CS9840

Learning and Computer Vision

Prof. Olga Veksler

Lecture 10

Neural Networks

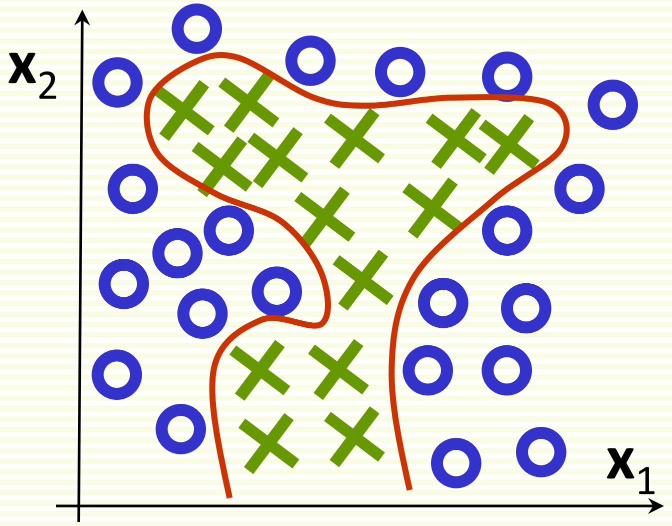
Many slides are from Andrew NG, Yann LeCun, Geoffry Hinton, Abin - Roozgard

Outline

- Short Intro
- Perceptron (1 layer NN)
- Multilayer Perceptron (MLP)
- Deep Networks
    - Convolutional Network

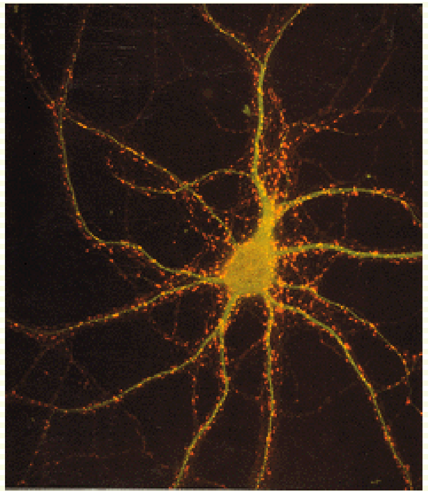
Neural Networks

- Neural Networks correspond to some discriminant function $\mathbf{g}_{NN}(\mathbf{x})$
- Can carve out arbitrarily complex decision boundaries without requiring so many terms as polynomial functions
- Neural Nets were inspired by research in how human brain works
- But also proved to be quite successful in practice
- Are used nowadays successfully for a wide variety of applications
    - took some time to get them to work
    - now used by US post for postal code recognition

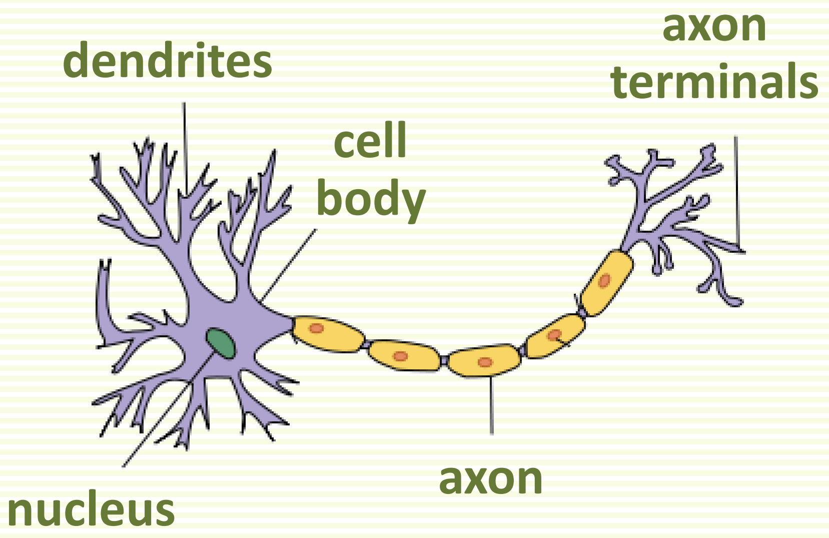


Neuron: Basic Brain Processor

- 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"

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
  - initially seemed promising, but too many claims were made

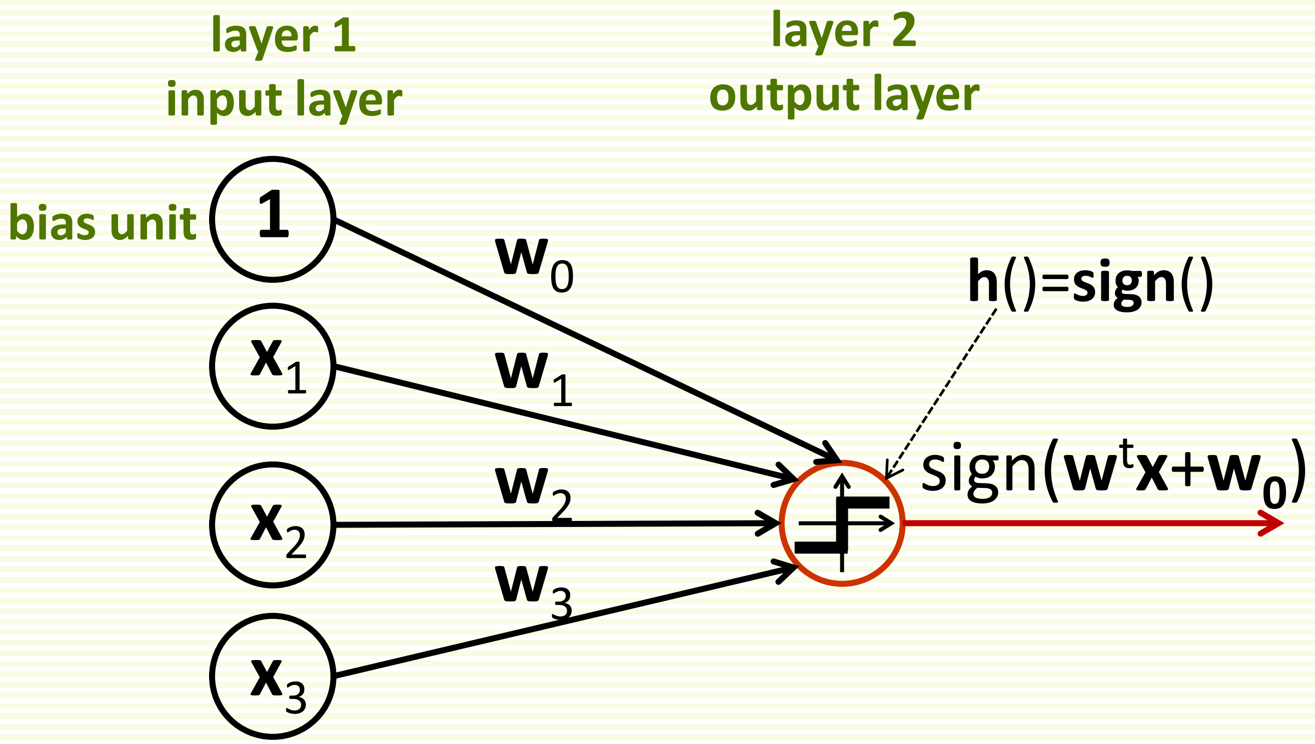
ANN History: Stagnation

- Early success lead to a lot of claims which were not fulfilled
- 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

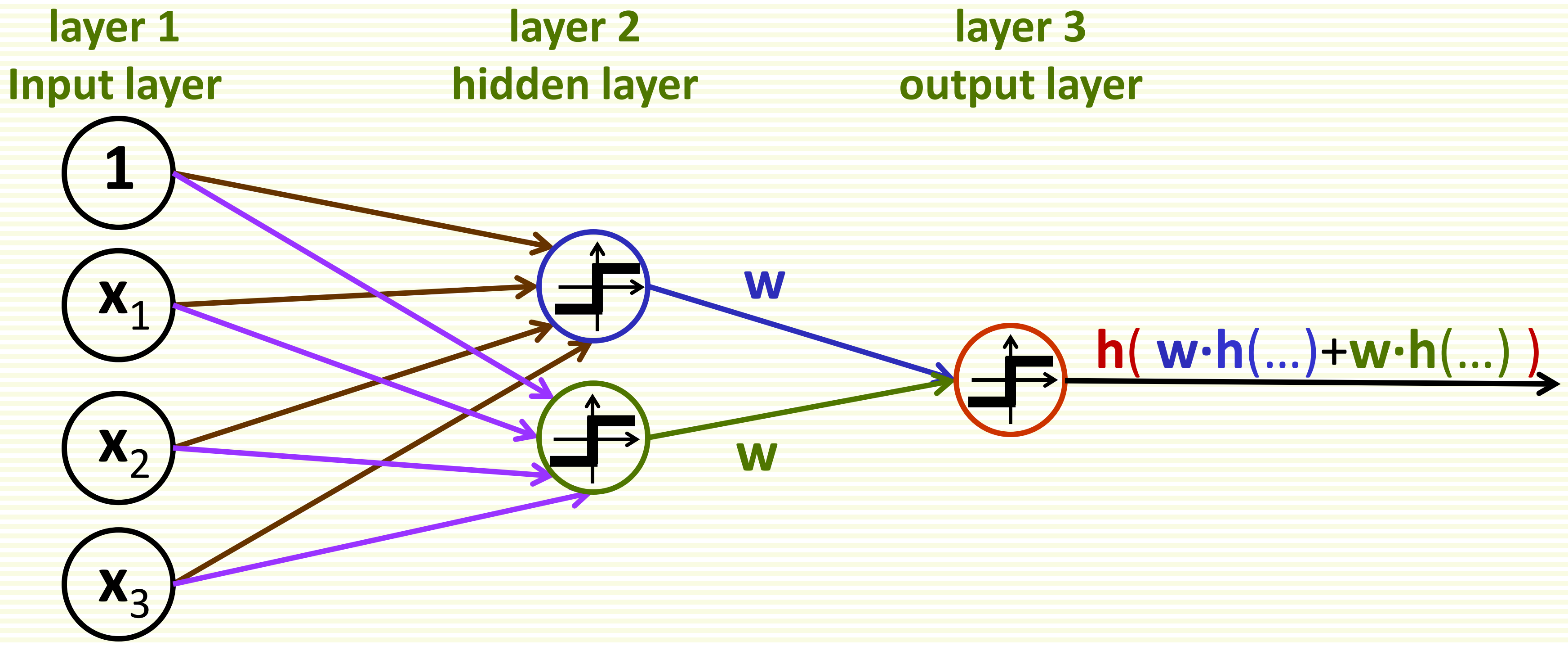
- Revival of ANN in 1980's
- 1982, J. Hopfield
  - New kind of networks (Hopfield's networks)
  - Not just model of how human brain might work, but also how to create useful devices
    - Implements associative memory
- 1982 joint US-Japanese conference on ANN
  - US worries that it will stay behind
- Many examples of mulitlayer Neural Networks appear
- 1986, re-discovery of backpropagation algorithm by Werbos, Rumelhart, Hinton and Ronald Williams
  - Allows a network to learn not linearly separable classes
- Lots of successes in the past 5 years due to better training

Artificial Neural Nets (ANN): Perceptron



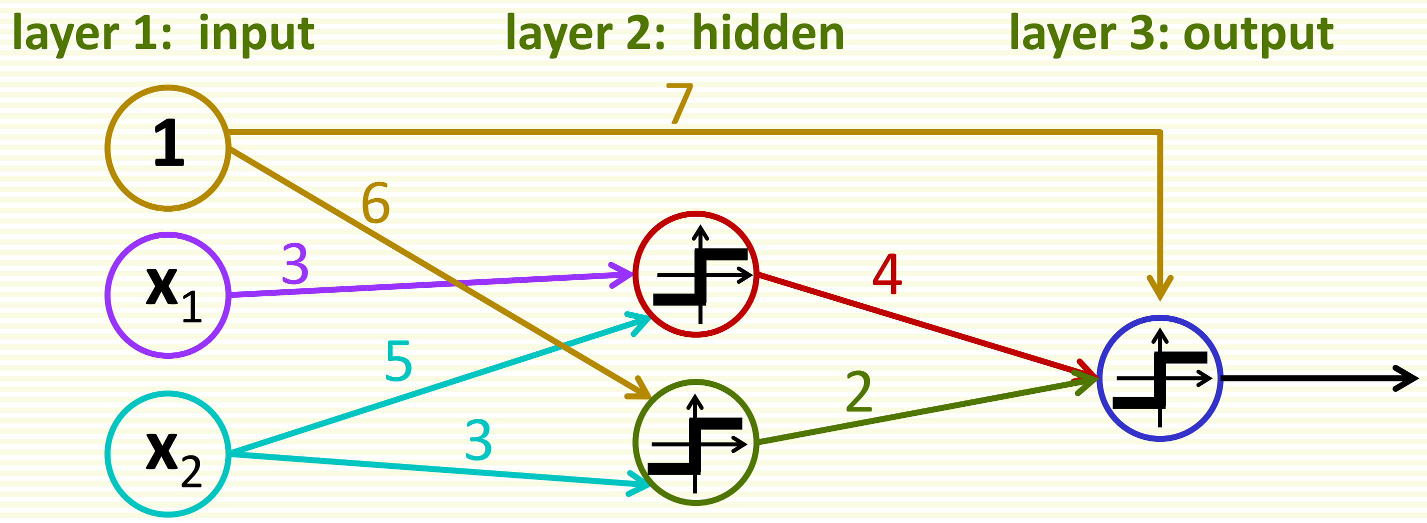
- Linear classifier **f**(**x**) = sign($\mathbf{w}^t\mathbf{x}+\mathbf{w_0}$) is a single neuron "net"
- Input layer units output features, except bias outputs "1"
- Output layer unit applies **sign**() or some other function **h**()
- **h**() is also called an *activation function*

Multilayer Neural Network (MLP)



- First hidden unit outputs: $h(\ldots) = h(w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3)$
- Second hidden unit outputs: $h(\ldots) = h(w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3)$
- Network corresponds to classifier $\mathbf{f}(\mathbf{x}) = h(\, w \cdot h(\ldots) + w \cdot h(\ldots) \,)$
- More complex than Perceptron, more complex boundaries

