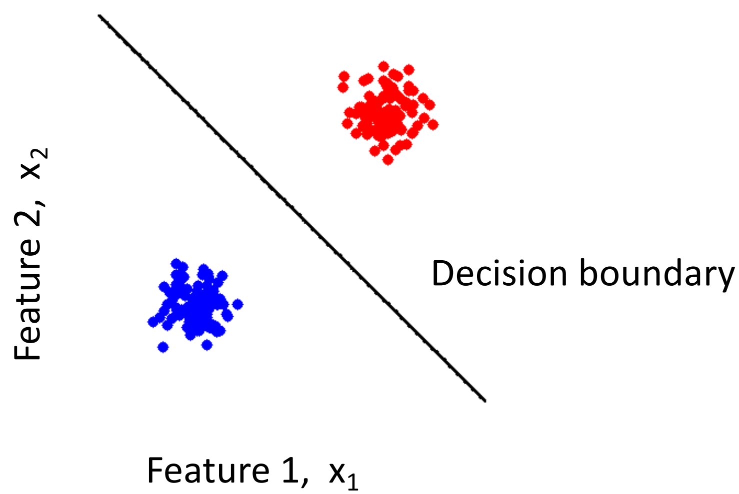
Linear classifiers (perceptrons)

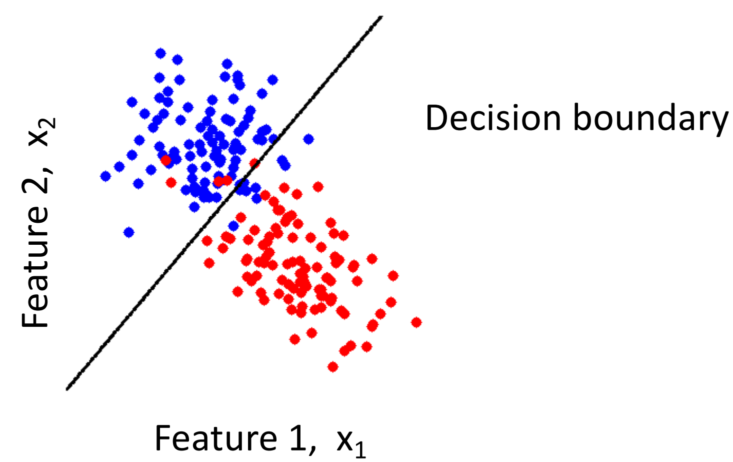
Linear Classifiers

- a linear classifier is a mapping that partitions feature space using a linear function (a straight line, or a hyperplane)
- separates the two classes using a straight line in feature space
- in 2 dimensions the decision boundary is a straight line

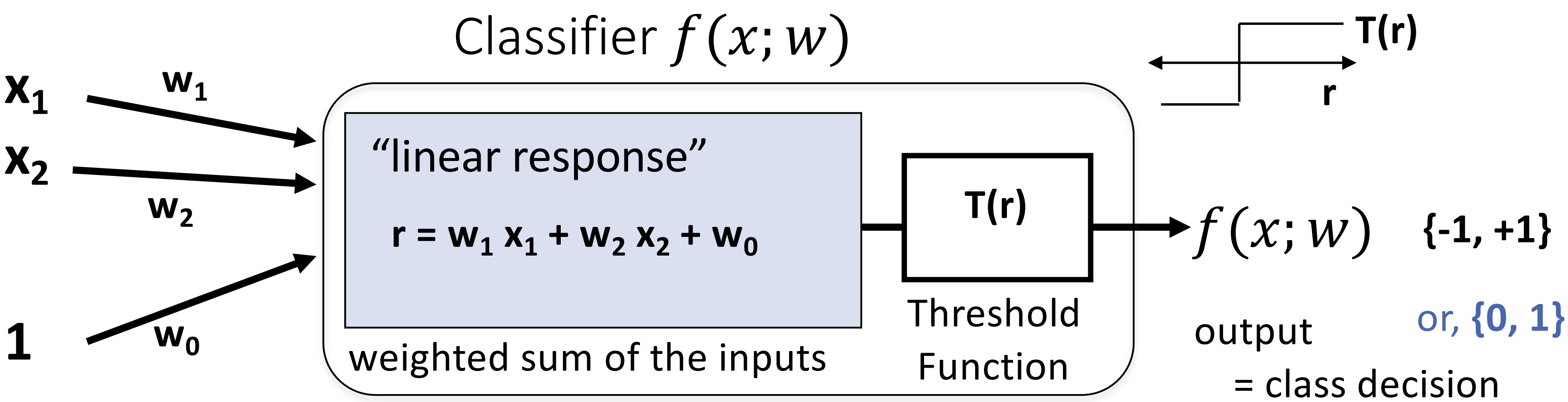
Linearly separable data



Linearly non-separable data



Perceptron Classifier (2 features)

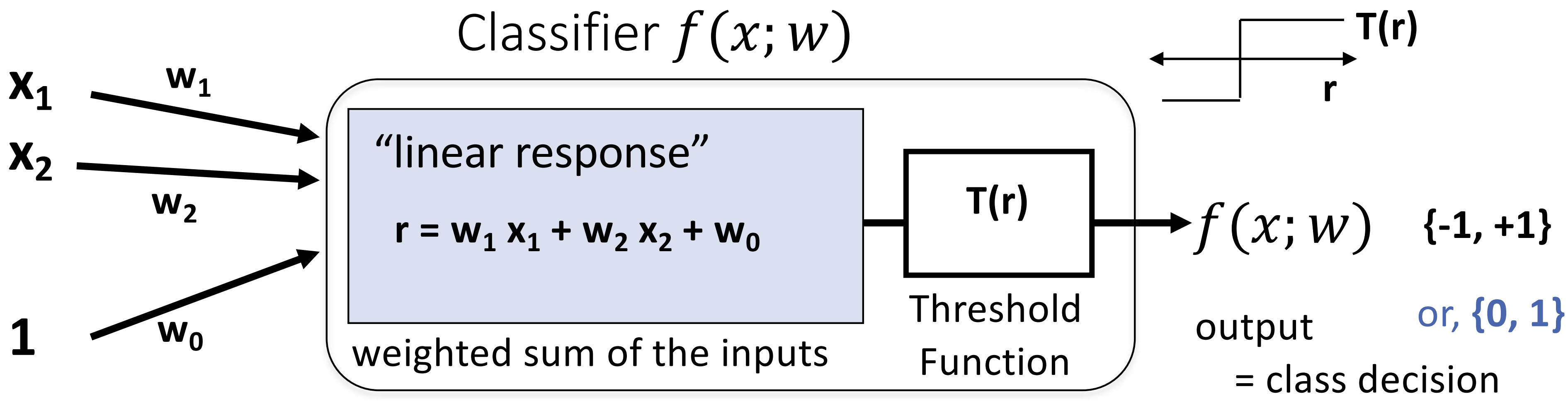



```
r = X.dot( theta.T );    # compute linear response
Yhat = 2*(r > 0)-1       # "sign": predict +1 / -1
```

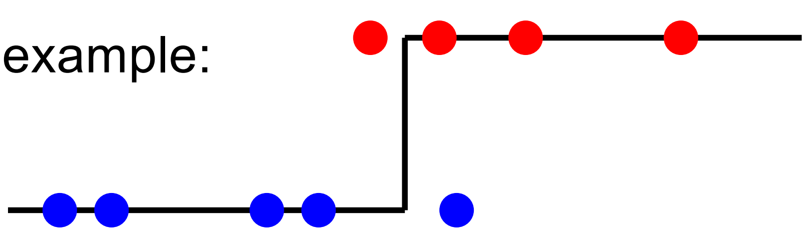
Decision Boundary at r(x) = 0

Solve: $X_2 = -w_1/w_2\, X_1 - w_0/w_2$ (Line)

Perceptron Classifier (2 features)



1D example:



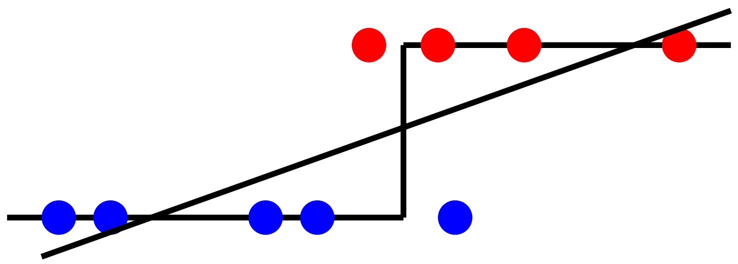
T(r) = -1 if r < 0
T(r) = +1 if r > 0

Decision boundary = "x such that T(w_1 x + w_0) transitions"

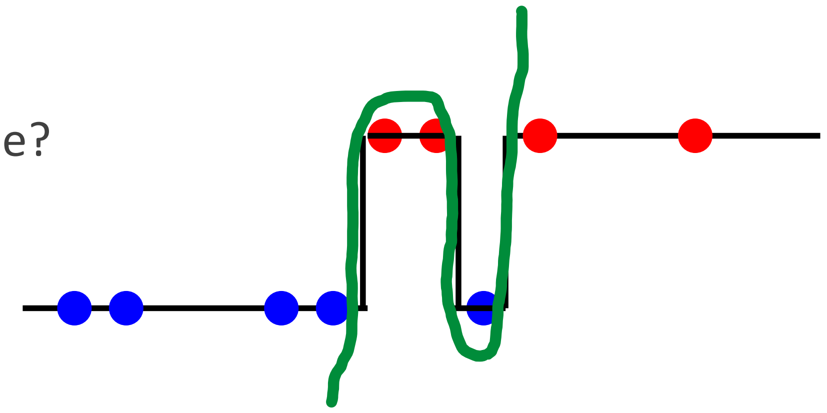
Features and perceptrons

Recall the role of features

- We can create extra features that allow more complex decision boundaries
- Linear classifiers
- Features [1,x]
  - Decision rule: T(ax+b) = ax + b >/< 0
  - Boundary ax+b =0 => point
- Features [1,x,x^2]
  - Decision rule T(ax^2+bx+c)
  - Boundary ax^2+bx+c = 0 = ?

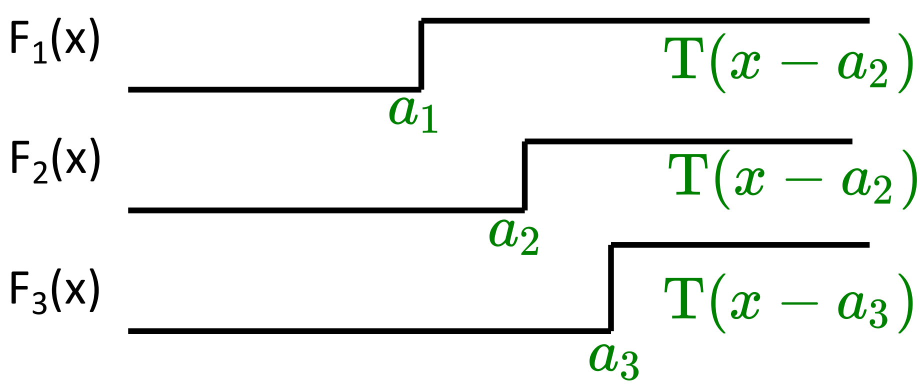


- What features can produce this decision rule?



