Coursera
Deep Learning
Specialization

deeplearning.ai
by Andrew Ng

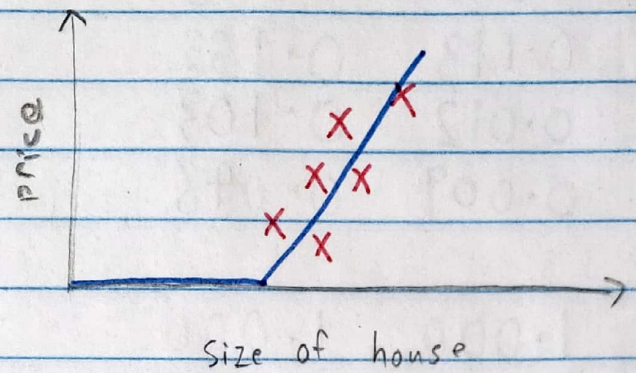
Mar 2020

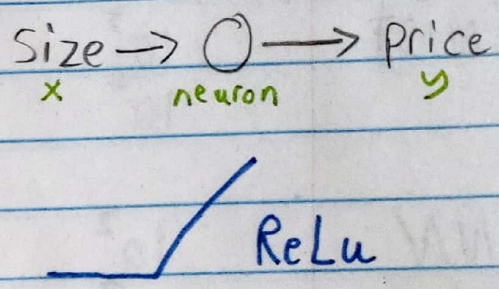
Coursera | Deep Learning Specialization : Course #1

Neural Networks & Deep Learning

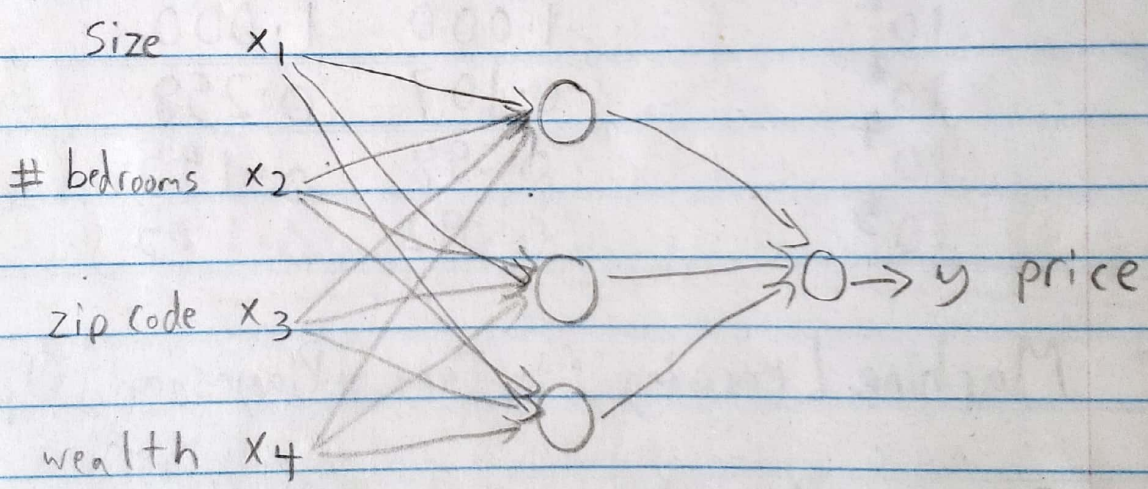
Intro

Example : Housing Price Prediction





→ With more features



Layers are densely connected as input features connected to every circle in the middle

Supervised Learning

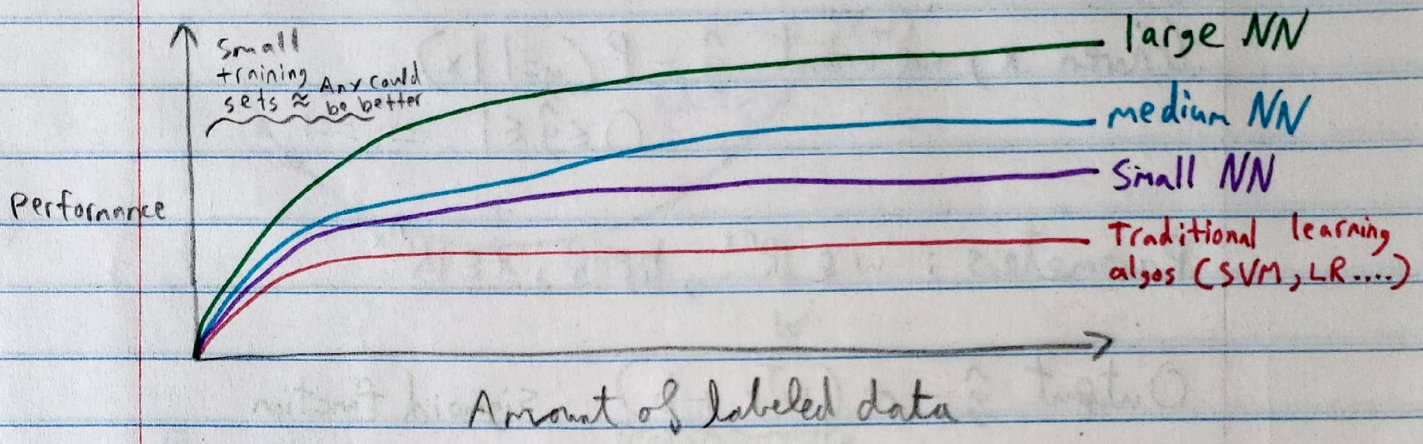
Structured Data

Size	# bedrooms	...	Price
2104	3		400
1600	3		330
2400	3		369
⋮	⋮		⋮
3000	4		540

Unstructured Data

- Audio
- Image
- Text

Scale drives deep learning progress



- Data
- Computation
- Algorithms

~~Week2 Binary Classification~~

Week2 - Logistic Regression as a Neural Network

Binary Classification

Image (64x64) ⟶ 1 (cat) vs 0 (non cat)

y

$64 \times 64 \times 3$ $\quad x = \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix}$ $\quad n = n_x = 12{,}288$

$x \rightarrow y$

m training examples: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$

$M = m_{train}$ $\quad m_{test}$ = # test examples

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \begin{matrix} \uparrow \\ n_x \\ \downarrow \end{matrix} \quad X \in \mathbb{R}^{n_x \times m}$$

$$Y = [y^{(1)}, y^{(2)}, \ldots, y^{(m)}] \quad Y \in \mathbb{R}^{1 \times m}$$

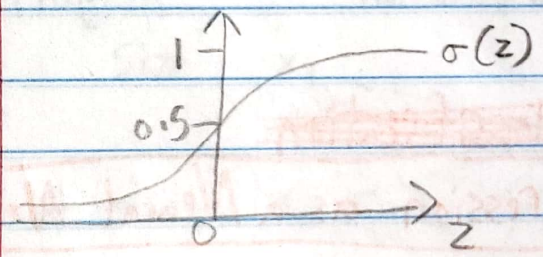
Logistic Regression

Given x, we want $\hat{y} = P(y=1|x)$

$0 \leq \hat{y} \leq 1$

Parameters: $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$, $x \in \mathbb{R}^{n_x}$

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



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

If z is large $\sigma(z) \approx \frac{1}{1+0} = 1$

If z is small $\sigma(z) \approx \frac{1}{1+\text{big num}} \approx 0$

Loss (error) function:

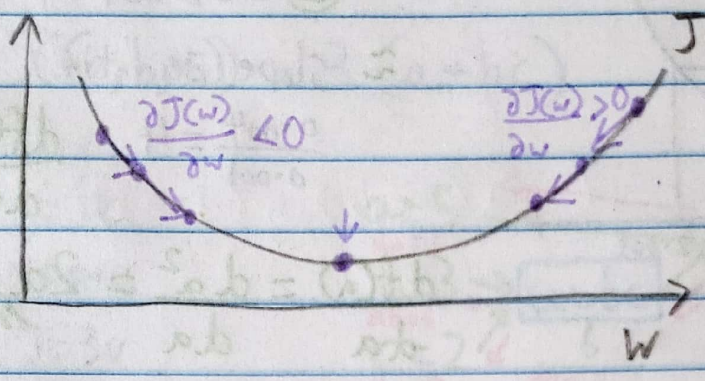
$$\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1-y) \log(1-\hat{y}))$$

Case 1 If $y=1$: $\mathcal{L}(\hat{y}, y) = -\log \hat{y}$ ← make $\hat{y}$ as large as possible

Case 2 If $y=0$: $\mathcal{L}(\hat{y}, y) = -\log(1-\hat{y})$ ← make $\hat{y}$ as small as possible

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

Gradient Descent



Repeat {

$w := w - \alpha \frac{\partial J(w)}{\partial w}$ ← learning rate

$\frac{\partial J(w)}{\partial w}$ [could be written as] → just dw

dw

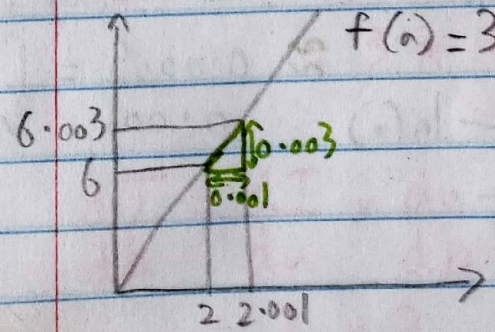
}

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

$\frac{\partial J(w,b)}{\partial b}$ [could be written as in our code] → just db

db

Derivatives



① $a = 2$ $f(a) = 6$

② $a = 2.001$ $f(a) = 6.003$

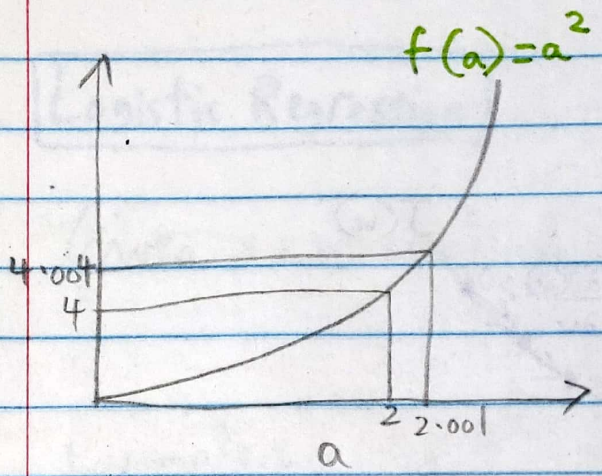
$\approx$ → slope (derivative) of $f(a)$ at $a = 2$ is 3

$\frac{h}{w} = \frac{0.003}{0.001}$

$f(a) = 3a$ $\frac{df(a)}{da} = 3$

Anywhere on the graph the slope is 3

- Not the case for every f_a!



① $a=2$ $\quad f(a)=4$

② $a=2.001$ $\quad f(a)\approx 4.004$

$\approx$ slope (derivative)

$\frac{0.004}{0.001}=4$ $\quad \frac{df(a)}{da}=4$ when $a=2$

$$\frac{df(a)}{da}=\frac{da^2}{da}=2a \;//$$

Anywhere on the graph, the derivative varies on the curve at diff points

- **Formal definition of a derivative:** What happens when we nudge a to the right by an infinitesimal amount $0.000\ldots\ldots1$

$f(a)=a^3$ $\quad \frac{df(a)}{da}=3a^2$

① $a=2$ $\quad f(a)=8$

② $a=2.001$ $\quad f(a)\approx 8.012$

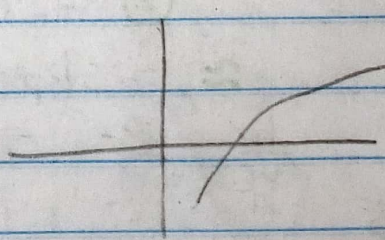
$\approx \frac{0.012}{0.001}=12 \;//$ slope when $a=2$

$f(a)=\log_e(a)$ or $\ln(a)$ $\quad \frac{df(a)}{da}=\frac{1}{a}$

① $a=2$ $\quad f(a)\approx 0.69315$

② $a=2.001$ $\quad f(a)\approx 0.69365$

$\approx \frac{0.0005}{0.001}=\frac{1}{2} \;//$ slope when $a=2$



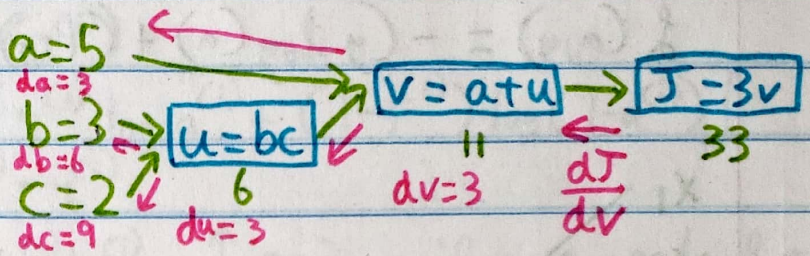
Computational Graph

$J(a,b,c) = 3(a+bc)$

$u = bc$

$v = a+u$

$J = 3v$



$\frac{dJ}{dv} = ? = 3$

$J = 3V$

$V = 11 \rightarrow 11.001$ $\quad \frac{0.003}{0.001} = 3$

$J = 33 \rightarrow 33.003$

$\frac{dJ}{da} = 3 = \frac{dJ}{dv} \times \frac{dv}{da}$ (chain rule)

$= 3 \times 1$

$a = 5 \rightarrow 5.001$

$v = 11 \rightarrow 11.001$

$J = 33 \rightarrow 33.003$

$\frac{dJ}{du} = 3 = \frac{dJ}{dv} \times \frac{dv}{du}$

$= 3 \times 1$

$u = 6 \rightarrow 6.001$

$v = 11 \rightarrow 11.001$

$J = 33 \rightarrow 33.003$

$\frac{dJ}{db} = \frac{dJ}{du} \times \frac{du}{db}$

$= 3 \times 2$

$= 6$

$b = 3 \rightarrow 3.001$

$u = 6 \rightarrow 6.002$ $\quad (b \cdot c)$

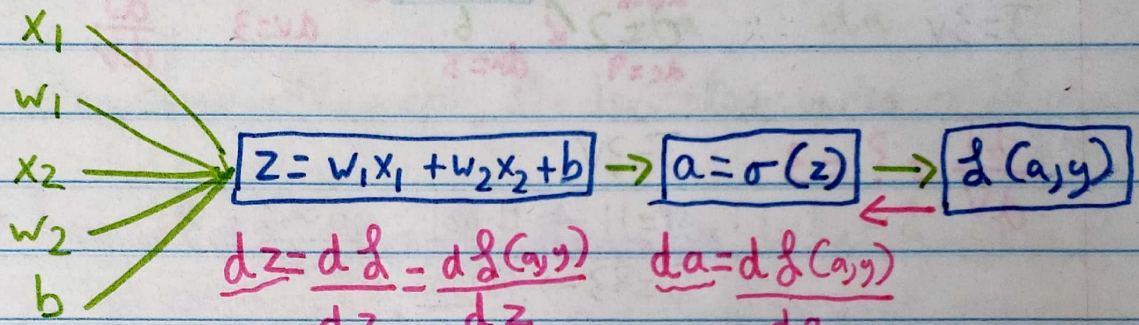
$J = 33 \rightarrow 33.006$

$\left[\frac{dJ}{dv} \times \frac{dv}{du} \times \frac{du}{db}\right] = [3 \times 1 \times 2] = [6]$

$\frac{dJ}{dc} = 3 \times 3 = 9$ //

Logistic Regression Gradient Descent

$$z = w^T x + b$$
$$\hat{y} = a = \sigma(z)$$
$$\mathcal{L}(a, y) = -(y \log(a) + (1-y) \log(1-a))$$



$$dz = \frac{d\mathcal{L}}{dz} = \frac{d\mathcal{L}(a,y)}{dz} \qquad da = \frac{d\mathcal{L}(a,y)}{da}$$

$$= \frac{-y}{a} + \frac{1-y}{1-a}$$

$$= a - y$$

$$= \frac{d\mathcal{L}}{da} \cdot \frac{da}{dz}$$

$$= \left[\frac{-y}{a} + \frac{1-y}{1-a}\right] \times [a(1-a)] = -y(1-a) + a(1-y)$$

$$= -y + ay + a - ay$$

$$= a - y \quad //$$

$$\frac{d\mathcal{L}}{dw_1} = \text{"}dw_1\text{"} = x_1 \cdot dz \qquad dw_2 = x_2 \cdot dz \qquad db = dz$$

$$w_1 := w_1 - \alpha\, dw_1$$
$$w_2 := w_2 - \alpha\, dw_2$$
$$b := b - \alpha\, db$$

$dz = \frac{d\mathcal{L}}{dz}$ The simplification. Could be written as just that in our code.

$J=0$; $dw_1=0$; $dw_2=0$; $db=0$

For $i=1$ to m

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

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

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

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

$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$ ↕ $n=2$ if more, need a for loop

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

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

∴ 2 'For' Loops

$J /= m$

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

→ $w_1 := w_1 - \alpha\, dw_1$

$w_2 := w_2 - \alpha\, dw_2$

$b := b - \alpha\, db$

To make this all more efficient Vectorize //

Vectorization "Art of getting rid of 'for' loops"

Numpy Python

$z = \underbrace{np.dot(w, x)}_{w^T x} + b$ → way faster than non-vectorized version

- Can be done on both CPU & GPU

$$v = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} \rightarrow u = \begin{bmatrix} e^{v_1} \\ \vdots \\ e^{v_n} \end{bmatrix}$$

```
import numpy as np
u = np.exp(v)
```

- Vectorized Logistic Regression (incomplete)

$J=0$, $dw = np.zeros((n\_x, 1))$ (n_x = # of features), $db = 0$

for $i = 1$ to m:

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

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

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

$dz^{(i)} = a^{(i)}(1-a^{(i)})$

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

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

$J /= m$, $dw /= m$, $db /= m$

- $z^{(i)} = w^T x^{(i)} + b \longrightarrow Z = np.dot(w^T, X) + b$

"Broadcasting"

$[z^1, z^2, \ldots, z^m]$ — $1 \times m$

$[w^1, \ldots, w^n]$ — $1 \times m$

$\begin{bmatrix} x^1 \\ \vdots \\ x^{n_x} \end{bmatrix}$ — size (n_x, m)

$[b^1 \ldots b^m]$ — $1 \times m$

The same thing for every other line

Complete Vectorized Logistic Regression

$J=0$, $dw = np.zeros((n_x, 1))$, $db=0$

$Z = w^T X + b$
$A = \sigma(Z)$
$dZ = A - Y$ (I believe same thing for J)
$dw = \frac{1}{m} X dZ^T$
$db = \frac{1}{m} np.sum(dz)$

$w := w - \alpha\, dw$
$b := b - \alpha\, db$

Broadcasting in Python expands to match array sizes

ex1 $\begin{bmatrix}1\\2\\3\\4\end{bmatrix} + \begin{bmatrix}100\\100\\100\\100\end{bmatrix}$ ~~100~~ $= \begin{bmatrix}101\\102\\103\\104\end{bmatrix}$

ex2 $\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + \begin{bmatrix}100 & 200 & 300\\100 & 200 & 300\end{bmatrix} = \begin{bmatrix}101 & 202 & 303\\104 & 205 & 306\end{bmatrix}$
1x3 → 2x3

ex3 $\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + \begin{bmatrix}100 & 100 & 100\\200 & 200 & 200\end{bmatrix} = \begin{bmatrix}101 & 102 & 103\\204 & 205 & 206\end{bmatrix}$
2x1 → 2x3

A.sum(axis=0) sums vertically ↓0 →1 =1 horizontally
cal.reshape(1,4) reshape to 1x4

- a = np.random.randn(5)
- print(a.shape) → result → (5,) Called Rank 1 array
- print(a) → [...] ← Only 1
- a = np.random.randn(5,1)
- print(a.shape) → (5,1) 5x1 matrix
- print(a) → [[...]] ← 2 square brackets

avoid

- assert(a.shape == (5,1)) To double check dimensions

Further explaining Logistic Regression cost function

If $y=1$ $\quad p(y|x) = \hat{y}$

If $y=0$ $\quad p(y|x) = 1-\hat{y}$

$\rightarrow p(y|x) = \hat{y}^{y}(1-\hat{y})^{(1-y)}$

If $y=1$ $\quad p(y|x) = \hat{y}$

If $y=0$ $\quad p(y|x) = 1-\hat{y}$

$$\log p(y|x) = \log \hat{y}^{y}(1-\hat{y})^{(1-y)} = y\log\hat{y} + (1-y)\log(1-\hat{y})$$

$$= -\mathcal{L}(\hat{y}, y)$$

↓ minimize loss

maximize log of probability

≈ maximum likelihood estimation

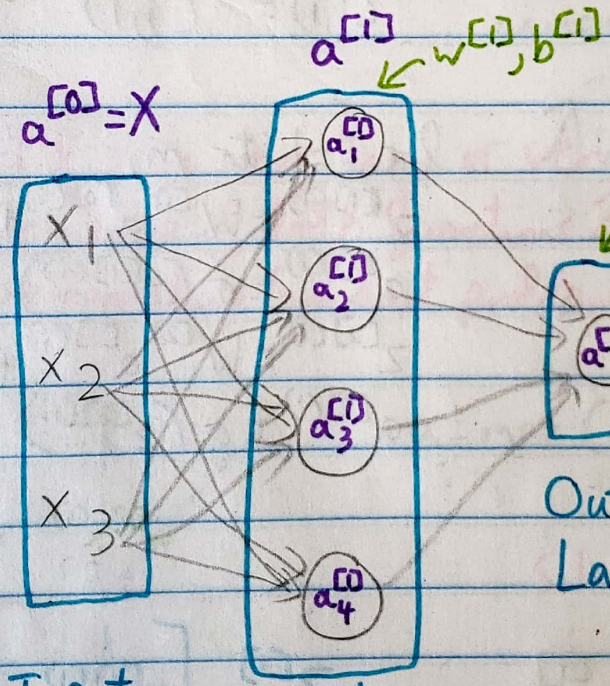
Cost: $J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)})$

(minimize

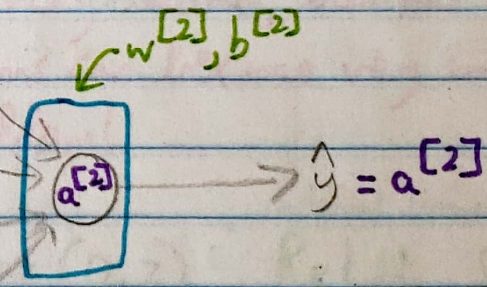
Useful tips

- A neuron computes a linear function ($z = Wx + b$) followed by an activation function
- Column vector $(x, 1)$ Row vector $(1, x)$
- np.dot(a,b) → matrix multiplication
- a * b → element-wise multiplication. Sizes must match! (can broadcast too!)

Week 3 - Shallow Neural Networks

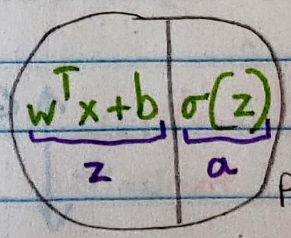


"2 layer NN"
input layer does not count
L=2



$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} \leftarrow \text{layer}$$

↖ node



$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}$ $\quad a_1^{[1]} = \sigma(z_1^{[1]})$

$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]}$ $\quad a_4^{[1]} = \sigma(z_4^{[1]})$

$(4,1)\ (4,3)\ (3,1)\ (4,1)$ $\quad (4,1)\ (4,1)$

$$\begin{bmatrix} -w_1^{[1]T}- \\ -w_2^{[1]T}- \\ \vdots \\ -w_4^{[1]T}- \end{bmatrix}$$

$z^{[1]} = W^{[1]} a^{[0]} + b^{[1]}$

$a^{[1]} = \sigma(z^{[1]})$

$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$

$a^{[2]} = \sigma(z^{[2]})$

$(1,1) = (1,4)(4,1) + (1,1)$

$(1,1) = (1,1)$

Without Vectorization ↖

for i = 1 to m,

$z^{[1](i)} = W^{[1]} a^{[0](i)} + b^{[1]}$

$a^{[1](i)} = \sigma(z^{[1](i)})$

$z^{[2](i)} = W^{[2]} a^{[1](i)} + b^{[2]}$

$a^{[2](i)} = \sigma(z^{[2](i)})$

Vectorizing

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

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

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

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

$W^{[1]} x^{(1)}$ $\quad W^{[1]} x^{(2)}$ $\quad W^{[1]} x^{(m)}$

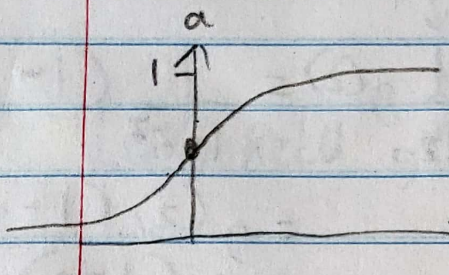
$z_1^{[1]}$ $\quad z_2^{[1]}$ $\quad z_m^{[1]}$

$$Z^{[1]} = \begin{bmatrix} z^{[1](1)} & z^{[1](2)} & \cdots & z^{[1](m)} \\ a^{[1](1)} & a^{[1](2)} & \cdots & a^{[1](m)} \end{bmatrix}$$

$a_1^{[1]}$ $\quad a_2^{[1]}$ $\quad a_m^{[1]}$

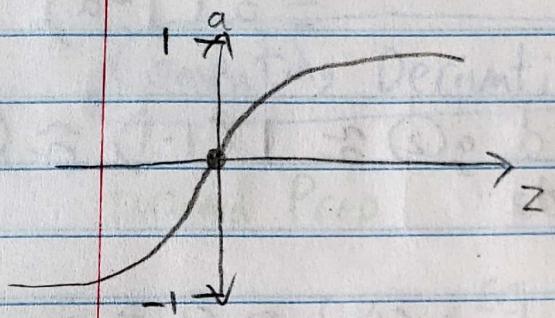
Hidden Units ↕ $\quad$ ↔ training examples

Activations



$a = \frac{1}{1+e^{-z}}$ Sigmoid {0,1} between

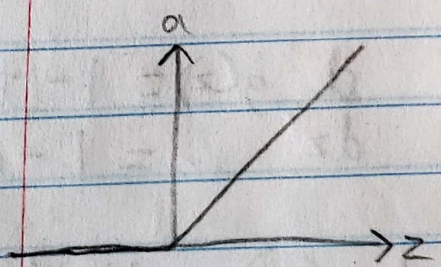
- never use as tanh superior
- maybe only for binary classifat



$a = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$ Tanh {-1, 1} between

- Almost always work better than sigmoid as mean is centered around 0

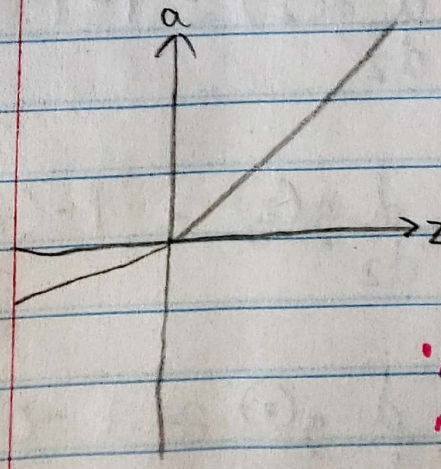
- If z is very large or very small, then the gradient or slope of both functions can become very small → Vanishing Gradient problem!



rectified linear unit

$a = \max(0, z)$ ReLU {0, ∞} between

Solves vanishing gradient problem



$a = \max(0.01z, z)$ Leaky ReLU

- Non linear functions allow you to compute more interesting features / functions

Activation Derivatives

(slope)

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

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

$= g(z)(1 - g(z))$

$= a(1-a)$

→ $z = 10$ $g(z) \approx 1$ $\frac{d}{dz} g(z) \approx 1(1-1) \approx 0$

→ $z = -10$ $g(z) \approx 0$ $\frac{d}{dz} g(z) \approx 0(1-0) \approx 0$

→ $z = 0$ $g(z) \approx \frac{1}{2}$ $\frac{d}{dz} g(z) \approx \frac{1}{2}(1-\frac{1}{2}) \approx \frac{1}{4}$

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

$= 1 - a^2$

→ $z = 10$ $g(z) \approx 1$ $\frac{d}{dz} g(z) \approx 1 - 1^2 \approx 0$

→ $z = -10$ $g(z) \approx -1$ $\frac{d}{dz} g(z) \approx 1 - (-1)^2 \approx 0$

→ $z = 0$ $g(z) = 0$ $\frac{d}{dz} g(z) \approx 1 - 0^2 = 1$

- ReLU $g(z) = \max(0, z)$ $g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$
- Leaky ReLU $g(z) = \max(0.01z, z)$ $g'(z) = \begin{cases} 0.01 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$

Computing Derivatives

Forward Prop

$$Z^{[1]} = W^{[1]} X + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$
$$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$
$$A^{[2]} = g^{[2]}(Z^{[2]}) = \sigma(Z^{[2]})$$

Back prop

$$dZ^{[2]} = A^{[2]} - Y$$

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$

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

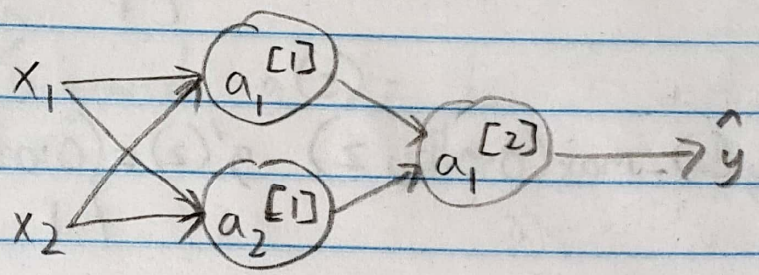
$(n^{[2]},) \rightarrow (n^{[2]}, 1)$

$$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]'}(Z^{[1]})$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$$

$$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, \ldots)$$

Random Initialization



$$W^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \qquad b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad W^{[2]} = [0\ 0]$$

$$a_1^{[1]} = a_2^{[1]} \qquad dz_1^{[1]} = dz_2^{[1]}$$

$$dw = \begin{bmatrix} u & v \\ u & v \end{bmatrix}$$ Symmetric ∴ Both hidden units computing same thing. Not useful! Cost does not really decrease