MLP Small Example



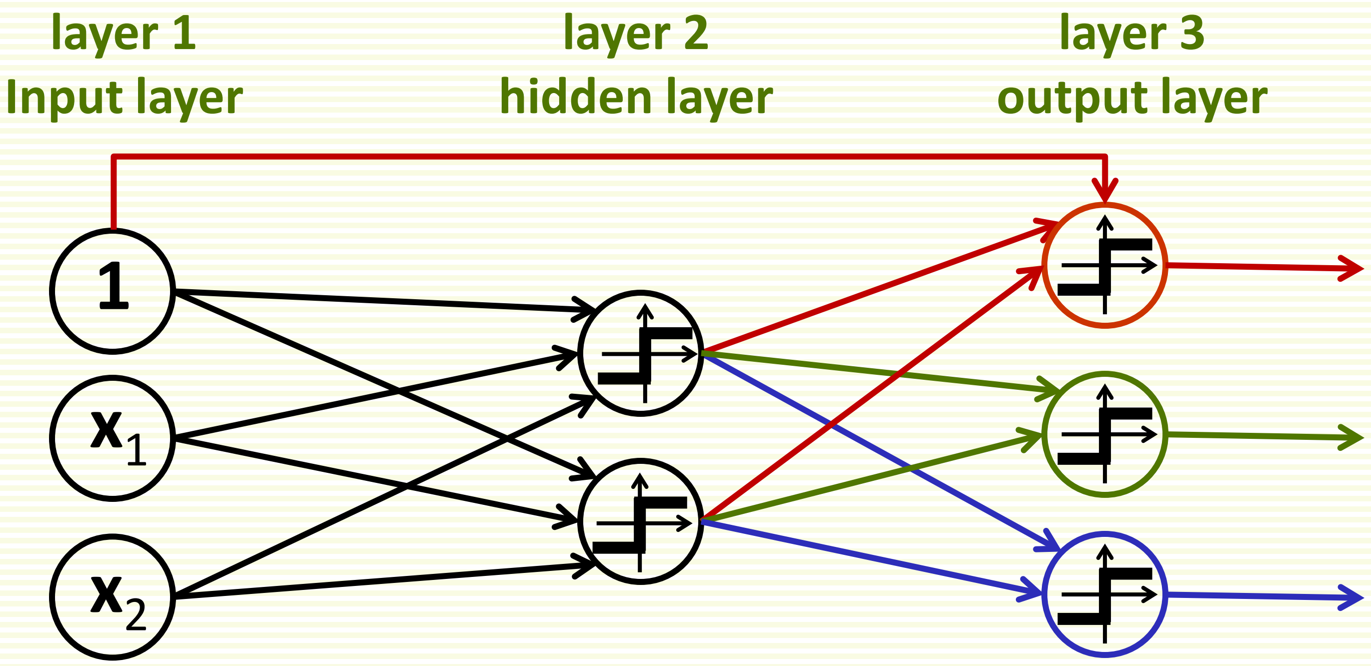
- Let activation function **h**() = sign()
- MLP Corresponds to classifier

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

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

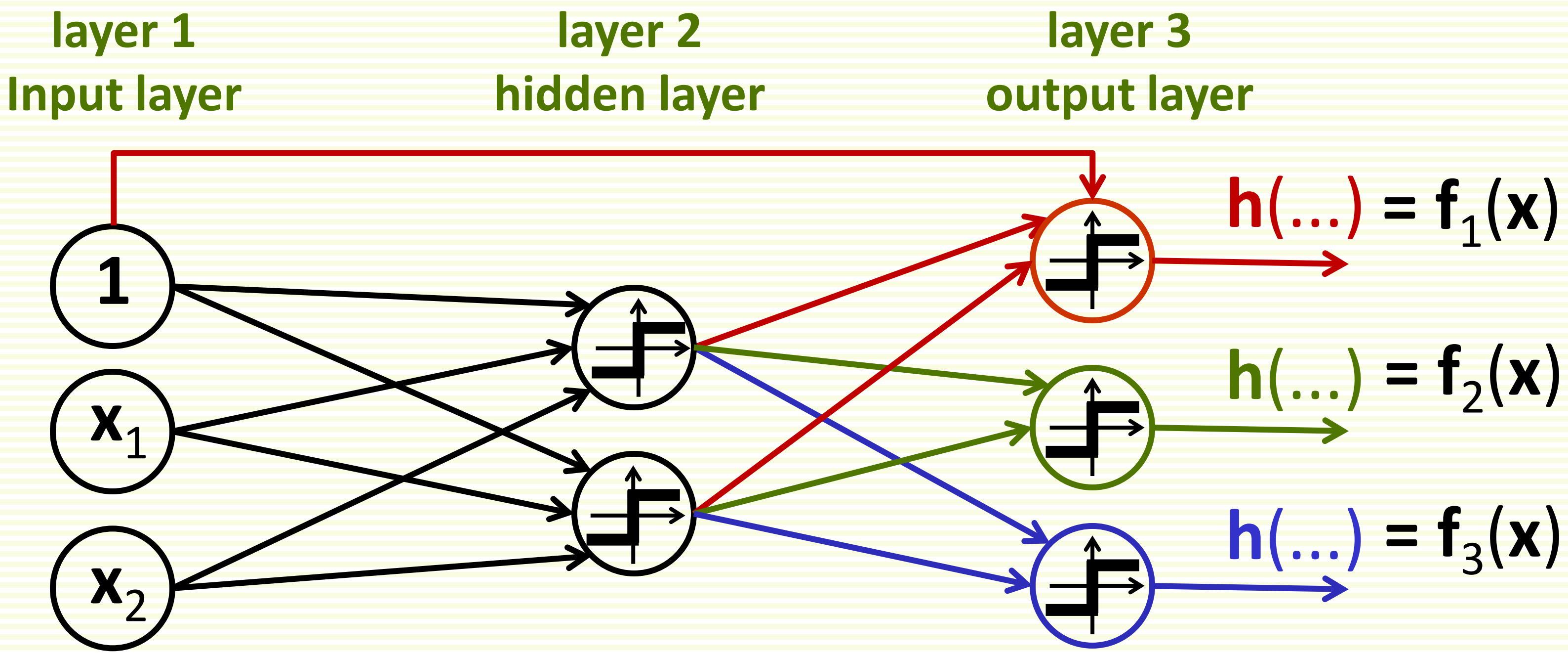
- MLP terminology: computing **f**(**x**) is called *feed forward operation*
  - graphically, function is computed from left to right
- Edge weights are learned through training

MLP: Multiple Classes



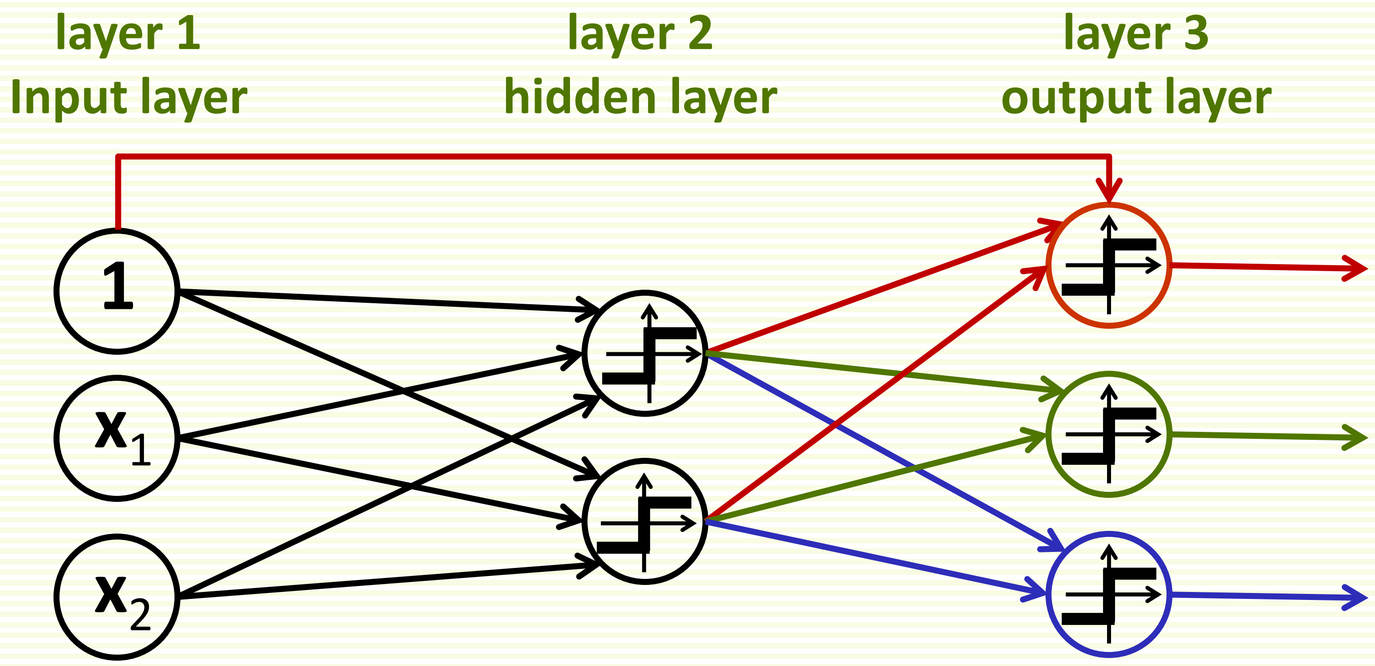
- 3 classes, 2 features, 1 hidden layer
  - 3 input units, one for each feature
  - 3 output units, one for each class
  - 2 hidden units
  - 1 bias unit, usually drawn in layer 1

MLP: General Structure


layer 1
Input layer
layer 2
hidden layer
layer 3
output layer
1
x_1
x_2
$h(\ldots) = f_1(\mathbf{x})$
$h(\ldots) = f_2(\mathbf{x})$
$h(\ldots) = f_3(\mathbf{x})$

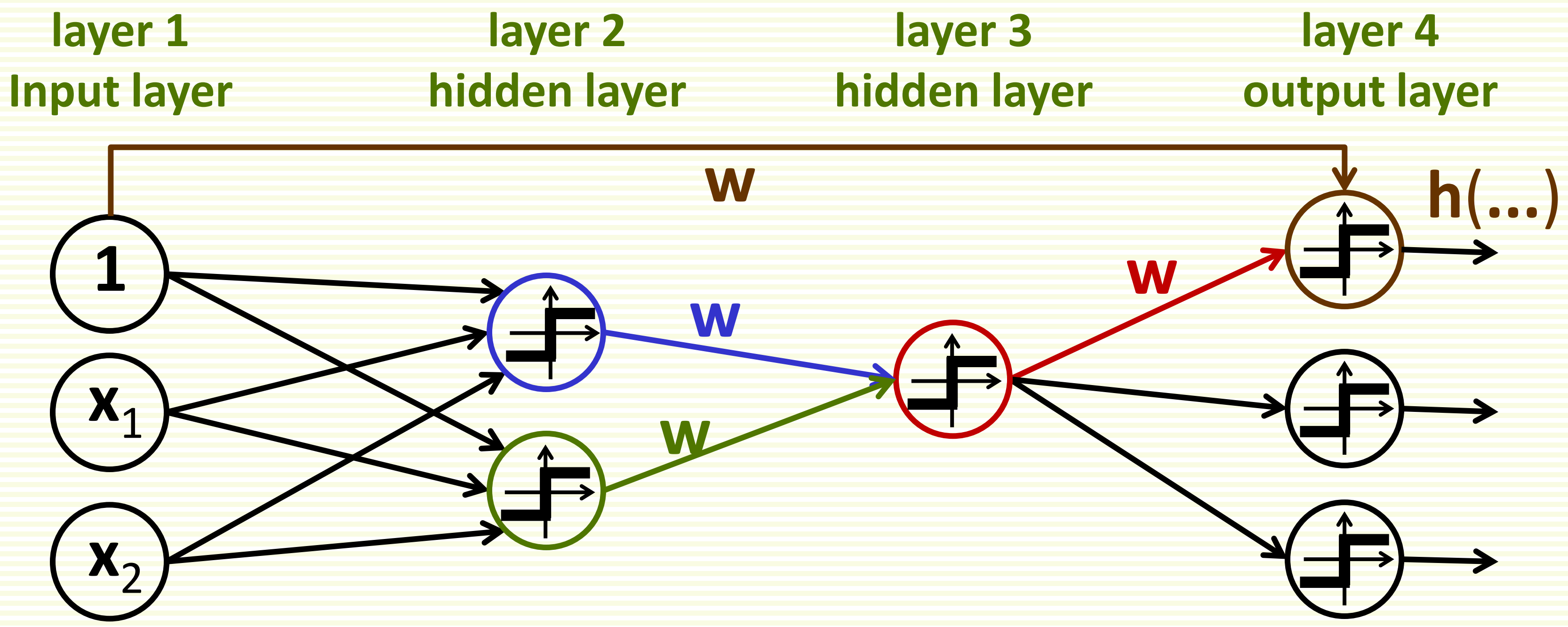
- $\mathbf{f}(\mathbf{x}) = [\mathbf{f}_1(\mathbf{x}), \mathbf{f}_2(\mathbf{x}), \mathbf{f}_3(\mathbf{x})]$ is multi-dimensional
- Classification:
  - If $\mathbf{f}_1(\mathbf{x})$ is largest, decide class 1
  - If $\mathbf{f}_2(\mathbf{x})$ is largest, decide class 2
  - If $\mathbf{f}_3(\mathbf{x})$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

MLP: General Structure

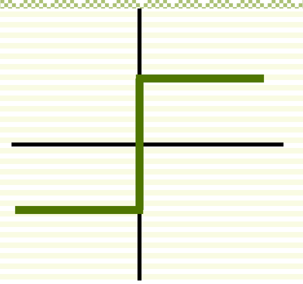


- Can have more than 1 hidden layer
  - **i**th layer connects to (**i+1**)th layer
    - except bias unit can connect to any layer
  - can have different number of units in each hidden layer
- First output unit outputs:

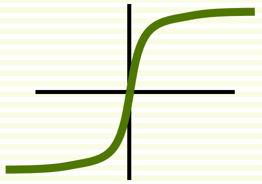
$$\mathbf{h}(\ldots) = \mathbf{h}(\ \mathbf{w}\cdot\mathbf{h}(\ldots) + \mathbf{w}\ ) = \mathbf{h}(\ \mathbf{w}\cdot\mathbf{h}(\mathbf{w}\cdot\mathbf{h}(\ldots) + \mathbf{w}\cdot\mathbf{h}(\ldots)) + \mathbf{w}\ )$$

MLP: Activation Function

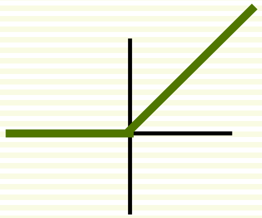
- **h**() = **sign**() is discontinuous, not good for gradient descent



- Instead can use continuous sigmoid function



- Rectified Linear is gaining popularity recently (ReLu)



- Or another differentiable function
- Can even use different activation functions at different layers/units
- From now, assume **h**() is a differentiable function

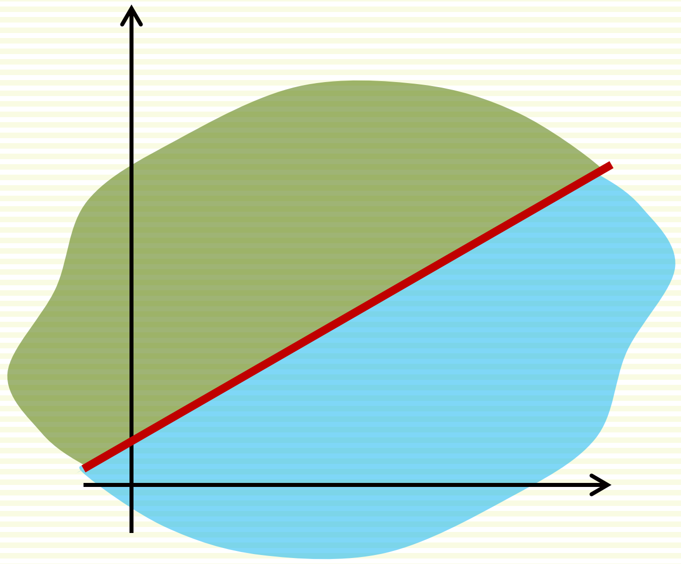
MLP: Overview

- A neural network corresponds to a classifier **f**(**x**,**w**) that can be rather complex
    - 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 neural network, just as before
    - formulate an objective function **J**(**w**)
    - optimize it with gradient descent
    - That's all!
    - Except we need quite a few slides to write down details due to complexity of **f**(**x**,**w**)

