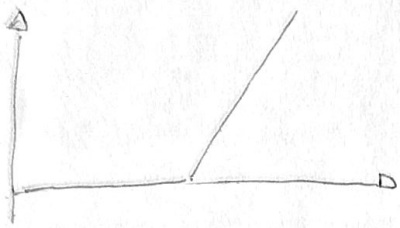
Deep learning $\xrightarrow{is}$ training Neural Networks

↓

Neural Network ⇒ input → ○ → output

" node / neuron

Function shape (common)

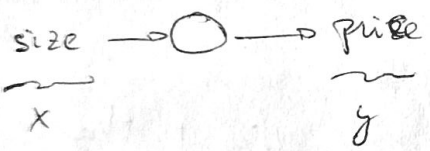


usually called ReLU [Rectified Linear Unit]

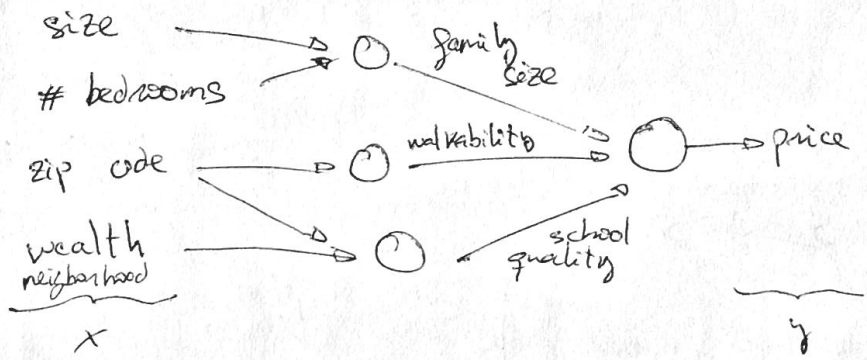
Big neural networks are composed by many of single neurons stacked together

Neural Network about Housing Price Prediction

Single neuron



Plural neurons



Every input layer feature is interconnected with every hidden layer feature

---

Supervised learning

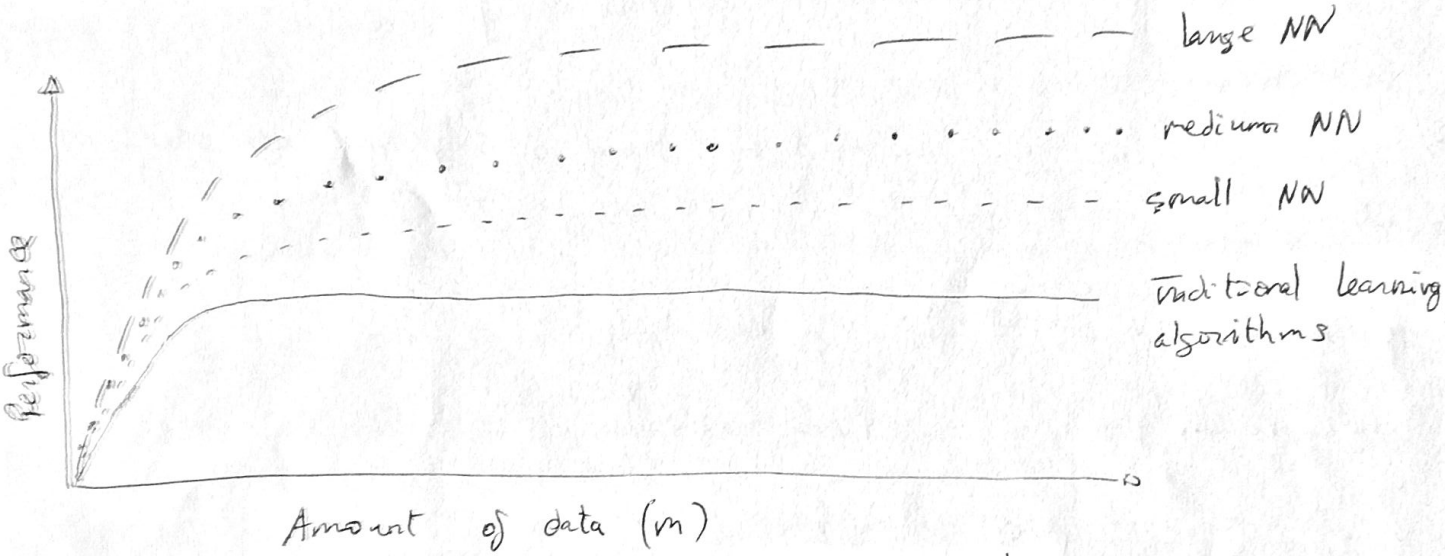
Input (x)	Output (y)	Application	
Home features →	Price	Real Estate	SNN
Ad, user info →	Click on ad? (0/1)	Online advertising	
Image →	Object (1, ..., 1000)	Photo tagging	CNN
Audio →	Text transcript	Speech recognition	RNN
English →	Chinese	Machine translation	
Image, Radar info →	Position of other cars	Autonomous driving	Hybrid

SNN (Standard Neural Network)
CNN (Convolutional " ")
RNN (Recurrent " ") ← Sequence data

} Architectures

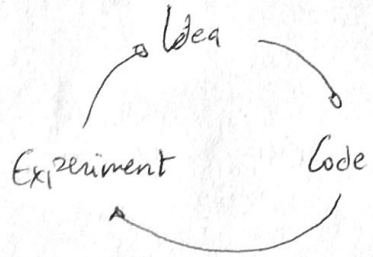
Data { Structured → Base data → Structured array
Unstructured → Audio, Image, Text

Idea: Crear app que imite el tono de voz del usuario



ReLU functions are ~~more~~ way more computationally more efficient than sigmoid ~~Sigmoid~~ ones

- When an engineer works on a new problem, they cannot use insight from previous problems to train a good model on the first try. Finding the characteristics of a model is key to have a good performance. It's required multiple iterations to build a good model.



- Being able to try out ideas quickly allows deep learning engineers to iterate more quickly
- Faster computation can help speed up how long a team takes to iterate to a good idea
- Training on a big dataset is usually slower than on a small one

Week 2

Image 64 pixels (4:4) $\xrightarrow{RGB}$ $\begin{cases} \text{Red} & 64 \text{ elements in a squared matrix} \\ \text{Green} & \text{" " " " " "} \\ \text{Blue} & \text{" " " "} \end{cases}$

$\hookrightarrow$ total elements $64 \times 64 \times 3 = 12\,288 = n_x$

		255	134	93	22
	255	134	202	22	2
255	231	42	22	4	30
123	94	83	2	192	124
34	44	187	42	34	142
34	76	232	124	94	
67	83	194	202		

B

G

R

$$X = \begin{bmatrix} 255 \\ 231 \\ \vdots \\ 255 \\ 134 \\ \vdots \\ 255 \\ 134 \\ \vdots \end{bmatrix}$$

image $\longrightarrow$ 1 (cat) vs 0 (no cat)

$x \longrightarrow y$

$(x, y) \quad x \in \mathbb{R}^{n_x}, \; y \in \{0, 1\}$

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

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

[Python : X shape = (n_x, m)]

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

[Python : Y shape = $(1, m)$]

Given input x $\xrightarrow{\text{algorithm that predicts}}$ estimation of y : $\hat{y} = P(y=1/x)$
being equal to 1

$\hookrightarrow \in [0,1]$

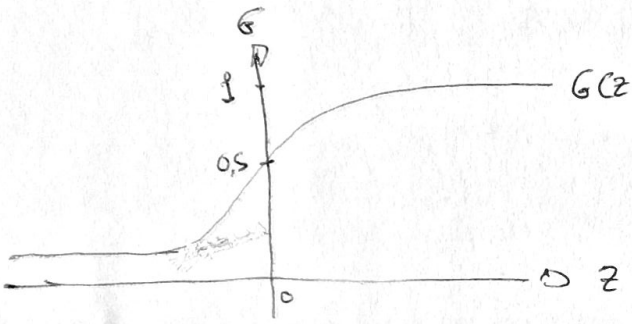
$x \in \mathbb{R}^{n_x}$ (vector with n_x dimensions) $\quad P(y/X) = \hat{y}^{y}(1-\hat{y})^{(1-y)}$

Parameters of logistic regression : $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

Output $\hat{y} = w^T x + b$ (linear regression) $\Rightarrow$ not good for binary classification

$\notin [0,1]$

? solution $\rightarrow$ $\hat{y} = \underbrace{\sigma}_{\text{sigmoid function}}(w^T x + b) = \sigma(z) = (1+e^{-z})^{-1}$



$$\sigma(z) = \frac{1}{1+e^{-z}} \qquad \begin{cases} \lim_{z\to\infty} \sigma(z) = 1 \\ \lim_{z\to-\infty} \sigma(z) = 0 \end{cases}$$

Cost function to train parameters of $\underbrace{\text{logistic regression}}_{\text{very small neural network}}$

$\hat{y}^{(i)} = \sigma(w^T x^{(i)} + b)$, where $\sigma(z^{(i)}) = \dfrac{1}{1+e^{-z^{(i)}}}$ $\qquad z^{(i)} = w^T x^{(i)} + b$

Given $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$, want $\hat{y}^{(i)} \approx y^{(i)}$ $\qquad i$-th example

Loss function error $\rightarrow$ measures how well is the algorithm doing

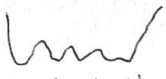
$\hookrightarrow$ ~~$L(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$~~ $\Rightarrow$ squared error $\rightarrow$ not adequated because optimization problems are not convex $\rightarrow$ multiple local optima

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

minimize Loss function

if $y=1$: $L(\hat{y}, 1) = -\log \hat{y}$ $\rightarrow$ want $\log \hat{y}$ large

want $\hat{y}$ large



$\hookrightarrow$ it may not find the global optima

if $y=0$: $L(\hat{y}, 0) = -\log(1-\hat{y})$ $\rightarrow$ want $\log(1-\hat{y})$ large

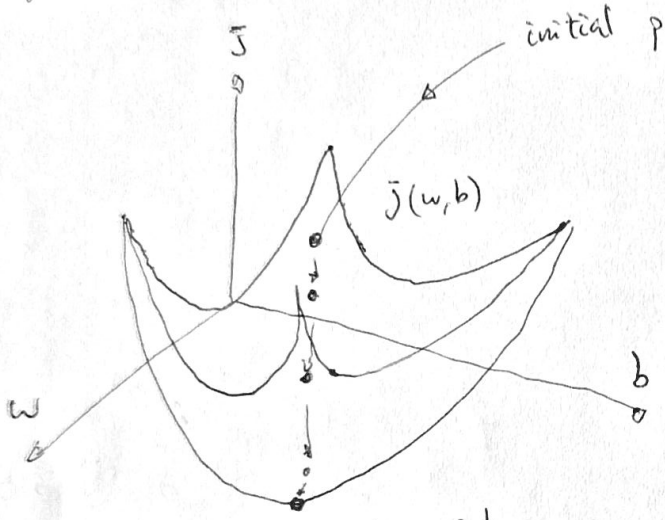
want $\hat{y}$ small

Cost function → how well is the entire training set doing

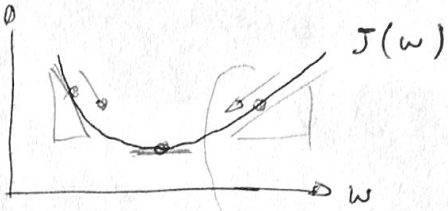
↳ $J(w,b) = \overset{\text{average}}{\frac{1}{m}} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log(1-\hat{y}^{(i)}) \right]$

↳ aim → minimize J looking for the appropriate w and b parameters

Gradient descent algorithm → want to find (w, b) that min J



↳ going downhill as fast as possible



Repeat {

learning rate alpha

$\underbrace{w :=}_{\text{update value}} w - \alpha \boxed{\frac{\partial J(w)}{\partial w}}$

slope of function

$J(w,b)$

$\overbrace{w := w - \alpha \frac{\partial J(w,b)}{\partial w}}^{dw \text{ in code}}$

$\underbrace{b := b - \alpha \frac{\partial J(w,b)}{\partial b}}_{db \text{ in code}}$

Computational graph

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

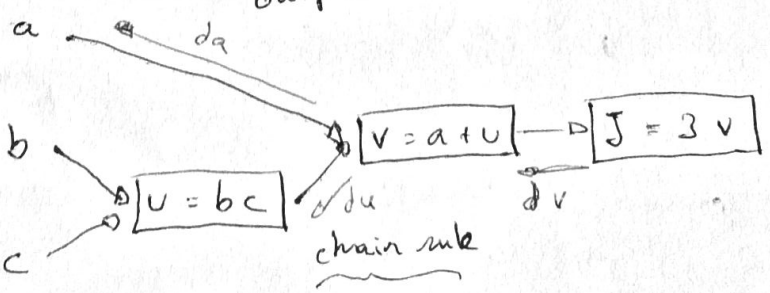
J

$u = bc$

$v = a + u$

$J = 3v$

One step of backward propagation on a computation graph yields derivative of final output variable



chain rule

$\frac{dJ}{dv} = 3 \qquad \frac{dJ}{da} = \frac{dJ}{dv}\frac{dv}{da} = 3$

2 features

x_1
w_1
x_2
w_2
b

$$\boxed{z = w_1 x_1 + w_2 x_2 + b} \rightarrow \boxed{\hat{y} = a = \sigma(z)} \rightarrow \boxed{L(a, y)}$$

$$a - y = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z} = \frac{\partial L(a,y)}{\partial z} = \partial z$$

$$\hookrightarrow a(1-a)$$

$$\text{"}da\text{"} = \frac{\partial L(a,y)}{\partial a} = -\frac{y}{a} + \frac{1-y}{1-a}$$

$$\frac{\partial L}{\partial w_1} = \text{"}dw_1\text{"} = x_1\, dz \qquad dw_2 = x_2\, dz \qquad db = dz$$

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

$$J(w, b) = \frac{1}{m}\sum_{i=1}^{m} L(a^{(i)}, y^{(i)})$$

$(x^{(i)}, y^{(i)})$

$dw_1^{(i)}, dw_2^{(i)}, db^{(i)}$

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

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

initialize: $J = 0,\ dw_1 = 0,\ dw_2 = 0,\ db = 0$

loop [For $i = 1$ to m

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

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

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

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

loops $\begin{cases} dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)} \\ dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)} \\ db \mathrel{+}= dz^{(i)} \end{cases}$

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

the more loops, the more computational cost

↳ vectorization → avoid loops

↳ more efficient neural networks

Vectorization → getting rid of explicit for loops in the code

$$z = w^T x + b \qquad w = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix} \qquad x = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix} \qquad w, x \in \mathbb{R}^{n_x}$$

Non-vectorial:

```
z = 0
for i in range (n_x):
    z += w[i] * x[i]
z += b
```

Vectorial → tremendously more efficient

$$z = \underbrace{np.dot(w, x)}_{w^T x} + b$$

$$u = A v$$

$$v_i = \sum_j A_{ij} v_j$$

```
v = np.zeros(
for i ...
    for j ...
        v[i] += A[i][j] * v[j]
```

NumPy → Python libraries for math

```
v = np.dot(A, v)
```

↳ function

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

```
u = np.zeros((n,1))
for i in range(n):
    v[i] = math.exp(v[i])
```

```
import numpy as np
v = np.exp(v)
    np.log(v)
    np.abs(v)
    np.maximum(v, 0)
np.v**2
np.1/v
```

$J = 0,\ dw_1 = 0,\ dw_2 = 0,\ db = 0$

for $i = 1$ to m:

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

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

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

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

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

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

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

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

$J = 0,\ db = 0$

for $i = 1$ to m:

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

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

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

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

$dw = np.zeros((n_x, 1))$

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

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

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

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

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

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

$$\underbrace{Z = [z^{(1)}\ z^{(2)} \cdots z^{(m)}] = w^T X + [b\ b \cdots b] = [(w^T x^{(1)} + b)\ (w^T x^{(2)} + b) \cdots \cdots (w^T x^{(m)} + b)]}_{1 \times m} \longrightarrow Z = np.dot(\underbrace{w.T}_{\text{transpose}}, X) + \underbrace{b}_{(1,1) \in \mathbb{R}^1}$$

$w \in \mathbb{R}^{(n_x \times 1)}$

$$\underbrace{Z}_{1\times m} = \underbrace{w^T}_{1\times n_x} \underbrace{X}_{n_x \times m} + \underbrace{b}_{1\times 1}$$

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

$$dZ = \underbrace{[dz^{(1)} \; dz^{(2)} \; \cdots \; dz^{(m)}]}_{1 \times m}$$

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

$$dZ = A - Y = [a^{(1)} - y^{(1)} \cdots a^{(m)} - y^{(m)}]$$

$$dw = 0 \qquad db = 0$$

$$\begin{bmatrix} dw^{(1)} += x^{(1)} dz^{(1)} & db += dz^{(1)} \\ dw^{(2)} += x^{(2)} dz^{(2)} & db += dz^{(2)} \\ \vdots & \vdots \end{bmatrix}$$

$$dw /= m \qquad db /= m$$

$$db = \frac{1}{m} \sum_{i=1}^{m} dz^{(i)}$$

$$= \frac{1}{m} \text{ np.sum}(dZ)$$

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

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

$$= \frac{1}{m} \underbrace{[x^{(1)} dz^{(1)} + \cdots + x^{(m)} dz^{(m)}]}_{n_x \times 1}$$

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

for $i = 1$ to m:

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

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

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

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

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

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

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

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

$$db /= m$$

$$Z = w^T X + b =$$

$$= \text{np.dot}(w.T, X) + b$$

$$A = \sigma(Z)$$

$$dZ = A - Y$$

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

$$db = \frac{1}{m} \text{ np.sum}(dZ)$$

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

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

Broadcasting → technique that makes Python code run faster

Calories from Carbs, Proteins, Fats in 100g of different foods

axis 1 → p, axis 0 ↓

$$\begin{array}{c} \\ \text{Carb} \\ \text{Proteins} \\ \text{Fat} \end{array} \begin{array}{c} \begin{array}{cccc} \text{Apples} & \text{Beef} & \text{Eggs} & \text{Potatoes} \end{array} \\ \begin{bmatrix} 56.0 & 0.0 & 4.4 & 68.0 \\ 1.2 & 104.0 & 52.0 & 8.0 \\ 1.8 & 135.0 & 99.0 & 0.9 \end{bmatrix} \end{array} = A \in \mathbb{R}^{3\times 4}$$

% calories of calories without using explicit for-loops

cal = A.sum(axis = 0) → sums columns

percentage = 100 * A /(cal. $\underbrace{\text{reshape}(1,4)}$)

→ raw and columns redundant because cal is already 1x4

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + \underbrace{\begin{bmatrix} 100 \\ 100 \\ 100 \\ 100 \end{bmatrix} 100}_{\text{autoexpands}} = \begin{bmatrix} 101 \\ 102 \\ 103 \\ 104 \end{bmatrix} \qquad \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 & 200 & 300 \\ 100 & 200 & 300 \end{bmatrix} = \begin{bmatrix} 101 & 202 & 303 \\ 104 & 205 & 306 \end{bmatrix}$$

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

To avoid bugs in the code

instead of a = np.random.randn(S) → rank 1 array
(S,)

→ a = np.random.randn(S, 1) → column vector

→ a = np.random.randn(1, S) → row vector

$\hat{y} = P(y=1/X)$

→ $\max P(y/X) = \hat{y}^{y}(1-\hat{y})^{(1-y)}$ → $\log P(y/X) = \log \hat{y}^{y}(1-\hat{y})^{(1-y)} =$

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

$\log P(\text{labels in training set}) = \log \prod_{i=1}^{m} P(y^{(i)} / x^{(i)})$

$$\log P(\text{" "}) = \sum_{i=1}^{m} \underbrace{\log P(y^{(i)} / x^{(i)})}_{-L(\hat{y}^{(i)}, y^{(i)})} = -\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

maximum likelyhood estimation

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

$\Downarrow$ minimise

A neuron computes a linear function ($z = Wx + b$) followed by an activation function

Element-wise multiplication is an operation ~~among~~ between element by element matrixes.

$$C = A * B = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} \\ a_{21}b_{21} & a_{22}b_{22} \end{bmatrix}$$

$$c = np.dot(A, B) = \cancel{\begin{bmatrix} a_{11}b & a_{12}b \\ a_{21}b & a_{22}b \end{bmatrix}}$$

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \qquad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \qquad \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{12}b_{12} + a_{22}b_{22} \\ a_{21}b_{11} + a_{22}b_{22} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

Logistic Loss $\quad L^{(i)}(\hat{y}^{(i)}, y^{(i)}) = -(y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}))$

```
# a.shape = (3,4)
# b.shape = (4,1)
for i in range(3):
    for j in range(4):
        c[i][j] = a[i][j] + b[j]
```

$c = a + \underbrace{b.T}_{\text{broadcast all the rows}}$

1 - Packages

```
import numpy as np
import matplotlib.pyplot as plt
import h5py
import scipy
from PIL import Image
from scipy import ndimage
from lr_utils import load_dataset
%matplotlib inline
```

- scientific computing in Python
- plot graphics
- interaction with dataset stored in H5 file
- to test model with your own pic

2 - Overview of the problem set

dataset in data.h5 contains
- a training set of m_trains images labeled as cat (y=1) or not (y=0)
- a test set of m_test images labeled as cat or not
- each pic is of shape (num_px, num_px, 3) where 3 is for the channels RGB

loading the data (cat / non-cat)

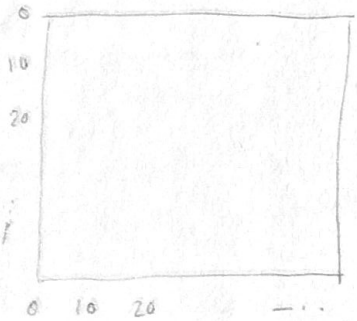
train_set_x_orig, train_set_y, test_set_x_orig, test_set_y, classes = load_dataset()

train/test_set_x_orig $\xrightarrow{\text{process}}$ train/test_set_x : array representing an image

example of a picture

```
index = 89
plt.imshow(train_set_x_orig[index])
print("y= " + str(train_set_y[:, index]) + ", it's a '" + classes[np.squeeze(
    (train_set_y[:, index])].decode("utf-8") + "' picture.")
```

y = [0], it's a 'non-cat' picture.



Many bugs in deep learning come from having matrix / vector dimensions that don't fit. train_set_x_orig is a numpy-array of shape (m_train, num_px, num_px, 3)

```
# Finding out values
m_train = train_set_x_orig.shape[0]
m_test = test_set_x_orig.shape[0]
num_px = train_set_x_orig.shape[1]
print("Number of training examples: m_train = " + str(m_train))
print("Number of testing examples: m_test = " + str(m_test))
print("Height/Width of each image: num_px = ") + str(num_px))
print("Each image is of size: (" + str(num_px) + "," + str(num_px) + ",3)")
print("train_set_x shape: " + str(train_set_x_orig.shape))
print("train_set_y shape: " + str(train_set_y.shape))
print("test_set_x shape: " + str(test_set_x_orig.shape))
print("test_set_y shape: " + str(test_set_y.shape))
```

```
Number of training examples: m_train = 209
Number of testing examples: m_test = 50
Height/Width of each image: num_px = 64
Each image is of size: (64,64,3)
train_set_x shape: (209,64,64,3)
train_set_y shape: (1,209)
test_set_x shape: (50,64,64,3)
test_set_y shape: (1,50)
```

For convenience, reshape images of shape (num_px, num_px, 3) in a flattened image in numpy-array (num_px * num_px * 3, 1). There should be m_train and m_test columns

Reshape the training and test examples

```
train_set_x_flatten = train_set_x_orig.reshape(train_set_x_orig[0],
                                                -1].T

test_set_x_flatten = test_set_x_orig.reshape(test_set_x_orig[0],
                                              -1].T

print("train_set_x_flatten shape:" + str(train_set_x_flatten.shape))
print("train_set_y shape:" + str(train_set_y.shape))
print("test_set_x_flatten shape:" + str(test_set_x_flatten.shape))
print("test_set_y shape:" + str(test_set_y.shape))
```

train_set_x_flatten shape: (12288, 209)

train_set_y shape: (1, 209)

test_set_x_flatten shape: (12288, 50)

test_set_y shape: (1, 50)

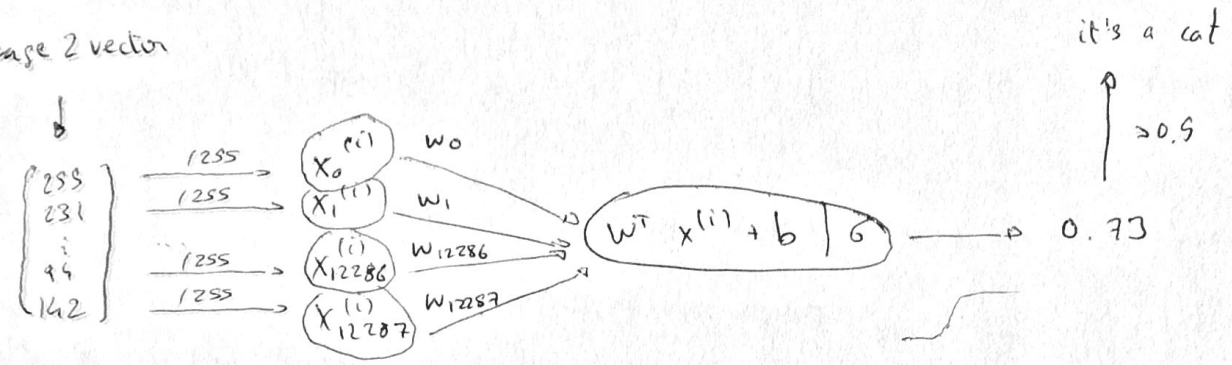
[Every pixel in RGB is a vector of tree numbers ranging from 0 to 255

Standarization of data sets

```
train_set_x = train_set_x_flatten / 255.
test_set_x = test_set_x_flatten / 255.
```

3 - General Architecture of the learning algorithm

Building a Logistic Regression using a Neural Network mindset



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

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

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

$$L(a^{(i)}, y^{(i)}) = -y^{(i)} \log(a^{(i)}) - (1-y^{(i)}) \log(1-a^{(i)})$$

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

4 - Building the parts of our algorithm

4.1 - Helper functions

```
# Graded function : sigmoid
def sigmoid (z):
    s = 1/(1 + np.exp(-z))
    return s
```

4.2 - Initializing parameters

```
# Graded function : initialize_with_zeros
def initialize_with_zeros (dim)
w = np.zeros((dim, 1))
b = 0
return w, b
```

[dim = size of the vector w, number of parameters in this case

4.3 - Forward and Backward propagation (for learning the parameters)

$$A = \sigma(w^T X + b) = (a^{(0)}, a^{(1)}, \dots, a^{(m)})$$

$$J = -\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \log(a^{(i)}) + (1-y^{(i)}) \log(1-a^{(i)})$$

$$\frac{\partial J}{\partial w} = \frac{1}{m} X (A-Y)^T \qquad \frac{\partial J}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} (a^{(i)} - y^{(i)})$$

```
# Graded function : propagate
def propagate (w, b, X, Y)                    w: weish    b: bias
    m = X.shape[1]
    A = sigmoid (np.dot (w.T, X) + b)
    cost = -1/m * np.sum (Y* np.log (A) + (1 - Y) * np.log (1-A))
    dw = 1/m * np.dot (X, (A-Y).T)
    db = 1/m * np.sum (A - Y)
    grads = { "dw": dw,
              "db": db }
    return grads, cost

w, b, X, Y = np.array ([[1], [2]]), 2, np.array ([[1, 2], [3, 4]]),
             np.array ([[1, 0]])
grads, cost = propagate (w, b, X, Y)
print ("dw = " + str( grads ["dw"]))
print ("db = " + str (grads ["db"]))
print ("cost = " + str (cost))
```

```
dw = [[ 0.99993216]
      [1.99980262]]
db = 0.499993523062 5
cost = 6.000064773 19
```

w: (num_px* num_px * 3, 1)

4.4 - Optimization (update parameters using gradient descent)

Learn w and b by minimizing the cost function J

$\theta = \theta - \alpha \, d\theta$ α: learning rate

Graded function : optimize

```
def optimize (w, b, X, Y, num_iterations, learning_rate, print_cost=False):

    costs = []
    for i in range(num_iterations):
        grads, cost = propagate (w, b, X, Y)
        dw = grads ["dw"]
        db = grads ["db"]
        w = w - learning_rate * dw
        b = b - learning_rate * db

        if i % 100 == 0:                                  } Record the cost
            costs.append (cost)

        if print_cost and i % 100 = 0:                    } Print the cost
            print ("Cost after iteration %i : %f" % (i, cost))   every 100 training examples

    params = { "w" : w,
               "b" : b }
    grads = { "dw" : dw,
              "db" : db }

    return params, grads, costs

params, grads, costs = optimize (w, b, X, Y, num_iterations = 100, learning_rate =
                                 = 0.009, print_cost = False )
print ("w = " + str (params ["w"]))
print ("b = " + str (params ["b"]))
print ("dw = " + str (grads ["dw"]))
print ("db = " + str (grads ["db"]))
```

w = [[0.1124579]
[0.23106775]]

b = 1.55930992989

dw = [[0.90158428]
[1.76250842]]

db = 0.43046207167 9

Graded function : predict

```
def predict (w, b, X) :
    m = X.shape [1]
    Y_prediction = np.zeros ((1, m))
    w = w.reshape (X.shape [0], 1)
    A = sigmoid (np.dot (w.T, X) + b)
    for i in range (A.shape [1]) :
        if A [0, i] <= 0.5 :
            Y_prediction [0, i] = 0
        else:
            Y_prediction [0, i] = 1
return Y_prediction
```

Cost after iteration 0 : 0.693147

...

