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, (harder than image classification), Object detection – identifying the objects in an image.

Deep learning everywhere – Internet and cloud – image classification, speech recognition, language translation, sentiment analysis
- Medicine & biology – cancer cell detection, diabetic grading, drug discovery
- Media and entertainment – video captioning, video search, real time translation
- Security and defense – Face detection, video surveillance, satellite imagery
- Autonomous machines – 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 – true or –ve?

Twitter
Facebook — New product

What is deep machine learning?

Traditional ML

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

Speech → Hand-crafted features MFCC (fixed) → Your favorite classifier (learned) → \'dēp\

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

How to automate feature extraction?

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

Neural network — 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 meaningless because there is no hierarchial learning.

Engines for Modern AI
- Google — Tensor flow (open-sourced) ** — Most popular, High movement
- Facebook — Big Sur
- Microsoft — CNTK
- Berkley — CAFFE

Course about 3 deep models
- Recurrent Neural Network (RNN) — NLP
- Deep Neural Network (DNN)
- Convolutional Neural Network (CNN) — Image

} Tensor flow supports all 3.

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

With GPU	Without GPU (only CPU)
10 min	4 hrs
3 hrs	1 week

} Time reduction with GPU

Neural Network — Structured data → only model building
Unstructured data → both feature extraction & model building.

For 'Neuron' – R & Python both help.

Tensor Flow (not in R) — Python, Scala, ~~C++~~

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

13/12/16

P1: Perceptron / Neuron
P2: Neural Network
P3: Neural Network for model building
} Traditional life cycle – Kaggle problem I

P4: Convolutional (2012) neural network – feature extraction in images { Image classification from Kaggle }

P5: Recurrent neural network (LSTM) – feature extraction in text / natural language (Sentiment classification / Spam classifier in Kaggle)

P6: Pretraining (2006) → Deep Boltzmann machines, Auto Encoders, Deep belief network → to initialize neural networks

(Deep learning life cycle)
P7: Deployment of deep learning models

Python → Jupiter IDE → Anaconda distribution

"Tensor Flow" 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
  - Batch gradient descent
  - Stochastic gradient descent
  - Batch stochastic gradient descent

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..
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 (whose/what 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.

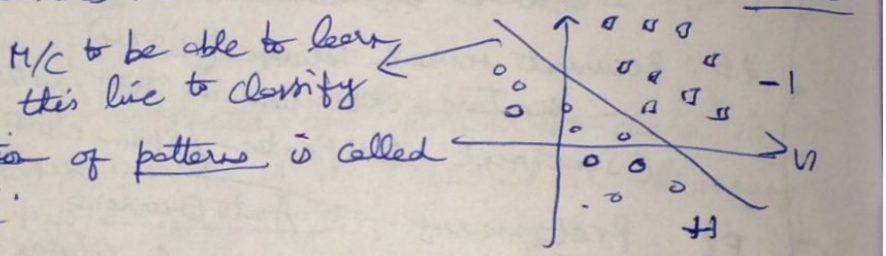
14/12/16

iris_train.csv analysis.

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

How to make machines learn some patterns?

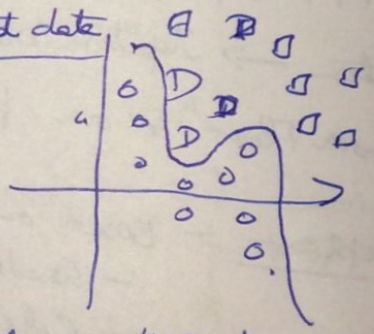
Linearly separated data



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.

Non-linearly separated data



Perceptron

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

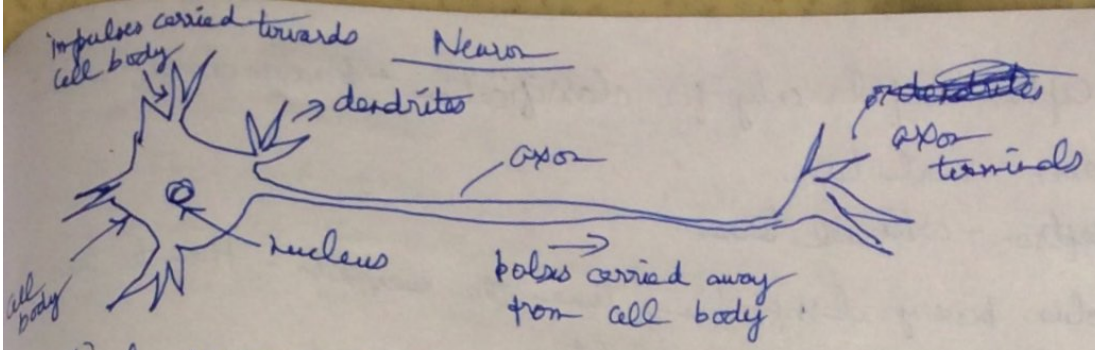
Neural networks are ultimately useful for "feature extraction". We see how to use perceptron/neuron for classification simply to lay the foundation for higher understanding.

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

Neuron – will fire/not fire based on threshold signal/voltage.

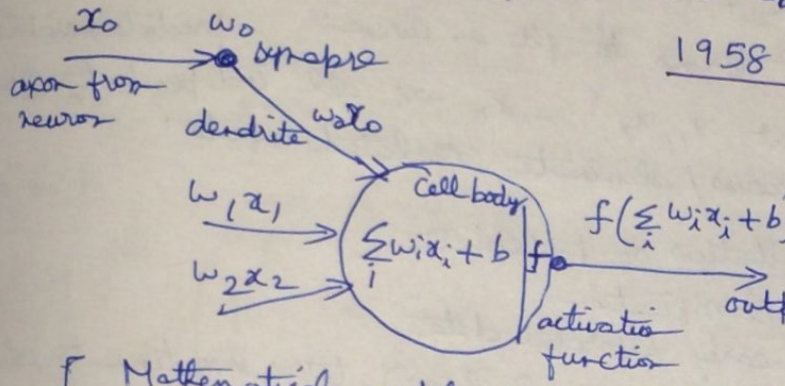
Fire ⇒ signals get transmitted out of a neuron and reach other neurons.

The same signal gets passed on to multiple neurons.



Neuron

- Each neuron gets inputs from multiple neurons.
- How much each neuron benefits from another neuron is dependent on 'Synaptic weights'.
- We have about 10^11 neurons with each about 10^3 weights.



[Mathematical model of neuron]

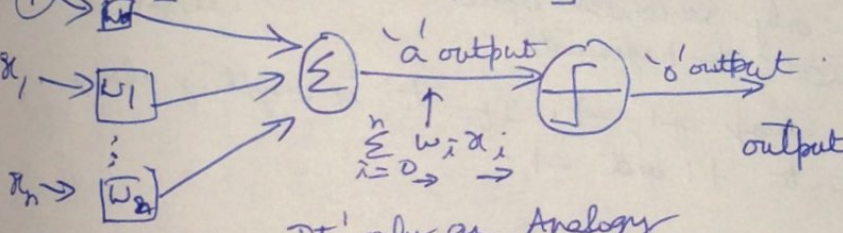
1958 : Frank Rosenblatt Perceptron

x_i – input from other neurons

- Solves 'binary classification' only. Simple training model/algorithm.
- Built using specialized hardware

- 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 – to control when to 'fire'.

[Fundamental unit of neuron]



$$\text{output} = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$$

(Step function)

It's only an Analogy

- 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, $\text{output} = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i - \theta \geq 0 \\ -1 \text{ if } \sum_{i} w_i x_i - \theta < 0. \end{cases}$

Challenge is to find the "weights" for a particular data.

(X) Assume $x_0 = 1$, $w_0 = b$ so that

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

So, challenge is to find w_i only; not

Perceptron 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 $w_0 x_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$ is the equation of an n-dimensional plane

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

(X) Neuron assumes that $x_1, x_2, \ldots x_n$ 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 → no logic to figure this out.
- Can handle only numerical inputs. Convert 'categorical' 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 $\geq \theta$ → cell "fires"

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

(bias) $< \theta$ – does not fire

This is modeled as $\sum_{i=0}^{n} w_i x_i > 0 \rightarrow$ output is +1

$\leq 0 \rightarrow$ 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 perceptron

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

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

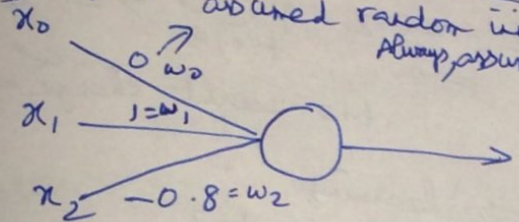
a) Compute the output of perceptron, 'o'

b) Update each weight, w_j, as follows:

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

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

$w = \begin{bmatrix} 0 \\ 1 \\ -0.8 \end{bmatrix}$ = random guess.



assumed random initial weights

Always, assumed $x_0 = 1$; assumed w_0 = randomly chosen

x_0	x_1	x_2	Output
1	1	2	1
1	−1	2	−1
1	0	−1	−1

[Currently using $\eta = 1$ for simplicity]

Epoch-1

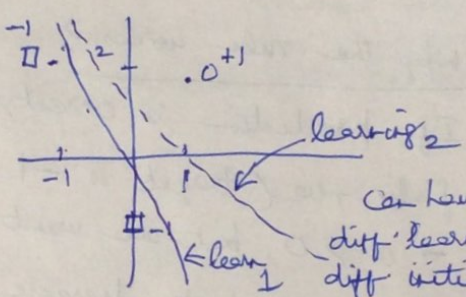
$x_0 = 1$, $x^{(1)} = x_0^{(1)}, x_1^{(1)}, x_2^{(1)}$, $w^{(1)}$

b) ① $\begin{bmatrix} 0 \\ 1 \\ -0.8 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix} = 0 + 1 - 1.6 = -0.6 = a$

$f(-0.6) = -1$ (Mismatch with out)

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

$= \begin{bmatrix} 2 \\ 3 \\ 3.2 \end{bmatrix}$ Refined (new weights)



Can have diff. learnings for diff initially guessed weights

Becomes linearly inseparable by Adding extra row, $\begin{bmatrix} -1 & 0 & +1 \\ x_1 & x_2 & \text{output} \end{bmatrix}$

② $\begin{bmatrix} 2 \\ 3 \\ 3.2 \end{bmatrix}\begin{bmatrix} 1 \\ -1 \\ 2 \end{bmatrix} = 2 - 3 + 6.4 = 5.4 = a$ $\quad f(a) = f(5.4) = +1$

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

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

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

Epoch = One complete pass over the entire training set.

