

12/12/16

Advanced data analytics

- Unstructured data analytics (text, image, video, voice)
- What is deep learning?
- Role of data scientist
- Kaggle problems
- Syllabus

Image captioning, Text detection and retrieval, Image classification, (order than image classification), Object detection - identifies the objects in an image.

Deep learning everywhere - Internet and cloud

- Medicine & biology - cancer cell detection, diabetic grading, drug discovery, image classification, speech recognition, language translation, sentiment analysis
- Media and entertainment - video captioning, video search, real time translation
- Security and defense - face detection, video surveillance, satellite imagery
- Autonomous vehicles - Pedestrian detection, Lane tracking, Recognize traffic signs

Machine → Understand language } Unstructured data analysis
→ Vision
→ Intelligence

Sentiment analysis (Text based) → What do people feel about the new product? What is the movie review - the or -ve?
Twitter →
Facebook → New product

What is deep machine learning?

Traditional ML
Vision → Hand-crafted features → Your favorite "car" classifier
SIFT / HOG (learned)
(fixed) Artistic work of human

Speech → Hand-crafted features → Your favorite "deep" classifier
MFCC (fixed) (learned)

NLP → Hand-crafted features → Your favorite "+" classifier
(Bag-of-words) (learned)
(fixed) Artistic work of human

How to automate feature extraction?

Most important discovery: Human brain understands by recognizing in a hierarchical fashion, the features.

Neural networks - inspired from human brain.

Neural networks with more than three levels of depth are called as deep networks.

- Used to learn multiple levels of abstraction (ie, hierarchical features) or unstructured data formats. Low level → Mid level → High level
- Eliminates need for "artistic work" by humans

Data scientist to build the deep networks and "fine tune" the networks based on the problem. Fine tuning → error feedback

- No need for manual feature engineering, although some data preprocessing may be needed.

How many layers to build, what to do in each layer, how to fine-tune each layer?

(*) Automated feature extraction on "unstructured data" is challenging because there is no hierarchical learning.

Entries for Modern AI

Google - TensorFlow	(open-sourced)	→ Most popular
Facebook - Big Sur		High movement
Microsoft - CNTK		
Berkeley - CAFFE		

Choose about 3 deep models

- Recurrent Neural Network (RNN)
- Deep Neural Network (DNN)
- Convolutional Neural Network (CNN)

NLP

} Tensor flow supports all 3.

Image

To measure accuracy even in unstructured data, use 'validation' data.

With GPU

10 min

3 hrs

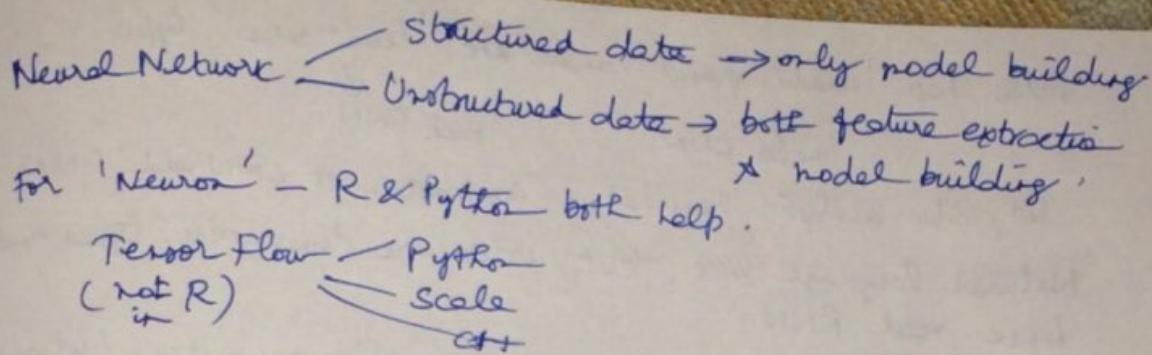
(only CPU)

Without GPU

4 hrs

1 week

} Time reduction with GPU



Perceptron, Neural networks are "objective-based" ML - Other
 objective-based are ~~Perceptron~~ & Neural networks Linear regression,
 Logistic regression, SVM
 Polynomial

13/12/16

- P1: Perceptron / Neuron
- P2: Neural Network
- P3: Neural Network for model building
- P4: Convolutional neural network
 - feature extraction in images { Image classifiers from Kaggle }
- P5: Recurrent neural network (LSTM)
 - feature extraction in text/natural language
 - (Sentiment classifier / Span classifier in Kaggle)
- P6: Pretraining → Auto Encoders → Deep belief network
 - (2006) to initialize neural networks

(Deep learning life cycle)

P7: Deployment of deep learning models

Python → Jupiter IDE → Anaconda distribution

"TensorFlow" is our framework of choice - uses 'distributed' framework automatically. If we use Spark, need to do many things manually.

- Perceptron
- Based on 'Optimization theory'
 - Gradient based approach (needed for CNN/RNN)
 - Calculus

GPU has fast algorithm for 'matrix multiplication' and hence
programmable NVIDIA GPU is preferred.

Digit recognizer - 99% accuracy with CNN.

Some problems need to be solved with a combination of CNN & RNN.

'Voice' can be analyzed by CNN / RNN.

"Tensorflow framework already has CNN & RNN libraries"

Tensorflow framework already has CNN & RNN libraries.

Spark is good for some simple ETL work.

Tensorflow allows us to deploy models to chip.

Video has 'image frame' and are 'time-series' type
↓ need CNN ↓ need RNN

So, video analysis requires a combination of CNN and RNN.

Natural language like text, voice are 'time-series' types and hence need RNN.

Object detection (where image is) and object recognition (what is in the image it is) are diff problems. In traditional approach, they were done separately. But in modern deep learning approaches, the two algorithms are implemented together.

14/12/16

iris-train.csv analysis

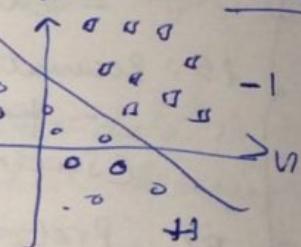
Perceptron
NN
CNN
RNN } All can be used for classification analytics

How to make machines learn some patterns? v2

M/C to be able to learn
this line to classify

Linearly separated
data

Automatic recognition of patterns is called
machine learning.

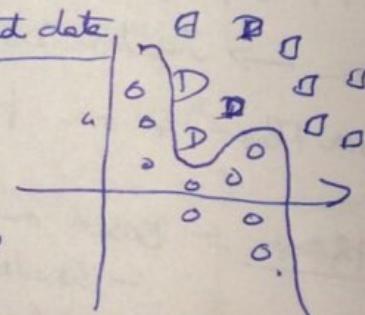


In this course, the machine learning approach is inspired by the human brain - Perceptron, Neural Network. There are many other approaches which are irrelevant to this course.

Perceptron

Can they do linear or non-linear classification/separation?

Non-linearly separated data

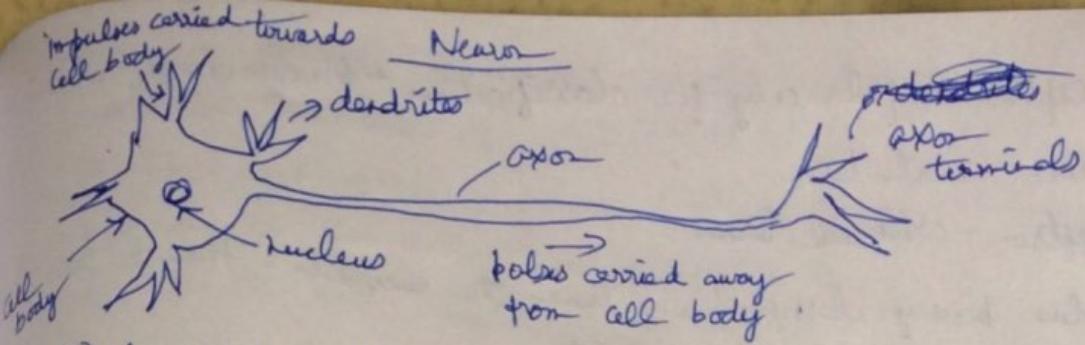


Neural networks are ultimately useful for "feature extraction". We see how to use perceptron/neuron for classification simply to lay the foundation for higher understanding.

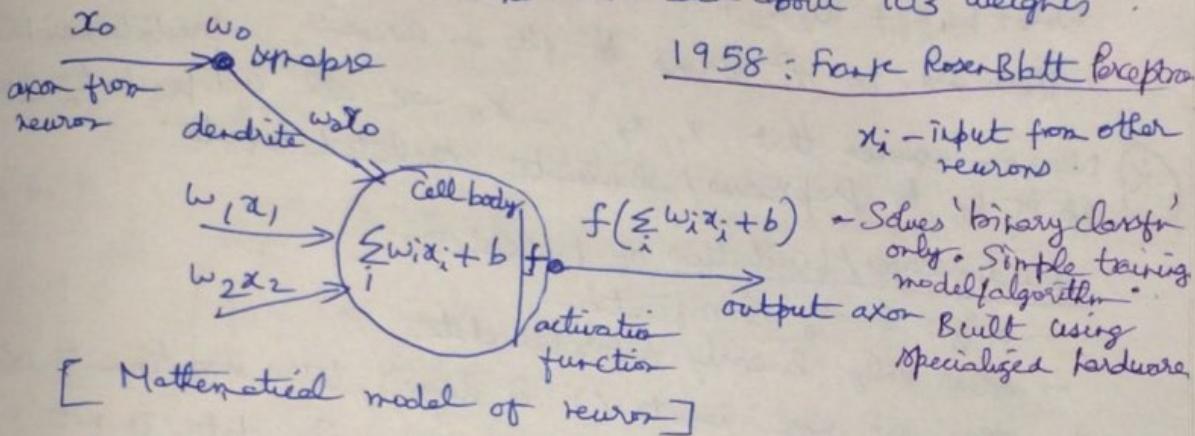
It is possible that trees will outperform neurons for simple classification.

Neuron - will fire/not fire based on threshold signal.
Fire \Rightarrow signals get transmitted out of a neuron and reach other neurons.

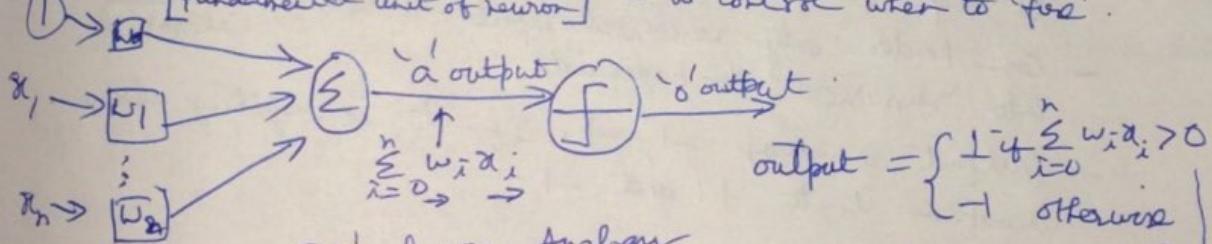
The same signal gets passed on to multiple neurons.



- Each neuron gets inputs from multiple neurons.
- How much each neuron benefits from another neuron is dependent on 'synaptic weights'.
- We have about 10^{10} neurons with each about 10³ weights.



- Frank's approach was to do a 'linear summation' = $(\sum_i w_i x_i)$. But actual neuron may do differently - this is working out well.
- 'b' is bias - send external signals to control 'internal state' of neuron [Fundamental unit of neuron] - to control when to 'fire'.



- Actually, synapses are more complicated than a single weight
- Neurons don't output a real number; instead they "fire" spikes at a (somewhat) regular rate
- Many different types of neurons
- Computations are not simply linear combinations of inputs transformed by the same activation function.

Using bias 'b' to do, output = $\begin{cases} 1 & \text{if } \sum_{i=0}^n w_i x_i - b \geq 0 \\ -1 & \text{if } \sum_{i=0}^n w_i x_i - b < 0. \end{cases}$

Challenge is to find the "weights" for a particular date.
 Assume $x_0 = 1$, $w_0 = b$ so that So, challenge is to

$$\left(\sum_{i=1}^n w_i x_i \right) + b = \sum_{i=0}^n w_i x_i$$

Perception useful only for classification - binary classification

Neuron - real term

Perceptron - artificial term

To solve binary classification, 'tune the weights' - How to tune?

- Start with random weights
- change weights based on data

Perceptron's binary classifier always solves "linear separation" problem successfully. This is because $w_0x_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n$ is the equation of an ~~line~~ n-dimensional plane.

There are no x^2 terms to fit a curve: predictor variables

 Neuron assumes that x_1, x_2, \dots, x_n are all independent. It is up to us to preprocess / eliminate correlated inputs.

Disadvantages / Limitations of perceptron

- only 'binary' classification
- solves only linearly separated data

When there are more inputs ('n' is large), takes long time to solve

- Can't find the 'best' possible cut when data is not linearly separated because there is no 'objective' fn. It just tries to classify everything perfectly. So, we need to tell it to stop after "some" iterations \rightarrow no logic to figure this out.
- Can handle only numerical inputs. Convert '~~continuous~~' data into 'numerical' to begin with.
- Output is always +1, -1. If it is anything else, transform it to +1 and -1.

15/12/16

Cell value $\geq \theta \rightarrow$ cell fires"

cell value $\pm \theta \geq 0$ - fires

(bias) $< \theta$ - does not fire

This is

Modelled as $\sum_{i=0}^n w_i x_i > \theta \rightarrow$ output is +1
 $\leq \theta \rightarrow$ output is -1

Learning some pattern in perceptron is equal to finding the right weight adjustments of a perceptron. The weights represent the learning in a perceptron.

Learning Rule

- 1) Assume random weights and random bias for perception

2) Do until all the samples are correctly classified or maximum numbers of epochs reached.

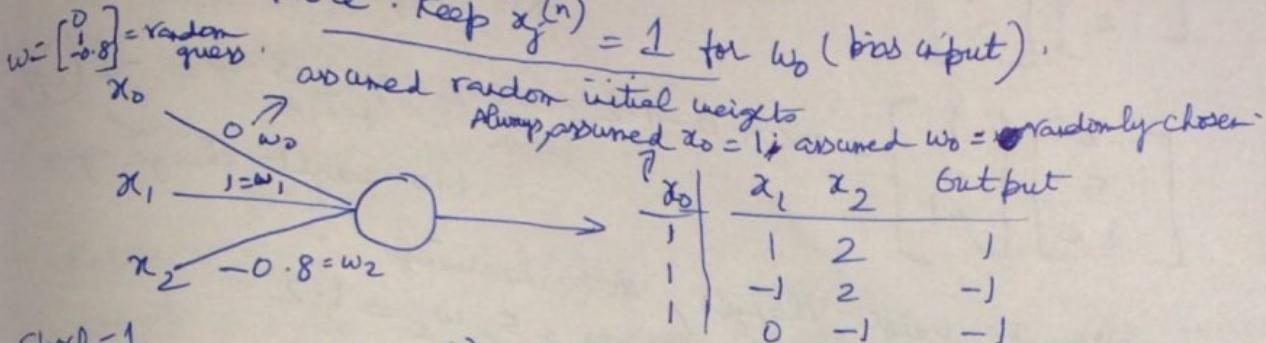
For each training sample $(x^{(n)}, y^{(n)})$

a) Compute the output of perceptron, \hat{o}

b) Update each weight, w_j , as follows:

$$w_j = w_j + \eta (y^{(n)} - \hat{o}^{(n)}) * x_j^{(n)}$$

Note: Keep $x_0^{(n)} = 1$ for w_0 (bias input).



Epoch-1

$$x_0 = 1 \quad x^{(1)} = x_0^{(1)}, x_1^{(1)}, x_2^{(1)}$$

b) ① $\begin{bmatrix} 0 \\ 1 \\ -0.8 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix} = 0+1-1 \cdot 6 = -0.6 = a$

$$f(-0.6) = -1$$

(Mismatch with out.)

$$w = \begin{bmatrix} 0 \\ 1 \\ -0.8 \end{bmatrix} + (1 - (-1)) \times \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}$$

$$= \begin{bmatrix} 2 \\ 3 \\ 3.2 \end{bmatrix} \quad \text{Refined new weights}$$

② $\begin{bmatrix} 2 \\ 3 \\ 3.2 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ 2 \end{bmatrix} = 2-3+6 \cdot 4 = 5.4 = a$

$$w = \begin{bmatrix} 2 \\ 3 \\ 3.2 \end{bmatrix} + (-1-1) \begin{bmatrix} 1 \\ -1 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 5 \\ -0.8 \end{bmatrix} \quad \text{Refined new weights}$$

③ $\begin{bmatrix} 0 \\ 5 \\ -0.8 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} = 0+0+0.8 = 0.8 = f(0.8) = 1 = f(a)$

$$w = \begin{bmatrix} 0 \\ 5 \\ -0.8 \end{bmatrix} + (-1-1) \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} = \begin{bmatrix} -2 \\ 5 \\ 1.2 \end{bmatrix} \quad \text{Refined new weights}$$

Epoch = One complete pass over the entire training set.

Epoch-2

$$\begin{bmatrix} -2 \\ 5 \\ 1.2 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix} = -2 + 5 + 2 \cdot 4 = \frac{5 \cdot 4}{a} = 1 \quad f(5 \cdot 4) = +1 \quad \text{net}(o/p_1)$$

So weights don't change
 $w_{new} = w_{old} + \theta [learning]$

$$\begin{bmatrix} -2 \\ 5 \\ 1.2 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ 2 \end{bmatrix} = -2 - 5 + 2 \cdot 4 = \frac{-4 \cdot 6}{a} = -1 \quad f(-4 \cdot 6) = -1 \quad \text{net}(o/p_2)$$

No weight change again

$$\begin{bmatrix} -2 \\ 5 \\ 1.2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} = -2 + 0 - 1 \cdot 2 = \frac{-3 \cdot 2}{a} = -1 \quad f(-3 \cdot 2) = -1 \quad \text{net}(o/p_3)$$

No weight change again.

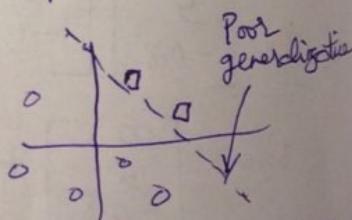
Since no weight changes, Stop 'learning'!
 Fixed learning is, $w_0 = -2, w_1 = 5, w_2 = 1.2$

Why the rule works?

