深度学习-背景知识与基础

史春奇

2017 年

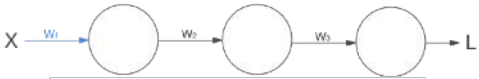
目录

第二部分（上）

- 基本网络结构的由来？
  - 卷积神经网络 CNN
  - 递归神经网络 RNN
  - 玻尔兹曼机 BM
  - 自动编码器 AE
  - 对抗网络 GAN
- 如何编写一个 CNN 网络？
  - 什么是 Filter、Stride、Padding、#Maps？
  - 经过卷积层，如何计算神经元数量？
- 如何训练好 CNN 网络
  - Xavier 初始化
  - BN 的意义
- 如何设计好深度 CNN 网络
  - Inception 模块要点和维数压缩

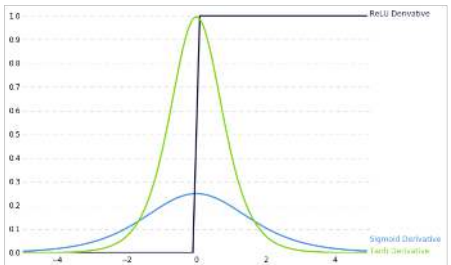
- 级联 Filter 的含义
- 并行 Ensemble 的含义
- FC 层为什么要取消

回顾反向传播 BP 算法



$$\frac{\partial Loss}{\partial W_1} = \frac{\partial Loss}{\partial f(z_3)} \cdot \frac{\partial f(z_3)}{\partial f(z_2)} \cdot \frac{\partial f(z_2)}{\partial f(z1)} \cdot \frac{\partial f(z_1)}{\partial W_1}$$

$$= \frac{\partial Loss}{\partial f(z_3)} \cdot f'(z_3) \cdot W_3 \cdot f'(z_2) \cdot W_2 \cdot f'(z_1) \cdot W_1$$



- 激活函数

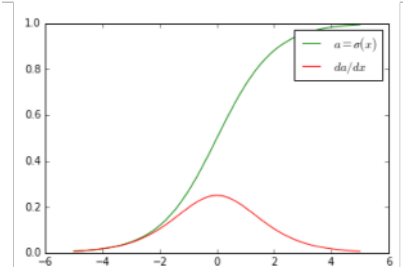
1. Sigmoid
2. Tanh
3. ReLU

- NoisyReLU
- LeakyReLU
- Parametric ReLu

- Sigmoid 和 Tanh 激活函数

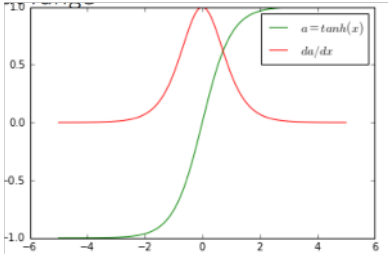
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\frac{\partial \sigma(\theta x)}{\partial \theta} = x \cdot \sigma(\theta x)(1 - \sigma(\theta x))$$



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

$$\frac{\partial a}{\partial x} = 1 - tanh^2(x)$$



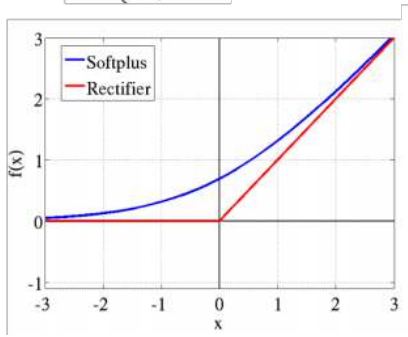
- RELU 和 SoftPlus 激活函数

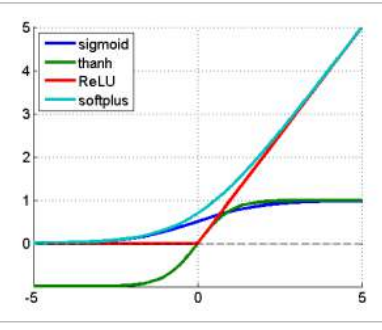
RELU: $h(x) = \max(0, x)$

$$\frac{\partial a}{\partial x} = \begin{cases} 0, if\ x \le 0 \\ 1, if x > 0 \end{cases}$$

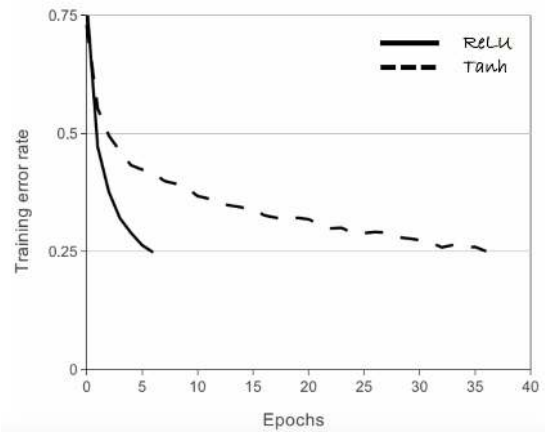
SoftPlus: $f(x) = \ln[1 + \exp(x)]$

$$f'(x) = \exp(x)/[\exp(x) + 1] = 1/[1 + \exp(-x)]$$

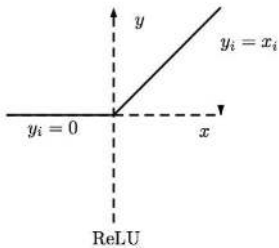




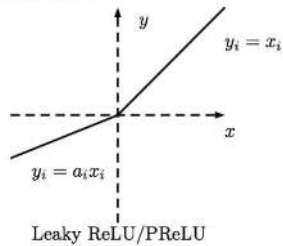
- RELU 和 Tanh 收敛速度对比



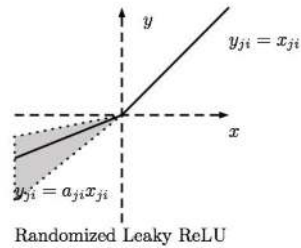
- 其他 RELU 系列
  - Noisy ReLU: $a = h(x) = \max(0, x + \varepsilon), \varepsilon \sim N(0, \sigma(x))$
  - Leaky ReLU: $a = h(x) = \begin{cases} x, if\ x > 0 \\ 0.01x\ otherwise \end{cases}$
  - Parametric ReLu: $a = h(x) = \begin{cases} x, if\ x > 0 \\ \beta x\ otherwise \end{cases}$ (parameter β is trainable)



ReLU

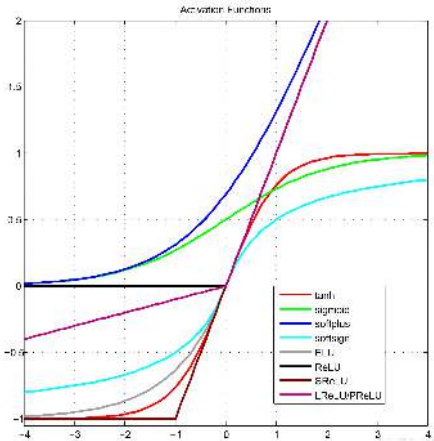


Leaky ReLU/PReLU



Randomized Leaky ReLU

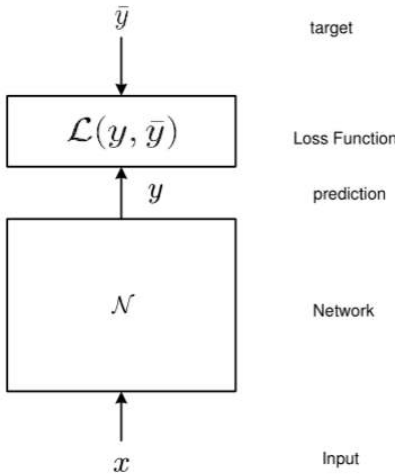
- 大部分激活函数



- 大部分激活函数表达式

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1-f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU)[2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU)[3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1+e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

- 损失函数 Loss Function

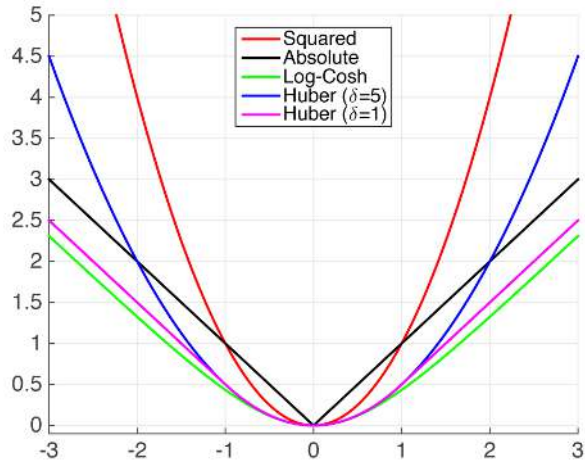


回归和分类两种损失函数

- 回归：最小平方误差 Square Error (SE)

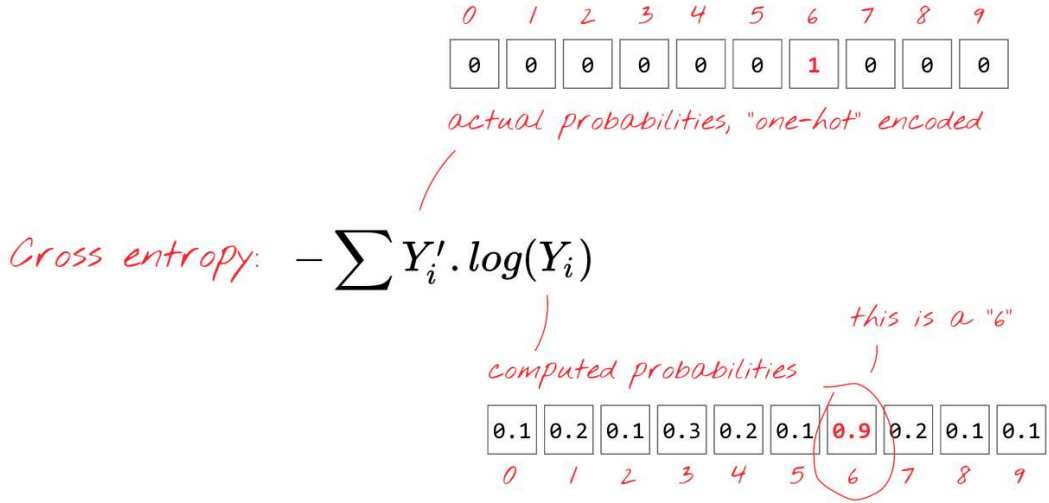
$$\mathcal{L}(y, \bar{y}) = \frac{1}{2}\sum_n (y_n - \bar{y}_n)^2 = \frac{1}{2}\sum_n \sum_k (y_{nk} - \bar{y}_{nk})^2$$

$$\frac{\partial \mathcal{L}(y, \bar{y})}{\partial y_{nk}} = (y_{nk} - \bar{y}_{nk})$$



其他回归损失函数

- 分类：Log Loss 损失 (LL)



多分类的逻辑回归的损失函数

- 分类：Log Loss 损失 (LL)

$$\mathcal{L}(\bar{y}, y) = -\sum_n \sum_k \bar{y}_{nk} \log y_{nk}$$

两类问题：

$$-\frac{1}{N}\sum_{i=1}^{N}[y_i \log p_i + (1-y_i)\log(1-p_i)].$$

多类问题简化：

$$\mathcal{L}(\bar{y}, y) = -\sum_n \log y_{n\tilde{k}_n}$$

$$\bar{y}_n = \{0, \cdots, 1, \cdots, 0\}$$

$$\bar{y}_n = \begin{cases} 1 & \text{if } k = \tilde{k}_n, \\ 0 & \text{otherwise} \end{cases}$$

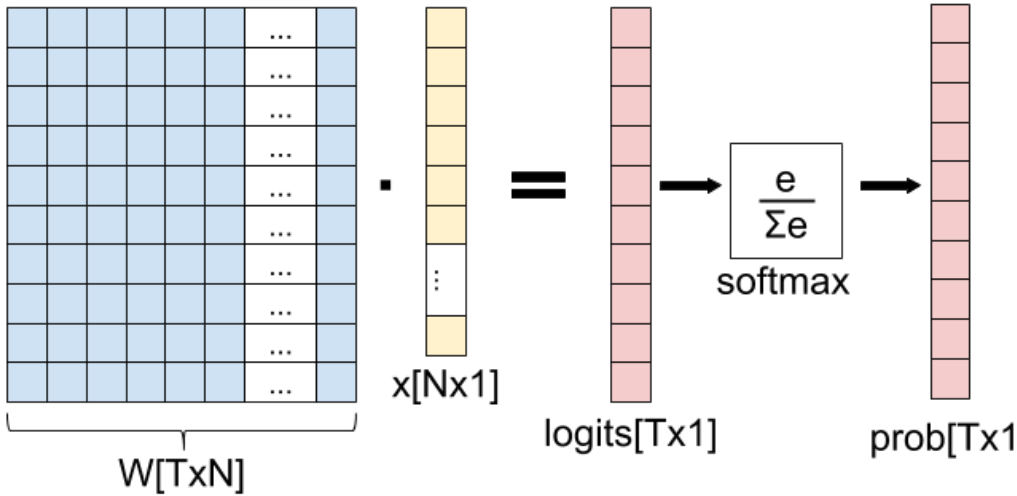
$$\mathcal{L}(\bar{y}, y) = \sum_n \mathcal{E}_n = \sum_n \Big(-\sum_i \bar{y}_{ni} \log y_{ni} \Big)$$

$$y_{ni} = \frac{e^{a_{ni}}}{\sum_j e^{a_{nj}}}$$

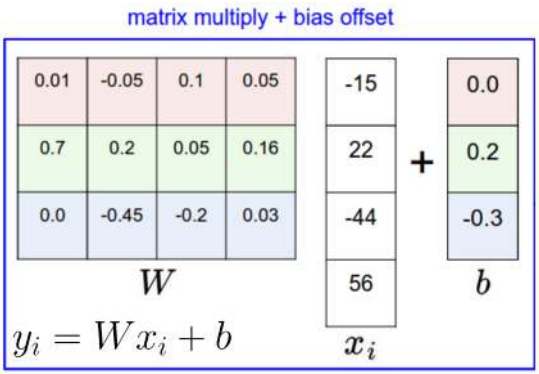
$$\begin{aligned}
\frac{\partial \mathcal{E}_n}{\partial a_{nk}} &= \frac{\partial}{\partial a_{nk}} \Big(-\sum_i \bar{y}_{ni} \log y_{ni} \Big) = -\sum_i \bar{y}_{ni} \frac{\partial \log y_{ni}}{\partial a_{nk}} \\
&= -\sum_i \bar{y}_{ni} \frac{\partial \log y_{ni}}{\partial y_{ni}} \frac{\partial y_{ni}}{\partial a_{nk}} = -\sum_i \frac{\bar{y}_{ni}}{y_{ni}} \frac{\partial y_{ni}}{\partial \log y_{ni}} \frac{\partial \log y_{ni}}{\partial a_{nk}} \\
&= -\sum_i \frac{\bar{y}_{ni}}{y_{ni}} \cdot y_{ni} \cdot (\delta_{ik} - y_{nk}) = -\sum_i \bar{y}_{ni} (\delta_{ik} - y_{nk}) \\
&= -\sum_i \bar{y}_{ni} \delta_{ik} + \sum_i \bar{y}_{ni} y_{nk} = -\bar{y}_{nk} + y_{nk} \Big(\sum_i \bar{y}_{n,i} \Big) \\
&= y_{nk} - \bar{y}_{nk}
\end{aligned}$$

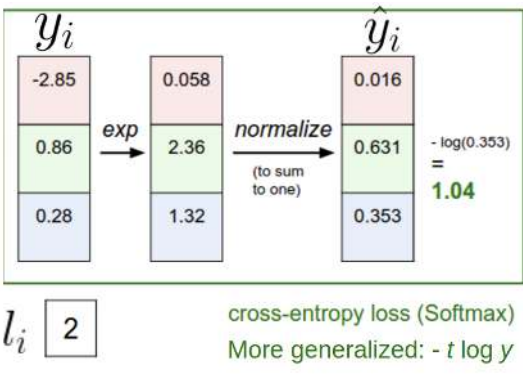
逻辑回归的损失函数

- Softmax: Logistic 的扩展



- Softmax: 归一化计算





- 分类：交互熵 Cross-Entropy (CE) [Log Loss 的等价]

Cross Entropy :

$$H(p,q) = H(p) + K(p||q)$$

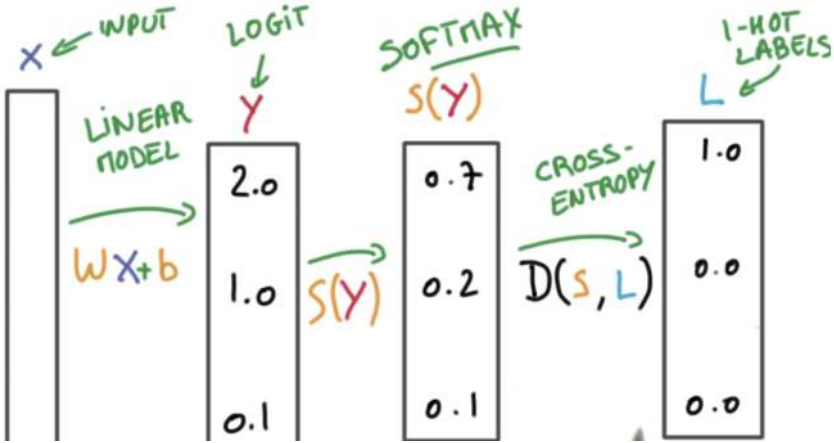
其中：

$$\begin{aligned}
K(p||q) &= \int_{(x,y)} p(x,y)\log(\frac{p(x,y)}{q(x,y)}) \\
&= \int_{(x,y)} p(x,y)\log(p(x,y)) - \int_{(x,y)} p(x,y)\log(q(x,y)) \\
&= \int_{(x,y)} p(x,y)\log(p(x,y)) - \int_{(x,y)} p(x,y)\log(q(y|x)p(x)) \\
&= \int_{(x,y)} p(x,y)\log(p(x,y)) - \int_{(x,y)} p(x,y)\log(p(x)) - \int_{(x,y)} p(x,y)\log(q(y|x)) \\
&= C - \int_{(x,y)} p(x,y)\log(q(y|x))
\end{aligned}$$

Log Loss :

$$\text{minimize}\, H(p,q) = \text{minimize}\, K(p||q) = \text{minimize} - \sum_i \log(q(y_i|x_i,\theta))$$

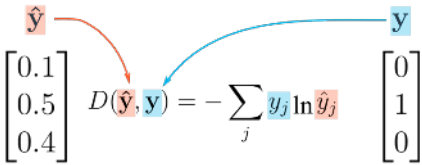
- Softmax: Logistic 的扩展



- Softmax-Log Loss: 小结

练习1： Softmax计算

练习2： Log Loss计算

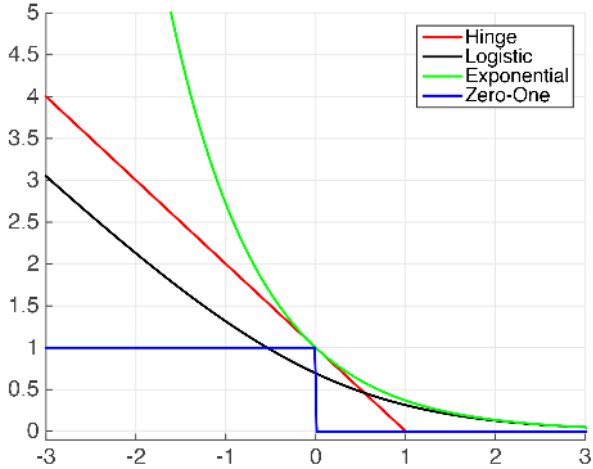


把Likelihood和Y真实值互换可以计算么？

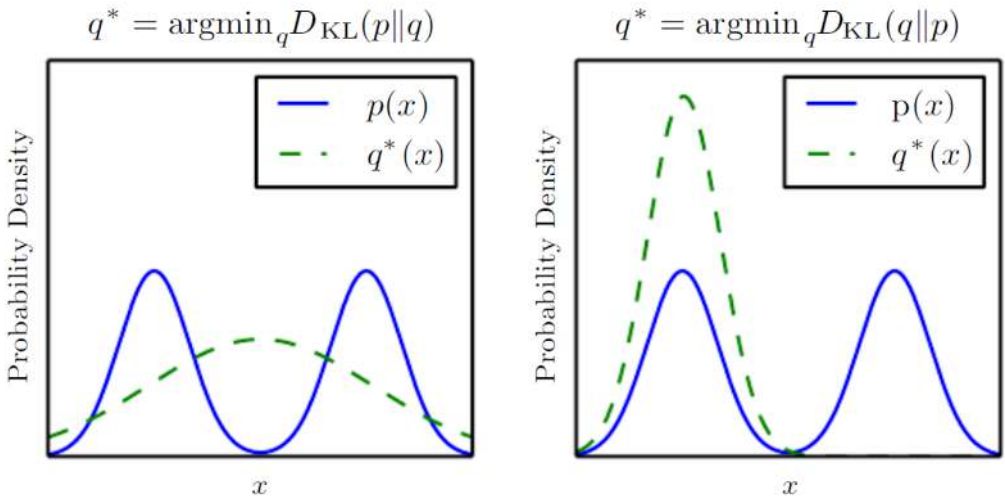
小结： 二项式分布和多项式分布：

Regression	Binomial	Multinomial (n>2)
$\hat{y}$	$\hat{y} = \frac{1}{1+e^{-\theta X}}$	$\hat{y} = \frac{e^{\theta_y X}}{\sum e^{\theta X}}$
cross entropy	$J(\theta) = -\frac{1}{n}\sum_i^n [y_i log(\hat{y}_i) + (1-y_i) log(1-\hat{y}_i)]$	$J(\theta) = -\sum_i y_i ln(\hat{y}_i)$
forward pass	$[NX1] = [NXD][DX1]$	$[NXC] = [NXD][DXC]$
response	$y = 0$ or 1	$y =$ one-hot-encoded

- 分类：其他分类损失函数

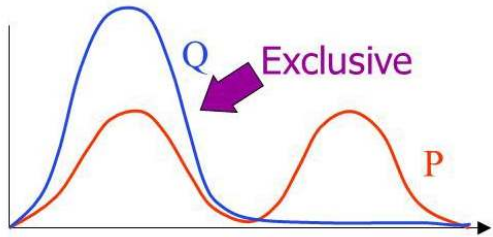

Hinge
Logistic
Exponential
Zero-One
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
-3
-2
-1
0
1
2
3

- KL 散度

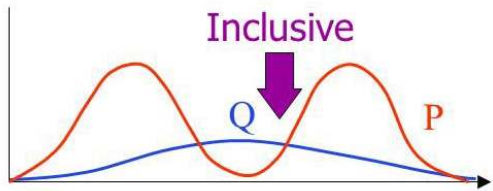

$q^* = \text{argmin}_q D_{\text{KL}}(p\|q)$
$p(x)$
$q^*(x)$
Probability Density
x
$q^* = \text{argmin}_q D_{\text{KL}}(q\|p)$
p(x)
$q^*(x)$
Probability Density
x

- KL 散度

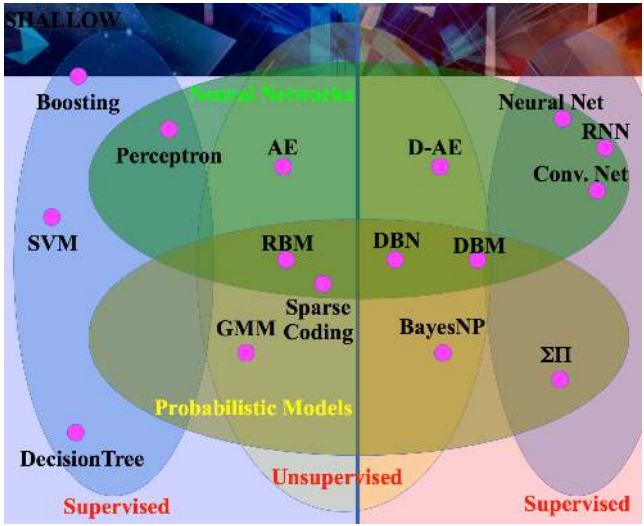
Minimising
$KL(Q||P)$
$= \sum_H Q(H) \ln \frac{Q(H)}{P(H|V)}$



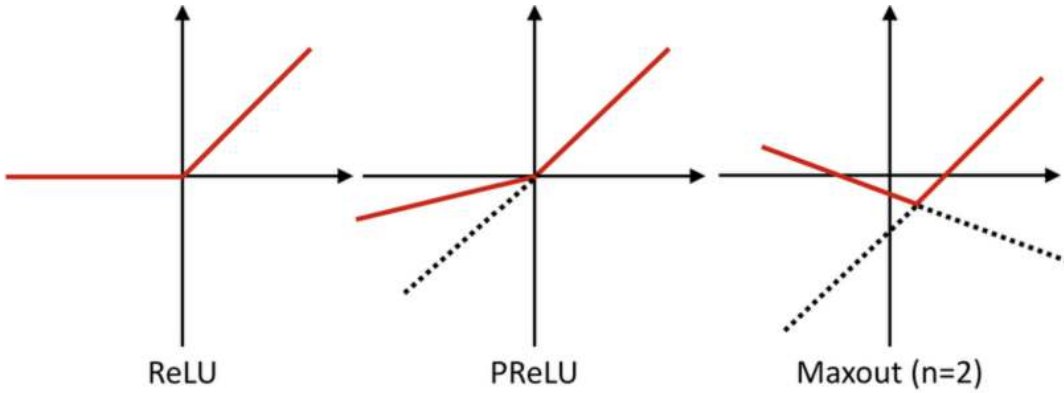
Minimising
$KL(P||Q)$
$= \sum_H P(H|V) \ln \frac{P(H|V)}{Q(H)}$



- 浅层和深层网络


SHALLOW
Boosting
Neural Networks
Perceptron
AE
D-AE
Neural Net
RNN
Conv. Net
SVM
RBM
DBN
DBM
Sparse
GMM
Coding
BayesNP
ΣΠ
Probabilistic Models
DecisionTree
Unsupervised
Supervised
Supervised

