# Backpropagation

A. M. Sadeghzadeh, Ph.D.

Sharif University of Technology
Computer Engineering Department (CE)
Data and Network Security Lab (DNSL)

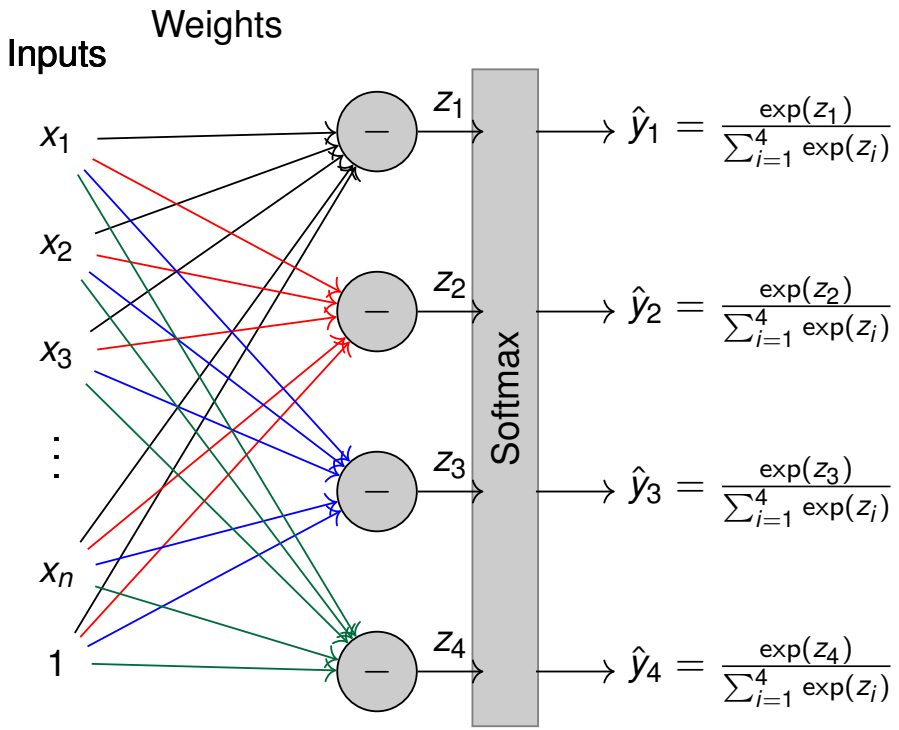

October 1, 2024

Most slides have been adapted from Bhiksha Raj, 11-785, CMU 2020, Justin Johnson, EECS 498-007, University of Michigan 2020, and Fei Fei Li, cs231n, Stanford 2017

# Todays agenda

# Recap

# Softmax classifier



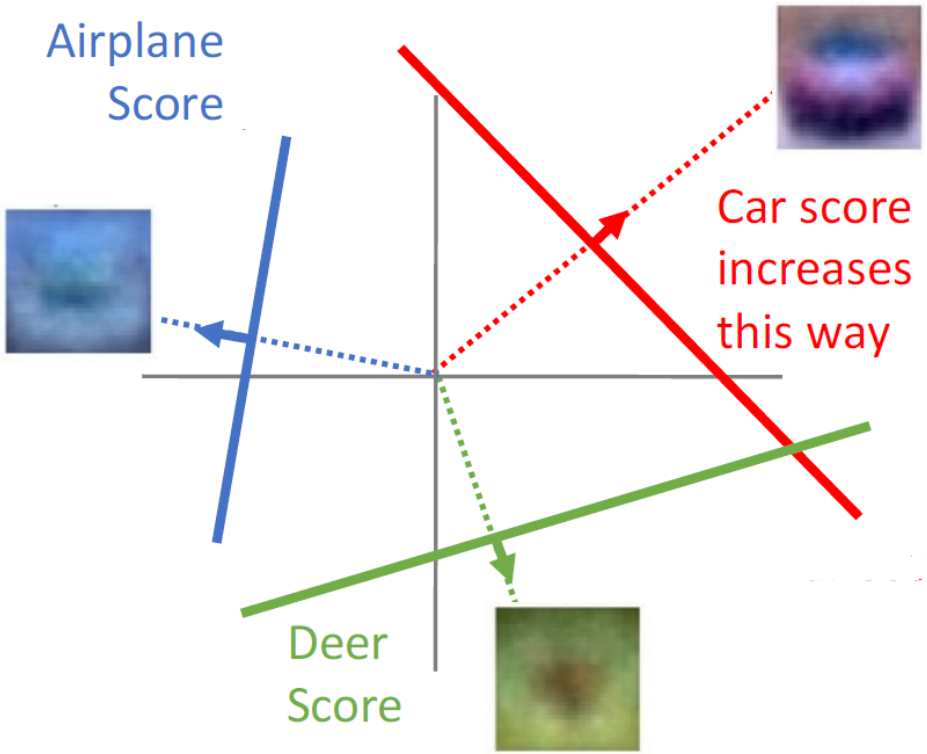

Softmax function:

$$P(\hat{y} = j|\boldsymbol{x}, W) = \frac{\exp(z_j)}{\sum_{i=1}^{K}\exp(z_i)} = \hat{y}_j$$

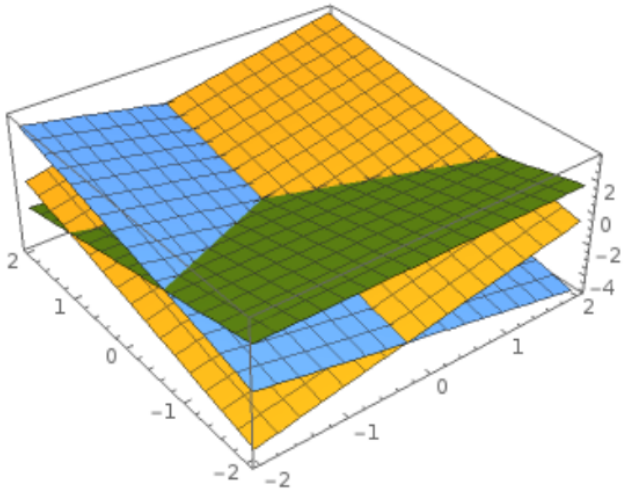
The output vector: $\hat{\boldsymbol{y}} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \hat{y}_3 \\ \hat{y}_4 \end{bmatrix}$

The output of classifier: $\underset{j}{\text{argmax}}\ \hat{y}_j$

# Softmax classifier



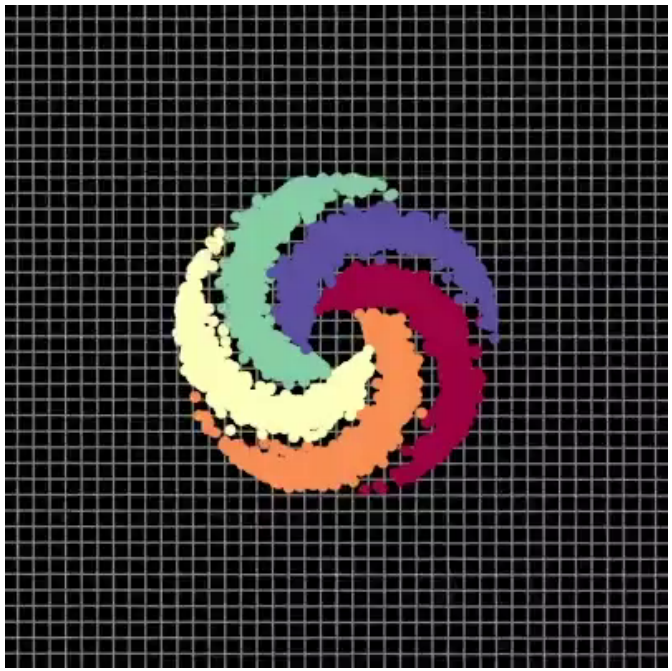

Hyperplanes carving up a high-dimensional space




Plot created using Wolfram Cloud

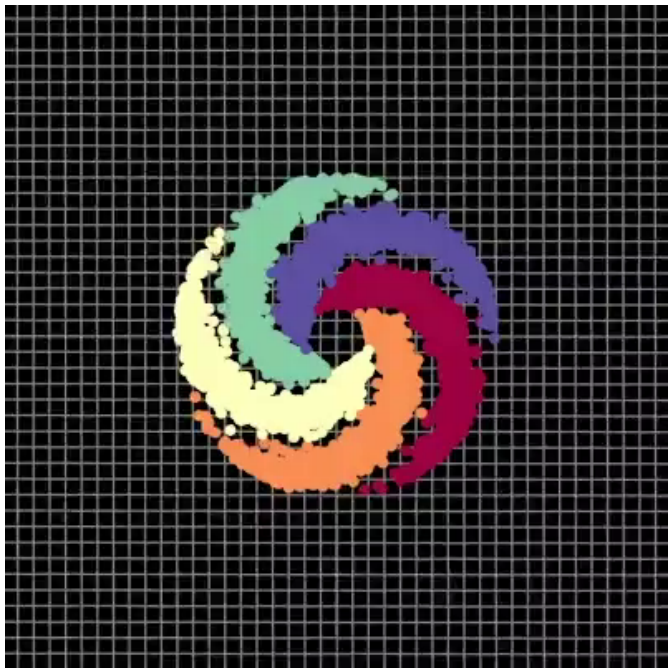
http://vision.stanford.edu/teaching/cs231n-demos/linear-classify/

# Needs more layers

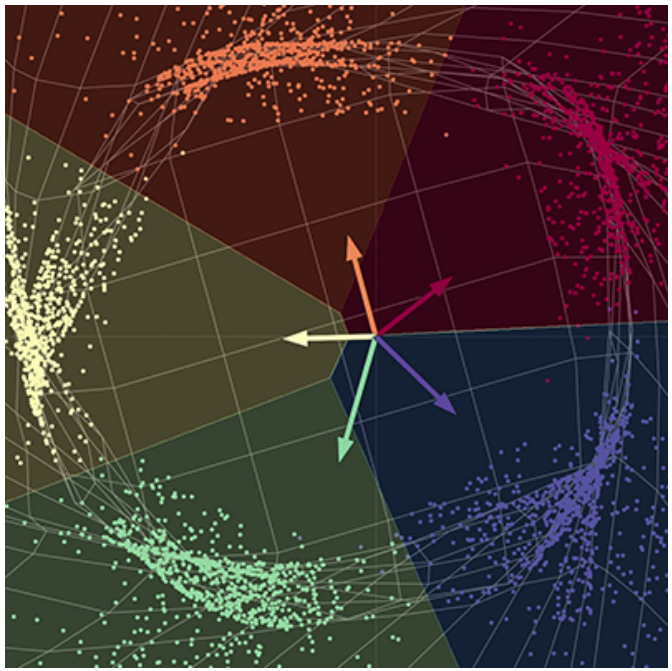


Input points, pre-network

# Needs more layers



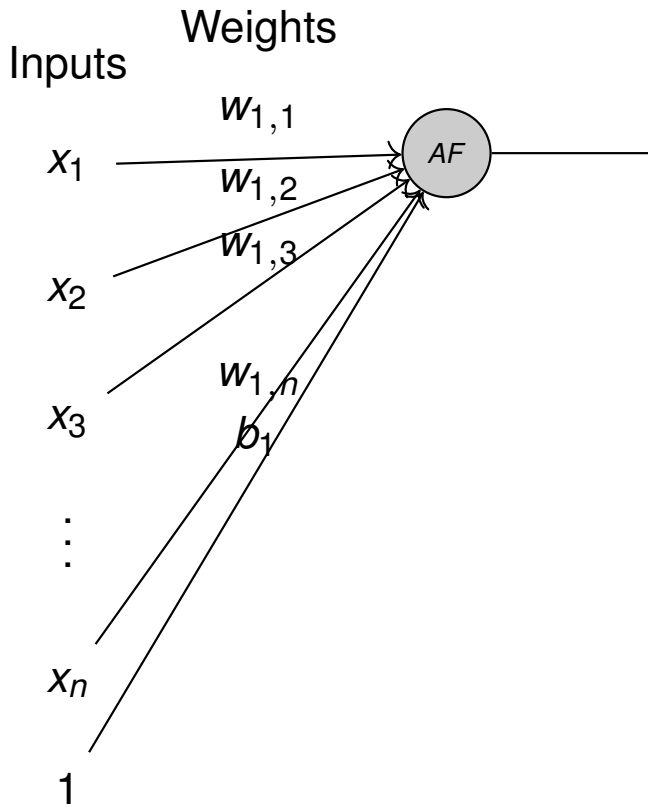
Input points, pre-network



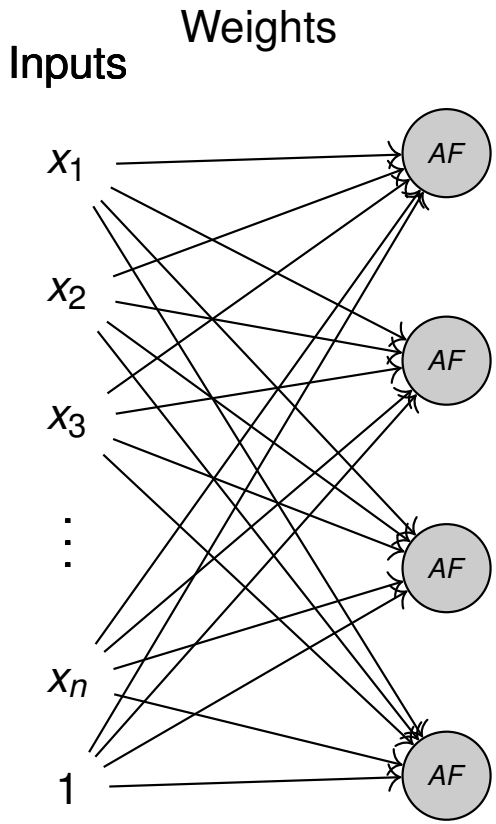
Output points, post-network

(Canziani, 2020)

# Fully connected neural networks

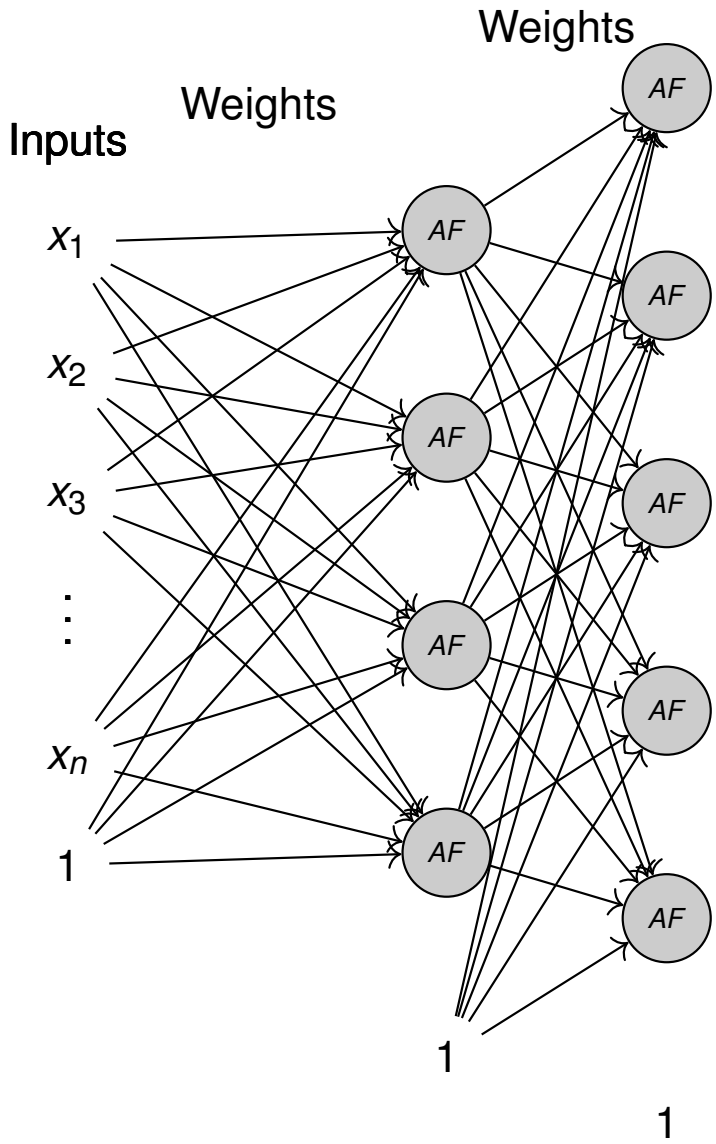


# Fully connected neural networks

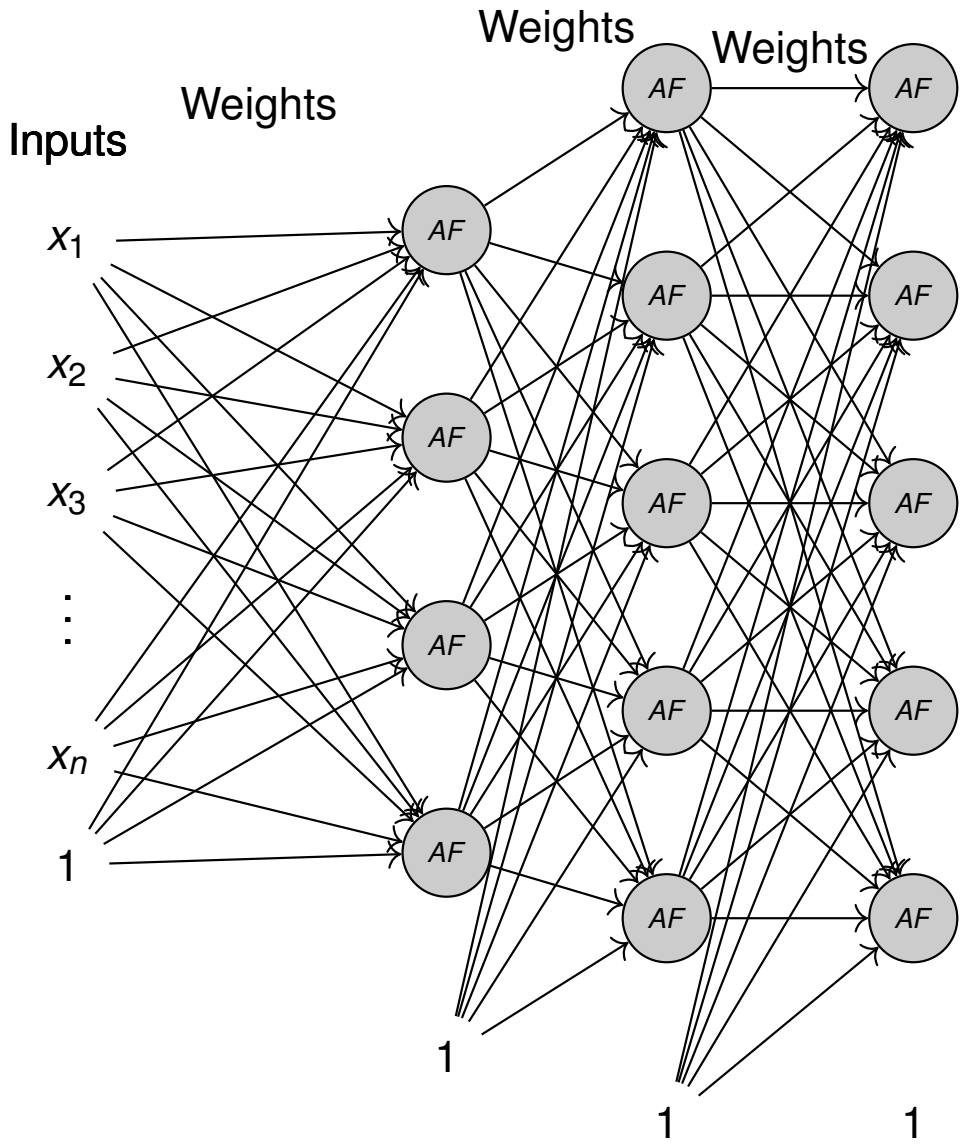



1

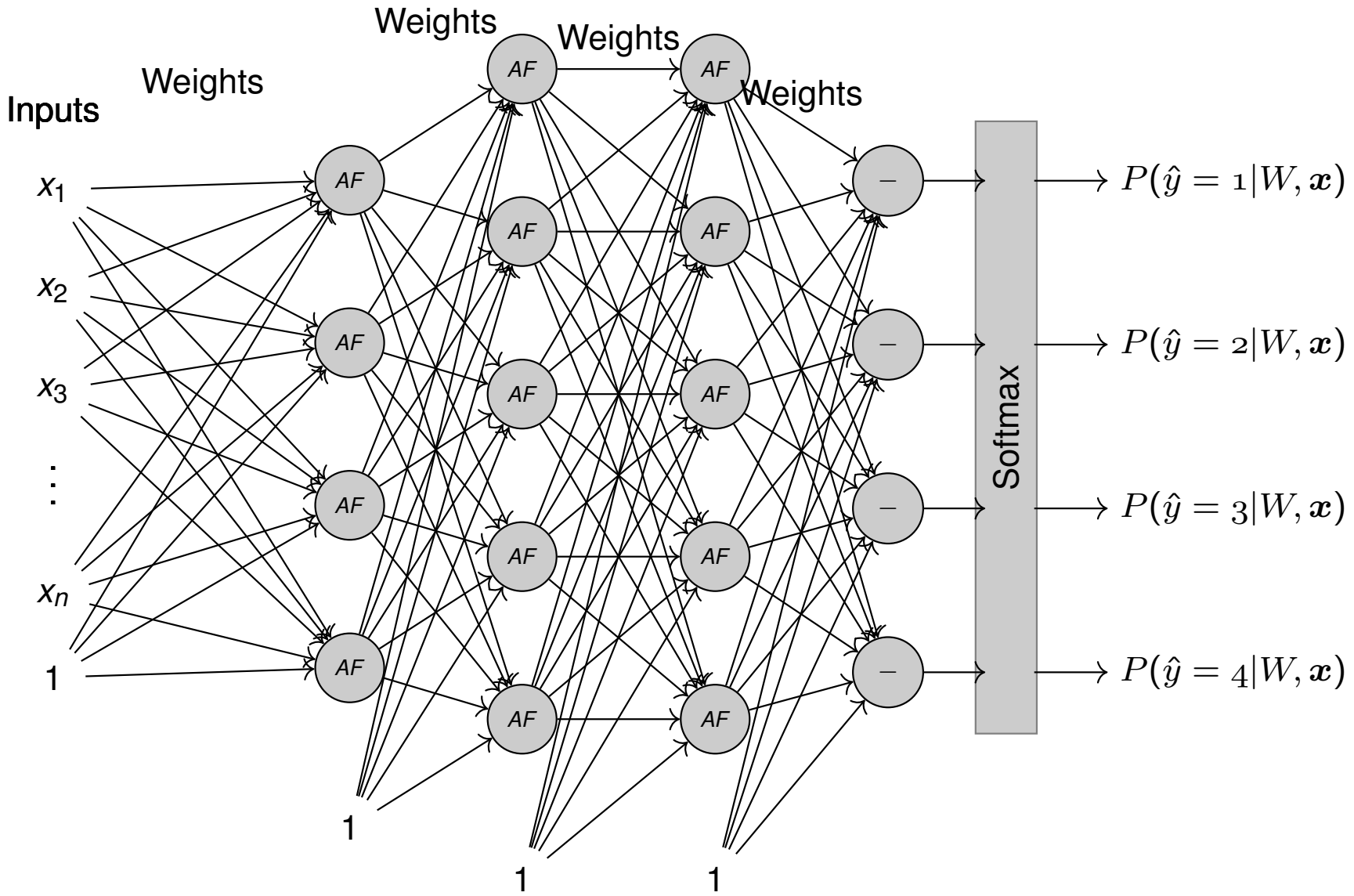
# Fully connected neural networks



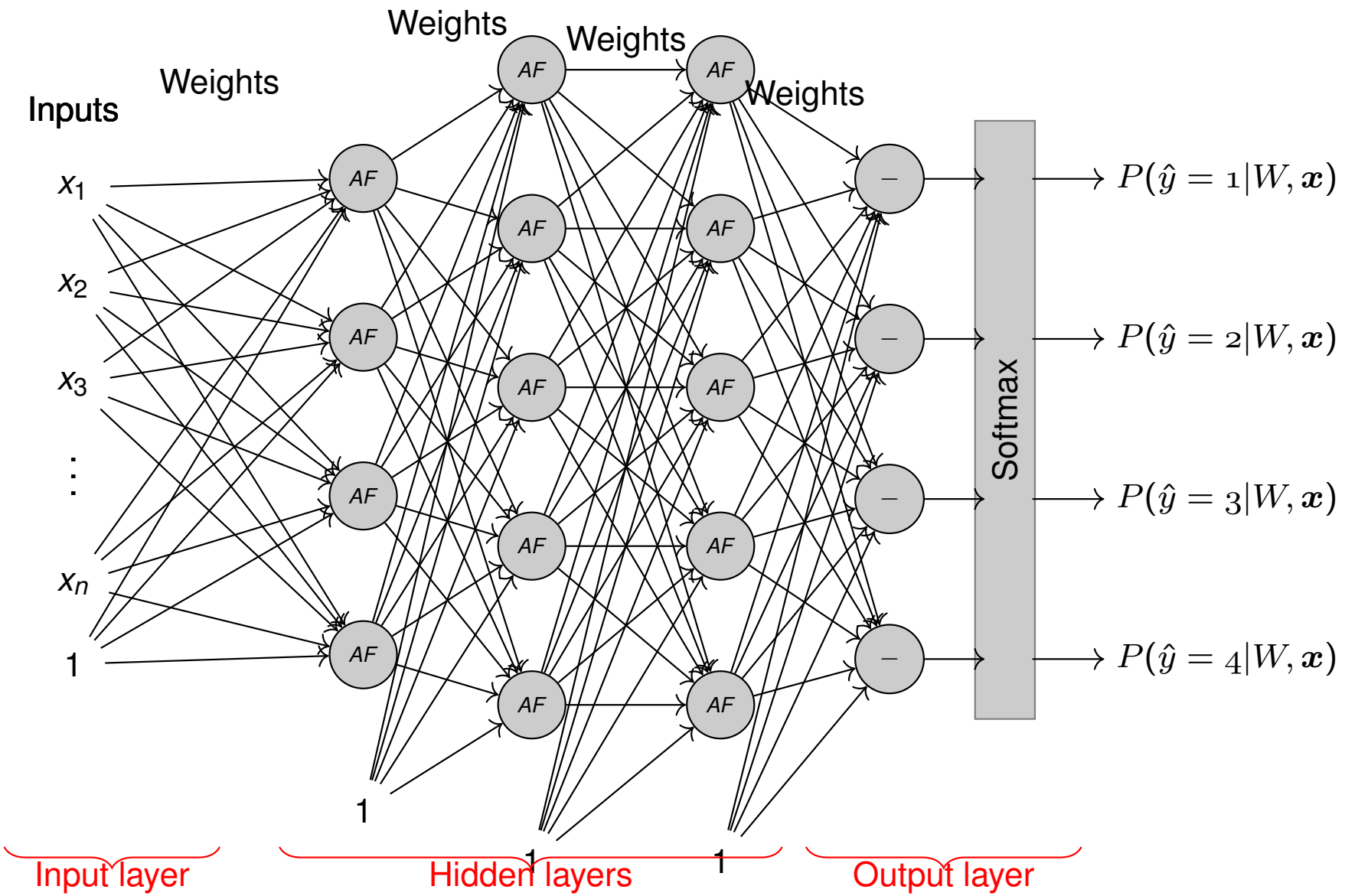
# Fully connected neural networks



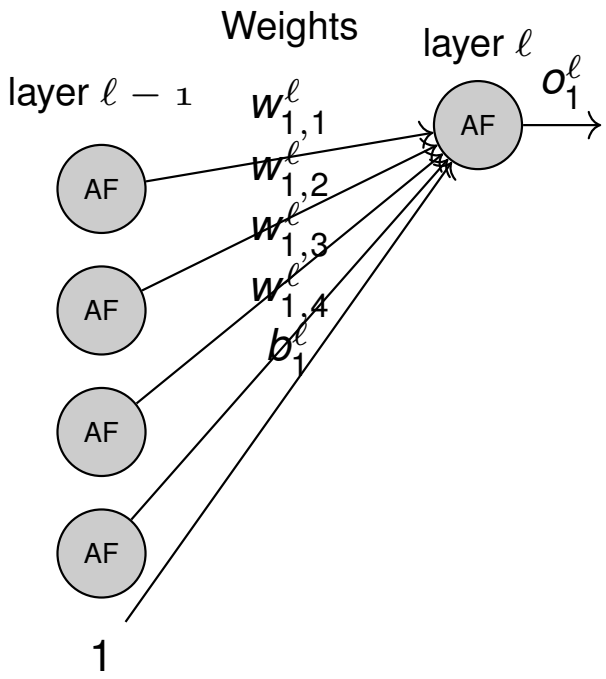
# Fully connected neural networks



# Fully connected neural networks



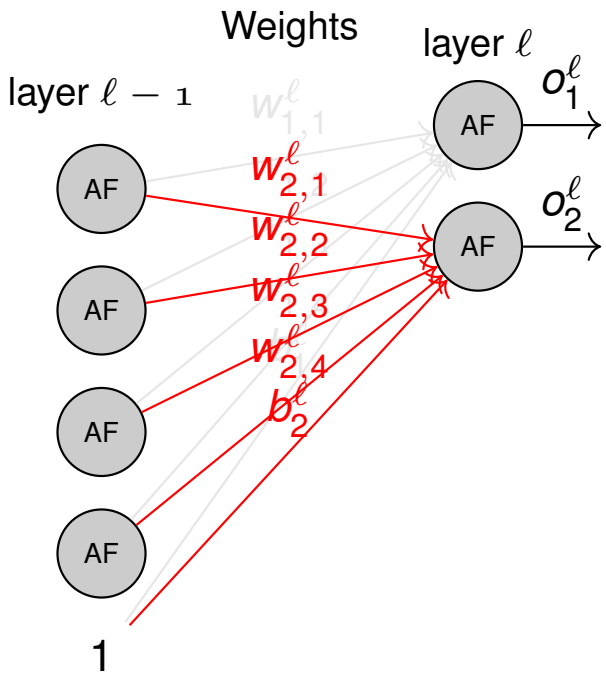
# Fully connected neural networks




$$z_1^\ell = \sum_i w_{1,i}^\ell o_i^{\ell-1} + b_1^\ell = \boldsymbol{w_1^\ell}^T \boldsymbol{o}^{\ell-1} + b_1^\ell, \quad o_1^\ell = AF(z_1^\ell)$$

$$z_1^\ell = \begin{bmatrix} w_{1,1}^\ell & w_{1,2}^\ell & w_{1,3}^\ell & w_{1,4}^\ell \end{bmatrix} \begin{bmatrix} o_1^{\ell-1} \\ o_2^{\ell-1} \\ o_3^{\ell-1} \\ o_4^{\ell-1} \end{bmatrix} + b_1^\ell$$

# Fully connected neural networks

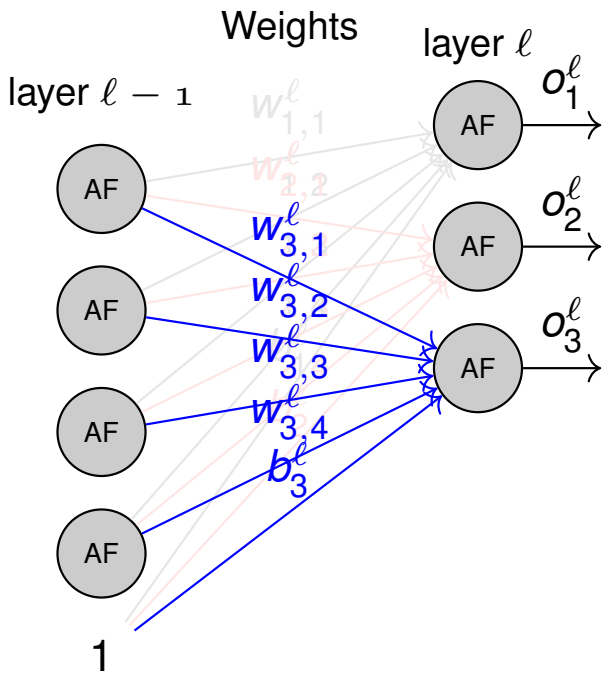



$$z_1^\ell = \sum_i w_{1,i}^\ell o_i^{\ell-1} + b_1^\ell = \boldsymbol{w_1^\ell}^T \boldsymbol{o^{\ell-1}} + b_1^\ell, \quad o_1^\ell = AF(z_1^\ell)$$

$$z_2^\ell = \sum_i w_{2,i}^\ell o_i^{\ell-1} + b_2^\ell = \boldsymbol{w_2^\ell}^T \boldsymbol{o^{\ell-1}} + b_2^\ell, \quad o_2^\ell = AF(z_2^\ell)$$

$$\begin{bmatrix} z_1^\ell \\ z_2^\ell \end{bmatrix} = \begin{bmatrix} w_{1,1}^\ell & w_{1,2}^\ell & w_{1,3}^\ell & w_{1,4}^\ell \\ w_{2,1}^\ell & w_{2,2}^\ell & w_{2,3}^\ell & w_{2,4}^\ell \end{bmatrix} \begin{bmatrix} o_1^{\ell-1} \\ o_2^{\ell-1} \\ o_3^{\ell-1} \\ o_4^{\ell-1} \end{bmatrix} + \begin{bmatrix} b_1^\ell \\ b_2^\ell \end{bmatrix}$$

# Fully connected neural networks



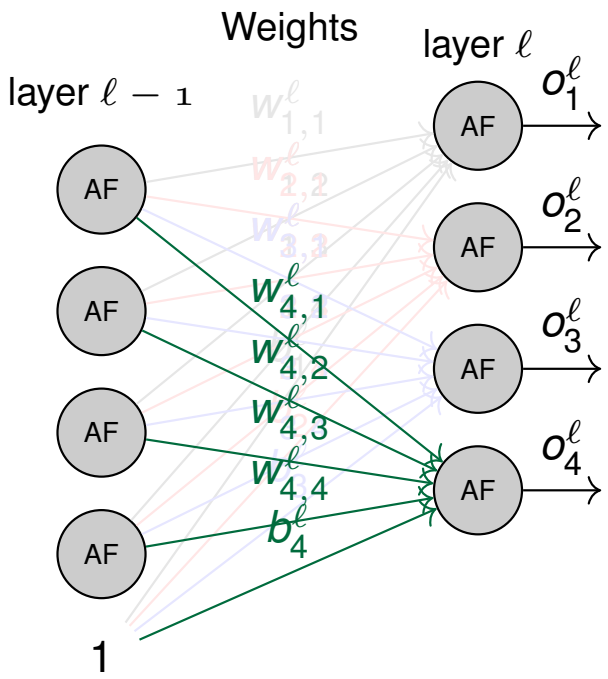

$$z_1^\ell = \sum_i w_{1,i}^\ell o_i^{\ell-1} + b_1^\ell = \boldsymbol{w_1^\ell}^T \boldsymbol{o^{\ell-1}} + b_1^\ell, \quad o_1^\ell = AF(z_1^\ell)$$

$$z_2^\ell = \sum_i w_{2,i}^\ell o_i^{\ell-1} + b_2^\ell = \boldsymbol{w_2^\ell}^T \boldsymbol{o^{\ell-1}} + b_2^\ell, \quad o_2^\ell = AF(z_2^\ell)$$

$$z_3^\ell = \sum_i w_{3,i}^\ell o_i^{\ell-1} + b_3^\ell = \boldsymbol{w_3^\ell}^T \boldsymbol{o^{\ell-1}} + b_3^\ell, \quad o_3^\ell = AF(z_3^\ell)$$

$$\begin{bmatrix} z_1^\ell \\ z_2^\ell \\ z_3^\ell \end{bmatrix} = \begin{bmatrix} w_{1,1}^\ell & w_{1,2}^\ell & w_{1,3}^\ell & w_{1,4}^\ell \\ w_{2,1}^\ell & w_{2,2}^\ell & w_{2,3}^\ell & w_{2,4}^\ell \\ w_{3,1}^\ell & w_{3,2}^\ell & w_{3,3}^\ell & w_{3,4}^\ell \end{bmatrix} \begin{bmatrix} o_1^{\ell-1} \\ o_2^{\ell-1} \\ o_3^{\ell-1} \\ o_4^{\ell-1} \end{bmatrix} + \begin{bmatrix} b_1^\ell \\ b_2^\ell \\ b_3^\ell \end{bmatrix}$$

# Fully connected neural networks




$$z_1^\ell = \sum_i w_{1,i}^\ell o_i^{\ell-1} + b_1^\ell = \boldsymbol{w_1^\ell}^T \boldsymbol{o^{\ell-1}} + b_1^\ell, \quad o_1^\ell = AF(z_1^\ell)$$

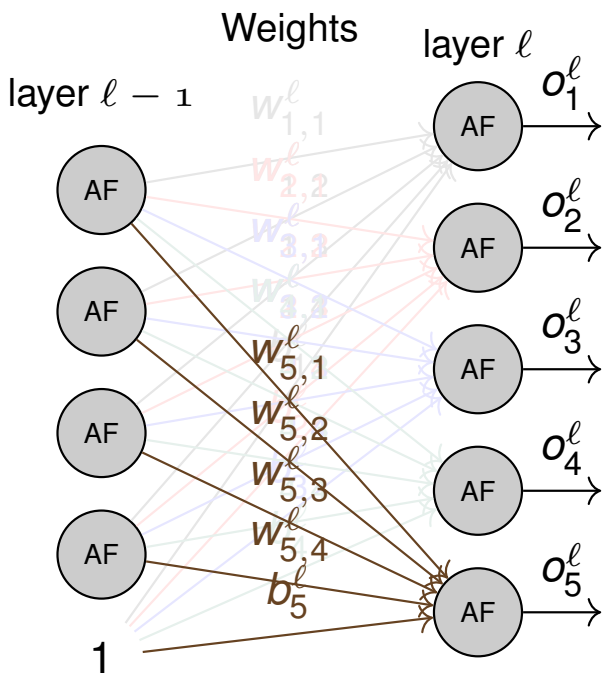
$$z_2^\ell = \sum_i w_{2,i}^\ell o_i^{\ell-1} + b_2^\ell = \boldsymbol{w_2^\ell}^T \boldsymbol{o^{\ell-1}} + b_2^\ell, \quad o_2^\ell = AF(z_2^\ell)$$

$$z_3^\ell = \sum_i w_{3,i}^\ell o_i^{\ell-1} + b_3^\ell = \boldsymbol{w_3^\ell}^T \boldsymbol{o^{\ell-1}} + b_3^\ell, \quad o_3^\ell = AF(z_3^\ell)$$

$$z_4^\ell = \sum_i w_{4,i}^\ell o_i^{\ell-1} + b_4^\ell = \boldsymbol{w_4^\ell}^T \boldsymbol{o^{\ell-1}} + b_4^\ell, \quad o_4^\ell = AF(z_4^\ell)$$

$$\begin{bmatrix} z_1^\ell \\ z_2^\ell \\ z_3^\ell \\ z_4^\ell \end{bmatrix} = \begin{bmatrix} w_{1,1}^\ell & w_{1,2}^\ell & w_{1,3}^\ell & w_{1,4}^\ell \\ w_{2,1}^\ell & w_{2,2}^\ell & w_{2,3}^\ell & w_{2,4}^\ell \\ w_{3,1}^\ell & w_{3,2}^\ell & w_{3,3}^\ell & w_{3,4}^\ell \\ w_{4,1}^\ell & w_{4,2}^\ell & w_{4,3}^\ell & w_{4,4}^\ell \end{bmatrix} \begin{bmatrix} o_1^{\ell-1} \\ o_2^{\ell-1} \\ o_3^{\ell-1} \\ o_4^{\ell-1} \end{bmatrix} + \begin{bmatrix} b_1^\ell \\ b_2^\ell \\ b_3^\ell \\ b_4^\ell \end{bmatrix}$$

# Fully connected neural networks




$$z_1^\ell = \sum_i w_{1,i}^\ell o_i^{\ell-1} + b_1^\ell = \boldsymbol{w_1^\ell}^T \boldsymbol{o^{\ell-1}} + b_1^\ell, \quad o_1^\ell = AF(z_1^\ell)$$

$$z_2^\ell = \sum_i w_{2,i}^\ell o_i^{\ell-1} + b_2^\ell = \boldsymbol{w_2^\ell}^T \boldsymbol{o^{\ell-1}} + b_2^\ell, \quad o_2^\ell = AF(z_2^\ell)$$

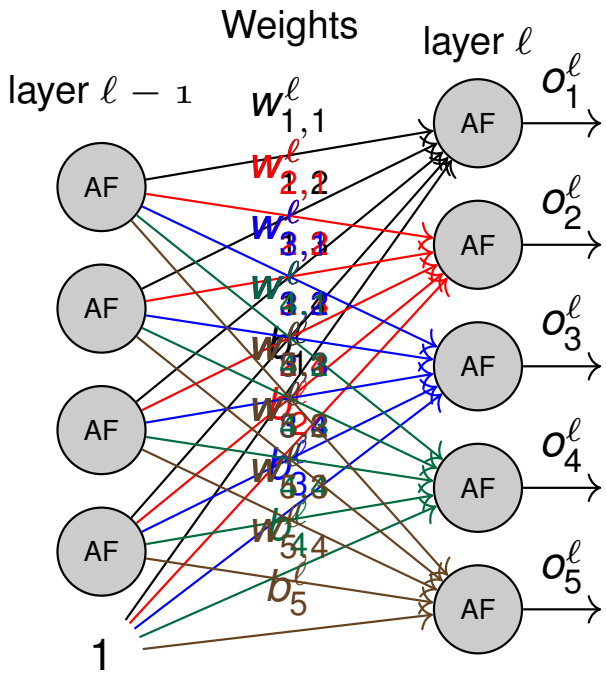
$$z_3^\ell = \sum_i w_{3,i}^\ell o_i^{\ell-1} + b_3^\ell = \boldsymbol{w_3^\ell}^T \boldsymbol{o^{\ell-1}} + b_3^\ell, \quad o_3^\ell = AF(z_3^\ell)$$

$$z_4^\ell = \sum_i w_{4,i}^\ell o_i^{\ell-1} + b_4^\ell = \boldsymbol{w_4^\ell}^T \boldsymbol{o^{\ell-1}} + b_4^\ell, \quad o_4^\ell = AF(z_4^\ell)$$

$$z_5^\ell = \sum_i w_{5,i}^\ell o_i^{\ell-1} + b_5^\ell = \boldsymbol{w_5^\ell}^T \boldsymbol{o^{\ell-1}} + b_5^\ell, \quad o_5^\ell = AF(z_5^\ell)$$

$$\begin{bmatrix} z_1^\ell \\ z_2^\ell \\ z_3^\ell \\ z_4^\ell \\ z_5^\ell \end{bmatrix} = \begin{bmatrix} w_{1,1}^\ell & w_{1,2}^\ell & w_{1,3}^\ell & w_{1,4}^\ell \\ w_{2,1}^\ell & w_{2,2}^\ell & w_{2,3}^\ell & w_{2,4}^\ell \\ w_{3,1}^\ell & w_{3,2}^\ell & w_{3,3}^\ell & w_{3,4}^\ell \\ w_{4,1}^\ell & w_{4,2}^\ell & w_{4,3}^\ell & w_{4,4}^\ell \\ w_{5,1}^\ell & w_{5,2}^\ell & w_{5,3}^\ell & w_{5,4}^\ell \end{bmatrix} \begin{bmatrix} o_1^{\ell-1} \\ o_2^{\ell-1} \\ o_3^{\ell-1} \\ o_4^{\ell-1} \end{bmatrix} + \begin{bmatrix} b_1^\ell \\ b_2^\ell \\ b_3^\ell \\ b_4^\ell \\ b_5^\ell \end{bmatrix}$$

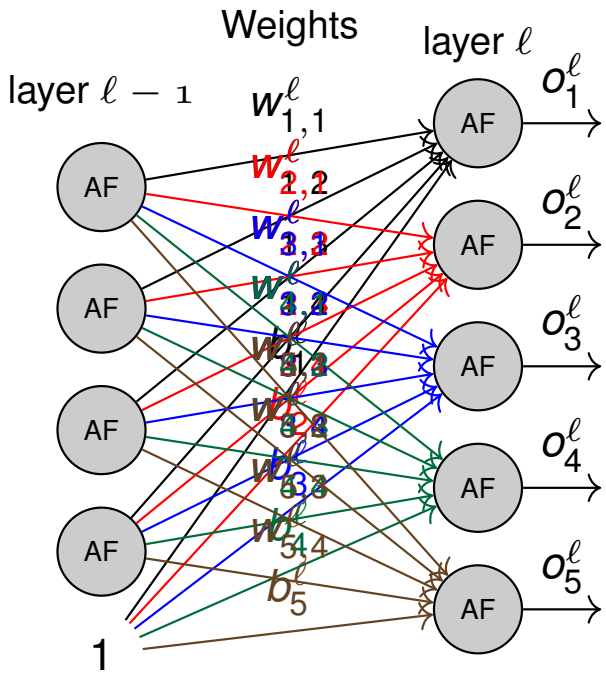
# Vector activation




$$\begin{bmatrix} z_1^\ell \\ z_2^\ell \\ z_3^\ell \\ z_4^\ell \\ z_5^\ell \end{bmatrix} = \begin{bmatrix} w_{1,1}^\ell & w_{1,2}^\ell & w_{1,3}^\ell & w_{1,4}^\ell \\ w_{2,1}^\ell & w_{2,2}^\ell & w_{2,3}^\ell & w_{2,4}^\ell \\ w_{3,1}^\ell & w_{3,2}^\ell & w_{3,3}^\ell & w_{3,4}^\ell \\ w_{4,1}^\ell & w_{4,2}^\ell & w_{4,3}^\ell & w_{4,4}^\ell \\ w_{5,1}^\ell & w_{5,2}^\ell & w_{5,3}^\ell & w_{5,4}^\ell \end{bmatrix} \begin{bmatrix} o_1^{\ell-1} \\ o_2^{\ell-1} \\ o_3^{\ell-1} \\ o_4^{\ell-1} \end{bmatrix} + \begin{bmatrix} b_1^\ell \\ b_2^\ell \\ b_3^\ell \\ b_4^\ell \\ b_5^\ell \end{bmatrix}$$

$$\begin{bmatrix} o_1^\ell \\ o_2^\ell \\ o_3^\ell \\ o_4^\ell \\ o_5^\ell \end{bmatrix} = \begin{bmatrix} AF(z_1^\ell) \\ AF(z_2^\ell) \\ AF(z_3^\ell) \\ AF(z_4^\ell) \\ AF(z_5^\ell) \end{bmatrix}$$

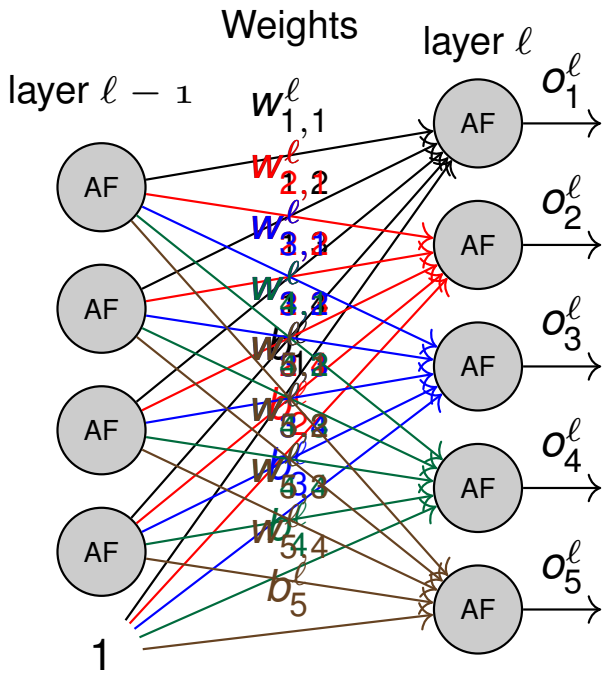
# Vector activation




$$\begin{bmatrix} z_1^\ell \\ z_2^\ell \\ z_3^\ell \\ z_4^\ell \\ z_5^\ell \end{bmatrix} = \begin{bmatrix} w_{1,1}^\ell & w_{1,2}^\ell & w_{1,3}^\ell & w_{1,4}^\ell \\ w_{2,1}^\ell & w_{2,2}^\ell & w_{2,3}^\ell & w_{2,4}^\ell \\ w_{3,1}^\ell & w_{3,2}^\ell & w_{3,3}^\ell & w_{3,4}^\ell \\ w_{4,1}^\ell & w_{4,2}^\ell & w_{4,3}^\ell & w_{4,4}^\ell \\ w_{5,1}^\ell & w_{5,2}^\ell & w_{5,3}^\ell & w_{5,4}^\ell \end{bmatrix} \begin{bmatrix} o_1^{\ell-1} \\ o_2^{\ell-1} \\ o_3^{\ell-1} \\ o_4^{\ell-1} \end{bmatrix} + \begin{bmatrix} b_1^\ell \\ b_2^\ell \\ b_3^\ell \\ b_4^\ell \\ b_5^\ell \end{bmatrix}$$

$$\begin{bmatrix} o_1^\ell \\ o_2^\ell \\ o_3^\ell \\ o_4^\ell \\ o_5^\ell \end{bmatrix} = \begin{bmatrix} AF(z_1^\ell) \\ AF(z_2^\ell) \\ AF(z_3^\ell) \\ AF(z_4^\ell) \\ AF(z_5^\ell) \end{bmatrix} = AF\left( \begin{bmatrix} z_1^\ell \\ z_2^\ell \\ z_3^\ell \\ z_4^\ell \\ z_5^\ell \end{bmatrix} \right)$$

# Vector activation

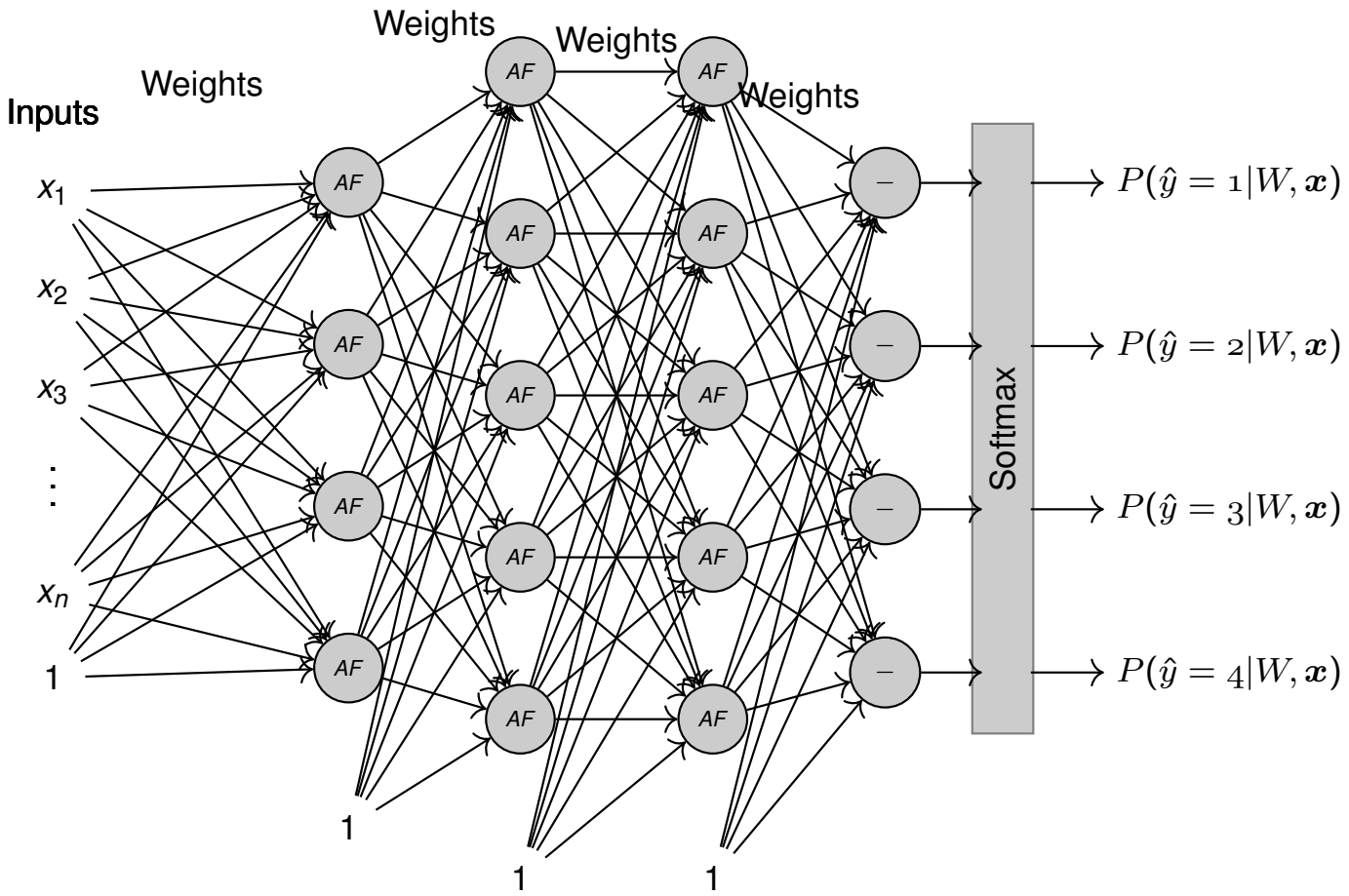



$$\begin{bmatrix} z_1^\ell \\ z_2^\ell \\ z_3^\ell \\ z_4^\ell \\ z_5^\ell \end{bmatrix} = \begin{bmatrix} w_{1,1}^\ell & w_{1,2}^\ell & w_{1,3}^\ell & w_{1,4}^\ell \\ w_{2,1}^\ell & w_{2,2}^\ell & w_{2,3}^\ell & w_{2,4}^\ell \\ w_{3,1}^\ell & w_{3,2}^\ell & w_{3,3}^\ell & w_{3,4}^\ell \\ w_{4,1}^\ell & w_{4,2}^\ell & w_{4,3}^\ell & w_{4,4}^\ell \\ w_{5,1}^\ell & w_{5,2}^\ell & w_{5,3}^\ell & w_{5,4}^\ell \end{bmatrix} \begin{bmatrix} o_1^{\ell-1} \\ o_2^{\ell-1} \\ o_3^{\ell-1} \\ o_4^{\ell-1} \end{bmatrix} + \begin{bmatrix} b_1^\ell \\ b_2^\ell \\ b_3^\ell \\ b_4^\ell \\ b_5^\ell \end{bmatrix}$$

$$\begin{bmatrix} o_1^\ell \\ o_2^\ell \\ o_3^\ell \\ o_4^\ell \\ o_5^\ell \end{bmatrix} = \begin{bmatrix} AF(z_1^\ell) \\ AF(z_2^\ell) \\ AF(z_3^\ell) \\ AF(z_4^\ell) \\ AF(z_5^\ell) \end{bmatrix} = AF\left( \begin{bmatrix} z_1^\ell \\ z_2^\ell \\ z_3^\ell \\ z_4^\ell \\ z_5^\ell \end{bmatrix} \right)$$

$$\boldsymbol{o}^\ell = AF(W^\ell \boldsymbol{o}^{\ell-1} + \boldsymbol{b}^\ell)$$

# Fully connected neural networks

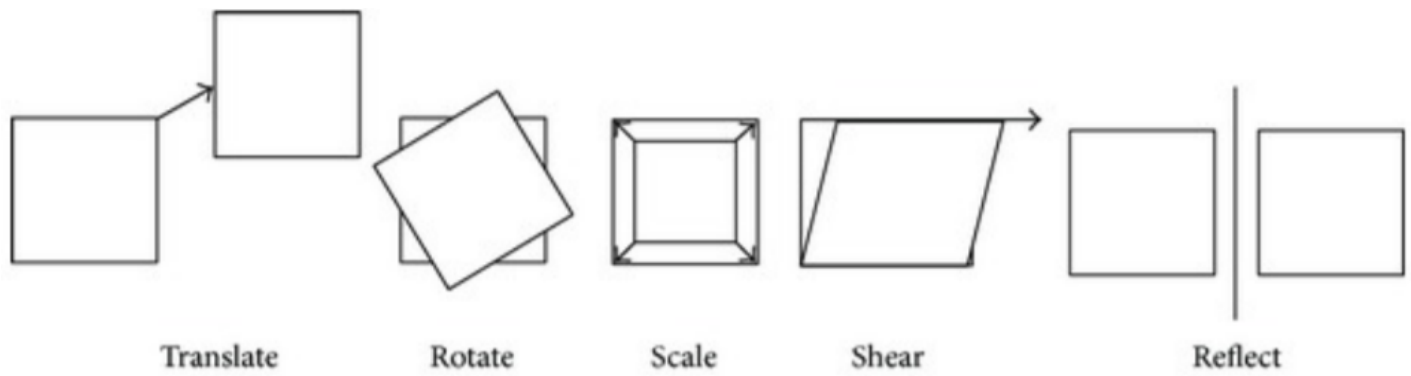



$$\hat{\boldsymbol{y}} = Softmax(W^4(AF(W^3(AF(W^2(AF(W^1\boldsymbol{x} + \boldsymbol{b}^1)) + \boldsymbol{b}^2)) + \boldsymbol{b}^3)) + \boldsymbol{b}^4)$$