Idea: use combinations of step functions

Using combinations of step functions, we can build more complex decision boundaries.



Linear function of features

$w_1\, F_1(x) + w_2 F_2\,(x) + w_3 F_3(x) + w_4$

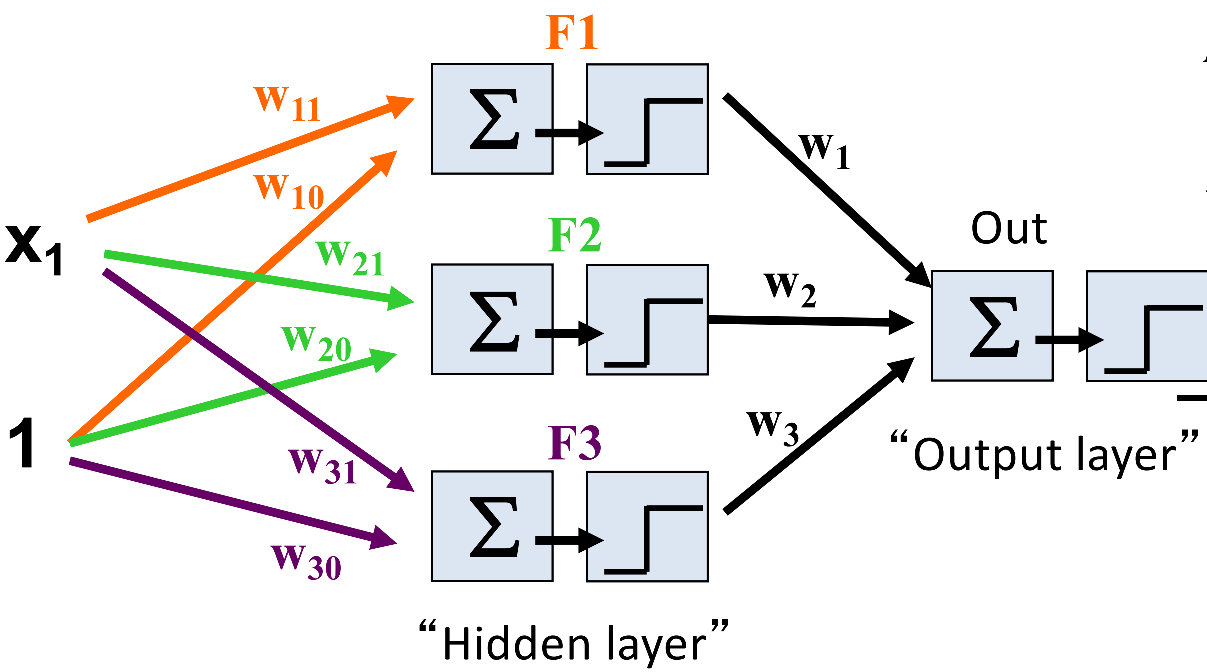
Example: $F_1 - F_2 + F_3$

Goal: learn optimal thresholds a_1, a_2, a_3 and function weights w_1, w_2, w_3, w_4.
This is a simple neural network (this will be explained in more detail)

Multi-layer perceptron model

Step functions are just perceptrons!

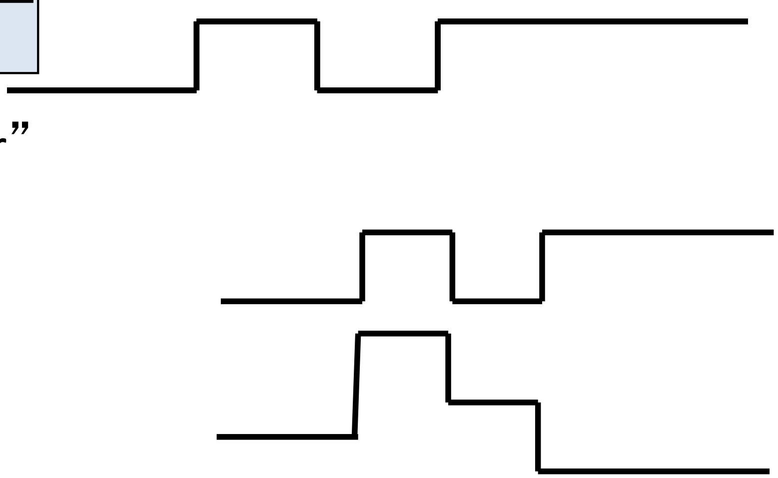
- Idea: instead of using original features, use outputs of other perceptrons



$F_i(x) = T(w_{i1}\, x_1 + w_{i0})$

Linear function of features

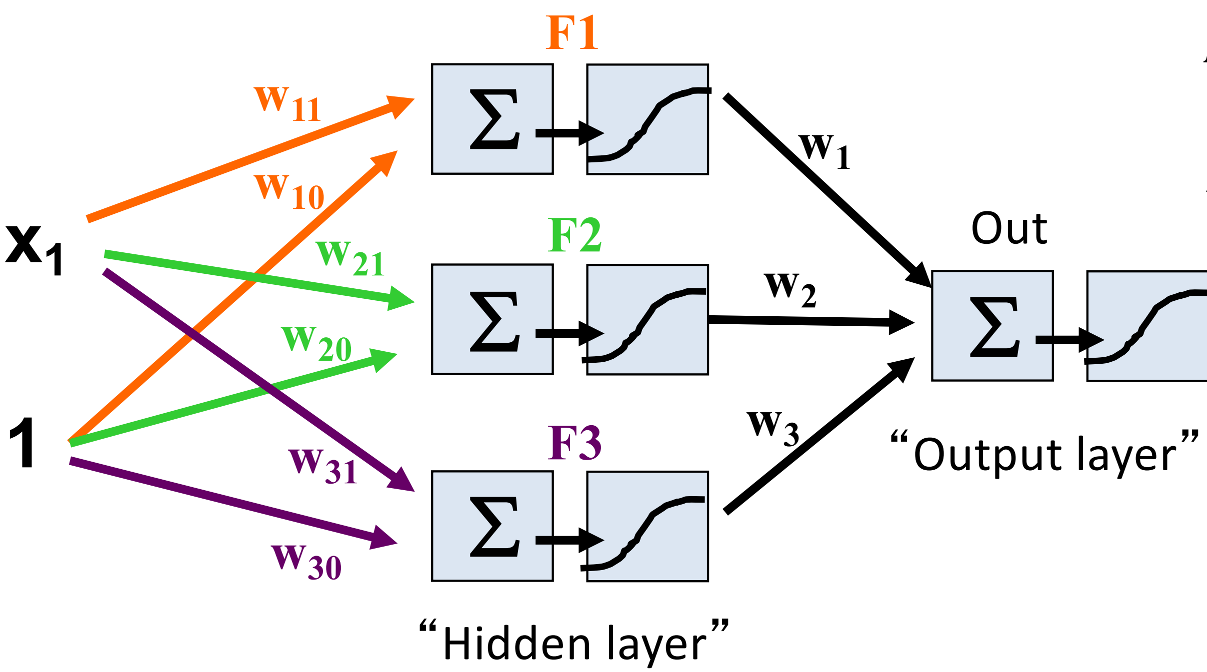
$w_1\, F_1(x) + w_2 F_2\,(x) + w_3 F_3(x) + w_4$



Multi-layer perceptron model

Step functions are just perceptrons!

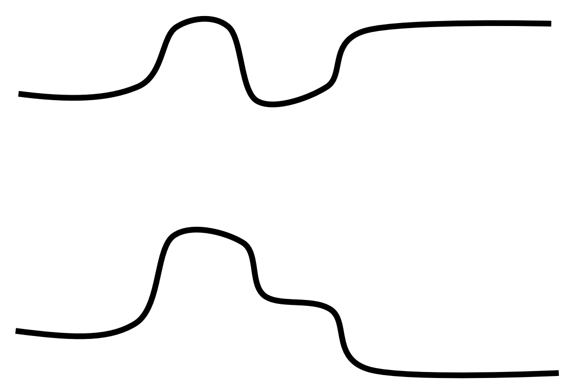
- Idea: instead of using original features, use outputs of other perceptrons



$F_i(x) = T(w_{i1}\, x_1 + w_{i0})$

Linear function of features

$w_1\, F_1(x) + w_2 F_2\,(x) + w_3 F_3(x) + w_4$

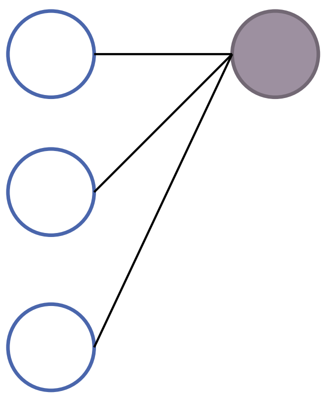


Features of MLPs

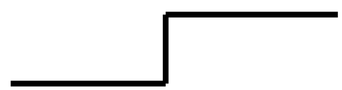
Simple building blocks

- Each element is just a perceptron

Can build upwards



Input Features



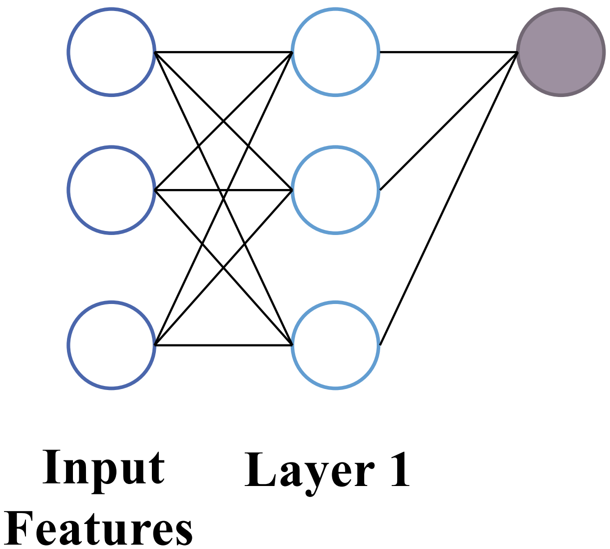
Perceptron:
Step function /
Linear partition

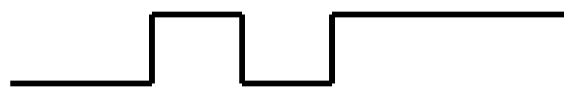
Features of MLPs

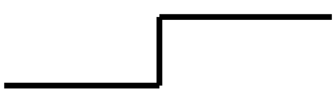
Simple building blocks

- Each element is just a perceptron

Can build upwards







2-layer:

