Logistic Regression

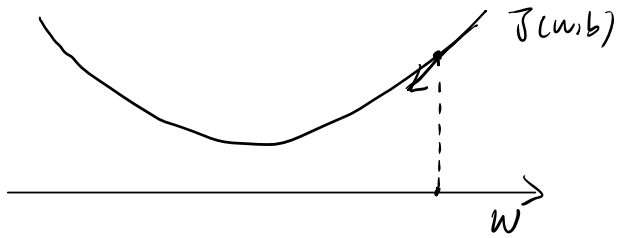
$$\hat{y} = \sigma(\underbrace{w^T x + b}_{z})$$

Notation:
- m: sample size
- n_x: # of feature.
- w: $n_x, 1$
- x: n_x, m (column/sample)

Loss fx: $L(y, \hat{y}) = -y \log \hat{y} - (1-y) \log(1-\hat{y})$

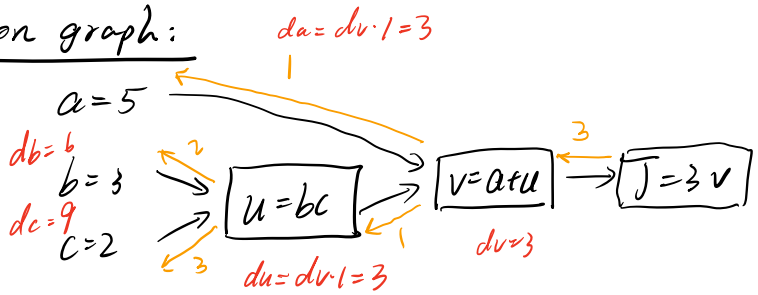
Cost fx: $J(w,b) = \frac{1}{m} \sum^{m} [-y \log \hat{y} - (1-y) \log(1-\hat{y})]$ — convex fx

$w: w - \alpha \frac{\partial J(w,b)}{\partial w}$ $\quad b: b - \alpha \frac{\partial J(w,b)}{\partial b}$



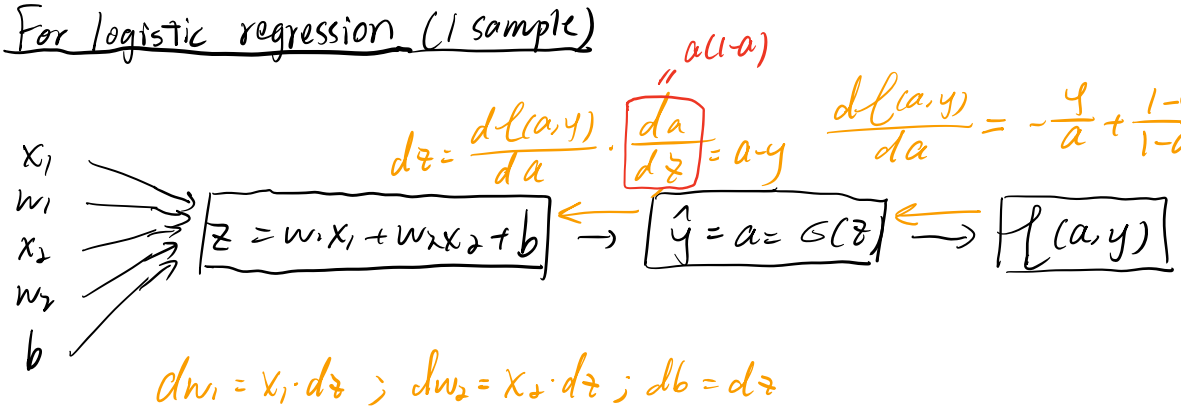
Chain rule & Computation graph:

$J(a,b,c) = 3(a + \underbrace{bc}_{u})$, $v = a + u$



$u = bc$; $v = a + u$; $J = 3v$

For logistic regression (1 sample)



$dw_1 = x_1 \cdot dz$; $dw_2 = x_2 \cdot dz$; $db = dz$

For logistic regression (m sample) — One iteration of batch gradient descent

$J = \frac{1}{m} \sum^{m}$ ∂ L ... (i) (ii).

$$\frac{\partial}{\partial w_1} J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \underbrace{\frac{\partial}{\partial w_1} L(a^{(i)}, y^{(i)})}_{dw_1^{(i)} \text{ — } (x^{(i)}, y^{(i)})}$$

$J=0;\ dw_1=0;\ dw_2=0;\ db=0$ → $dw = np.zeros((n_x, 1))$

For loop I

For $i=1$ to m:

$z^{(i)} = w^T x^{(i)} + b$

$a^{(i)} = \sigma(z^{(i)})$

$J \mathrel{+}= -[y^{(i)} \log a^{(i)} + (1-y^{(i)}) \log(1-a^{(i)})]$

For loop II.

$dz^{(i)} = a^{(i)} - y^{(i)}$

$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$

$dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)}$

$db \mathrel{+}= dz^{(i)}$

assuming $n=2$

$dw \mathrel{+}= x^{(i)} dz^{(i)}$

$J /= m;\ dw_1 /= m;\ dw_2 /= m;\ db /= m$

$w_1 := w_1 - \alpha dw_1$

$w_2 := w_2 - \alpha dw_2$

$b := b - \alpha db$

Vectorization: To get rid of explicit for loops.

Broadcasting Rule:

(m,n) matrix $\pm$ or $\times$ $(1,n) \rightsquigarrow (m,n)$

(m,n) matrix $\pm$ or $\times$ $(m,1) \rightsquigarrow (m,n)$

Vectorizing Logistic Regression

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$$

$X.shape = (n_x, m)$

$$[z^{(1)}\ z^{(2)} \cdots z^{(m)}] = w^T X + [b\ b \cdots b]$$

(1 × m)

$$= [w^T x^{(1)} + b \quad w^T x^{(2)} + b \quad \cdots \quad w^T x^{(m)} + b] = Z$$

$Z = np.dot(w.T, X) + b$ (broadcasting)

$$A = [a^{(1)}\ a^{(2)} \cdots a^{(m)}] = \sigma(Z) \qquad Y = [y^{(1)} \cdots y^{(m)}]$$

$Z = w^T X + b$

$A = \sigma(Z)$

$dZ = A - Y$

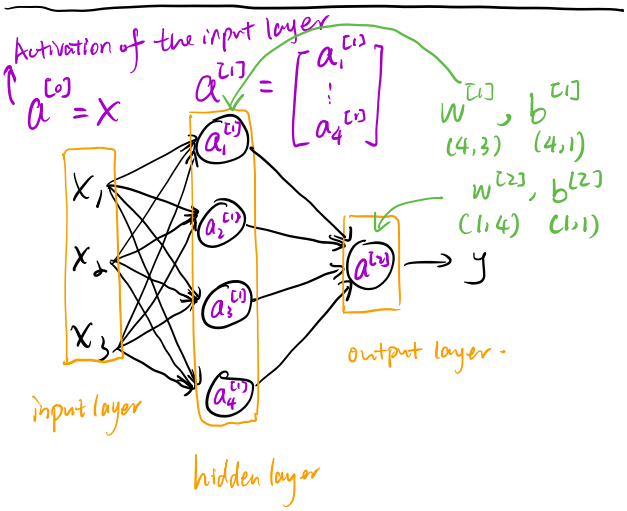
$dw = \frac{1}{m} X dZ^T$

$w := w - \alpha dw$

$b := b - \alpha db$

$db = \frac{1}{m}\,np.sum(dz)$

Neural Network Representation



Vectorizing across multiple examples

$$X = \begin{bmatrix} | & & | \\ x^{(1)} & \cdots & x^{(m)} \\ | & & | \end{bmatrix}$$

n_x, m

(4,m) (4,n_x) (n_x, m) (4,1)

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$

(4, m)

$$A^{[1]} = \sigma(Z^{[1]})$$

(1, m) (1,4) (4, m) (1,1)

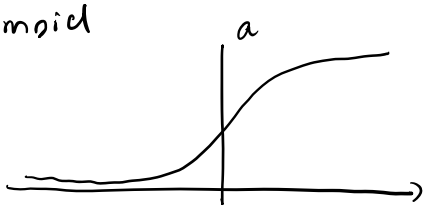
$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$

(1, m)

$$A^{[2]} = \sigma(Z^{[2]})$$

Derivatives of Activation functions

sigmoid

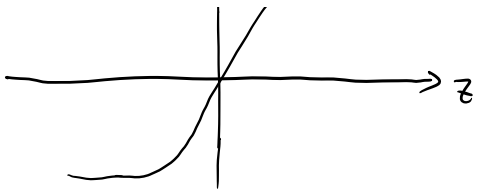


$$g(z) = \frac{1}{1+e^{-z}}$$

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

$$= \frac{e^{z}}{(1+e^{-z})^2} = \frac{e^{-z}}{(1+e^{-z})(1+e^{-z})} = \boxed{g(z)\cdot(1-g(z))}$$

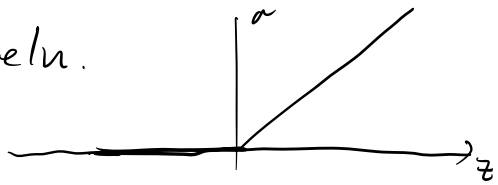
tanh



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

$$g'(z) = 1 - (\tanh(z))^2$$

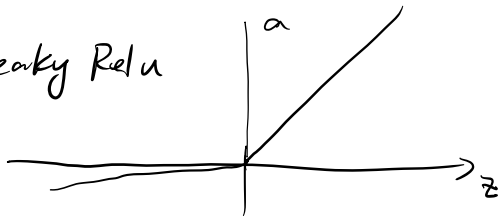
Relu.



$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \end{cases}$$

Leaky Relu



$$g(z) = \max(0.01z, z)$$

$$g'(z) = \begin{cases} 0.01 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \end{cases}$$

Gradient descent for neural Networks

Params: $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}$

$(n^{[1]}, n^{[0]})$ $(n^{[1]}, 1)$ $(n^{[2]}, n^{[1]})$ $(n^{[2]}, 1)$

$n_x = n^{[0]}$, $n^{[1]}$, $n^{[2]} = 1$

n_x: input feature; $n^{[1]}$: Hidden unit; $n^{[2]}$: output

Cost function: $J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}, y)$

Gradient descent:

Repeat { Compute predit $(\hat{y}^{(i)}, i = 1 \cdots m)$

$$dw^{[1]} = \frac{dJ}{dw^{[1]}}; \quad db^{[1]} = \frac{dJ}{db^{[1]}}$$

update $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}$ }

Forward propagation:

$$z^{[1]} = w^{[1]} X + b^{[1]}$$

$$A^{[1]} = g^{[1]}(z^{[1]})$$

$$z^{[2]} = w^{[2]} A^{[1]} + b^{[2]}$$

$$A^{[2]} = \underbrace{\sigma}_{\text{sigmoid}} z^{[2]}$$

Backward propa ation:

$(n^{[2]}, m)$ $dA^{[2]} = -\frac{Y}{A^{[2]}} + \frac{1-Y}{1-A^{[2]}}$

$(n^{[2]}, m)$ $dz^{[2]} = dA^{[2]} \sigma'(z^{[2]}) = A^{[2]} - Y$

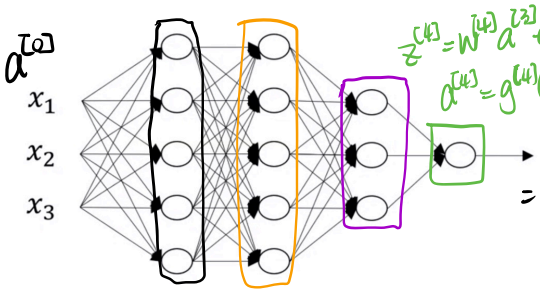
$(n^{[1]}, n^{[2]})$ $(n^{[1]}, m)$ $(m, n^{[2]})$ $dw^{[2]} = \frac{1}{m} dz^{[2]} A^{[1]T}$

$(n^{[2]}, 1)$ $(n^{[2]}, m)$ $db^{[2]} = \frac{1}{m} np.sum(dz^{[2]}, axis=1, keepdims=True)$

$(n^{[1]}, m)$ $(n^{[1]}, n^{[2]})$ $(n^{[2]}, m)$ $dA^{[1]} = W^{[2]T} dz^{[2]}$

$$dZ^{[l]} = dA^{[l]} * g^{[l]\prime}(Z^{[l]}) \quad (n^{[l]}, m)$$
$$dW^{[l]} = \frac{1}{m} dZ^{[l]} X^T \quad (n^{[l]}, n^{[l-1]})$$
$$db^{[l]} = \frac{1}{m} np.sum(dZ^{[l]}, axis=1, keepdims=True) \quad (n^{[l]}, 1)$$

Deep neural network notation



$Z^{[4]} = W^{[4]} a^{[3]} + b^{[4]}$

$a^{[4]} = g^{[4]}(Z^{[4]}) = \hat{y}$

$L = 4$ (# of layers)

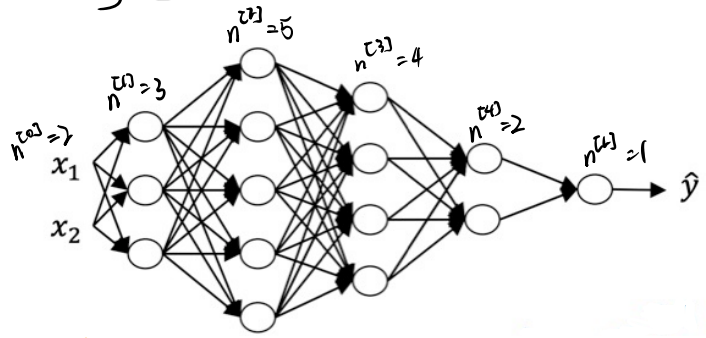
- $n^{[l]}$ = # unit in layer l

$n^{[1]} = 5, n^{[2]} = 3, n^{[3]} = n^{[L]} = 1$ (output unit), $n^{[0]} = n^{[x]} = 3$

- $a^{[l]} = g^{[l]}(Z^{[l]})$
- $W^{[l]}$ = weights for $Z^{[l]}$

$$Z^{[1]} = \begin{bmatrix} | & | & & | \\ z^{[1](1)} & z^{[1](2)} & \cdots & z^{[1](m)} \\ | & | & & | \end{bmatrix} \quad Z^{[1]}.shape = (5, m)$$

Getting matrix dimension right



$(3,1) \leftarrow (3,2) \quad (2,1)$

$$Z^{[1]} = W^{[1]} \cdot x + b^{[1]}$$

$(n^{[1]}, m) \quad (n^{[1]}, n^{[0]}) \quad (n^{[0]}, m)$

$W^{[1]}: (n^{[1]}, n^{[0]}) = (3, 2)$

$W^{[2]}: (n^{[2]}, n^{[1]}) = (5, 3)$

$W^{[3]}: (4, 5)$

$W^{[4]}: (2, 4)$

$W^{[5]}: (1, 2)$

$dW^{[l]} / W^{[l]}: n^{[l]}, n^{[l-1]}$

$db^{[l]} / b^{[l]}: n^{[l]}, 1$

$Z^{[l]} / A^{[l]}: n^{[l]}, m$

Forward and Backward functions

For layer l: $W^{[l]}, b^{[l]}$

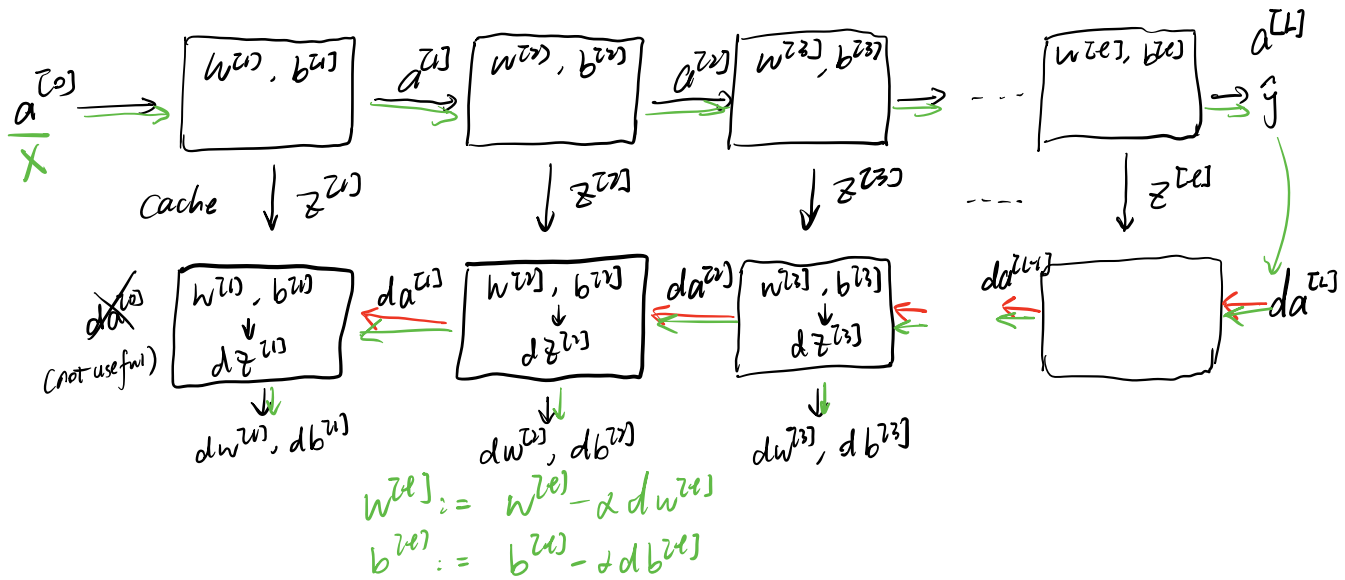
Forward: Input $a^{[l-1]}$, Output $a^{[l]}$

$$Z^{[l]} = W^{[l]} \cdot a^{[l-1]} + b^{[l]} \rightarrow \text{cache } Z^{[l]}$$