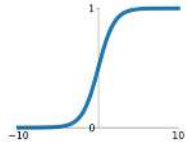
- Maxout 激活函数



- 常用激活函数

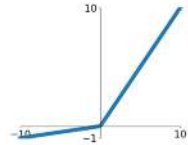
Sigmoid

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



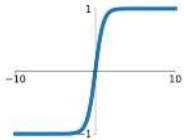
Leaky ReLU

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



tanh

$$\tanh(x)$$

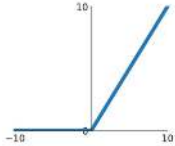


Maxout

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

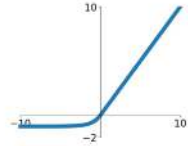
ReLU

$$\max(0, x)$$



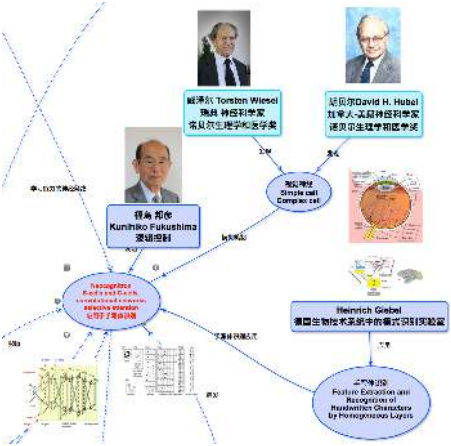
ELU

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



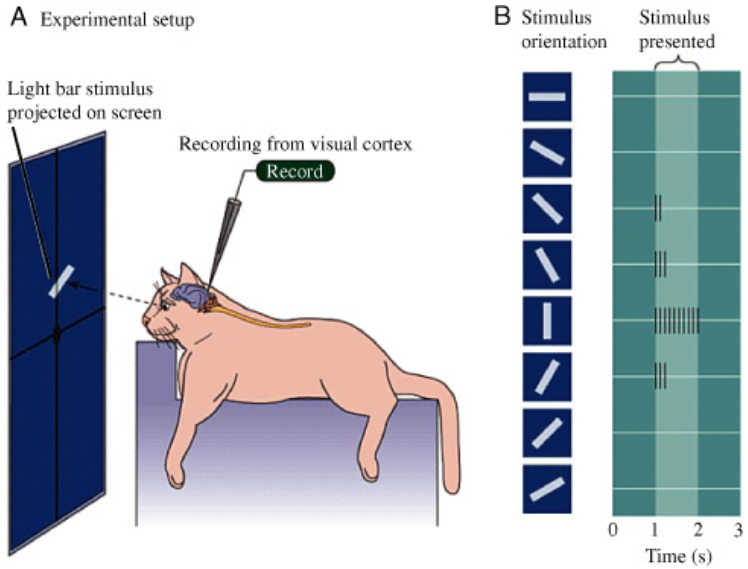
1 深度神经网络理论

1.1 卷积神经网络 CNN



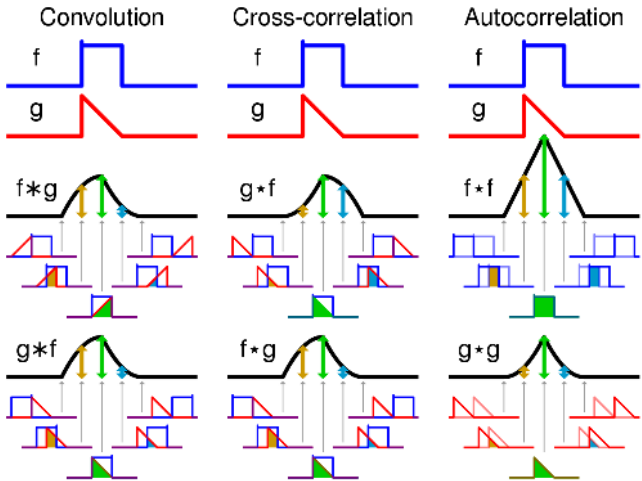
CNN 前生

- Hubel 和 Wiesel 的猫视觉皮层实验

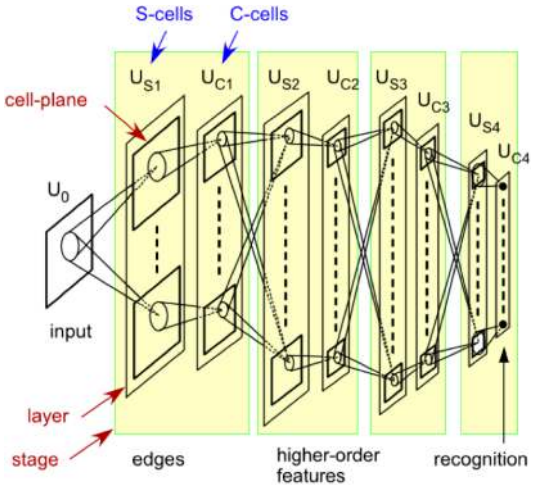


- Hubel 和 Wiesel 的猫视觉皮层实验

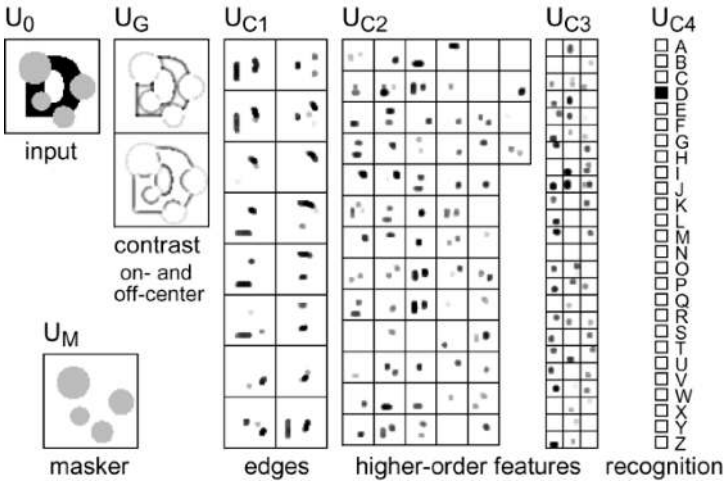
- 卷积的功能


Convolution
Cross-correlation
Autocorrelation
f
g
f∗g
g⋆f
f⋆f
g∗f
f⋆g
g⋆g

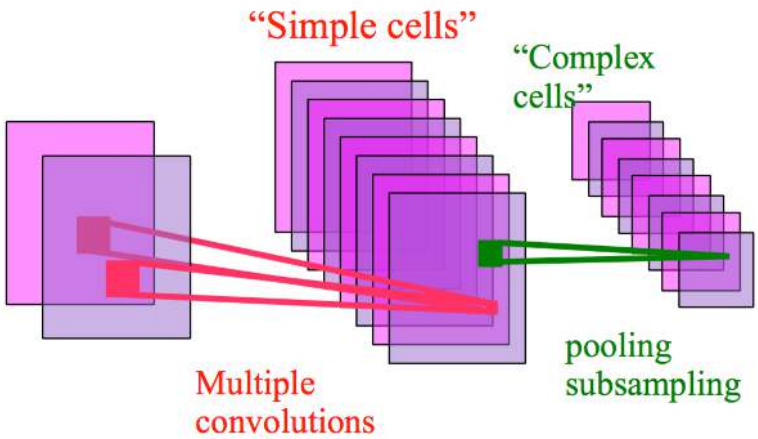
- 福島邦彦的 Neocognitron



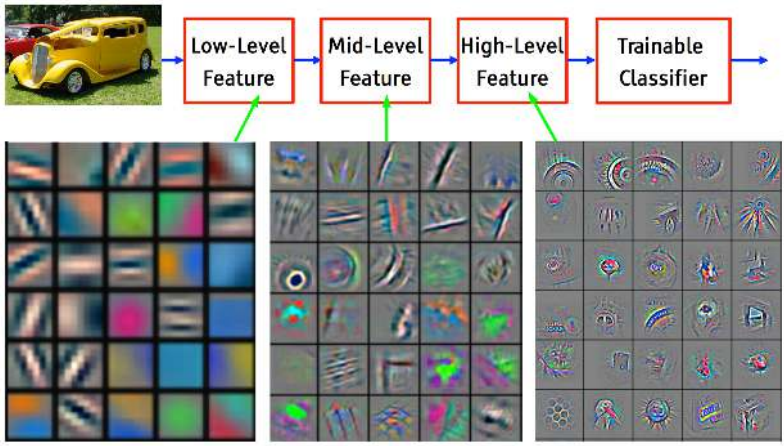
- Neocognitron 的手写体识别



- 卷积层和池化层的解读

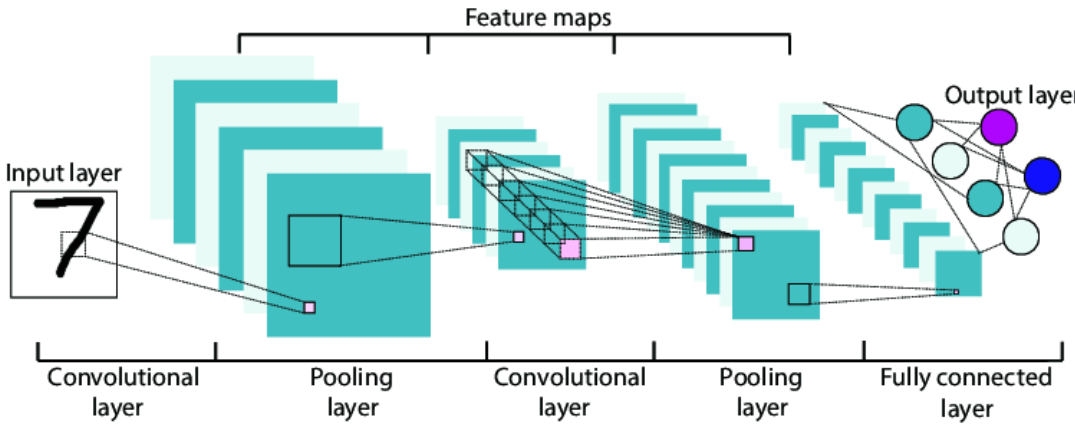

“Simple cells”
“Complex cells”
Multiple convolutions
pooling subsampling

- CNN 的解读



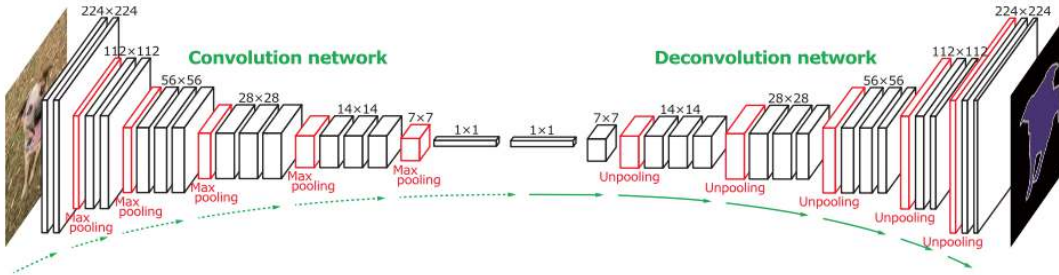
Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

- LeNet



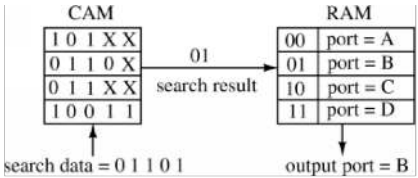
应了BP 的卷积网络!

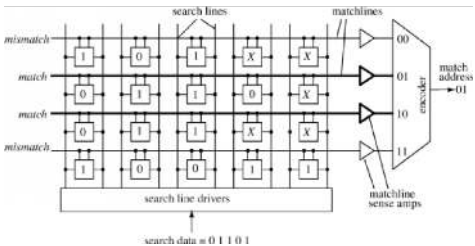
- Deconv


224×224
112×112
56×56
28×28
14×14
7×7
1×1
1×1
7×7
14×14
28×28
56×56
112×112
224×224
Convolution network
Deconvolution network
Max pooling
Max pooling
Max pooling
Max pooling
Max pooling
Unpooling
Unpooling
Unpooling
Unpooling
Unpooling

1.2 递归神经网络 RNN

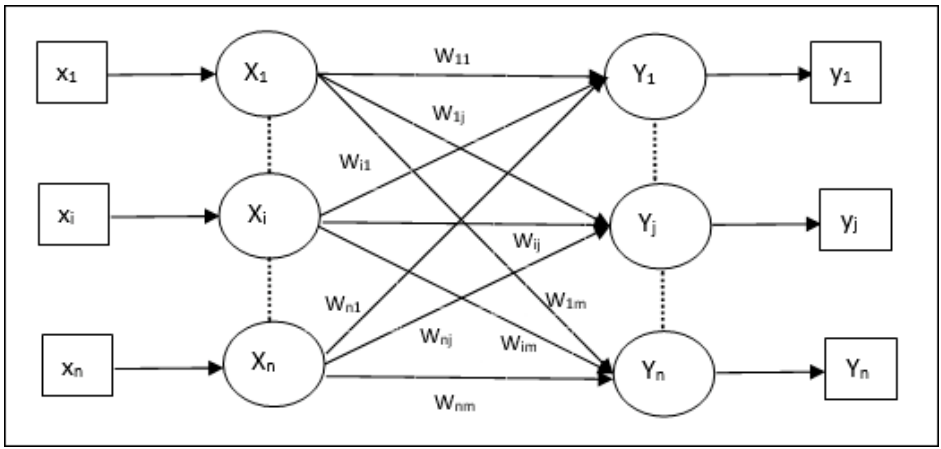
- 内容定址存储器 content-addressable memories (CAM)



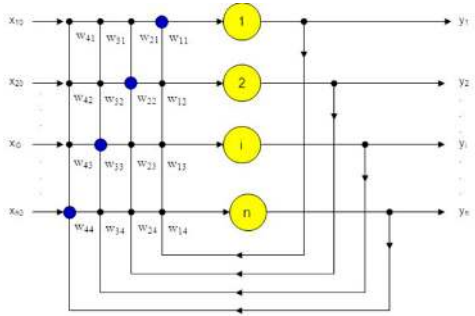


Associative Memory !

- 联想记忆神经网络 Associative Memory



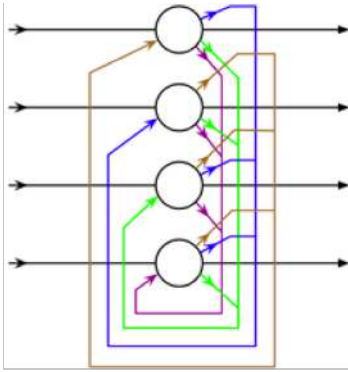
- 霍普菲尔德网络 Hopfield Network



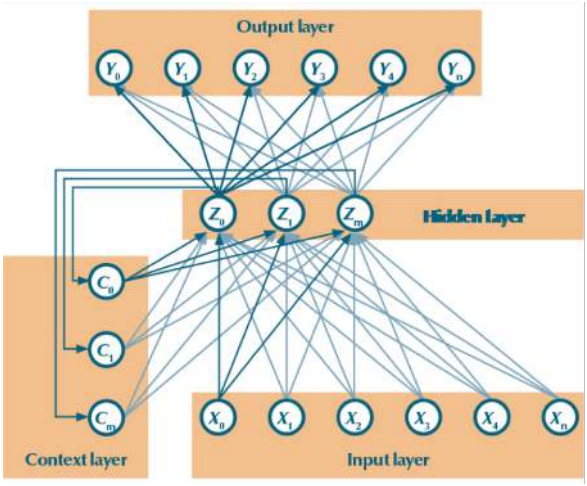
$$y_j(t) = \mathrm{sgn}(x_j(t)),\ \mathrm{sgn} = \begin{cases} +1 & \text{if } x > 0 \\ -1 & \text{if } x < 0 \end{cases}$$

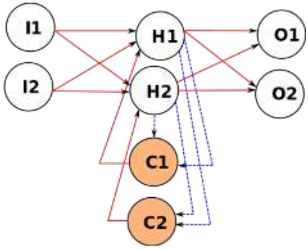
$$x_j(t) = \sum_{i=1}^{n} w_{ij} y_i(t-1)$$

It is a dynamic system:
$x(0) \rightarrow y(0) \rightarrow x(1) \rightarrow y(1) \ldots \rightarrow y^*$



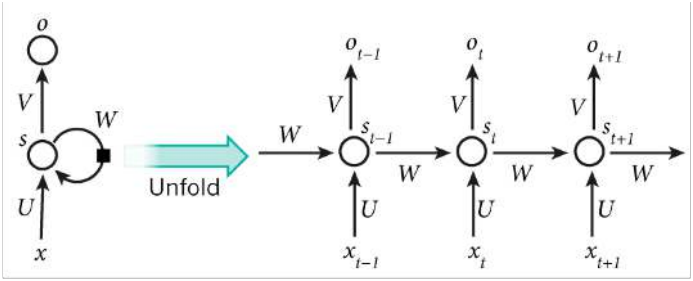
- Elman 网络 Simple Recurrent Network (SRN)

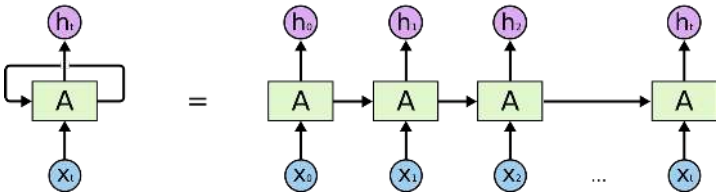




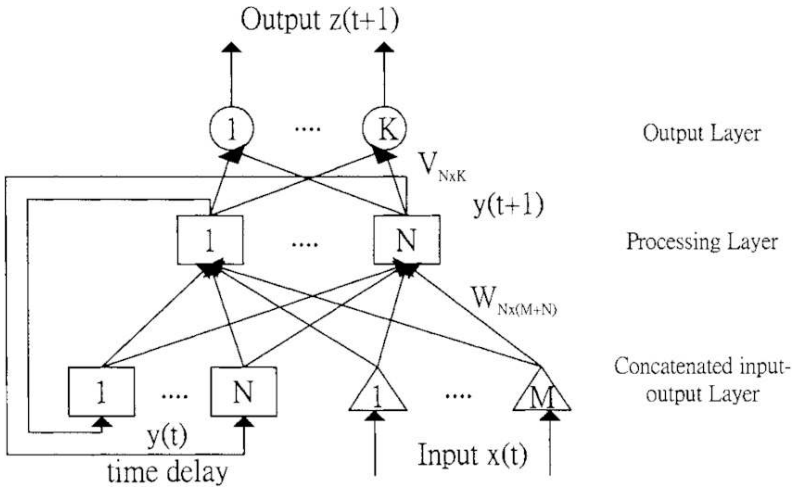
- SRN 训练
  1. back-propagation through time (BPTT)
  2. real-time recurrent learning (RTRL)
  3. extended Kalman filtering (EKF)

- BPTT

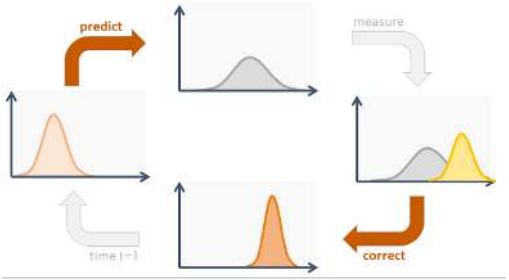




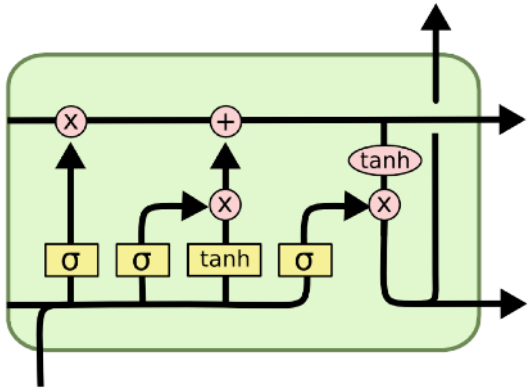
- RTRL


Output z(t+1)
1
....
K
V NxK
y(t+1)
1
....
N
W Nx(M+N)
1
....
N
1
....
M
y(t)
time delay
Input x(t)
Output Layer
Processing Layer
Concatenated input-output Layer

- EKF

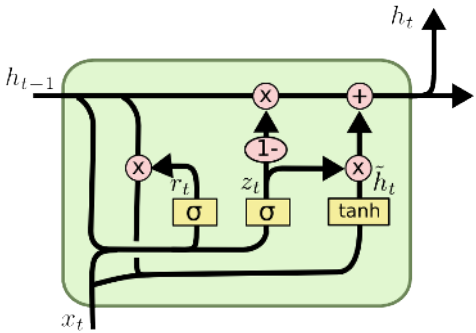

predict
measure
correct
time t+1
Prior knowledge of state
Prediction step
Based on e.g. physical model
Next timestep
Update step
Compare prediction to measurements
Measurements
Output estimate of state

- LSTM



$$
\begin{aligned}
f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \\
h_t &= o_t \circ \sigma_h(c_t)
\end{aligned}
$$

- GRU



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

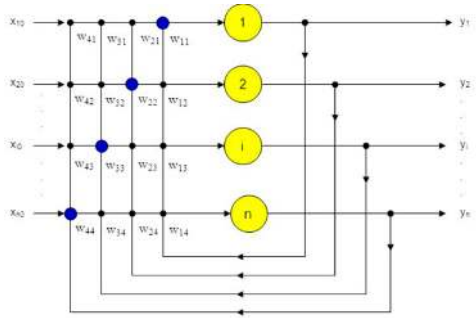
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

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

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

1.3 玻尔兹曼机 BM

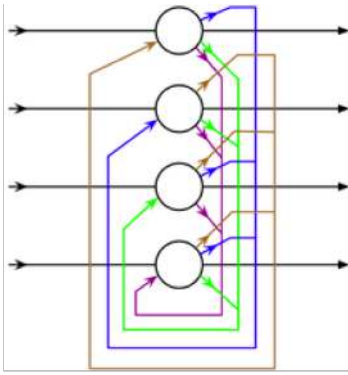
- 霍普菲尔德网络 Hopfield Network



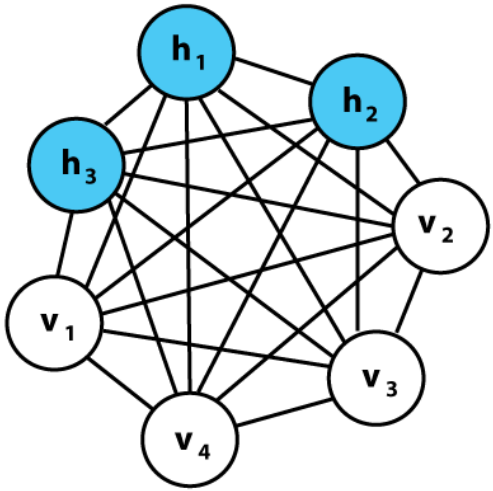
$$y_j(t) = \mathrm{sgn}(x_j(t)),\ \mathrm{sgn} = \begin{cases} +1 & \text{if } x > 0 \\ -1 & \text{if } x < 0 \end{cases}$$

$$x_j(t) = \sum_{i=1}^{n} w_{ij} y_i(t-1)$$

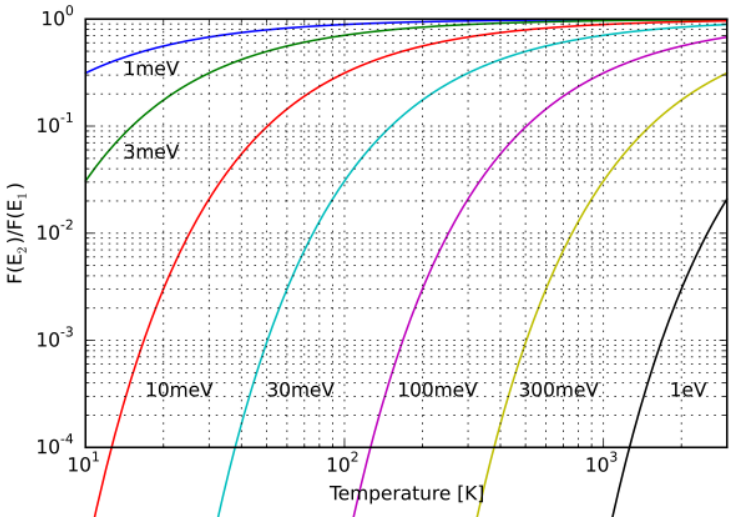
It is a dynamic system:
$x(0) \rightarrow y(0) \rightarrow x(1) \rightarrow y(1) \ldots \rightarrow y^*$



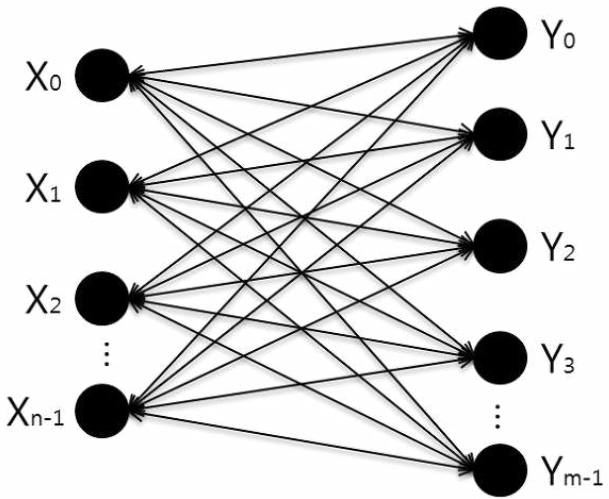
- 玻尔兹曼机 Boltzmann Machine



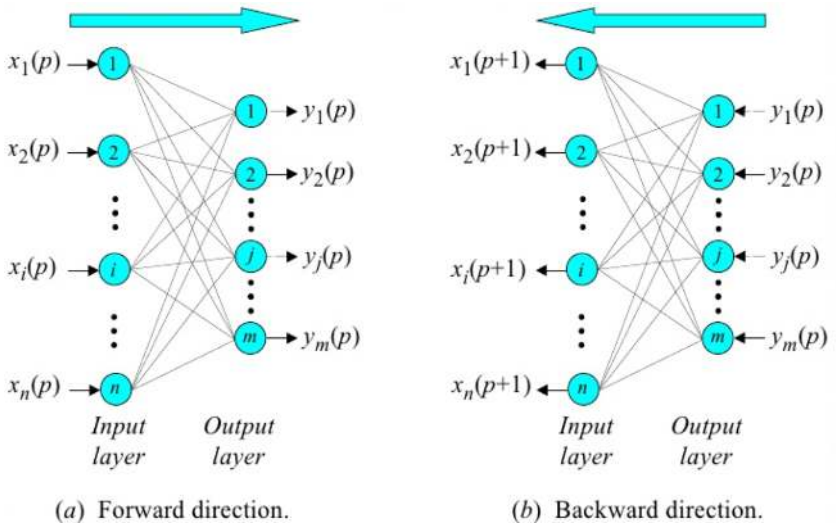
- 玻尔兹曼分布


10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
$F(E_2)/F(E_1)$
1meV
3meV
10meV
30meV
100meV
300meV
1eV
10^1
10^2
10^3
Temperature [K]

- Bidirectional Associative Memory (BAM)



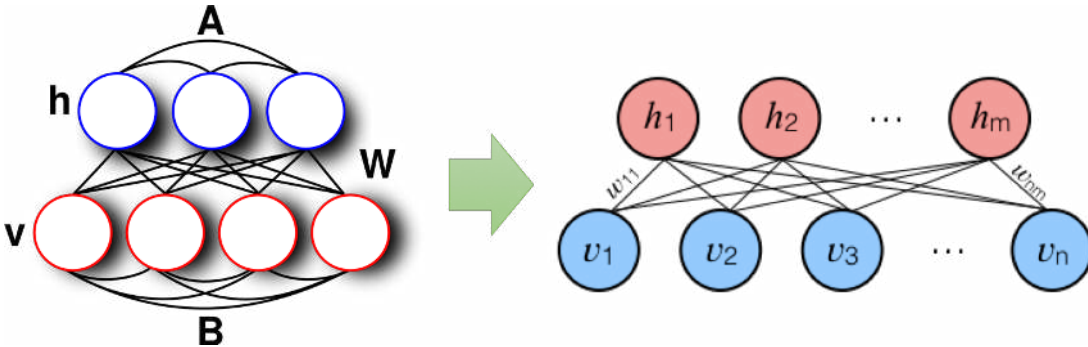
- Bidirectional Associative Memory (BAM)



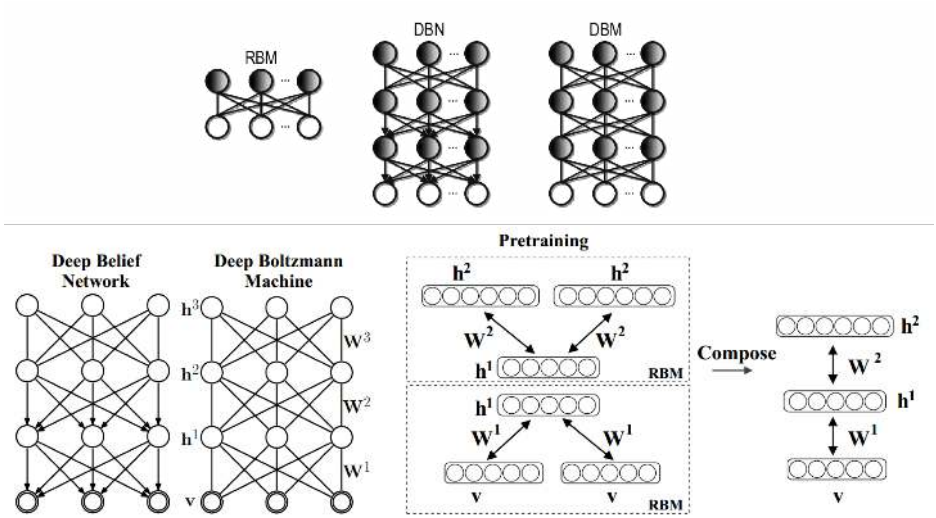
(*a*) Forward direction.

(*b*) Backward direction.

