DS 503: Advanced Data Analytics

Lecture 14: Into to Neural Nets

Based on MMDS Ch 14 and
CS771 slides by Dr. Piyush Rai

Instructor: Dr. Gagan Gupta

Example (Learning Non-Linear Boundaries)

Example 1: We have length 4 bit-vectors of the from $[x_1\ x_2\ x_3\ x_4]$
The output y is 1 when there are 2 consecutive 1s

Training Ex

$[0\,1\,1\,0], 1$ $\quad$ $[1\,1\,0\,0], 1$

$[0\,0\,1\,0], 0$ $\quad$ $[1\,0\,0\,1], 0$

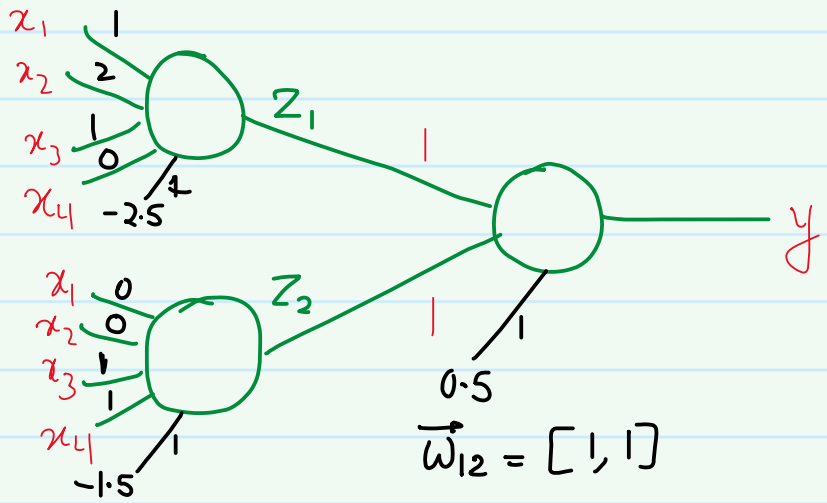
and so on

$\vec{w}_{11} = [1, 2, 1, 0]$

$b_{11} = -2.5$

$\vec{w}_{21} = [0, 0, 1, 1]$

$b_{21} = -1.5$



$\vec{w}_{12} = [1, 1]$

$b_{12} = 0.5$

Each node is a perceptron.

Q. When can z_1 be 1?

$x_1 + 2x_2 + x_3 - 2.5 > 0$

$\Rightarrow [x_2 > 0]$ & $[x_1$ or $x_3 > 0]$

Q. When can z_2 be 1?

Q. When can y be 1?

$[z_1 > 0]$ or $[z_2 > 0]$

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.

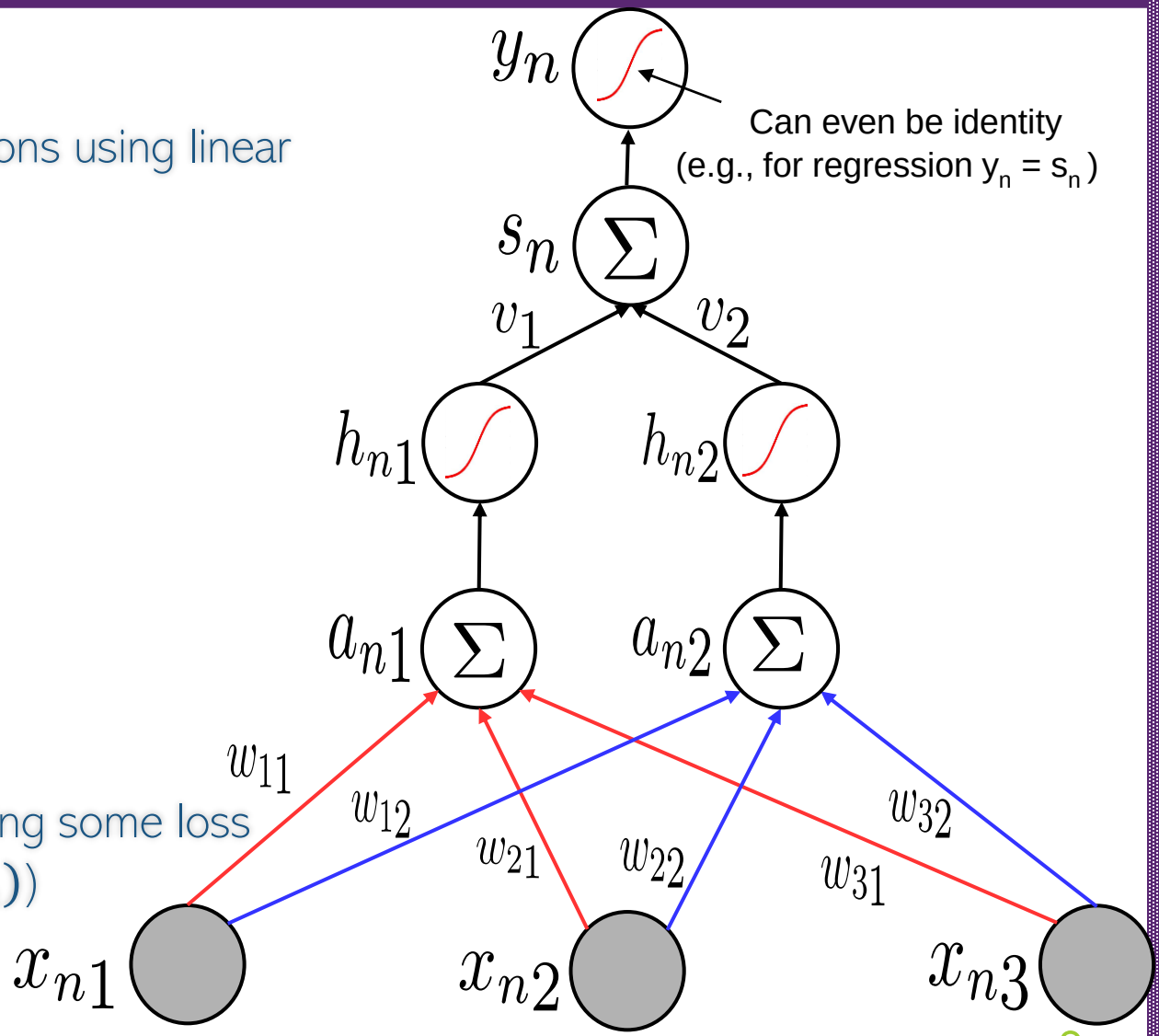
$$h_{nk} = g(a_{nk})$$

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

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

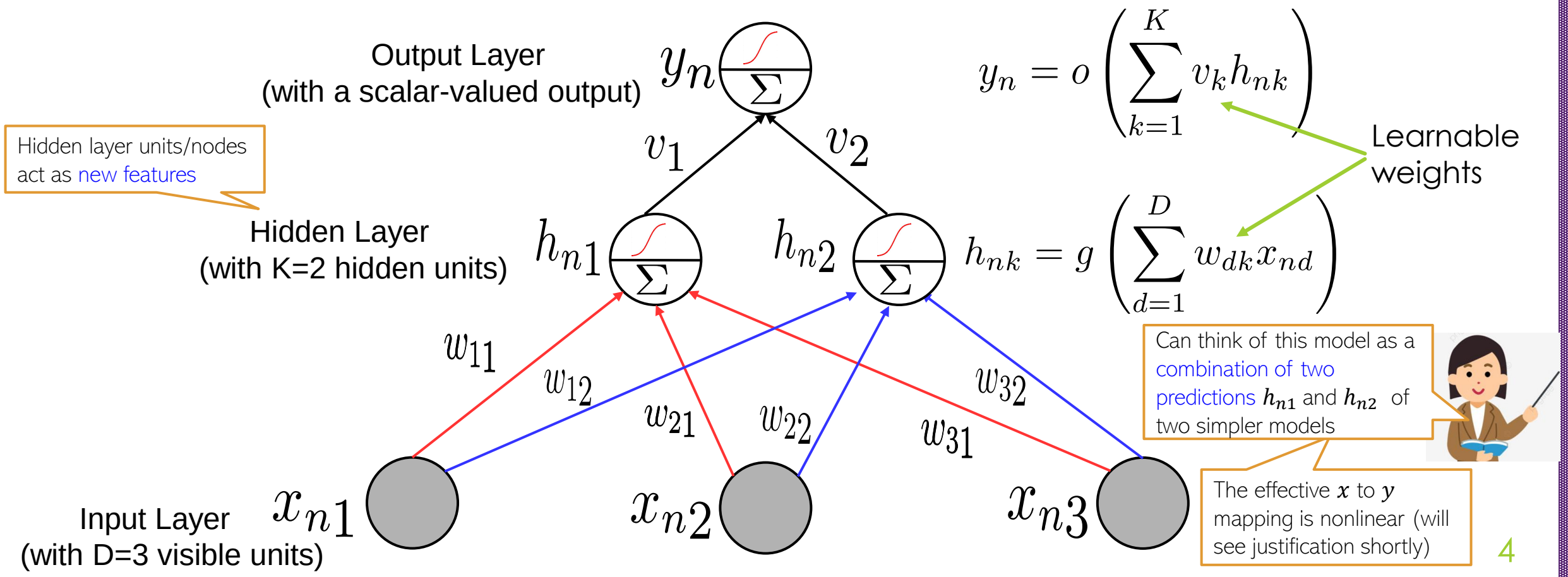
- Finally, output is produced as $\boldsymbol{y = o(s_n)}$
- Unknowns $(\boldsymbol{w_1, w_2, \ldots, w_K, v})$ learned by minimizing some loss function, for example $\boldsymbol{\mathcal{L}(W, v) = \sum_{n=1}^{N} \ell(y_n, o(s_n))}$

(squared, logistic, softmax, etc)



Neural Networks: Multi-layer Perceptron (MLP)

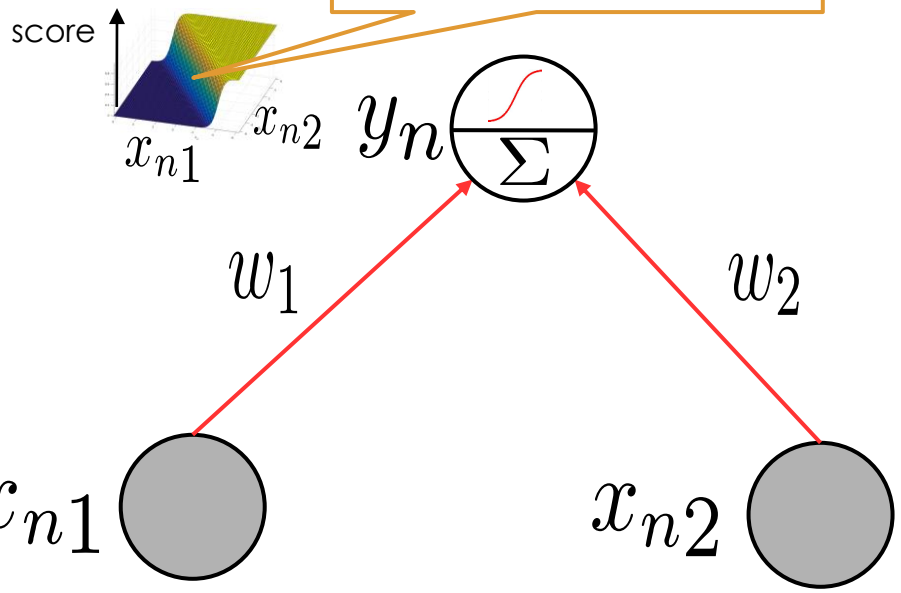
- An MLP consists of an input layer, an output layer, and one or more hidden layers



MLP Can Learn Nonlin. Fn: A Brief Justification

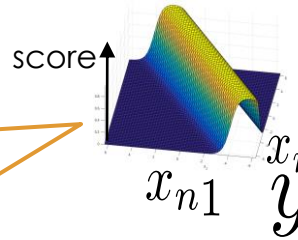
An MLP can be seen as a composition of multiple linear models combined nonlinearly

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

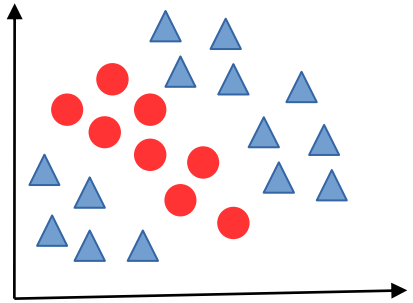


Standard Single "Perceptron" Classifier (no hidden units)

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

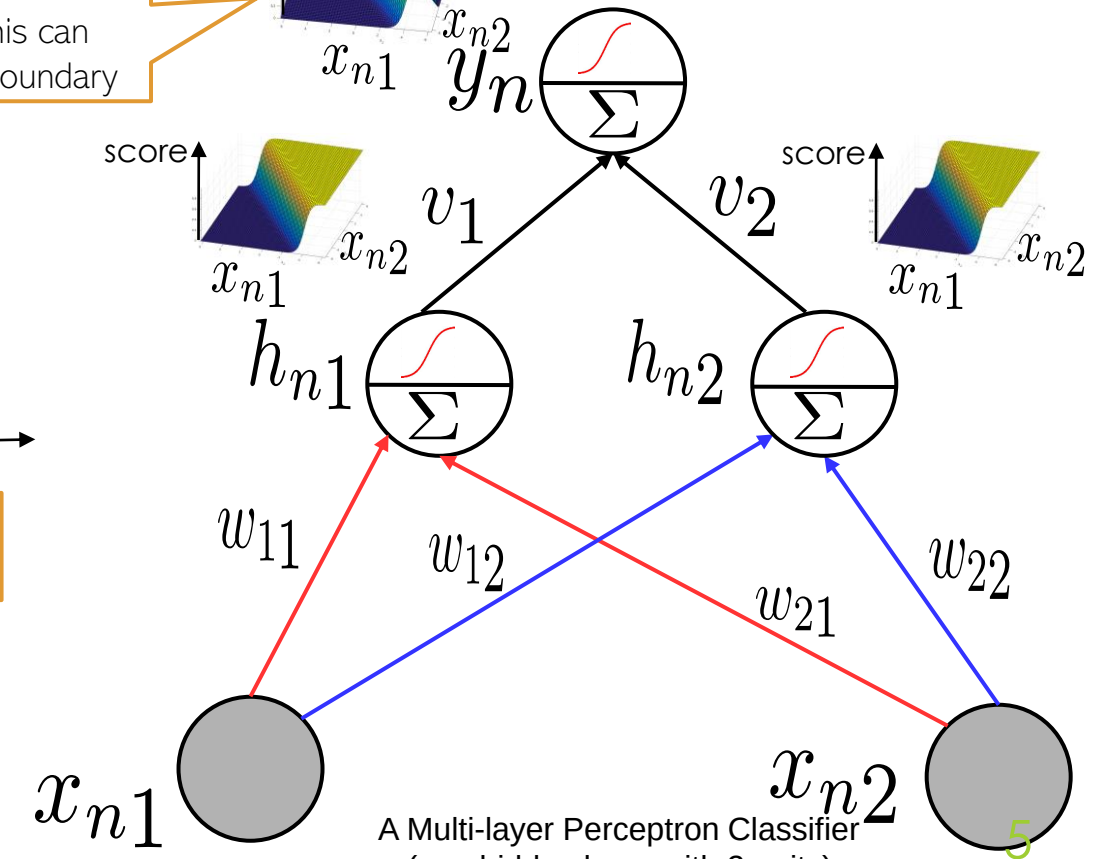


High-score in the middle and low-score on either of the two sides of it. Exactly what we want for the given classification problem