"Features" are now partitions

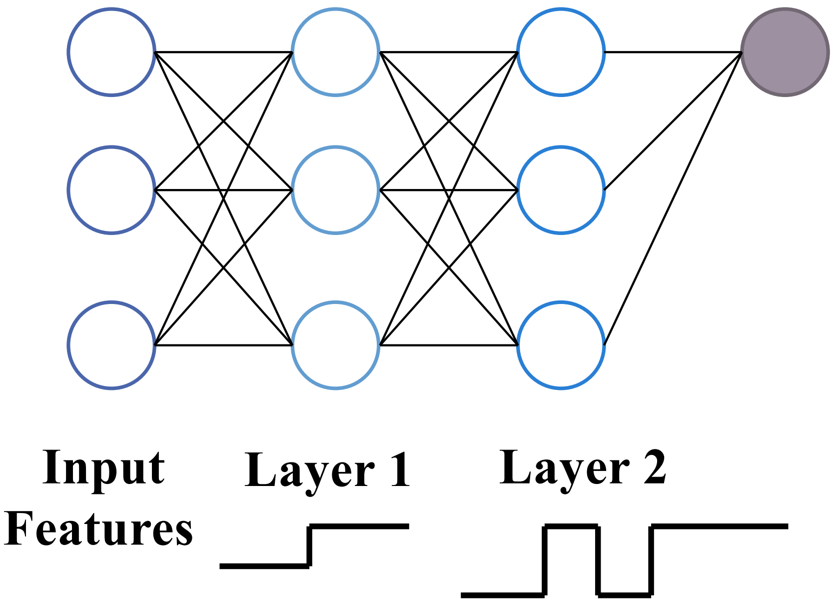
All linear combinations of those partitions

Features of MLPs

Simple building blocks

- Each element is just a perceptron

Can build upwards



3-layer:

"Features" are now complex functions

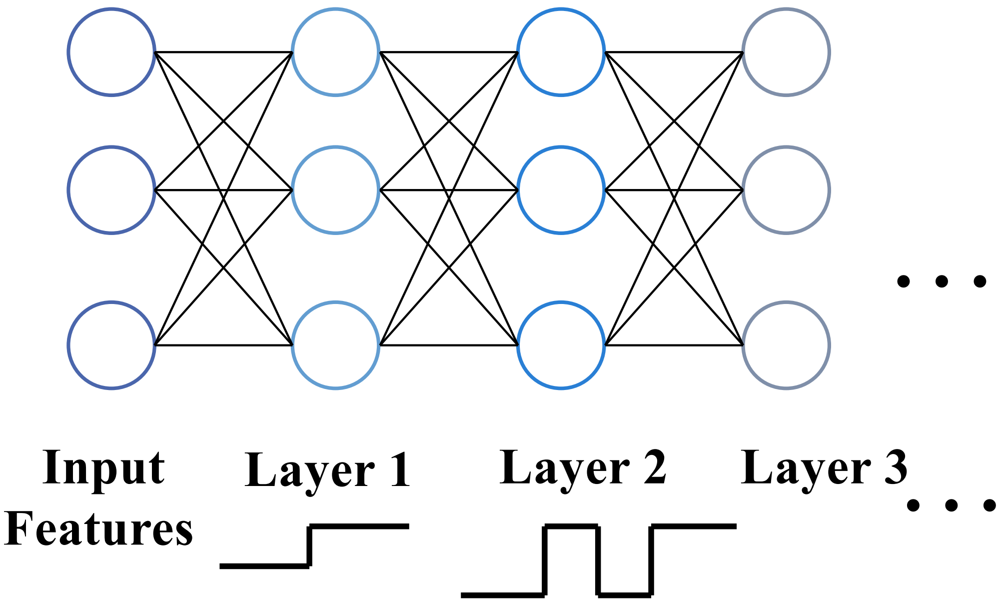
Output any linear combination of those

Features of MLPs

Simple building blocks

- Each element is just a perceptron

Can build upwards



Current research:
"Deep" architectures
(many layers)

Features of MLPs

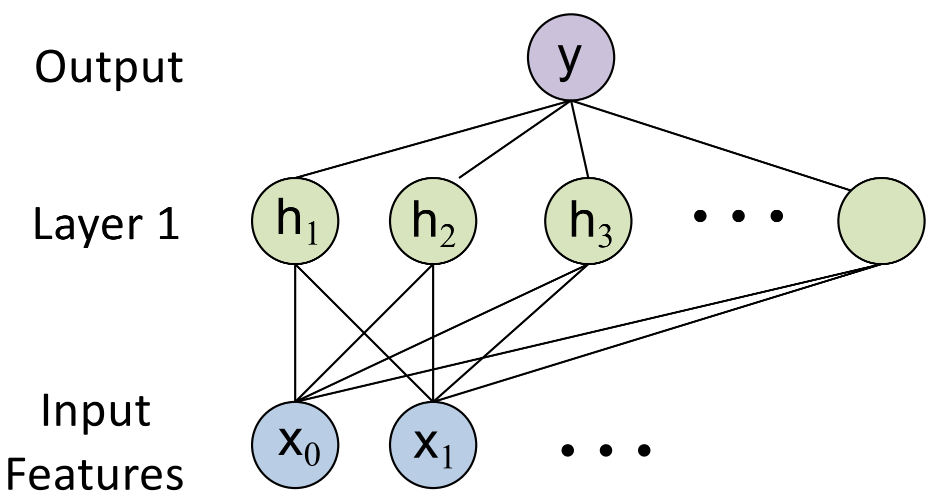
Simple building blocks

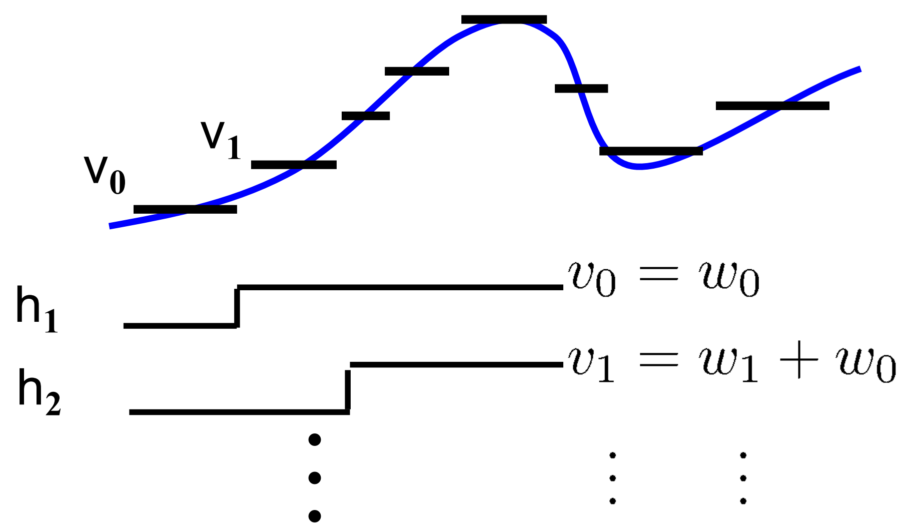
- Each element is just a perceptron

Can build upwards

Flexible function approximation

- Approximate arbitrary functions with enough hidden nodes

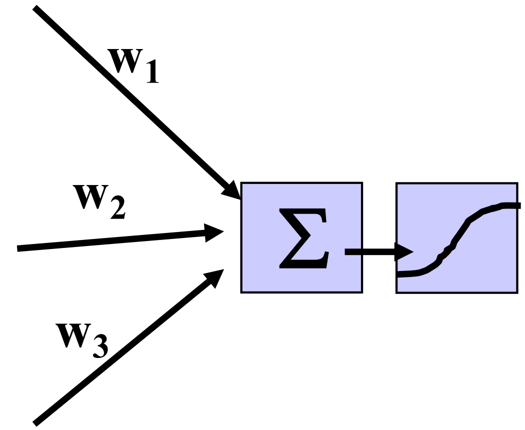




Neural networks

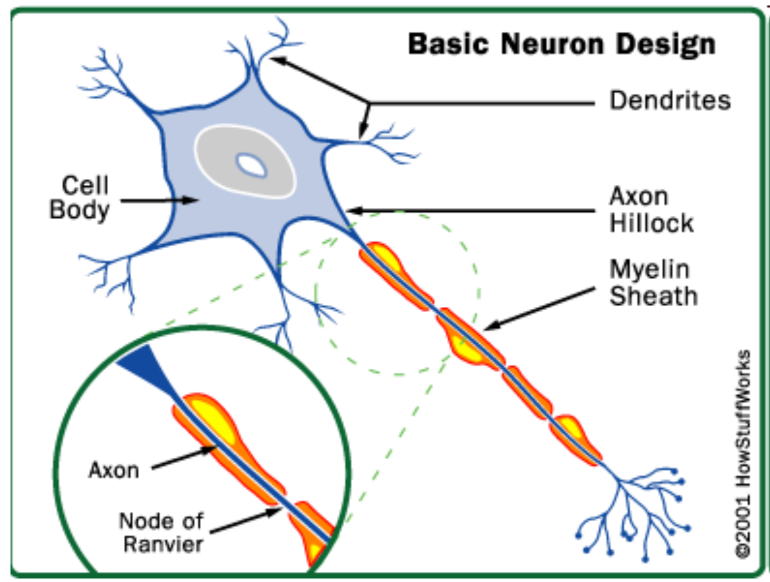
Another term for MLPs

Biological motivation



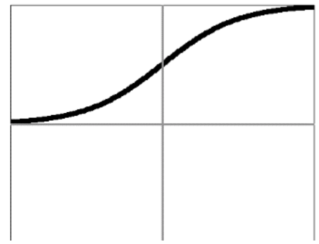
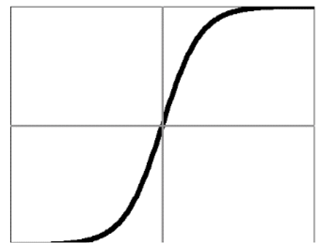
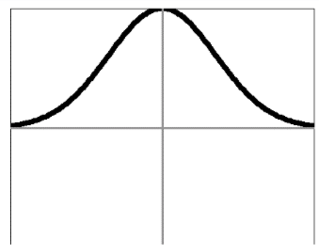
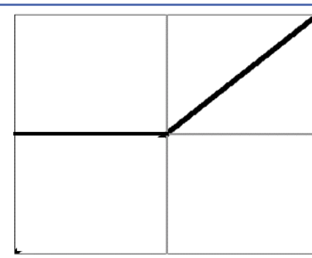
Neurons

