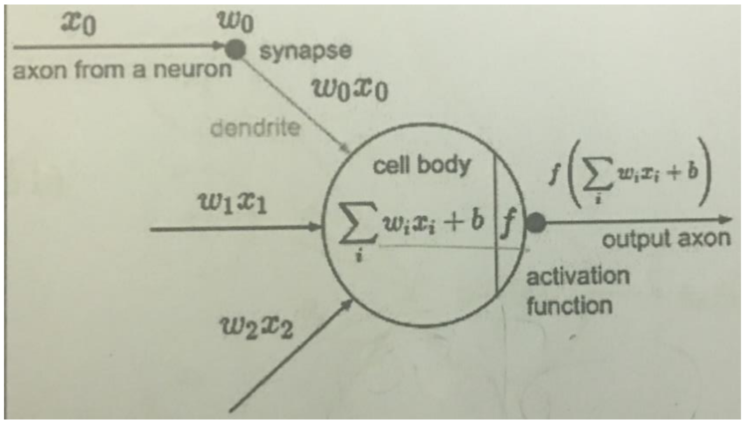
Neural Network

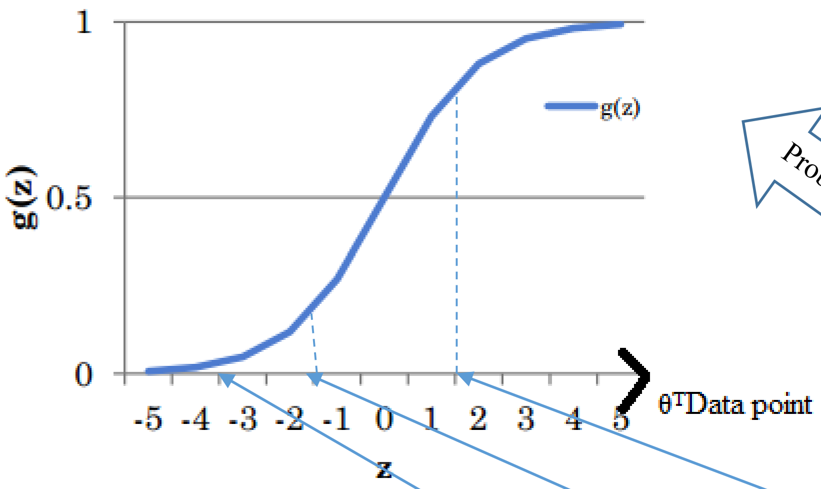
Introduction

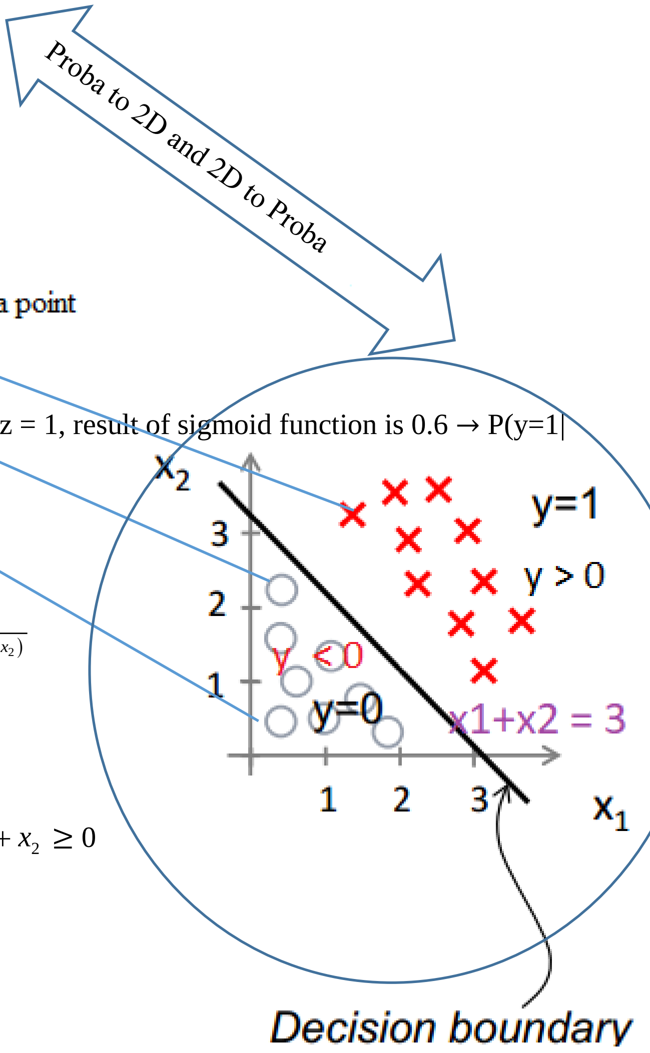


- The activation function needs to be a nonlinear function. The reason is if the activation function is linear:
  - You don't need hidden layer anymore because when sum functions go through one or more hidden layer, it's still linear one
  - Linear function cannot cover all dataset

Activation function

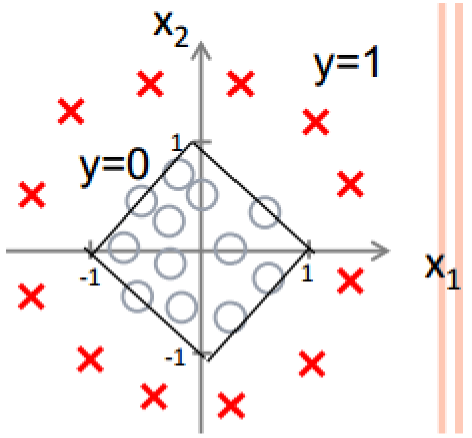
- **Sigmoid function**
  - Instead of using y = $\theta^T X$ in linear regression, you use $h_\theta(X) = \frac{1}{1+e^{-\theta^T X}}$: sigmoid function





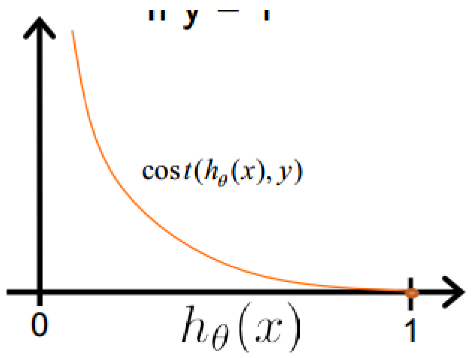
  - **Sigmoid function: the probability that y = 1 given x**. E,g. when z = 1, result of sigmoid function is 0.6 → P(y=1| x) = $h_\theta X$ = 0.6, P(y= 0| x) = 0.4 → when z = 1, g(z) = 1
    - y = 1 when $h_\theta X \geq 0.5$
      y = 0 when $h_\theta X < 0.5$
  - While $h_\theta X$ = P(y=1|x), $\theta^T X$ is the **decision boundary.** E,g.
    - $h_\theta(X) = g(\theta^T X) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2) = \frac{1}{1+e^{-(\theta_0+\theta_1 x_1+\theta_2 x_2)}}$
    - Given $\theta = \begin{bmatrix} -3 \\ 1 \\ 1 \end{bmatrix}$
    - When y = 1, $\theta^T X \geq 0$ → $\theta_0 + \theta_1 x_1 + \theta_2 x_2 \geq 0$ = -3 + $x_1 + x_2 \geq 0$
      → $x_1 + x_2 \geq 3$

        - This formula of decision boundary you need to choose. E.g, $h_\theta(X) = g(\theta^T X) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2) = \frac{1}{1+e^{-(\theta_0+\theta_1 x_1+\theta_2 x_2+\theta_3 x_1^2+\theta_4 x_2^2)}}$, next run logistics regression to determine θ and $-1 + x_1^2 + x_2^2 \geq 0$

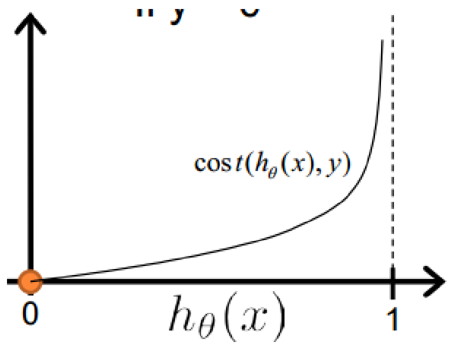


- The **loss function** computes the error for a **single training example**; the **cost function** is the **average of the loss** functions of the entire training set.
    - Cost function

$$\begin{cases} -\log(h_\theta x) & if\ y=1 \\ -\log(1-h_\theta x) & if\ y=0 \end{cases}$$



        - $$\rightarrow J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left(y_i \log(h_\theta x_i) + (1-y_i)\log(1-h_\theta x_i)\right)$$
        - It is very intuitive when analyzing the cost function
            - If y = 1 (real value is 1),
                - Cost = 0 if h_θx = 1, which means you predict right
                - Cost = ∞ if h_θx = 0, which means you predict wrong
            - If y = 0 (real value is 0),
                - Cost = 0 if h_θx = 0, which means you predict right
                - Cost = ∞ if h_θx = 1, which means you predict wrong



        - This is also advantages when compared to linear regression. In linear regression, when you predict wrong like 1 but the real values is 0, the residual sum of square just calculate this incorrect 1^2= 1, so small. However, in logistics regression, it calculates the incorrect is ∞
        - After that, you use gradient descent to optimize cost function to find θ

    - *Gradient Descent to solve Cost Function*
        - Best Way: Apply **tensorflow to solve derivative** of cost function
        - Solve Manually ():

$$\left.\begin{aligned} p(y_i = 1 | \mathrm{x}_i;\theta) = \sigma(\theta^T \mathrm{x}_i) = z_i \\ p(y_i = 0 | \mathrm{x}_i;\theta) = 1-\sigma(\theta^T \mathrm{x}_i) = 1 - z_i \end{aligned}\right\} \rightarrow p(y_i | \mathrm{x}_i;\theta) = z_i^{y_i}(1-z_i)^{1-y_i}$$

Maximum Likelihood Estimation: $\theta = \arg\max_{\theta} p(\mathrm{y} \mid \mathrm{X};\theta) = \arg\max_{\theta} \prod_{i=1}^{N} p(y_i \mid \mathrm{x}_i;\theta) = \arg\max_{\theta} \prod_{i=1}^{N} z_i^{y_i}(1-z_i)^{1-y_i}$

$$\rightarrow J(\theta) = -\frac{1}{N}\log p(\mathrm{y} \mid \mathrm{X};\theta) = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i \log z_i + (1-y_i)\log(1-z_i)\right)$$

Optimize Cost function: $\nabla_{\theta} J(y_i \mid \mathrm{x}_i;\theta) = \frac{\partial J}{\partial z}\frac{\partial z}{\partial \theta} = -\left(\frac{y_i}{z_i} - \frac{1-y_i}{1-z_i}\right)\frac{\partial z}{\partial \theta} = \frac{z_i - y_i}{z_i(1-z_i)} z_i(1-z_i)\mathrm{x}_i = (z_i - y_i)\mathrm{x}_i$

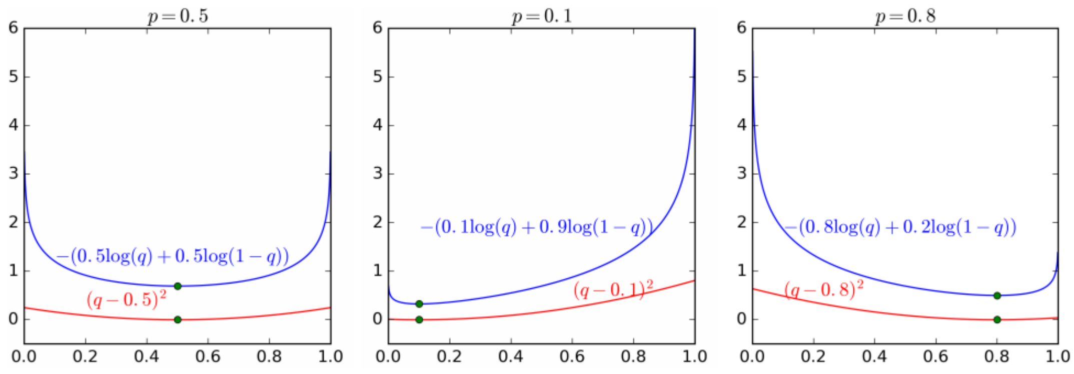
- *Regularization*
  - $$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left(y_i \log(h_\theta x_i) + (1-y_i)\log(1-h_\theta x_i)\right) + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

- **Softmax Regression**
  - Similar to Logistics Regression which classify based on the probability, but instead of classifying 2 class and 2 weights $[\boldsymbol{\theta}_1 \quad \boldsymbol{\theta}_2] = [\mathbf{0} \quad \boldsymbol{\theta}_2\text{-}\boldsymbol{\theta}_1]$, softmax regression classifies k classes and k weights
  - With dataset X(rows, feature + 1), you have **θ** = Weight Matrix(Feature + 1, K Classes)

$$\rightarrow \mathbf{X\theta} = \mathbf{X}\begin{bmatrix} | & | & | & | \\ \boldsymbol{\theta}^{(1)} & \boldsymbol{\theta}^{(2)} & \ldots & \boldsymbol{\theta}^{(K)} \\ | & | & | & | \end{bmatrix} = (rows, K\,Classes)$$

  - In each rows, find the greatest number to find which class this rows should be in
  - Actually, you need to pass the result of $\mathbf{X\theta}$ to sigmoid function to find the probability and choose the greatest probability. The sum of probability in each row = 1
  - Sigmoid Function: $y^{(i)} \in \{1,2,3,\ldots,K\}$: $\begin{cases} P\left(y^{(i)} = k \mid \mathbf{x}^{(i)};\boldsymbol{\theta}\right) = \dfrac{\exp\left(\boldsymbol{\theta}^{(k)T}\mathbf{x}^{(i)}\right)}{\sum_{j=1}^{K}\exp\left(\boldsymbol{\theta}^{(j)T}\mathbf{x}^{(i)}\right)} \\ P\left(y^{(i)} = k \mid \mathbf{x}^{(i)};\boldsymbol{\theta}\right) = \dfrac{\exp\left(\boldsymbol{\theta}^{(k)T}\mathbf{x}^{(i)} - z\right)}{\sum_{j=1}^{K}\exp\left(\boldsymbol{\theta}^{(j)T}\mathbf{x}^{(i)} - z\right)}, z = \max\left(\boldsymbol{\theta}^{(j)T}\mathbf{x}^{(i)}\right) : \text{Stable Softmax} \end{cases}$

  - Cost function
    - The target and output vector are all in probability. Cross entropy allows to measure the distance between them
    - Cross entropy between target **p** and output **q**: $-\sum_{i=1}^{K} p_i \log q_i$
    - The figure below: the red line is square distance and blue line is cross entropy distance between **p** and **q**



    - In 2 methods, the cost function is minimum when **p** = **q**, but in last 2 figures, when **p** ≠ **q**, the cost of cross entropy is much more than the square distance
    - Cost function: $J(\theta) = -\frac{1}{N}\sum_{i=1}^{N} y_{ij}\log a_{ij} \in \mathbb{R}$

    - And add regularization: $J(\theta) = -\frac{1}{N}\left(\sum_{i=1}^{N}\sum_{j=1}^{K} y_{ij}\log a_{ij} + \frac{\lambda}{2}\|\theta\|_F^2\right)$

- Optimize the Cost function
    - Assume SGD is applied

$$J_i(\theta) = -\sum_{j=1}^{K} y_{ij}\log a_{ij} = -\sum_{j=1}^{K} y_{ij}\log\frac{\exp\left(\theta^{(j)T}\mathrm{x}^{(i)}\right)}{\sum_{k=1}^{K}\exp\left(\theta^{(k)T}\mathrm{x}^{(i)}\right)} = -\sum_{j=1}^{K}\left(y_{ij}\theta^{(j)T}\mathrm{x}^{(i)} - y_{ij}\log\sum_{k=1}^{K}\exp\left(\theta^{(k)T}\mathrm{x}^{(i)}\right)\right)$$