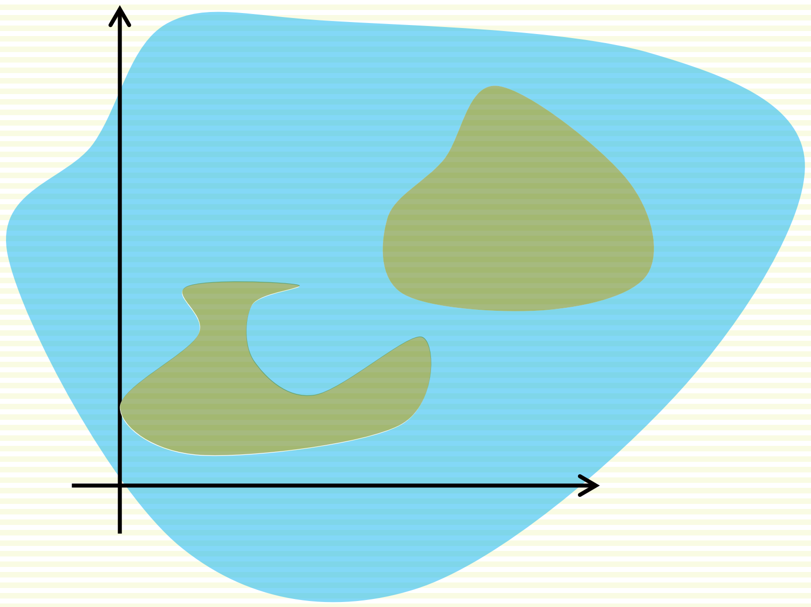
Expressive Power of MLP

- 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)

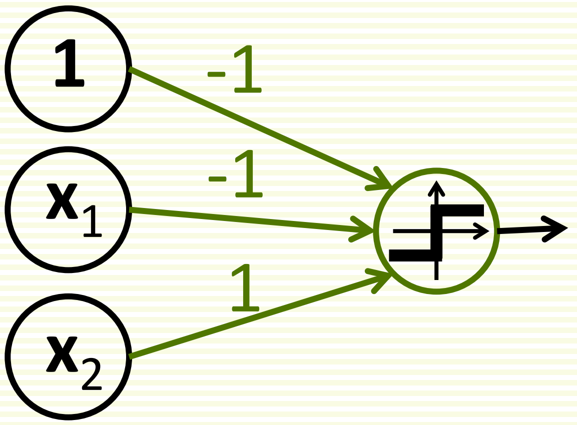


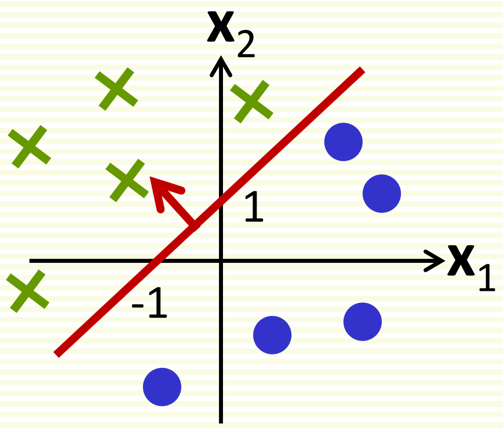
- Arbitrarily complex decision regions
- Even not contiguous

Nonlinear Decision Boundary: Example

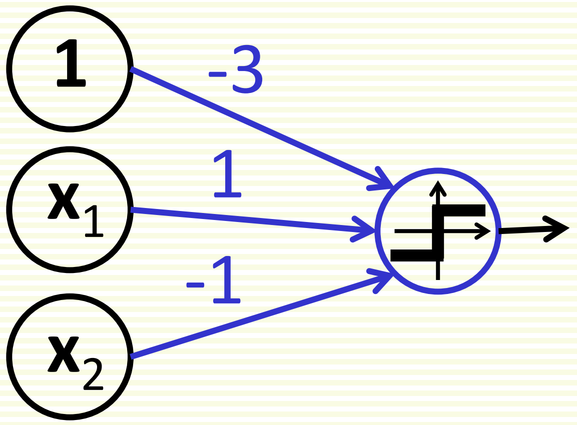
- Start with two Perceptrons, **h**() = sign()

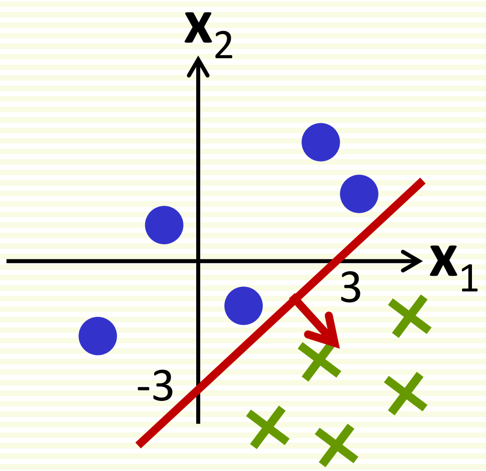
$-\mathbf{x}_1 + \mathbf{x}_2 - 1 > 0 \Rightarrow \text{class } 1$





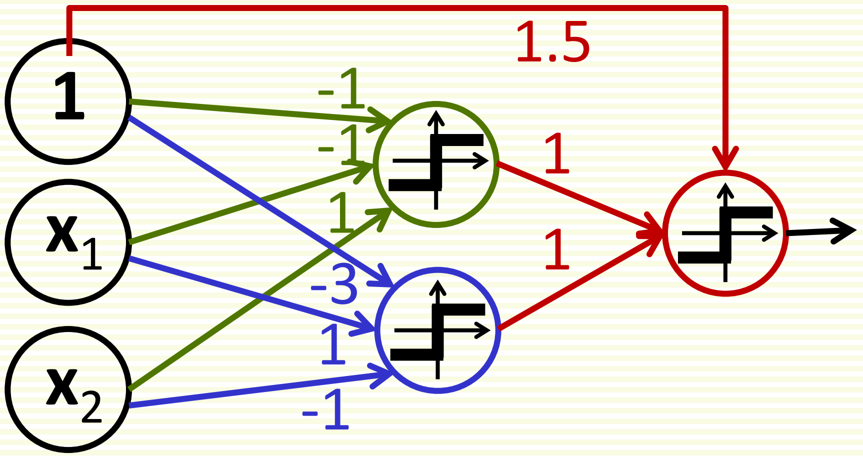
$\mathbf{x}_1 - \mathbf{x}_2 - 3 > 0 \Rightarrow \text{class } 1$

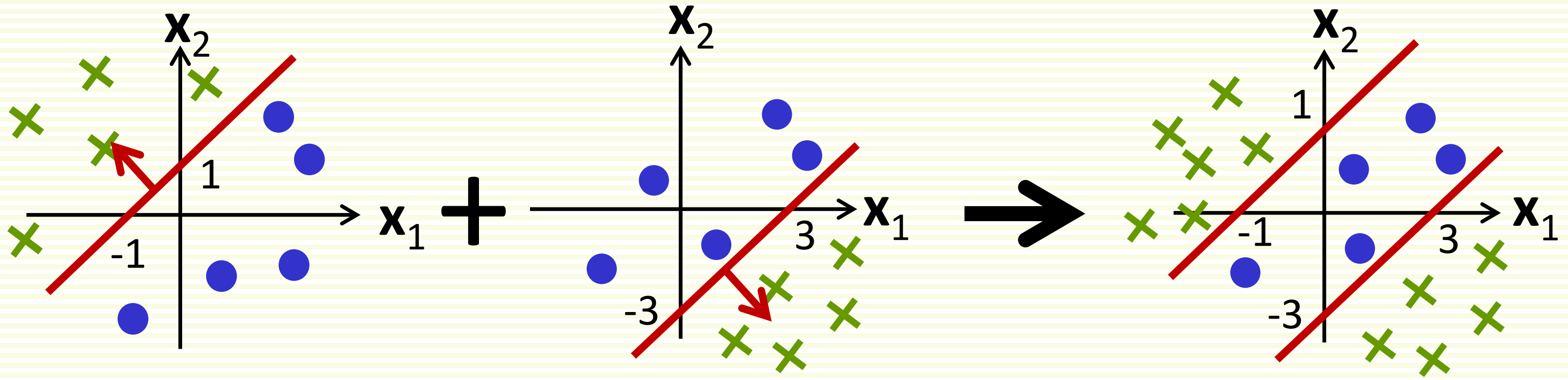




Nonlinear Decision Boundary: Example

- Now combine them into a 3 layer NN



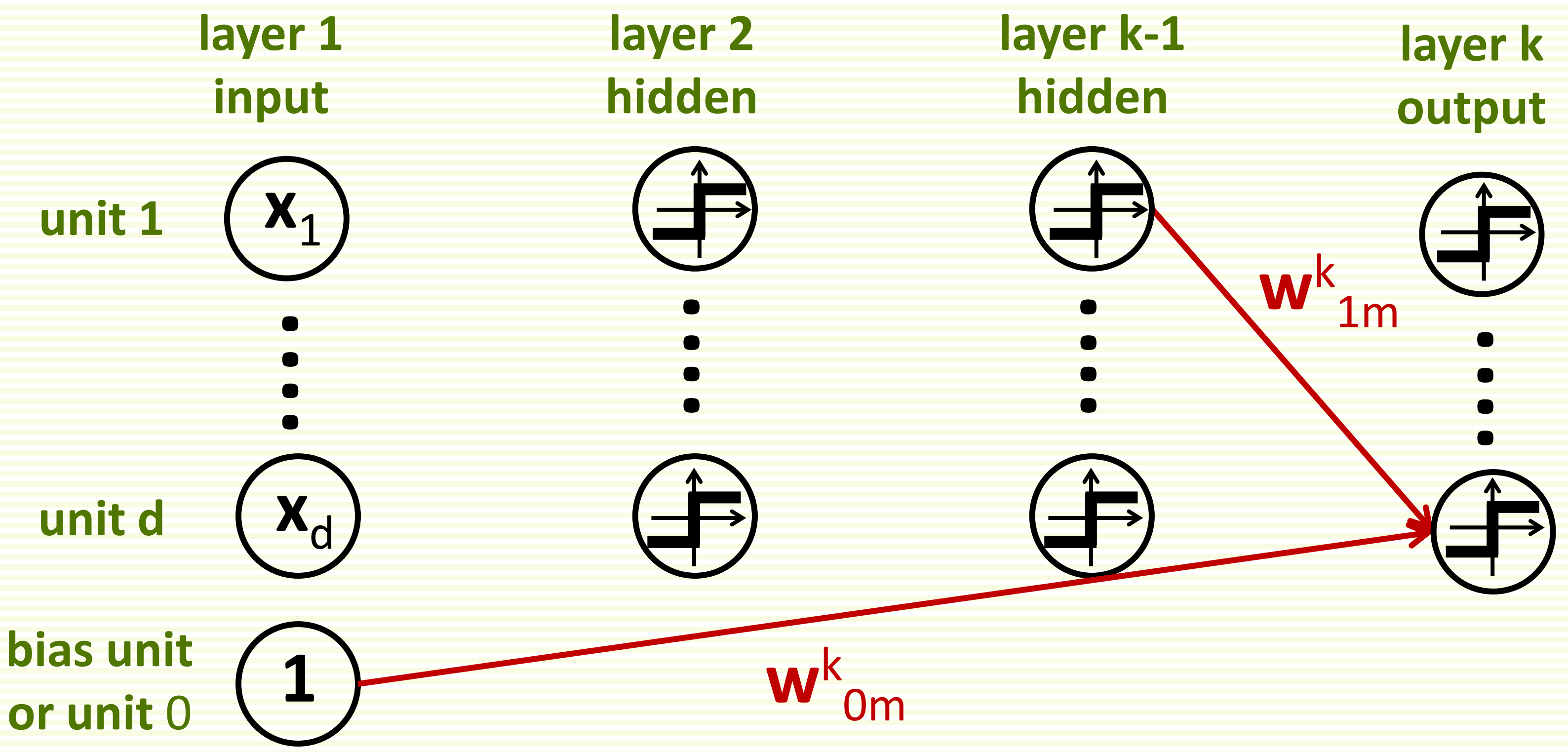


MLP: Modes of Operation

- For Neural Networks, due to historical reasons, training and testing stages have special names
  - **Backpropagation (or training)**

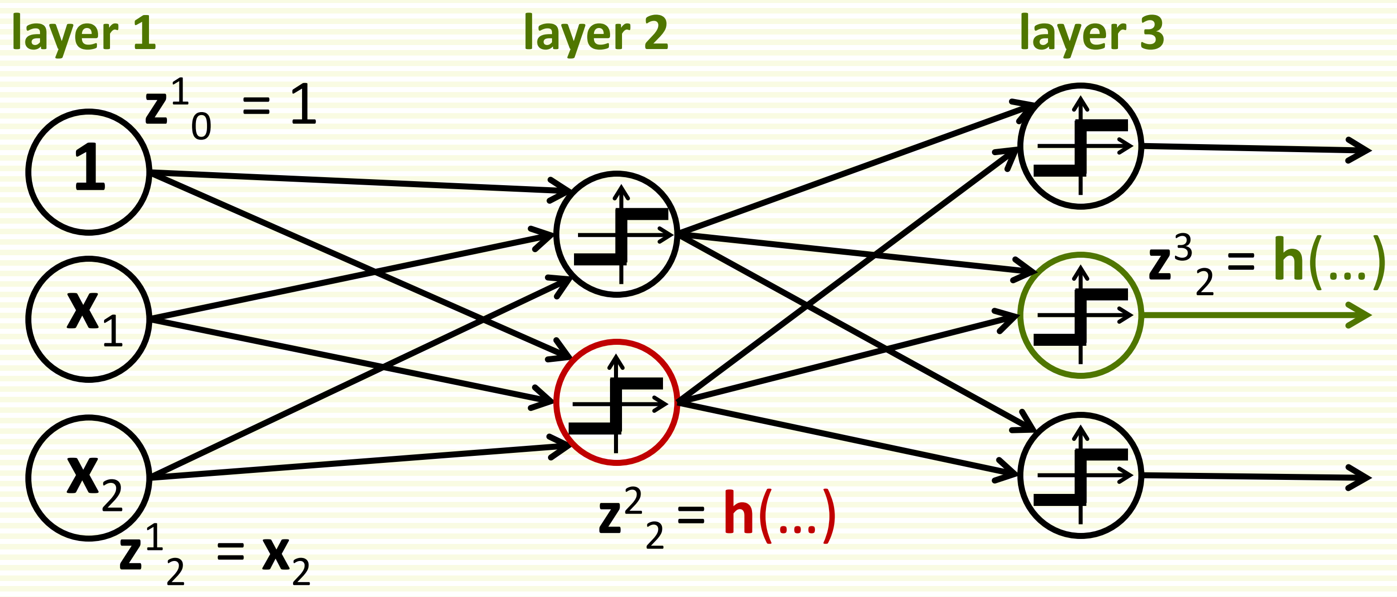
    Minimize objective function with gradient descent
  - **Feedforward (or testing)**

