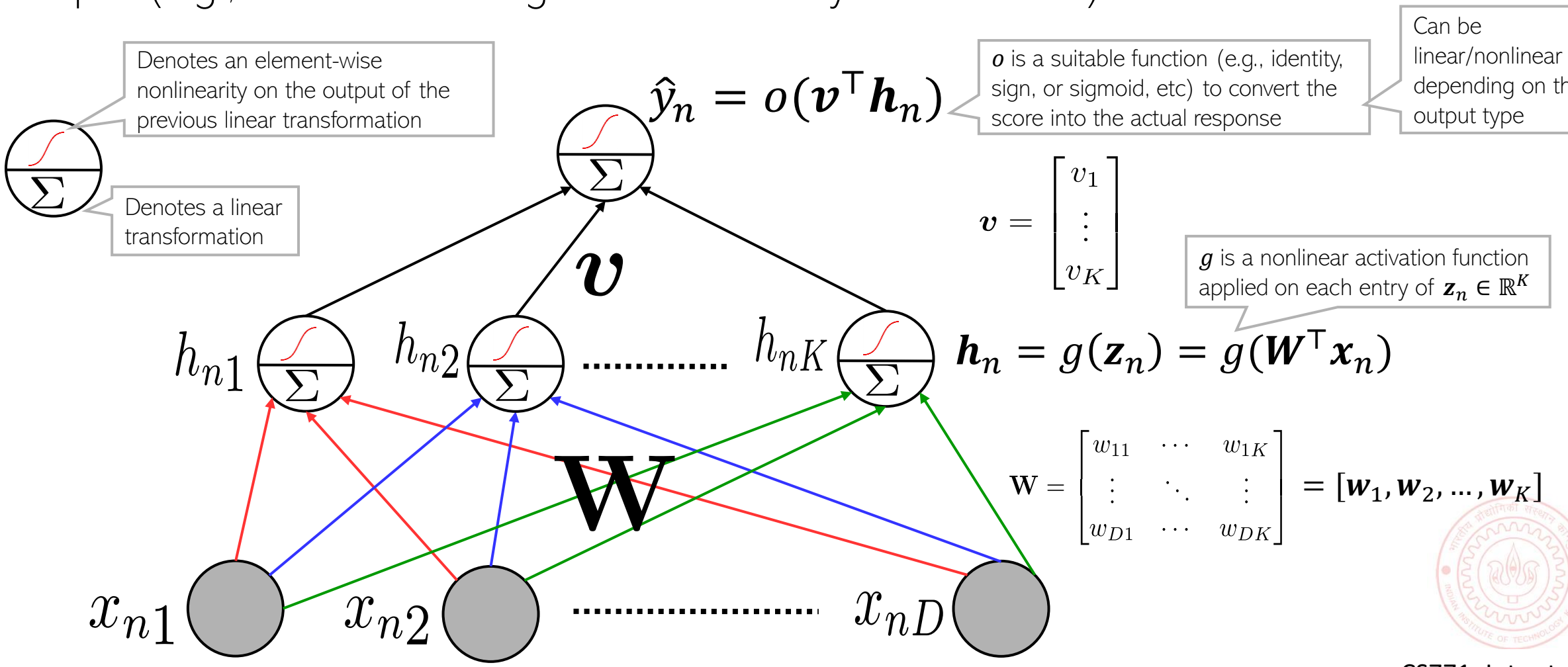
Deep Neural Networks (contd)

CS771: Introduction to Machine Learning

Single Hidden Layer and Single Output

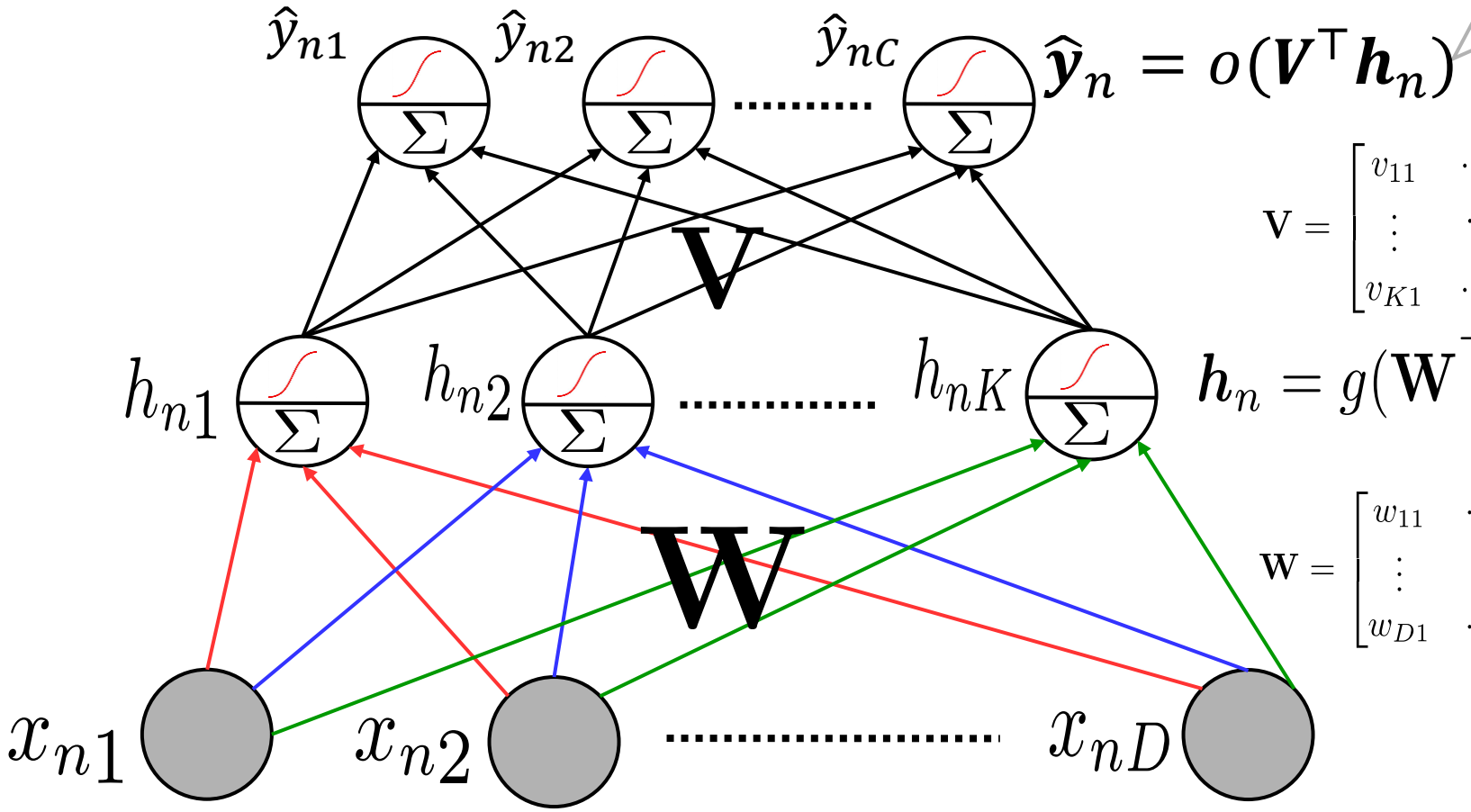
- Shown below: A neural network with one hidden layer with K nodes and a single output (e.g., scalar-valued regression or binary classification)



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)

