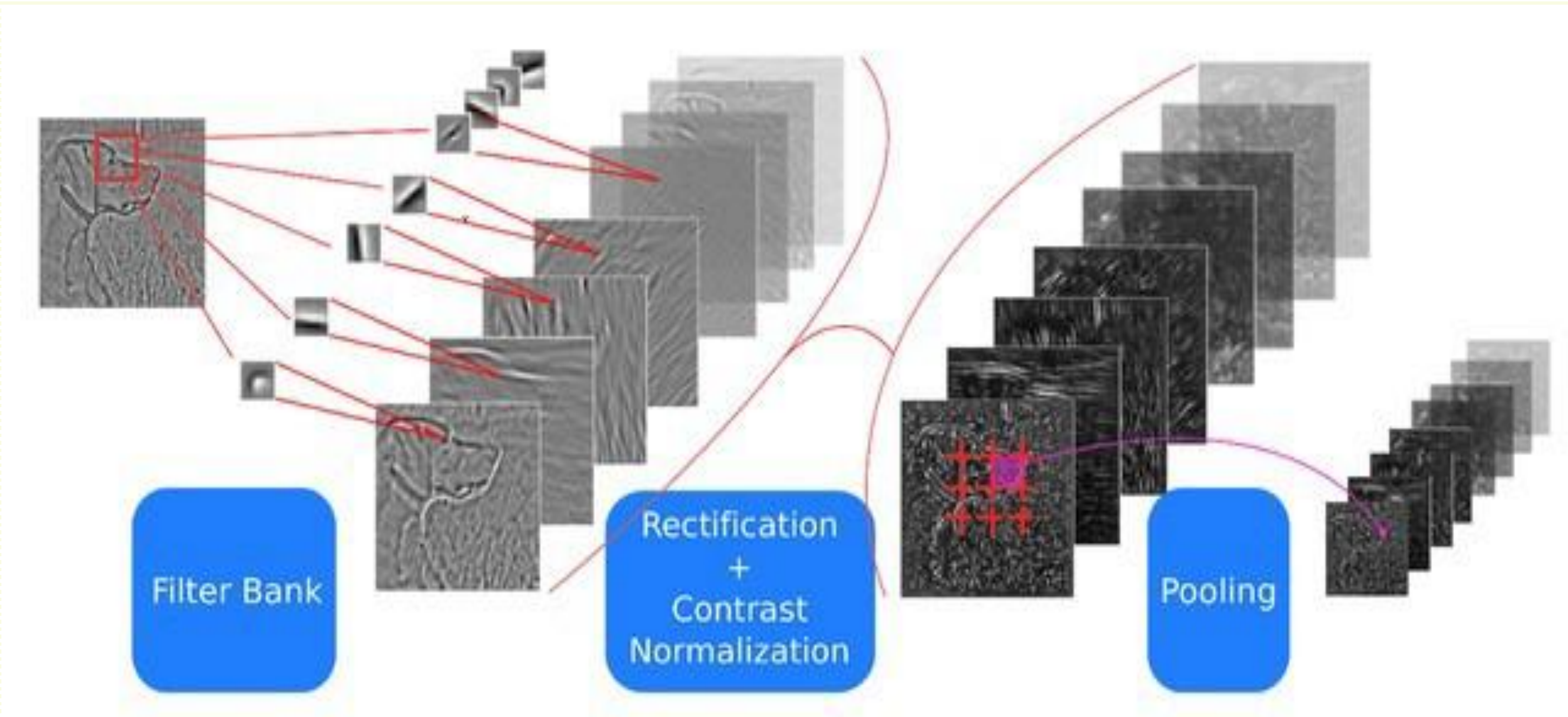
- Performed also across features and in higher layers
- Effects
    - Improves invariance
    - Improves optimization

ConvNets: Typical Stage

One Stage (zoom)

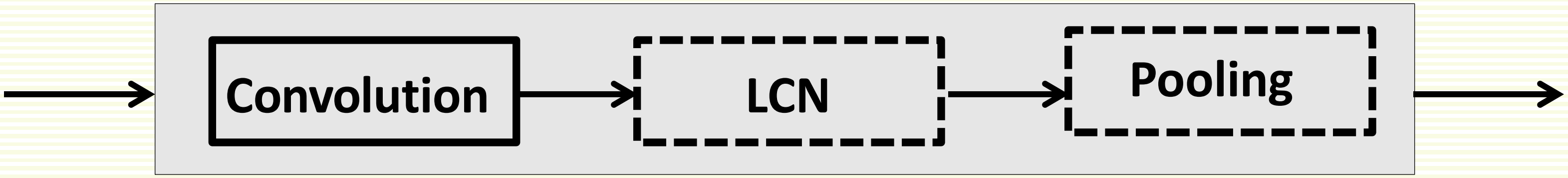


Conceptually similar to: SIFT, HoG, etc.