$a^{[l]} = g^{[l]}(z^{[l]})$

Backward: Input $da^{[l]}$, Output $da^{[l-1]}$

$z^{[l]}$ $dw^{[l]}, db^{[l]}$



$w^{[l]} := w^{[l]} - \alpha\, dw^{[l]}$

$b^{[l]} := b^{[l]} - \alpha\, db^{[l]}$

$dz^{[L]} = A^{[L]} - Y$

$dw^{[L]} = \frac{1}{m} dz^{[L]} A^{[L-1]T}$

$db^{[L]} = \frac{1}{m}\, np.sum(dz^{[L]}, axis=1, keepdims=True)$

$A^{[L-1]} = W^{[L]T} dz^{[L]}$

$dz^{[L-1]} = A^{[L-1]} * g'^{[L-1]}(z^{[L-1]})$

$\vdots$

$dz^{[1]} = \underbrace{W^{[2]T} dz^{[2]}}_{A^{[1]}} * g'^{[1]}(z^{[1]})$

$dW^{[1]} = \frac{1}{m} dz^{[1]} A^{[0]T}$

$db^{[1]} = \frac{1}{m}\, np.sum(dz^{[1]}, axis=1, keepdims=True)$

Course 2: Improving DNN - Hyperparameter tuning, Regularization and Optimization

Train / Dev / Test sets

- If sample size is small, could do 70/30 or 60/20/20 split
- If sample size is very large (e.g., $n > 1M$), could do 99/0.6/0.4 etc.
- Make sure dev/test sets come from the same distribution.
- Not having a test set might be okay

Bias / variance Diagnose

compared to human performance (~0%)

Train error:	1%	15%	15%	0.5%
Test error:	11%	16%	30%	1%
	high variance.	high bias	high bias & high variance.	low bias low variance.

* Human error $\approx$ 0% $\Rightarrow$ compared to train error to decide bias.

Optimal (Bayes) error: 0%

$\Rightarrow$ To diagonalize whether to algorithm's promble comes from bias / variance or both

Basic "Recipe" for machine Learning:

(Bias-Variance tradeoff (not necessarily))

① High bias or not? (training data performance) $\longrightarrow$ try: bigger network, train longer, CNN architecture)

↓ N

② High variance or not? (Dev set performance) $\longrightarrow$ try: More data, Regularization, CNN architecture)

↓

Done!

When high variance : Add regularization to the network.

Logistic regression:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\|w\|_2^2 + \underbrace{\frac{\lambda}{2m}b^2}_{\text{omit}}$$

(Euclidean norm square) $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2$

→ L2 regularization $\frac{\lambda}{2m}\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2$ (Euclidean norm square) $\|\cdot\|_2^2$

L1 regularization $\frac{\lambda}{2m}\|w\|_1 = \sum_{j=1}^{n_x} |w_j|$ ⇒ w will be sparse (a lot 0s)

λ = regularization parameter, decided by dev set.

Neural Network:

$$J(w^{[1]}, b^{[1]}, \cdots, w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} \|w^{[l]}\|^2$$

where $\|w^{[l]}\|^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} (w_{ij}^{[l]})^2$ given $w : (n^{[l]}, n^{[l-1]})$

↳ "Frobenius norm" $\|\cdot\|_F^2$

$$\frac{dJ}{dw^{[l]}} = (\text{from backprop}) + \frac{\lambda}{m} w^{[l]}$$

$$w^{[l]} = w^{[l]} - \alpha\, dw^{[l]} = \underbrace{w^{[l]} - \frac{\alpha\cdot\lambda}{m} w^{[l]}}_{\text{L2: weight decay.}} - \alpha(\text{from backprop})$$

Dropout regularization (Inverted dropout)

illustrate with layer $l=3$

chance of keeping the unit. ↑

Train:

- $d_3 = np.random.rand(a_3.shape[0], a_3.shape[1]) < keep\_prob$
- $a_3 = np.multiply(a_3, d_3)$
- $a_3 = a_3 / keep\_prob$ (scale it up, keep $[a^{[3]}]$ the same.)

Test time: Not use dropout

Intuition: Cannot rely on any one feature, so have to spread out weights.

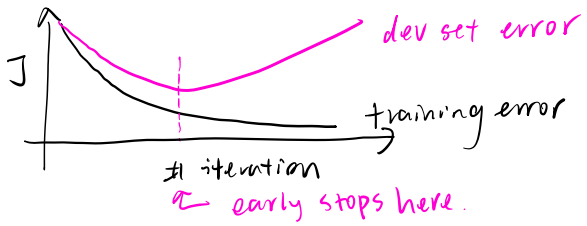
⇒ Shrink weights just like L2

Choosing keep-prob : Smaller keep-prob when for w that has many parameters.

No drop-out (or very close to 1) for input layer.

Other regularization methods

- Data augmentation. — creating more training data by modifying available training samples. (Flip horizontally, rotation)
- Early stopping.

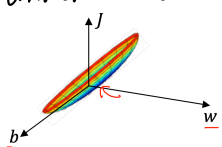


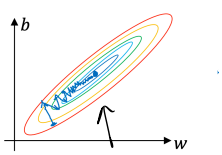
Normalizing Inputs

- Subtract mean with μ
- Normalize variance with σ^2

} use the same set of μ & σ^2 for train & test sets.

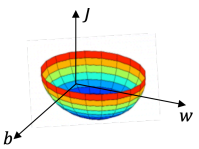
Unnormalized

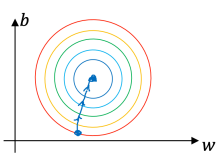




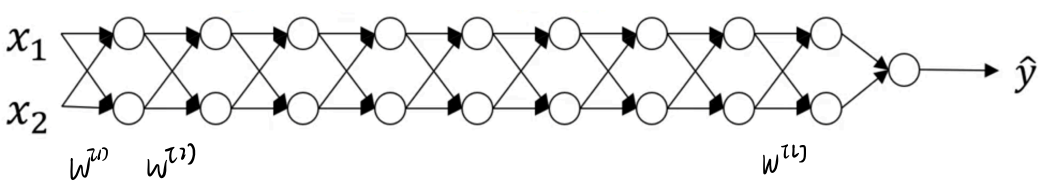
Need small α.

Normalized.





Vanishing / Exploding gradient



Let $g(z) = z$, $b^{[l]} = 0$

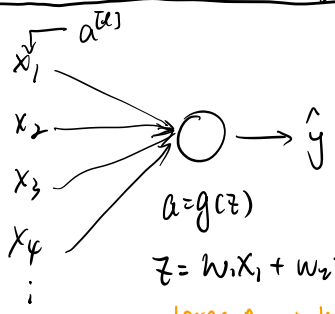
$L-1$ $\quad 1.5^{L-1} \to \infty$

$\hat{y} = w^{[L]} w^{[L-1]} \cdots \underbrace{w^{[2]} \underbrace{w^{[1]} X}_{z^{[1]} = a^{[1]}}}_{=a} \Rightarrow \hat{y} = w^{[L]} \cdot \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix} X \quad 0.5^{L-1} \to 0$

if $w^{[l]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$

Problem: If the weights are bigger/smaller than identity matrix, the activation could vanish / explode in a deep neural network.

Partial Solution: Weight Initialization



large $n \to$ want smaller w.

$Var(w_i) = \frac{1}{n}$ (if $g(z)$ is relu)

$w^{[l]} = np.random.randn(shape) * np.sqrt(\frac{1}{n^{[l-1]}})$

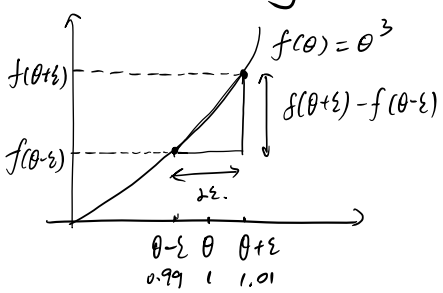
equivalent to setting var of the gaussian dist = 1

Other variants:

tanh $\sqrt{\frac{1}{n^{[l-1]}}}$ Xavier.

$\sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$

Gradient checking



$\frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon} \approx g(\theta) = 3\theta^2$.

$\frac{(1.01)^3 - (0.99)^3}{2(0.01)} = 3.0001 \approx 3.$ approx error: 0.0001

- Take $\underbrace{W^{[1]}, b^{[1]}, \cdots, W^{[L]}, b^{[L]}}_{\text{concatenate} \Rightarrow J(\theta)}$ and reshape into a big vector θ.
- Take $\underbrace{dW^{[1]}, db^{[1]}, \cdots, dw^{[L]}, db^{[L]}}_{\text{concatenate} \Rightarrow}$ and reshape into a big vector $d\theta$

Is $d\theta$ the gradient of $J(\theta)$?

For each i:

$d\theta^{[i]} = \frac{J(\theta_1, \theta_2, \cdots, \theta_i + \varepsilon, \cdots) - J(\theta_1, \theta_2, \cdots, \theta_i - \varepsilon, \cdots)}{2\varepsilon} \approx d\theta^{[i]} = \frac{\partial J}{\partial \theta_i} \quad ? \quad \approx d\theta$

$d\theta_{approx}$ - $\frac{\partial J}{\partial \theta_i} \approx d\theta$ | $d\theta_{approx} \approx d\theta$

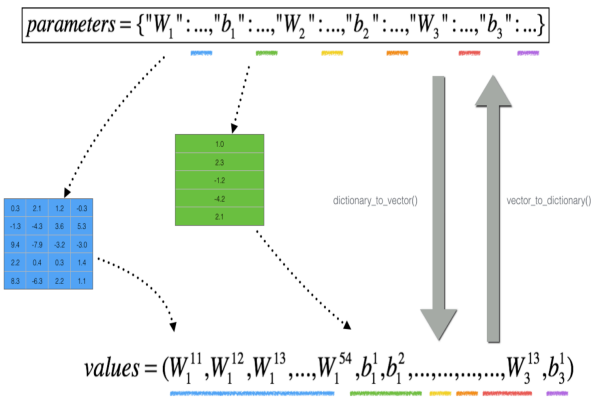
2ε

$\hookrightarrow 10^{-7}$

check $\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2} \approx$ 10^{-7} - great

10^{-5}

10^{-3} - worry.



Batch vs. Mini-batch gradient descent

$X = [\underbrace{x^{(1)}\ x^{(2)}\ x^{(3)} \cdots x^{(1000)}}_{X^{\{1\}}\ (n_x, 1000)} \mid \underbrace{x^{(1001)} \cdots x^{(2000)}}_{X^{\{2\}}\ (n_x, 1000)} \mid \cdots x^{(m)}]$ (n_x, m)

$Y = [\underbrace{y^{(1)}\ y^{(2)}\ y^{(3)} \cdots y^{(1000)}}_{Y^{\{1\}}\ (1, 1000)} \mid \underbrace{y^{(1001)} \cdots y^{(2000)}}_{Y^{\{2\}}\ (1, 1000)} \mid \cdots y^{(m)}]$ $(1, m)$

if $m = 5{,}000{,}000$, then 5000 minibatches $\rightarrow$ minibatch t: $X^{\{t\}}, Y^{\{t\}}$

for epoch = ... {

  for $t = 1, \ldots, 5000$ {

    forwardProp on $X^{\{t\}}$

$$Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$
$$\vdots$$
$$A^{[l]} = g^{[l]}(Z^{[l]})$$

vectorized implementation (1000 example)

Compute cost $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{e} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2000} \sum_{l} \|W^{[l]}\|_F^2$

cost of this minibatch — Indicate # of sample in minibatch — from minibatch $X^{\{t\}}, Y^{\{t\}}$ — # of layers.

    backprop to compute gradients wrt. $J^{\{t\}}$ (using $X^{\{t\}}, Y^{\{t\}}$))

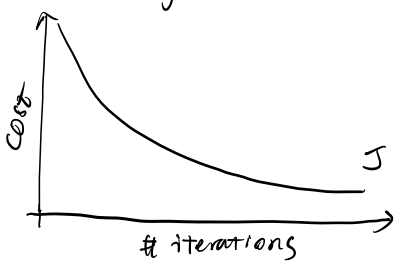
One epoch.

$$W^{[l]} := W^{[l]} - \alpha dW^{[l]} \quad ; \quad b^{[l]} := b^{[l]} - \alpha db^{[l]}$$

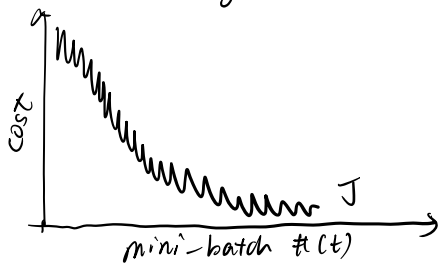
}

}

Batch gradient descent



Mini-batch gradient descent



Choosing mini-batch size:

- If mini-batch size = m: batch gradient descent (Takes too long per iteration) $(X^{\{t\}}, Y^{\{t\}}) = (X, Y)$
- If mini-batch size = 1: Stochastic gradient descent (lose speedup from vectorization) $(X^{\{t\}}, Y^{\{t\}}) = (x^{(1)}, x^{(2)}) \cdots$

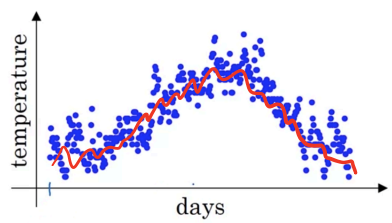
⇒ In between: Fastest learning → Vectorization speed up. → Make progress without processing entire set

⇒ Won't always converge, would bounce around a small region, could be solved by lr decay.

- If small training set ($m \leq 2000$): batch gradient descent.
- If big training set: mini batch sizes: 64-512 (better be 2^n)
- Make sure $X^{\{t\}}, Y^{\{t\}}$ fits CPU/GPU memory.

(*) High β: More weights on previous data points (Averaged across more previous data points). Line will become smoother.

Exponentially weighted averages



$V_0 = 0$

$V_1 = 0.9V_0 + 0.1\theta_1$

$V_2 = 0.9V_1 + 0.1\theta_2$

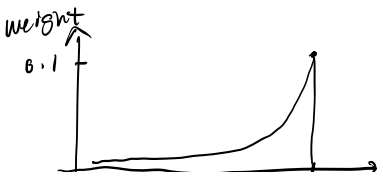
$V_3 = 0.9V_2 + 0.1\theta_3$

$\vdots$

$V_t = 0.9V_{t-1} + 0.1\theta_t$

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

V_t is approximatly averaging over $\approx \frac{1}{1-\beta}$ days' temparature.



$V_\theta = 0$

for $t \in \{0 \cdots 100\}$

$V_\theta := \beta v + (1-\beta)\theta_t$

}

bias correction ⟹

$V_\theta = 0$

for $t \in \{0 \cdots 100\}$

$V_\theta := \beta v + (1-\beta)\theta_t$

$V_\theta := \frac{V_\theta}{1-\beta^t}$

t=100 ... 1-β^t

Gradient descent with momentum

momentum:

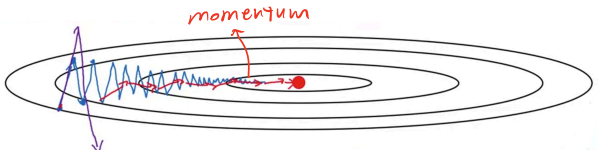
On iteration t:

Compute dw, db on current mini-batch

$V_{dw} = \beta V_{dw} + (1-\beta) dw$. (velocity) ⇒ moving average of dw for each mini-batch.