More on it in next course

In general, model will predict 0 for every example

If W too large → z too large → vanishing gradient problem + learning will be slow

Solution:

$W^{[1]} = np.random.rand((2,2)) * 0.01$

$b^{[1]} = np.zero((2,1))$ ← does not have the same symmetry breaking problem

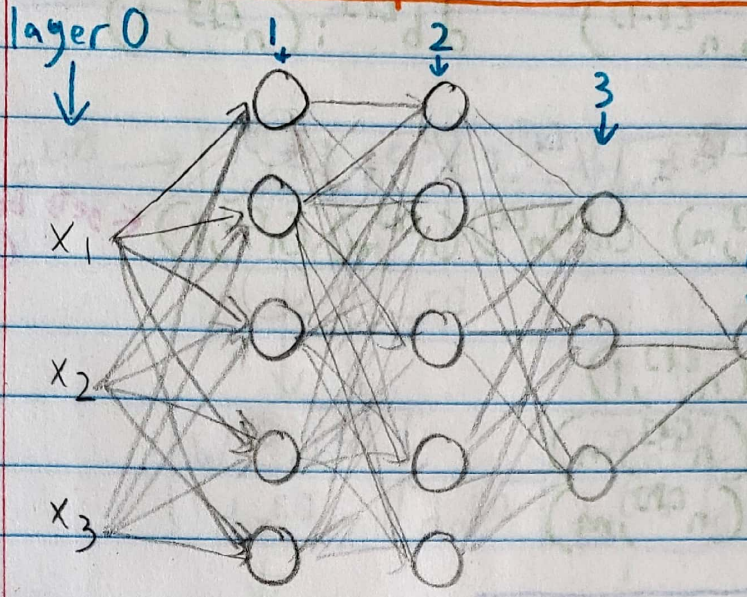
It's okay to have them as 0's

$W^{[2]} = \text{"}\text{"}$

$b^{[2]} = 0$

W too large: Cost starts very high e.g. Sigmoid activation outputs results that are very close to 0 or 1 for some examples, and when it gets it wrong it incurs a very high loss for that example. $\log(a^{[2]}) = \log(0)$ loss goes to ∞

Week 4 - Deep Neural Networks



4 layer NN

$x = a^{[0]}$

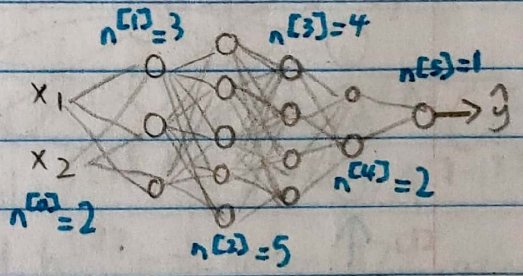
$L = 4$

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

$n^{[1]} = 5, n^{[2]} = 5, n^{[3]} = 3.$

$n^{[4]} = n^{[L]} = 1, n^{[0]} = n_x = 3$

Forward Propagation same way as we did before but now over 4 layers



Matrix Dimensions

$$z^{[1]} = W^{[1]} \cdot x + b^{[1]}$$
$$(3,1) \quad (3,2)\ (2,1) \quad (3,1)$$
$$(n^{[1]},1) \quad (n^{[1]},n^{[0]})\ (n^{[0]},1)$$

$$z^{[2]} = W^{[2]} \cdot a^{[1]} + b^{[2]}$$
$$(5,1) \quad (5,3)\ (3,1) \quad (5,1)$$
$$(n^{[2]},1) \quad (n^{[2]},n^{[1]})\ (n^{[1]},1)$$

Note: z & x not vectorized here. Check next pg

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

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

$W^{[l]} : (n^{[l]}, n^{[l-1]})$

$b^{[l]} : (n^{[l]}, 1)$

dW & db should be the same dimensions as W and b

$dW^{[l]} : (n^{[l]}, n^{[l-1]})$ $db^{[l]} : (n^{[l]}, 1)$

Vectorized: $Z^{[l]} = W^{[l]} \cdot X + b^{[l]}$

$(n^{[l]}, m)$ $(n^{[l]}, n^{[0]})$ $(n^{[0]}, m)$ $(n^{[l]}, 1)$ ← gets broadcasted in python $(n^{[l]}, m)$

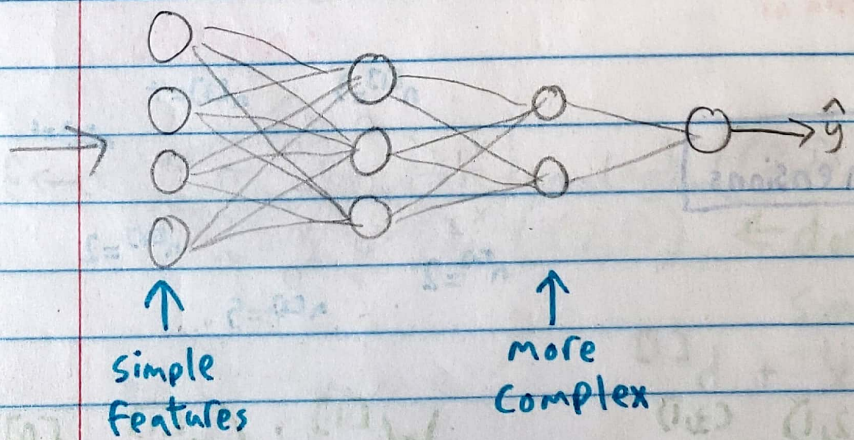
$Z^{[l]}, a^{[l]} : (n^{[l]}, 1)$

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

$dZ^{[l]}, dA^{[l]} : (n^{[l]}, m)$

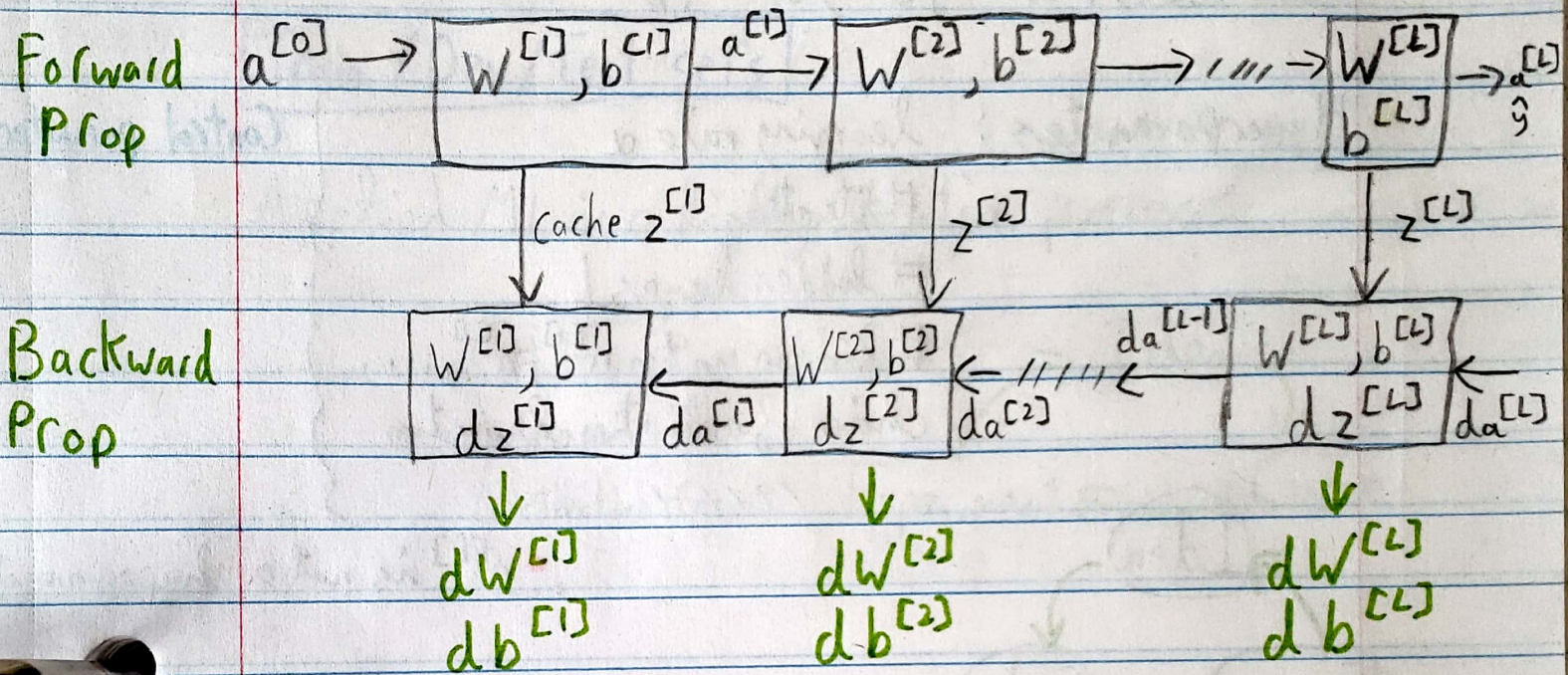
Why deep representation?

There are functions you can compute with a "small" L-layer deep neural network that shallower networks require exponentially more hidden units to compute



Audio → low level (audio waveform) → Phonemes → Words → sentences

Deep Learning Building Blocks



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

Forward Prop

$$Z^{[l]} = W^{[l]} \cdot A^{[l-1]} + b^{[l]}$$
$$A^{[l]} = g^{[l]}(Z^{[l]})$$

Backward Prop

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

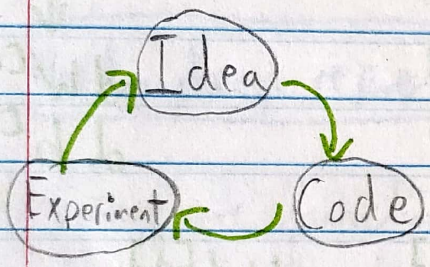
Paraneters vs Hyperparameters

Parameters: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}$ .....

Hyperparameters: learning rate α
iterations
hidden layers L
hidden units $n^{[1]}, n^{[2]}, \ldots$
choice of activation function
//////

Control paraneters

- $a^{[L]}$ is not a hyperparameter



Vectorization Note:

→ Vectorization does not allow you to compute forward propagation in an L-layer neural network without an explicit for-loop over layers $l = 1, 2, \ldots L$

NN Steps

1. Initialize parameters
2. Compute forward prop
3. Compute cost
4. Compute backward prop (get grads)
5. Update parameters

loop

Optimize cost

Coursera Deep Learning Specialization : Course #2

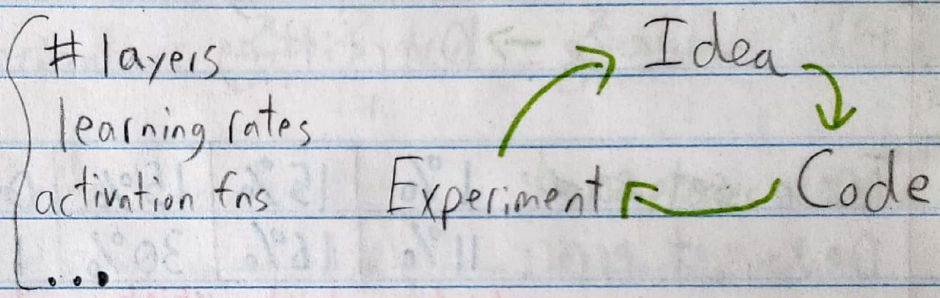
Improving Deep NN : Hyperparameter tuning, Regularization Optimization

Week 1 - Practical Aspects of Deep Learning

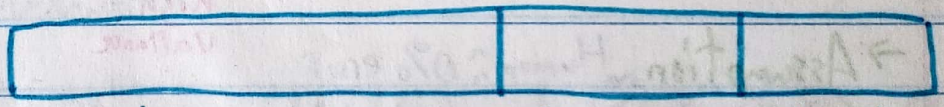
setting up ML

Train / Dev / Test sets

- Applied ML is a highly iterative process



- Data



training set | - Hold out CV | test set

- Development set "dev"

↑ To evaluate algorithms

↑ Given your final classifier, give a confident estimate of how well it is doing [unbiased]

- Previous era : 70/30 or 60/20/20 when data > 1,000,000
- Big data : 1,000,000 98/1/1 • We don't need that much for dev & test

- Mismatched train / test distribution

e.g Training set : cat pics from web

Dev / Test set : cat pics from user apps

→ Make sure dev & test come from same distribution

- Not having a test set might be okay (only dev set)
  → If you only want to iterate & evaluate to build the best model

Setting up ML

Bias / Variance

- High Bias → Underfitting
  High Variance → Overfitting

Train set error:	1%	15%	15%	0.5%
Dev set error:	11%	16%	30%	1%
	high variance	high bias	high bias & high variance	low bias & low variance

→ Assumption Human ≈ 0% error
Optimal (Bayes) error ≈ 0%

Setting up ML

Basic recipe for ML

1. High bias? (training data performance) — some fixes →
   - Get additional features
   - Add polynomial features
   - Decrease λ
   - NN architecture search

2. High variance? (dev set performance) — some fixes →
   - More training examples
   - Smaller set of features
   - Increase λ
   - NN architecture search

Modern era: No need for Bias / Variance trade offs due to the amount of tools we have

Regularization

Regularization

λ = regularization parameter

L_2 regularization: $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$

→ drive w to smaller values

→ used much more often

w1, w2, w3

[np.sum(np.square(w))

+ r, r, r, w2

+ r, r, r, w2]

* [λ/2m]

L_1 regularization: $\frac{\lambda}{2m} \sum_{i=1}^{n_x} |w| = \frac{\lambda}{2m} \|w\|_1$

w will be sparse → it will have a lot of 0's in it

↳ Compresses model ∴ use less memory

→ Weights end up smaller (Weight Decay)

- we don't regularize b as it's only 1 number, whereas w is a high dimensional parameter vector

Neural networks

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

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

"Frobenius norm" $\|\cdot\|_2^2$ $\|\cdot\|_F^2$

$\frac{d}{dw}\left(\frac{1}{2}\frac{\lambda}{m} w^2\right) = \frac{\lambda}{m} w$

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

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

$\alpha < 1$

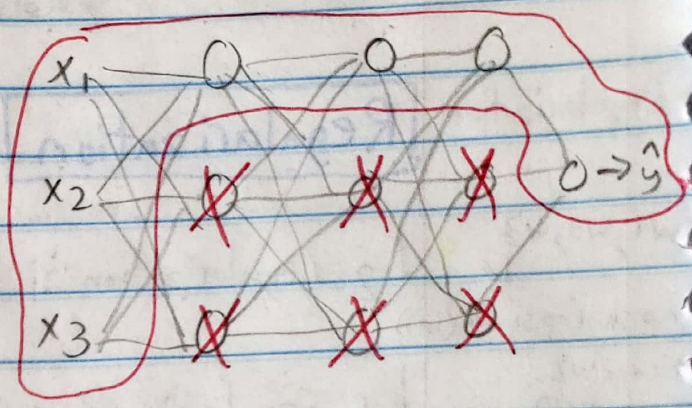
$\left(1 - \alpha\frac{\lambda}{m}\right)$

→ λ: penalizes weight matrices from being too large

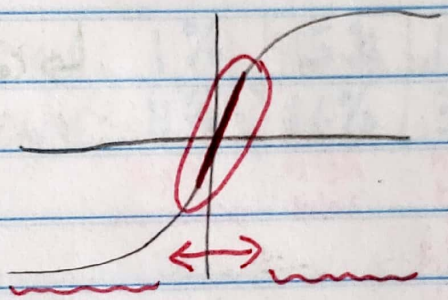
regularization

Why regularization reduces overfitting

- Intuition #1 $w^{[l]} \approx 0$



- Intuition #2



$g(z) : \tanh(z)$

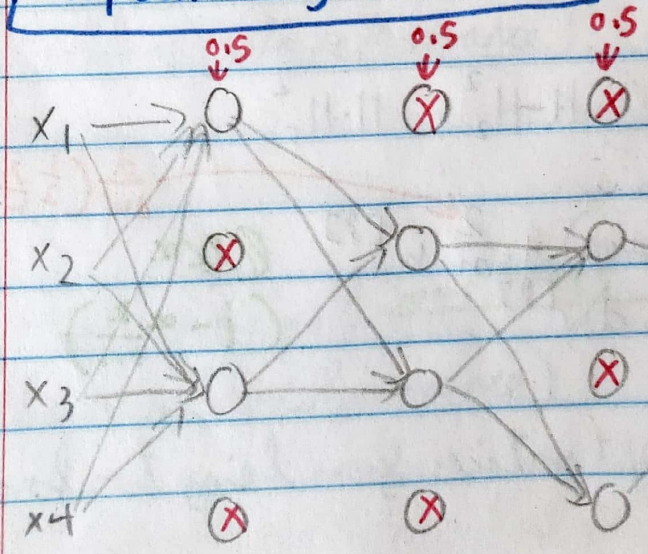
$z \uparrow \quad w^{[l]} \downarrow \qquad z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$

Every layer $\approx$ linear

regularization

Dropout Regularization

Reg technique to avoid overfitting



- For every example different neurons activated
  → % depends on dropout factor

Implementing "Inverted Dropout" Most common way

Illustrate with layer $l=3$ keep-prob $=0.8$

→ `d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob`
`a3 = np.multiply(a3, d3)`

`a3 /= keep_prob`

↓
Scale value of neurons that haven't been shut down + assure cost result = same expected val as without drop out

50 units ⇝ 10 units shut off
→ a3 reduced by 20%

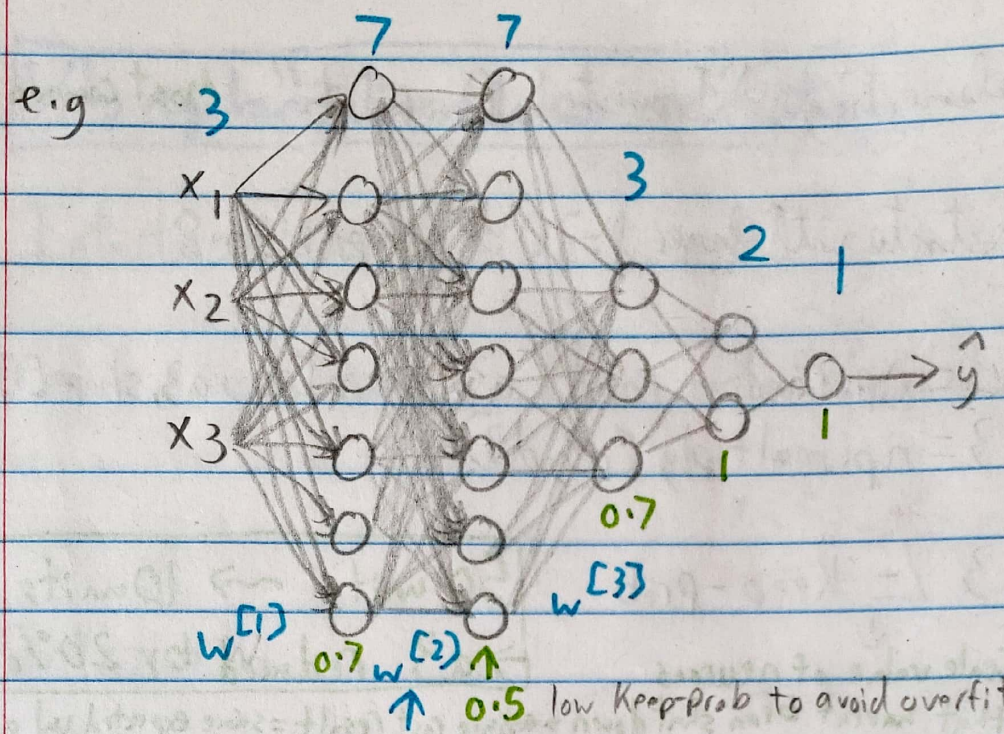
→ Makes test time easier as there is no scaling problem

- At test time → No dropout as it will just add noise to predictions (we don't want output to be random)
  → Computationally inefficient to average result over many iterations if dropout was used

Regularization

Understanding Dropout

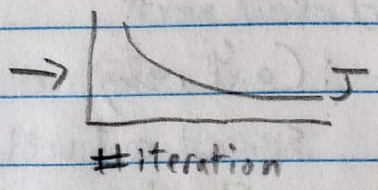
- Why does dropout work?
  → Intuition: Can't rely on any one feature, so have to spread out weights across all features.
  ∴ Shrinks weights!



→ e.g In computer vision due to low amount of data dropout almost always used by default to avoid overfitting

- Downside: J() not well defined due to the random activated neurons

→ Solution: Turn off dropout, make sure J() keeps decreasing

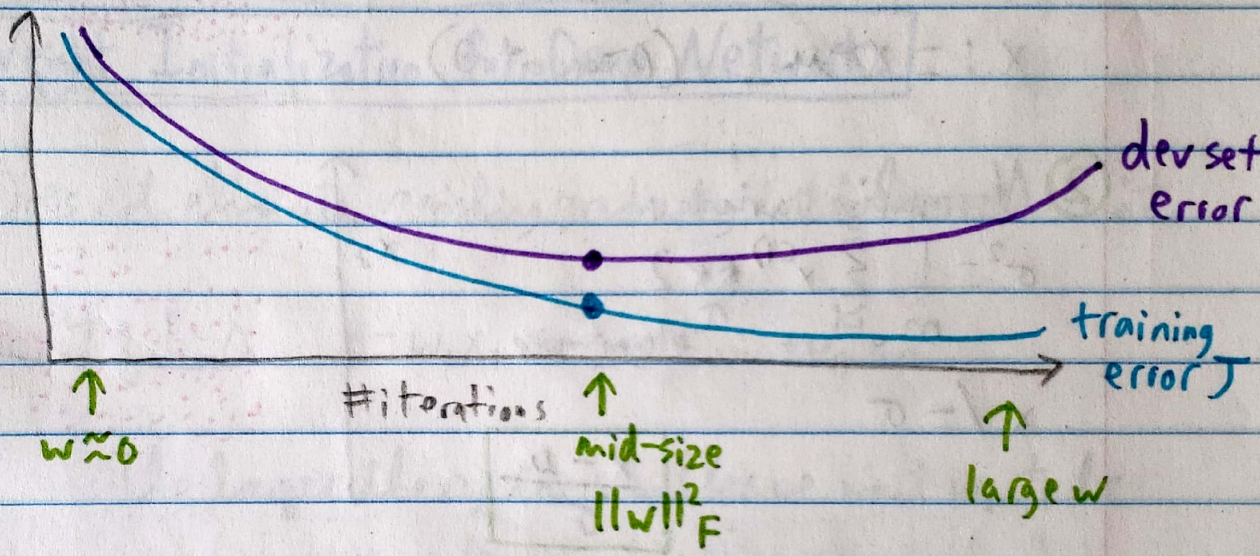


then turn it back on

- Apply dropout both during forward & backward propagation

Other Regularization Methods

- Data Augmentation eg flipping, zooming
- Early Stopping



→ Neural network was doing best during mid-size so we can stop then

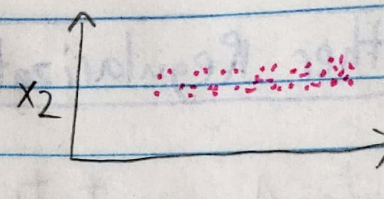
Downside those 2 are separate tasks

- Optimize cost function J
  - GD, Adam, ...
- Not overfit
  - Regularization, ...

Early stopping couples those 2 tasks. Does not work optimally for both. Need to use different tools for each task.

Setting up your optimization problem

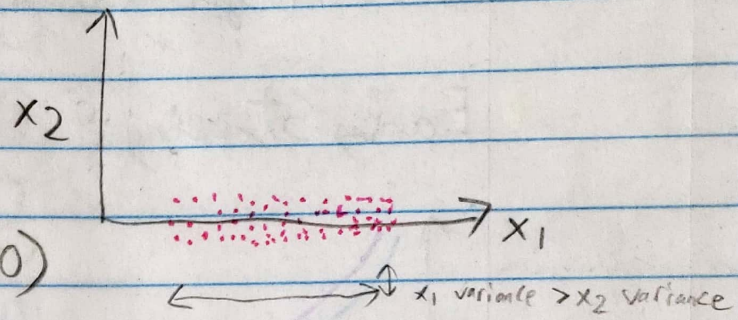
Normalizing Inputs



① Subtract mean:

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

$x := x - \mu$ (centers at 0)



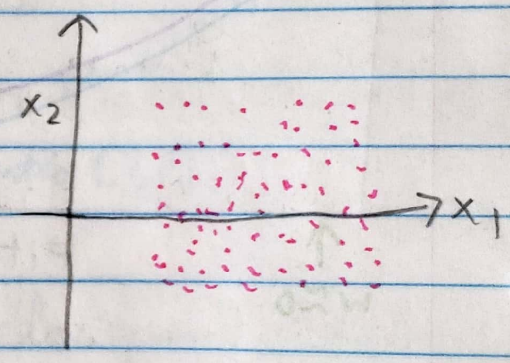
② Normalize variance:

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} **2$$

↑ element-wise

$x /= \sigma$

$$\frac{x - \mu}{\sigma}$$



Important: We use same μ and σ to normalize test set

- Why normalize inputs?
  - To make features on same scale, making cost function J easier and faster to optimize

Setting up your optimization problem

Vanishing / Exploding Gradients

- When derivatives/slopes get really big or really small, making training difficult.

If linear activation

If $W^{[l]} > 1$ Activations ~~Network~~ will explode exponentially
e.g 1.5^L where L is the number of layer

If $W^{[l]} < 1$ Activations will decrease exponentially
e.g 0.5^L

Setting up your optimization problem

Weight Initialization for Deep Networks

- Does not eliminate vanishing gradient, but helps reduce it

e.g If $b=0$ $\quad z = w_1 x_1 + w_2 x_2 + \dots + w_n x_n$

The larger the n, the smaller we want w_i to be

→ $Var(w_i) = \frac{1}{n}$ if ReLU $\frac{2}{n}$ works better

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

Xavier Initialization $\frac{1}{n^{[l-1]}}$ if tanh

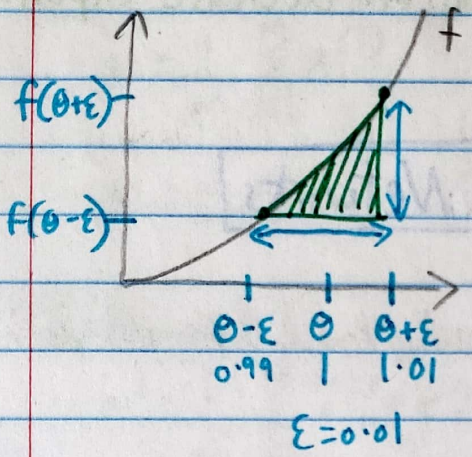
Other variants $\frac{2}{n^{[l-1]} + n^{[l]}}$ He $\frac{2}{n^{[l-1]}}$

- These could be hyperparameters that can be tuned if we would like
  → Low priority compared to other hyperparameters

Setting up your optimization problem

Numerical Approximations of Gradients & Checking your derivative computation

e.g. $f(\theta) = \theta^3$



$$\frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon} \approx g(\theta)$$

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

$$g(\theta) = 3\theta^2 = 3$$

approx error: 0.0001

- Get a much better gradient estimate when taking $(\theta-\varepsilon)$ into account

Gradient Checking

- To verify that backpropagation is correct
- Take $W^{[1]}, b^{[1]}, \ldots, W^{[L]}, b^{[L]}$ and reshape into a big vector θ

$$J(W^{[1]}, b^{[1]}, \ldots, W^{[L]}, b^{[L]}) = J(\theta)$$

Take $dW^{[1]}, db^{[1]}, \ldots, dW^{[L]}, db^{[L]}$ and reshape into a big vector $d\theta$

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

for each i:

grad approx = $d\theta_{approx}[i] = \frac{J(\theta_1, \theta_2, \ldots, \theta_i+\varepsilon, \ldots) - J(\theta_1, \ldots, \theta_i-\varepsilon, \ldots)}{2\varepsilon}$

$\approx d\theta[i] = \frac{\partial J}{\partial \theta_i}$ | $d\theta_{approx} \overset{?}{\approx} d\theta$

- Check $\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2}$ $\approx$ if $\varepsilon = 10^{-7}$
  - 10^{-7} great!
  - 10^{-3} worry

np.linalg.norm(...)

