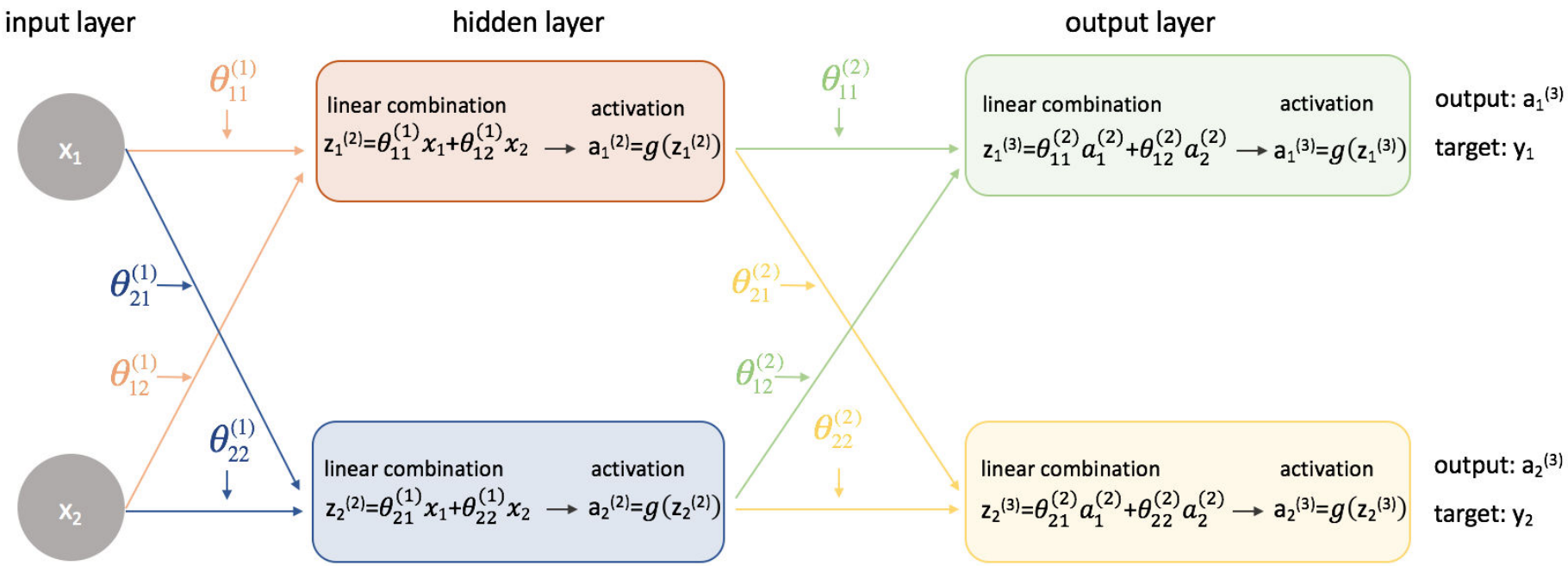
Gradient Descent

Source: https://www.jeremyjordan.me/neural-networks-training/


input layer
hidden layer
output layer
x_1
x_2
$\theta_{11}^{(1)}$
$\theta_{21}^{(1)}$
$\theta_{12}^{(1)}$
$\theta_{22}^{(1)}$
linear combination
activation
$z_1^{(2)}=\theta_{11}^{(1)}x_1+\theta_{12}^{(1)}x_2 \rightarrow a_1^{(2)}=g(z_1^{(2)})$
$z_2^{(2)}=\theta_{21}^{(1)}x_1+\theta_{22}^{(1)}x_2 \rightarrow a_2^{(2)}=g(z_2^{(2)})$
$\theta_{11}^{(2)}$
$\theta_{21}^{(2)}$
$\theta_{12}^{(2)}$
$\theta_{22}^{(2)}$
linear combination
activation
$z_1^{(3)}=\theta_{11}^{(2)}a_1^{(2)}+\theta_{12}^{(2)}a_2^{(2)} \rightarrow a_1^{(3)}=g(z_1^{(3)})$
$z_2^{(3)}=\theta_{21}^{(2)}a_1^{(2)}+\theta_{22}^{(2)}a_2^{(2)} \rightarrow a_2^{(3)}=g(z_2^{(3)})$
output: $a_1^{(3)}$
target: y_1
output: $a_2^{(3)}$
target: y_2

Layer 2 Parameters

$$\frac{\partial J(\theta)}{\partial \theta_{11}^{(2)}} = \left(\frac{\partial J(\theta)}{\partial a_1^{(3)}}\right)\left(\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}}\right)\left(\frac{\partial z_1^{(3)}}{\partial \theta_{11}^{(2)}}\right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{12}^{(2)}} = \left(\frac{\partial J(\theta)}{\partial a_1^{(3)}}\right)\left(\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}}\right)\left(\frac{\partial z_1^{(3)}}{\partial \theta_{12}^{(2)}}\right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{21}^{(2)}} = \left(\frac{\partial J(\theta)}{\partial a_2^{(3)}}\right)\left(\frac{\partial a_2^{(3)}}{\partial z_2^{(3)}}\right)\left(\frac{\partial z_2^{(3)}}{\partial \theta_{21}^{(2)}}\right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{22}^{(2)}} = \left(\frac{\partial J(\theta)}{\partial a_2^{(3)}}\right)\left(\frac{\partial a_2^{(3)}}{\partial z_2^{(3)}}\right)\left(\frac{\partial z_2^{(3)}}{\partial \theta_{22}^{(2)}}\right)$$

Layer 1 Parameters

$$\frac{\partial J(\theta)}{\partial \theta_{11}^{(1)}} = \left(\frac{\partial J(\theta)}{\partial a_1^{(3)}}\right)\left(\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}}\right)\left(\frac{\partial z_1^{(3)}}{\partial a_1^{(2)}}\right)\left(\frac{\partial a_1^{(2)}}{\partial z_1^{(2)}}\right)\left(\frac{\partial z_1^{(2)}}{\partial \theta_{11}^{(1)}}\right) + \left(\frac{\partial J(\theta)}{\partial a_2^{(3)}}\right)\left(\frac{\partial a_2^{(3)}}{\partial z_2^{(3)}}\right)\left(\frac{\partial z_2^{(3)}}{\partial a_1^{(2)}}\right)\left(\frac{\partial a_1^{(2)}}{\partial z_1^{(2)}}\right)\left(\frac{\partial z_1^{(2)}}{\partial \theta_{11}^{(1)}}\right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{12}^{(1)}} = \left(\frac{\partial J(\theta)}{\partial a_1^{(3)}}\right)\left(\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}}\right)\left(\frac{\partial z_1^{(3)}}{\partial a_1^{(2)}}\right)\left(\frac{\partial a_1^{(2)}}{\partial z_1^{(2)}}\right)\left(\frac{\partial z_1^{(2)}}{\partial \theta_{12}^{(1)}}\right) + \left(\frac{\partial J(\theta)}{\partial a_2^{(3)}}\right)\left(\frac{\partial a_2^{(3)}}{\partial z_2^{(3)}}\right)\left(\frac{\partial z_2^{(3)}}{\partial a_1^{(2)}}\right)\left(\frac{\partial a_1^{(2)}}{\partial z_1^{(2)}}\right)\left(\frac{\partial z_1^{(2)}}{\partial \theta_{12}^{(1)}}\right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{21}^{(1)}} = \left(\frac{\partial J(\theta)}{\partial a_1^{(3)}}\right)\left(\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}}\right)\left(\frac{\partial z_1^{(3)}}{\partial a_2^{(2)}}\right)\left(\frac{\partial a_2^{(2)}}{\partial z_2^{(2)}}\right)\left(\frac{\partial z_2^{(2)}}{\partial \theta_{21}^{(1)}}\right) + \left(\frac{\partial J(\theta)}{\partial a_2^{(3)}}\right)\left(\frac{\partial a_2^{(3)}}{\partial z_2^{(3)}}\right)\left(\frac{\partial z_2^{(3)}}{\partial a_2^{(2)}}\right)\left(\frac{\partial a_2^{(2)}}{\partial z_2^{(2)}}\right)\left(\frac{\partial z_2^{(2)}}{\partial \theta_{21}^{(1)}}\right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{22}^{(1)}} = \left(\frac{\partial J(\theta)}{\partial a_1^{(3)}}\right)\left(\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}}\right)\left(\frac{\partial z_1^{(3)}}{\partial a_2^{(2)}}\right)\left(\frac{\partial a_2^{(2)}}{\partial z_2^{(2)}}\right)\left(\frac{\partial z_2^{(2)}}{\partial \theta_{22}^{(1)}}\right) + \left(\frac{\partial J(\theta)}{\partial a_2^{(3)}}\right)\left(\frac{\partial a_2^{(3)}}{\partial z_2^{(3)}}\right)\left(\frac{\partial z_2^{(3)}}{\partial a_2^{(2)}}\right)\left(\frac{\partial a_2^{(2)}}{\partial z_2^{(2)}}\right)\left(\frac{\partial z_2^{(2)}}{\partial \theta_{22}^{(1)}}\right)$$

Layer 2 Parameters

$$\frac{\partial J(\theta)}{\partial \theta_{11}^{(2)}} = \left(\frac{\partial J(\theta)}{\partial a_1^{(3)}}\right)\left(\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}}\right)\left(\frac{\partial z_1^{(3)}}{\partial \theta_{11}^{(2)}}\right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{12}^{(2)}} = \left(\frac{\partial J(\theta)}{\partial a_1^{(3)}}\right)\left(\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}}\right)\left(\frac{\partial z_1^{(3)}}{\partial \theta_{12}^{(2)}}\right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{21}^{(2)}} = \left(\frac{\partial J(\theta)}{\partial a_2^{(3)}}\right)\left(\frac{\partial a_2^{(3)}}{\partial z_2^{(3)}}\right)\left(\frac{\partial z_2^{(3)}}{\partial \theta_{21}^{(2)}}\right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{22}^{(2)}} = \left(\frac{\partial J(\theta)}{\partial a_2^{(3)}}\right)\left(\frac{\partial a_2^{(3)}}{\partial z_2^{(3)}}\right)\left(\frac{\partial z_2^{(3)}}{\partial \theta_{22}^{(2)}}\right)$$

Layer 2 Parameters

$$\frac{\partial J(\theta)}{\partial \theta_{11}^{(2)}} = \left(\frac{\partial J(\theta)}{\partial \mathrm{a}_1^{(3)}}\right)\left(\frac{\partial \mathrm{a}_1^{(3)}}{\partial z_1^{(3)}}\right)\left(\frac{\partial z_1^{(3)}}{\partial \theta_{11}^{(2)}}\right) \qquad \delta_i^{(3)} = \frac{1}{m}\left(y_i - a_i^{(3)}\right) f'\left(a^{(3)}\right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{12}^{(2)}} = \left(\frac{\partial J(\theta)}{\partial \mathrm{a}_1^{(3)}}\right)\left(\frac{\partial \mathrm{a}_1^{(3)}}{\partial z_1^{(3)}}\right)\left(\frac{\partial z_1^{(3)}}{\partial \theta_{12}^{(2)}}\right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{21}^{(2)}} = \left(\frac{\partial J(\theta)}{\partial \mathrm{a}_2^{(3)}}\right)\left(\frac{\partial \mathrm{a}_2^{(3)}}{\partial z_2^{(3)}}\right)\left(\frac{\partial z_2^{(3)}}{\partial \theta_{21}^{(2)}}\right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{22}^{(2)}} = \left(\frac{\partial J(\theta)}{\partial \mathrm{a}_2^{(3)}}\right)\left(\frac{\partial \mathrm{a}_2^{(3)}}{\partial z_2^{(3)}}\right)\left(\frac{\partial z_2^{(3)}}{\partial \theta_{22}^{(2)}}\right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{11}^{(2)}} = \delta_1^{(3)} \left(\frac{\partial z_1^{(3)}}{\partial \theta_{11}^{(2)}} \right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{12}^{(2)}} = \delta_1^{(3)} \left(\frac{\partial z_1^{(3)}}{\partial \theta_{12}^{(2)}} \right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{21}^{(2)}} = \delta_2^{(3)} \left(\frac{\partial z_2^{(3)}}{\partial \theta_{21}^{(2)}} \right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{22}^{(2)}} = \delta_2^{(3)} \left(\frac{\partial z_2^{(3)}}{\partial \theta_{22}^{(2)}} \right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{11}^{(2)}} = \delta_1^{(3)} a_1^{(2)}$$

$$\frac{\partial J(\theta)}{\partial \theta_{12}^{(2)}} = \delta_1^{(3)} a_2^{(2)}$$

$$\frac{\partial J(\theta)}{\partial \theta_{21}^{(2)}} = \delta_2^{(3)} a_1^{(2)}$$

$$\frac{\partial J(\theta)}{\partial \theta_{22}^{(2)}} = \delta_2^{(3)} a_2^{(2)}$$

$$\frac{\partial J(\theta)}{\partial \theta_{11}^{(2)}} = \delta_1^{(3)} a_1^{(2)}$$

$$\frac{\partial J(\theta)}{\partial \theta_{12}^{(2)}} = \delta_1^{(3)} a_2^{(2)}$$

$$\frac{\partial J(\theta)}{\partial \theta_{21}^{(2)}} = \delta_2^{(3)} a_1^{(2)}$$

$$\frac{\partial J(\theta)}{\partial \theta_{22}^{(2)}} = \delta_2^{(3)} a_2^{(2)}$$

$$\frac{\partial J(\theta)}{\partial \theta_{jk}^{(2)}} = \delta_j^{(3)} a_k^{(2)}$$

$$\frac{\partial J(\theta)}{\partial \theta_{11}^{(2)}} = \delta_1^{(3)} a_1^{(2)}$$

$$\frac{\partial J(\theta)}{\partial \theta_{12}^{(2)}} = \delta_1^{(3)} a_2^{(2)}$$

$$\frac{\partial J(\theta)}{\partial \theta_{21}^{(2)}} = \delta_2^{(3)} a_1^{(2)}$$

$$\frac{\partial J(\theta)}{\partial \theta_{22}^{(2)}} = \delta_2^{(3)} a_2^{(2)}$$

$$\frac{\partial J(\theta)}{\partial \theta_{jk}^{(2)}} = \delta_j^{(3)} a_k^{(2)}$$

$$\delta^{(3)} = \begin{bmatrix} y_1 - a_1^{(3)} \\ y_2 - a_2^{(3)} \\ \cdots \\ y_j - a_j^{(3)} \end{bmatrix} f'\left(a^{(3)}\right)$$

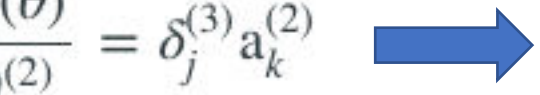
$$\frac{\partial J(\theta)}{\partial \theta_{11}^{(2)}} = \delta_1^{(3)} a_1^{(2)}$$

$$\frac{\partial J(\theta)}{\partial \theta_{12}^{(2)}} = \delta_1^{(3)} a_2^{(2)}$$

$$\frac{\partial J(\theta)}{\partial \theta_{21}^{(2)}} = \delta_2^{(3)} a_1^{(2)}$$

$$\frac{\partial J(\theta)}{\partial \theta_{22}^{(2)}} = \delta_2^{(3)} a_2^{(2)}$$

$$\frac{\partial J(\theta)}{\partial \theta_{jk}^{(2)}} = \delta_j^{(3)} a_k^{(2)}$$



$$\frac{\partial J(\theta)}{\partial \theta_{ij}^{(2)}} = \begin{bmatrix} \delta_1^{(3)} \\ \delta_2^{(3)} \end{bmatrix} \begin{bmatrix} a_1^{(2)} & a_2^{(2)} \end{bmatrix} = \begin{bmatrix} \delta_1^{(3)} a_1^{(2)} & \delta_1^{(3)} a_2^{(2)} \\ \delta_2^{(3)} a_1^{(2)} & \delta_2^{(3)} a_2^{(2)} \end{bmatrix}$$

$$\delta^{(3)} = \begin{bmatrix} y_1 - a_1^{(3)} \\ y_2 - a_2^{(3)} \\ \cdots \\ y_j - a_j^{(3)} \end{bmatrix} f'\left(a^{(3)}\right)$$

Layer 2

"error" term

$\delta^{(3)}$

$(y_i - a_i^{(3)})$ $f'(a^{(3)})$ input

$$\frac{\partial J(\theta)}{\partial \theta_{11}^{(2)}} = \left(\frac{\partial J(\theta)}{\partial a_1^{(3)}}\right)\left(\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}}\right)\left(\frac{\partial z_1^{(3)}}{\partial \theta_{11}^{(2)}}\right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{12}^{(2)}} = \left(\frac{\partial J(\theta)}{\partial a_1^{(3)}}\right)\left(\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}}\right)\left(\frac{\partial z_1^{(3)}}{\partial \theta_{12}^{(2)}}\right)$$

first output neuron

$$\frac{\partial J(\theta)}{\partial \theta_{21}^{(2)}} = \left(\frac{\partial J(\theta)}{\partial a_2^{(3)}}\right)\left(\frac{\partial a_2^{(3)}}{\partial z_2^{(3)}}\right)\left(\frac{\partial z_2^{(3)}}{\partial \theta_{21}^{(2)}}\right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{22}^{(2)}} = \left(\frac{\partial J(\theta)}{\partial a_2^{(3)}}\right)\left(\frac{\partial a_2^{(3)}}{\partial z_2^{(3)}}\right)\left(\frac{\partial z_2^{(3)}}{\partial \theta_{22}^{(2)}}\right)$$

second output neuron

$$\delta_i^{(3)} = \frac{1}{m}\left(y_i - a_i^{(3)}\right) f'\left(a^{(3)}\right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{ij}^{(2)}} = \begin{bmatrix} \delta_1^{(3)} \\ \delta_2^{(3)} \end{bmatrix} \begin{bmatrix} a_1^{(2)} & a_2^{(2)} \end{bmatrix} = \begin{bmatrix} \delta_1^{(3)} a_1^{(2)} & \delta_1^{(3)} a_2^{(2)} \\ \delta_2^{(3)} a_1^{(2)} & \delta_2^{(3)} a_2^{(2)} \end{bmatrix}$$

Layer 1

$$\frac{\partial J(\theta)}{\partial \theta_{11}^{(1)}} = \left(\frac{\partial J(\theta)}{\partial a_1^{(3)}}\right)\left(\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}}\right)\left(\frac{\partial z_1^{(3)}}{\partial a_1^{(2)}}\right)\left(\frac{\partial a_1^{(2)}}{\partial z_1^{(2)}}\right)\left(\frac{\partial z_1^{(2)}}{\partial \theta_{11}^{(1)}}\right) + \left(\frac{\partial J(\theta)}{\partial a_2^{(3)}}\right)\left(\frac{\partial a_2^{(3)}}{\partial z_2^{(3)}}\right)\left(\frac{\partial z_2^{(3)}}{\partial a_1^{(2)}}\right)\left(\frac{\partial a_1^{(2)}}{\partial z_1^{(2)}}\right)\left(\frac{\partial z_1^{(2)}}{\partial \theta_{11}^{(1)}}\right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{12}^{(1)}} = \left(\frac{\partial J(\theta)}{\partial a_1^{(3)}}\right)\left(\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}}\right)\left(\frac{\partial z_1^{(3)}}{\partial a_1^{(2)}}\right)\left(\frac{\partial a_1^{(2)}}{\partial z_1^{(2)}}\right)\left(\frac{\partial z_1^{(2)}}{\partial \theta_{12}^{(1)}}\right) + \left(\frac{\partial J(\theta)}{\partial a_2^{(3)}}\right)\left(\frac{\partial a_2^{(3)}}{\partial z_2^{(3)}}\right)\left(\frac{\partial z_2^{(3)}}{\partial a_1^{(2)}}\right)\left(\frac{\partial a_1^{(2)}}{\partial z_1^{(2)}}\right)\left(\frac{\partial z_1^{(2)}}{\partial \theta_{12}^{(1)}}\right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{21}^{(1)}} = \left(\frac{\partial J(\theta)}{\partial a_1^{(3)}}\right)\left(\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}}\right)\left(\frac{\partial z_1^{(3)}}{\partial a_2^{(2)}}\right)\left(\frac{\partial a_2^{(2)}}{\partial z_2^{(2)}}\right)\left(\frac{\partial z_2^{(2)}}{\partial \theta_{21}^{(1)}}\right) + \left(\frac{\partial J(\theta)}{\partial a_2^{(3)}}\right)\left(\frac{\partial a_2^{(3)}}{\partial z_2^{(3)}}\right)\left(\frac{\partial z_2^{(3)}}{\partial a_2^{(2)}}\right)\left(\frac{\partial a_2^{(2)}}{\partial z_2^{(2)}}\right)\left(\frac{\partial z_2^{(2)}}{\partial \theta_{21}^{(1)}}\right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{22}^{(1)}} = \left(\frac{\partial J(\theta)}{\partial a_1^{(3)}}\right)\left(\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}}\right)\left(\frac{\partial z_1^{(3)}}{\partial a_2^{(2)}}\right)\left(\frac{\partial a_2^{(2)}}{\partial z_2^{(2)}}\right)\left(\frac{\partial z_2^{(2)}}{\partial \theta_{22}^{(1)}}\right) + \left(\frac{\partial J(\theta)}{\partial a_2^{(3)}}\right)\left(\frac{\partial a_2^{(3)}}{\partial z_2^{(3)}}\right)\left(\frac{\partial z_2^{(3)}}{\partial a_2^{(2)}}\right)\left(\frac{\partial a_2^{(2)}}{\partial z_2^{(2)}}\right)\left(\frac{\partial z_2^{(2)}}{\partial \theta_{22}^{(1)}}\right)$$

$\delta^{(2)}$ = "proportional error" = $\delta_1^{(3)}$ $\theta^{(2)}$ $f'(a^{(2)})$ input

$\delta^{(2)}$ = "proportional error" = $\delta_2^{(3)}$ $\theta^{(2)}$ $f'(a^{(2)})$ input

derivative chain for $\delta_1^{(3)}$

derivative chain for $\delta_2^{(3)}$

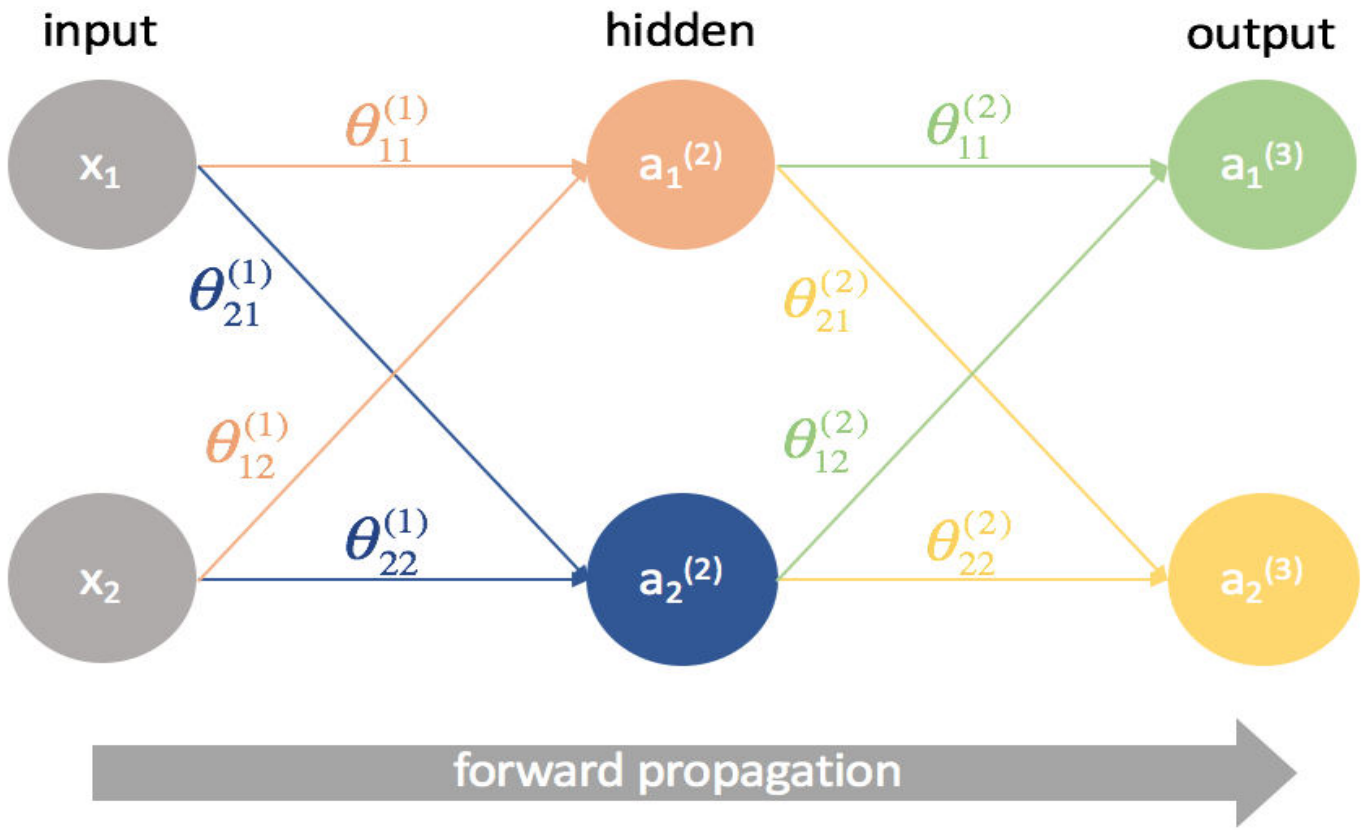
$$\frac{\partial J\left(\theta\right)}{\partial \theta_{11}^{(1)}} = \delta_1^{(3)} \left(\frac{\partial z_1^{(3)}}{\partial \mathrm{a}_1^{(2)}} \right) \left(\frac{\partial \mathrm{a}_1^{(2)}}{\partial z_1^{(2)}} \right) \left(\frac{\partial z_1^{(2)}}{\partial \theta_{11}^{(1)}} \right) + \delta_2^{(3)} \left(\frac{\partial z_2^{(3)}}{\partial \mathrm{a}_1^{(2)}} \right) \left(\frac{\partial \mathrm{a}_1^{(2)}}{\partial z_1^{(2)}} \right) \left(\frac{\partial z_1^{(2)}}{\partial \theta_{11}^{(1)}} \right)$$

$$\frac{\partial J\left(\theta\right)}{\partial \theta_{11}^{(1)}} = \delta_1^{(3)} \left(\frac{\partial z_1^{(3)}}{\partial \mathrm{a}_1^{(2)}} \right) \left(\frac{\partial \mathrm{a}_1^{(2)}}{\partial z_1^{(2)}} \right) \left(\frac{\partial z_1^{(2)}}{\partial \theta_{11}^{(1)}} \right) + \delta_2^{(3)} \left(\frac{\partial z_2^{(3)}}{\partial \mathrm{a}_1^{(2)}} \right) \left(\frac{\partial \mathrm{a}_1^{(2)}}{\partial z_1^{(2)}} \right) \left(\frac{\partial z_1^{(2)}}{\partial \theta_{11}^{(1)}} \right)$$

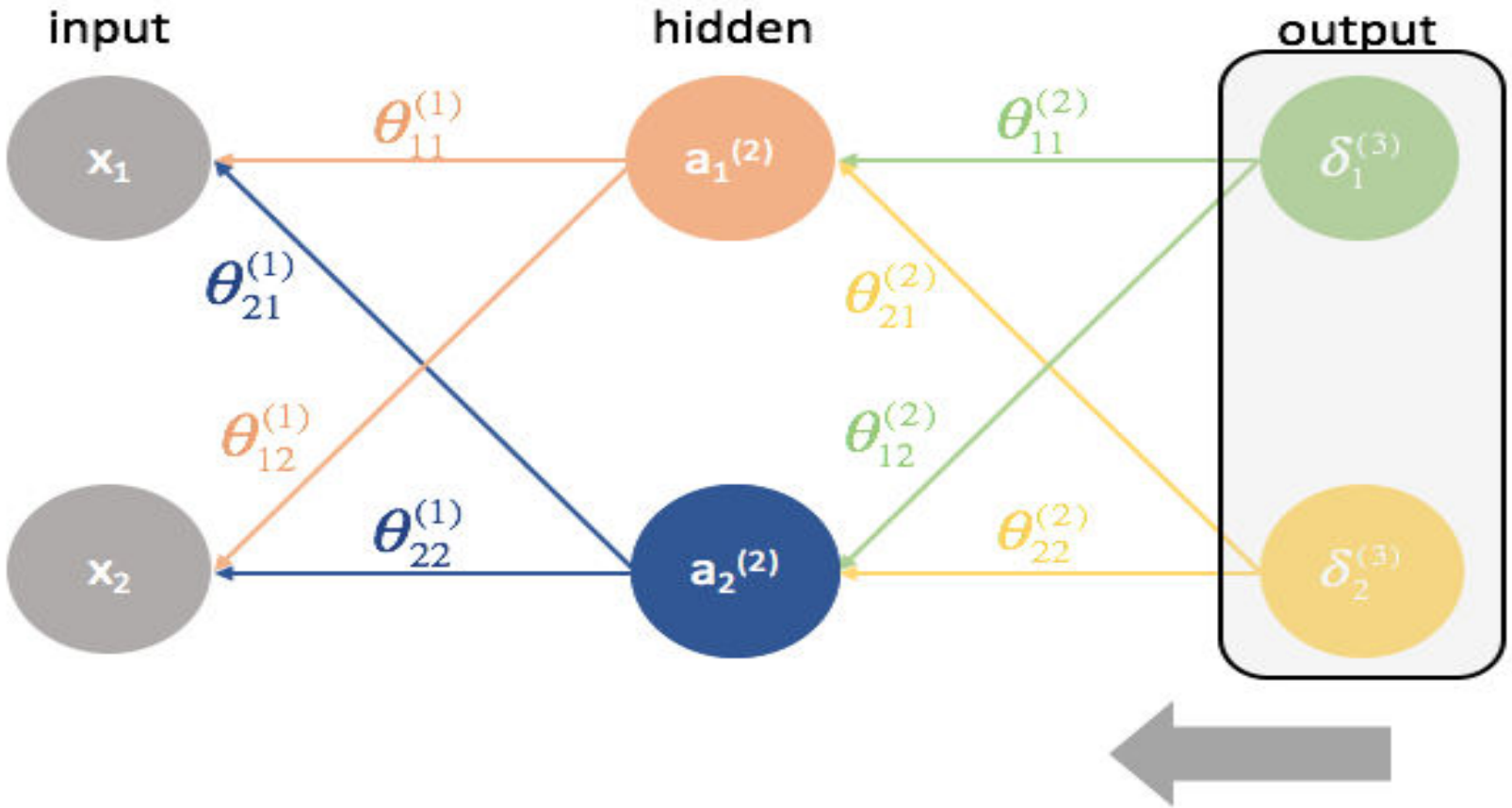
$$\frac{\partial J\left(\theta\right)}{\partial \theta_{11}^{(1)}} = \delta_1^{(3)} \theta_{11}^{(2)} f'\left(a^{(2)}\right) \left(\frac{\partial z_1^{(2)}}{\partial \theta_{11}^{(1)}} \right) + \delta_2^{(3)} \theta_{21}^{(2)} f'\left(a^{(2)}\right) \left(\frac{\partial z_1^{(2)}}{\partial \theta_{11}^{(1)}} \right)$$

$$\frac{\partial J\left(\theta\right)}{\partial \theta_{11}^{(1)}} = \left(\frac{\partial z_1^{(2)}}{\partial \theta_{11}^{(1)}} \right) \left(\delta_1^{(3)} \theta_{11}^{(2)} f'\left(a^{(2)}\right) + \delta_2^{(3)} \theta_{21}^{(2)} f'\left(a^{(2)}\right) \right)$$

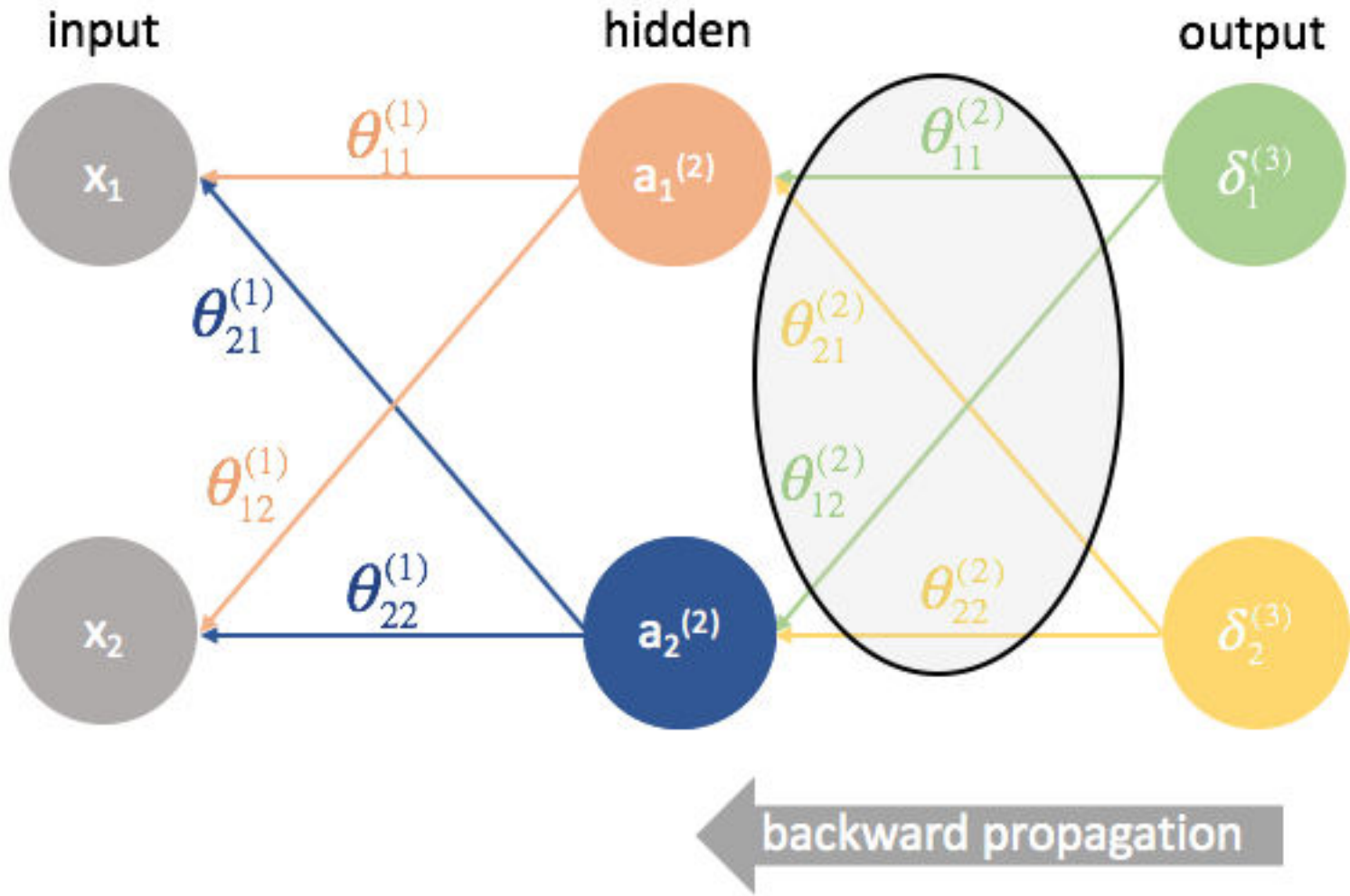
$$\frac{\partial J\left(\theta\right)}{\partial \theta_{11}^{(1)}} = \left(\frac{\partial z_1^{(2)}}{\partial \theta_{11}^{(1)}} \right) \left(\delta_1^{(2)} \right)$$


input
hidden
output
x_1
x_2
$a_1^{(2)}$
$a_2^{(2)}$
$a_1^{(3)}$
$a_2^{(3)}$
$\theta_{11}^{(1)}$
$\theta_{21}^{(1)}$
$\theta_{12}^{(1)}$
$\theta_{22}^{(1)}$
$\theta_{11}^{(2)}$
$\theta_{21}^{(2)}$
$\theta_{12}^{(2)}$
$\theta_{22}^{(2)}$
forward propagation

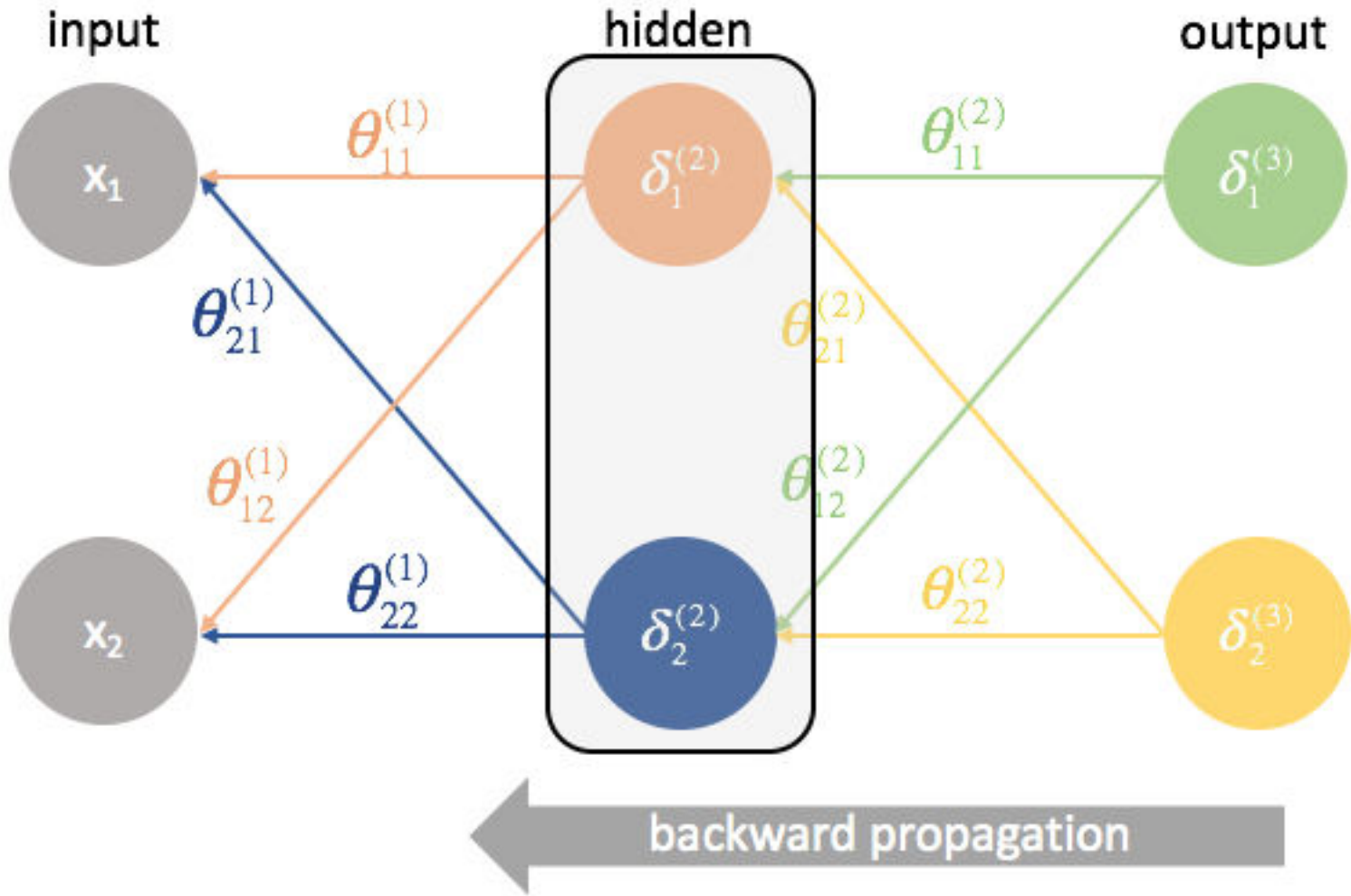
$$\delta^{(3)} = \frac{1}{m}\left(y - a^{(3)}\right) f'\left(a^{(3)}\right)$$



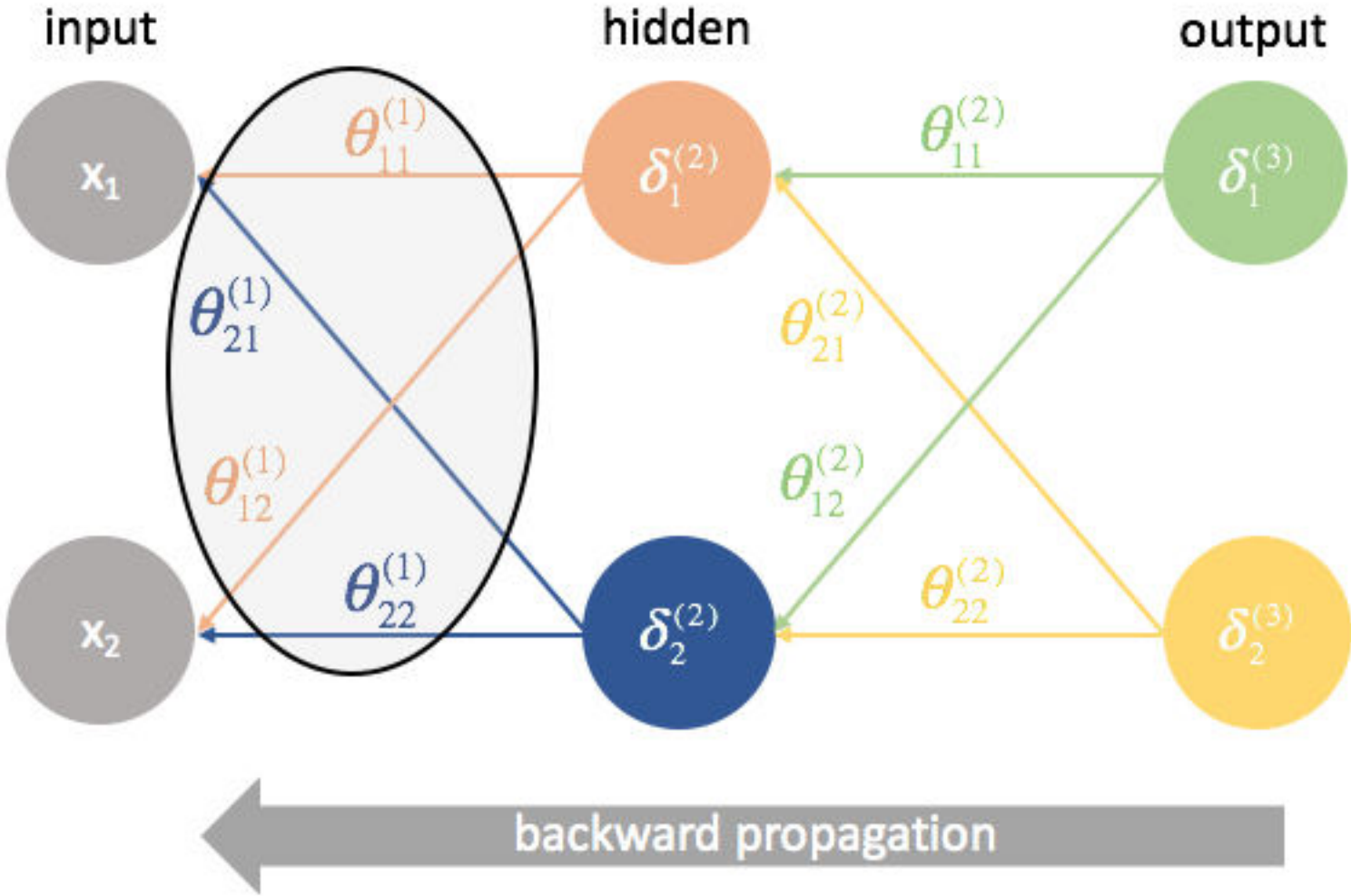
$$\frac{\partial J(\theta)}{\partial \theta_{ij}^{(2)}} = \left(\delta^{(3)}\right)^T a^{(2)}$$



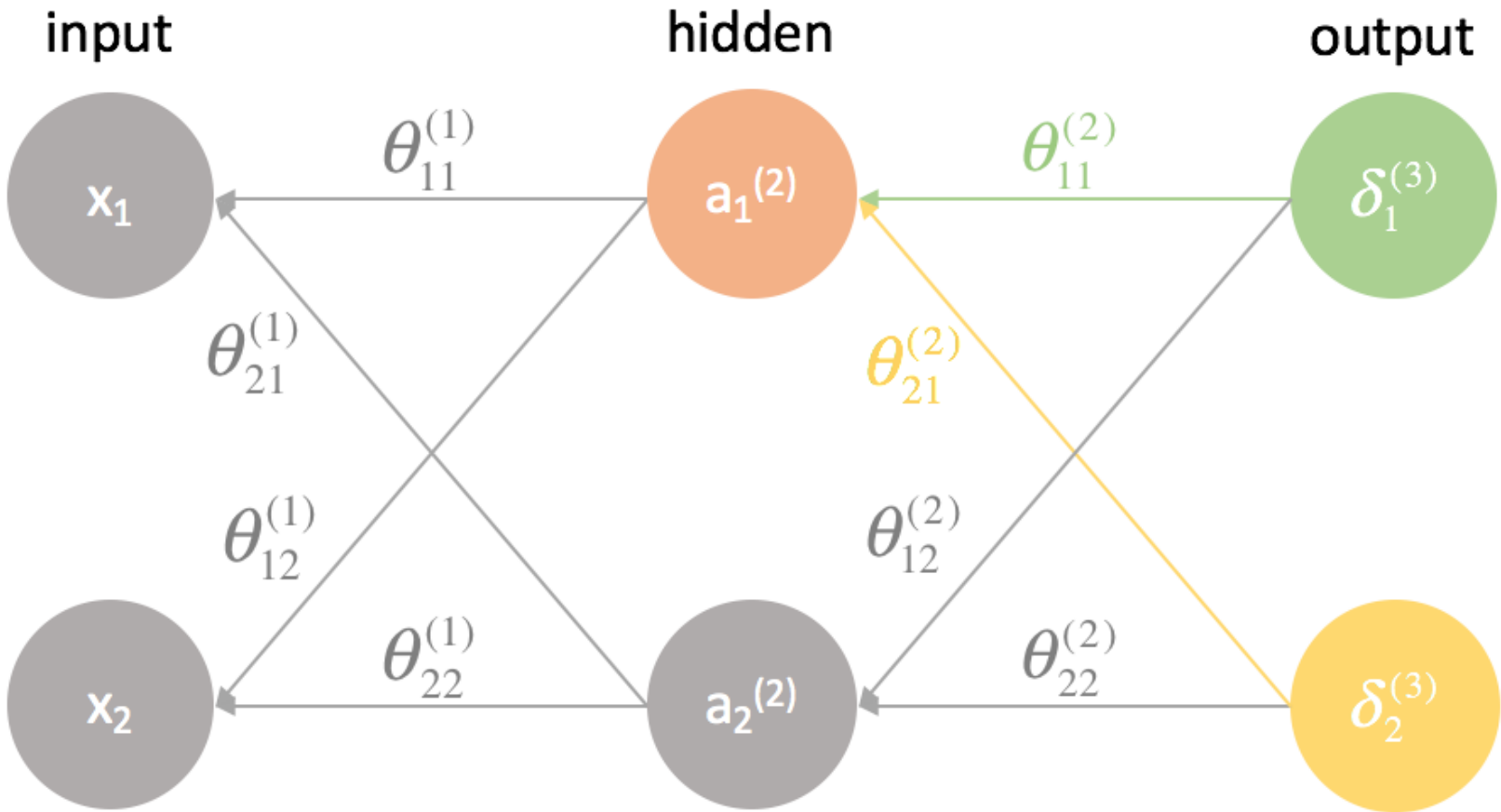
$$\delta^{(2)} = \delta^{(3)}\Theta^{(2)} f'\left(a^{(2)}\right)$$