# Deep Neural Networks

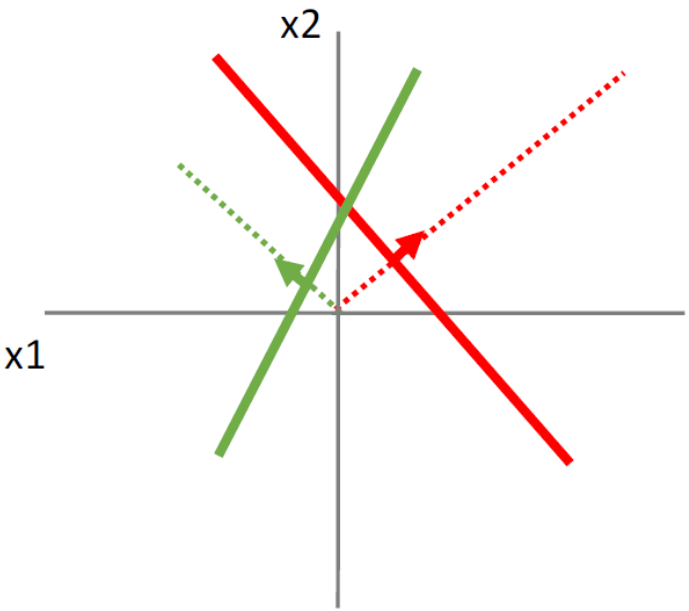
# Affine transformation

- Affine transformation
  - Translate
  - Rotate
  - Scale
  - Shear
  - Reflect
- Preserve parallel lines



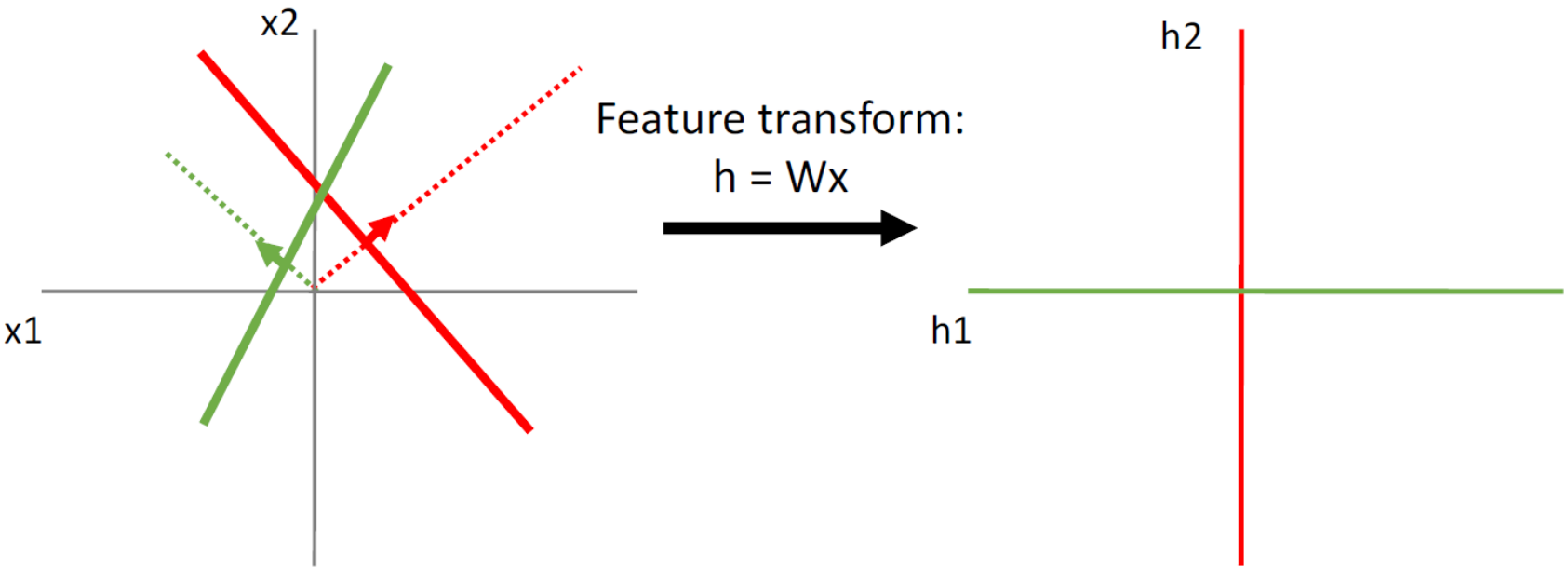
# Space warping

- Consider a linear transform: $\boldsymbol{h} = W\boldsymbol{x}$ Where $\boldsymbol{x}, \boldsymbol{h}$ are both 2-dimensional



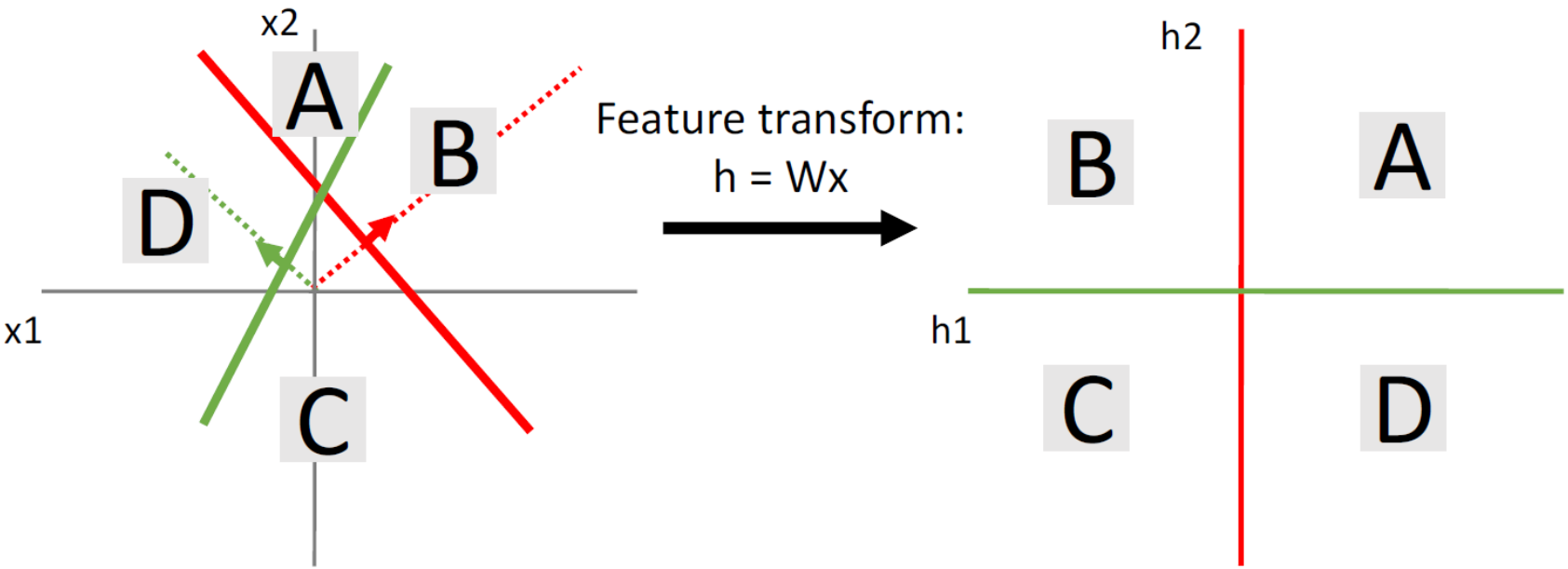
# Space warping

- Consider a linear transform: $\boldsymbol{h} = W\boldsymbol{x}$ Where $\boldsymbol{x}, \boldsymbol{h}$ are both 2-dimensional



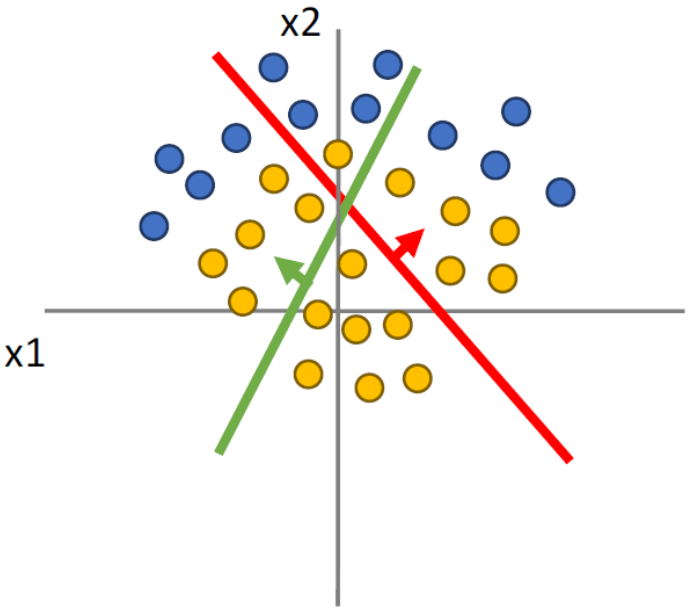
# Space warping

- Consider a linear transform: $\boldsymbol{h} = W\boldsymbol{x}$ Where $\boldsymbol{x}, \boldsymbol{h}$ are both 2-dimensional



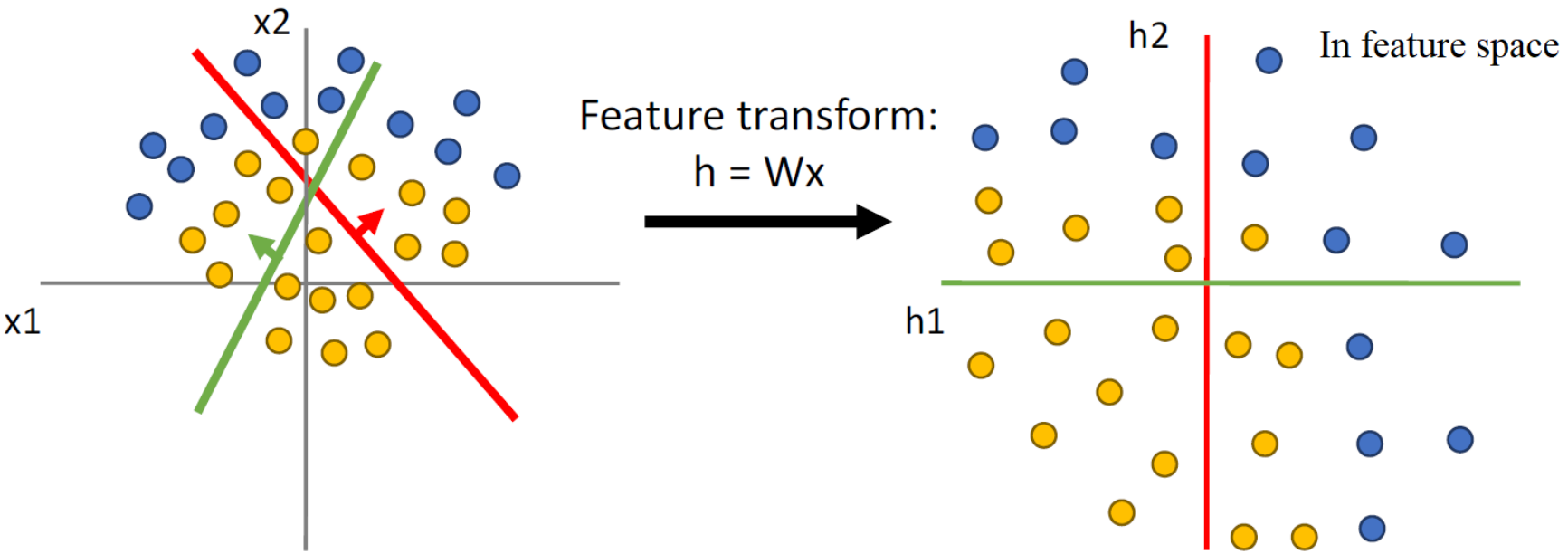
# Space warping

- Consider a linear transform: $\boldsymbol{h} = W\boldsymbol{x}$ Where $\boldsymbol{x}, \boldsymbol{h}$ are both 2-dimensional
- Points not linearly separable in original space



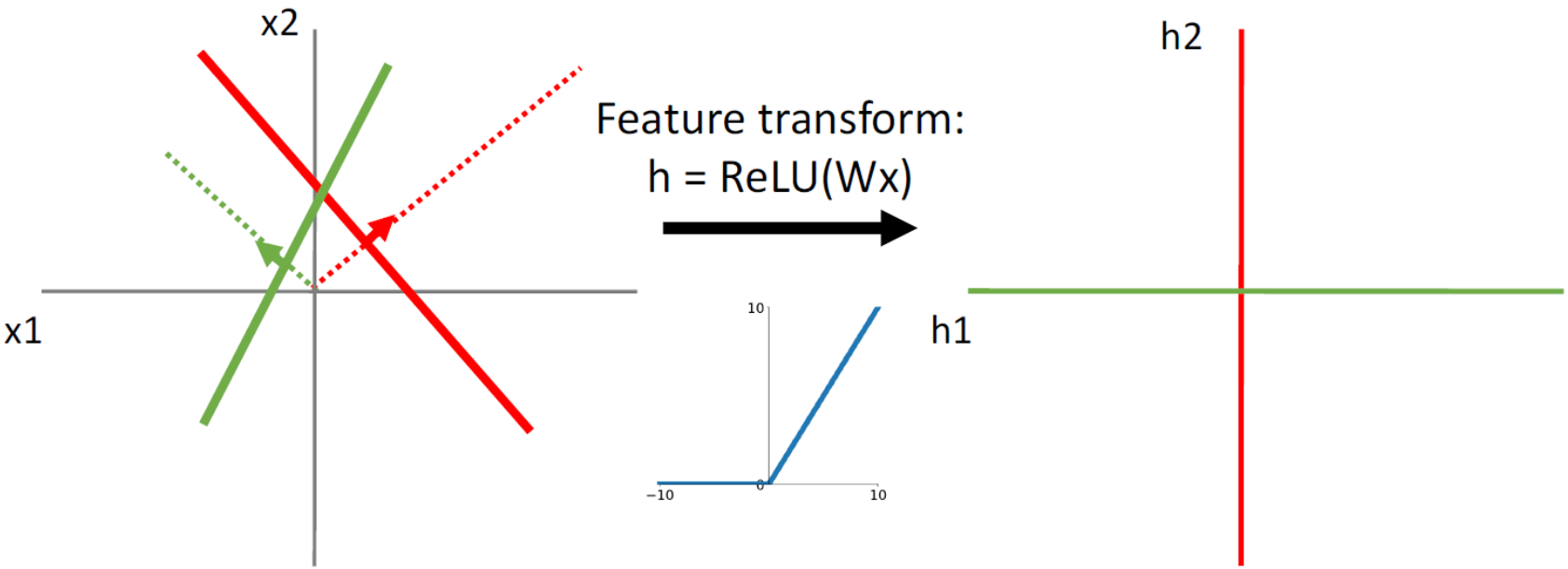
# Space warping

- Consider a linear transform: $\boldsymbol{h} = W\boldsymbol{x}$ Where $\boldsymbol{x}, \boldsymbol{h}$ are both 2-dimensional
- Points not linearly separable in original space
  - Not linearly separable in feature space



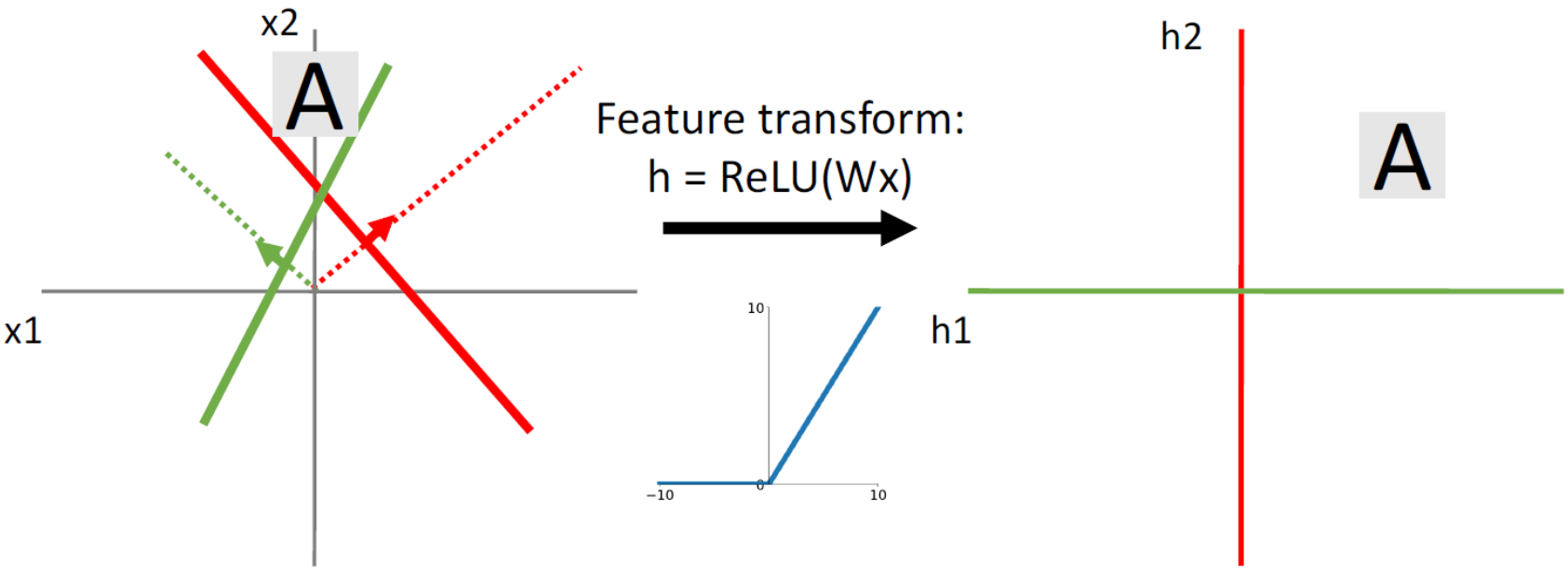
# Space warping

- Consider a neural net hidden layer: $\boldsymbol{h} = ReLU(W\boldsymbol{x}) = max(0, W\boldsymbol{x})$.
  - Where x, h are both 2-dimensional.



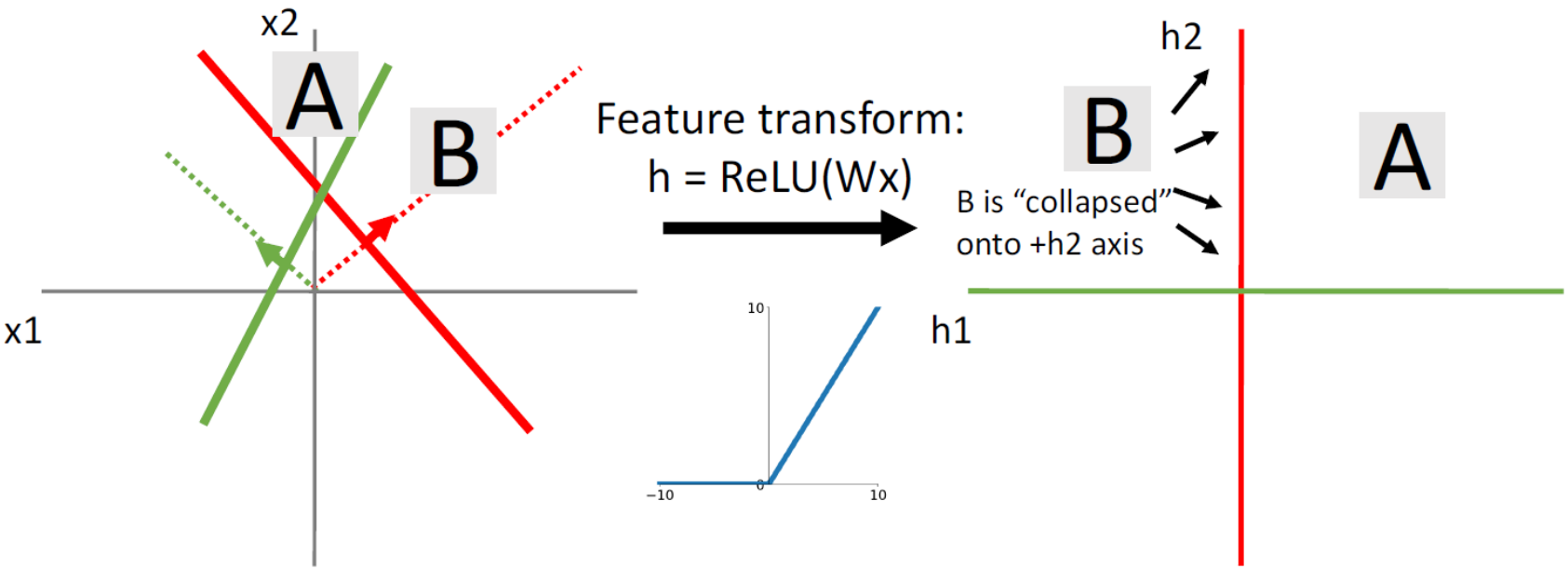
# Space warping

- Consider a neural net hidden layer: $\boldsymbol{h} = ReLU(W\boldsymbol{x}) = max(0, W\boldsymbol{x})$.
  - Where x, h are both 2-dimensional.



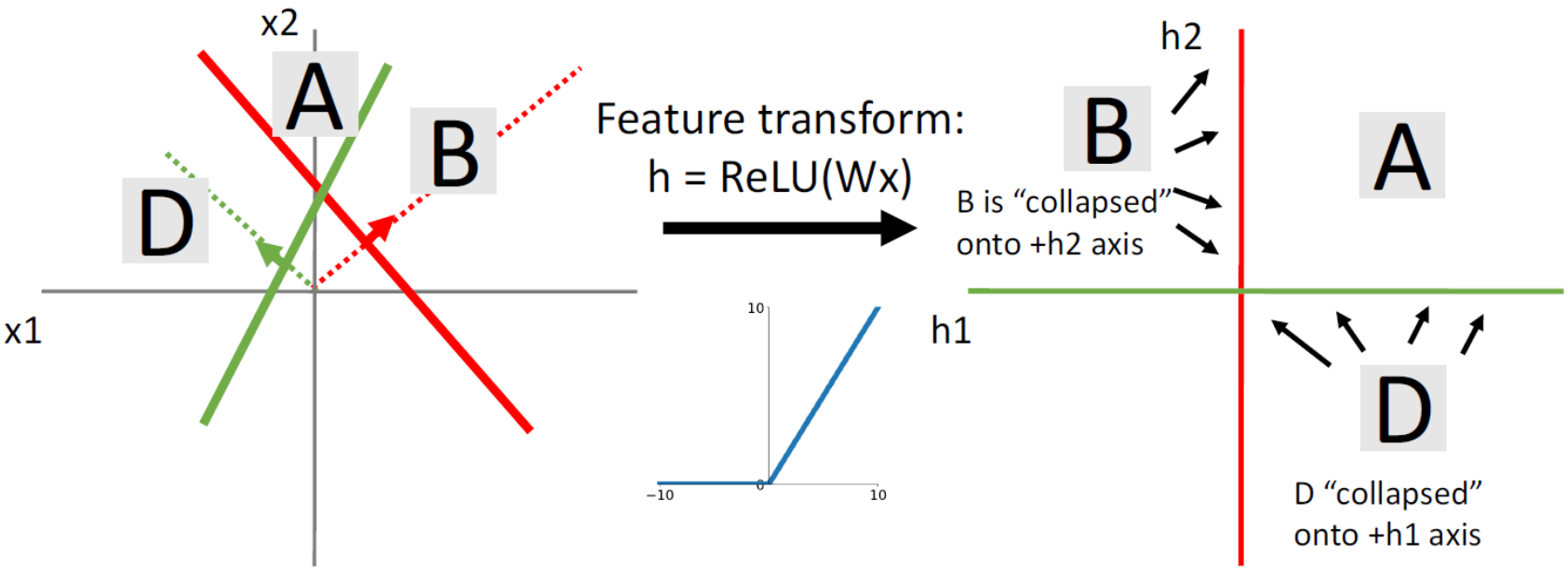
# Space warping

- Consider a neural net hidden layer: $\boldsymbol{h} = ReLU(W\boldsymbol{x}) = max(0, W\boldsymbol{x})$.
  - Where x, h are both 2-dimensional.



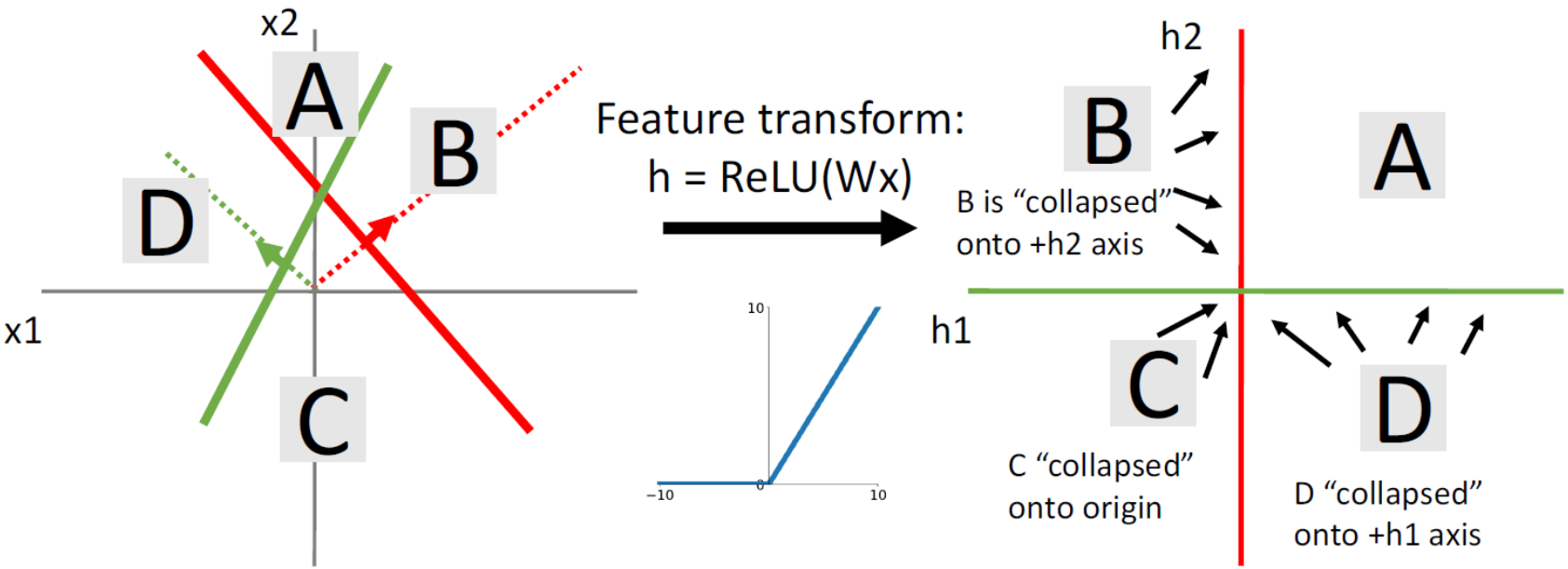
# Space warping

- Consider a neural net hidden layer: $\boldsymbol{h} = ReLU(W\boldsymbol{x}) = max(0, W\boldsymbol{x})$.
  - Where x, h are both 2-dimensional.



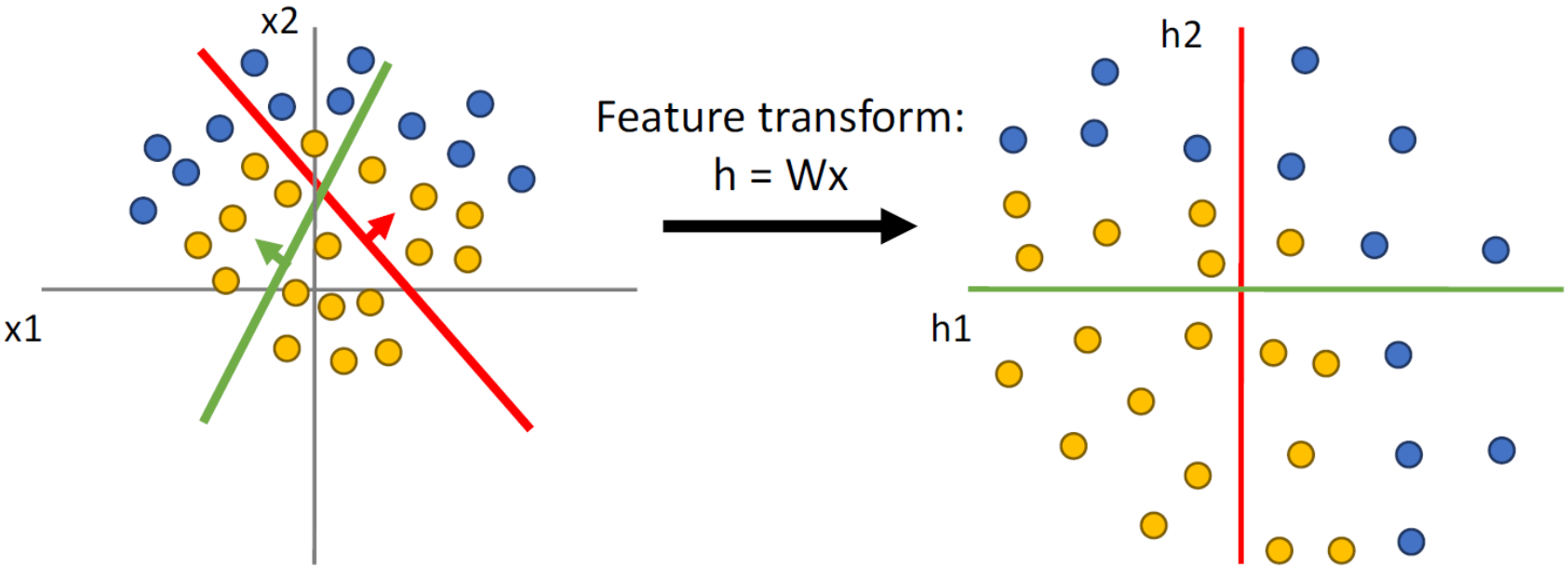
# Space warping

- Consider a neural net hidden layer: $\boldsymbol{h} = ReLU(W\boldsymbol{x}) = max(0, W\boldsymbol{x})$.
  - Where x, h are both 2-dimensional.



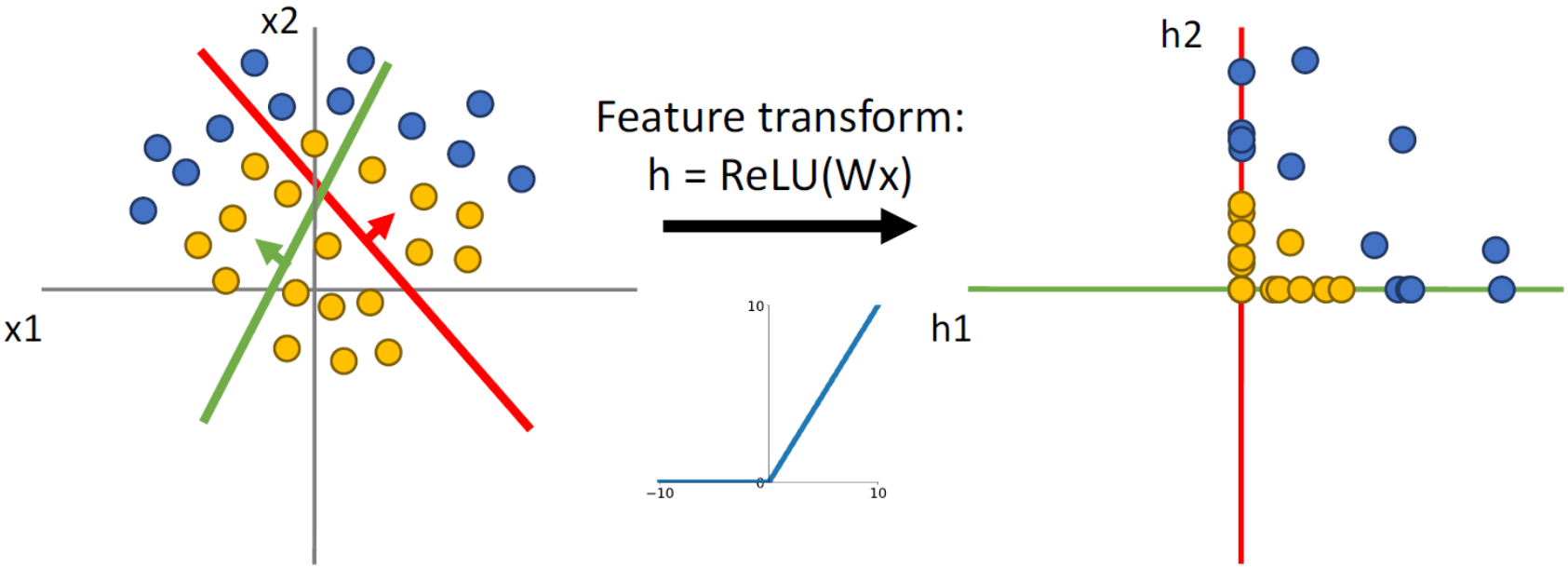
# Space warping

- Consider a neural net hidden layer: $\boldsymbol{h} = ReLU(W\boldsymbol{x}) = max(0, W\boldsymbol{x})$.
  - Where x, h are both 2-dimensional.
- Points not linearly separable in original space



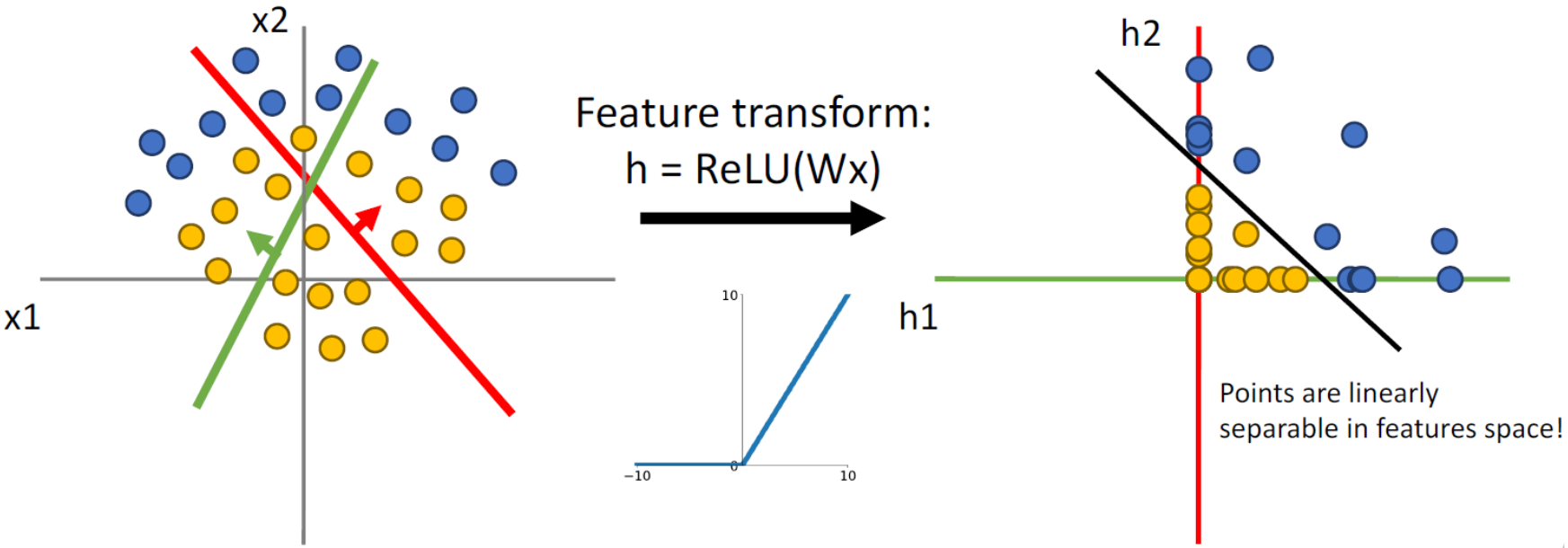
# Space warping

- Consider a neural net hidden layer: $\boldsymbol{h} = ReLU(W\boldsymbol{x}) = max(0, W\boldsymbol{x})$.
  - Where x, h are both 2-dimensional.
- Points not linearly separable in original space



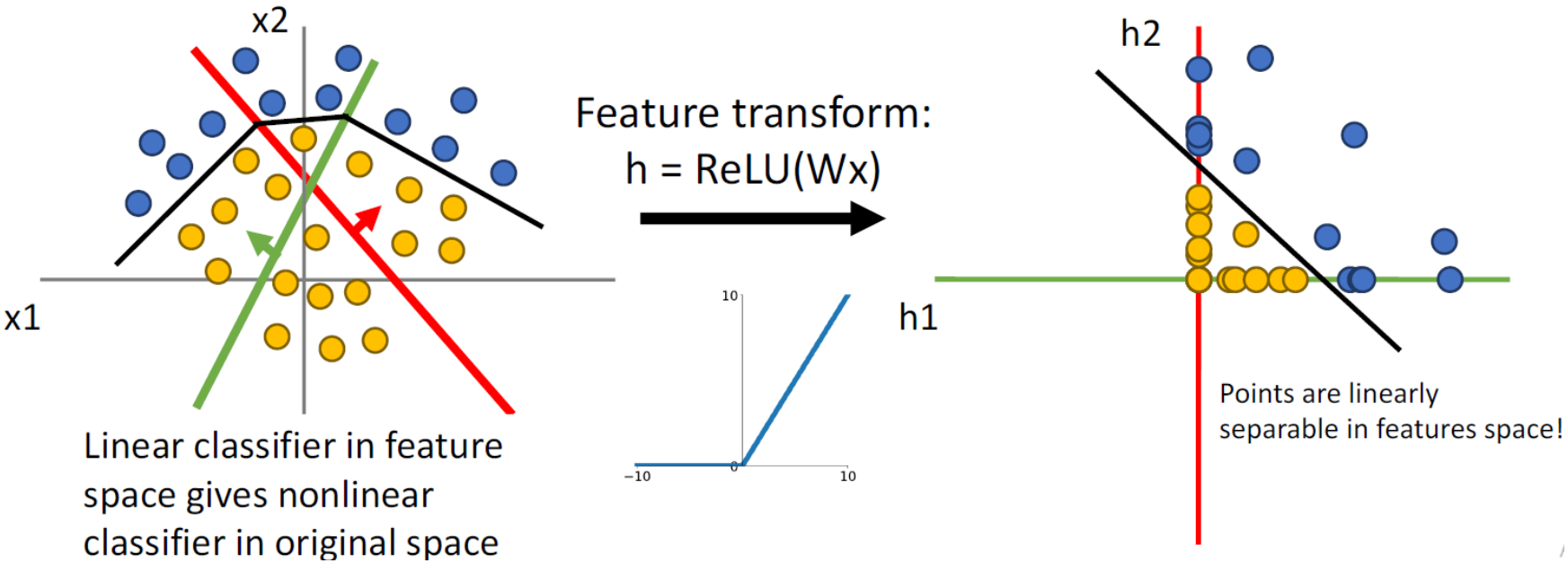
# Space warping

- Consider a neural net hidden layer: $\boldsymbol{h} = ReLU(W\boldsymbol{x}) = max(0, W\boldsymbol{x})$.
  - Where x, h are both 2-dimensional.
- Points not linearly separable in original space

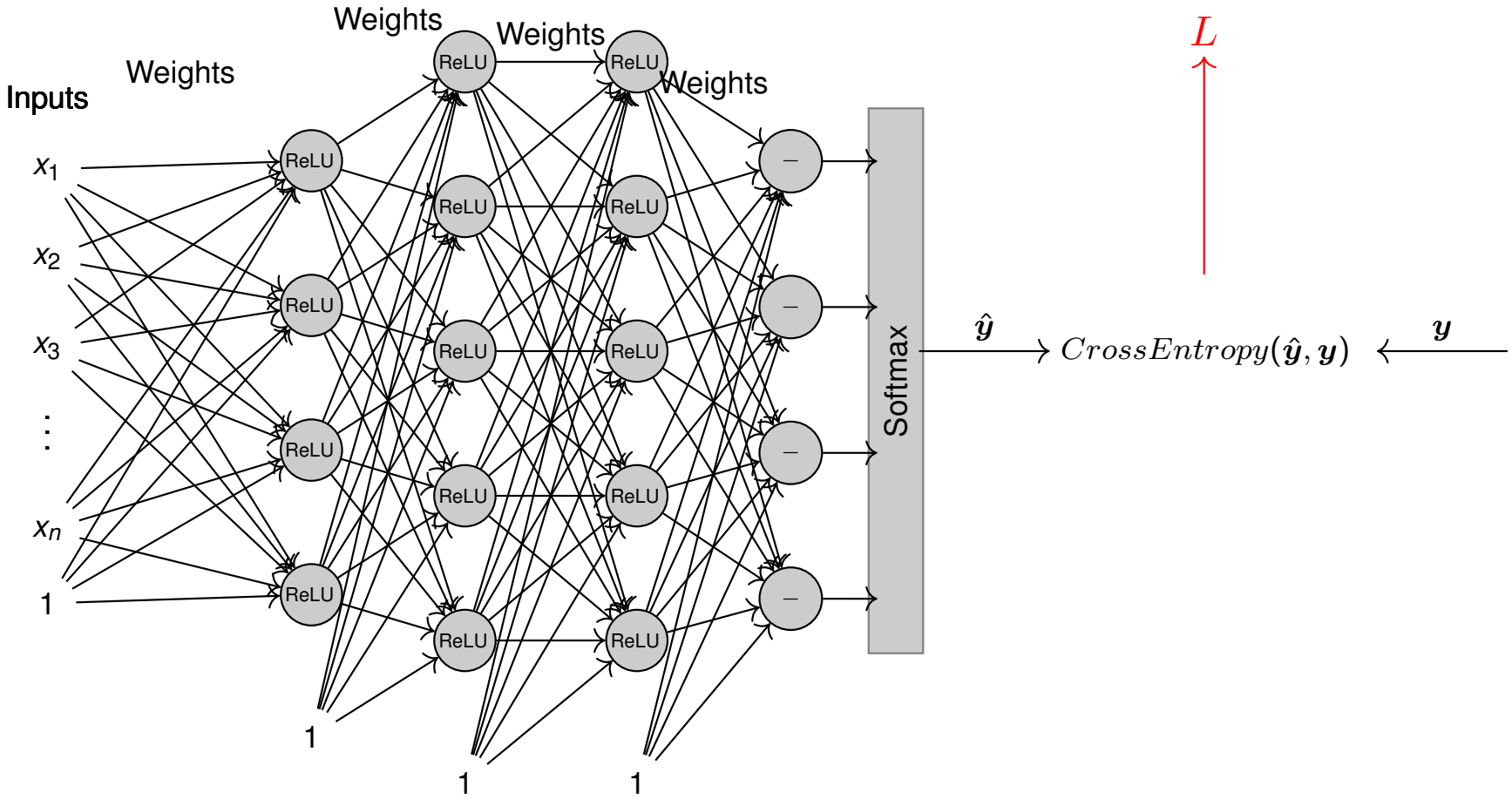


# Space warping

- Consider a neural net hidden layer: $\boldsymbol{h} = ReLU(W\boldsymbol{x}) = max(0, W\boldsymbol{x})$.
  - Where x, h are both 2-dimensional.
- Points not linearly separable in original space



# Fully connected neural network loss




$$\hat{y} = Softmax(W^4(max(0, W^3(max(0, W^2(max(0, W^1 x + b^1)) + b^2)) + b^3)) + b^4)$$

# Backpropagation

# Gradient descent

- The gradient descent method to find the minimum of a function iteratively

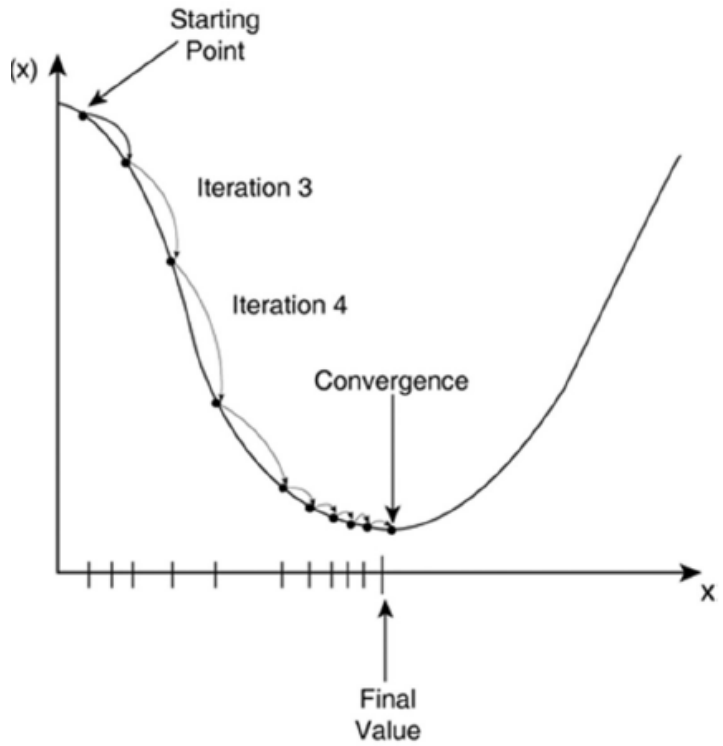
$$\boldsymbol{x}^{t+1} = \boldsymbol{x}^t - \eta \nabla_{\boldsymbol{x}} f(\boldsymbol{x}^t)$$

  - $\eta$ is the "step size" (Also called "learning rate")

- The gradient descent algorithm converges when the following criterion is satisfied.

$$|f(\boldsymbol{x}^{t+k}) - f(\boldsymbol{x}^t)| < \epsilon$$

  - $k$ is a hyperparameter.



# Training neural nets through gradient descent

- Total training loss

$$L(f(X, W), \boldsymbol{y}) = \frac{1}{N} \sum_i L_i(f(\boldsymbol{x}^{(i)}, W), y^{(i)})$$

- Gradient descent algorithm:
  - Initialize all weights and biases $w_{ij}^{(\ell)}$
    - Using the extended notation: the bias is also a weight
  - Do:
    - For every layer $\ell$ for all $i$, $j$ update:

$$w_{ij}^{(\ell)} = w_{ij}^{(\ell)} - \eta \frac{\partial L}{\partial w_{ij}^{(\ell)}}$$

  - Until loss has converged

# The derivative

- Total training Loss:

$$L(f(X, W), \boldsymbol{y}) = \frac{1}{N} \sum_i L_i(f(\boldsymbol{x}^{(i)}, W), y^{(i)})$$

- Computing the derivative

$$\frac{\partial L(f(X, W), \boldsymbol{y})}{\partial w_{ij}^{(\ell)}} = \frac{1}{N} \sum_i \frac{\partial L_i(f(\boldsymbol{x}^{(i)}, W), y^{(i)})}{\partial w_{ij}^{(\ell)}}$$

- So we must first figure out how to compute the derivative of divergences of individual training inputs

# Training by gradient descent

- Initialize all weights $w_{ij}^{(\ell)}$
- Do
  - For all $i, j, \ell$, initialize $\frac{\partial L}{\partial w_{ij}^{(\ell)}} = 0$
  - For all $n = 1 : N$
    - For every $\ell$ for all $i, j$:

      Compute $\frac{\partial L_n}{\partial w_{ij}^{(\ell)}}$

      $$\frac{\partial L}{\partial w_{ij}^{(\ell)}} += \frac{1}{N} \frac{\partial L_n}{\partial w_{ij}^{(\ell)}}$$
  - For every $\ell$ for all $i, j$:
    $$w_{ij}^{(\ell)} = w_{ij}^{(\ell)} - \eta \frac{\partial L}{w_{ij}^{(\ell)}}$$
- Until loss has converged

# Calculus refresher: chain rule

For any nested function $y = f(g(x))$

$$\frac{dy}{dx} = \frac{df}{dg(x)} \frac{dg(x)}{dx}$$

Check - we can confirm that : $\Delta y = \frac{dy}{dx} \Delta x$

$$z = g(x) \Rightarrow \Delta z = \frac{dg(x)}{dx} \Delta x$$

$$y = f(z) \Rightarrow \Delta y = \frac{df}{dz} \Delta z = \frac{df}{dg(x)} \frac{dg(x)}{dx} \Delta x$$

# Calculus refresher: distributed chain rule

$$y = f\left(g_1(x), g_2(x), \ldots, g_M(x)\right)$$

$$\frac{dy}{dx} = \frac{\partial f}{\partial g_1(x)}\frac{dg_1(x)}{dx} + \frac{\partial f}{\partial g_2(x)}\frac{dg_2(x)}{dx} + \cdots + \frac{\partial f}{\partial g_M(x)}\frac{dg_M(x)}{dx}$$

Check: $\Delta y = \frac{dy}{dx}\Delta x$ Let $z_i = g_i(x)$

$$\Delta y = \frac{\partial f}{\partial z_1}\Delta z_1 + \frac{\partial f}{\partial z_2}\Delta z_2 + \cdots + \frac{\partial f}{\partial z_M}\Delta z_M$$

$$\Delta y = \frac{\partial f}{\partial z_1}\frac{dz_1}{dx}\Delta x + \frac{\partial f}{\partial z_2}\frac{dz_2}{dx}\Delta x + \cdots + \frac{\partial f}{\partial z_M}\frac{dz_M}{dx}\Delta x$$

$$\Delta y = \left(\frac{\partial f}{\partial g_1(x)}\frac{dg_1(x)}{dx} + \frac{\partial f}{\partial g_2(x)}\frac{dg_2(x)}{dx} + \cdots + \frac{\partial f}{\partial g_M(x)}\frac{dg_M(x)}{dx}\right)\Delta x$$

# Calculus refresher: distributed chain rule

$$y = f\left(g_1(x), g_2(x), \ldots, g_M(x)\right)$$

$$\frac{dy}{dx} = \frac{\partial f}{\partial g_1(x)}\frac{dg_1(x)}{dx} + \frac{\partial f}{\partial g_2(x)}\frac{dg_2(x)}{dx} + \cdots + \frac{\partial f}{\partial g_M(x)}\frac{dg_M(x)}{dx}$$

Check: $\Delta y = \frac{dy}{dx}\Delta x$

$$\Delta y = \frac{\partial f}{\partial g_1(x)}\Delta g_1(x) + \frac{\partial f}{\partial g_2(x)}\Delta g_2(x) + \cdots + \frac{\partial f}{\partial g_M(x)}\Delta g_M(x)$$

$$\Delta y = \frac{\partial f}{\partial g_1(x)}\frac{dg_1(x)}{dx}\Delta x + \frac{\partial f}{\partial g_2(x)}\frac{dg_2(x)}{dx}\Delta x + \cdots + \frac{\partial f}{\partial g_M(x)}\frac{dg_M(x)}{dx}\Delta x$$

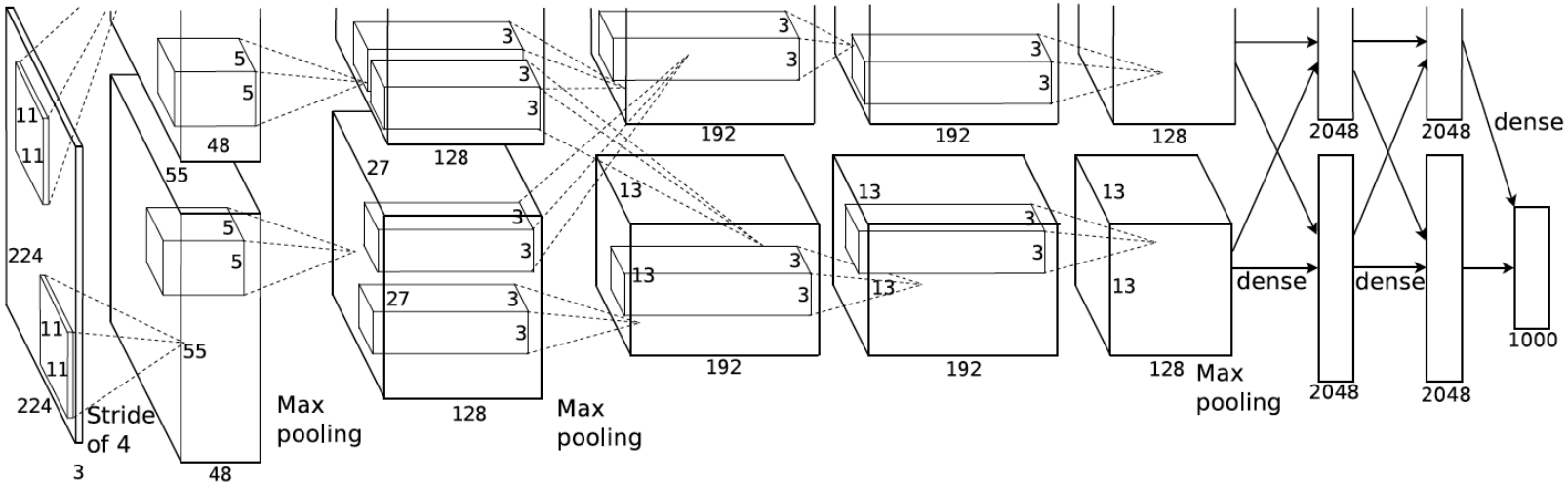
$$\Delta y = \left(\frac{\partial f}{\partial g_1(x)}\frac{dg_1(x)}{dx} + \frac{\partial f}{\partial g_2(x)}\frac{dg_2(x)}{dx} + \cdots + \frac{\partial f}{\partial g_M(x)}\frac{dg_M(x)}{dx}\right)\Delta x$$

# Idea: derive $\frac{\partial L}{\partial W}$ on paper

- Problems
  - Very tedious: Lots of matrix calculus, need lots of paper
  - What if we want to change loss? Need to re-derive from scratch. Not modular!
  - Not feasible for very complex models!

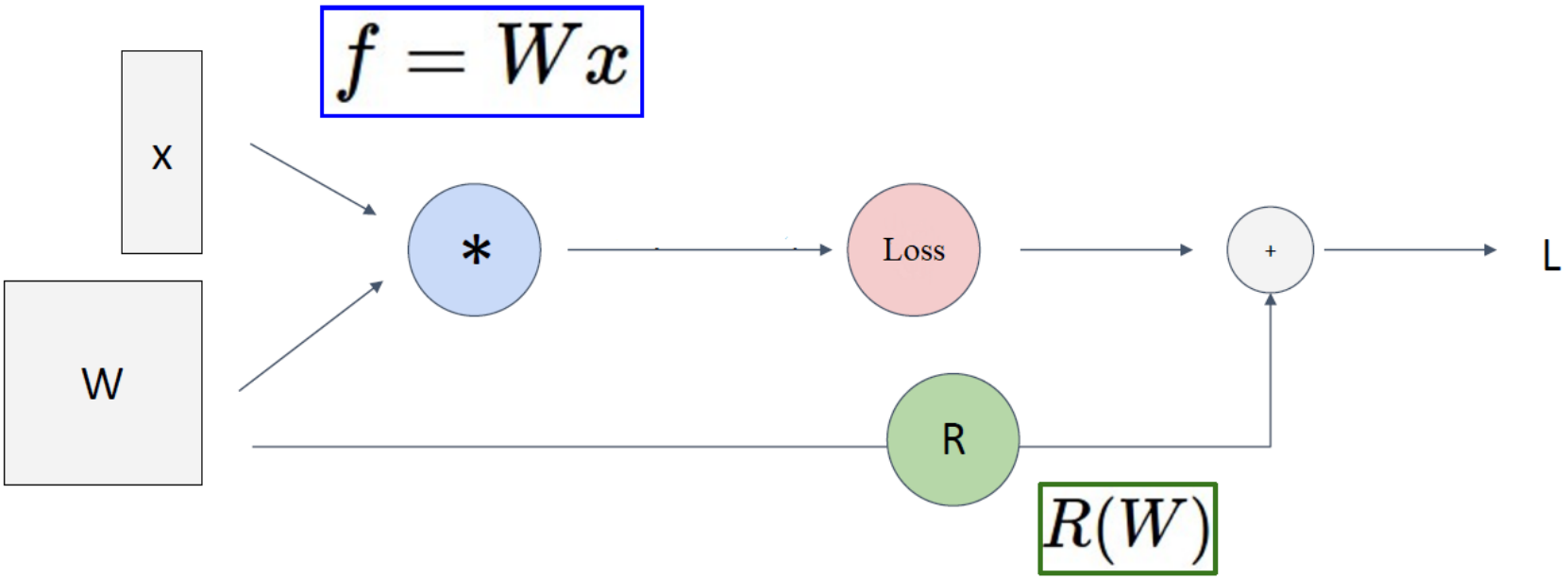
# Idea: derive $\frac{\partial L}{\partial W}$ on paper

- Problems
  - Very tedious: Lots of matrix calculus, need lots of paper
  - What if we want to change loss? Need to re-derive from scratch. Not modular!
  - Not feasible for very complex models!