MLP: Notation for Edge Weights



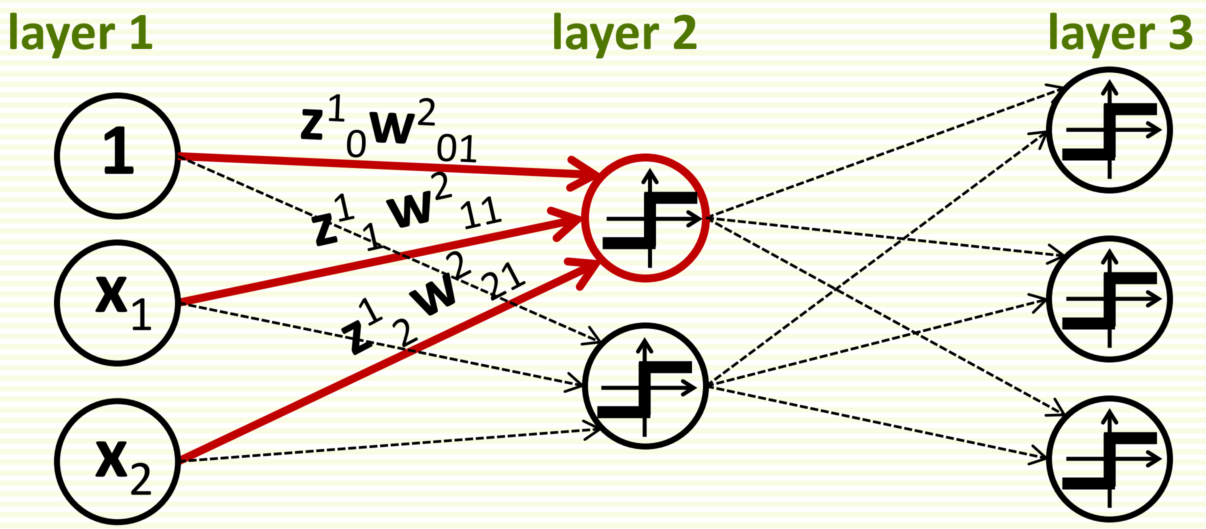
- $\mathbf{w}^k_{pj}$ is edge weight from unit **p** in layer **k**-1 to unit **j** in layer **k**
- $\mathbf{w}^k_{0j}$ is edge weight from bias unit to unit **j** in layer **k**
- $\mathbf{w}^k_j$ is all weights to unit **j** in layer **k**, i.e. $\mathbf{w}^k_{0j}$, $\mathbf{w}^k_{1j}$, …, $\mathbf{w}^k_{N(k-1)j}$
  - **N**(k) is the number of units in layer k, excluding the bias unit

MLP: More Notation


layer 1
layer 2
layer 3
$\mathbf{z}^1_0 = 1$
1
$\mathbf{x}_1$
$\mathbf{x}_2$
$\mathbf{z}^1_2 = \mathbf{x}_2$
$\mathbf{z}^2_2 = \mathbf{h}(\ldots)$
$\mathbf{z}^3_2 = \mathbf{h}(\ldots)$

- Denote the output of unit **j** in layer **k** as $\mathbf{z}^k_j$
- For the input layer (**k**=1), $\mathbf{z}^1_0 = 1$ and $\mathbf{z}^1_j = \mathbf{x}_j$, $\mathbf{j} \neq 0$
- For all other layers, (**k** > 1), $\mathbf{z}^k_j = \mathbf{h}(\ldots)$
- Convenient to set $\mathbf{z}^k_0 = 1$ for all **k**
- Set $\mathbf{z}^k = [\mathbf{z}^k_0, \mathbf{z}^k_1, \ldots, \mathbf{z}^k_{\mathbf{N(k)}}]$

MLP: More Notation



- Net activation at unit **j** in layer **k** > 1 is the sum of inputs

$$\mathbf{a}_j^k = \sum_{p=1}^{N_{k-1}} \mathbf{z}_p^{k-1} \mathbf{w}_{pj}^k + \mathbf{w}_{0j}^k = \sum_{p=0}^{N_{k-1}} \mathbf{z}_p^{k-1} \mathbf{w}_{pj}^k = \mathbf{z}^{k-1} \cdot \mathbf{w}_j^k$$

$$\mathbf{a}_1^2 = \mathbf{z}_0^1 \mathbf{w}_{01}^2 + \mathbf{z}_1^1 \mathbf{w}_{11}^2 + \mathbf{z}_2^1 \mathbf{w}_{21}^2$$

- For **k** > 1, $\mathbf{z}^k{}_j = \mathbf{h}(\mathbf{a}^k{}_j)$

MLP: Class Representation

- **m** class problem, let Neural Net have **t** layers
- Let $\mathbf{x}^i$ be a example of class **c**
- It is convenient to denote its label as $\mathbf{y}^i = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \begin{array}{l} \\ \\ \longleftarrow \text{row } \mathbf{c} \\ \\ \\ \end{array}$
- Recall that $\mathbf{z}^t_c$ is the output of unit **c** in layer **t** (output layer)
- $\mathbf{f}(\mathbf{x}) = \mathbf{z}^t = \begin{bmatrix} \mathbf{z}^t_1 \\ \vdots \\ \mathbf{z}^t_c \\ \vdots \\ \mathbf{z}^t_m \end{bmatrix}$. If $\mathbf{x}^i$ is of class **c**, want $\mathbf{z}^t = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \begin{array}{l} \\ \\ \longleftarrow \text{row } \mathbf{c} \\ \\ \\ \end{array}$

Training MLP: Objective Function

- Want to minimize difference between $\mathbf{y}^i$ and $\mathbf{f}(\mathbf{x}^i)$
- Use squared difference
- Let $\mathbf{w}$ be all edge weights in MLP collected in one vector
- Error on one example $\mathbf{x}^i$: $$J_i(\mathbf{w}) = \frac{1}{2}\sum_{c=1}^{m}\left(f_c\left(\mathbf{x}^i\right) - y_c^i\right)^2$$
- Error on all examples: $$J(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n}\sum_{c=1}^{m}\left(f_c\left(\mathbf{x}^i\right) - y_c^i\right)^2$$
- Gradient descent:

```
initialize w to random
choose ε, α
while α||∇J(w)|| > ε
    w = w - α∇J(w)
```

Training MLP: Single Sample

- For simplicity, first consider error for one example $\mathbf{x}^i$

$$\mathbf{J}_i(\mathbf{w}) = \frac{1}{2}\left\|\mathbf{y}^i - \mathbf{f}(\mathbf{x}^i)\right\|^2 = \frac{1}{2}\sum_{\mathbf{c}=1}^{\mathbf{m}}\left(\mathbf{f}_\mathbf{c}(\mathbf{x}^i) - \mathbf{y}_\mathbf{c}^i\right)^2$$

  - $\mathbf{f}_c(\mathbf{x}^i)$ depends on **w**
  - $\mathbf{y}^i$ is independent of **w**

- Compute partial derivatives w.r.t. $\mathbf{w}^k{}_{pj}$ for all k, p, j
- Suppose have **t** layers

$$\mathbf{f}_\mathbf{c}(\mathbf{x}^i) = \mathbf{z}_\mathbf{c}^\mathbf{t} = \mathbf{h}(\mathbf{a}_\mathbf{c}^\mathbf{t}) = \mathbf{h}(\mathbf{z}^{\mathbf{t}-1} \cdot \mathbf{w}_\mathbf{c}^\mathbf{t})$$

Training MLP: Single Sample

- For derivation, we use:

$$\mathbf{J}_i(\mathbf{w}) = \frac{1}{2}\sum_{c=1}^{m}\left(\mathbf{f}_c(\mathbf{x}^i) - \mathbf{y}_c^i\right)^2$$

$$\mathbf{f}_c(\mathbf{x}^i) = \mathbf{h}(\mathbf{a}_c^t) = \mathbf{h}(\mathbf{z}^{t-1}\cdot\mathbf{w}_c^t)$$

- For weights $\mathbf{w}^t{}_{pj}$ to the output layer **t**:

$$\frac{\partial}{\partial \mathbf{w}_{pj}^t}\mathbf{J}(\mathbf{w}) = \left(\mathbf{f}_j(\mathbf{x}^i) - \mathbf{y}_j^i\right)\frac{\partial}{\partial \mathbf{w}_{pj}^t}\left(\mathbf{f}_j(\mathbf{x}^i) - \mathbf{y}_j^i\right)$$

- $\dfrac{\partial}{\partial \mathbf{w}_{pj}^t}\left(\mathbf{f}_j(\mathbf{x}^i) - \mathbf{y}_j^i\right) = \mathbf{h}'(\mathbf{a}_j^t)\,\mathbf{z}_p^{t-1}$

- Therefore, $\dfrac{\partial}{\partial \mathbf{w}_{pj}^t}\mathbf{J}_i(\mathbf{w}) = \left(\mathbf{f}_j(\mathbf{x}^i) - \mathbf{y}_j^i\right)\mathbf{h}'(\mathbf{a}_j^t)\,\mathbf{z}_p^{t-1}$

    - both $\mathbf{h}'(\mathbf{a}_j^t)$ and $\mathbf{z}_p^{t-1}$ depend on $\mathbf{x}^i$. For simpler notation, we don't make this dependence explicit.

Training MLP: Single Sample

- For a layer **k**, compute partial derivatives w.r.t. $\mathbf{w}^k_{pj}$
- Gets complex, since have lots of function compositions
- Will give the rest of derivatives
- First define $\mathbf{e}^k_j$, the error attributed to unit **j** in layer **k**:
- For layer **t** (output): $\mathbf{e}_j^t = \left(\mathbf{f}_j\left(\mathbf{x}^i\right) - \mathbf{y}_j^i\right)$
- For layers **k** < **t**: $\mathbf{e}_j^k = \sum_{c=1}^{N(k+1)} \mathbf{e}_c^{k+1}\mathbf{h}'\left(\mathbf{a}_c^{k+1}\right)\mathbf{w}_{jc}^{k+1}$
- Thus for $2 \le \mathbf{k} \le \mathbf{t}$: $\frac{\partial}{\partial \mathbf{w}_{pj}^k}\mathbf{J}_i(\mathbf{w}) = \mathbf{e}_j^k\,\mathbf{h}'\left(\mathbf{a}_j^k\right)\mathbf{z}_p^{k-1}$

MLP Training: Multiple Samples

- Error on one example $\mathbf{x}^i$:

$$\mathbf{J_i(w)} = \frac{1}{2}\sum_{\mathbf{c}=1}^{\mathbf{m}}\left(\mathbf{f_c}\left(\mathbf{x^i}\right) - \mathbf{y^i_c}\right)^2$$

$$\frac{\partial}{\partial \mathbf{w^k_{pj}}}\mathbf{J_i(w)} = \mathbf{e^k_j}\,\mathbf{h'}\left(\mathbf{a^k_j}\right)\mathbf{z^{k-1}_p}$$

- Error on all examples:

$$\mathbf{J(w)} = \frac{1}{2}\sum_{\mathbf{i}=1}^{\mathbf{n}}\sum_{\mathbf{c}=1}^{\mathbf{m}}\left(\mathbf{f_c}\left(\mathbf{x^i}\right) - \mathbf{y^i_c}\right)^2$$

$$\frac{\partial}{\partial \mathbf{w^k_{pj}}}\mathbf{J(w)} = \sum_{\mathbf{i}=1}^{\mathbf{n}}\mathbf{e^k_j}\,\mathbf{h'}\left(\mathbf{a^k_j}\right)\mathbf{z^{k-1}_p}$$

Training Protocols

- Batch Protocol
  - true 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
- Mini Batch
  - Update weights after processing a ‘batch’ of examples
  - 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

MLP Training: Single Sample

```
initialize w to small random numbers
choose $\varepsilon, \alpha$
while $\alpha||\nabla J(\mathbf{w})|| > \varepsilon$
    for i = 1 to n
        r = random index from {1,2,…,n}
        $\mathbf{delta}_{pjk} = 0 \quad \forall\ \mathbf{p,j,k}$
        $\mathbf{e}_j^t = \left(\mathbf{f}_j\left(\mathbf{x}^r\right) - \mathbf{y}_j^r\right) \quad \forall \mathbf{j}$
        for k = t to 2
            $\mathbf{delta}_{pjk} = \mathbf{delta}_{pjk} - \mathbf{e}_j^k \mathbf{h}'\left(\mathbf{a}_j^k\right) \mathbf{z}_p^{k-1}$
            $\mathbf{e}_j^{k-1} = \sum_{c=1}^{N(k)} \mathbf{e}_c^k \mathbf{h}'\left(\mathbf{a}_c^k\right) \mathbf{w}_{jc}^k \quad \forall \mathbf{j}$
        $\mathbf{w}^k{}_{pj} = \mathbf{w}^k{}_{pj} + \mathbf{delta}_{pjk} \ \forall\ \mathbf{p,j,k}$
```

MLP Training: Batch

```
initialize w to small random numbers
choose  $\varepsilon, \alpha$
while $\alpha||\nabla J(\mathbf{w})|| > \varepsilon$
    for i = 1 to n
        $\mathrm{delta}_{pjk} = 0 \quad \forall\ p,j,k$
        $e_j^t = \left(f_j\left(x^i\right) - y_j^i\right) \quad \forall j$
        for k = t to 2
            $\mathrm{delta}_{pjk} = \mathrm{delta}_{pjk} - e_j^k h'\left(a_j^k\right) z_p^{k-1}$
            $e_j^{k-1} = \sum_{c=1}^{N(k)} e_c^k h'\left(a_c^k\right) w_{jc}^k \quad \forall j$
    $w^k_{pj} = w^k_{pj} + \mathrm{delta}_{pjk}\ \forall\ p,j,k$
```

BackPropagation of Errors

- In MLP terminology, training is called *backpropagation*
- errors computed (propagated) backwards from the output to the input layer

```
while $\alpha||\nabla J(w)|| > \varepsilon$
    for i = 1 to n
        $delta_{pjk} = 0 \quad \forall\, p,j,k$
        $e_j^t = \left(y_j^r - f_j\left(x^r\right)\right) \quad \forall j$
        for k = t to 2
            $delta_{pjk} = delta_{pjk} - e_j^k h'\left(a_j^k\right) z_p^{k-1}$
            $e_j^{k-1} = \sum_{c=1}^{N(k)} e_c^k h'\left(a_c^k\right) w_{jc}^k \quad \forall j$
    $w^k_{pj} = w^k_{pj} + delta_{pjk} \;\forall\, p,j,k$
```

first last layer errors computed

then errors computed backwards

MLP Training

- Important: weights should be initialized to random nonzero numbers

$$\frac{\partial}{\partial \mathbf{w}_{pj}^{k}} \mathbf{J}_i(\mathbf{w}) = -\mathbf{e}_j^k \mathbf{h}'\left(\mathbf{a}_j^k\right) \mathbf{z}_p^{k-1}$$

$$\mathbf{e}_j^k = \sum_{c=1}^{N(k+1)} \mathbf{e}_c^{k+1} \mathbf{h}'\left(\mathbf{a}_c^{k+1}\right) \mathbf{w}_{jc}^{k+1}$$

- if $\mathbf{w}^k_{jc} = 0$, errors $\mathbf{e}^k_j$ are zero for layers $\mathbf{k} < \mathbf{t}$
- weights in layers $\mathbf{k} < \mathbf{t}$ will not be updated

Practical tips for BP: Weight Decay

- To avoid overfitting, it is recommended to keep weights small
- Implement weight decay after each weight update:

$$\mathbf{w}^{new} = \mathbf{w}^{new}(1-\beta),\ 0 < \beta < 1$$

- Additional benefit is that "unused" weights grow small and may be eliminated altogether
  - a weight is "unused" if it is left almost unchanged by the backpropagation algorithm

Practical Tips for BP: 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:

$$\mathbf{w}^{t+1} = \mathbf{w}^{t} + (1-\beta)\underbrace{\left[\alpha \frac{\partial \mathbf{J}}{\partial \mathbf{w}}\right]}_{\text{steepest descent direction}} + \underbrace{\beta \Delta \mathbf{w}^{t-1}}_{\text{previous direction}}$$