Cost after iteration 1900 : 0.140872

train accuracy : 99.043 %

test accuracy : 70 %

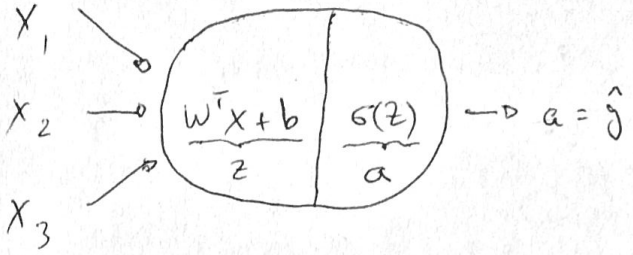
```
# Gradient function : model
def model (X_train, Y_train, X_test, Y_test, num_iterations = 2000,
        learning_rate = 0.5, print_cost = False )

    # Initialize parameters with zeros
    w, b = initialize_with_zeros (X_train.shape[0])

    # Gradient descent
    parameters, grads, costs = optimize (w, b, X_train, Y_train, num_iterations,
                                         learning_rate, print_cost)

    # Retrieve parameters w and b from dictionary "parameters"
    w = parameters["w"]
    b = parameters["b"]

    # Predict test / train set examples
    Y_prediction_test = predict (w, b, X_test)
    Y_prediction_train = predict (w, b, X_train)

    # Print train/test errors
    print ("train accuracy : {} %".format (100 - np.mean(np.abs(Y_prediction_train -
        - Y_train)) * 100)
    print ("test accuracy : {} %".format (100 - np.mean(np.abs(Y_prediction_test -
        - Y_test * 100 )

    d = {"costs": costs,
        "Y_prediction_test": Y_prediction_test,
        "Y_prediction_train": Y_prediction_train,
        "w": w,
        "b": b,
        "learning_rate": learning_rate,
        "num_iterations": num_iterations}

    return d

d = model (train_set_x, train_set_y, test_set_x, test_set_y, num_iterations = 2000,
        learning_rate = 0.005, print_cost = True )
```

Week 3

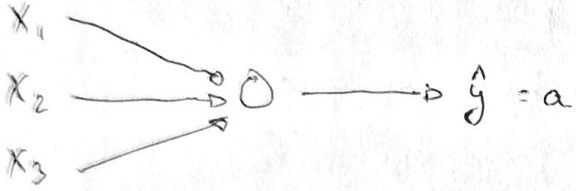
$x^{(i)} \neq x^{[i]}$

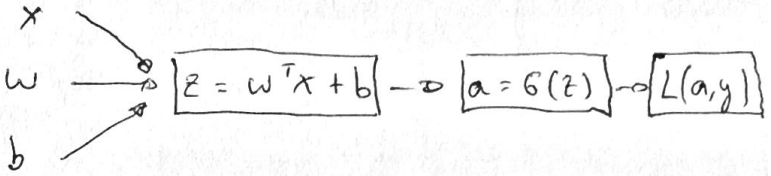
$\llcorner \to$ layer i

$\llcorner \to$ i^{th} piece of training

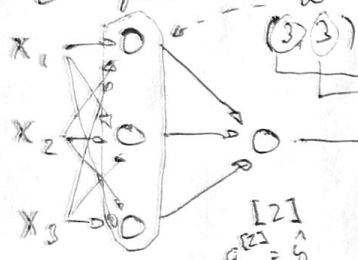


1 layer





2 layers



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

$(3, 3)$ $(3, 1)$

n° nodes n° inputs

$a^{[1]}$: activation function of i+1 layer

$a^{[0]} = x$ [1]

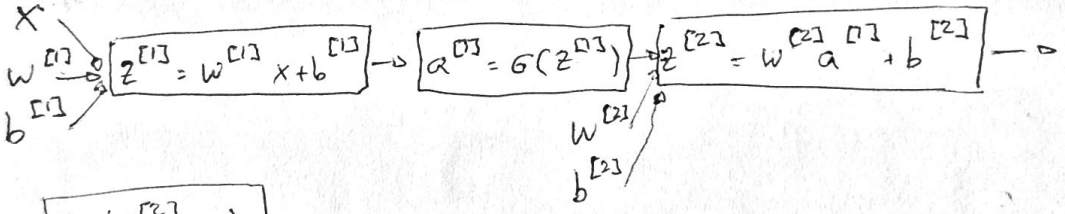
input layer

$a^{[1]}$ = hidden layer

[2]

$a^{[2]} = \hat{y}$

output layer

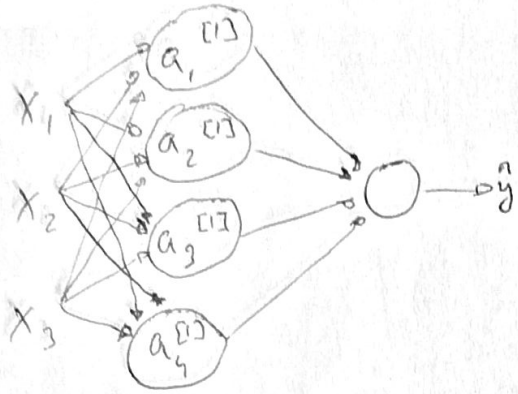


$\to \boxed{a^{[2]} = \sigma(z^{[2]})} \to \boxed{L(a^{[2]}, y)}$

Input layer is not counted for defining the dimensions (of layers) of a neural network

$a^{[l]}$ ← layer

a_i ← node in layer



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

$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]} \qquad a_2^{[1]} = \sigma(z_2^{[1]})$$

$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]} \qquad a_3^{[1]} = \sigma(z_3^{[1]})$$

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

Vectorization

$$z^{[1]} = \underbrace{\begin{bmatrix} \text{---} & w_1^{[1]T} & \text{---} \\ \text{---} & w_2^{[1]T} & \text{---} \\ \text{---} & w_3^{[1]T} & \text{---} \\ \text{---} & w_4^{[1]T} & \text{---} \end{bmatrix}_{4\times 3}}_{W^{[1]}} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_{3\times 1}}_{[x]} + \underbrace{\begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}_{4\times 1}}_{b^{[1]}} = \begin{bmatrix} w_1^{[1]T} x + b_1^{[1]} \\ w_2^{[1]T} x + b_2^{[1]} \\ w_3^{[1]T} x + b_3^{[1]} \\ w_4^{[1]T} x + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}_{4\times 1}$$

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix}_{4\times 1} = \sigma(z^{[1]})$$

$$\underset{(4,1)}{z^{[1]}} = W^{[1]} \underbrace{x}_{a^{[0]}} + b^{[1]} = \underset{(4,3)}{W^{[1]}} \underset{(3,1)}{a^{[0]}} + \underset{(4,1)}{b^{[1]}}$$

$$\underset{(4,1)}{a^{[1]}} = \underset{(4,1)}{\sigma(z^{[1]})}$$

$$\underset{(1,1)}{z^{[2]}} = \underset{(1,4)}{W^{[2]}} \underset{(4,1)}{a^{[1]}} + \underset{(1,1)}{b^{[2]}}$$

$$\underset{(1,1)}{a^{[2]}} = \underset{(1,1)}{\sigma(z^{[2]})}$$

$a^{[l](i)}$

i — example i

l — layer l

$$X \longrightarrow a^{[2]} = \hat{y}$$
$$x^{(1)} \longrightarrow a^{[2](1)} = \hat{y}^{(1)}$$
$$x^{(2)} \longrightarrow a^{[2](2)} = \hat{y}^{(2)}$$
$$\vdots \qquad \vdots \qquad \vdots$$
$$x^{(m)} \longrightarrow a^{[2](m)} = \hat{y}^{(m)}$$

$$z^{[1]} = w^{[1]} a^{[0]} + b^{[1]}$$
$$a^{[1]} = \sigma(z^{[1]})$$
$$z^{[2]} = w^{[2]} a^{[1]} + b^{[2]}$$
$$a^{[2]} = \sigma(z^{[2]})$$

for $i = 1$ to m

$$z^{[1](i)} = w^{[1]} a^{[0](i)} + b^{[1]}$$
$$a^{[1](i)} = \sigma(z^{[1](i)})$$
$$z^{[2](i)} = w^{[2]} a^{[1](i)} + b^{[2]}$$
$$a^{[2](i)} = \sigma(z^{[2](i)})$$

VECTORIZATION

$$z^{[1]} = w^{[1]} \overset{A^{[0]}}{X} + b^{[1]}$$
$$A^{[1]} = \sigma(z^{[1]})$$
$$z^{[2]} = w^{[2]} A^{[1]} + b^{[1]}$$
$$A^{[2]} = w^{[2]} A^{[1]} + b^{[2]}$$

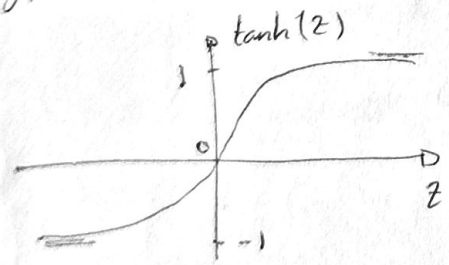
$$z^{[1]} = \begin{bmatrix} | & | & & | \\ z^{[1](1)} & z^{[1](2)} & \cdots & z^{[1](m)} \\ | & | & & | \end{bmatrix}$$

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

$$A^{[1]} = \underbrace{\begin{bmatrix} a^{[1](1)} & a^{[1](2)} & \cdots & a^{[1](m)} \end{bmatrix}}_{\left(\begin{array}{c}\#\text{ nodes in layer } 1 \\ \text{hidden units}\end{array} \times m \text{ examples}\right)}$$

Instead of using sigmoid function as activation function, is possible to use another alternatives:

- Hyperbolic tangent: $a^{[l]} = \tanh(z^{[l]})$

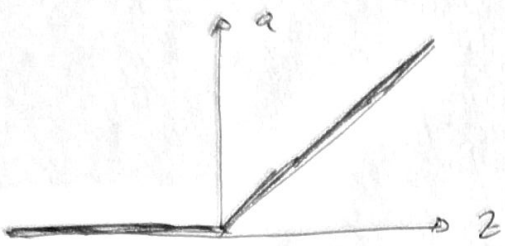


$$a = \tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

Down side: The gradient descent is slowed down because the slope is practically null when z is large or short

↳ The same with sigmoid $\sigma(z)$ functions happens

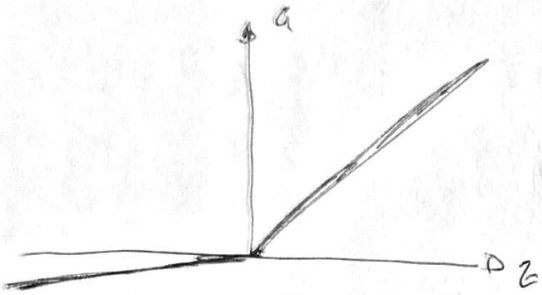
- ReLU (Rectified Linear Unit)



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

$$\frac{da}{dz} = \begin{cases} 0 & z < 0 \\ 1 & z \geq 0 \end{cases}$$

- Leaky ReLU → works better although is not more used than ReLU

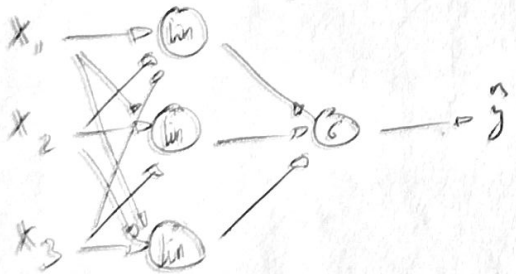


$$a = \max(0.01z, z)$$

With these both, the neural network runs faster

Sigmoid is OK just for classifications (i.e. cat vs non-cat)

Linear activation function? No, only when $\hat{y} \in \mathbb{R}$ (is a number) and only the output layer is linear
or not having it



Given x:

$$\begin{cases} z^{[1]} = W^{[1]} x + b^{[1]} \\ a^{[1]} = z^{[1]} \\ z^{[2]} = W^{[2]} x + b^{[2]} \\ a^{[2]} = z^{[2]} \end{cases}$$

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

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

$$= (W^{[2]} W^{[1]}) x + (W^{[2]} b^{[1]} + b^{[2]}) = W' x + b'$$

No matter how many layers the NN has, it's always computing linear activation function. NN are not more expressive than a standard logistic regression without any hidden layer.

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

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

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

$$g(z) = \tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}} \qquad g'(z) = 1 - \tanh^2(z)$$

Gradient descent to train parameters : $\underset{(n^{[1]}, n^{[0]})}{w^{[1]}}, \underset{(n^{[1]}, 1)}{b^{[1]}}, \underset{(n^{[2]}, n^{[1]})}{w^{[2]}}, \underset{(n^{[2]}, 1)}{b^{[2]}}$

$n_x = n^{[0]}, \underset{\substack{\downarrow \\ \text{\# variables in} \\ \text{hidden layer}}}{n^{[1]}}, n^{[2]} = 1$

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

Compute predicts $(\hat{y}^{(i)}, i = 1, \dots, m)$

$$dw^{[i]} = \frac{\partial J}{\partial w^{[i]}}, \qquad db^{[i]} = \frac{\partial J}{\partial b^{[i]}}$$

$$w^{[i]} = w^{[i]} - \alpha\, dw^{[i]} \qquad b^{[i]} = b^{[i]} - \alpha\, db^{[i]}$$

Forward propagation

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

Back propagation

$$dZ^{[2]} = A^{[2]} - Y \qquad Y = [y^{(1)} y^{(2)} \dots y^{(m)}] \; (n^{[2]},)$$
$$dw^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T} \qquad \downarrow (n^{[2]}, 1)$$
$$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$$
$$dZ^{[1]} = \underbrace{w^{[2]T} dZ^{[2]}}_{(n^{[1]}, m)} * \overbrace{g^{[1]\prime}(Z^{[1]})}^{(n^{[1]}, m)}$$

element-wise

$$dw^{[1]} = \frac{1}{m} dZ^{[1]} X^{T}$$
$$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, \overbrace{keepdims = True}^{(n^{[1]}, 1)})$$

It's necessary to initialize the parameters randomly. W necessary, b not really. If they are initialized with zeros, after any iteration, every component of W would behave as it was the same function (different neurons, turn out to behave as clons) which is useless. Symetry problem.

$W^{[1]}$ = np.random.randn((2,2)) * 0.01
↳ close to zero because it's where the slope is maximum (i.g. in sigmoid function)
↳ max (Gradient descent)
↳ fastest possible learning

$b^{[1]}$ = np.zero((2,1))

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

$b^{[2]}$ = 0

$$axis = \begin{cases} 0 \rightarrow \text{sum rows} \\ 1 \rightarrow \text{sum columns} \end{cases}$$

$$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$$
$$1 \leq l \leq L$$

- X is a matrix in which each column is one training example
- ~~$a_4^{[2]}$ is the activation output of the 2nd layer for the 4th training example~~
- $a^{[2]}$ denotes the activation vector of the 2nd layer
- $a_4^{[2]}$ is the activation output by the 4th neuron of the 2nd layer
- $a^{[2](12)}$ denotes the activation vector of the 2nd layer for the 12th training example
- The tanh activation usually works better than sigmoid activation function for hidden units because the mean of its output is closer to zero, and so it centers the data better for the next layer
- Sigmoid activation function is recommended for binary classificators
- If weights and biases are initialized to be zero, each neuron in the first hidden layer will perform the same computation. So even after multiple iterations of gradient descent each neuron in the layer will be computing the same thing as other neurons
- Logistic regression's weights can be initialized to be zeros because don't have a hidden layer

- If you have a NN using tanh activation for all hidden units and the weight are initialized with large values, will cause ~~the inputs of~~ ~~tanh~~ gradients to be close to zero. The optimization algorithm will thus become slow.

- Consider the following 1 hidden layer in a NN with 4 neurons and 2 inputs

$b^{[1]}$ will have shape $(4, 1)$

$W^{[1]}$ will have shape $(4, 2)$

$W^{[2]}$ will have shape $(1, 4)$

$b^{[2]}$ will have shape $(1, 1)$

$Z^{[1]}$ and $A^{[1]}$ are $(4, m)$

Week 4

L (# layers, hidden + output ones)

$n^{[l]}$ (# units in layer l) $\qquad l \in (0, L)$

$\hookrightarrow n^{[0]} = n_x, \quad n^{[L]} =$ n° outputs

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

$w^{[l]}$ (weights for $z^{[l]}$)

$b^{[l]}$ (biases for $z^{[l]}$)

$\hookrightarrow a^{[0]} = x, \quad a^{[L]} = \hat{y}$

Forward propagation

$$\left.\begin{aligned} z^{[l](i)} &= w^{[l]} a^{[l-1](i)} + b^{[l]} \\ a^{[l](i)} &= g^{[l]}(z^{[l](i)}) \end{aligned}\right\} \text{Stacking all training columns in a matrix} \left\{\begin{aligned} Z^{[l]} &= w^{[l]} A^{[l-1]} + b^{[l]} \\ A^{[l]} &= g^{[l]}(Z^{[l]}) \end{aligned}\right.$$

~~VECTORIZATION~~

for l to L

for i to m

for l to L

dimensions: $\left.\begin{aligned} w^{[l]} &: (n^{[l]}, n^{[l-1]}) \\ a^{[l](i)} &: (n^{[l]}, 1) \end{aligned}\right\} \quad Z^{[l](i)} : (n^{[l]}, n^{[l-1]}) \times \underbrace{(n^{[l]}, 1)}_{(n^{[l-1]}, 1)} :$

$b^{[l]} : z^{[l](i)} : w^{[l]} \times a^{[l](i)} \quad : (n^{[l]}, 1)$

$dw^{[l]} : w^{[l]} \qquad db^{[l]} : b^{[l]}$

$z^{[l](i)} : a^{[l](i)}$

only this changes dimensions

Vectorization: $Z^{[L]} = w^{[L]} A^{[L-1]} + b^{[L]}$

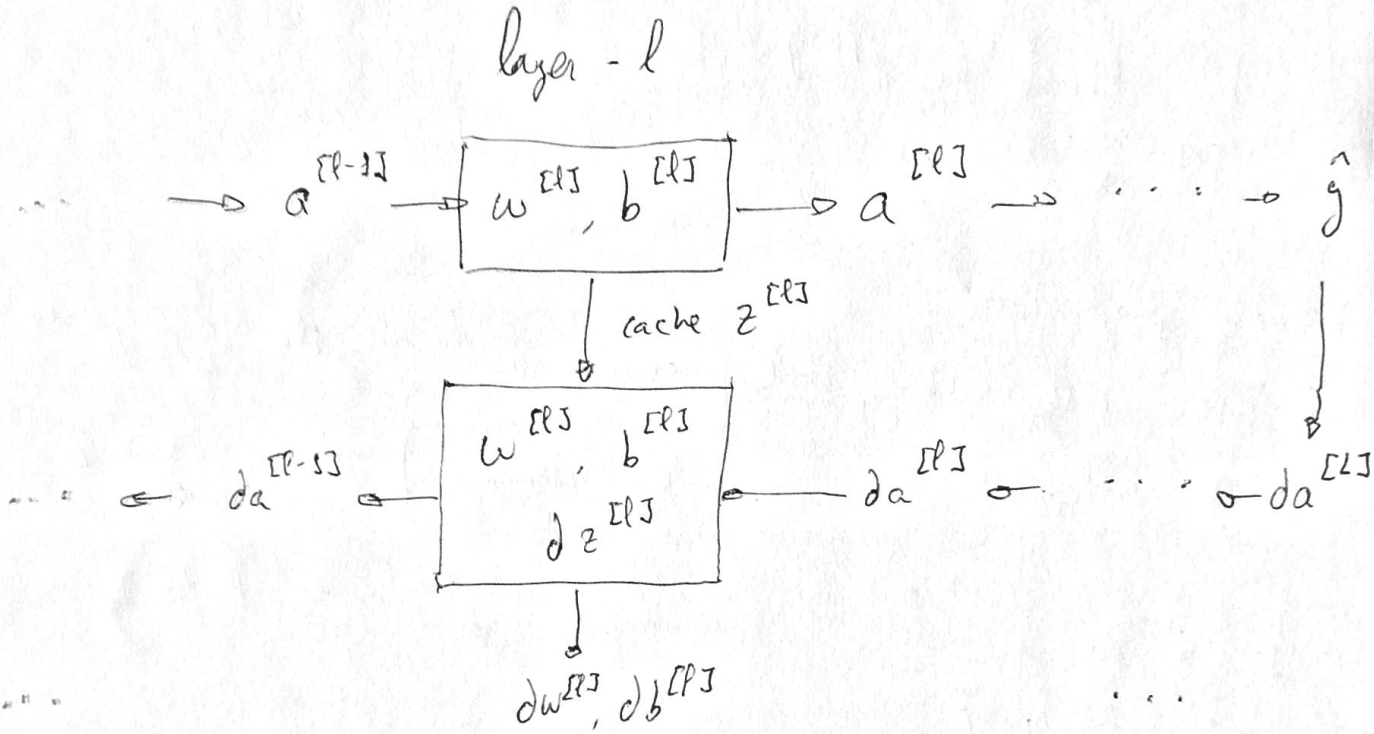
$(n^{[L]}, m) = (n^{[l]}, n^{[l-1]})(n^{[l]}, m) + \underbrace{(n^{[l]}, 1)}_{(n^{[l]}, m)}$ broadcasting Python

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

deep NN $\rightarrow O(\log x_n)$

shallow NN $\rightarrow O(2^n)$

↳ 1 hidden layer



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

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

Forward propagation

Input : $a^{[l-1]}$ $\quad w^{[l]}, b^{[l]}$

Output : $a^{[l]}$, cache $(z^{[l]})$

Backward propagation

Input : $da^{[l]}$

Output : $da^{[l-1]}$, $dw^{[l]}$, $db^{[l]}$

$$dz^{[l]} = da^{[l]} * g^{[l]\prime}(z^{[l]}) = w^{[l+1]T} dz^{[l+1]} * g^{[l]\prime}(z^{[l]})$$

$$dw^{[l]} = dz^{[l]} a^{[l-1]}$$

$$db^{[l]} = dz^{[l]}$$

$$da^{[l-1]} = w^{[l]T} dz^{[l]}$$

$$dZ^{[l]} = dA^{[l]} * g^{[l]\prime}(Z^{[l]})$$

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

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

$$dA^{[l-1]} = W^{[l]T} dZ^{[l]}$$

$$L(\hat{y}, y) \longrightarrow da^{[l](i)} = -\frac{y}{a} + \frac{(1-y)}{(1-a)}$$

$$\downarrow$$

$$dA^{[l]} = \left(-\frac{y^{(i)}}{a^{(i)}} + \frac{(1-y^{(i)})}{(1-a^{(i)})} \right)$$

Parameters : $W^{[l]}, b^{[l]}$

Hyperparameters : Learning rate α

\# iterations

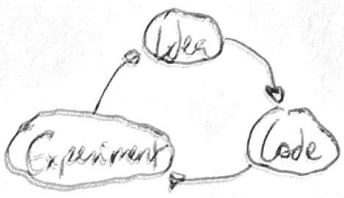
\# hidden layers L

\# hidden units $n^{[l]}$

choice of activation function

...

Applied deep learning is a very empirical process



What is the "cache" used for in our implementation of forward propagation and backward propagation?

We use it to pass variables computed during forward propagation to the corresponding backward propagation step. It contains useful values for backward propagation to compute derivatives (chain rule derivatives)

The deeper layers of a neural network are typically computing more complex features of the input than the deeper layers

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

During the backpropagation you need to know which activation was used in the forward propagation to be able to compute the correct derivative

To compute the function using a shallow network circuit, you'll need a large network (# logic gates), but to compute it using a deep NN, you only need exponentially smaller network.

Course 2

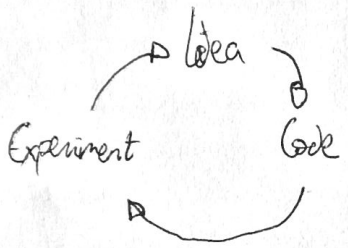
Week 1

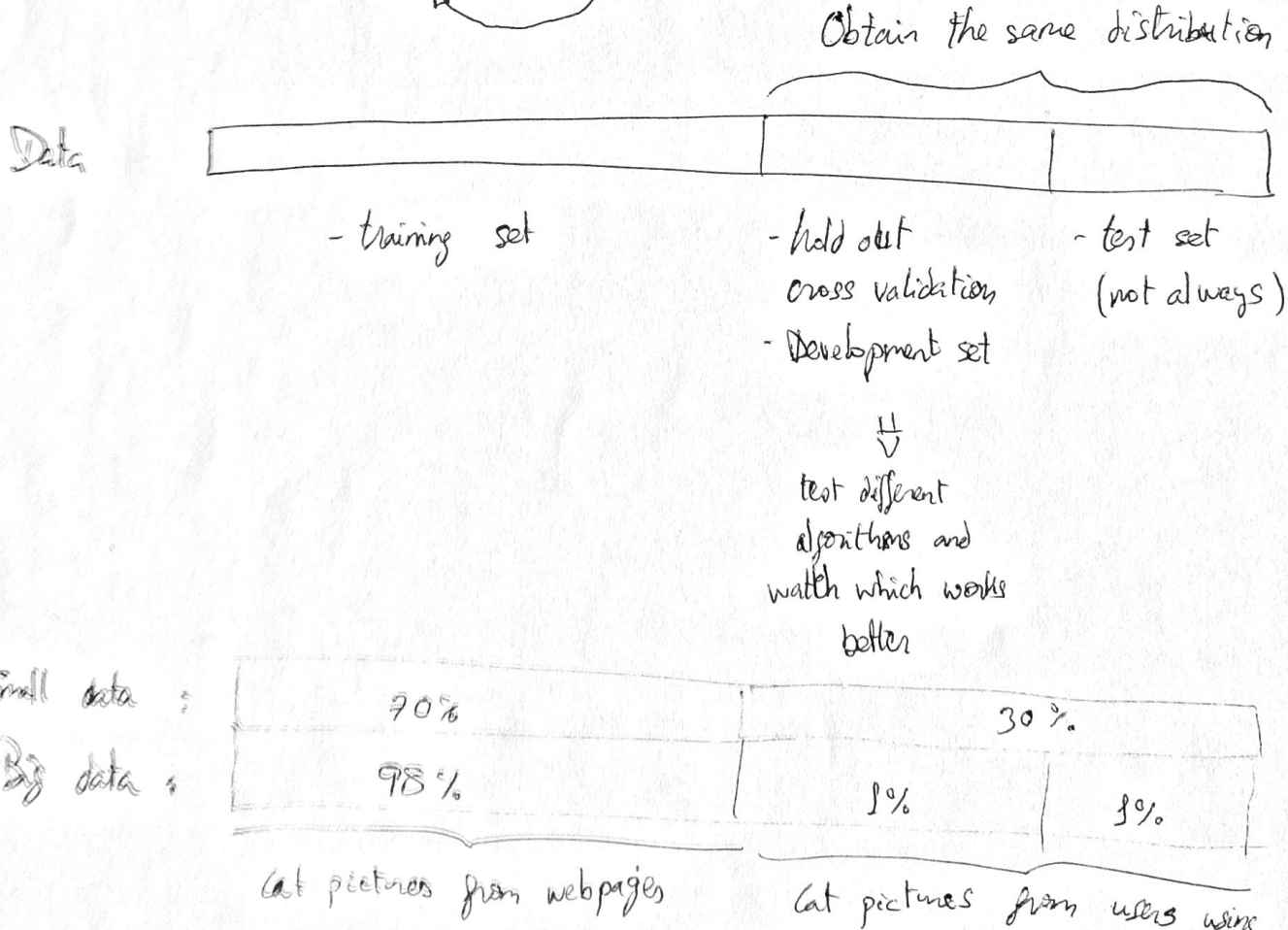
The choice of hyperparameters may depend on:

- Amount of data
- # inputs
- training on GPU or CPU

...

It's impossible at first to guess correctly the hyperparameters, so it's all about try and error:



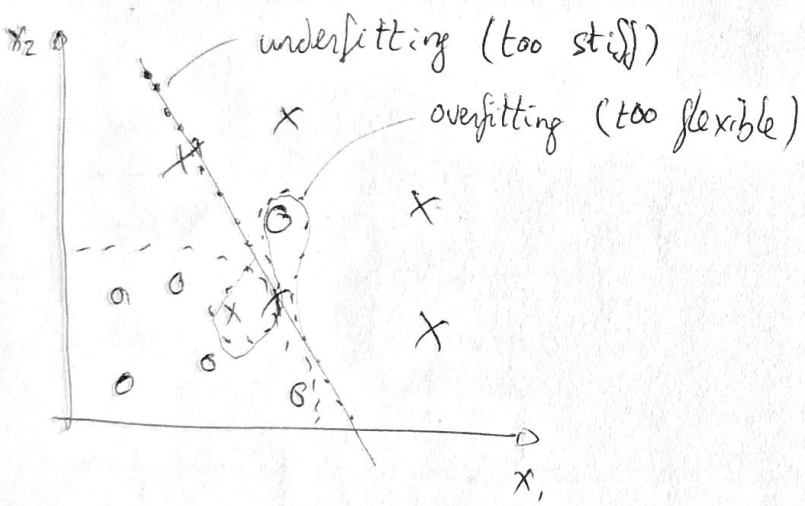


Bias and Variance

Cat classification Yes: $y = 1$ No: $y = 0$

Train set error	1%	15%	15%	0.5%
Dev set error	11%	16%	30%	1%
	high variance	high bias underfitting data	high bias high variance overfitting data	low bias low variance

Human: ≈ 0% Optimal (Bayesian) error



High bias?
(training data performance)
— Yes, try new → Bigger network; Train longer; Another NN architecture

↓ No

High variance?
(dev set performance)
— Yes → More training data; Regularization; Another NN architecture

↓ No

Done

Bias	variance	track off
↑	↓	
↓	↑	
↑	↑	

Regularization

- logistic Regression

$$\min_{w,b} J(w,b) \longrightarrow J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \boxed{\frac{\lambda}{2m}\|w\|_2^2}$$

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

L_1 regularization $\|w\|_1 = \sum_{j=1}^{n_x} |w_j| \longrightarrow \frac{\lambda}{2m}\|w_1\|$

- Neural Network

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

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

Frobenius norm $\Rightarrow$ matrixes $\|\cdot\|_2^2 \neq \|\cdot\|_F^2$

$$dw^{[l]} = (\text{from backpropagation}) + \frac{\lambda}{m} w^{[l]} \qquad (\text{weight decay}) = \frac{\partial J}{\partial w^{[l]}}$$

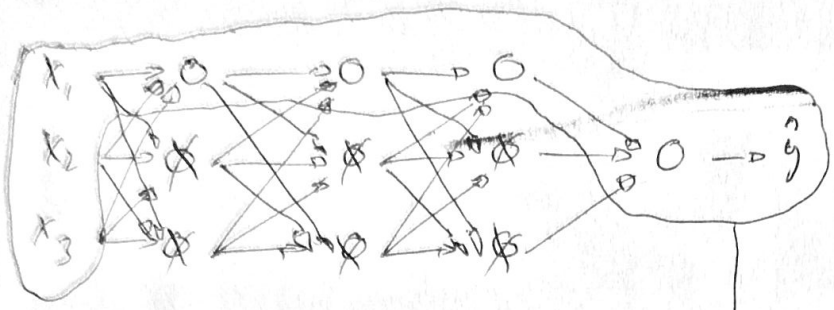
$$w^{[l]} := w^{[l]} - \alpha\, dw^{[l]} = w^{[l]} - \alpha\left[(\text{from backpropagation}) + \frac{\lambda}{m} w^{[l]}\right] =$$

$$= w^{[l]} - \alpha\frac{\lambda}{m} w^{[l]} - \alpha(\text{from backpropagation}) =$$

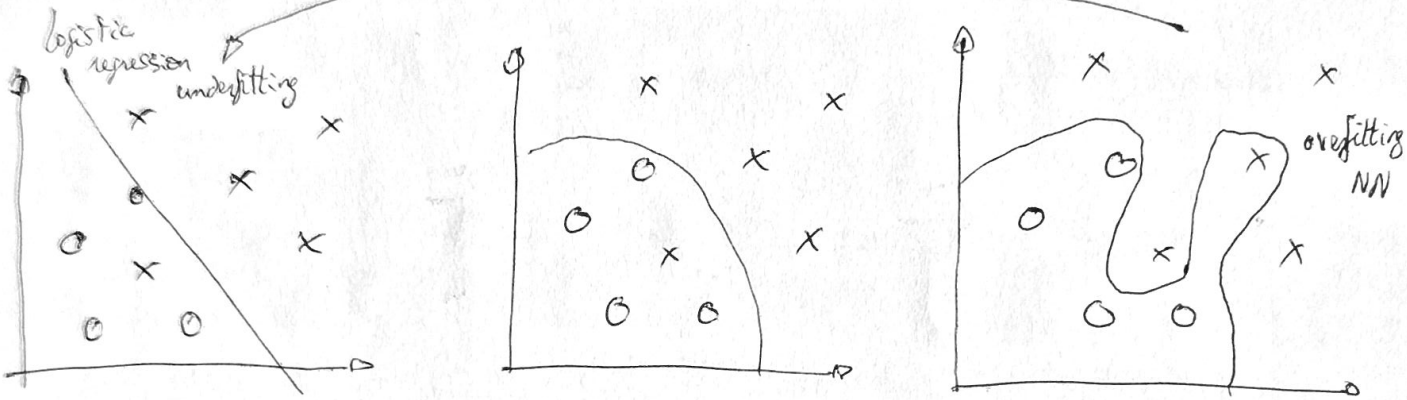
$$= \left(1 - \frac{\alpha\lambda}{m}\right) w^{[l]} - \alpha(\text{from backpropagation})$$

The extra term of the cost function penalizes the weight matrices from being too large so that was the frobenius norm

$\uparrow \lambda \quad \downarrow w^{[l]}$ | $\uparrow\uparrow\uparrow \lambda \quad w^{[l]} \approx 0$ → zeroing hidden units
↳ minimizing the impact of them



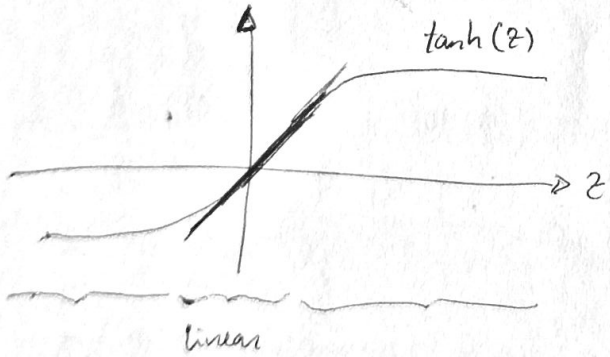
too high value of lambda



high bias | "just right" | high variance

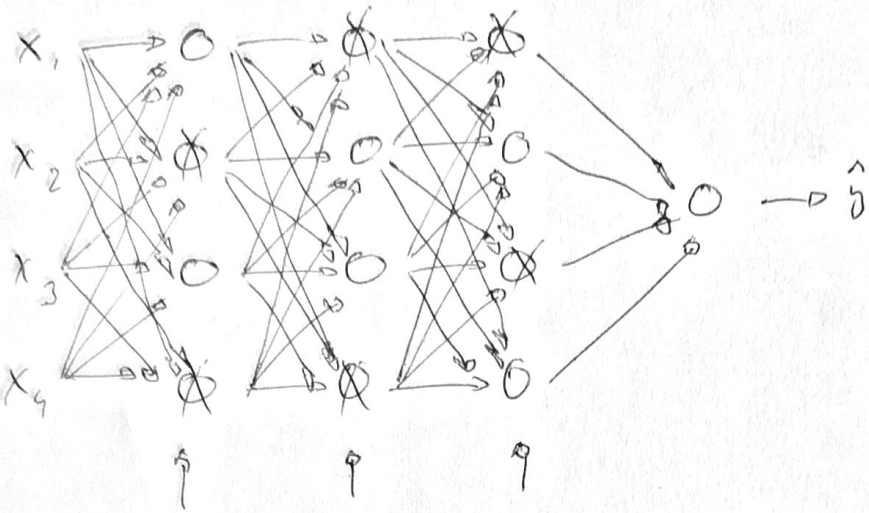
intermediate value of lambda

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

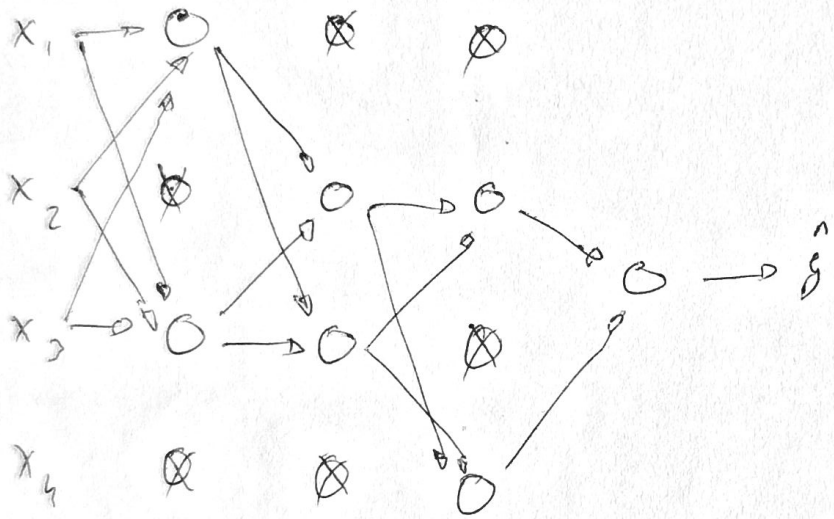


↳ activations of NN working on linear range → logistic regression
↳ linear layers

Dropout regularization



← give a chance to disable hidden units



Implementation of inverted dropout

keep_prob = 0.8 → (remove_prob = 0.2)

← dropout of l-layer

d(l-layer) = np.random.rand(al.shape[0], al.shape[1]) < keep_prob

al = np.multiply(al, dl)

al := al / keep_prob ← inverted dropout technique

50 units → 10 units shut off

$z^{[4]} = w^{[4]} a^{[3]} + b^{[4]}$

[reduced by 20%

No dropout at test time → only at training time

$a^{[0]} = x$

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

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

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

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

$\vdots$

$\hat{y}$

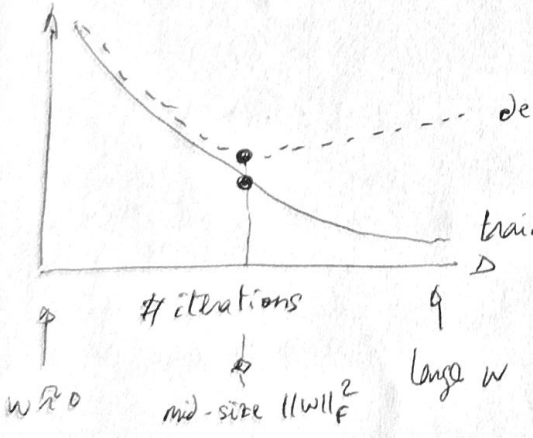
No dropout

There could be different values of keep_probs regarding the hidden layer

Another regularization techniques

- Data augmentation : example → flip randomly or horizontaly pictures
  rotate and add random distortions
  not vertically flipping is common

- Early stopping



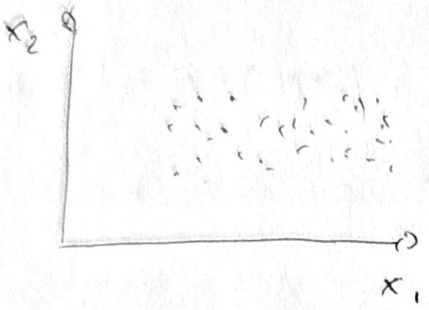
randomized closed to zero

Orthogonalization :

- Optimize cost function J
  - gradient, ...
- Not overfit
  - regularization, more data, ...

