Introduction to CNNs and RNNs

Nojun Kwak
nojunk@snu.ac.kr
http://mipal.snu.ac.kr

GSCST, Seoul National University, Korea

Aug. 2016

Many slides on CNNs are from *Fei-Fei Li @Stanford*, *Raquel Urtasun @U. Toronto*, and *Marc'Aurelio Ranzato @Facebook*.
RNN parts are based on Colah's blog: http://colah.github.io/
and the slides of Daniel Renshaw

Outline

- Traditional Neural Networks (MLPs, $\cdots$)
- Convolutional Neural Networks (CNNs)
- Recurrent Neural Networks (RNNs)
- Applications

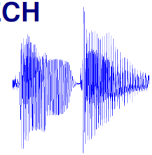
Traditional Pattern Recognition

VISION

→ **SIFT/HOG** → **K-Means/ pooling** → **classifier** → "car"

fixed | unsupervised | supervised

SPEECH



→ **MFCC** → **Mixture of Gaussians** → **classifier** → \'d ē p\

fixed | unsupervised | supervised

NLP

This burrito place is yummy and fun! → **Parse Tree Syntactic** → **n-grams** → **classifier** → "+"

fixed | unsupervised | supervised

Hierarchical Compositionality (DEEP)

VISION

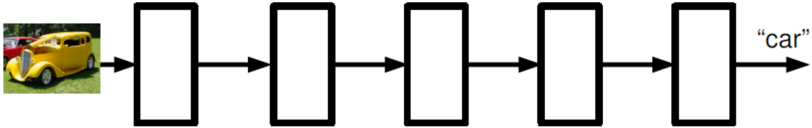
pixels → edge → texton → motif → part → object

SPEECH

sample → spectral band → formant → motif → phone → word

NLP

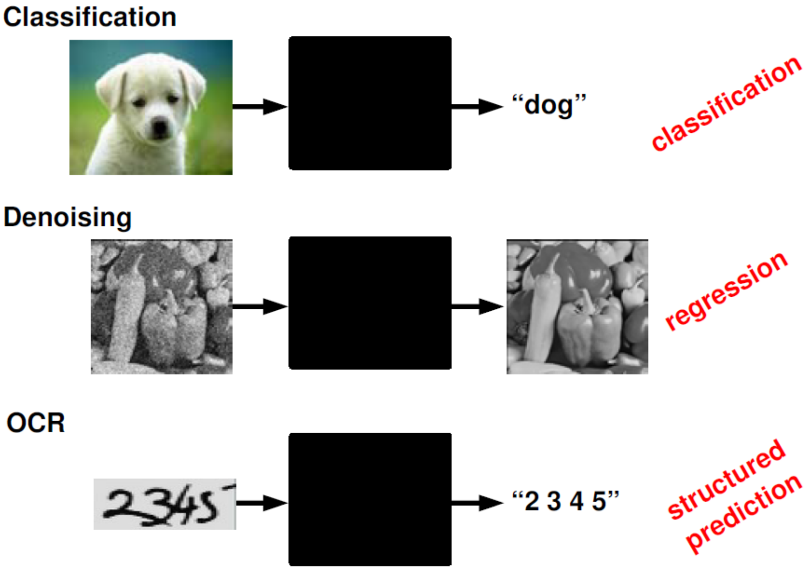
character → word → NP/VP/.. → clause → sentence → story



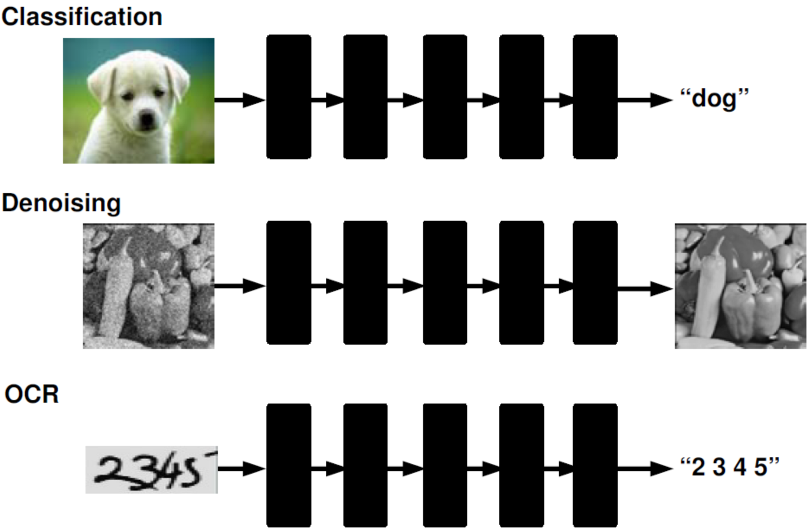
What is deep learning?

- Nothing new!
- (Many) cascades of nonlinear transformations
- End-to-end learning (no human intervention / no fixed features)

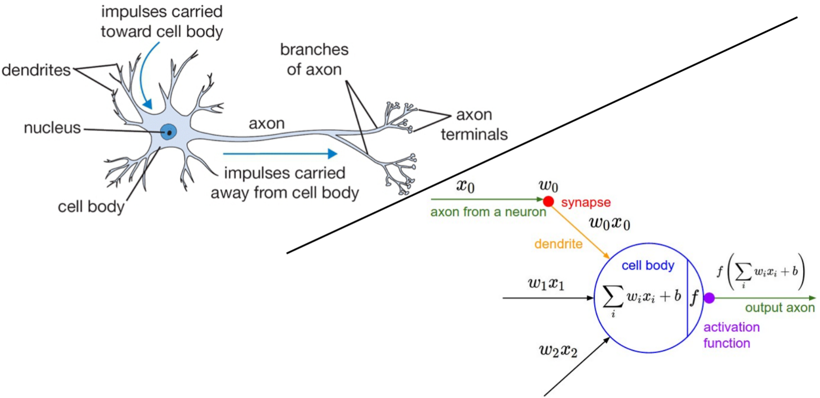
Shallow(?) Learning Examples - Supervised

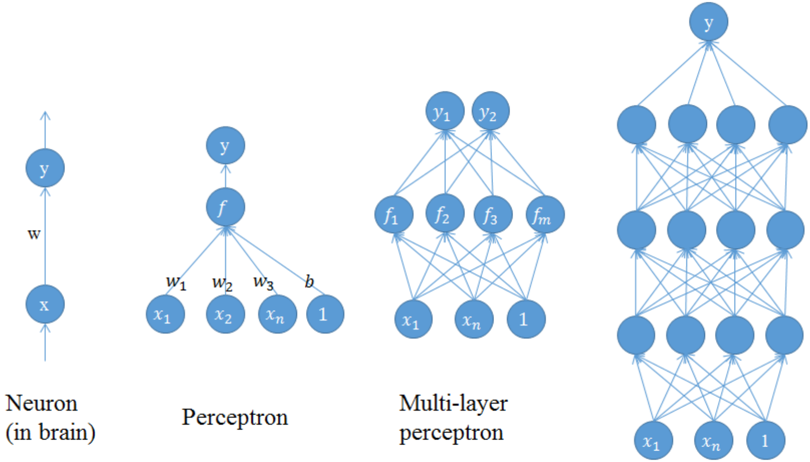


Deep Learning Examples - Supervised



- Biologically inspired models





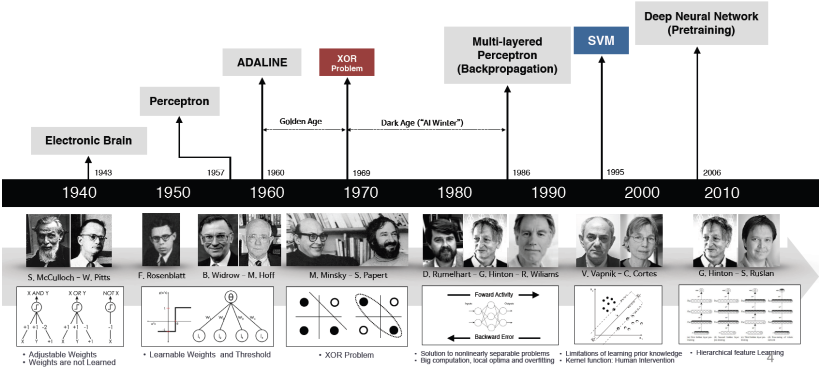
Neuron (in brain)

Perceptron

Multi-layer perceptron

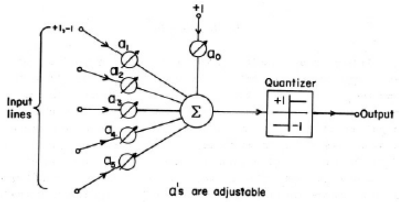
Deep neural network

A Brief History of ANNs

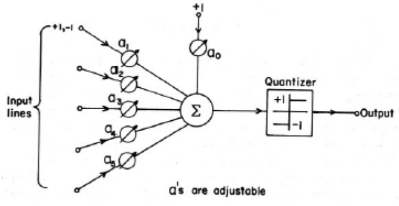


source of image: VUNO Inc.

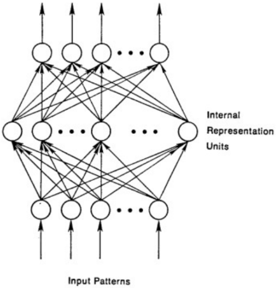
- First Generation: 1957 $\sim$
  - Perceptron: Rosenblatt, 1957
  - Adaline: Widrow and Hoff, 1960



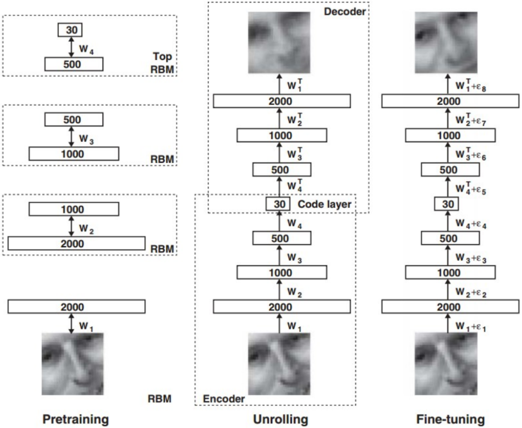
- First Generation: 1957 ∼
  - Perceptron: Rosenblatt, 1957
  - Adaline: Widrow and Hoff, 1960



- Second Generation: 1986 ∼
  - MLP with BP: Rumelhart



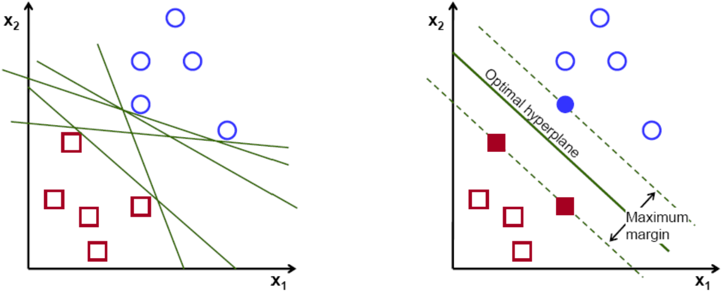
- Third Generation: 2006 ∼
  - RBM: Hinton and Salkhutdinov
  - Reinvigorated research in Deep Learning



- Given inputs $\boldsymbol{x}$, and outputs $t \in \{-1, 1\}$
- Find a hyperplane that divides the space into half (binary classification)

$$y_* = \text{sign}(\boldsymbol{w}_*^T \boldsymbol{x} + \boldsymbol{w}_0)$$

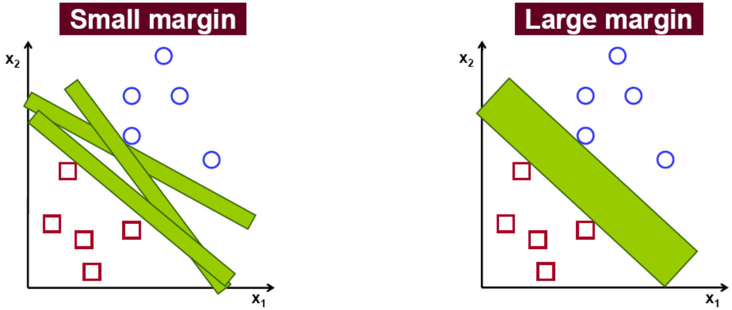
$\Rightarrow$ SVM tries to maximize the margin.



