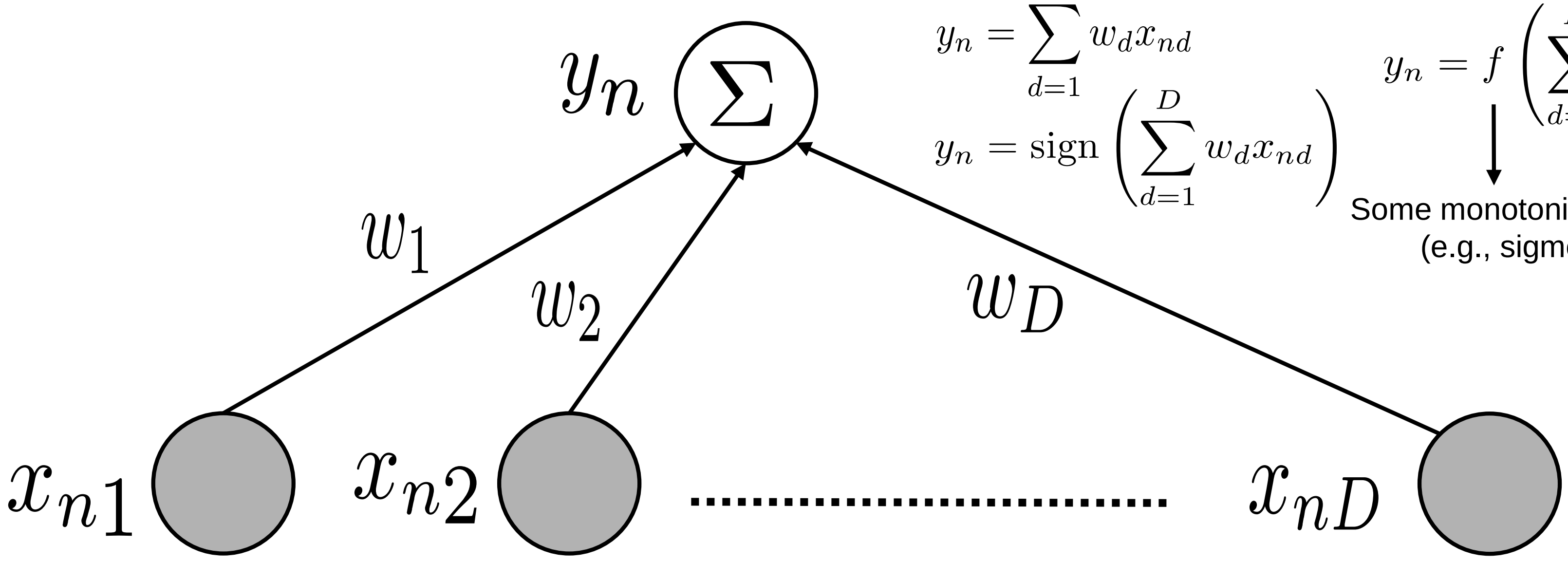
# Introduction to Deep Learning (1)

CS771: Introduction to Machine Learning

Piyush Rai

# Limitation of Linear Models

- Linear models: Output produced by taking a linear combination of input features



$$y_n = \sum_{d=1}^{D} w_d x_{nd}$$

$$y_n = \text{sign}\left(\sum_{d=1}^{D} w_d x_{nd}\right)$$

$$y_n = f\left(\sum_{d=1}^{D} w_d x_{nd}\right)$$

$f$: Some monotonic function (e.g., sigmoid)

Linear regression, logistic regression, SVM, etc

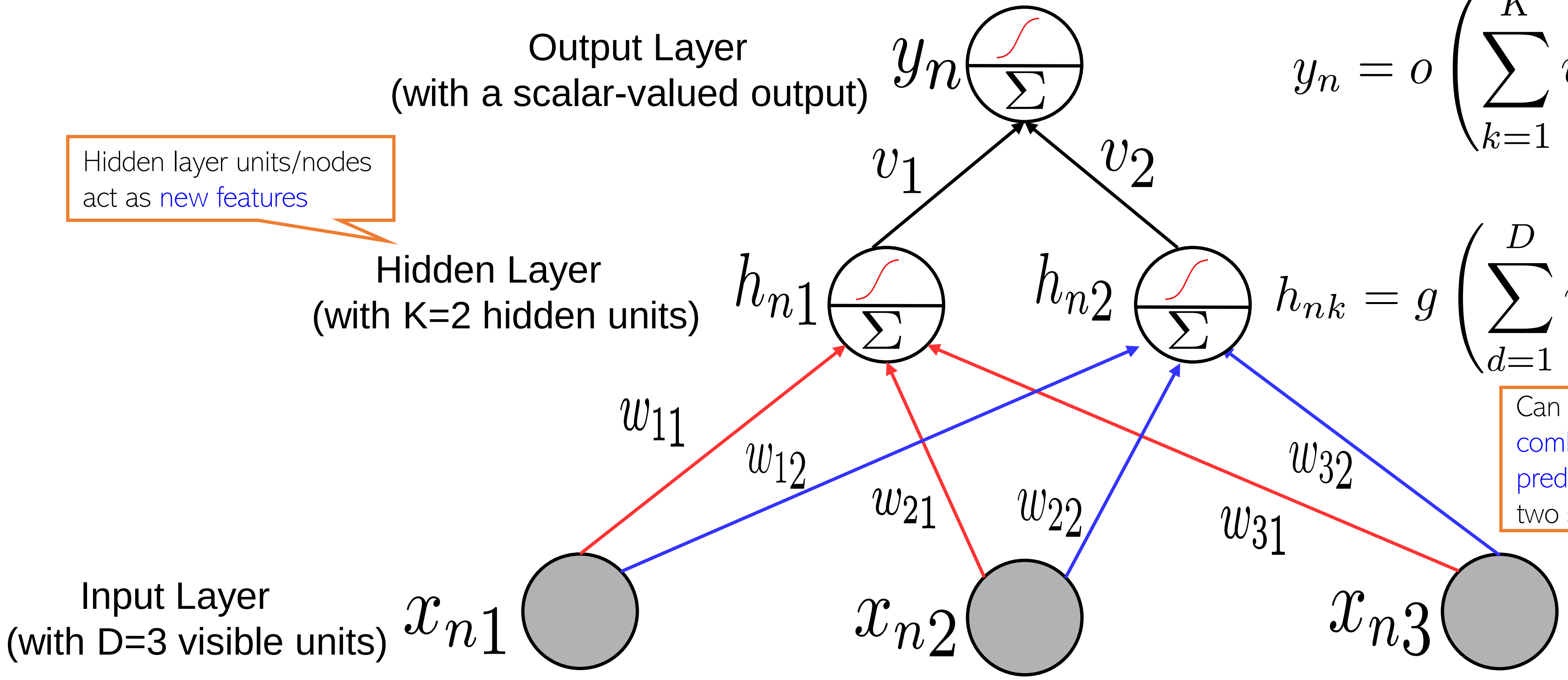
- This basic architecture is classically also known as the "Perceptron" (not to be confused with the Perceptron "algorithm", which learns a linear classification model)

Although can kernelize to make them nonlinear

- This can't however learn nonlinear functions or nonlinear decision boundaries

# Neural Networks: Multi-layer Perceptron (MLP)

- An MLP consists of an input layer, an output layer, and one or more hidden layers



Output Layer (with a scalar-valued output)
$y_n$
$v_1$
$v_2$
Hidden Layer (with K=2 hidden units)
$h_{n1}$
$h_{n2}$
$w_{11}$
$w_{12}$
$w_{21}$
$w_{22}$
$w_{32}$
$w_{31}$
Input Layer (with D=3 visible units)
$x_{n1}$
$x_{n2}$
$x_{n3}$

$$y_n = o\left(\sum_{k=1}^{K} v_k h_{nk}\right)$$

$$h_{nk} = g\left(\sum_{d=1}^{D} w_{dk} x_{nd}\right)$$

Learnable weights

Hidden layer units/nodes act as new features

Can think of this model as a combination of two predictions $h_{n1}$ and $h_{n2}$ of two simpler models

The effective $x$ to $y$ mapping is nonlinear (will see justification shortly)

# Illustration: Neural Net with One Hidden Layer

- Each input $\boldsymbol{x}_n$ transformed into several pre-activations using linear models

$$a_{nk} = \boldsymbol{w}_k^\top \boldsymbol{x}_n = \sum_{d=1}^{D} w_{dk} x_{nd}$$

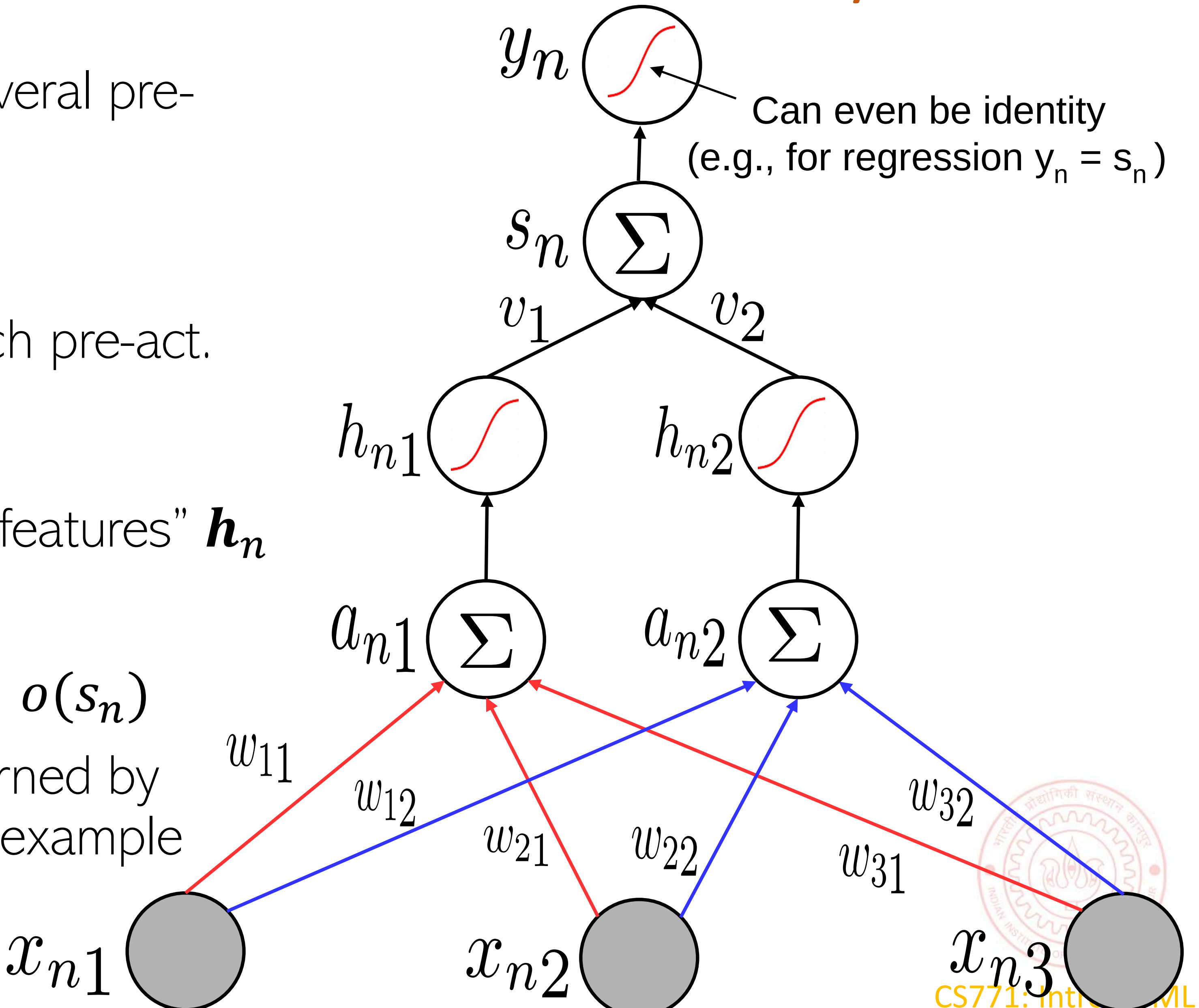
- Nonlinear activation applied on each pre-act.