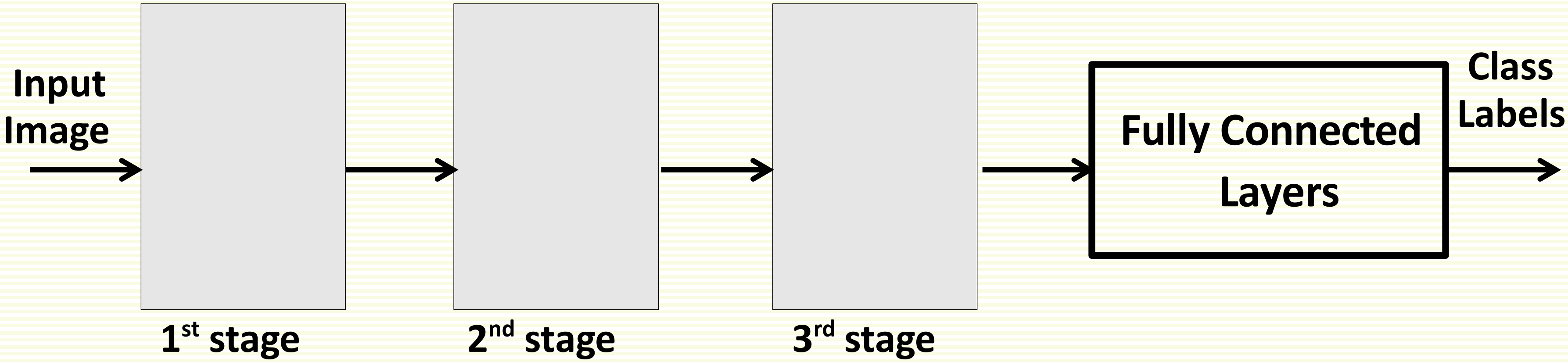


Typical Architecture

One Stage (zoom)



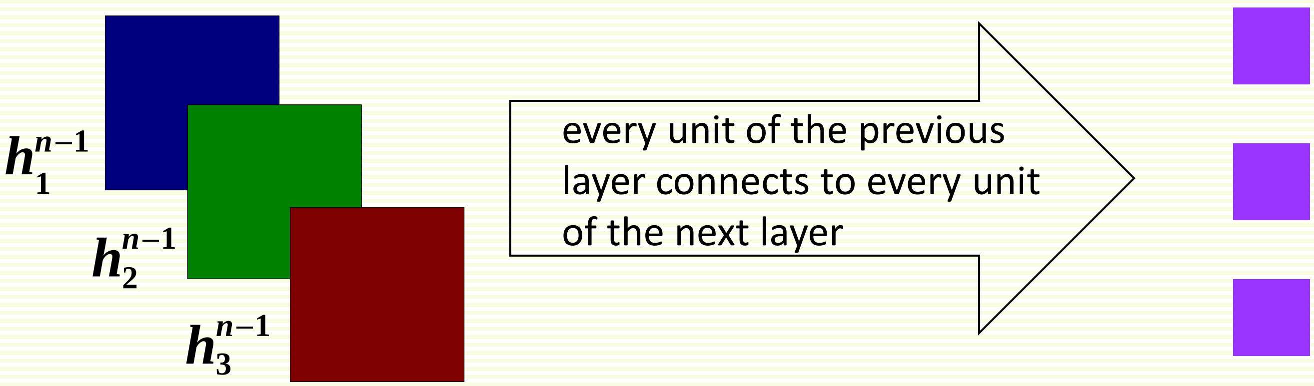
Whole System



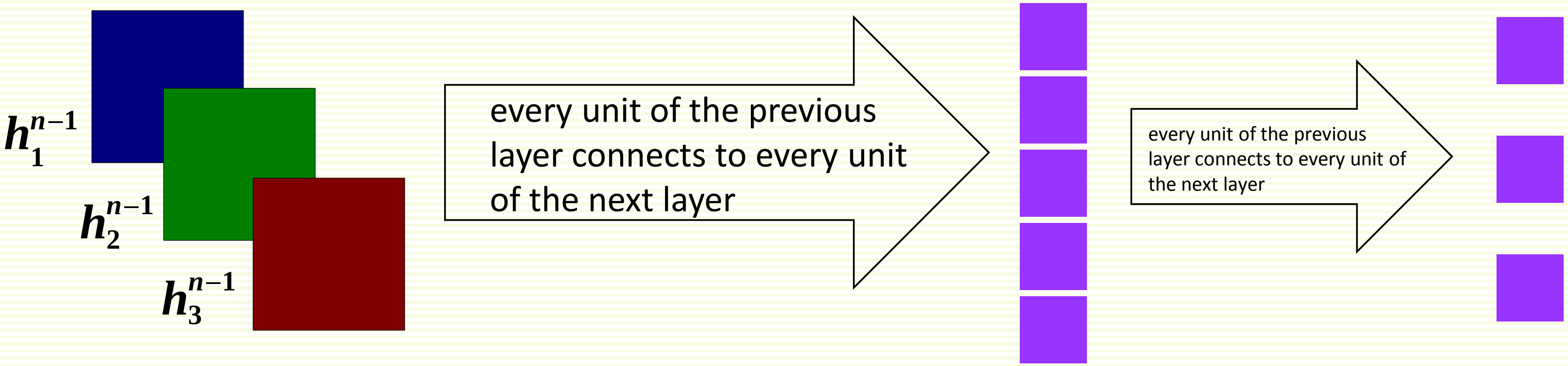
Conceptually similar to: SIFT → K-Means → Pyramid Pooling → SVM

Fully Connected Layer

- Can have just one fully connected layer
- Example for 3-class classification problem



- Can have many fully connected layer
- Example for 3-class classification problem