- “Simple” cells
- Dendrites sense charge
- Cell weighs inputs
- “Fires” axon



“How stuff works the brain”

Activation functions

Logistic	$\sigma(z) = \frac{1}{1+\exp(-z)}$		$\frac{\partial \sigma}{\partial z}(z) = \sigma(z)(1-\sigma(z))$
Hyperbolic Tangent	$\sigma(z) = \frac{1-\exp(-2z)}{1+\exp(-2z)}$		$\frac{\partial \sigma}{\partial z}(z) = 1-(\sigma(z))^2$
Gaussian	$\sigma(z) = \exp(-z^2/2)$		$\frac{\partial \sigma}{\partial z}(z) = -z\sigma(z)$
ReLU (rectified linear)	$\sigma(z) = \max(0, z)$		$\frac{\partial \sigma}{\partial z}(z) = \mathbb{1}[z>0]$
Linear	$\sigma(z) = z$		and many others…

Feed-forward networks

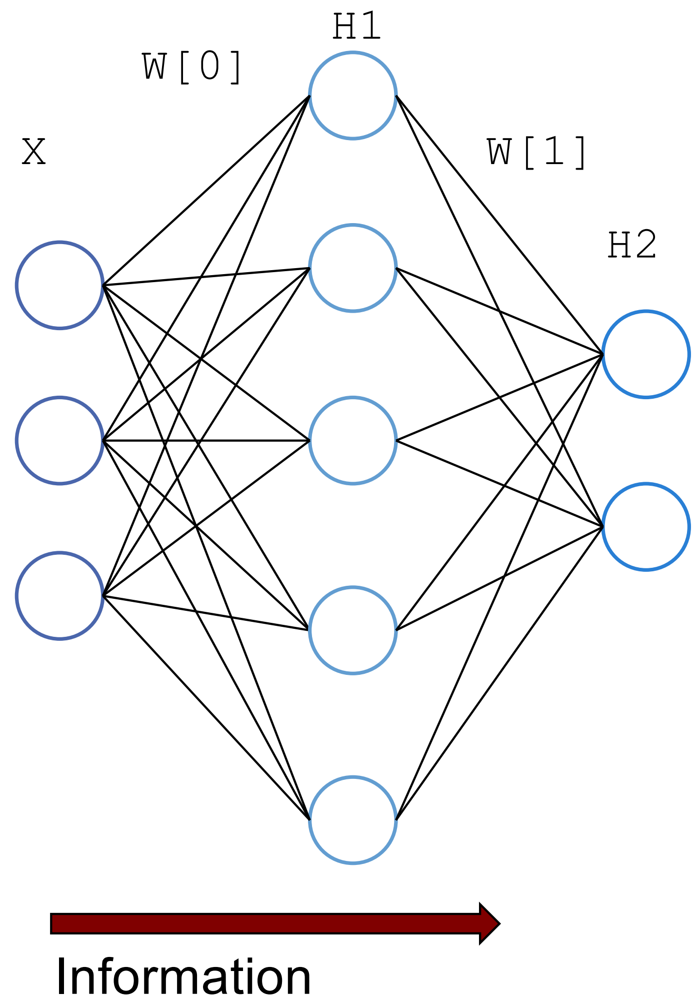
Information flows left-to-right
- Input observed features
- Compute hidden nodes (parallel)
- Compute next layer…

```
R = X.dot(W[0])+B[0]; # linear response
H1= Sig( R );         # activation f'n

S = H1.dot(W[1])+B[1]; # linear response
H2 = Sig( S );         # activation f'n

% ...
```

Alternative: recurrent NNs…



Feed-forward networks

A note on multiple outputs:

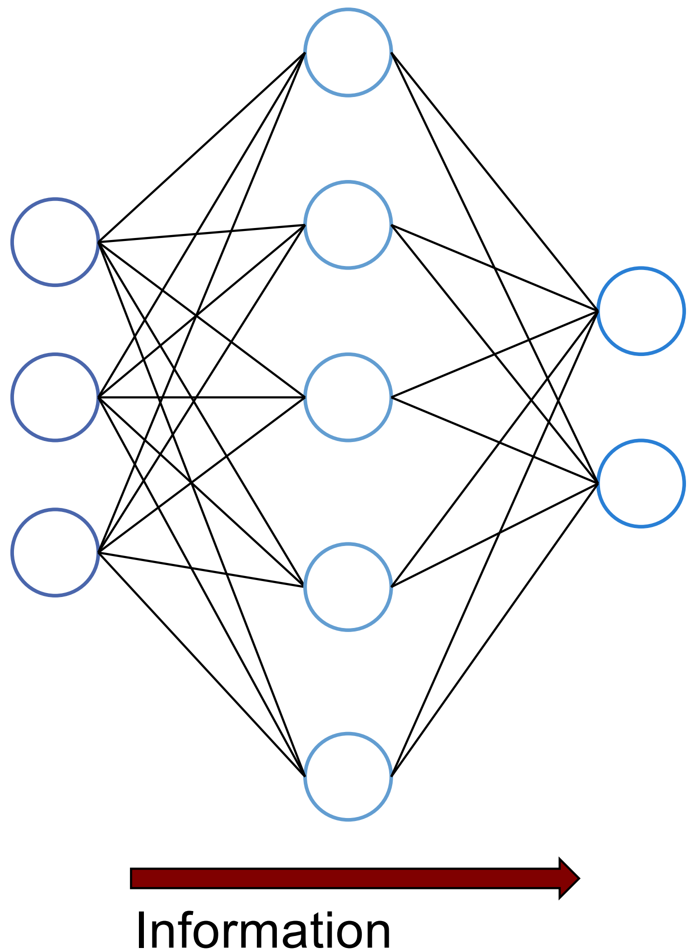
Regression:

- Predict multi-dimensional y
- “Shared” representation
- = fewer parameters

Classification

- Predict binary vector
- Multi-class classification

  y = 2 = [0 0 1 0 …]
- Multiple, joint binary predictions (image tagging, etc.)
- Often trained as regression (MSE) with saturating activation



Machine Learning

Multilayer Perceptrons

Learning: Backpropagation

Advanced Neural Networks

Training MLPs

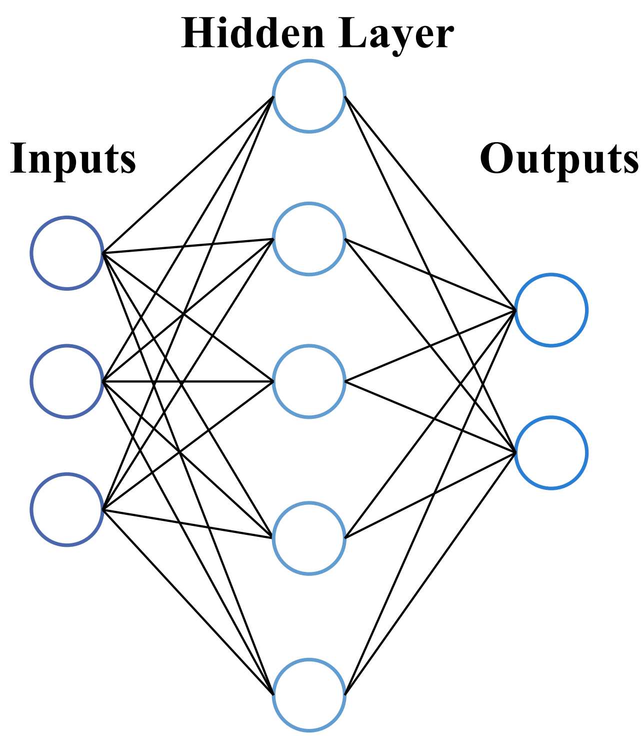
Observe features "x" with target "y"

Push "x" through NN = output is "ŷ"

Error: $(y-\hat{y})^2$

(Can use different loss functions if desired…)

How should we update the weights to improve?



Single layer

- Logistic sigmoid function
- Smooth, differentiable

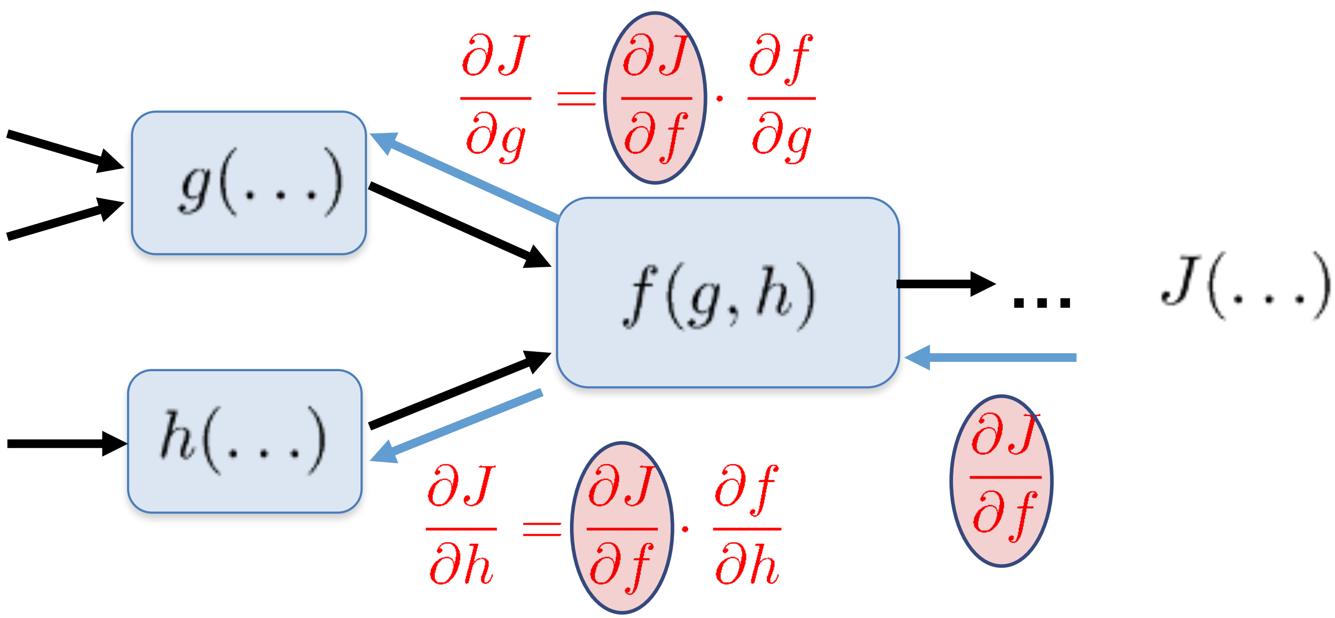
Optimize using:

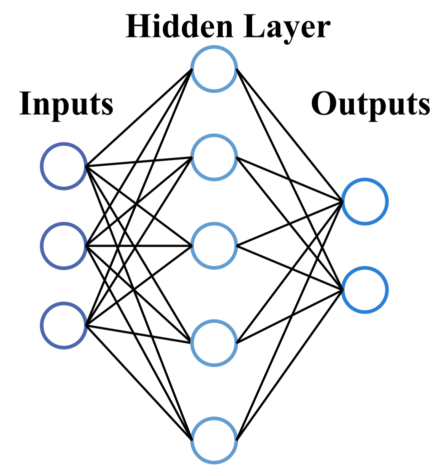
- Batch gradient descent
- Stochastic gradient descent

Gradient calculations

Think of NNs as composition of smaller functions

- Building blocks: summations & nonlinearities
- For derivatives, just apply the chain rule (and re-use partial derivatives)!