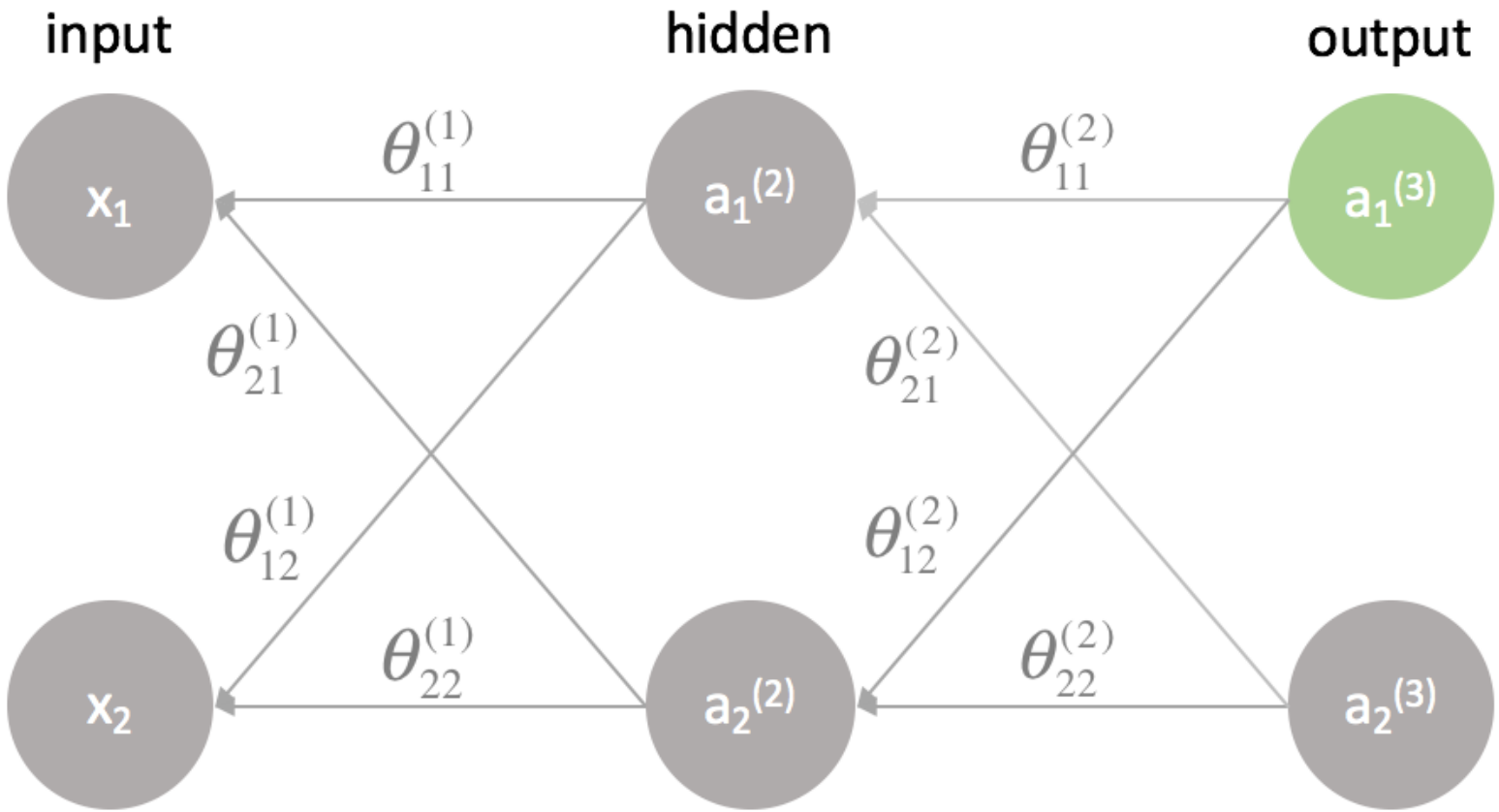
$$\frac{\partial J(\theta)}{\partial \theta_{ij}^{(1)}} = \left(\delta^{(2)}\right)^T x$$



$$\delta_1^{(2)} = \left(\delta_1^{(3)}\theta_{11}^{(2)} + \delta_2^{(3)}\theta_{21}^{(2)}\right) f'\left(a^{(2)}\right)$$



$$\delta_1^{(3)} = \frac{1}{m}\left(y_1 - a_1^{(3)}\right) f'\left(a^{(3)}\right)$$



Optimization for Neural Networks

Gradient Descent

- Gradient descent is an algorithm to minimize the loss function w.r.t parameters.
- Types of Gradient Descent
  - Batch Gradient Descent
  - Stochastic Gradient Descent
  - Mini Batch Gradient Descent

Gradient Descent

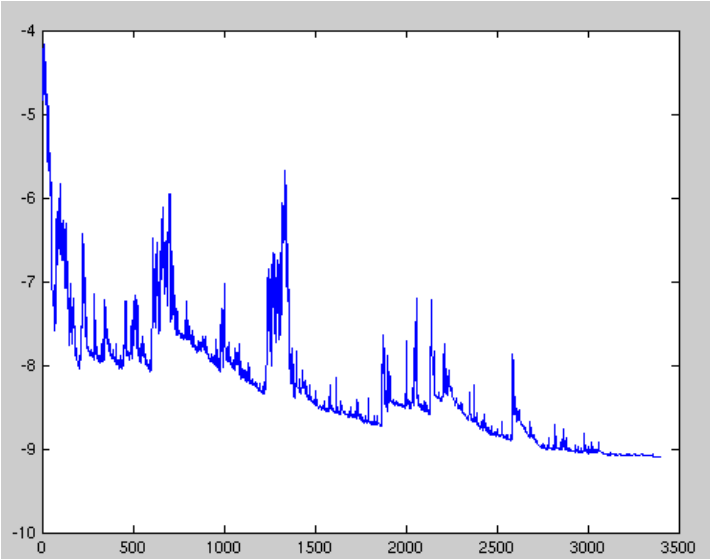
- Batch Gradient Descent

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta).$$

Gradient Descent

- Stochastic Gradient Descent

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)}).$$

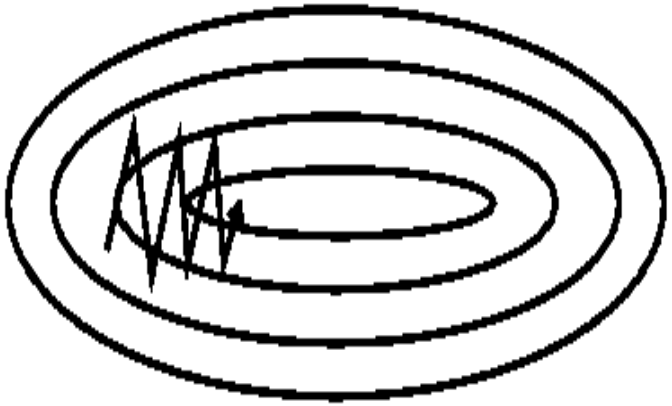


Gradient Descent

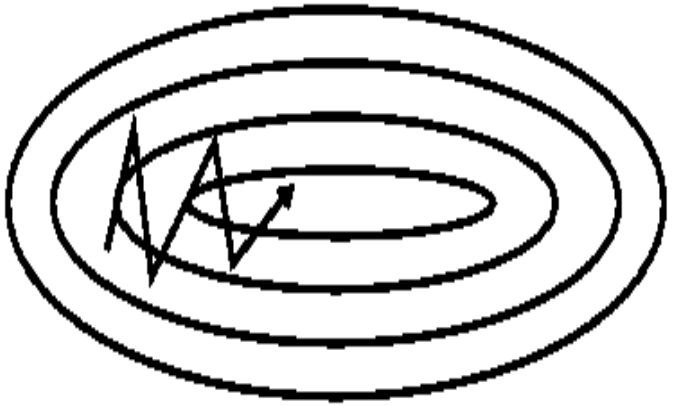
- Mini Batch Gradient Descent

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}).$$

Momentum

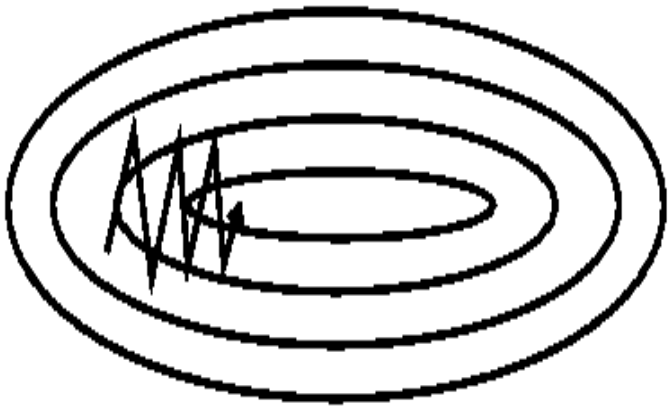


SGD without Momentum

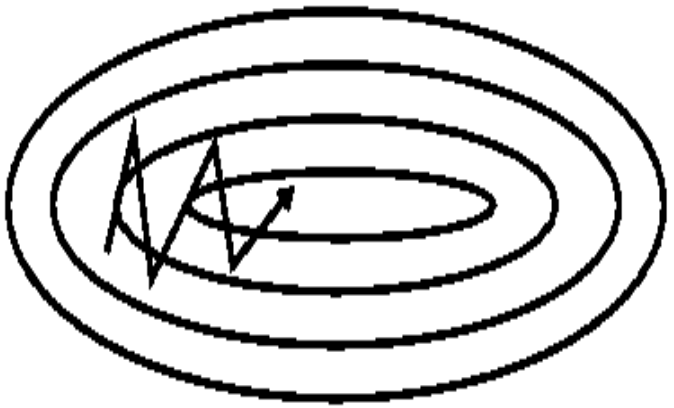


SGD with Momentum

Momentum



SGD without Momentum



SGD with Momentum

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

Nesterov accelerated gradient

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

Adagrad

$$g_{t,i} = \nabla_\theta J(\theta_{t,i}).$$

The SGD update for every parameter θ_i at each time step t then becomes:

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}.$$

In its update rule, Adagrad modifies the general learning rate η at each time step t for every parameter θ_i based on the past gradients that have been computed for θ_i:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}.$$

$G_t \in \mathbb{R}^{d \times d}$ here is a diagonal matrix where each diagonal element i, i is the sum of the squares of the gradients w.r.t. θ_i up to time step t [1]

Adadelta

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2.$$

$$\Delta\theta_t = -\eta \cdot g_{t,i}$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t.$$

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}.$$

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

RMSprop

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

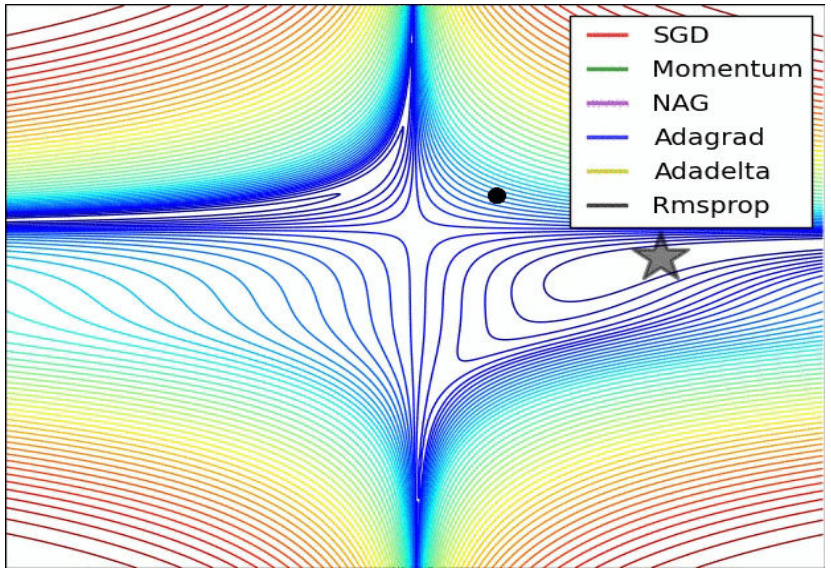
Adam

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

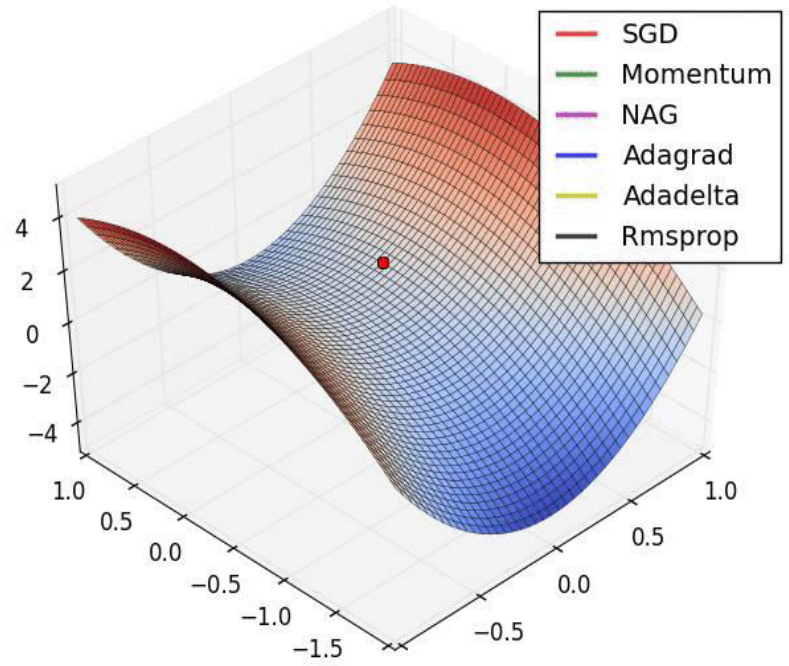
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t.$$

The authors propose default values of 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ϵ.



SGD optimization on loss surface contours

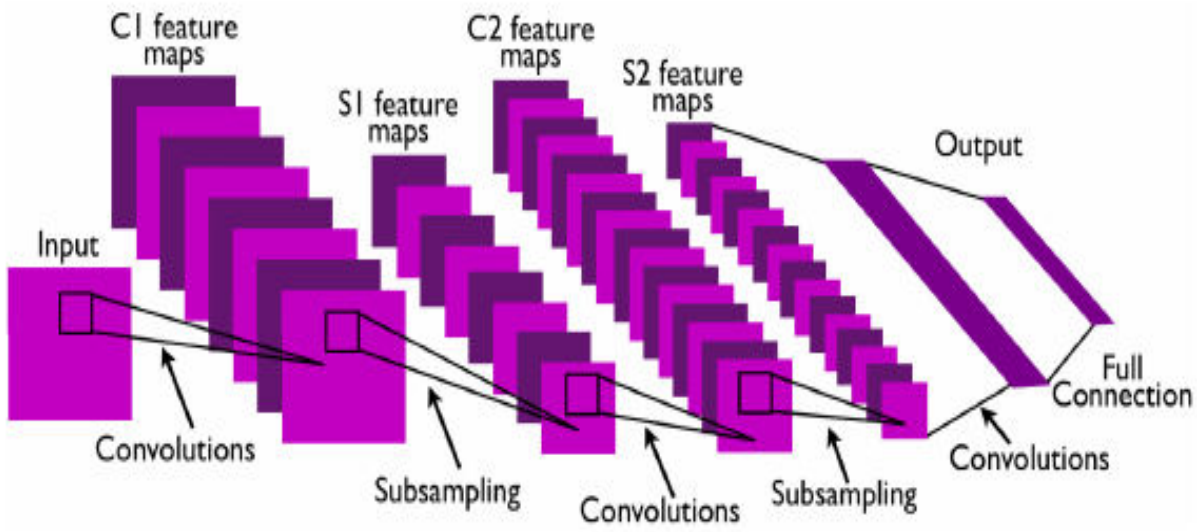


SGD optimization on saddle point

Introduction to Convolution Neural Networks(CNN)

Convolutional networks

- Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.



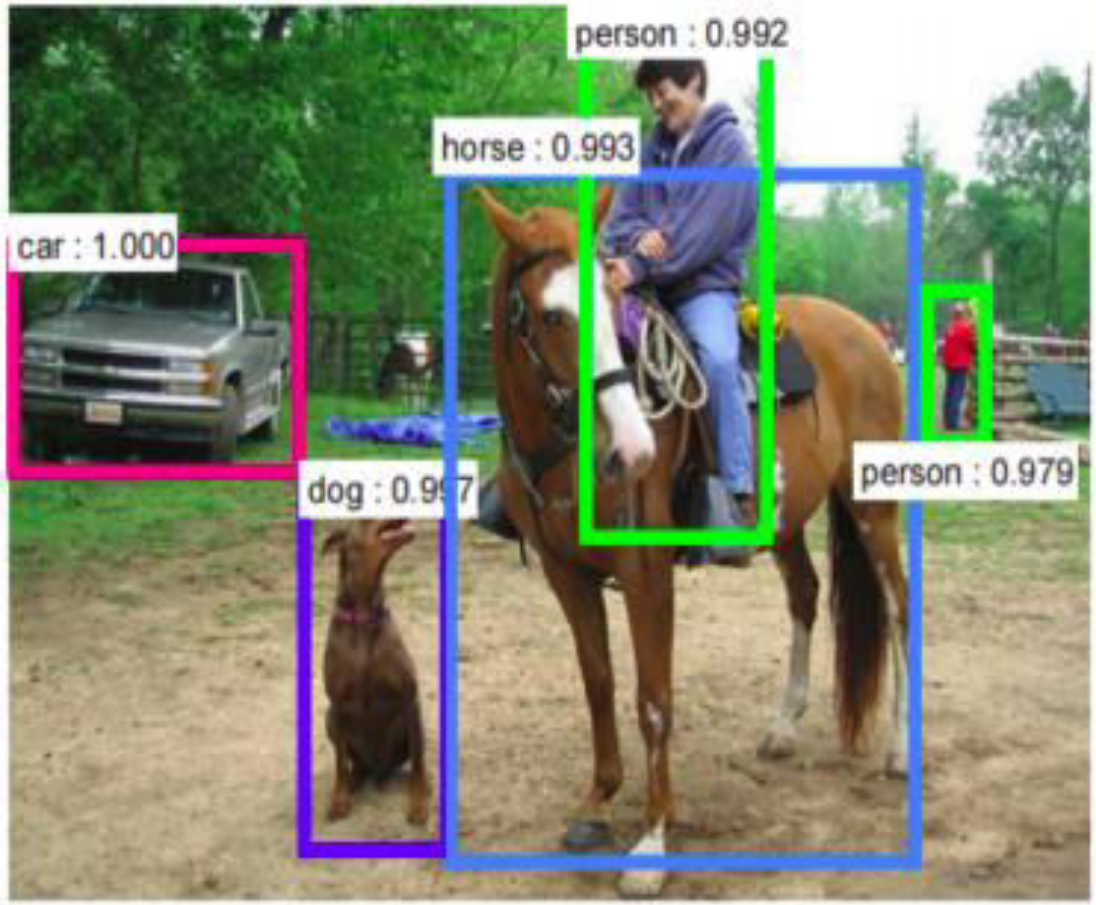
Caption Generation

a soccer player is kicking a soccer ball

a street sign on a pole in front of a building

a couple of giraffe standing next to each other

Object Detection



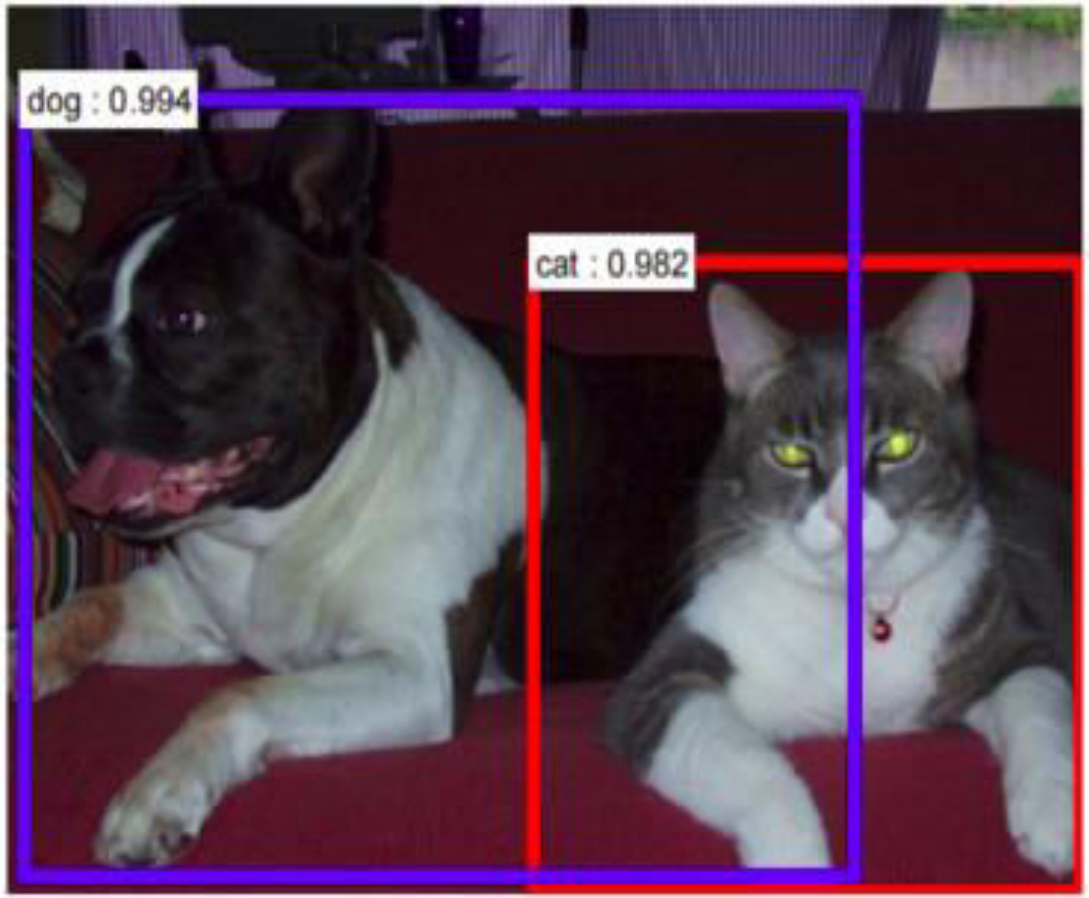
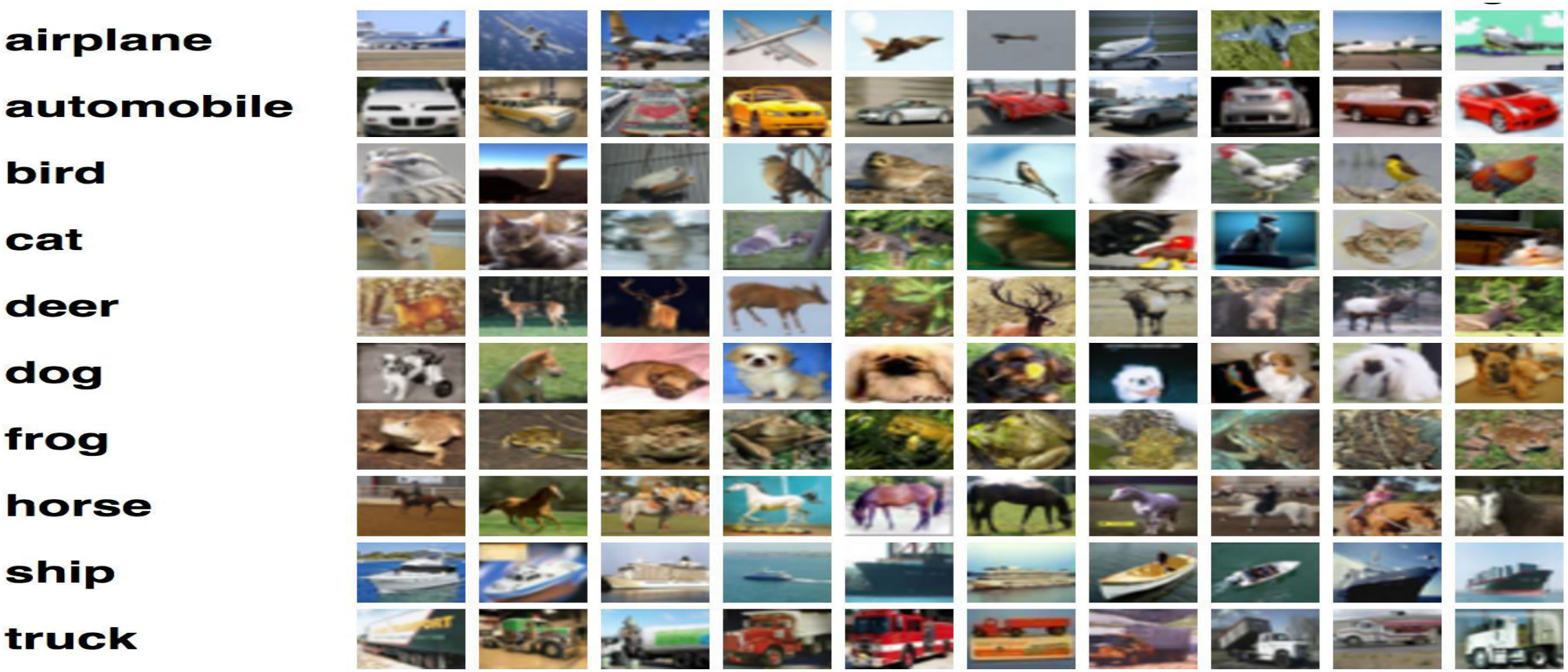


Image Segmentation

Image Classification



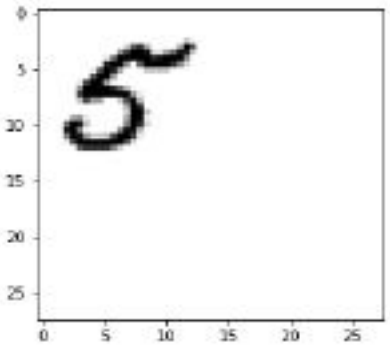
Why Image Classification is Hard

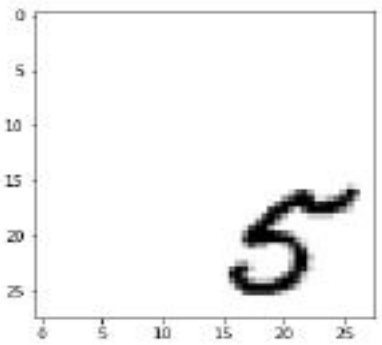
Different lighting, contrast, viewpoints, etc.

Image Source: twitter.com%2Fcats&psig=AOvVaw30_o-PCM-K21DiMAJQimQ4&ust=1553887775741551

Image Source: https://www.123rf.com/photo_76714328_side-view-of-tabby-cat-face-over-white.html

Or even simple translation



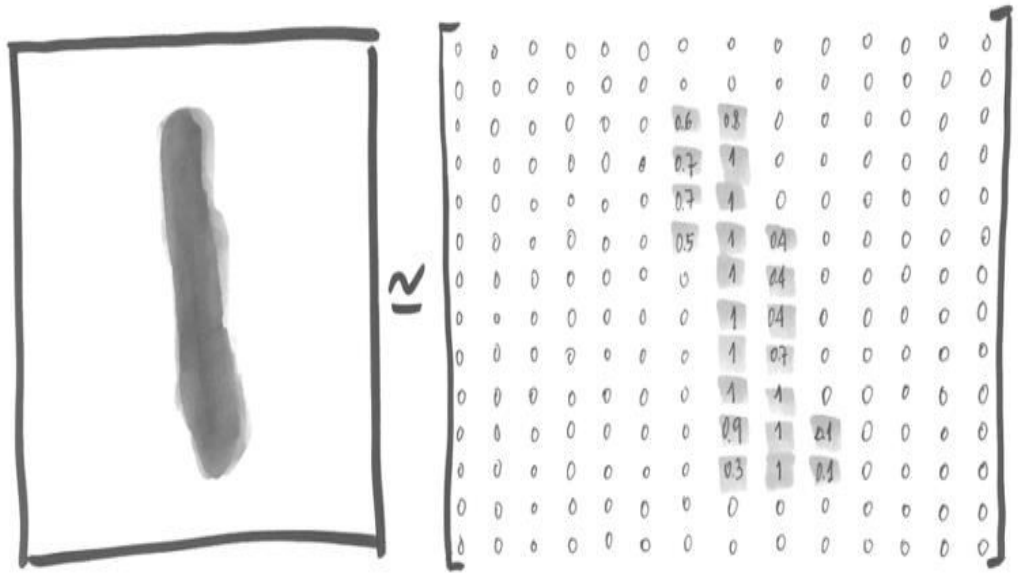


Advantages of Convolutional Neural Networks

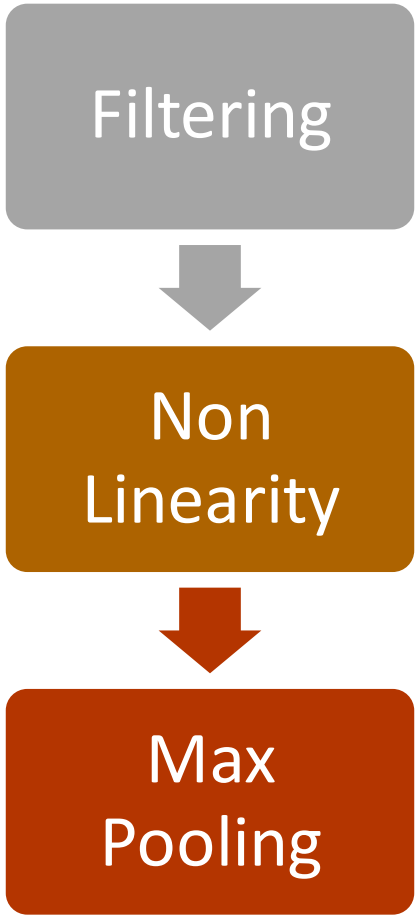
- Sparse-connectivity: A single element in the feature map is connected to only a small patch of pixels. (This is very different from connecting to the whole input image, in the case of multilayer perceptrons.)
- Parameter-sharing: The same weights are used for different patches of the input image.

ConvNets Building Blocks

- Image



Component of ConvNet



Convolution Operation(Filtering)

- consider a 5 x 5 image whose pixel values are only 0 and 1 (note that for a grayscale image, pixel values range from 0 to 255, the green matrix below is a special case where pixel values are only 0 and 1):
- Also, consider another 3 x 3 matrix as shown below:

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

Input

1	0	1
0	1	0
1	0	1

Filter / Kernel

Convolution Operation

1x1	1x0	1x1	0	0
0x0	1x1	1x0	1	0
0x1	0x0	1x1	1	1
0	0	1	1	0
0	1	1	0	0

Input x Filter

4		

Feature Map

Convolution Operation

1	1x1	1x0	0x1	0
0	1x0	1x1	1x0	0
0	0x1	1x0	1x1	1
0	0	1	1	0
0	1	1	0	0

Input x Filter

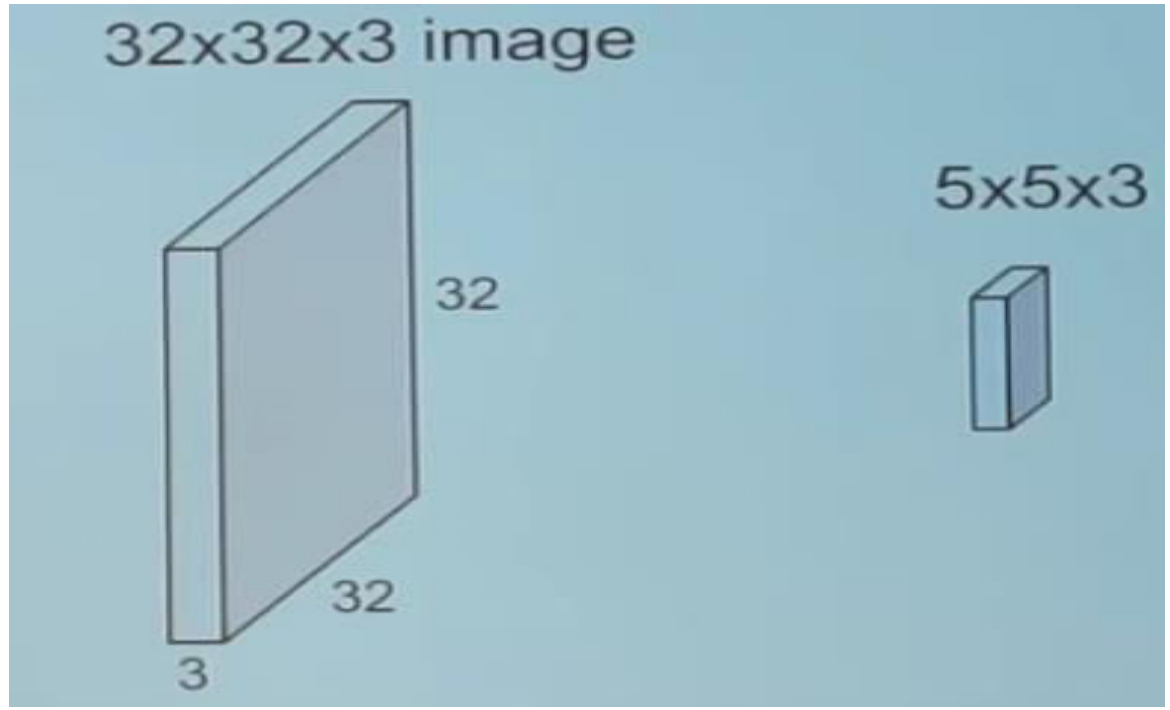
4	3	

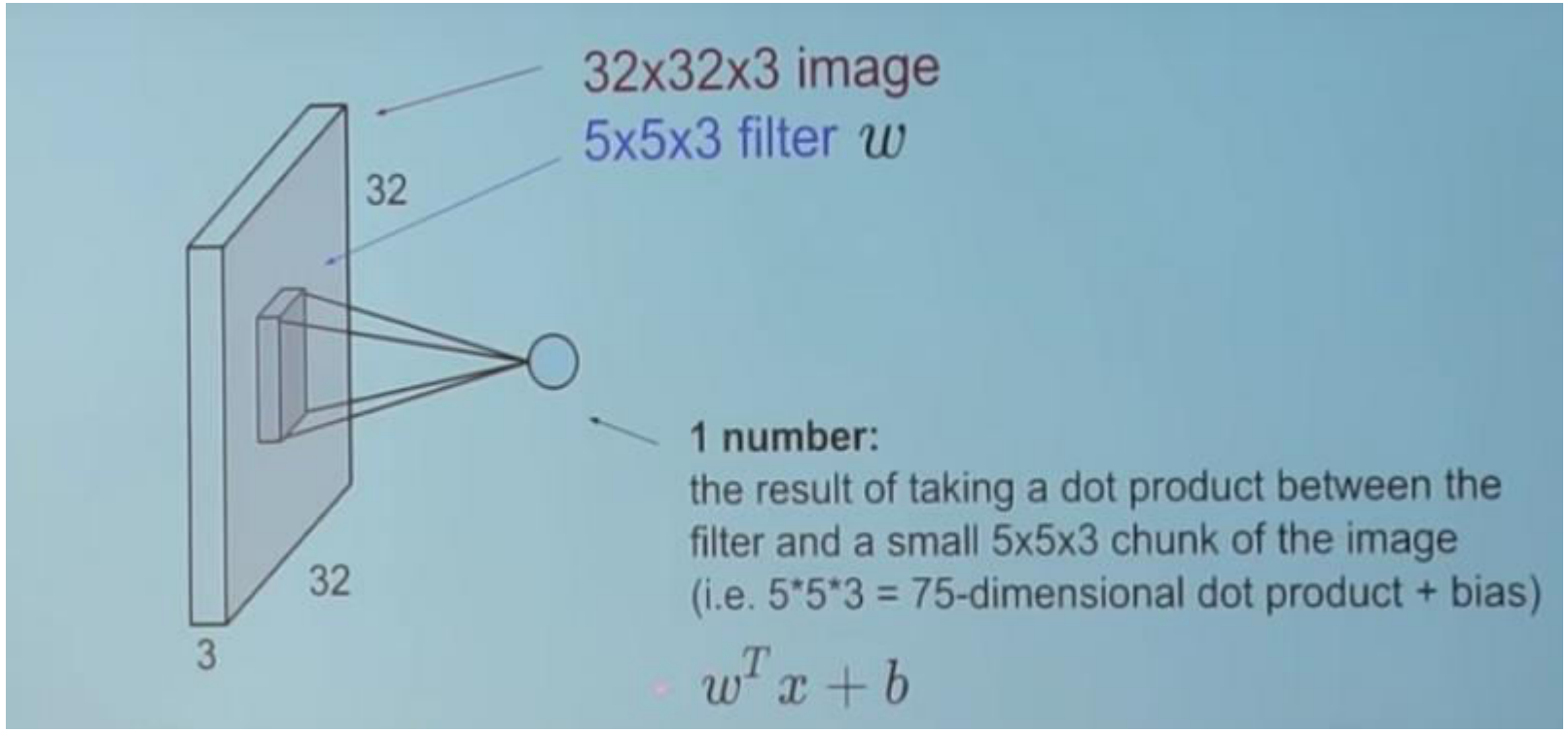
Feature Map

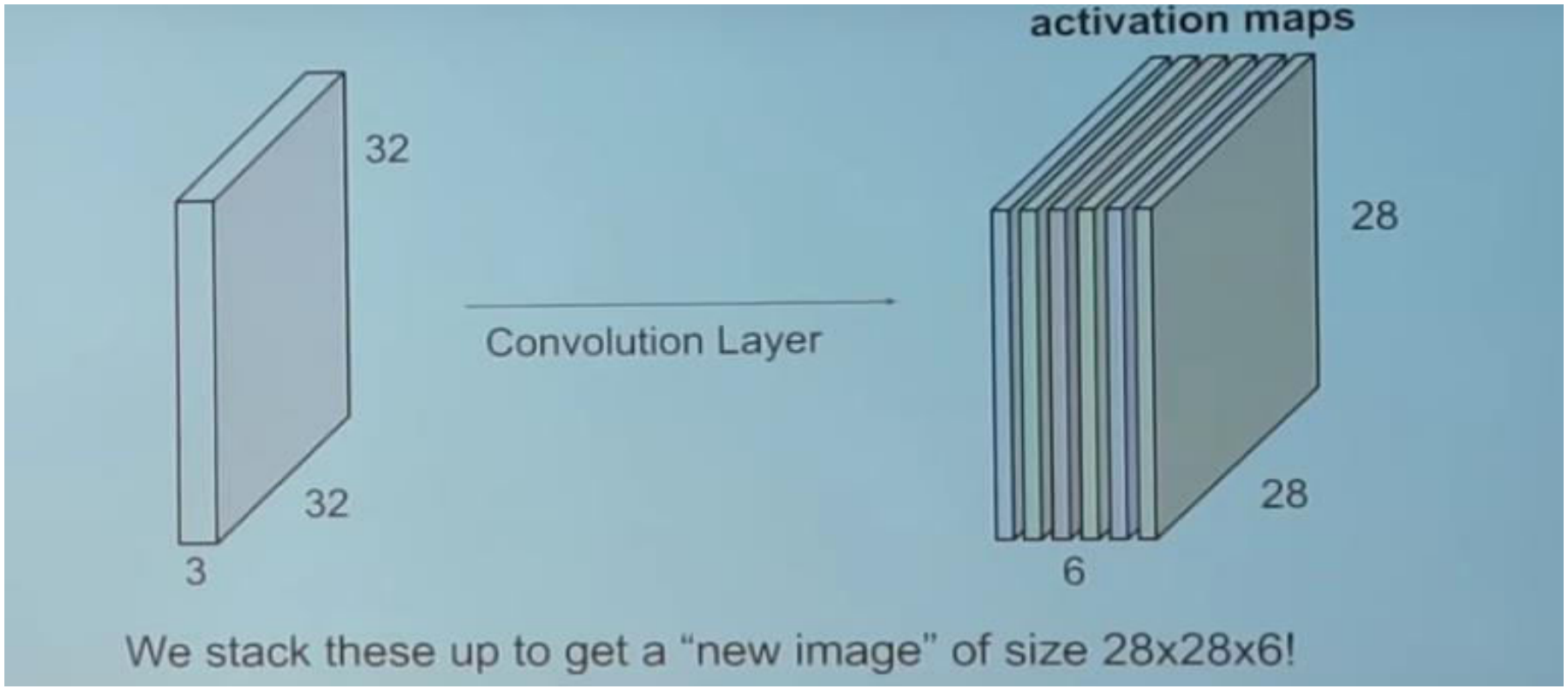
Convolution Operation

1	1	1	0	0
0	1	1	1	0
0	0	1x1	1x0	1x1
0	0	1x0	1x1	0x0
0	1	1x1	0x0	0x1

4	3	4
2	4	3
2	3	4

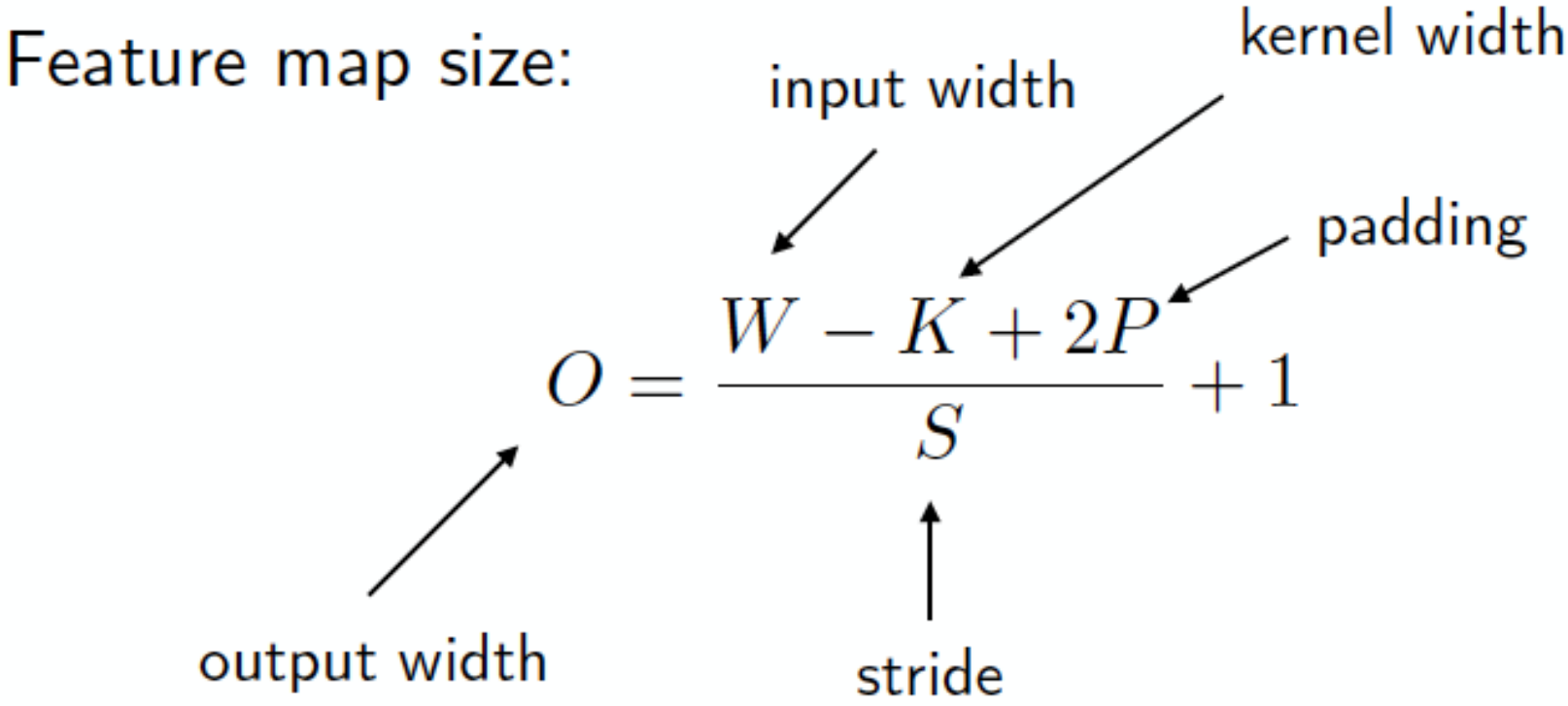

32x32x3 image
32
32
3
5x5x3


32x32x3 image
5x5x3 filter w
32
32
3
1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)
$w^T x + b$

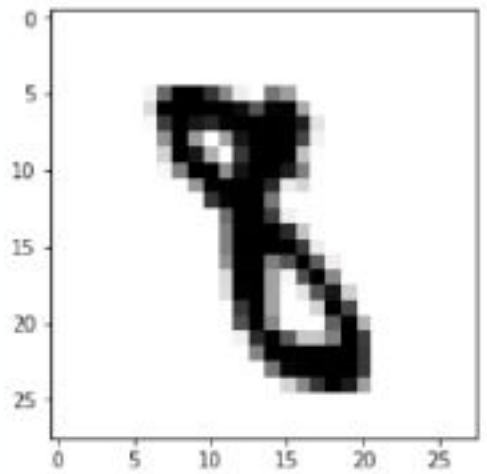

activation maps
32
32
3
Convolution Layer
28
28
6
We stack these up to get a "new image" of size 28x28x6!