Normalize training set

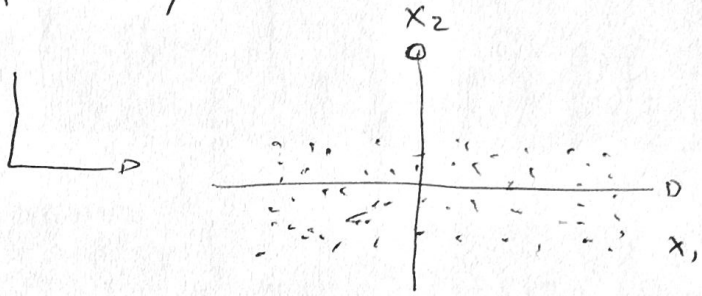
from a scatterplot



1° Substract mean:

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

$$X := X - \mu$$

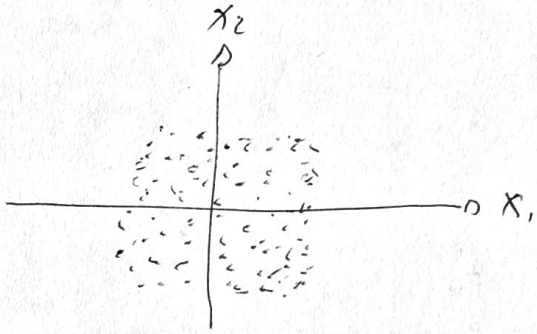


2° Normalize variance

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

element-wise

$$X := X / \sigma^2$$



Same μ and σ^2 for training and test set

Normalization makes data more symmetric and centric which makes descent gradient more stable and optimal → algorithm runs faster

Vanishing / exploding gradients

In very deep NN (with many layers):

$W^{[l]} > I$ → gradient explodes exponentially

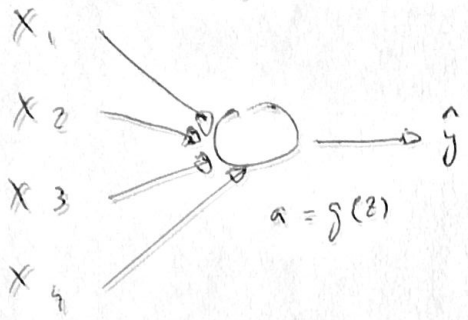
$W^{[l]} < I$ → " decreases "

Solution: carefully initialize the weights

$\hat{y} = W^{[L]} W^{[L-1]} \cdots W^{[2]} W^{[1]} X$ if $g(z) = z$ and $b^{[l]} = 0$

$z = w_1 x_1 + w_2 x_2 + \dots + w_n x_n \cancel{+ b}$

large $n \to$ smaller w_i



$$Var(w_i) = \frac{1}{n} \text{ or } \frac{2}{n} \text{ or } \dots$$

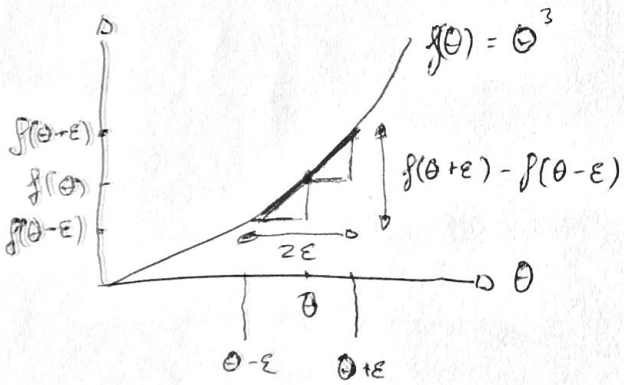
$$w^{[l]} = np.random.randn(shape) * np.sqrt(\gamma)$$

$$\gamma = \begin{cases} \frac{2}{n^{[l-1]}} & \text{when } g^{[l]}(z) = ReLU(z) \\ \frac{1}{n^{[l-1]}} & \text{when } g^{[l]}(z) = \tanh(z) \\ \frac{2}{n^{[l-1]} + n^{[l]}} & \text{Xavier initialization} \end{cases}$$

Gradient check $\to$ not to use in training, just to debug

$\to$ it does not work with <u>dropout</u>

Numerical approximation



$$\underset{g(\theta)}{f'(\theta)} \overset{\text{slope}}{\approx} \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon}$$

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

much more accurate than

$$f(\theta) = \lim_{\varepsilon \to 0} \frac{f(\theta+\varepsilon) - f(\theta)}{\varepsilon}$$

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

to big vector

$$(dW^{[1]}, db^{[1]}, \dots, dW^{[L]}, db^{[L]}) = d\theta$$

for each i: $J(\theta) = J(\theta_i) = J(\theta_1, \theta_2, \dots)$

$$d\theta_{approx}[i] = \frac{J(\theta_1, \theta_2, \dots, \theta_i + \varepsilon, \dots) - J(\theta_1, \theta_2, \dots, \theta_i - \varepsilon, \dots)}{2\varepsilon} \approx \underbrace{d\theta[i]}_{\frac{\partial J}{\partial \theta_i}}$$

$$d\theta_{approx} \overset{?}{\approx} d\theta$$

check $\to$ $$\frac{\| d\theta_{approx} - d\theta \|_2}{\| d\theta_{approx} \|_2 + \| d\theta \|_2} \approx \begin{cases} 10^{-7} \to \text{great! OK} \\ 10^{-5} \to \text{mediocre} \\ 10^{-3} \to \text{wrong} \end{cases}$$

With 10,000,000 examples, the train / dev / test set would be
98% / 1% / 1%

The dev and test set should come from the same distribution

If a NN model seems to have high variance things to try would be:

- Add regularization
- Get more data training

With 0.5% of training set error and 7% dev set error, things to try to improve the NN would be:

- Increase the regularization parameter lambda
- Get more training data

The weight decay is a regularization technique (such as L2 regularization) that results in gradient descent shrinking the weights on every iteration

Whe the regularization hyper parameter lambda is increased, weights are pushed toward becoming smaller (closer to 0)

Increasing the parameter keep_prob from 0.5 to 0.6 will likely cause:

- Reducing the regularization effect
- Causing the NN to end up with a lower training set error

Techiques for reducing variance (overfitting) are:

- Dropout
- L2 regularization
- Data augmentation

Inputs x are normalized because it makes the cost function faster to optimize

If a NN model seems to have a high bias, promising thing to do would be:

- make the NN deeper
- increase the number of units in each hidden layer

With the inverted dropout technique, at test time you do not apply dropout (do not randomly eliminate units) and do not keep the 1/keep_prob factor in the calculations used in training.

Week 2

If the training set it's really big $m \sim 1{,}000{,}000$ examples then a way more efficient algorithm is to split that training set in smaller ones $\longrightarrow$ mini-batches ($m \sim 1{,}000$ examples)

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

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

mini-batch t : $x^{\{t\}}, y^{\{t\}}$ $\longrightarrow$ much faster

Batch gradient descent is the standard one using the whole training set as one

1 step of gradient descent using $X^{\{t\}}, Y^{\{t\}}$ (as if $m = 1000$)

for $t = 1, \dots, 1000$ {

Forward propagation on $X^{\{t\}}$

$$\left.\begin{aligned} Z^{[1]} &= W^{[1]} X^{\{t\}} + b^{[1]} \\ A^{[1]} &= g^{[1]}(Z^{[1]}) \\ &\vdots \\ A^{[L]} &= g^{[L]}(Z^{[L]}) \end{aligned}\right\} \text{Vectorized implementation of 1000 examples}$$

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

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

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

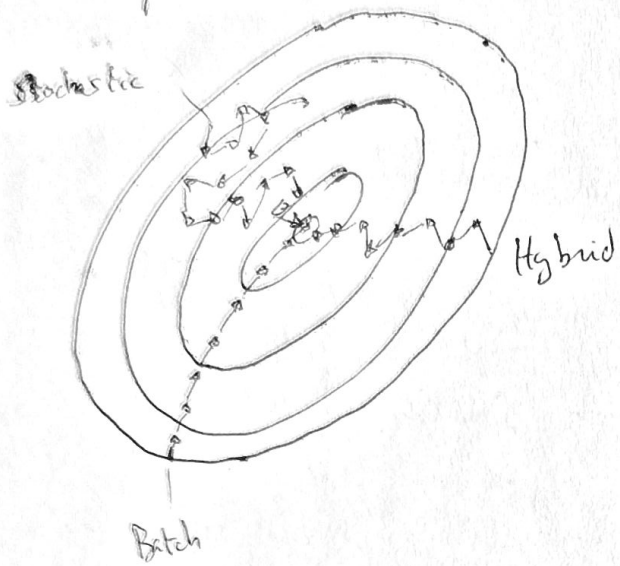
} "1 epoch" $\longrightarrow$ pass through training set

Choosing mini-batch size

if size = m : Batch gradient descent $(X^{\{1\}}, Y^{\{1\}}) = (X, Y)$

if size = 1 : Stochastic gradient descent. Every example is a mini-batch

in practice, size should be in-between 1 and m



Stochastic gradient descent	In-between (mini-batch size not too big / small)	Batch gradient descent
↓	↓	↓
Lose speed up from vectorization	Fastest learning • Vectorization (≈ 1000) • Make progress without processing entire training set	Too long per iteration

if small training set : use Batch gradient descent
$(m \leq 2000)$

typical mini-batch sizes : 2^6 → 64, 128, 256, 512, 1024
2^6 2^7 2^8 2^9 2^{10}

making sure that fits in CPU / GPU memory

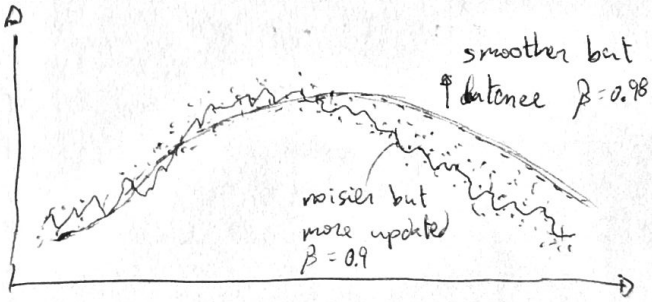
Exponentially weighted average

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

$\beta = 0.9$: ≈ 10 days' temperature

$\beta = 0.98$: ≈ 50 days' temperature

V_t as approximately average over $\approx \frac{1}{1-\beta}$ days' temperature



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

$$v_{100} = 0.9\, v_{99} + 0.1\, \theta_{100}$$

$$v_{99} = 0.9\, v_{98} + 0.1\, \theta_{99}$$

$$v_{98} = 0.9\, v_{97} + 0.1\, \theta_{98}$$

$$\vdots$$

$$v_{100} = 0.1\, \theta_{100} + 0.9\, \cancel{v_{99}}\,(0.1\, \theta_{99} + 0.9\, \cancel{v_{98}}\,(0.1\, \theta_{98} + 0.9\, \cancel{v_{97}}\,(\cdots))) \cdots =$$

$$= 0.1\, \theta_{100} + 0.1 \cdot 0.9\, \theta_{99} + 0.1\,(0.9)^2\, \theta_{98} + 0.1\,(0.9)^3\, \theta_{97} + \cdots =$$

$$v_t = \sum_{n=0}^{t} (\beta - 1)\, \beta^n\, \theta_{t-n}$$

$$(1-\varepsilon)^{1/\varepsilon} = \frac{1}{e} \quad \text{with} \quad \varepsilon = 1-\beta$$

$v_\theta = 0$

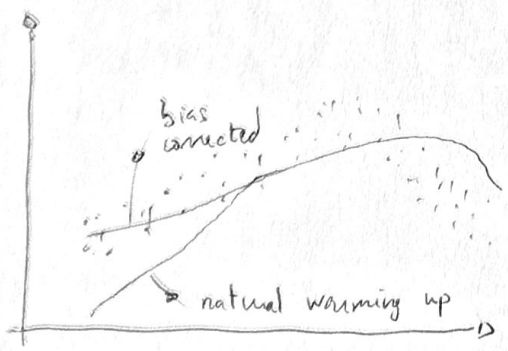
repeat {

get next θ_t

$v_\theta := \beta v_\theta + (1-\beta)\,\theta_t$

}

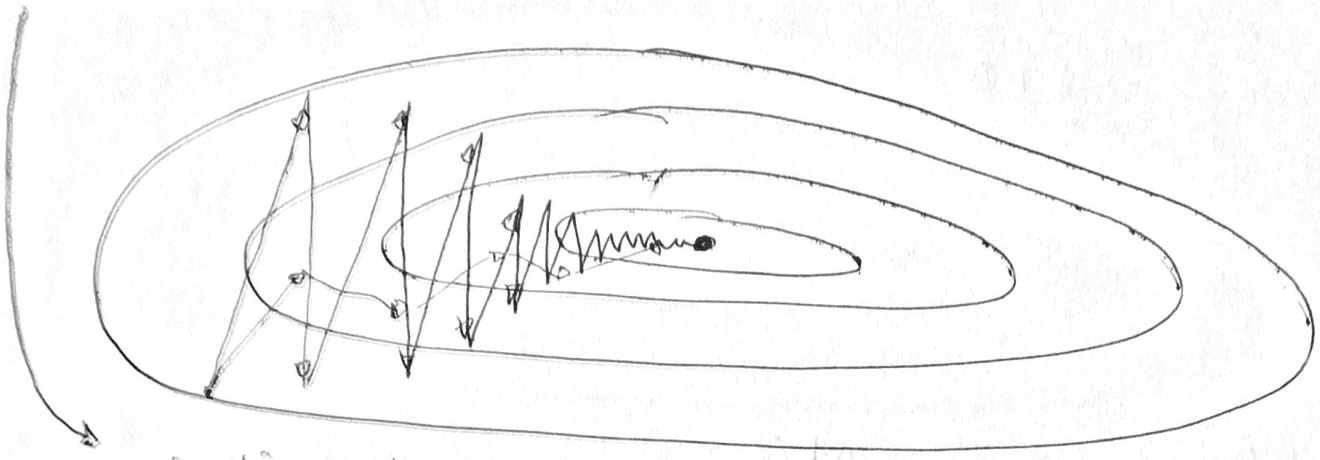
Bias correction for initial phase



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

correction $\rightarrow$ $\dfrac{v_t}{1-\beta^t} = \beta v_{t-1} + (1-\beta)\,\theta_t$

1° Gradient descent with momentum much faster than the standard one



initialization : $V_{dw} = 0 \quad V_{db} = 0$

Momentum :

on iteration t :

compute dw, db on current mini-batch

$$V_{dw} = \beta_1 V_{dw} + (1-\beta_1)\, dw$$

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

"friction" "velocity" "acceleration"

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

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

$\beta = 0.9$ is the common value

slower learning

faster learning

2° RMSprop

on iteration t :

compute dw, db on current mini-batch

element-wise

$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2)\, dw^2 \leftarrow \text{small}$$

$$S_{db} = \beta_2 S_{db} + (1-\beta_2)\, db^2 \leftarrow \text{large}$$

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

$\varepsilon \approx 10^{-8}$ as $\sqrt{S_{dw}}$ is very small

3° Adam optimization algorithm = 1° + 2°

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

On iteration t:

Compute dw, db using current mini-batch

$V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dw$, $V_{db} = \beta_1 V_{db} + (1-\beta_1) db$

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

$V_{dw}^{corrected} = V_{dw} / (1-\beta_1^t)$, $V_{db}^{corrected} = V_{db} / (1-\beta_1^t)$

$S_{dw}^{corrected} = S_{dw} / (1-\beta_2^t)$, $S_{db}^{corrected} = S_{db} / (1-\beta_2^t)$

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

Hyperparameters choice

α: needs to be tune

$\beta_1 : 0.9 \longrightarrow dw, db$

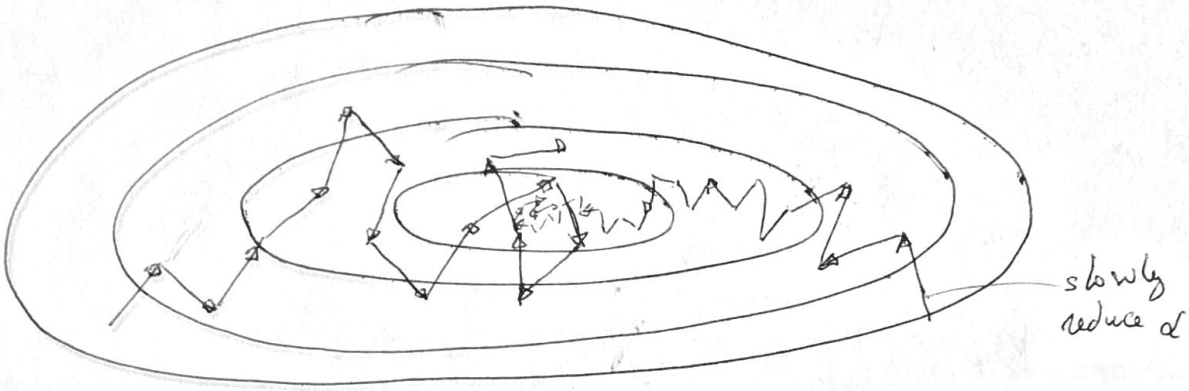
$\beta_2 : 0.999 \longrightarrow dw^2, db^2$

$\varepsilon : 10^{-8}$

$a^{[l]\{t\}(m)}$: $\begin{cases} l\text{- layer} \\ t\text{- mini-batch} \\ m\text{- train} \end{cases}$

If the mini-batch size is m, you end up with batch gradient descent, which has to process the whole training set before making progress. Otherwise, if size is 1, you lose the benefits of vectorization across examples in the mini-batch.

Learning rate decay



1 epoch : 1 pass through the data

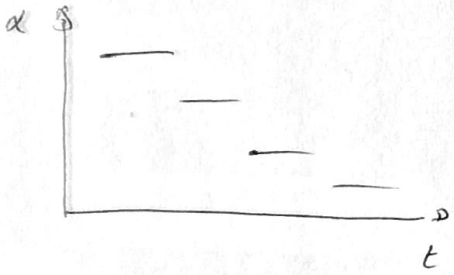
$$\boxed{X^{\{1\}} \quad X^{\{2\}} \quad \dots \quad X^{\{6\}}}$$

$\longrightarrow$ 1 epoch

$\longrightarrow$ 2 epoch

$$\alpha = \frac{1}{1 + \text{decay_rate} * \text{epoch_num}} \alpha_0$$

$$\alpha = \frac{k}{\sqrt{\text{epoch_num}}} \alpha_0 \quad \text{or} \quad \alpha = \frac{k}{\sqrt{t}} \alpha_0 \qquad \alpha = 0.95^t \alpha_0$$



discrete staircase

Plateaus can make learning slow

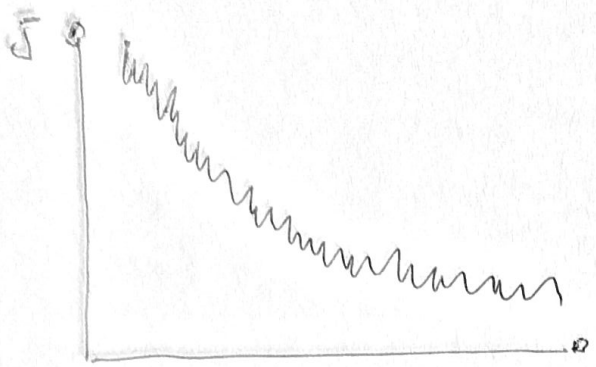
saddle point → puntos de silla de un plano thichdimensional

plateau

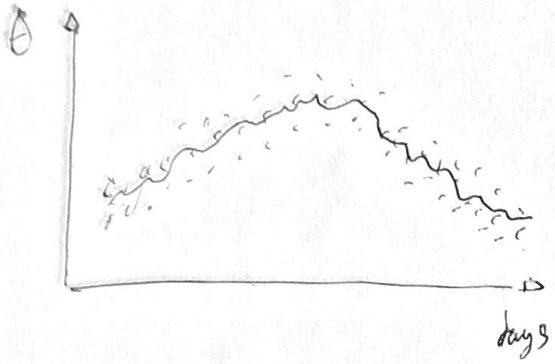
La meseta

(mínimos locales)

Unlikely to get stuck in a bad local optima (minimum)



If you're using mini-batch gradient descent, this looks acceptable. But if you're using batch gradient descent, something is wrong

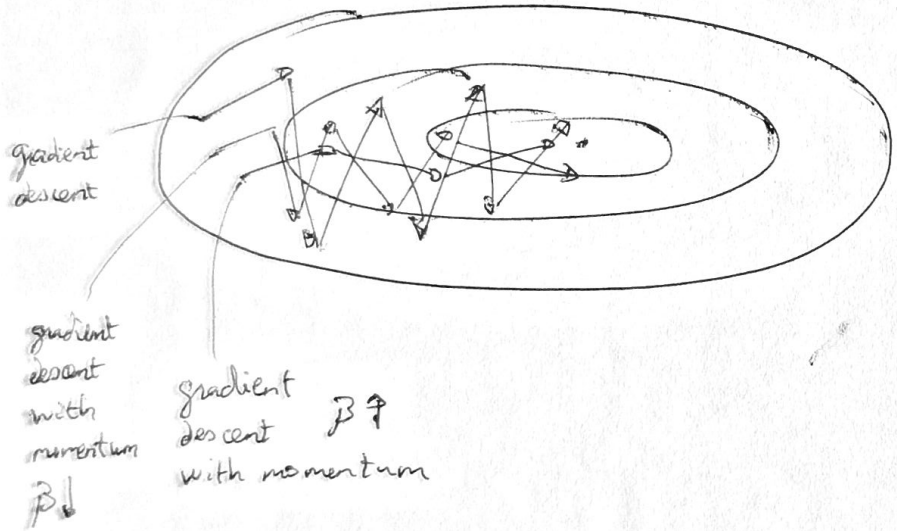


↑ β will shift the function to the right

↓ β will will create more oscillation

If batch gradient descent in a deep network is taking so long to find the value of the parameters that achieves a small value of J, techniques that could help would be:

- Using Adam
- Tuning learning rate α
- Try mini-batch gradient descent
- Try better random initialization of weights



About Adam:

- The learning rate hyperparameter α in Adam usually needs to be tuned
- We usually use default values for the hyperparameters β_1, β_2 and ε in Adam ($\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$)
- Adam combines the advantages of RMSProp and momentum

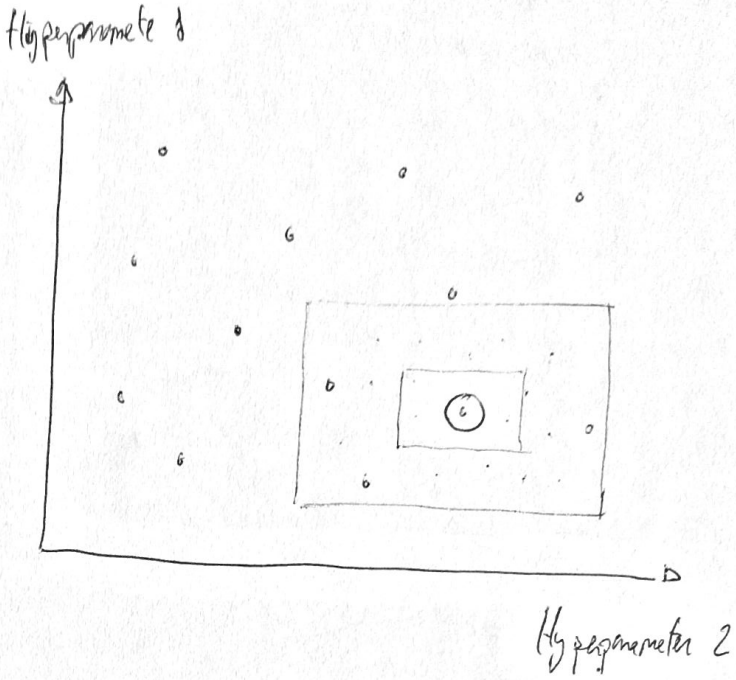
Week 3

Hyperparameters

α
β # hidden units mini-batch size
layers learning rate decay
$\beta_1, \beta_2, \varepsilon$

importance

Coarse to fine



Picking hyperparameters at random

- linear scale

$n^{[l]} = 50, \dots, 100$ # layers L : 2-4

- logarithmic scale

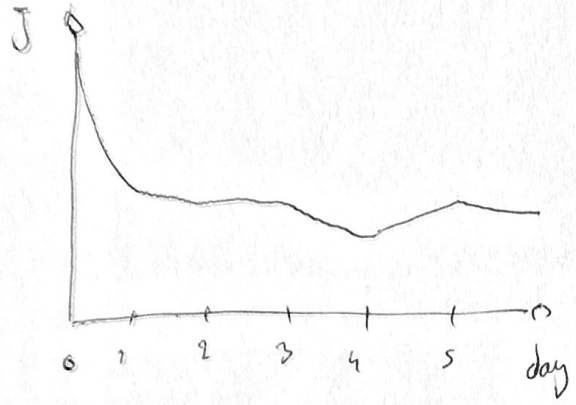
$\alpha = 0.0001, 0.001, 0.01, 0.1, 1.$

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

$\alpha = 10^{+r} \in [10^{-4}, 10^{0}] = [0.001, 1]$

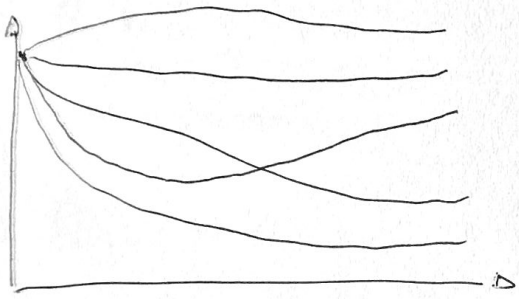
In a tuned model, sometimes a test is required after a long time just to check if the initial hyperparameters are still efficient for the model

If you don't have the computational capacity to train different models (the same one with different hyperparameters), the babysitting one model is the choice.

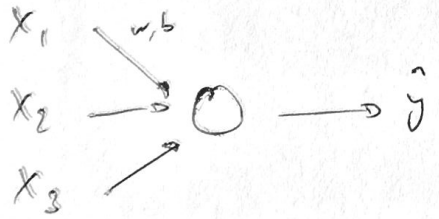


Every a designed time while training a model, hyperparameters are re-tuned a little bit to check responses

The other case is training many models in parallel



Normalizing inputs to speed up learning

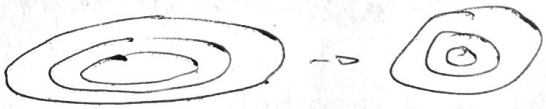


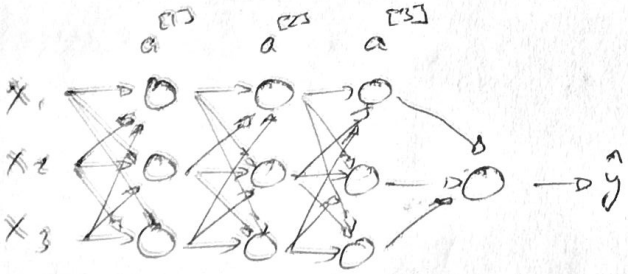
$$\mu = \frac{1}{m} \sum_i x^{(i)}$$

$$x = x - \mu$$

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

$$x = x / \sigma^2$$





$a^{[i]}$ (inputs for the next layer) can be normalized by normalizing $z^{[i]}$ so as to train w, b faster

Implementing Batch Norm to the hidden unit values

$$\mu = \frac{1}{m} \sum_i z^{(i)}$$

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

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

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

learnable parameters

If

$$\gamma = \sqrt{\sigma^2 + \varepsilon}$$

$$\beta = \mu$$

then $\tilde{z}^{(i)} = z^{(i)}$

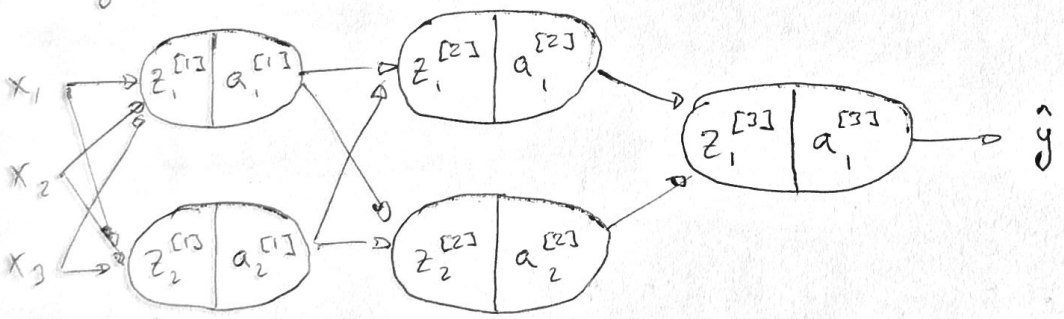
Given some intermediate values in NN

$z^{(i)}, \dots, z^{(m)}$

Use

$\tilde{z}^{(i)}, \dots, \tilde{z}^{(m)}$

Adding Batch Norm to a network



$$\underset{(n^{[l]},1)}{x} \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[\text{Batch Normalization}]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \to a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \to \dots \to a^{[L]} = g^{[L]}(\tilde{z}^{[L]})$$

$$\left.\begin{array}{l} d\beta^{[l]} \to \beta^{[l]} = \beta^{[l]} - \alpha\, d\beta^{[l]} \\ \underset{(n^{[l]},1)}{d\gamma^{[l]}} \to \gamma^{[l]} = \gamma^{[l]} - \alpha\, d\gamma^{[l]} \end{array}\right\} \text{extra parameters} \to \neq \beta \text{ from GD with momentum}$$

besides $\left\{ \begin{array}{l} w^{[l]}, dw^{[l]} \\ \cancel{b^{[l]}, db^{[l]}} \end{array}\right.$

With Batch Normalization $b^{[l]}$ is gonna get substracted out as it substracts the mean, so by adding any constant would be cancelled by BN so $b^{[l]}$ can be eliminated

for $t = 1, \dots,$ num_mini_batches

Compute forward propagation with $X^{\{t\}}$

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

Use backpropagation computing $dw^{[l]}$, $\cancel{db^{[l]}}$, $d\beta^{[l]}$, $d\gamma^{[l]}$

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

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

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

Works with momentum, RMSprop and Adam

Batch Norm as regularization: non-intended effect

- Each mini-batch is scaled by the mean/variance computed on just that mini-batch
- This adds some noise to the values $z^{[l]} \to \tilde{z}^{[l]}$ within that mini-batch. So similar to dropout, it adds some noise to each hidden layer's activations
- This has a slight regularization effect

$\uparrow$ mini-batch size $\downarrow$ noise $\downarrow$ regularization effect

BN at test time

μ, σ^2: stimated using exponentially weighted average (accross mini-batches)

$$X^{\{1\}}, \quad X^{\{2\}}, \quad X^{\{3\}}, \dots$$

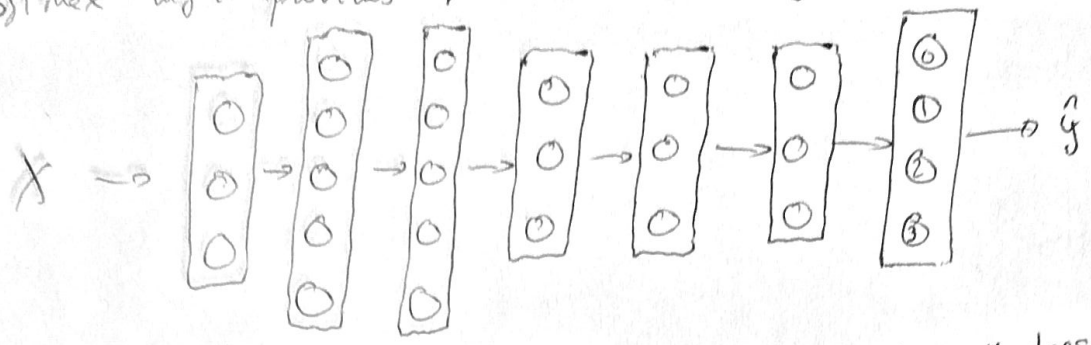
$$\downarrow \qquad \downarrow \qquad \downarrow$$

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

$$\theta_1 \qquad \theta_2 \qquad \theta_3$$

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

Softmax layer provides multi-class classification (binary vector of 0s and 1s)



C : # classes

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

$(4,1)$

Activation function :

$$t = e^{(z^{[L]})}$$

$$\underbrace{a^{[L]}}_{\hat{y}} = \frac{e^{z^{[L]}}}{\sum_{j=1}^{4} t_i}, \quad a_i^{[L]} = \frac{t_i}{\sum_{j=1}^{4} t_i}$$

$(4,1)$

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

$$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix} \qquad t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} \qquad a^{[L]} = g^{[L]}(z^{[L]}) = \begin{bmatrix} e^5/(e^5+e^2+e^{-1}+e^3) \\ e^2/(e^5+e^2+e^{-1}+e^3) \\ e^{-1}/(e^5+e^2+e^{-1}+e^3) \\ e^3/(e^5+e^2+e^{-1}+e^3) \end{bmatrix} =$$

$$= \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix} \xrightarrow[\text{max}]{\text{hard}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Softmax regression generalizes logistic regression to C classes

If $C = 2$, softmax reduces to logistic regression

- If searching among a large ~~amount~~ number of hyperparameters, it's ok to try values randomly
- Some hyperparameters, if set poorly, can have a huge negative impact on training, not all of them.
- During the hyperparameter search, whether you try to babysit one model (Panda strategy) or train a lot of models in parallel (Caviar strategy) is largely determined by the amount of availability of computational power

- Finding good values of hyperparameters is very time-consuming, and it's done iteratively
- In batch normalization is applied on $z^{[l]}$
- In the normalization formula $z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$, ε is used to avoid division by zero
- β and γ can be learned using Adam, GD with momentum or RMSprop, not just GD. They set the mean an variance of the linear variable $z^{[l]}$.
- After training a NN with BN, at test time, to evaluate the NN on a new example you should perform the needed normalizations, use μ and σ^2 estimated using an exponentially weighted average across mini-batches seen during training
- Even if a project is currently open source, good governance of the project helps ensure that it remains open even in the long term, rather than become closed or modified to benefit only one company
- A programming framework allows to code up deep learning algorithms with fewer lines of code than a lower-level language such as Python

Week 1 Course 3

Strategies to improve a NN model

- Collect more data
- Collect more diverse training set
- Train algorithm longer with gradient descent
- Try Adam instead of gradient descent
- Try bigger network or smaller network
- Try dropout
- Add L_2 regularization
- Modify network architecture
  - Activation functions
  - # hidden units
  - ...