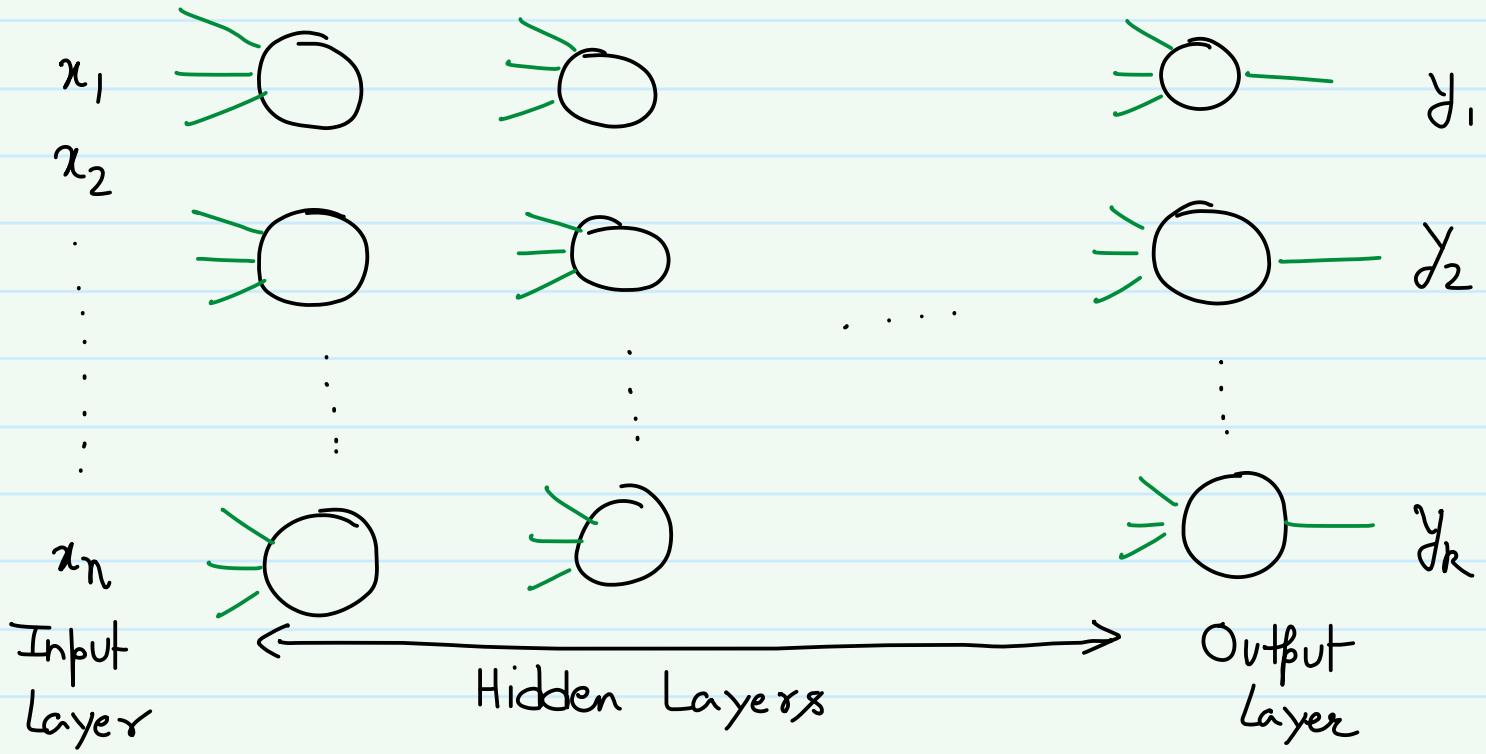
A nonlinear classification problem

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



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

Neural Net Designs

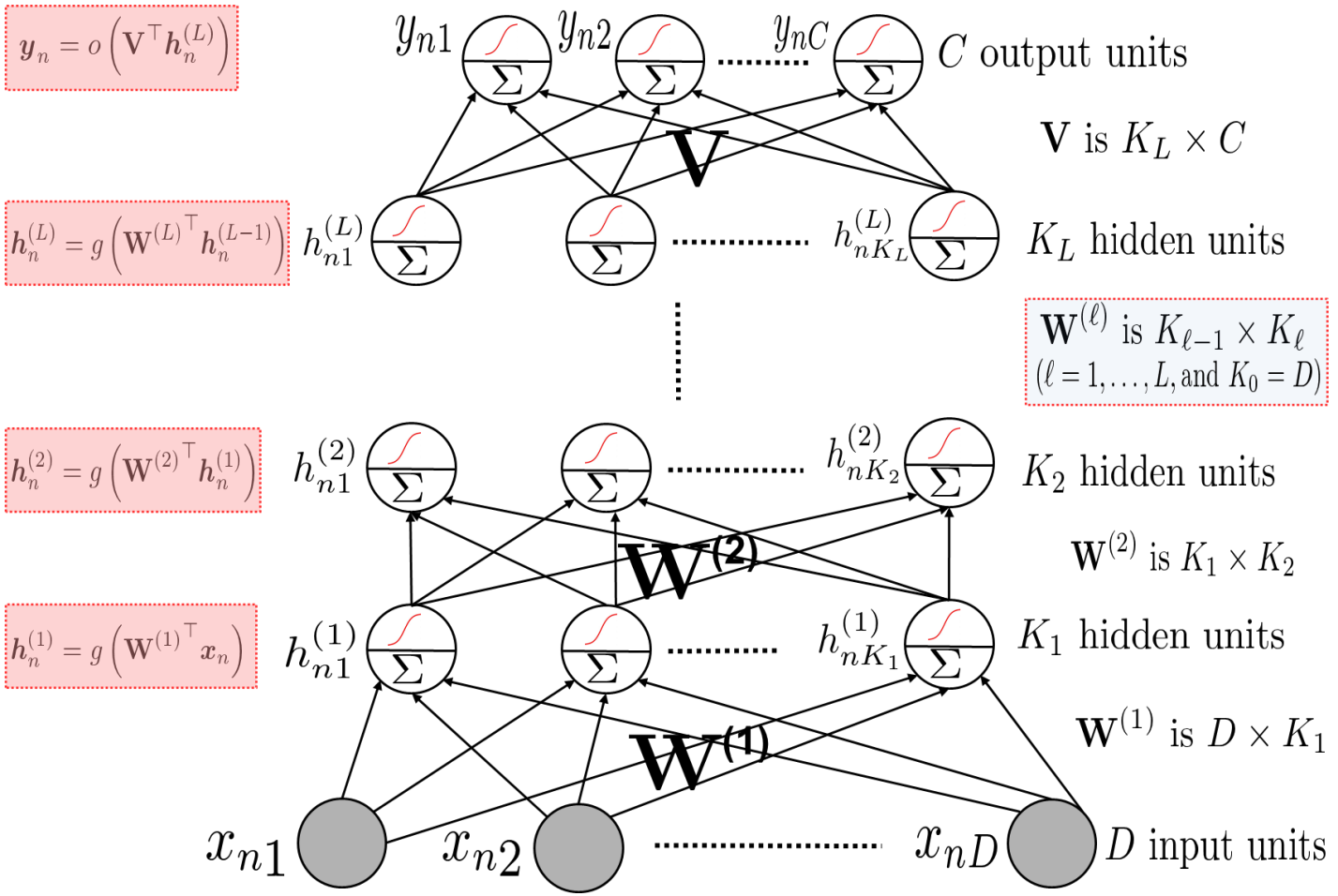


* Each layer can have different # of nodes
* Output layer can also have many nodes
  Eg. Multi-Labels

Output of each node depends on $\Sigma x_i w_i$ where the sum is over all inputs x_i & w_i is the weight of the input.

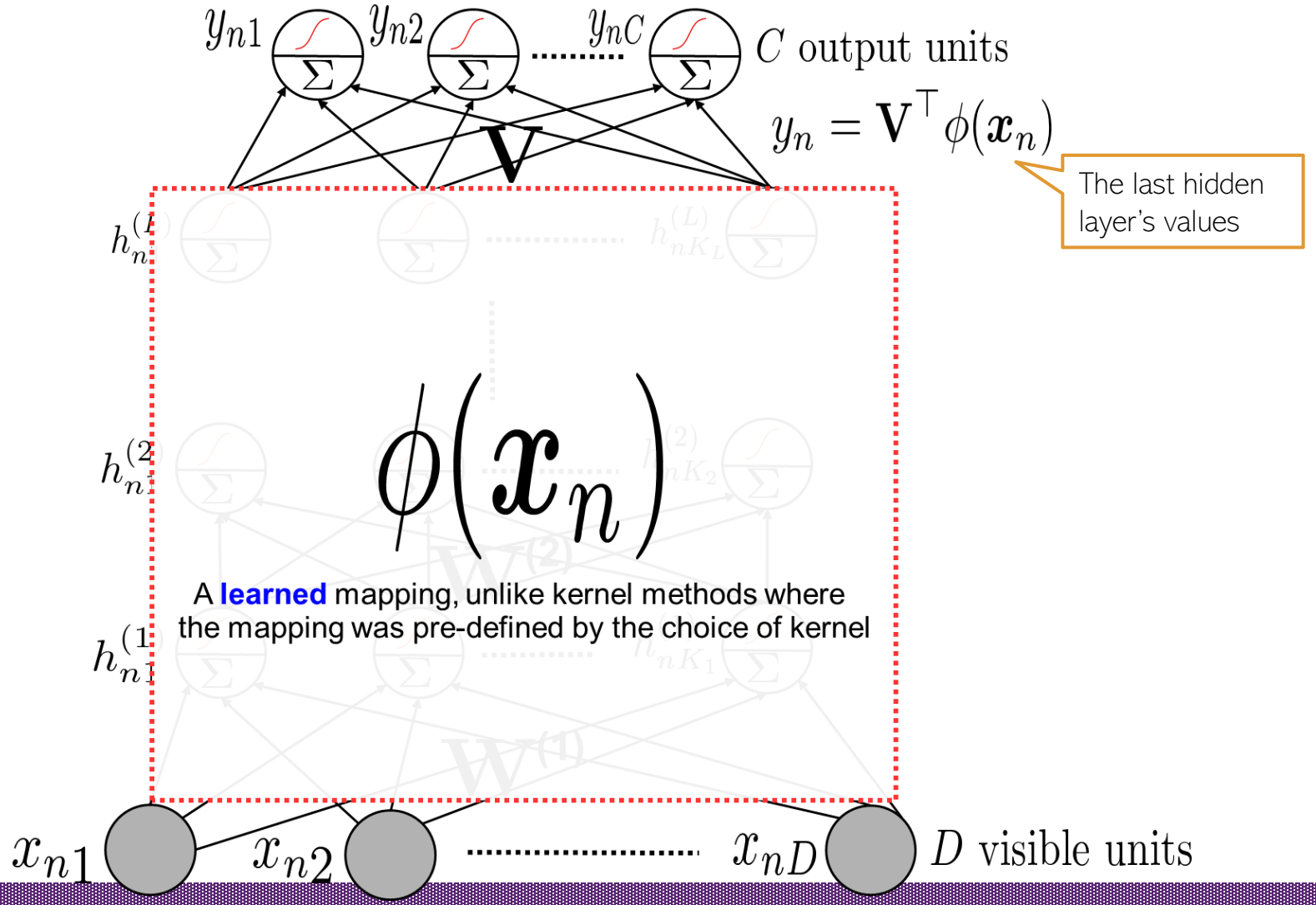
Multiple Hidden Layers (One/Multiple Outputs)

