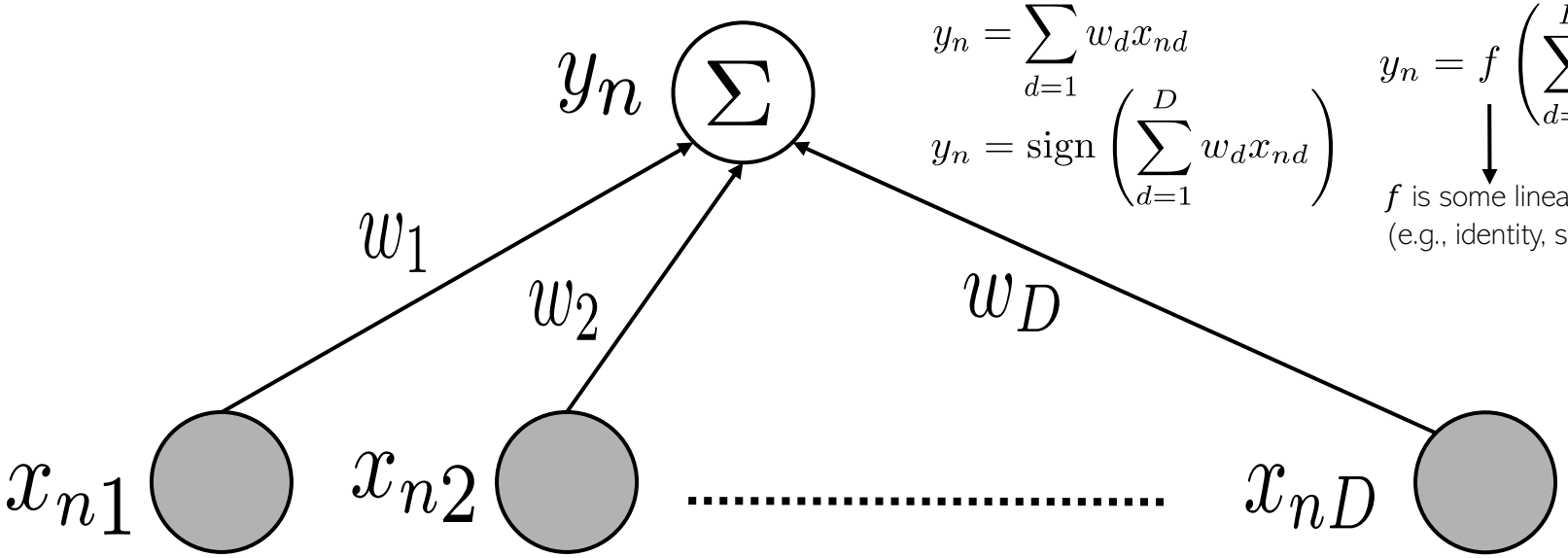
Intro to Deep Neural Networks: MLP and Backpropagation

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 is some linear/nonlinear function (e.g., identity, sign, or sigmoid)

Linear regression, logistic regression, SVM, etc

- A basic unit of the form $\boldsymbol{y = f(w^\top x)}$ is 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 is a network containing several Perceptron units across many layers
- An MLP consists of an input layer, an output layer, and one or more hidden layers

Output Layer
(with a scalar-valued output)

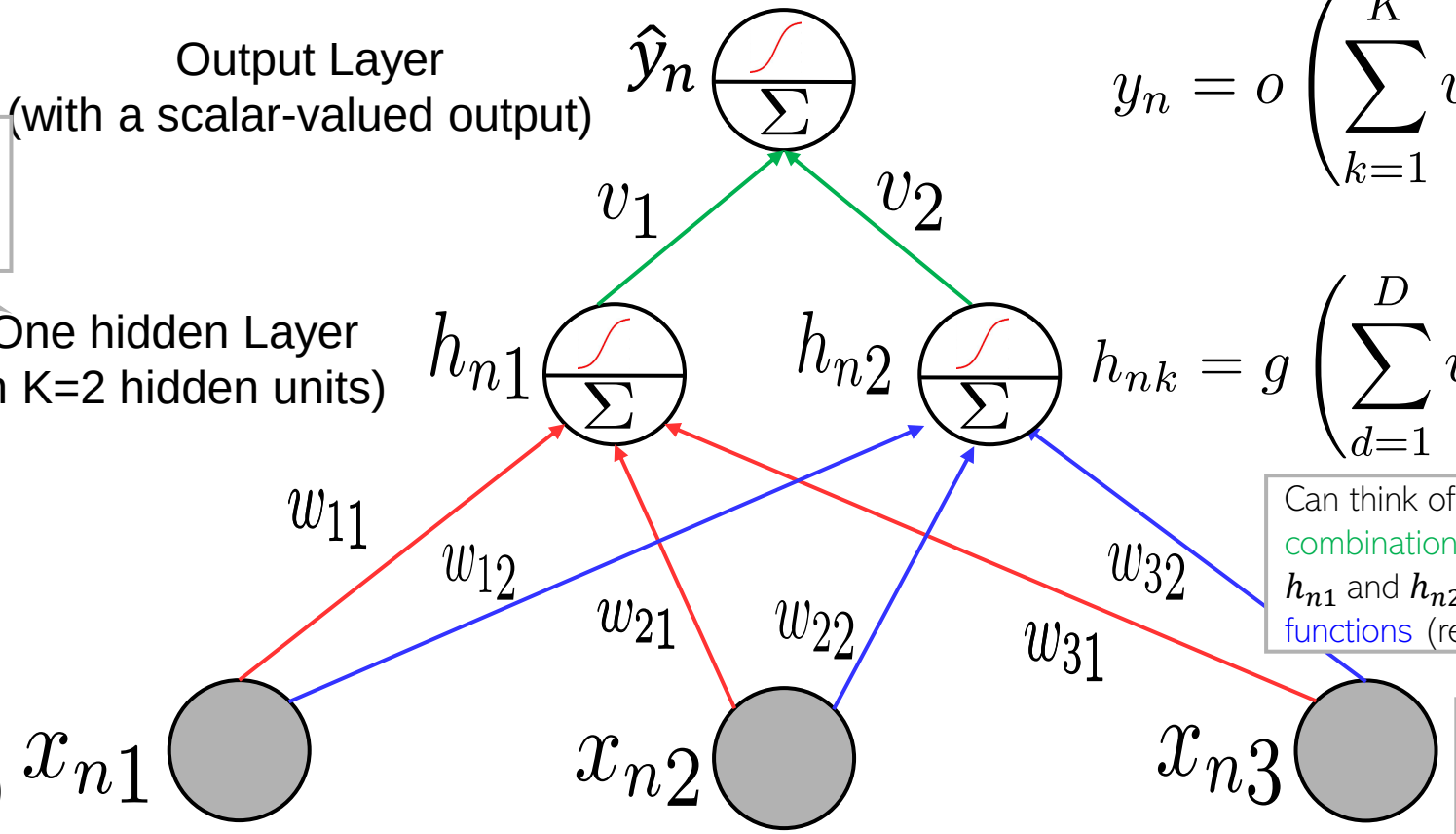
$$y_n = o\left(\sum_{k=1}^{K} v_k h_{nk}\right)$$

Learnable weights

Hidden layer units/nodes (a.k.a. "neurons") act as new features

One hidden Layer
(with K=2 hidden units)

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



Can think of this model as a combination of two predictions h_{n1} and h_{n2} of two simple functions (red and blue weights)

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

Input Layer
(with D=3 visible units)

Input layer units/nodes denote the original features of input x_n

MLP is also called feedforward fully-connected network

Illustration: Neural Net with Single Hidden Layer

- Compute K pre-activations for each input $\boldsymbol{x}_n$

A linear model with learnable weight vec $\boldsymbol{w}_k$

$$z_{nk} = \boldsymbol{w}_k^\top \boldsymbol{x}_n = \sum_{d=1}^{D} w_{dk} x_{nd} \quad (k = 1,2,\dots,K)$$

- Apply nonlinear activation on each pre-act

Called a hidden unit

$$h_{nk} = g(z_{nk}) \quad (k = 1,2,\dots,K)$$

- Apply a linear model with $\boldsymbol{h}_n$ acting as features

A linear model with learnable weight vec $\boldsymbol{v}$

Score of the input

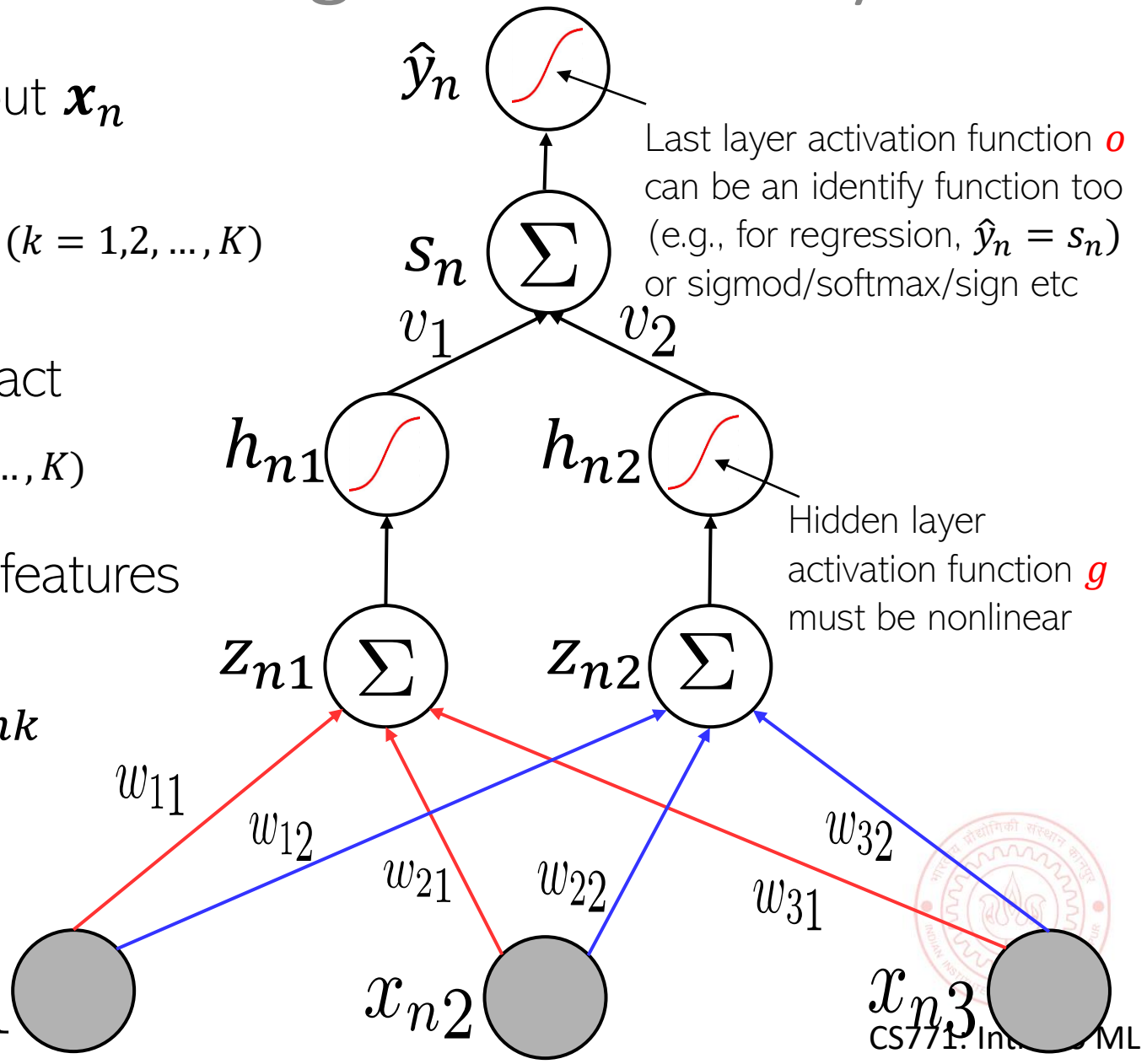
$$s_n = \boldsymbol{v}^\top \boldsymbol{h}_n = \sum_{k=1}^{K} v_k h_{nk}$$