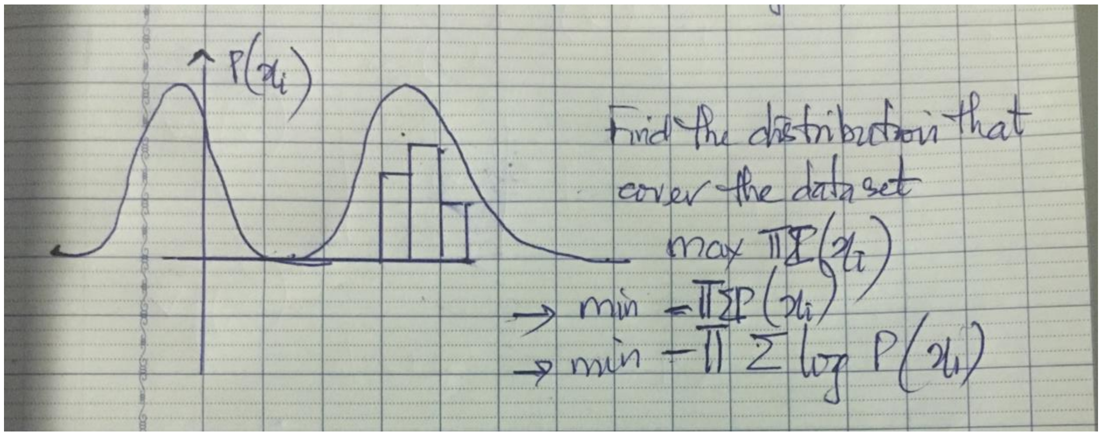
$$= \log\sum_{k=1}^{K}\exp\left(\theta^{(k)T}\mathrm{x}^{(i)}\right) - \sum_{j=1}^{K} y_{ij}\theta^{(j)T}\mathrm{x}^{(i)}$$

and $\nabla_\theta J_i(\theta) = \left[\nabla_{\theta^{(1)}}J_i(\theta) \quad \nabla_{\theta^{(2)}}J_i(\theta) \quad \ldots \quad \nabla_{\theta^{(K)}}J_i(\theta)\right]$

$$\nabla_{\theta^{(j)}}J_i(\theta) = \frac{\exp\left(\theta^{(j)T}\mathrm{x}^{(i)}\right)}{\sum_{k=1}^{K}\exp\left(\theta^{(k)T}\mathrm{x}^{(i)}\right)}\mathrm{x}^{(i)} - y_{ij}\mathrm{x}^{(i)} = a_{ij}\mathrm{x}^{(i)} - y_{ij}\mathrm{x}^{(i)} = \mathrm{x}^{(i)}\left(a_{ij} - y_{ij}\right)$$

$$\rightarrow \nabla_\theta J(\theta) = \frac{1}{N}\nabla_\theta J_i(\theta) \in \mathbb{R}^{Feature\times K}$$

    - Derivation of cost function in logistics and softmax regression



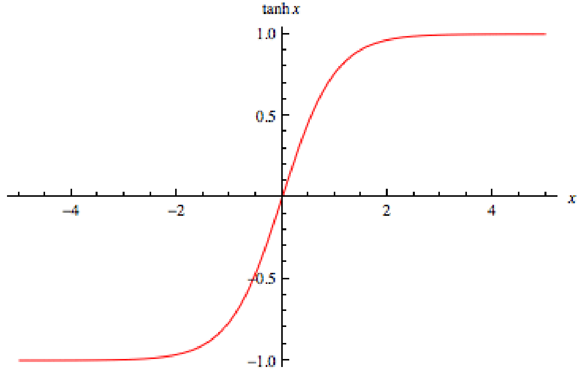
    - $1\{A\} = \begin{cases} 0 & A\,is\,False \\ 1 & A\,is\,True \end{cases}$
    - $J(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} 1\left\{y^{(i)} = k\right\} P\left(y^{(i)} = k \mid \mathbf{x}^{(i)};\boldsymbol{\theta}\right) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} 1\left\{y^{(i)} = k\right\}\log\frac{\exp\left(\boldsymbol{\theta}^{(k)T}\mathbf{x}^{(i)}\right)}{\sum_{j=1}^{K}\exp\left(\boldsymbol{\theta}^{(j)T}\mathbf{x}^{(i)}\right)}$: Cross Entropy/ KL Divergence
    - $\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}\left(1\left\{y^{(i)} = k\right\} - P\left(y^{(i)} = k \mid \mathbf{x}^{(i)};\boldsymbol{\theta}\right)\right)\mathbf{x}^{(i)}$

When k = 2, it's logistics regression and $h_{\boldsymbol{\theta}}(x_i) = h_{\boldsymbol{\theta}_2 - \boldsymbol{\theta}_1}(x_i)$,

$$\rightarrow J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left(y_i\log\left(h_\theta x_i\right) + \left(1 - y_i\right)\log\left(1 - h_\theta x_i\right)\right)$$

- Tanh function

$$\begin{cases} f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1 \\ f'(x) = 1 - f^2(x) \end{cases}$$

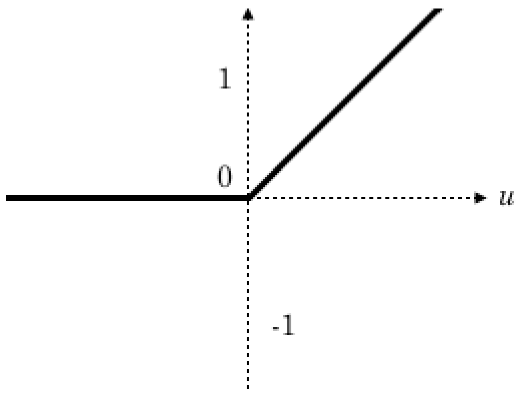


- ReLU function

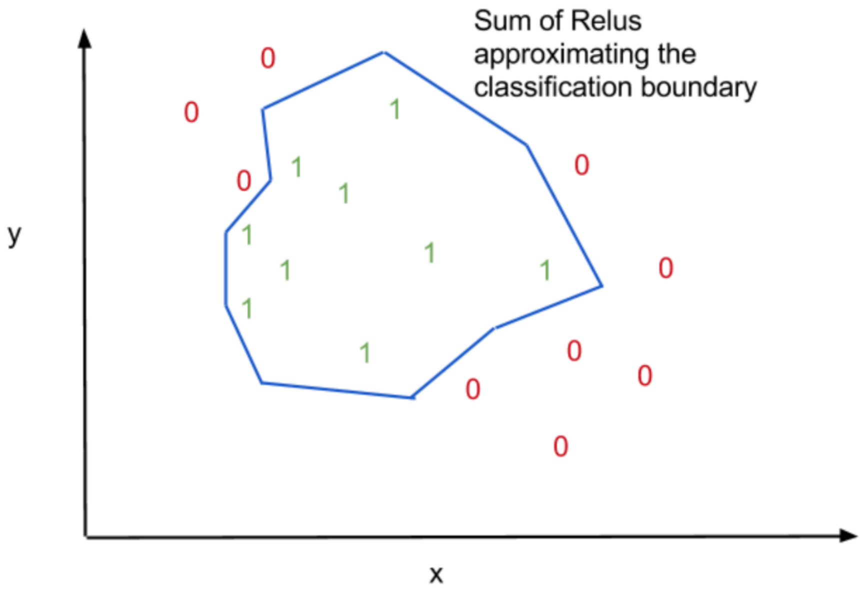
$$f(x)=\begin{cases}0 & for\ x<0\\ x & for\ x\geq 0\end{cases}$$

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

$$f(u)=\max(0,u)$$



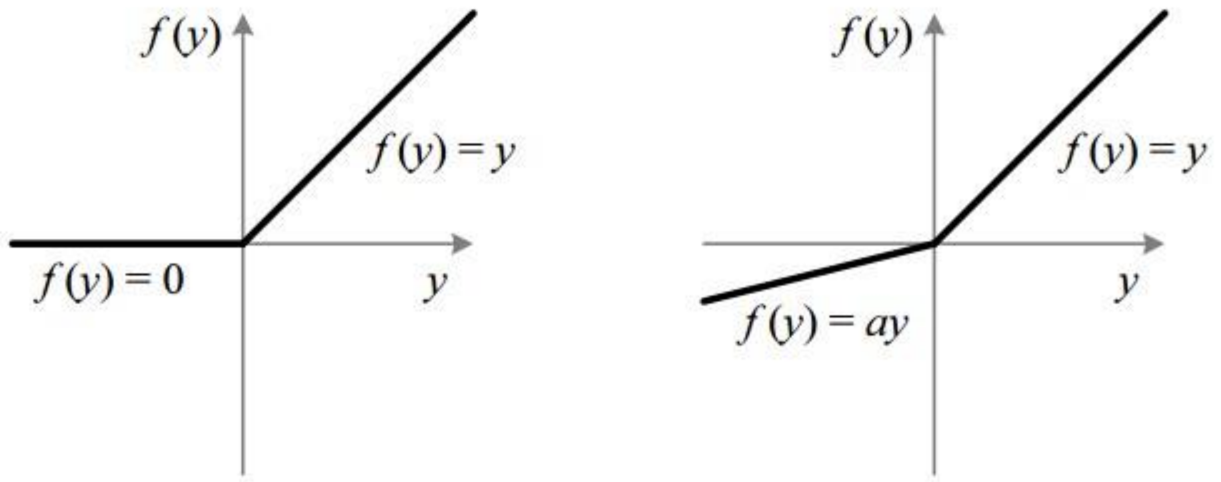
When you classify with activation function of ReLU function



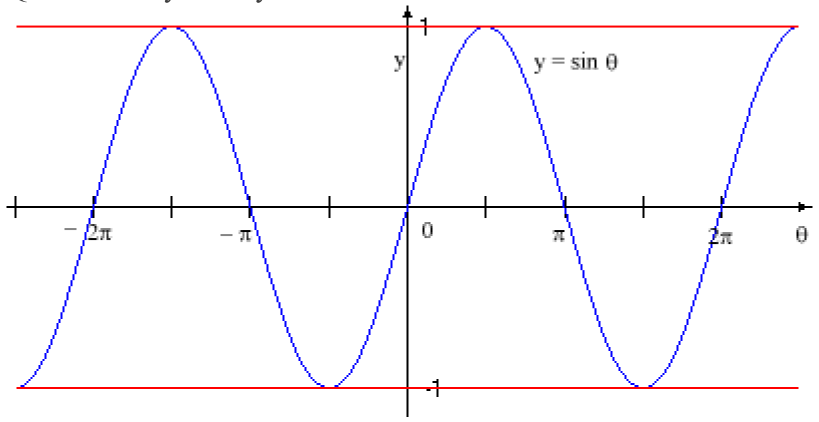
- **ReLU** is more **popular** than **Sigmoid** in activation function because when $x\to\infty, f'(x)\to 0$: **Gradient Vanishing** , so when backpropagation, θ easily go to 0
- However, ReLU also has its weakness when θ < 0 → $\nabla_\theta J=0$, the backpropagation will stop, Leaky/Dying ReLU will solve

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

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

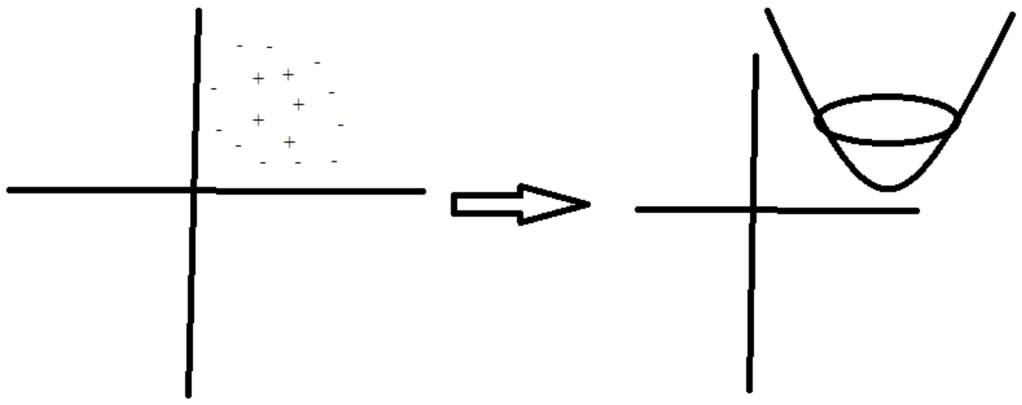


- Question: why don't you use another activation function like sine function?



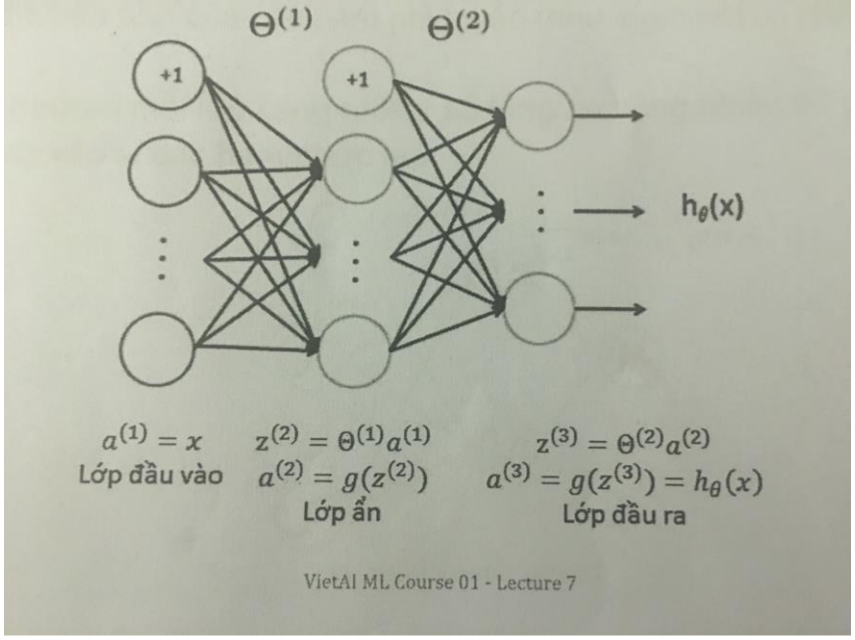
Because sine function is not asymptotic any axis or line, so when taking derivative to find optimal θ, it will not stop

- Feature Engineering in NN/ Distributed Representation



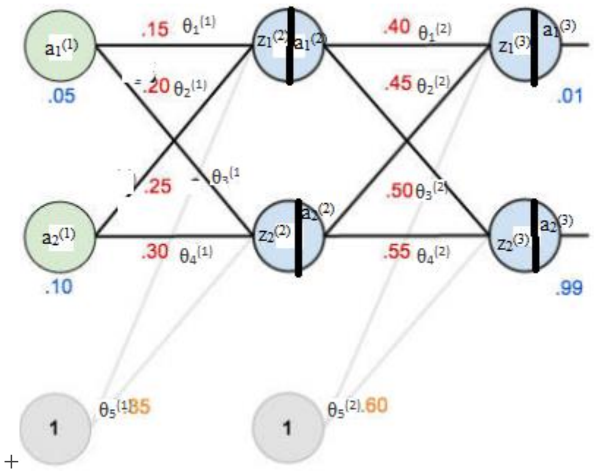
- When you classify first plot, you need to transform them to 3D like second plot to find hyperplane.
- In this case, you can increase number of node in hidden layer compared to number of feature
- In other cases, when you do feature selection, you can decrease increase number of node in hidden layer compared to number of feature

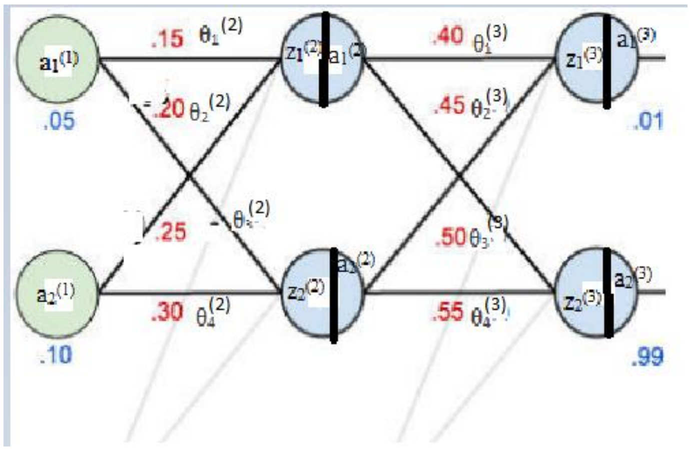
- Symbol


$\Theta^{(1)}$
$\Theta^{(2)}$
+1
+1
$h_\theta(x)$
$a^{(1)} = x$
Lớp đầu vào
$z^{(2)} = \Theta^{(1)}a^{(1)}$
$a^{(2)} = g(z^{(2)})$
Lớp ẩn
$z^{(3)} = \Theta^{(2)}a^{(2)}$
$a^{(3)} = g(z^{(3)}) = h_\theta(x)$
Lớp đầu ra
VietAI ML Course 01 - Lecture 7