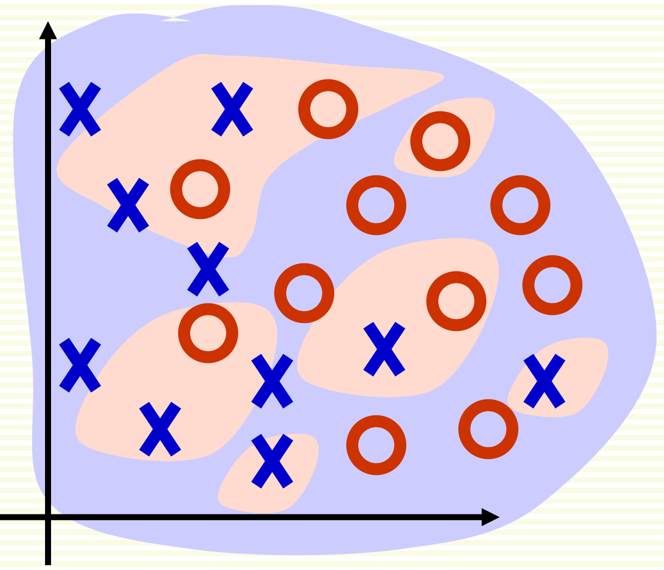
Practical Tips for BP: Normalization

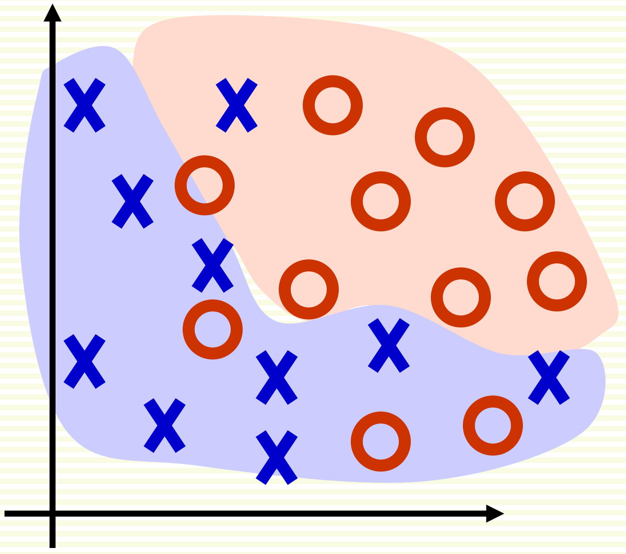
- Features should be normalized for faster convergence
- Suppose we measure fish length in meters and weight in grams
    - Typical sample [length = 0.5, weight = 3000]
    - Feature length will be almost ignored
    - If length is in fact important, learning will be very slow
- Any normalization we looked at before (lecture on kNN) will do
    - Test samples should be normalized exactly as the training samples

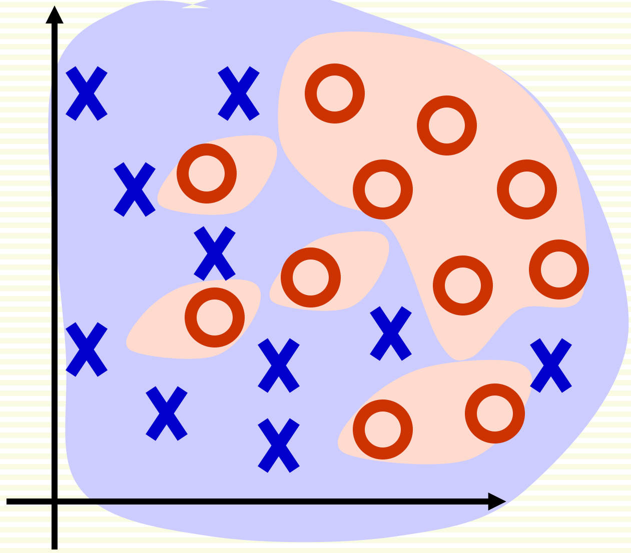
Practical Tips: Learning Rate

- As any gradient descent algorithm, backpropagation depends on the learning rate α
- Rule of thumb $\alpha = 0.1$
- However can adjust α at the training time
- The objective function $\mathbf{J}(\mathbf{w})$ should decrease during gradient descent
    - If $\mathbf{J}(\mathbf{w})$ oscillates, α is too large, decrease it
    - If $\mathbf{J}(\mathbf{w})$ goes down but very slowly, α is too small, increase it

MLP Training: How long to Train?







training time

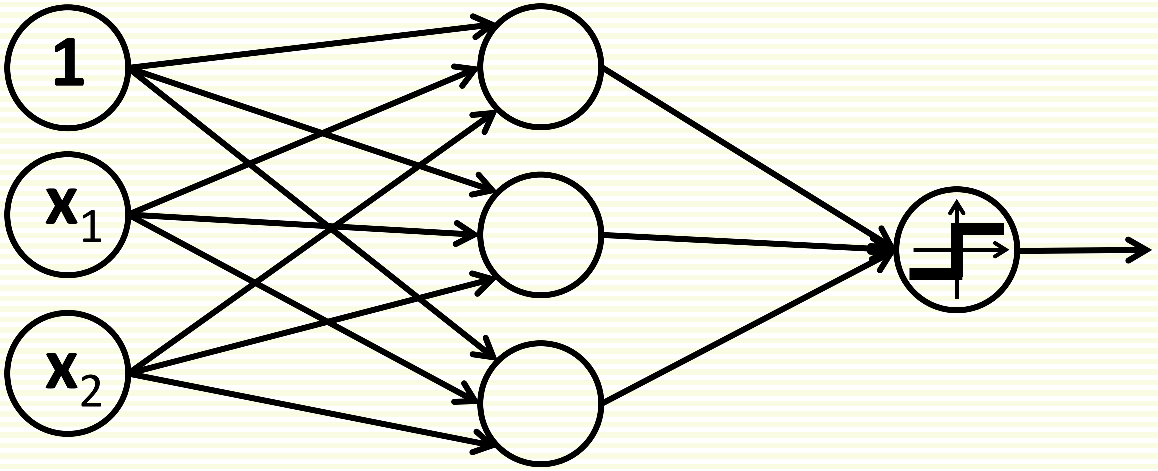
Large training error: random decision regions in the beginning - underfit

Small training error: decision regions improve with time

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

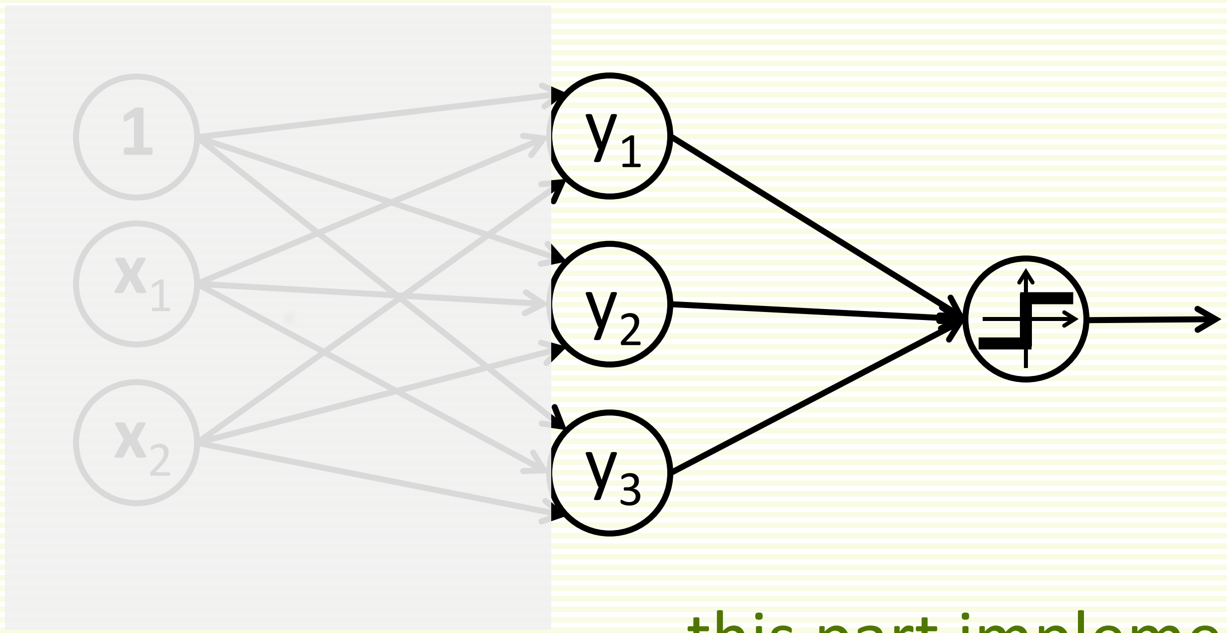
can learn when to stop training through validation

MLP as Non-Linear Feature Mapping



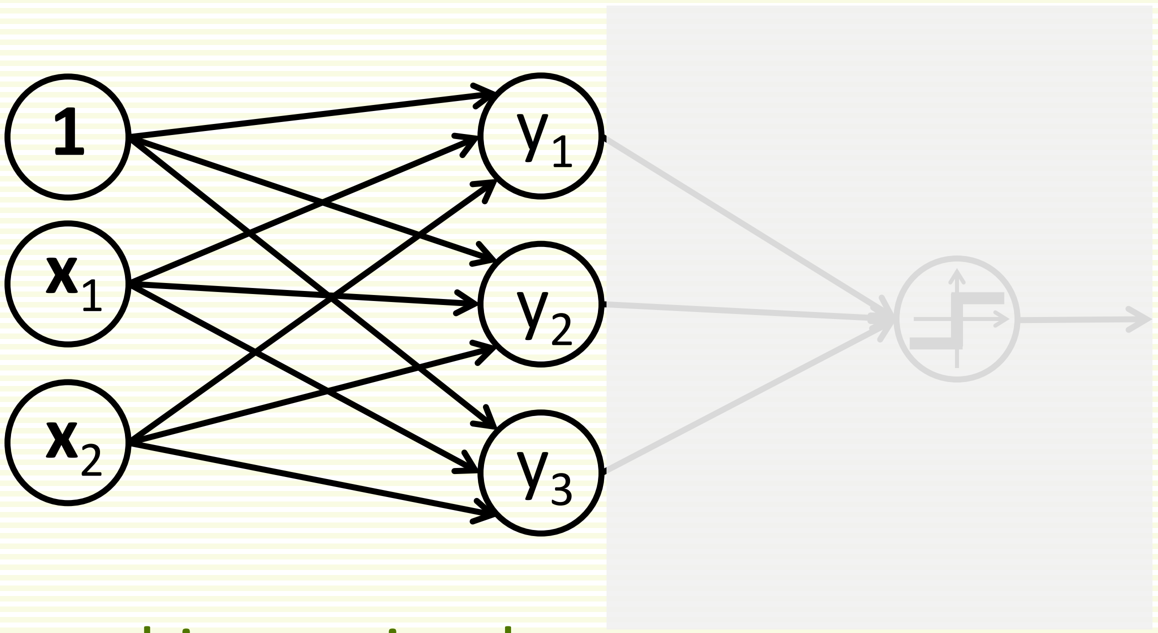
- MLP can be interpreted as first mapping input features to new features
- Then applying Perceptron (linear classifier) to the new features

MLP as Non-Linear Feature Mapping



this part implements
Perceptron (liner classifier)

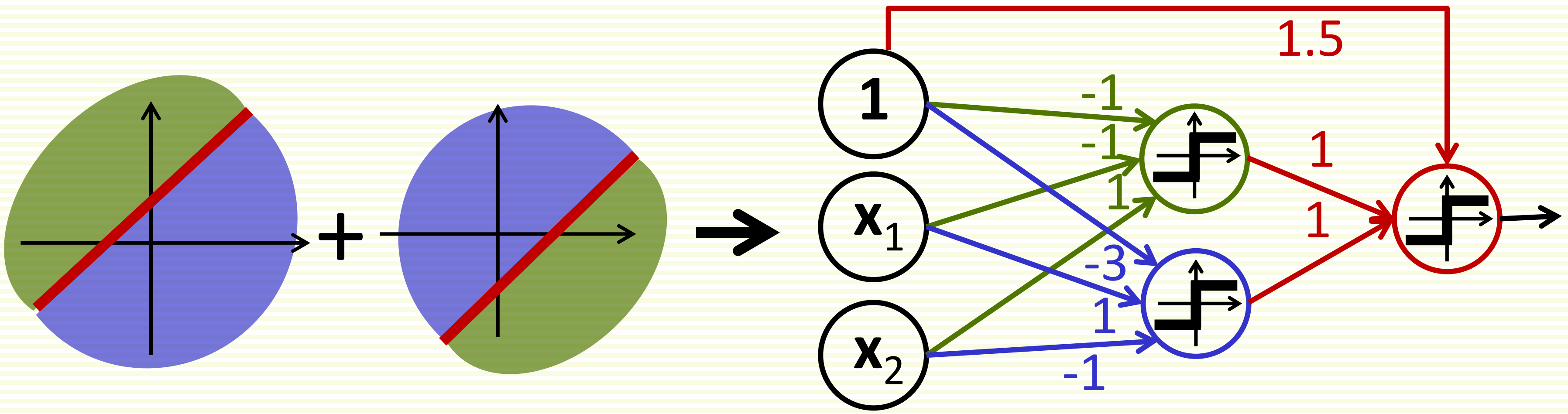
MLP as Non-Linear Feature Mapping

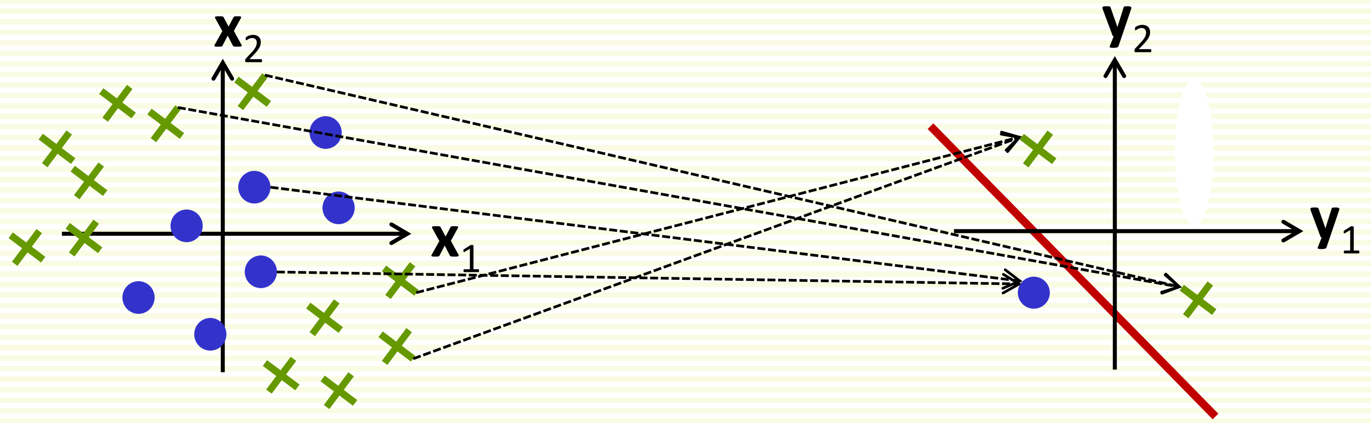


this part implements
mapping to new features **y**

MLP as Nonlinear Feature Mapping

- Consider 3 layer NN example we saw previously:





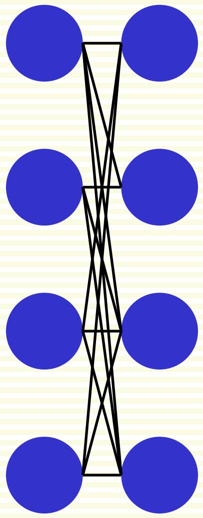
non linearly separable in the original feature space

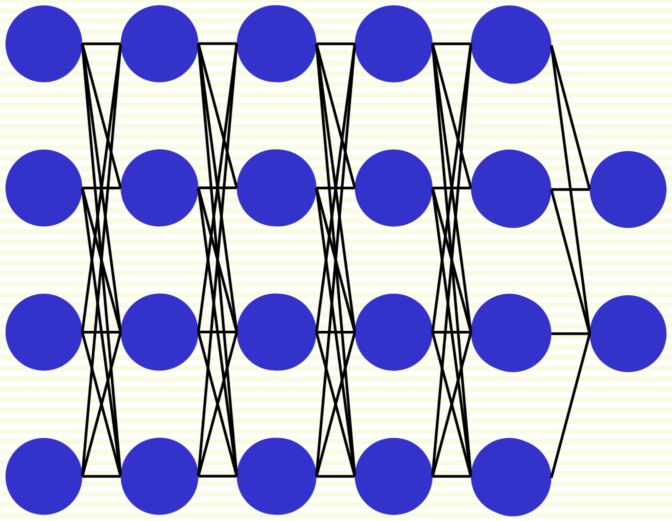
linearly separable in the new feature space

Shallow vs. Deep Architecture

- How many layers should we choose?