- 1) If prediction is correct no change in weights obviously.
- 2) False +ve (target is -1 but predicted as +1)
 - $a > 0$, but we want ' a ' to be -ve to get correct prediction
 - So, we need to decrease the weights of the +ve inputs and increase the weights of -ve inputs
- 3) False -ve (target is +1 but predicted as -1)
 - $a < 0$ but we want ' a ' to be +ve for correct prediction
 - So, we need to increase the weights of +ve inputs and decrease the weights of -ve inputs.

Issues with threshold perceptron

- 1) Will converge only if data is linearly separable
- 2) It stops learning immediately if all samples are classified correctly during learning. The linear separator may not have the best generalization capability.



Python language

- Python types
 - Python data structures (Containers)
 - Control statements, User defined functions, Lambda expressions
 - Packages for ML
- Statistics
- NumPy
 - SciPy (Scientific Python)
 - SciLearn
 - Matplotlib
 - Pandas
 - TensorFlow
- Perceptron } Basic
- Lp NN
- Matrix/Vector multiplications

Other popular Python packages are CAFFE, Theano for ML

Download Python 2.7
(latest Python 3.x is not backward compatible)

UC Berkeley

16/12/16

- * Python - General purpose, high-level language created by Guido Van Rossum. Google uses Python heavily.
- * Python supports
 - scripting
 - object-oriented
 - imperative/structured
 - functional

Python Editors

Jupyter / iPython notebook ('Ipylab')

is good for code development. Good for presentations and sharing finished code

True IDEs - for regular code development

- Spyder (Scientific Python Development Environment) - we use this
- Envy Jupyter Notebook
- Python Tools for Visual Studio
- Enthought Canopy Editor
- Wakari, a hosted Python data analysis environment

Jupyter and Spyder are part of Anaconda distribution. Anaconda is a distribution of Python modules/ Packages

Jupyter - Supports 3 languages.

Julia ↓ Python ↓ R ↓

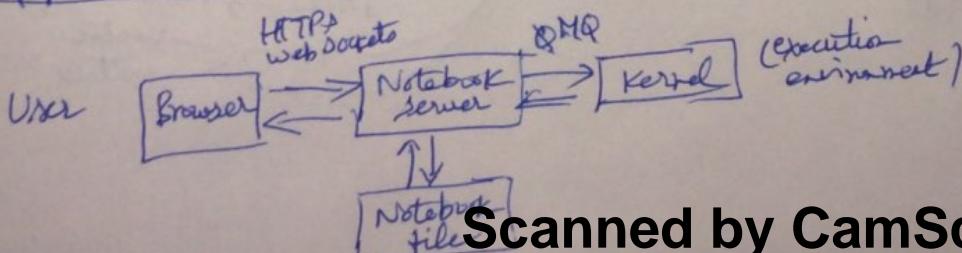
Literate programming - Explain program; write code. Coding to be natural like using a notebook.

Jupyter has 'R' kernels and 'Python' kernels, to interpret the languages.

Unlike 'R' work, we don't need to download packages from time to time. Anaconda distribution comes with 'all' packages. Only 'Tensorflow' needs to be separately installed.

Jupyter - old name is iPython (interactive Python). Now Jupyter does more than just Python.

Jupyter Notebook Components



Windows Environment

Shift + right-click > Open Command window here > "jupyter notebook"

In [1]:
 ↑
 Input `a = 10
print a`
 10

Code → Marks a cell as "Code"
 Markup → Marks a cell as "Comment"

One can write code in Spyder and publish using Jupyter.

To start Spyder,
 command prompt > "spyder"

"Variable explorer" — Workspace environment — can save only data.
 Spyder > Tools > Preferences

Why publish with Jupyter instead of GitHub?

- GitHub only to share code/comments
- Jupyter can store code/comments, the output, diagrams, many explorations. (People nowadays publish books/chapters using Jupyter)

R: ggplot :: Python: matplotlib

R: dataframes :: Python: Pandas

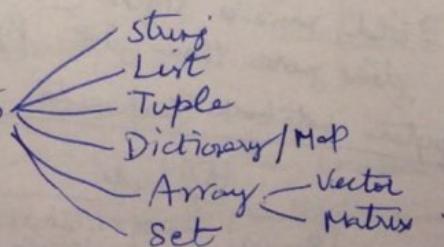
Python began as a general-purpose language and did not have many of R features for ML. Then, the general community developed many packages for ML in Python. Python has a more extensive Ecosystem/Community support than R.

Python is faster than R since it began as general-purpose
 'R Studio' and 'Spyder' IDEs are both inspired by "Visual Studio".

Python

- No explicit typing (int a - not needed - just use 'a')
- No explicit memory allocation/deallocation
- Performance poorer than Java/C/C++ but OK for interactive scripting environment

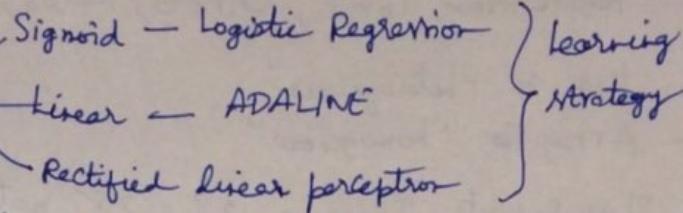
Python container/structures



19/12/16

Perceptron

- Other perception



- Limitations of perceptrons

- Neural Network

- Architecture
- Learning Strategy

- classification & regression model building

Feature characteristics for unstructured data

- Voice, Image, Natural language & Video

CNN (Convolutional NN)

- Principles
- Architecture
- Learning
- Applying CNN

Image

- RNN (Recurrent NN)
- Principles
- Architecture
- Learning
- Applying RNN

Sequence data /
Natural language

Python - If we change the value of a variable, its type will change. (eg) $x = 15$ (int) $\rightarrow x = 19.2$ (float)

Variable name can have '-' underscore but not '.'

List = An indexed container of "heterogeneous" elements

Module/ Package/ Namespace = group of related functions and objects

Tuple = immutable list ; cannot be modified after creation

21/12/16 Dictionary/ Map = Collection of key-value pairs {key: value}

Data Frame - not internally supported by Python

- use 'Pandas' package instead

Array - use 'NumPy' & 'SciPy' packages
- also not supported by basic Python

Set = unordered collection of distinct items { }

Digit Recognizer - used to identify numbers in 'images'.

22/12/16

Activation function

- Step function
- Sigmoid fn
- tanh

Loss function

- Squared loss
- Cross entropy loss

NN

Architecture

Learning

Back Propagation

Apply NN

Fine tuning of NN

CNN

Architecture

Ideas

Learning

Apply CNN

RNN

Memory

- 9) Tensor Flow, Torch, CAFFE, Keras, CNTK - Deep NN
 - List is "heterogeneous"
 - Array is "homogenous"
 Black & White Image = 28×28 matrix. Each cell value $\in [0, 255]$
 (Digit Recognizer)
 Color Image = 3 arrays of 28×28 matrix each.
 (R, G, B) . Each R/G/B cell value $\in [0, 255]$
 Input Image has 785 columns. $784 = 28 \times 28$.
 + 1 column for O/P $\in [0, 9]$.

We don't want 'array' to read train.csv for digit recognition because we want to consider O/P label as 'Categorical'. So, need to use "Dataframe".

"Digit Recognizer" open for 4+ years - waiting till we are able to get great prediction accuracy.

TensorFlow internally uses "Pandas" package; uses wrapper around it.

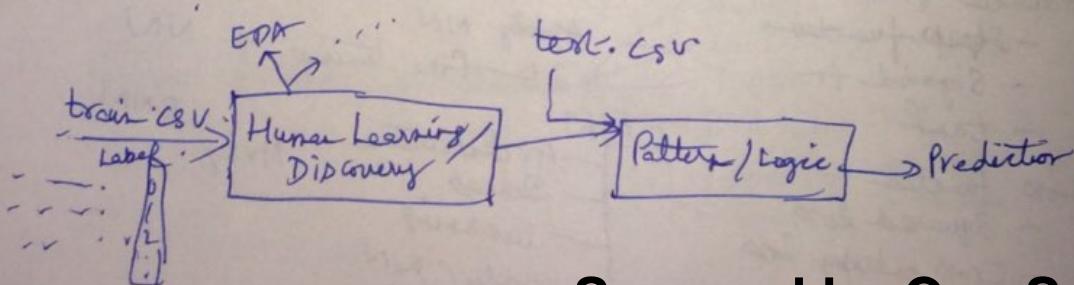
R: Vector :: Python: Numpy 1D-array (Homogenous Collection)

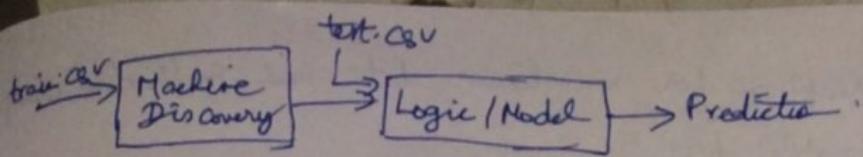
22/12/16

- Data frame - different columns are of different types.
- (R) DBMS = almost obsolete
- (H) DFS = presently used; distributed file system
- digit_train.describe() - mean, min, max, std, 25%, 50%, 75%
 - thus it tells about "central tendency" and "spread".
- Categorical variables not shown in describe() - Python handled categorical types very late.
 - Vectorization NOT by default on basic data types. Vectorization supported only in ~~not~~ np.array (from numpy package)

23/12/16

Perceptron
 Neural Network } Approaches for an m/c to learn & predict from brain data.

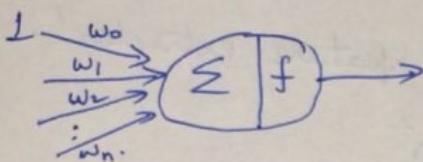




ML approaches

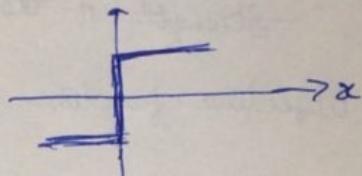
- 1) Information based learning (Tree)
- 2) Probabilistic learning (NB) Bayes
- 3) Ensembles
 - bagging
 - bagging + randomization
 - ← Boosting
- 4) Objective based learning
 - Support Vector Machine (SVM)
 - Linear regression
 - Logistic Regression → basic element of neural network; a type n
 - Perceptron { Speciality of this approach is "feature extraction"
 - Neural Network } Using them for classification to lay foundation

Perceptron / Neuron



Step function

$$f(x) = \begin{cases} 1, & x > 0 \\ -1, & x \leq 0 \end{cases}$$



Company offer

Package → Yes
good? → No.
(0.7)

Work environment → Good
(0.3)
Domain → OK
(0.5)
Not OK.

How to decide if I want to accept/reject an offer?

If $\sum w_i x_i > 2$, I accept offer
 ↑ "Threshold" parameter

For classification solution,
 $w_j = w_j + \eta (y - o) x_j$
 To learn weights

Memory + Analysis

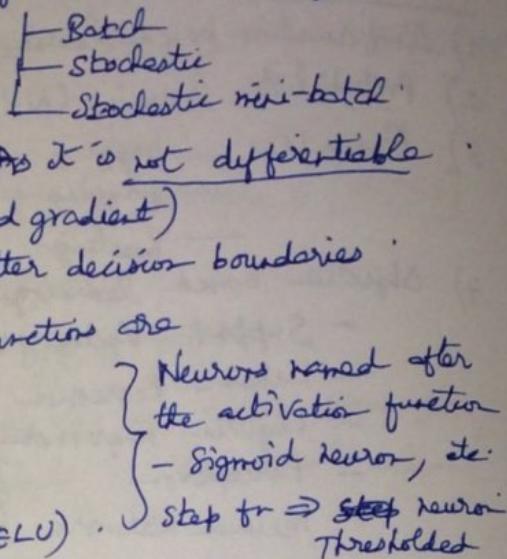
↳ Recurrent NN (Long Short-Term Memory - LSTM)

Perceptron problem for digit recognizer

- only binary classification (not multi-class classification)
- Works only for linearly separable data, no convergence
 - biggest drawback
 - Is it linearly separable in 784 dimensions?
 - 2nd issue → If linearly separable, how well does it separate? It stops learning immediately after a (sub-optimal) sol is found → Decision boundary not great.

How to adapt Perceptron for non-linear data and multi-class classifier?
For non-linearly separable data

- Modify objective function - use gradient descent algorithm
 - try to minimize objective for each w_i
- (X) - Step function to be changed - As it is not differentiable
 - Use differentiable fn (to find gradient)
 - makes Perceptron learn better decision boundaries
- Commonly used activation functions are
 - Sigmoid
 - tanh
 - Linear
 - Soft Max
 - Rectified linear unit (ReLU)



Sigmoid perceptron, tanh perceptron, linear perceptron, soft max perceptron etc.

Strength of these neurons is 'feature extraction'.

$$\text{Objective function} = \sum_{i=1}^n (y_i - o_i)^2$$
$$= \sum_{i=1}^n \left[y_i - f\left(\sum_{j=0}^n w_j x_j\right) \right]^2$$

For 'threshold perceptron', there is no objective function.
That's why there is no convergence.

24/12/16 Objective-based perceptron

- Needed to deal with non-linearly separable data (know when to stop)
 - To also find the ~~better~~ separation boundary - Though better than threshold perceptron, not optimal: linear boundary
 - Use squared loss as objective function for perceptron learning
- $$E = \frac{1}{2} \sum_{n=1}^N (y^{(n)} - o^{(n)})^2$$
- $$E(w) = \frac{1}{2} \sum_{n=1}^N \left[y^{(n)} - f(w x^{(n)}) \right]^2$$
- Vector notation
 $w = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix}$ $x = \begin{pmatrix} x_1 & x_2 & \cdots & x_n \end{pmatrix}$
 $wx = \text{dot product} = 1 \times 1 \text{ matrix}$
- find w that minimizes $E(w)$
 - solve using gradient descent
 - Solve using gradient descent
 - Batch gradient descent
 - Stochastic gradient descent
 - Stochastic mini-batch gradient descent
- $$w = w - n \times \text{gradient}$$
- $$= w - n \frac{\partial E}{\partial w}$$

Common activation functions

~~Linear~~ $g(2) = z$

~~ReLU; $g(a) = \max(0, a)$~~
(Rectified linear unit)

$$\begin{array}{c} 1.0 \\ \diagdown \\ 0.5 \end{array}$$

$$\text{Logistische (sigmoid) } g(z) = \frac{1}{(1+e^{-z})} \in [0,1]$$

1

Hyperbolic tangent (sigmoid) aka (tanh)

$$g(z) = \frac{e^{iz} - 1}{e^{iz} + 1} \in [-1, 1]$$

4

$$\text{Unit step } g(z) = \begin{cases} 1, & z \geq 0 \\ -1, & \text{otherwise} \end{cases}$$

15

$$g(z) = \begin{cases} 1, & z \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

} These are just smoothed versions of unit step functions

→ why e^{2z} instead of e^z ? Goal is not just $[-1, +1]$ but a smooth curve. Even step to less $[-1, +1]$ range

Perceptron Learning - BGD

- Assume random weights and random bias for perceptron
 - Do the following until error is below threshold or max. # epochs reached
 - a) Compute output of perceptron, O
 - b) Update each weight, w_j , of perceptron as follows :

$$w_j = w_j - \eta \frac{\partial E}{\partial w_j}$$

$$\Rightarrow O^{(n)} = g(w_0 x_0 + w_1 x_1 + \dots + w_j x_j + \dots + w_n x_n)$$

$$\eta = \sum w_i x_i = g(\sum w_i x_i)$$
 - c) $E = \sum_{i=1}^N (t^{(n)} - O^{(n)})^2$
 - d) $w_j = w_j + \eta \sum_{n=1}^N (g^{(n)} - O^{(n)}) x_j^{(n)} g'(x^{(n)})$ } Batch gradient
version.
 - e) $t^{(n)} = +1$ if $x_i^{(n)} > b^{(n)}$, $t^{(n)} = -1$ if $x_i^{(n)} < b^{(n)}$. Note : Keep $x_j^{(n)} = 1$ for w_0 (bias input).

Stochastic gradient version:

$$w_j = w_j + \eta (y^{(n)} - o^{(n)}) x_j^{(n)} g'(a)^{(n)}$$

This is only additional term
to Threshold perceptron.

\rightarrow Rule - (for differentiation)

$$\text{Rule: (for differentiation)} \quad \frac{\partial g}{\partial a} = \frac{\partial g}{\partial f_1} \frac{\partial f_1}{\partial x} \Rightarrow \frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial o^{(n)}} \frac{\partial o^{(n)}}{\partial w_i} = \frac{\partial E}{\partial o^{(n)}} \frac{\partial o^{(n)}}{\partial a} \frac{\partial a}{\partial w_i} = g(z)(1-g(z))$$

$$\text{if } g(z) = \frac{1}{1+e^{-z}} \Rightarrow g'(z) = \frac{e^{-z}}{(1+e^{-z})^2}$$

x_0	x_1	x_2	output(y)
1	1	2	1
1	-1	2	0
1	0	-1	0

$$\frac{\partial a}{\partial w_i} = \frac{\partial (\sum w_i x_i)}{\partial w_j} = x_j^{(n)}$$

$$\frac{\partial w_i}{\partial \theta^{(n)}} = \frac{\partial g(\alpha)}{\partial \theta} = g'(\alpha)^{(n)}$$

$$\frac{\partial \sigma}{\partial E} = \frac{1}{2} \sum (y^{(i)} - \hat{y}^{(i)})^2$$

$$w = \begin{bmatrix} 0 \\ -0.8 \end{bmatrix} + (1-0.35) \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix} \stackrel{(0.35)}{\rightarrow} \stackrel{(1-0.35)}{\rightarrow} g'(z) = g(z)(1-g(z))$$

$g(a) = g_0$

$$a = \sum w_i x_i = \begin{bmatrix} 0 \\ 1 \\ -0.8 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix} = -0.6.$$

$$\stackrel{(n)}{O} = g(a) = \frac{1}{1+e^{-a}} = \frac{1}{1+e^{0.6}} = 0.35$$

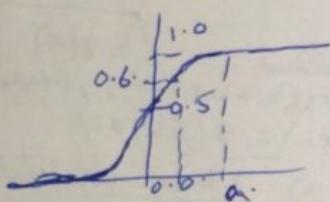
$$\therefore w = w + (y - g(a)) \alpha g(a)(1-g(a)), \text{ Assuming } n=1.$$

$$y_1 = 1.$$

So, in epoch 1, for 1st sample

$$\begin{aligned} w &= \begin{bmatrix} 0 \\ 1 \\ -0.8 \end{bmatrix} + (1-0.35) \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix} \stackrel{(0.35)}{\rightarrow} \stackrel{(1-0.35)}{\rightarrow} \\ &= \begin{bmatrix} 0 \\ 1 \\ -0.8 \end{bmatrix} + (0.147) \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 0.147 \\ 1.147 \\ -0.504 \end{bmatrix} \end{aligned}$$

Informal logic of sigmoid



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

$\therefore z$ is evidence $= a = \sum w_i x_i$.
only if evidence is high enough (threshold),
then fire neuron, or, switch your job. like 0.6

So, sigmoid perceptron is more practical/appels to intuition
than threshold perceptron

When to stop learning in Sigmoid perceptron?
Compute Error $E = \sum_{i=1}^n (y_i - \hat{y}_i)^2$ at the end of each epoch.

