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)

Illustration: Neural Net with One Hidden Layer

- Each input $\boldsymbol{x_n}$ transformed into several pre-activations using linear mo

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

- Nonlinear activation applied on each pre-act.

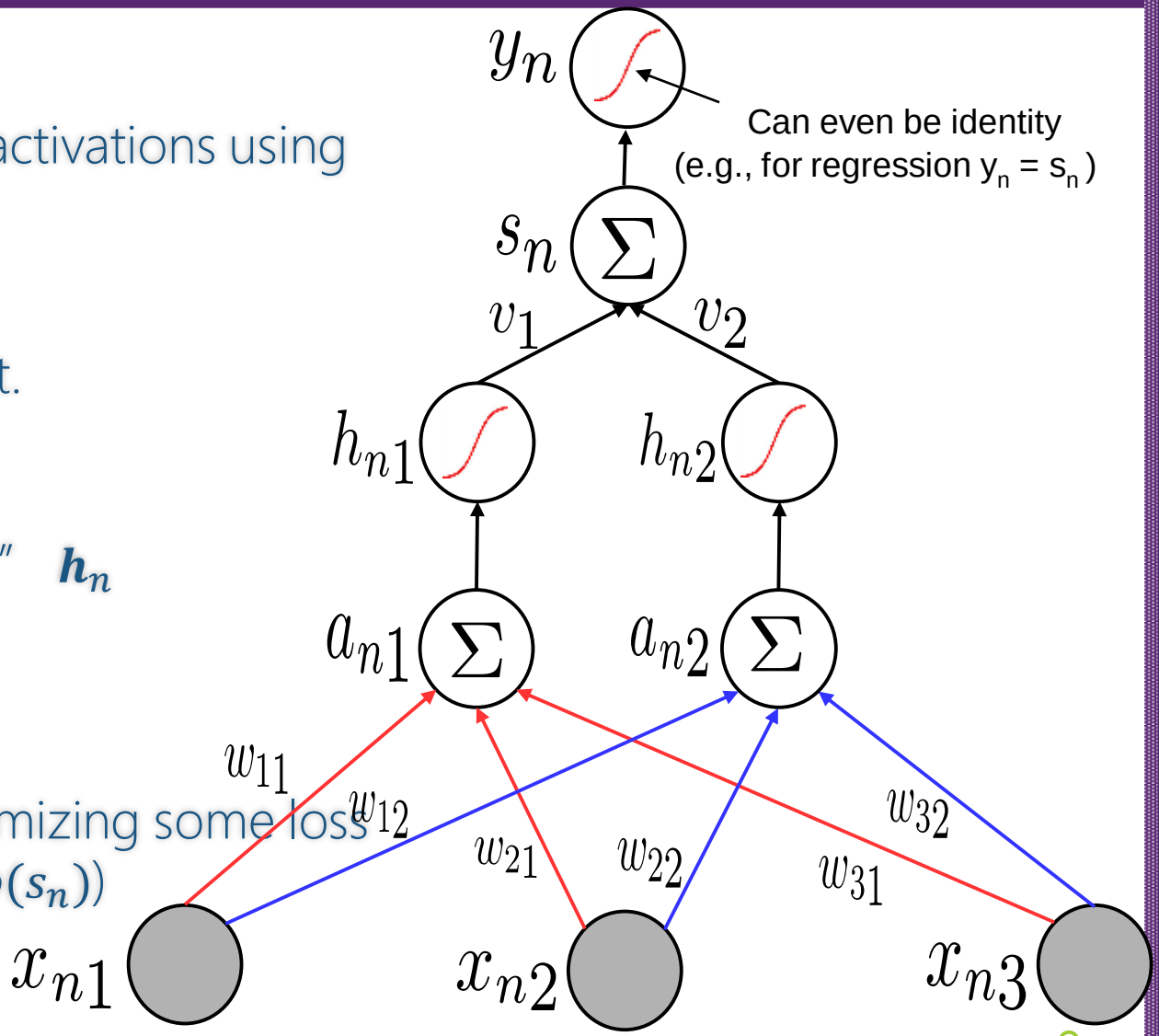
$$h_{nk} = g(a_{nk})$$

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

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

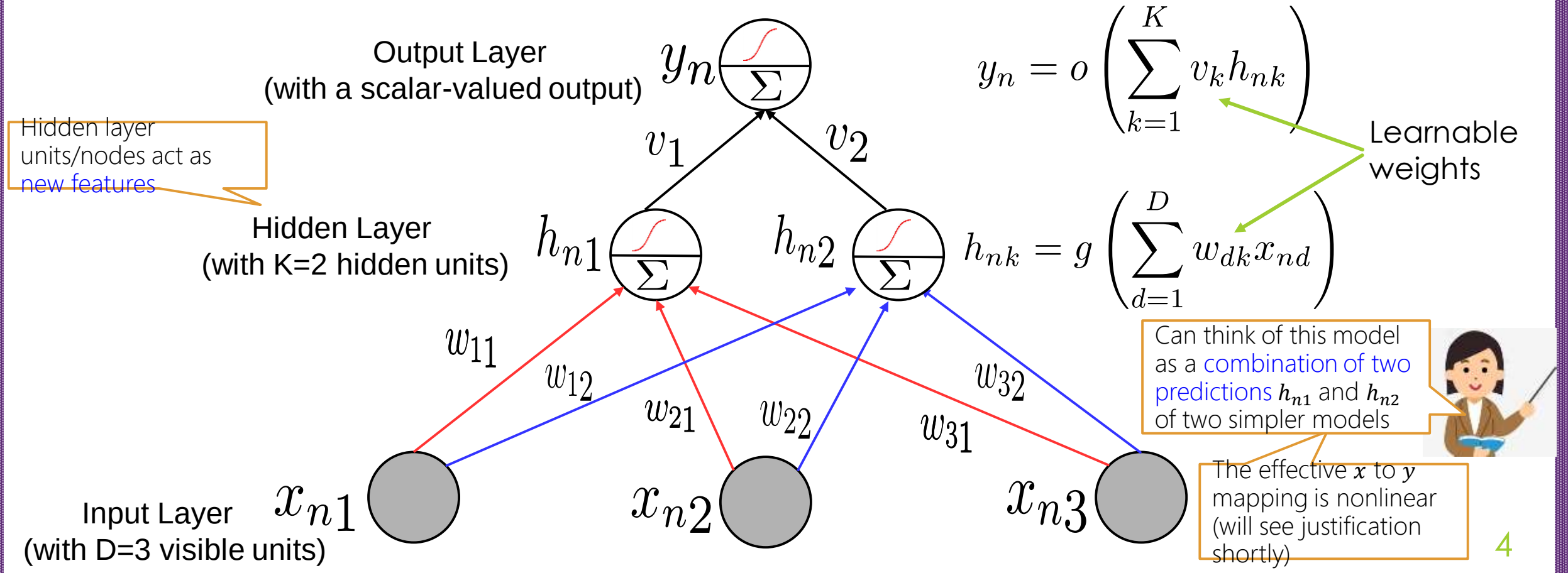
- Finally, output is produced as $\boldsymbol{y = o(s_n)}$
- Unknowns $(\boldsymbol{w_1, w_2, \ldots, w_K, v})$ learned by minimizing some loss function, for example $\boldsymbol{\mathcal{L}(W, v) = \sum_{n=1}^{N} \ell(y_n, o(s_n))}$

(squared, logistic, softmax, etc)



Neural Networks: Multi-layer Perceptron (MLP)

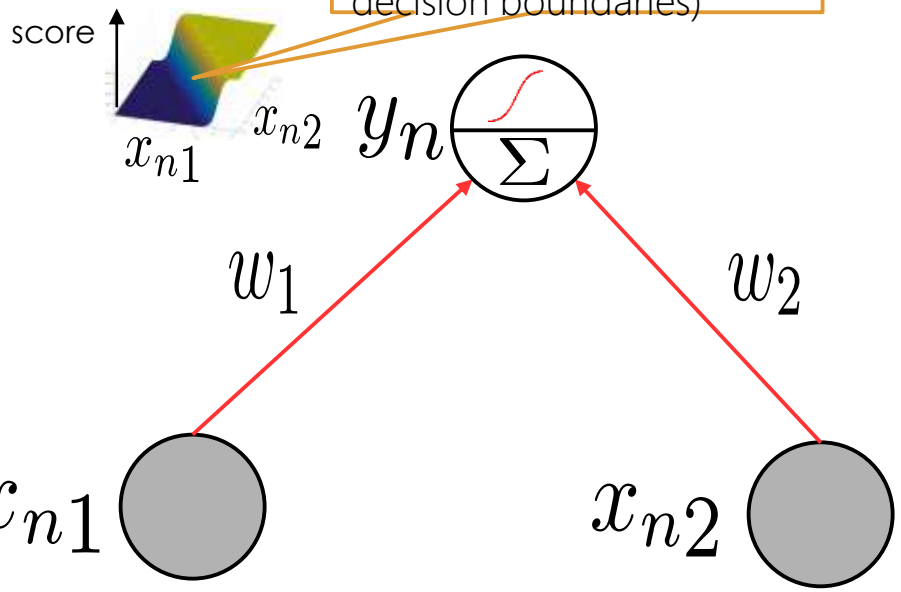
- An MLP consists of an input layer, an output layer, and one or more hidden layers



MLP Can Learn Nonlin. Fn: A Brief Justification

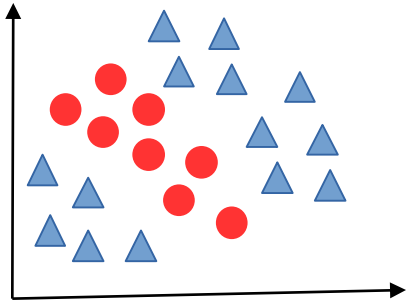
An MLP can be seen as a composition of multiple linear models combined nonlinearly

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



Standard Single "Perceptron" Classifier (no hidden units)

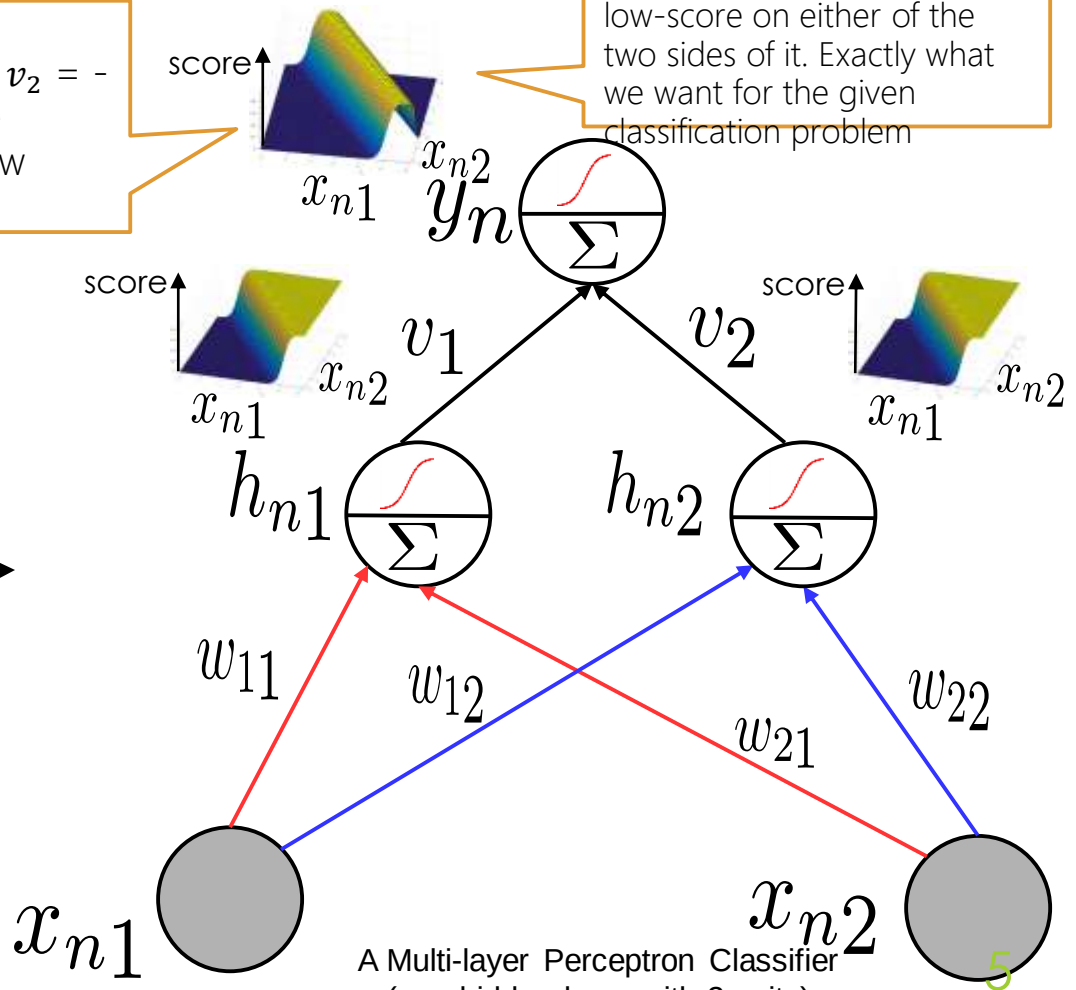
Obtained by composing the two one-sided increasing score functions (using v_1 = 1, and v_2 = -1 to "flip" the second one before adding). This can now learn nonlinear decision boundary