- Given inputs $\boldsymbol{x}$, and outputs $t \in \{-1, 1\}$
- Find a hyperplane that divides the space into half (binary classification)

$$y = \text{sign}(\boldsymbol{w}^T\boldsymbol{x} + b)$$

$\Rightarrow$ SVM tries to maximize the margin.



How can we make our classifier more powerful?

Nonlinear predictors

How can we make our classifier more powerful?

- Compute nonlinear functions of the input

$$y = F(\boldsymbol{x}, \boldsymbol{w})$$

How can we make our classifier more powerful?

- Compute nonlinear functions of the input

$$y = F(\boldsymbol{x}, \boldsymbol{w})$$

Two types of widely used approaches

How can we make our classifier more powerful?

- Compute nonlinear functions of the input

$$y = F(\boldsymbol{x}, \boldsymbol{w})$$

Two types of widely used approaches

- Kernel Trick: Fixed functions and optimize linear parameters on nonlinear mappings $\phi(\boldsymbol{x})$

$$y = \mathsf{sign}(\boldsymbol{w}^T \phi(\boldsymbol{x}) + b)$$

How can we make our classifier more powerful?

- Compute nonlinear functions of the input

$$y = F(\boldsymbol{x}, \boldsymbol{w})$$

Two types of widely used approaches

- Kernel Trick: Fixed functions and optimize linear parameters on nonlinear mappings $\phi(\boldsymbol{x})$

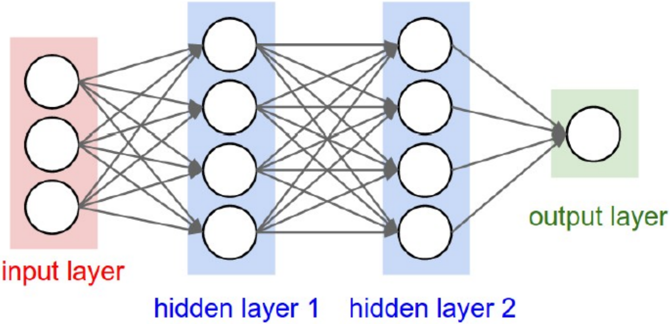
$$y = \text{sign}(\boldsymbol{w}^T \phi(\boldsymbol{x}) + b)$$

- Deep Learning: Learn parametric nonlinear functions

$$y = F(\boldsymbol{x}, \boldsymbol{w}) = \cdots (\boldsymbol{h}_2(\boldsymbol{w}_2^T \boldsymbol{h}_1(\boldsymbol{w}_1^T \boldsymbol{x} + b_1) + b_2) \cdots$$

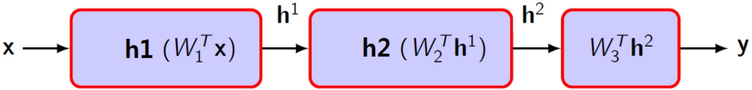
$\boldsymbol{h}_{1,2}$: activation function at layer 1 or 2

- Deep learning uses composite of simpler functions, e.g., ReLU, sigmoid, tanh, max
- Note: a composite of linear functions is linear!
- Example: 2 layer NNet (Convention: input and output layers are not taken as a layer)

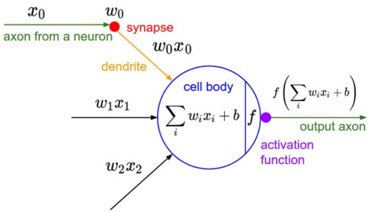


Neural Networks

- Deep learning uses composite of simpler functions, e.g., ReLU, sigmoid, tanh, max
- Note: a composite of linear functions is linear!
- Example: 2 layer NNet (Convention: input and output layers are not taken as a layer)



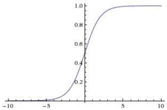
  - $\boldsymbol{x}$ is the input
  - $\boldsymbol{y}$ is the output
  - $\boldsymbol{h}^i$ is the i-th hidden layer output
  - W^i is the set of parameters of the i-th layer



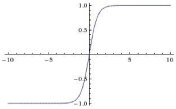
- A singlue neuron can be used as a binary linear classifier
- Regularization has the interpretation of gradual forgetting

Classical NNs used sigmoid or tanh function as an activation function.

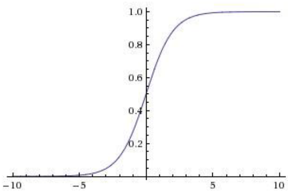
- sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$
- tanh: $\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$



Sigmoid

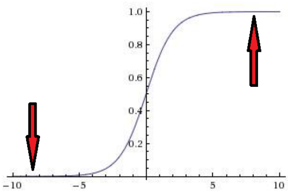


tanh(x)



Sigmoid

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

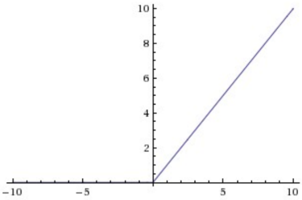


Sigmoid

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

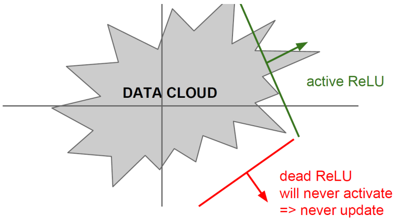
2 BIG problems:

1. Saturated neurons kill the gradients (cannot backprop further)
   $\Rightarrow$ Major bottleneck for the conventional NNs: not able to train more than 2 or 3 layers
2. Sigmoid outputs are not zero-centered
   $\Rightarrow$ Restriction on the gradient directions



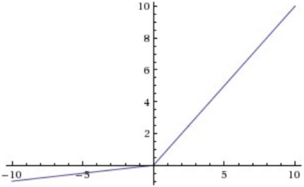
ReLU

- $f(x) = \max(0, x)$
- Does not saturate
- Computationally very efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)



- $f(x) = \max(0, x)$
- Does not saturate
- Computationally very efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

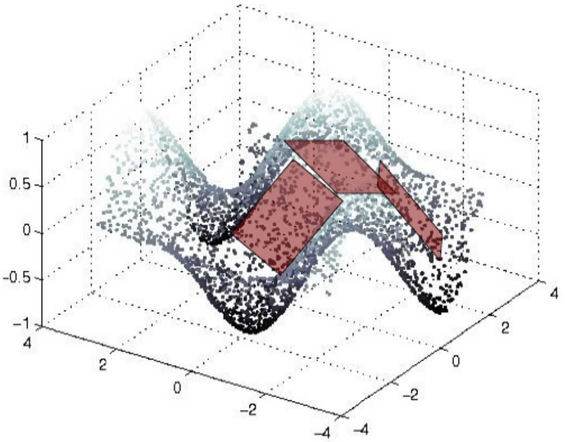
- One annoying problem $\Rightarrow$ Dead neurons



Leaky ReLU

- $f(x) = \max(0, x)$
- Does not saturate
- Computationally very efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

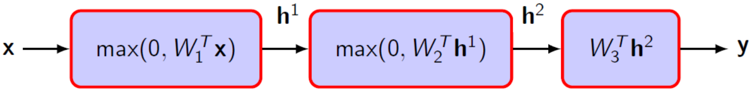
- One annoying problem $\Rightarrow$ Dead neurons
- Solution: leaky ReLU (small slope for negative input)
  - Never dies.
  - However, almost the same performance in practice.



Piecewise linear tiling: mapping is locally linear

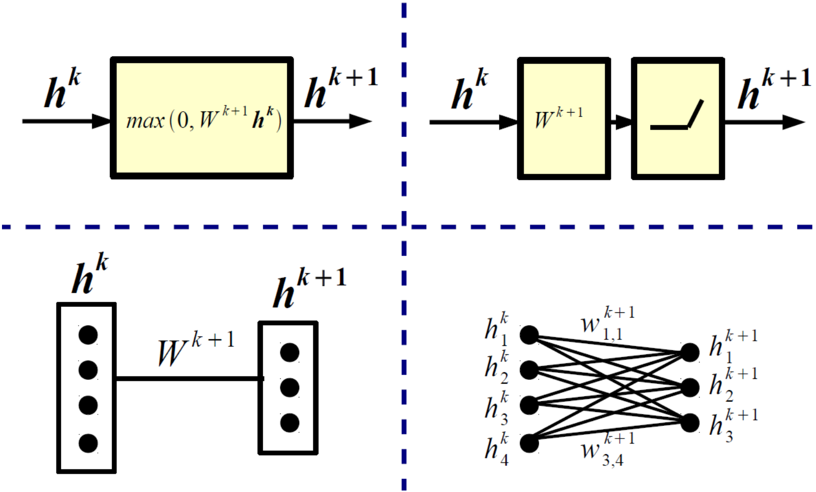
Montufar et al. "On the number of linear regions of DNNs", arXiv 2014

- Forward propagation: compute the output $\boldsymbol{y}$ given the input $\boldsymbol{x}$



- Fully connected layer
- Nonlinearity comes from ReLU
- Do it in a compositional way

$$\boldsymbol{x} \Rightarrow \boldsymbol{h}^1 \Rightarrow \boldsymbol{h}^2 \Rightarrow \boldsymbol{y}$$



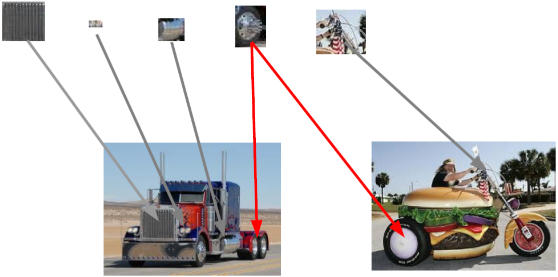
Slide from M. Ranzato

Why many layers?