Size Before and After Convolutions

Feature map size:



Pytorch implementation of convolution




```
a.shape
```

```
(1, 28, 28)
```

```
import torch
```

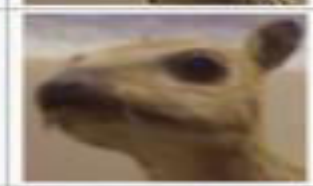
```
conv = torch.nn.Conv2d(in_channels=1,
                       out_channels=8,
                       kernel_size=(5, 5),
                       stride=(1, 1))
```

```
conv.weight.size()
```

```
torch.Size([8, 1, 5, 5])
```

```
conv.bias.size()
```

```
torch.Size([8])
```

Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

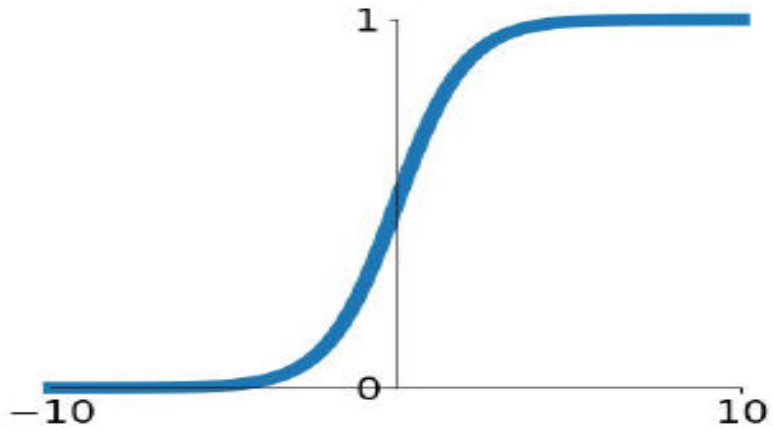
Input

Feature Map

Hyper Parameters for Features Map

- The size of the Feature Map (Convolved Feature) is controlled by three parameters
  1. Depth
  2. Stride
  3. Zero Padding

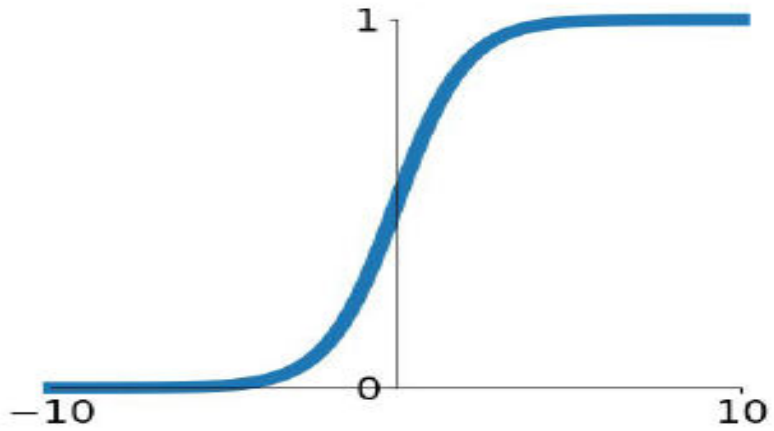
Activation Functions



Sigmoid

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

Activation Functions



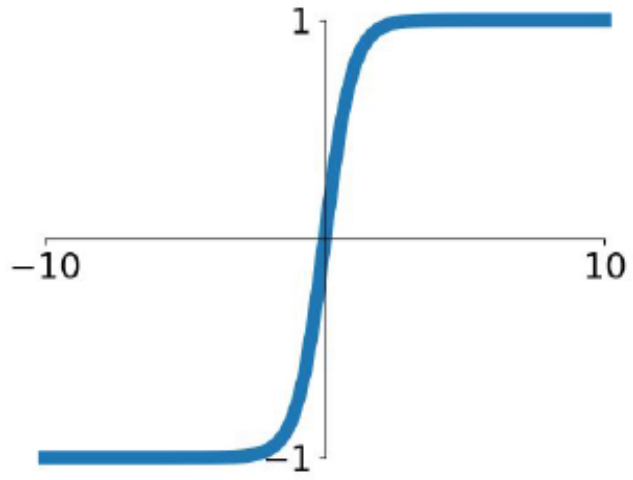
Sigmoid

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

3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
3. exp() is a bit compute expensive

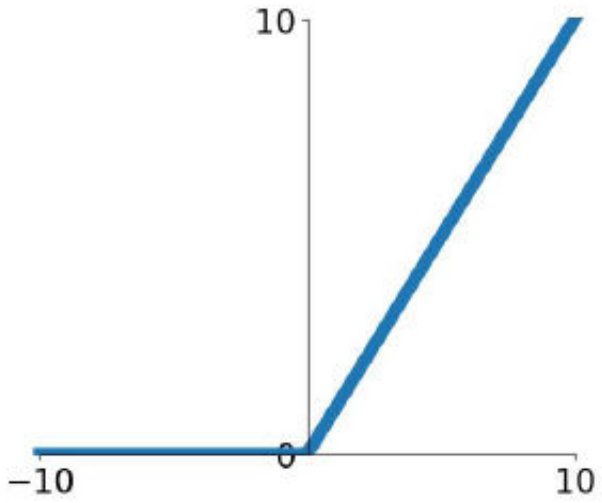
Activation Function



tanh(x)

- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

Activation Function

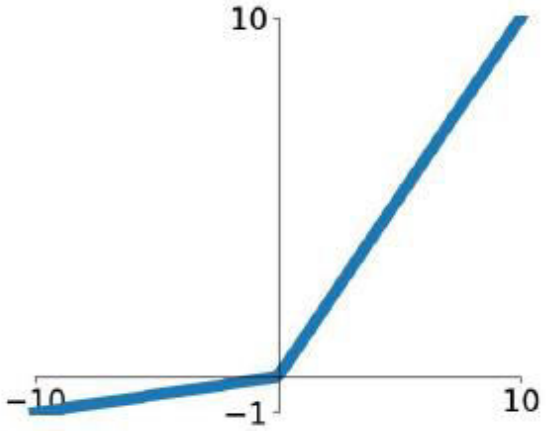


ReLU
(Rectified Linear Unit)

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice

Not zero-centered output
Saturates in –ve Region

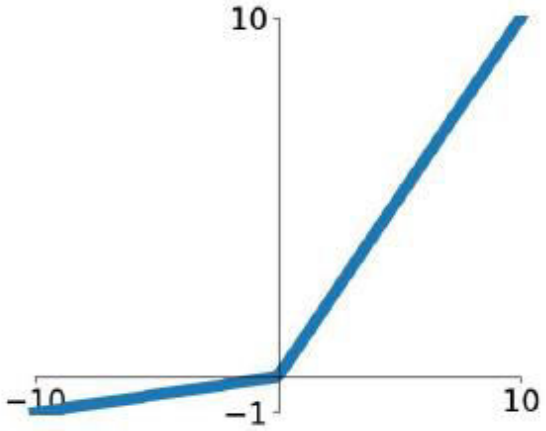
Activation Function



Leaky ReLU

$$f(x) = \max(0.01x, x)$$

Activation Function

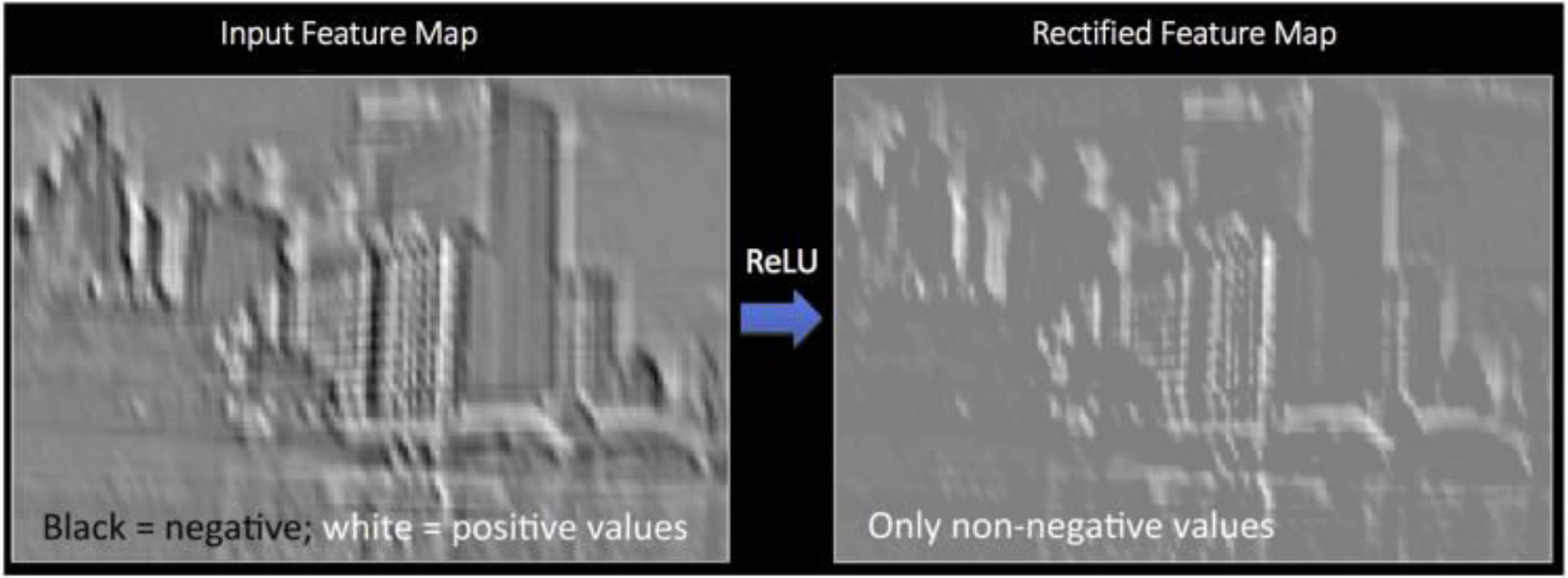


Leaky ReLU

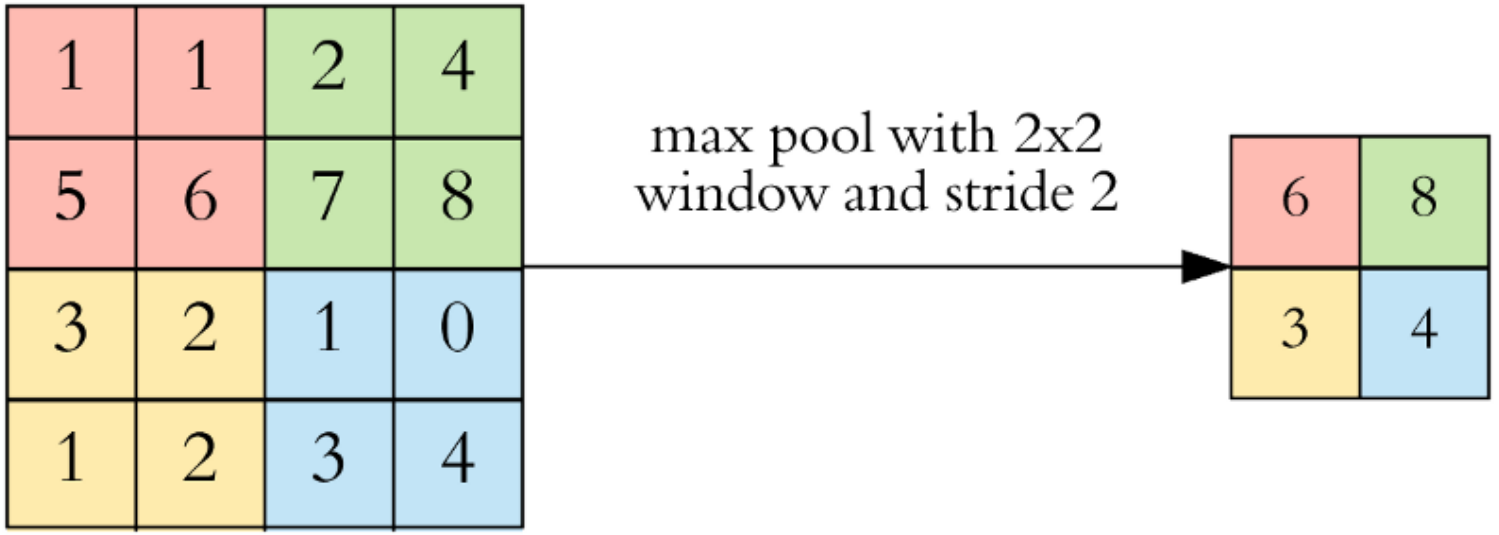
$$f(x) = \max(0.01x, x)$$

Parametric Rectifier (PReLU)

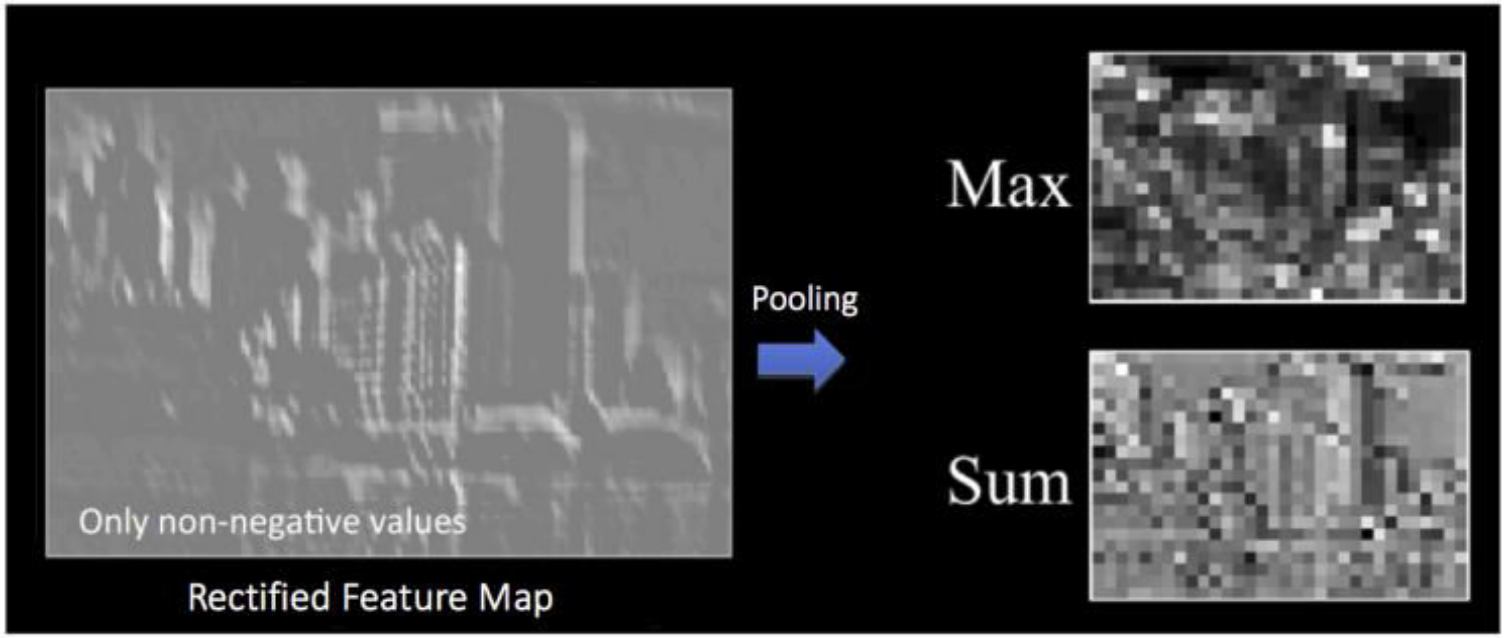
$$f(x) = \max(\alpha x, x)$$


Input Feature Map
Rectified Feature Map
ReLU
Black = negative; white = positive values
Only non-negative values

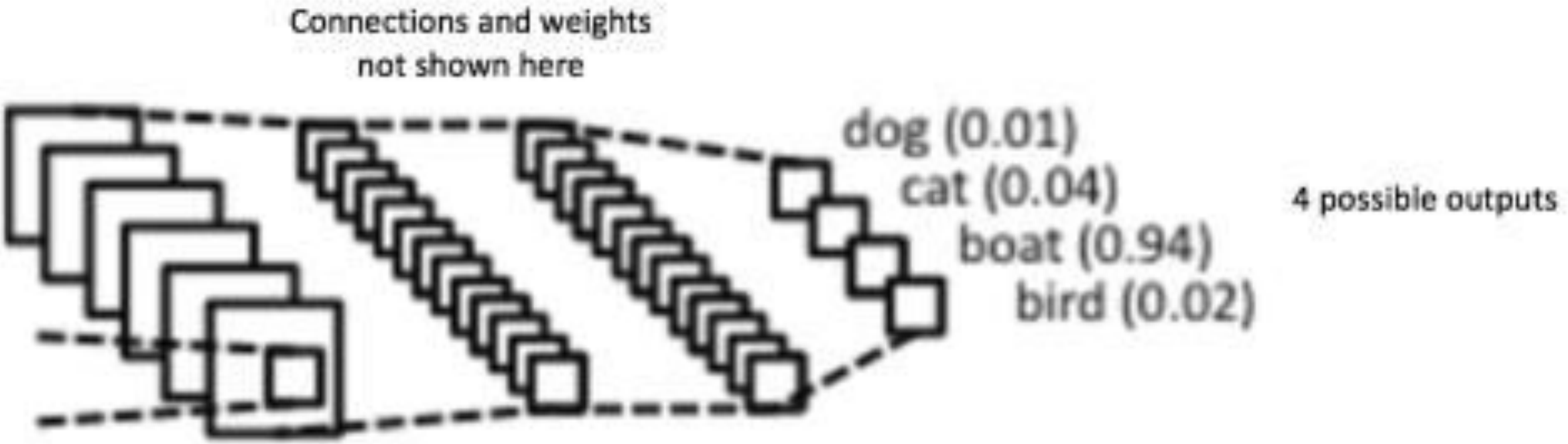
Pooling


1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
max pool with 2x2
window and stride 2
6 8
3 4

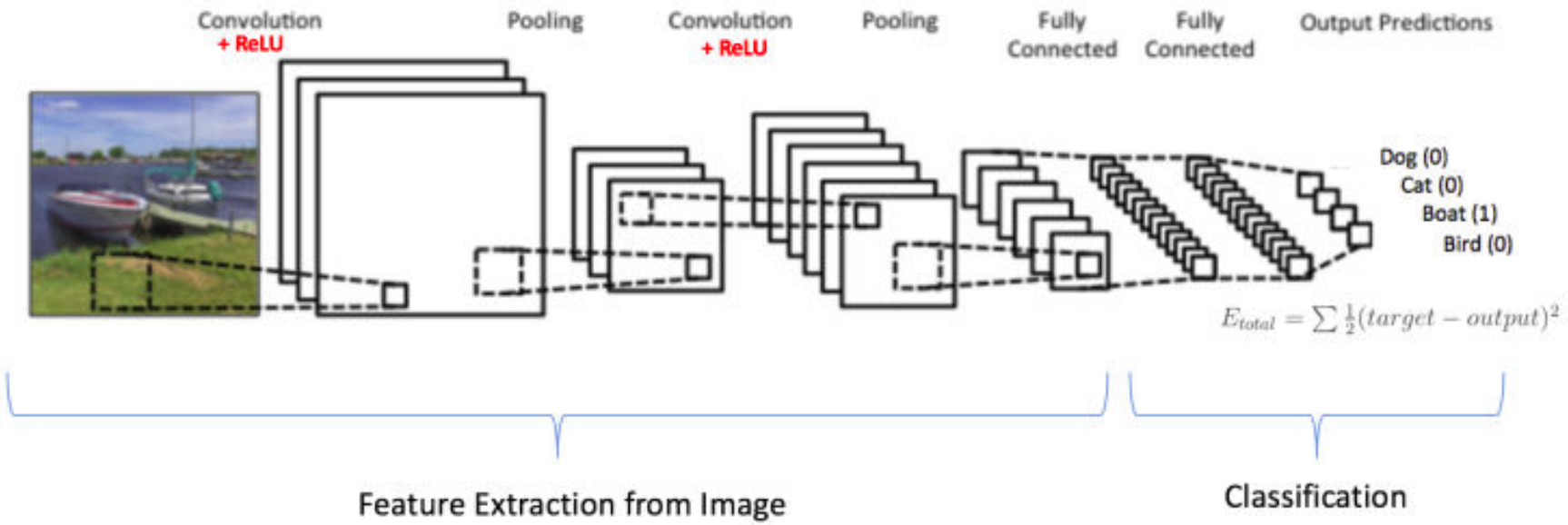
Pooling



1. Max Pooling
2. Average Pooling
3. Sum Pooling



Fully Connected Layer

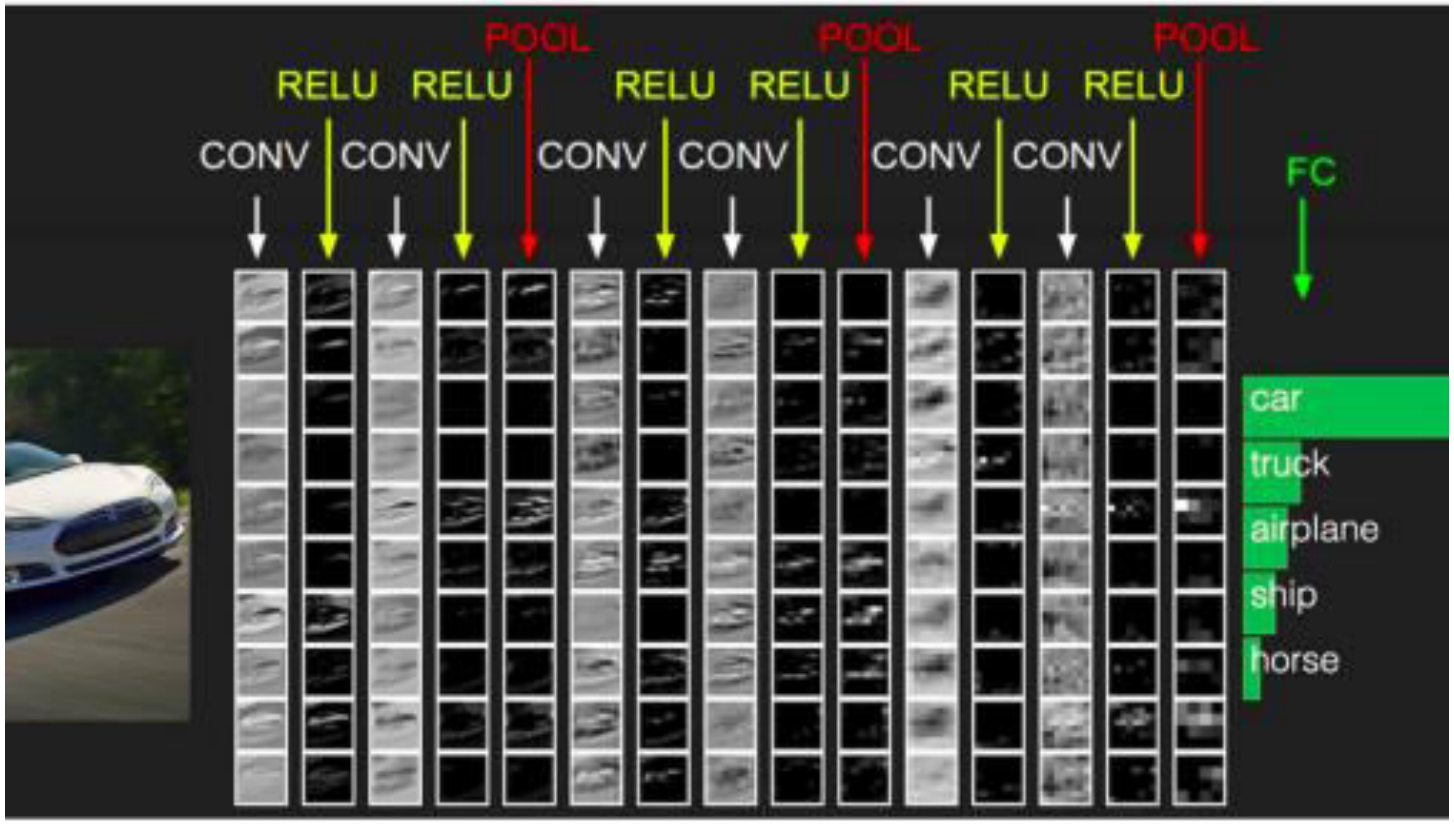


Putting it all together

- Input Boat Image
- Output: 0 1 0 0

Training ConvNet

1. Initialize all filters and parameters / weights with random values
2. Forward Pass
3. Calculate Loss
4. Apply Backpropagation to learn the feature map and filter



ConvNet Architecture

- http://scs.ryerson.ca/~aharley/vis/conv/flat.html

Key architectures in the recent evolution of ConvNets

LeNet(1988)

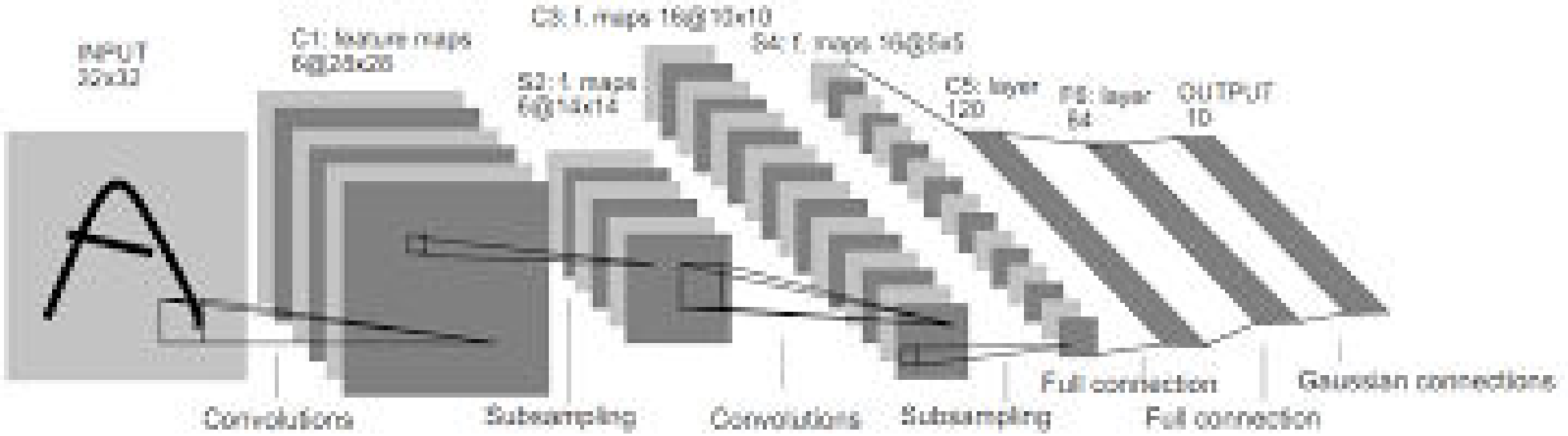
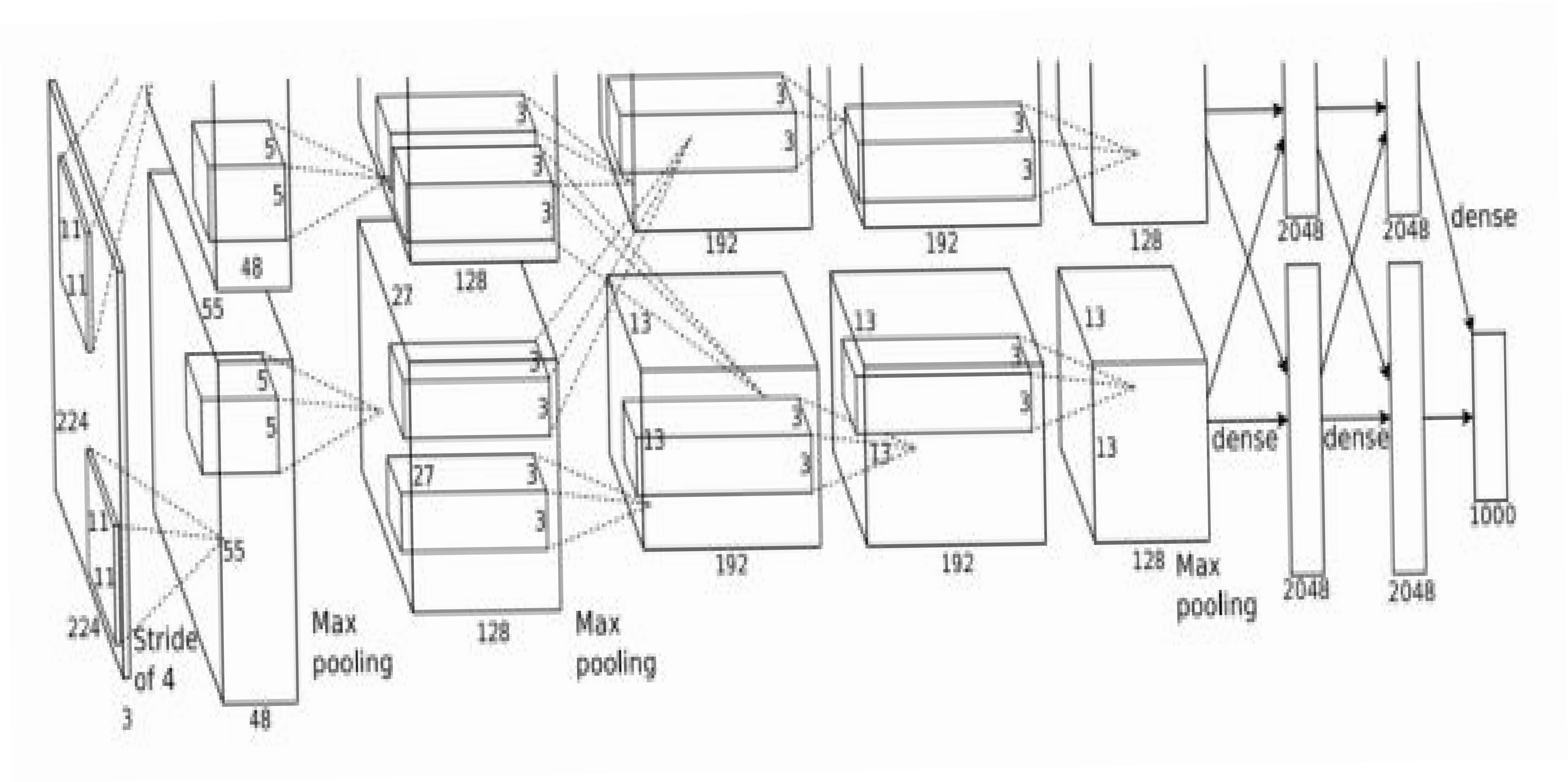


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

ImageNet Challange

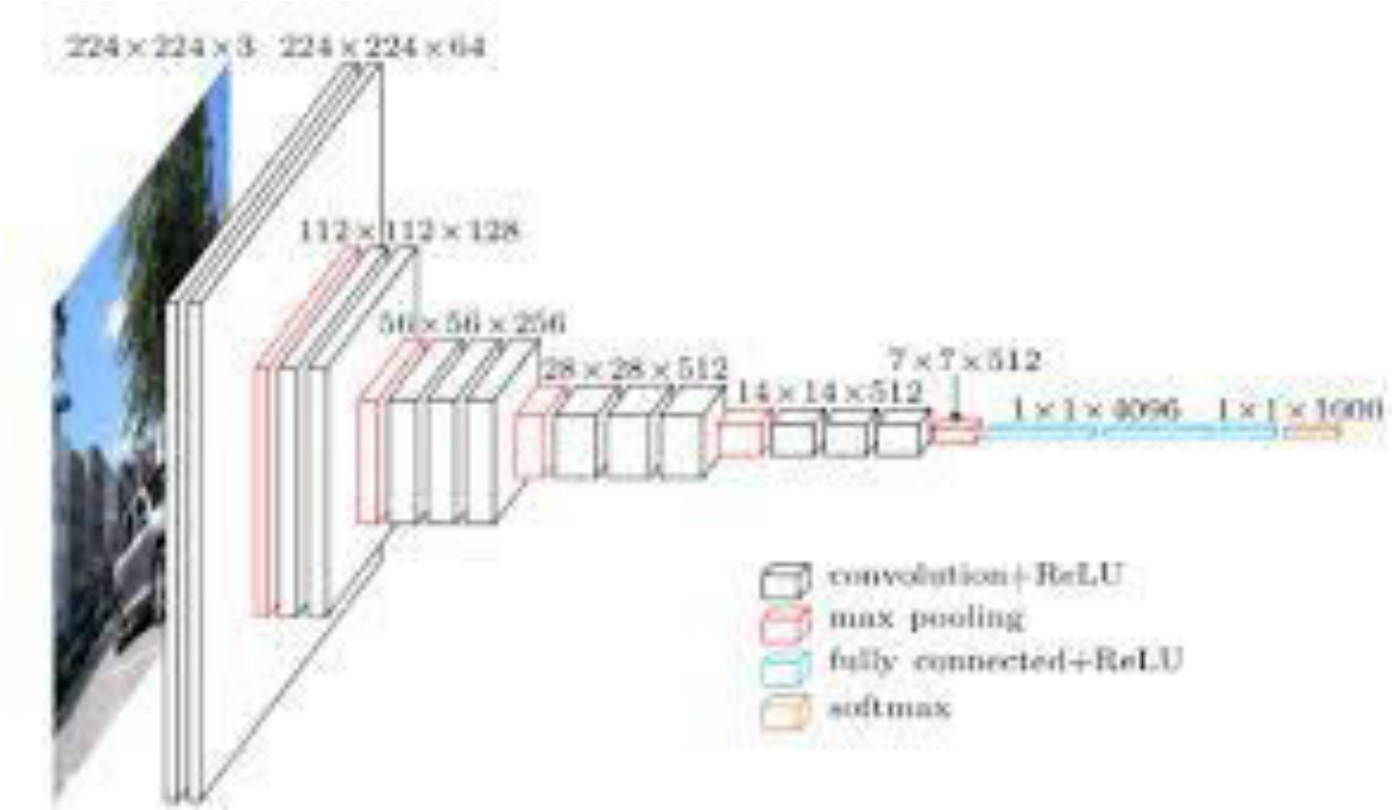
- The **ImageNet** project is a large visual database designed for use in visual object recognition software research.
- In 2010, the annual ImageNet Large Scale Visual Recognition Challenge (ILSVRC) was started.
- The goal of the challenge was to classify 10000000 images in 1000 different classes.

AlexNet



224
224
3
11
11
11
11
Stride of 4
55
55
48
48
5
5
5
5
Max pooling
27
27
128
128
3
3
Max pooling
13
13
13
13
13
192
192
192
192
128
128
Max pooling
dense
dense
dense
2048
2048
2048
2048
1000

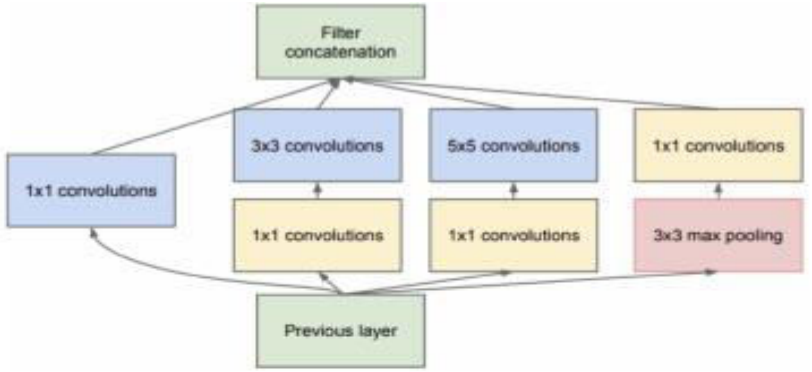
VGG Net



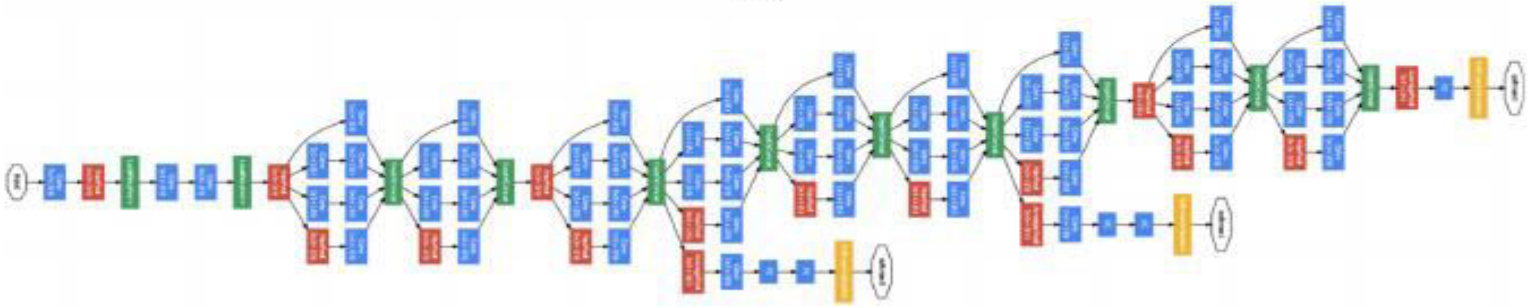
VERY DEEP CONVOLUTIONAL NETWORKS
FOR LARGE-SCALE IMAGE RECOGNITION

Karen Simonyan* **& Andrew Zisserman+**
Visual Geometry Group, Department of Engineering Science, University of Oxford
{karen,az}@robots.ox.ac.uk

GoogLeNet



(a)



(b)

Going deeper with convolutions

Christian Szegedy
Google Inc.

Wei Liu
University of North Carolina, Chapel Hill

Yangqing Jia
Google Inc.

Pierre Sermanet
Google Inc.

Scott Reed
University of Michigan

Dragomir Anguelov
Google Inc.

Dumitru Erhan
Google Inc.

Vincent Vanhoucke
Google Inc.

Andrew Rabinovich
Google Inc.

Residual Net

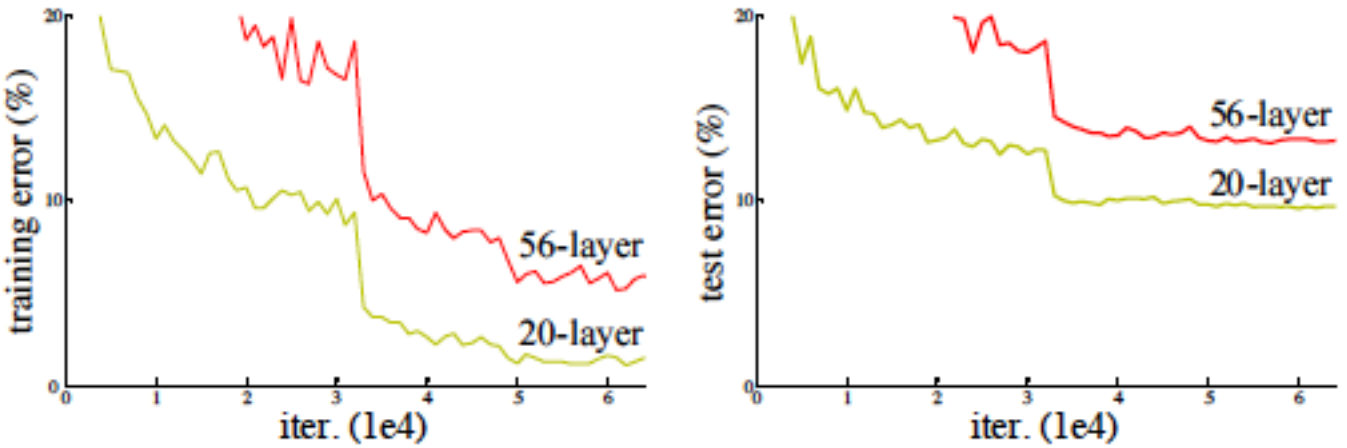
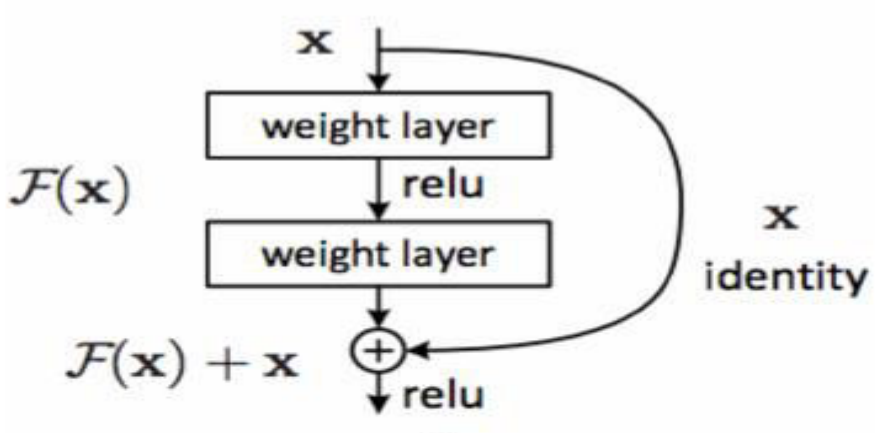


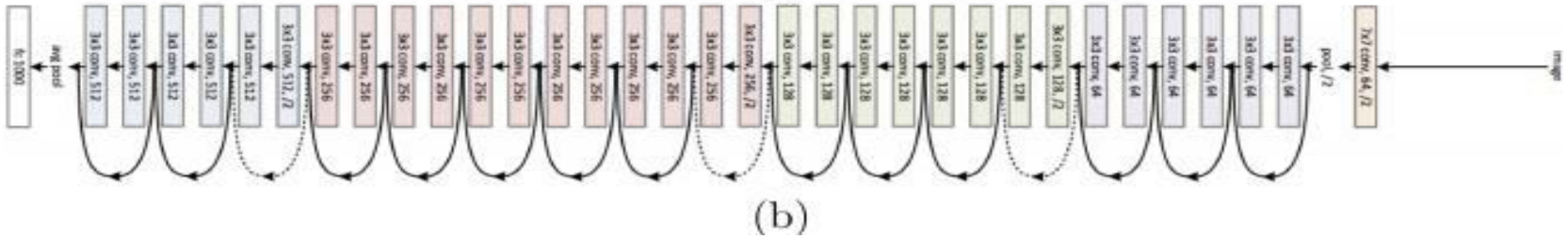
Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

Deep Residual Learning for Image Recognition

Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun
Microsoft Research
{kahe, v-xiangz, v-shren, jiansun}@microsoft.com

Residual Net





Deep Residual Learning for Image Recognition

Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun
Microsoft Research
{kahe, v-xiangz, v-shren, jiansun}@microsoft.com

DenseNet

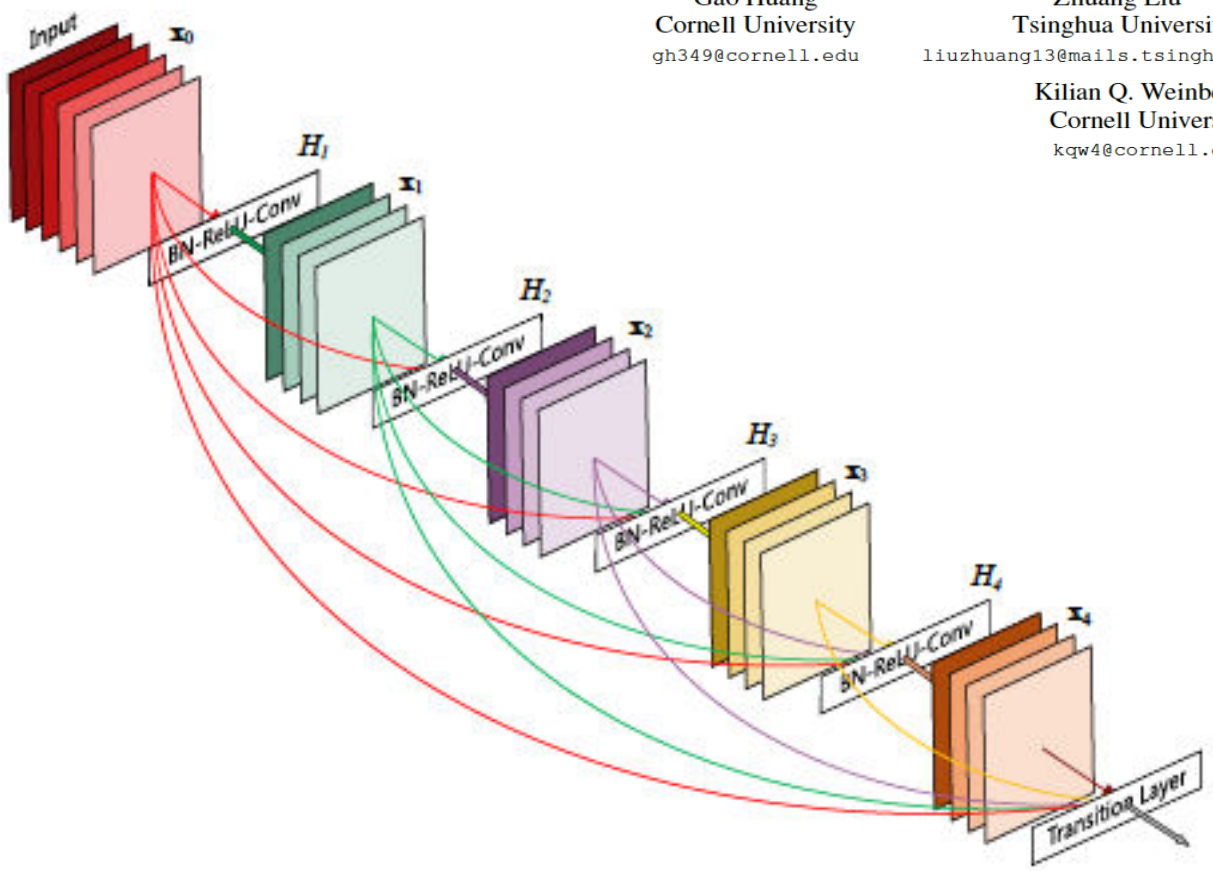
Densely Connected Convolutional Networks