- 受限玻尔兹曼机 Restricted Boltzmann Machine

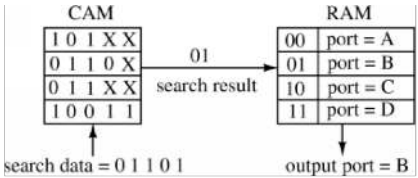


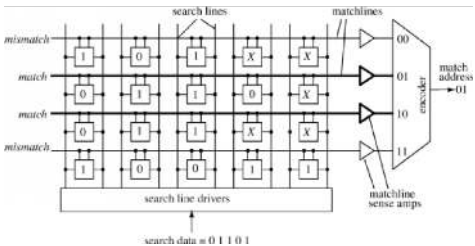
- 深度玻尔兹曼机 Deep Boltzmann Machine



1.4 自动编码器 AE

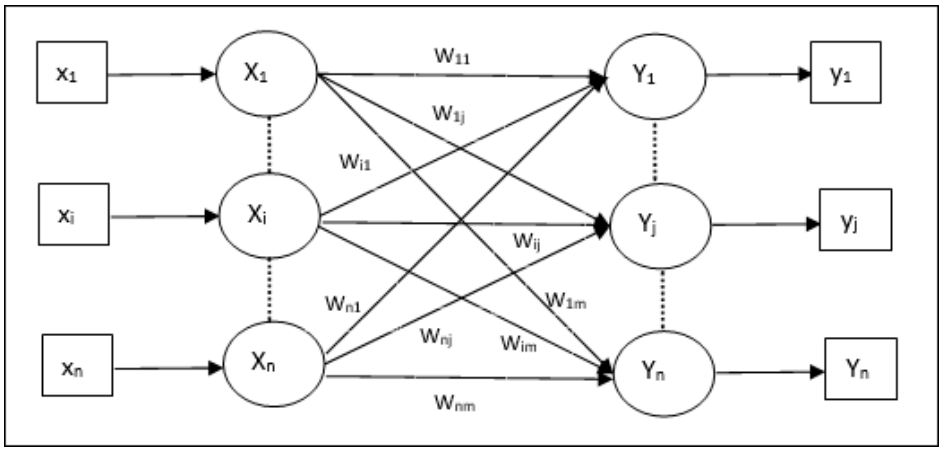
- 内容定址存储器 content-addressable memories (CAM)



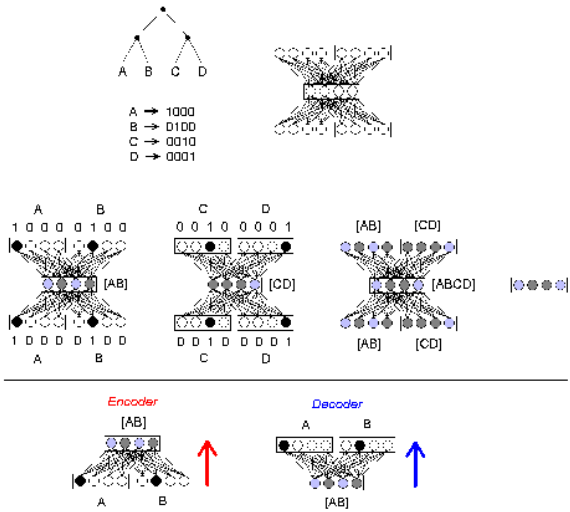


Associative Memory !

- 联想记忆神经网络 Associative Memory



- 递归自联想记忆网络 Recursive AutoAssociative Memory (RAAM)



- 递归自联想记忆网络编解码器

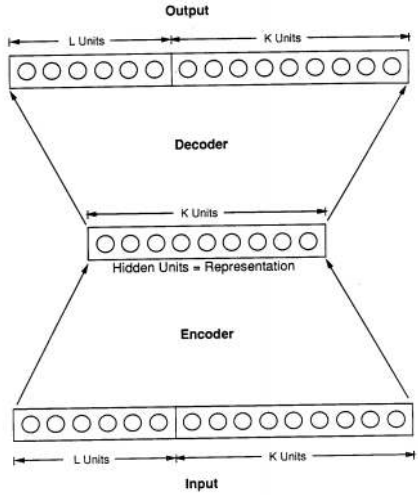
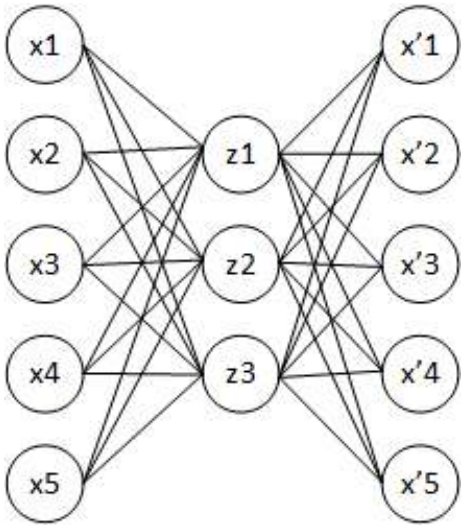
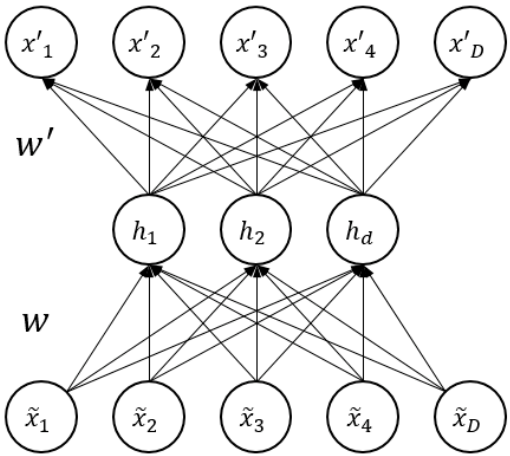


Figure 1: The Sequential RAAM Structure.

- 自动编码机 AutoEncoder

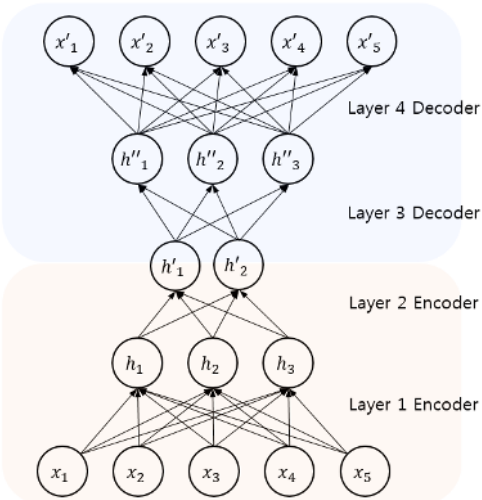


- 去噪自动编码机 Denoising AutoEncoder

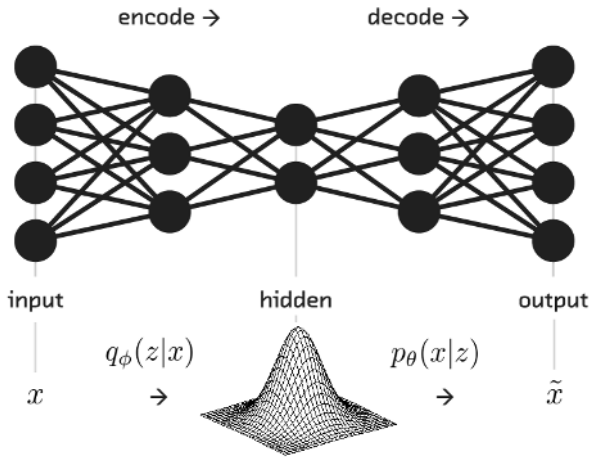


$\tilde{x}$ = x + noise

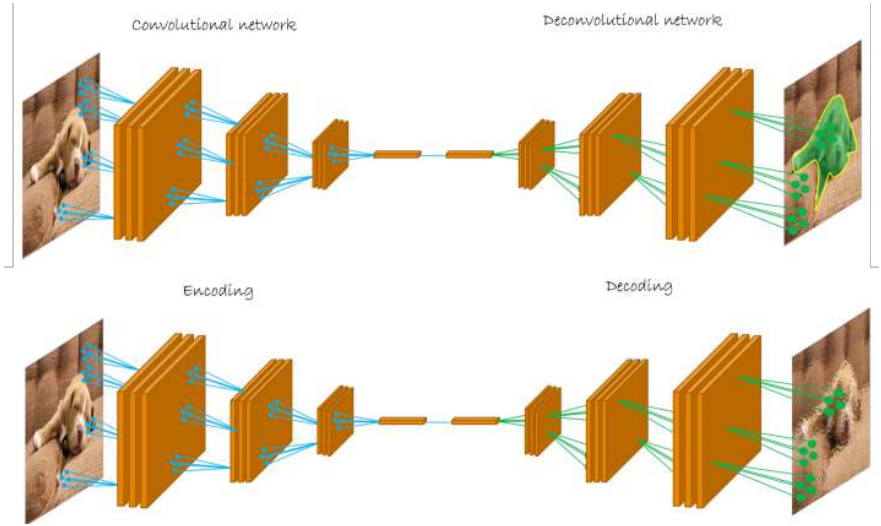
- 堆栈自动编码机 Stacked AutoEncoder



- 变分自动编码机 Variational AutoEncoder



- VAE 和 Deconv 网络对比



- 双口递归自联想记忆网络 Dual Ported RAAM

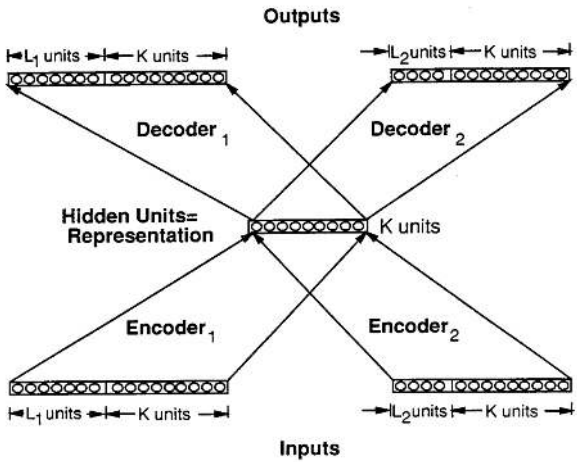


Figure 4: The Dual-Ported RAAM Architecture.

- 耦合结构的 RAAM

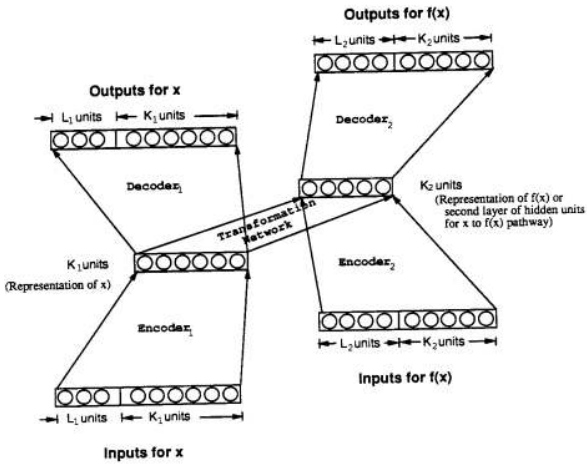
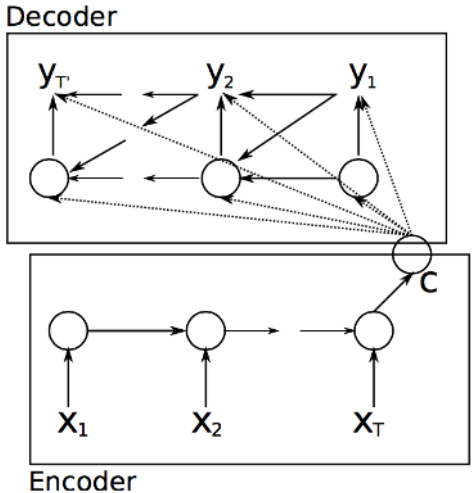


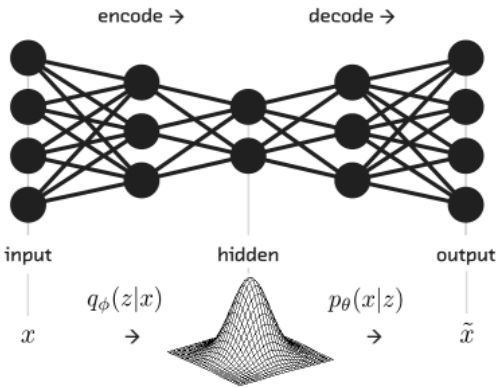
Figure 14: Coupled Hybrid Architecture.

- RNN 编码器解码器

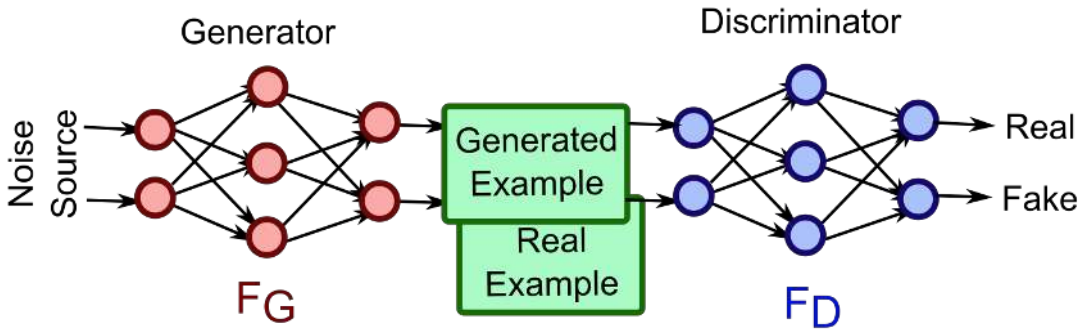


1.5 生成对抗网络 GAN

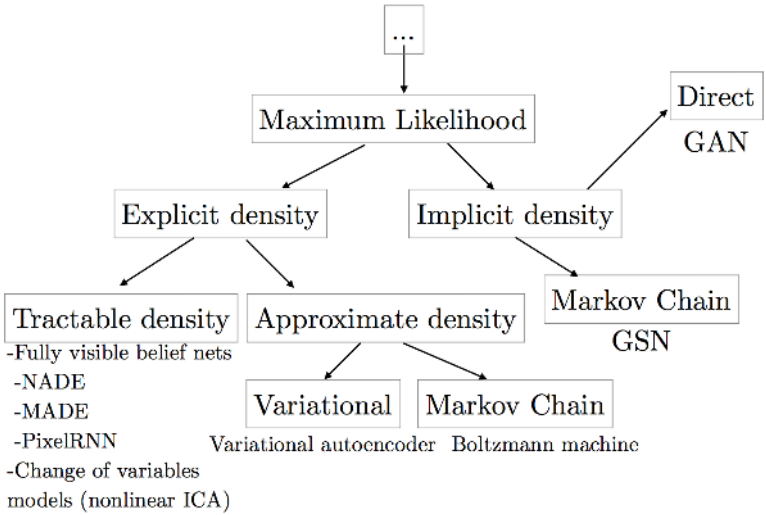
- 变分自动编码机 Variational AutoEncoder



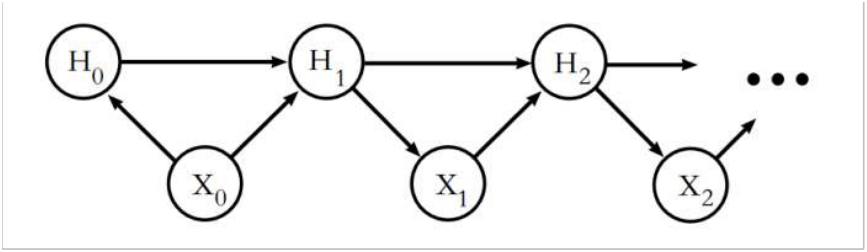
- 生成对抗网络 Generative Adversarial Network

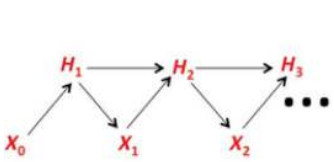


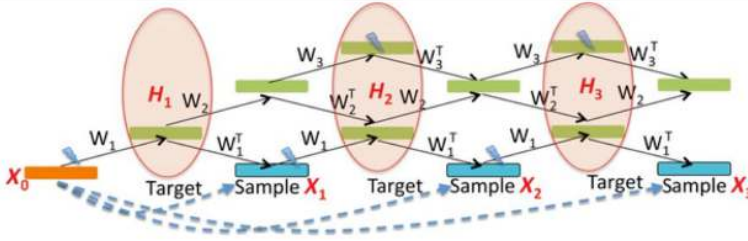
- 深度生成模型 Deep generative models



- 生成随机网络 Generative Stochastic Network



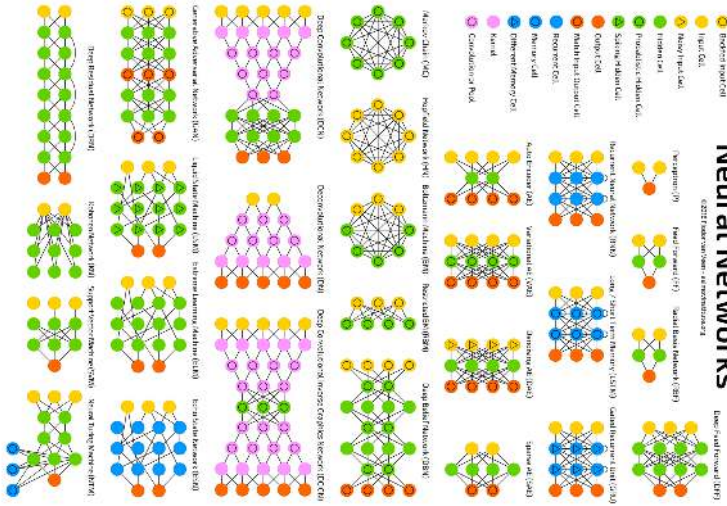




网络模型简单对比

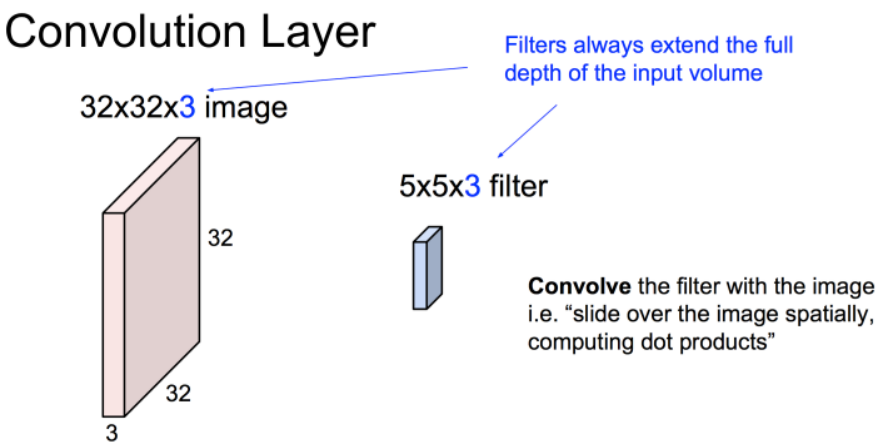
- 网络类型
  - 前向网络：CNN
  - 反馈网络：Deconv NN
  - 双向网络：DBM，Stacked AE
  - 顺序网络：LSTM，GRU
- 学习类型
  - 监督学习：CNN
  - 无监督学习：AE, Layer-wise Training
  - 半监督学习，强化学习：

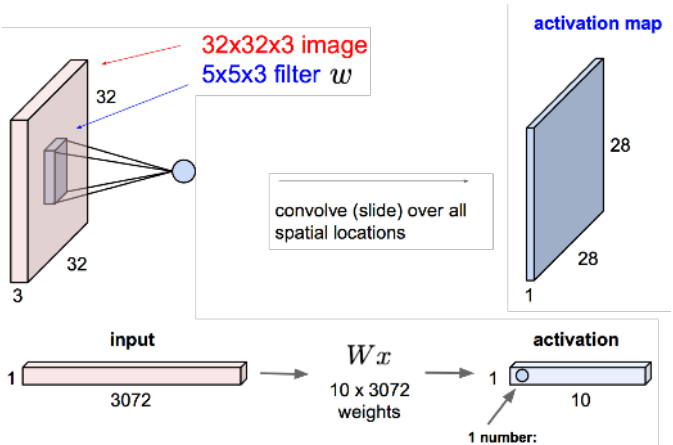
- 神经网络结构越来越多



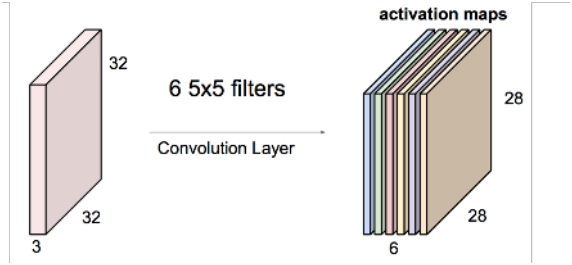
2 卷积神经网络 CNN 核心概念

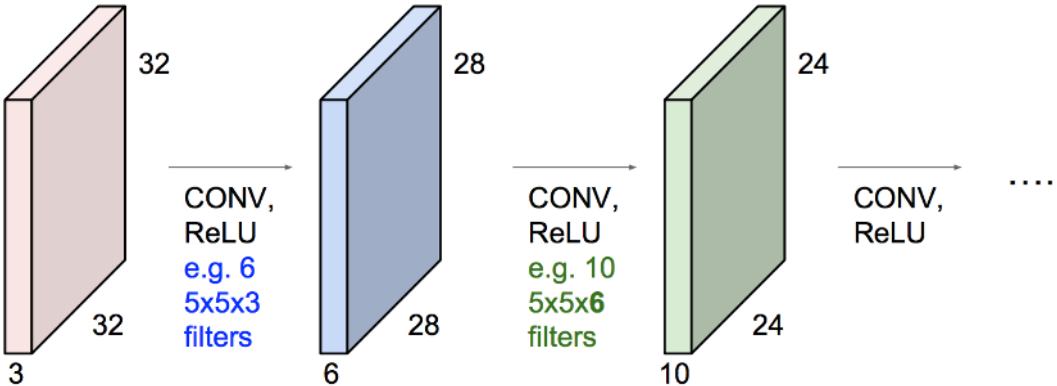
2.1 卷积层 Convolution

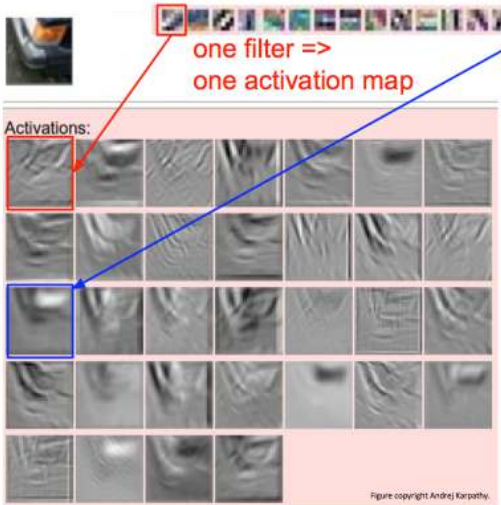



32x32x3 image
5x5x3 filter w
32
32
3
convolve (slide) over all
spatial locations
activation map
28
28
1
input
1
3072
Wx
10 x 3072
weights
activation
1
10
1 number:


32x32x3 image
5x5x3 filter
32
32
3
第2个绿色Filter
convolve (slide) over all
spatial locations
activation maps
28
28
1


32
32
3
6 5x5 filters
Convolution Layer
activation maps
28
28
6


32
32
3
CONV,
ReLU
e.g. 6
5x5x3
filters
28
28
6
CONV,
ReLU
e.g. 10
5x5x6
filters
24
24
10
CONV,
ReLU
....

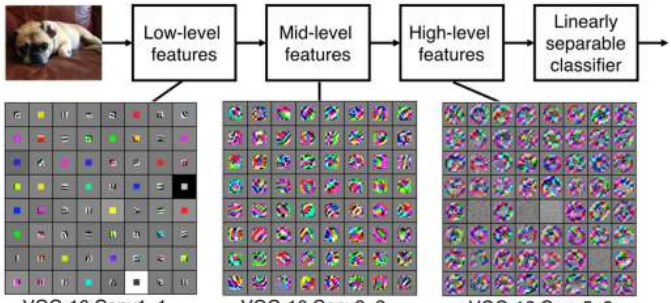


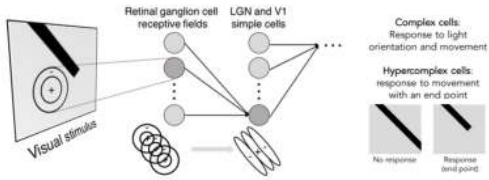
example 5x5 filters
(32 total)

We call the layer convolutional because it is related to convolution of two signals:

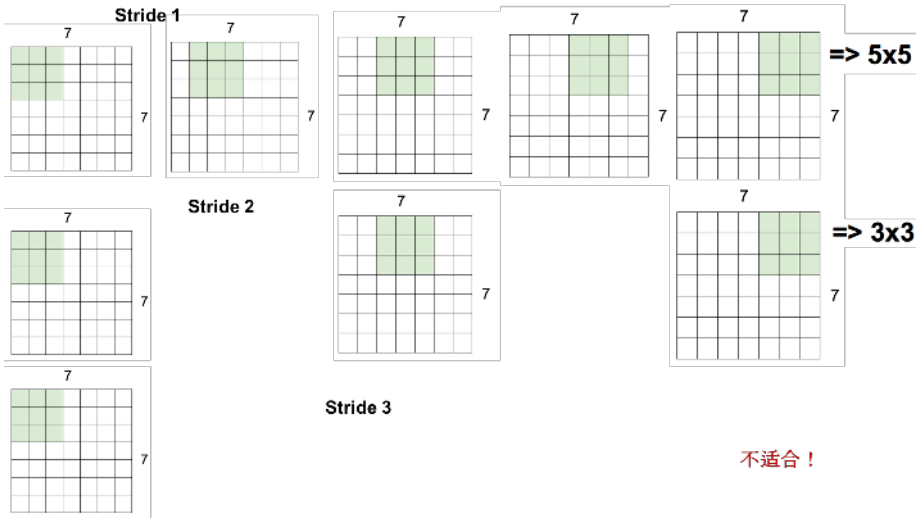
$$f[x,y] * g[x,y] = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} f[n_1,n_2] \cdot g[x-n_1, y-n_2]$$

elementwise multiplication and sum of a filter and the signal (image)

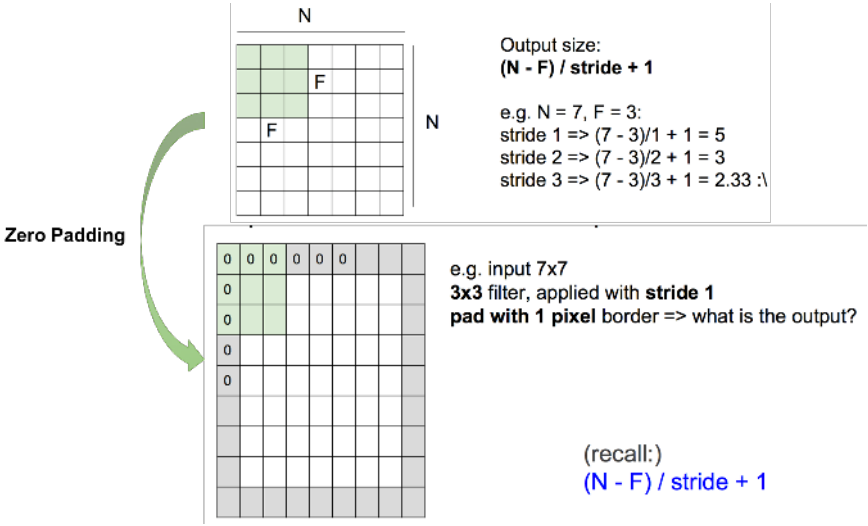

Low-level features
Mid-level features
High-level features
Linearly separable classifier
VGG-16 Conv1_1
VGG-16 Conv3_2
VGG-16 Conv5_3


Retinal ganglion cell receptive fields
LGN and V1 simple cells
Visual stimulus
Complex cells: Response to light orientation and movement
Hypercomplex cells: response to movement with an end point
No response
Response (end point)

- Stride 操作

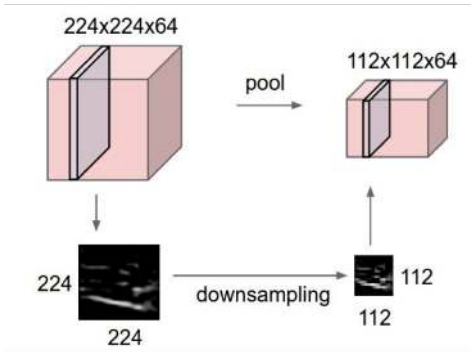


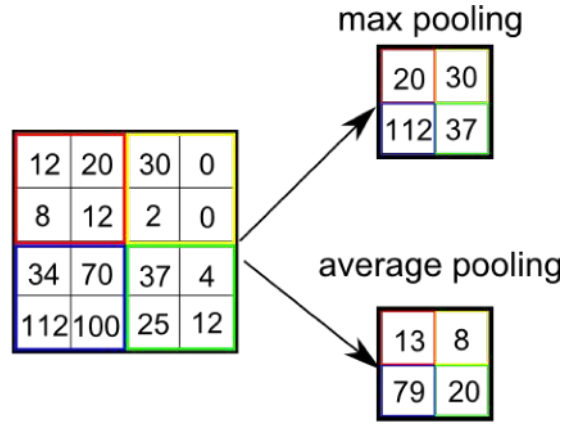
- Pad 操作


N
F
F
N
Output size:
(N - F) / stride + 1
e.g. N = 7, F = 3:
stride 1 => (7 - 3)/1 + 1 = 5
stride 2 => (7 - 3)/2 + 1 = 3
stride 3 => (7 - 3)/3 + 1 = 2.33 :\
Zero Padding
e.g. input 7x7
3x3 filter, applied with stride 1
pad with 1 pixel border => what is the output?
(recall:)
(N - F) / stride + 1

