

Table of Contents

1. [Preface](#) 1.1
2. [Introduction](#) 1.2
3. [Machine Learning](#) 1.3
 1. [Linear Algebra](#) 1.3.1
 2. [Supervised Learning](#) 1.3.2
 1. [Neural Networks](#) 1.3.2.1
 2. [Linear Classification](#) 1.3.2.2
 3. [Loss Function](#) 1.3.2.3
 4. [Model Optimization](#) 1.3.2.4
 5. [Backpropagation](#) 1.3.2.5
 6. [Feature Scaling](#) 1.3.2.6
 7. [Model Initialization](#) 1.3.2.7
 8. [Recurrent Neural Networks](#) 1.3.2.8
 3. [Deep Learning](#) 1.3.3
 1. [Convolution](#) 1.3.3.1
 2. [Convolutional Neural Networks](#) 1.3.3.2
 3. [Fully Connected Layer](#) 1.3.3.3
 4. [Relu Layer](#) 1.3.3.4
 5. [Dropout Layer](#) 1.3.3.5
 6. [Convolution Layer](#) 1.3.3.6
 1. [Making faster](#) 1.3.3.6.1
 7. [Pooling Layer](#) 1.3.3.7
 8. [Batch Norm layer](#) 1.3.3.8
 9. [Model Solver](#) 1.3.3.9
 10. [Object Localization and Detection](#) 1.3.3.10
 11. [Single Shot Detectors](#) 1.3.3.11
 12. [Image Segmentation](#) 1.3.3.12
 13. [GoogleNet](#) 1.3.3.13
 14. [Residual Net](#) 1.3.3.14
 15. [Deep Learning Libraries](#) 1.3.3.15
 4. [Unsupervised Learning](#) 1.3.4
 1. [Principal Component Analysis](#) 1.3.4.1
 2. [Generative Models](#) 1.3.4.2
 5. [Distributed Learning](#) 1.3.5
 6. [Methodology for usage](#) 1.3.6
4. [Artificial Intelligence](#) 1.4
 1. [OpenAI Gym](#) 1.4.1
 2. [Tree Search](#) 1.4.2
 3. [Markov Decision process](#) 1.4.3
 4. [Reinforcement Learning](#) 1.4.4
 1. [Q Learning Simple](#) 1.4.4.1
 2. [Deep Q Learning](#) 1.4.4.2
 3. [Deep Reinforcement Learning](#) 1.4.4.3
5. [Appendix](#) 1.5
 1. [Lua and Torch](#) 1.5.1
 2. [Tensorflow](#) 1.5.2
 1. [Multi Layer Perceptron MNIST](#) 1.5.2.1
 2. [Convolution Neural Network MNIST](#) 1.5.2.2
 3. [SkFlow](#) 1.5.2.3

Preface

Documentation on all topics that I learn on both Artificial intelligence and machine learning.

Topics Covered:

- Artificial Intelligence Concepts
- Search
- Decision Theory
- Reinforcement Learning
- Artificial Neural Networks
- Back-propagation
- Feature Extraction
- Deep Learning
- Convolutional Neural Networks
- Deep Reinforcement Learning
- Distributed Learning
- Python/Matlab deep learning library

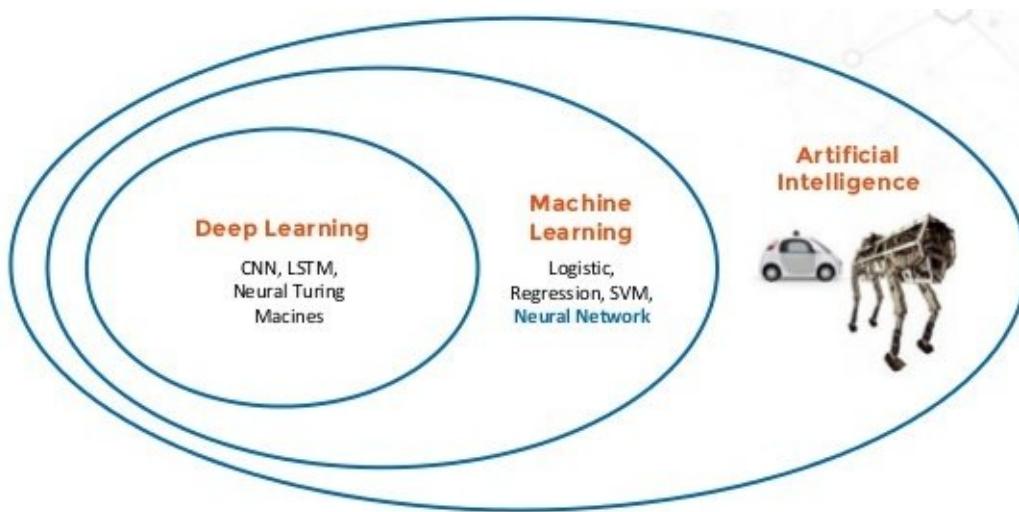
Introduction

What is this book about

This book will cover the basics needed to implement and understand your own Artificial Intelligence and Machine Learning library. All formulas and concepts will be presented with code in both Matlab and Python.

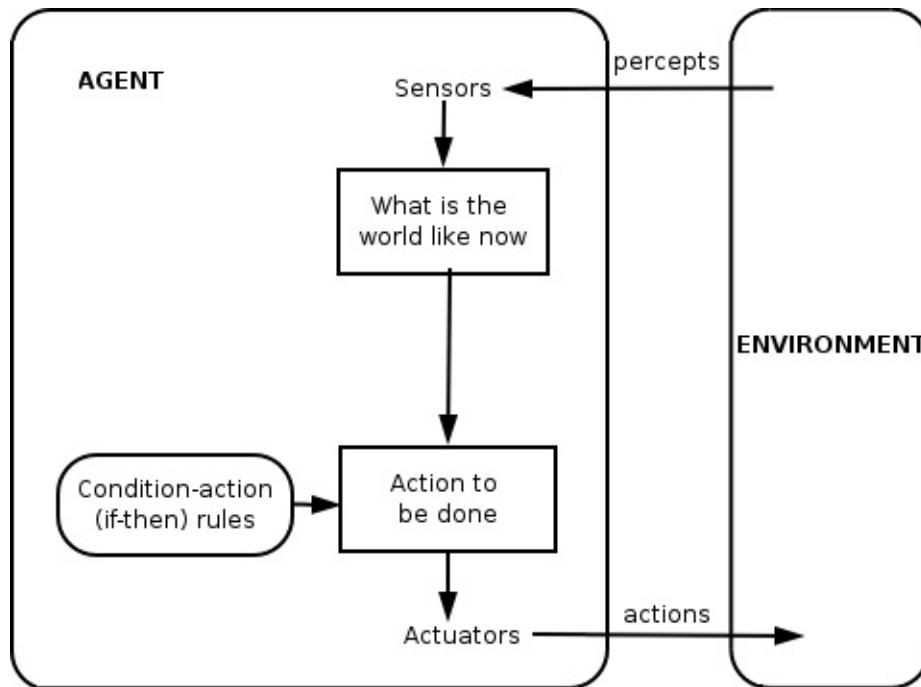
Artificial Intelligence and Deep Learning

Today we have a lot of confusion around artificial intelligence, machine learning, and deep learning. Actually those terms are just a subset of the Artificial Intelligence.



What is Artificial Intelligence

Field of study which studies how to create computing systems that are capable of intelligent behavior. Some other texts define it as the study/design of intelligent agents. Here agent is a system(Software/Hardware) that perceives its environment and takes actions that maximize its chances of success.



Intelligence definition

For the scope of this book, an intelligent agent is an agent that solve a problem optimally, which means that the system will figure out alone what is the best course of action to take.

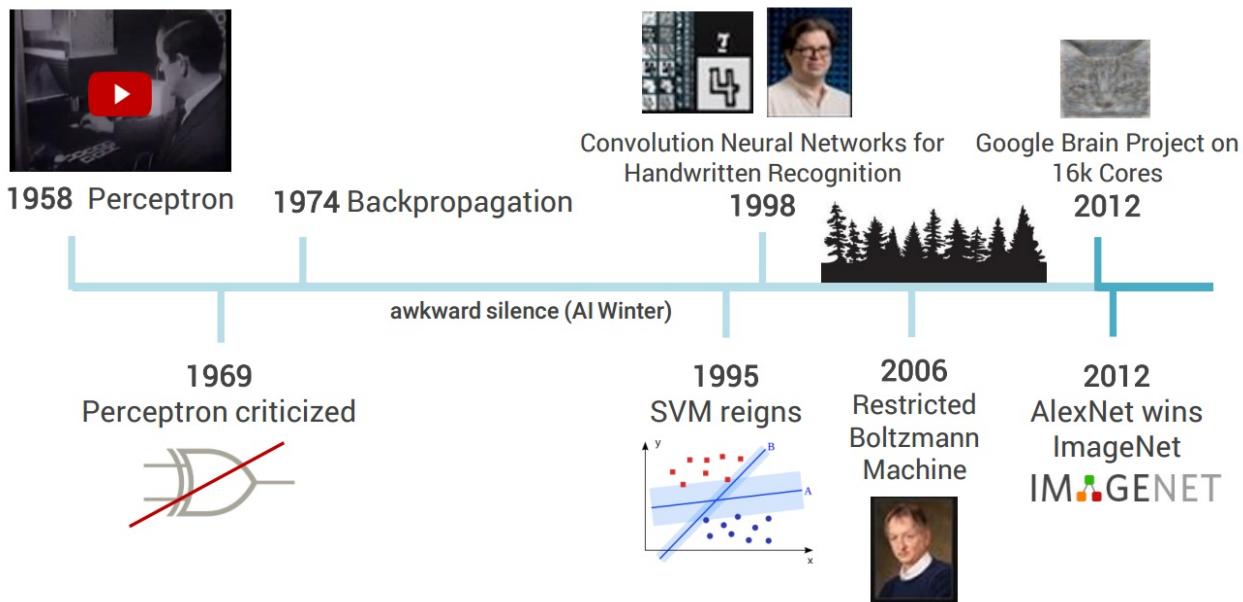
Ways for taking a intelligent decision

- Keep track of all your actions and check if they were good or bad, then compare a new action with one of them.
- Before take an action, simulate all the possible outcomes (was a good or bad action) then choose the less bad. So you need an abstraction (model) of the world, just remember that a model of the world is not the world.

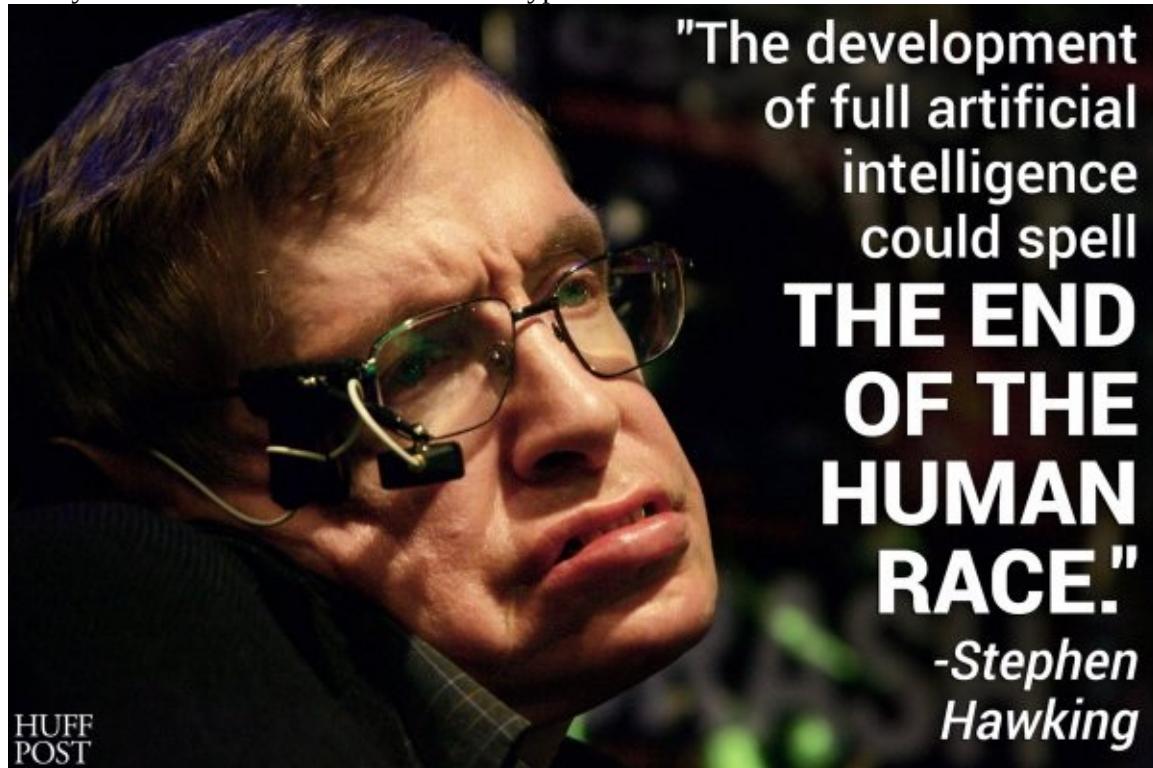
A funny fact about artificial intelligence is that after a problem is fully solved it's not called intelligent anymore... (ie: Make a computer play chess was the highest display of intelligence, now people don't consider that anymore)

History

Basically through the history of artificial intelligence we had some periods of surprising/hope and disappointment.



Funny fact is that now we're on a mix of hype/fear



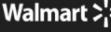
Why Now

The machine learning algorithms (Even the deep ones) are there for decades so why we have now this buzzword?

Basically due to the advance of computing power through (GPUs, multi-core CPU systems, and FPGAs) and the availability of data (Big data) through internet.

Also the amount of data that need to be classified nowadays become to big to be handled manually, so big companies Google, Microsoft, Facebook, start to invest heavily on the subject.

Three Driving Factors...

Big Data Availability	New ML Techniques	Compute Density	
facebook  350 millions images uploaded per day			
Walmart  2.5 Petabytes of customer data hourly	Deep Neural Networks	GPUs	
YouTube  100 hours of video uploaded every minute			
ML systems extract value from Big Data			
Batch Size	Training Time CPU	Training Time GPU	GPU Speed Up
64 images	64 s	7.5 s	8.5X
128 images	124 s	14.5 s	8.5X
256 images	257 s	28.5 s	9.0X

The new Hype

The last years 2013/2016 artificial intelligence (Machine learning) is surprising people with results closer or sometimes better than humans. For example:

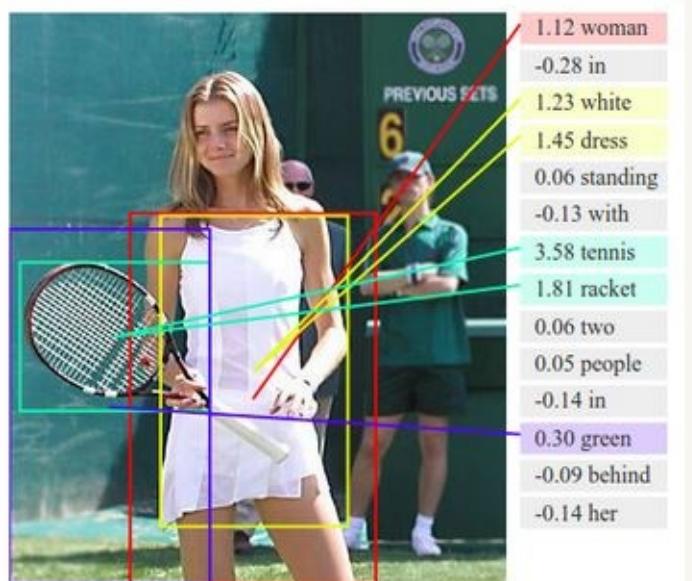
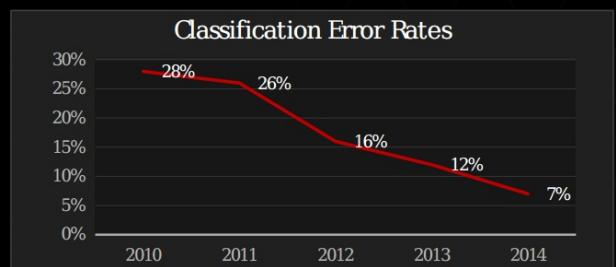
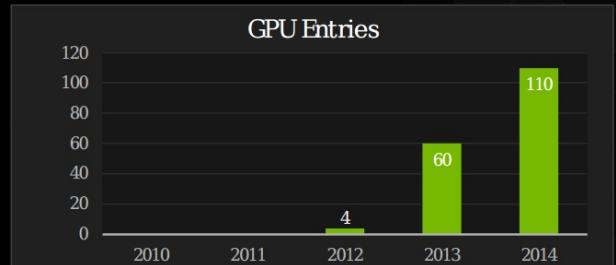
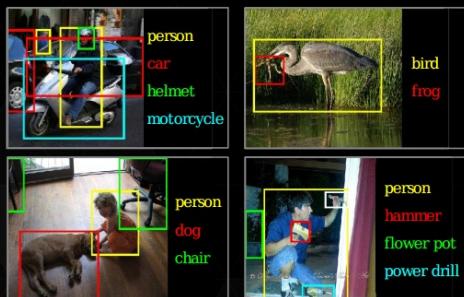
- Speech and natural language processing
- Face Recognition
- Image Classification, object detection
- Car Driving
- Playing complex games (Alpha Go)
- Control strategies (Control engineering)

Image Recognition Challenge

1.2M training images • 1000 object categories

Hosted by

IMAGENET





So basically people start to become afraid of loosing their jobs and some artificial intelligence server taking over the world.

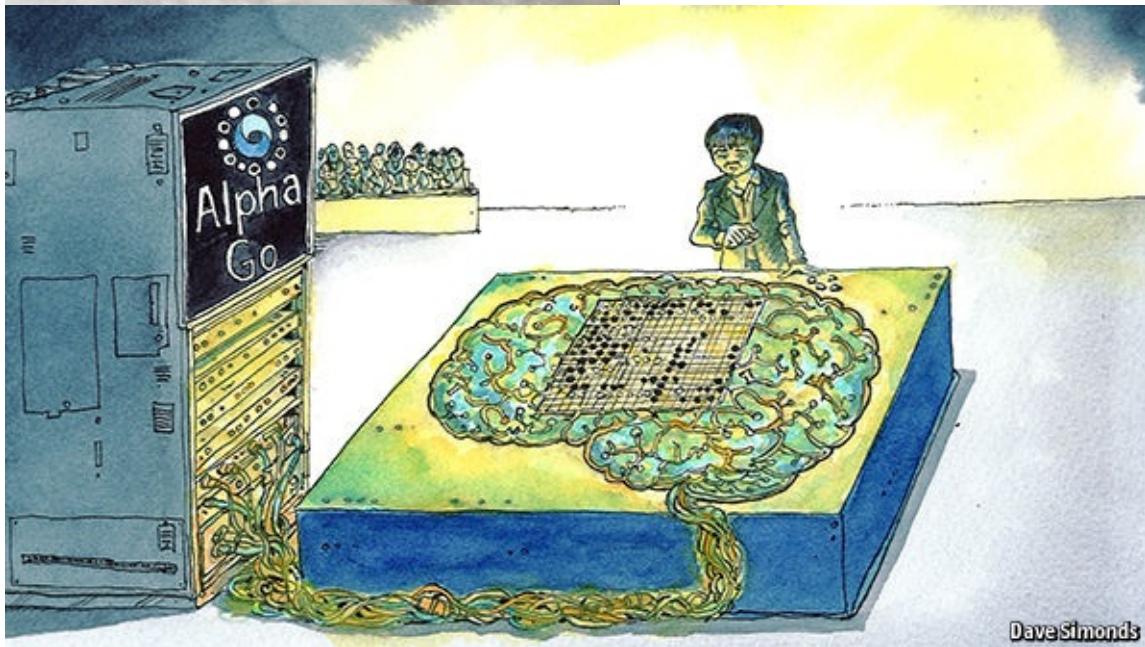
Computing power Comparison

On the table bellow we present you a table with the amount of possible operations per second and cost of some hardware platforms

Type	Name	Flops	Cost
Mobile	Raspberry Pi 1 st Gen, 700 Mhz	0,04 Gflops	\$35
Mobile	Apple A8	1,4 Gflops	\$700 (in iPhone 6)
CPU	Intel Core i7-4930K (Ivy Bridge), 3.7 GHz	140 Gflops	\$700
CPU	Intel Core i7-5960X (Haswell), 3.0 GHz	350 Gflops	\$1300
GPU	NVidia GTX 980	4612 Gflops (single precision), 144 Gflops (double precision)	\$600 + cost of PC (~\$1000)
GPU	NVidia Tesla K80	8740 Gflops (single precision), 2910 Gflops (double precision)	\$4500 + cost of PC (~1500)

Deepmind hardware

Just to illustrate, the picture bellow is the hardware used to play against one of the best Go players in the world.



Machine Learning

Introduction

Machine learning is all about using your computer to "learn" how to deal with problems without "programming". (It's a branch of artificial intelligence)

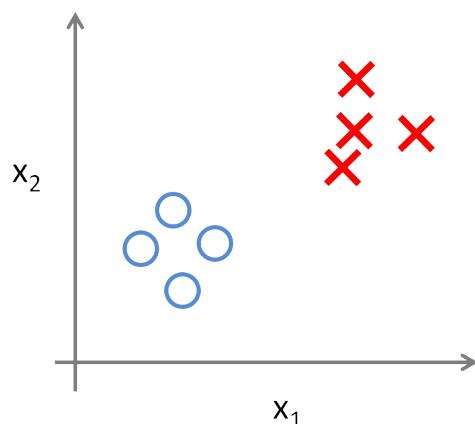
We take some data, train a model on that data, and use the trained model to make predictions on new data. Basically is a way to make the computer create a program that gives some output with a known input and that latter give a intelligent output to a different but similar input.

We need machine learning on cases that would be difficult to program by hand all possible variants of a classification/prediction problem

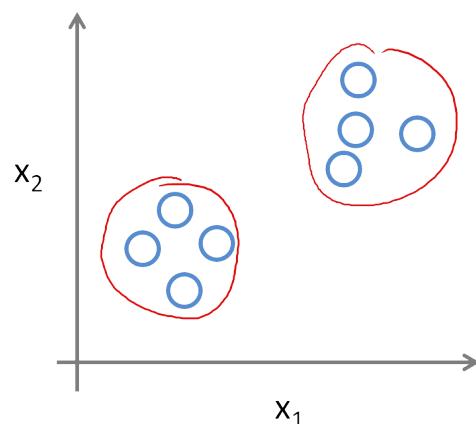
The basic Idea of Machine Learning is to make the computer learn something from the data. Machine learning comes in two flavors:

- Supervised Learning: You give to the computer some pairs of inputs/outputs, so in the future new when new inputs are presented you have an intelligent output.
- Unsupervised Learning: You let the computer learn from the data itself without showing what is the expected output.

Supervised Learning

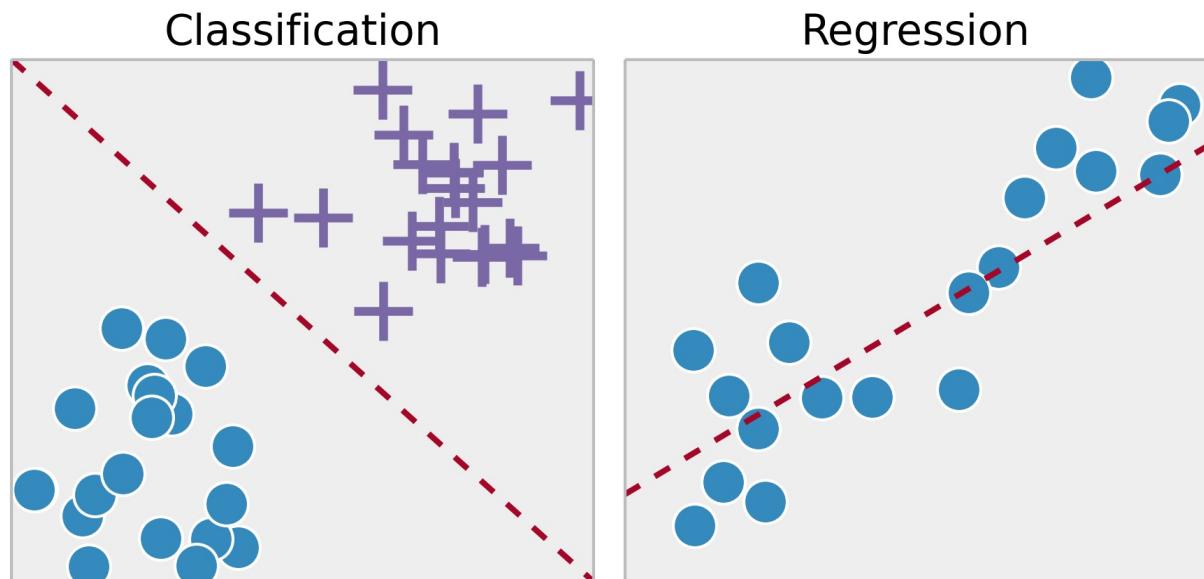


Unsupervised Learning



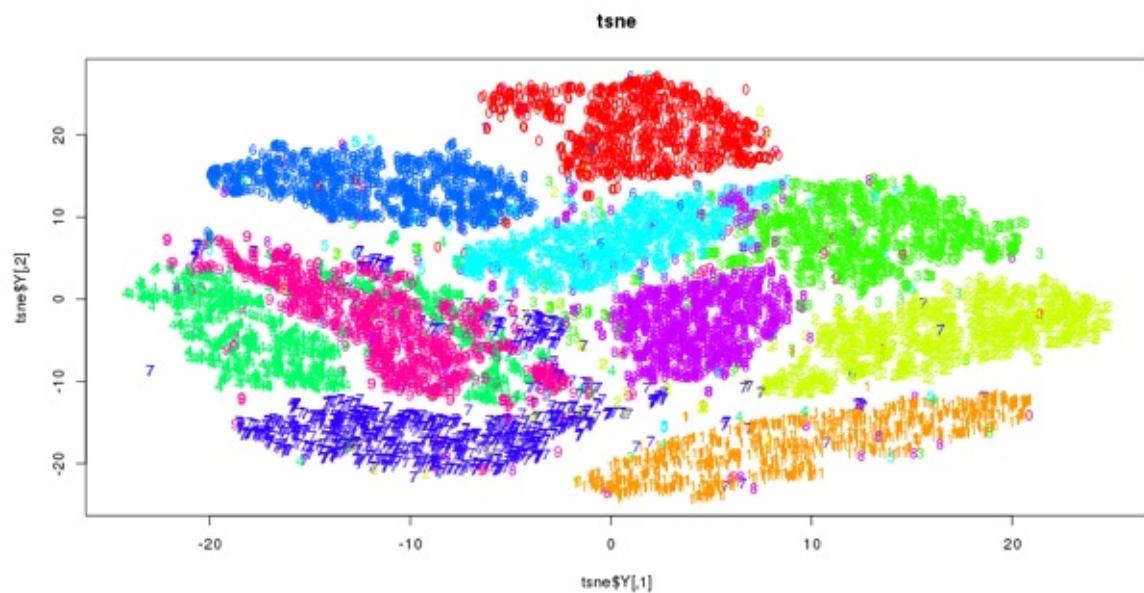
Examples of Supervised Learning

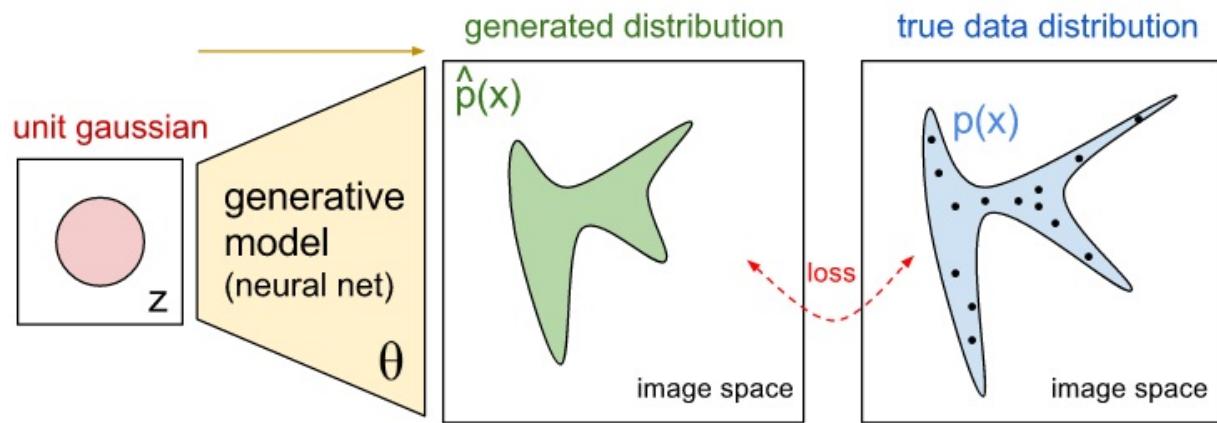
1. Image Classification: You train with images/labels. Then on the future when you give a new image expecting that the computer will recognise the new object (Classification)
2. Market Prediction: You train the computer with historical market data, and ask the computer to predict the new price on the future (Regression)



Examples of Unsupervised Learning

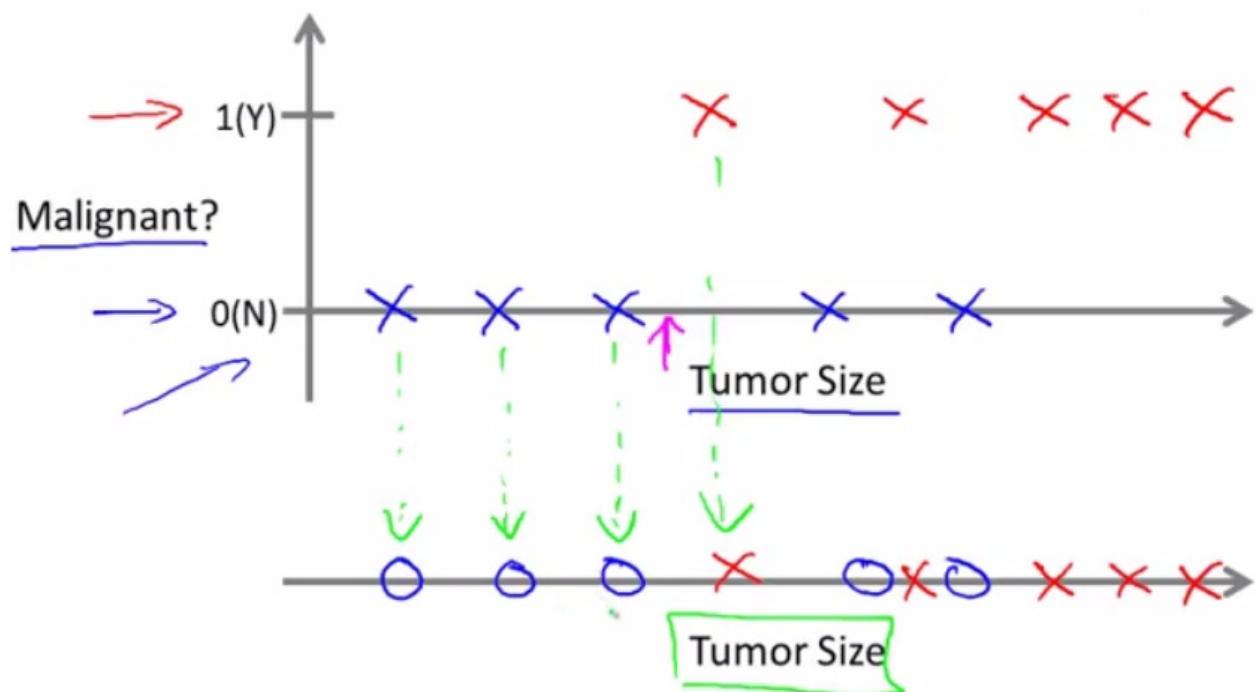
1. Clustering: You ask the computer to separate similar data on clusters, this is essential in research and science.
2. High Dimension Visualisation: Use the computer to help us visualise high dimension data.
3. Generative Models: After a model capture the probability distribution of your input data, it will be able to generate more data. This is very useful to make your classifier more robust.



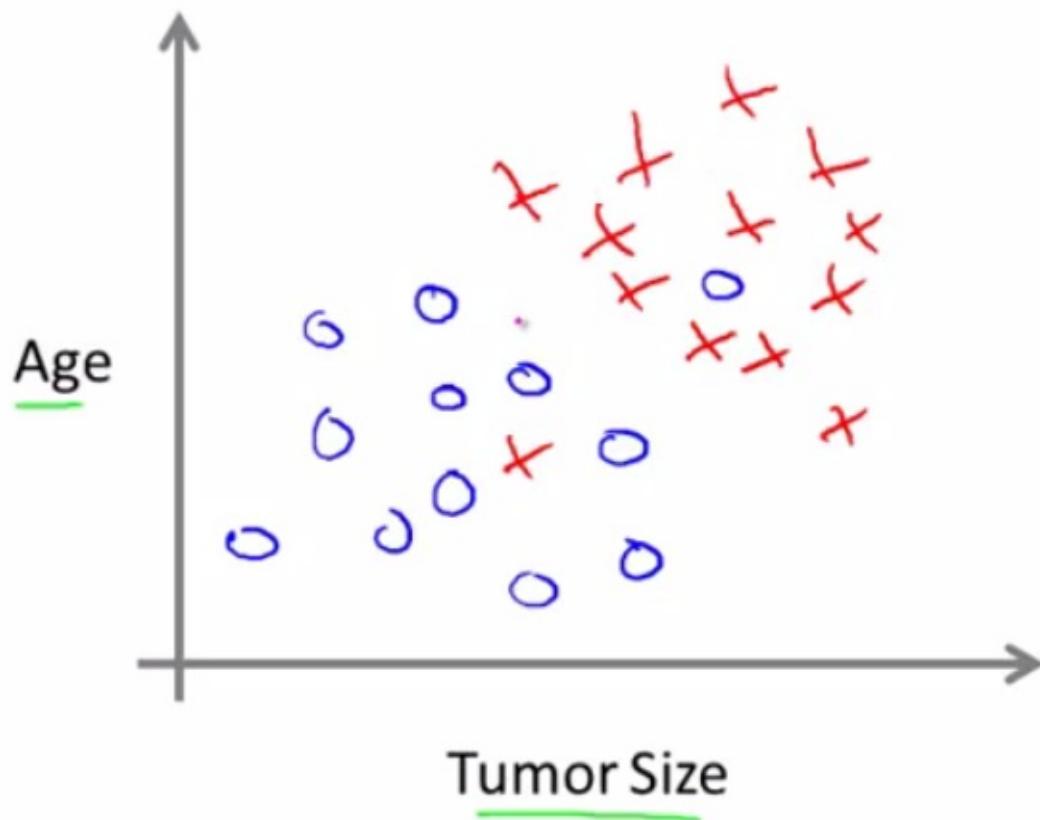


Features

Imagine the following problem, you are working on a system that should classify if a tumour is benign or malignant, at a first moment the only information that you have to decide is the tumour size. We can see the your training data distribution bellow. Observe that the characteristic (or feature) tumour size does not seems to be alone a good indicator to decide if the tumour is malignant or benign.



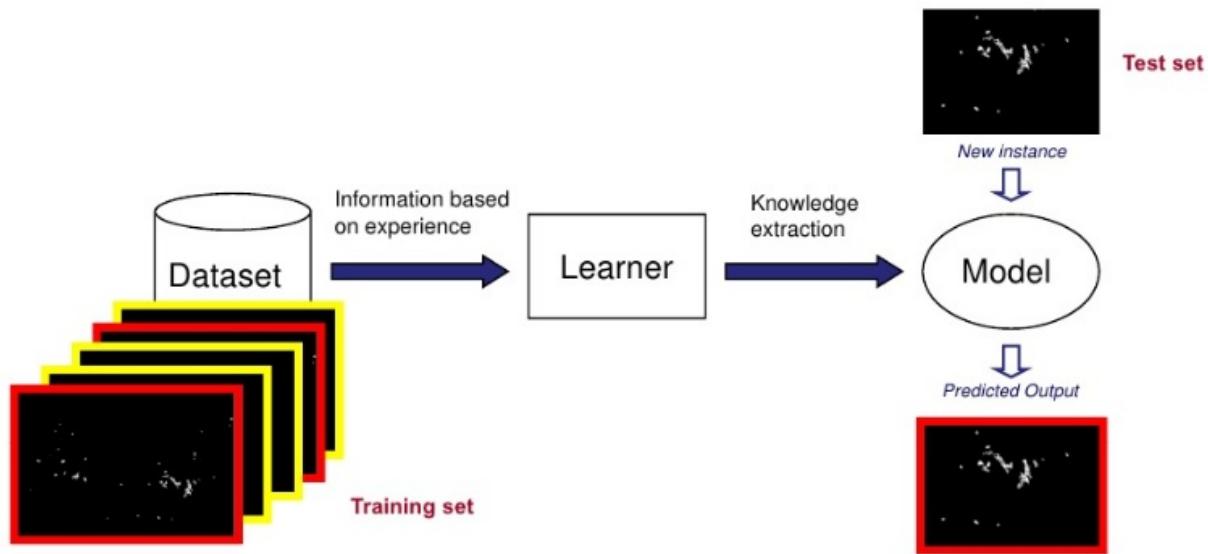
Now consider that we add one more feature to the problem (Age).



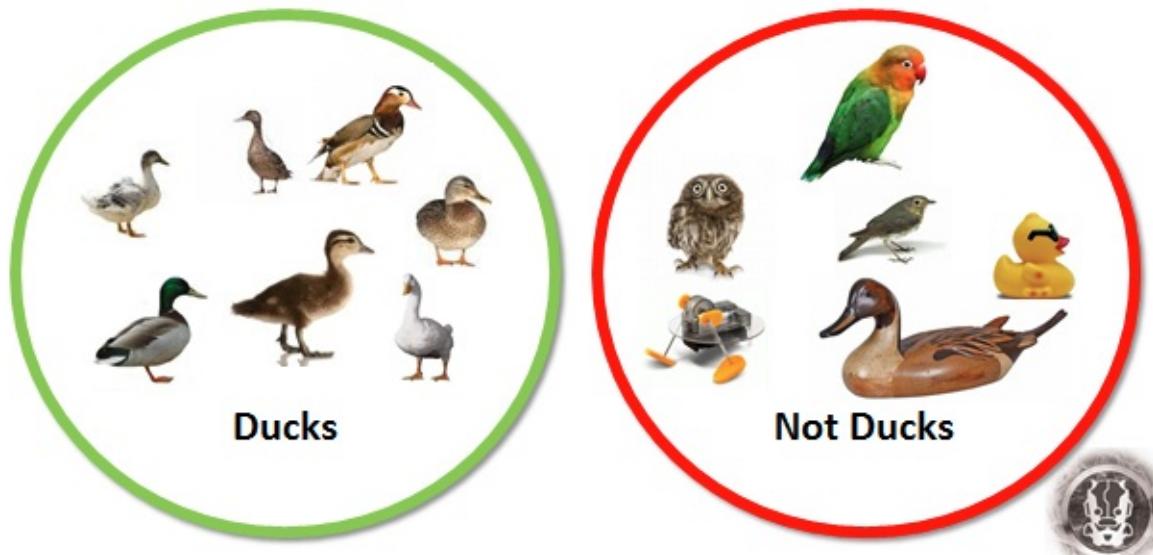
The intuition is that with more features that are relevant to the problem that you want to classify, you will make your system more robust. Complex systems like this one could have up to thousand of features. One question that you may ask is how can I determine the features that are relevant to my problem. Also which algorithm to use best to tackle infinite amount of features, for example Support Vector machines have mathematical tricks that allow a very large number of features.

Training

The idea is to give a set of inputs and it's expected outputs, so after training we will have a model (hypothesis) that will then map new data to one of the categories trained.

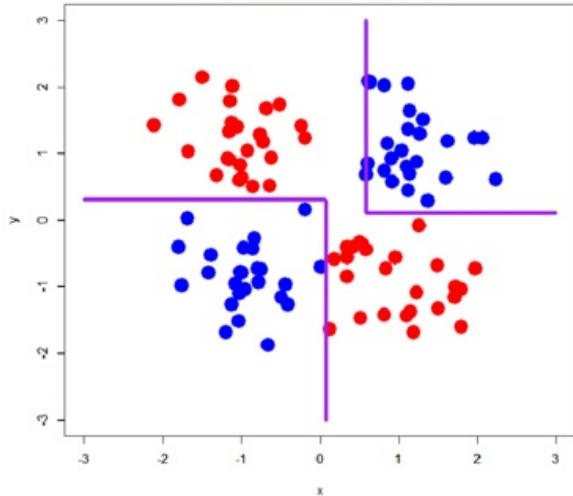
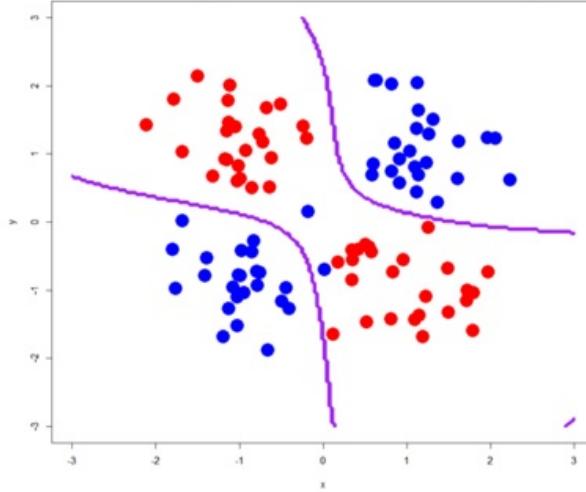
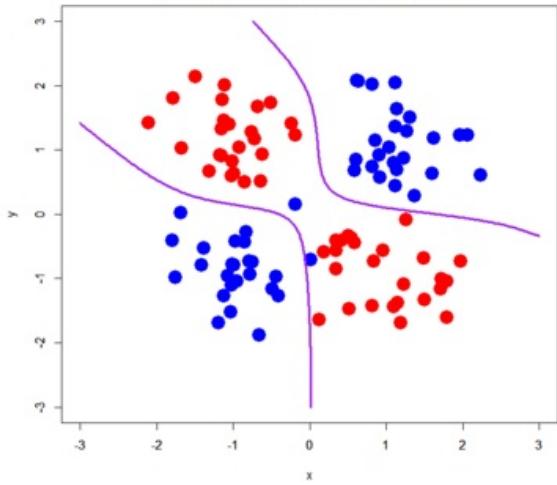
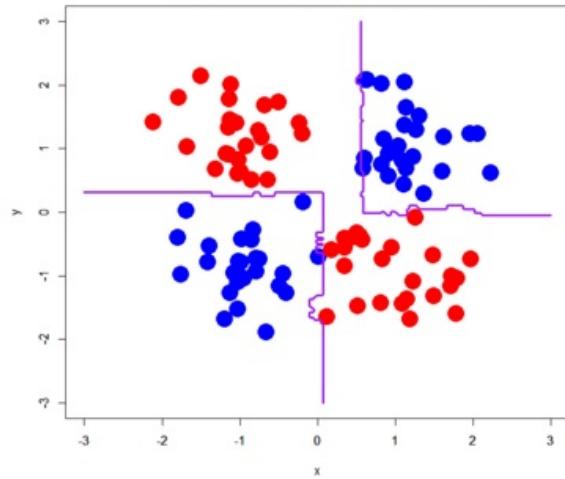


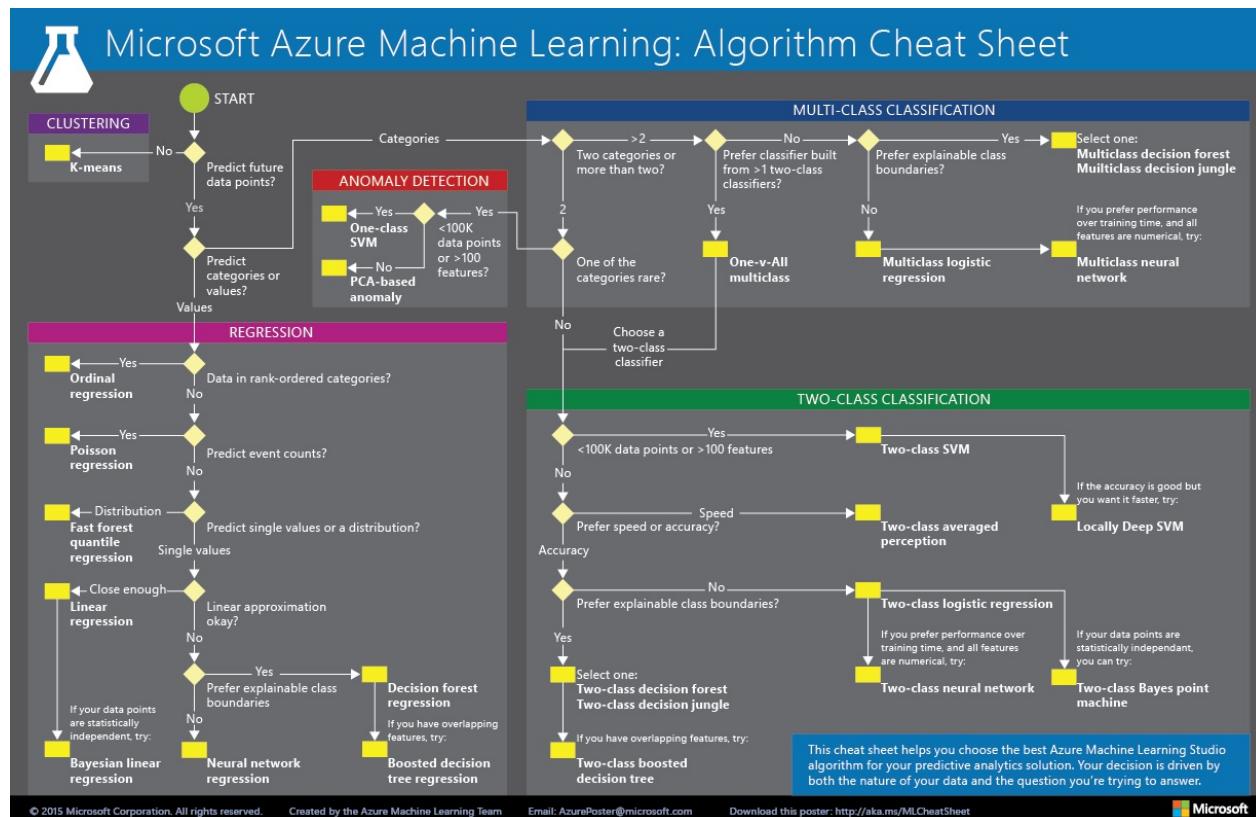
Ex: Imagine that you give a set of images and the following categories, duck or not duck, the idea is that after training you can get an image of a duck from internet and the model should say "duck"



Bag of tricks

There are a lot of different machine learning algorithms, on this book we will concentrate more on neural networks, but there is no one single best algorithm it all depends on the problem that you need to solve, the amount of data available.

Decision Tree**SVM (Gaussian kernel)****Neural Network****Random Forest**

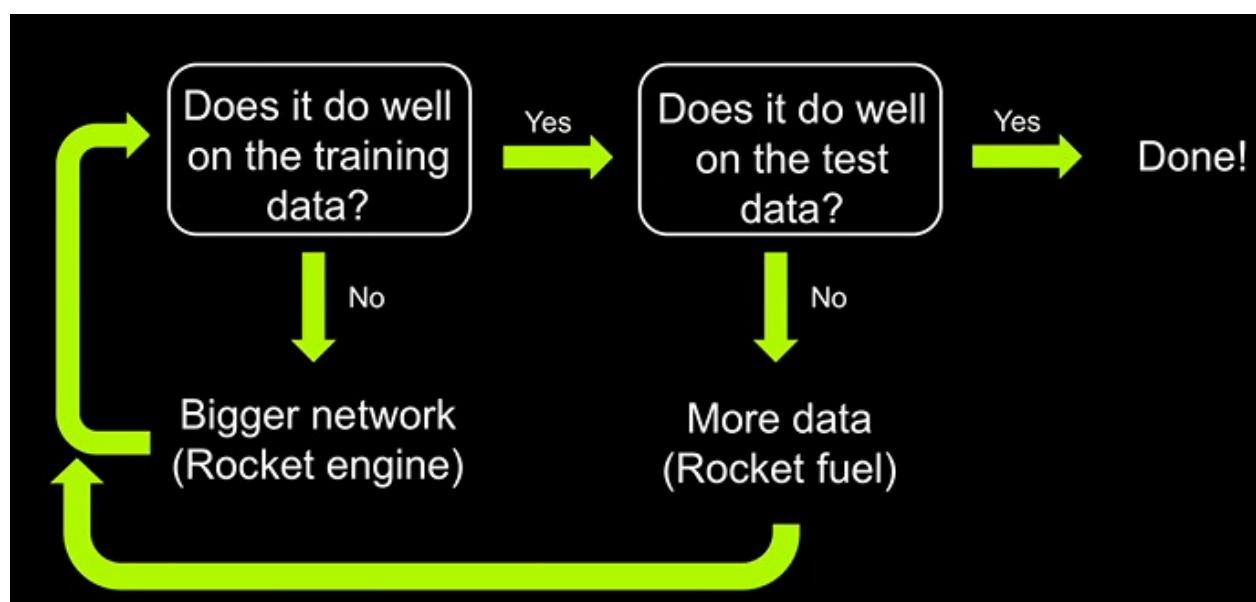


The Basic Recipe

This is the super simple recipe (maybe cover 50%), we will explain the “how” later but this gives some hint on how to think when dealing with a machine learning problem.

- First check if your model works well on the training data, and if not make the model more complex (Deeper, or more neurons)
- If yes then test on the “test” data, if not you overfit, and the most reliable way to cure overfit is to get more data (Putting the test data on the train data does not count)

By the way the biggest public image dataset (imagenet) is not big enough to the 1000 classes imangenet competition



Next Chapter

On the next chapter we will learn the basics of Linear Algebra needed on artificial intelligence.

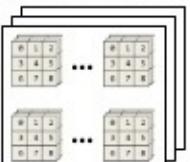
Linear Algebra

Introduction

Linear Algebra is a omit important topic to understand, a lot of deep learning algorithms use it, so this chapter will teach the topics needed to understand what will come next.

Scalars, Vectors and Matrices

- Scalars: A single number
- Vector: Array of numbers, where each element is identified by an single index
- Matrix: Is a 2D array of numbers, bellow we have a (2-row)X(3-col) matrix. On matrix a single element is identified by two indexes instead of one

Scalar	Vector	Matrix									
24	$\begin{bmatrix} 2 & -8 & 7 \end{bmatrix}$ <small>row</small> or <small>column</small> $\begin{bmatrix} -6 \\ -4 \\ 27 \end{bmatrix}$	$\begin{bmatrix} 6 & 4 & 24 \\ 1 & -9 & 8 \end{bmatrix}$ <small>row(s) × column(s)</small>									
Dimensions	Example	Terminology									
1	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>0</td><td>1</td><td>2</td></tr> </table>	0	1	2	Vector						
0	1	2									
2	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>0</td><td>1</td><td>2</td></tr> <tr><td>3</td><td>4</td><td>5</td></tr> <tr><td>6</td><td>7</td><td>8</td></tr> </table>	0	1	2	3	4	5	6	7	8	Matrix
0	1	2									
3	4	5									
6	7	8									
3	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>0</td><td>1</td><td>2</td></tr> <tr><td>3</td><td>4</td><td>5</td></tr> <tr><td>6</td><td>7</td><td>8</td></tr> </table>	0	1	2	3	4	5	6	7	8	3D Array (3 rd order Tensor)
0	1	2									
3	4	5									
6	7	8									
N		ND Array									

Here we show how to create them on matlab and python(numpy)

Command Window

```
>> a = 24;
>> b = [2 -8 7]

b =
2     -8      7

>> c = [-6;-4;27]

c =
-6
-4
27

>> D = [6 4 24; 1 -9 8]

D =
6     4     24
1    -9      8

>> D(2,3)

ans =
8

>> b(3)

ans =
7

>> size(D)

ans =
2     3
```

Workspace

Name	Value	Min	Max
a	24	24	24
ans	[2,3]	2	3
b	[2,-8,7]	-8	7
c	[-6;-4;27]	-6	27
D	[6,4,24;1,-9,8]	-9	24

MATLAB Coder: C/C++ S

```

leo@monsterpc2:~$ ipython
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
Type "copyright", "credits" or "license" for more information.

IPython 2.3.0 -- An enhanced Interactive Python.
?           -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help        -> Python's own help system.
object?    -> Details about 'object', use 'object??' for extra details.

In [1]: import numpy as np

In [2]: a = 24

In [3]: b = np.array([[2, -8, 7]])

In [4]: c = np.array([-6, -4, 27])

In [5]: D = np.array([6, 4, 24], [1, -9, 8])

In [6]: print(b)
[[ 2 -8  7]]

In [7]: print(c)
[-6
 -4
 27]

In [8]: print(D)
[[ 6  4 24]
 [ 1 -9  8]]

In [9]: D[1,2]
Out[9]: 8

In [10]: D.shape
Out[10]: (2, 3)

```

Matrix Operations

Here we will show the important operations.

Transpose

If you have an image(2d matrix) and multiply with a rotation matrix, you will have a rotated image. Now if you multiply this rotated image with the transpose of the rotation matrix, the image will be "un-rotated"

Basically transpose a matrix is to swap it's rows and cols. Or rotate the matrix around it's main diagonal.

**A**

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 \end{bmatrix}^T = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

$$\cdot \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$\cdot \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

Addition/Subtraction

Basically we add 2 matrices by adding each element with the other. Both matrices need to have same dimension

$$\begin{bmatrix} 3 & 8 \\ 4 & 6 \end{bmatrix} + \begin{bmatrix} 4 & 0 \\ 1 & -9 \end{bmatrix} = \begin{bmatrix} 7 & 8 \\ 5 & -3 \end{bmatrix}$$

$$\begin{bmatrix} 3 & 8 \\ 4 & 6 \end{bmatrix} - \begin{bmatrix} 4 & 0 \\ 1 & -9 \end{bmatrix} = \begin{bmatrix} -1 & 8 \\ 3 & 15 \end{bmatrix}$$

$3+4=7$
 $3-4=-1$

Multiply by scalar

Multiply all elements of the matrix by a scalar

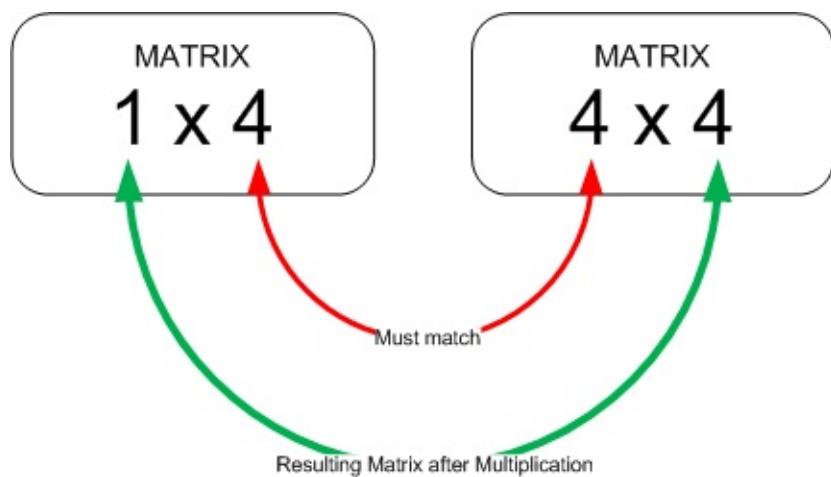
$$2 \times \begin{bmatrix} 4 & 0 \\ 1 & -9 \end{bmatrix} = \begin{bmatrix} 8 & 0 \\ 2 & -18 \end{bmatrix}$$

2x4=8

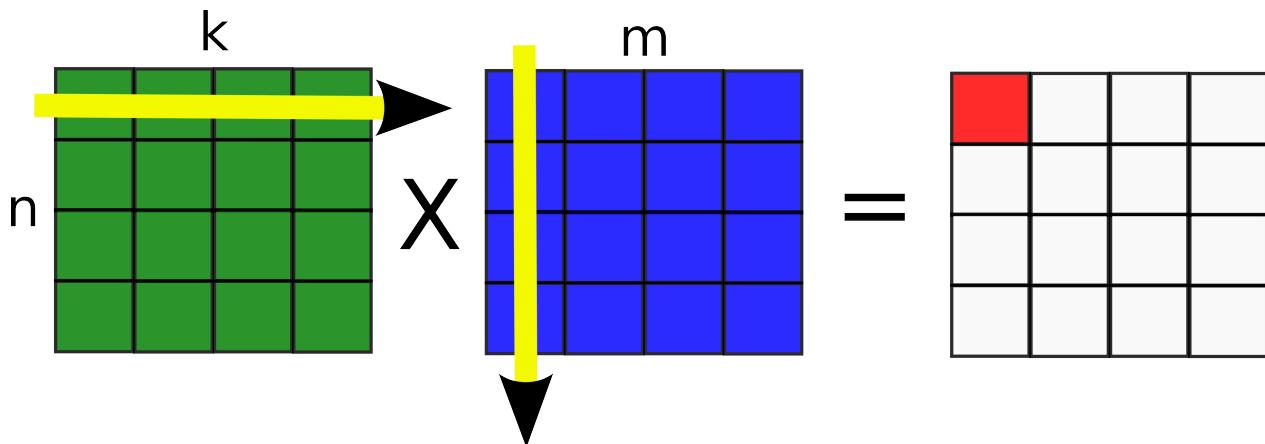
Matrix Multiplication

The matrix product of an $n \times m$ matrix with an $m \times l$ matrix is an $n \times l$ matrix. The (i,j) entry of the matrix product AB is the dot product of the i th row of A with the j th column of B .

The number of columns on the first matrix must match the number of rows on the second matrix. The result will be another matrix or a scalar with dimensions defined by the rows of the first matrix and columns of the second matrix.



Basically the operation is to "dot product" each row of the first matrix(k) with each column of the second matrix(m).



Some examples

How to multiply 2 matrices?

$$\begin{bmatrix} 1 & 2 \\ 2 & 4 \\ 3 & 5 \end{bmatrix} \times \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix} \quad 1 \times 1 + 2 \times 2 = 1 + 4 = 5$$

$$\begin{bmatrix} 1 & 2 \\ 2 & 4 \\ 3 & 5 \end{bmatrix} \times \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix} \quad 1 \times 2 + 2 \times 4 = 2 + 8 = 10$$

$$\begin{bmatrix} 1 & 2 \\ 2 & 4 \\ 3 & 5 \end{bmatrix} \times \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix} \quad 2 \times 1 + 4 \times 2 = 2 + 8 = 10$$

& so on

$$\begin{bmatrix} \$3 & \$4 & \$2 \end{bmatrix} \times \begin{bmatrix} 13 & 9 & 7 & 15 \\ 8 & 7 & 4 & 6 \\ 6 & 4 & 0 & 3 \end{bmatrix} = \begin{bmatrix} \$83 & \$63 & \$37 & \$75 \end{bmatrix}$$

$\$3 \times 13 + \$4 \times 8 + \$2 \times 6$

"Dot Product"

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & 64 \end{bmatrix}$$

Commutative property

Matrix multiplication is not always commutative $A \cdot B \neq B \cdot A$, but the dot product between 2 vectors is commutative, $x^T \cdot y = y^T \cdot x$.

Types of Matrix

There are some special matrices that are interesting to know.

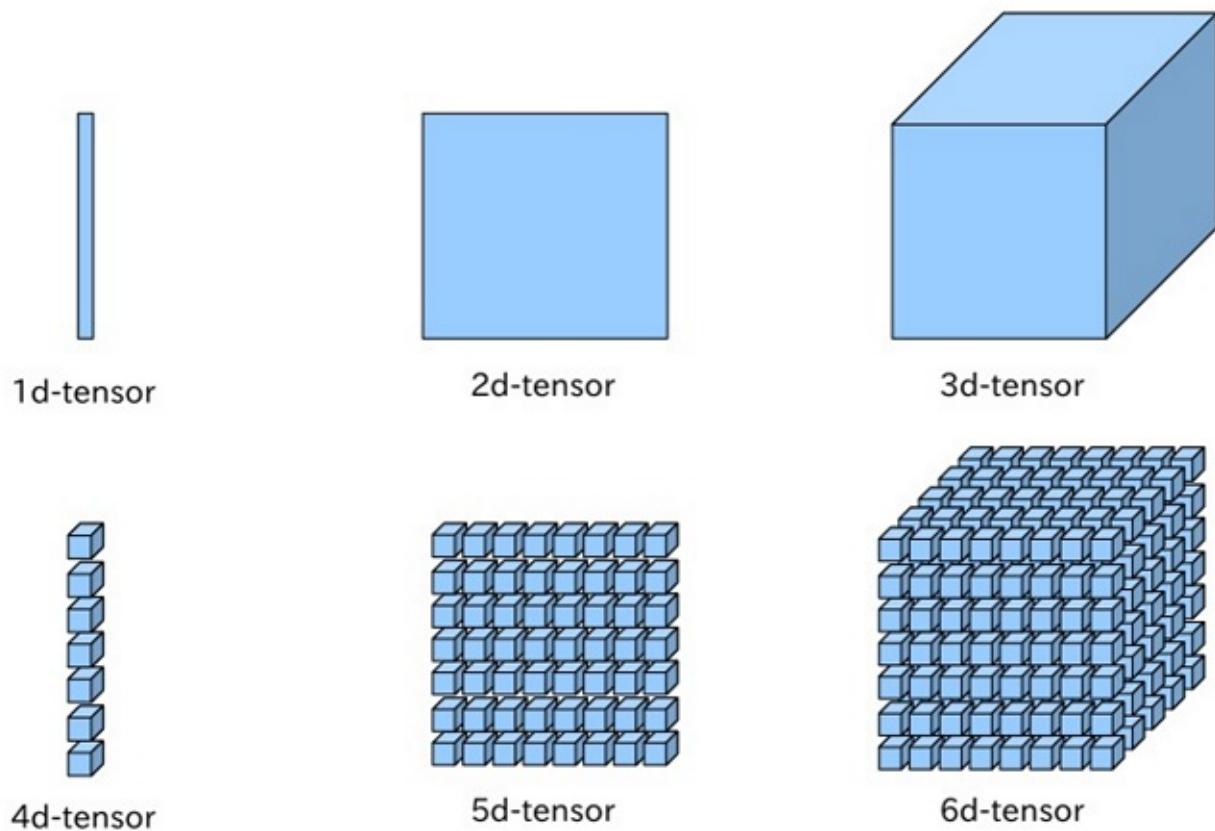
- Identity: If you multiply a matrix B by the identity matrix you will have the matrix B as result, the diagonal of the identity matrix is filled with ones, all the rest are zeros.
- Inverse: Used on matrix division and to solve linear systems.

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Tensors

Sometimes we need to organize information with more than 2 dimensions, we called tensor an n-dimension array.

For example an 1d tensor is a vector, a 2d tensor is a matrix, a 3d tensor is a 3d tensor is a cube, and a 4d tensor is a vector of cubes, a 5d tensor is a matrix of cubes.



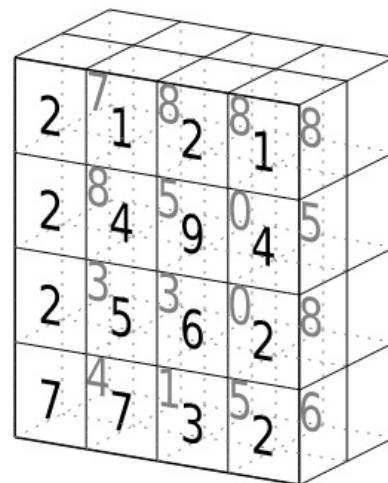
tensor

't'
'e'
'n'
's'
'o'
'r'

tensor of dimensions [6]
(vector of dimension 6)

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

tensor of dimensions [6,4]
(matrix 6 by 4)

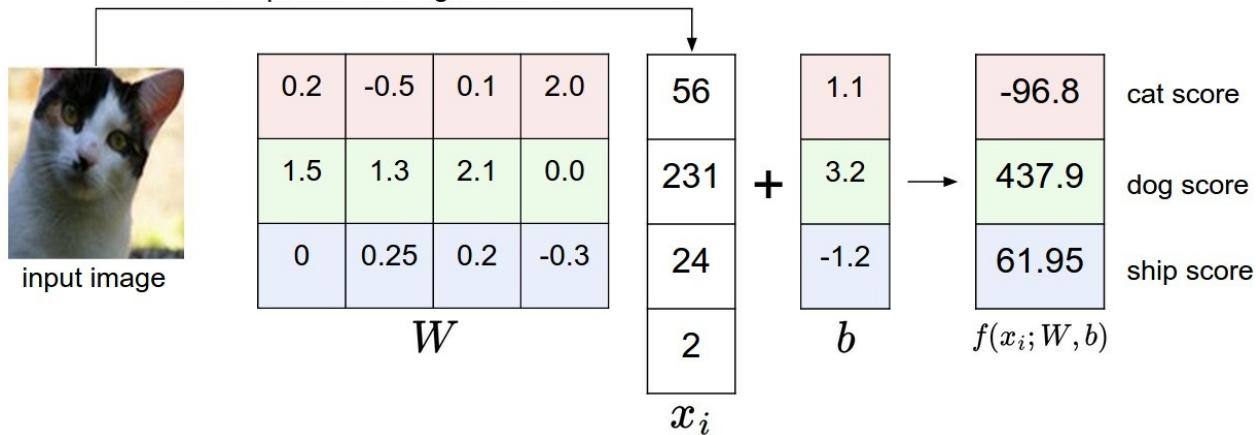


tensor of dimensions [4,4,2]

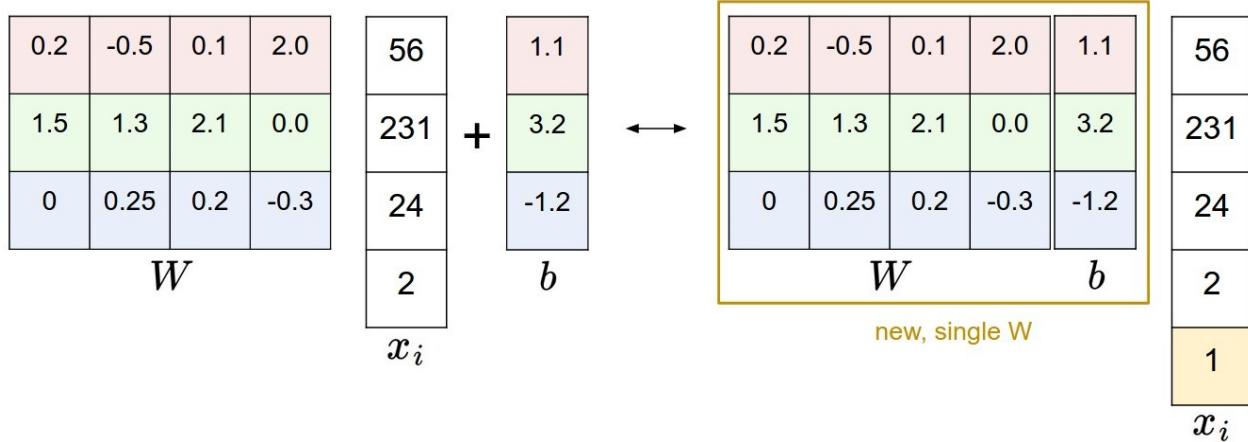
Practical example

Here we will show to use matrix multiplication to implement a linear classifier. Don't care now about what linear classifier does, just pay attention that we use linear algebra do solve it.

stretch pixels into single column



Merging the weights and bias (bias trick) to solve the linear classification as a single matrix multiplication



On Matlab

```
>> W_b = [0.2 -0.5 0.1 2 1.1; 1.5 1.3 2.1 0 3.2; 0 0.25 0.2 -0.3 -1.2]
W_b =
0.2000   -0.5000    0.1000    2.0000    1.1000
1.5000    1.3000    2.1000     0    3.2000
0    0.2500    0.2000   -0.3000   -1.2000

>> x = [56 231 24 2 1]'

x =
56
231
24
2
1

>> W_b*x

ans =
-96.8000
437.9000
60.7500
```

Name	Value	Min	Max
ans	[-96.8000;437.9000;60...]	-96.80...	437.90...
W_b	3x5 double	-1.2000	3.2000
x	[56;231;24;2;1]	1	231

On Python

```
IPython 2.3.0 -- An enhanced Interactive Python.
?          -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: import numpy as np

In [2]: W_b = np.array([ [0.2,-0.5,0.1,2,1.1],[1.5,1.3,2.1,0,3.2],[0,0.25,0.2,-0.3,-1.2] ])

In [3]: x = np.array([56,231,24,2,1])

In [4]: scores = W_b.dot(x)

In [5]: print scores
[ -96.8    437.9     60.75]
```

Next Chapter

The next chapter we will learn about Linear Classification.