- 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/Encoders

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



Kernel Methods vs Neural Nets

- 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

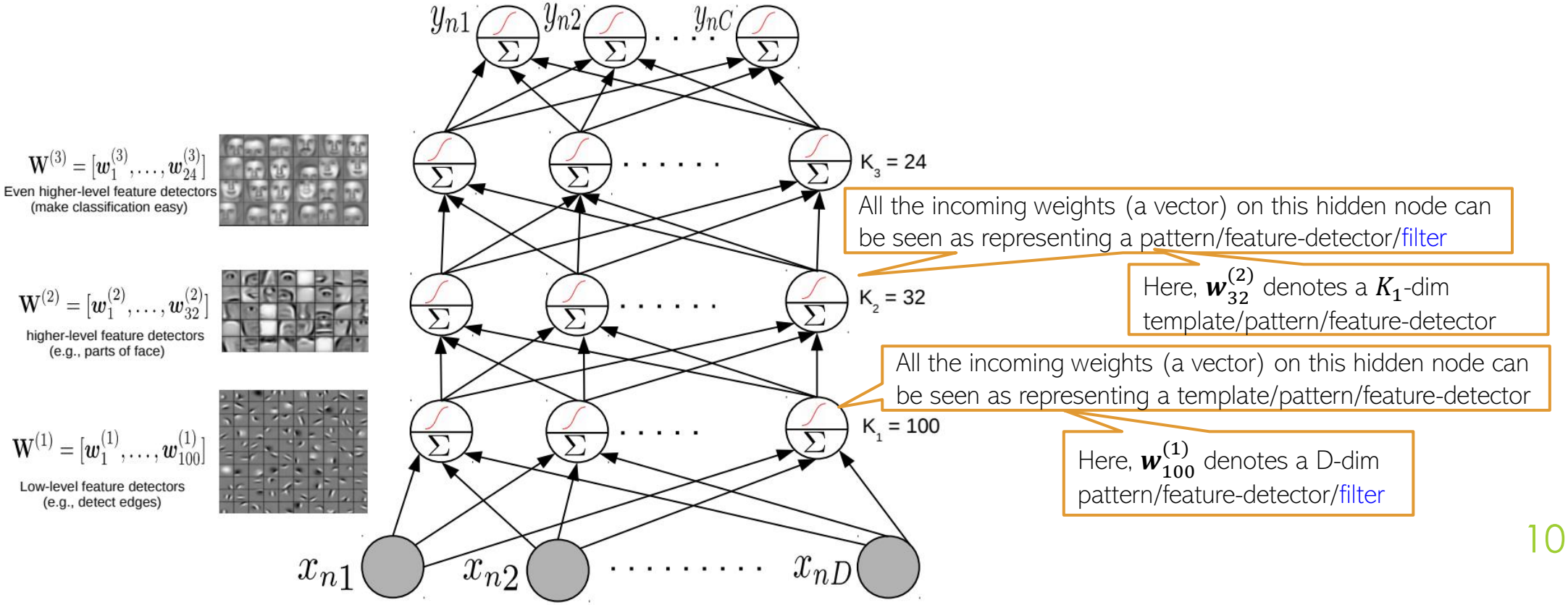
$$y = \sum_{k=1}^{K} v_k h_k$$

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

- 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

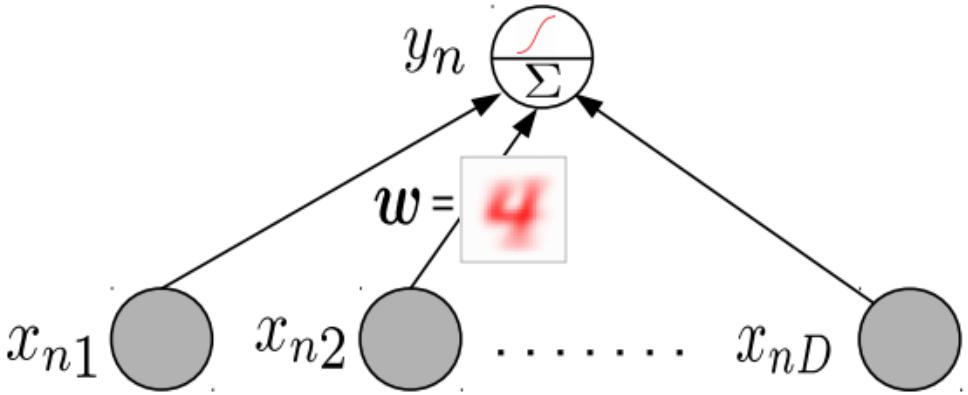
Feature Learned by a Neural Network

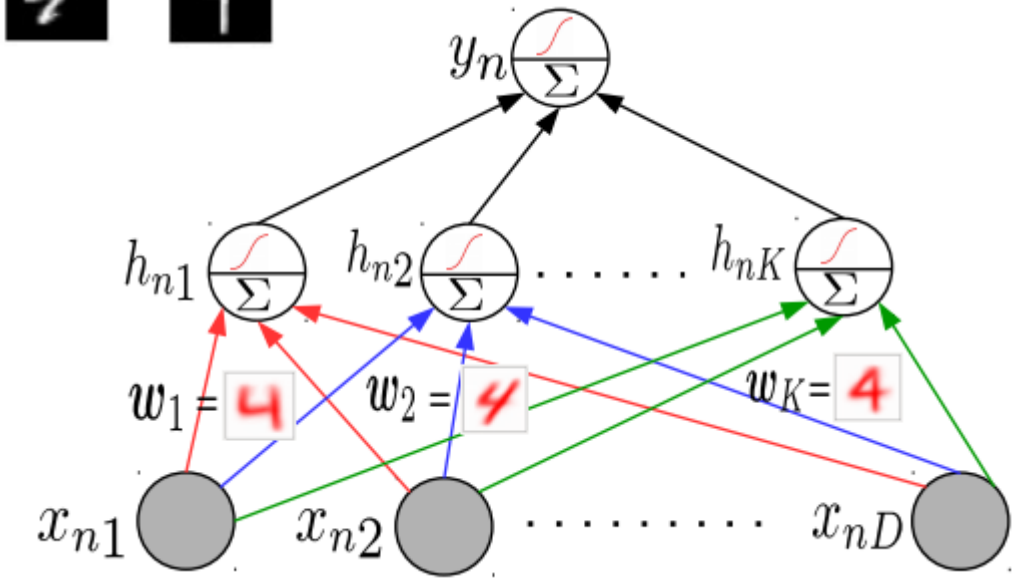
- Node values in each hidden layer tell us how much a "learned" feature is active in $\boldsymbol{x_n}$
- Hidden layer weights are like pattern/feature-detector/filter



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





Interconnection Among Nodes

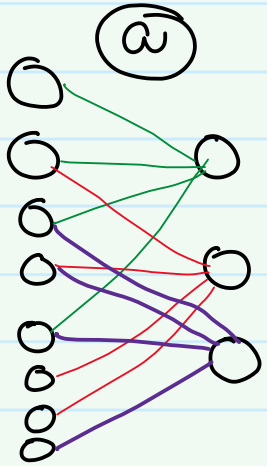
Neural nets can differ in how nodes in one layer interconnect to the next layer (tonight)

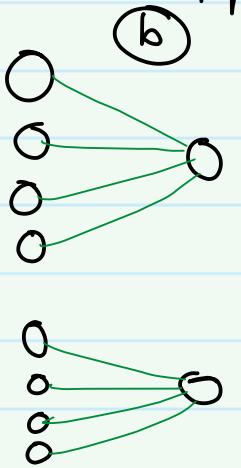
Most General Case :- Fully connected. [#of Parameters = $(n_1+1)n_2$] (bias, Left layer, Right layer nodes)

Other options :-

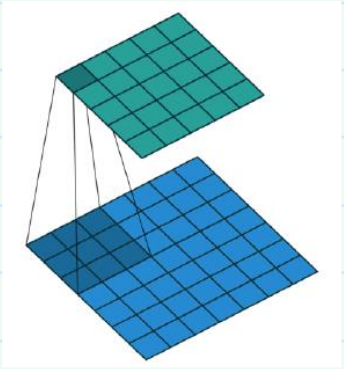
@ Random : For some 'm', pick m out of n_1 nodes & make them input $({}^{n_1}C_m)$ options

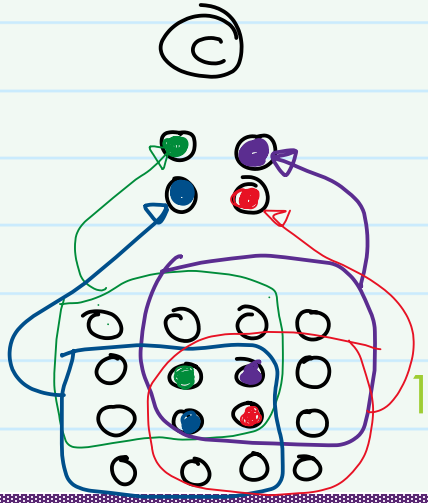
(b) Pooled : Partition nodes of one layer into some # of clusters. In next layer (pooling layer) there is one node for each cluster & this node has all & only the member of its clusters as inputs.





Convolutional.





Design Issues for Neural Nets

- Building a neural net to solve a given problem is partially art and partially science
- Before we begin to train a net by finding the weights on the inputs that serve our goals, we have to make a number of design decisions.
  - How many hidden layers should we use?
  - How many nodes will there be in each of the hidden layers?
  - In what manner will we inter-connect the outputs of one layer to the inputs of the next layer?
  - What activation function to use at each node?
  - What cost function should we minimize to express what weights are the best?
  - How to compute the outputs of each node as a function of the inputs?
- What algorithms shall we use to exploit the training examples in order to optimize the weights?

Dense Feed-forward Networks

* Ability to learn weights from training data.

Linear Algebra Notation.

Example problem :- Consecutive 1's in 4 bit vectors.

input, $X = [x_1 \; x_2 \; x_3 \; x_4]$

$w_{11} = [w_{111} \; w_{112} \; w_{113} \; w_{114}]$

$w_{12} = [w_{121} \; w_{122} \; w_{123} \; w_{124}]$

hidden node vector $h = [h_1, h_2]$

$h_1 = \text{sgn}(w_{1i}^T X + b_1)$

$h_2 = \text{sgn}(w_{2i}^T X + b_2)$

i.e., $h_i = \text{sgn}(w_i^T X + b_i)$ for $i = 1, 2$

By organizing w_{11} & w_{12} in a 2×4 matrix W_1, where i^{th} row is w_i^T

$$h = \text{sgn}(W_1 X + b)$$

↑ works element-wise on vectors

$$y = \text{sgn}(w_2 h + c)$$

Information Flow

Each Layer has its own additional matrix of Weights & vector of biases + Activation functions.

Perceptrons used sign function, but other choices exist.

Since edges are directed forward. Information flows from left to right.

$\longrightarrow$ Feed forward networks. (No cycles)

Suppose there are l hidden layers & additional output layer $l+1$

Let weight matrix for i^{th} layer be W_i & bias vector be b_i.

$W_1 W_2 \dots W_{l+1}$ & $b_1 b_2 \dots b_{l+1}$

These are the parameters of the model.

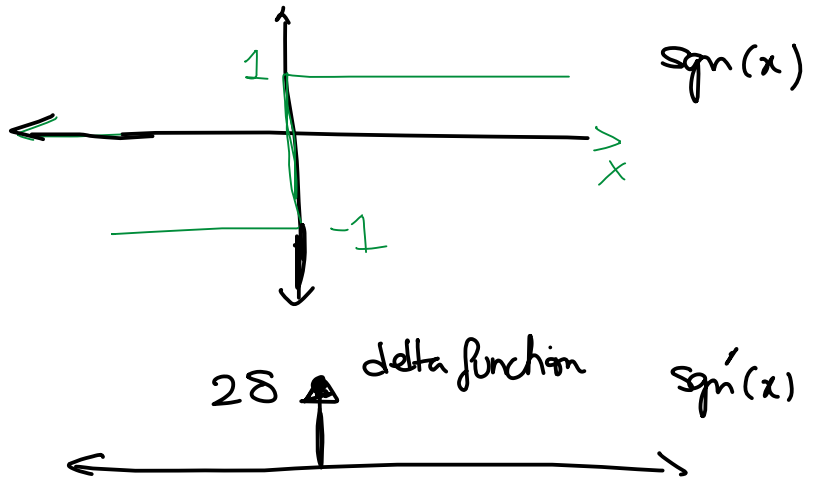
Activation Functions

- $y = F_{l+1}\ (F_l(F_{l-1} \dots (F_2(F_1(x) + b_1) + b_2) \dots + b_{l-1}) + b_l) + b_{l+1}$
- Since we typically use gradient descent to solve for optimal value of parameters, we look for activation functions with the following properties
  1. The function is continuous and differentiable everywhere (or almost everywhere)
  2. The derivative of the function doesn't saturate (i.e. become very small, tending to zero) over its expected input range. Very small derivatives tend to stall the learning process
  3. The derivative of the function doesn't explode (i.e. become very large, tending to infinity), since this would lead to issues of numerical instability

The sign function doesn't satisfy conditions 2, 3.

It's derivatives explode at 0 and is 0 everywhere else.

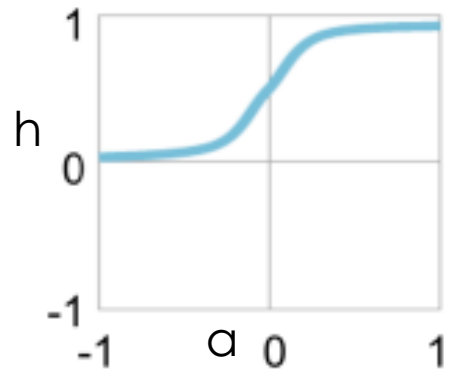
Let's therefore consider other activation functions.



https://en.wikipedia.org/wiki/Dirac_delta_function

Activation Functions: Some Common Choices

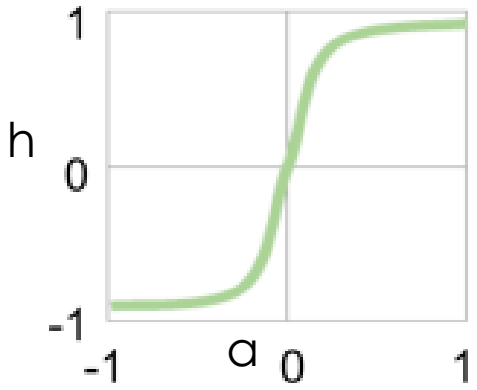
sigmoid



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

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

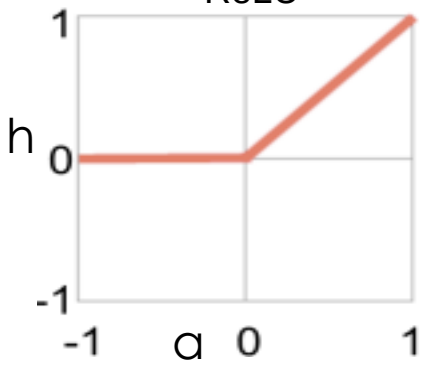
tanh



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)

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

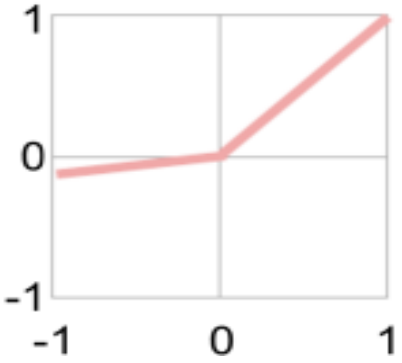
ReLU



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

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

Leaky ReLU



ReLU and Leaky ReLU are among the most popular ones

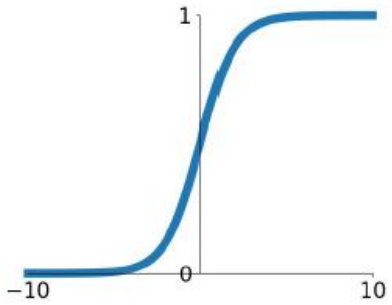
Leaky ReLU: $h = \max(\beta a, a)$
where β is a small postive number

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

Commonly Used Activation Functions

Sigmoid

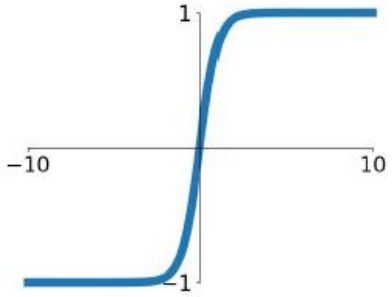
$$\sigma(x) = \frac{1}{1+e^{-x}}$$



tanh

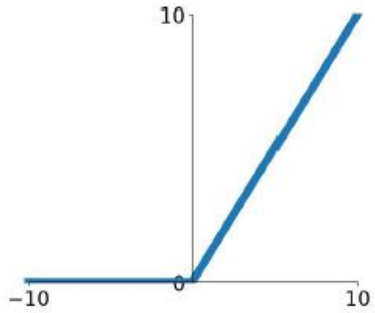
$$\tanh(x)$$

$$= \frac{e^{2x}-1}{e^{2x}+1}$$



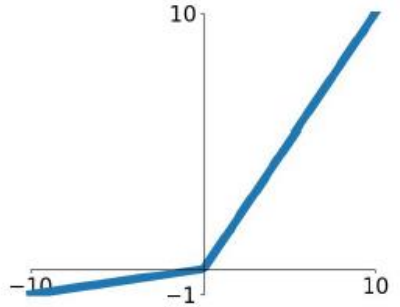
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$



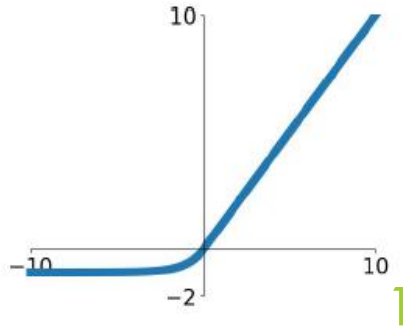
Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

$\alpha(e^x - 1) < 0$



Softmax

1. Operates on the entire vector.

If $X = [x_1, x_2, \ldots, x_n]$, then its soft-max $u(X) = [u(x_1), u(x_2) \ldots u(x_n)]$

where $$u(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Pushes the largest component of the vector towards 1.
All others towards 0.