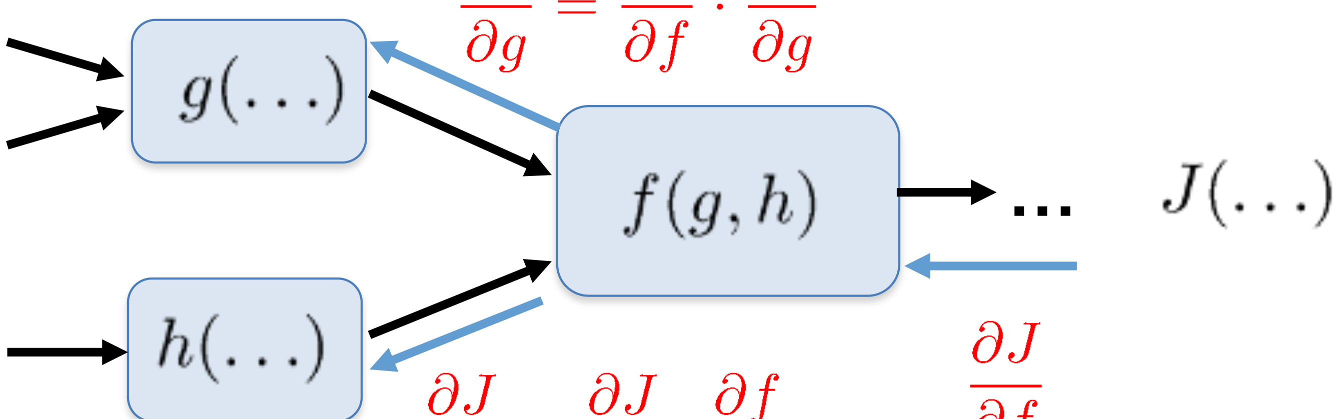
- Backpropagation is just the chain rule (+ tricks to avoid re-computations)

Gradient calculations

Think of NNs as compositions of smaller functions

- Building blocks: summations & nonlinearities

$$\frac{\partial J}{\partial g} = \frac{\partial J}{\partial f} \cdot \frac{\partial f}{\partial g}$$



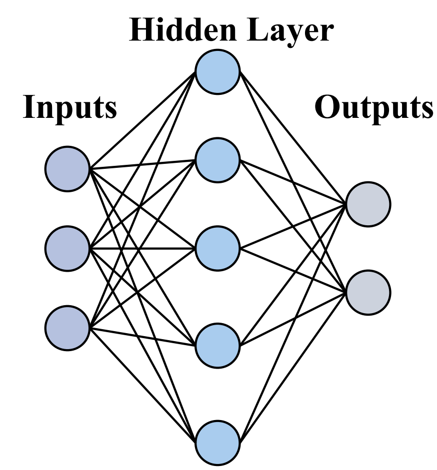
$$\frac{\partial J}{\partial h} = \frac{\partial J}{\partial f} \cdot \frac{\partial f}{\partial h}$$

$$\frac{\partial J}{\partial f}$$

Ex: $f(g,h) = g^2 h$

$$\frac{\partial J}{\partial g} = \frac{\partial J}{\partial f} \cdot 2\, g(\cdot)\, h(\cdot) \qquad \frac{\partial J}{\partial h} = \frac{\partial J}{\partial f} \cdot g^2(\cdot)$$

save & reuse info (g,h) from forward computation!



Backpropagation

Forward pass

Loss function

$$J_i(W) = \sum_k \, (y_k^{(i)} - \hat{y}_k^{(i)})^2$$

Output layer

$$\hat{y}_k = \sigma(s_k) = \sigma(\textstyle\sum_j w_{kj}^2 h_j)$$

Hidden layer

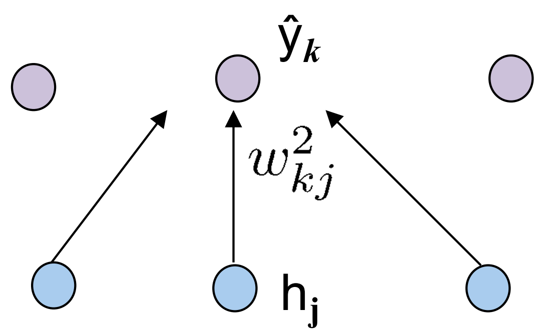
$$h_j = \sigma(t_j) = \sigma(\textstyle\sum_i w_{ji}^1 x_i)$$

Just gradient descent… (the chain rule)

$$\frac{\partial J}{\partial w_{kj}^2} = -2 \sum_{k'} (y_{k'} - \hat{y}_{k'}) \; (\partial \hat{y}_{k'})$$

$$= -2(y_k - \hat{y}_k) \; \sigma'(s_k) \; h_j$$

(Identical to logistic mse regression with inputs "h_j")



Backpropagation

Just gradient descent… (the chain rule)

$$\frac{\partial J}{\partial w^2_{kj}} = -2\sum_{k'}(y_{k'} - \hat{y}_{k'})\ (\partial \hat{y}_{k'})$$

$$= \underbrace{\boxed{-2(y_k - \hat{y}_k)\ \sigma'(s_k)}}_{\beta^2_k}\ h_j$$

(Identical to logistic mse regression with inputs "h_j")

Forward pass

Loss function

$$J_i(W) = \sum_k \left(y_k^{(i)} - \hat{y}_k^{(i)}\right)^2$$

Output layer

$$\hat{y}_k = \sigma(s_k) = \sigma\left(\sum_j w^2_{kj} h_j\right)$$

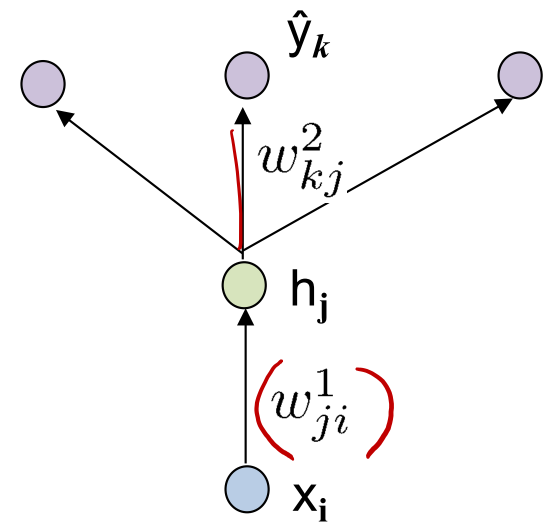
Hidden layer

$$h_j = \sigma(t_j) = \sigma\left(\sum_i w^1_{ji} x_i\right)$$

$$\frac{\partial J}{\partial w^1_{ji}} = \sum_k -2(y_k - \hat{y}_k)\ (\partial \hat{y}_k)$$

$$= \sum_k -2(y_k - \hat{y}_k)\ \sigma'(s_k)\ w^2_{kj}\ \partial h_j$$

$$= \sum_k \underbrace{\boxed{-2(y_k - \hat{y}_k)\ \sigma'(s_k)}}_{\beta^2_k}\ w^2_{kj}\ \sigma'(t_j)\ x_i$$



Backpropagation

Just gradient descent… (the chain rule)

$$\frac{\partial J}{\partial w^2_{kj}} = \boxed{-2(y_k - \hat{y}_k)\ \sigma'(s_k)}\ h_j$$

β^2_k

$$\frac{\partial J}{\partial w^1_{ji}} = \sum_k \boxed{-2(y_k - \hat{y}_k)\ \sigma'(s_k)}\ w^2_{kj}\ \sigma'(t_j)\ x_i$$

Forward pass

Loss function

$$J_i(W) = \sum_k (y_k^{(i)} - \hat{y}_k^{(i)})^2$$

Output layer

$$\hat{y}_k = \sigma(s_k) = \sigma(\sum_j w^2_{kj} h_j)$$

Hidden layer

$$h_j = \sigma(t_j) = \sigma(\sum_i w^1_{ji} x_i)$$

```
% X  : (1xN1)
H   = Sig(X1.dot(W[0]))
% W1 : (N2 x N1+1)
% H  : (1xN2)
Yh  = Sig(H1.dot(W[1]))
% W2 : (N3 x N2+1)
% Yh : (1xN3)
```

```
B2 = (Y-Yhat) * dSig(S)  #(1xN3)
G2 = B2.T.dot( H )       #(N3x1)*(1xN2)=(N3xN2)

B1 = B2.dot(W[1])*dSig(T)#(1xN3).(N3*N2)*(1xN2)
G1 = B1.T.dot( X )       #(N2 x N1+1)
```

Example: Regression, MCycle data

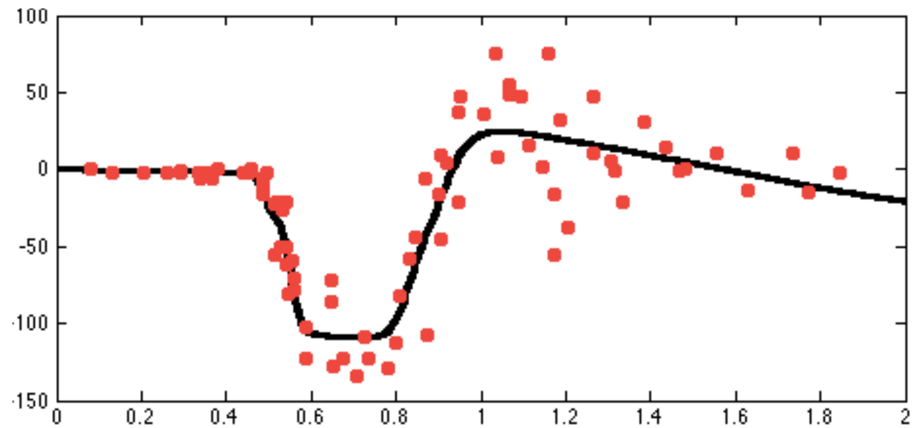
Train NN model, 2 layer

- 1 input features => 1 input units
- 10 hidden units
- 1 target => 1 output units
- Logistic sigmoid activation for hidden layer, linear for output layer

Data:

+

learned prediction f'n:

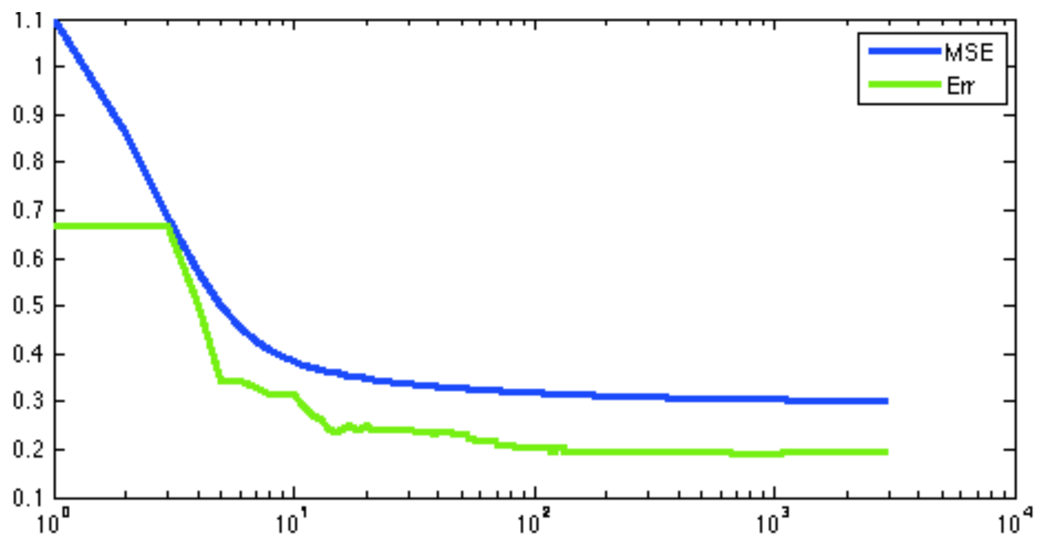


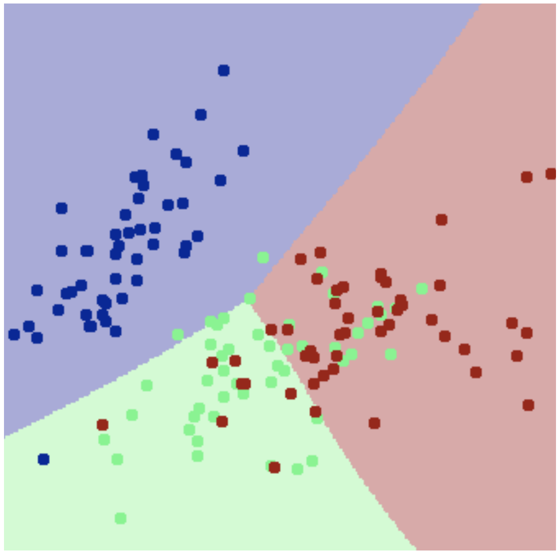
Responses of hidden nodes
(= features of linear regression):
select out useful regions of "x"

Example: Classification, Iris data

Train NN model, 2 layer

- 2 input features => 2 input units
- 10 hidden units
- 3 classes => 3 output units   (y = [0 0 1], etc.)
- Logistic sigmoid activation functions
- Optimize MSE of predictions using stochastic gradient





Demo Time!

http://playground.tensorflow.org/

Machine Learning

Multilayer Perceptrons

Learning: Backpropagation

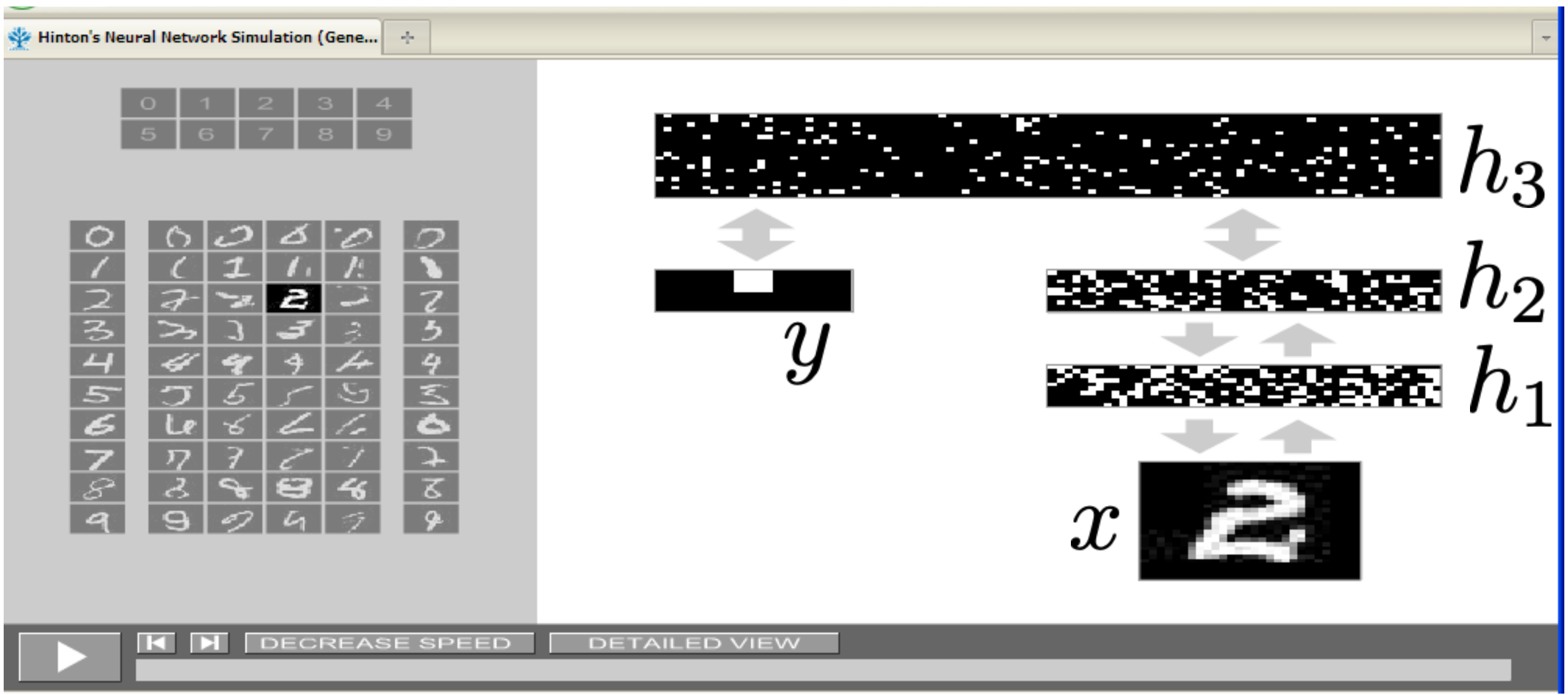
Advanced Neural Networks

[Hinton et al. 2007]

MLPs in practice

Example: Deep belief nets

- Handwriting recognition (Online demo)
- 784 pixels ⇔ 500 mid ⇔ 500 high ⇔ 2000 top ⇔ 10 labels



[Hinton et al. 2007]

MLPs in practice

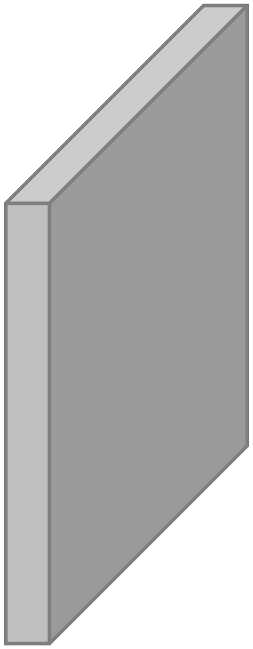
Example: Deep belief nets

- Handwriting recognition (Online demo)
- 784 pixels ⇔ 500 mid ⇔ 500 high ⇔ 2000 top ⇔ 10 labels

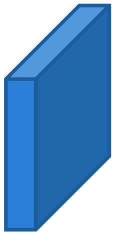
Convolutional networks

Organize & share the NN weights (vs “dense”), Group into “filters”

Input: 28x28 image

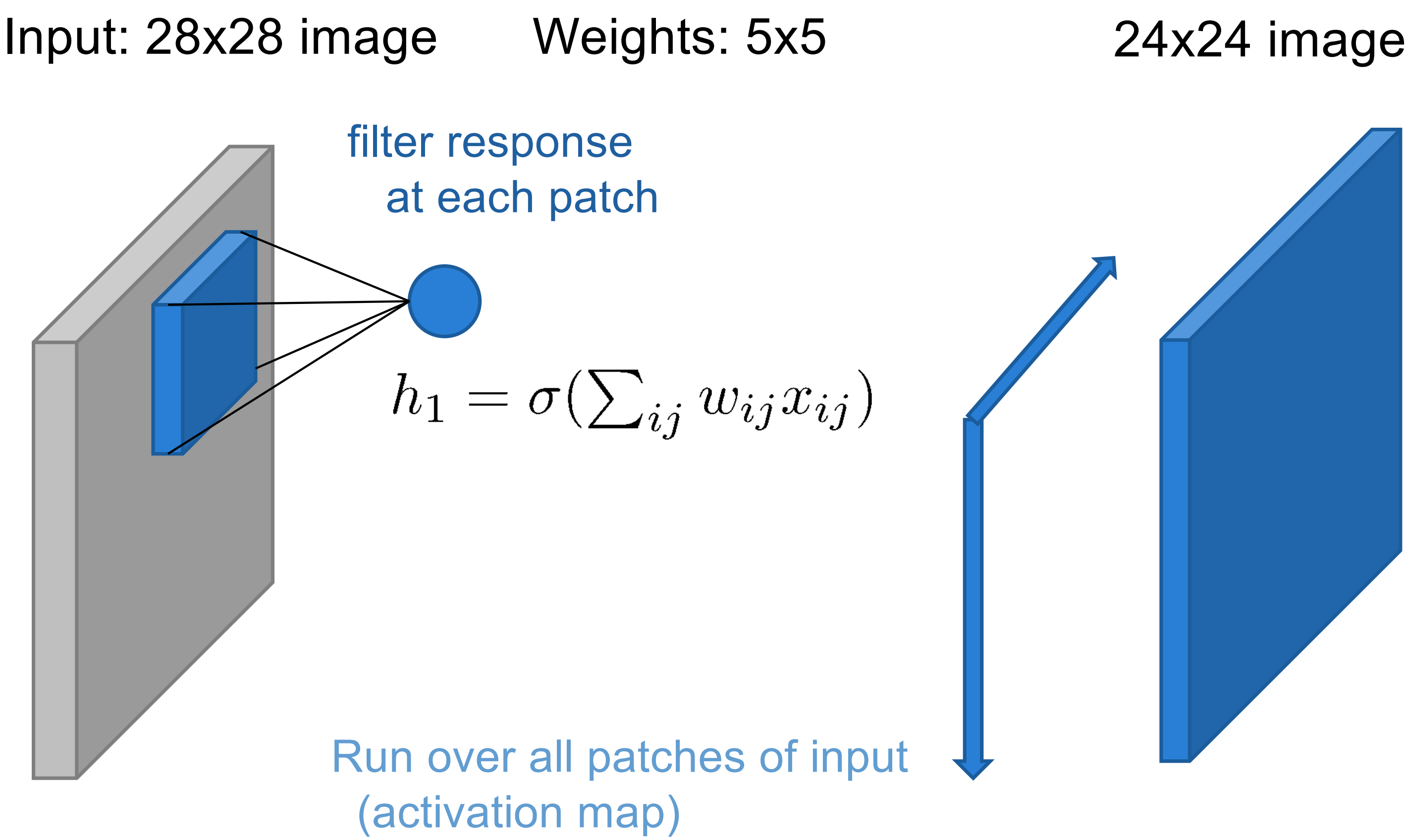


Weights: 5x5



Convolutional networks

Organize & share the NN weights (vs “dense”), Group into “filters”