A nonlinear classification problem

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

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

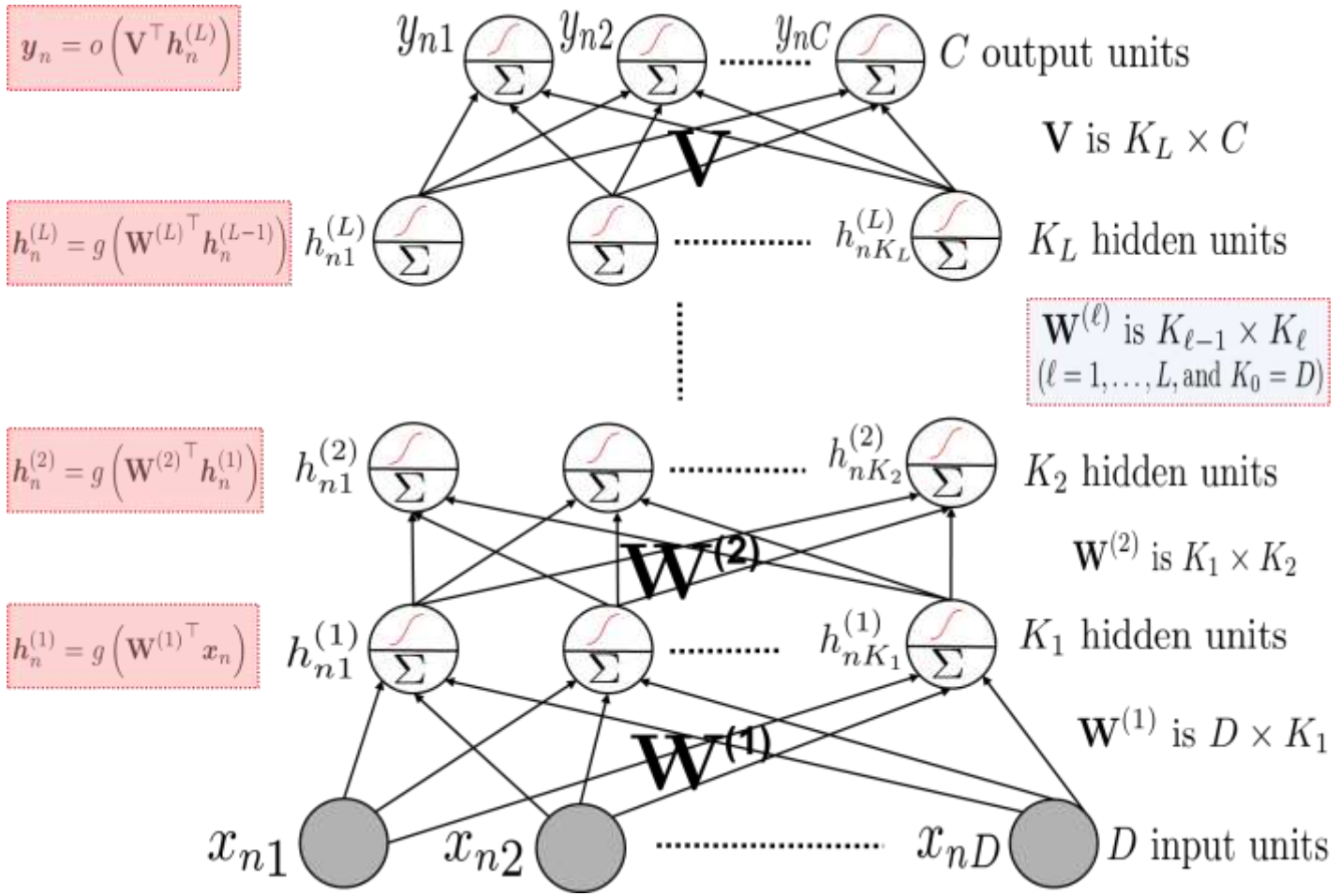


A Multi-layer Perceptron Classifier (one hidden layer with 2 units) Capable of learning nonlinear boundaries

Neural Net Designs

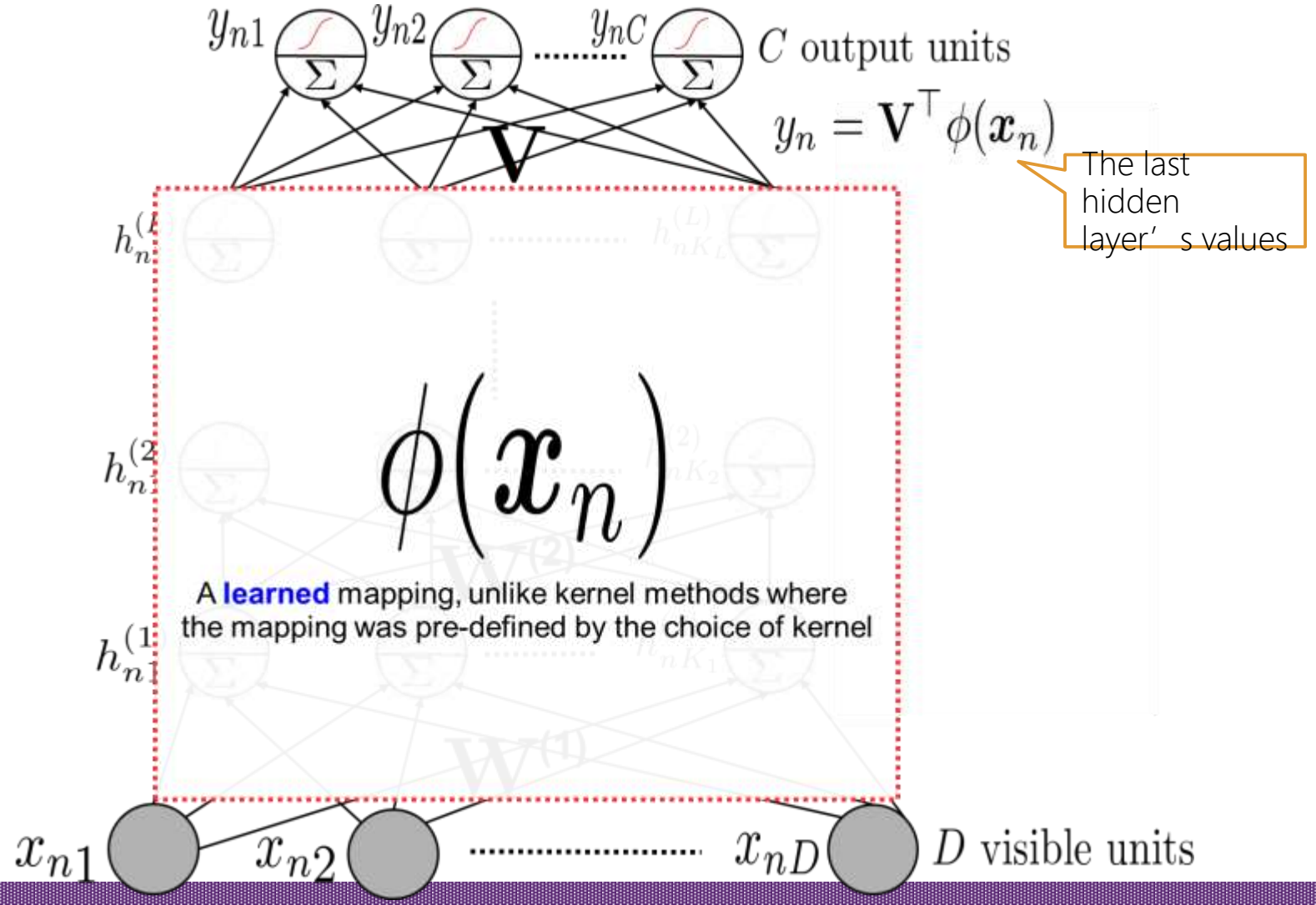
Multiple Hidden Layers (One/Multiple Outputs)