$V_{db} = \beta V_{db} + (1-\beta) db$. (acceleration) $\beta = 0.9$

$w := w - \alpha V_{dw}$; $b := b - \alpha V_{db}$



RMSprop: (root-mean-square prop)

On iteration t:

Compute dw, db on the current mini-batch

$S_{dw} = \beta_2 S_{dw} + (1-\beta_2)(dw)^2$ (element-wise)

$S_{db} = \beta_2 S_{db} + (1-\beta_2)(db)^2$

$w := w - \alpha \frac{dw}{\sqrt{S_{dw}} + \varepsilon}$; $b := b - \alpha \frac{db}{\sqrt{S_{db}} + \varepsilon}$

* Decay learning rate faster for steep dimension than dimensions with gentler slopes.
* Minimize oscillation
* Faster learning with diverging.

Adam: Adaptive momentum

$V_{dw}, S_{dw}, V_{db}, S_{db} = 0$

On iteration t:

Compute dw, db using current mini-batch

$V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dw$; $V_{db} = \beta_1 V_{db} + (1-\beta_1) db$ ← momentum

$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dw^2$; $S_{db} = \beta_2 S_{db} + (1-\beta_2) db^2$ (item-wise) ← RMSprop.

$V_{dw}^{corrected} = \frac{V_{dw}}{1-\beta_1^t}$; $V_{db}^{corrected} = \frac{V_{db}}{1-\beta_1^t}$

$$S_{dw}^{corrected} = \frac{S_{dw}}{1-\beta_2^t} \; ; \; S_{db}^{corrected} = \frac{S_{db}}{1-\beta_2^t}$$

$$w := w - \alpha \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected}} + \varepsilon} \; ; \; b := b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \varepsilon}$$

Hyperparameter choices:

α: Needs to be tuned

β_1: Moving weighted average of dw / db 0.9

β_2: Moving weighted average of dw^2 / db^2 0.999

ε: 10^{-8}

Learning Rate Decay (Seperate from Adam)

$$\alpha = \frac{1}{1 + \text{decayRate} * \text{epoch\_Num}} \alpha_0$$

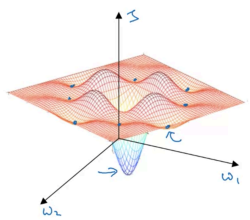
Epoch	α
1	0.1
2	0.06
3	0.05
4	0.04

if $\alpha_0 = 0.2$

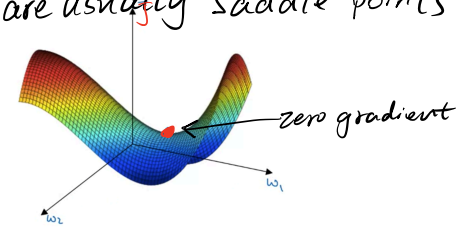
$$\alpha = 0.95^{\text{epoch\_num}} \cdot \alpha_0 \quad \text{or} \quad \alpha = \frac{k}{\sqrt{\text{epoch\_Num}}} \cdot \alpha_0$$

Local Optima in neural networks

used to think:



Local optima in high dimension NN are usually saddle points



⇒ Unlikely to get stuck in a bad local optima.

⇒ Plateaus can make learning slow (Where Adam can help)

Hyperparameter tuning process

α, β_1 (~0.9), β_2, ε, # layers, # hidden units, learning rate decay, mini-batch size ....

→ Try random values : Don't use a grid.

→ Coarse to fine sampling scheme.

Appropriate scale for hyperparameters : log scale

e.g. $\alpha = 0.0001 \dots 1$

$r = -4 * np.random.rand()$ ← $r \in [-4, 0]$ ← sample r uniformly

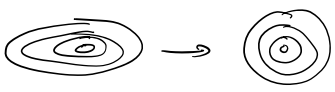
$\alpha = 10^r$ ← $10^{-4} \dots 10^0$

Hyperparameters for exponentially weighted averages :

$\beta = 0.9 \dots 0.999$ ⇒ $r \in [-3, -1]$ $1-\beta = 10^r$ $\beta = 1 - 10^r$

$1-\beta = 0.1 \dots 0.001$

Batch Normalization

Recall : Normalizing inputs to speed up learning 

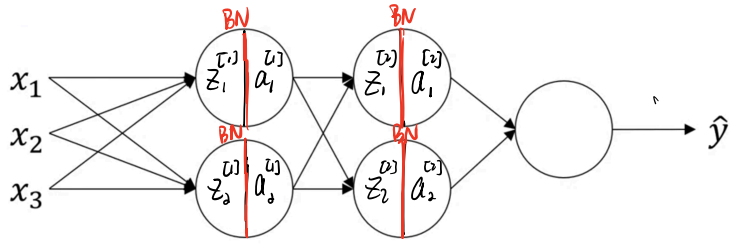
→ Can we normalize $a^{[\ell]}$ (or $z^{[\ell]}$) so as to train $w^{[\ell+1]}$, $b^{[\ell+1]}$ faster ?

Given some intermediate values in NN, $z^{[\ell](1)}, \dots, z^{[\ell](m)}$

$$\mu = \frac{1}{m}\sum_i z^{[\ell](i)} \quad ; \quad \sigma^2 = \frac{1}{m}\sum_i (z^{[\ell](i)} - \mu)^2$$

$$z^{[\ell](i)}_{norm} = \frac{z^{[\ell](i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}} \quad ; \quad \tilde{z}^{[\ell](i)} = \gamma z^{[\ell](i)}_{norm} + \beta$$

γ, β : learnable parameters of model.



$X \xrightarrow{W^{[1]}, b^{[1]}} Z^{[1]} \xrightarrow[\text{Batch Norm (BN)}]{\beta^{[1]}, \gamma^{[1]}} \tilde{Z}^{[1]} \rightarrow a^{[1]} = g^{[1]}(\tilde{Z}^{[1]}) \xrightarrow{W^{[2]}, b^{[2]}} Z^{[2]} \xrightarrow{\beta^{[2]}, \gamma^{[2]}} \tilde{Z}^{[2]} \rightarrow a^{[2]}$

↳ will center at 0 before BN either way, not useful if using BN.

parameters: $w^{[1]}$, ~~$b^{[1]}$~~, $w^{[2]}$, ~~$b^{[2]}$~~, ..., $w^{[L]}$, ~~$b^{[L]}$~~

$\beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \ldots, \beta^{[L]}, \gamma^{[L]}$

$(n^{[1]}, 1)$ $(n^{[2]}, 1)$

$\beta^{[l]} = \beta^{[l]} - \alpha\, d\beta^{[l]}$

Implementing gradient Descent for Batch Normalization:

for t = 1 ... num_mini_batches:

Compute forward prop on $X^{\{t\}}$

In each hidden layer, use BN to replace $z^{[l]}$ with $\tilde{z}^{[l]}$ (have normalized mean/variance)

Use backprop to compute $dw^{[l]}$, $d\beta^{[l]}$, $d\gamma^{[l]}$

update parameters

$$w^{[l]} = w^{[l]} - \alpha\, dw^{[l]}$$
$$\beta^{[l]} = \beta^{[l]} - \alpha\, d\beta^{[l]}$$
$$\gamma^{[l]} = \gamma^{[l]} - \alpha\, d\gamma^{[l]}$$

Can use other optimizers as well.

Why does batch normalization work?

→ Learning on shifting input distribution

→ Reduces the amount that the distribution of the hidden units' activation changes.

→ Mean/variance within each mini-batch add some noise to the value $z^{[l]}$ and $\tilde{z}^{[l]}$ → regularization effect.

Batch Norm at test time

Estimate μ, σ^2 using exponentially weighted average (across mini-batches)

$X^{\{1\}}, X^{\{2\}}, X^{\{3\}}, \ldots$

$\mu^{\{1\}[l]} \quad \mu^{\{2\}[l]} \quad \mu^{\{3\}[l]} \rightarrow \mu$

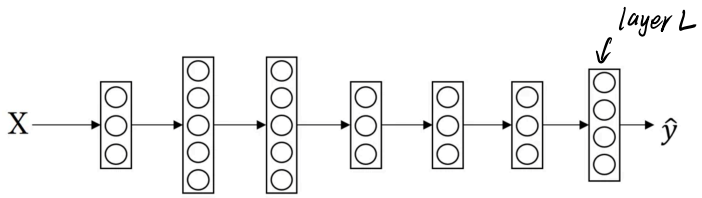
$\sigma^{2\{1\}[l]} \quad \sigma^{2\{2\}[l]} \quad \sigma^{2\{3\}[l]} \rightarrow \sigma^2$

At test (with each testing sample).

$$z_{norm} = \frac{z - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

$$\tilde{z} = \gamma z_{norm} + \beta$$

Multi-class classification (Softmax layer)



$$z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$$

Softmax activation fx: $t = e^{(z^{[l]})}$; $a^{[l]} = \frac{e^{(z^{[l]})}}{\sum_{i=1}^{c} t_i}$; $a_i^{[l]} = \frac{t_i}{\sum_{i=1}^{c} t_i}$

e.g. $z^{[l]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$ $t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}$ $\sum_{i=1}^{c} t_i = 176.3$ $a^{[l]} = \frac{t}{176.3} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix} = \hat{y}$

Loss function:

$$\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{4} y_j \log \hat{y}_j \quad \Rightarrow \text{Make } \hat{y}_j \text{ as big as possible.}$$

$$J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}, y)$$

$Y = [y^{(1)} \; y^{(2)}, \cdots, y^{(m)}]$

$= \begin{bmatrix} 0 & 0 & 1 & \\ 0 & 1 & 0 & \cdots \\ 1 & 0 & 0 & \\ 0 & 0 & 0 & \end{bmatrix}$ One hot

$(4, m)$ if 4 classes in total

$\hat{Y} = [\hat{y}^{(1)} \; \hat{y}^{(2)} \cdots \hat{y}^{(m)}]$

$= \begin{bmatrix} 0.3 & & \\ 0.2 & \cdots & \\ 0.1 & & \\ 0.4 & & \end{bmatrix}$

$(4, m)$

Backprop: $dz^{[l]} = \hat{y} - y$

Class 3: Structuring Machine Learning project

Chain of assumptions in ML:

① Fit training set well on cost function (≈ human level performance) ⇒ bigger network, optimization (e.g., Adam)

↓

② Fit Dev set well on cost function ⇒ Regularization, bigger training set

↓

③ Fit test set well on cost function ⇒ bigger dev set.

↓

④ Performs well in real world. ⇒ change dev set, change cost function

→ Using a single number evaluation metric

- Instead of using precision & recall → use F1 score = $\frac{2}{\frac{1}{P}+\frac{1}{R}}$ "harmonic mean"
- Choosing the "best" model from a bunch. → Speed up iterating

→ Satisficing and Optimizing metric

eg. classifier	Accuracy (optimizing ↑)	running time (satisficing ↑)
A	90%	80ms
B	92%	95ms
C	95%	1500ms

metric:

Maximize Accuracy subject to running time ≤ 100ms

- If N metrics: 1 optimizing; N-1 satisfying.

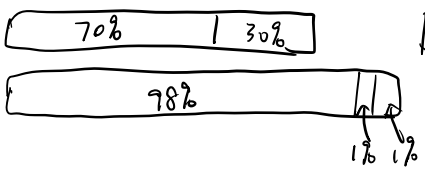
→ Train/dev/test distributions

- Make dev & test set from the same distribution

eg., Regions: US; UK; Other Europe; South America; India; China; Other Asia → randomly shuffle into dev/test

→ Size of dev and test sets

- Old way of splitting data



60% | 20% | 20% when m = 100, 1000, 10000

When m is super large.

→ When to change Dev/Test sets and metrics?

- If doing well on your metric + dev/test set does not correspond to doing well on your application, change your metric and/or dev/test sets.

Summary of bias/variance with human-level performance.

- 0%
- Human level error (proxy for bayes error)
- Training error
- Dev error

"Avoidable bias"

"variance"

bias

The 2 fundamental assumptions of supervised learning.

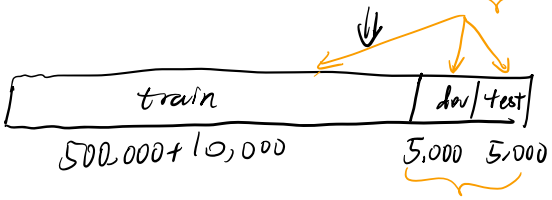
1. You can fit the training set pretty well. ~ low avoidable bias
   - → Train bigger model
   - → Train bigger/better optimization algorithms (momentum; RMSprop ...)
   - → NN architecture/hyperparameters search.
2. The training set performance generalizes pretty well to the dev/test set => More training data/Regularization (Dropout; L2)
   - → More data
   - → Regularization (L2; dropout; data augmentation)
   - → NN architecture/hyperparameters search

When training & testing data do not come from the same distribution:

eg., Speech recognition problem:

Purchased data
Smart speaker control / Voice keyboard
} 500,000

Speech activated mirror } 20,000 ← What we care about



Make sure dev & test come from the same distribution → Setting the aim correctly.

Bias/Variance on mismatched training and dev/test sets

Human level	4%	available bias
Training set error	7%	variance
Training-Dev set error	10%	data mismatch
Dev error	12%	Degree of overfitting to dev set
Test error	12%	

Transfer Learning

image recognition
(x, y) pre-train

(x, y)
radiology image
diagnose : Fine tuning.

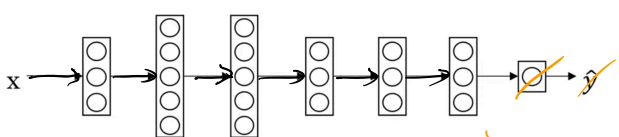


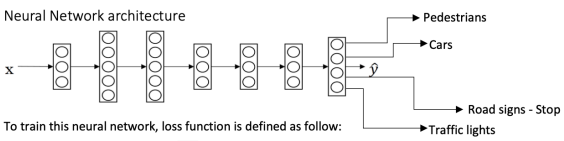
image recognition

$\hat{y}$ radiology diagnosis

$W^{[l]}$ $b^{[l]}$

Multiple task learning

Neural Network architecture



To train this neural network, loss function is defined as follow:

$$-\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{4}\left(y_j^{(i)}\log\left(\hat{y}_j^{(i)}\right)+\left(1-y_j^{(i)}\right)\log\left(1-\hat{y}_j^{(i)}\right)\right)$$

Also, the cost can be compute such as it is not influenced by the fact that some entries are not labeled.
Example:

$$Y=\begin{bmatrix}1 & 0 & ? & ?\\ 0 & 1 & ? & 0\\ 0 & 1 & ? & 1\\ ? & 0 & 1 & 0\end{bmatrix}$$

End-to-end learning

x → audio → MFCC → features → ML → Phonemes → Words → transcript → y

audio → transcript (end-to-end DL)

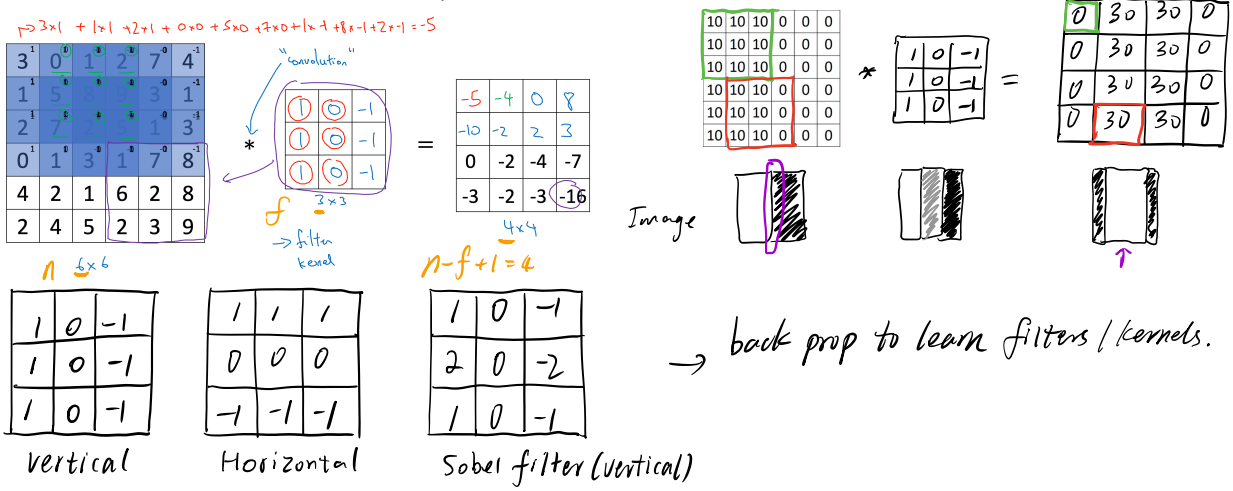
- Pros: Let data speak
  Less hand-designing of components needed
- Cons: May need large amount of data
  Excludes potentially useful hand-designed components

Class 4: Convolution neural Network

Conv
Pool
FC

Computer vision problems: Image classification, Object detection, Neural style transfer

Convolution: Vertical edge detection



→ back prop to learn filters / kernels.