- Finally, output is produced as

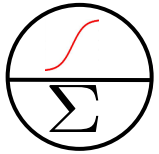
Score converted to the actual prediction

$$\hat{y}_n = o(s_n)$$

- Loss: $\mathcal{L}(\boldsymbol{W}, \boldsymbol{v}) = \sum_{n=1}^{N} \ell(y_n, \hat{y}_n)$

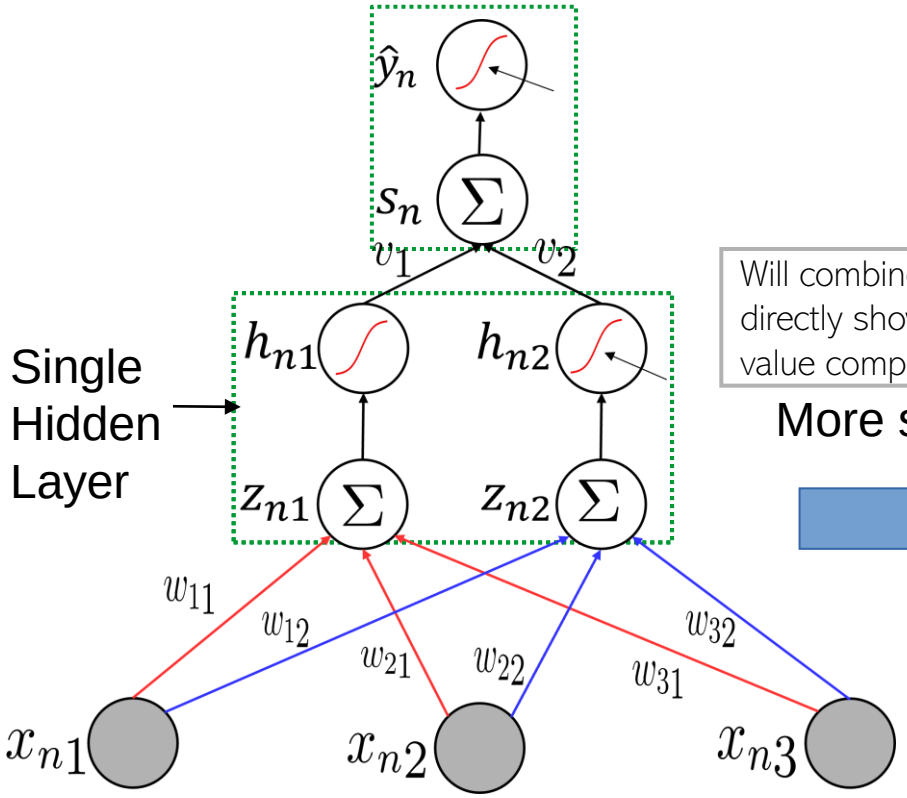


Neural Nets: A Compact Illustration



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

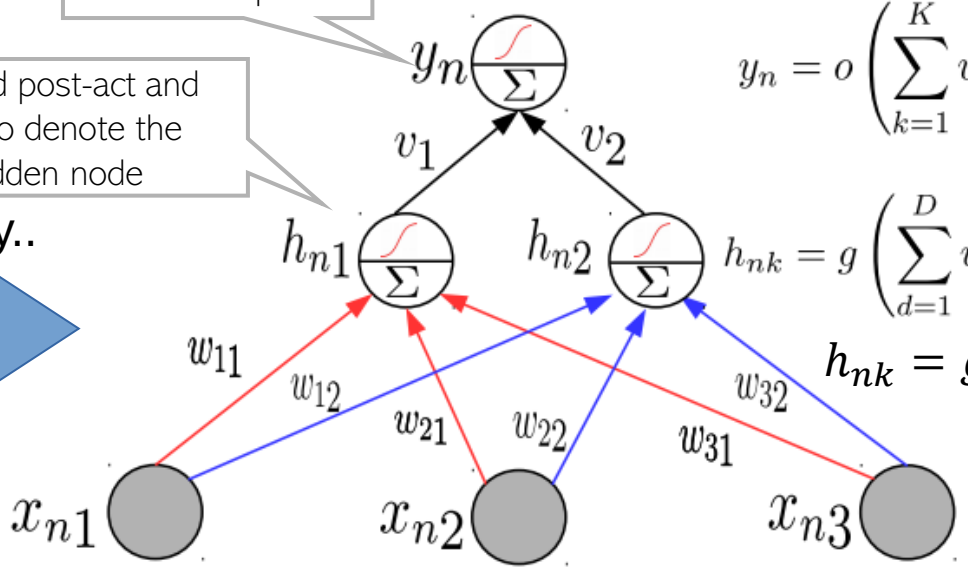
- Note: Hidden layer pre-act z_{nk} and post-act h_{nk} will be shown together for brevity



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



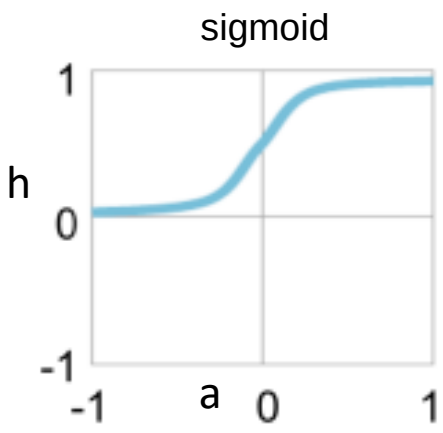
$$y_n = o\left(\sum_{k=1}^{K} v_k h_{nk}\right)$$

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

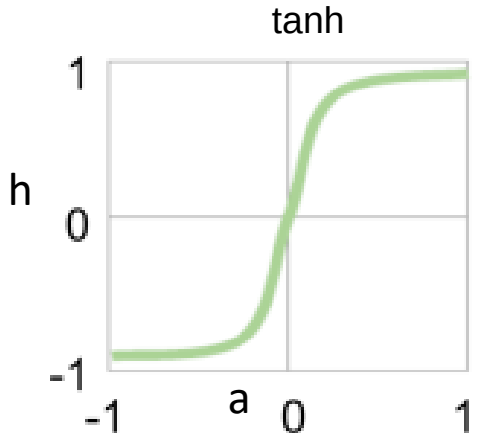
$$h_{nk} = g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)$$

- Denoting $\boldsymbol{W} = [\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_K]$, $\boldsymbol{w}_k \in \mathbb{R}^D$, $\boldsymbol{h}_n = g(\boldsymbol{W}^\top \boldsymbol{x}_n) \in \mathbb{R}^K$ $(K = 2, D = 3$ above). Note: g applied elementwise on pre-activation vector $\boldsymbol{z}_n = \boldsymbol{W}^\top \boldsymbol{x}_n$

Activation Functions: Some Common Choices



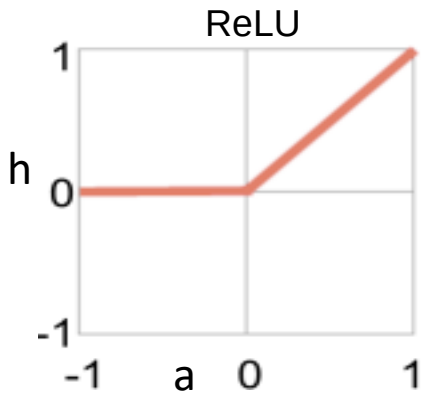
For sigmoid as well as tanh, gradients saturate (become close to zero as the function tends to its extreme values)



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)

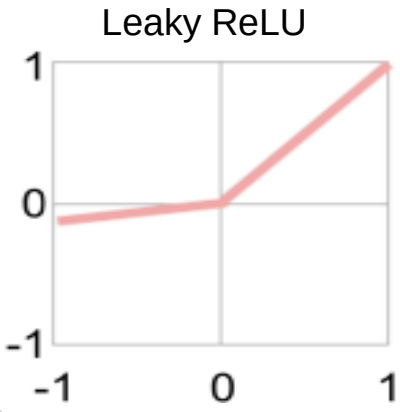
Sigmoid: $h = \sigma(a) = \frac{1}{1+\exp(-a)}$

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



ReLU and Leaky ReLU are among the most popular ones (also efficient to compute)

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



$y = \boldsymbol{v}^\top(g(\boldsymbol{W}^\top x))$
$= \boldsymbol{v}^\top \boldsymbol{W}^\top x$

Still linear

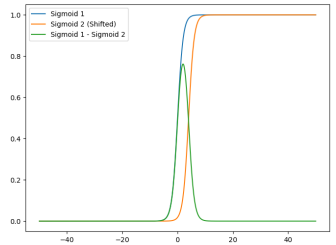
Imp: Without nonlinear activation, a deep neural network is equivalent to a linear model no matter how many layers we use

Most activation functions are monotonic but there exist some non-monotonic activation functions as well (e.g., Swish: $a \times \sigma(\beta a)$)

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

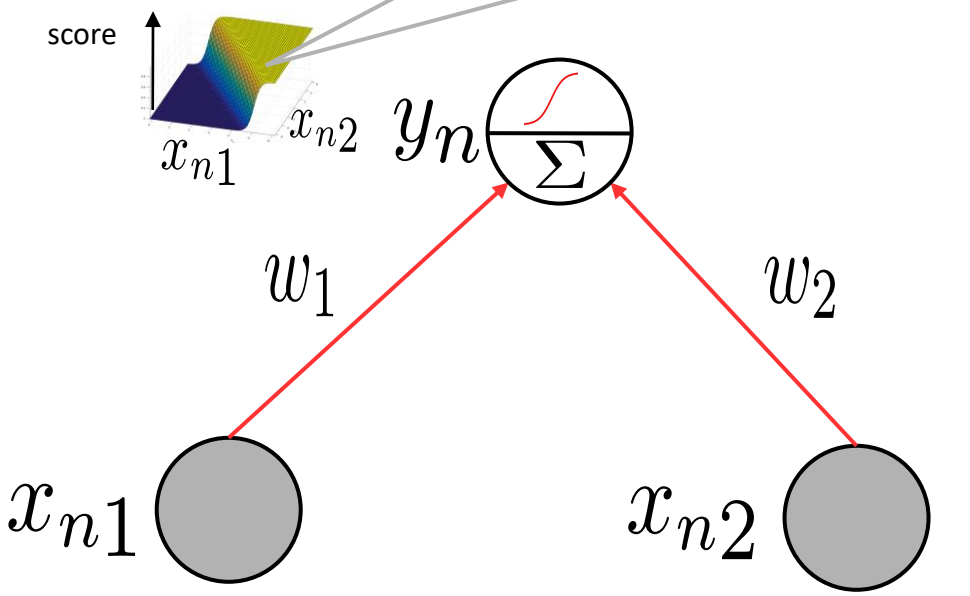
Leaky ReLU: $h = \max(\beta a, a)$
where β is a small postive number