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

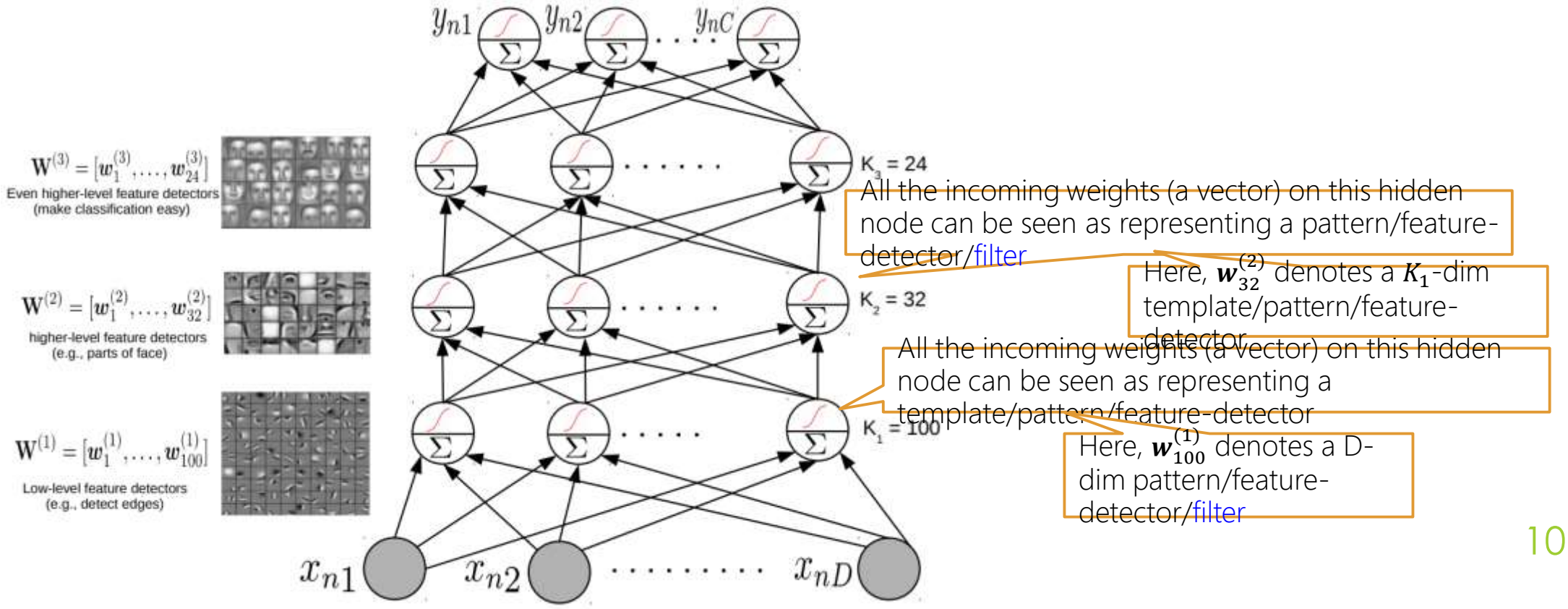
$$y = \sum_{k=1}^{K} v_k h_k$$

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

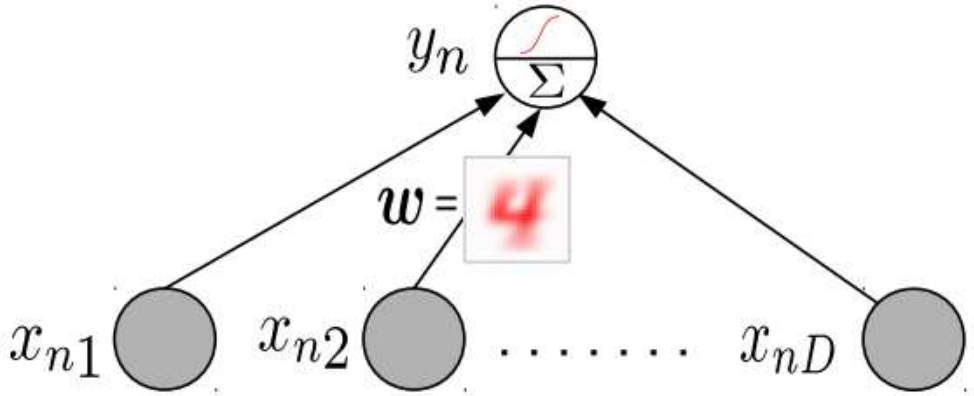
Feature Learned by a Neural Network

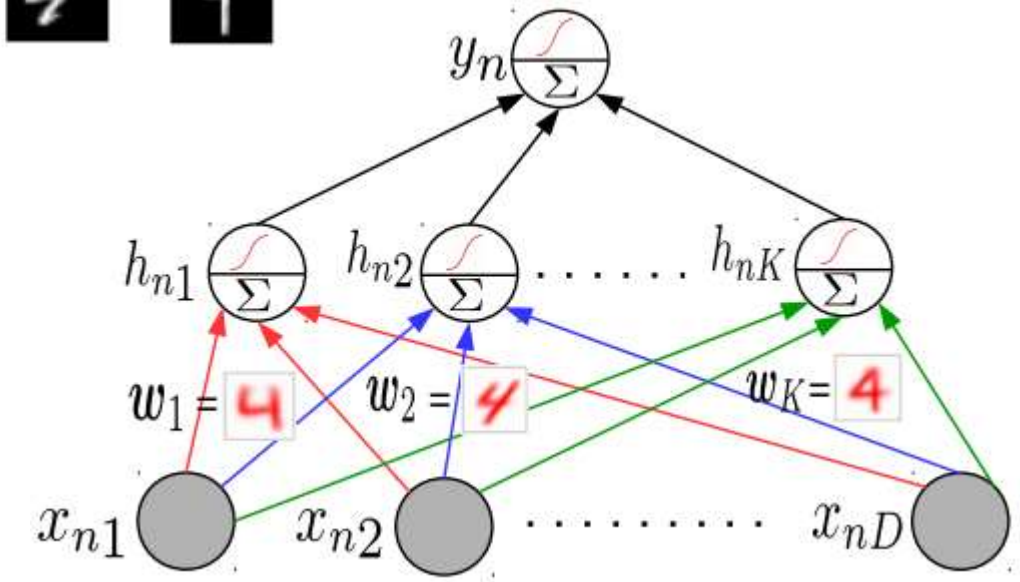
- Node values in each hidden layer tell us how much a "learned" feature is active in $\boldsymbol{x_n}$
- Hidden layer weights are like pattern/feature-detector/filter



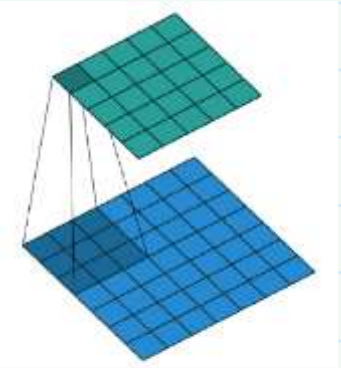
Why Neural Networks Work Better: Another View

- 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



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

Information Flow

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

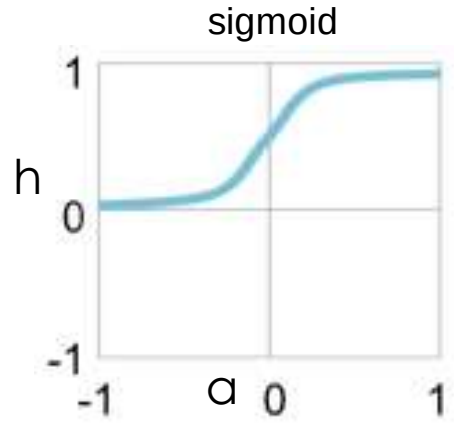
The sign function doesn't satisfy conditions 2, 3.

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

Let's therefore consider other activation functions.

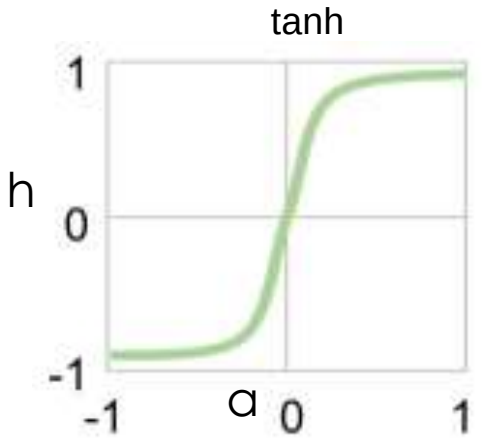
https://en.wikipedia.org/wiki/Dirac_delta_function

Activation Functions: Some Common Choices



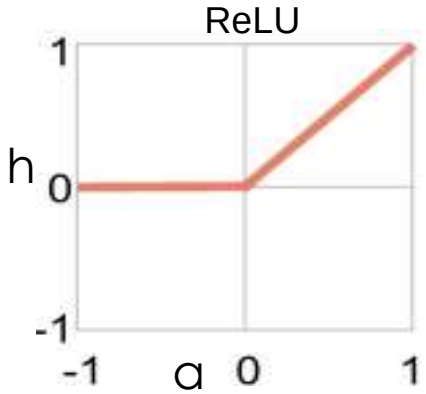
Sigmoid: $h = \sigma(a) = \frac{1}{1+\exp(-a)}$

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



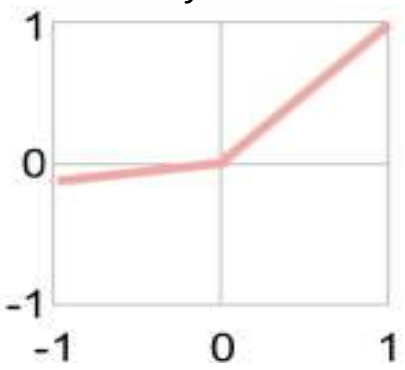
tanh (tan hyperbolic): $h = \frac{\exp(a)-\exp(-a)}{\exp(a)+\exp(-a)} = 2\sigma(2a) - 1$

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): $h = \max(0, a)$

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



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

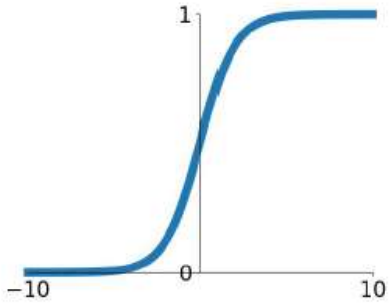
ReLU and Leaky ReLU are among the most popular ones