Gradient Checking Implementation Notes

- Don't use in training - only use to debug because it's slow.
- If algorithm fails grad check, look at components (e.g. $db^{[l]}$ or $dW^{[l]}$) to try to identify bug.
- Remember regularization.
- Doesn't work with dropout. (Keep-prob=1)
- Run at random initialization; perhaps again after some training

$$\frac{\partial J}{\partial \theta} = \lim_{\varepsilon \to 0} \frac{J(\theta+\varepsilon) - J(\theta-\varepsilon)}{2\varepsilon}$$

Assignments

Initialization Assignment Conclusions:

- Different initializations lead to different results
- Random initialization is used to break symmetry & for units to learn different things
- Don't initialize to values that are too large
- He initialization works well for networks with ReLu activations

	Train acc	
0's initialization	50%	Fails to break symmetry
Random " "	83%	Too large weights
He " "	99%	Recommended method

Week 2 - Optimization Algorithms

Mini-Batch Gradient Descent

$X = [\underbrace{x^{(1)}\ x^{(2)}\ x^{(3)} \ldots x^{(1000)}}_{X^{\{1\}}\ (n_x, 1000)} \mid \underbrace{x^{(1001)} \ldots x^{(2000)}}_{X^{\{2\}}\ (n_x, 1000)} \mid \ldots \mid \underbrace{\ldots x^{(m)}}_{X^{\{5000\}}\ (n_x, 1000)}]$

(n_x, m)

$Y = [\underbrace{y^{(1)}\ y^{(2)}\ y^{(3)} \ldots y^{(1000)}}_{Y^{\{1\}}\ (1, 1000)} \mid \underbrace{y^{(1001)} \ldots y^{(2000)}}_{Y^{\{2\}}\ (1, 1000)} \mid \ldots \mid \underbrace{\ldots y^{(m)}}_{Y^{\{5000\}}\ (1, 1000)}]$

$(1, m)$

- If $m = 5{,}000{,}000$
- 5,000 mini-batches of 1,000 each

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

→ Forward prop 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

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

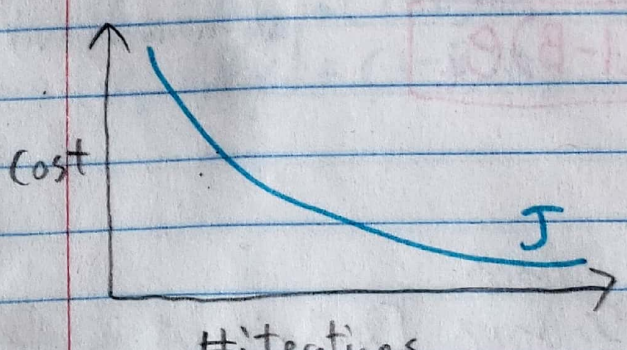
→ Backward prop to compute gradients w.r.t $J^{\{t\}}$

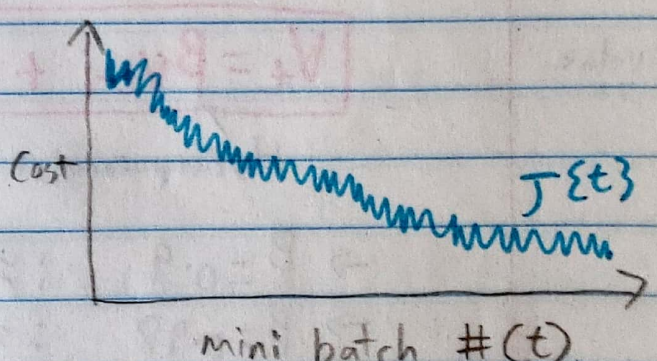
$$W^{[l]} := W^{[l]} - \alpha\, dW^{[l]}, \quad b^{[l]} := b^{[l]} - \alpha\, db^{[l]}$$

} "1 epoch" ≈ 1 pass through training set

→ Much faster than Batch Gradient Descent

Batch Gradient Descent	Mini-Batch Gradient Descent





Plot $J^{\{t\}}$ using $X^{\{t\}}, Y^{\{t\}}$

Choosing Mini-Batch Size

- If mini-batch size = m : Batch Gradient Descent
  If mini-batch size = 1 : Stochastic Gradient Descent

Stochastic GD	Mini-Batch GD	Batch GD
→ Lose speedup from vectorization	→ Value between 1 & m → Fastest learning • Vectorization • Doesn't process entire training set	→ Too long per iteration

- It's another hyperparameter that can be tuned
- If small training set ($m < 2000$), use batch GD
- Typical mini-batch sizes: 64, 128, 256, 512
- Make sure mini-batch fit in CPU/GPU memory

Exponentially Weighted Averages (moving average)

- Reduces noise and locates trends via local weighted average

V = value

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

Hyperparameter

V_t averaging over $\approx \frac{1}{1-\beta}$

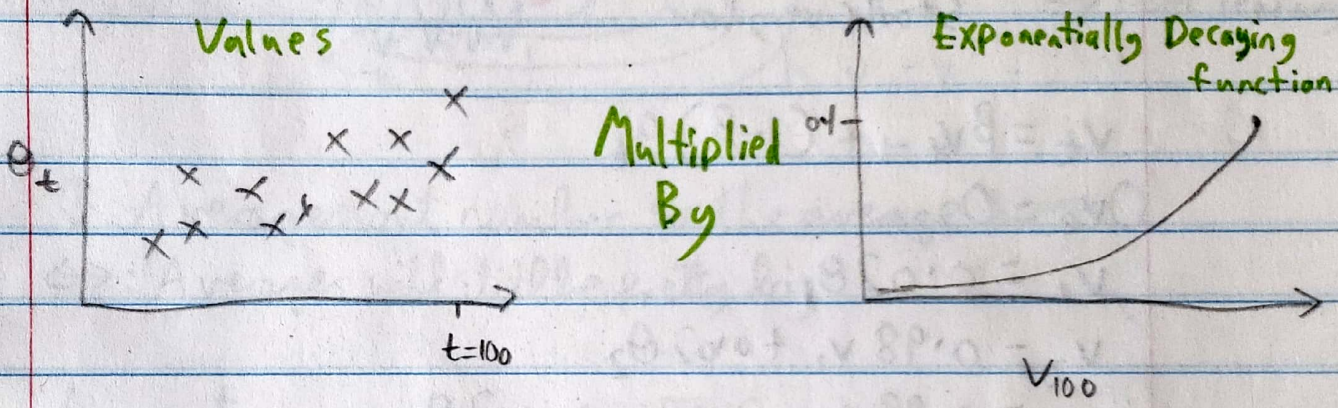
→ $\beta = 0.9$: $\approx$ 10 days
→ $\beta = 0.98$: $\approx$ 50 days [least noisy but graph shifts to the right]
→ $\beta = 0.5$: $\approx$ 2 days

- Averaging over short window
- More noisy
- More susceptible to outliers, but adapts quickly

e.g $\beta = 0.9$

$$V_{100} = 0.9 V_{99} + 0.1 \theta_{100}$$
$$V_{99} = 0.9 V_{98} + 0.1 \theta_{99}$$
$$V_{98} = 0.9 V_{97} + 0.1 \theta_{98}$$

$$V_{100} = 0.1\theta_{100} + 0.9 V_{99} \quad (0.1\theta_{99} + 0.9 V_{98}) \rightarrow 0.1\theta_{98} + 0.9 V_{97}$$
$$= 0.1\theta_{100} + 0.1 \times 0.9\,\theta_{99} + 0.1 \times (0.9)^2 \theta_{98} + 0.1 (0.9)^3 \theta_{97} + \ldots\ldots$$

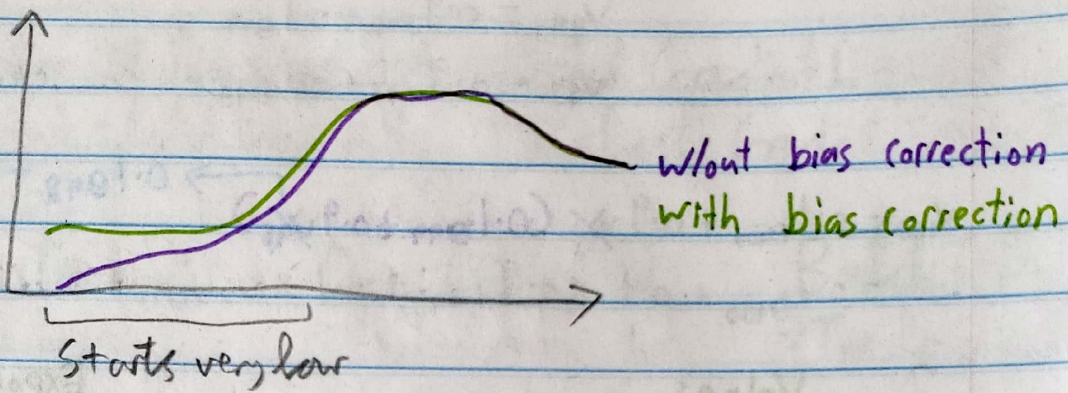


Why 10 days? $0.9^{10} \approx 0.35 \approx \frac{1}{e}$ $\underbrace{(1-\varepsilon)}_{0.9}^{\frac{1}{\varepsilon}} = \frac{1}{e}$

e.g 0.98 $\underbrace{(1-\varepsilon)}_{0.98}^{\frac{1}{\varepsilon}} = 0.98^{\frac{1}{0.02}} = 0.98^{50} = \frac{1}{e}$ $\therefore$ 50 days

$$V_0 = 0$$
$$V_1 = \beta V_0 + (1-\beta)\theta_1$$
$$V_2 = \beta V_1 + (1-\beta)\theta_2$$
$$\vdots$$

Bias Correction with Exponentially Weighted Averages



$v_t = \beta v_{t-1} + (1-\beta)\theta_t$

$v_0 = 0$

$v_1 = 0.02\theta_1$

$v_2 = 0.98 v_1 + 0.02\theta_2$

$= 0.98 \times 0.02 \times \theta_1 + 0.02\theta_2$

$= 0.0196\theta_1 + 0.02\theta_2$ Not quite accurate

Introduce bias correction $\frac{v_t}{1-\beta^t}$

$t=2: \quad 1-\beta^t = 1-(0.98)^2 = 0.0396$

$\frac{v_2}{0.0396} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$ numerator = denominator

When t is large $\rightarrow \beta^t = 0$

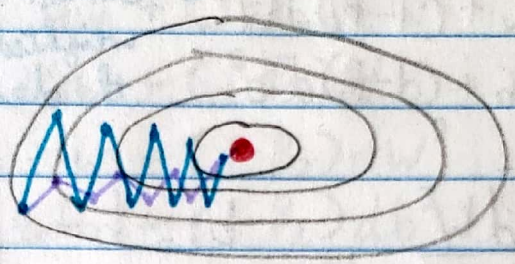
→ Help obtain better estimate! (during initial phase of learning)

Gradient Descent with Momentum

- Smoothes out the steps of gradient descent

GD

GD w/ Momentum



We want ↕ slower learning
↔ faster learning

↕: Averages out number so the average is ≈ 0
↔: Averages will still be pretty big

- Momentum

On iteration t:

Compute dW, db on current mini-batch

$V_{dW} = \beta V_{dW} + (1-\beta) dW$ initialized to 0, same size as W

$V_{db} = \beta V_{db} + (1-\beta) db$ initialized to 0, same size as b

β: Friction; V_{dW}, V_{db}: Velocity; dW, db: Acceleration

$W := W - \alpha V_{dW}$

$b := b - \alpha V_{db}$

Hyperparameters: α, β β usually $= 0.9$

↑ Average over last ≈ 10 gradients

- In practice, bias correction is not really implemented. If $\beta = 0.9$, algorithm would have warmed up after 10 iterations & there isn't a need for bias estimate.

RMSprop Root-Mean-Squared Prop

On iteration t:

Compute dW, db on current mini-batch

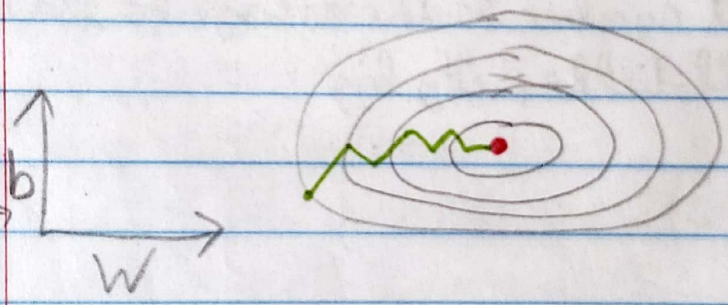
β_2: β for RMSprop

$S_{dW} = \beta_2 S_{dW} + (1-\beta_2) dW^2$ ← element-wise. Keeps exponentially weighted average of the squares of the derivatives

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

$$W := W - \alpha \frac{dW}{\sqrt{S_{dW}} + \varepsilon} \qquad b := b - \alpha \frac{db}{\sqrt{S_{db}} + \varepsilon}$$

Ensures not dividing by zero



Just for illustration → in real life, very high dimensions

What we are hoping for: S_{dW} → small, inorder to speedup $\frac{dW}{\sqrt{S_{dW}}}$

S_{db} → large, inorder to slow down $\frac{db}{\sqrt{S_{db}}}$

Adam Adaptive moment estimation

Adam = RMSprop + Gradient Descent with momentum

Algorithm →

dW 1st moment

dW^2 2nd moment

$V_{dW} = 0$, $S_{dW} = 0$ $\quad V_{db} = 0$, $S_{db} = 0$

On iteration t:

Compute dW, db using current mini-batch

Weighted avg of past grads / Weighted avg of Squares of past grads

$$V_{dW} = \beta_1 V_{dW} + (1-\beta_1)\, dW$$
$$V_{db} = \beta_1 V_{db} + (1-\beta_1)\, db$$
"Momentum"

$$S_{dW} = \beta_2 S_{dW} + (1-\beta_2)\, dW^2$$
$$S_{db} = \beta_2 S_{db} + (1-\beta_2)\, db^2$$
"RMS prop"

$$V_{dW}^{corrected} = \frac{V_{dW}}{1-\beta_1^t} \qquad V_{db}^{corrected} = \frac{V_{db}}{1-\beta_1^t}$$

$$S_{dW}^{corrected} = \frac{S_{dW}}{1-\beta_2^t} \qquad S_{db}^{corrected} = \frac{S_{db}}{1-\beta_2^t}$$
Bias Correction

$$W := W - \alpha \frac{V_{dW}^{corrected}}{\sqrt{S_{dW}^{corrected}} + \varepsilon} \qquad b := b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \varepsilon}$$
Updates

Hyperparameters choice:

α : needs to be tuned

β_1 : 0.9 (dW) Commonly Used

β_2 : 0.999 (dW^2) Adam's authors recommendation

ε : 10^{-8} Adam's authors recommendation

Learning Rate Decay

To learn fast in the beginning then converge to local minimum

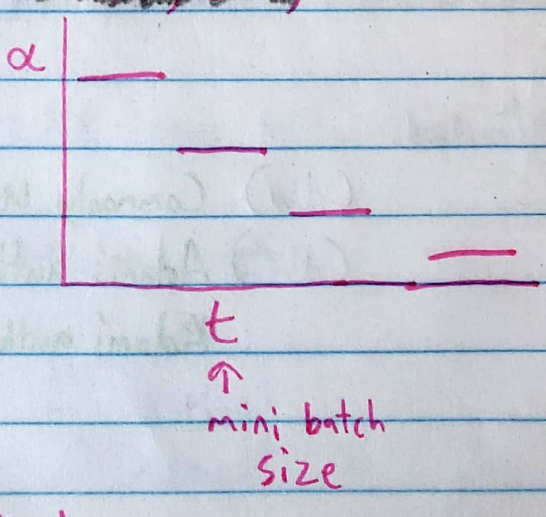
$$\alpha = \frac{1}{1 + \text{decay\_rate} * \text{epoch-num}} \alpha_0$$

e.g $\alpha_0 = 0.2$

decay_rate = 1

Epoch	α
1	0.1
2	0.067
3	0.05
4	0.04
⋮	

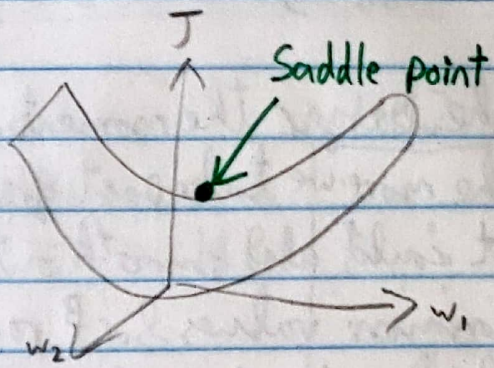
Other formulas

- $\alpha = 0.95^{\text{epoch-num}} \cdot \alpha_0$ exponential decay
- $\alpha = \frac{K}{\sqrt{\text{epoch-num}}} \cdot \alpha_0$
- [redacted]
- 

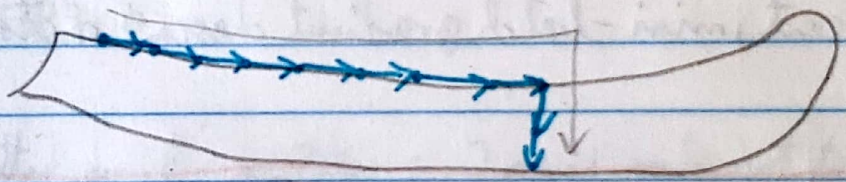
discrete staircase
- manually decay

Local Optima Problem

In very high dimensions
→ could get stuck at a saddle point
→ 0 gradients
(In low dimensions it gets stuck in local optimas)



→ Plateaus can make learning slow



→ This where momentum, RMS prop, Adam etc help with speeding up the rate at which you could get down the plateau and move out of the plateau

Assignment

SGD

- When training set is large, SGD can be faster than BGD, but Parameters will oscillate towards minimum rather than converge smoothly.
→ 3 for loops total
  - Over the number of iterations
  - Over the m training examples
  - Over the layers to update params from $(W^{[l]}, b^{[l]})$ to $^{[l]}$, $^{[l]}$

Mini-batch

Shuffling & Partitioning are 2 required steps for mini-batch G-D

Momentum

- The larger the momentum β is, the smoother the update because the more we take past gradients into account. But if β is too big, it could also smooth out the updates too much.
- Common values for β range from 0.8 to 0.999. If you don't feel inclined to tune this, $\beta = 0.9$ is reasonable by default
- Momentum takes past gradients into account to smooth out the steps of gradient descent. It can be applied to batch gradient descent, mini-batch gradient descent or stochastic gradient descent

Week 3 - Hyperparameter Tuning, Regularization and Optimization

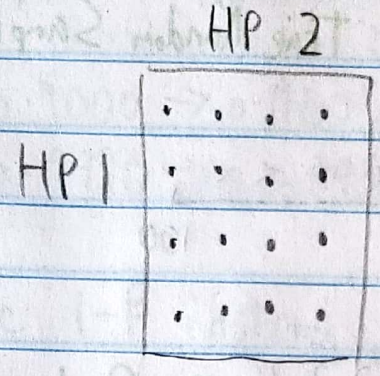
Hyperparameter Tuning

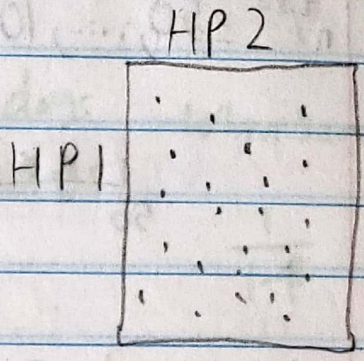
Tuning Process

- So far, here are our hyperparameters:
  - α ← highest importance
  - β
  - $\beta_1, \beta_2, \varepsilon$ ~ 0.9, 0.999, 10^{-8} (Andrew doesn't tune this)
  - #layers
  - #hidden units
  - learning rate decay
  - mini-batch size

- Try random values: Don't use a grid

e.g



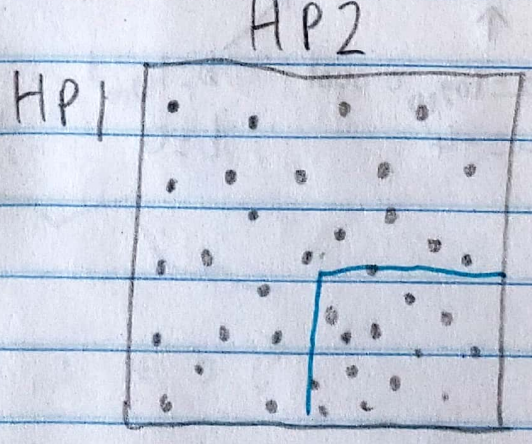


e.g HP1 α
HP2 ε

α matters much more than ε. In grid, only got to try 4 values of α, where in random, we got to try 16 different values. In practice, there are much more hyperparametes.

- It's usually hard to know in advance which hyperparametes matter more than others for the application in hand

- Another common practice: Coarse to fine

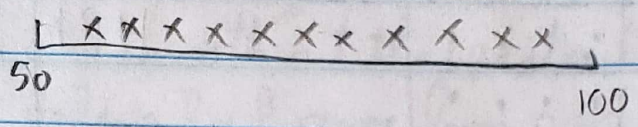


- Found best samples that lie in • region
- Sample further values in that region (smaller square)
- Pick value that works best on train/dev set

Hyperparameter Tuning

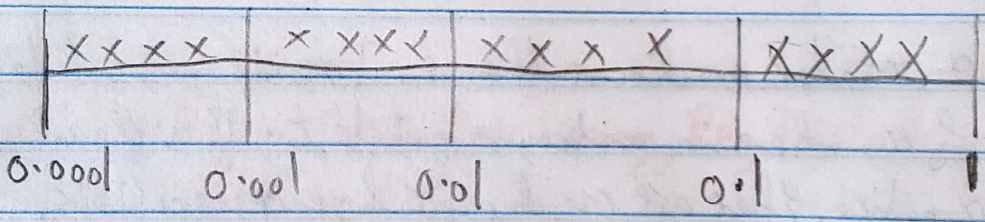
Using an appropriate scale to pick hyperparameters

- $n^{[l]} = 50, \ldots, 100$ e.g. True Random Sampling works



#layers L: 2-4 2,3,4 True Random Sampling Works

- $\alpha = 0.001, \ldots, 1$ Use Logarithmic Scale



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

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

Generally if $\alpha = 10^a \ldots 10^b$

$a = \log_{10} 0.0001$ $b = \log_{10} 1$

$a = -4$ $b = 0$

$r \in [a, b]$ $[-4, 0]$ $\alpha = 10^r$

$\beta = 0.9 \ldots 0.999$ ⇐ Bad idea to use for sampling

↓ 10 days ↓ 1000 days

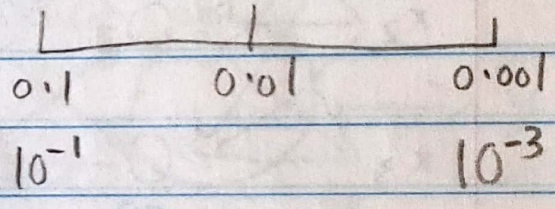
$0.9000 \rightarrow 0.9005$] 10 extra days
$0.9990 \rightarrow 0.9995$] 1000 extra days

When same increase

$\frac{1}{1-\beta}$

∴ use $1-\beta$ instead

$1-\beta = 0.1 \ldots 0.001$

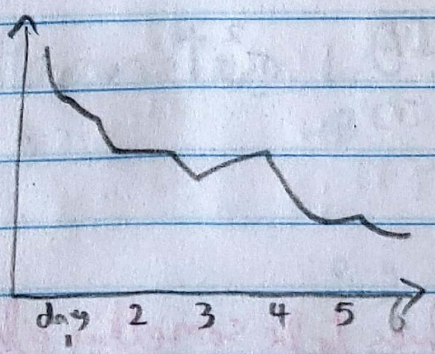


$r \in [-3, -1]$
$1-\beta = 10^r$
$\beta = 1-10^r$

Hyperparameter Tuning

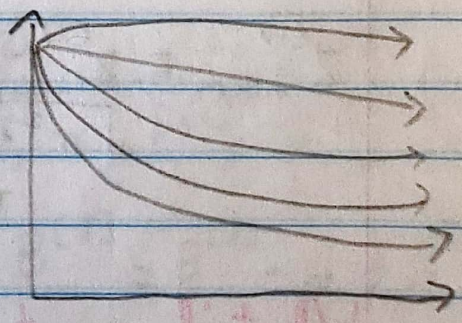
Hyperparameter tuning in practice

Babysitting one mode (low on resources)



"Panda" approach

Training many models in parallel



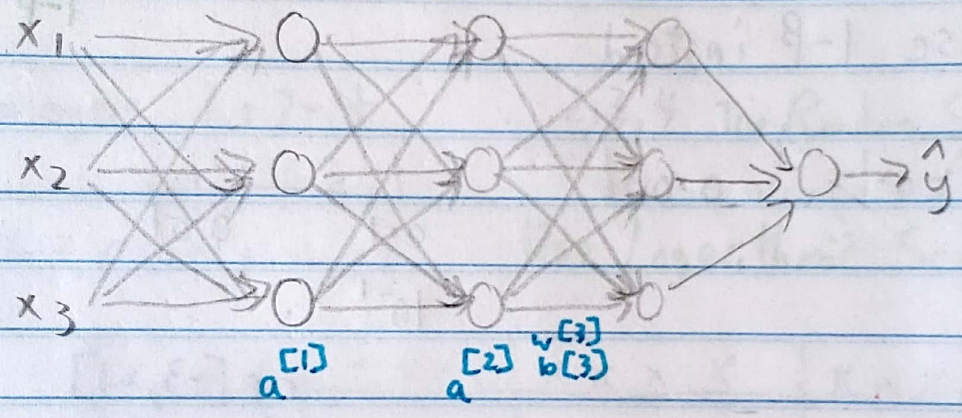
"Caviar" approach

Batch Normalization

Normalizing activations in a network

- Normalizing inputs speeds up learning.

What about in deeper network?



Can we normalize $a^{[2]}$ so we can train $w^{[3]}, b^{[3]}$ faster?

→ Normalize $z^{[2]}$

- Implementing Batch Norm

$$\mu = \frac{1}{m}\sum_i z^{(i)} \qquad \sigma^2 = \frac{1}{m}\sum_i (z_i - \mu)^2$$

$$z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}} \qquad \mu = 0 \quad \sigma^2 = 1$$

↑ To avoid 0's

What if we want μ and σ^2 values to be something else?

→ $\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$

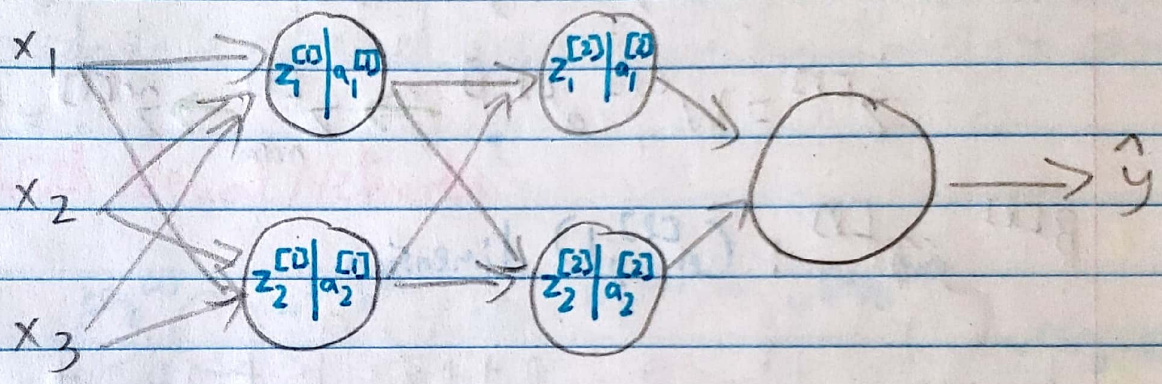
γ, β — Learnable parameters

Use $\tilde{z}^{[l](i)}$ instead of $z^{[l](i)}$

→ To better take advantage of the non-linearity of the activation function

Batch Normalization

Fitting Batch Norm into a neural network



$X \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \rightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]})$

Batch Norm (BN)

If mini-batches

$X^{\{1\}}$
$X^{\{2\}}$ all separately
$\vdots$
$X^{\{t\}}$

$a^{[1]} \xrightarrow{w^{[2]}, b^{[2]}} z^{[2]} \xrightarrow[BN]{\beta^{[2]}, \gamma^{[2]}} \tilde{z}^{[2]} \rightarrow a^{[2]} \rightarrow \dots$

Parameters: $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}, \dots, w^{[L]}, b^{[L]}$
$\beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \dots, \beta^{[L]}, \gamma^{[L]}$

e.g. $d\beta^{[l]}$ $\quad \beta^{[l]} = \beta^{[l]} - \alpha d\beta^{[l]}$

If using batch-norm: $b^{[l]}$ can be eliminated

$$z^{[l]} = W^{[l]} a^{[l-1]} \boxed{+ b^{[l]}}$$

Subtracting and adding constants to find mean ends up equating 0 for batch-norm

Parameters: $W^{[l]}$, ~~$b^{[l]}$~~, $\beta^{[l]}$, $\gamma^{[l]}$

$$z^{[l]} = W^{[l]} a^{[l-1]} \rightarrow z^{[l]}_{norm} \rightarrow \tilde{z}^{[l]} = \gamma^{[l]} z^{[l]}_{norm} + \beta^{[l]}$$

This new controls bias terms

- $\beta^{[l]}, \gamma^{[l]}$: $(n^{[l]}, 1)$ dimension

- Implementing mini-batch gradient descent

for $t = 1 \ldots$ num Mini-Batches

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

In each hidden layer, use BN to replace $z^{[l]}$ with $\tilde{z}^{[l]}$

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

Update params

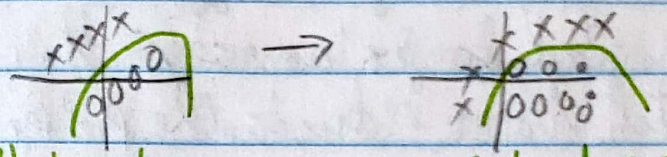
$$W^{[l]} := W^{[l]} - \alpha dW^{[l]}$$
$$\beta^{[l]} := \beta^{[l]} - \alpha d\beta^{[l]}$$
$$\gamma^{[l]} := \gamma^{[l]} - \alpha d\gamma^{[l]}$$

Works with momentum, RMSprop, Adam....

Batch Normalization

Why does Batch Norm work?