Orthogonalization is the process of tuning the hyperparameters of a NN in order to achieve a desired effect

Chain of assumptions in Machine Learning

- Fit training set well on cost function { bigger NN, Adam, ... }
- Fit dev set well on cost function — { Regularization, bigger train set }
- Fit test set well on cost function — bigger dev set
- Performs well in real world — change dev set or cost function

} early stopping

complicated to control as it affects two fields of the NN at the same time

Using a single number evaluation metric

Classifier	Precision (P)	Recall (R)	F1 Score
NN	Of examples recognised as cat, what % actually are cats?	What % of actual cats are correctly recognised?	$\frac{2}{\frac{1}{P} + \frac{1}{R}}$

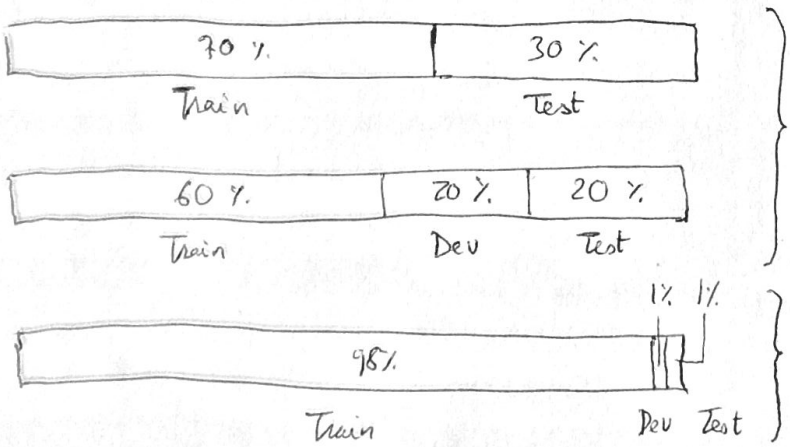
Harmonic mean

Dev set + Single real number evaluation metric → speed up iterating

Classifier	Accuracy	Running time	N metrics
NN	Optimize	Satisficing	→ 1 optimizing
	↳ maximize	↳ a condition	→ N-1 satisficing
	(i.e.: max(accuracy))	(i.e.: $t \leq 100$ ms)	

The input data should be shuffled in order to mix the data so that dev set and test set have ~~similar~~ same distributions

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



100
1000 samples
10 000

+ 1000 000 samples
↳ modern deep learning
↳ large data sets

Test set: after finished developing a system, it evaluates how good the performance is

The size of test set has to be big enough to give high confidence in the overall performance of the system :

≈ 10 000 samples → less than 30% of all data
≈ 100 000

If it's not required a high confidence, test set could be omitted

Cat data set examples → shows pics of cats

- Metric : classification error ⇐ target
  - Algorithm A : 3% error $\xrightarrow{\text{but}}$ shows porno content as error sometimes
  - Algorithm B : 5% error

Metric + Dev ⇒ prefers A
You + Users ⇒ prefers B
} solution ? → ponderation of error's weights
↳ ↑↑ penalization (weight) on porn pics

$$\varepsilon = \frac{1}{\sum w^{(i)}} \sum_{i=1}^{m_{dev}} w^{(i)} \mathbb{1}\{\underbrace{y_{pred}^{(i)}}_{\text{predicted value (0/1)}} \neq y^{(i)}\}$$

$$w^{(i)} = \begin{cases} 1 & \text{if } x^{(i)} \text{ is non-porn} \\ 10 \text{ (or bigger)} & \text{if } x^{(i)} \text{ is porn} \end{cases}$$

Orthogonalization

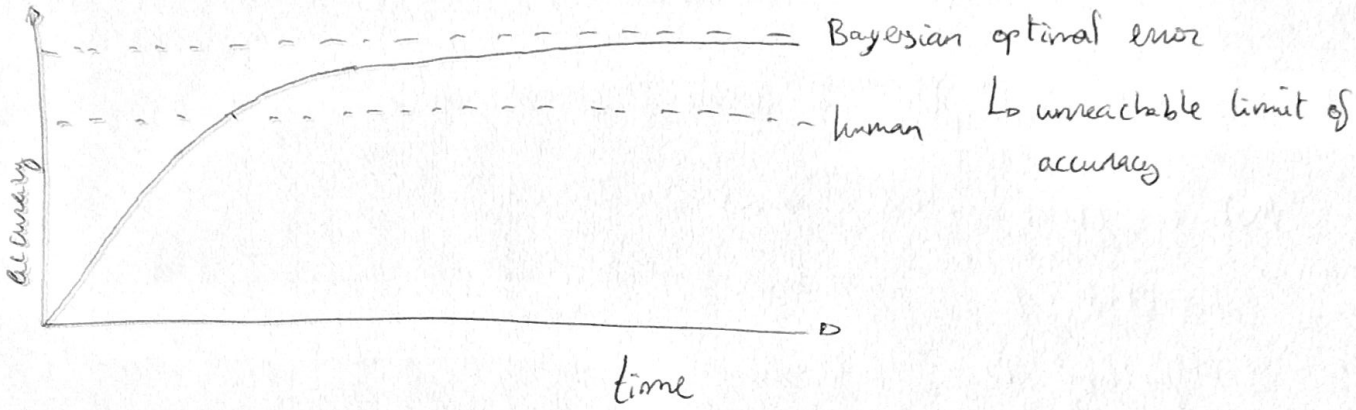
1° So far we've only discussed how to define a metric to evaluate classifiers
↳ Place target

2° Worry separately about how to do well on this metric
↳ How to shoot at the target

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

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

Comparing to human-level performance



Humans are quite good at a lot of tasks. So as long as Machine Learning is worse than humans, you can:

- Get labeled data from humans
- Gain insight from manual error analysis:
  Why did a person get this right?
- Better analysis of bias / variance

You can't do better than Bayes error unless you are overfitting

Humans (≈ Bayes)	1%	7.5%	Avoidable bias
Training error	8% (7%)	8% (0.5%)	
Dev error	10% (2%)	10% (2%)	Variance
	focus on bias	focus on variance	

Medical image classification example

Suppose:

a) Typical human : 3% error

b) Typical doctor : 1% error

c) Experienced doctor : 0.7% error

d) Team of experienced doctors : 0.5% error

↑ Bayesian error $\leq$ 0.5% error

What if training error and dev error are lower than team of experienced doctors error? → Surpassing human-level performance

Not clear then whether focus on reducing bias or variance

- Structured data (easier for computers than for humans)
  - Online advertising
  - Product recommendations
  - Logistics (predicting transit times)
  - Loan approvals
- Unstructured data (natural perception tasks easier for humans than for computers)
  - Speech recognition
  - Image recognition
  - Medical diagnostics

Reducing (avoidable) bias and variance

Human-level

↕ Avoidable bias →
- Decrease regularization
- Train bigger model
- Train longer / better optimization algorithms
  - Momentum, RMS prop, Adam
- NN architecture / hyperparameters search
  - RNN, CNN

Training error

↕ Variance →
- More data
- Regularization
  - L_2, drop out, data augmentation
- NN architecture / hyperparameters search

Dev error

Having more evaluation metrics makes it harder for you to quickly choose between two different algorithms, and will slow down the speed with which your team can iterate

Adding extra labeled data which have different distribution than the original data to the training set won't cause the training and dev/test set distributions become different. So it won't hurt dev and test set performance.

Adding that extra data to the test set:

- Does not longer reflect the distribution of data you care about
- Would cause the dev and test set distributions to become different.

Not good idea because you're not aiming where you want to hit.

A learning algorithm's performance can be better than human-level performance but it can never be better than Bayes error

Human-level performance	0.1 %
Training set error	2 %
Dev set error	2.1 %
Test set error	7 %

↕ solution? / meaning?

- Dev set is overfitted
- Bigger dev set → ↓ dev set error

If training and dev set performance is > human-level performance

- It's harder to measure avoidable bias, thus progress will be slower going forward
- If the test set is big enough for the 0.05% error estimate to be accurate, this implies Bayes error is ≤ 0.05%

Having already an accurate working model which recognises birds, your team is asked now to build one for cats. For the new case 100,000,000 pics (huge amount of data) are available and takes about 2 weeks to train the model:

- Needing two weeks to train will limit the speed at which your team can iterate
- You might be better just training the model with 10,000,000 examples to ~~gain~~ save ≈ 10 times your time, even if each model performs a bit worse because it's trained on less data
- Buying faster computers could speed up your team's iteration speed and thus your team's productivity

For the bird recognition already built model, is required an update a new species of birds appeared: There are now available 1000 pics of the new samples:

- Use the data you have to define a new evaluation metric (using a new dev/test set) taking into account the new species and use that to drive further progress for your team

Despite having higher accuracy than other models, it also gives more false positives so the solution would be:

- Rethink the appropiate metric for this task, and ask your team to tune to the new metric

Week 2

Cat classifier with 90% accuracy (10% error) is performing much worse than expected → misclassifying because it recognises some dogs as cats

- Should you try to make the cat classifier do better on dogs?

Answer: error analysis

- Get 100 mislabeled dev set examples
- Examine them manually

A)) 5% of the examples got ~~wrong~~ are dogs

if you spend time on dog problem → the error will decrease $\underbrace{0.05}_{5\%} \cdot \underbrace{0.1}_{10\%} = \underbrace{0.005}_{0.5\%}$

10% − 0.5% = 9.5% error result

50% of the examples got wrong are dogs

" " " → the error will decrease $\underbrace{0.5}_{50\%} \cdot \underbrace{0.1}_{10\%} = \underbrace{0.05}_{5\%}$

10% − 5% = 5% error result

Evaluate multiple ideas in parallel

- Ideas for cat detection
  - Fix pics of dog being recognised as cats
  - Fix great cats (lions, panthers, ...) being misrecognised
  - Improved performance on blurry pics
  - Fix "Instagram" filters on pics

On a spreadsheet, pick the cathegories with higher proportions →

Image	Dog	Great cats	Blurry	Instagram filters	Comments
1	✓			✓	Pitbull
2			✓	✓	
3		✓	✓		Rainy day at zoo
⋮	⋮	⋮	⋮	⋮	⋮
% of total	8%	43%	61%	12%	

→ focus on the higher category or work on both of them if you have teams

What happens if output labels Y are incorrectly labeled?

Deep learning algorithms are quite robust to <u>random</u> errors in the training set. If the errors are systematic, then it's a problem. Sometimes, if the training data is large and the labeler accidentally labeled some examples wrong, there is no problem if we don't fix them as long as the percentage of errors is low in the training set.

Image	Dog	Great cats	Blurry	Incorrectly labeled	Comments
...					
98				✓	Labeled missed cat in background
99		✓			
100				✓	Drawing of a cat Not a real cat
% of total	8%	43%	61%	6%	

Overall dev set error	10%	2%
Errors due incorrect labels	0.1 · 6 = 0.6%	0.6% → 30% of the 2% error
" " other causes	9.4%	1.4% → here it would be worthwhile

In this case, it does not worth to fix the incorrect labels

Goal of dev set is to help you select between two classifiers A and B

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 algorithms got right as well as ones it got wrong (2%) (98%)
- Train and dev/test data may now come from slightly different distributions

Building a brand new machine learning, some steps are recommended:

- Set up dev/test set and metric
- Build initial system quickly
  - ↳ so that we can localize bias/variance
- Use bias/variance analysis and error analysis to prioritize next steps

Build your first system quickly and then iterate

↳ even though if it's quick and dirty at the beginning

What if training and test distributions differ from each other?

- Cat recognizer example

Data from web sites (higher quality pic) ≈ 200,000

Dat from mobile app ≈ 10,000 ← you care about this

Option 1: Merge distributions (not recommended)

210,000

↓ shuffle

train	dev	test
98% = 205,000	1% 2,500	1% 2,500

200K / 210K

2,500 → 2381 web, 119 mobile

↑ bias (sesgo)

Option 2 : aim the prior distribution in dev and test set

OR

web and mobile	mobile	mobile
train $\underbrace{200{,}000 + 5{,}000}_{205{,}000}$	2,500 dev	2,500 test

despite making different distributions of train and dev / test sets → better performance over long term

or

200,000	5,000	5,000

Estimating the bias and variance in the learning algorithm helps prioritize what to work on next

Cat classifier example

- Assume humans get ≈ 0% error : Bayes error
- Training error 1%
- Dev error 10%

a) The algorithm saw data in training set only

b) The distribution of data in the dev set is different

train	train-dev	dev	test

Training error	1%	variance	1%
Training - dev error	9%		1.5%
Dev error	10%		10%

data mismatch → different distributions

Human error	0%	avoidable bias	0%	avoidable bias
Training error	10%		10%	
Training - dev error	11%		11%	data mismatch problem
Dev error	12%		20%	

	Human level	4%	↕ avoidable bias
Training distribution	Training ~~dev set~~ set	7%	↕ variance
	Training - dev set	10%	↕ data mismatch
Dev/test distribution	Dev set	12%	↕ degree of overfitting to the dev set
	Test set	12%	

Rear mirror recognisers

↓ added distribution

General speech recognition

Rear mirror speech recognition

6% (human labeled)

Human level

Error on examples trained on

Error on examples NOT trained on

"Human level" 4%

↕ Avoidable bias

"Training error" 7%

↕ Variance

"Training dev-set error" 10%

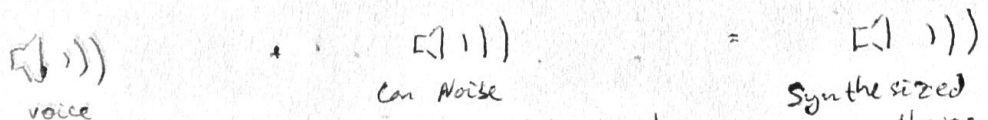
"Dev / Test error" 6%

↔ data mismatch

subset

Addressing data mismatch

- Carry out manual error analysis to try to understand difference between training and dev/test sets
- Make training data more similar, or collect more data similar to dev/test sets
- Artificial data synthesis → problem is covering just a portion of the set of all possible → subset → overfitting



Human voice

"The quick brown fox jumps over a lazy dog" → contains all alphabet

10,000 h

Car Noise

1 h → looped to 10,000 h

Risk

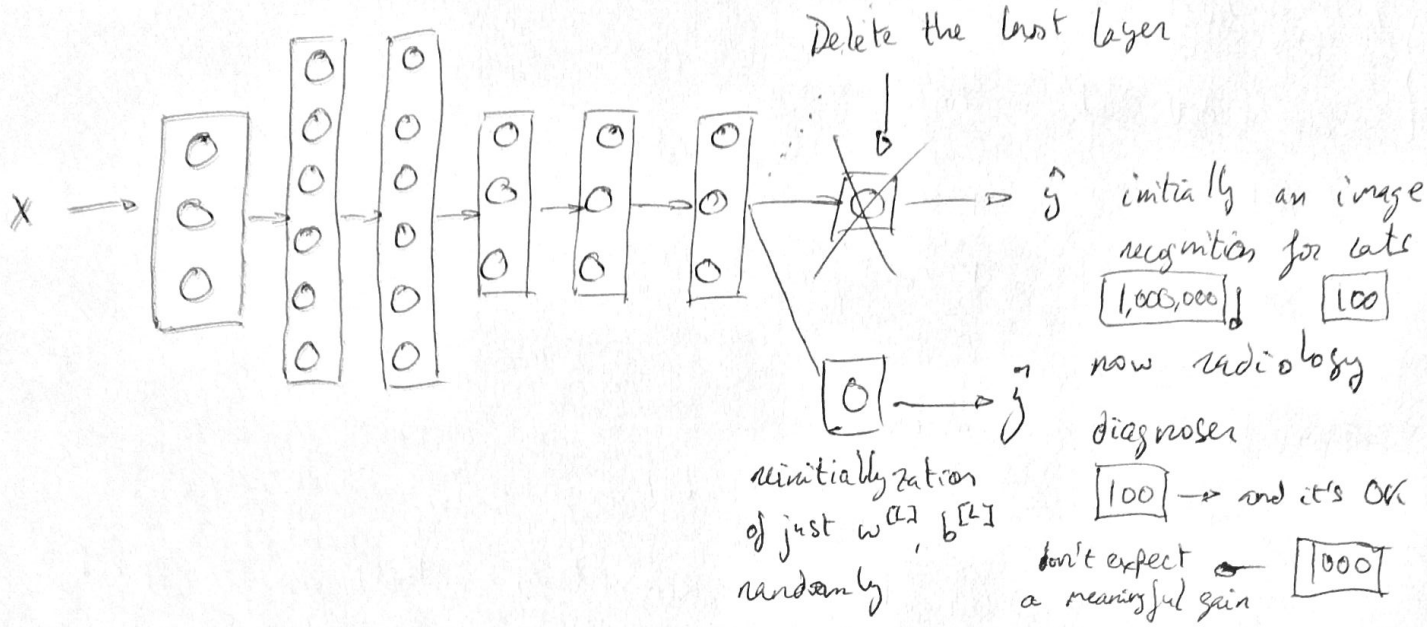
↳ Overfit to 1 h of car noise

Synthesized in-car audio

synthesize

set of all in-car audio

Transfer learning



fine tuning : using image recognition data to preinitialize parameters of all layers and then update them using radiology diagnoses

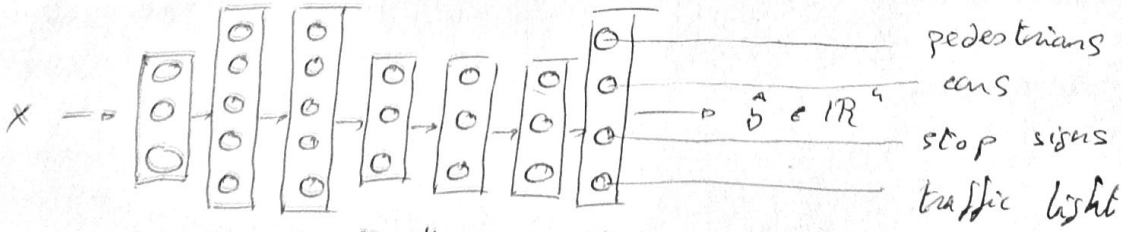
It's helpful because lots of the low level features such as detecting edges, curves, positive objects... are compatible for both NN

↑↑ performance

- Task A and B have the same input x
- You have a lot more data for Task A than Task B
- Low level features from A could be helpful for learning B

Multitask learning

Autonomous driving car with AI



Loss : $\hat{y}^{(i)}_{(4,1)}$, $\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{4} L(\hat{y}_j^{(i)}, y_j^{(i)})$ usual logistic ~~log~~ loss

$-y_j^{(i)} \log \hat{y}_j^{(i)} - (1-y_j^{(i)}) \log (1-\hat{y}_j^{(i)})$

Online (Softmax regression : One example, One label), multitask learning one image can have multiple labels

Alternative: Train four separate NN instead of one to do 4 thing

IS some of the earlier features in NN can be shared between these different types of objects → 1 NN (multitasker) performs better

It does not matter that the labeled just labeled partially Y, the training set works well though.

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

Multitask learning makes sense when:

- Training on a set of tasks that could benefit from having shared lower-level features
- Usually: Amount of data you have for each task is quite similar
- Can train a big enough NN to do well on all the tasks

Transfer learning is usually more used than multitask NN

End-to-end deep learning: Agrupation of multiple stages of processing in a single NN

Speech recognition example

traditional pipeline approach (A) { X audio $\xrightarrow{\text{MFCC}}$ features $\xrightarrow{\text{ML}}$ Phonemes → words → Y transcript

end to end DL (B) { audio → transcript

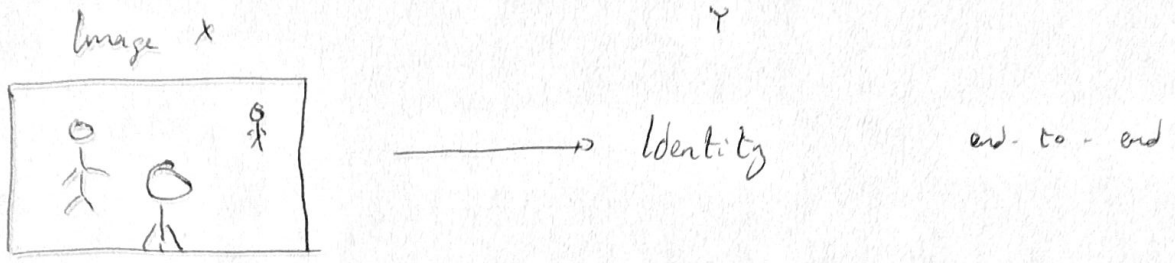
(A) With 3,000 h of training set is enought to work well

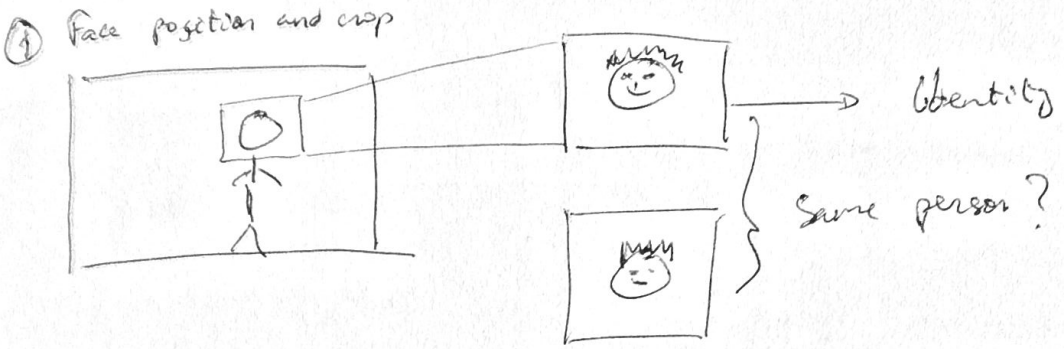
(B) More than + 10,000 h (larger data sets) are required to work just as well

hybrid (C) audio → Phonemes → transcript

End to end does not really work best in practice in some cases such as:

Face recognition





If there is not big data, multistep approach is much better than end-to-end approach

Pros and cons of end-to-end deep learning

Pros:

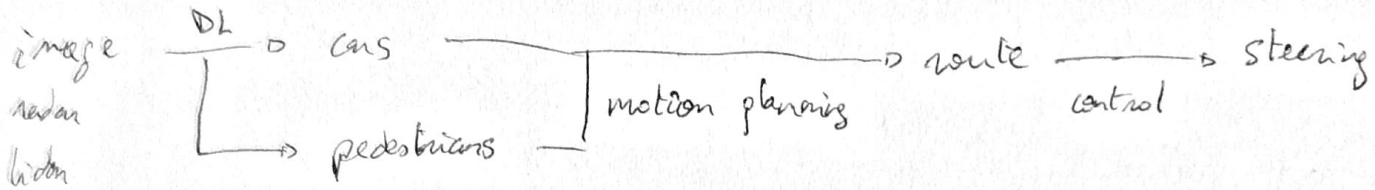
- Let the data speak: Not to submit the NN to human preconceptions that could actually not be natural
  e.g.: phonemes
- Less hand-designing of components needed

Cons:

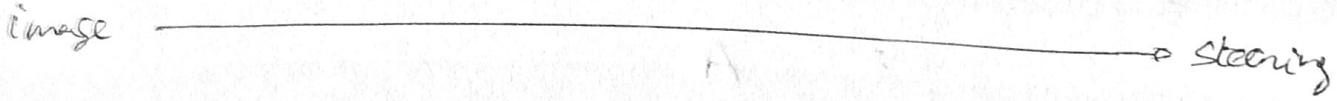
- May need large amount of data
- Excludes potentially useful hand-designed components
  ↳ can help or hurt the system

Applying end-to-end deep learning

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



- Use DL to learn individual components
- Carefully choose X -> Y depending what tasks you can get data for



You are starting on complex machine learning project for autonomous cars. What is the first thing you do?

- Spend a few days training a basic model and see what mistakes it makes as applied ML is a highly iterative process. This will show you to carry out an initial analysis error so it will help point you in more promising directions

Your goal is to detect different features in images. You plan to use a deep NN with ReLU units in the hidden layers.

For the output layer, a softmax activation wouln't be a good choice for the output layer as this is a multitask learning problem. It'd be OK if only one of the possibilities was present in each image.

You are carrying out error analysis and counting up what errors the algorithm makes. The datasets that should be manually and carefully examined are a portion of the images on which the algorithm made a mistake ($\approx 5\%$)

After working on the data for several weeks about a multitask learning problem there is no need to have all $y^{(i)}$ vectors fully labeled. As the cost wouldn't be influenced by the unlabeled entries.

The distribution of data you care about (A) contains less images but larger (B) than another one which is different. The dataset should be split as:

$$\text{Training set} \begin{cases} 100\% \text{ of } B \\ 80\% \text{ of } A \end{cases} \quad \text{Dev set } \{10\% \text{ of } A \quad \text{Test set } \{10\% \text{ of } A$$

Dev and test set should have the closest possible distribution to "real" data. It's also important for the training set to contain enough "real" data to avoid having a data-mismatch problem

Dataset	Error of the algorithm	Contains
Training	8.8%	940 000 images randomly picked from 900000 B + 60000 A
Training Dev	9.1%	20.000 images randomly picked from 900.000 B + 60.000 A
Dev	14.3%	20.000 images from A
Test	14.8%	20.000 images from A

Human-level error is 0,5%

- You have a large data-mismatch problem as your model does a lot better on the training-dev set than on the dev set
- You have a large avoidable-bias problem because your training error is quite a higher than the Bayes error

The algorithm does better on the distribution of data it trained on. But you don't know if it's because on that distribution or if it's really easier. To get a better sense, measure human-level error separately on both distributions.

~~You decide to buy a device that~~

After working further on a problem, you've decided to correct the incorrectly labeled data on the dev set. But also on the test set so that they remain in the same distribution.

On the other hand you should NOT correct them in the training set as well so as to avoid it now being even more different from the dev set

In many fields, end-to-end learning works better in practice but requires a large amount of data

You decid to use data augmentation to address foggy images. You find 1000 of fog off the internet, and add them to clean images to synthesize foggy days:

As long as the synthesized fog looks realistic to the human eye, you can be confident the synthesized data is accurately capturing the distribution of real foggy images, since human vision is very accurate for the problem you're solving

Bring more foggy pics into the training set so as to address it's error in that category. False because this would depend on how easy it is to add this data and how much you think your team thinks it'll help.

Convolutional Neural Networks

Earlier layers detect edges, later layers detect objects and deeper layers detect complex objects

Convolution

vertical edge

$$\underbrace{\begin{bmatrix} 3 & 0 & 1 & 2 & 7 & 4 \\ 1 & 5 & 8 & 9 & 3 & 1 \\ 2 & 7 & 2 & 5 & 1 & 3 \\ 0 & 1 & 3 & 1 & 7 & 8 \\ 4 & 2 & 1 & 6 & 2 & 8 \\ 2 & 4 & 5 & 2 & 3 & 9 \end{bmatrix}}_{6\times 6\ (n\times n)} * \underbrace{\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}}_{3\times 3\ (f\times f)} = \underbrace{\begin{bmatrix} -5 & -4 & 0 & 8 \\ -10 & -2 & 2 & 3 \\ 0 & -2 & -4 & -7 \\ -3 & -2 & -3 & -16 \end{bmatrix}}_{4\times 4\ (\gamma\times\gamma)}$$

filter

$\hookrightarrow \gamma = n - f + 1$

$\hookrightarrow$ element wise multiplication

python: conv_forward tensorflow: tf.nn.conv2d Keras: conv2D

The output size shrinks and first and last rows and columns information is thrown away

$$\underbrace{\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}}_{\text{sobel filter}} \qquad \underbrace{\begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix}}_{\text{Scharr filter}} \qquad \underbrace{\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}}_{\text{filter for horizontal edges}} \qquad \underbrace{\begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{bmatrix}}_{\text{filter to be learned by training for:}}$$

$\hookrightarrow$ more weight in the central row

$\hookrightarrow$ $-45°$, $-70°$, ...

