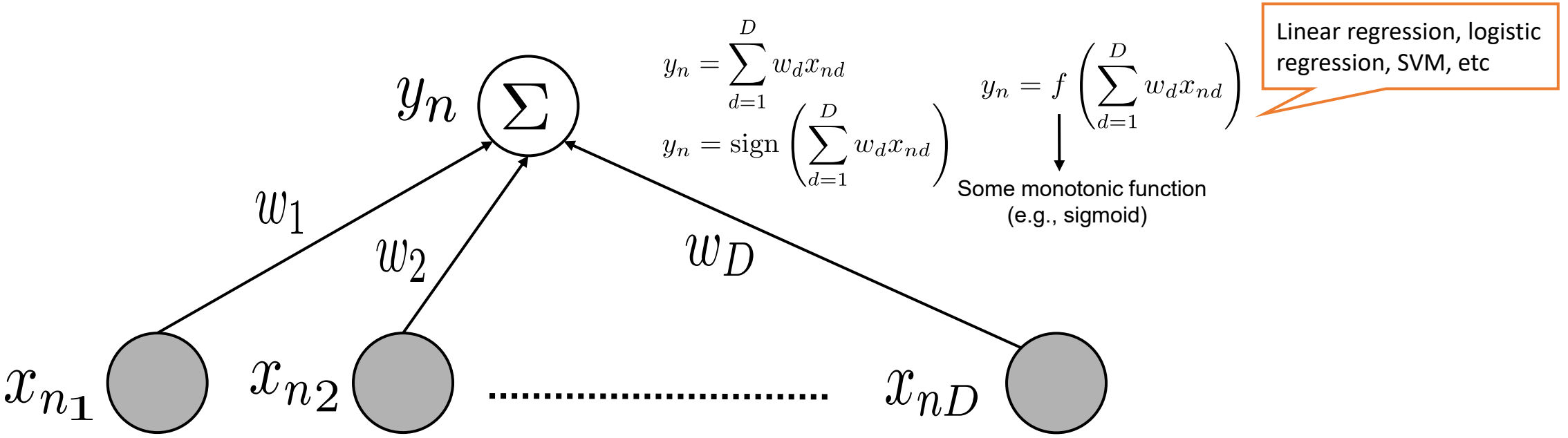
Introduction to Machine Learning

Deep Neural Networks: Multi Layer Perceptron

Prof. Subrahmanya Swamy Peruru
Department of Electrical Engineering
IIT Kanpur

Linear Models

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



- This basic architecture is classically also known as the “Perceptron”
- This can’t model nonlinear functions or decision boundaries

Lecture slides adapted from Prof. Piyush Rai

Neural Networks: Multi-layer Perceptron (MLP)

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

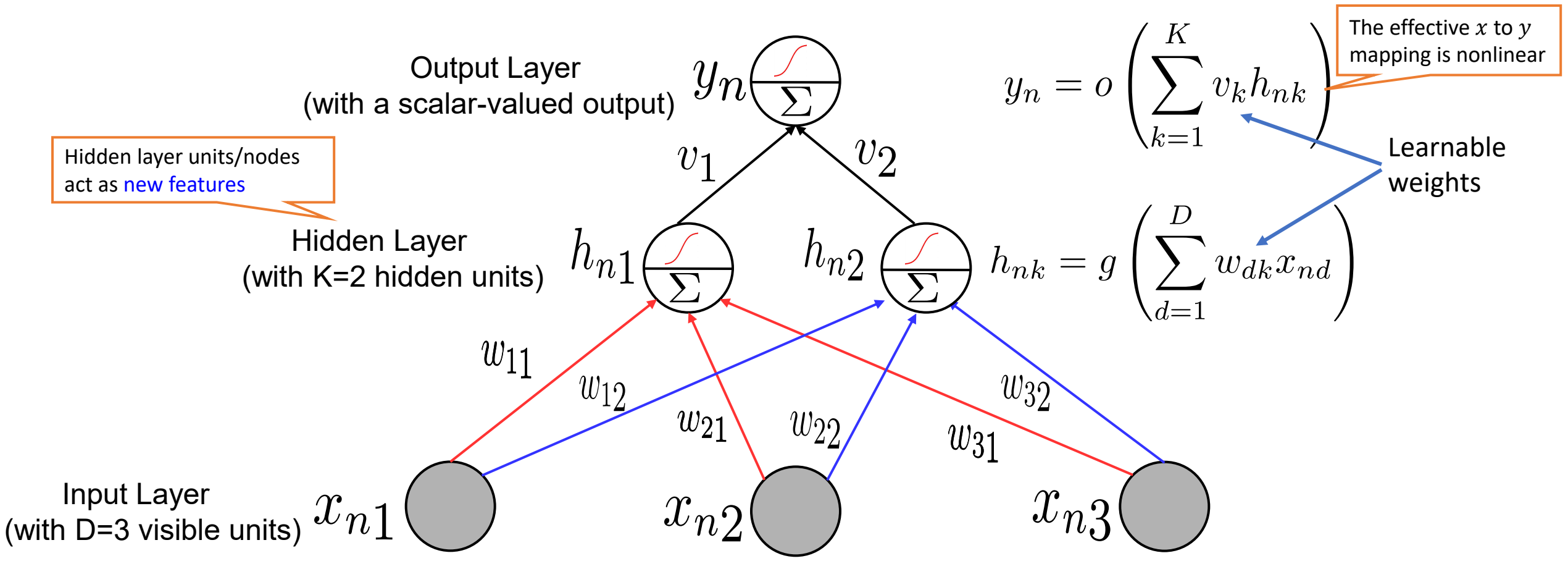


Illustration: Neural Net with One Hidden Layer

- Several linear combinations of the input features (called pre-activations)

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

- Nonlinear activation applied on each pre-act.