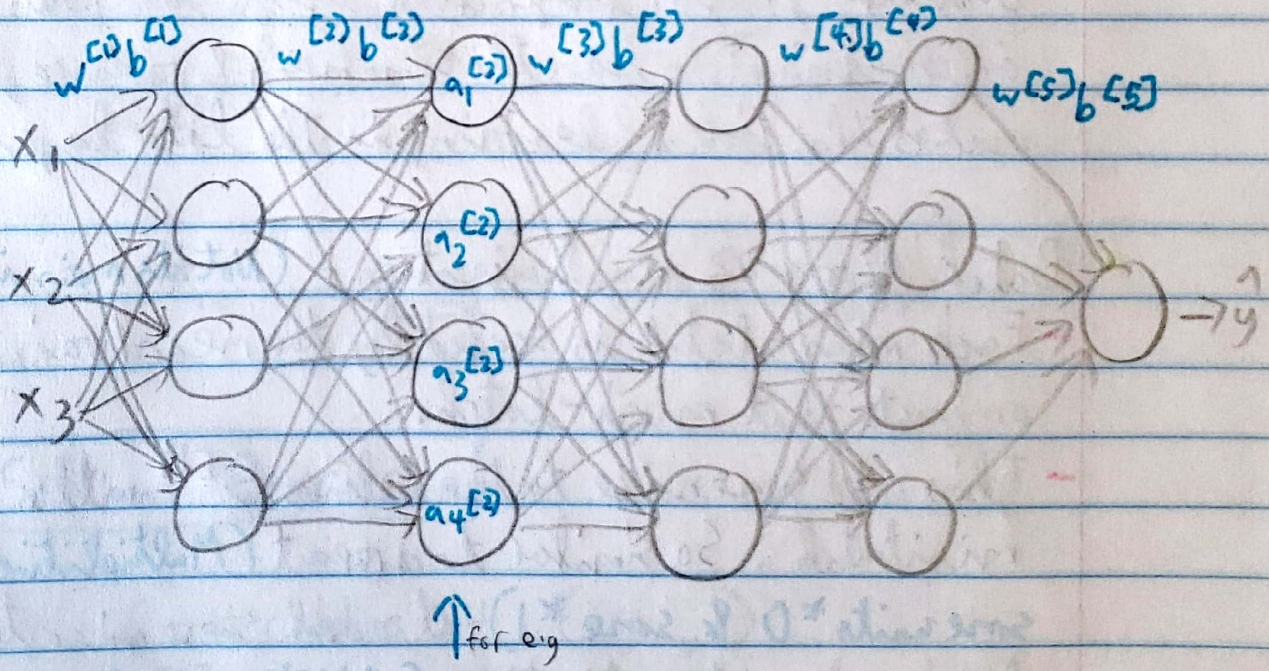
e.g



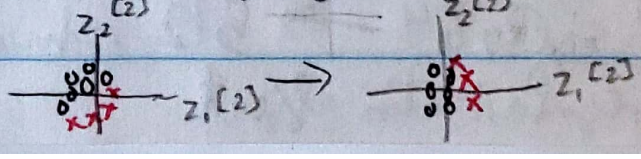
"Covariate shift" $x \to y$

- If the distribution of x changes, then the algorithm needs to be retrained, even if the function remains unchanged

With Neural Networks



- $w^{[1]}, b^{[1]}$ & $w^{[2]}, b^{[2]}$ keep on changing, so the $a^{[2]}$ values keep on changing ∴ suffers from covariate shift problem
- Batch norm reduces the distribution that the $a^{[2]}$ values shift around. e.g if mean 0 & variance 1



keeps all a's around that μ & σ^2

- Limits the amount to which updating the parameters in the earlier layers affect the distribution of values that the 3rd layer now sees & ∴ has to learn on. So Batch Norm reduces the problem of the input values changing.
- Causes these values to become more stable so the later layers in the network have a firm ground to stand on.
- Weakens coupling between earlier layers & later layers allows each layer to learn more independently, speeds up learning of the whole network
- Basically, earlier layers don't shift that much as they are constrained to have the same mean & variance, which makes the job of learning in the later layers easier

- Batch Norm as regularization: (Not its main intent though)
- Each mini-batch is scaled by the mean/variance computed on just that mini-batch.
- This add some noise to the values $z^{[l]}$ within that minibatch. So similar to dropout (Multiplitive noise as some units * 0 & some * 1), it adds some noise to each hidden layer's activations (Additive noise due to mean & multiplitive due to variance).
- This has a slight regularization effect. (Doesn't rely on one hidden unit by downstream hidden units)

- Mini batch size eg 64 $\xrightarrow{\text{increase}}$ 512 reduces Batch Norm noise ∴ reduces regularization effect

Batch Normalisation

Batch Norm at test time

- During training time, μ and σ^2 computed over all examples in 1 mini-batch. Each mini-batch has a different μ and σ^2.
- But during testing:
  Estimate μ, σ^2 using exponentially weighted average across all mini-batches

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

$\mu^{\{1\}[l]}$ (θ_1), $\mu^{\{2\}[l]}$ (θ_2), $\mu^{\{3\}[l]}$ (θ_3) $\longrightarrow \mu$

$\sigma^{2\{1\}[l]} \longrightarrow \sigma^2$

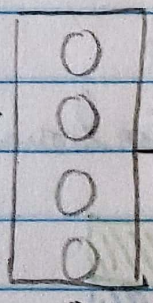
- $\therefore \quad z_{norm} = \dfrac{z - \mu}{\sqrt{\sigma^2 + \varepsilon}} \qquad \tilde{z} = \gamma z_{norm} + \beta$

Multi-class classification

Softmax Regression

- e.g. $C = \#\text{classes} = 4 \quad (0, \ldots, 3)$

  $n^{[L]} = 4 = C$

$X \longrightarrow \longrightarrow$  $\longrightarrow \hat{y}$ (4,1)

$\hat{y}$ sum = 1

$n^{[L]} = 4$

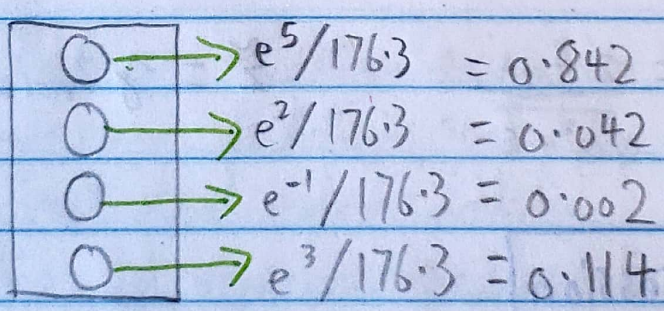
- Activation function: $t = e^{(z^{[L]})}$ (4,1) dim

$$a^{[L]} = \frac{e^{z^{[L]}}}{\sum_{i=1}^{4} t_i}, \quad a_i^{[L]} = \frac{t_i}{\sum_{i=1}^{4} t_i}$$ (4,1) dim

($\hat{y}$)

- e.g $z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$ $t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}$

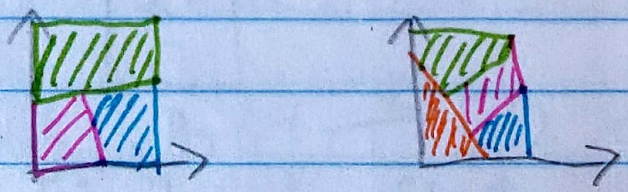
$\sum_{i=1}^{4} t_i = 176.3$ $\quad a^{[L]} = \frac{t}{176.3}$



- $a^{[L]} = g^{[L]}(z^{[L]})$ (4,1) (4,1)

- The decision boundary between classes is linear

e.g



Multi-class Classification

Training a softmax classifier

Softmax $\begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$ Hard max $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$

- Softmax regression generalizes logistic regression to C classes
  - If C=2, softmax reduces to logistic regression

- Loss function $\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{4} y_j \log \hat{y}_j$

e.g $y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ $\hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$

$\mathcal{L}(\hat{y}, y) = -y_2 \log \hat{y}_2$

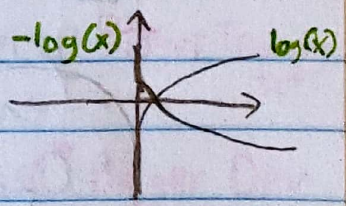
$= -\log \hat{y}_2$

↑ To minimize this

↑ we need to maximize this

$Y = [y^{(1)}, y^{(2)}, \dots, y^{(m)}]$ (4, m)

$\hat{Y} = [\hat{y}^{(1)}, \hat{y}^{(2)}, \dots, \hat{y}^{(m)}]$ (4, m)



- Backprop/Gradient Descent

$X \rightarrow (\dots) \rightarrow [\ ] \rightarrow \hat{y}$

$z^{[L]} \rightarrow a^{[L]}$

$dz^{[L]} = \hat{y} - y$ (4,1)

$\left(\frac{dJ}{dz^{[L]}}\right)$

Programming Frameworks

- Caffe / Caffe2
- CNTK
- DL4J
- Keras
- Lasagne
- mxnet
- PaddlePaddle
- TensorFlow
- Theano
- Torch

Choosing deep learning frameworks:

- Ease of programming (development and deployment)
- Running speed
- Truly open (open source with good governance)

Tensorflow

- Coding session

- Two main object classes:
  - Tensors (Variables, Placeholders, ...)
  - Operations (tf.matmul, tf.add, ...)

Coursera Deep Learning Specialization : Course #3

Structuring Machine Learning Projects

Week 1 - Why ML Strategy

Intro

Orthogonalization

- Eg radio tuning : if the radio sound was too noisy, we want 1 knob to tune in order to adjust the frequency of the radio channel. We don't want to have to carefully adjust five different knobs that affect different things.

- Chain of assumptions in ML

Fit training set well on cost function — If it doesn't: bigger network, Adam

↓

Fit dev set well on cost function — Regularization, Bigger Training set

↓

Fit test set well on cost function — Bigger Dev set

↓

Performs well in real world — Change Dev set or cost function or metric

Setting up your goal

Single Number Evaluation Metric

Idea → Code → Experiment → Idea

Classifier	Precision	Recall	F1 Score
A	95%	90%	92.4%
B	98%	85%	91.0%

⇒ Dev set (well defined) + Single real number evaluation metric

Speeds up iterating

e.g

Algorithm	US	China	India	Other	Average
A	3%	7%	5%	9%	6%
B	5%	6%	5%	10%	6.5%
C	2%	3%	4%	5%	3.5%
D	5%	8%	7%	2%	5.25%
E	4%	5%	2%	4%	3.75%
F	7%	11%	8%	12%	9.5%

Setting up your goal

Satisficing and Optimizing Metrics

e.g

Classifier	Accuracy (Optimizing)	Running Time (Satisficing)	
A	90%	80ms	
B	92%	95ms	← Best Classifier
C	95%	1500ms	

→ Maximize accuracy subject to runningTime ≤ 100ms

- N metrics : 1 optimizing
  N-1 satisficing

- e.g voice controlled devices
  → Maximize accuracy subject to ≤ 1 false pos every 24 hours
  Optimizing : Accuracy
  Satisficing : False Positives

Setting up your goal

Train / dev / test distributions

- Make sure dev and test set data come from the same distribution
  → Distributions e.g Regions, high/low income individuals

- Choose a dev set and test set to reflect data you expect to get in the future and consider important to do well on.

Setting up your goal

Size of the dev and test sets

- Earlier era 70/30 or 60/20/20 for < 10K data
  Now 98/1/1 for > 1000K data

- Set your test set to be big enough to give high confidence in the overall performance of your system.

setting up your goal

When to change dev/test sets and metrics

Metric: classification error
Algorithm A: 3% error → Has "unacceptable" images
Algorithm B: 5% error → Doesn't

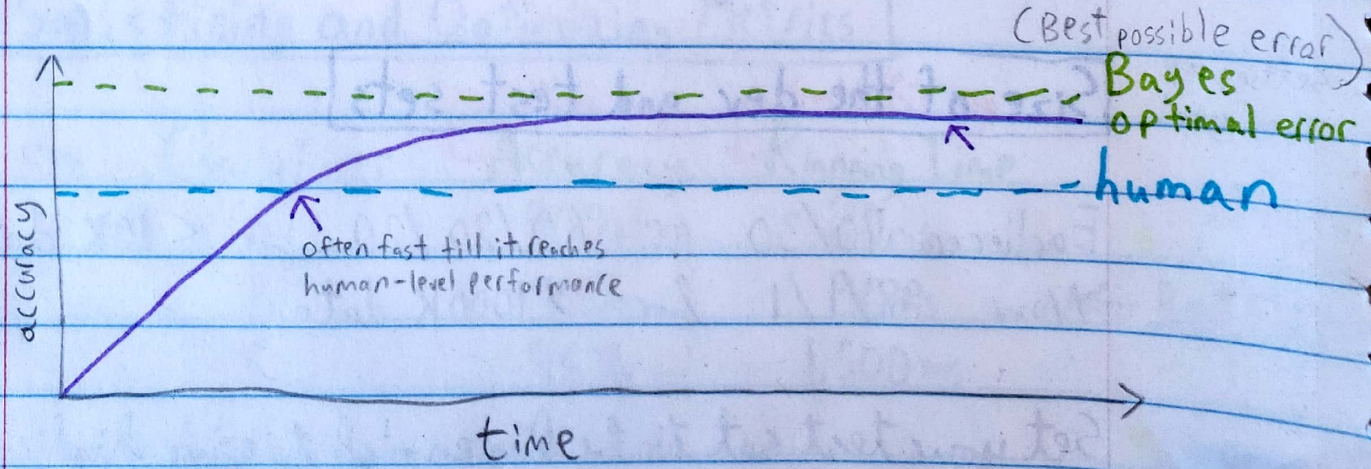
- Metric + Dev: Prefer A
- You/users: Prefer B

$$\text{Error} = \frac{1}{\sum_i w^{(i)}} \frac{1}{m_{dev}} \sum_{i=1}^{m_{dev}} w^{(i)} \mathbb{1}\{y_{pred}^{(i)} \neq y^{(i)}\}$$

$$\rightarrow w^{(i)} = \begin{cases} \text{e.g. } 1 \text{ if acceptable image} \\ \text{e.g. } 10 \text{ if unacceptable image} \end{cases}$$

Comparing to human level performance

Why human-level performance?



Why progress often slows down:

- Human-level performance for many tasks not far off from Bayes optimal error
- If ML is worse than humans, we can:
  → Get labeled data from humans
  → Gain insight from manual error analysis
  → Better analysis of bias/variance

Comparing to human-level performance

Avoidable Bias

		0%	0%
e.g	Humans (≈Bayes)	1%	7.5%
	Training error	8%	8%
	Dev error	10%	10%
		↓	↓
		Focus on Bias	Focus on Variance
	Biggest gap:	1% → 8%	8% → 10%

Bias

↕ Avoidable Bias

↕ Variance

Human-level error acts as a proxy/estimate for Bayes error

Comparing to human-level performance

Understanding human-level performance

Human-level error as a proxy for Bayes error

e.g Medical Image Classification example:

(a) Typical human 3% error (c) Experienced doctor 0.7% error

(b) Typical doctor 1% error (d) Team of experienced doctors 0.5% error

- What is "human-level" error? Bayes error ≤ 0.5%

As machine learning algorithm is doing really well, ML project progress gets harder. Pro human-level performance becomes harder to tease out the bias and variance effects

Human	1% 0.7% 0.5%	1% 0.7% 0.5%	0.7% 0.5%
	↕ 4-4.5%	↕ 0-0.5%	↕ 0-0.2%
Training error	5%	1%	0.7%
	↕ 1%	↕ 4%	↕ 0.1%
Dev error	6%	5%	0.8%
	↑	↑	Hard!
	Bias	Variance	

Comparing to human level performance

Surpassing human-level performance

Team of humans	0.5%
Training error	0.3%
Dev error	0.4%

What is avoidable bias? Very unclear. Its hard for us to know if we should focus on teasing out bias or variance effects.

- Online advertising
- Product recommendations
- Logistics (predicting transit time)
- Loan approvals

- These all learn from structural data. Nothing to do with natural perception, which we humans are really good at.
- Lots of data

Comparing to human-level performance

Improving your model performance

The two fundamental assumptions of supervised learning:
1. You can fit the training set pretty well ~Avoidable Bias
2. The training set performance generalizes pretty well to the dev/test set. ~Variance

Avoidable Bias:
→ Train bigger model
→ Train longer/better optimization algorithms
→ NN architecture/hyperparameters search (RNN, CNN etc)

Variance:
→ More data
→ Regularization (L2, dropout, data augmentation)
→ NN architecture/hyperparameters search

Week 2 - ML Strategy (2)

Error Analysis

Error Analysis

- Evaluate multiple ideas in parallel. Ideas:
  - Fix pictures of dogs being recognized as cats.
  - Fix great cats (lions, panthes, etc...) being misrecognized
  - Improve performance on blurry images

	Image	Dog	Great Cats	Blurry	Comments
Dev set	1	✓			Pitbull
	2			✓	
	3		✓	✓	Rainy day at zoo
	⋮				
	% of total	8%	43%	61%	

Can have separate teams working on them

↑ It's okay to start adding new filters or categories

Error Analysis

Cleaning up incorrectly labeled data

- DL algorithms are quite robust to random errors in the training set.
  - Less robust to systematic errors.

- If Dev set

Image	Dog	GreatCat	Blurry	Incorrectly Labeled ↓	Comments
...					
98				✓	labeler missed cat in backgr.
99		✓			
100				✓	Drawing of cat, not real
% of total	8%	43%	61%	6%	

	if		if	
Overall dev set error		10%		2%
Errors due incorrect labels		0.6%		0.6% ← much more worthwhile to fix
Errors due to other causes		9.4% ↑ Focus on this		1.4%

- Goal of dev set is to help you select between two classifiers A & B. To know which one to trust, it's good to first eliminate incorrect labels

- Correcting incorrect dev/test set examples:
  - Apply same process to your dev and test sets to make sure they continue to come from the same distribution
  - Consider examining examples your algorithm got right as well as ones it got wrong to avoid biased advantages.
  - Train and dev/test data may now come from slightly different distributions.

Error Analysis

Build your first system quickly, then iterate

- Speech recognition example. Algorithm could be built to be more robust to:
  - Noisy background
  - Accented speech
  - Far from microphone
  - Young children's speech
  - Stuttering

But which one to focus on? Build system quickly then iterate!
- Set up dev/test set and metric
- Use Bias/Variance analysis & Error analysis to prioritize next steps.

- Less strongly applicable to domains we're experts in or in domains with huge literature available.

Mismatched training and dev/test set

Training and Testing on different distributions

- Cat app example
  - Data from webpages (web crawl) ≈ 200,000
  - Data from mobile app (users - what we care about) ≈ 10,000

Good option

Train	Dev	Test
200,000 web + 5,000 app	2,500 app	2,500 app

optimises on the mobile app data as it's the whole dev set

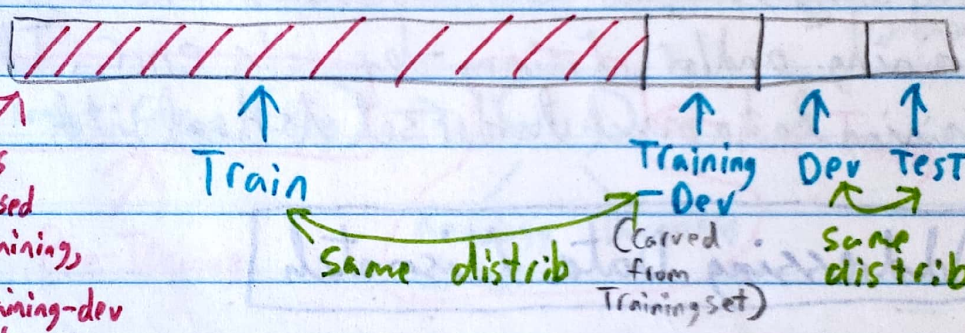
Mismatched training and dev/test set

Bias and Variance with mismatched data distributions

Example training error 1%
dev error 10% ↕ 9%

Is it a variance problem or mismatched distributions

→ Create Training-Dev set: Same distribution as training set but not used for training



For example Training error 1%
Training-dev err 9% ↕ Variance error!
dev error 10%

Training error 1%
Training-dev err 1.5% ↕ Data mismatch problem!
dev error 10%

Key Quantities to look at

Human level 0%
Training error 10% ↕ Avoidable bias
Training-dev error 11%
dev error 12%

Could be a combination of both

Human-level
Training set error
Training-dev set error
Dev error
Test error

↕ Avoidable Bias
↕ Variance
↕ Data Mismatch
↕ Degree of overfitting to dev set
(Get bigger dev set)

- It could happen that dev/test error is lower than training and/or training-dev sets error. This means training had a much harder set to deal with

Mismatched training & dev/test set

Addressing Data Mismatch

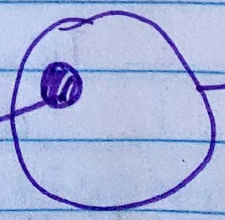
- Carry out manual error analysis to try to understand difference between training and dev/test sets. e.g. noisy data

- Make training data more similar; or collect more data similar to dev/test sets e.g. simulate noisy in-car data

→ Artificial Data Synthesis (When you need more specific data)

e.g. Man talking + Car Noise = Synthesized in-car audio

Need to be careful



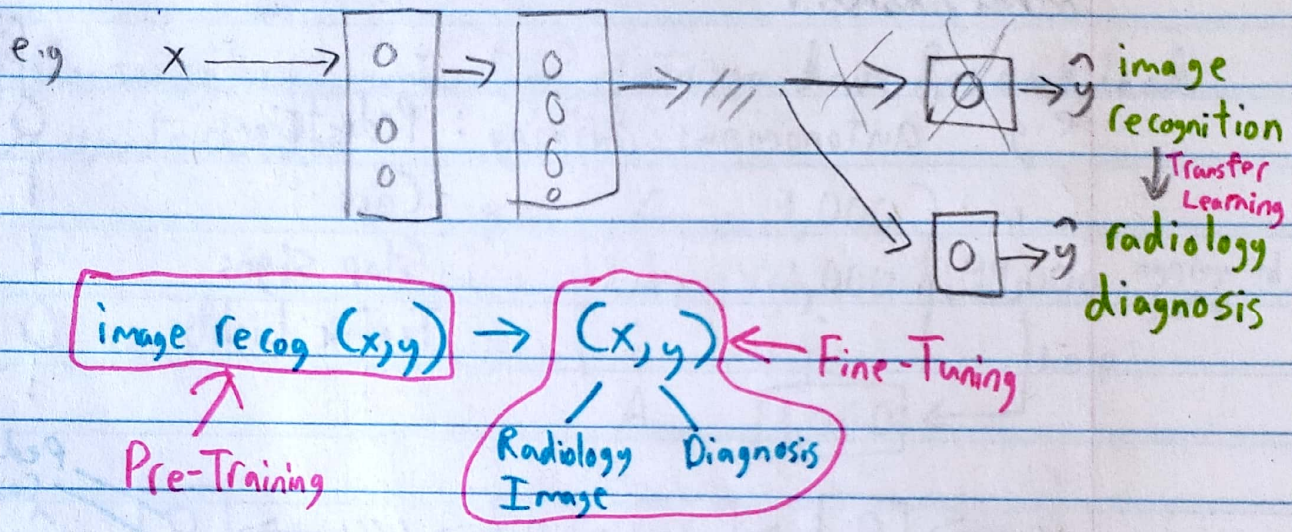
Synthesized

All cars or all audio in car (or all possibilities)

Learning from multiple tasks

Transfer Learning (Done Sequentially)

- Take knowledge the neural network has learned from one task and apply that knowledge to a separate task.



- Transfer learning makes sense when you have a lot of data for the problem you're transfering from and usually relatively less data for the problem you're transfering to.

e.g image recognition 1,000,000 data
radiology diagnosis 100 data

- When task A and B have the same input x. (both are images)
- You have a lot more data for Task A than Task B.
- Low level features from A could be helpful for learning B.

- Used very often

Learning from multiple tasks

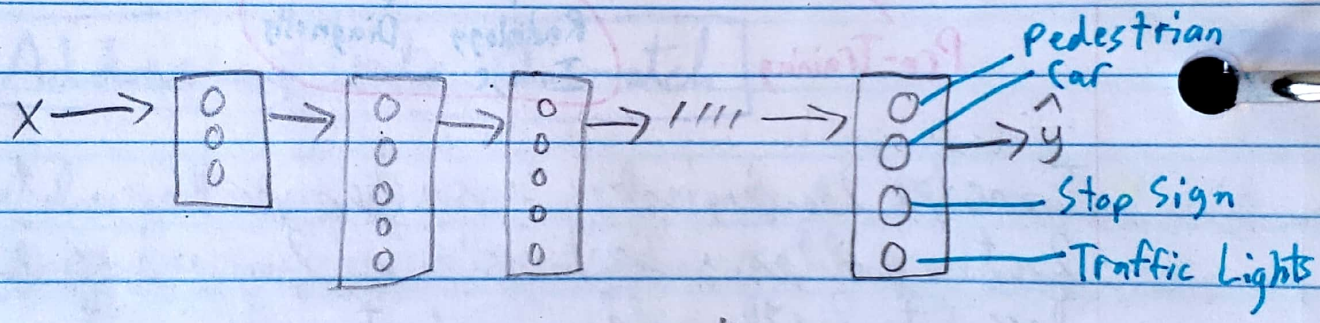
Multi-task learning

- Simultaneously have one neural network do several things at the same time, and then each of these tasks helps all the other tasks.

e.g. autonomous driving : Pedestrians, Cars, Stop signs, Traffic Lights

$y^{(i)}$ $(4,1)$

$$y^{(i)} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ \vdots \end{bmatrix}$$



Loss: $\hat{y}^{(i)}$ $(4,1)$

$$\frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{4} \mathcal{L}(\hat{y}_j^{(i)}, y_j^{(i)})$$

main difference

↳ Usual logistic loss $-y_j^{(i)} \log \hat{y}_j - (1-y) \log (1-y^{(i)})$

≈ Image can have multiple labels

[Unlike softmax regression where one image can have only 1 label]

- If some of the earlier features in neural network can be shared between these different types of objects, then you find that training one neural network to do four things results in better

performance than training four completely separate neural networks to do the four tasks separately.

- Works even when some $y_i^{(1)}$ values are unlabeled

- Makes sense when amount of data you have for each task is quite similar

e.g.

A_1	1,000	99,000	aggregated data
A_2	1,000		
⋮	⋮		help
A_{100}	1,000		

- Makes sense when you can train a big enough neural network to do well on all the tasks.
  → It actually hurts performance when network not big enough.

End-to-end deep learning

What is end-to-end deep learning?

- Traditional Pipeline: audio (x) —MFCC→ features —ML→ Phonemes → Words → Transcript
  End-to-end learning: audio → transcript
  ↑ works really well when you have lots of data
  → Sometimes breaking things into 2 sub-problems works better than pure end-to-end DL approach

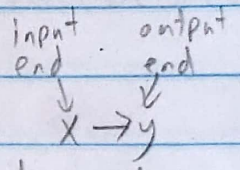
- Nowadays works well with machine translation e.g. English → French, not so well with estimating child's age from hand X-ray so (lack of data) Image → Bones → Age instead of Image → Age

End-to-end deep learning

Whether to use end-to-end deep learning

- Pros:
  - Let the data speak (when you have enough data)... it does its own $x \to y$ mapping rather than being forced to human misconceptions
  - Less hand-designing of components needed

- Cons:
  - May need large amount of data
  - Excludes potentially useful hand-designed components



- Key question: Do you have sufficient data to learn a function of the complexity needed to map x to y?