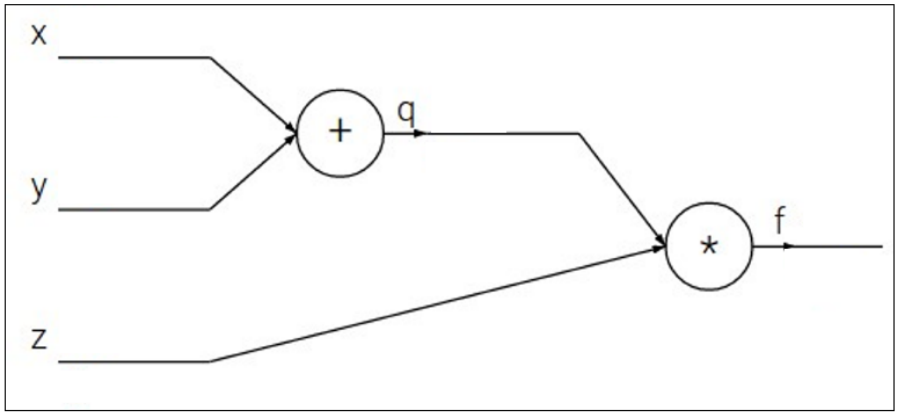

AlexNet has about 660K units, 61M parameters,

# Better Idea: Computational Graphs



# Backpropagation: simple example

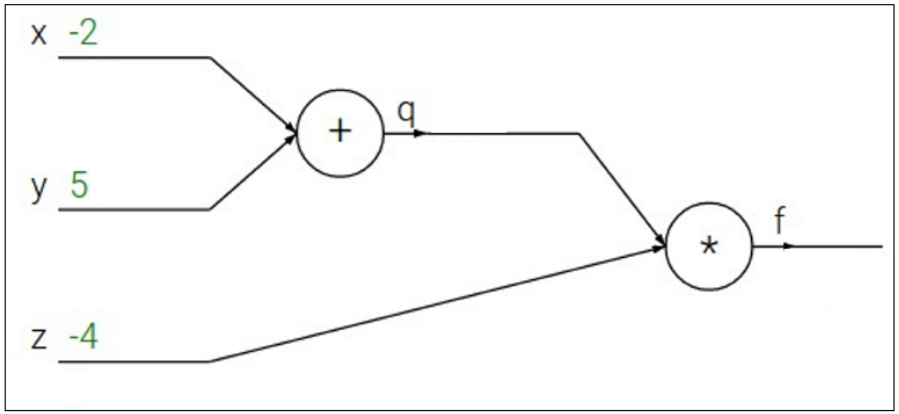
$$f(x, y, z) = (x + y)z$$



# Backpropagation: simple example

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

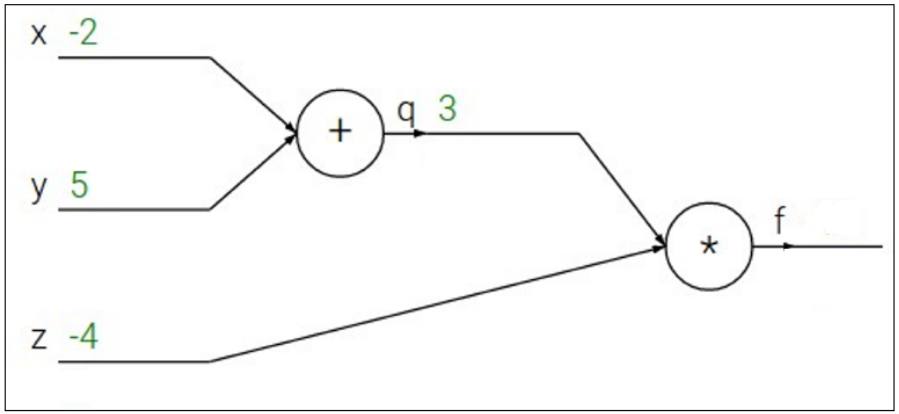
e.g. x = -2, y = 5, z = -4



# Backpropagation: simple example

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

e.g. x = -2, y = 5, z = -4



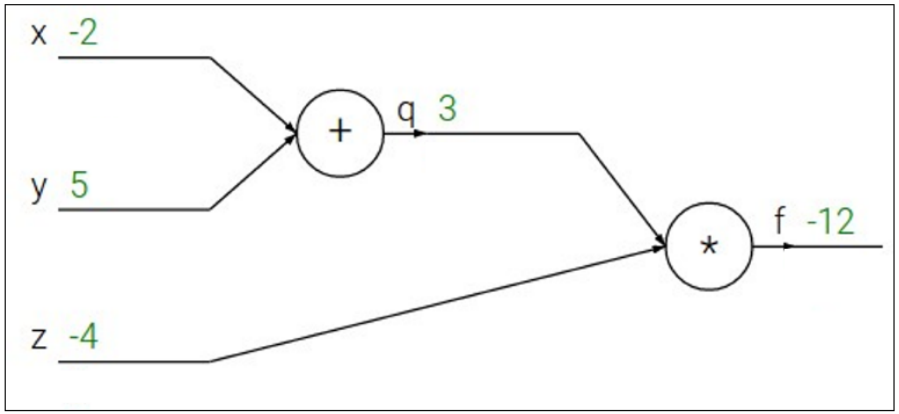

**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

# Backpropagation: simple example

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

e.g. x = -2, y = 5, z = -4



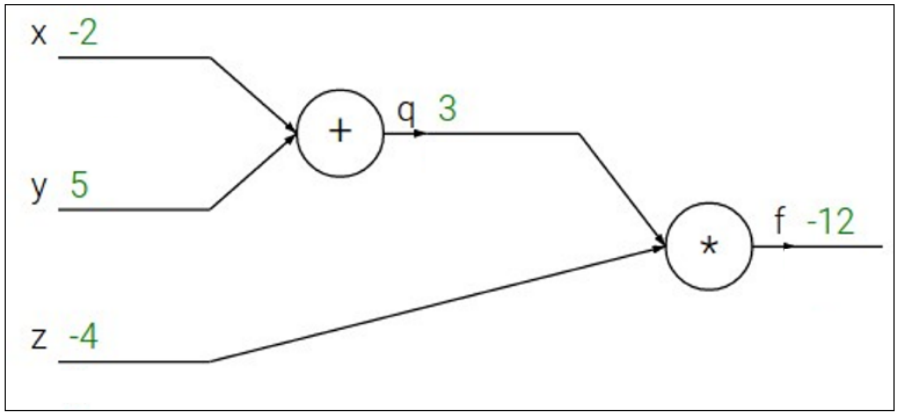

**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

# Backpropagation: simple example

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

e.g. x = -2, y = 5, z = -4




**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

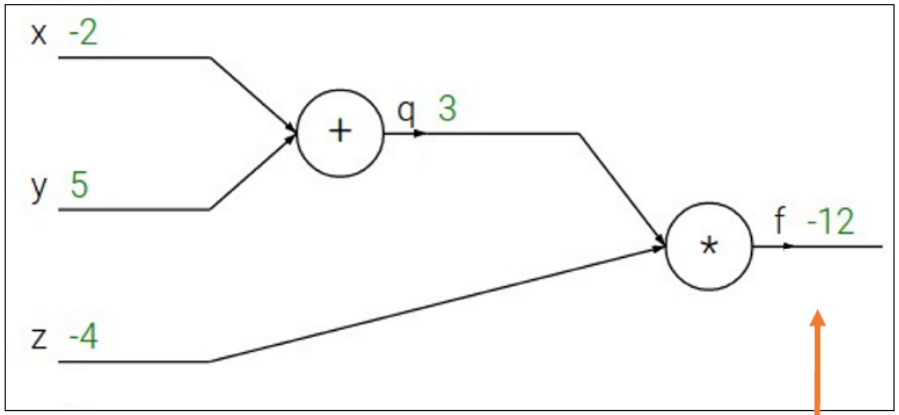
**2. Backward pass**: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

# Backpropagation: simple example

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

e.g. x = -2, y = 5, z = -4




**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

**2. Backward pass**: Compute derivatives

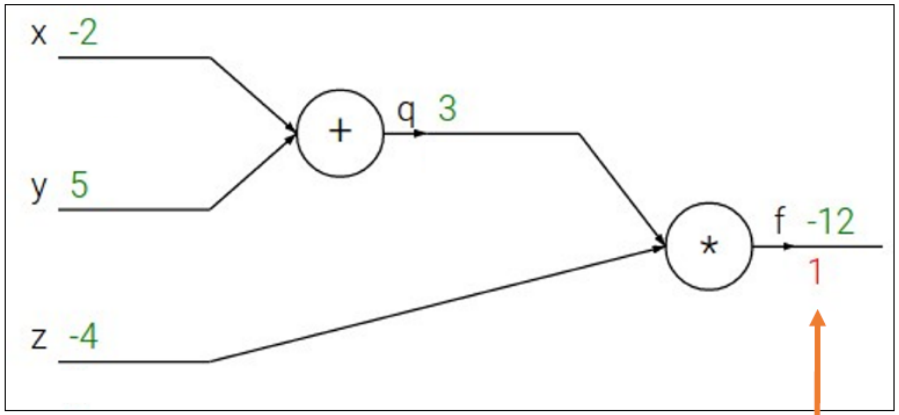
Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

$\frac{\partial f}{\partial f}$

# Backpropagation: simple example

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

e.g. x = -2, y = 5, z = -4




**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

**2. Backward pass**: Compute derivatives

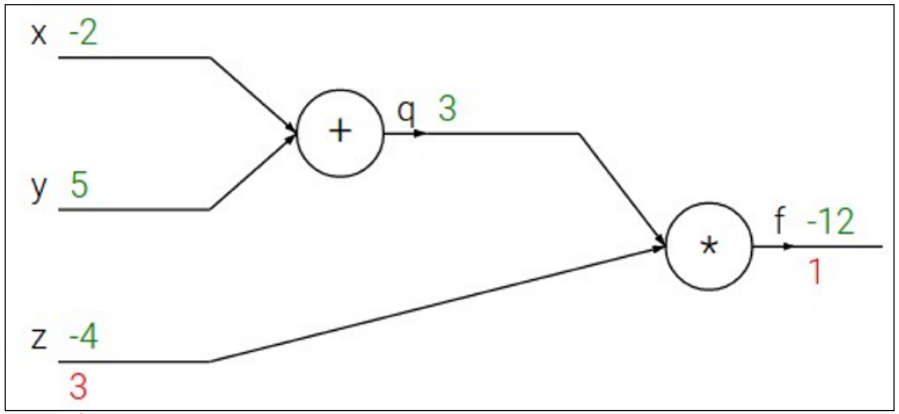
Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

$$\frac{\partial f}{\partial f}$$

# Backpropagation: simple example

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

e.g. x = -2, y = 5, z = -4




**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

**2. Backward pass**: Compute derivatives

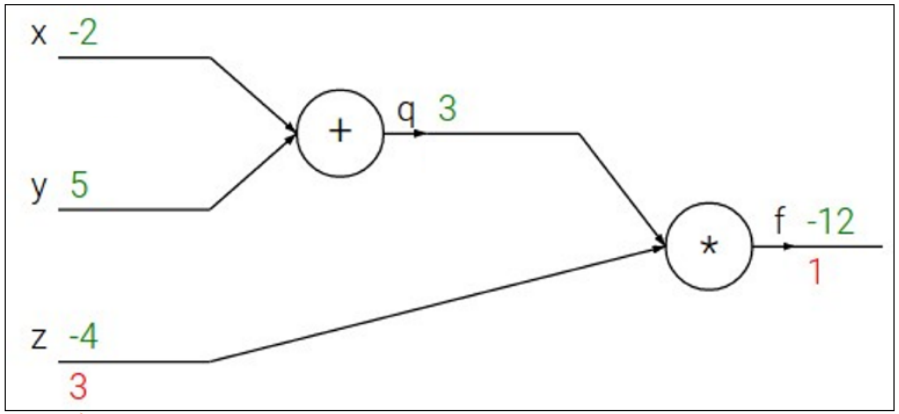
$$\frac{\partial f}{\partial z}$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

# Backpropagation: simple example

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

e.g. x = -2, y = 5, z = -4




**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

**2. Backward pass**: Compute derivatives

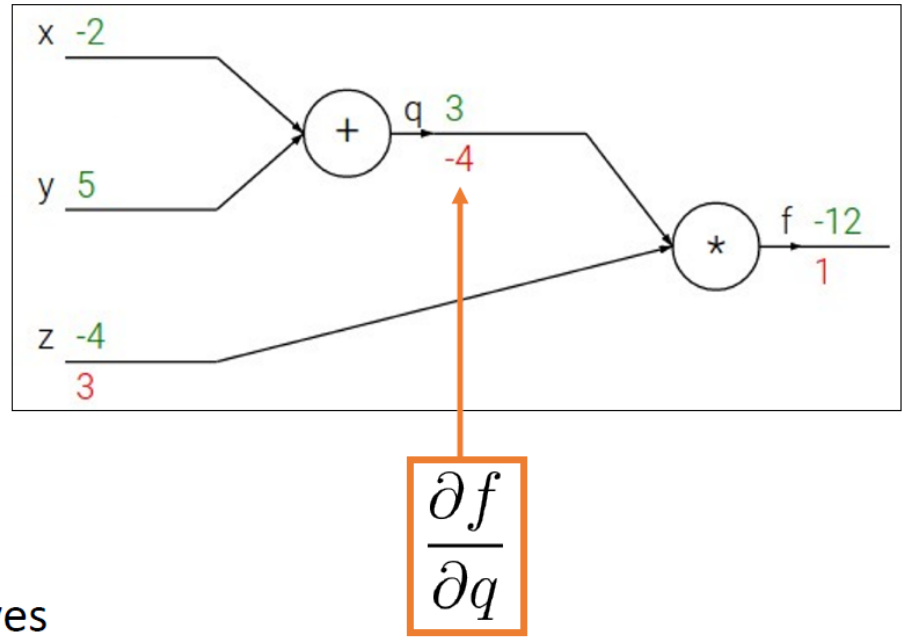
Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

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

# Backpropagation: simple example

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

e.g. x = -2, y = 5, z = -4




**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

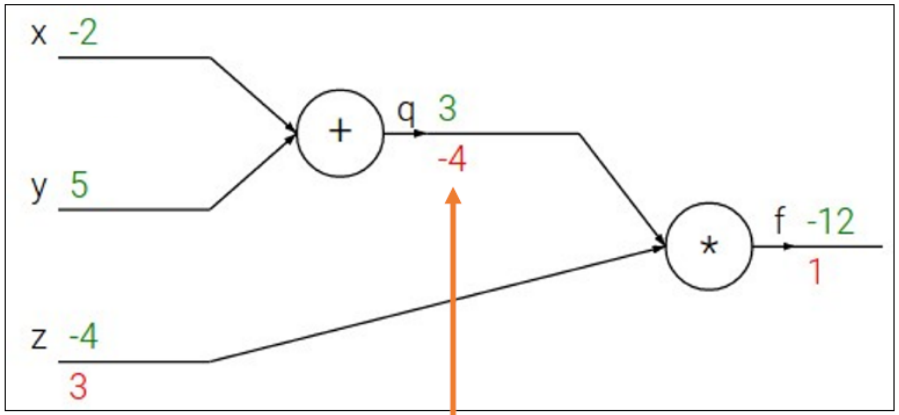
**2. Backward pass**: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

# Backpropagation: simple example

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

e.g. x = -2, y = 5, z = -4




**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

**2. Backward pass**: Compute derivatives

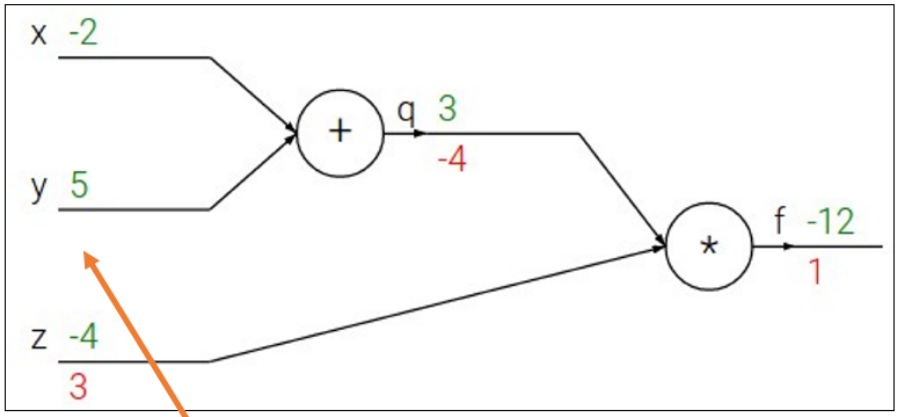
$$\frac{\partial f}{\partial q} = z$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

# Backpropagation: simple example

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

e.g. x = -2, y = 5, z = -4




**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

**2. Backward pass**: Compute derivatives

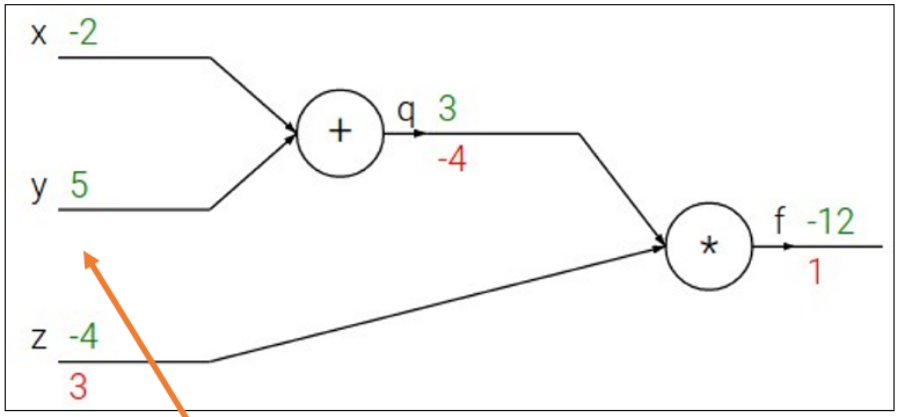
Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

$\frac{\partial f}{\partial y}$

# Backpropagation: simple example

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

e.g. x = -2, y = 5, z = -4




**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

**2. Backward pass**: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

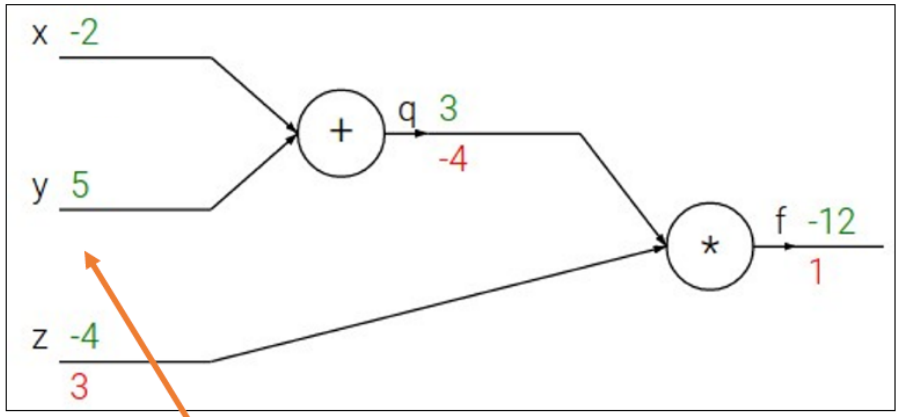
**Chain Rule**

$$\frac{\partial f}{\partial y} = \frac{\partial q}{\partial y}\frac{\partial f}{\partial q}$$

# Backpropagation: simple example

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

e.g. x = -2, y = 5, z = -4




**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

**2. Backward pass**: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

**Chain Rule**

$$\frac{\partial f}{\partial y} = \frac{\partial q}{\partial y}\frac{\partial f}{\partial q}$$

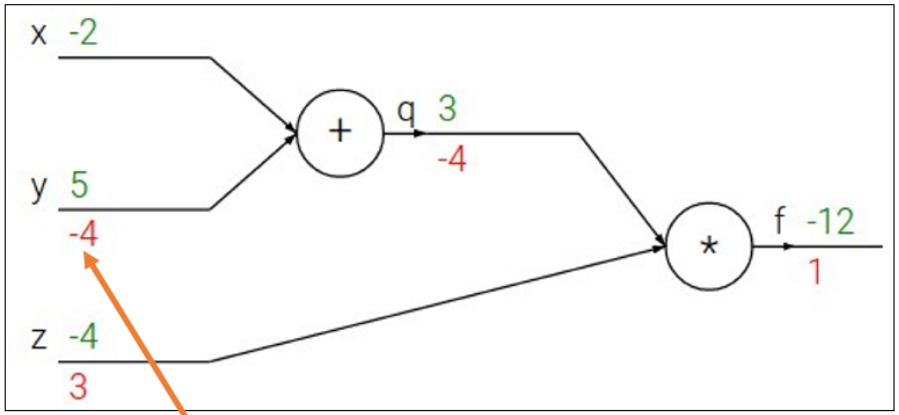
$$\frac{\partial q}{\partial y} = 1$$

**Downstream Gradient** | **Local Gradient** | **Upstream Gradient**

# Backpropagation: simple example

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

e.g. x = -2, y = 5, z = -4




**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

**2. Backward pass**: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

**Chain Rule**

$$\frac{\partial f}{\partial y} = \frac{\partial q}{\partial y}\frac{\partial f}{\partial q}$$

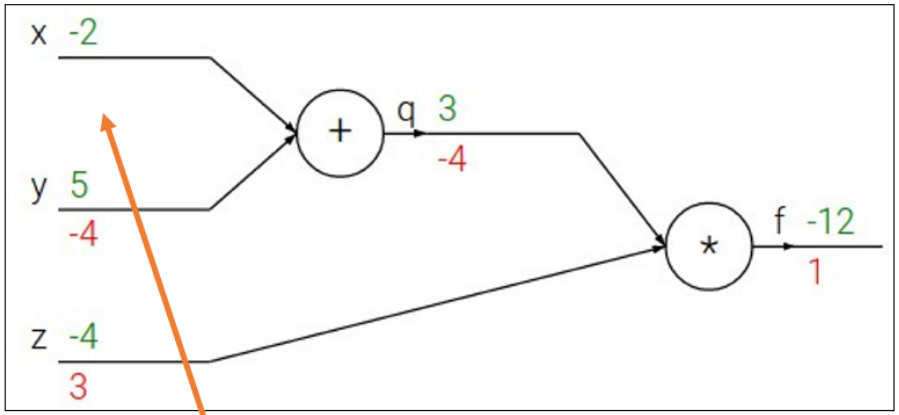
$$\frac{\partial q}{\partial y} = 1$$

**Downstream Gradient** **Local Gradient** **Upstream Gradient**

# Backpropagation: simple example

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

e.g. x = -2, y = 5, z = -4




**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

**2. Backward pass**: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

**Chain Rule**

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

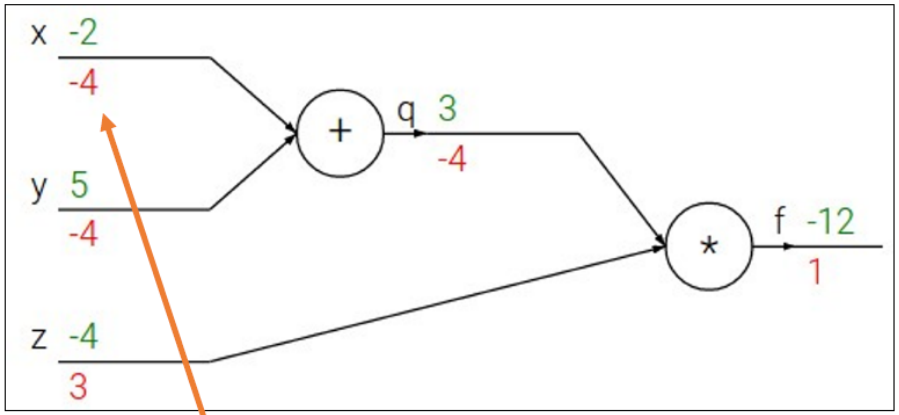
$$\frac{\partial q}{\partial x} = 1$$

**Downstream Gradient** **Local Gradient** **Upstream Gradient**

# Backpropagation: simple example

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

e.g. x = -2, y = 5, z = -4




**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

**2. Backward pass**: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

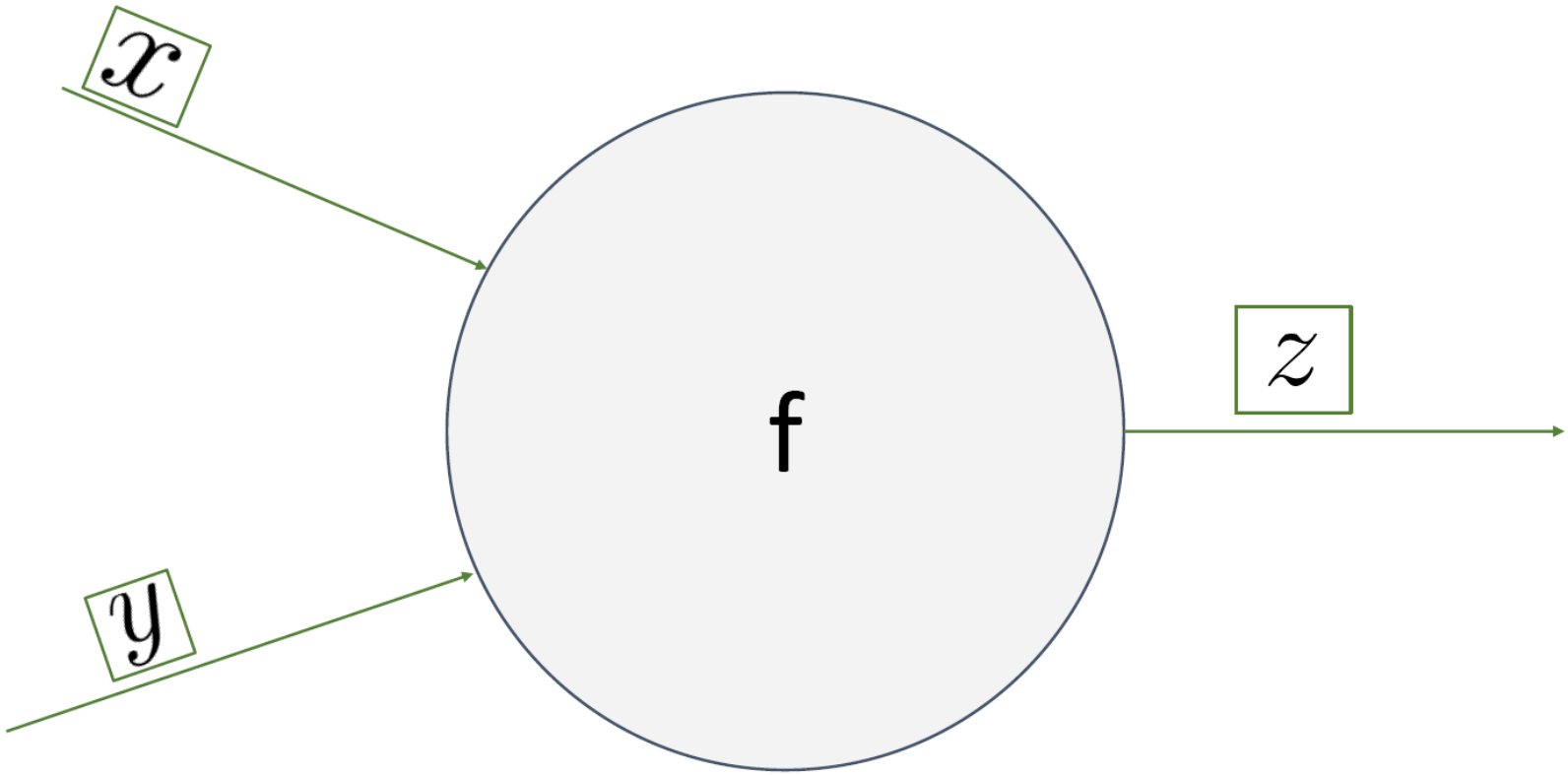
**Chain Rule**

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

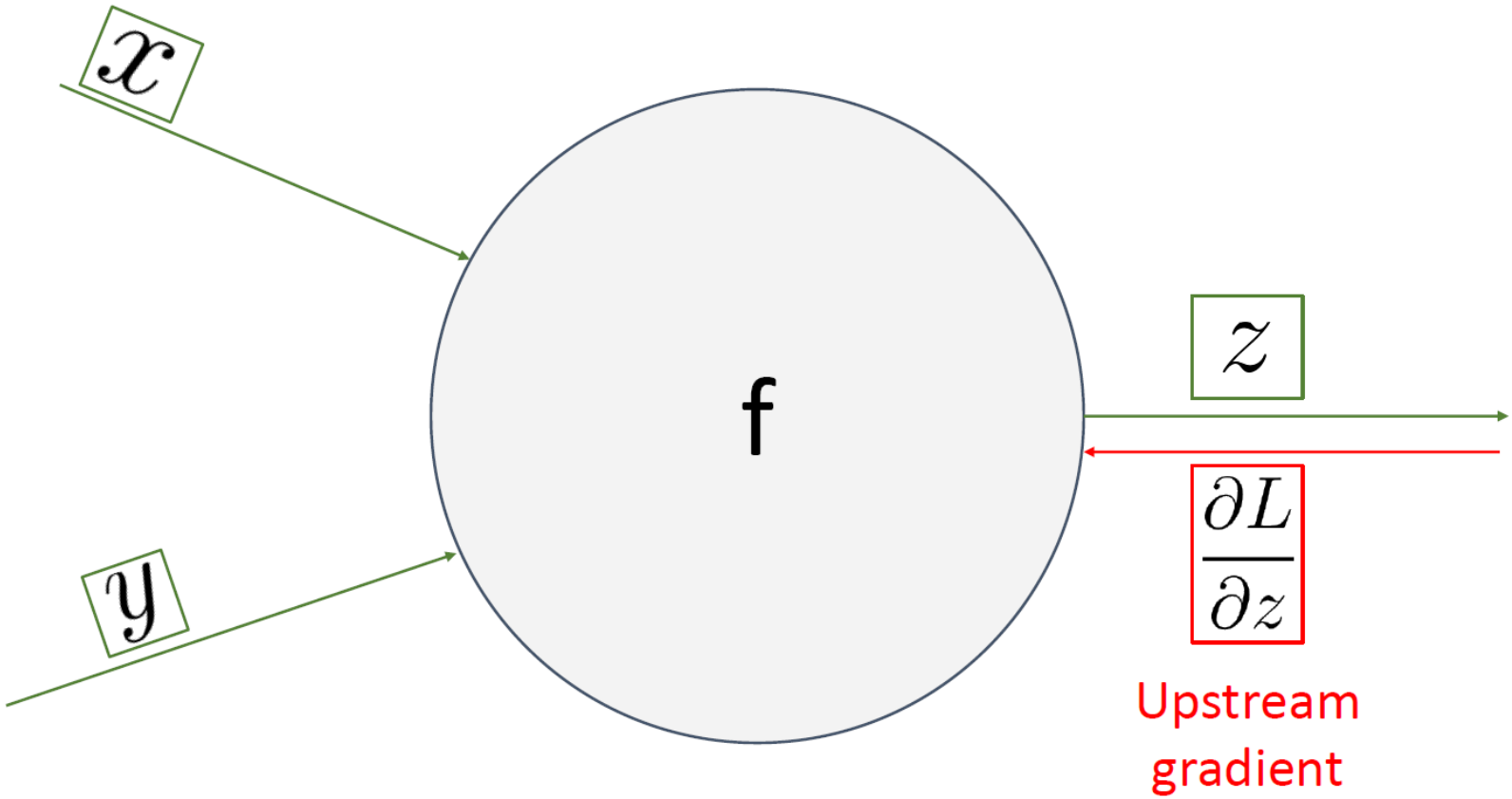
$$\frac{\partial q}{\partial x} = 1$$

**Downstream Gradient** **Local Gradient** **Upstream Gradient**

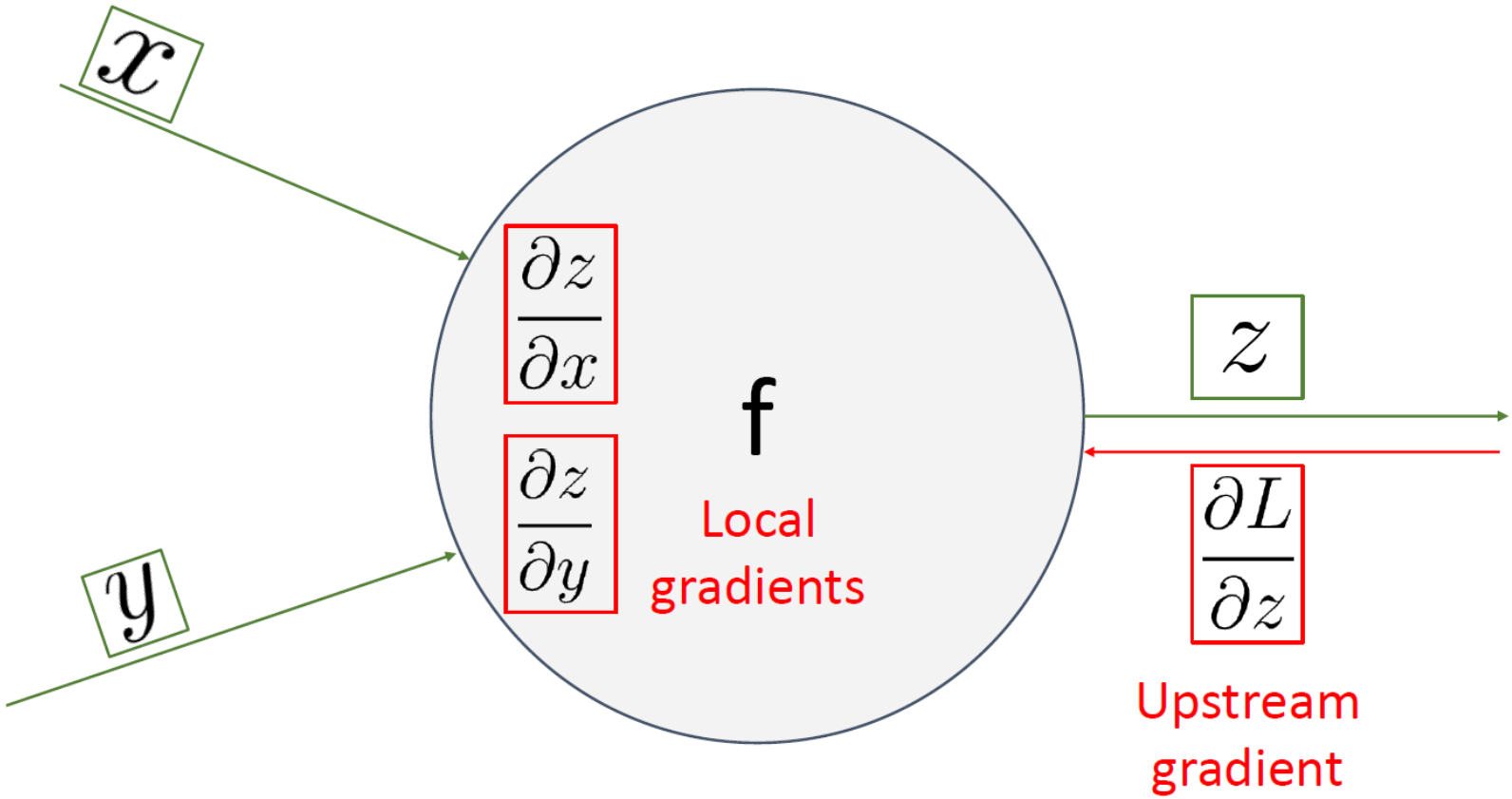
# Up/Down stream gradients



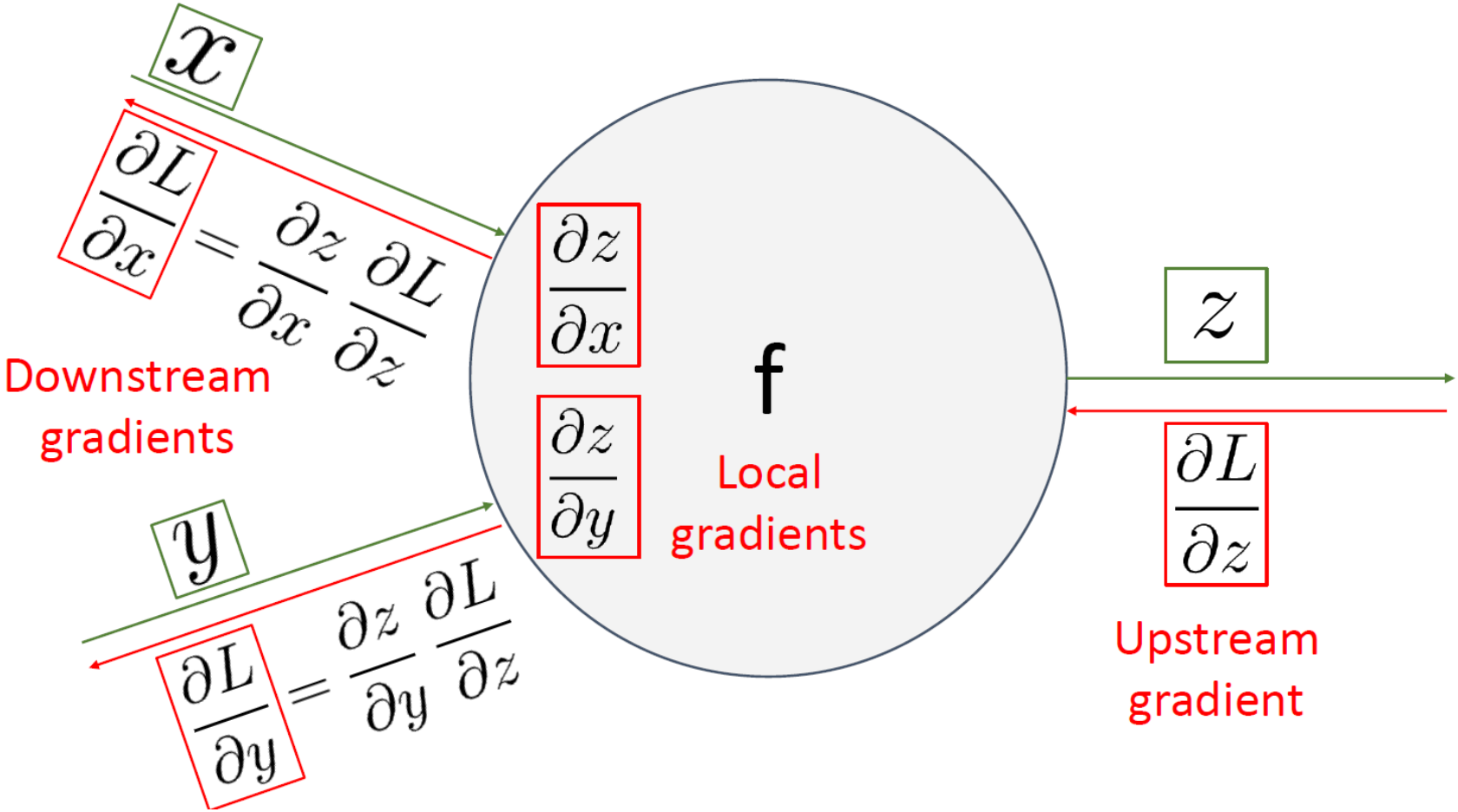
# Up/Down stream gradients



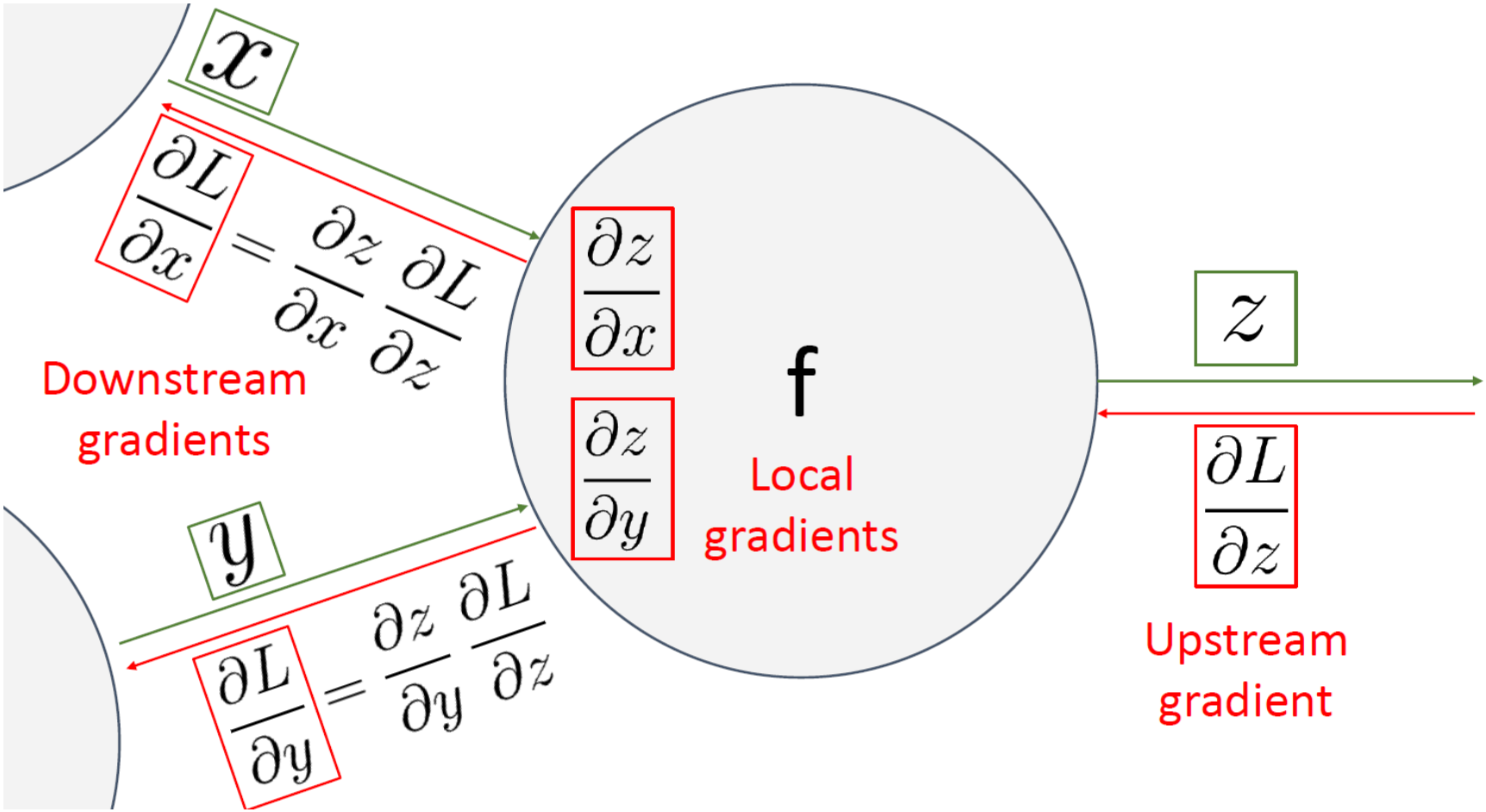
# Up/Down stream gradients



# Up/Down stream gradients

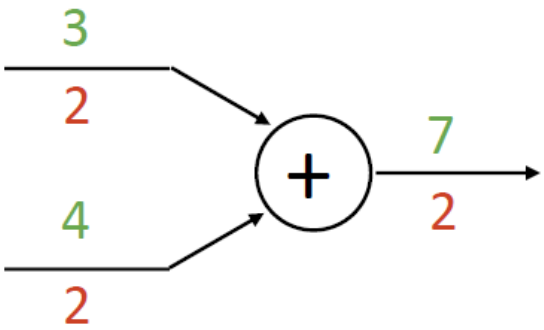


# Up/Down stream gradients



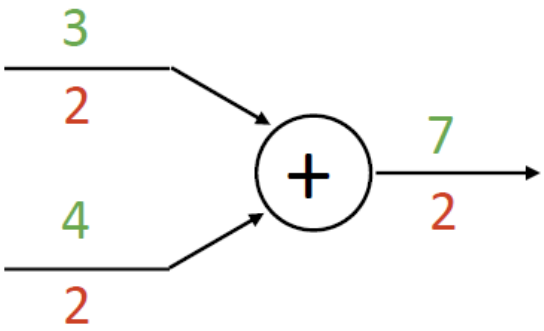
# Patterns in Gradient Flow

**add** gate: gradient distributor

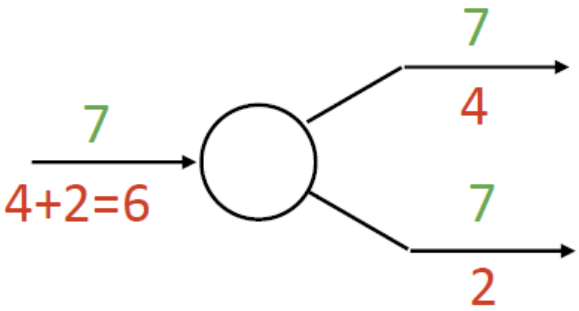


# Patterns in Gradient Flow

**add** gate: gradient distributor

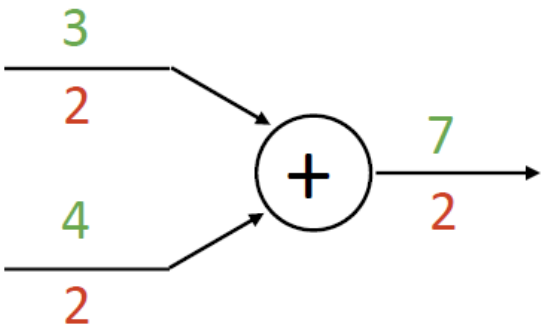



**copy** gate: gradient adder

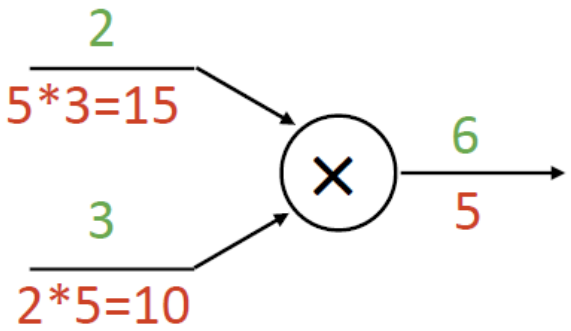


# Patterns in Gradient Flow

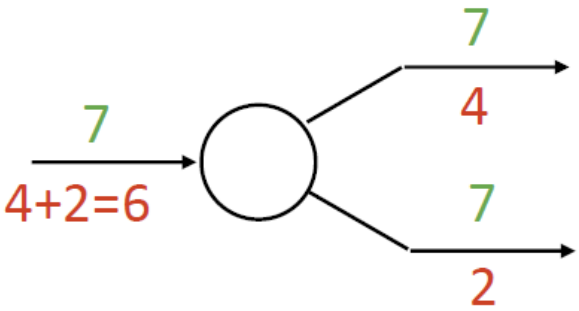
**add** gate: gradient distributor




**mul** gate: “swap multiplier”

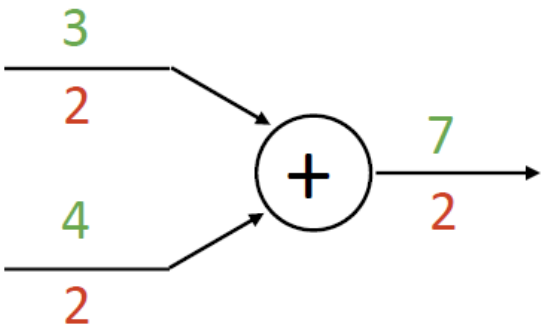



**copy** gate: gradient adder

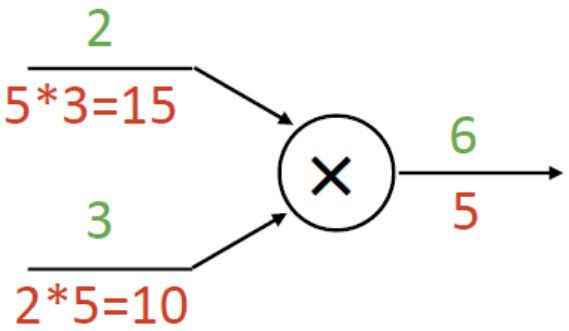


# Patterns in Gradient Flow

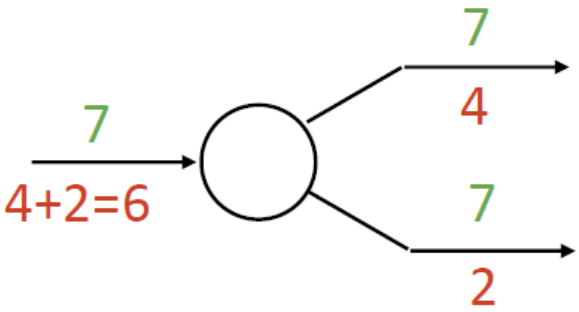
**add** gate: gradient distributor



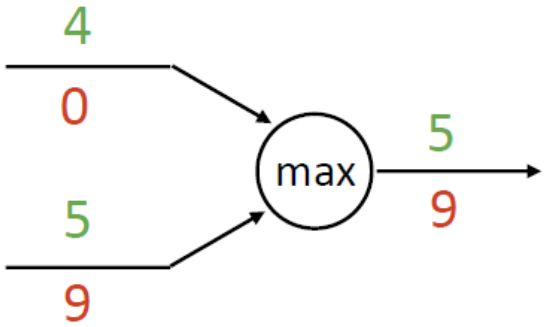

**mul** gate: "swap multiplier"




**copy** gate: gradient adder

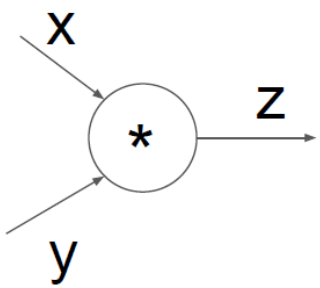



**max** gate: gradient router



# Modularized implementation: forward / backward API

- Actual PyTorch code




(x,y,z are scalars)

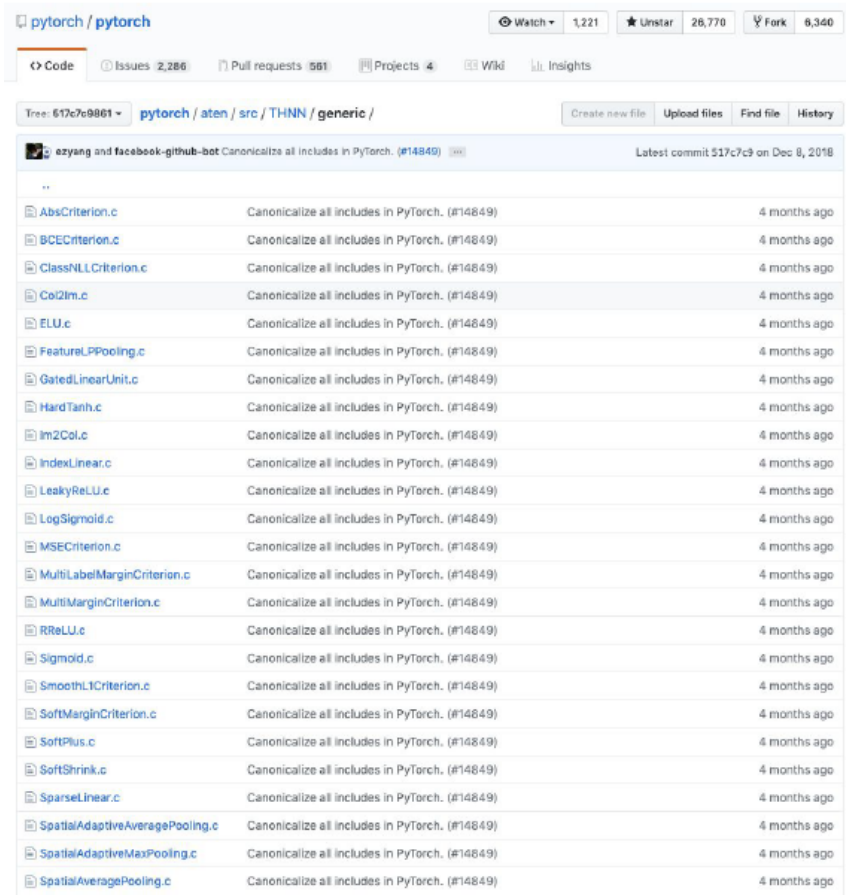
```
class Multiply(torch.autograd.Function):
  @staticmethod
  def forward(ctx, x, y):
    ctx.save_for_backward(x, y)
    z = x * y
    return z
  @staticmethod
  def backward(ctx, grad_z):
    x, y = ctx.saved_tensors
    grad_x = y * grad_z   # dz/dx * dL/dz
    grad_y = x * grad_z   # dz/dy * dL/dz
    return grad_x, grad_y
```

Need to stash some values for use in backward

Upstream gradient

Multiply upstream and local gradients

# Example: PyTorch operators



pytorch / pytorch

Watch 1,221 | Unstar 26,770 | Fork 8,340

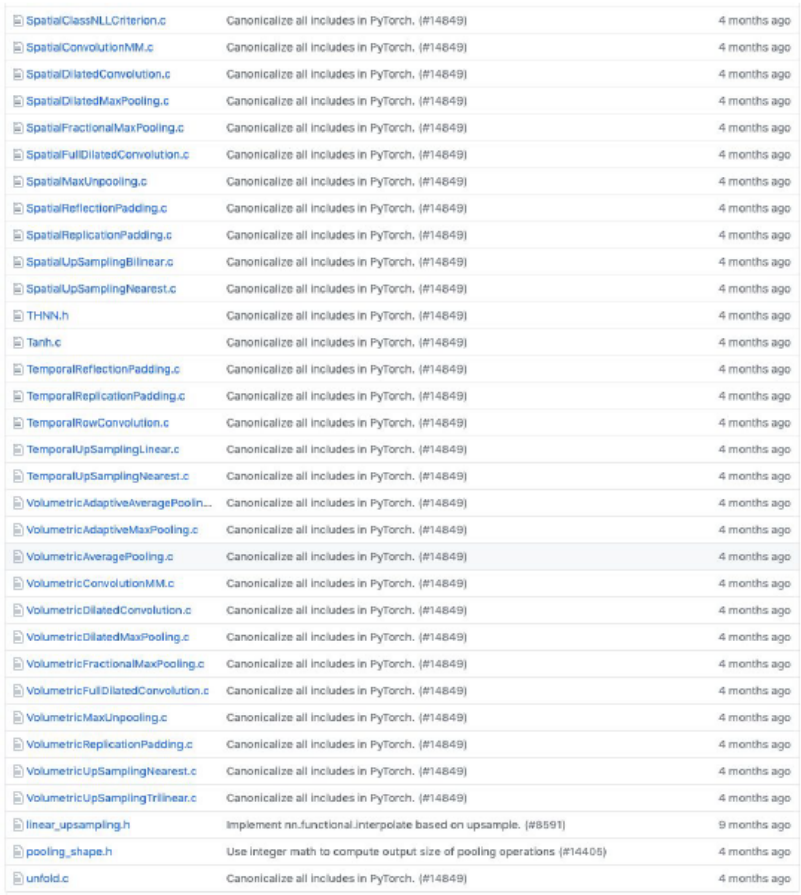
Code | Issues 2,286 | Pull requests 561 | Projects 4 | Wiki | Insights

Tree: 517c7c9861 ▾ | pytorch / aten / src / THNN / generic / | Create new file | Upload files | Find file | History

ezyang and facebook-github-bot Canonicalize all includes in PyTorch. (#14849) ... Latest commit 517c7c9 on Dec 8, 2018

..