Epoch-2

$\begin{bmatrix}-2\\5\\1.2\end{bmatrix}\begin{bmatrix}1\\1\\2\end{bmatrix} = -2+5+2.4 = 5.4 = a$ $f(a)=$ $f(5.4)=+1$ (matches O/P 1)

So weights don't change

$W_{new} = W_{old} + \theta[\text{scaling}]$

$\begin{bmatrix}-2\\5\\1.2\end{bmatrix}\begin{bmatrix}1\\-1\\2\end{bmatrix} = -2-5+2.4 = -4.6 = a$ $f(-4.6)=-1$ (matches O/P 2)

No weight change again

$\begin{bmatrix}-2\\5\\1.2\end{bmatrix}\begin{bmatrix}1\\0\\-1\end{bmatrix} = -2+0-1.2 = -3.2 = a$ $f(-3.2)=-1$ (matches O/P 3)

No weight change again!

Since no weight changes, stop 'learning'.

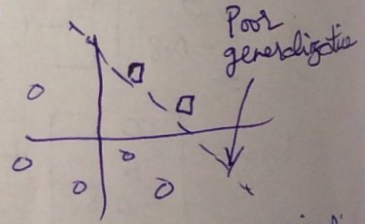
Final learning is, $w_0 = -2$, $w_1 = 5$, $w_2 = 1.2$

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 +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

- OO concepts
  - Python types
  - Python data structures (Containers)
  - Control statements, User defined functions, Lamda expressions
  - Packages for ML
    - NumPy — Matrix/Vector multiplications
    - SciPy (scientific python) — Statistics
    - SciLearn — Perceptron, LR, NN } Basic
    - MatPyLab
    - Pandas — Data Frame
    - TensorFlow — RNN

Other popular Python packages are CAFFE, Theano for ML (UC Berkley)

Download Python 2.7

(Latest Python 3.X is not backward compatible)

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 (*.Ipynb) — supports literate programming, but not so 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
- Enaes 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

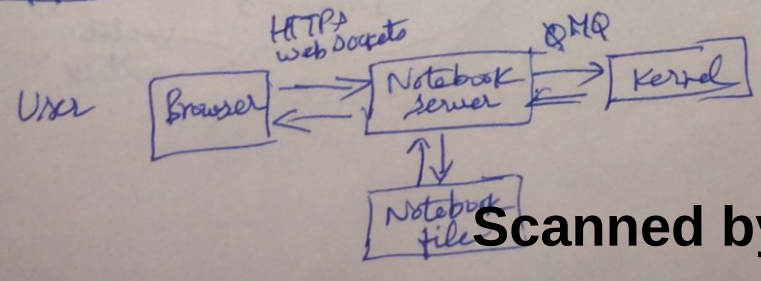
Literate programming — Explain program; write code. Coding to be natural like using a notebook.

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]: a = 10
print a
10

↑ Input

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 any 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 explanations. (People nowadays publish books/chapters using Jupyter)

R : ggplot :: Python : matplotlib
R : dataframe :: 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/data structures
- String
- List
- Tuple
- Dictionary/Map
- Array — Vector, Matrix
- Set

19/12/16

- Perceptron
- Other perceptron
  - Sigmoid — Logistic Regression
  - Linear — ADALINE
  - Rectified linear perceptron

  } Learning Strategy
- 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) — Image
- Principles
- Architecture
- Learning
- Applying CNN

RNN (Recurrent NN) — Sequence data / Natural language
- Principles
- Architecture
- Learning
- Applying RNN

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

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

List = An indexed container of "heterogenous" elements

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

Tuple = immutable list; cannot be modified after creation

21/12/16

Dictionary / Map = Collection of key-value pairs {"key": value}

DataFrame — 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 'cheques'.

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

Mirly RNN

9) TensorFlow, Torch, CAFFE, Keras, CNTK — Deep NN frameworks

- List is "heterogenous"
- Array is "homogenous"

Black & White Image = 28 x 28 matrix. Each cell value ∈ [0,255]
(Digit Recognizer)

Color Image = 3 arrays of 28×28 matrix each.
(R, G, B). Each R|G|B cell value ∈ [0,255]

Input Image has 785 columns. 784 = 28×28.
+1 column for O/P ∈ [0,9].

We don't want 'array' to read train.csv for digit recognizer because we want to contain O/P label as 'Categorical column'.
So, need to use "DataFrame".

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

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.

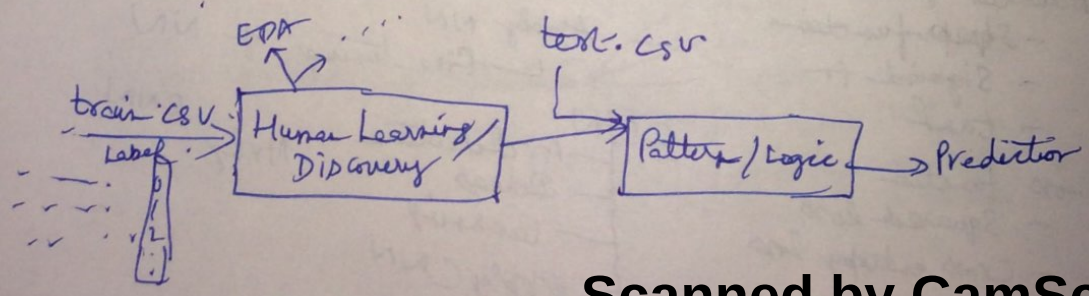
(R)DBMS = almost obsolete
(H)DFS = presently used; distributed file system

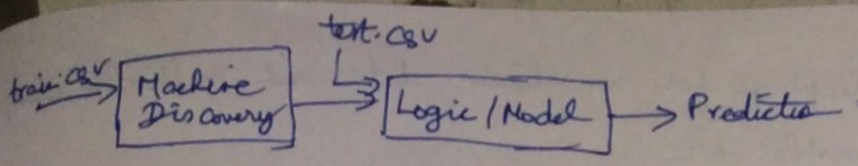
digit_train.describe() — mean, min, max, sd, 25%, 50%, 75% (quartiles)
— thus it tells about "central tendency" and "spread".

- Categorical variables not shown in describe() — Python handled categorical types very late.
- Vectorization NOT by default on basic data types. Vectorization supported only in np.array (from NumPy package)

23/12/16

Perceptron
Neural Network } Approaches for a m/c to learn a pattern from train data.
⋮

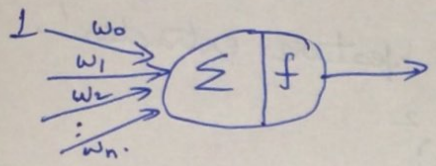




ML approaches

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 of neuron
   - Perceptron } Speciality of this approach is "feature extraction"
   - Neural Network } Using them for classification to lay foundation

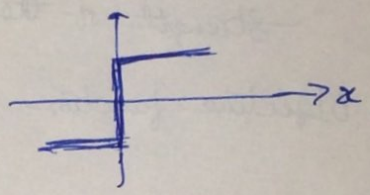
Perceptron / Neuron



Step function

$f(x) = 1, \; x > 0$

$-1, \; x \leq 0$



Company offer

Package good? (0.7) — Yes / No.

Work environment (0.3) — Good / Bad

Domain (0.5) — OK / 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,
$w_j = w_j + \eta (y - 0) x_j$
↓
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) soln is found → Decision boundary not great.

How to adapt Perceptron for non-linear data and multi-class clamp?

For non-linearly separable data

- Modify objective function - use gradient descent algorithm
  - Batch
  - Stochastic
  - Stochastic mini-batch
  - try to minimize objective for each w_i
- (*) - step function (Activation fn) 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)

  Neurons named after the activation function
  - Sigmoid neuron, etc.

  Step fn ⇒ ~~Step~~ neuron Thresholded

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

- Strength of these neurons is 'feature extraction'.

$$\text{Objective function} = \sum_{i=1}^{n} (y_i - o_i)^2$$

$$= \sum_{i=1}^{n} \left[y_i - f\left(\sum_{i=0}^{n} w_i x_i \right) \right]^2$$

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 ~~optimal~~ 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

$W = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix}$ $X = (x_1 x_2 \cdots x_n)$

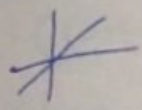
WX = dot product = 1×1 matrix

- Find W that minimizes $E(W)$
- Solve using gradient descent
  - Batch gradient descent
  - Stochastic gradient descent
  - Stochastic mini-batch gradient descent

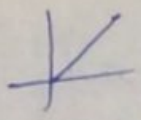
$$w = w - \eta \times \text{gradient}$$

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

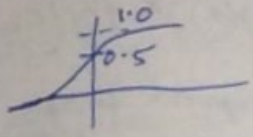
Common activation functions



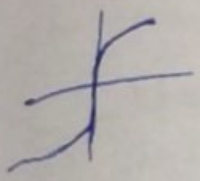
Linear $g(z) = z$



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

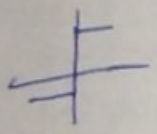


Logistic (sigmoid) $g(z) = \frac{1}{(1+e^{-z})}$ $\in [0,1]$.

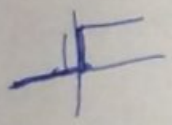


Hyperbolic tangent (sigmoid) aka (tanh) $g(z) = \frac{e^{2z}-1}{e^{2z}+1}$ $\in [-1,+1]$.