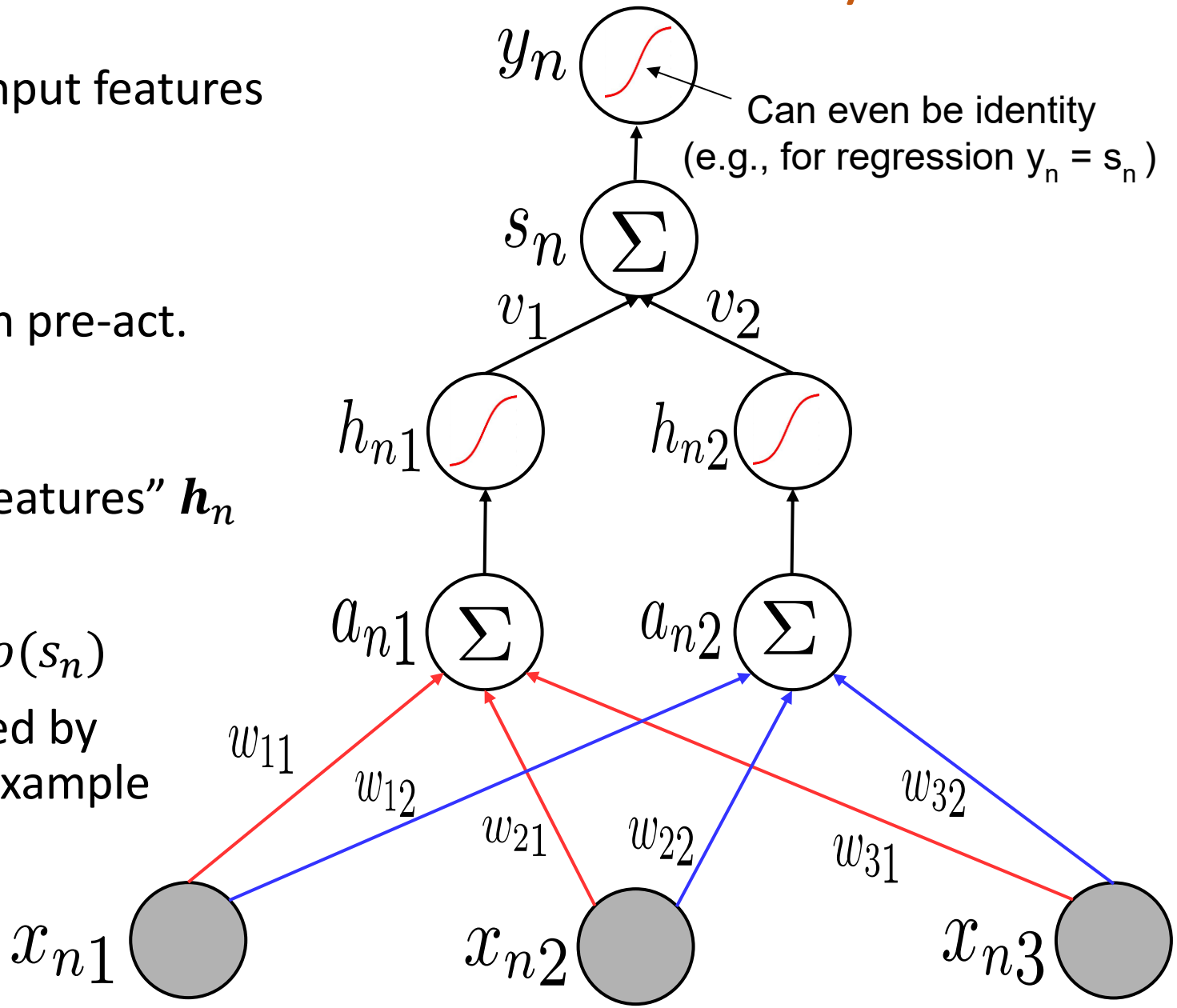
$$h_{nk} = g(a_{nk})$$

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

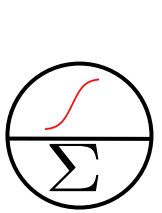
$$s_n = \boldsymbol{v}^\top \boldsymbol{h}_n = \sum_{k=1}^{K} v_k h_{nk}$$

- Finally, output is produced as $y = o(s_n)$
- Unknowns $(\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_K, \boldsymbol{v})$ 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, cross entropy, etc)

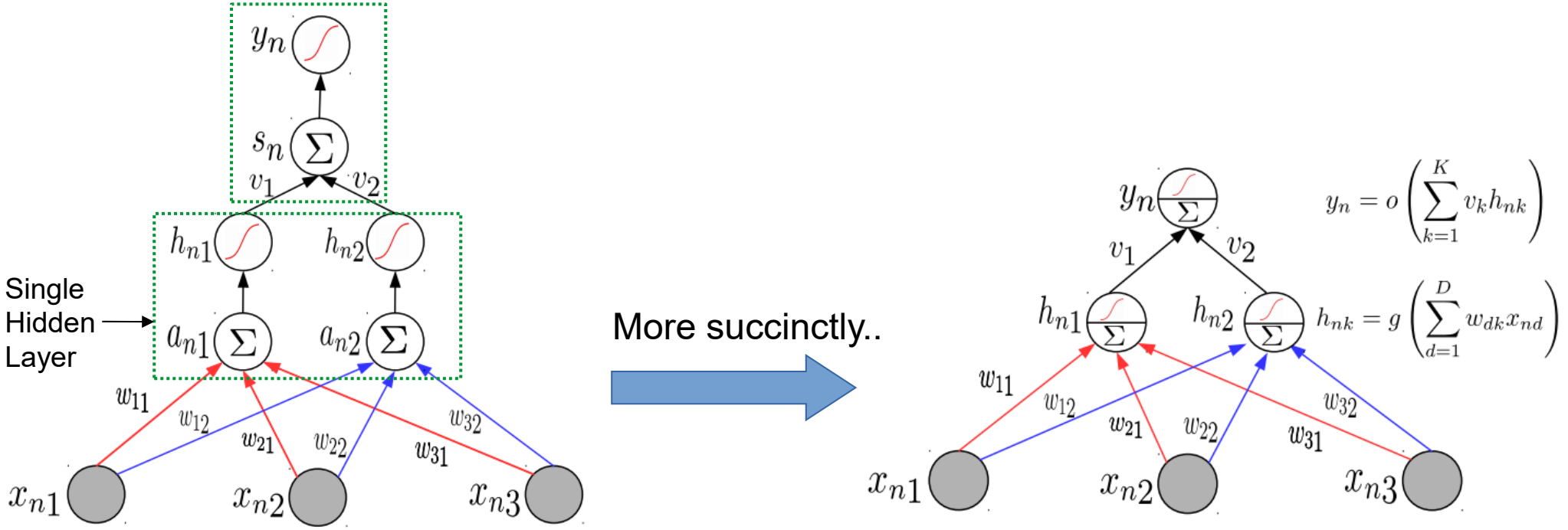


Neural Nets: A Compact Illustration

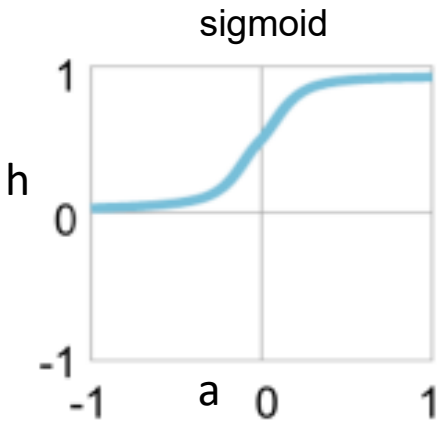


Will denote a linear combination of inputs followed by a nonlinear operation on the result

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

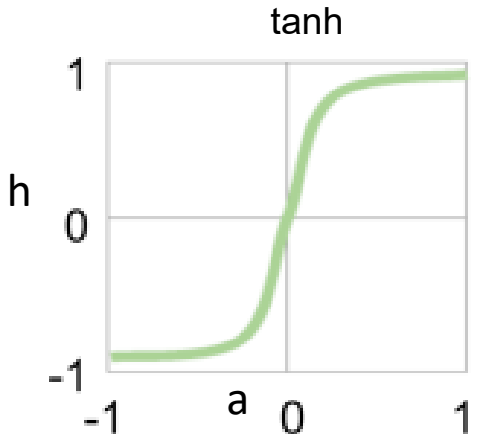


Activation Functions: Some Common Choices



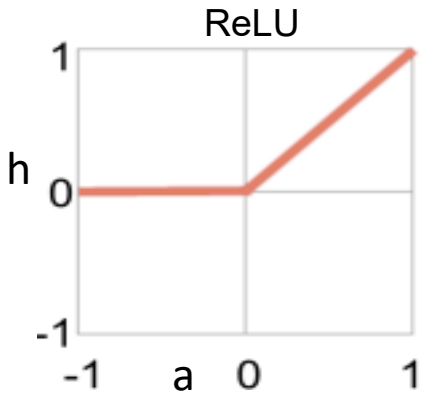
Sigmoid: $h = \sigma(a) = \frac{1}{1+\exp(-a)}$