Supervised Learning

Introduction

In this chapter we will learn about Supervised learning as well as talk a bit about Cost Functions and Gradient descent. Also we will learn about 2 simple algorithms:

- Linear Regression (for Regression)
- Logistic Regression (for Classification)

The first thing to learn about supervised learning is that every sample data point x has an expected output or label y , in other words your training is composed of $(x^{(i)}, y^{(i)})$ pairs.

For example consider the table below:

Size in feet (x) Price (y)

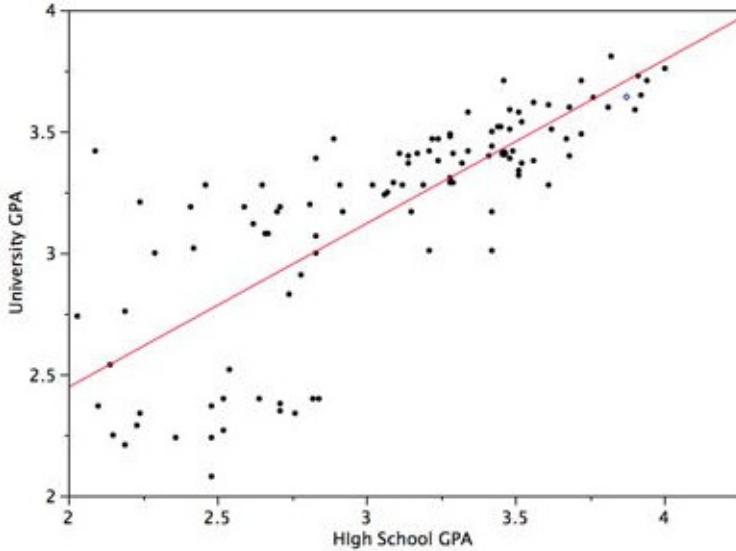
2104	460
1416	232
1534	315
852	178

This table (or training set) shows the sizes of houses along with their price. So a house of size 2104 feet costs 460.

The idea is that we could use data like this to create models that can predict some outcome (ie: Price) from different inputs (ie: Size in feet).

Linear Regression

Regression is about returning a continuous scalar number from some input. The model or hypothesis that we will learn will predict a value following a linear rule. However sometimes a linear model is not enough to capture the underlying nature of your data.



Hypothesis:

Basically linear regression tries to create a line $f(x) = \beta + \alpha \cdot x$, that fits the data on training, e.g.

$$h_{\theta_i}(x) = \theta_0 + \theta_1 \cdot x$$

where $\theta_i \in \{\theta_0, \theta_1\}$

The whole idea of supervised learning is that we try to learn the best parameters (theta in this case) from our training set.

Cost Function

Before we talk about how to learn the parameters (also called weights) of our hypothesis we need to know how to evaluate if our current set of weights are already doing a good job. The function that does this job is called Loss or Cost function. Basically it will return a scalar value between 0 (No error) and infinity (Really bad).

An example of such a function is given below:

$$J(\theta_i) = \frac{1}{2m} \cdot \sum_{i=1}^m [h_{\theta_i}(x^{(i)}) - y^{(i)}]^2$$

where

- m: Number of items in your dataset (i): i-th element of your dataset y: Label(expected value) in dataset

This particular cost function is called mean-squared error loss, and is actually very useful for regression problems.

During training we want to minimise our loss by continually changing the theta parameters. Another nice feature of this particular function is that it is a convex function so it is guaranteed to have no more than one minimum, which will also be its global minimum. This makes it easier for us to optimise.

Our task is therefore to find:

$$\min_{\theta} J(\theta_i)$$

Gradient descent

Gradient descent is a simple algorithm that will try to find the local minimum of a function. We use gradient descent to minimise our loss function. One important feature to observe on gradient descent is that it will more often than not get stuck into the first local minimum that it encounters. However there is no guarantee that this local minimum it finds is the best (global) one.

Algorithm 1 Gradient Descent

Input: Differentiable function $f(x)$ where $f(x) : \mathbf{R}^n \rightarrow \mathbf{R}$

Start point x_{old}

Output: The local minima x^* that minimize $f(x)$

```

1: while TRUE do
2:   tmpDelta ←  $x_{old} - \alpha \cdot (\nabla f(x_{old}))$ 
3:   if abs(tmpDelta -  $x_{old}$ ) < CRITERIA then
4:     break
5:   end if
6:    $x_{old} \leftarrow \text{tmpDelta}$ 
7: end while

```

The gradient descent needs the first derivative of the function that you want to minimise. So if we want to minimise some function by changing parameters, you need to derive this function with respect to these parameters.

One thing to note is as this algorithm will execute on all samples in your training set it does not scale well for bigger datasets.

Simple example

Below we have some simple implementation in matlab that uses gradient descent to minimise the following function:

$$f(x) = x^4 - 3.x^3 + 2$$

It's derivative with respect to x is:

$$\frac{\partial f(x)}{\partial x} = 4.x^3 - 9.x^2$$

The code to find at what point our local minimum occurs is as follows:

```

% Some tests on Gradient descent
% Define parameters start at 3.5
x_old=3.5; alpha=0.01; precision=0.0001;

% Define function

```

```

x_input = [-1:0.01:3.5];
f = @(x) x.^4 - 3*x.^3 + 2;
df = @(x) 4*x.^3 - 9*x.^2;
y_output = f(x_input);
plot(x_input, y_output);

%% Gradient descent algorithm
% Keep repeating until convergence
while 1
    % Evaluate gradients
    tmpDelta = x_old - alpha*(df(x_old));
    % Check Convergence
    diffOldTmp = abs(tmpDelta - x_old);
    if diffOldTmp < precision
        break;
    end
    % Update parameters
    x_old = tmpDelta;
end
fprintf('The local minimum is at %d\n', x_old);

```

Gradient descent for linear regression

In order to use gradient descent for linear regression you need to calculate the derivative of it's loss (means squared error) with respect to it's parameters.

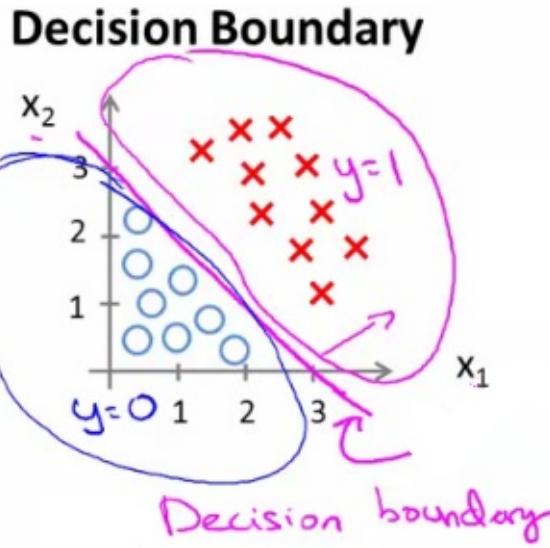
This derivative will be:

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{m} \cdot \sum_{i=1}^m (h_{\theta_i}(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

Logistic Regression

The name may sound confusing but actually Logistic Regression is simply all about classification. For instance consider the example below where we want to classify 2 classes: x's and o's. Now our output y will have two possible values [0,1].

Normally we do not use Logistic Regression if we have a large number of features (e.g. more than 100 features).



Hypothesis

Our hypothesis is almost the same compared to the linear regression however the difference is that now we use a function that will force our output to give $y \in [0, 1]$

$$h_{\theta}(x) = g(\theta^T \cdot x)$$

where

$g(z) = \frac{1}{1+e^{-z}}$ is the logistic or sigmoid function, therefore

$$h_{\theta}(x) = \frac{1}{1+e^{(\theta^T \cdot x)}}$$

Here the sigmoid function will convert a scalar number to some probability between 0 and 1.

Cost function for classification

When dealing with classification problems, we should use a different cost/loss function. A good candidate for classification is the cross-entropy cost function. By the way this function can be found by using the Maximum Likelihood estimation method.

The cross-entropy cost function is:

$$J(\theta) = \frac{1}{m} \cdot \sum_{i=1}^m [y^{(i)} \cdot \log(h_{\theta}(x^{(i)})) + (1 - y^{(i)}) \cdot \log(1 - h_{\theta}(x^{(i)}))]$$

Again our objective is to find the best parameter theta that minimize this function, so to use gradient descent we need to calculate the derivative of this function with respect to theta

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{m} \cdot \sum_{i=1}^m (h_{\theta_i}(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

One cool thing to note is that the derivative is the same as the derivative from linear regression.

Overfitting (Variance) and Underfitting (Bias)

The main idea of training in machine learning is to let the computer learn the underlying structure of the training set and not just the specific training set that it sees. If it does not learn the underlying structure and instead learns just the structure of the training samples then we say our model has overfit.

We can tell overfitting has occurred when our model has a very good accuracy on the training set (e.g.: 99.99%) but does not perform that well on the test set (e.g.: 60%).

This basically occurs when your model is too complex in relation to the available data, and/or you don't have enough data to capture the underlying data pattern.

The opposite of this problem is called Underfitting, basically it happens when your model is too simple to capture the concept given in the training set.

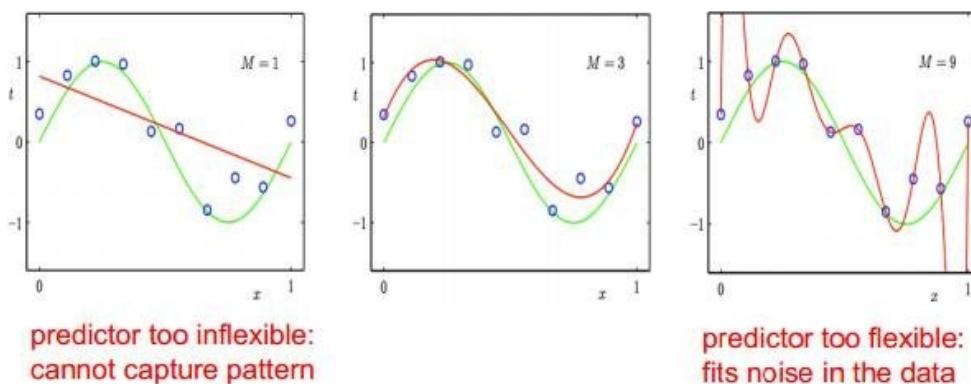
Some graphical examples of these problems are shown below:

Left: Underfit

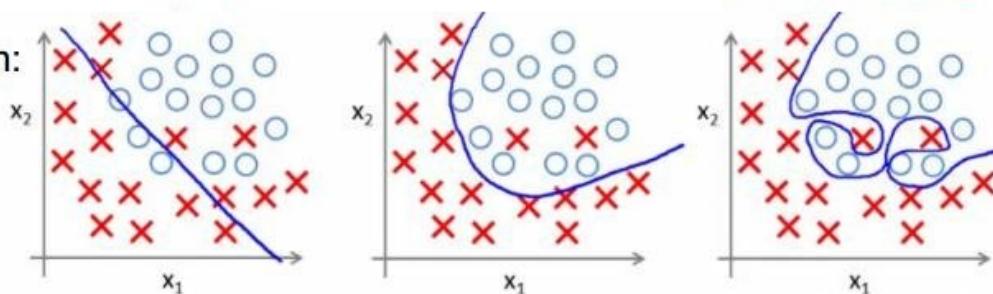
Middle: Perfect

Right: Overfit

Regression:



Classification:



Solving Overfitting

- Get more data
- Use regularisation
- Check other model architectures

Solving Underfitting

- Add more layers or more parameters

- Check other architectures

Regularization

It's a method that helps overfitting by forcing your hypothesis parameters to have nice small values and to force your model to use all the available parameters. To use regularization you just need to add an extra term to your cost/loss function during training. Below we have an example of the Mean squared error loss function with an added regularization term.

$$J(\theta) = \frac{1}{2m} \left[\sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^n \theta_j^2 \right]$$

Also the regularized version of the cross-entropy loss function

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$$

This term will basically multiply by λ all parameters θ from your hypothesis

Normally we don't need to regularise the bias term of our hypothesis.

During gradient descent you also need to calculate the derivative of those terms.

Intuition

You can imagine that besides forcing the weights to have lower values, the regularisation will also spread the "concept" across more weights. One way to think about what happens when we regularise is shown below.

For our input x we can have 2 weight vectors w_1 or w_2 . When we multiply our input with our weight vector we will get the same output of 1 for each. However the weight vector w_2 is a better choice of weight vector as this will look at more of our input x compared to w_1 which only looks at one value of x .

$$x = [1, 1, 1, 1]$$

$$w_1 = [1, 0, 0, 0]$$

$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

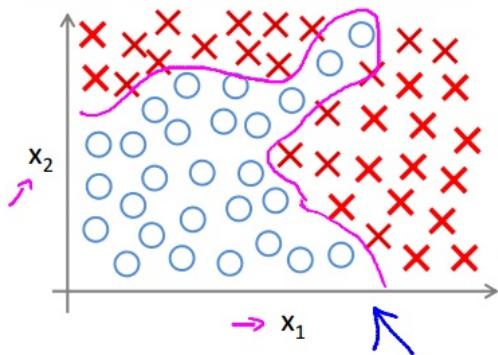
$$\therefore w_1^T \cdot x = w_2^T \cdot x = 1$$

In practice this might reduce performance on our training set but will improve generalisation and thus performance when testing.

Non-Linear Hypothesis

Sometimes our hypothesis needs to have non-linear terms to be able to predict more complex classification/regression problems. We can for instance use Logistic Regression with quadratic terms to capture this complexity. As mentioned earlier this does not scale well for large number of features.

Non-linear Classification



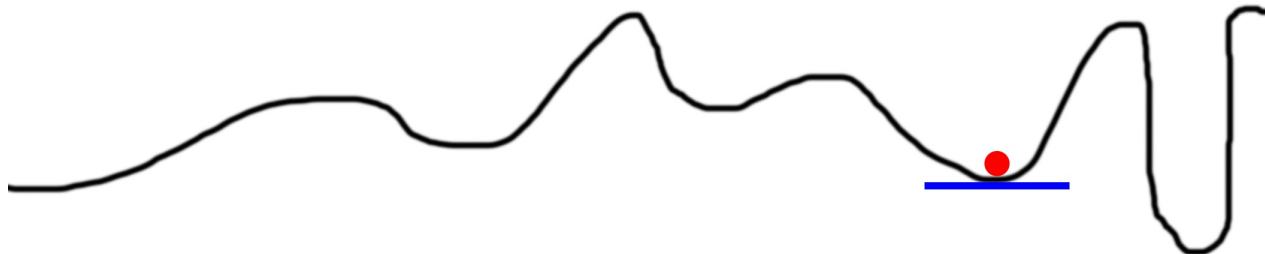
$$\begin{aligned}
 & g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 \\
 & + \theta_3 x_1 x_2 + \theta_4 x_1^2 x_2 \\
 & + \theta_5 x_1^3 x_2 + \theta_6 x_1 x_2^2 + \dots) \\
 \rightarrow & x_1^2, x_1 x_2, x_1 x_3, x_1 x_4 \dots x_1 x_{100} \\
 & x_2^2, x_1 x_3 \dots \\
 \approx & 5000 \text{ feature } \delta(n^2)
 \end{aligned}$$

For instance if we would like to classify a 100×100 grayscale image using logistic regression we would need 50 million parameters to include the quadratic terms. Training with this number of features will need at least 10x more images (500 millions) to avoid overfitting. Also the computational cost will be really high.

In the next chapters we will learn other algorithms that scale better for bigger number of features.

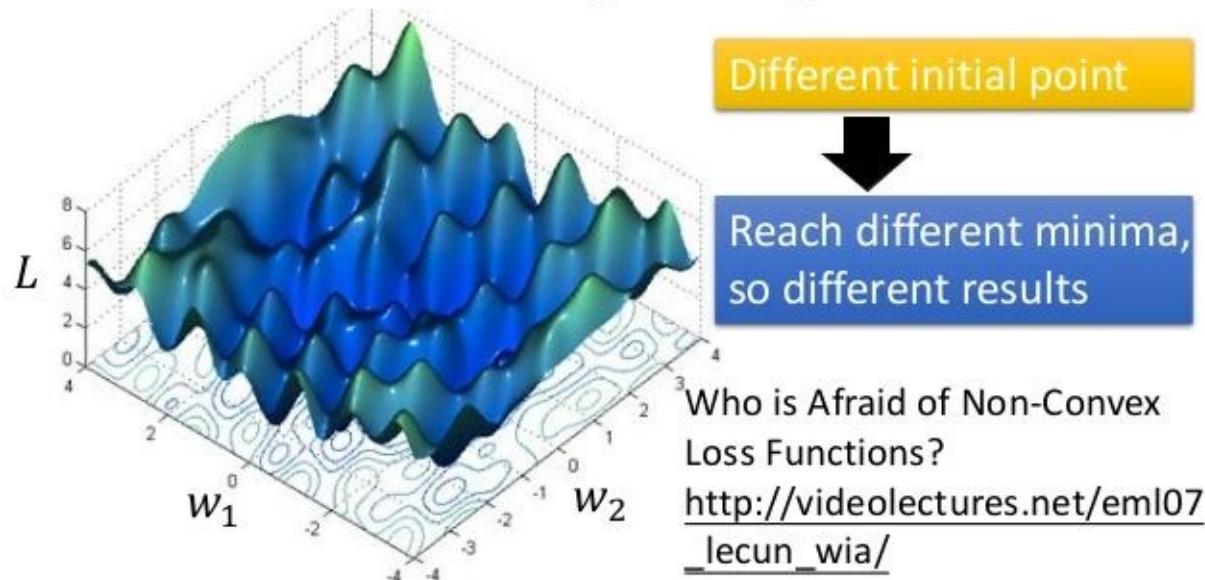
Local minimas

As we increase the number of parameters on our model our loss function will start to find multiple local minima. This can be a problem for training because the gradient descent can get stuck in one of them.



Actually some recent papers also try to prove that some of the methods used today (ie: Deep Neural networks) the local-minima is actually really close to the global minima, so you don't need to care about using a Convex loss or about getting stuck in a local-minima.

- Gradient descent never guarantee global minima



Neural Networks

Introduction

Family of models that takes a very “loose” inspiration from the brain, used to approximate functions that depends on a large number of inputs. (Is a very good Pattern recognition model).

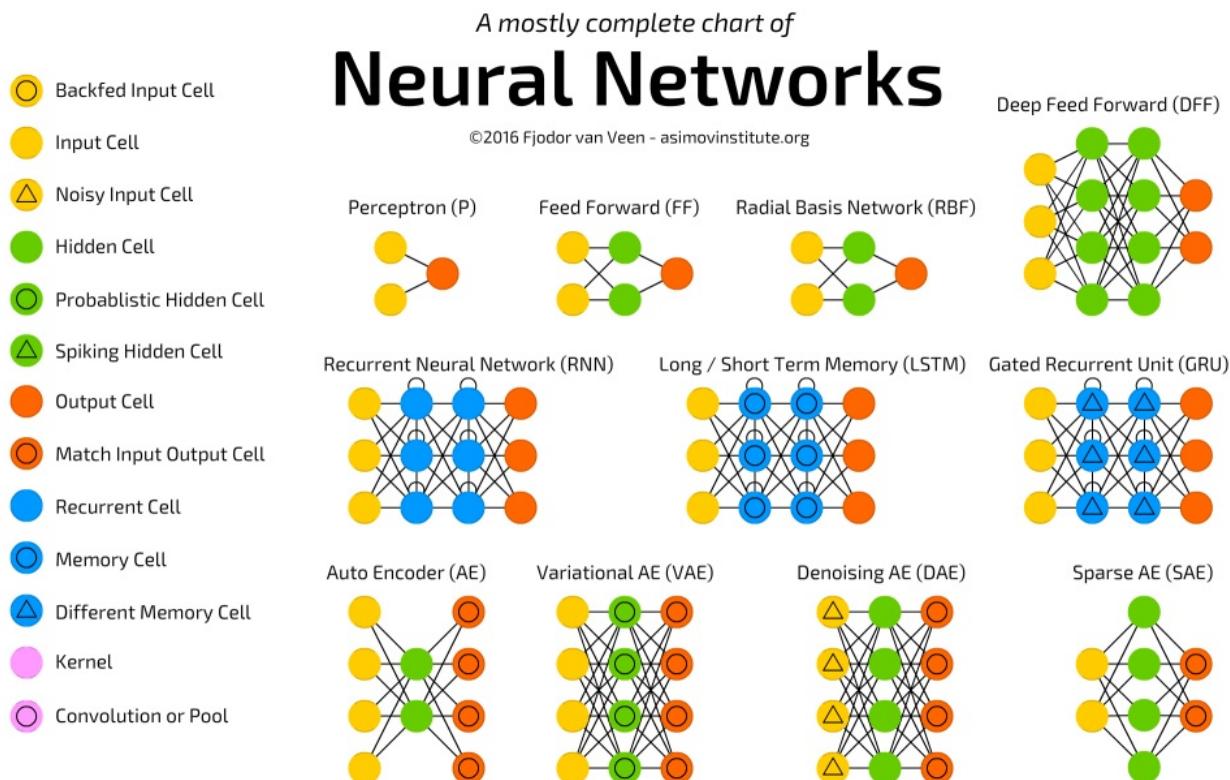
Neural networks are examples of Non-Linear hypothesis, where the model can learn to classify much more complex relations. Also it scale better than Logistic Regression for large number of features.

It's formed by artificial neurons, where those neurons are organised in layers. We have 3 types of layers:

- Input layer
- Hidden layers
- Output layer

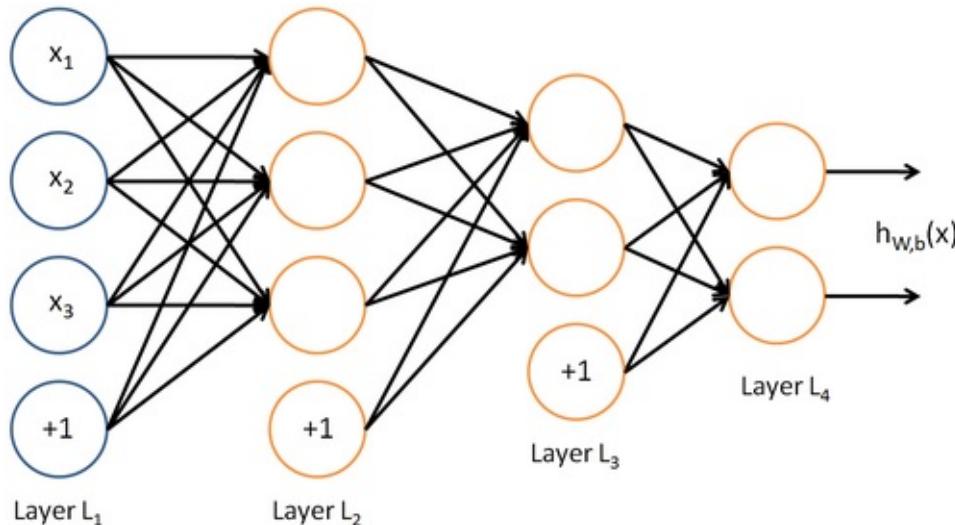
We classify the neural networks from their number of hidden layers and how they connect, for instance the network above have 2 hidden layers. Also if the neural network has/or not loops we can classify them as Recurrent or Feed-forward neural networks.

Neural networks from more than 2 hidden layers can be considered a deep neural network. The advantage of using more deep neural networks is that more complex patterns can be recognised.



Bellow we have an example of a 2 layer feed forward artificial neural network. Imagine that the connections between neurons are the parameters that will be learned during training. On this example

Layer L₁ will be the input layer, L₂/L₃ the hidden layer and L₄ the output layer



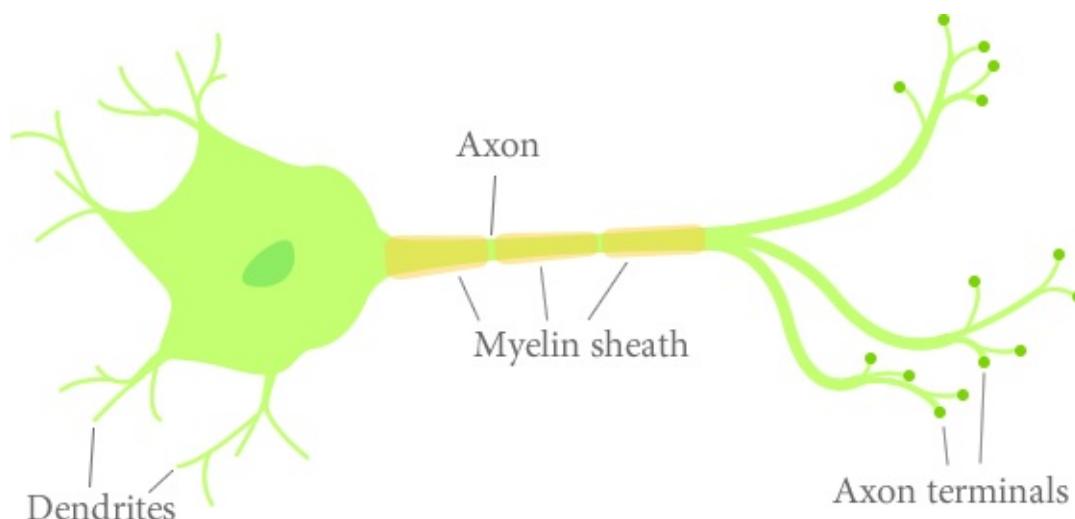
Brain Differences

Now wait before you start thinking that you can just create a huge neural network and call strong AI, there are some few points to remember:

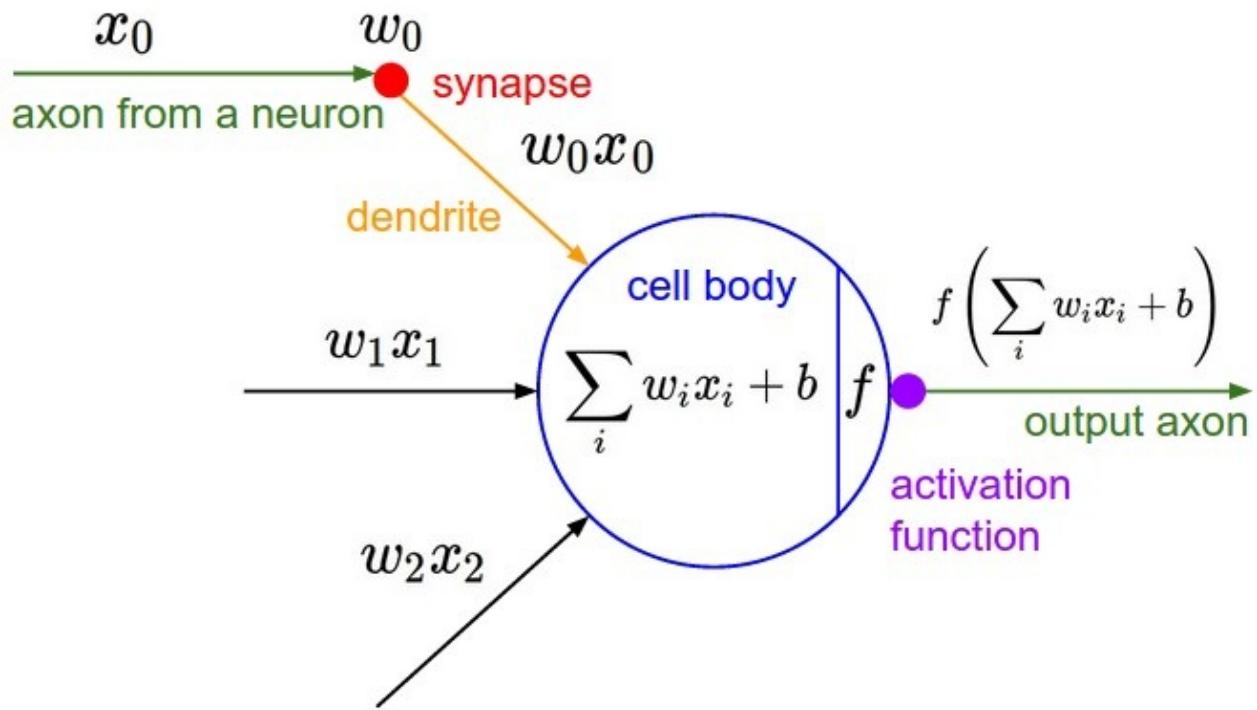
Just a list:

- The artificial neuron fires totally different than the brain
- A human brain has 100 billion neurons and 100 trillion connections (synapses) and operates on 20 watts(enough to run a dim light bulb) - in comparison the biggest neural network have 10 million neurons and 1 billion connections on 16,000 CPUs (about 3 million watts)
- The brain is limited to 5 types of input data from the 5 senses.
- Children do not learn what a cow is by reviewing 100,000 pictures labelled “cow” and “not cow”, but this is how machine learning works.
- Probably we don't learn by calculating the partial derivative of each neuron related to our initial concept. (By the way we don't know how we learn)

Real Neuron



Artificial Neuron



The single artificial neuron will do a dot product between w and x , then add a bias, the result is passed to an activation function that will add some non-linearity. The neural network will be formed by those artificial neurons.

The non-linearity will allow different variations of an object of the same class to be learned separately. Which is a different behaviour compared to the linear classifier that tries to learn all different variations of the same class on a single set of weights. More neurons and more layers is always better but it will need more data to train.

Each layer learn a concept, from it's previous layer. So it's better to have deeper neural networks than a wide one. (Took 20 years to discover this)

Activation Functions

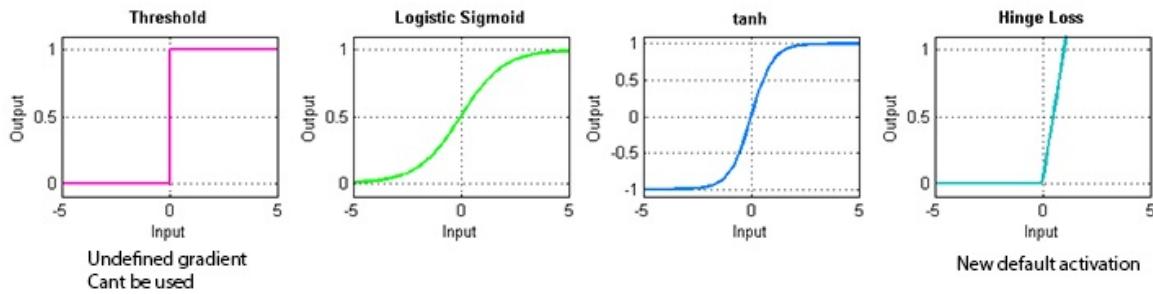
After the neuron do the dot product between it's inputs and weights, it also apply a non-linearity on this result. This non-linear function is called Activation Function.

On the past the popular choice for activation functions were the sigmoid and tanh. Recently it was observed the ReLU layers has better response for deep neural networks, due to a problem called vanishing gradient. So you can consider using only ReLU neurons.

$$\text{sigmoid: } \sigma(x) = \frac{1}{1+e^{-x}}$$

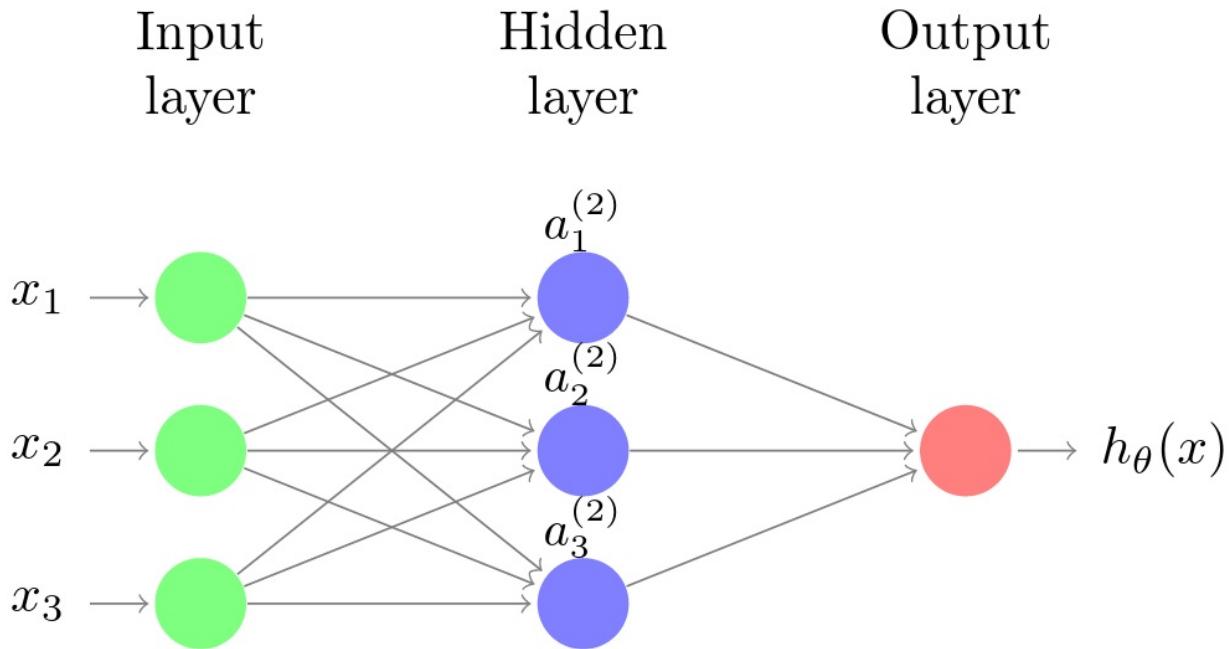
$$\text{tanh: } \sigma(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\text{ReLU: } \sigma(x) = \max(0, x)$$



Example of simple network

Consider the Neural network bellow with 1 hidden layer, 3 input neurons, 3 hidden neurons and one output neuron.



We can define all the operation that this network will do as follows.

$$\begin{aligned} a_1^{(2)} &= g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3) \\ a_2^{(2)} &= g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3) \\ a_3^{(2)} &= g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3) \\ h_{\theta}(x) = a_1^{(3)} &= g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)}) \end{aligned}$$

Here $a_1^{(2)}$ means the activation(output) of the first neuron of layer 2 (hidden layer on this case). The first layer, (input layer) can be considered as $a_n^{(1)}$ and it's values are just the input vector.

Consider the connections between each layer as a matrix of parameters. Consider those matrices as the connections between layers. On this case we have to matrices:

- $\theta^{(1)}$: Map the layer 1 to layer 2 (Input and Hidden layer)
- $\theta^{(2)}$: Map layer 2 to layer 2 (Hidden and output layer)

Also you consider the dimensions of $\theta^{(1)}$ as [number of neurons on layer 2] x [Number of neurons layer 1 + 1]. In other words:

$$s_{j+1} \times (s_j + 1) = 4 \times 3$$

Where:

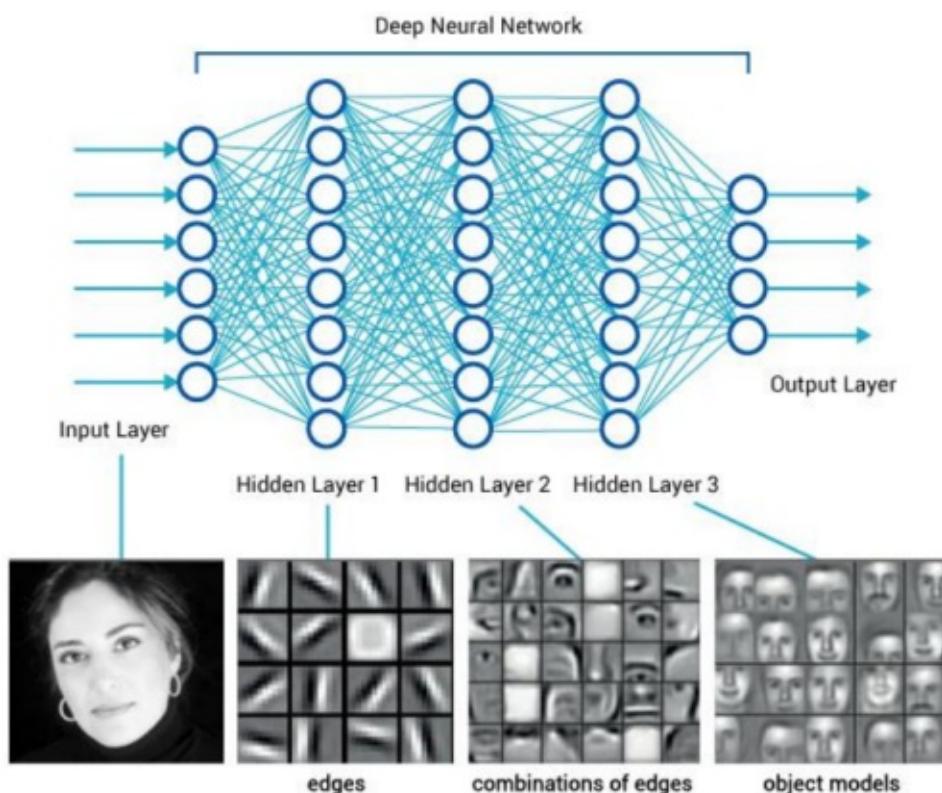
s_{j+1} : Number of neurons on next layer

$s_j + 1$: Number of neurons on the current layer + 1

Notice that this is only true if we consider to add the bias as part of our weight matrices, this could depend from implementation to implementation.

Why is better than Logistic Regression

Consider that the neural network as a cascaded chain of logistic regression, where the input of each layer is the output of the previous one. Another way to think on this is that each layer learn a concept with the output of the previous layer.



This is nice because the layer does not need to learn the whole concept at once, but actually build a chain of features that build that knowledge.

Vectorized Implementation

You can calculate the output of the whole layer $a^{(n)}$ as a matrix multiplication followed by a element-wise activation function. This has the advantage of performance, considering that you are using tools like Matlab, Numpy, or also if you are implementing on hardware.

The mechanism of calculating the output of each layer with the output of the previous layer, from the beginning(input layer) to it's end(output layer) is called forward propagation.

Given one training example (x, y):

Forward propagation:

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

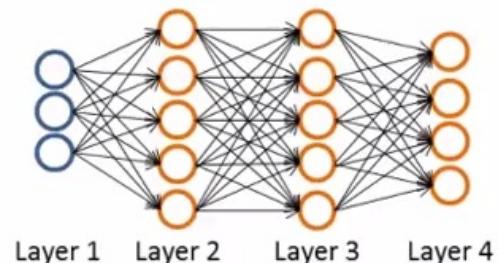
$$a^{(2)} = g(z^{(2)}) \quad (\text{add } a_0^{(2)})$$

$$z^{(3)} = \Theta^{(2)} a^{(2)}$$

$$a^{(3)} = g(z^{(3)}) \quad (\text{add } a_0^{(3)})$$

$$z^{(4)} = \Theta^{(3)} a^{(3)}$$

$$a^{(4)} = h_{\Theta}(x) = g(z^{(4)})$$



To better understanding let's break the activation of some layer as following:

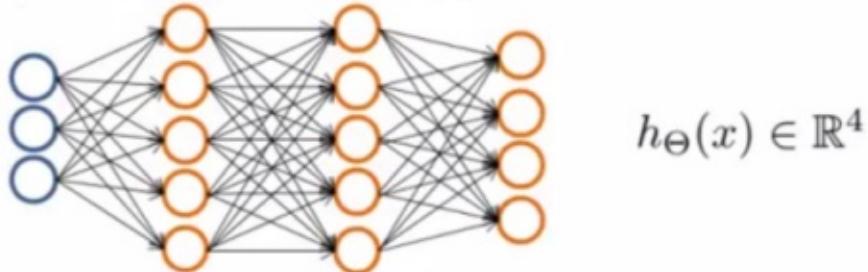
$$A^{(n)} = g(Z^{(n)})$$

$$Z^{(n)} = \Theta^{(n-1)} \cdot A^{(n-1)}$$

So using this formulas you can calculate the activation of each layer.

Multiple class problems

On the multi-class classification problem you need to allocate one neuron for each class, than during training you provide a one-hot vector for each one of your desired class. This is somehow easier than the logistic regression one-vs-all training.



Want $h_{\Theta}(x) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, $h_{\Theta}(x) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$, $h_{\Theta}(x) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$, etc.
 when pedestrian when car when motorcycle

Training set: $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})$

$y^{(i)}$ one of $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$
 pedestrian car motorcycle truck

Cost function (Classification)

The cost function of neural networks, it's a little more complicated than the logistic regression. So for classification on Neural networks we should use:

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K \left[y_k^{(i)} \log((h_{\Theta}(x^{(i)}))_k) + (1 - y_k^{(i)}) \log(1 - (h_{\Theta}(x^{(i)}))_k) \right] + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\Theta_{j,i}^{(l)})^2$$

Where:

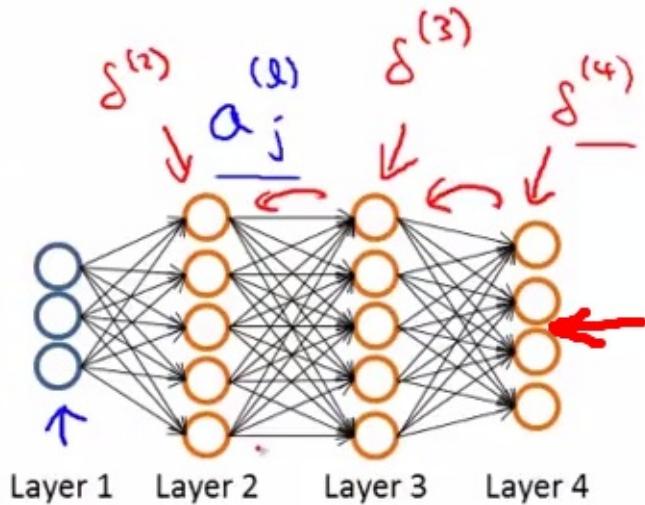
- L: Number of layers
- m: Dataset size
- K: Number of classes
- s_l : Number of neurons (not counting bias) from layer l

During training we need to calculate the partial derivative of this cost function with respect to each parameter on your neural network. Actually what we need to compute is:

- The loss itself (Forward-propagation)
- The derivative of the loss w.r.t each parameter (Back-propagation)

Backpropagation

Backpropagation it's an efficient algorithm that helps you calculate the derivative of the cost function with respect to each parameter of the neural network. The name backpropagation comes from the fact that now we start calculating errors from all your neurons from the output layer to the input layer direction. After those errors are calculated we simply multiply them by the activation calculated during forward propagation.



Doing the backpropagation (Vectorized)

As mentioned the backpropagation will flow on the reverse order iterating from the last layer. So starting from the output layer we calculate the output layer error.

The "error values" for the last layer are simply the differences of our actual results in the last layer and the correct outputs in y .

$$\delta^{(L)} = y - a^{(L)}$$

Where

- y : Expected output from training
- $a^{(L)}$: Network output/activation of the last L layer

For all other layers (layers before the last layer until the input)

$$\delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)}) . * g'(z^{(l)})$$

Where

- $\delta^{(l)}$: Error of layer l
- $g'(x)$: Derivative of activation function
- $z^{(l)}$: Pre-activation of layer l
- $.*$: Element wise multiplication

After all errors(delta) are calculated we need to actually calculate the derivative of the loss, which is the product of the error times the activation of each respective neuron:

$$\frac{\partial J(\Theta)}{\partial \Theta_{i,j}^{(l)}} = \frac{1}{m} \sum_{t=1}^m a_j^{(t)(l)} \delta_i^{(t)(l+1)}$$

Now we're ignoring the regularisation term.

Complete algorithm

Bellow we describe the whole procedure in pseudo-code

Algorithm 1 General training

Input: Training-set, model

Output: model trained

```

1: for i = 1 to (numEpoch) do
2:    $\Delta^{(l)} \leftarrow 0$ 
3:   for m = 1 to (sizeDataset) do
4:     do forward Propagation
5:     get loss
6:      $\Delta_{\text{backProp}} \leftarrow$  do backward Propagation
7:      $\Delta^{(l)} \leftarrow \Delta^{(l)} + \Delta_{\text{backProp}}$ 
8:   end for
9:    $D^{(l)} \leftarrow \frac{1}{\text{sizeDataset}} (\Delta^{(l)} + \lambda \Theta^{(l)})$ ,  $D^{(l)} \leftarrow \frac{1}{\text{sizeDataset}} \Delta^{(l)}$ 
10: end for
```

Algorithm 2 Forward Forward

Input: Input X

Output: $h_\theta(x)$, A, Z for each layer

```

1: for l = 1 to (numLayers) do
2:    $Z^{(l+1)} \leftarrow \Theta^{(l)}.A^{(l)}$ 
3:    $A^{(l+1)} \leftarrow g(Z^{(l+1)})$ 
4: end for
```

Algorithm 3 Backward Forward

Input: Loss, Model activations and pre-activations

Output: Δ_{backprop}

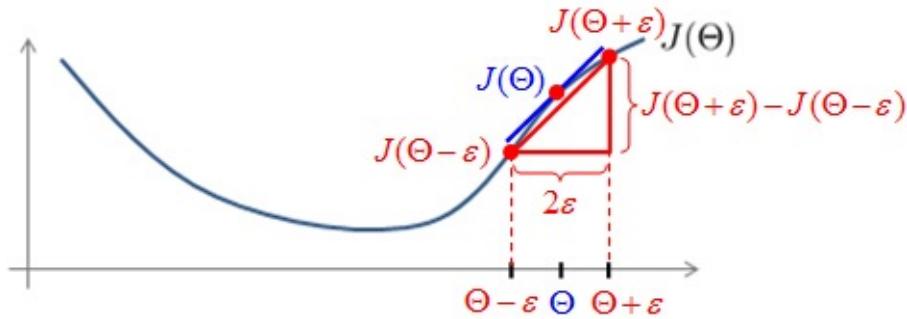
```

1:  $\delta^{(\text{numLayers})} = y - A^{(\text{numLayers})}$ 
2: for l=(numLayers-1) downto 1 do
3:    $\delta^{(l)} \leftarrow ((\Theta^{(l)})^T \delta^{(l+1)}) . * g'(Z^{(l)})$ 
4: end for
5: for l=(numLayers-1) downto 1 do
6:    $\Delta_{\text{backprop}}^{(l)} \leftarrow \delta^{(l+1)}(a^{(l)})^T$ 
7: end for
```

Gradient Checking

In order to verify if your backpropagation code is right we can estimate the gradient, using other algorithm, unfortunately we cannot use this algorithm in practice because it will make the training slow, but we can use to compare it's results with the backpropagation. Basically we will calculate numerically the derivative of the loss with respect to each parameter by calculating the loss and

adding a small perturbation (ie: 10^{-4}) to each parameter one at a time.



$$\frac{\partial}{\partial \Theta} J(\Theta) \approx \frac{J(\Theta + \epsilon) - J(\Theta - \epsilon)}{2\epsilon}$$

Actually you will compare this gradient with the output of the backpropagation

$$\frac{\partial}{\partial \Theta} J(\Theta) \approx \Delta_{\text{Backprop}}$$

Again this will be really slow because we need to calculate the loss again with this small perturbation twice for each parameter.

For example suppose that you have 3 parameters $\Theta \in \{\theta_1, \theta_2, \theta_3\}$

$$\frac{\partial}{\partial \theta_1} J(\Theta) \approx \frac{J(\theta_1 + \epsilon, \theta_2, \theta_3) - J(\theta_1 - \epsilon, \theta_2, \theta_3)}{2\epsilon}$$

$$\frac{\partial}{\partial \theta_2} J(\Theta) \approx \frac{J(\theta_1, \theta_2 + \epsilon, \theta_3) - J(\theta_1, \theta_2 - \epsilon, \theta_3)}{2\epsilon}$$

$$\frac{\partial}{\partial \theta_3} J(\Theta) \approx \frac{J(\theta_1, \theta_2, \theta_3 + \epsilon) - J(\theta_1, \theta_2, \theta_3 - \epsilon)}{2\epsilon}$$

Weights initialization

The way that you initialize your network parameters is also important, you cannot for instance initialize all your weights to zero, normally you want to initialize them with small random values on the range $[-\epsilon_{\text{rand}}, \epsilon_{\text{rand}}]$ but somehow also take into account that you don't want to have some sort of symmetry of the random values between layers.

```
Theta1 = rand(10,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
Theta2 = rand(10,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
Theta3 = rand(1,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
```

What we can see from the code above is that we create random numbers independently for each layer,

and all of them in between the range $[-\epsilon_{rand}, \epsilon_{rand}]$

Training steps

Now just to list the steps required to train a neural network. Here we mention the term epoch which means a complete pass intro all elements of your training set. Actually you repeat your training set over and over because the weights don't completely learn a concept in a single epoch.

1. Initialize weights randomly
2. For each epoch
3. Do the forward propagation
4. Calculate loss
5. Do the backward propagation
6. Update weights with Gradient descent (Optionally use gradient checking to verify backpropagation)
7. Go to step 2 until you finish all epochs

Training/Validation/Test data

Some good practices to create your dataset for training your hypothesis models

1. Collect as many data as possible
2. Merge/Shuffle all this data
3. Divide this dataset into train(60%)/validation(20%)/test(20%) set
4. Avoid having test from a different distribution of your train/validation
5. Use the validation set to tune your model (Number of layers/neurons)
6. Check overall performance with the test set

When we say to use the validation set it means that we're going to change parameters of our model and check which one get better results on this validation set, don't touch your training set. If you are having bad results on your test set consider getting more data, and verify if your train/test/val come from the same distribution.

Effects of deep neural networks

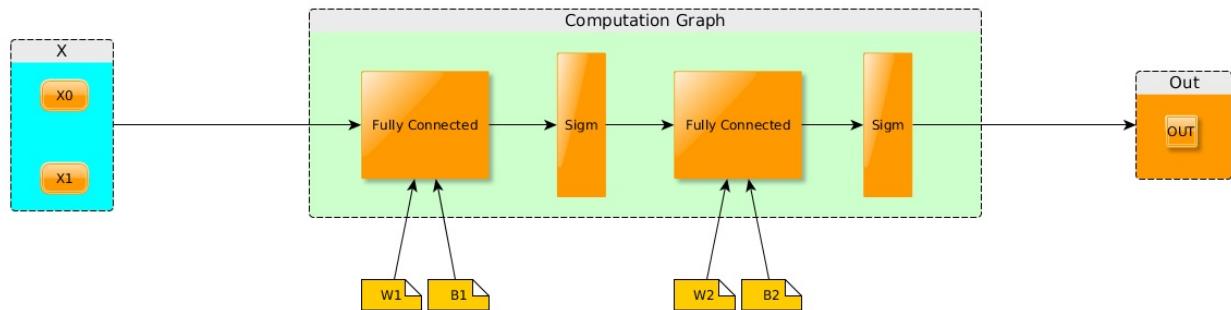
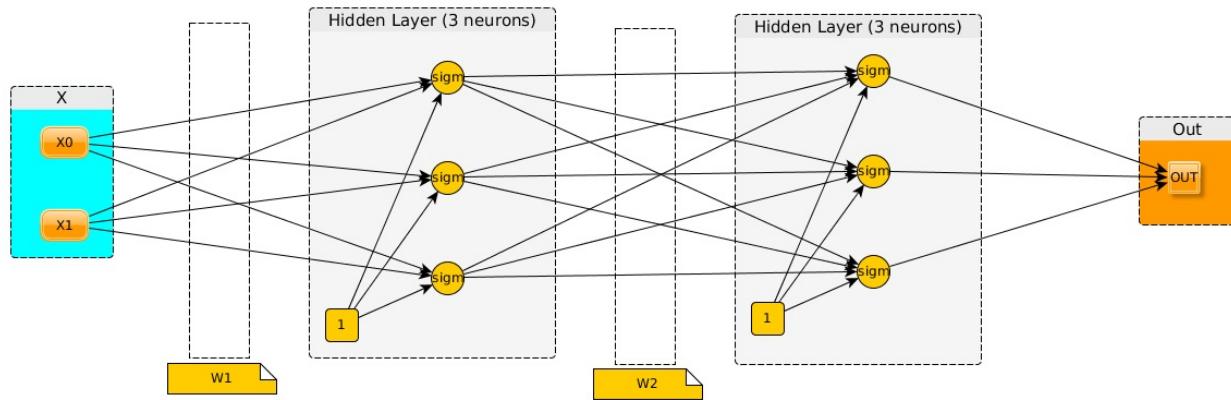
As mentioned earlier having deeper and bigger neural networks is always better in terms of recognition performance, but some problems also arise with more complex models.

1. Deeper and more complex neural networks, need more data to train (10x number of parameters)
2. Over-fit can become a problem so do regularization (Dropout, L2 regularization)
3. Prediction time will increase.

Neural networks as computation graphs

In order to calculate the back-propagation, it's easier if you start representing your hypothesis as computation graphs. Also in next chapters we use different types of layers working together, so to simplify development consider the neural networks as computation graphs. The idea is that if you

provide for each node of your graph the forward/backward implementation, the back propagation becomes much more easier.

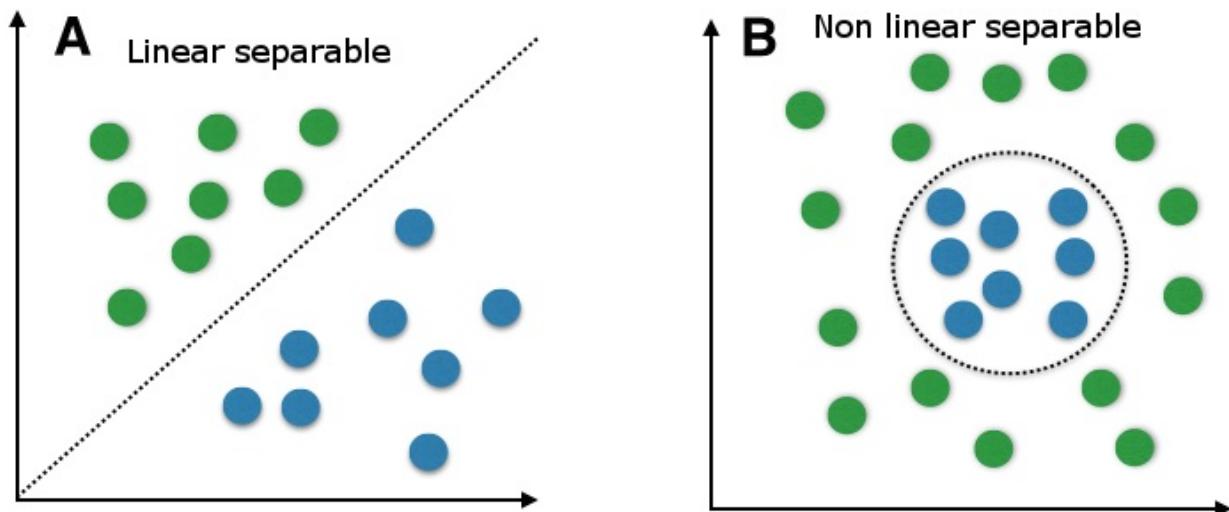


Linear Classification

Introduction

A linear classifier does classification decision based on the value of a linear combination of the characteristics. Imagine that the linear classifier will merge into its weights all the characteristics that define a particular class. (Like merge all samples of the class cars together)

This type of classifier works better when the problem is linear separable.

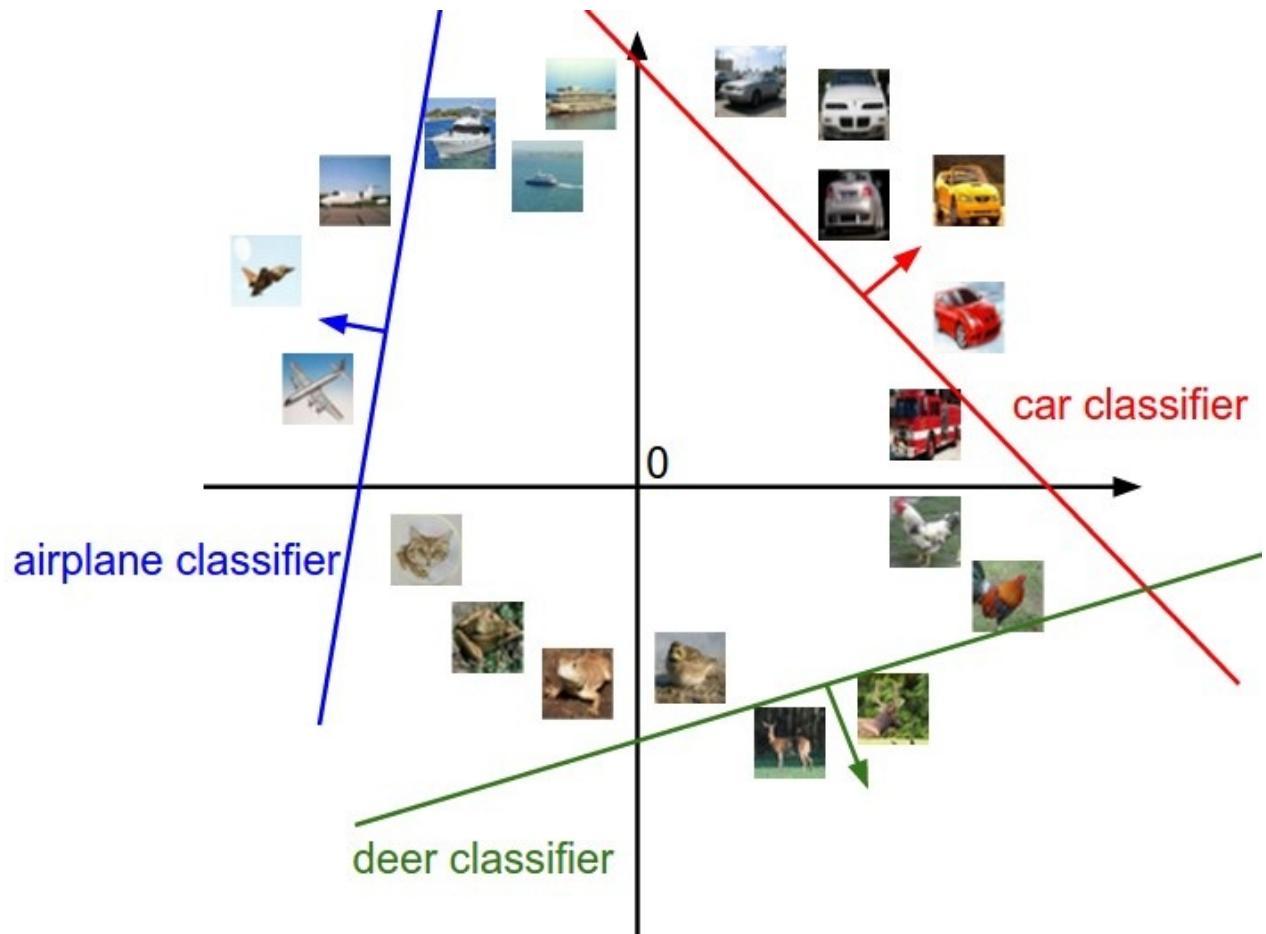


$$f(\vec{x}, \vec{W}, \vec{b}) = \sum j(W_j x_j) + b$$

x: input vector

W: Weight matrix

b: Bias vector



The weight matrix will have one row for every class that needs to be classified, and one column for every element(feature) of x . On the picture above each line will be represented by a row in our weight matrix.

Weight and Bias Effect

The effect of changing the weight will change the line angle, while changing the bias, will move the line left/right

Parametric Approach



image parameters

$$f(\mathbf{x}, \mathbf{W})$$

10 numbers,
indicating class
scores

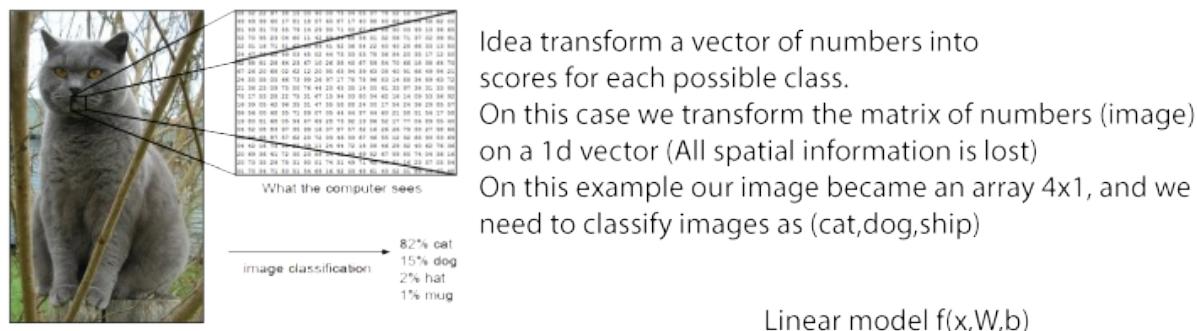
[32x32x3]
array of numbers 0...1
(3072 numbers total)

The idea is that our hypothesis/model has parameters, that will aid the mapping between the input vector to a specific class score. The parametric model has two important components:

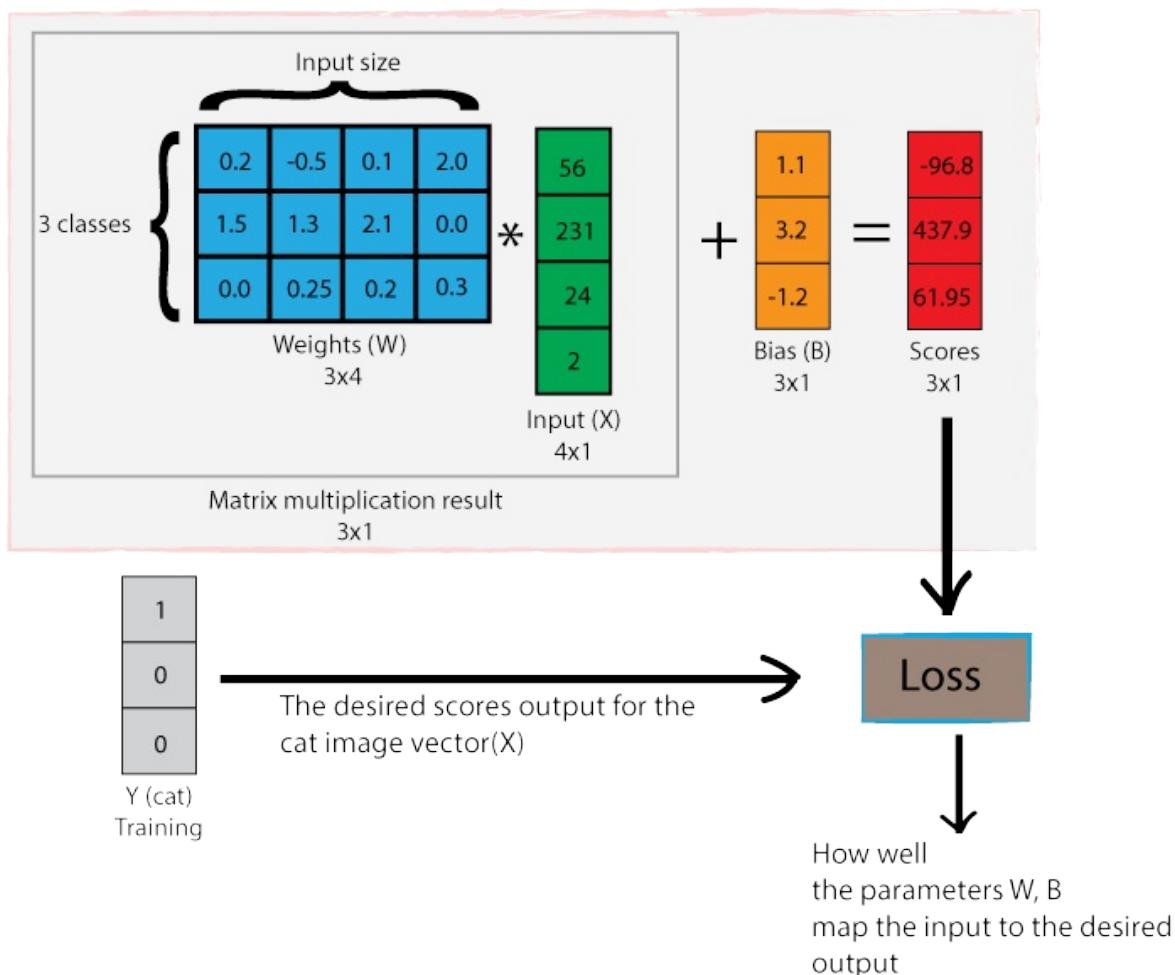
- Score Function: Is a function $f(x, W, b)$ that will map our raw input vector to a score vector
- Loss Function: Quantifies how well our current set of weights maps some input x to a expected output y , the loss function is used during training time.

On this approach, the training phase will find us a set of parameters that will change the hypothesis/model to map some input, to some of the output class.

During the training phase, which consist as a optimisation problem, the weights (W) and bias (b) are the only thing that we can change.