Coursera Deep Learning Specialization Course #4

Coursera: Convolutional Neural Networks

Week 1 : Foundations of CNNs

★ Vertical edge detection

3	0	1	2	7	4
1	5	8	9	3	1
2	7	2	5	1	3
0	1	3	1	7	8
4	2	1	6	2	8
2	4	5	2	3	9

6x6

n x n

* "convolution"

1	0	-1
1	0	-1
1	0	-1

3x3

Filter/Kernel

f x f

=

-5	-4	0	8
-10	-2	2	3
0	-2	-4	-7
-3	-2	-3	-16

4x4

n-f+1 x n-f+1

e.g

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

6x6

*

1	0	-1
1	0	-1
1	0	-1

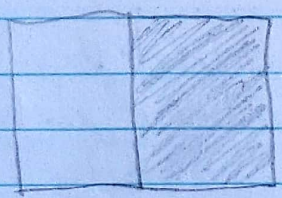
3x3

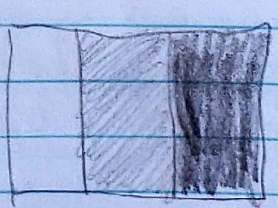
Filter/Kernel

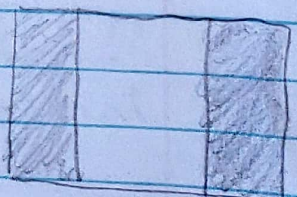
=

0	30	30	0
0	30	30	0
0	30	30	0
0	30	30	0

4x4

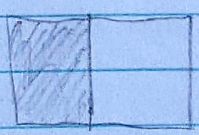


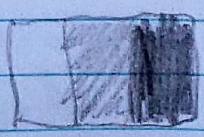




Indicates high likelihood of v edge in that area

if





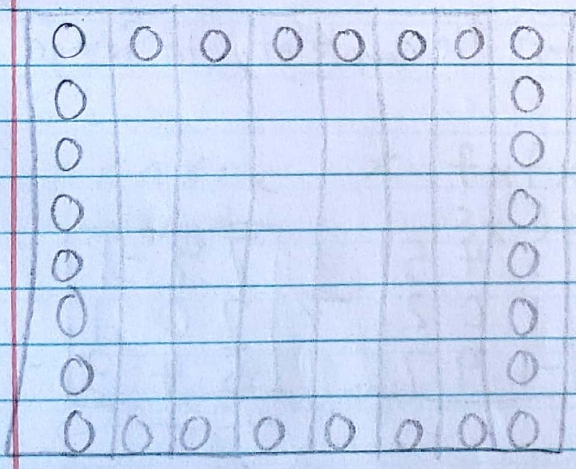
- Horizontal Edge Filter/Kernel

1	1	1
0	0	0
-1	-1	-1

- Different weights & filters could be used for vertical & horizontal edge detections
  - → Sometimes those numbers can be learned
  - → Can be also used to detect edges at angles e.g. 45°, 70°

- ★ To ensure numbers on the edge of an image gets fashioned in as much as numbers in other places of the image, we introduce **padding**
  - → Solves shrinking output
  - → Solves throwing away lots of info from edges

e.g. padding = 1



$6 \times 6 \rightarrow 8 \times 8$

$* \quad 3 \times 3 \quad = \quad 4 \times 4 \rightarrow 6 \times 6$

$n + 2p - f + 1$

$6 + 2 - 3 + 1$

$= 6$

- "Valid Convolution": $n \times n \quad * \quad f \times f \quad \rightarrow \quad n-f+1 \times n-f+1$

  $6 \times 6 \qquad 3 \times 3 \qquad\qquad 4 \times 4$

- "Same Convolution": Pad so that output size is the same as the input size

$$p = \frac{f-1}{2}$$

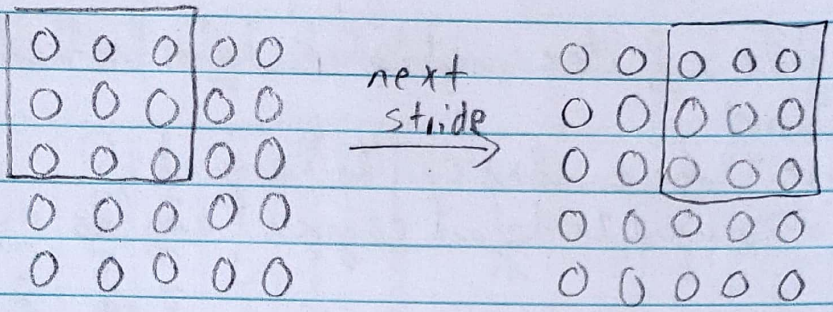
f is usually odd

$f = 3 \times 3 \quad p = 1$

$f = 5 \times 5 \quad p = 2$

★ Stided Convolution

eg stride = 2



$[z]$ = floor (z)

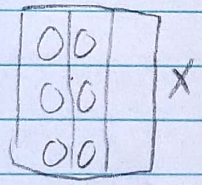
$n \times n \quad * \quad f \times f$

padd$y$ p stride$s$

$$\left[\frac{n+2p-f}{s}+1\right] \times \left[\frac{n+2p-f}{s}+1\right]$$

e.g 7×7 3×3 $p=0$ $s=2$

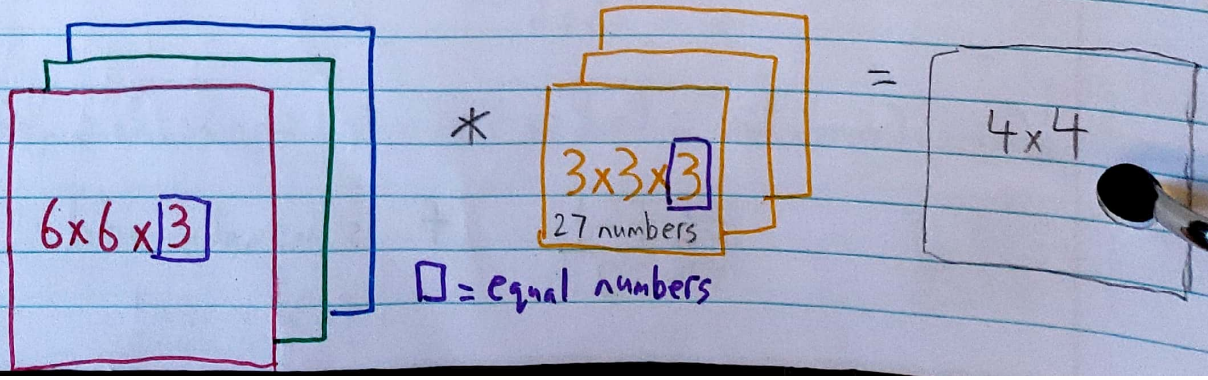
$$\frac{7+0-3}{2}+1=3 \quad \therefore 3 \times 3$$

if  filter must lie completely in image

- Convolution in maths is to $\begin{bmatrix} 3 & 4 & 5 \\ 1 & 0 & 2 \\ -1 & 9 & 7 \end{bmatrix} \rightarrow \begin{bmatrix} 7 & 9 & -1 \\ 2 & 0 & 1 \\ 5 & 4 & 3 \end{bmatrix}$

What we do in ML is called cross-correlation in maths, but ML results are the same so flipping isn't usually done

★ Convolutions on RGB images (volumes)



e.g. if we want to detect only Red v edges

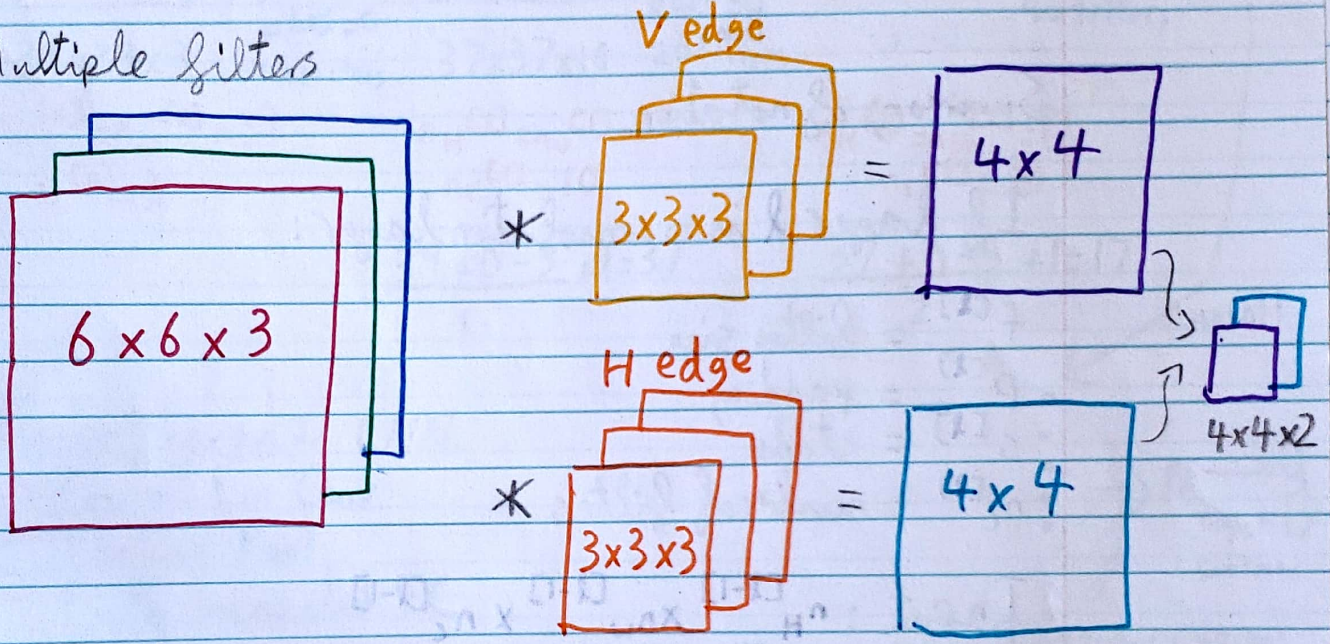
R

1	0	-1
1	0	-1
1	0	-1

G & B

0	0	0
0	0	0
0	0	0

- Multiple filters



$n \times n \times n_c$ * $f \times f \times n_c$ → $n-f+1 \times n-f+1 \times n_c'$

$6 \times 6 \times 3$ $3 \times 3 \times 3$ $4 \times 4 \times 2$ (2 ↑ # filters)

- Example of a layer

(using above figures)

$a^{[0]}$ $w^{[1]}$ $w^{[1]} a^{[0]} = z^{[1]}$

$= \text{ReLU}\left(\begin{bmatrix}4\times4\end{bmatrix} + b_1\right) \rightarrow 4\times4$

$\text{ReLU}(\ldots + b_2) \rightarrow 4\times4$

$= 4 \times 4 \times 2$ $a^{[1]}$

$z^{[1]} = w^{[1]} a^{[0]} + b^{[1]}$

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

in [] indicates layer

$6\times6\times3$ ($a^{[0]}$) → $4\times4\times2$ ($a^{[1]}$)

- Number of parameters in one layer

e.g. 10 filters $3 \times 3 \times 3$ in one NN layer params?

$3 \times 3 \times 3 + 1 = 28$ (↑ Bias) $\qquad 28 \times 10 = 280$ //

- Summary of notation

If layer l is a convolution layer:

$f^{[l]}$ = filter size
$p^{[l]}$ = padding
$s^{[l]}$ = stride
$n_c^{[l]}$ = number of filters

Input : $n_H^{[l-1]} \times n_w^{[l-1]} \times n_c^{[l-1]}$
Output : $n_H^{[l]} \times n_w^{[l]} \times n_c^{[l]}$

$$n_{H \text{ or } W}^{[l]} = \left\lfloor \frac{n^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$$

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

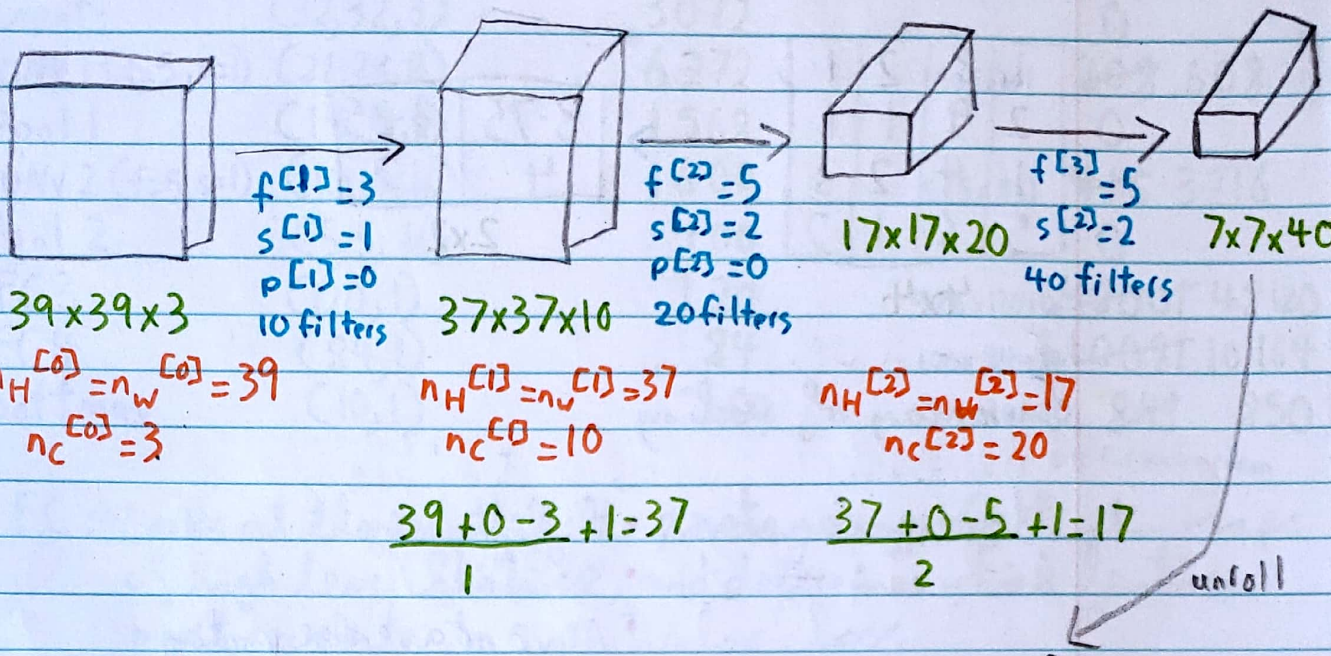
Activations : $a^{[l]} \rightarrow n_H^{[l]} \times n_w^{[l]} \times n_c^{[l]}$ $\qquad A \rightarrow m \times a^{[l]}$

Weights : $f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$

Bias : $n_c^{[l]} - (1, 1, 1, n_c^{[l]})$

→ The n_i can sometimes be $n_c \times n_H \times n_w$.
As long as we are consistent

Example ConvNet



$n_H^{[0]} = n_W^{[0]} = 39$
$n_C^{[0]} = 3$

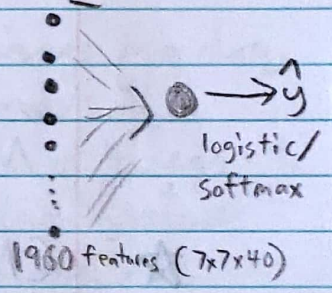
$n_H^{[1]} = n_W^{[1]} = 37$
$n_C^{[1]} = 10$

$n_H^{[2]} = n_W^{[2]} = 17$
$n_C^{[2]} = 20$

$\frac{39+0-3}{1}+1 = 37$

$\frac{37+0-5}{2}+1 = 17$

unroll



Types of layers in CNN
- → Convolution Conv
- → Pooling Pool
- → Fully connected FC

Pooling layers

-Max pooling (works well... proven in so many experiments)

1	3	2	1
2	9	1	1
1	3	2	3
5	6	1	2

f=2
s=2

9	2
6	3

4x4 → 2x2

Basically the max from each region

1	3	2	1	3
2	9	1	1	5
1	3	2	3	2
8	3	5	1	0
5	6	1	2	9

f=3
s=1

9	9	5
9	9	5
8	6	9

5x5 → 3x3

$\frac{n+2p-f}{s}+1$ $\quad \frac{5-3}{1}+1 = 3\times3$

- Average Pooling

1	3	2	1
2	9	1	1
1	4	2	3
5	6	1	2

4x4 → | 3.75 | 1.25 |
|---|---|
| 4 | 2 |

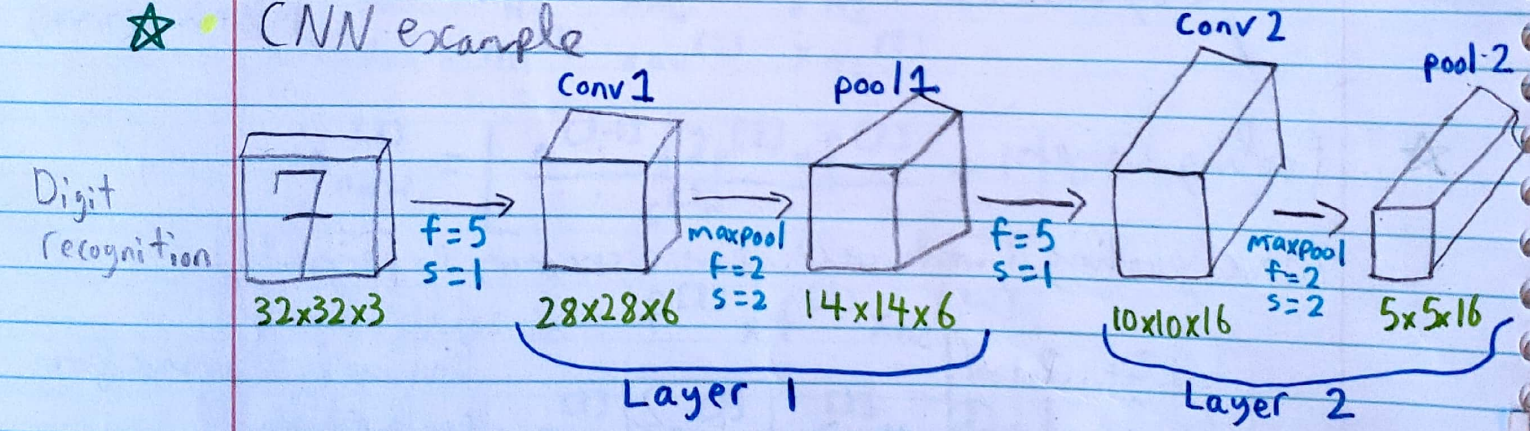
2x2

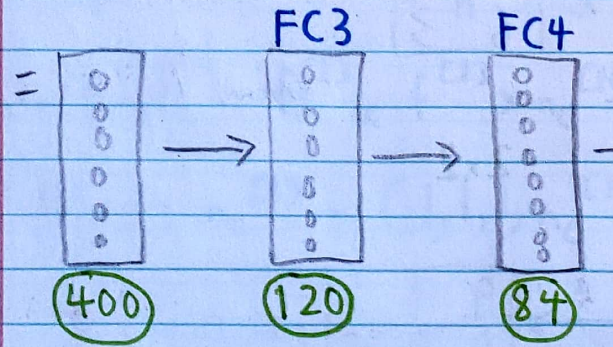
- Summary of pooling

Hyperparameters:
- f: filter size
- s: stride
- Max or average pooling
- p: padding (not usually with max pooling)

No parameters to learn!

- CNN example





→ softmax

if output 0→9
10 outputs

$n_H, n_W \downarrow$
$n_C \uparrow$

$f^{(l)} \times f^{(l)} \times n_c^{(l-1)} + b$

	Activation Shape	Activation Size	# parameters
Input:	(32,32,3)	3072	0
CONV1 (f=5, s=1)	(28,28,8)	6272	5x5x3+1 ~~208~~ 608
Pool 1	(14,14,8)	1568	0
CONV2 (f=5, s=1)	(10,10,16)	1600	5x5x8+1 ~~416~~ 3216
Pool 2	(5,5,16)	400	0
FC3	(120,1)	120	400x120+120 ~~48001~~ 48120
FC4	(84,1)	84	120x84+84 ~~10081~~ 10164
Softmax	(10,1)	10	84x10+10 ~~841~~ 850

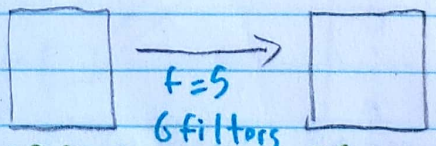
↑ Bias for each neuron

- FC :→ Looks at the output of the previous layer (activation maps of high level features) and determines which features most correlate to a particular class.
  - → e.g if the program is predicting that some image is a dog, it will have high values in the activation maps that represent high level features like a paw or 4 legs etc
  - → Basically a FC layer looks at what high level features most strongly correlate to a particular class and has particular weights so that when you compute the products between the weights and previous layer, you get the correct probabilities for the different classes.

(Backpropagation makes all this work)

- Why convolutions?

e.g



Instead of $3072 \times 4704 \approx 14M$

only $(5*5*3+1) \times 6 = 456$

$f^{(l)}\ f^{(l)}\ n_c^{(l-1)}\ b\ \ n_c^{(l)}$

→ 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.

e.g 3x3 filter/parameter $\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$ across each part of the image

→ Sparsity of connections: In each layer, each output value depends only on a small number of inputs

→ Very good at capturing translation invariance (a shifted image) ≈ still will be able to detect features

- Less prone to overfitting
- Can work with smaller training sets

Training set $(x^{(1)}, y^{(1)}) \dots (x^{(m)}, y^{(m)})$

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

Use GD or RMSprop or ADAM or something to optimize params & reduce cost J

Week 2 - Deep CNN: Case Studies

case studies

Why look at case studies?

- The same way we learn how to code is by looking at other people's code, we can learn from or use other built CNN models for our task in hand

- Classic Networks:
  - LeNet-5
  - AlexNet
  - VGG

- ResNet

- Inception

Classic Networks

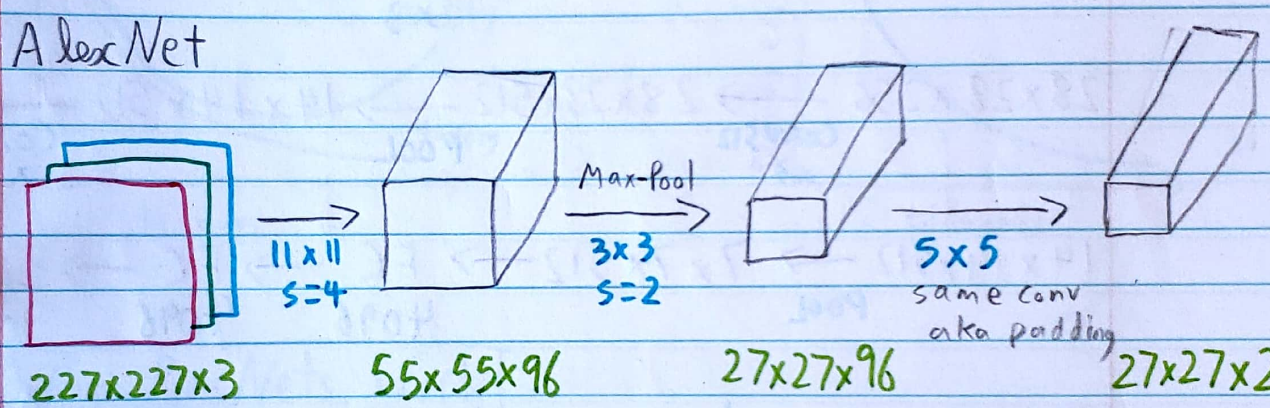
- LeNet-5 : Used originally to detect handwritten digits (1998)

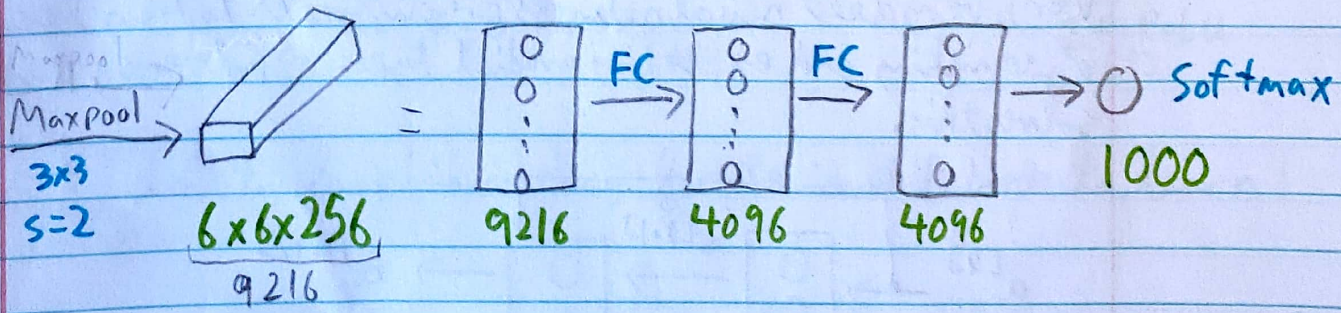
- Same exact model as the CNN model 2 pages prior, but instead of max pool avg pool was used as it was more common back then and input was 32x32x1 as opposed to 32x32x3, as image was grayscale.

≈ 60K parameters

- Sigmoid / tanh instead of ReLU

- AlexNet





≈ 60 million parameters

- ReLU
- Local Response Normalization (not used often nowadays)

VGG-16

- CONV = 3×3 filter, s=1, same
  MAX-POOL = 2×2, s=2

$224 \times 224 \times 3 \xrightarrow{\text{CONV64 ×2}} 224 \times 224 \times 64 \xrightarrow{\text{POOL}} 112 \times 112 \times 64 \xrightarrow{\text{CONV128 ×2}}$

$112 \times 112 \times 128 \xrightarrow{\text{POOL}} 56 \times 56 \times 128 \xrightarrow{\text{CONV256 ×3}} 56 \times 56 \times 256 \xrightarrow{\text{POOL}}$

$28 \times 28 \times 256 \xrightarrow{\text{CONV512 ×3}} 28 \times 28 \times 512 \xrightarrow{\text{POOL}} 14 \times 14 \times 512 \xrightarrow{\text{CONV512 ×3}}$

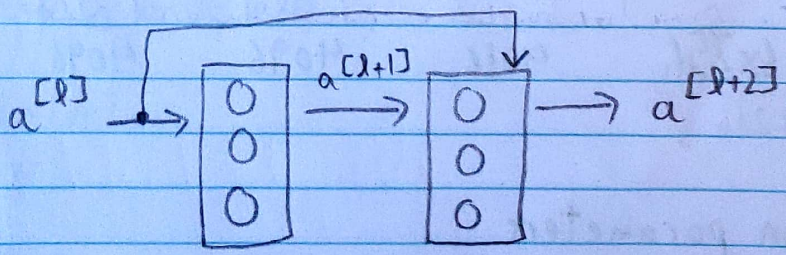
$14 \times 14 \times 512 \xrightarrow{\text{POOL}} 7 \times 7 \times 512 \xrightarrow{\text{FC 4096}} \text{FC } 4096 \xrightarrow{\text{FC 1000}} \text{Softmax}$

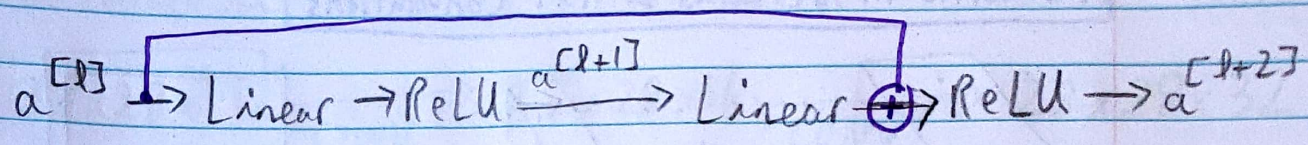
≈ 138M parameters

Case Studies

ResNets

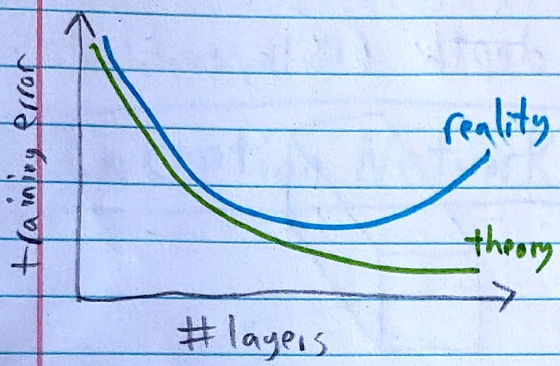
- Very, very deep neural networks are difficult to train because of vanishing and exploding gradient types of problems. ResNets solves this.



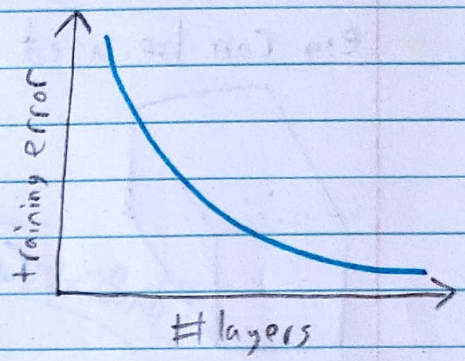


$$a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$$

Plain Network



ResNet



Case Studies

Why ResNets Work

usually same dimensions. If not, multiply $a^{[l]}$ by a matrix to make it same dimensions

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

$$= g(W^{[l+2]} a^{[l+1]} + b^{[l+2]} + a^{[l]})$$

If = 0 e.g. from weight decay

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

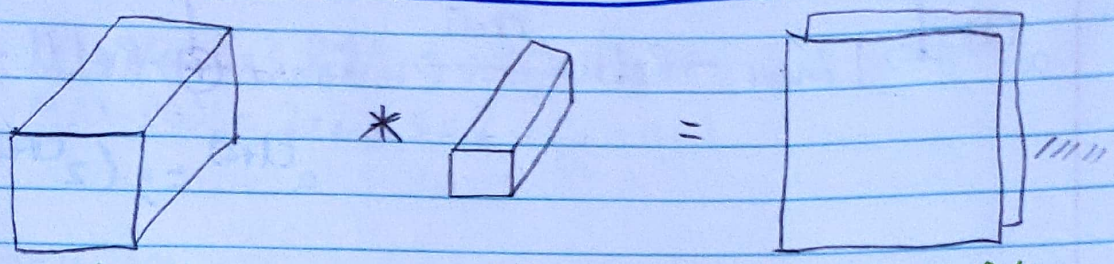
If ReLU

$$= a^{[l]}$$

→ Identity function is easy for Residual Block to learn!

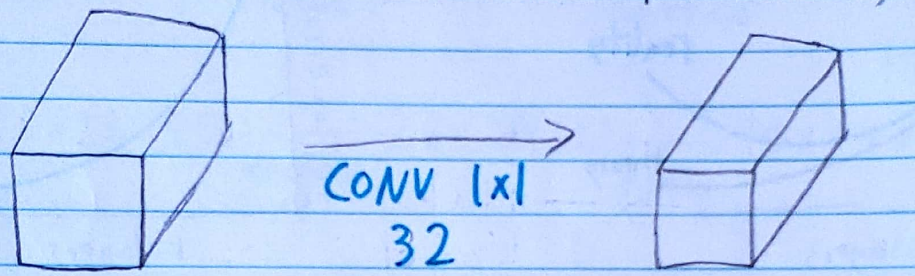
Case Studies

Networks in Networks & 1x1 Convolutions



6x6x32 * 1x1x32 = 6x6 x # filters

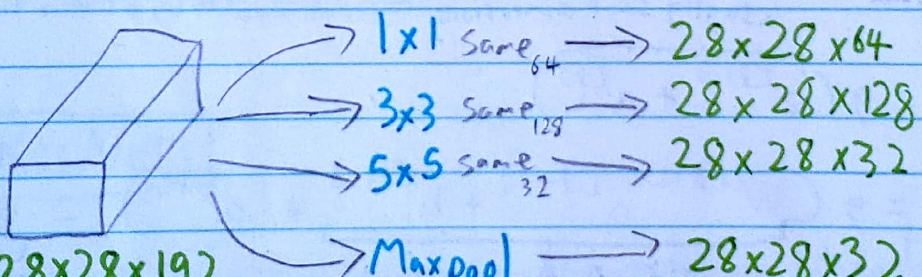
- e.g. Can be used to shrink depth (Pooling only shrinks H & W)



28x28x192 → 28x28x32

Case Studies

Inception Network Motivation



28x28x192 → ≈ 28x28x256

- Instead of deciding what to do, perform different operations and concatenate outputs. Let model decide then what to do later.

- Computational cost problem

e.g. 28x28x192 → (Conv 5x5 same 32) → 28x28x32

28x28x32 x 5x5x192 (filters)

= 120M parameters

To reduce it : Use 1x1 Convolution.