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



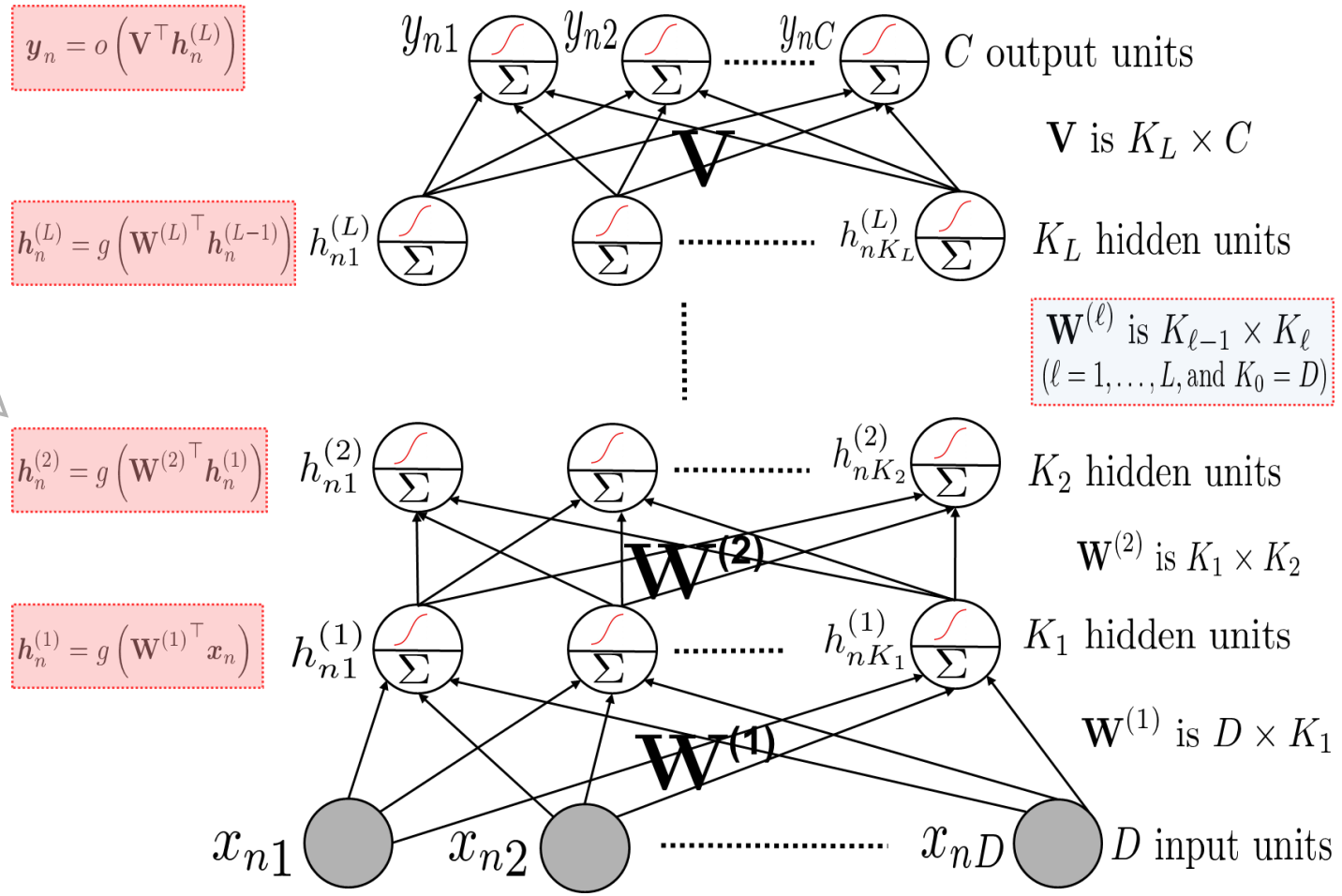
$$\mathbf{V} = \begin{bmatrix} v_{11} & \cdots & v_{1C} \\ \vdots & \ddots & \vdots \\ v_{K1} & \cdots & v_{KC} \end{bmatrix}$$

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



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

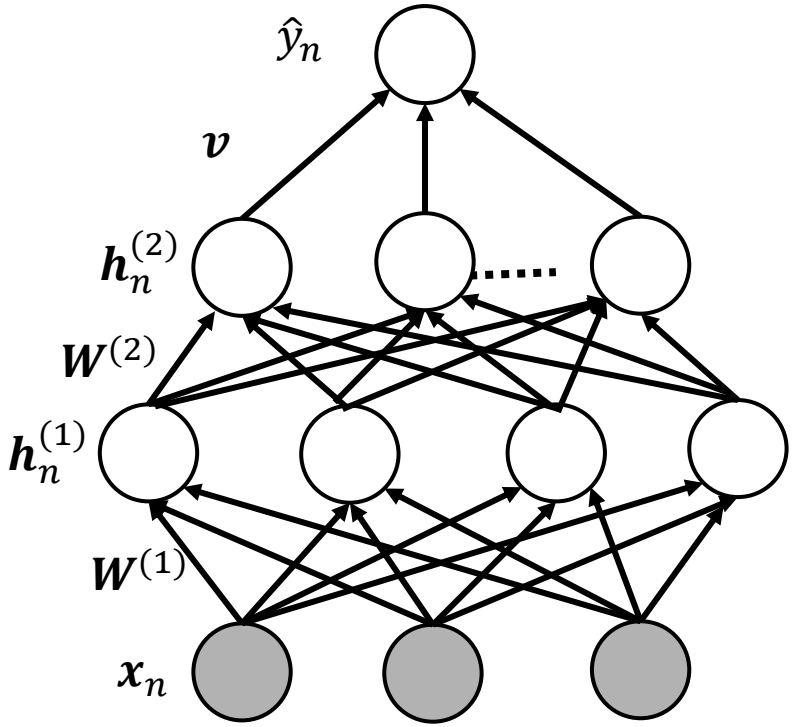
Note: Nonlinearity g is applied element-wise on its inputs so $h_n^{(\ell)}$ has the same size as vector $W^{(\ell)} h_n^{(\ell-1)}$

The Bias Term

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

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

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



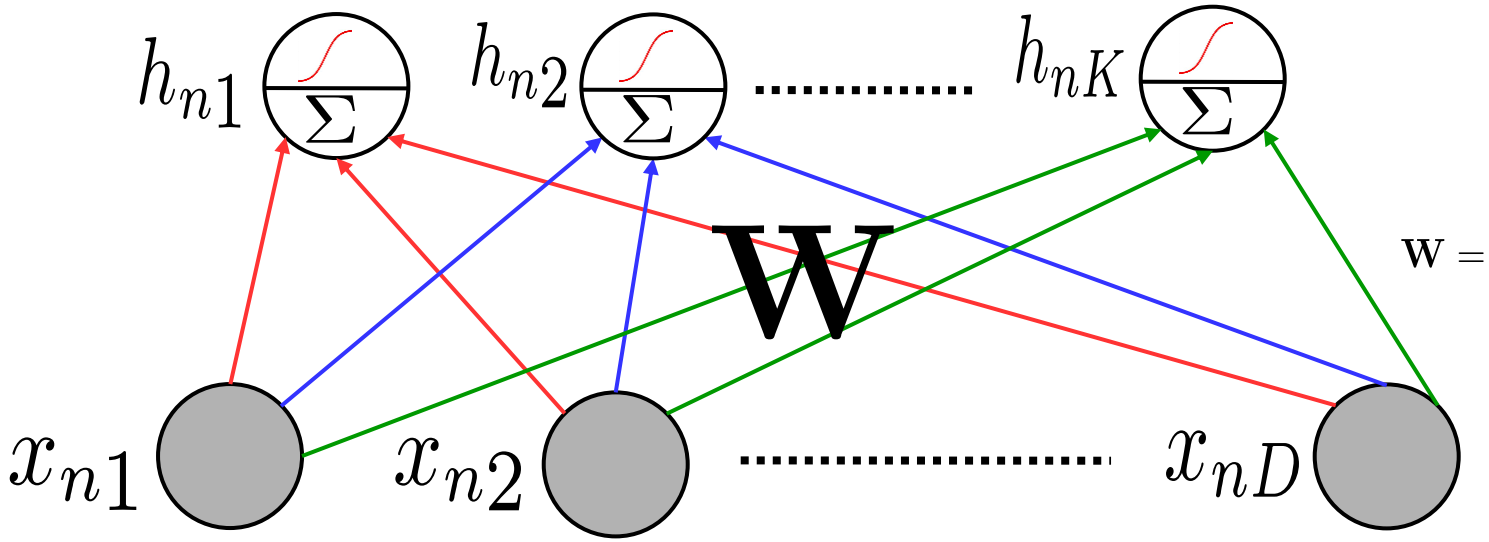
- Bias term increases the expressiveness of the network and ensures that we have nonzero activations/pre-activations even if this layer's input is a vector of all zeros