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

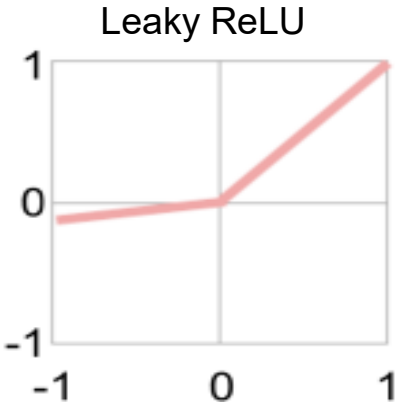


tanh (tan hyperbolic): $h = \frac{\exp(a)-\exp(-a)}{\exp(a)+\exp(-a)} = 2\sigma(2a) - 1$

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)



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



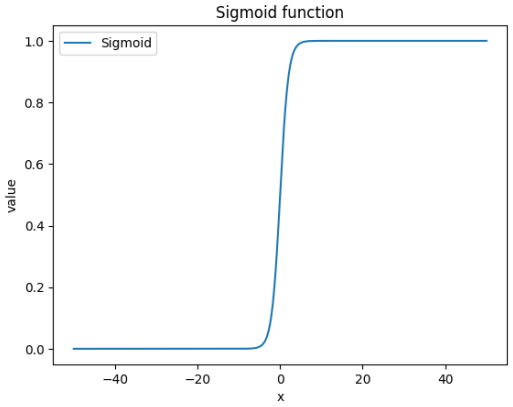
Leaky ReLU: $h = \max(\beta a, a)$
where β is a small postive number

ReLU and Leaky ReLU are among the most popular ones

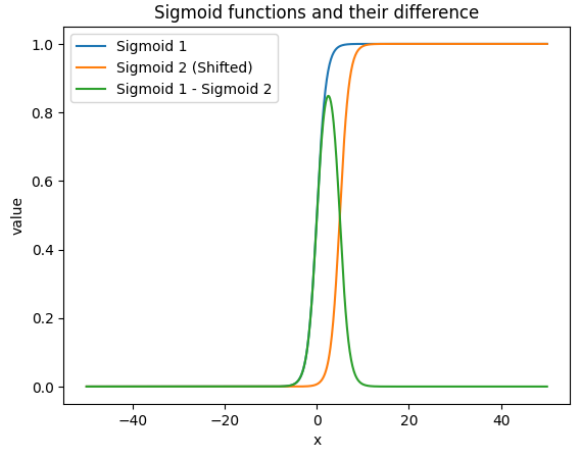
Note: Without nonlinear activation, a deep neural network is equivalent to a linear model no matter how many layers we use

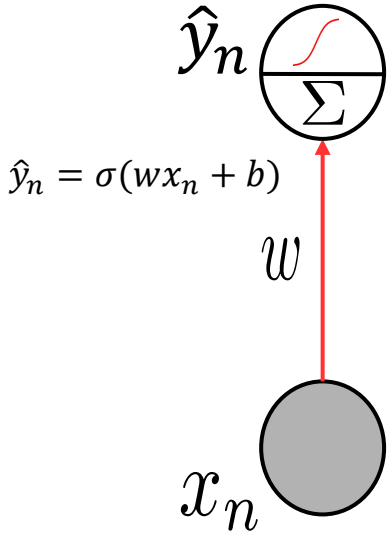
MLP Can Learn Any Nonlinear Function

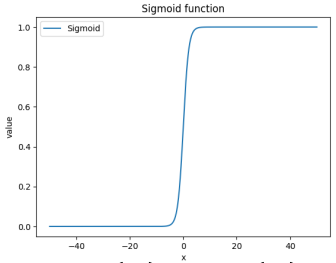
$$p(y = \text{red}|x) = h_{n1}(x) - h_{n2}(x)$$