Padding:

To not ① Shrink output

② Throw away info from edges

"Valid" convolution: No padding. $n \times n * f \times f = n-f+1 \times n-f+1$

"Same" convolution : Pad so that output size is the same as the input size.

$p = \frac{f-1}{2}$ (f is usually odd)

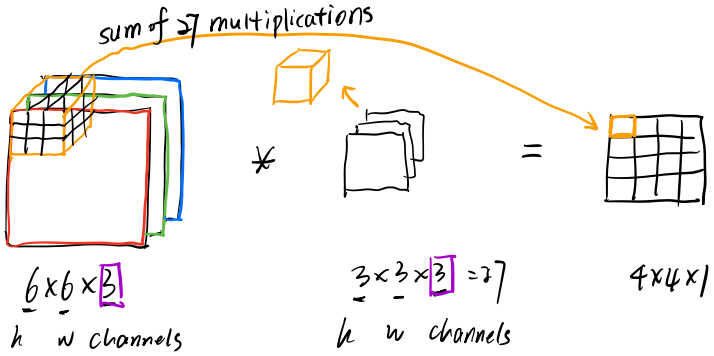
Strided Convolution :

$n \times n \quad * \quad f \times f \longrightarrow \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$

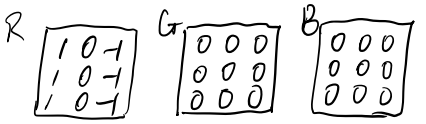
w/ Padding P & Stride S.

$\lfloor z \rfloor = floor(z)$

Convolutions over volumes :

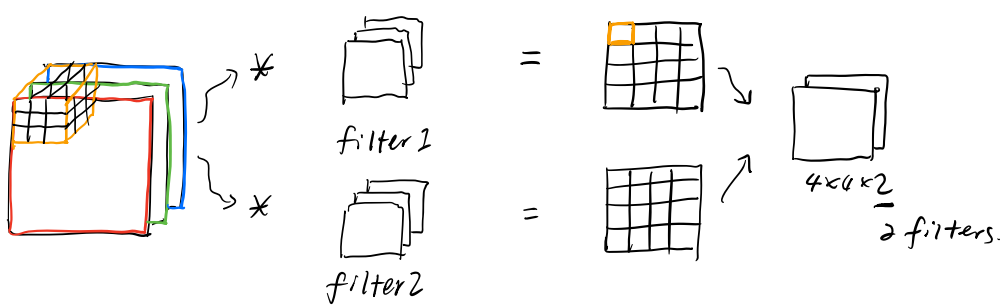


filter examples :



$\rightarrow 3 \times 3 \times 3$: A filter that detects edges in the Red channel of the image.

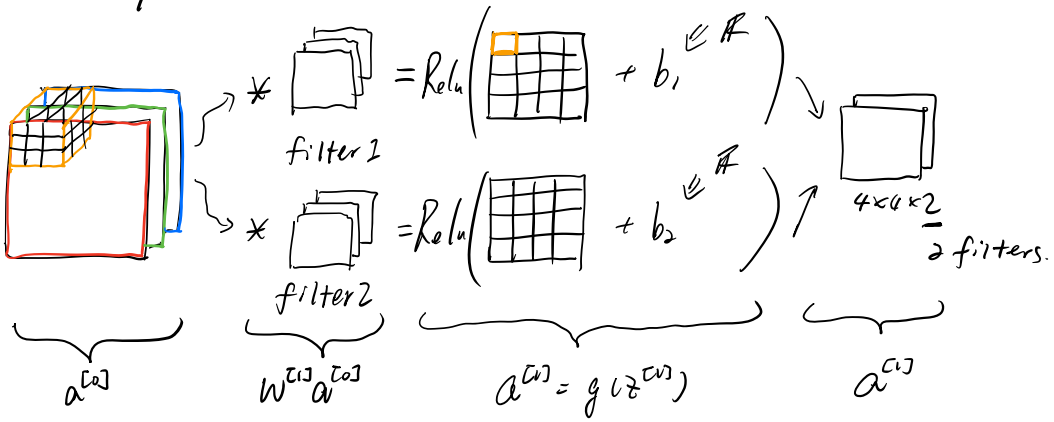
: A filter that detects edges in any color



Summary : $n \times n \times \underline{n_c} \quad * \quad f \times f \times \underline{n_c} \longrightarrow n-f+1 \times n-f+1 \times \underline{n_c'}$

channel/Depth channel/Depth # of filters.

One layer of Convolutional Neural Network.



- If you have 10 filters that are 3×3×3 in one layer of a CNN, # of parameters?

$(3\times3\times3 + \underbrace{1}_{\text{bias}}) \times 10 = 280.$

Notations: If layer l is a convolution layer:

$f^{[l]}$ = filter size

$p^{[l]}$ = padding

$s^{[l]}$ = stride

$n_c^{[l]}$ = number of filters

Each filter is: $f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$

Activations: $a^{[l]} \rightarrow n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$

$A^{[l]} \rightarrow m \times n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$

Weights: $f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$ ← # of filters in layer l

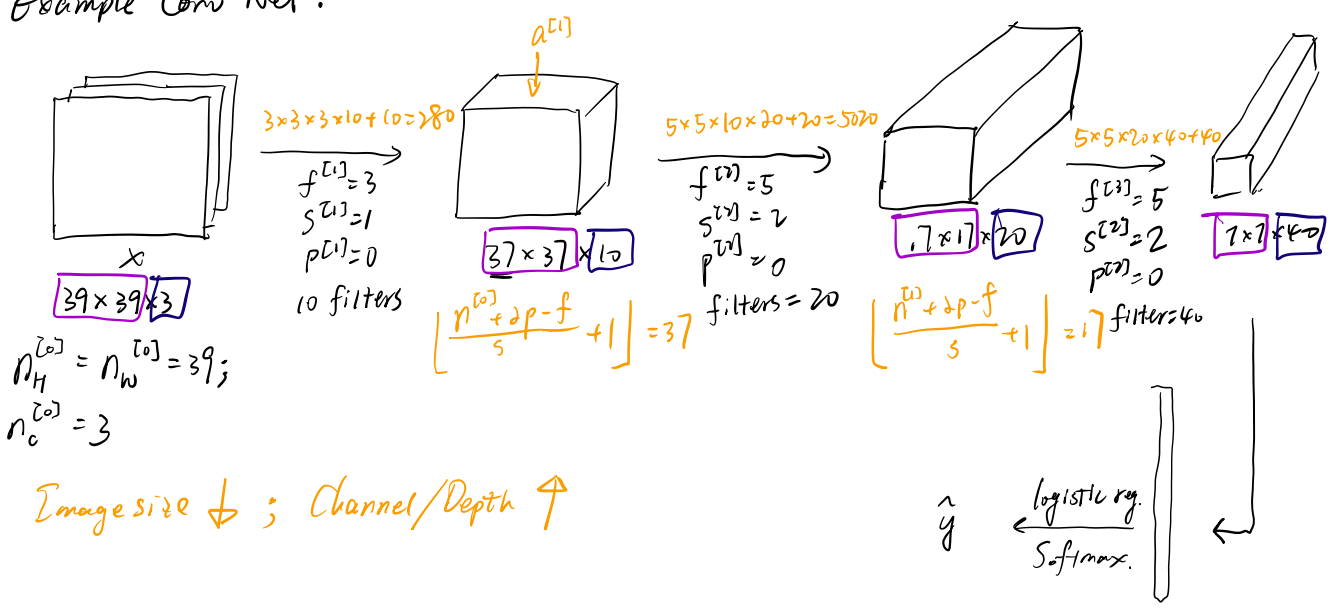
bias: $1 \times 1 \times 1 \times n_c^{[l]}$

Input $= n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$

Output $= n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$

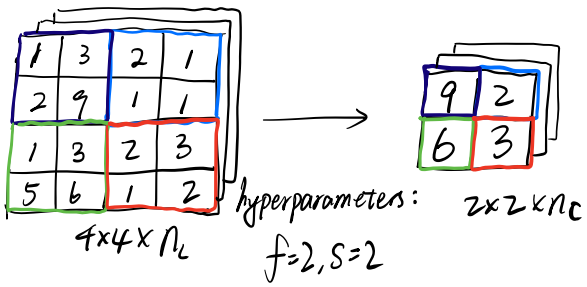
$$n_H^{[l]} = \left\lfloor \frac{n_H^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$$

Example Conv Net:

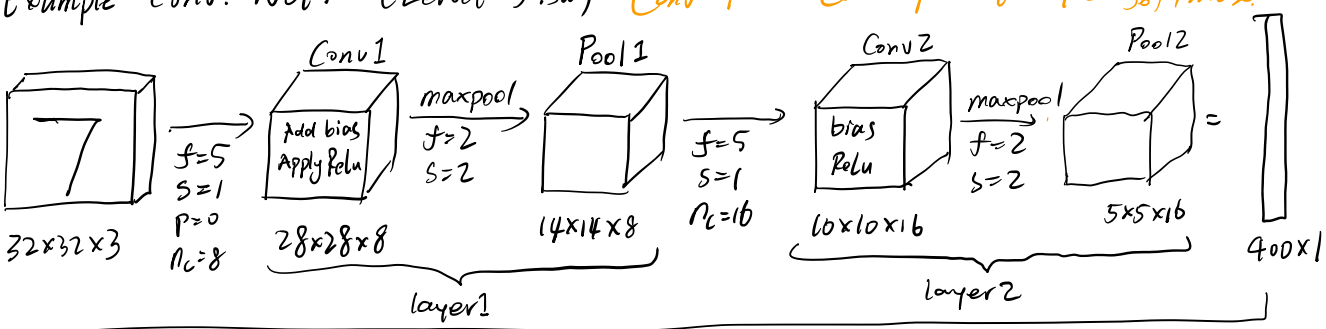


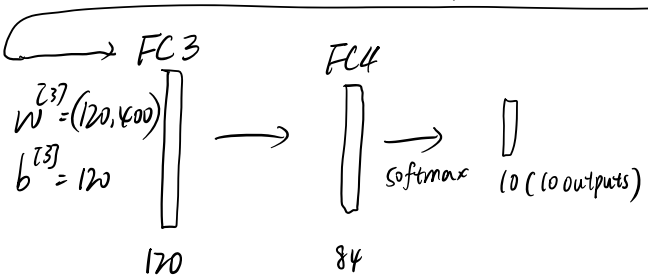
vectorize 196

Pooling layer: Max pooling.



Example Conv. Net: (LeNet-5 ish) Conv - pool - Conv - pool - Fc - Fc - Softmax





Params: layer1:
5x5x3x8+8 = 608
layer2:
5x5x8x16+16 = 3216
layer3:
120x400+120 = 48120
layer4:
84x120+84 = 10164

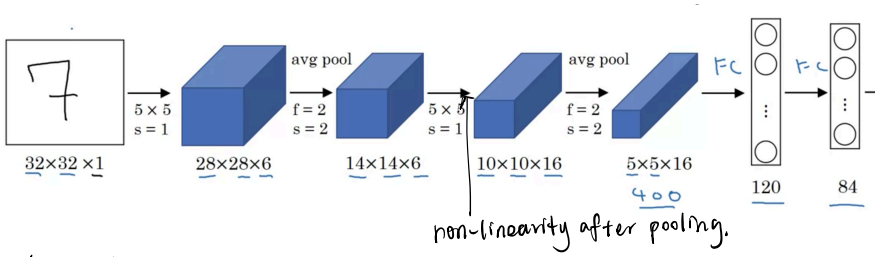
softmax:
10x84+10 = 850

Why Convolutions:

1. Parameter Sharing: A feature detector (such as a vertical edge detector) that's useful in one part of the image is probably useful in another part of the image.
2. Sparsity of Connections: In each layer, each output value depends only on a small number of inputs (prevent overfitting) → translation invariance

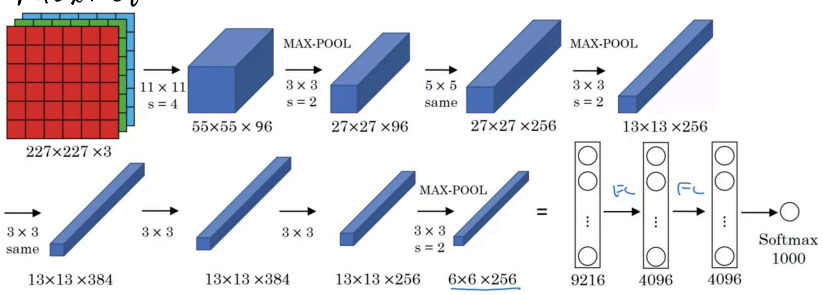
CNN architectures:

LeNet-5 (Document recognition on grayscale image)



- ~60k parameters
- $n_H, n_W \downarrow$; $n_c \uparrow$
- conv-pool-conv-pool-fc-fc-output

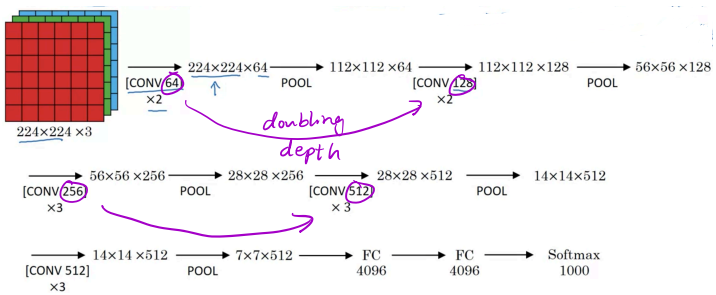
AlexNet



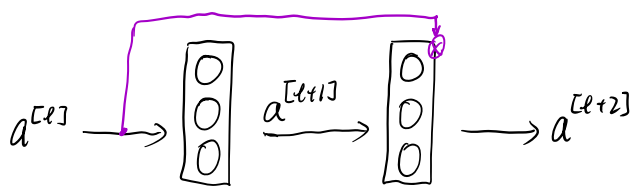
- ~60M parameters.
- ReLu
- Multiple GPU
- Local response normalization (LRN)
  → normalize along depth

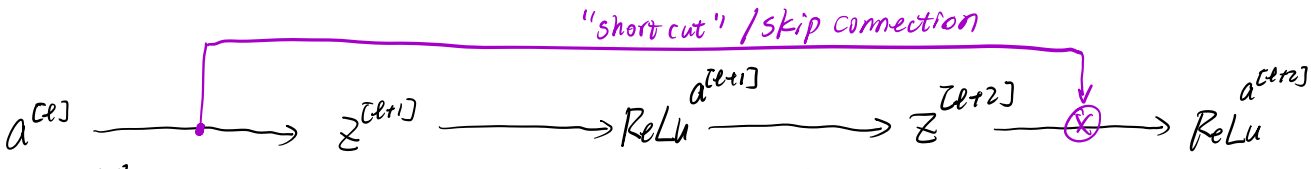
VGG-16 : Conv = 3×3 filter, s=1, padding="same"

Maxpool = 2×2, s=2



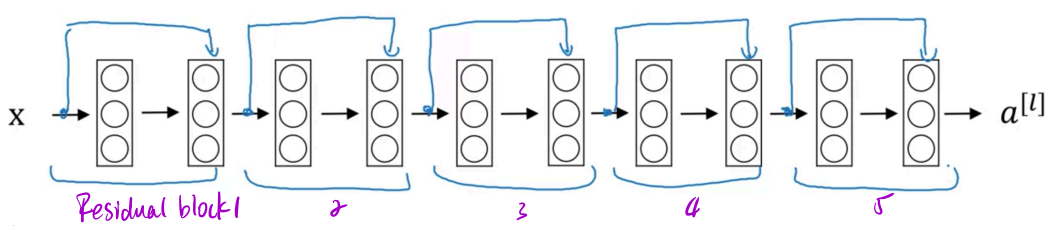
Residual Network (skip connections)