Hierarchically distributed representations

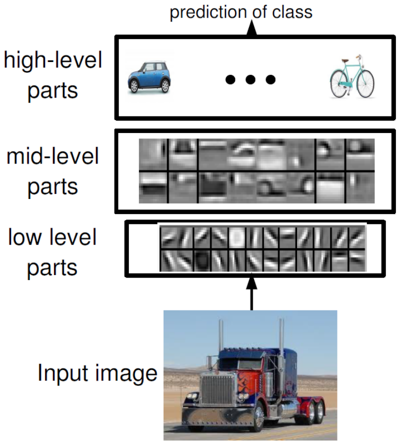
[1 1 0 0 0 1 0 **1** 0 0 0 0 1 1 0 1 . . .] motorbike

[0 0 1 0 0 0 0 **1** 0 0 1 1 0 0 1 0 . . .] truck



Lee et al. "Convolutional DBN's · · · " ICML 2009

Hierarchically distributed representations



- distributed representations
- feature sharing
- compositionality

Lee et al. “Convolutional DBN's · · · ” ICML 2009

Training

$$\mathbf{x} \longrightarrow \boxed{\max(0, W_1^T\mathbf{x})} \xrightarrow{\mathbf{h}^1} \boxed{\max(0, W_2^T\mathbf{h}^1)} \xrightarrow{\mathbf{h}^2} \boxed{W_3^T\mathbf{h}^2} \longrightarrow \mathbf{y}$$

- We want to estimate the parameters, biases and hyper-parameters (e.g., number of layers, number of neurons) for good predictions.
- Collect a training set of input-output pairs $\{\boldsymbol{x}_i, t_i\}_{i=1}^N$.
- Encode the output with 1-K encoding $t = [0, \cdots, 1, \cdots, 0]$.

Training

$$\mathbf{x} \longrightarrow \boxed{\max(0, W_1^T \mathbf{x})} \xrightarrow{\mathbf{h}^1} \boxed{\max(0, W_2^T \mathbf{h}^1)} \xrightarrow{\mathbf{h}^2} \boxed{W_3^T \mathbf{h}^2} \longrightarrow \mathbf{y}$$

- We want to estimate the parameters, biases and hyper-parameters (e.g., number of layers, number of neurons) for good predictions.
- Collect a training set of input-output pairs $\{\boldsymbol{x}_i, t_i\}_{i=1}^N$.
- Encode the output with 1-K encoding $t = [0, \cdots, 1, \cdots, 0]$.
- Define a loss per training example and minimize the empirical loss

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{N}\sum_{i=1}^{N} l(\boldsymbol{w}, \boldsymbol{x}_i, t_i) + \mathcal{R}(\boldsymbol{w})$$

N: number of training examples

$\mathcal{R}$: regularizer

$\boldsymbol{w}$: set of all parameters

Loss functions

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{N}\sum_{i=1}^{N} l(\boldsymbol{w}, \boldsymbol{x}_i, t_i) + \mathcal{R}(\boldsymbol{w})$$

- Softmax (Probability of class k given input):

$$p(c_k = 1|\boldsymbol{x}) = \frac{\exp(y_k)}{\sum_{j=1}^{C}\exp(y_j)}$$

Loss functions

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{N}\sum_{i=1}^{N} l(\boldsymbol{w}, \boldsymbol{x}_i, t_i) + \mathcal{R}(\boldsymbol{w})$$

- Softmax (Probability of class k given input):

$$p(c_k = 1|\boldsymbol{x}) = \frac{\exp(y_k)}{\sum_{j=1}^{C}\exp(y_j)}$$

- Cross entropy (most popular loss function for classification):

$$l(\boldsymbol{w}, \boldsymbol{x}, t) = -\sum_{k=1}^{C} t^{(k)} \log p(c_k|\boldsymbol{x})$$

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{N}\sum_{i=1}^{N} l(\boldsymbol{w}, \boldsymbol{x}_i, t_i) + \mathcal{R}(\boldsymbol{w})$$

- Softmax (Probability of class k given input):

$$p(c_k = 1|\boldsymbol{x}) = \frac{\exp(y_k)}{\sum_{j=1}^{C}\exp(y_j)}$$

- Cross entropy (most popular loss function for classification):

$$l(\boldsymbol{w}, \boldsymbol{x}, t) = -\sum_{k=1}^{C} t^{(k)} \log p(c_k|\boldsymbol{x})$$

- Gradient descent to train the network

$$\boldsymbol{w}^* = \underset{\boldsymbol{w}}{\operatorname{argmin}}\, \mathcal{L}(\boldsymbol{w})$$

Backpropagation

- Efficient way of computing gradient (Chain rule)
- Partial derivatives and gradients

$$f(x,y) = xy \quad \rightarrow \quad \frac{\partial f}{\partial x} = y \quad \frac{\partial f}{\partial y} = x$$

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \qquad f(x+h) = f(x) + h\frac{df(x)}{dx}$$

- Efficient way of computing gradient (Chain rule)
- Partial derivatives and gradients

$$f(x,y) = xy \quad \rightarrow \quad \frac{\partial f}{\partial x} = y \quad \frac{\partial f}{\partial y} = x$$

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \qquad f(x+h) = f(x) + h\frac{df(x)}{dx}$$

- Example: $x = 4, y = -3 \Rightarrow f(x,y) = -12$

$$\frac{\partial f}{\partial x} = -3 \quad \frac{\partial f}{\partial x} = 4 \qquad \nabla f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right]$$

- Efficient way of computing gradient (Chain rule)
- Partial derivatives and gradients

$$f(x,y) = xy \quad \rightarrow \quad \frac{\partial f}{\partial x} = y \quad \frac{\partial f}{\partial y} = x$$

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \qquad f(x+h) = f(x) + h\frac{df(x)}{dx}$$

- Example: $x = 4, y = -3 \Rightarrow f(x,y) = -12$

$$\frac{\partial f}{\partial x} = -3 \quad \frac{\partial f}{\partial x} = 4 \qquad \nabla f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right]$$

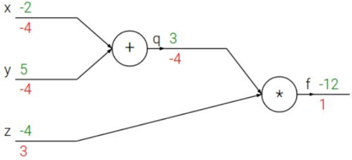
- Question: If I increase x by h, how would the output f change?

Compound expressions with graphics (example from F.F. Li)

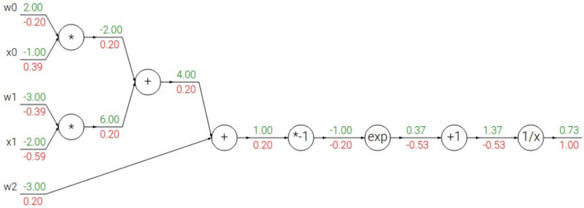
$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial x} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Chain rule: $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$

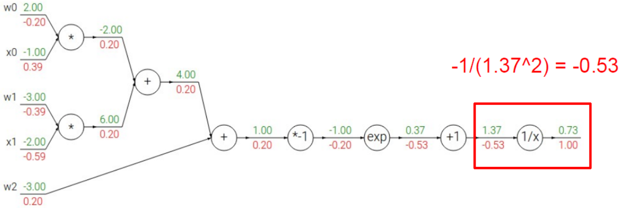


Another example: $f(w,x) = \dfrac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$



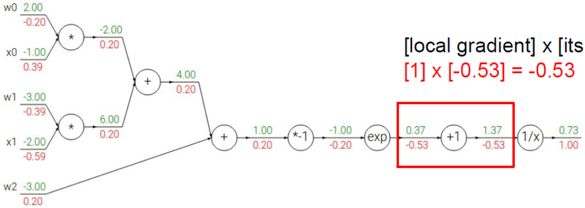
$f(x) = e^x$	$\rightarrow$	$\dfrac{df}{dx} = e^x$	$f(x) = \dfrac{1}{x}$	$\rightarrow$	$\dfrac{df}{dx} = -1/x^2$
$f_a(x) = ax$	$\rightarrow$	$\dfrac{df}{dx} = a$	$f_c(x) = c + x$	$\rightarrow$	$\dfrac{df}{dx} = 1$

Another example: $$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



$f(x) = e^x$	$\rightarrow$	$\frac{df}{dx} = e^x$	$f(x) = \frac{1}{x}$	$\rightarrow$	$\frac{df}{dx} = -1/x^2$
$f_a(x) = ax$	$\rightarrow$	$\frac{df}{dx} = a$	$f_c(x) = c + x$	$\rightarrow$	$\frac{df}{dx} = 1$

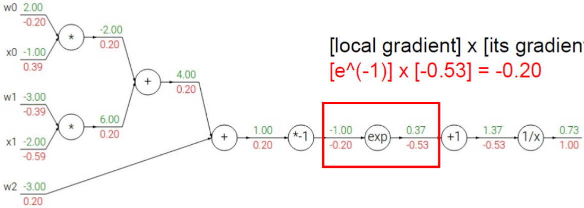
Another example: $f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$



[local gradient] x [its gradient]
[1] x [-0.53] = -0.53

$f(x) = e^x$	$\rightarrow$	$\frac{df}{dx} = e^x$	$f(x) = \frac{1}{x}$	$\rightarrow$	$\frac{df}{dx} = -1/x^2$
$f_a(x) = ax$	$\rightarrow$	$\frac{df}{dx} = a$	$f_c(x) = c + x$	$\rightarrow$	$\frac{df}{dx} = 1$

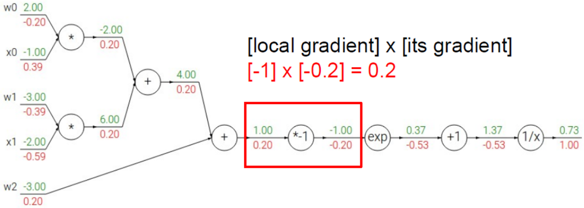
Another example: $f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$



[local gradient] x [its gradient]
[e^(-1)] x [-0.53] = -0.20

$f(x) = e^x$	$\rightarrow$	$\frac{df}{dx} = e^x$	$f(x) = \frac{1}{x}$	$\rightarrow$	$\frac{df}{dx} = -1/x^2$
$f_a(x) = ax$	$\rightarrow$	$\frac{df}{dx} = a$	$f_c(x) = c + x$	$\rightarrow$	$\frac{df}{dx} = 1$

Another example: $$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



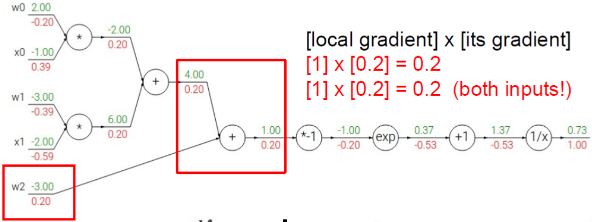
[local gradient] x [its gradient]

[-1] x [-0.2] = 0.2

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x \qquad\qquad f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a \qquad\qquad f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

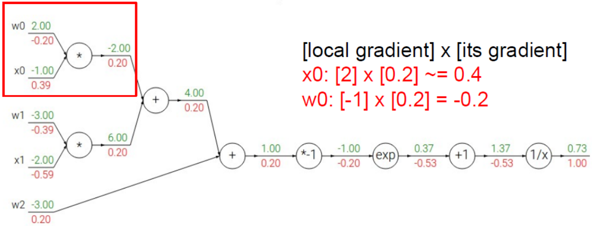
Another example: $f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$



[local gradient] x [its gradient]
[1] x [0.2] = 0.2
[1] x [0.2] = 0.2 (both inputs!)