Feed Forward and Back Propagation

- In order to have some numbers to work with, here are the **initial weights, the biases, and training inputs/outputs**





+

Assignment 2

- Step 1: The forward pass

$$\begin{cases} z_1^{(2)} = \theta_1^{(1)}a_1^{(1)} + \theta_2^{(1)}a_2^{(1)} + \theta_5^{(1)} = 0.15*0.05 + 0.2*0.1 = 0.3775 \\ a_1^{(2)} = \dfrac{1}{1+e^{-z_1^{(2)}}} = \dfrac{1}{1+e^{-0.3775}} = 0.592 \end{cases}$$

$$\begin{cases} z_2^{(2)} = \theta_3^{(1)}a_1^{(1)} + \theta_4^{(1)}a_2^{(1)} + \theta_5^{(1)} \\ a_2^{(2)} = \dfrac{1}{1+e^{-z_2^{(2)}}} = 0.596 \end{cases}$$

$$\begin{cases} z_1^{(3)} = \theta_1^{(2)}a_1^{(2)} + \theta_2^{(2)}a_2^{(2)} + \theta_5^{(2)} \\ a_1^{(3)} = \dfrac{1}{1+e^{-z_1^{(3)}}} = 0.751 \end{cases}$$

$$\begin{cases} z_2^{(3)} = \theta_3^{(2)}a_1^{(2)} + \theta_4^{(2)}a_2^{(2)} + \theta_5^{(2)} \\ a_2^{(3)} = \dfrac{1}{1+e^{-z_2^{(3)}}} = 0.772 \end{cases}$$

- Step 2: Calculating Cost function

$$J\left(\mathbf{a}^{(3)}\right)=\sum\frac{1}{2}\left(target-a^{(3)}\right)^2=\frac{1}{2}\left(target-a_1^{(3)}\right)^2+\frac{1}{2}\left(target-a_2^{(3)}\right)^2=\frac{1}{2}(0.01-0.751)^2+\frac{1}{2}(0.01-0.772)^2=0.298$$

- Step 3: The Back Propagation
  - Now, you need to calculate $\frac{\partial J}{\partial\theta_1^{(2)}}$ to update $\theta_1^{(2)}$ by gradient descent
  - $$d\theta^{(2)}=\frac{\partial J}{\partial\theta_1^{(2)}}=\underbrace{\frac{\partial J}{\partial a_1^{(3)}}\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}}}_{dz^{(1)}}\frac{\partial z_1^{(3)}}{\partial\theta_1^{(2)}}$$
    - $da^{(3)}=\frac{\partial J}{\partial a_1^{(3)}}=-\frac{1}{2}2\left(target-a^{(3)}\right)=-(0.01-0.751)=0.74$
    - $a_1^{(3)}=\frac{1}{1+e^{-z_1^{(3)}}}\rightarrow\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}}=a_1^{(3)}\left(1-a_1^{(3)}\right)=0.75(1-0.75)=0.186$
    - $z_1^{(3)}=\theta_1^{(2)}a_1^{(2)}+\theta_2^{(2)}a_2^{(2)}+\theta_5^{(2)}\rightarrow\frac{\partial z_1^{(3)}}{\partial\theta_1^{(2)}}=a_1^{(2)}=0.592$

$$\rightarrow\frac{\partial J}{\partial\theta_1^{(2)}}=\frac{\partial J}{\partial a_1^{(3)}}\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}}\frac{\partial z_1^{(3)}}{\partial\theta_1^{(2)}}=0.74*0.186*0.592=0.082$$

$$\rightarrow\theta_1^{(2)}=\theta_1^{(2)}-\frac{\partial J}{\partial\theta_1^{(2)}}$$

  - Similarly, you can calculate all $\boldsymbol{\theta}^{(2)}$
  - However, When calculating the $\boldsymbol{\theta}^{(1)}$, you need more calculation
  - $$\frac{\partial J}{\partial\theta_1^{(1)}}=\frac{\partial J}{\partial a_1^{(2)}}\frac{\partial a_1^{(2)}}{\partial z_1^{(2)}}\frac{\partial z_1^{(2)}}{\partial\theta_1^{(1)}}$$
    - $\frac{\partial J}{\partial a_1^{(2)}}=\frac{\partial J\left(a_1^{(3)}\right)}{\partial a_1^{(2)}}+\frac{\partial J\left(a_2^{(3)}\right)}{\partial a_1^{(2)}}=\left(\frac{\partial J\left(a_1^{(3)}\right)}{\partial z_1^{(3)}}\frac{\partial z_1^{(3)}}{\partial a_1^{(2)}}+\frac{\partial J\left(a_2^{(3)}\right)}{\partial z_2^{(3)}}\frac{\partial z_2^{(3)}}{\partial a_1^{(2)}}\right)=(0.13*0.4+(-0.019))=0.0363$
    - $a_1^{(2)}=\frac{1}{1+e^{-z_1^{(2)}}}\rightarrow\frac{\partial a_1^{(2)}}{\partial z_1^{(2)}}=a_1^{(2)}\left(1-a_1^{(2)}\right)=0.59(1-0.759)=0.241$
    - $z_1^{(2)}=\theta_1^{(1)}a_1^{(1)}+\theta_2^{(1)}a_2^{(1)}+\theta_5^{(1)}\rightarrow\frac{\partial z_1^{(2)}}{\partial\theta_1^{(1)}}=a_1^{(1)}=0.05$

$$\rightarrow\frac{\partial J}{\partial\theta_1^{(1)}}=\frac{\partial J}{\partial a_1^{(2)}}\frac{\partial a_1^{(2)}}{\partial z_1^{(2)}}\frac{\partial z_1^{(2)}}{\partial\theta_1^{(1)}}=0.0363*0.241*0.05=0.0004$$

$$\rightarrow\theta_1^{(1)}=\theta_1^{(1)}-\frac{\partial J}{\partial\theta_1^{(1)}}$$

Feed-forward:

$$\mathrm{a}^{(0)}=\mathrm{x}$$

$$\mathrm{z}^{(l)}=\mathrm{a}^{(l-1)}\theta^{(l)}+\mathrm{b}^{(l)},\ \theta^{(l)}\in\mathbb{R}^{d^{l-1}\times d^l},\ \mathrm{b}^{(l)}\in\mathbb{R}^{d^l}$$

$$\mathrm{a}^{(l)}=f\left(\mathrm{z}^{(l)}\right)$$

$$\hat{y}=a^{(L)}$$

Back-propagation:

$$\text{At output layer: }\begin{cases}\mathrm{e}^{(L)}=\nabla_{\mathrm{z}^{(L)}}J\in\mathbb{R}^{d^{(L)}}\\ \nabla_{\theta^{(L)}}J=a^{(L-1)T}\mathrm{e}^{(L)}\in\mathbb{R}^{d^{(L-1)}\times d^{(L)}}\\ \nabla_{\mathrm{b}^{(L)}}J=\mathrm{e}^{(L)}\in\mathbb{R}^{d^{(L)}}\end{cases}$$

At layer $l = L-1, L-2, \ldots, 1$:

$$\begin{cases} \mathrm{e}^{(l)} = \left(\mathrm{e}^{(l+1)}\theta^{(l+1)T}\right) \odot f'\left(\mathrm{z}^{(L)}\right) \in \mathbb{R}^{d^{(l)}} \\ \left| \begin{array}{l} \nabla_{\theta^{(l)}} J = a^{(l-1)T}\mathrm{e}^{(l)} \in \mathbb{R}^{d^{(l-1)} \times d^{(l)}} \\ \nabla_{\mathrm{b}^{(l)}} J = \mathrm{e}^{(l)} \end{array} \right. \end{cases}$$

$$A^{[0]} = X \in \mathbb{R}^{m \times d^0}$$
$$Z^{[l]} = A^{[l-1]}\boldsymbol{\theta}^{[l]} + b^{[l]} \in \mathbb{R}^{m \times d^l} s, \boldsymbol{\theta}^{[l]} \in \mathbb{R}^{d^{l-1} \times d^l}, b^{[l]} \in \mathbb{R}^{d^l}$$
$$A^{[l]} = f\left(Z^{[l]}\right)$$
$$\hat{Y} = A^{[L]}$$

At output layer:

$$\begin{cases} \mathrm{E}^{[L]} = \nabla_{Z^{[L]}} J \in \mathbb{R}^{m \times d^L} \\ \nabla_{\theta^{[L]}} J = A^{[L-1]T}\mathrm{E}^{[L]} \in \mathbb{R}^{d^{L-1} \times d^L}, A^{[L-1]} \in \mathbb{R}^{m \times d^{L-1}} \\ \nabla_{\mathrm{b}^{[L]}} J = \mathrm{E}^{[L]} \in \mathbb{R}^{d^L} \end{cases}$$

At layer $l = L-1, L-2, \ldots, 1$:

$$\begin{cases} \mathrm{E}^{[l]} = \left(\mathrm{E}^{[l+1]}\theta^{[l+1]T}\right) \odot f'\left(\mathrm{Z}^{[l]}\right) \in \mathbb{R}^{m \times d^l}, \mathrm{E}^{[l+1]} \in \mathbb{R}^{m \times d^{l+1}} \\ \qquad , \theta^{[l+1]} \in \mathbb{R}^{d^l \times d^{l+1}} \\ \left| \begin{array}{l} \nabla_{\theta^{[l]}} J = A^{[l-1]T}\mathrm{E}^{[l]} \in \mathbb{R}^{d^{l-1} \times d^l}, A^{[l-1]} \in \mathbb{R}^{m \times d^{l-1}} \\ \nabla_{\mathrm{b}^{[l]}} J = \mathrm{E}^{[l]} \in \mathbb{R}^{d^l} \end{array} \right. \end{cases}$$

$$A^{[0]} = X \in \mathbb{R}^{d^0 \times m}$$
$$Z^{[l]} = \boldsymbol{\theta}^{[l]}A^{[l-1]} + b^{[l]} \in \mathbb{R}^{d^l \times m} s, \boldsymbol{\theta}^{[l]} \in \mathbb{R}^{d^l \times d^{l-1}}, b^{[l]} \in \mathbb{R}^{d^l}$$
$$A^{[l]} = f\left(Z^{[l]}\right)$$
$$\hat{Y} = A^{[L]}$$

At output layer:

$$\begin{cases} \mathrm{E}^{[L]} = \nabla_{Z^{[L]}} J \in \mathbb{R}^{d^L \times m} \\ \nabla_{\theta^{[L]}} J = \mathrm{E}^{[L]}A^{[L-1]T} \in \mathbb{R}^{d^L \times d^{L-1}}, A^{[L-1]} \in \mathbb{R}^{d^{L-1} \times m} \\ \nabla_{\mathrm{b}^{[L]}} J = \mathrm{E}^{[L]} \in \mathbb{R}^{d^L} \end{cases}$$

At layer $l = L-1, L-2, \ldots, 1$:

$$\begin{cases} \mathrm{E}^{[l]} = \left(\theta^{[l+1]T}\mathrm{E}^{[l+1]}\right) \odot f'\left(\mathrm{Z}^{[l]}\right) \in \mathbb{R}^{d^l \times m}, \mathrm{E}^{[l+1]} \in \mathbb{R}^{d^{l+1} \times m} \\ \qquad , \theta^{[l+1]} \in \mathbb{R}^{d^{l+1} \times d^l} \\ \left| \begin{array}{l} \nabla_{\theta^{[l]}} J = \mathrm{E}^{[l]}A^{[l-1]T} \in \mathbb{R}^{d^l \times d^{l-1}}, A^{[l-1]} \in \mathbb{R}^{d^{l-1} \times m} \\ \nabla_{\mathrm{b}^{[l]}} J = \mathrm{E}^{[l]} \in \mathbb{R}^{d^l} \end{array} \right. \end{cases}$$

Initialization

- Dataset is always separated into 3 types: training set, dev set (Hold-out cross validation/ Development set/ Public test set), Test set (Private test set)
- For **small dataset** (100/ 1000/ 10000): Training set and Dev set are usually **70/ 30**
- For **large dataset** (> 1.000.000): Training set and Dev set are usually **98/ 1**
- Not having test set might be fine (Only Dev set)

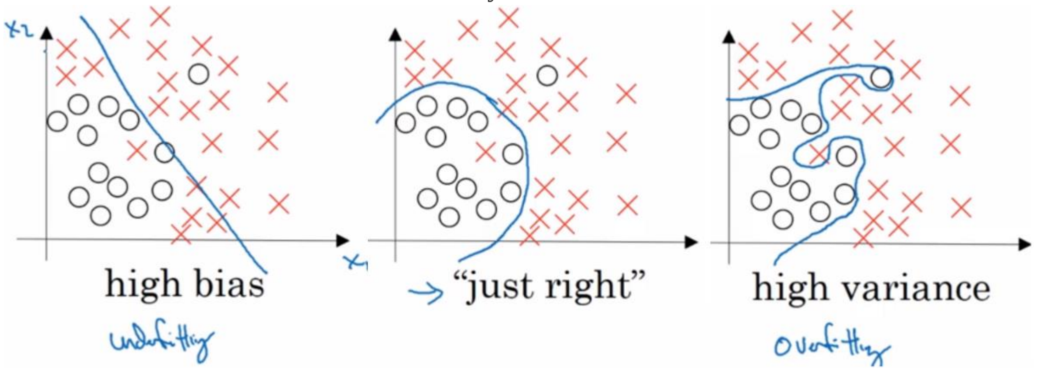
- Dev set and Test set should come from same distribution. For example, It's OK if in Training set, Cat pictures from the webpages, in Dev/ Test set, Cat pictures from users using your app
- There are 3 popular initialization types (which can solve **Vanishing/ Exploding gradients**):
    - He initialization

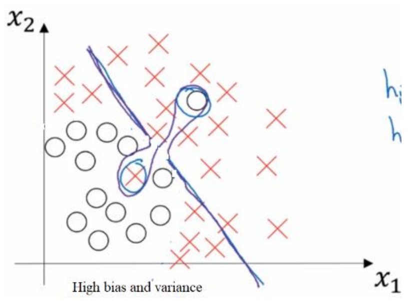
```
    for l in range(1, L):
        parameters['W'+str(l)] = np.random.randn(layers_dims[l], layers_dims[l-1]) * np.sqrt(2./layers_dim
s[l-1])
        parameters['b'+str(l)] = np.zeros((layers_dims[l], 1))
```

    - Tanh initialization: $\sqrt{\frac{1}{layers\_dims^{[l-1]}}}$
    - Xavier initialization: $\sqrt{\frac{2}{layers\_dims^{[l-1]} + layers\_dims^{[l]}}}$

Introduction to High Variance (Overfitting) and High bias (Underfitting)

- There are 2 ways to identify high variance or bias
    - Based on the boundary of classification



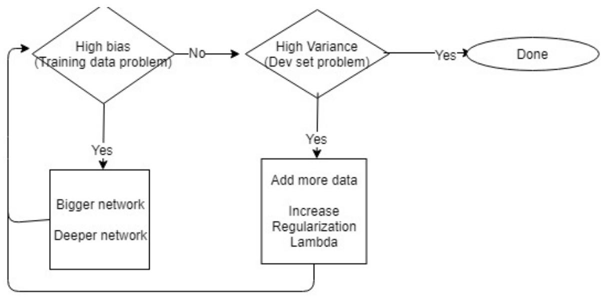


    - Based on Training set and Dev set error

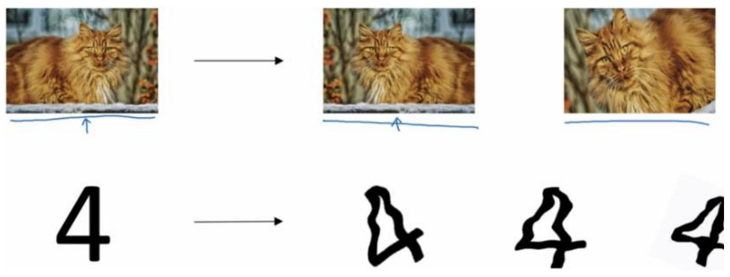
Table 1: Baseline error (Human detection) = 1%