Gao Huang*
Cornell University
gh349@cornell.edu

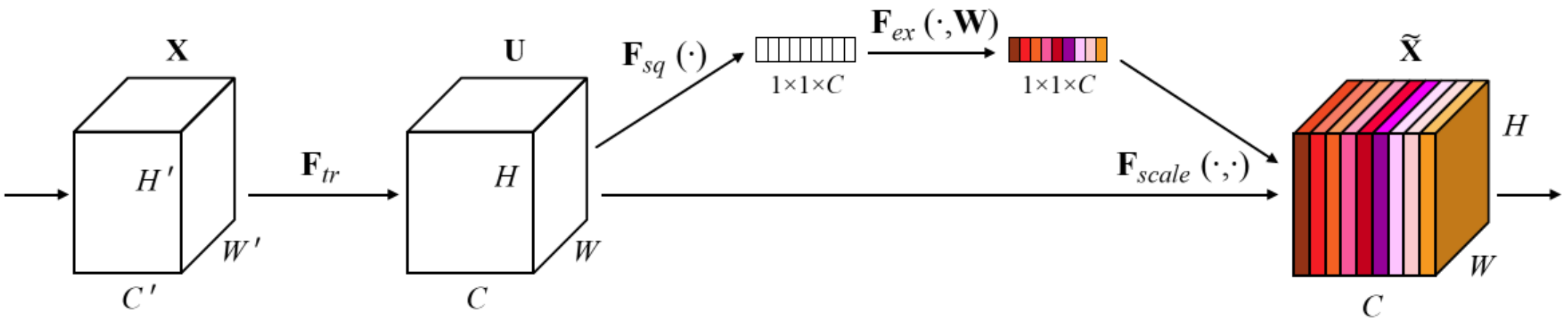
Zhuang Liu*
Tsinghua University
liuzhuang13@mails.tsinghua.edu.cn

Laurens van der Maaten
Facebook AI Research
lvdmaaten@fb.com

Kilian Q. Weinberger
Cornell University
kqw4@cornell.edu



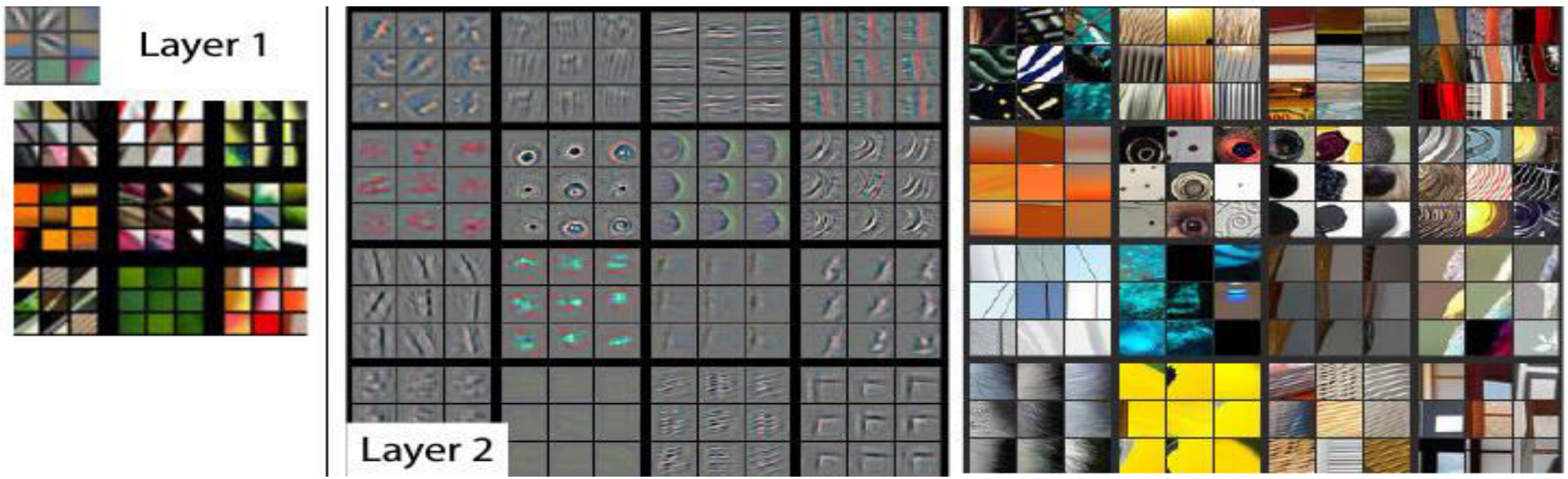
SENet — Squeeze-and-Excitation Network

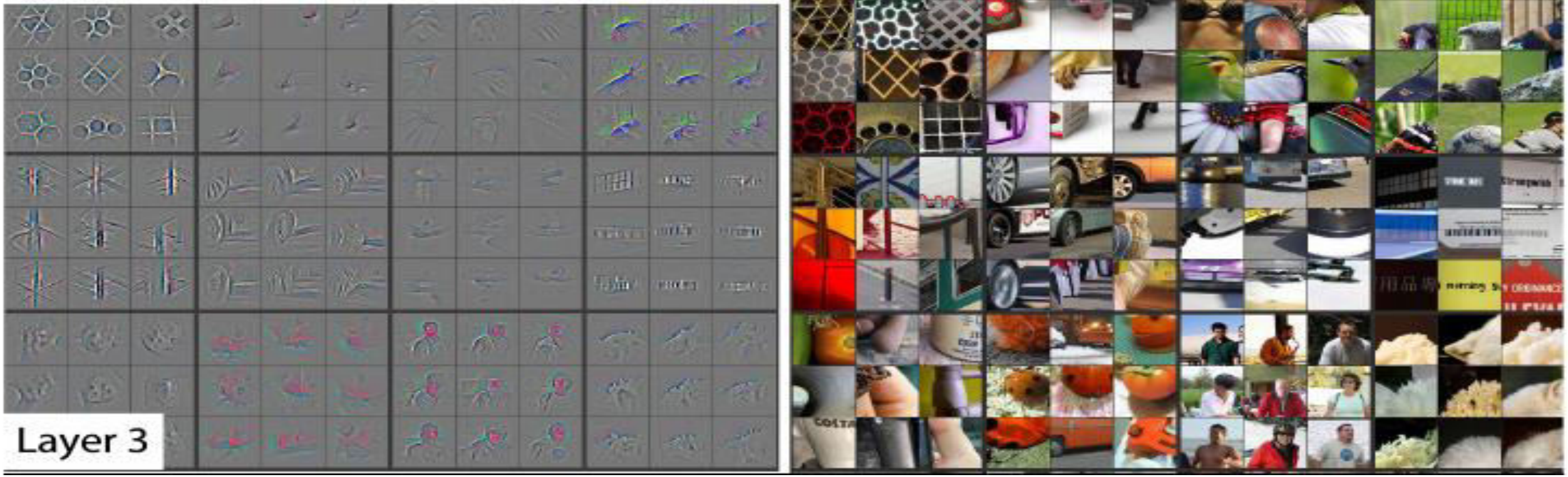


Visualizing and Understanding Convolutional Networks

Matthew D. Zeiler ZEILER@CS.NYU.EDU
Dept. of Computer Science, Courant Institute, New York University

Rob Fergus FERGUS@CS.NYU.EDU
Dept. of Computer Science, Courant Institute, New York University




Layer 3

Regularization

Regularization

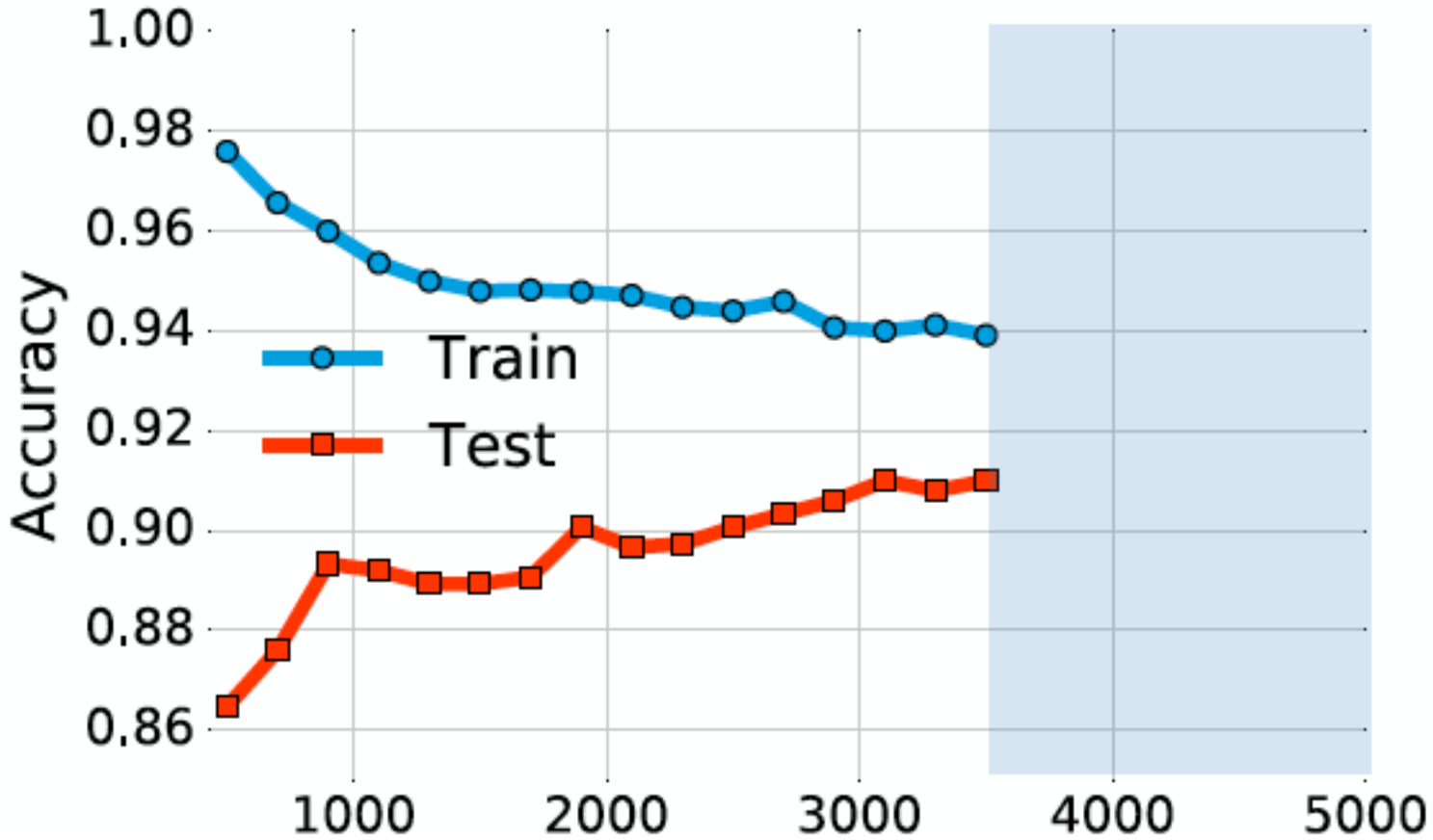
Goal: reduce overfitting

- usually achieved by reducing model capacity and/or reduction of
- the variance of the predictions

Regularization

- Early stopping
- L1/L2 regularization (norm penalties)
- Dropout

Best Way to Reduce Overfitting is Collecting More Data



Best Way to Reduce Overfitting is Collecting More Data

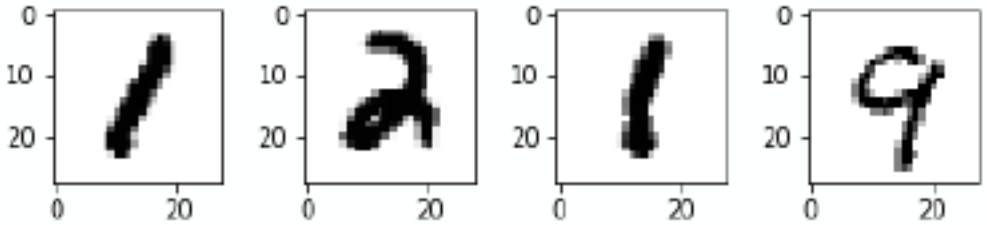
- Collecting more data is always recommended
- If not possible, data augmentation is also helpful (e.g., for images: random rotation, crop, translation ...) -- actually, this is always recommended (and easy to do)
- Additionally, reducing the capacity (e.g., regularization) helps

Data Augmentation in PyTorch via TorchVision

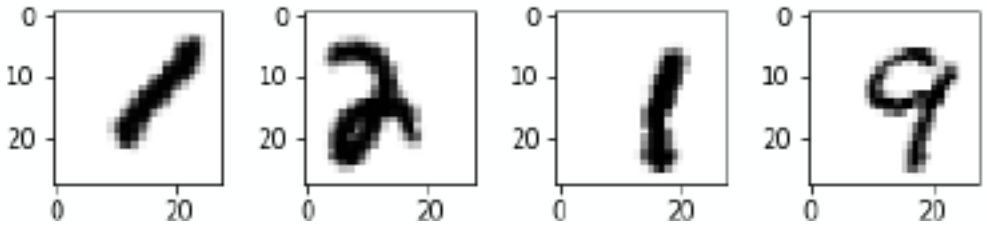
```
training_transforms = torchvision.transforms.Compose([
    #torchvision.transforms.RandomRotation(degrees=20),
    #torchvision.transforms.Resize(size=(34, 34)),
    #torchvision.transforms.RandomCrop(size=(28, 28)),
    torchvision.transforms.RandomAffine(degrees=(-20, 20), translate=(0.15, 0.15),
                                        resample=PIL.Image.BILINEAR),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5)),
    # normalize does (x_i - mean) / std
    # if images are [0, 1], they will be [-1, 1] afterwards
])

test_transforms = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5)),
])

# for more see
# https://pytorch.org/docs/stable/torchvision/transforms.html

train_dataset = datasets.MNIST(root='data',
                               train=True,
                               transform=training_transforms,
                               download=True)

test_dataset = datasets.MNIST(root='data',
                              train=False,
                              transform=test_transforms)
```

Data Augmentation in PyTorch via TorchVision

Original



Augmented



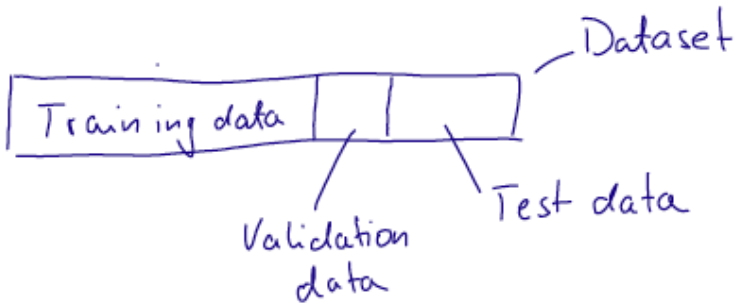
Reducing Network's Capacity

- smaller architecture: fewer hidden layers & units, dropout, (dead ReLUs, L1 norm penalty)
- smaller weights: Early stopping, norm penalties
- adding noise: Dropout

Early Stopping

Step 1: Split your dataset into 3 parts (always recommended)

- use test set only once at the end (for unbiased estimate of generalization performance)
- use validation accuracy for tuning (always recommended)



Step 2: Early stopping (not very common anymore)

- reduce overfitting by observing the training/validation accuracy gap during training and then stop at the "right" point

L1/L2 Regularization

$$\text{Cost}_{\mathbf{w},\mathbf{b}} = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(y^{[i]}, \hat{y}^{[i]})$$

$$\text{L2-Regularized-Cost}_{\mathbf{w},\mathbf{b}} = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(y^{[i]}, \hat{y}^{[i]}) + \frac{\lambda}{n}\sum_{j} w_j^2$$

where: $\sum_{j} w_j^2 = ||\mathbf{w}||_2^2$

and λ is a hyperparameter

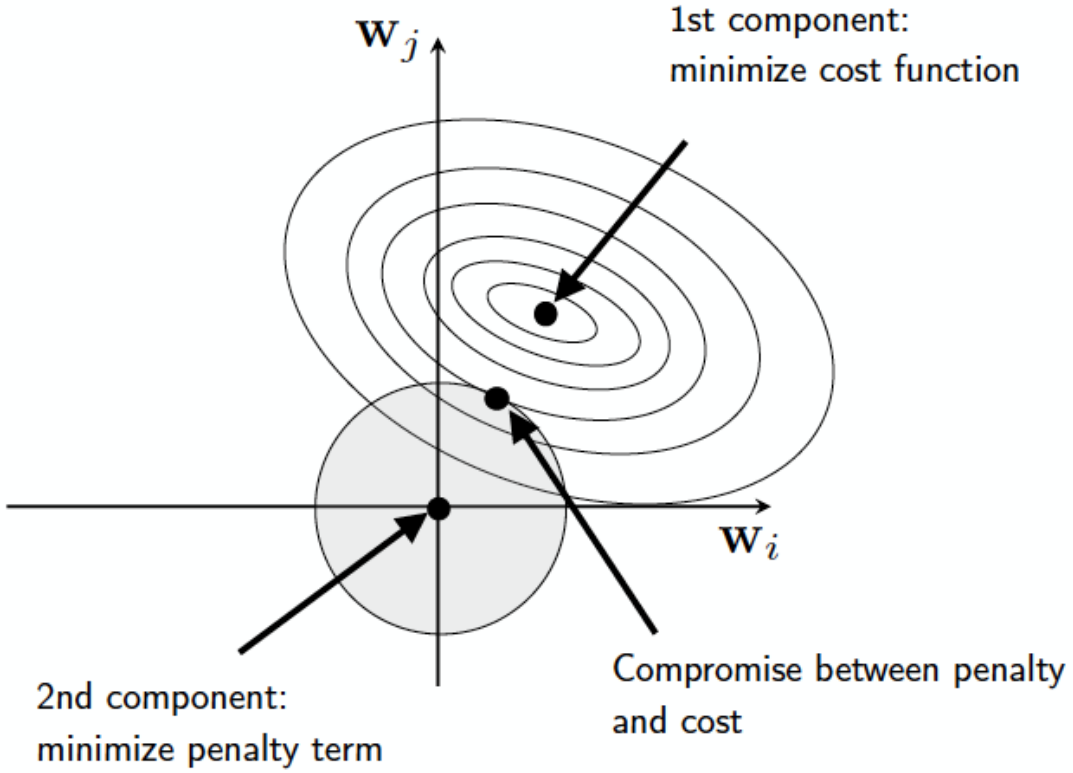
L1/L2 Regularization

$$\text{L1-Regularized-Cost}_{\mathbf{w},\mathbf{b}} = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(y^{[i]}, \hat{y}^{[i]}) + \frac{\lambda}{n}\sum_{j}|w_j|$$

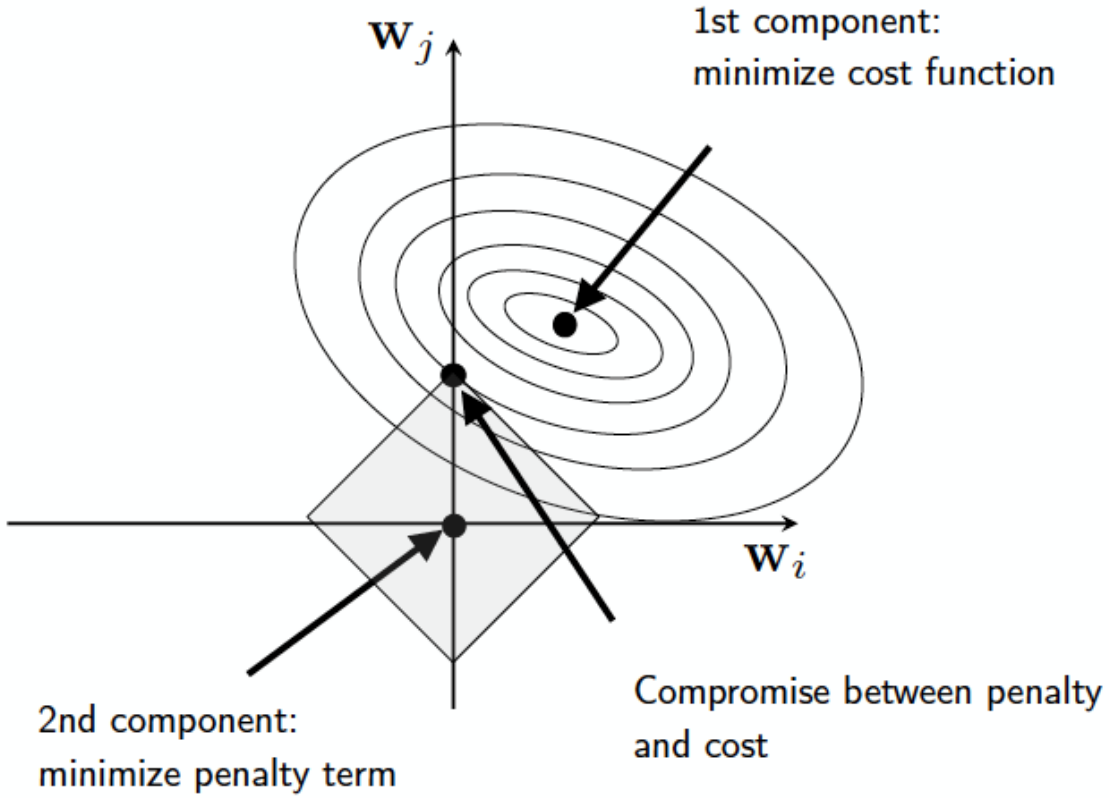
where: $\sum_{j}|w_j| = ||\mathbf{w}||_1$

- L1-regularization encourages sparsity (which may be useful)
- However, usually L1 regularization does not work well in practice and is very rarely used
- Also, it's not smooth and harder to optimize

Geometric Interpretation of L2 Regularization



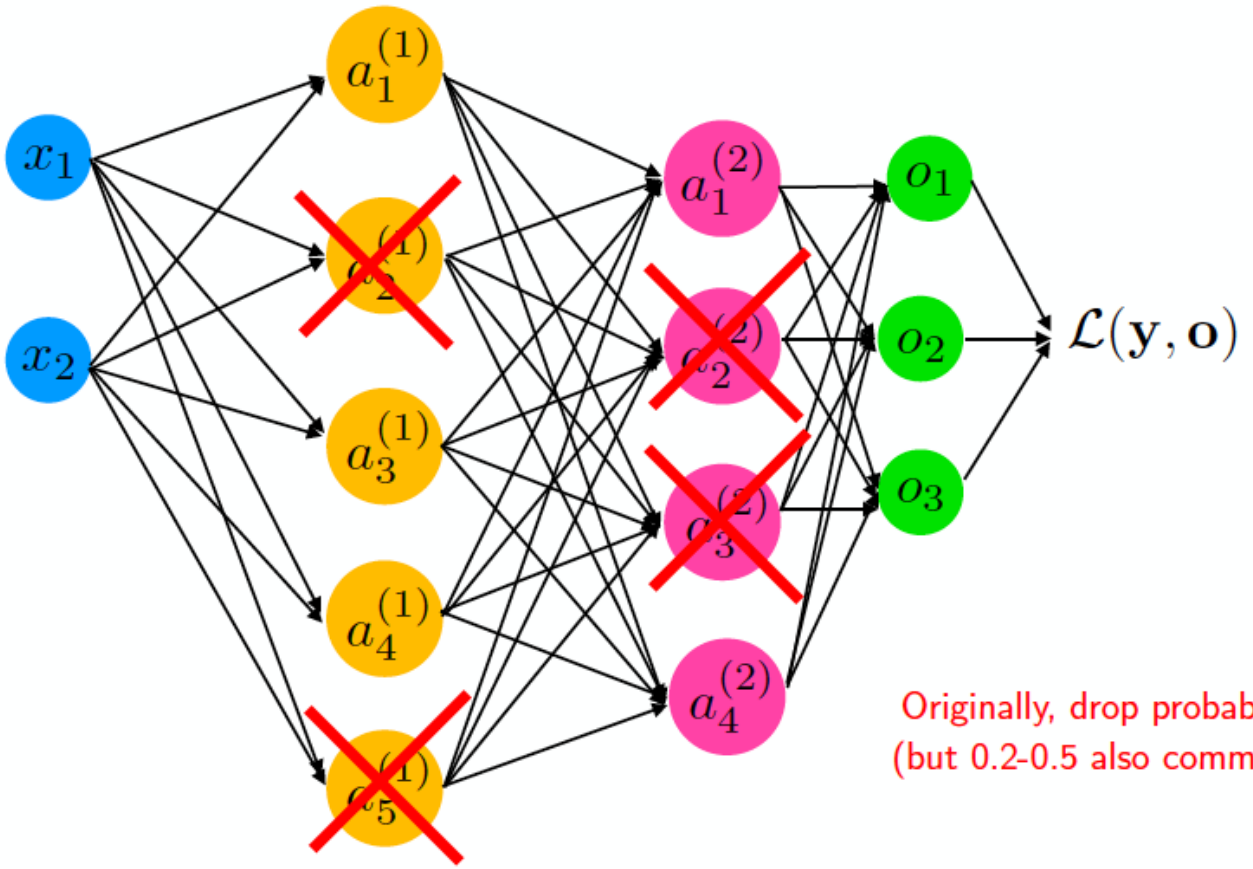
Geometric Interpretation of L1 Regularization



L2 Regularization for Logistic Regression in PyTorch

```
optimizer = torch.optim.SGD(model.parameters(),
                            lr=0.1,
                            weight_decay=LAMBDA)
#--------------------------------------------------


for epoch in range(num_epochs):

    #### Compute outputs ####
    out = model(X_train_tensor)

    #### Compute gradients ####
    cost = F.binary_cross_entropy(out, y_train_tensor, reduction='sum')
    optimizer.zero_grad()
    cost.backward()
```

Dropout in a Nutshell: Dropping Nodes



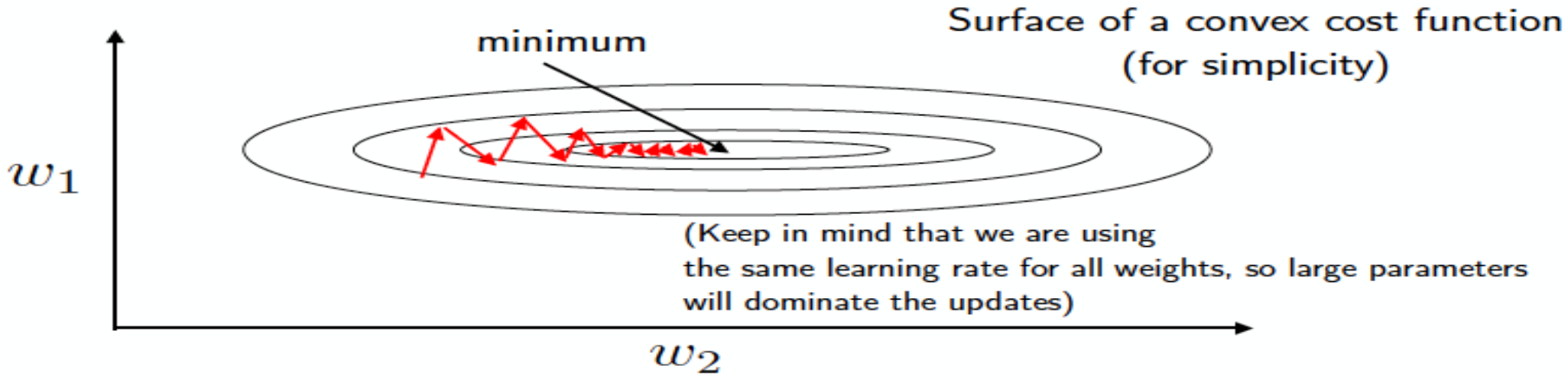
Dropout

Why does Dropout work well?

- Network will learn not to rely on particular connections too heavily
- Thus, will consider more connections (because it cannot rely on individual ones)
- The weight values will be more spread-out (may lead to smaller weights like with L2 norm)
- Side note: You can certainly use different dropout probabilities in different layers (assigning them proportional to the number of units in a layer is not a bad idea, for example)

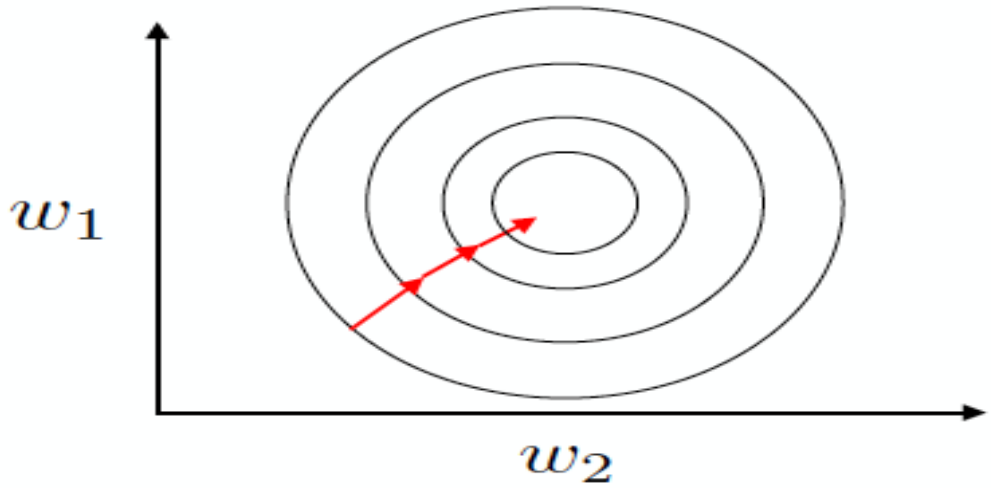
Features Normalization

Why We Normalize Inputs for Gradient Descent



Surface of a convex cost function
(for simplicity)

(Keep in mind that we are using
the same learning rate for all weights, so large parameters
will dominate the updates)



"Standardization" of input features

$$x_j'^{[i]} = \frac{x_j^{[i]} - \mu_j}{\sigma_j}$$

(scaled feature will have
zero mean, unit variance)

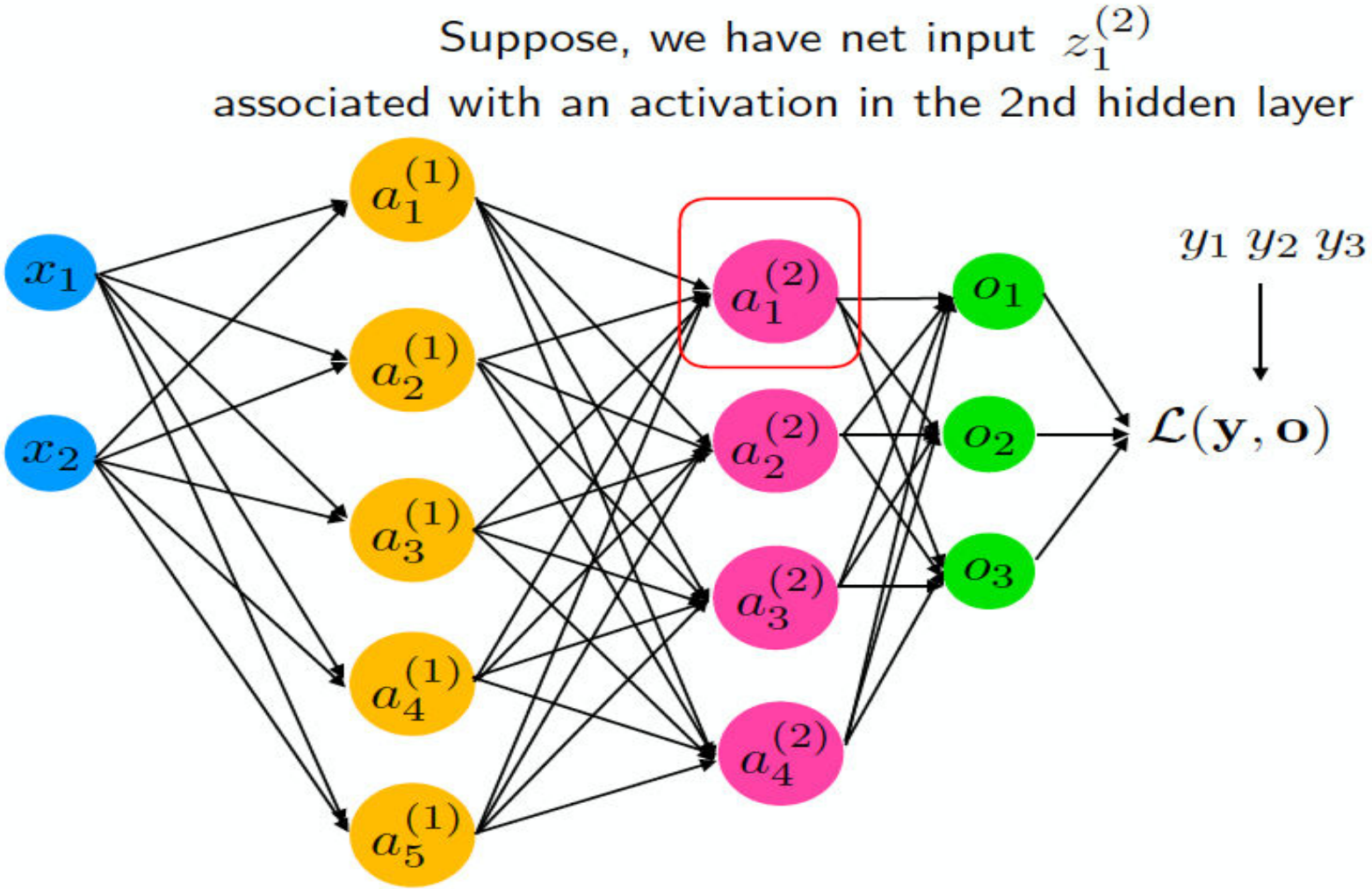
Batch Normalization

Ioffe, S., & Szegedy, C. (2015, June). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In International Conference on Machine Learning (pp. 448-456). http://proceedings.mlr.press/v37/ioffe15.html

Batch Normalization

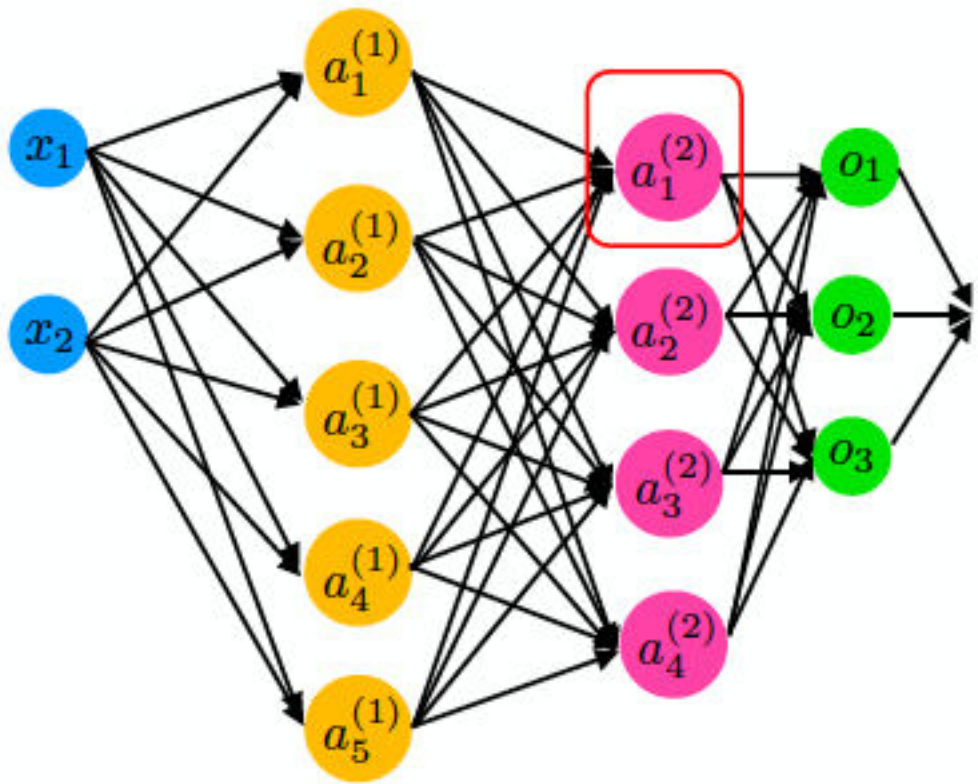
- Normalization of inputs for hidden layers
- Helps with exploding/vanishing gradient problems
- Can increase training stability and convergence rate
- Can be understood as additional normalization layers (with additional parameters)

Batch Normalization



Now, consider all examples in a minibatch such that the net input of a given training example
at layer 2 is written as $z_1^{(2)[i]}$

where $i \in \{1, \ldots, n\}$



BatchNorm Step 1: Normalize Net Inputs

$$\mu_j = \frac{1}{n}\sum_i z_j^{[i]}$$

$$\sigma_j^2 = \frac{1}{n}\sum_i (z_j^{[i]} - \mu_j)^2$$

$$z'^{[i]}_j = \frac{z_j^{[i]} - \mu_j}{\sigma_j}$$

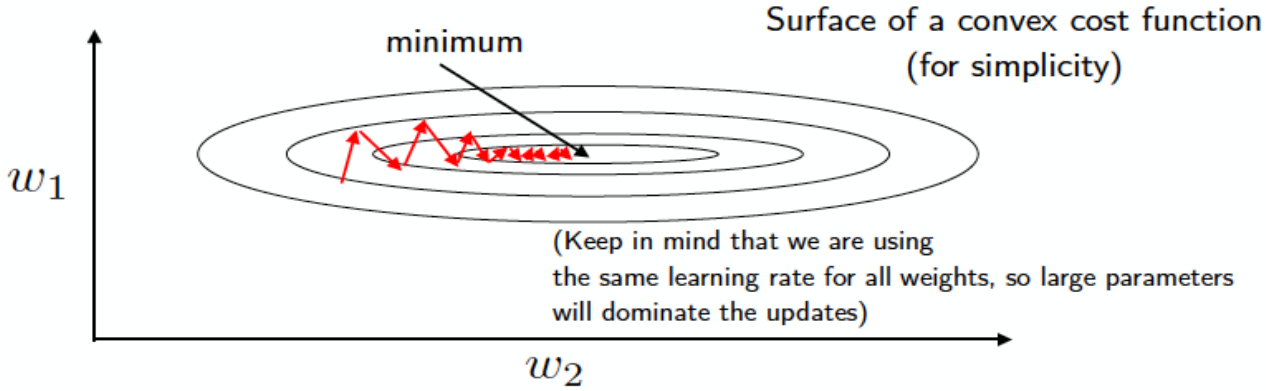
BatchNorm Step 2: Pre-Activation Scaling

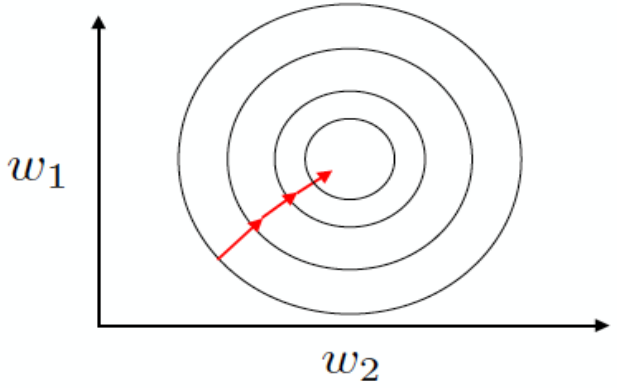
$$z'^{[i]}_j = \frac{z^{[i]}_j - \mu_j}{\sqrt{\sigma^2_j + \epsilon}}$$

$$a'^{[i]}_j = \gamma_j \cdot z'^{[i]}_j + \beta_j$$

Weight Initialization

- we want to initialize weight to small, random numbers to break symmetry



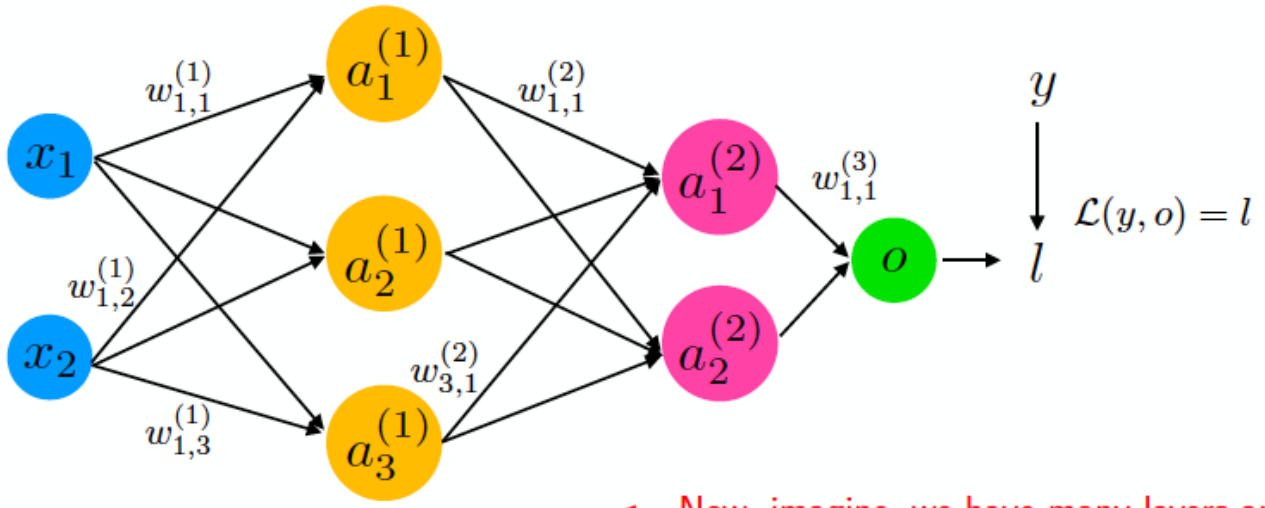


"Standardization" of input features

$$x_j'^{[i]} = \frac{x_j^{[i]} - \mu_j}{\sigma_j}$$

(scaled feature will have
zero mean, unit variance)

Vanishing/Exploding Gradient problems



Now, imagine, we have many layers and sigmoid activations ...

$$\frac{\partial l}{\partial w_{1,1}^{(1)}} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1^{(2)}} \cdot \frac{\partial a_1^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}} + \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2^{(2)}} \cdot \frac{\partial a_2^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}}$$

Vanishing/Exploding Gradient problems

$$\frac{\partial l}{\partial w_{1,1}^{(1)}} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1^{(2)}} \cdot \frac{\partial a_1^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}} + \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2^{(2)}} \cdot \frac{\partial a_2^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}}$$

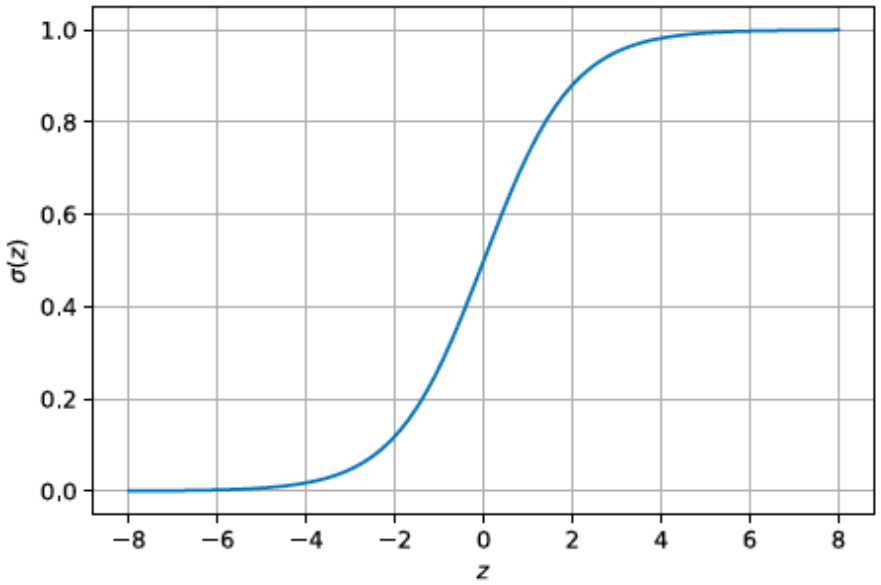
Now, imagine, we have many layers and logistic sigmoid activations ...

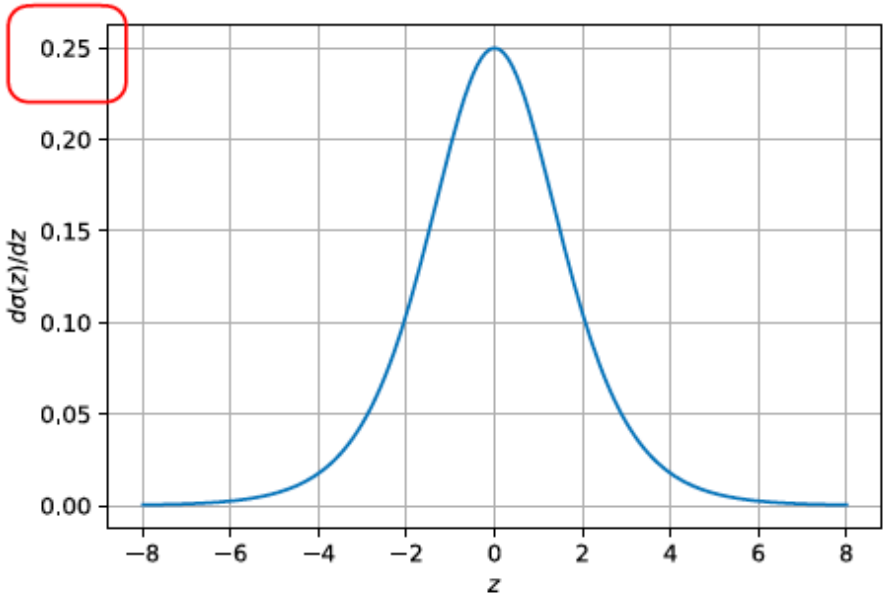
$$\sigma'(z^{[i]}) = \sigma(z^{[i]}) \cdot (1 - \sigma(z^{[i]}))$$

Vanishing/Exploding Gradient problems

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

$$\frac{d}{dz}\sigma(z) = \frac{e^{-z}}{(1+e^{-z})^2} = \sigma(z)(1-\sigma(z))$$





Vanishing/Exploding Gradient problems

Assume, we have the largest gradient:

$$\frac{d}{dz}\sigma(0.0) = \sigma(0.0)(1 - \sigma(0.0)) = 0.25$$

Even then, for, e.g., 10 layers, we degrade the other gradients substantially!

$$0.25^{10} \approx 10^{-6}$$

Weight Initialization

- Traditionally, we can initialize weights by sampling from a random uniform distribution in range [0, 1], or better, [-0.5, 0.5]
- Or, we could sample from a Gaussian distribution with mean 0 and small variance (e.g., 0.1 or 0.01)

Custom Weight Initialization in PyTorch

```
class MLP(torch.nn.Module):

    def __init__(self, num_features, num_hidden, num_classes):
        super(MLP, self).__init__()

        self.num_classes = num_classes

        ### 1st hidden layer
        self.linear_1 = torch.nn.Linear(num_features, num_hidden)
        self.linear_1.weight.detach().normal_(0.0, 0.1)
        self.linear_1.bias.detach().zero_()

        ### Output layer
        self.linear_out = torch.nn.Linear(num_hidden, num_classes)
        self.linear_out.weight.detach().normal_(0.0, 0.1)
        self.linear_out.bias.detach().zero_()

    def forward(self, x):
        out = self.linear_1(x)
        out = torch.sigmoid(out)
        logits = self.linear_out(out)
        probas = torch.sigmoid(logits)
        return logits, probas
```

Weight Initialization -- Xavier Initialization

- Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks."Proceedings of the thirteenth international conference on artificial intelligence and statistics. 2010.
- TanH is a bit more robust regarding vanishing gradients (compared to logistic sigmoid)
- It still has the problem of saturation (near zero gradients if inputs are very large, positive or negative values)
- Xavier initialization is a small improvement for initializing weights for tanH

Weight Initialization -- Xavier Initialization

Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010.