These are just smoothened versions of unit step functions



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



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

Why e^{2z} instead of e^{z}? Goal is not just $[-1,+1]$ but a smooth curve. Every step fn has $[-1,+1]$ 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:

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

$o^{(n)} = g(w_0 x_0 + w_1 x_1 + \dots + w_j x_j + \dots + w_n x_n) = g(a)$
$a = \sum w_i x_i = g(\sum w_i x_i)$

$E = \frac{1}{2}\sum_{n=1}^{N} (y^{(n)} - o^{(n)})^2$

$$w_j = w_j + \eta \sum_{n=1}^{N} (y^{(n)} - o^{(n)}) x_j^{(n)} g'(a)^{(n)}$$

} Batch gradient version.

'-' → '+' because $-(-x_j^{(n)})$.

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.

Chain Rule: (for differentiation)

$$\frac{\partial g}{\partial x} = \frac{\partial g}{\partial f_1}\frac{\partial f_1}{\partial x} \Rightarrow \frac{\partial E}{\partial w_j} = \frac{\partial E}{\partial o^{(n)}}\frac{\partial o^{(n)}}{\partial w_j} = \frac{\partial E}{\partial o^{(n)}}\frac{\partial o^{(n)}}{\partial a}\frac{\partial a}{\partial w_j}$$

$[a = \sum w_i x_i]$

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

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

Initial $(0, 1, -0.8)$

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

$\frac{\partial o^{(n)}}{\partial a} = \frac{\partial g(a)}{\partial a} = g'(a)^{(n)}$

$\frac{\partial E}{\partial o^{(n)}} = \frac{\partial \frac{1}{2}\sum (y^{(n)} - o^{(n)})^2}{\partial o^{(n)}} = -(y^{(n)} - o^{(n)})$

$\Rightarrow \frac{\partial E}{\partial w_j} = -(y^{(n)} - o^{(n)}) g'(a)^{(n)} x_j^{(n)}$

$$w = \begin{bmatrix} 0 \\ 1 \\ -0.8 \end{bmatrix} + (1-0.35)\begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}(0.35)(1-0.35)$$

$w = w + \eta(y-o)x\, g'(a)$

$g'(a) = g(a)(1-g(a))$.

$\eta = 1$.

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

↳ for sigmoid perceptron.

~~$g(a) = g_a$~~

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

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

$\therefore w = w + (y - g(a))x\, g(a)(1-g(a))$, Assuming $\eta = 1$.

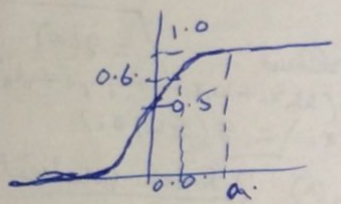
$y_1 = 1$.

So, in epoch 1, for 1st sample

$$w = \begin{bmatrix} 0 \\ 1 \\ -0.8 \end{bmatrix} + (1-0.35)\begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}(0.35)(1-0.35)$$

$$= \begin{bmatrix} 0 \\ 1 \\ -0.8 \end{bmatrix} + (0.147)\begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 0.147 \\ 1.147 \\ -0.504 \end{bmatrix}$$

Informal logic of Sigmoid



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

'z' is evidence = $a = \sum w_i x_i$.

Only if evidence is high enough (threshold), then fire neuron, or, switch your job. ie, like 0.6 > than

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

When to stop learning in Sigmoid perceptron?

Compute Error $E = \sum_{n=1}^{n} (y^{(n)} - o^{(n)})^2$ at the end of each epoch.

If (Error after epoch i − Error after epoch $i+1$) > threshold, then stop.
(Error after epoch i − Error after epoch $i+1$)

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

Objective functions – 3 types

- Squared error loss
- Cross entropy loss
- Log loss

Limitations of Perceptron

- Can only perform binary linear classification
  - although they find better boundary than threshold perceptron
- Solution: Use multiple perceptrons for multiple classifications
  K-perceptrons for K-class classification

- Solution 2 ; Use multiple layers of perceptron but with different non-linear activation functions with bad errors (for non-linear separation).
- Slower learning, due to squared error objective. Use cross-entropy objective fn for faster learning.

Perceptron to solve Titanic problem — convert to all continuous variables.

In multiclass classification with perceptrons, wrt digit-recognizer convert $y_i = [0,9]$ range into $(y_0, y_1, \cdots, y_9)$ 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 fixing

- Each incoming 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

Given 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 → Switch job or don't switch job if evidence ≥ 0 or < 0.
- Sigmoid fn → 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

Diff. 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 perceptron learns very slowly even with initial big mistake (using squared error loss objective function)

$w = w + (y - g(a)) x\, g(a)(1-g(a))$; $\quad y=1,\ o=0.1$

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

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

$y - g(a) = 0.9$.

So, $w = w + (0.9) x (0.9)(0.1)$

$w = w + (0.081)x \Rightarrow$ very slow learning.

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

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

$\Rightarrow$ If $y=0, o=0$, $E=0$
Also, if $y=1, o=1$, $E=0$.

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

output, $o \in [0,1] \Rightarrow \log o < 0$ & $\log(1-o) < 0$.
Actual, y

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

Intuitive reasoning

- Since output $\in [0,1]$, error is always +ve
- Cost tends to be close to zero if o/p approaches actual value

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

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

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. (*)

Hold?

Stochastic-gradient version of cross-entropy function is

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

$a = \sum_i w_i x_i$
$o^{(n)} = g(a)$
$\frac{o^{(n)}}{dw_j} = o'^{(n)}$
$= g'(a) x_j$

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

~~ie, $o' = o(1-o)$~~

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

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

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

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

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

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

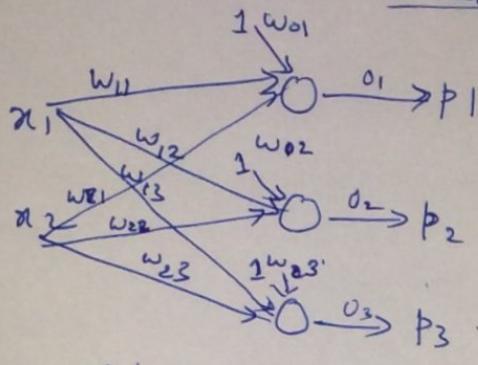
So, we can rewrite $w_j = w_j + \eta(y^{(n)} - o^{(n)}) x_j$ [$\because \frac{\eta}{N}$ can be absorbed in diff. η]

27/12/16

How to use perceptron for multi-class classification?

- Use sigmoid perceptron with cross-entropy objective.

K sigmoid perceptrons for K-class classification



E = Overall error = Sum of errors.

Single Layer Neural Network.

Sigmoid with X entropy

⇓

cross-entropy (aka) negative log

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

$$E = \sum_{n=1}^{N} \sum_{K} -y_K \log o_K - (1-y_K)\log(1-o_K)$$

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

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

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} * \frac{\partial o_j}{\partial a_j} * \frac{\partial a_j}{\partial w_{ij}}$$

$$\frac{\partial a_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(\sum_K w_{Kj} x_K\right) = x_i$$

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

$$\frac{\partial E}{\partial o_j} = -1 \times \frac{\partial}{\partial o_j}\left(\sum_K y_K \log o_K + (1-y_K)\log(1-o_K)\right)$$

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

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

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

$$= \frac{o_j - y_j}{g(a_j)[1-g(a_j)]} * g(a_j)[1-g(a_j)] * x_i$$

$$= (o_j - y_j) x_i$$

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

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

But $\sum_K o_i \neq 1$ with sigmoid perceptrons.

So, change sigmoid to 'softmax'. Softmax useful only for Layer of perceptron.

(X) Softmax = Normalised sigmoid $\frac{p_i}{\sum_j p_j} = \frac{\exp(a_i)}{\sum_j \exp(a_j)}$ $\sum_j p_j = 1$.

28/12/16 How to overcome limitation of 'only linear classification'?
- Add hidden layers of perceptron for non-linear classification.

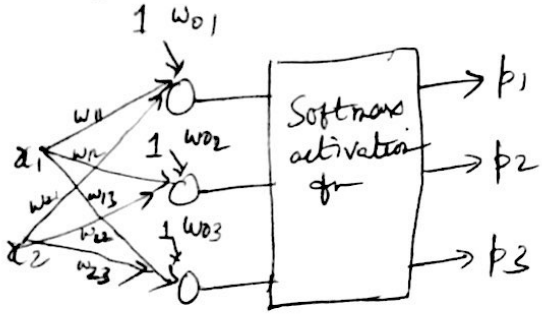
(*) Why not use just 2 perceptrons for 4-class classification?
00, 01, 10, 11 ⇒ 4 possible combined output.
- This won't work because of lack of interpretability of 01, 10, etc.

(*) 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.



$$p_i = \frac{e^{a_i}}{\sum_j e^{a_j}} = o_i$$

$$o_1 = \frac{e^{a_1}}{e^{a_1} + e^{a_2} + e^{a_3}}$$

Note: Softmax is not exactly 'normalised sigmoid'.

$$\text{Normalised sigmoid} = \frac{\frac{1}{1+e^{-a_i}}}{\sum_K \frac{1}{1+e^{-a_K}}} = \frac{\frac{e^{a_i}}{e^{a_i}+1}}{\sum \frac{e^{a_K}}{e^{a_K}+1}}$$

$$= \frac{e^{a_i}}{(e^{a_i}+1)\sum \frac{e^{a_K}}{e^{a_K}+1}}$$

$$\text{But Softmax} = \frac{e^{a_i}}{\sum e^{a_K}}$$

Why Softmax instead of 'Normalised sigmoid'?
In normalised sigmoid, we normalise individual 'S' curve values, which may not be meaningful. Better to use 'S' curve-like Softmax fn.

29/12/16

Softmax fn : Sigmoid for multi-classes [Uses cross entropy]

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

* Properties of softmax perceptron:
  - $0 < O_j < 1$
  - $\sum_k O_k = 1$

* 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

$\frac{\partial a_j}{\partial w_{ij}} = x_i$, 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.