Training set error	1%	15%	15%	0.5%
Dev set error	11%	16%	30%	1%
	High variance	High bias	High bias & variance	Low bias & variance

        - High variance when there is a big difference between Training and Dev set error
        - High bias when there is a big difference between Baseline and Training set error
- Basic recipe for Machine Learning



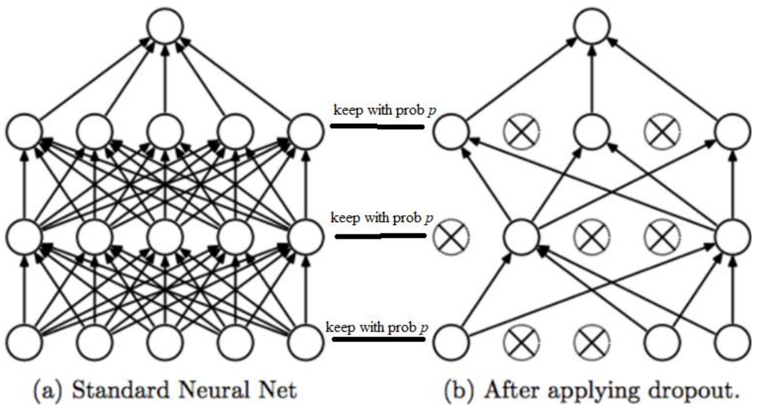
High bias (Training data problem)
No
High Variance (Dev set problem)
Yes
Done
Yes
Bigger network
Deeper network
Yes
Add more data
Increase Regularization Lambda

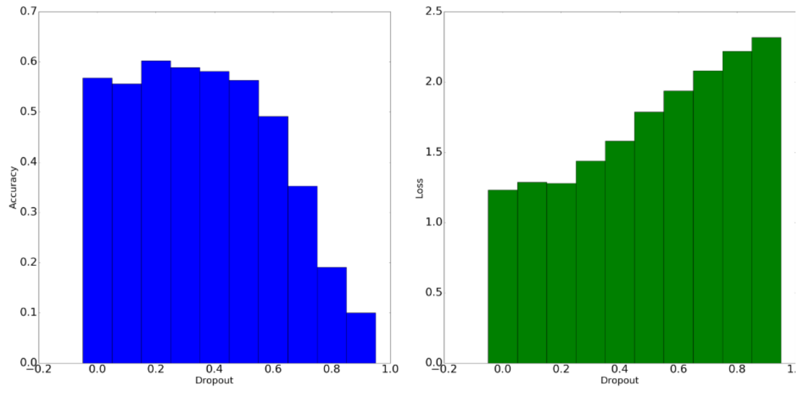


- Regularization
    - Weight Decay results in gradient descent shrinking the weights on every iteration $\left(1-\frac{\alpha\lambda}{m}\right)\omega$
    - Data Augmentation (example of *Cat and 4*)
- Early Stopping

Overfitting

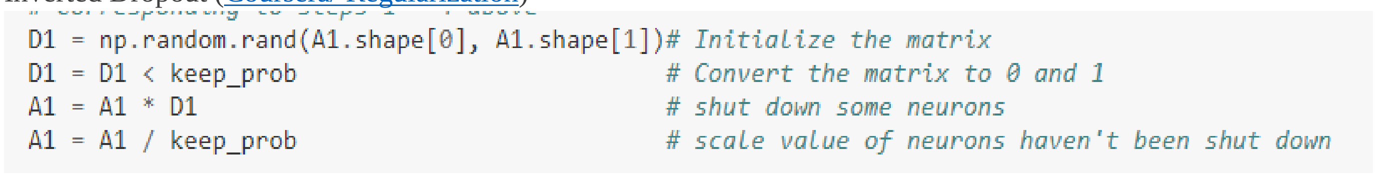
- Similar to overfitting problem in regression and classification
- Dropout (only applied in Train time)
    - refers to ignoring nodes (neurons) during the training phase which is chosen at random
    - Individual nodes are either dropped out of the net with probability *1-p* or kept with probability *p*
    - Why we need dropout
        - In regularization, the model is trained such that it does not learn interdependent features, so the features can be removed
        - Dropout is an approach to regularization in neural networks which helps reducing interdependent learning amongst the neurons
    - Procedures
        - Training Phase: For each hidden layer, for each training sample, for each iteration, ignore (zero out) a random fraction, p, of nodes (and corresponding activations)





Accuracy vs Dropout (Left) and Loss vs Dropout

- Experiment
    - Finally, I used dropout in all layers and increase the fraction of **dropout from 0.0 (no dropout at all) to 0.9** with a step size of 0.1 and ran each of those to 20 epochs
    - From the above graphs we can conclude that with increasing the dropout, there is some increase in validation accuracy and decrease in loss initially before the trend starts to go down.
    - **0.2 is the best dropout ratio**
- Inverted Dropout (Coursera/ Regularization)



```
D1 = np.random.rand(A1.shape[0], A1.shape[1])# Initialize the matrix
D1 = D1 < keep_prob                          # Convert the matrix to 0 and 1
A1 = A1 * D1                                 # shut down some neurons
A1 = A1 / keep_prob                          # scale value of neurons haven't been shut down
```

- Intuition of Why drop-out works: using dropout prevents the learning process from relying "too much" on a certain input variable, by randomly assigning it a value of 0 and therefore producing a large error. This error is then "corrected" by the learning algorithm and therefore the entire model becomes more robust. Therefore, increasing keep_prob will cause the neural network to end up with a lower training set error

Underfitting

- Similar to underfitting problem in regression and classification
- Output layer needs to be equivalent with your desired output. E.g. if your problem is classification, your output layer needs to be sigmoid, otherwise if your problem us regression, you don’t need to add activation function

Gradient Checking

- Code here, Doesn’t work with drop-out, Don’t use in training time – Only to debug
- If gradient check fails (10^{-3}), identify the bugs
- Remember the regularization (just need to replace *forward_propagation_neuralnet* by *forward_propagation_neuralnet_regularization*)

Problem of Optima

- Local optima in neural network is very rare because neural network has many variable and the condition to get optima is partial derivatives of all these variables must be 0
- More common problem in neural network is Saddle point which occurs when more than one of these variables are 0

- One more common problem in neural network is plateaus which make the parameters hard to update

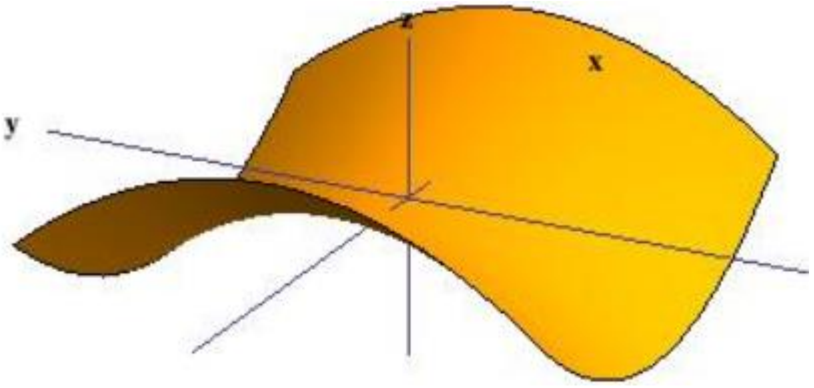
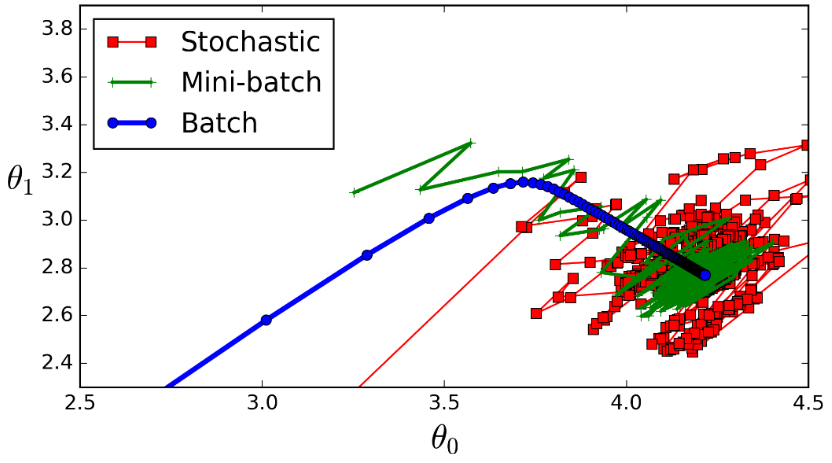


Figure 1: Saddle point

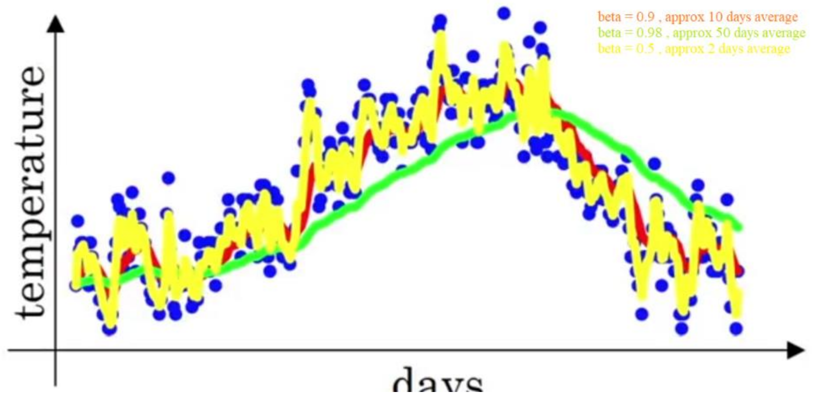
Gradient Descent

- *Batch* will use **full training data** at each iteration, with could be very expensive if our dataset is large and our networks have a lot of parameters.
    - In the first stage, θ may go fast to minimum values, but in the final stage, it goes so slowly because dataset may have some outliers so it may result in conflict among trend of data points
- *Stochastic* uses **only single data point** to propagate the error, which would make the convergence slow, as **the variance is big** (because Law of Large Numbers doesn't apply). and cannot use Vectorization
- *Mini-batch* is combining the best of both worlds. We don't use full dataset but we also don't use single data point either. For example, **we're using 50, 100, 200 or batch-size random subsets** of our dataset each time we train the networks. This way, we lower the computation cost, and yet we're still get lower variance than by using the Stochastic version.



- Exponentially Weighted Average (root of Momentum)

$$V_t(\text{temperature}) = \beta V_{t-1} + (1-\beta)\theta_t(\text{days})$$



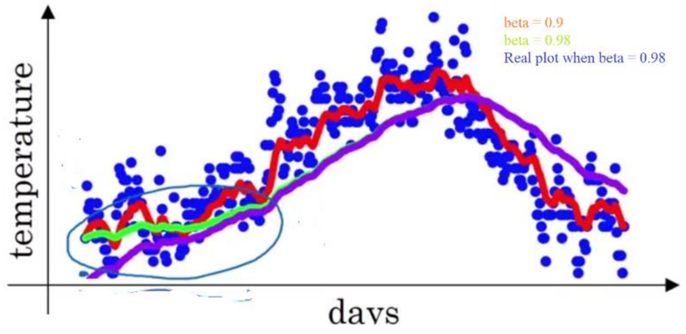
$$\begin{cases} v_{100} = 0.9v_{99} + 0.1\theta_{100} \\ v_{99} = 0.9v_{98} + 0.1\theta_{99} \\ v_{98} = 0.9v_{97} + 0.1\theta_{98} \\ \vdots \\ v_0 = 0 \end{cases}$$

$$\rightarrow v_{100} = 0.1\theta_{100} + 0.9(0.9v_{98} + 0.1\theta_{99}) = 0.1\theta_{100}$$
$$+0.9\cdot 0.1\theta_{99} + (0.9)^2\cdot 0.1\theta_{98} + (0.9)^3\cdot 0.1\theta_{97}$$
$$+(0.9)^4\cdot 0.1\theta_{96} + \dots$$

- v_{100} is **the average** of all θ, but $\varepsilon = 1-\beta = 0.1$, $(1-\varepsilon)^{\frac{1}{\varepsilon}} = 0.9^{10} \approx 0.35 \approx \frac{1}{e}$, because the last 10 days has coefficient of one third, coefficient of last >10 days is not considerable, so v_{100} is the **average of last 10 days**

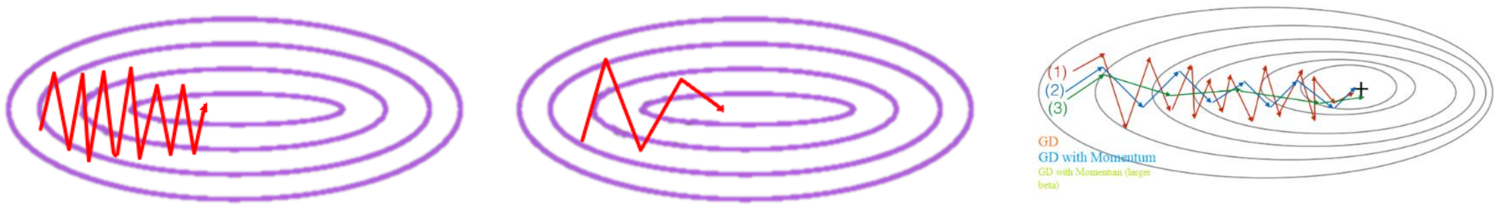
- Increasing β will make the plot smoother because it averages more days to eliminate the noise of certain days
- Decreasing β will create more oscillation within the red line because noise of certain days will make next day more oscillate

- Bias of Correction



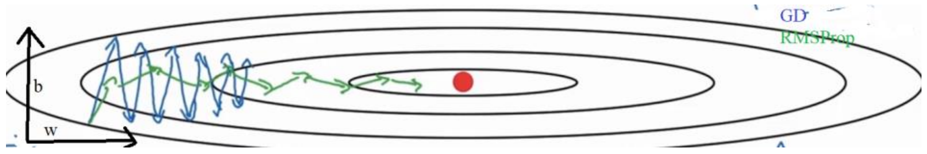
  - Because $v_0 = 0$, $v_1=0.9v_0+0.1\theta_1=0.1\theta_1$, first values of v (temperature) will be very small, but when calculating the next days, they will be correct and same as the green line
  - To correct the first values of v, we use $v_t = \frac{v_t}{1-\beta^t}$ and when t is larger $\frac{1}{1-\beta^t} \to 1$ and will not exert any impact on v_t

- Momentum



  - Original GD just care about the derivative of model, so it can't avoid the case of local minimum
  - Momentum GD not only calculate the derivative at one point but the momentum v_{t-1} : $\begin{cases} v^{[l]} = \beta v^{[l]} + (1-\beta)\nabla_{\theta^{[l]}} J\left(\theta^{[l]}\right) \\ \theta^{[l]} = \theta^{[l]} - v^{[l]} \end{cases}$
  - v_t is the average of $\frac{1}{1-\beta}\ \nabla_\theta J\left(\theta\right)$. Because in y-axis, average of up and down will be minus each other, in x-axis, when β is larger, it averages less amount of $\nabla_\theta J\left(\theta\right)$, so it will make Gradient Descent faster

- RMSProp



  - At very first derivatives, values of y-axis (db) is large, value of x-axis (dw) is small, but we want to update dw larger than db

$$\begin{cases} S_{dW^{[l]}} = \beta S_{dW^{[l]}} + (1-\beta) dW^{[l]^2} \\ S_{db^{[l]}} = \beta S_{db^{[l]}} + (1-\beta) db^{[l]^2} \end{cases} \to \begin{cases} w^{[l]} = w^{[l]} - \alpha \frac{dW^{[l]}}{\sqrt{S_{dW^{[l]}}}} \\ b^{[l]} = b^{[l]} - \alpha \frac{db^{[l]}}{\sqrt{S_{db^{[l]}}}} \end{cases}$$

  - Because dW is small, dW^2 is smaller, $\frac{dW}{\sqrt{S_{dw}}}$ is larger than just dW. Because db is large, db^2 is larger, $\frac{db}{\sqrt{S_{db}}}$ is smaller than db
  - x-axis and y-axis corresponding to w and b is just illustration in 2-dimension. In higher dimension, RMSProp helps to make feature variables which are small in very first derivative more effective to update these feature variables