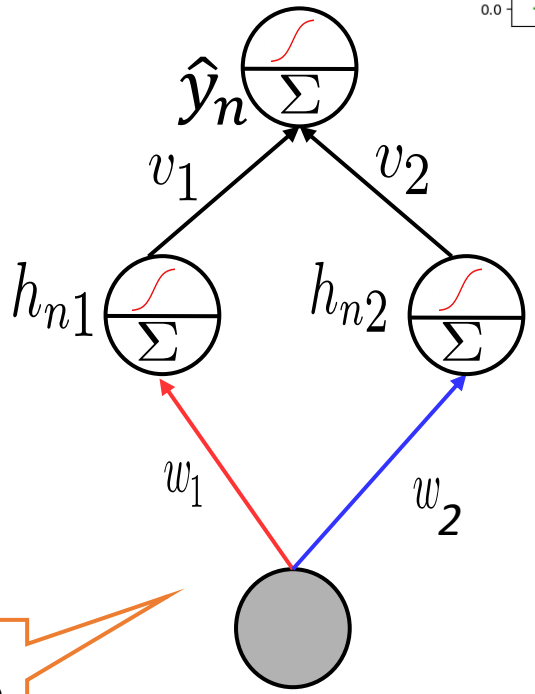
Nonlinear separation boundary

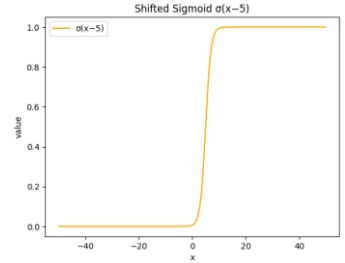






$h_{n1}(x) = \sigma(x)$





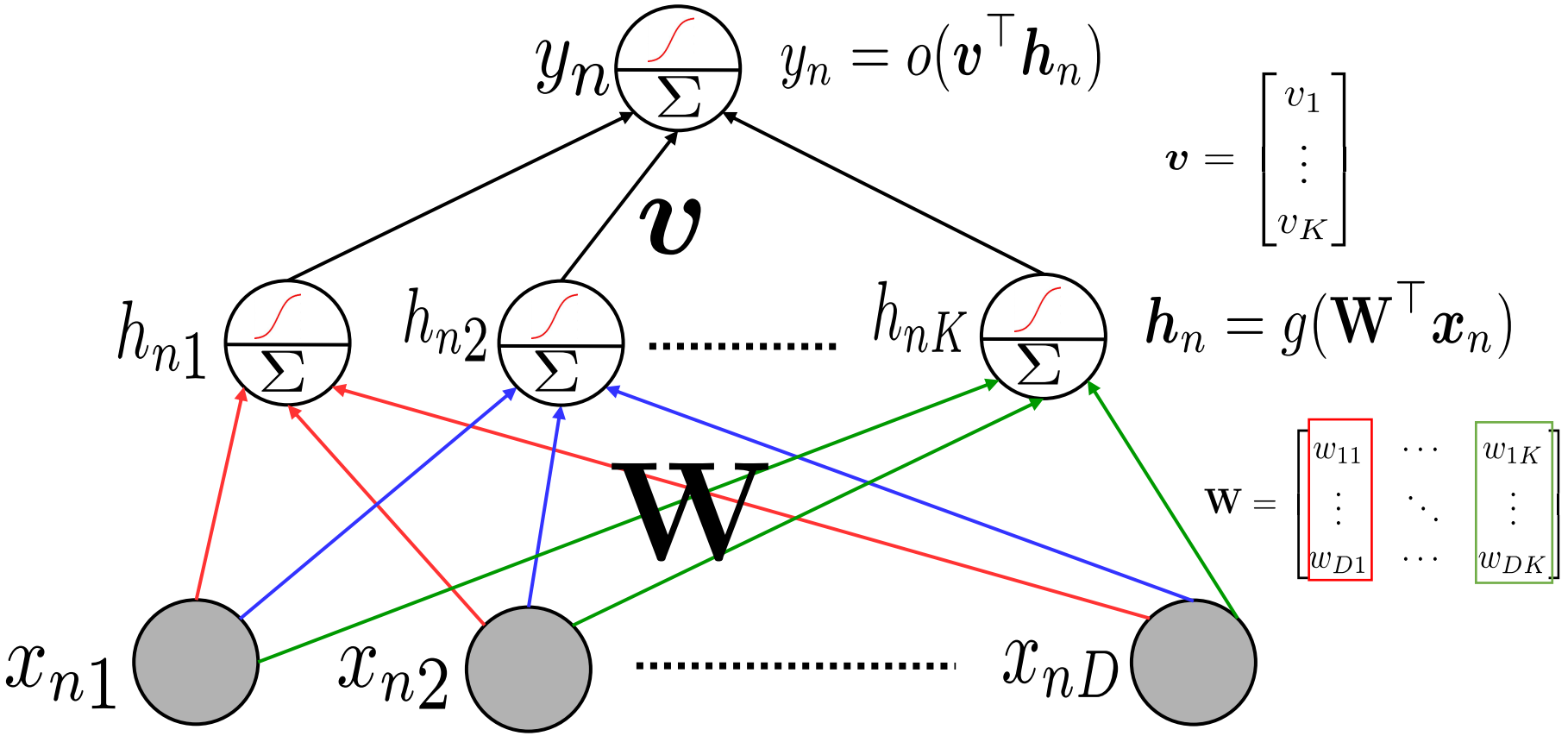
$h_{n2}(x) = \sigma(x - 5)$

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

Examples of some basic NN/MLP architectures

Single Hidden Layer and Single Outputs

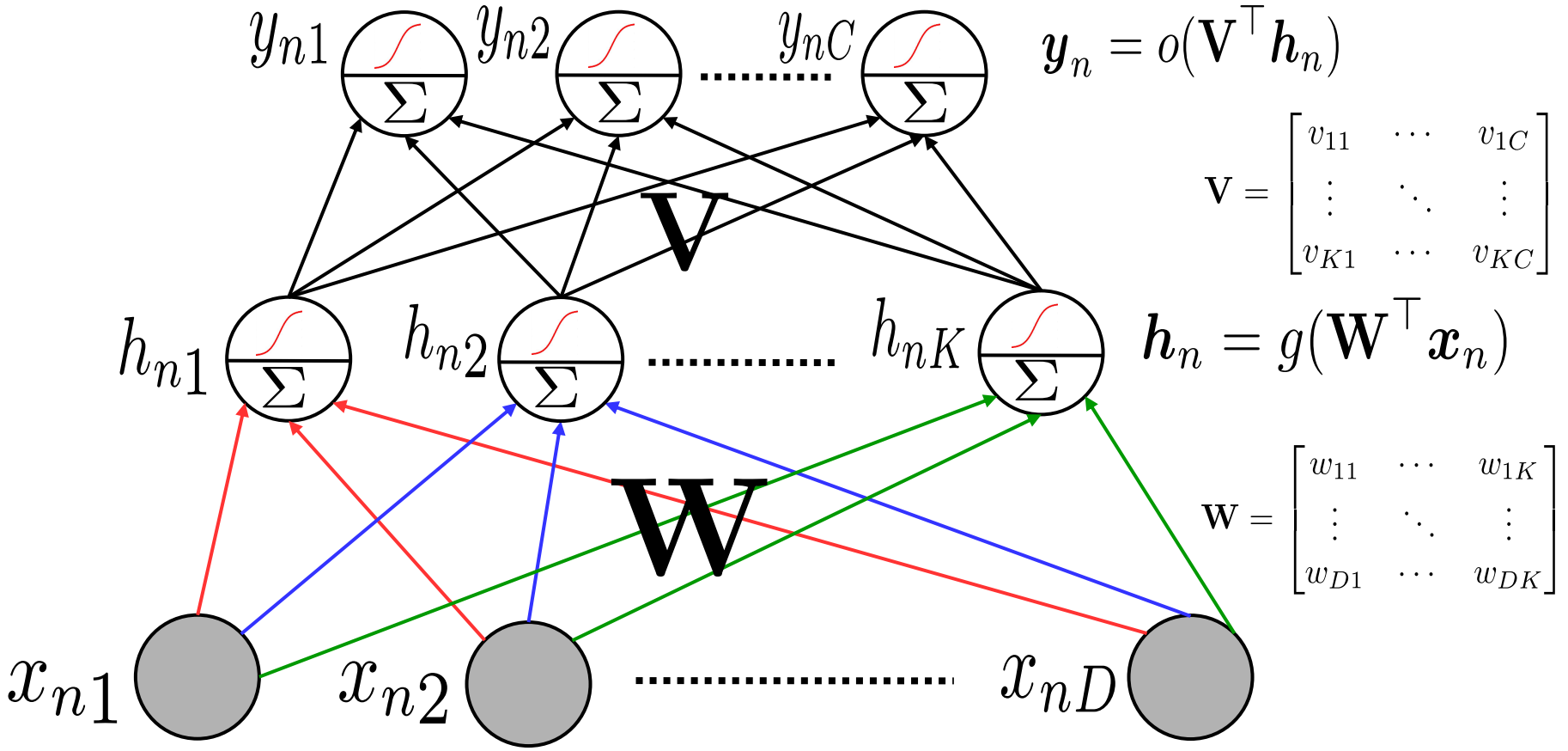
- One hidden layer with K nodes and a single output (e.g., scalar-valued regression or binary classification)