Padding

Rows and columns are added to the limits of the matrix input to prevent the output from shrinking. Less information is thrown away

$$\begin{bmatrix} 0 & 0 & 0 & \cdots & & & & & \\ 0 & 0 & 0 & 0 & \cdots & & & & \\ 0 & 0 & 3 & 0 & 1 & 2 & 7 & 4 & \\ \vdots & \vdots & 1 & 5 & 8 & 9 & 3 & 1 & \\ & & 2 & 7 & 2 & 5 & 1 & 3 & \\ & & 0 & 1 & 3 & 1 & 7 & 8 & \\ & & 4 & 2 & 1 & 6 & 2 & 8 & \\ & & 2 & 4 & 5 & 2 & 3 & 9 & \end{bmatrix} \begin{matrix} \} p \\ \} 2p \end{matrix}$$

$\underbrace{p}_{2p}$

Output size: $(n + 2p - f + 1,\ \text{"}\ \text{"})$

Scanned by CamScanner

Valid convolution : no padding $\rightarrow (n \times n) * (f \times f) \rightarrow (n-f+1, \text{''})$

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

$$\cancel{n} + 2p - f + 1 = \cancel{n} \rightarrow p = \frac{f-1}{2}$$ where f is usually odd (impar)

strided convolution

stride = 2

not computer if it covers out the matrix

$$\begin{bmatrix} 2 & 3 & 7 & 4 & 6 & 2 & 9 \\ 6 & 6 & 9 & 8 & 7 & 4 & 3 \\ 3 & 4 & 8 & 3 & 8 & 9 & 7 \\ 7 & 8 & 3 & 6 & 6 & 3 & 4 \\ 4 & 2 & 1 & 8 & 3 & 4 & 6 \\ 3 & 2 & 4 & 1 & 9 & 8 & 3 \\ 0 & 1 & 3 & 9 & 2 & 1 & 4 \end{bmatrix} * \begin{bmatrix} 3 & 4 & 4 \\ 1 & 0 & 2 \\ -1 & 0 & 3 \end{bmatrix} = \begin{bmatrix} 91 & 100 & 83 \\ 69 & 91 & 127 \\ 44 & 72 & 74 \end{bmatrix}$$

(3×3)

$$\gamma = \frac{n + 2p - f}{s} + 1$$

Cross correlation

$$* \begin{bmatrix} 3 & 4 & 5 \\ 1 & 0 & 2 \\ -1 & 9 & 7 \end{bmatrix} =$$

the filter matrix is "minored"

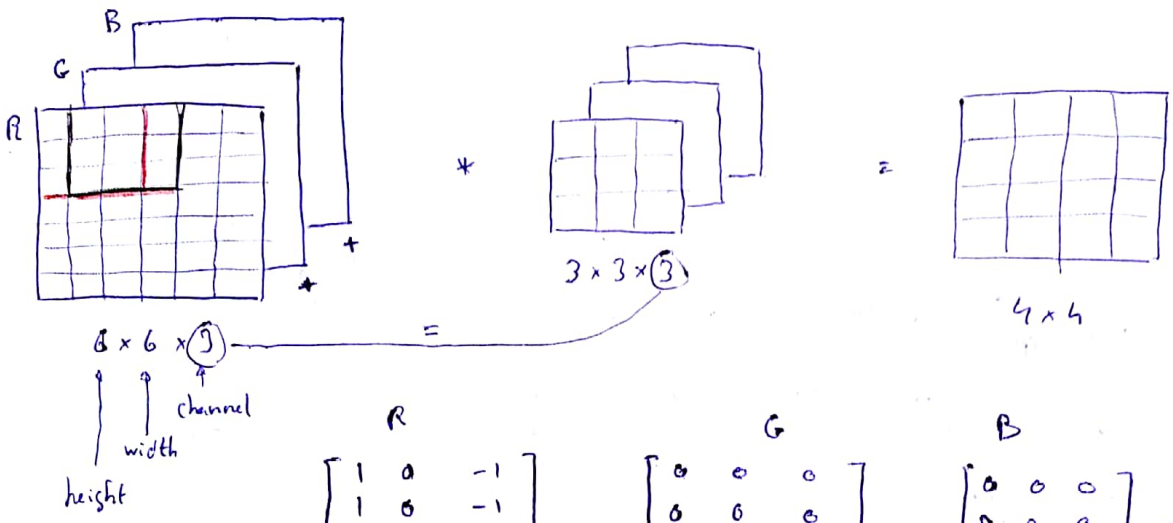
$$* \begin{bmatrix} 7 & 2 & 5 \\ 9 & 0 & 4 \\ -1 & 1 & 3 \end{bmatrix}$$

$(A * B) * C = A * (B * C) \rightarrow$ associative property:

- $\rightarrow$ good for signal processing
- $\rightarrow$ it doesn't matter in neural networks

Scanned by CamScanner

Convolutions on RGB image → Volumes



R

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

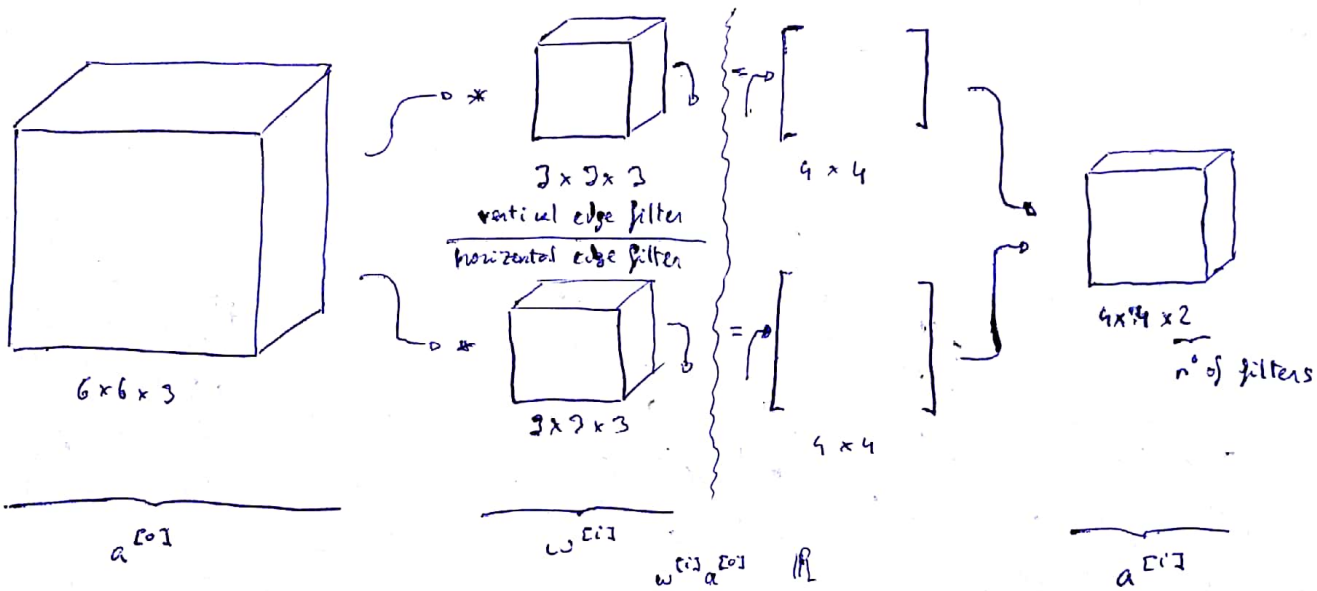
G

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

B

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

→ filter that detects just red vertical edges



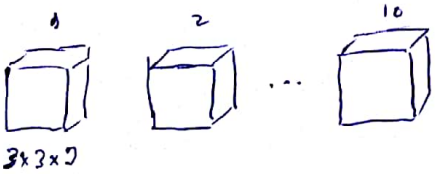
$$z^{[i]} = w^{[i]} a^{[0]} + b^{[i]}$$

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

$$\rightarrow \text{ReLU}\left(\underbrace{\overset{w^{[i]}a^{[0]}}{\begin{bmatrix} \quad \end{bmatrix}}}_{4\times 4} + \overset{\mathbb{R}}{b_1}\right) \rightarrow$$

$$\rightarrow \text{ReLU}\left(\overbrace{\underbrace{\begin{bmatrix} \quad \end{bmatrix}}_{4\times 4} + b_2}^{z^{[i]}}\right) \rightarrow$$

If you have 10 filters of 3×3×3 in one layer of a NN, how many parameters does that layer have?



27 parameters
+ bias

28 parameters · 10 = $\boxed{280}$

Scanned by CamScanner

If layer l is a convolutional 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: $n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$

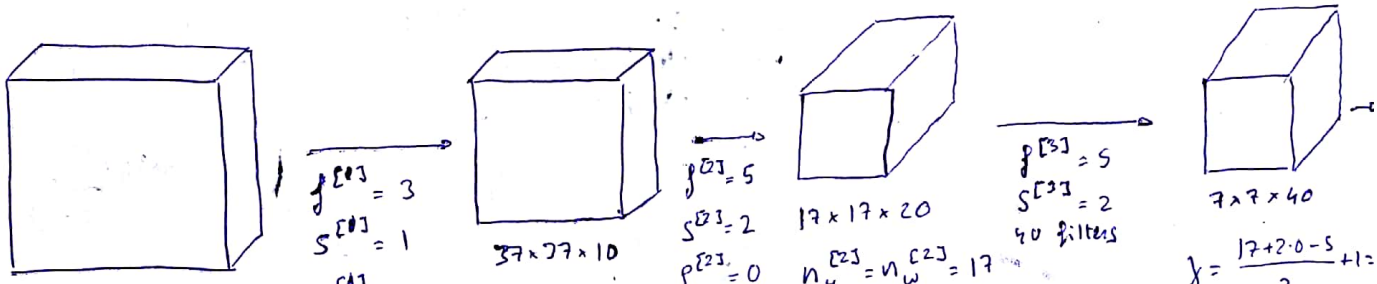
$$n_{H,W}^{[l]} = \frac{n_{H,W}^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1$$

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

Activations: $a^{[l]} \to n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$ Mini-batch descent: $m \times n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$

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

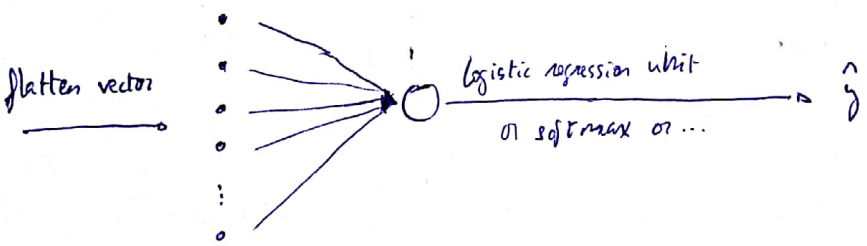
Bias: $n_c^{[l]} - (1, 1, 1, n_c^{[l]})$



$$\gamma = \frac{n + 2p - f}{s} + 1 = \frac{39 + 2 \cdot 0 - 3}{1} + 1 = 37$$

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

$$\gamma = \frac{17 + 2 \cdot 0 - 5}{2} + 1 = 7$$



(1960)

Scanned by CamScanner

Pooling layers

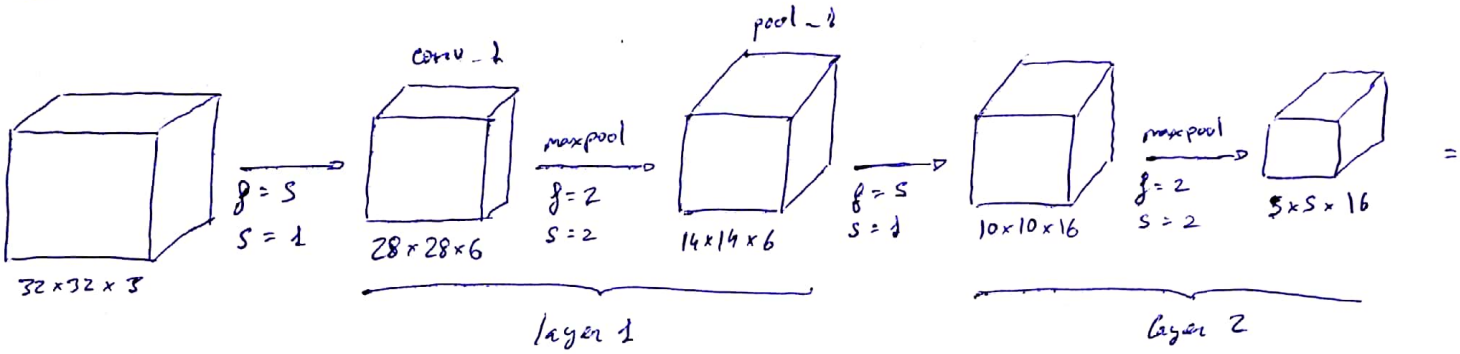
- Max pooling [does not require learning parameters]

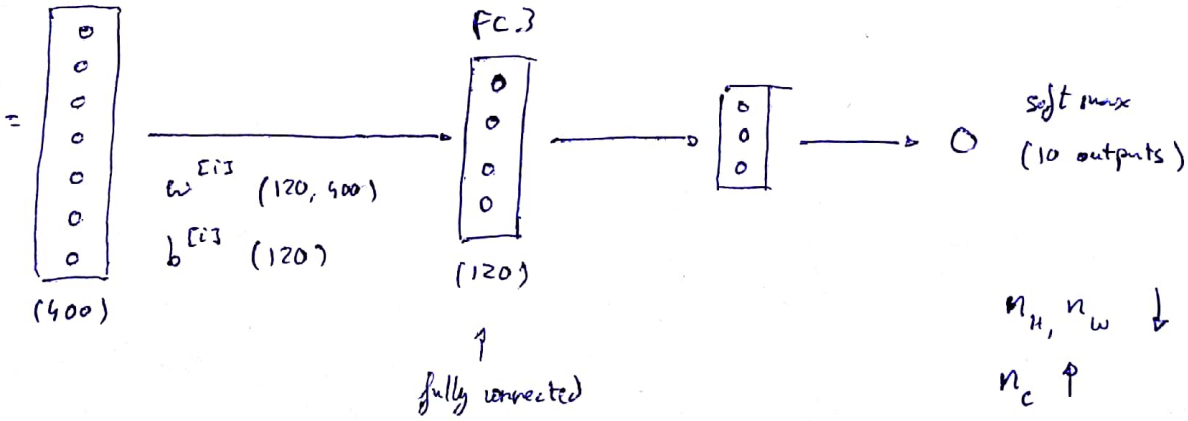
$$\begin{bmatrix} 1 & 3 & 2 & 1 \\ 2 & 9 & 1 & 1 \\ 1 & 3 & 2 & 3 \\ 5 & 6 & 1 & 2 \end{bmatrix} \xrightarrow[\substack{f=2 \\ s=2}]{\text{hyperparameters}} \begin{bmatrix} 9 & 2 \\ 6 & 3 \end{bmatrix}$$

max of the quadrant

- Average pooling

↳ for this method: padding $p = 0$





$n_H, n_W \downarrow$

$n_C \uparrow$

conv_pool_conv_pool_fc_fc_fc_softmax

Activation size tends to shrink

Input and pools don't have parameters to learn

There are few of them in convs and softmax and most of them in fcs

Why convolutions?

- 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
  ↳ reduces total number of parameters, thus reducing overfitting
- Sparsity of connections: in each layer, each output value depends only on a small number of inputs

Scanned by CamScanner

$$\begin{bmatrix} 0 & 1 & -1 & 0 \\ 1 & 3 & -3 & -1 \\ 1 & 3 & -3 & -1 \\ 0 & 1 & -1 & 0 \end{bmatrix}$$

Suppose your input is a 300 × 300 RGB image, and you are not using a convolutional network. If the hidden layer has 100 neurons, each one fully connected to the input, how many parameters does this hidden layer have including the bias parameters?

$$\underbrace{(300 \times 300 \times 3) \times 100}_{w} + \underbrace{100}_{b} = 27\,000\,100$$

Suppose your input is a 300 by 300 RGB image, and you use a convolutional layer with 100 filters that are each 5×5. How many parameters does this hidden layer have including the bias parameters?

$$\underbrace{5 \times 5 \times 3}_{w} + \underbrace{1}_{b} = 76 \xrightarrow{\times 100} 7600$$

You have an input volume that is 63 × 63 × 16, and convolve it with 32 filters that are 7×7, using a stride of 2 and no padding. What is the output volume?

$$y = \frac{n + \cancel{2p} - f}{s} + 1 = \frac{63 - 7}{2} + 1 = 29 \Rightarrow 29 \times 29 \times 32$$

You have an input volume that is 15×15×8, and pad it using "pad = 2". What is the dimension of the resulting volume (after padding)?

$$\left.\begin{matrix} n = 15 \\ p = 2 \end{matrix}\right\} n' = 15 + 2 \cdot 2 = 19 \longrightarrow 19 \times 19 \times 8$$

You have an input that is 63 × 63 × 16, convolve it with 32 filters that are 7×7, and stride of 1. You want to use a same convolution. What is the padding?

$$y = \frac{\cancel{n} + 2p - f}{\cancel{s}\,1} + 1 = \cancel{n} \rightarrow p = \frac{f-1}{2} = \frac{7-1}{2} = 3$$

You have an input volume that is 32×32×16, and apply max pooling with a stride of 2 and a filter size of 2. What is the output volume? $y = \frac{32 - 2}{2} + 1 = 16$

pooling layers do not have parameters, but they do affect the backpropagation (derivatives calculation)

Scanned by CamScanner

Classic Networks

- LeNet-5
  - recognize hand-written images
  - no padding was applied → valid convolution
  - average pooling layers
  - input image in greyscale → 1 channel

$n_H, n_W \downarrow \qquad n_c \uparrow$

conv - pool - conv - pool - fc - fc - (Softmax)

↑ new version

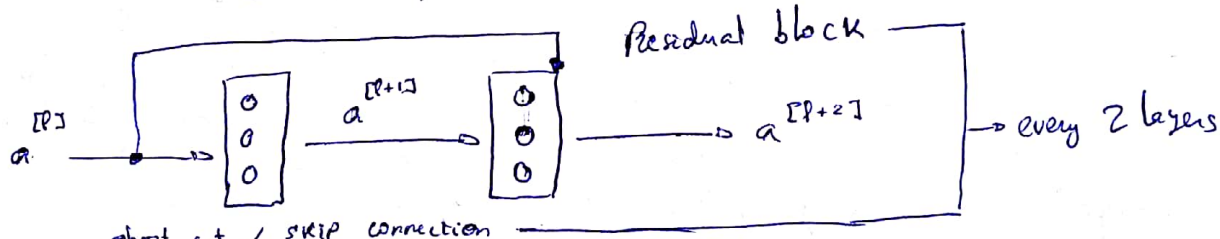
- AlexNet → ReLU non-linearity

conv - maxpool - conv - maxpool - $\overset{\text{same}}{\text{conv}}$ - $\overset{\text{same}}{\text{conv}}$ - $\overset{\text{same}}{\text{conv}}$ - maxpool - fc - fc - softmax

- VGG16 → conv = 3x3 filter, s = 1, same
  max-pool = 2x2, s = 2

Very deep NN are difficult to train because of vanishing and exploding gradient problems

Res Net → enables training very deep networks ($l > 100$)



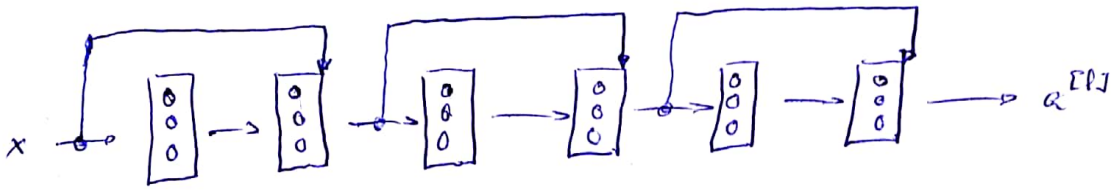
$a^{[l]}$ → linear → ReLU → linear → ReLU → $a^{[l+2]}$

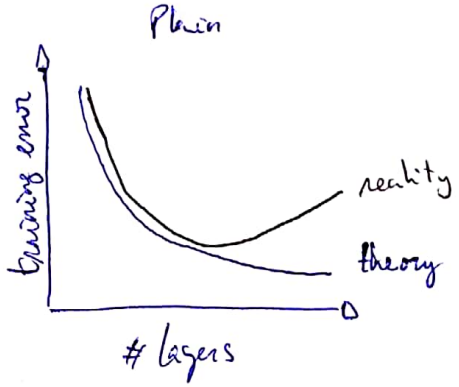
$z^{[l+1]} = W^{[l+1]} a^{[l]} + b^{[l+1]} \rightarrow a^{[l+1]} = g(z^{[l+1]}) \rightarrow z^{[l+2]} = W^{[l+2]} a^{[l+1]} + b^{[l+2]} \rightarrow$

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

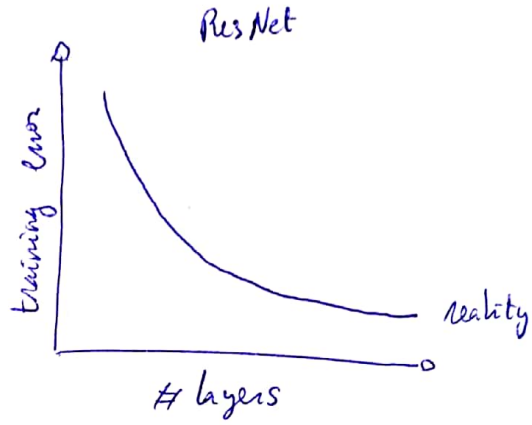
Scanned by CamScanner

Plain network



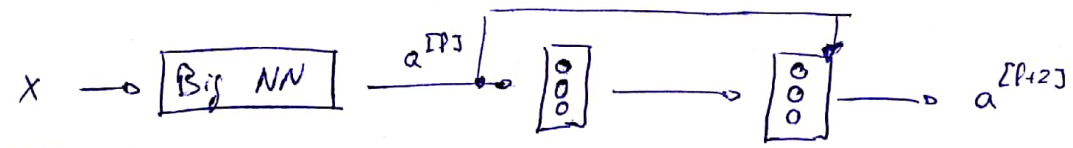


Training set



$X \rightarrow \boxed{\text{Big NN}} \rightarrow a^{[l]}$

Identity function is easy for residual block to learn



ReLU $a \geq 0$

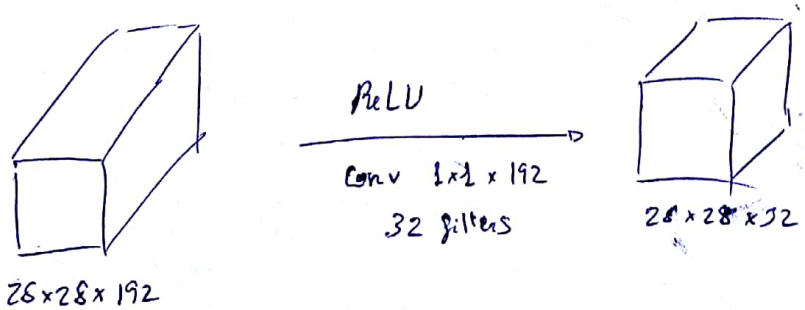
otherwise ...

"same" convolution

$$\underset{(256)}{a^{[l+2]}} = g(z^{[l+2]} + a^{[l]}) = g(\underbrace{w^{[l+2]} a^{[l+1]} + b^{[l+2]}}_{\text{if } w^{[l+2]} = b^{[l+2]} = 0} + \underset{(256\times128)}{(W_s)} \underset{(128)}{a^{[l]}}) = g(a^{[l]})$$

matrix adjustment so dimensions make sense

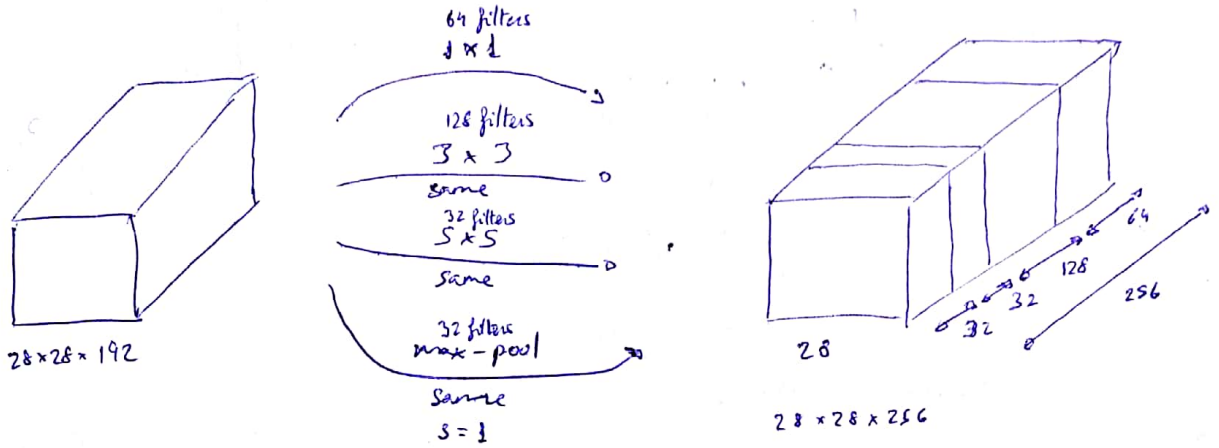
1×1 convolution (Network in Network)
- $f = 1$
- useful to shrink n_c ↓, while n_H and n_W remain the same and adds non-linearity which improves the learning
- helps build up inception network



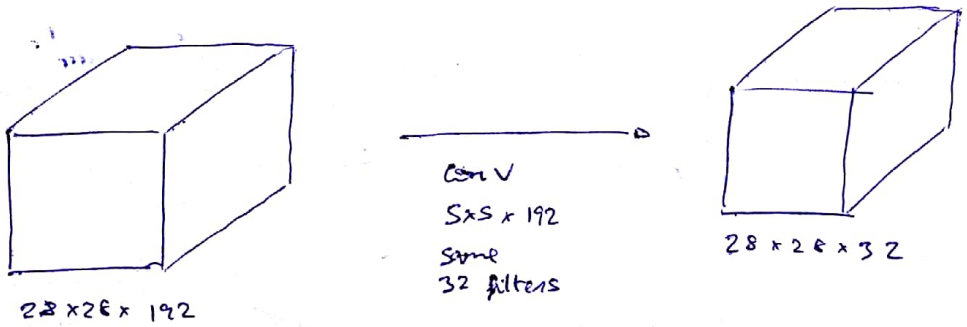
Scanned by CamScanner

Inception network → helps decide hyperparameters and other features

- instead of picking one of these filter sizes or pooling, you can do them all
- computational cost is the problem

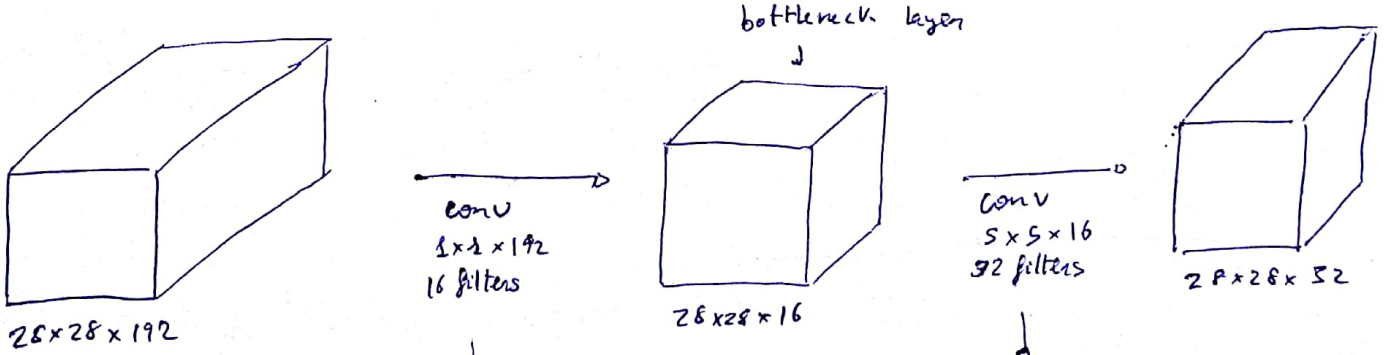


The problem of computational cost



$(28 \times 28 \times 32) \cdot (5 \times 5 \times 192) = 120\ M$ of operations

which is the cost of the convolutional layer

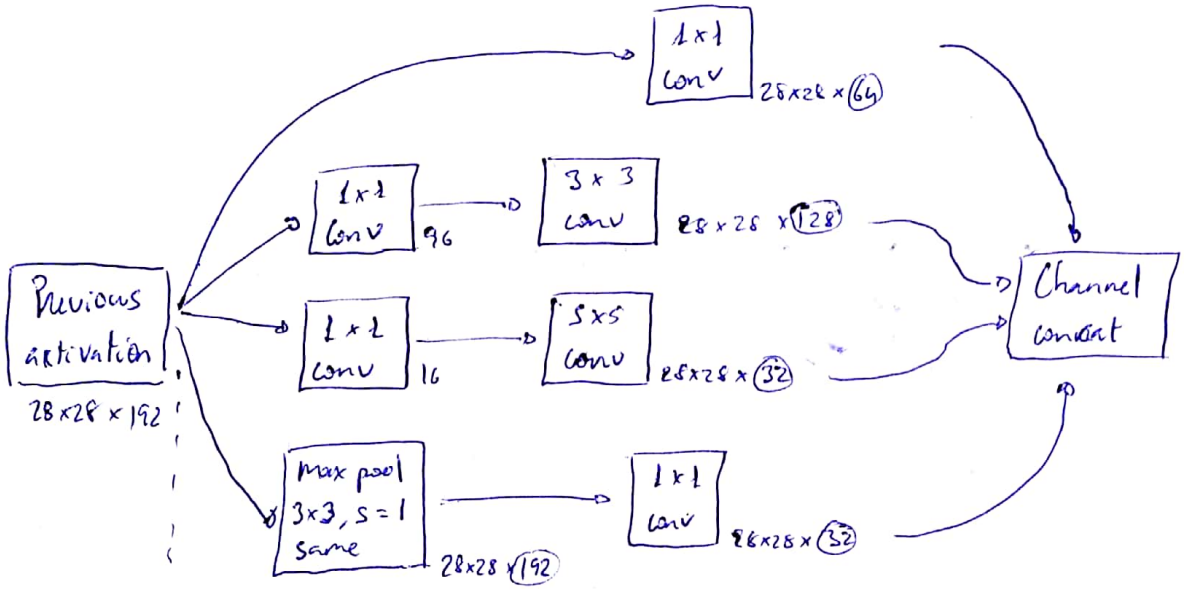


$(28 \times 28 \times 32) \cdot (5 \times 5 \times 16) = 10\ M$

$(28 \times 28 \times 16) \cdot (1 \times 1 \times 192) = 2.4\ M$

$12.4\ M \ll 120\ M$

Scanned by CamScanner



Inception block

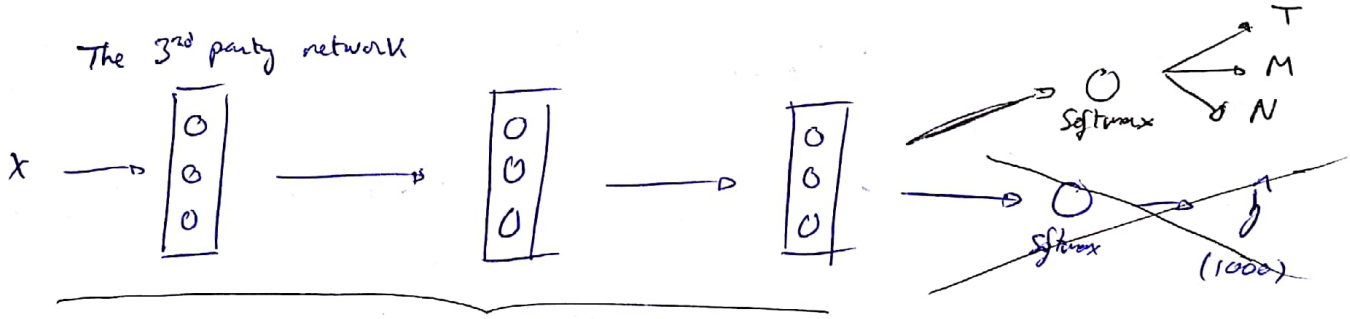
by adding a branch in intermediate inception blocks, helps prevent the network from overfitting

- - - → FC → FC → softmax

Transfer learning → download the weights of another trained network

Our case : cat detector → Tiger / Misty / Neither

The 3rd party network



freeze = trainable Parameter = 0 or freeze = 1

+

train softmax layers weights

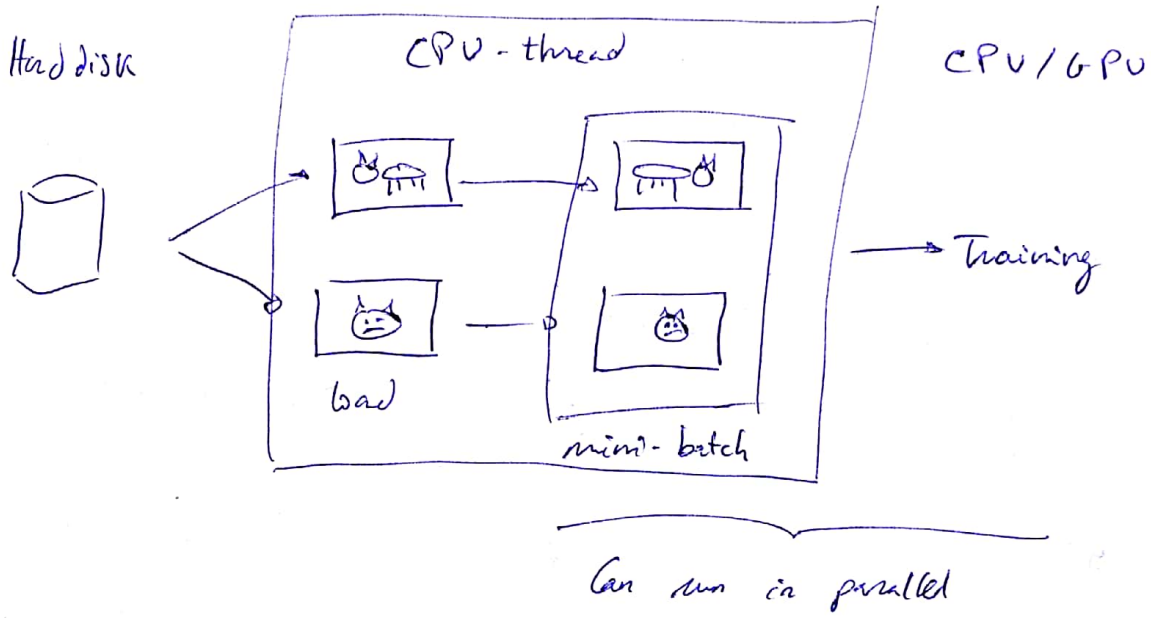
freeze if we have big data of Tiger / Misty / Neither

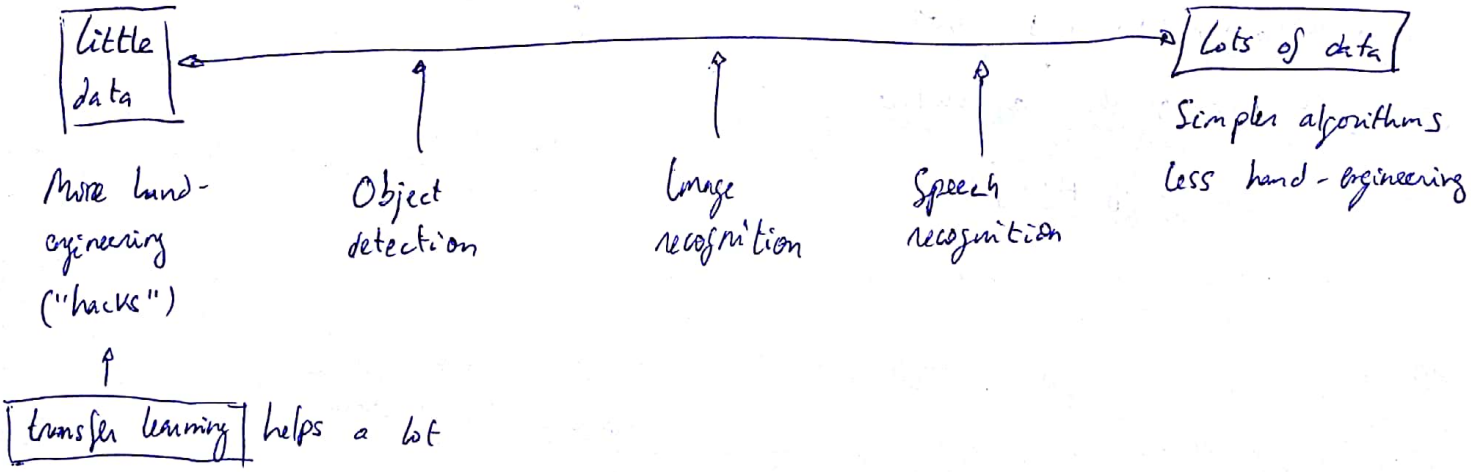
or train everything if we have large big data set

Scanned by CamScanner

Augmentation Data in computer vision

- Methods on pictures
  - Mirroring
  - Random cropping
  - Rotation
  - Shearing
  - Local warping
  - Color shifting → add distortion to any of the RGB channels





Two sources of knowledge:

- Labeled data
- Hand engineered features / network architecture / other components

Scanned by CamScanner

Tips for doing well on benchmarks / winning competitions

- Ensembling

  3-15

  - Train several networks independently (with different random seeds) and average their outputs ? (until 2% of improvement)

- Multicrop at test time

  - Run classifier on multiple versions of test images and average results

  10-crop : $1 + 4 + \underbrace{1 + 4}_{\text{mirrored}}$

Use open source code

- Use architectures of networks published in the literature
- Use open source implementations if possible
- Use pretrained models and fine-tune on your dataset

Suppose you have an input volume of dimension $64 \times 64 \times 16$. How many parameters would a single 1×1 convolutional filter have including bias?