Linear model $f(x, W, b)$



Now some topics that are important on the diagram above:

1. The input image x is stretched to a single dimension vector, this loose spatial information
2. The weight matrix will have one column for every element on the input
3. The weight matrix will have one row for every element of the output (on this case 3 labels)
4. The bias will have one row for every element of the output (on this case 3 labels)
5. The loss will receive the current scores and the expected output for it's current input X

Consider each row of W a kind of pattern match for a specified class. The score for each class is calculated by doing a inner product between the input vector X and the specific row for that class. Ex:

$$score_{cat} = [0.2(56) - 0.5(231) + 0.1(24) + 2(2)] + 1.1 = -96.8$$

Example on Matlab

```
>> W = [0.2 -0.5 0.1 2; 1.5 1.3 2.1 0; 0 0.25 0.2 0.3]

W =
0.2000    -0.5000    0.1000    2.0000
1.5000    1.3000    2.1000    0
0    0.2500    0.2000    0.3000

>> X = [56 231 24 2]'

X =
56
231
24
2

>> B = [1.1 3.2 -1.2]

B =
1.1000    3.2000   -1.2000

>> scores = (W*X)+B'

scores =
-96.8000
437.9000
61.9500
```

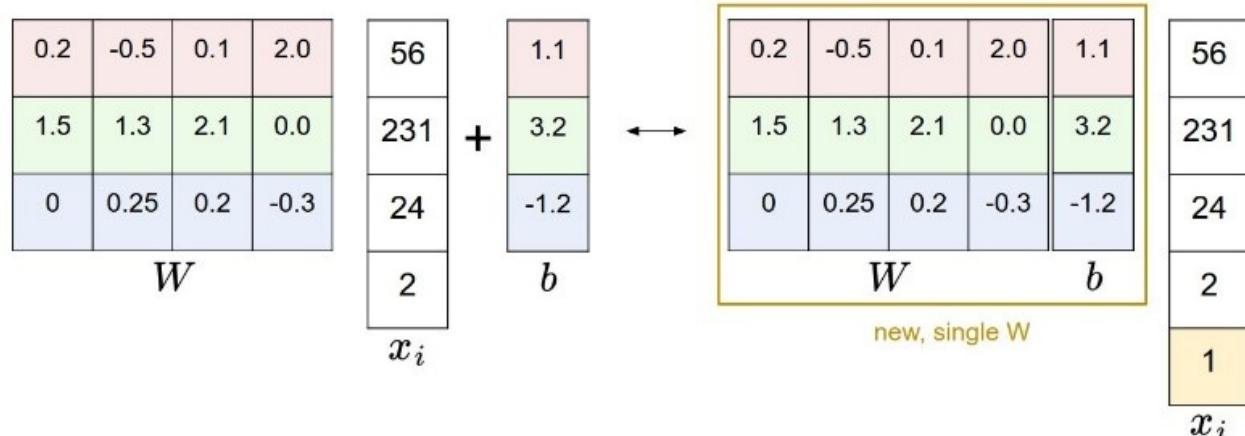
The image bellow reshape back the weights to an image, we can see by this image that the training try to compress on each row of W all the variants of the same class. (Check the horse with 2 heads)



Bias trick

Some learning libraries implementations, does a trick to consider the bias as part of the weight matrix, the advantage of this approach is that we can solve the linear classification with a single matrix multiplication.

$$f(x, W) = W \cdot x$$



Basically you add an extra row at the end of the input vector, and concatenate a column on the W matrix.

Input and Features

The input vector sometimes called feature vector, is your input data that is sent to the classifier. As the linear classifier does not handle non-linear problems, it is the responsibility of the engineer, process this data and present it in a form that is separable to the classifier.

The best case scenario is that you have a large number of features, and each of them has a high correlation to the desired output and low correlation between them

Loss Function

Introduction

As mention earlier the Loss/Cost functions are mathematical functions that will answer how well your classifier is doing it's job with the current set of parameters (Weights and Bias). One important step on supervised learning is the choice of the right loss function for the job/task.

Model Optimization

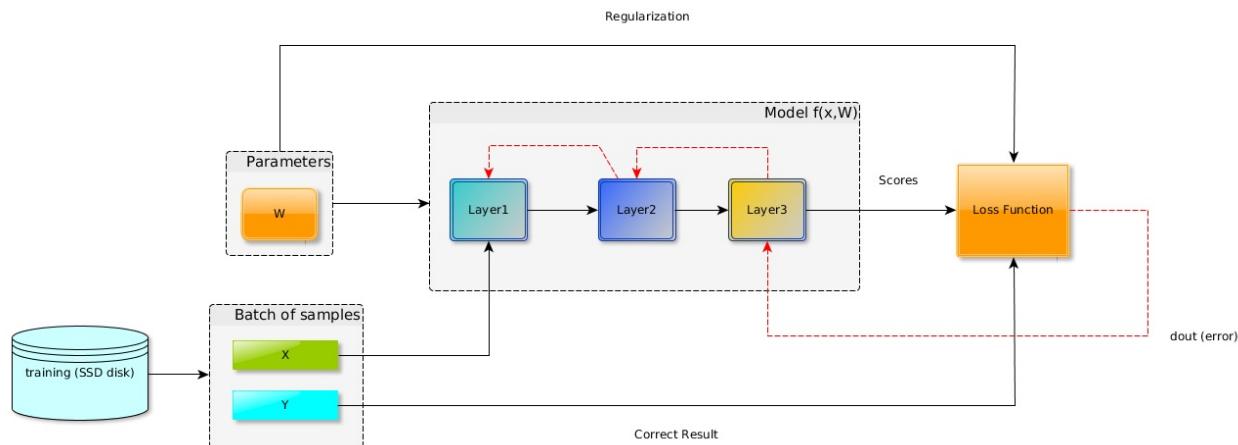
Model Optimization

Introduction

Machine learning models learn by updating its parameters (weights and biases) towards the direction of the correct classification.

Basic structure of a learning model

On the picture below we will show the basic blocks of a machine learning model



On this picture we can detect the following components

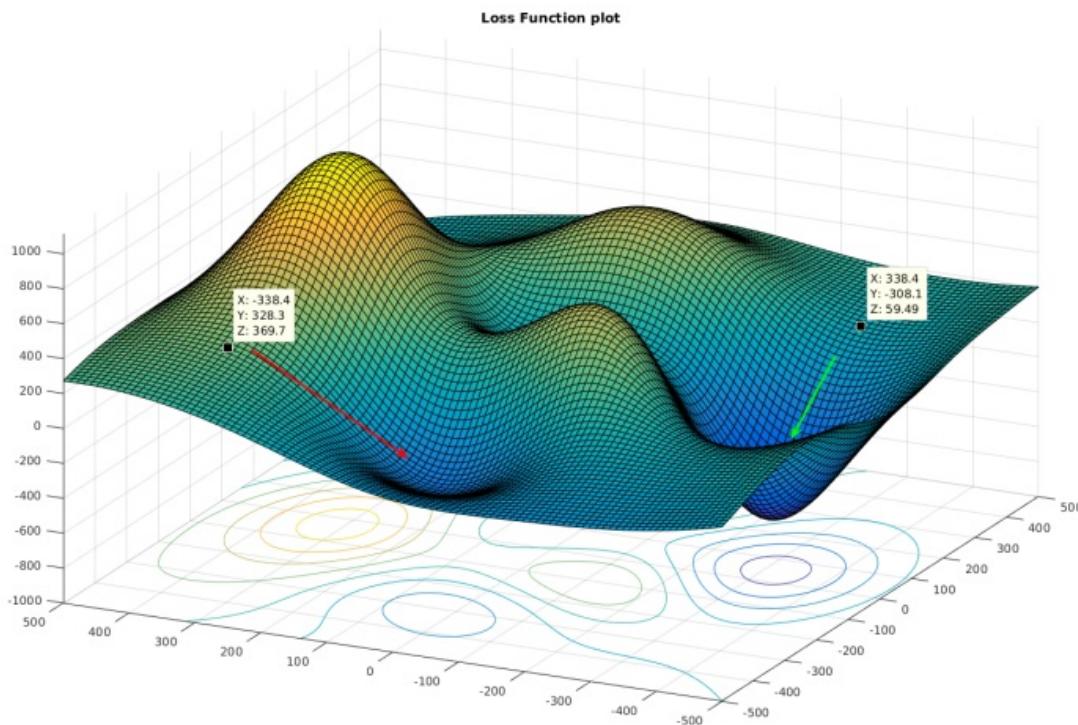
1. Training dataset: Basically a high speed disk containing your training data
2. Batch of samples: A list of pairs (X,Y), consisting on inputs, expected outputs, for example X can be a image and Y the label "cat"
3. Parameters: Set of parameters used by your model layers, to map X to Y
4. Model: Set of computing layers that transform an input X and weights W, into a score (probable Y)
5. Loss function: Responsible to say how far our score is from the ideal response Y, the output of the loss function is a scalar. Another way is also to consider that the loss function say how bad is your current set of parameters W.

What we will do?

Basically we need an algorithm that will change our weight and biases, in order to minimize our loss function.

The Loss function

You can imagine the loss function here as a place with some mountains, and your objective is to find its vale (lowest) place. Your only instrument is the gadget that returns your altitude (loss). You need to find out which direction to take.



Here we can also observe two things:

- You have more than one value (Local minima)
- Depending where you landed you probably find one instead of the other (Importance of weight initialization)

Which direction to take (Gradient descent)

We can use calculus to discover which direction to take, for instance if we follow the derivative of the loss function, we can guarantee that we always go down. This is done just by subtracting the current set of weights by derivative of the loss function evaluated at that point.

$$W_{new} = W_{old} - (\gamma \cdot \nabla_{wL})$$

On multiple dimensions we have a vector of partial derivatives, which we call gradient.

Observe that we multiply the gradient by a factor γ (step-size, learning-rate), that is normally a small value ex: 0.001

Simple Example

To illustrate the gradient descent method let's follow a simple case in 1D. Consider the initial weight (-1.5)

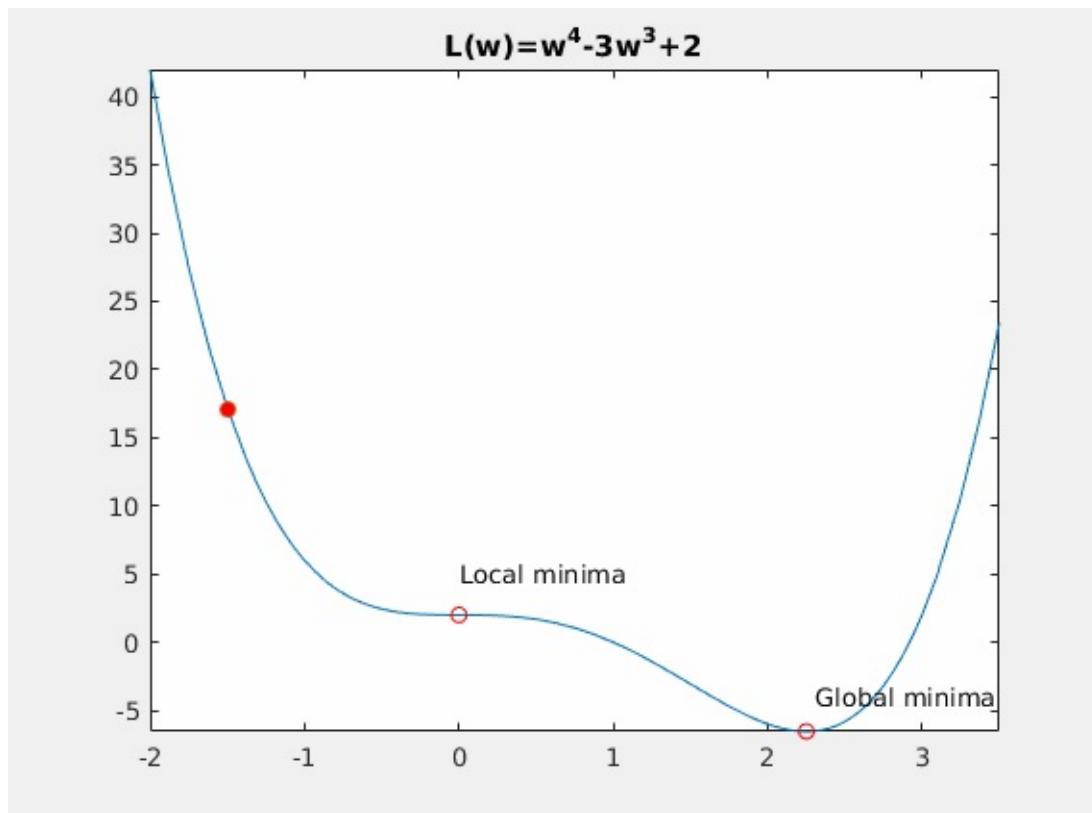
$$L(w) = w^4 - 3w^3 + 2 \therefore W_{old} = -1.5$$

$$\nabla_L = \frac{dL(w)}{dw} = 4w^3 - 9w^2 \therefore \text{evaluate on current weight}$$

$$\frac{dL(-1.5)}{dw} = 4(1.5)^3 - 9(-1.5)^2 = -33.75 \Rightarrow \nabla_{Lw}$$

$$\nabla_{Lw} = -33.75 \therefore$$

$$W_{new} = (-1.5) - (0.001 \cdot (-33.75)) \Rightarrow -1.49$$



On Matlab

```

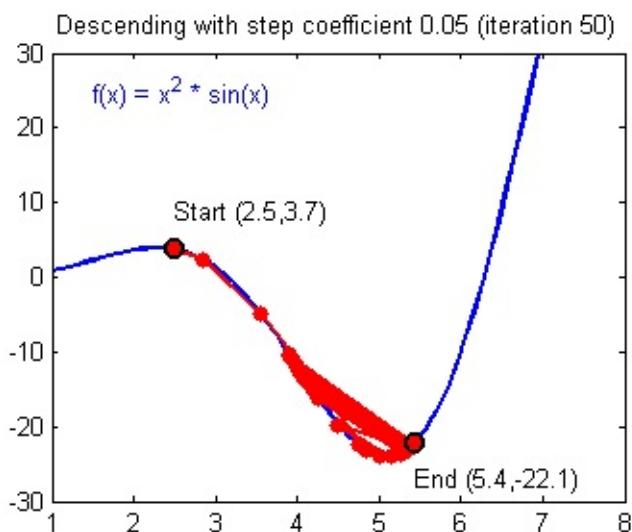
1 % Define function symbolically
2 - syms w; L_w = w^4-3*w^3+2;
3
4 % Get numeric version of loss function
5 - L_w_mat = matlabFunction(L_w);
6
7 % Calculate derivative of loss related to w
8 - deriv_f_w = diff(L_w,w);
9
10 % Get numeric version of derivative of loss function
11 - f_derivative = matlabFunction(deriv_f_w);
12
13 % Symbolic find the critical points (Solve derivative equal to zero)
14 - crit_pts = solve(deriv_f_w);
15
16 % Initial weight value and step size
17 - weight = -1.5;
18 - step = 0.001;
19
20 %% Gradient descent
21 - for iters=1:100
22 -     % Get gradient by evaluating derivative of Loss related to W
23 -     weight_grad = f_derivative(weight);
24 -     weight = weight - (step*weight_grad);
25 - end

```

We can observe that after we use calculus do find the derivative of the loss function, we just needed to evaluate it with the current weight. After that we just take the evaluated value from the current weight.

Learning rate too big

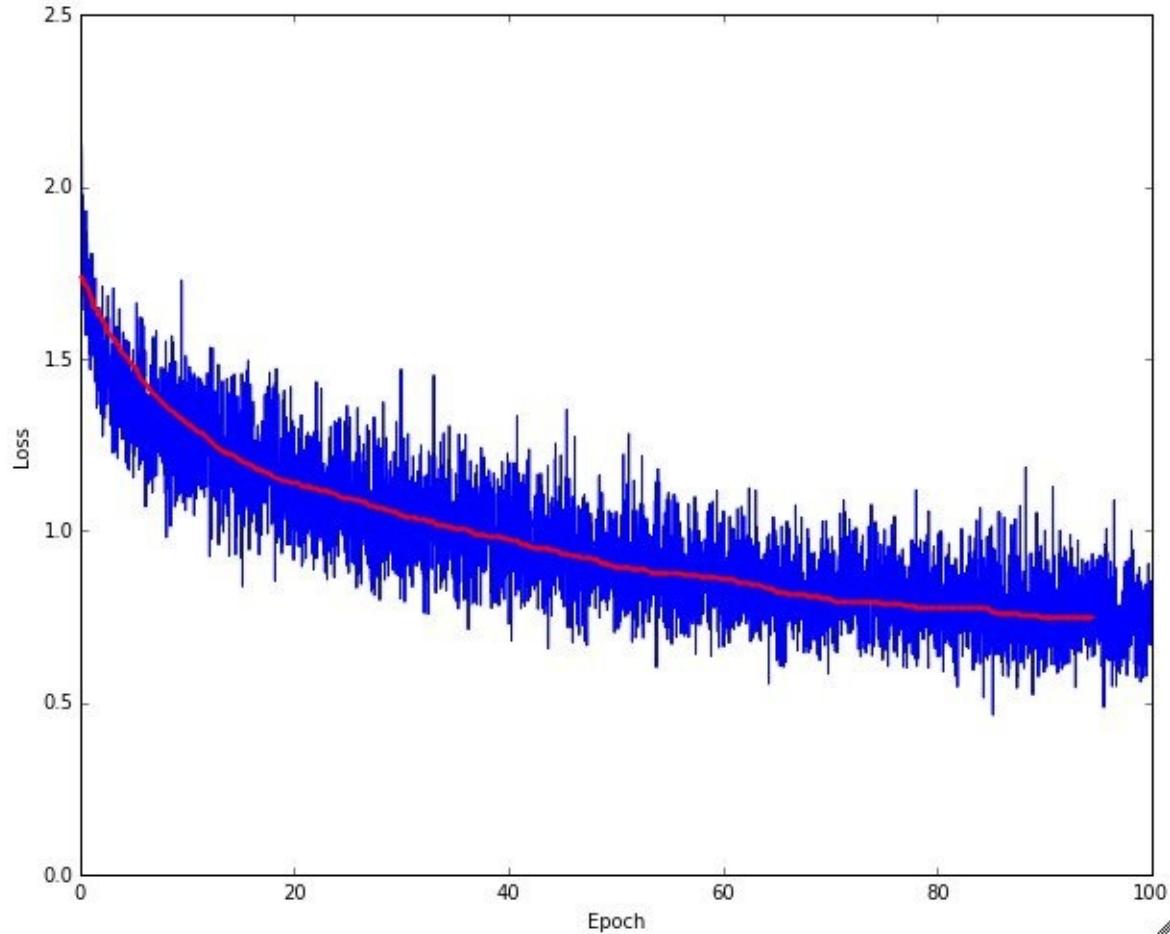
Using a big learning rate can accelerate convergence, but could also make the gradient descent oscillate and diverge.



Numerical Gradient

Slow method to evaluate the gradient, but we can use it to verify if our code is right

Mini Batch Gradient Descent



Instead of running through all the training set to calculate the loss, then do gradient descent, we can do in a small (batch) portions. This leads to similar results and compute faster. Normally the mini-batch size is dependent of the GPU memory.

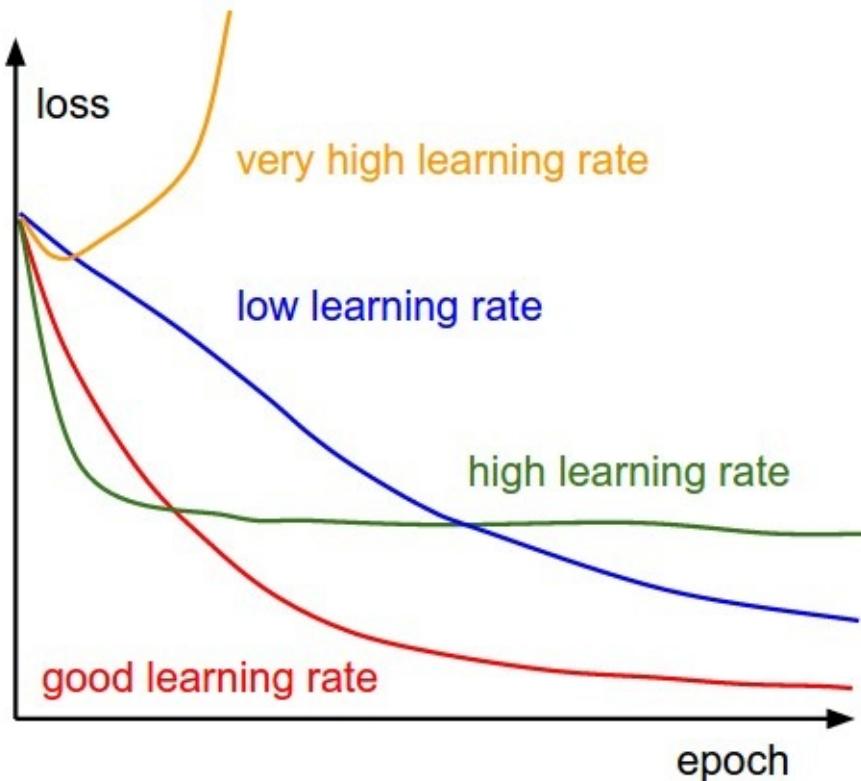
If you analyze your loss decay over time the full-batch version is less noisy but much more difficult to compute.

If your mini-batch size goes to 1, which means you compute the gradient descent for each sample. On this case we have the Stochastic Gradient Descent.

This is relatively less common to see because in practice due to vectorized code optimizations it can be computationally much more efficient to evaluate the gradient for 100 examples, than the gradient for one example 100 times.

Effects of learning rate on loss

Actually what people do is to choose a high (but not so high) learning rate, then decay with time.



Here are 2 ways to decay the learning rate with time while training:

Divide the learning rate (α) by 2 every x epochs

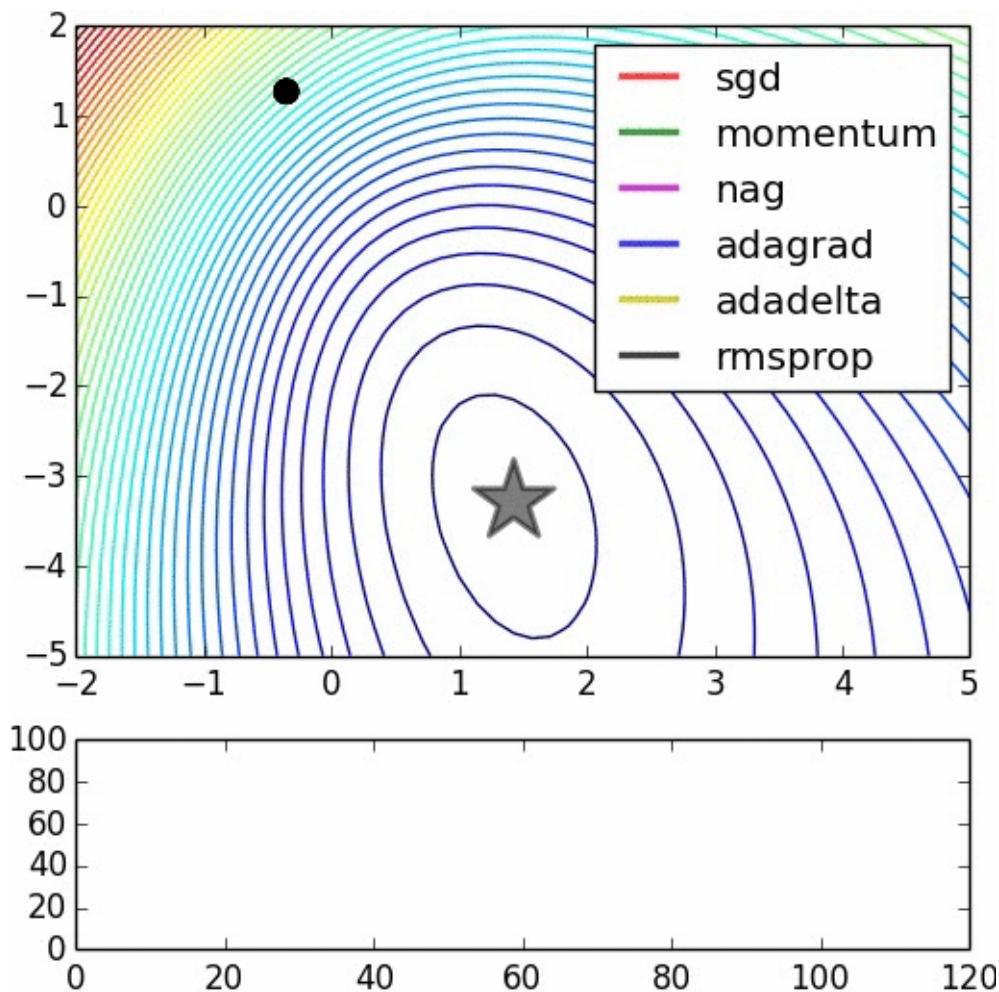
$$\alpha = \alpha_0 \cdot e^{-k \cdot t}, \text{ where } t \text{ is time and } k \text{ is the decay parameter}$$

$$\alpha = \frac{\alpha_0}{1+k \cdot t}, \text{ where } t \text{ is the time and } k \text{ the decay parameter}$$

Other algorithms

The gradient descent is the simplest idea to do model optimization. There are a few other nice algorithms to try when thinking about model optimization:

- Stochastic Gradient descent
- Stochastic Gradient descent with momentum (Very popular)
- Nag
- Adagrad
- Adam (Very good because you need to take less care about learning rate)
- rmsprop



Next Chapter

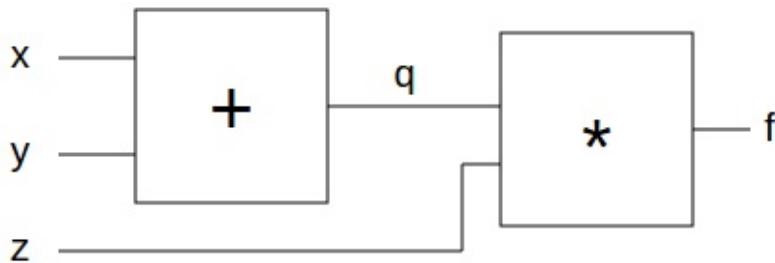
Now what we need is a way to get the gradient vector of our model, next chapter we will talk about back-propagation.

Backpropagation

Back-propagation

Introduction

Backpropagation is an algorithm that calculate the partial derivative of every node on your model (ex: Convnet, Neural network). Those partial derivatives are going to be used during the training phase of your model, where a loss function states how much far you are from the correct result. This error is propagated backward from the model output back to it's first layers. The backpropagation is more easily implemented if you structure your model as a computational graph.

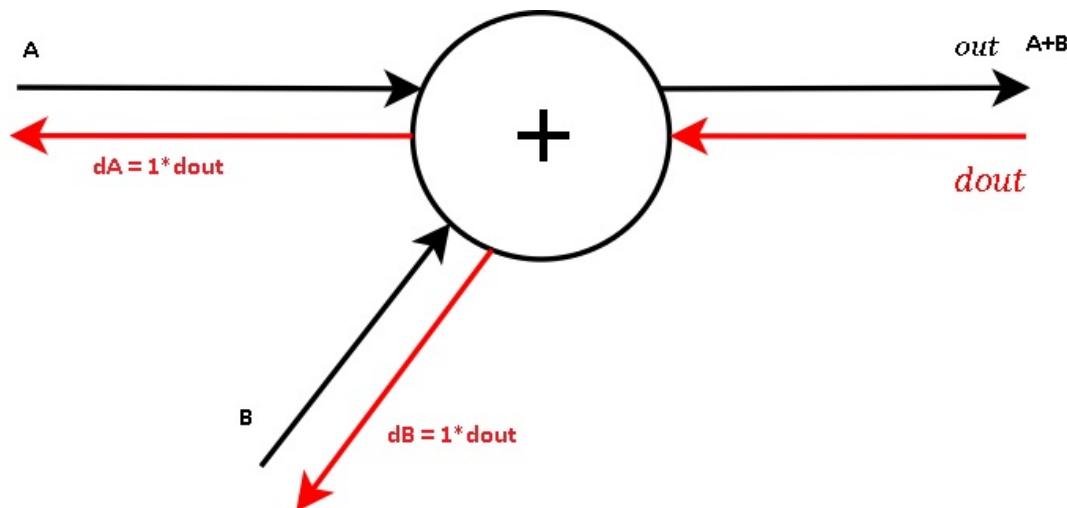


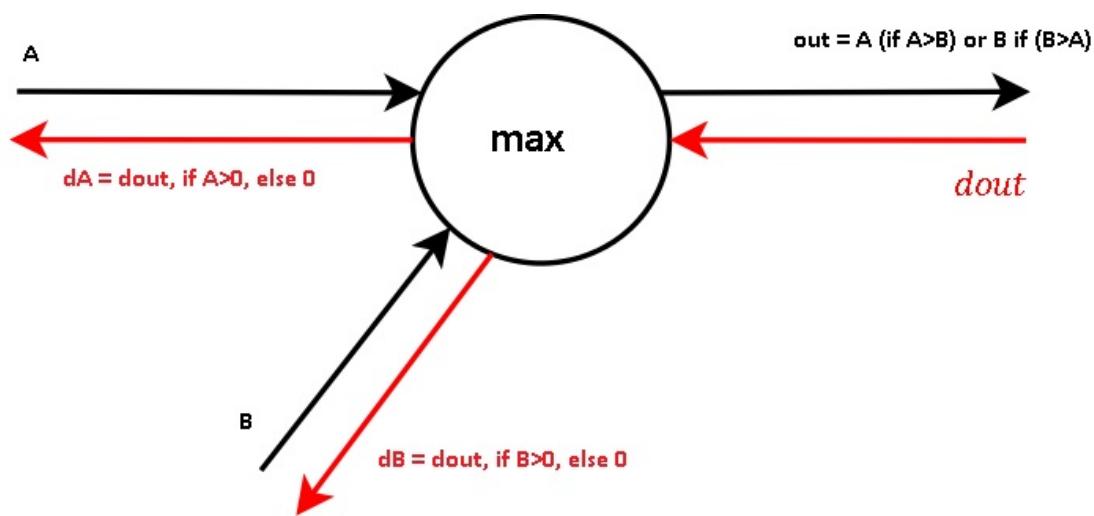
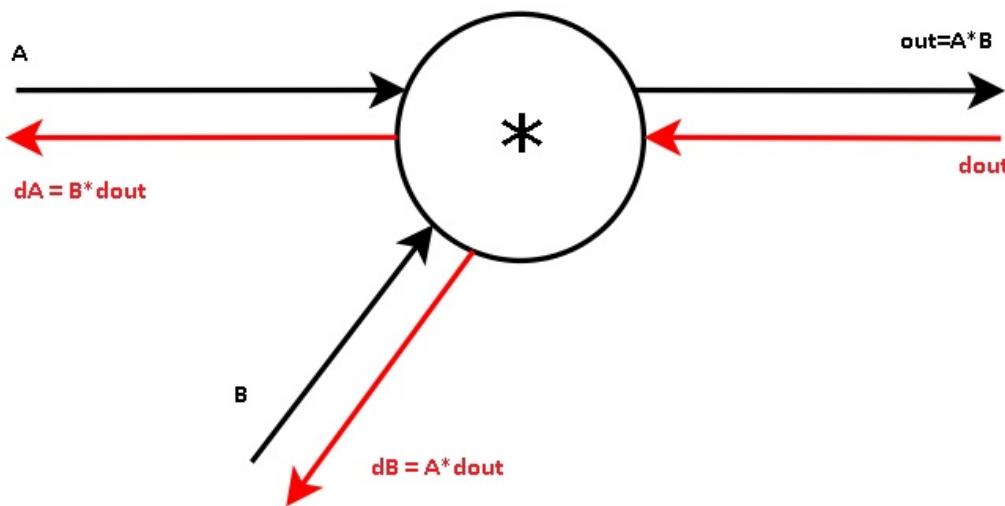
$$f(x, y, z) = (x + y)z.$$

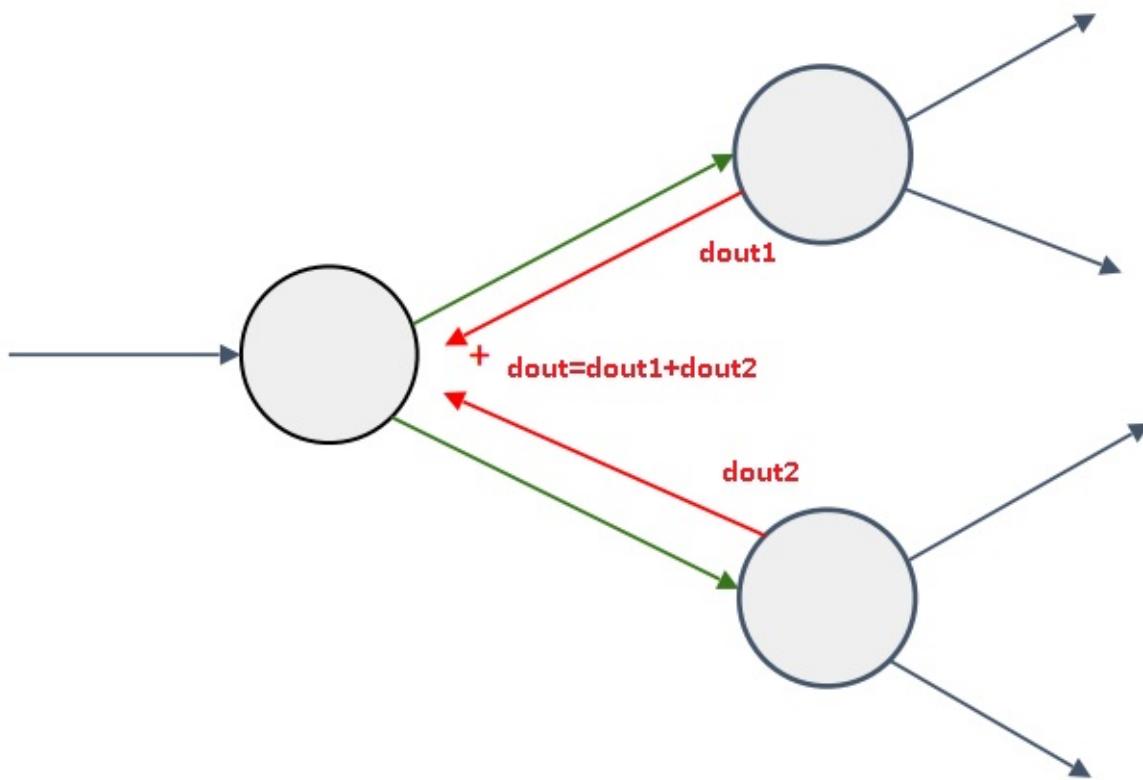
The most important thing to have in mind here is how to calculate the forward propagation of each block and it's gradient. Actually most of the deep learning libraries code is about implementing those gates forward/backward code.

Basic blocks

Some examples of basic blocks are, add, multiply, exp, max. All we need to do is observe their forward and backward calculation







Some other derivatives

$$\begin{aligned} f(x) &= \frac{1}{x^2} \\ f(x) &= 1c + x \\ f(x) &= e^x \\ f(x) &= ax \end{aligned}$$

Observe that we output 2 gradients because we have 2 inputs... Also observe that we need to save (cache) on memory the previous inputs.

Chain Rule

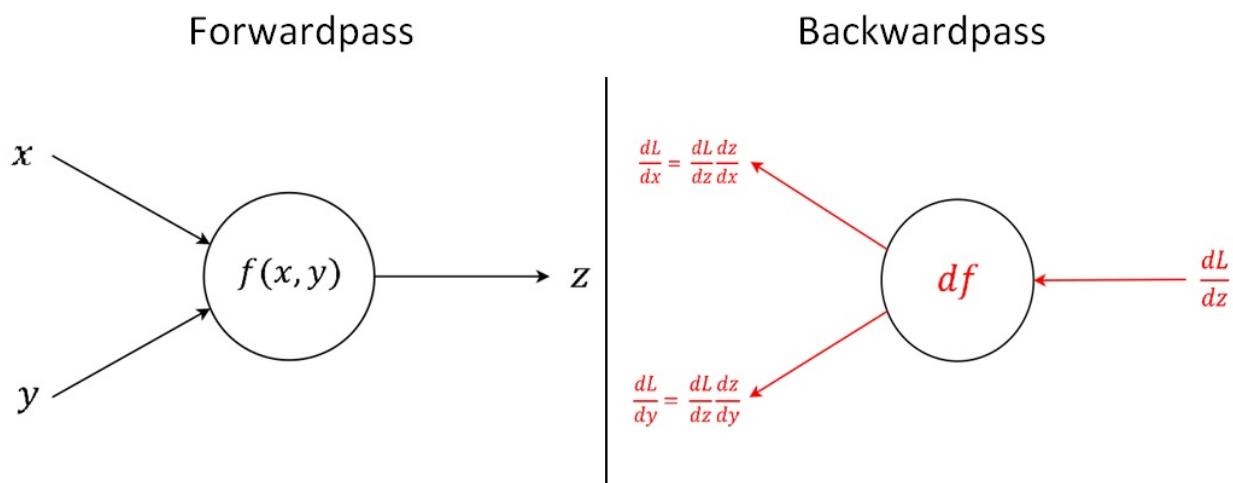
Imagine that you have an output y , that is function of g , which is function of f , which is function of x . If you want to know how much g will change with a small change on dx (dg/dx), we use the chain rule. Chain rule is a formula for computing the derivative of the composition of two or more functions.

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

$$\frac{dg}{dx} = \frac{dg}{df} \cdot \frac{df}{dx}$$



The chain rule is the work horse of back-propagation, so it's important to understand it now. On the picture bellow we get a node $f(x,y)$ that compute some function with two inputs x,y and output z . Now on the right side, we have this same node receiving from somewhere (loss function) a gradient dL/dz which means. "How much L will change with a small change on z ". As the node has 2 inputs it will have 2 gradients. One showing how L will change with a small change dx and the other showing how L will change with a small change (dz)



In order to calculate the gradients we need the input dL/dz (dout), and the derivative of the function $f(x,y)$, at that particular input, then we just multiply them. Also we need the previous cached input, saved during forward propagation.

Gates Implementation

Observe bellow the implementation of the multiply and add gate on python

```

class MultiplyGate(object):
    # Implement Multiply gate
    def forward(self,x,y):
        z = x*y
        self.x = x;
        self.y = y;
        return z

    # Observe that we return a gradient for each input
    def backward(self,dz):
        dx = self.y * dz
        dy = self.x * dz
        return [dx,dy]

class AddGate(object):
    # Implement Add gate
    def forward(self,x,y):
        z = x+y
        self.x = x;
        self.y = y;
        return z

    # Observe that we return a gradient for each input
    def backward(self,dz):
        dx = 1*dz
        dy = 1*dz
        return [dx,dy]

In [2]: import gatesUtils as gates
In [3]: gateMul = gates.MultiplyGate(); gateAdd = gates.AddGate();
In [4]: gateAdd.forward(5,6)
Out[4]: 11

In [5]: gateMul.forward(2,3)
Out[5]: 6

In [6]: gateMul.backward(2)
Out[6]: [6, 4]

In [7]: gateAdd.backward(3)
Out[7]: [3, 3]

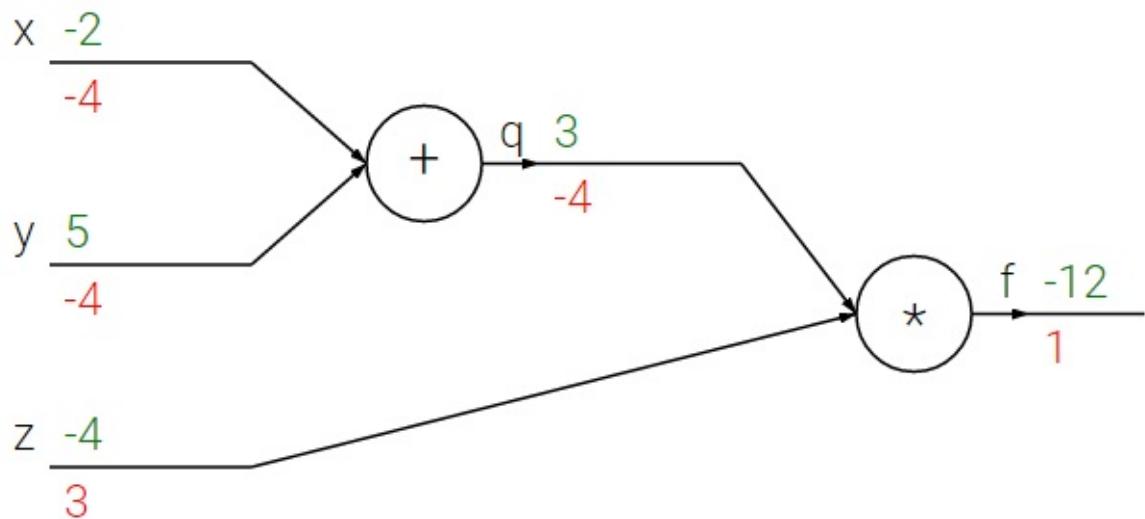
```

Step by step example

With what we learn so far, let's calculate the partial derivatives of some graphs

Simple example

Here we have a graph for the function $f(x, y, z) = (x + y) \cdot z$

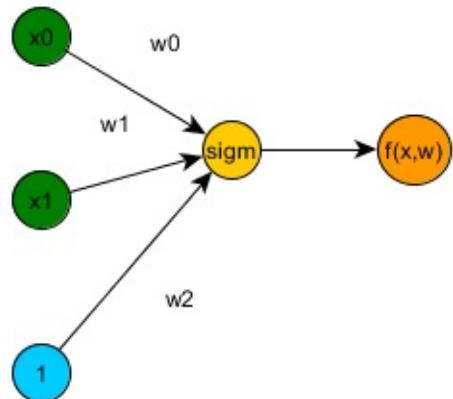


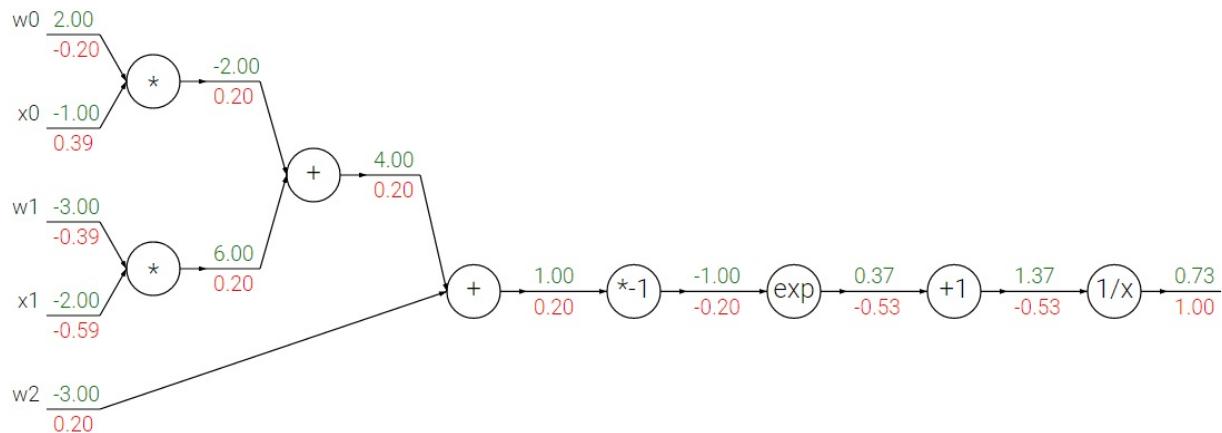
1. Start from output node f , and consider that the gradient of f related to some criteria is 1 ($dout$)
2. $dq = (dout \cdot z)$, which is -4 (How the output will change with a change in q)
3. $dz = (dout \cdot q)$, which is 3 (How the output will change with a change in z)
4. The sum gate distribute its input gradients, so $dx = -4$, $dy = -4$ (How the output will change with x, z)

Perceptron with 2 inputs

This following graph represent the forward propagation of a simple 2 inputs, neural network with one output layer with sigmoid activation.

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$





1. Start from the output node, considering that or error($dout$) is 1
2. The gradient of the input of the $1/x$ will be $-1/(1.37^2)$, -0.53
3. The increment node does not change the gradient on its input, so it will be $(-0.53 * 1)$, -0.53
4. The exp node input gradient will be $(\exp(-1(cached\ input)) * -0.53)$, -0.2
5. The negative gain node will be its input gradient $(-1 * -0.2)$, 0.2
6. The sum node will distribute the gradients, so, $dw_2=0.2$, and the sum node also 0.2
7. The sum node again distribute the gradients so again 0.2
8. dw_0 will be $(0.2 * -1)$, -0.2
9. dx_0 will be $(0.2 * 2)$. 0.4

Next Chapter

Next chapter we will learn about Feature Scaling.

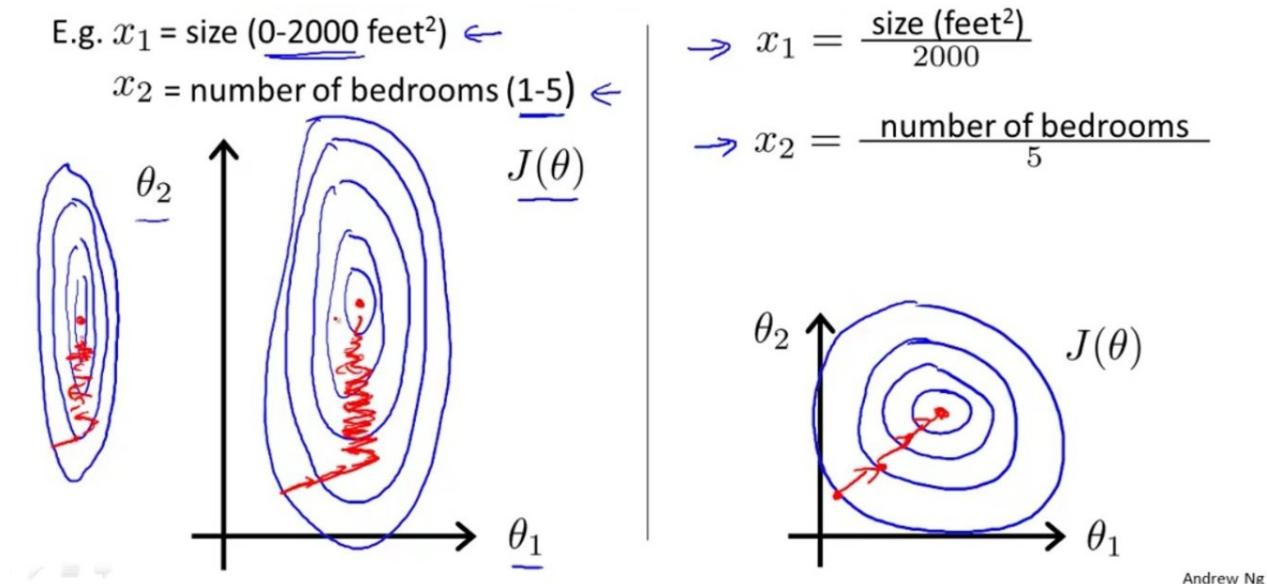
Feature Scaling

Feature Scaling

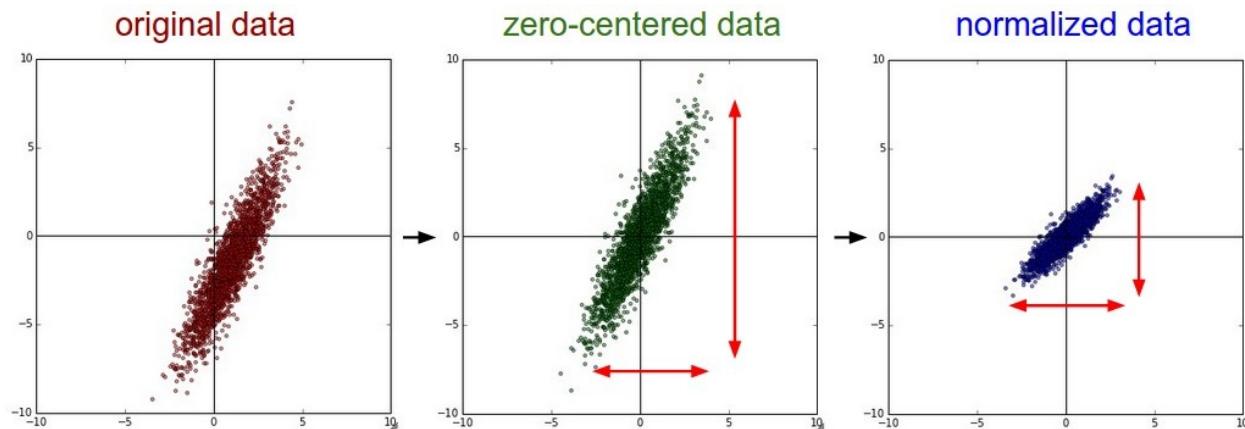
Introduction

In order to make the life of gradient descent algorithms easier, there are some techniques that can be applied to your data on training/test phase. If the features $x_1, x_2, x_3 \dots x_n$ on your input vector $X = [x_1, x_2, x_3 \dots x_n]$, are out of scale your loss space $J(\theta) = [\theta_1, \theta_2]$ will be somehow stretched. This will make the gradient descent convergence harder, or at least slower.

On the example below your input X has 2 features (house size, and number of bedrooms). The problem is that house size feature range from 0...2000, while number of bedrooms range from 0...5.



Centralize data and normalize



Below we will pre-process our input data to fix the following problems:

- Data not centered around zero
- Features out of scale

Consider your input data $X = [NxD]$, where N is the number of samples on your input data (batch size) and D the dimensions (On the previous example D is 2, size house, num bedrooms).

The first thing to do is to subtract the mean value of the input data, this will centralize the data dispersion around zero

$$X = X - np.\text{mean}(X, \text{axis} = 0)$$

On prediction phase is common to store this mean value to be subtracted from a test example. On the case of image classification, it's also common to store a mean image created from a batch of images on the training-set, or the mean value from every channel.

After your data is centralized around zero, you can make all features have the same range by dividing X by its standard deviation.

$$X = X / np.\text{std}(X, \text{axis} = 0)$$

Again this operation fix our first, problem, because all features will range similarly. But this should be used if somehow you know that your features have the same "weight". On the case of image classification for example all pixels have the same range (0..255) and a pixel alone has no bigger meaning (weight) than the other, so just mean subtraction should suffice for image classification.

Common mistake:

An important point to make about the preprocessing is that any preprocessing statistics (e.g. the data mean) must only be computed on the training data, and then applied to the validation / test data. Computing the mean and subtracting it from every image across the entire dataset and then splitting the data into train/val/test splits would be a mistake. Instead, the mean must be computed only over the training data and then subtracted equally from all splits (train/val/test).

Next Chapter

Next chapter we will learn about Neural Networks.

Model Initialization

Model Initialization

Introduction

One important topic we should learn before we start training our models, is about the weight initialization. Bad weight initialization, can lead to a "never convergence training" or a slow training.

Weight matrix format

As observed on previous chapters, the weight matrix has the following format:

$$\underbrace{\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix}}_{\text{One column per x dimension}} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \therefore$$

One column per x dimension

$$H(X) = (W \cdot x) + b^T$$

$$H(X) = (W^T \cdot x) + b$$

Consider the number of outputs fan_{out} , as rows and the number of inputs fan_{in} as columns. You could also consider another format:

$$([x_1 \quad x_2 \quad x_3] \cdot \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix}) + [b_1 \quad b_2] = [y_1 \quad y_2] \therefore$$

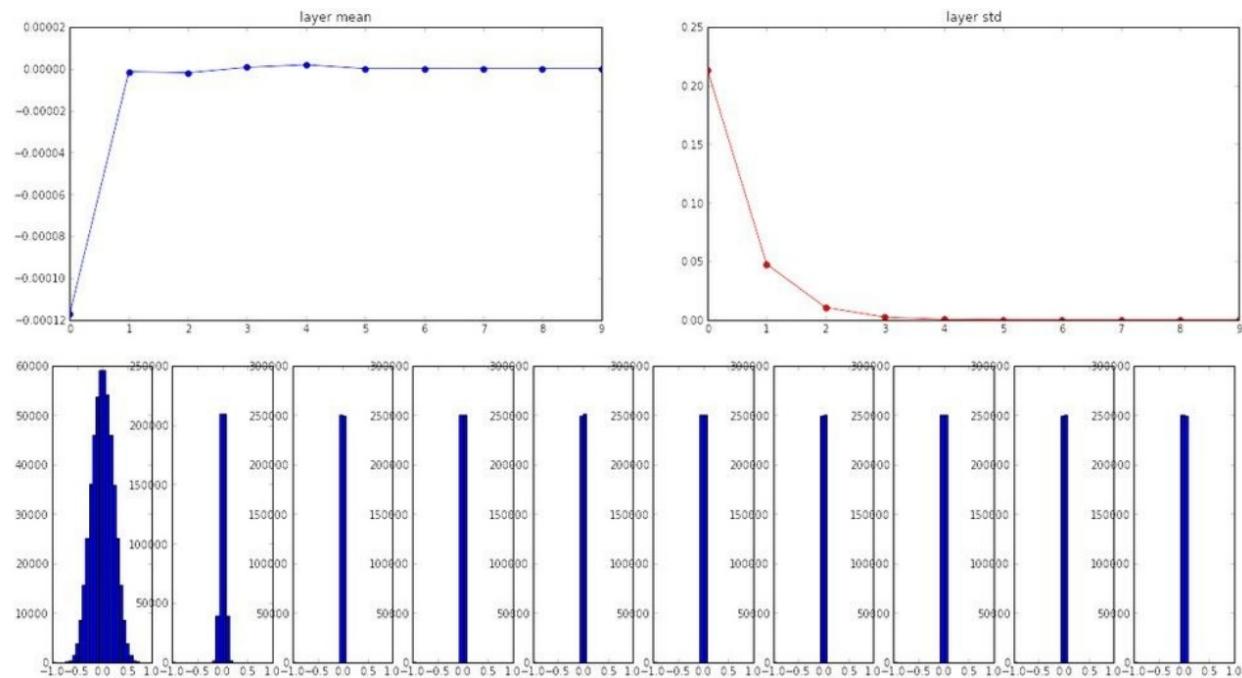
$$H(X) = (x \cdot W^t) + b$$

Here fan_{out} as columns and fan_{in} as rows.

The whole point is that our weights is going to be a 2d matrix function of fan_{in} and fan_{out}

Initialize all to zero

If you initialize your weights to zero, your gradient descent will never converge

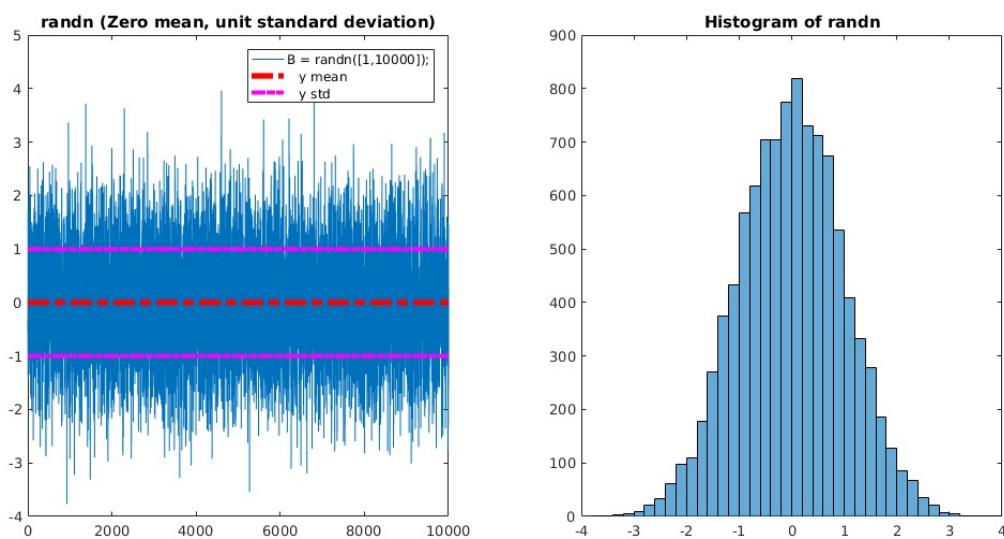


Initialize with small values

A better idea is to initialize your weights with values close to zero (but not zero), ie: 0.01

$$W = 0.01 * np.random.randn(fan_{in}, fan_{out})$$

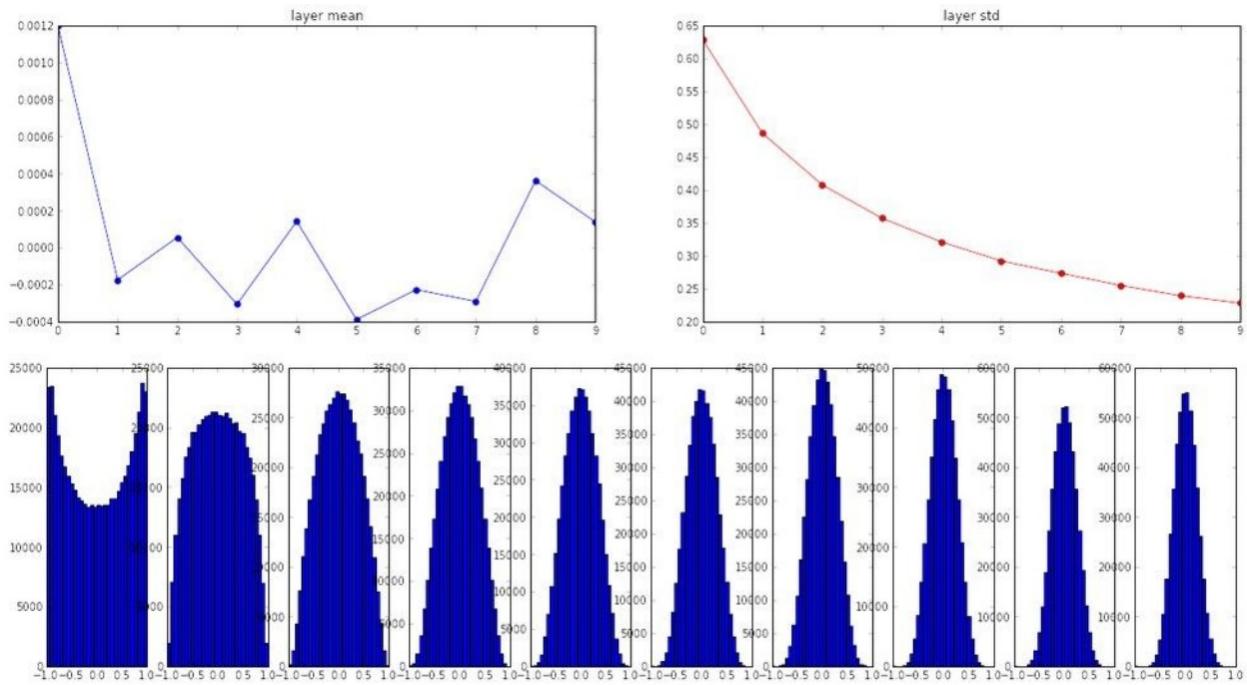
Here randn gives random data with zero mean, unit standard deviation. fan_{in} , fan_{out} are the number of input and outputs. The 0.01 term will keep the random weights small and close to zero.



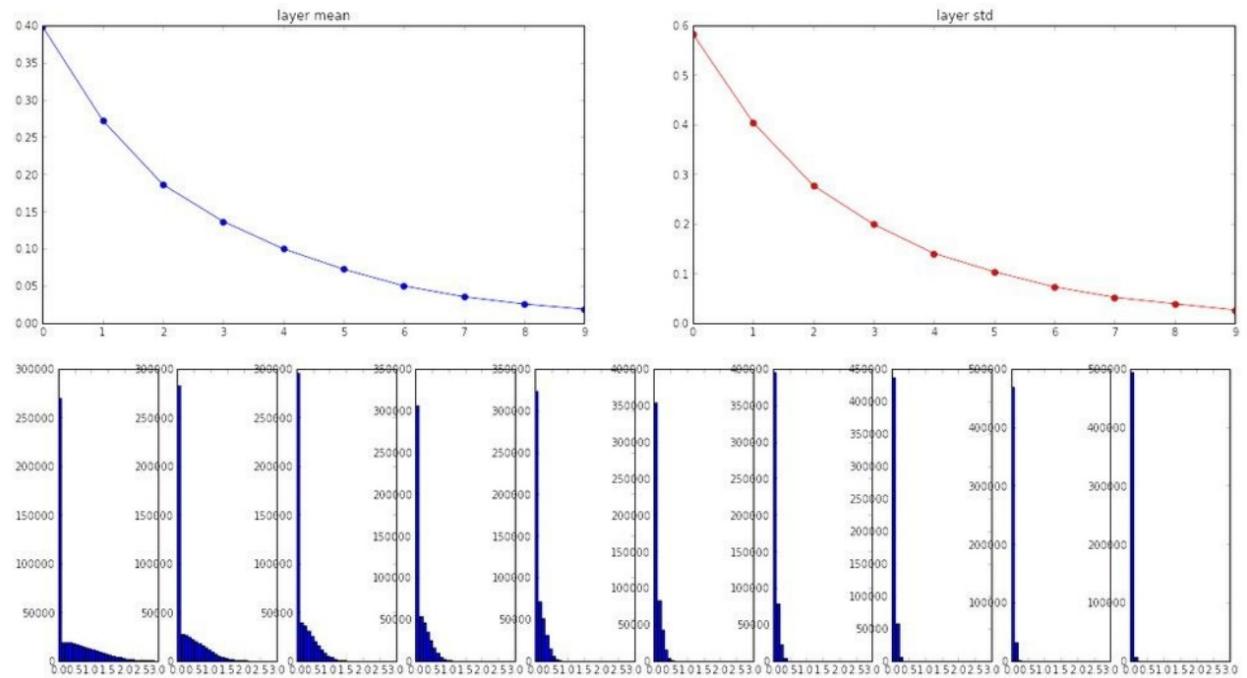
The problem with the previous way to do initialization is that the variance of the outputs will grow with the number of inputs. To solve this issue we can divide the random term by the square root of the number of inputs.

$$W = (np.random.randn(fan_{in}, fan_{out})) / np.sqrt(fan_{in})$$

Model Initialization

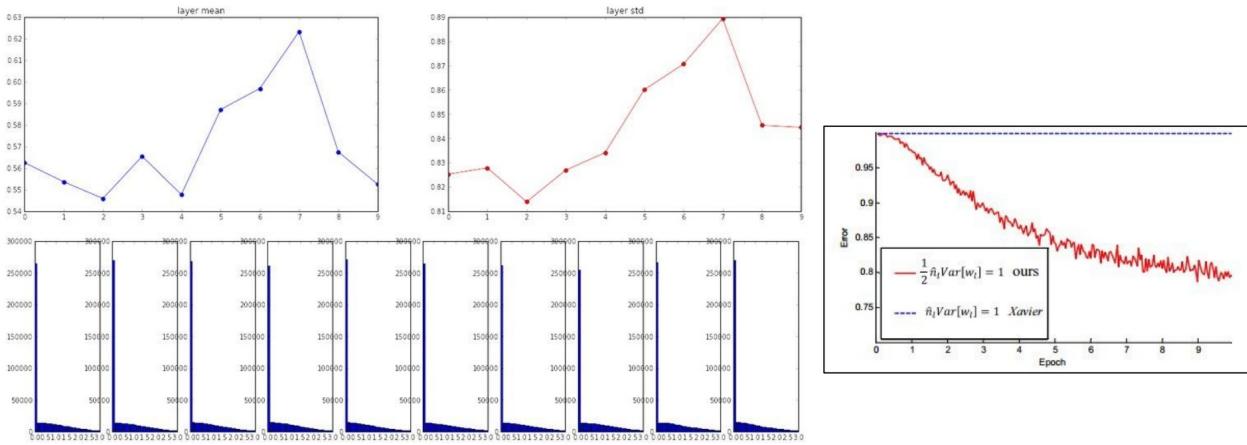


Now it seems that we don't have dead neurons, the only problem with this approach is to use it with Relu neurons.



To solve this just add a simple (divide by 2) term....

$$W = (\text{np.random.randn}(fan_{in}, fan_{out})) / \text{np.sqrt}(fan_{in}/2)$$



So use this second form to initialize Relu layers.

Bath Norm layer

On future chapters we're going to learn a technique that make your model more resilient to specific initialization.

Next chapter

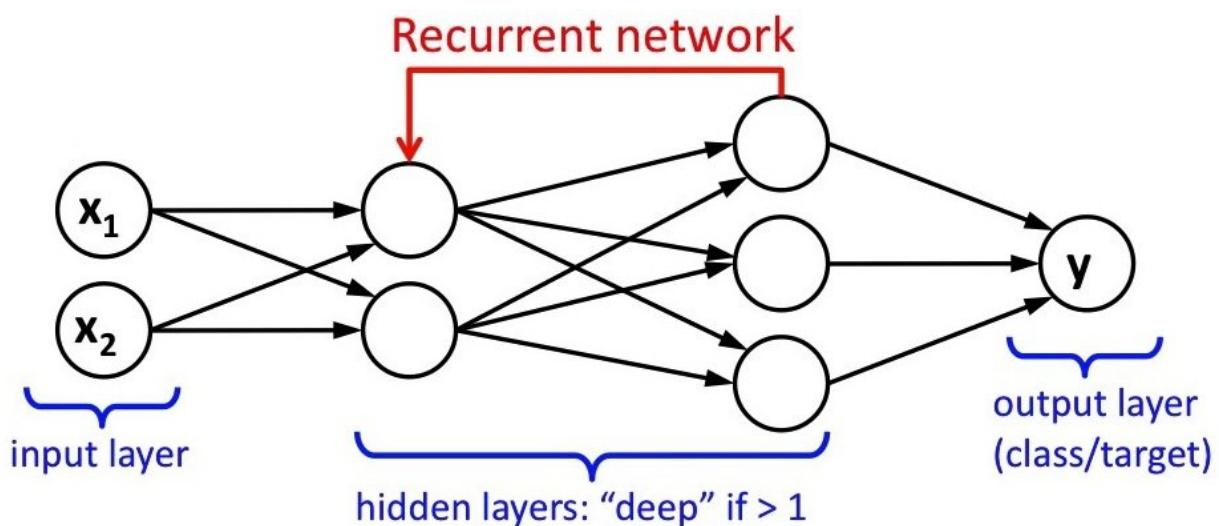
Next chapter we start talk about convolutions.

Recurrent Neural Networks

Recurrent Neural Networks

Introduction

On previous forward neural networks, our output was a function between the current input and a set of weights. On recurrent neural networks(RNN), the previous network state is also influence the output, so recurrent neural networks also have a "notion of time". This effect by a loop on the layer output to it's input.



In other words, the RNN will be a function with inputs x_t (input vector) and previous state h_{t-1} .

The new state will be h_t .

The recurrent function, f_W , will be fixed after training and used to every time step.

Recurrent Neural Networks are the best model for regression, because it take into account past values.

RNN are computation "Turing Machines" which means, with the correct set of weights it can compute anything, imagine this weights as a program.

Just to not let you too overconfident on RNN, there is no automatic back-propagation algorithms, that will find this "perfect set of weights".