- Each column in W matrix correspond to one hidden unit

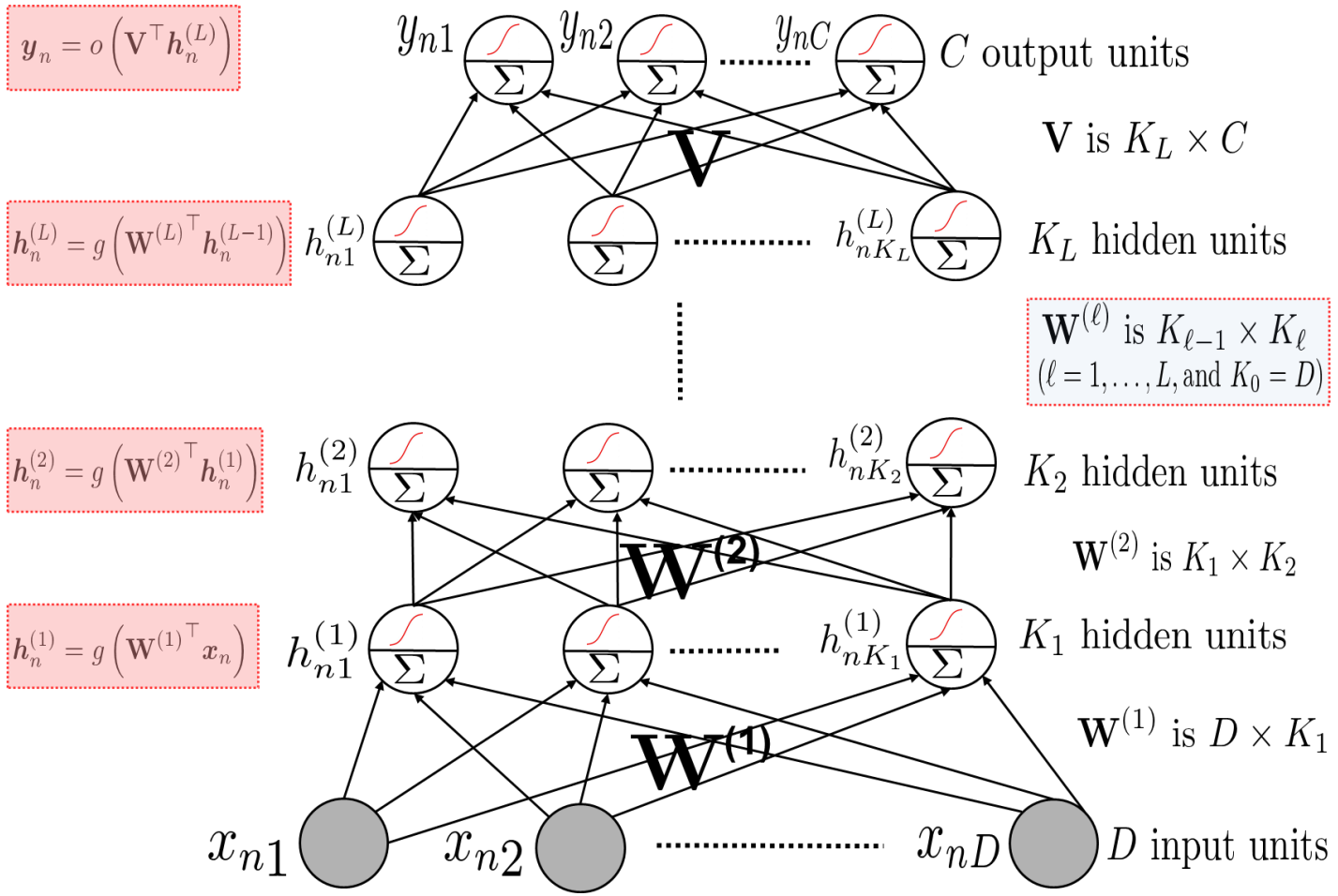
Single Hidden Layer and Multiple Outputs

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



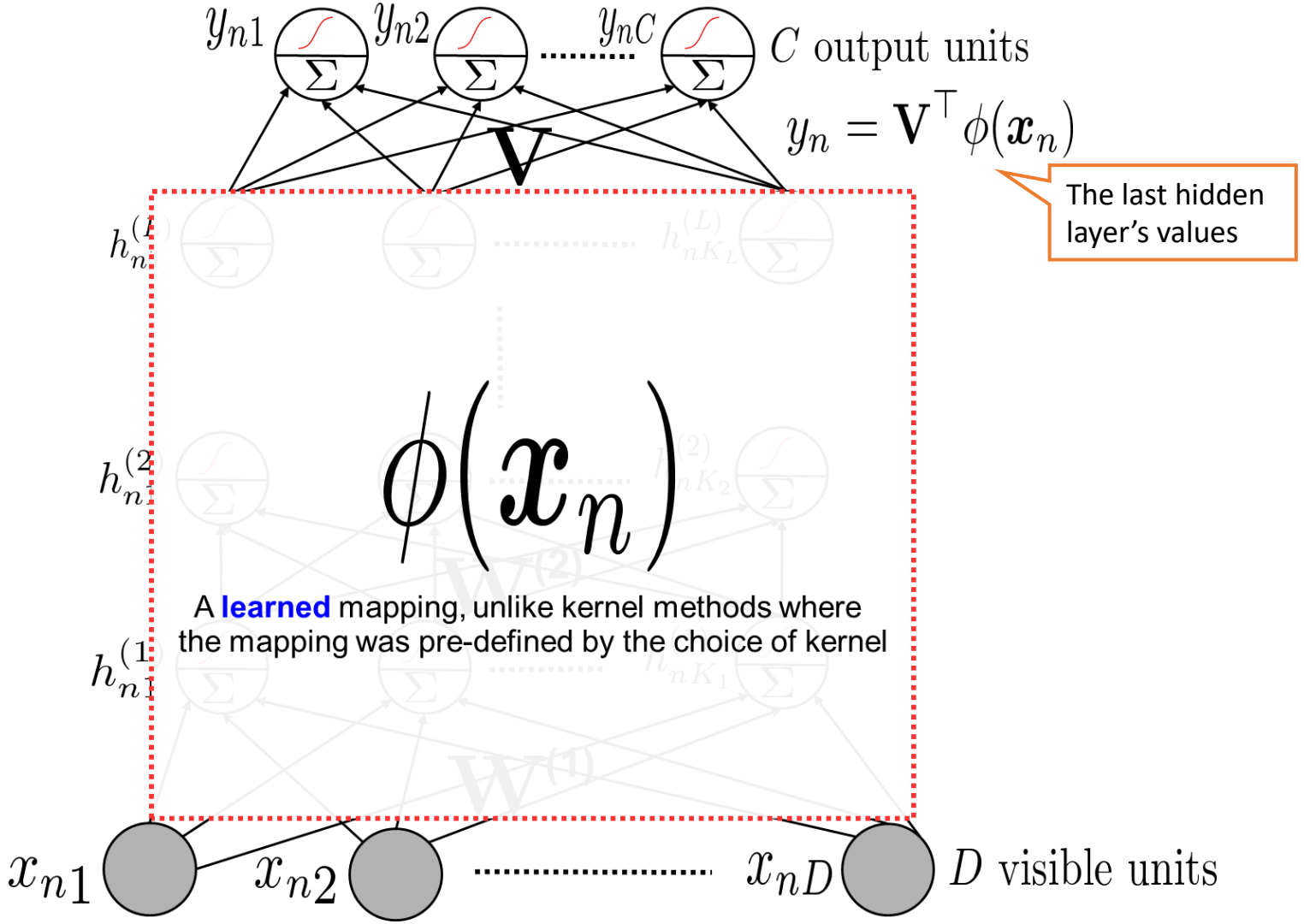
Multiple Hidden Layers (One/Multiple Outputs)