If $\left| \frac{\text{Error after epoch } i}{\text{Error after epoch } i+1} - 1 \right| > \text{threshold}$, then stop.

After the stop happens and learning is done, we decide
whether to use > 0.7 as +1, ≤ 0.7 as 0 } ie, what step to
(or) > 0.3 as -1, ≤ 0.3 as 0 } to use.

Objective functions - 3 types

- Squared error loss
- Cross entropy loss
- Log loss

Limitations of Perceptron

- Can only perform binary linear classification threshold perceptron
- Although they find better boundary than

- Solution: Use multiple perceptrons for multiple classifications
K-perceptrons for K-class classifications

- Solution 2 : Use multiple layers of perceptrons but with different non-linear activation functions
with backerrors (for non-linear separation)
- Slower learning, due to squared error objective. Use cross-entropy objective for faster learning.

perceptrons to solve Titanic problem - convert to all continuous variables.

In multi-class classification with perceptrons, convert $y = [0, 9]$ range w.r.t digit-recognizer into (y_0, y_1, \dots, y_9) by one-hot encoding.

for Perceptron 0, target is y_0 .

for Perceptron 1, target is y_1 .

for Perceptron 9, target is y_9 .

Finally, overall error = sum of errors of each perceptron.

26/12/16

Meaning of firing

- Backpropagation connection has a weight and the neuron simply sums up all the weighted inputs
- Weight of impact
 - Features with +ve weights cause evidence to increase
 - " -ve " " decrease

Perceptron for decision making

Gives a job offer, give weights for diff. parameters & set a threshold. If weighted score \geq threshold, accept job offer.

Two perspectives of activation fns

- * Model perspective - interpretation of activation fn curves based on evidence vs outcome
- * Learning perspective - Some activation fns provide much smoother learning compared to others.

Model Perspective

- { Step fn \rightarrow Switch job or don't switch job if evidence ≥ 0 or < 0 .
- { Sigmoid fn \rightarrow Switch job with a confidence level of 0.7 if evidence = 0.5. 0.5 if evidence = 0.
- Hence more preferred than step fn.

Learning perspective

Dif. activation fns have diff. slopes. So, if w_j (learning) changes slightly, and the outcome changes slightly, then we can fine-tune w_j to get slightly different outcomes.

Sigmoid - small change in weight (or bias) causes small change in output

Sigmoid \rightarrow learns very slowly even with initial big mistake using squared error loss objective function

$$w = w + (y - g(a)) \times g(a)(1-g(a))$$

If my required output $y=1$, but initially got $g(a)$ output = 0.1.

Then $g(a)(1-g(a)) = 0.9(0.1)$

$$y - g(a) = 0.9$$

$$\text{So, } w = w + 0.9 \times 0.9(0.1)$$

$w = w + 0.081 \times \Rightarrow \text{very slow learning.}$

Overcome slow-learning in sigmoid perceptron w/ cross-entropy objective.

$$E = -\frac{1}{N} \sum_{n=1}^N [y^{(n)} \log \hat{o}^{(n)} + (1-y^{(n)}) \log (1-\hat{o}^{(n)})] \Rightarrow \begin{cases} \text{If } y=0, \hat{o}=0, \\ E=\infty \\ \text{Also, if } y=1, \hat{o}=1, \\ E=0. \end{cases}$$

Like squared error, always the error $\cdot y^{(n)} = 0$ or 1.

Output $\hat{o} \in [0, 1] \Rightarrow \log 0 < 0 > \log(1-0) < 0$.

Intuitive meaning: Since output $\in [0, 1]$, error is always tan.

- Cost tends to be close to zero if \hat{o}/p approaches actual value.

$y \neq 0$ always have only 0, 1.

Now, $w_j = w_j - n \frac{\partial E}{\partial w_j}$ {Stochastic gradient version}

$$= w_j + n (y^{(n)} - \hat{o}^{(n)}) x_j^{(n)}$$

So, we've removed the slow learning part of $g(a)(1-g(a))$.

For linear perceptron, squared error is good.

For sigmoid perceptron, cross-entropy error is good.

How?

Stochastic-gradient version of cross-entropy function

$$E = -\frac{1}{N} [y^{(n)} \log \hat{o}^{(n)} + (1-y^{(n)}) \log (1-\hat{o}^{(n)})]$$

$$\frac{\partial E}{\partial w_j} = -\frac{1}{N} \left[\frac{y^{(n)} \hat{o}'^{(n)}}{\hat{o}^{(n)}} + \frac{(1-y^{(n)}) (-1) \hat{o}'^{(n)}}{1-\hat{o}^{(n)}} \right]$$

$$= -\frac{1}{N} \left[\frac{y^{(n)} \hat{o}'^{(n)} - y^{(n)} \hat{o}'^{(n)} / \hat{o}^{(n)} + y^{(n)} \hat{o}'^{(n)} \hat{o}'^{(n)} - \hat{o}^{(n)} \hat{o}'^{(n)}}{\hat{o}^{(n)} (1-\hat{o}^{(n)})} \right]$$

$$= -\frac{1}{N} \cdot \frac{(y^{(n)} - \hat{o}^{(n)}) \hat{o}'^{(n)}}{\hat{o}^{(n)} (1-\hat{o}^{(n)})} = -\frac{1}{N} \frac{(y^{(n)} - \hat{o}^{(n)}) g'(a) x_j}{g(a) [1-g(a)]}$$

$$= -\frac{1}{N} [y^{(n)} - \hat{o}^{(n)}] x_j \text{ because for sigmoid perceptron, we know that } g'(a) = g(a)(1-g(a))$$

$$\text{So, } w_j = w_j - n \frac{\partial E}{\partial w_j}$$

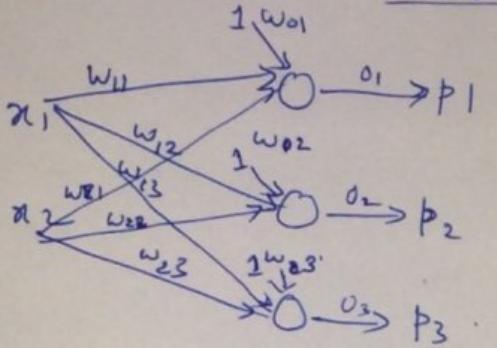
$$= w_j + \frac{n}{N} (y^{(n)} - \hat{o}^{(n)}) x_j$$

So, we can rewrite

$$w_j = w_j + n (y^{(n)} - \hat{o}^{(n)}) x_j \quad \left[\because \frac{n}{N} \text{ can be absorbed into diff. } n \right]$$

27/12/16

- Use sigmoid perceptron with cross-entropy objective.
- Use K sigmoid perceptrons for K-class classification



E = Overall error = Sum of errors.

Single Layer Neural Network

Sigmoid with Xentropy

cross-entropy
(act_y)
negative log

Convert 'y' with K-levels into a vector of size K using one-hot encoding

$$E = \sum_{n=1}^N \sum_k -y_k \log \sigma_k - (1-y_k) \log (1-\sigma_k) \quad .$$

$$w_{ij}^{(n+1)} = w_{ij}^{(n)} - \eta \frac{\partial E}{\partial w_{ij}^{(n)}} = w_{ij}^{(n)} + \eta \left(t_j^{(n)} - o_j^{(n)} \right) x_i^{(n)}$$

Note: keep $x_i^{(n)} = 1$ for w_{0j} (bias input)

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} + \frac{\partial o_j}{\partial a_j} + \frac{\partial a_j}{\partial w_{ij}} \quad \frac{\partial a_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \left(\sum_k w_{kj} x_k \right) = x_i$$

$$\frac{\partial \phi_j}{\partial a_j} = g'(a_j)$$

$$\frac{\partial G}{\partial o_j} = -1 \times \frac{\partial}{\partial o_j} \left(\sum_k y_k \log o_k + (1-y)_k \log (1-o_k) \right)$$

$$= - \frac{\partial}{\partial o_j} \left(y_j \log o_j + (1-y_j) \log (1-o_j) \right)$$

$$= - \left(\frac{y_j}{o_j} + \frac{(1-y_j)(-1)}{1-o_j} \right) = \frac{1-y_j}{1-o_j} \cdot \frac{-y_j}{o_j} = \frac{o_j - y_j}{o_j(1-o_j)}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{o_j - y_j}{o_j(1-o_j)} \times g'(a_j) * x_{ik} = \frac{o_j - y_j}{g(a_j)(1-g(a_j))} * g'(a_j) * x_{ik}$$

$$= \frac{0_j - y_j}{g(a_j) [1 - g(a_j)]} + g(a_j) [1 - g(a_j)] + x_i$$

$$= (\underline{0_j - y_j}) x_i$$

$$\text{So, } w_{ij} = w_{ij} - n (\bar{o_j} - \bar{y_j}) x_i$$

We get K outputs o_1, o_2, \dots, o_K . If $\sum_i o_i = 1$, then we are good to choose the highest o_i and say the final classification is ' i '.

But $\sum o_i + 1$ with sigmoid perceptrons
 so, change sigmoid to 'softmax'. Softmax useful only for
 Layer of perceptrons

\otimes Softmax = Normalised Logits $P_i = \exp(\alpha_i) / \sum_j \exp(\alpha_j)$ $\sum_i P_i = 1$

28/12/16 How to overcome limitation of 'only linear classification'?

- Add hidden layers of perceptrons for non-linear classification.

(*) Why not use just 2 perceptrons for 4-class classifier?

$00, 01, 10, 11 \Rightarrow 4$ possible combined output

- This won't work because of lack of interpretability of
 $01, 10, \text{etc}$.

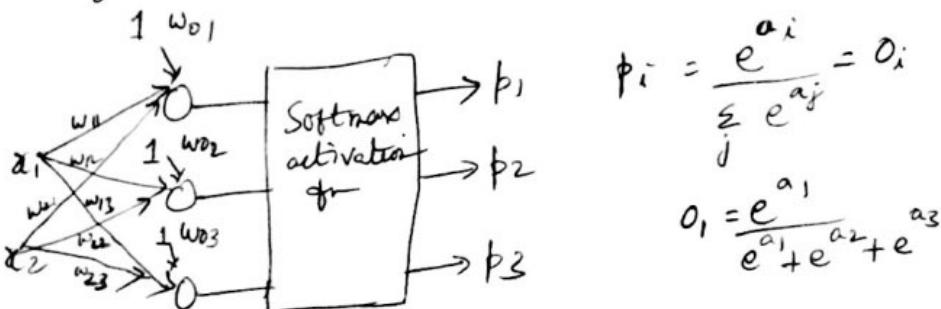
(*) Why not use 2 perceptrons for 2-class (binary) classification?

- Actually, using 2 perceptrons is better than using only 1
perceptron because there is more learning/interpretability.
But 1 perceptron used only for simplicity.

(*) Activation fns 'tanh' and 'ReLU' are used in 'hidden layers' of
perceptrons.

(*) For regression, only one output perceptron with 'linear' ~~activation~~
activation function and 'squared error' objective function.
- No 'slow learning' issue because activation fn is 'linear' and not
sigmoid; So, $g'(a) = 1$: ($\because g(a) = a$)
only in sigmoid, $g'(a) = g(a)(1-g(a)) \Rightarrow$ slow learning.

(*) Cross-entropy objective fn makes sense only for classification; it is
meaningless for regression.



Note: Softmax is not exactly 'normalized sigmoid'.

$$\begin{aligned} \text{Normalized sigmoid} &= \frac{1}{1+e^{-a_i}} = \frac{e^{a_i}}{e^{a_i}+1} \\ &= \frac{\sum_k \frac{1}{1+e^{-a_k}}}{\sum_k \frac{1}{1+e^{-a_k}}} = \frac{\sum_k \frac{e^{a_k}}{e^{a_k}+1}}{\sum_k \frac{e^{a_k}}{e^{a_k}+1}} \\ &= \frac{e^{a_i}}{(e^{a_i}+1) \sum_k e^{a_k}} \end{aligned}$$

$$\text{But Softmax} = \frac{e^{a_i}}{\sum_k e^{a_k}}$$

Why Softmax instead of 'Normalized Sigmoid'?

In normalized sigmoid, we normalize individual S curve
values, which may not be 'meaningful'. Better to use
 S' curve-like Softmax function.

29/12/16
Softmax fn: Sigmoid for multi-classes [Uses cross entropy]

$$\text{Softmax output } O_j = g(a_j) = \frac{e^{a_j}}{\sum_k e^{a_k}}$$

* Properties of softmax perception.

$$- 0 < O_j < 1$$

$$- \sum_k O_k = 1$$

* Individual sigmoid units also provide values in [0,1] range but the summation of o/p sigmoid units may not equal to 1.

* Learning algorithm is the same as that of sigmoid activation fn because the partial derivative of softmax is $g(a_j)(1-g(a_j))$.

In both sigmoid & softmax,

$$\frac{\partial E}{\partial O_j}$$
 are the same since 'E' is cross-entropy in both cases

$$\frac{\partial a_j}{\partial w_{ij}} = x_i, \text{ same in both fns.}$$

$\frac{\partial O_j}{\partial a_j}$ needs to be computed separately, but in both cases,

$$\frac{\partial O_j}{\partial a_j} = g'(a_j) = g(a_j)(1-g(a_j))$$

⇒ Both have the same learning algorithm.

$$\text{In Softmax, } O_j = \frac{e^{a_j}}{\sum_k e^{a_k}} = g(a_j)$$

$$\left[\frac{d}{dx}(u_j) = \frac{u'_j - u_j}{v^2} \right]$$

$$\therefore \frac{\partial O_j}{\partial a_j} = g'(a_j) = \frac{e^{a_j}(\sum_k e^{a_k}) - e^{a_j}(e^{a_j})}{[\sum_k e^{a_k}]^2}$$

$$= \frac{e^{a_j}}{\sum_k e^{a_k}} \left[\frac{\sum_k e^{a_k} - e^{a_j}}{\sum_k e^{a_k}} \right]$$

$$= \frac{e^{a_j}}{\sum_k e^{a_k}} \left[1 - \frac{e^{a_j}}{\sum_k e^{a_k}} \right]$$

$$= g(a_j)(1-g(a_j)).$$

X		Output
x_1	x_2	
1	2	c_1
-1	2	c_2
0	1	c_3

one-hot encoding

x_0	x_1	x_2	y_1	y_2	y_3
1	1	2	1	0	0
1	-1	2	0	1	0
1	0	1	0	0	1

Epoch 1

$$\begin{bmatrix} w_{00} & w_{11} & w_{21} \\ w_{01} & w_{12} & w_{22} \\ w_{02} & w_{13} & w_{23} \end{bmatrix}$$

$$= \begin{bmatrix} 1 & w_{11}' & w_{21}' \\ w_{02}' & w_{12}' & w_{22}' \\ w_{03}' & w_{13}' & w_{23}' \end{bmatrix} = \begin{bmatrix} w_{01} & w_{11} & w_{21} \\ w_{02} & w_{12} & w_{22} \\ w_{03} & w_{13} & w_{23} \end{bmatrix} + I(a-b) \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}$$

Add this bias to the matrix

More convenient If we had taken the transpose, we would have added the RHS term column-wise

Scanned by CamScanner

28/12/16 How to overcome limitation of 'only linear classification'?

- Add hidden layers of perceptrons for non-linear classification.

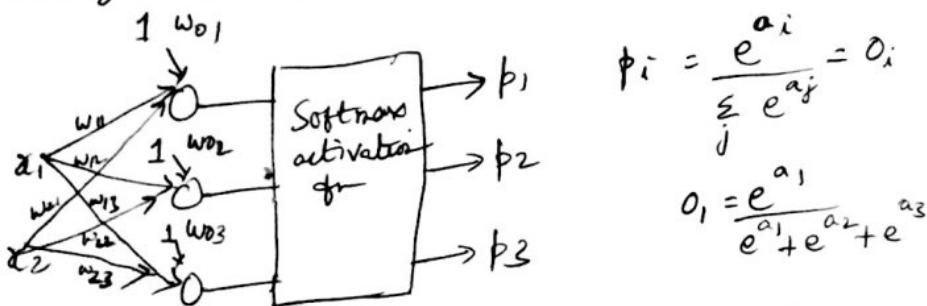
* Why not use just 2 perceptrons for 4-class classification?
 $00, 01, 10, 11 \Rightarrow 4$ possible combined output.
- This won't work because of lack of interpretability of
 $01, 10, \text{etc.}$

* Why not use 2 perceptrons for 2-class (binary) classification?
- Actually, using 2 perceptrons is better than using only 1 perceptron because there is more learning/interpretability.
But 1 perceptron used only for simplicity.

* Activation fns 'tanh' and 'ReLU' are used in 'hidden layers' of perceptrons.

* For regression, only one output perceptron with 'linear' activation function and 'squared error' objective function.
- No 'slow learning' issue because activation fn is 'linear' and not sigmoid; So, $g'(a) = 1$. ($\because g(a) = a$)
Only in Sigmoid, $g'(a) = g(a)(1-g(a)) \Rightarrow$ slow learning.

* Cross-entropy objective fn makes sense only for classification; it is meaningless for regression.



Note: Softmax is not exactly 'normalized sigmoid'.

$$\text{Normalized sigmoid} = \frac{1}{1+e^{-a_i}} = \frac{e^{a_i}}{e^{a_i}+1}$$

$$= \frac{\frac{1}{1+e^{-a_k}}}{\sum_k \frac{1}{1+e^{-a_k}}} = \frac{\sum_k \frac{1}{1+e^{-a_k}}}{\sum_k \frac{e^{a_k}}{e^{a_k}+1}}$$

$$= \frac{e^{a_i}}{(e^{a_i}+1) \sum_k \frac{e^{a_k}}{e^{a_k}+1}}$$

$$\text{But Softmax} = \frac{e^{a_i}}{\sum_k e^{a_k}}$$

Why Softmax instead of 'Normalized Sigmoid'?
In normalized sigmoid, we normalize individual a_i 's curve values, which may not be meaningful. Better to use 'S' curve-like Softmax function.