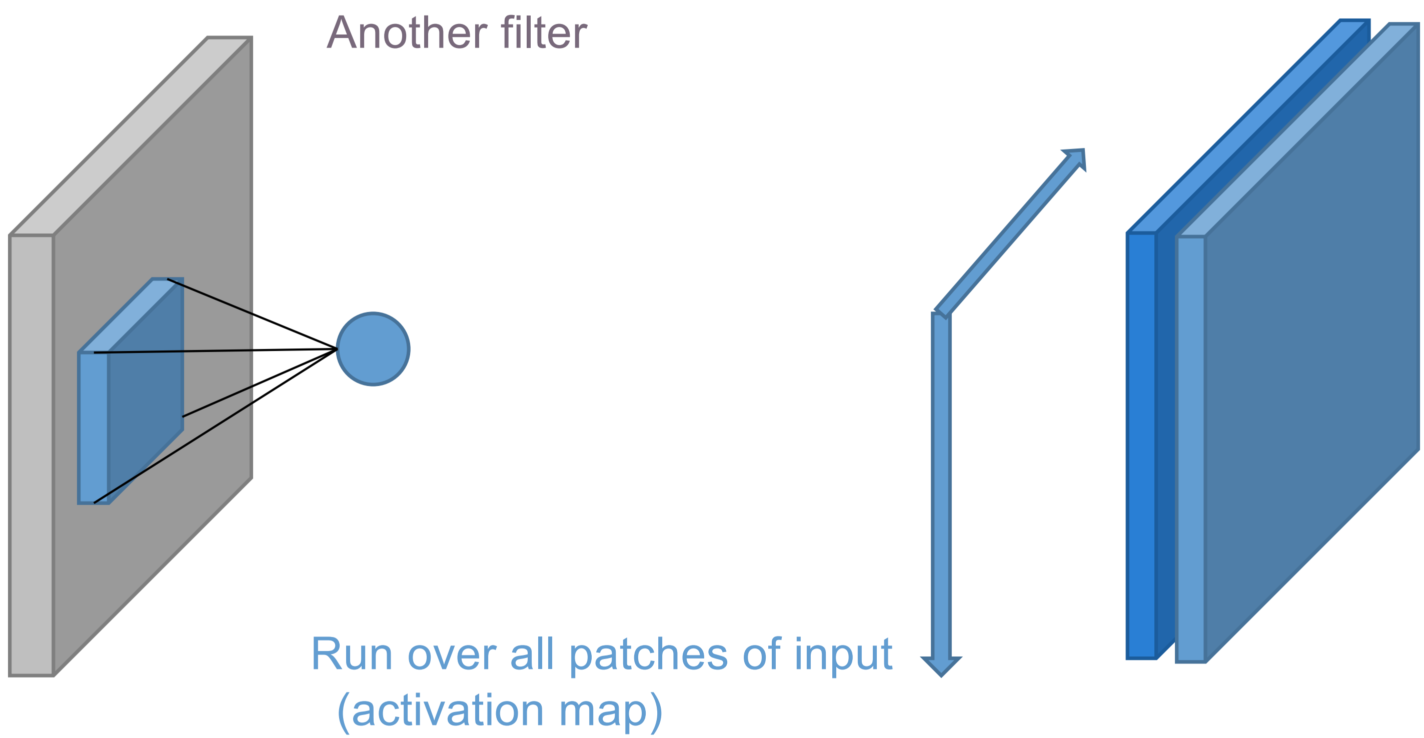
$$h_1 = \sigma(\sum_{ij} w_{ij} x_{ij})$$

Convolutional networks

Organize & share the NN weights (vs “dense”), Group into “filters”

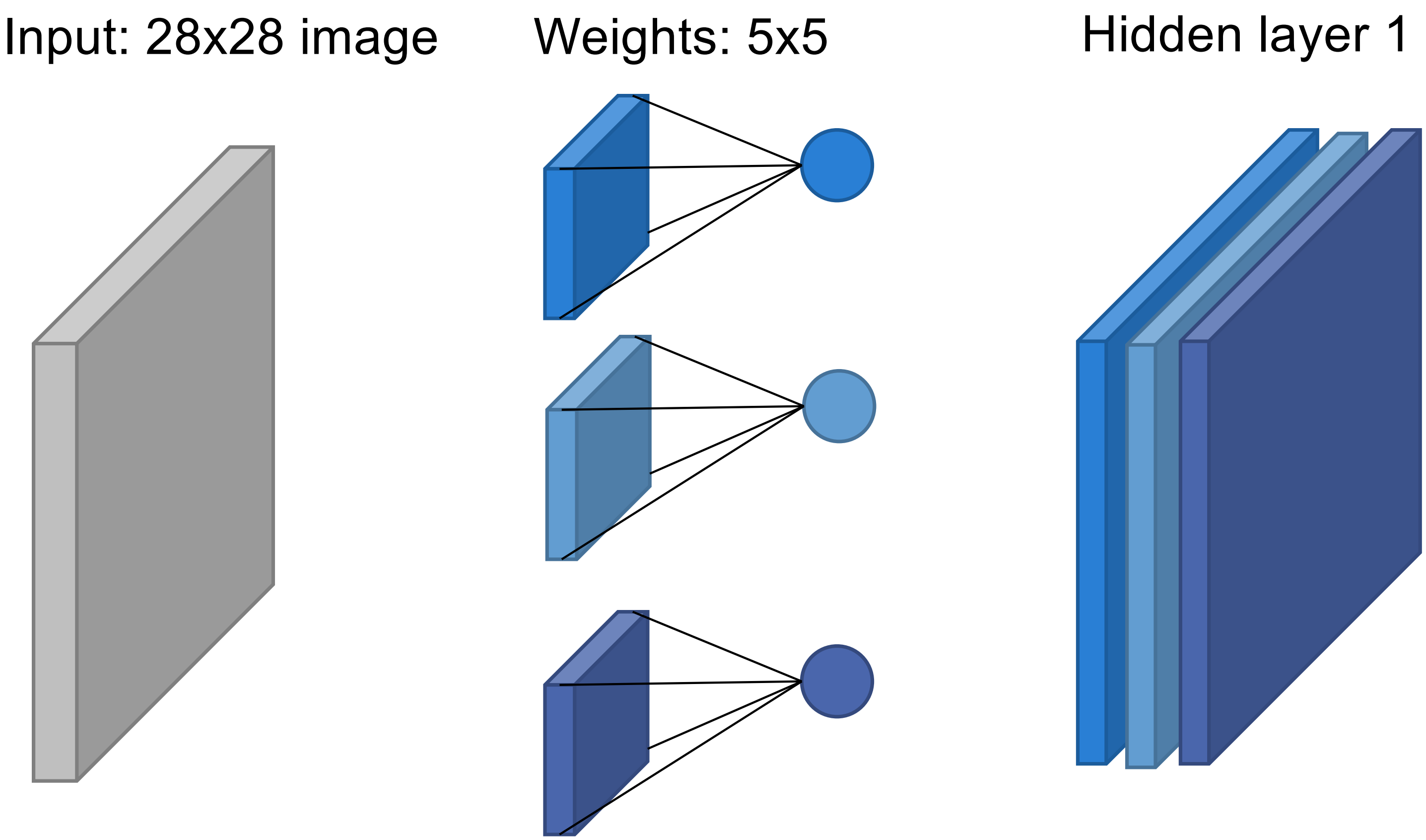
Input: 28x28 image   Weights: 5x5

Many hidden nodes, but few parameters!



Convolutional networks

Organize & share the NN weights (vs “dense”), Group into “filters”

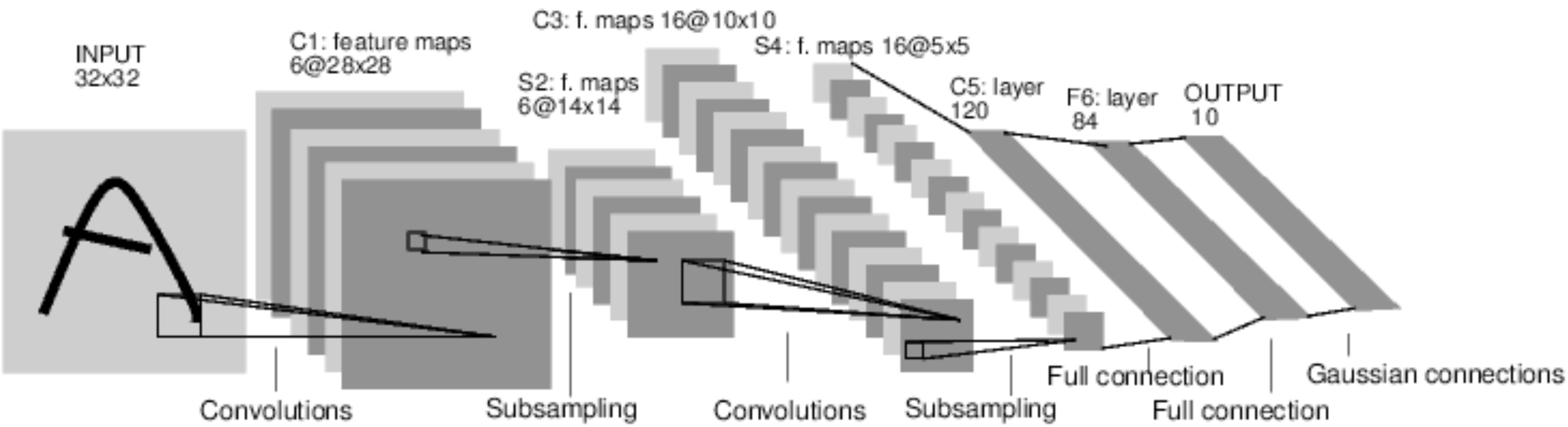


Convolutional networks

Again, can view components as building blocks

Design overall, deep structure from parts

- Convolutional layers
- “Max-pooling” (sub-sampling) layers
- Densely connected layers