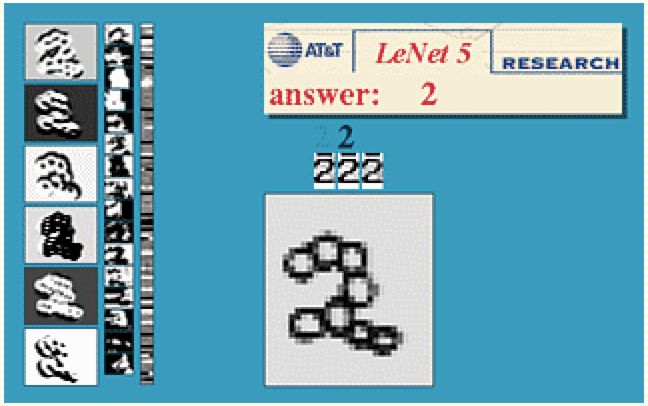


ConvNets: Training

- All Layers are differentiable
- Use standard back-propagation (gradient descent)
- At test time, run only in forward mode

Conv Nets: Character Recognition

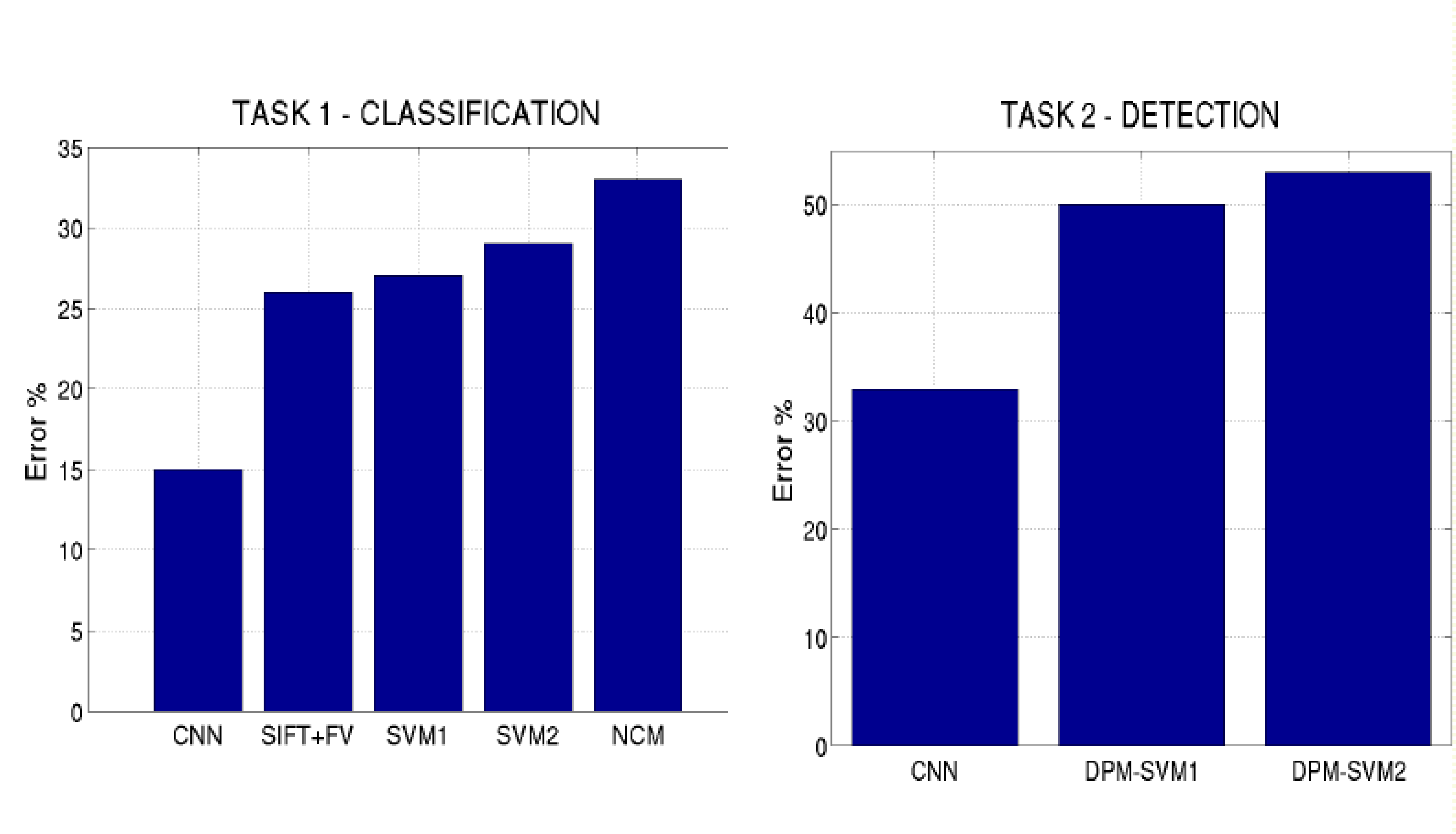
- http://yann.lecun.com/exdb/lenet/index.html