Method:

Step 1: Initialize weights from Gaussian or uniform distribution with (previous slide)

Step 2: Scale the weights proportional to the number of inputs to the layer

(For the first hidden layer, that is the number of features in the dataset;
for the second hidden layer, that is the number of units in the 1st hidden layer
etc.)

Weight Initialization -- Xavier Initialization

Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010.

Method:

Scale the weights proportional to the number of inputs to the layer

In particular, scale as follows:

$$\mathbf{W}^{(l)} := \mathbf{W}^{(l)} \cdot \sqrt{\frac{1}{m^{(l-1)}}}$$

where m is the number of input units to the next layer

e.g.,

$$W_{i,j}{}^{(l)} \sim N(\mu = 0, \sigma^2 = 0.01)$$

(or uniform distr. in a fixed interval, as in the original paper)

Weight Initialization -- Xavier Initialization

Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010.

$$\mathrm{Var}\left(a^{(l)}\right) \approx \mathrm{Var}\left(z^{(l)}\right) = \mathrm{Var}\left(z_j^{(l)}\right) = \mathrm{Var}\left(\sum_{i=1}^{m_{l-1}} W_{jk}^{(l)} a_k^{(l-1)}\right)$$

$$= \sum_{i=1}^{m^{(l-1)}} \mathrm{Var}\left[W_{jk}^{(l)} a_k^{(l-1)}\right] = \sum_{i=1}^{m^{(l-1)}} \mathrm{Var}\left[W_{jk}^{(l)}\right] \mathrm{Var}\left[a_k^{(l-1)}\right]$$

$$= \sum_{i=1}^{m^{(l-1)}} \mathrm{Var}\left[W^{(l)}\right] \mathrm{Var}\left[a^{(l-1)}\right] = m^{(l-1)} \mathrm{Var}\left[W^{(l)}\right] \mathrm{Var}\left[a^{(l-1)}\right]$$

Xavier Initialization in PyTorch

Semi-Automatic:

```
...
self.linear = torch.nn.Linear(...)
torch.nn.init.xavier_uniform_(conv1.weight)
...
```

More conveniently for all weights in e.g., fully-connected layers:

```
...
def weights_init(m):
    if isinstance(m, nn.Linear):
        torch.nn.init.xavier_uniform_(m.weight)
        torch.nn.init.xavier_uniform_(m.bias)

model.apply(weights_init)
...
```

Weight Initialization -- Xavier Initialization

$$\mathbf{W}^{(l)} := \mathbf{W}^{(l)} \cdot \sqrt{\frac{1}{m^{[l-1]}}}$$

From the original paper:

We initialized the biases to be 0 and the weights W_{ij} at each layer with the following commonly used heuristic:

$$W_{ij} \sim U\Big[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\Big], \qquad (1)$$

where $U[-a, a]$ is the uniform distribution in the interval $(-a, a)$ and n is the size of the previous layer (the number of columns of W).

Also from the original paper:

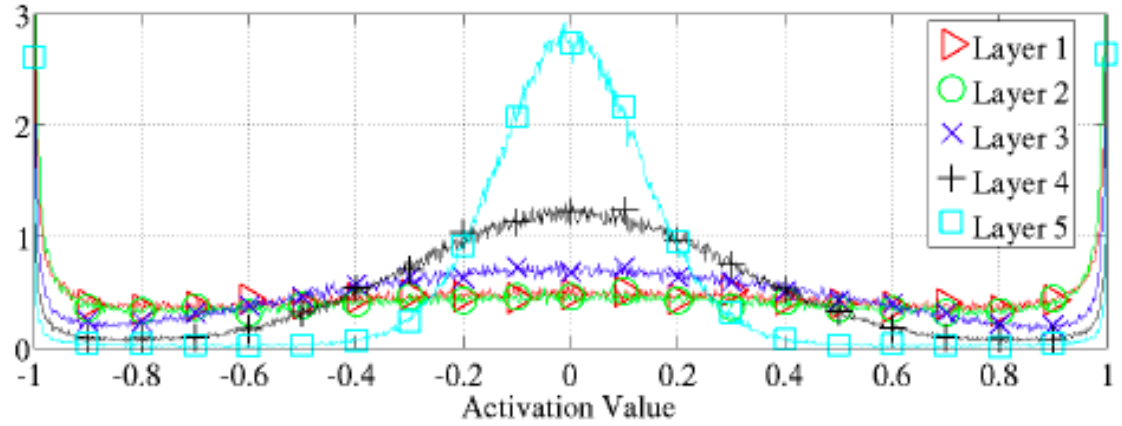
The normalization factor may therefore be important when initializing deep networks because of the multiplicative effect through layers, and we suggest the following initialization procedure to approximately satisfy our objectives of maintaining activation variances and back-propagated gradients variance as one moves up or down the network. We call it the **normalized initialization**:

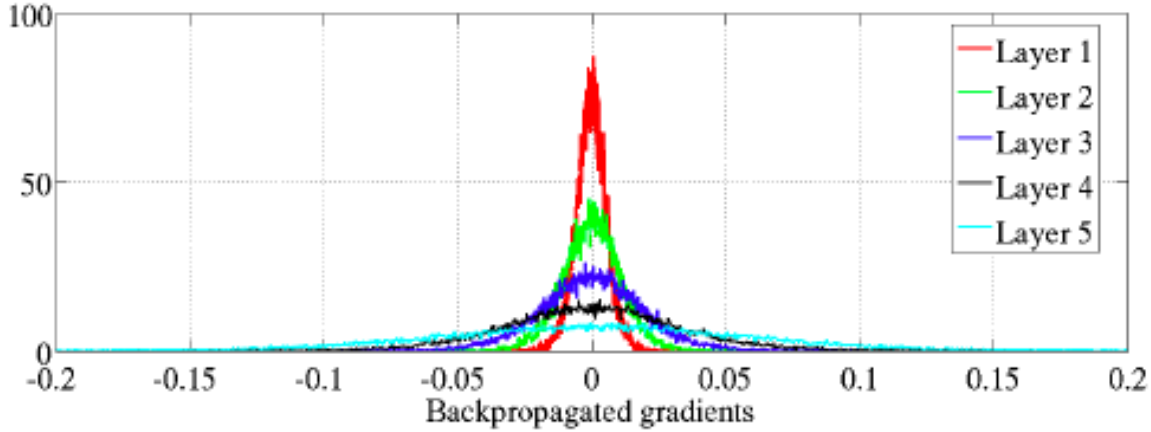
$$W \sim U\Big[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\Big] \qquad (16)$$

Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010.

Weight Initialization -- Xavier Initialization

Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010.





Exploding gradient problem!

Weight Initialization -- Xavier Initialization

Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010.

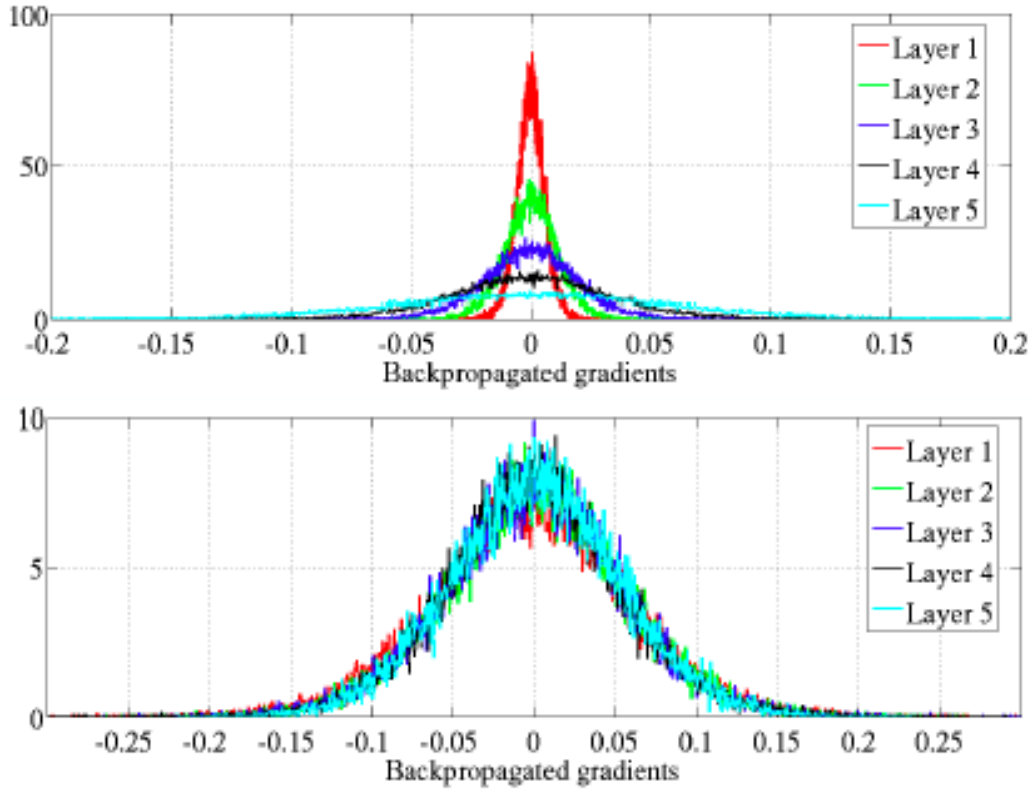


Figure 7: *Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers.*

Weight Initialization -- He Initialization

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." In *Proceedings of the IEEE international conference on computer vision*, pp. 1026-1034. 2015.

- Assuming activations with mean 0, which is reasonable, Xavier Initialization assumes a derivative of 1 for the activation function (which is reasonable for tanH)
- For ReLU, this is different, as the activations are not centered at zero anymore
- He initialization takes this into account (to see that worked out in math, see the paper)
- The result is that we add a scaling factor of $2^{0.5}$

$$\mathbf{W}^{(l)} := \mathbf{W}^{(l)} \cdot \sqrt{\frac{2}{m^{[l-1]}}}$$

Introduction to Recurrent Neural Networks

Application of RNN

- Sequence Modelling
- Text Classification
- Machine Translation
- Stock market prediction
- Speech Recognition
- Image captioning

Before RNN

Bag of words

1) Suppose you want to design a classifier and you have a training dataset consisting of 3 examples (sentences)

$$\mathbf{x}^{[1]} = \text{"The sun is shining"}$$

$$\mathbf{x}^{[2]} = \text{"The weather is sweet"}$$

$$\mathbf{x}^{[3]} = \text{"The sun is shining, the weather is sweet, and one and one is two"}$$

https://github.com/rasbt/python-machine-learning-book-3rd-edition/blob/master/ch08/ch08.ipynb

Before RNN

Bag of words

2) Based on ALL your data, you would construct a vocabulary of all unique words

$\mathbf{x}^{[1]}$ = "The sun is shining"

$\mathbf{x}^{[2]}$ = "The weather is sweet"

$\mathbf{x}^{[3]}$ = "The sun is shining,
the weather is sweet, and one and one is two"

```
vocabulary = {
    'and': 0,
    'is': 1
    'one': 2,
    'shining': 3,
    'sun': 4,
    'sweet': 5,
    'the': 6,
    'two': 7,
    'weather': 8,
}
```

https://github.com/rasbt/python-machine-learning-book-3rd-edition/blob/master/ch08/ch08.ipynb

Before RNN

Bag of words

3) Use the vocabulary to transform the dataset into bag-of-words vectors (vector size is determined by the vocabulary size)

$\mathbf{x}^{[1]} =$ "The sun is shining"

$\mathbf{x}^{[2]} =$ "The weather is sweet"

$\mathbf{x}^{[3]} =$ "The sun is shining, the weather is sweet, and one and

```
vocabulary = {
    'and': 0,
    'is': 1
    'one': 2,
    'shining': 3,
    'sun': 4,
    'sweet': 5,
    'the': 6,
    'two': 7,
    'weather': 8,
}
```

$$\mathbf{X} = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 2 & 3 & 2 & 1 & 1 & 1 & 2 & 1 & 1 \end{bmatrix}$$

https://github.com/rasbt/python-machine-learning-book-3rd-edition/blob/master/ch08/ch08.ipynb

Before RNN
Bag of words

$$\mathbf{X} = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 2 & 3 & 2 & 1 & 1 & 1 & 2 & 1 & 1 \end{bmatrix}$$

Rows are training examples

Columns are features

Feature Extraction from words

- Word count
- One hot Encoding
- TF-IDF
- Embeddings(word2vec, glove etc)

One Hot Encoding

$\mathbf{x}^{[1]}$ = "The sun is shining"

$\mathbf{x}^{[2]}$ = "The weather is sweet"

$\mathbf{x}^{[3]}$ = "The sun is shining,
the weather is sweet, and one and one is two"

```
vocabulary = {
    'and': 0,
    'is': 1
    'one': 2,
    'shining': 3,
    'sun': 4,
    'sweet': 5,
    'the': 6,
    'two': 7,
    'weather': 8,
}
```

TF_IDF

- *TF = (Number of repetitions of word in a document) / (# of words in a document)*
- *IDF =Log[(# Number of documents) / (Number of documents containing the word)]*
- *Tf-idf=TF*IDF*

TF-IDF

$\mathbf{x}^{[1]} =$ "The sun is shining"

$\mathbf{x}^{[2]} =$ "The weather is sweet"

$\mathbf{x}^{[3]} =$ "The sun is shining,
the weather is sweet, and one and one is two"

Token	Sen-1	Sen-2	Sen-3	TF(Sen-1)	TF(Sen-2)	TF(Sen-3)
and	0	0	2	0	0	0.14285714
is	1	1	3	0.25	0.25	0.21428571
one	0	0	2	0	0	0.14285714
shining	1	0	1	0.25	0	0.07142857
sun	1	0	1	0.25	0	0.07142857
sweet	0	1	1	0	0.25	0.07142857
the	1	1	2	0.25	0.25	0.14285714
two	0	0	1	0	0	0.07142857
weather	0	1	1	0	0.25	0.07142857

Token	Sen-1	Sen-2	Sen-3	Word Freq in Doc	IDF
and	0	0	2	2	0.17609126
is	1	1	3	5	-0.2218487
one	0	0	2	2	0.17609126
shining	1	0	1	2	0.17609126
sun	1	0	1	2	0.17609126
sweet	0	1	1	2	0.17609126
the	1	1	2	4	-0.1249387
two	0	0	1	1	0.47712125
weather	0	1	1	2	0.17609126

TF-IDF

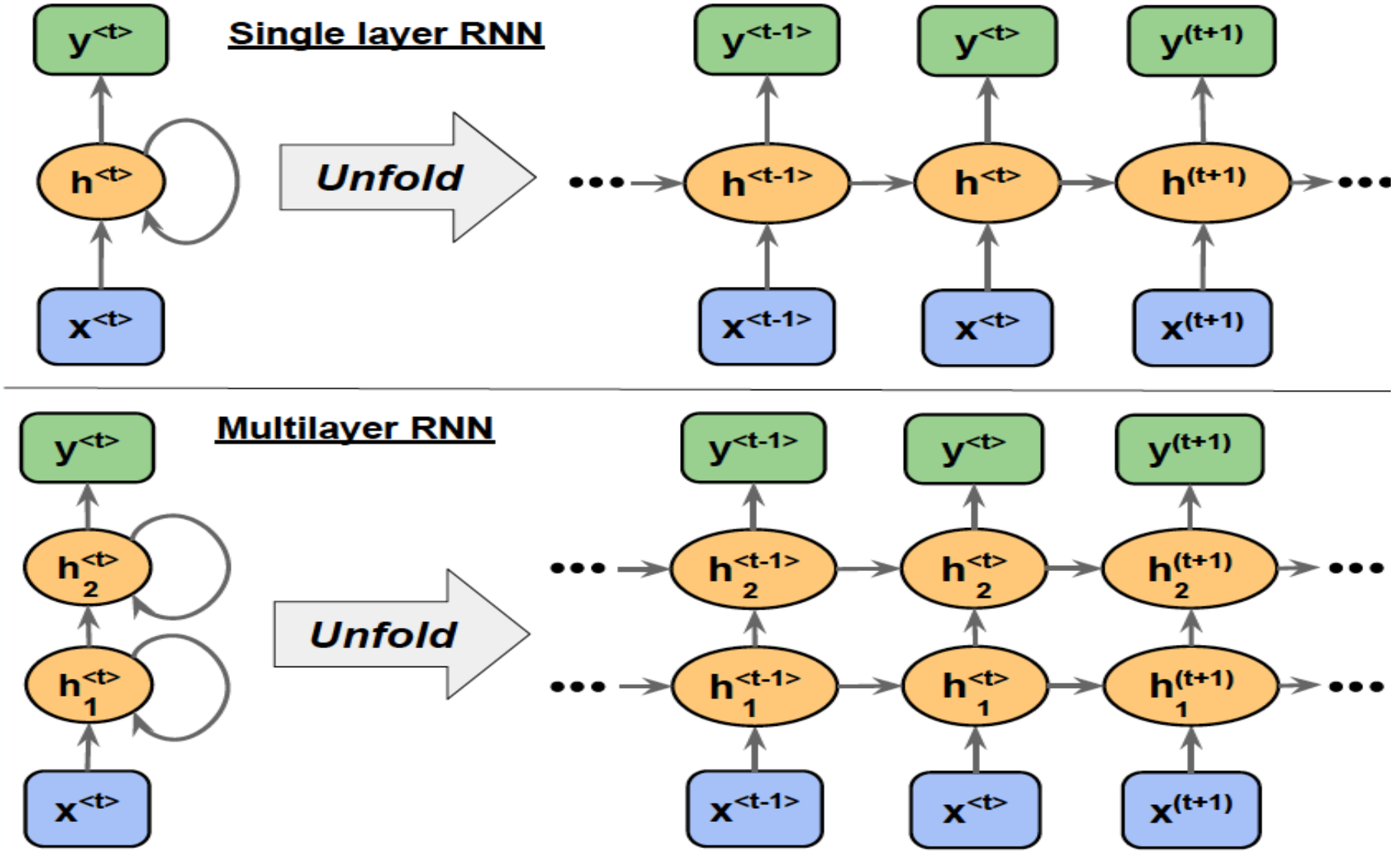
$\mathbf{x}^{[1]}$ = "The sun is shining"

$\mathbf{x}^{[2]}$ = "The weather is sweet"

$\mathbf{x}^{[3]}$ = "The sun is shining,
the weather is sweet, and one and one is two"

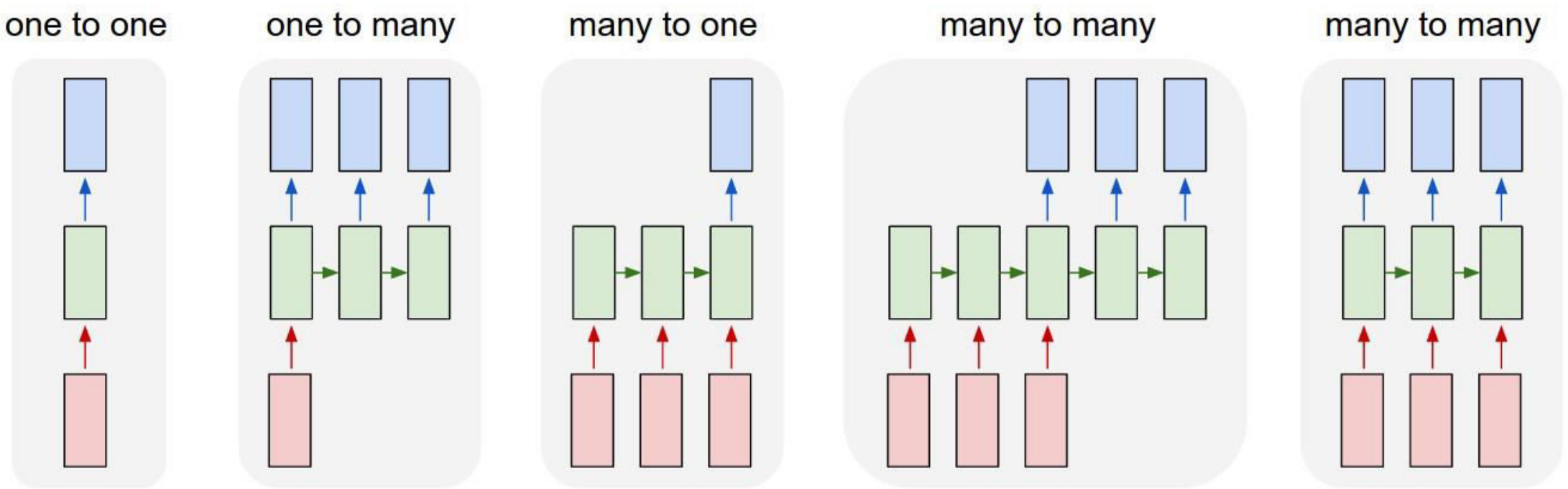
Token	Sen-1	Sen-2	Sen-3	TF(Sen-1)	TF(Sen-2)	TF(Sen-3)	IDF	TF-IDF(Sen-1)	TF-IDF(Sen-2)	TF-IDF(Sen-3)
and	0	0	2	0	0	0.14285714	0.176091259	0	0	0.025155894
is	1	1	3	0.25	0.25	0.21428571	-0.22184875	-0.055462187	-0.055462187	-0.047539018
one	0	0	2	0	0	0.14285714	0.176091259	0	0	0.025155894
shining	1	0	1	0.25	0	0.07142857	0.176091259	0.044022815	0	0.012577947
sun	1	0	1	0.25	0	0.07142857	0.176091259	0.044022815	0	0.012577947
sweet	0	1	1	0	0.25	0.07142857	0.176091259	0	0.044022815	0.012577947
the	1	1	2	0.25	0.25	0.14285714	-0.124938737	-0.031234684	-0.031234684	-0.017848391
two	0	0	1	0	0	0.07142857	0.477121255	0	0	0.03408009
weather	0	1	1	0	0.25	0.07142857	0.176091259	0	0.044022815	0.012577947

Introduction



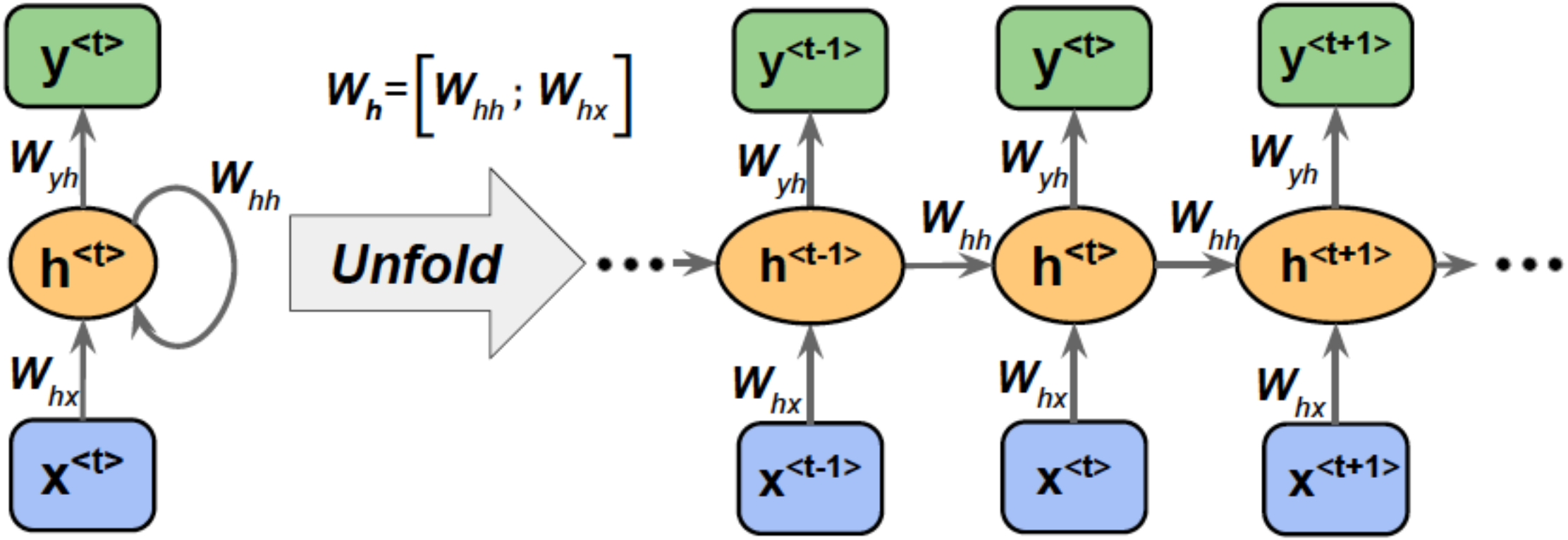
https://sebastianraschka.com/pdf/lecture-notes/stat479ss19/L14_intro-rnn-part2_slides.pdf

Types of RNN



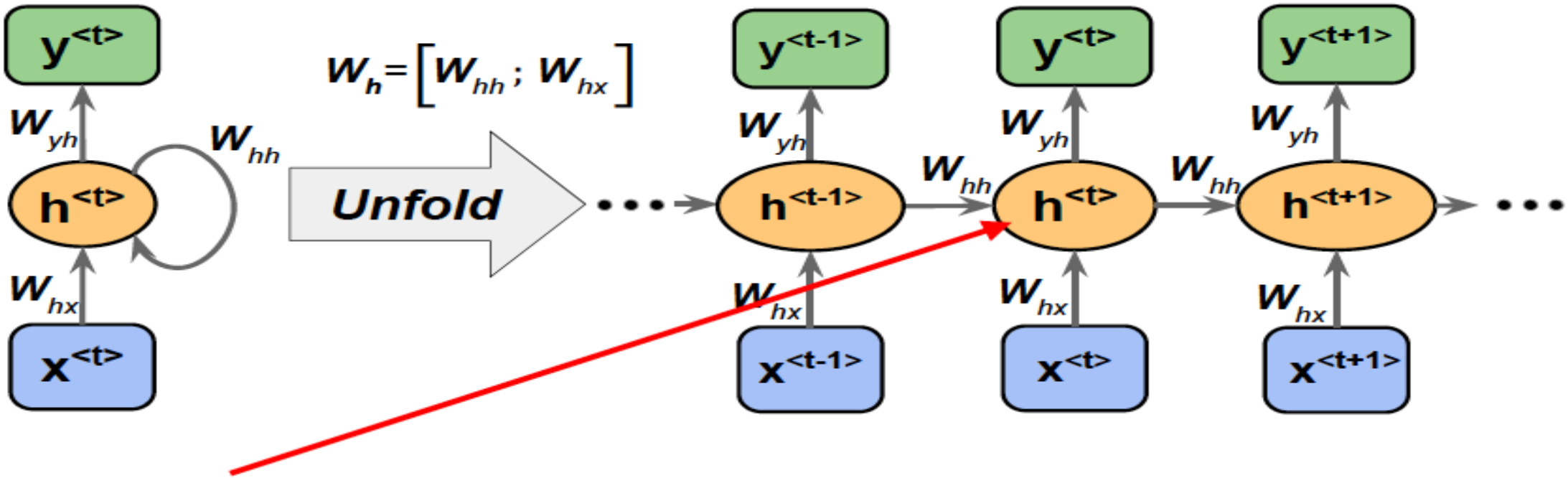
http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Weight Matrix



https://sebastianraschka.com/pdf/lecture-notes/stat479ss19/L14_intro-rnn-part2_slides.pdf

Weight Matrix



Net input:

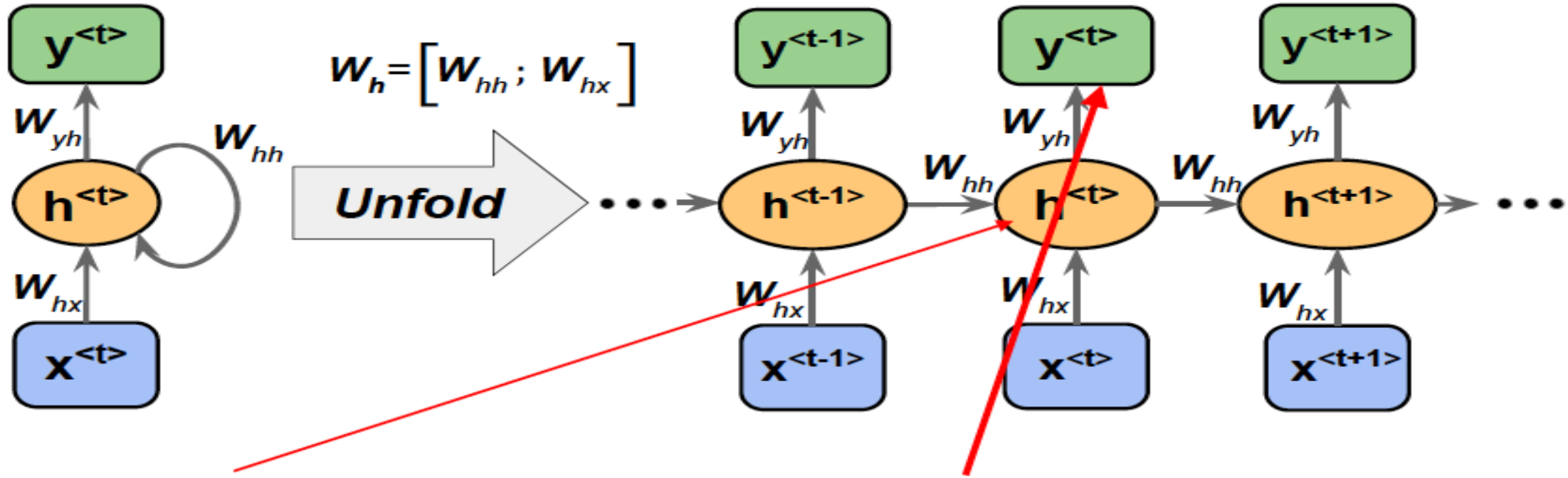
$$\mathbf{z}_h^{\langle t \rangle} = \mathbf{W}_{hx}\mathbf{x}^{\langle t \rangle} + \mathbf{W}_{hh}\mathbf{h}^{\langle t-1 \rangle} + \mathbf{b}_h$$

Activation:

$$\mathbf{h}^{\langle t \rangle} = \sigma_h\left(\mathbf{z}_h^{\langle t \rangle}\right)$$

https://sebastianraschka.com/pdf/lecture-notes/stat479ss19/L14_intro-rnn-part2_slides.pdf

Weight Matrix



Net input:

$$\mathbf{z}_h^{\langle t \rangle} = \mathbf{W}_{hx}\mathbf{x}^{\langle t \rangle} + \mathbf{W}_{hh}\mathbf{h}^{\langle t-1 \rangle} + \mathbf{b}_h$$

Activation:

$$\mathbf{h}^{\langle t \rangle} = \sigma_h\left(\mathbf{z}_h^{\langle t \rangle}\right)$$

Net input:

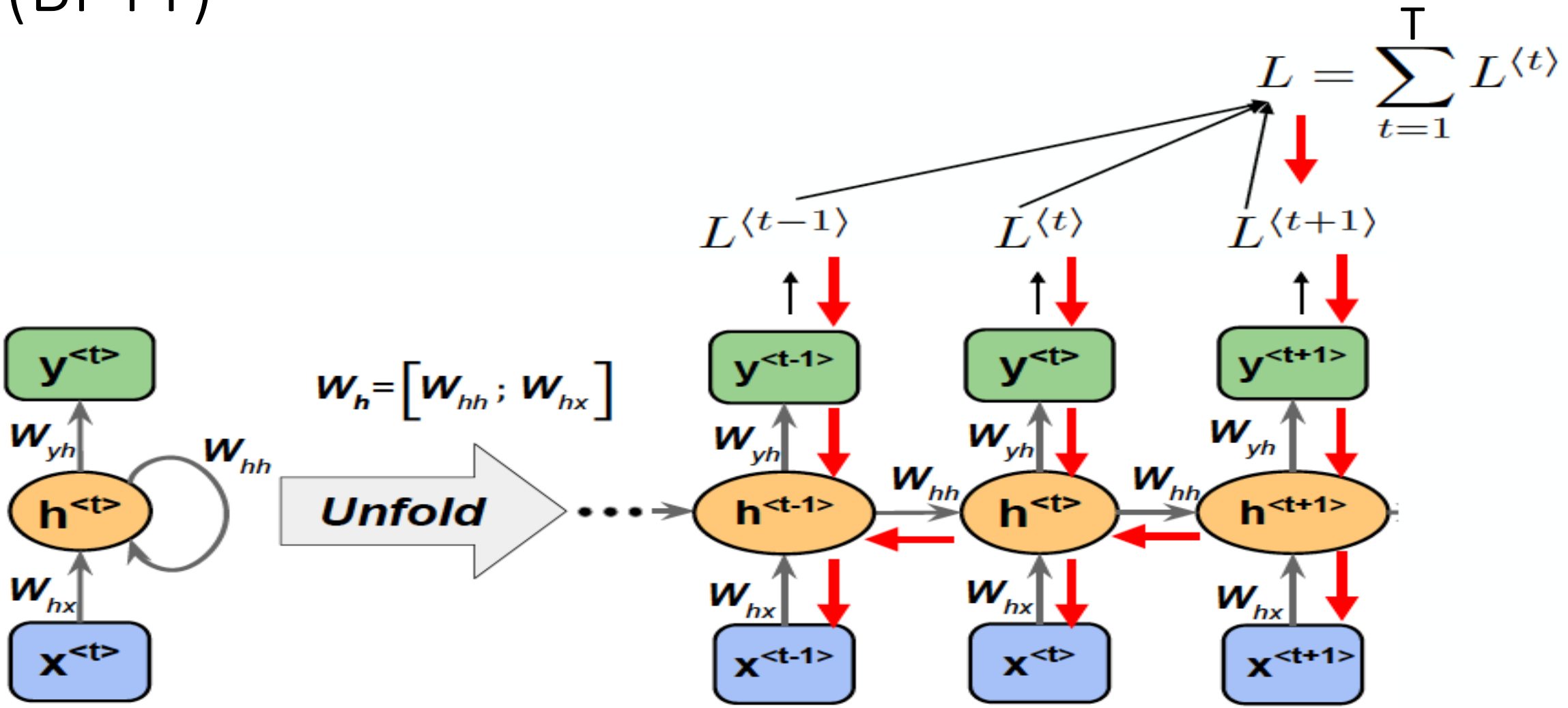
$$\mathbf{z}_y^{\langle t \rangle} = \mathbf{W}_{yh}\mathbf{h}^{\langle t \rangle} + \mathbf{b}_y$$

Output:

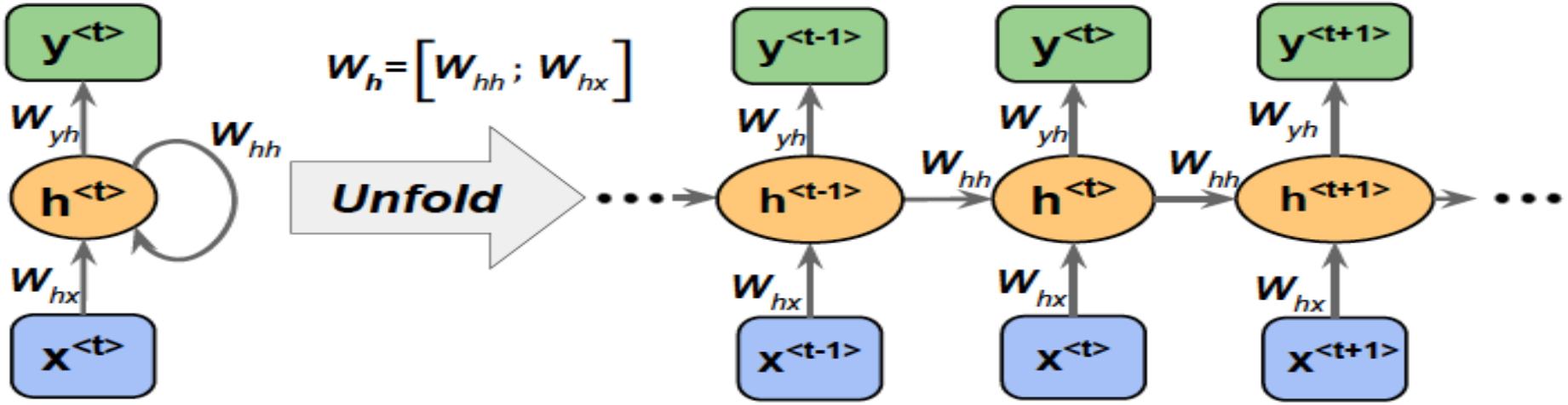
$$\mathbf{y}^{\langle t \rangle} = \sigma_y\left(\mathbf{z}_y^{\langle t \rangle}\right)$$

https://sebastianraschka.com/pdf/lecture-notes/stat479ss19/L14_intro-rnn-part2_slides.pdf

Backpropagation through time (BPTT)



Backpropagation through time (BPTT)



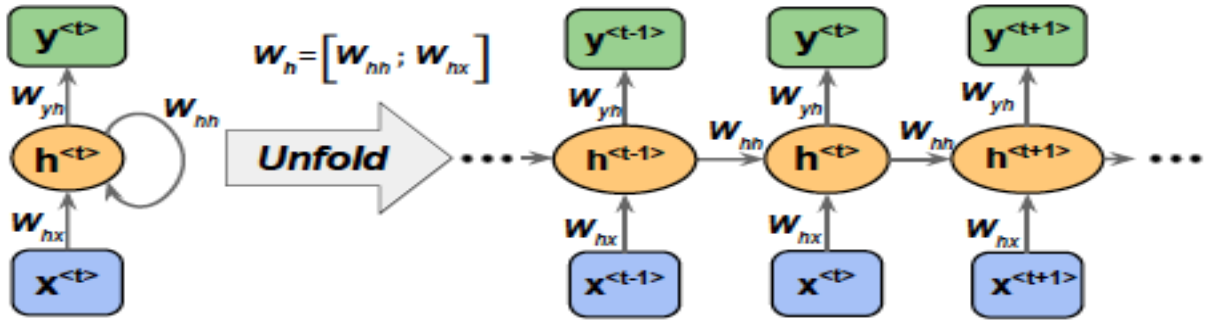
Werbos, Paul J. "Backpropagation through time: what it does and how to do it." *Proceedings of the IEEE* 78, no. 10 (1990): 1550-1560.

$$L = \sum_{t=1}^{T} L^{(t)}$$

$$\frac{\partial L^{(t)}}{\partial \mathbf{W}_{hh}} = \frac{\partial L^{(t)}}{\partial y^{(t)}} \cdot \frac{\partial y^{(t)}}{\partial \mathbf{h}^{(t)}} \cdot \left(\sum_{k=1}^{t} \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} \cdot \frac{\partial \mathbf{h}^{(k)}}{\partial \mathbf{W}_{hh}} \right)$$

https://sebastianraschka.com/pdf/lecture-notes/stat479ss19/L14_intro-rnn-part2_slides.pdf

Backpropagation through time (BPTT)



Werbos, Paul J. "Backpropagation through time: what it does and how to do it." *Proceedings of the IEEE* 78, no. 10 (1990): 1550-1560.

$$L = \sum_{t=1}^{T} L^{(t)} \qquad \frac{\partial L^{(t)}}{\partial \mathbf{W}_{hh}} = \frac{\partial L^{(t)}}{\partial y^{(t)}} \cdot \frac{\partial y^{(t)}}{\partial \mathbf{h}^{(t)}} \cdot \left(\sum_{k=1}^{t} \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} \cdot \frac{\partial \mathbf{h}^{(k)}}{\partial \mathbf{W}_{hh}} \right)$$

computed as a multiplication of adjacent time steps:

$$\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} = \prod_{i=k+1}^{t} \frac{\partial \mathbf{h}^{(i)}}{\partial \mathbf{h}^{(i-1)}}$$

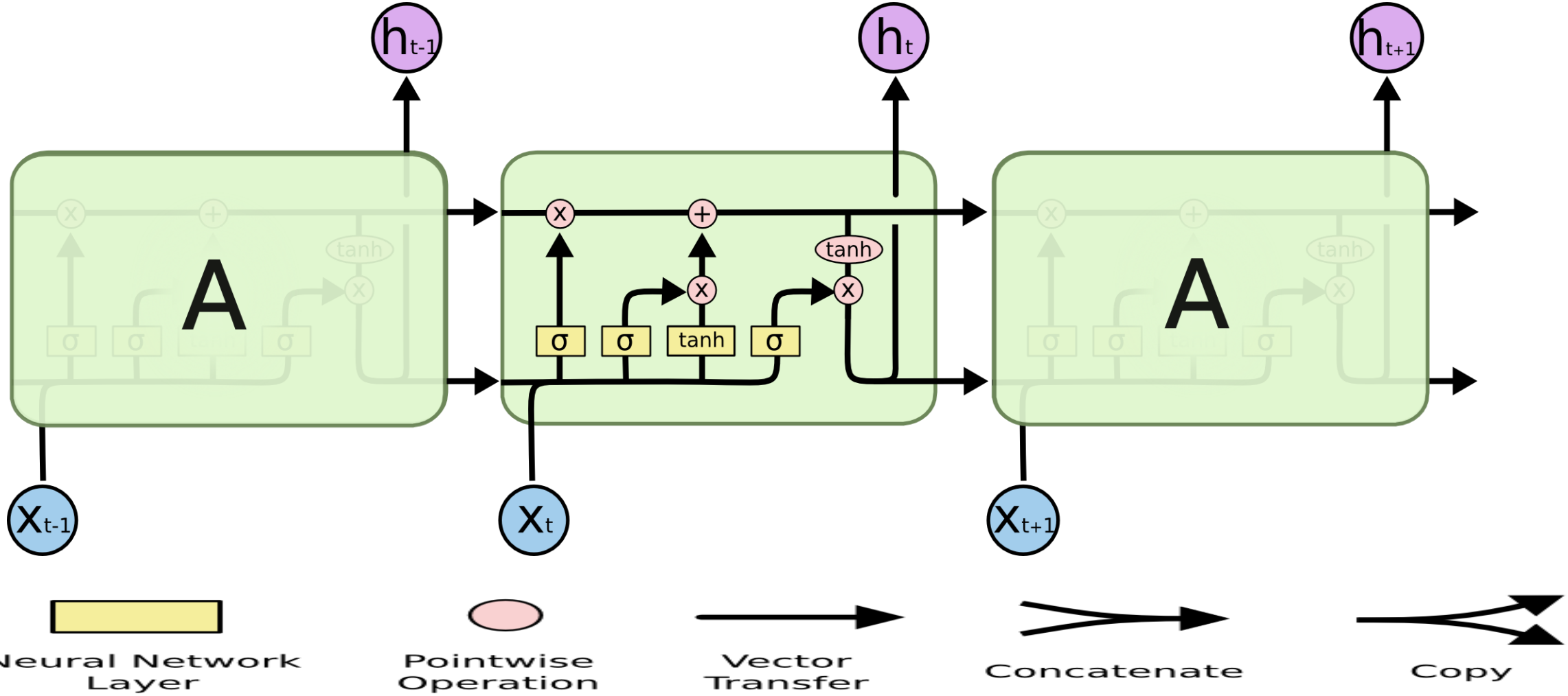
https://sebastianraschka.com/pdf/lecture-notes/stat479ss19/L14_intro-rnn-part2_slides.pdf

Solutions to the vanishing/exploding gradient problems

- Gradient Clipping
- Truncated backpropagation through time (TBPTT)
- Long short-term memory (LSTM)

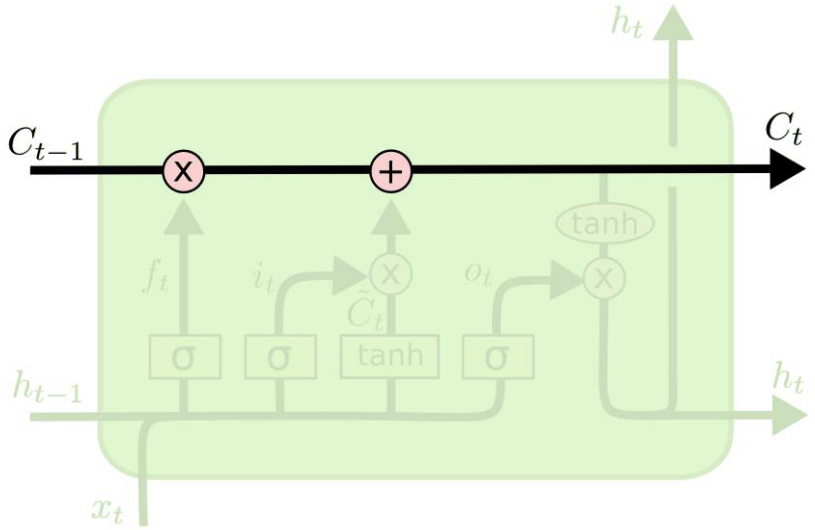
Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." *Neural computation* 9, no. 8 (1997): 1735-1780.