$$h_{nk} = g(a_{nk})$$

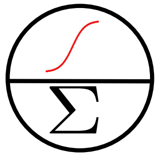
- Linear model learned on the new "features" $\boldsymbol{h}_n$

$$s_n = \boldsymbol{v}^\top \boldsymbol{h}_n = \sum_{k=1}^{K} v_k h_{nk}$$

- Finally, output is produced as $y = o(s_n)$
- Unknowns $(\boldsymbol{w}_1, \boldsymbol{w}_2, \dots, \boldsymbol{w}_K, \boldsymbol{v})$ learned by minimizing some loss function, for example $\mathcal{L}(\boldsymbol{W}, \boldsymbol{v}) = \sum_{n=1}^{N} \ell(y_n, o(s_n))$ (squared, logistic, softmax, etc)

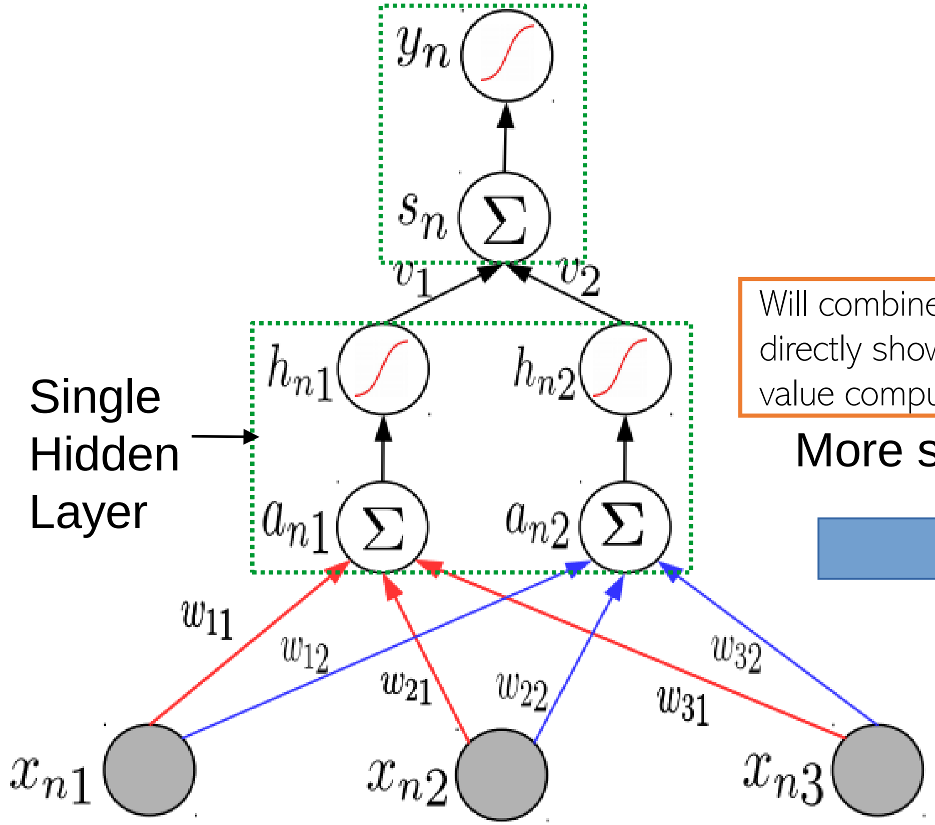


# Neural Nets: A Compact Illustration



Will denote a linear combination of inputs followed by a nonlinear operation on the result

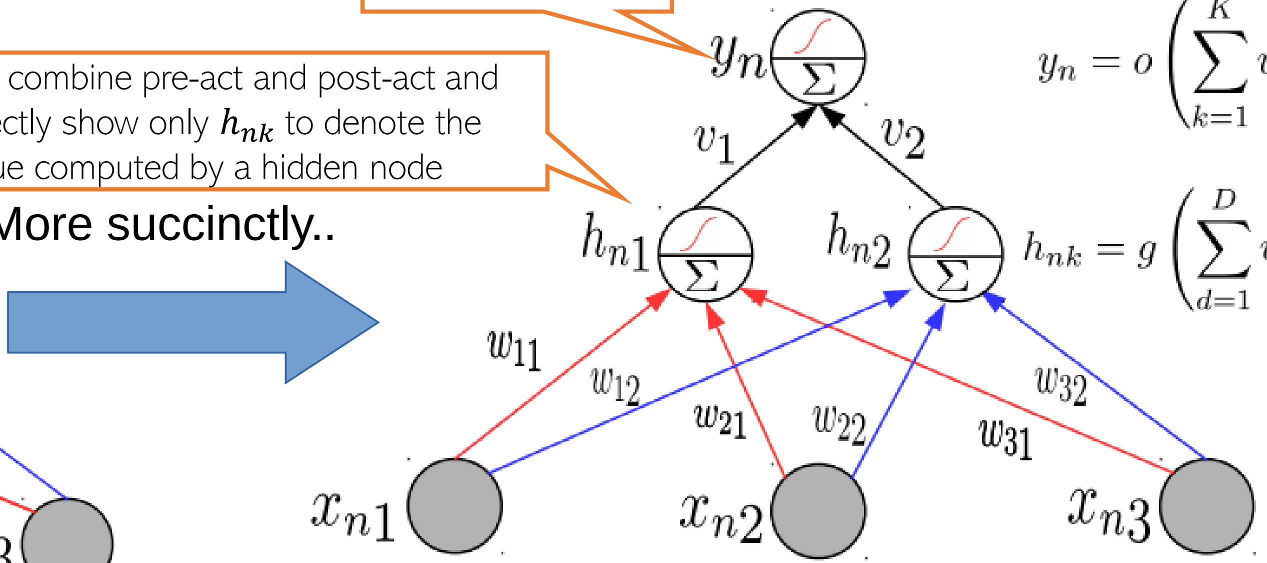
- Note: Hidden layer pre-act $a_{nk}$ and post-act $h_{nk}$ will be shown together for brevity



Will combine pre-act and post-act and directly show only $h_{nk}$ to denote the value computed by a hidden node

More succinctly..

Will directly show the final output

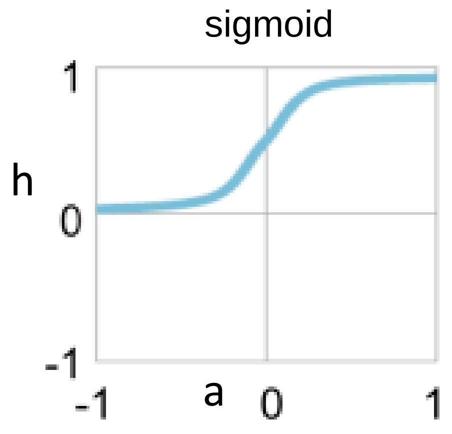


$$y_n = o\left(\sum_{k=1}^{K} v_k h_{nk}\right)$$

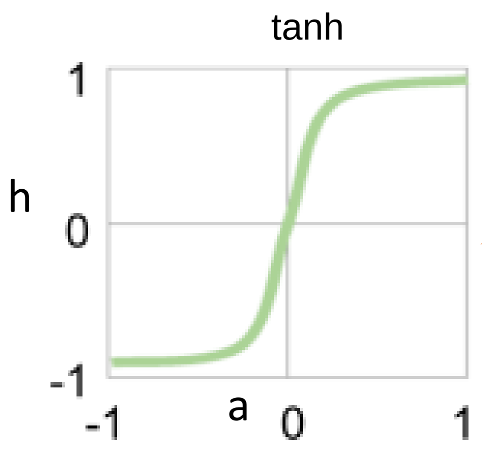
$$h_{nk} = g\left(\sum_{d=1}^{D} w_{dk} x_{nd}\right)$$

- Different layers may use different non-linear activations. Output layer may have none.