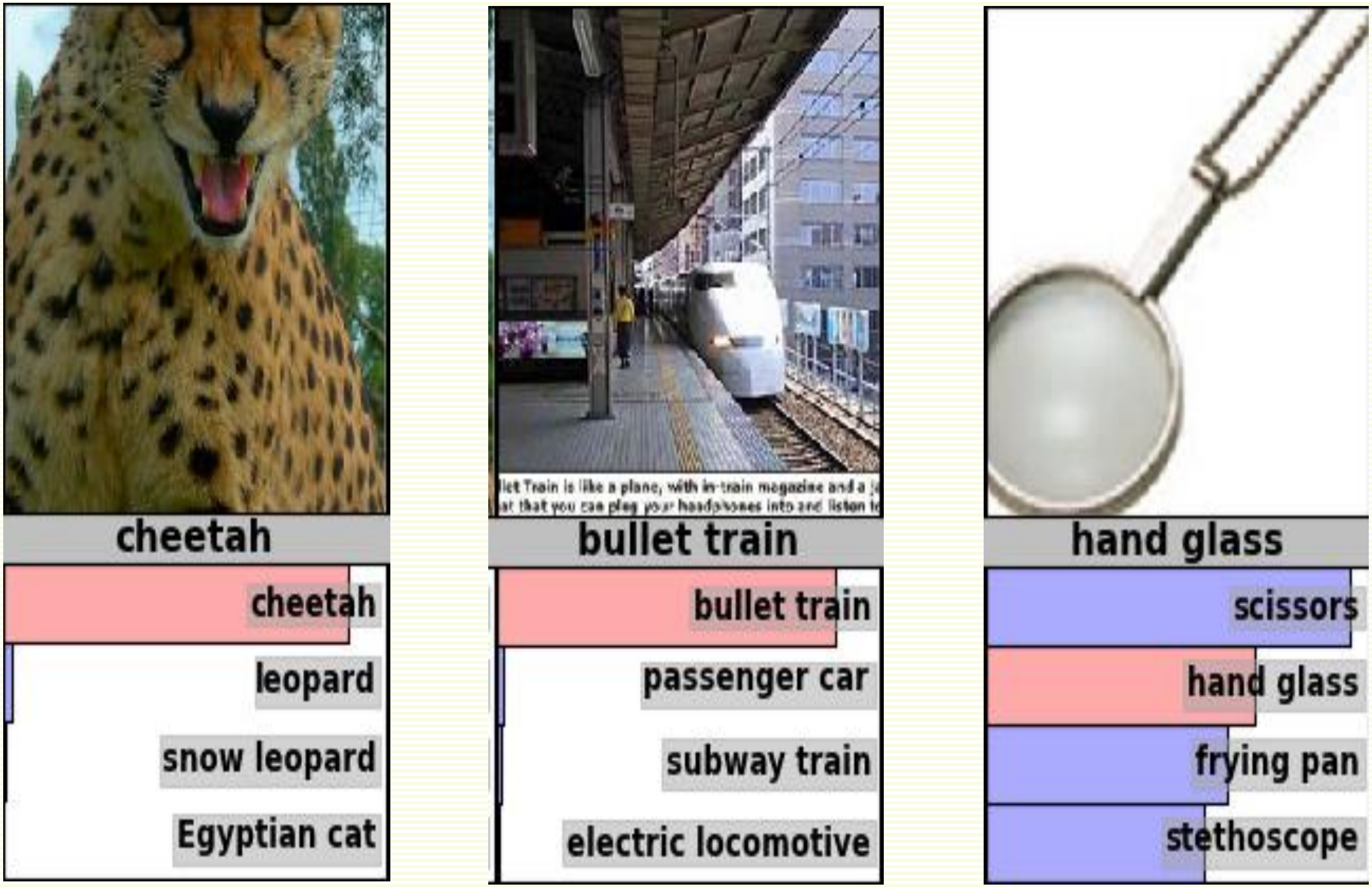
ConvNet for ImageNet

- Krizhevsky et.al.(NIPS 2012) developed deep convolutional neural net of the type pioneered by Yann LeCun
- Architecture:
  - 7 hidden layers not counting some max pooling layers.
  - the early layers were convolutional.
  - the last two layers were globally connected.
- Activation function:
  - rectified linear units in every hidden layer
  - train much faster and are more expressive than logistic unit

Results: ILSVRC 2012

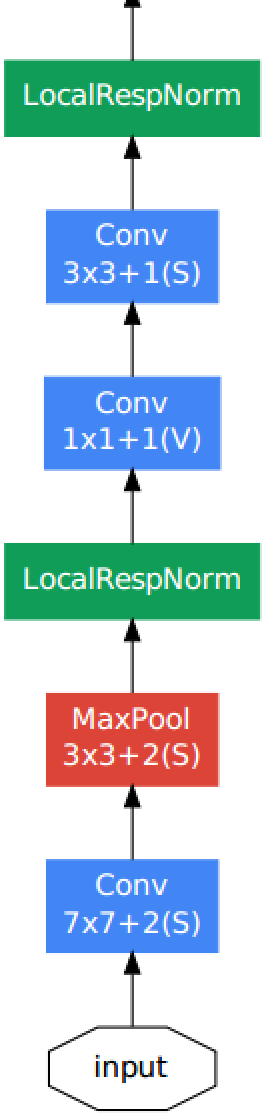


ConvNet on Image Classification



Tricks to Improve Generalization

- To get more data:
  - Use left-right reflections of the images
  - Train on random 224x224 patches from the 256x256 images
- At test time:
  - combine the opinions from ten different patches:
    - four 224x224 corner patches plus the central 224x224 patch
    - the reflections of those five patches
- Use *dropout* to regularize weights in the fully connected layers
  - half of the hidden units in a layer are randomly removed for each training example
- This stops hidden units from relying too much on other hidden units