In softmax, $O_j = \frac{e^{a_j}}{\sum_k e^{a_k}} = g(a_j)$ $\quad \left[\frac{d}{dx}\left(\frac{u}{v}\right) = \frac{u'v - uv'}{v^2}\right]$

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

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

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

$$= g(a_j)\left(1 - g(a_j)\right).$$

x_1	x_2	Output
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

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

Add this column to the matrix

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

28/12/16 How to overcome limitation of 'only linear classification'?
- Add hidden layers of perceptron for non-linear classification.

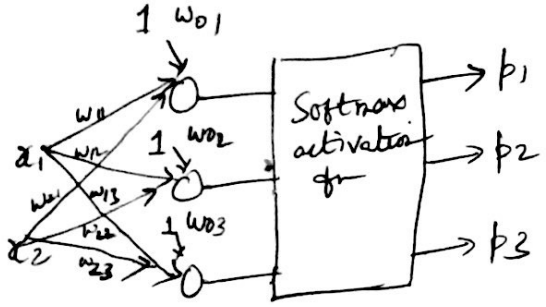
(*) Why not use just 2 perceptrons for 4-class classification?
00, 01, 10, 11 ⇒ 4 possible combined output
- This won't work because of lack of interpretability of 01, 10, etc

(*) 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.



$$p_i = \frac{e^{a_i}}{\sum_j e^{a_j}} = o_i$$

$$o_1 = \frac{e^{a_1}}{e^{a_1} + e^{a_2} + e^{a_3}}$$

Note: Softmax is not exactly 'normalized sigmoid'.

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

$$= \frac{e^{a_i}}{(e^{a_i}+1)\sum \frac{e^{a_K}}{e^{a_K}+1}}$$

$$\text{But Softmax} = \frac{e^{a_i}}{\sum e^{a_K}}$$

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.

Softmax fn : Sigmoid for multi-classes [Uses cross entropy]

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

* Properties of softmax perceptron:
  - $0 < O_j < 1$
  - $\sum_k O_k = 1$

* 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

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

$\frac{\partial O_j}{\partial a_j}$ 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.

In softmax, $O_j = \frac{e^{a_j}}{\sum_k e^{a_k}} = g(a_j)$ $\quad \left[\frac{d}{dx}\left(\frac{u}{v}\right) = \frac{u'v - uv'}{v^2}\right]$

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

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

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

$$= g(a_j)\left(1 - g(a_j)\right).$$

x		
x_1	x_2	Output
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

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

Add this column-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'.

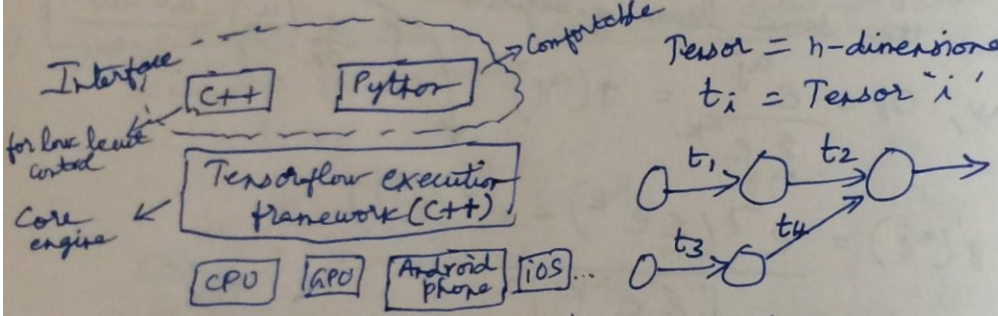
If Softmax AF + X entropy objective,
Shuffle train data before each 'epoch' begins and use "stochastic gradient descent". Shuffling enables randomisation in stochastic gradient version.

After all the learning for 3-class classification, 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' ⇒ Randomly choose 'class'.
If Softmax o/p = (0.5, 0.3, 0.2) ⇒ class C1
(0.3, 0.5, 0.2) ⇒ class C2
(0.2, 0.3, 0.5) ⇒ class C3

30/12/16

Tensor flow
↓
n×n matrix

↳ It allows us to build graphs for neural net computations
Has built-in learning algorithms for any neural net computation
Can run the graph on CPU, GPU, mobiles.



Tensor = n-dimensional array
t_i = Tensor 'i'

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 → install CentOS7 → Tensorflow on top of this.

CentOS7 → Community enterprise – close to RedHat Linux.
VirtualBox → To install Linux on windows machines
Anaconda for Linux – Install Python 3.5 version (64-bit installer)
– Compliants with 2.7 version.

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

– This is why we use matrix multiplication. For concise expression of equations.

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

> bash .sh file

2/1/17 root password - root123

Tensorflow toolkit hierarchy

tf.[Contrib].learn } To simplify Python coding

Components useful when building custom NN models → tf.layers, tf.losses, tf.metrics

Python API gives full cabre → Tensorflow Python (100 lines of code = 10 lines using above package) like train() method in caret package (or) sklearn in Python.

C++ API quite low level → Tensorflow C++

CPU | GPU | TPU | Android

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, eg, softmax computation
- Edge : Input for nodes = tensors (n×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 "required" Python packages.

To install 'Theano' 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 versions will cause problems; also cannot revert from Tensorflow to default Python. Hence need separate environments.

* In CentOS 7 on 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 . (When 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

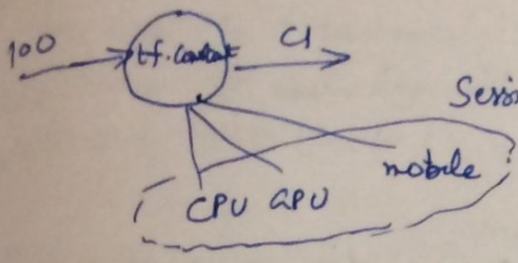
To enable internet access on conda OS VM,
  - become a super user ($ su) and give root password
  - vi /etc/sysconfig/network-scripts/ifcfg-enp0s3
    - change 'onboot' value from 'no' to 'yes'

To check which version of spyder or jupter is active
  - which spyder
  - which jupyter

To invoke the IDEs
  - $ spyder
    (or)
  - $ jupyter notebook

3/1/2017

Tensorflow execution



Session hides the execution mechanism — whether to run it on CPU, GPU, mobile, etc. Hides the device

Tensorflow – always construct a graph - Not a regular programming model.

Operation ≡ Vertex
Tensor ≡ Edge

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

C1 = tf.Constant(100) # 0-dimensional array.

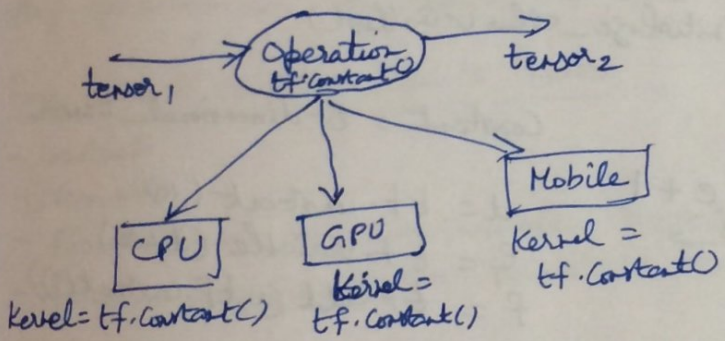
0-dimension = 10
1-dimension = [-]
2-dimension = [- -]

Tensorflow – Lazy execution model

Kernel = Implementation of operations on a specific device
Operation = 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 interfaces/operations

TensorFlow framework has its own data types (like FLOAT, INT_32, etc).

Not depending on Python for more control.

operations are tf.xxx(). Other object.xxx() like z1.eval() and z1.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.

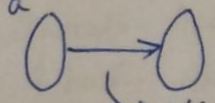
4/1/2017

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.variables

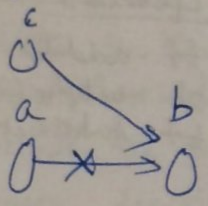
```
a = tf.Constant(10)
b = tf.variable(a+0) # only creates todos
```



Does not create dependencies. But initialises in the right hardware in session.

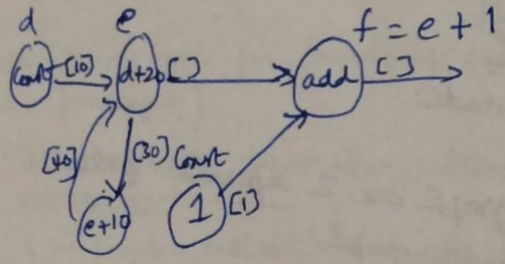
This linkage happens by the call ~~tf.initialise_all_variables()~~ tf.initialise_all_variables().
The call creates ~~dependencies~~ among variables
↳ no execution happens.

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

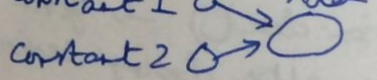
```
$ tensorboard --logdir = ./output
```

http://127.0.0.1:6006 → click on 'Graphs'.