$(1 \times 1 \times 16) + 1 = 17$ $\qquad$ $\lambda = n - \cancel{f} + \cancel{1} = 64$

Which of the following do you typically see as you move to deeper layers in a ConvNet?

n_H and n_W decrease, while n_C increases

Which of the following do you typically see in a ConvNet?

- Multiple CONV layers followed by a POOL layer
- FC layers in the last few layers

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

Scanned by CamScanner

Regarding Residual Networks

- The skip-connection makes it easy for the network to learn an identity mapping between the input and the output within the ResNet block
- Using a skip-connection helps the gradient to backpropagate and thus helps you to train deeper networks

Regarding Inception Networks

- A single inception block allows the network to use a combination of 1x1, 3x3 and 5x5 convolutions and pooling
- Inception blocks usually use 1x1 convolutions to reduce the input data volume's size before applying 3x3 and 5x5 convolutions

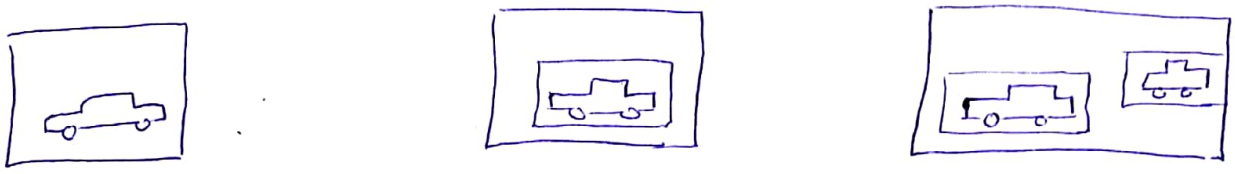
What are the reasons of using open-source implementations of ConvNets (both the model and/or weights)?

- It's a convenient way to get working an implementation of a complex ConvNet architecture
- Parameter trained for one computer vision task are often useful as pretraining for other computer vision tasks

Scanned by CamScanner

Week 3

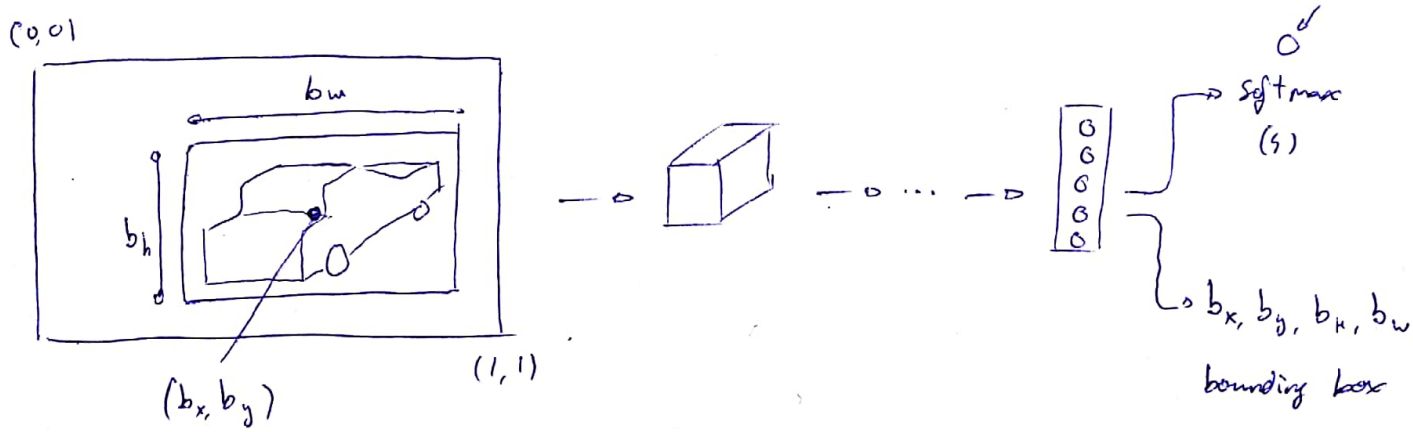
Image classification | Classification with localization | Detection



"car" | "car"

1 object

multiple objects



1 - pedestrian
2 - car
3 - motorcycle
} $P_c = 1$

4 - background } $P_c = 0$

$b_x = 0.6$
$b_y = 0.5$
$b_h = 0.55$
$b_w = 0.5$

If $P_c = 0$ → the rest of parameters remain undefined
↳ "don't care"

Need to output b_x, b_y, b_h, b_w, diff component

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

P_c → is there any object? 0/1

c_1 → pedestrian
c_2 → car
c_3 → motorcycle
} 0/1

$$y = \begin{bmatrix} 0 \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \end{bmatrix}$$

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

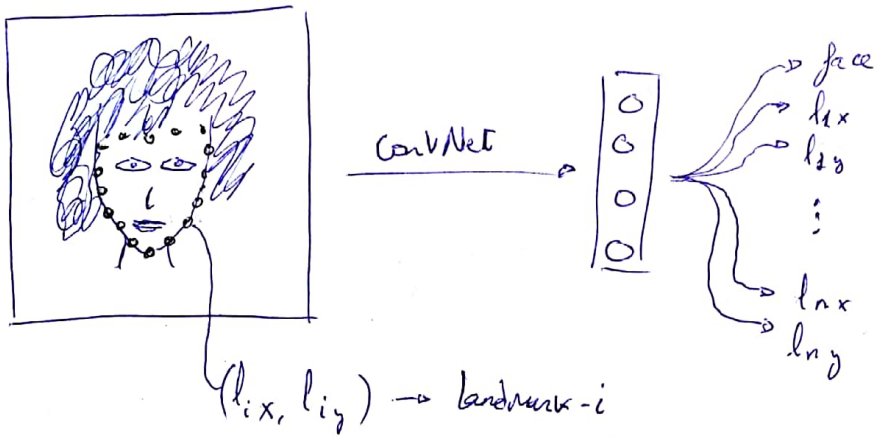
Scanned by CamScanner

Landmark detection

↳ instead of a boundary box, the NN positionates a point in the picture

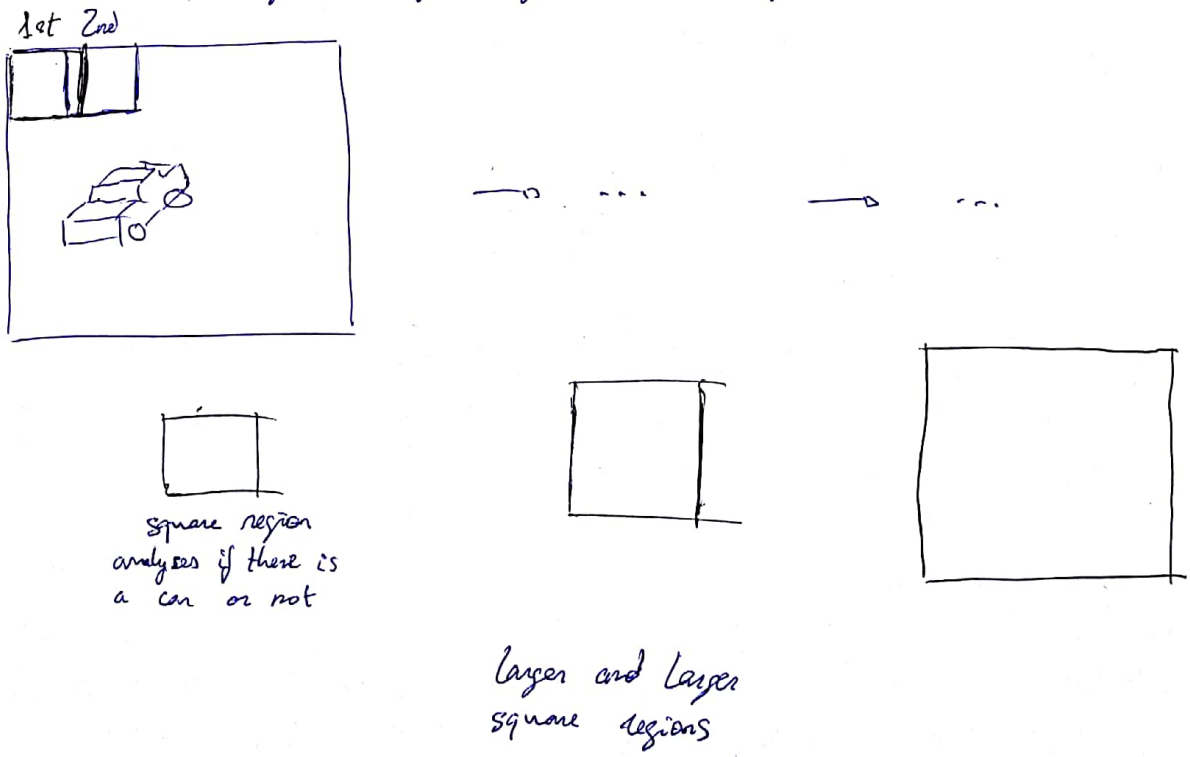
Applications → recognition of emotions in faces

The test set is laborious to get, as someone has to spot manually the dots



Sliding windows detection

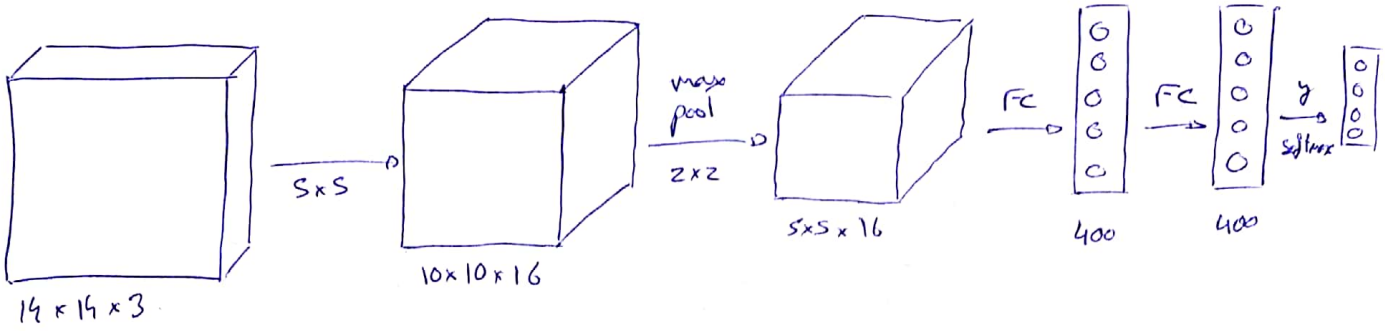
iteration → it goes through every region of the picture with a s stride

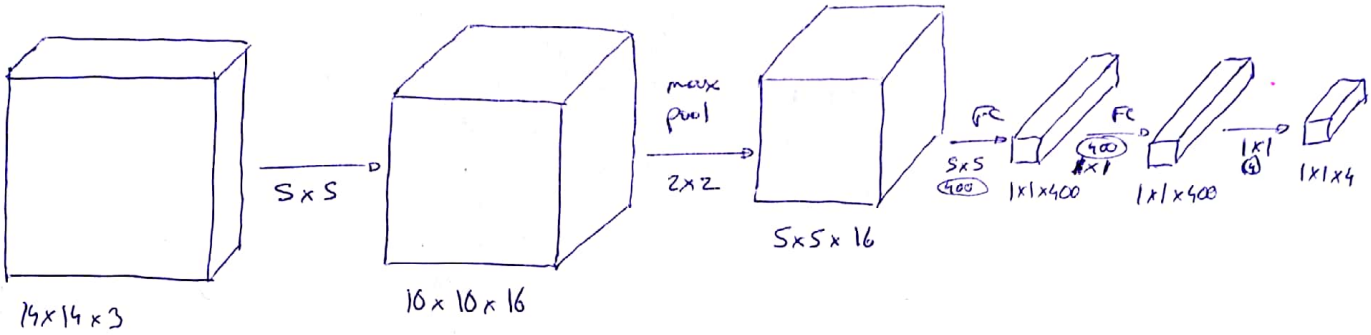


↑↑ computational cost with ConvNets (it was ok back then when there were used linear functions instead)

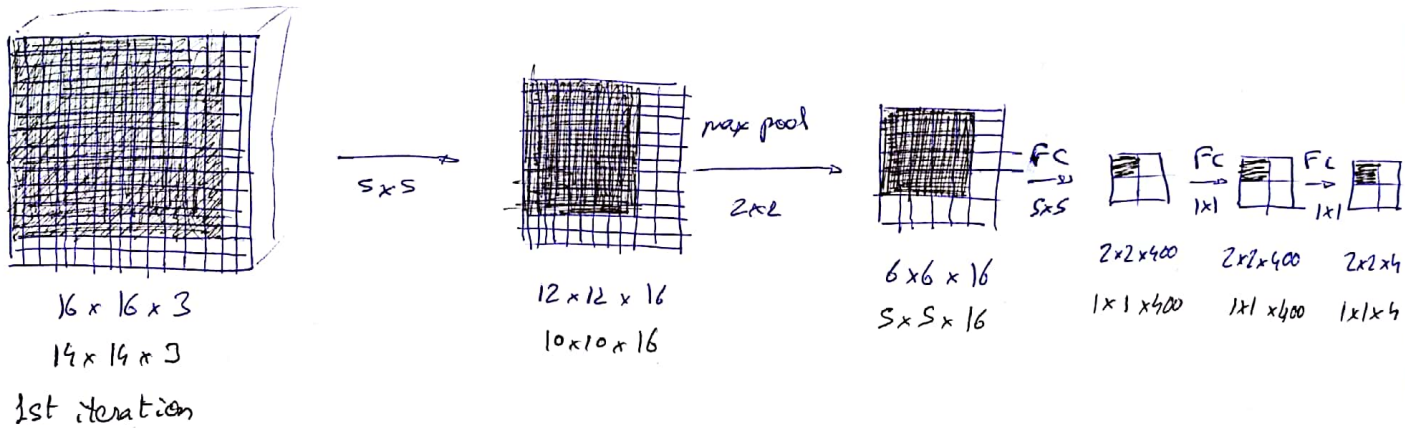
Scanned by CamScanner

Turning FC layer into convolutional layers





Convolutional implementations of sliding windows



Instead of doing it sequentially, with this convolutional implementation we'd do all the predictions at the same time

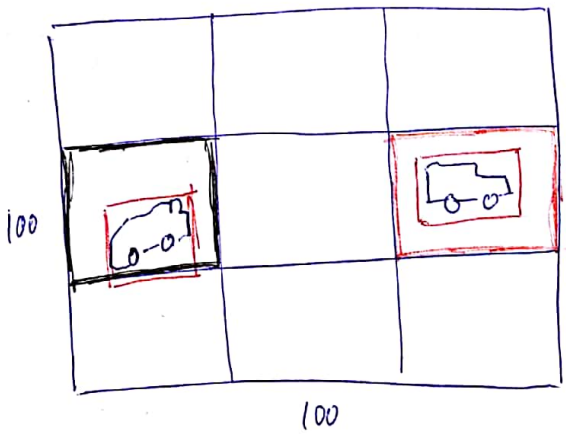
Scanned by CamScanner

Bounding box predictions

- matches the object approximately
- adjusts the aspect ratio (rectangle, square ?)

YOLO → you only look once

The picture is split in grid cells which individually the algorithm is applied



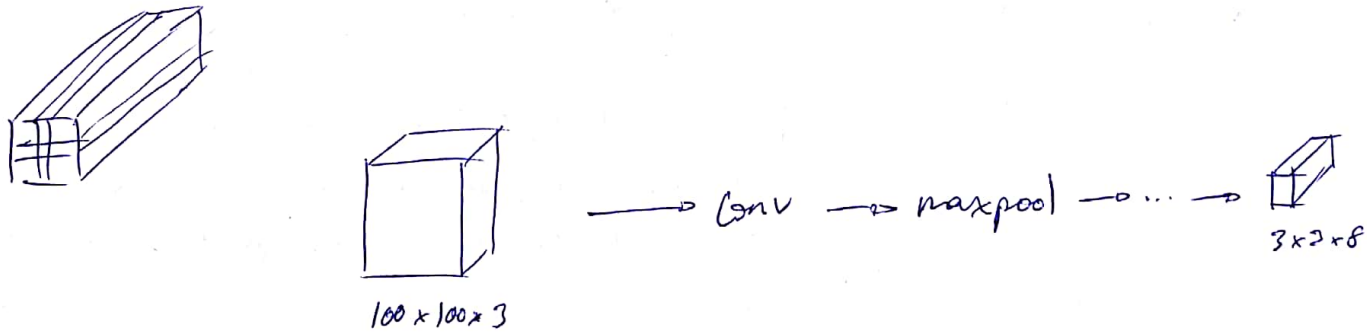
9 grid cells (example)

Labels for training

For each grid cell:

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

Target output : $3 \times 3 \times 8$



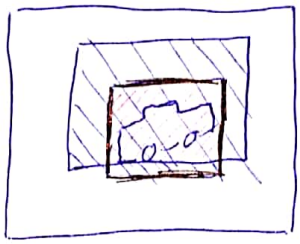
The smaller the grid is, the less likely is to get multiple cars in a grid

$$y = \begin{bmatrix} 1 \\ b_x \\ b_y \\ b_h \\ b_w \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad \begin{matrix} 0,4 \\ 0.3 \\ 0,9 \\ 0.5 \end{matrix} \quad \begin{matrix} \in [0,1] \\ \\ \text{can be} > 1 \end{matrix}$$

Scanned by CamScanner

Intersection ~~Object~~ Over Union → indicator of the accuracy of the boundary box and a measure of overlap between two boundary boxes

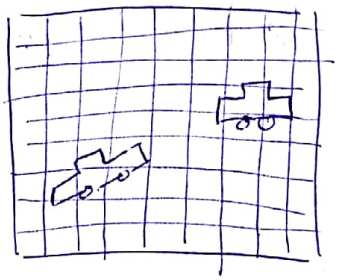
$$IoU = \frac{\text{size of } \square}{\text{size of } \square}$$

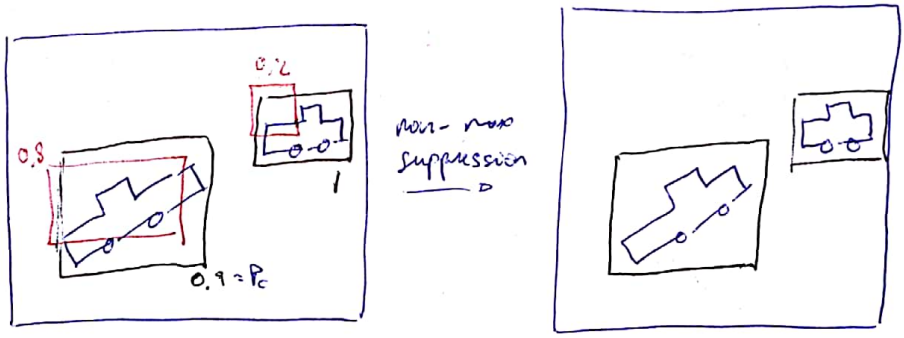


Correct if $IoU \geq \alpha$

Non-max suppression → suppress copies (on cars)

Every grid would detect the car independently, so multiple detections on the same car would appear





Each output prediction is:

$$\begin{bmatrix} P_c \\ b_x \\ b_y \\ b_h \\ b_w \end{bmatrix}$$

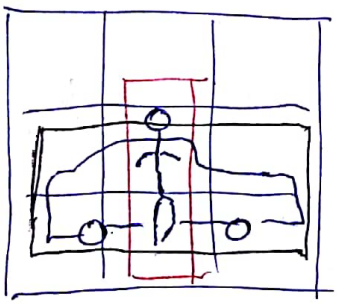
Discard all boxes with $P_c \leq 0.6$

While there are any remaining boxes:

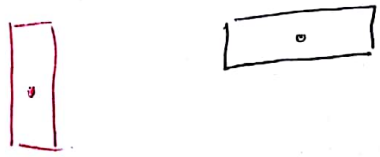
- Pick the box with the largest P_c
  Output that as a prediction
- Discard any remaining box with $IoU \geq 0.5$ with the box output in the previous step

Anchor boxes provide the grids the possibility of detecting different objects

Overlapping objects:



Anchor box 1 Anchor box 2



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

Scanned by CamScanner

Previously

Each object in training image is assigned to grid cell that contains that object's midpoint

Output y (3 x 3 x 8)

The shape of the anchor boxes may be choosen manually by following pattens of the different shapes of a kind of an object

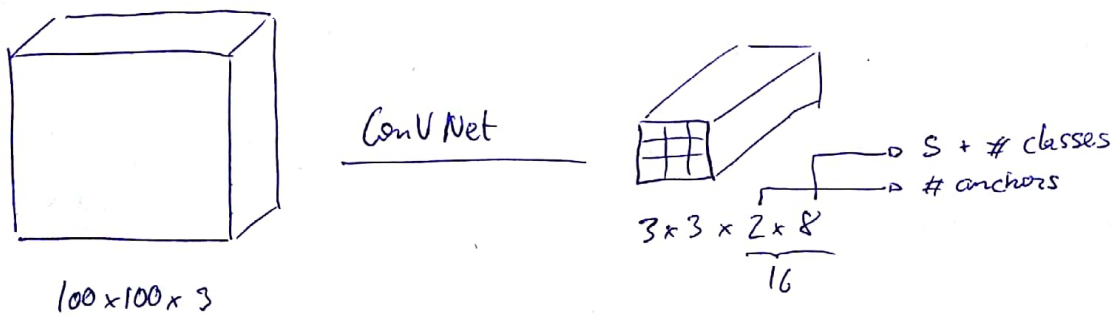
With two anchor boxes

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

Output y (3 x 3 x 16)

It allows the algorithm to specialize

- flat : cars
- tall : pedestrians



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

Scanned by CamScanner

- You are working on a factory automation task. Your system will see a can of soft-drink coming down a conveyor belt, and you want it to take a picture and decide if there is a soft-drink can in the image, and if so its bounding box. Since the can is round from above, the bounding box is always square, and the can always appears as the same size in the image. The most appropriate set of output units for your NN are:

  - logistic unit, b_x and b_y

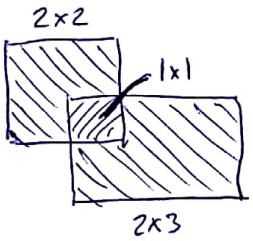
- If you build a NN that inputs a picture of a person's face and outputs N landmarks on the face, how many outputs will have the NN?

  - 2N

- You need a training set that contains many pictures of the object(s) you wish to detect as well as bounding boxes provided in the training set

- If you apply a sliding windows classifier (non-convolutional implementation), increasing the stride would tend to decrease accuracy but decrease computational cost

- In the YOLO algorithm, at training time, only one cell (the one containing the center / midpoint of an object) is responsible for detecting this object

- 

$$IoU = \frac{(1 \cdot 1)}{(2 \cdot 2) + (2 \cdot 3) - (1 \cdot 1)} = \frac{1}{9}$$

Scanned by CamScanner

Week 4

Face recognition VS Face verification

- Verification

  1:1

  - Input image, name ID
  - Output whether the input image is that of the claimed person

- Recognition

  - Has a database of K persons

  1:K

  - Get an input image
  - Output ID if the image is any of the K people (or not recognized)

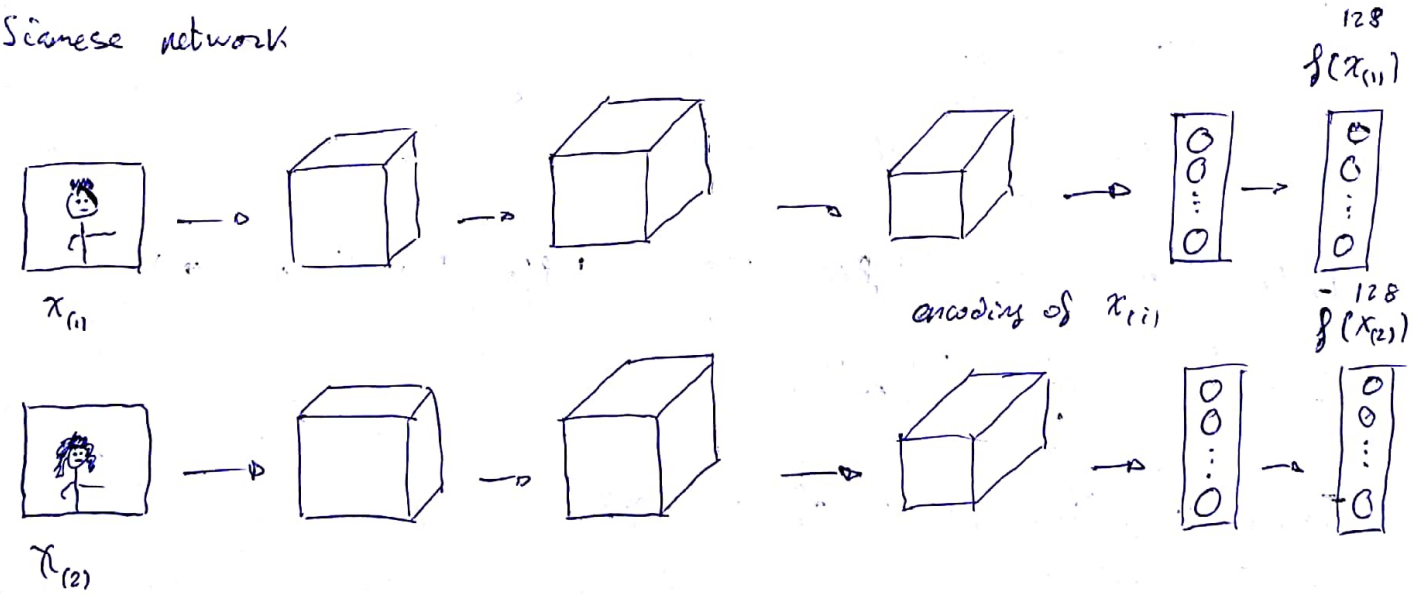
One-shot learning → recognize a person given a single image

Learning a "similarity" function

$d(\text{img } 1, \text{img } 2)$ = degree of difference between images

if $d(\text{img } 1, \text{img } 2) \leq \tau$ "same"
$> \tau$ "different" } verification

Siamese network



$d(x_{(1)}, x_{(2)}) = \|f(x_{(1)}) - f(x_{(2)})\|_2^2$ Parameters of NN define an encoding $f(x_{(i)})$ 128

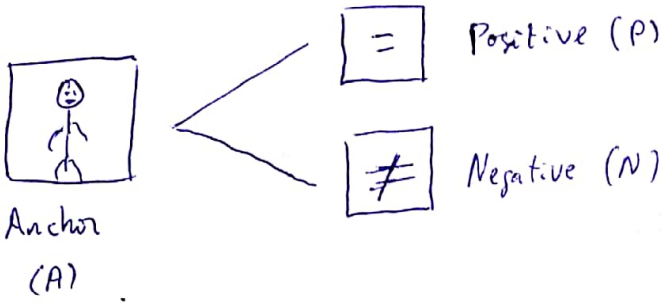
Learn parameters so that:

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, $\|f(x_{(i)}) - f(x_{(j)})\|$ is large

Scanned by CamScanner

Applied gradient descent on the triplet loss function



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

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

trivial solutions: $d(A,P) = d(A,N)$

$d(A,P) = 0, \quad d(A,N) = 0$

to avoid the NN from training of trivial solutions:

$$\|f(A)-f(P)\|^2 - \|f(A)-f(N)\|^2 \leq \underbrace{-\alpha}_{\text{margin (ex: 0.2)}}$$

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

Loss function → triplet

Given 3 images A, P, N :

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

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

Training set: $\boxed{10\,k}$ pictures of 1 k people

Scanned by CamScanner

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 → gradient descent won't do much progress

So choose triplets that are "hard" to train on: $d(A,P) \approx d(A,N)$
→ gradient descent works hard → much more efficient algorithm

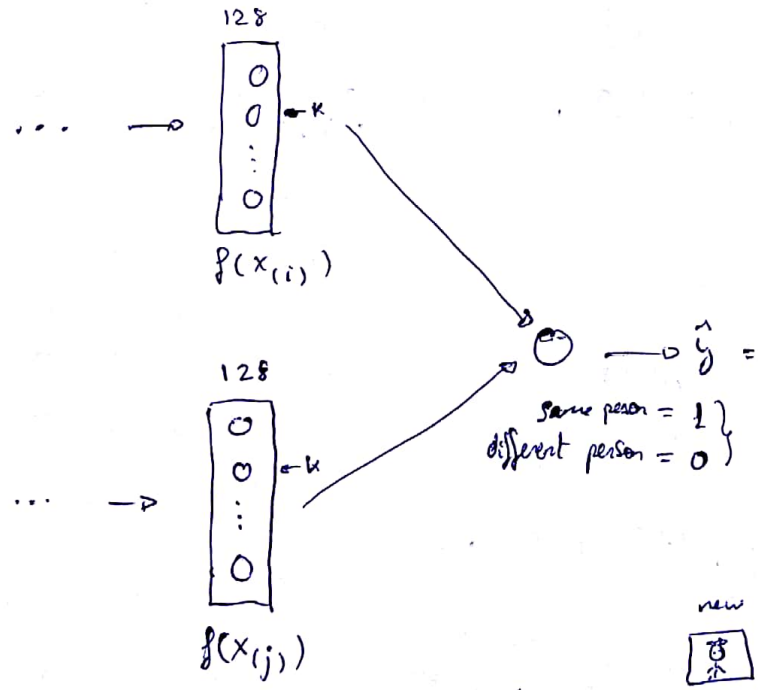
In work papers of deep learnin comunity, usually are named after like this:

____ Net

Deep ____

Companies have trained already large networks and posted parameters online, so rather than trying to train one of these networks from scratch you can use transfer data of volume sizes → pre-trained model

Alternative of the triple loss → binary classification problem



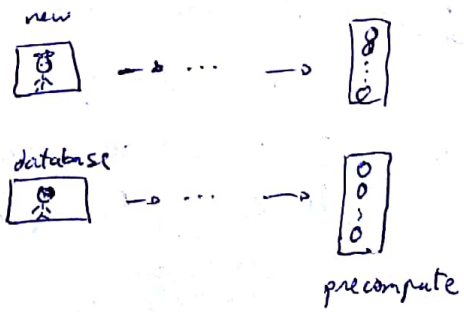
$$\chi^2 = \frac{f(x_{(i)})_k - f(x_{(j)})_k}{f(x_{(i)})_k + f(x_{(j)})_k}$$

(in other papers)

or instead

$$\hat{y} = \sigma\left(\sum_{k=1}^{128} w_i \overbrace{|f(x_{(i)})_k - f(x_{(j)})_k|}^{\text{element wise difference}} + b\right)$$

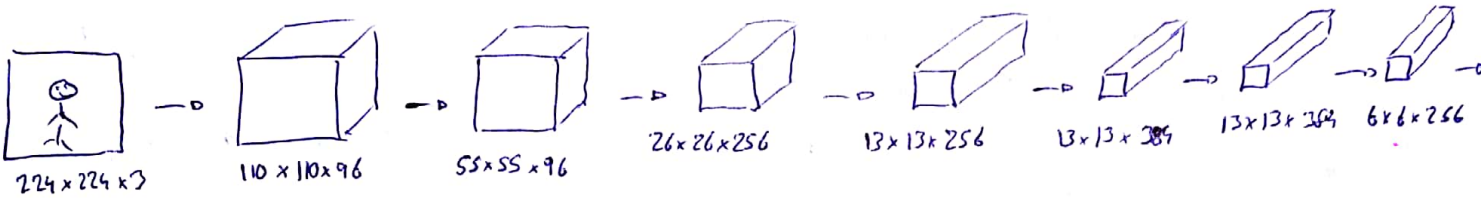
Face recognition of employees:

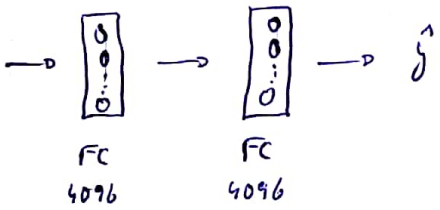


Scanned by CamScanner

Neural Style Transfer

- Content (C) : picture you want to modify its artistic style
- Style (S) : style reference
- Generated image (G) : result





Pick a unit in layer 1. Find the nine image patches that maximize the unit's activation

Repeat for other units

Later units see larger image patches

Every layer sees a different set of portion of the original image but in other format such as a set of edges, circles, ...

The deeper layers the more complex comprehension they have about detecting things. From shallow to deep layers.