Can be interpreted as a probability distribution. $\sum_i u(x_i) = 1$

2. Common application: Output layer for a classification problem.
Output vector has a component for each class.

3. Works well with cross-entropy loss to handle saturation {very small gradients}

Accurate Calculation trick :- $$u(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} = \frac{e^{x_i - c}}{\sum_j e^{x_j - c}}$$ for any c

Choose $c = \max_j x_j$ so that $x_j - c \leq 0$ for all j. Thus $\forall_j \; 0 \leq e^{x_j - c} \leq 1$

Cross Entropy Loss

Consider a multi-class classification problem with target classes $C_1, C_2, C_3 \dots C_n$

Suppose each point in training set is of the form (x, p) where x is input & $p = [p_1\ p_2\ p_3 \dots p_n]$ is output

p_i is the probability that input x belongs to class C_i with $\sum p_i = 1$.

If we are certain that an input belongs to a particular class C_i then $p_i = 1$ & $p_j = 0\ \forall j \neq i$

We can design our neural-net to produce output $q = [q_1\ q_2 \dots q_n]$, using softmax for eg.

Hence both input & output are probability distributions.

Recall that entropy $= \sum_i -p_i \log p_i = H(p)$

Cross entropy $= -\sum_i p_i \log q_i = H(p, q)$

- In information theory it is used to quantify sub-optimal codes.
- p_i: actual probability (ground truth)
- $\log q_i$: Estimated probability

In general

$H(p, q) \geq H(p)$

KL Divergence

$D(p||q) = H(p, q) - H(p)$

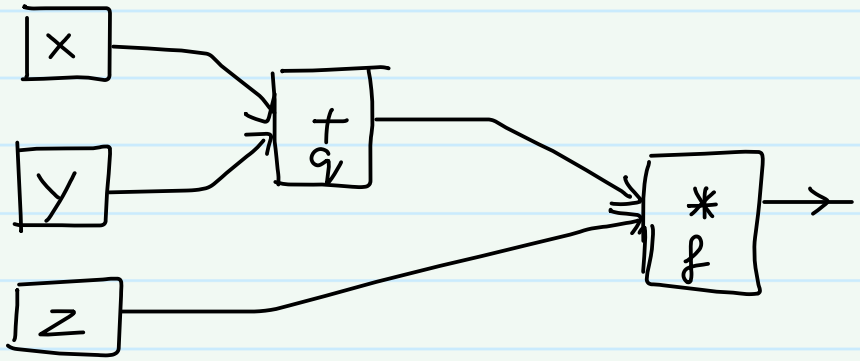
$= \sum_i^n p_i \log \frac{p_i}{q_i}$

Compute Graph

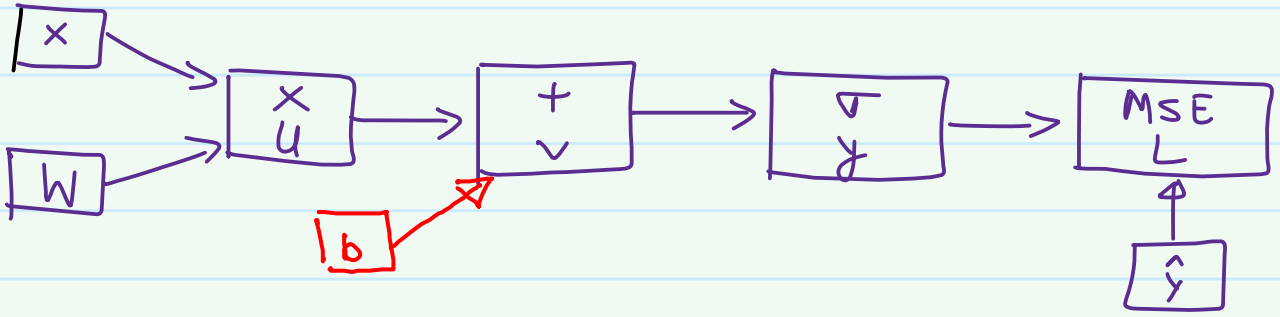
Find good values for parameters of the model. [weights & thresholds]

Compute Graph :- Directed graph, acyclic [DAG]

Each node has an operand or an operator (+, x, σ, MSE)
When a node has both an operand & operator, operator is written above the operand



$$f = (x + y)z$$



$$y = \sigma(Wx + b)$$
$$u = Wx$$
$$v = u + b$$
$$y = \sigma(v)$$
$$L = MSE(y, \hat{y})$$

Example of Backprop

$f(x, y, z) = (x + y)z$

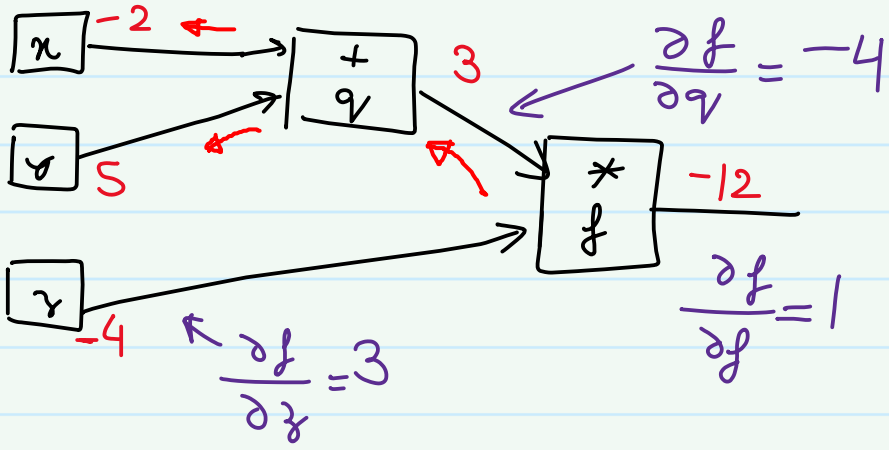
$q = x + y$

$\frac{\partial q}{\partial x} = 1 \qquad \frac{\partial q}{\partial y} = 1$

$f = qz$

$\frac{\partial f}{\partial q} = z \qquad \frac{\partial f}{\partial z} = q$

Compute $\frac{\partial f}{\partial x} \quad \frac{\partial f}{\partial y} \quad \frac{\partial f}{\partial z}$



Upstream gradient

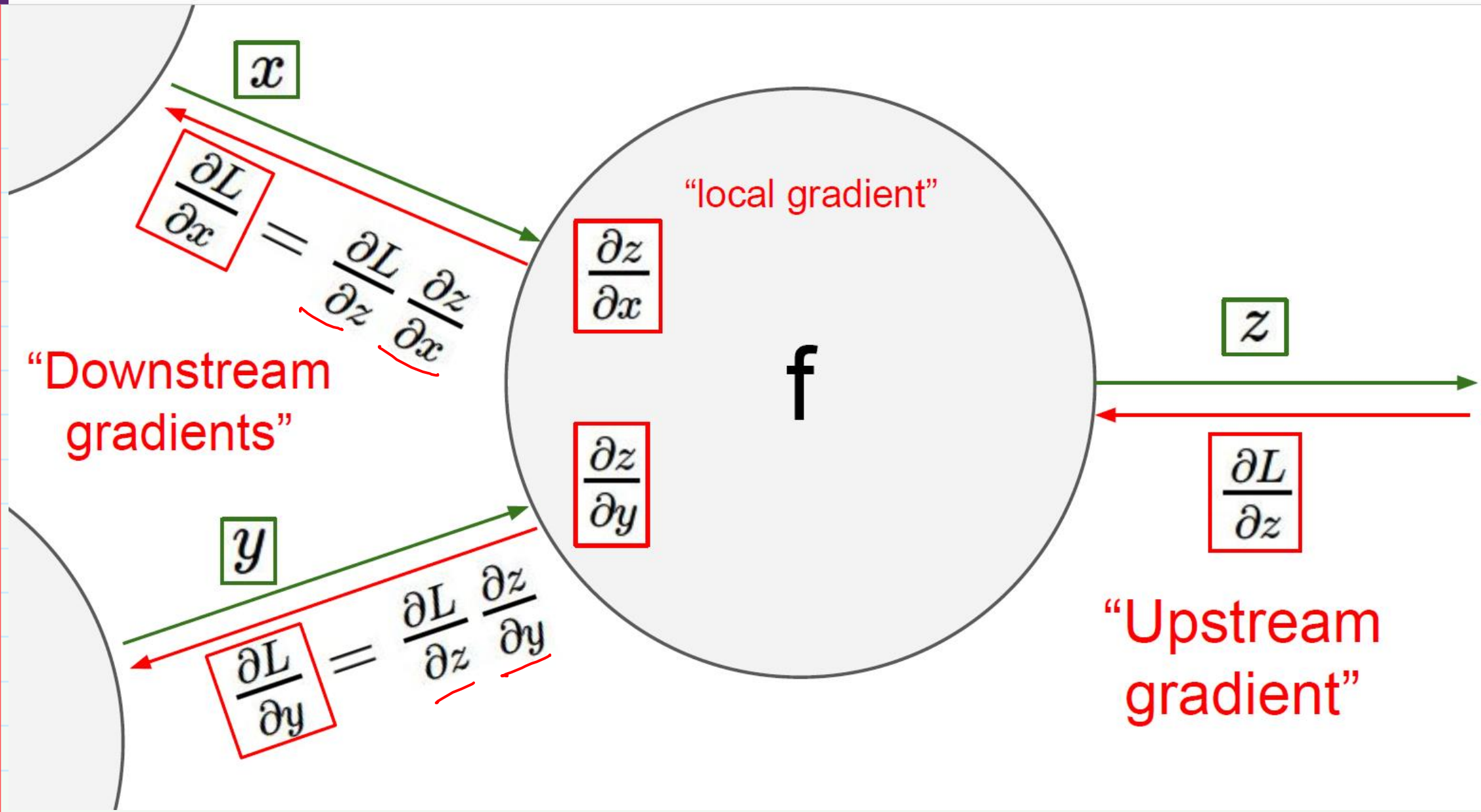
$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial y}$ → Local gradient

$-4 \cdot 1 = -4$

$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial x}$

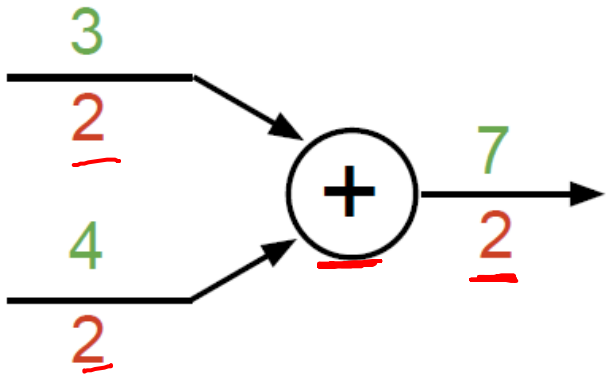
$= -4 \cdot 1 = -4$

Upstream and Local Gradients

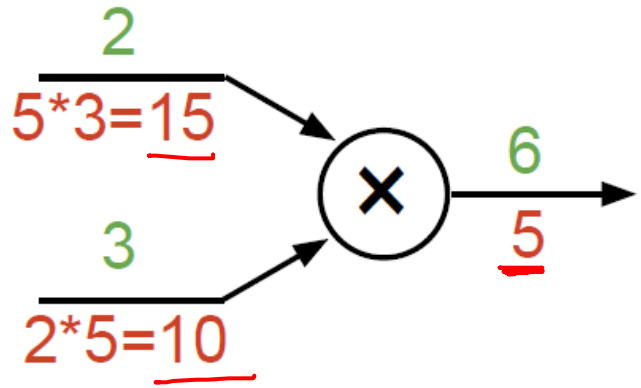


Patterns in Gradient Flow

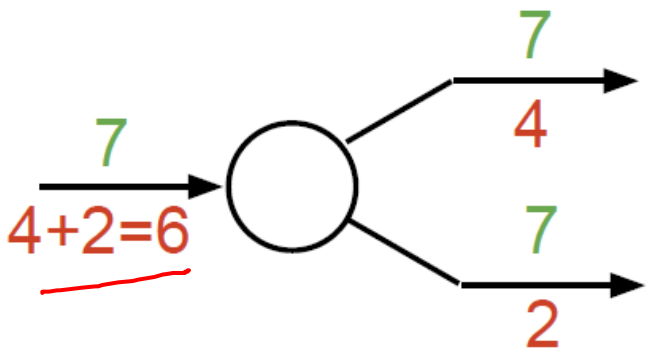
add gate: gradient distributor



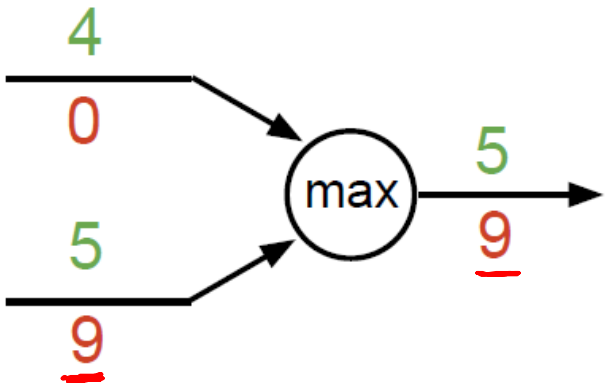
mul gate: "swap multiplier"



copy gate: gradient adder



max gate: gradient router



Bonus Question 14.1 (5 marks)

* $f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$

Draw the compute graph for this function.

Assume $w_0 = 2$, $w_1 = -3$, $x_0 = -1$, $x_1 = -2$, $w_2 = -3$ are input nodes

Now perform the forward & backprop to compute gradients w.r.t. each input x_i & w_i

Gradients, Jacobians, and Chain Rule

Goal :- Compute Gradient of loss function w.r.t. parameters of the network.