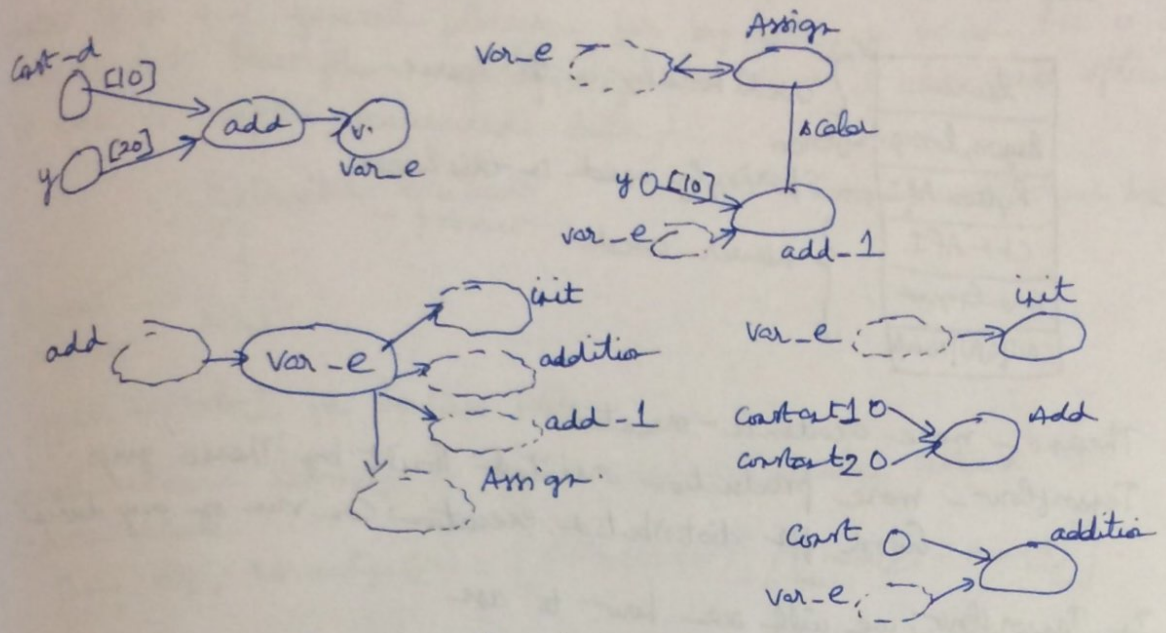
Code

```
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 = tf.Constant(10, name = "Const_d")
e = tf.Variable(d+20, name = "var_e")
f = tf.add(e, tf.Constant(1), name = "addition")
session.run(tf.initialize_all_variables())
update = e.assign(e+10)
```



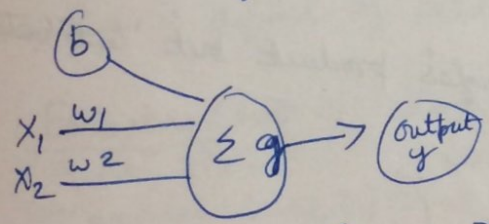
Const - does not have 'init' part.
- no 'input'

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 'panda' to read files into tensorflow

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

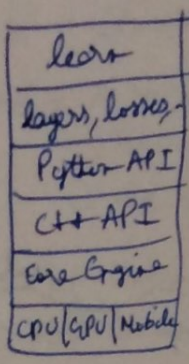


y, output = sigmoid $\left([X_1 X_2] \cdot \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} + b \right)$

placeholder; (input) placeholders; (learnings keep changing) variables.

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'.



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

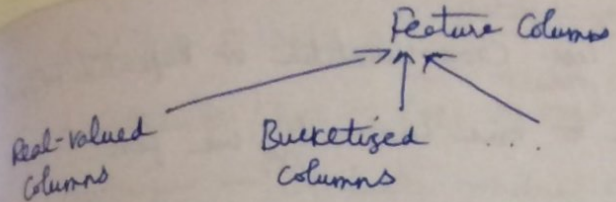
1) Build Graph
   - Tensor
   - operation
2) Execute graph

7/1/2017 Deep network models
- Objective perceptron
- Layers of perceptron
- DNN
- CNN
- RNN

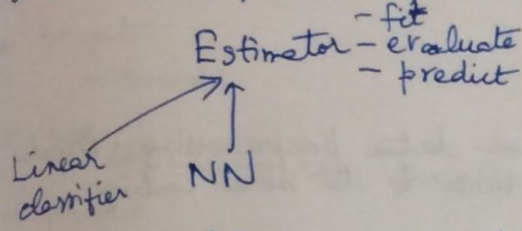
tf.[contrib] package - not Google's product but 'contributed' by open source community.

$ source activate tensorflow
$ conda install pandas

Tensorflow wants all features to be 'feature column' type: To do so, 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;



fit - learning weights and bias

Spyder - ctrl+I for syntax help

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

That's why, normalize(x) = [X - mean(x)] / [max(x) - 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(output) > 0.5 ⇒ 1 output
< 0.5 ⇒ 0 output
} by default

$w_1 x_1 + w_2 x_2 + b = 0$

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

(eg) Linear classifier (Sigmoid Perceptron / Logistic regression)
Linear Regressor
DNN classifier
DNN Regressor
Tensorflow RNN classifier
Tensorflow RNN Regressor

(CNN classifiers/ Regressors still not released)

In real-world work, 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 'hold out'.

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

~~Advantage of using~~

Estimators work on (ie, fit, evaluate, predict methods)
- numpy array
- function returning tensor
  * Advantage is, we can use data preprocessing. Provides 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 = model_function, params = model_params)

Skeleton of model fn:

model_function(features, targets, mode, params): → multiple target cols when target is one-hot-encoded
→ optional

logic of custom model

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 (eg) stochastic gradient descent

learning_rate = η ie, $\omega = \omega - \eta \frac{\partial E}{\partial \omega}$

11/11/2017

[Sub-packages]

tf.contrib
- layers → [operations] (eg) fully_connected
- losses — cross entropy, squared_error
- metrics —
- monitors — for debugging
- distributions — probability distributions
- learn → estimators, input processing, graph actions
- crf — conditional random field
- bayesflow

→ high-level nn ops
- optimisers (eg) optimize_loss
- regularizers
- initializers
- summaries

[Overview of tf.contrib module]

Prediction ≡ output

Output for custom estimators

Tensor containing a scalar loss value.

Objective fn / Loss — (eg) squared error, cross entropy

Optimizer — (eg) 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, fit
* target
  - tensors
* mode
  - 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, [])
predictions_dict = {"key": predictions}

↓

Define loss fn for training/evaluation ← loss = tf.contrib.losses.mean_squared_error(predictions, targets)
loss = tf.contrib.losses.log_loss(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["param"]
optimizer = "SGD")

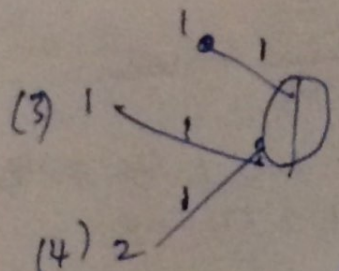
fully connected — all inputs are given to all perceptrons.

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

(*) For classification, use log_loss
For regression, use mean_squared_error.

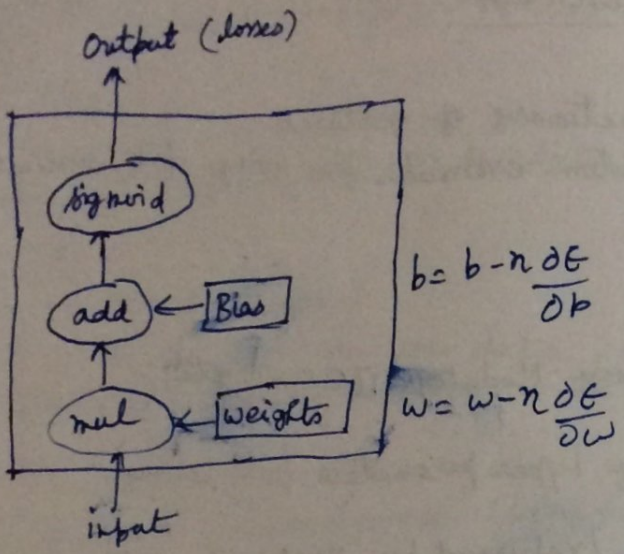
12/1/2017

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



Input (1/2) ; Output (1) 1 + 1(1) + 2(1) =
(3/4) : Output 3(1) + 4(1) + 1(1) =

13/1/2017



nn = learn.Estimator(...)
- No need to call 'Session', everything done internally by Estimator

nn.fit()
- model building algorithm starts here

(X) Open a session to run the graph.
'model_dir' stores the graph - does not store the final model with weights and biases.

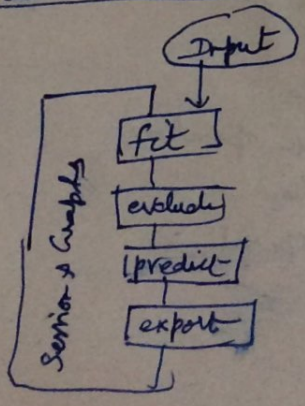
To see graph
$ tensorboard --logdir="./tmpdir/.."
http://127.0.0.1:6006

To save final model, do
$ nn.export(...)

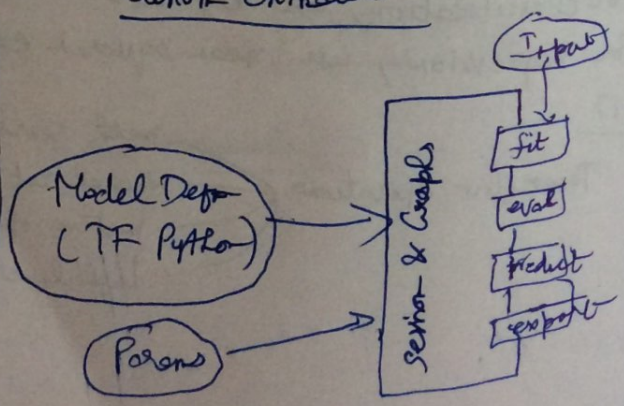
To evaluate/validate unstructured data, due to huge data noise, we do a one-time holdout only.