| | | |
|---|---|---|
| AbsCriterion.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| BCECriterion.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| ClassNLLCriterion.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| Col2Im.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| ELU.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| FeatureLPPooling.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| GatedLinearUnit.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| HardTanh.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| Im2Col.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| IndexLinear.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| LeakyReLU.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| LogSigmoid.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| MSECriterion.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| MultiLabelMarginCriterion.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| MultiMarginCriterion.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| RReLU.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| Sigmoid.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SmoothL1Criterion.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SoftMarginCriterion.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SoftPlus.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SoftShrink.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SparseLinear.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SpatialAdaptiveAveragePooling.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SpatialAdaptiveMaxPooling.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SpatialAveragePooling.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |



| | | |
|---|---|---|
| SpatialClassNLLCriterion.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SpatialConvolutionMM.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SpatialDilatedConvolution.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SpatialDilatedMaxPooling.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SpatialFractionalMaxPooling.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SpatialFullDilatedConvolution.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SpatialMaxUnpooling.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SpatialReflectionPadding.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SpatialReplicationPadding.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SpatialUpSamplingBilinear.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SpatialUpSamplingNearest.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| THNN.h | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| Tanh.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| TemporalReflectionPadding.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| TemporalReplicationPadding.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| TemporalRowConvolution.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| TemporalUpSamplingLinear.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| TemporalUpSamplingNearest.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| VolumetricAdaptiveAveragePoolin... | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| VolumetricAdaptiveMaxPooling.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| VolumetricAveragePooling.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| VolumetricConvolutionMM.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| VolumetricDilatedConvolution.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| VolumetricDilatedMaxPooling.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| VolumetricFractionalMaxPooling.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| VolumetricFullDilatedConvolution.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| VolumetricMaxUnpooling.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| VolumetricReplicationPadding.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| VolumetricUpSamplingNearest.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| VolumetricUpSamplingTrilinear.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| linear_upsampling.h | Implement nn.functional.interpolate based on upsample. (#8591) | 9 months ago |
| pooling_shape.h | Use integer math to compute output size of pooling operations (#14405) | 4 months ago |
| unfold.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |

# PyTorch sigmoid layer

```
#ifndef TH_GENERIC_FILE
#define TH_GENERIC_FILE "THNN/generic/Sigmoid.c"
#else

void THNN_(Sigmoid_updateOutput)(
          THNNState *state,
          THTensor *input,
          THTensor *output)
{
  THTensor_(sigmoid)(output, input);
}

void THNN_(Sigmoid_updateGradInput)(
          THNNState *state,
          THTensor *gradOutput,
          THTensor *gradInput,
          THTensor *output)
{
  THNN_CHECK_NELEMENT(output, gradOutput);
  THTensor_(resizeAs)(gradInput, output);
  TH_TENSOR_APPLY3(scalar_t, gradInput, scalar_t, gradOutput, scalar_t, output,
    scalar_t z = *output_data;
    *gradInput_data = *gradOutput_data * (1. - z) * z;
  );
}

#endif
```

Forward

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

Source

# PyTorch sigmoid layer

```
#ifndef TH_GENERIC_FILE
#define TH_GENERIC_FILE "THNN/generic/Sigmoid.c"
#else

void THNN_(Sigmoid_updateOutput)(
          THNNState *state,
          THTensor *input,
          THTensor *output)
{
  THTensor_(sigmoid)(output, input);
}

void THNN_(Sigmoid_updateGradInput)(
          THNNState *state,
          THTensor *gradOutput,
          THTensor *gradInput,
          THTensor *output)
{
  THNN_CHECK_NELEMENT(output, gradOutput);
  THTensor_(resizeAs)(gradInput, output);
  TH_TENSOR_APPLY3(scalar_t, gradInput, scalar_t, gradOutput, scalar_t, output,
    scalar_t z = *output_data;
    *gradInput_data = *gradOutput_data * (1. - z) * z;
  );
}

#endif
```

Forward

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

```
static void sigmoid_kernel(TensorIterator& iter) {
  AT_DISPATCH_FLOATING_TYPES(iter.dtype(), "sigmoid_cpu", [&]() {
    unary_kernel_vec(
        iter,
        [=](scalar_t a) -> scalar_t { return (1 / (1 + std::exp((-a)))); },
        [=](Vec256<scalar_t> a) {
          a = Vec256<scalar_t>((scalar_t)(0)) - a;
          a = a.exp();
          a = Vec256<scalar_t>((scalar_t)(1)) + a;
          a = a.reciprocal();
          return a;
        });
  });
}
```

Forward actually defined elsewhere...

```
return (1 / (1 + std::exp((-a))));
```

Source

# PyTorch sigmoid layer

```
#ifndef TH_GENERIC_FILE
#define TH_GENERIC_FILE "THNN/generic/Sigmoid.c"
#else

void THNN_(Sigmoid_updateOutput)(
          THNNState *state,
          THTensor *input,
          THTensor *output)
{
  THTensor_(sigmoid)(output, input);
}

void THNN_(Sigmoid_updateGradInput)(
          THNNState *state,
          THTensor *gradOutput,
          THTensor *gradInput,
          THTensor *output)
{
  THNN_CHECK_NELEMENT(output, gradOutput);
  THTensor_(resizeAs)(gradInput, output);
  TH_TENSOR_APPLY3(scalar_t, gradInput, scalar_t, gradOutput, scalar_t, output,
    scalar_t z = *output_data;
    *gradInput_data = *gradOutput_data * (1. - z) * z;
  );
}

#endif
```

Forward

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

```
static void sigmoid_kernel(TensorIterator& iter) {
  AT_DISPATCH_FLOATING_TYPES(iter.dtype(), "sigmoid_cpu", [&]() {
    unary_kernel_vec(
        iter,
        [=](scalar_t a) -> scalar_t { return (1 / (1 + std::exp((-a)))); },
        [=](Vec256<scalar_t> a) {
          a = Vec256<scalar_t>((scalar_t)(0)) - a;
          a = a.exp();
          a = Vec256<scalar_t>((scalar_t)(1)) + a;
          a = a.reciprocal();
          return a;
        });
  });
}
```

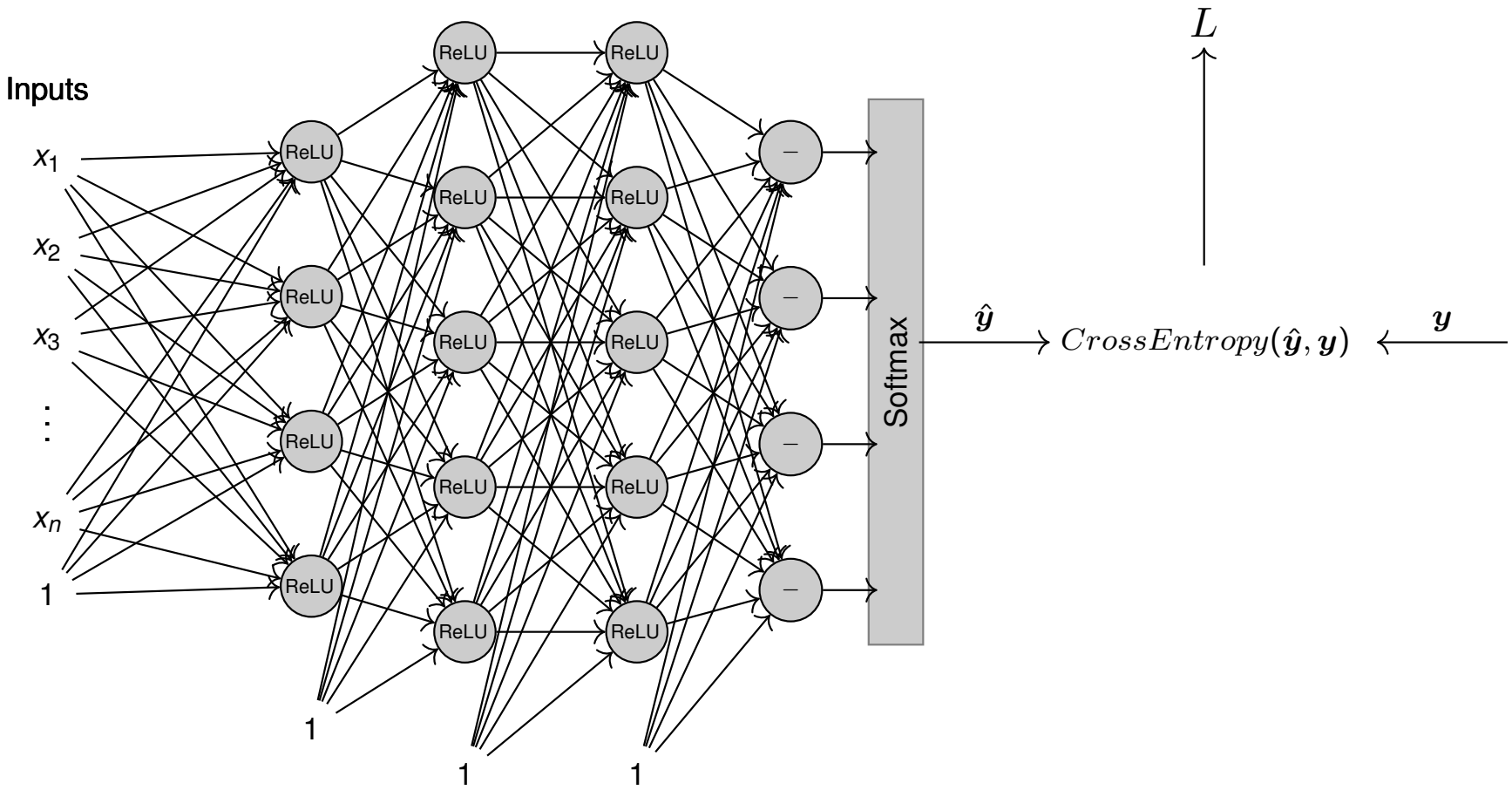
Forward actually defined elsewhere...

Backward

$$(1-\sigma(x))\,\sigma(x)$$

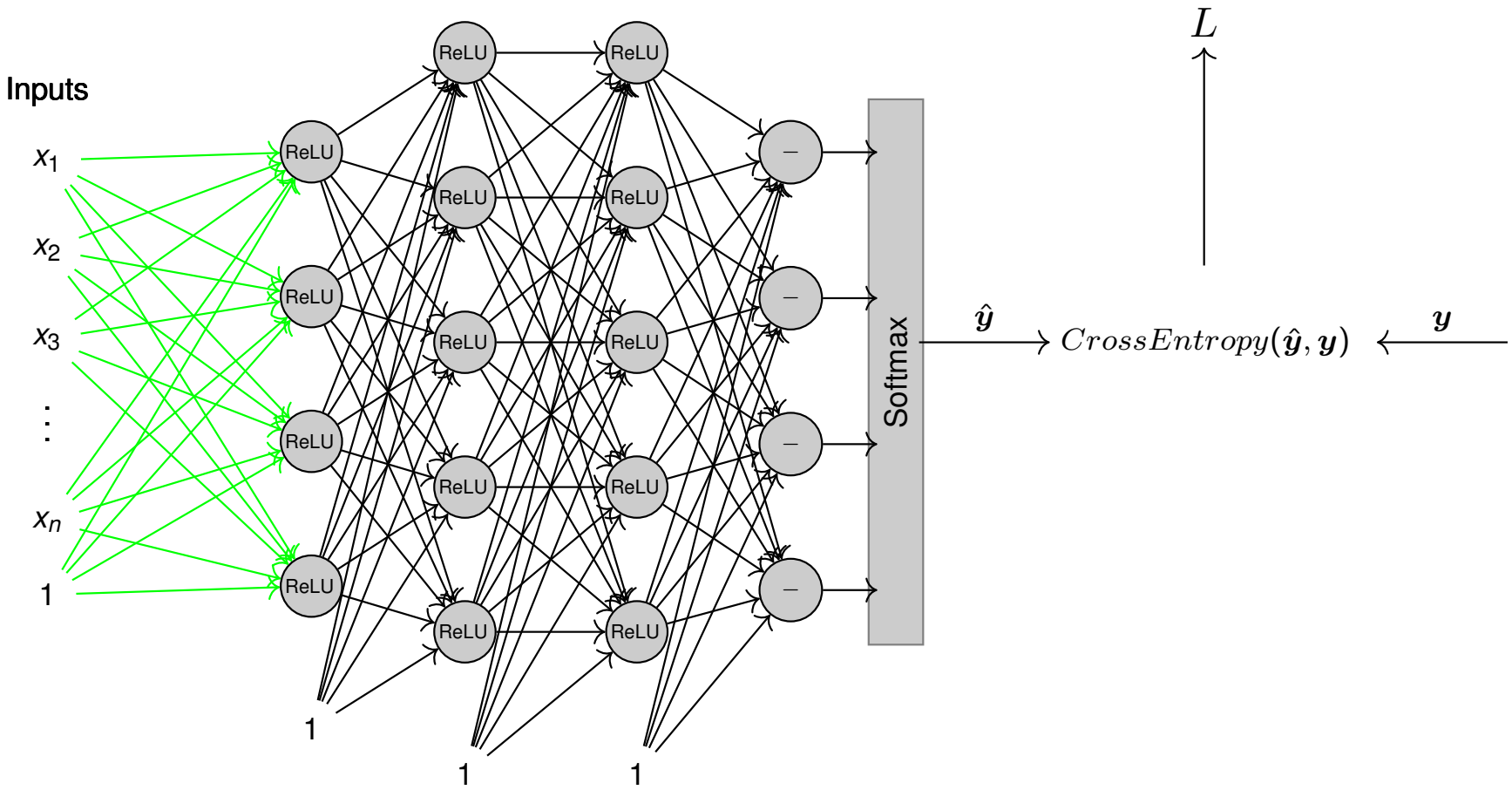
Source

# Fully connected neural network - Forward Pass



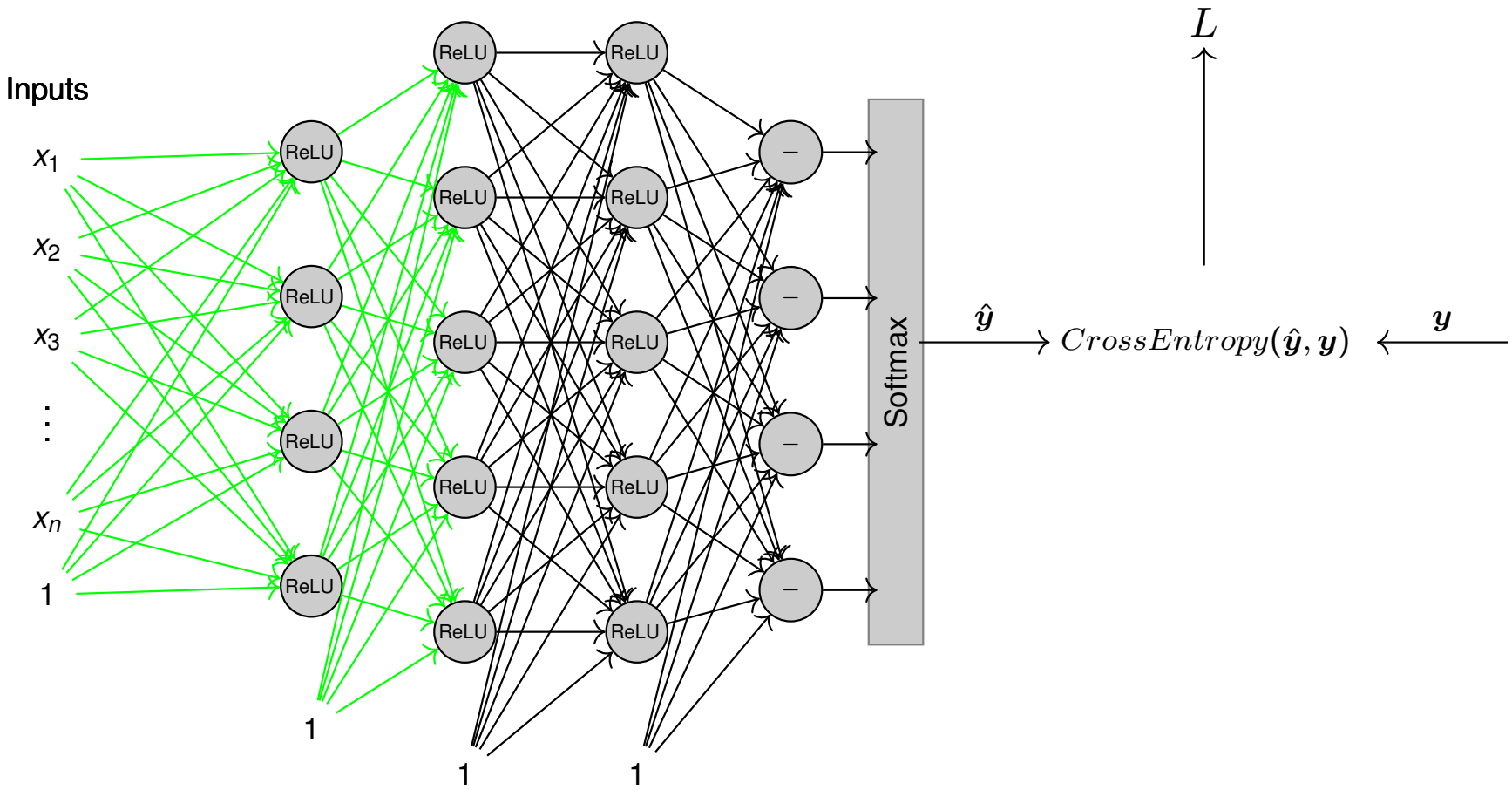

$$\hat{y} = Softmax(W^4(max(0, W^3(max(0, W^2(max(0, W^1 x + b^1)) + b^2)) + b^3)) + b^4)$$

# Fully connected neural network - Forward Pass



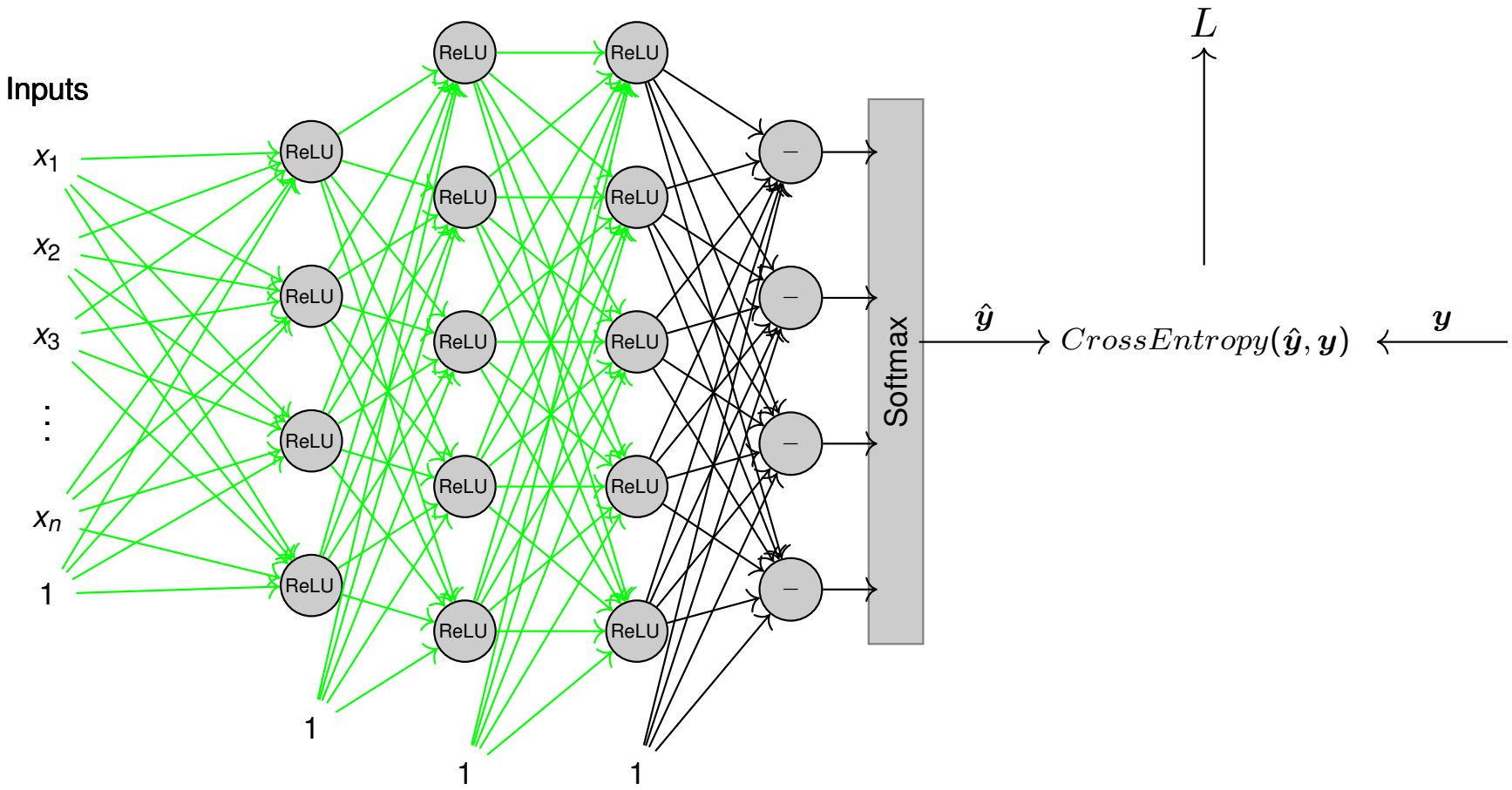

$$z^1 = W^1 x + b^1, \quad o^1 = ReLU(z^1)$$

# Fully connected neural network - Forward Pass



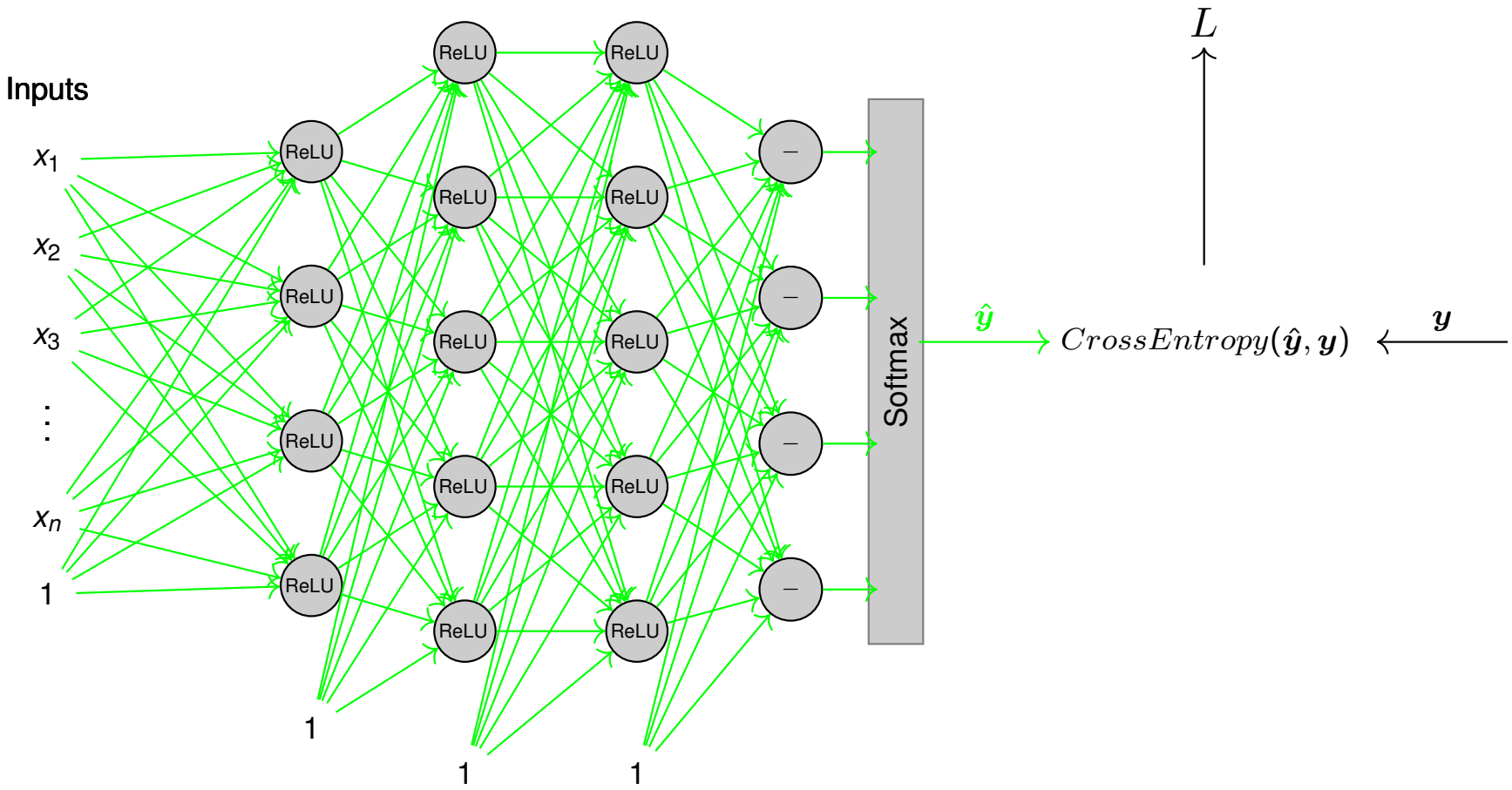

$$z^2 = W^2 o^1 + b^2, \quad o^2 = ReLU(z^2)$$

# Fully connected neural network - Forward Pass



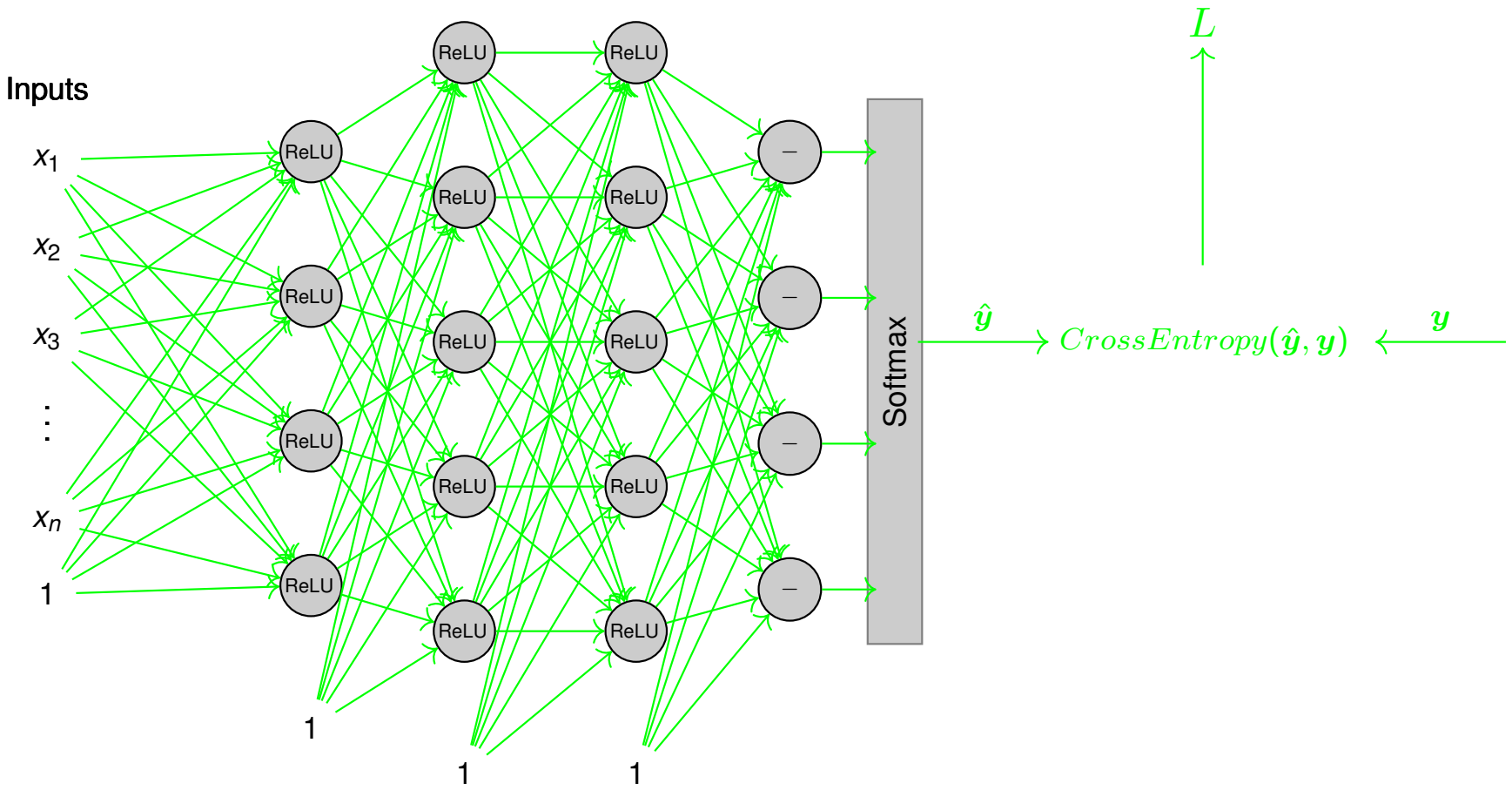

$$z^3 = W^3 o^2 + b^3, \quad o^3 = ReLU(z^3)$$

# Fully connected neural network - Forward Pass



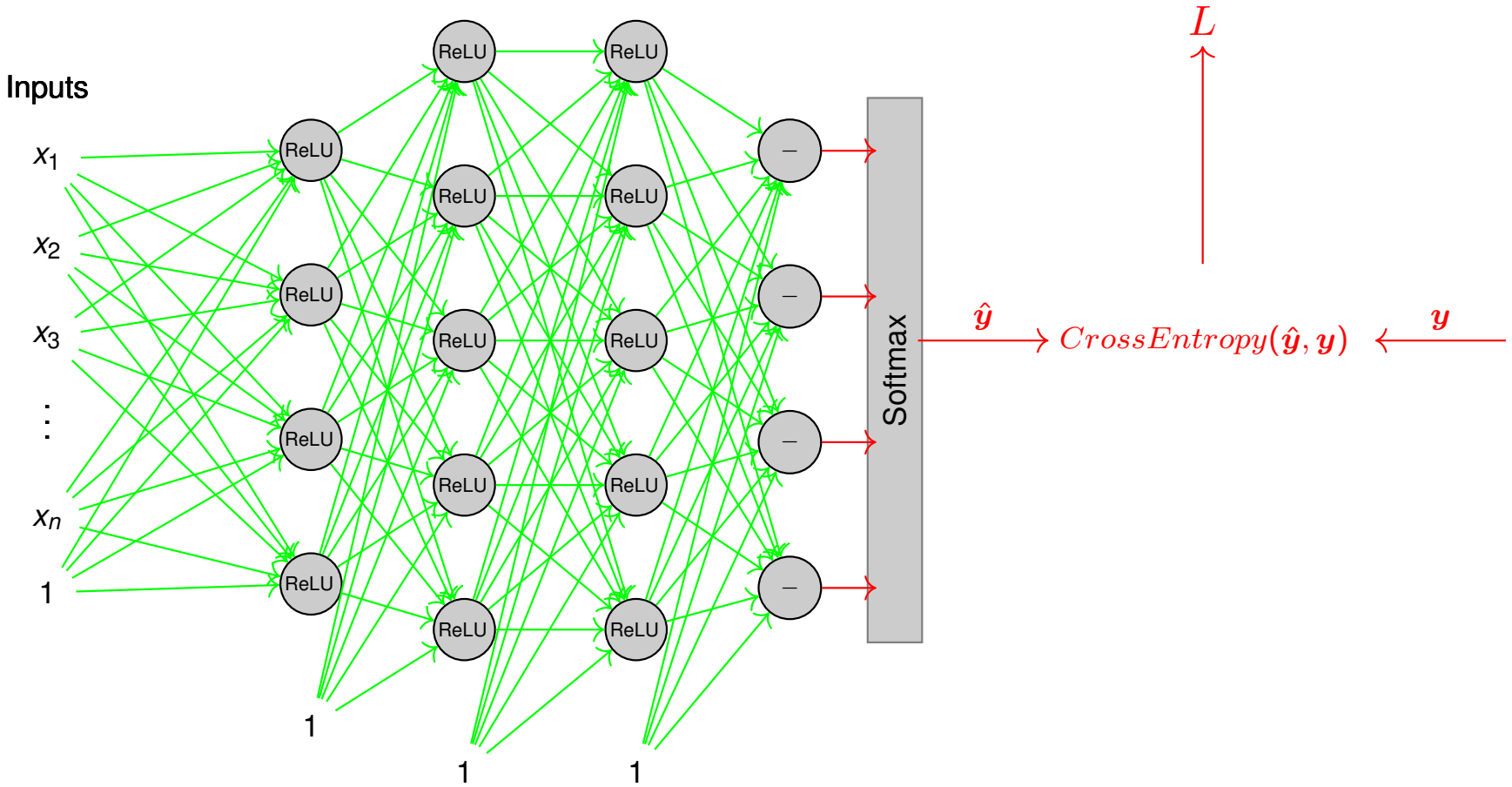

$$z^4 = W^4 o^3 + b^4, \quad \hat{y} = softmax(z^4)$$

# Fully connected neural network - Forward Pass



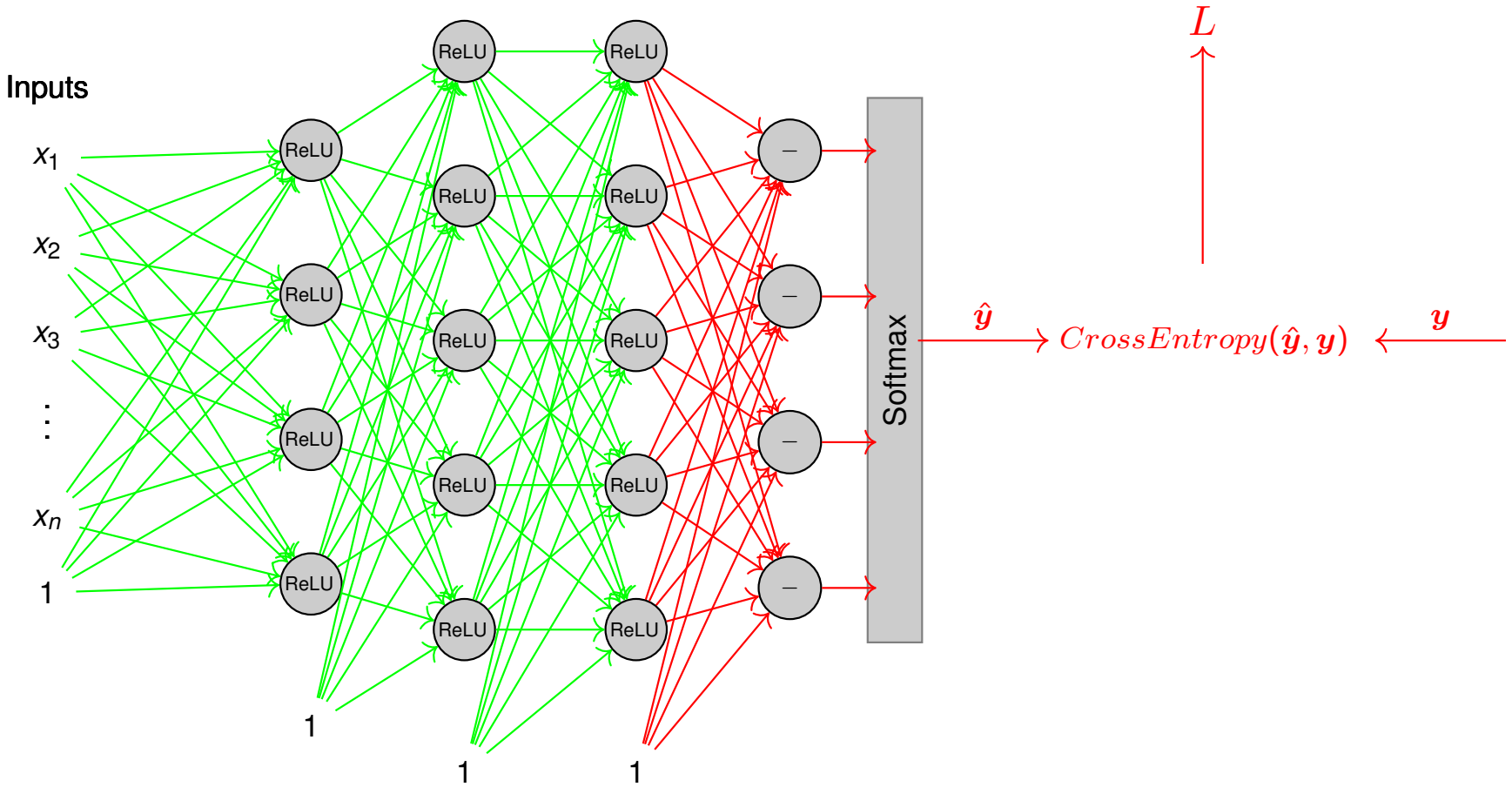

$$L = CrossEntropy(\hat{\boldsymbol{y}}, \boldsymbol{y})$$

# Fully connected neural network - Backward Pass



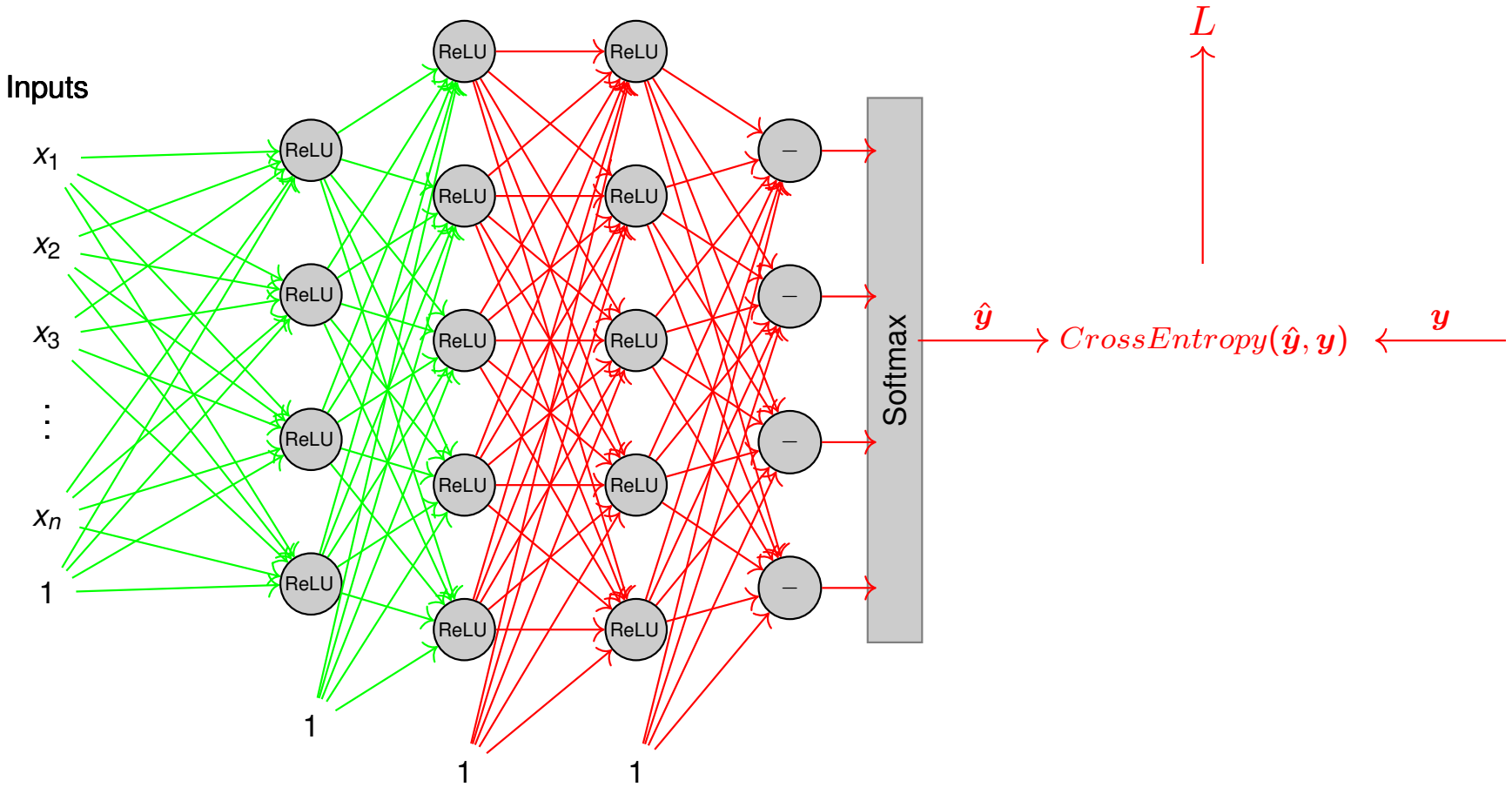

$$\frac{\partial L}{\partial z^4} = \hat{y} - y$$

# Fully connected neural network - Backward Pass



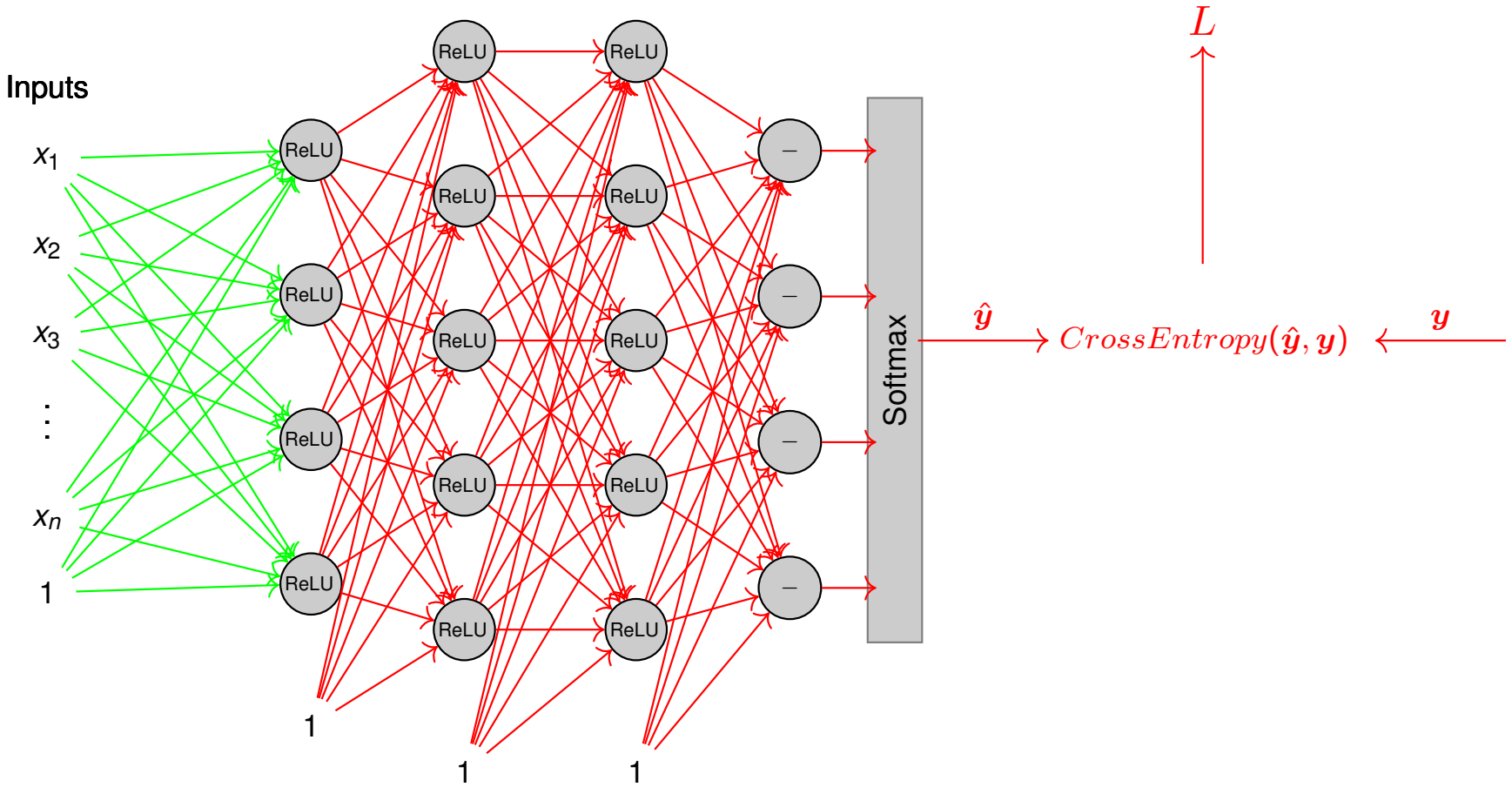

$$\frac{\partial L}{\partial W^4} = \frac{\partial L}{\partial z^4}\frac{\partial z^4}{\partial W^4} = \frac{\partial L}{\partial z^4}o^3, \quad \frac{\partial L}{\partial b^4} = \frac{\partial L}{\partial z^4}\frac{\partial z^4}{\partial b^4} = \frac{\partial L}{\partial z^4}, \quad \frac{\partial L}{\partial o^3} = \frac{\partial L}{\partial z^4}\frac{\partial z^4}{\partial o^3} = \frac{\partial L}{\partial z^4}W^4$$

# Fully connected neural network - Backward Pass



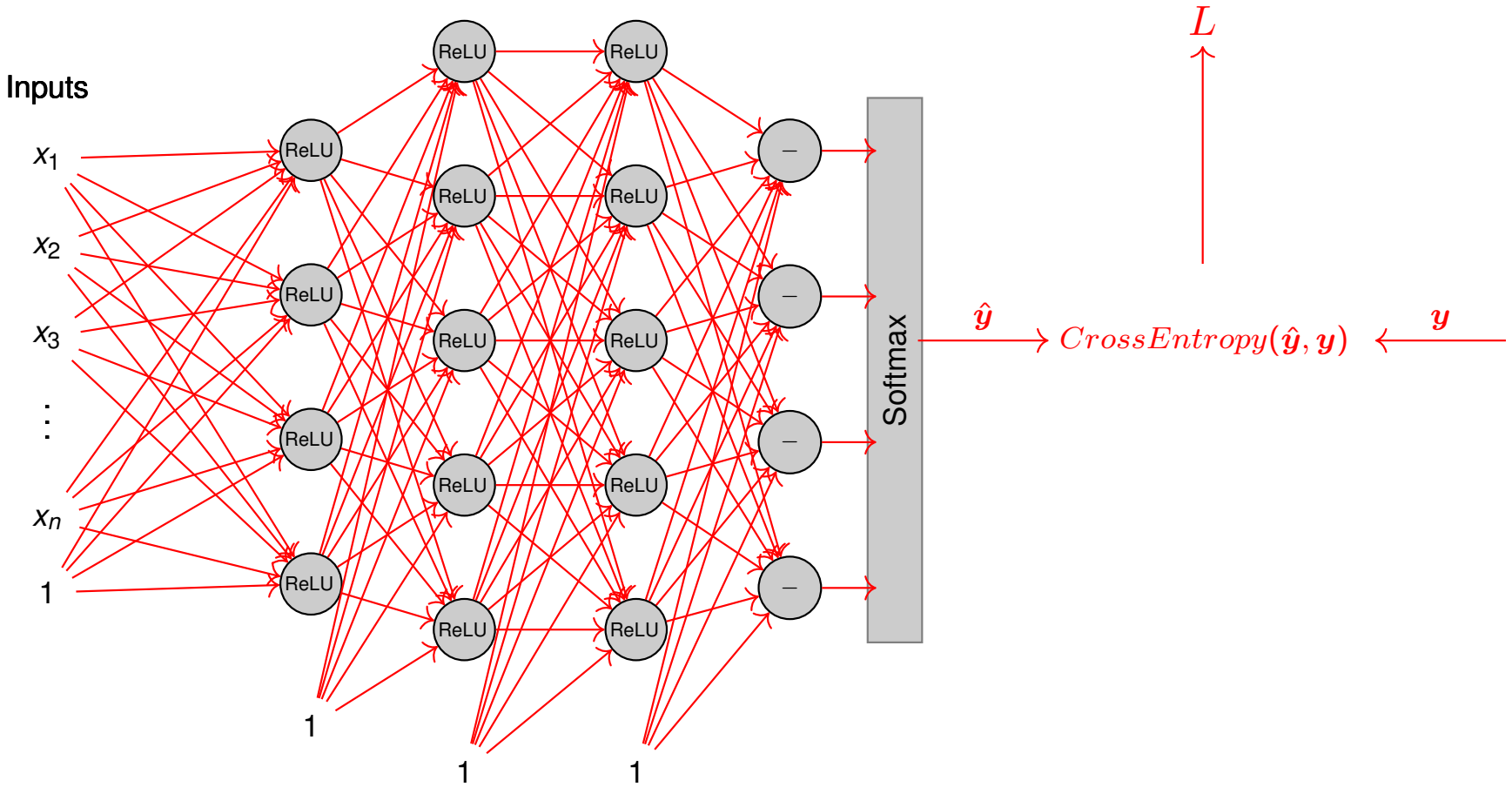

$$\frac{\partial L}{\partial z^3} = \frac{\partial L}{\partial o^3}\frac{\partial o^3}{\partial z^3} = \frac{\partial L}{\partial o^3} ReLU(sign(z^3)), \quad \frac{\partial L}{\partial W^3} = \frac{\partial L}{\partial z^3} o^2, \quad \frac{\partial L}{\partial b^3} = \frac{\partial L}{\partial z^3}, \quad \frac{\partial L}{\partial o^2} = \frac{\partial L}{\partial z^3} W^3$$

# Fully connected neural network - Backward Pass




$$\frac{\partial L}{\partial z^2} = \frac{\partial L}{\partial o^2} ReLU(sign(z^2)), \quad \frac{\partial L}{\partial W^2} = \frac{\partial L}{\partial z^2} o^1, \quad \frac{\partial L}{\partial b^2} = \frac{\partial L}{\partial z^2}, \quad \frac{\partial L}{\partial o^1} = \frac{\partial L}{\partial z^2} W^2$$

# Fully connected neural network - Backward Pass




$$\frac{\partial L}{\partial z^1} = \frac{\partial L}{\partial o^1} ReLU(sign(z^1)), \quad \frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial z^1} x, \quad \frac{\partial L}{\partial b^1} = \frac{\partial L}{\partial z^1}, \quad \frac{\partial L}{\partial x} = \frac{\partial L}{\partial z^1} W^1$$

# What about vector-valued functions

- **Scalar to Scalar**
  - Derivative

$$x \in \mathbb{R}, y \in \mathbb{R} \rightarrow \frac{dy}{dx} \in \mathbb{R}$$

  - If $x$ changes by a small amount, how much will $y$ change?

# What about vector-valued functions

- **Scalar to Scalar**
  - Derivative

$$x \in \mathbb{R}, y \in \mathbb{R} \rightarrow \frac{dy}{dx} \in \mathbb{R}$$

  - If $x$ changes by a small amount, how much will $y$ change?
- **Vector to Scalar**
  - Derivative is Gradient

$$\boldsymbol{x} \in \mathbb{R}^N, y \in \mathbb{R} \rightarrow \frac{\partial y}{\partial \boldsymbol{x}} \in \mathbb{R}^N, \left(\frac{\partial y}{\partial \boldsymbol{x}}\right)_n = \frac{\partial y}{\partial x_n}$$

  - For each element of $\boldsymbol{x}$, if it changes by a small amount then how much will $y$ change?
  - A vector of the same size of $\boldsymbol{x}$

# What about vector-valued functions

- **Scalar to Scalar**
  - Derivative

$$x \in \mathbb{R}, y \in \mathbb{R} \rightarrow \frac{dy}{dx} \in \mathbb{R}$$

  - If $x$ changes by a small amount, how much will $y$ change?
- **Vector to Scalar**
  - Derivative is Gradient

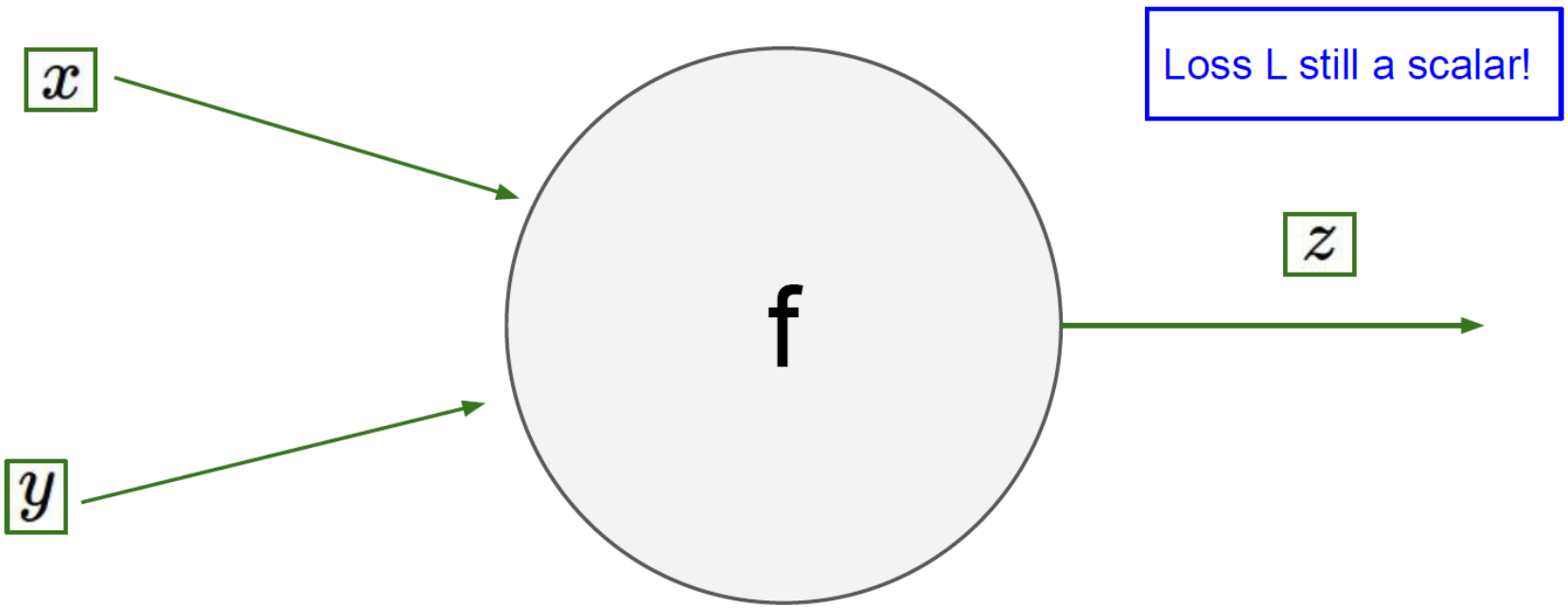
$$\boldsymbol{x} \in \mathbb{R}^N, y \in \mathbb{R} \rightarrow \frac{\partial y}{\partial \boldsymbol{x}} \in \mathbb{R}^N, \left(\frac{\partial y}{\partial \boldsymbol{x}}\right)_n = \frac{\partial y}{\partial x_n}$$

  - For each element of $\boldsymbol{x}$, if it changes by a small amount then how much will $y$ change?
  - A vector of the same size of $\boldsymbol{x}$
- **Vector to Vector**
  - Derivative is Jacobian

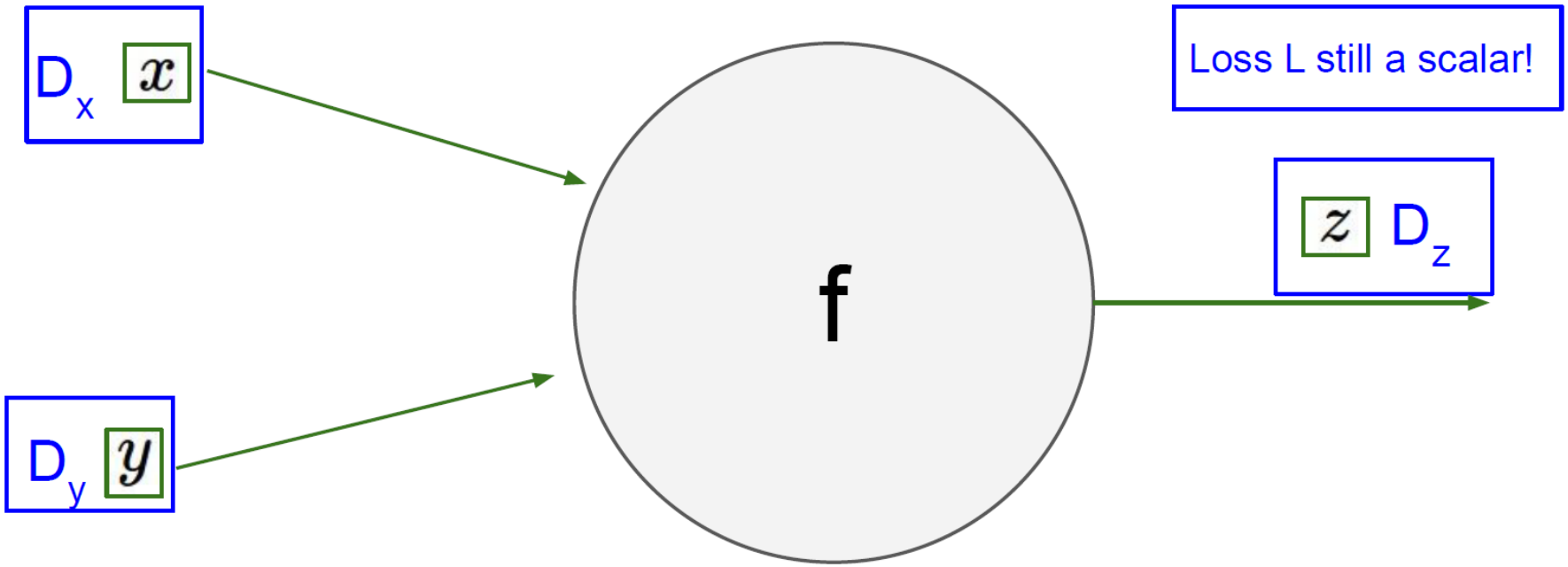
$$\boldsymbol{x} \in \mathbb{R}^N, \boldsymbol{y} \in \mathbb{R}^M \rightarrow \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} \in \mathbb{R}^{N \times M}, \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)_{n,m} = \frac{\partial y_m}{\partial x_n}$$

  - For each element of $\boldsymbol{x}$, if it changes by a small amount then how much will each element of $\boldsymbol{y}$ change?