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

1. Initiate G randomly

   $G : 100 \times 100 \times 3$

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

$$G := G - \frac{\partial}{\partial G} J(G)$$

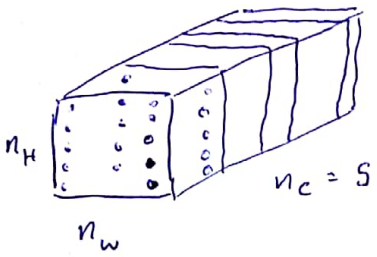
- Say you have used hidden layer l to compute content cost
- Use pretrained ConvNet (e.g. VGG network)
- let $a^{[l](C)}$ and $a^{[l](G)}$ be the activation of layer l on the images
- If $a^{[l](C)}$ and $a^{[l](G)}$ are similar, both images have similar content

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

Scanned by CamScanner

J_{style}

- Say you are using layer l's activation to measure "style"
- Define style as correlation between activations across channels



How correlated are the activations across different channels?

Style matrix (Gram matrix)

- let $a^{[l]}_{i,j,k}$ = activation at $(\underset{H}{i}, \underset{W}{j}, \underset{C}{k})$. $G^{[l]}$ is $n_c^{[l]} \times n_c^{[l]}$

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

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

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

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

1D and 3D convolutions

- computed tomography
- electrocardiography

Scanned by CamScanner

Face verification requires comparing a new picture against one person's face, whereas face recognition requires comparing a new picture against k persons' faces

We learn $d(img1, img2)$ function for face verification as

- this allows us to learn to recognize a new person given just a single image of that person
- we need to solve a one-shot learning problem

In the deeper layers of a ConvNet, each channel corresponds to a different feature detector. The style matrix $G^{[l]}$ measures the degree to which the activations of different feature detectors in layer l vary (or correlate) together with each other

In a neural style transfer, what is updated in each iteration of the optimization algorithm?

- The pixel values of the generated image G

You are working with 3D data. You are building a network layer whose input volume has size $32 \times 32 \times 32 \times 16$ (this volume has 16 channels), and applies convolutions with 32 filters of dimension $3 \times 3 \times 3$ (no padding, stride 1). What is the resulting output volume?

- $30 \times 30 \times 30 \times 32$ $\qquad n - f + 1 = 30$

Scanned by CamScanner

Sequence Models

- Examples → Supervised learning → label data X/Y as training set
  - Speech recognition
  - Music generation → the input can be an empty set or an integer referring the kind of music you want to generate
  - Sentiment classification
  - DNA sequence analysis
  - Machine translation
  - Video activity recognition
  - Name-entity recognition

Input	Output	
Sequence	Sequence	→ they can have different sizes
Sequence	—	
—	Sequence	

- Motivating example

X:	Harry	Potter	and	Hermione	Granger	invented	a	new	spell
	$x^{<1>}$	$x^{<2>}$	$x^{<3>}$	...		$x^{<t>}$	...		$x^{<9>}$
y:	1	1	0	1	1	0	0	0	0
	$y^{<1>}$	$y^{<2>}$	$y^{<3>}$	...		$y^{<t>}$	...		$y^{<9>}$

$$\left.\begin{array}{l} T_x = 9 \\ T_y = T_x \end{array}\right\} \text{length}$$

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

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

Scanned by CamScanner

[Natural language processing]

x: Harry Potter and Hermione Granger invented a new spell

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

One-hot vectors

Vocabulary

a	1	0		0	
aaron	2	0		⋮	
⋮	⋮	⋮			
and	367				
⋮	⋮				
harry	4075	1	4075		
⋮	⋮	⋮			
Potter	6530			1	6530
⋮	⋮			⋮	
Zulu	10000	0		0	

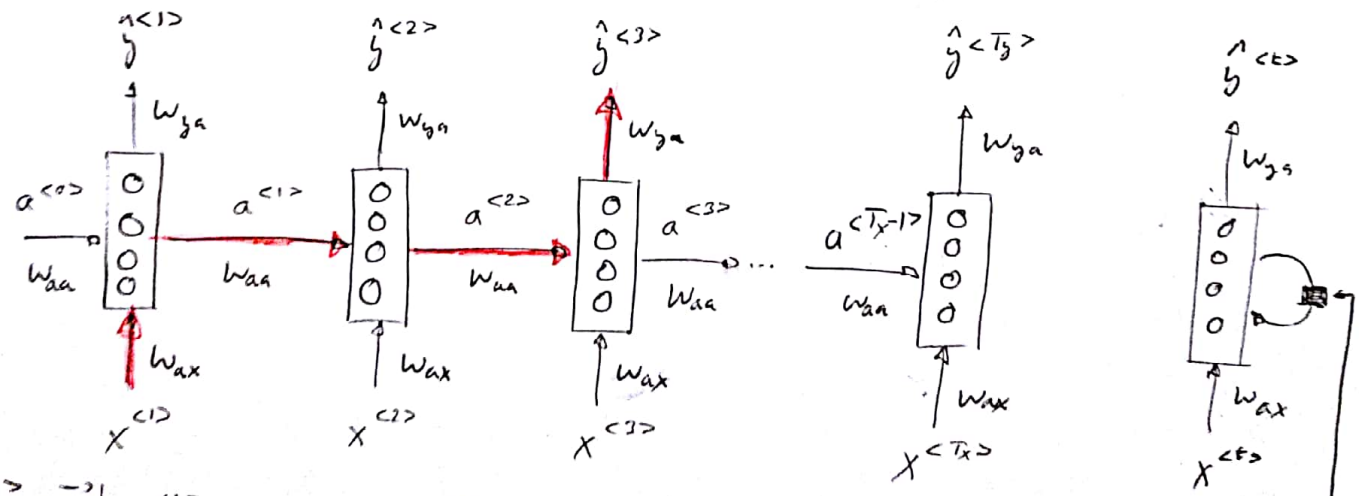
- Why not a standard NN?

Problems

- Inputs and outputs can be different lengths in different examples
- Doesn't share features learned across different positions of text

The weakness of this RNN is that it only uses the information that is earlier in the sequence to make a prediction. But not the information later in the sequence

→ Solution? → Bidirectional RNN



$a^{<0>} = \vec{0}$ or random vector

$$a^{<1>} = g_1(W_{ax}x^{<1>} + W_{aa}a^{<0>} + b_a) \leftarrow \text{tanh / ReLU}$$

$$\hat{y}^{<1>} = g_2(W_{ya}a^{<1>} + b_y) \leftarrow \text{sigmoid}$$

Scanned by CamScanner

$$a^{<t>} = g_1\left(\underset{(100,100)}{W_{aa}}\ \underset{(100)}{a^{<t-1>}} + \underset{(100,10000)}{W_{ax}}\ \underset{(10000)}{x^{<t>}} + b_a\right)$$

$$\hat{y}^{<t>} = g_2\left(W_{ya}\ a^{<t>} + b_y\right)$$

- Simplified notation

$$a^{<t>} = g_1\left(W_a\left[a^{<t-1>}, x^{<t>}\right] + b_a\right)$$

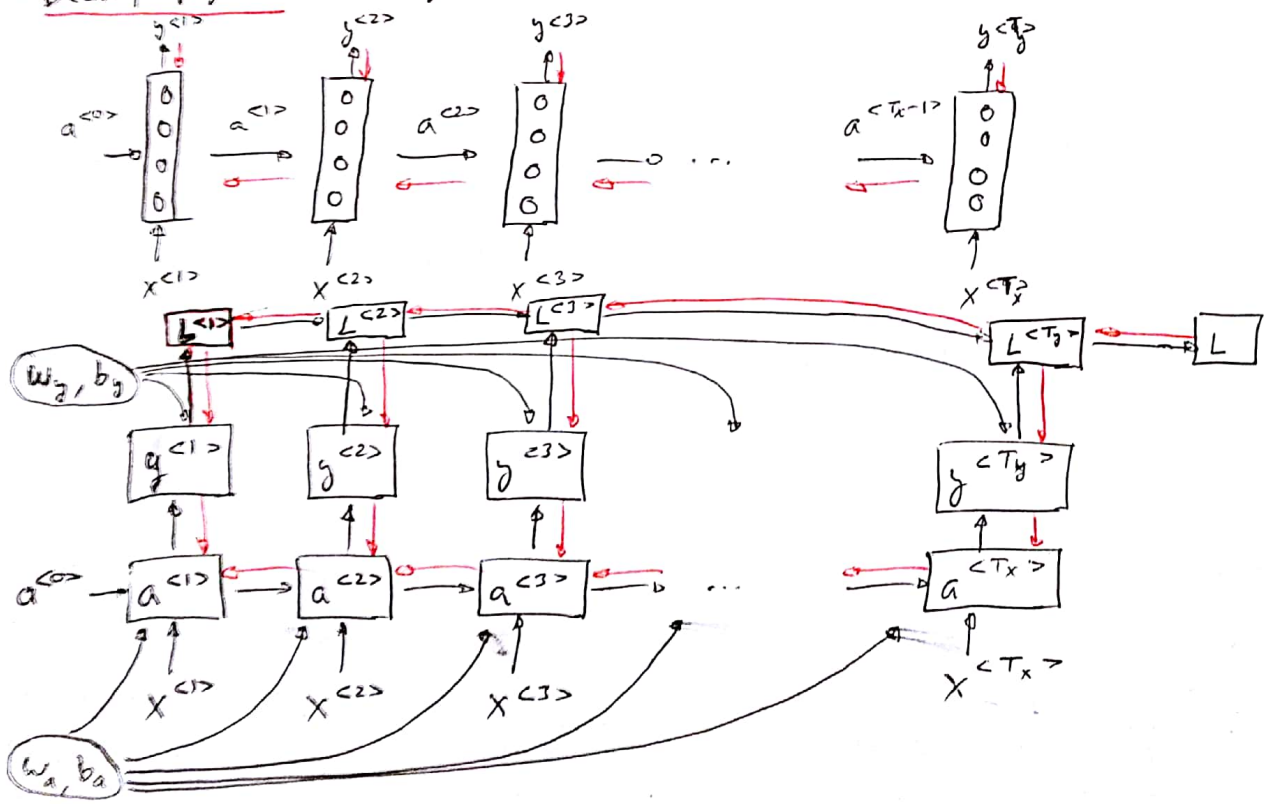
$$\begin{array}{c} 100 \\ \\ \end{array}\left[\underbrace{W_{aa}}_{100} \mid \underbrace{W_{ax}}_{10000}\right] = \underset{(100,\,10100)}{W_a}$$

$$\left[W_{aa} \mid W_{ax}\right]\left[\begin{array}{c} a^{<t-1>} \\ \hline x^{<t>} \end{array}\right] = W_{aa}\,a^{<t-1>} + W_{ax}\,x^{<1>}$$

$$\begin{array}{c} 10100 \\ \\ \end{array}\left[\begin{array}{c} 100 \\ 10000 \end{array}\left[\begin{array}{c} a^{<t-1>} \\ \hline x^{<t>} \end{array}\right]\right] = \left[a^{<t-1>}, x^{<t>}\right]$$

$$\hat{y}^{<t>} = g_2\left(W_y\ a^{<t>} + b_y\right)$$

- Back propagation and forward propagation



For computing Back propagation → necessary a loss function → standard/entropy loss

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

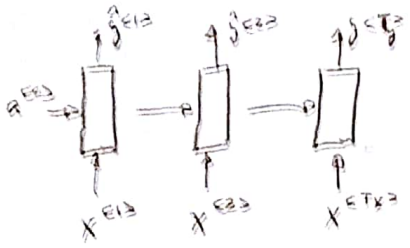
$$L\left(\hat{y}^{<t>}, y^{<t>}\right) = \sum_{t=1}^{T_y} L^{<t>}\left(\hat{y}^{<t>}, y^{<t>}\right)$$

Scanned by CamScanner

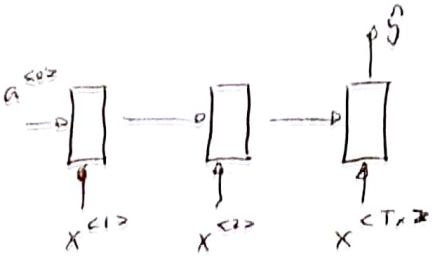
- RNN architectures

Same length

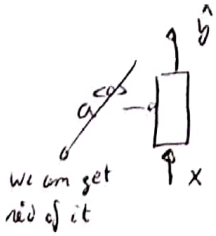
Many-to-many
(inputs) (outputs)



Many-to-one
- Sentiment classification

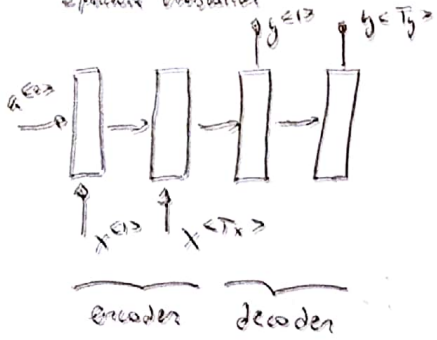


One-to-one
- Standard NN

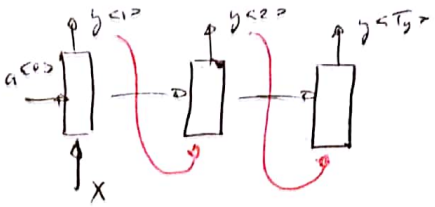


Different length

Many-to-many
- Machine translation



One to many
- Music generation



↳ integer or null

np. random. choice

- Language model

  - Speech recognition

    { "The apple and pair salad" ✗
    "The apple and pear salad" ✓

    A speech recognition systems picks the ambiguous words by chance of appearing sequently

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

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

    $P(\text{sentence}) = ?$ $\quad P(y^{<1>}, \dots y^{<T_y>})$

  - Training set: large corpus of a language text

  - Tokenize sentence → input sentence into mapping of individual tokens

    period can act as an specific token, but it's not necessary

    Cats average 15 hours of sleep a day.

    $y^{<1>}$ $y^{<2>}$ ... $y^{<8>}$ <eos> end of sentence

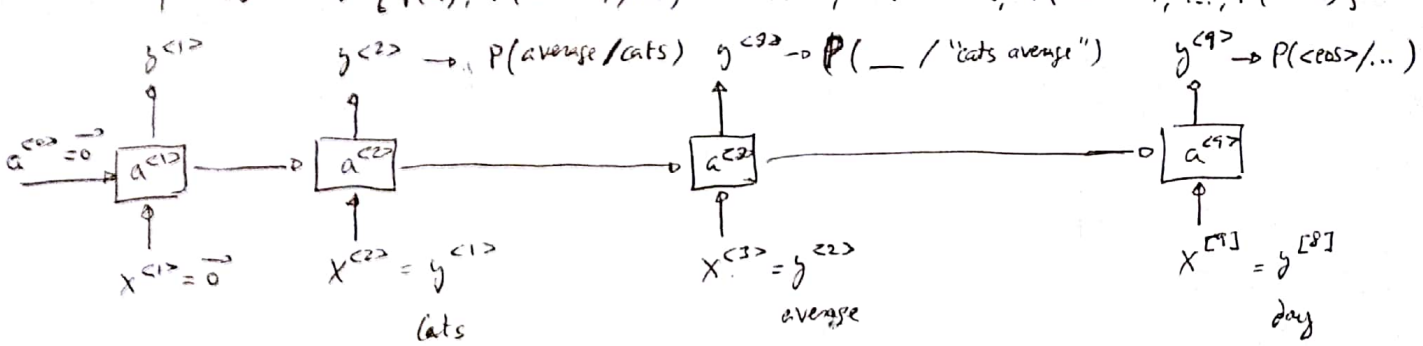
    If a word is not in the dictionary
    ↳ token is <unk> (unknown)

$x^{<t>} = y^{<t-1>}$

Scanned by CamScanner

- RNN model

softmax of $\{P(a), P(aaron), \dots, P(cats), \dots, P(<unk>), P(<eos>), \dots, P(zulu)\}$



Cats average 15 hours of sleep a day. <eos>

Softmax loss function $L(\hat{y}^{<t>}, y^{<t>}) = -\sum y_i^{<t>} \log \hat{y}_i^{<t>}$

$$L = \sum_t L^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

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

$P(y^{<t>} \mid y^{<1>}, y^{<2>}, \dots y^{<t-1>})$ at the tth step is what the RNN is doing

- Novel sequence sampling

Word language model Vocabulary = [a, aaron, ..., zulu, <unk>]

Character " " " = [a, b, c, ..., z, _, ., ,, ;, 0, ..., 9, A, ..., Z]

- there are not unknown tokens
- Many sequence → non-zero probability
- Disadvantage
  - much longer sequences
  - not good for long range dependencies between the earlier part of the sentence and the latter one

- A basic RNN algorithm → runs into vanishing gradient problems
  - weakness

The [cat], which already ate ..., [was] full
The [cats], " " ..., [were] full

There can be dependencies among certain words which can be distant. So in some cases is difficult to backpropagate all the way back just to correct a distant term

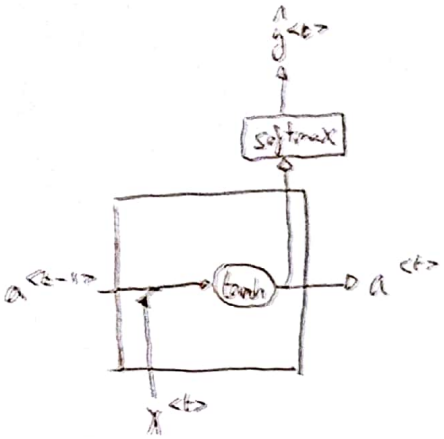
Vanishing gradients are more complicated to solve than exploding g.
When exploding gradient happens it can be catastrophic

- cause large parameters → a mess
- but easy to spot bcs the parameters just blow up → NaN → numerical overflow (not a number)
- Solution → apply gradient clipping
  - checking gradient vectors → re-scale them if the values are above certain threshold

The chances of this of appearing are exponentially function of # layers

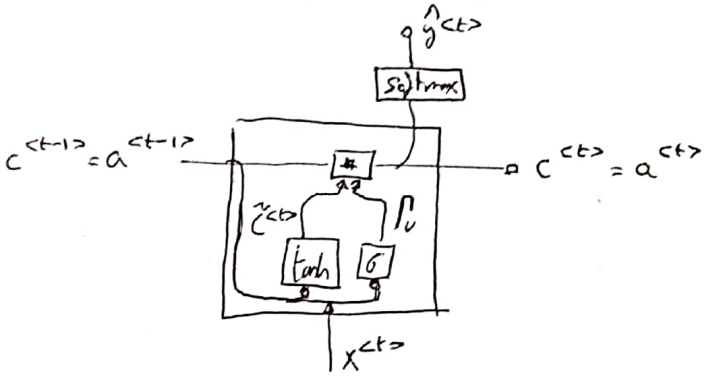
Scanned by CamScanner

- Gated Recurrent Unit → modification to the RNN hidden layer



- better at capturing long range connections
- helps a lot the vanishing gradient problems

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



- Simplified GRU

C = memory cell

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

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

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

W_{gate} → sigmoid can be $\begin{cases} 0 \rightarrow \text{don't update} \\ 1 \rightarrow \text{update the value} \end{cases}$

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

- Full GRU

The same but with a modification in $\tilde{c}^{<t>}$ equation

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

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

LSTM

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

Scanned by CamScanner

- Long short term memory (LSTM) unit
  - More powerful than GRU

<u>GRU equations</u> → it is a Pavia's simplification of the LSTM model
more computionally economical and simpler

$\tilde{c}^{<t>} = \tanh(W_c(\Gamma_r c^{<t-1>}, x^{<t>}) + b_c)$ → candidate for replace memory cell

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

$\Gamma_r = \sigma(W_u(c^{<t-1>}, x^{<t>}) + b_r)$ → relevance gate

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

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

<u>LSTM equations</u> appeared ~~later~~ earlier than GRU equations
more powerful and flexible

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

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

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

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

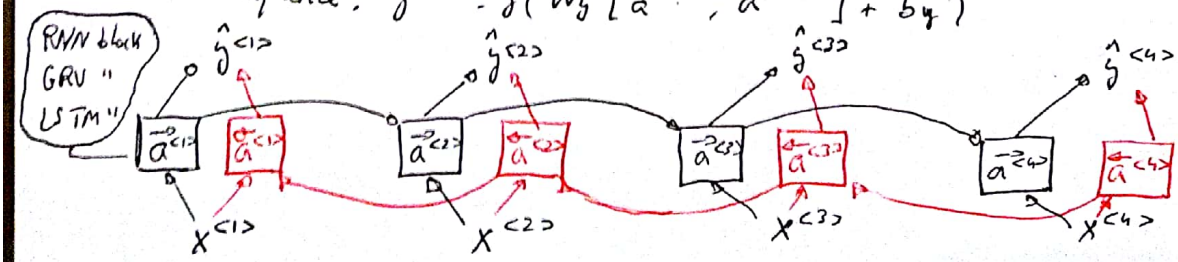
$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f c^{<t-1>}$

$a^{<t>} = \Gamma_o * \tanh(c^{<t>})$ $\qquad y^{<t>} = \text{softmax}(W_y a^{<t>} + b_y)$

Variant → Gates' W_i are also function of $c^{<t-1>}$ → Peephole connection

- Bidirectional RNNs

Let you in a point of time to take information from both earlier and later in the sequence. $\hat{y}^{<t>} = g(W_y[\overrightarrow{a}^{<t>}, \overleftarrow{a}^{<t>}] + b_y)$

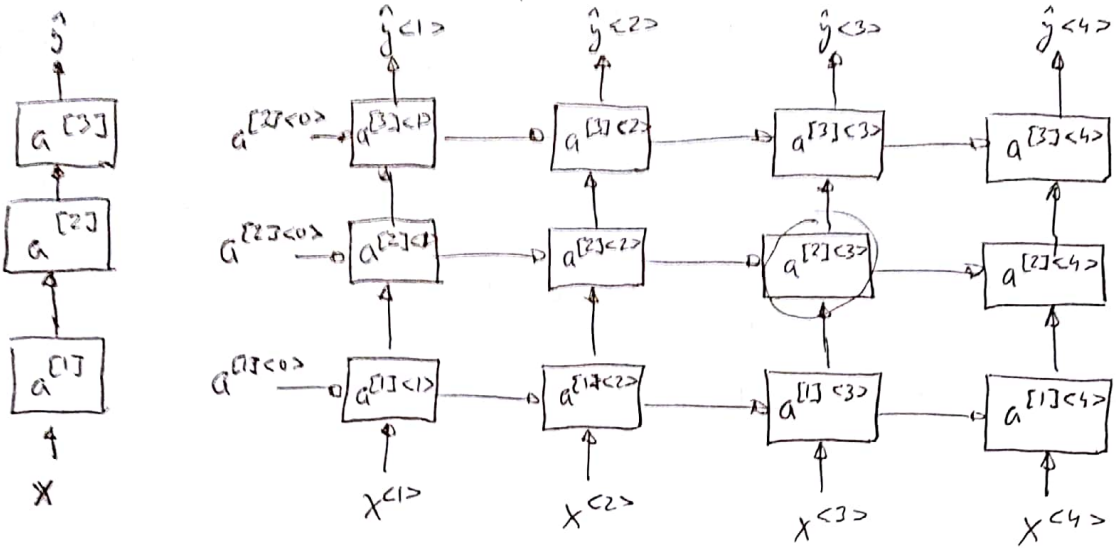


Acyclic graph He said "Teddy Roosevelt" → is this a person's name?

Scanned by CamScanner

- Deep RNN

Stacking layers → 3 is quite a lot



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

- Tests

- Γ_u is a vector of dimension equal to the # of hidden units in the LSTM
- In the GRU if Γ_r is removed, because if $\Gamma_u \approx 0$ for a timestep, the gradient can propagate back through that time step without much decay. For the signal to backpropagate without vanishing, we need $c^{<t>}$ to be highly dependant on $c^{<t-1>}$
- $\underbrace{\Gamma_u \text{ and } \Gamma_r}_{\text{LSTM}}$ play a similar role to $\underbrace{\Gamma_u \text{ and } 1-\Gamma_u}_{\text{GRU}}$

Scanned by CamScanner

Word Embeddings and NLP

↳ a way of representing words

↳ algorithms understand analogies between words

- Instead of one-hot representation → Featurized representation

	Man (5391)	Woman (9853)	King (4914)	Queen (7157)	Apple (456)	Orange (6257)
Gender	-1	1	-0.95	0.97	0	0.01
Royal	0.01	0.02	0.93	0.95	-0.01	0.
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
⋮	⋮	⋮	⋮	⋮	⋮	⋮
Size						
Cost						
Verb						
	e_{5391} e_{man}	e_{9853} e_{woman}				

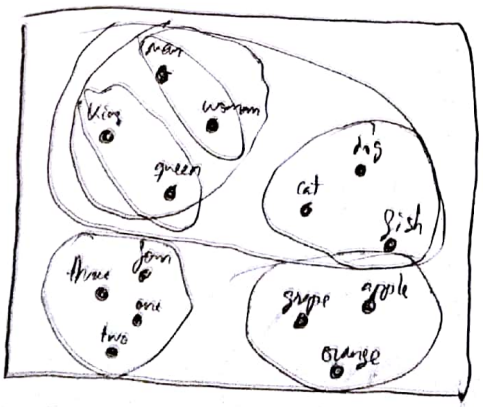
300 dimensional embedding

I want a glass of orange juice

I want a glass of apple juice

This allows to require much lesser data set amounts (label data)

- Visualizing word embeddings → t-sne representation



~~One-hot-vectors~~ → Embedding vectors

Scanned by CamScanner

- Transfer learning and word embeddings

(A) 1. Learn word embeddings from large text corpus (1-100 B words)
(or download pre-trained embedding online)

(B) 2. Transfer embeddings to new task with smaller training set
(say 100 K words)

3. Optional: Continue to finetune the word embeddings with new data

- Relation to face encoding (embedding)

If the input is not in the dictionary → declared as unknown

Interchangeability → encoding ≈ embedding

- Analogy reasoning

Watching the featurized table...

Man → Woman King → ... ?

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

$e_{man} - e_{woman} \approx e_{King} - e_w \rightarrow e_w = e_{Woman}$

Find word w: $\arg\max_w \; \underset{\substack{\uparrow \\ \text{similarity function}}}{\text{sim}} \left(e_w, \underbrace{e_{King} - e_{man} + e_{woman}}_{30 \sim 75\%}\right)$

In t-SNE parallelogram relationships don't hold ~~the~~ time → non-linear mapping 2D
↳ converts the 300 Dimensional data into 2D

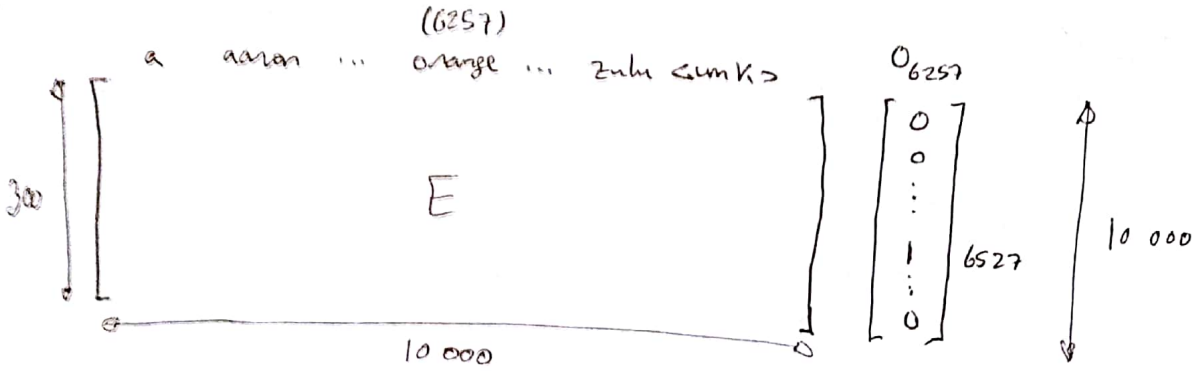
- Cosine similarity

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

Scanned by CamScanner

- Embedding matrix

  - But in practice instead of matrix multiplication → specialized function to look up an embedding



$$\underset{(300,10K)}{E}\ \underset{(10K,1)}{O_{6257}} = \underset{(300,1)}{e_{6257}} \rightarrow E\,O_j = e_j \Leftarrow \text{embedding for word } j$$

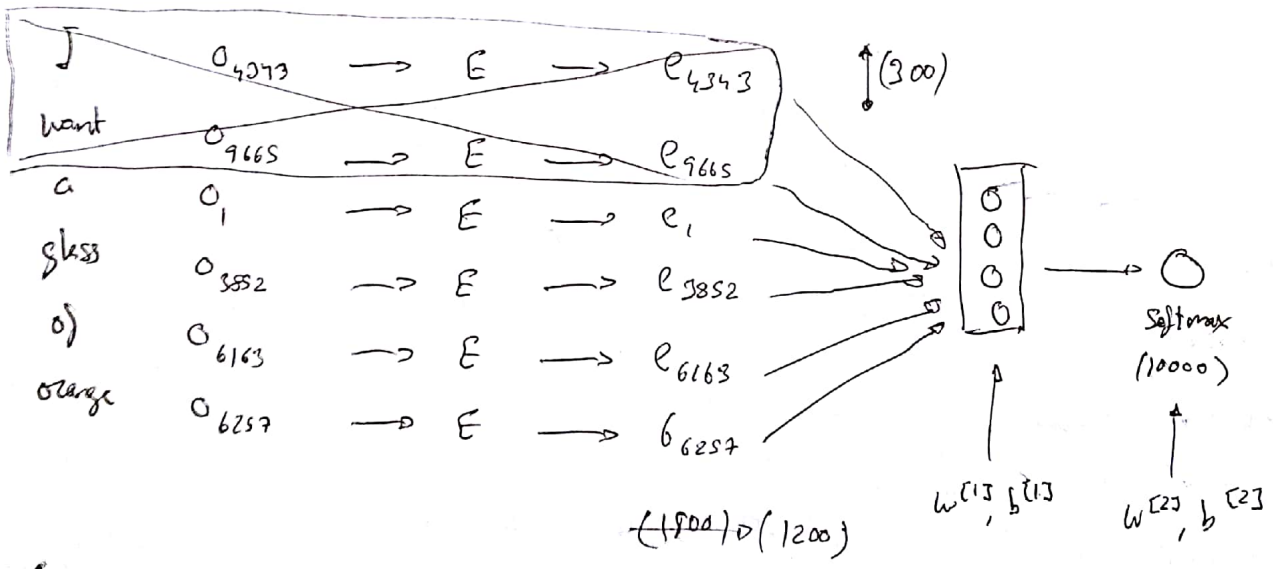
- Neural language model

4 word history ← enough to figure out the unknown word

I want a glass of orange ___

(4343) (9665) (1) (3852) (6163) (6257)

↑ context



Context : 4 last words → ~~usual~~ standard by default

4 words on left and right

Last 1 word
Nearby 1 word } Skip gram model → works remarkably well

Scanned by CamScanner

- Word 2 Vec

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

Context	Target
Orange	juice
Orange	glass
Orange	my

- CBow model (Continuous backwards model) → takes the surrounding words of the model
- Skip - gram model

Vocab size = 10 000 words

Context c ("orange") → Target t ("juice")
6257 4834

$o_c \rightarrow E \rightarrow e_c \rightarrow \text{softmax} \rightarrow \hat{y}$

$e_c = E o_c$

Softmax : $P(t|c) = \dfrac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10000} e^{\theta_j^T e_c}}$

θ_t : parameter associated with output t

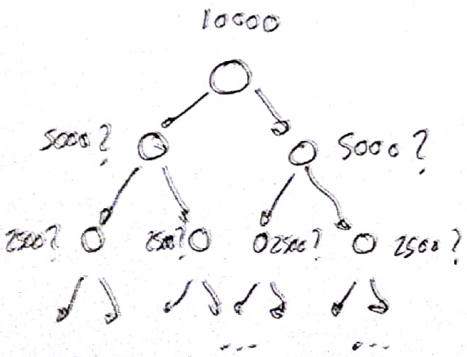
$L(\hat{y}, y) = -\sum_{i=1}^{10000} y_i \log \hat{y}_i$

$y = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \leftarrow 4834$

Problem → computational speed

↳ Solution → Hierarchical softmax classifier

Symmetrical tree



↳ it diminishes the computation time by $\log |V|$
↓
words in vocabulary

Asymmetrical tree → better

↳ Most common words in the top

How to sample the context c ?

Selecting for training set the not that common words

Avoid : the, of, a, and, to...

Scanned by CamScanner

- Negative sampling

↳ much more efficient than skip-gram algorithm

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

$x \longrightarrow y$ (Any association ?)

Context	Word	Target ?
orange	juice	1
orange	King	0
orange	book	0
orange	the	0
orange	of	0

K

deliberately generated samples

↳ how to select them ?

$K = 5-20$ smaller datasets

$K = 2-5$ larger "

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

Orange (6257)

$O_{6257} \rightarrow E \rightarrow e_{6257}$ → 0, 0, 0, 0, 0 juice ?

10000 binary classification problem

K+1

$$P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=1}^{10k} f(w_j)^{3/4}} \longrightarrow \frac{1}{|V|}$$

frequency of ith word

Scanned by CamScanner

- GloVe algorithm (global vectors for word representation)

  ↳ less used but more simple

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

  c t

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

$X_{ij} = X_{ji}$

minimize $\sum_{i=1}^{10K} \sum_{j=1}^{10K} \underbrace{f(X_{ij})}_{\text{weight term}} (\theta_i^T e_j + \overset{t}{b_i} + \overset{c}{b_j} - \log X_{ij})^2$ → 0?