Long short-term memory (LSTM)



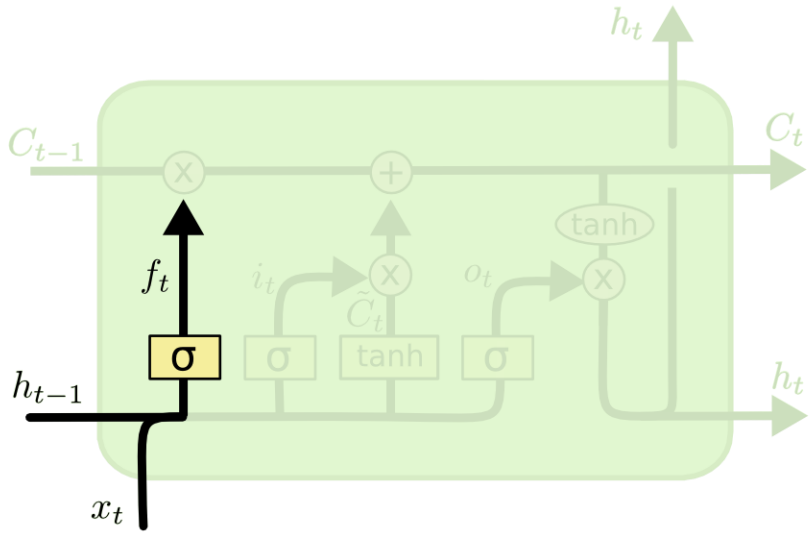
https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long short-term memory (LSTM)



https://colah.github.io/posts/2015-08-Understanding-LSTMs/

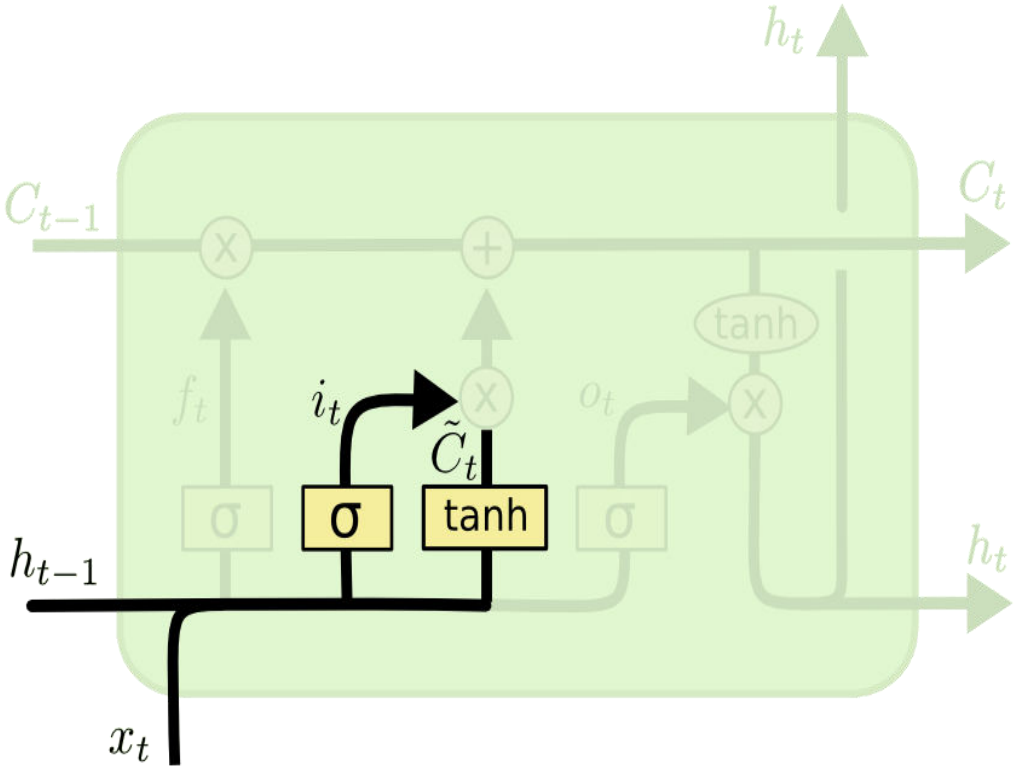
Long short-term memory (LSTM)



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long short-term memory (LSTM)

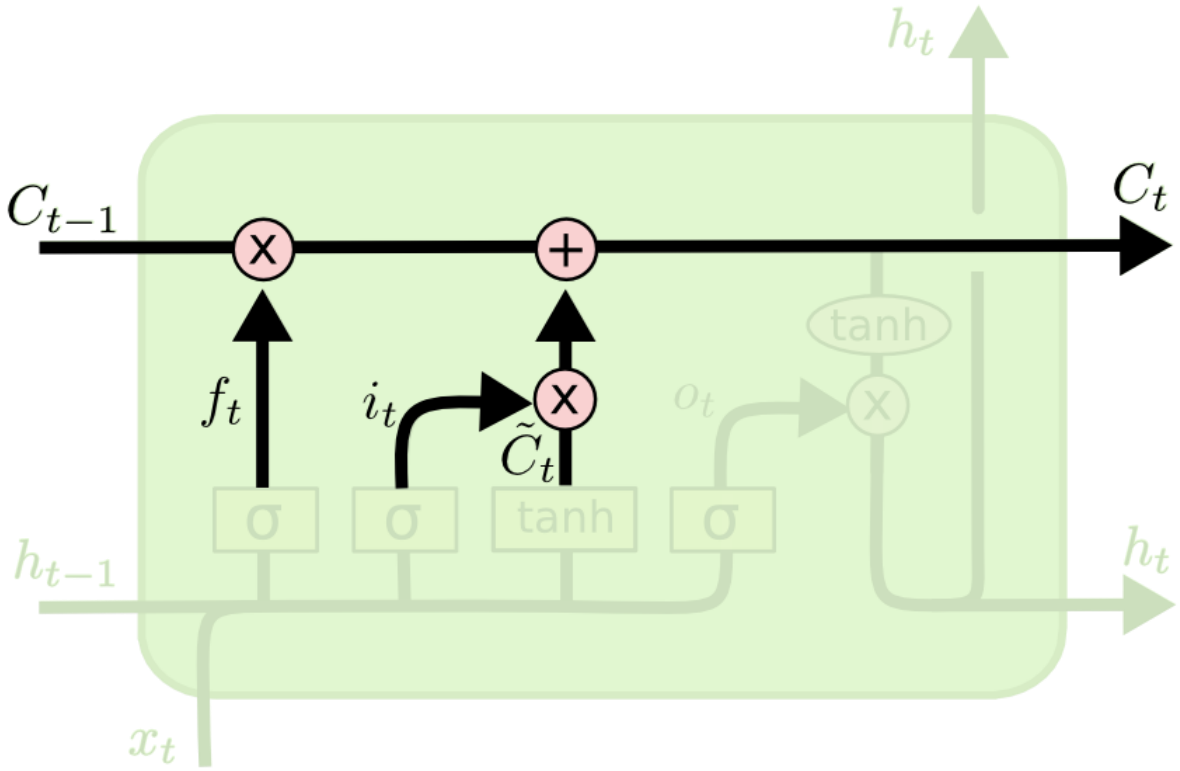


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

https://colah.github.io/posts/2015-08-Understanding-LSTMs/

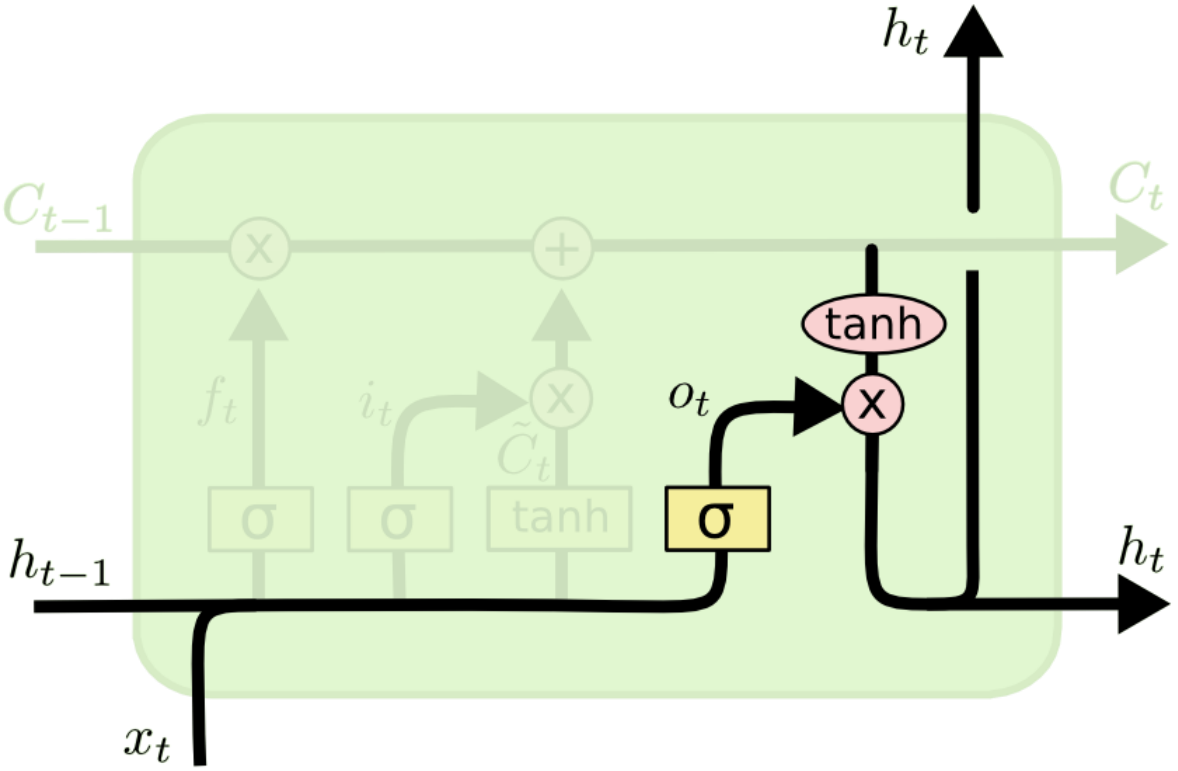
Long short-term memory (LSTM)



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long short-term memory (LSTM)

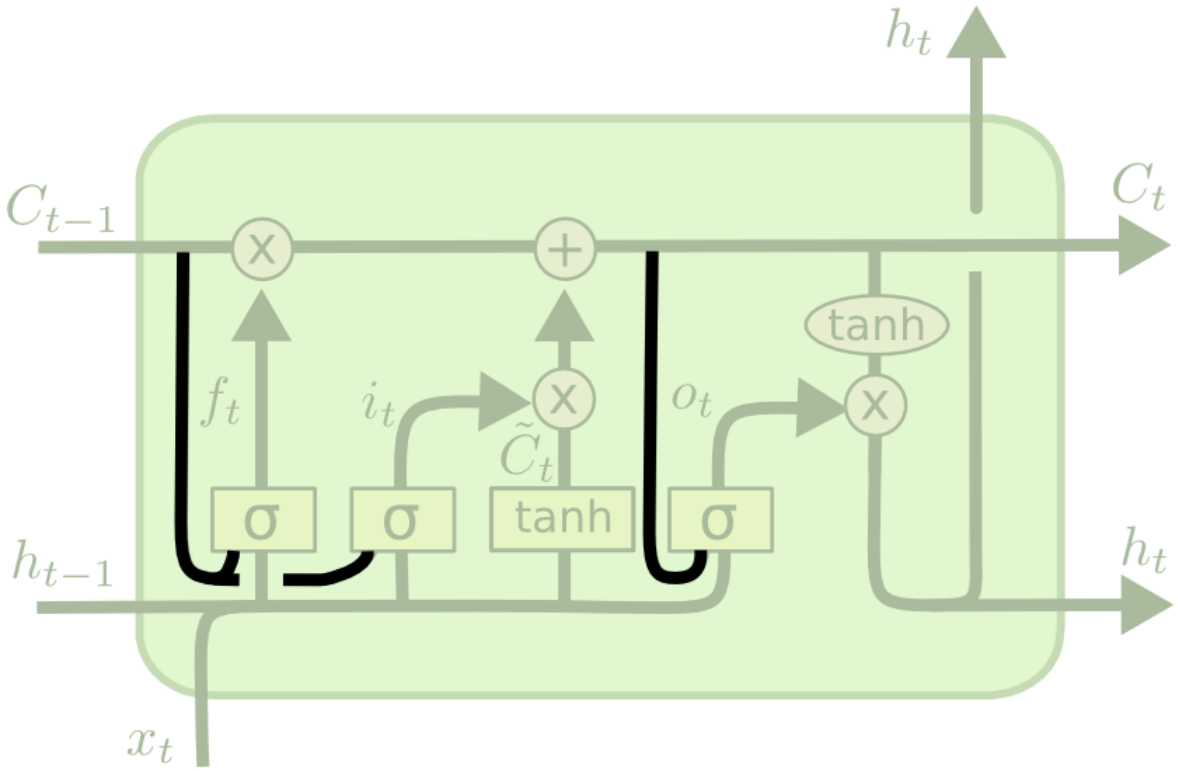


$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long short-term memory (LSTM)



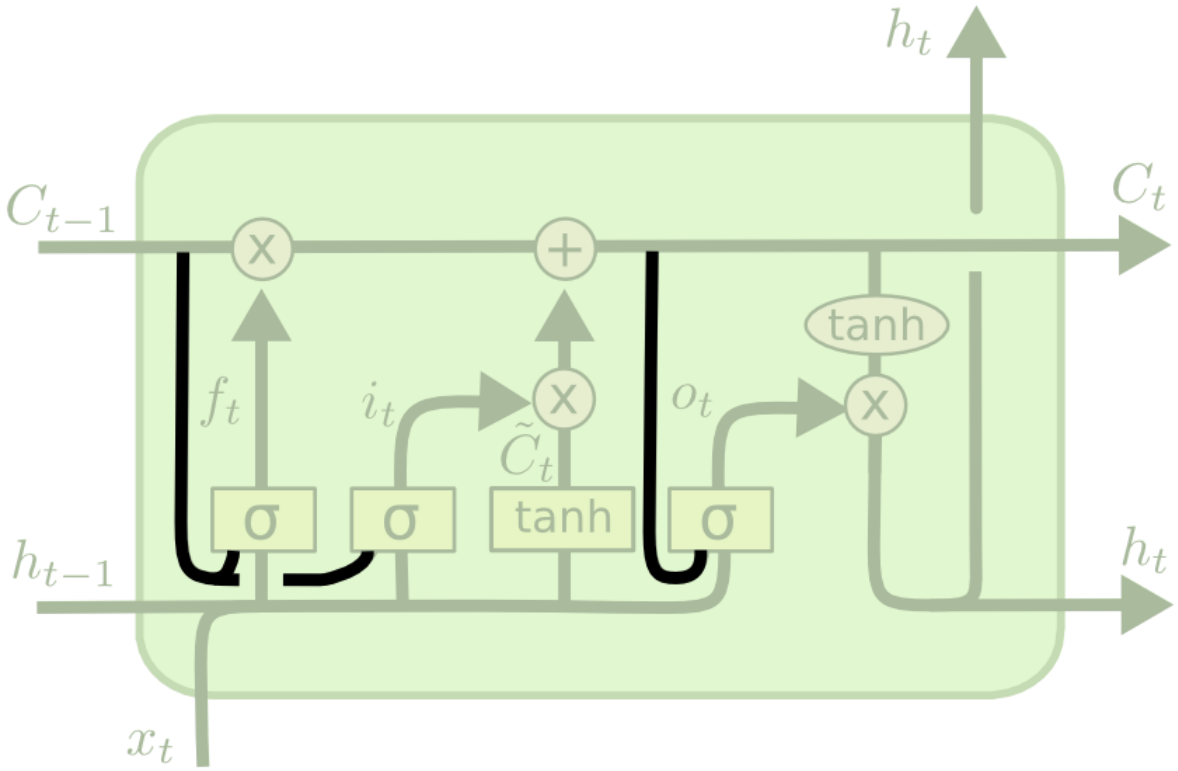
$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_f\right)$$

$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_i\right)$$

$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \; + \; b_o\right)$$

https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long short-term memory (LSTM)



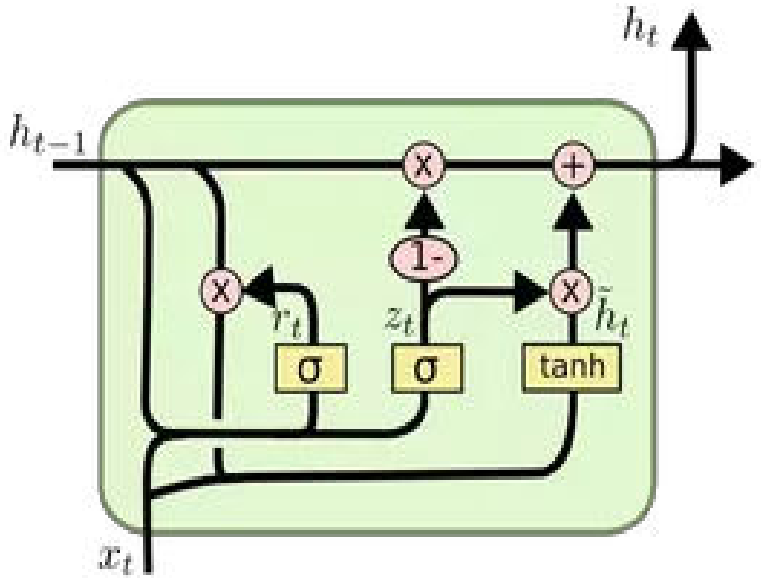
$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_f\right)$$

$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_i\right)$$

$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \; + \; b_o\right)$$

https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Gated Recurrent Unit(GRU)



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation

Kyunghyun Cho
Bart van Merriënboer **Caglar Gulcehre**
Université de Montréal
firstname.lastname@umontreal.ca

Dzmitry Bahdanau
Jacobs University, Germany
d.bahdanau@jacobs-university.de

Fethi Bougares **Holger Schwenk**
Université du Maine, France
firstname.lastname@lium.univ-lemans.fr

Yoshua Bengio
Université de Montréal, CIFAR Senior Fellow
find.me@on.the.web

Autoencoder

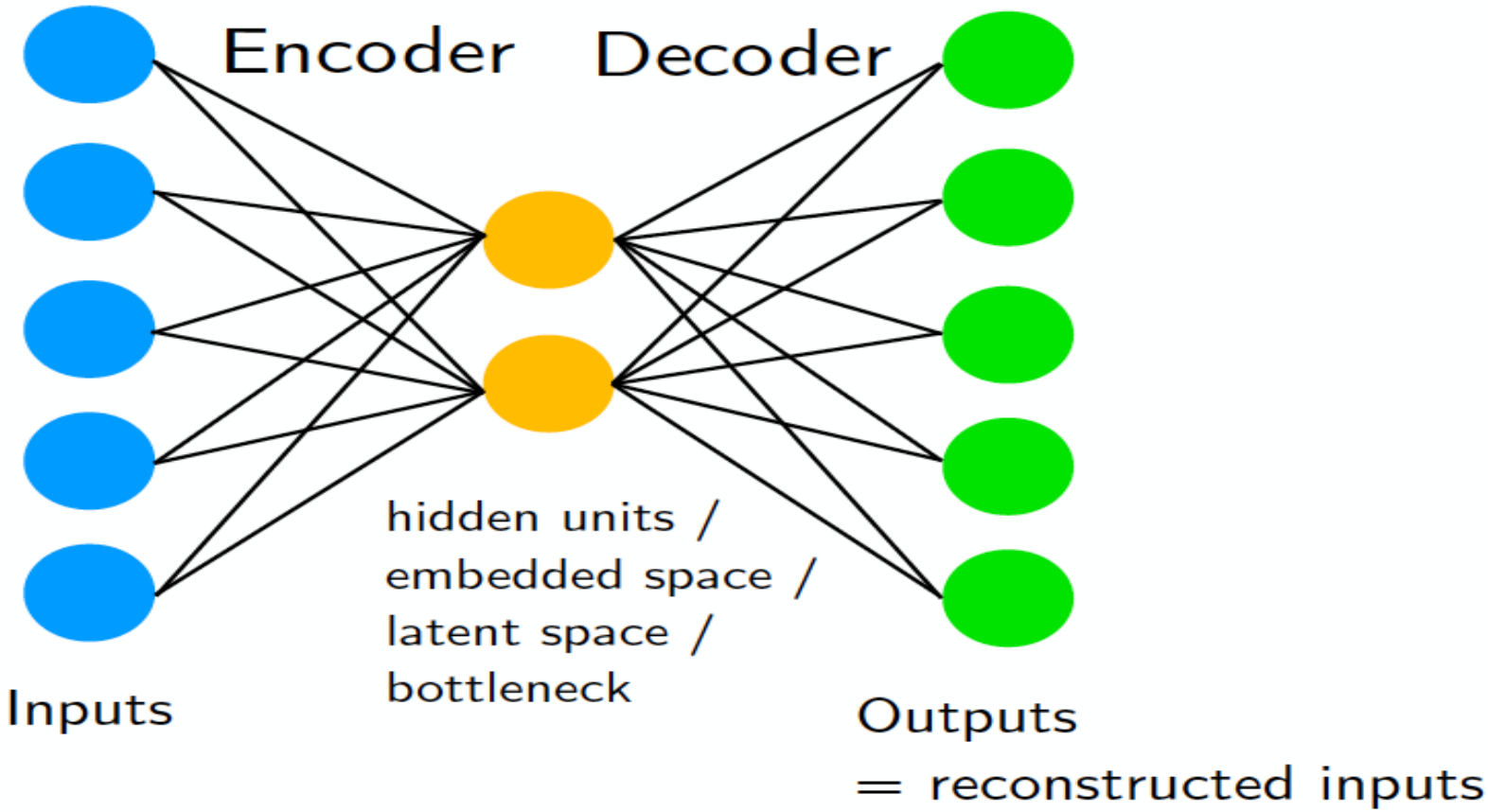
Unsupervised Learning

- Process of learning from a data without its corresponding label.

Application:

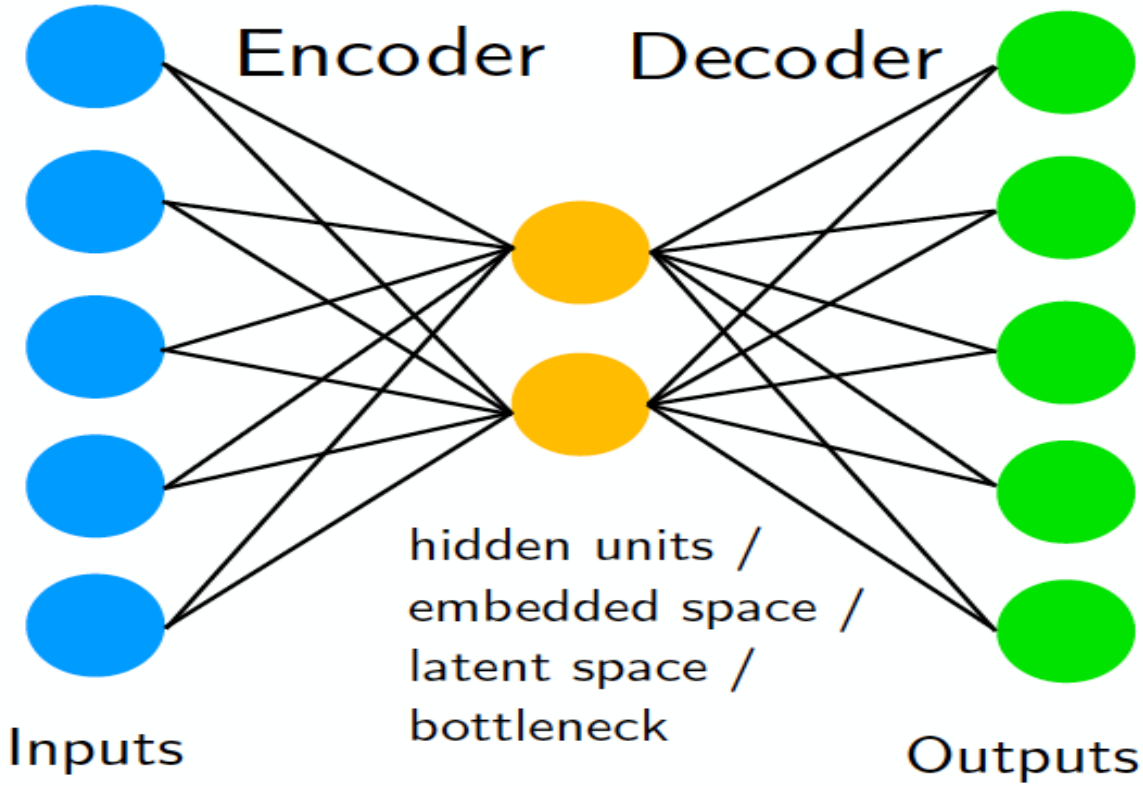
- Finding hidden structures in data
- Data compression
- Clustering
- Retrieving similar objects
- Exploratory Data Analysis
- Generating new examples

Fully-Connected (Multilayer-Perceptron) Autoencoder

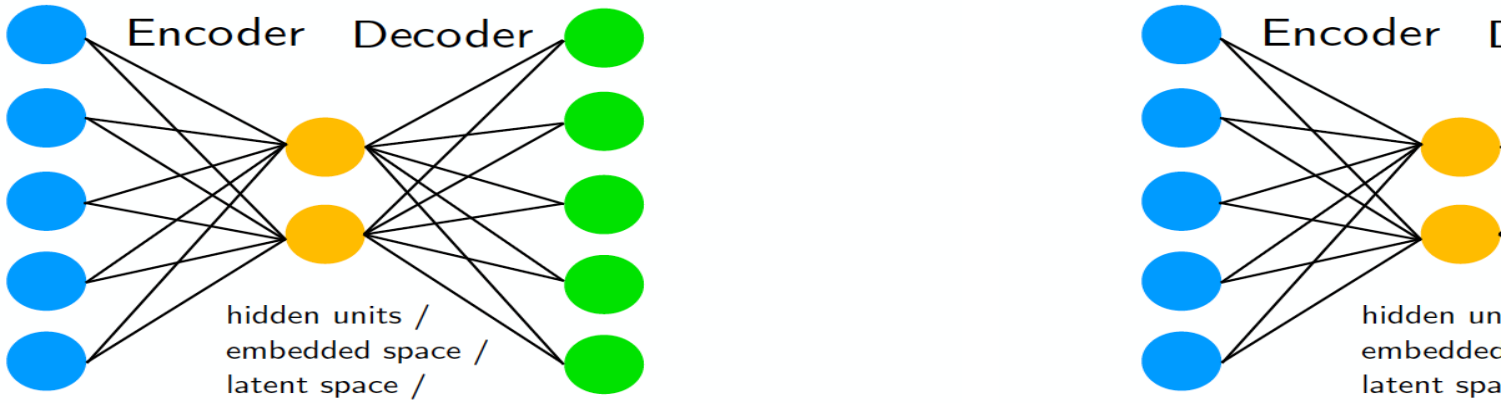


Fully-Connected (Multilayer-Perceptron) Autoencoder

$$\mathcal{L}(\mathbf{x}, \mathbf{x}') = ||\mathbf{x} - \mathbf{x}'||_2^2 = \sum (x_i - x_i')^2$$



Autoencoder Applications

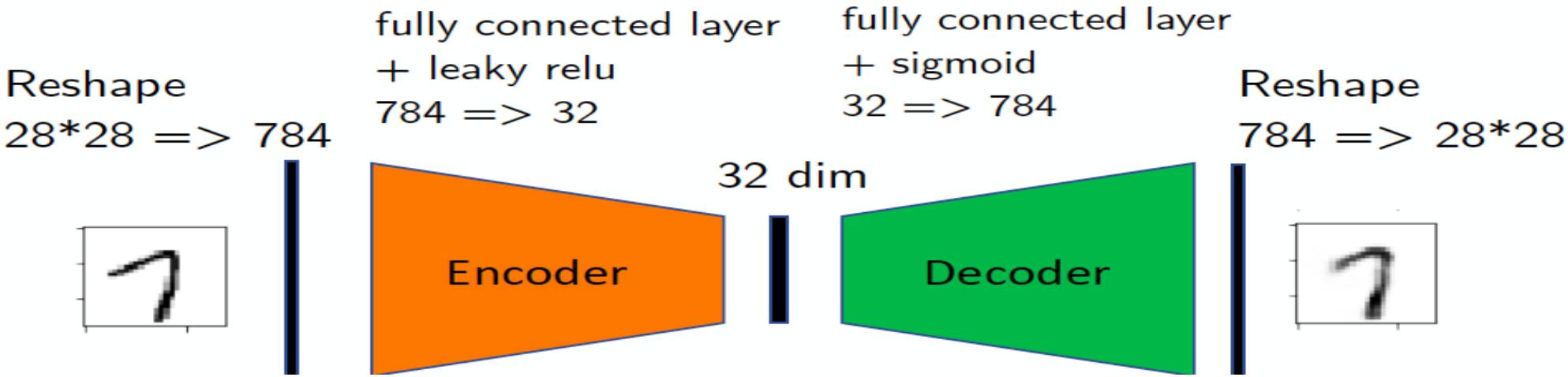


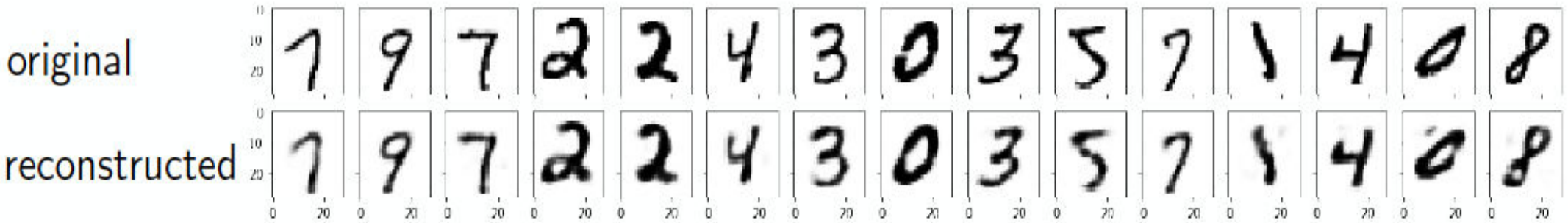
1. Use embedding as input to classic machine learning methods (SVM, KNN, Random Forest, ...)

2. Or, similar to transfer learning, train autoencoder on large image dataset, then fine tune encoder part on your own, smaller datset and/or provide your own output (classification) layer

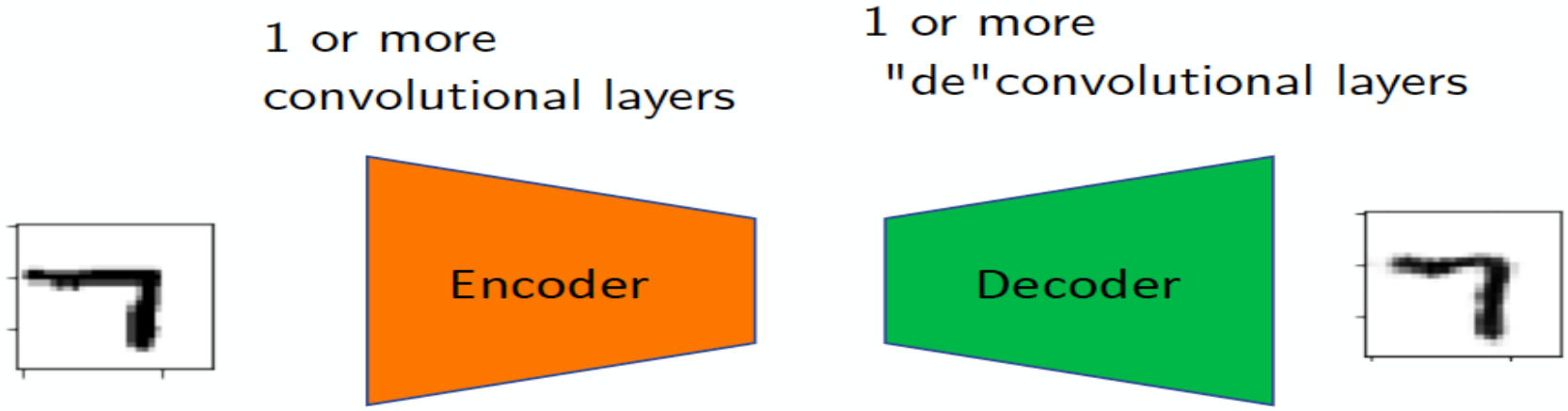
3. Latent space can also be used for visualization (EDA, clustering), but there are better methods for that

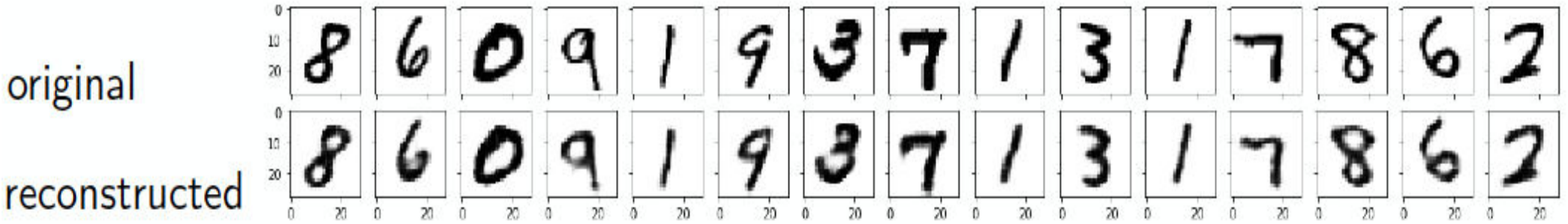
A Simple Autoencoder





A Convolutional Autoencoder





Up-Sampling Method

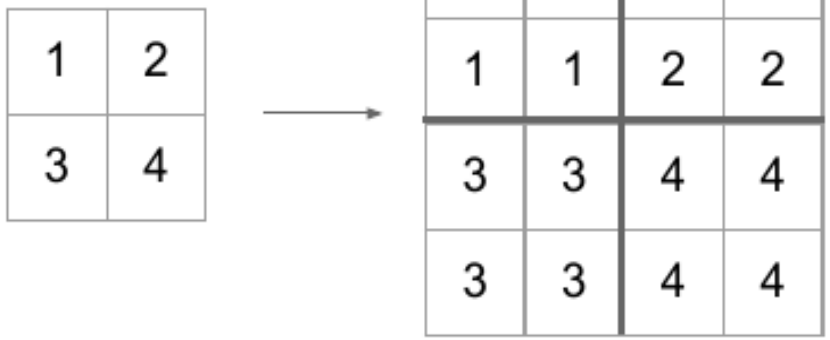
- Allow to increase the size of feature map from input

Up-sampling Method:

- Nearest Neighbour
- Bi-Linear Interpolation
- Bed Of Nails
- Max-Unpooling
- Transposed Convolution

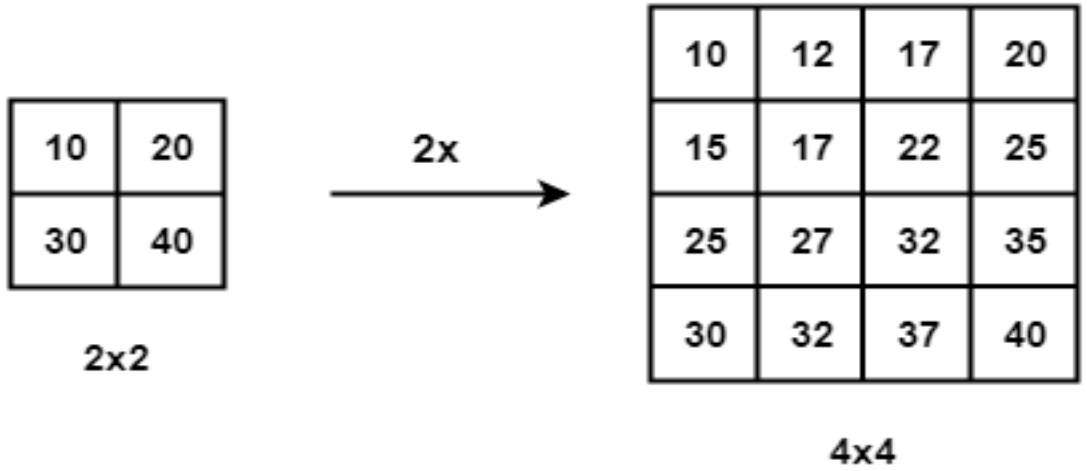
Up-Sampling Method

Nearest Neighbor



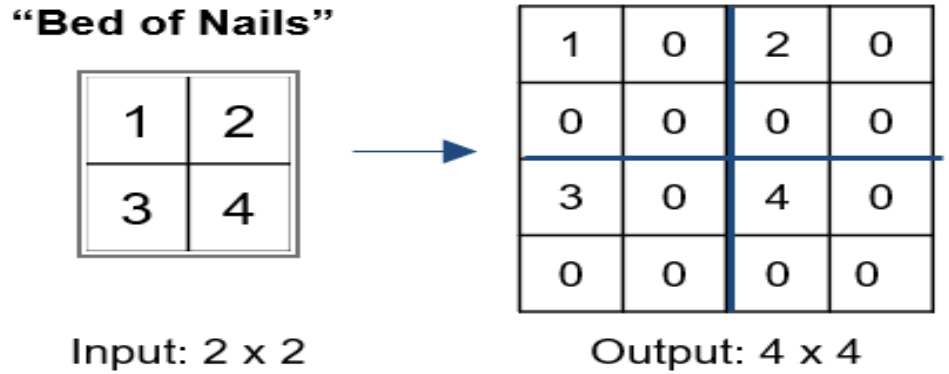
Input: 2 x 2 Output: 4 x 4

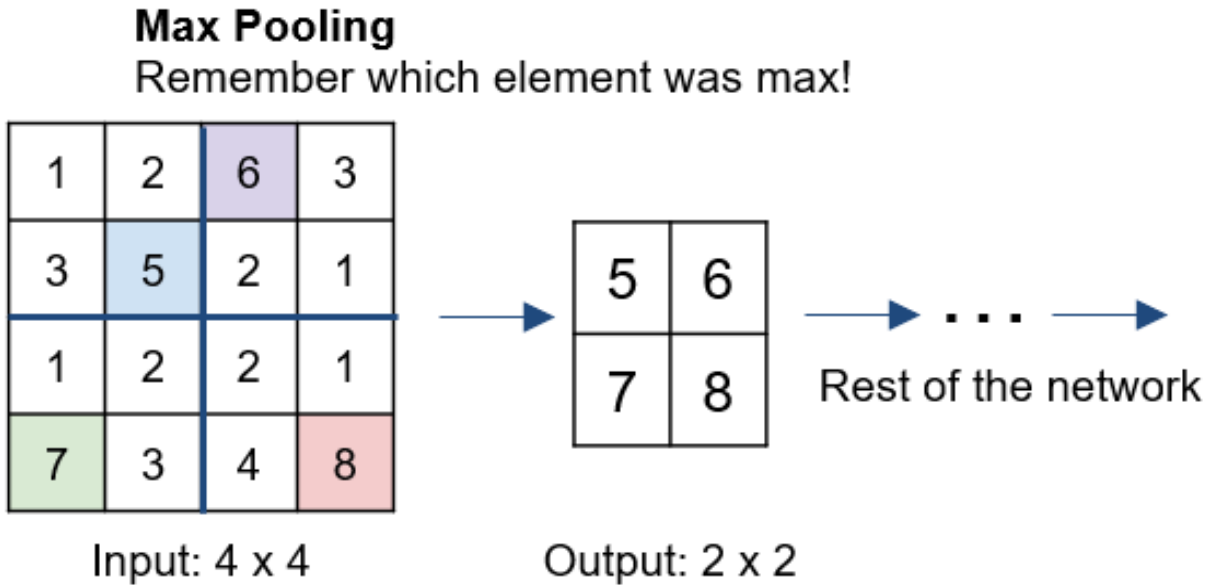
Bi Linear interpolation

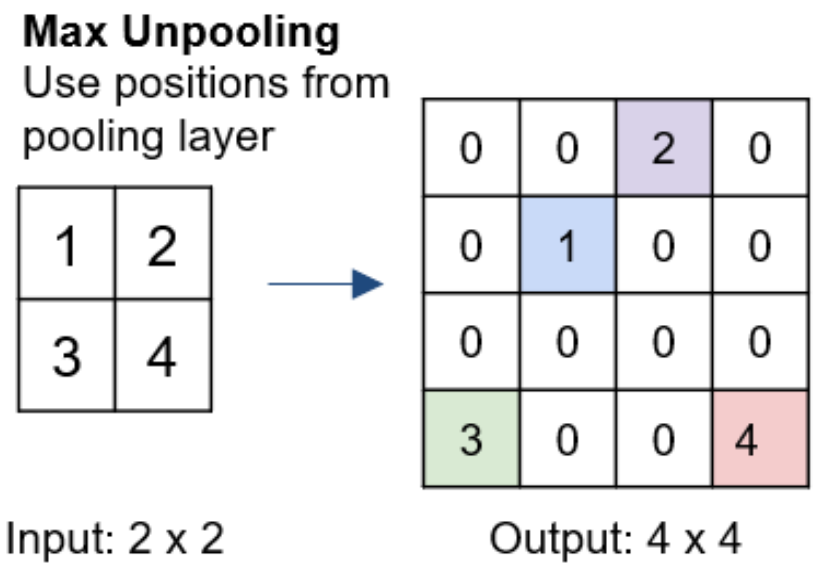


2x2 4x4

Up-Sampling Method







Transposed Convolution

Regular Convolution:

output

input

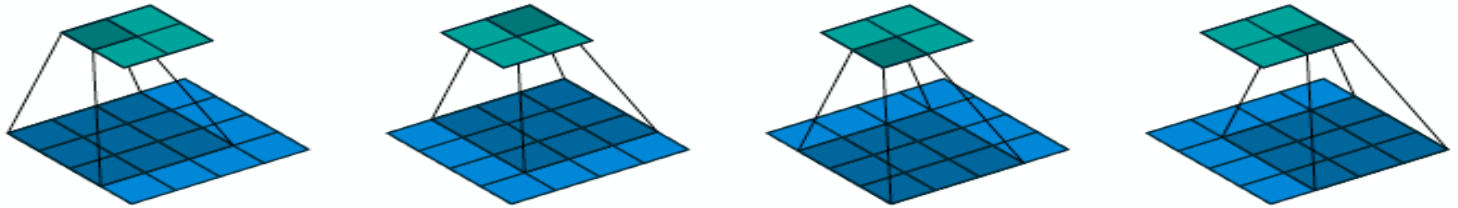
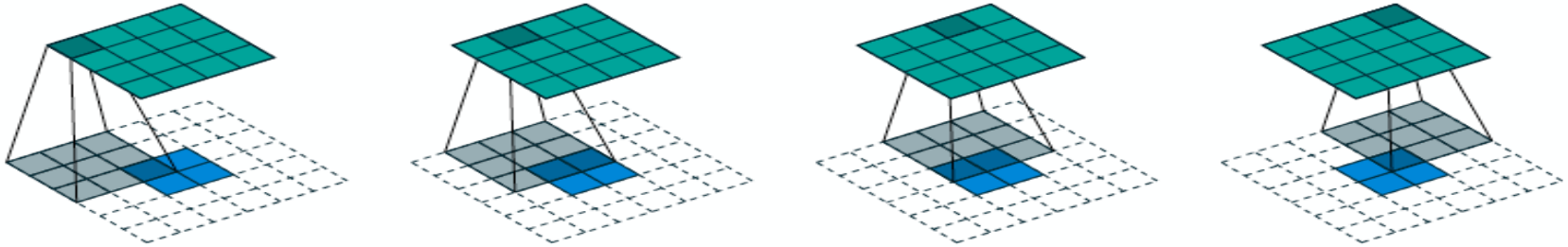


Figure 2.1: (No padding, unit strides) Convolving a 3×3 kernel over a 4×4 input using unit strides (i.e., $i = 4$, $k = 3$, $s = 1$ and $p = 0$).

Transposed Convolution (emulated with direct convolution):

output

input



Dumoulin, Vincent, and Francesco Visin. "A guide to convolution arithmetic for deep learning." *arXiv preprint arXiv:1603.07285* (2016).

Transposed Convolution

```
import torch
```

```
torch.manual_seed(123)
a = torch.rand(4).view(1, 1, 2, 2)

conv_t = torch.nn.ConvTranspose2d(in_channels=1,
                                  out_channels=1,
                                  kernel_size=(3, 3),
                                  padding=0,
                                  stride=1)

# output = s(n-1)+k-2p = 1*(2-1)+3-2*0 = 4
conv_t(a)
```

```
tensor([[[[-0.2863, -0.2766, -0.1478, -0.3274],
          [-0.3522, -0.5356, -0.1591, -0.2911],
          [-0.3054, -0.4644, -0.3286, -0.2444],
          [-0.2332, -0.2557, -0.1876, -0.3970]]]],
       grad_fn=<ThnnConvTranspose2DBackward>)
```

$$\text{output} = s(n-1) + k - 2p$$

```
torch.manual_seed(123)
a = torch.rand(16).view(1, 1, 4, 4)

conv_t = torch.nn.ConvTranspose2d(in_channels=1,
                                  out_channels=1,
                                  kernel_size=(3, 3),
                                  padding=0,
                                  stride=1)

# output = s(n-1)+k-2p = 1*(4-1)+3-2*0 = 6
conv_t(a).size()
```

```
torch.Size([1, 1, 6, 6])
```

```
torch.manual_seed(123)
a = torch.rand(64).view(1, 1, 8, 8)

conv_t = torch.nn.ConvTranspose2d(in_channels=1,
                                  out_channels=1,
                                  kernel_size=(3, 3),
                                  padding=0,
                                  stride=1)

# output = s(n-1)+k-2p = 1*(8-1)+3-2*0 = 10
conv_t(a).size()
```

```
torch.Size([1, 1, 10, 10])
```

Transposed Convolution

- strides: in transposed convolutions, we stride over the output; hence, larger strides will result in larger outputs (opposite to regular convolutions)

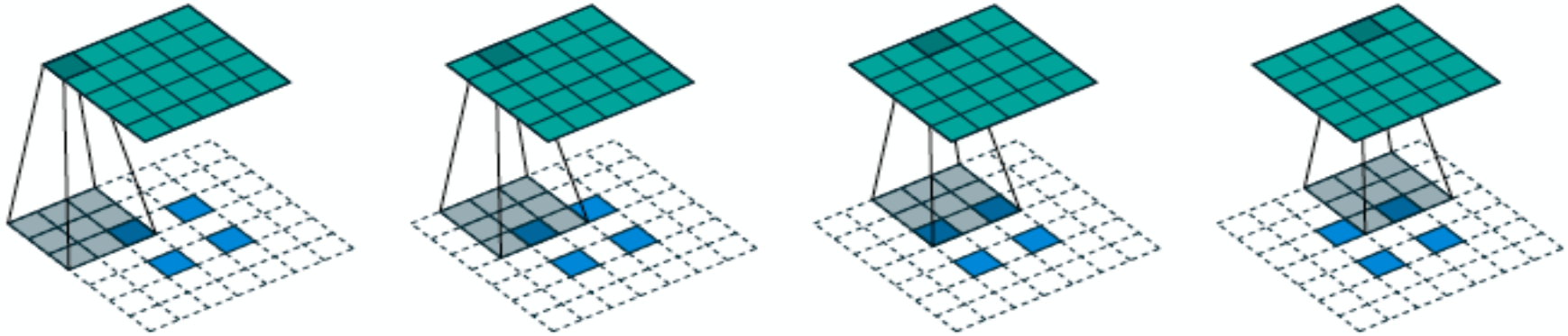


Figure 4.5: The transpose of convolving a 3×3 kernel over a 5×5 input using 2×2 strides (i.e., $i = 5$, $k = 3$, $s = 2$ and $p = 0$). It is equivalent to convolving a 3×3 kernel over a 2×2 input (with 1 zero inserted between inputs) padded with a 2×2 border of zeros using unit strides (i.e., $i' = 2$, $\tilde{i}' = 3$, $k' = k$, $s' = 1$ and $p' = 2$).