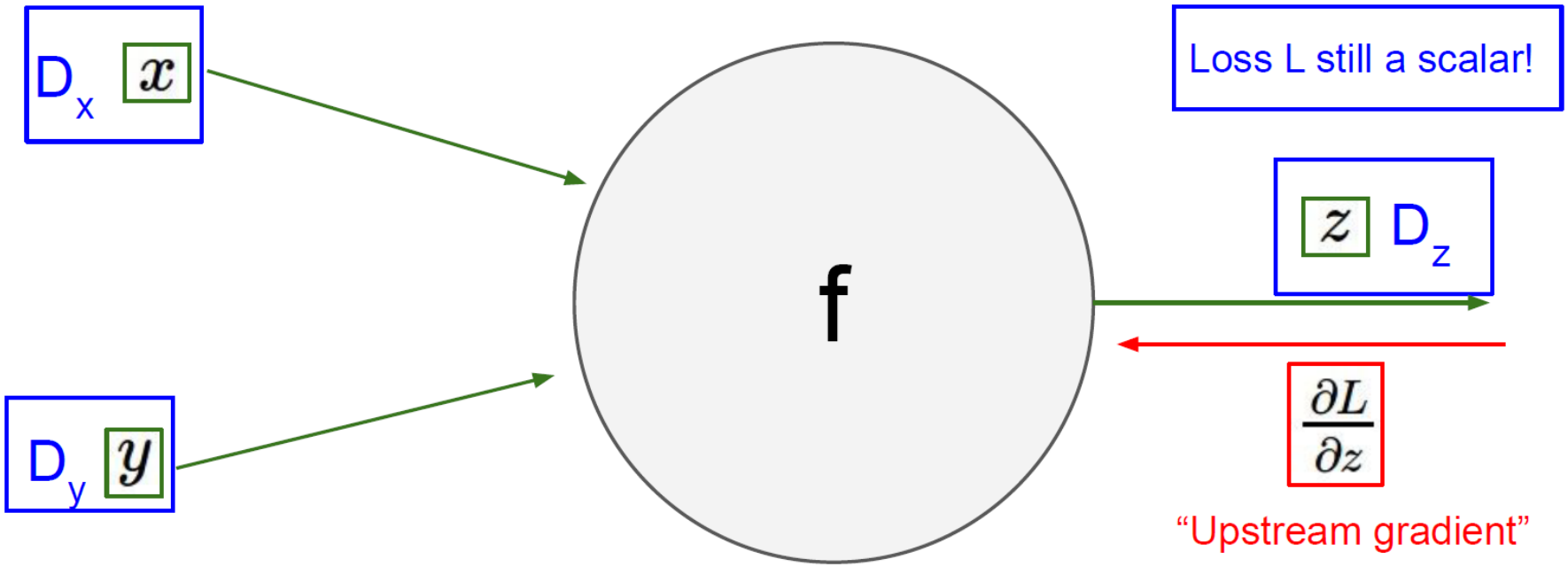
# Backprop with Vectors



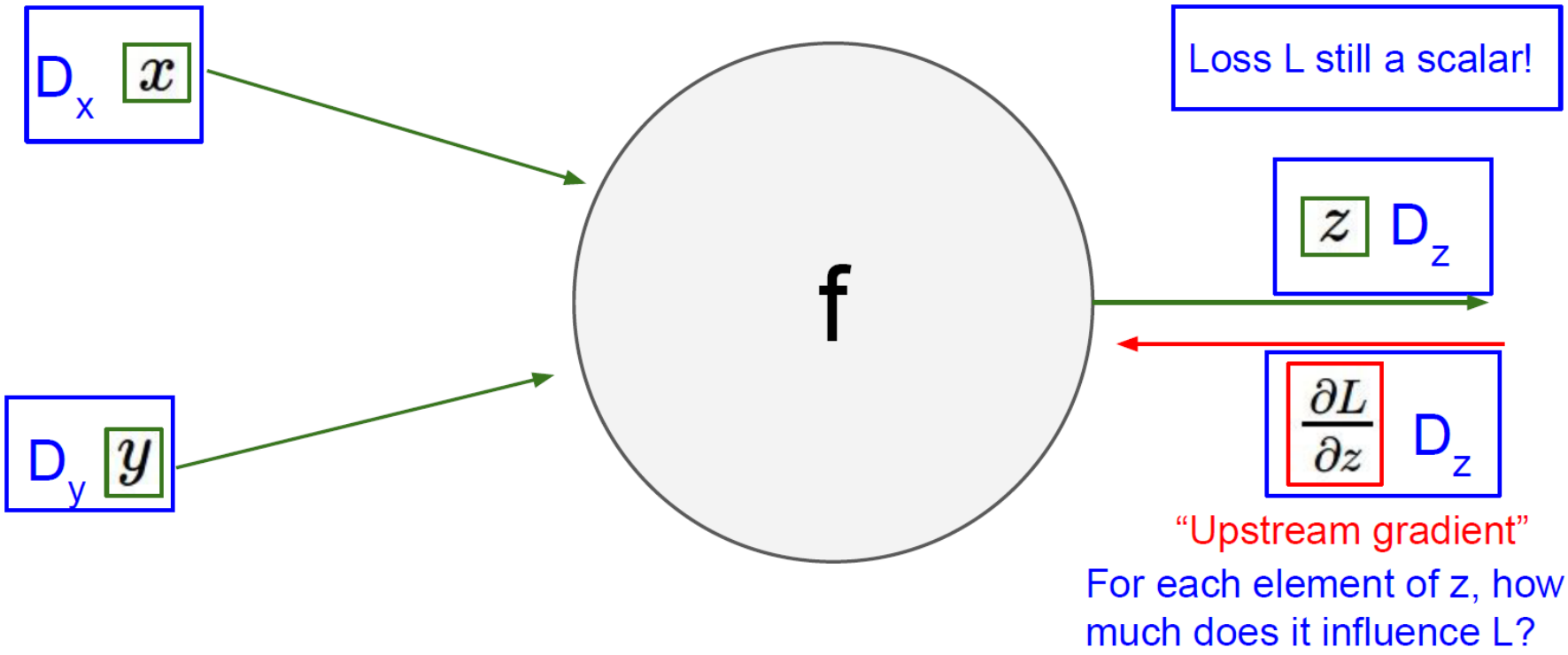
# Backprop with Vectors



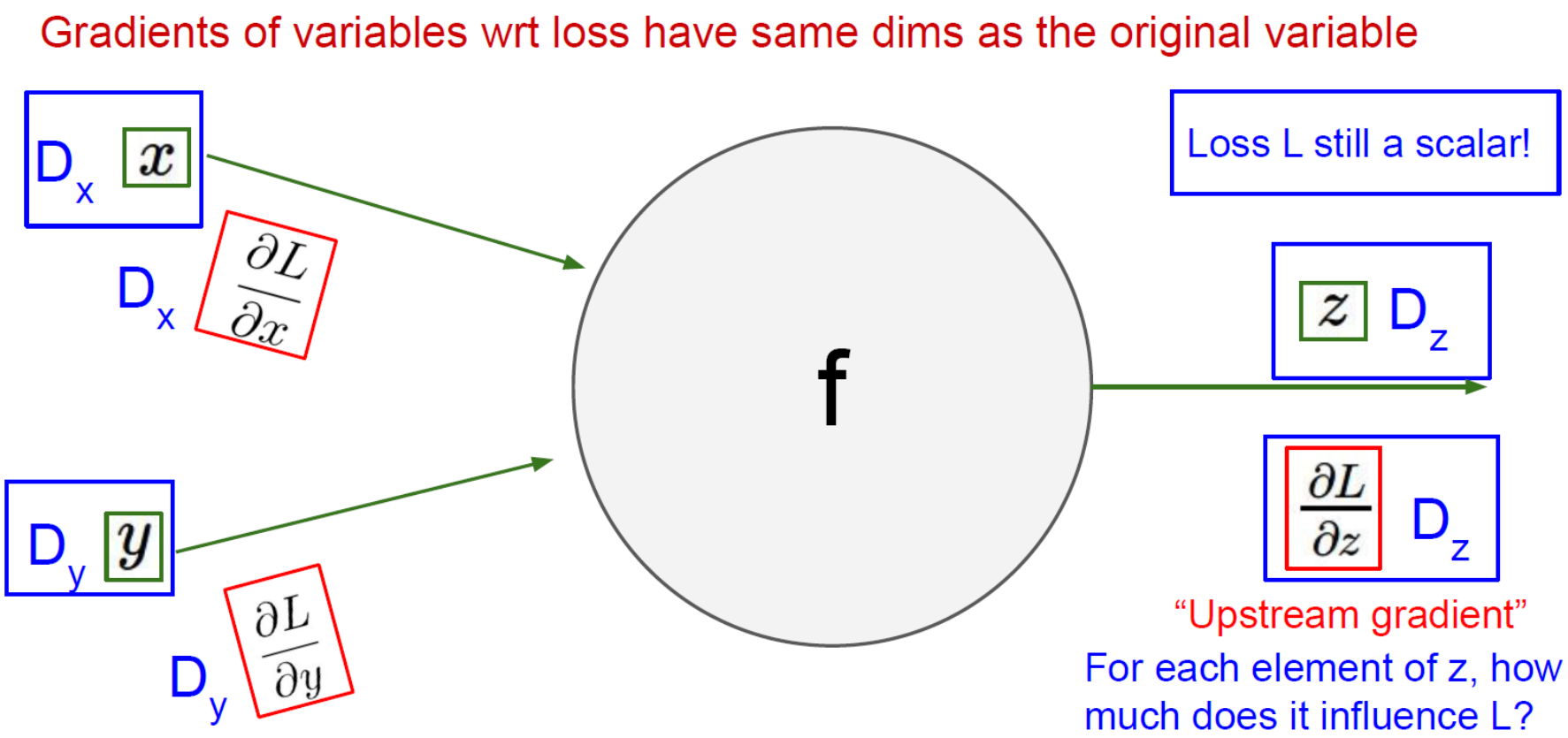
# Backprop with Vectors



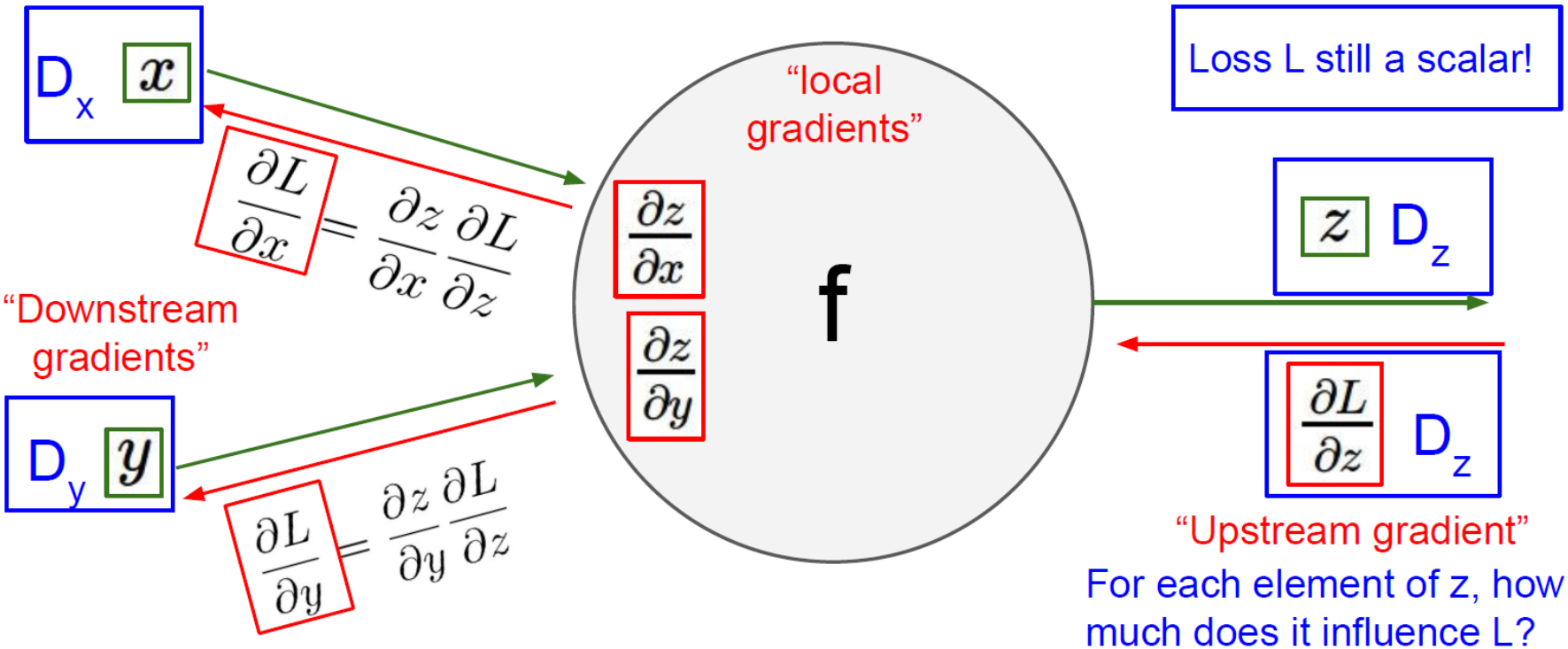
# Backprop with Vectors



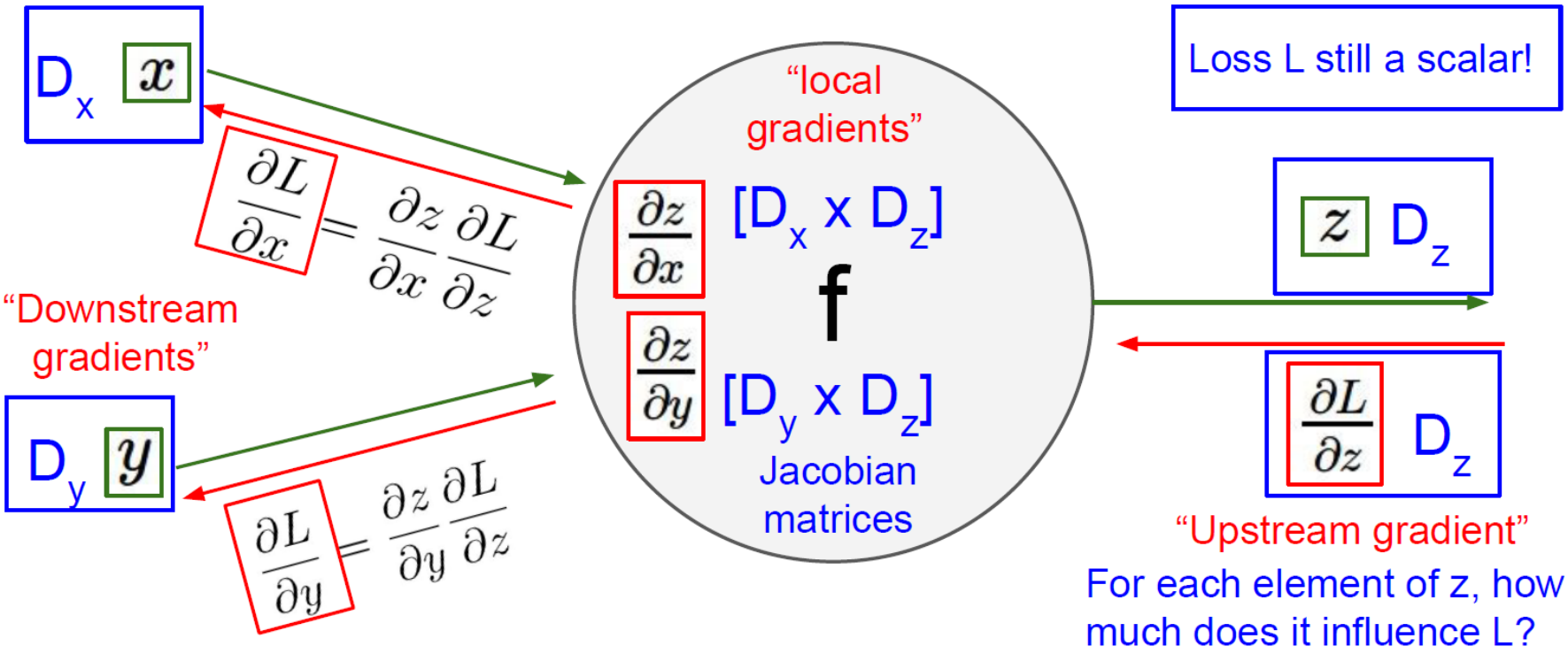
# Backprop with Vectors



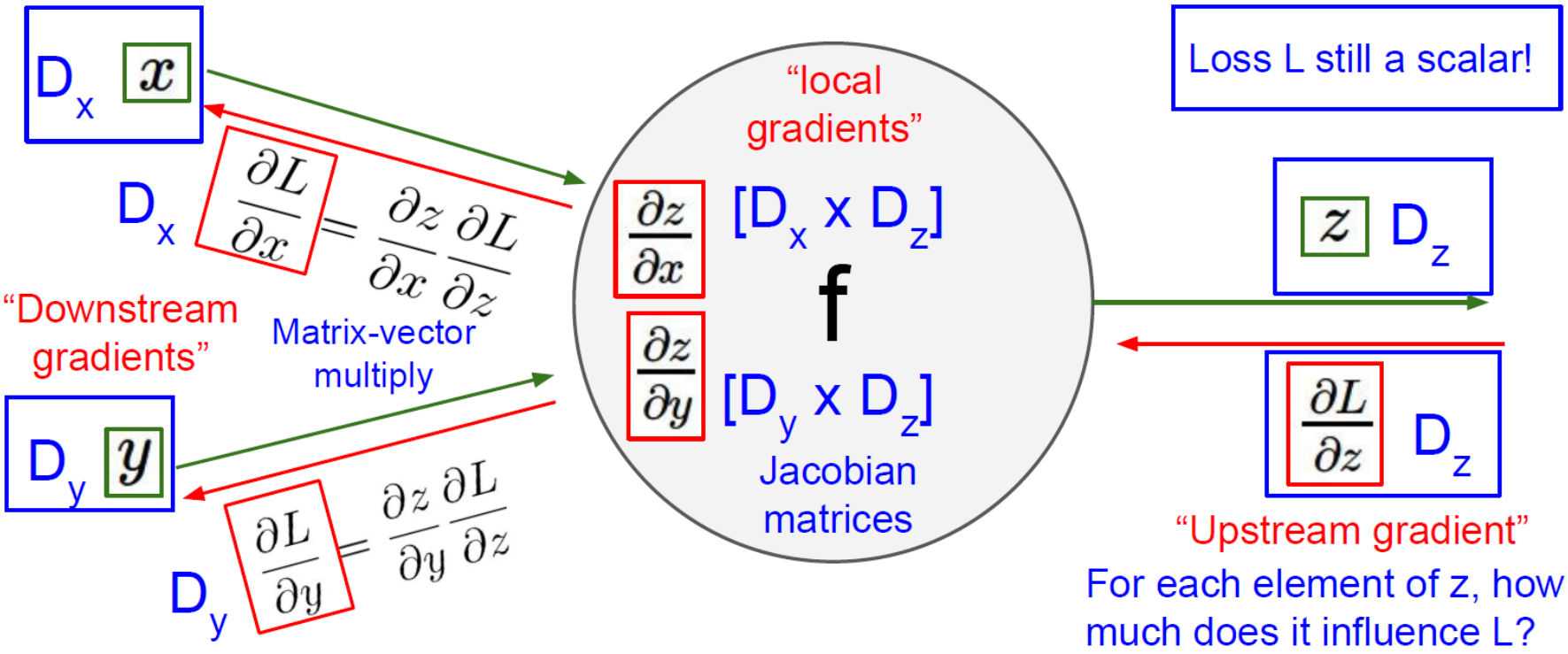
# Backprop with Vectors



# Backprop with Vectors



# Backprop with Vectors



# Epoch

- Epoch
  - An epoch means training the neural network with all the training data for one cycle.
  - In an epoch, we use all of the data exactly once.
  - A forward pass and a backward pass together are counted as one pass: An epoch is made up of one or more batches, where we use a part of the dataset to train the neural network.

# Gradient Descent

- Stochastic Gradient Descent
  - Stochastic gradient descent is a variation of the gradient descent algorithm that calculates the error and updates the model for *each example* in the training dataset.
    - The frequent updates immediately give an insight into the performance of the model and the rate of improvement.
    - Avoid local minima
    - The frequent updates can result in a noisy gradient signal, which may cause the model parameters and in turn the model error to jump around.
    - Updating the model so frequently is more computationally expensive than other configurations of gradient descent, taking significantly longer to train models on large datasets.

# Gradient Descent

- Stochastic Gradient Descent
  - Stochastic gradient descent is a variation of the gradient descent algorithm that calculates the error and updates the model for *each example* in the training dataset.
    - The frequent updates immediately give an insight into the performance of the model and the rate of improvement.
    - Avoid local minima
    - The frequent updates can result in a noisy gradient signal, which may cause the model parameters and in turn the model error to jump around.
    - Updating the model so frequently is more computationally expensive than other configurations of gradient descent, taking significantly longer to train models on large datasets.
- Batch (Full-batch) Gradient Descent
  - Full-batch gradient descent is a variation of the gradient descent algorithm that calculates the error for each example in the training dataset, but only updates the model after all training examples have been evaluated.
    - The decreased update frequency results in a more stable error gradient and may result in a more stable convergence on some problems.
    - Parallel processing based implementations.
    - It requires the entire training dataset in memory and available to the algorithm.

# Gradient Descent

- Stochastic Gradient Descent
  - Stochastic gradient descent is a variation of the gradient descent algorithm that calculates the error and updates the model for *each example* in the training dataset.
    - The frequent updates immediately give an insight into the performance of the model and the rate of improvement.
    - Avoid local minima
    - The frequent updates can result in a noisy gradient signal, which may cause the model parameters and in turn the model error to jump around.
    - Updating the model so frequently is more computationally expensive than other configurations of gradient descent, taking significantly longer to train models on large datasets.
- Batch (Full-batch) Gradient Descent
  - Full-batch gradient descent is a variation of the gradient descent algorithm that calculates the error for each example in the training dataset, but only updates the model after all training examples have been evaluated.
    - The decreased update frequency results in a more stable error gradient and may result in a more stable convergence on some problems.
    - Parallel processing based implementations.
    - It requires the entire training dataset in memory and available to the algorithm.
- Mini-Batch Gradient Descent
  - Mini-batch gradient descent is a variation of the gradient descent algorithm that splits the training dataset into small batches that are used to calculate model error and update model weights.
    - Mini-batch gradient descent seeks to find a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent. It is the most common implementation of gradient descent used in the field of deep learning.
    - 32 / 64 / 128 are common sizes for a mini-batch
    - It is a new hyper-parameter!

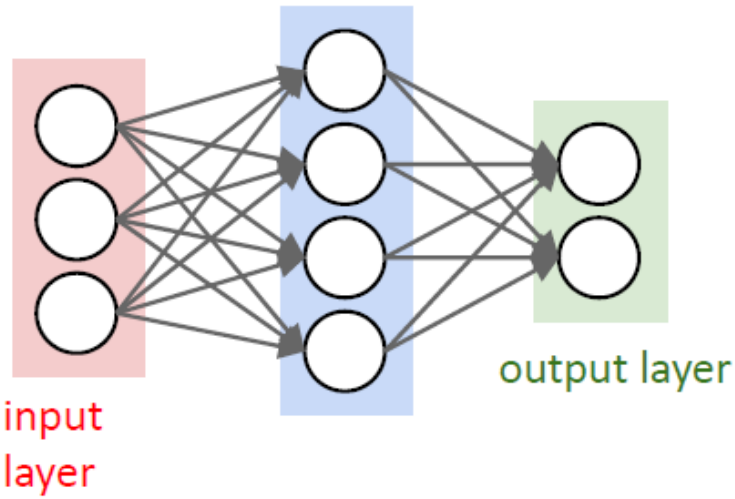
Source

# Convolutional Neural Networks

# The model so far

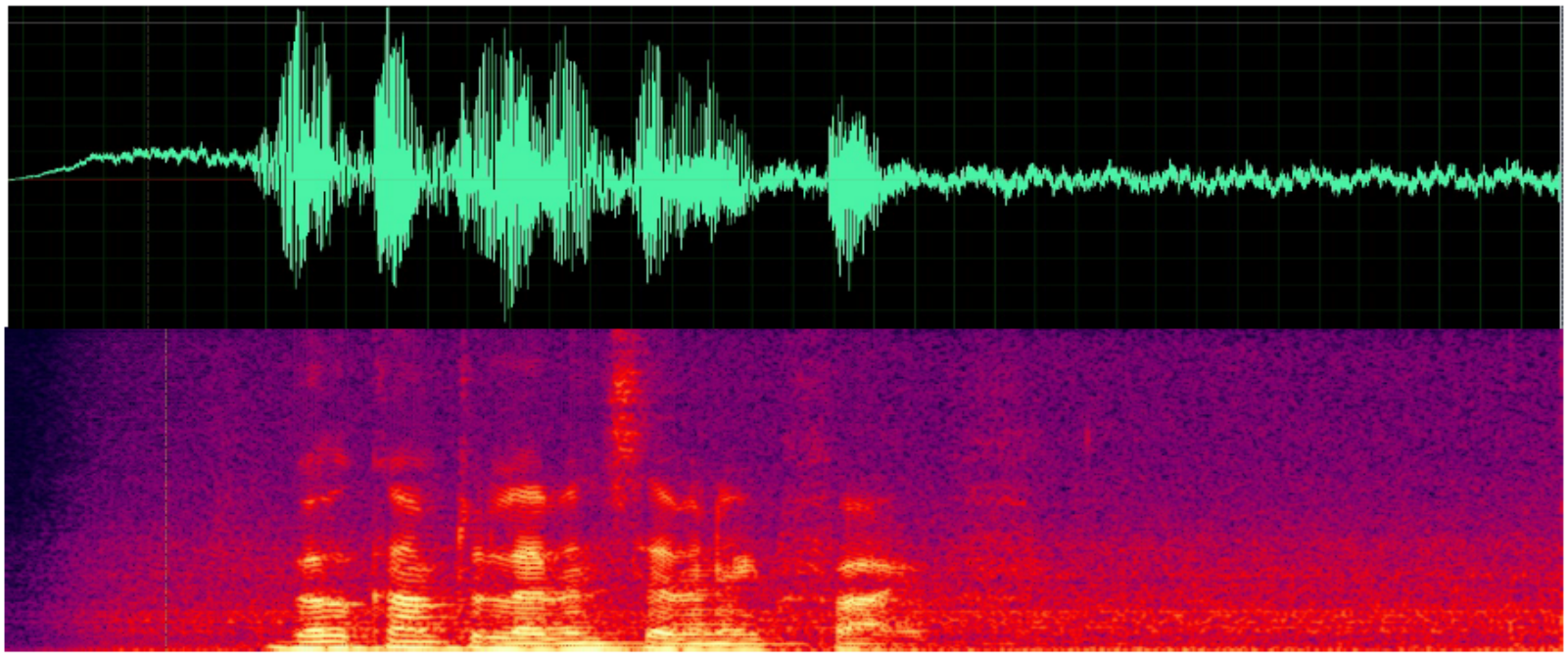
- Can recognize patterns in data
  - E.g. digits
  - Or any other vector data

Or, more generally
a vector input



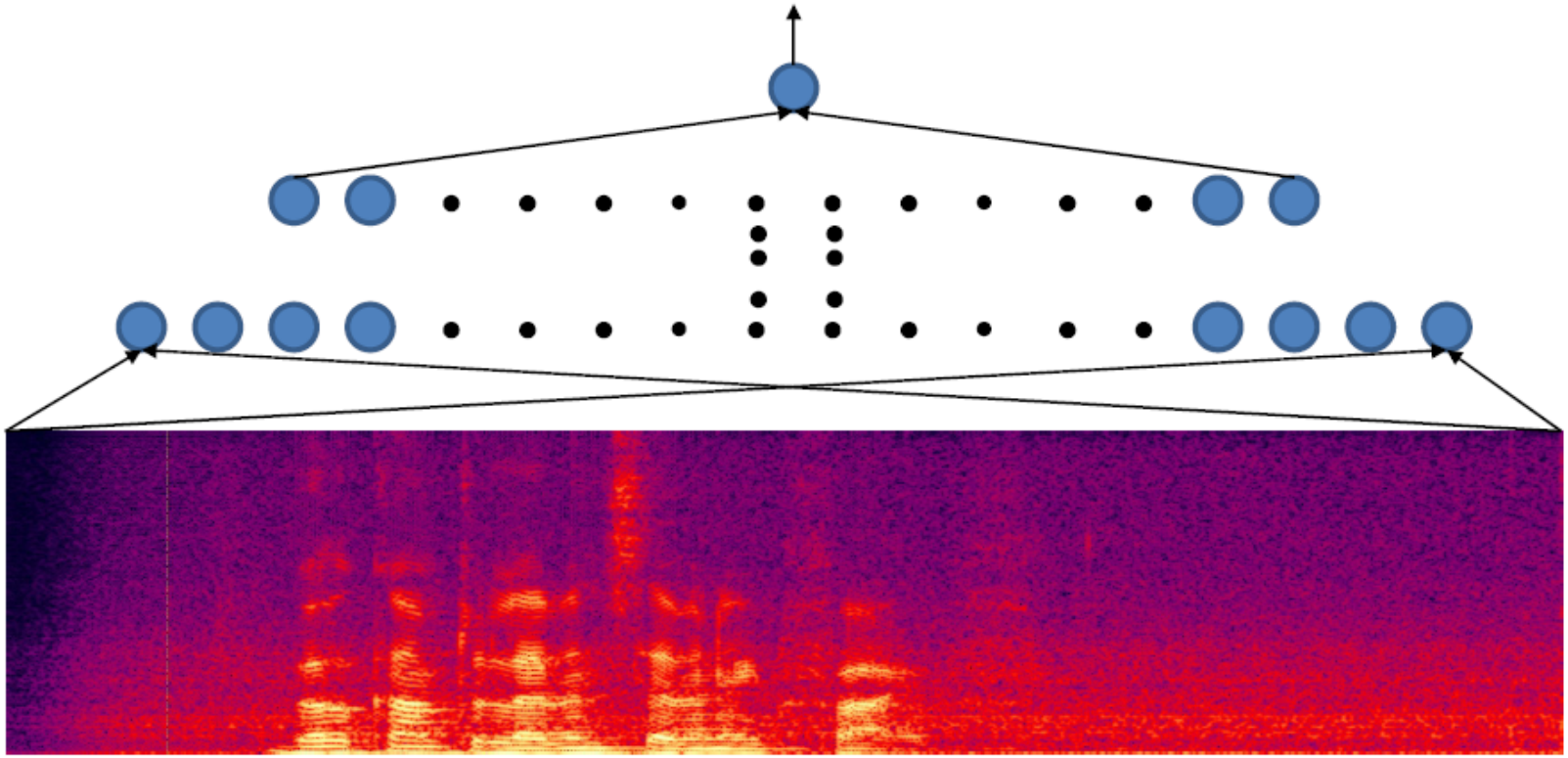
# A new problem

- Does this signal contain the word Welcome?
- Compose an MLP for this problem.
  - Assuming all recordings are exactly the same length.



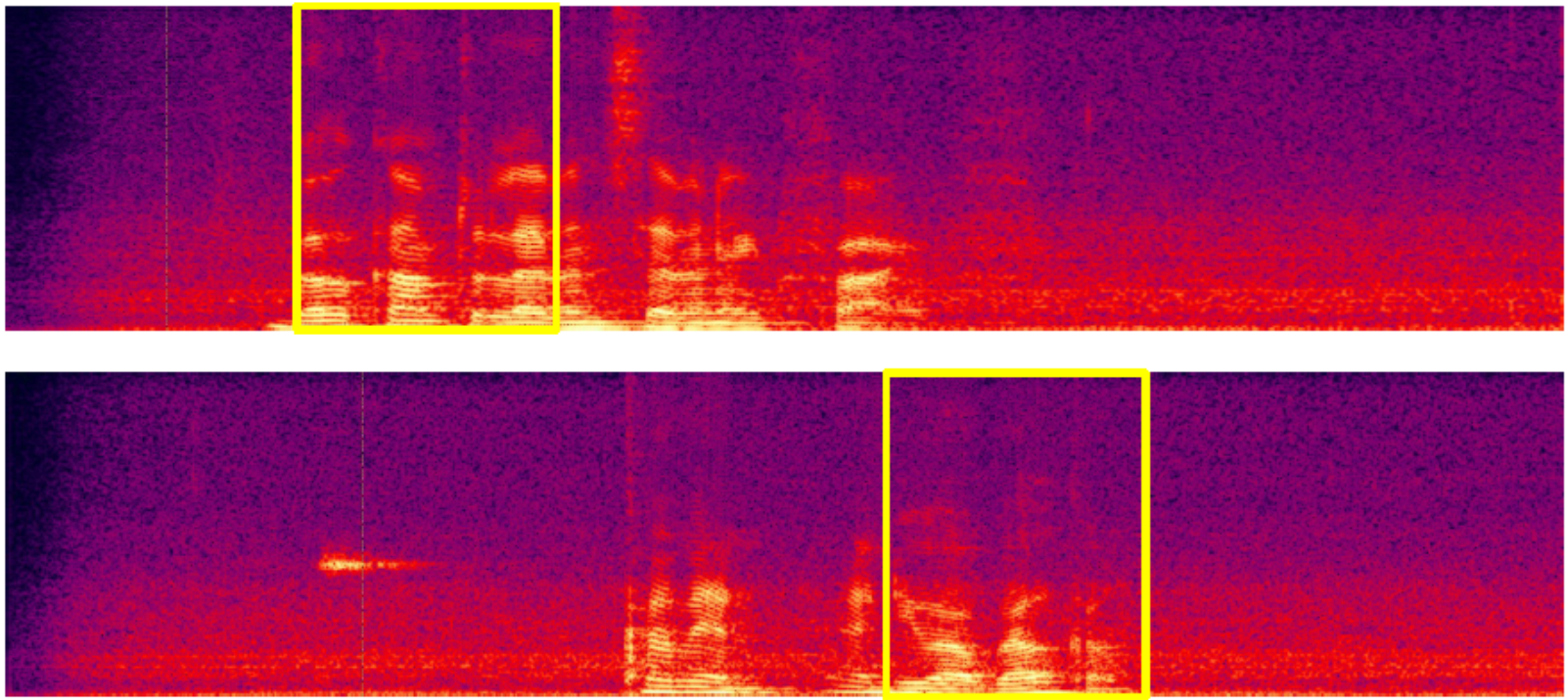
# Finding a Welcome

- Trivial solution: Train an MLP for the entire recording



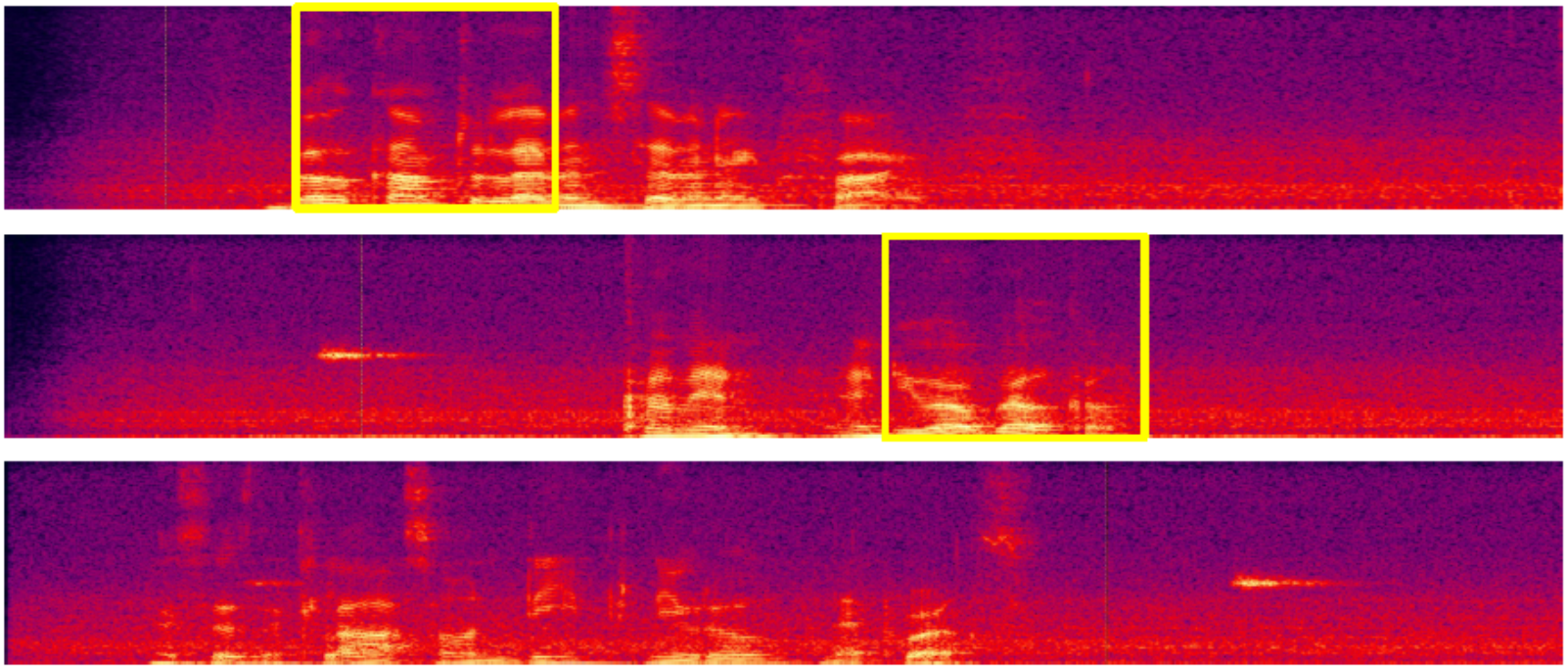
# Finding a Welcome

- Problem with trivial solution: Network that finds a welcome in the top recording will not find it in the lower one
  - Unless trained with both
  - Will require a very large network and a large amount of training data to cover every case



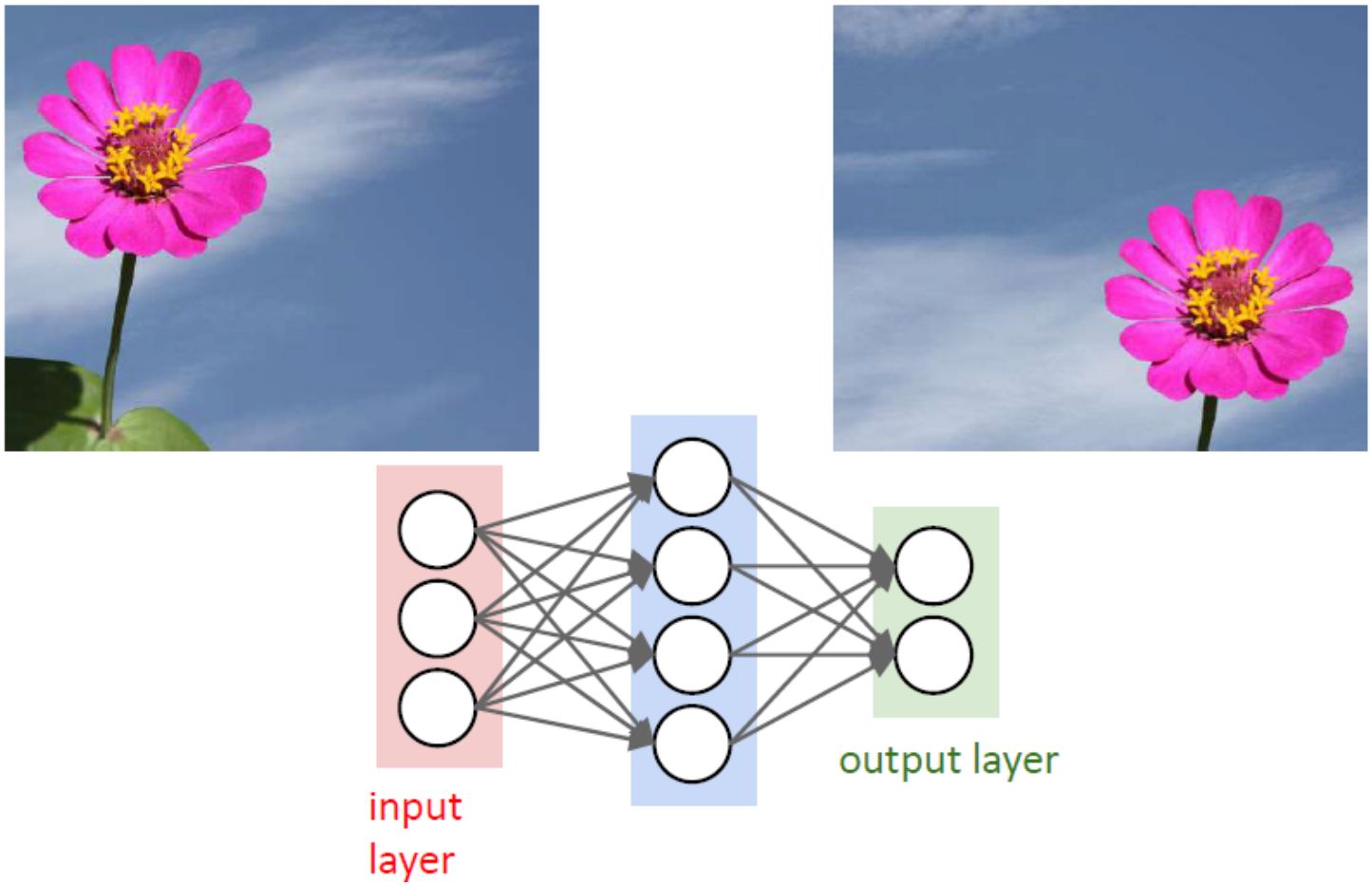
# Finding a Welcome

- Need a simple network that will fire regardless of the location of Welcome
  - and not fire when there is none



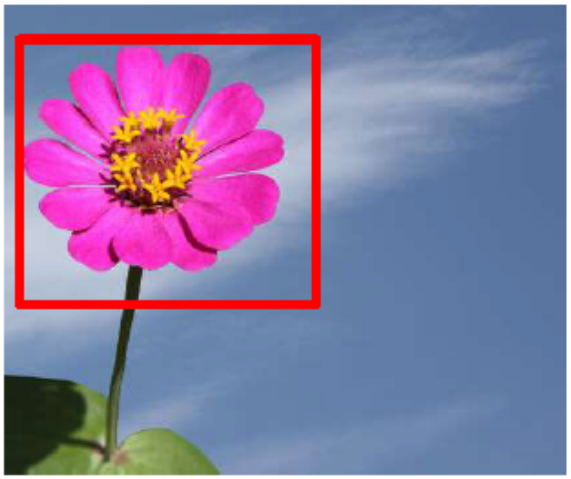
# A problem

- Will an MLP that recognizes the left image as a flower also recognize the one on the right as a flower?



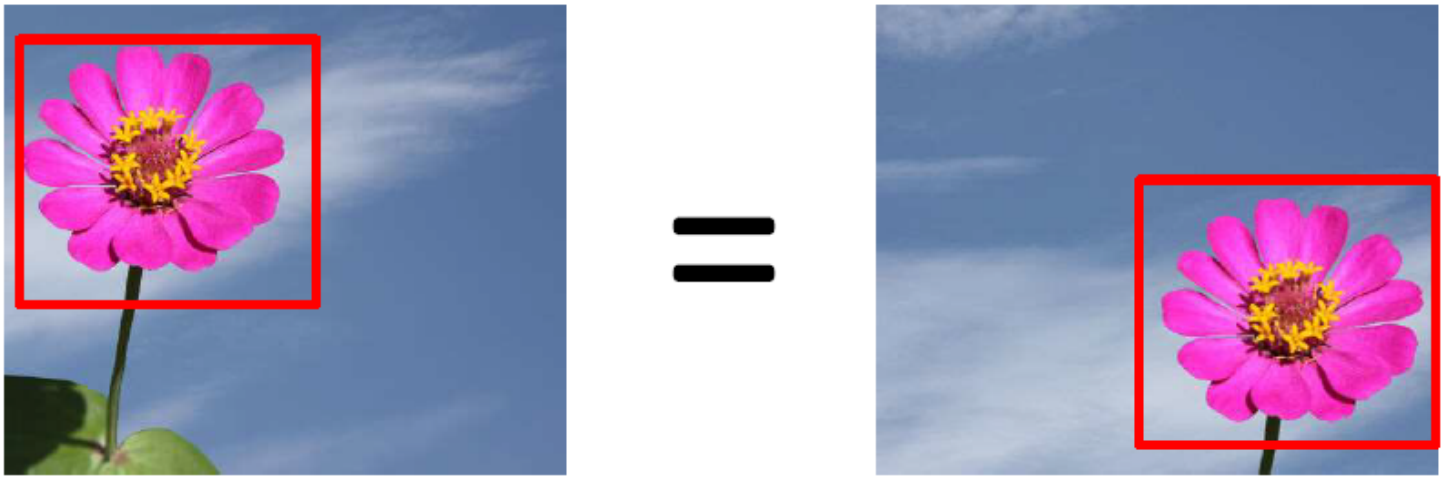
# A problem

- Need a network that will fire regardless of the precise location of the target object



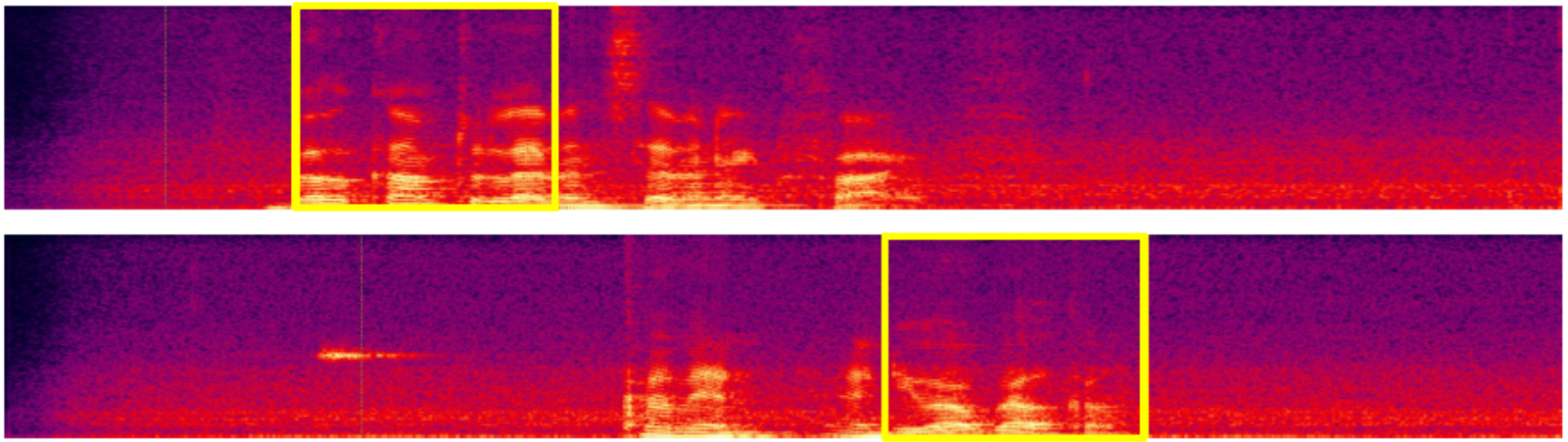
# A problem

- In many problems the location of a pattern is not important
  - Only the presence of the pattern
- Conventional MLPs are sensitive to the location of the pattern
  - Moving it by one component results in an entirely different input that the MLP wont recognize
- Requirement: Network must be **shift invariant**



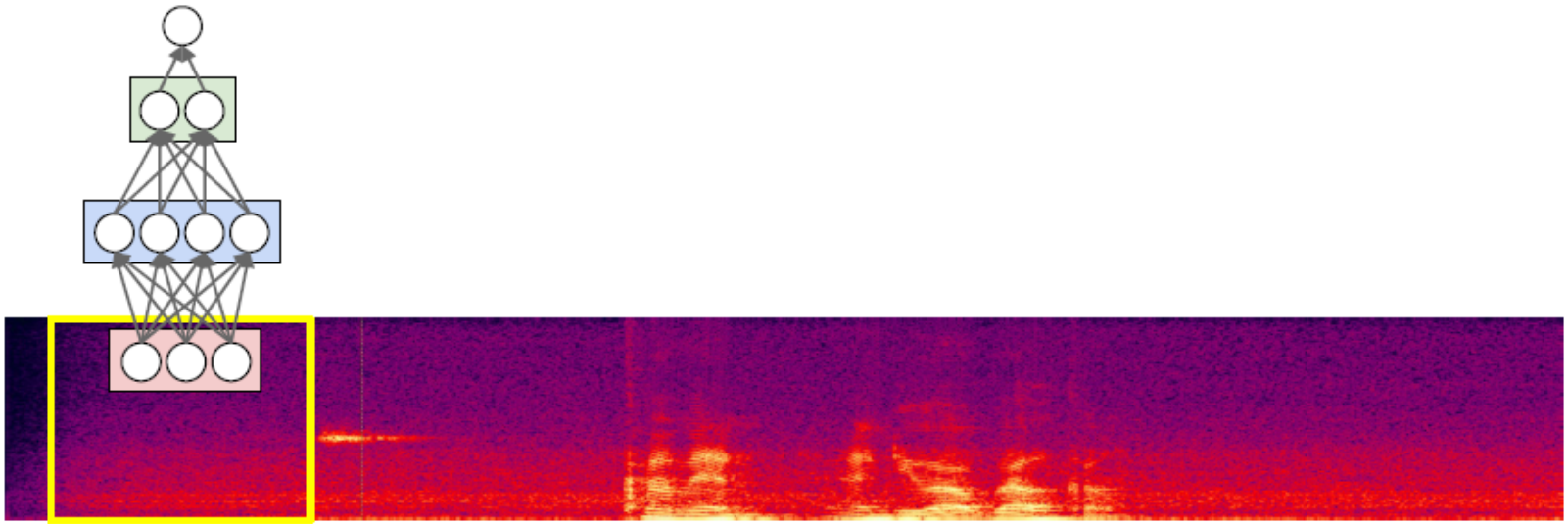
# A problem

- In many problems the location of a pattern is not important
  - Only the presence of the pattern
- Conventional MLPs are sensitive to the location of the pattern
  - Moving it by one component results in an entirely different input that the MLP wont recognize
- Requirement: Network must be **shift invariant**



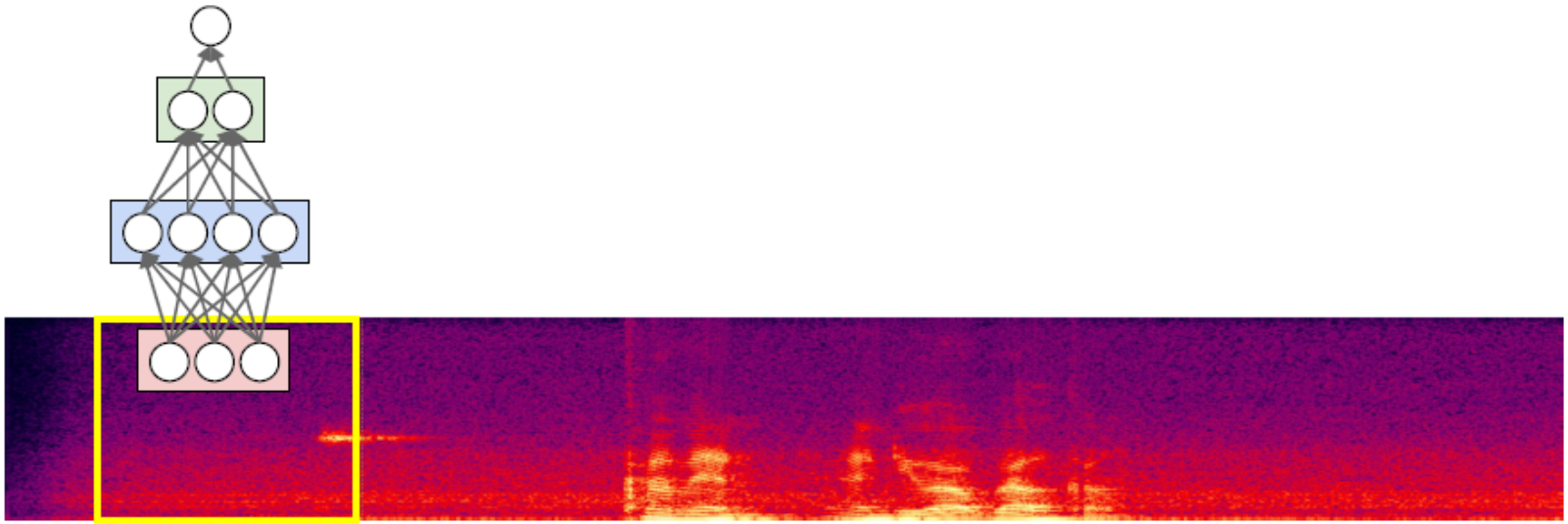
# Solution: Scan

- Scan for the target word
  - The spectral time-frequency components in a window are input to a welcome-detector MLP



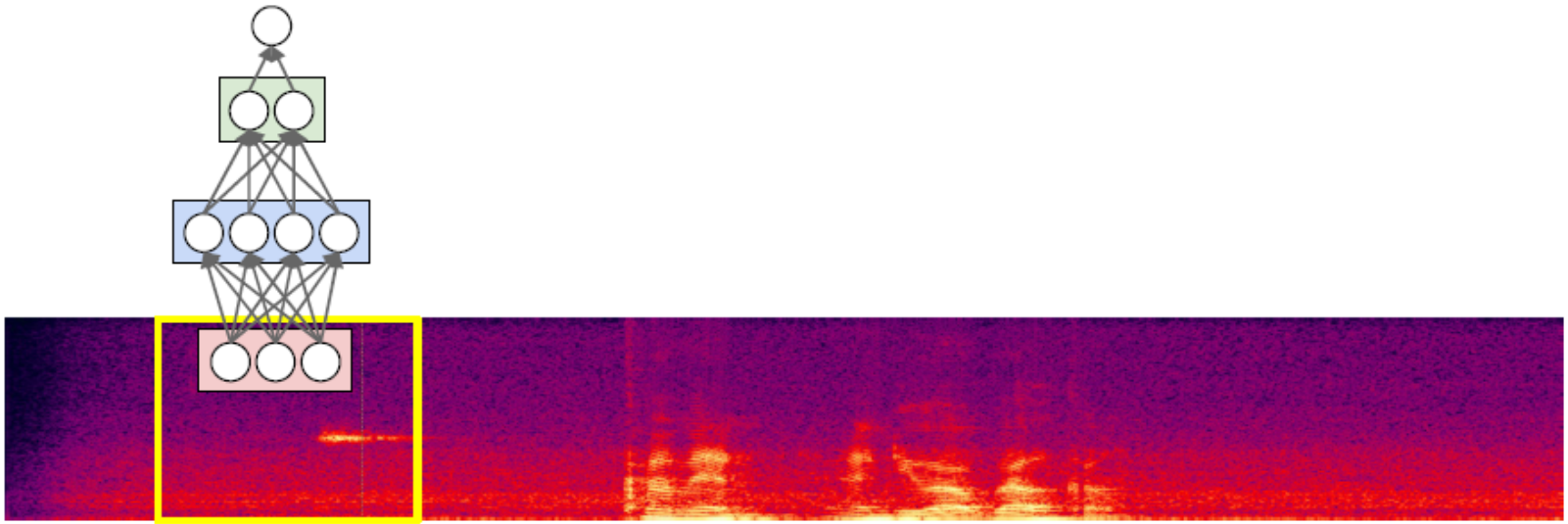
# Solution: Scan

- Scan for the target word
  - The spectral time-frequency components in a window are input to a welcome-detector MLP