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



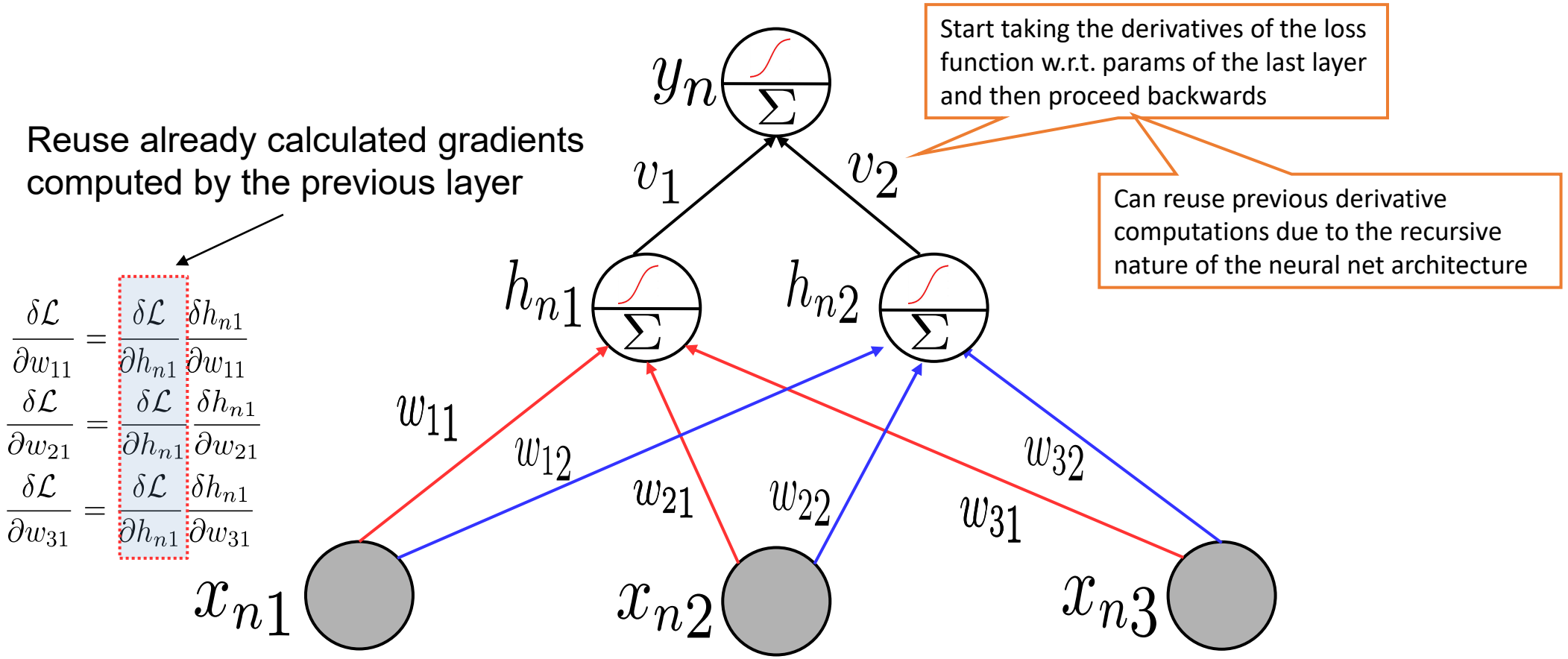
Neural Nets are Feature Learners

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



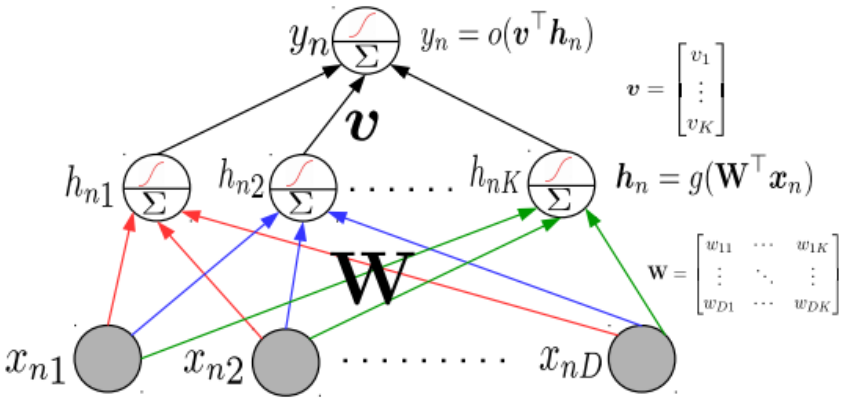
Backpropagation: To Learn the weights

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



Backpropagation through an example

Consider a single hidden layer MLP



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 $\mathbf{W}$, $\boldsymbol{v}$, 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

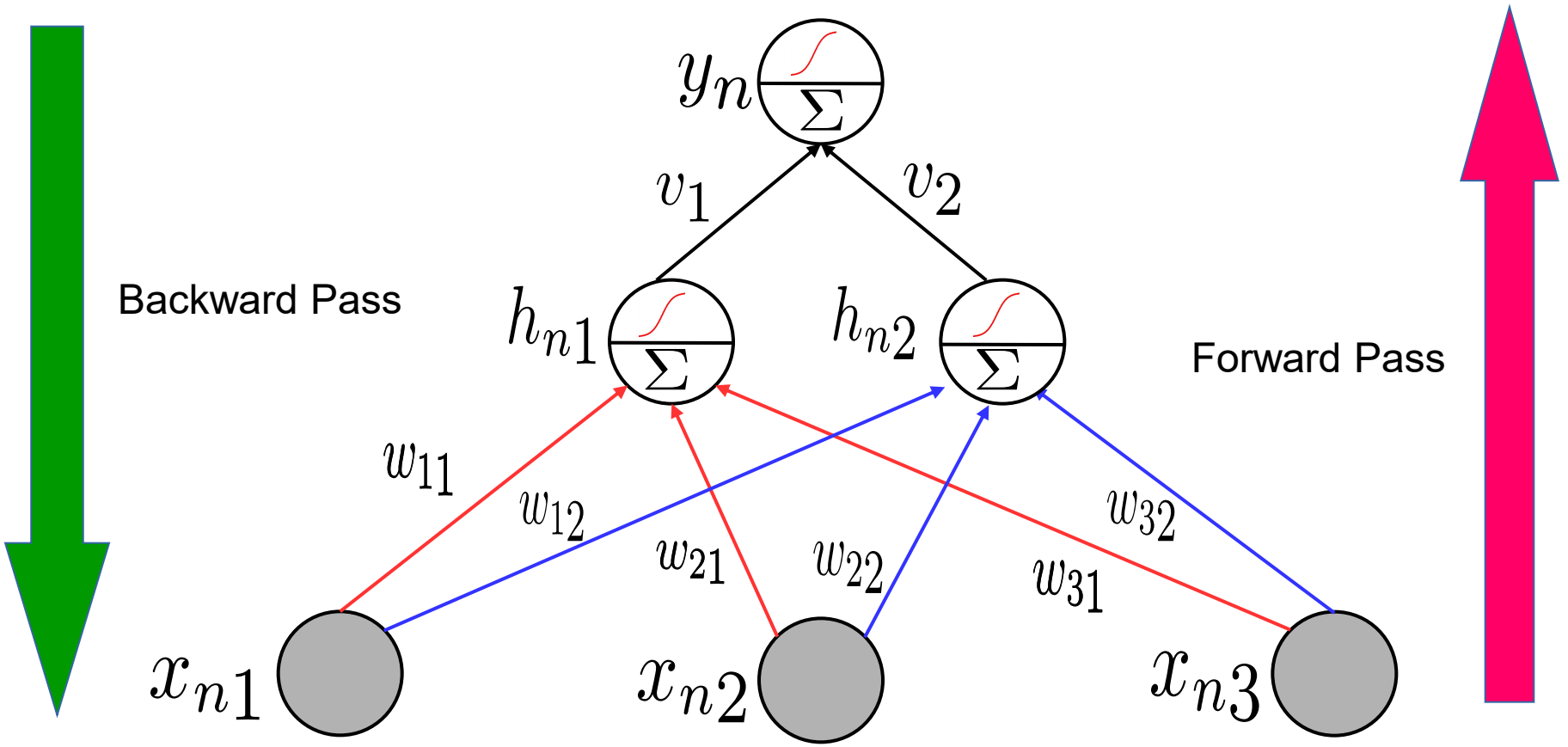
$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial w_{dk}} &= \sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial h_{nk}} \frac{\partial h_{nk}}{\partial w_{dk}} \\ \frac{\partial \mathcal{L}}{\partial h_{nk}} &= -(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)) v_k = -e_n v_k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