# Activation Functions: Some Common Choices



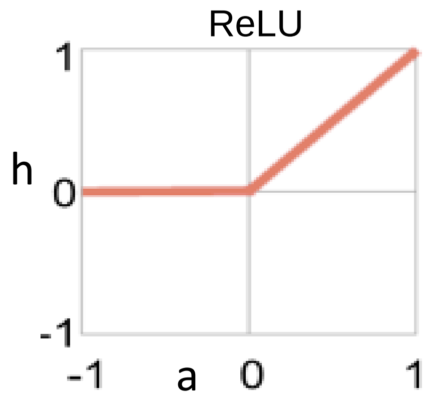
For sigmoid as well as tanh, gradients saturate (become close to zero as the function tends to its extreme values)



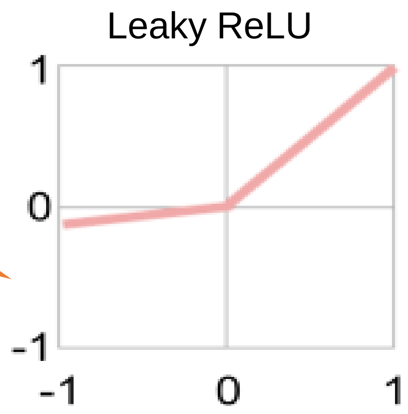
Preferred more than sigmoid. Helps keep the mean of the next layer's inputs close to zero (with sigmoid, it is close to 0.5)

Sigmoid: $h = \sigma(a) = \frac{1}{1+\exp(-a)}$

tanh (tan hyperbolic): $h = \frac{\exp(a)-\exp(-a)}{\exp(a)+\exp(-a)} = 2\sigma(2a) - 1$



Helps fix the dead neuron problem of ReLU when $a$ is a negative number



ReLU and Leaky ReLU are among the most popular ones

ReLU (Rectified Linear Unit): $h = \max(0, a)$

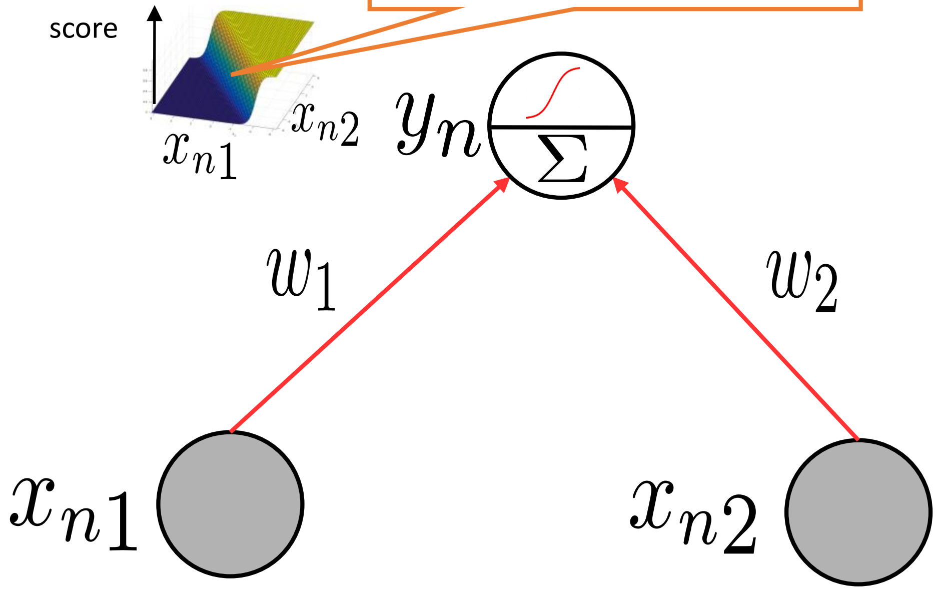
Leaky ReLU: $h = \max(\beta a, a)$
where $\beta$ is a small postive number

Without nonlinear activation, a deep neural network is equivalent to a linear model no matter how many layers we use

## MLP Can Learn Nonlin. Fn: A Brief Justification

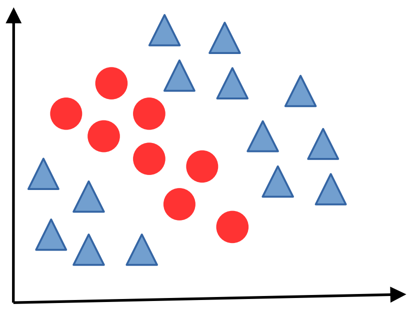
- An MLP can be seen as a composition of multiple linear models combined nonlinearly

Score monotonically increases. One-sided increase (not ideal for learning nonlinear decision boundaries)



Standard Single “Perceptron” Classifier (no hidden units)

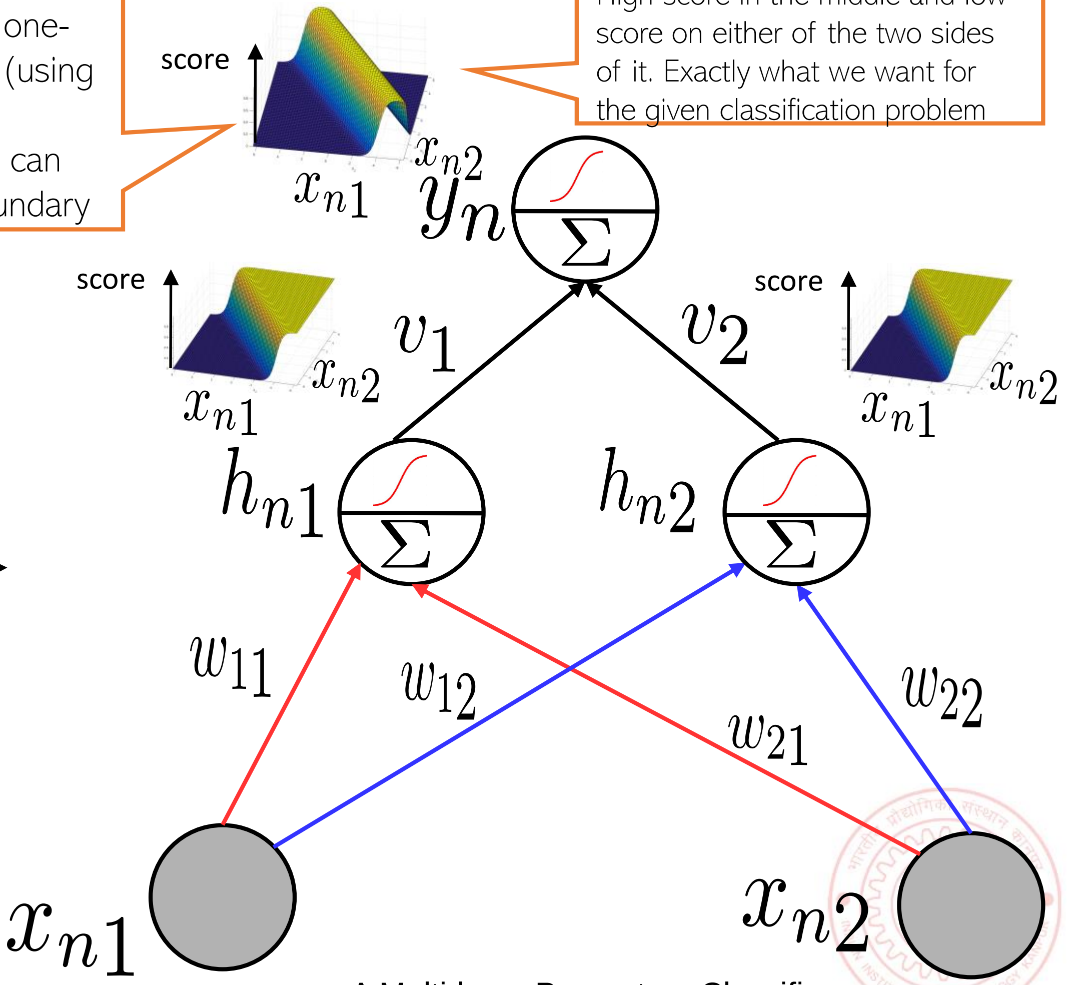
Obtained by composing the two one-sided increasing score functions (using $v_1$ = 1, and $v_2$ = -1 to “flip” the second one before adding). This can now learn nonlinear decision boundary



A nonlinear classification problem

A single hidden layer MLP with sufficiently large number of hidden units can approximate any function (Hornik, 1991)

High-score in the middle and low-score on either of the two sides of it. Exactly what we want for the given classification problem

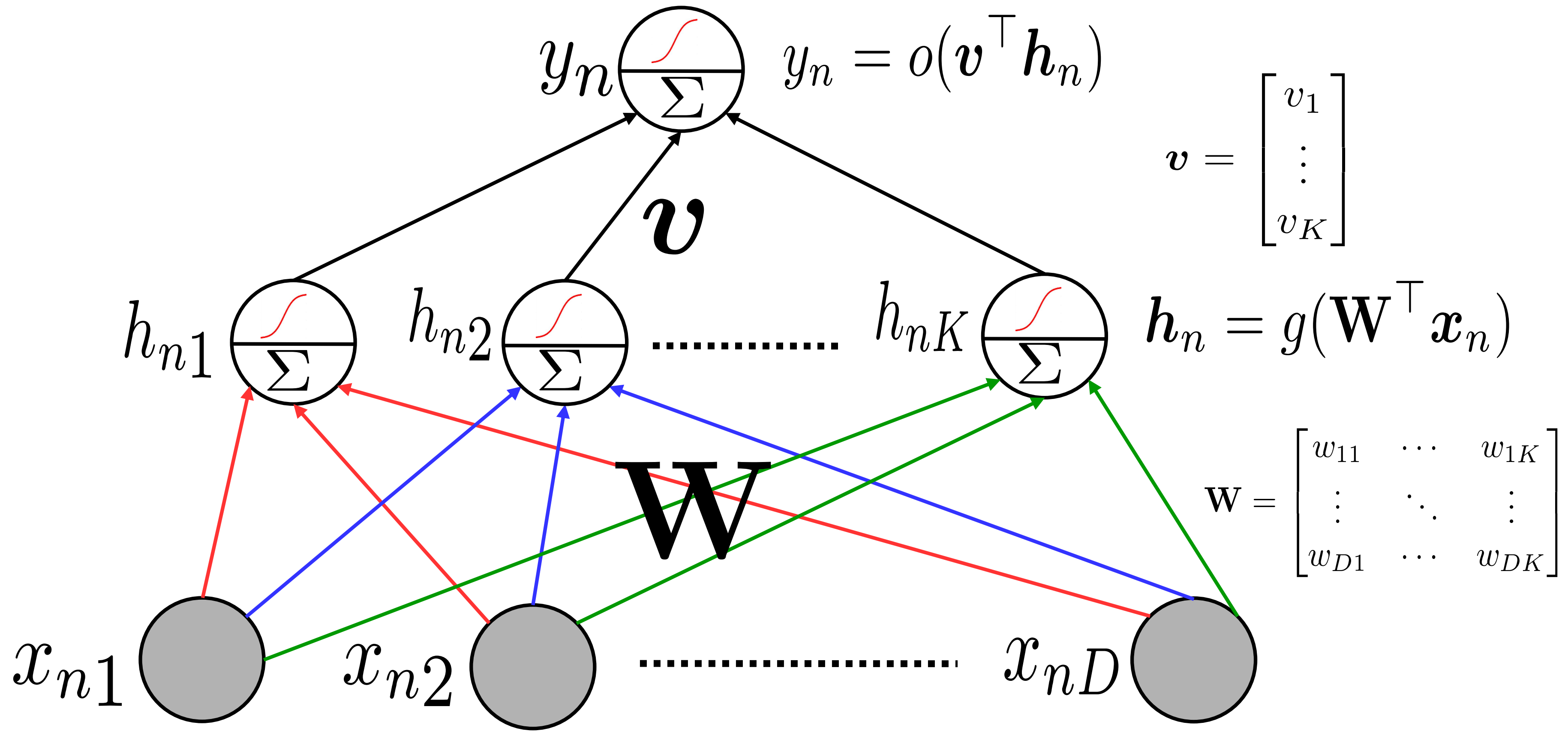


A Multi-layer Perceptron Classifier
(one hidden layer with 2 units)
Capable of learning nonlinear boundaries