Going Deeper with Convolutions
http://arxiv.org/abs/1409.4842

Difficulties in Supervised Training of Deep Networks

- Early layers do not get trained well
  - *diffusion of gradient* – error attenuates as it propagates to earlier layers
  - exacerbated since top layers can learn any task pretty well
  - thus error to earlier layers drops quickly as the top layers "mostly" solve the task
  - lower layers never get the opportunity to use their capacity to improve results, they just do a random feature map
  - need a way for early layers to do effective work
- Often not enough labeled data available while there may be lots of unlabeled data
  - can we use unsupervised/semi-supervised approaches to take advantage of the unlabeled data

Greedy Layer-Wise Training

- Greedy layer-wise training to insure lower layers learn

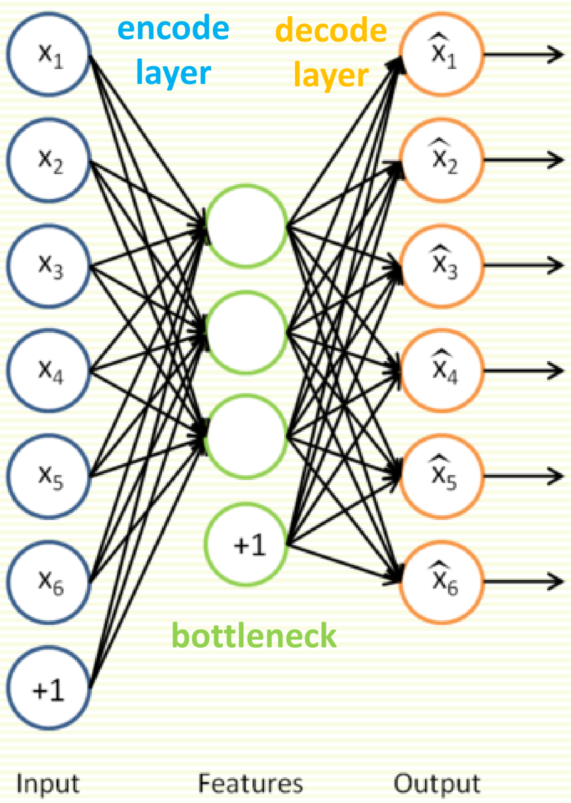
1. Train first layer using your data without the labels (unsupervised)
   - we do not know targets at this level anyway
   - can use the more abundant unlabeled data which is not part of the training set
2. Freeze the first layer parameters and start training the second layer using the output of the first layer as the unsupervised input to the second layer
3. Repeat this for as many layers as desired
   - This builds our set of robust features
4. Use the outputs of the final layer as inputs to a supervised layer/model and train the last supervised layer(s)
   - leave early weights frozen
5. Unfreeze all weights and fine tune the full network by training with a supervised approach, given the pre-processed weight settings

Greedy Layer-Wise Training

- Greedy layer-wise training avoids many of the problems of trying to train a deep net in a supervised fashion
  - Each layer gets full learning focus in its turn since it is the only current "top" layer
  - Can take advantage of the unlabeled data
  - When you finally tune the entire network with supervised training the network weights have already been adjusted so that you are in a good error basin and just need fine tuning This helps with problems of
    - ineffective early layer learning
    - deep network local minima