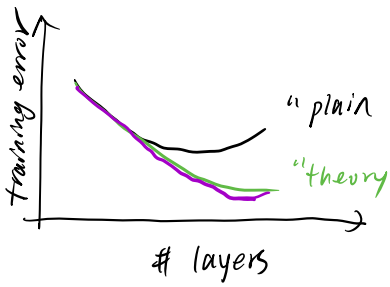




$z^{[l+1]} = w^{[l+1]} a^{[l]} + b^{[l+1]}$ $\quad a^{[l+1]} = g(z^{[l+1]})$ $\quad z^{[l+2]} = w^{[l+2]} a^{[l+1]} + b^{[l+2]}$ $\quad a^{[l+2]} = g(z^{[l+2]})$

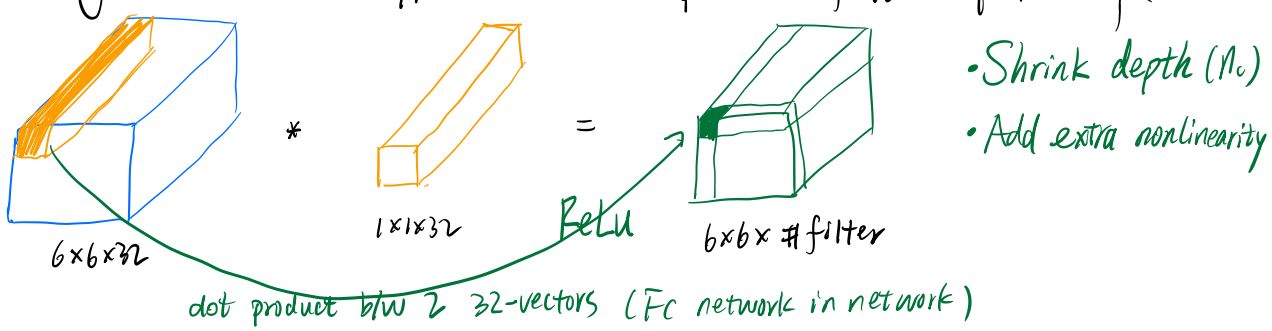
$a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$





Networks in networks and 1x1 convolutions

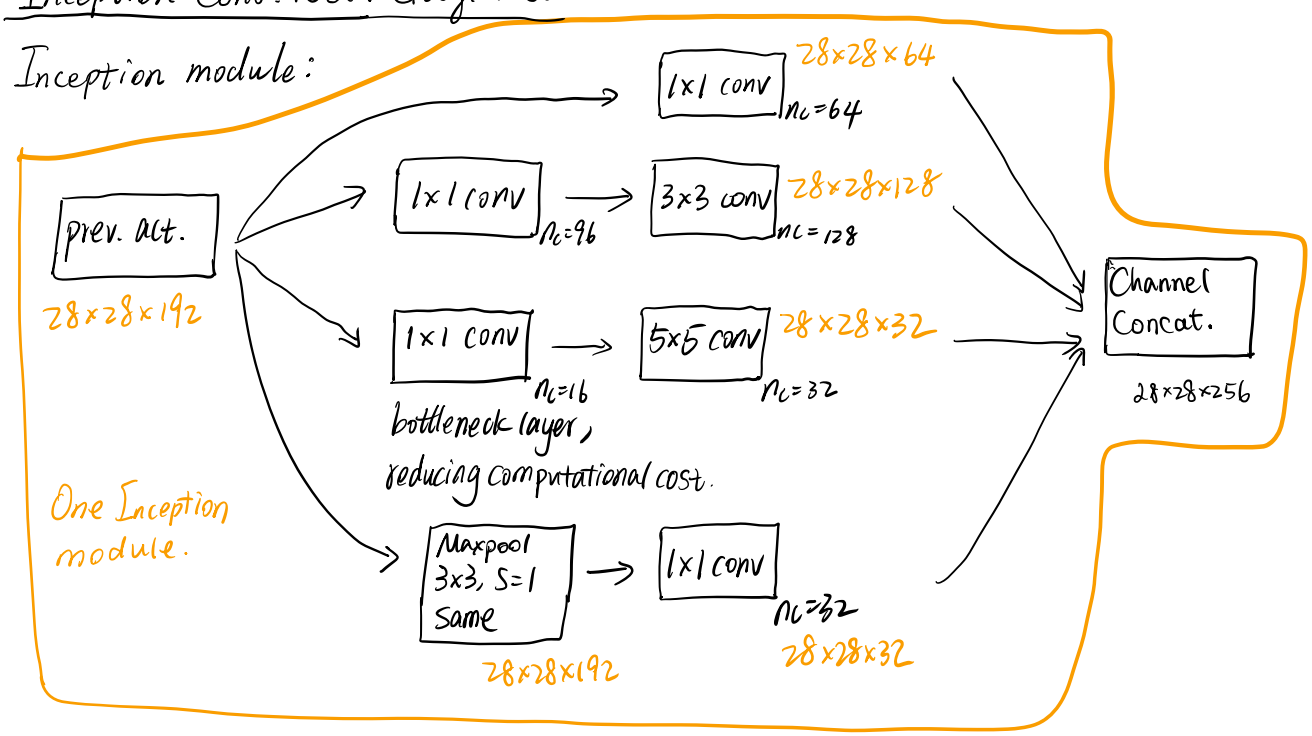
- Make sense when depth > 1
- Having FC networks applied to each position of the input image



- Shrink depth (n_c)
- Add extra nonlinearity

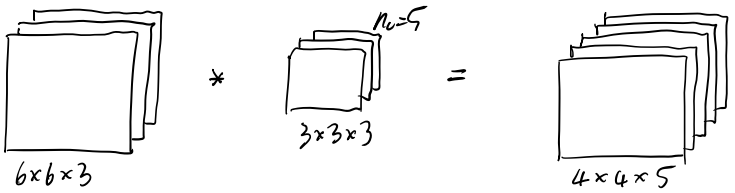
Inception Conv. Net: GoogleNet

Inception module:



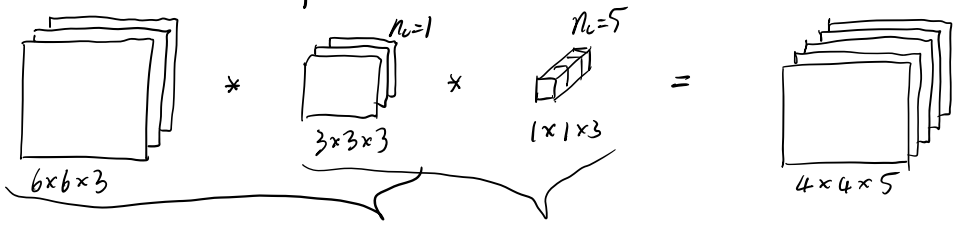
MobileNet

Key idea: Normal vs. depthwise-separable convolutions.



Normal convolution computation cost = # of filter parameters × # of filter positions × n_c

$= 27 \times 16 \times 5 = 2160$



depthwise-separable conv computation cost $= 27 \times 16 + 3 \times 16 \times 5 = 672$

Object classification with localization

Define the target label y : Need to output b_x, b_y, b_h, b_w, class label (1–4)

1. pedestrian
2. car
3. motorcycle
4. background

$$y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

→ Is there an object? 1: If there is 0: background.

} which object

$$\mathcal{L}(\hat{y}, y) = \begin{cases} (\hat{y}_1 - y_1)^2 + (\hat{y}_2 - y_2)^2 + (\hat{y}_3 - y_3)^2 + \cdots + (\hat{y}_8 - y_8)^2 & \text{if } y_1 = 1 \\ (\hat{y}_1 - y_1)^2 & \text{if } y_1 = 0 \end{cases}$$

Object detection : Could have multiple objects in one image

⇒ Sliding window method

↓ closely cropped image

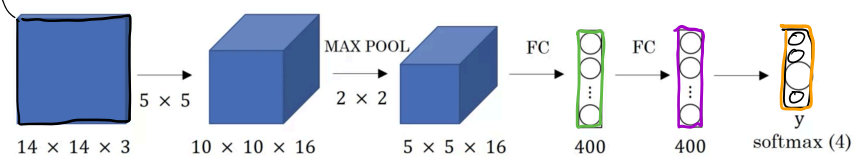
→ Conv. Net → $\hat{y}$ (0, 1)

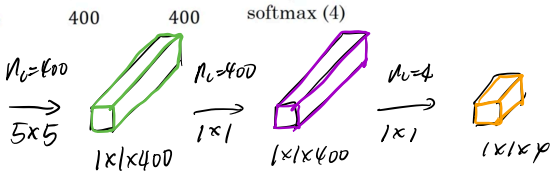


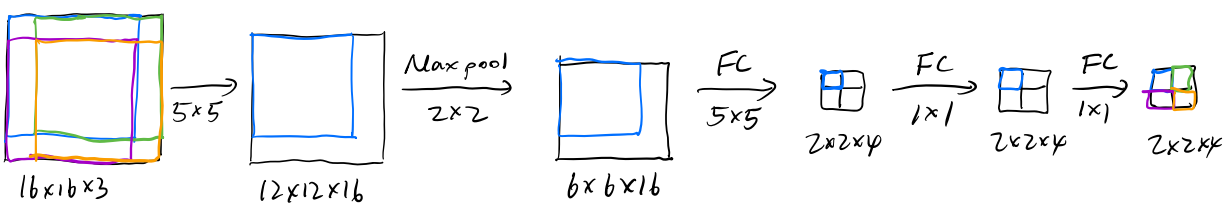
Computationally expensive.

Convolutional Implementation of sliding window

Turning FC layer into convolutional layers

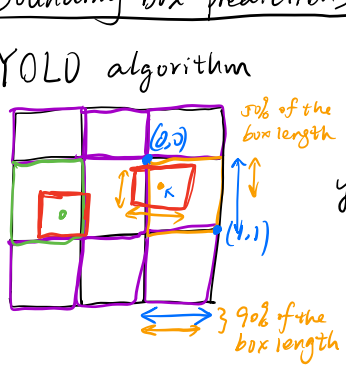




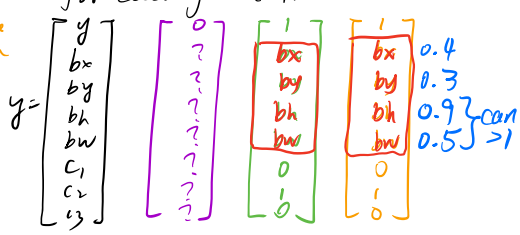


Bounding box predictions.

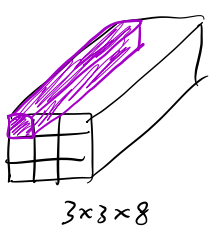
YOLO algorithm



labels for training for each grid cell

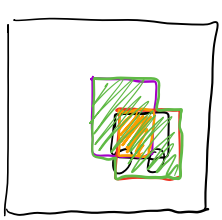


target output volume



object assigned based on mid-point location

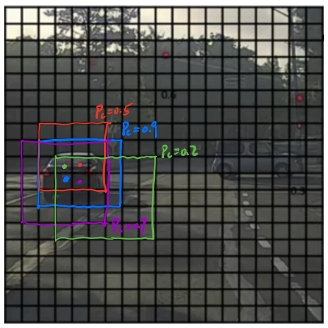
Intersection over union : Evaluating object localization



Intersection over union (IoU) = $\frac{\text{Size of } \square}{\text{size of } \square}$

Crt if IoU ≥ 0.5

Non-max suppression: Make the algorithm detect an object only once.



19x19

Each output prediction is: $\begin{bmatrix} P_c \\ b_x \\ b_y \\ b_h \\ b_w \end{bmatrix}$

- Discard all boxes with $P_c \leq 0.6$
- While there are any remaining boxes:
  - → Pick the box with the largest P_c, output that as pred.
  - → Discard any remaining box with IoU ≥ 0.5 with the box output in the previous step.

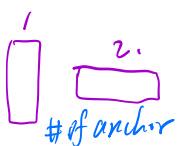
Anchor boxes: Letting 1 grid cell to detect multiple objects.

Anchor box 1: Anchor box 2:


$y = \begin{bmatrix} P_c \\ b_x \\ b_y \\ b_w \\ b_h \\ c_1 \\ c_2 \\ c_3 \\ P_c \\ b_x \\ \vdots \end{bmatrix}$ anchor box 1 (Pedestrain) / anchor box 2 (car)

With two anchor boxes: Each object in training image is assigned to grid cell that contains object's midpoint and anchor box for the grid cell with highest IoU

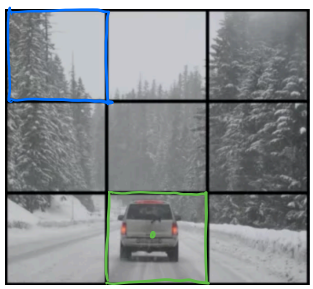
YOLO Algorithm



Training: (3 class labels)

if using 2 anchor boxes: y.shape = (3, 3, 2, 8) or (3, 3, 16)

3x3 grid cell — Pc, box, 3 classes



Predicting

① For each grid call, get 2 predicted bounding boxes, 1 for each anchor.

② Get rid of low P_c boxes

③ For each class (Pedestrian, car, motorcycle) use non-max suppression to generate final pred.

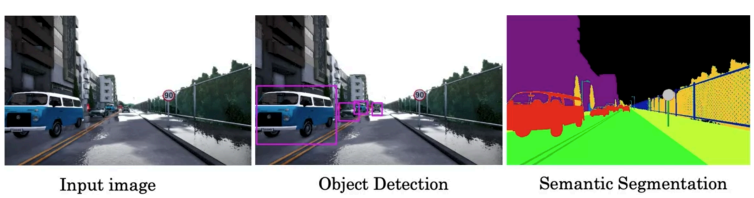
R-CNN: Region proposal CNN.

R-CNN: Propose regions. Classify proposed regions one at a time.
Output label + bounding box

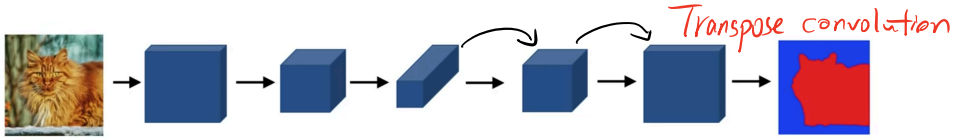
Fast R-CNN: Propose regions. Use convolution implementation of sliding windows to classify all the proposed regions

Faster R-CNN: Use convolutional network to propose regions.

Semantic segmentation: label every pixel

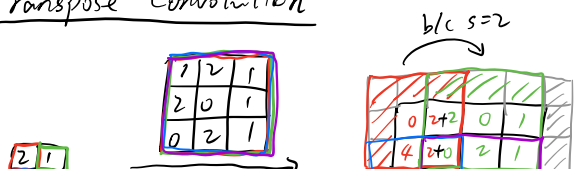


Motivation: locating tumor in a MR scan

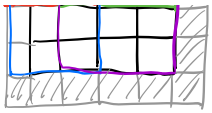


→ Instead of getting smaller and smaller, width and height go back to normal and n_c decreases to # class

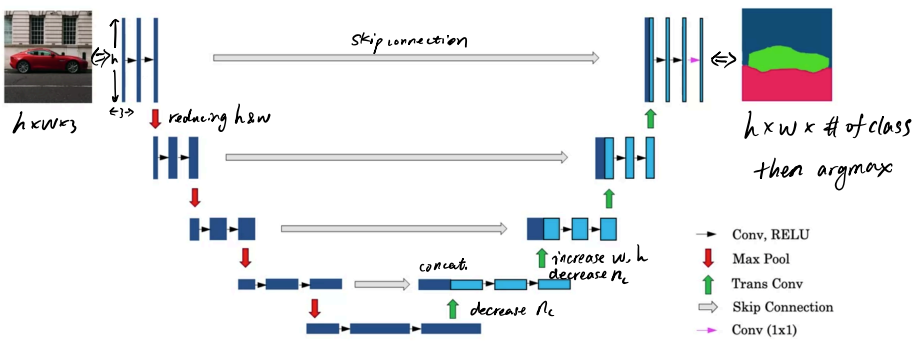
Transpose Convolution



Filter $f \times f = 3 \times 3$
padding $p = 1$
$s = 2$



U-Net Architecture



Face Verification vs. Face recognition

verification: Whether the ID matches the face.

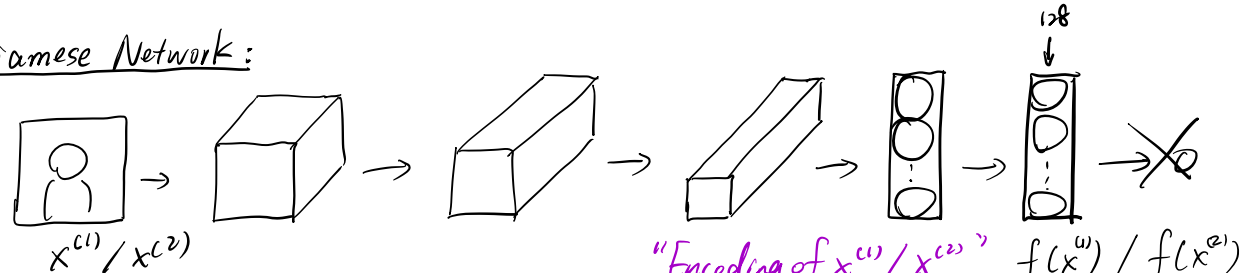
recognition: Output ID if the face is any of the K person in the database.