~~softmax~~: Sigmoid for multi-classed [Uses cross entropy]

Softmax output $O_j = g(a_j) = \frac{e^{a_j}}{\sum_k e^{a_k}}$

Properties of softmax perception:

- $0 < O_j < 1$
- $\sum_k O_k = 1$

Individual sigmoid units also provide values in $[0, 1]$ range but the sum total of o/p sigmoid units may not equal to 1.

Learning algorithm is the same as that of sigmoid activation fn because the partial derivative of softmax is $g(a_j)(1-g(a_j))$.

In both sigmoid & softmax,

$\frac{\partial E}{\partial O_j}$ are the same since 'E' is cross-entropy in both cases

$$\frac{\partial a_j}{\partial w_{ij}} = x_i, \text{ same in both fns.}$$

$\frac{\partial O_j}{\partial a_j}$ needs to be computed separately, but in both cases,

$$\frac{\partial O_j}{\partial a_j} = g'(a_j) = g(a_j)(1-g(a_j))$$

\Rightarrow Both have the same learning algorithm.

$$\text{In softmax, } O_j = \frac{e^{a_j}}{\sum_k e^{a_k}} = g(a_j)$$

$$\left[\frac{d}{dx} \left(\frac{u}{v} \right) = \frac{u'v - uv'}{v^2} \right]$$

$$\begin{aligned} \therefore \frac{\partial O_j}{\partial a_i} &= g'(a_j) = \frac{e^{a_j} \left(\sum_k e^{a_k} \right) - e^{a_j} (e^{a_i})}{\left[\sum_k e^{a_k} \right]^2} \\ &= \frac{e^{a_j}}{\sum_k e^{a_k}} \left[\frac{\sum_k e^{a_k} - e^{a_j}}{\sum_k e^{a_k}} \right] \\ &= \frac{e^{a_j}}{\sum_k e^{a_k}} \left[1 - \frac{e^{a_j}}{\sum_k e^{a_k}} \right] \\ &= g(a_j)(1-g(a_j)). \end{aligned}$$

x		Output			
x_1	x_2	c_1	x_0	x_1	x_2
1	2	c_1	1	1	2
-1	2	c_2	1	-1	2
0	1	c_3	1	0	1

one-hot encoding \rightarrow

$$\text{Epoch 1} \quad \begin{bmatrix} w_{10} & w_{11} & w_{12} & w_{13} \\ w_{20} & w_{21} & w_{22} & w_{23} \end{bmatrix} = \begin{bmatrix} w'_{01} & w'_{11} & w'_{21} \\ w'_{02} & w'_{12} & w'_{22} \\ w'_{03} & w'_{13} & w'_{23} \end{bmatrix} = \begin{bmatrix} w_{01} & w_{11} & w_{21} \\ w_{02} & w_{12} & w_{22} \\ w_{03} & w_{13} & w_{23} \end{bmatrix} + I(a-b) \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}$$

Add this row wise to the matrix

More convenient If we had taken the transpose we would have added the RHS term column-wise

We thus see a lot of matrix-like computations needed in perceptron learning. Since GPU has specialised hardware for matrix computations, it makes calculations faster for 'Deep learning'.

If Softmax AF + X-entropy objective,

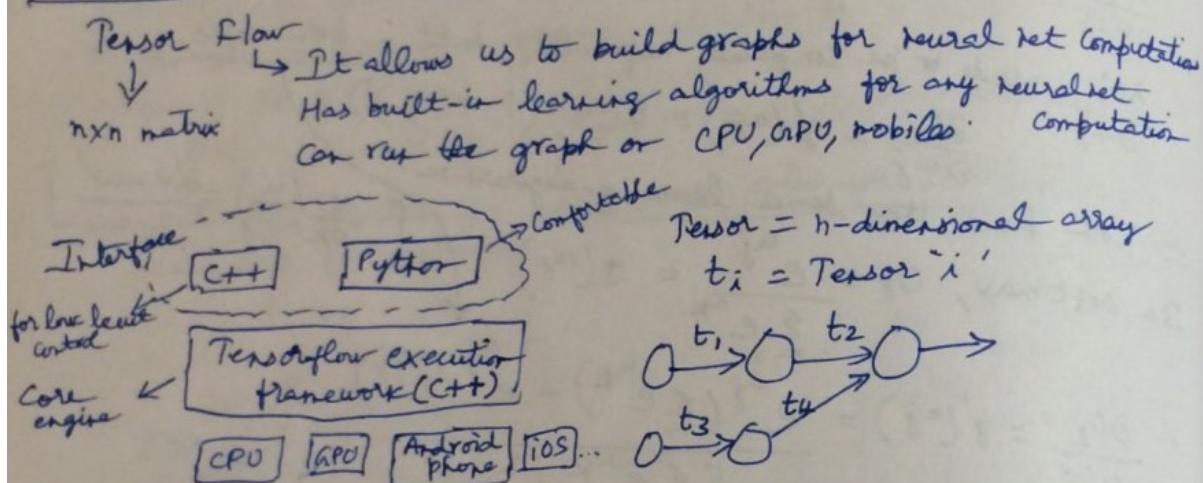
Shuffle train data before each 'epoch' begins and use 'Stochastic gradient descent'. Shuffling enables randomisation in stochastic gradient version.

After all the learning for 3-class classifier, suppose we try to predict for 'test' input. Softmax o/p = $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ or $(\frac{2}{5}, \frac{2}{5}, \frac{1}{5})$.

Now we have 'tie' \Rightarrow Randomly choose 'class'.

If Softmax o/p = $(0.5, 0.3, 0.2) \Rightarrow$ class C1
 $(0.3, 0.5, 0.2) \Rightarrow$ class C2
 $(0.2, 0.3, 0.5) \Rightarrow$ class C3

30/12/16



In this course, we will use 'Python' Interface.
Execution framework built using C++ for high performance.

'Tensorflow' inspired by 'Theano' (has Python interface too)

CAFFE framework - config driven - outdated

Numpy - allows to build arrays, but has no learning algorithms.

For Windows Tensorflow - Install VirtualBox \rightarrow install CentOS 7

Tensorflow on top of this.

CentOS 7 \rightarrow Community enterprise - close to Redhat Linux

VirtualBox \rightarrow To install Linux on windows machine

Anaconda for Linux - Install Python 3.5 version (64-bit installer)
- Complaints with 2.7 version.

$$\begin{aligned} a_1x + a_2y &= 10 \\ b_1x + b_2y &= 20 \end{aligned} \quad \begin{bmatrix} a_1 & a_2 \\ b_1 & b_2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 10 \\ 20 \end{bmatrix}$$

- This is why we use matrix multiplication. For concise expression of equation.

Power of matrix multiplication can be seen in perceptron learning
- for doing weight updates.

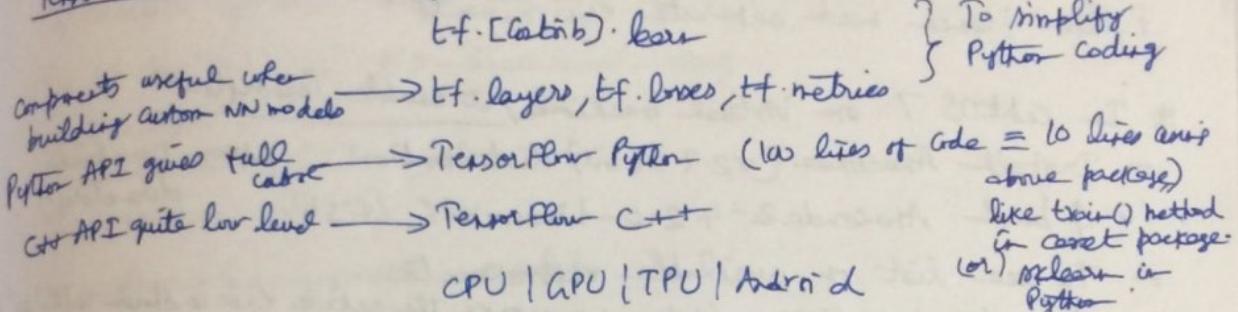
31/12/16 Python code for dataframes - Titanic
" images - Coffee

Digit Recogniser - No manual feature extraction.
Use neural network to extract features automatically

> bash script file

2/1/17 Root password - root123

TensorFlow toolkit hierarchy



Why install CentOS 7.0 and not 6.5?

Tensorflow needs 'glibc' version 2.17 and above but CentOS 6.5 has only glibc version 2.14. Very important reason for using CentOS 7.0.

TensorFlow very good for deep learning. Free license.

Why TensorFlow?

- CAFFE mostly config driven
- TensorFlow & Theano provide programming control - more control.
Also, they support Python
- Torch good but requires learning a new language called "Lua".
Hence, TensorFlow is best.

Tensorflow graphs have nodes and edges

- Nodes: The operations we do, e.g., softmax computation
- Edge: Input for nodes = tensors ($n \times n$ arrays).

Data move from one edge to edge.

Tensors (arrays) are flowing through the graphs of nodes & edges?

Do we train (after building the layers) this network on a CPU or GPU or mobile etc?

- Can learn on any hardware. Very big advantage here using TensorFlow. In other frameworks, we need to write extra code to train on GPU. Mobile deployment also very simplified.

Also easy option to say, "run my TensorFlow on Google's cluster". (Cloud)

We can write code in any layer (package, Python, C++) and still run it on Google cluster.

In linux, default anaconda (v2.7 64 bit) installation comes with the 'Conda' environment. But we want to create a new 'TensorFlow' environment.

TensorFlow comes with its own packages as well as its own version of Python. So, to avoid mix-up, cannot just install extra packages. Need a separate environment - with its own spyder, jupyter - install only the ~~the~~ "required" Python packages.

To install 'Tensor' in the future, create a separate environment. Regular Python may need some latest versions and 'Tensorflow' may need some older versions of Python packages/libraries. So, mixing both will cause problems; also cannot revert from Tensorflow to default Python. Hence need separate environments.

* In centOS 7 or virtual machine, to enable Tensorflow

- Install Anaconda (v2.7 64 bit) distribution (from 'Downloads' directory)
\$ bash Anaconda2-4.2.0-Linux-x86_64.sh.
- To see list of available environments:
\$ conda env list . (After many envs, the active env. is shown with *)
- To create separate environment for tensorflow:
\$ conda create -n tensorflow Python=2.7
- Activate the tensorflow environment
\$ source activate tensorflow
- Deactivate the tensorflow environment
\$ source deactivate tensorflow

[Conda is a tool for managing and deploying applications, environments and packages]

- To install tensorflow, notebook and spyder in tensorflow environment
 - Conda install -c conda-forge tensorflow
 - conda install notebook ipykernel
 - conda install spyder

To enable internet access on conda OS VM,

- become a super user (\$su) and give root password
- vi /etc/sysconfig/network-scripts/ifcfg-enp0s3
 - change 'onboot' value from 'no' to 'yes'

To check which version of spyder or jupyter is active

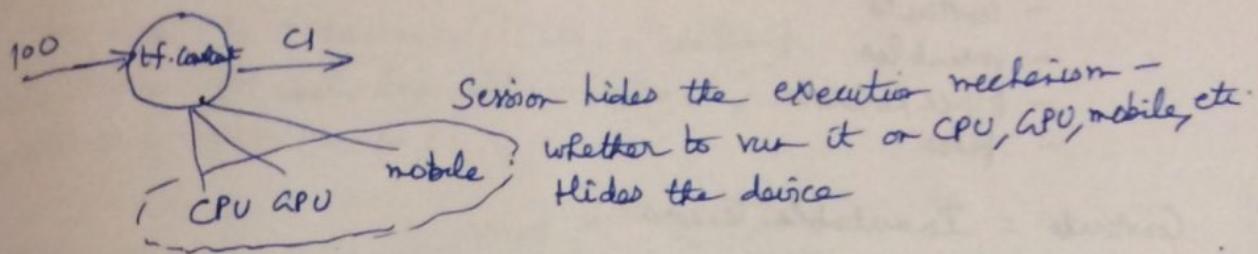
- which spyder
- which jupyter

To invoke the IDEs

- \$ Spyder
(or)
- \$ Jupyter notebook

3/1/2017

Tensorflow execution



Tensorflow - always construct a graph - Not a regular programming model.
 operator = Vertex
 tensor = Edge

To execute graph, always open session, execute and close.

$c1 = \text{tf.constant}(100)$ # 0-dimensional array

0-dimension = 10

1-dimension = [-]

2-dimension = [- -]

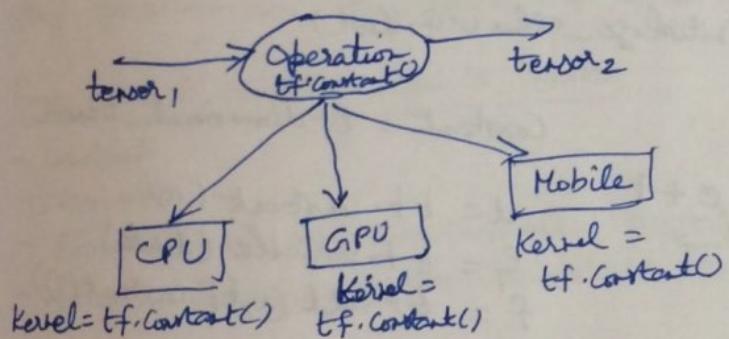
Tensorflow - Lazy execution model

Kernel = Implementation of operations on a specific device

Operations = Abstract representation for functionality

TF Operations

`tf.add()`
`tf.multiply()`
`tf.subtract()`



Construction and execution of graph are 2 separate processes.

Open session when we want to execute graph.

Graph is an abstraction. Operations of graph are 'interfaces' in OOP.

Kernel has implementation of interface/operations

TensorFlow framework has its own data-types (like FLOAT, INT-32, etc).

Not depending on Python for more control.

Operations are `tf.xxxx()`. Other object. `xxx()` like `zl.eval()` and `zl.get_shape` are "methods" - they don't have a place in the graph.

`session.close()` → good practice; frees up house-keeping. Even after closing session, we can sometimes run the operations and methods but they are not guaranteed to run.

4/11/2017

- Getting data in tensorflow
- constants
 - variables
 - placeholders
 - files

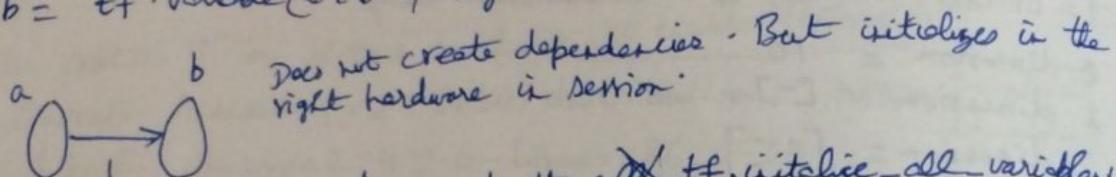
Constants = Immutable tensors

Variable = A modifiable tensor that lives in tensorflow's graph of interacting operations

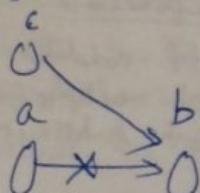
- Use `tf.Variable`

`a = tf.constant(10)`

`b = tf.Variable(a + 0)` # only creates nodes



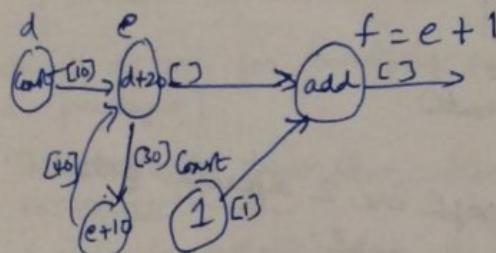
Every time a new variable is declared, we must call `tf.initialize_all_variables()`.



`b = tf.Variable(c + 1)`

`tf.initialize_all_variables()`

Constant = 0-dimensional tensor



`d = tf.constant(10)`

`e = tf.Variable(d + 20)`

`f = tf.add(e, tf.constant(1))`

`update = e.assign(e + 10)`

`update.eval()`

5/1/2017

Visualization of Tensorflow

Tensorboard -> Source activate tensorflow

\$ tensorboard --logdir = ./output

`http://127.0.0.1:6006` → click on 'Graph'.

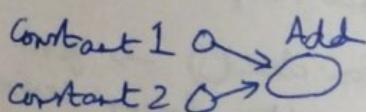
Code

```
writer = tf.train.SummaryWriter("output", session.graph)
writer.close()
```

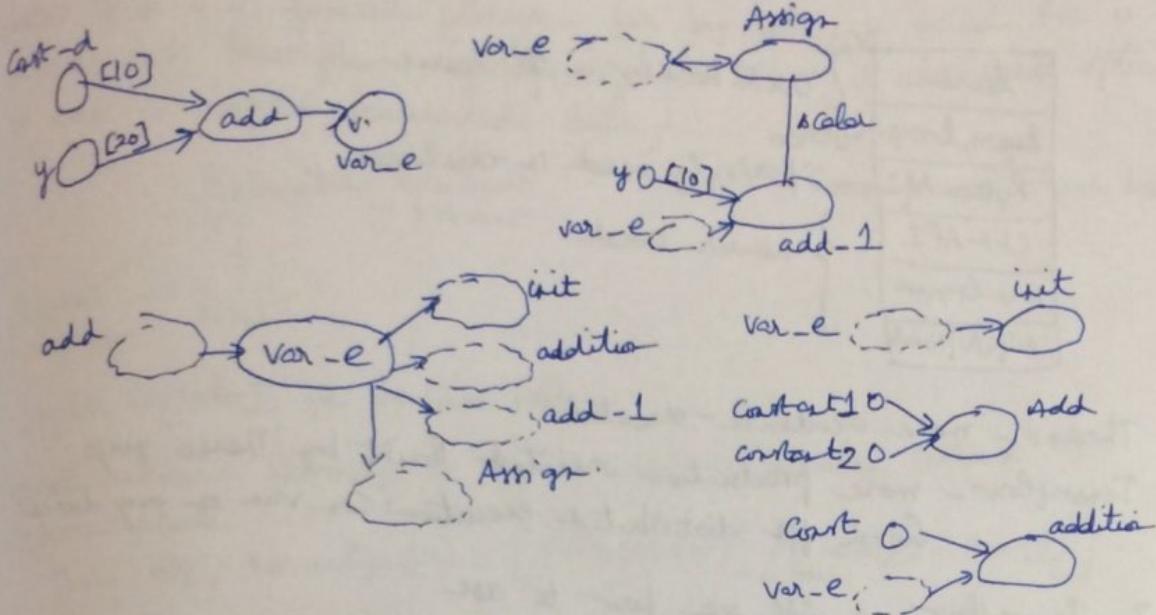
`a = tf.constant(10, name = "constant1")`

`b = tf.constant(20, name = "constant2")`

`c = tf.add(a, b)`



$d = \text{tf} \cdot \text{Constant}(10, \text{name} = "Const_d")$
 $e = \text{tf} \cdot \text{Variable}(d + 20, \text{name} = "var_e")$
 $f = \text{tf} \cdot \text{add}(e, \text{tf} \cdot \text{Constant}(1), \text{name} = "addition")$
 $\text{session} \cdot \text{Run}(\text{tf} \cdot \text{PrintAllVariables}())$
 $\text{update} = e \cdot \text{Assign}(e + 10)$



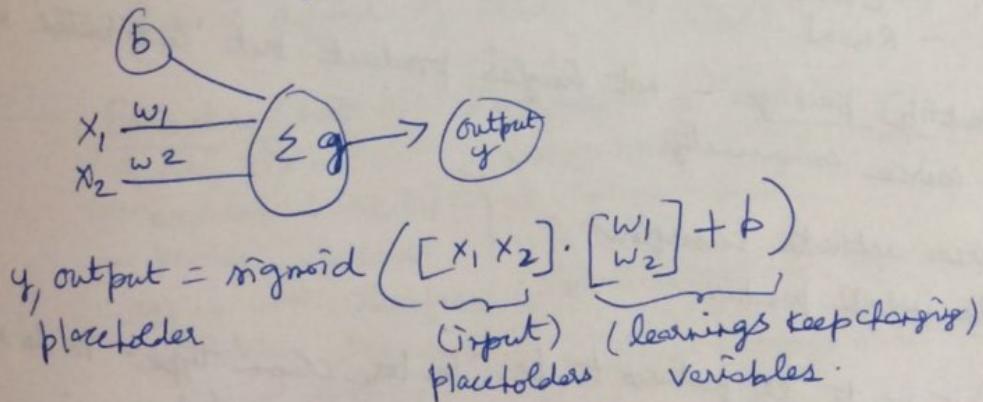
Cont - does not have 'init' part
- no 'input'

Any operation on 'tf' creates 1 or more nodes.