$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$	$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$
$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$	$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$

Another example: $f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$



[local gradient] x [its gradient]

x0: [2] x [0.2] ~= 0.4

w0: [-1] x [0.2] = -0.2

$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$

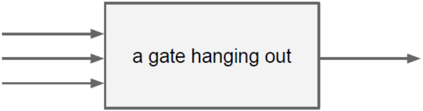
$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$

$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$

$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$

Fei-Fei Li & Andrej Karpathy Lecture 5 - 20 21 Jan 2015

Backpropagation: Key Idea



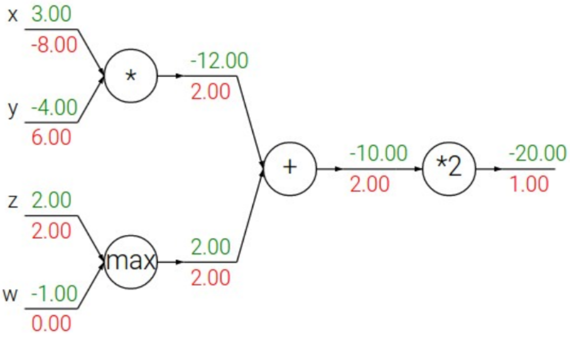
Every gate during backprop computes, for all its inputs:

[LOCAL GRADIENT] x [GATE GRADIENT]

Can be computed right away, even during forward pass

The gate receives this during backpropagation

- Add: gradient distributor
- Max: gradient router
- Mul: gradient switcher



- Gradient descent to train the network

$$\boldsymbol{w}^* = \operatorname*{argmin}_{\boldsymbol{w}} \frac{1}{N}\sum_{i=1}^{N} l(\boldsymbol{w}, \boldsymbol{x}_i, t_i) + \mathcal{R}(\boldsymbol{w})$$

- At each iteration, we need to compute

$$\boldsymbol{w}_{n+1} = \boldsymbol{w}_n - \gamma_n \nabla \mathcal{L}(\boldsymbol{w}_n)$$

- Use the backward pass to compute $\nabla \mathcal{L}(\boldsymbol{w}_n)$ efficiently
- Recall that the backward pass requires the forward pass first

- At each iteration, we need to compute

$$\boldsymbol{w}_{n+1} = \boldsymbol{w}_n - \gamma_n \nabla \mathcal{L}(\boldsymbol{w}_n)$$

with

$$\nabla \mathcal{L}(\boldsymbol{w}_n) = \frac{1}{N} \sum_{i=1}^{N} \nabla l(\boldsymbol{w}_n, \boldsymbol{x}_i, t_i) + \nabla \mathcal{R}(\boldsymbol{w}_n)$$

- Too expensive when having millions of training examples

Dealing with Big Data

- At each iteration, we need to compute

$$\boldsymbol{w}_{n+1} = \boldsymbol{w}_n - \gamma_n \nabla \mathcal{L}(\boldsymbol{w}_n)$$

with

$$\nabla \mathcal{L}(\boldsymbol{w}_n) = \frac{1}{N} \sum_{i=1}^{N} \nabla l(\boldsymbol{w}_n, \boldsymbol{x}_i, t_i) + \nabla \mathcal{R}(\boldsymbol{w}_n)$$

- Too expensive when having millions of training examples
- Instead, approximate the gradient with a mini-batch (subset of examples: 100 $\sim$ 1,000) - called stochastic gradient descent

$$\frac{1}{N} \sum_{i=1}^{N} \nabla l(\boldsymbol{w}_n, \boldsymbol{x}_i, t_i) \approx \frac{1}{|S|} \sum_{i \in S} \nabla l(\boldsymbol{w}_n, \boldsymbol{x}_i, t_i)$$

SGD with momentum

- Stochastic Gradient Descent update

$$\boldsymbol{w}_{n+1} = \boldsymbol{w}_n - \gamma_n \nabla \mathcal{L}(\boldsymbol{w}_n)$$

with

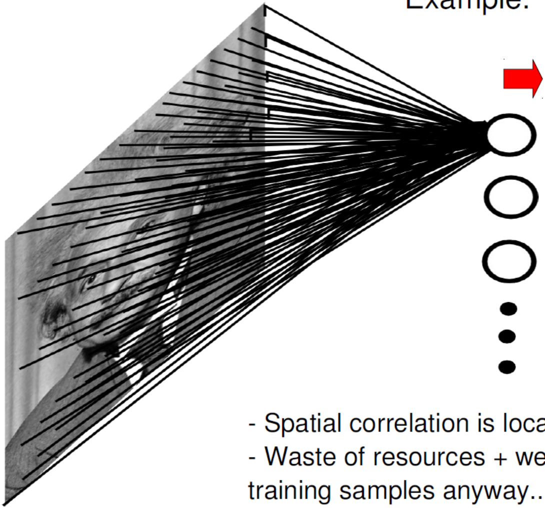
$$\nabla \mathcal{L}(\boldsymbol{w}_n) = \frac{1}{|S|} \sum_{i \in S} \nabla l(\boldsymbol{w}_n, \boldsymbol{x}_i, t_i)$$

- We can use momentum

$$\boldsymbol{w} \longleftarrow \boldsymbol{w} - \gamma \Delta$$
$$\Delta \longleftarrow \kappa \Delta + \nabla \mathcal{L}$$

- We can also decay learning rate γ as iterations goes on

Fully Connected Layer

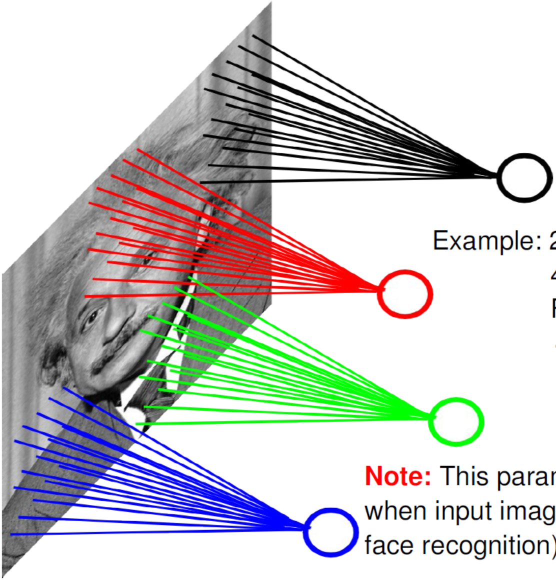


Example: 200x200 image

40K hidden units

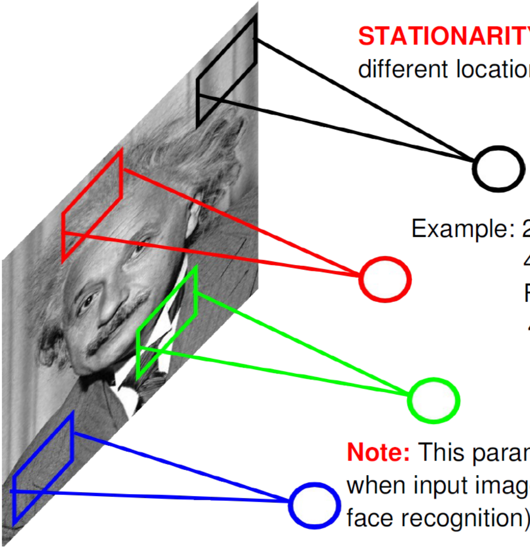
~2B parameters!!!

- Spatial correlation is local
- Waste of resources + we have not enough training samples anyway..



Example: 200x200 image
40K hidden units
Filter size: 10x10
4M parameters

Note: This parameterization is good when input image is registered (e.g., face recognition).

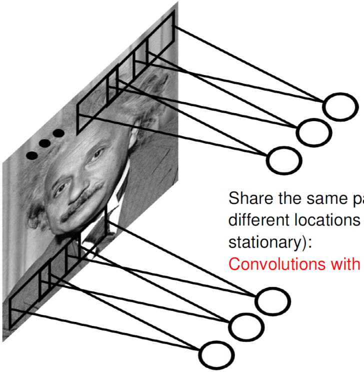


STATIONARITY? Statistics is similar at different locations

Example: 200x200 image
40K hidden units
Filter size: 10x10
4M parameters

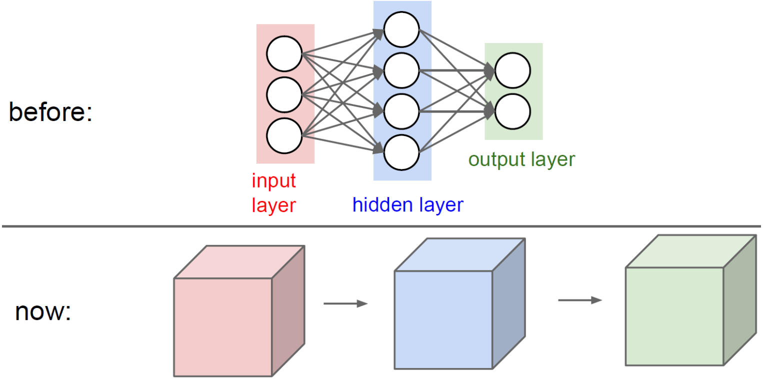
Note: This parameterization is good when input image is registered (e.g., face recognition).