One-Shot learning: Learning from 1 example to recognize the person again.

→ Learning a "similarity" function: $d(\text{img1}, \text{img2})$ = degree of difference b/w images

if $d(\text{img1}, \text{img2}) \leq \tau$ same person
$> \tau$ different person } verification

Siamese Network:



$$d(x^{(1)}, x^{(2)}) = \| f(x^{(1)}) - f(x^{(2)}) \|_2^2$$

- Parameters of NN define an encoding $f(x^{(i)})$
- Learn parameters so that: if $x^{(i)}$, $x^{(j)}$ are the same person, $\| f(x^{(i)}) - f(x^{(j)}) \|_2^2$ is small

Objective function: Triplet loss (anchor, positive, negative images)

Goal: $\underbrace{\|f(A)-f(P)\|^2}_{d(A,P)} + \alpha \leq \underbrace{\|f(A)-f(N)\|^2}_{d(A,N)}$

$\|f(A)-f(P)\|^2 - \|f(A)-f(N)\|^2 + \alpha \leq 0$ (preventing the NN from encoding everything the same)

Given 3 images: A, P, N

$\mathcal{L}(A,P,N) = \max\left(\underbrace{\|f(A)-f(P)\|^2 - \|f(A)-f(N)\|^2 + \alpha}_{\leq 0}, 0\right)$

$J = \sum_{i=1}^{m} \mathcal{L}(A^{(i)}, P^{(i)}, N^{(i)})$

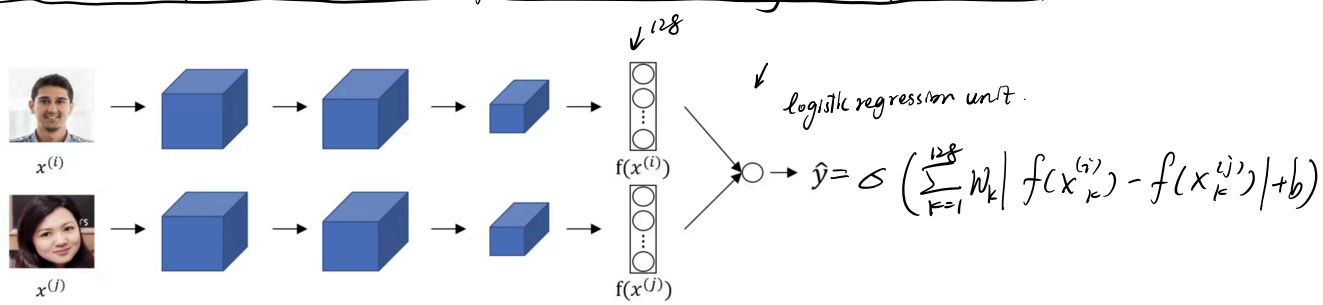
E.g., Training set: 10k pictures of 1k persons (i.e., need multiple images of a person)

⇒ The trained parameters might be able to solve the one-shot learning problem.

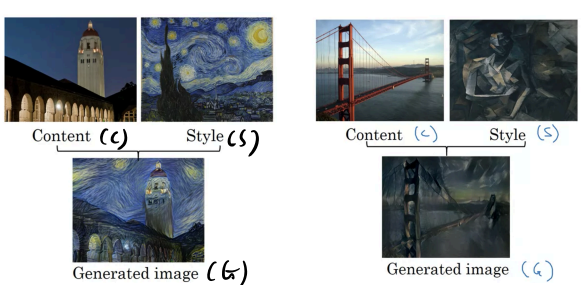
How to choose triplets: If A, P, N are randomly choosen, then

$d(A,P) + \alpha \leq d(A,N)$ is easily satisfied.

Another Approach: Face Verification and Binary classification

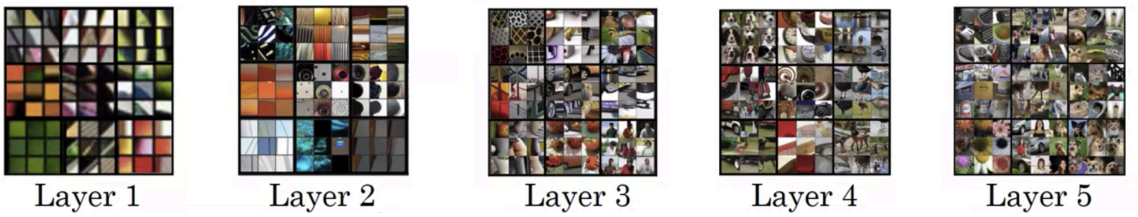


Neural Style Transfer



Visualizing what a deep network is learning:

- Pick a unit in layer 1. Find the nine image patches that maximize the unit's activation; Repeat for other units



Gatys et al., 2015. A neural algorithm of artistic style.

Cost function: Content component + Style component.

$J(G) = \alpha J_{content}(C, G) + \beta J_{style}(S, G)$

similarity b/w contents; similarity b/w styles

1. Initiate G randomly
2. Use gradient descent to minimize $J(G)$

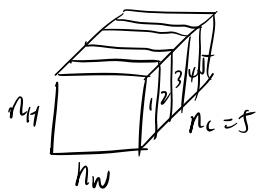
Content cost function:

- Say you use hidden layer l to compute content cost
- Use pretrained ConvNet (e.g., VGG network)
- Let $a^{[l](C)}$ and $a^{[l](G)}$ be the activation of layer l on the images

$J_{content}(C, G) = \| a^{[l](C)} - a^{[l](G)} \|^2$ (element-wise sum of square differences)

Style cost function:

- Say you are using layer l's activation to measure "style"
- Define style as correlation b/w activations across channels at layer l
  → high correlation indicates the high probability of co-occurence of certain features.



- Style matrix:

Let $a^{[l]}_{i,j,k}$ = activation at (i, j, k) (h, w, c). $G^{[l]}$ is $n_c^{[l]} \times n_c^{[l]}$ correlation matrix b/w each pair of channel.

$$G^{[l](S)}_{kk'} = \sum_i^{n_H} \sum_j^{n_W} a^{[l](S)}_{ijk} a^{[l](S)}_{ijk'} \qquad G^{[l](G)}_{kk'} = \sum_i^{n_H} \sum_j^{n_W} a^{[l](G)}_{ijk} a^{[l](G)}_{ijk'}$$

(Gram matrix)

$a^{[l](S)}_{ijk}$: some type of co-variation

$$J^{[l]}_{style}(S, G) = \| G^{[l](S)} - G^{[l](G)} \|_F^2 = \frac{1}{2(n_c^{[l]} n_H^{[l]} n_W^{[l]})^2} \sum_k \sum_{k'} \left(G^{[l](S)}_{kk'} - G^{[l](S)}_{kk'} \right)^2$$

⇓ sum over all layers,

$$J_{style}(S, G) = \sum_l \lambda^{[l]} J^{[l]}_{style}(S, G)$$

Class 5: Recurrent Neural Network

Use cases: Speech recognition; Music generation; Sentiment classification; DNA sequence analysis; Machine translation; Video activity recognition; Name entity recognition.

Notations (using Name entity recognition as an example)

X: Harry Potter and Hermione Granger invented a new spell

$x^{<1>}$ $x^{<2>}$ $x^{<3>}$ ... $x^{<t>}$ ... $x^{<9>}$ $T_x = 9$

y: 1 1 0 1 1 0 0 0 0

$y^{<1>}$ $y^{<2>}$ $y^{<3>}$ ... $y^{<t>}$ ... $y^{<9>}$ $T_y = 9$

$x^{(i)<t>}$: t^{th} element in the sequence of training example i

$T_x^{(i)}$ (examples in the training set can have different length):

Input sequence length for training example i

Representing words:

Vocab: a, and, harry, Potter, ..., Zulu → 1, 367, 4075, 6830, ..., 10,000

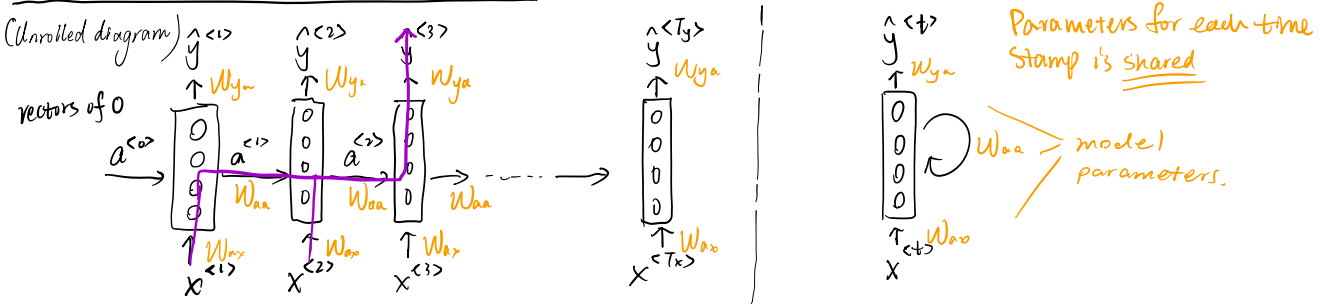
$x^{<1>}$ (harry): [0, 0, 0, ..., 1, 0, ..., 0] (4075)

$x^{<2>}$ (potter): [0, 0, 0, ..., 1, 0] (6830)

one-hot representation.

10,000 vector 10,000 vector 10,000 vector

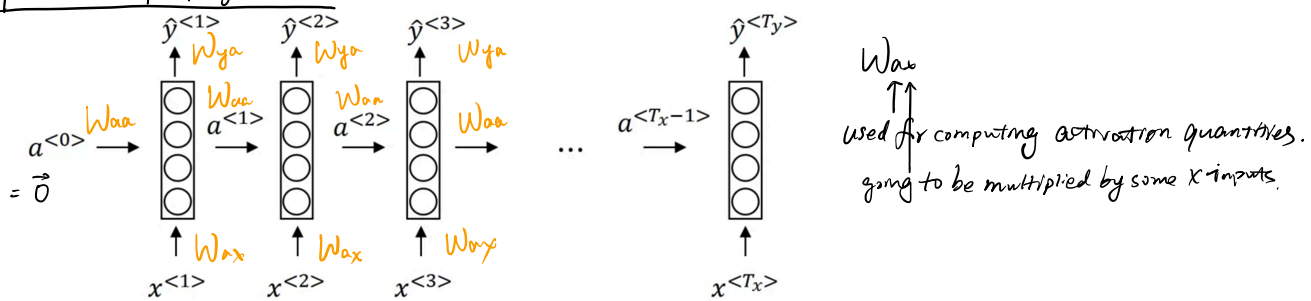
Recurrent Neural Network model ($T_x = T_y$)



Limitation: Prediction for a certain timepoint relies on the information ealier in the sequence only

→ Bidirectional RNN (BRNN)

Forward Propagation:



$a^{<1>} = g(W_{aa}a^{<0>} + W_{ax}x^{<1>} + b_a)$ g = tanh/ReLu

$\hat{y}^{<1>} = g(W_{ya}a^{<1>} + b_y)$ g depends on the type of output.

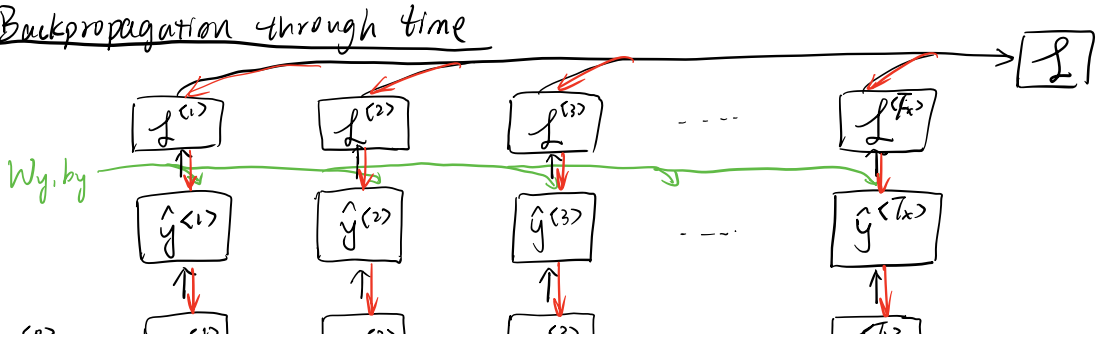
$a^{<t>} = g(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a) \longrightarrow a^{<t>} = g(W_a[a^{<t-1>}, x^{<t>}] + b_a)$

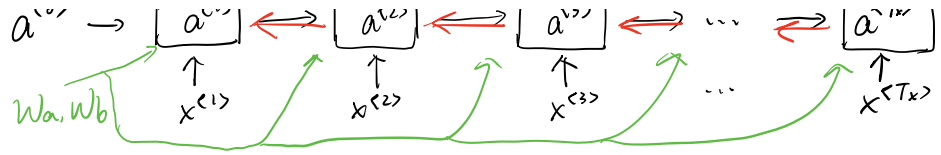
($a^{<t>}$: 100×1; W_{aa}: 100×100; $a^{<t-1>}$: 100×1; W_{ax}: 100×10000; $x^{<t>}$: 10000×1; W_a: 100×10100; $[a^{<t-1>}, x^{<t>}]$: 10100×1)

$W_a = [W_{aa} \mid W_{ax}]$ (100 × 100 | 100 × 10000)

$\hat{y}^{<t>} = g(W_{ya}a^{<t>} + b_y) \longrightarrow \hat{y}^{<t>} = g(W[a^{<t>}] + b_y)$

Backpropagation through time



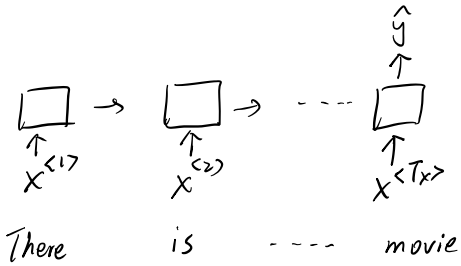


$$\mathcal{L}^{\langle t \rangle}(\hat{y}^{\langle t \rangle}, y^{\langle t \rangle}) = -y^{\langle t \rangle} \log \hat{y}^{\langle t \rangle} - (1-y^{\langle t \rangle}) \log (1-\hat{y}^{\langle t \rangle})$$

$$\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T_y} \mathcal{L}^{\langle t \rangle}(\hat{y}^{\langle t \rangle}, y^{\langle t \rangle})$$

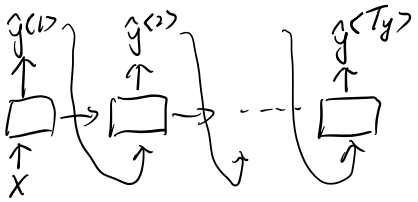
Different types of RNNs (T_x may not equal to T_y)

- Sentiment classification : X = text ; Y = 0/1 or 1 ··· 5



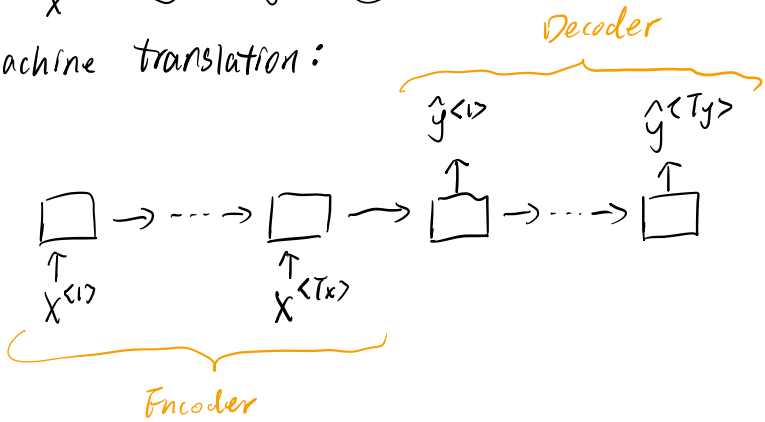
Many-to-one architecture.

- Music generation : X = genre / first note Y = sequence of music notes



One-to-many architecture

- Machine translation :



Many-to-many archi..
($T_x \neq T_y$)

Language Model and sequence generation

- Speech recognition : Model $P(\text{sentence}) = ?$ $P(y^{\langle 1 \rangle}, y^{\langle 2 \rangle}, \cdots, y^{\langle T_y \rangle})$

Training set : Large corpus of english text.