- 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

- The neural network weights can be seen as as feature learners



$$\mathbf{W} = \begin{bmatrix} w_{11} & \cdots & w_{1K} \\ \vdots & \ddots & \vdots \\ w_{D1} & \cdots & w_{DK} \end{bmatrix} = [\boldsymbol{w_1}, \boldsymbol{w_2}, \boldsymbol{w_3}, \ldots, \boldsymbol{w_K}]$$

- The K weight vectors $\boldsymbol{w_1}, \boldsymbol{w_2}, \ldots, \boldsymbol{w_K}$ (hidden layer wts) are like K feature detectors
- The vector $\boldsymbol{w_k}$ detects how much feature $\boldsymbol{h_{nk}}$ is present in the input $\boldsymbol{x_n}$

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

Dot product high means the feature is prominent in $\boldsymbol{x_n}$

Also note that the nonlinear g is monotonic

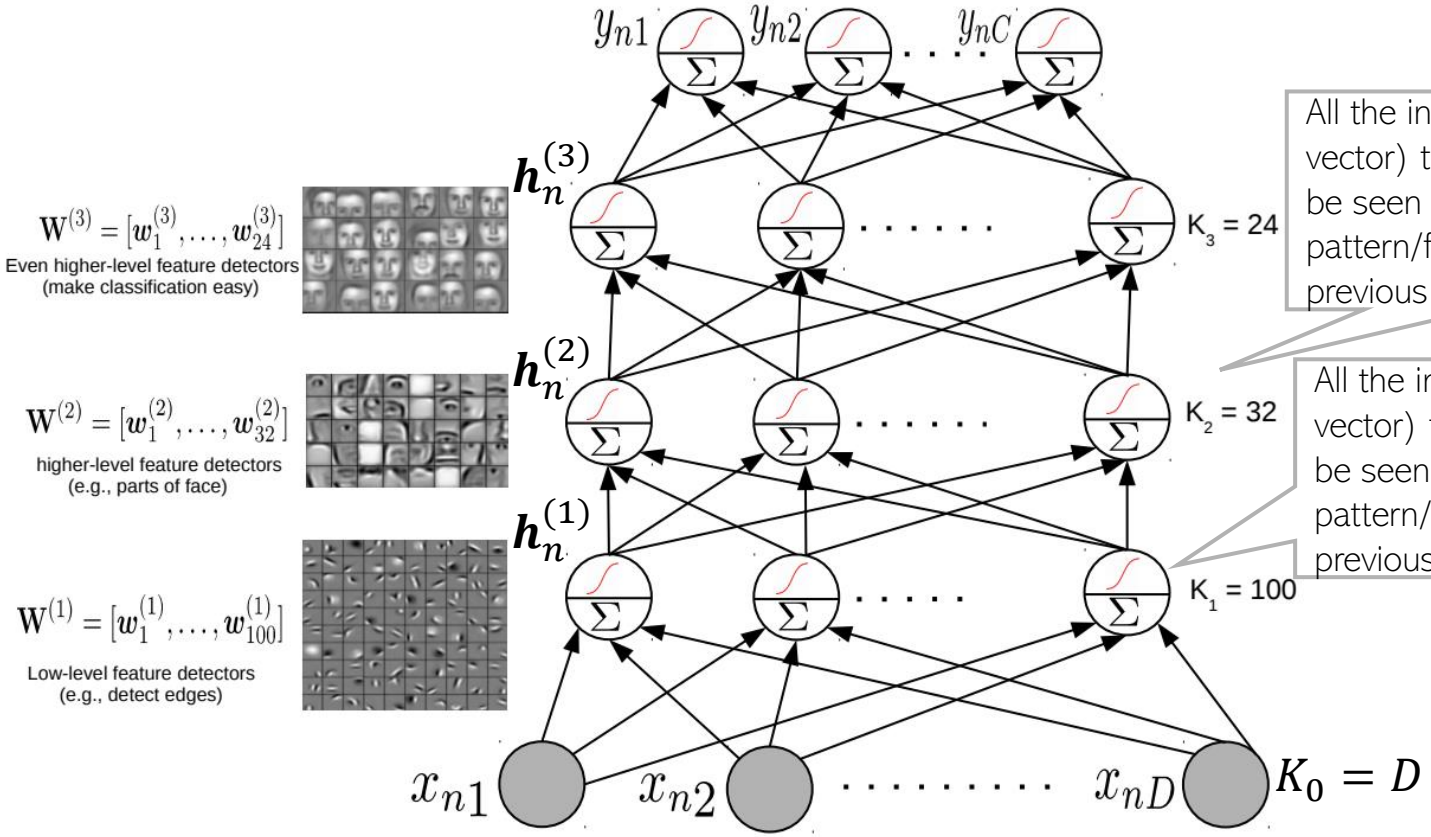
- Each $\boldsymbol{w_k}$ can also be thought of as a "filter" (a terminology used in CNNs)

Neural Nets are Feature Learners

$\boldsymbol{W}^{(\ell)} = [\boldsymbol{w}_1^{(\ell)}, \boldsymbol{w}_2^{(\ell)}, \ldots, \boldsymbol{w}_{K_\ell}^{(\ell)}]$ 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 k of layer ℓ
- For input $\boldsymbol{x}_n$, $h_{nk}^{(\ell)} = g(\boldsymbol{w}_k^{(\ell)\top} \boldsymbol{h}_n^{(\ell-1)})$ is the value of feature k in layer ℓ



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

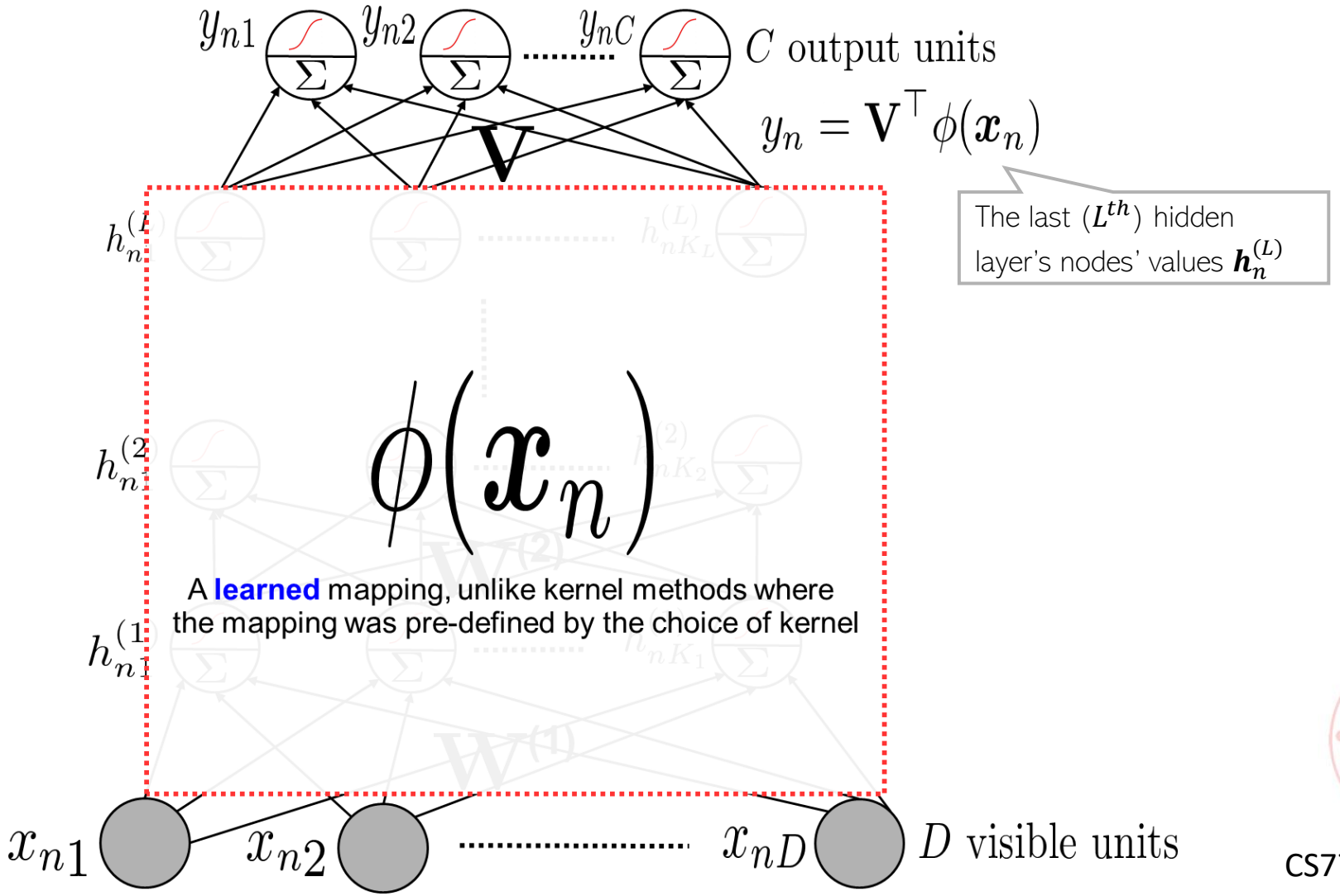
E.g., $\boldsymbol{w}_{32}^{(2)}$ denotes a feature detector