MLP Can Learn Any Nonlinear Function



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

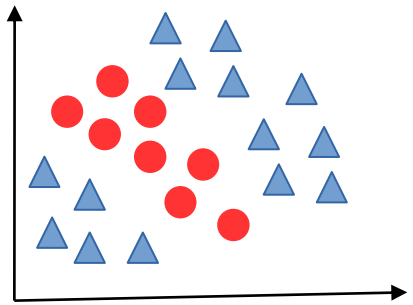
Score monotonically increases. One-sided increase (not ideal for learning nonlinear decision boundaries). Just a linear model



Standard Single "Perceptron" Classifier (no hidden units)

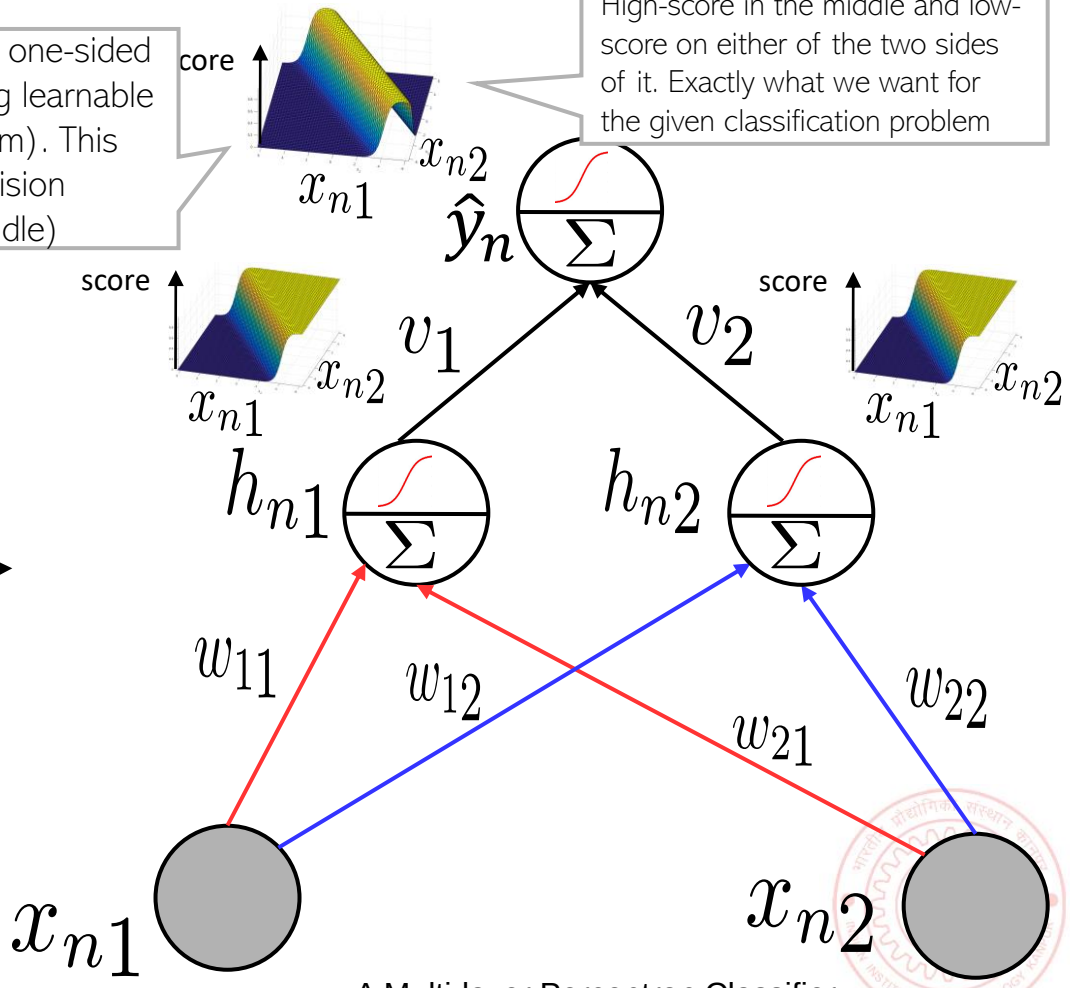
A single hidden layer MLP with sufficiently large number of hidden units can approximate any function (Hornik, 1991)

Obtained by composing the two one-sided increasing score functions (using learnable weights v_1 and v_2 to "add" them). This can now learn the nonlinear decision boundary (high score in the middle)



A nonlinear classification problem

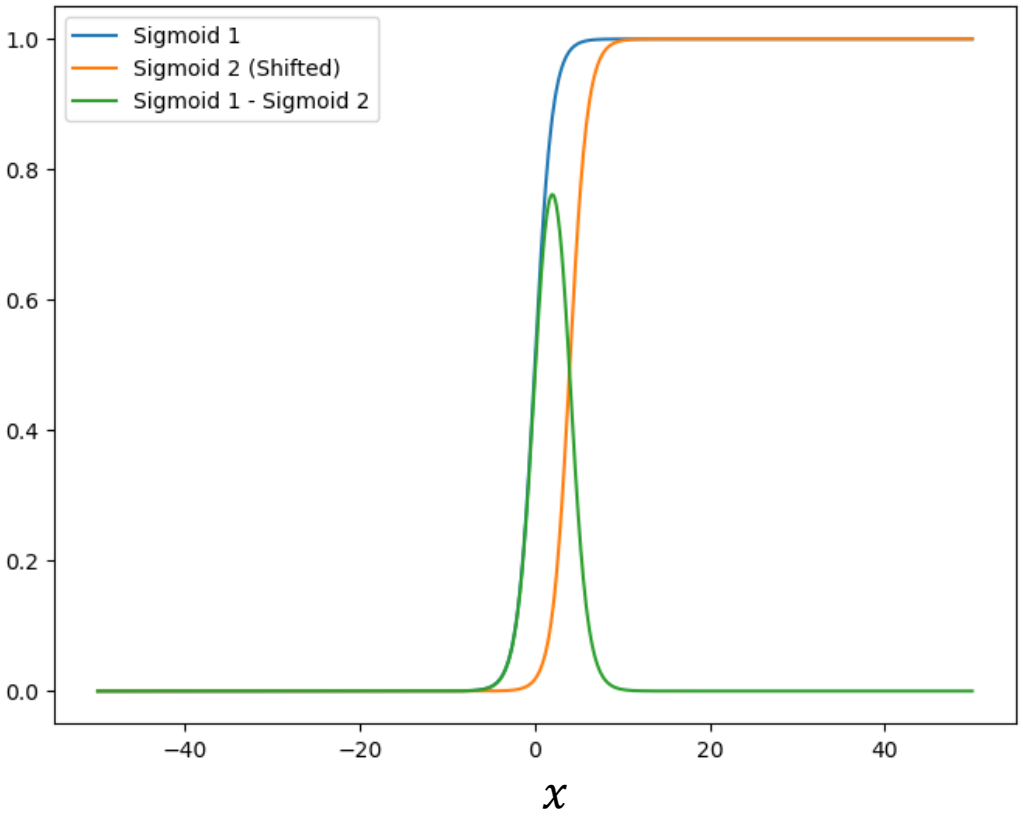
High-score in the middle and low-score on either of the two sides of it. Exactly what we want for the given classification problem



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

Superposition of two linear models = Nonlinear model

$p(y = \text{red}|x)$



Two sigmoids (blue and orange) can be combined via a shift and a subtraction operation to result in a nonlinear separation boundary

Likewise, more than two sigmoids can be combined to learn even more sophisticated separation boundaries

Nonlinear separation boundary

Examples of some basic NN/MLP architectures

Single Hidden Layer and Single Output

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

$$\hat{y}_n = o(\boldsymbol{v}^\top \boldsymbol{h}_n)$$

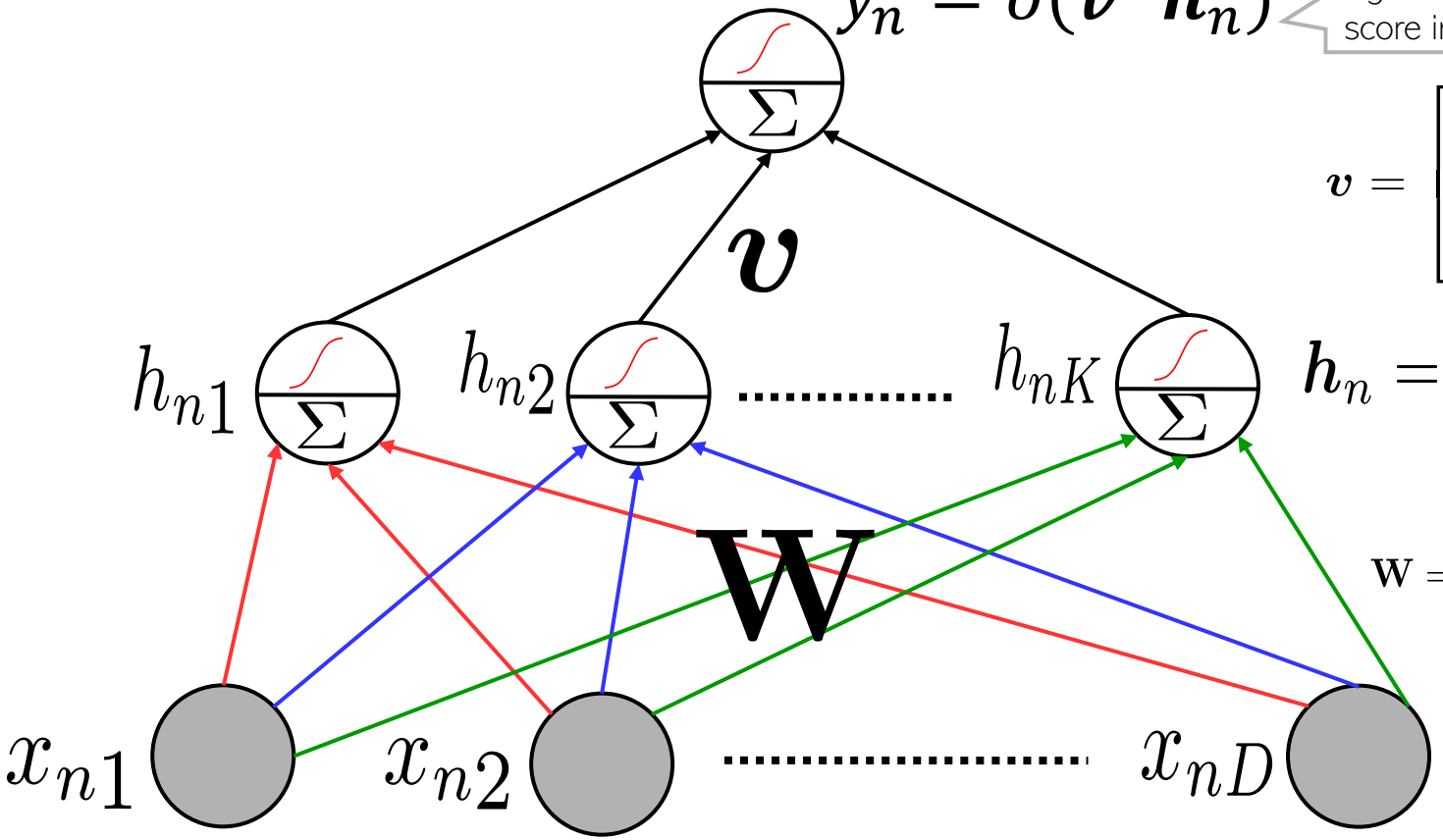
o is a suitable function (e.g., identity, sign, or sigmoid, etc) to convert the score into the actual response