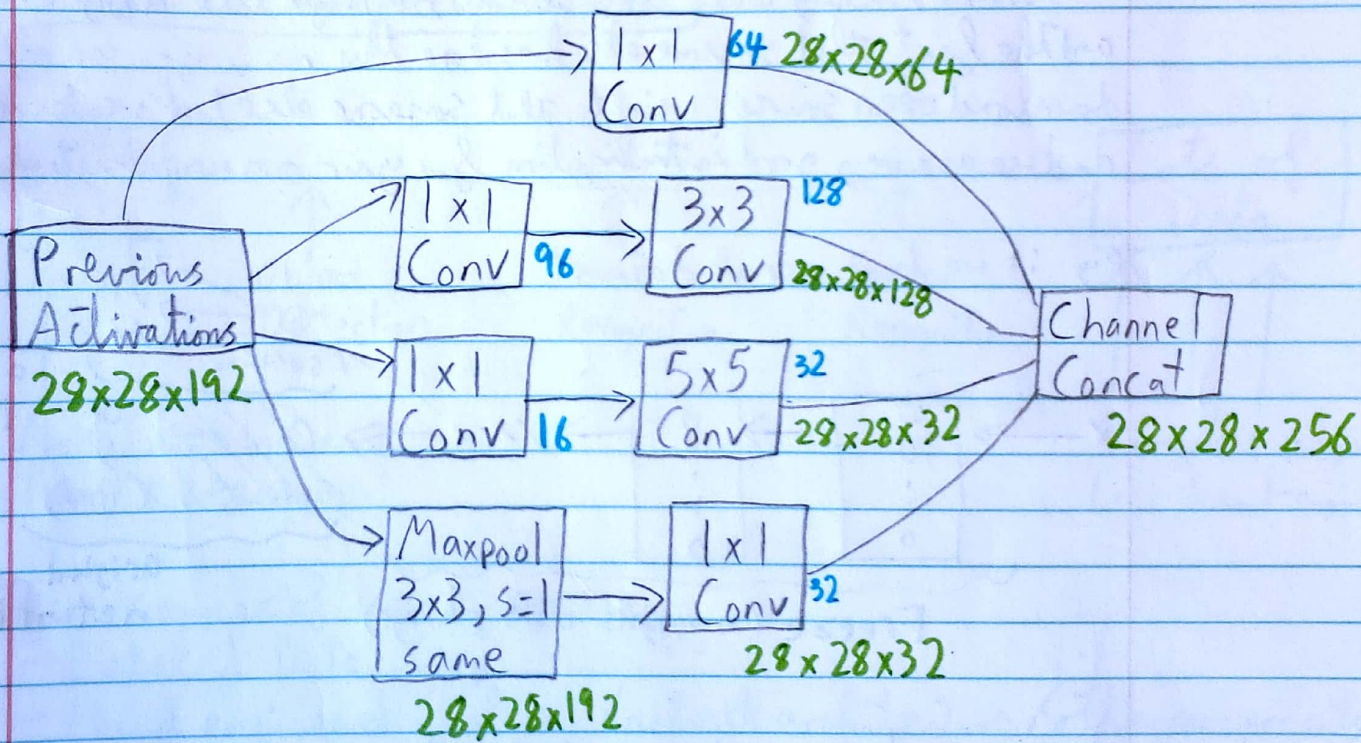
$28 \times 28 \times 192 \xrightarrow{\text{CONV } 1\times1,\ 16,\ 1\times1\times192} 28 \times 28 \times 16 \xrightarrow{\text{CONV } 5\times5,\ 32,\ 5\times5\times16} 28 \times 28 \times 32$

"Bottleneck layer"

$28 \times 28 \times 16 \times 1 \times 1 \times 192 = 2.4M$
$28 \times 28 \times 32 \times 5 \times 5 \times 16 = 10M$
$\rightarrow$ 12.4M parameters

Case studies

Inception Network



↑ 1 inception module. The network puts these modules together.

Practical Advices

Using open-source implementations

- Replicating other people's work would be much faster if their code is available open-source e.g on Github git clone.....
  → Could be already trained on extensive computation resources

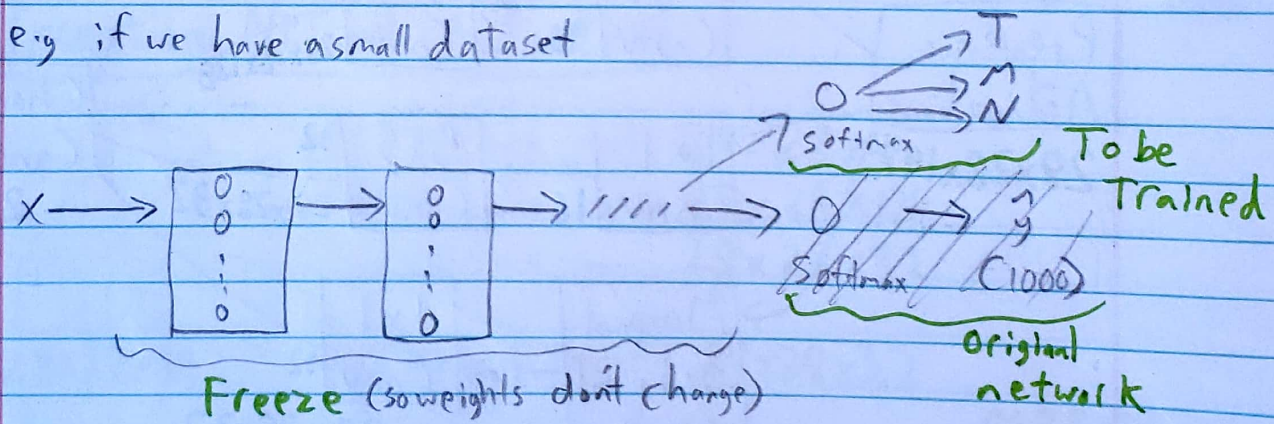
Practical Advices

Transfer Learning

- Often make much faster progress if you download weights that someone else has already trained on the network architecture and use that as pre-training and transfer that to a new task that you might be interested in.

- Sometimes training could take several and might take many GPUs and the fact that someone else has done this means you can often download open source weights that someone else took weeks or months and use as a very good initialization for your own neural network.

- e.g if we have a small dataset



- e.g if we have a larger dataset

→ Freeze fewer layers
→ Number of layers to be trained on top increases

- e.g if we have a HUGE dataset

→ We don't have to freeze any layers and use weights as initialization

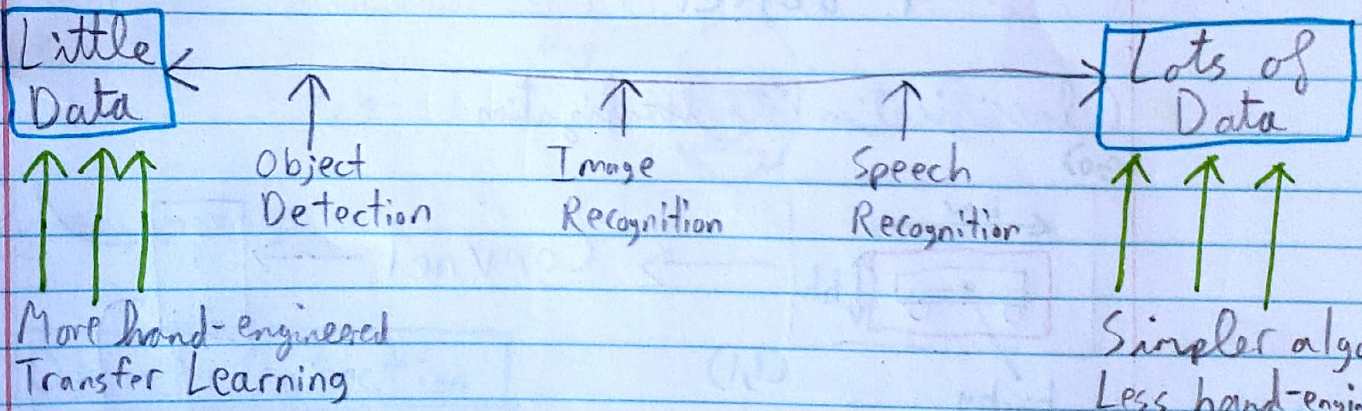
Practical Advices

Data Augmentation

- Used to improve performance of computer vision systems
- Common methods:
  - → Mirroring
  - → Random Cropping
  - → Rotation
  - → Shearing
  - → Local Warping
  - → ....
  - → Color Shifting

Practical Advices

State of Computer Vision

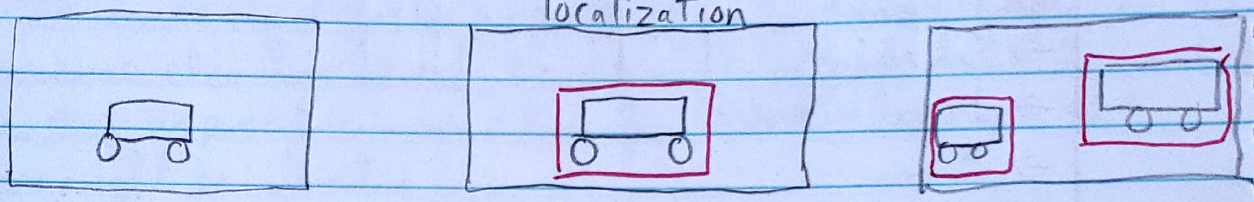


- Two sources of knowledge:
  - Labeled Data (x,y)
  - Hand engineered features / network architecture / other components
- Tips for doing well on benchmarks / winning competitions
  - Ensembling
    - → Train several networks [3-15 networks] independently and average their outputs
  - Multi-crop at test time
    - → Run classifier on multiple versions

Week 3 - Detection Algorithms

Object Localization

- Image Classification | Classification with localization | Detection

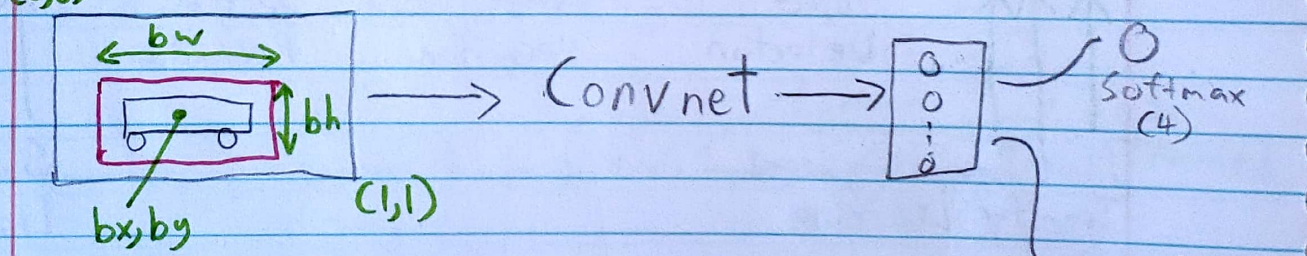


"Car" "Car"

1 object

multiple objects

- Classification with localization



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 any obj?

e.g. car

$$\begin{bmatrix} 1 \\ b_x \\ b_y \\ b_h \\ b_w \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

background

$$\begin{bmatrix} 0 \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \end{bmatrix}$$ "don't care"

if squared loss

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

Landmark Detection

Person's face

So people's faces → Convnet → face? l_{1x} l_{1y} ..... l_{64x} l_{64y} — 129

l_{1x}, l_{1y}
l_{2x}, l_{2y}
⋮
l_{64x}, l_{64y}
} x, y

e.g.

Object Detection

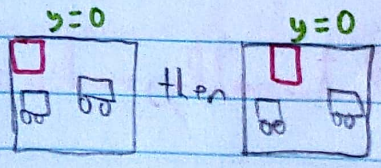
e.g. Car Detection example

Training set:

x	y
closely cropt pics of cars or background	1 - car 0 - no car

image → Convnet → y

Sliding Windows Detection



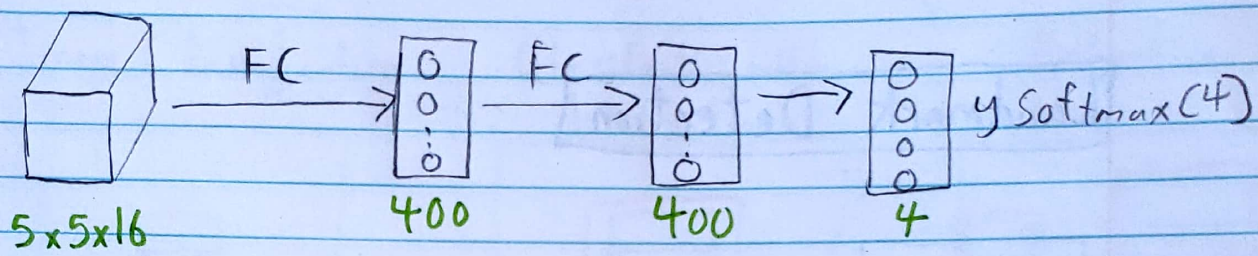
& so on till window slid across every position in the image

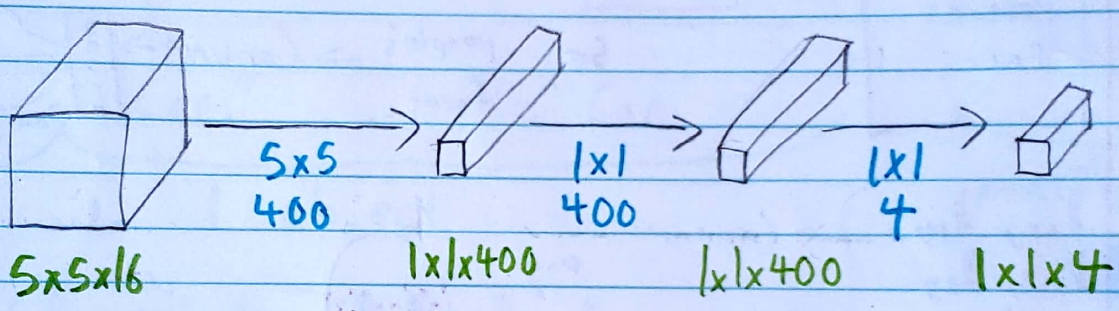
Then bigger windows

Computation cost : HIGH!

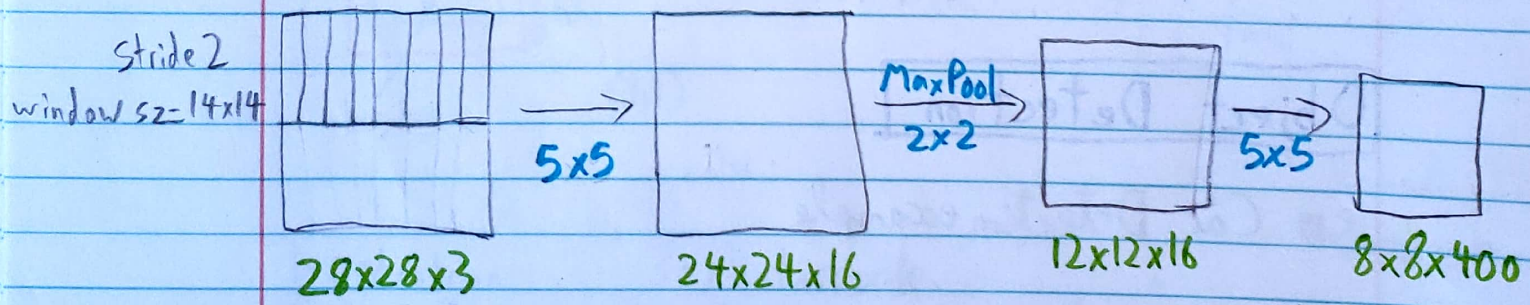
Convolutional Implementation of Sliding Windows

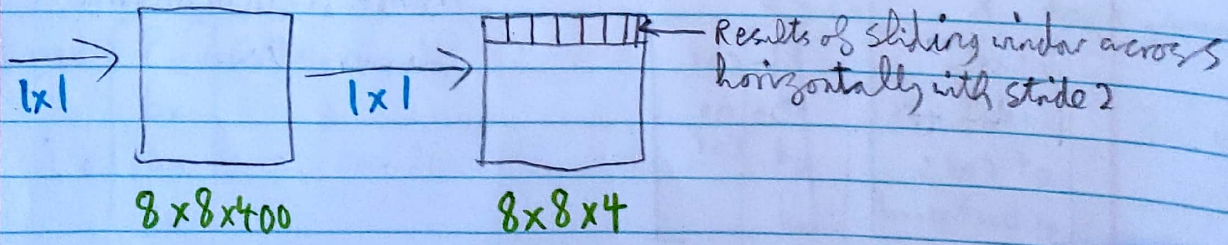
- Turning FC layer into convolutional layers





- Convolutional implementation of sliding windows

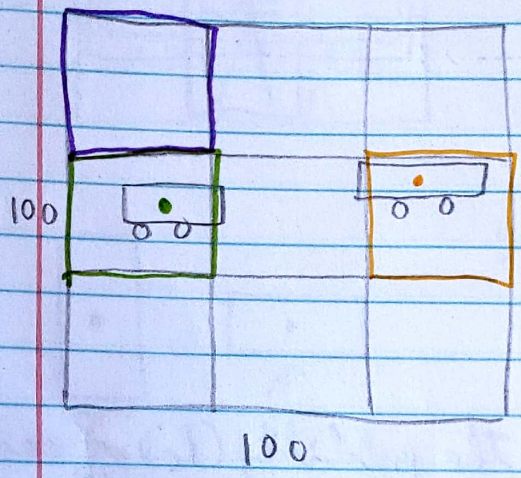




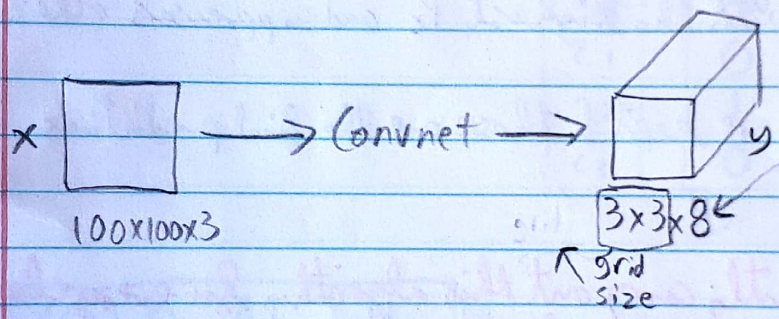
→ Convolutionally, make all predictions at the same time

Bounding Box Predictions

- YOLO Algorithm [You Only Look Once]



$$y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} \quad \begin{bmatrix} 0 \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \end{bmatrix} \quad \begin{bmatrix} 1 \\ b_x \\ b_y \\ b_h \\ b_w \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 1 \\ b_x \\ b_y \\ b_h \\ b_w \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

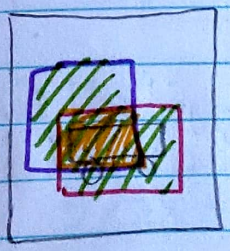


Usually finer grid to avoid having multiple objects in one box

b_x, b_y, b_h, b_w relative to grid size

Intersection Over Union [IoU]

- Evaluating object localization



- Predicted
- Union
- Intersection

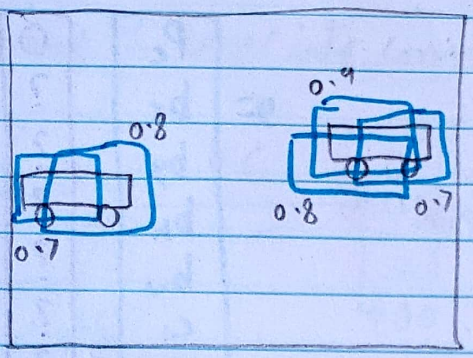
Intersection over union = size of ▨ / size of ▨

eg Correct if IoU ≥ 0.5

⇒ More generally, IoU is a measure of the overlap between 2 bounding boxes

Non-max suppression

- e.g 19x19 grid, can have multiple detections per car



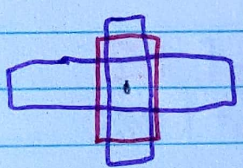
- Non-max suppression looks at the probability (P_c) of each box
- Highlights the one with the highest P_c and suppresses others with high IoU
- Highlighted ones are kept & those are the final predictions

∴ 1 box per car

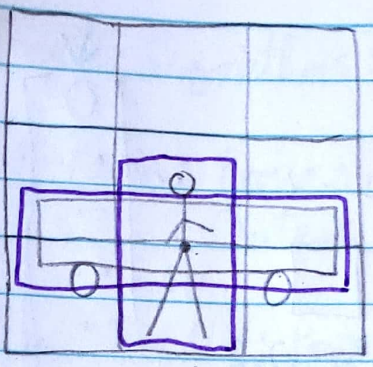
Note: Independently, carry out this algorithm for every class
e.g cars, pedestrians, busses .....

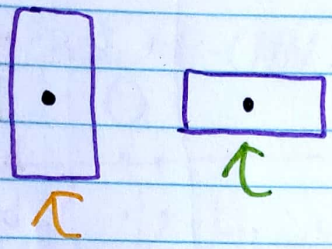
Anchor Boxes

- Previously:
  Each object in training image is assigned to grid cell that contains that object's midpoint. 3x3x8



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. 3x3x16
(3x3x2x8)





$y = \begin{bmatrix} P_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \\ P_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 1 \\ b_x \\ b_y \\ b_h \\ b_w \\ 1 \\ 0 \\ 0 \\ 1 \\ b_x \\ b_y \\ b_h \\ b_w \\ 0 \\ 1 \\ 0 \end{bmatrix}$

Car only?

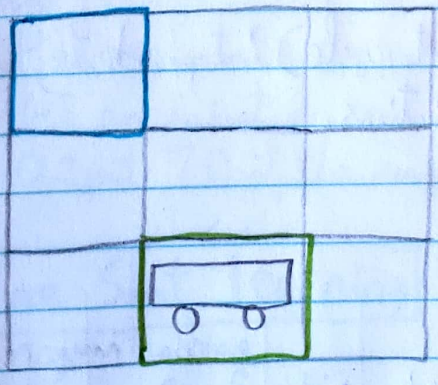
$\begin{bmatrix} 0 \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \\ 1 \\ b_x \\ b_y \\ b_h \\ b_w \\ 0 \\ 1 \\ 0 \end{bmatrix}$

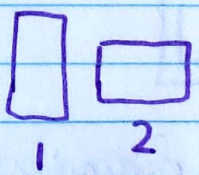
YOLO Algorithm

Putting all the components learnt together

- Training

1- pedestrian
2- car
3- motorcycle





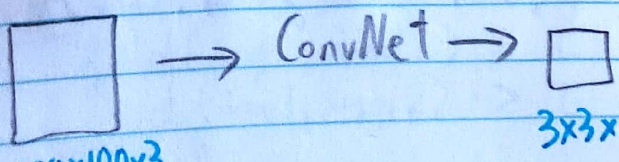
1 2

$y = \begin{bmatrix} P_c \\ b_x \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \\ P_c \\ b_x \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 0 \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \\ 0 \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \end{bmatrix}$

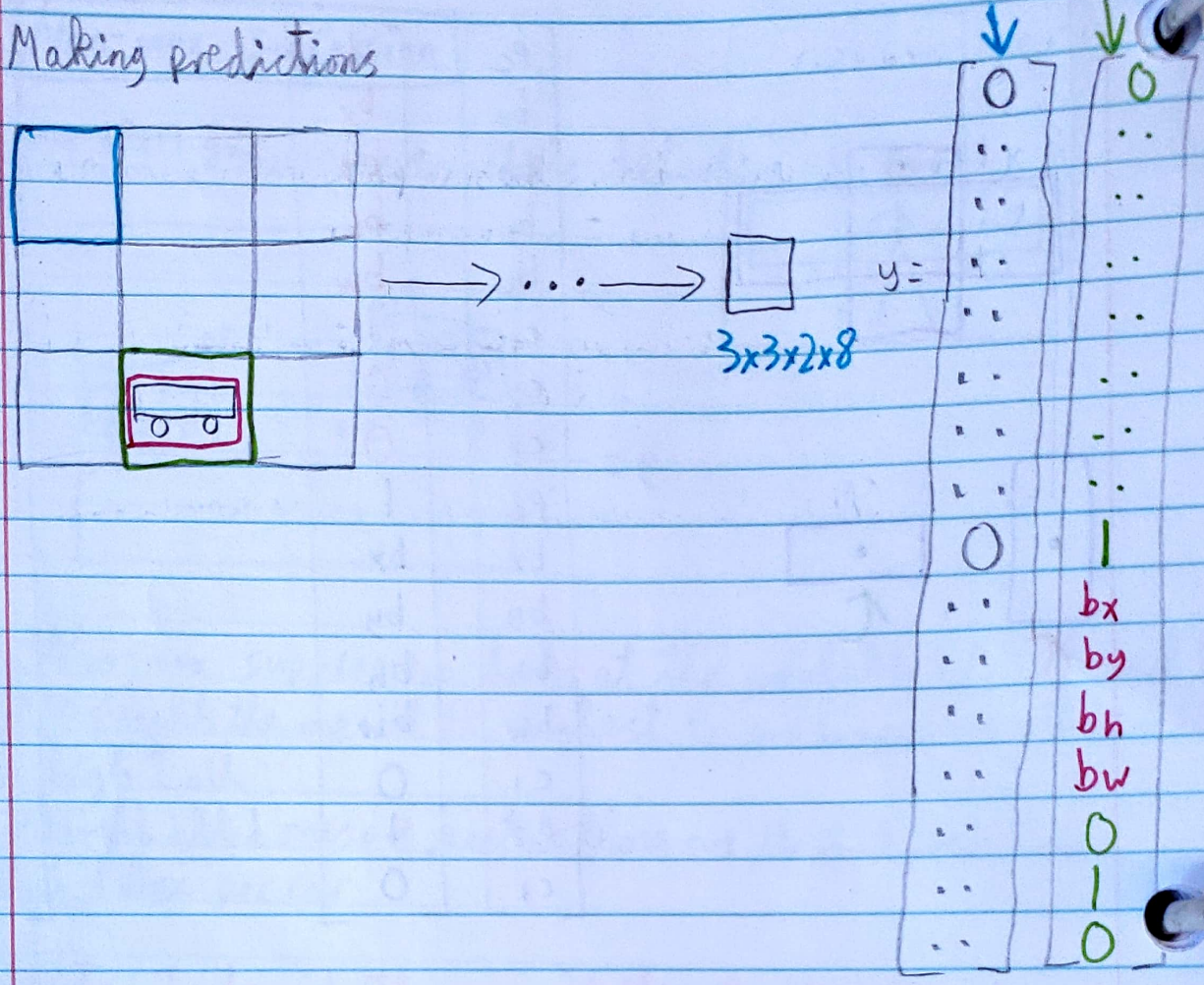
$\begin{bmatrix} 0 \\ ? \\ ? \\ ? \\ ? \\ 2 \\ ? \\ ? \\ 1 \\ b_x \\ b_y \\ b_w \\ 0 \\ 1 \\ 0 \end{bmatrix}$

y 3x3x16

(Bounding box needs to be provided)



- Making predictions



- Outputting the non-max supressed outputs
  - For each grid cell, get 2 predicted bounding boxes
  - Get rid of low probability predictions
  - For each class (pedestrian, car, motorcycle) independently, use non-max suppression to generate final predictions.

Region Proposal

- R-CNN (Region) : Instead of running sliding windows across all image parts only run detection algo or classifier on just a few windows / regions

→ Regions selected based on segmentation algo (similar to an image heatmap)

Faster algorithms:

- R-CNN: Propose regions. Classify proposed regions one at a time. Output label + bounding box.
- Fast R-CNN: Propose regions. Use convolutional implementation of sliding windows to classify all the proposed regions.
- Faster R-CNN: Use convolutional network to propose regions.

Week 4 - Face Recog & Neural style transfer

Face Recog

What is Face Recog?

- Face verification: 1:1
  - Input image, name/ID
  - Output whether the input image is that of the claimed person
- Face recognition:
  - Has a database of K persons 1:K
  - Get an input image
  - Output ID if the image is any of the K persons (or "not recognized")

Face Recog

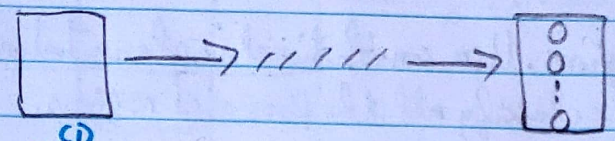
One Shot learning

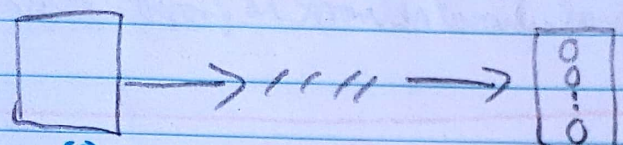
- Learning from one example to recognize the person again
- Learn a "similarity" function
  → d(img 1, img 2) = degree of difference between images
  If $d(img\,1, img\,2) \leq \tau$ "same"
  If $d(img\,1, img\,2) > \tau$ "different"
  } Verification

Face Recog

Siamese Network

- The idea of running two identical, CNN on two different inputs and then comparing them.





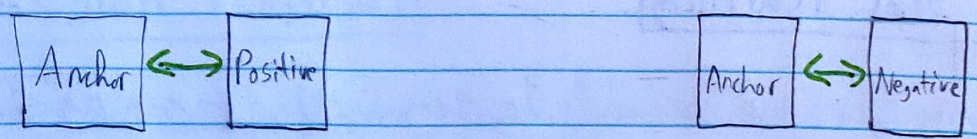
$$d(x^{(1)}, x^{(2)}) = \| f(x^{(1)}) - f(x^{(2)}) \|_2^2$$

- Goal of learning
  - → Parameters of NN define an encoding $f(x^{(i)})$
  - → Learn parameters so that:
    - If $x^{(i)}, x^{(j)}$ are the same person, $\| f(x^{(i)}) - f(x^{(j)}) \|^2$ is small
    - If $x^{(i)}, x^{(j)}$ are different persons, $\| f(x^{(i)}) - f(x^{(j)}) \|^2$ is large

Face Recog

Triplet Loss

- Learning Objective



eg 0.5 0.2 0.7

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

Pushes the pairs further away from each other

$$\|f(A)-f(P)\|^2 - \|f(A)-f(N)\|^2 + \alpha \leq 0$$

- Loss function

  Given 3 images A, P, N

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

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

- Choosing the triplets A, P, N

During training if A, P, N are chosen randomly, $d(A,P) + \alpha \leq d(A,N)$ is easily satisfied.