Four ways to supply data to tensors

- Constants
 - Variables
 - Placeholders - symbolic variables; must be fed with data on execution
 - files - datafiles - use 'placeholder' to read files into tensorflow

We code mostly in low-level (instead of high level package) to get more control. In competition, we see low-level code. Not enough maturity in high-level code yet for fine control.



6/1/2017 Manual feature extraction / Feature engineering very difficult for unstructured data. Automation possible with DNN, CNN, RNN.

Try to understand 'plain neural network' before understanding 'deep neural network'.

lectures	} used heavily in this course
layers, losses, metrics	- sparingly used in this course
Python API	
C++ API	} Never used
Core Engine	
CPU, GPU, Mobile	

Theano - more academic-oriented

Tensorflow - more production oriented - Built by 'Theano' guys.
Good for distributed execution: Can run on any device.

In Tensorflow, we will see how to use

- objective perceptron
- single layer perceptron
- multi-layer perceptron

1) Build Graph

- Tensor
- operation

2) Execute graph

7/1/2017

Deep network models

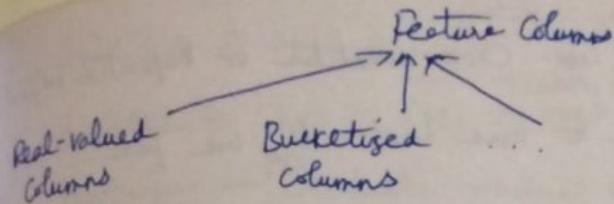
- Objective perception
- Layers of perception
- DNN
- CNN
- RNN

tf.contrib package - not Google's product but 'contributed' by open source community.

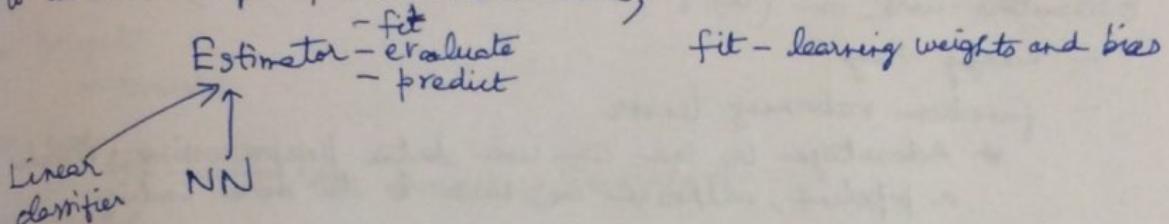
\$ source activate tensorflow

\$ conda install pandas

Tensorflow wants all features to be 'feature column' type. To do this, use layers. Real-valued column for 'continuous' types.
Dataframe understandable to Pandas but not to Tensorflow



Tensorflow can handle big data efficiently even for structured data. Spark is a general platform for big data, of which ML is a subset. But Tensorflow is specifically for ML - it also gives options to work on simpler/structured data.



Spyder - cbt I for syntax help

Neural network learning is faster with [0,1] scale instead of Z-score scale.

That's why, $\text{normalize}(x) = [x - \text{mean}(x)] / [\text{max}(x) - \text{min}(x)]$.

In unstructured data, we don't deal much with 'missing values' as the input is images. Mostly, we just do normalization.

9/1/2017

To solve issues with guest additions installation

- ↳ yum update Kernel
- ↳ yum install gcc
- ↳ yum install kernel-devel

Install VirtualBox 5.1.12 guest additions

Sigmoid perceptron. If $p(\text{output}) > 0.5 \Rightarrow 1 \text{ output}$ } by default
 $< 0.5 \Rightarrow 0 \text{ output}$ }

$$w_1x_1 + w_2x_2 + b = 0$$

so, line is defined by points $(0, -\frac{b}{w_2})$ and $(-\frac{b}{w_1}, 0)$

10/1/2017

Estimators (built-in/custom) support in Tensorflow.

- fit (train)
- evaluate (validate)
- predict (inference)

Accept inputs of type numpy array as well as tensorflow types

e.g) Linear classifier

(Sigmoid Perceptron / Logistic Regression)

Linear Regressor

(CNN classifiers still not released)

DNN classifier

Regressor

DNN Regressor

Tensorflow RNN classifier

Tensorflow RNN Regressor

In real-world works we need to build custom estimators

Generally for model evaluation, we use cross-validation or Repeated k-fold. But for unstructured data, due to time constraints, we just use 'holdout'.

The classifier's evaluate() method returns the confusion matrix.

Advantages of using

Estimators work on (i.e., fit, evaluate, predict methods)

- numpy array
- function returning tensor

* Advantage is, we can use data preprocessing - Provide a pipeline; although we have to do more coding.

Creating custom estimators

model_params = { "learning_rate": LEARNING_RATE }

nn = tf.contrib.learn.Estimator(model_fn=modelfunction,
params= model_params)

Skeleton of model_fn: → multiple target (do user target this
modelfunction(features, targets, mode, params); with one-hot-encoding
logic of custom model → optional
return predictions, loss, train_op / optimizer

mode - says if it is called by fit, eval or predict; context.

params ← model_params

loss - loss function

train_op / optimizer - say how to minimize loss (e.g.) stochastic gradient descent

learning_rate = η i.e., $w = w - \eta \frac{\partial E}{\partial w}$

11/11/2017

[Sub-packages]

tf.contrib →
layers →
losses → crossentropy, squared error
metrics →
monitors → for debugging
distribution → probability distributions
learns →
crf
bayesflow

[Operations]

high-level nn ops (e.g.) fully connected
optimizers (e.g.) optimize loss
regularizers
initializers
summaries
estimators
input processing
graph actions

conditional random field framework

[Overview of tf.contrib module]

Predictions = output

Tensor containing a scalar loss value.
Objective fn / Loss - (eg) squared error, cross entropy

Optimizer - ^(M) Stochastic gradient descent; Gradient descent - An op. that runs one step of training.

Inputs for custom estimators

* features

- represents dictionary of features
- passed to custom estimator fn using fit, evaluate, predict

* target

- tensors

* node

- tf.contrib.learn.ModeKeys.TRAIN, etc.

* params

- dictionary of hyper parameters for training

'Loss' is meaningful / returned by EVAL and TRAIN mode only.

Estimator logic

Configure model via
tensorflow operations

layer = tf.contrib.layers.fully_connected(
inputs = features,
num_outputs = 5,
activation_fn = tf.sigmoid)

Find predictions of model

predictions = tf.reshape(layer, [-1])
predictions_dict = {“keys”: predictions}

Define loss fn for
training / evaluation

loss = tf.contrib.losses.mean_squared_error(
predictions, targets)

Define optimizer / training
operation

train_op = tf.contrib.layers.optimize_loss(
loss = loss,
global_step = tf.contrib.framework.get_global_step(),
learning_rate = params[“learning_rate”],
optimizer = “SGD”)

fully connected - all inputs are given to all perceptrons.

global_step - store count of the epochs used. We don't use simple integer because in distributed environment, we want counts across machines

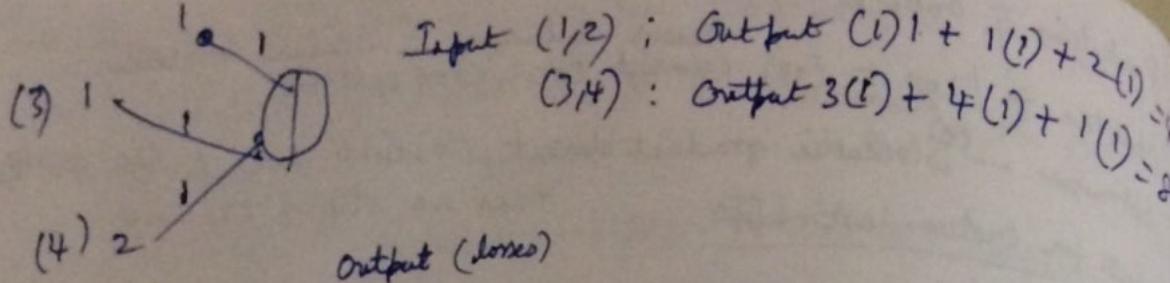
④ For classification, use log-loss

For regression, use mean-squared error.

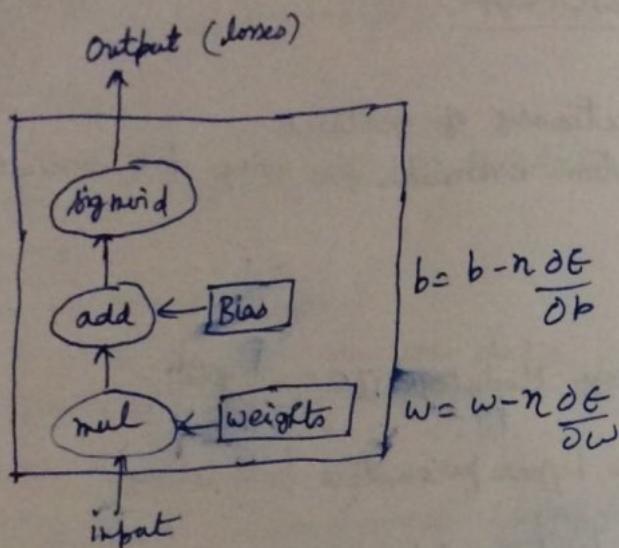
12/1/2017

Tensorflow operations

- math operations
- Neural net work operations
- matrix operations
- high level operations



13/1/2017



nn = learn. Estimator (- -)

- No need to call 'Session'; everything done internally by Estimator

nn. fit()

- model building algorithm starts here

(*) Operate a session to run the graph.

'model-dir' stores the graph - does not store the final model with weights and biases.

To see graph

\$ tensorboard --logdir=“./tmpdir/”

<http://127.0.0.1:6006>

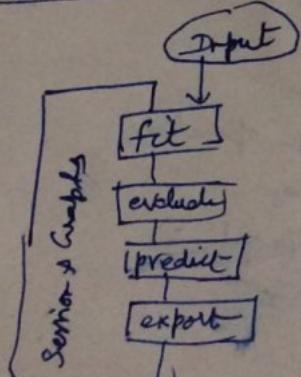
To save final model, do

\$ nn.export (- -)

To evaluate/validate unstructured data, due to huge data size,
we do a one-time holdout only.

We are using tensorflow version 0.11. 0.12 & 1.0 versions also available

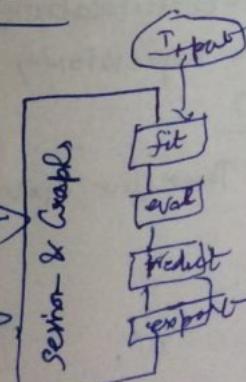
Built-in estimators



Custom estimators

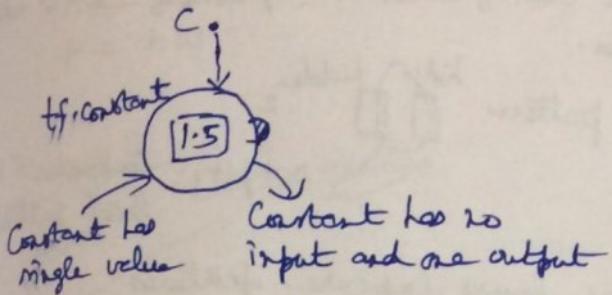
Model Defn
(TF Python)

Params



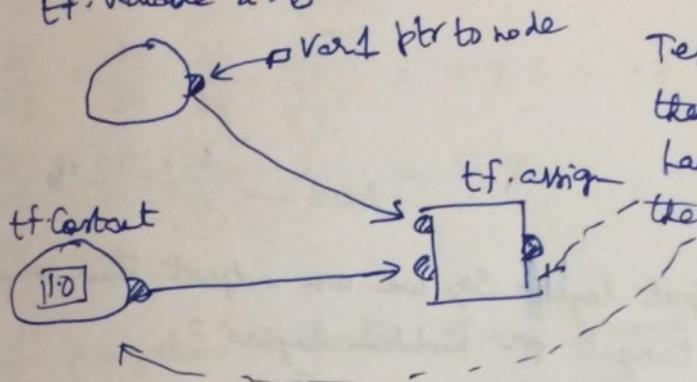
$c = \text{tf.constant}(1.5)$

'c' is a pointer to the 'constant' node in graph.



Tensor Graph

2) $\text{var1} = \text{tf.Variable}(1.0, \text{name} = "x")$ $"x"$ is the name of node
 $\text{tf.Variable } x : 0$



Tensorflow automatically creates these two other nodes to handle the assignment of the initial value.

Need to execute

`session.run(tf.initialize_all_variables())`
and only later

`session.run(var1)`

$\rightarrow (\text{es}) \text{ layers} \cdot \text{fully-connected}(\dots)$

When we `session.run` (a node), all operations (dependencies) until that node get executed automatically.

14/11/2017 Multi-layer perceptrons

With single layer, we get to multi-class linear classification.

Perceptrons only create linear classifications.

Feed forward neural network

N-layer neural network = 1 input layer + $(N-1)$ hidden layers + 1 output layer.

(The input layer is not counted as a layer).

Why more layers?

- Individual features may represent local patterns & etc

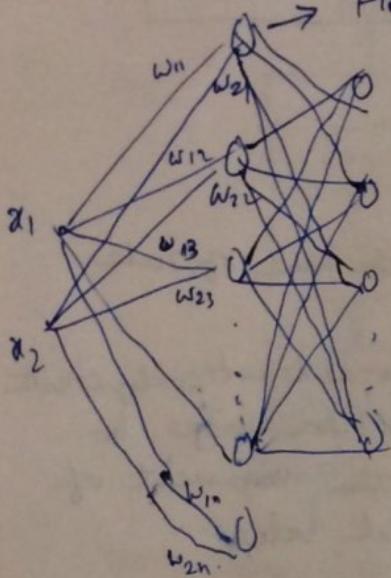
- Complex patterns: combinations of local patterns.

Shallow NN - only one/two hidden layer with lots of perceptrons

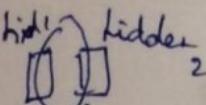
Deep NN - more hidden layers, less perceptrons in each layer.

- # hidden layers > 2 .

If, in two NN, both have same # hidden layers. Still, we can say that having more neurons in each hidden layer will perform better \rightarrow NN is not just for classification & regression. It is also to extract features.



Finds local patterns



Output₁ = Combination of local patterns

Hidden layers capture features.

- 1st hidden layers captures small features
- 2nd hidden layers - more complex features

(*) We know targets of output layer. So, we can adjust their weights.
But how to set the target for hidden layers?

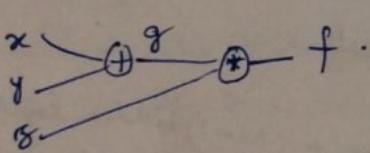
Output layer mostly has 'softmax' activation for interpretability.
But hidden layers does not need interpretability; Sigmoid and mostly.

Can we use a combination of activation fns in a single layer?

) $f(x, y, g) = (x+y)g$

$$g = x+y$$

$$f = g \cdot h$$



Need to compute $\frac{\partial f}{\partial x}$, $\frac{\partial f}{\partial y}$, $\frac{\partial f}{\partial g}$

x	y	g	g	f
1	2	3	3	9

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial x} = g \cdot 1 = g$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial y} = g \cdot 1 = g$$

$$\frac{\partial f}{\partial g} = g$$

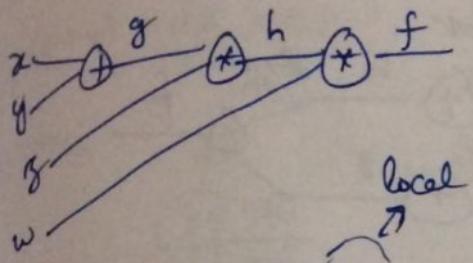
} Find rate of change of output w.r.t all inputs

$$2) f(w, x, y, z) = (x+y)wg'$$

$$g = x + y$$

$$h = gy$$

$$f = hw$$



$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial h} \cdot \frac{\partial h}{\partial g} \left(\frac{\partial g}{\partial x} \right) = w \cdot g \cdot 1 = wg$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial h} \cdot \frac{\partial h}{\partial g} \left(\frac{\partial g}{\partial y} \right) = w \cdot g \cdot 1 = wg$$

$$\frac{\partial f}{\partial g} = \frac{\partial f}{\partial h} \left(\frac{\partial h}{\partial g} \right) \rightarrow \text{local} = w \cdot g$$

$$\frac{\partial f}{\partial w} = h.$$

$$x = x - n \frac{\partial f}{\partial x}$$

'Back propagation' is faster than 'forward propagation'.

- Compute $\frac{\partial f}{\partial h}$ first; reuse it in multiple places.