$$\boldsymbol{v} = \begin{bmatrix} v_1 \\ \vdots \\ v_K \end{bmatrix}$$

g is a nonlinear activation function

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

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

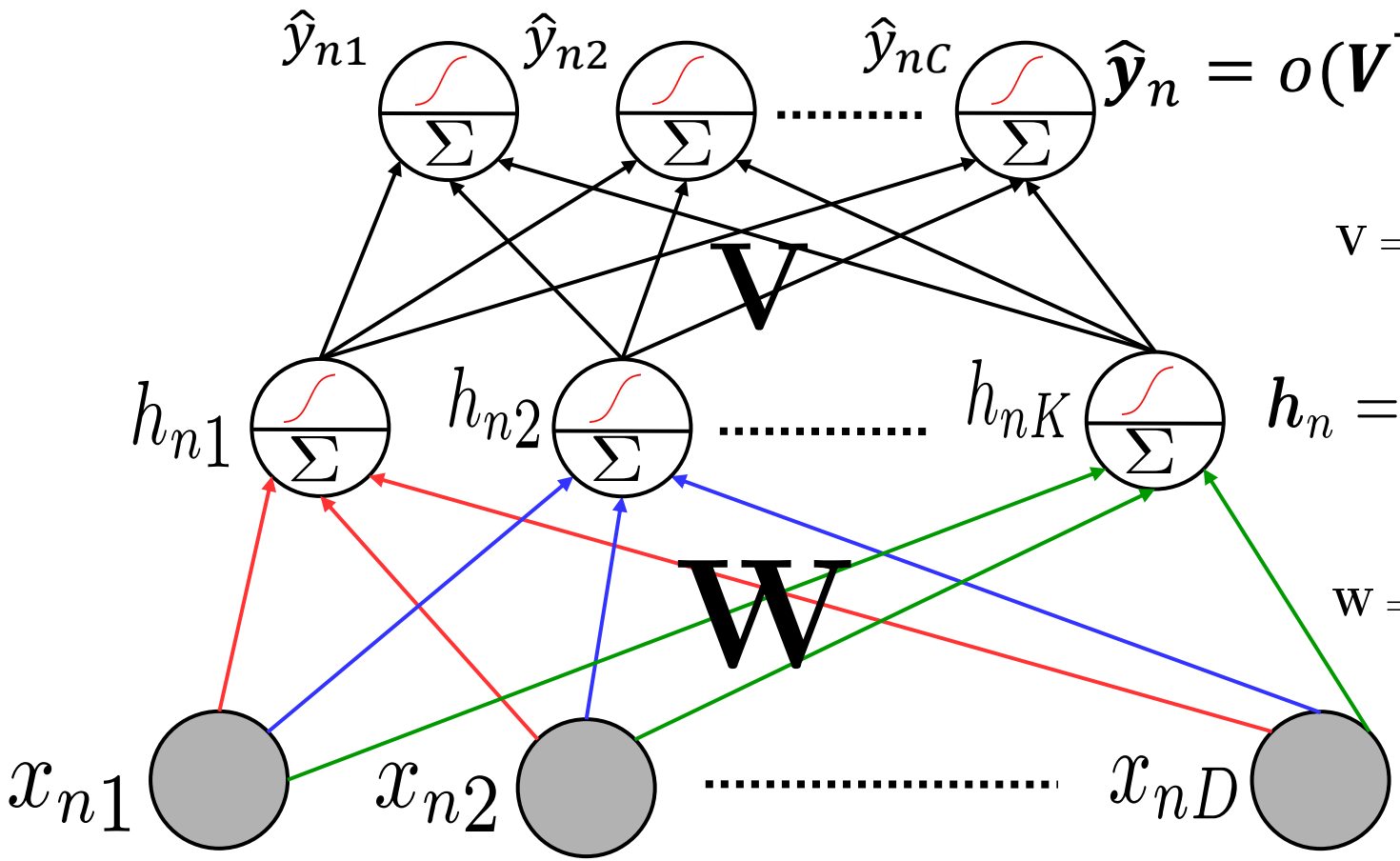


Single Hidden Layer and Multiple Outputs

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

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

o is a suitable function (e.g., identity, sign, or softmax, etc) to convert the score into the actual response vector for $\boldsymbol{x}_n$



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

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

g is a nonlinear activation function

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

Multiple Hidden Layers (One/Multiple Outputs)

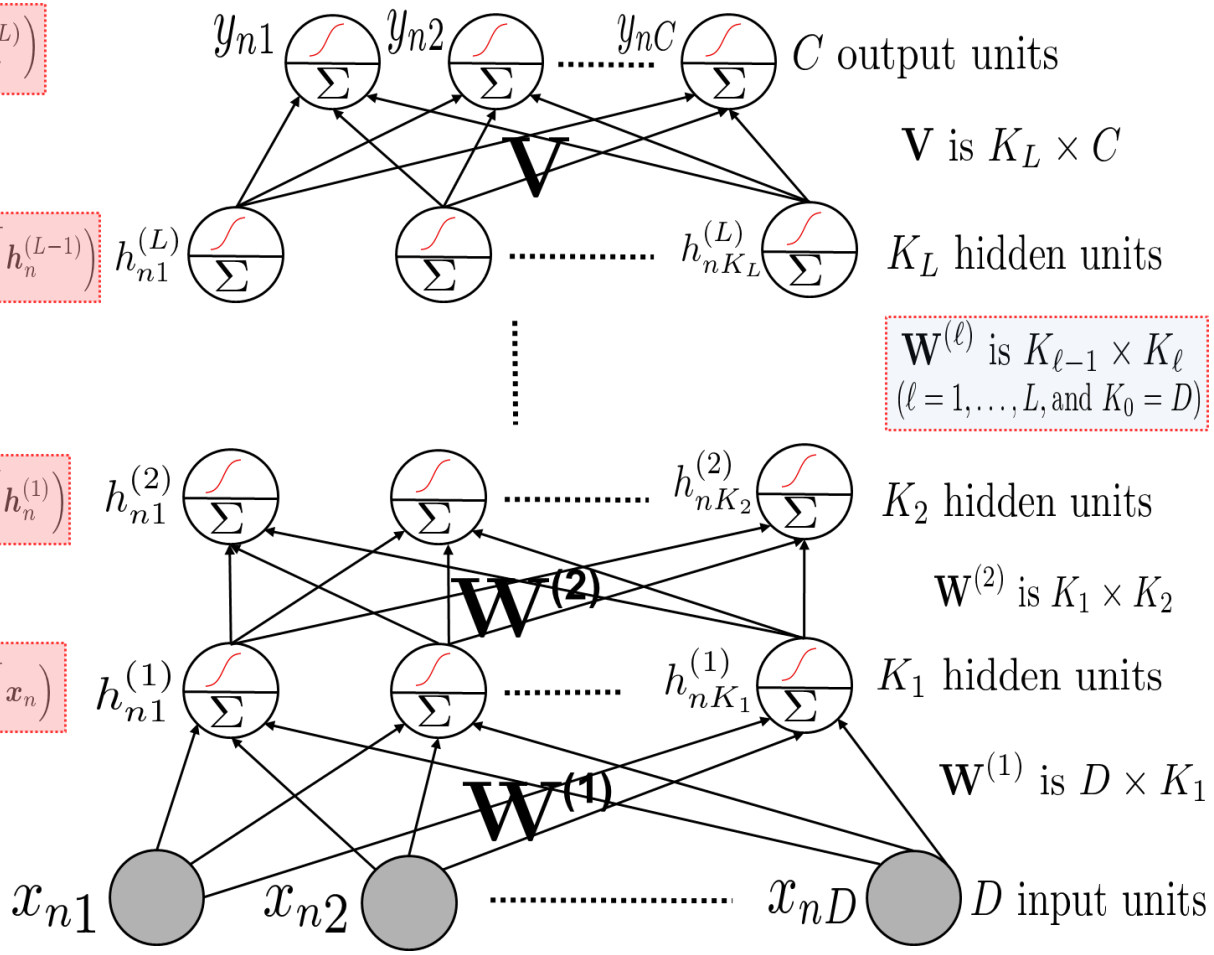
- Most general case: Multiple hidden layers with (with same or different number of hidden nodes in each) and a scalar or vector-valued output

$y_n = o\left(\mathbf{V}^\top h_n^{(L)}\right)$

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

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

$h_n^{(1)} = g\left(\mathbf{W}^{(1)\top} x_n\right)$



Each hidden layer uses a nonlinear activation function g (essential, otherwise the network can't learn nonlinear functions and reduces to a linear model)

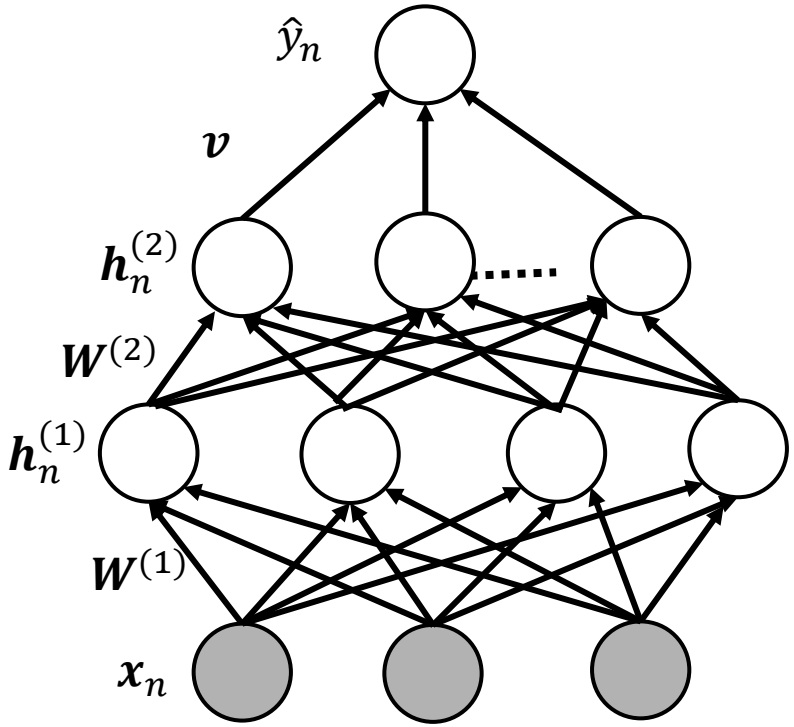
Note: Nonlinearity g is applied element-wise on its inputs so $h_n^{(\ell)}$ has the same size as vector $W^{(\ell)} h_n^{(\ell-1)}$