Problems with Transposed Convolutions

- Transposed convolutions suffer from checkerboard effects as shown below

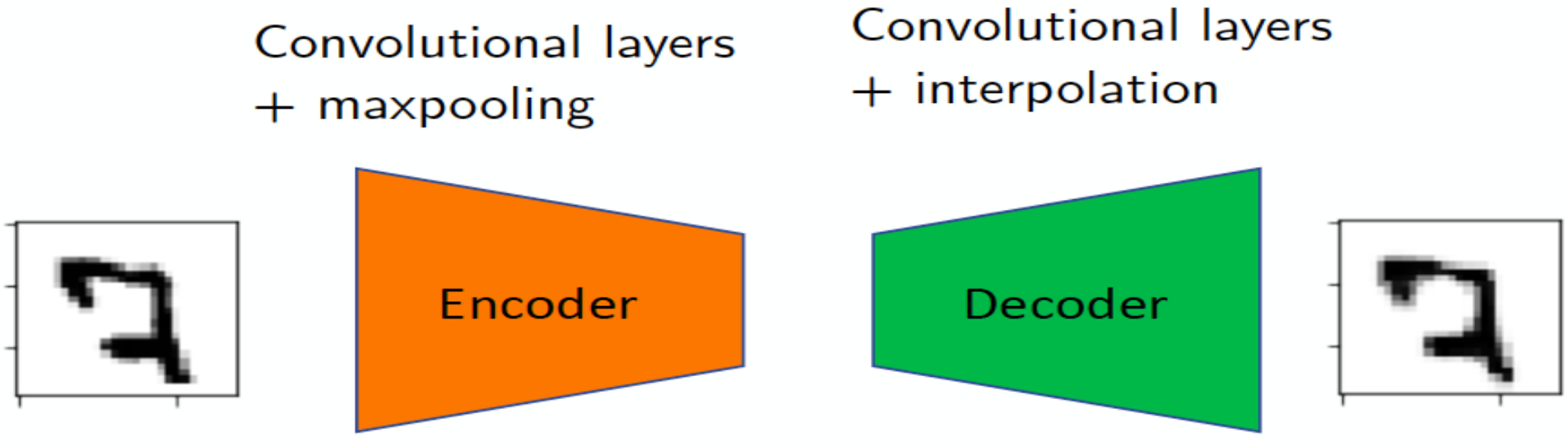
https://distill.pub/2016/deconv-checkerboard/

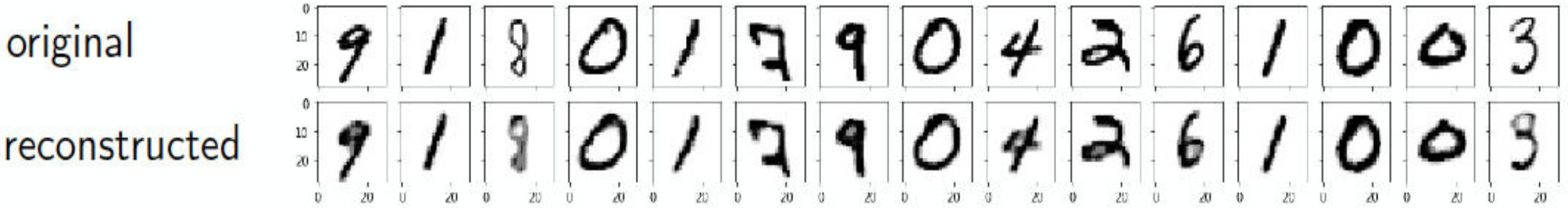
Problems with Transposed Convolutions

https://distill.pub/2016/deconv-checkerboard/

it is recommended to replace transposed conv. by upsampling (interpolation) followed by regular convolution

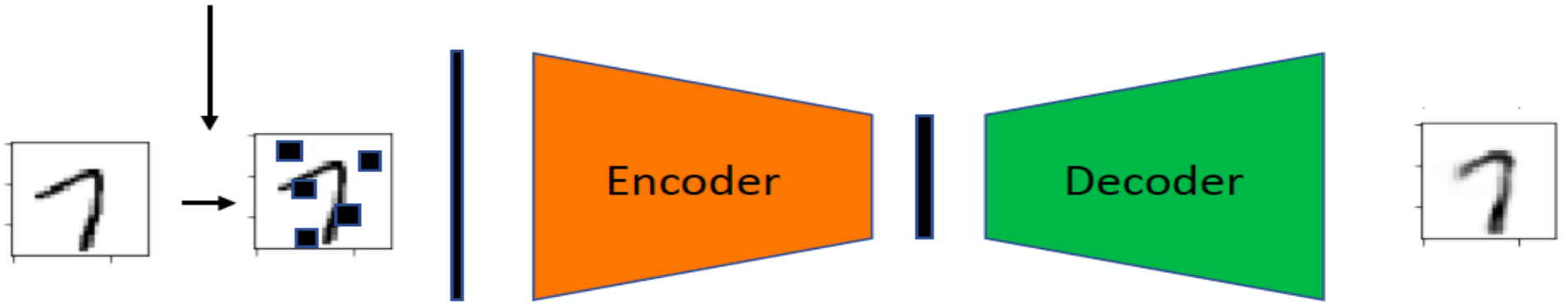
A Convolutional Autoencoder with Nearest Neighbor Interpolation





Denoising Autoencoder

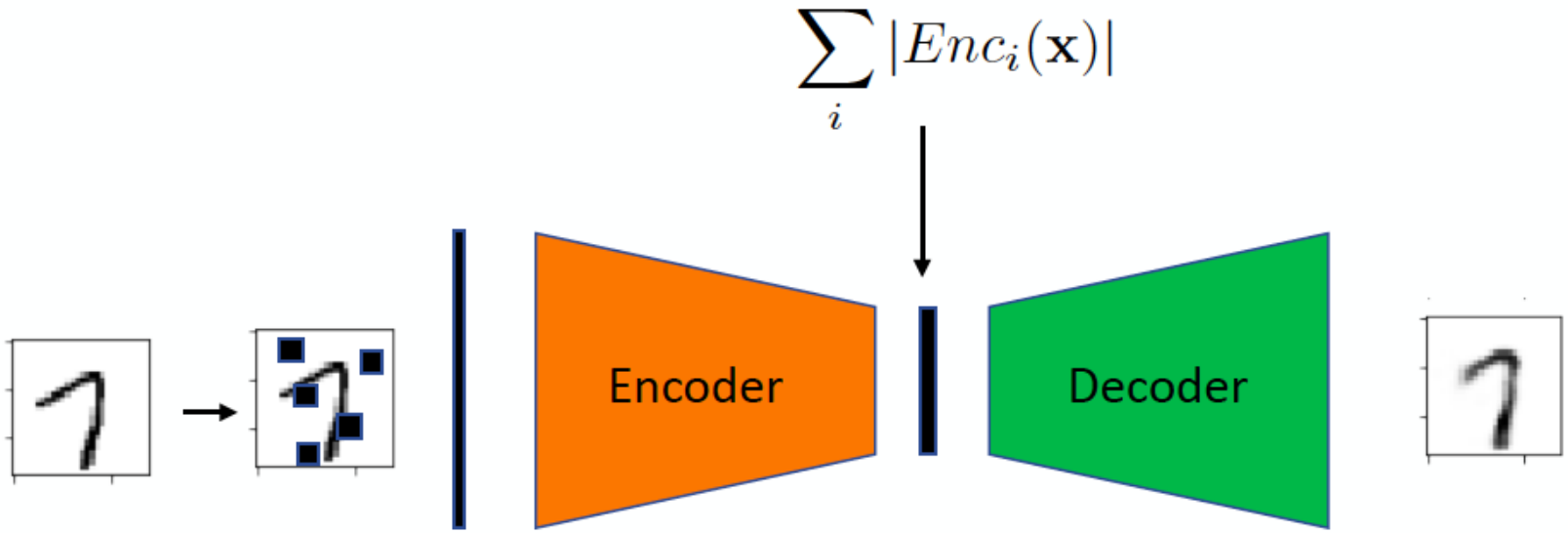
Add dropout after the input, or add noise to the input to learn to denoise images



Vincent, P., Larochelle, H., Bengio, Y., & Manzagol, P. A. (2008, July). Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th International Conference on Machine Learning* (pp. 1096-1103). ACM.

Sparse Autoencoder

Add L1 penalty to the loss to learn sparse feature representations



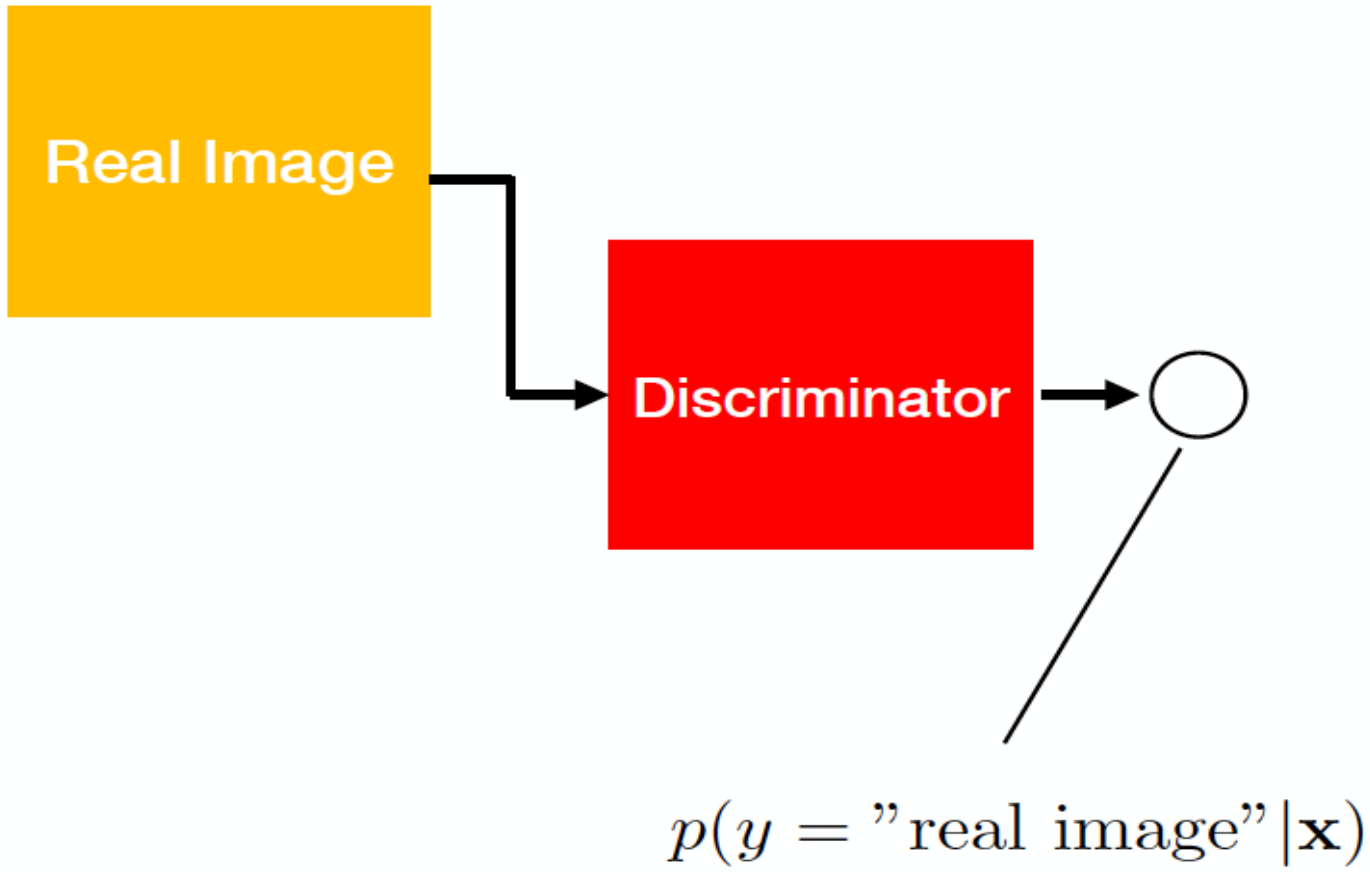
$$\mathcal{L} = ||\mathbf{x} - Dec(Enc(\mathbf{x}))||_2^2 + \sum_i |Enc_i(\mathbf{x})|$$

Generative Adversarial Nets(GANs)

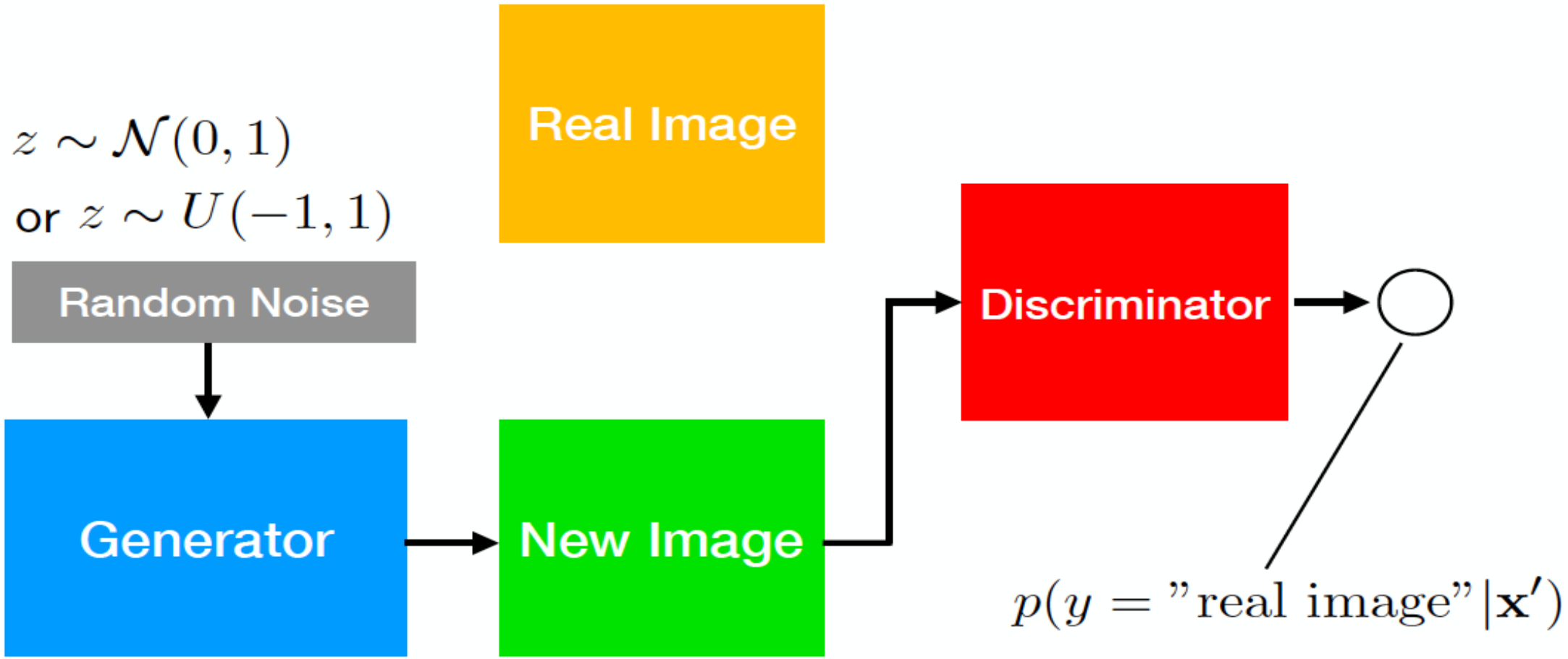
Generative Adversarial Nets(GANs)

- The original purpose is to generate new data Classically for generating new images, but applicable to wide range of domains
- Learns the training set distribution and can generate new images that have never been seen before
- In contrast to e.g., autoregressive models or RNNs (generating one word at a time), GANs generate the whole output all at once

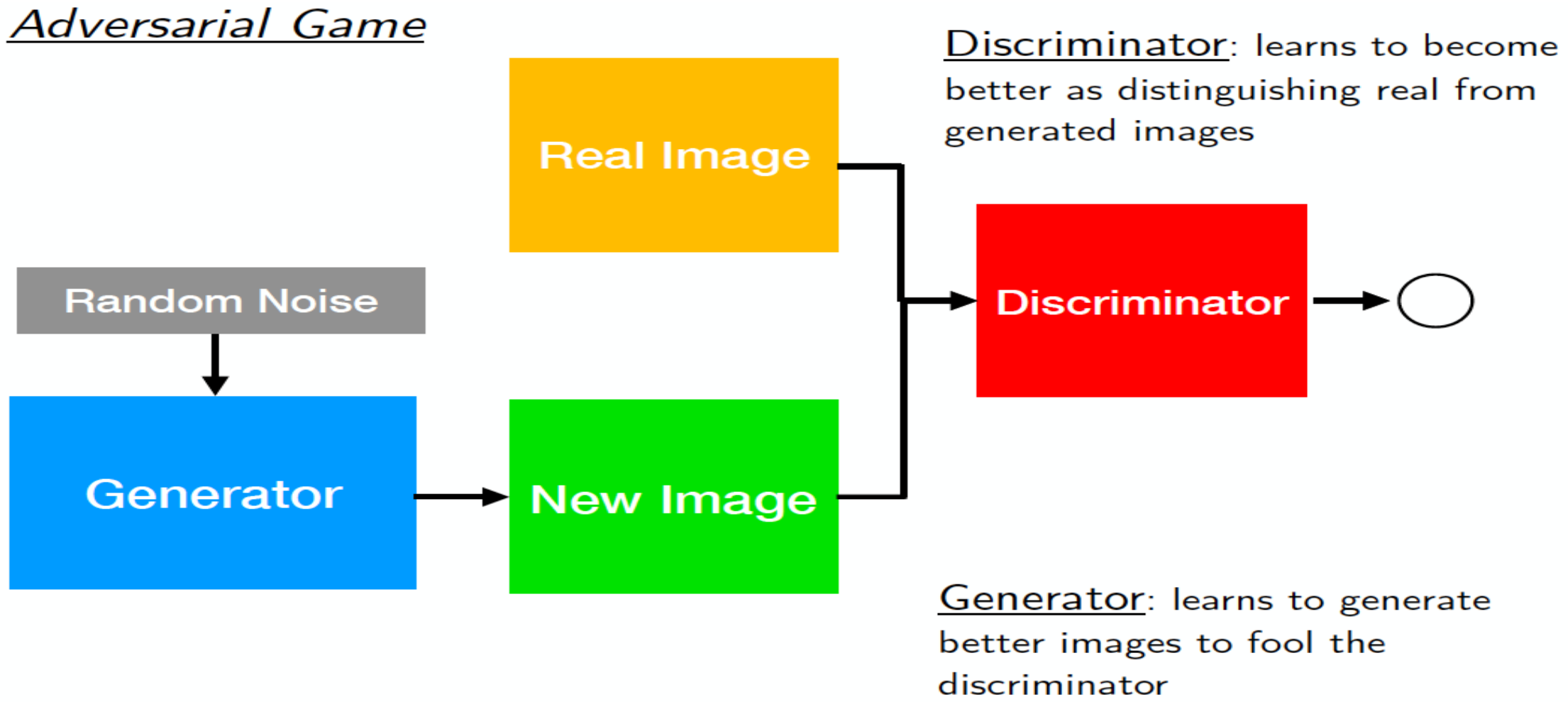
GANs



GANs


$z \sim \mathcal{N}(0, 1)$
or $z \sim U(-1, 1)$
Random Noise
Real Image
Generator
New Image
Discriminator
$p(y = \text{"real image"} | \mathbf{x}')$

GANs


Adversarial Game
Real Image
Random Noise
Discriminator
Generator
New Image
Discriminator: learns to become better as distinguishing real from generated images
Generator: learns to generate better images to fool the discriminator

GANs Objective

$$\min_G \max_D L(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\tan}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{x}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$

GANs Objective

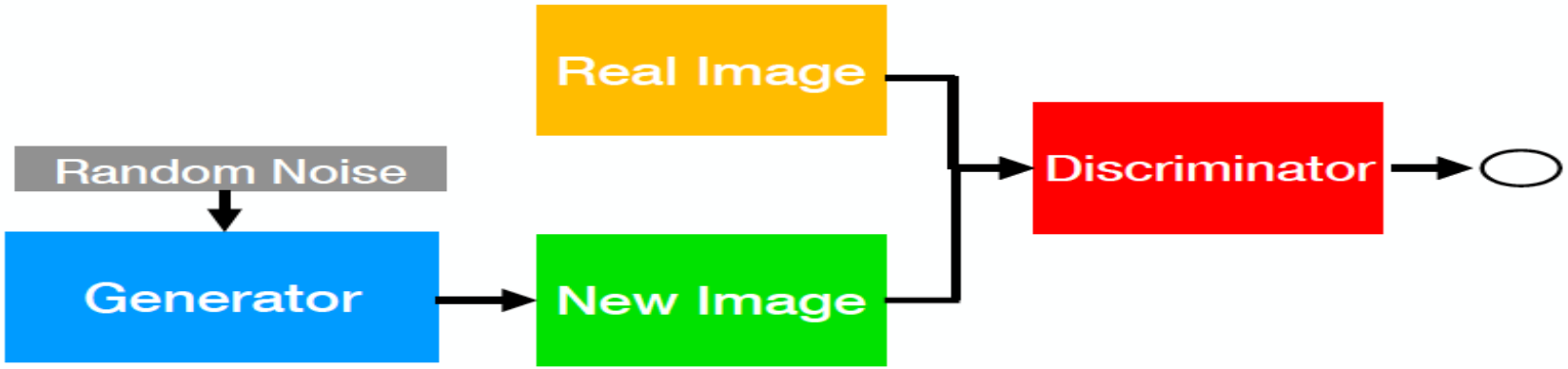
$$\min_G \max_D L(D, G) = \mathbb{E}_{\boldsymbol{x} \sim p_{\tan}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{x}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$

Discriminator gradient for update (gradient ascent):

predict well on real images => probability close to 1

predict well on fake images => probability close to 0

$$\nabla_{\mathbf{w}_D} \frac{1}{n} \sum_{i=1}^{n} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right]$$



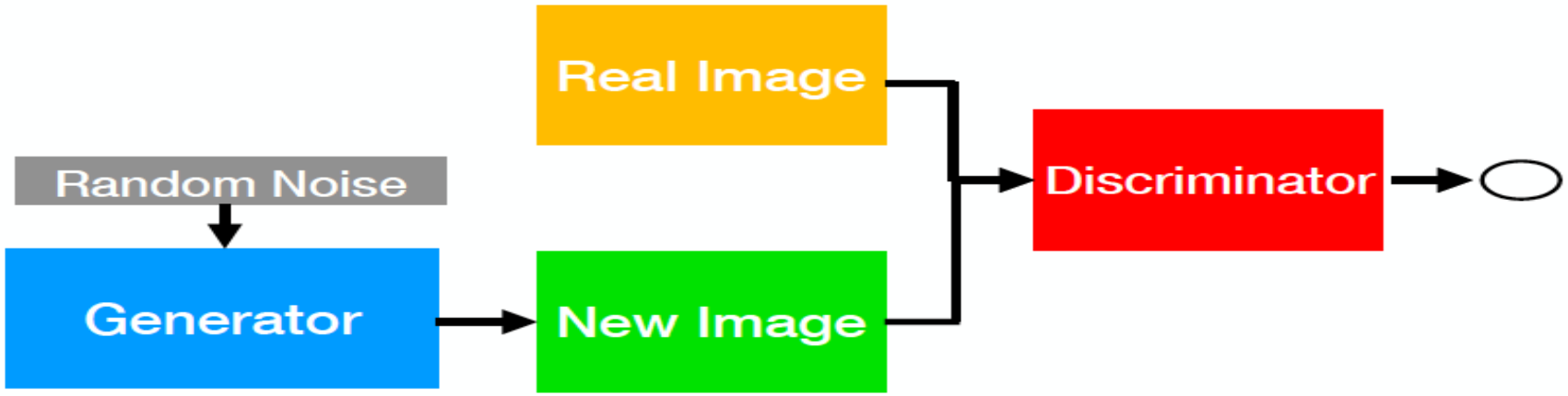
GANs Objective

$$\min_G \max_D L(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\tan}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{x}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))$$

Generator gradient for update (gradient descent):

predict badly on fake images
=> probability close to 1

$$\nabla \mathbf{w}_G \frac{1}{n} \sum_{i=1}^{n} \log \left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)$$



Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

  for k steps **do**

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
  - Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

- Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative Adversarial Nets." In *Advances in Neural Information Processing Systems*, pp. 2672-2680. 2014.

GAN Convergence

- Converges when Nash-equilibrium (Game Theory concept) is reached in the minmax (zero-sum) game

$$\min_G \max_D L(D, G) = \mathbb{E}_{\boldsymbol{x} \sim p_{\tan}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{x}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$

- Nash-Equilibrium in Game Theory is reached when the actions of one player won't change depending on the opponent's actions
- Here, this means that the GAN produces realistic images and the discriminator outputs random predictions (probabilities close to 0.5)

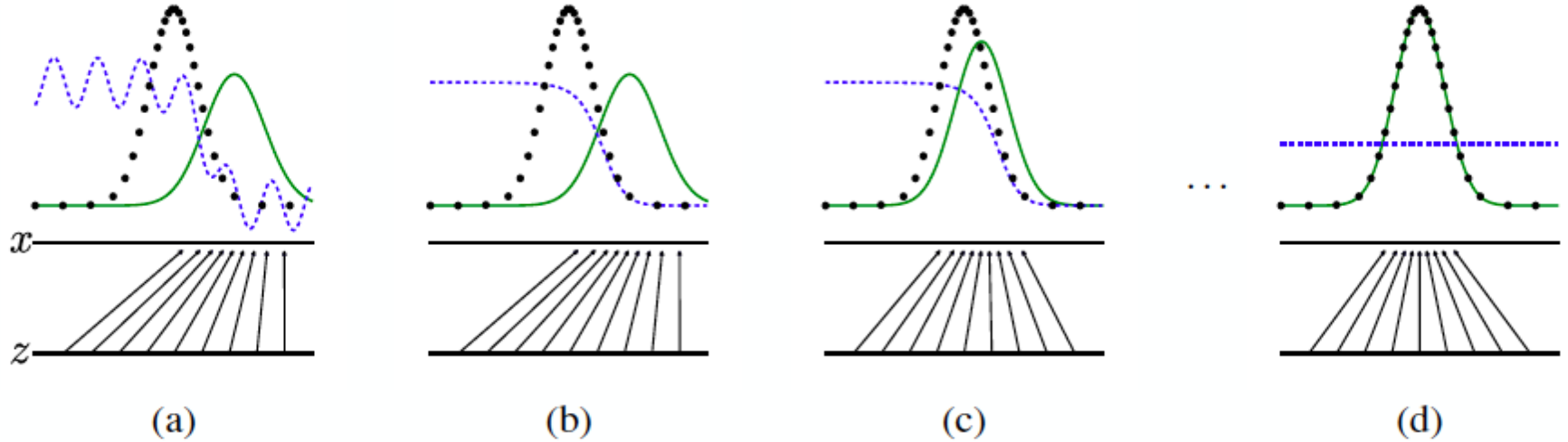
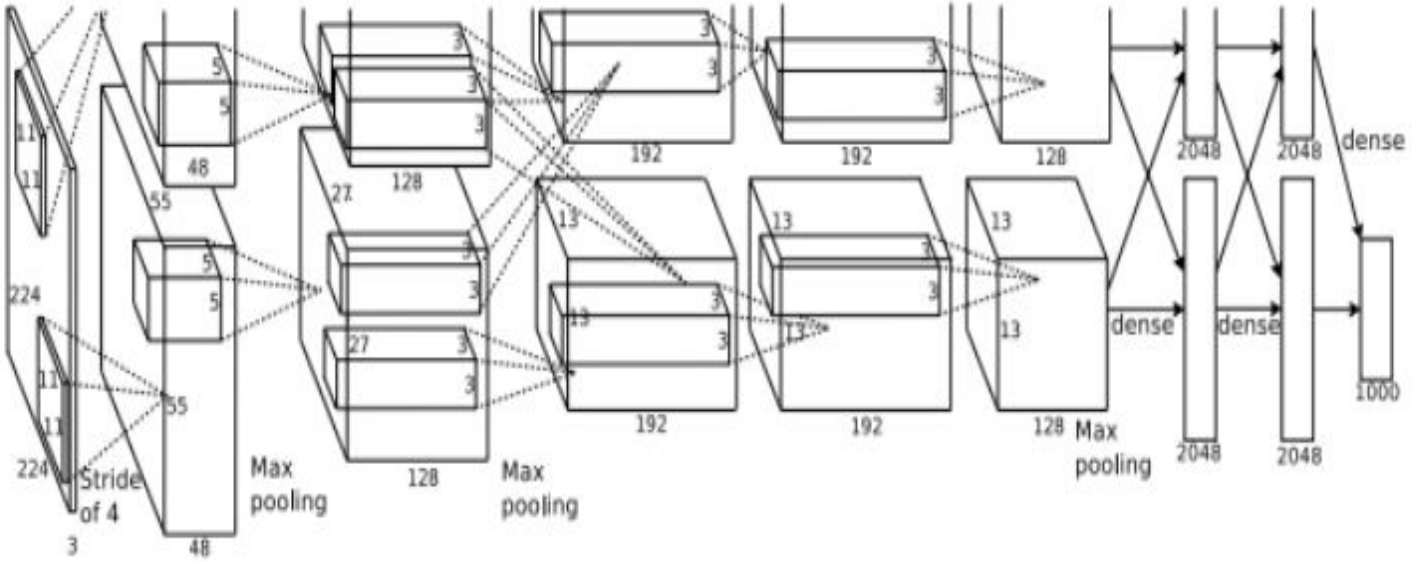


Figure 1: Generative adversarial nets are trained by simultaneously updating the discriminative distribution (D, blue, dashed line) so that it discriminates between samples from the data generating distribution (black, dotted line) p_x from those of the generative distribution p_g (G) (green, solid line). The lower horizontal line is the domain from which z is sampled, in this case uniformly. The horizontal line above is part of the domain of x. The upward arrows show how the mapping $x = G(z)$ imposes the non-uniform distribution p_g on transformed samples. G contracts in regions of high density and expands in regions of low density of p_g. (a) Consider an adversarial pair near convergence: p_g is similar to p_{data} and D is a partially accurate classifier. (b) In the inner loop of the algorithm D is trained to discriminate samples from data, converging to $D^*(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x)+p_g(x)}$. (c) After an update to G, gradient of D has guided $G(z)$ to flow to regions that are more likely to be classified as data. (d) After several steps of training, if G and D have enough capacity, they will reach a point at which both cannot improve because $p_g = p_{\text{data}}$. The discriminator is unable to differentiate between the two distributions, i.e. $D(x) = \frac{1}{2}$.

- Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative Adversarial Nets." In *Advances in Neural Information Processing Systems*, pp. 2672-2680. 2014.

Neural Style Transfer

AlexNet



Class scores for 1000 Classes

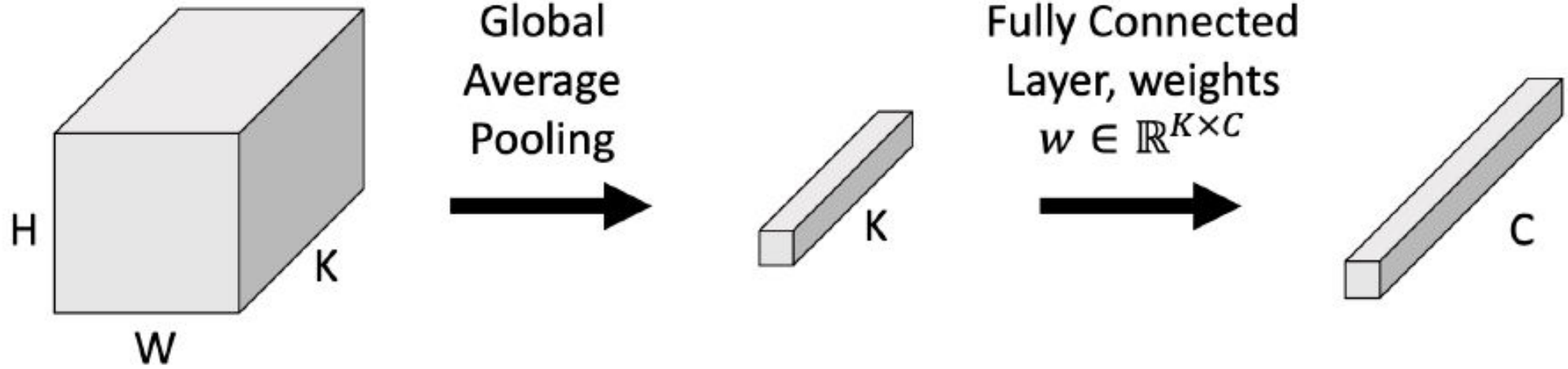
CNN Visualization

- Generative Methods
  - Gradient Ascent
  - Feature Inversion
  - Deep Dream

Class Activation Map (CAM)

Learning Deep Features for Discriminative Localization

Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, Antonio Torralba
Computer Science and Artificial Intelligence Laboratory, MIT
{bzhou,khosla,agata,oliva,torralba}@csail.mit.edu



Last layer CNN features:
$f \in \mathbb{R}^{H \times W \times K}$

Pooled features:
$F \in \mathbb{R}^{K}$

Class Scores:
$S \in \mathbb{R}^{C}$

$$F_k = \frac{1}{HW} \sum_{h,w} f_{h,w,k}$$

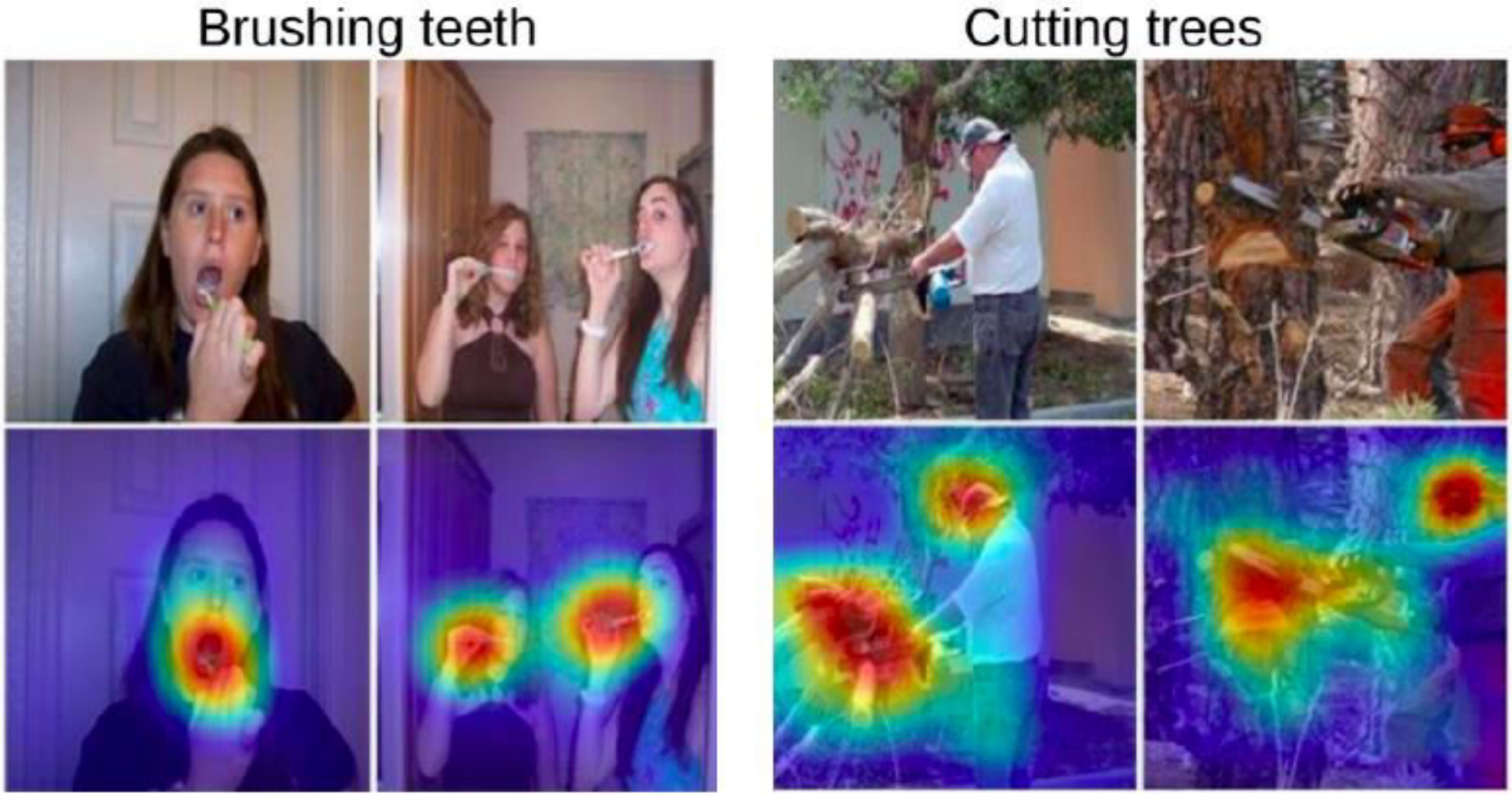
$$S_c = \sum_k w_{k,c} F_k$$

$$= \frac{1}{HW} \sum_{h,w} \sum_k w_{k,c} f_{h,w,k}$$

$$M_{c,h,w} = \sum_k w_{k,c} f_{h,w,k}$$

Class Activation Map

Class Activation Maps

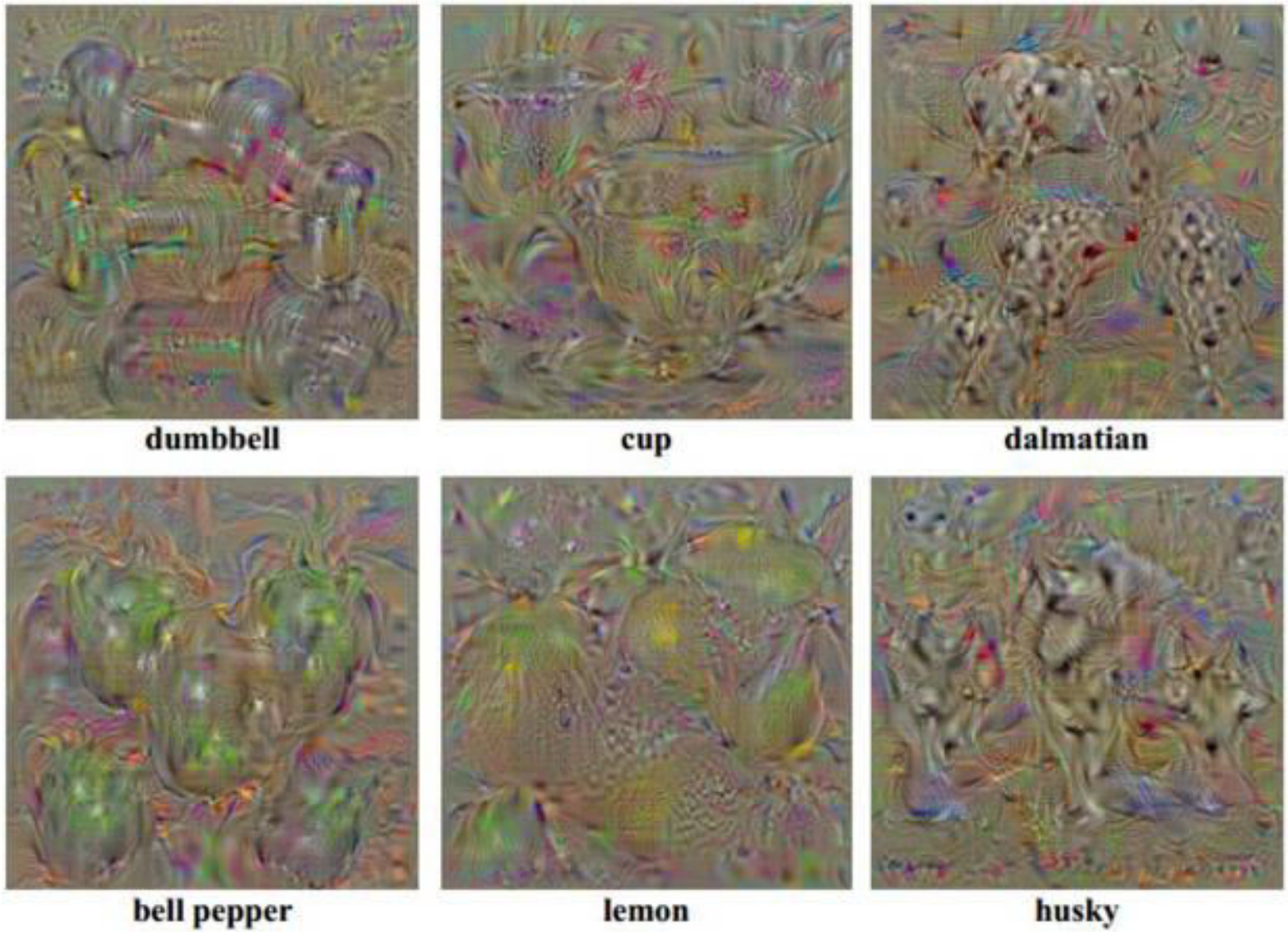


Gradient Ascent

$$I^\star = \arg\max_I \{S_c(I) - \lambda \|I\|^2\}$$

- Initialize image $I = 0$
- Forward image to compute scores $S_c(I)$
- Backprop to get gradients of neuron values wrt image pixels
- Update image as $I \leftarrow I + \frac{\partial S_c(I)}{\partial I}$

Gradient Ascent



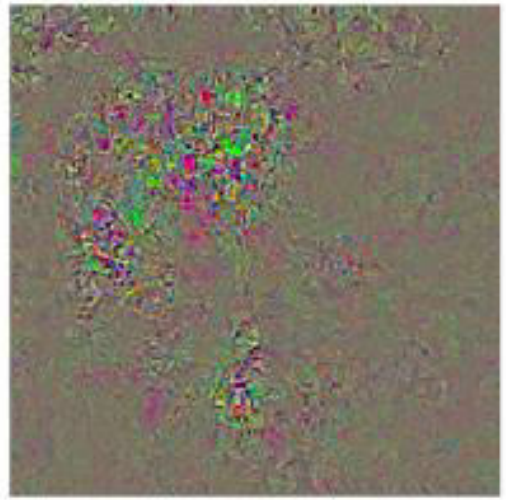
Adversarial Training: Examples

African elephant

koala

Difference

10x Difference

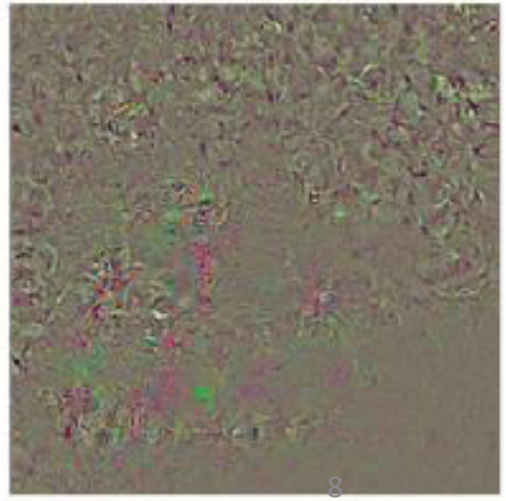


schooner

iPod

Difference

10x Difference

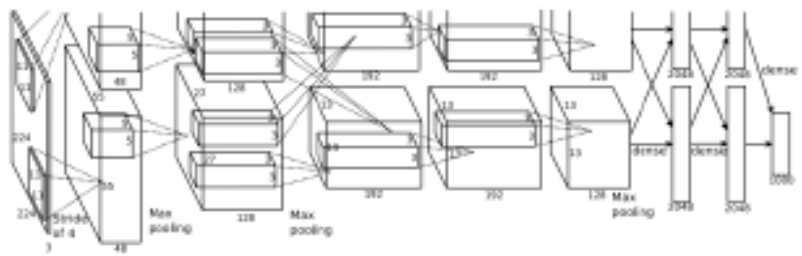


Feature Inversion

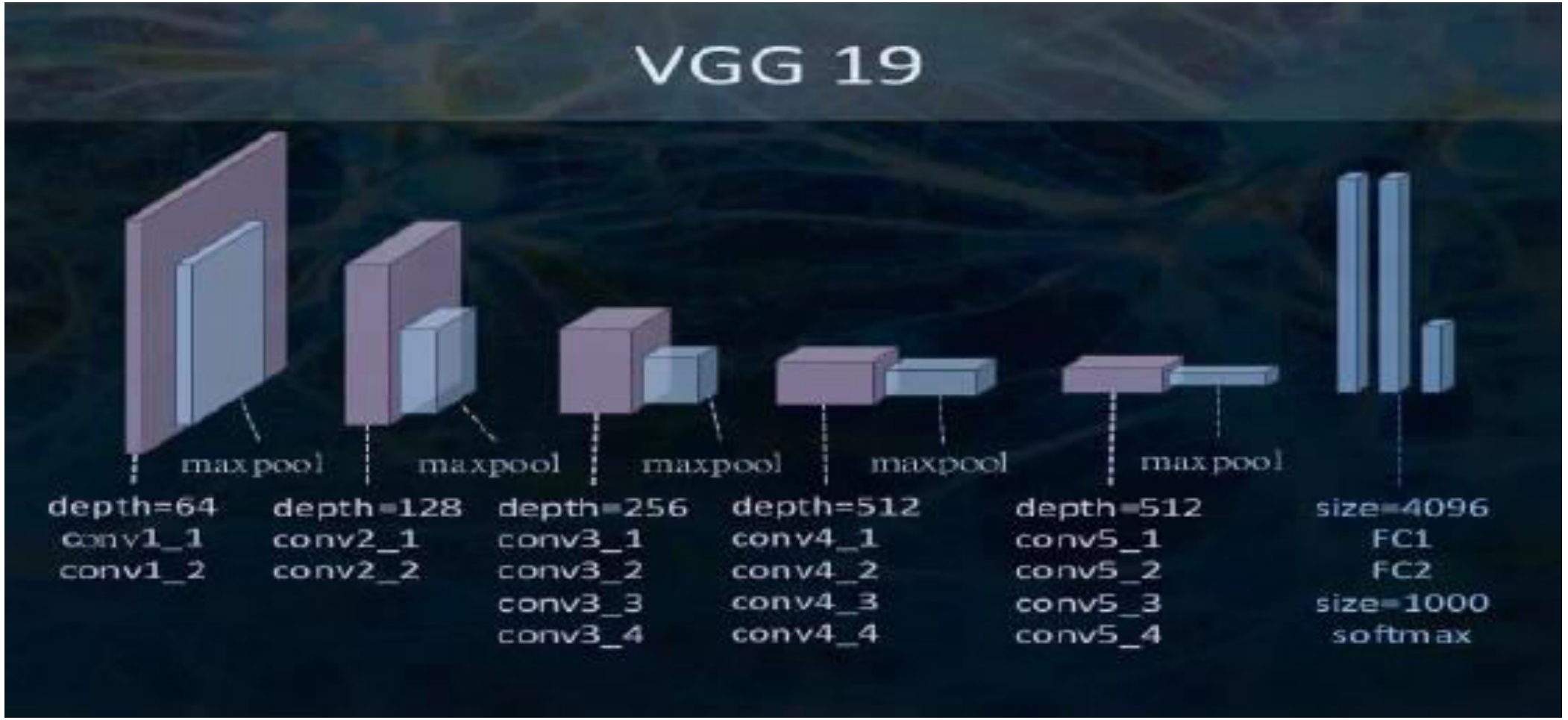
$$x^\star = \arg\min_I \{ l(\Phi(I), \Phi_0) + \lambda \mathcal{R}(I) \}$$

$$l(\Phi(I), \Phi_0) = \|\Phi(I) - \Phi_0\|^2$$

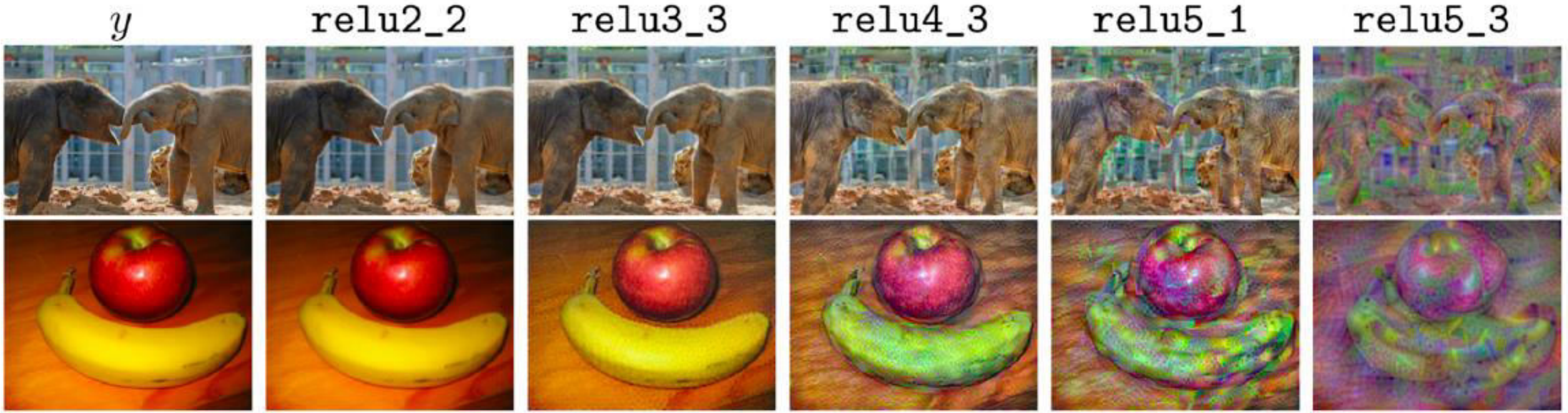
$$\mathcal{R}_\beta(I) = \sum_{i,j} \left(I(i, j+1) - I(i,j) \right)^2 + \left(I(i+1, j) - I(i,j) \right)^2 \Big)^{\frac{\beta}{2}}$$