- efficiently computes gradients, $\frac{\partial E}{\partial w_{ij}}$, by reusing the intermediates.

it one pass of hidden layers.

In 'forward propagation', we can't reuse.

'Shallow NN' creates more weights to learn; slower

'Back propagation' does 'global learning'.

(*) Learning objective for each 'layer' and learning is different; it is called 'local learning' — aka, Boltzmann machine / Auto-encoder.

Issue with backpropagation — How to deal with unsupervised forward learning? — Open research area

Extracting features from images

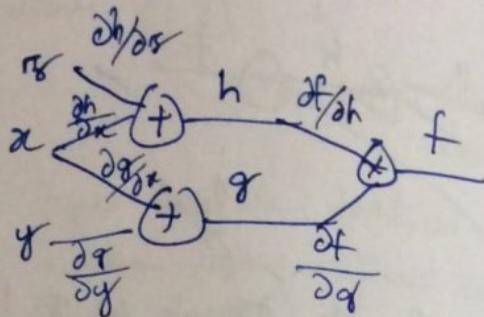
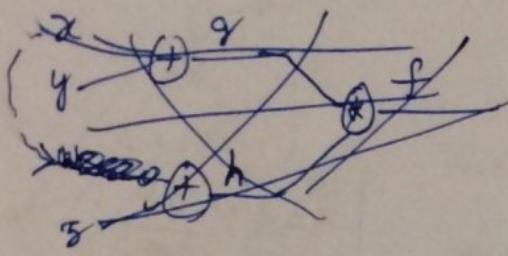
- Still supervised learning
- hidden layers luckily extracting features

$$3) f(x, y, z) = (x+y)(x+z)$$

$$g = x+y$$

$$h = x+z$$

$$f = gh$$

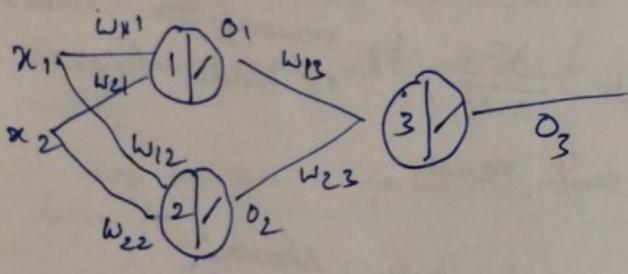


$$\frac{\partial f}{\partial g} = \frac{\partial f}{\partial h} \cdot \frac{\partial h}{\partial g}$$

$$\frac{\partial f}{\partial gy} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial y}$$

$$\frac{\partial f}{\partial x} = \left(\frac{\partial f}{\partial h} \cdot \frac{\partial h}{\partial x} \right) + \left(\frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial x} \right)$$

\xrightarrow{x}



E = Error fn.

Linear activation fn.

$$\frac{\partial E}{\partial w_{13}} = \frac{\partial E}{\partial o_3} \cdot \frac{\partial o_3}{\partial w_{13}}$$

$$\frac{\partial E}{\partial w_{23}} = \frac{\partial E}{\partial o_3} \cdot \frac{\partial o_3}{\partial w_{23}}$$

$$\frac{\partial E}{\partial w_{11}} = \frac{\partial E}{\partial o_3} \cdot \frac{\partial o_3}{\partial o_1} \cdot \frac{\partial o_1}{\partial w_{11}}$$

$$\frac{\partial E}{\partial w_{21}} = \frac{\partial E}{\partial o_3} \cdot \frac{\partial o_3}{\partial o_1} \cdot \frac{\partial o_1}{\partial w_{21}}$$

Need 1 forward pass + 1 backward

1 forward pass to

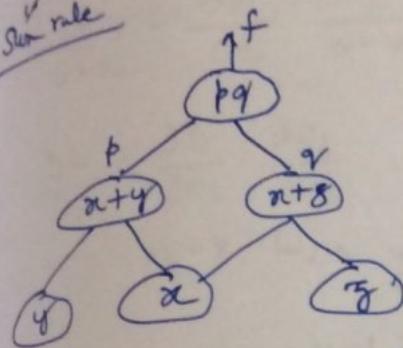
- learn o_1, o_2, o_3

1 backward pass to

- learn gradients

16/11/2017

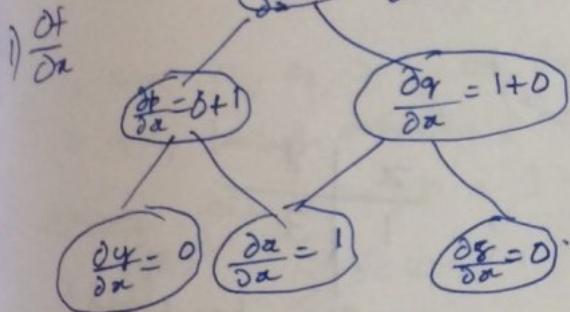
<u>Forward Propagation</u> $\frac{d}{dx}(-) = ?$ <u>Product rule</u> $\frac{d}{dx}(pq) = p \frac{dq}{dx} + q \frac{dp}{dx}$ <u>Sum rule</u> $\frac{d}{dx}(p+q) = \frac{dp}{dx} + \frac{dq}{dx}$	<u>Back propagation</u> $\frac{df}{d(-)} = ?$ $\frac{df}{dx} = \frac{\partial f}{\partial p} \cdot \frac{\partial p}{\partial x}$ (chain rule)
--	--



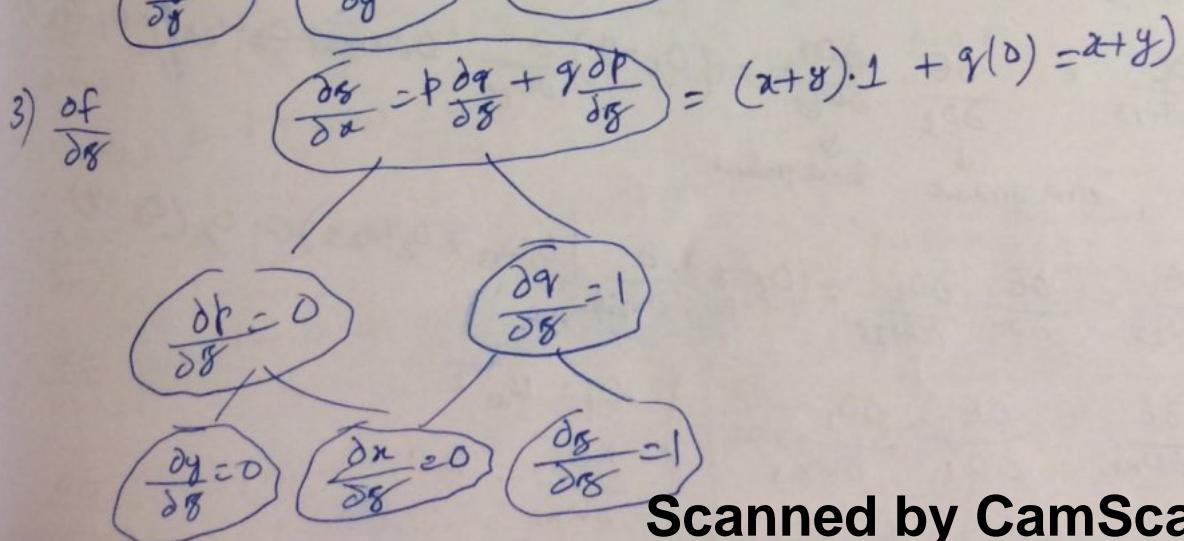
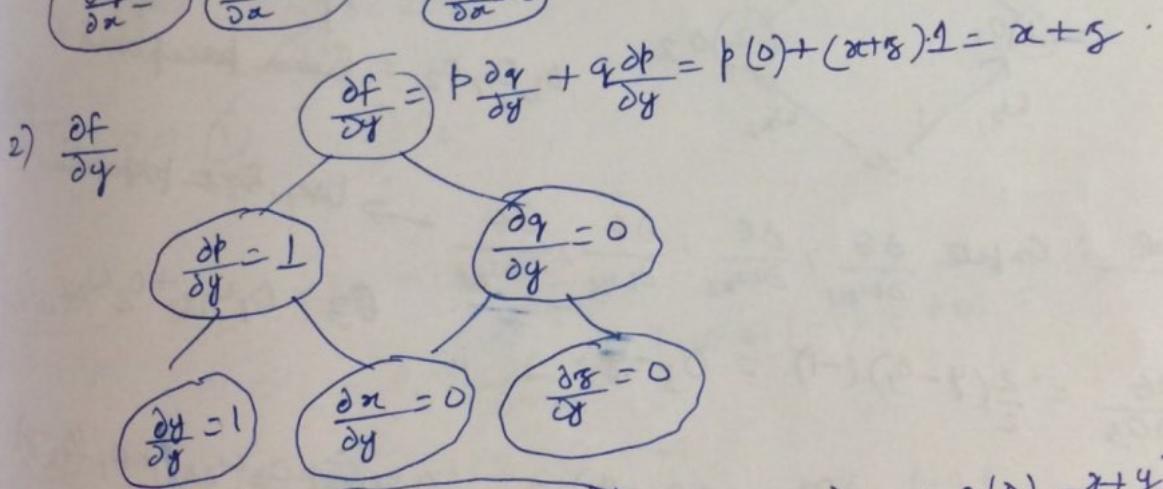
Compute $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

Forward propagation

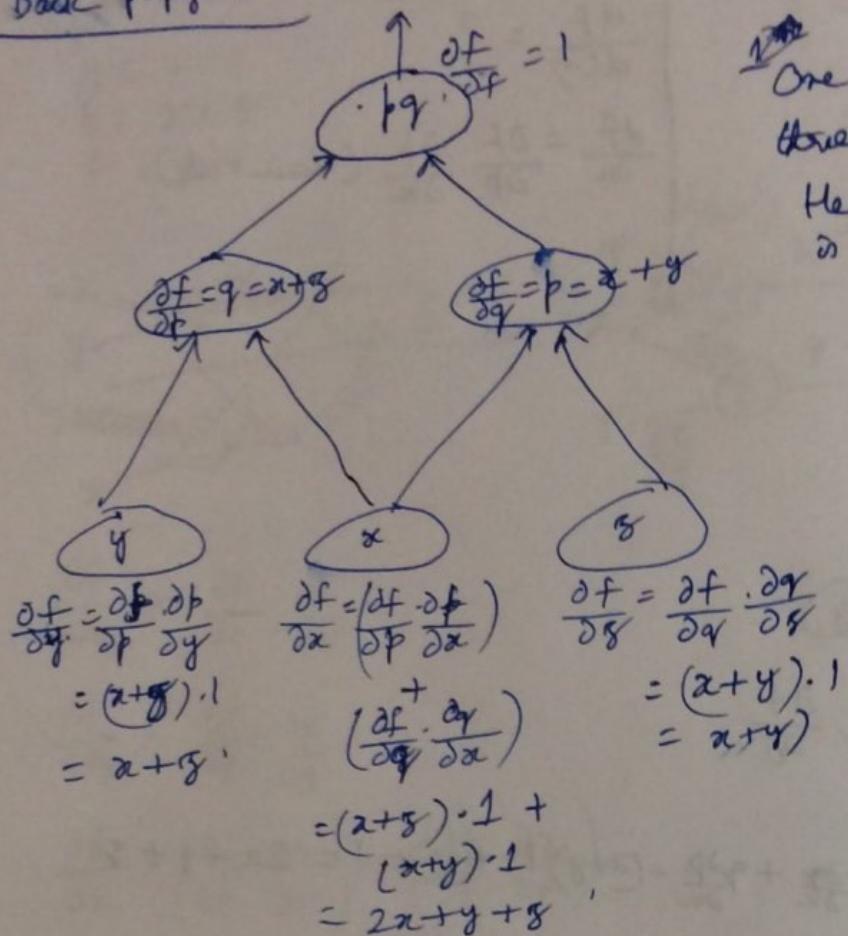
$$\frac{\partial f}{\partial x} = p \frac{\partial x}{\partial x} + q \frac{\partial p}{\partial x} = (x+y) \cdot 1 + (z+y) \cdot 1 = 2x + y + z$$



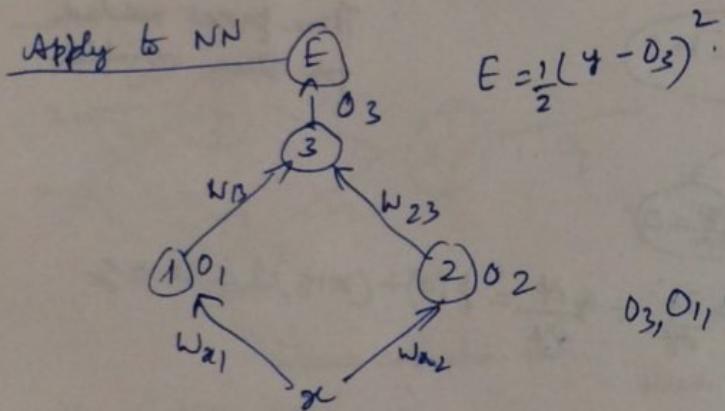
Three passes needed
for three variables



Back propagation



One pass enough for all three variables.
Hence, back propagation is preferred.



x		y
1		3

o_3, o_1, o_2 = linear perceptions

Goal: Compute $\frac{\partial E}{\partial w_{11}}, \frac{\partial E}{\partial w_{22}}, \frac{\partial E}{\partial w_{13}}, \frac{\partial E}{\partial w_{23}}$ → Use Back propagation
 $\theta_3 = o_1 w_{13} + o_2 w_{23}$

$$\frac{\partial E}{\partial o_3} = \frac{1}{2}(y - o_3)(-1) = o_3 - y$$

$$1) \frac{\partial E}{\partial w_{13}} = \frac{\partial E}{\partial o_3} \cdot \frac{\partial o_3}{\partial w_{13}} = (o_3 - y) \frac{\partial}{\partial w_{13}} (o_1 w_{13} + o_2 w_{23}) = o_1 (o_3 - y)$$

\downarrow error gradient \downarrow local gradient

$$2) \frac{\partial E}{\partial w_{23}} = \frac{\partial E}{\partial o_3} \cdot \frac{\partial o_3}{\partial w_{23}} = (o_3 - y) \frac{\partial}{\partial w_{23}} (o_1 w_{13} + o_2 w_{23}) = o_2 (o_3 - y)$$

$$3) \frac{\partial E}{\partial w_{11}} = \frac{\partial E}{\partial o_1} \cdot \frac{\partial o_1}{\partial w_{11}}$$

\downarrow error gradient \downarrow local gradient

$$\left. \begin{array}{l} o_1 = w_{x_1} x \\ \partial o_1 = x \end{array} \right\}$$

$$\frac{\partial E}{\partial O_1} = \frac{\partial E}{\partial O_3} \cdot \frac{\partial O_3}{\partial O_1}$$

$$= (O_3 - y) w_{13}$$

$$\therefore \frac{\partial E}{\partial w_{11}} = (O_3 - y) w_{13} \cdot x$$

$$4) \frac{\partial E}{\partial w_{22}} = \frac{\partial E}{\partial O_2} \cdot \frac{\partial O_2}{\partial w_{22}}$$

$$O_2 = w_{22} \cdot x$$

$$\frac{\partial O_2}{\partial w_{22}} = x$$

$$\frac{\partial E}{\partial O_2} = \frac{\partial E}{\partial O_3} \cdot \frac{\partial O_3}{\partial O_2}$$

$$= (O_3 - y) \cdot w_{23}$$

$$\therefore \frac{\partial E}{\partial w_{22}} = (O_3 - y) w_{23} x$$

Learning in NN \notin Gradient descent

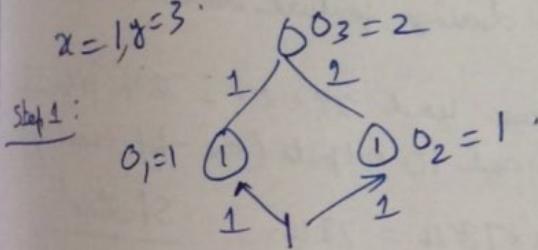
$$w_{ij} = w_{ij} - n \frac{\partial E}{\partial w_{ij}}$$

To know O_1, O_2, O_3 , go forward first.

To apply partial derivatives, go in back direction.

Average, all 4 weights = 1. Ignoring bias for simplicity
 $n = 0.01$

$$x = 1, y = 3$$



Step 2: $w_{13} = w_{13} - n \cdot \frac{\partial E}{\partial w_{13}}$

$$= 1 - 0.01 \frac{\partial E}{\partial w_{13}} = 1 - 0.01 (-1) = 1.01$$

$$\frac{\partial E}{\partial w_{13}} = O_1 (O_3 - y) = 1 \cdot (2 - 3) = -1$$

$$w_{23} = 1 - 0.01 \frac{\partial E}{\partial w_{23}} = 1 - 0.01 (-1) = 1.01$$

$$\frac{\partial E}{\partial w_{23}} = O_2 (O_3 - y) = 1 (2 - 3) = -1$$

$$\frac{\partial E}{\partial w_{21}} = 1 - 0.01 \frac{\partial E}{\partial w_{21}} = 1.01$$

$$\frac{\partial E}{\partial w_{21}} = (O_3 - y) w_{13} \cdot x$$

$$= (2 - 3) \cdot 1 \cdot 1 = -1$$

$$\frac{\partial E}{\partial w_{22}} = 1 - 0.01 \frac{\partial E}{\partial w_{22}} = 1.01$$

$$\frac{\partial E}{\partial w_{22}} = (O_3 - y) w_{23} \cdot x$$

$$= (2 - 3) \cdot 1 \cdot 1 = -1$$

Summary

- 1) Compute partial derivatives using back propagation
- 2) Learn O_1, O_2, O_3 using forward calculation
- 3) Learn the weights using partial derivatives & O_i values
 - direction of weight learning does not matter; That is, the
 - Order of weight updation does not matter

$$\frac{\partial O_j}{\partial w_{ij}} = \frac{\partial O_j}{\partial a_j} \cdot \frac{\partial a_j}{\partial w_{ij}}$$

For linear perceptron, $\frac{\partial O_j}{\partial a_j} = 1$.

$$= g'(a_j) \underbrace{\frac{\partial \sum_k w_{kj} O_k}{\partial w_{ij}}}_{\text{local gradients will change}}$$

For sigmoid perceptron,

$$\delta_j = \frac{\partial E}{\partial O_j} = E'(O_j)$$

=

In books, they compute error gradients first, and then they compute the local gradients, and finally apply weights.

- ⊗ For classification/regression, the famous 'threshold' of sigmoid perceptron is not useful. Learning happens based on difference in target and actual output of ~~sigmoid~~ sigmoid perceptrons.
Firing threshold is useful only during feature detection.