The Bias Term

- Each layer's pre-activations $\boldsymbol{z}_n^{(\ell)}$ have a an add bias term $\boldsymbol{b}^{(\ell)}$ (has the same size as $\boldsymbol{z}_n^{(\ell)}$ and $\boldsymbol{h}_n^{(\ell)}$) as well

$$\boldsymbol{z}_n^{(\ell)} = \boldsymbol{W}^{(\ell)\top}\boldsymbol{h}_n^{(\ell-1)} + \boldsymbol{b}^{(\ell)}$$

$$\boldsymbol{h}_n^{(\ell)} = g(\boldsymbol{z}_n^{(\ell)})$$



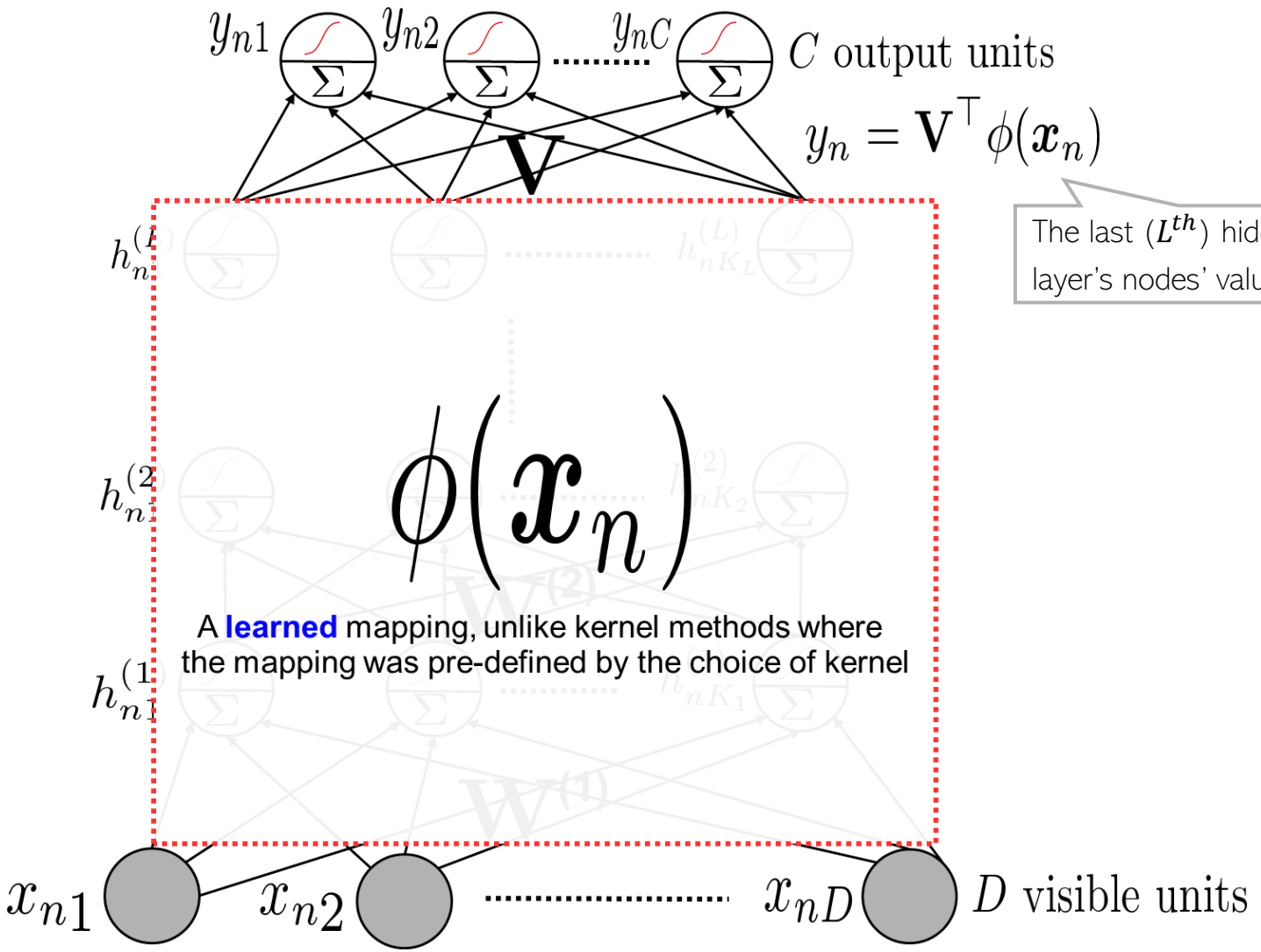
- Bias term increases the expressiveness of the network and ensures that we have nonzero activations/pre-activations even if this layer's input is a vector of all zeros

- Note that the bias term is the same for all inputs (does not depend on $\boldsymbol{n}$)

- The bias term $\boldsymbol{b}^{(\ell)}$ is also learnable

Neural Nets are Feature Learners

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



$$y_n = \mathbf{V}^\top \phi(\boldsymbol{x}_n)$$

The last (L^{th}) hidden layer's nodes' values $\boldsymbol{h}_n^{(L)}$

Kernel Methods vs Neural Nets

Also note that neural nets are faster than kernel methods at test time since kernel methods need to store the training examples at test time whereas neural nets do not

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

$$y_n = \boldsymbol{w}^\top \phi(\boldsymbol{x}_n) \qquad \text{OR} \qquad y_n = \sum_{i=1}^{N} \alpha_n k(\boldsymbol{x}_i, \boldsymbol{x}_n)$$

- It's like a one hidden layer NN with
  - Pre-defined N features $\{k(\boldsymbol{x}_i, \boldsymbol{x}_n)\}_{i=1}^{N}$ acting as feature vector $\boldsymbol{h}_n$
  - $\{\alpha_i\}_{n=1}^{N}$ are learnable output layer weights

- It's also like a one hidden layer NN with
  - Pre-defined M features $\boldsymbol{\phi}(\boldsymbol{x}_n)$ (M being size of feature mapping $\boldsymbol{\phi}$) acting as feature vector $\boldsymbol{h}_n$
  - $\boldsymbol{w} \in \mathbb{R}^M$ are learnable output layer weights

- Both kernel methods and deep neural networks extract new features from the inputs
  - For kernel methods, the features are pre-defined via the kernel function
  - For deep NN, the features are learned by the network

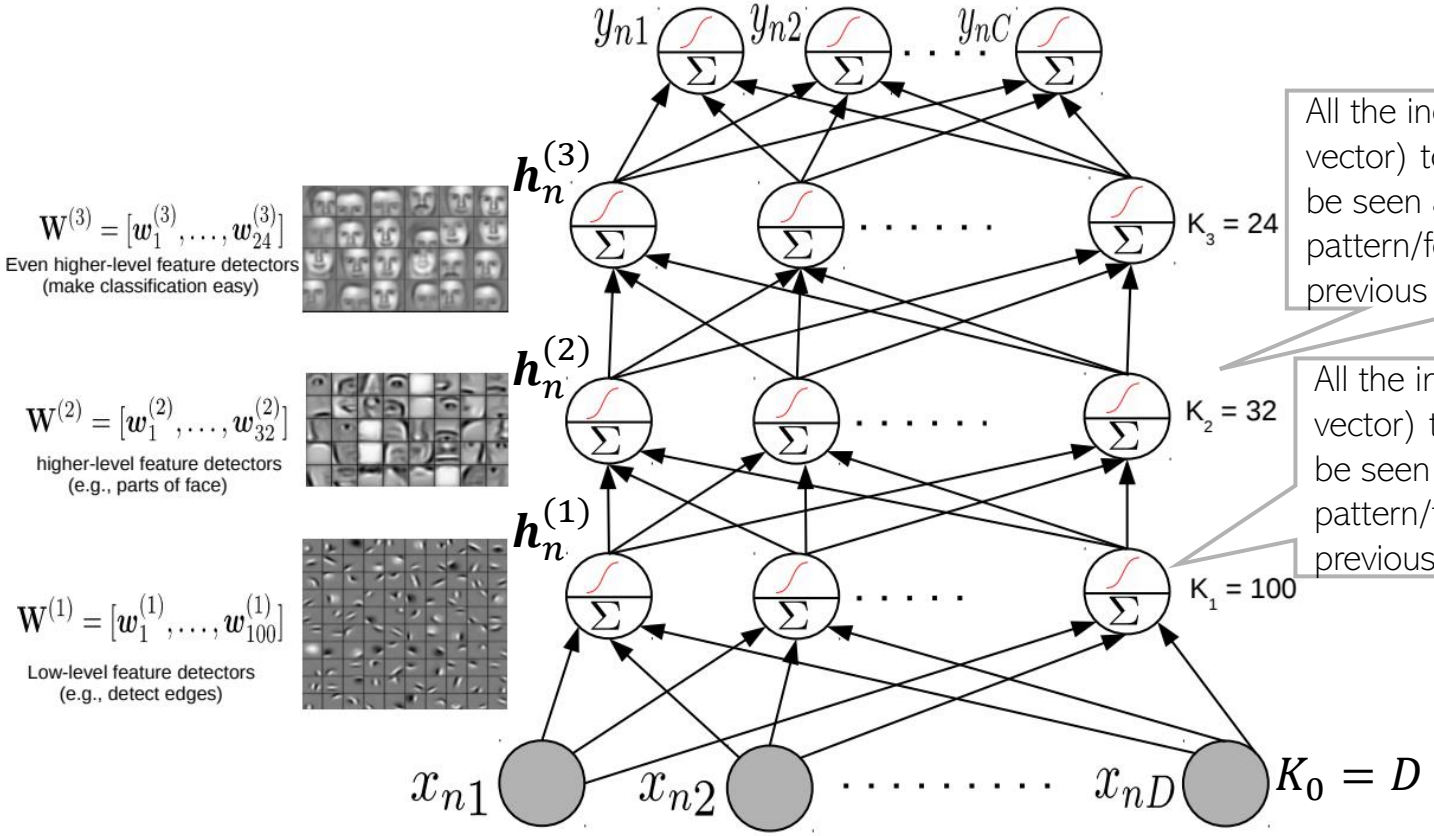
- Note: Kernels can also be learned from data ("kernel learning") in which case kernel methods and deep neural nets become even more similar in spirit ☺

Features Learned by a Neural Network

$\boldsymbol{W}^{(\ell)} = [\boldsymbol{w}_1^{(\ell)}, \boldsymbol{w}_2^{(\ell)}, \ldots, \boldsymbol{w}_{K_\ell}^{(\ell)}]$ is $K_{\ell-1} \times K_\ell$ matrix of weights between layer $\ell - 1$ and ℓ

$\boldsymbol{W}^{(\ell)}$ collectively represents the K_ℓ feature detectors for layer ℓ

- $\boldsymbol{w}_k^{(\ell)} \in \mathbb{R}^{K_{\ell-1}}$ denotes "feature detector" for feature $\boldsymbol{k}$ of layer $\boldsymbol{\ell}$
- For input $\boldsymbol{x}_n$, $h_{nk}^{(\ell)} = g(\boldsymbol{w}_k^{(\ell)\top} \boldsymbol{h}_n^{(\ell-1)})$ is the value of feature $\boldsymbol{k}$ in layer $\boldsymbol{\ell}$



All the incoming weights (a vector) to a hidden node can be seen as representing a pattern/feature-detector of previous layer's inputs

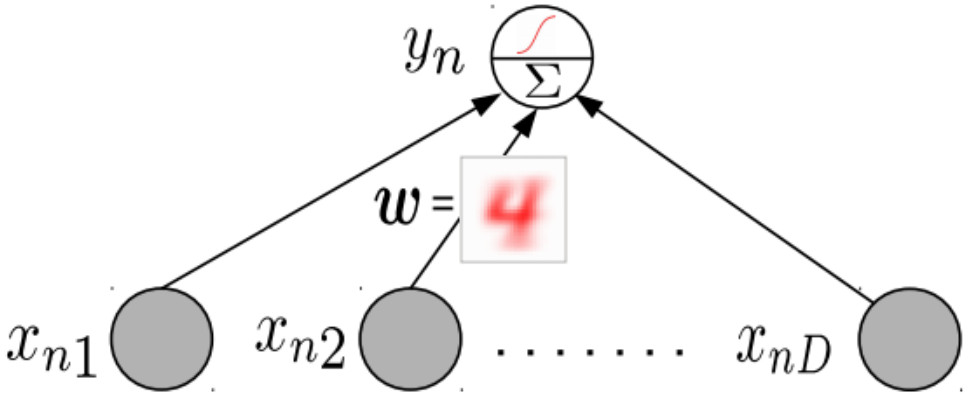
E.g., $\boldsymbol{w}_{32}^{(2)}$ denotes a feature detector