- Adam
  - Adam combines 2 advantages of Momentum and RMSProp and apply bias-correction in corrected version, *t* is each step of mini-batch or batch or stochastic batch

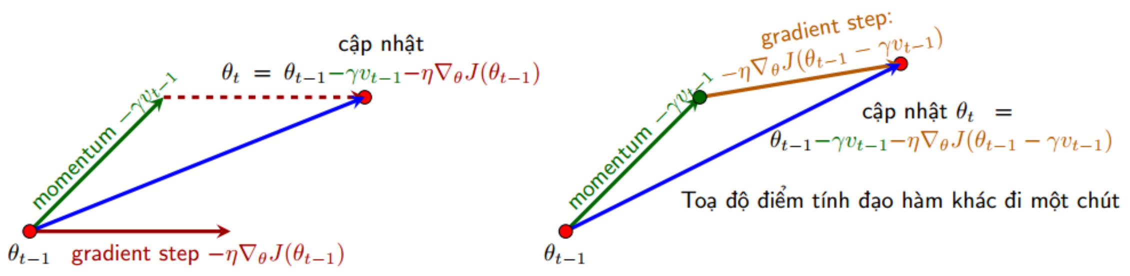
$$\begin{cases} v_{dW^{[l]}} = \beta_1 v_{dW^{[l]}} + (1-\beta_1) dW^{[l]} \\ v^{corrected}_{dW^{[l]}} = \dfrac{v_{dW^{[l]}}}{1-\beta_1^t} \\ S_{dW^{[l]}} = \beta S_{dW^{[l]}} + (1-\beta_2) dW^{[l]^2} \\ S^{corrected}_{dW^{[l]}} = \dfrac{S_{dW^{[l]}}}{1-\beta_2^t} \\ w^{[l]} = w^{[l]} - \alpha \dfrac{v^{corrected}_{dW^{[l]}}}{\sqrt{S^{corrected}_{dW^{[l]}}} + \varepsilon} \end{cases} \text{ and } \begin{cases} v_{db^{[l]}} = \beta_1 v_{db^{[l]}} + (1-\beta_1) db^{[l]} \\ v^{corrected}_{db^{[l]}} = \dfrac{v_{db^{[l]}}}{1-\beta_1^t} \\ S_{db^{[l]}} = \beta S_{db^{[l]}} + (1-\beta_2) db^{[l]^2} \\ S^{corrected}_{db^{[l]}} = \dfrac{S_{db^{[l]}}}{1-\beta_2^t} \\ b^{[l]} = b^{[l]} - \alpha \dfrac{v^{corrected}_{db^{[l]}}}{\sqrt{S^{corrected}_{db^{[l]}}} + \varepsilon} \end{cases}$$

  - There are various parameters needed to be tunes: α (need tuning), $\beta_1 = 0.9$ (usually used), $\beta_2 = 0.999$ (usually used), $\varepsilon = 10^{-8}$ (usually used)
- Learning rate decay
  - We need the larger learning rate at the first epochs, after that decrease the learning rate with certain pattern
  - Some popular patterns of learning rate decay:

$$\begin{cases} \alpha_t = \dfrac{1}{1 + decay\ \ rate \bullet epoch\_num} \alpha_{t-1} \left(= \dfrac{1}{1+2t} \alpha_0 \right) \\ \alpha_t = \text{exponential}\ \ decay^{epoch\_num} \alpha_{t-1} \left(= 0.95^t \alpha_0 \right) \\ \alpha_t = \dfrac{k}{\sqrt{epoch\_num}} \alpha_{t-1} \left(= \dfrac{1}{\sqrt{t}} \alpha_0 \right) \\ Decay\ \ staircase(decrease\ t\ times\ after\ each\ epoch) \end{cases}$$

The last pattern is **Manual Decay**, tune the learning rate based on the case we get, e.g. if model get a day or longer to train, one option is changing the learning rate

- Nesterov Accelerated Gradient (NAG)

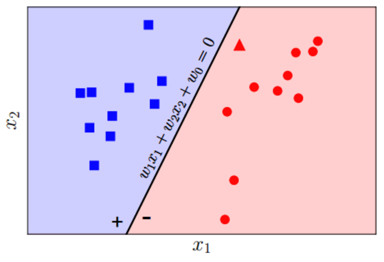

cập nhật
$\theta_t = \theta_{t-1} - \gamma v_{t-1} - \eta \nabla_\theta J(\theta_{t-1})$
momentum $-\gamma v_{t-1}$
θ_{t-1}
gradient step $-\eta \nabla_\theta J(\theta_{t-1})$
gradient step: $-\eta \nabla_\theta J(\theta_{t-1} - \gamma v_{t-1})$
momentum $-\gamma v_{t-1}$
cập nhật θ_t =
$\theta_{t-1} - \gamma v_{t-1} - \eta \nabla_\theta J(\theta_{t-1} - \gamma v_{t-1})$
Toạ độ điểm tính đạo hàm khác đi một chút
θ_{t-1}

(a) Momentum gradient descent. (b) Nesterov accelerated gradient.

  - Momentum GD can help to overcome the local minimum, but the side effect is wasting several iterations to oscillate about the global minimum
  - NAG will take derivative at the point which uses Momentum: $\begin{cases} v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1}) \\ \theta_{t+1} = \theta_t - v_t \end{cases}$

Perceptron Learning Algorithm

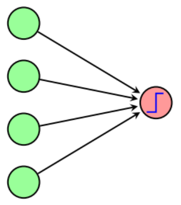
$$\operatorname{sgn}(\theta^T \mathrm{x}) = label(\mathrm{x}) = \begin{cases} 1\ if\ \ \theta^T \mathrm{x} \geq 1 \\ -1\ o.w \end{cases}$$



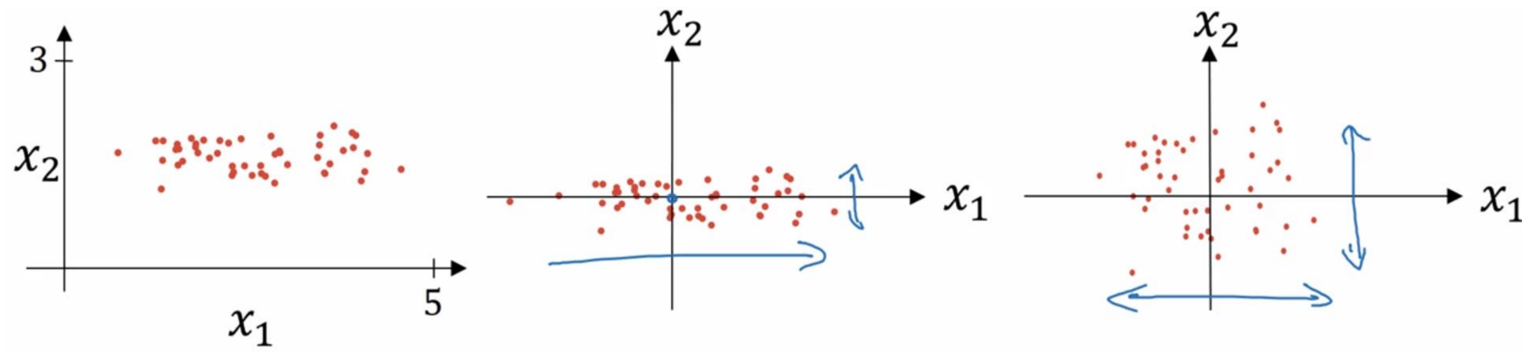
- Cost function
  - If one point is misclassified, $\operatorname{sgn}(\theta^T \mathrm{x}_i) \neq y_i$, so $J_1(\theta) = \sum_{\mathrm{x}_i \in M} -y_i \operatorname{sgn}(\theta^T \mathrm{x}_i)$, M is set of misclassified points, but the **sgn** function is almost impossible to take derivative due to discrete function
  - Instead, $J_1(\mathrm{w}) = \sum_{\mathrm{x}_i \in M} -y_i \mathrm{w}^T \mathrm{x}_i$. This function has its own advantage: if one point is misclassified and this point is far from boundary, the cost function will be larger

- Optimize the Cost function
  - Assume SGD is applied:

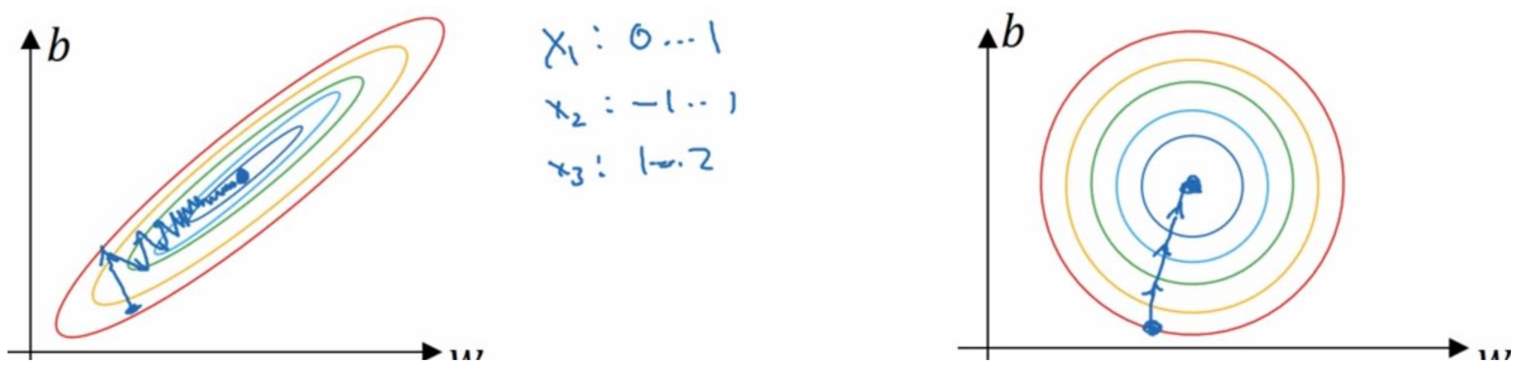
$$J(\theta; x_i; y_i) = -y_i\theta^T x_i \rightarrow \nabla_\theta J = -y_i x_i \rightarrow \theta_{t+1} = \theta_t + \eta y_i x_i$$



Input Normalization



- So it makes the cost function faster to optimize



Vanishing/ Exploding Gradient

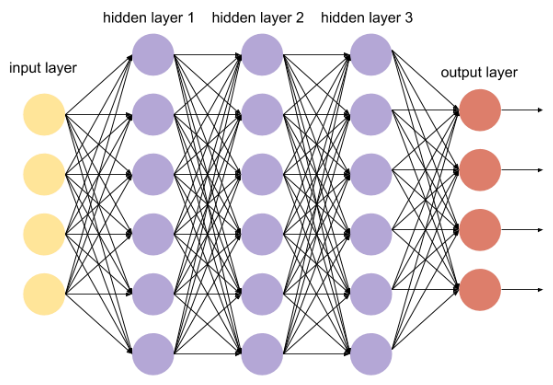
- When $W^{[l]} \gg I$ is too large and ReLu is applied, $\hat{y} = W^{[L]}W^{[L-1]}\ldots W^{[1]}\mathbf{x} = \left(W^{[l]}\right)^L \mathbf{x} \rightarrow \infty$ will be exploding , we cap them to a Threshold value to prevent the gradients from getting too large

$$W_{new}^{[l]} = \frac{W^{[l]}\ \mathbf{x}\ \textit{threhold}}{\left\|W^{[l]}\right\|_2}$$

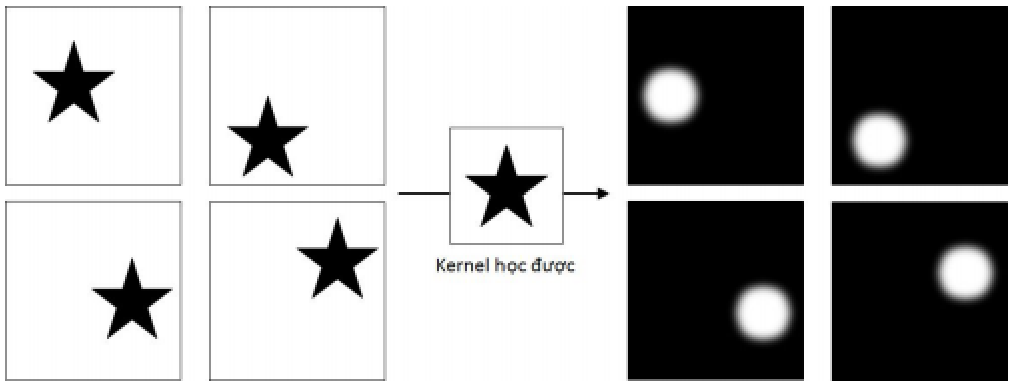
- When $W^{[l]} \approx \mathbf{0}$ is too small and ReLu is applied, $\hat{y} \rightarrow 0$ will be vanishing
- Another solution to solve vanishing or exploding gradient is **Initialization**

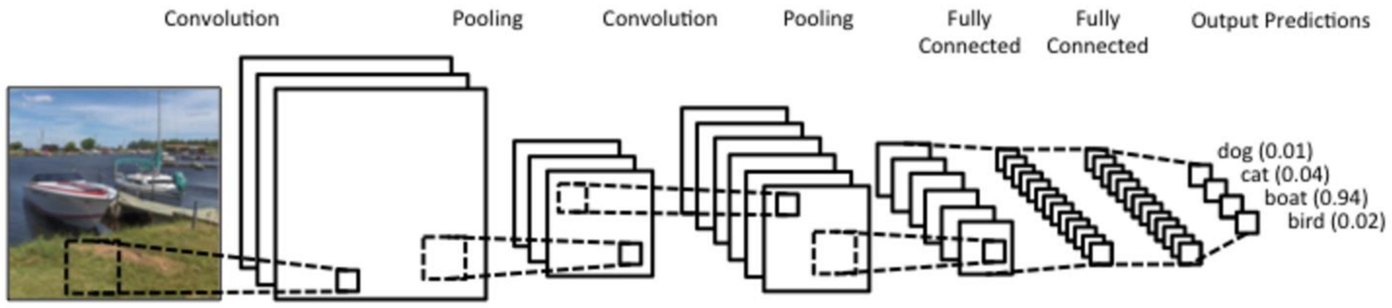
Convolutional Neural Network

- The neural network in the first part is Fully Connected Neural Network



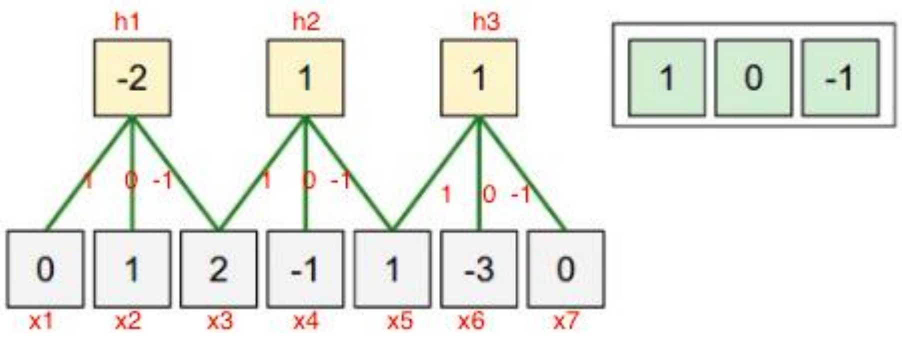
    - Each input layer is each row in your dataset and each output layer is all probability that this row may belong to in softmax regression
- *However, fully connected neural network has some drawbacks*
    - When your dataset is small, fully connected gives you lots of weight, for example, you have image of 256x256x3, and in first hidden layer, you need 256 neurons, so you need 256*256*256*3=50331648 weights →Your model easily get overfitting
    - In computer vision, you have to flatten the image from (# of image, height, width, depth) to (# of image, height x width x depth) in logistics regression when implementing in Neural Network, so Your Neural Network cannot adapt with **Translational Invariance**, which occurs when your train and test image is mostly identical except the position of object you need to detect. In this case, your neural network just predicts the image which has position of object similar with train image, when there is image with another position of same object, your NN cannot predict. This is also the overfitting in image processing
    - More about Computer Vision with NN, your model should be train to be invariant with
        - position of object (which CNN will help)
        - rotation of object (you need to input the proper data)
        - zoom of object (you need to input the proper data)
- *Convolutional Neural Network will help to solve some drawback of Fully Connected*
    - It requires fewer weights → avoid overfitting
    - Avoid translational invariance →your model can adapt with position of object