- Idea: statistics are similar at different locations (Lecun 1998)
- Connect each hidden unit to a small input patch and share the weight across space
- This is called convolution layer and the network is a convolutional neural network



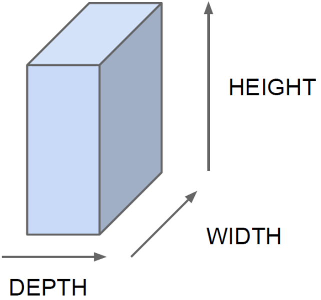
Share the same parameters across different locations (assuming input is stationary):
Convolutions with learned kernels

- Number of filters (neurons) is considered as a new dimension (depth)
  $\Rightarrow$ Volumetric representation



- Number of filters (neurons) is considered as a new dimension (depth)
  $\Rightarrow$ Volumetric representation

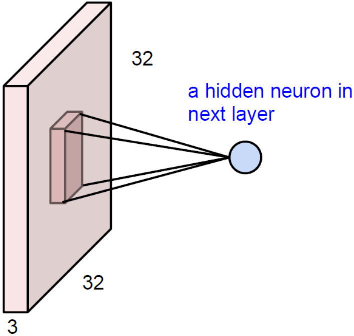
All Neural Net activations arranged in **3 dimensions:**



For example, a CIFAR-10 image is a 32x32x3 volume
32 width, 32 height, 3 depth (RGB channels)

CNNs are just neural nets BUT:

1. Local connectivity



CNNs are just neural nets BUT:

1. Local connectivity

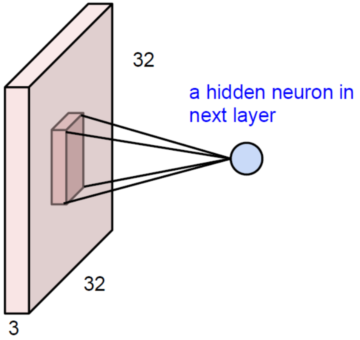


image: 32x32x3 volume

before: fully connected: 32x32x3 weights

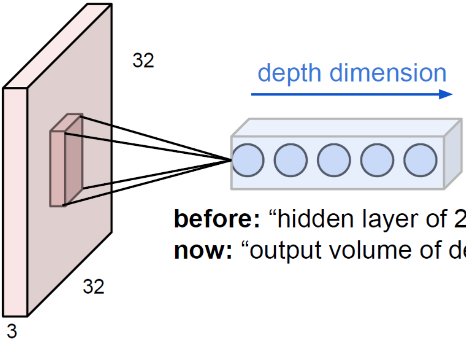
now: one neuron will connect to, e.g., 5x5x3 chunk (receptive field) and only have 5x5x3 weights

connectivity is:

- local in space (5x5 instead of 32x32)
- but full in depth (all 3 depth channels)

CNNs are just neural nets BUT:

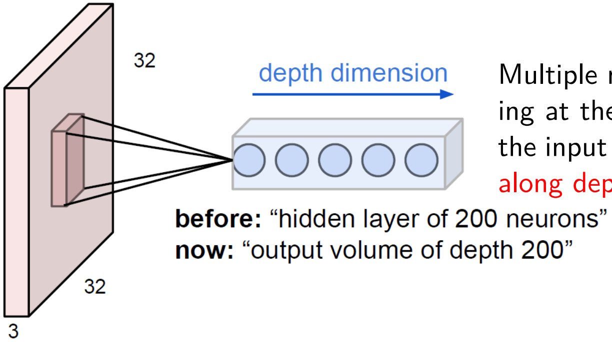
1. Local connectivity



before: "hidden layer of 200 neurons"
now: "output volume of depth 200"

CNNs are just neural nets BUT:

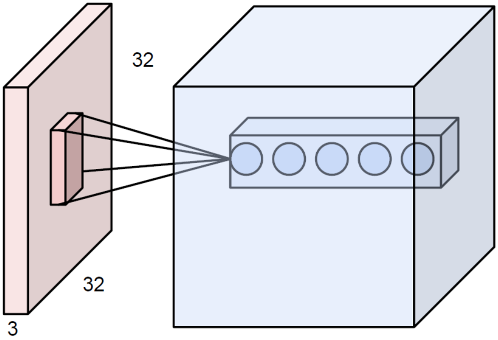
1. Local connectivity



Multiple neurons all looking at the same region of the input volume, stacked along depth.

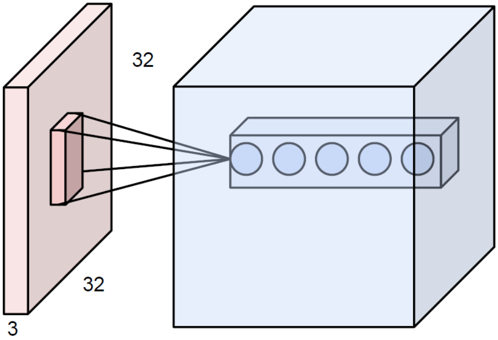
CNNs are just neural nets BUT:

2. Weight sharing



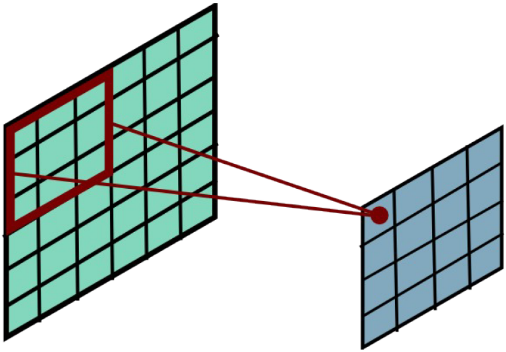
CNNs are just neural nets BUT:

2. Weight sharing

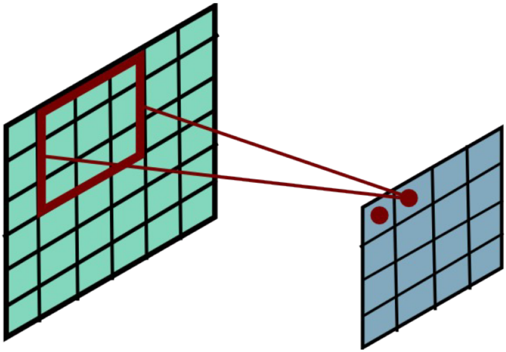


- Weights are shared across different locations
- Each depth slice is called one feature map

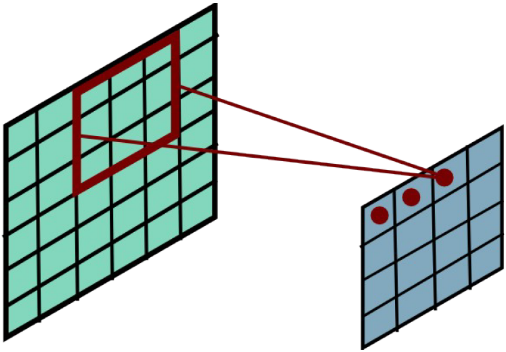
Convolutional Layer



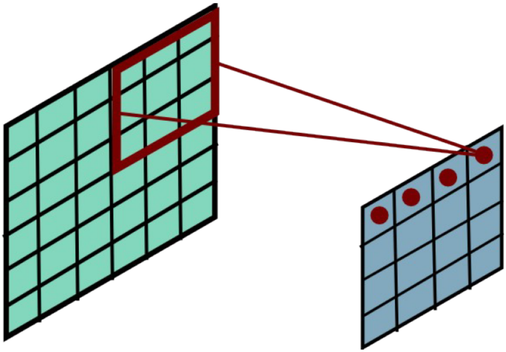
Convolutional Layer



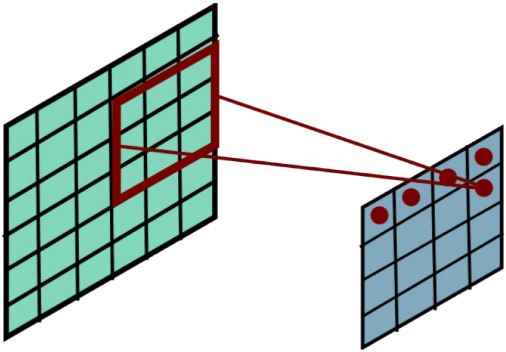
Convolutional Layer



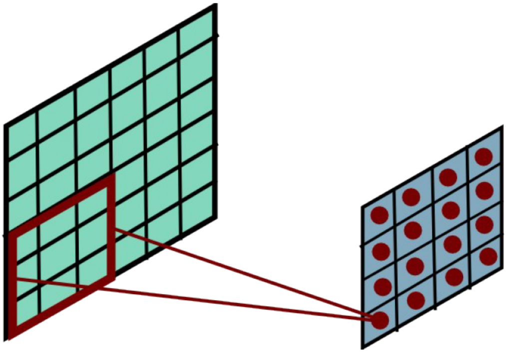
Convolutional Layer



Convolutional Layer

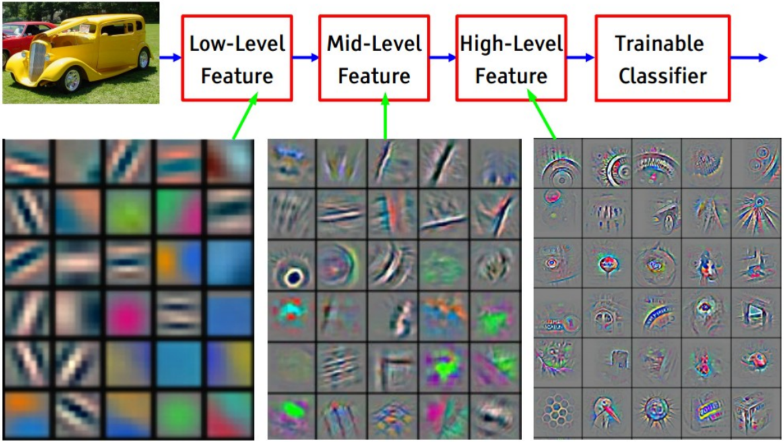


Convolutional Layer



- Input volume of size [W1 x H1 x D1]
- using K neurons with receptive fields F x F
- and applying them at strides of S gives
- Output volume: [W2, H2, D2]
  W2 = (W1-F)/S+1,
  H2 = (H1-F)/S+1,
  D2 = K

Feature (Filter) Visualization

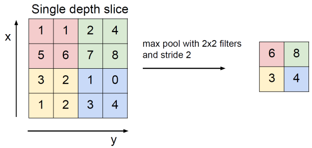


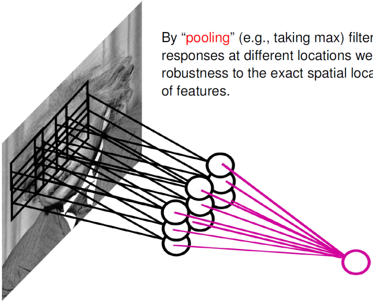
Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

Pooling

In CNNs, Conv layers are often followed by Pool layers

- Pooling layer: makes the representations smaller and more manageable without losing too much information
- Increased receptive field
- Most common: MAX pooling
- Others: average, L2 pooling $\cdots$

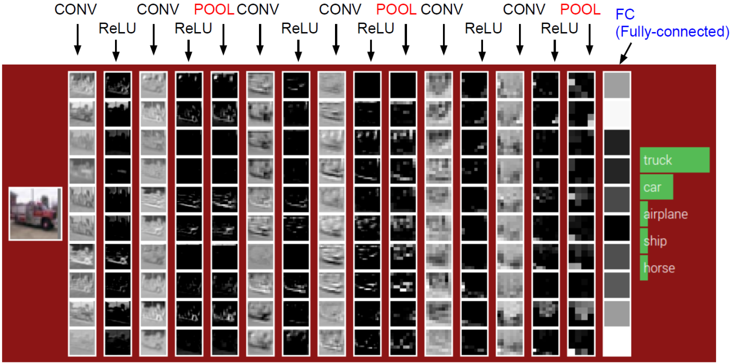


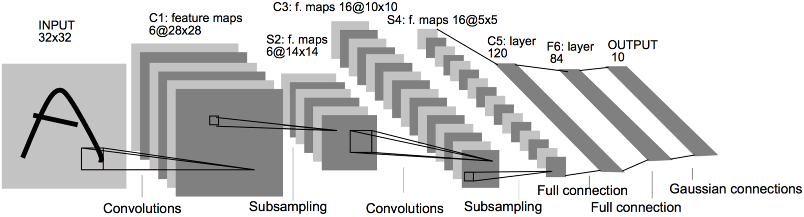


By “pooling” (e.g., taking max) filter responses at different locations we gain robustness to the exact spatial location of features.