- 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

Backpropagation

- Forward prop computes errors $\boldsymbol{e}_n$ using current $\mathbf{W}$, $\boldsymbol{v}$.
  Backprop updates NN params $\mathbf{W}$, $\boldsymbol{v}$ using grad methods

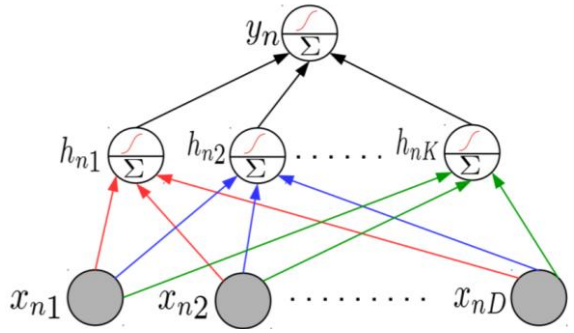
- Backprop iterates between a forward pass and a backward pass

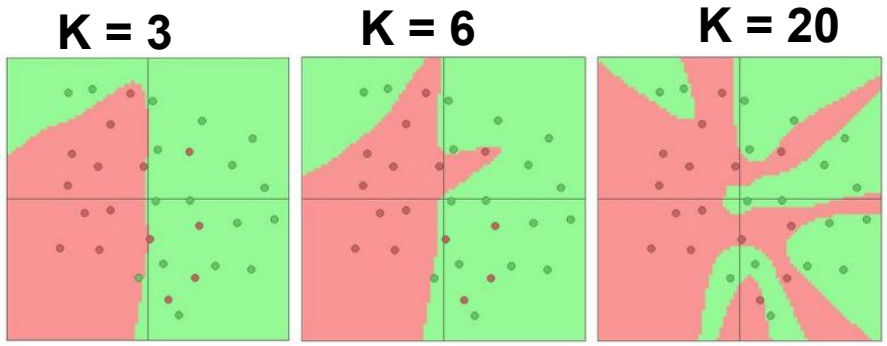


- Software frameworks such as Tensorflow and PyTorch support this already so you don't need to implement it by hand (so no worries of computing derivatives etc)

Representational Power of Neural Nets

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





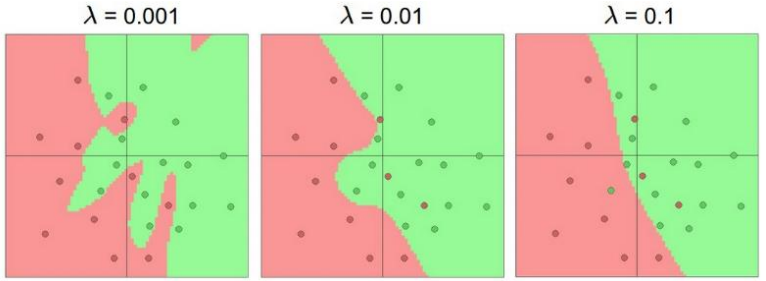
- Recall that each hidden unit "adds" a function to the overall function
- Increasing K (number of hidden units) will result in a more complex function
- Very large K seems to overfit (see above fig). Should we instead prefer small K?
- No! It is better to use large K and regularize well. Reason/justification: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

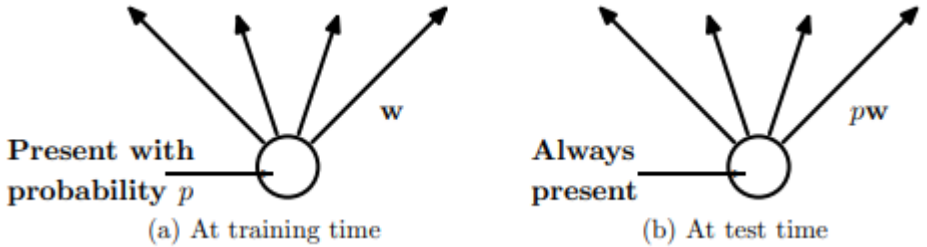
Various other tricks, such as weight sharing across different hidden units of the same layer (used in convolutional neural nets or CNN)

- Neural nets can overfit. Many ways to avoid overfitting, such as
  - Standard regularization on the weights, such as ℓ_2, ℓ_1, etc (ℓ_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



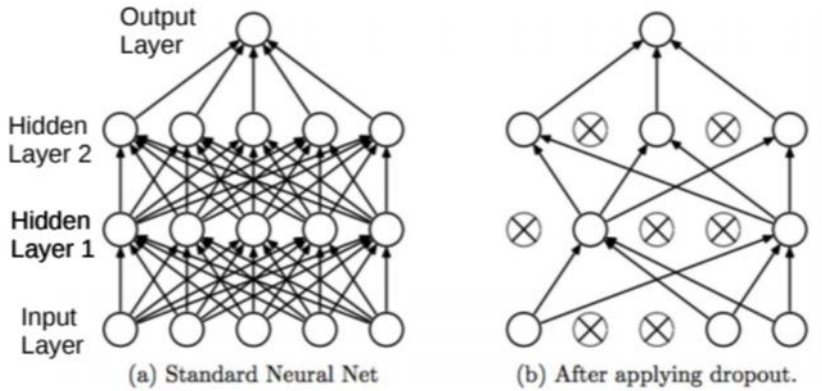
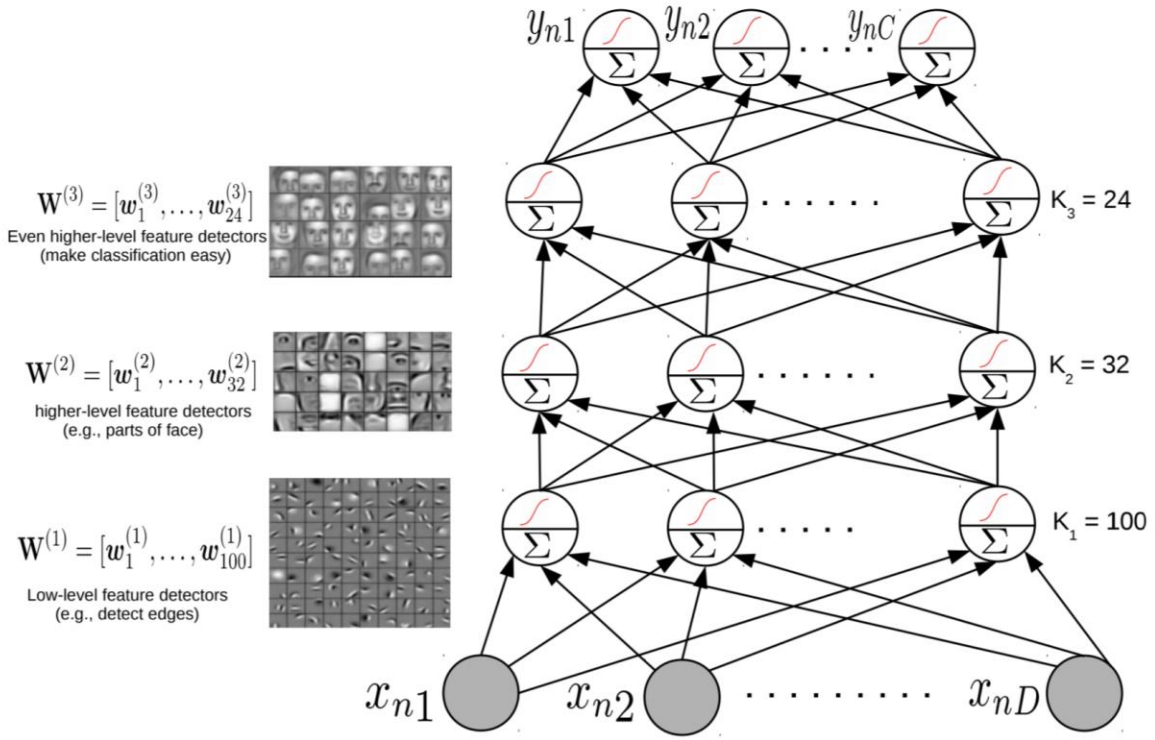