Tokenize :

"Cats average 15 hours of sleep a day." <Eos> (end of sentence)

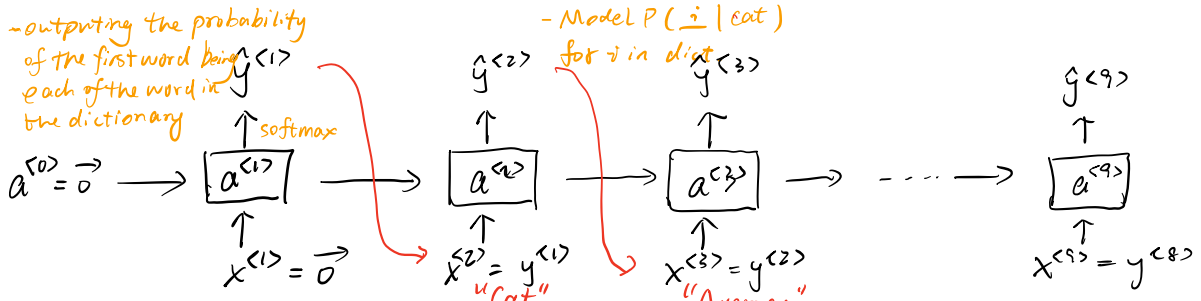
$y^{<1>}$ $y^{<2>}$ $y^{<3>}$ ... $y^{<8>}$ $y^{<9>}$

"The Egyptian Mau is a bread of cat." <Eos>

<UNK> (unknow token if this word is not in the dictionary)

RNN model (For language model)

"Cats average 15 hours of sleep a day."

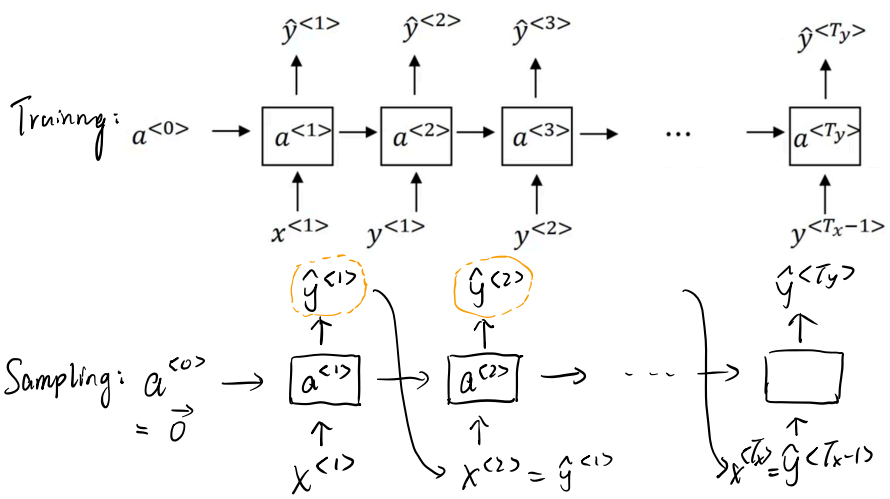


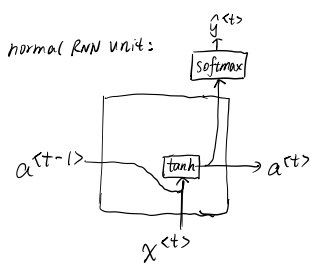
- If dict size is 10002, $\hat{y}^{<1>}$ is a 10002-vector. (<Eos>; <UNK>)

$$\mathcal{L}(\hat{y}^{<t>}, y^{<t>}) = -\sum_i y_i^{<t>} \log \hat{y}_i^{<t>}; \quad \mathcal{L} = \sum_t \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

$$P(y^{<1>}, y^{<2>}, y^{<3>}) = P(y^{<1>}) \cdot P(y^{<2>} | y^{<1>}) \cdot P(y^{<3>} | y^{<1>}, y^{<2>})$$

Sampling a Novel sequence from a trained RNN (Word level)





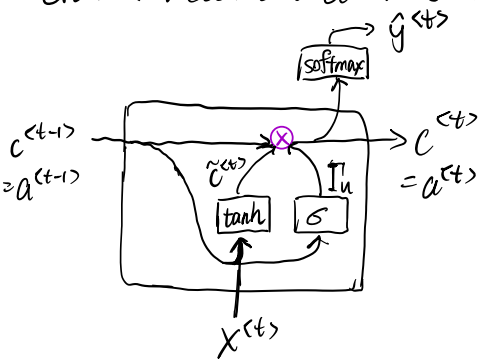
→ P(a) P(aaron) ... P(zulu) np.random.choice

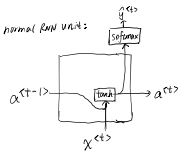
np.random.choice([5,3], p=[0.1, 0, 0.3, 0.6, 0])
→ array([3, 3, 0])

Vanishing gradient with RNN

$\hat{y}^{<t>}$ is mainly influenced by values close to $\hat{y}^{<t>}$ (very local influence)

- Gated Recurrent Unit (GRU) : RNN unit





c = memory cell: same dimension vector

$c^{<t>} = a^{<t>}$

$\tilde{c}^{<t>} = \tanh(W_c[c^{<t-1>}, x^{<t>}] + b_c)$ · Candidate value for memory cell

gamma $\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u)$ · update gate: when to update memory cell w/ $\tilde{c}^{<t>}$

$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1-\Gamma_u) * c^{<t-1>}$

element-wise

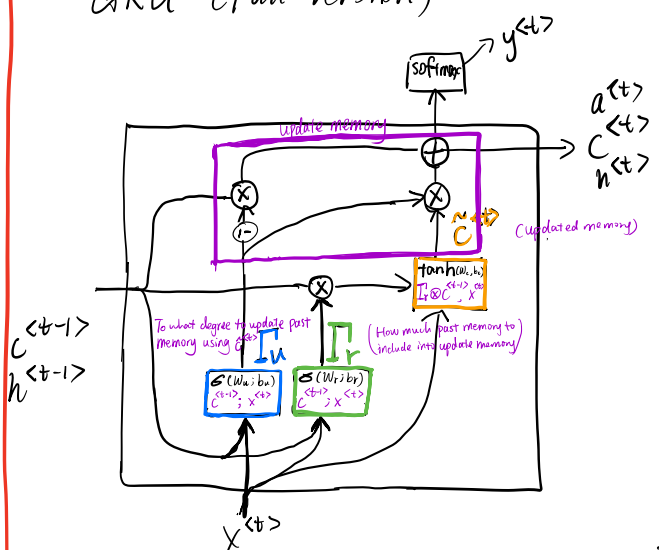
$\Gamma_u^{<t>} = 1$ $\Gamma_u^{<t>} = 0$ $\Gamma_u^{<t>} = 0$ $\Gamma_u^{<t>} = 0$

→ $c^{<t>} = 1$ (for singular)

still remember that the cat is singular

"The cat, which already ate ...., was full"

- GRU (Full version)



$\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u)$

$\Gamma_r = \sigma(W_r[c^{<t-1>}, x^{<t>}] + b_r)$

$\tilde{c}^{<t>} = \tanh(W_c[\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c)$

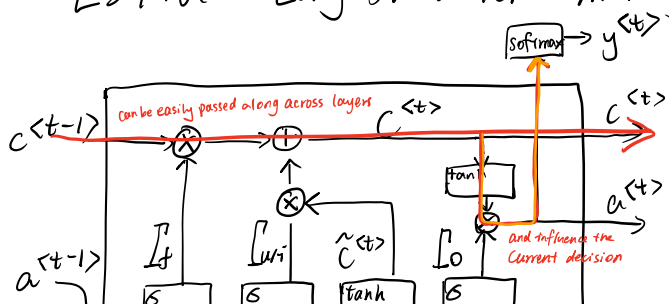
$c^{<t>} = (1-\Gamma_u)\, c^{<t-1>} + \Gamma_u \tilde{c}^{<t>}$

Γ_u: giving memory ($c^{<t-1>}$) and current input ($x^{<t>}$); How much should we update memory, resulting c $c^{<t-1>}$)

Γ_r: The update memory is a combination of past memory ($c^{<t-1>}$) and $x^{<t>}$. How much past memory should be included in the update memory?

How relevant is $c^{<t-1>}$ to compute $c^{<t>}$

- LSTM : Long Short term memory unit.

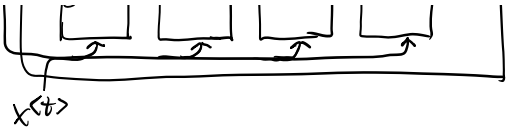


g $\tilde{c}^{<t>} = \tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c)$

i $\Gamma_u = \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_c)$

f $\Gamma_f = \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f)$

o $\Gamma_o = \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o)$



$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * \tilde{c}^{<t-1>}$$

$$a^{<t>} = \Gamma_o * \tanh(c^{<t>})$$

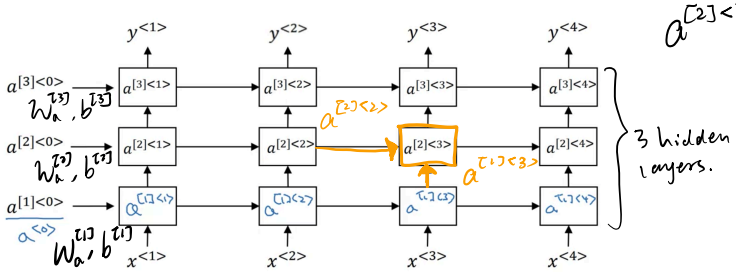
Bidirectional RNN : Getting information from the future.

→ He said, "Teddy bears are on sale"

whether this is a human name depends on future information

→ He said, "Teddy Roosevelt was a great president"

Deep RNNs



layer. <t> t^{th} item in the sequence.

$$a^{[2]<3>} = g(W_a^{[2]}[a^{[2]<2>}, a^{[1]<3>}] + b_a^{[2]})$$

Word Representation

- 1-hot representation : Inner product b/w any 2 1-hot representations = 0

thus correlation matrix doesn't convey any semantic related info.

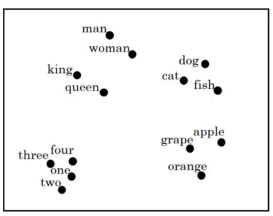
- Featurized representation : Word embedding

embedding matrix

feature \ word	Man (5391)	Woman (9853)	king (4914)	Queen (7157)	Apple (456)	Orange (6257)
Gender	-1	1	-0.95	0.97	0.00	0.01
Royal	0.01	0.02	0.93	0.95	-0.01	0.00
Age	0.03	0.02	0.7	0.69	0.03	-0.02
food	0.09	0.01	0.02	0.01	0.95	0.97
⋮	⋮	⋮				
size						
cost						

300 features

e_{5391} e_{9853} (300-vector)



300D → 2D

sklearn.manifold.tsne

Using Word Embeddings (Like the encoding of faces in the Siamese network)

① Learn/Download word embedding from a large sample of unlabeled data (~1B)

Transfer learning

② Train, say named entity recognition model, using word embedding to represent each word ($\sim 10k$)

Note: Cosine Similarity & Pearson correlation look very similar, But:

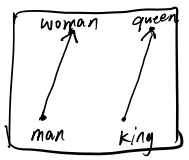
① Cosine similarity computes the similarity b/w 2 samples from the same dist.

② Pearson correlation computes the relationship b/w 2 jointly distributed random variables

Properties of word embeddings

$e_{man} - e_{woman} = \begin{bmatrix} -2 \\ 0 \\ 0 \\ 0 \end{bmatrix}$

$e_{king} - e_{queen} = \begin{bmatrix} -2 \\ 0 \\ 0 \\ 0 \end{bmatrix}$



Can learn Man : Woman as Boy : Girl

Big : Bigger as Tall : Taller

$\rightarrow e_w$

$\underset{w}{\text{argmax}}\ Sim(e_w, e_{king} - e_{man} + e_{woman})$

$Sim(u, v) = \frac{\langle u, v \rangle}{\|u\|_2 \|v\|_2}$

Learning word embeddings (For language model)

I want a glass of orange ?

4343 9665 1 3852 6163 6257

i^{th} word in the corpus.

O_i : One-hot representation of the word.

E : embedding matrix

$e_i = EO_i$: Featurized representation of the word

I $\rightarrow E \cdot O_{4343} \rightarrow e_{4343}$

want $\rightarrow E \cdot O_{9665} \rightarrow e_{9665}$

a $\rightarrow E \cdot O_1 \rightarrow e_1$

glass $\rightarrow E \cdot O_{3852} \rightarrow e_{3852}$

of $\rightarrow E \cdot O_{6163} \rightarrow e_{6163}$

orange $\rightarrow E \cdot O_{6257} \rightarrow e_{6257}$

$\rightarrow$ [ ] $\rightarrow O$ (softmax)

$w^{[1]}, b^{[1]}$ $\quad w^{[2]}, b^{[2]}$

10,000 (if len(corpus) = 10,000)

Use backprop to train E

For constructing embedding matrix

Skip Gram model : Learn context $\rightarrow$ target mapping, targets are ± 10 from context

$O_{context} \cdot E \rightarrow e_{context} \rightarrow O$ (softmax) $\rightarrow \hat{y}$

$p(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10,000} e^{\theta_j^T e_c}}$ $\quad \mathcal{L}(\hat{y}, y) = -\log \hat{y}_T$

slow

To solve this:

Negative sampling

choose context: orange

choose target: juice

"I want a glass of orange juice to go along with my cereal" Corpus = (0000)

X y

Context	word	target
orange	juice	1
orange	king	0
orange	book	0
orange	the	0
orange	of	0

k: orange

negative examples

k≥5-20 for small dataset
=2-5 for large dataset

c: context, t: target, y: 1 or 0

$P(y=1 \mid c,t) = \sigma(\theta_t^T e_c)$

$O_c \cdot E \rightarrow e_c$ (10000,1)

$\theta^T e_c$ (10000×10000)

10000 logistic regression classifiers

Turning 10000 way softmax to 10000 logistic regressions

GloVe (global vector for word representation) model

X_{ij} = # times j appears in context of i

minimize $\sum_{i=1}^{10000} \sum_{j=1}^{10000} f(X_{ij})(\theta_i^T e_j + b_i + b_j' - \log X_{ij})^2$

weighting term measures how related is i & j

Sentiment classification problem

"The dessert is excellent" → ★★★★

The	O_{8928}	E →	e_{8928}
dessert	O_{2468}	E →	e_{2468}
is	O_{4694}	E →	e_{4694}
excellent	O_{3180}	E →	e_{3180} (300D)

Avg/Sum (300D) → w/b softmax (1star – 5star) → $\hat{y}$

But ignore word order. (e.g., "completely lacking in good taste")

can be trained on a much larger dataset.

Many-to-one architecture:

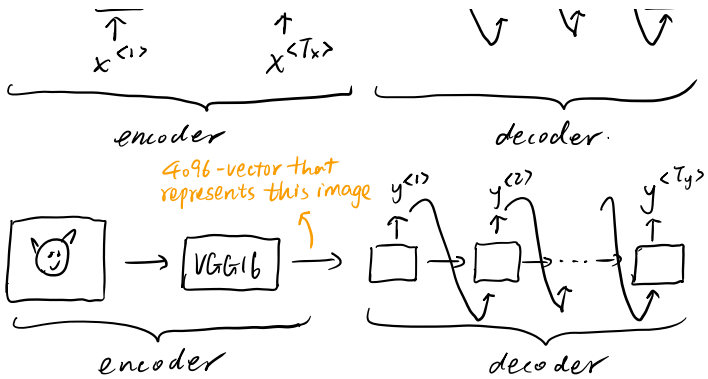
$a^{<0>}$ → $a^{<1>}$ → $a^{<2>}$ → $a^{<3>}$ → $a^{<4>}$ --- $a^{<10>}$ → softmax → $\hat{y}$

e_{1825}	e_{4966}	e_{4427}	e_{3882}	---	e_{337}
completely	lacking	in	good	---	ambience

Sequence to sequence model (e.g. Machine translation / Image Captioning)

output a vector that represents input sentence.