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

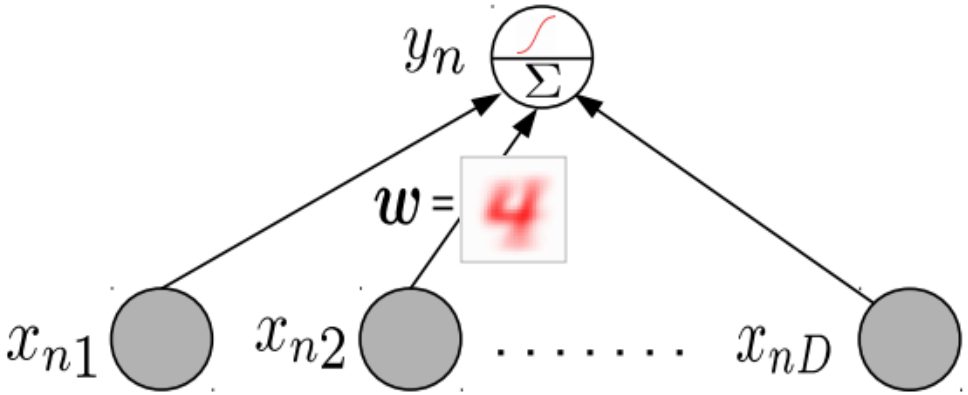
Neural Nets are Feature Learners

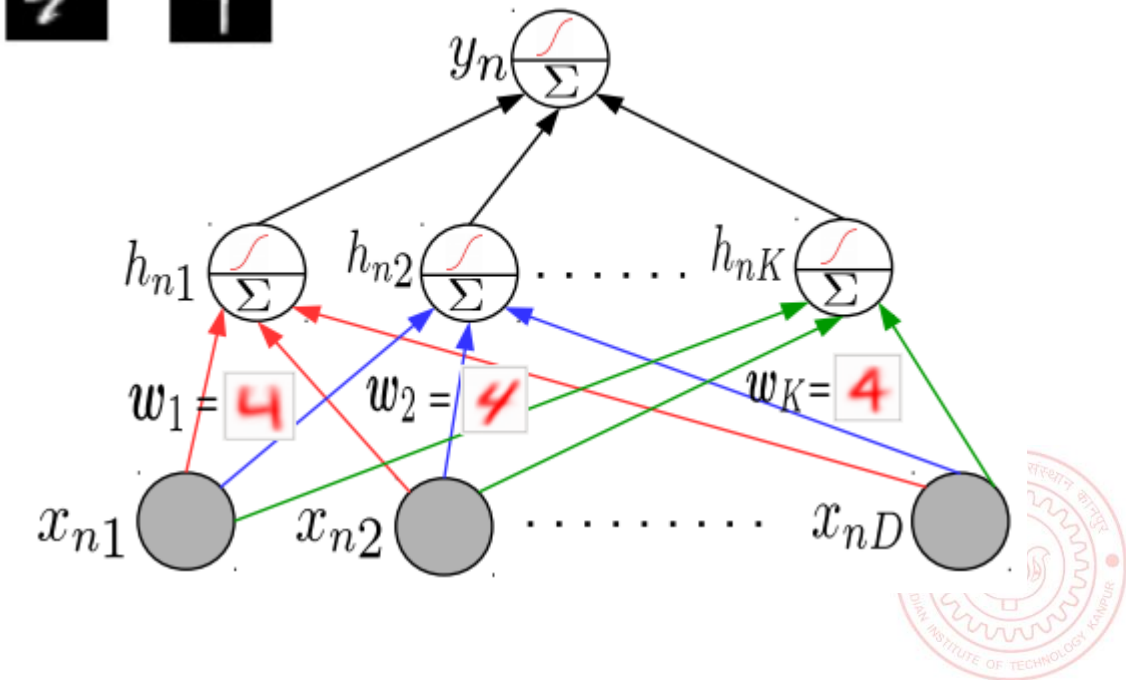
- Multiple hidden layers can be seen as collectively learning a feature mapping $\boldsymbol{\phi}$



Why Neural Networks Work Better: Another View

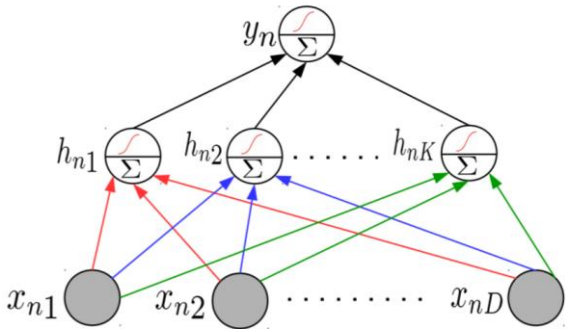
- Linear models tend to only learn the "average" pattern
  - E.g., Weight vector of a linear classification model represent average pattern of a class
- Deep models can learn multiple patterns (each hidden node can learn one pattern)
  - Thus deep models can learn to capture more subtle variations that a simpler linear model