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

Commonly Used Activation Functions

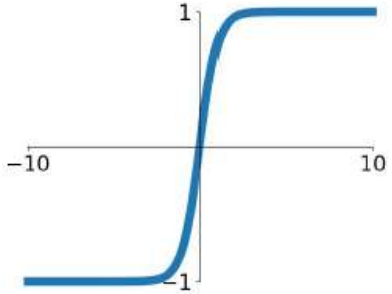
Sigmoid

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



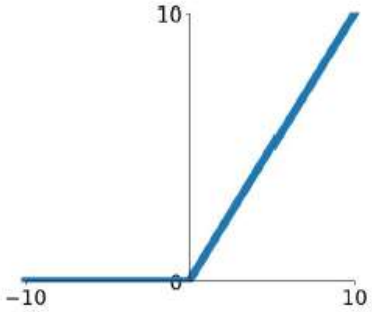
tanh

$$\tanh(x)$$



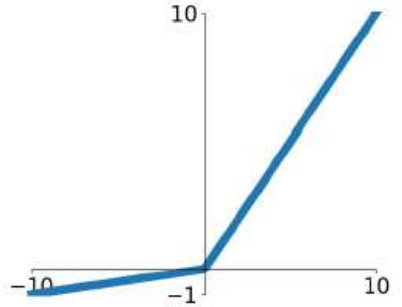
ReLU

$$\max(0, x)$$



Leaky ReLU

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

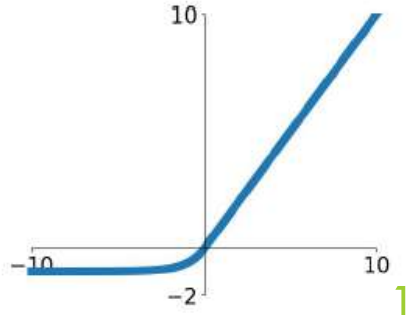


Maxout

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

ELU

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



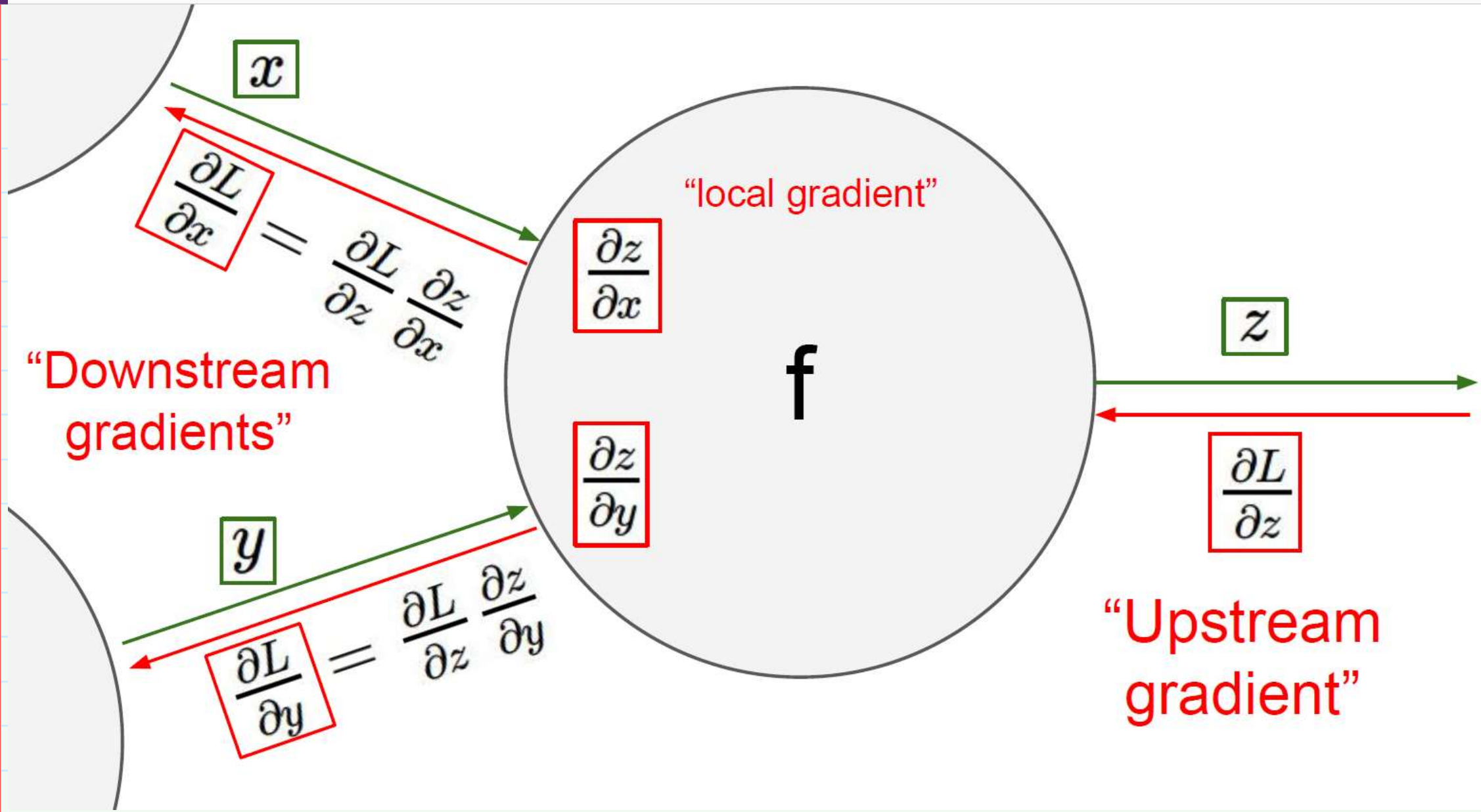
Softmax

Cross Entropy Loss

Compute Graph

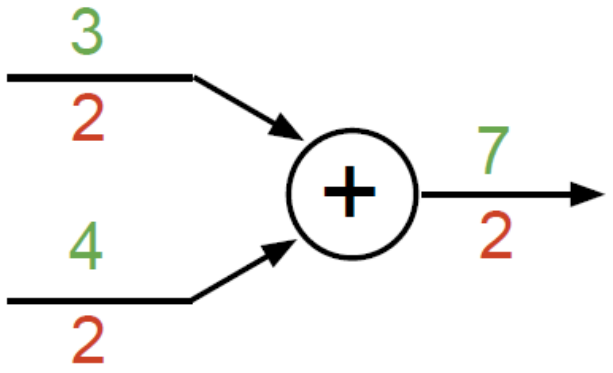
Example of Backprop

Upstream and Local Gradients

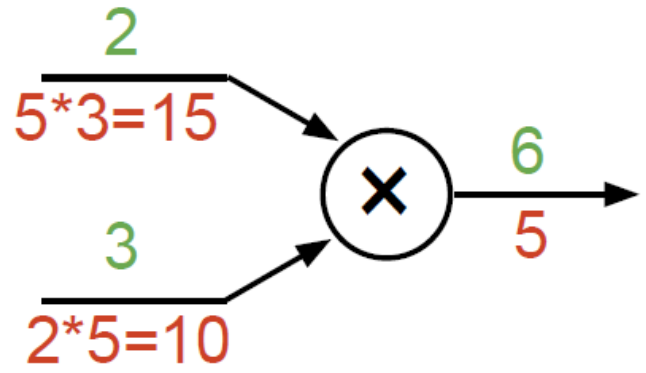


Patterns in Gradient Flow

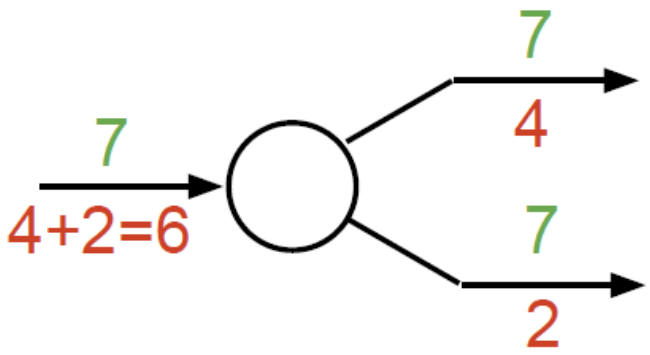
add gate: gradient distributor



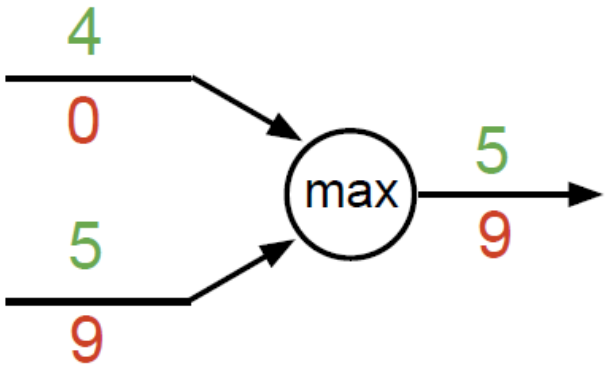
mul gate: "swap multiplier"



copy gate: gradient adder



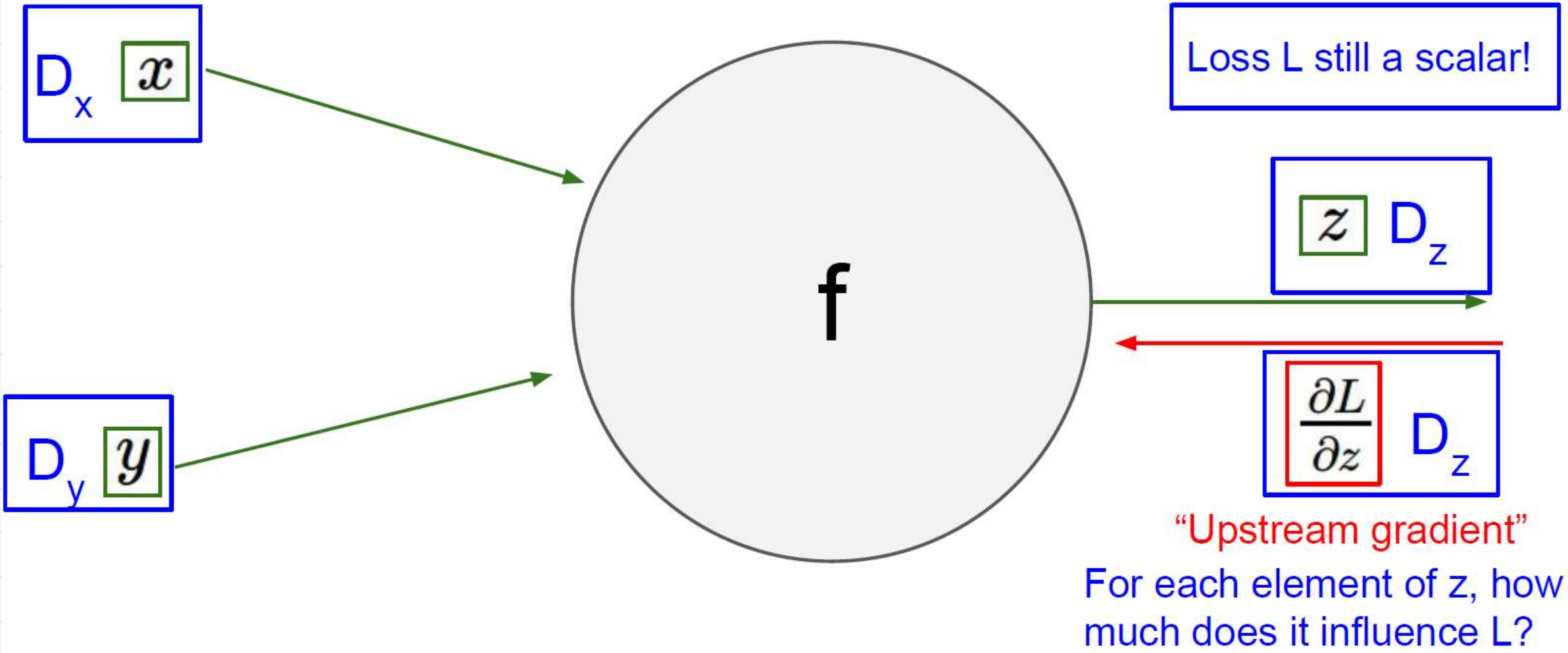
max gate: gradient router



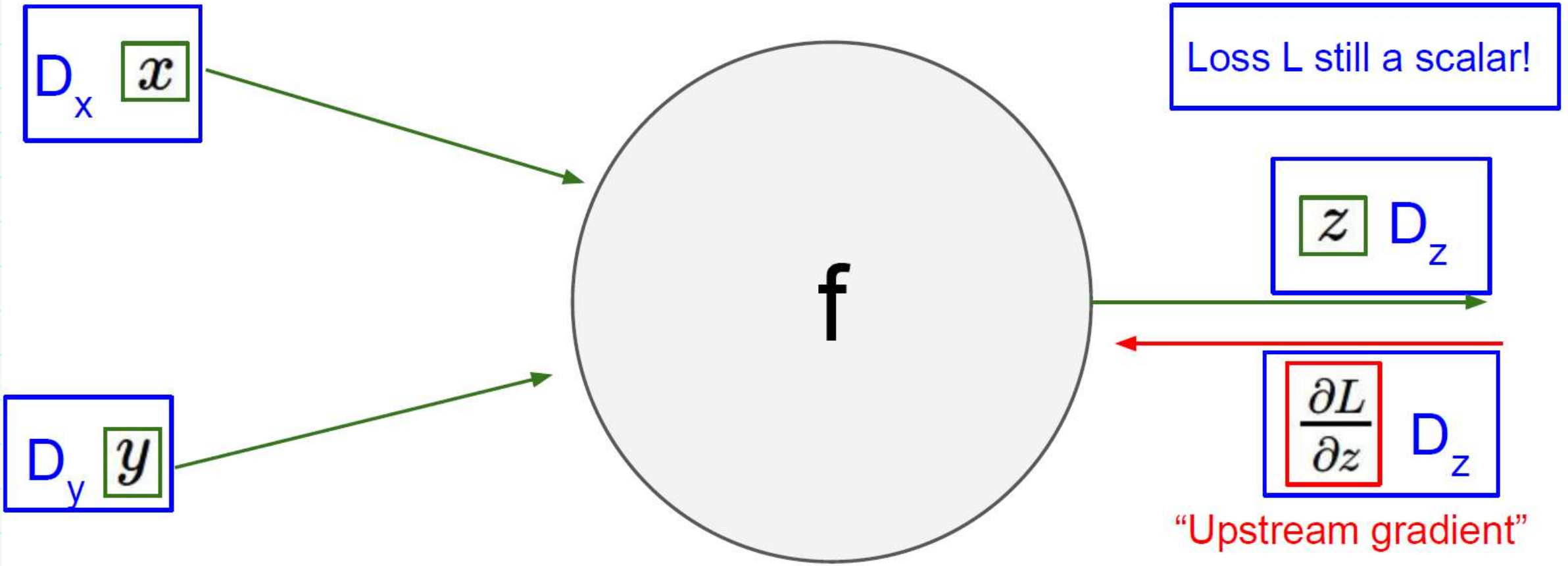
Bonus Question 14.1 (5 marks)