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

Built-in estimators

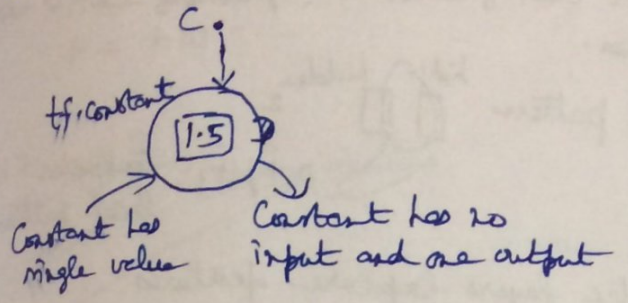


Custom estimators



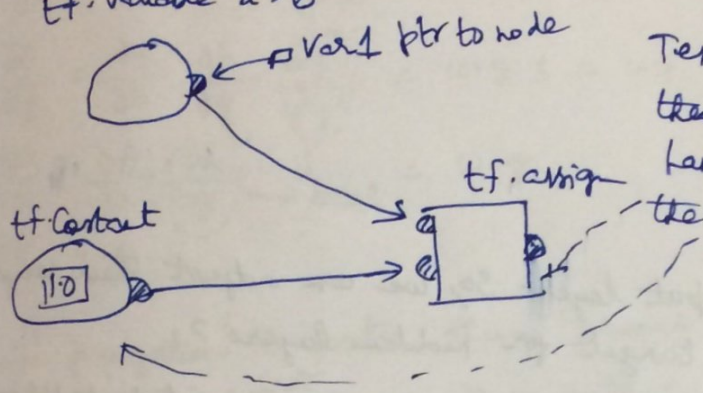
1) `c = tf.constant(1.5)`

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



Tensor Graph

2) `Var1 = tf.Variable(1.0, name = "x")` ← 'x' is the name of node



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

Need to execute

`session.run(tf.initialise_all_variables)`

and only later

`session.run(result)`

↳ (eg) `layers.fully_connected(...)`

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

14/1/2017 **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?

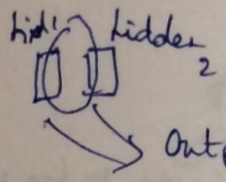
- Individual features may represent local patterns in data
- Complex patterns: combinations of local patterns.

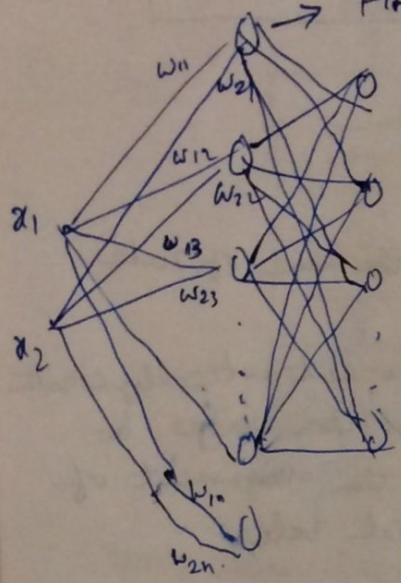
Shallow NN – only one/two hidden layer with lots of perceptrons

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: → NN is not just for classification & regression. It is also to extract features.





Hidden layers capture features.
- 1st hidden layers captures small features
- 2nd hidden layers – more complex features

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

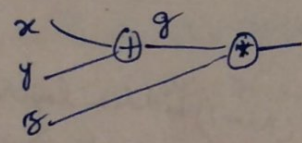
Output layer mostly has 'softmax' activation for interpretability.
But hidden layers does not need interpretability; sigmoid used mostly.

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

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

$q = x+y$

$f = qz$



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

x	y	z	q	f
1	2	3	3	9

$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial x} = z \cdot 1 = z$

$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial y} = z \cdot 1 = z$

$\frac{\partial f}{\partial z} = q$

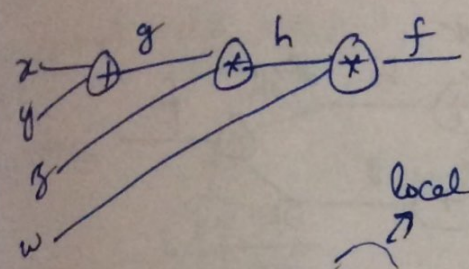
} Find rate of change of output w.r.t all inputs.

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

$g = x + y$

$h = gz$

$f = hw$.



local

$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial h} \cdot \frac{\partial h}{\partial g} \cdot \frac{\partial g}{\partial x} = w \cdot z \cdot 1 = wz$

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

$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial h} \cdot \frac{\partial h}{\partial g} \cdot \frac{\partial g}{\partial y} = w \cdot z \cdot 1 = wz$.

$\frac{\partial f}{\partial z} = \frac{\partial f}{\partial h} \cdot \frac{\partial h}{\partial z} \rightarrow$ local $= w \cdot g$

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

'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 of hidden layers in one pass

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

'Shallow NN' creates more weights to learn; slower

'Back propagation' does 'global learning'.

Giving objective for for each 'layer' and learning is different; it is called 'local learning' — aka, Boltzman machine / Auto encoder.

Issue with back/forward propagation — How to deal with unsupervised learning? — Open research area

Extracting features from images

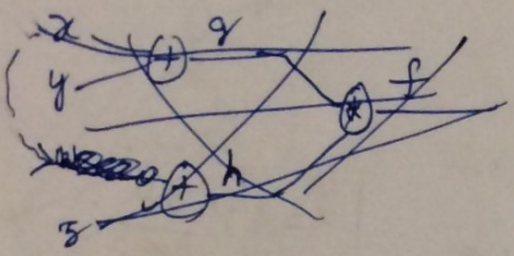
- Still supervised learning
- Hidden layers luckily extracting features

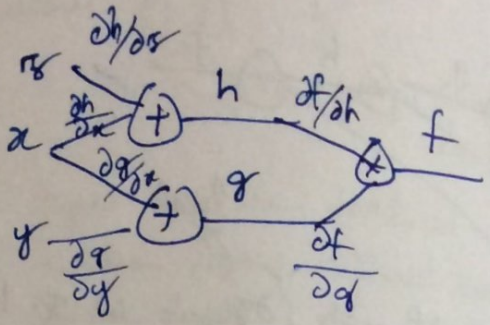
3) $f(x,y,z) = (x+y)(x+z)$

$q = x+y$

$h = x+z$

$f = gh$

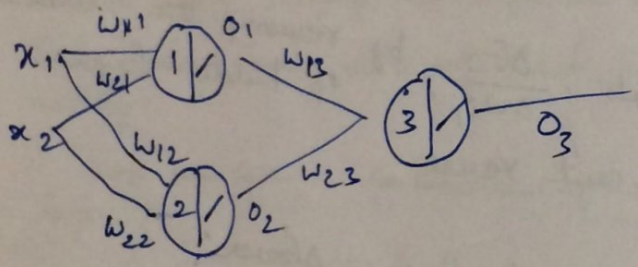




$$\frac{\partial f}{\partial z} = \frac{\partial f}{\partial h} \cdot \frac{\partial h}{\partial z}$$

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

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



Linear activation fn.

E = Error 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 pass

1 forward pass to
- learn o_1, o_2, o_3.

1 backward pass to
- learn gradients.

16/1/2017

Forward Propagation

$\frac{d}{dx}(-) = ?$

Product rule

$\frac{d}{dx}(pq) = p\frac{dq}{dx} + q\frac{dp}{dx}$

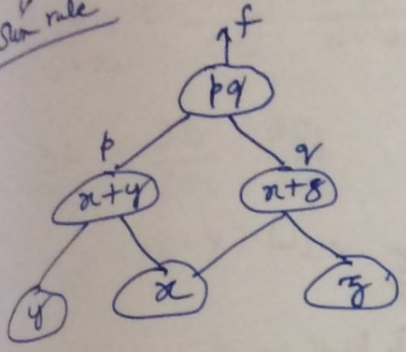
$\frac{\partial}{\partial x}(p+q) = \frac{dp}{dx} + \frac{dq}{dx}$

Sum rule

Back propagation

$\frac{df}{d(-)} = ?$

$\frac{df}{dx} = \frac{\partial f}{\partial p}\cdot\frac{\partial p}{\partial x}$ (chain rule)

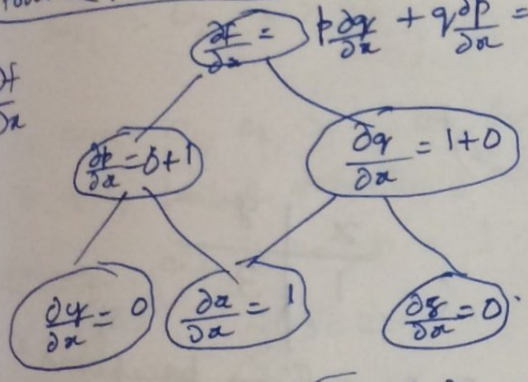


Compute $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

Forward propagation

1) $\frac{\partial f}{\partial x}$

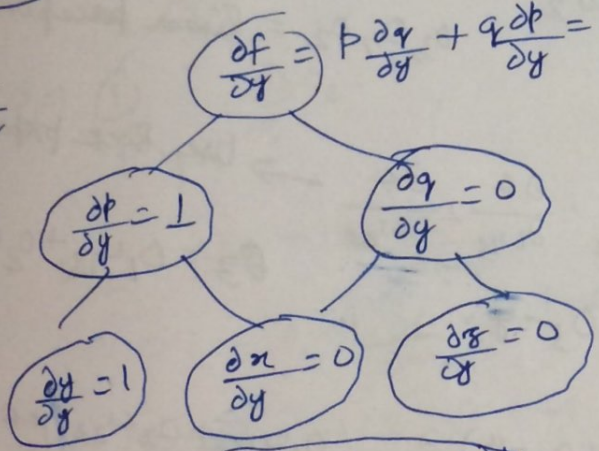
$\frac{\partial f}{\partial x} = p\frac{\partial q}{\partial x} + q\frac{\partial p}{\partial x} = (x+y)\cdot 1 + (x+z) = 1 = 2x + y + z$



Three passes needed for three variables

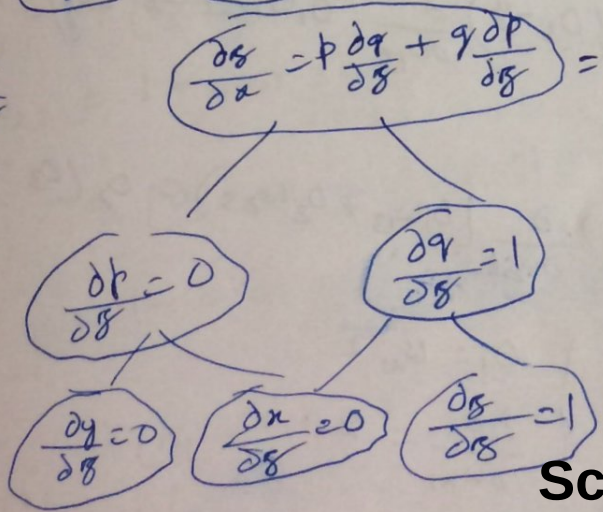
2) $\frac{\partial f}{\partial y}$

$\frac{\partial f}{\partial y} = p\frac{\partial q}{\partial y} + q\frac{\partial p}{\partial y} = p(0) + (x+z)\cdot 1 = x + z$