Bellow we have a simple implementation of RNN recurrent function: (Vanilla version)

$$\begin{aligned}
 h_t &= f_{weights}(h_{t-1}, x_t) \therefore \\
 h_t &= \tanh(W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t) \\
 y_t &= W_{hy} \cdot h_t
 \end{aligned}$$

The code that calculate up to the next state h_t looks like this:

```

def rnn_step_forward(x, prev_h, Wx, Wh, b):
    # We separate on steps to make the backpropagation easier
    # forward pass in steps
    # step 1
    xWx = np.dot(x, Wx)

    # step 2
    phWh = np.dot(prev_h, Wh)

    # step 3
    # total
    affine = xWx + phWh + b.T

    # step 4
    next_h = np.tanh(t)

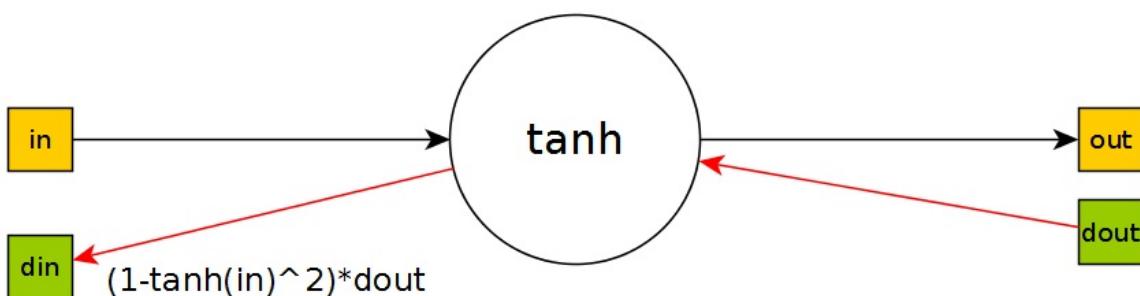
    # Cache inputs, state, and weights
    # we are having prev_h.copy() since python params are pass by reference
    cache = (x, prev_h.copy(), Wx, Wh, next_h, affine)

    return next_h, cache

```

Observe that in our case of RNN we are now more interested on the next state, h_t not exactly the output, y_t

Before we start let's just make explicit how to backpropagate the tanh block.



Now we can do the backpropagation step (For one single time-step)

```

def rnn_step_backward(dnext_h, cache):
    (x, prev_h, Wx, Wh, next_h, affine) = cache

    #backward in step
    # step 4
    # dt delta of total
    # Gradient of tanh times dnext_h
    dt = (1 - np.square(np.tanh(affine))) * (dnext_h)

    # step 3
    # Gradient of sum block
    dxWx = dt

```

```

dphwh = dt
db = np.sum(dt, axis=0)

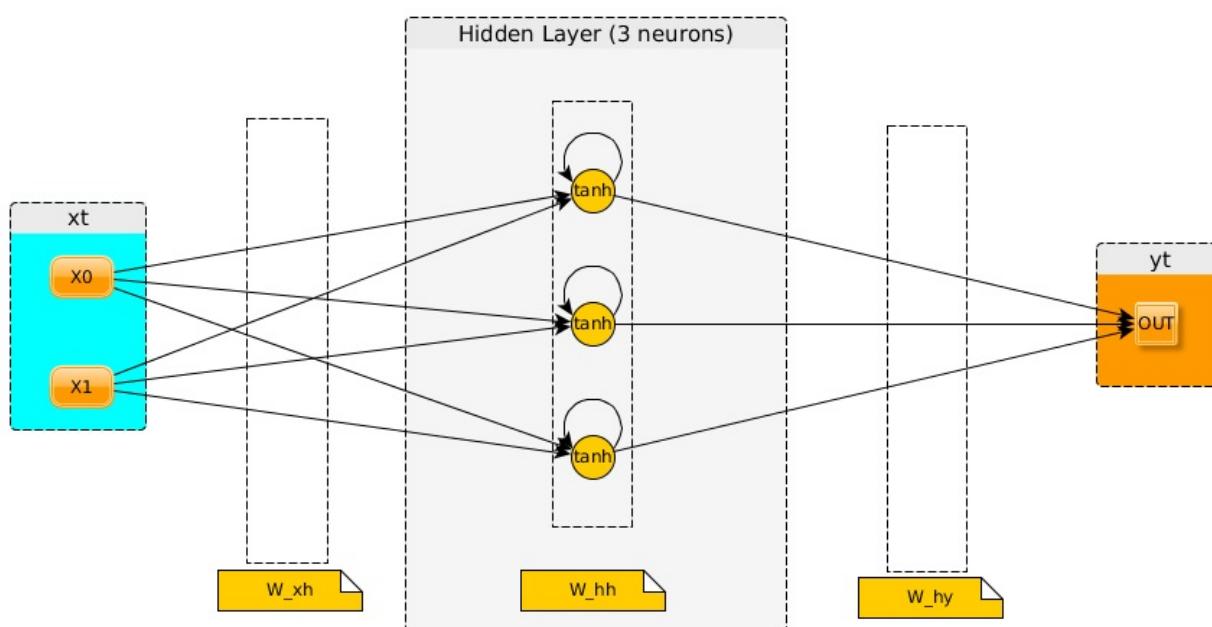
# step 2
# Gradient of the mul block
dwh = prev_h.T.dot(dphwh)
dprev_h = Wh.dot(dphwh.T).T

# step 1
# Gradient of the mul block
dx = dxWx.dot(Wx.T)
dwx = x.T.dot(dxWx)

return dx, dprev_h, dwx, dwh, db

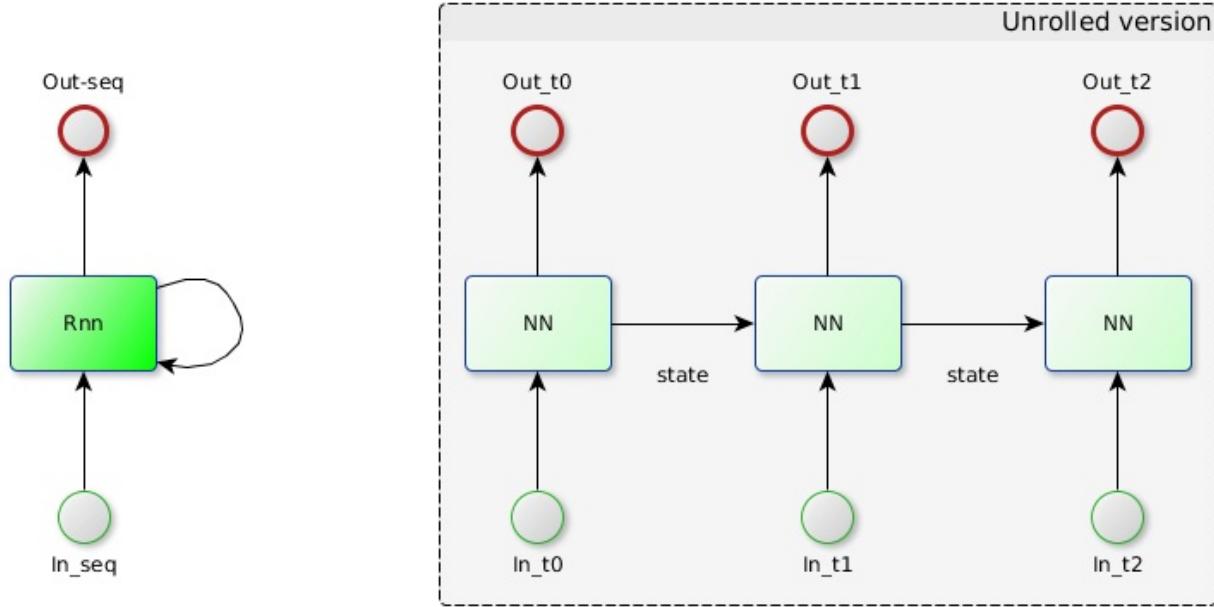
```

A point to be noted is that the same function $f_{weights}$ and the same set of parameters will be applied to every time-step.



A good initialization for the RNN states h_t is zero. Again this is just the initial RNN state not it's weights.

These looping feature on RNNs can be confusing first but actually you can think as a normal neural network repeated(unrolled) multiple times. The number of times that you unroll can be consider how far in the past the network will remember. In other words each time is a time-step.



Forward and backward propagation on each time-step

From the previous examples we presented code for forward and backpropagation for one time-step only. As presented before the RNN are unrolled for each time-step (finite). Now we present how to do the forward propagation for each time-step.

```
def rnn_forward(x, h0, Wx, Wh, b):
    """
    Run a vanilla RNN forward on an entire sequence of data. We assume
    sequence composed of T vectors, each of dimension D. The RNN uses a
    size of H, and we work over a minibatch containing N sequences. After
    the RNN forward, we return the hidden states for all timesteps.
    """

    Run a vanilla RNN forward on an entire sequence of data. We assume
    sequence composed of T vectors, each of dimension D. The RNN uses a
    size of H, and we work over a minibatch containing N sequences. After
    the RNN forward, we return the hidden states for all timesteps.
```

Inputs:

- `x`: Input data for the entire timeseries, of shape (N, T, D) .
- `h0`: Initial hidden state, of shape (N, H)
- `Wx`: Weight matrix for input-to-hidden connections, of shape (D, H)
- `Wh`: Weight matrix for hidden-to-hidden connections, of shape (H, H)
- `b`: Biases of shape $(H,)$

Returns a tuple of:

- `h`: Hidden states for the entire timeseries, of shape (N, T, H) .
- `cache`: Values needed in the backward pass

```
# Get shapes
N, T, D = x.shape
# Initialization
h, cache = None, None
H = h0.shape[1]
h = np.zeros((N,T,H))
```

```
# keeping the initial value in the last element
# it will be overwritten
```

```

h[:, -1, :] = h0
cache = []

# For each time-step
for t in xrange(T):
    h[:, t, :], cache_step = rnn_step_forward(x[:, t, :], h[:, t-1, :], Wx,
                                                cache.append(cache_step)

# Return current state and cache
return h, cache

def rnn_backward(dh, cache):
    """
    Compute the backward pass for a vanilla RNN over an entire sequence.

    Inputs:
    - dh: Upstream gradients of all hidden states, of shape (N, T, H)

    Returns a tuple of:
    - dx: Gradient of inputs, of shape (N, T, D)
    - dh0: Gradient of initial hidden state, of shape (N, H)
    - dWx: Gradient of input-to-hidden weights, of shape (D, H)
    - dWh: Gradient of hidden-to-hidden weights, of shape (H, H)
    - db: Gradient of biases, of shape (H, )
    """
    dx, dh0, dWx, dWh, db = None, None, None, None, None
    # Get shapes
    N, T, H = dh.shape
    D = cache[0][0].shape[1] # D taken from x in cache

    # Initialization keeping the gradients with the same shape it's received
    dx, dprev_h = np.zeros((N, T, D)), np.zeros((N, H))
    dWx, dWh, db = np.zeros((D, H)), np.zeros((H, H)), np.zeros((H,))
    dh = dh.copy()

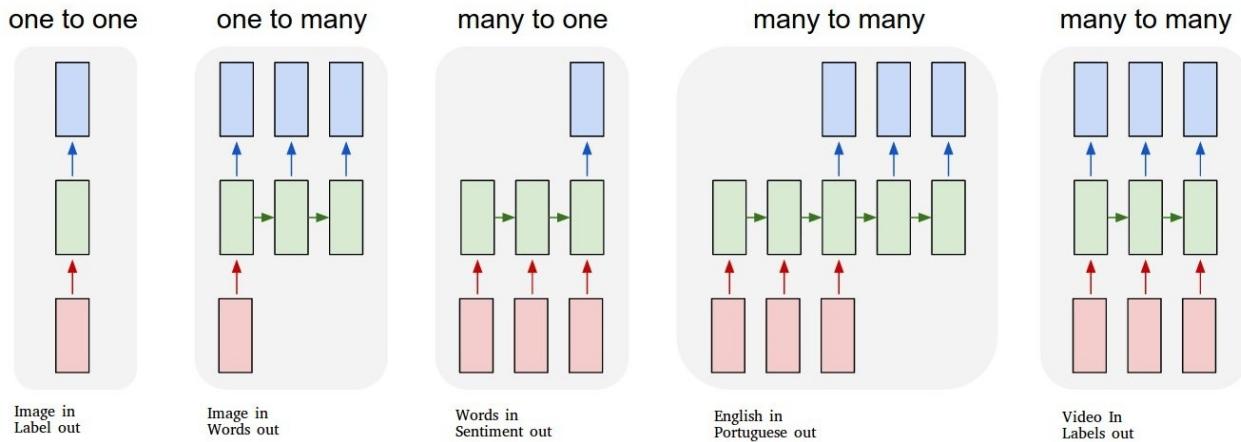
    # For each time-step
    for t in reversed(xrange(T)):
        dh[:, t, :] += dprev_h # updating the previous layer dh
        dx_, dprev_h, dWx_, dWh_, db_ = rnn_step_backward(dh[:, t, :], cache[t])
        # Observe that we sum each time-step gradient
        dx[:, t, :] += dx_
        dWx += dWx_
        dWh += dWh_
        db += db_

    dh0 = dprev_h

    return dx, dh0, dWx, dWh, db

```

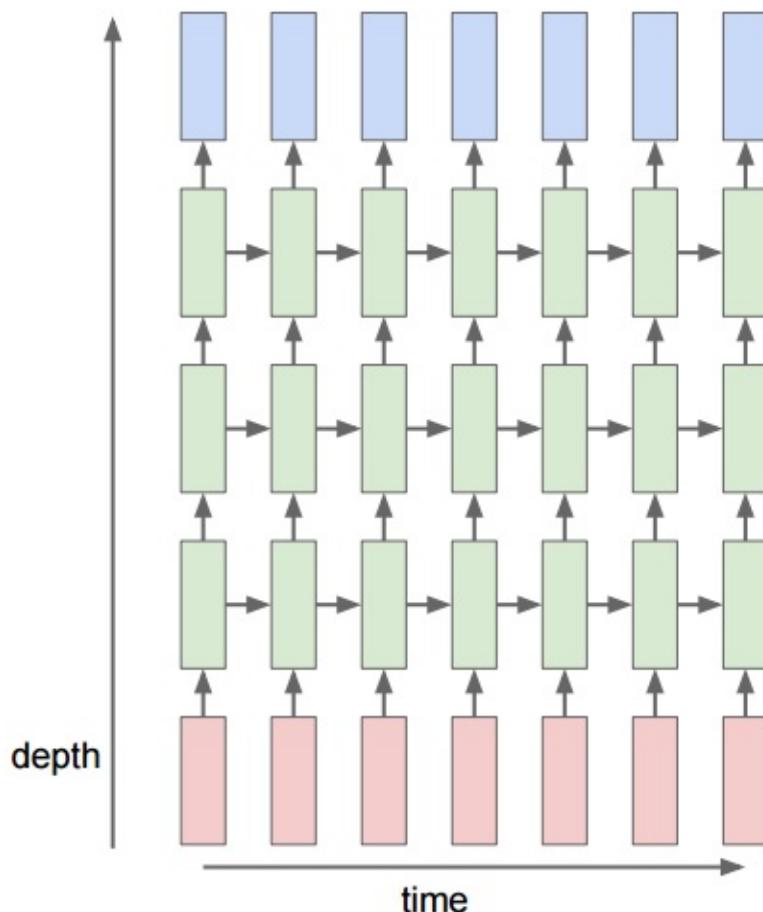
Bellow we show a diagram that present the multiple ways that you could use a recurrent neural network compared to the forward networks. Consider the inputs the red blocks, and the outputs the blue blocks.



- One to one: Normal Forward network, ie: Image on the input, label on the output
- One to many(RNN): (Image captioning) Image in, words describing the scene out (CNN regions detected + RNN)
- Many to one(RNN): (Sentiment Analysis) Words on a phrase on the input, sentiment on the output (Good/Bad) product.
- Many to many(RNN): (Translation), Words on English phrase on input, Portuguese on output.
- Many to many(RNN): (Video Classification) Video in, description of video on output.

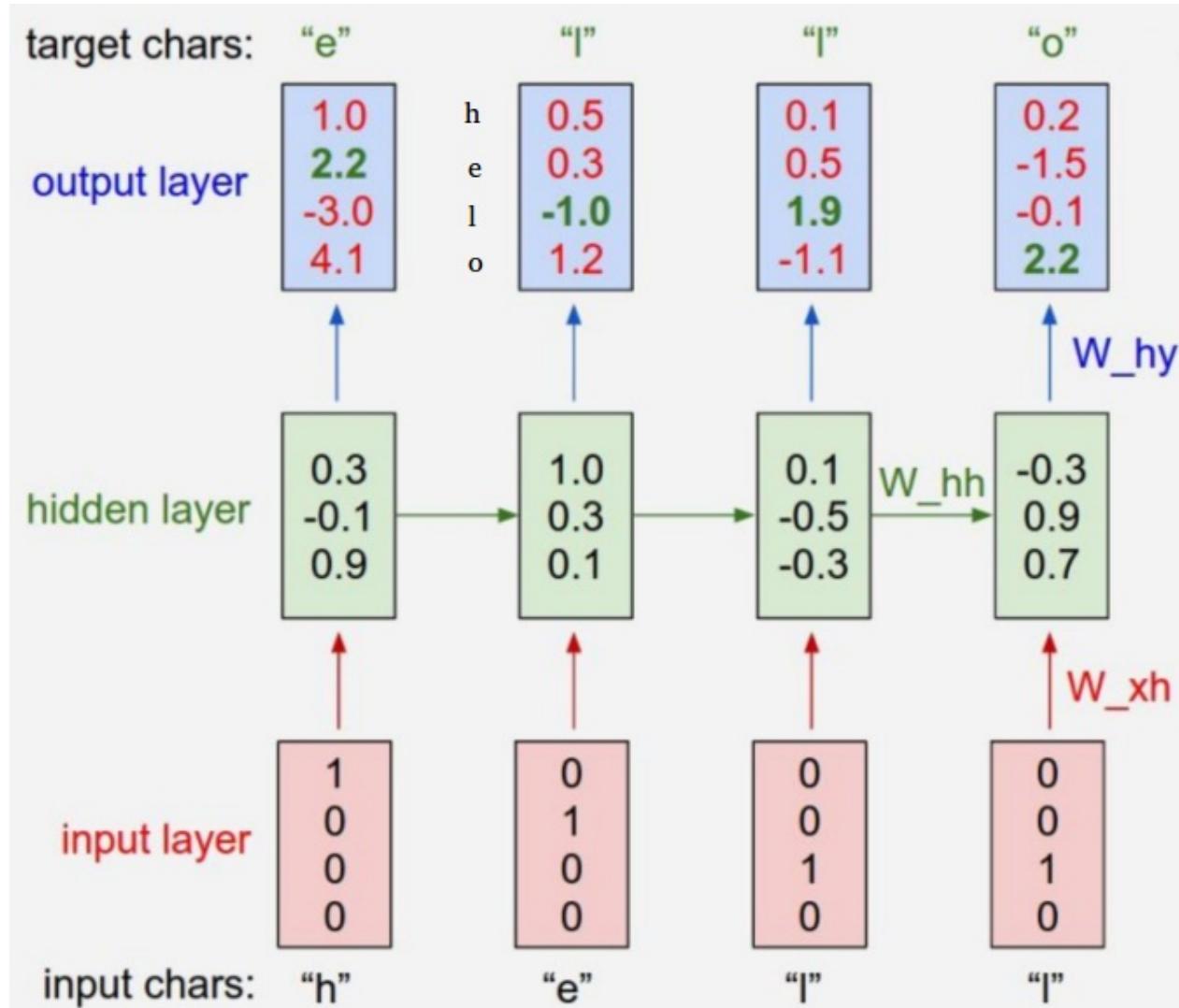
Stacking RNNs

Bellow we describe how we add "depth" to RNN and also how to unroll RNNs to deal with time. Observe that the output of the RNNs are feed to deeper layers, while the state is feed for dealing with past states.



Simple regression example

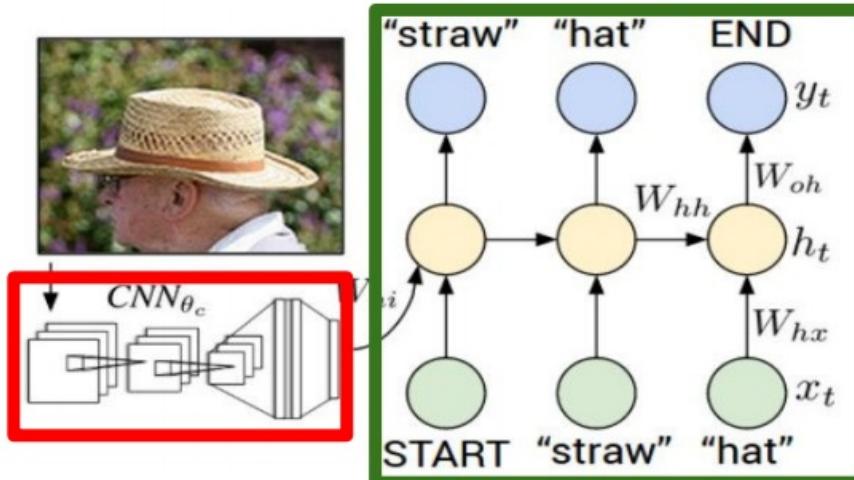
Here we present a simple case where we want the RNN to complete the word, we give to the network the characters h,e,l,l , our vocabulary here is [h,e,l,o]. Observe that after we input the first 'h' the network want's to output the wrong answer (right is on green), but near the end, after the second 'l' it want's to output the right answer 'o'. Here the order that the characters come in does matter.



Describing images

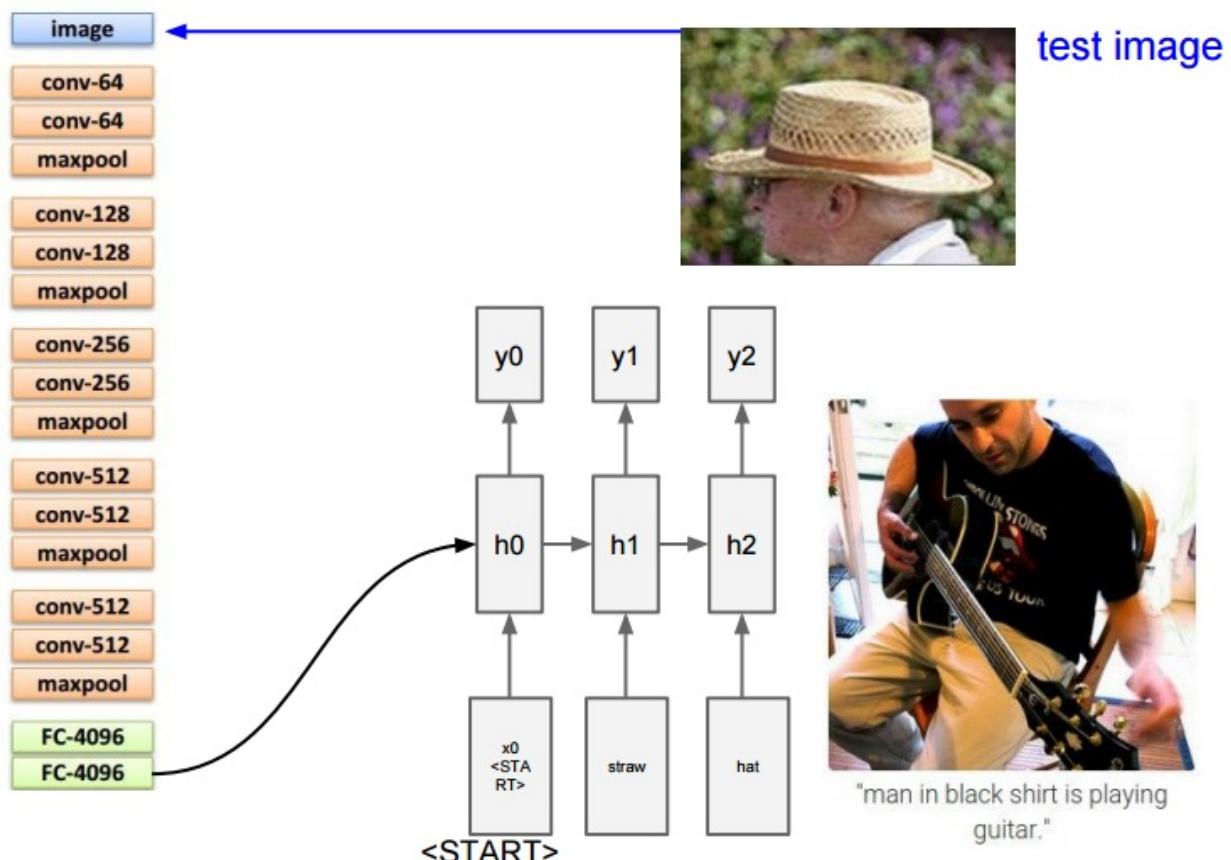
If you connect a convolution neural network, with pre-trained RNN. The RNN will be able to describe what it "see" on the image.

Recurrent Neural Network



Convolutional Neural Network

Basically we get a pre-trained CNN (ie: VGG) and connect the second-to-last FC layer and connect to a RNN. After this you train the whole thing end-to-end.



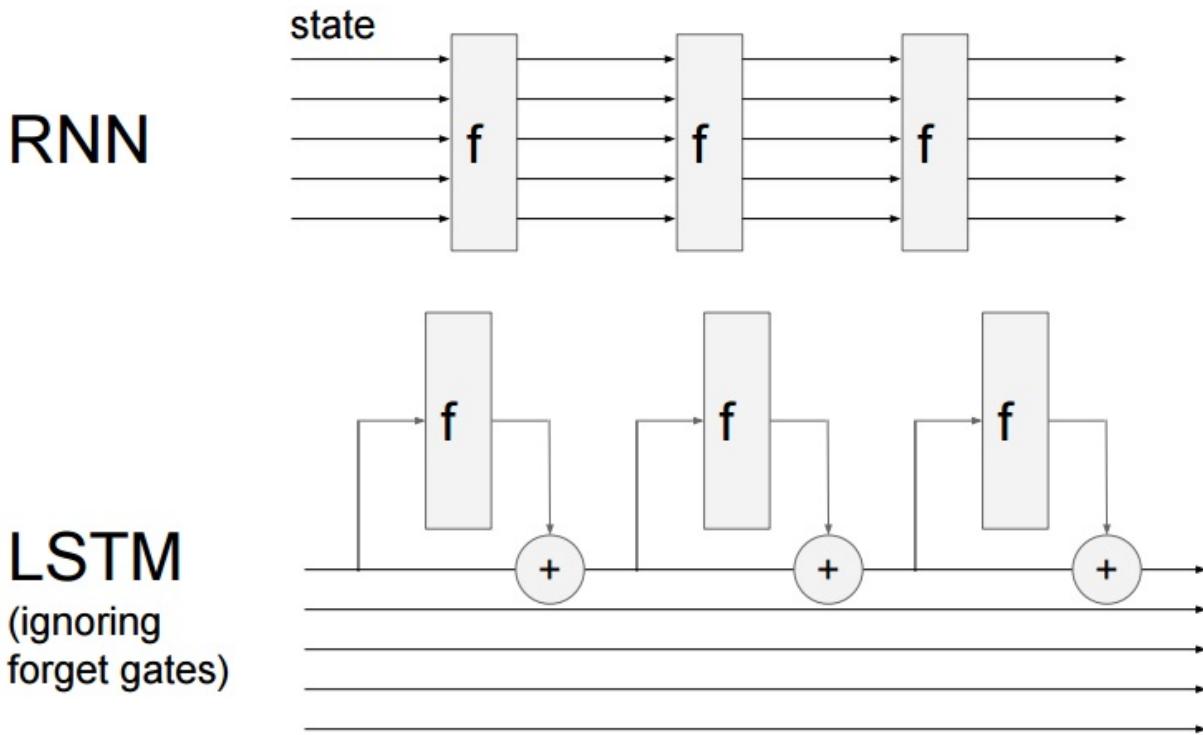
Long Short Term Memory networks(LSTM)

LSTM provides a different recurrent formula f_W , it's more powerful than vanilla RNN, due to it's complex f_W that add "residual information" to the next state instead of just transforming each state.

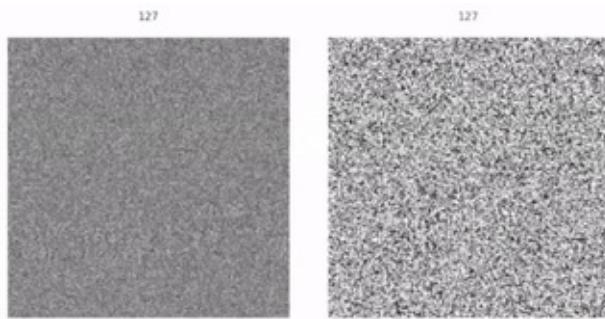
Imagine LSTM are the "residual" version of RNNs.

In other words LSTM suffer much less from vanishing gradients than normal RNNs. Remember that the plus gates distribute the gradients.

So by suffering less from vanishing gradients, the LSTMs can remember much more in the past. So from now just use LSTMs when you think about RNN.



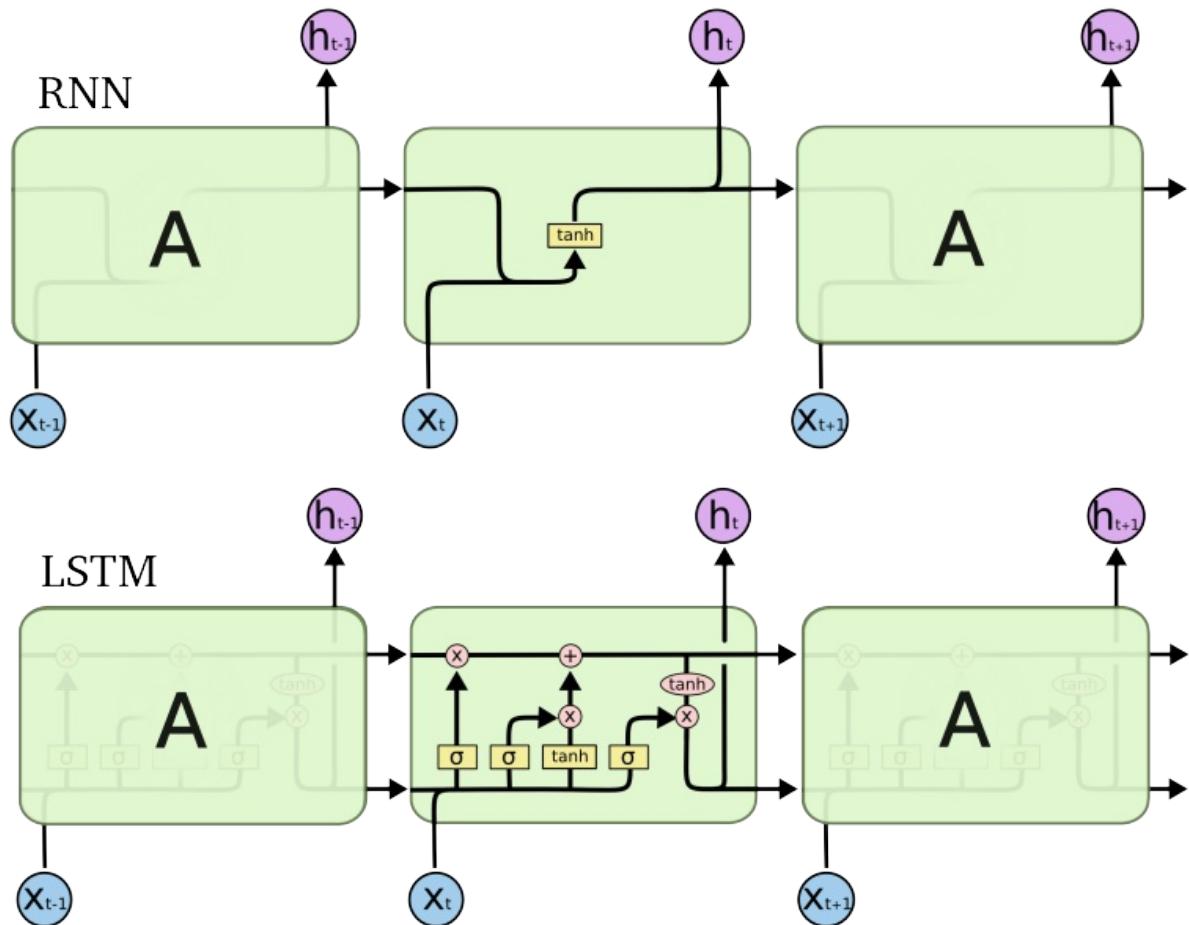
Observe from the animation bellow how fast the gradients on the RNN disappear compared to LSTM.



The vanishing problem can be solved with LSTM, but another problem that can happen with all recurrent neural network is the exploding gradient problem.

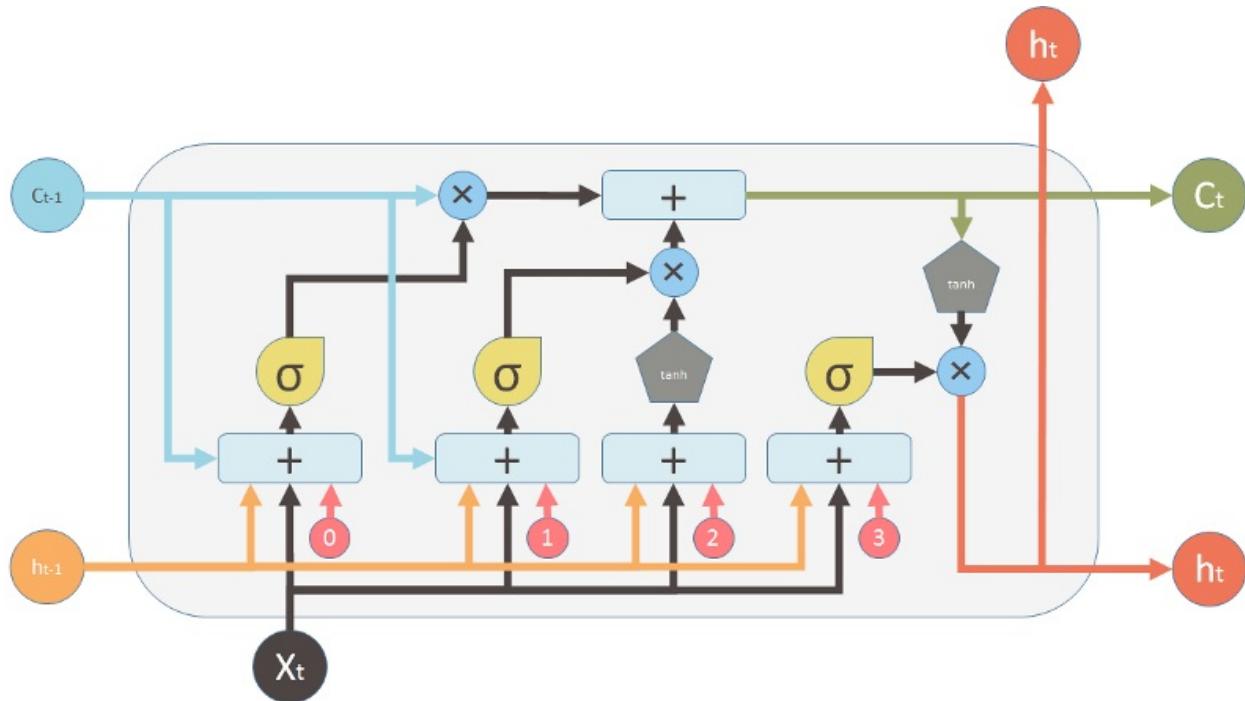
To fix the exploding gradient problem, people normally do a gradient clipping, that will allow only a maximum gradient value.

This highway for the gradients is called Cell-State, so one difference compared to the RNN that has only the state flowing, on LSTM we have states and the cell state.



LSTM Gate

Doing a zoom on the LSTM gate. This also improves how to do the backpropagation.



Inputs:	outputs:	Nonlinearities:	Vector operations:
X_t	Input vector	σ	Element-wise multiplication
C_{t-1}	Memory from previous block	σ	Element-wise Summation / Concatenation
h_{t-1}	Output of previous block	tanh	
	Bias: 0		

Code for lstm forward propagation for one time-step

```
def lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b):
    N,H = prev_c.shape

    #forward pass in steps
    # step 1: calculate activation vector
    a = np.dot(x, Wx) + np.dot(prev_h,Wh) + b.T

    # step 2: input gate
    a_i = sigmoid(a[:,0:H])

    # step 3: forget gate
    a_f = sigmoid(a[:,H:2*H])

    # step 4: output gate
    a_o = sigmoid(a[:,2*H:3*H])

    # step 5: block input gate
    a_g= np.tanh(a[:,3*H:4*H])

    # step 6: next cell state
```

```

next_c = a_f * prev_c + a_i * a_g

# step 7: next hidden state
next_h = a_o * np.tanh(next_c)

# we are having *.copy() since python params are pass by reference
cache = (x, prev_h.copy(), prev_c.copy(), a, a_i, a_f, a_o, a_g, n)

return next_h, next_c, cache

```

Now the backward propagation for one time-step

```

def lstm_step_backward(dnext_h, dnext_c, cache):
    (x, prev_h, prev_c, a, a_i, a_f, a_o, a_g, next_h, next_c, Wx, Wh)
    N, H = dnext_h.shape
    da = np.zeros(a.shape)

    # step 7:
    dnext_c = dnext_c.copy()
    dnext_c += dnext_h * a_o * (1 - np.tanh(next_c) ** 2)
    da_o = np.tanh(next_c) * dnext_h

    # step 6:
    da_f = dnext_c * prev_c
    dprev_c = dnext_c * a_f
    da_i = dnext_c * a_g
    da_g = dnext_c * a_i

    # step 5:
    da[:, 3*H:4*H] = (1 - np.square(a_g)) * da_g

    # step 4:
    da[:, 2*H:3*H] = (1 - a_o) * a_o * da_o

    # step 3:
    da[:, H:2*H] = (1 - a_f) * a_f * da_f

    # step 2:
    da[:, 0:H] = (1 - a_i) * a_i * da_i

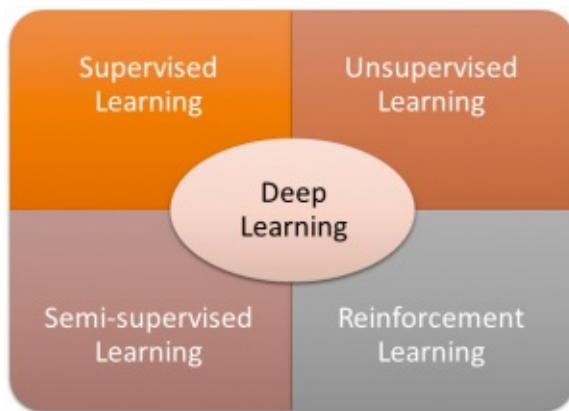
    # step 1:
    db = np.sum(da, axis=0)
    dx = da.dot(Wx.T)
    dWx = x.T.dot(da)
    dprev_h = da.dot(Wh.T)
    dWh = prev_h.T.dot(da)

    return dx, dprev_h, dprev_c, dWx, dWh, db

```

Deep Learning

Introduction



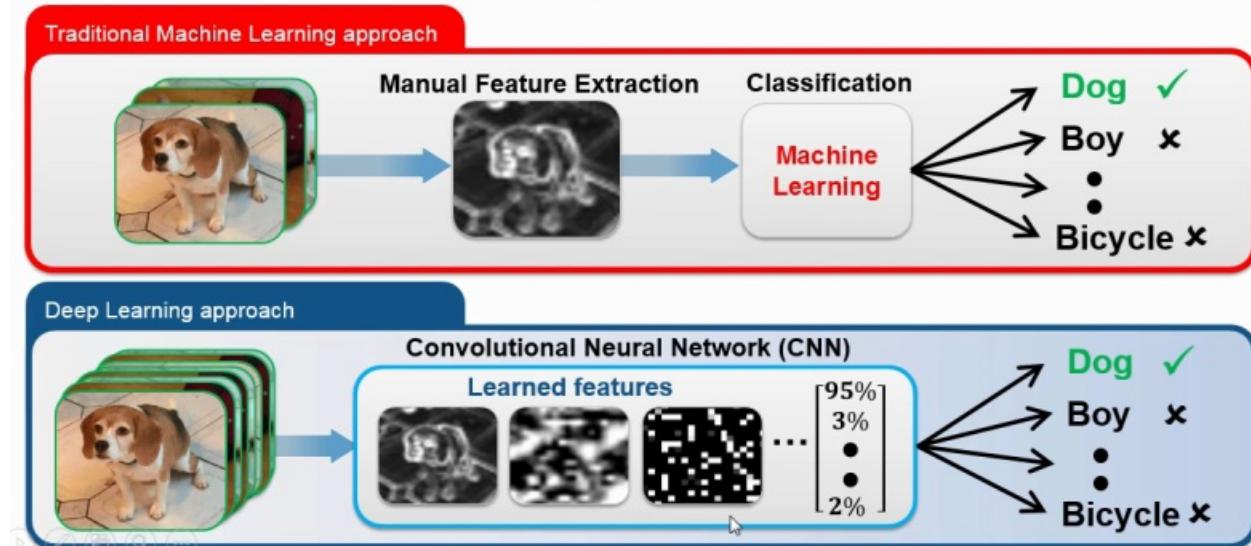
Deep learning is a branch of machine learning based on a set of algorithms that learn to represent the data. Below we list the most popular ones.

- Convolutional Neural Networks
- Deep Belief Networks
- Deep Auto-Encoders
- Recurrent Neural Networks (LSTM)

One of the promises of deep learning is that they will substitute hand-crafted feature extraction. The idea is that they will "learn" the best features needed to represent the given data.

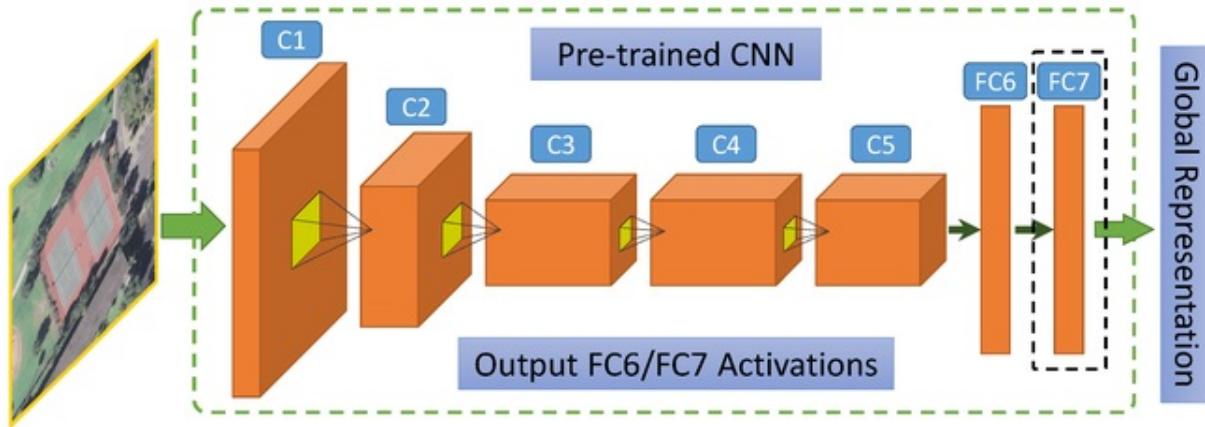
Deep Learning

Deep learning is a **machine learning** technique that can learn **useful representations or features** directly from **images, text and sound**



Layers and layers

Deep learning models are formed by multiple layers. In the context of artificial neural networks the multi layer perceptron (MLP) with more than 2 hidden layers is already a Deep Model. As a rule of thumb deeper models will perform better than shallow models, the problem is that more deep you go more data, you will need to avoid over-fitting.



Layer types

Here we list some of the most used layers

1. Convolution layer
2. Pooling Layer
3. Dropout Layer
4. Batch normalization layer
5. Fully Connected layer
6. Relu, Tanh, sigmoid layer
7. Softmax, Cross Entropy, SVM, Euclidean (Loss layers)

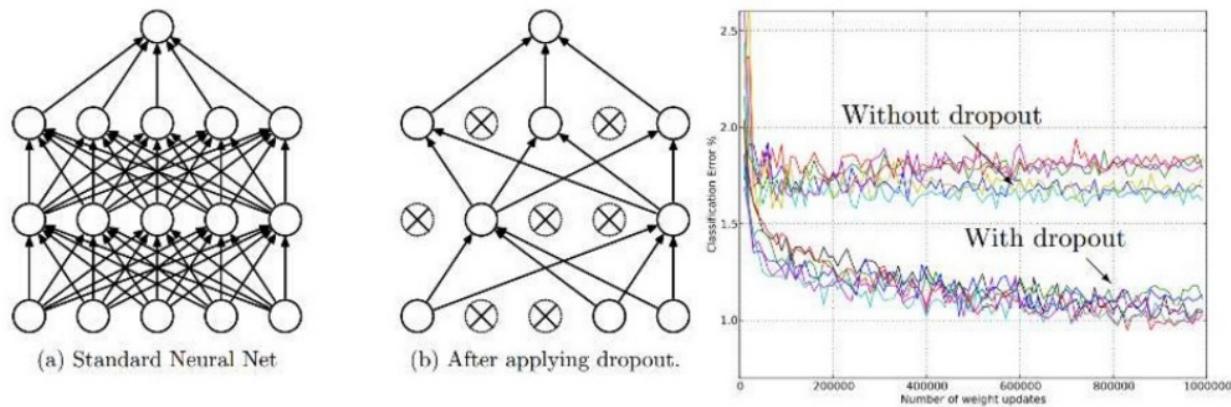
Avoid over-fitting

Besides getting more data, there are some techniques used to combat over-fitting, here is a list of the most common ones:

- Dropout
- Regularization
- Data Augmentation

Dropout

It's a technique that randomly turns off some neurons from the fully connected layer during training.



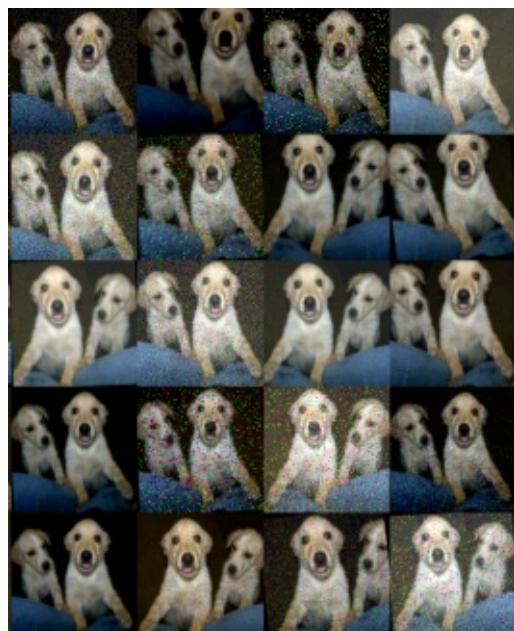
The dropout forces the fully connected layers to learn the same concept in different ways

Regularization

dasdaasdadasdad

Data Augmentation

Synthetically create new training examples by applying some transformations on the input data. For example flipping images. During Imagenet Competition, Alex Krizhevsky (Alexnet) used data augmentation of a factor of 2048, where each class on imangenet has 1000 elements.





a. No augmentation (= 1 image)



b. Flip augmentation (= 2 images)



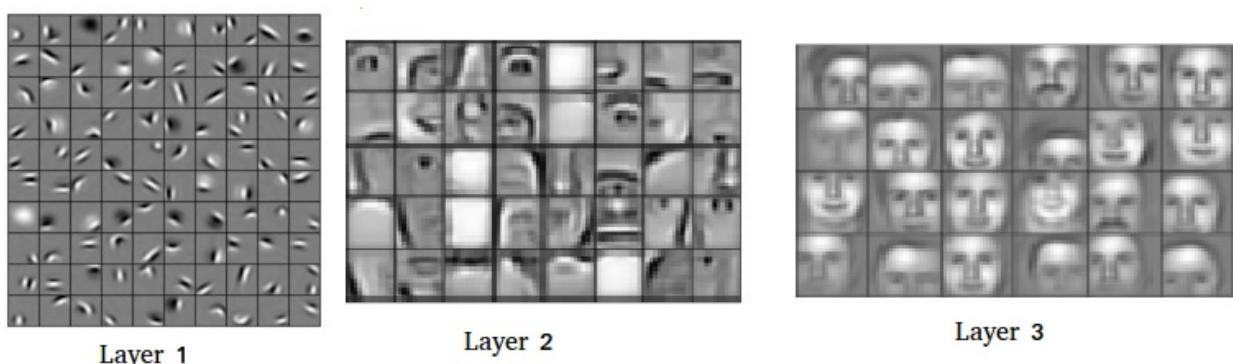
c. Crop+Flip augmentation (= 10 images)

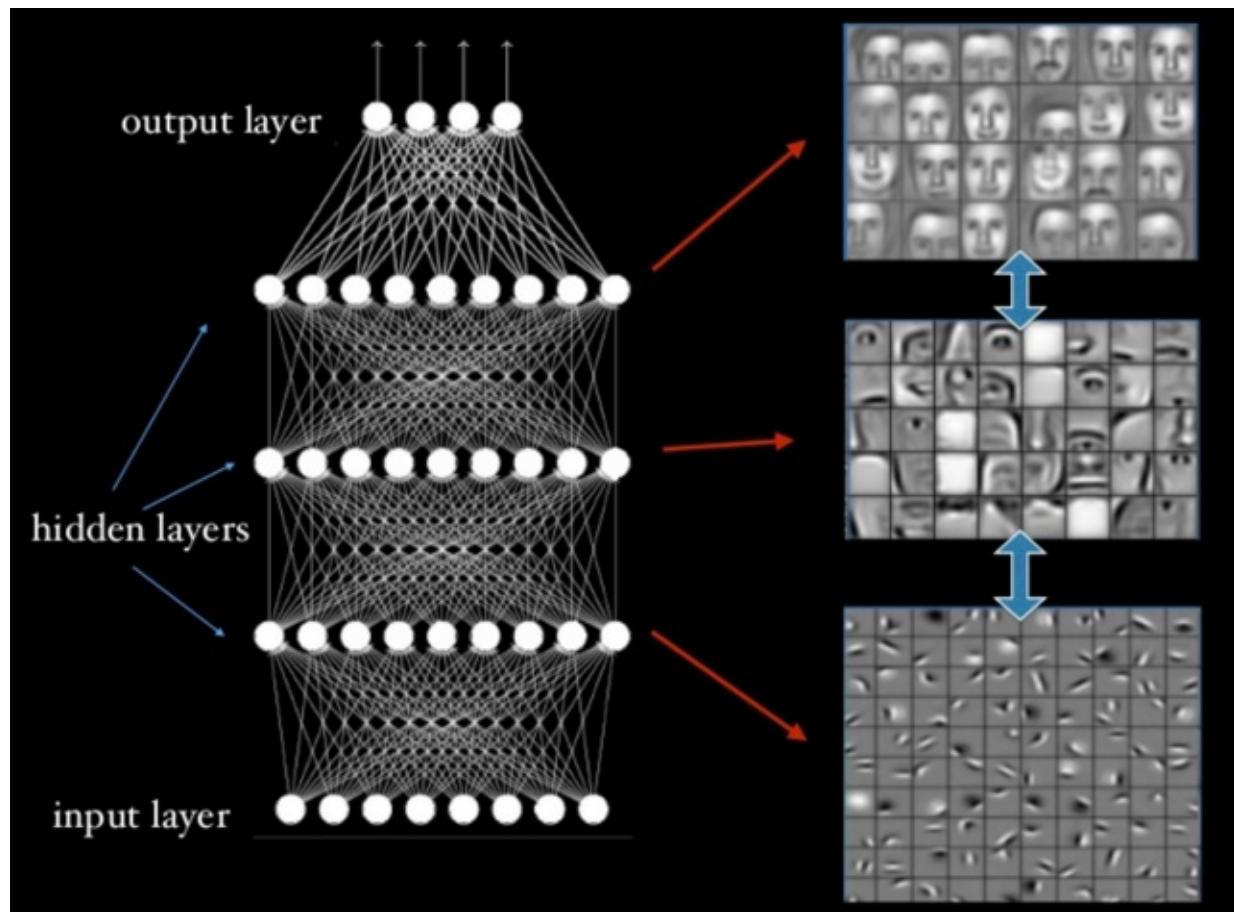


Automatic Hierarchical representation

The idea is to let the learning algorithm to find the best representation that it can for every layer starting from the inputs to the more deepest ones.

The shallow layers learn to represent data on its simpler form and deepest layers learn to represent the data with the concepts learned from the previous ones.





Old vs New

Actually the only new thing is the usage of something that will learn how to represent the data (feature selection) automatically and based on the dataset given. Is not about saying that SVM or decision trees are bad, actually some people use SVMs at the end of the deep neural network to do classification.

The only point is that the feature selection can be easily adapted to new data.

The biggest advantage on this is that if your problem get more complex you just make your model "deeper" and get more data (a lot) to train to your new problem.

Some guys from Deep Learning

*backpropagation,
boltzmann machines*



Geoff Hinton
Google

convolution



Yann Lecun
Facebook

*stacked auto-
encoders*



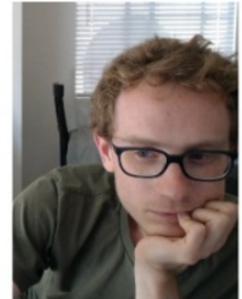
Yoshua Bengio
U. of Montreal

GPU utilization



Andrew Ng
Baidu

dropout



Alex Krizhevsky
Google

Next Chapter

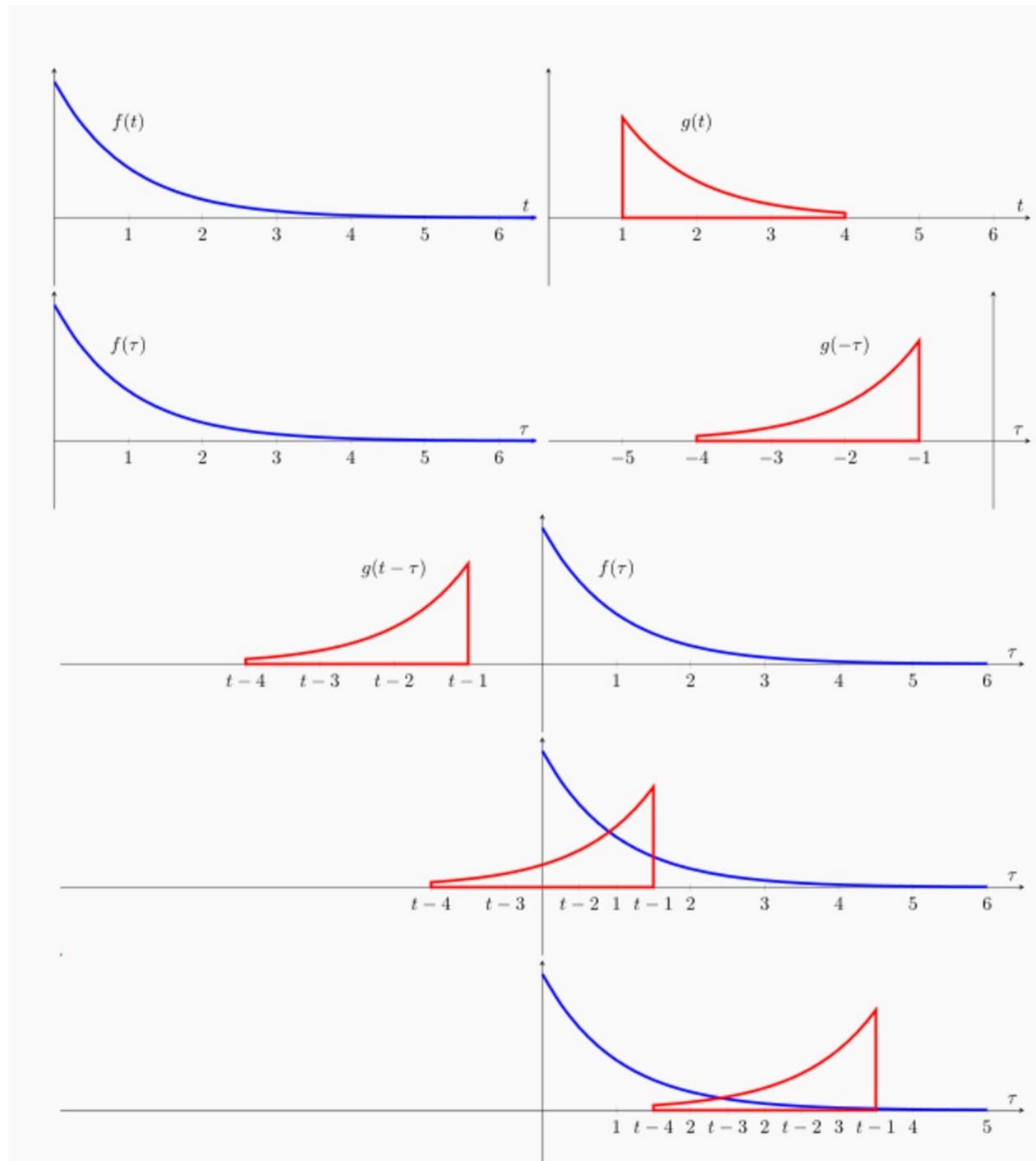
Next chapter we will learn about Convolution Neural Networks.

Convolution

Convolution

Introduction

Convolution is a mathematical operation that does the integral of the product of 2 functions(signals), with one of the signals flipped. For example bellow we convolve 2 signals $f(t)$ and $g(t)$.



So the first thing to do is to flip (180 degrees) the signal g , then slide the flipped g over f , multiplying and accumulating all it's values.

The order that you convolve the signals does not matter for the end result, so $\text{conv}(a,b) == \text{conv}(b,a)$

On this case consider that the blue signal $f(\tau)$ is our input signal and $g(t)$ our kernel, the term kernel is used when you use convolutions to filter signals.

Output signal size 1D

On the case of 1d convolution the output size is calculated like this:

$$\text{outputSize} = (\text{InputSize} - \text{KernelSize}) + 1$$

Application of convolutions

People use convolution on signal processing for the following use cases

- Filter Signals (1d audio, 2d image processing)
- Check how much a signal is correlated to another
- Find patterns on signals

Simple example in matlab and python(numpy)

Bellow we convolve to signals $x = (0,1,2,3,4)$ with $w = (1,-1,2)$.

```
>> x = [0 1 2 3 4]; w = [1 -1 2];
>> conv(x,w)

ans =
    0     1     1     3     5     2     8

>> conv(x,w,'valid')

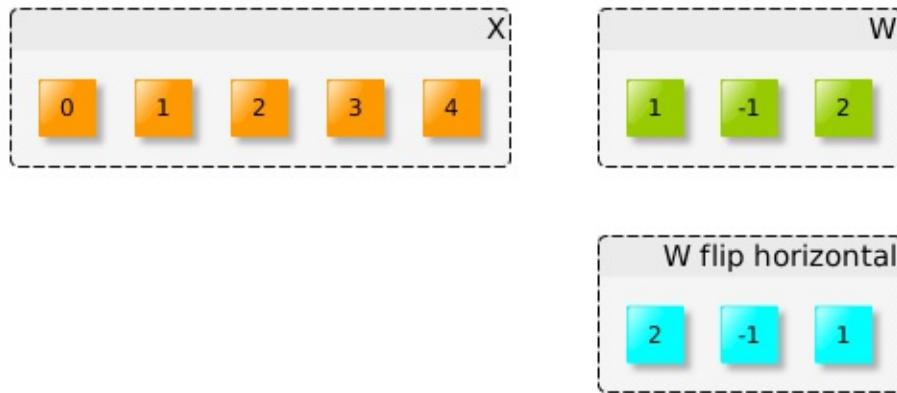
ans =
    1     3     5
```

```
In [1]: import numpy as np
In [2]: x = np.array([0,1,2,3,4])
In [3]: w = np.array([1,-1,2])
In [4]: res = np.convolve(x,w)
In [5]: print res
[0 1 1 3 5 2 8]
```

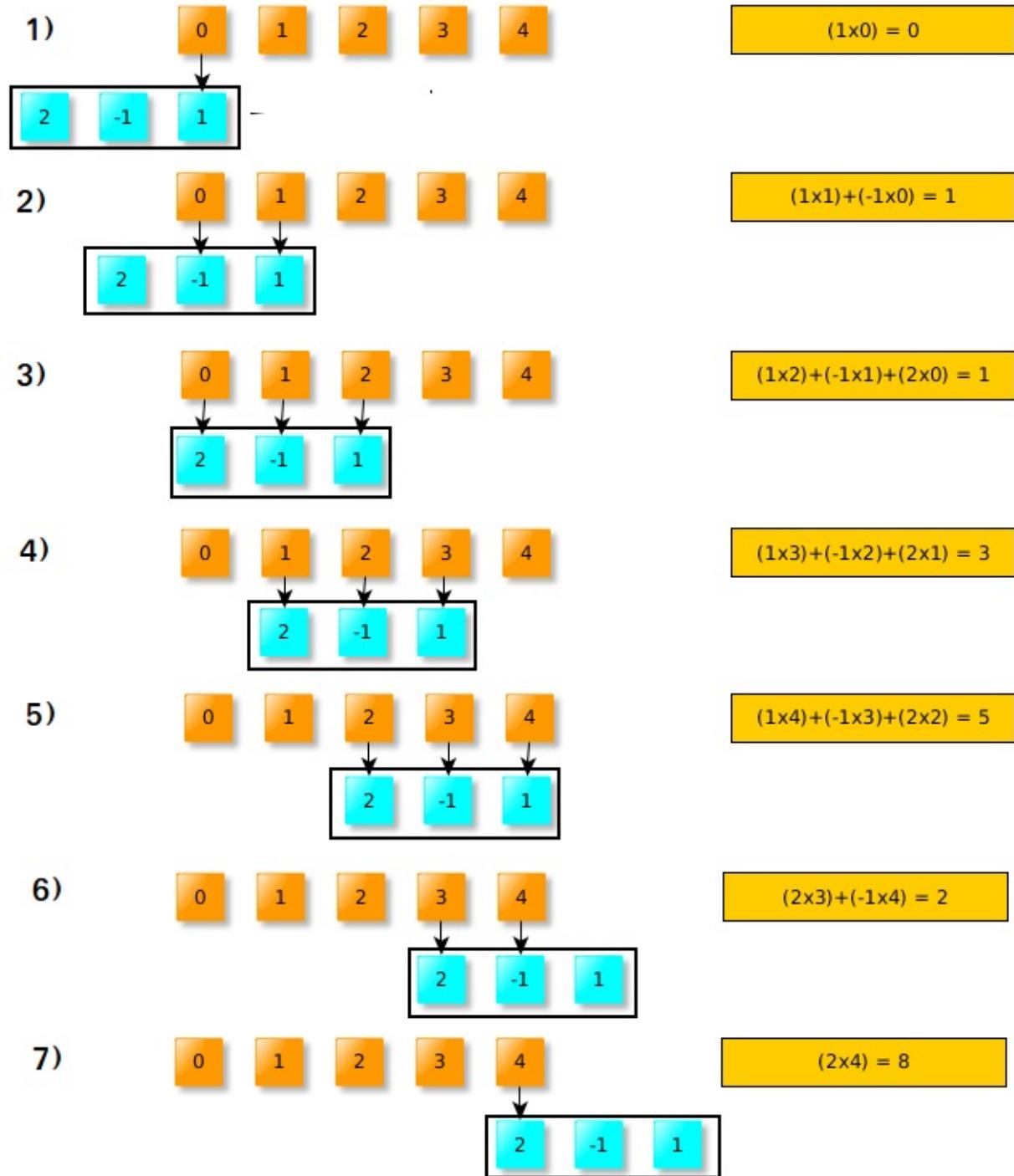
Doing by hand

To understand better then concept of convolution let's do the example above by hand. Basically we're going to convolve 2 signals (x, w). The first thing is to flip W horizontally (Or rotate to left 180 degrees)

Convolution



After that we need to slide the flipped W over the input X



Observe that on steps 3,4,5 the flipped window is completely inside the input signal. Those results are called 'valid'. The cases where the flipped window is not fully inside the input window(X), we can consider to be zero, or calculate what is possible to be calculated, ex on step 1 we multiply 1 by zero, and the rest is simply ignored.

Input padding

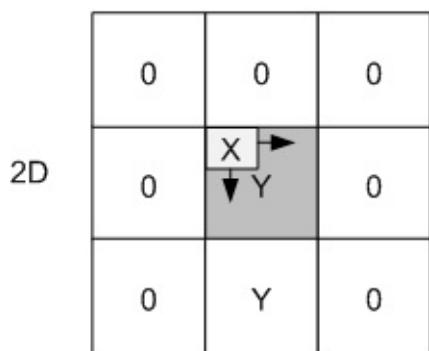
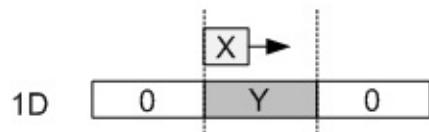
In order to keep the convolution result size the same size as the input, and to avoid an effect called circular convolution, we pad the signal with zeros.