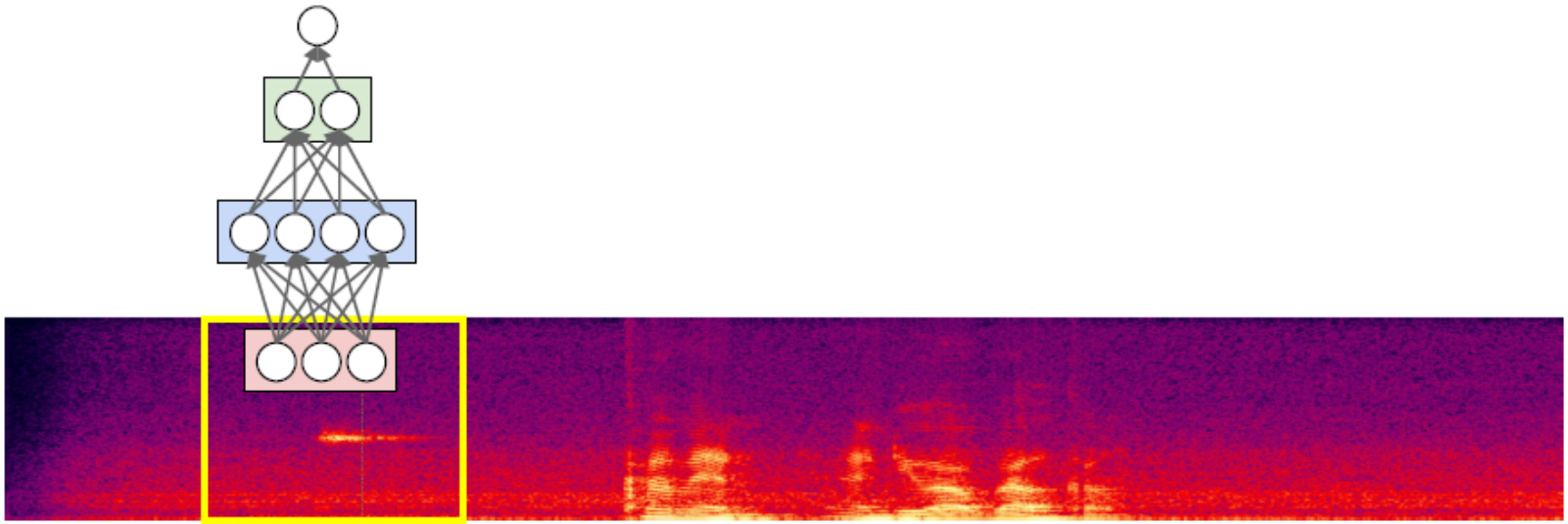
# Solution: Scan

- Scan for the target word
  - The spectral time-frequency components in a window are input to a welcome-detector MLP



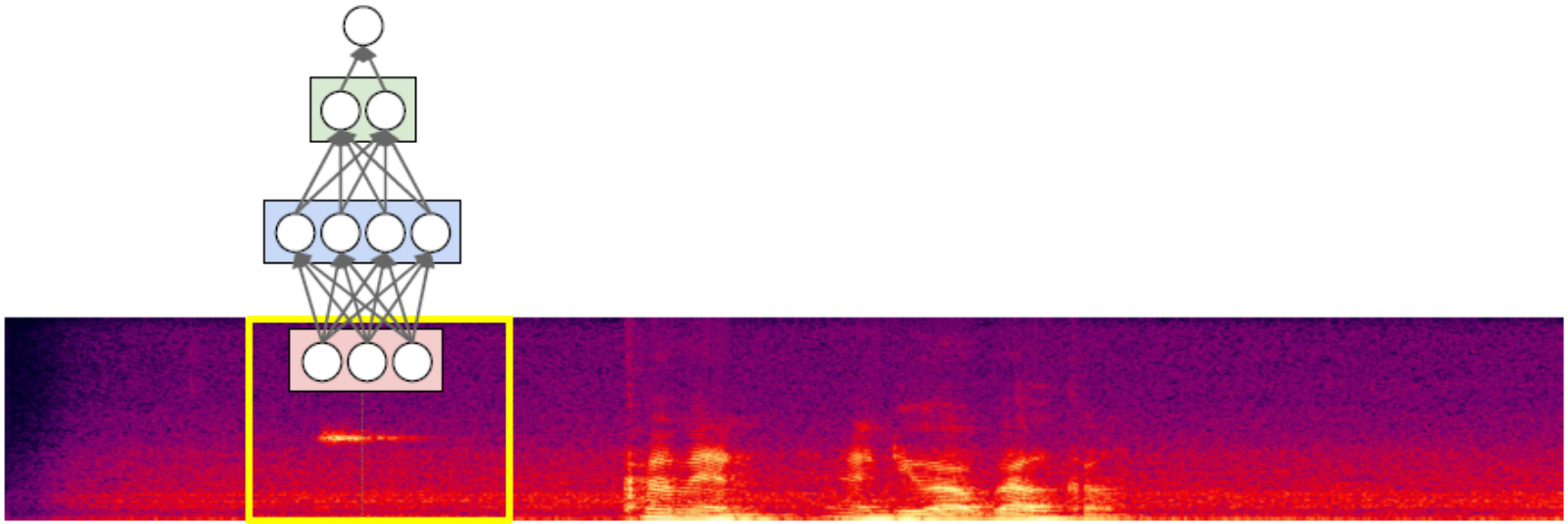
# Solution: Scan

- Scan for the target word
  - The spectral time-frequency components in a window are input to a welcome-detector MLP



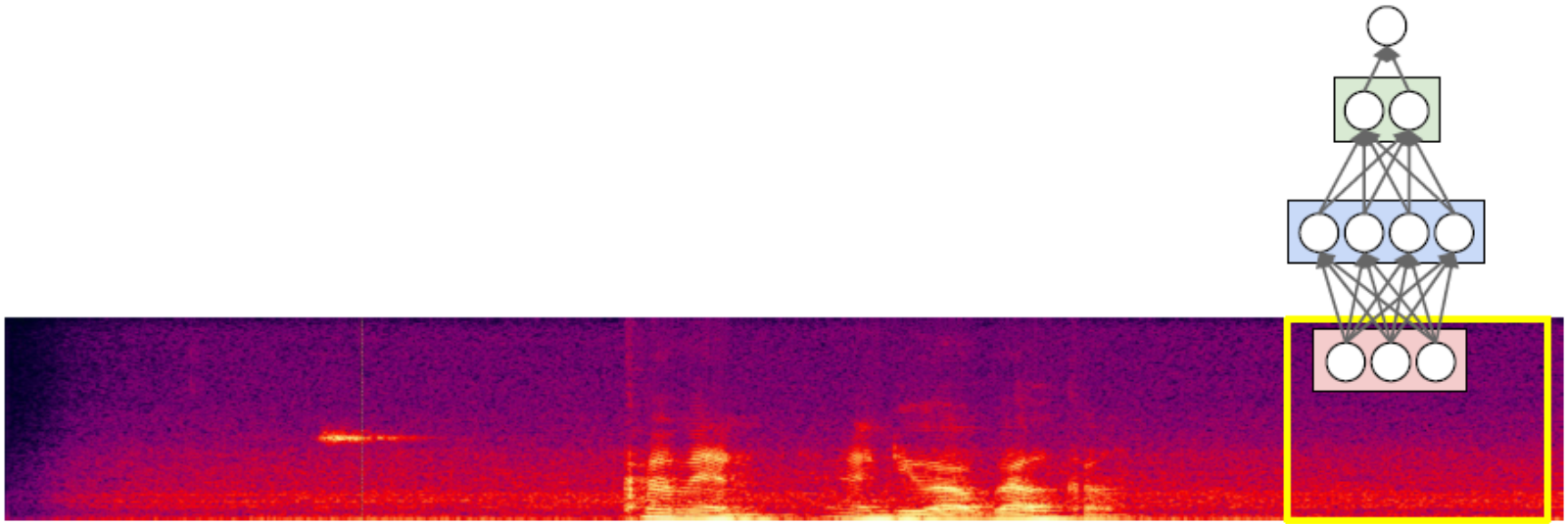
# Solution: Scan

- Scan for the target word
  - The spectral time-frequency components in a window are input to a welcome-detector MLP



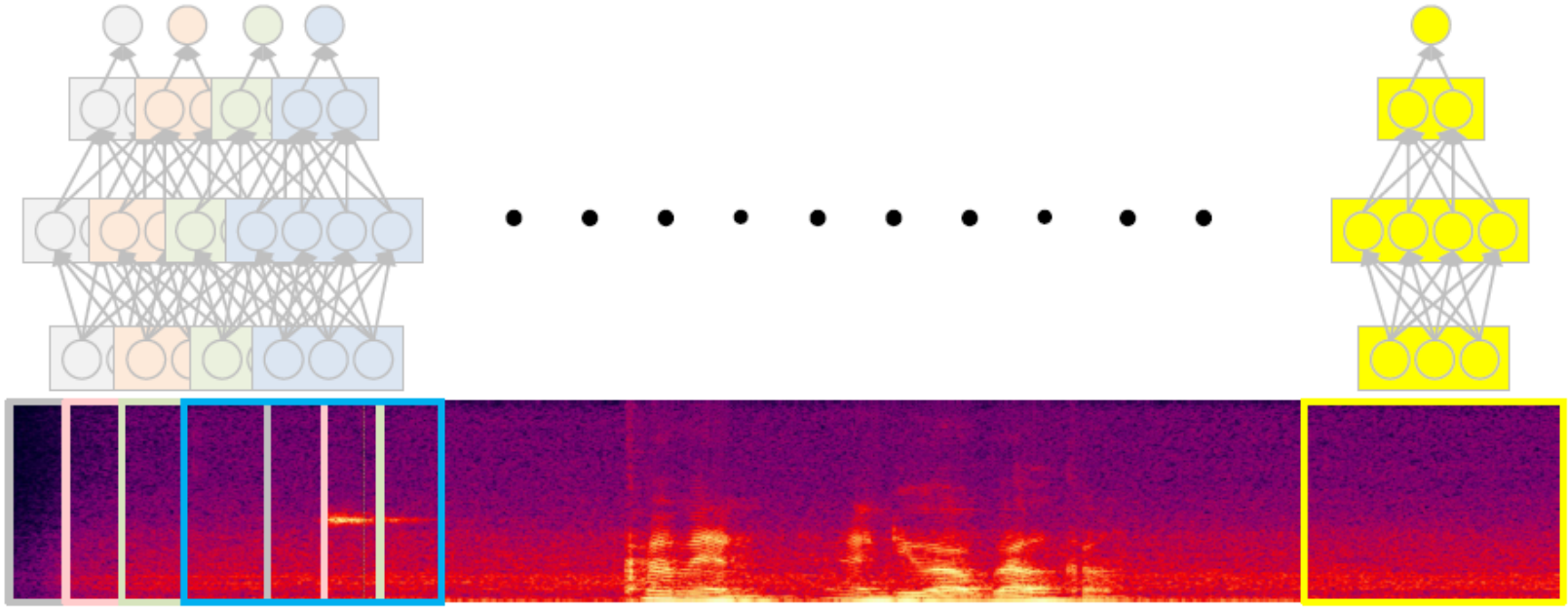
# Solution: Scan

- Scan for the target word
  - The spectral time-frequency components in a window are input to a welcome-detector MLP



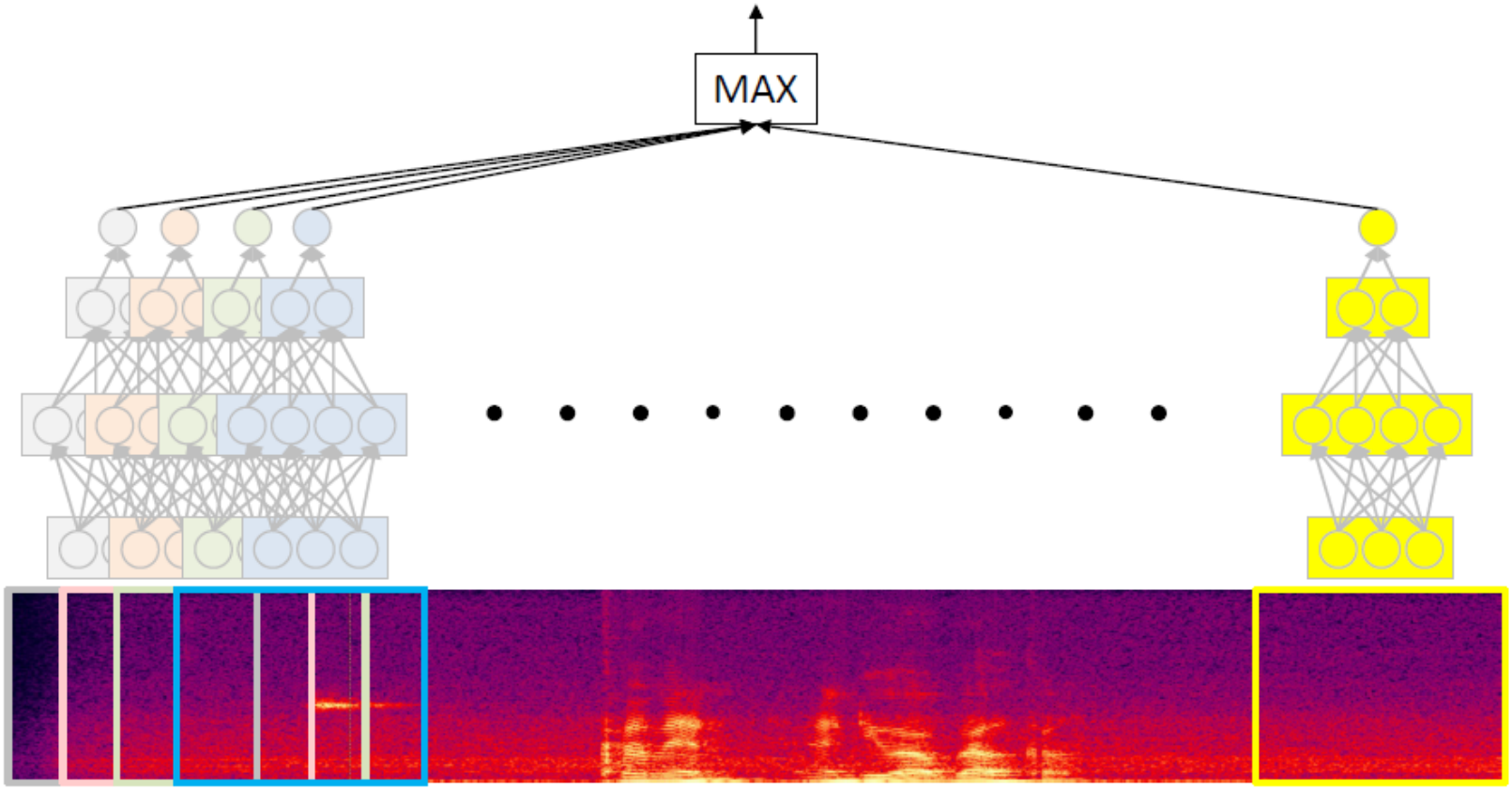
# Solution: Scan

- Does welcome occur in this recording?
  - We have classified many windows individually
  - Welcome may have occurred in any of them



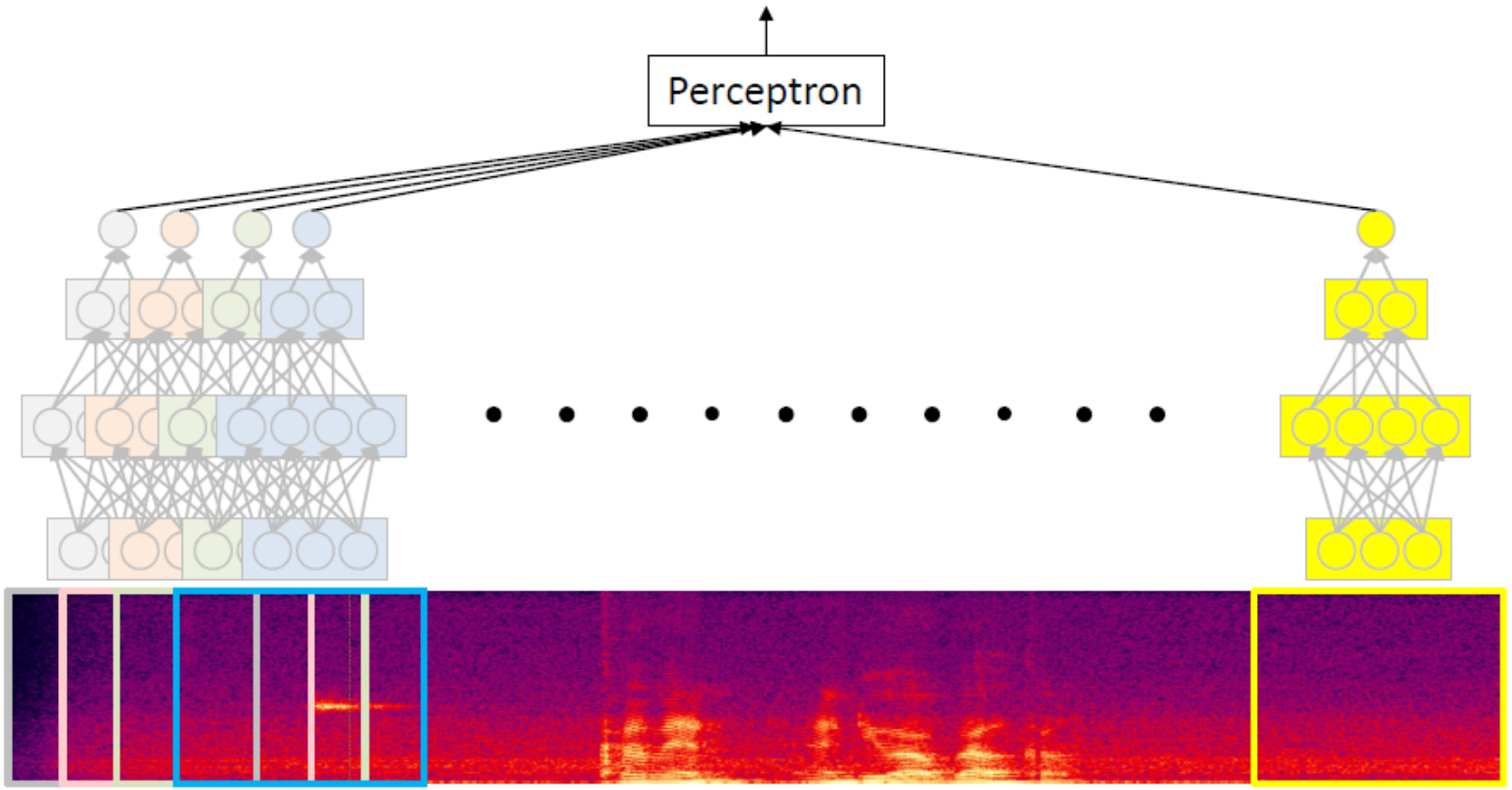
# Solution: Scan

- Does welcome occur in this recording?
  - We have classified many windows individually
  - Welcome may have occurred in any of them
  - Maximum of all the outputs (Equivalent of Boolean OR)



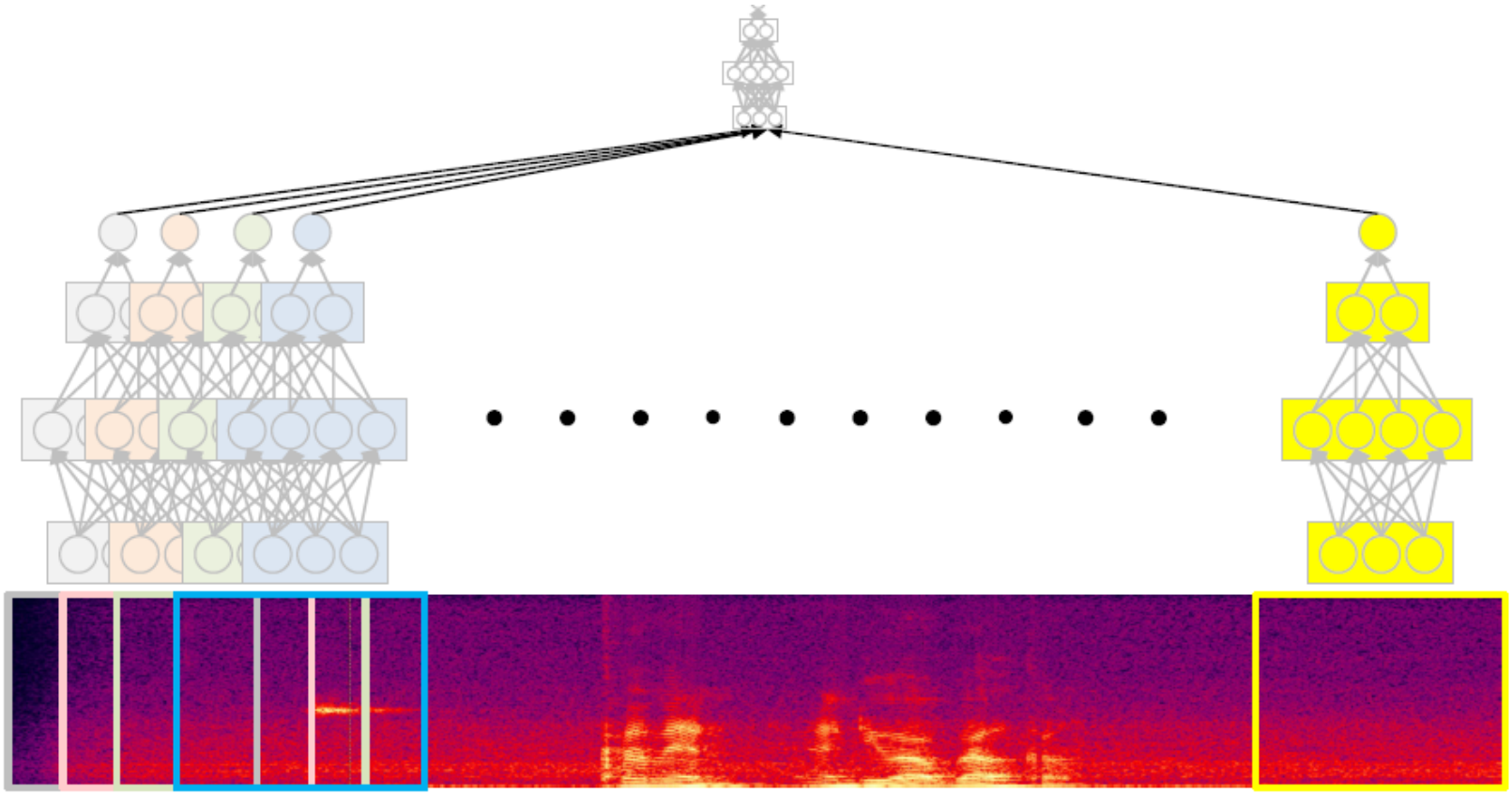
# Solution: Scan

- Does welcome occur in this recording?
  - Maximum of all the outputs (Equivalent of Boolean OR)
  - Or a proper softmax/logistic
    - Finding a welcome in adjacent windows makes it more likely that we didnt find noise
    - Adjacent windows can combine their evidence



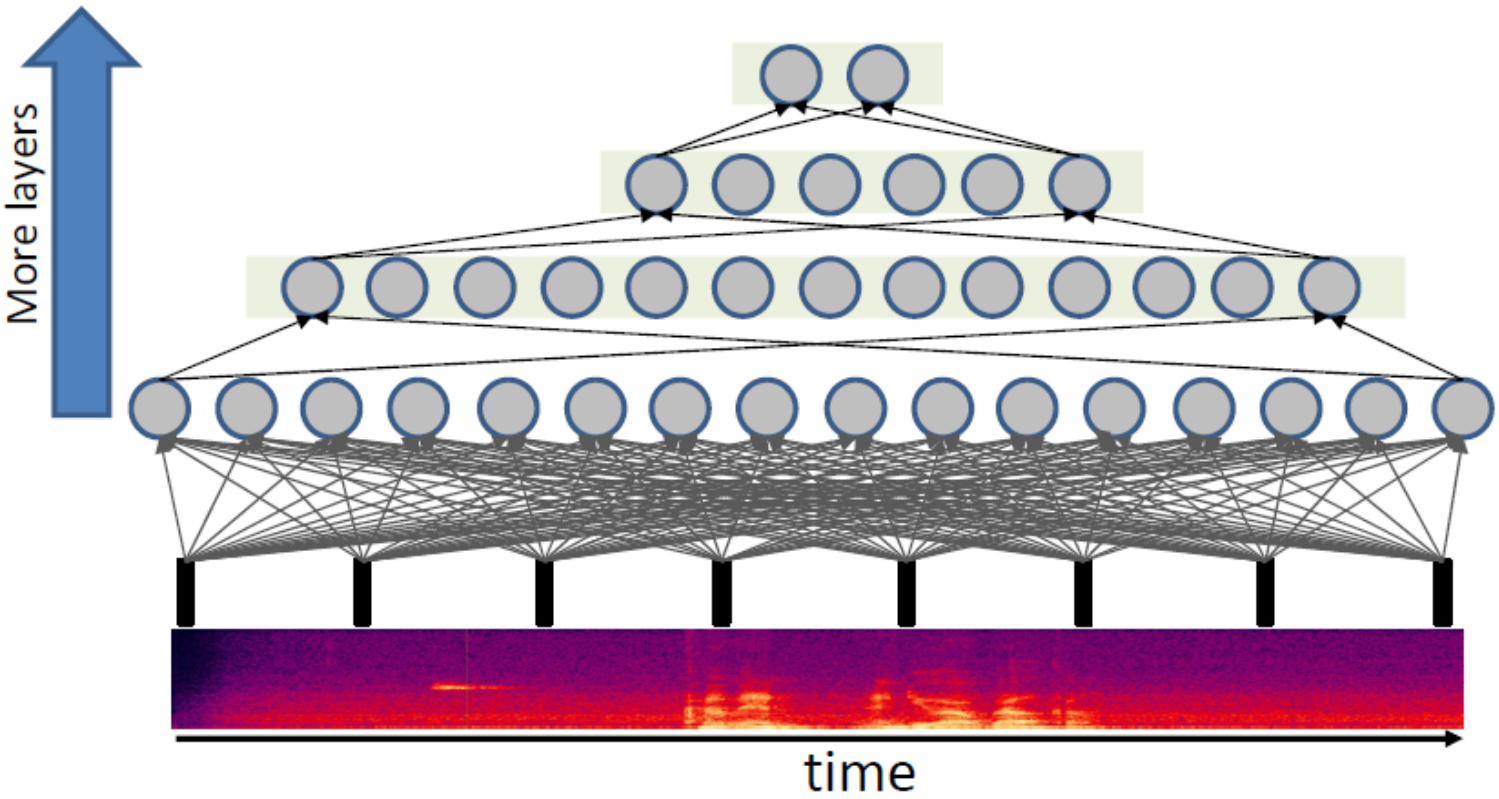
# Solution: Scan

- Does welcome occur in this recording?
  - Maximum of all the outputs (Equivalent of Boolean OR)
  - Or a proper softmax/logistic
  - Or even an MLP



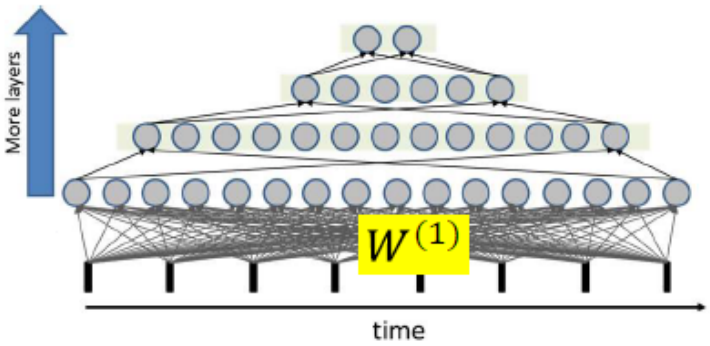
# Regular networks vs. scanning networks

- In a regular MLP every neuron in a layer is connected by a unique weight to every unit in the previous layer
  - All entries in the weight matrix are unique
  - The weight matrix is (generally) full



# Regular network

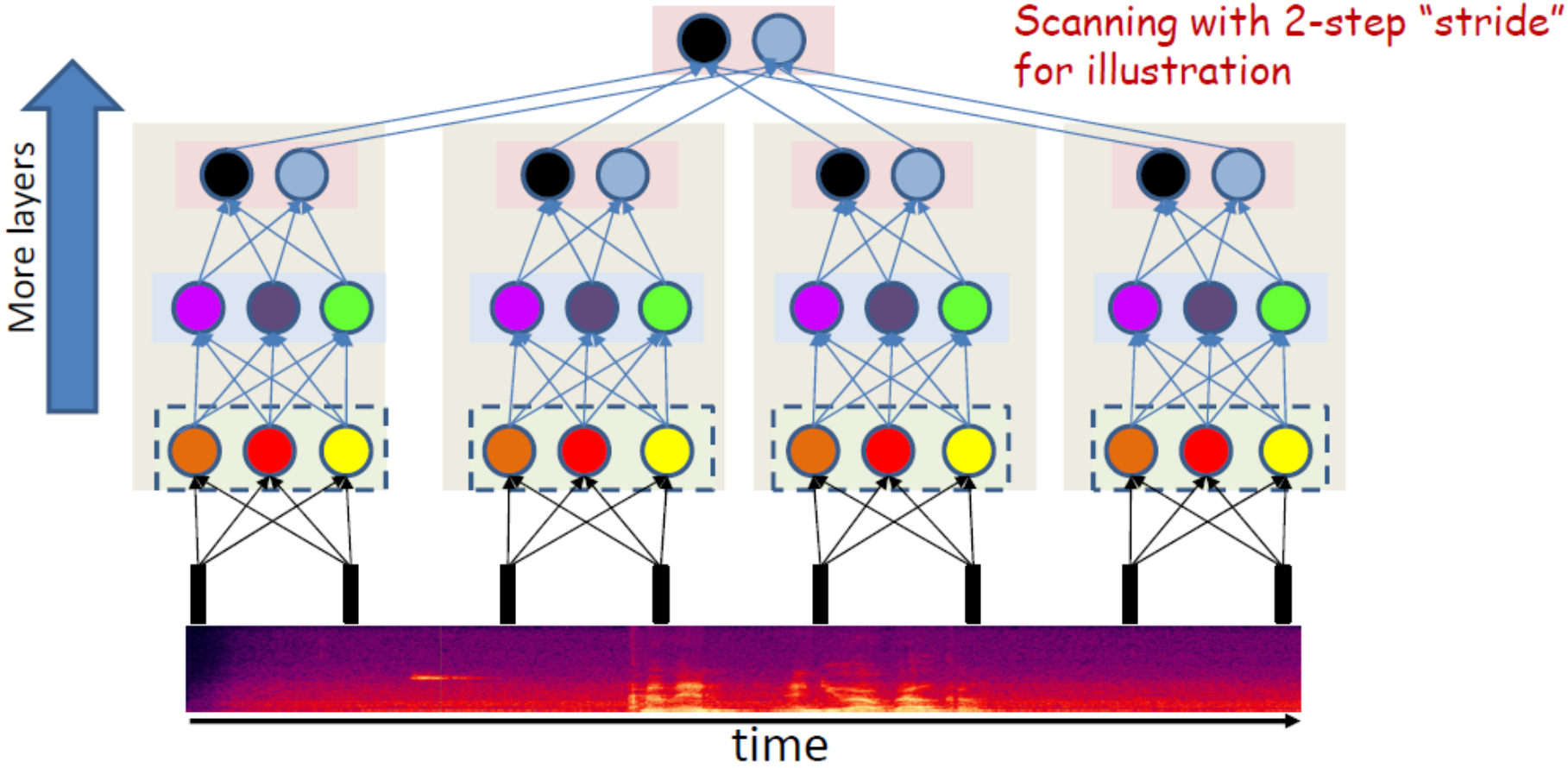
- Consider the first layer
  - Assume $N$ inputs and $M$ outputs
- The weights matrix is a full $M \times N$ matrix
  - Requiring $MN$ unique parameters




$$W^{(1)^{\mathrm{T}}} = \begin{bmatrix} w_{11} & w_{21} & w_{31} & w_{41} & \cdots & \cdots & \cdots & w_{N1} \\ w_{12} & w_{22} & w_{32} & w_{42} & \cdots & \cdots & \cdots & w_{N2} \\ w_{13} & w_{23} & w_{33} & w_{43} & \cdots & \cdots & \cdots & w_{N3} \\ w_{14} & w_{24} & w_{34} & w_{44} & \cdots & \cdots & \cdots & w_{N4} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ w_{1M} & w_{2M} & w_{3M} & w_{4M} & \cdots & \cdots & \cdots & w_{NM} \end{bmatrix}$$

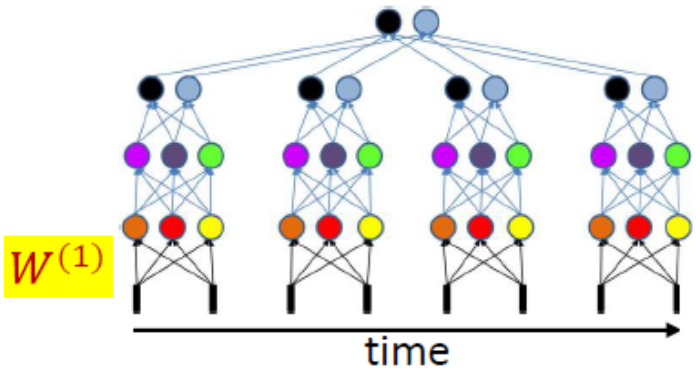
# Scanning networks

- In a scanning MLP each neuron is connected to a subset of neurons in the previous layer
  - The weights matrix is sparse
  - The weights matrix is block structured with identical blocks



# Scanning networks

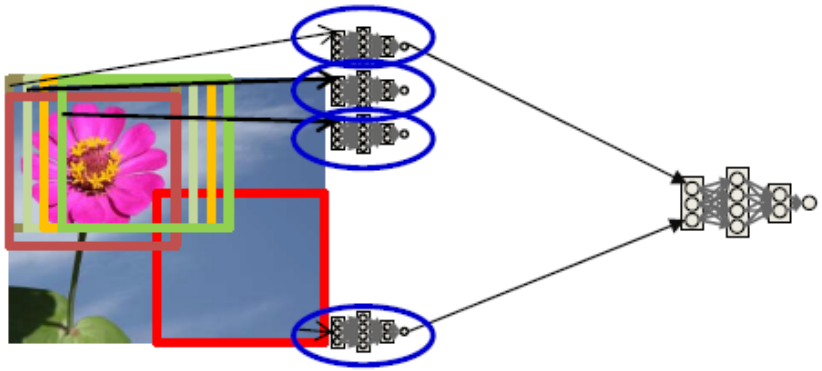
- In a scanning MLP each neuron is connected to a subset of neurons in the previous layer
  - The weights matrix is sparse
  - The weights matrix is block structured with identical blocks
  - **The network is a shared parameter model**
  - **Also, far fewer parameters**

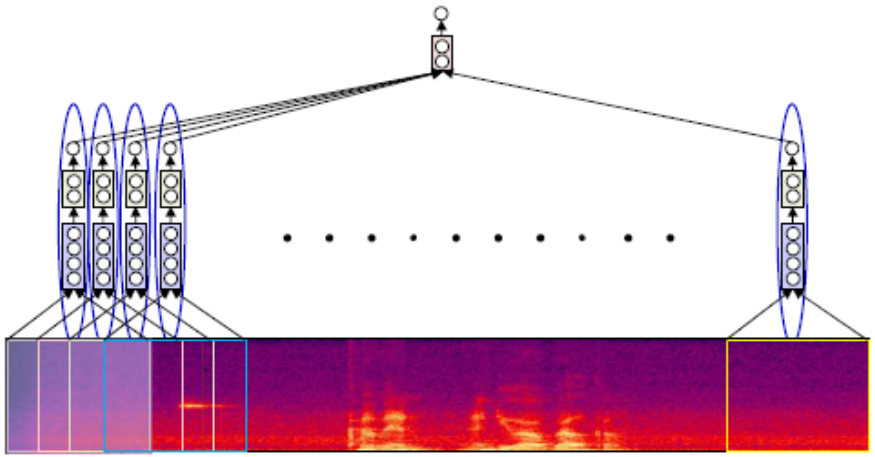



$$W^{(1)} = \begin{bmatrix} w_{11} & w_{12} & 0 & 0 & 0 & \cdots & \cdots & \cdots & 0 \\ w_{21} & w_{22} & 0 & 0 & 0 & \cdots & \cdots & \cdots & 0 \\ w_{31} & w_{32} & 0 & 0 & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & 0 & w_{11} & w_{12} & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & w_{21} & w_{22} & 0 & 0 & 0 & \vdots & \vdots \\ 0 & 0 & w_{31} & w_{32} & 0 & 0 & 0 & \vdots & \vdots \\ 0 & 0 & 0 & 0 & w_{11} & w_{12} & 0 & 0 & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & w_{31} & w_{32} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \\ . \\ . \\ X_N \end{bmatrix}$$

# Training the network: constraint

- These are shared parameter networks
  - All lower-level subnets are identical
    - Are all searching for the same pattern
  - Any update of the parameters of one copy of the subnet must equally update all copies





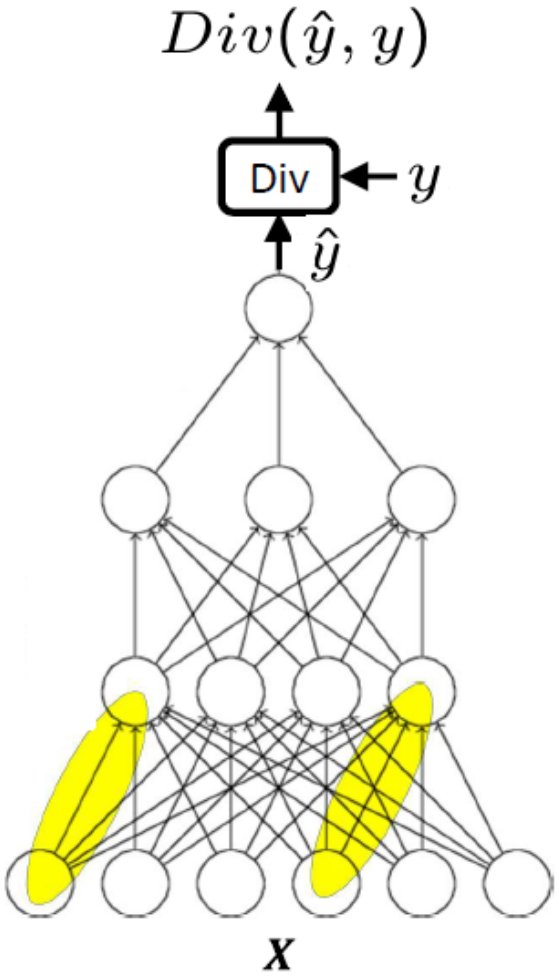
# Learning in shared parameter networks

- Consider a simple network with shared weights

$$w_{ij}^{k} = w_{mn}^{\ell} = w^{s}$$

  - A weight $w_{ij}^{k}$ is required to be identical to the weight $w_{mn}^{\ell}$

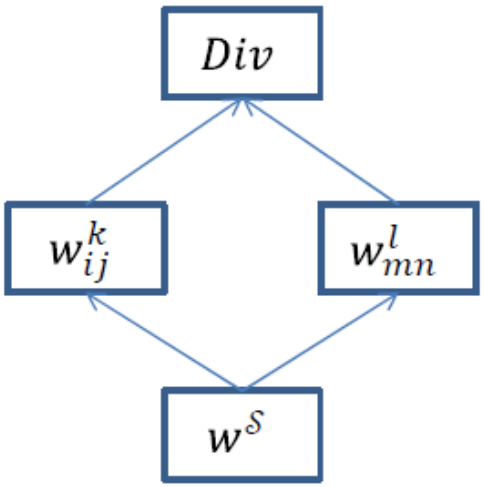
- For any training instance $X$, a small perturbation of $w^{s}$ perturbs both $w_{ij}^{k}$ and $w_{mn}^{\ell}$ identically
  - Each of these perturbations will individually influence the loss $Div(\hat{y}, y)$



# Computing the divergence of shared parameters

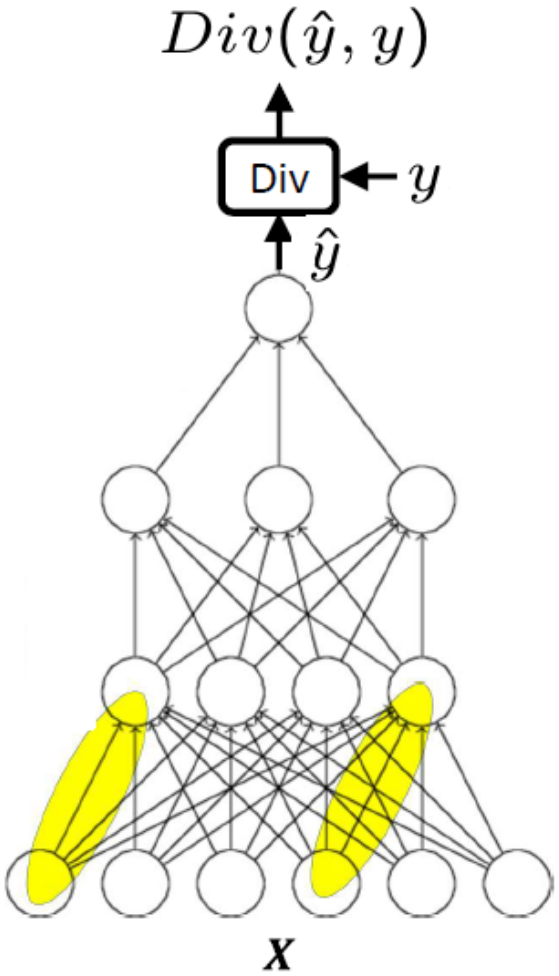
- Each of the individual terms can be computed via backpropagation

Influence diagram




$$\frac{dDiv}{dw^S} = \frac{\partial Div}{\partial w_{ij}^k}\frac{dw_{ij}^k}{dw^S} + \frac{\partial Div}{\partial w_{mn}^l}\frac{dw_{mn}^l}{dw^S}$$

$$= \frac{\partial Div}{\partial w_{ij}^k} + \frac{\partial Div}{\partial w_{mn}^l}$$

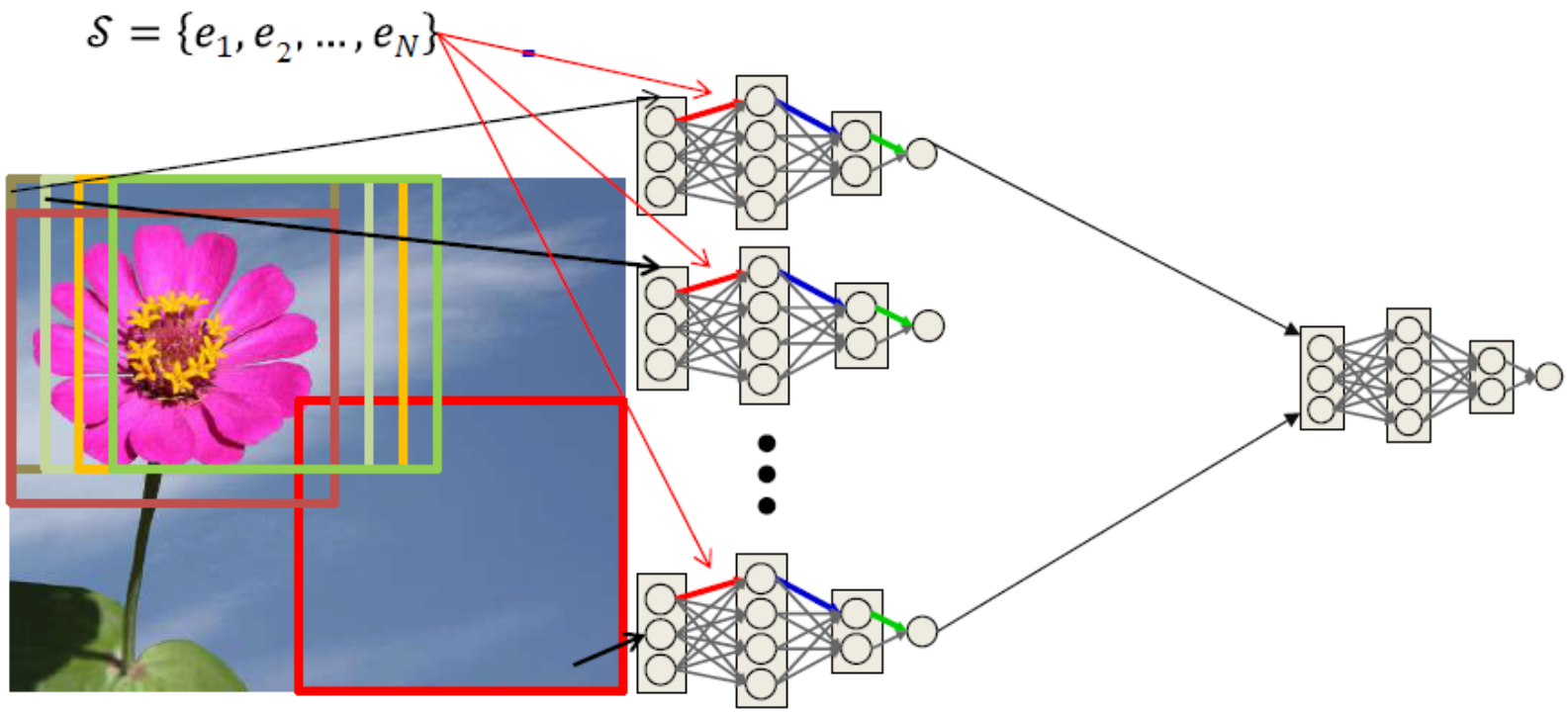


# Computing the divergence of shared parameters

- More generally, let $S$ be any set of edges that have a common value, and $w^S$ be the common weight of the set
  - E.g. the set of all red weights in the figure

$$\frac{d\,Div}{d\,w^S} = \sum_{i=1}^{N} \frac{\partial Div}{\partial w^{e_i}}$$

- The individual terms in the sum can be computed via backpropagation



# Training networks with shared parameters

- Gradient descent algorithm:
- Initialize all weights $\mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_K$
- Do:
  - For every set $\mathcal{S}$:
    - Compute:

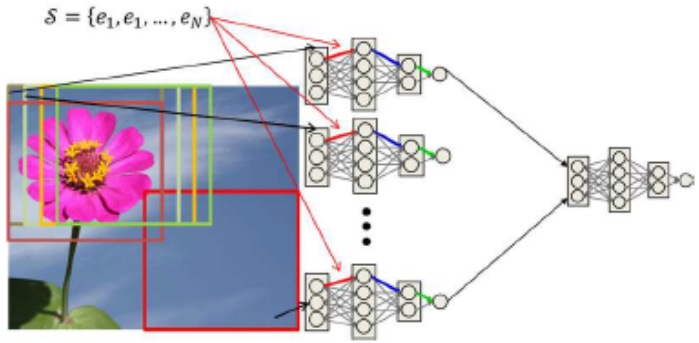
$$\nabla_{\mathcal{S}} Loss = \frac{dLoss}{dw^{\mathcal{S}}}$$

$$w^{\mathcal{S}} = w^{\mathcal{S}} - \eta \nabla_{\mathcal{S}} Loss^T$$

    - For every $(k, i, j) \in \mathcal{S}$ update:

$$w_{i,j}^{(k)} = w^{\mathcal{S}}$$

- Until $Loss$ has converged




63

# Training networks with shared parameters

- Gradient descent algorithm:
- Initialize all weights $\mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_K$
- Do:
  - For every set $\mathcal{S}$:
    - Compute:

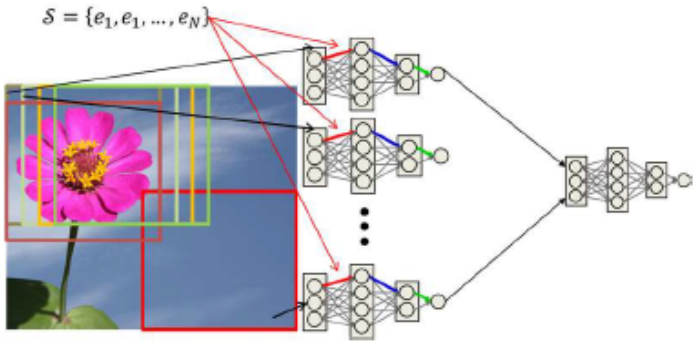
$$\nabla_{\mathcal{S}} Loss = \frac{dLoss}{dw^{\mathcal{S}}}$$

$$w^{\mathcal{S}} = w^{\mathcal{S}} - \eta \nabla_{\mathcal{S}} Loss^T$$

    - For every $(k, i, j) \in \mathcal{S}$ update:

$$w_{i,j}^{(k)} = w^{\mathcal{S}}$$

- Until $Loss$ has converged




64

# Training networks with shared parameters

- For every training instance $X$
  - For every set $\mathcal{S}$:
    - For every $(k, i, j) \in \mathcal{S}$:

$$\frac{dLoss}{dw^{\mathcal{S}}} \mathrel{+}= \frac{\partial Div}{\partial w_{i,j}^{(k)}}$$

  - $\nabla_{\mathcal{S}} Loss = \frac{dLoss}{dw^{\mathcal{S}}}$
  - Compute:

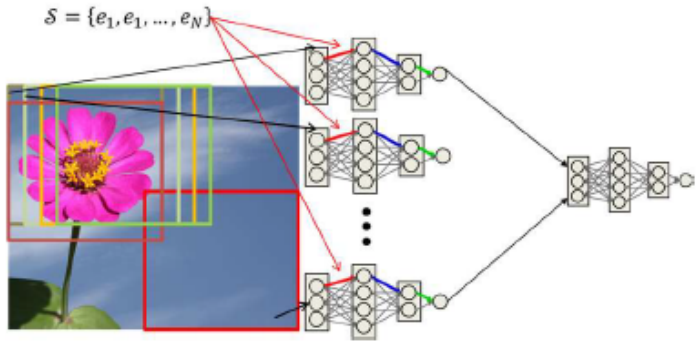
$$\nabla_{\mathcal{S}} Loss = \frac{dLoss}{dw^{\mathcal{S}}}$$

$$w^{\mathcal{S}} = w^{\mathcal{S}} - \eta \nabla_{\mathcal{S}} Loss^{T}$$

  - For every $(k, i, j) \in \mathcal{S}$ update:

$$w_{i,j}^{(k)} = w^{\mathcal{S}}$$

- Until $Loss$ has converged




65

# Training networks with shared parameters

- For every training instance $X$
  - For every set $\mathcal{S}$:
    - For every $(k,i,j) \in \mathcal{S}$:

$$\frac{dLoss}{dw^{\mathcal{S}}} \mathrel{+}= \frac{\partial Div}{\partial w_{i,j}^{(k)}}$$

Computed by Backprop

  - $\nabla_{\mathcal{S}} Loss = \frac{dLoss}{dw^{\mathcal{S}}}$
  - Compute:

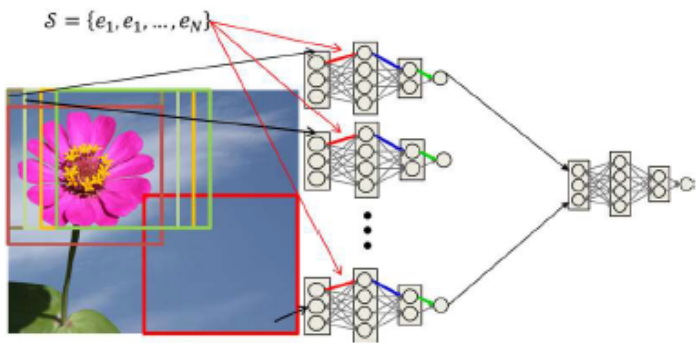
$$\nabla_{\mathcal{S}} Loss = \frac{dLoss}{dw^{\mathcal{S}}}$$

$$w^{\mathcal{S}} = w^{\mathcal{S}} - \eta \nabla_{\mathcal{S}} Loss^T$$

  - For every $(k,i,j) \in \mathcal{S}$ update:

$$w_{i,j}^{(k)} = w^{\mathcal{S}}$$

- Until $Loss$ has converged




66

# What is a convolution

- Scanning an image with a filter
  - Note: a filter is really just a perceptron, with weights and a bias

Example 5x5 image with binary pixels

| | | | | |
|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |

Example 3x3 filter

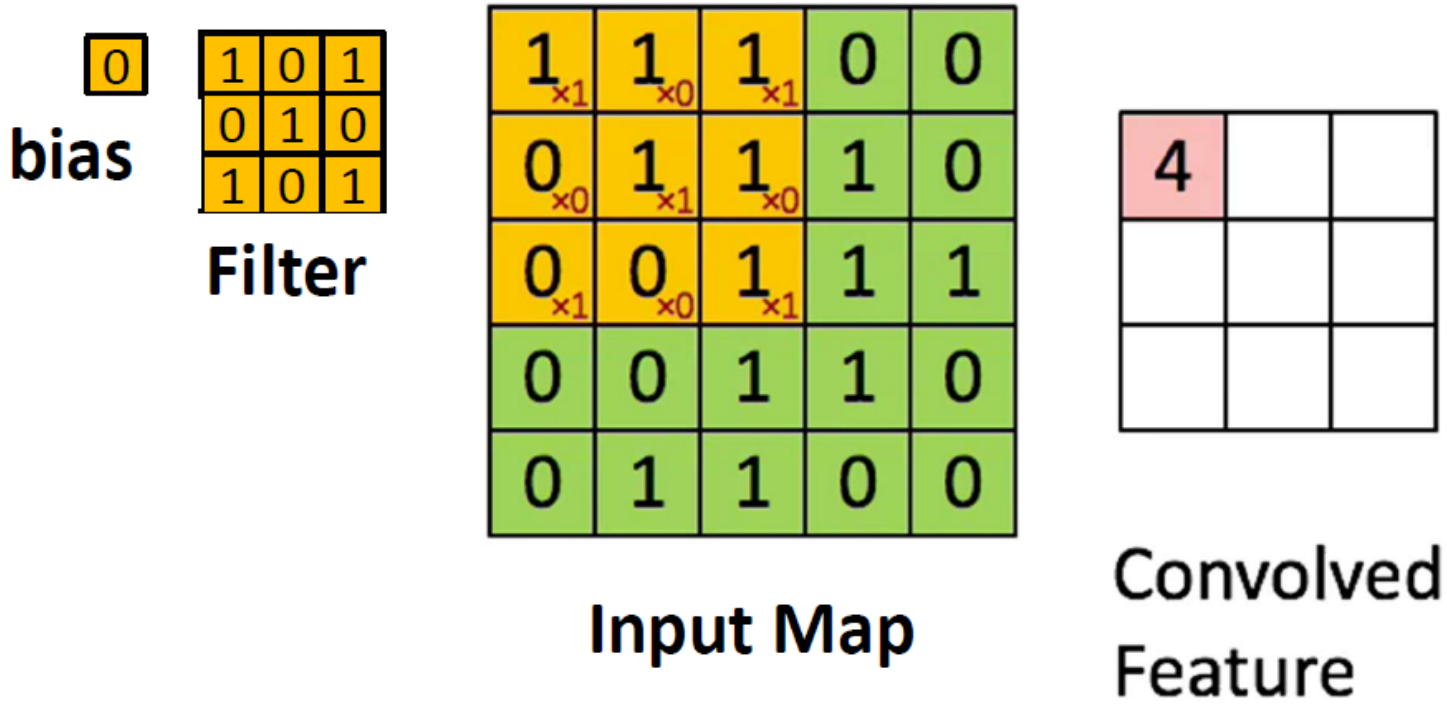
| | | |
|---|---|---|
| 1 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |

bias

| |
|---|
| 0 |

# What is a convolution

- Scanning an image with a filter
  - At each location, the filter and the underlying map values are multiplied component wise, and the products are added along with the bias



# What is a convolution

- Scanning an image with a filter
  - The filter may proceed by more than 1 pixel at a time
  - E.g. with a stride of two pixels per shift