I  $\Phi(I)$

VGG Net


VGG 19
maxpool
maxpool
maxpool
maxpool
maxpool
depth=64
conv1_1
conv1_2
depth=128
conv2_1
conv2_2
depth=256
conv3_1
conv3_2
conv3_3
conv3_4
depth=512
conv4_1
conv4_2
conv4_3
conv4_4
depth=512
conv5_1
conv5_2
conv5_3
conv5_4
size=4096
FC1
FC2
size=1000
softmax

Feature Inversion

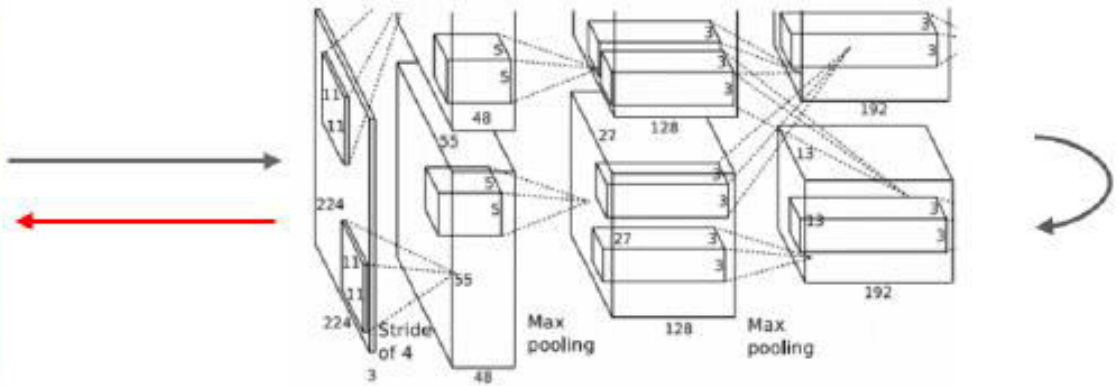


Deep Dream: Amplify Features

Inceptionism: Going Deeper into Neural Networks

Wednesday, June 17, 2015

Posted by Alexander Mordvintsev, Software Engineer, Christopher Olah, Software Engineering Intern and Mike Tyka, Software Engineer



- Compute activations at chosen layer via forward pass
- Set gradient of chosen layer equal to its activation
- Compute gradient on image via backprop
- Update image

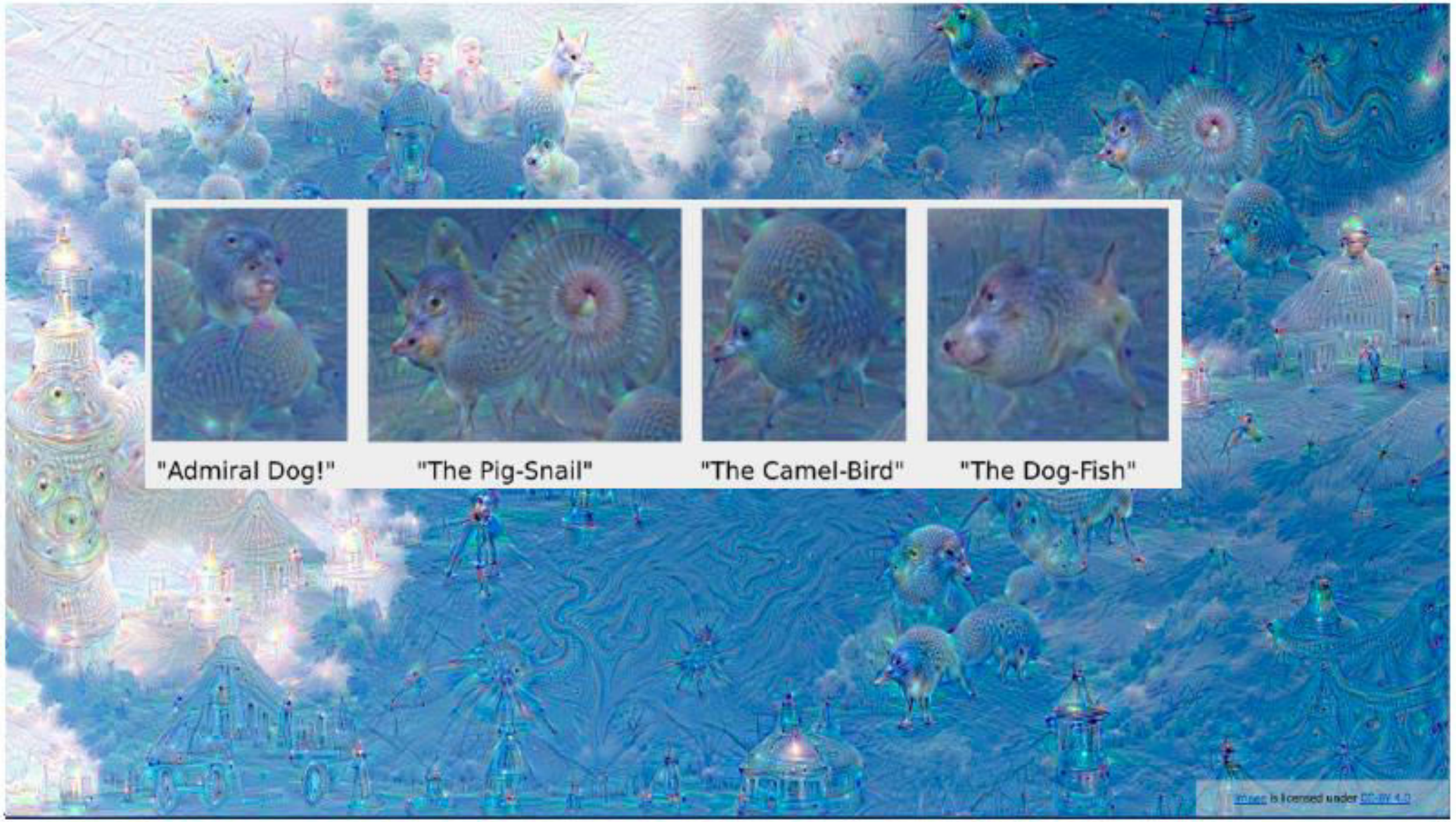
$$I^{\star} = \arg\max_{I} \sum_{i} f_i(I)^2$$

DeepDream

DeepDream

DeepDream

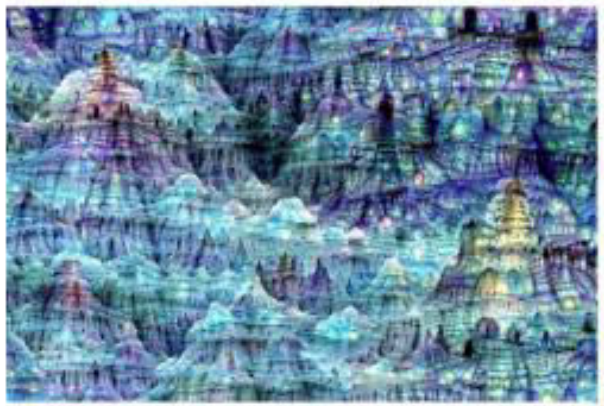
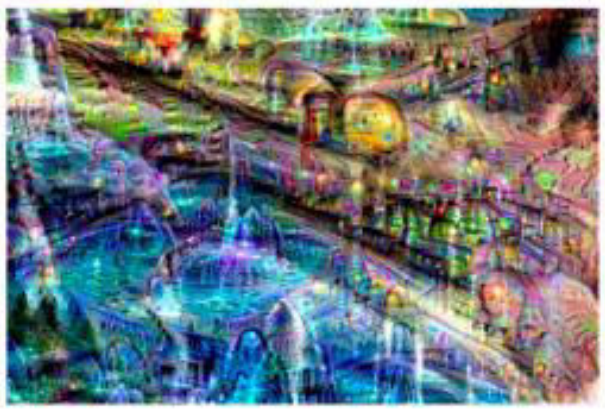
DeepDream

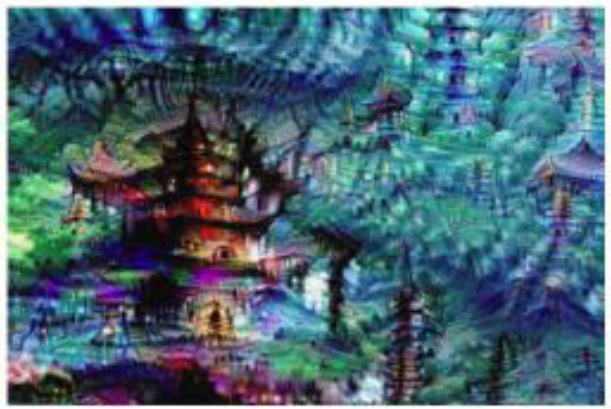


DeepDream

DeepDream



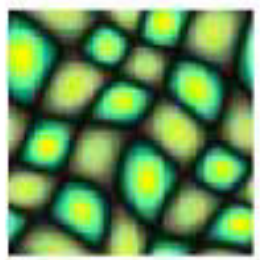


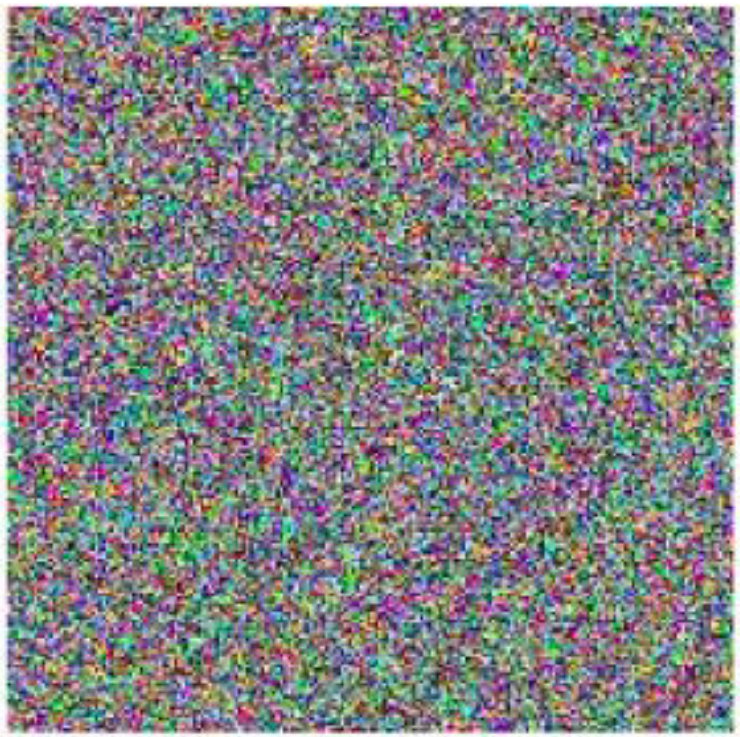
Texture Synthesis

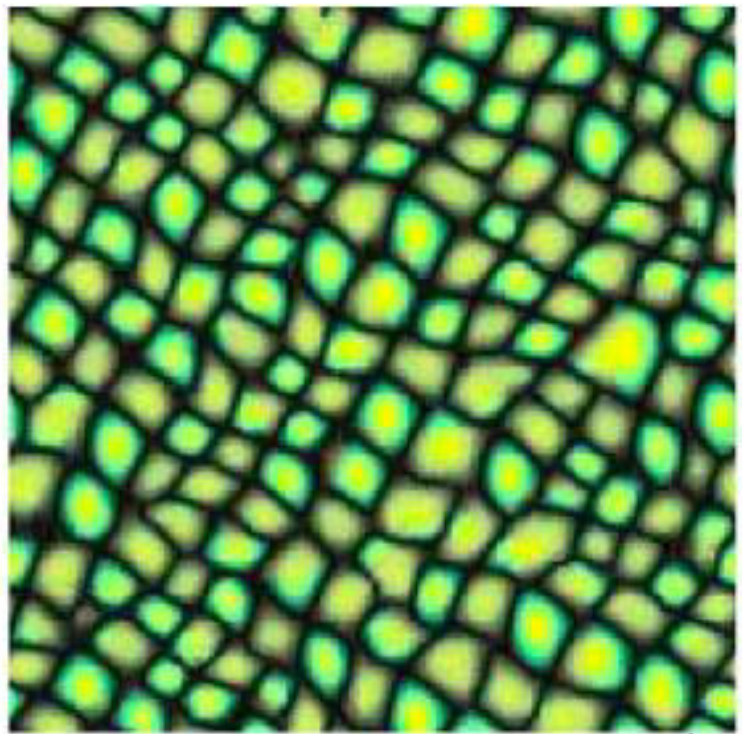
Fast Texture Synthesis using Tree-structured Vector Quantization

Li-Yi Wei Marc Levoy

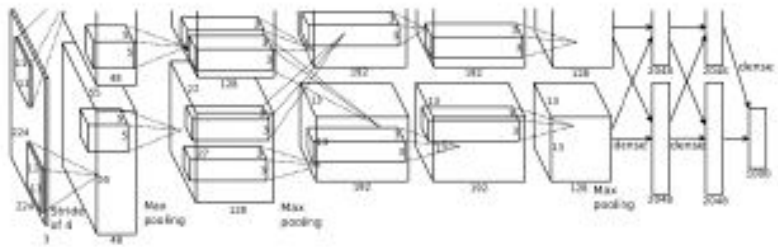
Stanford University *

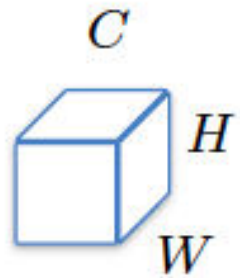






Neural Texture Synthesis: Gram Matrix



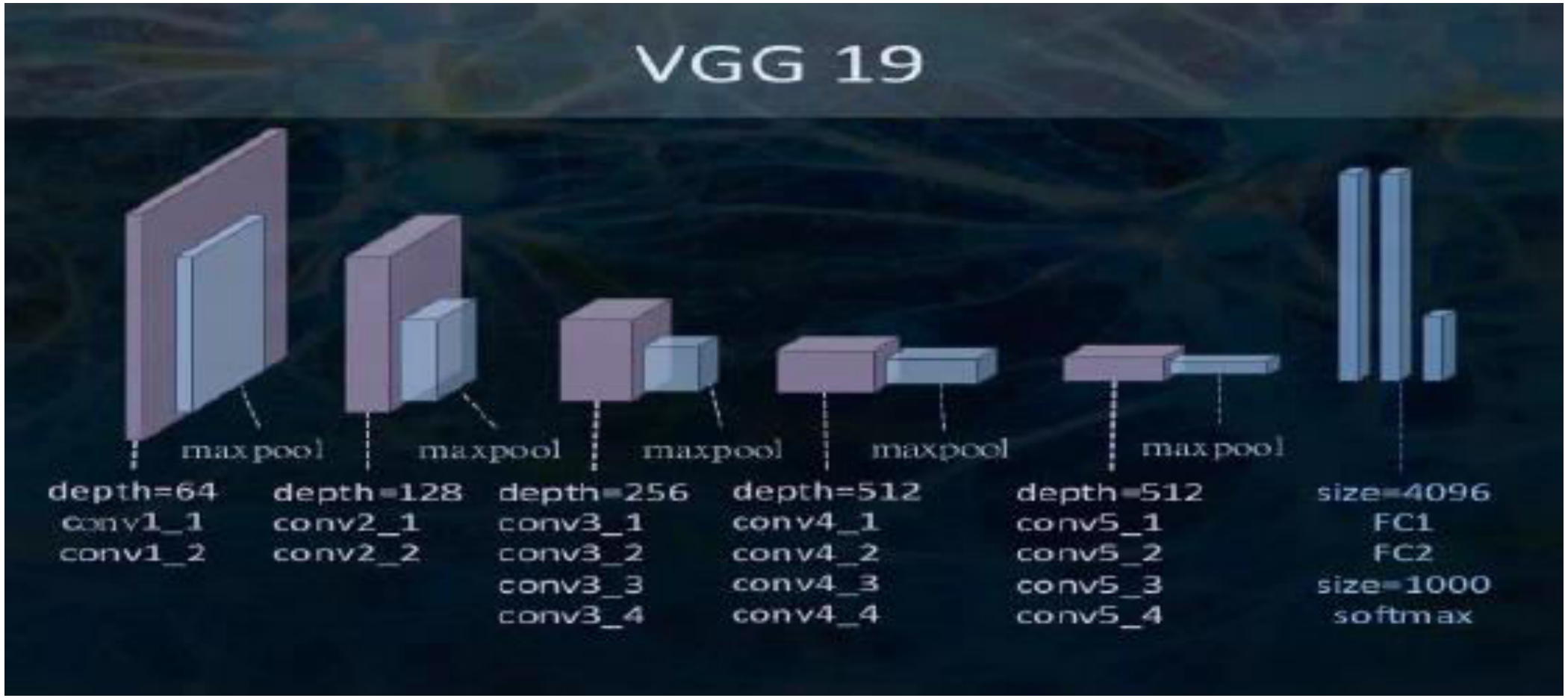


G

Outer product of two C-dimensional vectors outputs C times C matrix (covariance matrix)

$$G := \sum_{i=1}^{HW} \mathbf{x}_i \mathbf{x}_i^T$$

VGG Net

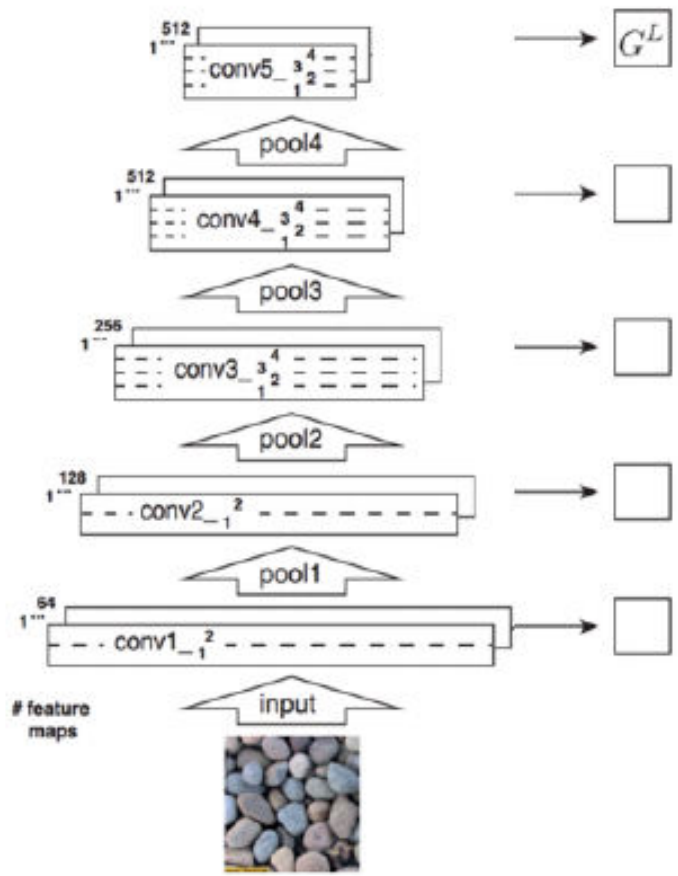

VGG 19
maxpool
maxpool
maxpool
maxpool
maxpool
depth=64
conv1_1
conv1_2
depth=128
conv2_1
conv2_2
depth=256
conv3_1
conv3_2
conv3_3
conv3_4
depth=512
conv4_1
conv4_2
conv4_3
conv4_4
depth=512
conv5_1
conv5_2
conv5_3
conv5_4
size=4096
FC1
FC2
size=1000
softmax

Neural Texture Synthesis

1. Compute VGG features.
2. Given image I, compute features at different levels.
3. Compute Gram matrices at different levels.

$$G^l_{ij} := \sum_k F^l_{ik} F^l_{jk}$$

4. Initialize with random (noise) matrix.



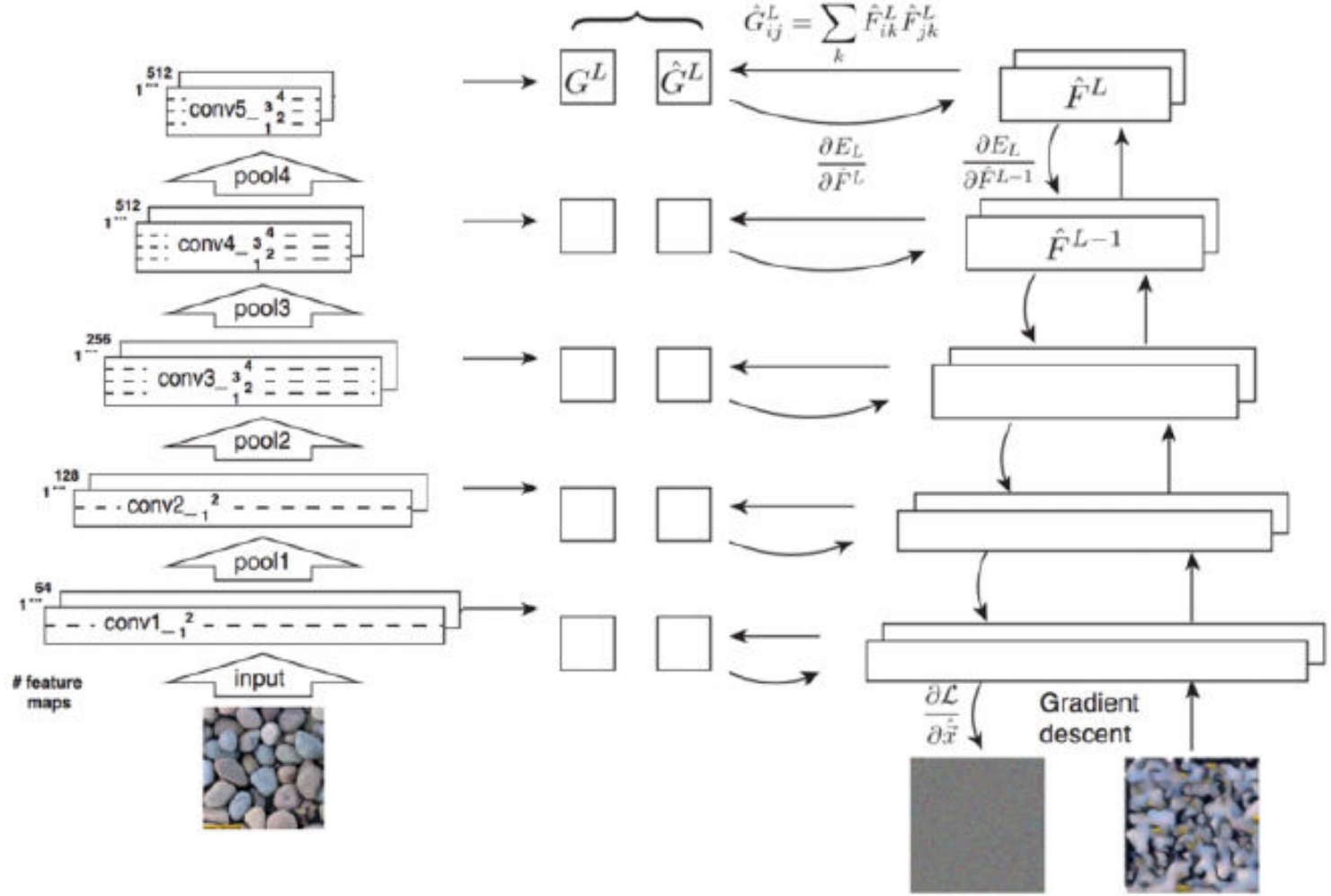
Neural Texture Synthesis

$$E_l = \frac{1}{N_l^2 M_l^2} \sum_{i,j} \left(G_{ij}^l - \hat{G}_{ij}^l \right)^2 \qquad \mathcal{L}(I) := \sum_l \alpha_l E_l$$

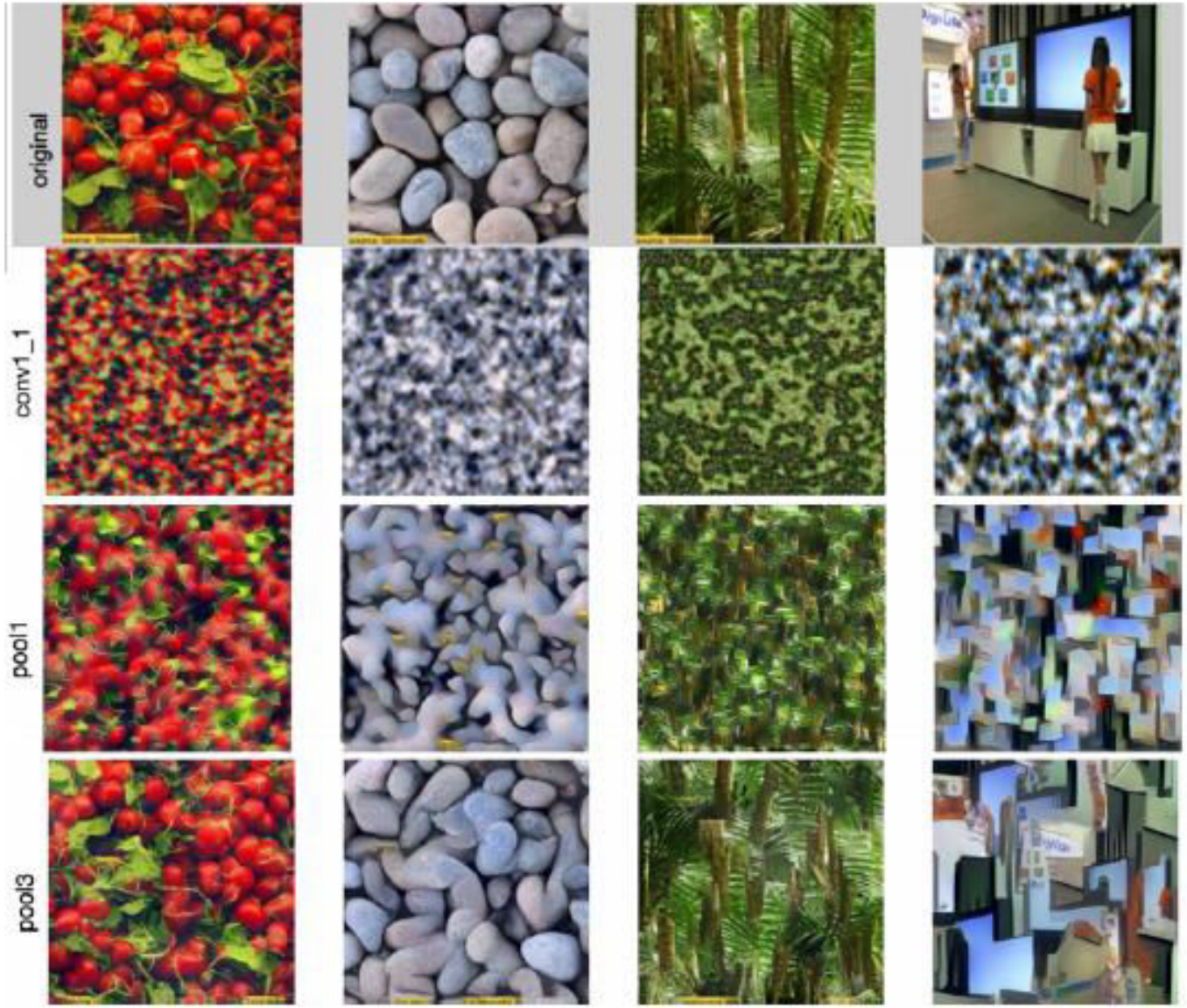
1. Compute VGG features.
2. Given image I, compute features at different levels.
3. Compute Gram matrices at different levels.

$$G_{ij}^l := \sum_k F_{ik}^l F_{jk}^l$$

4. Initialize with random (noise) matrix.
5. Compute features at each level.
6. Compute loss, backprop wrt image pixel. Goto step 5.

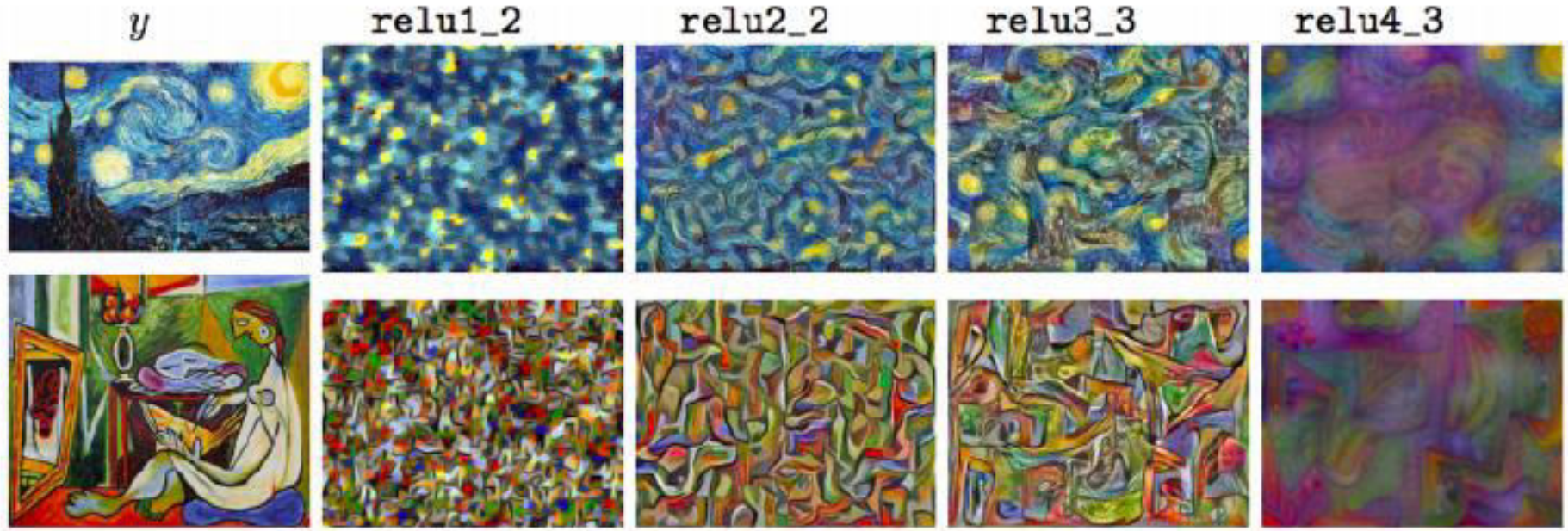


Neural Texture Synthesis



More Texture Synthesis

Texture synthesis (Gram reconstruction)



Neural Style Transfer

Content Image

+

Style Image

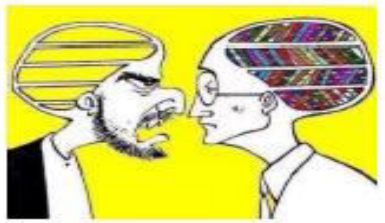
=

Output Image

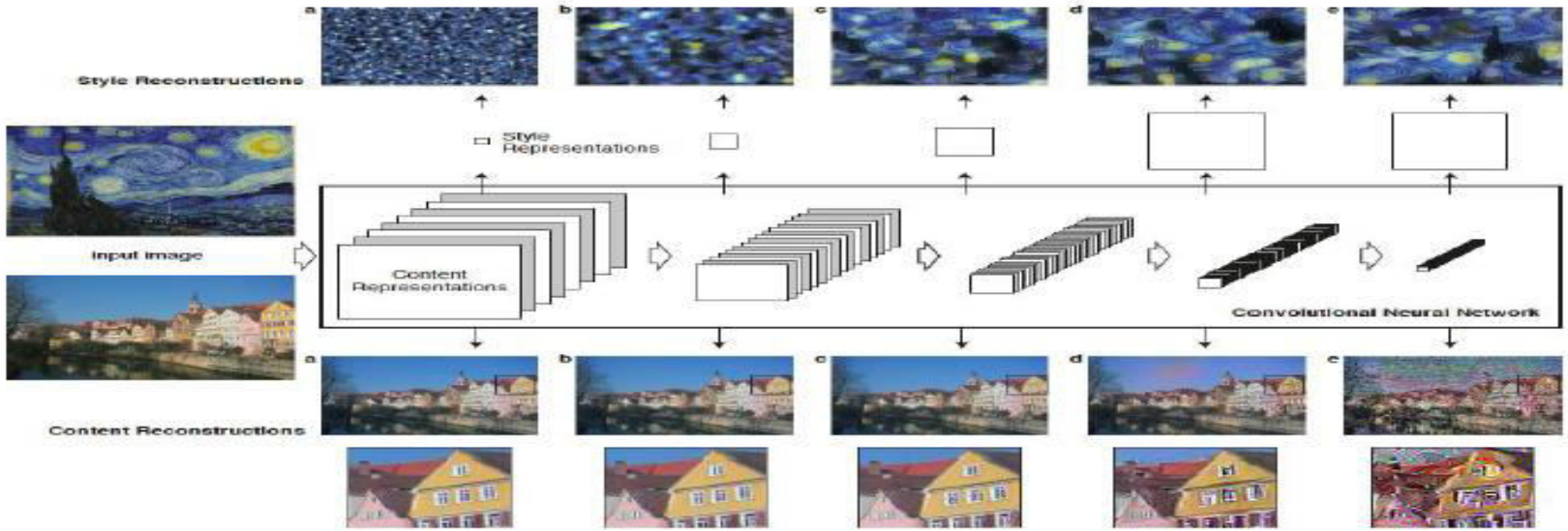
Neural Style Transfer

- Content: Global structure. Style: Colours; local structures
  Like naturalistic, photographic, abstract, symbolistic



- Use CNNs to capture style from one image and content from another image.



- Feature representations → Content
  Filter correlations → Style

Reconstructing an image from a convolutional layer

- Representation function: $\Phi : \Re^{H \times W \times C} \rightarrow \Re^{d}$ (image space to feature space)
- Target Representation: $\Phi_0 = \Phi(x_0)$ (x0 is the original image)
- We need to find: $x \in \Re^{H \times W \times C}$ by minimizing:

$$x^* = \arg\min_{x \in \Re^{H \times W \times C}} l(\Phi(x), \Phi_0) + \lambda R(x)$$

"Understanding Deep Image Representations by Inverting Them", by Aravindh Mahendran and Andrea Vedaldi.

Content Reconstruction

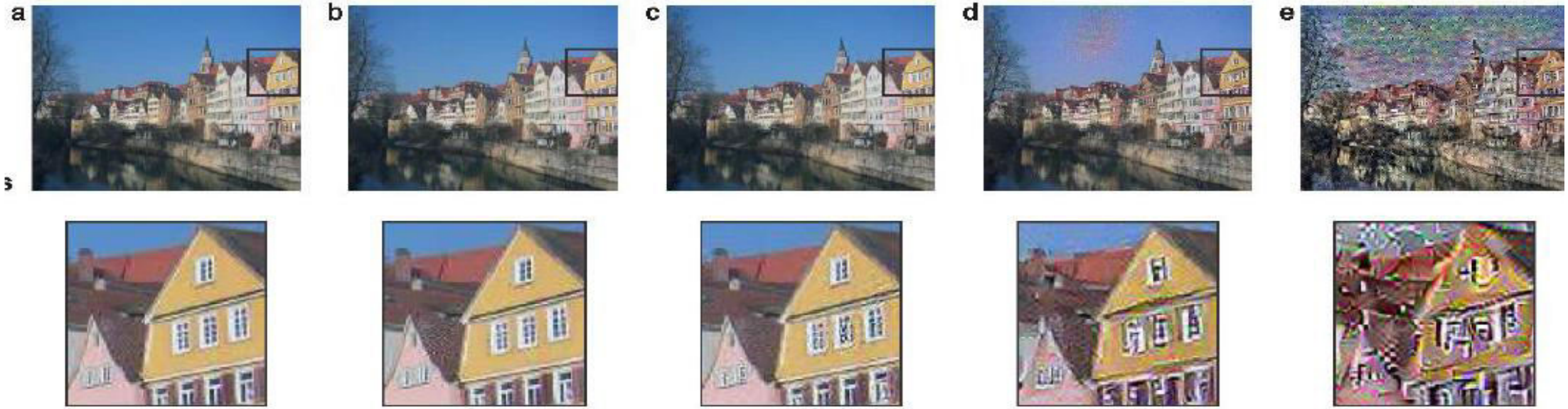


Image reconstructed from layers
(a)'conv1_1',
(b)'conv2_1',
(c)'conv3_1',
(d)'conv4_1' and
(e)'conv5_1'
of the original VGG-Network

Content Loss Function

- Filters (Depths) at layer l: N_l
- The height times the width of the feature map at layer l: M_l
- Response at layer l: $F_l \in \Re^{N_l \times M_l}$

  $F^l{}_{ij}$ represents the ith filter at position j in layer l
- Original image: $\vec{p}$
- We generate image: $\vec{x}$ (randomly initialized)
- Squared-error loss:

$$L_{content} = \frac{1}{2} \sum_{i,j} (F^l{}_{ij} - P^l{}_{ij})^2$$

Style Reconstruction

Style representations (filer correlations) from:
(a) 'conv1_1',
(b) 'conv1_1', 'conv2_1',
(c) 'conv1_1', 'conv2_1', 'conv3_1',
(d) 'conv1_1', 'conv2_1', 'conv3_1', 'conv4_1',
(e) 'conv1_1', 'conv2_1', 'conv3_1', 'conv4_1', 'conv5_1'.

Style Loss Function

- Filter correlations are given by the Gram matrix:

$$G^l \in \Re^{N_l \times N_l}$$

- $G^l{}_{ij}$ is the inner product between the filters i and j in layer l:

$$G^l{}_{ij} = \sum_k F^l{}_{ik} F^l{}_{jk}$$

- The loss at layer l:

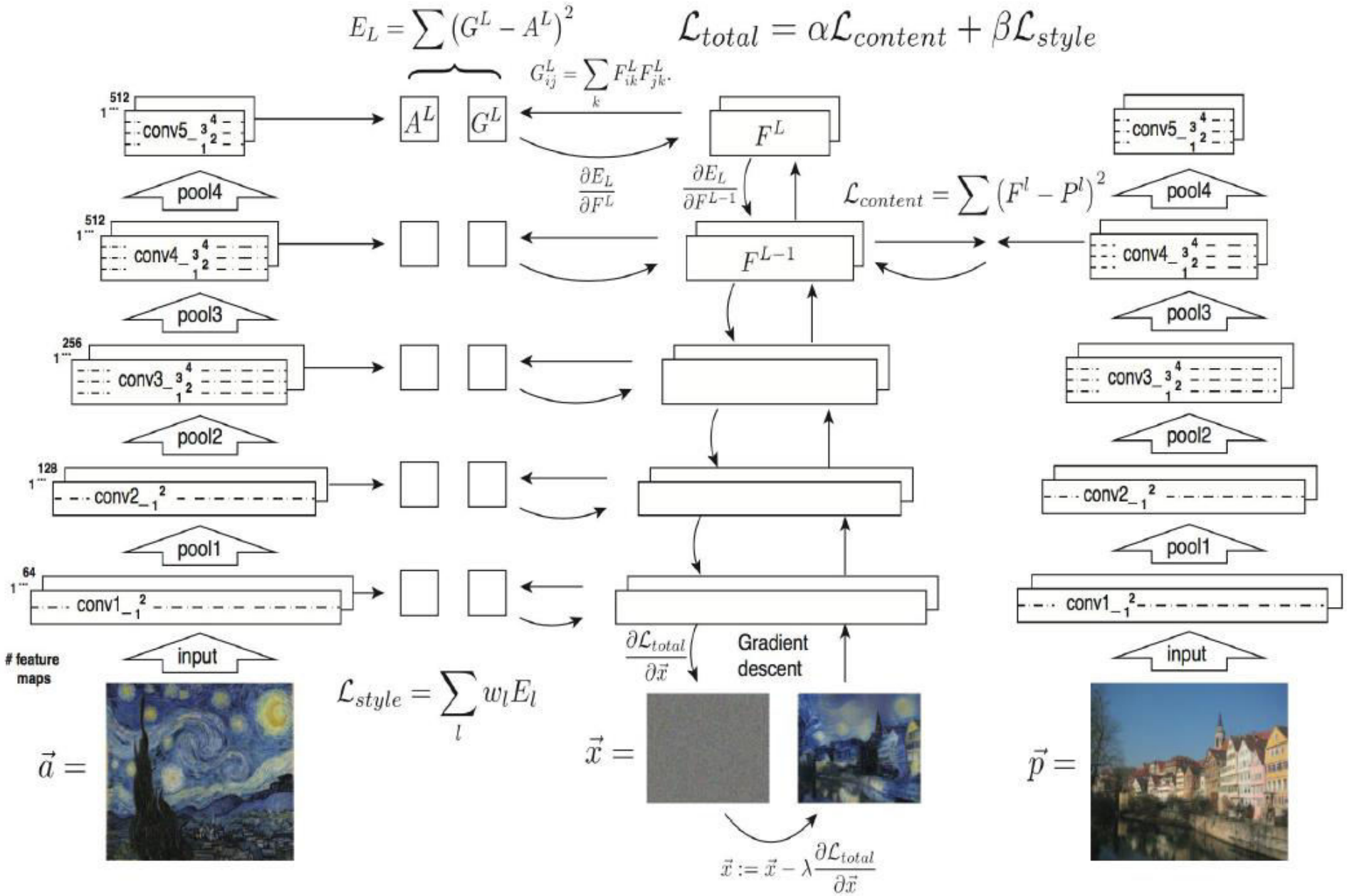
$$E_l = \frac{1}{4N^2{}_l M^2{}_l} \sum_{i,j} (G^l{}_{ij} - A^l{}_{ij})^2$$

A <-> original image
G <-> generated image

- The total style loss:

$$L_{style} = \sum_{l=0}^{L} w_l E_l$$

The Total Loss Function




A
B
C
D
E
F

10^{-4}

10^{-3}

10^{-2}

10^{-1}

$$\mathcal{L}_{total} = \alpha \mathcal{L}_{content} + \beta \mathcal{L}_{style}$$

References

- https://smartgeometry-ucl.github.io/comp0169-20.github.io/

Image Style Transfer Using Convolutional Neural Networks

Leon A. Gatys
Centre for Integrative Neuroscience, University of Tübingen, Germany
Bernstein Center for Computational Neuroscience, Tübingen, Germany
Graduate School of Neural Information Processing, University of Tübingen, Germany
leon.gatys@bethgelab.org

Alexander S. Ecker
Centre for Integrative Neuroscience, University of Tübingen, Germany
Bernstein Center for Computational Neuroscience, Tübingen, Germany
Max Planck Institute for Biological Cybernetics, Tübingen, Germany
Baylor College of Medicine, Houston, TX, USA

Matthias Bethge
Centre for Integrative Neuroscience, University of Tübingen, Germany
Bernstein Center for Computational Neuroscience, Tübingen, Germany
Max Planck Institute for Biological Cybernetics, Tübingen, Germany