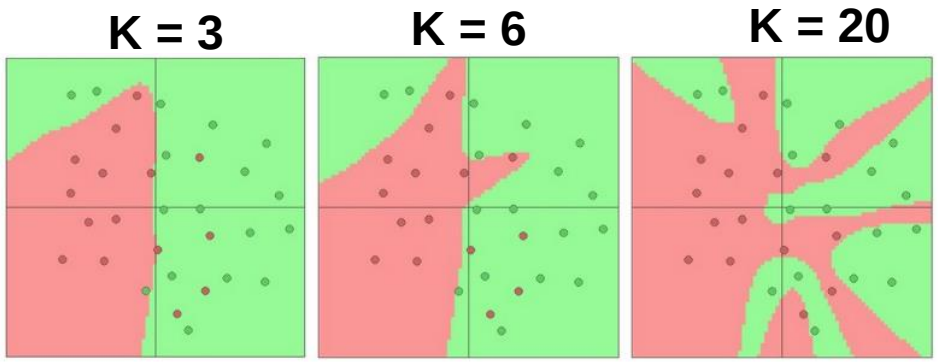




Representational Power of Neural Nets

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

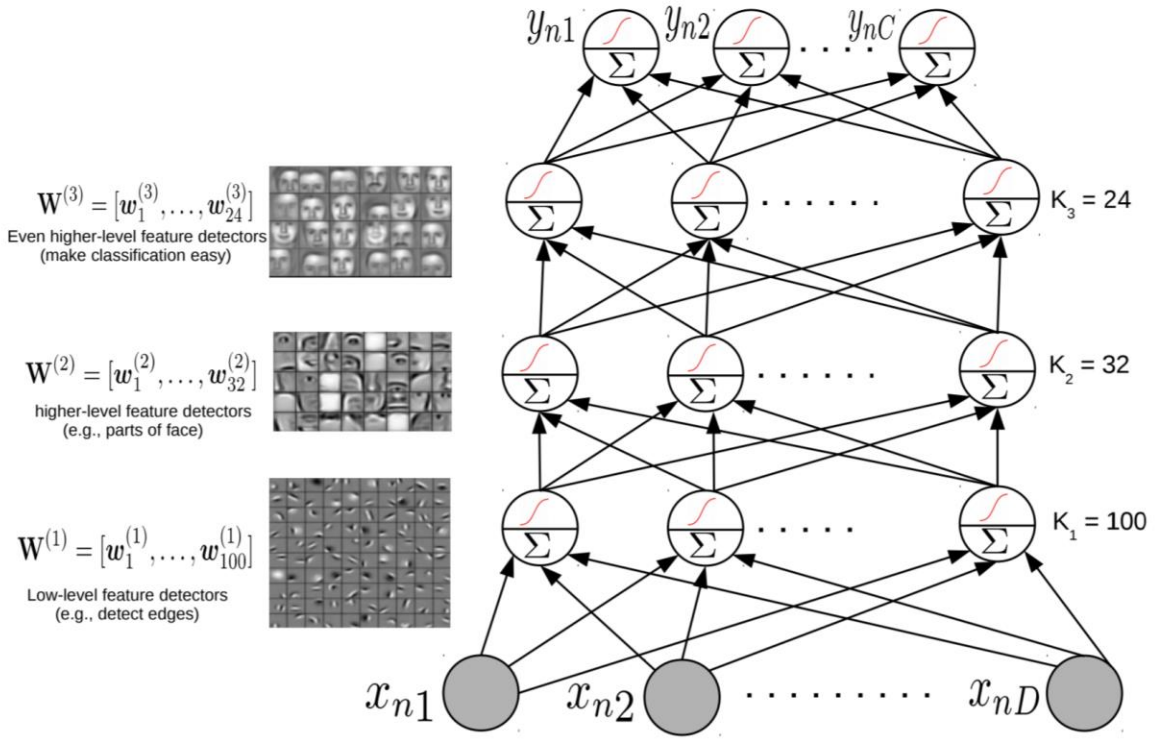




- Recall that each hidden unit "adds" a simple function to the overall function
- Increasing K (number of hidden units) will result in a more complex function
- Very large K seems to overfit (see above fig). Should we instead prefer small K?
- No! It is better to use large K and regularize well. Reason/justification: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

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)

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

- 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 ☺

Next..

- Training deep neural nets using backpropagation
- Some important aspects related to training of deep neural nets
  - Vanishing/exploding gradients
  - Initialization
  - Normalization layers (batch and layer normalization)
  - Dropout as a means to regularization
  - Residual/skip connections

Background: Gradient and Jacobian

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

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

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

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

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

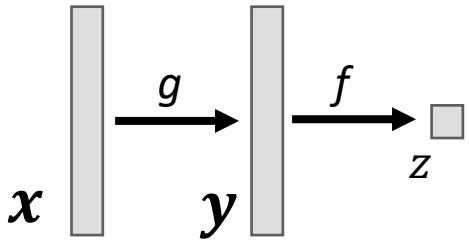
For a function $\boldsymbol{f}: \mathbb{R}^P \rightarrow \mathbb{R}^Q$, the Jacobian $J^f \in \mathbb{R}^{Q \times P}$ (a matrix)

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

More generally, for a function $\boldsymbol{f}: \mathbb{R}^{I_1 \times I_2 \times \cdots} \rightarrow \mathbb{R}^{O_1 \times O_2 \times \cdots}$, 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

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

Scalar derivative

$1 \times P$ vector derivative

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 ☺

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

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

$Q \times P$ Jacobian of 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! ☺)

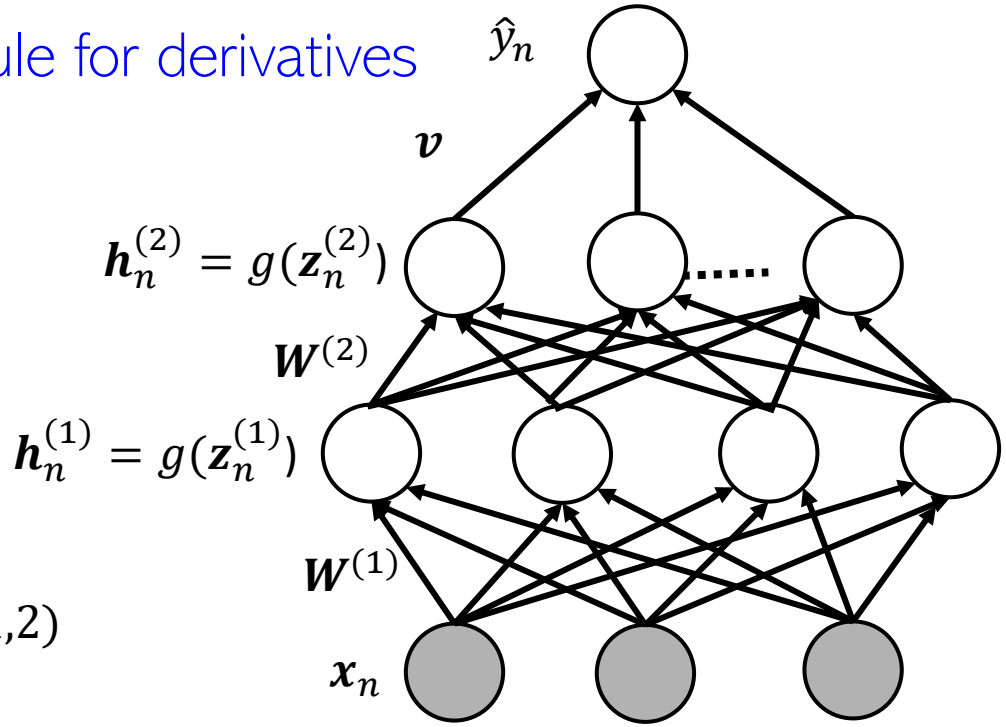
$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{w}} = J^f \times J^g \times J^h \in \mathbb{R}^{S \times P}$$