Recall :- If $f : \mathbb{R}^N \to \mathbb{R}$ from real-valued vector to scalar
$X = [x_1, x_2 \cdots ; x_n]$ & $y = f(x)$

$$\nabla_x y = \left[\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, \cdots, \frac{\partial y}{\partial x_n} \right]$$

Vector Valued :- $f : \mathbb{R}^n \to \mathbb{R}^m$

$$J = \begin{bmatrix} (\nabla f_1)^T \\ (\nabla f_2)^T \\ \vdots \\ (\nabla f_n)^T \end{bmatrix}$$

Chain Rule

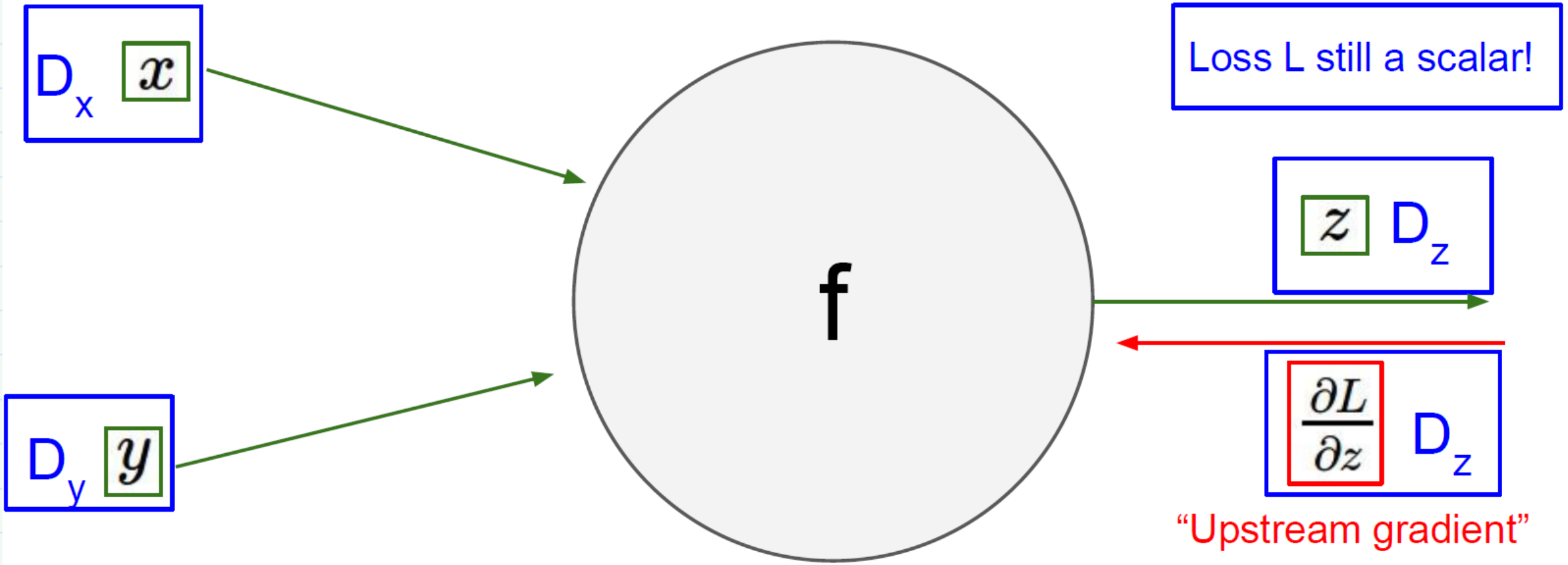
$y = g(x)$, $z = f(y) = f(g(x))$

$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}$$

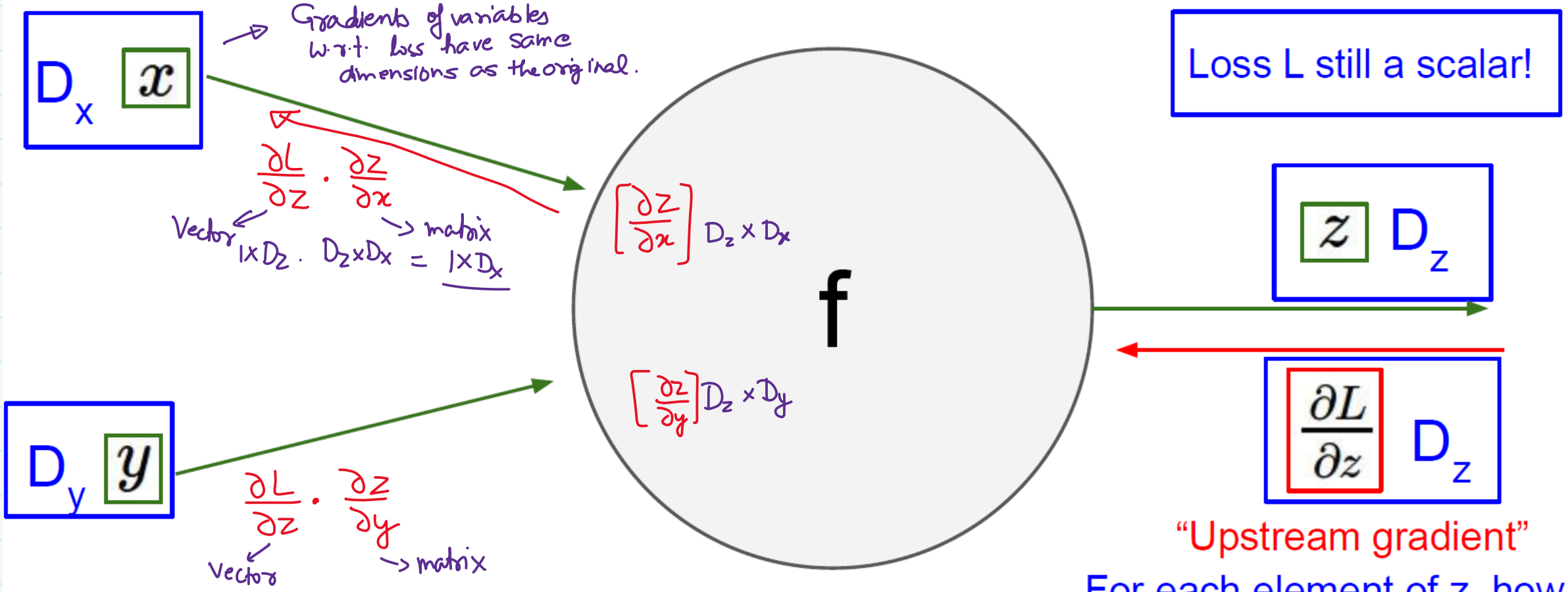
If $z = f(u, v)$ where $u = g(x)$ & $v = h(x)$

$$\frac{dz}{dx} = \frac{dz}{du} \cdot \frac{du}{dx} + \frac{dz}{dv} \cdot \frac{dv}{dx}$$

Backprop with vectors



Backprop with vectors



Backpropagation with vectors

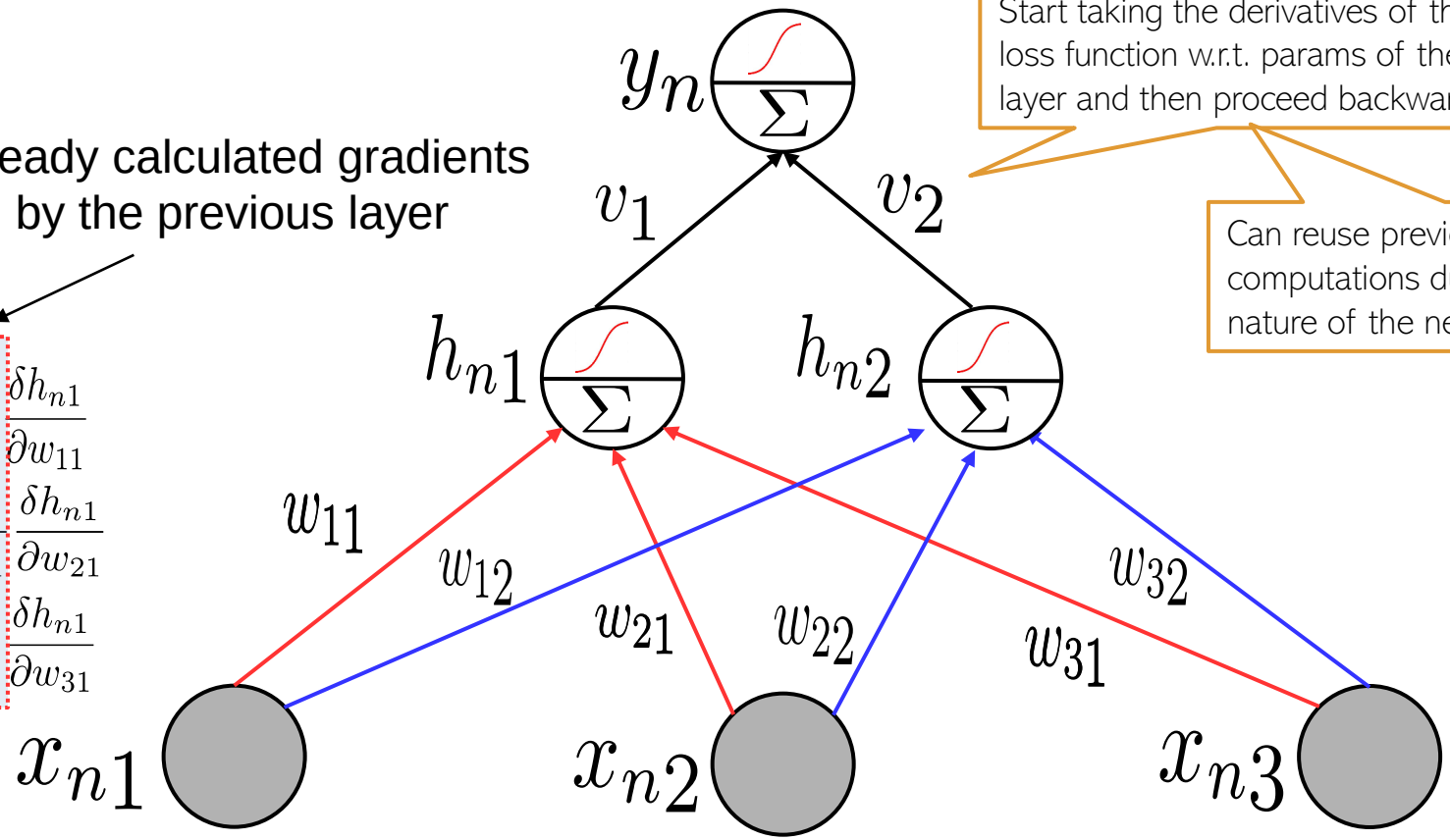
- Backpropagation = Gradient descent using chain rule of derivatives
- Chain rule of derivatives: Example, if $y = f_1(x)$ and $x = f_2(z)$ then $\frac{\partial y}{\partial z} = \frac{\partial y}{\partial x}\frac{\partial x}{\partial z}$

Reuse already calculated gradients computed by the previous layer

$$\frac{\delta \mathcal{L}}{\partial w_{11}} = \frac{\delta \mathcal{L}}{\partial h_{n1}} \frac{\delta h_{n1}}{\partial w_{11}}$$
$$\frac{\delta \mathcal{L}}{\partial w_{21}} = \frac{\delta \mathcal{L}}{\partial h_{n1}} \frac{\delta h_{n1}}{\partial w_{21}}$$
$$\frac{\delta \mathcal{L}}{\partial w_{31}} = \frac{\delta \mathcal{L}}{\partial h_{n1}} \frac{\delta h_{n1}}{\partial w_{31}}$$

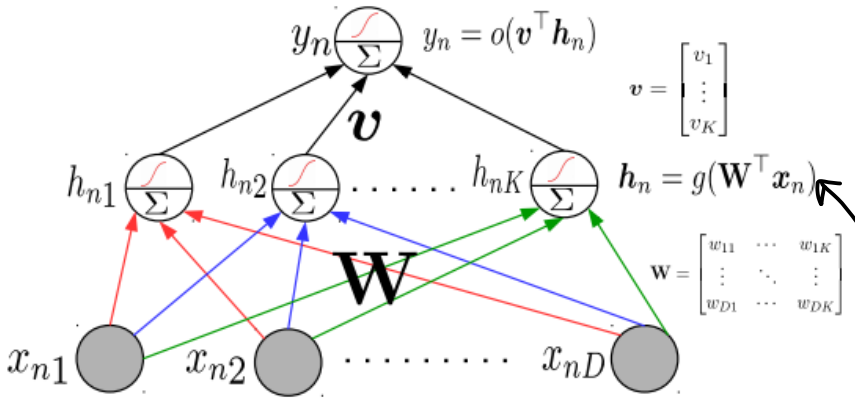
Start taking the derivatives of the loss function w.r.t. params of the last layer and then proceed backwards

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



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

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

error

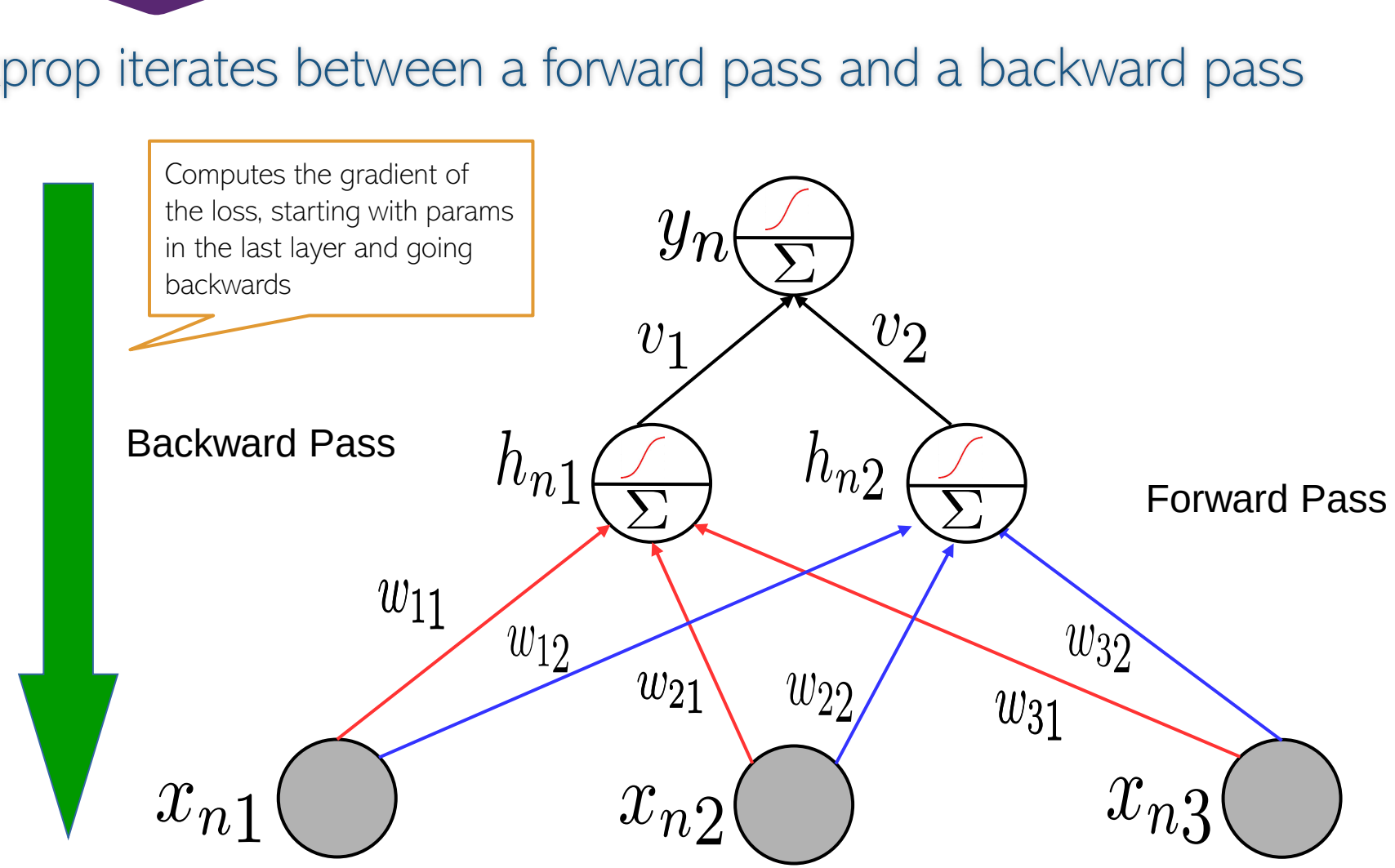
- Gradient of $\mathcal{L}$ w.r.t. $\mathbf{W}$ requires chain rule