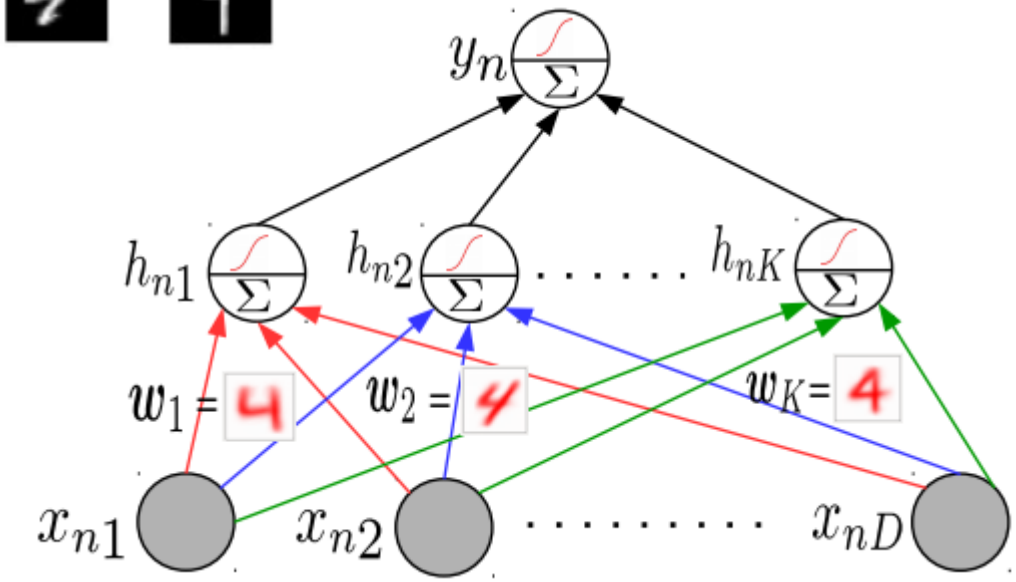
All the incoming weights (a vector) to a hidden node can be seen as representing a pattern/feature-detector of previous layer's inputs

E.g., $\boldsymbol{w}_{100}^{(1)}$ denotes a feature detector

Why Neural Networks Work Better: Another View

- Linear models tend to only learn the "average" pattern
  - E.g., Weight vector of a linear classification model represent average pattern of a class
- 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

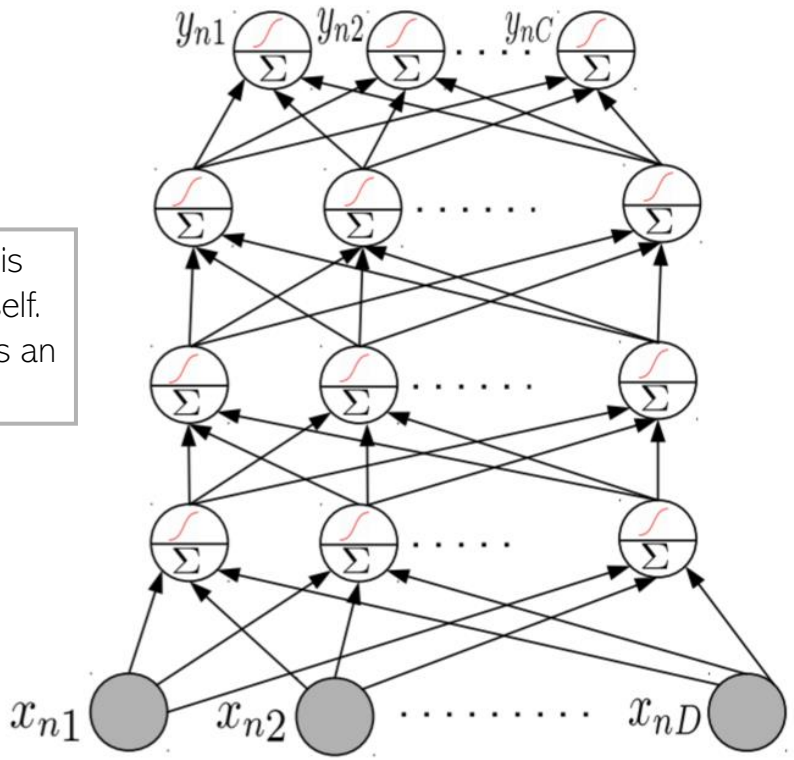




Neural Nets: Some Aspects

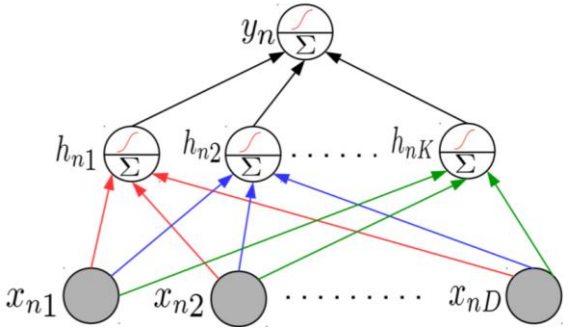
- Much of the magic lies in the hidden layers
- Hidden layers learn and detect good features
- Need to consider a few aspects
  - Number of hidden layers, number of units in each hidden layer
  - Why bother about many hidden layers and not use a single very wide hidden layer (recall Hornik's universal 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)?

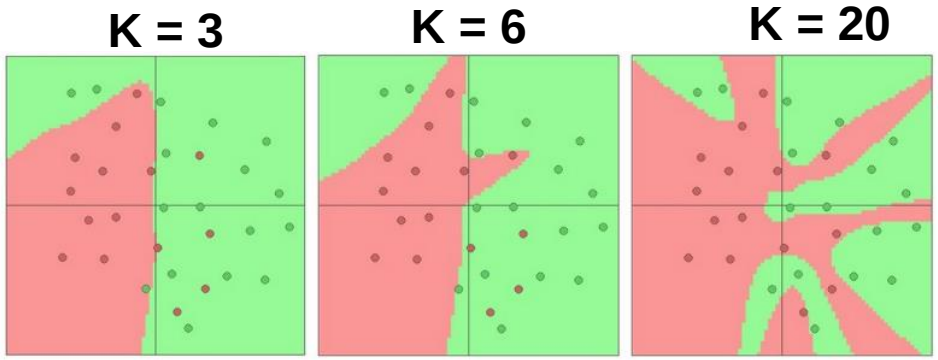
Choosing the right NN architecture is important and a research area in itself. Neural Architecture Search (NAS) is an automated technique to do this



Representational Power of Neural Nets

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

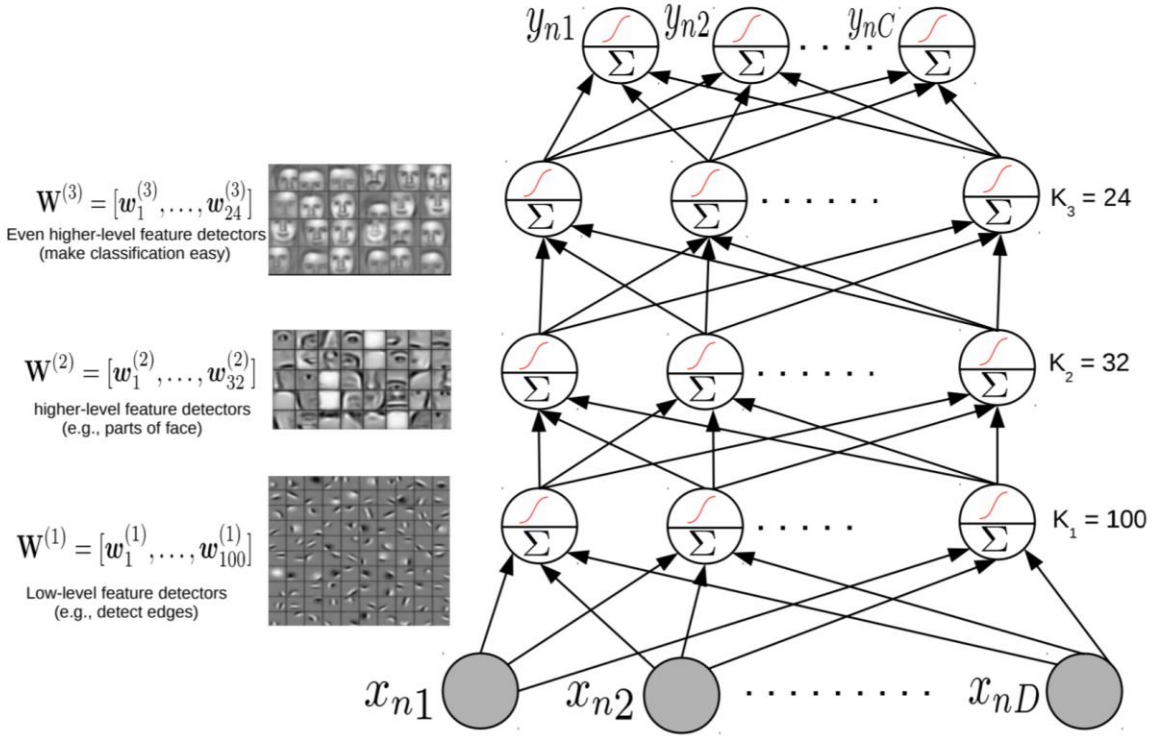




- Recall that each hidden unit "adds" a simple function to the overall function
- Increasing K (number of hidden units) will result in a more complex function
- Very large K seems to overfit (see above fig). Should we instead prefer small K?
- No! It is better to use large K and regularize well. Reason/justification:
  - 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

Wide or Deep?