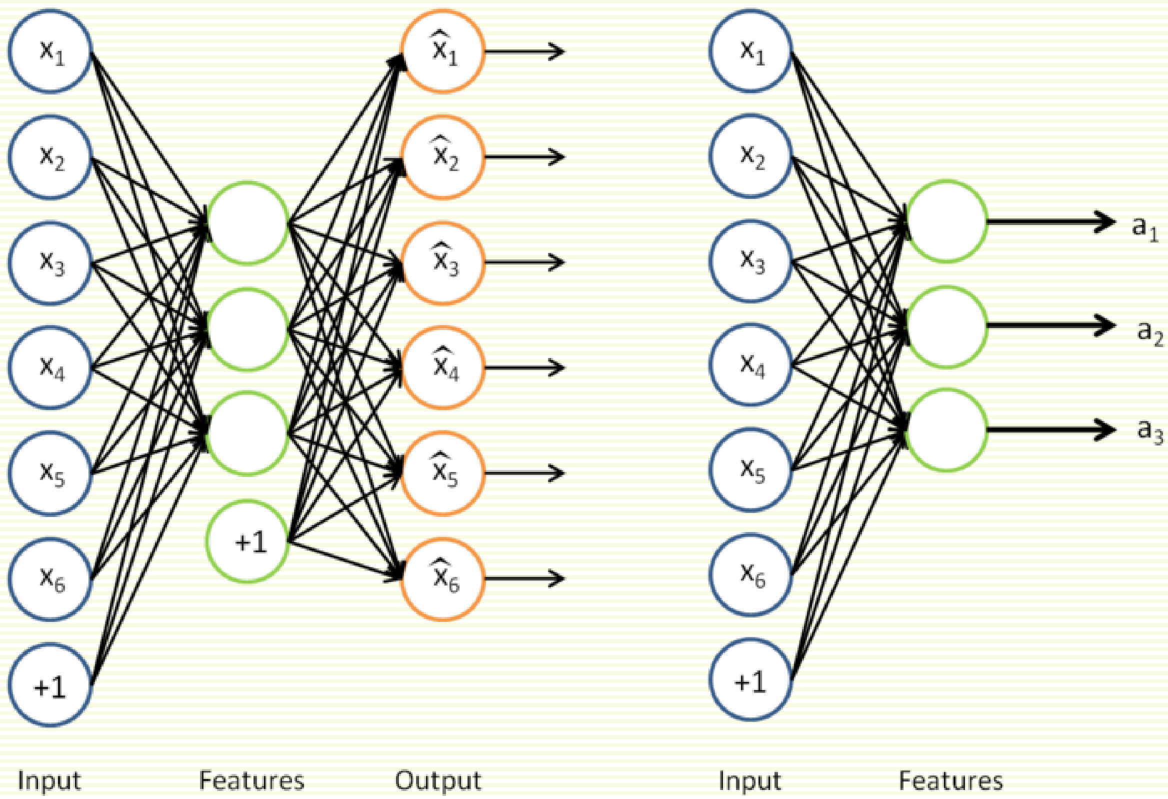
Auto-Encoders

- Unsupervised learning to discover generic features of the data
  - Learn identity function **f(x,w)** = **x**
  - through learning important sub-features, not by just passing through data
  - Constrain layer 2 to have less units than the input layer, or to be sparse