## Examples of some basic NN/MLP architectures

## Single Hidden Layer and Single Outputs

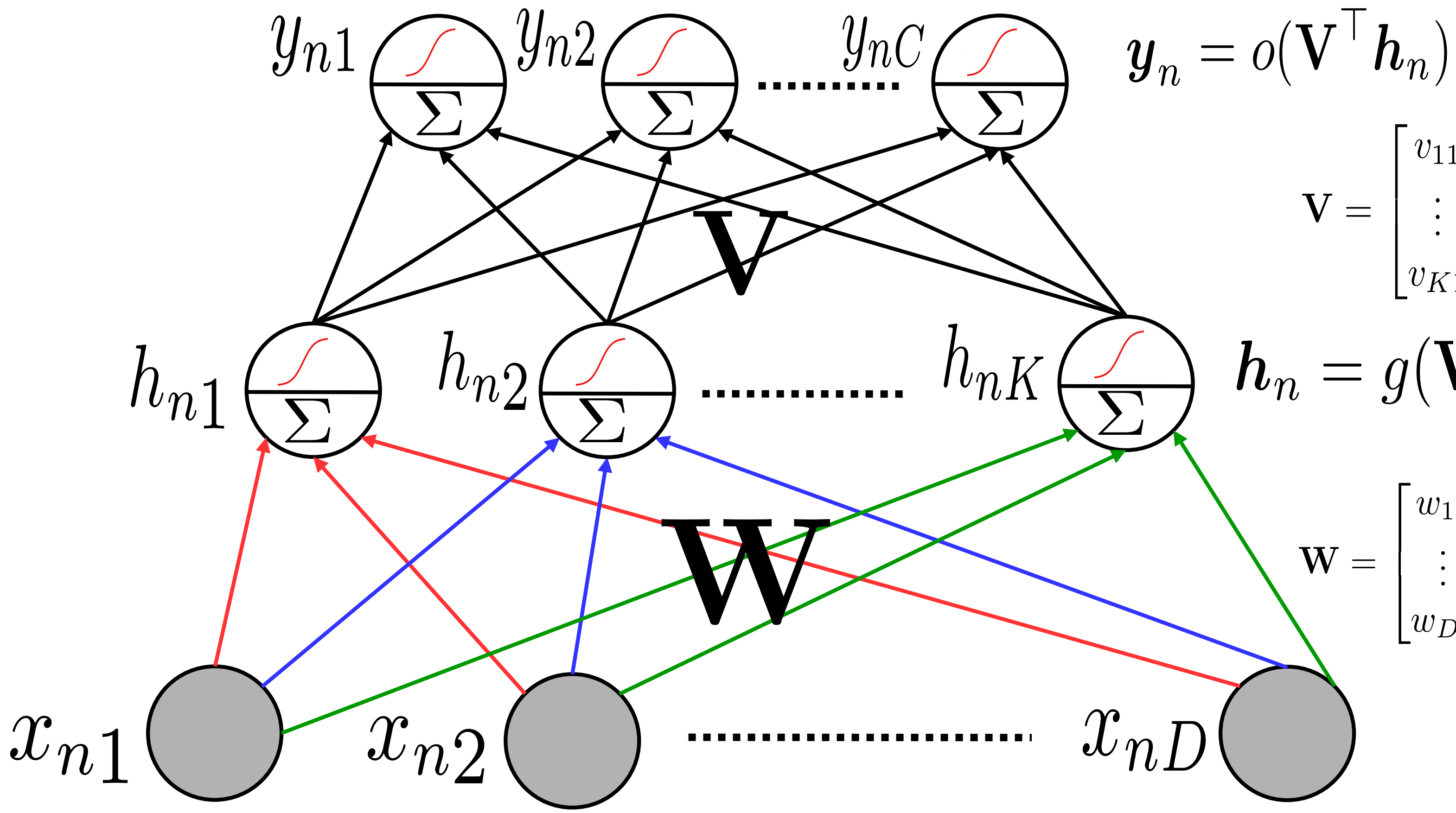
- One hidden layer with $K$ nodes and a single output (e.g., scalar-valued regression or binary classification)



$y_n$
$y_n = o(\boldsymbol{v}^\top \boldsymbol{h}_n)$
$\boldsymbol{v} = \begin{bmatrix} v_1 \\ \vdots \\ v_K \end{bmatrix}$
$\boldsymbol{v}$
$h_{n1}$
$h_{n2}$
$h_{nK}$
$\boldsymbol{h}_n = g(\mathbf{W}^\top \boldsymbol{x}_n)$
$\mathbf{W}$
$\mathbf{W} = \begin{bmatrix} w_{11} & \cdots & w_{1K} \\ \vdots & \ddots & \vdots \\ w_{D1} & \cdots & w_{DK} \end{bmatrix}$
$x_{n1}$
$x_{n2}$
$x_{nD}$

## Single Hidden Layer and Multiple Outputs

- One hidden layer with $K$ nodes and a vector of $C$ output (e.g., vector-valued regression or multi-class classification or multi-label classification)



$$\boldsymbol{y}_n = o(\mathbf{V}^\top \boldsymbol{h}_n)$$

$$\mathbf{V} = \begin{bmatrix} v_{11} & \cdots & v_{1C} \\ \vdots & \ddots & \vdots \\ v_{K1} & \cdots & v_{KC} \end{bmatrix}$$

$$\boldsymbol{h}_n = g(\mathbf{W}^\top \boldsymbol{x}_n)$$

$$\mathbf{W} = \begin{bmatrix} w_{11} & \cdots & w_{1K} \\ \vdots & \ddots & \vdots \\ w_{D1} & \cdots & w_{DK} \end{bmatrix}$$

## Multiple Hidden Layers (One/Multiple Outputs)

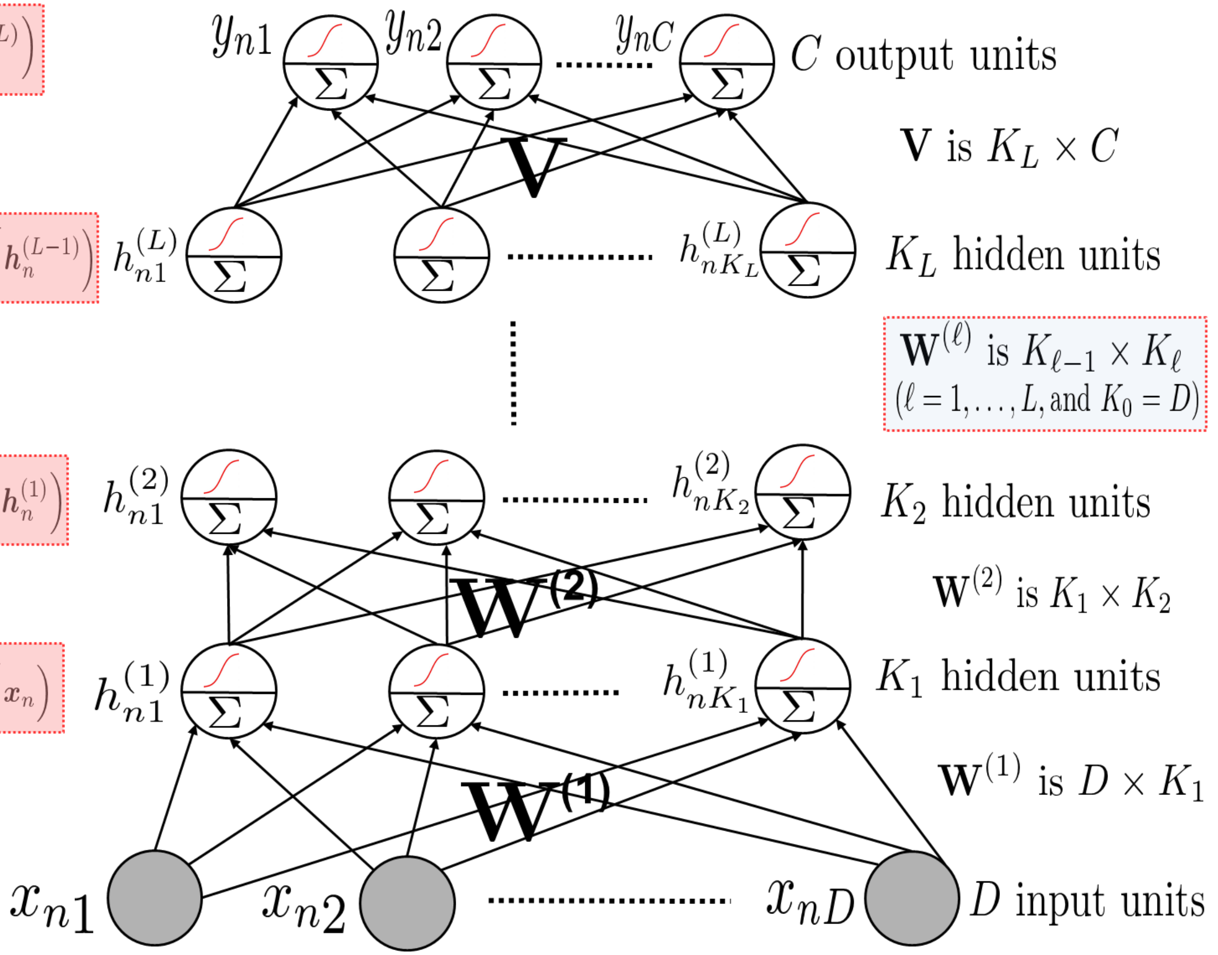
- Most general case: Multiple hidden layers with (with same or different number of hidden nodes in each) and a scalar or vector-valued output