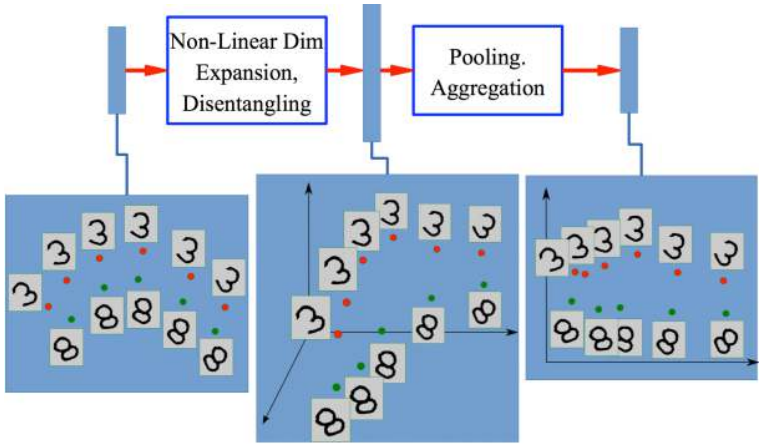
2.2 池化层 Pooling

- Pooling



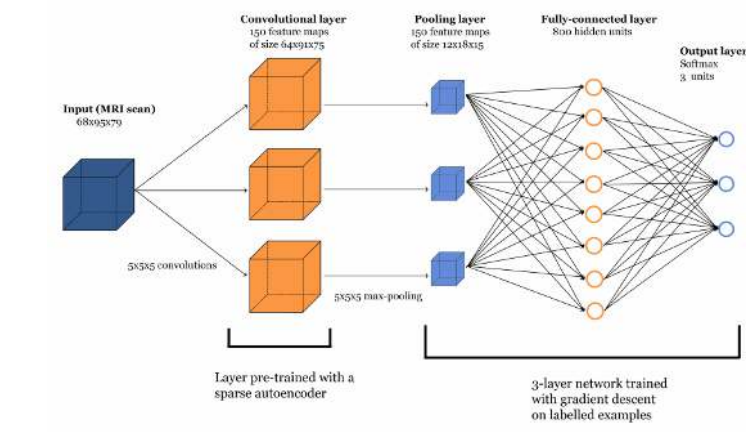


- 集成学习理解 Pooling



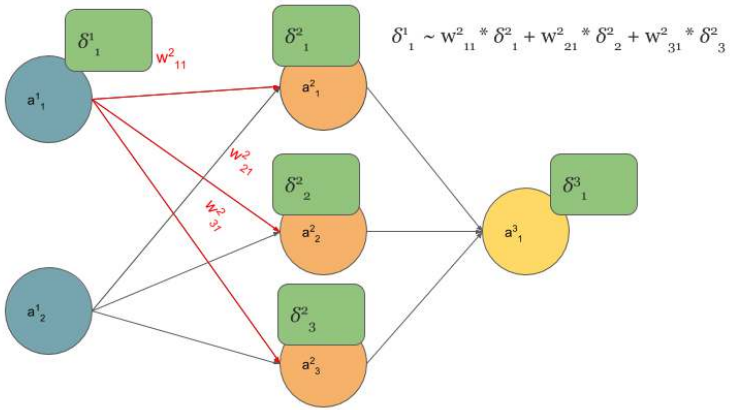
Average or Maximum！

- 全链接层 FC

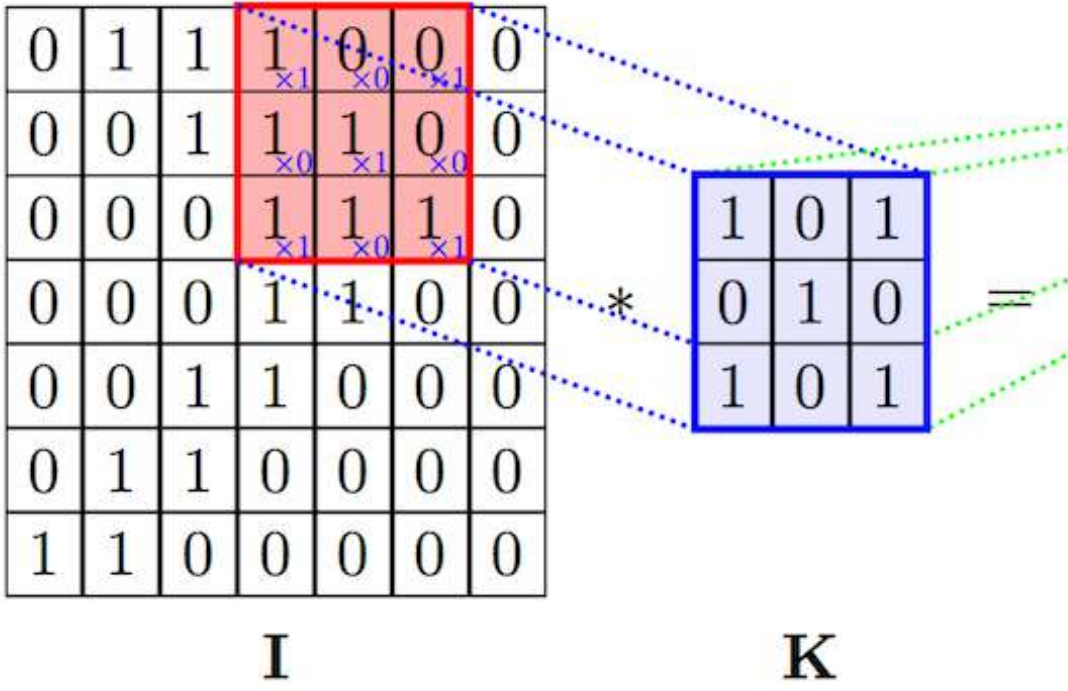

Convolutional layer
150 feature maps
of size 64x91x75
Pooling layer
150 feature maps
of size 12x18x15
Fully-connected layer
800 hidden units
Output layer
Softmax
3 units
Input (MRI scan)
68x95x79
5x5x5 convolutions
5x5x5 max-pooling
Layer pre-trained with a
sparse autoencoder
3-layer network trained
with gradient descent
on labelled examples

2.3 CNN 的 BP

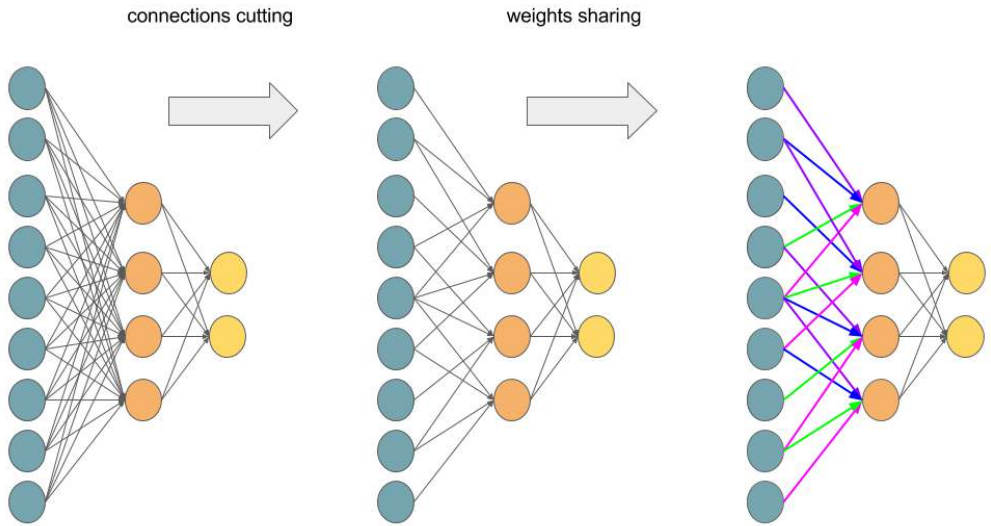
- MLP 的经典的 BP 算法



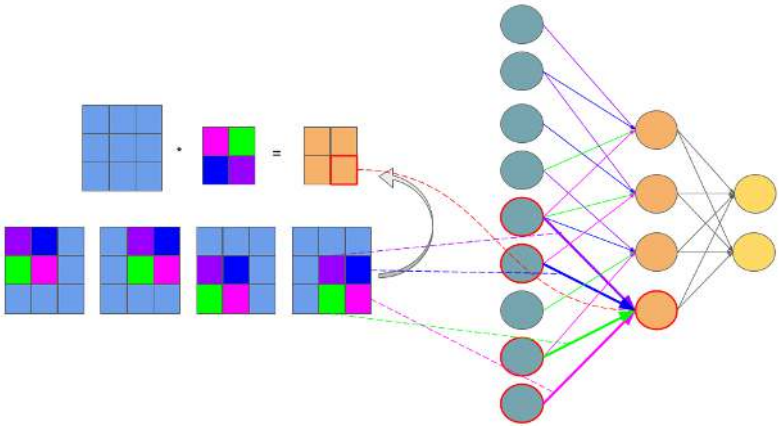
- CNN 层?


0 1 1 1 0 0 0
0 0 1 1 1 0 0
0 0 0 1 1 1 0
0 0 0 1 1 0 0
0 0 1 1 0 0 0
0 1 1 0 0 0 0
1 1 0 0 0 0 0
×1 ×0 ×1
×0 ×1 ×0
×1 ×0 ×1
I
*
1 0 1
0 1 0
1 0 1
=
K

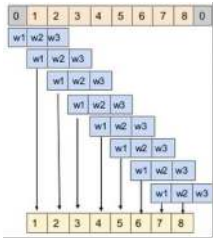
- CNN 等价到 MLP



- 权重共享



- Toeplitz 矩阵表达权重共享



$$f[n]=\{\underline{1},\ 2,\ 2\},\ h[n]=\{\underline{1},\ -1\},\ N_1=3, N_2=2$$

$$\mathbf{g}=\mathbf{H}\mathbf{f}=\begin{bmatrix}1&0&0\\-1&1&0\\0&-1&1\\0&0&-1\end{bmatrix}\begin{bmatrix}1\\2\\2\end{bmatrix}=\begin{bmatrix}1\\1\\0\\-2\end{bmatrix}$$

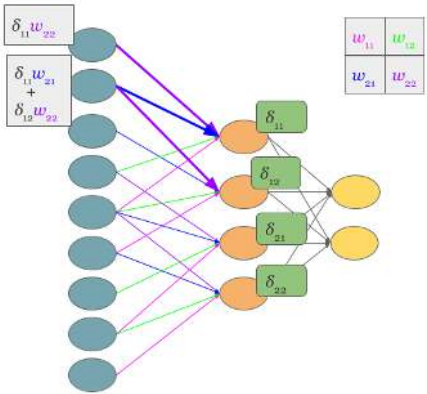
$$g[n]=\{\underline{1},\ 1,\ 0,\ -2\}$$

Toeplitz matrix

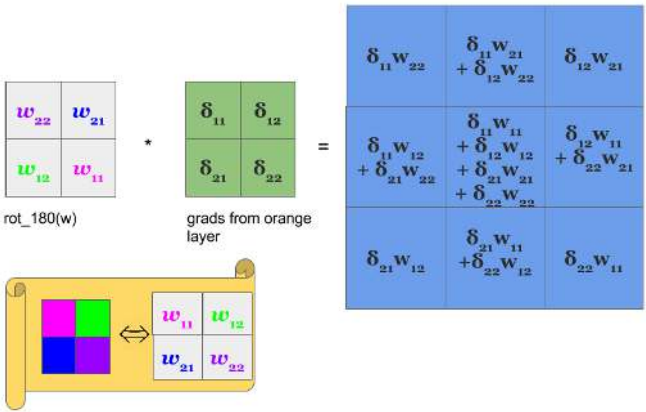
$$\begin{bmatrix}y_1\\y_2\\y_3\\y_4\\y_5\\y_6\\y_7\end{bmatrix}=\begin{bmatrix}x_1&0&0\\x_2&x_1&0\\x_3&x_2&x_1\\x_4&x_3&x_2\\x_5&x_4&x_3\\0&x_5&x_4\\0&0&x_5\end{bmatrix}\begin{bmatrix}f_1\\f_2\\f_3\end{bmatrix}$$

Otto Toeplitz

- BP 应用于等价 MLP 网络

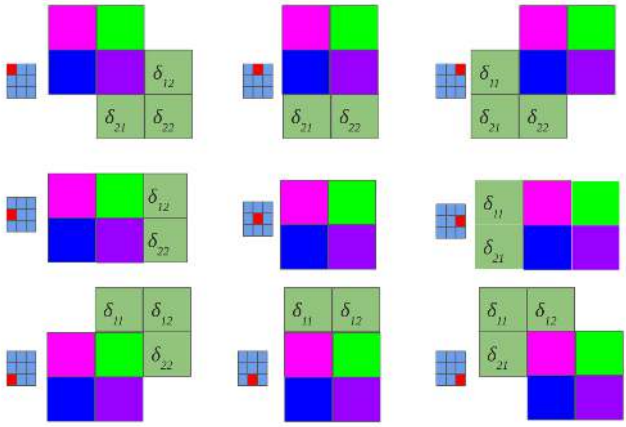


- 如何计算反向传播 delta？

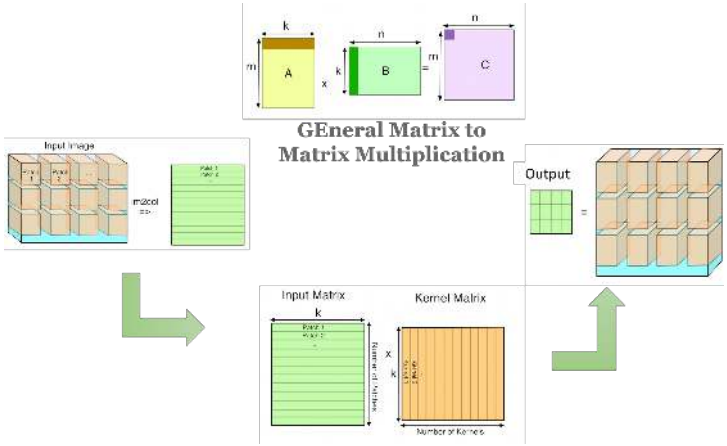


也是卷积

- 如何计算反向传播 delta？



- GEMM 广义矩阵到矩阵乘法

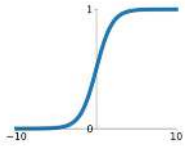


3 卷积神经网络 CNN 训练技巧

3.1 激活函数 Activation Functions

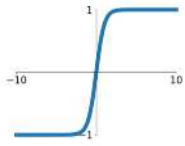
Sigmoid

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



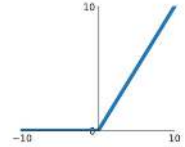
tanh

$\tanh(x)$



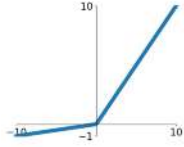
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

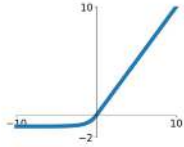


Maxout

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

ELU

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



- 为什么以前用 Sigmoid

1. 对 step function 很好的光滑近似
2. Universal function approximator （Universal approximation theorem）
3. 良好的导数形式 (BP 计算简单)

$$(dy)/(dx) = y(1-y)$$

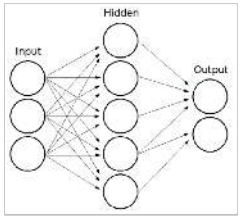
4. 良好的积分形式

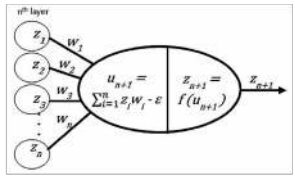
$$\int ydx = ln(1+e^x)$$

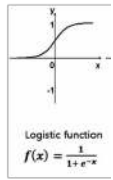
5. 良好的概率模型：逻辑回归等价
6. 连续的压缩到 [0, 1] 区间
7. 对神经元 0-1 激发率变化 fire rate 很好的模拟

- Universal approximation theorem 1 层隐含层

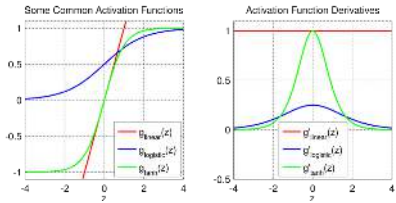
George Cybenko 1989年 Sigmoid 激活函数

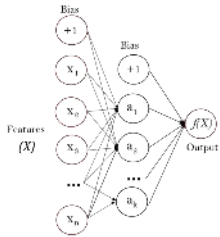






Kurt Honik 1991年 和激活函数不严格有关系

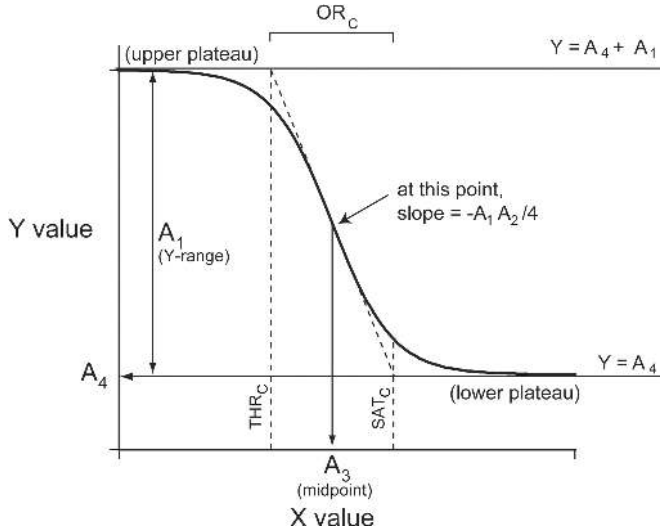




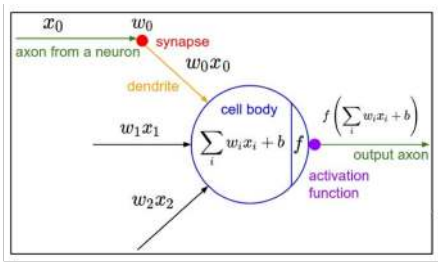
- 为什么现在不用 Sigmoid

1. 饱和的神经元没有梯度
2. 指数 exp() 计算依然代价太大
3. 不是以 0 为中心对称的

- 饱和神经元 Saturated



- 不是以 0 为中心对称的输出问题

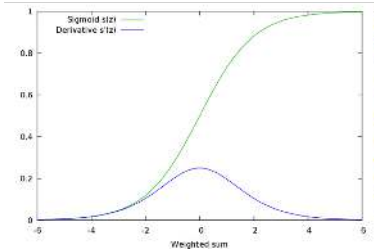


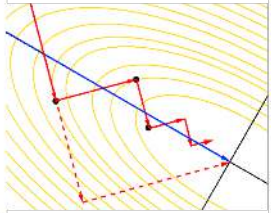
$$f = \sum w_i x_i + b$$

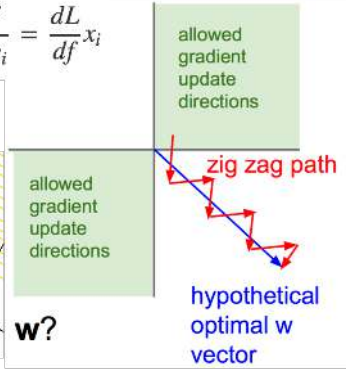
$$\frac{df}{dw_i} = x_i$$

$$\frac{dL}{dw_i} = \frac{dL}{df}\frac{df}{dw_i} = \frac{dL}{df}x_i$$

$$f\left(\sum_i w_i x_i + b\right)$$







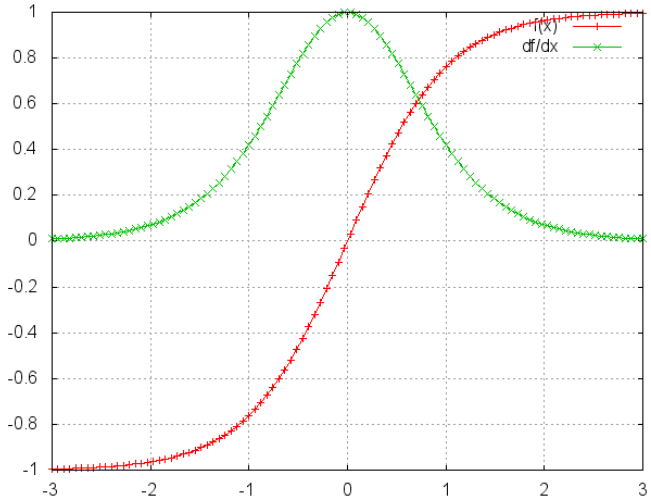
- tanh 激活函数

1. [LeCun et al., 1991] 提出
2. 对称输出 [-1, 1]，但是和生物上不太符合
3. 饱和造成梯度消失
4. 以 0 为中心对称
5. exp() 计算量大

$$tanh(x) = (e^x - e^{-x})/(e^x + e^{-x})$$

$$\begin{aligned}\frac{dtanh(x)}{dx} &= [(e^x + e^{-x})(e^x + e^{-x}) - (e^x - e^{-x})(e^x - e^{-x})]/(e^x + e^{-x})^2 \\ &= 1 - ((e^x - e^{-x})^2)/(e^x + e^{-x})^2 = 1 - tanh^2(x)\end{aligned}$$

- tanh 激活函数

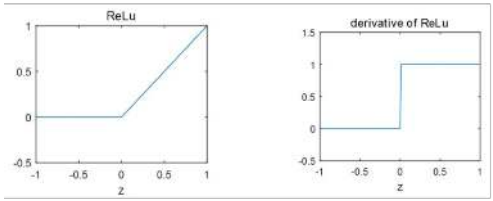


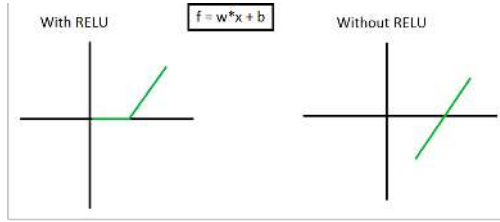
- ReLU 激活函数

1. [Krizhevsky et al., 2012] 提出
2. 不会过饱和
3. 计算简单
4. 收敛要比 sigmoid 和 tanh 快
5. 斜率为 1 的部分避免梯度消失
6. 斜率为 0 的部分实现稀疏性特征删除
7. 对激活没有上限限制，引起 blow up
8. 斜率为 0，导致 dying ReLU 问题
9. 不是以 0 为中心的输出
10. 0 点不可以导
11. 和 sigmoid 关系：
    - ReLU 的光滑近似 softplus 是 sigmoid 的积分
    - ReLU 导数的光滑近似是 sigmoid

- ReLU 激活函数

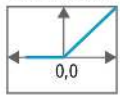
$$\sigma(z) = max(0, z) \qquad \frac{\partial \sigma(z)}{\partial z} = \begin{cases} 0 & if\ z \leq 0 \\ 1 & if\ z > 0 \end{cases}$$



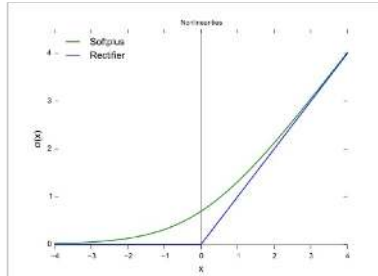


Transfer Function

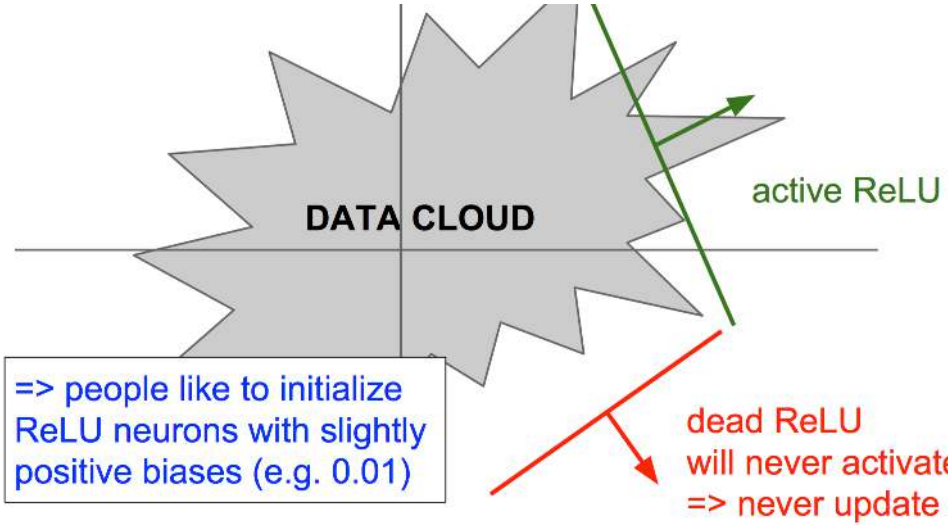
15	20	-10	35
18	-110	25	100
20	-15	25	-10
101	75	18	23



15	20	0	35
18	0	25	100
20	0	25	0
101	75	18	23



- Dying ReLU (Dead ReLU)



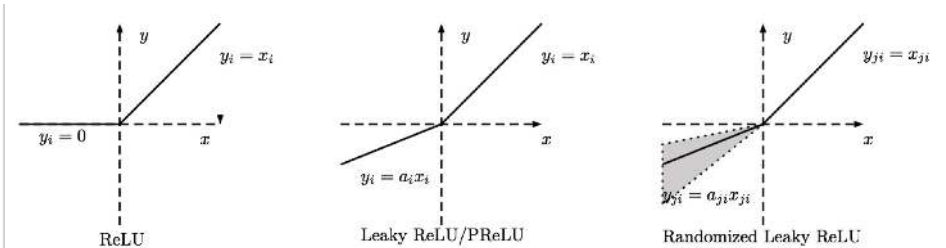
- Leaky ReLU / Parametric ReLU 激活函数

1. [Mass et al., 2013] 提出 LReLU
2. [He et al., 2015] 提出 PReLU
3. 收敛要比 sigmoid/tanh 快
4. 没有 dying ReLU 的问题
5. 没有 Batch Normalization 的时候 PReLU 效果要好点

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

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

- Randomized Leaky ReLU (RReLU)



Leaky ReLU

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{otherwise} \end{cases}$$

Parametric ReLU

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ ax & \text{otherwise} \end{cases}$$

Randomized leaky ReLU

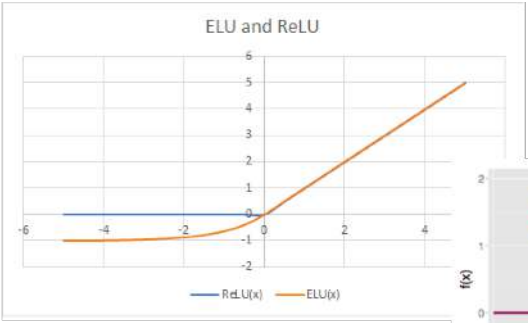
$$f(x) = \begin{cases} x & \text{if } x > 0 \\ ax & \text{otherwise} \end{cases}$$

- Exponential Linear Units (ELU) 激活函数

1. [Clevert et al., 2015] 提出
2. 部分解决以 0 为中心的问题
3. 稳定性效果比 ReLU 要好
4. exp() 计算量问题
5. 注意区别 SELU 和 SReLU

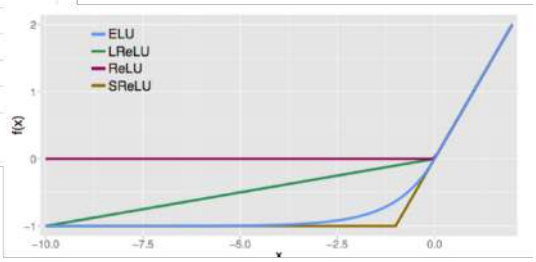
$$\max(0, ax) = a \max(0, x)$$

- Exponential Linear Units (ELU) 激活函数



Scaled ELU:

$$f(\alpha, x) = \lambda \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$



$$f(x) = \begin{cases} x & \text{if } x >= 0 \\ a[\exp(x) - 1] & \text{otherwise} \end{cases}$$