LeNet-5 [LeCun 1980]

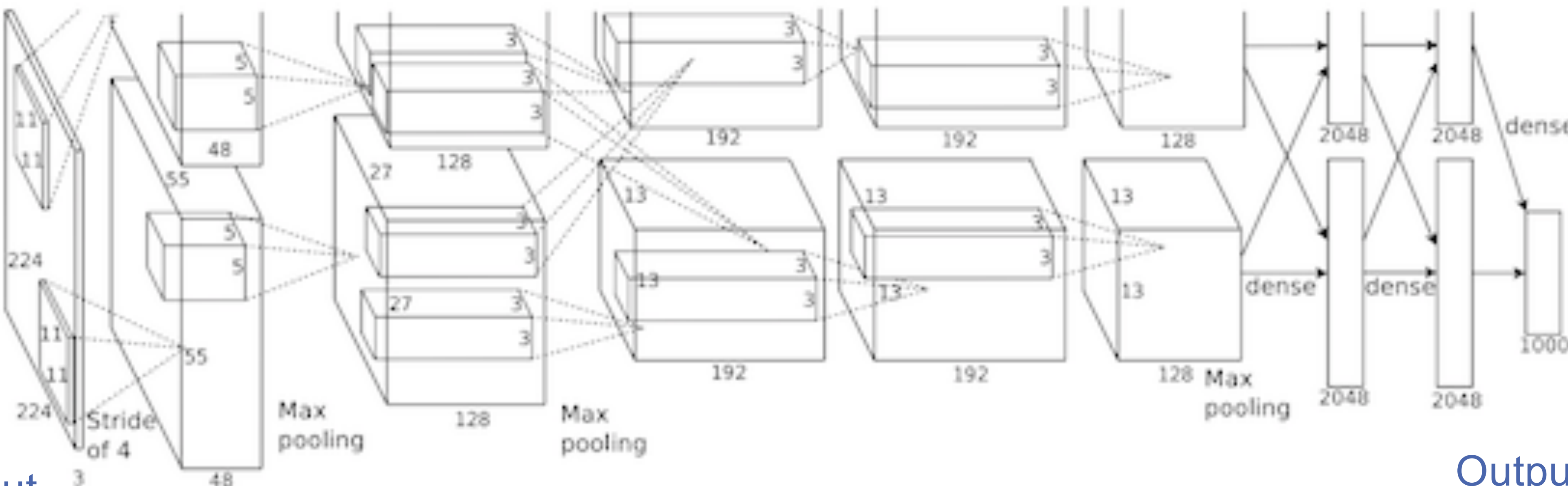
Ex: AlexNet

Deep NN model for ImageNet classification

- 650k units; 60m parameters
- 1m data; 1 week training (GPUs)

Convolutional Layers (5)

Dense Layers (3)

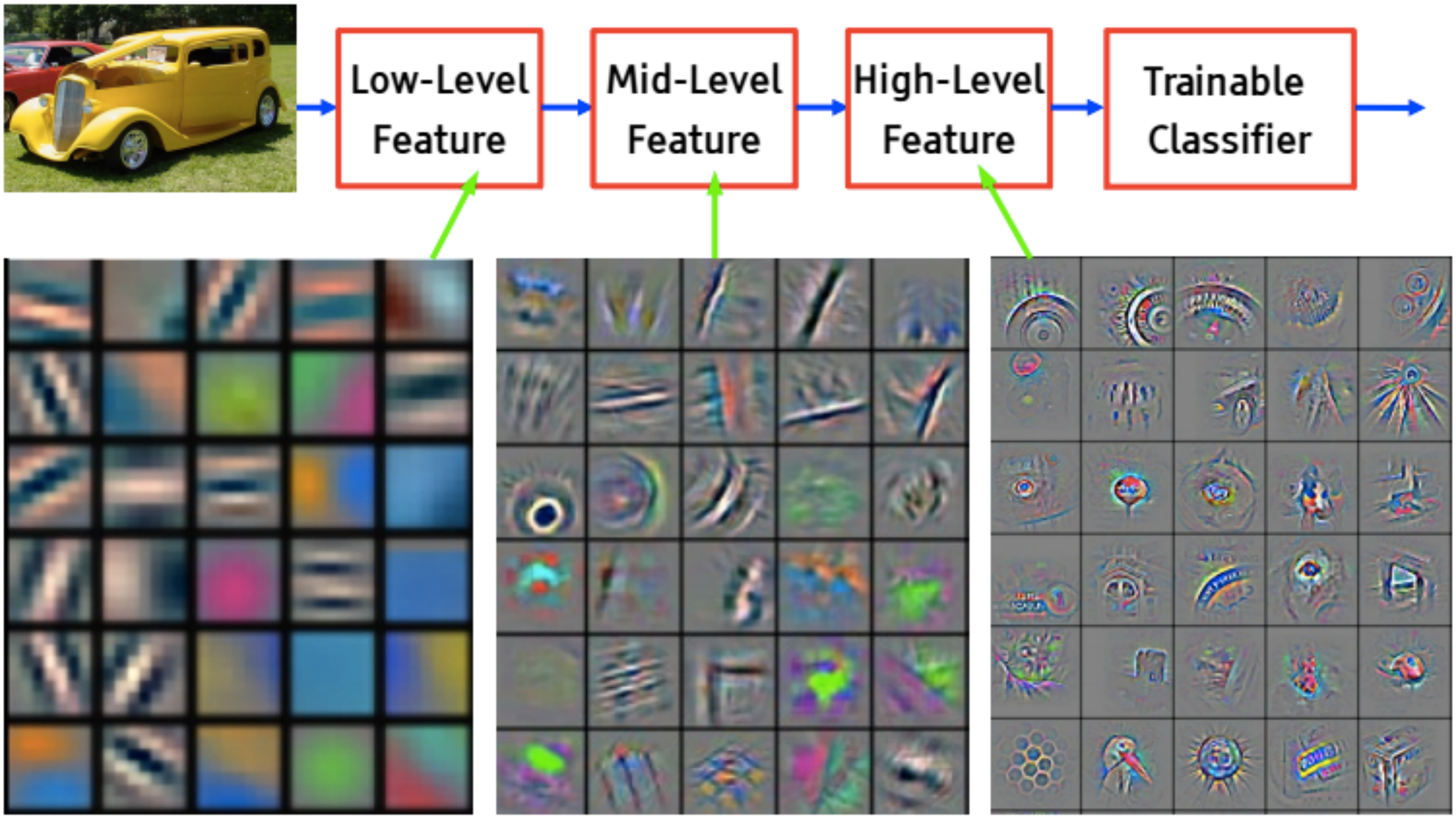


Input
224x224x3

Output
(1000 classes)

[Krizhevsky et al. 2012]

Hidden layers as “features”



Slide image from Yann LeCun:
https://drive.google.com/open?id=0BxKBnD5y2M8NclFWSXNxa0JlZTg

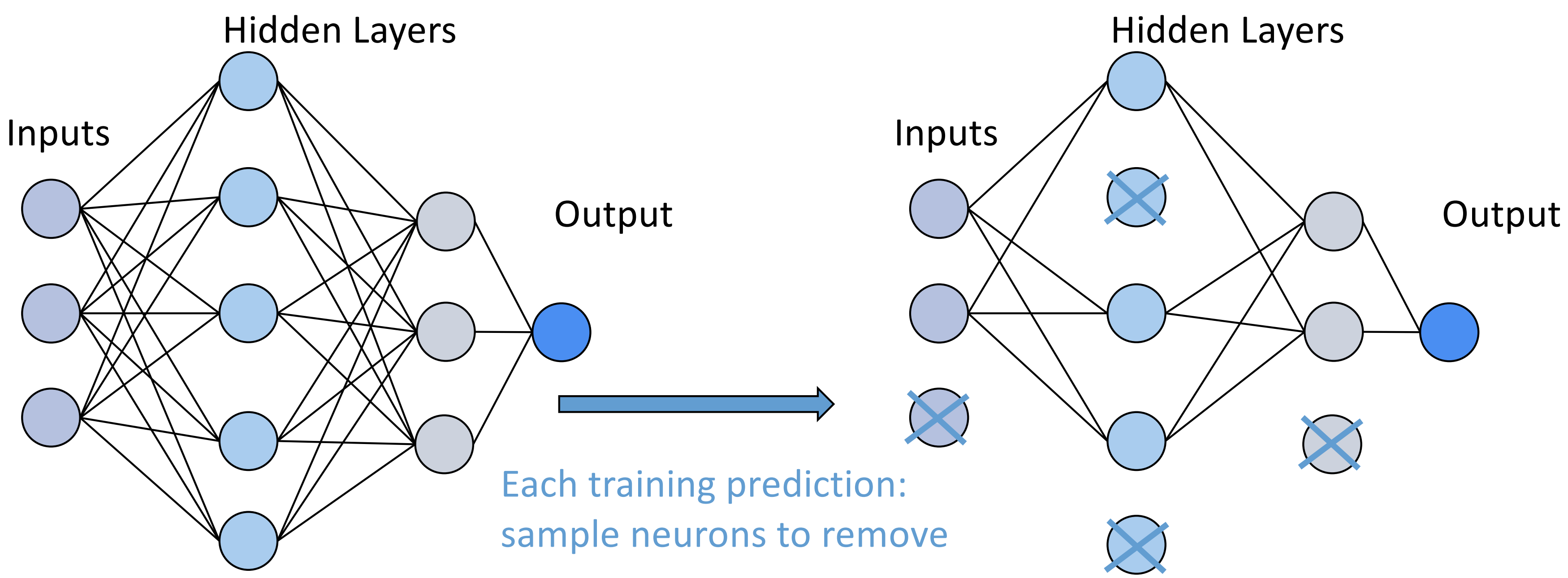
[Zeiler & Fergus 2013]

Dropout

```
% ... during training ...
R = X.dot(W[0])+B[0];       # linear response
H1= Sig( R );               # activation f’n
H1 *= np.random.rand(*H1.shape)<p; #drop out!
% ...
```

Another recent technique

- Randomly “block” some neurons at each step
- Trains model to have redundancy (predictions must be robust to blocking)



Summary

Neural networks, multi-layer perceptrons

Cascade of simple perceptrons
- Each just a linear classifier
- Hidden units used to create new features

Together, general function approximators
- Enough hidden units (features) = any function
- Can create nonlinear classifiers
- Also used for function approximation, regression, …

Training via backprop
- Gradient descent; logistic; apply chain rule. Building block view.

Advanced: deep nets, conv nets, dropout, …