θ_i and e_j are symmetric

$e_w^{(final)} = \frac{e_w + \theta_w}{2}$

$f(X_{ij}) = 0$ if $X_{ij} = 0$ → "$0 \log 0$" $= 0$

↳ avoids $-\infty$ values NaN

↳ high values when → this, is, of, a, ... common words

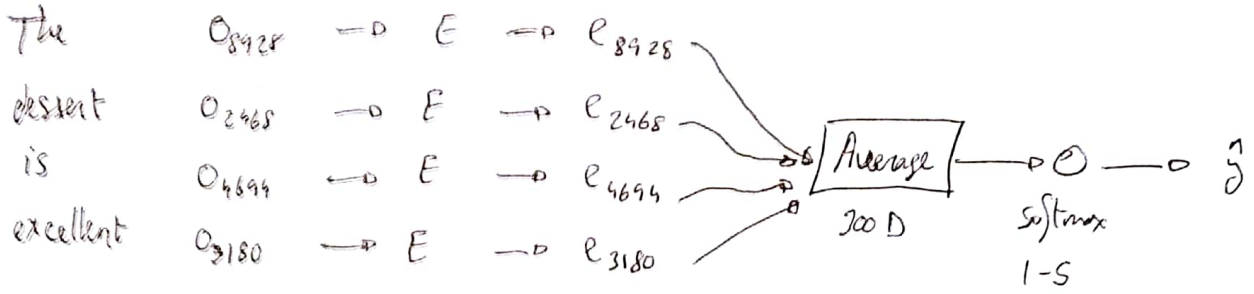
↳ low values when → durian uncommon words

The axis used to represent the features will be well-aligned

Scanned by CamScanner

- Sentiment classification

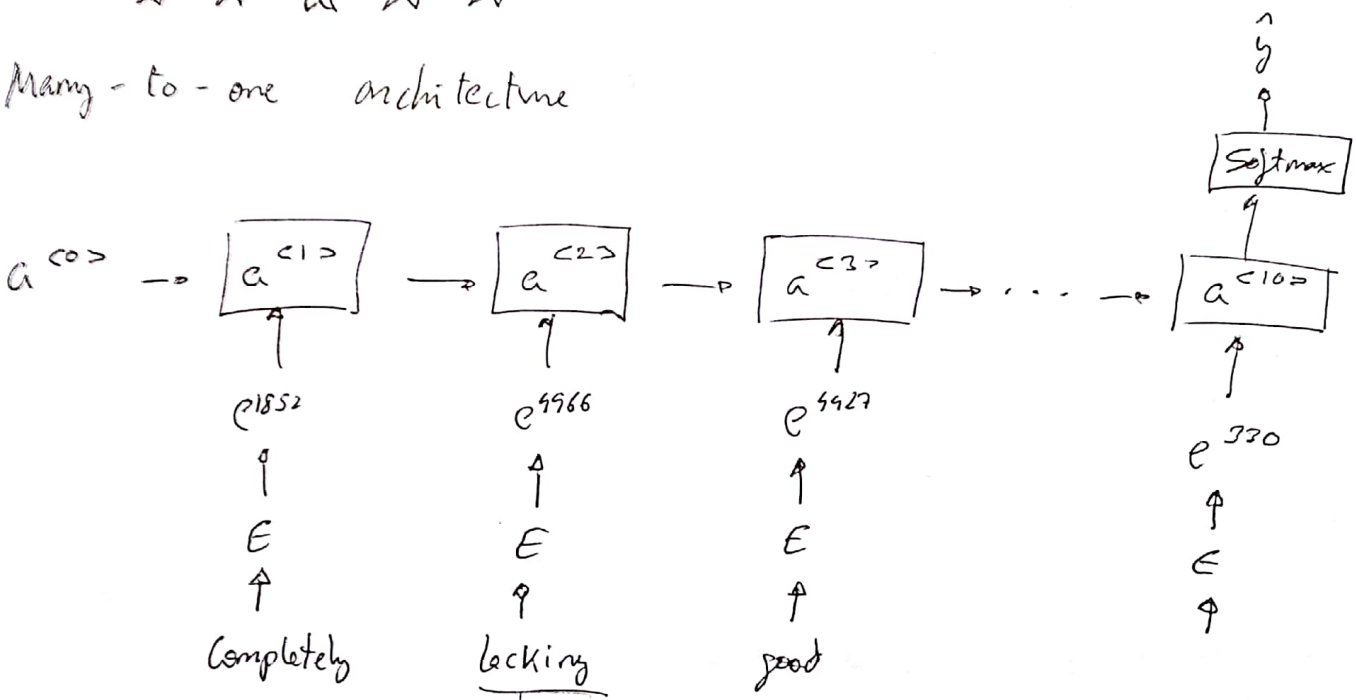
The dessert is excellent
8928 2468 4694 3180



"Completely lacking good taste, good service, and good ambience"

↳ ★ ☆ ☆ ☆ ☆

Many-to-one architecture



↳ even this word is not in the vocabulary, this RNN provides good results

Scanned by CamScanner

- Debiasing word embedding → trust results

↳ can affect gender, ethnicity, age, sexual orientation and others ~~bias~~

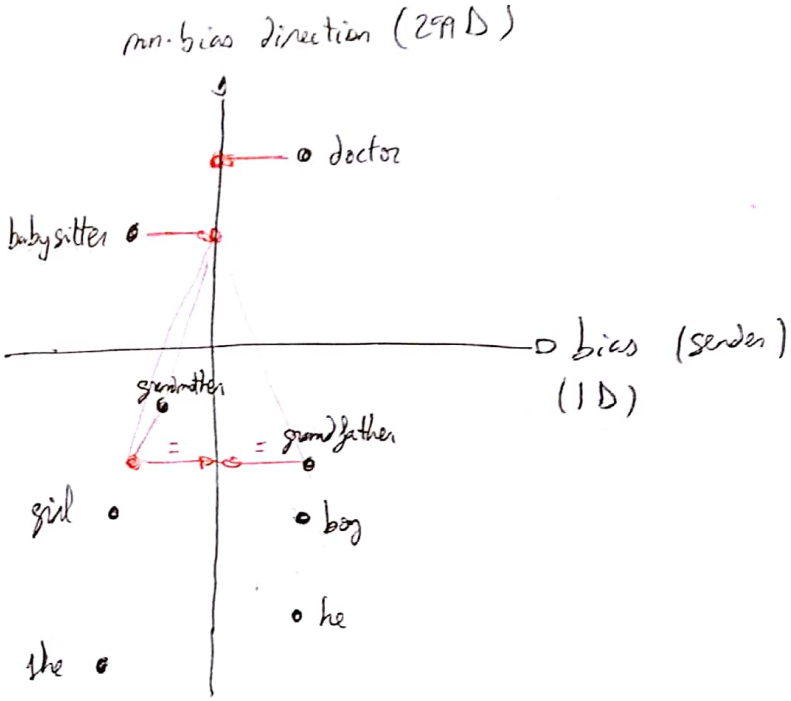
Machine learning is used to make important unbiased decisions

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

not an gender intrinsec word such as boy, girl, ...

3. Equalize pairs of definitional words

grandfather - grandmother

- Tests

- The dimension of word vectors is usually smaller than the size of the vocabulary. Most common sizes of word vectors ranges between 50 and 400.
- The t-SNE is a non-linear dimensionality reduction technique
- Word vectors empower your model with an incredible ability to generalize
- The following equations provide a good word embedding

$e_{boy} - e_{girl} \approx e_{brother} - e_{sister} \rightarrow e_{boy} - e_{brother} \approx e_{girl} - e_{sister}$

- When learning word embeddings, we create an artificial task of estimating $P(t|c)$. It is okay if we do poorly on this artificial prediction task; the more important by-product of this task is that we learn a useful set of word embeddings

Scanned by CamScanner

- Tests

- In the word2vec algorithm, you estimate $P(t|c)$ where t is the target word and c is a context word,
  t and c are chosen to be nearby words

- Suppose you have a 10000 word vocabulary, and are learning 500-dimensional word embeddings. The word2vec model uses the following softmax function

$$P(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10K} e^{\theta_j^T e_c}}$$

  - ☐ θ_t and e_c are both 500D vectors
  - ☐ θ_t and e_c are both trained with an optimization algorithm such as Adam or gradient descent

- Suppose you have a 10k word vocabulary, and are learning 500D word embeddings. The GloVe model minimizes this objective

$$\min \sum_{i=1}^{10K} \sum_{j=1}^{10K} f(X_{ij})(\theta_i^T e_j + b_i + b_j' - \log X_{ij})^2$$

  - ~~☐ θ_i and e_j should be initialized to 0 at the beginning of training~~
  - ☐ θ_i and e_j should be initialized randomly at the beginning of training
  - ☐ X_{ij} is the number of times word i appears in the context of word j
  - ☐ The weighting function $f(.)$ must satisfy $f(0) = 0$
    → helps prevent learning only from extremely common word pairs

- You have trained word embeddings using a text dataset of m_1 words. You are considering using these word embeddings for a language task, for which you have a separate labeled dataset of m_2 words

$m_1 >> m_2$

Scanned by CamScanner

- Let E be an embedding matrix, and let O_{1234} be a one-hot vector corresponding to word 1234. Then to get the embedding word 1234, why don't we call $E \cdot O_{1234}$ in Python?
  - Because it is computationally wasteful

Scanned by CamScanner

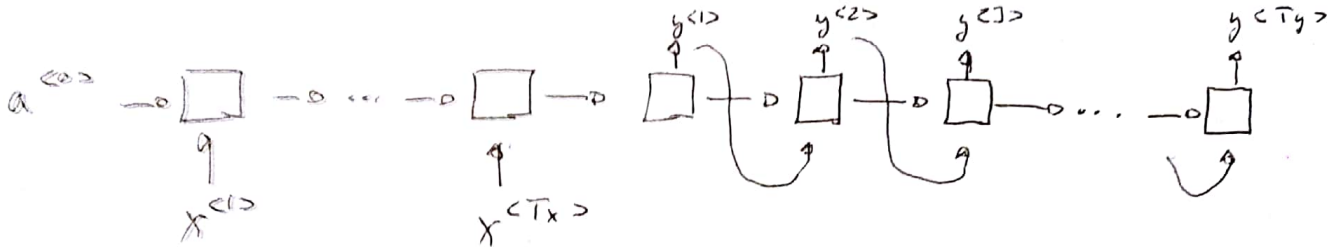
Sequence to sequence models

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

Jane visite l'Afrique en septembre

Jane is visiting Africa in September

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



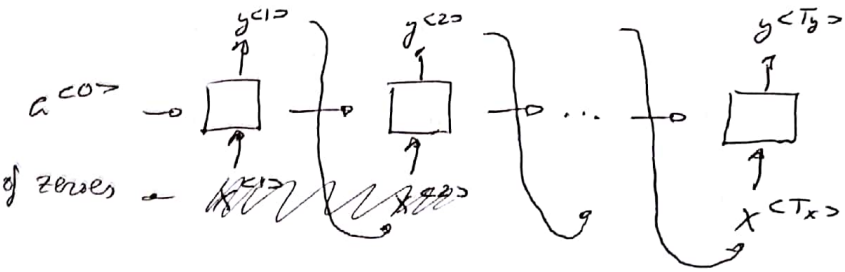
encoder network — decoder network

- RNN
  - GRU
  - LSTM

It's similar in image-to-sequence and image captioning models

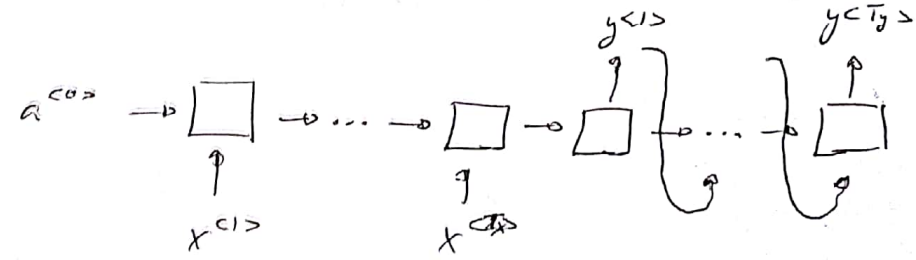
- Differences → Machine translation as building a conditional language model

- Language model:



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

- Machine translation:



Conditional language model: encoder — decoder

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

Scanned by CamScanner

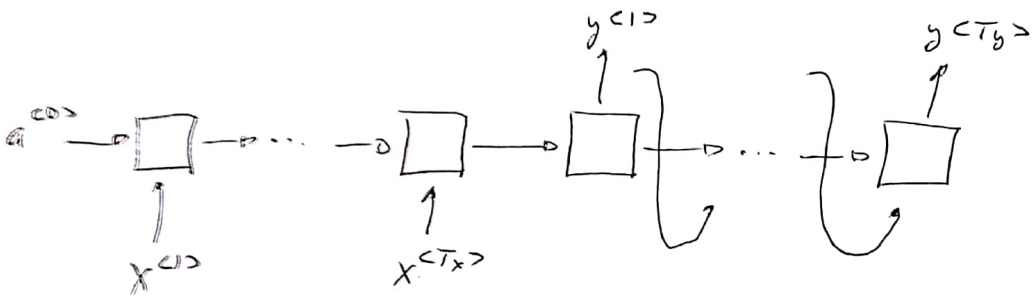
- Finding the most likely translation

Jane visite l'Afrique en septembre $\quad P(\underbrace{y^{<1>}, \dots, y^{<T_y>}}_{\text{English}}, \underbrace{x}_{\text{French}})$

- 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

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

- Why not a greedy search?



$y^{<i>}$ picks up the most likely word, $<t>$ by $<t>$

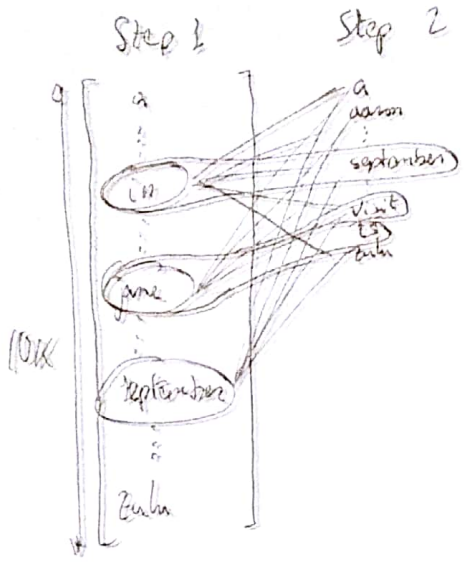
After having picked the first best word, it tries to pick the second best word

- This does not work → less optimal
  - Jane is visiting Africa in September
  - Jane is going to be visiting Africa in September

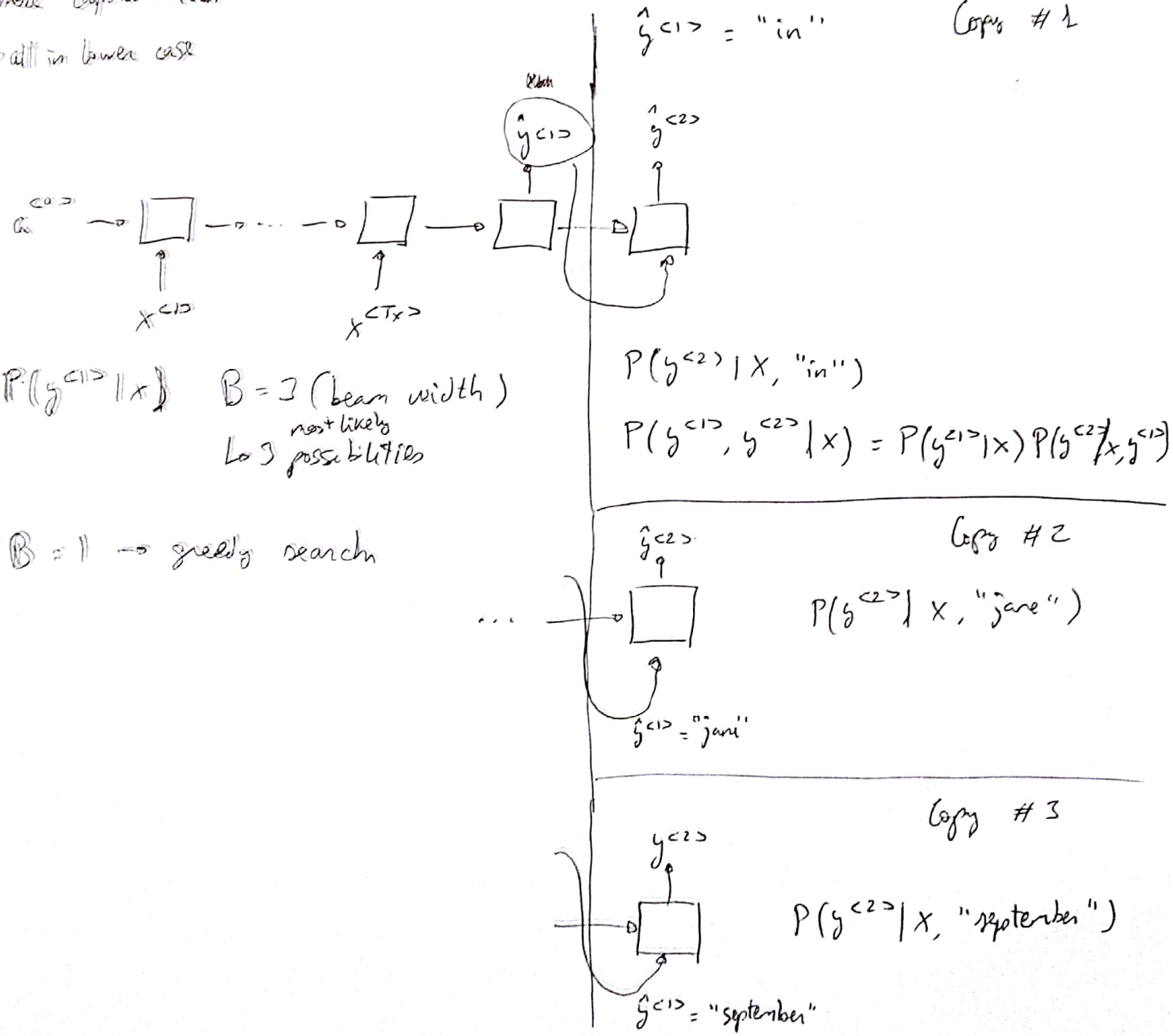
$\hookrightarrow P(\text{Jane is going} \mid x) > P(\text{Jane is visiting} \mid x)$

Scanned by CamScanner

- Beam search algorithm



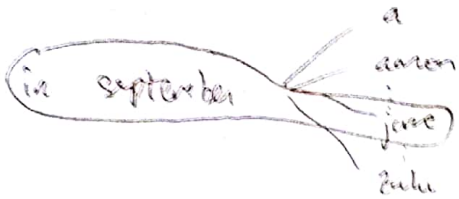
↳ ignore capitalization
↳ all in lower case



$P(y^{<1>} | x)$ B = 3 (beam width)
↳ 3 most likely possibilities

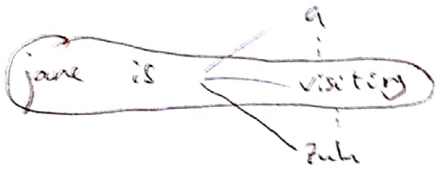
B = 1 → greedy search

Scanned by CamScanner

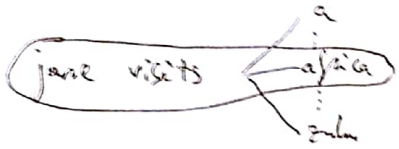


$y^{<1>}$ = "in" $\quad y^{<2>}$ = "september"

$P(y^{<3>} \mid X, \text{"in september"})$



$y^{<1>}$ = "jane" $\quad y^{<2>}$ = "is"



$y^{<1>}$ = "jane" $\quad y^{<2>}$ = "visits"

jane visits africa in september. <EOS>

$P(y^{<1>}, y^{<2>} \mid X)$

- length normalization → small change to the beam search algorithm can help get much better results

numerical underflow (opposite to NaN) → very small numbers for the floating part representation

$$\arg\max_y \underbrace{\prod_{t=1}^{T_y} P(y^{<t>} \mid X, y^{<1>}, \dots, y^{<t-1>})}$$

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

$$\arg\max_y \sum_{t=1}^{T_y} \log P(y^{<t>} \mid X, y^{<1>}, \dots y^{<t-1>})$$

T_y can be till ~ 30

normalization

$$\frac{1}{T_y^{\alpha}} \sum_{t=1}^{T_y} \log P(y^{<t>} \mid X, y^{<1>}, \dots, y^{<t-1>})$$

$\alpha = 0.7 \in [0, 1]$

Scanned by CamScanner

Beam width ? B ? → ↑ the more possibilities you consider, better results
but the more computationally expensive

$B > 100$ → is considered very large

Beam search does NOT guarantee to find an exact maximum

- Error analysis

Jane visite l'Afrique in septembre

Human: Jane visits Africa in September (y^*)

Algorithm: Jane visited Africa last September $(\hat{y})$

RNN computes $P(y^*|x) \gtreqless P(\hat{y}|x)$

Case 1: $P(y^*|x) > P(\hat{y}|x)$

↳ Beam search chose $\hat{y}$. But $\hat{y}$ attains higher $\underbrace{P(\hat{y}|x)}_{\arg\max_y P(y|x)}$

↳ at fault

Case 2: $P(y^*|x) \leq P(\hat{y}|x)$

y^* is a better translation than $\hat{y}$. But RNN predicted $P(y^*|x) < P(\hat{y}|x)$

RNN model at fault

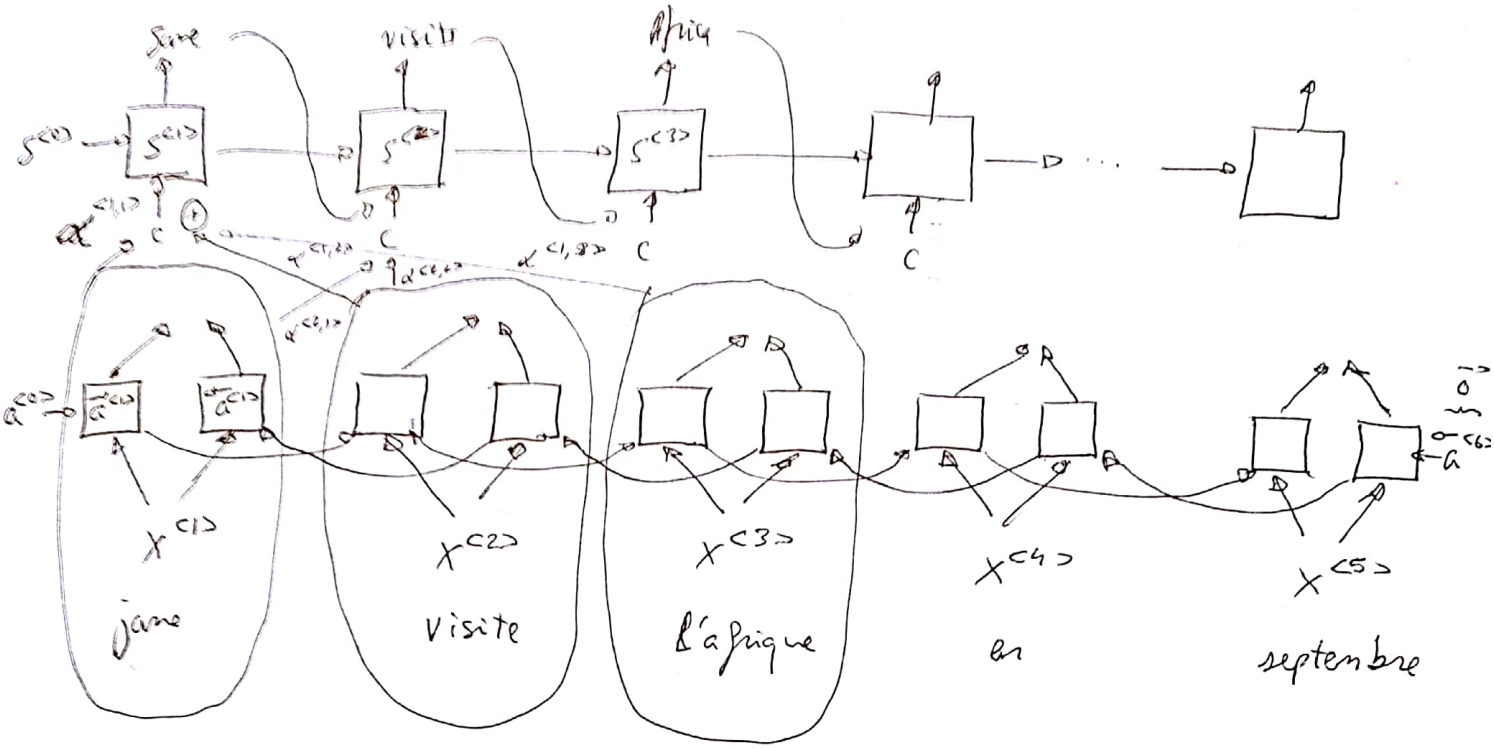
Responsible of errors
- Beam search → Increase beam width
- RNN model at fault → Add regularization, get more training data, different network architectures, ...

Scanned by CamScanner

- A modification : Attention model

The performance comes down if sentence length is higher than 20 words

It is difficult to memorize long sentences



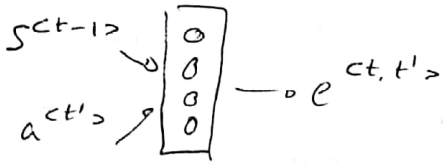
$$a^{<t'>} = (\overrightarrow{a}^{<t>}, \overleftarrow{a}^{<t>})$$

$\alpha^{<t,t'>}$: how much context would depend on the features amount of attention $y^{<t>}$ should pay to $a^{<t'>}$

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

$$C^{<1>} = \sum_{t'} \alpha^{<1,t'>} a^{<t'>} \qquad C^{<2>} = \sum_{t'} \alpha^{<2,t'>} a^{<t'>}$$

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



- Downside of this algorithm → it takes quadratic time / cost to get run

Total output # attention parameters $= T_x \cdot T_y$

Scanned by CamScanner

Speech recognition

x ——→ y

audio clip — transcript

Preprocessing → spectrograms function of (time, frequency, intensity)

Phonemes: basic units of sound → best way to do speech recognition

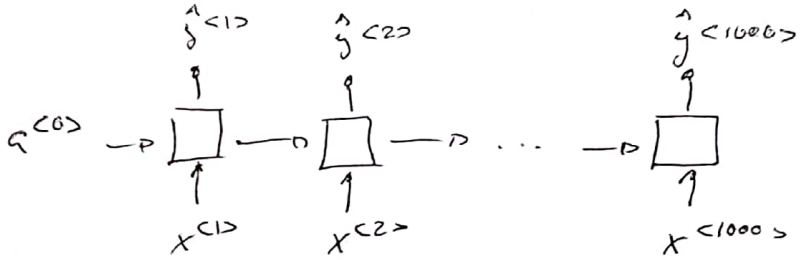
↓

No longer necessary with the end-to-end deep learning

- CTC cost for speech recognition

(Connectionist temporal classification)

"the quick brown fox" → 19 characters

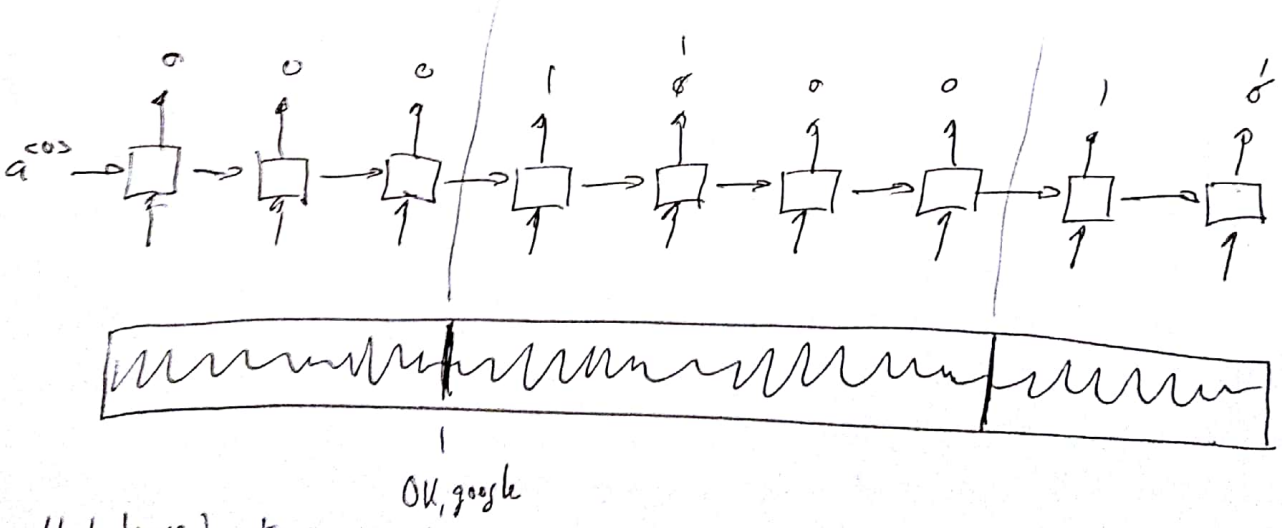


tt_h_eee ---␣--- qqq __ → the␣q

"space" "blank"

collapse

- Trigger word detection



Unbalanced training set → lots of more zeros than ones

↳ trick → instead of only just a one, output more ones

Scanned by CamScanner

Tests

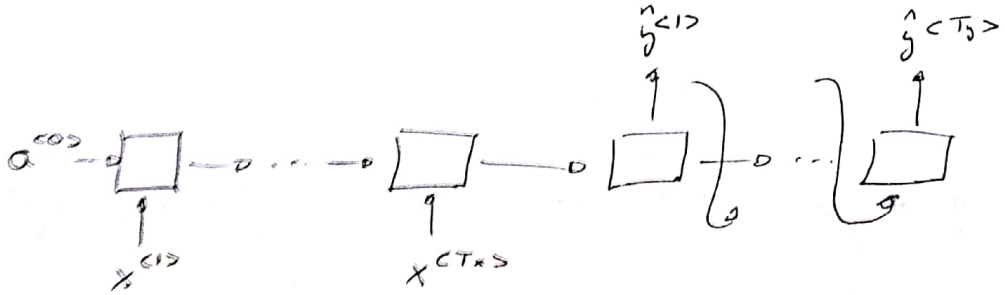
- In trigger word detection, $x^{<t>}$ is:
  - Features of the audio (such as spectrogram features) at time t
- Under the CTC model, identical repeated characters not separated by the "blank" character (_) are collapsed. Under the CTC model, what does the following string collapse to?

  __c_oo_o_kk__b_oooo_oo__kk

  - cookbook
- Compared to the encoder-decoder model shown in Question 1 of this quiz (which does not use attention mechanism), we expect the attention model to have the greatest advantage when
  - The input sequence length T_x is large
- The network learns where to pay attention by learning the values $e^{<t,t'>}$, which are computed using a small neural network:

  We can't replace $s^{<t-1>}$ with $s^{<t>}$ as an input to this NN. This is because $s^{<t>}$ depends on $\alpha^{<t,t'>}$ which in turn depends on $e^{<t,t'>}$; so at the time we need to evaluate this network, we haven't computed $s^{<t>}$ yet.

Q1)



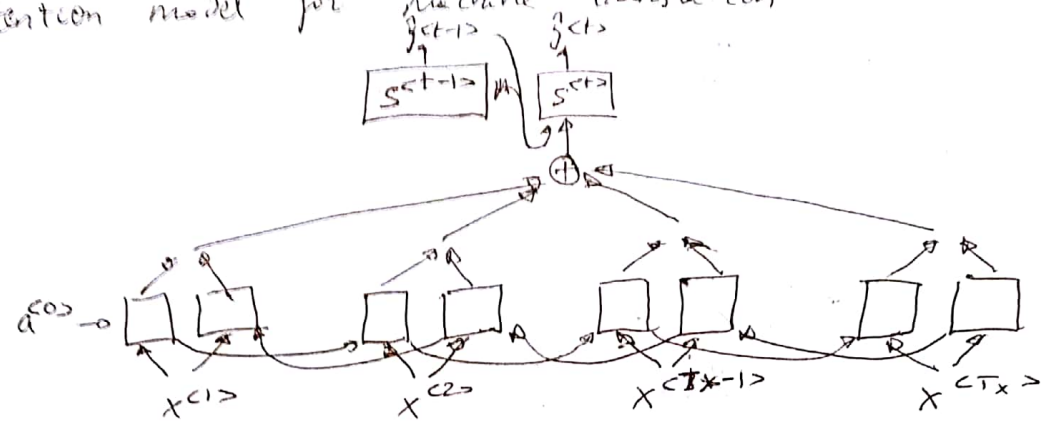
- In beam search, if you increase the beam width B
  - Beam search will run slower
  - " " " use up more memory
  - " " " generally find better solutions (i.e. do a better job maximizing $P(y|x)$)

Scanned by CamScanner

Tests

- In machine translation, if we carry out beam search without using sentence normalization, the algorithm will tend to output overly short translations

- Attention model for machine translation



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

- We expect $\alpha^{<t,t'>}$ to be generally larger for values of $a^{<t'>}$ that are highly relevant to the value the network should output for $y^{<t>}$
- $\sum_{t'} \alpha^{<t,t'>} = 1$

Q4) • Suppose you are building a speech recognition system, which uses a RNN model to map from audio clip x to a text transcript y. Your algorithm uses beam search to try to find the value of y that maximizes $P(y|x)$. On a dev set example, given an input audio clip, your algorithm outputs the transcript $\hat{y}$ = "I'm building a A Eye system in Silly con Valley", whereas a human gives a much superior transcript y^* = "I'm building an AI system in Silicon Valley". According to your model:

$P(\hat{y}|x) = 1.09 \cdot 10^{-7}$ $\quad P(y^*|x) = 7.21 \cdot 10^{-8}$

- You don't expect increasing beam width B to correct this example as $P(y^*|x) \leq P(\hat{y}|x)$ indicates the error should be attributed to the RNN rather than to the search algorithm

Scanned by CamScanner

- Continuing from Q4, suppose you work on your algorithm for a few weeks more, and now you find that the vast majority of examples on which your algorithm makes a mistake, $P(y^*|x) > P(\hat{y}|x)$. This suggest you should focus your attention on improving the search algorithm.

Scanned by CamScanner