Example 5x5 image with binary pixels

| | | | | |
|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |

Example 3x3 filter

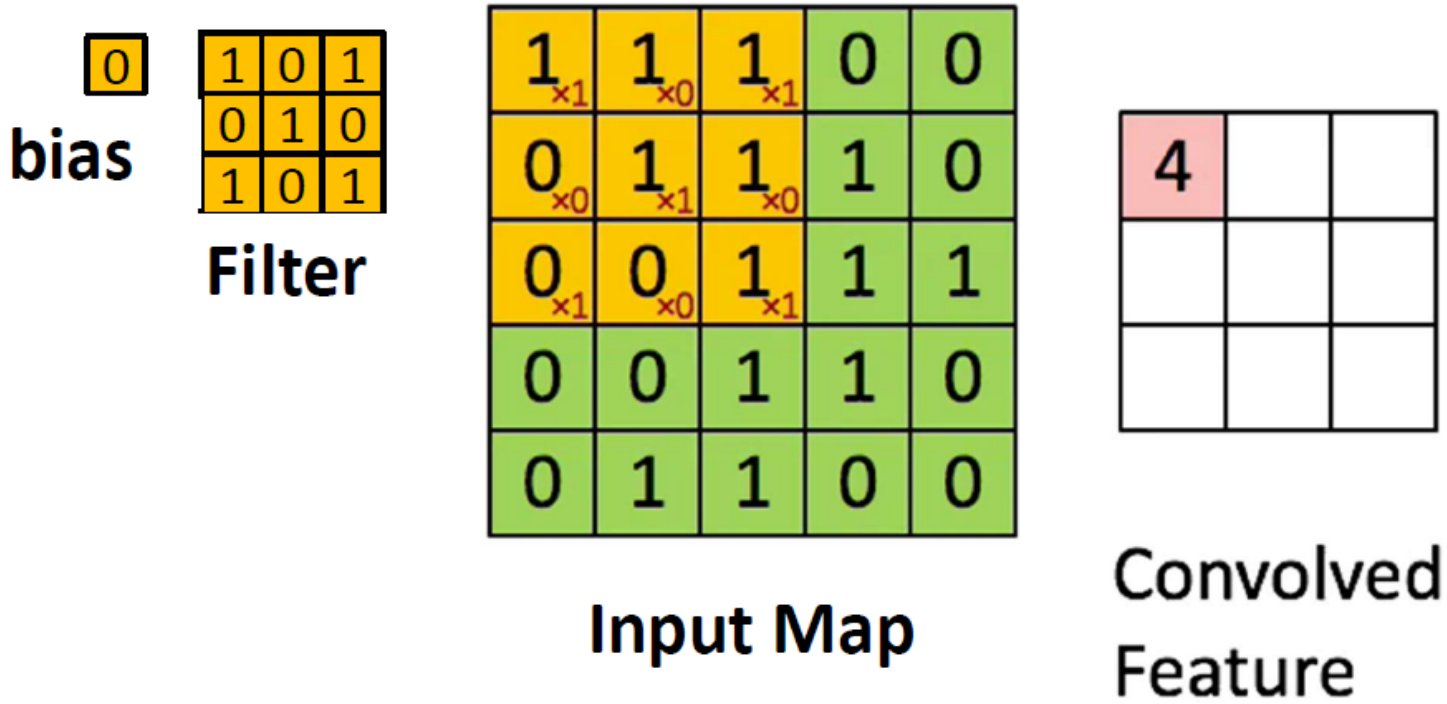
| | | |
|---|---|---|
| 1 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |

bias

| |
|---|
| 0 |

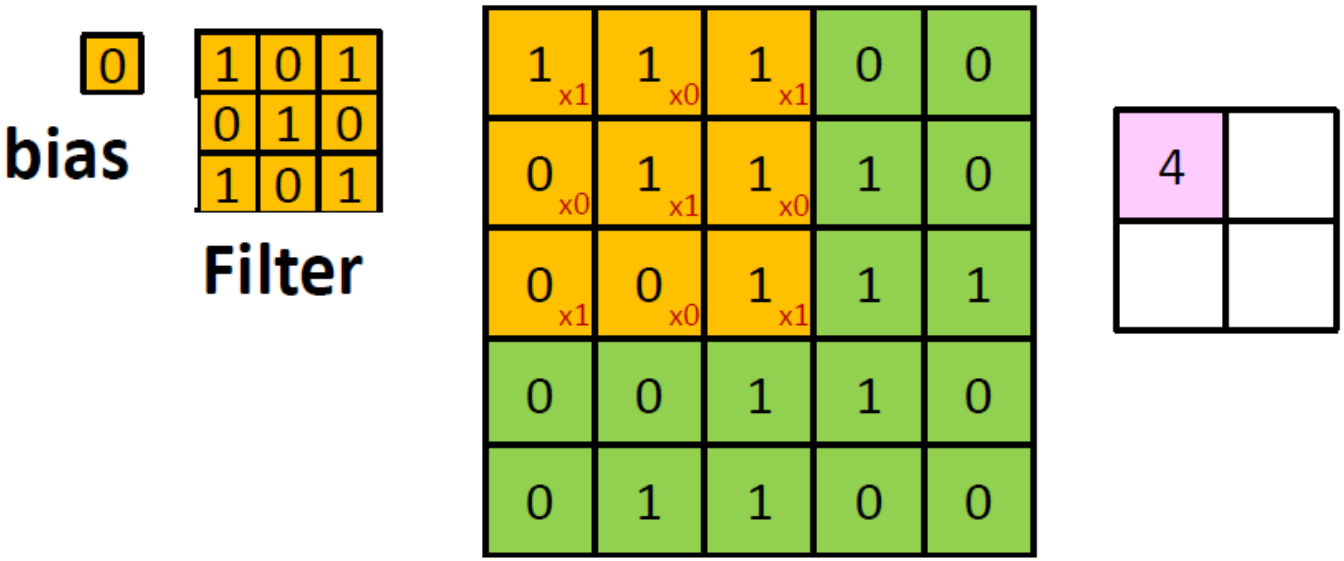
# What is a convolution

- Scanning an image with a filter
  - The filter may proceed by more than 1 pixel at a time
  - E.g. with a stride of two pixels per shift



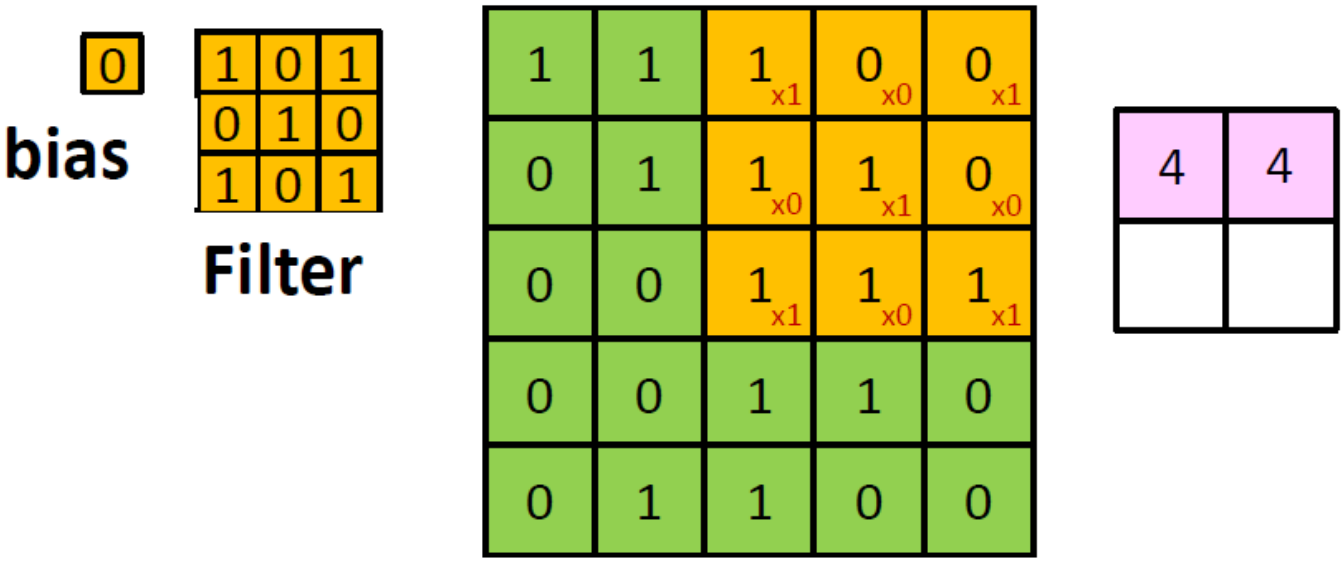
# What is a convolution

- Scanning an image with a filter
  - The filter may proceed by more than 1 pixel at a time
  - E.g. with a stride of two pixels per shift



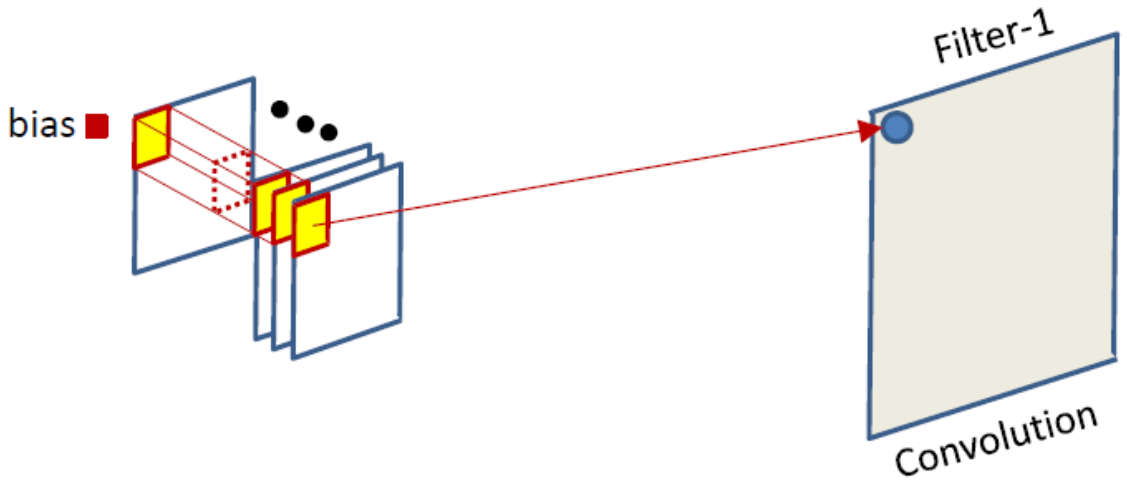
# What is a convolution

- Scanning an image with a filter
  - The filter may proceed by more than 1 pixel at a time
  - E.g. with a stride of two pixels per shift



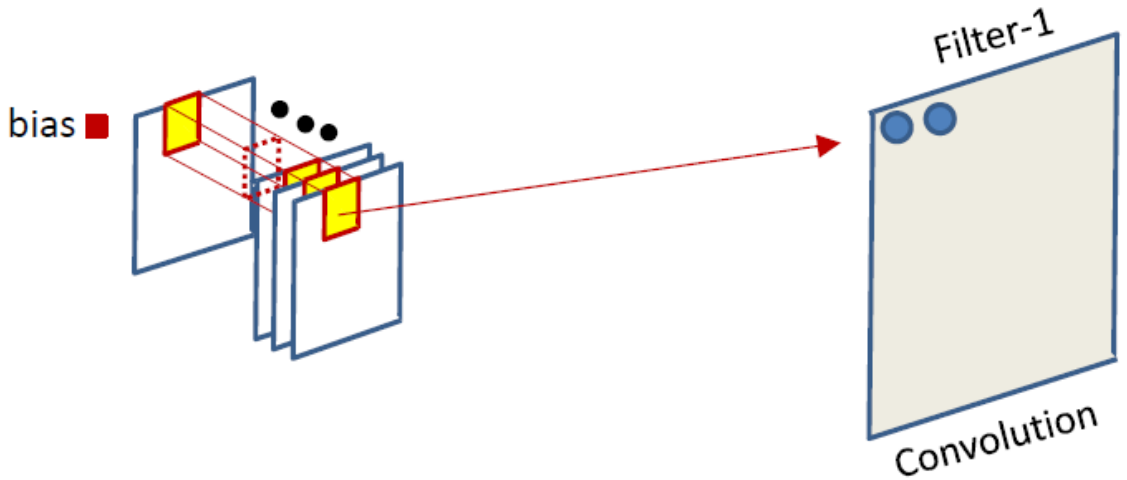
# The cube view of input

- The computation of the convolutional map at any location sums the convolutional outputs at all planes



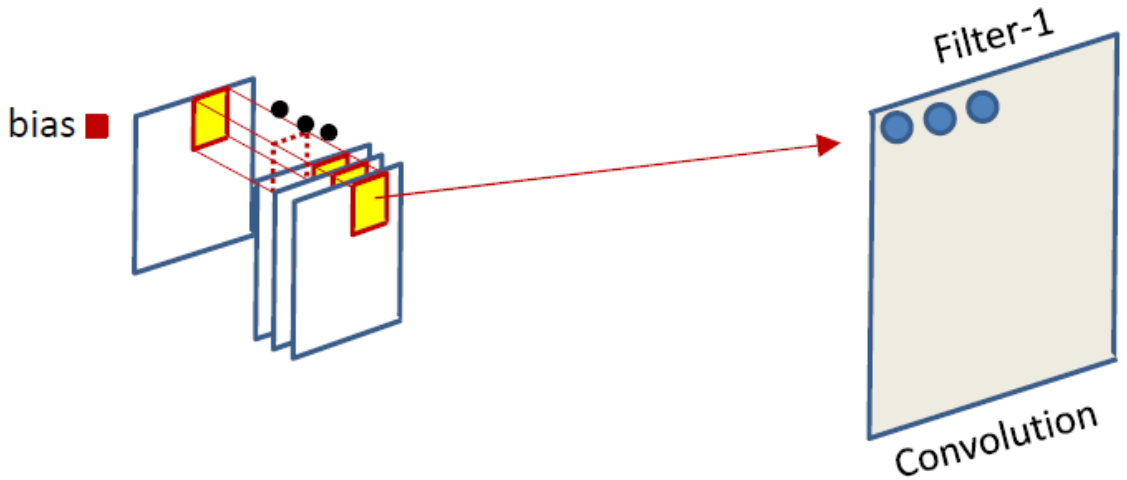
# The cube view of input

- The computation of the convolutional map at any location sums the convolutional outputs at all planes



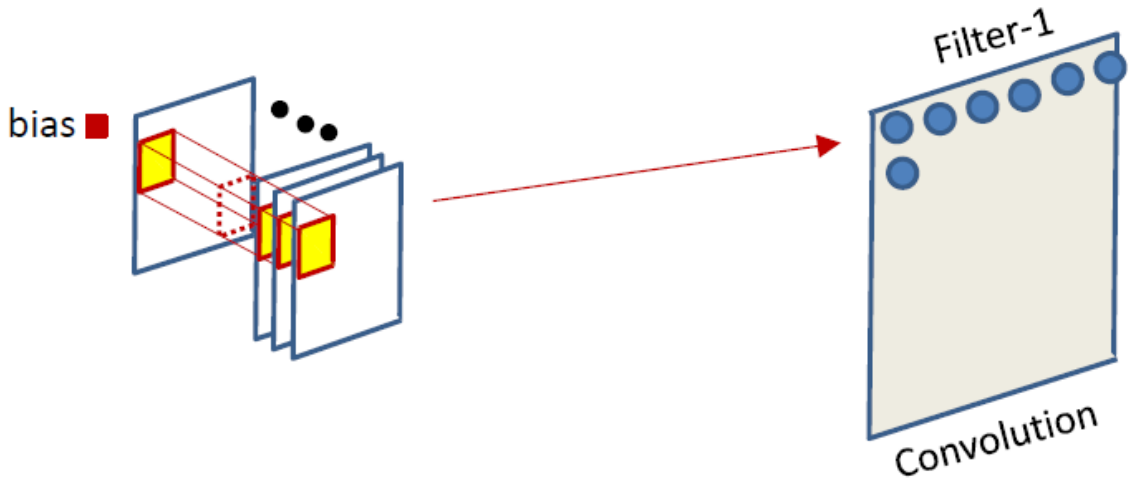
# The cube view of input

- The computation of the convolutional map at any location sums the convolutional outputs at all planes



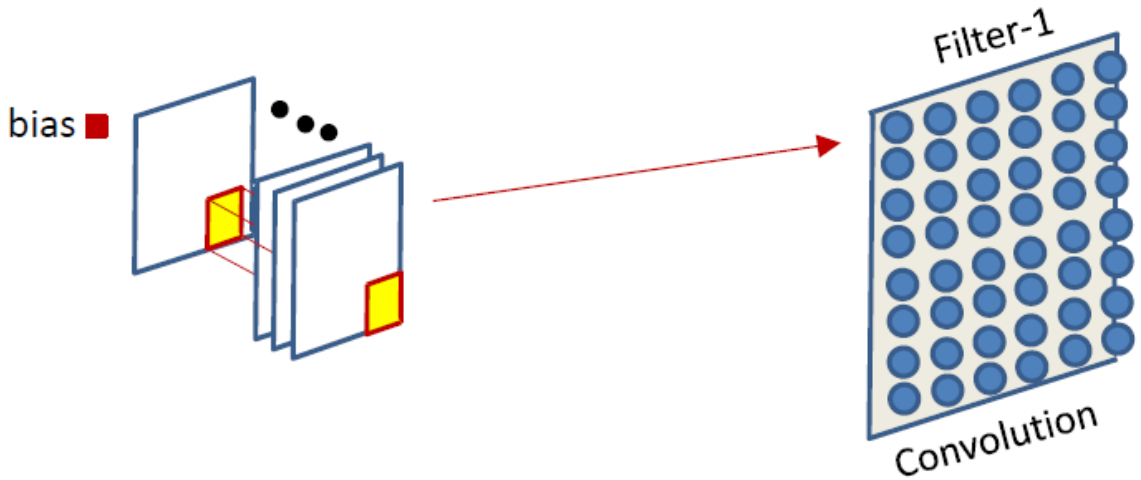
# The cube view of input

- The computation of the convolutional map at any location sums the convolutional outputs at all planes



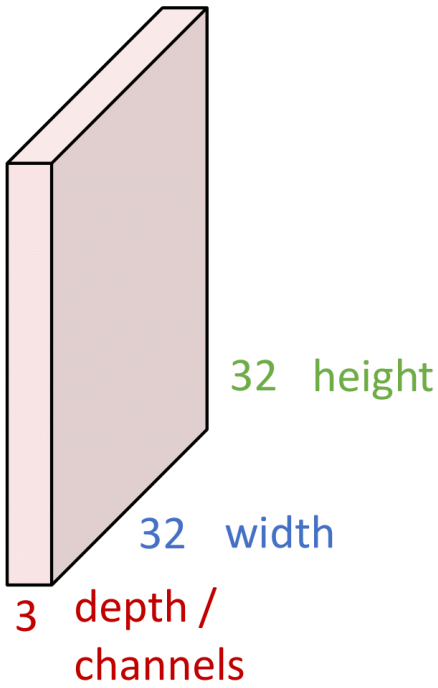
# The cube view of input

- The computation of the convolutional map at any location sums the convolutional outputs at all planes



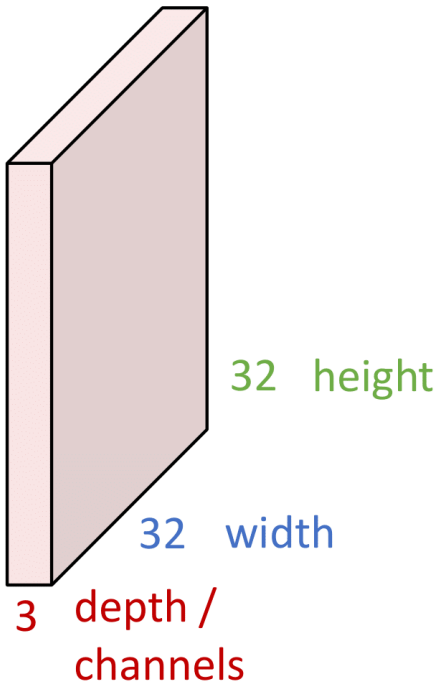
# Convolution Layer

3x32x32 image: preserve spatial structure

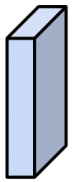


# Convolution Layer

3x32x32 image

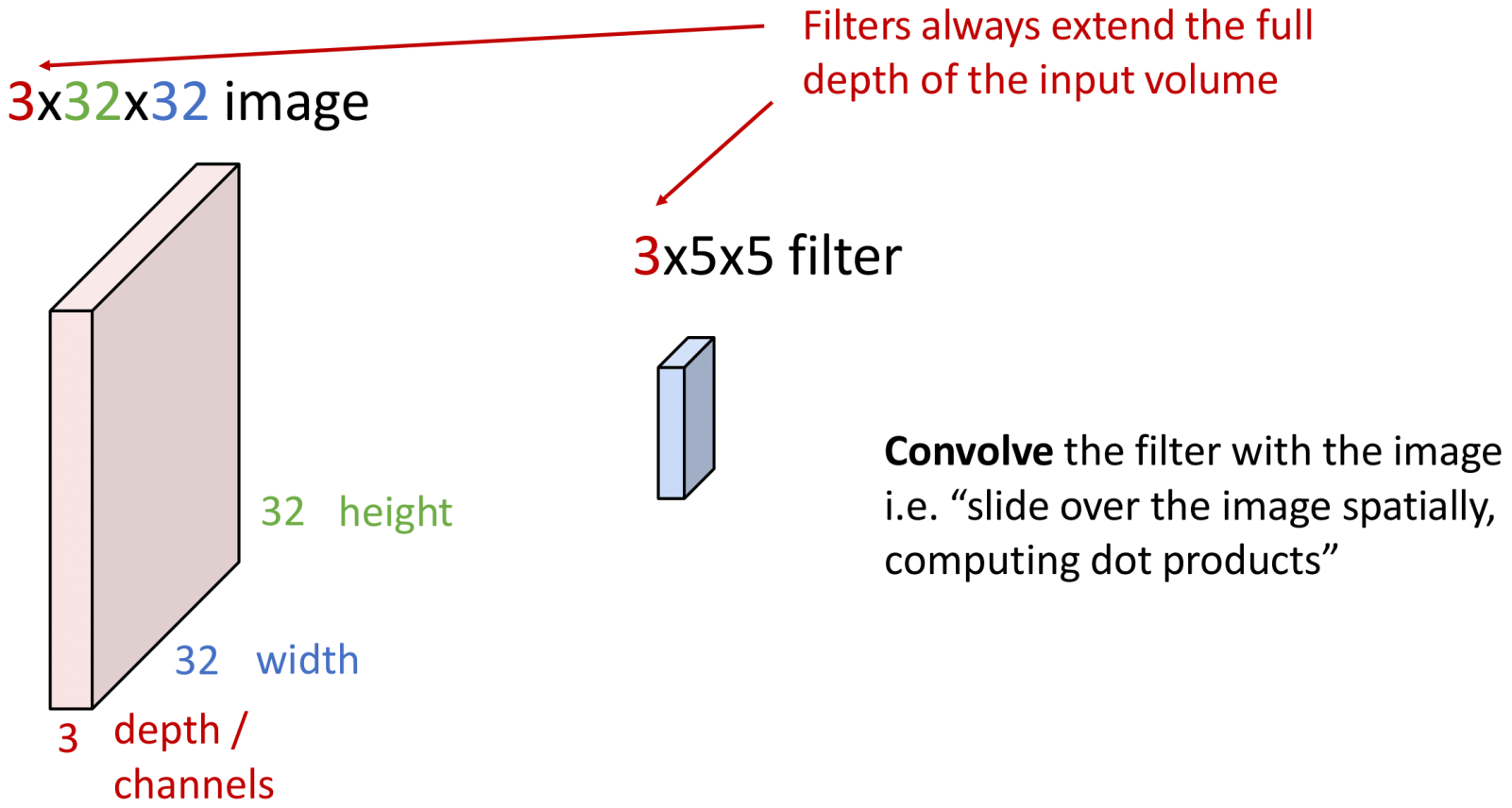



3x5x5 filter

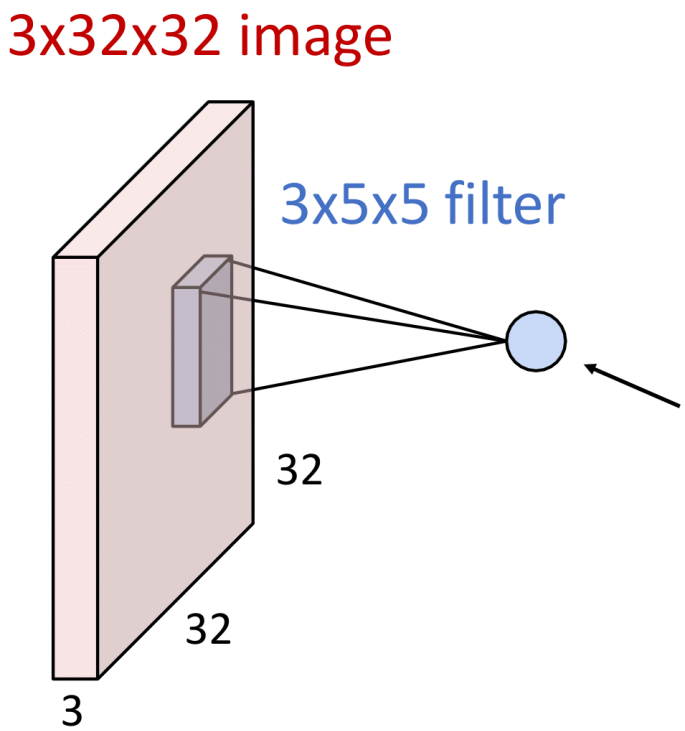


**Convolve** the filter with the image i.e. “slide over the image spatially, computing dot products”

# Convolution Layer



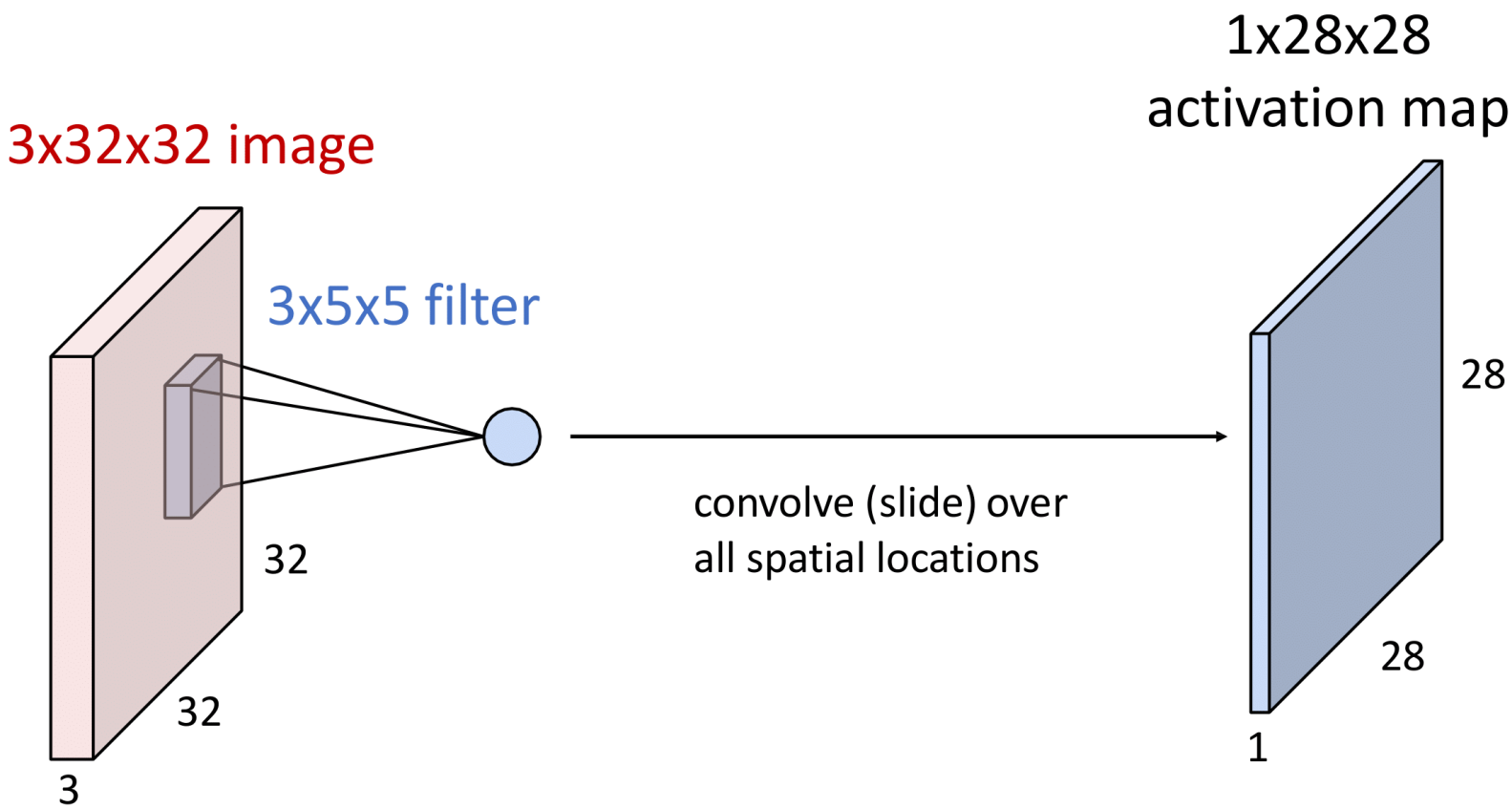
# Convolution Layer



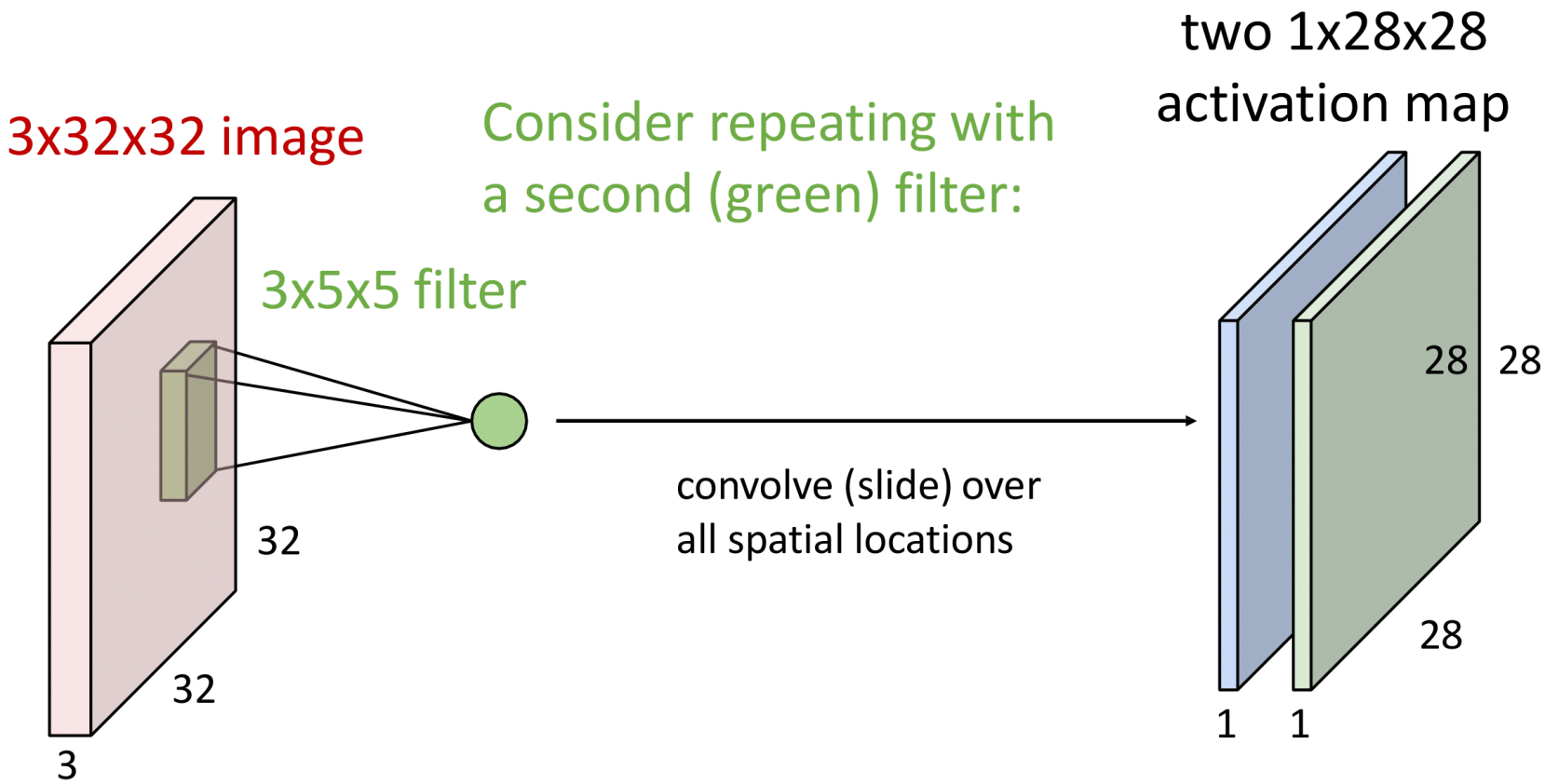

**1 number:**
the result of taking a dot product between the filter and a small 3x5x5 chunk of the image
(i.e. 3*5*5 = 75-dimensional dot product + bias)

$$w^T x + b$$

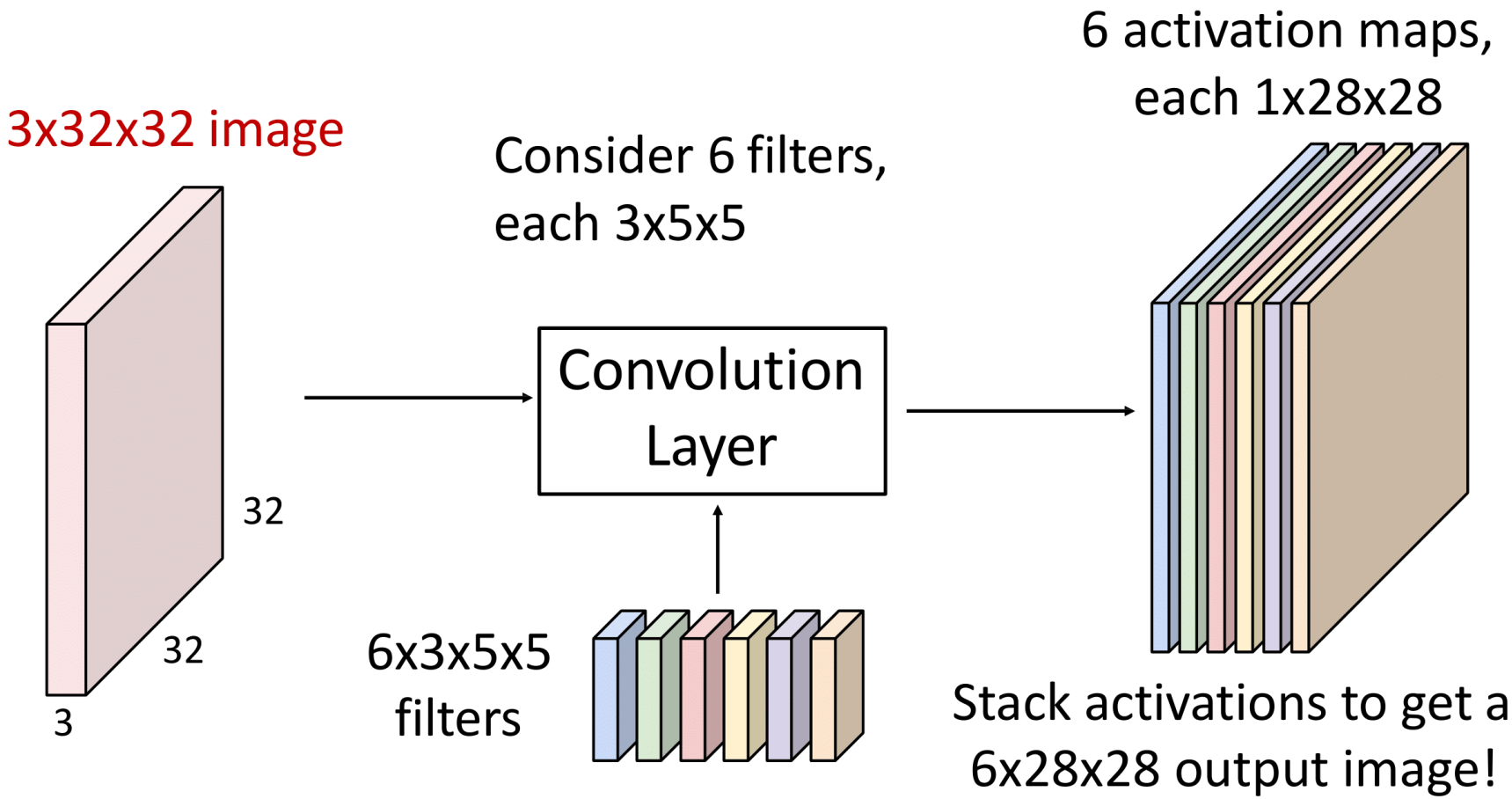
# Convolution Layer



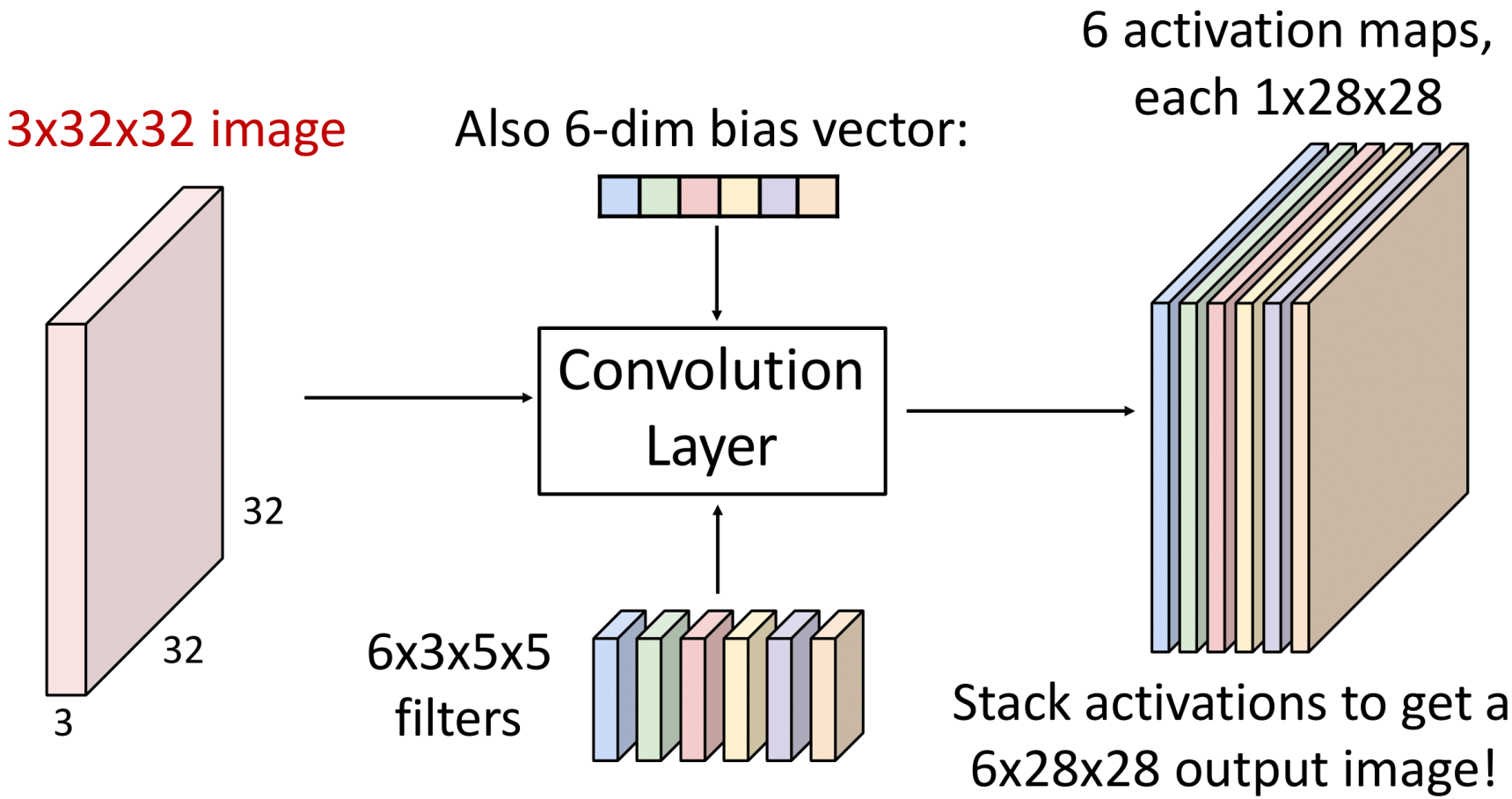
# Convolution Layer



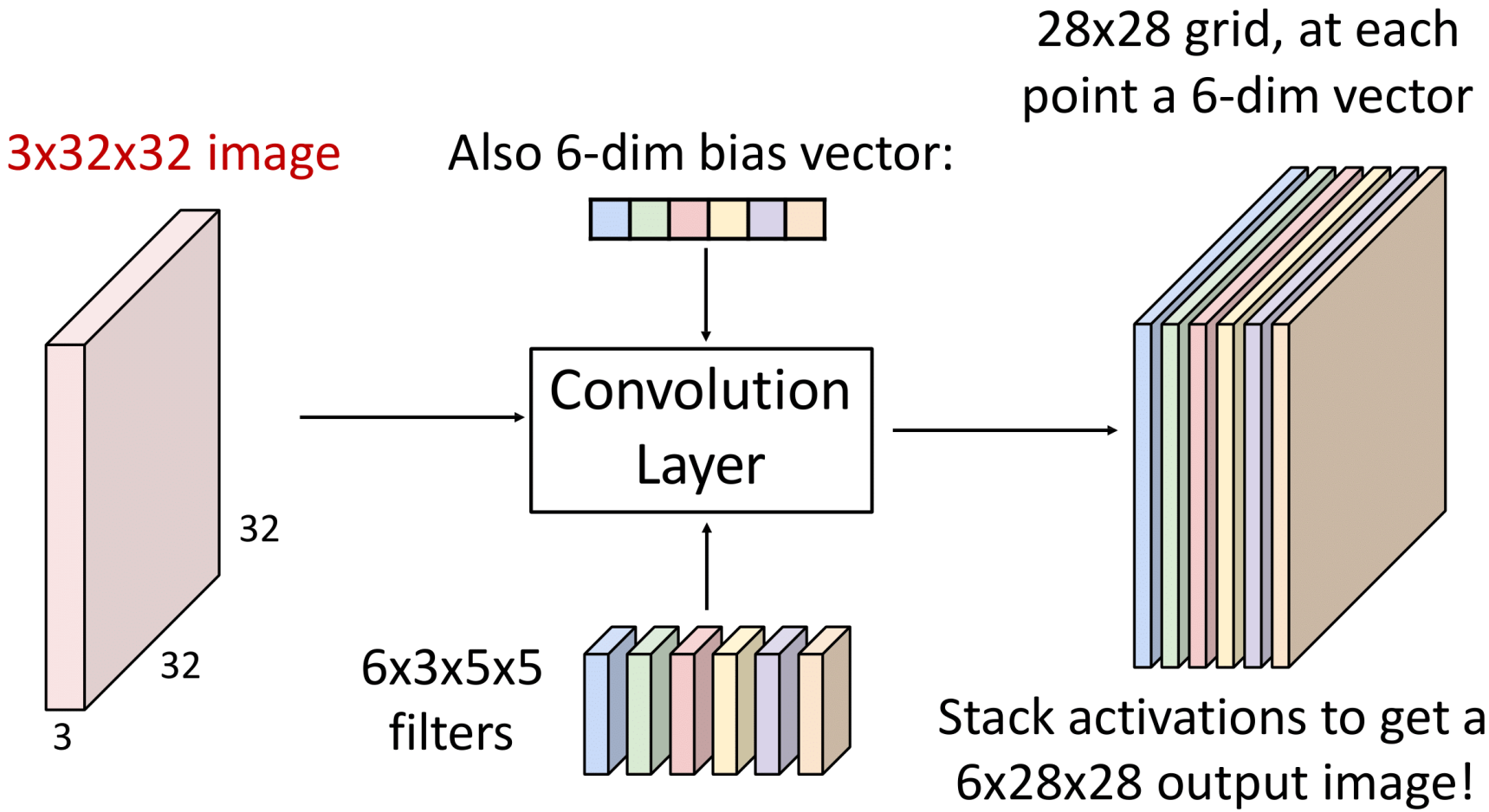
# Convolution Layer



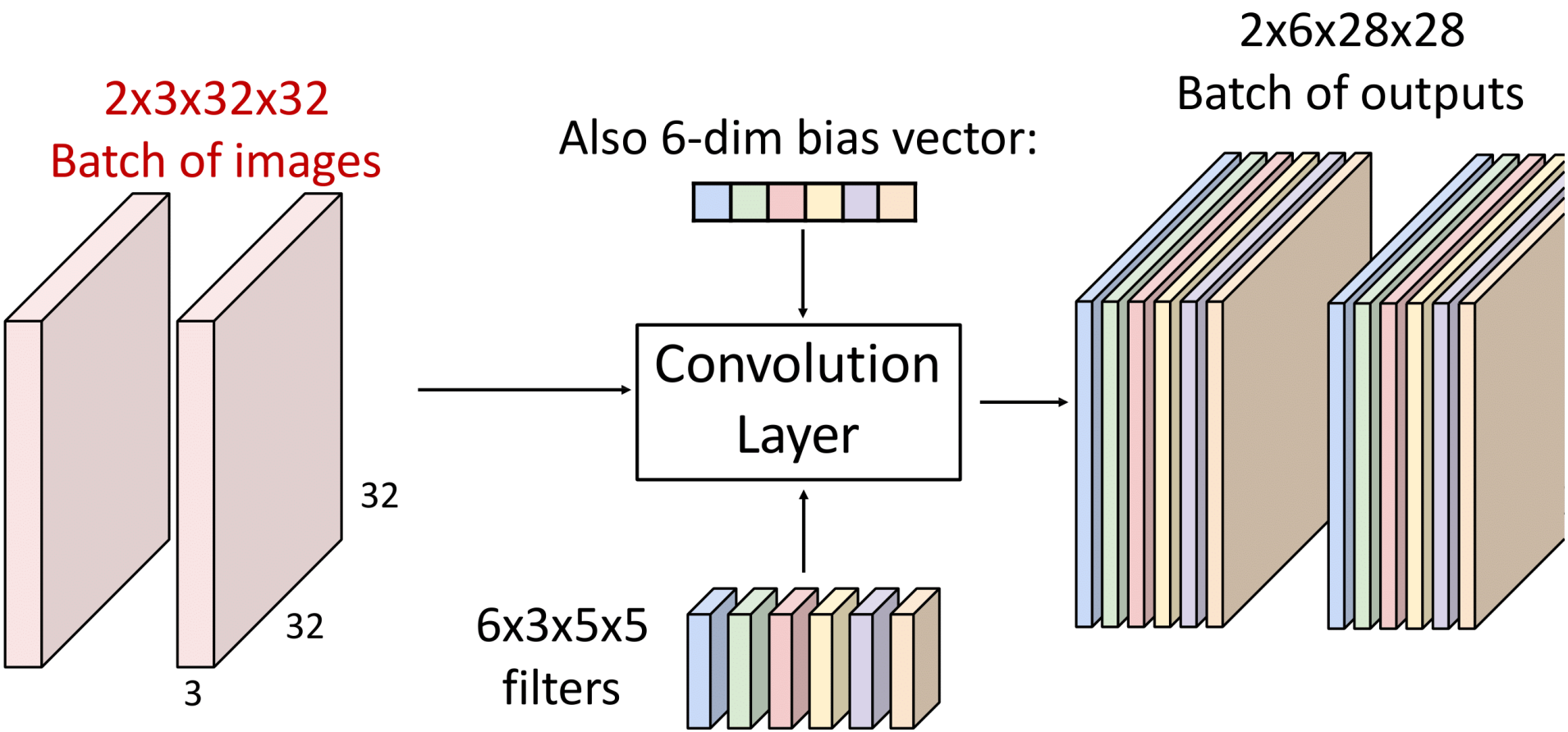
# Convolution Layer



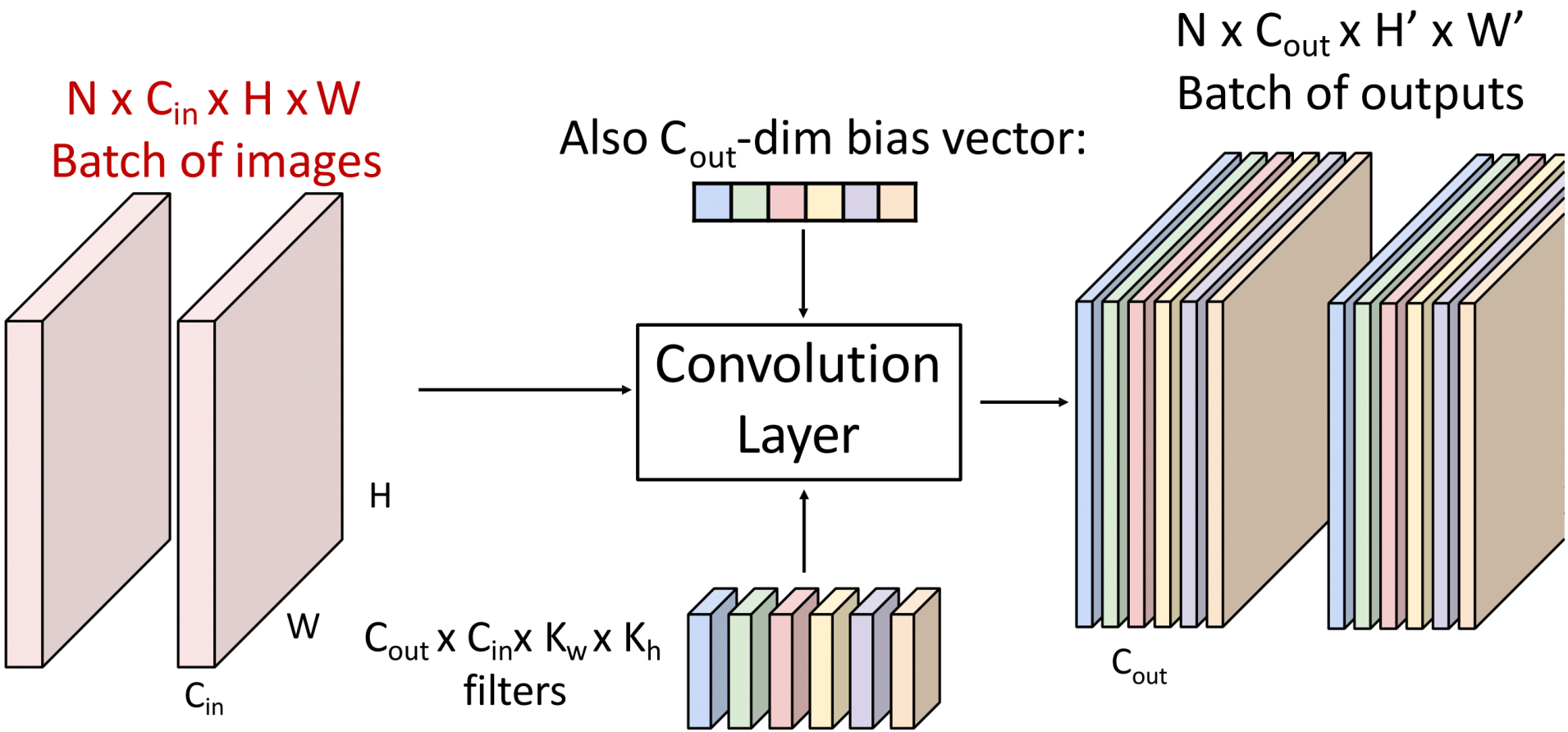
# Convolution Layer



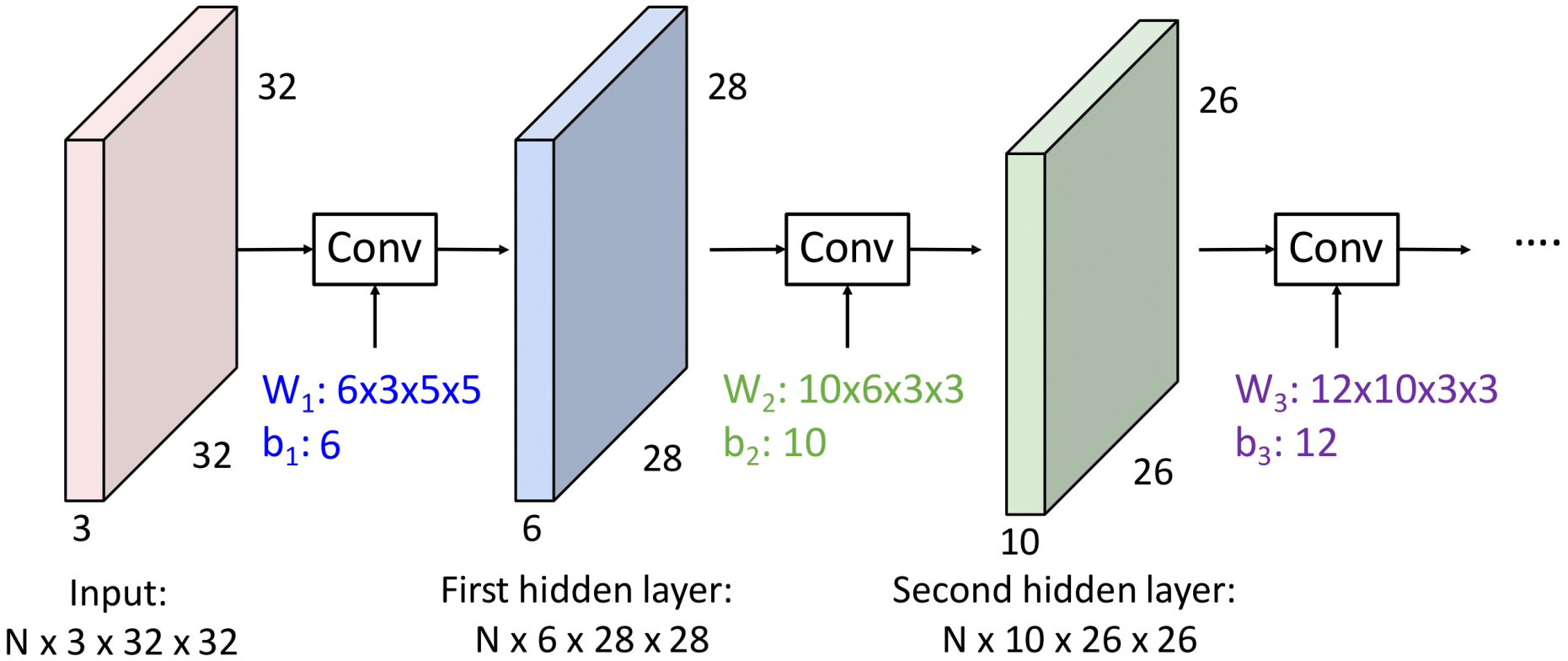
# Convolution Layer



# Convolution Layer

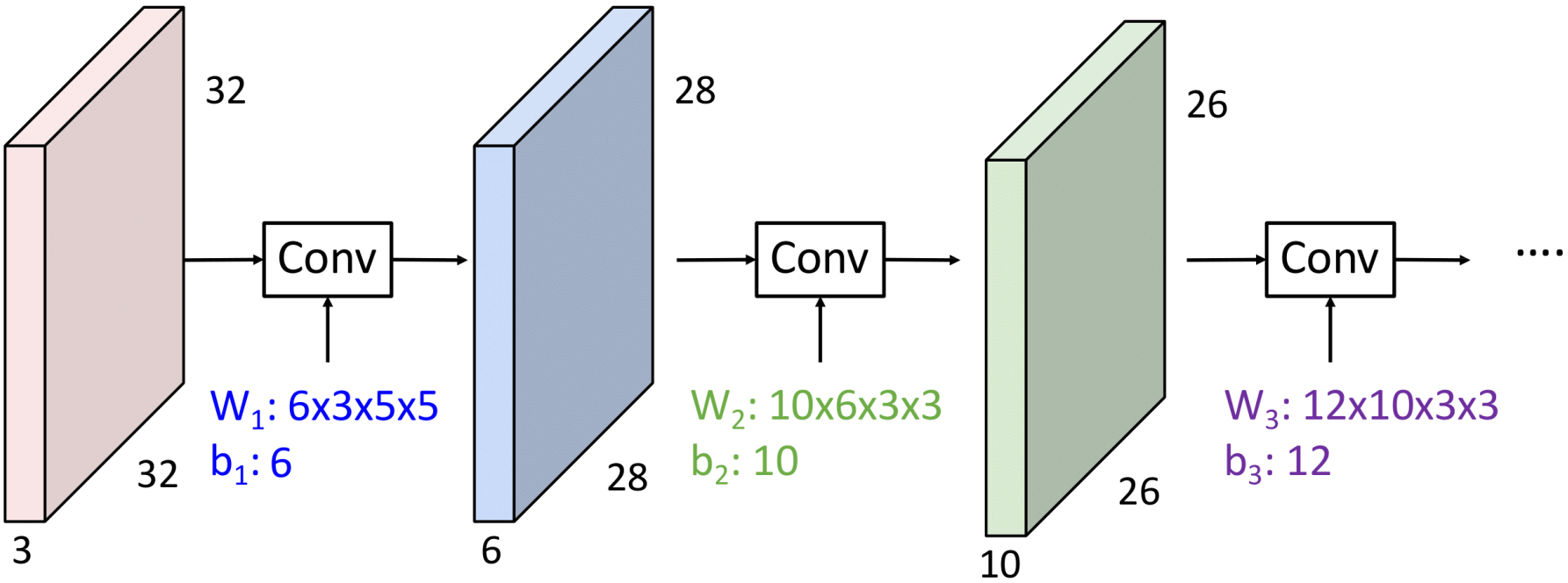


# Stacking convolutions



# Stacking convolutions

**Q**: What happens if we stack two convolution layers?




Input:
N x 3 x 32 x 32

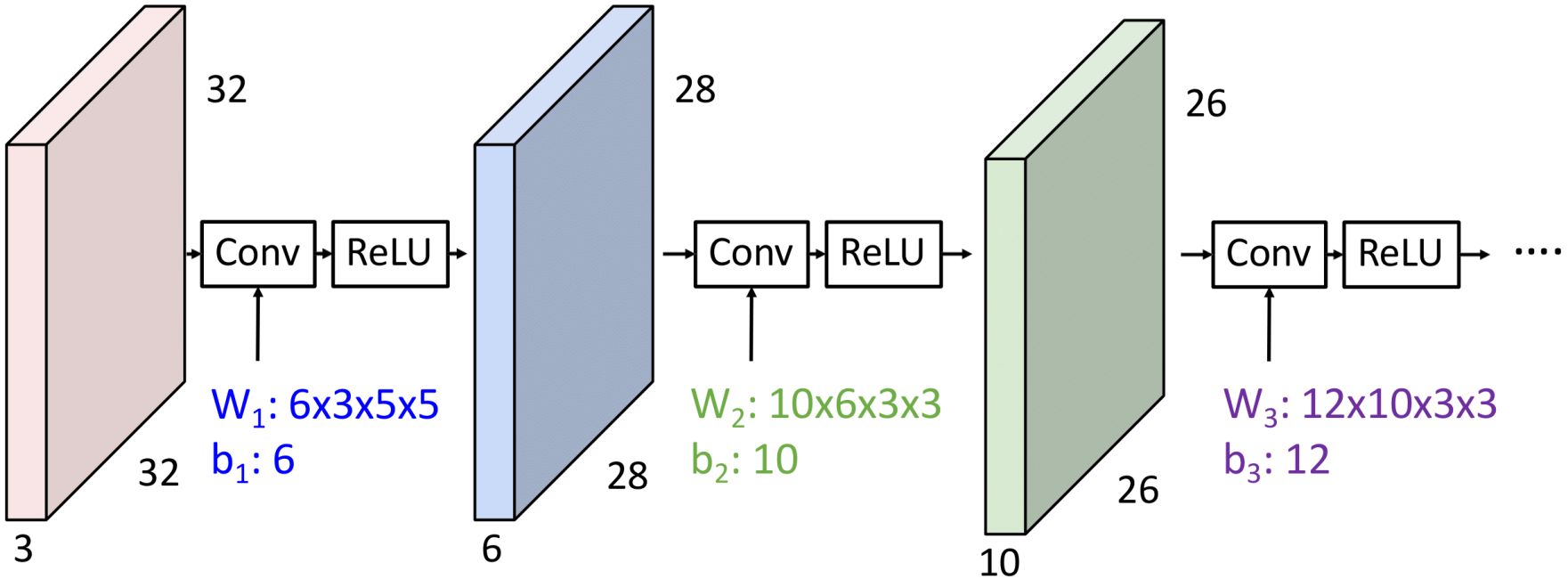
First hidden layer:
N x 6 x 28 x 28

Second hidden layer:
N x 10 x 26 x 26

# Stacking convolutions

**Q**: What happens if we stack two convolution layers? (Recall $y=W_2W_1x$ is a linear classifier)

**A**: We get another convolution!

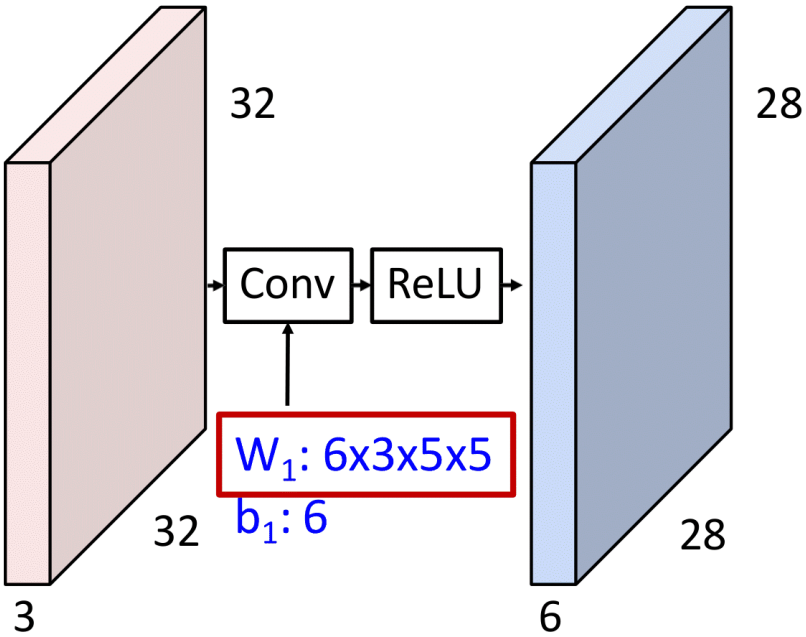



Input:
N x 3 x 32 x 32

First hidden layer:
N x 6 x 28 x 28

Second hidden layer:
N x 10 x 26 x 26

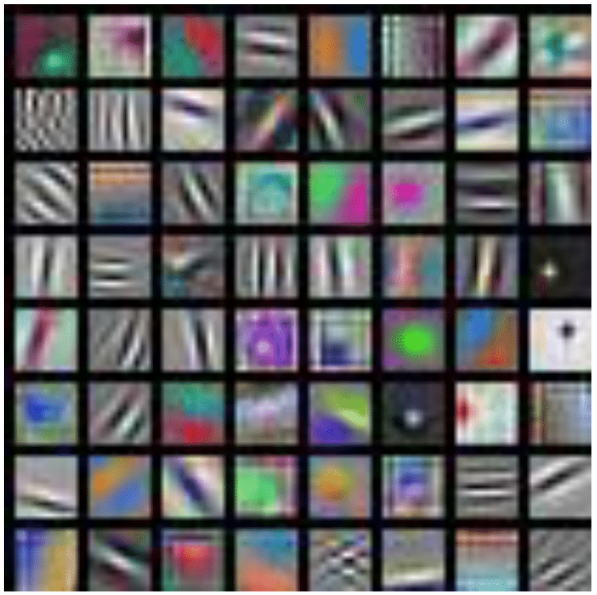
# What do convolutional filters learn?