Gradients, Jacobians, and Chain Rule

Backprop with vectors



Backprop with vectors



Backpropagation with vectors

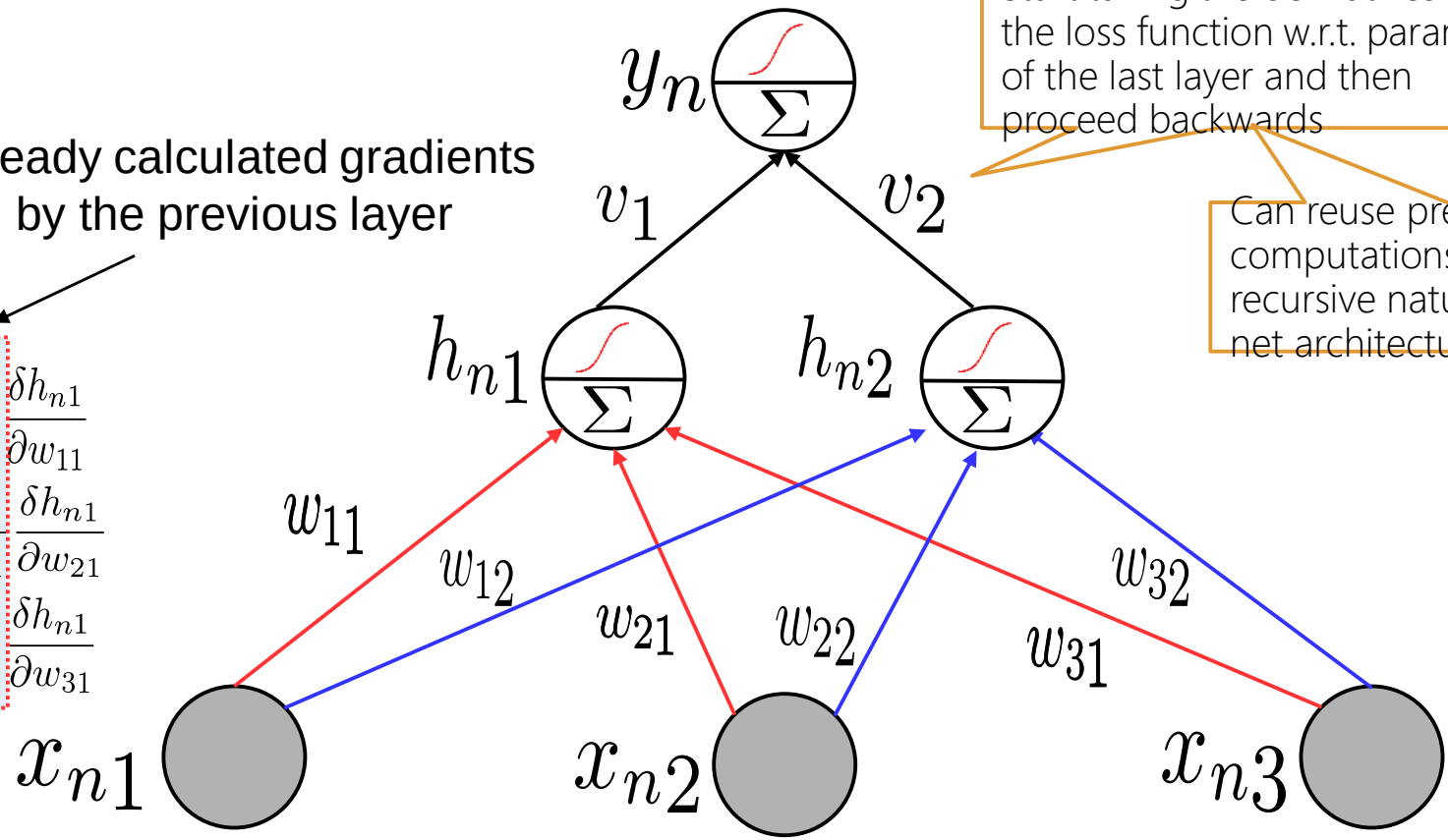
- Backpropagation = Gradient descent using chain rule of derivatives
- Chain rule of derivatives: Example, if $y = f_1(x)$ and $x = f_2(z)$ then $\frac{\partial y}{\partial z} = \frac{\partial y}{\partial x}\frac{\partial x}{\partial z}$

Reuse already calculated gradients computed by the previous layer

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

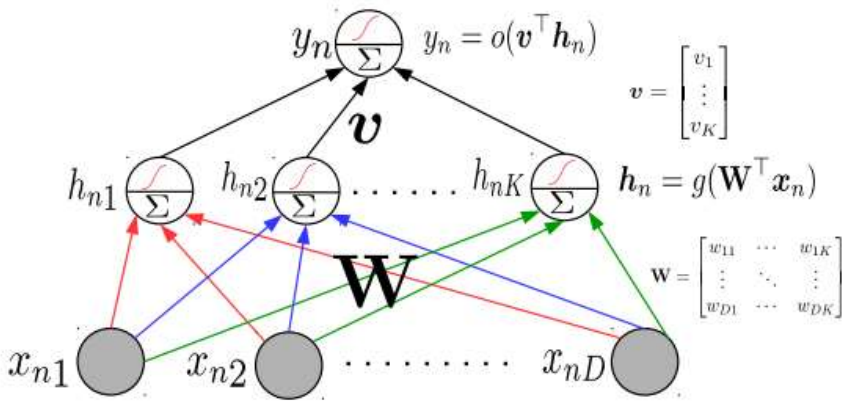
Start taking the derivatives of the loss function w.r.t. params of the last layer and then proceed backwards

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

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

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

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

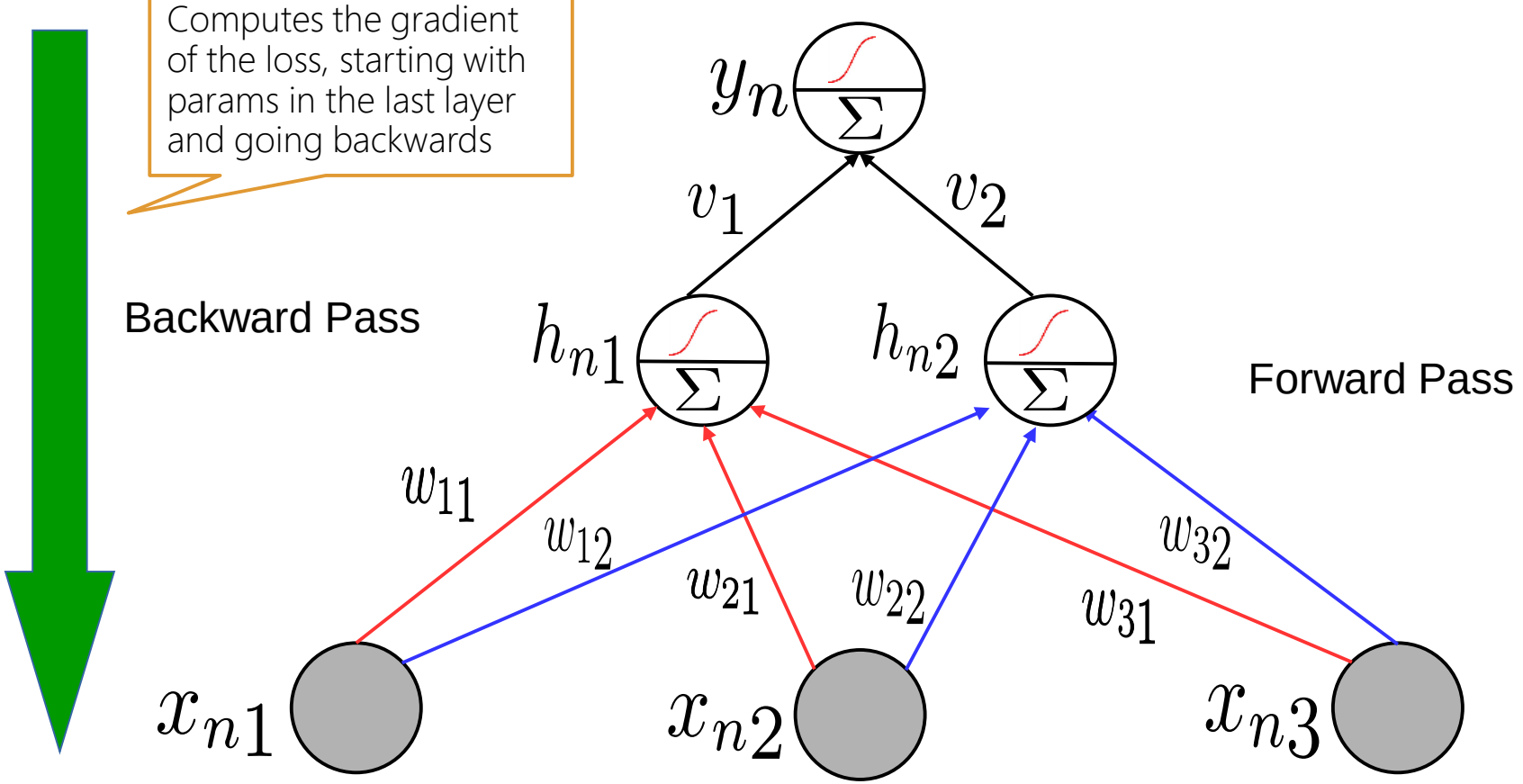
- Forward prop computes errors $\boldsymbol{e}_n$ using current $\mathbf{W}$, $\boldsymbol{v}$. Backprop updates NN params $\mathbf{W}$, $\boldsymbol{v}$ using grad methods
- Backprop caches many of the calculations for reuse

Backpropagation

- Backprop iterates between a forward pass and a backward pa

Computes loss using current values of the parameters

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



Using computational graphs

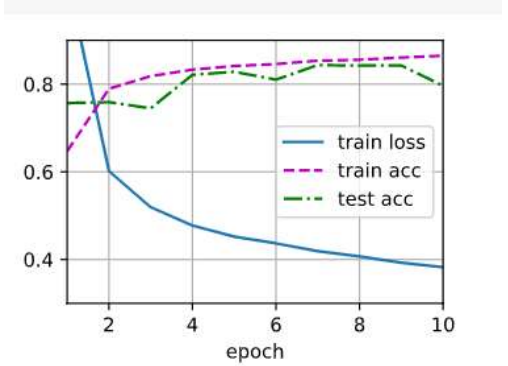
Software frameworks such as Tensorflow and PyTorch support this already so you don't need to implement it by hand (so no worries of computing derivatives etc)