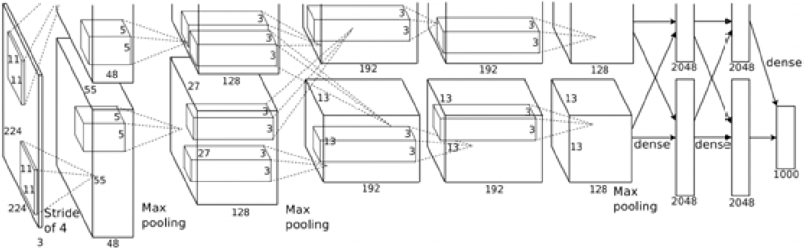
- Weight sharing (Reduce the number of parameters)
- Data augmentation (e.g., jittering, noise injection, tranformations)
- Dropout [Hinton et al.]: randomly drop units (along with their connections) from the neural network during training. Use for the fully connected layers only
- Regularization: Weight decay (L2, L1)
- Sparsity in the hidden units
- Multi-task learning
- Transfer learning

Typical ConvNet:
Image $\rightarrow$ [Conv - ReLU] $\rightarrow$ (Pool) $\rightarrow$ [Conv - ReLU] $\rightarrow$ (Pool) $\rightarrow$ FC (fully-connected) $\rightarrow$ Softmax

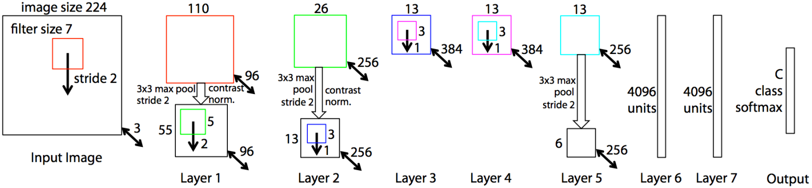




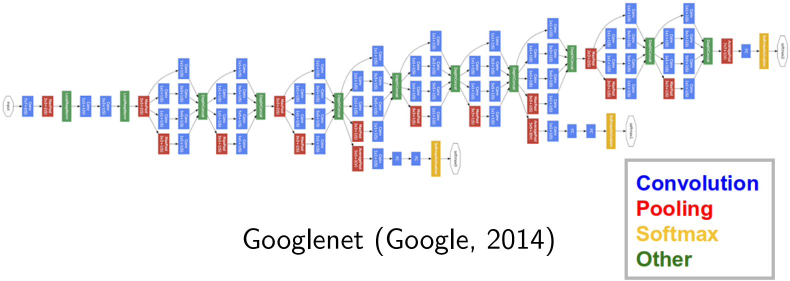
Lenet5 (Yann Lecun 1998)



Alexnet (Alex Krizhevsky et. al., 2012)

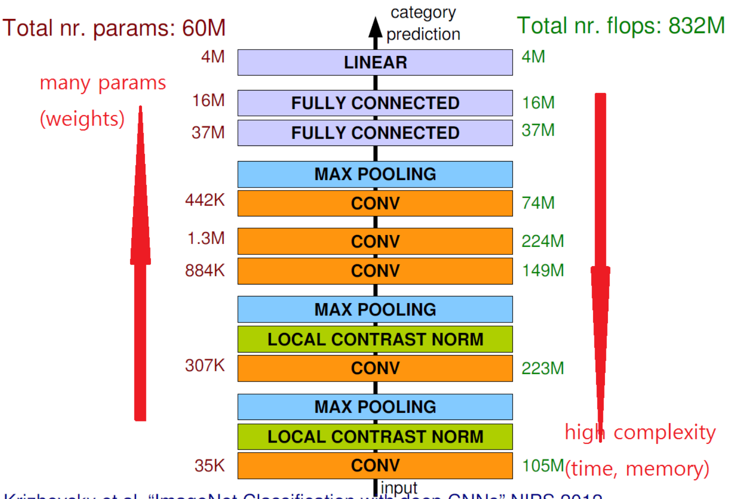


ZFnet: Clarifai (Matt Zeiler and Rob Fergus, 2013)

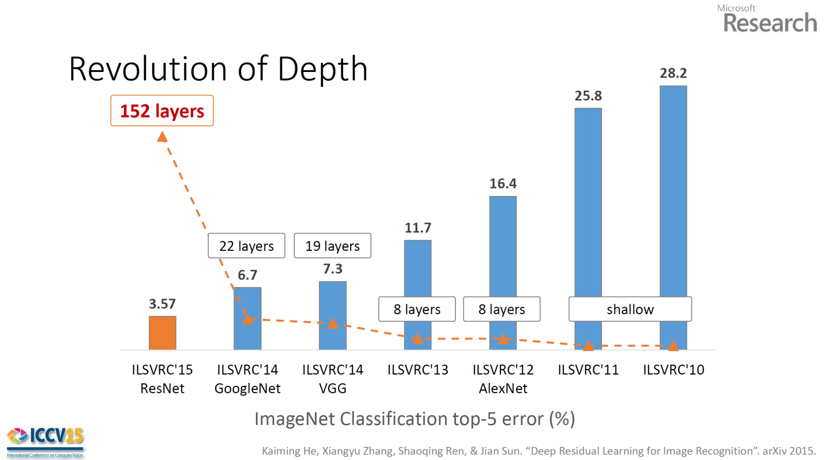


Googlenet (Google, 2014)

Complexity (Alexnet)



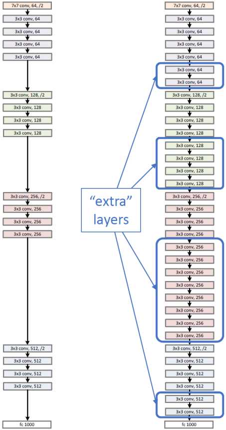
Krizhevsky et al. “ImageNet Classification with deep CNNs” NIPS 2012



- Human: 5.1% (Karpathy), Baidu cheating (2015.05) - 4.58%

Microsoft Research

a shallower model (18 layers)



a deeper counterpart (34 layers)

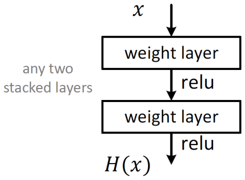
- A deeper model should not have **higher training error**
- A solution *by construction*:
  - original layers: copied from a learned shallower model
  - extra layers: set as identity
  - at least the same training error
- Optimization difficulties: solvers cannot find the solution when going deeper...

ICCV15 International Conference on Computer Vision

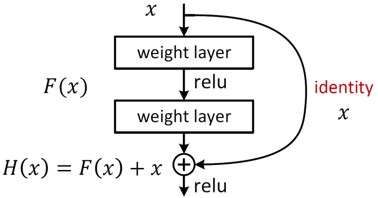
Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". arXiv 2015.

Recent Trend

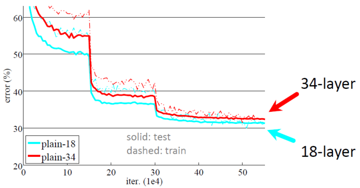
- Plaint net



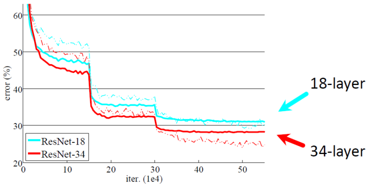
- Residual net



ImageNet plain nets



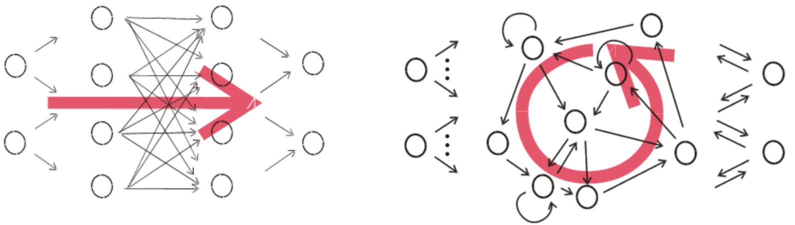
ImageNet ResNets



- Deep ResNets can be trained without difficulties
- Deeper ResNets have **lower training error**, and also lower test error

Conclusions - CNNs

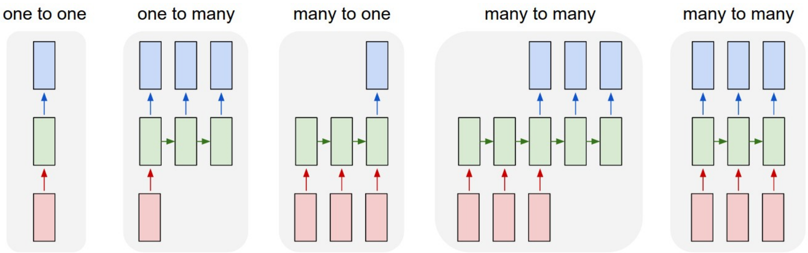
- Deep Learning = learning hierarchical models.
- ConvNets are the most successful example. Leverage large labeled datasets.
- Optimization
  - Don't we get stuck in local minima? No, they are all the same!
  - In large scale applications, local minima are even less of an issue.
- Scaling
  - GPUs
  - Distributed framework (Google)
  - Better optimization techniques
- Generalization on small datasets (curse of dimensionality):
  - data augmentation
  - weight decay
  - dropout
  - unsupervised learning
  - multi-task learning



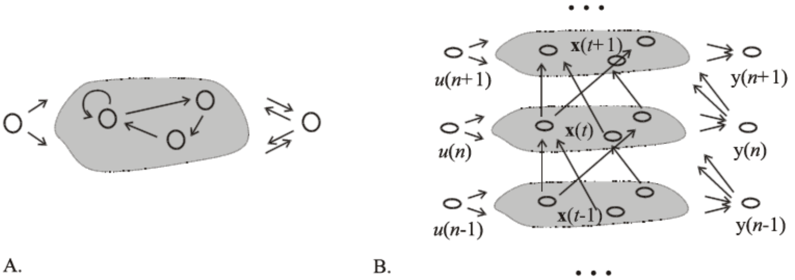
- Feedforward networks
  - Activation is fed forward from input to output through "hidden layers"
  - Static input-output mappings (functions)
  - Basic theoretical result: MLPs can approximate arbitrary (term needs some qualification) nonlinear maps with arbitrary precision ("universal approximation property")
  - Most popular supervised training algorithm: backpropagation algorithm

  - Huge literature, 95% of neural network publications concern feedforward nets
  - have proven useful in many practical applications as approximators of nonlinear functions and as pattern classifiers.
- Recurrent Neural Networks
  - All biological neural networks are recurrent
  - RNNs implement dynamical systems
  - Basic theoretical result: RNNs can approximate arbitrary (term needs some qualification) dynamical systems with arbitrary precision ("universal approximation property")
  - Several types of training algorithms are known, no clear winner
  - theoretical and practical difficulties by and large have prevented practical applications so far (?)