17/11/17 For digit recognizer, suppose we used $28 \times 28 = 784$ inputs and $\frac{784}{100}$ perceptions in hidden layer 1 and 10 outputs (one-hot encoding)
 $\# \text{weights to layer} = 100 \times 784 = 78400$ Shallow
 $+ 10 \times 100 = \frac{1000}{79400}$

With a 'deeper' network,

$$\begin{aligned} \text{hidden layer 1} &= 50 \text{ perceptions} \\ \text{layer 2} &= 10 \text{ perceptions} \end{aligned}$$

Shallow vs Deep

$$\begin{aligned} \text{Total weight} &= 50 \times 784 = 39200 \\ &+ 10 \times 50 = 500 \\ &10 \times 10 = 100 \\ &\hline 39800 \end{aligned}$$

Less learning with layers

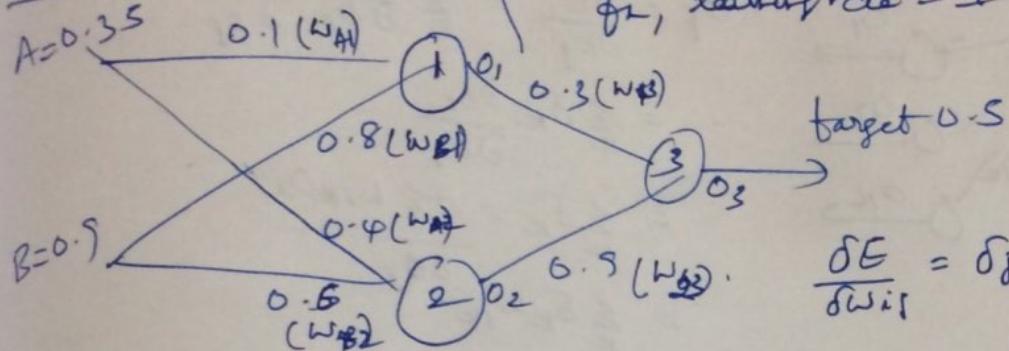
weights in deep networks!

Also, hidden nesting to hidden layers.

Local gradients

$$\begin{aligned}\frac{\partial o_j}{\partial w_{ij}} &= \frac{\partial o_j}{\partial a_j} \cdot \frac{\partial a_j}{\partial w_{ij}} \\ &= g'(a_j) \frac{\partial \sum_k w_{kj} o_k}{\partial w_{ij}} \\ &= g'(a_j) o_i\end{aligned}$$

Input



Apply back propagation for below.
Sigmoid perceptrons, squared loss objective
for, learning rate = 1

$$\frac{\delta E}{\delta w_{ij}} = \delta_j g'(a_j) o_i$$

$$o_1 = \text{sigmoid}(0.35 \times 0.1 + 0.9 \times 0.8) = \text{sigmoid}(0.755) = 0.68$$

$$o_2 = \text{sigmoid}(0.35 \times 0.4 + 0.6 \times 0.9) = \text{sigmoid}(\dots) = 0.66$$

$$o_3 = \text{sigmoid}(0.3 o_1 + 0.9 o_2) = 0.69$$

$$\begin{aligned}E &= \frac{1}{2} (y - o_3)^2 \\ \delta_3 &= \frac{\partial E}{\partial o_3} = o_3 - y\end{aligned}$$

$$\delta_3 = (o_3 - y) = (0.69 - 0.5) = 0.19$$

$$\delta_2 = 0.9 \delta_3 = 0.9 \times 0.19 = 0.171$$

$$\delta_1 = 0.3 \delta_3 = 0.3 \times 0.19 = 0.057$$

$$\begin{aligned}w_{A1} &= w_{A1} - n \delta_1 g(a_1) [1 - g(a_1)] A \\ &= 0.1 - 0.057 (0.68) (1 - 0.68) 0.35 \\ &= 0.096\end{aligned}$$

$$\begin{aligned}w_{B1} &= w_{B1} - n \delta_1 g(a_1) [1 - g(a_1)] B \\ &= 0.8 - 0.057 \times 0.68 (1 - 0.68) 0.9 = 0.694\end{aligned}$$

$$\begin{aligned}w_{A2} &= w_{A2} - n \delta_2 g(a_2) [1 - g(a_2)] A \\ &= 0.4 - 0.171 \times 0.66 (1 - 0.66) 0.35 = 0.387\end{aligned}$$

$$\begin{aligned}w_{B2} &= w_{B2} - n \delta_2 g(a_2) [1 - g(a_2)] B \\ &= 0.6 - 0.171 (0.66) (1 - 0.66) 0.9 = 0.567\end{aligned}$$

$$\begin{aligned}w_{13} &= w_{13} - n \delta_3 g(a_3) [1 - g(a_3)] o_1 \\ &= 0.3 - 0.19 \times 0.69 (1 - 0.69) 0.68 = 0.223\end{aligned}$$

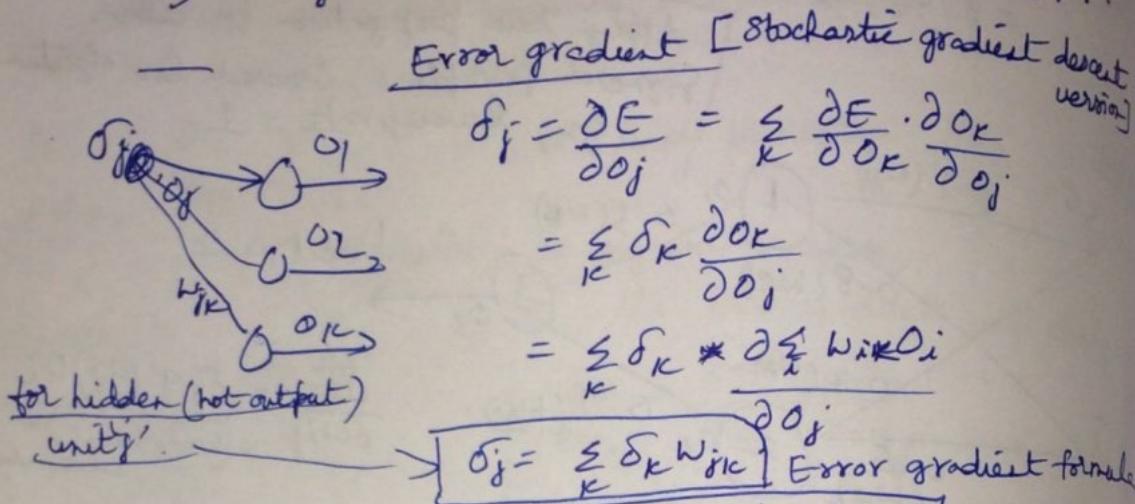
$$\begin{aligned}w_{23} &= w_{23} - n \delta_3 g(a_3) [1 - g(a_3)] o_2 \\ &= 0.9 - 0.19 \times 0.69 (1 - 0.69) 0.66 \\ &= 0.874\end{aligned}$$

After 1st epoch/iteration,

$$o_1 = \text{sigmoid}(0.096 \times 0.35 + 0.694 \times 0.9) = 0.658$$

$$o_2 = \text{sigmoid}(0.387 \times 0.35 + 0.567 \times 0.9) = 0.658$$

$$o_3 = \text{sigmoid}(o_1 \times 0.273 + o_2 \times 0.874) = 0.679.$$



$$w_{ij} = w_{ij} - n \frac{\partial E}{\partial w_{ij}} = [w_{ij} - n \delta_j g'(o_j) o_i]$$

Weight adjustment formula

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \cdot \frac{\partial o_j}{\partial w_{ij}} = \delta_j \cdot \frac{\partial o_j}{\partial w_{ij}} = \delta_j g'(o_j) o_i.$$

For sigmoid perceptron, $g'(o_j) = g(o_j) [1 - g(o_j)]$

$$\Rightarrow \frac{\partial E}{\partial w_{ij}} = \delta_j g(o_j) [1 - g(o_j)] o_i.$$

$$\therefore w_{ij} = w_{ij} - n \delta_j g(o_j) [1 - g(o_j)] o_i$$

For 'output' (not hidden) unit j , $\delta_j = \frac{\partial E}{\partial o_j} = E'(o_j)$.

18/11/2017

MNIST data - digit recognizer - Custom estimator code

19/11/2017

fit (—, steps = —, $\underbrace{\text{batch size} = \text{—}}_{\text{optional}}$)

steps = one iteration

batch size = # samples to consider for single step of learning

$$w_{ij}' = w_{ij} - n \sum_{n=1}^N \delta_j^{(n)} o_i^{(n)} g'(o_j)$$

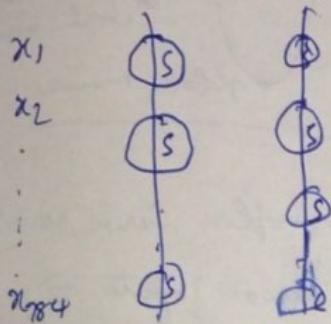
$o = \text{batch size} = 1 \Rightarrow \text{SGD}$
 $= n \Rightarrow \text{GD}$
 $= n-k \Rightarrow \text{minibatch}$

$$\frac{\partial E}{\partial w_{ij}} = \frac{1}{n} \sum_{i=1}^N (y_i - o_i)^2 = \frac{1}{n} (o - y_i)$$

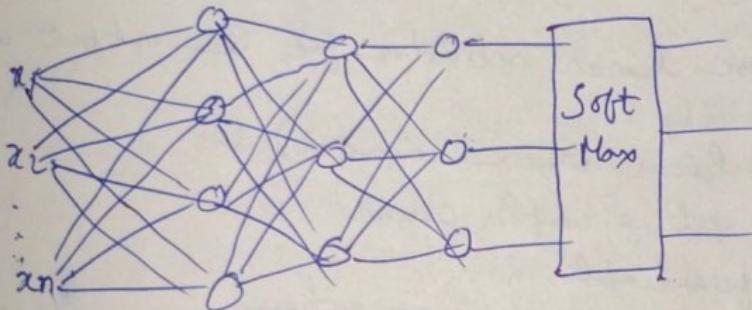
So, $\frac{\partial E}{\partial w_{ij}}$ itself involves summation. So, w_{ij}' computation will also involve summation.

There is empirical proof that SGD (batch size = 1) will do convergence eventually.

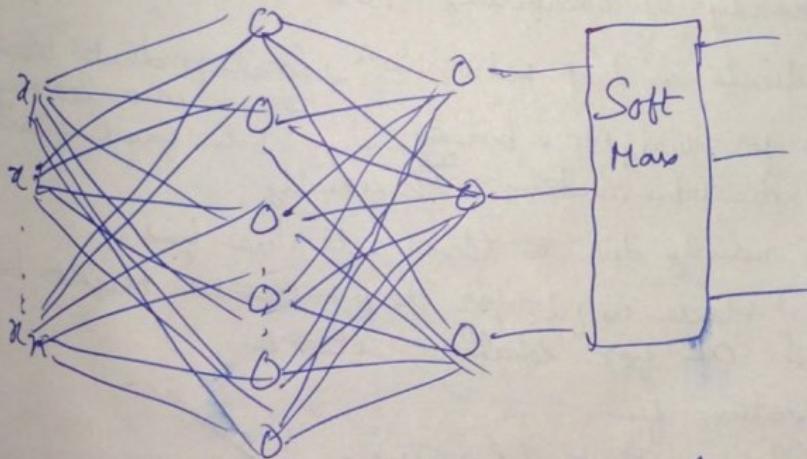
With SGD, the loss may not always decrease with each step. It may increase a bit sometimes, but will finally keep decreasing over many steps. With 'Batch' and 'Mini-batch', loss decrease is not so erratic.



20/1/2017 Deep Network



Shallow Network



Both are powerful, but takes more learning in shallow network if we try to improve by adding more perceptions in a layer.

Deep layer advantages

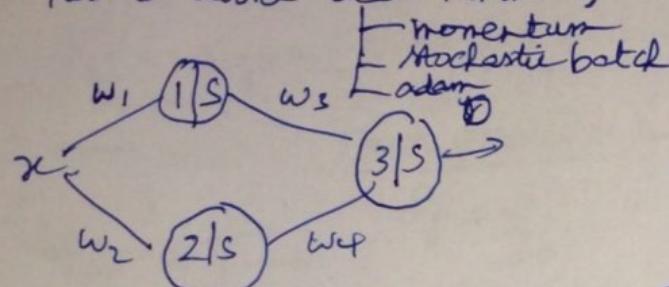
- Pattern recognition (shallow don't do this)
- Noise reduction
- Detect features (biggest strength)
- Much lesser number of perceptions

Issues with NN learning

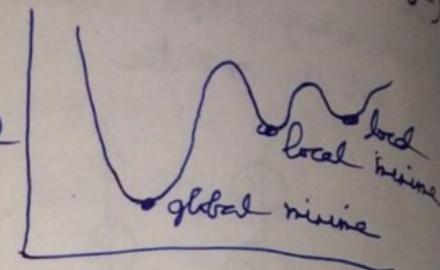
• Objective (loss) function is non-convex

- can have multiple minima

1) - How to avoid local minima?



Because the output of NN is result of many fns (and because of the objective fn)



$$O = g(g(w_1x)w_3 + g(w_2x)w_4)$$

$$f_1 = x+2, f_2 = x+5, f_3 = 3x+6$$

$$f = f_3(f_2(f_1(x))) = f_3(f_2(x+2)) = f_3(x+7) = 3x+21$$

Complex Curve, non-convex
linear inputs \Rightarrow linear output

If we simply use linear activation fn, the output will be a linear fn only.

So, use a) non-linear (sigmoid) fn

- for getting complex curves

or b) differentiable fn

- for doing back propagation

regularization (L1, L2, L1+L2)

\hookrightarrow dropout
 \hookrightarrow data augmentation

2) Due to complex curve

- NN can easily do overfitting. How to handle it?

3) Vanishing gradients in deep networks \hookrightarrow local gradients becoming zero \Rightarrow no back learning.

4) Initial values for weights & biases \hookrightarrow Soln found finally
 \hookrightarrow Xavier Initialization strategy \hookrightarrow O' usually.

Vanishing gradient mainly due to the activation fn

- sigmoid
- tanh

Hence unfit for deep networks. Might be OK for shallow network.

Alternative activation fn

- Rectified Linear Unit (ReLU).

- non-linear fn

Issue with Sigmoid activation fn

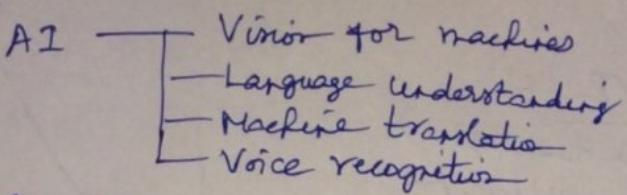
\rightarrow slope becomes '0' $\quad g'(a_j) = g(a_j)(1-g(a_j))$
as value increases so, when $g(a_j)=1$,
 $g'(a_j)=0$.

In ReLU, slope non-zero with

higher values. Since non-linear, it does identify complex patterns.

'Dropout' more popular than L1+L2 in NN for overfitting
- Selectively drop a few neurons.

23/1/17



Single layer

Classification — Cross entropy + sigmoid/softmax
Regression — Squared loss + Linear.

Multiple layers

Classification:

Hidden layers — Sigmoid + cross entropy
Output layer — Softmax + cross entropy.

Regression:

Hidden layer — Sigmoid + squared error
Output layer — Linear + squared error.

Issues with sigmoid activation for

i) Outputs are not zero-centered (but 0.5 centered).

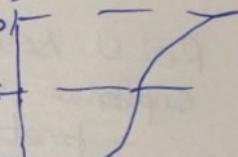
- If inputs are always +ve, then
the gradients or w_i are always all +ve
or all -ve.

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \cdot \frac{\partial o_j}{\partial w_{ij}}$$
$$= \delta_j g'(o_j) o_i$$

This decides always +ve or -ve of the entire sigmoid product

If gradients all +ve or all -ve, then sliding up/down towards the minima won't happen.

- If all -ve, always slide down; all +ve, always slide up.



evidence = $a = \sum w_i x_i$

Thus the evidence 'a' will be all +ve or all -ve. But this is not how real network of neurons learn. Back propagation problematic.

Sol: Use 'tanh' instead of 'sigmoid'.

2) Saturated sigmoids kill the gradient to '0'.
- Saturated tanh also has 0 gradient.

Saturated \Rightarrow
sigmoid output = 0, 1.

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}} &= \delta_j g'(o_j) o_i \\ &= \delta_j g(o_j) [1 - g(o_j)] o_i \\ &= \delta_j \underbrace{o_j (1 - o_j)}_{g'(o_j)} o_i\end{aligned}$$

If $o_j = 1$ or 0 ,
 $g'(o_j) = 1$.

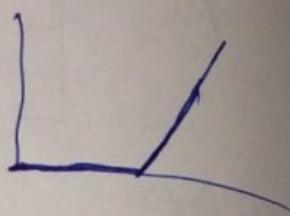
Improper weight initialization can cause this saturated sigmoid/tanh.

Solve to saturation - Use ReLU or maxout in deep

(*) Sigmoid good only for single layer neural network

Rectified linear unit (ReLU)

- Does not saturate (in the region)
- very computationally efficient
- Converges much faster than sigmoid or tanh (6 times faster)
- Not zero-centered output

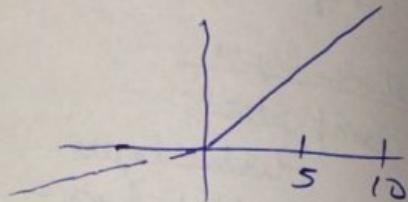


$$g(x) = \max(0, x)$$

Leaky ReLU

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

- Does not saturate (in both the +ve and -ve region)



(*) If we use completely linear activation fns, output will also be linear - NN will identify only linear boundaries.

- 1) Start NN with right weight - Pretraining
 - Solved issue of saturation with sigmoid perceptron
- 2) ReLU solved need for pretraining and lead to explosive use of NN
 - pretraining of sigmoid perceptron took a lot of computational time

Issues with initial weights

- Smaller or larger weight initialisation leads to sigmoid/tanh perceptron saturation. [weights $\rightarrow [0, 1]$] If weights are initialised in $[0, 1]$ range, small/large weights lead to quick saturation
- USE
 - a) supervised learning (pretraining) - only in sigmoid/tanh
 - b) Xavier initialisation
 - Not needed in ReLU but still used for faster learning.

Why weights in $[0, 1]$?

- Normalisation

Linearity in ReLU not affecting 'smooth learning'?