S-shaped ReLU:

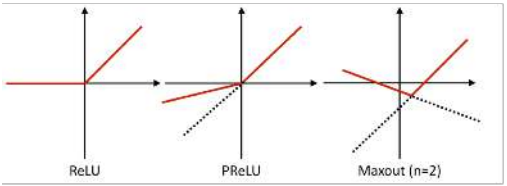
$$f'(\alpha, x) = \begin{cases} f(\alpha, x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$$

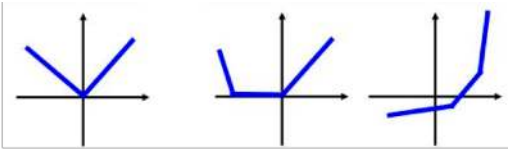
$$f_{t_l, a_l, t_r, a_r}(x) = \begin{cases} t_l + a_l(x - t_l) & \text{for } x \leq t_l \\ x & \text{for } t_l < x < t_r \\ t_r + a_r(x - t_r) & \text{for } x \geq t_r \end{cases}$$

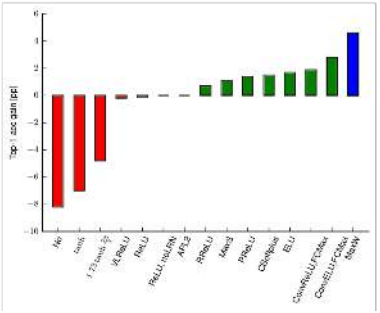
- Maxout 激活函数

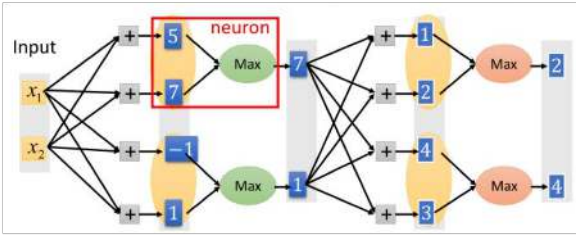
1. [Ian J Goodfellow, ICML 2013] 提出
2. 可以涵盖 ReLU 及其扩展
3. 可以学习线性凸函数
4. 计算参数增加

- Maxout 激活函数









- 其他激活函数

1. Noisy ReLUs

$$f(x) = \max(0, x + Y)\ Y \sim \mathcal{N}(0, \sigma(x))$$

2. Concatenated ReLU:

$$f(x) = (max(x, 0), max(-x, 0))$$

3. Softsign:

$$f(x) = x/(abs(x) + 1)$$

4. Softplus:

$$f(x) = x/(abs(x) + 1)$$

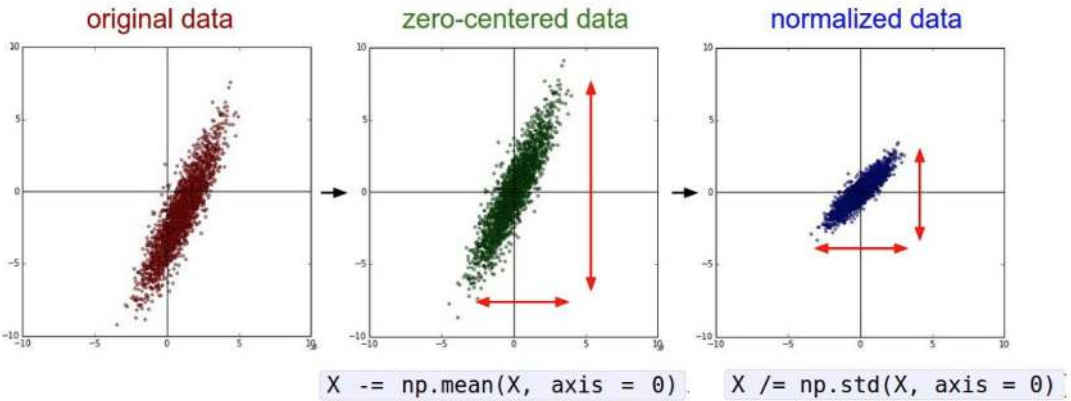
5. ReLU6:

$$f(x) = min(max(x, 0), 6)$$

- 激活函数小结：

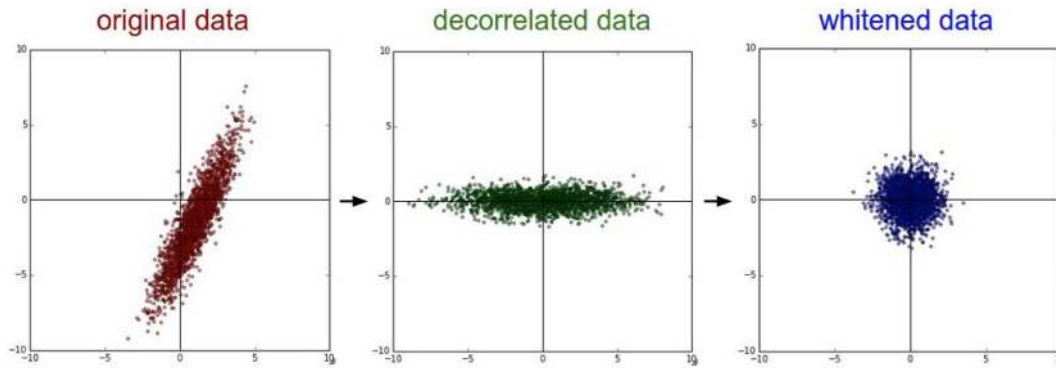
1. 神经元饱和 saturated 导致逼死 kill 梯度
2. 没有上下限，导致 blow up
3. 不以 0 为中心，导致收敛慢
4. Dying ReLU 问题
5. 计算量问题 exp(), # parameters
6. 不可导问题
7. 近似稀疏选择
8. 符合实际生物特征
9. 基本首次使用 ReLU
    - 速度和效果的平衡
    - 其他改进不如正则化效果好 (dropout, BN)
    - 之后再改进，尝试 LReLU, Maxout, ELU, CReLU

3.2 数据预处理



均值为 0，或者标准化，以 0 为中心，加速收敛

- PCA 解耦合数据

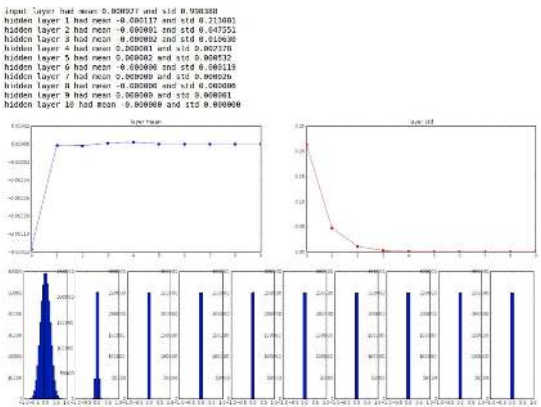

original data
decorrelated data
whitened data
(data has diagonal covariance matrix)
(covariance matrix is the identity matrix)

- 数据预处理小结

1. 均值为 0，中心化数据到 0
2. 分组中心化数据
3. 基本不做方差归一化
4. 基本不做 PCA 等解耦合

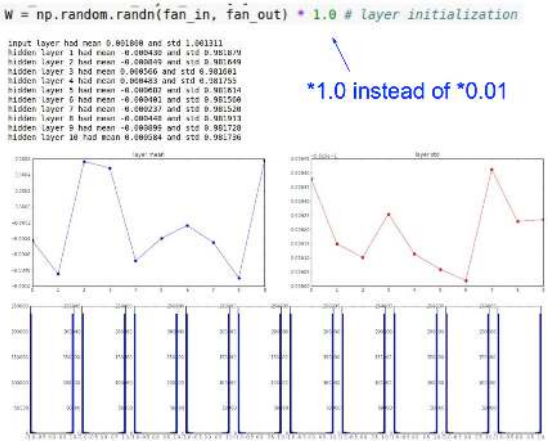
3.3 权重初始化

- 神经元死亡



激励函数输入全是 0

- 神经元饱和



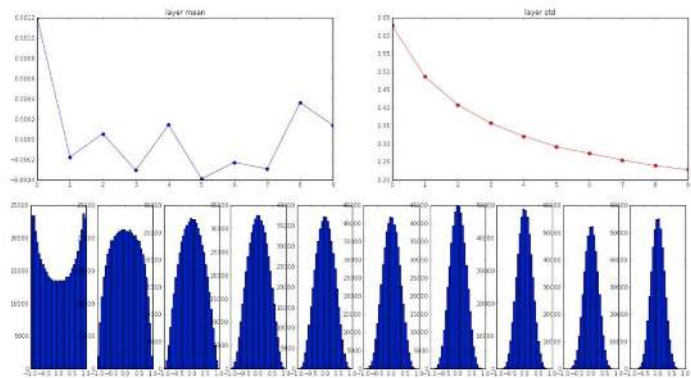
饱和神经元逼死梯度

- Xavier 初始化

```
input layer had mean 0.001000 and std 1.001311
hidden layer 1 had mean 0.001198 and std 0.627953
hidden layer 2 had mean -0.000175 and std 0.486051
hidden layer 3 had mean 0.000055 and std 0.407723
hidden layer 4 had mean -0.000306 and std 0.357108
hidden layer 5 had mean 0.000142 and std 0.320917
hidden layer 6 had mean -0.000389 and std 0.292116
hidden layer 7 had mean -0.000228 and std 0.273387
hidden layer 8 had mean -0.000291 and std 0.254935
hidden layer 9 had mean 0.000361 and std 0.239266
hidden layer 10 had mean 0.000139 and std 0.228088
```

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

"Xavier initialization"
[Glorot et al., 2010]



Reasonable initialization.
(Mathematical derivation assumes linear activations)

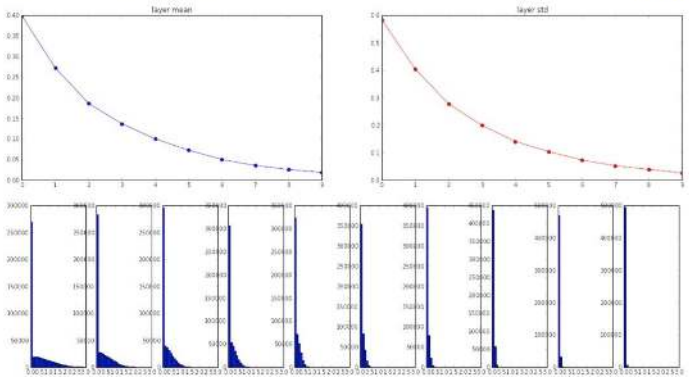
合理的权重初始化

- Xavier 初始化 ReLU 下断裂

```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.398623 and std 0.582273
hidden layer 2 had mean 0.272352 and std 0.403795
hidden layer 3 had mean 0.186076 and std 0.276912
hidden layer 4 had mean 0.136442 and std 0.198685
hidden layer 5 had mean 0.099568 and std 0.140299
hidden layer 6 had mean 0.072234 and std 0.103280
hidden layer 7 had mean 0.049775 and std 0.072748
hidden layer 8 had mean 0.035138 and std 0.051572
hidden layer 9 had mean 0.025404 and std 0.038583
hidden layer 10 had mean 0.018408 and std 0.026076
```

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

but when using the ReLU nonlinearity it breaks.



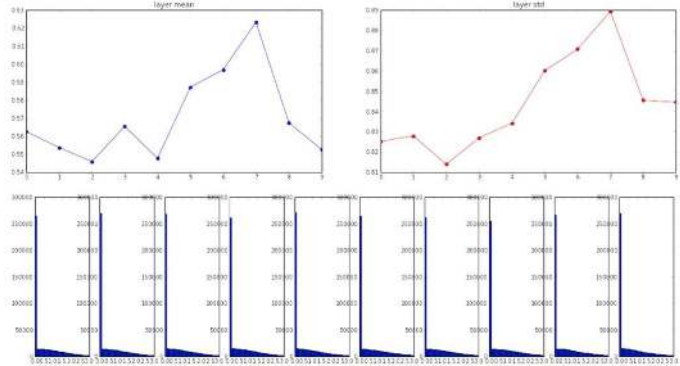
大部分输出进入 0 后

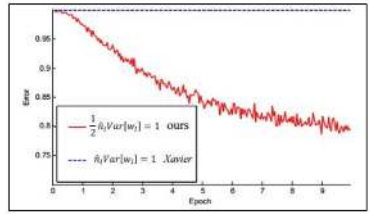
- Xavier 初始化 ReLU 下修正

```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.828982
hidden layer 5 had mean 0.547678 and std 0.834092
hidden layer 6 had mean 0.587103 and std 0.860035
hidden layer 7 had mean 0.596867 and std 0.878610
hidden layer 8 had mean 0.623214 and std 0.889348
hidden layer 9 had mean 0.567498 and std 0.845357
hidden layer 10 had mean 0.552531 and std 0.844523
```

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in/2) # layer initialization
```

He et al., 2015
(note additional /2)





加大权重

- Xavier 初始化

1. 假设全连接

$$Y = W_1X_1 + W_2X_2 + \cdots + W_nX_n$$

2. 每个输入的方差

$$\mathrm{Var}(XY) = [E(X)]^2\,\mathrm{Var}(Y) + [E(Y)]^2\,\mathrm{Var}(X) + \mathrm{Var}(X)\,\mathrm{Var}(Y).$$

所以有：

$$\mathrm{Var}(W_iX_i) = E[X_i]^2\mathrm{Var}(W_i) + E[W_i]^2\mathrm{Var}(X_i) + \mathrm{Var}(W_i)\mathrm{Var}(i_i)$$

3. 数据预处理下期望为 0

$$\mathrm{Var}(W_iX_i) = \mathrm{Var}(W_i)\mathrm{Var}(X_i)$$

4. 输出方差

$$\mathrm{Var}(Y) = \mathrm{Var}(W_1X_1 + W_2X_2 + \cdots + W_nX_n) = n\mathrm{Var}(W_i)\mathrm{Var}(X_i)$$

5. 维持输入输出方差一致

$$\mathrm{Var}(W_i) = \frac{1}{n} = \frac{1}{n_{\mathrm{in}}}$$

6. 考虑反向传播

$$\mathrm{Var}(W_i) = \frac{1}{n_{\mathrm{out}}}$$

7. 考虑反向传播

$$\mathrm{Var}(W_i) = \frac{2}{n_{\mathrm{in}} + n_{\mathrm{out}}}$$

8. tanh 几乎是线性，适用 Xavier

$$\mathrm{Var}(W) = \frac{1}{n_{\mathrm{in}}}$$

9. ReLU 只有一半输入被选中

$$\mathrm{Var}(W) = \frac{2}{n_{\mathrm{in}}}$$

10. sigmoid 中心斜率为 1/4

$$\mathrm{Var}(W) = \frac{13}{n_{\mathrm{in}}}$$

- Xavier 初始化均匀分布假设有 0 点对称均匀分布（Uniform）：

$$W \overset{iid}{\sim} U\left[-\theta, \theta\right]$$

均匀分布的方差：

$$\left\{x \sim U[a,b] \implies Var[x] = \frac{(b-a)^2}{12}\right\}$$

计算 0 点对称均匀分布的方差：

$$Var[W] = \frac{(2\theta)^2}{12} \tag{1}$$

$$Var[W] = \frac{\theta^2}{3} \tag{2}$$

根据 Xavier 初始化要求：

$$nVar[W] = 1$$

$$n\frac{\theta^2}{3} = 1 \tag{3}$$

$$\theta^2 = \frac{3}{n} \tag{4}$$

$$\theta = \frac{\sqrt{3}}{\sqrt{n}} \tag{5}$$

化简得到：

$$W \sim U\left[-\frac{\sqrt{3}}{\sqrt{n}}, \frac{\sqrt{3}}{\sqrt{n}}\right] \tag{6}$$

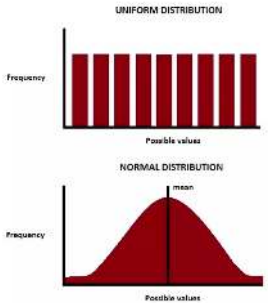
如果使用输入输出：

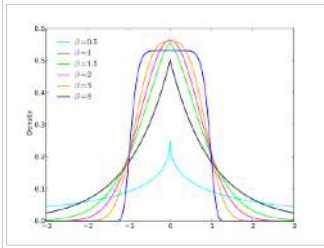
$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}, \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}\right] \tag{7}$$

就是大家最常见的一种均匀分布初始化：

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n+m}}, \frac{\sqrt{6}}{\sqrt{n+m}}\right] \tag{8}$$

- Xavier 初始化均匀分布近似





$$Var[W^i] = \frac{1}{n_{i+1}}$$

近似

$$W_{i,j} \sim U\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right]$$

- 各种初始化

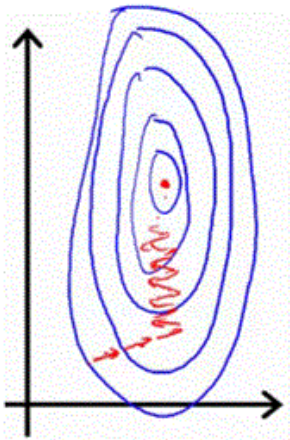
1. Glorot normal (Xavier normal) [Glorot & Bengio, AISTATS 2010]
   - stddev = sqrt(2 / (fan_in + fan_out))
2. Glorot uniform
   - sqrt(6 / (fan_in + fan_out))
3. He normal [He et al., http://arxiv.org/abs/1502.01852]
   - stddev = sqrt(2 / fan_in)
4. Lecun normal
   - stddev = sqrt(1 / fan_in)

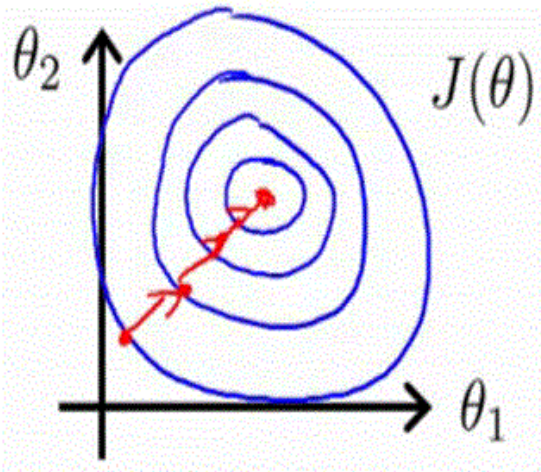
- 权重初始化小结

1. 0 附近的权重，在多层网络下，很容易消失
2. 1 附近的权重，在多层网络下，很容易饱和
3. Xavier 权重初始化
4. 根据实际情况修正

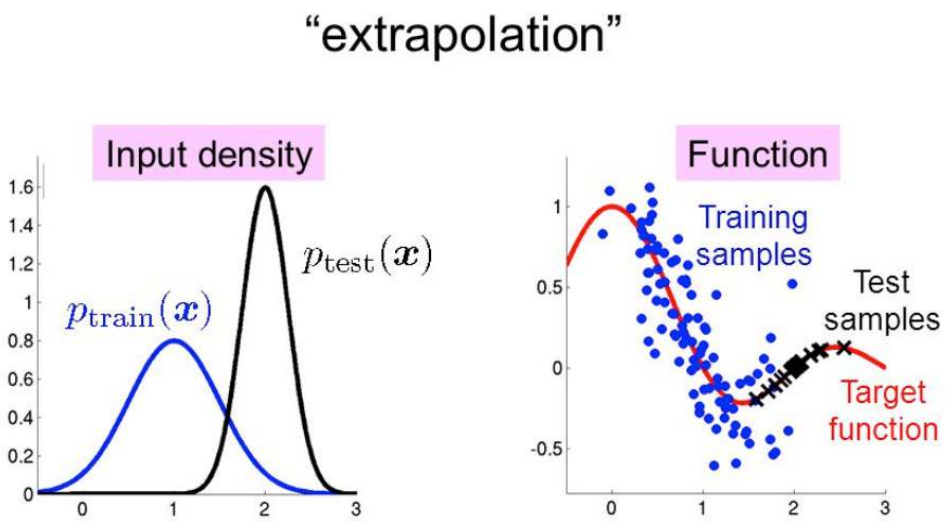
3.4 批标准化 Batch Normalization

- 标准化的好处: 收敛快

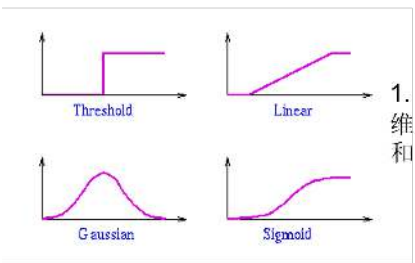




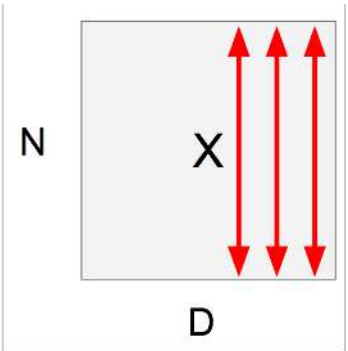
- 标准化的好处: Covariance Shift

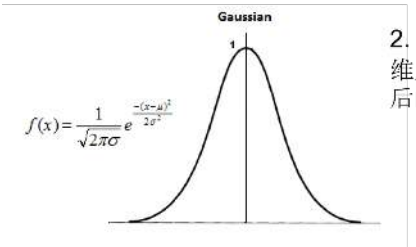


- 批标准化 [Ioffe and Szegedy, 2015]



1. 计算每个维度的均值和方差

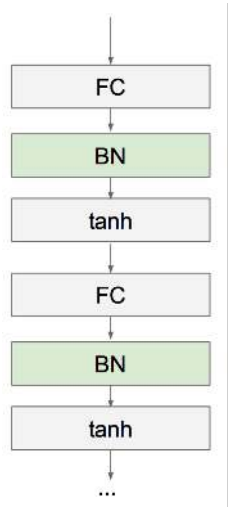




2. 计算这个维度标准化后的值

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

- BN 层一般放在激活层前



$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

$$y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}$$

$$\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathrm{E}[x^{(k)}]$$

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

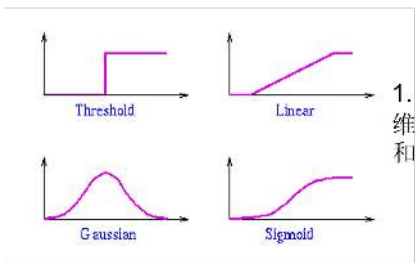
$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

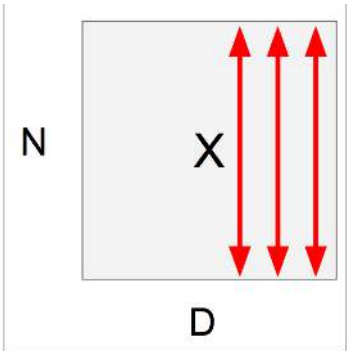
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

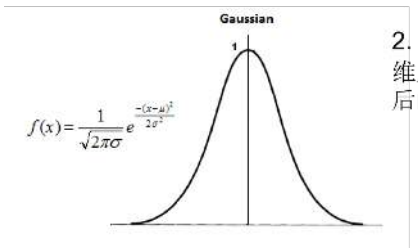
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- BN 层一般放在激活层前或者 FC/Conv 层后



1. 计算每个维度的均值和方差

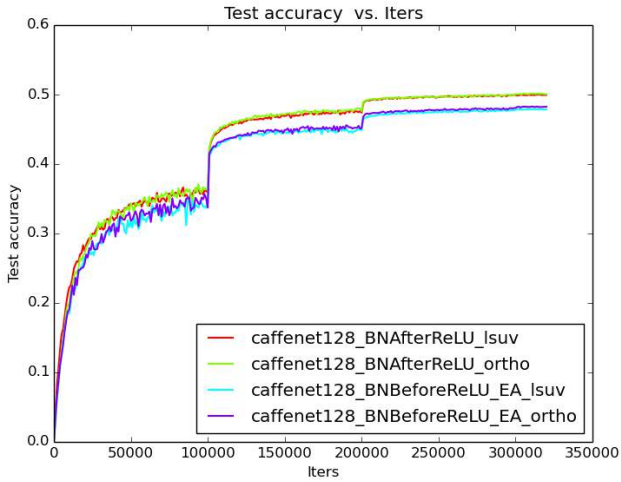




2. 计算这个维度标准化后的值

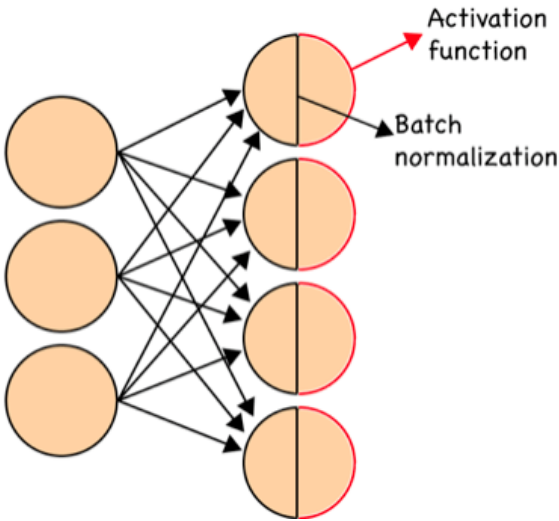
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

- BN 层放在激活层后效果不一定差



- BN 层加速

1. 不是每个批次都计算均值方差
2. 设定一个均值方差，一直使用

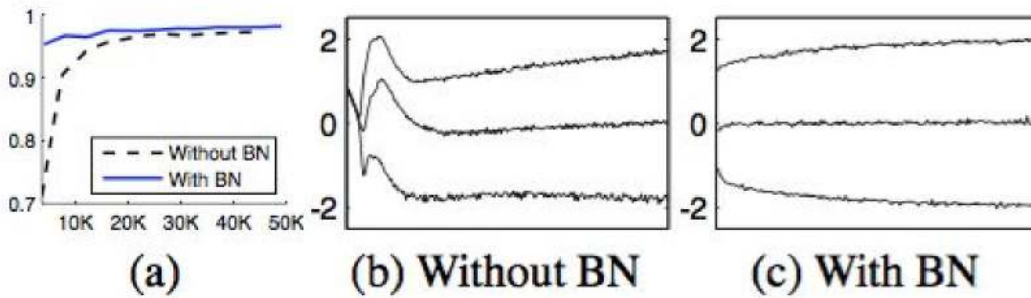


$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$$

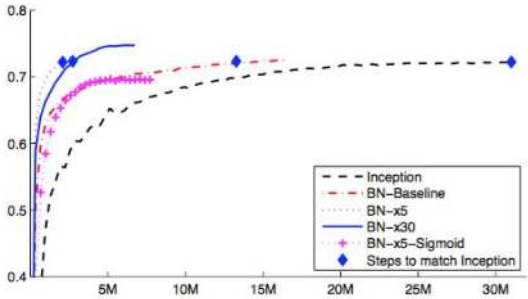
- MNIST 的 BN 效果

- Faster Convergence
- More Stable input distribution



- ImageNet 使用 Inception Net 的 BN 效果

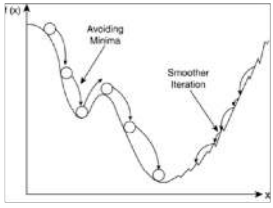
- Faster Convergence (30x)
- Same Accuracy

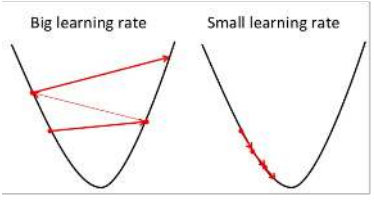


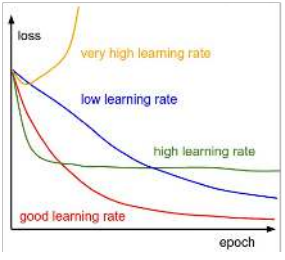
Model	Steps to 72.2%	Max accuracy
Inception	$31.0 \cdot 10^6$	72.2%
BN-Baseline	$13.3 \cdot 10^6$	72.7%
BN-x5	$2.1 \cdot 10^6$	73.0%
BN-x30	$2.7 \cdot 10^6$	74.8%
BN-x5-Sigmoid		69.8%

3.5 学习率 Learning Rate

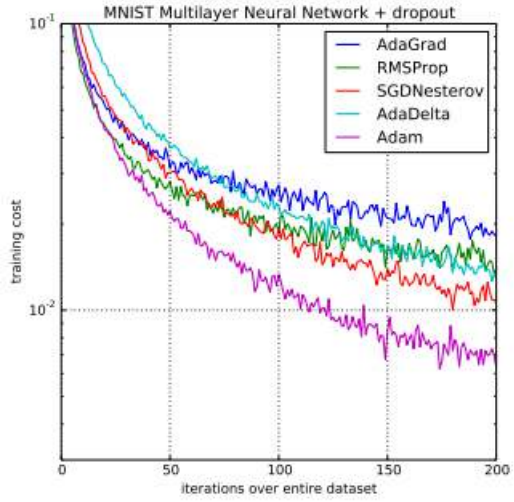
- 学习率的重要性



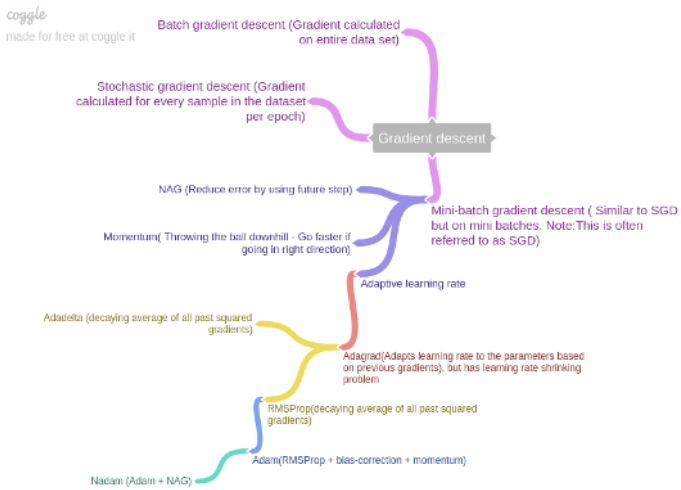




- 什么是自适应的学习率


MNIST Multilayer Neural Network + dropout
AdaGrad
RMSProp
SGDNesterov
AdaDelta
Adam
training cost
iterations over entire dataset
10^{-1}
10^{-2}
0
50
100
150
200

- 漫长的演化

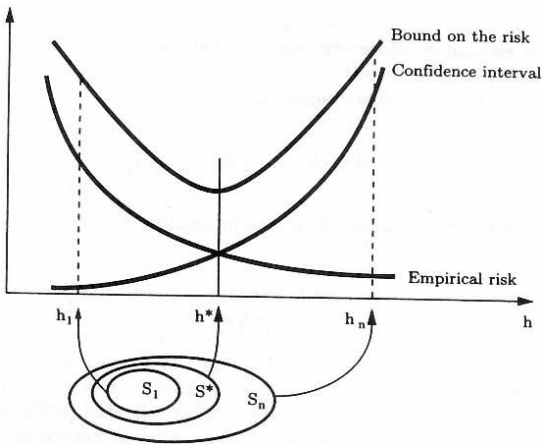

coggle
made for free at coggle.it
Batch gradient descent (Gradient calculated on entire data set)
Stochastic gradient descent (Gradient calculated for every sample in the dataset per epoch)
Gradient descent
NAG (Reduce error by using future step)
Momentum(Throwing the ball downhill - Go faster if going in right direction)
Mini-batch gradient descent (Similar to SGD but on mini batches. Note:This is often referred to as SGD)
Adaptive learning rate
Adadelta (decaying average of all past squared gradients)
Adagrad(Adapts learning rate to the parameters based on previous gradients), but has learning rate shrinking problem
RMSProp(decaying average of all past squared gradients)
Adam(RMSProp + bias-correction + momentum)
Nadam (Adam + NAG)

- 学习率的小结

1. 自适应的学习率已经成为主流
2. 基本使用 Adam 系列的优化器
3. 学习率的进步需要深厚的数学功底
4. 以后专题进行扩展

3.6 正则化

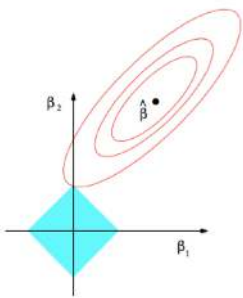
- 什么是正则化？结构风险最小的结构参数限制

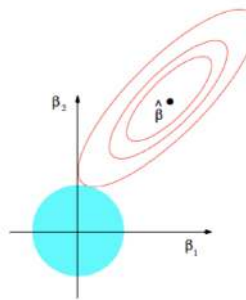


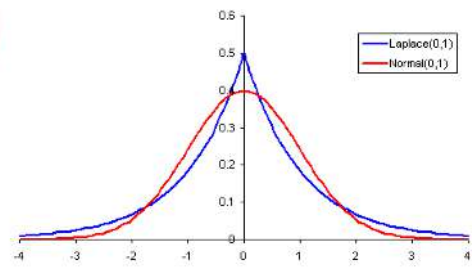
- 最常见的正则化 L1 、L2

$$L = \frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq y_i}\max(0, f(x_i;W)_j - f(x_i;W)_{y_i} + 1) + \lambda R(W)$$

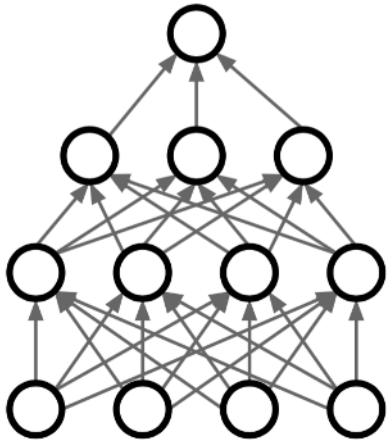
L2 regularization	$R(W) = \sum_k \sum_l W_{k,l}^2$ (Weight decay)
L1 regularization	$R(W) = \sum_k \sum_l \lvert W_{k,l}\rvert$
Elastic net (L1 + L2)	$R(W) = \sum_k \sum_l \beta W_{k,l}^2 + \lvert W_{k,l}\rvert$

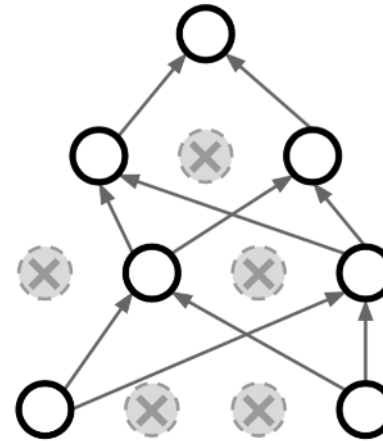






- 神经网络的正则化 Dropout

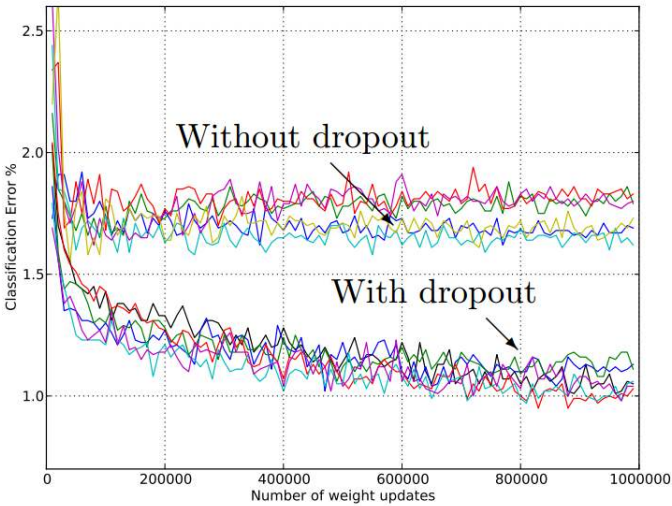




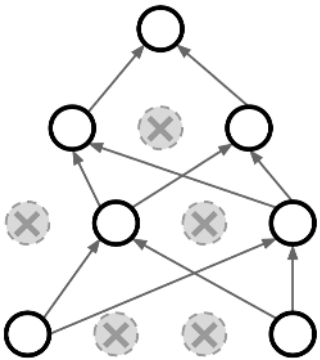
- Dropout 掩码 Mask

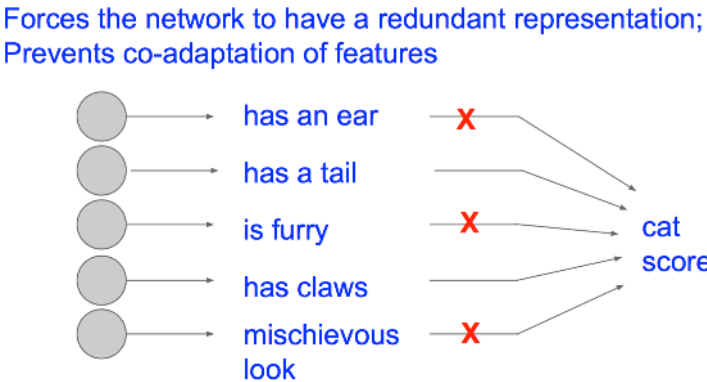
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)
```