Where you put the zeros depends on what you want to do, ie: on the 1d case you can concatenate them on the end, but on 2d is normally around the original signal

Convolution



linear



On matlab you can use the command padarray to pad the input:

```
>> x
```

```
x(:,:,1) =
```

1	1	0	2	0
2	2	2	2	1
0	0	0	2	1
2	2	2	2	1
2	0	2	2	1

```
x(:,:,2) =
```

2	1	0	0	0
0	2	0	1	0
1	0	1	2	0
1	2	0	2	1
1	2	1	2	2

```
x(:,:,3) =
```

2	1	1	2	2
1	1	1	0	0
2	0	1	0	2
0	2	0	2	1
0	0	2	1	0

```
>> padarray(x,[1 1])
```

```
ans(:,:,1) =
```

0	0	0	0	0	0	0
0	1	1	0	2	0	0
0	2	2	2	2	1	0
0	0	0	0	2	1	0
0	2	2	2	2	1	0
0	2	0	2	2	1	0
0	0	0	0	0	0	0

```
ans(:,:,2) =
```

0	0	0	0	0	0	0
0	2	1	0	0	0	0
0	0	2	0	1	0	0
0	1	0	1	2	0	0
0	1	2	0	2	1	0
0	1	2	1	2	2	0
0	0	0	0	0	0	0

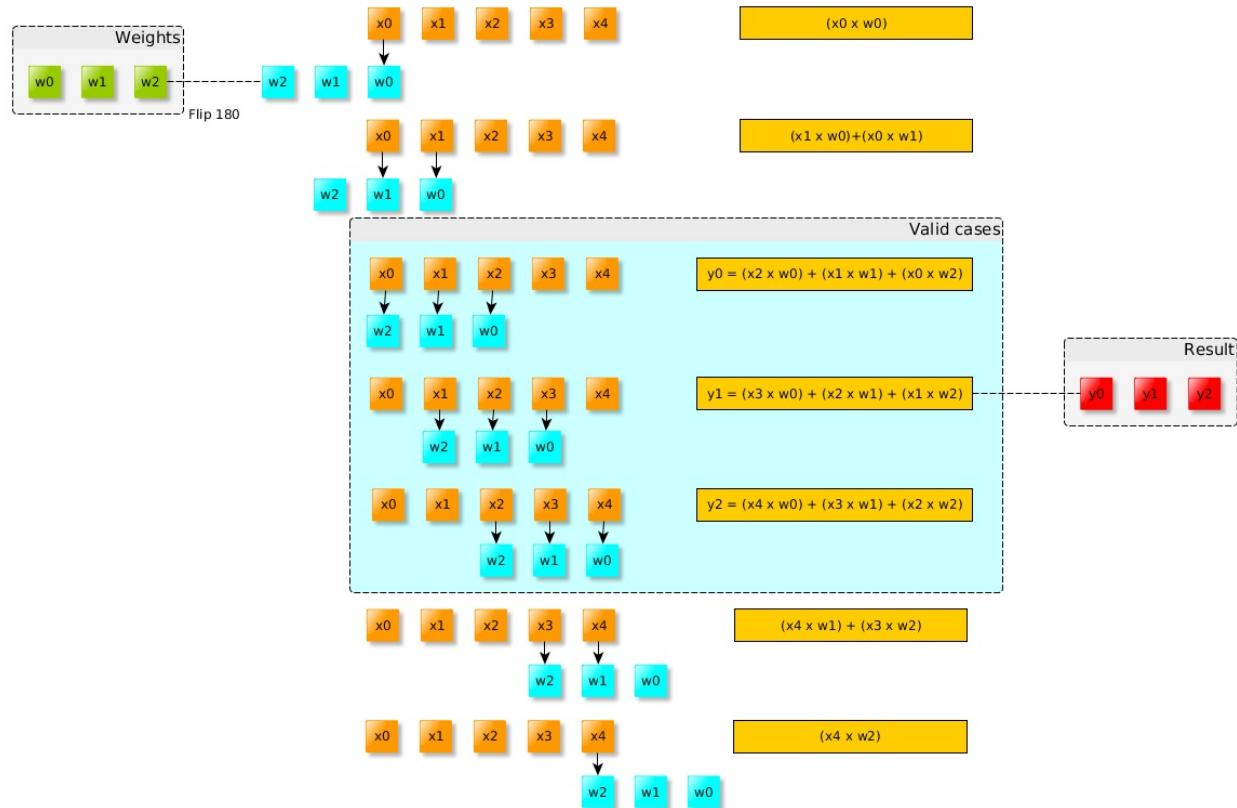
```
ans(:,:,3) =
```

0	0	0	0	0	0	0
0	2	1	1	2	2	0
0	1	1	1	0	0	0
0	2	0	1	0	2	0
0	0	2	0	2	1	0
0	0	0	2	1	0	0
0	0	0	0	0	0	0

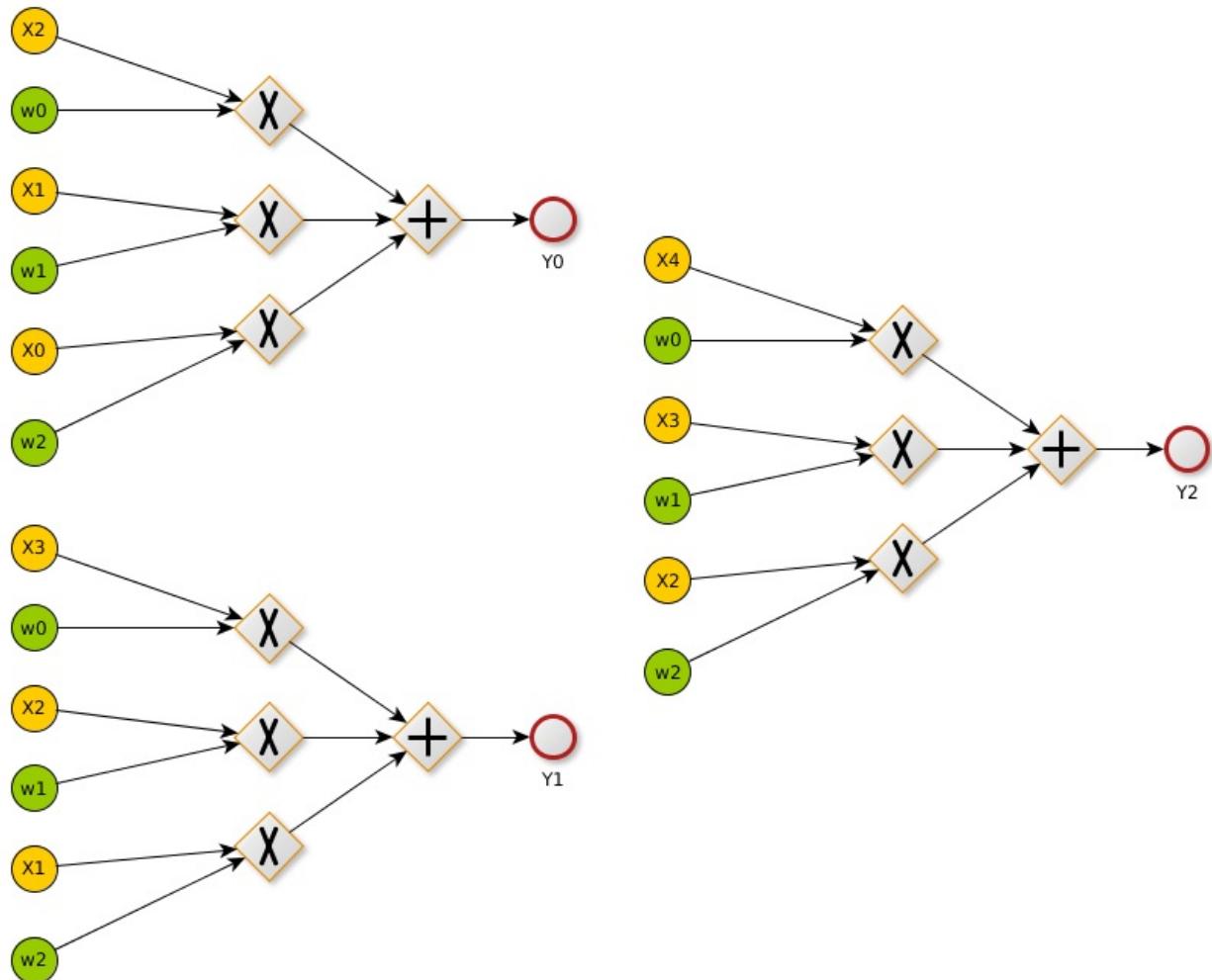
Transforming convolution to computation graph

In order to calculate partial derivatives of every node inputs and parameters, it's easier to transform it to a computational graph. Here I'm going to transform the previous 1d convolution, but this can be extended to 2d convolution as well.

Convolution



Here our graph will be created on the valid cases where the flipped kernel(weights) will be fully inserted on our input window.



We're going to use this graph in the future to infer the gradients of the inputs (x) and weights (w) of the convolution layer

2d Convolution

Extending to the second dimension, 2d convolutions are used on image filters, and when you would like to find a specific patch on image.



$*$

1	0	-1
2	0	-2
1	0	-1





Matlab and python examples

```
>> a = [1 3 1; 0 -1 1; 2 2 -1]
```

```
a =
```

1	3	1
0	-1	1
2	2	-1

```
>> w = [1 2; 0 -1]
```

```
w =
```

1	2
0	-1

```
>> conv2(a,w)
```

```
ans =
```

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

```
>> conv2(a,w,'valid')
```

```
ans =
```

-2	-4
6	4

Doing by hand

First we should flip the kernel, then slide the kernel on the input signal.

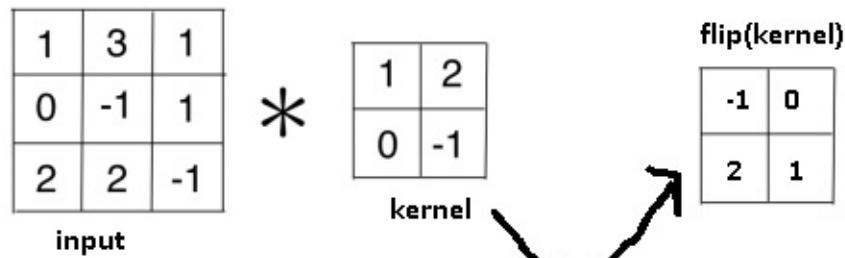
Before doing this operation by hand check out the animation showing how this sliding works

1 _{x1}	1 _{x0}	1 _{x1}	0	0
0 _{x0}	1 _{x1}	1 _{x0}	1	0
0 _{x1}	0 _{x0}	1 _{x1}	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved Feature



Multiply the window element by element with `flip(kernel)`, then sum the results

1	3	1
0	-1	1
2	2	-1

=> $1(-1)+3(0)+0(2)-1(1) = -2$

1	3	1
0	-1	1
2	2	-1

=> $3(-1)+1(0)-1(2)+1(1) = -4$

1	3	1
0	-1	1
2	2	-1

=> $0(-1)-1(0)+2(2)+2(1) = 6$

1	3	1
0	-1	1
2	2	-1

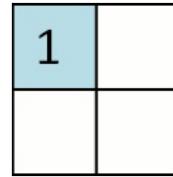
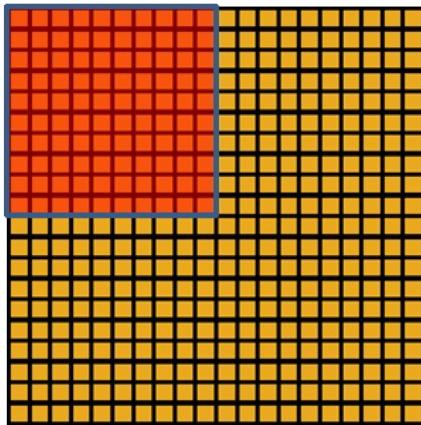
=> $-1(-1)+1(0)+2(2)-1(1) = 4$

result(valid)
-2 -4 6 4

Stride

By default when we're doing convolution we move our window one pixel at a time (`stride=1`), but sometimes in convolutional neural networks we move more than one pixel. Strides of 2 are used on pooling layers.

Observe that below the red window is moving much more than one pixel time.



Convolved feature Pooled feature

Output size for 2d

If we consider the padding and stride, the output size of convolution is defined as:

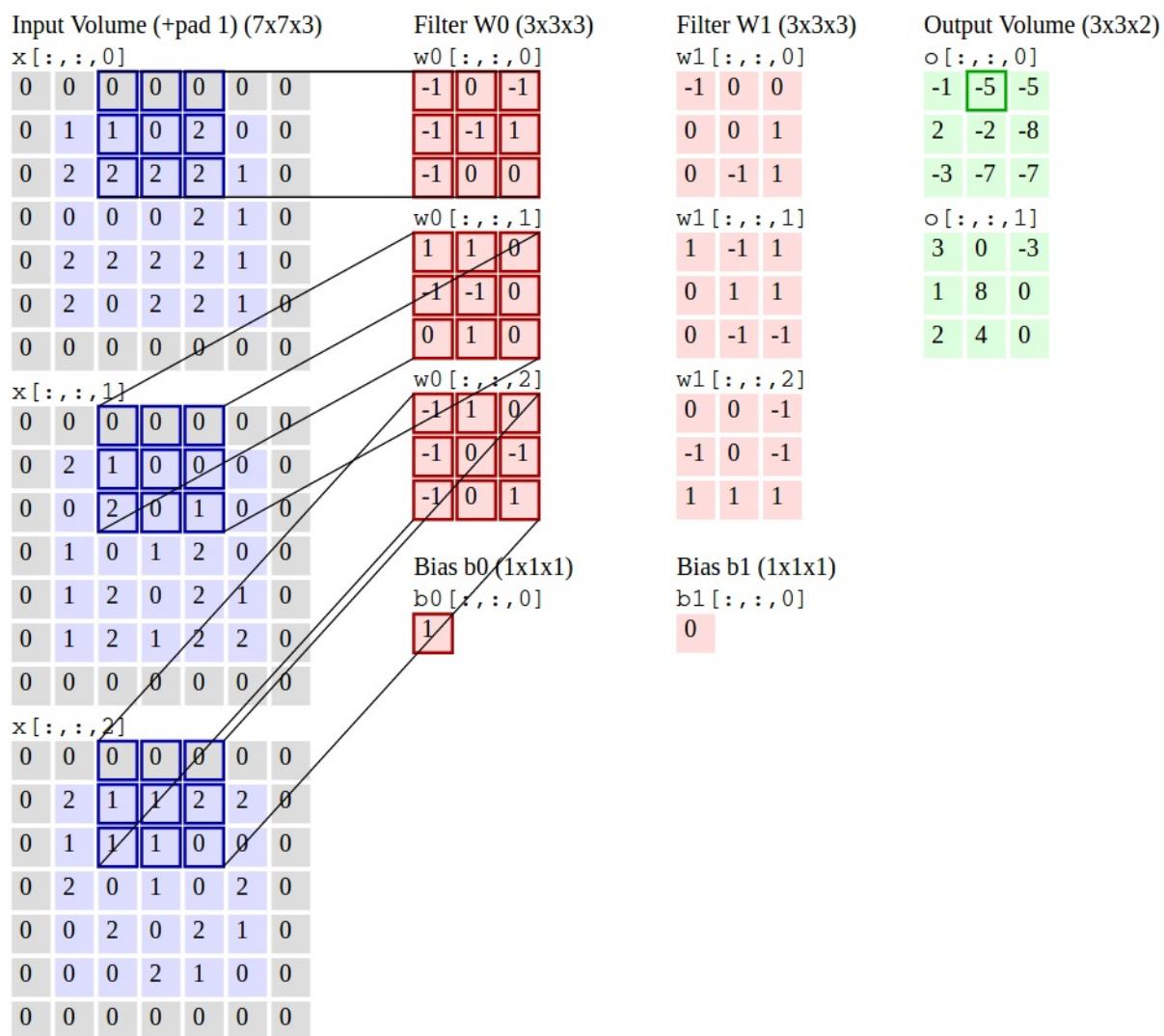
$$\text{outputSize}_W = (W - F + 2P)/S + 1$$

$$\text{outputSize}_H = (H - F + 2P)/S + 1$$

F is the size of the kernel, normally we use square kernels, so F is both the width and height of the kernel

Implementing convolution operation

The example below will convolve a 5x5x3 (WxHx3) input, with a conv layer with the following parameters Stride=2, Pad=1, F=3(3x3 kernel), and K=2 (two filters). Our input has 3 channels, so we need a 3x3x3 kernel weight. We have 2 filters (K=2) so we have 2 output activation (3x3x2). Calculating the output size we have: $(5 - 3 + 2)/2 + 1 = 3$



So basically we need to calculate 2 convolutions, one for each 3x3x3 filter (w_0, w_1), and remembering to add the bias.

```
Command Window
New to MATLAB? See resources for Getting Started.
>> result = convn_vanilla(A_pad,w0,2)+1

result =
    -1     -5     -5
     2     -2     -8
    -3     -7     -7

>> result = convn_vanilla(A_pad,w1,2)+0

result =
     3      0     -3
     1      8      0
     2      4      0
```

The code below (vanilla version) cannot be used on real life, because it will be slow. Usually deep learning libraries do the convolution as a matrix multiplication, using im2col/col2im.

```

%% Convolution n dimensions
% The following code is just a extension of conv2d_vanila for n dime
% Parameters:
% Input: H x W x depth
% K: kernel F x F x depth
% S: stride (How many pixels he window will slide on the input)
% This implementation is like the 'valid' parameter on normal convolu

function outConv = convn_vanilla(input, kernel, S)
% Get the input size in terms of rows and cols. The weights should ha
% same depth as the input volume(image)
[rowsIn, colsIn, ~] = size(input);

% Get volume dimensio
depthInput = ndims(input);

% Get the kernel size, considering a square kernel always
F = size(kernel,1);

%% Initialize outputs
sizeRowsOut = ((rowsIn-F)/S) + 1;
sizeColsOut = ((colsIn-F)/S) + 1;
outConvAcc = zeros(sizeRowsOut , sizeColsOut, depthInput);

%% Do the convolution
% Convolve each channel on the input with it's respective kernel chai
% at the end sum all the channel results.
for depth=1:depthInput
    % Select input and kernel current channel
    inputCurrDepth = input(:,:,depth);
    kernelCurrDepth = kernel(:,:,depth);
    % Iterate on every row and col, (using stride)
    for r=1:S:(rowsIn-1)
        for c=1:S:(colsIn-1)
            % Avoid sampling out of the image.
            if (((c+F)-1) <= colsIn) && (((r+F)-1) <= rowsIn)
                % Select window on input volume (patch)
                sampleWindow = inputCurrDepth(r:(r+F)-1,c:(c+F)-1);
                % Do the dot product
                dotProd = sum(sampleWindow(:) .* kernelCurrDepth(:))
                % Store result
                outConvAcc(ceil(r/S),ceil(c/S),depth) = dotProd;
            end
        end
    end
end

% Sum elements over the input volume dimension (sum all the channels
outConv = sum(outConvAcc,depthInput);
end

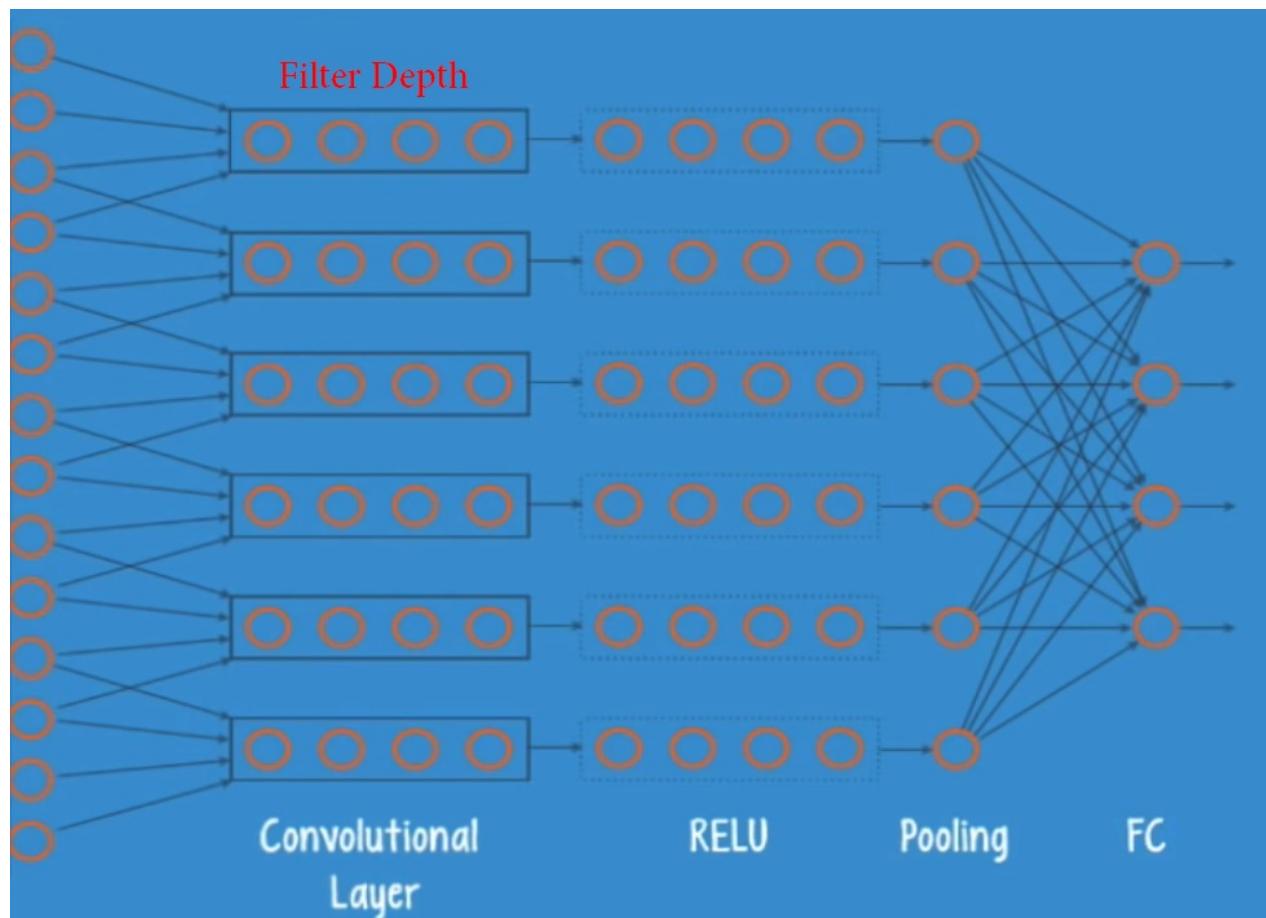
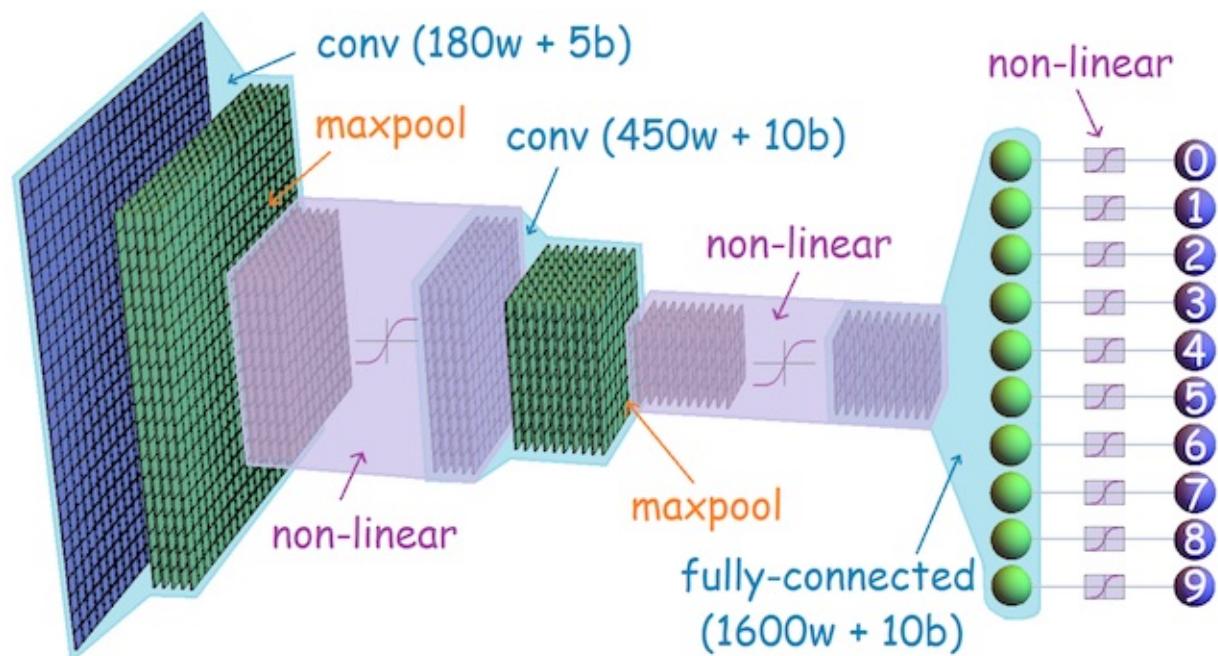
```

Next Chapter

Next chapter we will learn about Deep Learning.

Convolutional Neural Networks

Convolutional Neural Networks

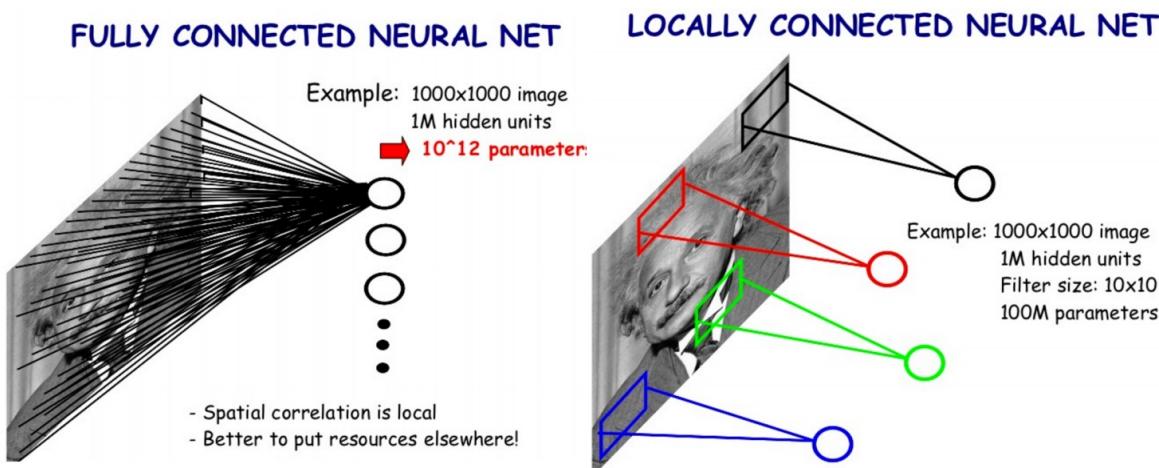


Introduction

A CNN is composed of layers that filters(convolve) the inputs to get useful information. These convolutional layers have parameters(kernel) that are learned so that these filters are adjusted automatically to extract the most useful information for the task at hand without feature selection. CNN are better to work with images. Normal Neural networks does not fit well on image classification problems

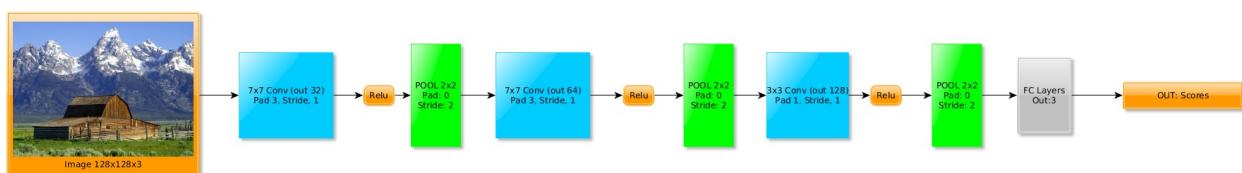
Comparison of Normal Neural network

On normal neural networks, we need to convert the image to a single 1d vector $[1, (width \cdot height \cdot channels)]$, then send this data to a hidden layer which is fully connected. On this scenario each neuron will have 10^{12} parameters per neuron.



Common architecture

Normally the pattern [CONV->ReLU->Pool->CONV->ReLU->Pool->FC->Softmax_loss(during train)] is quite common.

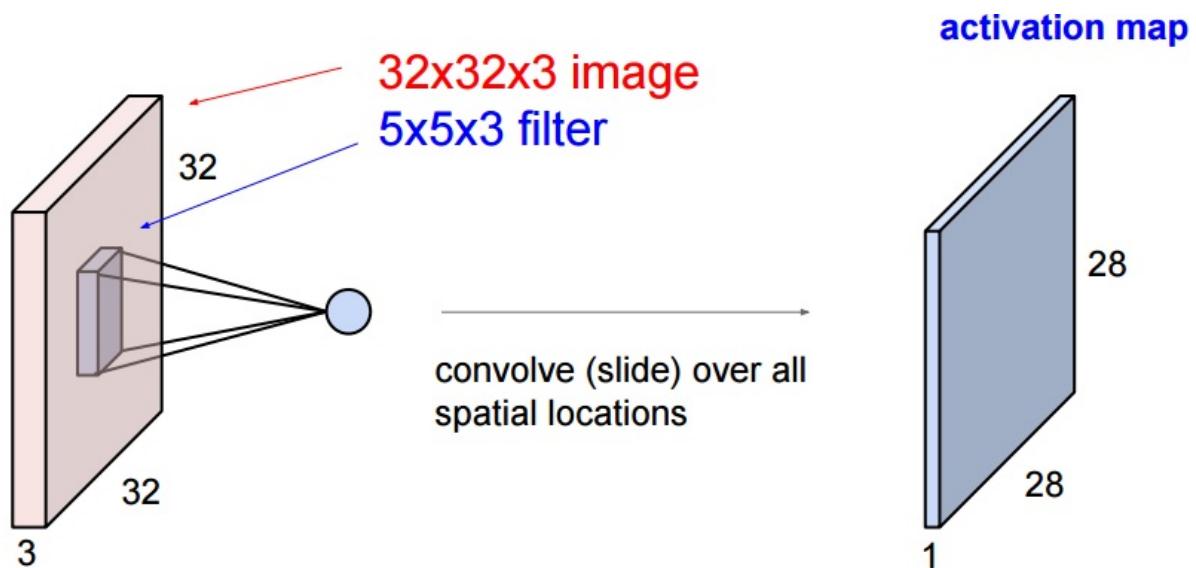


Main actor the convolution layer

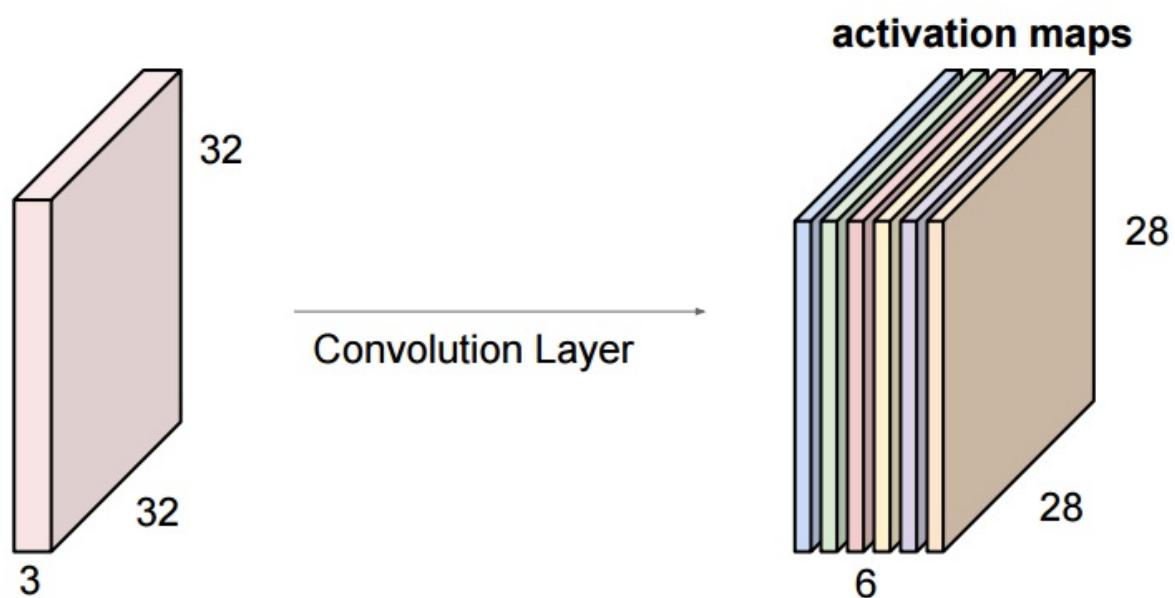
The most important operation on the convolutional neural network are the convolution layers, imagine a 32x32x3 image if we convolve this image with a 5x5x3 (The filter depth must have the same depth as the input), the result will be an activation map 28x28x1.



The filter will look for a particular thing on all the image, this means that it will look for a pattern in the whole image with just one filter.



Now consider that we want our convolution layer to look for 6 different things. On this case our convolution layer will have 6 5x5x3 filters. Each one looking for a particular pattern on the image.



By the way the convolution by itself is a linear operation, if we don't want to suffer from the same

problem of the linear classifiers we need to add at the end of the convolution layer a non-linear layer. (Normally a Relu)

Another important point of using convolution as pattern match is that the position where the thing that we want to search on the image is irrelevant. On the case of neural networks the model/hypothesis will learn an object on the exact location where the object is located during training.

Convolution layer Hyper parameters

Those are the parameters that are used to configure a convolution layer

- Kernel size(K): Small is better (But if is on the first layer, takes a lot of memory)
- Stride(S): How many pixels the kernel window will slide (Normally 1, in conv layers, and 2 on pooling layers)
- Zero Padding(pad): Put zeros on the image border to allow the conv output size be the same as the input size (F=1, PAD=0; F=3, PAD=1; F=5, PAD=2; F=7, PAD=3)
- Number of filters(F): Number of patterns that the conv layer will look for.

Output size

By default the convolution output will always have a result smaller than the input. To avoid this behaviour we need to use padding. To calculate the convolution output (activation map) size we need this formula:

$$H_{out} = 1 + \frac{H_{in} + (2 \cdot pad) - K_{height}}{S}$$

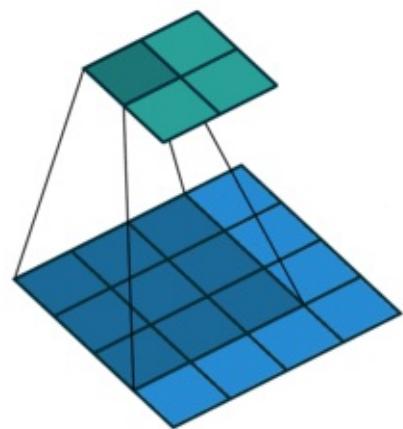
$$W_{out} = 1 + \frac{W_{in} + (2 \cdot pad) - K_{width}}{S}$$

Vizualizing convolution

Here we will see some examples of the convolution window sliding on the input image and change some of it's hyper parameters.

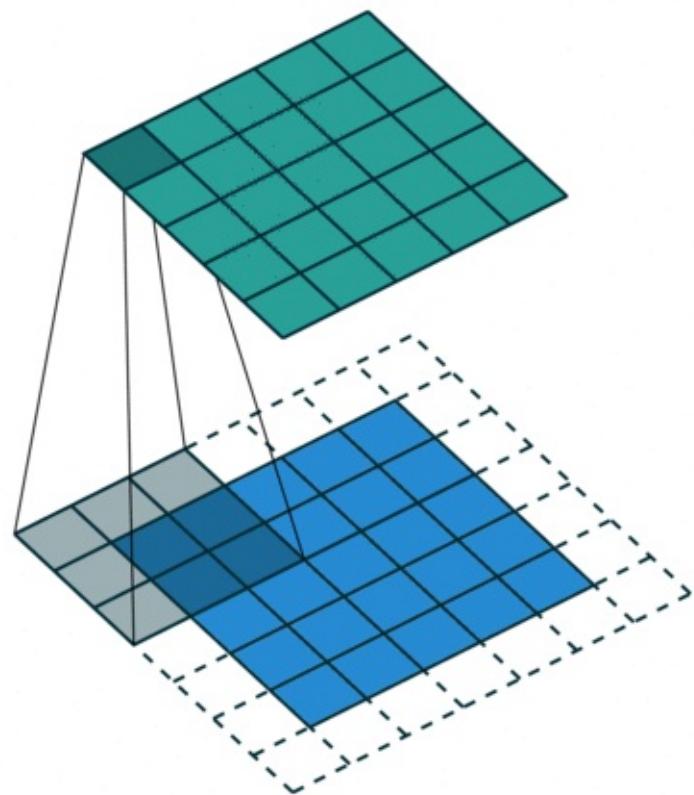
Convolution with no padding and stride of 1

Here we have a input 4x4 convolved with a filter 3x3 (K=3) with stride (S=1) and padding (pad=0)



Convolution with padding and stride of 1

Now we have an input 5x5 convolved with a filter 3x3 ($k=3$) with stride ($S=1$) and padding (pad=1). On some libraries there is a feature that always calculate the right amount of padding to keep the output spatial dimensions the "same" as the input dimensions.



Number of parameters(weights)

Here we show how to calculate the number of parameters used by one convolution layer. We will illustrate with a simple example:

Input: 32x32x3, 32x32 RGB image

CONV: Kernel(F):5x5, Stride:1, Pad:2, numFilters:10

$$\begin{aligned} num_{parameters} &= ((F * F * depth_{input}) + 1) * num_{filters} \\ \therefore num_{parameters} &= ((5 * 5 * 3) + 1) * 10 = 760 \end{aligned}$$

You can omit the "+1" parameter (Bias), to simplify calculations.

Amount of memory

Here we show how to calculate the amount of memory needed on the convolution layer.

Input: 32x32x3, 32x32 RGB image

CONV: Kernel(F):5x5, Stride:1, Pad:2, numFilters:10, as we use padding our output volume will be 32x32x10, so the amount of memory in bytes is: 10240 bytes

So the amount of memory is basically just the product of the dimensions of the output volume which is a 4d tensor.

$$mem = [N_{batch}, C_{depth}, H_{out}, W_{out}]$$

Where:

- N_{batch} : Output batch size
- C_{depth} : The output volume or on the case of convolution the number of filters
- H_{out} : The height of the output activation map
- W_{out} : The width of the output activation map

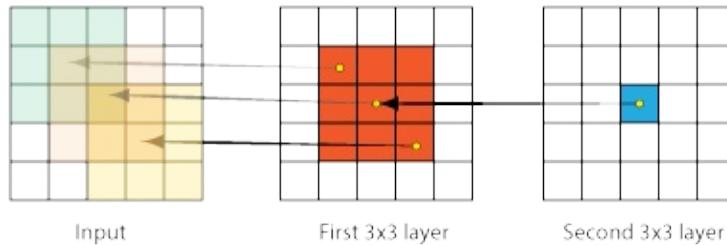
1x1 Convolution

This type of convolution is normally used to adapt depths, by merging them, without changing the spatial information.

Substituting Big convolutions

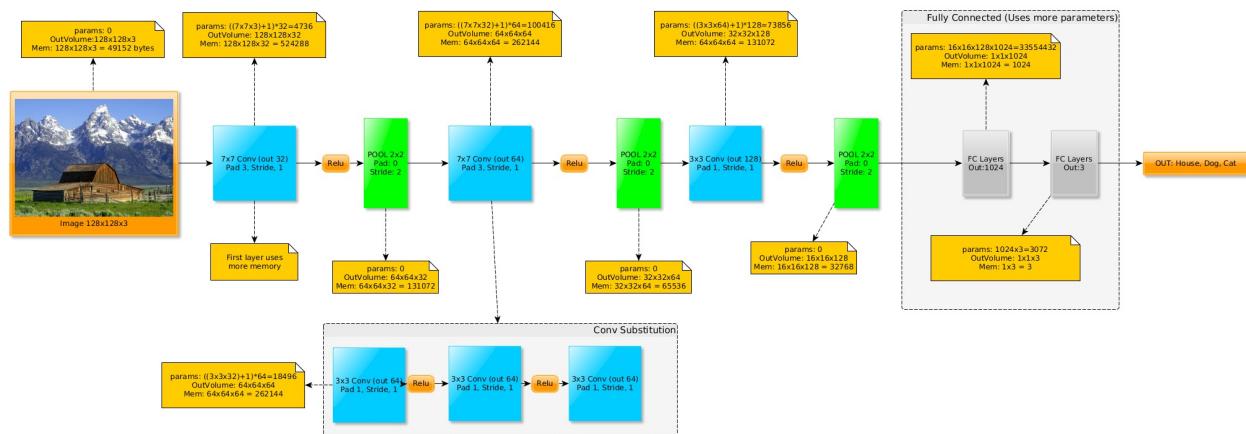
Here we explain what is the effect of cascading several small convolutions, on the diagram below we have 2 3x3 convolution layers. If you start from the second layer on the right, one neuron on the second layer, has a 3x3 receptive field, and each neuron on the first layer creates a 5x5 receptive field on the input.

So in simpler words cascading can be used to represent bigger ones.



Observe that cascading 2 3x3 convolutions, your receptive field on the input has size 5x5

The new trend on new successful models is to use smaller convolutions, for example a 7x7 convolution can be substituted with 3 3x3 convolutions with the same depth. This substitution cannot be done on the first conv layer due to the depth mismatch between the first conv layer and the input file depth (Unless if your first layer has only 3 filters).



On the diagram above we substitute one 7x7 convolution by 3 3x3 convolutions, observe that between them we have relu layers, so we have more non-linearities. Also we have less weights and multiply-add operations so it will be faster to compute.

Calculating the substitution of a 7x7 convolution

Imagine a 7x7 convolution, with C filters, being used on a input volume WxHxC we can calculate the number of weights as:

$$numWeights_{7x7} = C(7.7.C) \therefore 49.C^2$$

Now if we use 3 3x3 convolutions with C filters, we would have

$$numWeights_{3x3} = C(3.3.C) \therefore 9.C^2 \therefore \text{due to 3 filters } 3.(9.C^2) = 27.C^2$$

We still have less parameters, as we need to use Relu between the layers to break the linearity (otherwise the conv layers in cascade will appear as a single 3x3 layer) we have more non-linearity, less parameters, and more performance.

Substitution on the first layer

As mentioned before, we cannot substitute large convolutions on the first layer. Actually small convolutions on the first layer cause a memory consume explosion.

To illustrate the problem let's compare the first layer of a convolution neural network as been 3×3 with 64 filters and stride of 1 and the same depth with 7×7 and stride of 2, consider the image size to be $256 \times 256 \times 3$.

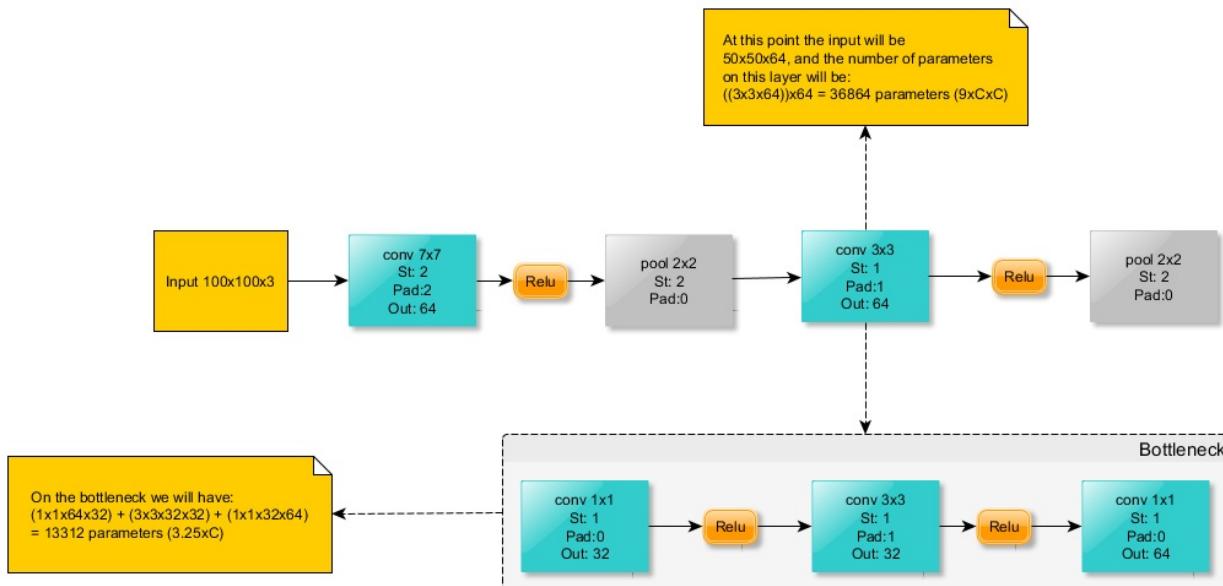
$$mem_{3 \times 3} = 256.256.64 \therefore 4mb$$

TODO: How the stride and convolution size affect the memory consumption

Continuing on 3×3 substitution (Bottleneck)

It's also possible to simplify the 3×3 convolution with a mechanism called bottleneck. This again will have the same representation of a normal 3×3 convolution but with less parameters, and more non-linearities.

Observe that the substitution is made on the 3×3 convolution that has the same depth as the previous layer (On this case $50 \times 50 \times 64$)



Here we calculate how much parameters we use on the bottleneck, remember that on 3×3 is $9.C^2$

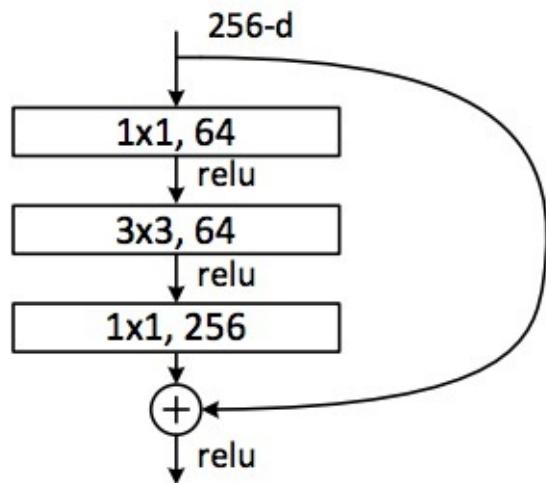
```
>> syms H W C
>> numParams = ((1*1*C)*C/2) + (3*3*C/2*C/2) + ((1*1*C/2)*C);
>> pretty(numParams)
 2
13 C
-----
 4

>> 13/4

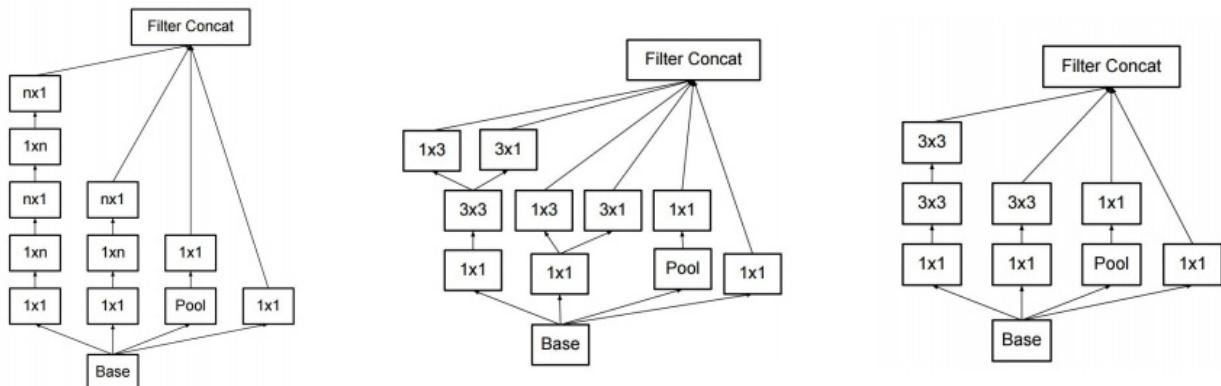
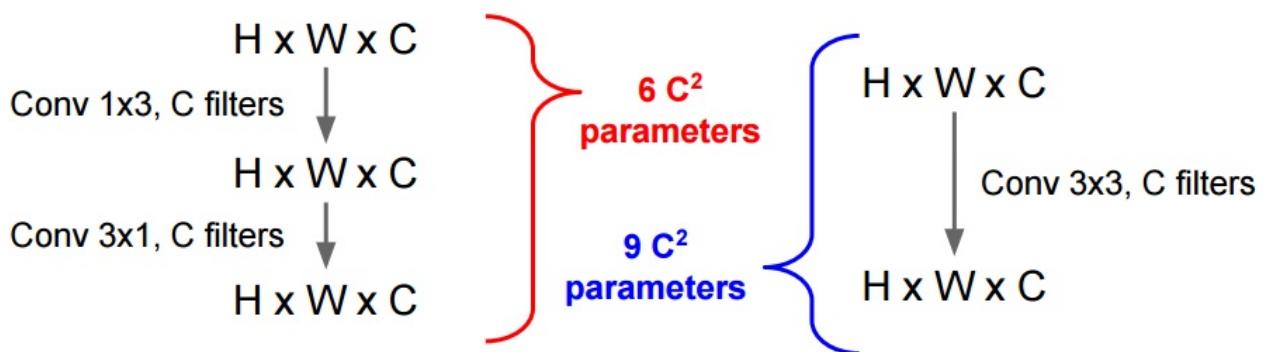
ans =
 3.2500
```

So the bottleneck uses $(3.25)C^2$, which is less.

The bottleneck is also used on microsoft residual network.



Another option to break $3 \times 3 \times C$ convolutions is to use $1 \times 3 \times C$, then $3 \times 1 \times C$, this has been used on residual googlenet inception layer.



FC -> Conv Layer Conversion

It's possible to convert Fully connected layers to convolution layers and vice-versa, but we are more interested on the FC->Conv conversion. This is done to improve performance.

For example imagine a FC layer with output $K=4096$ and input $7 \times 7 \times 512$, the conversion would be:
CONV: Kernel:7x7, Pad:0, Stride:1, numFilters:4096.

Using the 2d convolution formula size:

$outputSize_W = (W - F + 2P)/S + 1 \therefore (7 - 7 + 2(0))/1 + 1$, which will be $1 \times 1 \times 4096$.

In resume what you gain by converting the FC layers to convolution:

- Performance: It's faster to compute due to the weight sharing
- You can use images larger than the ones that you trained, without changing nothing
- You will be able to detect 2 objects on the same image (If you use a bigger image) your final output will be bigger then a single row vector.



Next Chapter

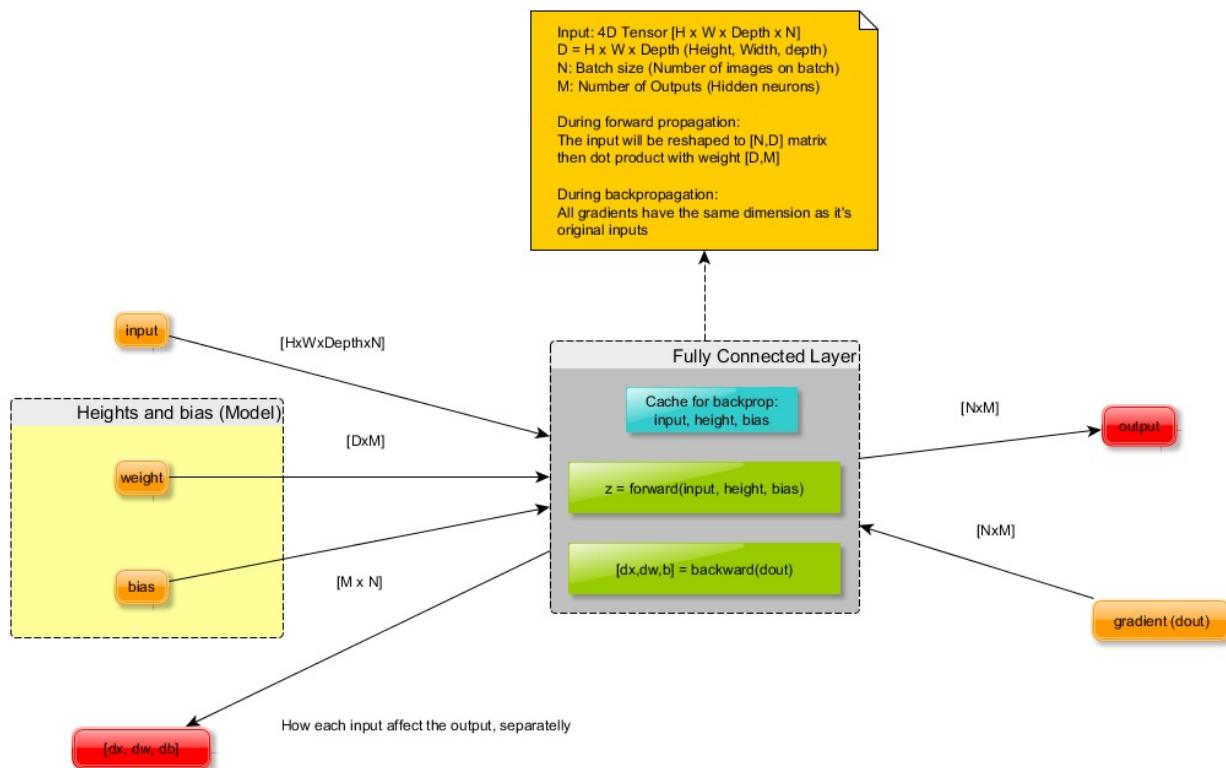
Next chapter we will learn about Fully Connected layers.

Fully Connected Layer

Fully Connected Layer

Introduction

This chapter will explain how to implement in matlab and python the fully connected layer, including the forward and back-propagation.



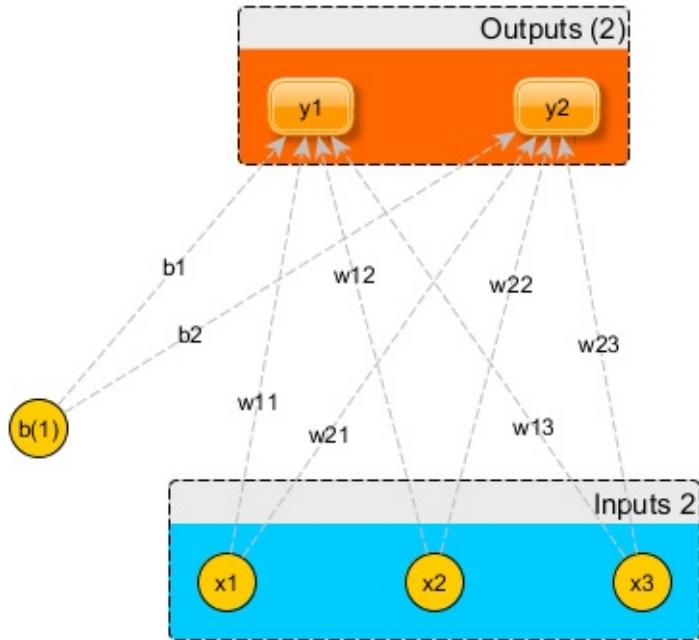
First consider the fully connected layer as a black box with the following properties: On the forward propagation

1. Has 3 inputs (Input signal, Weights, Bias)
2. Has 1 output

On the back propagation

1. Has 1 input ($dout$) which has the same size as output
2. Has 3 (dx, dw, db) outputs, that has the same size as the inputs

Neural network point of view



Just by looking the diagram we can infer the outputs:

$$y_1 = [(w_{11} \cdot x_1) + (w_{12} \cdot x_2) + (w_{13} \cdot x_3)] + b_1$$

$$y_2 = [(w_{21} \cdot x_1) + (w_{22} \cdot x_2) + (w_{23} \cdot x_3)] + b_2$$

Now vectorizing (put on matrix form): (Observe 2 possible versions)

$$\underbrace{\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix}}_{\text{One column per } x \text{ dimension}} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \therefore$$

One column per x dimension

$$H(X) = (W \cdot x) + b^T$$

$$H(X) = (W^T \cdot x) + b$$

Depending on the format that you choose to represent W attention to this because it can be confusing.

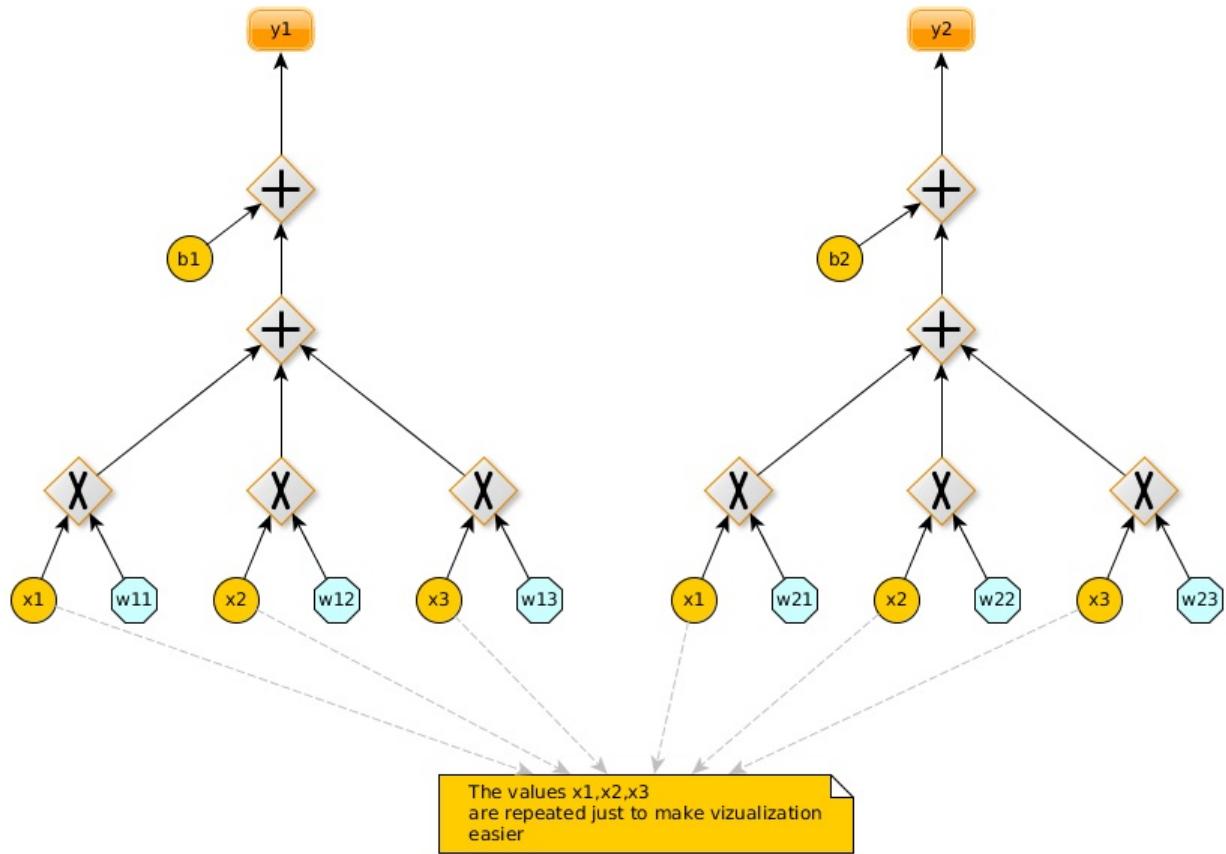
For example if we choose X to be a column vector, our matrix multiplication must be:

$$([x_1 \ x_2 \ x_3] \cdot \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix}) + [b_1 \ b_2] = [y_1 \ y_2] \therefore$$

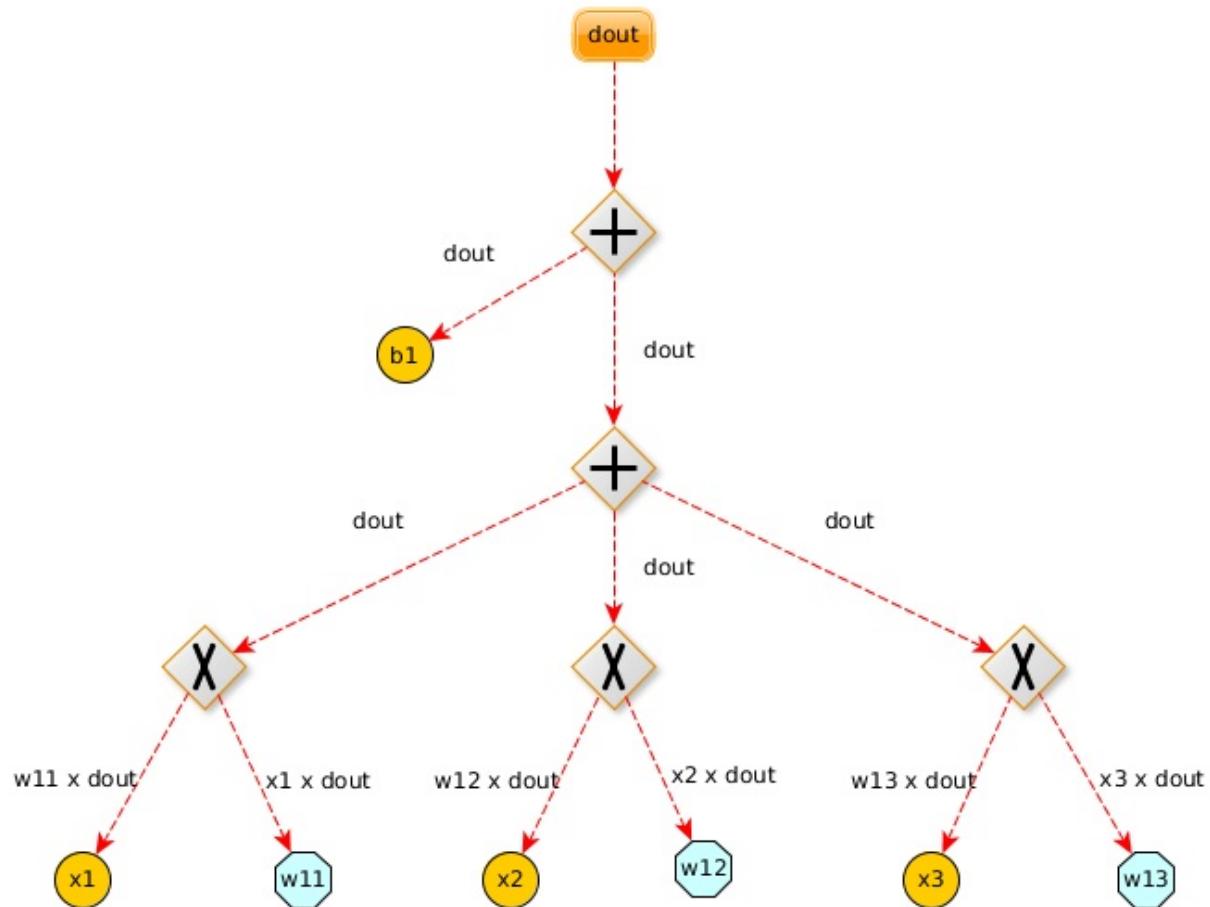
$$H(X) = (x \cdot W^t) + b$$

Computation graph point of view

In order to discover how each input influence the output (backpropagation) is better to represent the algorithm as a computation graph.



Now for the backpropagation let's focus in one of the graphs, and apply what we learned so far on backpropagation.



Summarizing the calculation for the first output (y_1), consider a global error $L(\text{loss})$ and $dout_{y_1} = \frac{\partial L}{\partial y_1}$

$$\frac{\partial L}{\partial x_1} = dout_{y_1} \cdot w_{11}$$

$$\frac{\partial L}{\partial x_2} = dout_{y_1} \cdot w_{12}$$

$$\frac{\partial L}{\partial x_3} = dout_{y_1} \cdot w_{13}$$

$$\frac{\partial L}{\partial w_{11}} = dout_{y_1} \cdot x_1$$

$$\frac{\partial L}{\partial w_{12}} = dout_{y_1} \cdot x_2$$

$$\frac{\partial L}{\partial w_{13}} = dout_{y_1} \cdot x_3$$

$$\frac{\partial L}{\partial b_1} = dout_{y_1}$$

Also extending to the second output (y_2)

$$\frac{\partial L}{\partial x_1} = dout_{y2} \cdot w21$$

$$\frac{\partial L}{\partial x_2} = dout_{y2} \cdot w22$$

$$\frac{\partial L}{\partial x_3} = dout_{y2} \cdot w23$$

$$\frac{\partial L}{\partial w_{21}} = dout_{y2} \cdot x1$$

$$\frac{\partial L}{\partial w_{22}} = dout_{y2} \cdot x2$$

$$\frac{\partial L}{\partial w_{23}} = dout_{y2} \cdot x3$$

$$\frac{\partial L}{\partial b_2} = dout_{y2}$$

Merging the results, for dx:

$$\frac{\partial L}{\partial x_1} = [dout_{y1} \cdot w11 + dout_{y2} \cdot w21]$$

$$\frac{\partial L}{\partial x_2} = [dout_{y1} \cdot w12 + dout_{y2} \cdot w22]$$

$$\frac{\partial L}{\partial x_3} = [dout_{y1} \cdot w13 + dout_{y2} \cdot w23]$$

On the matrix form

$$\frac{\partial L}{\partial X} = [dout_{y1} \quad dout_{y2}] \cdot \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix}, \text{ or}$$

$$\frac{\partial L}{\partial X} = \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix} \cdot \begin{bmatrix} dout_{y1} \\ dout_{y2} \end{bmatrix}.$$

Depending on the format that you choose to represent X (as a row or column vector), attention to this because it can be confusing.

Now for dW It's important to note that every gradient has the same dimension as its original value, for instance dW has the same dimension as W, in other words:

$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \cdot \frac{\partial L}{\partial W} = \begin{bmatrix} \frac{\partial L}{\partial w_{11}} & \frac{\partial L}{\partial w_{12}} & \frac{\partial L}{\partial w_{13}} \\ \frac{\partial L}{\partial w_{21}} & \frac{\partial L}{\partial w_{22}} & \frac{\partial L}{\partial w_{23}} \end{bmatrix}$$

$$\frac{\partial L}{\partial W} = \begin{bmatrix} dout_{y1} \\ dout_{y2} \end{bmatrix} \cdot [x_1 \quad x_2 \quad x_3] = \begin{bmatrix} \frac{\partial L}{\partial w_{11}} & \frac{\partial L}{\partial w_{12}} & \frac{\partial L}{\partial w_{13}} \\ \frac{\partial L}{\partial w_{21}} & \frac{\partial L}{\partial w_{22}} & \frac{\partial L}{\partial w_{23}} \end{bmatrix}$$

$$\text{And } d\mathbf{B} \frac{\partial L}{\partial b} = [dout_{y1} \quad dout_{y2}]$$

Expanding for bigger batches

All the examples so far, deal with single elements on the input, but normally we deal with much more than one example at a time. For instance on GPUs it is common to have batches of 256 images at the same time. The trick is to represent the input signal as a 2d matrix [NxD] where N is the batch size and D the dimensions of the input signal. So if you consider the CIFAR dataset where each digit is a 28x28x1 (grayscale) image D will be 784, so if we have 10 digits on the same batch our input will be [10x784].

For the sake of argument, let's consider our previous samples where the vector X was represented like $X = [x_1 \quad x_2 \quad x_3]$, if we want to have a batch of 4 elements we will have:

$$X_{batch} = \begin{bmatrix} x_{1sample1} & x_{2sample1} & x_{3sample1} \\ x_{1sample2} & x_{2sample2} & x_{3sample2} \\ x_{1sample3} & x_{2sample3} & x_{3sample3} \\ x_{1sample4} & x_{2sample4} & x_{3sample4} \end{bmatrix} \therefore X_{batch} = [4, 3]$$

In this case W must be represented in a way that supports this matrix multiplication, so depending on how it was created it may need to be transposed

$$W^T = \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix}$$

Continuing the forward propagation will be computed as:

$$\begin{aligned} & \left(\begin{bmatrix} x_{1sample1} & x_{2sample1} & x_{3sample1} \\ x_{1sample2} & x_{2sample2} & x_{3sample2} \\ x_{1sample3} & x_{2sample3} & x_{3sample3} \\ x_{1sample4} & x_{2sample4} & x_{3sample4} \end{bmatrix} \cdot \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix} \right) + \begin{bmatrix} b_{1sample1} & b_{2sample1} \\ b_{1sample2} & b_{2sample2} \\ b_{1sample3} & b_{2sample3} \\ b_{1sample4} & b_{2sample4} \end{bmatrix} \\ &= \begin{bmatrix} y_{1sample1} & y_{2sample1} \\ y_{1sample2} & y_{2sample2} \\ y_{1sample3} & y_{2sample3} \\ y_{1sample4} & y_{2sample4} \end{bmatrix} \end{aligned}$$

One point to observe here is that the bias has repeated 4 times to accommodate the product $X.W$ that in this case will generate a matrix [4x2]. On matlab the command "repmat" does the job. On python it does automatically.

```
>> b = [1 2]

b =
1     2

>> repmat(b,[4 1])

ans =
1     2
1     2
1     2
1     2
```

Using Symbolic engine

Before jumping to implementation is good to verify the operations on Matlab or Python (sympy) symbolic engine. This will help visualize and explore the results before acutally coding the functions.