$a^{<0>}$ → ... → $y^{<1>}$ → $y^{<2>}$ → ... → $y^{<Ty>}$ (until it hits EOS token)

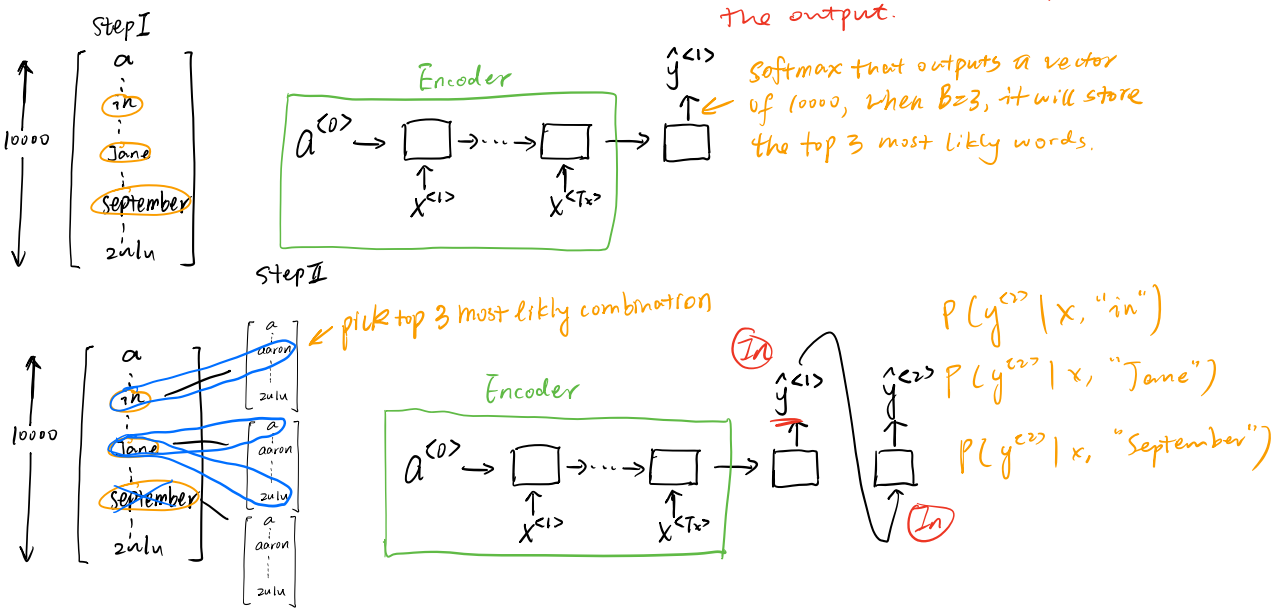


Picking the most likely sentence

- In a regular language model, we train a RNN to model $P(y^{<1>}, y^{<2>}, \dots, y^{<T_y>})$, and then randomly sample $\hat{y}$ from this distribution.
- In the case of machine translation, the encoder-decoder model $P(y^{<1>} \dots y^{<T_y>} \mid x)$, but we do not want to randomly sample our output from this distribution.

Instead $\underset{y^{<1>} \dots y^{<T_y>}}{\text{argmax}} P(y^{<1>}, \dots, y^{<T_y>} \mid x)$ ← Find y that maximize this conditional joint prob.

Beam Search algorithm (B=3 ; beam width) : At each step, instantiate 3 copies of the network to evaluate partial sentence of the output.



- Length Normalization:

$\underset{y}{\text{argmax}} \prod_{t=1}^{T_y} P(y^{<t>} \mid x, y^{<1>} \dots y^{<T_y-1>})$ → ① Can be too small & ② long sentence would have

small probability.

$$\arg\max_y \frac{1}{T_y^{\alpha}} \sum_{t=1}^{T_y} \log P(y^{<t>} | x, y^{<1>} \cdots y^{<T_y-1>})$$

- Large B: better result, slower.
  Small B: worse result, faster.

encoder + decoder.

Error analysis in Beam search: Problem due to beam search or RNN model?

e.g., Jane visits Africa in September (y^*)
Jane visited Africa last September ($\hat{y}$) → Decide whether the problem is the beam width (B)?

→ Test $p(y^*|x)$ vs. $p(\hat{y}|x)$
↳ Meaning first pass X through encoder, then use decoder softmax outputs to compute $p(y^*|x)$ & $p(\hat{y}|x)$

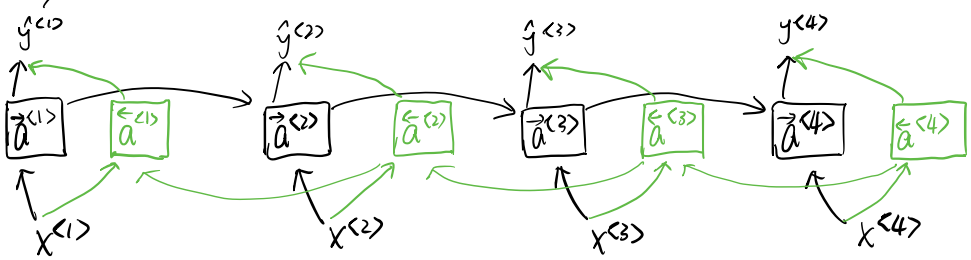
- If $p(y^*|x) > p(\hat{y}|x)$: problem with beam width.
- If $p(y^*|x) \leq p(\hat{y}|x)$: RNN model is at fault. (After length normalization)

Attention: Improving sequence to sequence model

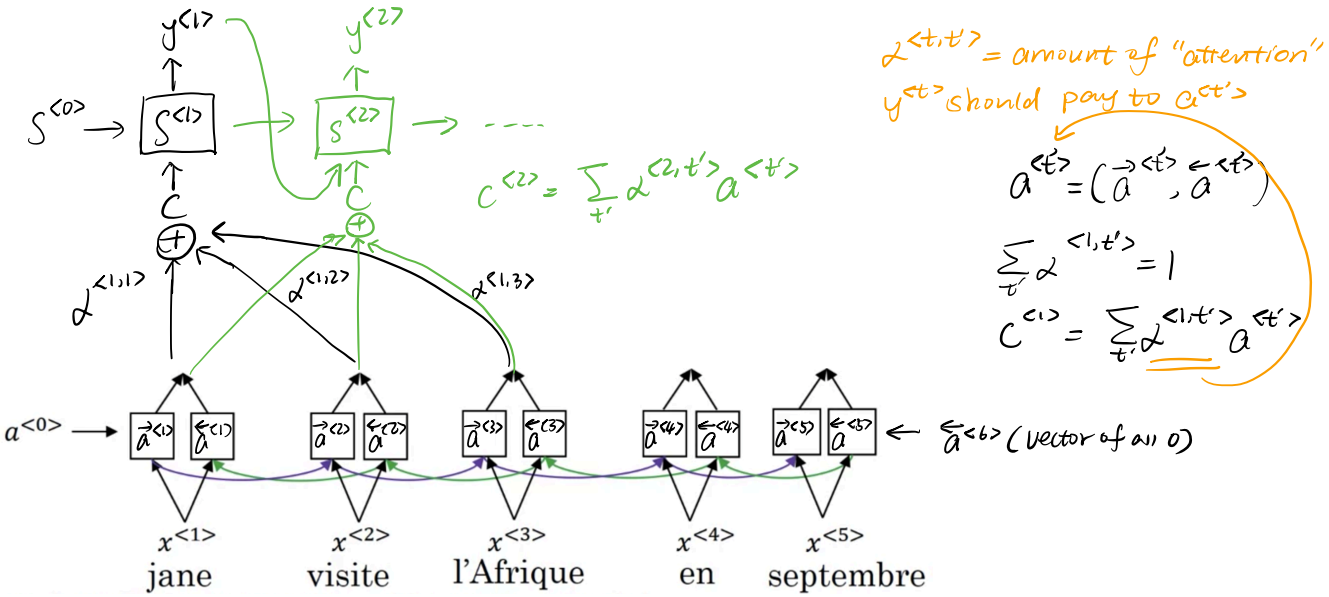
- In the case of machine translation, the encoder needs to "remember" the whole sentence before passing it to ther decoder, which is not the case for human.
- With the Attention model, the RNN could look at part of the sentence at each time.

Recall BRNN: First compute all hidden layer activations, then make predictions

$$\hat{y}^{<t>} = g(W_y[\vec{a}^{<t>}, \overleftarrow{a}^{<t>}] + b_y)$$



Attention model:



- Computing attention $\alpha^{<t,t'>}$

$\alpha^{<t,t'>}$ = amount of attention $y^{<t>}$ should pay to $a^{<t'>}$

$$\alpha^{<t,t'>} = \frac{\exp(e^{<t,t'>})}{\sum_{t'=1}^{T_x} \exp(e^{<t,t'>})}$$

$s^{<t-1>}$, $a^{<t'>}$ → $e^{<t,t'>}$

Transformer Network Intuition : Attention + CNN.

Vaswani et al. 2017 Attention Is All You Need.

→ Self Attention (Machine Translation as an example)

- Calculate for each input word attention-based vector representation

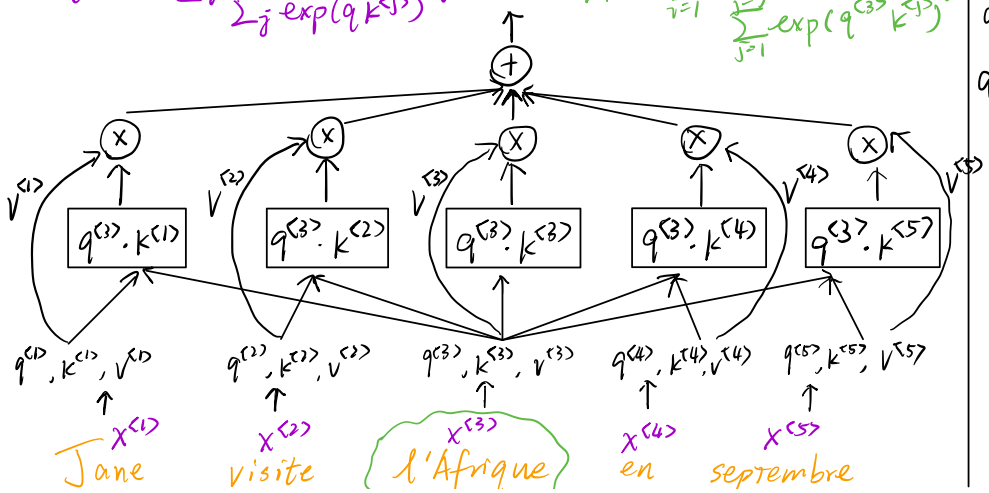
$$A(q,k,v) = \sum_j \frac{\exp(q k^{<j>})}{\sum_j \exp(q k^{<j>})} v^{<j>}$$

$A^{<3>}$

$$A^{<3>} = \sum_{i=1}^{i=5} \frac{\exp(q^{<3>} k^{<i>})}{\sum_{j=1}^{i=5} \exp(q^{<3>} k^{<j>})} v^{<i>}$$

$q^{<3>}$: query about $x^{<3>}$

$q^{<3>} \cdot k^{<t>}$: How good is $x^{<t>}$ an answer to $q^{<3>}$



$q^{<3>} = W^q x^{<3>}$ (query)

$k^{<3>} = W^k x^{<3>}$ (key)

$v^{<3>} = W^v x^{<3>}$ (value)

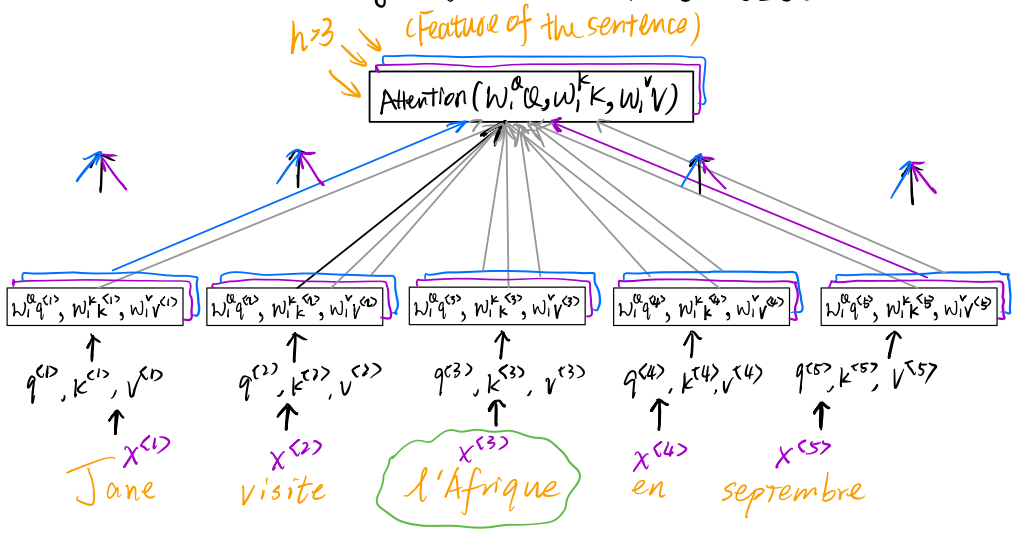
$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

scale parameter

→ The word would not be a fixed word embedding, but defined by the sequence.

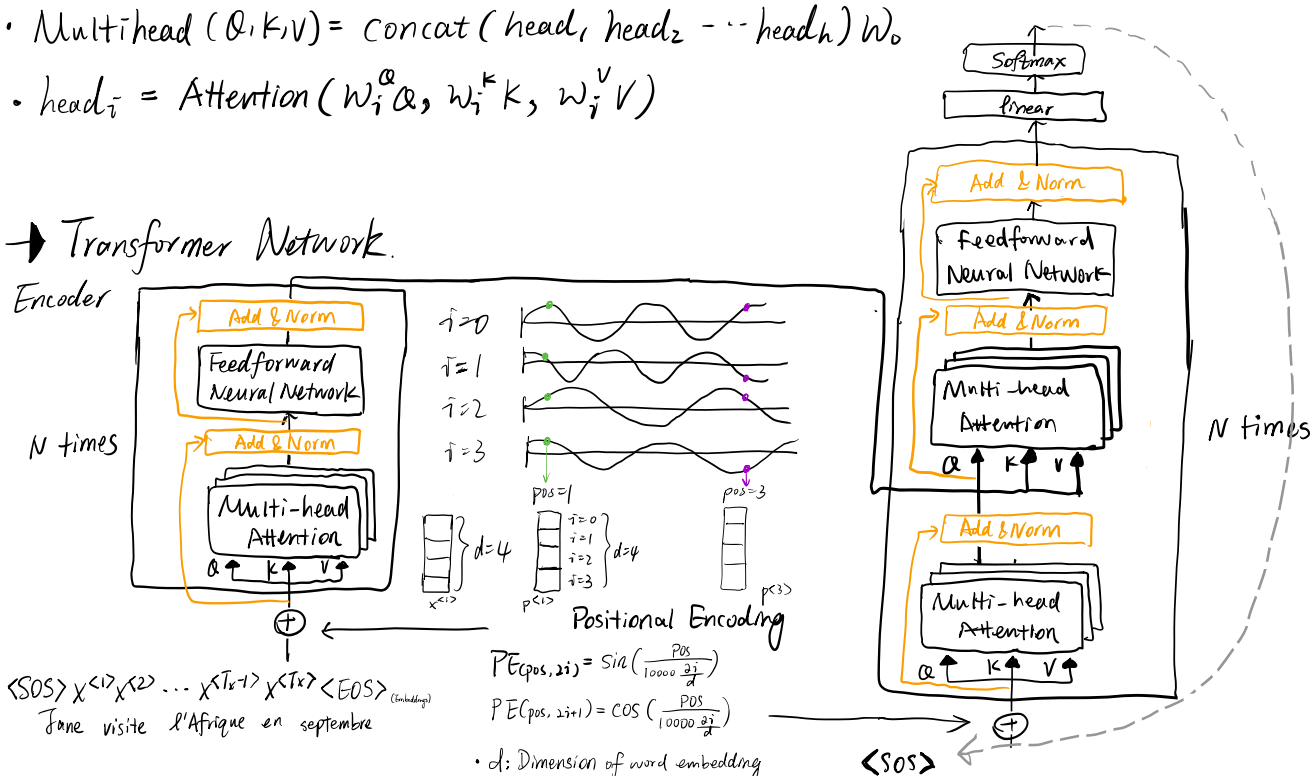
→ Multi-head Attention :

- Each time calculating self-Attention for a sequence is called a head



- $\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$
- $\text{Multihead}(Q, K, V) = \text{concat}(\text{head}_1, \text{head}_2 \cdots \text{head}_h)W_o$
- $\text{head}_i = \text{Attention}(W_i^Q Q, W_i^K K, W_i^V V)$

→ Transformer Network



- $p^{<1>}$: positional encoding of the 1st word.
- i: different dimension
→ Thus same word's embedding would be different given different pos.

Intuition of Decoder:

- Ask questions (query) from the current translation and get context from encoder
- Residue connection: Pass along positional encoding to the whole network.