- Dropout 的效果

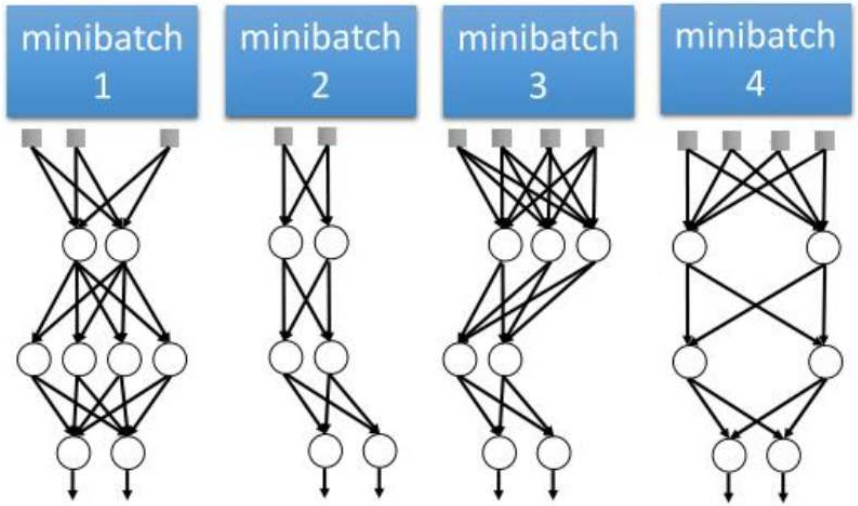

Without dropout
With dropout
Classification Error %
Number of weight updates
2.5
2.0
1.5
1.0
0
200000
400000
600000
800000
1000000

- Dropout 效果解释：特征选择

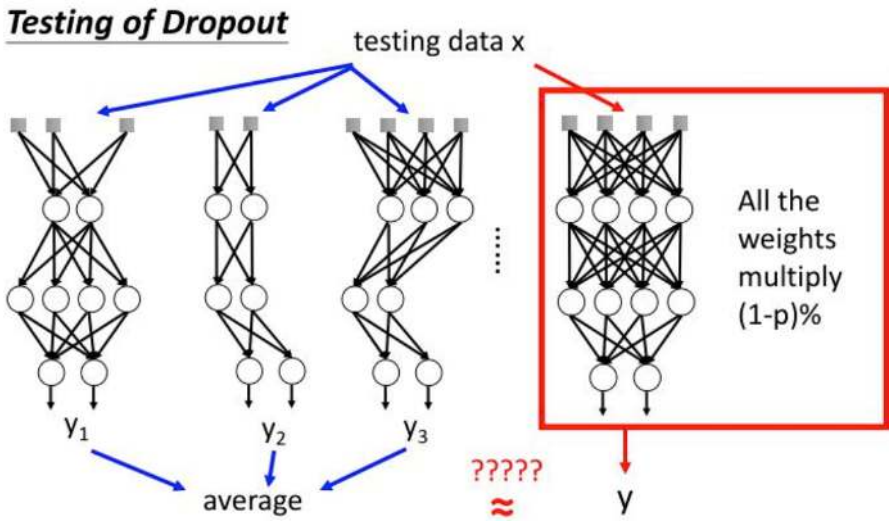




- Dropout 效果解释：集成学习



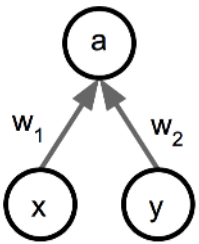
- Dropout 看成集成学习：如何测试？



- Dropout 测试：乘以 dropout 概率

Want to approximate the integral

$$y = f(x) = E_z\big[f(x,z)\big] = \int p(z) f(x,z) dz$$



Consider a single neuron.

At test time we have: $E[a] = w_1 x + w_2 y$

During training we have:

$$\begin{aligned} E[a] =& \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) \\ &+ \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) \\ =& \frac{1}{2}(w_1 x + w_2 y) \end{aligned}$$

- Dropout 训练和测试

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """

p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

Dropout Summary

drop in forward pass

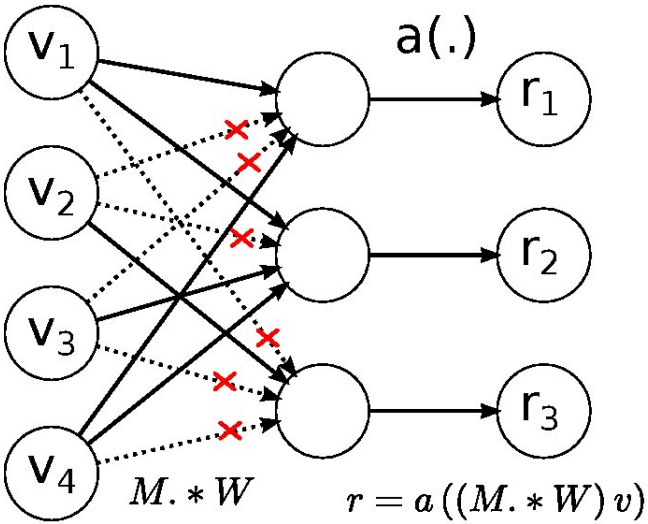
scale at test time

- Invert Dropout 训练和测试

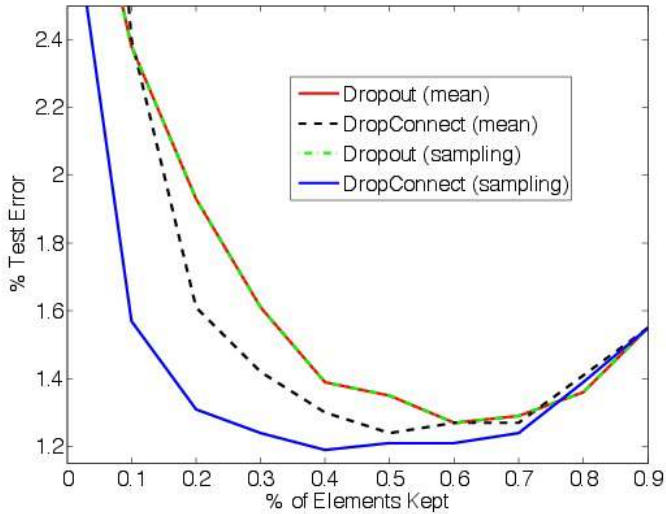
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

test time is unchanged!

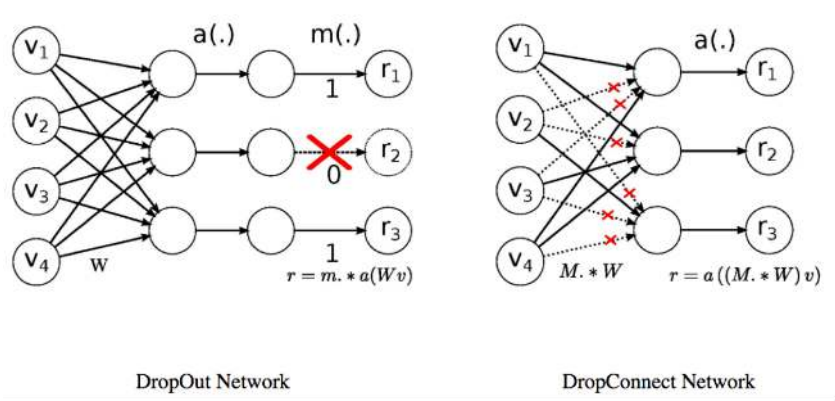
- 神经网络的正则化 Dropconnect



- Dropconnect 效果



- Dropconnect 实现

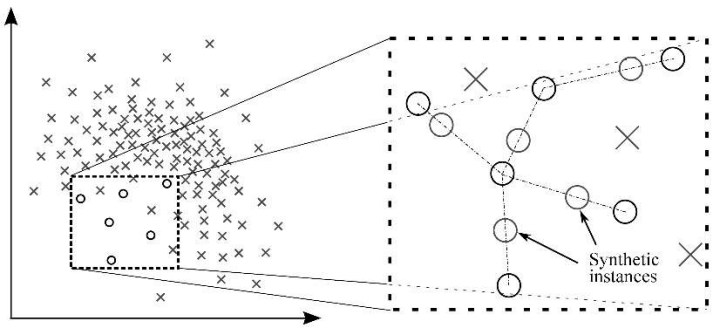



```
def dropconnect(W, p):
    return tf.nn.dropout(W, keep_prob=p) * p
```

3.7 数据增强 Data Augmentation

- 经典的表数据增强：

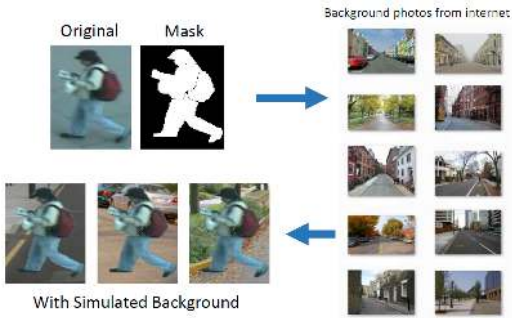
1. 再采样：up-sampling
2. 模拟数据：SMOTE，插值



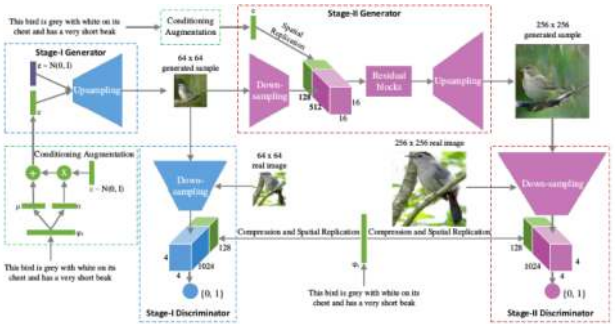
- 传统的图像数据增强：变换数据

1. 平移，缩放，旋转，折叠，扭曲
2. 色彩，亮度，光泽调整

- 传统的图像数据增强：加噪声，重建数据



- 新型的图像数据增强：GAN、VAE 再生数据



- 传统图像数据增强的效果要好

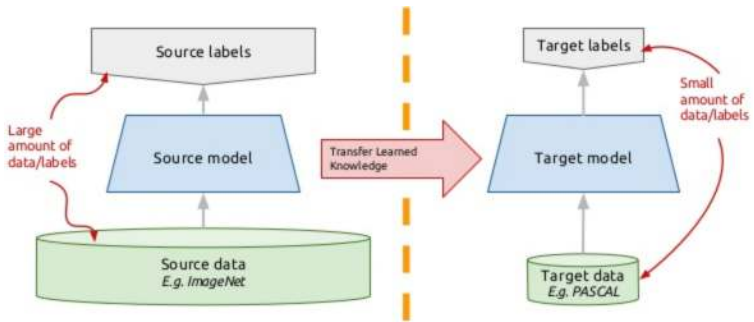
Dogs vs Cat	
Augmentation	Val. Acc.
None	0.705
Traditional	**0.775**
GANs	0.720
Neural + No Loss	0.765
Neural + Content Loss	0.770
Neural + Style	0.740
Control	0.710

- 数据增强小结

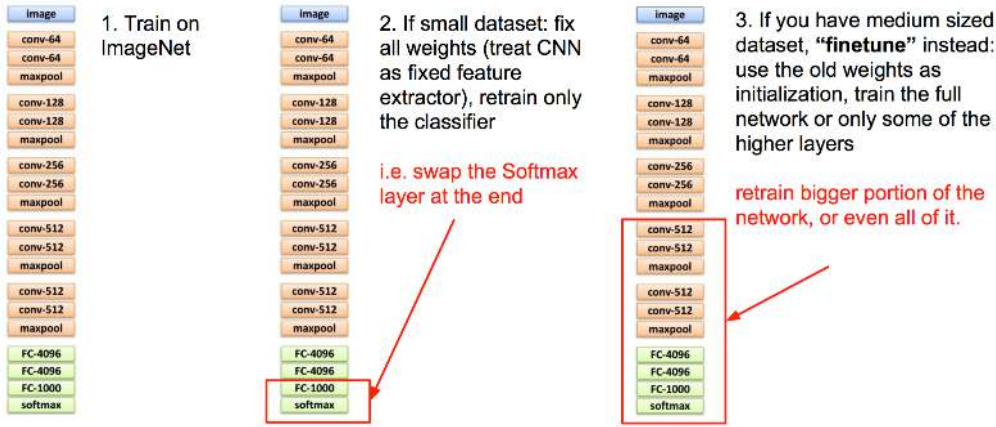
1. 插值技术：同类数据插值
2. 标记技术：按组重新标记数据
3. 集成技术：利用均值等集成
4. 概率技术：按分布生成数据

3.8 迁移学习 Transfer Learning

- 大小数据集条件下的迁移学习



- 迁移学习的模式



按数据量控制已经学习好的模型的学习层

3.9 梯度检查

3.10 训练技巧举例

- “Dogs vs Cats” Kaggle challenge

- 技巧使用与效果提升

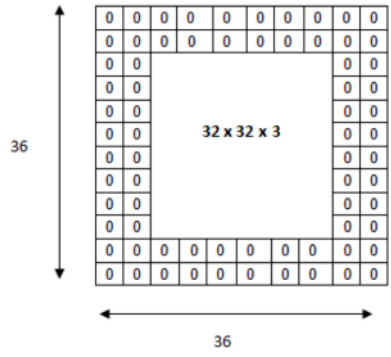
1. 数据增强 rotation + cropping + flipping + resizing：78% -> 87%
2. 迁移学习：78% -> 91%
3. Dropout: 91.5% -> 92.6%
4. BN：92.6% -> 94.9%
5. Ensemble 技术: 94.9% -> 97.5%
    - Global Maxpooling 池化层: Multi-scale training
6. ReLU -> PReLU: 97.5% -> 97.9%

以后我们会详细进行实验！

4 卷积神经网络 CNN 主流架构

- 卷积网络大小计算

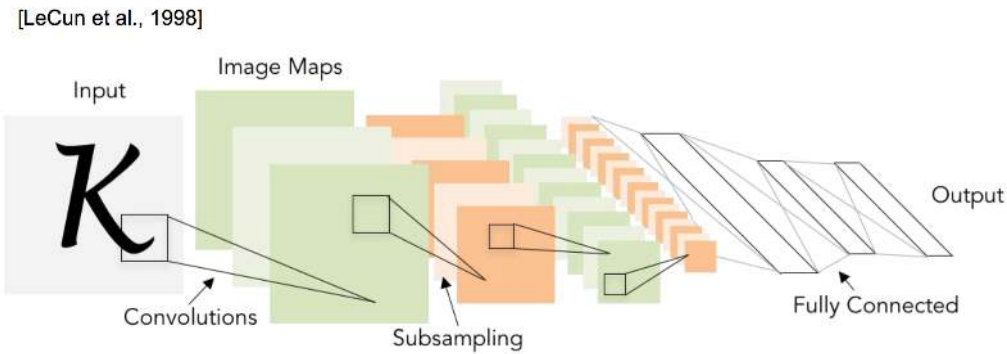
$$O = \frac{(W - K + 2P)}{S} + 1$$




```
def outFromIn(isz, layernum = 9, net = convnet):
    if layernum>len(net): layernum=len(net)

    totstride = 1
    insize = isz
    #for layerparams in net:
    for layer in range(layernum):
        fsize, stride, pad = net[layer]
        outsize = (insize - fsize + 2*pad) / stride + 1
        insize = outsize
        totstride = totstride * stride

    RFsize = isz - (outsize - 1) * totstride

    return outsize, totstride, RFsize
```

4.1 LeNet

- LeNet 5 层

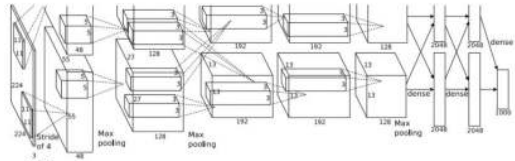


4.2 AlexNet

- AlexNet 8 层

Case Study: AlexNet

[Krizhevsky et al. 2012]



Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

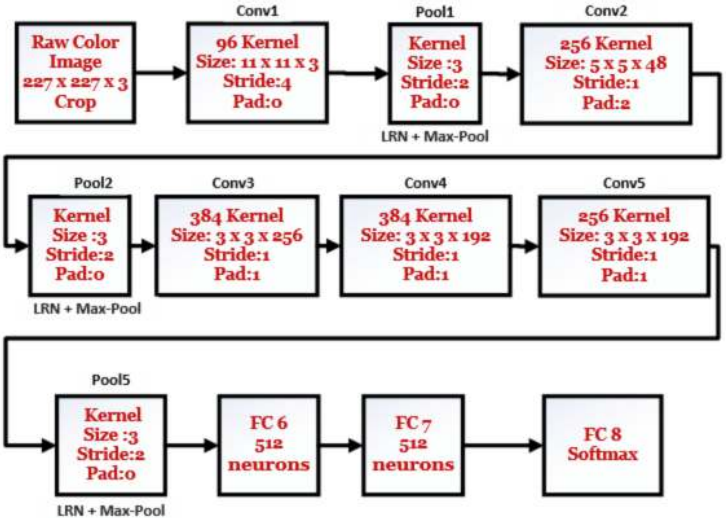
- 从 LeNet 到 AlexNet

1. 首次使用 ReLU
2. 首次使用 Normalization 层
3. 重量型数据增强
4. 使用 Dropout
5. 结构变化
   - 从 5 层编程 8 层
   - 增加 2 层 Conv，1 层 FC
   - 首次出现连续 3 层 Conv
6. 主要参数
   - Dropout:50%
   - batch size 128
   - SGD Momentum 0.9
   - Learning rate 1e-2, reduced by 10 manually when val accuracy plateaus
   - L2 weight decay 5e-4

- 7 CNN ensemble: 18.2% -> 15.4%

7. 硬件：首次使用 2 个 GTX 580 硬件 (3G 内存)
   - 每个 GPU 跑一半神经元
   - CONV1, CONV2, CONV4, CONV5
   - CONV3, FC6, FC7, FC8
   - 跑了 5、6 天

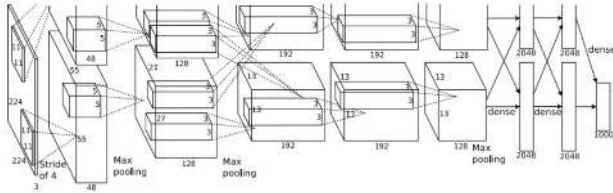
- AlexNet Stride 和 Pad 变化



- 回顾 AlexNet 输出神经元个数计算

Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images
After CONV1: 55x55x96

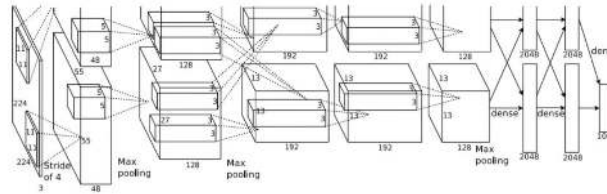
Second layer (POOL1): 3x3 filters applied at stride 2

Q: what is the output volume size? Hint: (55-3)/2+1 = 27

- AlexNet 卷积层参数

Case Study: AlexNet

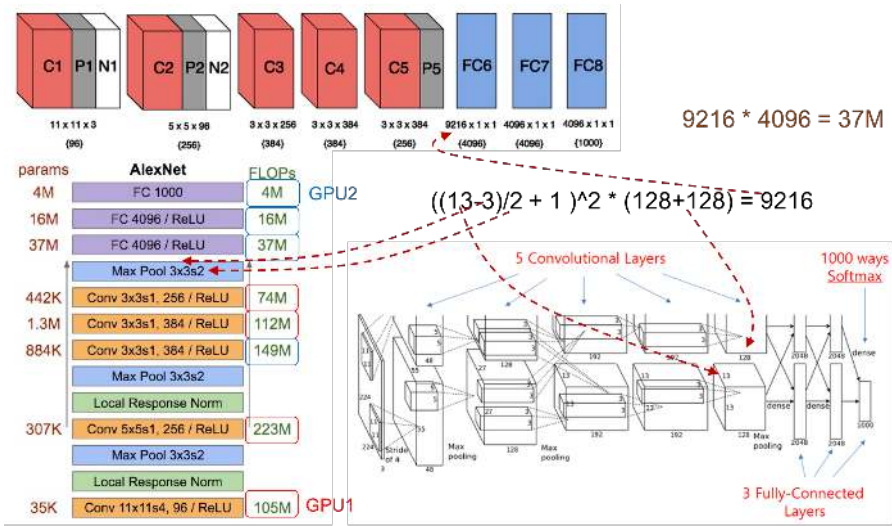
[Krizhevsky et al. 2012]