Symbolic forward propagation on Matlab

Here after we defined the variables which will be symbolic, we create the matrix W,X,b then calculate $y = (W \cdot X) + b$, compare the final result with what we calculated before.

```
>> syms x1 x2 x3 w11 w12 w13 w21 w22 w23 b1 b2
X = [x1; x2; x3];
W = [w11 w12 w13; w21 w22 w23];
b = [b1; b2];
Y = (W*X)+b;

pretty(W);
pretty(X);
pretty(y);
/ w11, w12, w13 \
|           |
\ w21, w22, w23 /

/ x1 \
|   |
| x2 |
|   |
\ x3 /

/ b1 + w11 x1 + w12 x2 + w13 x3 \
|           |
\ b2 + w21 x1 + w22 x2 + w23 x3 /
```

Symbolic backward propagation on Matlab

Now we also confirm the backward propagation formulas. Observe the function "latex" that convert an expression to latex on matlab

```
>> syms x1 x2 x3 douty1 douty2 w11 w12 w13 w21 w22 w23
dW = [douty1; douty2] * [x1 x2 x3];
dX = [douty1 douty2] * [w11 w12 w13; w21 w22 w23];
>> pretty(dW)
/ douty1 x1, douty1 x2, douty1 x3 \
| |
\ douty2 x1, douty2 x2, douty2 x3 /

>> pretty(dX)
(douty1 w11 + douty2 w21, douty1 w12 + douty2 w22, douty1 w13 + douty2 w23)

>> latex(dW)

ans =

\left(\begin{array}{ccc} \mathrm{douty1}\mathrm{x1} & \mathrm{douty1}\mathrm{x2} & \mathrm{douty1}\mathrm{x3} \\ \mathrm{douty2}\mathrm{x1} & \mathrm{douty2}\mathrm{x2} & \mathrm{douty2}\mathrm{x3} \end{array}\right)
```

Here I've just copy and paste the latex result of dW or " $\frac{\partial L}{\partial W}$ " from matlab

$$\begin{pmatrix} \mathrm{douty1}\mathrm{x1} & \mathrm{douty1}\mathrm{x2} & \mathrm{douty1}\mathrm{x3} \\ \mathrm{douty2}\mathrm{x1} & \mathrm{douty2}\mathrm{x2} & \mathrm{douty2}\mathrm{x3} \end{pmatrix}$$

Input Tensor

Our library will be handling images, and most of the time we will be handling matrix operations on hundreds of images at the same time. So we must find a way to represent them, here we will represent batch of images as a 4d tensor, or an array of 3d matrices. Below we have a batch of 4 rgb images (width:160, height:120). We're going to load them on matlab/python and organize them one a 4d matrix



Observe that in matlab the image becomes a matrix 120x160x3. Our tensor will be 120x160x3x4

Command Window

Name ↗	Value
airplane	120x160x3 uint8
cat	120x160x3 uint8
dog	120x160x3 uint8
duck	120x160x3 uint8
inputTensor	4-D double

```
New to MATLAB? See resources for Getting Started.
>> cat = imread('/home/leo/Pictures/Cat.jpg');
>> dog = imread('/home/leo/Pictures/Dog.jpg');
>> duck = imread('/home/leo/Pictures/Duck.jpg');
>> airplane = imread('/home/leo/Pictures/Airplane.jpg');
>> inputTensor = zeros(120,160,3,4);
>> inputTensor(:,:,:,:1) = cat;
>> inputTensor(:,:,:,:2) = dog;
>> inputTensor(:,:,:,:3) = duck;
>> inputTensor(:,:,:,:4) = airplane;
fx >> |
```

On Python before we store the image on the tensor we do a transpose to convert our image 120x160x3 to 3x120x160, then to store on a tensor 4x3x120x160

Name	Type	Size	Value
airplane	uint8	(120, 160, 3)	array([[[48, 84, 146], [48, 84, 146], ...])
cat2	uint8	(120, 160, 3)	array([[[27, 53, 54], [26, 54, 55], ...])
cat_trans	uint8	(3, 120, 160)	array([[[27, 26, 25, ..., 46, 45, 49], [28, 26, 25, ..., 53, ...], ...])
dog	uint8	(120, 160, 3)	array([[[121, 198, 82], [122, 200, 78], ...])
duck	uint8	(120, 160, 3)	array([[[72, 83, 40], [72, 83, 40], ...])
input_tensor	float64	(4, 3, 120, 160)	array([[[[27., 26., 25., ..., 46., 45., 49.], [28., 26. ...], ...]])

```
In [94]: from scipy.misc import imread
In [95]: cat = imread('/home/leo/Pictures/Cat.jpg');
...: dog = imread('/home/leo/Pictures/Dog.jpg');
...: duck = imread('/home/leo/Pictures/Duck.jpg');
...: airplane = imread('/home/leo/Pictures/Airplane.jpg');
...:
In [96]: input_tensor = np.zeros((4, 3, 120, 160))
In [97]: cat_trans = cat.transpose((2, 0, 1))
In [98]: input_tensor[0, :, :, :] = cat_trans
...: input_tensor[1, :, :, :] = dog.transpose((2, 0, 1))
...: input_tensor[2, :, :, :] = duck.transpose((2, 0, 1))
...: input_tensor[3, :, :, :] = airplane.transpose((2, 0, 1))
...:
```

Python Implementation

Forward Propagation

```
1 import numpy as np
2
3 def fc_forward(x,w,b):
4     """
5         Computes the forward pass for an affine (fully-connected) layer.
6
7         Inputs:
8         - x: Input Tensor (N, d_1, ..., d_k)
9         - w: Weights (D, M)
10        - b: Bias (M,)
11
12        N: Mini-batch size
13        M: Number of outputs of fully connected layer
14        D: Input dimension
15
16        Returns a tuple of:
17        - out: output, of shape (N, M)
18        - cache: (x, w, b)
19        """
20        out = None
21
22        # Get batch size (first dimension)
23        N = x.shape[0]
24
25        # Reshape activations to [Nx(d_1, ..., d_k)], which will be a 2d matrix
26        # [NxD]
27        reshaped_input = x.reshape(N, -1)
28
29        # Calculate output
30        out = np.dot(reshaped_input,w) + b.T
31
32        # Save inputs for backward propagation
33        cache = (x,w,b)
34        return out, cache
```

Backward Propagation

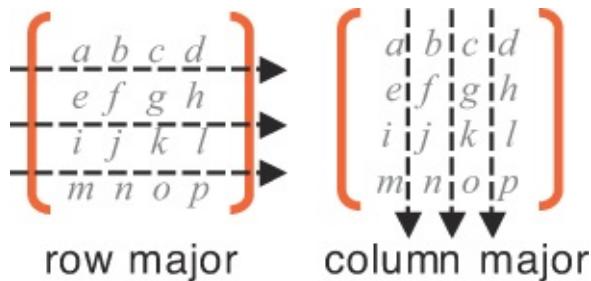
```

39     def fc_backward(dout, cache):
40         """
41             Computes the backward pass for an affine (fully-connected) layer.
42
43             Inputs:
44             - dout: Layer partial derivative wrt loss of shape (N, M) (Same as output)
45             - cache: (x,w,b) inputs from previous forward computation
46
47             N: Mini-batch size
48             M: Number of outputs of fully connected layer
49             D: Input dimension
50             d_1, ..., d_k: Single input dimension
51
52             Returns a tuple of:
53             - dx: Gradient with respect to x, of shape (N, d1, ..., dk)
54             - dw: Gradient with respect to w, of shape (D, M)
55             - db: Gradient with respect to b, of shape (M,)
56         """
57         x, w, b = cache
58         dx, dw, db = None, None, None
59
60         # Get batch size (first dimension)
61         N = x.shape[0]
62
63         # Get dX (Same format as x)
64         dx = np.dot(dout, w.T)
65         dx = dx.reshape(x.shape)
66
67         # Get dw (Same format as w)
68         # Reshape activations to [Nx(d_1, ..., d_k)], which will be a 2d matrix
69         # [NxD]
70         reshaped_input = x.reshape(N, -1)
71         # Transpose then dot product with dout
72         dw = reshaped_input.T.dot(dout)
73
74         # Get db (Same format as b)
75         db = np.sum(dout, axis=0)
76
77         # Return outputs
78         return dx, dw, db

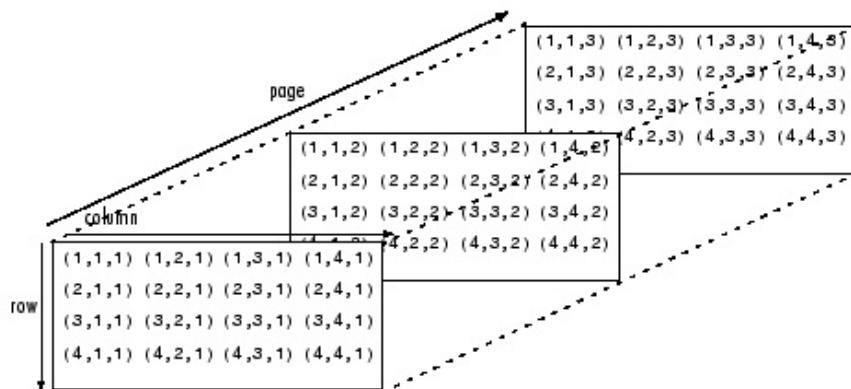
```

Matlab Implementation

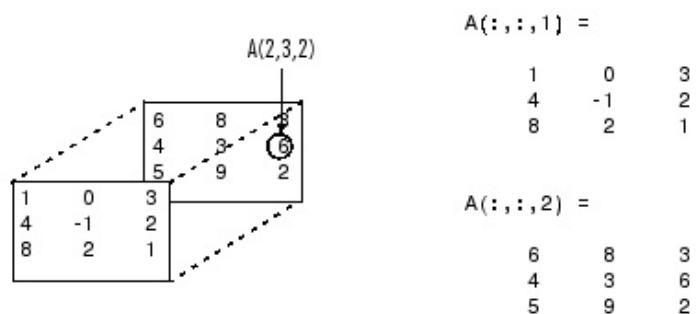
One special point to pay attention is the way that matlab represent high-dimension arrays in contrast with matlab. Also another point that may cause confusion is the fact that matlab represent data on col-major order and numpy on row-major order.



Multidimensional arrays in python and matlab



To access the element in the second row, third column of page 2, for example, you use the subscripts $(2, 3, 2)$.



One difference on how matlab and python represent multidimensional arrays must be noticed. We want to create a 4 channel matrix 2×3 . So in matlab you need to create a array $(2,3,4)$ and on python it need to be $(4,2,3)$

The image shows two side-by-side code editors. On the left is the MATLAB Command Window, and on the right is the IPython console.

MATLAB Command Window:

```
>> ones(2,3,4)

ans(:,:,1) =
1 1 1
1 1 1

ans(:,:,2) =
1 1 1
1 1 1

ans(:,:,3) =
1 1 1
1 1 1

ans(:,:,4) =
1 1 1
1 1 1
```

IPython console:

```
In [27]: import numpy as np
In [28]: np.ones((2,3,4))
Out[28]:
array([[[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]],

      [[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]]])

In [29]: np.ones((4,2,3))
Out[29]:
array([[[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]],

      [[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]],

      [[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]],

      [[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]]])
```

Matlab Reshape order

As mentioned before matlab will run the command reshape one column at a time, so if you want to change this behavior you need to transpose first the input matrix.

Fully Connected Layer

```
>> A = [1 2 3; 4 5 6]
```

```
A =
```

```
1     2     3  
4     5     6
```

```
>> reshape(A,6,[])
```

```
ans =
```

```
1  
4  
2  
5  
3  
6
```

```
>> reshape(A',6,[])
```

```
ans =
```

```
1  
2  
3  
4  
5  
6
```

If you are dealing with more than 2 dimensions you need to use the "permute" command to transpose. Now on Python the default of the reshape command is one row at a time, or if you want you can also change the order (This options does not exist in matlab)

IPython console

In [64]: A = np.array([[1,2,3], [4,5,6]])

In [65]: print(A)

```
[[1 2 3]
 [4 5 6]]
```

In [66]: A_res = A.reshape(6,-1)

In [67]: print(A_res)

```
[[1]
 [2]
 [3]
 [4]
 [5]
 [6]]
```

In [68]: A_res = A.reshape([6,-1],order='F')

In [69]: print(A_res)

```
[[1]
 [4]
 [2]
 [5]
 [3]
 [6]]
```

In [70]: A_res = np.reshape(A,6,-1)

In [71]: print(A_res)

```
[1 4 2 5 3 6]
```

Bellow we have a reshape on the row-major order as a new function:

```
reshape_row_major.m
```

```
function [ reshaped_matrix ] = reshape_row_major( matrix_in, shape )
%RESHAPE_ROW_MAJOR Do the reshape with row_major scan
% This is needed because other numerical libraries like numpy used
% different order (row_major) compared to matlab(col_major)
% Example:
% a = [1:1:15]
% reshape(a,[3,5])
%   1     4     7    10    13
%   2     5     8    11    14
%   3     6     9    12    15
%
% reshape_row_major(a,[3,5])
%   1     2     3     4     5
%   6     7     8     9    10
%   11    12    13    14    15
%
% Reshape with the shape inverted
res_trans = reshape(matrix_in,fliplr(shape));
%
% Transpose res_trans permuting all it's dimensions
reshaped_matrix = permute(res_trans,[ndims(res_trans):-1:1]);
end
```

The other option would be to avoid this permutation reshape is to have the weight matrix on a different order and calculate the forward propagation like this:

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \cdot \begin{bmatrix} x_{1sample1} & x_{1sample2} \\ x_{2sample1} & x_{2sample2} \\ x_{3sample1} & x_{3sample2} \end{bmatrix} + \begin{bmatrix} b_1 & b_1 \\ b_2 & b_2 \end{bmatrix} = \begin{bmatrix} y_{1sample1} & y_{1sample2} \\ y_{2sample1} & y_{2sample2} \end{bmatrix}$$

With x as a column vector and the weights organized row-wise, on the example that is presented we keep using the same order as the python example.

Forward Propagation

```
fc_forward.m  x  +
1 function [out, cache] = fc_forward(x,w,b)
2 -     cache = {x,w,b};
3 -     % Get the batchsize
4 -     lenSizeActivations = length(size(x));
5 -     szX = size(x);
6 -     if (lenSizeActivations < 3)
7 -         N = szX(1);
8 -     else
9 -         N = size(x,ndims(x));
10 -    end
11
12    % Reshape activations to [Nx(d_1, ..., d_k)], which will be a 2d matrix
13    % [NxD]
14 -    D = prod(szX(1:end-1));
15 -    res_x = reshape_row_major(x,[N, D]);
16
17 -    out = (res_x*w) + (repmat(b,size(res_x,1),1));
18 -    end
```

Backward Propagation

```
fc_backward.m x +
1 function [ dx, dw, db ] = fc_backward( dout, cache )
2 - x = cache{1}; w = cache{2}; b = cache{3};
3
4 % Get the batchsize
5 - lenSizeActivations = length(size(x));
6 - szX = size(x);
7 - if (lenSizeActivations < 3)
8 -     N = szX(1);
9 - else
10 -    N = size(x,ndims(x));
11 - end
12
13 % Get dx (Same format as x)
14 - dx = dout * w';
15 - szX = size(x);
16 - dx = reshape_row_major(dx,szX);
17
18 % Get dw (Same format as w)
19 % Reshape activations to [Nx(d_1, ..., d_k)], which will be a 2d matrix
20 % [NxD]
21 - szX = size(x);
22 - D = prod(szX(1:end-1));
23 - res_x = reshape_row_major(x,[N, D]);
24 - dw = res_x' * dout;
25
26 % Get db (Same format as x)
27 % Sum all columns of dout
28 - db = sum(dout,1);
29
30 - end
```

Next Chapter

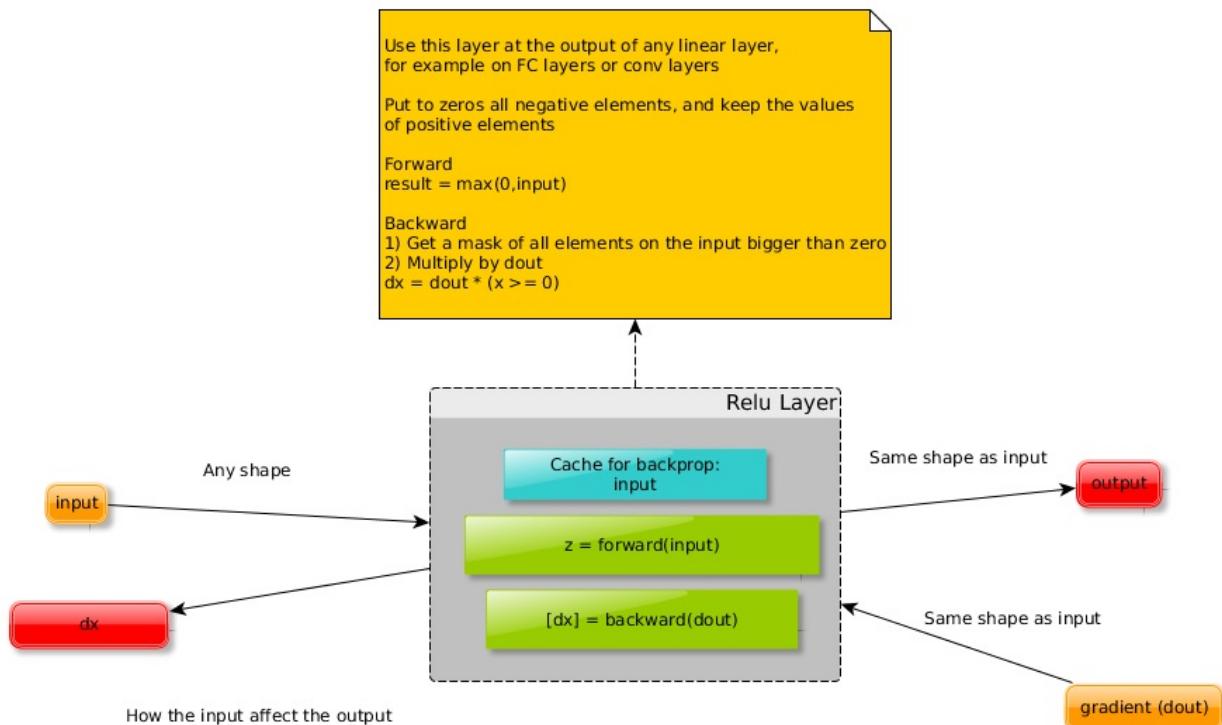
Next chapter we will learn about Relu layers

Relu Layer

Rectified-Linear unit Layer

Introduction

We will start this chapter explaining how to implement in Python/Matlab the ReLU layer.



In simple words, the ReLU layer will apply the function $f(x) = \max(0, x)$ in all elements on a input tensor, without changing it's spatial or depth information.

```
>> A = randn(4,4)

A =

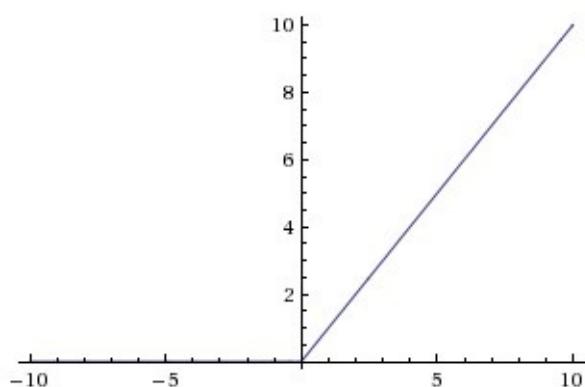
 -0.1961    1.5877   -0.2437    0.1049
 1.4193   -0.8045    0.2157    0.7223
 0.2916    0.6966   -1.1658    2.5855
 0.1978    0.8351   -1.1480   -0.6669

>> relu_A = max(0,A)

relu_A =

      0    1.5877        0    0.1049
 1.4193        0    0.2157    0.7223
 0.2916    0.6966        0    2.5855
 0.1978    0.8351        0        0
```

f(x) >> |



From the picture above, observe that all positive elements remain unchanged while the negatives become zero. Also the spatial information and depth are the same.

Thinking about neural networks, it's just a new type of Activation function, but with the following features:

1. Easy to compute (forward/backward propagation)
2. Suffer much less from vanishing gradient on deep models
3. A bad point is that they can irreversibly die if you use a big learning rate

Forward propagation

Change all negative elements to zero while retaining the value of the positive elements. No spatial/depth information is changed.

Python forward propagation

```

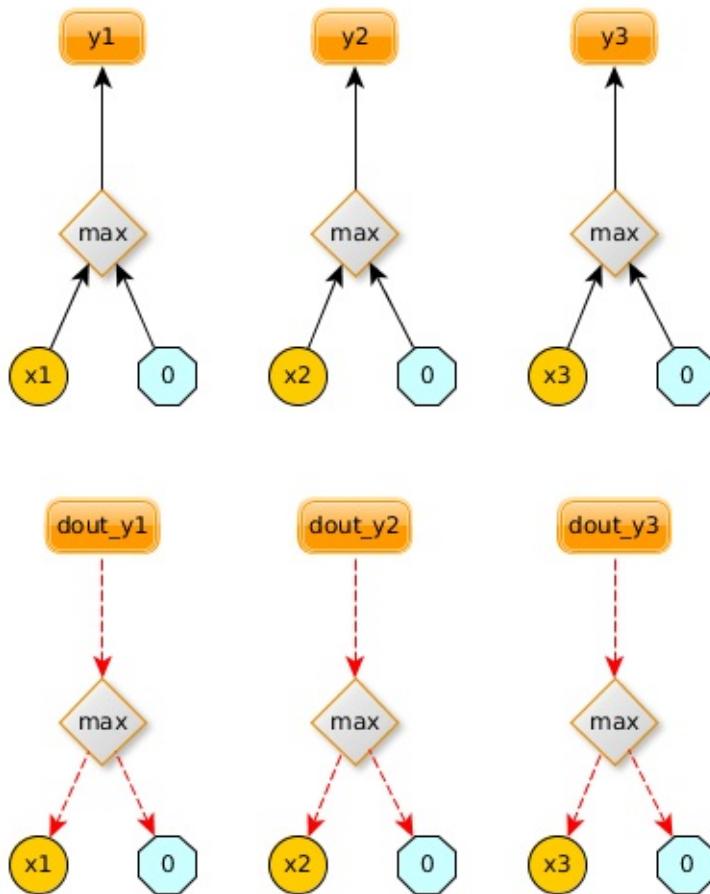
3  def relu_forward(x):
4      """
5          Computes the forward pass for ReLU
6          Input:
7              - x: Inputs, of any shape
8
9          Returns a tuple of: (out, cache)
10         The shape on the output is the same as the input
11         """
12     out = None
13
14     # Create a function that receive x and return x if x is bigger
15     # than zero, or zero if x is negative
16     relu = lambda x: x * (x > 0).astype(float)
17     out = relu(x)
18
19     # Cache input and return outputs
20     cache = x
21     return out, cache

```

Matlab forward propagation

Backward propagation

Basically we're just applying the $\max(0,x)$ function to every $X = [x_1, x_2, x_3]$ input element. From the back-propagation chapter we can notice that the gradient dx will be zero if the element x_n is negative or $dout_n$ if the element is positive.



Python backward propagation

```

3  def relu_backward(dout, cache):
4      """
5          Computes the backward pass for ReLU
6          Input:
7              - dout: Upstream derivatives, of any shape
8              - cache: Previous input (used on forward propagation)
9
10     Returns:
11         - dx: Gradient with respect to x
12     """
13
14     # Initialize dx with None and x with cache
15     dx, x = None, cache
16
17     # Make all positive elements in x equal to dout while all the other elements
18     # Become zero
19     dx = dout * (x >= 0)
20
21     # Return dx (gradient with respect to x)
22     return dx

```

Next Chapter

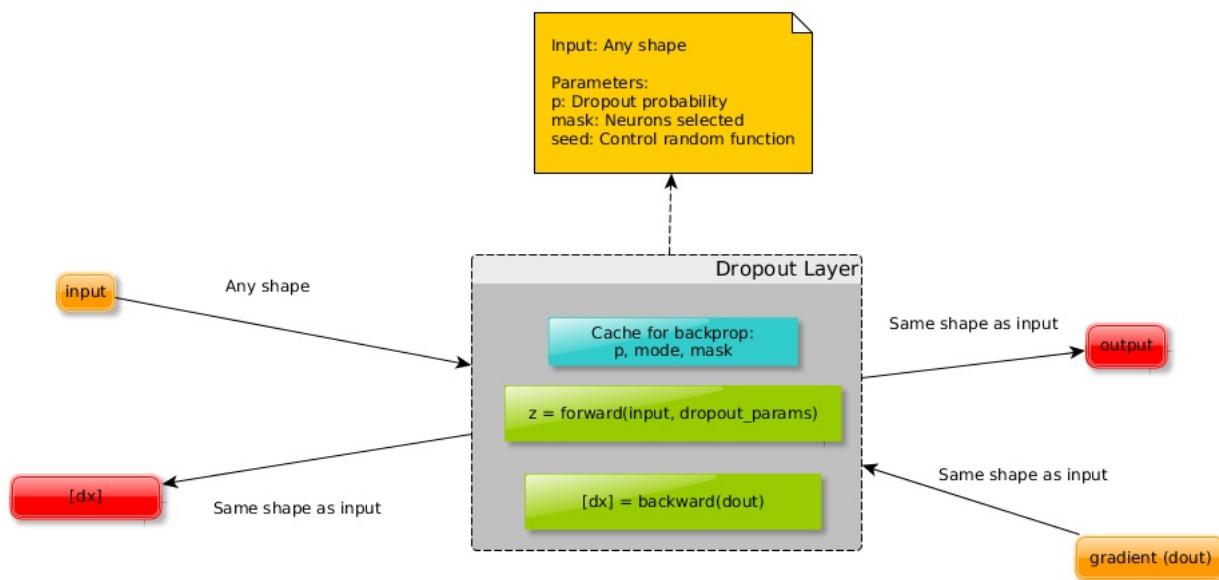
Next chapter we will learn about Dropout layers

Dropout Layer

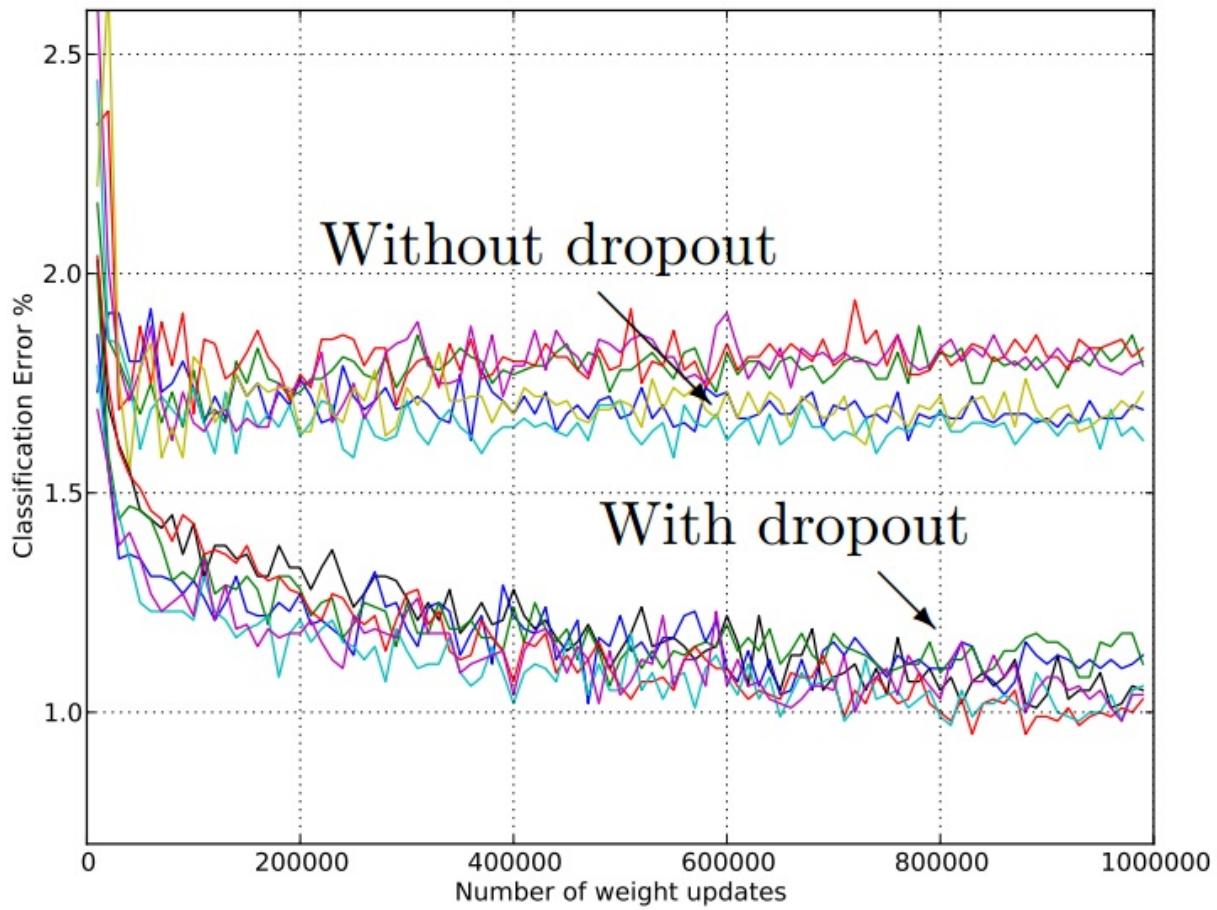
Dropout Layer

Introduction

Dropout is a technique used to improve over-fit on neural networks, you should use Dropout along with other techniques like L2 Regularization.

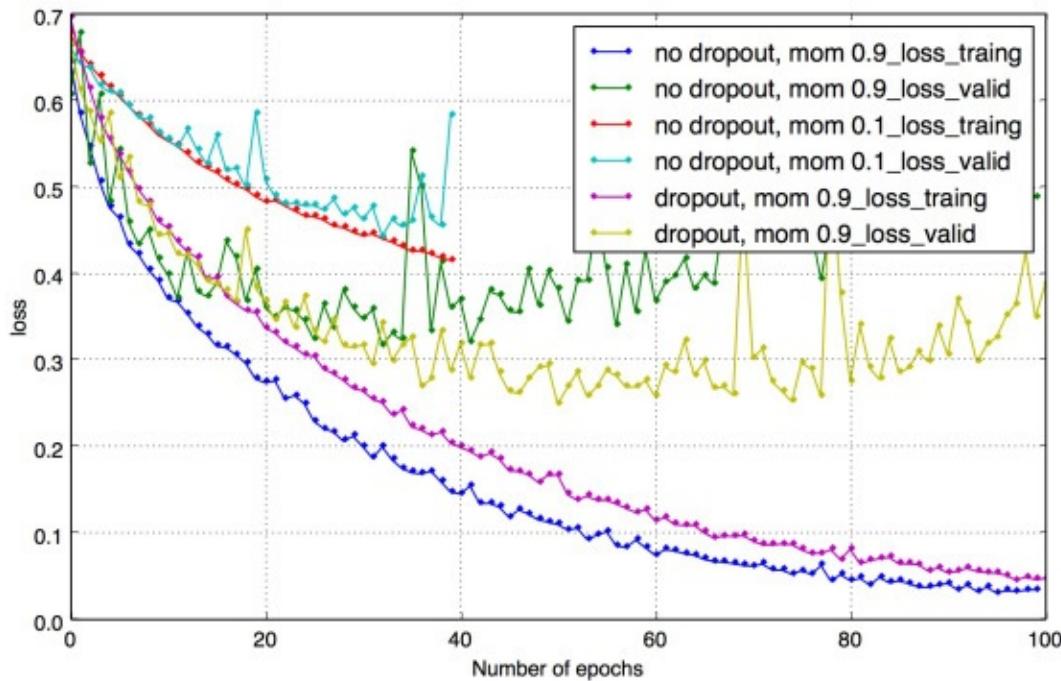


Bellow we have a classification error (Not including loss), observe that the test/validation error is smaller using dropout



As other regularization techniques the use of dropout also make the training loss error a little worse. But that's the idea, basically we want to trade training performance for more generalization. Remember that's more capacity you add on your model (More layers, or more neurons) more prone to over-fit it becomes.

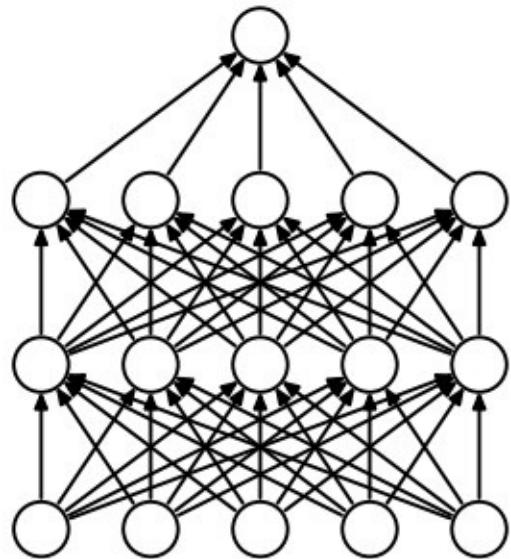
Bellow we have a plot showing both training, and validation loss with and without dropout



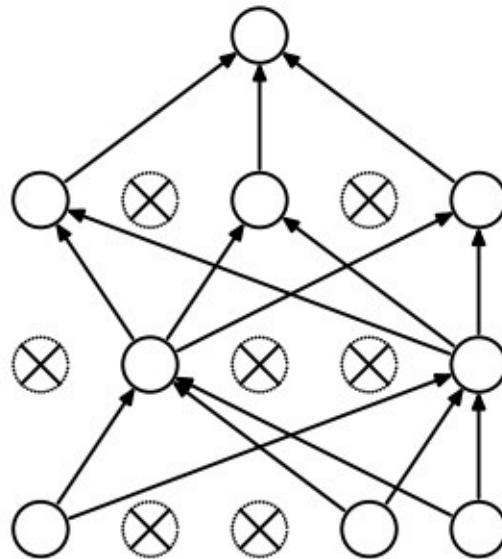
How it works

Basically during training half of neurons on a particular layer will be deactivated. This improve generalization because force your layer to learn with different neurons the same "concept".

During the prediction phase the dropout is deactivated.



(a) Standard Neural Net



(b) After applying dropout.

Where to use Dropout layers

Normally some deep learning models use Dropout on the fully connected layers, but is also possible

to use dropout after the max-pooling layers, creating some kind of image noise augmentation.

Implementation

In order to implement this neuron deactivation, we create a mask(zeros and ones) during forward propagation. This mask is applied to the layer outputs during training and cached for future use on back-propagation. As explained before this dropout mask is used only during training.

On the backward propagation we're interested on the neurons that was activated (we need to save mask from forward propagation). Now with those neurons selected we just back-propagate dout. The dropout layer has no learnable parameters, just it's input (X). During back-propagation we just return "dx". In other words: $dx = dout \cdot mask_{cached}$

Python Forward propagation

```

3  def dropout_forward(x, dropout_param):
4      """
5          Performs the forward pass for (inverted) dropout.
6          Inputs:
7              - x: Input data, of any shape
8              - dropout_param: A dictionary with the following keys: (p,test/train,seed)
9          Outputs: (out, cache)
10         """
11        # Get the current dropout mode, p, and seed
12        p, mode = dropout_param['p'], dropout_param['mode']
13        if 'seed' in dropout_param:
14            np.random.seed(dropout_param['seed'])
15
16        # Initialization of outputs and mask
17        mask = None
18        out = None
19
20        if mode == 'train':
21            # Create an apply mask (normally p=0.5 for half of neurons), we scale all
22            # by p to avoid having to multiply by p on backpropagation, this is called
23            # inverted dropout
24            mask = (np.random.rand(*x.shape) < p) / p
25            # Apply mask
26            out = x * mask
27        elif mode == 'test':
28            # During prediction no mask is used
29            mask = None
30            out = x
31
32        # Save mask and dropout parameters for backpropagation
33        cache = (dropout_param, mask)
34
35        # Convert "out" type and return output and cache
36        out = out.astype(x.dtype, copy=False)
37        return out, cache

```

Python Backward propagation

```
3  def dropout_backward(dout, cache):
4      """
5          Perform the backward pass for (inverted) dropout.
6          Inputs:
7          - dout: Upstream derivatives, of any shape
8          - cache: (dropout_param, mask) from dropout_forward.
9      """
10     # Recover dropout parameters (p, mask , mode) from cache
11     dropout_param, mask = cache
12     mode = dropout_param['mode']
13
14     dx = None
15     # Back propagate (Dropout layer has no parameters just input X)
16     if mode == 'train':
17         # Just back propagate dout from the neurons that were used during dropout
18         dx = dout * mask
19     elif mode == 'test':
20         # Disable dropout during prediction/test
21         dx = dout
22
23     # Return dx
24     return dx
```

Next Chapter

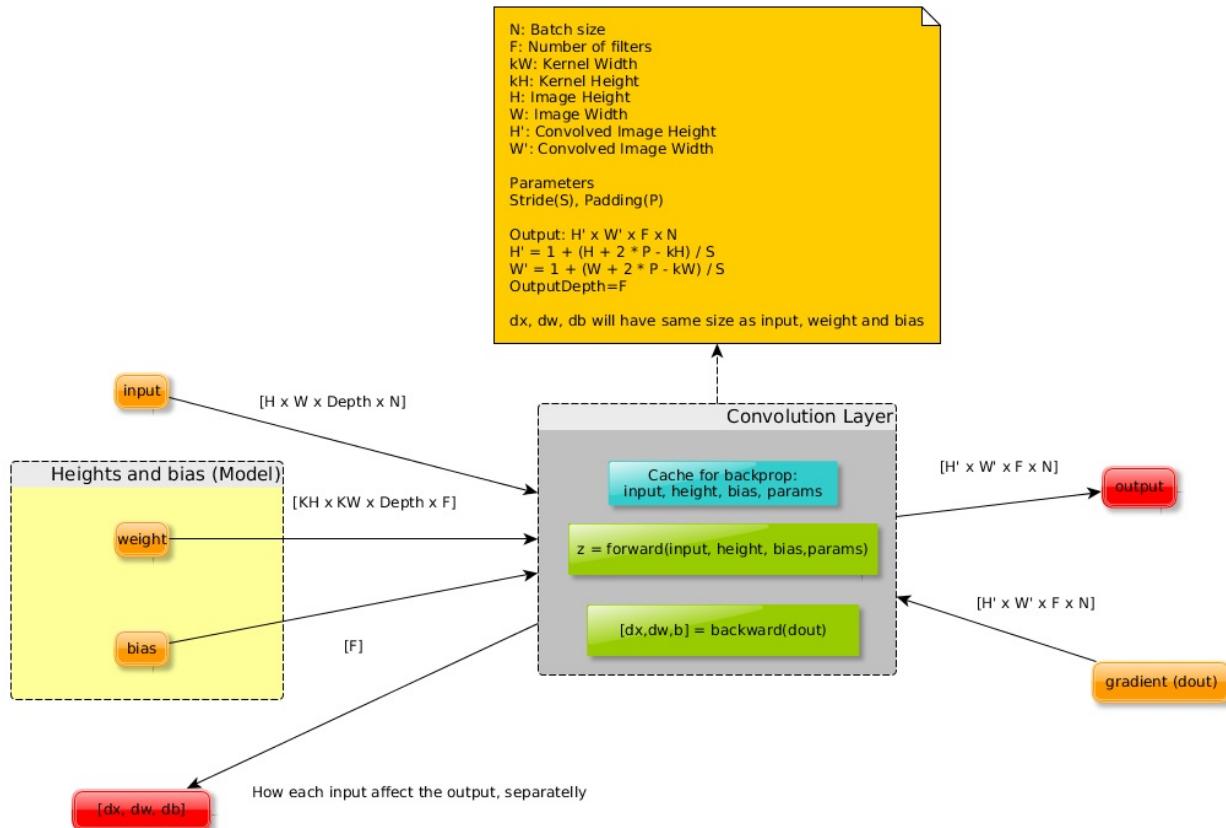
Next chapter we will learn about Convolution layer

Convolution Layer

Convolution Layer

Introduction

This chapter will explain how to implement the convolution layer on python and matlab.

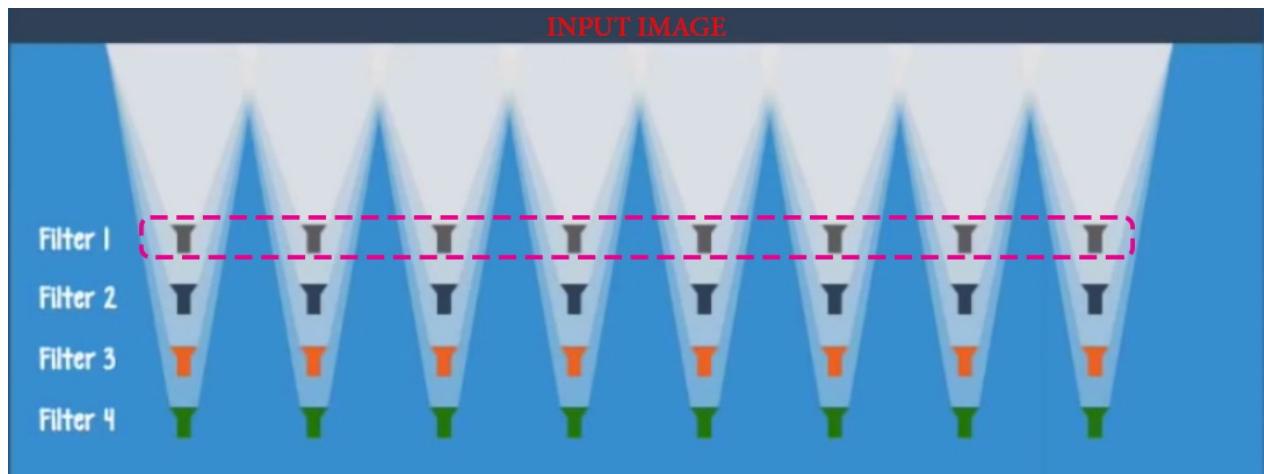


In simple terms the convolution layer, will apply the convolution operator on all images on the input tensor, and also transform the input depth to match the number of filters. Below we explain it's parameters and signals:

1. N: Batch size (Number of images on the 4d tensor)
2. F: Number of filters on the convolution layer
3. kW/kH: Kernel Width/Height (Normally we use square images, so kW=kH)
4. H/W: Image height/width (Normally H=W)
5. H'/W': Convolved image height/width (Remains the same as input if proper padding is used)
6. Stride: Number of pixels that the convolution sliding window will travel.
7. Padding: Zeros added to the border of the image to keep the input and output size the same.
8. Depth: Volume input depth (ie if the input is a RGB image depth will be 3)
9. Output depth: Volume output depth (same as F)

Forward propagation

On the forward propagation, you must remember, that we're going to "convolve" each input depth with a different filter, and each filter will look for something different on the image.



Here observe that all neurons(flash-lights) from layer 1 share the same set of weights, other filters will look for different patterns on the image.

Matlab Forward propagation

Basically we can consider the previous "convn_vanilla" function on the [Convolution chapter](#) and apply for each depth on the input and output.

```

conv_forward.m  x [ + ]
1  function [out, cache] = conv_forward(x,w,b,params)
2
3  % N (input volume), F(output volume)
4  % C channels
5  % H (rows), W(cols)
6  [H, W, C, N] = size(x);
7  [HH, WW, C, F] = size(w);
8
9  % Calculate output size, and allocate result
10 - H_R = ((H + (2*params.numPad) - HH) / params.stepStride) + 1;
11 - W_R = ((W + (2*params.numPad) - WW) / params.stepStride) + 1;
12 - out = zeros(H_R,W_R,F,N);
13
14  % Pad if needed
15 - if (params.numPad > 0)
16 -     x = padarray(x,[params.numPad params.numPad 0 0]);
17 - end
18
19  % Convolve for each input/output depth
20 - for idxBatch=1:N
21 -     for idxFilter=1:F
22
23         % Select weights and inputs
24 -         weights = w(:,:,:,:,idxFilter);
25 -         input = x(:,:,:,:,idxBatch);
26
27         % Do naive(slow) convolution
28 -         resConv = convn_vanilla(input,weights,params.stepStride);
29
30         % Add bias and store
31 -         out(:,:,idxFilter,idxBatch) = resConv + b(idxFilter);
32 -     end
33 - end
34
35  % Cache parameters and inputs for backpropagation
36 - cache = {x,w,b,params};
37 - end
38

```

Python Forward propagation

The only point to observe here is that due to the way the multidimensional arrays are represented in python our tensors will have different order.

```

80     def conv_forward_naive(x, w, b, conv_param):
81         """
82             Computes the forward pass for the Convolution layer. (Naive)
83             Input:
84             - x: Input data of shape (N, C, H, W)
85             - w: Filter weights of shape (F, C, HH, WW)
86             - b: Biases, of shape (F,)
87             - conv_param: A dictionary with the following keys:
88                 - 'stride': How much pixels the sliding window will travel
89                 - 'pad': The number of pixels that will be used to zero-pad the input.
90
91             N: Mini-batch size
92             C: Input depth (ie 3 for RGB images)
93             H/W: Image height/width
94             F: Number of filters on convolution layer (Will be the output depth)
95             HH/WW: Kernel Height/Width
96
97             Returns a tuple of:
98             - out: Output data, of shape (N, F, H', W') where H' and W' are given by
99                 H' = 1 + (H + 2 * pad - HH) / stride
100                W' = 1 + (W + 2 * pad - WW) / stride
101            - cache: (x, w, b, conv_param)
102        """
103        out = None
104        N, C, H, W = x.shape
105        F, C, HH, WW = w.shape
106
107        # Get parameters
108        P = conv_param["pad"]
109        S = conv_param["stride"]
110
111        # Calculate output size, and initialize output volume
112        H_R = 1 + (H + 2 * P - HH) / S
113        W_R = 1 + (W + 2 * P - WW) / S
114        out = np.zeros((N, F, H_R, W_R))
115
116        # Pad images with zeros on the border (Used to keep spatial information)
117        x_pad = np.lib.pad(x, ((0,0),(0,0), (P,P), (P,P)), 'constant', constant_values=0)
118
119        # Apply the convolution
120        for n in xrange(N): # For each element on batch
121            for depth in xrange(F): # For each input depth
122                for r in xrange(0,H,S): # Slide vertically taking stride into account
123                    for c in xrange(0,W,S): # Slide horizontally taking stride into account
124                        out[n,depth,r/S,c/S] = np.sum(x_pad[n,:,r:r+HH,c:c+WW] * w[depth,:,:,:]) + b[depth]
125
126        # Cache parameters and inputs for backpropagation and return output volume
127        cache = (x, w, b, conv_param)
128        return out, cache

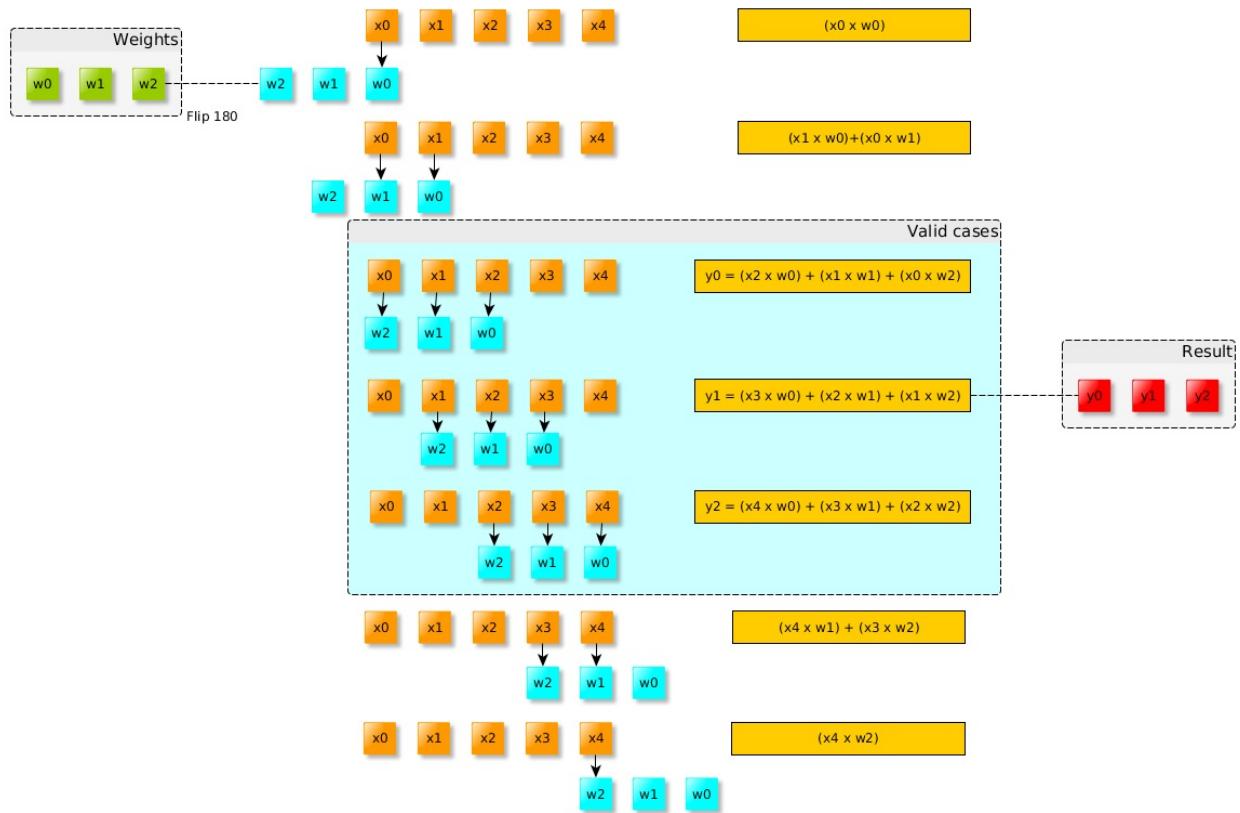
```

Back-propagation

In order to derive the convolution layer back-propagation it's easier to think on the 1d convolution, the results will be the same for 2d.

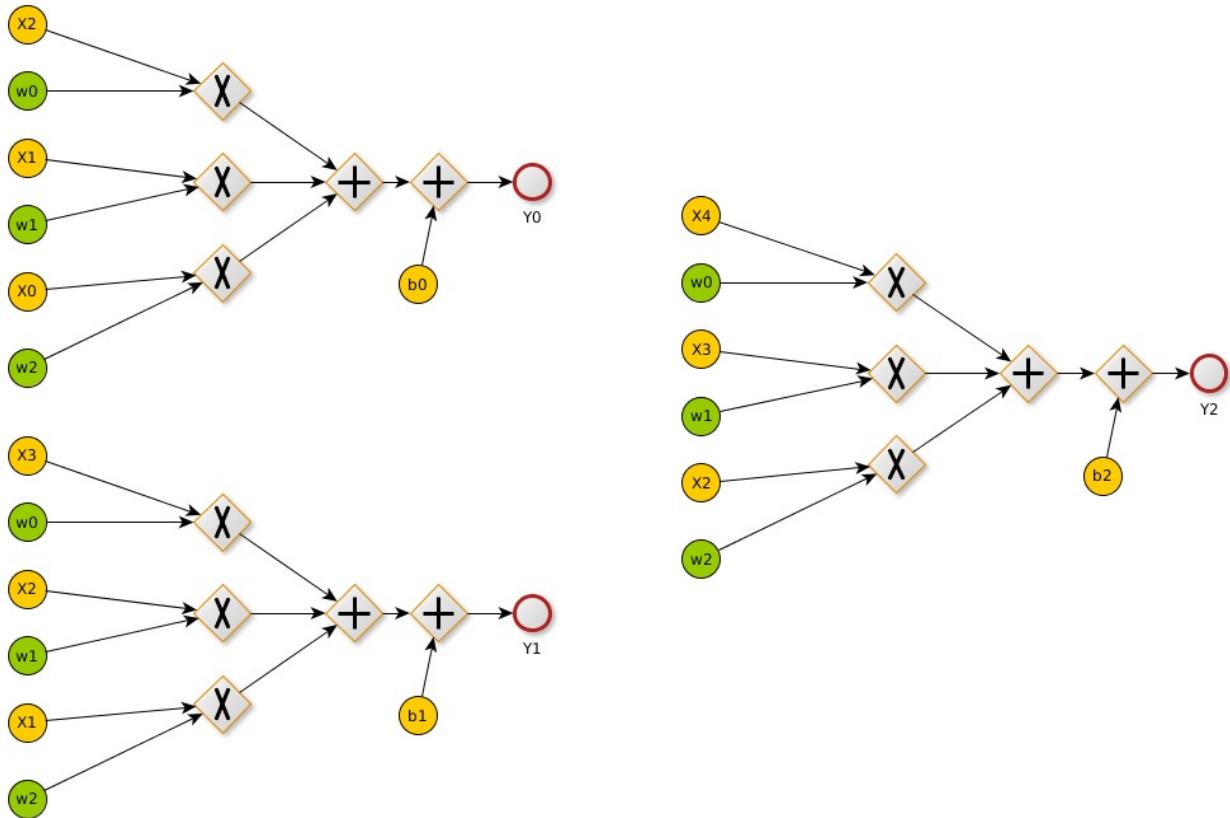
So doing a 1d convolution, between a signal $X = [x_0, x_1, x_2, x_3, x_4]$ and $W = [w_0, w_1, w_2]$, and without padding we will have $Y = [y_0, y_1, y_2]$, where

$Y = X * \text{flip}(W)$. Here flip can be consider as a 180 degrees rotation.

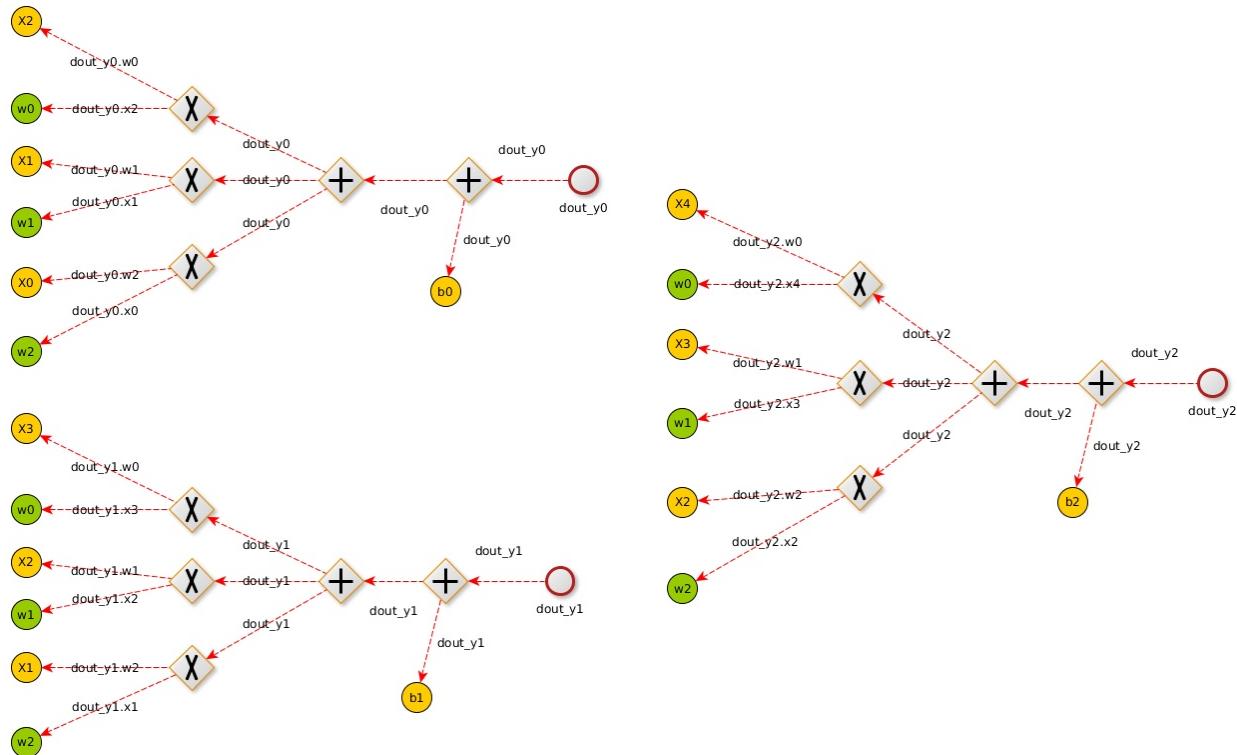


Now we convert all the "valid cases" to a computation graph, observe that for now we're adding the bias because it is used on the convolution layer.

Observe that the graphs are basically the same as the fully connected layer, the only difference is that we have shared weights.



Now changing to the back-propagation



If you follow the computation graphs backward, as was presented on the [Backpropagation chapter](#) we will have the following formulas for $\frac{\partial L}{\partial X}$, which means how the loss will change with the input X

$$\frac{\partial L}{\partial x_0} = (w2.dout_{y0})$$

$$\frac{\partial L}{\partial x_1} = (w1.dout_{y0}) + (w2.dout_{y1})$$

$$\frac{\partial L}{\partial x_2} = (w0.dout_{y0}) + (w1.dout_{y1}) + (w2.dout_{y2})$$

$$\frac{\partial L}{\partial x_3} = (w0.dout_{y1}) + (w1.dout_{y2})$$

$$\frac{\partial L}{\partial x_4} = (w0.dout_{y2})$$

Now consider some things:

1. dX must have the same size of X, so we need padding
2. dout must have the same size of Y, which in this case is 3 (Gradient input)
3. To save programming effort we want to calculate the gradient as a convolution
4. On dX gradient all elements are been multiplied by W so we're probably convolving W and dout

Following the output size rule for the 1d convolution:

$outputSize = (InputSize - KernelSize + 2P) + 1$ Our desired size is 3, our original input size is 3, and we're going to convolve with the W matrix that also have 3 elements. So we need to pad our input with 2 zeros.



The convolution above implement all calculations needed for $\frac{\partial L}{\partial X}$, so in terms of convolution:

flipped K or W

$$\frac{\partial L}{\partial X} = \underbrace{dout}_{\text{zero padded}} * \overbrace{W}^{\text{flipped K or W}}$$

Now let's continue for $\frac{\partial L}{\partial W} = [\partial w_0, \partial w_1, \partial w_2]$, considering that they must have the same size

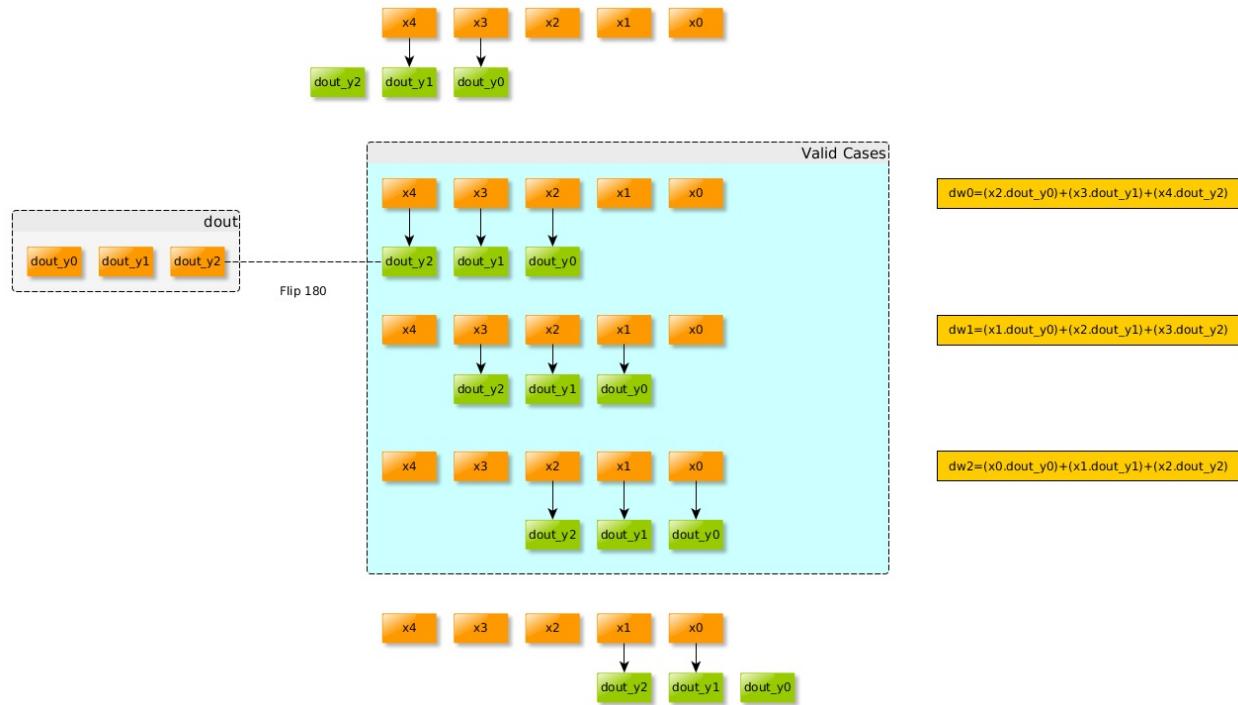
$$\frac{\partial L}{\partial w_0} = (x2.dout_{y0}) + (x3.dout_{y1}) + (x4.dout_{y2})$$

$$\frac{\partial L}{\partial w_1} = (x1.dout_{y0}) + (x2.dout_{y1}) + (x3.dout_{y2})$$

$$\frac{\partial L}{\partial w_2} = (x0.dout_{y0}) + (x1.dout_{y1}) + (x2.dout_{y2})$$

as W.

Again by just looking to the expressions that we took from the graph we can see that is possible to represent them as a convolution between dout and X. Also as the output will be 3 elements, there is no need to do padding.



$$\frac{\partial L}{\partial W} = \underbrace{\hat{X}}_{\text{flipped X}} * dout$$

So in terms of convolution the calculations for $\frac{\partial L}{\partial W}$ will be:

Just one point to remember, if you consider X to be the kernel, and dout the signal, X will be

$$\frac{\partial L}{\partial W} = dout * X$$

automatically flipped.

Now for the bias, the calculation will be similar to the Fully Connected layer. Basically we have one

155

bias per filter (depth)

$$\frac{\partial L}{\partial b} = [\sum_{batch} (dout_{y0}) \quad \sum_{batch} (dout_{y1}) \quad \sum_{batch} (dout_{y2})]$$

Implementation Notes

Before jumping to the code some points need to be reviewed:

1. If you use some parameter (ie: Stride/Pad) during forward propagation you need to apply them on the backward propagation.
2. On Python our multidimensional tensor will be "input=[N x Depth x H x W]" on matlab they will be "input=[H x W x Depth x N]"
3. As mentioned before the gradients of a input, has the same size as the input itself "size(x)==size(dx)"

Matlab Backward propagation

```

conv_backward.m  x] +
1  function [ dx, dw, db ] = conv_backward( dout, cache )
2 -    x = cache{1}; w = cache{2}; b = cache{3}; params = cache{4};
3 -    % N (Batch size), F(output volume, number of filters)
4 -    % C channels (input volume)
5 -    % H (rows), W(cols)
6 -    [H_R, W_R, F, N] = size(dout);
7 -    [H, W, C, N] = size(x);
8 -    [HH, WW, C, F] = size(w);
9 -    S = params.stepStride;
10 -   % Pad if needed
11 -   if (params.numPad > 0)
12 -     x = padarray(x,[params.numPad params.numPad 0 0]);
13 -   end
14
15 -   dx = zeros(size(x)); dw = zeros(size(w)); db = zeros(size(b));
16
17 -   % Calculate dx
18 -   for n=1:N
19 -     for depth=1:F
20 -       weights = w(:,:,:,:depth);
21 -       for r=1:S:H
22 -         for c=1:S:W
23 -           input = dout(ceil(r/S),ceil(c/S),depth,n);
24 -           prod = weights * input;
25 -           dx(r:(r+HH)-1,c:(c+WW)-1,:,:n) = dx(r:(r+HH)-1,c:(c+WW)-1,:,:n) + prod;
26 -         end
27 -       end
28 -     end
29 -   end
30 -   % Delete padded rows
31 -   dx = dx(1+params.numPad:end-params.numPad, 1+params.numPad:end-params.numPad,:,:);
32
33 -   % Calculate dw
34 -   for n=1:N
35 -     for depth=1:F
36 -       for r=1:H_R
37 -         for c=1:W_R
38 -           input = dout(r,c,depth,n);
39 -           weights = x(r*S:(r*S+HH)-1,c*S:(c*S+WW)-1,:,:n);
40 -           prod = weights * input;
41 -           dw(:,:,:,:depth) = dw(:,:,:,:depth) + prod;
42 -         end
43 -       end
44 -     end
45 -   end
46
47 -   % Calculate db
48 -   for depth=1:F
49 -     selDoutDepth = dout(:,:,depth,:);
50 -     db(depth) = sum( selDoutDepth(:) );
51 -   end
52
53 - end

```

Python Backward propagation

```

130     def conv_backward_naive(dout, cache):
131         """
132             Computes the backward pass for the Convolution layer. (Naive)
133             Inputs:
134             - dout: Upstream derivatives.
135             - cache: A tuple of (x, w, b, conv_param) as in conv_forward_naive
136             Returns a tuple of: (dw,dx,db) gradients
137         """
138         dx, dw, db = None, None, None
139         x, w, b, conv_param = cache
140         N, F, H_R, W_R = dout.shape
141         N, C, H, W = x.shape
142         F, C, HH, WW = w.shape
143         P = conv_param["pad"]
144         S = conv_param["stride"]
145         # Do zero padding on x_pad
146         x_pad = np.lib.pad(x, ((0,0),(0,0), (P,P), (P,P)), 'constant', constant_values=0)
147
148         # Initialize outputs
149         dx = np.zeros(x_pad.shape)
150         dw = np.zeros(w.shape)
151         db = np.zeros(b.shape)
152
153         # Calculate dx, with 2 extra col/row that will be deleted
154         for n in xrange(N): # For each element on batch
155             for depth in xrange(F): # For each filter
156                 for r in xrange(0,H,S): # Slide vertically taking stride into account
157                     for c in xrange(0,W,S): # Slide horizontally taking stride into account
158                         dx[n,:,r:r+HH,c:c+WW] += dout[n,depth,r/S,c/S] * w[depth,:,:,:]
159
160         #deleting padded rows to match real dx
161         delete_rows = range(P) + range(H+P,H+2*P,1)
162         delete_columns = range(P) + range(W+P,W+2*P,1)
163         dx = np.delete(dx, delete_rows, axis=2)      #height
164         dx = np.delete(dx, delete_columns, axis=3)    #width
165
166         # Calculate dw
167         for n in xrange(N): # For each element on batch
168             for depth in xrange(F): # For each filter
169                 for r in xrange(H_R): # Slide vertically taking stride into account
170                     for c in xrange(W_R): # Slide horizontally taking stride into account
171                         dw[depth,:,:,:]+= dout[n,depth,r,c] * x_pad[n,:,r*S:r*S+HH,c*S:c*S+WW]
172
173         # Calculate db, 1 scalar bias per filter, so it's just a matter of summing
174         # all elements of dout per filter
175         for depth in range(F):
176             db[depth] = np.sum(dout[:, depth, :, :])
177
178         return dx, dw, db

```

Next Chapter

Next chapter we will learn about Pooling layer

Convolution Layer

Making faster

Making faster

Introduction

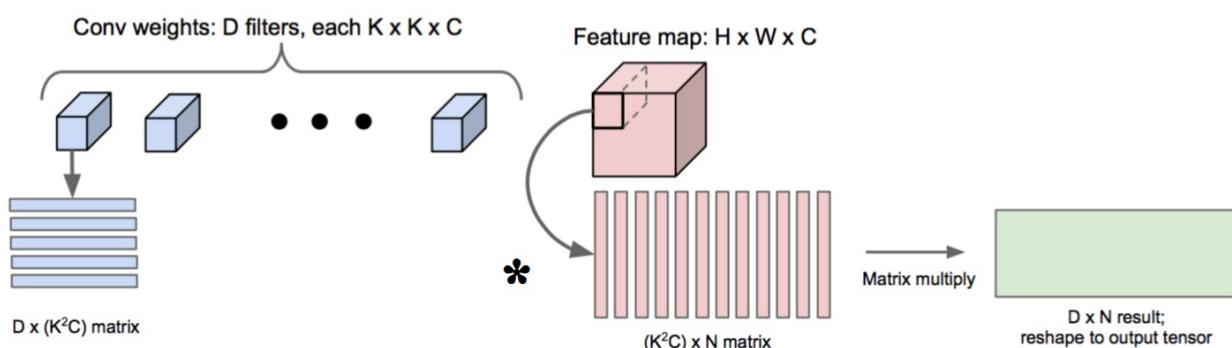
On this chapter we show a way to convert your convolution operation into a matrix multiplication. This has the advantage to compute faster, at the expense of more memory usage. We employ the **im2col** operation that will transform the input image or batch into a matrix, then we multiply this matrix with a reshaped version of our kernel. Then at the end we reshape this multiplied matrix back to an image with the **col2im** operation.