- It's an issue only in single layer NN. With multiple layers, no issue of non-smooth learning.

weight/bias regularizer - used to overcome overfitting

Issues with objectives fn

1) NN objective is non-convex - learning might produce weights and bias that corresponds to local-minima.

Solutions per heuristics (not math)

- Use stochastic gradient instead of batch gradient as it might skip local minima

Overfitting in NN

- High expressivity increases risk of overfitting -
- # parameters often larger than the amount of data

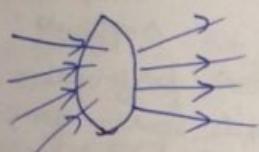
Solutions:

- Regularisation
- Dropout
- Data augmentation
- Unsupervised pretraining
- Batch normalisation

24/11/17

Xavier Initialiser

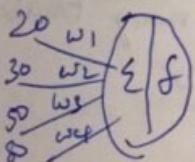
$$w_{ij} \sim U\left(-\frac{\sqrt{6}}{\sqrt{n+m}}, \frac{\sqrt{6}}{\sqrt{n+m}}\right); n \text{ and } m \text{ number of inputs & outputs}$$



$$n=4 \quad m=4 \quad w \sim U\left(-\frac{\sqrt{6}}{\sqrt{18}}, \frac{\sqrt{6}}{\sqrt{18}}\right) \rightarrow \begin{matrix} \text{take random values} \\ \text{in this range} \end{matrix}$$

Gradients will be in the same scale. So, vanishing gradients issue will be mitigated.

Why to normalize the input of NN in $[0,1]$ or $[-1,+1]$ range?



1) With large weights, $a = \sum w_i x_i$ is large, so saturation happens fast

$$2) \frac{\partial E}{\partial w_{ij}} = \delta_j g'(a_j) \delta_i$$

If this is high, gradient will be large, and hence learning is ^{not} smoother.

⊗ NN can be used to simulate all 'objective fn' based algorithms.

- SVM (to find large margin)

- Logistic regression / cross entropy (Classification)

- Squared error (for Regression)

⊗ NN cannot be used to simulate 'Tree learning', 'KNN' etc as they don't have objective fn.

25/1/17 With multiple layers, lower layers are trying to learn smaller pieces of the image/input.
So, lower hidden layer perceptions can learn some parts of image like understanding parts like '^', '(', ')', ']'.

Learning through a hierarchy of features

This is 'Supervised' deep learning, distributed among objects of ^{life}.

So, feature extraction possible with deep NN.

Human voice also has hierarchical features

- ① Shallow NN - Only pattern recognition
Deep NN - pattern recognition + feature extraction (with hierarchical pattern)

CNN is a modification of DNN that helps us to understand the whole story by reverse engineering and seeing the intermediate features extracted.

- ② The last layer of DNN is just 'softmax' for selection.
The last hidden layer o/p → softmax → digit.
Suppose, we feed last hidden layer o/p → tree/random forest → might outperform softmax.
Direct input image of 784 pixels → tree/random forest → Poor performance:
can't learn using individual pixels
- ③ How to make machines understand varied images of a person too
(e.g) old age, barbershop, diff hair styles, disguise etc
In Facebook, after an image is taken, they make many transforms of the image and feed all those images for learning.

26/1/17

DNN can discover 'non-linear decision boundary' for classification. And non-linear for regression.
- Handle overfitting with regularization.

L1- Regularisation:

$$J(W) = E(W) + \lambda |W|$$

- Penalize the weight by absolute terms.
- Encourages.

L2 Regularisation $J(W) = E(W) + \lambda |W|^2$

- Weights with less influence on error get smaller value.
- Penalizes the weight by squared terms.

Extracting features from images

- Extract features by applying a sliding window fr across images, called as convolution
- Why sliding window?
 - Detect the same feature at different positions in the input image

Convolution is = $W \times X$

filter weights image patch

3×3 image patch on a large 5×5 image gives

3×3 ways of patching. So, we get a resultant 3×3 matrix called 'Feature Map'.

0	-1	0
-1	4	-1
0	-1	0

→ This filter ^{highlights} gives the 'edges' of images.

Previously, we had to discover the 'filter' array/matrix manually. CNN discovers it automatically but we have to specify the filter size (e.g. 3×3) .

Parameters for convolutional filters

Size of feature maps controlled by following parameters:

- * Depth - No. of filters to use
- * Stride - No. of pixels by which we slide our filter matrix over the input matrix.
- * Padding - Adding zeros around the borders; To give importance to edges also.

7×7 input $\xrightarrow{3 \times 3 \text{ filter}}$ 5×5 output

7×7 input + 0-padding $\xrightarrow{(gives 9 \times 9 \text{ input})}$ $\xrightarrow{3 \times 3 \text{ filter}}$ 7×7 output.

So, we padding to make the output have the same size as input

area input $\xrightarrow{b \times b \text{ filter}}$ $(a-b+1) \times (a-b+1)$ output.

filters are usually 'squares'; never rectangles.

Rectified feature maps

- ReLU is an element-wise operation (applied per pixel) and replaces all -ve values in the feature maps by '0'.
- 'Black' on image has -ve values; remove them.
- Applied on top of 'convolution'.

Max pooling

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

max pool with 2x2
filters and stride 2

$$\text{Max}(1,1,5,6) = 6$$

6	8
3	4

Pooling gives 'Shift invariance' \rightarrow Even if image is shifted across the photo, we can recognize it.

Fully Connected layer

- The o/p from convolutional and pooling layers represent high-level features of the input image
- The Convolution + Pooling layers act as feature extractors from the input image while fully connected layer acts as a classifier.
 n feature map \rightarrow n distinct perceptrons.

Feature map = o/p of 1 filter going through entire image.
Suppose 7×7 $\xrightarrow{3 \times 3 \text{ filter}}$ 5×5 feature map.

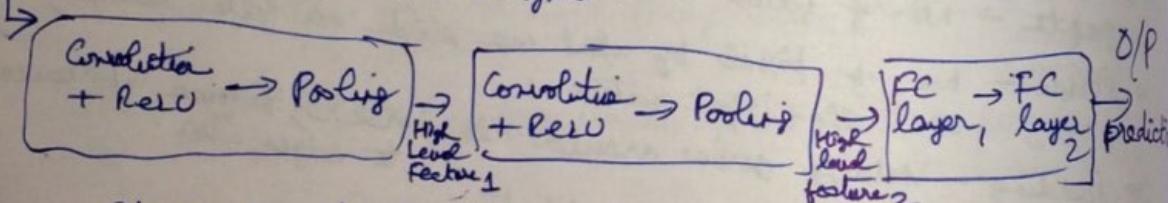
How do we slide thru' the image?

5×5 feature map \Rightarrow 25 features

- So, we use 25 similar perceptrons (of same filter weight) to extract 25 features from an image.

$$\begin{matrix} (e_1) & 0 & -1 & 0 \\ & -1 & 4 & -1 \\ & 0 & -1 & 0 \end{matrix}$$

Input image All perceptions of the same filter share the same weights.



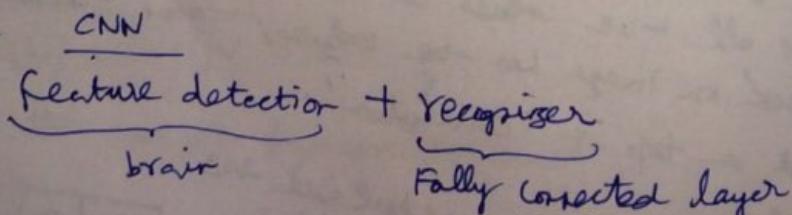
Sparse connectivity + Weight sharing happens in CNN

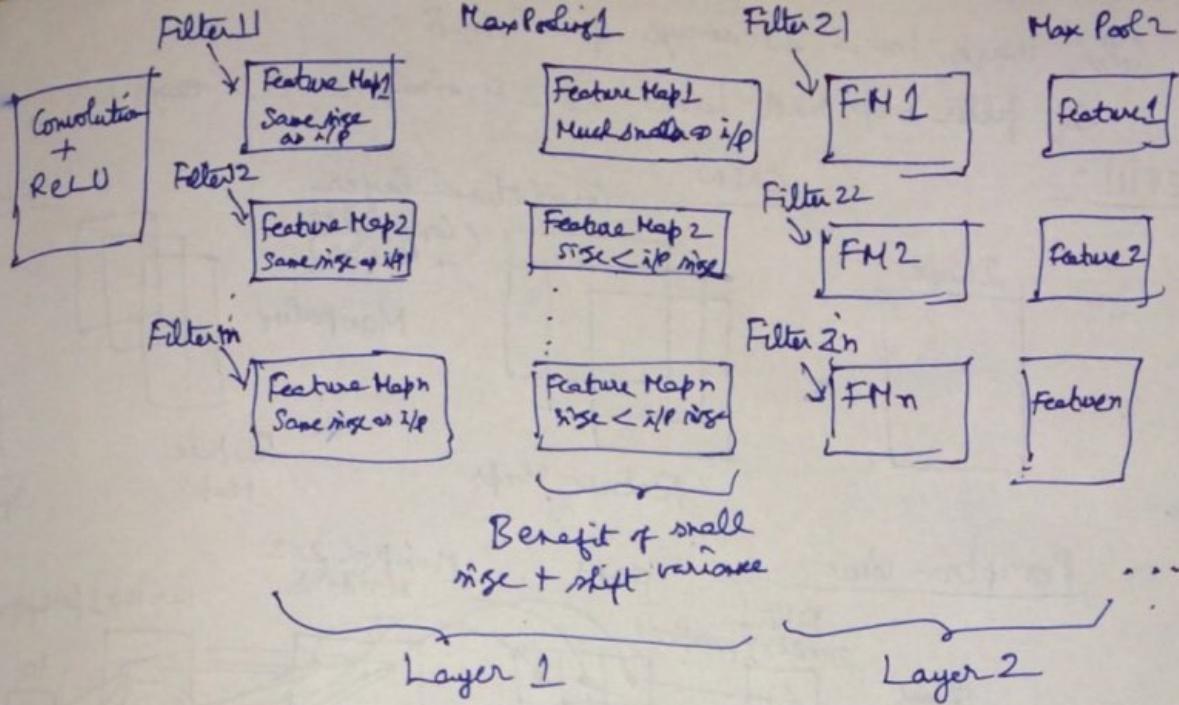
Image cut into 5×5 parts
Each part is connected to one of the 25 filters $\left\{ \begin{array}{l} \text{All pixels in one part connected to a} \\ \text{perceptron} \\ \text{in another part of image connected to another} \\ \text{perceptron} \end{array} \right.$
W.X for each of the 25 parts

The weights of the filter learnt automatically by back propagation \equiv automated feature extraction.

ReLU used mainly to handle 'Vanishing error gradients', but as a side effect, it also eliminates the -ve values.

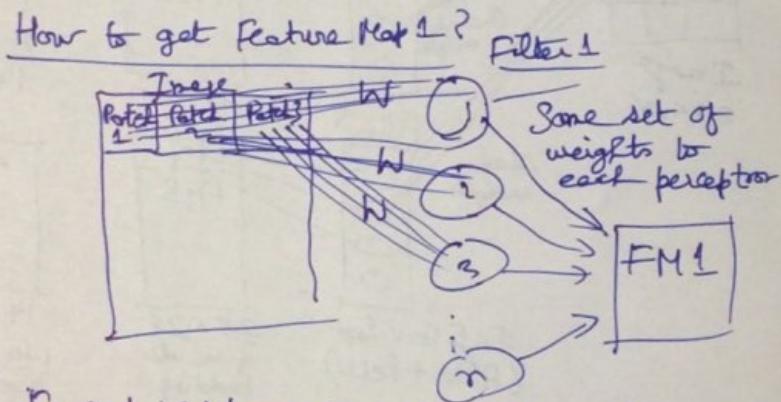
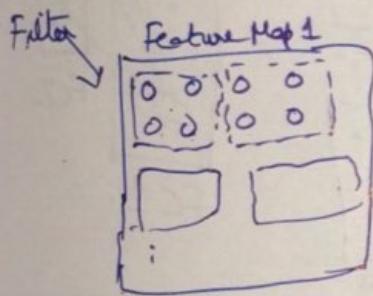
27/1/2017





A small patch in MaxPool o/p corresponds to a higher size patch in input image. The group of maxpool o/p's highlight diff. high-level features across diff. location in the image.

Use a DNN (fully connected) on the o/p of MaxPool n to recognise non-linear relationship between high-level features and the input image.



Local patch connectivity
↓
Shared weights

Each percept connected to one patch only. Same weights to all perceptions in Filter 1.

↓
Faster learning in CNN ∵ fewer weights
Also not fully connected.

∴ CNN + ReLU + MaxPool. → DNN
(Faster learning,
better performance).

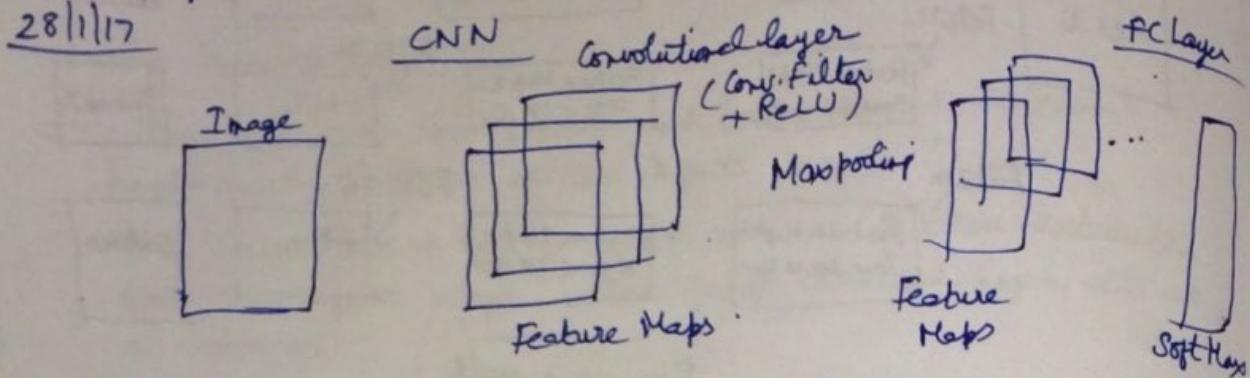
Smaller patch size ⇒ more patches ⇒ more perceptions.

Back propagation from MaxPool o/p

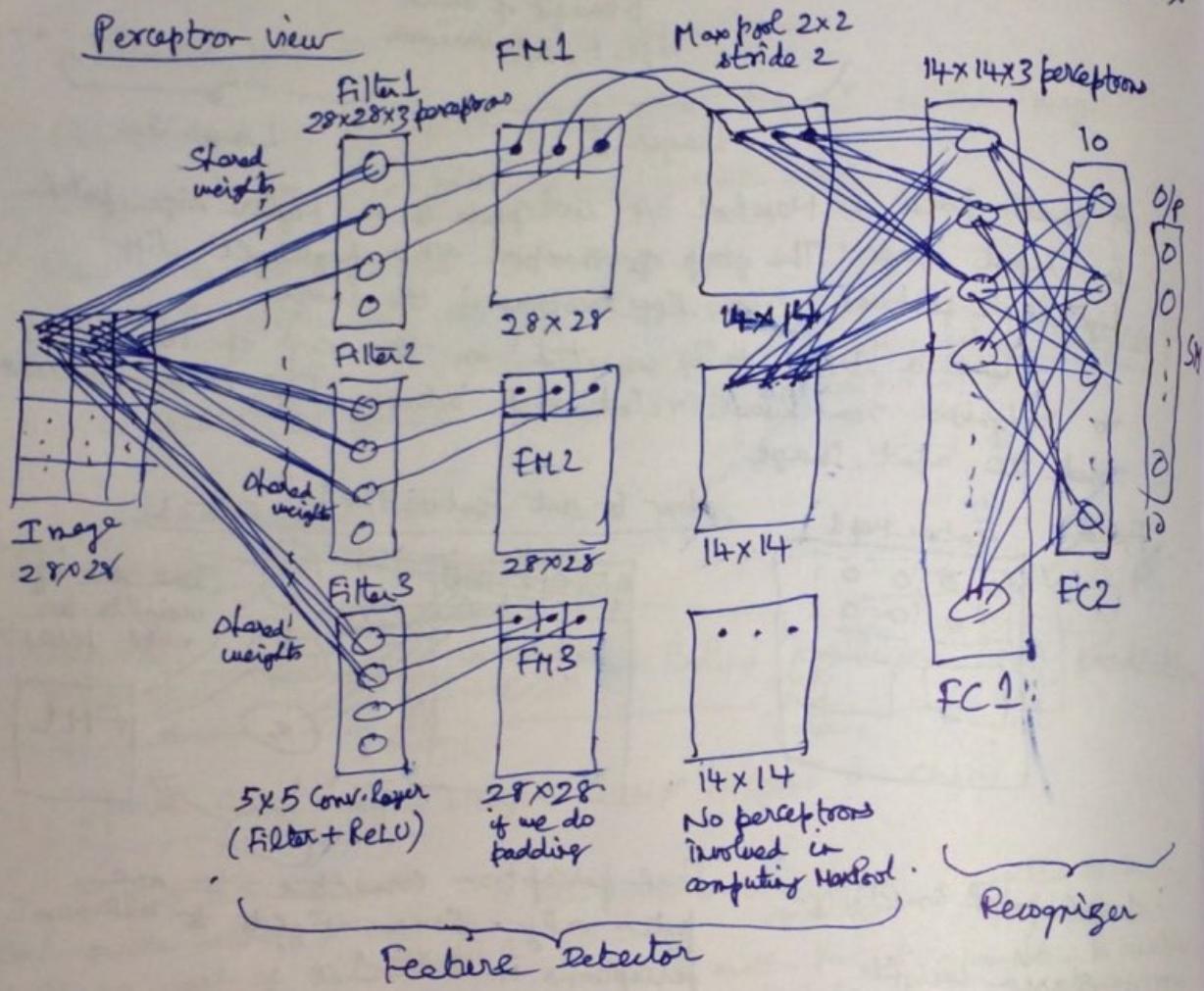
Suppose max. value changed from 8 to 6. Then, we need to update ^{all} the FM data corresponding to 8, to 6. Other FM parts are not changed.

Color image has 3 2D arrays for R/G/B
 So, filter map must have $3 \times 2D$ arrays = 3D arrays.

28/11/17



Perception view



(*) Size of filter \propto Feature level.

If very small filter size, we extract very low level features
 The smaller the filter size, the more CNN layers are needed

- Kaggle - Facial Keypoints Detection
 - Dogs vs. Cats
 - Right Whale Recognition