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. $\hat{y}_n$

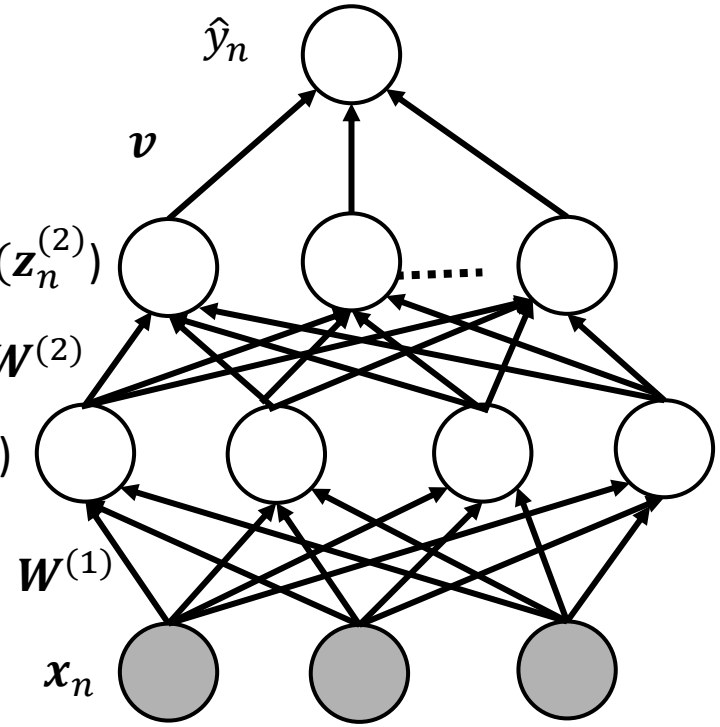
$$\frac{\partial \ell_n}{\partial \boldsymbol{v}} = \frac{\partial \ell_n}{\partial \hat{y}_n} \frac{\partial \hat{y}_n}{\partial \boldsymbol{v}} = \ell'(y_n, \hat{y}_n) \boldsymbol{h}_n^{(2)}$$

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

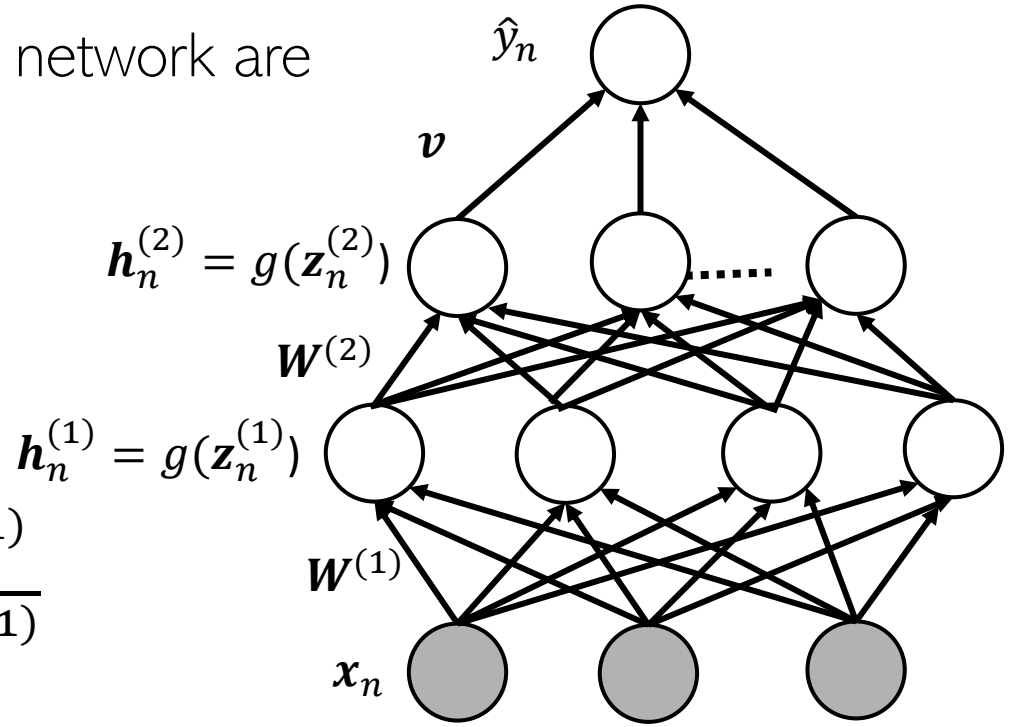
Backpropagation: Computation Reuse

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

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

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

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

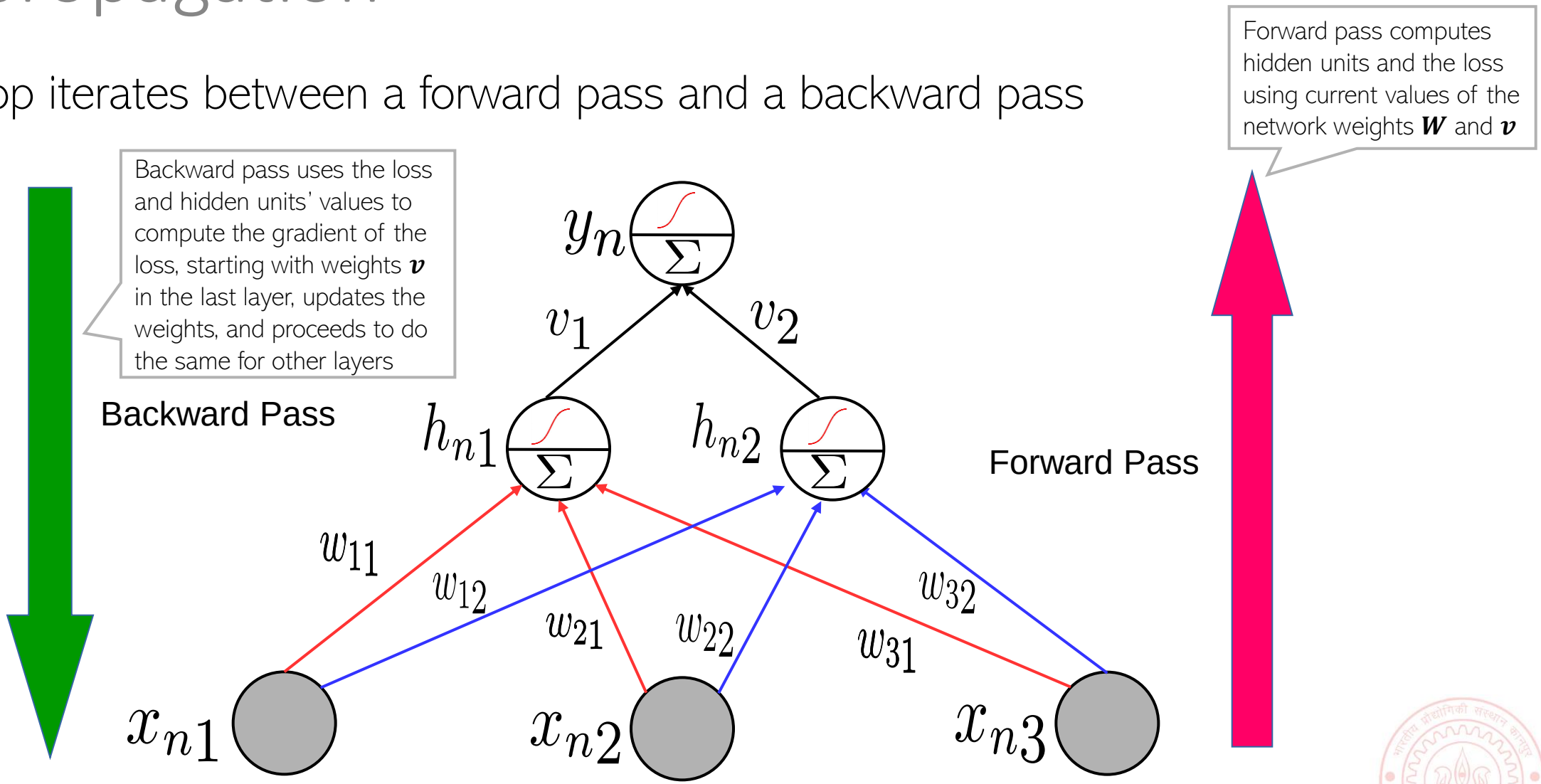


- Thus gradient computations done in upper layers can be stored and reused when computing the gradients in the lower layers (libraries like Tensorflow and Pytorch do so efficiently)
- Vanishing gradients: $\frac{\partial \boldsymbol{h}_n^{(i)}}{\partial \boldsymbol{z}_n^{(i)}} = \operatorname{diag}\left(g'\left(z_{n1}^{(i)}\right),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