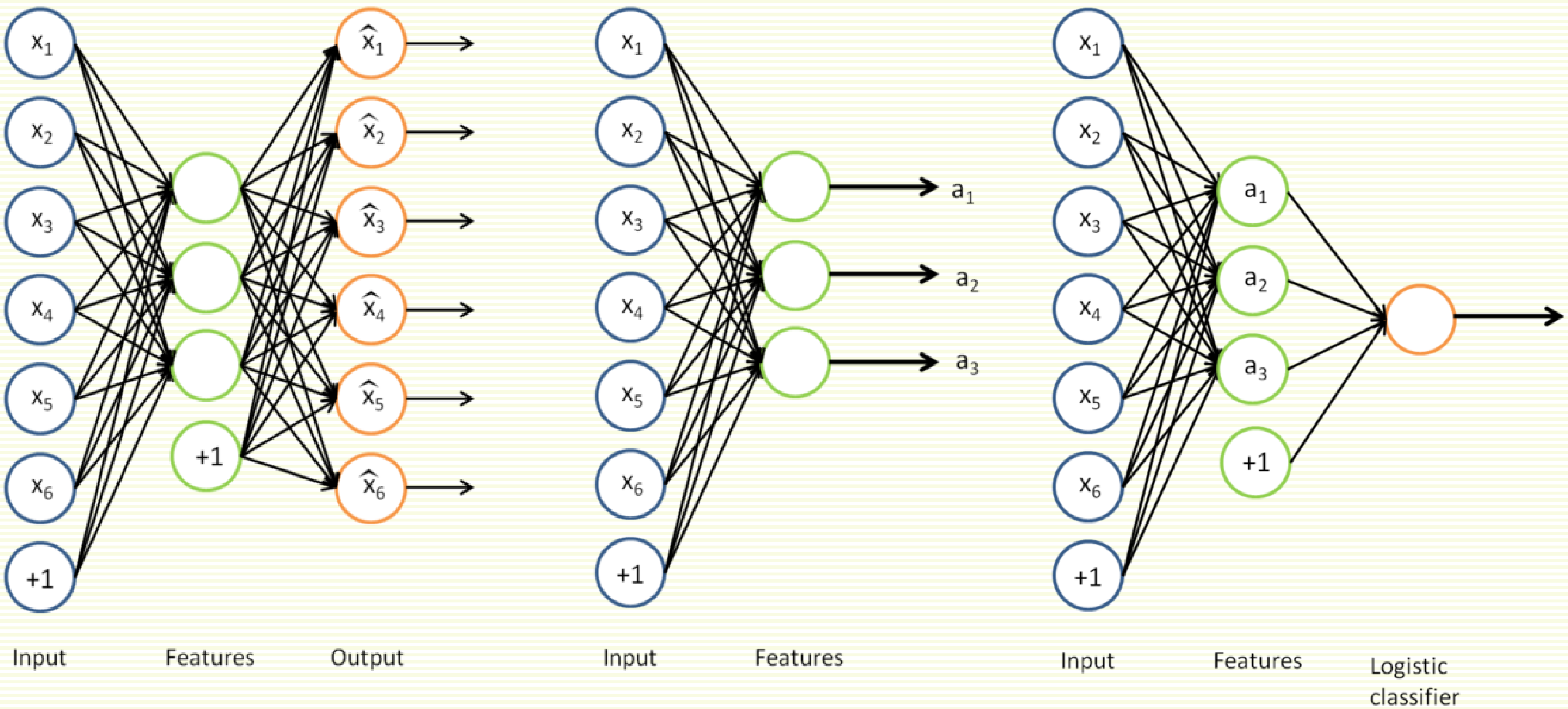
Auto-Encoders

- Layer 2 units are the new learned features



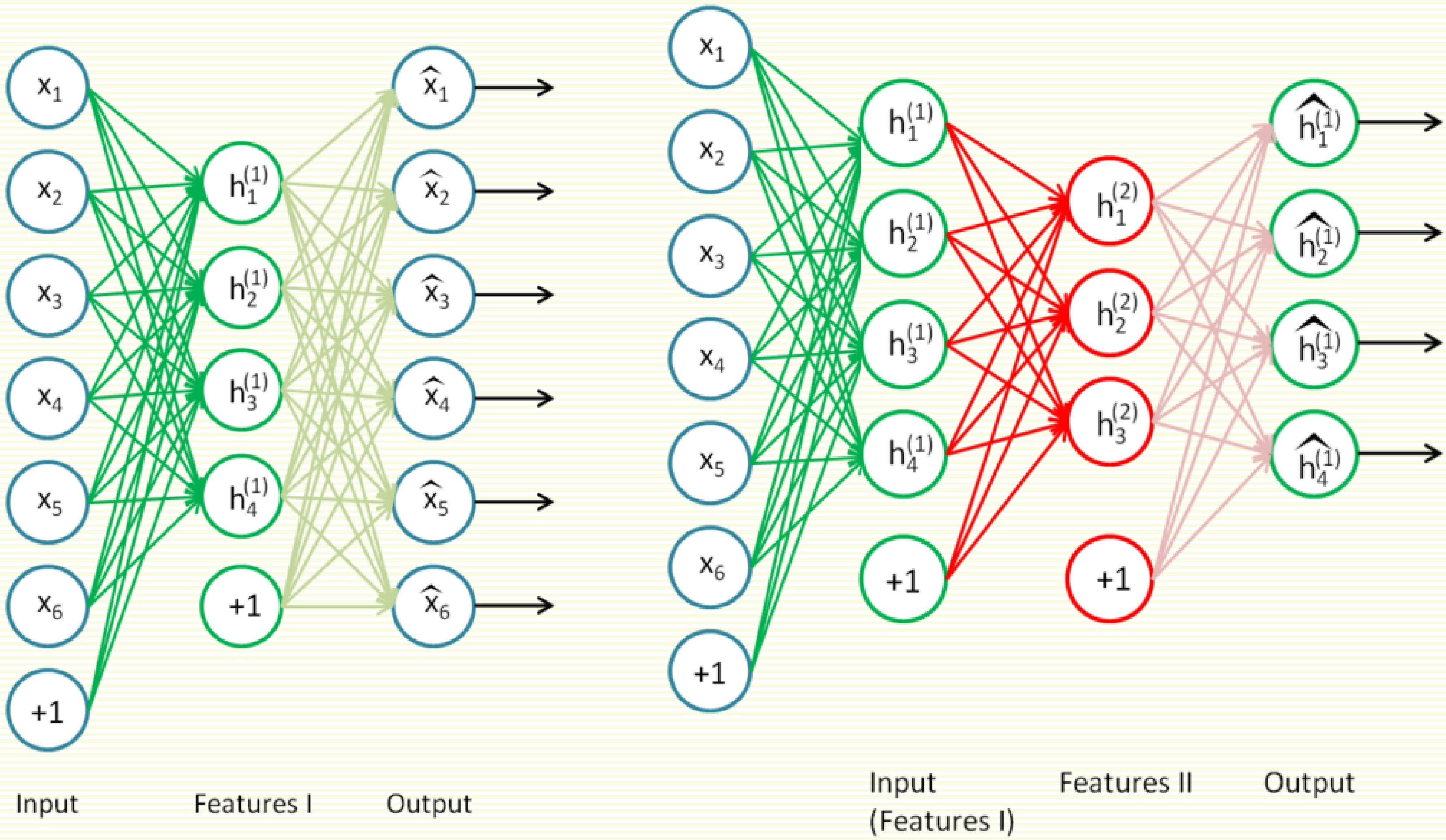
Auto-Encoders

- Layer 2 units are the new learned features
- Can fix their weights, replace decode layer with supervised learning layer and do supervised learning



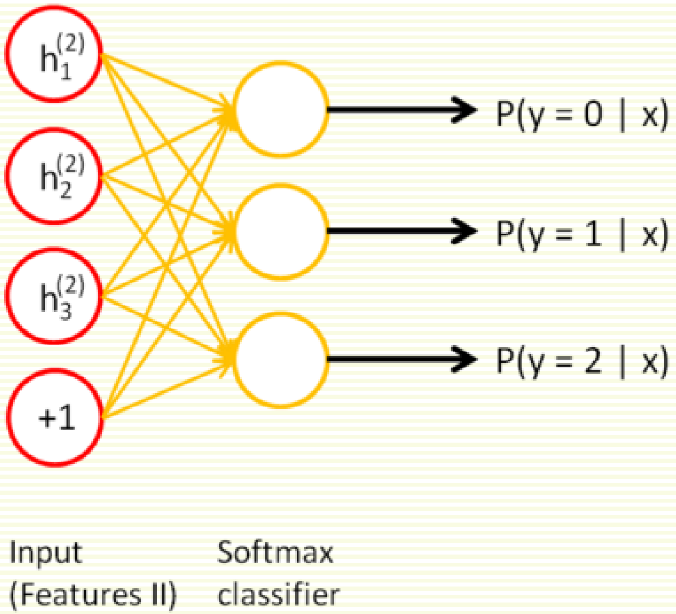
Stacked Auto-Encoders

- Stack many (sparse) auto-encoders in succession and train them using greedy layer-wise training
- Drop the decode output layer each time