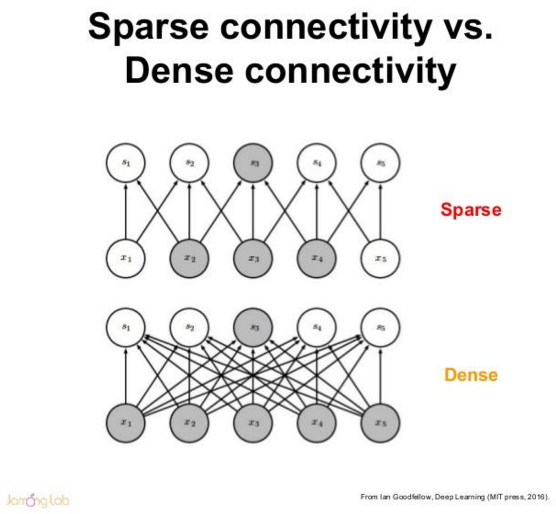




    - *2 features make CNN stands for when compared to Fully Connected NN*

- Weight sharing/ filter: difference outputs can use same weights

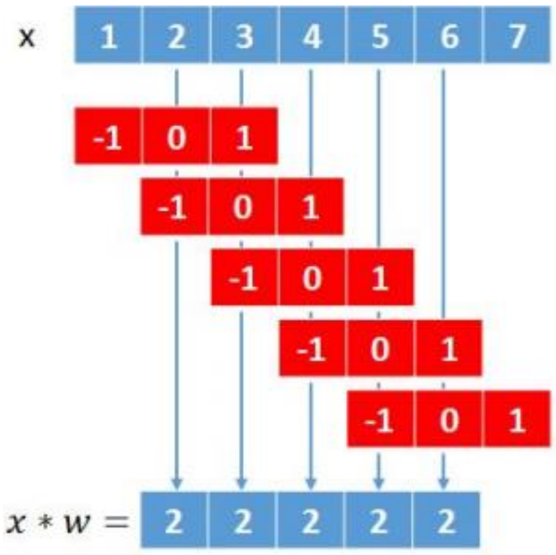



Sparse connectivity vs. Dense connectivity
Sparse
Dense
From Ian Goodfellow, Deep Learning (MIT press, 2016).

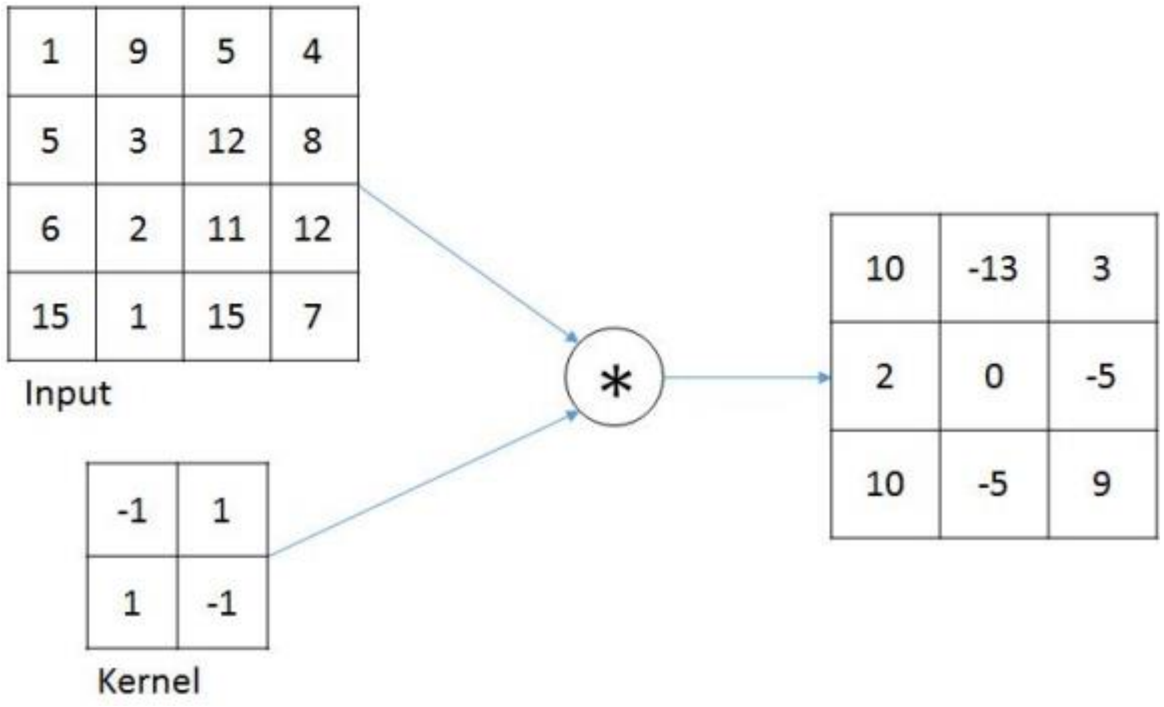
- Sparsely connected

  - **1 CNN usually includes**
    - **Convolution layers**
    - **Activation Function (ReLU, Sigmoid, Tanh)**
    - **Pooling layer**
    - **Fully-connected layer**

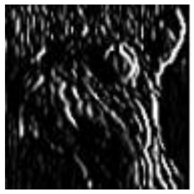
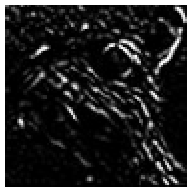
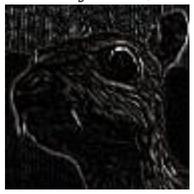
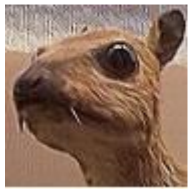
- *Convolution*
  - It helps you to dot product two vector which are not same dimension
  - Suppose you have w = [1, 0, -1], you need to make w_1 = [-1, 0, 1]. Now convolution between x and w_1

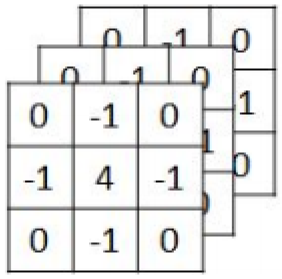


  - Convolution Matrix

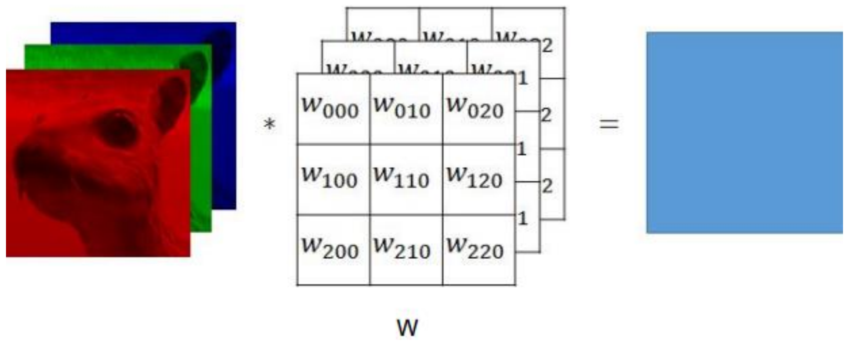


        - In the above example, you slide filter 2 x 2 on 4 x 4 image, so the output is 4-2+1 = 3x3
- Some popular filters

    - Sobel X: $\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$ 
        - Partial Derivative based on x axis, so you can see the differences between pixel based on x axis
    - Sobel Y: $\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$ 
        - Partial Derivative based on y axis, so you can see the differences between pixel based on y axis
    - Laplacian: $\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$ 
        - Second derivative, so it can find the max between pixel
    - Sharpen: $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$ 
    - When you apply above filter to image, it needs to be:

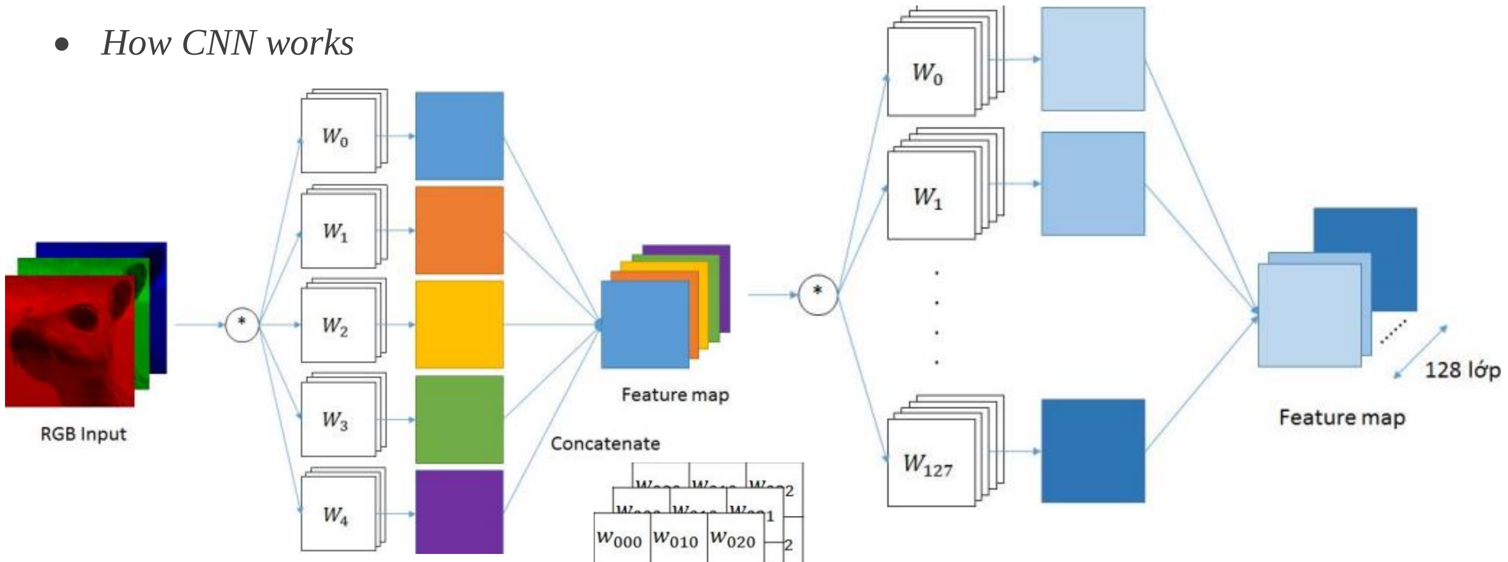


or more exact

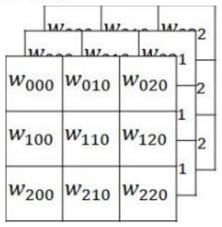


w_{000} w_{010} w_{020}
w_{100} w_{110} w_{120}
w_{200} w_{210} w_{220}
W

- **In CNN. the above weights are what you need to find out**
- *How CNN works*



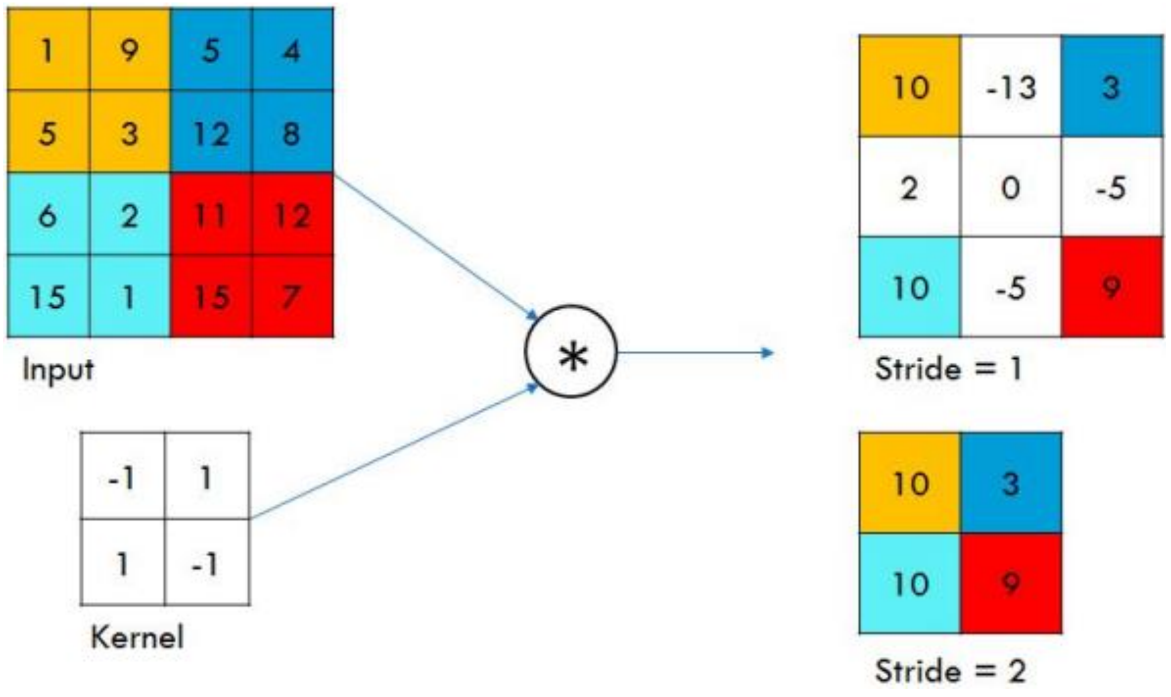
- In this case, W$_0$ is equivalent



w_{000} w_{010} w_{020}
w_{100} w_{110} w_{120}
w_{200} w_{210} w_{220}

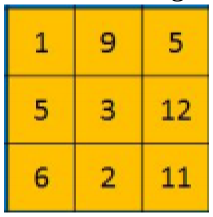
- In the first convolution layer, you have tensor W = 3 x 3 x 3 x 5, so the output you have 5-layer feature map
- In another convolution, you have tensor W = 3 x 3 x 5 x 128 because your input is 5-layer feature map and number of weights is 128

- *Stride*
  - Number of steps the filter slides



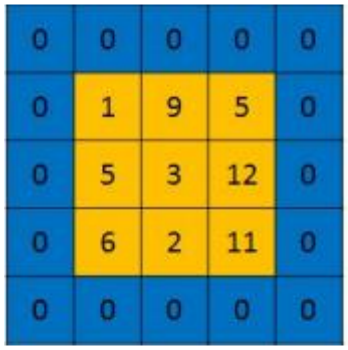
- *Padding*

- When you do convolution, the border of the image will be lost, padding is how you cover this border
- The image before padding:



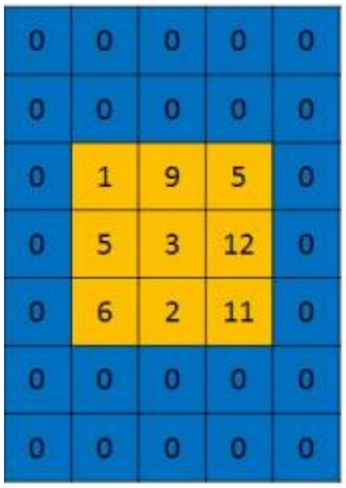
1	9	5
5	3	12
6	2	11

- Zero Padding with dimension of (1, 1):



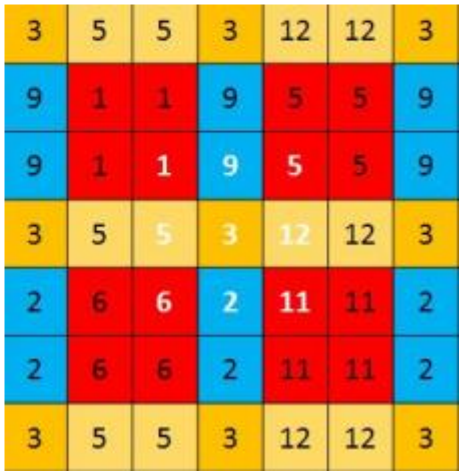
0	0	0	0	0
0	1	9	5	0
0	5	3	12	0
0	6	2	11	0
0	0	0	0	0