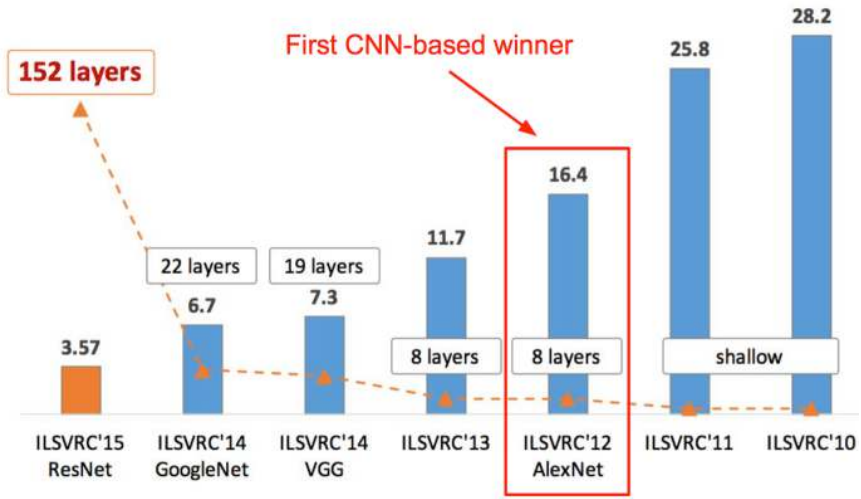
Input: 227x227x3 images

First layer (CONV1): 96 11x11 filters applied at stride 4
=>
Output volume **[55x55x96]**
Parameters: (11*11*3)*96 = **35K**

- AlexNet FC 层参数

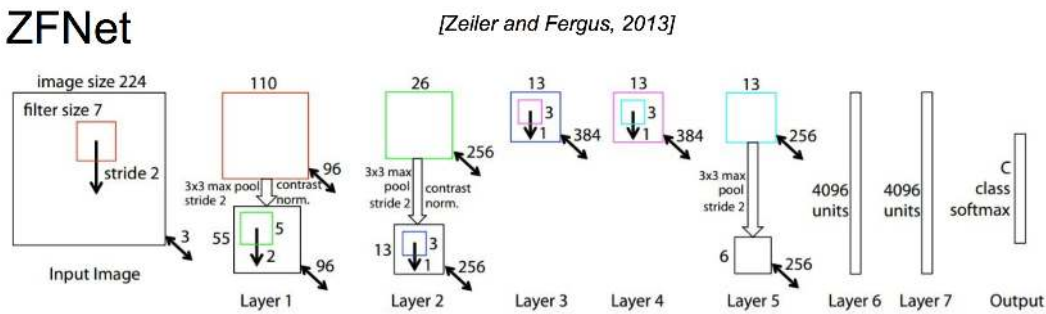


- AlexNet 突破性效果：ILSVRC2012



4.3 ZFNet

- ZFNet 8 层

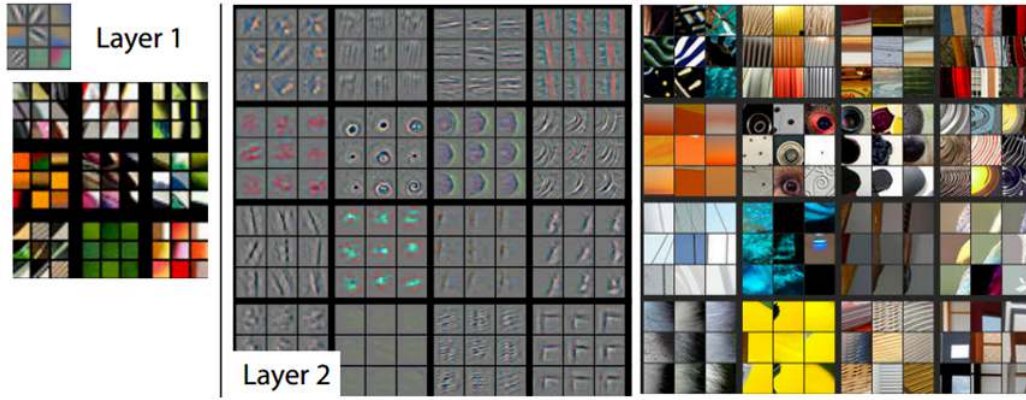


TODO: remake figure

AlexNet but:
CONV1: change from (11x11 stride 4) to (7x7 stride 2)
CONV3,4,5: instead of 384, 384, 256 filters use 512, 1024, 512

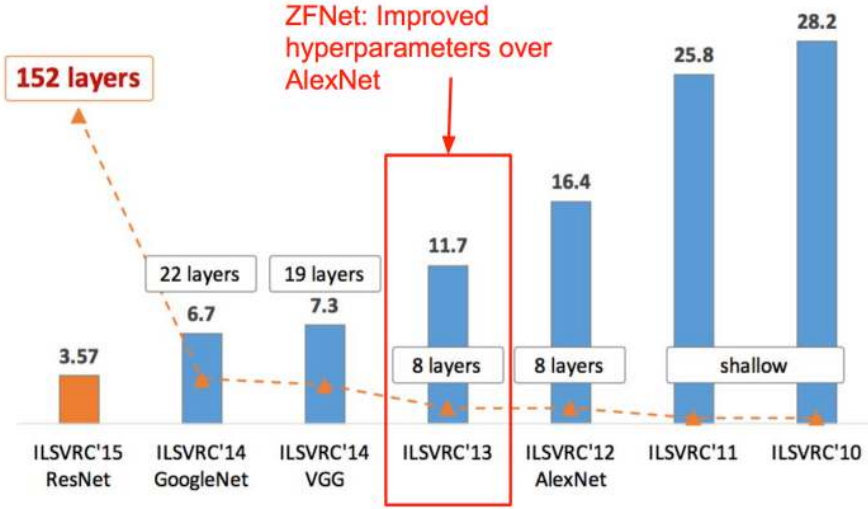
ImageNet top 5 error: 16.4% -> 11.7%

- ZFNet 引入训练可视化:DeConv



Visualizations of Layer 1 and 2. Each layer illustrates 2 pictures, one which shows the filters themselves and one that shows what part of the image are most strongly activated by the given filter. For example, in the space labled Layer 2, we have representations of the 16 different filters (on the left)

- ZFNet 效果：ILSVRC2013

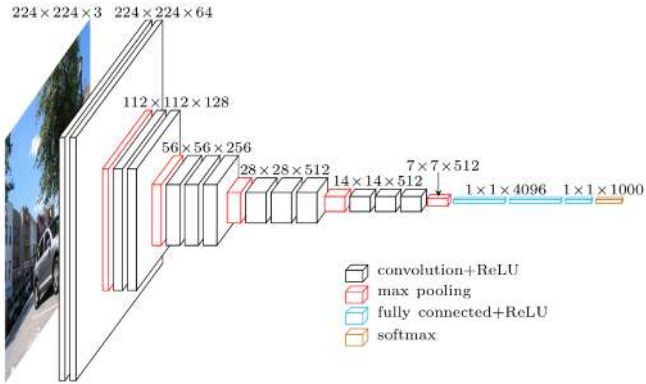


- 从 AlexNet 到 ZFNet

1. Conv1 卷积层 Filter 大小：11x11 -> 7x7
2. Conv3,4,5 卷积层神经元数目: 384, 384, 256 -> 512, 1024, 512
3. 同样的 8 层结构，减小的 Filter 大小，增大的神经元数目和参数数量
4. 第一次使用了 Cross-Entropy 损失。
5. 一个 GPU，跑了 12 天
6. 发明了 Deconvolution 网络检查训练效果，开启可视化训练

4.4 VGGNet

- VGGNet 16 层



- VGGNet 卷积结构

Case Study: VGGNet

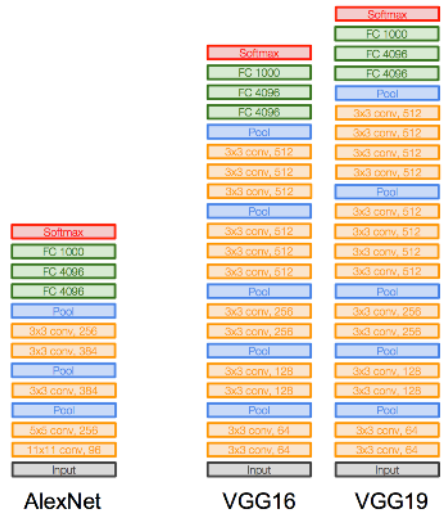
[Simonyan and Zisserman, 2014]

Small filters, Deeper networks

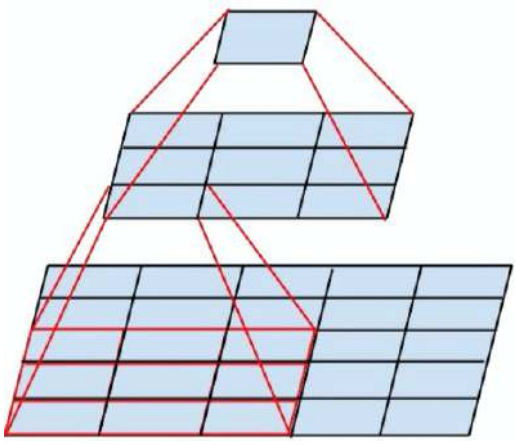
8 layers (AlexNet)
-> 16 - 19 layers (VGG16Net)

Only 3x3 CONV stride 1, pad 1
and 2x2 MAX POOL stride 2

11.7% top 5 error in ILSVRC'13
(ZFNet)
-> 7.3% top 5 error in ILSVRC'14



- Conv5x5 等价于 2 层 Conv3x3

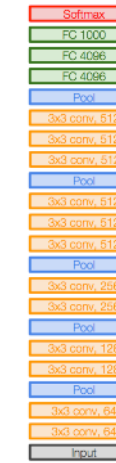


但是参数会少很多！ $2 * (3^2C^2) vs. 5^2C^2$

- VGGNet 的参数

(not counting biases)

INPUT: [224x224x3] memory: 224*224*3=150K params: 0
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456
POOL2: [56x56x128] memory: 56*56*128=400K params: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K params: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K params: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K params: 0
FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = 102,760,448
FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000

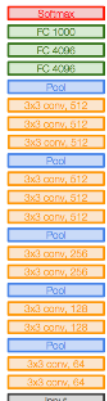


VGG16

- VGGNet 的内存占用

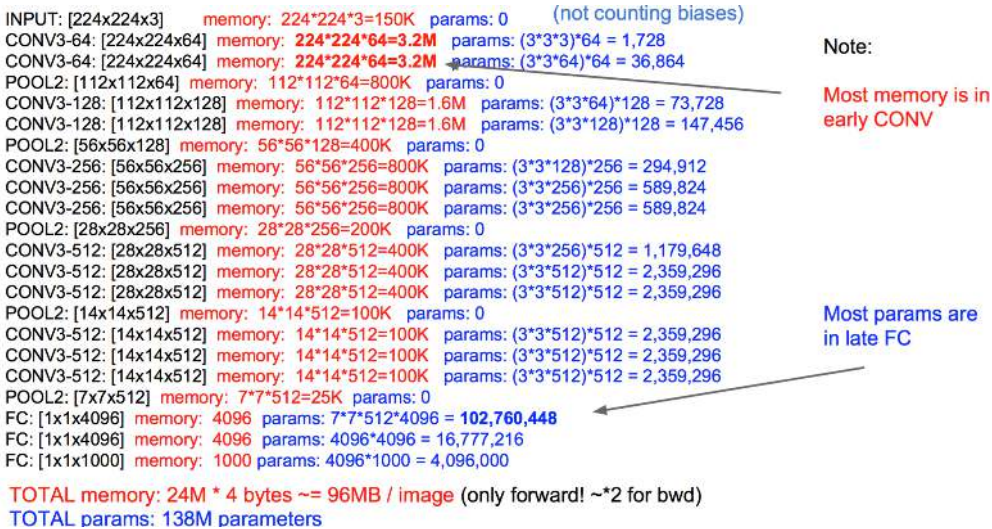
INPUT: [224x224x3] memory: 224*224*3=150K params: 0 (not counting biases)
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456
POOL2: [56x56x128] memory: 56*56*128=400K params: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K params: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K params: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K params: 0
FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = 102,760,448
FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000

TOTAL memory: 24M * 4 bytes ~= 96MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters



VGG16

- VGGNet 的卷积和 FC 层内存和参数占用对比

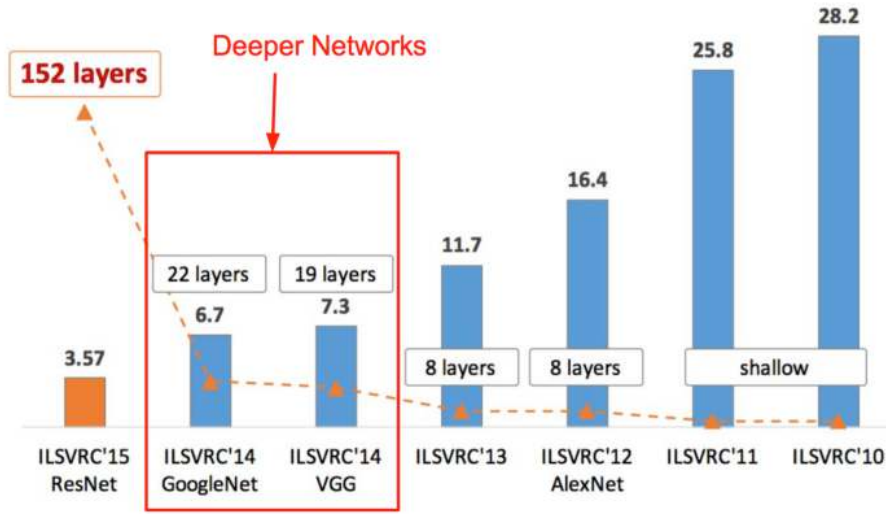


- VGGNet 的 16 和 19 的对比

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

VGG19 比 VGG16 效果稍佳，但是参数多很多。

- VGGNet 效果：ILSVRC2014 第二名

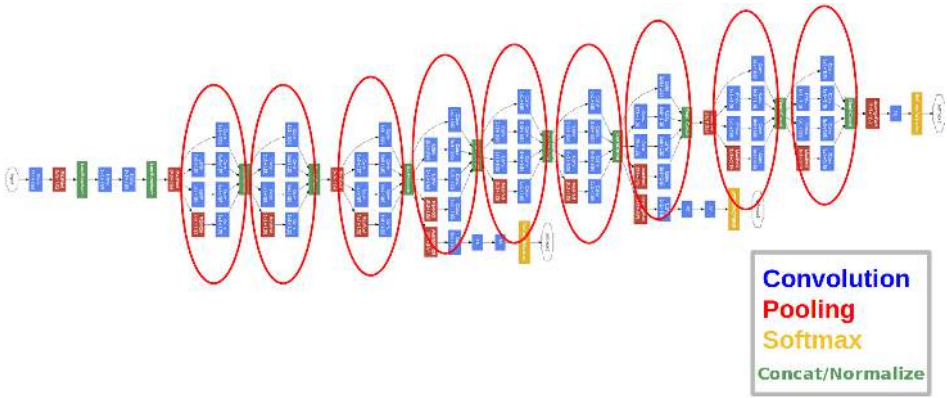


- 从 ZFNet 到 VGGNet

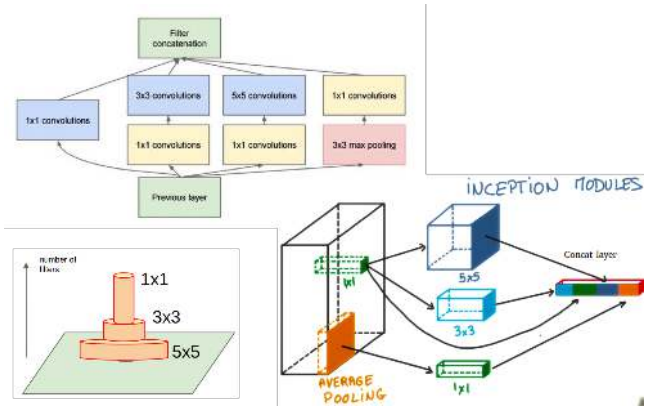
1. 层数从 8 到 16、19 层
2. 减小 Conv Filter 固定为 3x3s1p1，Max Pooling 固定为 2x2s2
3. 基本连续 2-4 层 Conv Filter，等价 (5x5, 7x7, 9x9)
4. 集成从 64 分模块叠加，128，256，512。
5. 去除了 Normalization 标准化层
6. 强化了集成的效应
7. 开启了固定模块化的先驱！
8. 使用了随机方式数据增强 scale jittering
9. 4 Nvidia Titan Black GPU 跑了 2、3 周

4.5 GoogLeNet

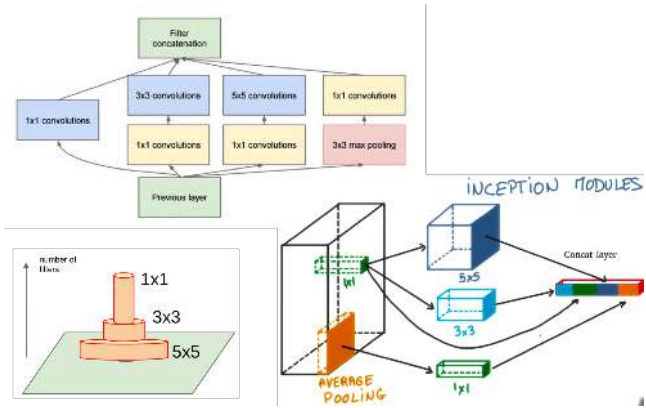
- GoogLeNet 22 层



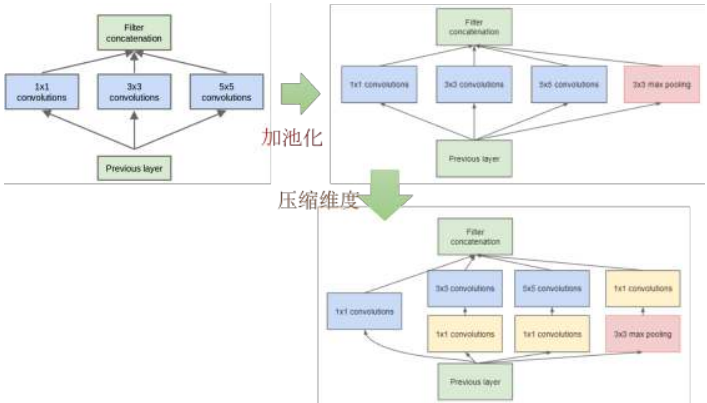
- GoogLeNet Inception 模块



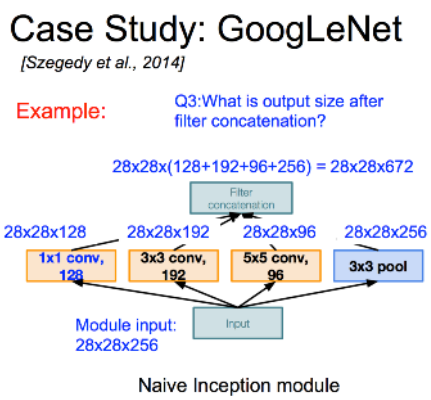
- GoogLeNet Inception 模块



- GoogLeNet Inception 模块发展



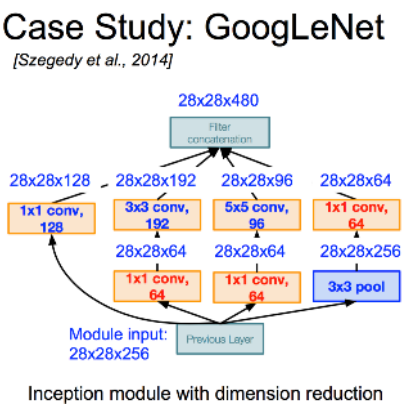
- Naive Inception 需要的计算量过大



Q: What is the problem with this?
[Hint: Computational complexity]

Conv Ops:
[1x1 conv, 128] 28x28x128x1x1x256
[3x3 conv, 192] 28x28x192x3x3x256
[5x5 conv, 96] 28x28x96x5x5x256
Total: 854M ops

- 降维后，数据量减半

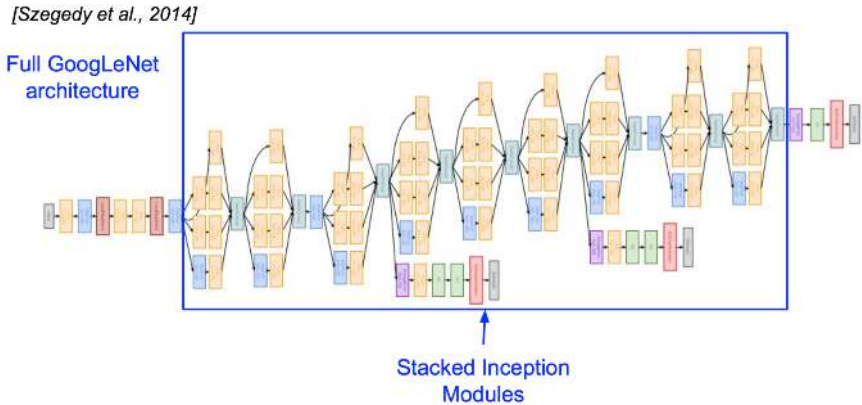


Using same parallel layers as naive example, and adding "1x1 conv, 64 filter" bottlenecks:

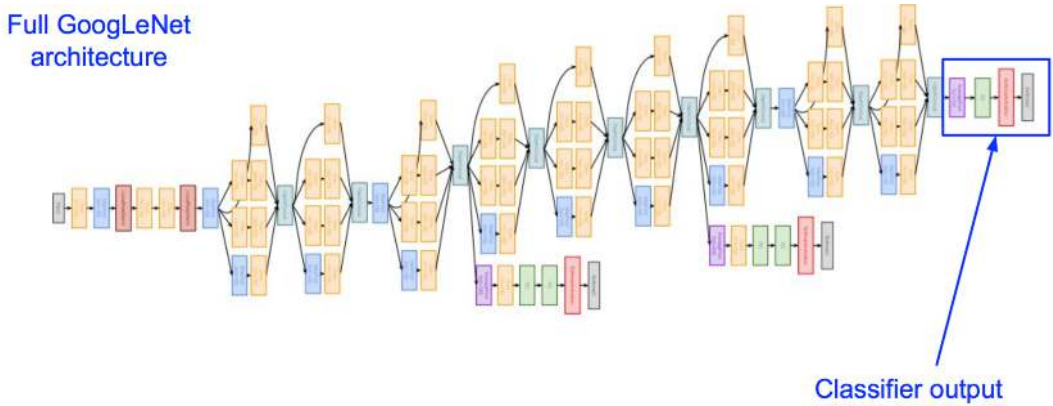
Conv Ops:
[1x1 conv, 64] 28x28x64x1x1x256
[1x1 conv, 64] 28x28x64x1x1x256
[1x1 conv, 128] 28x28x128x1x1x256
[3x3 conv, 192] 28x28x192x3x3x64
[5x5 conv, 96] 28x28x96x5x5x64
[1x1 conv, 64] 28x28x64x1x1x256
Total: 358M ops

Compared to 854M ops for naive version
Bottleneck can also reduce depth after pooling layer

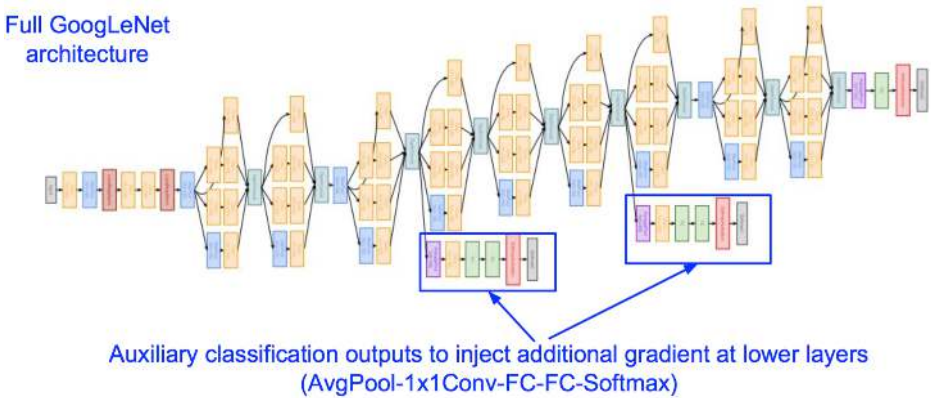
- 9 层栈式 Inception



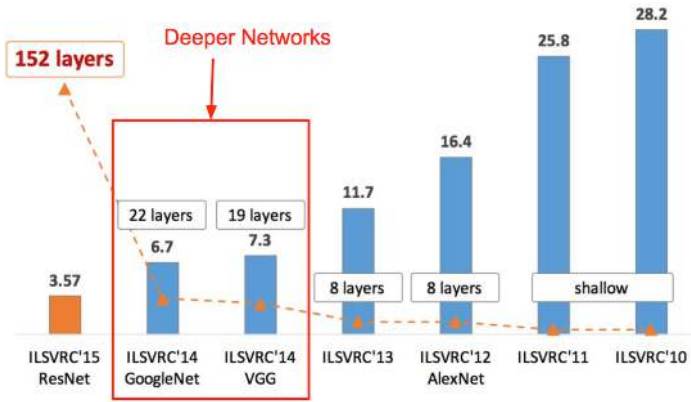
- 输出没有级联的 FC 层



- 辅助输出有 2 层级联的 FC 层: 加速梯度运算



- GoogLeNet 效果：ILSVRC2014 第一名

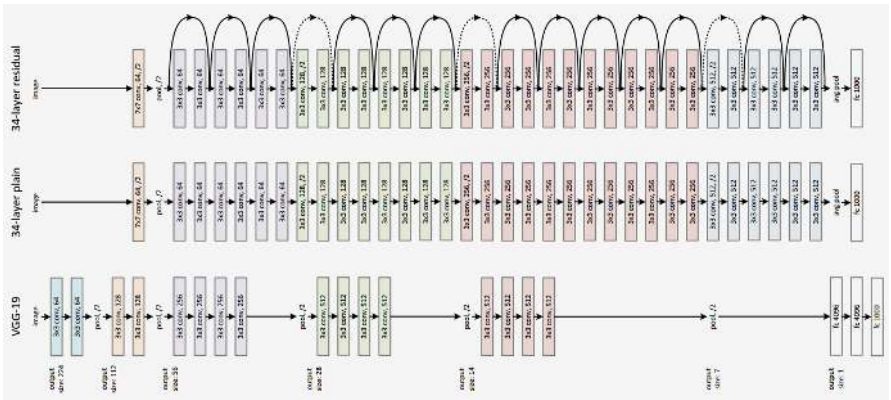


- ZFNet 到 GoogLeNet

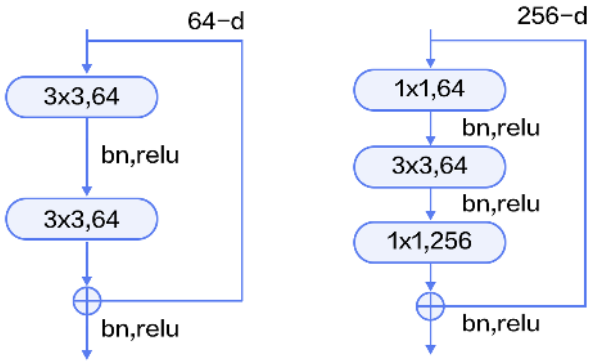
1. 引入 Inception 模块
2. 引入辅助输出模块导入梯度，加速收敛
3. 输出层没有 FC 级联
4. 22 层的 Conv 层但是，参数只有 AlexNet 的 1/12
5. 使用了 Average Pooling，而不是 Max Pooling
6. 第一次除了横向扩展外还有纵向扩展
7. 几个高端 GPU，跑了 1 周