$\boldsymbol{y}_n = o\left(\mathbf{V}^\top \boldsymbol{h}_n^{(L)}\right)$

$\boldsymbol{h}_n^{(L)} = g\left({\mathbf{W}^{(L)}}^\top \boldsymbol{h}_n^{(L-1)}\right)$

$\boldsymbol{h}_n^{(2)} = g\left({\mathbf{W}^{(2)}}^\top \boldsymbol{h}_n^{(1)}\right)$

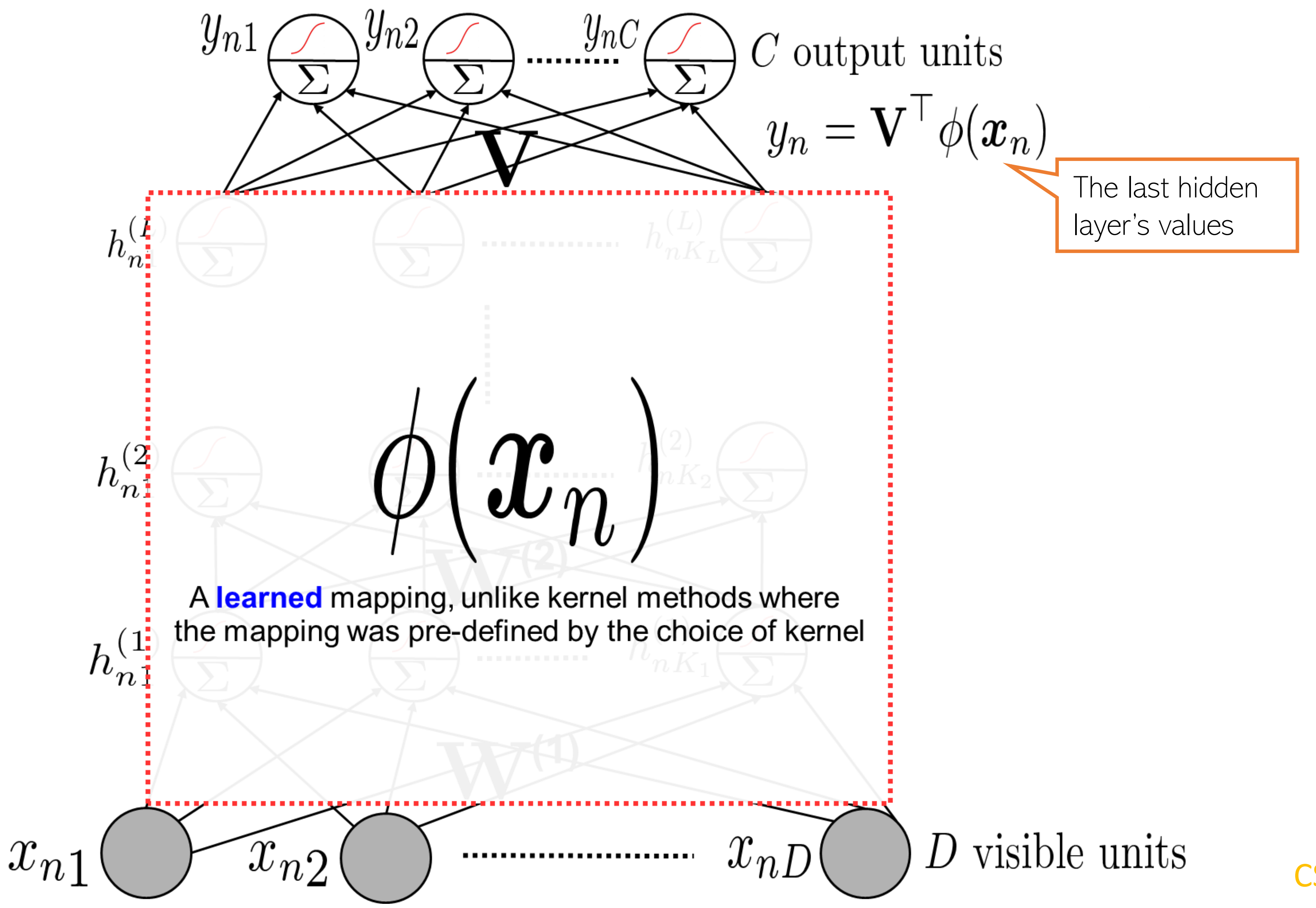
$\boldsymbol{h}_n^{(1)} = g\left({\mathbf{W}^{(1)}}^\top \boldsymbol{x}_n\right)$



$\mathbf{W}^{(\ell)}$ is $K_{\ell-1} \times K_\ell$
$(\ell = 1, \ldots, L$, and $K_0 = D)$

## Neural Nets are Feature Learners

- Hidden layers can be seen as learning a feature rep. $\boldsymbol{\phi}(\boldsymbol{x}_n)$ for each input $\boldsymbol{x}_n$



## Kernel Methods vs Neural Nets

- Recall the prediction rule for a kernel method (e.g., kernel SVM)

$$y = \sum_{n=1}^{N} \alpha_n k(\boldsymbol{x}_n, \boldsymbol{x})$$

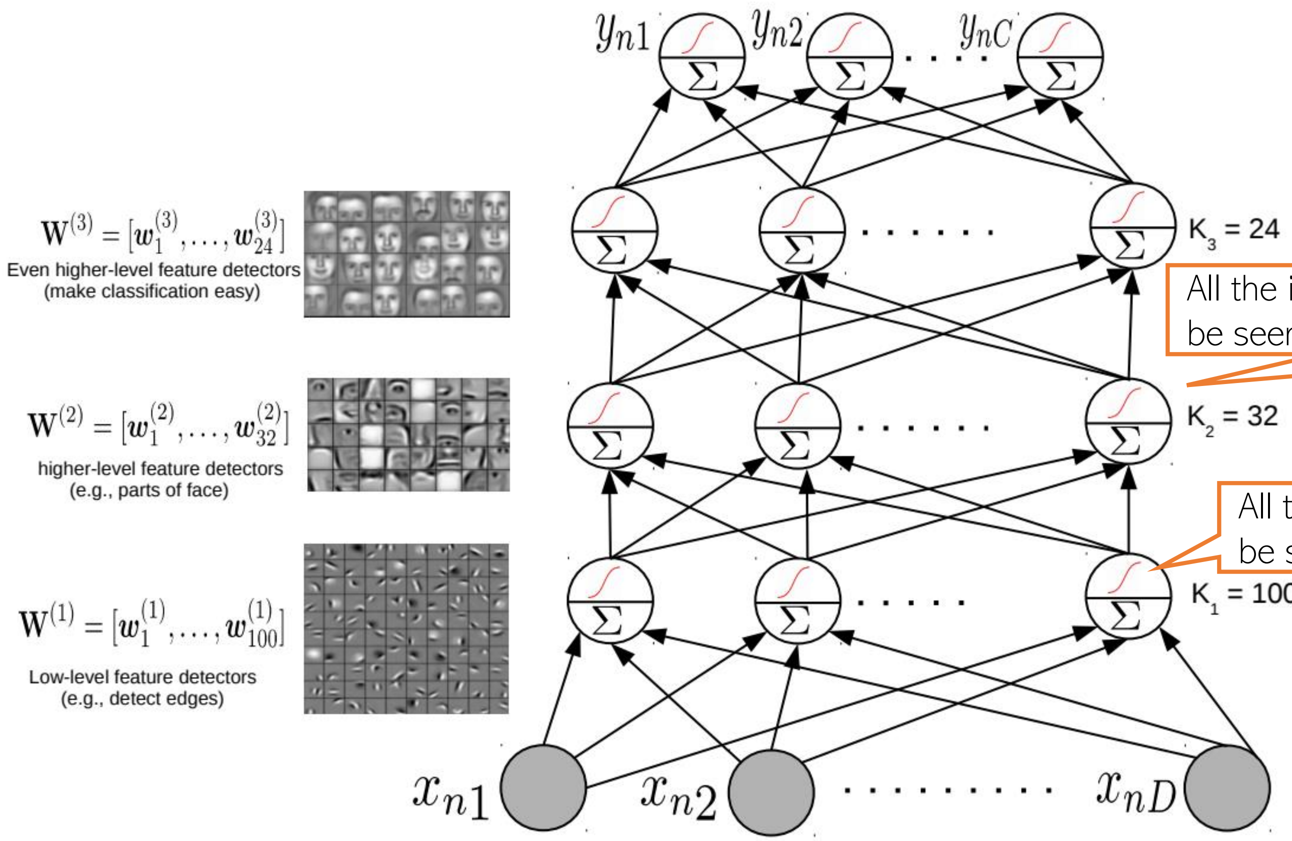
- This is analogous to a single hidden layer NN with fixed/pre-defined hidden nodes $\{k(\boldsymbol{x}_n, \boldsymbol{x})\}_{n=1}^{N}$ and output weights $\{\alpha_n\}_{n=1}^{N}$
- The prediction rule for a deep neural network

$$y = \sum_{k=1}^{K} v_k h_k$$

- Here, the $\boldsymbol{h}_k$'s are learned from data (possibly after multiple layers of nonlinear transformations)
- Both kernel methods and deep NNs be seen as using nonlinear basis functions for making predictions. Kernel methods use fixed basis functions (defined by the kernel) whereas NN learns the basis functions adaptively from data

# Feature Learned by a Neural Network

- Node values in each hidden layer tell us how much a "learned" feature is active in $\boldsymbol{x}_n$
- Hidden layer weights are like pattern/feature-detector/filter



$\mathbf{W}^{(3)} = [\boldsymbol{w}_1^{(3)}, \ldots, \boldsymbol{w}_{24}^{(3)}]$
Even higher-level feature detectors (make classification easy)
$\mathbf{W}^{(2)} = [\boldsymbol{w}_1^{(2)}, \ldots, \boldsymbol{w}_{32}^{(2)}]$
higher-level feature detectors (e.g., parts of face)
$\mathbf{W}^{(1)} = [\boldsymbol{w}_1^{(1)}, \ldots, \boldsymbol{w}_{100}^{(1)}]$
Low-level feature detectors (e.g., detect edges)
$y_{n1}$
$y_{n2}$
$y_{nC}$
$K_3 = 24$
$K_2 = 32$
$K_1 = 100$
$x_{n1}$
$x_{n2}$
$x_{nD}$