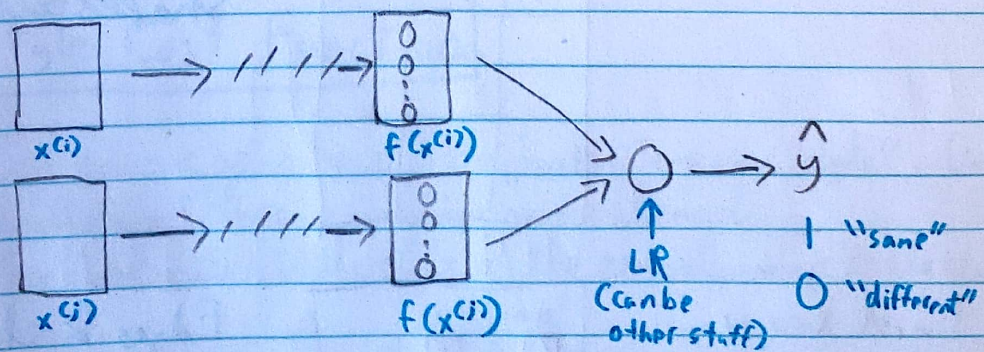
→ ∴ Choose triplets that are hard to train on

2015 FaceNet paper Schroff et al

Face Recog

Face Verification and Binary Classification

- Learning the similarity function



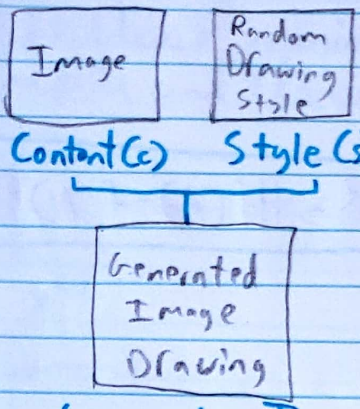
$$\hat{y} = \sigma\left(\sum_{k=1}^{128} w_k \underbrace{|f(x^{(i)})_k - f(x^{(j)})_k|}_{\text{Could be other formulas}} + b\right)$$

$$\text{eg } \frac{f(x^{(i)})_k - f(x^{(j)})_k^2}{f(x^{(i)})_k + f(x^{(j)})_k}$$

- If image is from database, encoding could be precomputed to avoid running the convNets everytime and avoid storing the raw images, saving significant computation

Neural Style Transfer

What is Neural Style Transfer? (Optimization Technique)



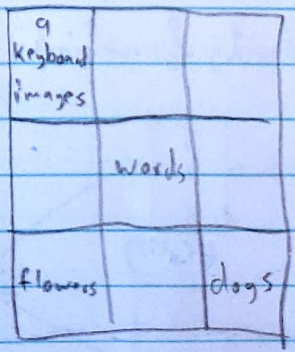
← example

Neural Style Transfer

What are deep ConvNets learning?

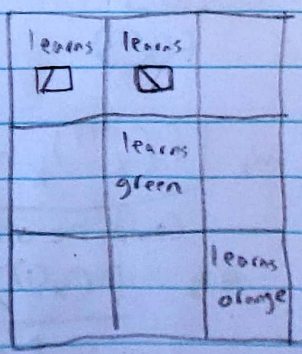
- Pick a unit in layer 1. e.g Find the nine image patches that maximize the unit's activation. Repeat for other units. Then move to other
- Usually in earlier layers, small image patches are examined and network learns about edges or shadows of colors
- In deeper layers, larger image patches are examined i.e more complex shapes and patterns

e.g In last layer



More complex shapes and patterns

First layer



Edges & shadows of colors

Neural Style Transfer

Cost Function

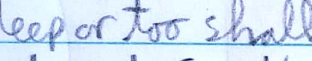
$$J(G) = \alpha J_{content}(C,G) + \beta J_{style}(S,G)$$

1) Initiate G randomly
   G: 100 x 100 x 3 (Basically random noise)

2) Use GD to minimize $J(G)$
   $G := G - \frac{\partial}{\partial G} J(G)$

Neural Style Transfer

Content Cost Function

- Say you use hidden layer l to compute content cost. l should not be too deep or too shallow. e.g.  (good choice)
- Use pre-trained ConvNet. (e.g. VGG network)
- Let $a^{[l](C)}$ and $a^{[l](G)}$ be the activation of layer l on the images.
- If $a^{[l](C)}$ and $a^{[l](G)}$ are similar, both images have similar content

$$J_{content}(C,G) = \frac{1}{2} \| a^{[l](C)} - a^{[l](G)} \|^2$$

Neural Style Transfer

Style Cost Function

- Say you are using layer l's activation to measure "style". Define style as correlation between activations across channels.
- See how similar is the style of the generated image to the style of the input image

Style matrix (gram matrix) (H, W, C)
Let $a^{[l]}_{i,j,k}$ = activation at (i,j,k). $G^{[l]}$ is $n_c^{[l]} \times n_c^{[l]}$

$G^{[l]}_{kk'}$, $k = 1, \ldots, n_c^{[l]}$

$$G^{[l](S)}_{kk'} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a^{[l](S)}_{ijk} \, a^{[l](S)}_{ijk'}$$

$$G^{[l](G)}_{kk'} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a^{[l](G)}_{ijk} \, a^{[l](G)}_{ijk'}$$

$$J^{[l]}_{style}(S,G) = \frac{1}{(2n_H^{[l]} n_W^{[l]} n_C^{[l]})^2} \sum_K \sum_{K'} \underbrace{(G^{[l](S)}_{KK'} - G^{[l](G)}_{KK'})^2}_{\| G^{[l](S)} - G^{[l](G)} \|_F^2}$$

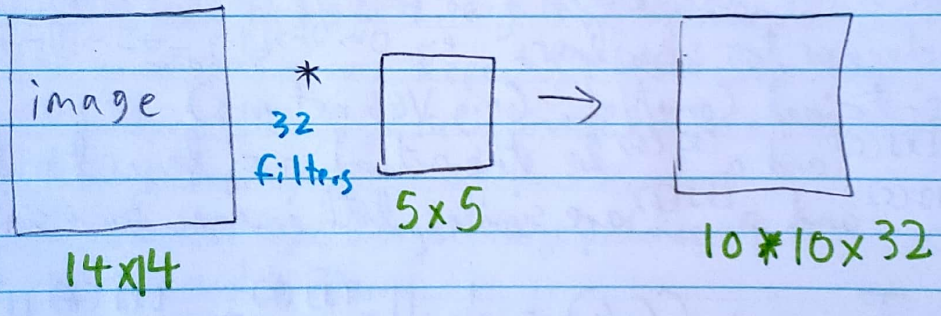
$$J_{style}(S,G) = \sum_l \lambda^{[l]} J^{[l]}_{style}(S,G)$$

↑↑

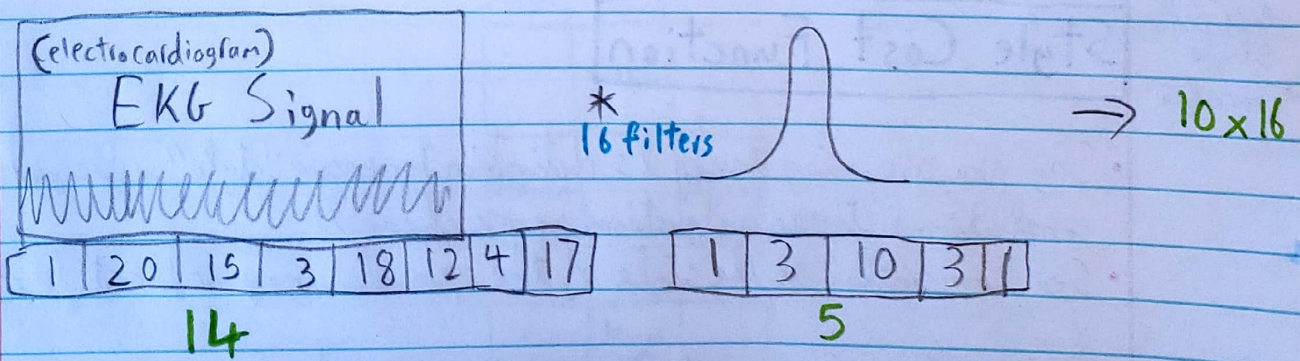
Take both low level and high level correlation into account when computing style

1D and 3D Generalizations

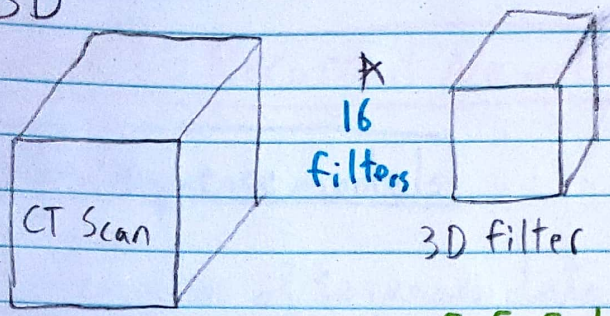
- 2D



- 1D



3D



3D volume

16 filters

3D filter

5x5x5x1

→ 10x10x10x(1)x16

14x14x14x1

↑ ↑ ↑ ↑

H W D Channels

Coursera: Deep Learning Specialization Course #5

Sequence Models

Week 1 - Recurrent Neural Networks

RNNs

Why Sequence models

- Examples of sequence data:
  - Speech recognition
  - Music generation
  - Sentiment classification (e.g. "Movie sucks" → ★☆☆☆☆)
  - DNA sequence analysis
  - Machine translation (e.g. French → English)
  - Video activity recognition
  - Name entity recognition

RNNs

Notation

- Motivating example (name entity recognition)

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

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

$x^{(i)<t>}$ $T_x^{(i)}$

$y^{(i)<t>}$ $T_y^{(i)}$

- Representing words (One-hot) Not in vocab? <UNK>

Vocabulary

a	1
aaron	2
and	367
harry	4075
potter	6830
zulu	10000

10,000

$x^{<1>}$ ← 4075 $x^{<2>}$ ← 6830 $x^{<3>}$ ← 367

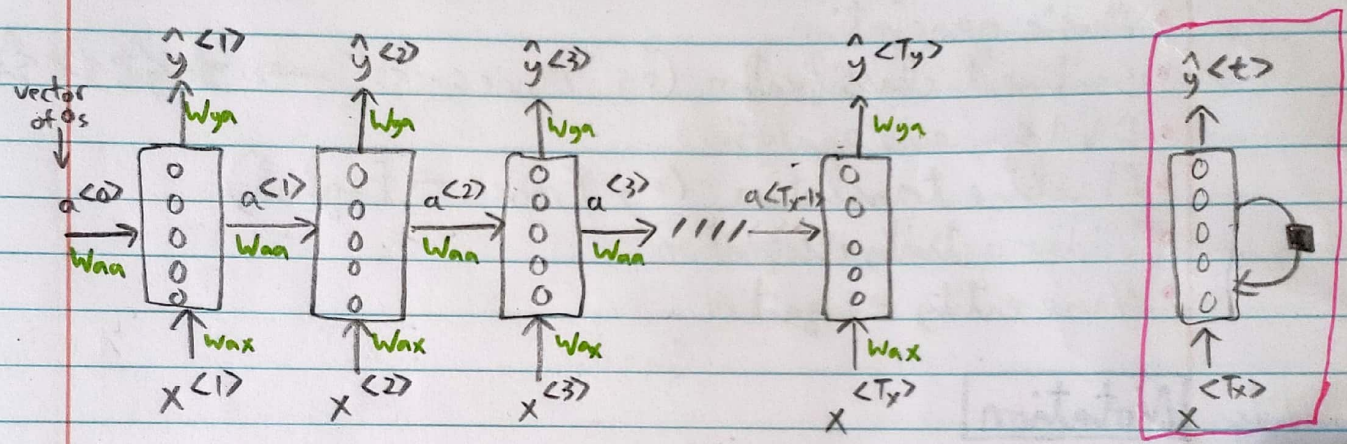
10,000

RNNs

Recurrent Neural Network Model

- Problems of a standard network:
  - Inputs, outputs can be different lengths in different examples
  - Doesn't share features learned across different positions of text

- Simple unidirectional RNN



Limitation:
→ He said "Teddy Roosevelt was a great President"
→ He said "Teddy bears are on sale!"

Need bidirectional RNN //

- Forward Propagation

$$a^{<0>} = \vec{0}$$

more generally

$$\begin{cases} a^{<1>} = g_1(W_{aa}\, a^{<0>} + W_{ax}\, x^{<1>} + b_a) & \leftarrow \tanh / \text{ReLU} \\ \hat{y}^{<1>} = g_2(W_{ya}\, a^{<1>} + b_y) & \leftarrow \text{sigmoid} \end{cases}$$

$$\begin{cases} a^{<t>} = g(W_{aa}\, a^{<t-1>} + W_{ax}\, x^{<t>} + b_a) \\ \hat{y}^{<t>} = g(W_{ya}\, a^{<t>} + b_y) \end{cases}$$

$$W_a = [W_{aa} \mid W_{ax}]$$

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

RNNs

Backpropagation through time

- Arrows go on opposite direction of forward propagation: Scan from right to left

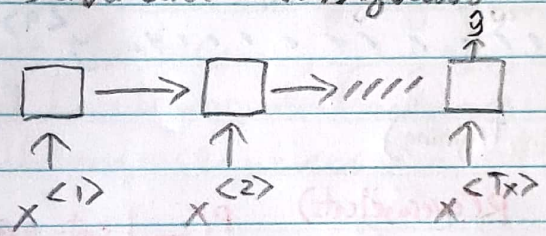
$$\mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>}) = -y^{(t)} \log \hat{y}^{<t>} - (1-y^{<t>}) \log (1-\hat{y}^{<t>})$$

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

RNNs

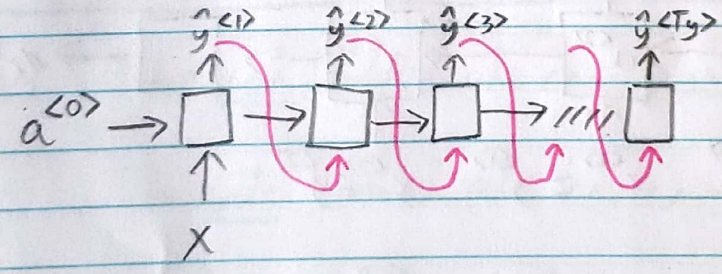
Different types of RNNs

- The RNN we have seen so far is many-to-many
- e.g. in Sentiment Classification x = text y = 0/1 or 1...5



many-to-one

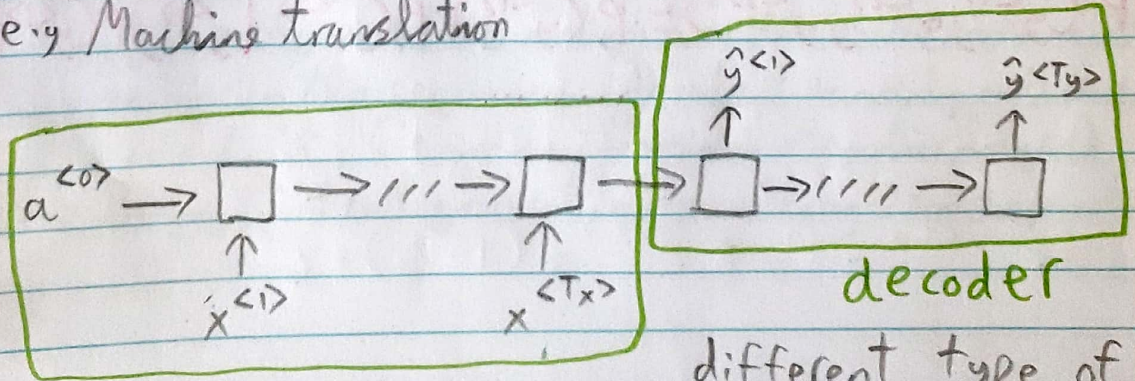
- e.g. Music generation $x \rightarrow y^{<1>} y^{<2>} \dots y^{<T_y>}$



one-to-many

x could be a note, a single integer...

- e.g. Machine translation



different type of many-to-many

RNNs

Language Model and Sequence Generation

- What is language modelling? e.g. in Speech Recognition

$P(\text{The apple and pair salad}) = 3.2 \times 10^{-13}$

$P(\text{The apple and pear salad}) = 5.7 \times 10^{-10}$

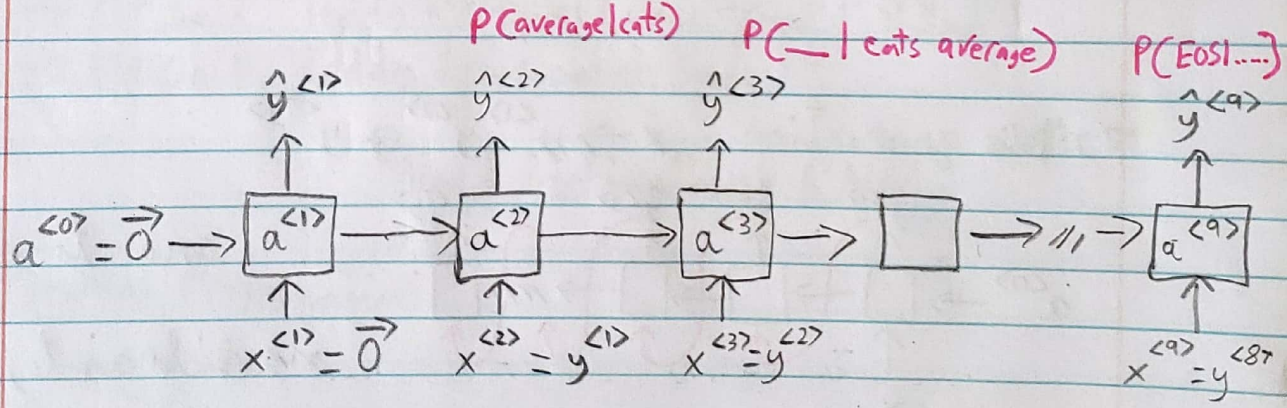
$P(y^{<1>}, y^{<2>}, \ldots y^{<T_y>})$

- Training set: large corpus of english text

e.g. Cats average 15 hours of sleep a day. <EOS> end of sentence

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

→ RNN model (During Training)

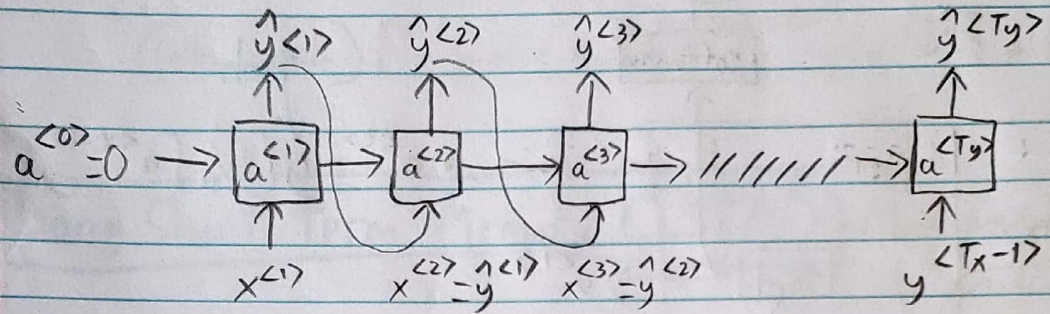


$P(y^{<1>}, y^{<2>}, y^{<3>}) = P(y^{<1>})\, P(y^{<2>} | y^{<1>})\, P(y^{<3>} | y^{<1>}, y^{<2>})$

.....

Sampling novel sequences

- Sampling a sequence from a trained RNN



- Word-level language model
  Vocab = [a, aaron, ..., zulu, <UNK>]

Character-level language model
Vocab = [a, b, c, ..., z, ␣, ., ,, ;, ..., 0, ..., 9, ..., A, ..., Z]

- Don't have to worry about unknown word token
- Much longer sequences than word-level, computationally more expensive to train

RNNs

Vanishing Gradients with RNNs

- e.g. The cat, which ate ....., was full

  The cats, which ate ....., were full

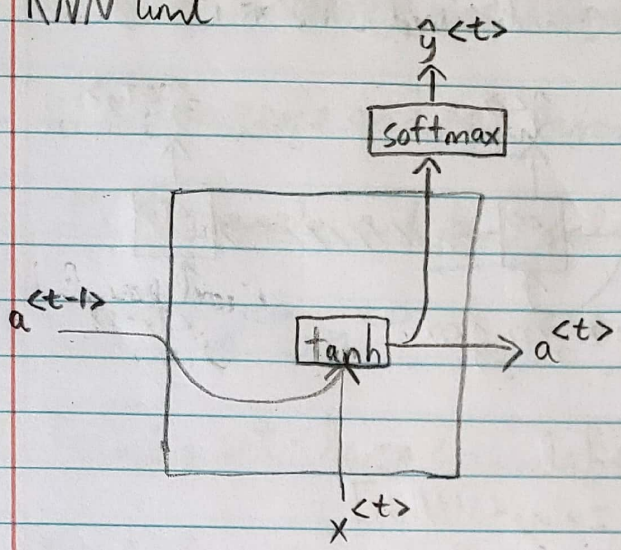
→ Hard for $\hat{y}^{<T_y>}$ to backpropagate all the way to the beginning of the sequence and modify how the RNN is doing the computations earlier in the sequence. An output is mainly influenced by values close to it. (i.e. modify the weights of earlier layers)

→ Basic RNN not good at capturing long-term dependencies

RNNs

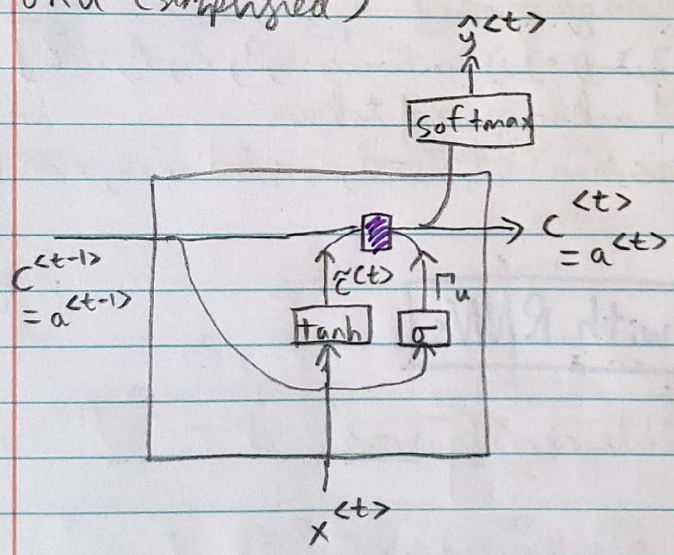
Gated Recurrent Unit (GRU)

- RNN unit



- $a^{<t>} = g(W_a[a^{<t-1>}, x^{<t>}] + b_a)$ — tanh

- GRU (simplified)

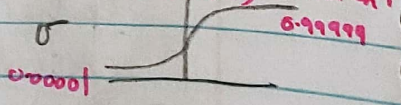


- c = memory cell $\quad c^{<t>} = a^{<t>}$

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

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

Helps with vanishing gradient



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

element-wise

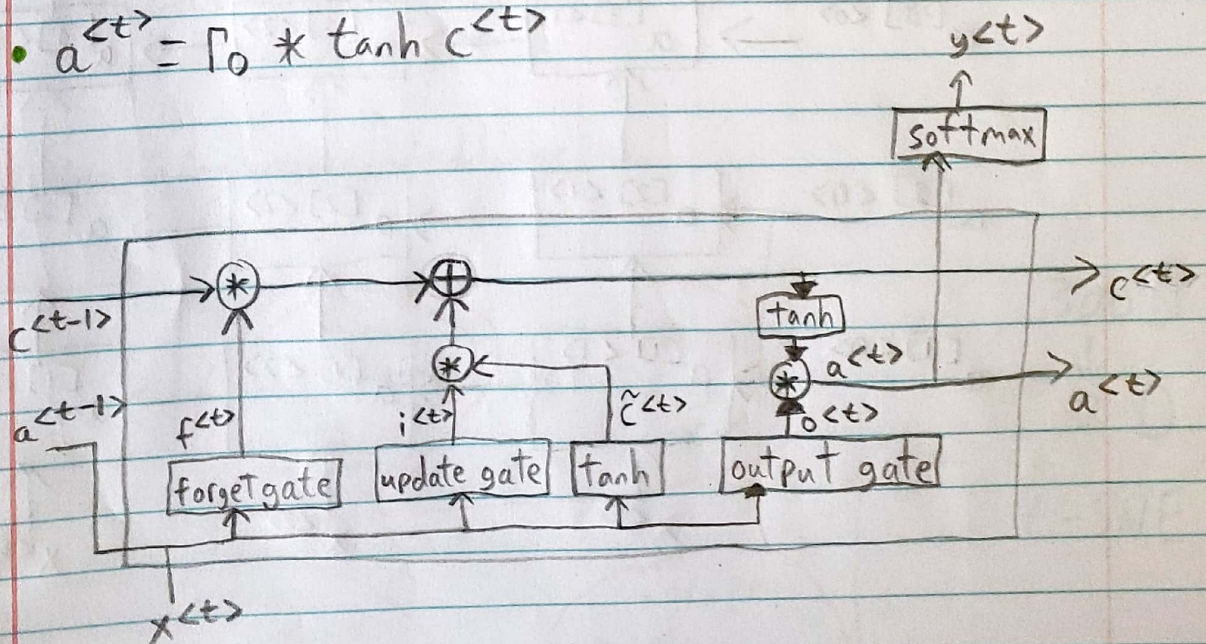
Full GRU

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

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

RNNs **Long Short Term Memory (LSTM)** More powerful & general than GRU

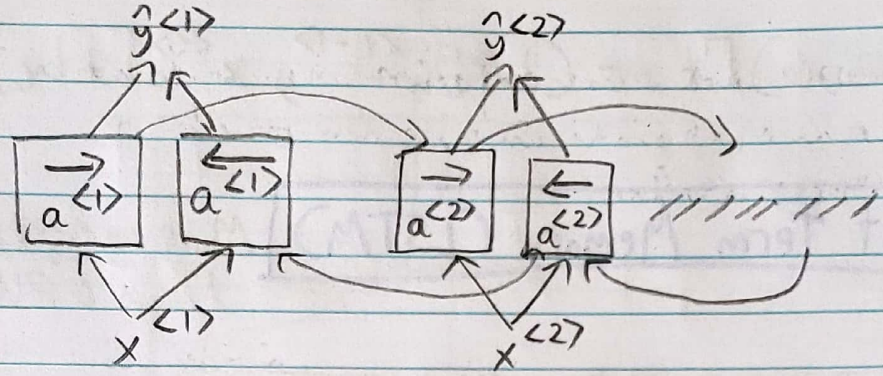
- $\tilde{c}^{<t>} = \tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c)$
- $\Gamma_u = \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u)$ update gate
- $\Gamma_f = \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f)$ forget gate
- $\Gamma_o = \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o)$ output gate
- $c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>}$
- $a^{<t>} = \Gamma_o * \tanh c^{<t>}$



RNNs

Bidirectional RNN (BRNN)

- It's an Acyclic graph



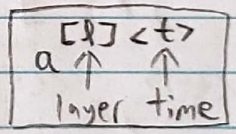
These blocks can be GRU LSTM ////

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

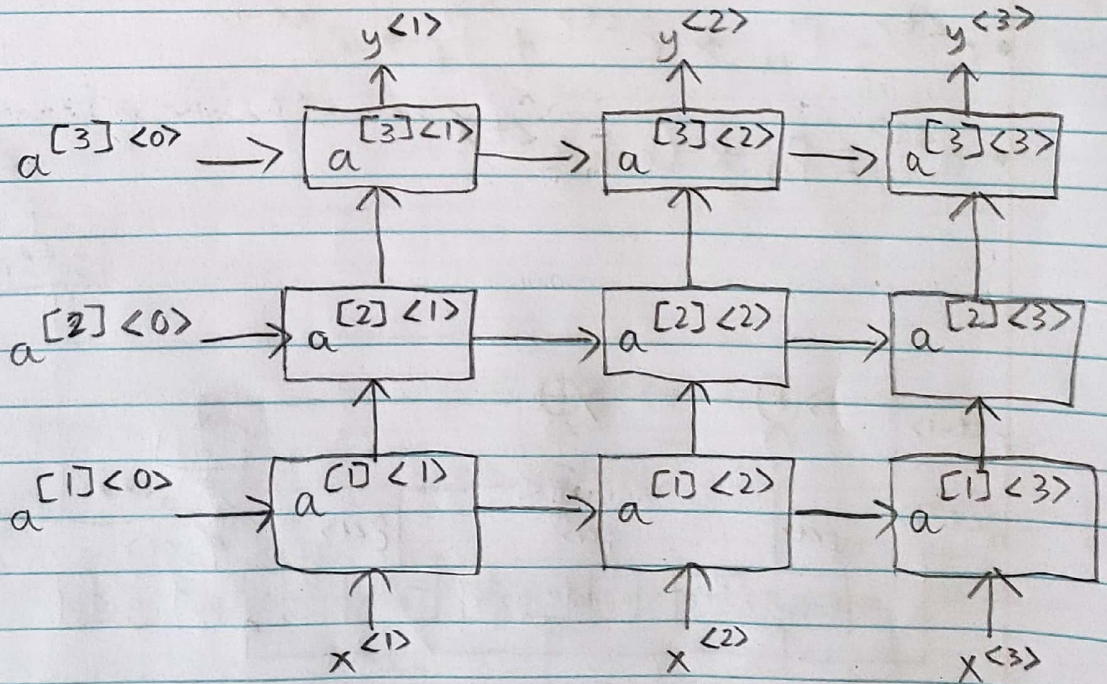
Disadvantage: Need entire sequence before being able to do a prediction