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



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

Training Deep Neural Networks: The Backpropagation Algorithm

Background: Gradient and Jacobian

- Let $\boldsymbol{y} = f(\boldsymbol{x})$, where $f: \mathbb{R}^P \to \mathbb{R}^Q$, $\boldsymbol{x} \in \mathbb{R}^P$, $\boldsymbol{y} \in \mathbb{R}^Q$. Denote $\boldsymbol{y} = [f_1(\boldsymbol{x}), \dots, f_Q(\boldsymbol{x})]$
- The gradient of each component $y_i = f_i(\boldsymbol{x}) \in \mathbb{R}$ $(i = 1,2,\dots,Q)$ w.r.t. $\boldsymbol{x} \in \mathbb{R}^P$ is

$$\nabla f_i(\boldsymbol{x}) = \frac{\partial y_i}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial y_i}{\partial x_1} & \dots & \frac{\partial y_i}{\partial x_P} \end{bmatrix} \in \mathbb{R}^{1\times P}$$

Note: Gradient expressed here as a row vector (has the same length as $\boldsymbol{x}$ which is a column vector) for notational convenience later

- Likewise, the gradient of whole vector $\boldsymbol{y} \in \mathbb{R}^Q$ w.r.t. vector $\boldsymbol{x} \in \mathbb{R}^P$ can be defined using the $Q \times P$ Jacobian matrix J^f whose rows consist of the above gradients

Row i contains the gradient vector of $y_i = f_i(\boldsymbol{x})$ w.r.t. $\boldsymbol{x}$

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = J^f \qquad J^f_{ij} = \frac{\partial y_i}{\partial x_j} \qquad J^f = \begin{bmatrix} \nabla f_1(\boldsymbol{x}) \\ \vdots \\ \nabla f_Q(\boldsymbol{x}) \end{bmatrix} \in \mathbb{R}^{Q\times P}$$

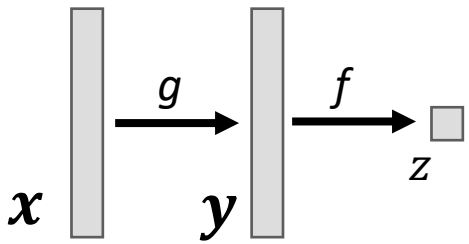
For a function $f: \mathbb{R}^P \to \mathbb{R}^Q$, the Jacobian $J^f \in \mathbb{R}^{Q\times P}$ (a matrix)

Likewise, for a function $f: \mathbb{R}^{P\times Q} \to \mathbb{R}^{R\times S}$, the Jacobian $J^f \in \mathbb{R}^{R\times S\times P\times Q}$ (4D tensor)

More generally, for a function $f: \mathbb{R}^{I_1\times I_2\times\cdots} \to \mathbb{R}^{O_1\times O_2\times\cdots}$, its Jacobian $J^f \in \mathbb{R}^{O_1\times O_2\times..\times I_1\times I_2\times\cdots}$ (a tensor)

Background: Multivariate Chain Rule of Calculus

- Let $\boldsymbol{x} \in \mathbb{R}^P$, $\boldsymbol{y} = g(\boldsymbol{x}) \in \mathbb{R}^Q$, $z = f(\boldsymbol{y}) \in \mathbb{R}$, where $g: \mathbb{R}^P \to \mathbb{R}^Q$, $f: \mathbb{R}^Q \to \mathbb{R}$



$1 \times P$ vector derivative

Scalar derivative

$1 \times P$ vector derivative

$$\frac{\partial z}{\partial \boldsymbol{x}} = \sum_{i=1}^{Q} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial \boldsymbol{x}}$$

Used chain rule of total derivatives

Sum is needed since z depends on $\boldsymbol{y}$, and $\boldsymbol{y}$ is a vector

- The above can be written as a product of a vector and a matrix

$1 \times P$

Turns out to be a product of Jacobian of f and g in that order ☺

$1 \times Q$ gradient (same as Jacobian since f is scalar)

$Q \times P$ Jacobian of g

$$\frac{\partial z}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial z}{\partial y_1} & \cdots & \frac{\partial z}{\partial y_Q} \end{bmatrix} \times \begin{bmatrix} \frac{\partial y_1}{\partial \boldsymbol{x}} \\ \vdots \\ \frac{\partial y_Q}{\partial \boldsymbol{x}} \end{bmatrix} = \nabla f(\boldsymbol{y}) \times \begin{bmatrix} \nabla g_1(\boldsymbol{x}) \\ \vdots \\ \nabla g_P(\boldsymbol{x}) \end{bmatrix} = \nabla f(\boldsymbol{y}) \times J^g$$

- More generally, let $\boldsymbol{w} \in \mathbb{R}^P$, $\boldsymbol{x} = h(\boldsymbol{w}) \in \mathbb{R}^Q$, $\boldsymbol{y} = g(\boldsymbol{x}) \in \mathbb{R}^R$, $\boldsymbol{z} = f(\boldsymbol{y}) \in \mathbb{R}^S$

Product of the 3 Jacobians in that order (simple! ☺)

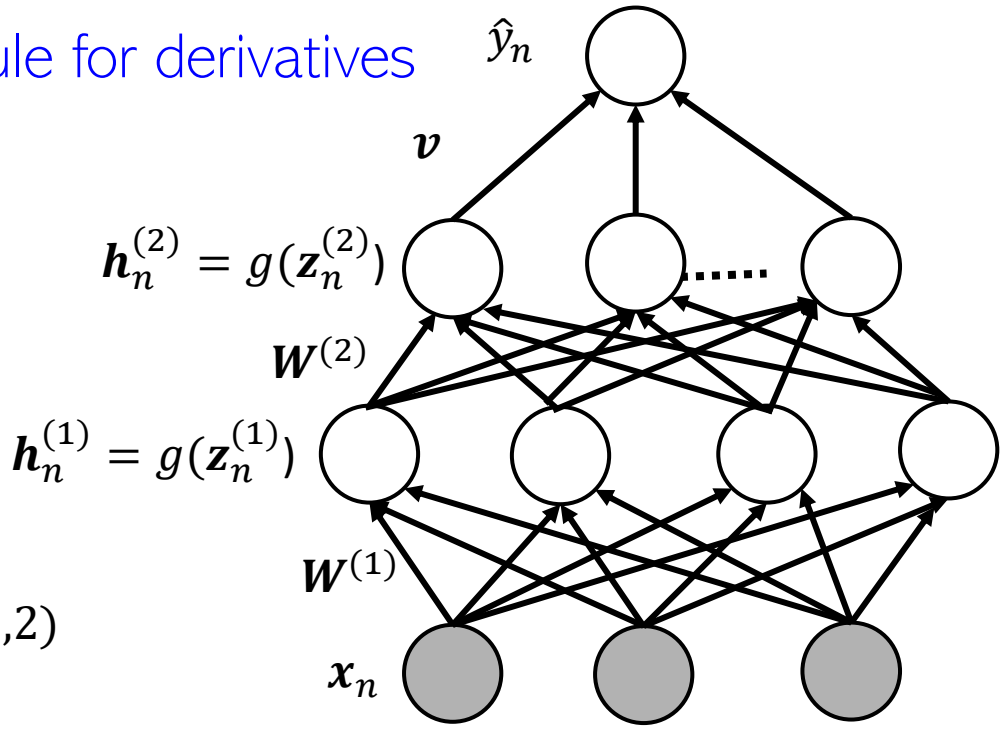
$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{w}} = J^f \times J^g \times J^h \in \mathbb{R}^{S \times P}$$

Note that chain rule for scalar variables w, x, y, z is defined in a similar way as $\frac{\partial z}{\partial w} = f'(y)g'(x)h'(w)$

Backpropagation (Backprop)

- Backprop is gradient descent with multivariate chain rule for derivatives
- Consider a two hidden layer neural network

$$\mathcal{L}(\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}, \boldsymbol{v}) = \sum_{n=1}^{N} \ell(y_n, \hat{y}_n) = \sum_{n=1}^{N} \ell_n$$



- We wish to minimize the loss
- The gradient based updates will be

$$\boldsymbol{v} = \boldsymbol{v} - \eta \frac{\partial \mathcal{L}}{\partial \boldsymbol{v}} \qquad \boldsymbol{W}^{(i)} = \boldsymbol{W}^{(i)} - \eta \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^{(i)}} \qquad (i = 1,2)$$

- Since $\mathcal{L} = \sum_{n=1}^{N} \ell_n$, we need to compute $\frac{\partial \ell_n}{\partial \boldsymbol{v}}$ and $\frac{\partial \ell_n}{\partial \boldsymbol{W}^{(i)}}$ $(i = 1,2)$
- Assume output activation o as identity $(\hat{y}_n = \boldsymbol{v}^\top \boldsymbol{h}_n^{(2)})$

Derivative of ℓ_n w.r.t. $\hat{y}_n$

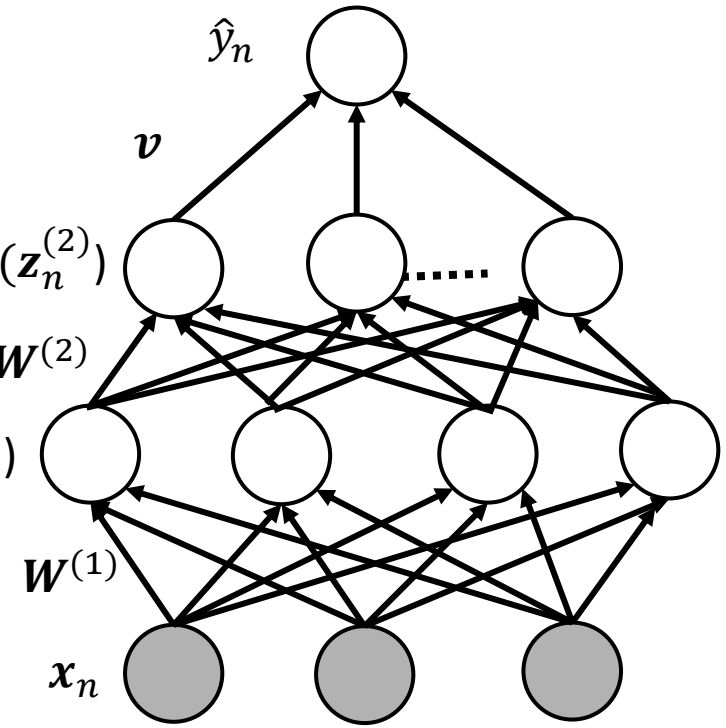
$$\frac{\partial \ell_n}{\partial \boldsymbol{v}} = \frac{\partial \ell_n}{\partial \hat{y}_n} \frac{\partial \hat{y}_n}{\partial \boldsymbol{v}} = \ell'(y_n, \hat{y}_n) \boldsymbol{h}_n^{(2)}$$

Backpropagation in detail

- Let's now look at $\frac{\partial \ell_n}{\partial \boldsymbol{W}^{(2)}}$ where $\ell_n = \ell(y_n, \hat{y}_n)$ and $\hat{y}_n = \boldsymbol{v}^\top \boldsymbol{h}_n^{(2)}$

$$\frac{\partial \ell_n}{\partial \boldsymbol{W}^{(2)}} = \frac{\partial \ell_n}{\partial \hat{y}_n}\frac{\partial \hat{y}_n}{\partial \boldsymbol{W}^{(2)}} = \ell'(y_n, \hat{y}_n)\frac{\partial \hat{y}_n}{\partial \boldsymbol{W}^{(2)}}$$

$$\frac{\partial \hat{y}_n}{\partial \boldsymbol{W}^{(2)}} = \frac{\partial \hat{y}_n}{\partial \boldsymbol{h}_n^{(2)}}\frac{\partial \boldsymbol{h}_n^{(2)}}{\partial \boldsymbol{W}^{(2)}} + \frac{\partial \hat{y}_n}{\partial \boldsymbol{v}}\frac{\partial \boldsymbol{v}}{\partial \boldsymbol{W}^{(2)}}$$



- Since $\boldsymbol{v}$ doesn't depend on $\boldsymbol{W}^{(2)}$, $\frac{\partial \boldsymbol{v}}{\partial \boldsymbol{W}^{(2)}} = 0$

Using transpose since we assume gradient to be a row vector

Jacobian of size $1 \times K_2 \times K_1$

$$\frac{\partial \ell_n}{\partial \boldsymbol{W}^{(2)}} = \ell'(y_n, \hat{y}_n)\frac{\partial \hat{y}_n}{\partial \boldsymbol{h}_n^{(2)}}\frac{\partial \boldsymbol{h}_n^{(2)}}{\partial \boldsymbol{W}^{(2)}} = \ell'(y_n, \hat{y}_n)\boldsymbol{v}^\top \frac{\partial \boldsymbol{h}_n^{(2)}}{\partial \boldsymbol{W}^{(2)}}$$

- We now need $\frac{\partial \boldsymbol{h}_n^{(2)}}{\partial \boldsymbol{W}^{(2)}}$. Using $\boldsymbol{h}_n^{(2)} = g(\boldsymbol{z}_n^{(2)})$ where $\boldsymbol{z}_n^{(2)} = \boldsymbol{W}^{(2)\top}\boldsymbol{h}_n^{(1)}$ and g is elementwise applied nonlinearity on the vector $\boldsymbol{z}_n^{(2)}$