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

- Forward prop computes errors 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

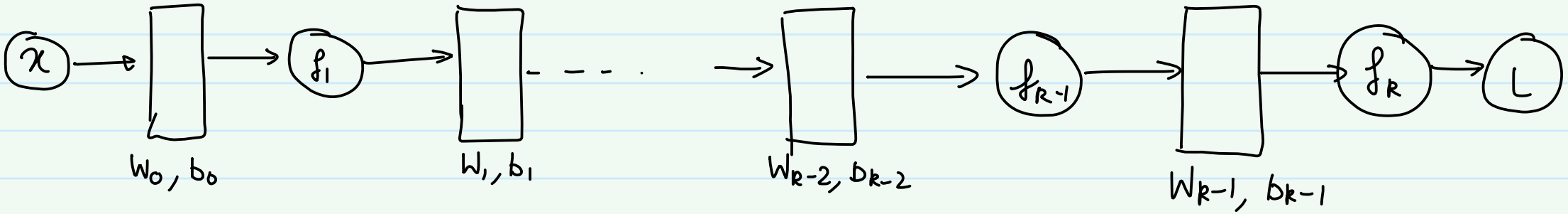
Backpropagation

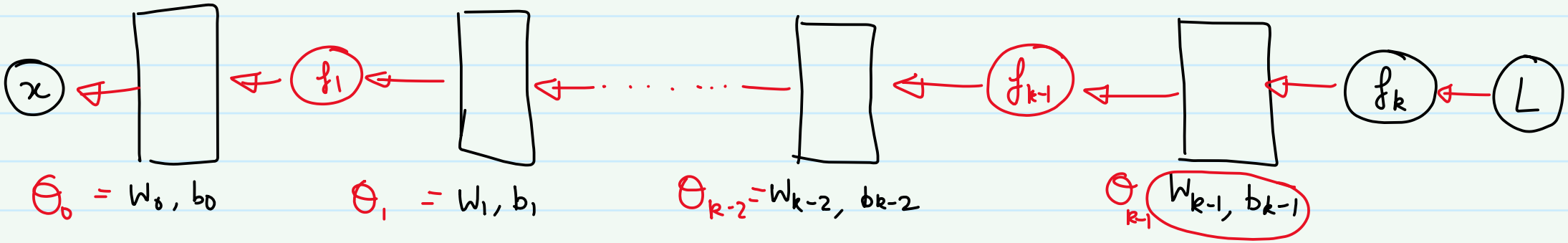
- Backprop iterates between a forward pass and a backward pass



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)

Gradients in a deep neural net



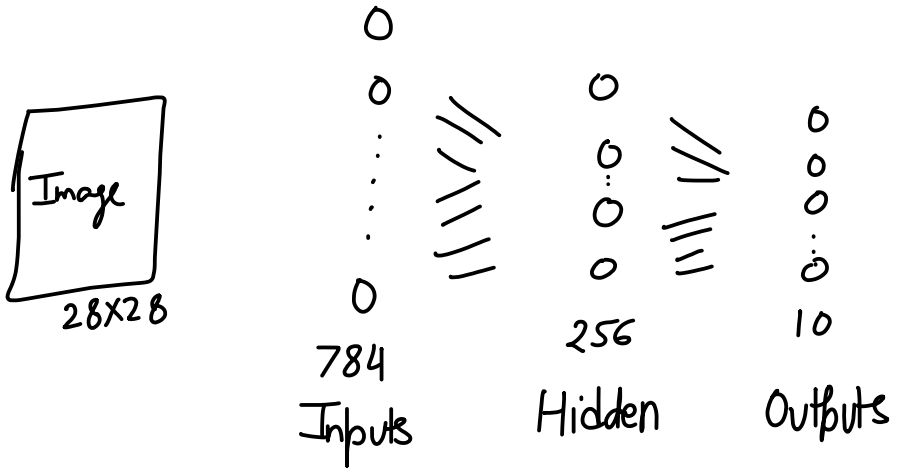


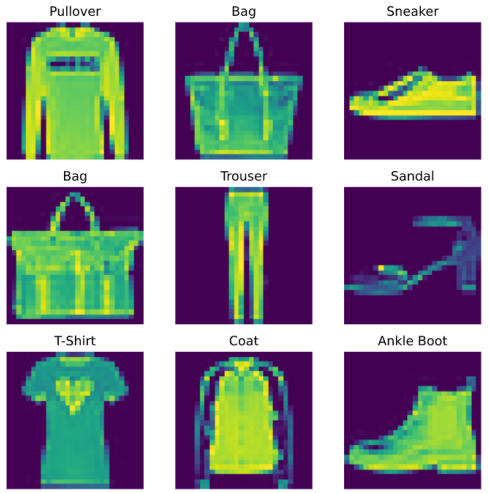
$$\frac{\partial L}{\partial \theta_{k-1}} = \frac{\partial L}{\partial f_k} \cdot \frac{\partial f_k}{\partial \theta_{k-1}}$$

$$\frac{\partial L}{\partial \theta_{k-2}} = \frac{\partial L}{\partial f_k} \cdot \frac{\partial f_k}{\partial f_{k-1}} \cdot \frac{\partial f_{k-1}}{\partial \theta_{k-2}}$$

$$\frac{\partial L}{\partial \theta_i} = \frac{\partial L}{\partial f_k} \cdot \frac{\partial f_k}{\partial f_{k-1}} \cdots \frac{\partial f_{i+2}}{\partial f_{i+1}} \cdot \frac{\partial f_{i+1}}{\partial \theta_i}$$

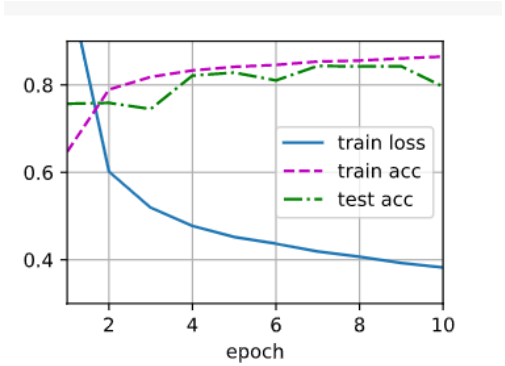
Example of Feed-forward Neural Network






```
num_inputs, num_outputs, num_hiddens = 784, 10, 256

W1 = nn.Parameter(
    torch.randn(num_inputs, num_hiddens, requires_grad=True) * 0.01)
b1 = nn.Parameter(torch.zeros(num_hiddens, requires_grad=True))
W2 = nn.Parameter(
    torch.randn(num_hiddens, num_outputs, requires_grad=True) * 0.01)
b2 = nn.Parameter(torch.zeros(num_outputs, requires_grad=True))

params = [W1, b1, W2, b2]
```

```
def relu(X):
    a = torch.zeros_like(X)
    return torch.max(X, a)


def net(X):
    X = X.reshape((-1, num_inputs))
    H = relu(X @ W1 + b1)  # Here '@' stand
s for matrix multiplication
    return (H @ W2 + b2)


loss = nn.CrossEntropyLoss()


num_epochs, lr = 10, 0.1
updater = torch.optim.SGD(params, lr=lr)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, updater)
```



Convolutional Neural Networks

Consider a fully connected net for processing 1024 x 1024 [1MP] images x 3 channels RGB

# of parameters for each output node = $10^6 \times 3$

Suppose we have 320 nodes ⇒ 10^9 (billion parameters)

Training set size usually 10,000 to 10^5.

Result : Overfitting.

Solution :- Need regularization

ⓐ Reduce connections

ⓑ Share parameters across nodes.

ⓒ Aggregate information

① Convolutional layers : Use local patterns

5X5 Filter example

$$z_{ij} = \sum_{k=0}^{4} \sum_{l=0}^{4} x_{i+k, j+l} w_{kl} + b$$

↳ Can be passed through ReLU

② Pooling : Aggregation over small spatial regions

Compute max of every 2x2 region.

↳ Non overlapping regions

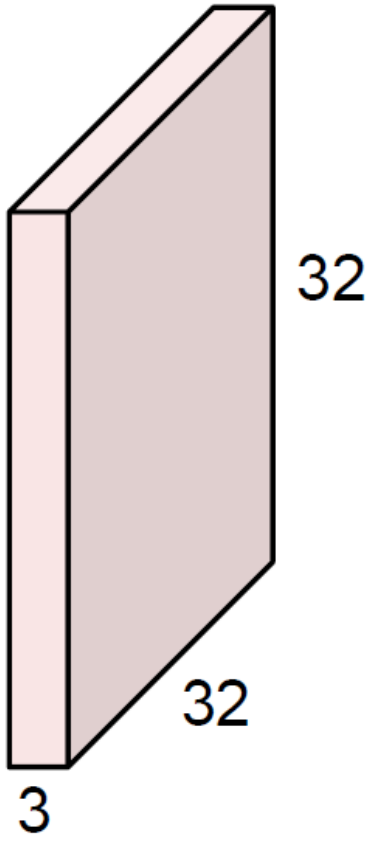
↳ 224 x 224 x 3 ⇒ 112 x 112 x 3

Create Trainable Filters

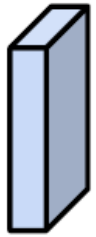
Convolution Layer

Filters always extend the full depth of the input volume

32x32x3 image

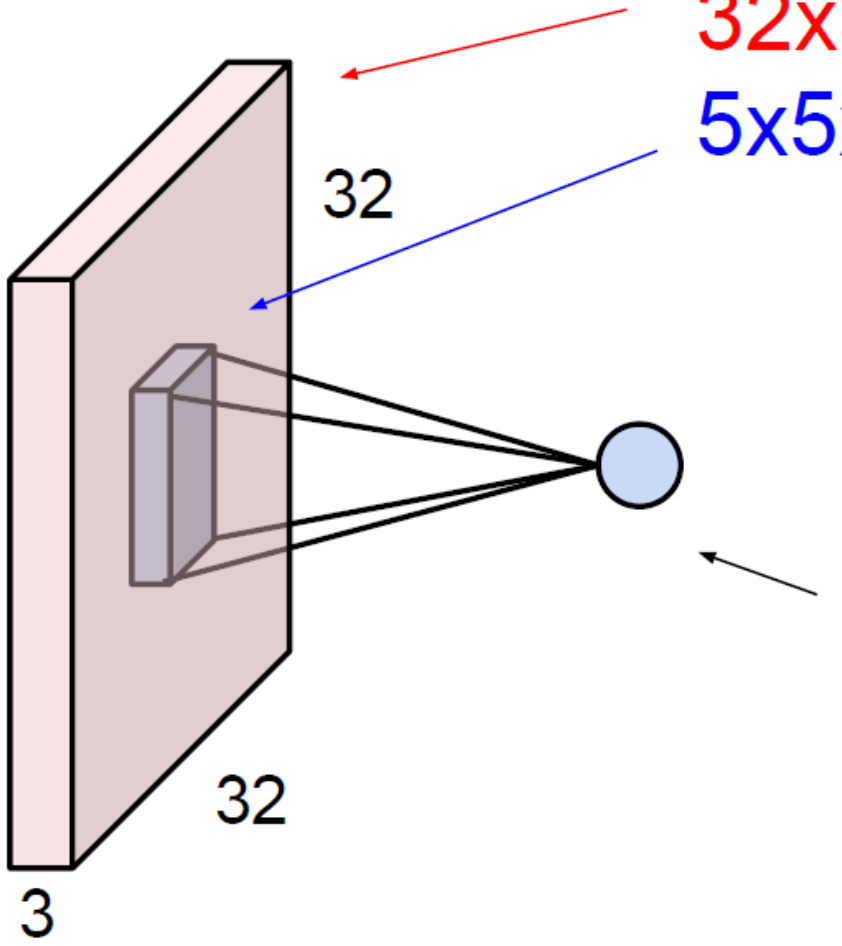


5x5x3 filter



Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”