Shallow network **Deep network**

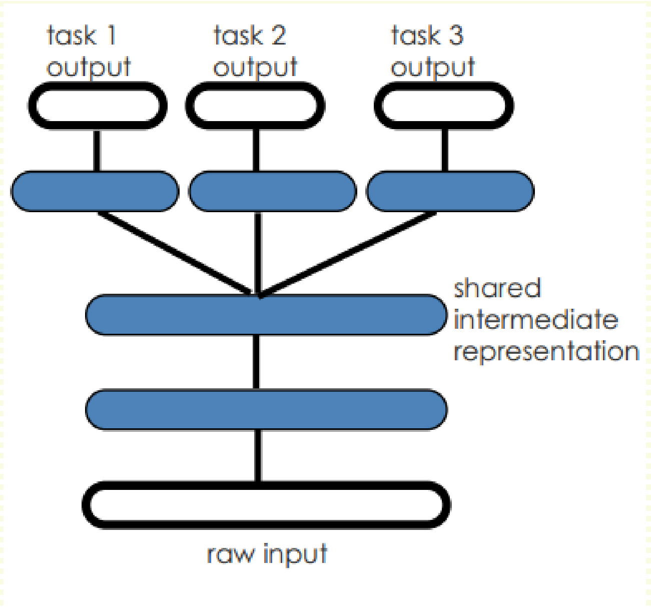




- Deep network lead to many successful applications recently

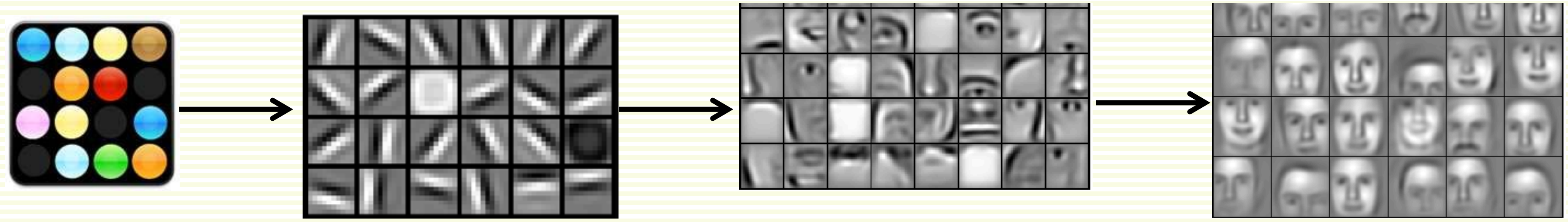
Why Deep Networks

- 2 layer networks can represent any function
- But deep architectures are more efficient for representing some classes of functions
  - problems which 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
  - maybe especially in image processing, with structure being mainly local
- 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 are natural, especially in vision
- Each stage is a trainable feature transform
- Level of abstraction increases with the level



Input layer: pixels

First layer: edges

Second layer: object parts (combination of edges)

Third layer: objects (combinations of object parts)

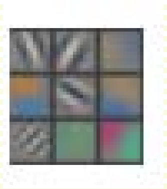
Why Deep Networks: Hierarchical Feature Extraction

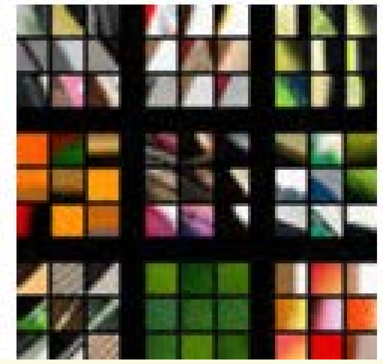
- Another example (from M. Zeiler'2013)

Visualization of learned features

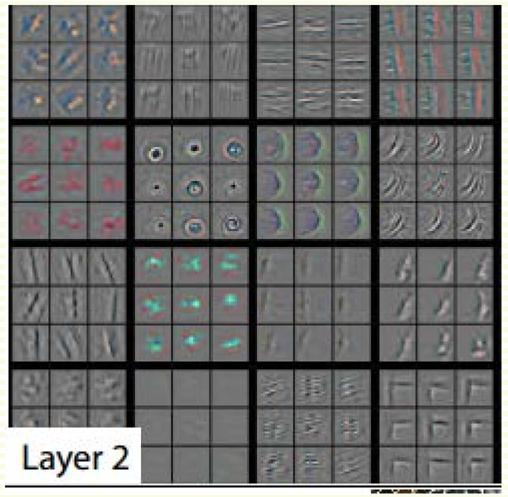
Patches that result in high response

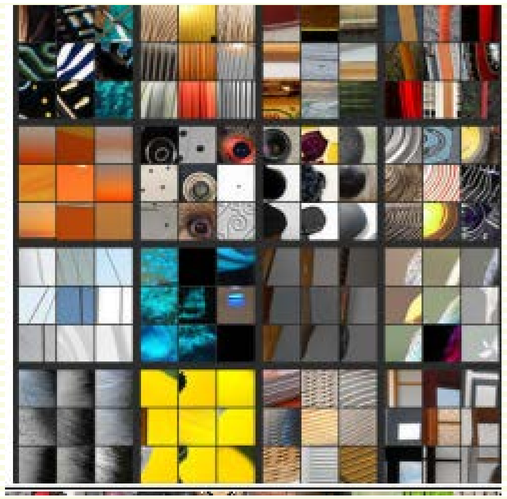
Layer 1





Layer 2



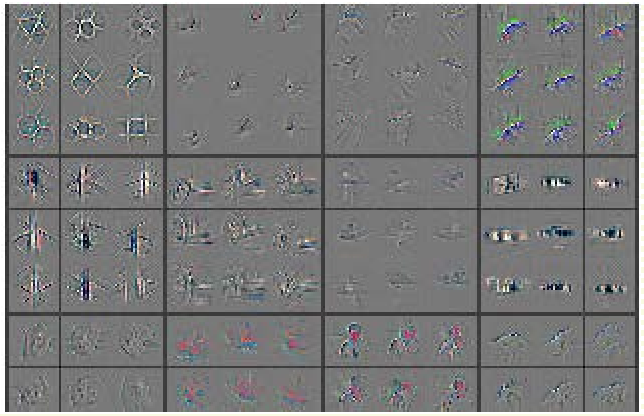


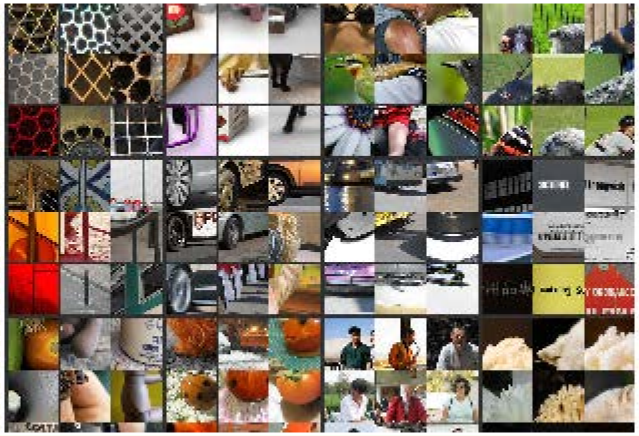
Why Deep Networks: Hierarchical Feature Extraction

Visualization of learned features

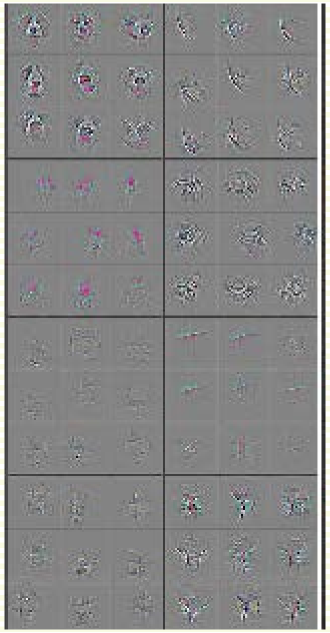
Patches that result in high response

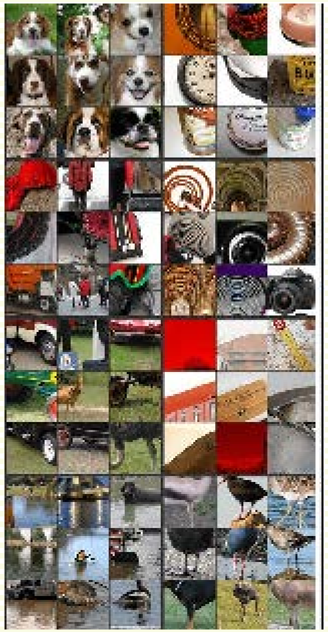
Layer 3





Layer 4





Early Work

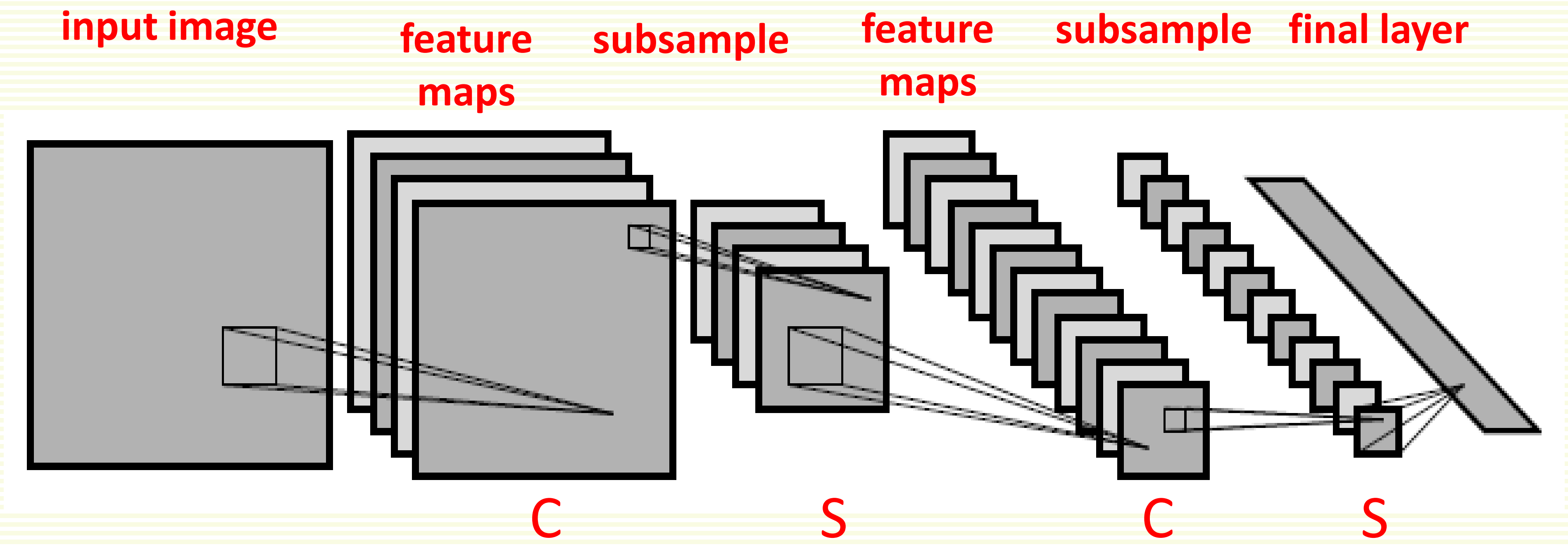
- Fukushima (1980) – Neo-Cognitron
- LeCun (1998) – Convolutional Neural Networks
  - Similarities to Neo-Cognitron
- Many layered Networks trained with backpropagation
  - Tried early but without much success
    - Very slow
    - Diffusion of gradient
  - recent work has shown significant training improvements with various tricks (drop-out, unsupervised learning of early layers, etc.)

Prior Knowledge for Network Architecture

- We can put our prior knowledge about the task into the network by designing appropriate
  - 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 Nets

- Neural Networks with special type of architecture that is particularly good for image processing

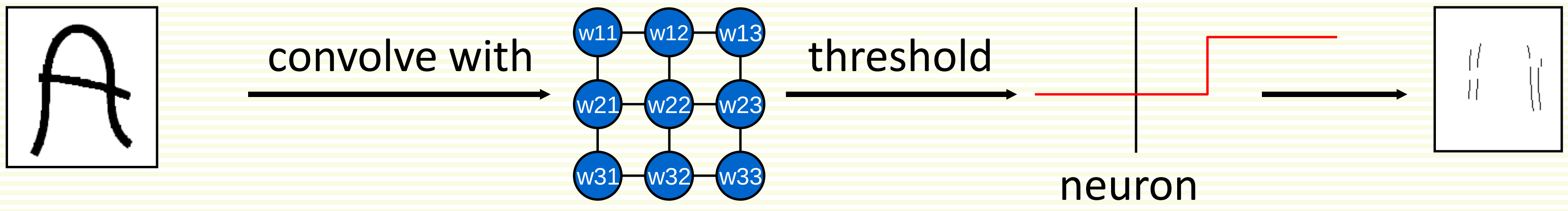


Convolution layer: feature extraction

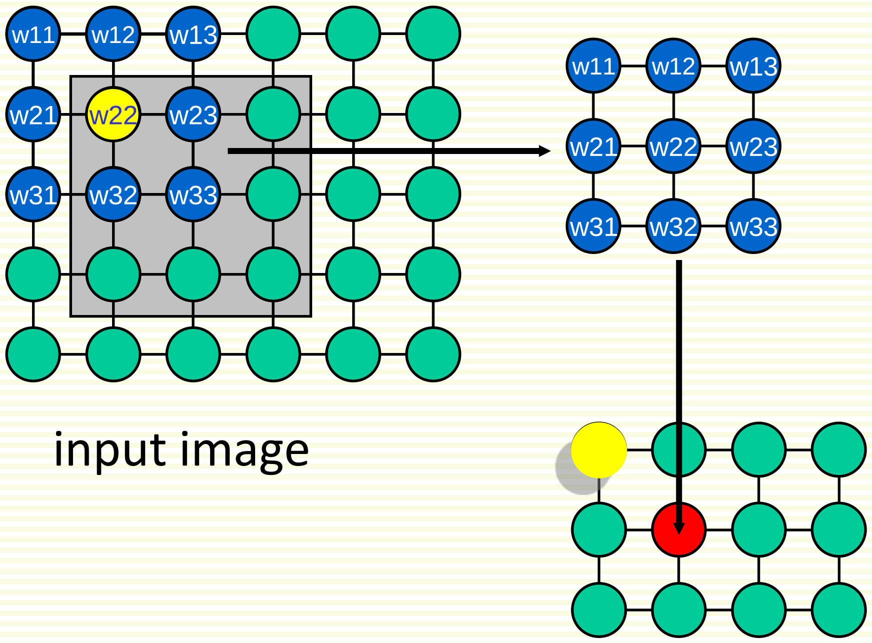
Subsampling layer: shift and distortion invariance

Feature extraction or Convolution layer

- Use convolution (and thresholding) to detect the same feature at different image positions