Fig courtesy: Dropout: A Simple Way to Prevent Neural Networks from Overfitting (Srivastava et al, 2014)

Wide or Deep?

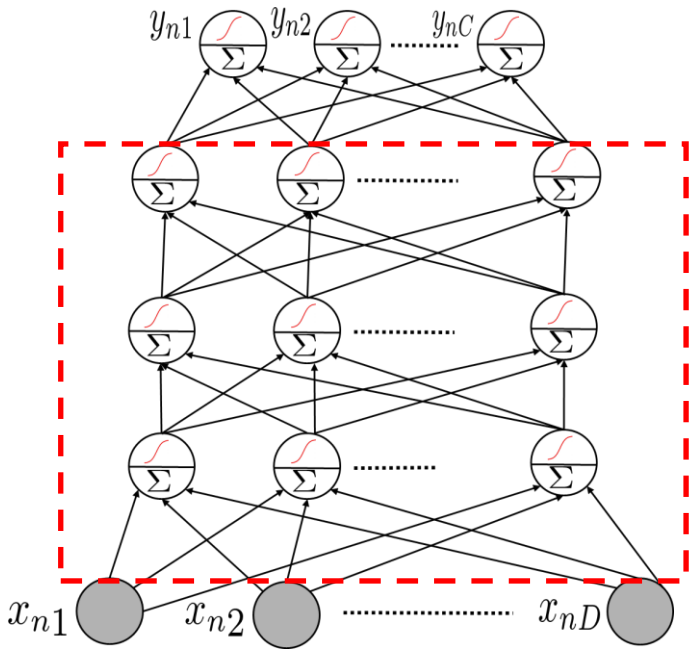
- While very wide single hidden layer can approx. any function, often we prefer many, less wide, hidden layers



- 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



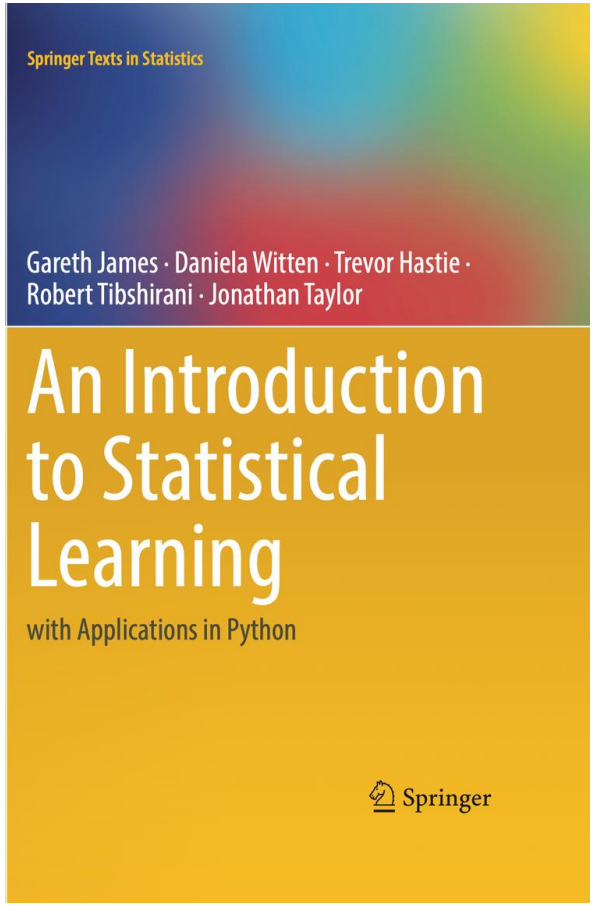
This part of a pre-trained net can be used as a feature extractor on some new task

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

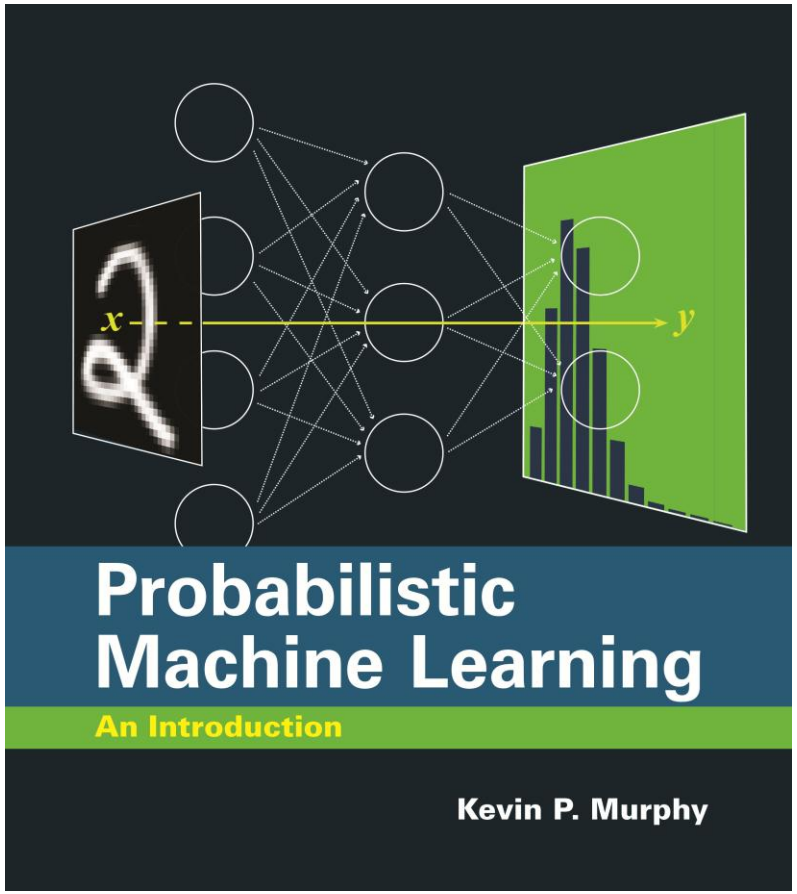
  - Fine-tuning: Use a pre-trained net, use its weights as initialization to train a deep net for a new but related task (useful when we don't have much training data for the new task)

References

Chapter 10



Chapter 13



Thank you