All the incoming weights (a vector) on this hidden node can be seen as representing a pattern/feature-detector/filter

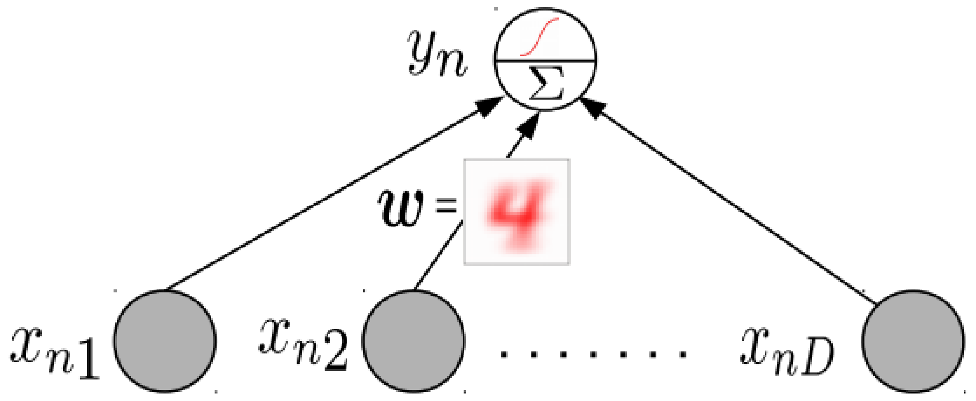
Here, $\boldsymbol{w}_{32}^{(2)}$ denotes a $K_1$-dim template/pattern/feature-detector

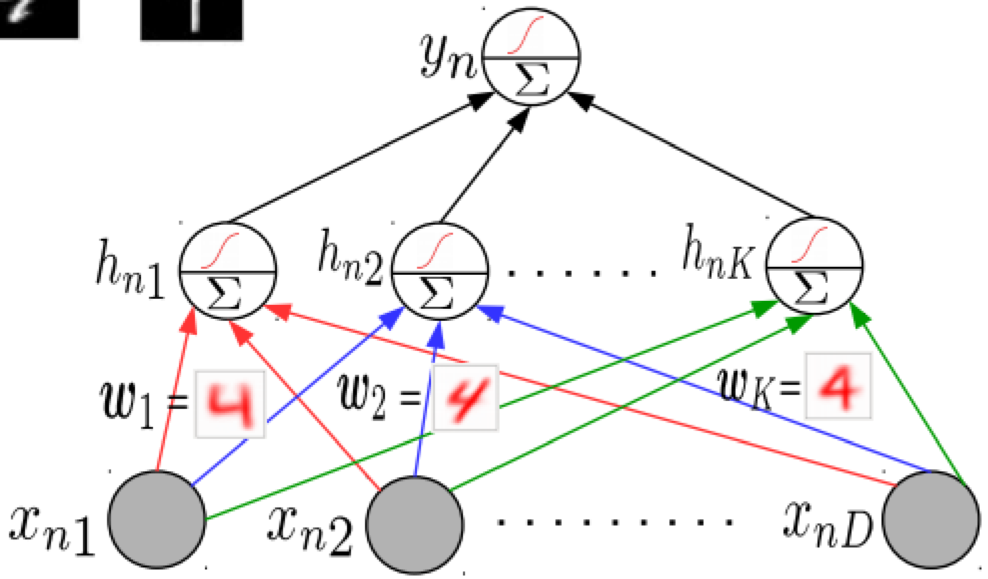
All the incoming weights (a vector) on this hidden node can be seen as representing a template/pattern/feature-detector

Here, $\boldsymbol{w}_{100}^{(1)}$ denotes a D-dim pattern/feature-detector/filter

## Why Neural Networks Work Better: Another View

- Linear models tend to only learn the "average" pattern
- Deep models can learn multiple patterns (each hidden node can learn one pattern)
  - Thus deep models can learn to capture more subtle variations that a simpler linear model





## Backpropagation

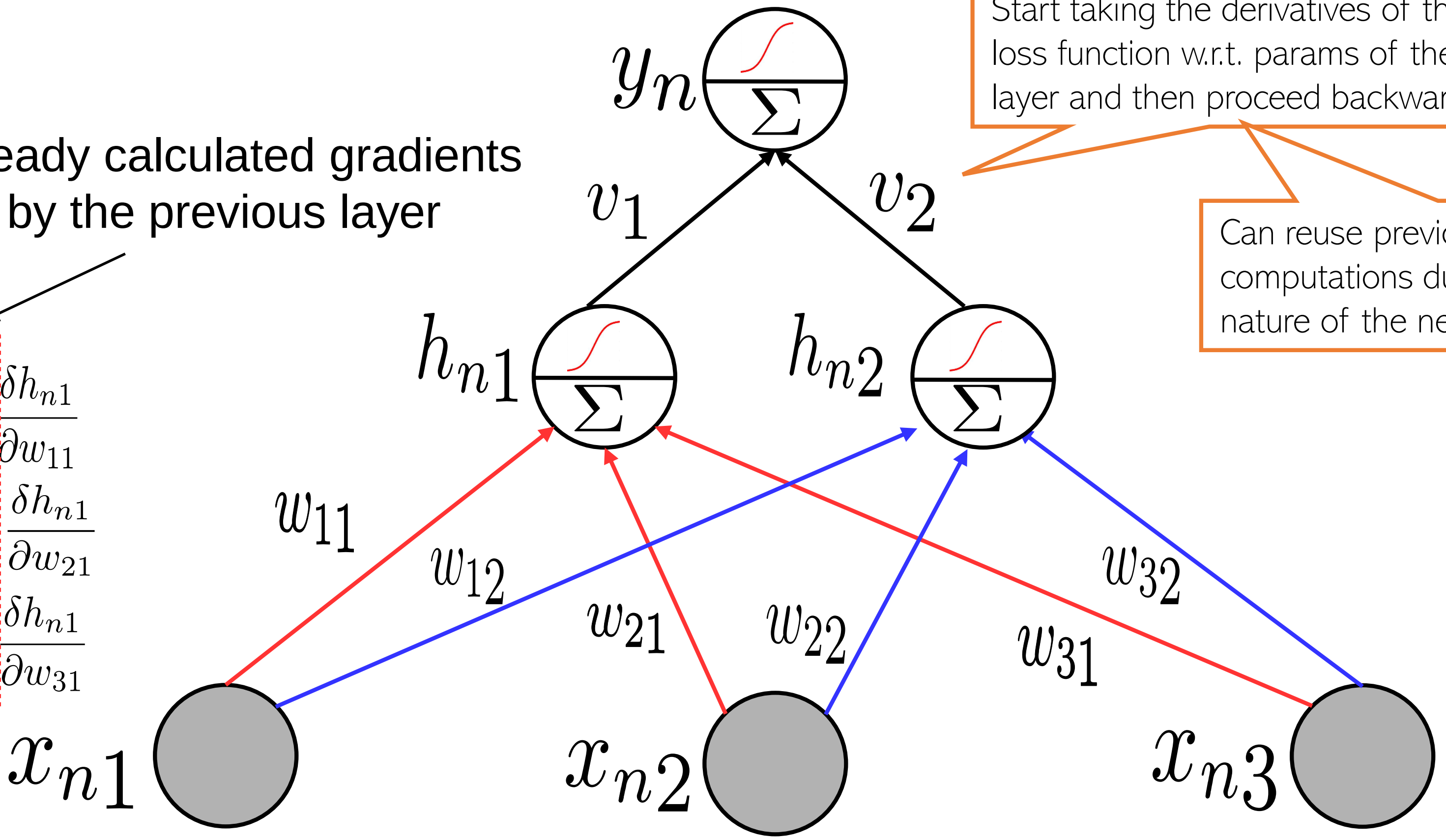
- Backpropagation = Gradient descent using chain rule of derivatives
- Chain rule of derivatives: Example, if $y = f_1(x)$ and $x = f_2(z)$ then $\frac{\partial y}{\partial z} = \frac{\partial y}{\partial x}\frac{\partial x}{\partial z}$

Reuse already calculated gradients computed by the previous layer

$$\frac{\delta \mathcal{L}}{\partial w_{11}} = \frac{\delta \mathcal{L}}{\partial h_{n1}} \frac{\delta h_{n1}}{\partial w_{11}}$$

$$\frac{\delta \mathcal{L}}{\partial w_{21}} = \frac{\delta \mathcal{L}}{\partial h_{n1}} \frac{\delta h_{n1}}{\partial w_{21}}$$

$$\frac{\delta \mathcal{L}}{\partial w_{31}} = \frac{\delta \mathcal{L}}{\partial h_{n1}} \frac{\delta h_{n1}}{\partial w_{31}}$$

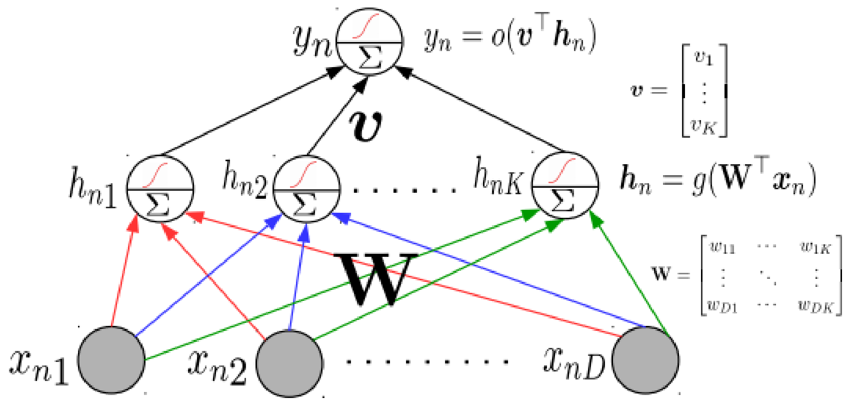


Start taking the derivatives of the loss function w.r.t. params of the last layer and then proceed backwards

Can reuse previous derivative computations due to the recursive nature of the neural net architecture