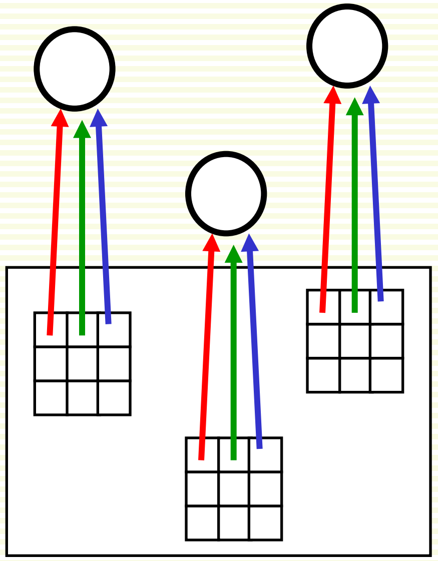
- Recall how convolution works:



convolved thresholded image = feature map

Feature extraction

- Convolution masks are learned
  - tunable parameters of the network
- Shared weights: all neurons in a feature share the same weights
  - but not the biases
- Thus all neurons detect the same feature at different positions in the input image
  - if a feature is useful in one image location, it should be useful in all other locations
  - also greatly reduces the number of tunable parameters



The red connections all have the same weight
The red connections all have the same weight
The red connections all have the same weight

Weight Sharing Constraints

- It is easy to modify backpropagation algorithm to incorporate weight sharing
- We compute the gradients as usual, and then modify the gradients so that they satisfy the constraints.
  - so if the weights started off satisfying the constraints, they will continue to satisfy them

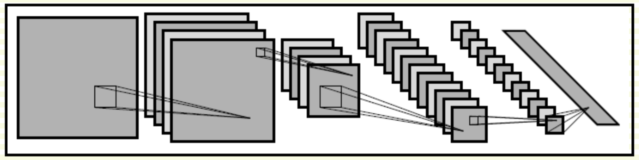
$$To\ constrain: \quad w_1 = w_2$$

$$we\ need: \quad \Delta w_1 = \Delta w_2$$

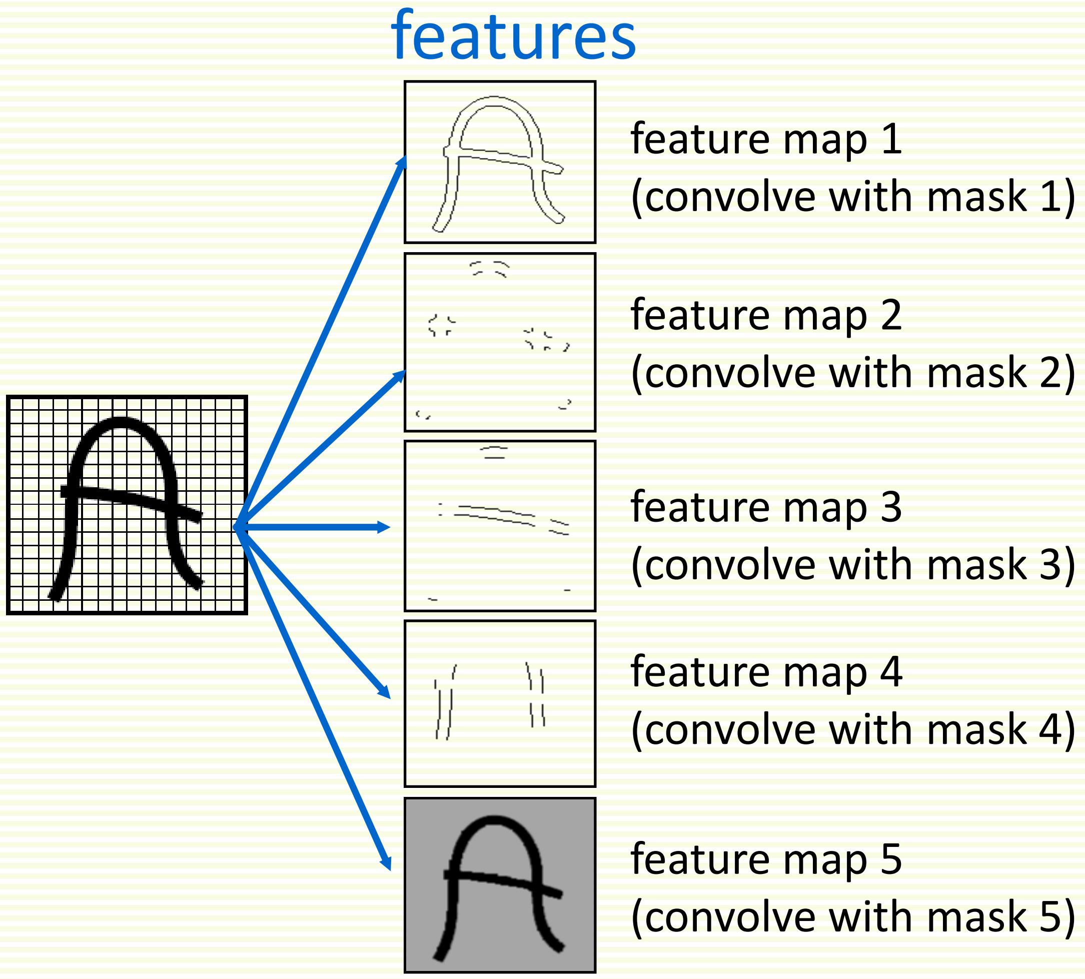
$$compute: \quad \frac{\partial E}{\partial w_1} \quad and \quad \frac{\partial E}{\partial w_2}$$

$$use \quad \frac{\partial E}{\partial w_1} + \frac{\partial E}{\partial w_2} \quad for\ w_1\ and\ w_2$$

Feature extraction or Convolution layer

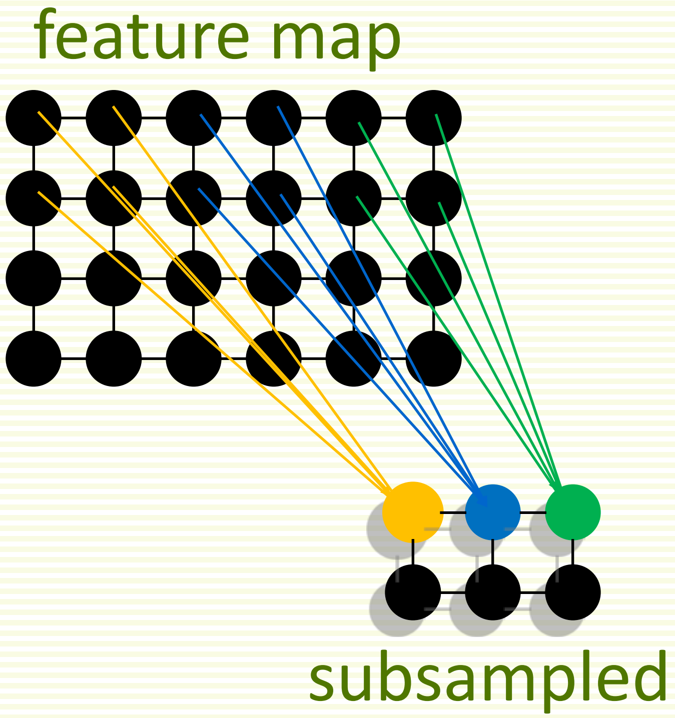


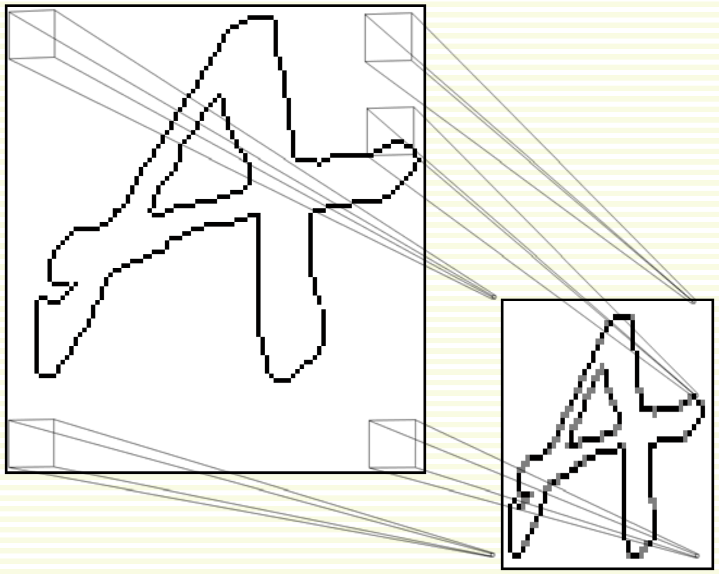
- Use several different feature types, each with its own mask, and the resulting map of replicated detectors
- Allows each patch of image to be represented in several ways



Subsampling Layer

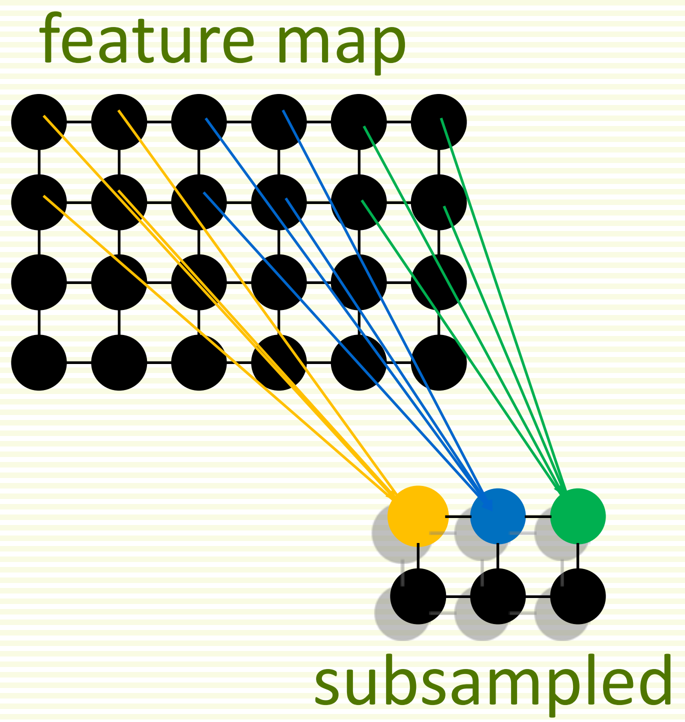
- Subsampling layers reduce spatial resolution of each feature map
  - weighted sum
- This achieves certain degree of shift and distortion invariance
- Weight sharing is also applied in subsampling layers
  - reduce the effect of noise and shift or distortion





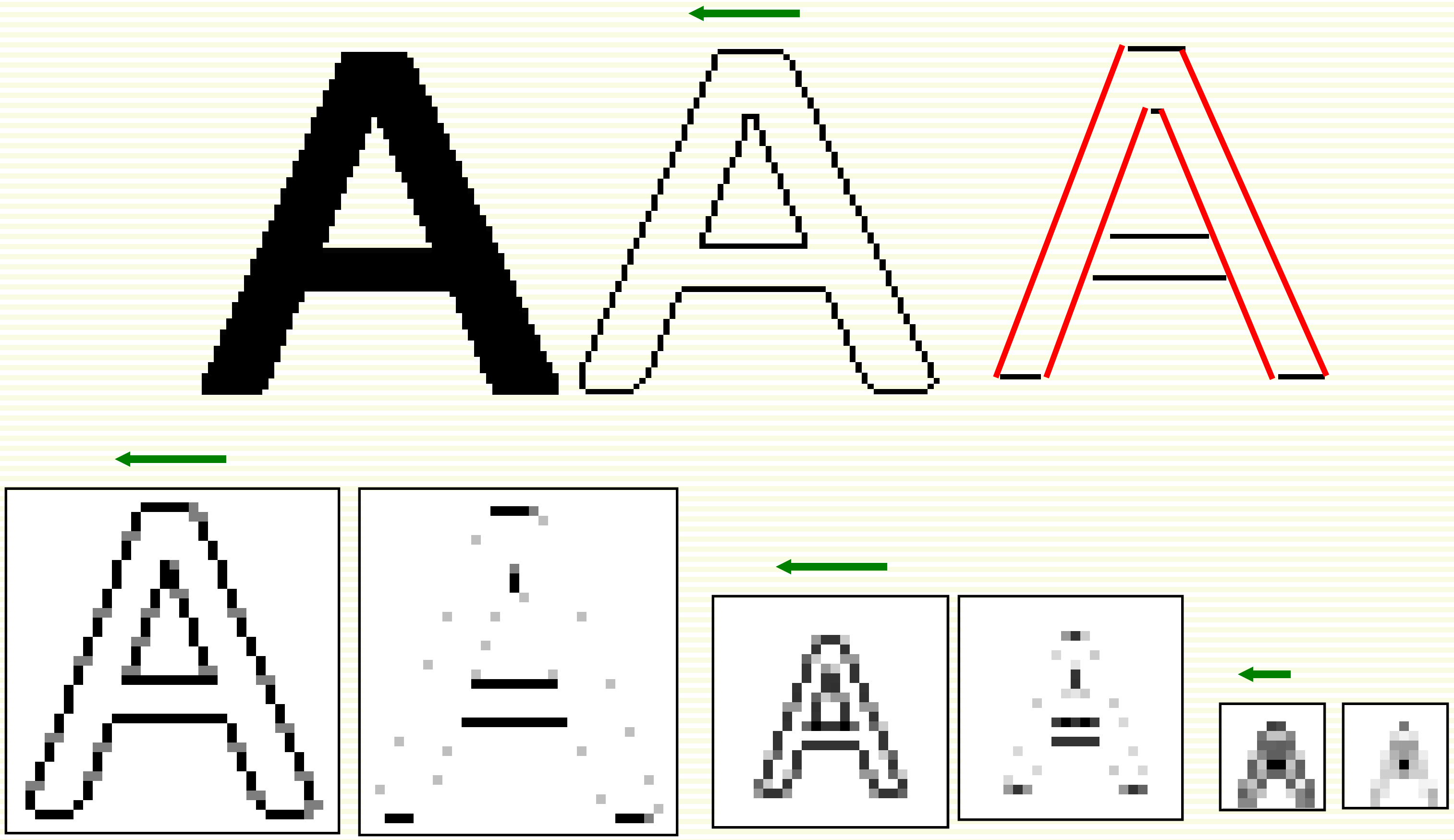
Subsampling Layer

- Subsampling is also called *pooling*
- Instead of subsampling, sometimes max pooling works betters
  - replace *weighted sum* with *max* operation