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

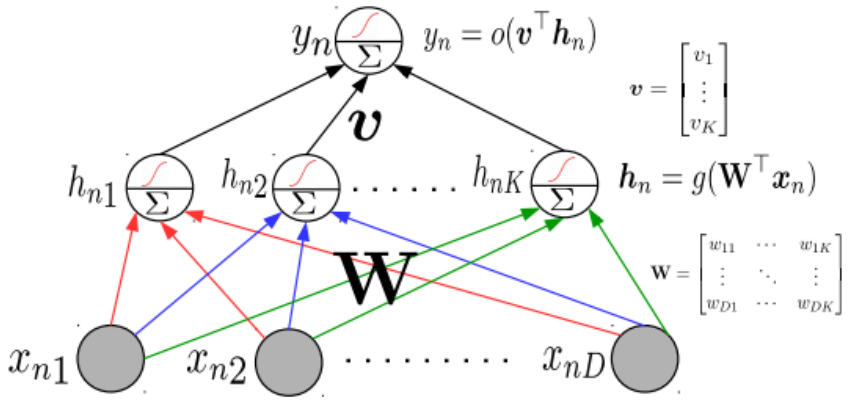
- Backprop iterates between a forward pass and a backward pass



- 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

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

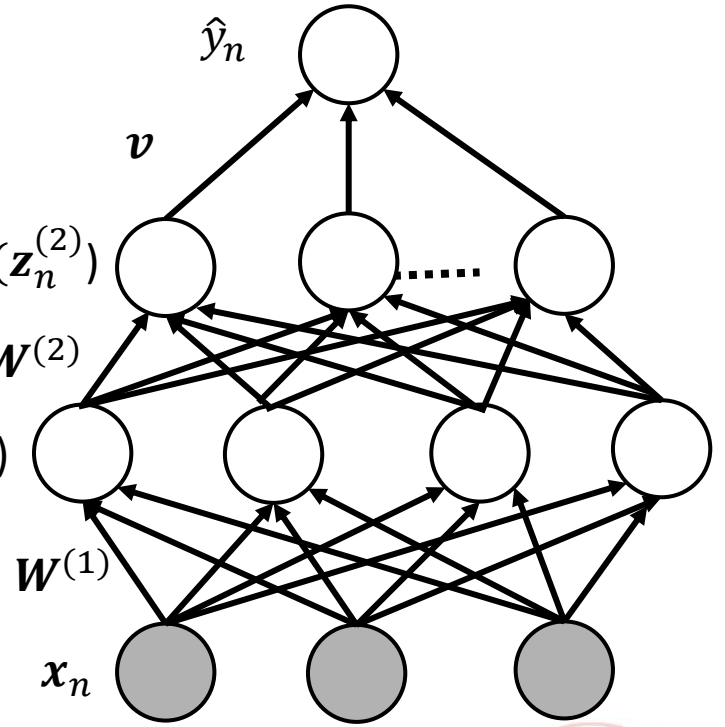
$$\frac{\partial \mathcal{L}}{\partial h_{nk}} = -(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n))v_k = -e_n v_k$$

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

- Forward prop computes errors 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

Problem of Exploding/Vanishing Gradients

- MLPs/CNNs have many hidden layers and gradients in each layer are a product of several Jacobians
- Result of these products depends on the eigenvalues of each of these Jacobians
  - If they are large (>1), gradients might blow up (explode)
  - If they are small (<1), gradients might vanish
- To prevent blow up, we can use gradient clipping
  - Simply cap the magnitude of the gradients!
- To prevent vanishing gradients, several options
  - Use non-saturating activation functions (recall that the gradient is a product of terms like $\frac{\partial \boldsymbol{h}_n^{(i)}}{\partial \boldsymbol{z}_n^{(i)}} = \text{diag}\left(g'\left(z_{n1}^{(i)}\right), \ldots, g'\left(z_{nK_i}^{(i)}\right)\right)$, so the derivative $\boldsymbol{g'}$ doesn't become too small
  - Use other architectures such as skip- connections (will discuss later)



Training of DNNs: Some Important Aspects

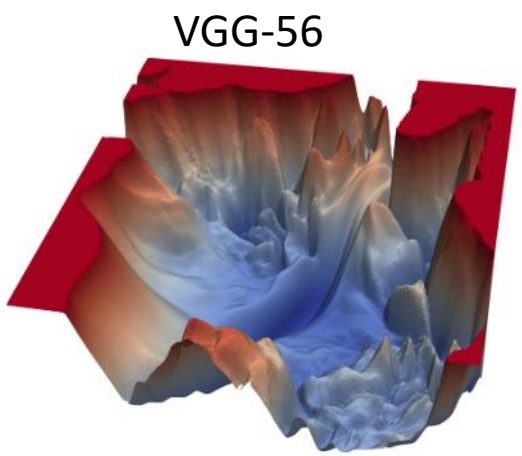
- Deep neural net training can be hard due to non-convex loss functions
- Several ways to address this, e.g.,
  - Good choice of learning rate of (S)GD
    - We have already seen this
  - Good initialization of parameters, e.g., initialize each weight, say w_{ij}, randomly as

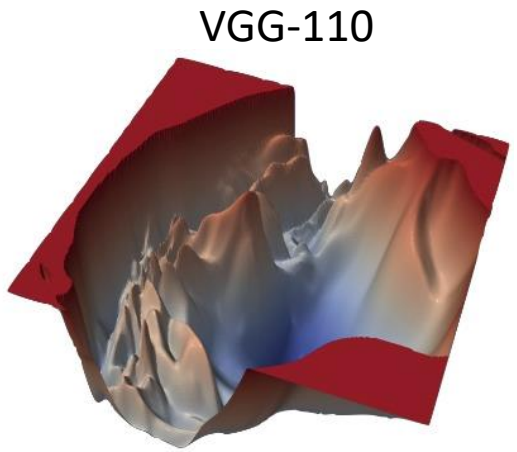
$$w_{ij} \sim \mathcal{N}(0, \sigma^2) \quad \text{or} \quad w_{ij} \sim \text{Uniform}(-a, a)$$

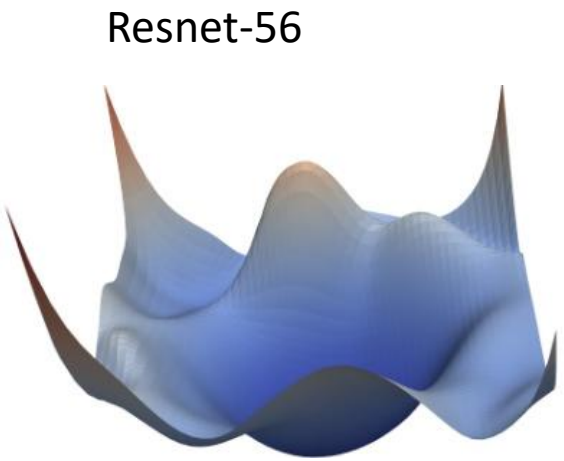
Xavier/Gloret initialization, LeCun init, He init, etc

and set the "spread" of these distribution as inversely proportional to $n_{\text{in}} + n_{\text{out}}$

  - Careful design of the network architecture, e.g.,
    - Networks with "skip connections" (will see later) which lead to less non-convex (more smooth) loss surfaces (figures on the right)
- Vanishing/exploding gradients (already saw)







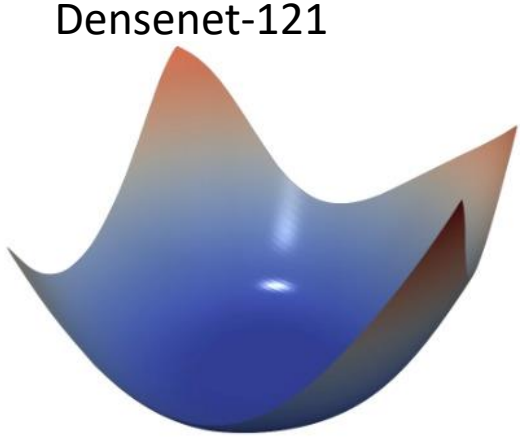


Fig credit: https://www.cs.umd.edu/~tomg/projects/landscapes/

Normalization Layer

Note: Batch-norm assumes sufficiently large mini-batch $\mathcal{B}$ to work well. There are variants such as "layer normalization" and "instance normalization" that don't require a mini-batch can be computed using a single training example