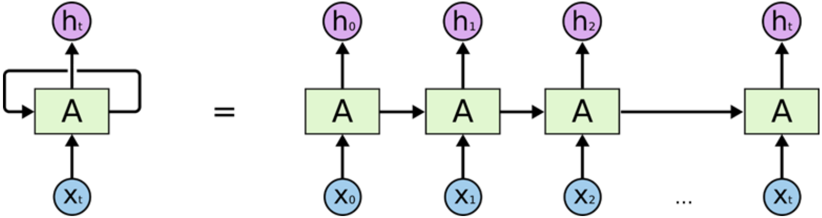
The power of RNN - flexibility



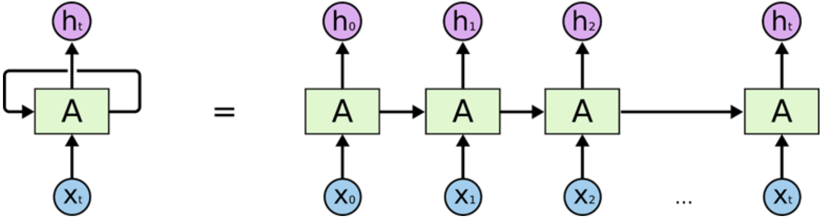
- RNN as dynamic classifiers (variable length output)



- Unfolding
- Weight sharing
- How many stacks?
- Vanishing & exploding gradients $\rightarrow$ ReLU?



- Like HMM, the model can connect previous information to the present task. (video, NLP, $\cdots$)
- Predicting the last word: "the clouds are in the ______"



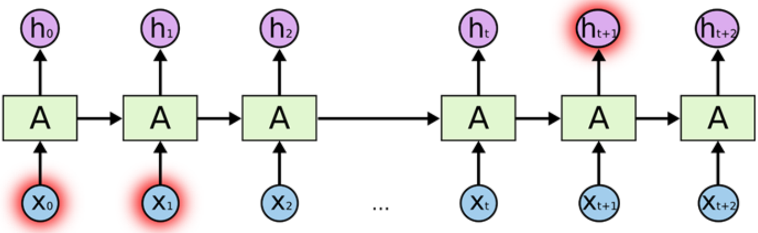
- Like HMM, the model can connect previous information to the present task. (video, NLP, $\cdots$)

- Predicting the last word: "the clouds are in the sky"

$\Rightarrow$ Nearby information is passed to predict the present word.

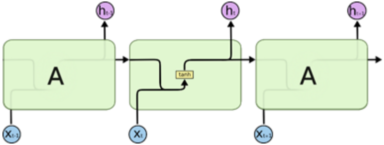
Limitations of RNNs

- Predicting the last word:
  "I grew up in France. $\cdots\cdots$, I speak fluent ______."
- Long gap between the hint and the word needing prediction.
- RNNs are unable to deliver information through the long gap.

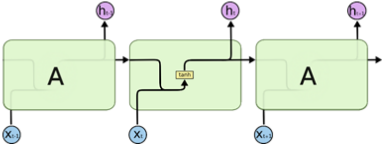


- The limitation is almost the same as that of HMM.

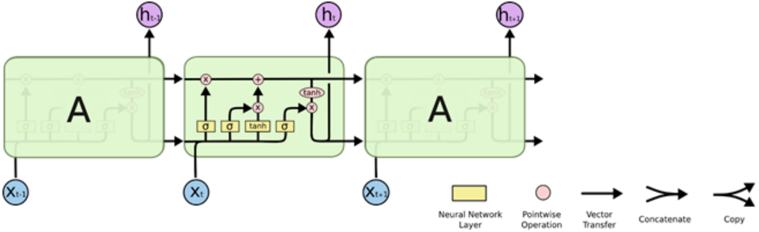
- Long Short Term Memory Networks (LSTMs): Hochreiter & Schmidhuber (1997)
- Standard RNNs - one layer of tanh



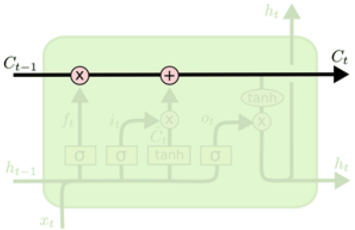
- Long Short Term Memory Networks (LSTMs): Hochreiter & Schmidhuber (1997)
- Standard RNNs - one layer of tanh

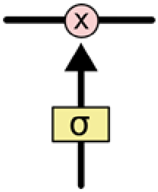


- 4 components – forget, input, output gates + information (states)



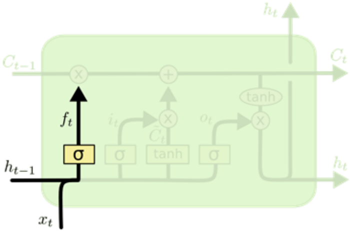
- Cell states: information





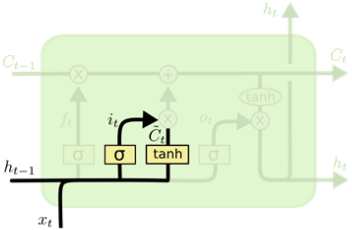
- 3 Gates: forget, input, output

- Outputs a number between 0 and 1 for each cell state C_{t-1}.
- How much information from the previous states should we keep?



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

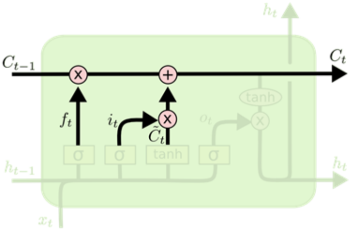
- Input gates: How much information should we update from the input?



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

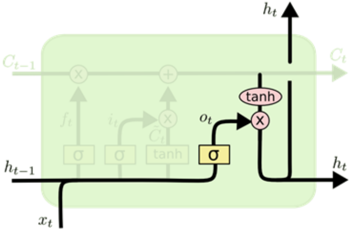
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

- Tanh layer (information): creates new additive information value $\tilde{C}_t$.



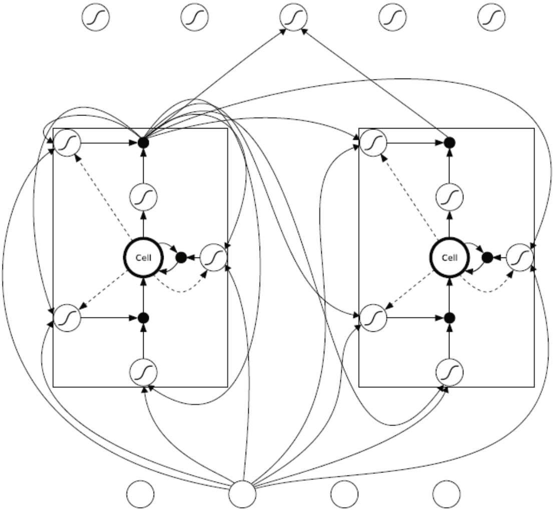
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

- Output: filtered cell states
- How much information should be output?

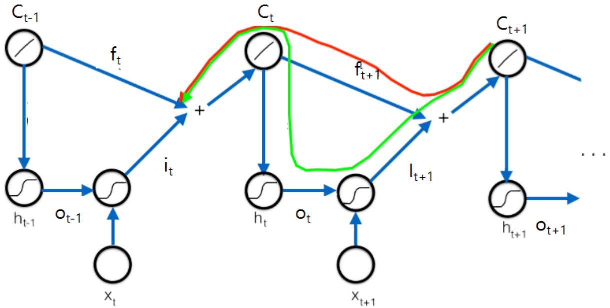


$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Putting it all together



- $\cdots >$: peephole



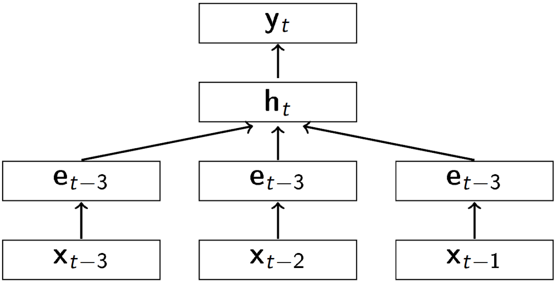
- $f = 1, i = 0$: long term dependency
- $f = 0, i = 1$: standard RNN
- red: linear (easy) path, green: nonlinear (difficult) path

- RNNs suffer from the problem of Vanishing Gradients
- The sensitivity of the network decays over time as new inputs overwrite the activations of the hidden layer, and the network forgets the first inputs.
- This problem is remedied by using LSTM blocks instead of sigmoid cells in the hidden layer.
- LSTM blocks can choose to retain their memory over arbitrary periods of time and also forget if necessary.
- Very good at finding hierarchical structure
- Can induce nonlinear oscillation (for counting and timing)
- But error flow among blocks truncated
- Difficult to train: weights into gates are sensitive

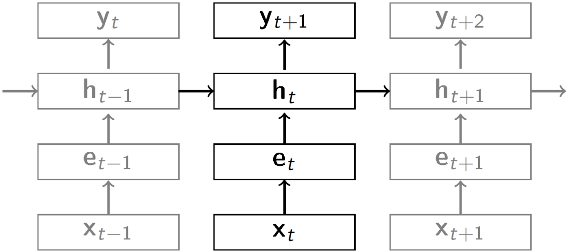
- Vocabulary, size V.
- $x_t \in \mathbb{R}^V$: true word in position t (one-hot)
- $y_t \in \mathbb{R}^V$: predicted word in position t (distribution)
- Assume all sentences are zero padded to length L.
- Model: $y_{t+1} = p(x_{t+1}|x_t, x_{t-1}, \cdots, x_1)$ for $1 \leq t < L$
- Minimize cross-entropy objective:

$$J = \sum_{t=2}^{L} H(y_t, x_t) \triangleq -\sum_{t=2}^{L} \sum_{i=1}^{V} x_{t,i} \log(y_{t,i})$$

- $\sigma(\cdot)$ is some sigmoid-like (squashing) function (e.g. logistic or tanh)
- $b\cdot$ is a bias vector, $W\cdot$ is a weight vector.