4.6 ResNet

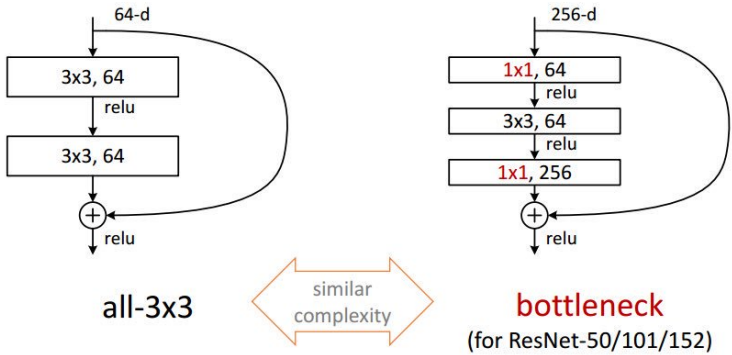
- ResNet 152 层



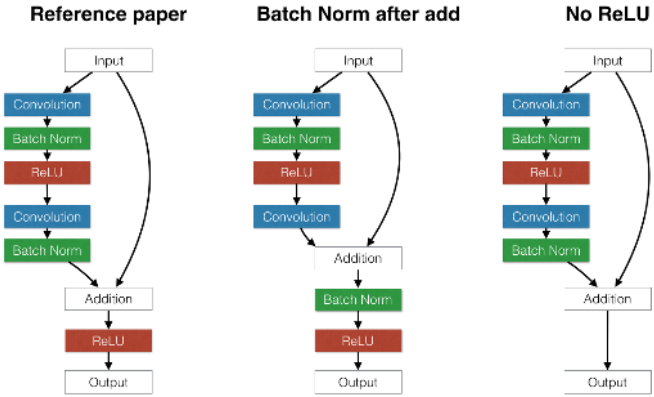
- 残差模块：Residual block



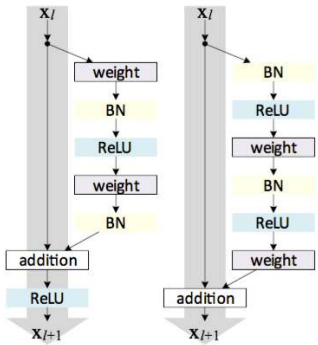
- 残差模块：bottleneck (50 层以上)



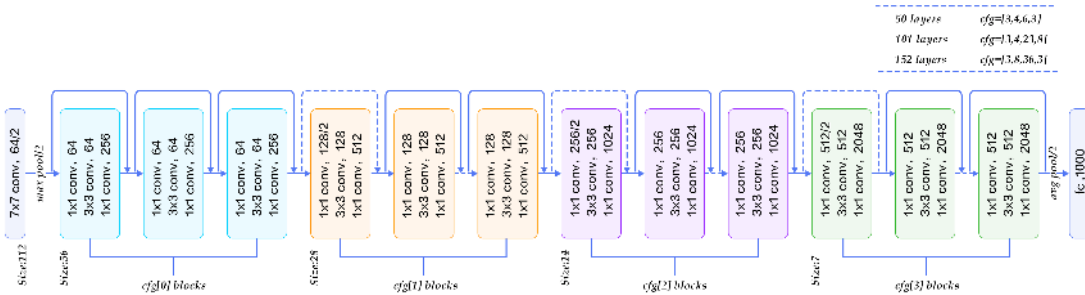
- 残差模块变种


Reference paper
Input
Convolution
Batch Norm
ReLU
Convolution
Batch Norm
Addition
ReLU
Output
Batch Norm after add
Input
Convolution
Batch Norm
ReLU
Convolution
Addition
Batch Norm
ReLU
Output
No ReLU
Input
Convolution
Batch Norm
ReLU
Convolution
Batch Norm
Addition
Output

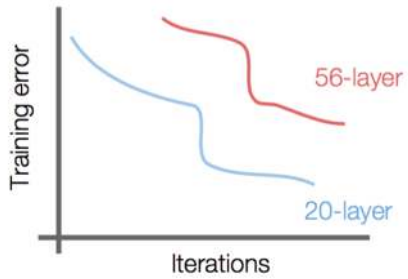
- 残差模块一种推荐

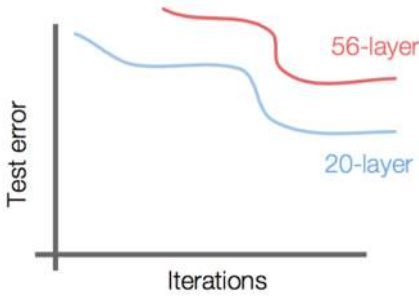

x_l
weight
BN
ReLU
weight
BN
addition
ReLU
x_{l+1}
x_l
BN
ReLU
weight
BN
ReLU
weight
addition
x_{l+1}
(a) original
(b) proposed

- ResNet 配置

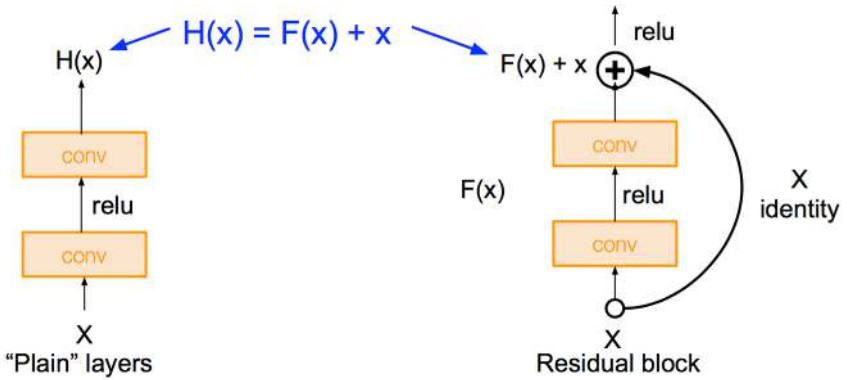

50 layers cfg=[3,4,6,3]
101 layers cfg=[3,4,23,3]
152 layers cfg=[3,8,36,3]
7x7 conv, 64/2
max pool/2
1x1 conv, 64
3x3 conv, 64
1x1 conv, 256
size/12
size/56
cfg[0] blocks
1x1 conv, 128/2
3x3 conv, 128
1x1 conv, 512
1x1 conv, 128
3x3 conv, 128
1x1 conv, 512
size/28
cfg[1] blocks
1x1 conv, 256/2
3x3 conv, 256
1x1 conv, 1024
1x1 conv, 256
3x3 conv, 256
1x1 conv, 1024
size/14
cfg[2] blocks
1x1 conv, 512/2
3x3 conv, 512
1x1 conv, 2048
1x1 conv, 512
3x3 conv, 512
1x1 conv, 2048
size/7
cfg[3] blocks
avg pool/2
fc, 1000

- ResNet 收敛的难易

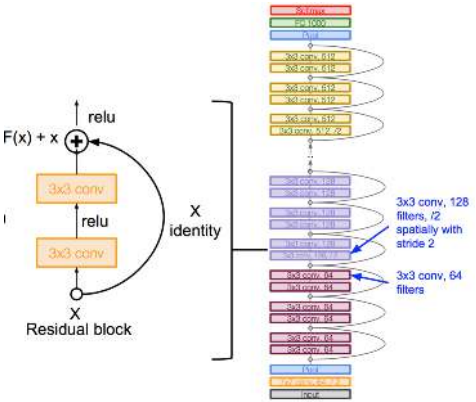




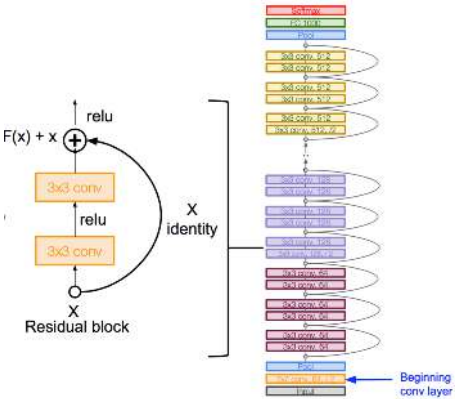
- 残差的由来


H(x) = F(x) + x
H(x)
conv
relu
conv
X
"Plain" layers
relu
F(x) + x
conv
F(x)
relu
conv
X
identity
X
Residual block
Use layers to fit residual F(x) = H(x) - x instead of H(x) directly

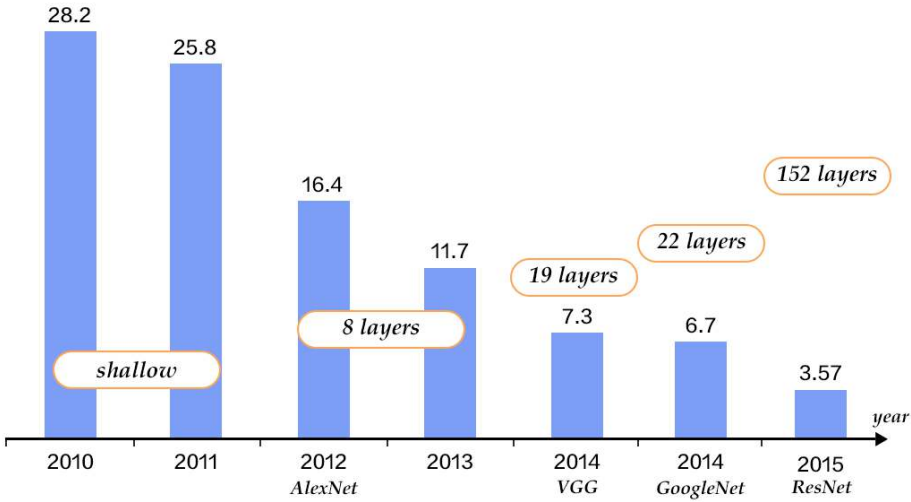
- ResNet 的 Strike



- 输入层的单一卷积层


relu
F(x) + x
3x3 conv
relu
3x3 conv
X
Residual block
X
identity
Beginning
conv layer

- ResNet 效果：ILSVRC2015 第一名，超越人类

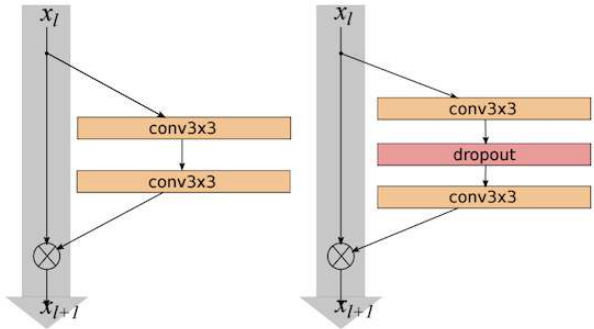


- GoogLeNet 到 ResNet

1. 超级简单残差模块

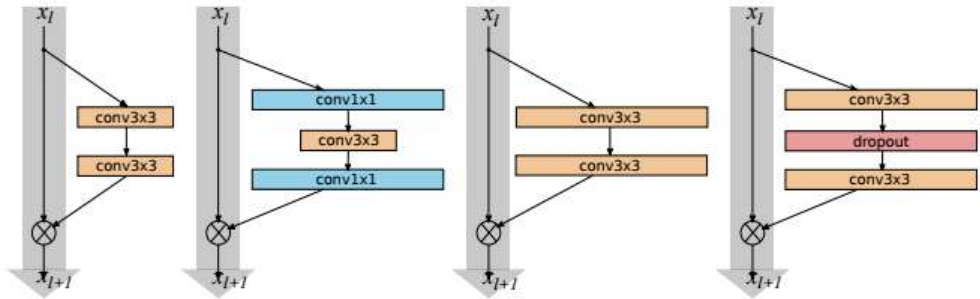
2. 超级深 Ultra-deep: 152 层
3. 继承 GoogLeNet 的维度压缩思想：bottleneck
4. 常见的有 34, 50, 101, 152 层
5. 8 个 GPU，跑了 2、3 周
6. 具体参数
    - 每个 Conv 后面可以有 BN 层
    - Xavier/2 初始化
    - 没有使用 Dropout

- Dropout 的残差模块



(c) basic-wide (d) wide-dropout

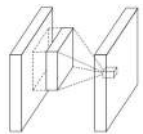
- 四种残差模块



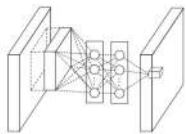
(a) basic (b) bottleneck (c) basic-wide (d) wide-dropout

4.7 NiN、改进 ResNet、超越 ResNet

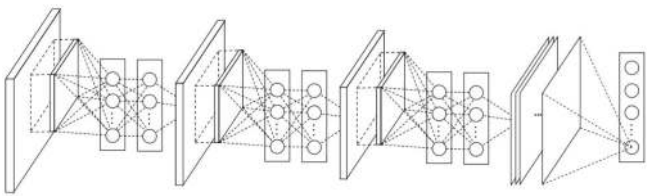
- Network in Network (NiN) [Lin et al. 2014]



(a) Linear convolution layer

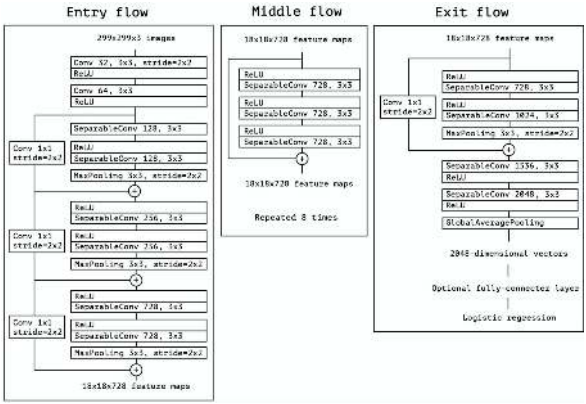


(b) Mlpconv layer

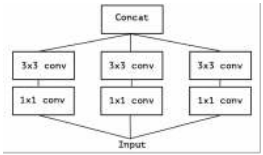


多层网络，维度压缩，是 GoogLeNet 和 ResNet bottleneck 先驱

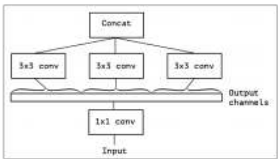
- Xception

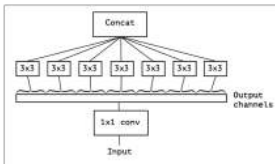

Entry flow
Middle flow
Exit flow
299x299x3 images
Conv 32, 3x3, stride=2x2
ReLU
Conv 64, 3x3
ReLU
SeparableConv 128, 3x3
Conv 1x1 stride=2x2
ReLU
SeparableConv 128, 3x3
MaxPooling 3x3, stride=2x2
ReLU
SeparableConv 256, 3x3
Conv 1x1 stride=2x2
ReLU
SeparableConv 256, 3x3
MaxPooling 3x3, stride=2x2
ReLU
SeparableConv 728, 3x3
Conv 1x1 stride=2x2
ReLU
SeparableConv 728, 3x3
MaxPooling 3x3, stride=2x2
19x19x728 feature maps
ReLU
SeparableConv 728, 3x3
ReLU
SeparableConv 728, 3x3
ReLU
SeparableConv 728, 3x3
19x19x728 feature maps
Repeated 8 times
19x19x728 feature maps
ReLU
SeparableConv 728, 3x3
Conv 1x1 stride=2x2
ReLU
SeparableConv 1024, 3x3
MaxPooling 3x3, stride=2x2
SeparableConv 1536, 3x3
ReLU
SeparableConv 2048, 3x3
ReLU
GlobalAveragePooling
2048 dimensional vectors
Optional fully-connected layer
Logistic regression

- Xception Module

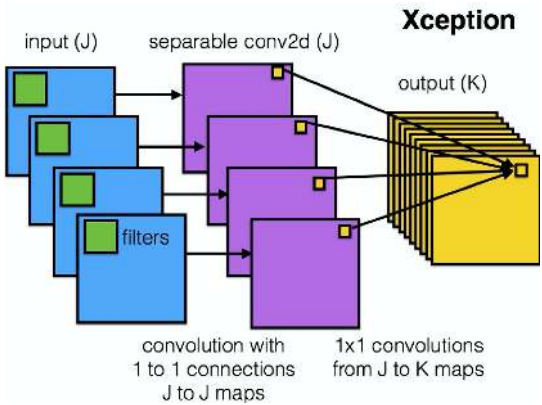


“extreme” Inception module

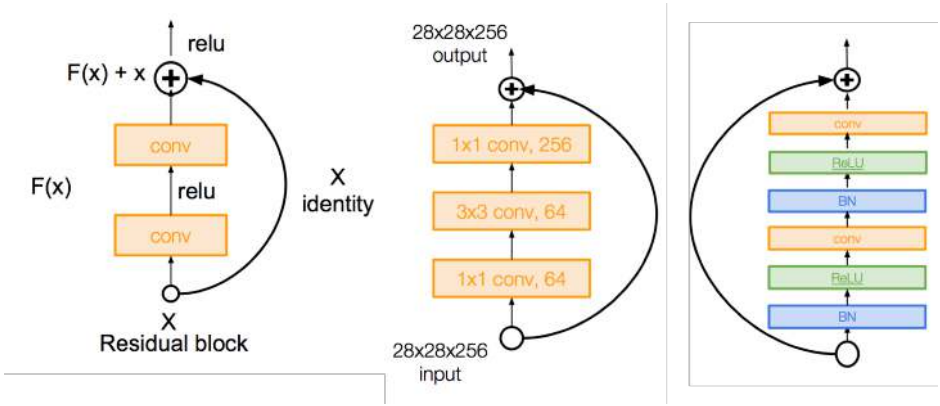




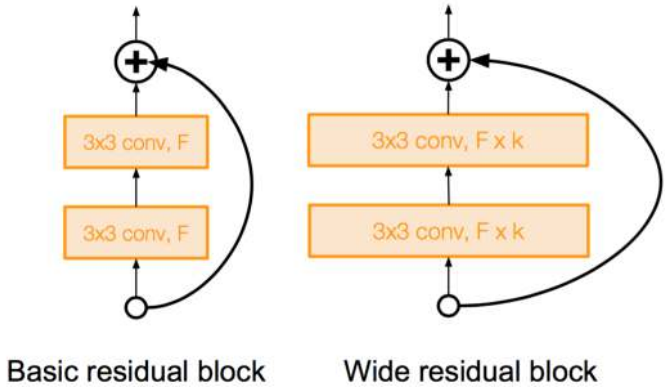
- Separable Convolution



- Deep Residual Networks

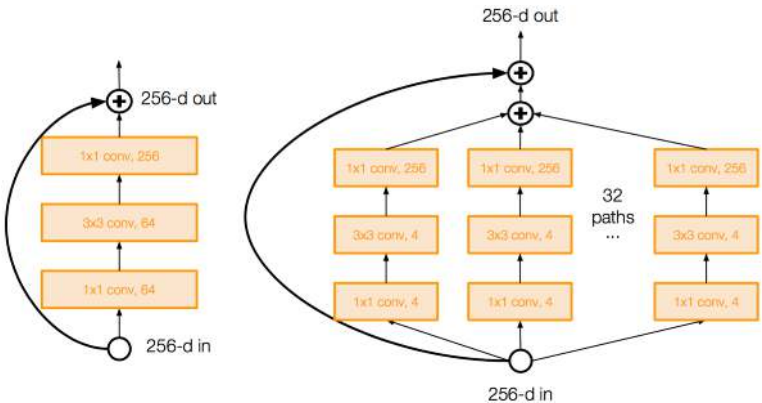


- Wide Residual Networks



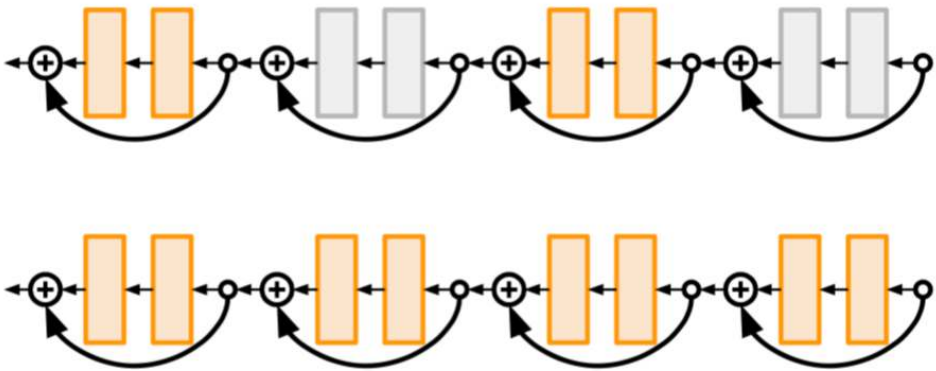
50 层的 wide ResNet 要比 152 层的原生 ResNet 效果好

- ResNeXt



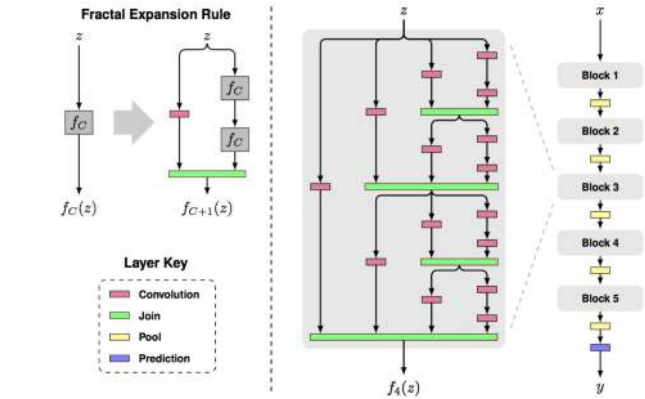
多路并行，模仿 Inception 模块

- Stochastic Depth ResNet



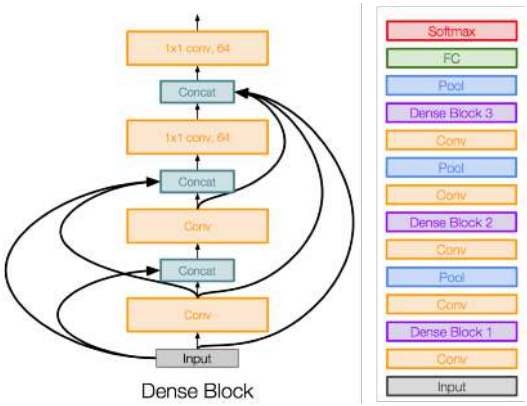
随机 Drop 部分子模块

- FractalNet：分形网络



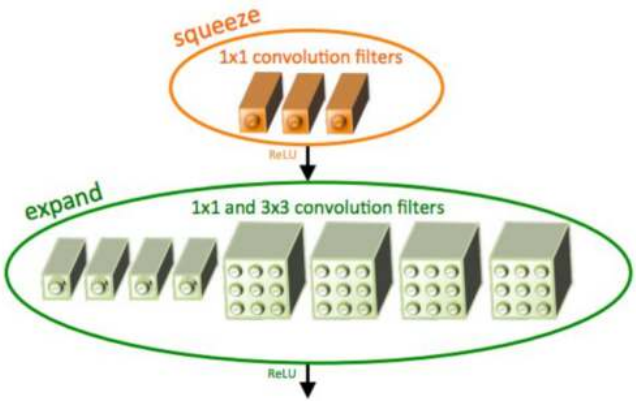
训练时候 Dropout 子路径

- Densely Connected CNN



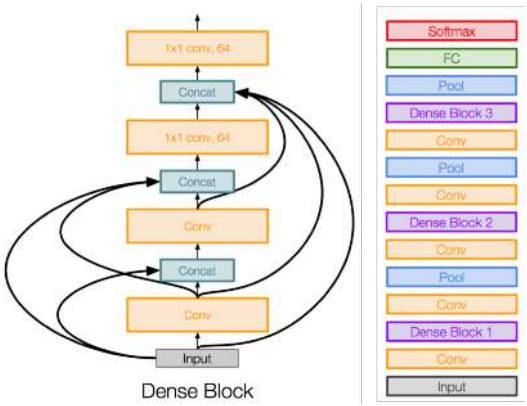
密集连接减轻梯度消失

- SqueezeNet：压缩扩展维度



同样效果，参数比 AlexNet 少 50x 倍

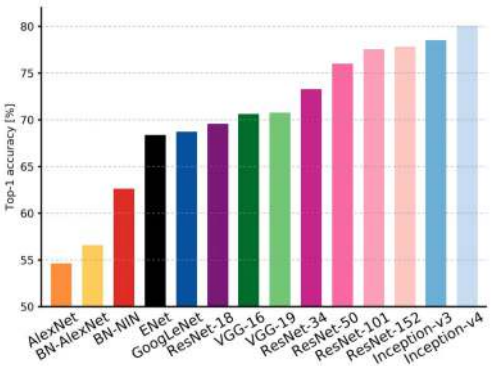
- Densely Connected CNN

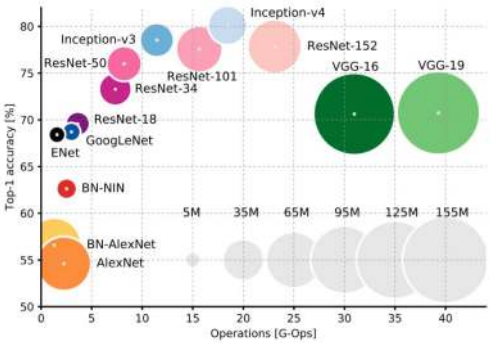


密集连接减轻梯度消失

4.8 CNN 深度网络对比

- 准确率复杂度比较



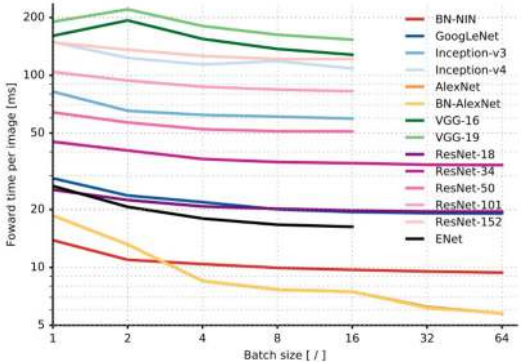


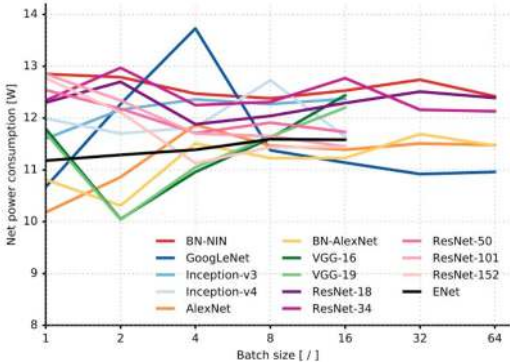
An Analysis of Deep Neural Network Models for Practical Applications, 2017.

- 准确率复杂度比较

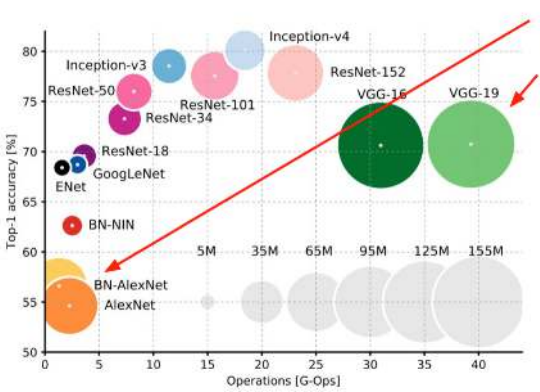
1. Inception-v4(Resnet + Inception): 最高准确率
2. VGG: 最高内存 (主要操作对象)
3. GoogLeNet: 最有效，单位内存的准确度比值高
4. AlexNet：准确率最低，但是内存占用高
5. ResNet：模型的有效性和深度有关系

- 时间和电力消耗


Foward time per image [ms]
Batch size [/]
BN-NIN
GoogLeNet
Inception-v3
Inception-v4
AlexNet
BN-AlexNet
VGG-16
VGG-19
ResNet-18
ResNet-34
ResNet-50
ResNet-101
ResNet-152
ENet


Net power consumption [W]
Batch size [/]
BN-NIN
GoogLeNet
Inception-v3
Inception-v4
AlexNet
BN-AlexNet
VGG-16
VGG-19
ResNet-18
ResNet-34
ResNet-50
ResNet-101
ResNet-152
ENet

- CNN 选择：取消 FC 的好处



AlexNet and VGG have tons of parameters in the fully connected layers

AlexNet: ~62M parameters

FC6: 256x6x6 -> 4096: 38M params
FC7: 4096 -> 4096: 17M params
FC8: 4096 -> 1000: 4M params
~59M params in FC layers!

- CNN 小结

1. VGG, GoogLeNet, ResNet 用的最多
2. 默认优先尝试 ResNet (默认效果好，扩展方式多样)
3. 越来越倾向于 eXtreamly (Ultral) Deep
4. 网络层，连接，跳转的设计
    - 网络宽带
    - 网络深度
    - 残差连接 (梯度消失对抗)
    - 维度压缩
5. 梯度可视化效果

- 下次讲解

1. 递归神经网络 RNN
2. 玻尔兹曼机 BM
3. 自动编码器 AE
4. 对抗网络 GAN

感谢 Stanford, CMU, MIT 等网上公开课程和资料！

AI2ML