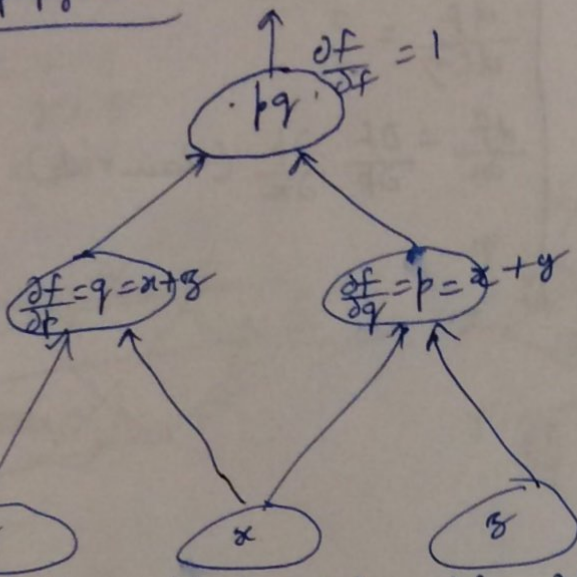


3) $\frac{\partial f}{\partial z}$

$\frac{\partial f}{\partial x} = p\frac{\partial q}{\partial z} + q\frac{\partial p}{\partial z} = (x+y)\cdot 1 + q(0) = x+y$



Back propagation



One pass enough for all three variables.

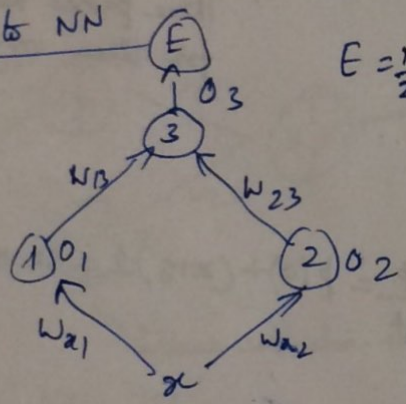
Hence, back propagation is preferred.

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

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

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

Apply to NN



$E = \frac{1}{2}(y - O_3)^2$

x	y
1	3

O_3, O_1, O_2 = linear perceptrons

Goal: Compute $\frac{\partial E}{\partial w_{x1}}, \frac{\partial E}{\partial w_{x2}}, \frac{\partial E}{\partial w_{13}}, \frac{\partial E}{\partial w_{23}}$ → Use Back propagation

$\theta_3 = O_1 w_{13} + O_2 w_{23}$

$\frac{\partial E}{\partial O_3} = \frac{2}{2}(y - O_3)(-1) = O_3 - y$

1) $\frac{\partial E}{\partial w_{13}} = \frac{\partial E}{\partial O_3}\cdot\frac{\partial O_3}{\partial w_{13}} = (O_3 - y)\frac{\partial}{\partial w_{13}}(O_1 w_{13} + O_2 w_{23}) = O_1(O_3 - y)$

($\frac{\partial E}{\partial O_3}$: error gradient; $\frac{\partial O_3}{\partial w_{13}}$: 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) $\frac{\partial E}{\partial w_{x1}} = \frac{\partial E}{\partial O_1}\cdot\frac{\partial O_1}{\partial w_{x1}}$

($\frac{\partial E}{\partial O_1}$: error gradient; $\frac{\partial O_1}{\partial w_{x1}}$: local gradient)

$O_1 = w_{x1} x$

$\frac{\partial O_1}{\partial w_{x1}} = x$

$$\frac{\partial E}{\partial o_1} = \frac{\partial E}{\partial o_3}\frac{\partial o_3}{\partial o_1}$$

$$= (o_3 - y) w_{13}$$

$$\therefore \frac{\partial E}{\partial w_{x1}} = (o_3 - y) w_{13} \cdot x$$

4) $$\frac{\partial E}{\partial w_{x2}} = \frac{\partial E}{o_2}\cdot\frac{\partial o_2}{\partial w_{x2}}$$

$o_2 = w_{x2} \cdot x$

$\frac{\partial o_2}{\partial w_{x2}} = x$

$$\frac{\partial E}{\partial o_2} = \frac{\partial E}{\partial o_3}\cdot\frac{\partial o_3}{\partial o_2}$$

$$= (o_3 - y)\cdot w_{23}$$

$$\therefore \frac{\partial E}{\partial w_{x2}} = (o_3 - y) w_{23} x$$

Learning in NN ≡ Gradient descent

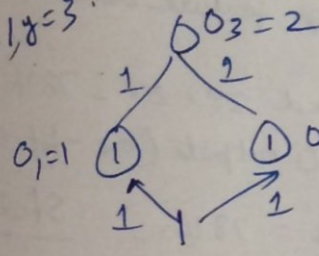
$$w_{ij} = w_{ij} - \eta \frac{\partial E}{\partial w_{ij}}$$

To know o_1, o_2, o_3, Go forward first.

To apply partial derivatives, go in back direction.

Assume, all 4 weights = 1. Ignoring bias for simplicity.

$x = 1, y = 3$. $\eta = 0.01$



Step 1:

Step 2 $= w_{13} = w_{13} - \eta \cdot \frac{\partial E}{\partial w_1}$

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

$$\frac{\partial E}{\partial w_{13}} = o_1 (o_3 - y) = 1 \cdot (2 - 3) = -1$$

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

$$\frac{\partial E}{\partial w_{23}} = o_2 (o_3 - y) = 1 (2 - 3) = -1$$

$$\frac{\partial E}{\partial w_{x1}} = 1 - 0.01 \frac{\partial E}{\partial w_{x1}} = 1.01$$

$$\frac{\partial E}{\partial w_{x1}} = (o_3 - y) w_{13} \cdot x$$

$$\frac{\partial E}{\partial w_{x1}} = (2 - 3) 1 \cdot 1 = -1$$

$$\frac{\partial E}{\partial w_{x2}} = 1 - 0.01 \frac{\partial E}{\partial w_{x2}} = 1.01$$

$$\frac{\partial E}{\partial w_{x2}} = (o_3 - y) w_{23} x$$

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

$$= -1$$

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

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

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

$$= g'(a_j) \frac{\partial \sum_k v_{kj} O_k}{\partial w_{ij}}$$

For sigmoid perceptron, the local gradients will change

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

$$= \cdot$$

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

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

17/1/17

For digit recognizer, suppose we used $28 \times 28 = 784$ inputs and 100 perceptrons in hidden layer 1 and 10 outputs (one-hot encoding)

weights to learn = $100 \times 784 = 78400$
$+ \ 10 \times 100 = 1000$
79400

Shallow

With a 'deeper' network,
Hidden layer 1 = 50 perceptrons
2 = 10 perceptrons

Shallow vs Deep

Total weight = $50 \times 784 = 39200$
$+ 10 \times 50 = 500$
$10 \times 10 = 100$
39800

Less Learning with deeper # weights in deep networks.
Also, hidden meaning to hidden layers.

Local gradients

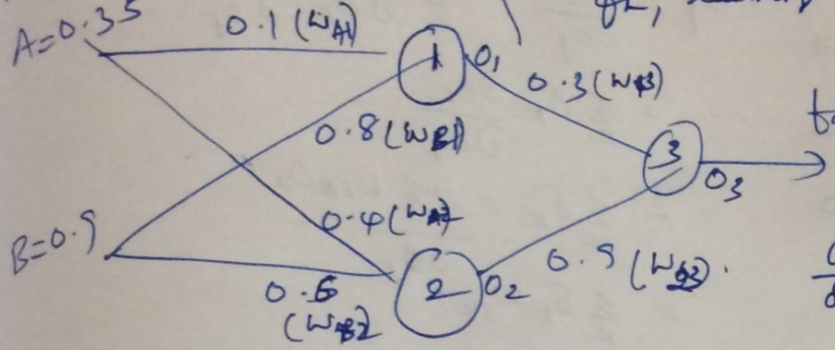
$$\frac{\partial o_j}{\partial w_{ij}} = \frac{\partial o_j}{\partial a_j} \cdot \frac{\partial a_j}{\partial w_{ij}}$$

$$= g'(a_j) \frac{\partial \sum_k w_{kj} o_k}{\partial w_{ij}}$$

$$= g'(a_j) o_i$$

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

Input



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

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

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

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

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

$\delta_3 = \frac{\partial E}{\partial o_3} = o_3 - y$

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

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

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

$a_1 = 0.1 \times 0.35 + 0.8 \times 0.9 = 0.755$

$a_2 = (0.35 \times 0.4) + (0.6 \times 0.9)$

$a_3 = w_{13} o_1 + w_{23} o_2$

$a_1 = A w_{A1} + B w_{B1}$

$a_2 = A w_{A2} + B w_{B2}$

$o_1 = \text{sigmoid}(a_1) = g(a_1)$

$o_2 = \text{sigmoid}(a_2) = g(a_2)$

$w_{A1} = w_{A1} - \eta \delta_1 g(a_1)[1 - g(a_1)] A$

$= 0.1 - 0.057 (0.68)(1 - 0.68) 0.35$

$= 0.096$

$w_{B1} = w_{B1} - \eta \delta_1 g(a_1)[1 - g(a_1)] B$

$= 0.8 - 0.057 \times 0.68 (1 - 0.68) 0.9 = 0.794$

$w_{A2} = w_{A2} - \eta \delta_2 g(a_2)[1 - g(a_2)] A$

$= 0.4 - 0.171 \times 0.66 (1 - 0.66) 0.35 = 0.387$

$w_{B2} = w_{B2} - \eta \delta_2 g(a_2)[1 - g(a_2)] B$

$= 0.6 - 0.171 (0.66)(1 - 0.66) 0.9 = 0.567$

$w_{13} = w_{13} - \eta \delta_3 g(a_3)[1 - g(a_3)] o_1$

$= 0.3 - 0.19 \times 0.68 (1 - 0.68) 0.68 = 0.223$

$w_{23} = w_{23} - \eta \delta_3 g(a_3)[1 - g(a_3)] o_2$

$= 0.9 - 0.19 \times 0.68 (1 - 0.68) 0.66$

$= 0.874$

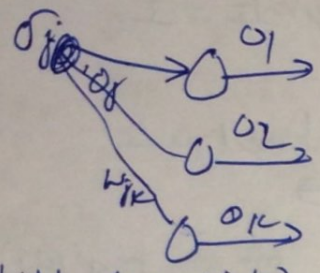
After 1st epoch / iteration,

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

$o_2 = \text{sigma}(0.387 \times 0.35 + 0.567 \times 0.9) = 0.656$

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

Error gradient [Stochastic gradient descent version]