$$
\begin{aligned}
y_{t+1} &= \mathsf{softmax}(W^{yh} h_t) \\
h_t &= \sigma(W^{he}[e_{t-1}; e_{t-2}; e_{t-3}] + b^h) \\
e_t &= W^{ex} x_t
\end{aligned}
$$



$$
\begin{aligned}
y_{t+1} &= \mathrm{softmax}(W^{yh} h_t) \\
h_t &= \sigma(W^{he} e_t + W^{hh} h_{t-1} + b^h) \\
e_t &= W^{ex} x_t
\end{aligned}
$$

- Error gradients pass through nonlinearity every step

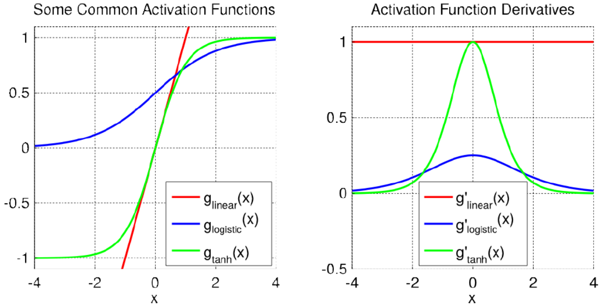
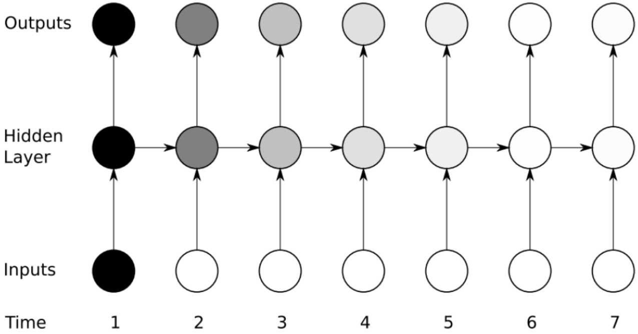


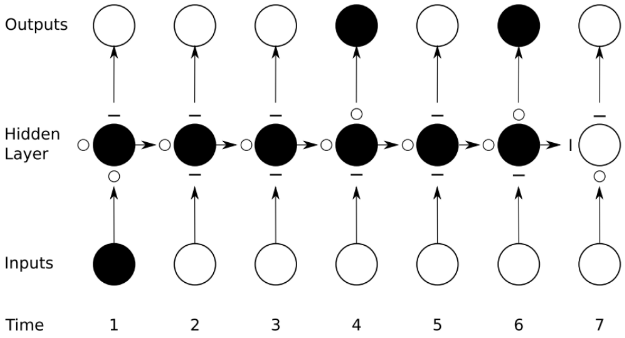
Image from https://theclevermachine.wordpress.com

- Unless weights large, error signal will degrade
$\delta_h = \sigma'(\cdot) W^{(h+1)h} \delta_{h+1}$

- Gradients may vanish or explode
- Can affect any ‘deep’ network
  - e.g. fine-tuning a non-recurrent deep NN.

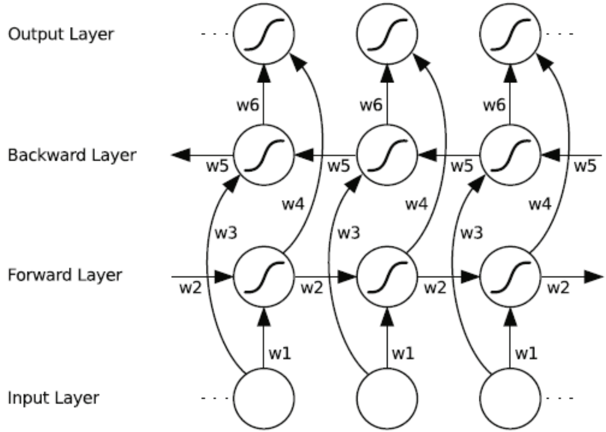


- If the input gate = 0 and forget gate = 1, gradient pass through the cell

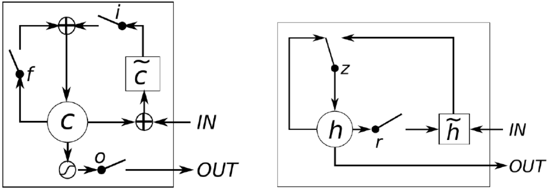


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

- Bidirectional



Chung, J.,et al. Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv'14.



(a) Long Short-Term Memory (b) Gated Recurrent Unit

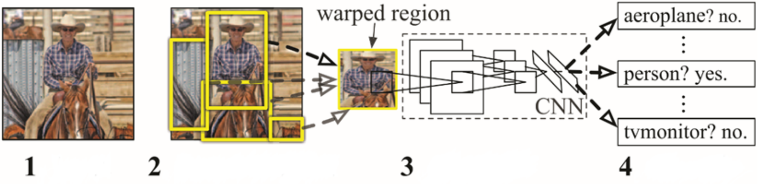
LSTM Unit:

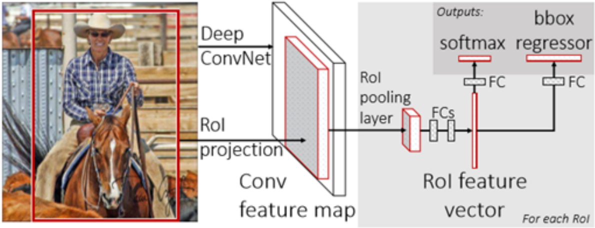
$$
\begin{aligned}
h_t^j &= o_t^j \tanh(c_t^j) \\
o_t^j &= \sigma(W_o \mathbf{x}_t + U_o \mathbf{h}_{t-1} + V_o \mathbf{c}_t)^j \\
c_t^j &= f_t^j c_{t-1}^j + i_t^j \tilde{c}_t^j \\
\tilde{c}_t^j &= \tanh(W_c \mathbf{x}_t + U_c \mathbf{h}_{t-1})^j \\
f_t^j &= \sigma(W_f \mathbf{x}_t + U_f \mathbf{h}_{t-1} + V_f \mathbf{c}_t)^j \\
i_t^j &= \sigma(W_i \mathbf{x}_t + U_i \mathbf{h}_{t-1} + V_i \mathbf{c}_t)^j
\end{aligned}
$$

GRU Unit:

$$
\begin{aligned}
h_t^j &= (1 - z_t^j) h_{t-1}^j + z_t^j \tilde{h}_t^j \\
z_t^j &= \sigma(W_z \mathbf{x}_t + U_z \mathbf{h}_{t-1})^j \\
\tilde{h}_t^j &= \tanh(W \mathbf{x}_t + U(\mathbf{r}_t \odot \mathbf{h}_{t-1}))^j \\
r_t^j &= \sigma(W_r \mathbf{x}_t + U_r \mathbf{h}_{t-1})^j
\end{aligned}
$$

- Object detection: R-CNN, fast-RCNN, faster-RCNN

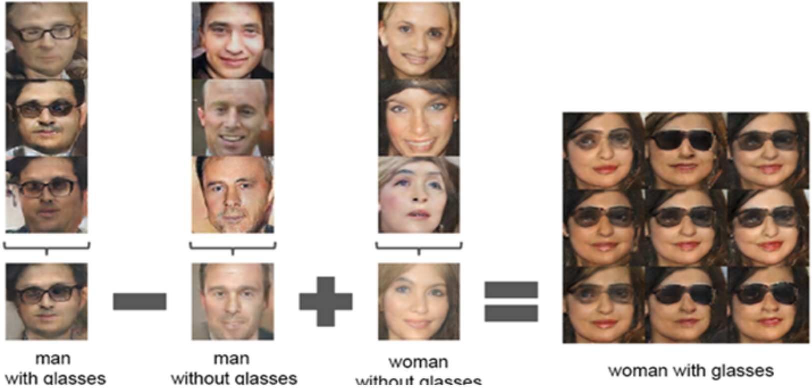




Synthetic bedroom [1]

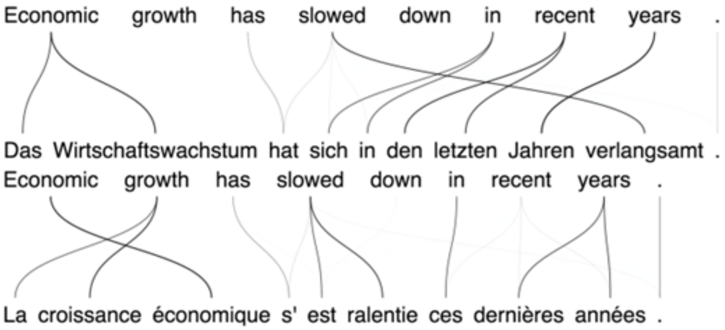
[1] Radford, Alec et al. "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," arXiv:1511.06434, 2015.

Synthetic face with glasses [2]

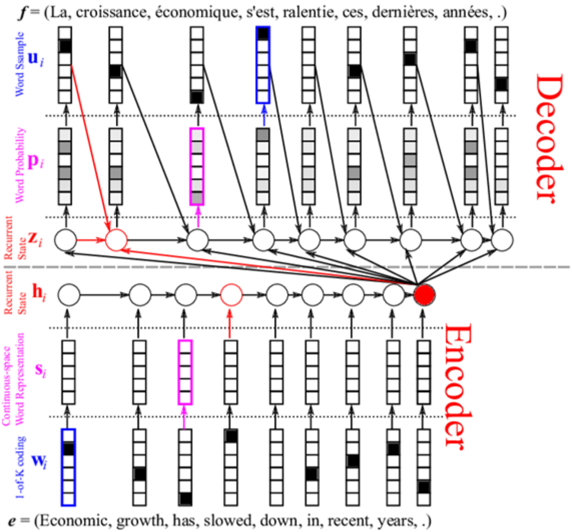


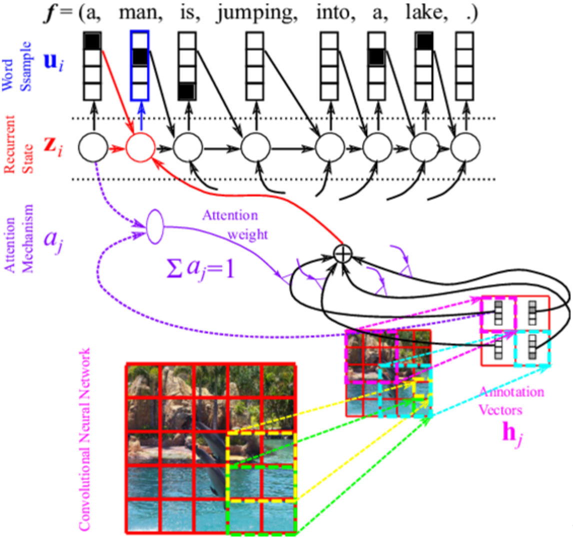
[2] Radford, Alec et al. "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," arXiv:1511.06434, 2015.

Neural Machine Translation (NMT) I



Neural Machine Translation (NMT) II





[3]

[3] "Introduction to Neural Machine Translation with GPUs,"