## Backpropagation through an example

Consider a single hidden layer MLP


$y_n$
$y_n = o(\boldsymbol{v}^\top \boldsymbol{h}_n)$
$\boldsymbol{v}$
$\boldsymbol{v} = \begin{bmatrix} v_1 \\ \vdots \\ v_K \end{bmatrix}$
$h_{n1}$
$h_{n2}$
$h_{nK}$
$\boldsymbol{h}_n = g(\mathbf{W}^\top \boldsymbol{x}_n)$
$\mathbf{W}$
$\mathbf{W} = \begin{bmatrix} w_{11} & \cdots & w_{1K} \\ \vdots & \ddots & \vdots \\ w_{D1} & \cdots & w_{DK} \end{bmatrix}$
$x_{n1}$
$x_{n2}$
$x_{nD}$

Assuming regression ($o$ = identity), the loss function for this model

$$\begin{aligned} \mathcal{L} &= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \boldsymbol{v}^\top \boldsymbol{h}_n\right)^2 \\ &= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k h_{nk}\right)^2 \\ &= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)\right)^2 \end{aligned}$$

- To use gradient methods for **W**, ***v***, we need gradients.
- Gradient of $\mathcal{L}$ w.r.t. ***v*** is straightforward

$$\frac{\partial \mathcal{L}}{\partial v_k} = -\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)\right) h_{nk} = \sum_{n=1}^{N} \boldsymbol{e}_n h_{nk}$$

- Gradient of $\mathcal{L}$ w.r.t. **W** requires chain rule

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial w_{dk}} &= \sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial h_{nk}} \frac{\partial h_{nk}}{\partial w_{dk}} \\ \frac{\partial \mathcal{L}}{\partial h_{nk}} &= -(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n))v_k = -\boldsymbol{e}_n v_k \\ \frac{\partial h_{nk}}{\partial w_{dk}} &= g'(\boldsymbol{w}_k^\top \boldsymbol{x}_n)x_{nd} \qquad (\text{note: } h_{nk} = g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)) \end{aligned}$$

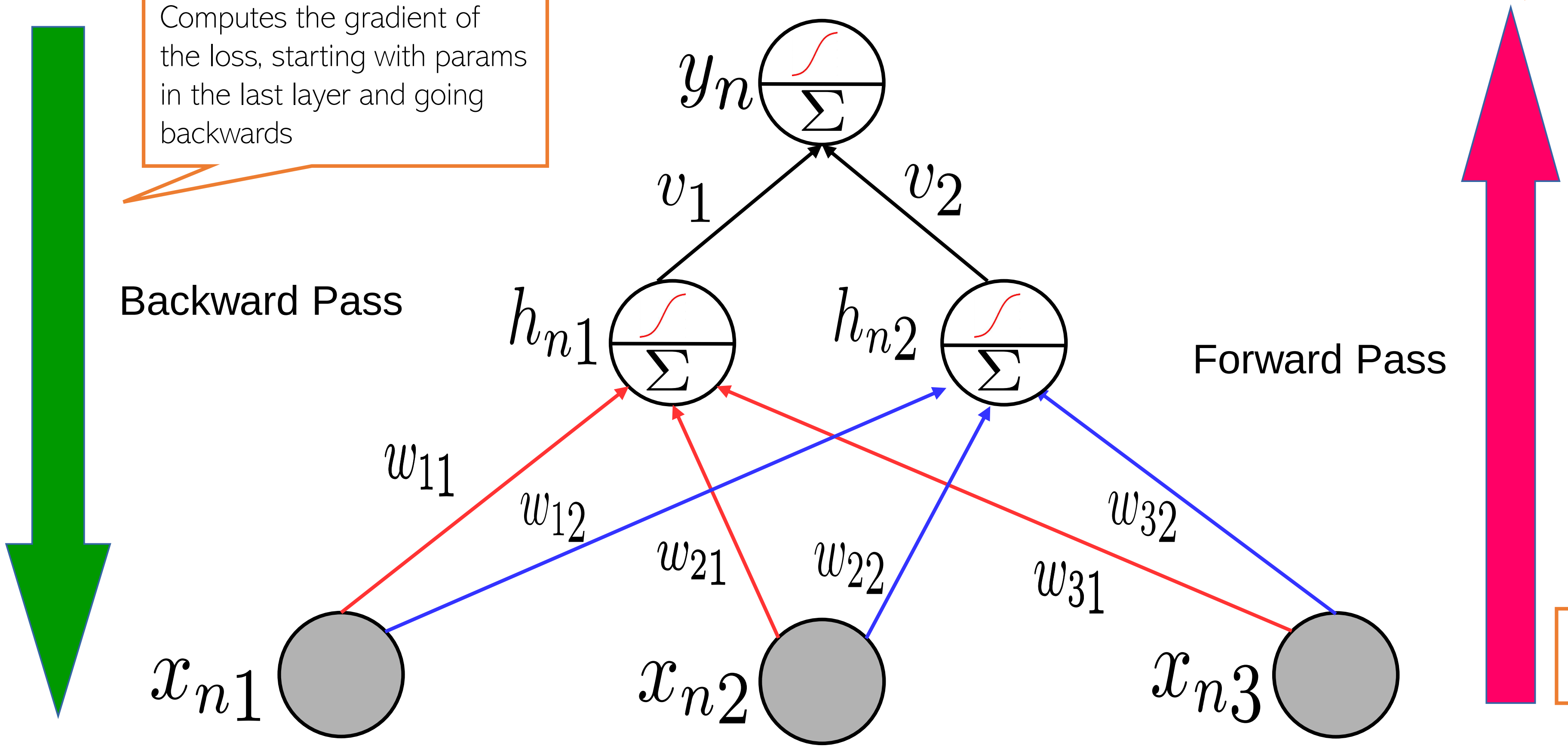
- Forward prop computes errors $\boldsymbol{e}_n$ using current **W**, ***v***. Backprop updates NN params **W**, ***v*** using grad methods
- Backprop caches many of the calculations for reuse

## Backpropagation

- Backprop iterates between a forward pass and a backward pass

Computes loss using current values of the parameters

Computes the gradient of the loss, starting with params in the last layer and going backwards



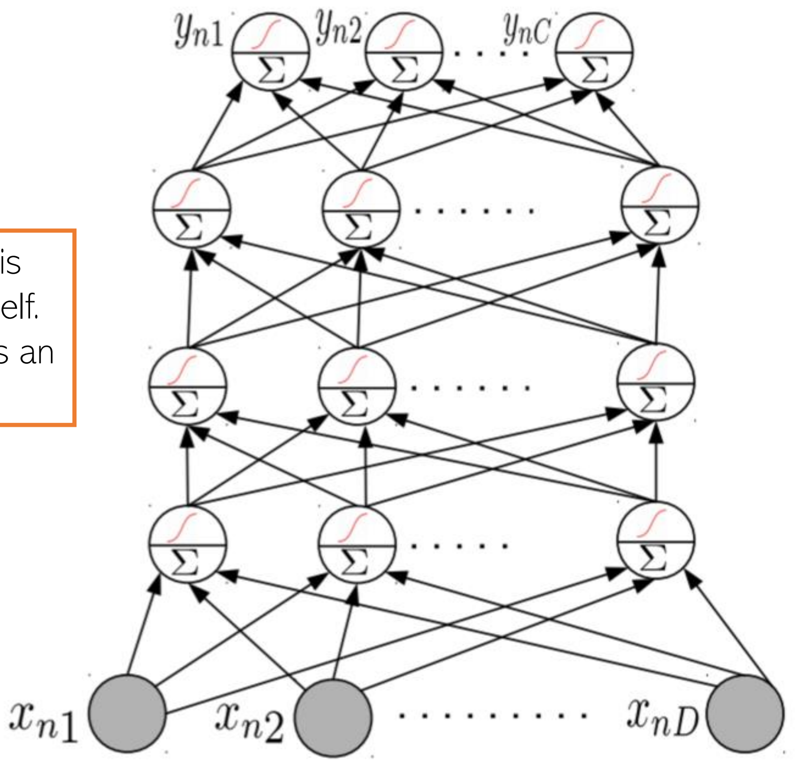
Using computational graphs

- Software frameworks such as Tensorflow and PyTorch support this already so you don't need to implement it by hand (so no worries of computing derivatives etc)

# Neural Nets: Some Aspects

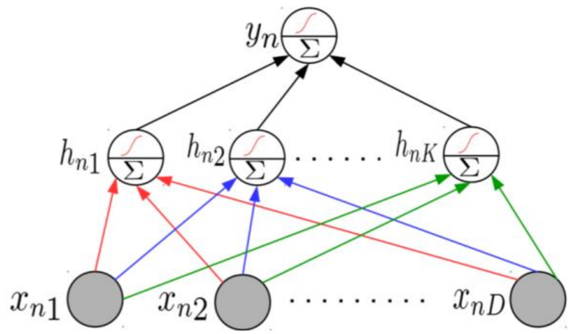
- Much of the magic lies in the hidden layers
- Hidden layers learn and detect good features
- Need to consider a few aspects
  - Number of hidden layers, number of units in each hidden layer
  - Why bother about many hidden layers and not use a single very wide hidden layer (recall Hornik's universal function approximator theorem)?
  - Complex networks (several, very wide hidden layers) or simpler networks (few, moderately wide hidden layers)?
  - Aren't deep neural network prone to overfitting (since they contain a huge number of parameters)?