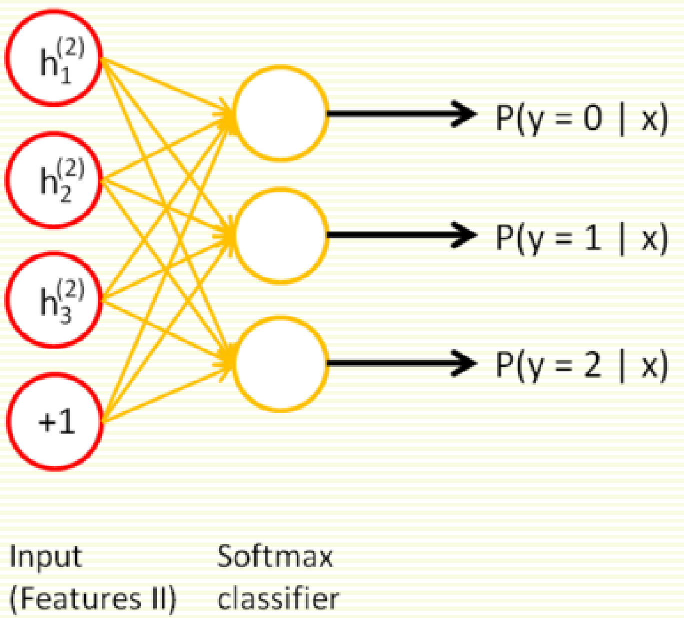
Stacked Auto-Encoders

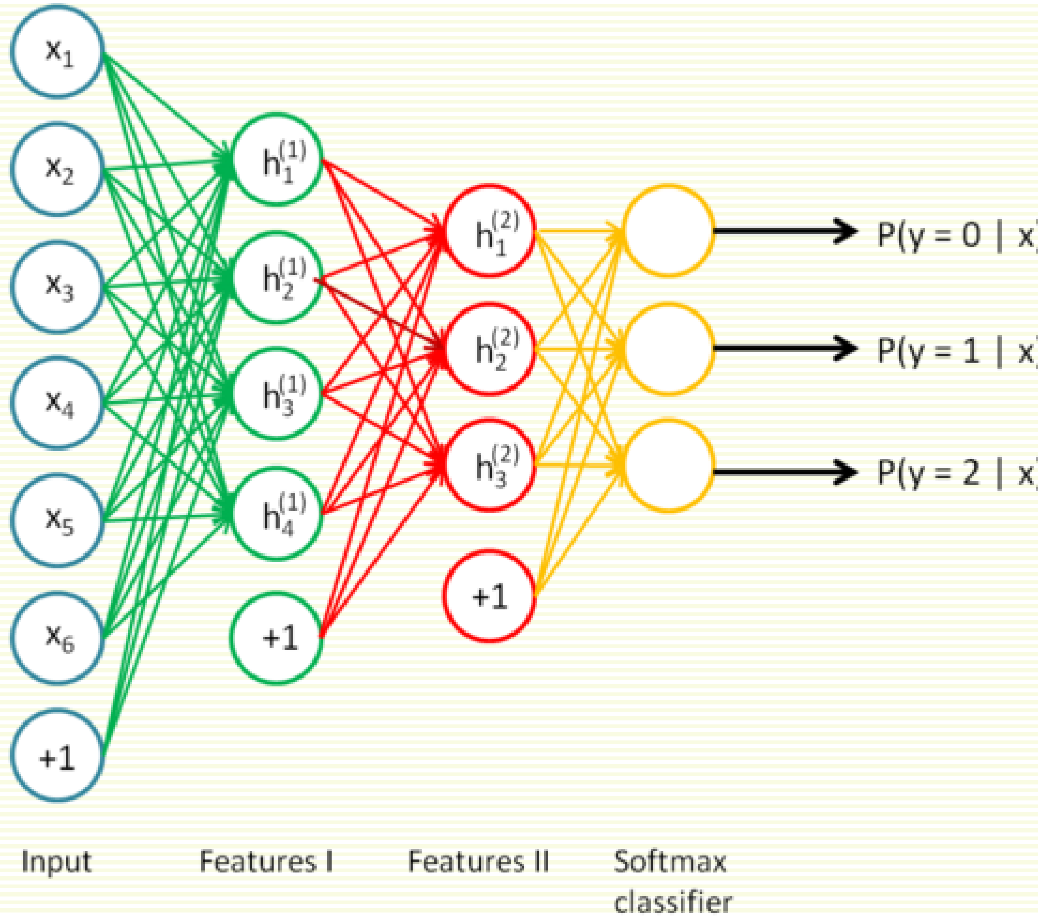
- Do supervised training on the last layer using final features



Stacked Auto-Encoders

- Do supervised training on the last layer using final features
- Then do supervised training on full network to fine-tune all weights





Concluding Remarks

- Advantages
  - NN can learn complex mappings from inputs to outputs, based only on the training samples
  - Easy to incorporate a lot of heuristics
  - Many competitions won recently
- Disadvantages
  - A lot of tricks for successful implementation
  - Theory is not as developed yet