- Zero Padding with dimension of (2, 1):



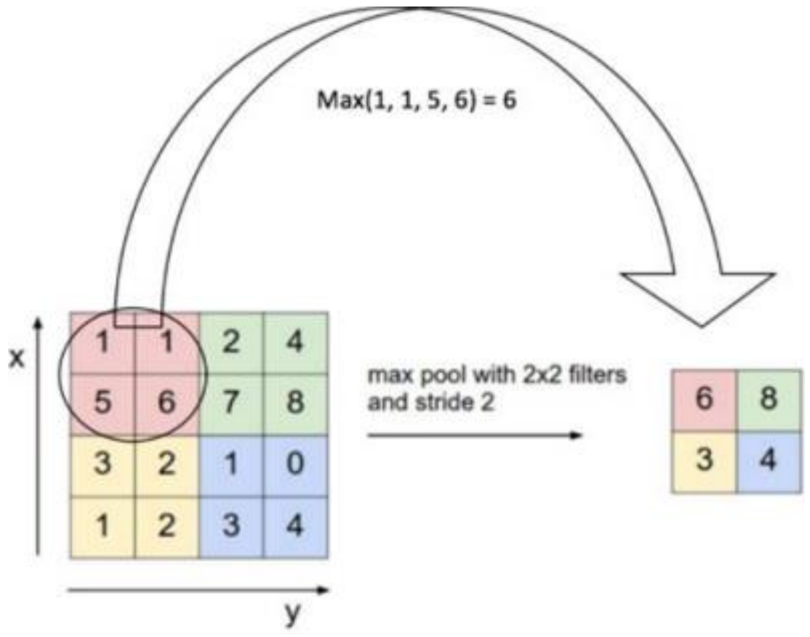
0	0	0	0	0
0	0	0	0	0
0	1	9	5	0
0	5	3	12	0
0	6	2	11	0
0	0	0	0	0
0	0	0	0	0

- Symmetric Padding with dimension of (2, 2)

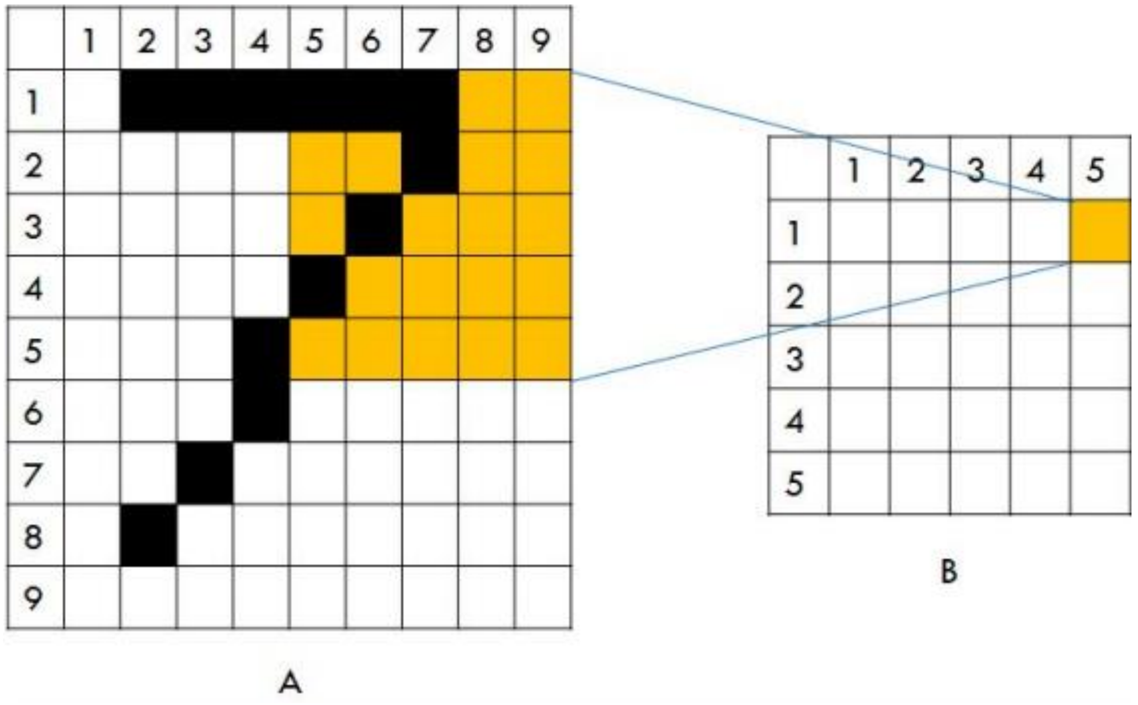


3	5	5	3	12	12	3
9	1	1	9	5	5	9
9	1	1	9	5	5	9
3	5	5	3	12	12	3
2	6	6	2	11	11	2
2	6	6	2	11	11	2
3	5	5	3	12	12	3

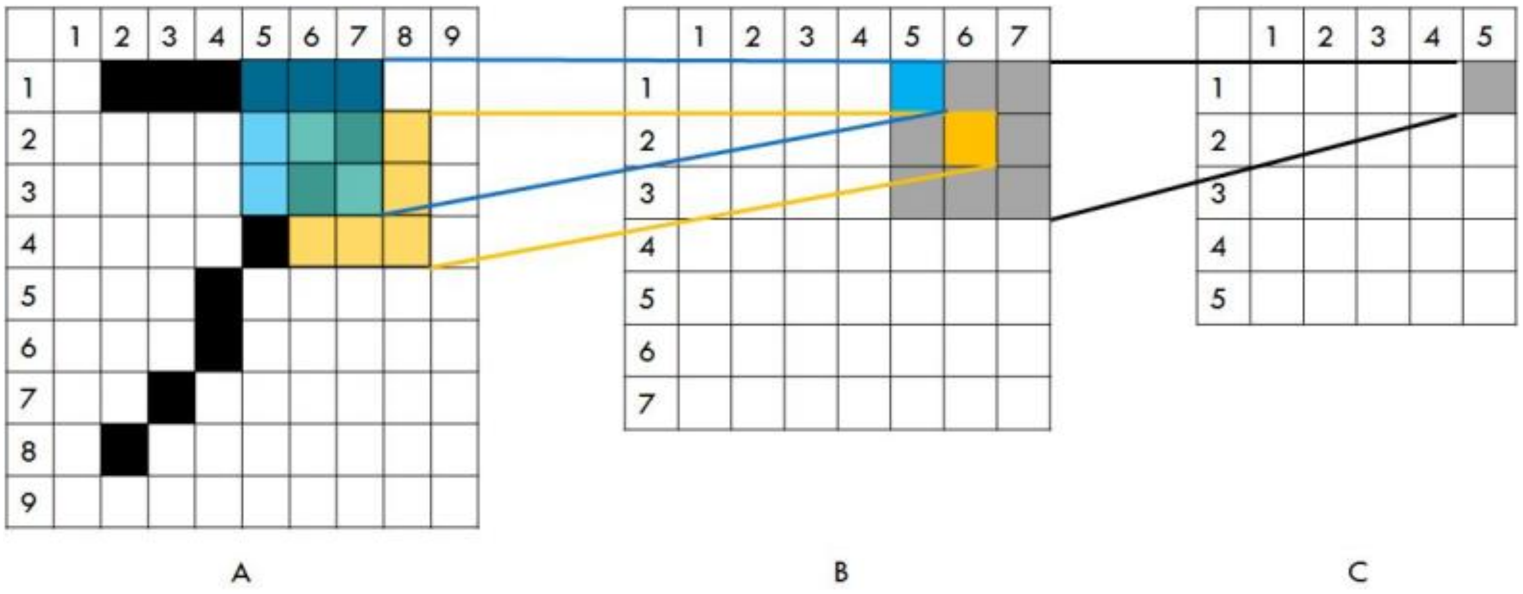
- *Pooling*
  - 2 types: Max Pooling and Average Pooling
  - Max Pooling is the special kernel/ filter which find the max value in the kernel
  - Stride in Max pooling = dimension of kernel



- *Receptive field*
  - The way you design the weights
  - If the weight matrix = (5, 5), so number of weight = 25, but it may not capture all features



  - If the weight matrix = (3, 3) and in the next layer, weight matrix = (3, 3), so number of weight = 3 x 3 + 3 x 3 = 18 and your model can capture more feature when compared to above example



- So the problem is you need to design the weight such that it can capture as many features as possible

Process in NN

Input (can be an image or 55000 images with dimension of 28x28 = matrix(55000 x (28 x28) = matrix(55000 x 784))

→ Conv1_1 (128 channels/128 filters/ 128 kernels) → ReLU→ Feature Map (128 features/ images) → Conv1_2 (128 channels/128 filters/ 128 kernels) → ReLU: Stack 1

→ Pooling 1

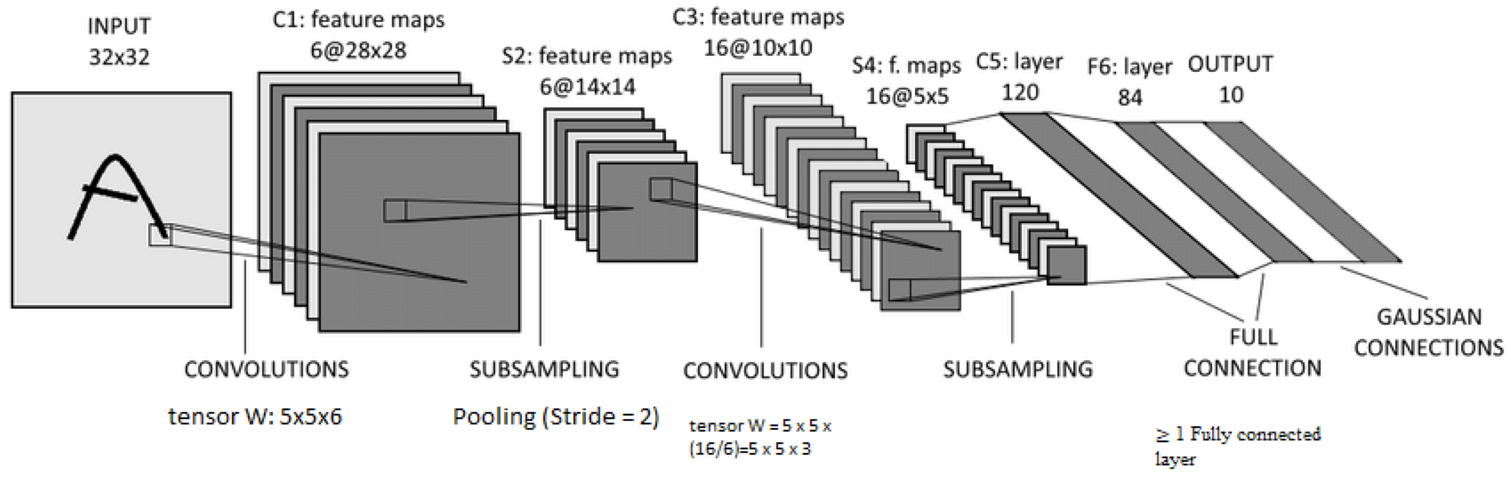
→ Conv2_1 (256 channels) → ReLU→ Feature Map (256 features/ images) → Conv1_2 (256 channels) → ReLU: Stack 2

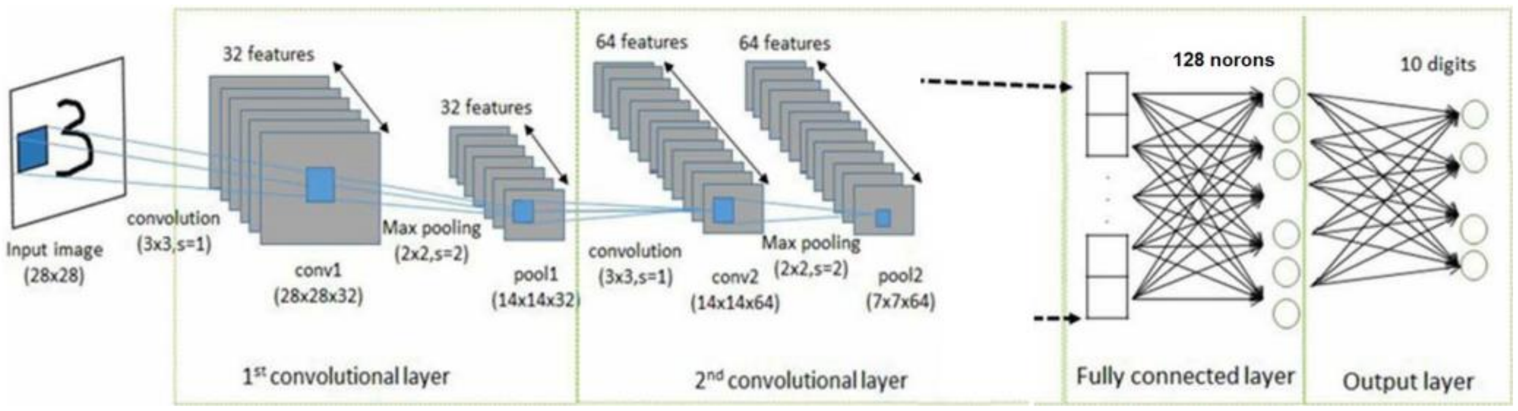
→ Pooling 2

…

→ FC (≥ 1 Fully connected layer)

LeNet

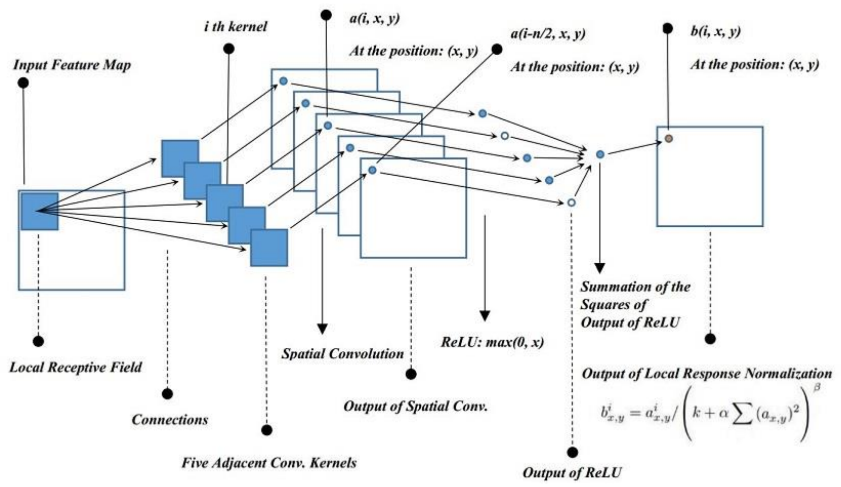




- Identify hand written digit
- Compare to original LeNet in 1998, Current LeNet has some differences
    - Activation Function: ReLU rather than Sigmoid
    - Pooling: Max Pooling rather than Average Pooling
    - Use Dropout

Normalization

Local Response Normalization



$$b^i_{x,y} = a^i_{x,y} \Big/ \left(k + \alpha \sum_{j=\max\left(0,i-\frac{n}{2}\right)}^{\min\left(N-1,i+\frac{n}{2}\right)} \left(a^j_{x,y}\right)^2\right)^\beta$$

$a^i_{x,y}$: output (after ReLU) of *i* th convolution kernel at the position (*x*, *y*) in the feature map

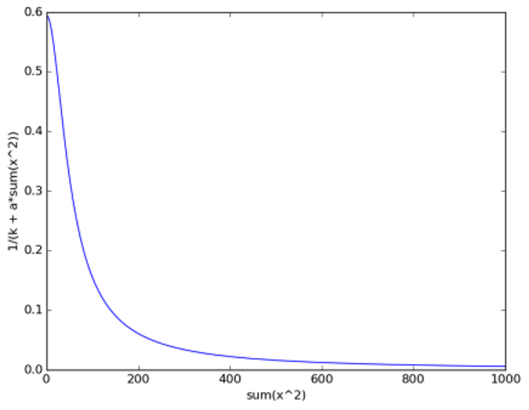
$b^i_{x,y}$: output of LRN, input for next layer