$$\delta_j = \frac{\partial E}{\partial o_j} = \sum_K \frac{\partial E}{\partial o_K} \cdot \frac{\partial o_K}{\partial o_j}$$

$$= \sum_K \delta_K \frac{\partial o_K}{\partial o_j}$$

$$= \sum_K \delta_K * \frac{\partial \sum_i w_{iK} o_i}{\partial o_j}$$

for hidden (not output) unit j.

$$\boxed{\delta_j = \sum_K \delta_K w_{jK}}$$ Error gradient formula

$$w_{ij} = w_{ij} - \eta \frac{\partial E}{\partial w_{ij}} = \boxed{w_{ij} - \eta \delta_j g'(a_j) o_i}$$ Weight adjustment formula

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

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

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

$$\therefore, w_{ij} = w_{ij} - \eta \delta_j g(a_j)[1 - g(a_j)] o_i$$

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

18/1/2017 MNIST data - digit recognizer - Custom estimators code

19/1/2017

fit (—, steps = —, batch_size = —)

↳ optional.

steps = one iteration

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

$$w'_{ij} = w_{ij} - \eta \sum_{n=1}^{0} \delta_j^{(n)} o_i^{(n)} g'(a_j)$$

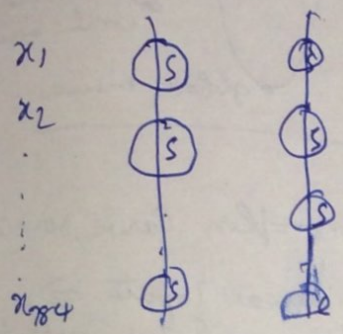
0 = batch size = 1 ⇒ SGD
= n ⇒ GD
= n-k ⇒ mini

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

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

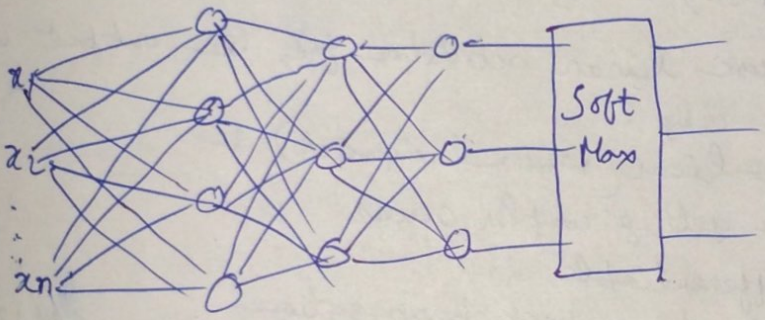
There is empirical proof that SGD (batch size = 1) will also converge eventually.

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

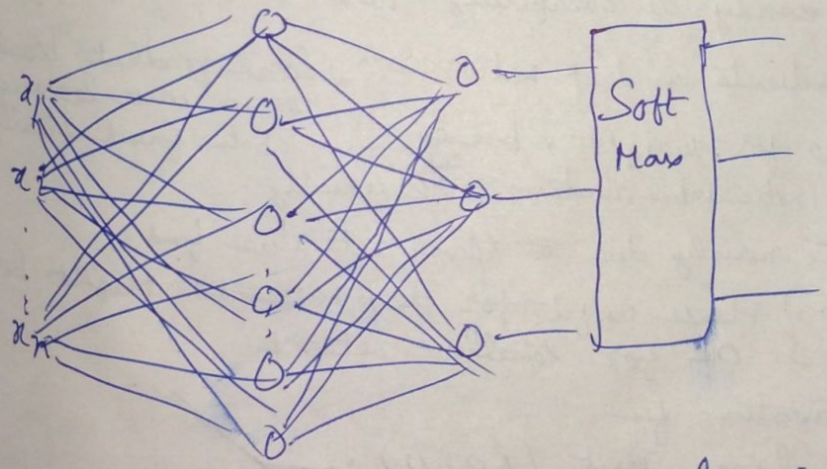


20/1/2017

Deep Network



Shallow Network



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

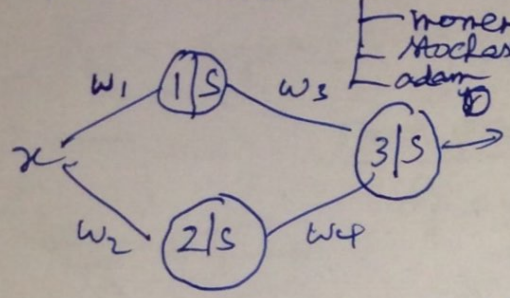
Deep layer advantages

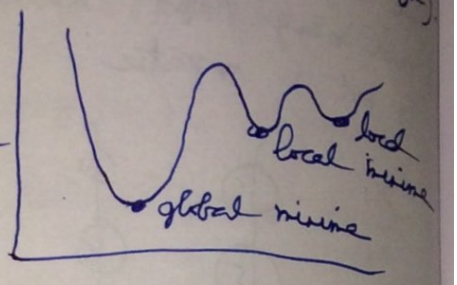
- Pattern recognition (shallow don do this)
- Noise reduction
- Detect features (biggest strength)
- Much lesser number of perceptrons

Issues with NN learning

Because the output of NN is a result of many fns (and not because of the objective fn)

1) Objective (loss) function is non-convex
   - can have multiple minima
2) - How to avoid local minima?
   - momentum
   - Stochastic batch
   - adam





$O = g\left(g(w_1 x) w_3 + g(w_2 x) w_4\right)$ → complex curve, non-convex

$f_1 = x+2,\ f_2 = x+5,\ f_3 = 3x+6.$ → linear inputs ⇒ linear output

$f = f_3(f_2(f_1(x))) = f_3(f_2(x+2)) = f_3(x+7) = 3x+27$

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

So, use a) non-linear (minimal sigmoid) fn
   - for getting complex curves
   b) differentiable fn
   - for doing back propagation

2) Due to complex curve → regularisation (L1, L2, L1+L2) → dropout → data augmentation
   - NN can easily do overfitting. How to handle it?

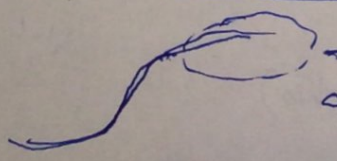
3) Vanishing gradients in deep networks → local gradients becoming 0 so no back learning. Soln found finally

4) Initial values for weights & biases
   - Xavier initialisation strategy
   - '0' usually.

Vanishing gradient mainly due to these activation fns
   - 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



→ slope becomes '0' as value increases

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

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

AI
- Vision for machines
- Language understanding
- Machine translation
- Voice recognition

Single layer

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

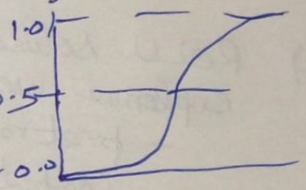
Multiple layers

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 fn

1) Outputs are not zero-centered (but 0.5 centered).
- If inputs are always +ve, then the gradients of 'w' are always all +ve or all -ve.



→ 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.

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial O_j}\cdot\frac{\partial O_j}{\partial w_{ij}}$$
$$= \delta_j g'(a_j) O_i$$

This decides +ve or -ve of the entire product

O_i always +ve ∵ sigmoid output

evidence = $a = \sum w_i x_i$

with 'tanh' $g'(a_j)$ is (-1, +1)
- So, "$\delta_j g'(a_j)$" decides overall +ve & -ve.

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

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

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

Saturated ⇒ sigmoid output = 0, 1.

$$\frac{\partial E}{\partial w_{ij}} = \delta_j g'(a_j) O_i$$
$$= \delta_j g(a_j)[1-g(a_j)] O_i$$
$$= \delta_j O_j (1-O_j) O_i$$

($O_j (1-O_j)$ = $g'(a_j)$)

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

So, no more learning.

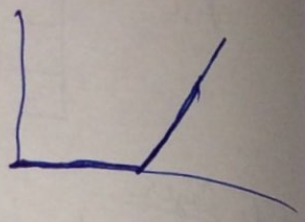
Improper weight initialisation ... saturated sigmoid/tanh

Soln to saturation — Use ReLU or maxout in deep

(*) Sigmoid good only for single layer neural network task

Rectified linear unit (ReLU)

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

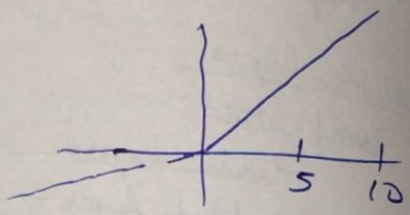


$g(x) = \max(0, x)$

Leaky ReLU

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

- Does not saturate (in both +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 weights — 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 taken [0,1]~~ If weights are initialized in [0,1] range, small/large weights lead to quick saturation
- use
  a) Unsupervised 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' ?
- Its 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 objective fn

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

Soln as per heuristics (but not all)
- 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

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