Gradients in a deep neural net

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)
```



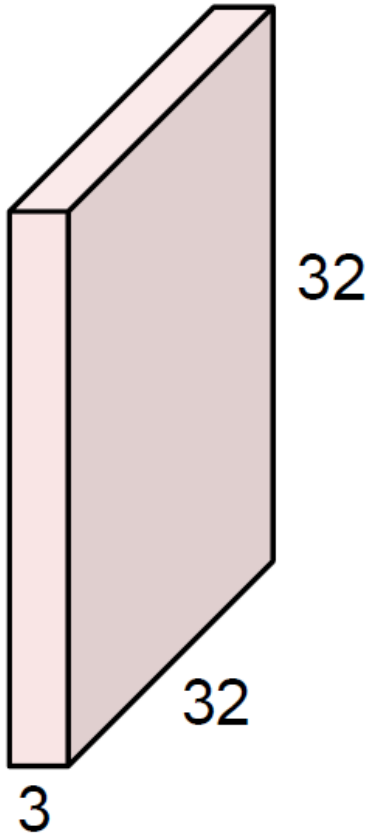
Convolutional Neural Networks

Create Trainable Filters

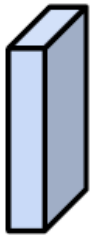
Convolution Layer

Filters always extend the full depth of the input volume

32x32x3 image

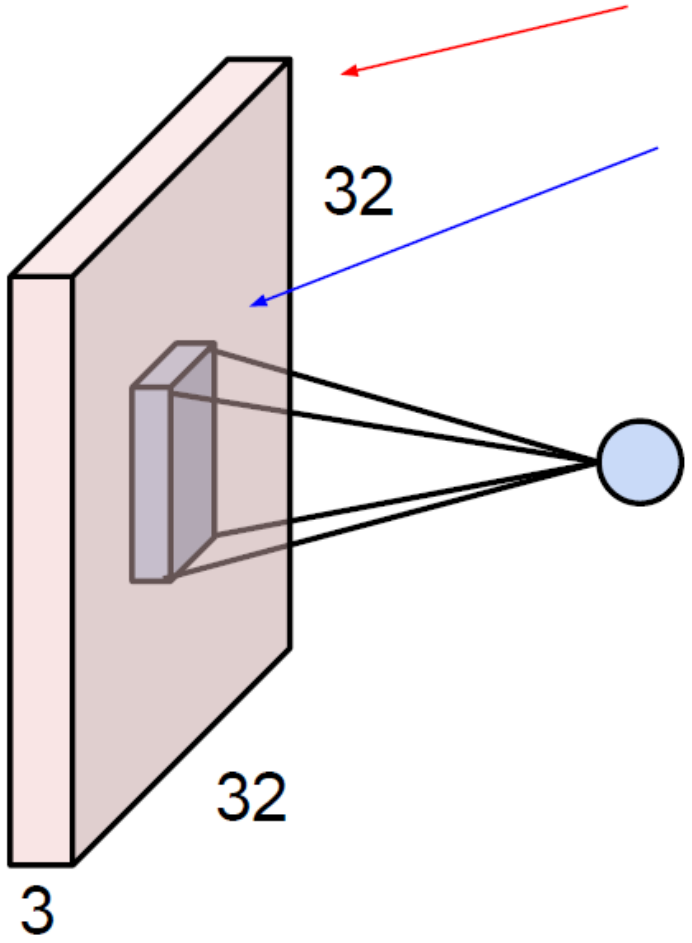


5x5x3 filter



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

Take Dot Products with the Filter



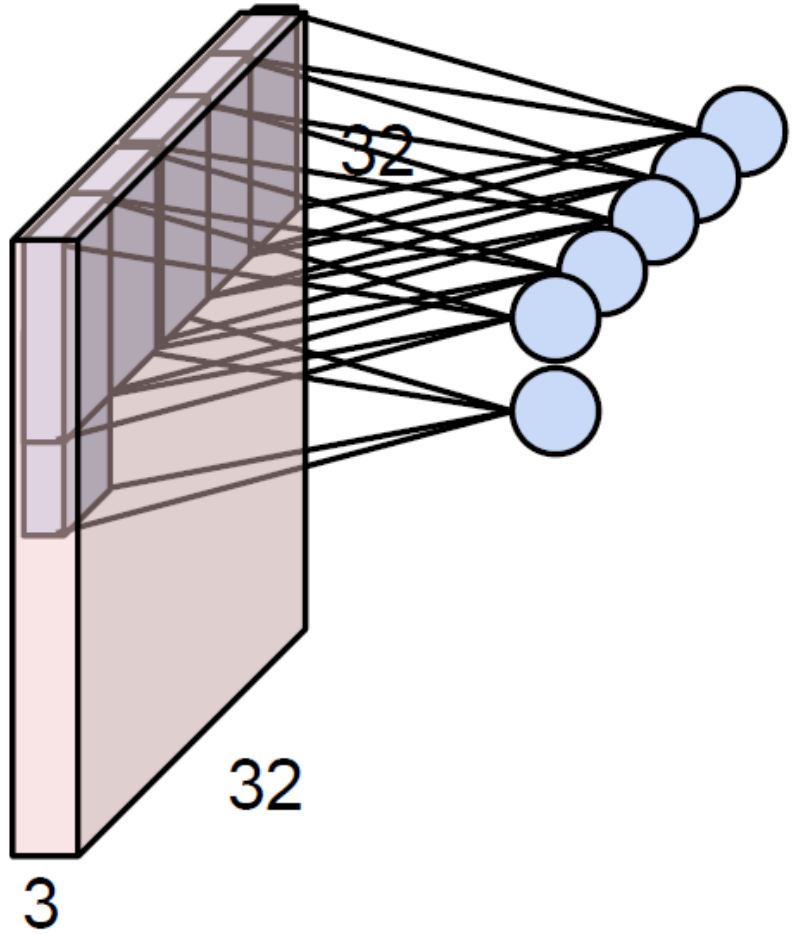
32x32x3 image

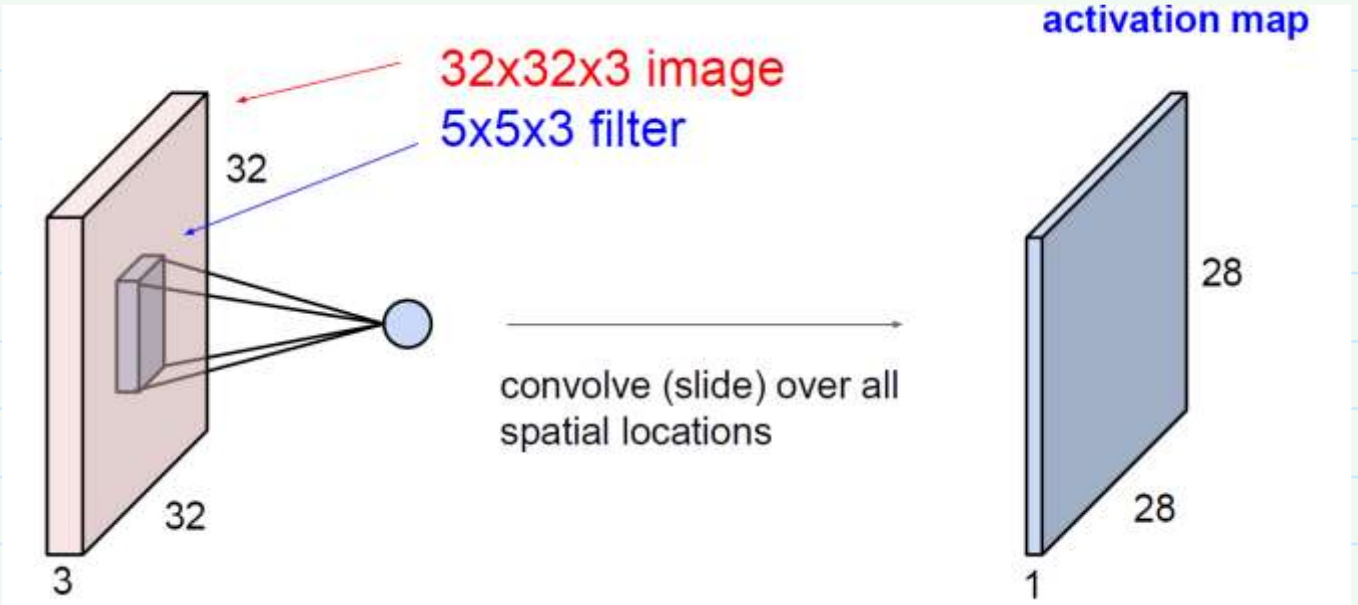
5x5x3 filter w

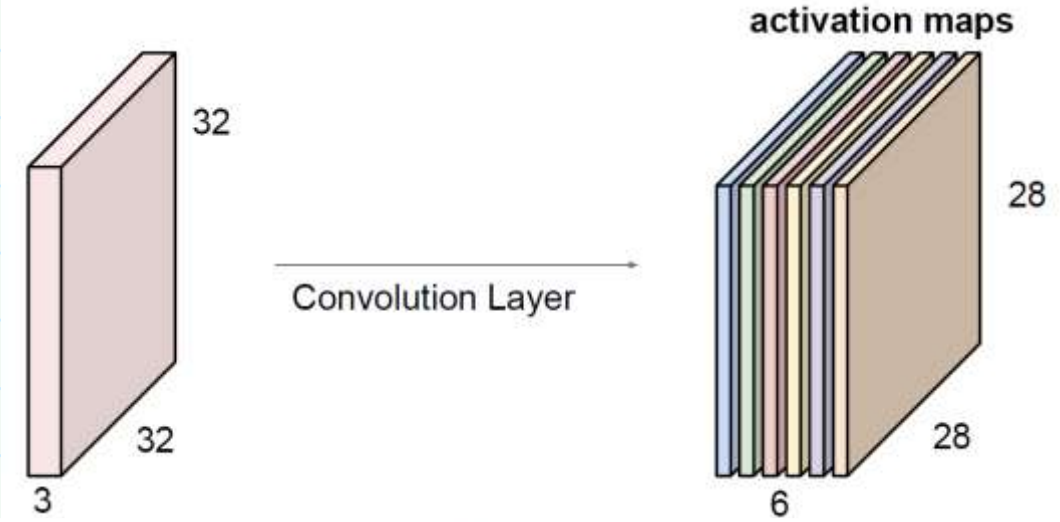
1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image (i.e. 5*5*3 = 75-dimensional dot product + bias)

$$w^T x + b$$

Slide the filter all over the image to create maps

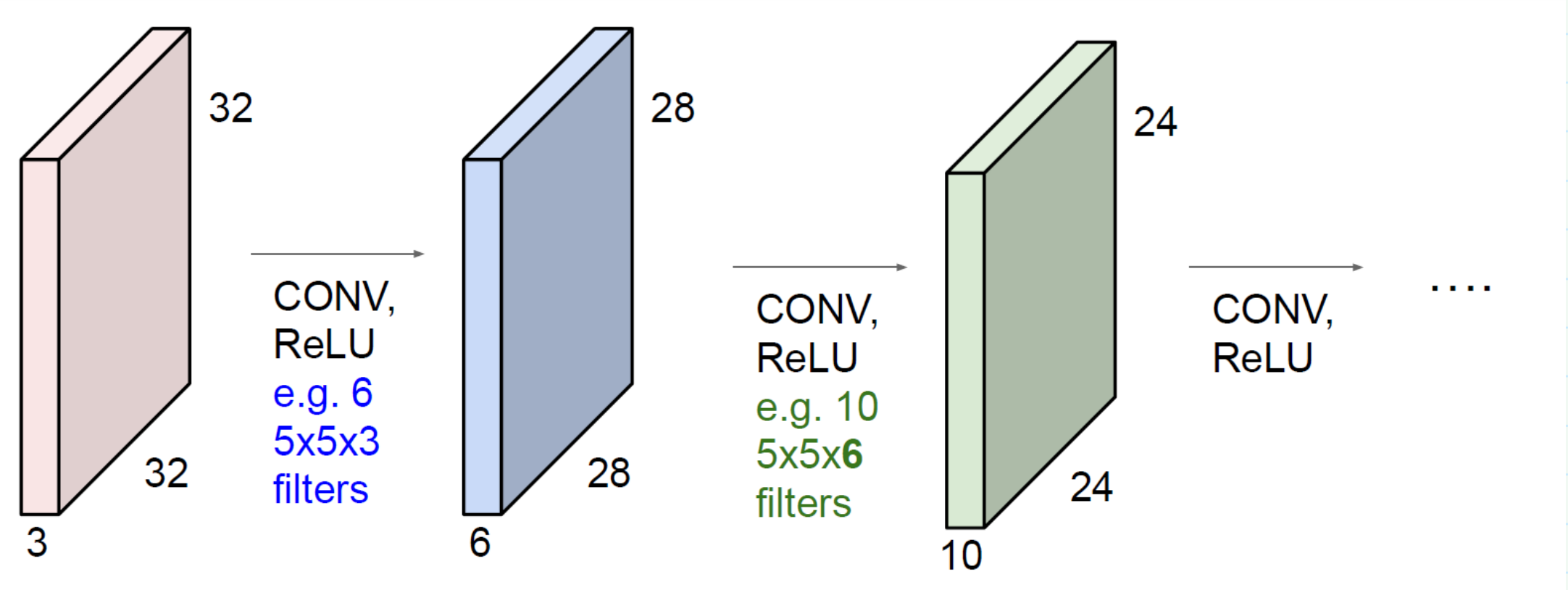




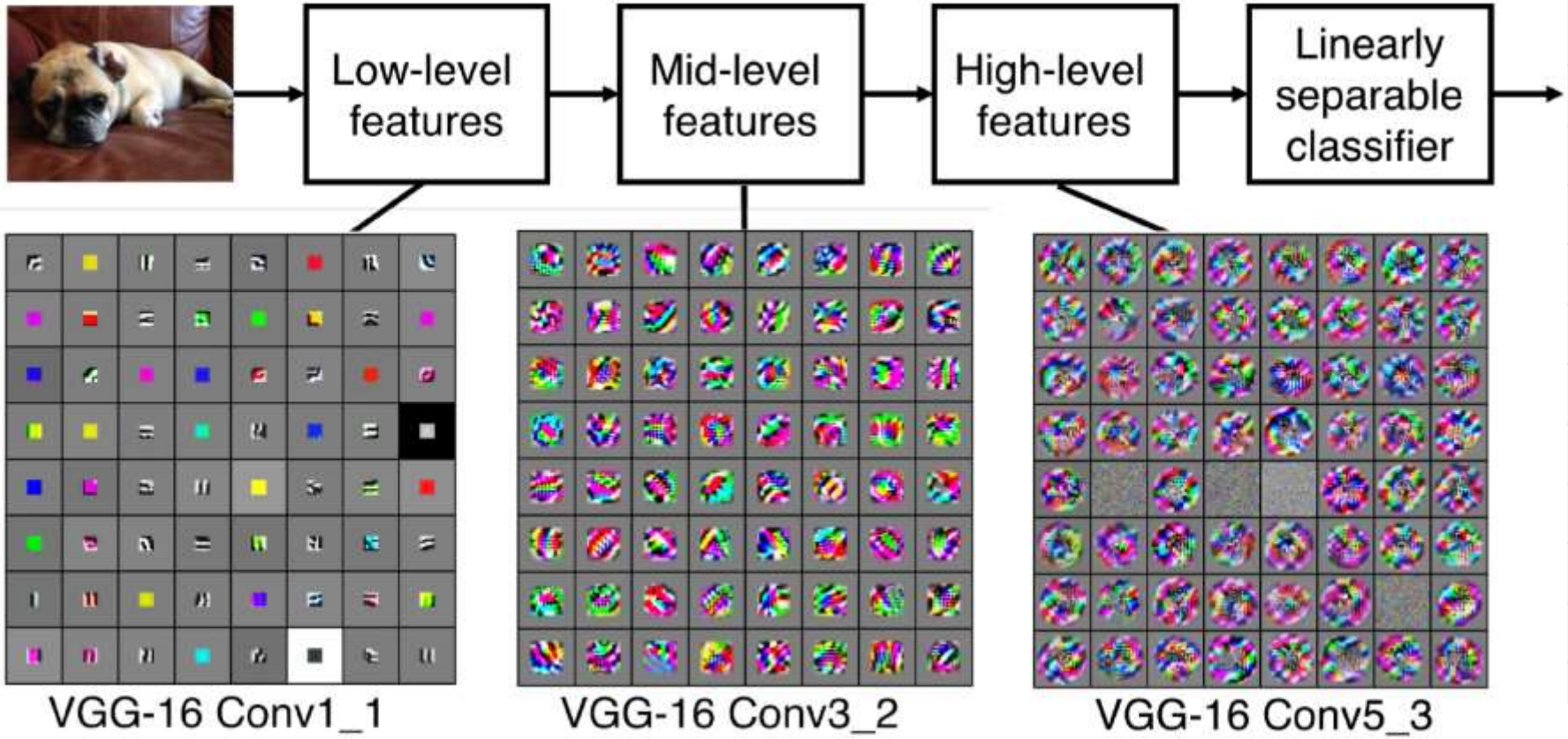


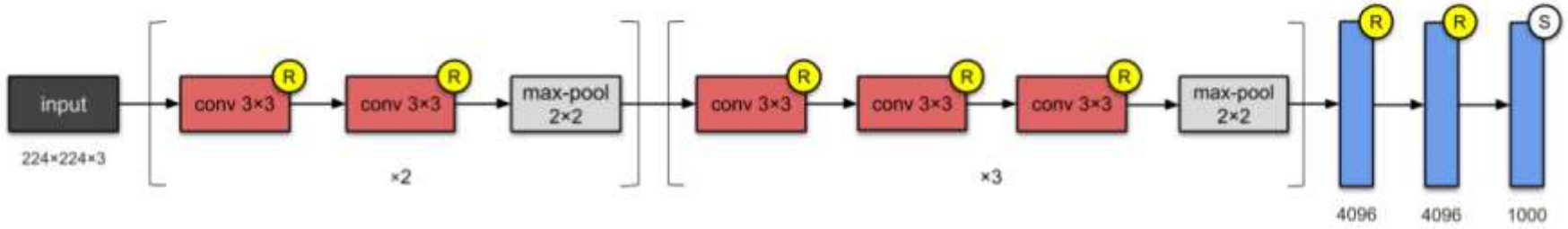
We stack these up to get a "new image" of size 28x28x6!

Now we have a recipe!

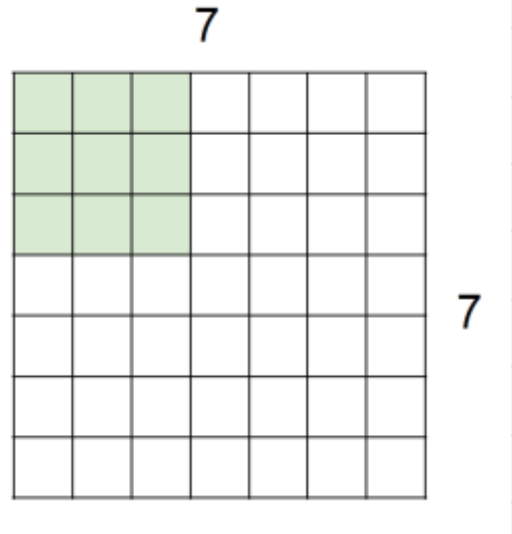


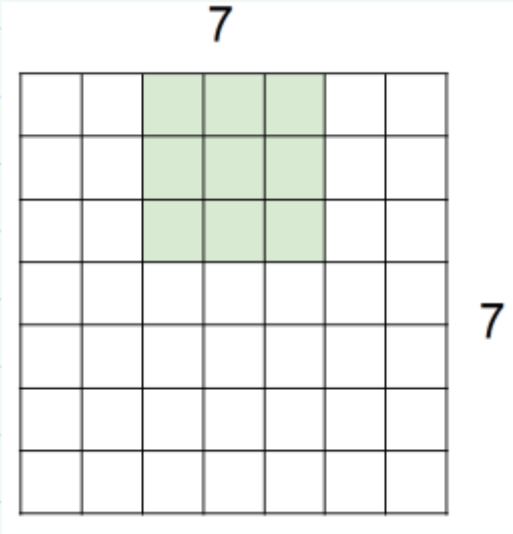
VGG-16 Design

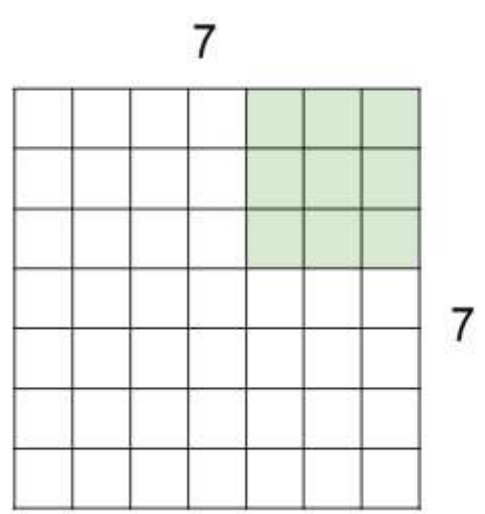




Using Strides





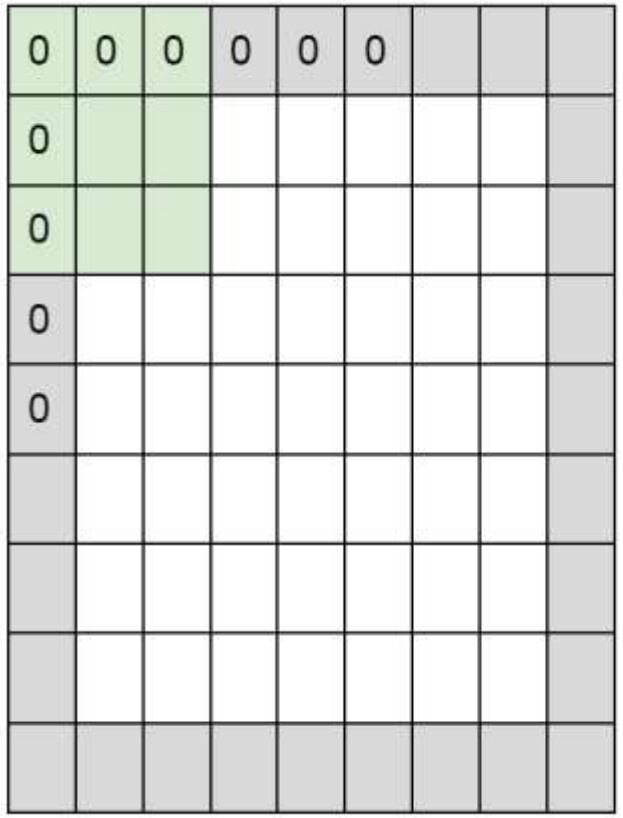


7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**
=> 3x3 output!

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?

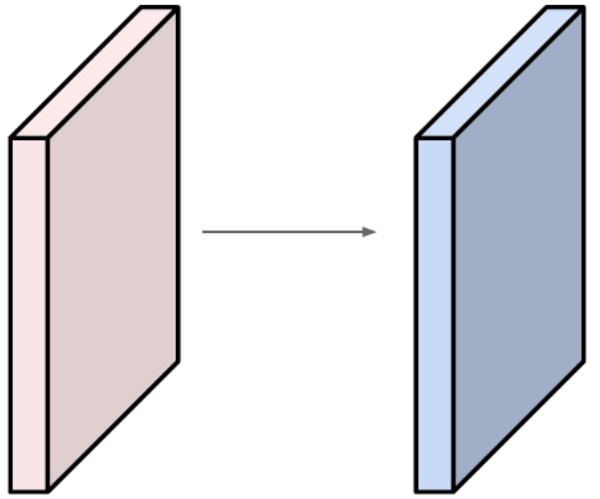
7x7 output!

(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

Calculating the number of parameters

Examples time:

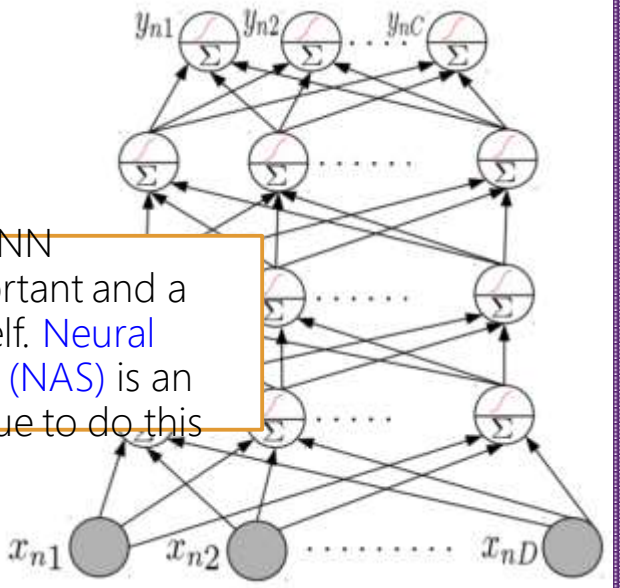


Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

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

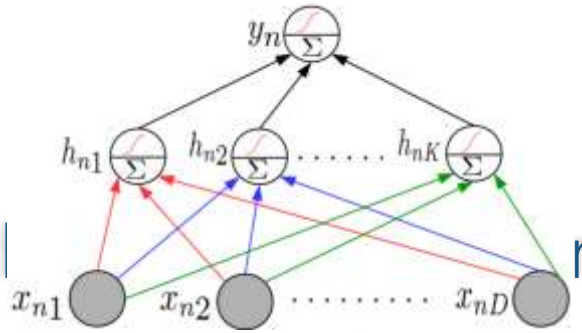
- Much of the magic lies in the hidden layers
- Hidden layers learn and detect good features
- Need to consider a few aspects
  - Number of hidden layers, number of units in each hidden layer
  - Why bother about many hidden layers and not use a single very wide hidden layer (recall Hornik's universal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

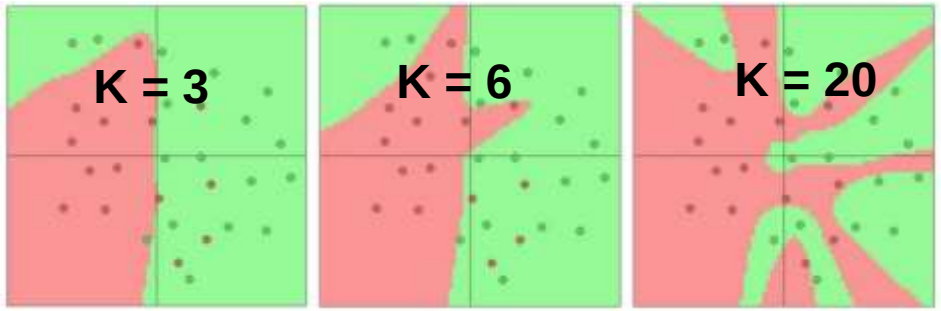


Choosing the right NN architecture is important and a research area in itself. Neural Architecture Search (NAS) is an automated technique to do this