Jacobian of size $K_2 \times K_2 \times K_1$

$$\frac{\partial \boldsymbol{h}_n^{(2)}}{\partial \boldsymbol{W}^{(2)}} = \frac{\partial \boldsymbol{h}_n^{(2)}}{\partial \boldsymbol{z}_n^{(2)}}\frac{\partial \boldsymbol{z}_n^{(2)}}{\partial \boldsymbol{W}^{(2)}} = \operatorname{diag}\left(g'\left(z_{n1}^{(2)}\right), \ldots, g'\left(z_{nK_2}^{(2)}\right)\right)\frac{\partial \boldsymbol{z}_n^{(2)}}{\partial \boldsymbol{W}^{(2)}}$$

This Jacobian is a tensor of size $K_2 \times K_2 \times K_1$

Diagonal matrix of size $K_2 \times K_2$ with Jacobian (gradient vector) of g along the diagonals

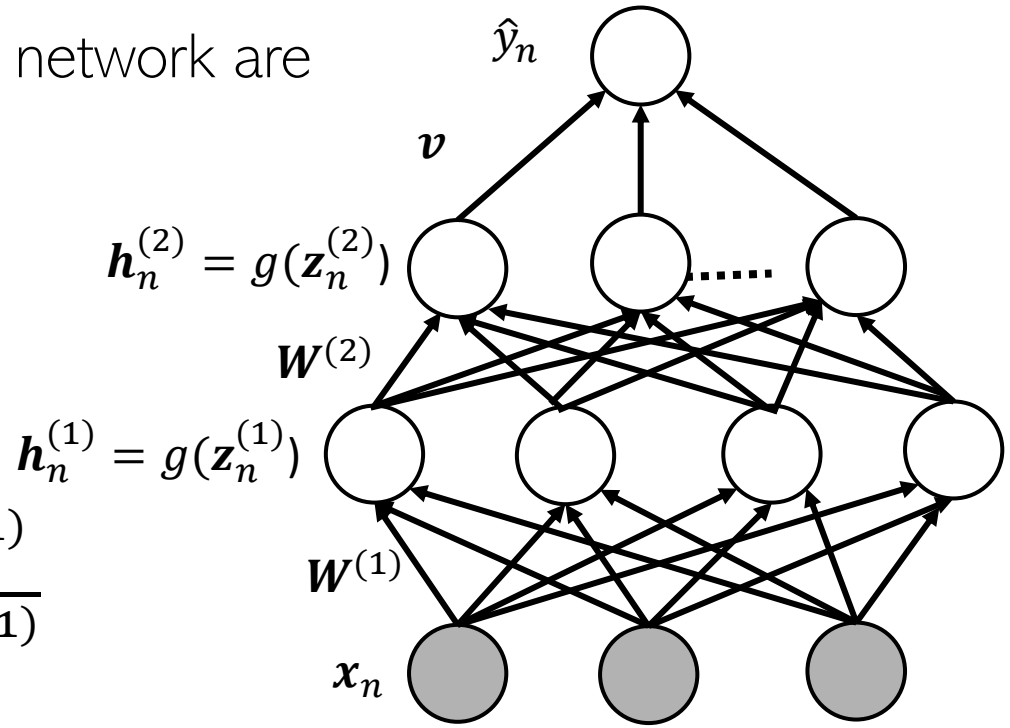
Backpropagation: Computation Reuse

- Summarizing, the required gradients/Jacobians for this network are

$$\frac{\partial \ell_n}{\partial \boldsymbol{v}} = \ell'(y_n, \hat{y}_n)\boldsymbol{h}_n^{(2)}$$

$$\frac{\partial \ell_n}{\partial \boldsymbol{W}^{(2)}} = \ell'(y_n, \hat{y}_n)\frac{\partial \hat{y}_n}{\partial \boldsymbol{h}_n^{(2)}}\frac{\partial \boldsymbol{h}_n^{(2)}}{\partial \boldsymbol{z}_n^{(2)}}\frac{\partial \boldsymbol{z}_n^{(2)}}{\partial \boldsymbol{W}^{(2)}}$$

$$\frac{\partial \ell_n}{\partial \boldsymbol{W}^{(1)}} = \ell'(y_n, \hat{y}_n)\frac{\partial \hat{y}_n}{\partial \boldsymbol{h}_n^{(2)}}\frac{\partial \boldsymbol{h}_n^{(2)}}{\partial \boldsymbol{z}_n^{(2)}}\frac{\partial \boldsymbol{z}_n^{(2)}}{\partial \boldsymbol{h}_n^{(1)}}\frac{\partial \boldsymbol{h}_n^{(1)}}{\partial \boldsymbol{z}_n^{(1)}}\frac{\partial \boldsymbol{z}_n^{(1)}}{\partial \boldsymbol{W}^{(1)}}$$

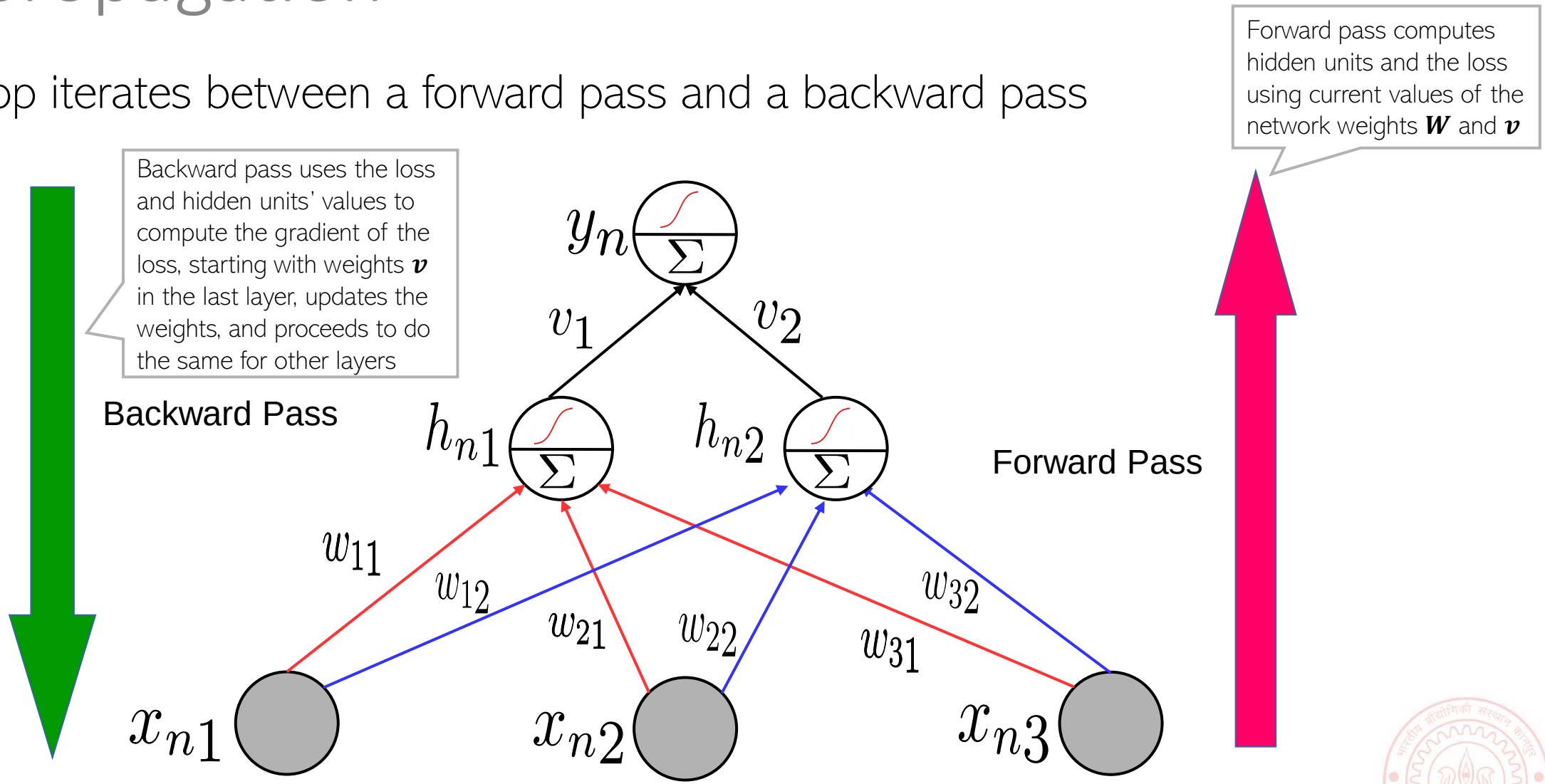


- Thus gradient computations done in upper layers can be stored and reused when computing the gradients in the lower layers (libraries like Tensorflow and Pytorch do so efficiently)
- Vanishing gradients: $\frac{\partial \boldsymbol{h}_n^{(i)}}{\partial \boldsymbol{z}_n^{(i)}} = \mathrm{diag}\left(g'\left(z_{n1}^{(i)}\right), \ldots, g'\left(z_{nK_i}^{(i)}\right)\right)$

Gradients in lower layers will have product of many such terms $\frac{\partial \boldsymbol{h}_n^{(i)}}{\partial \boldsymbol{z}_n^{(i)}}$. If g' is small (e.g., gradient of sigmoid or tanh), the gradient becomes vanishingly small for lower layers and becomes an issue (thus ReLU and other with non-saturating activations are preferred)

Backpropagation

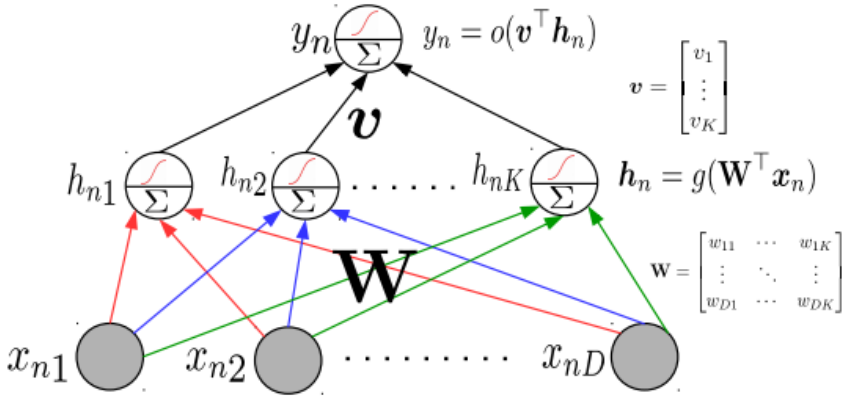
- Backprop iterates between a forward pass and a backward pass



- 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)

Backpropagation through an example

Consider a single hidden layer MLP



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

$$\mathcal{L} = \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \boldsymbol{v}^\top \boldsymbol{h}_n\right)^2$$

$$= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k h_{nk}\right)^2$$

$$= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)\right)^2$$

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

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

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

$$\frac{\partial \mathcal{L}}{\partial w_{dk}} = \sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial h_{nk}} \frac{\partial h_{nk}}{\partial w_{dk}}$$

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

$$\frac{\partial h_{nk}}{\partial w_{dk}} = g'(\boldsymbol{w}_k^\top \boldsymbol{x}_n)x_{nd} \qquad \text{(note: } h_{nk} = g(\boldsymbol{w}_k^\top \boldsymbol{x}_n))$$

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