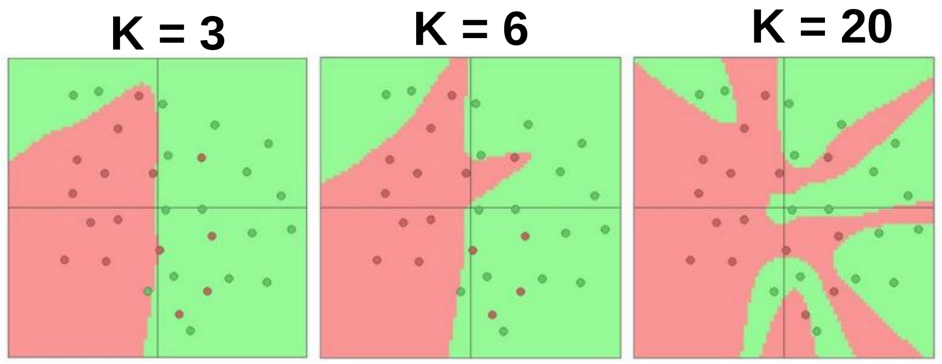
Choosing the right NN architecture is important and a research area in itself. Neural Architecture Search (NAS) is an automated technique to do this



# Representational Power of Neural Nets

- Consider a single hidden layer neural net with $K$ hidden nodes





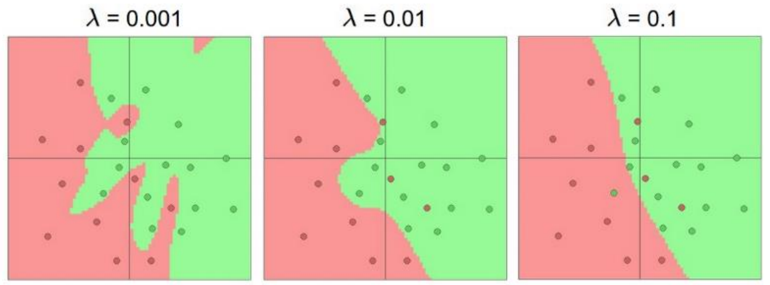
- Recall that each hidden unit "adds" a function to the overall function
- Increasing $K$ (number of hidden units) will result in a more complex function
- Very large $K$ seems to overfit (see above fig). Should we instead prefer small $K$?
- No! It is better to use large $K$ and regularize well. Reason/justification:
  - Simple NN with small $K$ will have a few local optima, some of which may be bad
  - Complex NN with large K will have many local optimal, all equally good (theoretical results on this)
- We can also use multiple hidden layers (each sufficiently large) and regularize well

# Preventing Overfitting in Neural Nets

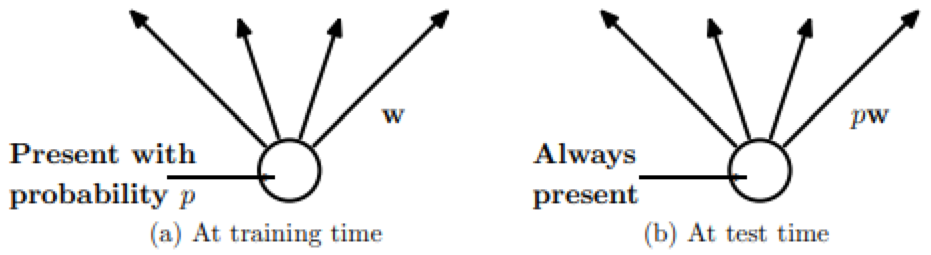
Various other tricks, such as weight sharing across different hidden units of the same layer (used in convolutional neural nets or CNN)

- Neural nets can overfit. Many ways to avoid overfitting, such as
  - Standard regularization on the weights, such as $\ell_2, \ell_1$, etc ($\ell_2$ reg. is also called weight decay)

Single Hidden Layer NN with K = 20 hidden units and L2 regularization



  - Early stopping (traditionally used): Stop when validation error starts increasing
  - Dropout: Randomly remove units (with some probability $p \in (0,1)$) during training



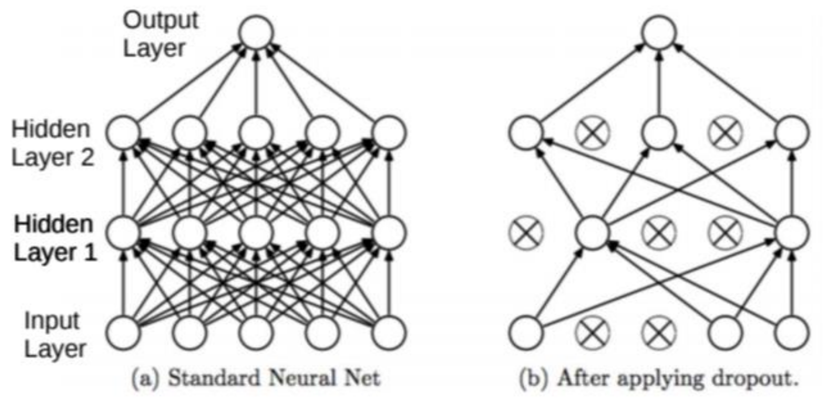
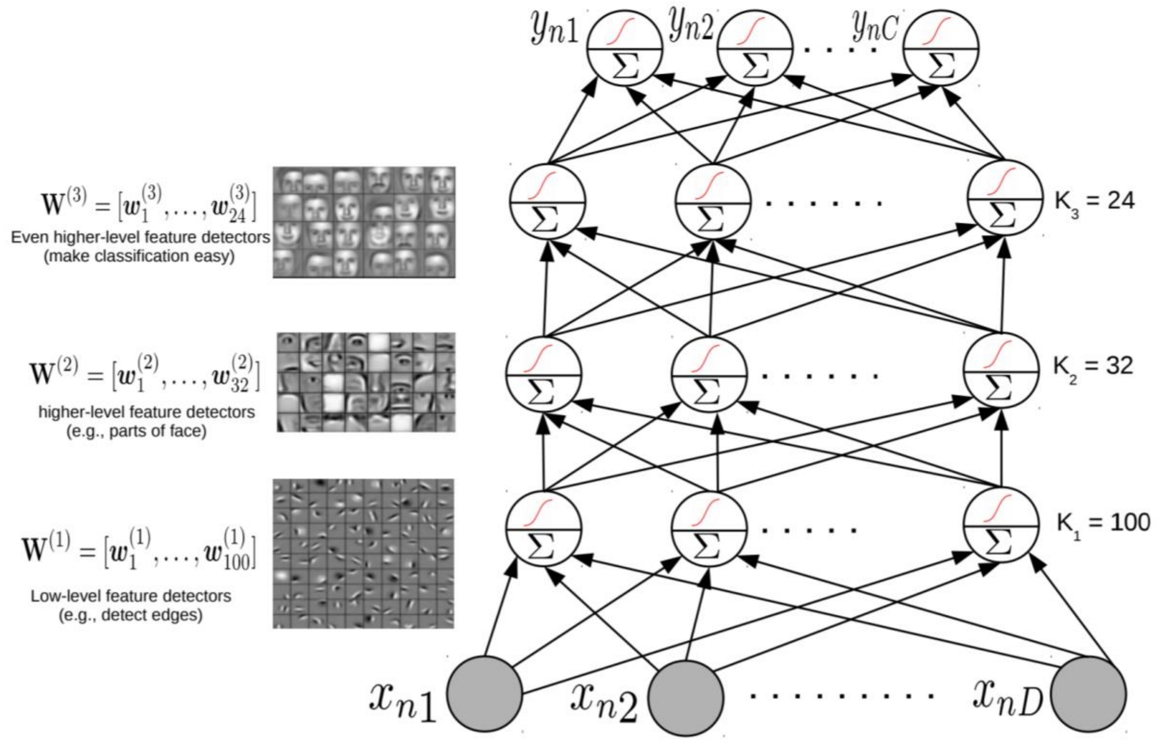


Fig courtesy: Dropout: A Simple Way to Prevent Neural Networks from Overfitting (Srivastava et al, 2014)

## Wide or Deep?

- While very wide single hidden layer can approx. any function, often we prefer many, less wide, hidden layers

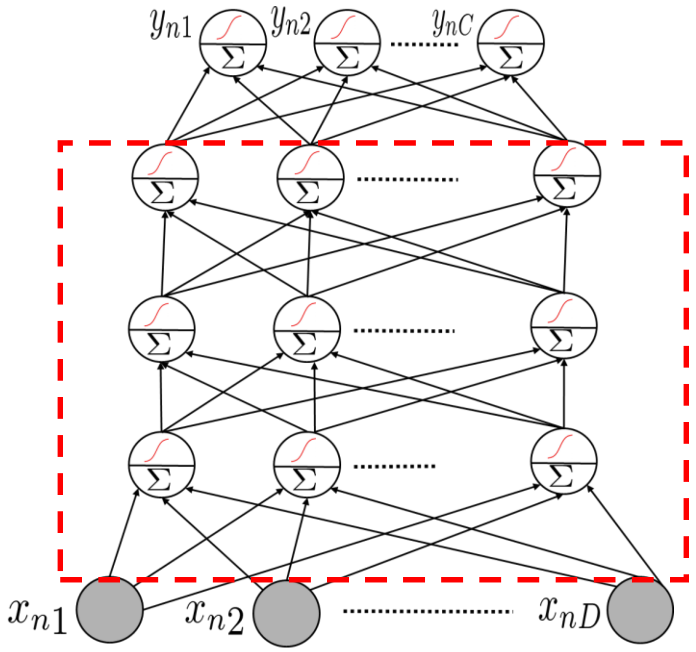


$y_{n1}$
$y_{n2}$
$y_{nC}$
$\mathbf{W}^{(3)} = [\mathbf{w}_1^{(3)}, \dots, \mathbf{w}_{24}^{(3)}]$
Even higher-level feature detectors (make classification easy)
$\mathbf{W}^{(2)} = [\mathbf{w}_1^{(2)}, \dots, \mathbf{w}_{32}^{(2)}]$
higher-level feature detectors (e.g., parts of face)
$\mathbf{W}^{(1)} = [\mathbf{w}_1^{(1)}, \dots, \mathbf{w}_{100}^{(1)}]$
Low-level feature detectors (e.g., detect edges)
$K_3 = 24$
$K_2 = 32$
$K_1 = 100$
$x_{n1}$
$x_{n2}$
$x_{nD}$

- Higher layers help learn more directly useful/interpretable features (also useful for compressing data using a small number of features)

# Using a Pre-trained Network

- A deep NN already trained in some "generic" data can be useful for other tasks, e.g.,
  - Feature extraction: Use a pre-trained net, remove the output layer, and use the rest of the network as a feature extractor for a related dataset



This part of a pre-trained net can be used as a feature extractor on some new task

Many packages, like Tensorflow and PyTorch provide such pre-trained module ready to be used

Sometimes also known as "transfer learning" in the context of neural nets

  - Fine-tuning: Use a pre-trained net, use its weights as initialization to train a deep net for a new but related task (useful when we don't have much training data for the new task)

## Deep Neural Nets: Some Comments

- Highly effective in learning good feature rep. from data in an "end-to-end" manner
- The objective functions of these models are highly non-convex
  - But fast and robust non-convex opt algos exist for learning such deep networks
- Training these models is computationally very expensive
  - But GPUs can help to speed up many of the computations
- Also useful for unsupervised learning problems (will see some examples)
  - Autoencoders for dimensionality reduction
  - Deep generative models for generating data and (unsupervisedly) learning features – examples include generative adversarial networks (GAN) and variational auto-encoders (VAE)

## Coming up next

- Convolutional neural nets
- Neural nets for sequential data
- Neural networks for unsupervised learning and generation