Representational Power of Neural Nets

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





- Recall t nit rall 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)

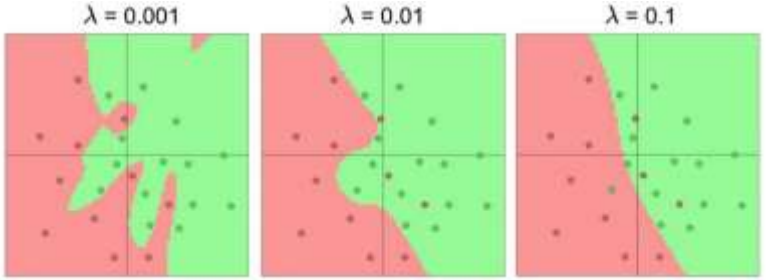
- We can also use multiple hidden layers (each sufficiently large) and

Preventing Overfitting in Neural Nets

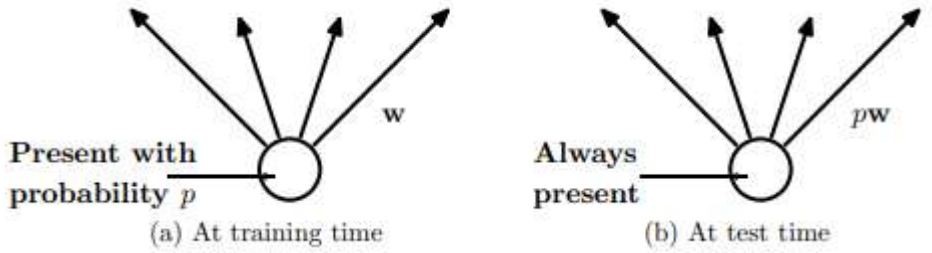
Various other tricks, such as weight sharing across different hidden units of the same layer (used in convolutional neural nets or CNN)

- 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



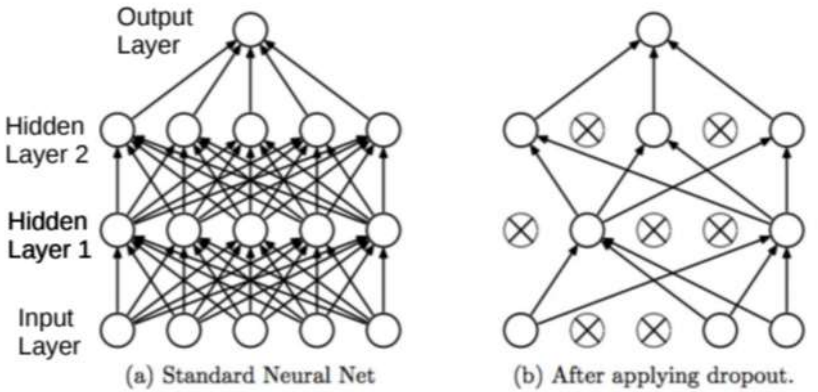
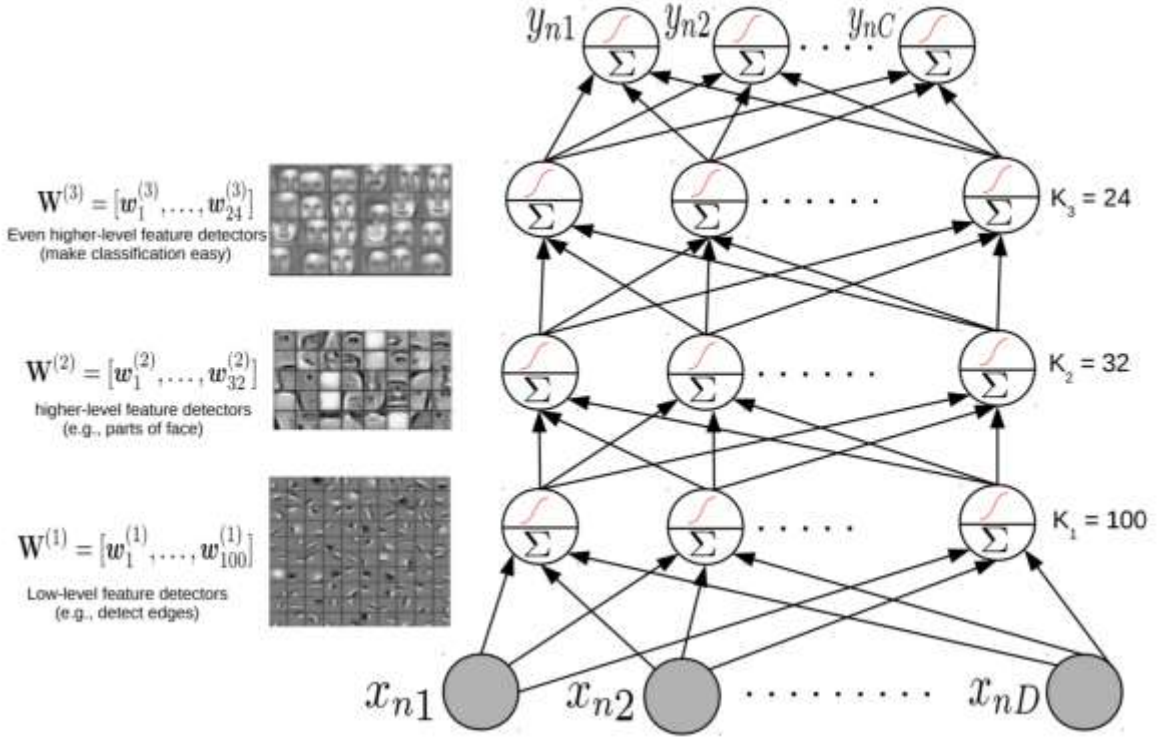


Fig courtesy: Dropout: A Simple Way to Prevent Neural Networks from Overfitting (Srivastava et al, 2014)

Wide or Deep?

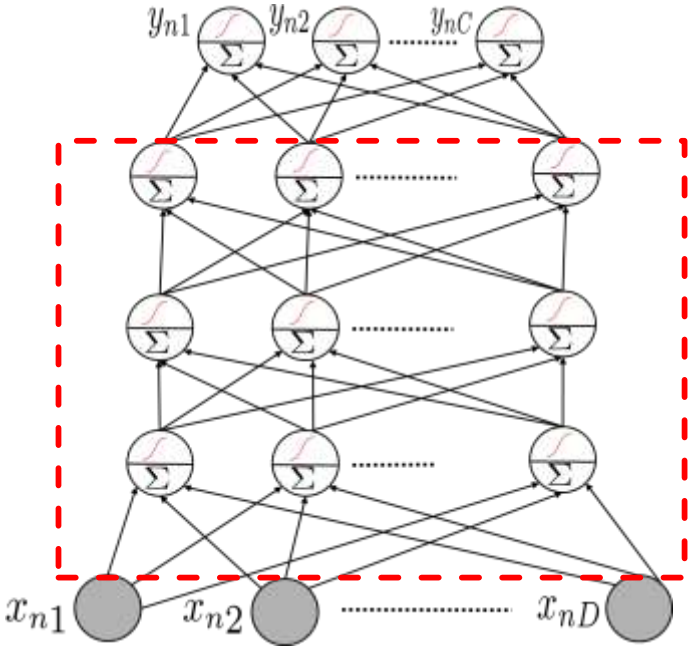
- While very wide single hidden layer can approx. any function, often we prefer many, less wide, hidden layers



- 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

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

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

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

BQ 2.2 solutions

- f_2

$$\boldsymbol{x}^\top \boldsymbol{y} = \sum_i x_i y_i$$

$$\frac{\partial f_2}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial f_2}{\partial x_1} & \cdots & \frac{\partial f_2}{\partial x_n} \end{bmatrix} = \begin{bmatrix} y_1 & \cdots & y_n \end{bmatrix} = \boldsymbol{y}^\top \in \mathbb{R}^n$$

$$\frac{\partial f_2}{\partial \boldsymbol{y}} = \begin{bmatrix} \frac{\partial f_2}{\partial y_1} & \cdots & \frac{\partial f_2}{\partial y_n} \end{bmatrix} = \begin{bmatrix} x_1 & \cdots & x_n \end{bmatrix} = \boldsymbol{x}^\top \in \mathbb{R}^n$$

$$\implies J = \begin{bmatrix} \frac{\partial f_2}{\partial \boldsymbol{x}} & \frac{\partial f_2}{\partial \boldsymbol{y}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{y}^\top & \boldsymbol{x}^\top \end{bmatrix} \in \mathbb{R}^{1 \times 2n}$$

BQ 2.3 Solutions

- $f_3 : \mathbb{R}^n \to \mathbb{R}^{n \times n}$

$$\boldsymbol{x}\boldsymbol{x}^\top = \begin{bmatrix} x_1\boldsymbol{x}^\top \\ x_2\boldsymbol{x}^\top \\ \vdots \\ x_n\boldsymbol{x}^\top \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}x_1 & \boldsymbol{x}x_2 & \cdots & \boldsymbol{x}x_n \end{bmatrix} \in \mathbb{R}^{n \times n}$$