RNNs

Deep RNNs



e.g



Week 2 - Natural language processing & word embeddings

Intro to word embeddings

Word Representation

Instead of one-hot representation where words are not related to each other and are either 0/1 in the vocab, we can learn featurized representation

Vocab position	5391	9853	4914	7157	456	6257
	Man	Woman	King	Queen	Apple	Oranges
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	0.04	0.01	0.02	0.01	0.95	0.97
⋮	⋮					

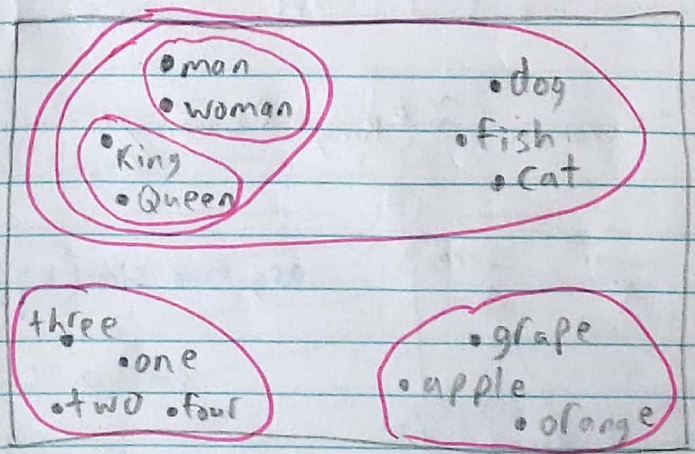
300

300 dimensional vector to represent man

e_{5391} (embedding vector)

→ Allows better generalizations across different words and better relationships

- Visualizing word embeddings



300D → 2D

t-SNE

Learn similar features for concepts that feel like they should be related

Intro to word embeddings

Using word embeddings

- Transfer learning and word embeddings

1. Learn word embeddings from large text corpus (1-100B words) (or download pre-trained embedding online)
2. Transfer embedding to new task with smaller set e.g. Name entity recognition task, 100K words
3. Optional: Continue to finetune the word embeddings with new data

Related to face encoding/embedding of siamese network of CNN
difference: lets say we have a fixed vocab of 10,000 words, we'll learn fixed encodings for each of 10K words in the vocab. In CNN, can learn encoding for any input image

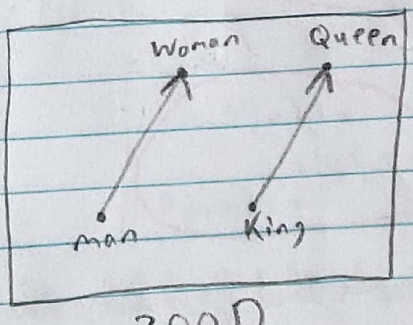
Intro to word embeddings

Properties of word embeddings

- Man → Woman as King → ?

$$e_{man} - e_{woman} \approx \begin{bmatrix} -2 \\ 0 \\ 0 \\ 0 \\ \vdots \end{bmatrix} \qquad e_{King} - e_{Queen} \approx \begin{bmatrix} -2 \\ 0 \\ 0 \\ 0 \\ \vdots \end{bmatrix}$$

$$e_{man} - e_{woman} \approx e_{King} - e_{Queen}$$

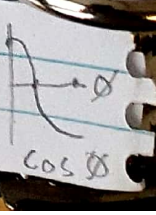


$$\arg\max \; sim(e?, e_{King} - e_{man} + e_{woman})$$

↓

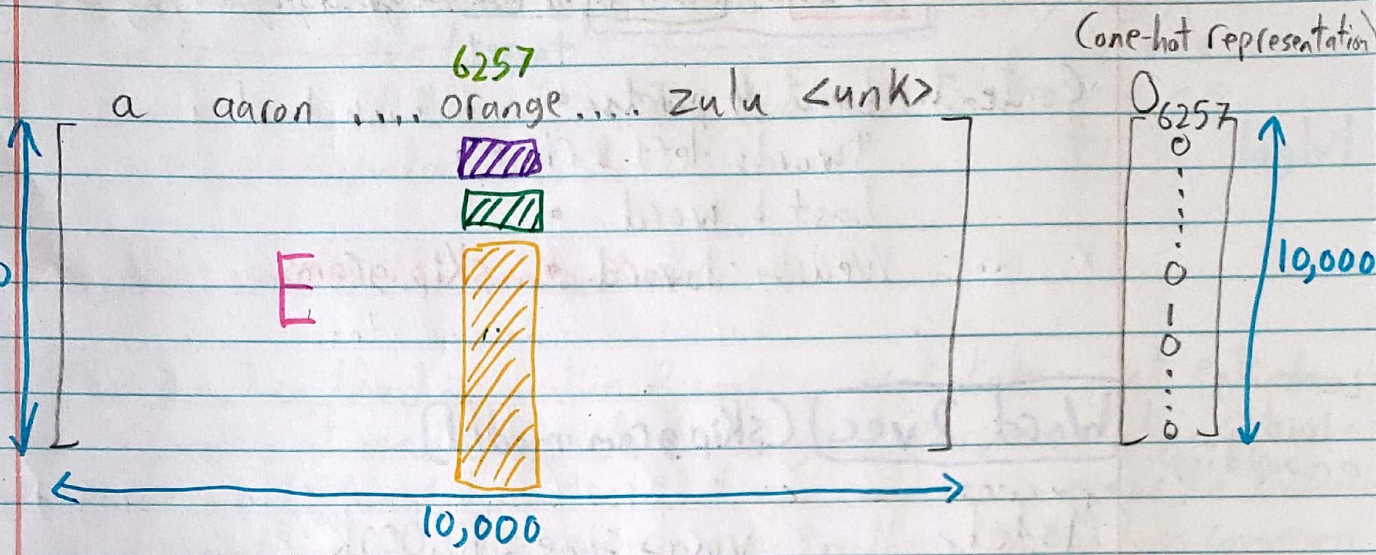
could be cosine similarity

$$sim(u,v) = \frac{u^T v}{\|u\|_2 \|v\|_2}$$



Intro to word embeddings

Embedding Matrix



$E \cdot O_{6257} = e_{6257}$ (embedding for orange)

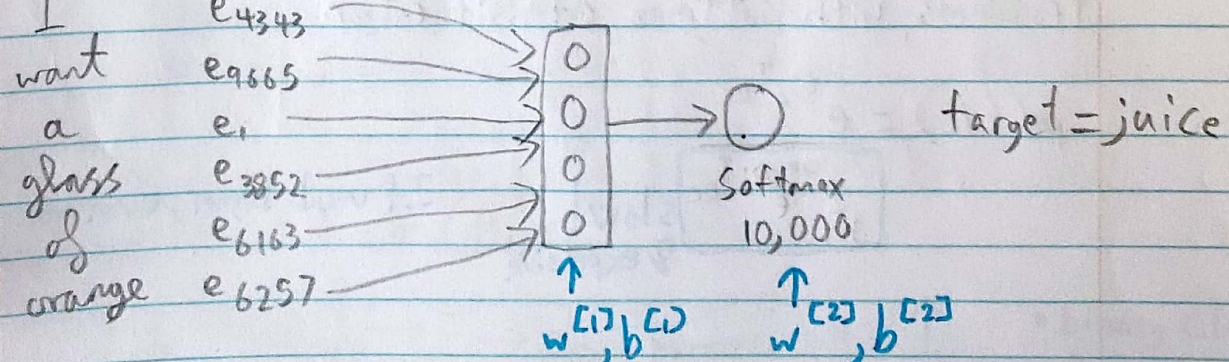
(300, 10K) · (10K, 1) = (300, 1)

→ In practice, use specialized function to look up an embedding as its way more efficient

Learning word embeddings

Learning Word Embeddings

- Yoshua Bengio : A neural probabilistic language model 2003



Learning word embeddings

- Other context/target pairs

e.g.,

I want a glass of orange juice to go along with my cereal

target

Context: Last 4 words •
4 words left & right •
Last 1 word •
Nearby 1 word • skip grams

Word 2vec (skip gram model)

- Model Vocab size = 10,000K

We would like to learn mapping context c → target t

$O_c \rightarrow E \rightarrow e_c \longrightarrow o \longrightarrow \hat{y}$ (o → ŷ: Softmax)

Softmax: • $p(t/c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10,000} e^{\theta_j^T e_c}}$

θ_t = parameter associated with output t (given by softmax)

• $\mathcal{L}(\hat{y}, y) = -\sum_{i=1}^{10,000} y_i \log \hat{y}_i$

- Problems with softmax classification

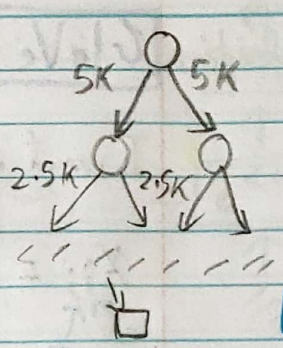
$p(t/c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10,000} e^{\theta_j^T e_c}}$

slow! & expensive

If vocab bigger, even worse!

Potential solution: Hierarchical Softmax

→ Can be developed such that common words are at the top of the tree, and less common at the bottom. They don't have to be equal on both sides.



How to sample the context c?

→ In practice, the distribution of words pc isn't taken just entirely uniformly at random for the training set purpose, but instead there are different heuristics that you could use in order to balance out something from the common words together with the less common words.

common → to, the, a, from
uncommon → orange, durian, apple

Learning word embeddings

Negative Sampling

I want a glass of orange juice to go along with my cereal

context	word	target
orange	juice	1
orange	King	0
orange	book	0
orange	the	0
orange	of	0

x → context, word
y → target (supervised)
K

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

orange 6257 → O_{6257} → E → e_{6257}

10,000

- binary classification problem
- Trained on K+1

Learning word embeddings

GloVe word vectors — Global Vectors for word Representation

I want a glass of orange juice to go along with my cereal

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

(i ↔ t, j ↔ c)

X_{ij} ⇒ Count that captures how often do words i and j appear with each other or close to each other

$$\text{minimize} \sum_{i=1}^{10{,}000} \sum_{j=1}^{10{,}000} f(X_{ij}) \left(\theta_i^T e_j + b_i + b_j' - \log X_{ij}\right)^2$$

(θ_i: t, e_j: c, symmetric)

weighting term

makes sure it doesn't give the frequent words too much weight & the less frequent too little weight

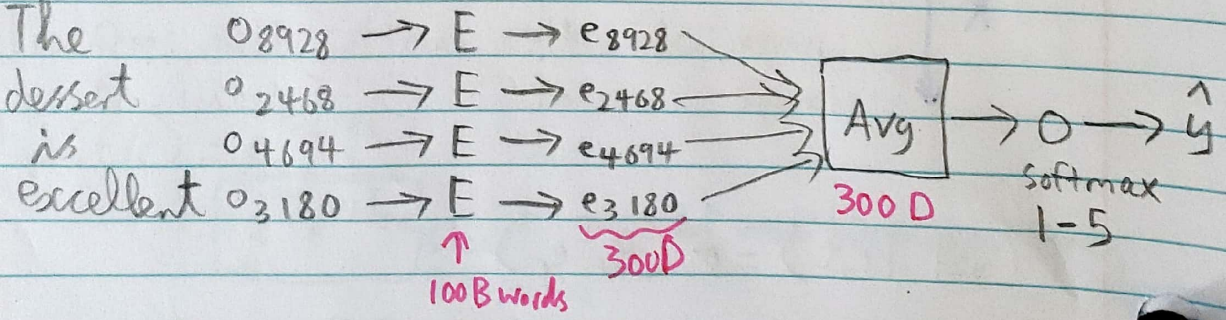
Applications using word embeddings

Sentiment Classification

Simple sentiment classification model

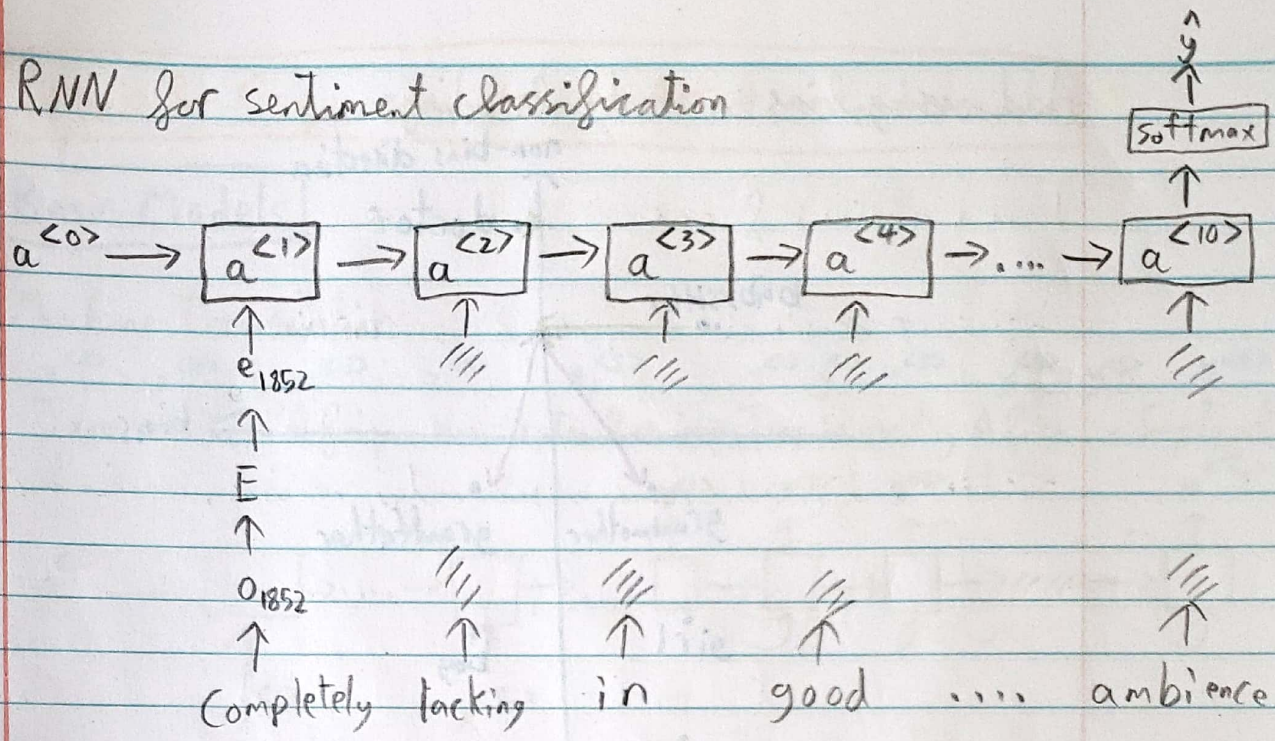
The dessert is excellent ★★★★☆

8928 2468 4694 3180



Does not take word order into consideration

RNN for sentiment classification



many-to-one

→ Generalize much better to words that were in the training set used to train the word embeddings, if word was in the 100B word corpus used to train word embeddings.

Applications using word embeddings

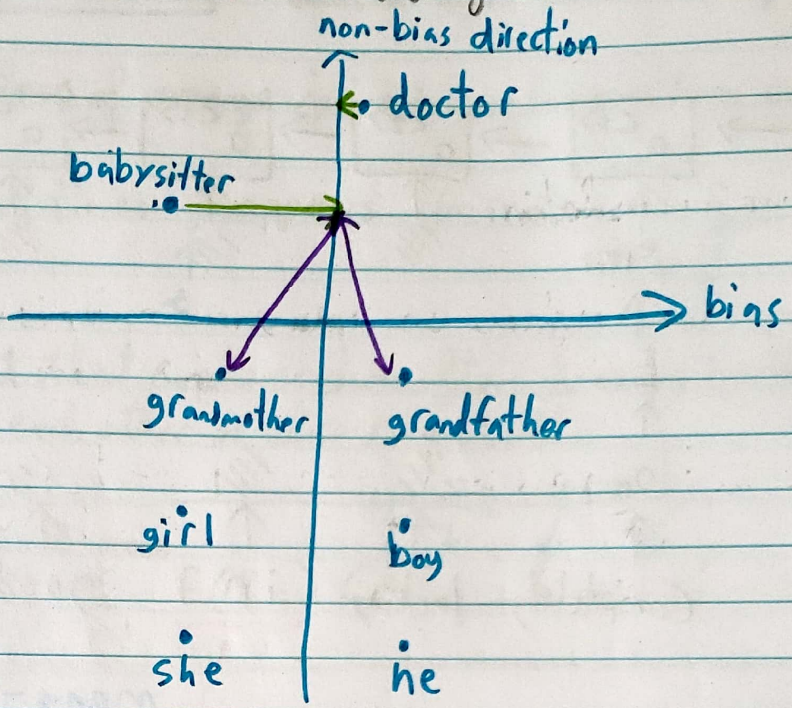
Debiasing word embeddings

The problem of bias in word embeddings

Man : Woman as King : Queen
Man : Computer Programmer as Woman : Homemaker ✗
Father : Doctor as Mother : Nurse ✗

Word embeddings can reflect gender, ethnicity, age, sexual orientation and other biases of the text used to train the model.

Addressing bias in word embeddings



1. Identify bias direction

$e_{he} - e_{she}$
$e_{male} - e_{female}$
⋮
→ average

2. Neutralize: For every word that is not definitional, project to get rid of bias

3. Equalize pairs: grandmother - grandfather
girl - boy

Make sure same distance

Week 3- Sequence models & Attention mechanism

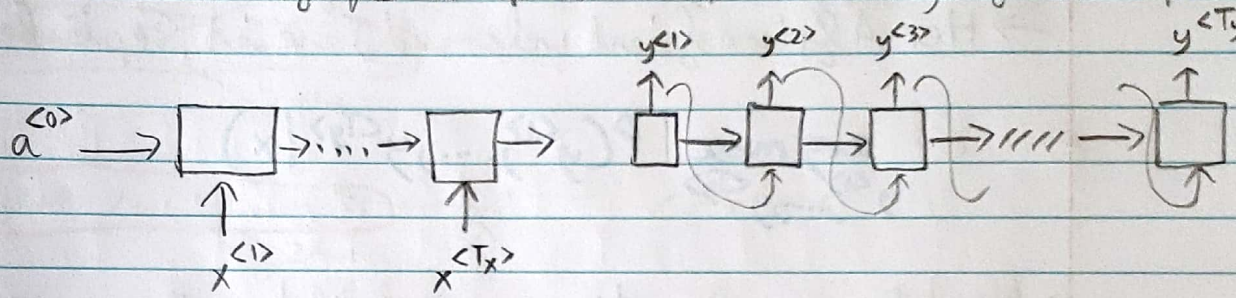
Sequence to Sequence Architectures

Basic Models

- Machine Translation

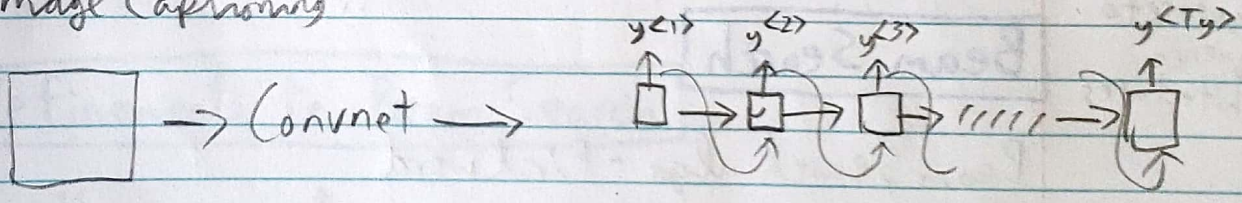
$x^{<1>}$ $x^{<2>}$ $x^{<3>}$ $x^{<4>}$ $x^{<5>}$ $y^{<1>}$ $y^{<2>}$ $y^{<3>}$ $y^{<4>}$ $y^{<5>}$ $y^{<6>}$

Jane visite l'Afrique en septembre → Jane is visiting Africa in September



encoder decoder

- Image Captioning



cat image → A cat sitting on a chair

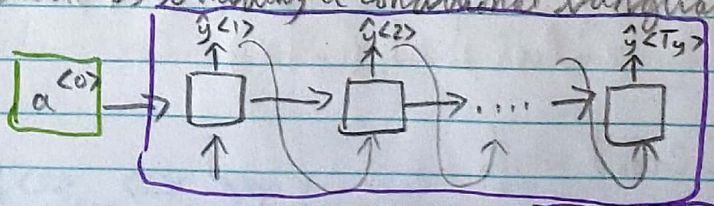
x → $y^{<1>}$ $y^{<2>}$ ... $y^{<6>}$

Sequence to sequence architectures

Picking the most likely sentence

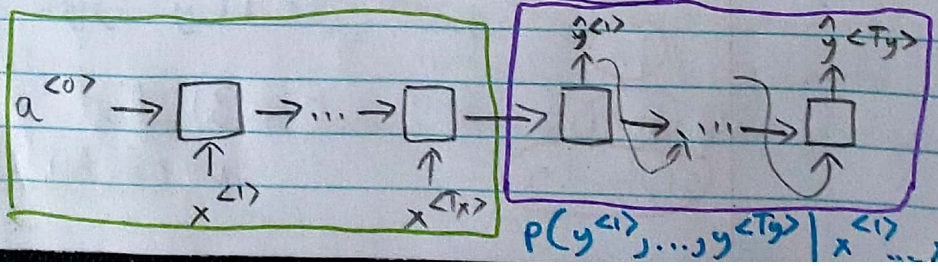
- Machine translation as building a conditional language model

Language model:



$P(y^{<1>}, \dots, y^{<Ty>})$

Machine translation:



$P(y^{<1>}, \dots, y^{<Ty>} \mid x^{<1>}, \dots, x^{<Tx>})$

- Finding the most likely translation

English French

Jane visite l'Afrique en septembre $P(y^{<1>}, \dots, y^{<Ty>} | x)$

→ Jane is visiting Africa in September
→ Jane is going to be visiting Africa in September
→ In September, Jane will visit Africa
→ Her African friend welcomed Jane in September

$$\underset{y^{<1>}, \dots, y^{<Ty>}}{\arg\max} \; P(y^{<1>}, \dots, y^{<Ty>} | x)$$

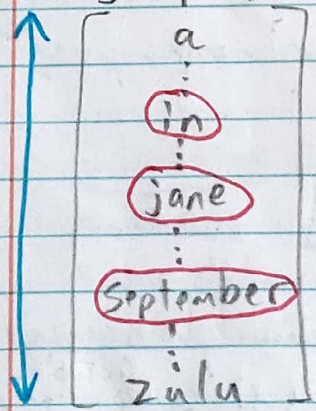
- Need to maximize entire probability, not one word at a time. ∴ Beam search & not greedy search

Sequence to sequence architectures

Beam Search

- Beam Search algo - First word

Step 1

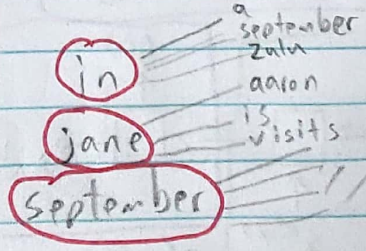


B=3 (Beam Width)
Keeps the highest 3 possibilities in memory!!

$P(y^{<1>} | x)$

(if B=1, this is equivalent to greedy search)

- Second word



$P(y^{<1>}, y^{<2>} | x) = P(y^{<1>} | x) P(y^{<2>} | x, y^{<1>})$

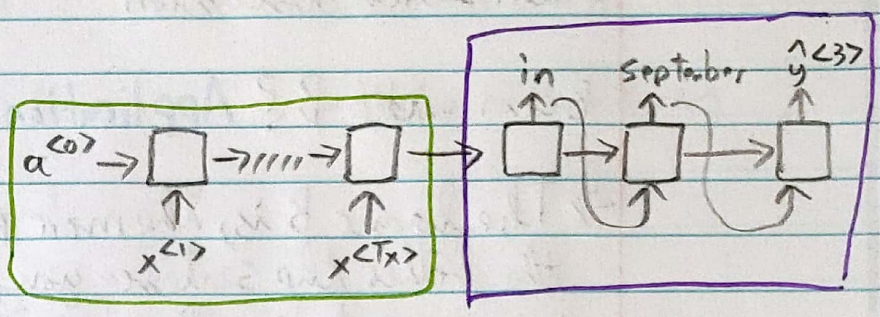
3 Highest possibilities: in september, jane is, jane visits

- Third word +

in september → a, aaron, jane, zulu

jane is → a, visiting, zulu

jane visits → a, africa, zulu



Outcome: jane visits africa in september. <EOS>

Refinements to Beam Search

- Length normalization

Right now $\arg\max P(y^{<1>}, \dots, y^{<T_y>} | x)$

$= P(y^{<1>}|x)\, P(y^{<2>}|x, y^{<1>}) \dots P(y^{<T_y>}|x, y^{<1>}, \dots, y^{<T_y-1>})$

→ Probabilities are all numbers less than 1. Often much less than 1. Multiplying them all together will result in a tiny, tiny number, which can result in numerical underflow. Meaning it's too small for the floating part representation in your computer to store accurately.

objective function: $\therefore \arg\max \sum_{t=1}^{T_y} \log P(y^{<t>} | x, y^{<1>} \dots y^{<t-1>})$ more numerically stable

The more terms, the more negative it becomes

$\frac{1}{T_y^{\alpha}}$ Normalization

Avg of the log of the prob of each word

Beam search discussion

Beam width B? Application & Domain dependent. 10, 100, 1000, 3000

→ The larger B is, the more possibilities you're considering, and the better the sentence you find. But the larger B is, the more computationally expensive the algorithm is, because you're also keeping a lot more possibilities around.

→ Unlike exact search algorithms like BFS (Breadth First Search) or DFS (Depth First Search), Beam Search runs faster but it is not guaranteed to find exact maximum for $\arg\max P(y|x)$

sequence to sequence architectures

Error Analysis in Beam Search

Example

Jane visits l'Afrique en septembre

- Human: Jane visits Africa in September (y^*)
- Algorithm: Jane visited Africa last September ($\hat{y}$)

Which one is at fault? → RNN?
→ Beam search?

① $P(y^*|x) > P(\hat{y}|x)$ Beam search at fault

② $P(y^*|x) \leq P(\hat{y}|x)$ RNN model is at fault

Then figure out what fraction of errors are "due to" beam search vs. RNN model

Sequence sequence architectures

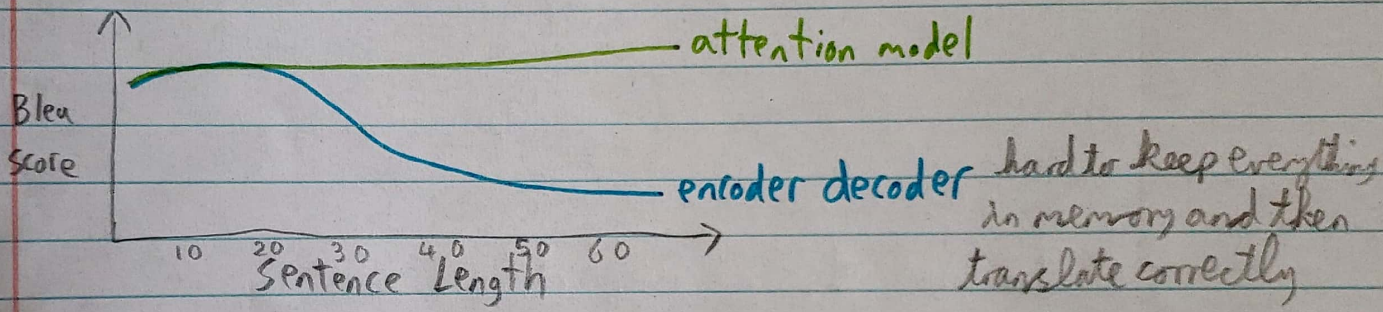
Bleu Score Bilingual Evaluation Understudy

- Measuring accuracy when there are a couple of great answers
- Video too noisy to hear!

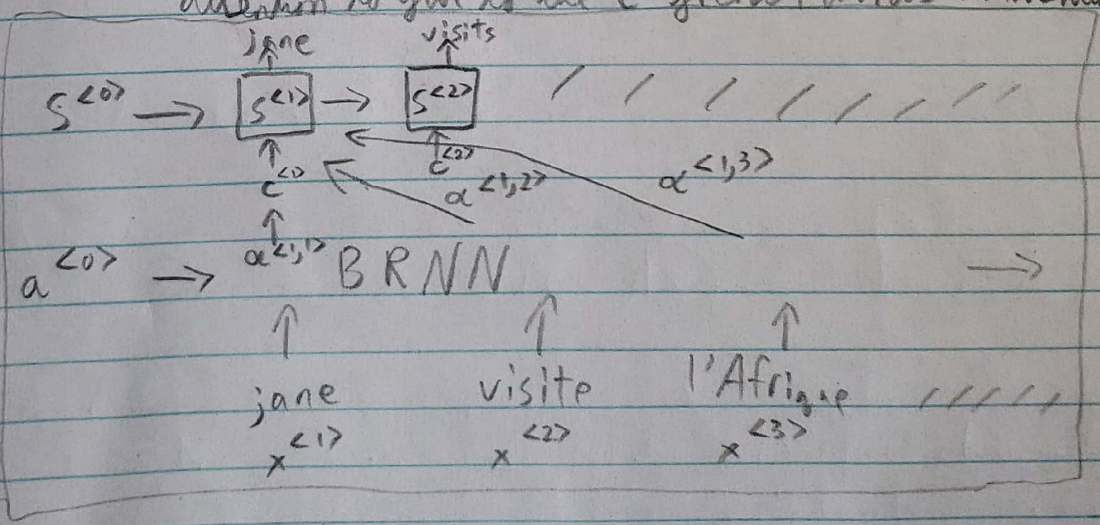
Sequence to sequence architectures

Attention Model Intuition

- The problem of long sequences



- $\alpha^{<t,t'>}$ when you're trying to generate t english word, how much attention to give to the t' french words. Attention weights!



$$c^{<1>} = \sum_{t'} \alpha^{<1,t'>} a^{<t'>}$$

$$c^{<2>} = \sum_{t'} \alpha^{<2,t'>} a^{<t'>}$$

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

Quadratic cost time!