Im2col

As shown on previous source code, we use a lot for for-loops to implement the convolutions, while this is useful for learning purpose, it's not fast enough. On this section we will learn how to implement convolutions on a vectorized fashion.

First, if we inspect closer the code for convolution is basically a dot-product between the kernel filter and the local regions selected by the moving window, that sample a patch with the same size as our kernel.

What would happens if we expand all possible windows on memory and perform the dot product as a matrix multiplication. Answer 200x or more speedups, at the expense of more memory consumption.



For example, if the input is $[227 \times 227 \times 3]$ and it is to be convolved with $11 \times 11 \times 3$ filters at stride 4 and padding 0, then we would take $[11 \times 11 \times 3]$ blocks of pixels in the input and stretch each block into a column vector of size $11 * 11 * 3 = 363$.

Calculating with input 227 with stride 4 and padding 0, gives $((227-11)/4)+1 = 55$ locations along both width and height, leading to an output matrix X_{col} of size $[363 \times 3025]$.

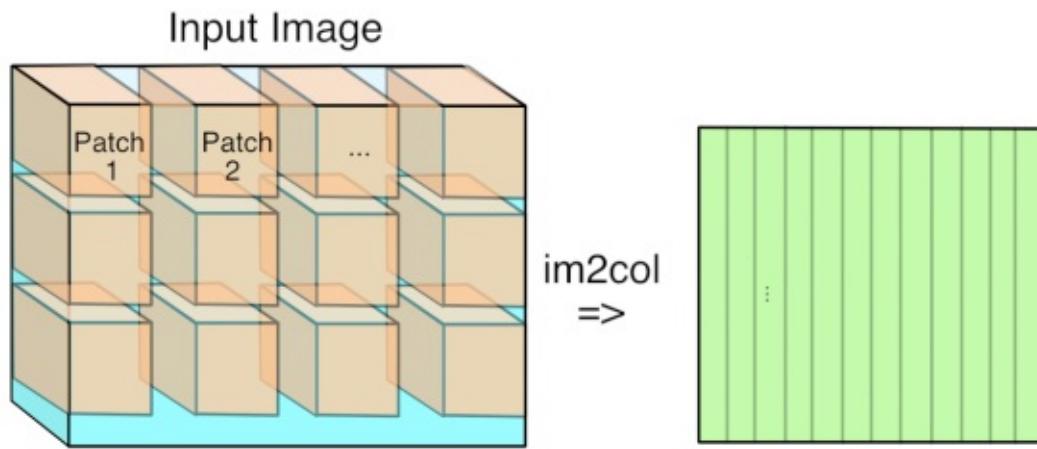
Here every column is a stretched out receptive field (patch with depth) and there are $55*55 = 3025$ of them in total.

To summarize how we calculate the im2col output sizes:

```
[img_height, img_width, img_channels] = size(img);
newImgHeight = floor(((img_height + 2*p - kszie) / s)+1);
```

```
newImgWidth = floor(((img_width + 2*P - ksize) / S)+1);
cols = single(zeros((img_channels*ksize*ksize),(newImgHeight * newImgWidth)));
```

The weights of the CONV layer are similarly stretched out into rows. For example, if there are 96 filters of size [11x11x3] this would give a matrix W_row of size [96 x 363], where $11 \times 11 \times 3 = 363$



After the image and the kernel are converted, the convolution can be implemented as a simple matrix multiplication, in our case it will be $W_{\text{col}}[96 \times 363]$ multiplied by $X_{\text{col}}[363 \times 3025]$ resulting as a matrix $[96 \times 3025]$, that need to be reshaped back to $[55 \times 55 \times 96]$.

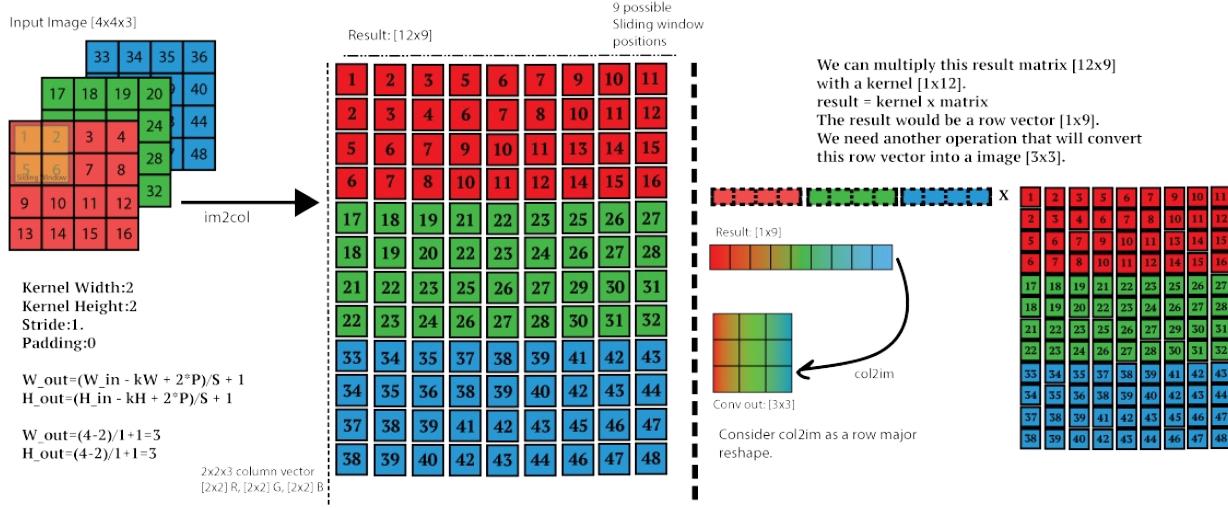
This final reshape can also be implemented as a function called col2im.

Notice that some implementations of im2col will have this result transposed, if this is the case then the order of the matrix multiplication must be changed.

Convolution Layer

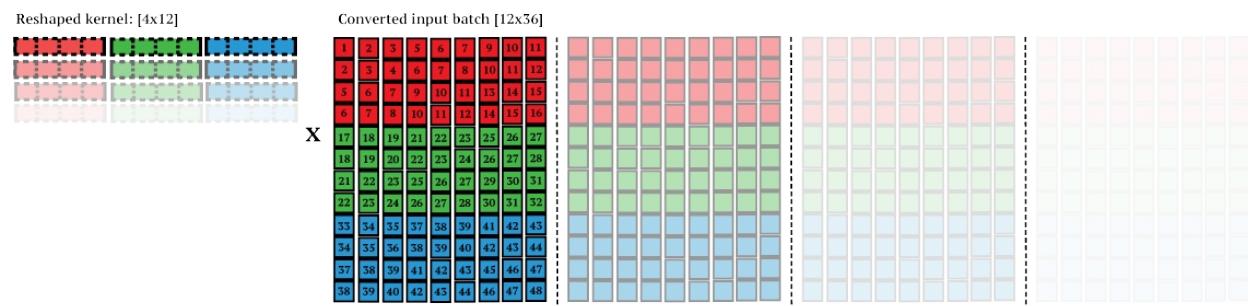
Image to column operation (im2col)

Slide the input image like a convolution but each patch become a column vector.



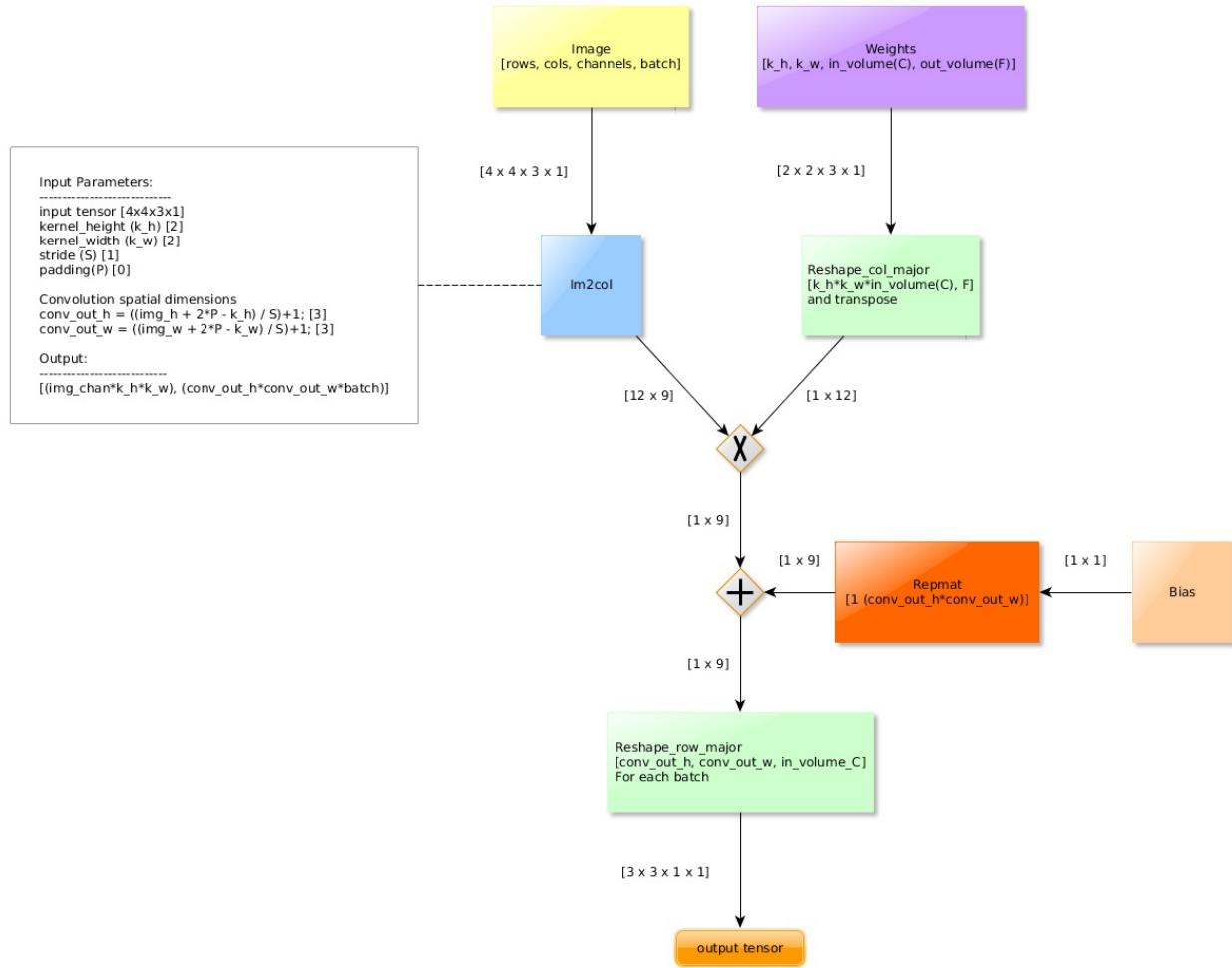
We get true performance gain

when the kernel has a large number of filters, ie: F=4 and/or you have a batch of images (N=4). Example for the input batch [4x4x5x4], convolved with 4 filters [2x2x5x2]. The only problem with this approach is the amount of memory



Forward graph

In order to help the usage of im2col with convolution and also to derive the back-propagation, let's show the convolution with im2col as a graph. Here the input tensor is single a 3 channel 4x4 image. That will pass to a convolution layer with S:1 P:0 K:2 and F:1 (Output volume).

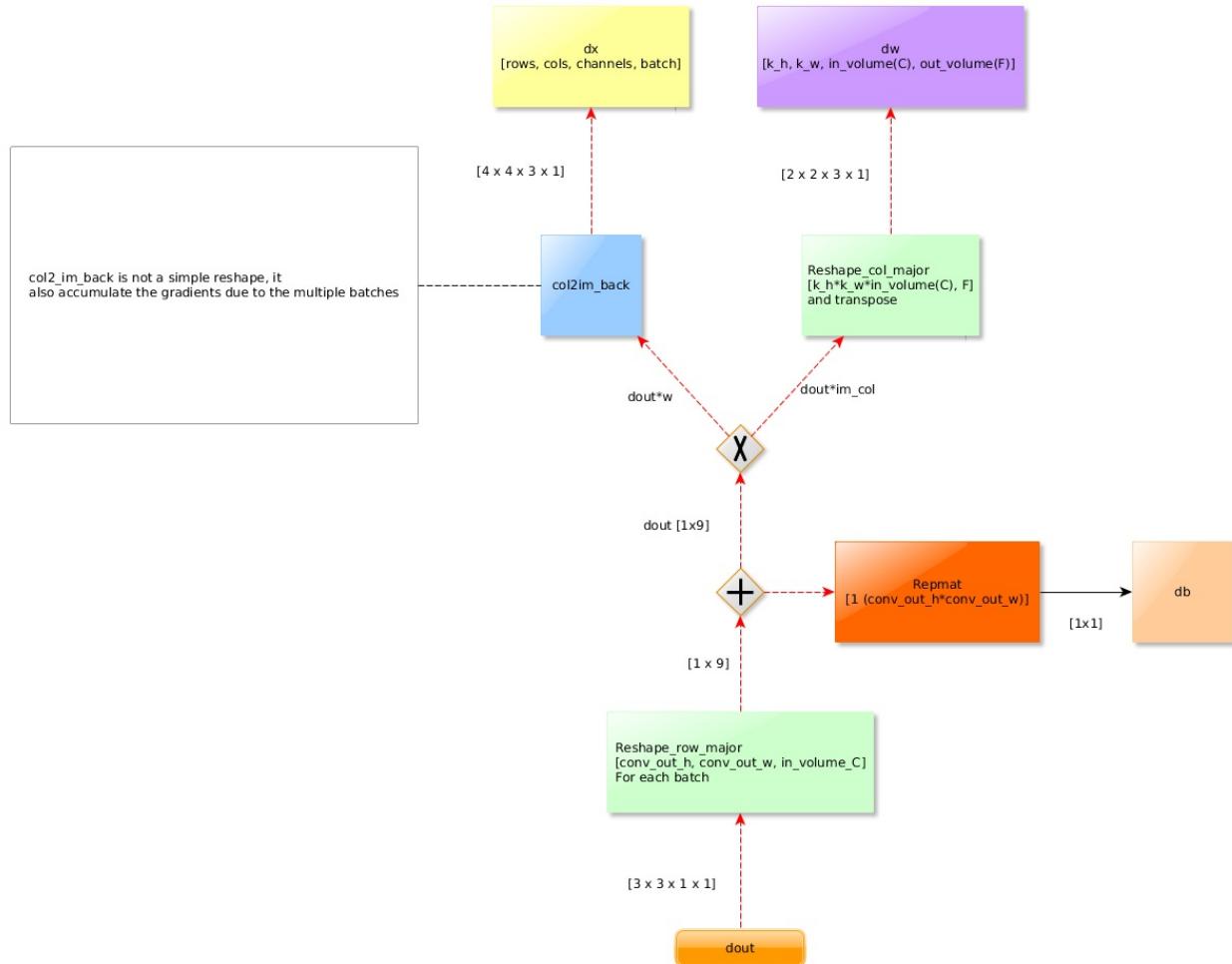


Backward graph

Using the im2col technique the computation graph resembles the FC layer with the same format $f(x, \theta, \beta) = (x \cdot \theta^T) + \beta$, the difference that now we have a bunch of reshapes, transposes and the im2col block.

About the reshapes and transposes during back propagation you just need to invert their operations using again another reshape or transpose, the only important thing to remember is that if you use a reshape row major during forward propagation you need to use a reshape row major on the backpropagation.

The only point to pay attention is the im2col backpropagation operation. The issue is that it cannot be implemented as a simple reshape. This is because the patches could actually overlap (depending on the stride), so you need to sum the gradients where the patches intersect.



Matlab forward propagation

```

function [activations] = ForwardPropagation(obj, input, weights, bias)
    % Tensor format (rows,cols,channels, batch) on matlab
    [H,W,~,N] = size(input);
    [HH,WW,C,F] = size(weights);

    % Calculate output sizes
    H_prime = (H+2*obj.m_padding-HH)/obj.m_stride +1;
    W_prime = (W+2*obj.m_padding-WW)/obj.m_stride +1;

    % Alocate memory for output
    activations = zeros([H_prime,W_prime,F,N]);

    % Preparing filter weights
    filter_col = reshape(weights,[HH*WW*C F]);
    filter_col_T = filter_col';

    % Preparing bias
    if ~isempty(bias)
        bias_m = repmat(bias,[1 H_prime*W_prime]);
    else
        b = zeros(size(filter_col,1),1);
        bias_m = repmat(b,[1 H_prime*W_prime]);
    end

```

```

end

% Here we convolve each image on the batch in a for-loop, but the
% could also handle a image batch at the input, so all computation
% be just one big matrix multiplication. We opted now for this to
% par-for implementation with OpenMP on CPU
for idxBatch = 1:N
    im = input(:,:,:,:idxBatch);
    im_col = im2col_ref(im,HH,WW,obj.m_stride,obj.m_padding,1);
    mul = (filter_col_T * im_col) + bias_m;
    activations(:,:,:,:idxBatch) = reshape_row_major(mul,[H_prime
end

% Cache results for backpropagation
obj.activations = activations;
obj.weights = weights;
obj.biases = bias;
obj.previousInput = input;
end

```

Matlab backward propagation

```

function [gradient] = BackwardPropagation(obj, dout)
dout = dout.input;
[H,W,~,N] = size(obj.previousInput);
[HH,WW,C,F] = size(obj.weights);

% Calculate output sizes
H_prime = (H+2*obj.m_padding-HH)/obj.m_stride +1;
W_prime = (W+2*obj.m_padding-WW)/obj.m_stride +1;

% Preparing filter weights
filter_col_T = reshape_row_major(obj.weights,[F HH*WW*C]);

% Initialize gradients
dw = zeros(size(obj.weights));
dx = zeros(size(obj.previousInput));
% Get the bias gradient which will be the sum of dout over the
% dimensions (batches(4), rows(1), cols(2))
db = sum(sum(dout, 1), 2), 4);

for idxBatch = 1:N
    im = obj.previousInput(:,:,:,:idxBatch);
    im_col = im2col_ref(im,HH,WW,obj.m_stride,obj.m_padding,1);
    dout_i = dout(:,:,:,:idxBatch);

    dout_i_reshaped = reshape_row_major(dout_i,[F, H*W]);

    dw_before_reshape = dout_i_reshaped * im_col';
    dw_i = reshape(dw_before_reshape',[HH, WW, C, F]);
    dw = dw + dw_i;

    % We now have the gradient just before the im2col

```

```

grad_before_im2col = (dout_i_reshaped' * filter_col_T);
% Now we need to backpropagate im2col (im2col_back),
% results will padded by one always
dx_padded = im2col_back_ref(grad_before_im2col,H_prime, W_prime,
% Now we need to take out the padding
dx(:,:,idxBatch) = dx_padded(2:end-1, 2:end-1,:);
end

%% Output gradients
gradient.bias = db;
gradient.input = dx;
gradient.weight = dw;

end

```

Matlab im2col

```

function [ img_matrix ] = im2col_ref( inputImg, k_height, k_width, S )
%IM2COL Convert image to a matrix, this step is used to accelerate
%convolutions, implementing the convolution as a matrix multiplication.
% This version currently does not support batch of images, we choose
% because we're first going to use the CPU mode, and we want to rely
% parfor (OpenMP)
coder.extrinsic('warning')
% Get Image dimensions
[imgHeight, imgWidth, imgChannels] = size(inputImg);

% Calculate convolved result size.
newImgHeight = ((imgHeight + 2*P - k_height) / S)+1;
newImgWidth = ((imgWidth + 2*P - k_width) / S)+1;
offset_K_Height = 0;
offset_K_Width = 0;

% Check if it is a real number
if rem(newImgHeight,1) ~= 0 || rem(newImgWidth,1) ~= 0
    warning('warning: Invalid stride or pad for input\n');
    if isConv
        % Convolution do a floor
        newImgHeight = floor(((imgHeight + 2*P - k_height) / S)+1);
        newImgWidth = floor(((imgWidth + 2*P - k_width) / S)+1);
    else
        % Pooling do a ceil and adapt the sampling window
        newImgHeight = ceil(((imgHeight + 2*P - k_height) / S)+1);
        newImgWidth = ceil(((imgWidth + 2*P - k_width) / S)+1);
        offset_K_Height = k_height - 1;
        offset_K_Width = k_width - 1;
    end
end

% Allocate output sizes
img_matrix = zeros(...,
    (imgChannels*k_height*k_width), (newImgHeight * newImgWidth) ...

```

```

);

% Only pad if needed
if P ~= 0
    inputImg_Padded = padarray(inputImg, [P P]);
    % Get dimensions again before iterate on padded image, otherwise
    % keep sampling with the old (unpadded size)
    [imgHeight, imgWidth, ~] = size(inputImg_Padded);
end

% Iterate on the input image like a convolution
cont = 1;
for r=1:S:(imgHeight-offset_K_Height)
    for c=1:S:(imgWidth-offset_K_Width)
        % Avoid slide out of the image (Security buffer overflow)
        if (((c+k_width)-1) <= imgWidth) && (((r+k_height)-1) <= imgHeight)
            % Select window on input volume
            if P == 0
                patch = inputImg(r:(r+k_height)-1,c:(c+k_width)-1,:);
            else
                patch = inputImg_Padded(r:(r+k_height)-1,c:(c+k_width)-1,:);
            end

            % Convert patch to a col vector, the matlab reshape order is
            % row major while other languages (C/C++, python) are column
            % major, on this particular case (im2col, then matrix multiplication
            % the kernel) this order will not matter, but it's not always true...
            patchRow = reshape(patch,[],1);

            % Append the transformed patch into the output matrix
            img_matrix(:,cont) = patchRow;
            cont = cont+1;
        end
    end
end
end

```

Matlab im2col backward propagation

```

function [ img_grad ] = im2col_back_ref( dout, dout_H, dout_W, S, HH,
    %IM2COL_BACK_REF Backpropagation of im2col
    % dout: (
    % Return
    % Image gradient (H,W,C)

    % Calculate the spatial dimensions of im_grad
    % Observe that the result will be "padded"
    H = (dout_H - 1) * S + HH;
    W = (dout_W - 1) * S + WW;

```

```

img_grad = zeros(H,W,C);

for ii=1:(dout_H*dout_W)
    row = dout(ii,:);

    % Create a patch from the row
    patch = reshape_row_major(row, [HH WW C]);
    %patch = reshape(row, [HH WW C]);

    % Calculate indexes on dx
    h_start = floor(((ii-1) / dout_W) * S);
    w_start = mod((ii-1),dout_W) * S;
    h_start = h_start + 1;
    w_start = w_start + 1;

    img_grad(h_start:h_start+HH-1, w_start:w_start+WW-1, :) = img_grad;
end
end

```

Python example for forward propagation

```

def conv_forward_naive(x, w, b, conv_param):
    """
    A naive implementation of the forward pass for a convolutional layer.

    The input consists of N data points, each with C channels, height H
    and width W. We convolve each input with F different filters, where each
    filter spans all C channels and has height HH and width HH.

    Input:
    - x: Input data of shape (N, C, H, W)
    - w: Filter weights of shape (F, C, HH, WW)
    - b: Biases, of shape (F,)
    - conv_param: A dictionary with the following keys:
        - 'stride': The number of pixels between adjacent receptive fields
          in horizontal and vertical directions.
        - 'pad': The number of pixels that will be used to zero-pad the
          input.
    Returns a tuple of:
    - out: Output data, of shape (N, F, H', W') where H' and W' are given by
      H' = 1 + (H + 2 * pad - HH) / stride
      W' = 1 + (W + 2 * pad - WW) / stride
    - cache: (x, w, b, conv_param)
    """
    out = None
    pad_num = conv_param['pad']
    stride = conv_param['stride']
    N,C,H,W = x.shape
    F,C,HH,WW = w.shape
    H_prime = (H+2*pad_num-HH) // stride + 1
    W_prime = (W+2*pad_num-WW) // stride + 1
    out = np.zeros([N,F,H_prime,W_prime])

```

```
#im2col
for im_num in range(N):
    im = x[im_num, :, :, :]
    im_pad = np.pad(im, ((0, 0), (pad_num, pad_num), (pad_num, pad_num)), (pad_num, pad_num))
    im_col = im2col(im_pad, HH, WW, stride)
    filter_col = np.reshape(w, (F, -1))
    mul = im_col.dot(filter_col.T) + b
    out[im_num, :, :, :] = col2im(mul, H_prime, W_prime, 1)
cache = (x, w, b, conv_param)
return out, cache
```

Python example for backward propagation

```
def conv_backward_naive(dout, cache):
    """
    A naive implementation of the backward pass for a convolutional layer.

    Inputs:
    - dout: Upstream derivatives.
    - cache: A tuple of (x, w, b, conv_param) as in conv_forward_naive

    Returns a tuple of:
    - dx: Gradient with respect to x
    - dw: Gradient with respect to w
    - db: Gradient with respect to b
    """
    dx, dw, db = None, None, None

    x, w, b, conv_param = cache
    pad_num = conv_param['pad']
    stride = conv_param['stride']
    N, C, H, W = x.shape
    F, C, HH, WW = w.shape
    H_prime = (H+2*pad_num-HH) // stride + 1
    W_prime = (W+2*pad_num-WW) // stride + 1

    dw = np.zeros(w.shape)
    dx = np.zeros(x.shape)
    db = np.zeros(b.shape)

    # We could calculate the bias by just summing over the right dimension
    # Bias gradient (Sum on dout dimensions (batch, rows, cols))
    #db = np.sum(dout, axis=(0, 2, 3))

    for i in range(N):
        im = x[i, :, :, :]
        im_pad = np.pad(im, ((0, 0), (pad_num, pad_num), (pad_num, pad_num)), (pad_num, pad_num))
        im_col = im2col(im_pad, HH, WW, stride)
        filter_col = np.reshape(w, (F, -1)).T

        dout_i = dout[i, :, :, :]
        dbias_sum = np.reshape(dout_i, (F, -1))
```

```

dbias_sum = dbias_sum.T

#bias_sum = mul + b
db += np.sum(dbias_sum, axis=0)
dmul = dbias_sum

#mul = im_col * filter_col
dfilter_col = (im_col.T).dot(dmul)
dim_col = dmul.dot(filter_col.T)

dx_padded = col2im_back(dim_col,H_prime,W_prime,stride,HH,WW,C)
dx[i,:,:,:] = dx_padded[:,pad_num:H+pad_num,pad_num:W+pad_num]
dw += np.reshape(dfilter_col.T,(F,C,HH,WW))

return dx, dw, db

```

Im2col and Col2im sources in python

This implementation will receive a image on the format of a 3 dimension tensor [channels, rows, cols] and will create a 2d matrix on the format [rows=(new_h*new_w), cols=(kw*kw*C)] notice that this algorithm will output the transposed version of the diagram above.

```

def im2col(x, hh, ww, stride):
    """
    Args:
        x: image matrix to be translated into columns, (C, H, W)
        hh: filter height
        ww: filter width
        stride: stride
    Returns:
        col: (new_h*new_w, hh*ww*C) matrix, each column is a cube that \
              new_h = (H-hh) // stride + 1, new_w = (W-ww) // stride +
    """
    c, h, w = x.shape
    new_h = (h-hh) // stride + 1
    new_w = (w-ww) // stride + 1
    col = np.zeros([new_h*new_w, c*hh*ww])

    for i in range(new_h):
        for j in range(new_w):
            patch = x[..., i*stride:i*stride+hh, j*stride:j*stride+ww]
            col[i*new_w+j, :] = np.reshape(patch, -1)

    return col

def col2im(mul, h_prime, w_prime, C):
    """
    Args:
        mul: (h_prime*w_prime*w, F) matrix, each col should be reshaped
        h_prime: reshaped filter height
        w_prime: reshaped filter width
        C: reshaped filter channel, if 0, reshape the filter to 2D, Oth
    Returns:
    """

```

```

    if C == 0: (F,h_prime,w_prime) matrix
    Otherwise: (F,C,h_prime,w_prime) matrix
"""
F = mul.shape[1]
if(C == 1):
    out = np.zeros([F,h_prime,w_prime])
    for i in range(F):
        col = mul[:,i]
        out[i,:,:] = np.reshape(col,(h_prime,w_prime))
else:
    out = np.zeros([F,C,h_prime,w_prime])
    for i in range(F):
        col = mul[:,i]
        out[i,:,:] = np.reshape(col,(C,h_prime,w_prime))

return out

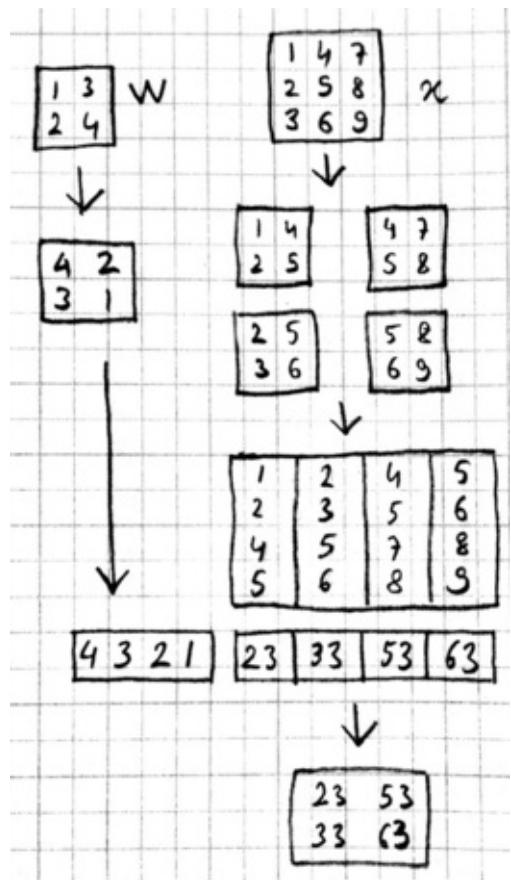
def col2im_back(dim_col,h_prime,w_prime,stride, hh,ww,c):
"""
Args:
    dim_col: gradients for im_col,(h_prime*w_prime, hh*ww*c)
    h_prime,w_prime: height and width for the feature map
    stride: stride
    hh,ww,c: size of the filters
Returns:
    dx: Gradients for x, (C,H,W)
"""
H = (h_prime - 1) * stride + hh
W = (w_prime - 1) * stride + ww
dx = np.zeros([c,H,W])
for i in range(h_prime*w_prime):
    row = dim_col[i,:]
    h_start = (i / w_prime) * stride
    w_start = (i % w_prime) * stride
    dx[:,h_start:h_start+hh,w_start:w_start+ww] += np.reshape(row,(hh,ww,c))
return dx

```

Smaller example

To make things simpler on our heads, follow the simple example of convolving X[3x3] with W[2x2]

Convolution Layer



Convolution Layer

```
>> W = [1 3; 2 4];
>> X = [1 4 7; 2 5 8; 3 6 9];
>> reference_result = conv2(X,W,'valid')

reference_result =
23     53
33     63

>> W = flipud(fliplr(W));
W_col = W(:)'

W_col =
4     3     2     1

>> X_col = im2col(X,size(W));
>> result_im2col_conv = reshape(W_col * X_col, size(reference_result))

result_im2col_conv =
23     53
33     63

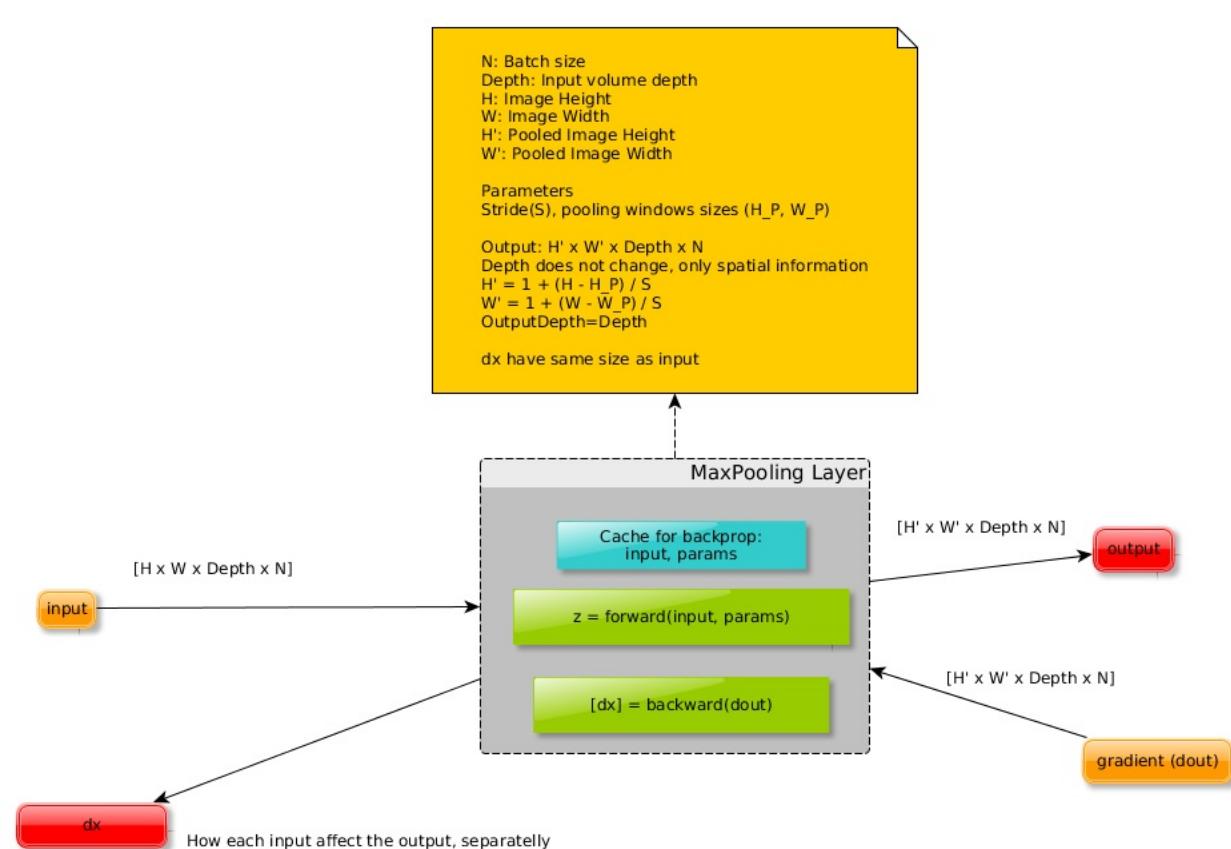
>> X_col

X_col =
1     2     4     5
2     3     5     6
4     5     7     8
5     6     8     9
```

Pooling Layer

Pooling Layer

Introduction

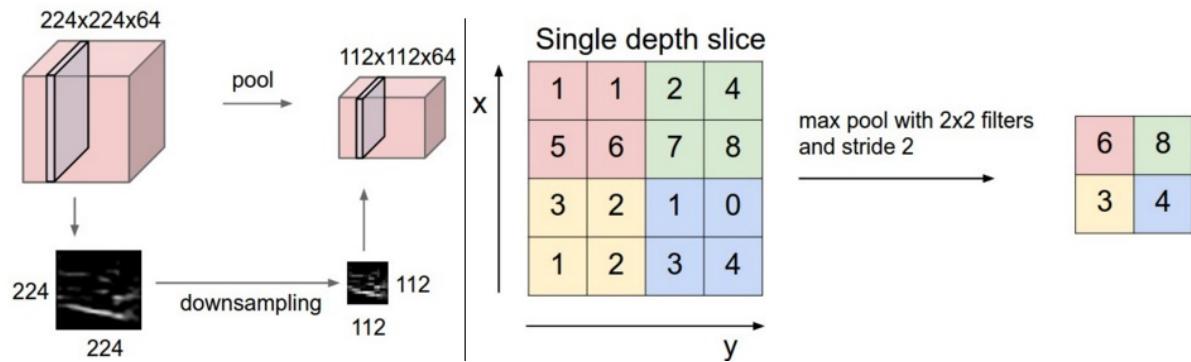


The pooling layer, is used to reduce the spatial dimensions, but not depth, on a convolution neural network, model, basically this is what you gain:

1. By having less spatial information you gain computation performance
2. Less spatial information also means less parameters, so less chance to over-fit
3. You get some translation invariance

Some projects don't use pooling, specially when they want to "learn" some object specific position. Learn how to play atari games.

On the diagram bellow we show the most common type of pooling the max-pooling layer, which slides a window, like a normal convolution, and get the biggest value on the window as the output.



The most important parameters to play:

- Input: $H_1 \times W_1 \times \text{Depth_In} \times N$
- Stride: Scalar that control the amount of pixels that the window slide.
- K: Kernel size

Regarding it's Output $H_2 \times W_2 \times \text{Depth_Out} \times N$:

$$W_2 = (W_1 - K)/S + 1$$

$$H_2 = (H_1 - K)/S + 1$$

$$\text{Depth}_{\text{out}} = \text{Depth}_{\text{In}}$$

It's also valid to point out that there is no learnable parameters on the pooling layer. So it's backpropagation is simpler.

Forward Propagation

The window movement mechanism on pooling layers is the same as convolution layer, the only change is that we will select the biggest value on the window.

Python Forward propagation

```

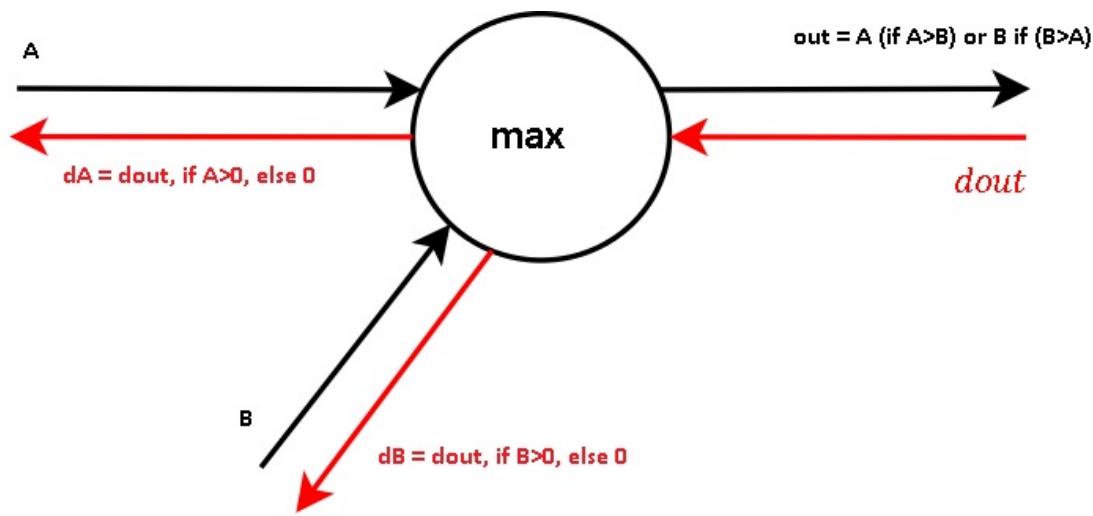
180     def max_pool_forward_naive(x, pool_param):
181         """
182             Compute the forward max pooling (naive way)
183             Inputs:
184                 - x: 4d Input tensor , of shape (N, C, H, W)
185                 - pool_param: dictionary with the following keys:
186                     - 'pool_height/width': Sliding window height/width
187                     - 'stride': Sliding moving distance
188             N: Mini-batch size
189             C: Input depth (ie 3 for RGB images)
190             H/W: Image height/width
191             HH/WW: Kernel Height/Width
192
193             Returns a tuple of: (out, cache)
194             """
195             # Get input tensor and parameter data
196             N, C, H, W = x.shape
197             S = pool_param["stride"]
198             # Consider H_P and W_P as the sliding window height and width
199             H_P = pool_param["pool_height"]
200             W_P = pool_param["pool_width"]
201
202             # Calculate output size
203             out = None
204             HH = 1 + (H - H_P) / S
205             WW = 1 + (W - W_P) / S
206             out = np.zeros((N,C,HH,WW))
207
208             # Calculate output (Both for loops do the same thing ....)
209             #for n in xrange(N): # For each element on batch
210                 #for depth in xrange(C): # For each input depth
211                     #for r in xrange(HH): # Slide vertically
212                         #for c in xrange(WW): # Slide horizontally
213                             # Get biggest element on the window
214                             #out[n,depth,r,c] = np.max(x[n,depth,r*S:r*S+H_P,c*S:c*S+W_P])
215
216             # Calculate output
217             for n in xrange(N): # For each element on batch
218                 for depth in xrange(C): # For each input depth
219                     for r in xrange(0,H,S): # Slide vertically taking stride into account
220                         for c in xrange(0,W,S): # Slide horizontally taking stride into account
221                             # Get biggest element on the window
222                             out[n,depth,r/S,c/S] = np.max(x[n,depth,r:r+H_P,c:c+W_P])
223
224             # Return output and save inputs and paramters to cache
225             cache = (x, pool_param)
226             return out, cache

```

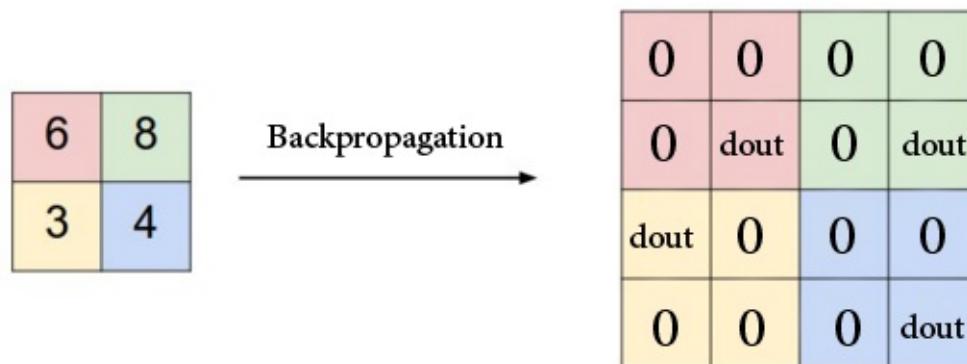
Matlab Forward propagation

Backward Propagation

From the [backpropagation chapter](#) we learn that the max node simply act as a router, giving the input gradient "dout" to the input that has value bigger than zero.



You can consider that the max pooling use a series of max nodes, on it's computation graph. So consider the backward propagation of the max pooling layer as a product between a mask containing all elements that were selected during the forward propagation and dout.



In other words the gradient with respect to the input of the max pooling layer will be a tensor make of zeros except on the places that was selected during the forward propagation.

Python Backward propagation

```

228     def max_pool_backward_naive(dout, cache):
229         """
230             Compute the backward propagation of max pooling (naive way)
231             Inputs:
232                 - dout: Upstream derivatives, same size as cached x
233                 - cache: A tuple of (x, pool_param) as in the forward pass.
234             Returns:
235                 - dx: Gradient with respect to x
236         """
237         # Get data back from cache
238         x, pool_param = cache
239
240         # Get input tensor and parameter
241         N, C, H, W = x.shape
242         S = pool_param["stride"]
243         H_P = pool_param["pool_height"]
244         W_P = pool_param["pool_width"]
245         N, C, HH, WW = dout.shape
246
247         # Initialize dx
248         dx = None
249         dx = np.zeros(x.shape)
250
251         # Calculate dx (mask * dout)
252         for n in xrange(N): # For each element on batch
253             for depth in xrange(C): # For each input depth
254                 for r in xrange(HH): # Slide vertically (use stride on the fly)
255                     for c in xrange(WW): # Slide horizontally (use stride on the fly)
256                         # Get window and calculate the mask
257                         x_pool = x[n, depth, r*S:r*S+H_P, c*S:c*S+W_P]
258                         mask = (x_pool == np.max(x_pool))
259                         # Calculate mask*dout
260                         dx[n, depth, r*S:r*S+H_P, c*S:c*S+W_P] = mask*dout[n, depth, r, c]
261
262         # Return dx
263         return dx

```

Improving performance

On future chapter we will learn a technique that improves the convolution performance, until them we will stick with the naive implementation.

Next Chapter

Next chapter we will learn about Batch Norm layer

Pooling Layer

Batch Norm layer

Batch Norm layer

Introduction

On this chapter we will learn about the batch norm layer. Previously we said that [feature scaling](#) make the job of the gradient descent easier. Now we will extend this idea and normalize the activation of every Fully Connected layer or Convolution layer during training. This also means that while we're training we will select an batch calculate it's mean and standard deviation.

You can think that the batch-norm will be some kind of adaptive (or learnable) pre-processing block with trainable parameters. Which also means that we need to back-propagate them.

The original batch-norm paper can be found [here](#).

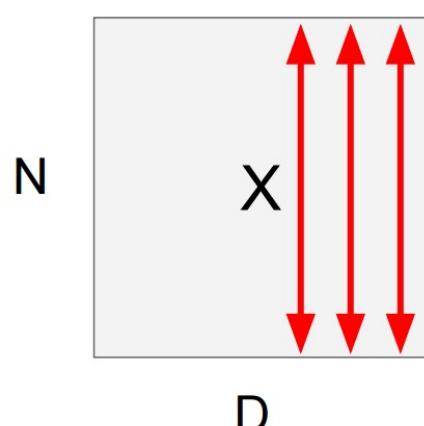
Here is the list of advantages of using Batch-Norm:

1. Improves gradient flow, used on very deep models (Resnet need this)
2. Allow higher learning rates
3. Reduce dependency on initialization
4. Gives some kind of regularization (Even make Dropout less important but keep using it)
5. As a rule of thumb if you use Dropout+BatchNorm you don't need L2 regularization

It basically force your activations (Conv,FC ouputs) to be unit standard deviation and zero mean.

To each learning batch of data we apply the following normalization.

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$



1. compute the empirical mean and variance independently for each dimension.

2. Normalize

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

The output of the batch norm layer, has the γ, β are parameters. Those parameters will be learned to best represent your activations. Those parameters allows a learnable (scale and shift) factor

$$y^k = \gamma^k \cdot \hat{x}^{(k)} + \beta^k$$

Now summarizing the operations:

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots m\}$;

Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$$

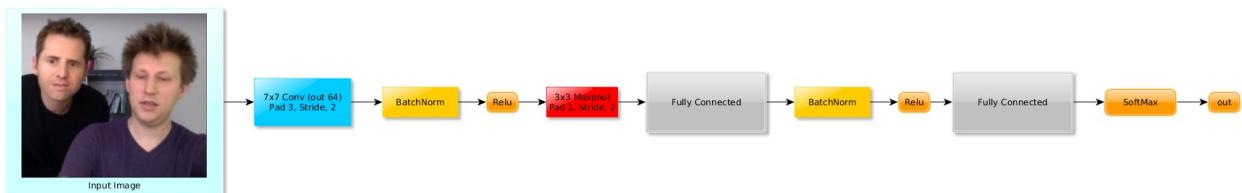
$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$

Here, ϵ is a small number, 1e-5.

Where to use the Batch-Norm layer

The batch norm layer is used after linear layers (ie: FC, conv), and before the non-linear layers (relu). There is actually 2 batch norm implementations one for FC layer and the other for conv layers.



Test time

At prediction time that batch norm works differently. The mean/std are not computed based on the batch. Instead, we need to build a estimate during training of the mean/std of the whole dataset(population) for each batch norm layer on your model.

One approach to estimating the population mean and variance during training is to use an [exponential](#)

moving average.

$$S_t = \alpha \cdot S_{t-1} + (1 - \alpha) \cdot Y_t$$

Where: S_t, S_{t-1} : Current and previous estimation (α) : Represents the degree of weighting decrease, a constant smoothing factor between 0 and 1 Y_t : Current value (could be mean or std) that we're trying to estimate

Normally when we implement this layer we have some kind of flag that detects if we're on training or testing.

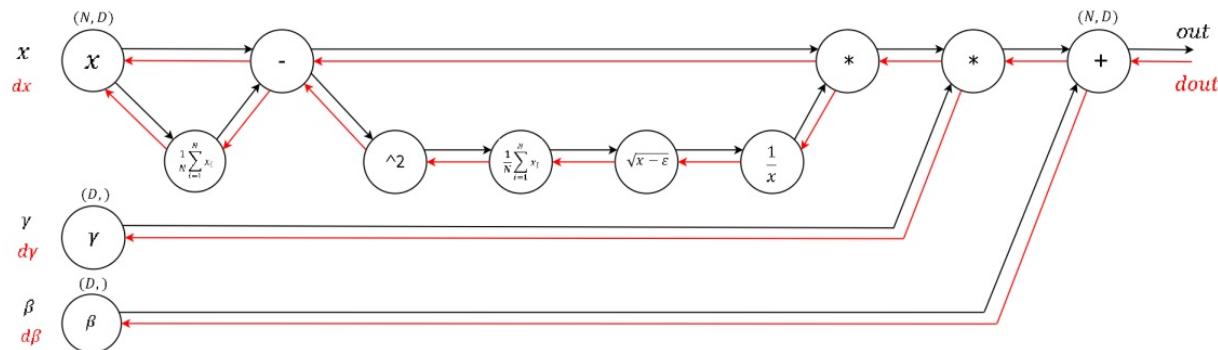
As reference we can find some tutorials with [Tensorflow](#) or [manually on python](#).

Backpropagation

As mentioned earlier we need to know how to backpropagate on the batch-norm layer, first as we did with other layers we need to create the computation graph. After this step we need to calculate the derivative of each node with respect to it's inputs.

Computation Graph

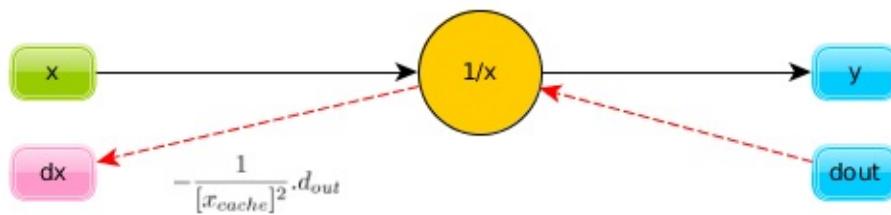
In order to find the partial derivatives on back-propagation is better to visualize the algorithm as a computation graph:



New nodes

By inspecting this graph we have some new nodes ($\frac{1}{N} \cdot \sum_{i=1}^N X(i)$, x^2 , $\sqrt{x - \epsilon}$, $\frac{1}{x}$). To simplify things you can use [Wolfram alpha](#) to find the derivatives. For backpropagate other nodes refer to the [Back-propagation chapter](#)

Block 1/x

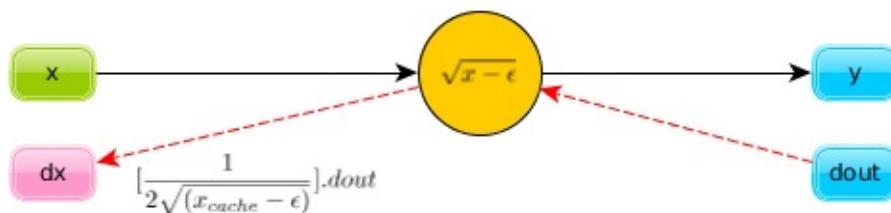


In other words:

$$\frac{\partial(\frac{1}{x})}{\partial x} = -\frac{1}{x^2} \therefore dx = -\frac{1}{x_{cache}^2} \cdot dout$$

Where: x_{cache} means the cached (or saved) input from the forward propagation. $dout$ means the previous block gradient

Block sqrt(x-epsilon)

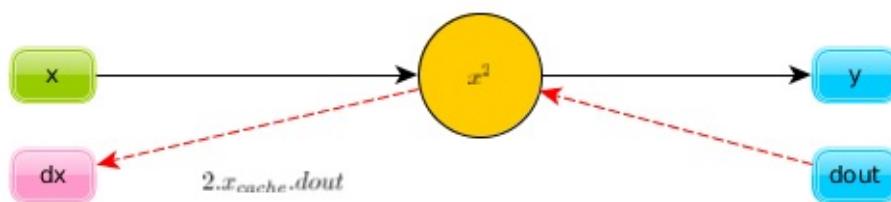


In other words:

$$\frac{\partial(\sqrt{x-\epsilon})}{\partial x} = \frac{1}{2\sqrt{(x_{cache}-\epsilon)}} \therefore dx = [\frac{1}{2\sqrt{(x_{cache}-\epsilon)}}] \cdot dout$$

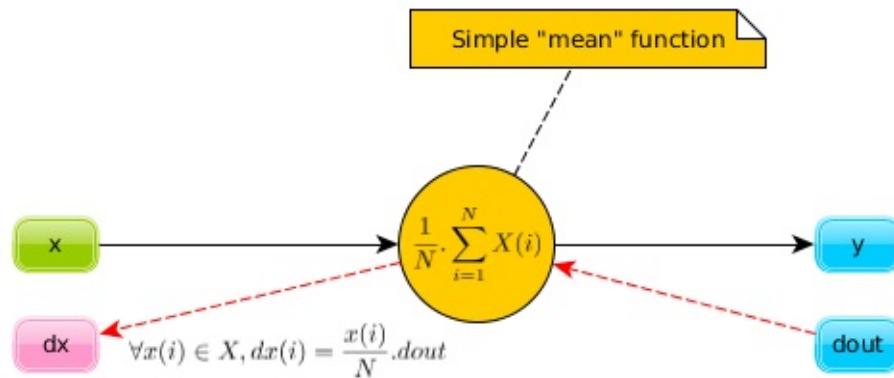
Where: x_{cache} : the cached (or saved) input from the forward propagation. $dout$: the previous block gradient ϵ : Some small number 0.00005

Block x^2



In other words:

$$\frac{\partial x^2}{\partial x} = 2 \cdot x \therefore dx = 2 \cdot x_{cache} \cdot dout$$

Block Summation

Like the SUM block this block will copy the input gradient $dout$ equally to all its inputs. So for all elements in X we will divide by N and multiply by $dout$.

Implementation**Python Forward Propagation**

```

4     def batchnorm_forward(x, gamma, beta, bn_param):
5         """
6             Forward pass for batch normalization (Use on FC layers).
7             Input:
8                 - x: Data of shape (N, D)
9                 - gamma,beta: Scale/Shift parameter of shape (D,)
10                - bn_param: Dictionary with the following keys: (mode,eps,momentum,r_mean/var)
11            Returns a tuple of: (out, cache)
12            """
13        mode = bn_param['mode']
14        eps = bn_param.get('eps', 1e-5)
15        momentum = bn_param.get('momentum', 0.9)
16
17        N, D = x.shape
18        running_mean = bn_param.get('running_mean', np.zeros(D, dtype=x.dtype))
19        running_var = bn_param.get('running_var', np.zeros(D, dtype=x.dtype))
20
21        out, cache = None, None
22        if mode == 'train':
23            # Forward pass
24            # Step 1: Calculate mean
25            mu = 1 / float(N) * np.sum(x, axis=0)
26            # Step 2: Subtract the mean of every training sample
27            xmu = x - mu
28            # Step 3 - Calculate denominator
29            carre = xmu**2
30            # Step 4 - Calculate variance
31            var = 1 / float(N) * np.sum(carre, axis=0)
32            # Step 5 - Add eps for numerical stability then get square root
33            sqrtvar = np.sqrt(var + eps)
34            # Step 6 - Invert square root
35            invvar = 1. / sqrtvar
36            # Step 7 - Calculate normalization
37            va2 = xmu * invvar
38            # Step 8 - Calculate
39            va3 = gamma * va2
40            # Step 9 - Shape out (N,D)
41            out = va3 + beta
42
43            # Calculate running mean and variance to be used on prediction
44            running_mean = momentum * running_mean + (1.0 - momentum) * mu
45            running_var = momentum * running_var + (1.0 - momentum) * var
46            # Store values
47            cache = (mu, xmu, carre, var, sqrtvar, invvar,
48                  va2, va3, gamma, beta, x, bn_param)
49        elif mode == 'test':
50            # On prediction get the running mean/variance
51            running_mean = bn_param['running_mean']
52            running_var = bn_param['running_var']
53            xbar = (x - running_mean)/np.sqrt(running_var+eps)
54            out = gamma*xbar + beta
55            cache = (x, xbar, gamma, beta, eps)
56        else:
57            raise ValueError('Invalid forward batchnorm mode "%s"' % mode)
58        # Save updated running mean/variance
59        bn_param['running_mean'] = running_mean
60        bn_param['running_var'] = running_var
61        # Return outputs
62        return out, cache

```

Python Backward Propagation

```

3     def batchnorm_backward(dout, cache):
4         """
5             Backward pass for batch normalization (Use on FC layers).
6             Use computation graph to guide the backward propagation!
7             Inputs:
8                 - dout: Upstream derivatives, of shape (N, D)
9                 - cache: Variable of intermediates from batchnorm_forward.
10            Returns a tuple of: (dx(N,D), dgamma(D), dbeta(D))
11        """
12        dx, dgamma, dbeta = None, None, None
13
14        # http://cthorey.github.io./backpropagation/
15        mu, xmu, carre, var, sqrtvar, invvar, va2, va3, gamma, beta, x, bn_param = cache
16        eps = bn_param.get('eps', 1e-5)
17        N, D = dout.shape
18
19        # Backprop Step 9
20        dva3 = dout
21        dbeta = np.sum(dout, axis=0)
22        # Backprop step 8
23        dva2 = gamma * dva3
24        dgamma = np.sum(va2 * dva3, axis=0)
25        # Backprop step 7
26        dxmu = invvar * dva2
27        dinvvar = np.sum(xmu * dva2, axis=0)
28        # Backprop step 6
29        dsqrtvar = -1. / (sqrtvar**2) * dinvvar
30        # Backprop step 5
31        dvar = 0.5 * (var + eps)**(-0.5) * dsqrtvar
32        # Backprop step 4
33        dcarre = 1 / float(N) * np.ones((carre.shape)) * dvar
34        # Backprop step 3
35        dxmu += 2 * xmu * dcarre
36        # Backprop step 2
37        dx = dxmu
38        dmu = - np.sum(dxmu, axis=0)
39        # Basckprop step 1
40        dx += 1 / float(N) * np.ones((dxmu.shape)) * dmu
41
42        return dx, dgamma, dbeta
43

```

Next Chapter

Next chapter we will learn about how to optimize our model weights.

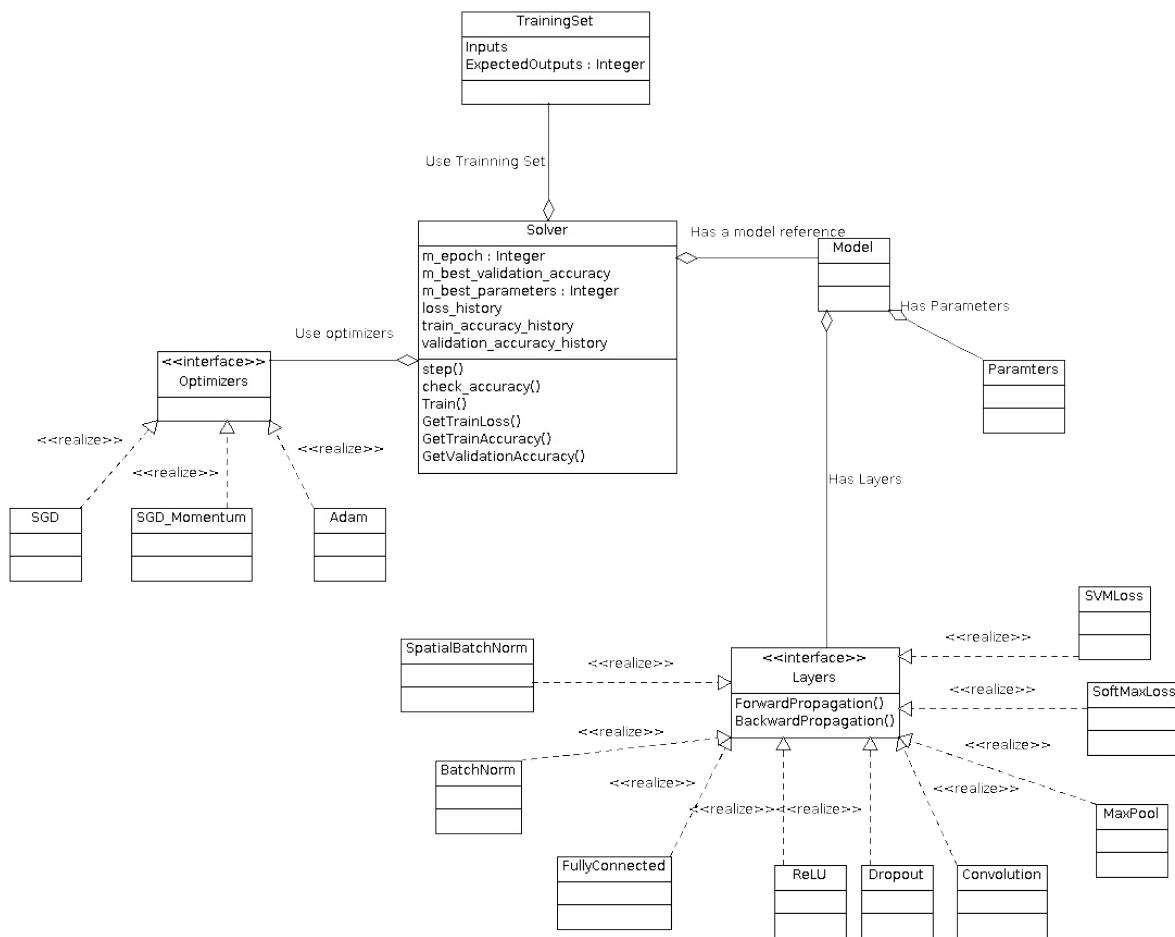
Batch Norm layer

Model Solver

Model Solver

Introduction

The mission of the model solver is to find the best set of parameters, that minimize the train/accuracy errors. On this chapter we will give a UML description with some piece of python/matlab code that allows you implement it yourself.



From the UML description we can infer some information about the `Solver` class:

1. It uses the training set, and has a reference to your model
2. Uses different type of optimizers(ex: SGD, ADAM, SGD with momentum)
3. Keep tracks of all the loss, accuracy during the training phase
4. Keep the set of parameters, that achieved best validation performance

Usage example

```

3   data = {
4       'X_train': # training data
5       'y_train': # training labels
6       'X_val': # validation data
7       'X_train': # validation labels
8   }
9   model = MyAwesomeModel(hidden_size=100, reg=10)
10  solver = Solver(model, data,
11      update_rule='sgd',
12      optim_config={
13          'learning_rate': 1e-3,
14      },
15      lr_decay=0.95,
16      num_epochs=10, batch_size=100,
17      print_every=100)
18  solver.train()

```

Train operation

This is the method called when you actually want to start a model training, the methods Step, Check_Accuracy are called inside the Train method:

1. Calculate number of iterations per epoch, based on number of epochs, train size, and batch size
2. Call step, for each iteration
3. Decay the learning rate
4. Calculate the validation accuracy
5. Cache the best parameters based on validation accuracy

Step operation

Basically during the step operation the following operations are done:

1. Extract a batch from the training set.
2. Get the model loss and gradients
3. Perform a parameter update with one of the optimizers.

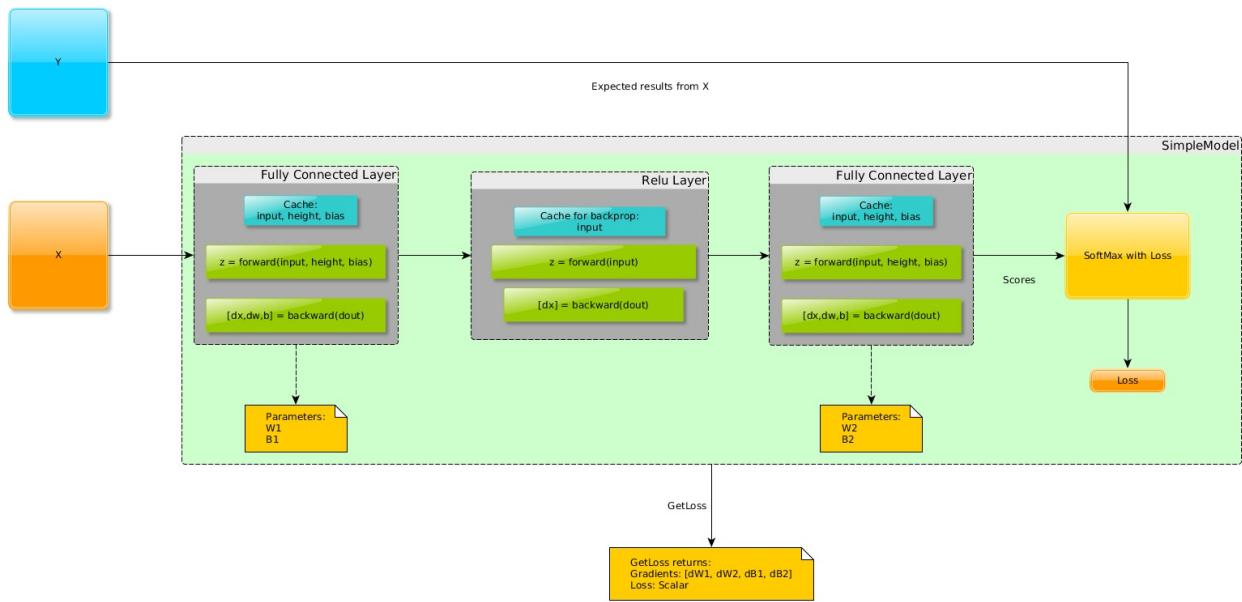
Check Accuracy

This method basically is called at the end of each epoch. Basically it uses the current set of parameters, and predict the whole validation set. The objective is at the end get the accuracy.
 $accuracy = mean(y_{predicted} == y_{validation})$

Model loss operation

We mentioned during the "Step" operation that we get the model loss and gradients. This operation is

implemented by the "getLoss" method. Consider the following basic model.