N: total number of kernels in the layer

n: hyper-parameter, number of adjacent kernels, here we choose *n = 5*

k, α, β: hyper-parameter, here we choose, k = 2, α = 10^{-4}, β = 0.75

- Intuition



$$y-axis = \frac{b^i_{x,y}}{a^i_{x,y}} = 1 \Big/ \left(k + \alpha \sum_{j=\max\left(0,i-\frac{n}{2}\right)}^{\min\left(N-1,i+\frac{n}{2}\right)} \left(a^j_{x,y}\right)^2\right)^\beta$$

If 2 values of $a^i_{x,y}$ pass through this function, because the slope in the beginning is very steep, little difference between 2 inputs $a^i_{x,y}$ will be very large in the output

The idea behind is to improve the “peaks” and diminish the “flat” response and easily for model to detect the features

Batch Normalization

- Internal Covariate Shift
    - Each layer tries to model the input from previous hidden layer
    - If input statistical distribution keep changing after a few iterations, which is called Internal Covariate Shift, the hidden layers will keep trying to adapt to that new distribution, so it will slow down the convergence
    - Batch Normalization tries to normalize the input of each of hidden layer, so the convergence may occur faster
    - Batch Normalization make GD work in linear region so avoid vanishing gradient

- Process
  - Input: Values of *x* over a mini-batch $B = \{x_1, x_2, \ldots, x_m\}$ and parameter to be learned: γ and β
  - Output: $\{y_i = BN_{\gamma,\beta}(x_i)\}$

$$\begin{cases} \mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \\ \sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \\ \hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \\ y_i = \gamma \hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i) \end{cases}$$

- Advantage
  - **Model train faster**: while training will be slower due to extra calculation and hyper-parameter in the backprop, model should converge more quickly so training will be faster overall → **Make the activation function** more effective by avoiding the special regions in activation function
  - **Allow higher learning rate**: in the common model, increasing learning rate may make model not to converge, but when BN is applied, we can increase learning rates to make model converge faster
  - **Makes weight less sensitive to initial weight**
  - **Reduce the dropout:** dropout may remove the important feature

From LeNet to DenseNet

LeNet

- Input shape: $32 \times 32 \times 3$
- Layer 1

  Convolution layer 1
  - Kernel: $5 \times 5 \times 3$, number of filter: 6 → Tensor W$_1$ = $\begin{bmatrix} 5 & 5 & 3 & 6 \end{bmatrix}$
  - Stride: 1
  - No Padding

  → Output: $\begin{bmatrix} 28 & 28 & 6 \end{bmatrix}$

  Max-pooling layer 1
  - Pooling size: 2×2
  - Stride: 2
  - Padding = "same"

  → Output: $\begin{bmatrix} 14 & 14 & 6 \end{bmatrix}$

- Layer 2

  Convolution layer 2
  - Kernel: $5 \times 5 \times 6$, number of filter: 16 → Tensor W$_1$ = $\begin{bmatrix} 5 & 5 & 6 & 16 \end{bmatrix}$
  - Stride: 1
  - No Padding

  → Output: $\begin{bmatrix} 10 & 10 & 16 \end{bmatrix}$

  Max-pooling layer 2
  - Pooling size: 2×2
  - Stride: 2
  - Padding = "same"

→ Output: $\begin{bmatrix}5 & 5 & 16\end{bmatrix}$

- Flatten output: $5\times5\times16=400$
- Fully connected 1: output = 120
- Fully connected 2: output = 84
- Softmax layer, output = 10 (for 10 digits)

Alexnet (2012)

- Summary
  - **Architect**: 5 convolution layer and 3 FC layer, input: imagenet size is $227\times227\times3$ and 1000 class
  - Overlap Pooling
  - Use Local Response Normalization to normalize each layer
  - Use Data Augmentation to synthesize the training data by translations and horizontal reflections
  - Learning rate = 0.01, Momentum = 0.9, Weight Decay = 0.005
- Input shape: $227\times227\times3$
- Layer 1

  Convolution layer 1

  - Kernel: $11\times11\times3$, number of filter: 96 →Tensor W_1 = $\begin{bmatrix}11 & 11 & 3 & 96\end{bmatrix}$
  - Stride: 4
  - No Padding

→ Output: $\begin{bmatrix}\frac{W-F_w+2P}{S_w}+1 & \frac{H-F_h+2P}{S_h}+1 & 96\end{bmatrix}$ in which input size: $W\times H\times3$ and kernel size: $F_w\times F_h\times3$, P = 0 if no padding

$$=\begin{bmatrix}\frac{227-11}{4}+1 & \frac{227-11}{4}+1 & 6\end{bmatrix}=\begin{bmatrix}55 & 55 & 96\end{bmatrix}$$

  Max-pooling layer 1

  - Pooling size: 3×3
  - Stride: 2
  - Padding = “same”

  → Output: $\begin{bmatrix}27 & 27 & 96\end{bmatrix}$

  Normalize layer

- Layer 2

  Convolution layer 2

  - Kernel: $3\times3\times96$, number of filter: 256 →Tensor W_1 = $\begin{bmatrix}3 & 3 & 96 & 256\end{bmatrix}$
  - Stride: 1
  - Padding = “same”

  → Output: $\begin{bmatrix}27 & 27 & 256\end{bmatrix}$

  Max-pooling layer 2

  - Pooling size: 3×3
  - Stride: 2
  - Padding = “same”

  → Output: $\begin{bmatrix}13 & 13 & 256\end{bmatrix}$

  Normalize layer

- Layer 3

Convolution layer 3

- Kernel: $3\times3\times256$, number of filter: 384 →Tensor W$_1$ = $\begin{bmatrix}3 & 3 & 256 & 384\end{bmatrix}$
- Stride: 1
- Padding = "same"

→ Output: $\begin{bmatrix}13 & 13 & 384\end{bmatrix}$

- Layer 4

  Convolution layer 4
  - Kernel: $3\times3\times384$, number of filter: 384 →Tensor W$_1$ = $\begin{bmatrix}3 & 3 & 384 & 384\end{bmatrix}$
  - Stride: 1
  - Padding = "same"

  → Output: $\begin{bmatrix}13 & 13 & 384\end{bmatrix}$
- Layer 5

  Convolution layer 5
  - Kernel: $3\times3\times384$, number of filter: 256 →Tensor W$_1$ = $\begin{bmatrix}3 & 3 & 384 & 256\end{bmatrix}$
  - Stride: 1
  - Padding = "same"

  → Output: $\begin{bmatrix}13 & 13 & 256\end{bmatrix}$

  Max-pooling layer 5

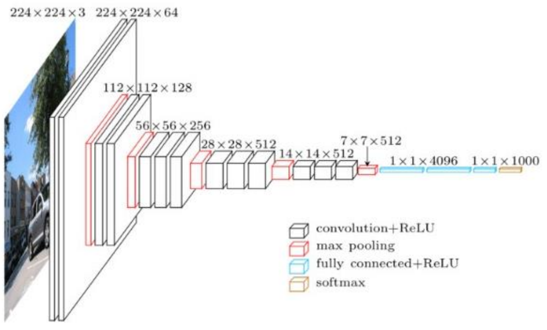
  - Pooling size: 3×3
  - Stride: 2
  - Padding = "same"

  → Output: $\begin{bmatrix}6 & 6 & 256\end{bmatrix}$

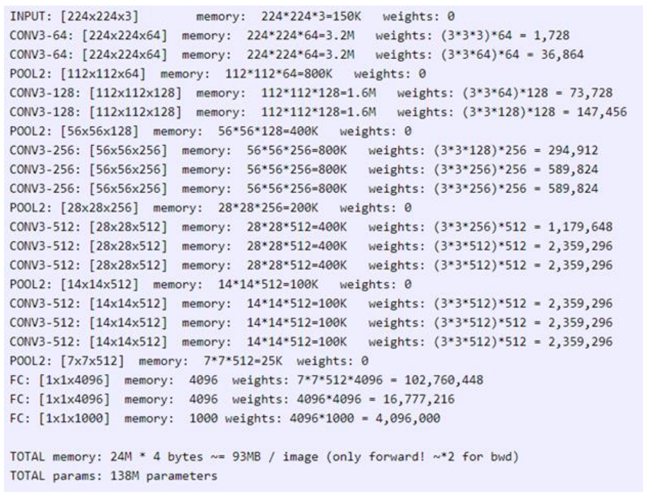
- Flatten output: $6\times6\times256=9216$
- FC layer 1: output = 4096 + dropout(0.5)
- FC layer 2: output = 4096 + dropout(0.5)
- Softmax layer: output = 1000

VGGNet (2014)

- Deeper than AlexNet, both LeNet and AlexNet use Convolution – Maxpooling, VGG use the convolution chain in the middle and end. Despite the computational cost, the model can gain more features than max-pooling followed by convolution



- VGG16 includes: 13 convolution layer (2 conv-conv layer and 3 conv-conv-conv layer) has kernel size of 3×3. After conv chain, max-pooling to 0.5 is needed and 3 FC layer



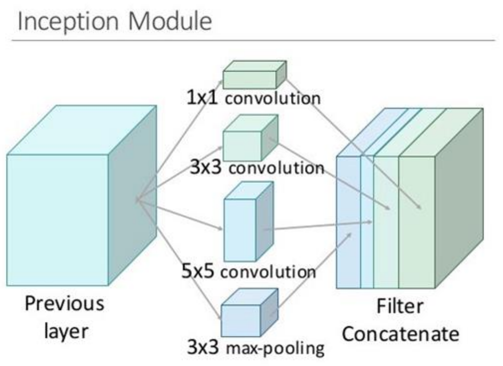
```
INPUT: [224x224x3]        memory:  224*224*3=150K   weights: 0
CONV3-64: [224x224x64]  memory:  224*224*64=3.2M   weights: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64]  memory:  224*224*64=3.2M   weights: (3*3*64)*64 = 36,864
POOL2: [112x112x64]  memory:  112*112*64=800K   weights: 0
CONV3-128: [112x112x128]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

TOTAL memory: 24M * 4 bytes ~= 93MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters
```

- Summary
  - Kernel size: 3×3 instead of 11×11 in AlexNet. Based on Receptive Field, 2 stack of kernel 3×3 is better than 1 kernel 5×5, 3 stack of kernel 3×3 is better than kernel 7×7
  - Use batch GD
  - Use data augmentation during training

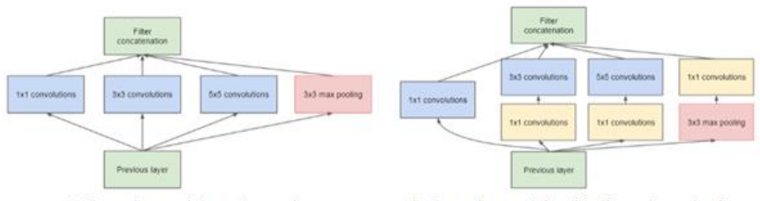
GoogleNet (2014)

- GoogleNet Performance (6.7% error rate) is better than VGG (7.3% error rate)
- It includes multiple modules: Inception with parameter of 5M instead of 60M in AlexNet
- Training wider instead of deeper which idea VGGNet implements
- Output of kernel size of 5×5 and 3×3 are different. Inception module will **calculate possible kernel size** from one input and concatenate them to output



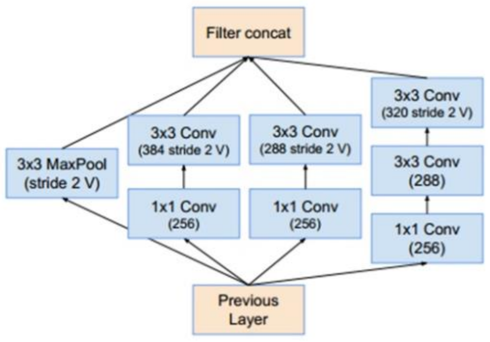
- We need 1×1 convolution because of **dimensionality reduction** which reduces the number of parameter and value in convolution process (γ). E.g.
  - Input shape: $28\times28\times192$, kernel size: 5×5 and number of kernel: 32 → output: $28\times28\times32$ $\rightarrow \gamma = 28\times28\times192\times5\times5\times32 = 120M$
  - Input shape: $28\times28\times192$ pass through 2 convolution, first kernel: $1\times1\times192$, number of kernel: 16 and second kernel: $5\times5\times16$, number of kernel: 32→ output is still: $28\times28\times32$, but $\gamma = (28\times28\times16)\times192 + (28\times28\times32\times5\times5\times16) = 12.4M$ is much smaller than first case

- Inception v1



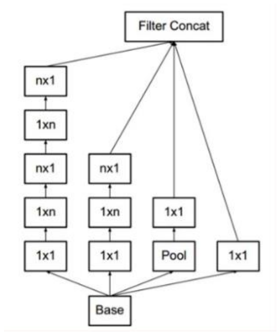
(a) Inception module, naïve version (b) Inception module with dimension reductions

- Inception v2: some points upgraded compared to v1



  - Add Batch Normalization to avoid Internal Covariate Shift, so output ~ N(0, 1)
  - Convolution 5×5 is replaced by 2 convolution 3×3

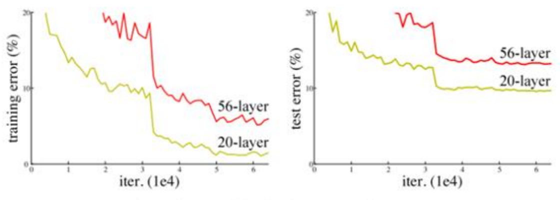
- Inception v3
  - Factorization: convolution 7×7 will be analyzed into 7×1 and 1×7 . Similarly, 3×3 into 3×1 and 1×3



- Detail GoogleNet
- Summary
  - Includes 22 layers, in the first 4 layers, there are simple convolutions and then max pooling
  - 9 inception modules are used so it makes model more deeper
  - Not flatten feature map before FC layer but use average pooling

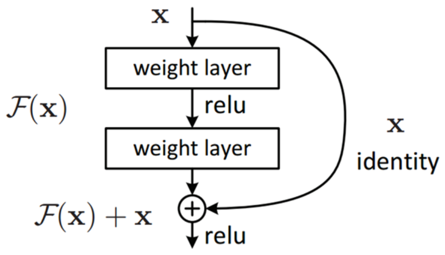
ResNets (2015)

- Developed by Microsoft with error rate of 3.57% in ILSVRC, it solves the problem when making the model deeper
- When model is deeper
  - The gradient easily gets vanishing or explodes, in this case, Batch Normalization may solve to normalize the input of every hidden layer to make it not so big or small so, as mentioned before, the model may converge faster
  - The model may get the problem: **degradation**, when the model accuracy score starts to saturate, the action, e.g. making the model deeper, cannot make score increase



increasing network depth leads to worse performance

- The problem degradation occurs because the model is trained to the upper threshold, cannot learn any more feature even make it deeper, ResNet solves it by Residual Block: model may easily learn more features when adding the features before


x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

- Summary
  - Batch Normalization after Convolution layer
  - **Initialization Xavier / 2 = 1/2n**
  - SGD and momentum (0.9)
  - Learning rate: 0.1 or 0.01 if error doesn't decrease
  - Mini batch size: 256, Weight decay: 10^{-5}
  - Dropout is not used

Transfer Learning

- Some popular NN: VGG 16 (16 layer), ResNet (152 layers), GoogleNet(22 layers)
- Transfer Learning: because implementing NN consume much resource, you should get the weight from already implemented NN to your model

Where CNN Should use and not use

- Dành cho Dataset có đặc trưng liên quan với nhau về mặt không gian: Ảnh, Sóng Âm
- Các loại dataset trong Data Science, không nên áp dụng CNN

Apply

- Object Segmentation
- Image Captioning
- Enhance quality of image

Lab 02

```
conv = tf.nn.conv2d(images, kernel, [1, 1, 1, 1], padding='SAME')
```

- A 4-dim tensor has shape [batch, height, width, channel]. For example, we could have a tensor with shape [10, 80, 120, 3] which means the batch has 10 images, each of which are 80 pixels high and 120 pixels wide, with 3 channels (e.g. RGB colors).
- The word **stride** is similar in meaning to a step-size
- [1, H, W, 1] mean that after calculating each convolution, we move filter H pixels in height of image and W pixels in weight of image
- There are 2 types of padding
  - SAME: zero-padding
  - VALID: without padding

```
# Layer 1: Conv
with tf.name_scope('conv1') as scope:
  kernel = tf.Variable(tf.random_normal([3, 3, 1, 32], dtype=tf.float32, stddev=1e-1,
    name='weights'))
```