Input:
N x 3 x 32 x 32

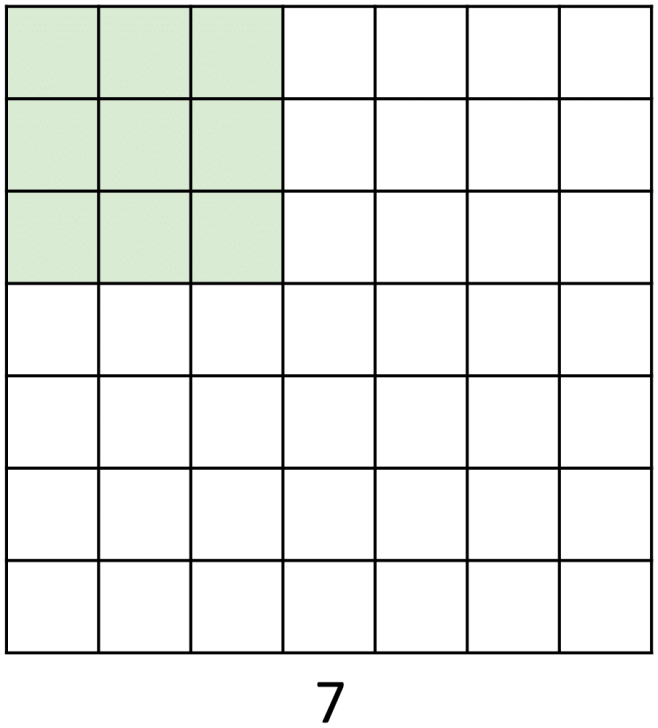
First hidden layer:
N x 6 x 28 x 28

First-layer conv filters: local image templates
(Often learns oriented edges, opposing colors)



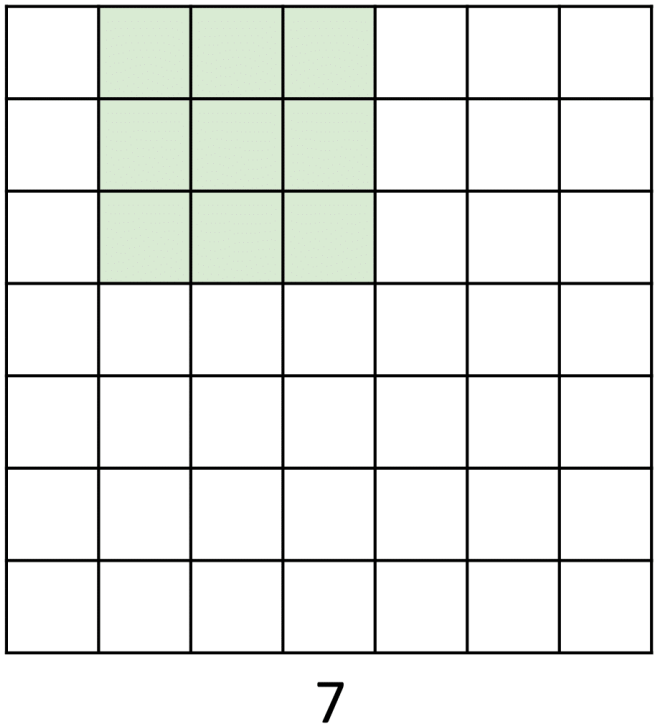
AlexNet: 64 filters, each 3x11x11

# A closer look at spatial dimensions



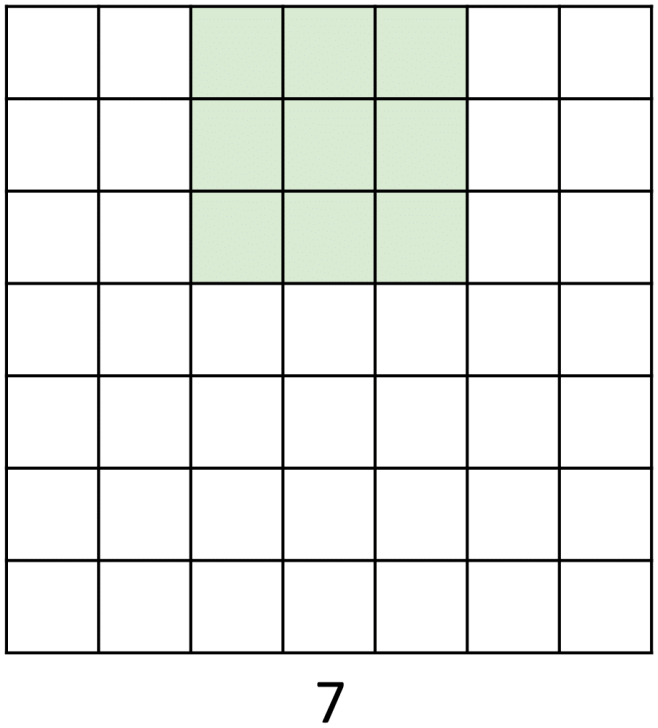

Input: 7x7
Filter: 3x3

# A closer look at spatial dimensions



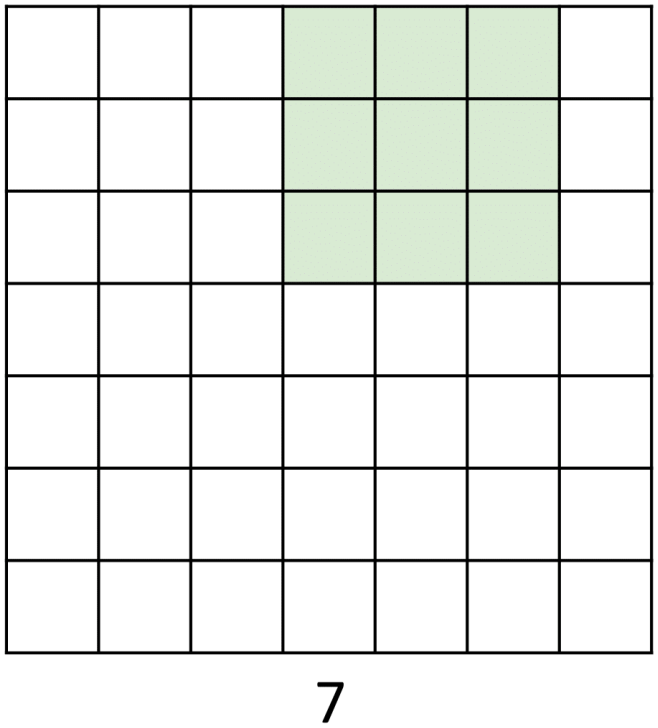

Input: 7x7
Filter: 3x3

# A closer look at spatial dimensions



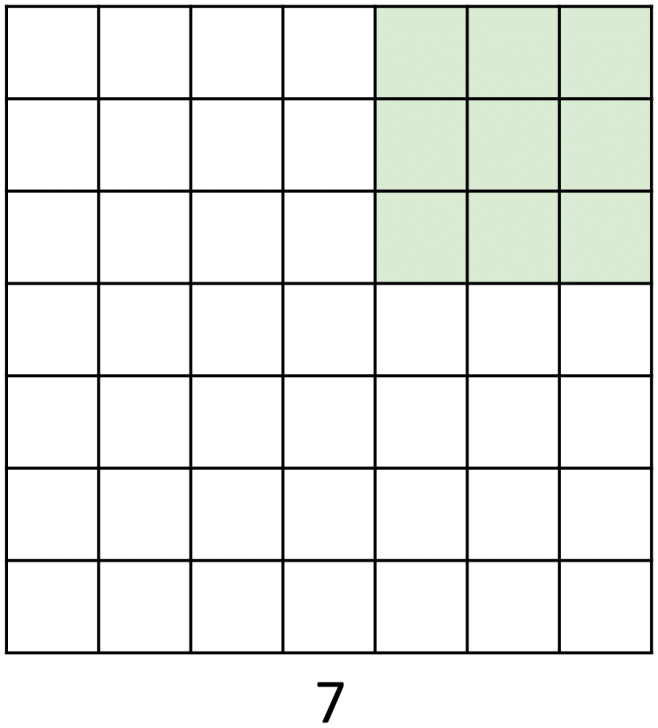

Input: 7x7
Filter: 3x3

# A closer look at spatial dimensions



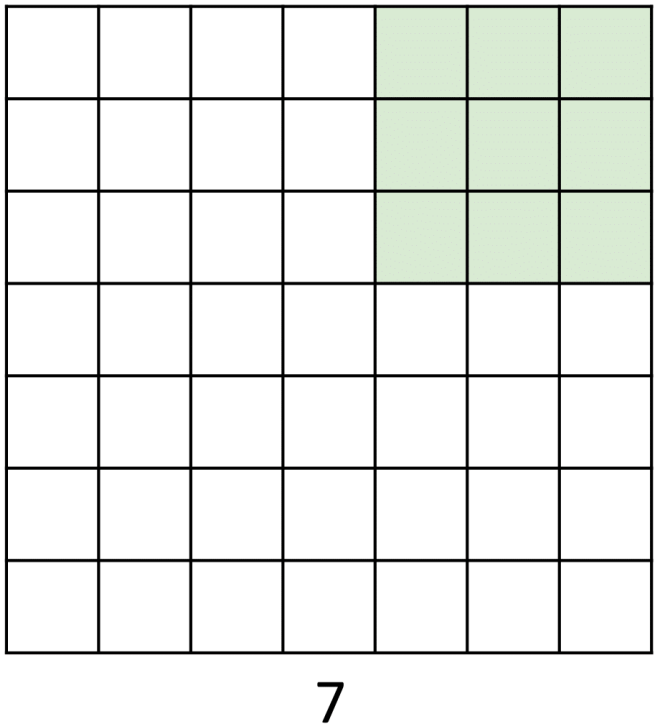

Input: 7x7
Filter: 3x3

# A closer look at spatial dimensions




Input: 7x7
Filter: 3x3
Output: 5x5

# A closer look at spatial dimensions




Input: 7x7
Filter: 3x3
Output: 5x5

In general:
Input: W
Filter: K
Output: W – K + 1

Problem: Feature maps "shrink" with each layer!

# A closer look at spatial dimensions

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | | | | | | | | 0 |
| 0 | | | | | | | | 0 |
| 0 | | | | | | | | 0 |
| 0 | | | | | | | | 0 |
| 0 | | | | | | | | 0 |
| 0 | | | | | | | | 0 |
| 0 | | | | | | | | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Input: 7x7
Filter: 3x3
Output: 5x5

In general:
Input: W
Filter: K
Output: W – K + 1

Problem: Feature maps "shrink" with each layer!

Solution: **padding**
Add zeros around the input

# A closer look at spatial dimensions

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | | | | | | | | 0 |
| 0 | | | | | | | | 0 |
| 0 | | | | | | | | 0 |
| 0 | | | | | | | | 0 |
| 0 | | | | | | | | 0 |
| 0 | | | | | | | | 0 |
| 0 | | | | | | | | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

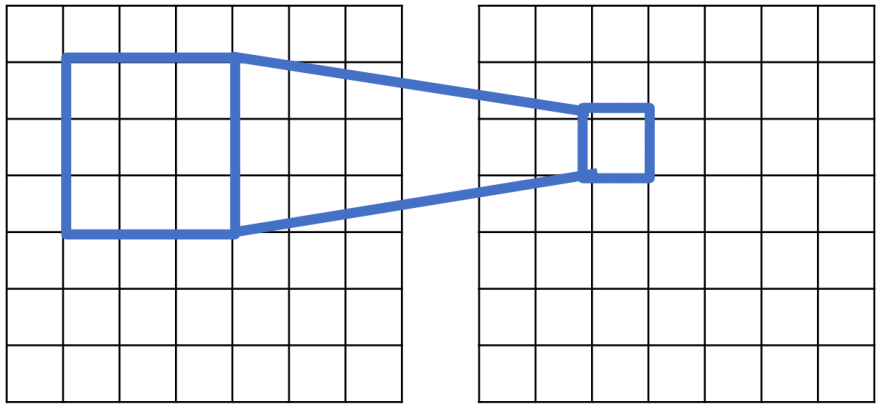
Input: 7x7
Filter: 3x3
Output: 5x5

In general:
Input: W
Filter: K
Padding: P
Output: W – K + 1 + 2P

Very common:
Set P = (K – 1) / 2 to make output have same size as input!

# Receptive Fields

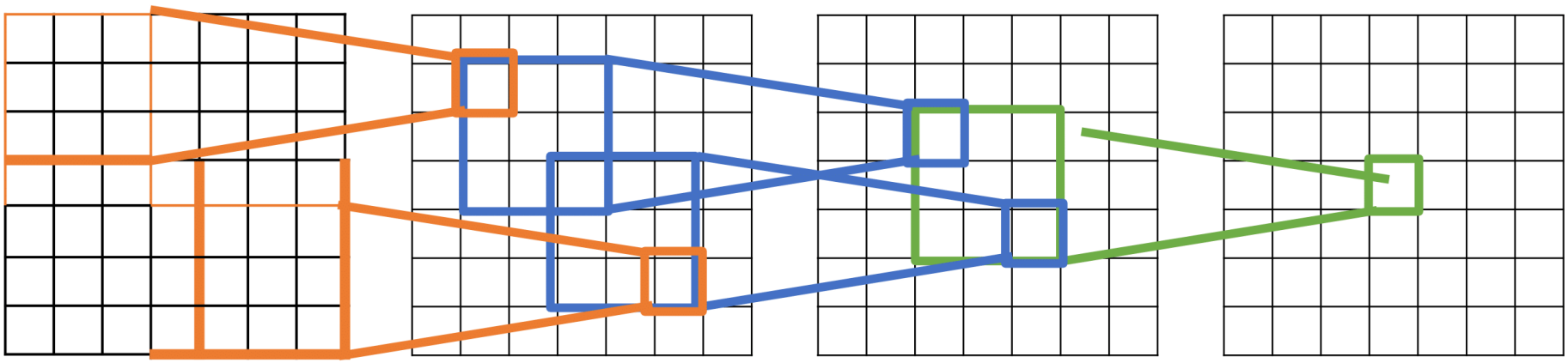
For convolution with kernel size K, each element in the output depends on a K x K **receptive field** in the input



Input Output

# Receptive Fields

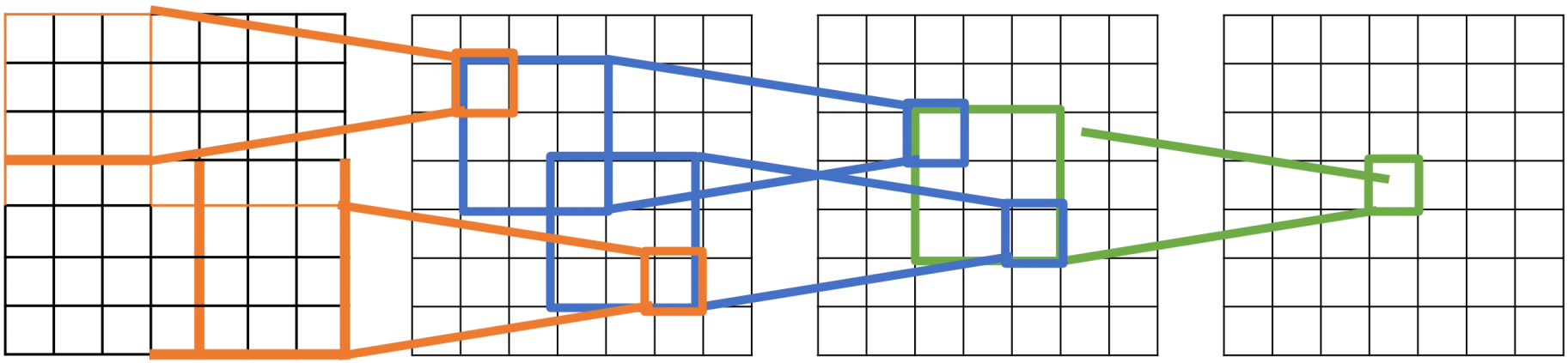
Each successive convolution adds K – 1 to the receptive field size
With L layers the receptive field size is 1 + L * (K – 1)




Be careful – ”receptive field in the input” vs “receptive field in the previous layer”
Hopefully clear from context!

# Receptive Fields

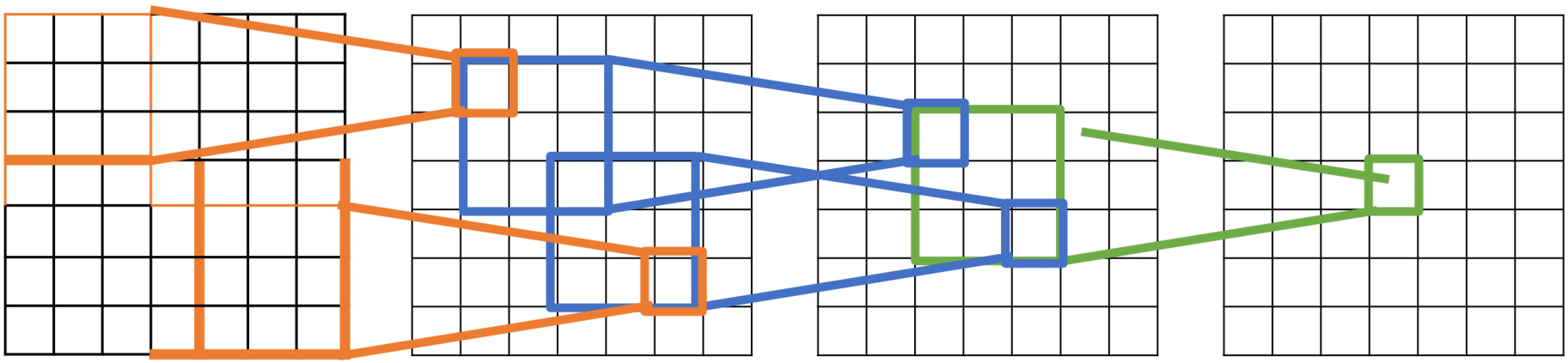
Each successive convolution adds K – 1 to the receptive field size
With L layers the receptive field size is 1 + L * (K – 1)




Problem: For large images we need many layers for each output to “see” the whole image image

# Receptive Fields

Each successive convolution adds K – 1 to the receptive field size
With L layers the receptive field size is 1 + L * (K – 1)



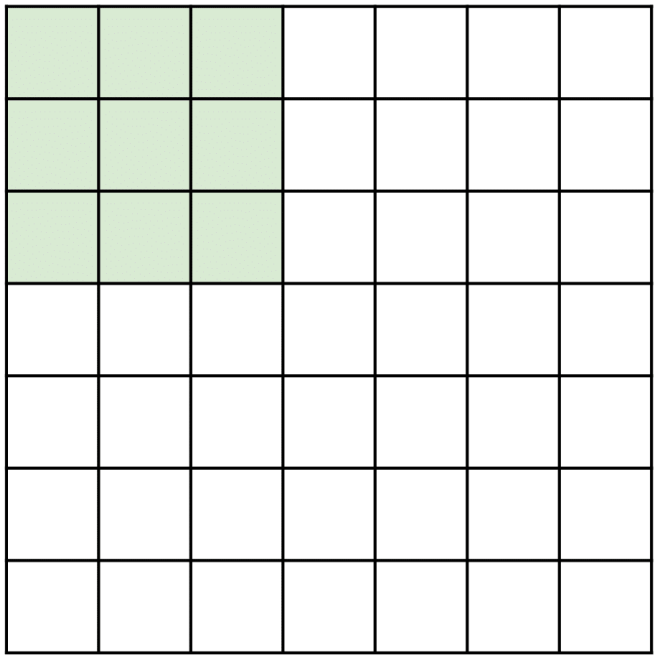
Input

Problem: For large images we need many layers for each output to “see” the whole image image

Solution: Downsample inside the network

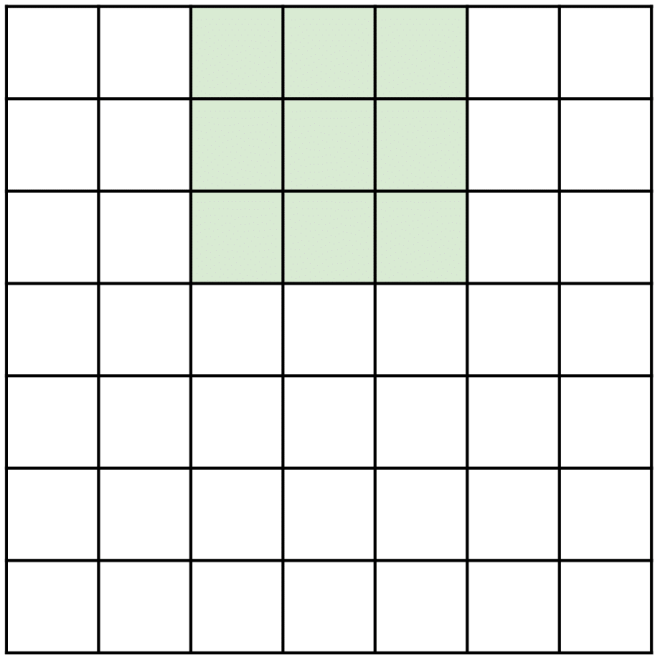
Output

# Strided Convolution



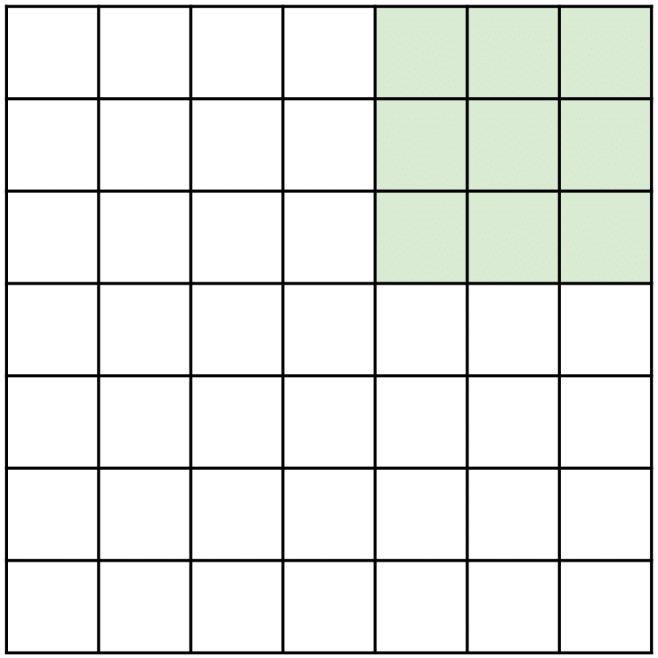
Input: 7x7
Filter: 3x3
Stride: 2

# Strided Convolution



Input: 7x7
Filter: 3x3
Stride: 2

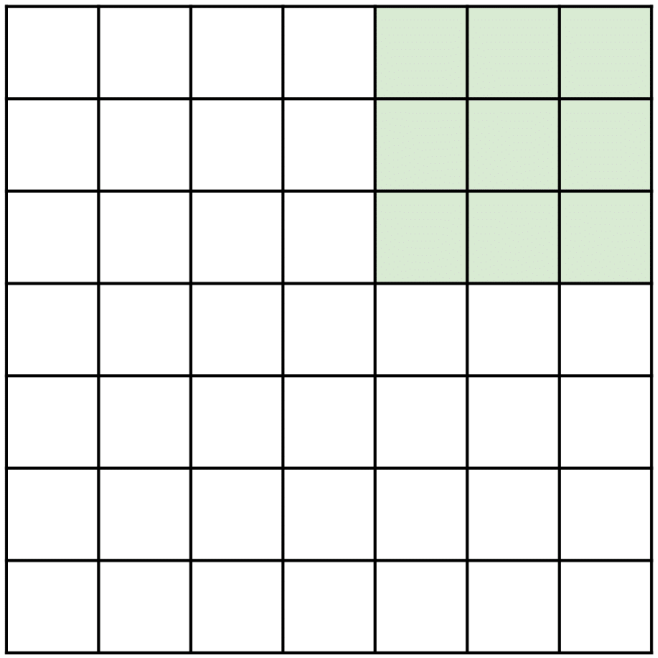
# Strided Convolution



Input: 7x7
Filter: 3x3
Stride: 2

Output: 3x3

# Strided Convolution



Input: 7x7
Filter: 3x3
Stride: 2

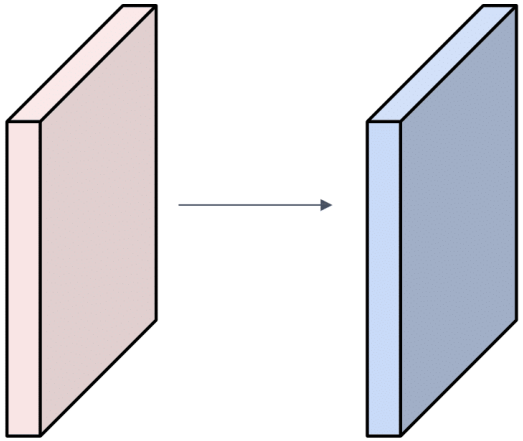
Output: 3x3

In general:
Input: W
Filter: K
Padding: P
Stride: S
Output: (W – K + 2P) / S + 1

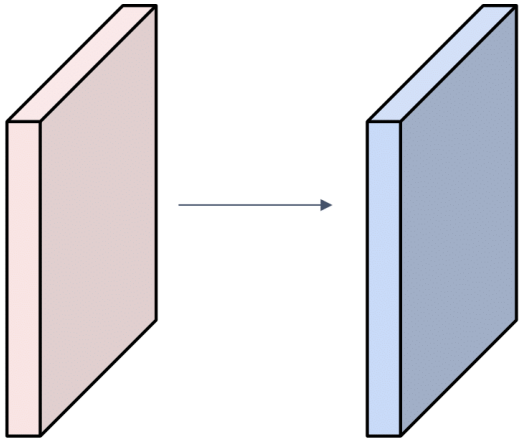
# Convolution example

Input volume: 3 x 32 x 32
10 5x5 filters with stride 1, pad 2

Output volume size: ?



# Convolution example



Input volume: 3 x 32 x 32

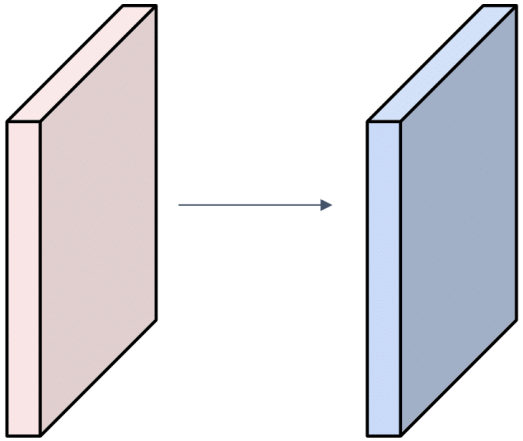
10 5x5 filters with stride 1, pad 2

Output volume size:

(32+2*2-5)/1+1 = 32 spatially, so

10 x 32 x 32

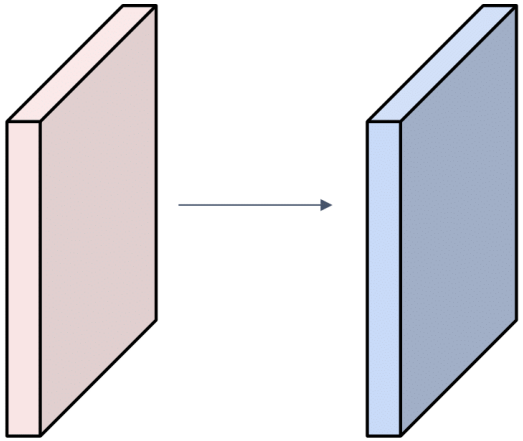
# Convolution example



Input volume: 3 x 32 x 32
10 5x5 filters with stride 1, pad 2

Output volume size: 10 x 32 x 32
Number of learnable parameters: ?

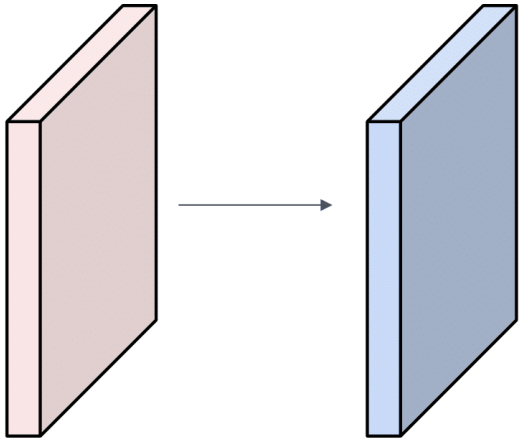
# Convolution example



Input volume: 3 x 32 x 32
10 5x5 filters with stride 1, pad 2

Output volume size: 10 x 32 x 32
Number of learnable parameters: **760**
Parameters per filter: 3*5*5 + 1 (for bias) = 76
10 filters, so total is 10 * 76 = **760**

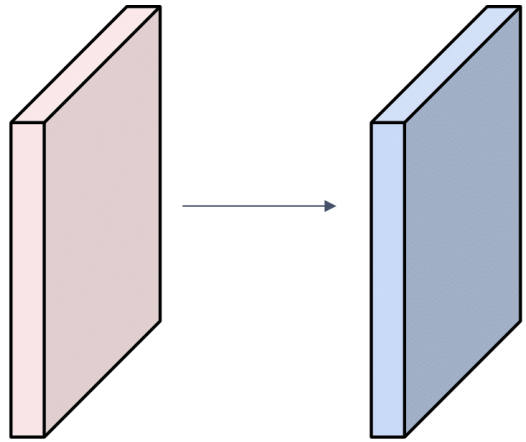
# Convolution example



Input volume: 3 x 32 x 32
10 5x5 filters with stride 1, pad 2

Output volume size: 10 x 32 x 32
Number of learnable parameters: 760
Number of multiply-add operations: ?

# Convolution example



Input volume: **3** x 32 x 32
10 **5x5** filters with stride 1, pad 2

Output volume size: **10 x 32 x 32**
Number of learnable parameters: 760
Number of multiply-add operations: **768,000**
**10*32*32** = 10,240 outputs; each output is the inner product of two **3**x**5x5** tensors (75 elems); total = 75*10240 = **768K**

# Convolution summary

**Input**: $C_{in}$ x H x W

**Hyperparameters**:

- **Kernel size**: $K_H$ x $K_W$
- **Number filters**: $C_{out}$
- **Padding**: P
- **Stride**: S

**Weight matrix**: $C_{out}$ x $C_{in}$ x $K_H$ x $K_W$

giving $C_{out}$ filters of size $C_{in}$ x $K_H$ x $K_W$

**Bias vector**: $C_{out}$

**Output size**: $C_{out}$ x H' x W' where:

- H' = (H – K + 2P) / S + 1
- W' = (W – K + 2P) / S + 1

# Convolution summary

**Input**: $C_{in}$ x H x W

**Hyperparameters**:

- **Kernel size**: $K_H$ x $K_W$
- **Number filters**: $C_{out}$
- **Padding**: P
- **Stride**: S

**Weight matrix**: $C_{out}$ x $C_{in}$ x $K_H$ x $K_W$
giving $C_{out}$ filters of size $C_{in}$ x $K_H$ x $K_W$

**Bias vector**: $C_{out}$

**Output size**: $C_{out}$ x H' x W' where:

- H' = (H – K + 2P) / S + 1
- W' = (W – K + 2P) / S + 1

Common settings:

$K_H = K_W$ (Small square filters)

P = (K – 1) / 2 ("Same" padding)

$C_{in}$, $C_{out}$ = 32, 64, 128, 256 (powers of 2)

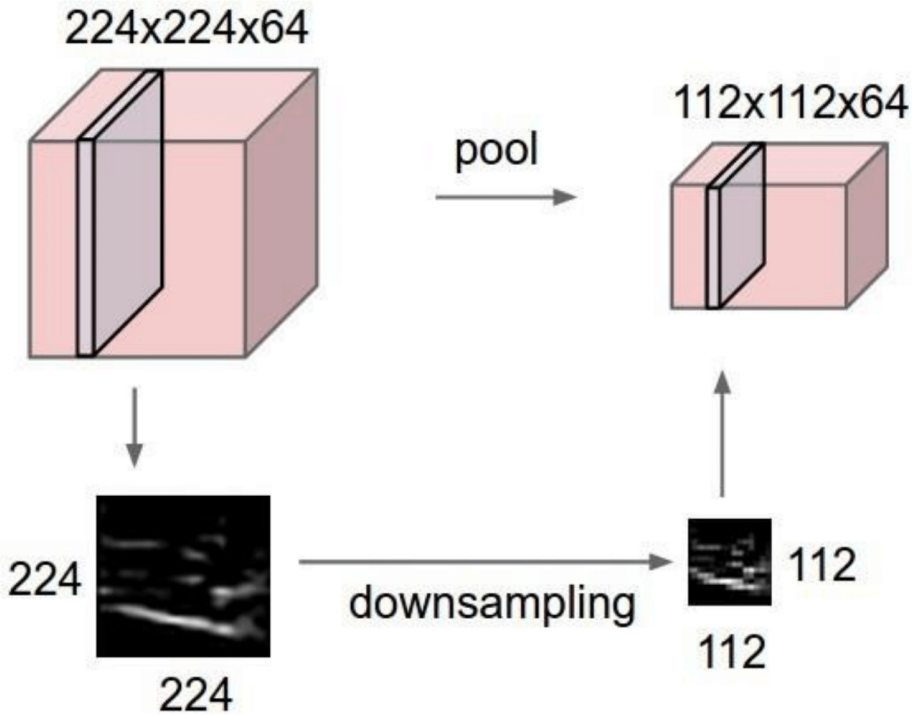
K = 3, P = 1, S = 1 (3x3 conv)

K = 5, P = 2, S = 1 (5x5 conv)

K = 1, P = 0, S = 1 (1x1 conv)

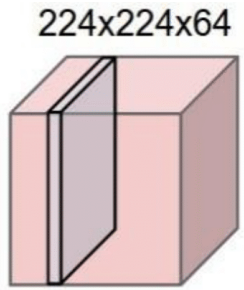
K = 3, P = 1, S = 2 (Downsample by 2)

# Pooling Layers: Another way to downsample




**Hyperparameters**:
Kernel Size
Stride
Pooling function

# Max Pooling




Single depth slice

x

| | | | |
|---|---|---|---|
| 1 | 1 | 2 | 4 |
| 5 | **6** | 7 | **8** |
| **3** | 2 | 1 | 0 |
| 1 | 2 | 3 | **4** |

y

Max pooling with 2x2 kernel size and stride 2

| | |
|---|---|
| 6 | 8 |
| 3 | 4 |

Introduces **invariance** to small spatial shifts

No learnable parameters!

# Pooling Summary

**Input**: C x H x W

**Hyperparameters**:

- Kernel size: K
- Stride: S
- Pooling function (max, avg)

**Output**: C x H' x W' where

- H' = (H – K) / S + 1
- W' = (W – K) / S + 1

**Learnable parameters**: None!

Common settings:

max, K = 2, S = 2

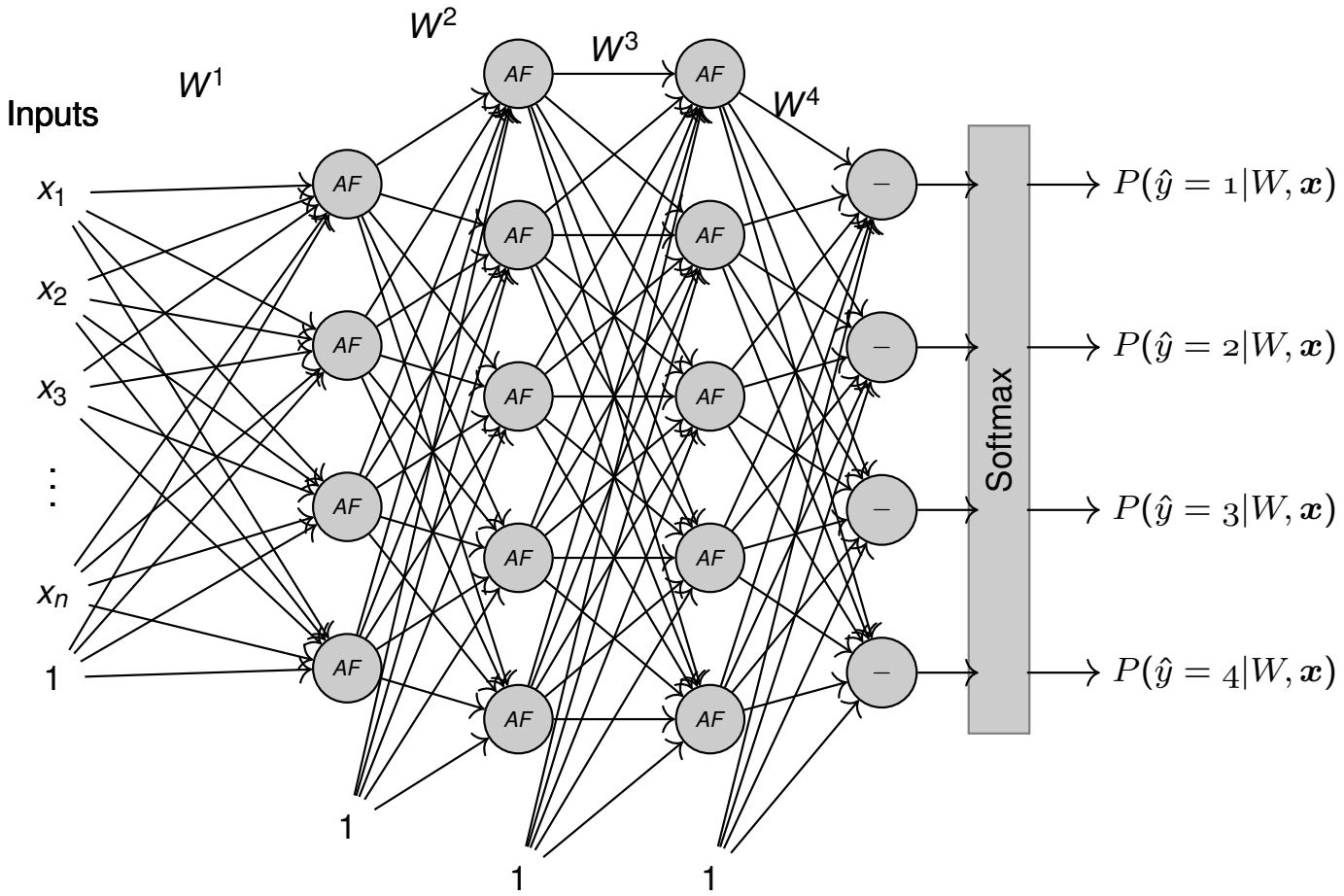
max, K = 3, S = 2 (AlexNet)

# Vanishing/Exploding Gradient

# The vanishing gradient problem for deep networks

- A particular problem with training deep networks
  - The gradient of the error with respect to weights is unstable

# Fully connected neural networks




$$\hat{y} = f^N(W^N f^{N-1}(W^{N-1} f^{N-2}(...f^2(W^2(f^1(W^1\boldsymbol{x}))))))$$

# The problem with training deep networks

- A multilayer perceptron is a nested function

$$\hat{y} = f^N(W^N f^{N-1}(W^{N-1} f^{N-2}(...f^2(W^2(f^1(W^1\boldsymbol{x})))))$$

  - $W^k$ is the weights matrix at the $k^{th}$ layer
  - $f^k$ is the activation function of the $k^{th}$ layer
- The error for $X$ can be written as

$$L = CE(f^N(W^N f^{N-1}(W^{N-1} f^{N-2}(...f^2(W^2(f^1(W^1\boldsymbol{x}))))), y)$$

  - $CE$ is cross entropy

# Training deep networks

- Let $\boldsymbol{z}^{\ell} = W^{\ell}\boldsymbol{o}^{\ell-1}$ and $\boldsymbol{o}^{\ell} = f^{\ell}(\boldsymbol{z}^{\ell})$

$$\frac{\partial L}{\partial \boldsymbol{o}^{\ell-1}} = ---$$

# Training deep networks

- Let $\boldsymbol{z}^{\ell} = W^{\ell}\boldsymbol{o}^{\ell-1}$ and $\boldsymbol{o}^{\ell} = f^{\ell}(\boldsymbol{z}^{\ell})$

$$\frac{\partial L}{\partial \boldsymbol{o}^{\ell-1}} = --\frac{\partial L}{\partial \boldsymbol{o}^{\ell}}$$

# Training deep networks

- Let $\boldsymbol{z}^{\ell} = W^{\ell}\boldsymbol{o}^{\ell-1}$ and $\boldsymbol{o}^{\ell} = f^{\ell}(\boldsymbol{z}^{\ell})$

$$\frac{\partial L}{\partial \boldsymbol{o}^{\ell-1}} = -\frac{\partial \boldsymbol{o}^{\ell}}{\partial \boldsymbol{z}^{\ell}}\frac{\partial L}{\partial \boldsymbol{o}^{\ell}}$$

# Training deep networks

- Let $\boldsymbol{z}^{\ell} = W^{\ell}\boldsymbol{o}^{\ell-1}$ and $\boldsymbol{o}^{\ell} = f^{\ell}(\boldsymbol{z}^{\ell})$

$$\frac{\partial L}{\partial \boldsymbol{o}^{\ell-1}} = \frac{\partial \boldsymbol{z}^{\ell}}{\partial \boldsymbol{o}^{\ell-1}} \frac{\partial \boldsymbol{o}^{\ell}}{\partial \boldsymbol{z}^{\ell}} \frac{\partial L}{\partial \boldsymbol{o}^{\ell}}$$

# Training deep networks

- Let $\boldsymbol{z}^{\ell} = W^{\ell}\boldsymbol{o}^{\ell-1}$ and $\boldsymbol{o}^{\ell} = f^{\ell}(\boldsymbol{z}^{\ell})$

$$\frac{\partial L}{\partial \boldsymbol{o}^{\ell-1}} = \frac{\partial \boldsymbol{z}^{\ell}}{\partial \boldsymbol{o}^{\ell-1}} \frac{\partial \boldsymbol{o}^{\ell}}{\partial \boldsymbol{z}^{\ell}} \frac{\partial L}{\partial \boldsymbol{o}^{\ell}}$$

$$\nabla_{\boldsymbol{o}^{\ell-1}} L = W^{\ell}.\nabla_{\boldsymbol{z}^{\ell}} f^{\ell}.\nabla_{\boldsymbol{o}^{\ell}} L$$

- Where
  - $\nabla_{\boldsymbol{x}} f$ is the jacobian matrix of $f(\boldsymbol{x})$ w.r.t $\boldsymbol{x}$

# Training deep networks

- For

$$L = CE(f^N(W^N f^{N-1}(W^{N-1} f^{N-2}(...f^1(W^1 \boldsymbol{x})))), y)$$

# Training deep networks

- For

$$L = CE(f^N(W^N f^{N-1}(W^{N-1} f^{N-2}(...f^1(W^1 \boldsymbol{x})))), y)$$

- We get

$$\nabla_{W^k} L = \boldsymbol{o}^{k-1}.\nabla_{\boldsymbol{z}^k} f^k.W^{k+1}.\nabla_{\boldsymbol{z}^{k+1}} f^{k+1}.....W^{N-1}.\nabla_{\boldsymbol{z}^{N-1}} f^{N-1}.W^N.\nabla_{\boldsymbol{z}^N} f^N.\nabla_{\hat{\boldsymbol{y}}} CE$$

$$\nabla_{W^k} L = \boldsymbol{o}^{k-1}.\nabla_{\boldsymbol{z}^k} f^k.\left(\prod_{i=k+1}^{N} W^i.\nabla_{\boldsymbol{z}^i} f^i\right).\nabla_{\hat{\boldsymbol{y}}} CE$$

- Where
  - $W^N$ is the gradient of the affine transforation $\boldsymbol{z}^n$ w.r.t the output of the $n-1^{th}$ layer ($\boldsymbol{o}^{n-1}$) of the network
  - $\nabla_{\boldsymbol{z}^n} f^n$ is gradient of activation function $f^n(.)$ w.r.t to affine transforation $\boldsymbol{z}^n$ in the $n^{th}$ layer of the network
  - $\boldsymbol{o}^{n-1}$ is the gradient of the affine transforation $\boldsymbol{z}^n$ w.r.t the weights of the $n^{th}$ layer ($W^n$) of the network
  - $\nabla_{\hat{\boldsymbol{y}}} CE$ is the gradient of cross entropy w.r.t the output of the network ($\hat{\boldsymbol{y}}$)

# Training deep networks

- For

$$L = CE(f^N(W^N f^{N-1}(W^{N-1} f^{N-2}(...f^1(W^1 \boldsymbol{x})))), y)$$

- We get

$$\nabla_{W^k} L = \boldsymbol{o}^{k-1}.\nabla_{\boldsymbol{z}^k} f^k.W^{k+1}.\nabla_{\boldsymbol{z}^{k+1}} f^{k+1}.....W^{N-1}.\nabla_{\boldsymbol{z}^{N-1}} f^{N-1}.W^N.\nabla_{\boldsymbol{z}^N} f^N.\nabla_{\hat{\boldsymbol{y}}} CE$$

$$\nabla_{W^k} L = \boldsymbol{o}^{k-1}.\nabla_{\boldsymbol{z}^k} f^k.\left(\prod_{i=k+1}^{N} W^i.\nabla_{\boldsymbol{z}^i} f^i\right).\nabla_{\hat{\boldsymbol{y}}} CE$$

- Where
  - $W^N$ is the gradient of the affine transforation $\boldsymbol{z}^n$ w.r.t the output of the $n-1^{th}$ layer ($\boldsymbol{o}^{n-1}$) of the network
  - $\nabla_{\boldsymbol{z}^n} f^n$ is gradient of activation function $f^n(.)$ w.r.t to affine transforation $\boldsymbol{z}^n$ in the $n^{th}$ layer of the network
  - $\boldsymbol{o}^{n-1}$ is the gradient of the affine transforation $\boldsymbol{z}^n$ w.r.t the weights of the $n^{th}$ layer ($W^n$) of the network
  - $\nabla_{\hat{\boldsymbol{y}}} CE$ is the gradient of cross entropy w.r.t the output of the network ($\hat{\boldsymbol{y}}$)

# Training deep networks

- For

$$L = CE(f^N(W^N f^{N-1}(W^{N-1} f^{N-2}(...f^1(W^1 \boldsymbol{x})))), y)$$

- We get

$$\nabla_{W^k} L = \boldsymbol{o}^{k-1} . \nabla_{\boldsymbol{z}^k} f^k . W^{k+1} . \nabla_{\boldsymbol{z}^{k+1}} f^{k+1} ..... W^{N-1} . \nabla_{\boldsymbol{z}^{N-1}} f^{N-1} . W^N . \nabla_{\boldsymbol{z}^N} f^N . \nabla_{\hat{\boldsymbol{y}}} CE$$

$$\nabla_{W^k} L = \boldsymbol{o}^{k-1} . \nabla_{\boldsymbol{z}^k} f^k . \left( \prod_{i=k+1}^{N} W^i . \nabla_{\boldsymbol{z}^i} f^i \right) . \nabla_{\hat{\boldsymbol{y}}} CE$$

- Where
  - $W^N$ is the gradient of the affine transforation $\boldsymbol{z}^n$ w.r.t the output of the $n-1^{th}$ layer ($\boldsymbol{o}^{n-1}$) of the network
  - $\nabla_{\boldsymbol{z}^n} f^n$ is gradient of activation function $f^n(.)$ w.r.t to affine transforation $\boldsymbol{z}^n$ in the $n^{th}$ layer of the network
  - $\boldsymbol{o}^{n-1}$ is the gradient of the affine transforation $\boldsymbol{z}^n$ w.r.t the weights of the $n^{th}$ layer ($W^n$) of the network
  - $\nabla_{\hat{\boldsymbol{y}}} CE$ is the gradient of cross entropy w.r.t the output of the network ($\hat{\boldsymbol{y}}$)

# Training deep networks

- For

$$L = CE(f^N(W^N f^{N-1}(W^{N-1} f^{N-2}(...f^1(W^1 \boldsymbol{x})))), y)$$

- We get

$$\nabla_{W^k} L = \boldsymbol{o}^{k-1} . \nabla_{\boldsymbol{z}^k} f^k . W^{k+1} . \nabla_{\boldsymbol{z}^{k+1}} f^{k+1} ..... W^{N-1} . \nabla_{\boldsymbol{z}^{N-1}} f^{N-1} . W^N . \nabla_{\boldsymbol{z}^N} f^N . \nabla_{\hat{\boldsymbol{y}}} CE$$

$$\nabla_{W^k} L = \boldsymbol{o}^{k-1} . \nabla_{\boldsymbol{z}^k} f^k . \left( \prod_{i=k+1}^{N} W^i . \nabla_{\boldsymbol{z}^i} f^i \right) . \nabla_{\hat{\boldsymbol{y}}} CE$$

- Repeated multiplication by the weights matrix and jacobian matrices of activation functions will result in exploding or vanishing gradients
  - Depends on the spectral norm (largest singular value) of jacobian matrices
- The gradients in the lower/earlier layers can explode or vanish
  - Resulting in insignificant or unstable gradient descent updates
  - Problem gets worse as network depth increases

# References


- Deep Learning, Ian Goodfellow, MIT Press, Ch.6, 7 ,and 8