$$\implies \frac{\partial f_3}{\partial x_1} = \underbrace{\begin{bmatrix} \boldsymbol{x}^\top \\ \mathbf{0}_n^\top \\ \vdots \\ \mathbf{0}_n^\top \end{bmatrix}}_{\in \mathbb{R}^{n \times n}} + \underbrace{\begin{bmatrix} \boldsymbol{x} & \mathbf{0}_n & \cdots & \mathbf{0}_n \end{bmatrix}}_{\in \mathbb{R}^{n \times n}} \in \mathbb{R}^{n \times n}$$

$$\implies \frac{\partial f_3}{\partial x_i} = \underbrace{\begin{bmatrix} \mathbf{0}_{(i-1) \times n} \\ \boldsymbol{x}^\top \\ \mathbf{0}_{(n-1+1) \times n} \end{bmatrix}}_{\in \mathbb{R}^{n \times n}} + \underbrace{\begin{bmatrix} \mathbf{0}_{n \times (i-1)} & \boldsymbol{x} & \mathbf{0}_{n \times (n-i+1)} \end{bmatrix}}_{\in \mathbb{R}^{n \times n}} \in \mathbb{R}^{n \times n}$$

To get the Jacobian, we need to concatenate all partial derivatives $\frac{\partial f_3}{\partial x_i}$ and obtain

$$J = \begin{bmatrix} \frac{\partial f_3}{\partial x_1} & \cdots & \frac{\partial f_3}{\partial x_n} \end{bmatrix} \in \mathbb{R}^{(n \times n) \times n}$$

BQ 3.1 solution

BQ 3.2 solution

- The trace for $\boldsymbol{T} \in \mathbb{R}^{D \times D}$ is defined as

$$\text{tr}(\boldsymbol{T}) = \sum_{i=1}^{D} T_{ii}$$

A matrix product $\boldsymbol{ST}$ can be written as

$$(\boldsymbol{ST})_{pq} = \sum_{i} S_{pi} T_{iq}$$

The product $\boldsymbol{AXB}$ contains the elements

$$(\boldsymbol{AXB})_{pq} = \sum_{i=1}^{E} \sum_{j=1}^{F} A_{pi} X_{ij} B_{jq}$$

When we compute the trace, we sum up the diagonal elements of the matrix. Therefore we obtain,

$$\text{tr}(\boldsymbol{AXB}) = \sum_{k=1}^{D} (\boldsymbol{AXB})_{kk} = \sum_{k=1}^{D} \left(\sum_{i=1}^{E} \sum_{j=1}^{F} A_{ki} X_{ij} B_{jk} \right)$$

$$\frac{\partial}{\partial X_{ij}} \text{tr}(\boldsymbol{AXB}) = \sum_{k} A_{ki} B_{jk} = (\boldsymbol{BA})_{ji}$$

We know that the size of the gradient needs to be of the same size as $\boldsymbol{X}$ (i.e., $E \times F$). Therefore, we have to transpose the result above, such that we finally obtain

$$\frac{\partial}{\partial \boldsymbol{X}} \text{tr}(\boldsymbol{AXB}) = \underbrace{\boldsymbol{A}^\top}_{E \times D} \underbrace{\boldsymbol{B}^\top}_{D \times F}$$