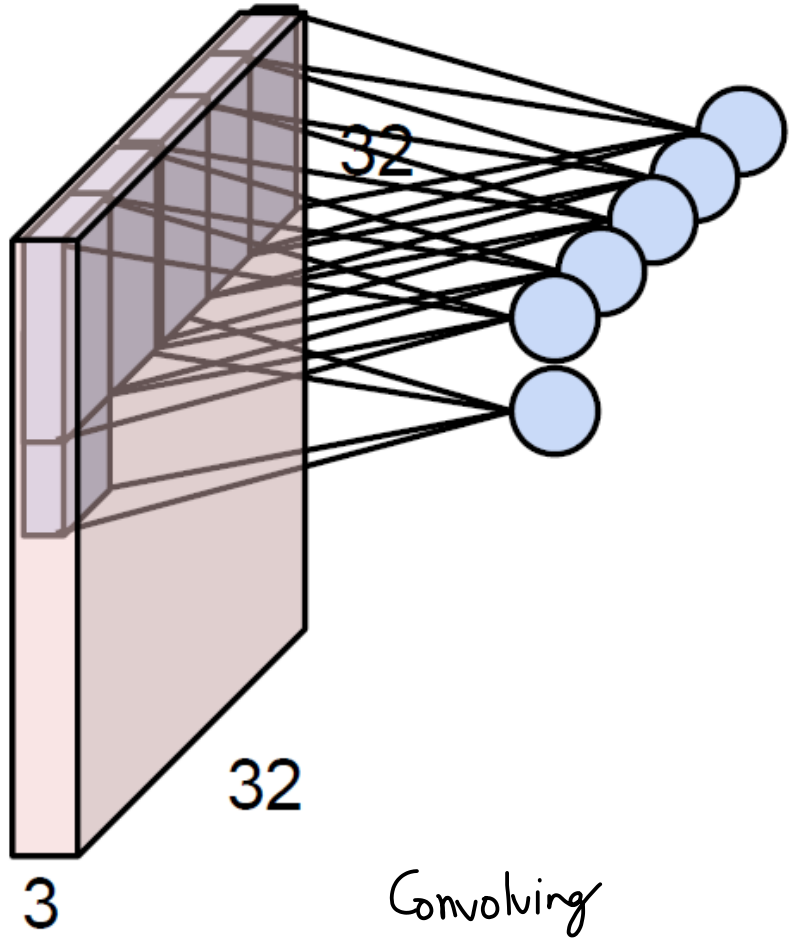
Take Dot Products with the Filter



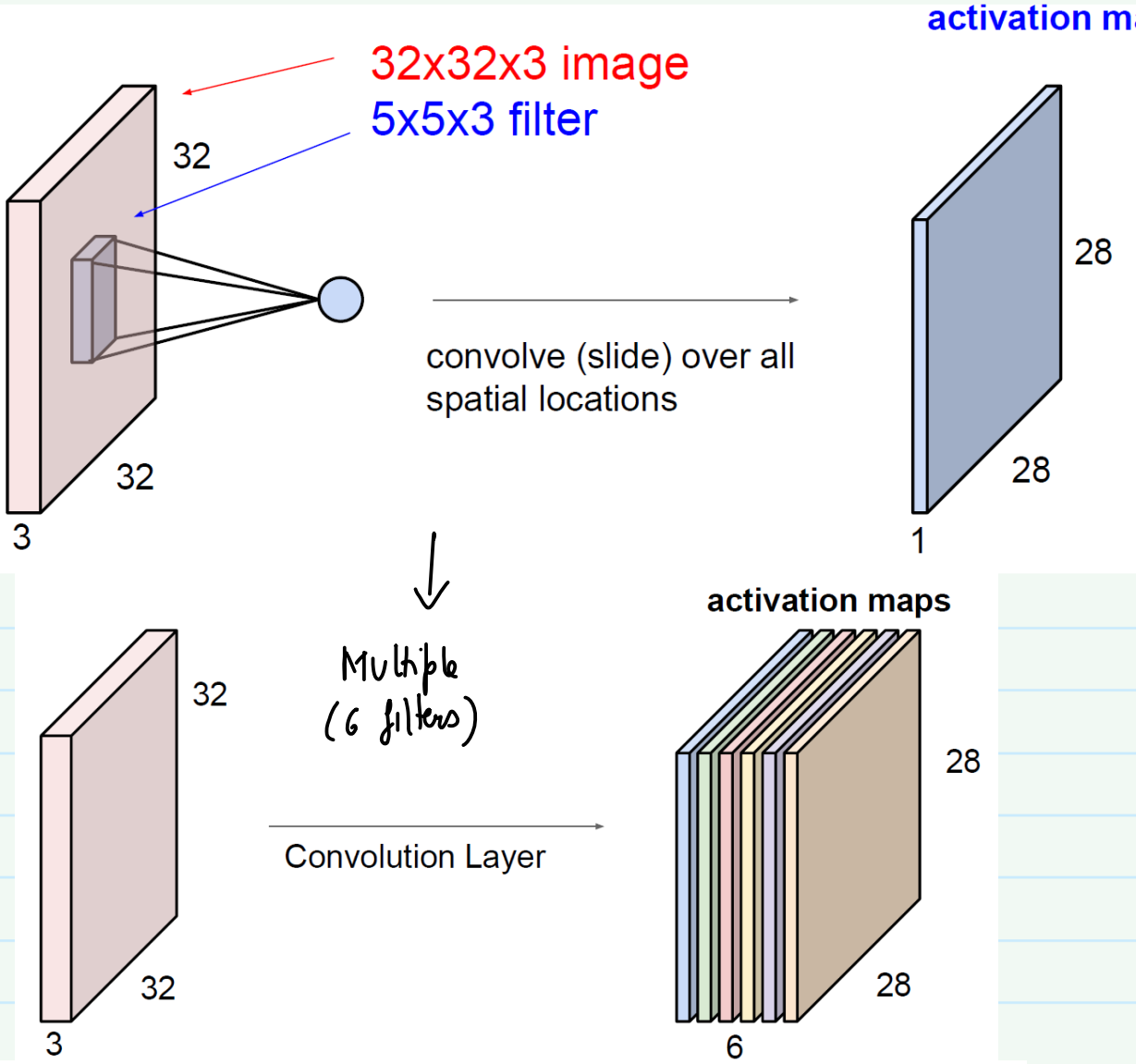
1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

$$w^T x + b$$

Slide the filter all over the image to create maps

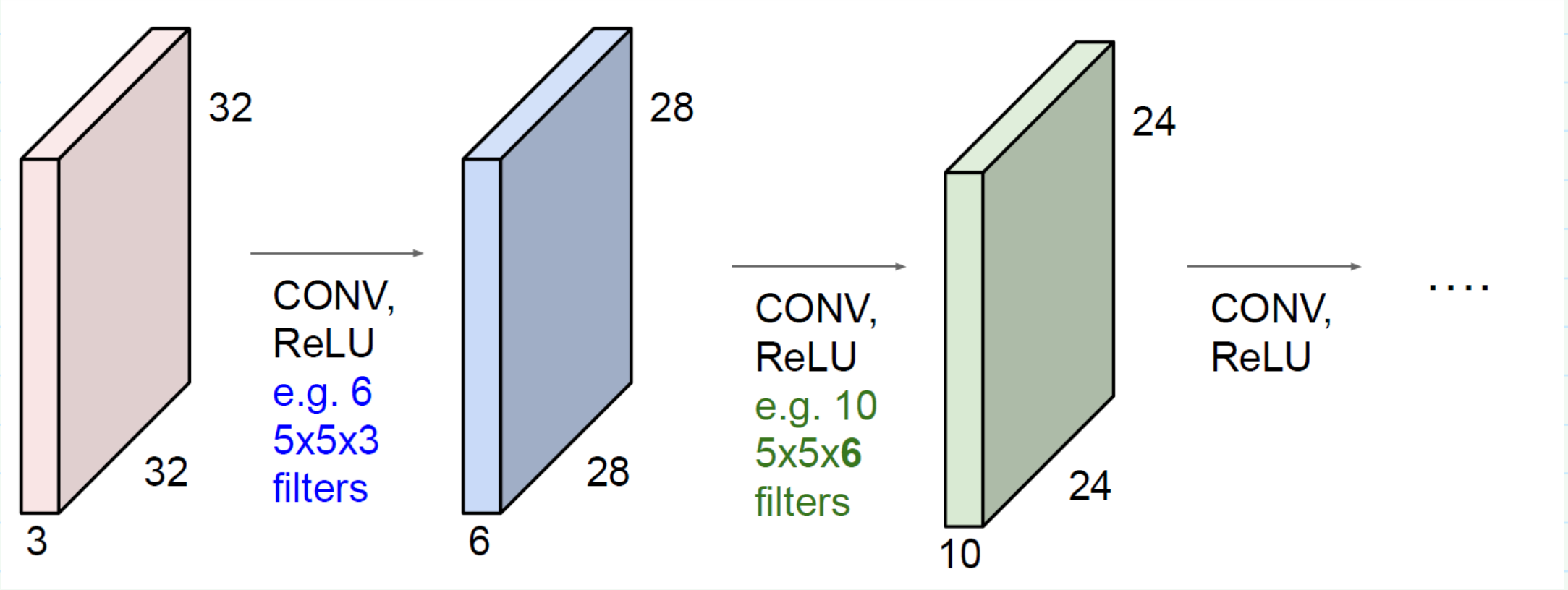


Convolving



We stack these up to get a "new image" of size 28x28x6!

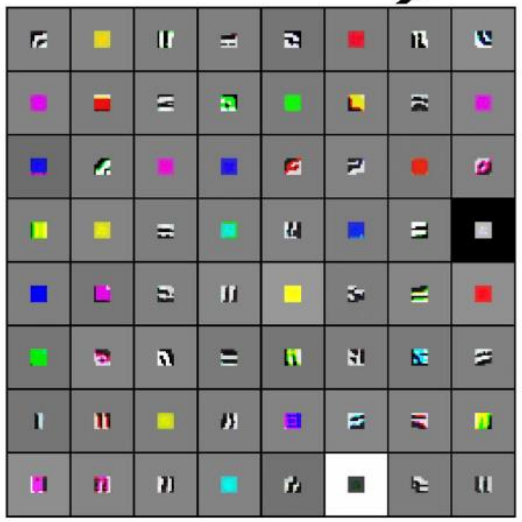
Now we have a recipe!



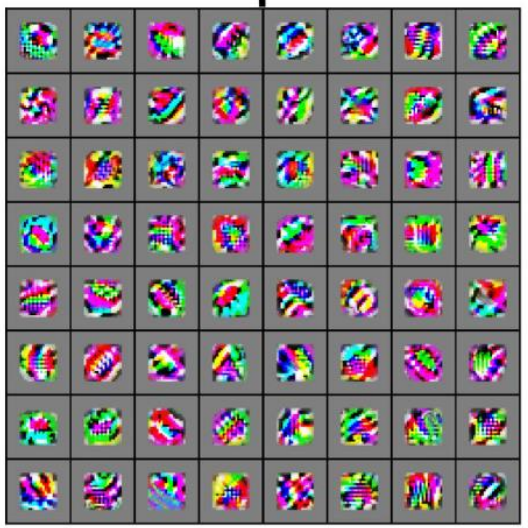
Convolutional Layers + Activation Functions.

VGG-16 Design

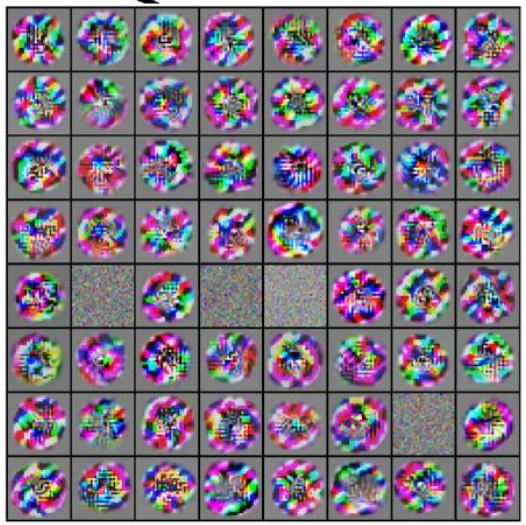
Low-level features → Mid-level features → High-level features → Linearly separable classifier



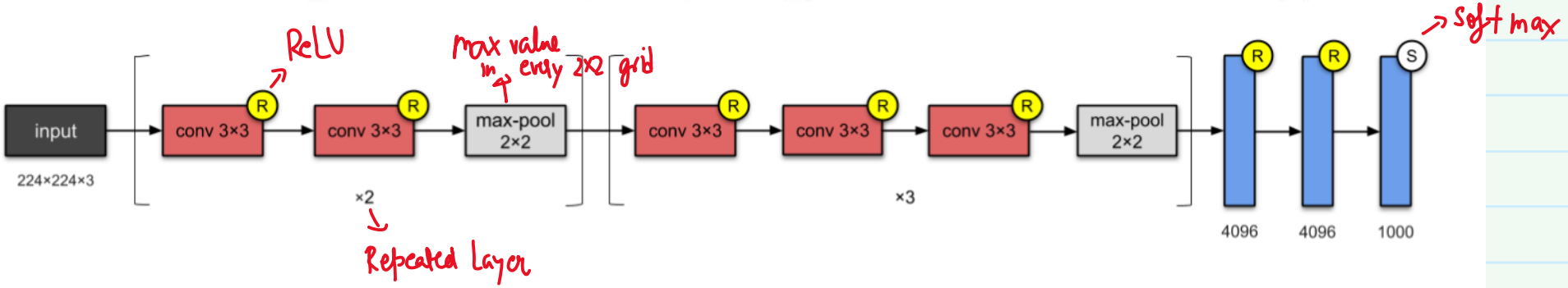
VGG-16 Conv1_1



VGG-16 Conv3_2

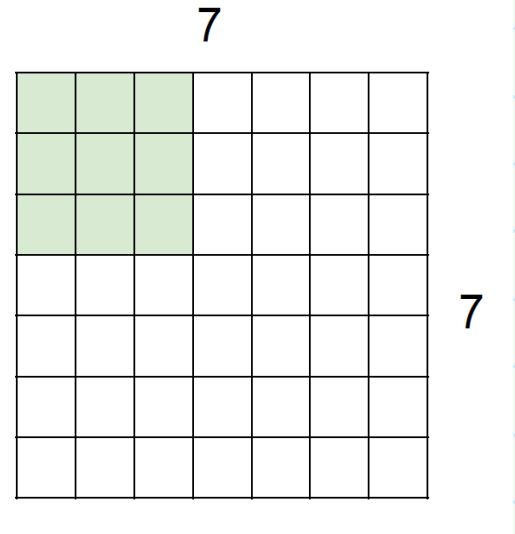


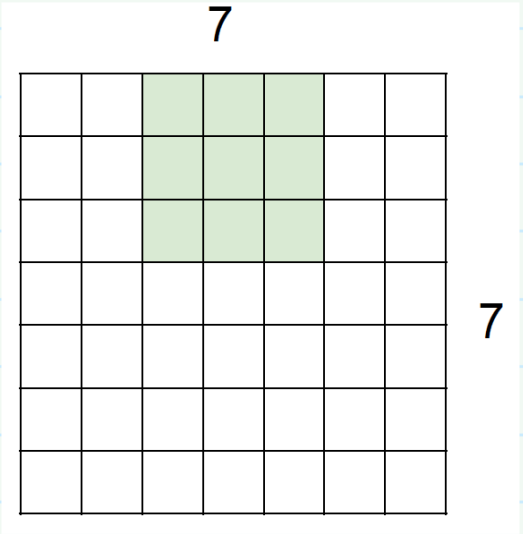
VGG-16 Conv5_3

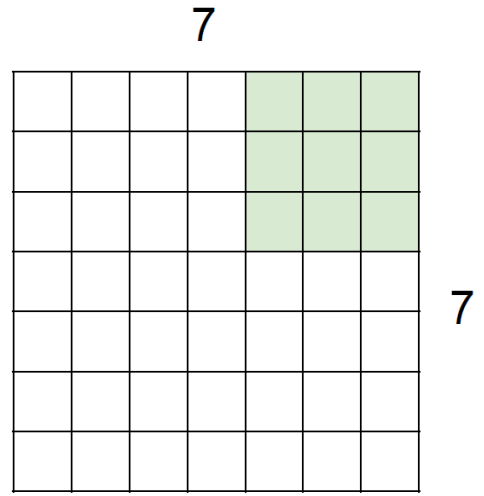


which has 13 convolutional and 3 fully-connected layers, carrying with

Using Strides







7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**
=> 3x3 output!