Batch normalization is used in MLP, CNN, and various other architectures

- Each hidden layer is a nonlinear transformation of the previous layer's inputs
- To prevent distribution drift in activations' distribution, we often "standardize" each layer
- Standardize = activation $\boldsymbol{h}_{nk}^{(\ell)}$ should have zero mean and unit variance across all $\boldsymbol{n}$
- It is achieved by inserting a "batch normalization" layer after each hidden layer
- To do so, during training, (omitting layer number ℓ) we replace each $\boldsymbol{h}_n$ by $\widetilde{\boldsymbol{h}}_n$

$\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are trainable batch-norm parameters

$$\widetilde{\boldsymbol{h}}_n = \boldsymbol{\gamma} \odot \widehat{\boldsymbol{h}}_n + \boldsymbol{\beta} \qquad \widehat{\boldsymbol{h}}_n = \frac{\boldsymbol{h}_n - \boldsymbol{\mu}_{\mathcal{B}}}{\sqrt{\boldsymbol{\sigma}_{\mathcal{B}}^2 + \epsilon}}$$

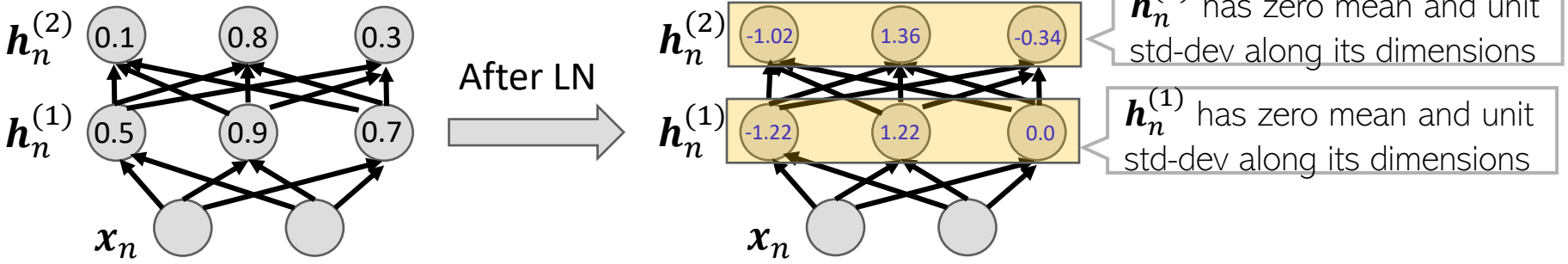
We compute $\boldsymbol{\mu}_{\mathcal{B}}$ and $\boldsymbol{\sigma}_{\mathcal{B}}^2$ using the data from the current minibatch of examples $\mathcal{B}$ (thus the name "batch norm"

$$\boldsymbol{\mu}_{\mathcal{B}} = \frac{1}{|\mathcal{B}|}\sum_{\boldsymbol{h}\in\mathcal{B}} \boldsymbol{h} \qquad \boldsymbol{\sigma}_{\mathcal{B}}^2 = \frac{1}{|\mathcal{B}|}\sum_{\boldsymbol{h}\in\mathcal{B}} (\boldsymbol{h} - \boldsymbol{\mu}_{\mathcal{B}})^2$$

- After training, we store $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ + the statistics $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2$ computed on the whole training data, and use these values to apply batch-norm on each test input

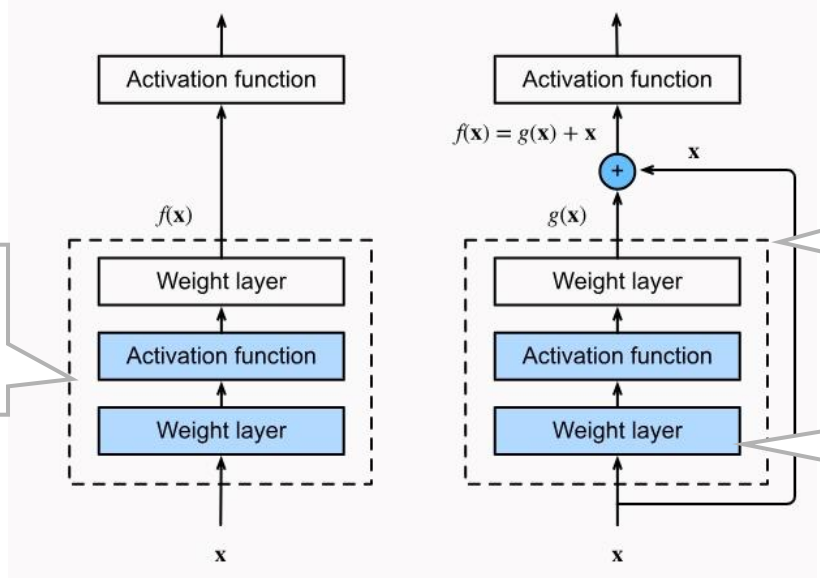
Layer Normalization

- Normalization helps improve training and performance overall
- Unlike batch normalization (BN), which we already saw, layer normalization (LN) normalizes each $\boldsymbol{h_n}$ across its dimensions (not across all minibatch examples)
  - Often used for sequence data models (will see later) where BN is difficult to apply
  - Also useful when batch sizes are small where BN statistics (mean/var) aren't reliable
- For an MLP, the LN operation would look like this



Residual/Skip Connections

- Many modern deep nets contain a very large number of layers
- In general, just stacking lots of layer doesn't necessarily help a deep learning model
  - Vanishing/exploding gradient may make learning difficult
- Skip connections or "residual connections" help if we want very deep networks
  - This idea was popularized by "Residual Networks"* (ResNets) which can have hundreds of layers
- Basic idea: Don't force a layer to learn everything about a mapping



These layers trying to learn some function $f(x)$

May need to perform an additional projection/adjustment to that the sizes of x and $g(x)$ match

Added a "residual branch" or "short-cut" connection to connect x to the residual output $g(x)$ of these layers

Reducing their burden by just asking them to learn the "residual" $g(x) = f(x) - x$

*Deep Residual Learning for Image Recognition (He et al, 2015)

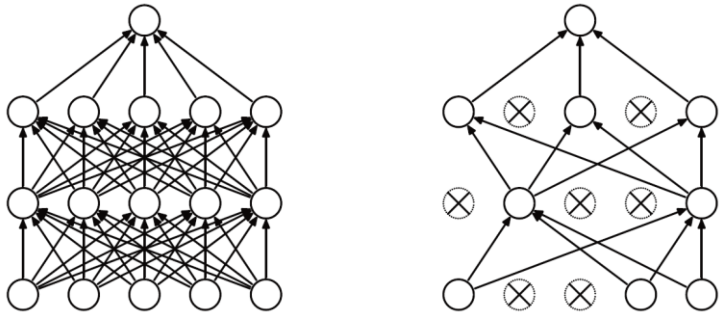
Pic source: https://www.d2l.ai/index.html

Dropout Layer

- Deep neural networks can overfit when trained on small datasets
- Dropout is a method to regularize without using an explicit regularizer
- In every update of the network, drop neuron i in layer ℓ with probability p

$$\epsilon_i^{(\ell)} \sim \text{Bernoulli}(1-p)$$

- If $\epsilon_i^{(\ell)} = 0$, set all outgoing weights $w_{ij}^{(\ell)}$ from neuron i to 0



- Each update of weights will change a different subset of weights
  - In doing so, we are making individual neurons more self-reliant and less dependent on others
- At test time, no dropout is used. After training is complete, we multiply each weight by the keep probability $1-p$ and use these weights for predictions