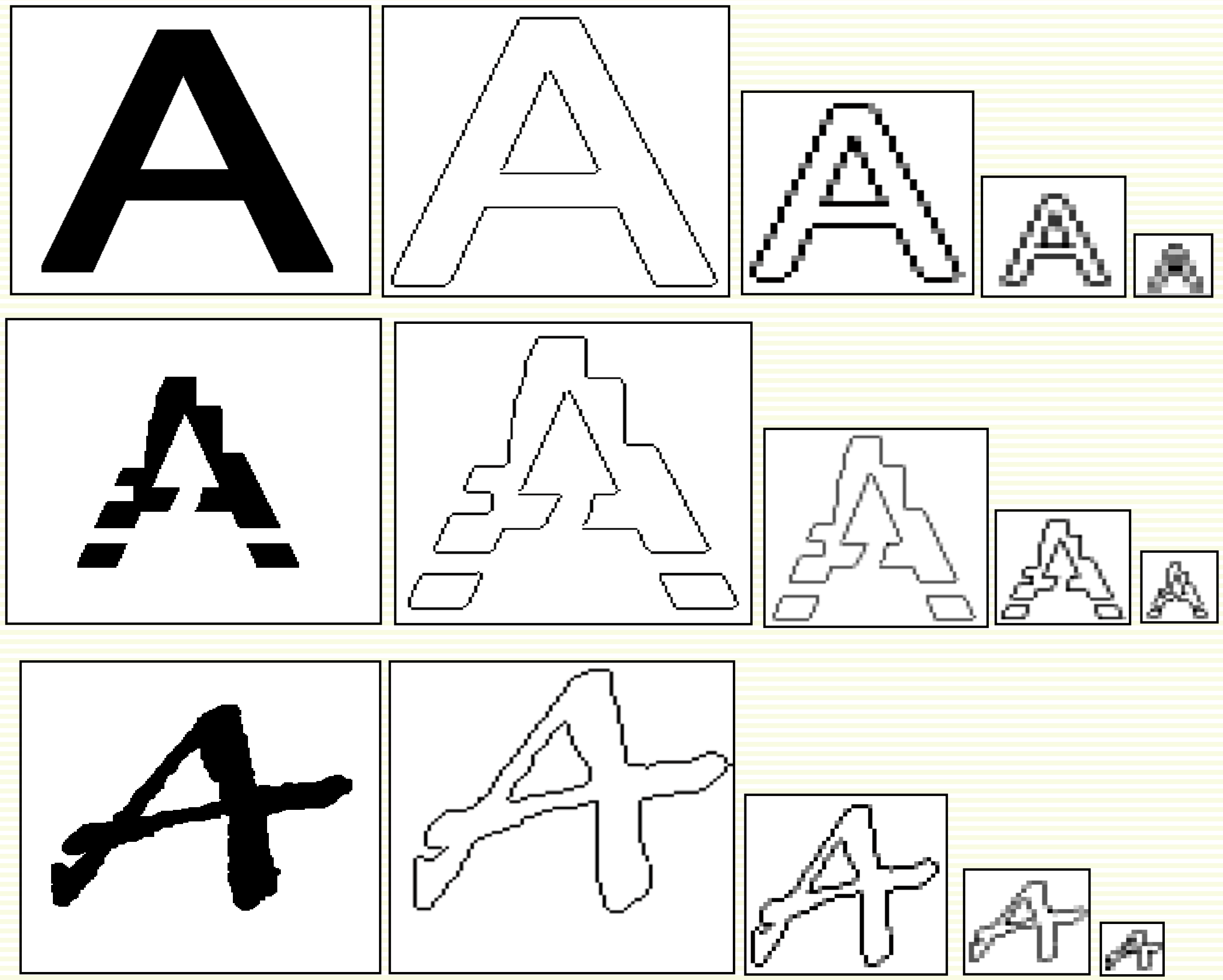


Subsampling Layer

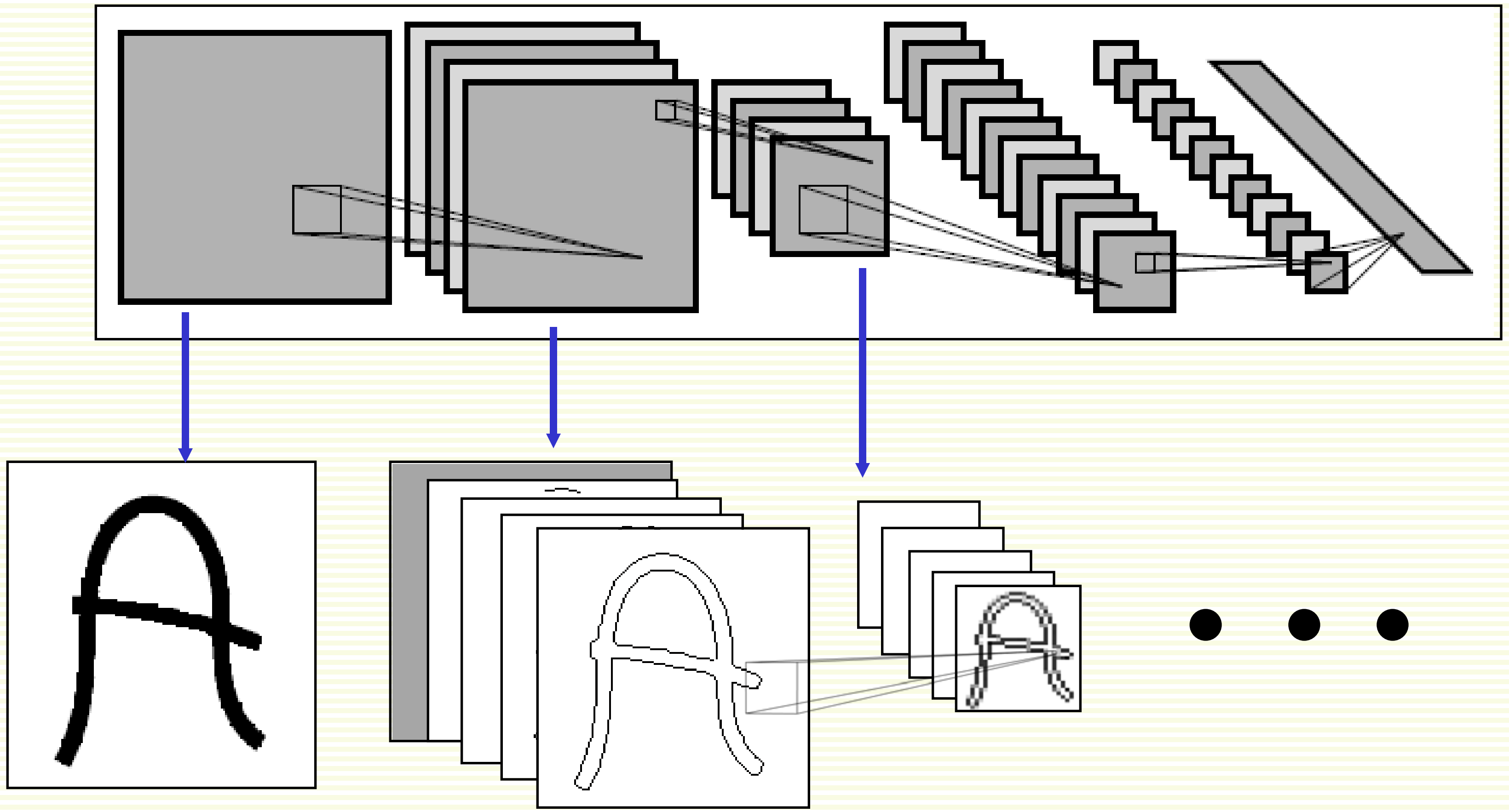
- Subsampling achieves certain degree of shift and distortion invariance is



Subsampling Layer



Convolutional Nets



Problem with Subsampling (Pooling)

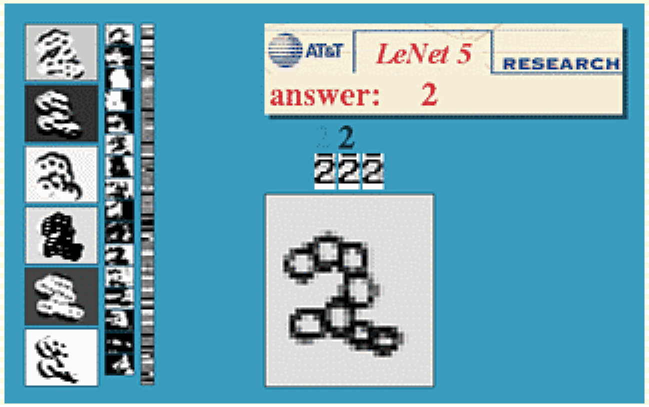
- Averaging (or taking a max) of four pixels gets a small amount of shift invariance
- Problem to be aware of
  - After several levels of pooling, we have lost information about the precise positions of things
  - This makes it impossible to use the precise spatial relationships between high-level parts for recognition.

Le Net

- Yann LeCun and his collaborators developed a good recognizer for handwritten digits by using backpropagation in a convoluitonal net
- LeNet uses knowledge about the invariance to design
  - local connectivity
  - weight-sharing
  - pooling
- This net was used for reading ~10% of the checks in North America
- Look the impressive demos of LENET at http://yann.lecun.com

Conv Nets: Character Recognition

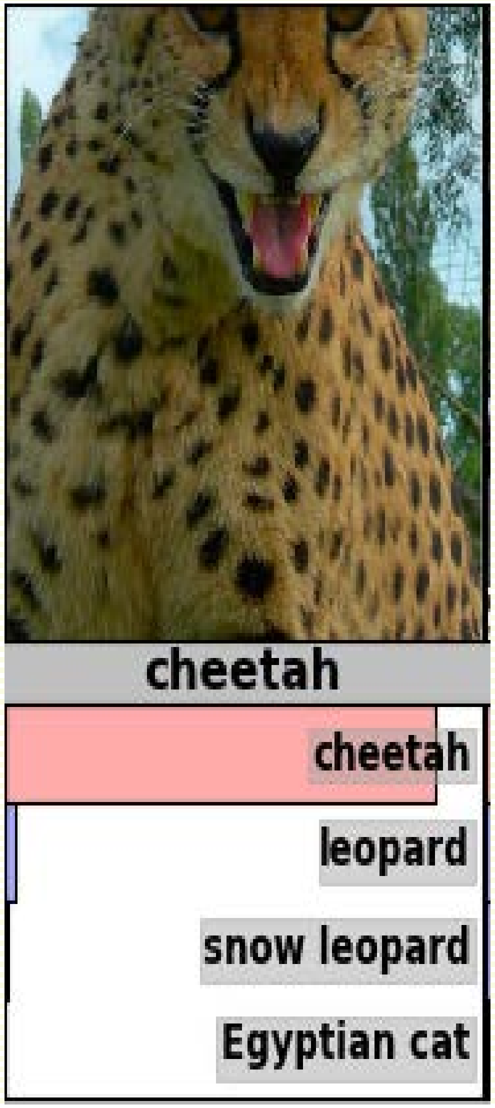
- http://yann.lecun.com/exdb/lenet/index.html

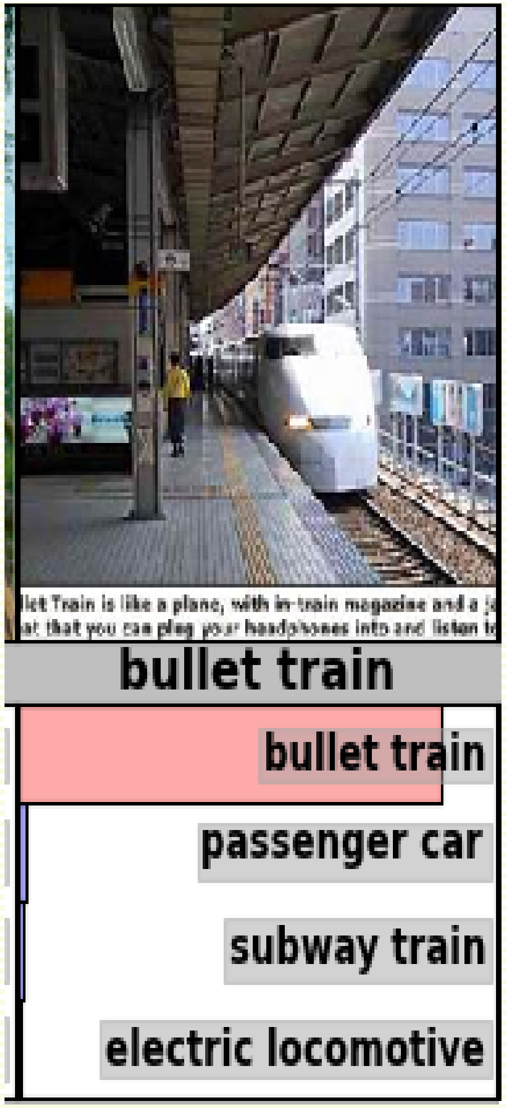


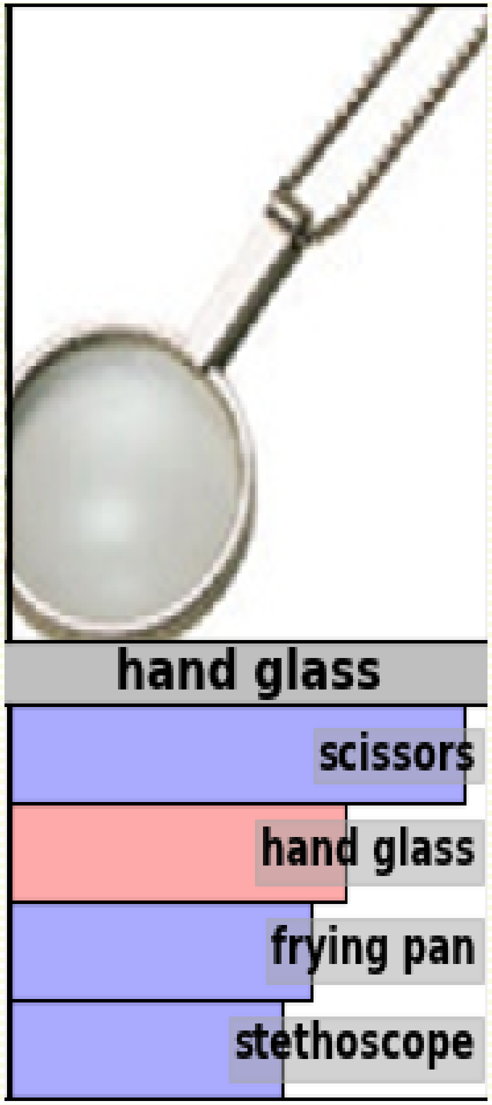
ConvNet for ImageNet

- 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

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

Training Deep Networks

- Difficulties of supervised training of deep networks
  - Early layers of MLN do not get trained well
    - Diffusion of Gradient – error attenuates as it propagates to earlier layers
    - Exacerbated since top couple layers can usually learn any task "pretty well" and thus the 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
  - Deep networks tend to have more local minima problems than shallow networks during supervised training

Greedy Layer-Wise Training

- Greedy layer-wise training to insure lower layers learn

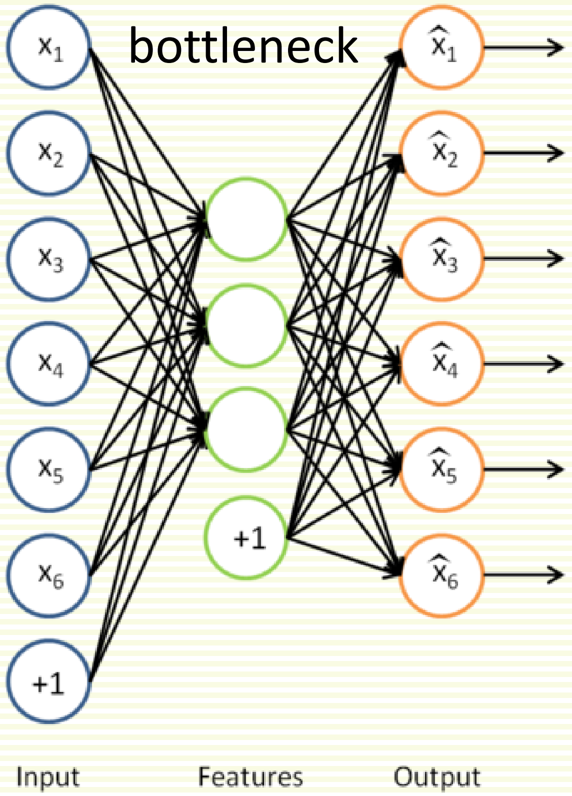
1. Train first layer using your data without the labels (unsupervised)
   - we do not know targets at this level anyway
   - can use the more abundant unlabeled data which is not part of the training set
2. Freeze the first layer parameters and start training the second layer using the output of the 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

Unsupervised Learning: Auto-Encoders

- A type of unsupervised learning which tries to discover generic features of the data
- Input sample = output
- Learn identity function by learning important sub-features, not by just passing through data



Concluding Remarks

- Advantages
  - MLP 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
  - May be difficult to analyze and predict its behavior
  - May take a long time to train
  - May get trapped in a bad local minima
  - A lot of tricks for successful implementation