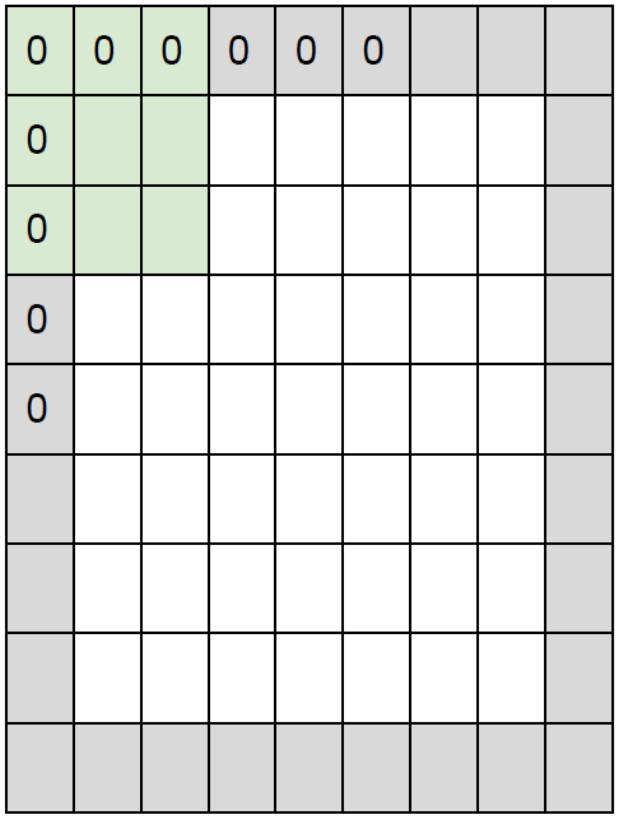
Can we use 3x3 filter with stride 3?

No, doesn't fit!

Output size:
(N - F) / stride + 1

e.g. N = 7, F = 3:
stride 1 => (7 - 3)/1 + 1 = 5
stride 2 => (7 - 3)/2 + 1 = 3
stride 3 => (7 - 3)/3 + 1 = 2.33

Zero Padding the border



e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!

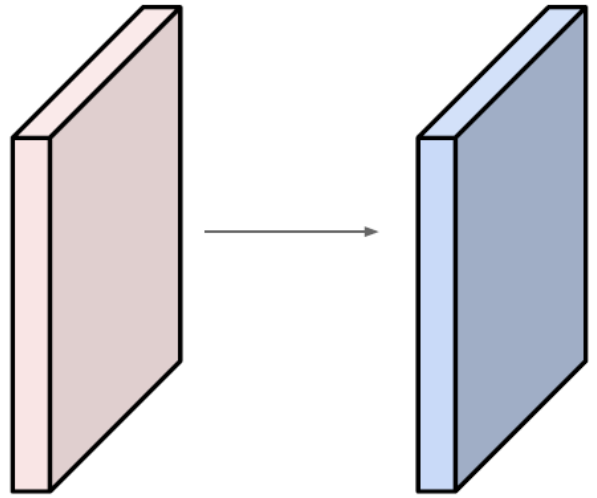
(recall:)
(N + 2P - F) / stride + 1

in general, common to see CONV layers with stride 1, filters of size FxF, and zero-padding with (F-1)/2. (will preserve size spatially)
e.g. F = 3 => zero pad with 1
F = 5 => zero pad with 2
F = 7 => zero pad with 3

Shrinking the image size too fast isn't considered good.

Calculating the number of parameters

Examples time:

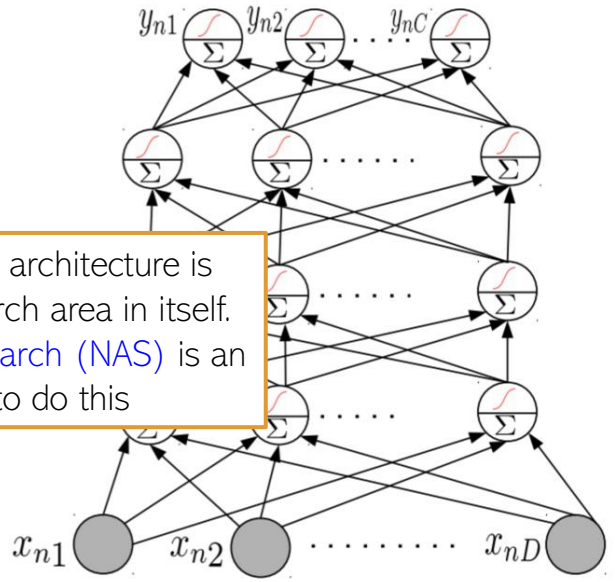


Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?
each filter has 5*5*3 + 1 = 76 params (+1 for bias)
=> 76*10 = **760**

Neural Nets: Some Design Aspects

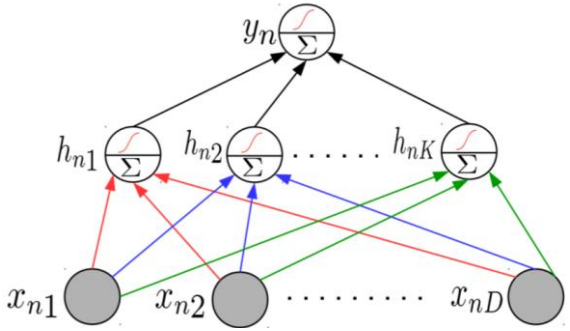
- Much of the magic lies in the hidden layers
- Hidden layers learn and detect good features
- Need to consider a few aspects
  - Number of hidden layers, number of units in each hidden layer
  - Why bother about many hidden layers and not use a single very wide hidden layer (recall Hornik's universal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

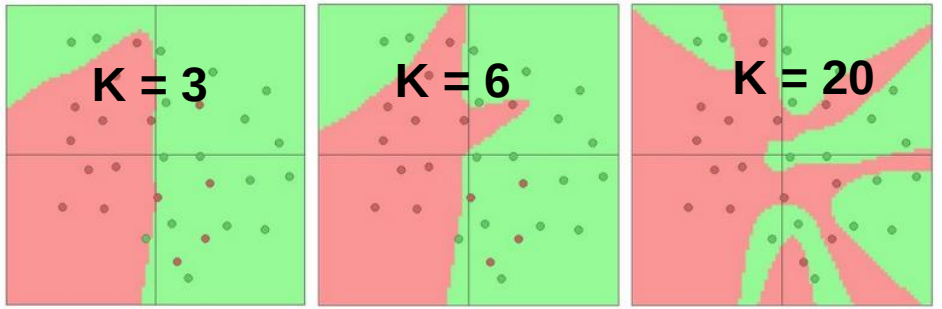


Choosing the right NN architecture is important and a research area in itself. Neural Architecture Search (NAS) is an automated technique to do this

Representational Power of Neural Nets

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





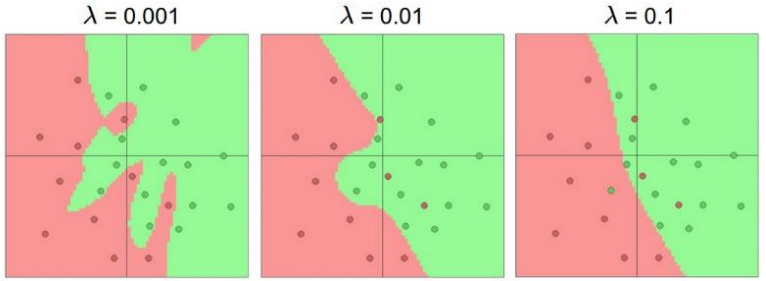
- Recall that each hidden unit "adds" a function to the overall function
- Increasing K (number of hidden units) will result in a more complex function
- Very large K seems to overfit (see above fig). Should we instead prefer small K?
- No! It is better to use large K and regularize well. Reason/justification: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

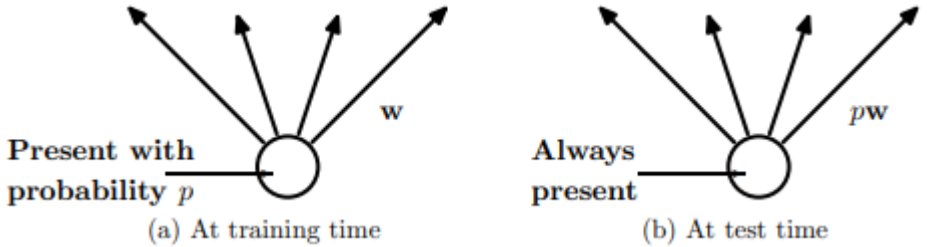
Various other tricks, such as weight sharing across different hidden units of the same layer (used in convolutional neural nets or CNN)

- Neural nets can overfit. Many ways to avoid overfitting, such as
  - Standard regularization on the weights, such as 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



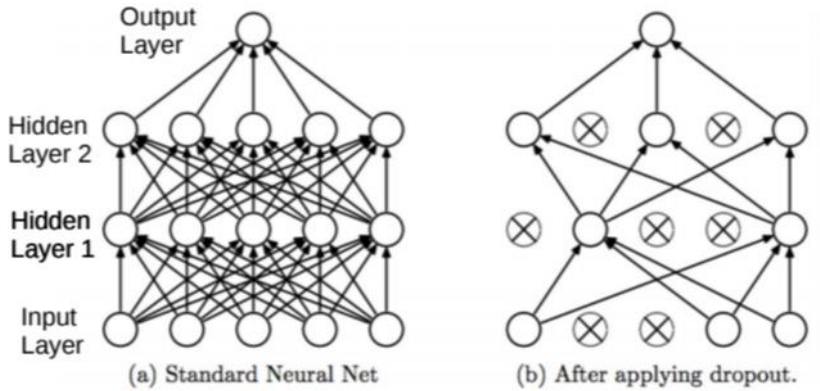
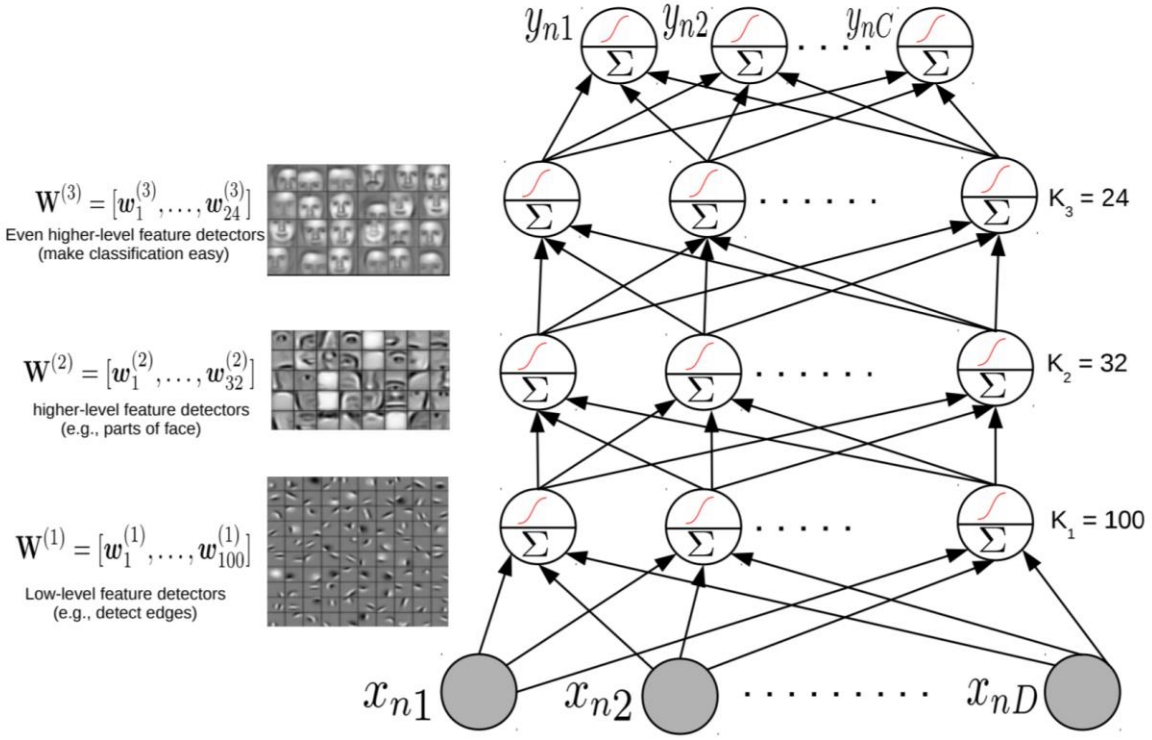


Fig courtesy: Dropout: A Simple Way to Prevent Neural Networks from Overfitting (Srivastava et al, 2014)

Wide or Deep?

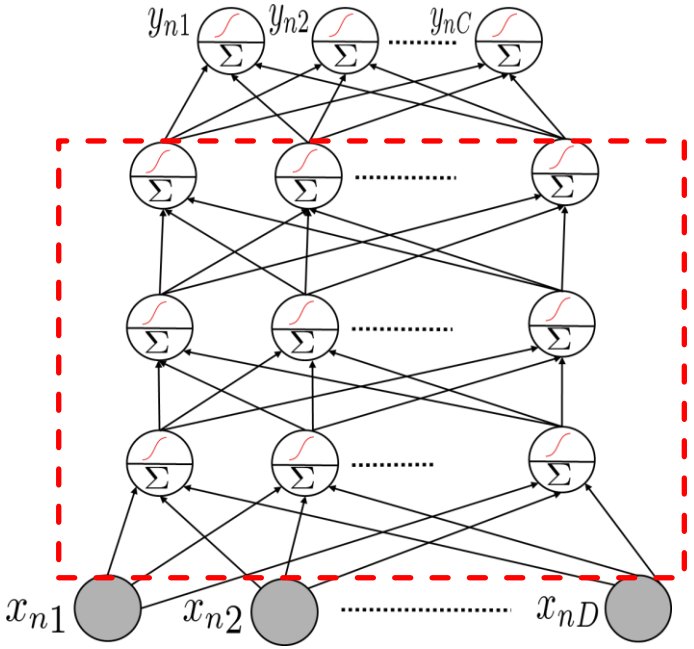
- While very wide single hidden layer can approx. any function, often we prefer many, less wide, hidden layers



- 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

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)

Bonus Question 2.1 Solutions

- $f_1 = \sin(x_1)\cos(x_2), \quad x \in \mathbb{R}^2$

$$\frac{\partial f_1}{\partial x_1} = \cos(x_1)\cos(x_2)$$

$$\frac{\partial f_1}{\partial x_2} = -\sin(x_1)\sin(x_2)$$

$$\implies J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \end{bmatrix} = \begin{bmatrix} \cos(x_1)\cos(x_2) & -\sin(x_1)\sin(x_2) \end{bmatrix} \in \mathbb{R}^{1\times 2}$$