Bellow we have the "getLoss" function for the previous simple model.

```

30     def getLoss(self, X, y=None):
31         """
32             Compute the loss and gradient if y is give, or prediction scores
33         """
34         scores = None
35
36         # Do the forward propagation
37         # architecure affine - relu - affine - softmax
38         z1, cache1 = affine_forward(X, self.params['W1'], self.params['b1'])
39         a2, cache1r = relu_forward(z1)
40         z2, cache2 = affine_forward(a2, self.params['W2'], self.params['b2'])
41         scores = z2
42
43         # If y is None then we're doing prediction so we just return the scores
44         if y is None:
45             return scores
46         loss, grads = 0, {}
47
48         # If y is given we're in the middle of the training, so we need to calculate
49         # the loss and all the parameter gradients with respect to the loss
50
51         # Backward propagation of the model
52         loss, dout = softmax_loss(z2, y)
53         delta2, dW2, db2 = affine_backward(dout, cache2)
54         relu_delta2 = relu_backward(delta2, cache1r)
55         delta1, dW1, db1 = affine_backward(relu_delta2, cache1)
56
57         W1 = self.params['W1']
58         W2 = self.params['W2']
59
60         ## Apply regularization
61         dW1 += self.reg * W1
62         dW2 += self.reg * W2
63
64         # Save all gradients to a dictionary
65         grads["W1"] = dW1
66         grads["W2"] = dW2
67         grads["b1"] = db1
68         grads["b2"] = db2
69
70         # Calculate regularized loss
71         loss += 0.5 * self.reg * (np.sum(W1*W1) + np.sum(W2*W2))
72
73         # Return loss and gradients dictionary
74         return loss, grads

```

$$\frac{\partial L}{\partial X_{scores}}$$

Also bellow we have the "softmax_loss" function including "dout",

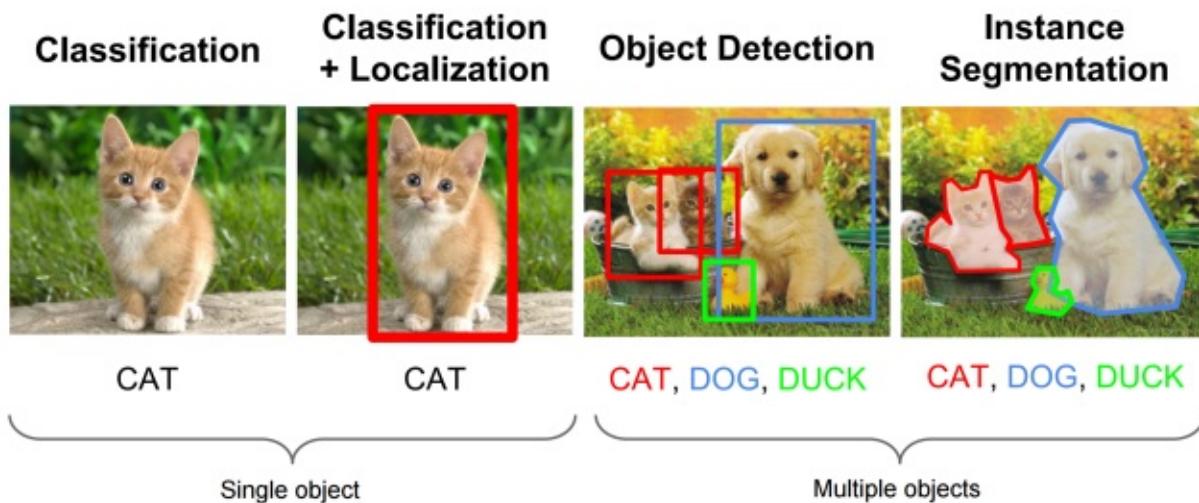
```
3  def softmax_loss(x, y):
4      """
5          Computes the loss and gradient for softmax classification.
6          Inputs:
7          - x: Input Scores (predicted scores from the model) shape (N,C)
8          - y: Correct scores (Training set correct labels) shape (N,)
9          N: Batch size
10         C: number of classes
11         Returns a tuple of: (loss, dout)
12         """
13         # Just fix numerical instability
14         probs = np.exp(x - np.max(x, axis=1, keepdims=True))
15         # Get probabilities and normalize
16         probs /= np.sum(probs, axis=1, keepdims=True)
17
18         # N will be the batch size
19         N = x.shape[0]
20
21         # Calculate loss
22         loss = -np.sum(np.log(probs[np.arange(N), y])) / N
23
24         # Calculate dout gradient (How loss change with respect to x)
25         dout_x = probs.copy()
26         dout_x[np.arange(N), y] -= 1
27         # Scale gradient with relation to N
28         dout_x /= N
29
30         # Return loss and dout (Loss gradient with respect to x)
31         return loss, dout_x
```

Object Localization and Detection

Object Localization and Detection

Introduction

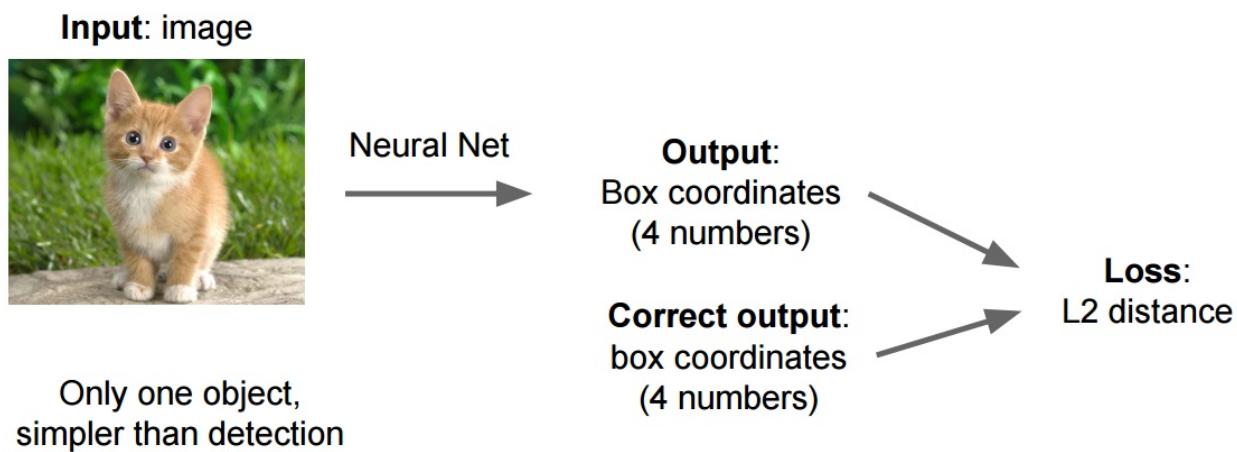
On this chapter we're going to learn about using convolution neural networks to localize and detect objects on images



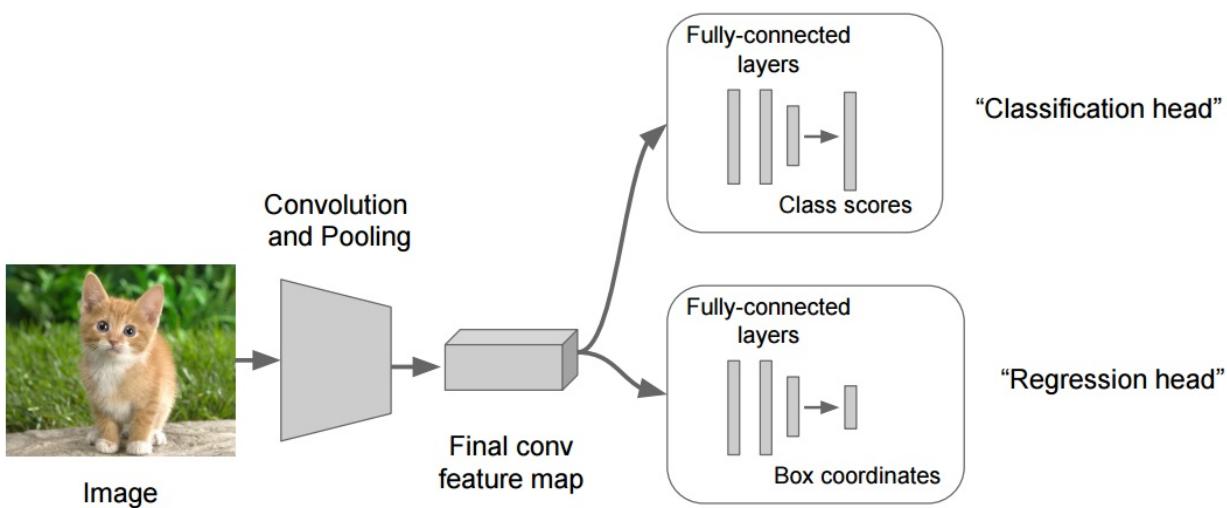
- RCNN
- Fast RCNN
- Faster RCNN
- Yolo
- SSD

Localize objects with regression

Regression is about returning a number instead of a class, in our case we're going to return 4 numbers ($x_0, y_0, \text{width}, \text{height}$) that are related to a bounding box. You train this system with an image an a ground truth bounding box, and use L2 distance to calculate the loss between the predicted bounding box and the ground truth.



Normally what you do is attach another fully connected layer on the last convolution layer

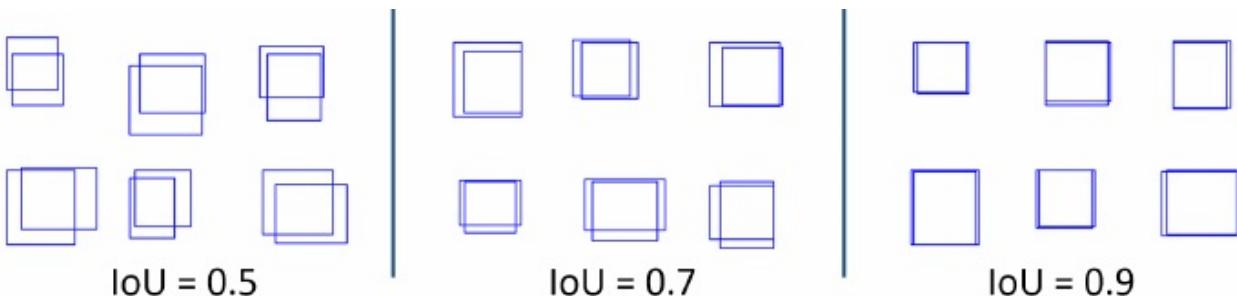


This will work only for one object at a time.

Some people attach the regression part after the last convolution (Overfeat) layer, while others attach after the fully connected layer (RCNN). Both works.

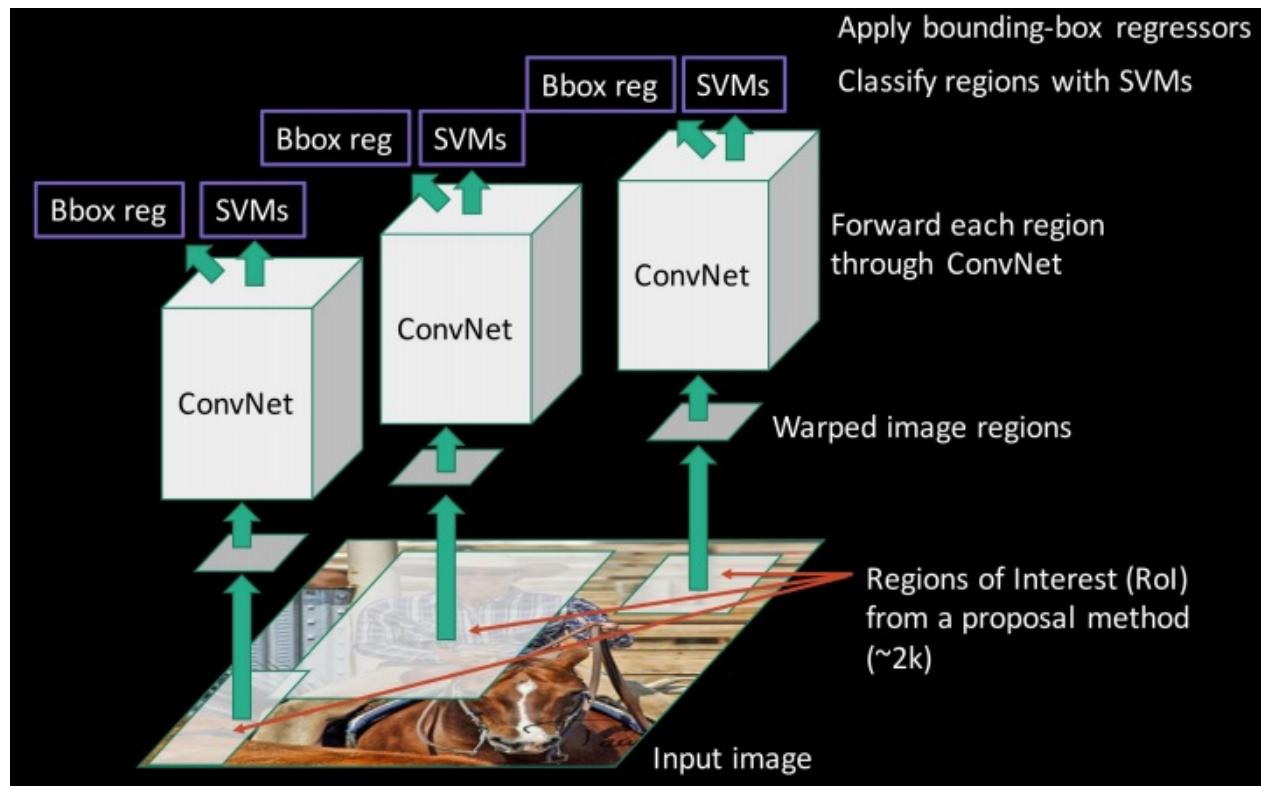
Comparing bounding box prediction accuracy

Basically we need to compare if the Intersect Over Union (IoU) between the prediction and the ground truth is bigger than some threshold (ex > 0.5)



RCNN

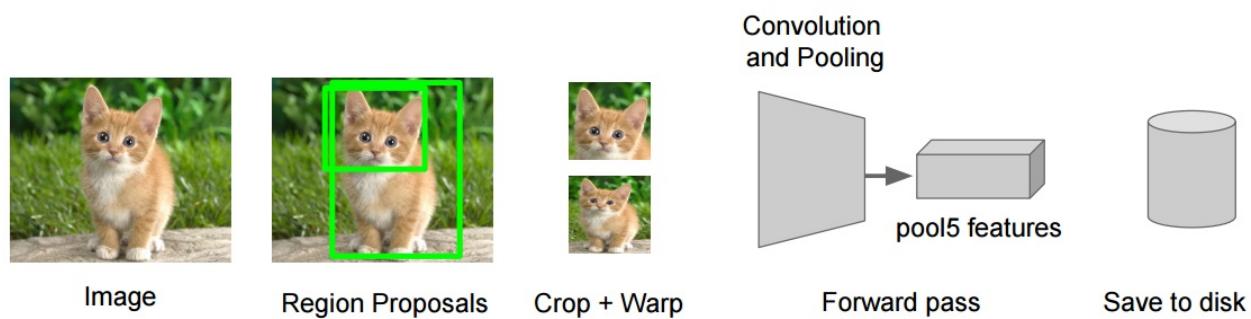
RCNN (Regions + CNN) is a method that relies on an external region proposal system.



The problem of RCNN is that it's never made to be fast, for instance the steps to train the network are these:

1. Take a pre-trained imagenet cnn (ex Alexnet)
2. Re-train the last fully connected layer with the objects that need to be detected + "no-object" class
3. Get all proposals(=~2000 p/image), resize them to match the cnn input, then save to disk.
4. Train SVM to classify between object and background (One binary SVM for each class)
5. BB Regression: Train a linear regression classifier that will output some correction factor

Step 3 Save and pre-process proposals

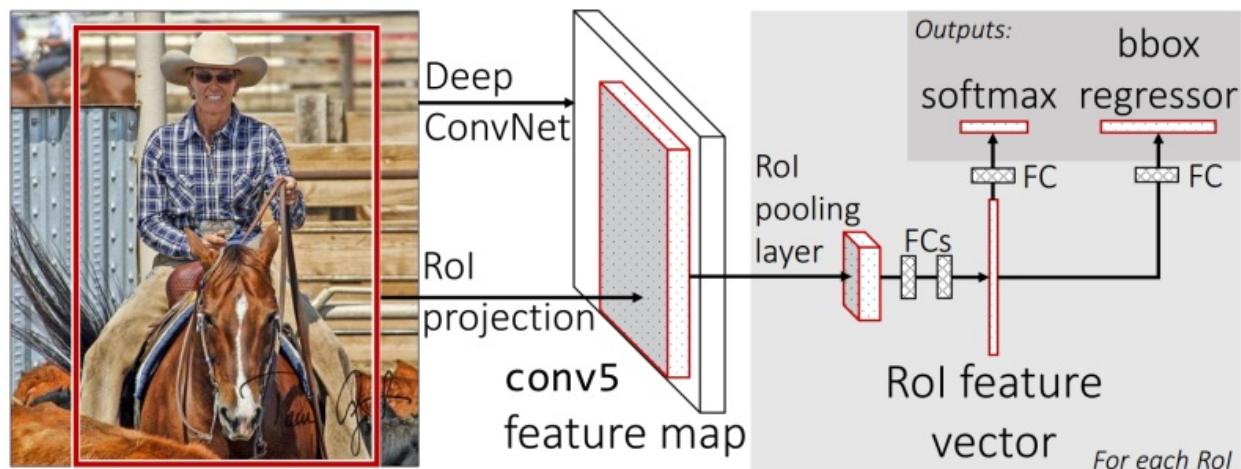


Step 5 (Adjust bounding box)

Training image regions			
Cached region features			
Regression targets (dx, dy, dw, dh) Normalized coordinates	(0, 0, 0, 0) Proposal is good	(.25, 0, 0, 0) Proposal too far to left	(0, 0, -0.125, 0) Proposal too wide

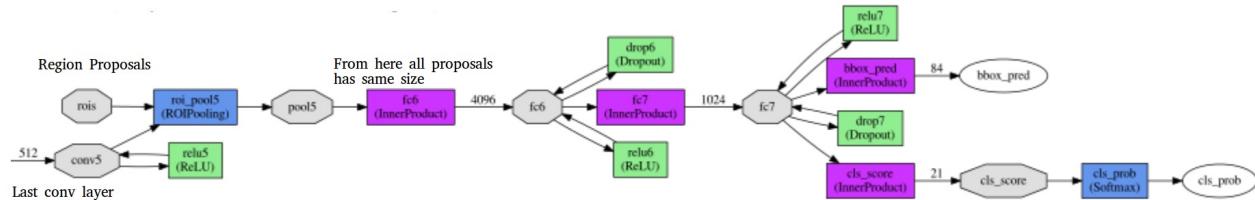
Fast RCNN

The Fast RCNN method receive region proposals from some external system (Selective search). This proposals will sent to a layer (Roi Pooling) that will resize all regions with their data to a fixed size. This step is needed because the fully connected layer expect that all the vectors will have same size



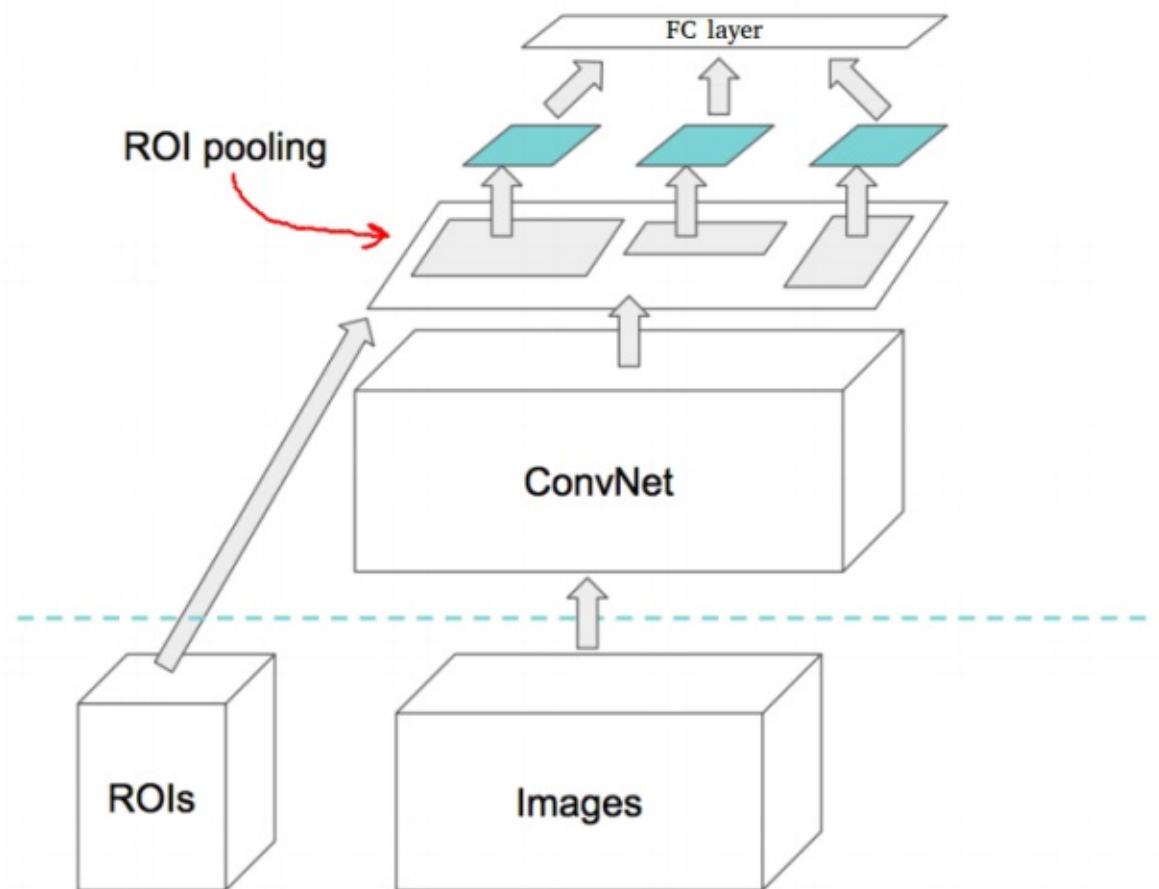
Proposals example, boxes=[r, x1, y1, x2, y2]





Still depends on some external system to give the region proposals (Selective search)

Roi Pooling layer

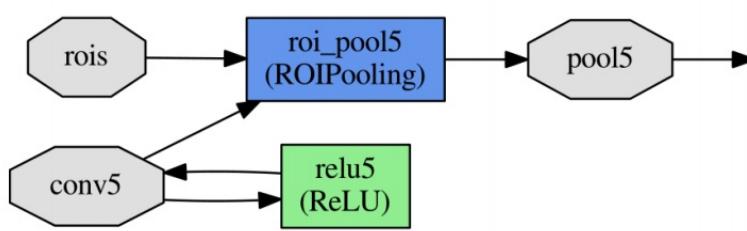


It's a type of max-pooling with a pool size dependent on the input, so that the output always has the same size. This is done because fully connected layer always expected the same input size.

```

188 | layer {
189 |   name: "roi_pool5"
190 |   type: "ROIPooling"
191 |   bottom: "conv5"
192 |   bottom: "rois"
193 |   top: "pool5"
194 |   roi_pooling_param {
195 |     pooled_w: 6
196 |     pooled_h: 6
197 |     spatial_scale: 0.0625 # 1/16
198 |   }
199 |

```

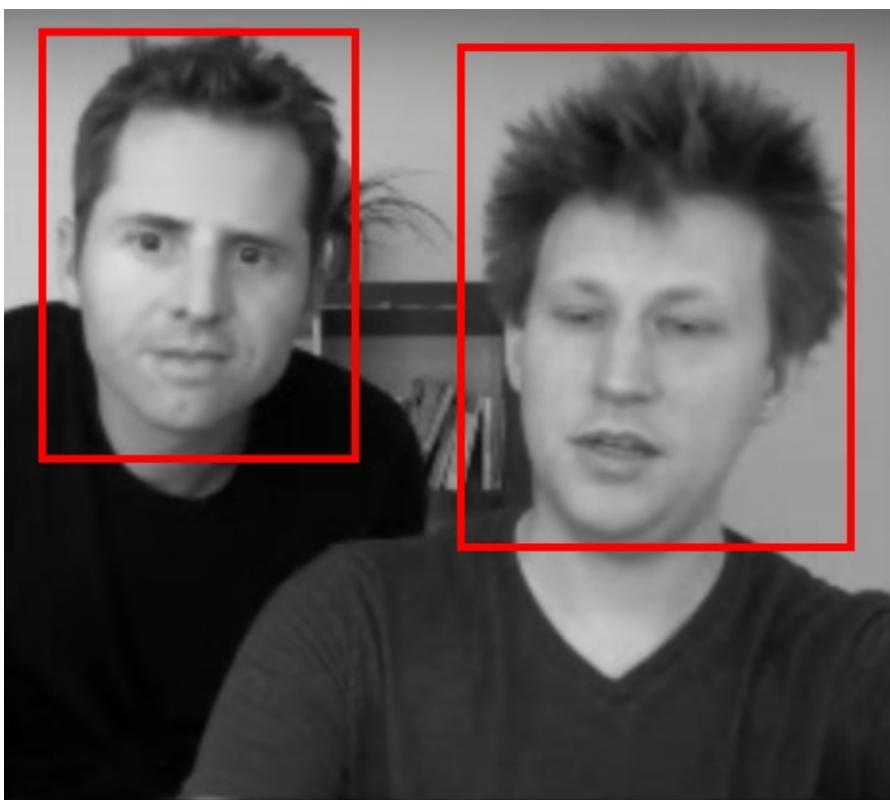


The inputs of the Roi layer will be the proposals and the last convolution layer activations. For example consider the following input image, and it's proposals.

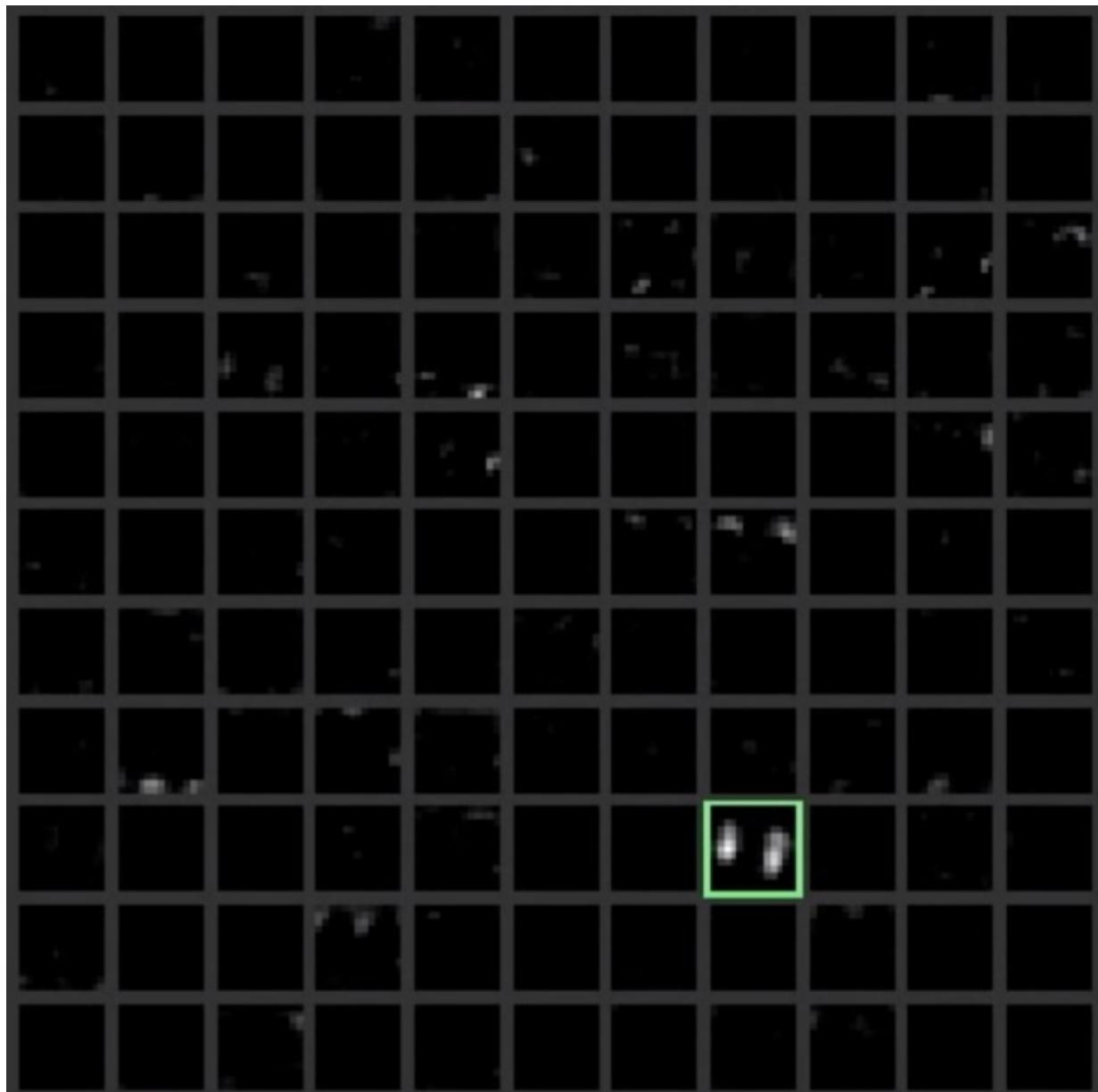
Input image



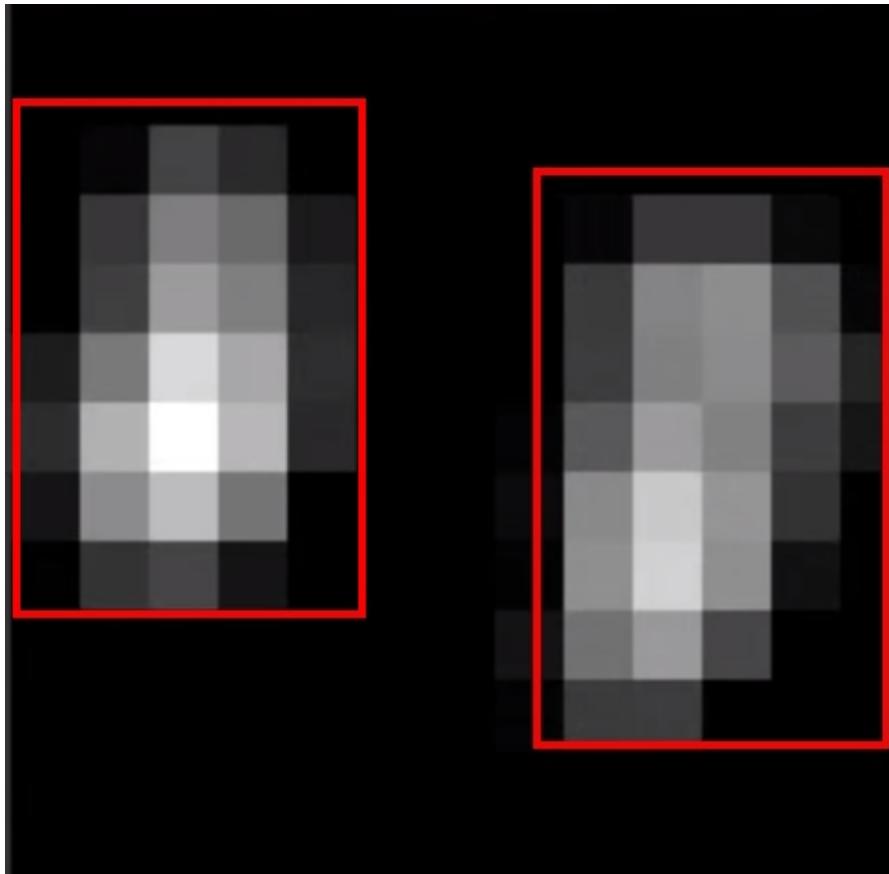
Two proposed regions



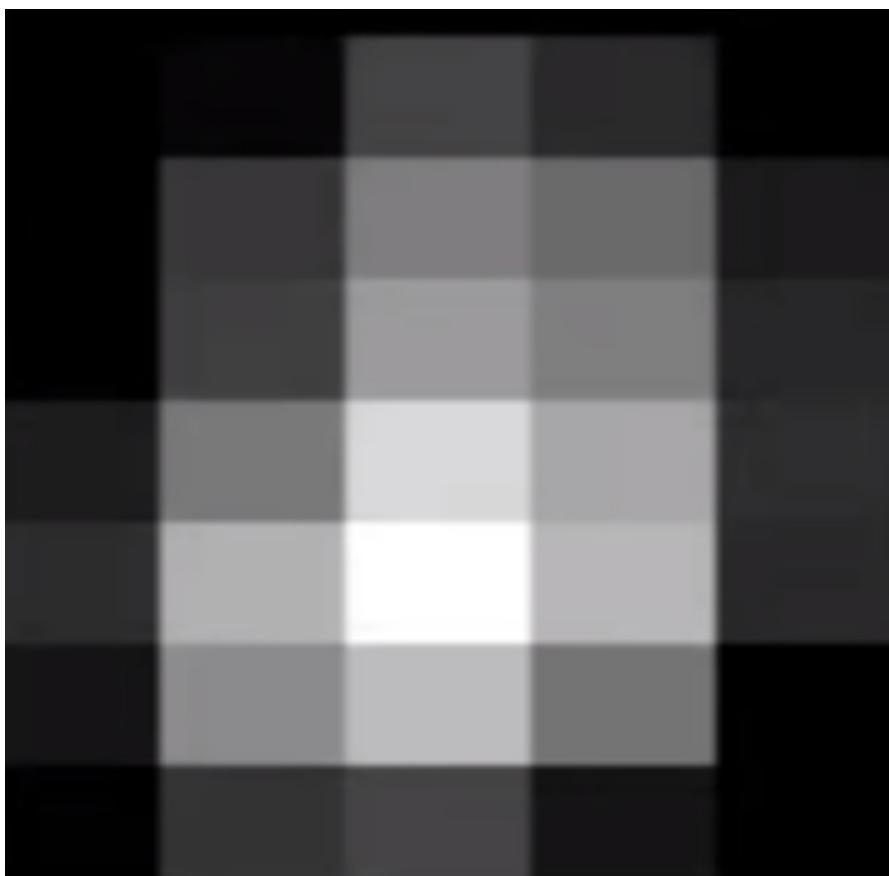
Now the activations on the last convolution layer (ex: conv5)

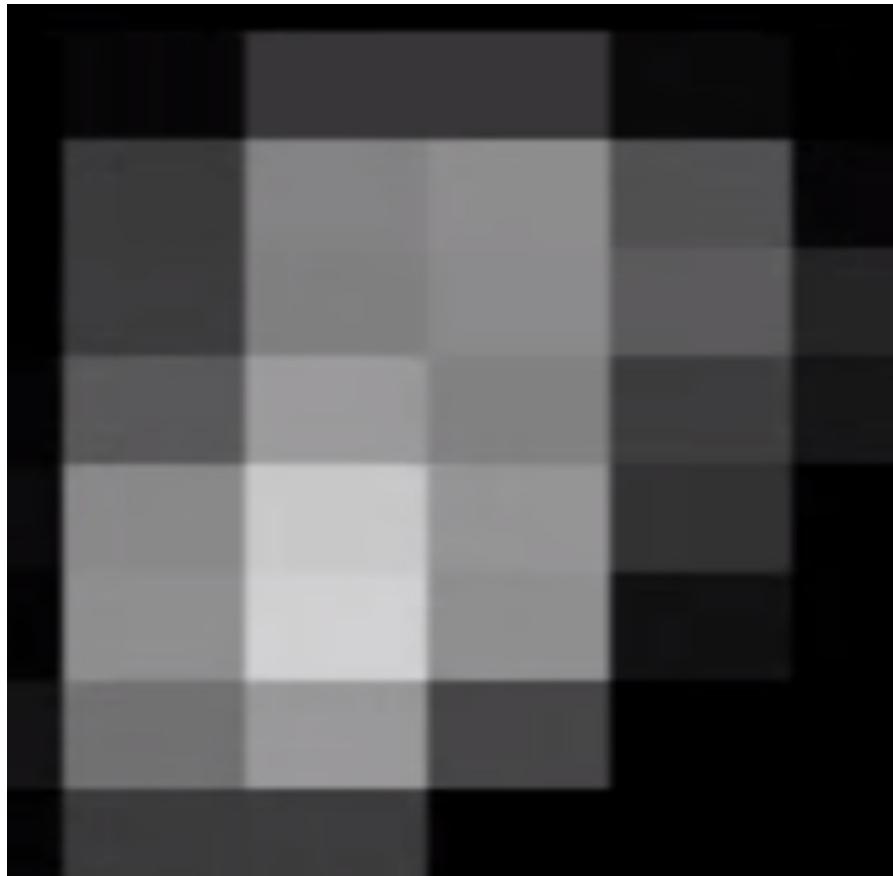


For each convolution activation (each cell from the image above) the Roi Pooling layer will resize, the region proposals (in red) to the same resolution expected on the fully connected layer. For example consider the selected cell in green.

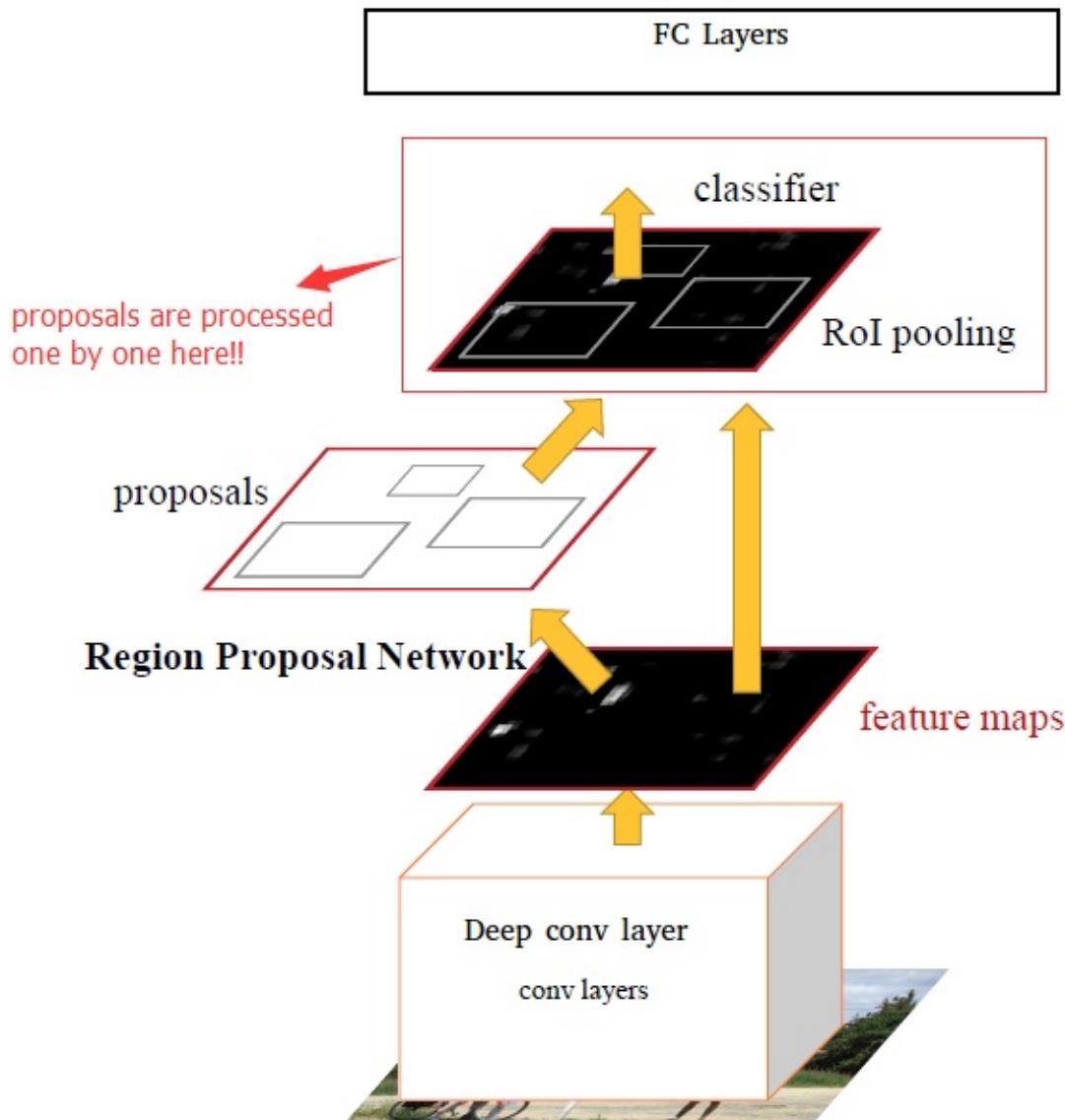


Here the output will be:





Faster RCNN



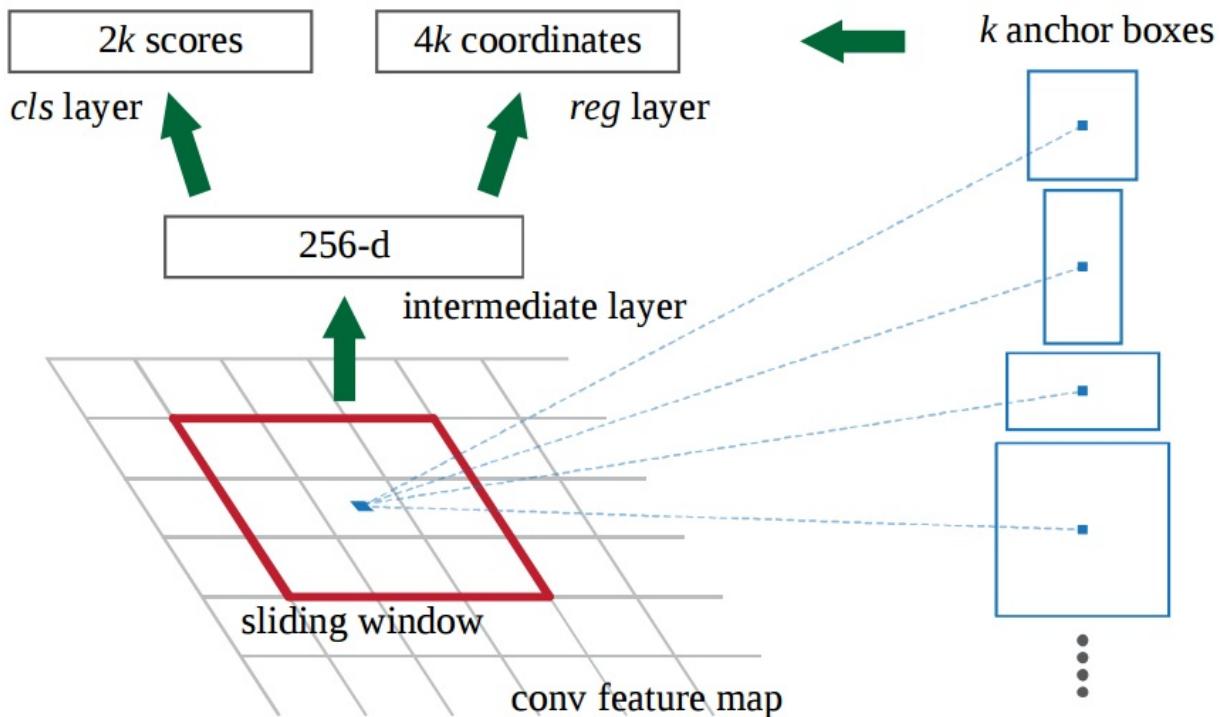
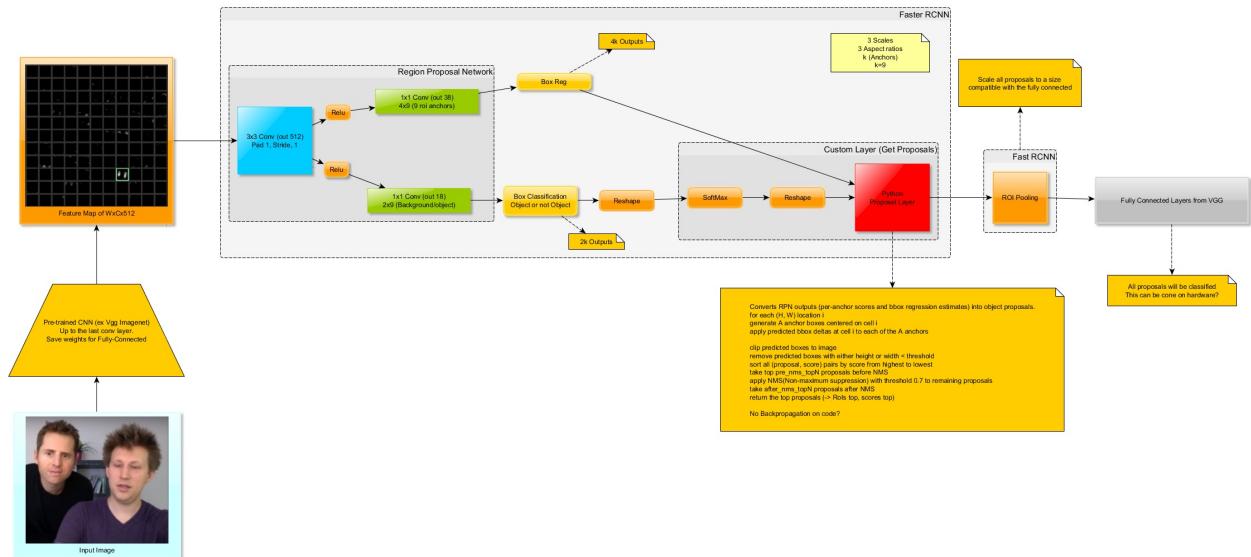
The main idea is use the last (or deep) conv layers to infer region proposals.
Faster-RCNN consists of two modules.

- RPN (Region proposals): Gives a set of rectangles based on deep convolution layer
- Fast-RCNN Roi Pooling layer: Classify each proposal, and refining proposal location

Region proposal Network

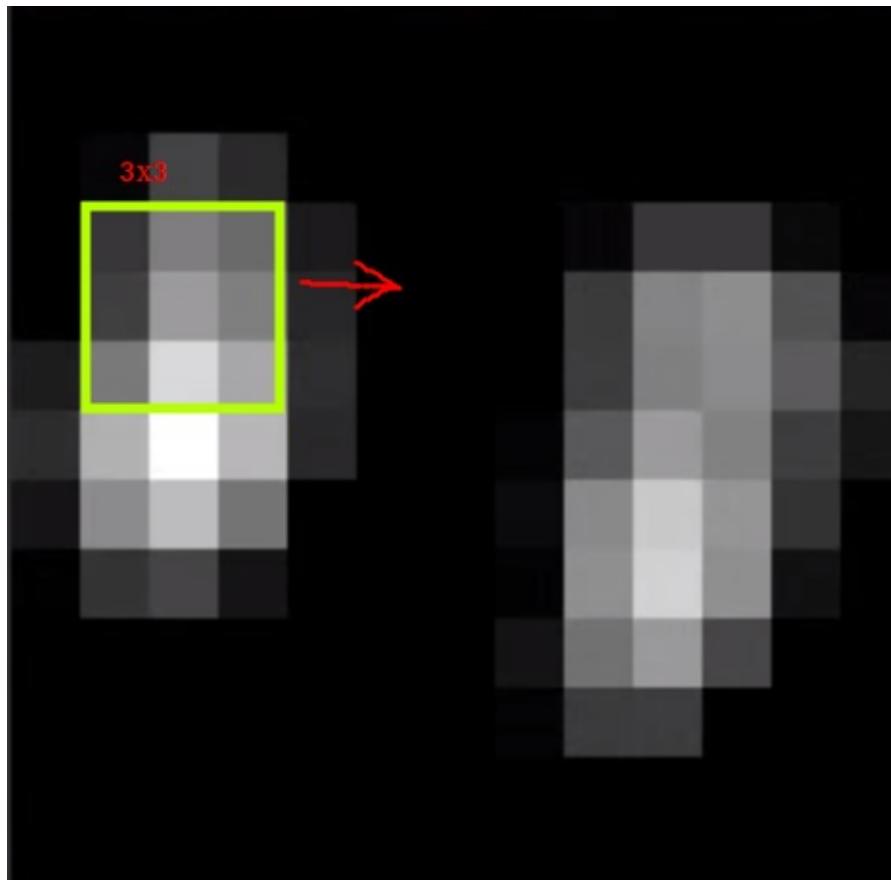
Here we break on a block diagram how Faster RCNN works.

1. Get a trained (ie imagenet) convolution neural network
2. Get feature maps from the last (or deep) convolution layer
3. Train a region proposal network that will decide if there is an object or not on the image, and also propose a box location
4. Give results to a custom (python) layer
5. Give proposals to a ROI pooling layer (like Fast RCNN)
6. After all proposals get reshaped to a fix size, send to a fully connected layer to continue the classification



How it works

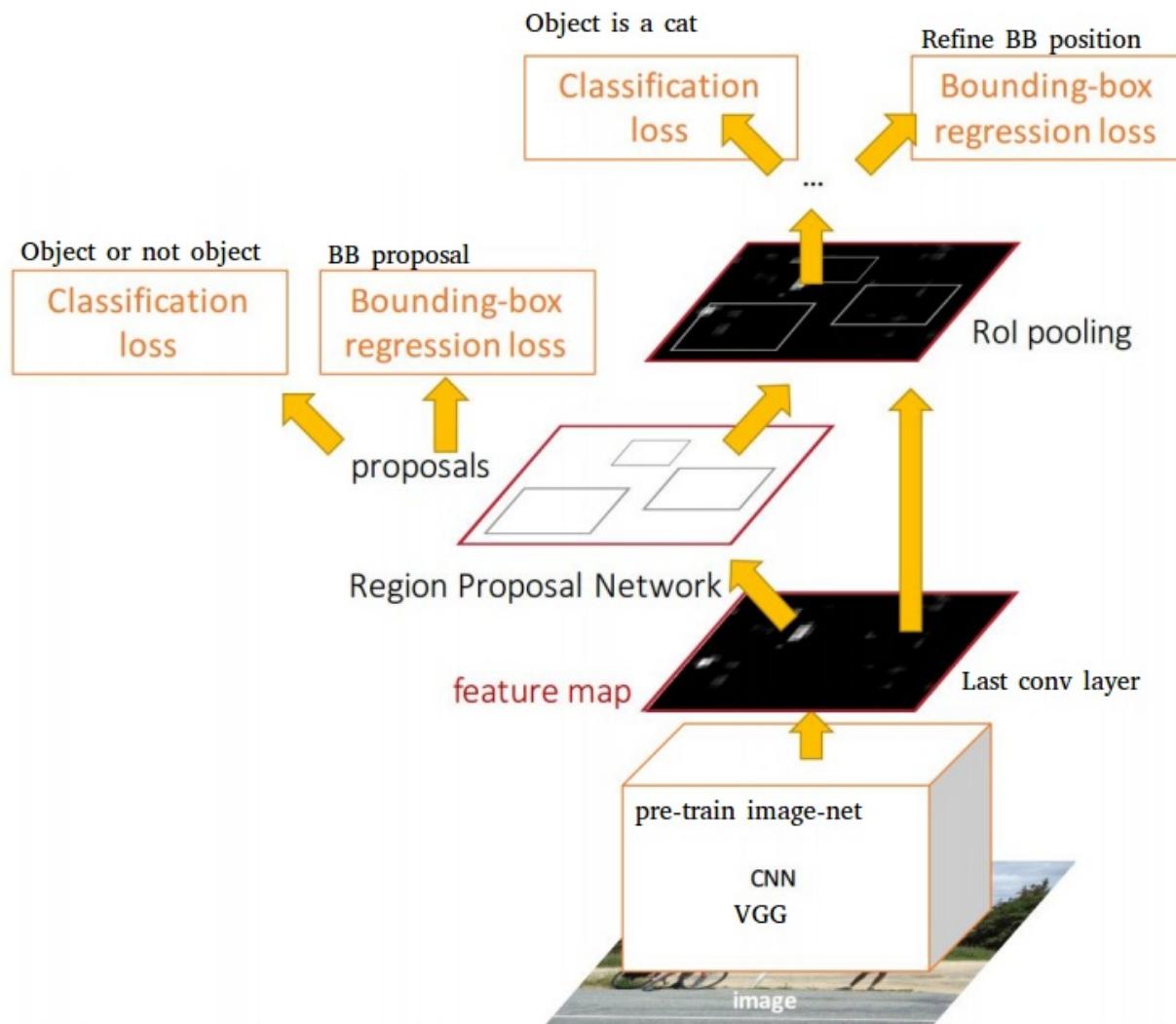
Basically the RPN slides a small window (3×3) on the feature map, that classify what is under the window as object or not object, and also gives some bounding box location.
 For every sliding window center it creates fixed k anchor boxes, and classify those boxes as been object or not.



Faster RCNN training

On the paper, each network was trained separately, but we also can train it jointly. Just consider the model having 4 losses.

- RPN Classification (Object or not object)
- RPN Bounding box proposal
- Fast RCNN Classification (Normal object classification)
- Fast RCNN Bounding-box regression (Improve previous BB proposal)



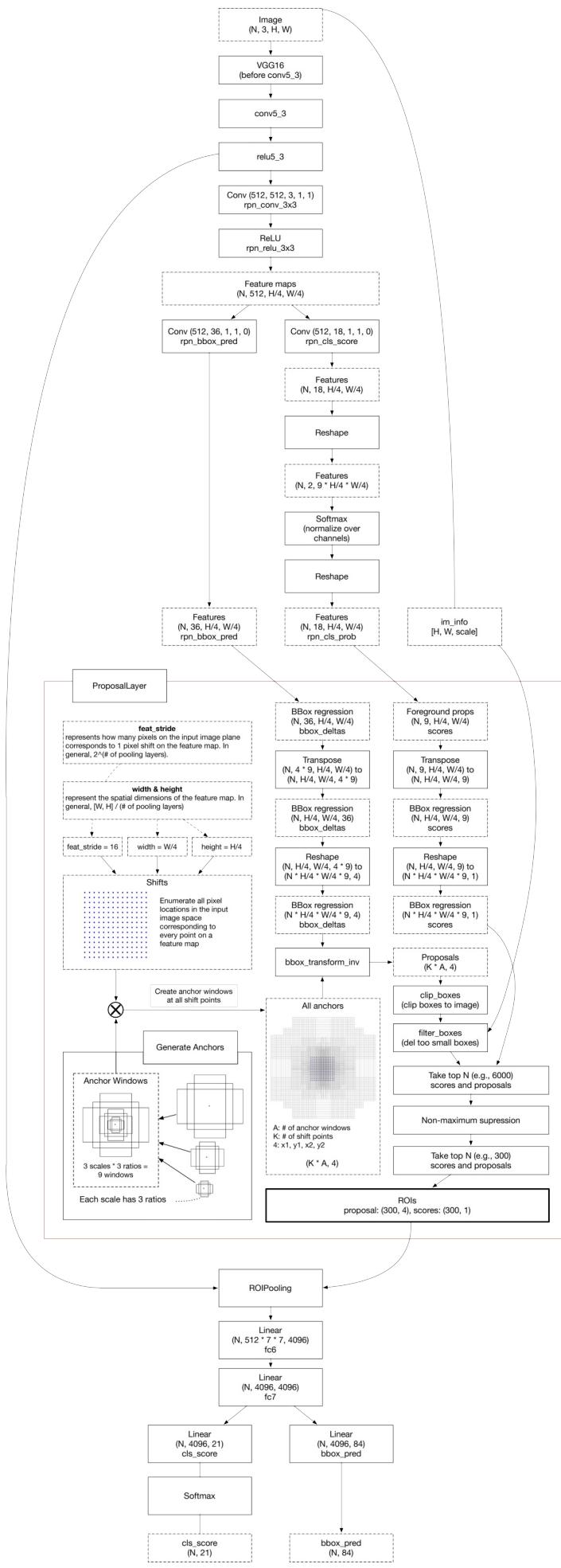
Faster RCNN results

The best result now is Faster RCNN with a resnet 101 layer.

	R-CNN	Fast R-CNN	Faster R-CNN
Test time per image (with proposals)	50 seconds	2 seconds	0.2 seconds
(Speedup)	1x	25x	250x
mAP (VOC 2007)	66.0	66.9	66.9

Complete Faster RCNN diagram

This diagram represents the complete structure of the Faster RCNN using VGG16, I've found on a github project [here](#). It uses a framework called [Chainer](#) which is a complete framework using only python (Sometimes cython).



Next Chapter

On the next chapter we will discuss a different type of object detector called single shot detectors.

Single Shot Detectors

Single Shot detectors

Introduction

The previous methods of object detection all share one thing in common: they have one part of their network dedicated to providing region proposals followed by a high quality classifier to classify these proposals. These methods are very accurate but come at a big computational cost (low frame-rate), in other words they are not fit to be used on embedded devices.

Another way of doing object detection is by combining these two tasks into one network. We can do this by instead of having a network produce proposals we instead have a set of pre defined boxes in which to look for objects.

Using convolutional features maps from later layers of a network we run small conv filters over these features maps to predict class scores and bounding box offsets.

Here is the family of object detectors that follow this strategy:

- SSD: Uses different activation maps (multiple-scales) for prediction of classes and bounding boxes
- YOLO: Uses a single activation map for prediction of classes and bounding boxes
- R-FCN(Region based Fully-Convolution Neural Networks): Like Faster Rcnn, but faster due to less computation per box.
- Multibox: asdasdas

Using these multiple scales helps to achieve a higher mAP(mean average precision) by being able to detect objects with different sizes on the image.

Summarising the strategy of these methods

1. Train a CNN with regression(box) and classification objective (loss function).
2. Use sliding window (conv) and non-maxima suppression during prediction on the conv feature maps (output of conv-relu)

On this kind of detectors it is typical to have a collection of boxes overlaid on the image at different spatial locations, scales and aspect ratios that act as “anchors” (sometimes called “priors” or “default boxes”).

A model is then trained to make two predictions for each anchor:

1. A discrete class prediction for each anchor
2. A continuous prediction of an offset by which the anchor needs to be shifted to fit the ground-truth bounding box.

Also the loss used on this methods are a combination of the 2 objectives, localization(regression) and classification.

Image Segmentation

Image Segmentation

Introduction

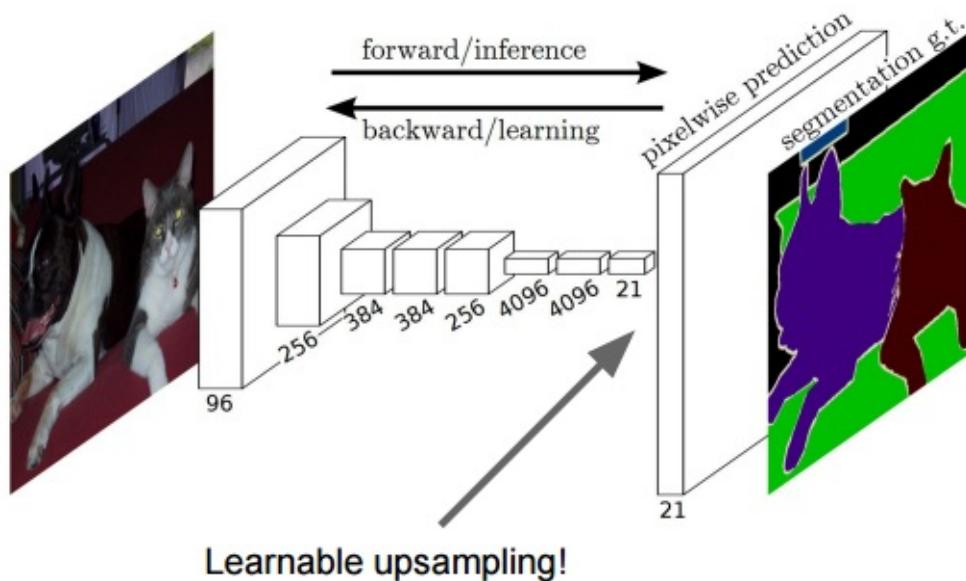
Now we're going to learn how to classify each pixel on the image, the idea is to create a map of all detected object areas on the image. Basically what we want is the image below where every pixel has a label.

On this chapter we're going to learn how convolutional neural networks (CNN) can do the job.

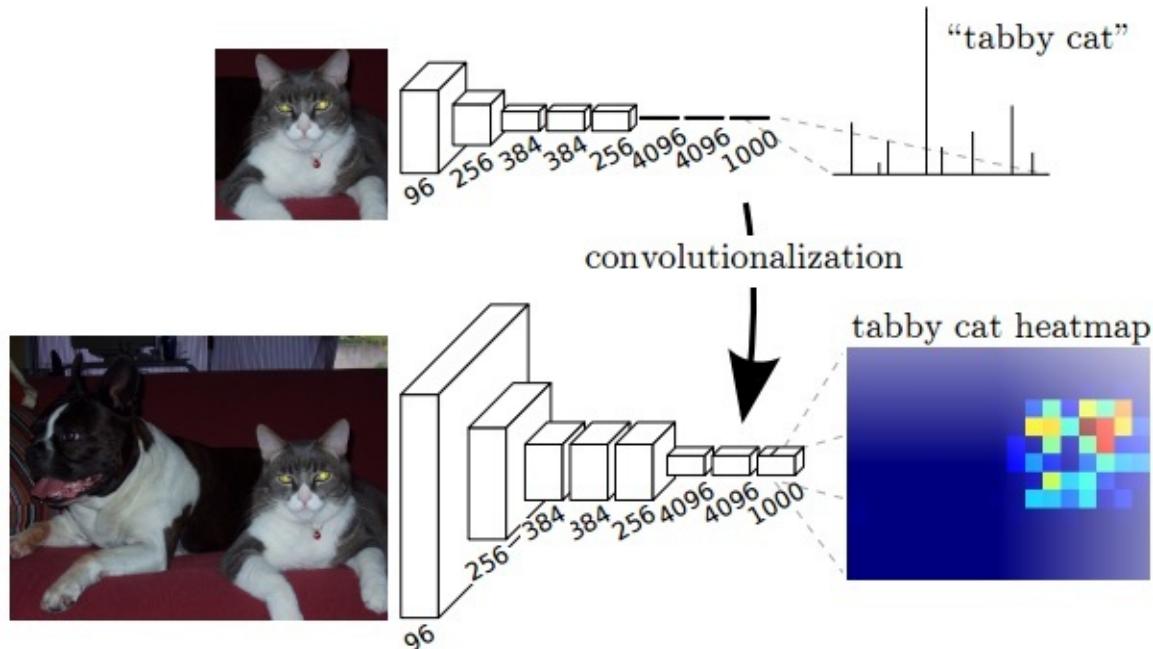


Fully Convolutional network for segmentation

A Fully Convolutional neural network (FCN) is a normal CNN, where the last fully connected layer is substituted by another convolution layer with a large "receptive field". The idea is to capture the global context of the scene (Tell what we have on the image and also give some rude location where it is).



Just remember that when we convert our last fully connected (FC) layer to a convolutional layer we gain some form of localization if we look where we have more activations.

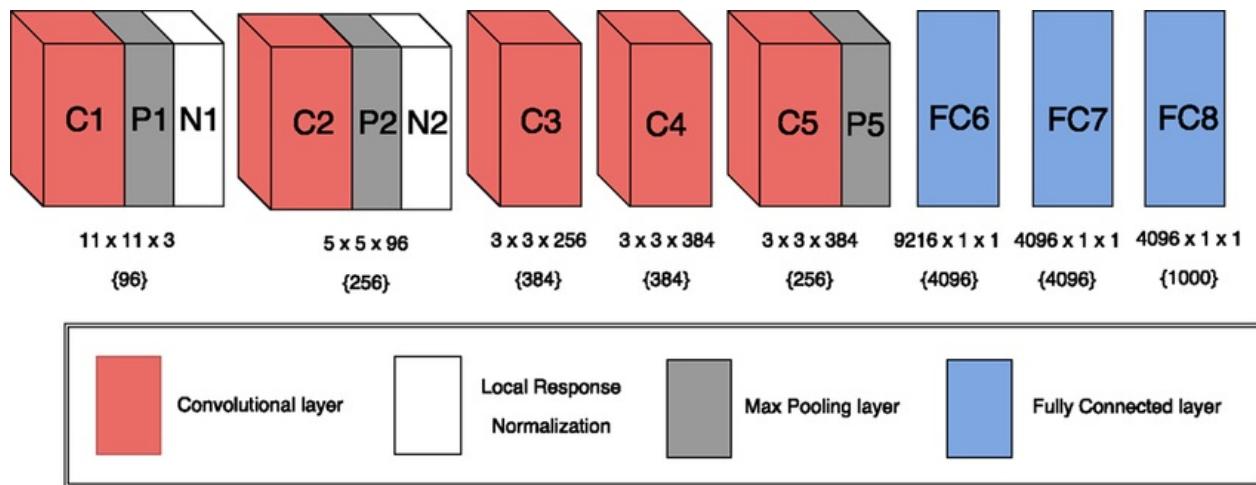


The idea is that if we choose our new last conv layer to be big enough we will have this localization effect scaled up to our input image size.

Conversion from normal CNN to FCN

Here is how we convert a normal CNN used for classification, ie: Alexnet to a FCN used for segmentation.

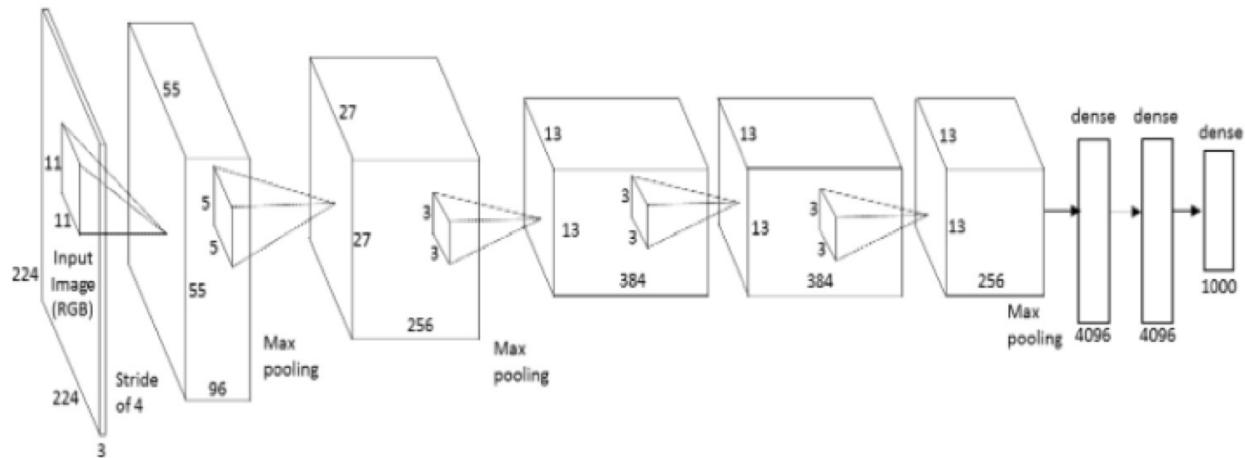
Just to remember this is how Alexnet looks like:



Below is also show the parameters for each layer

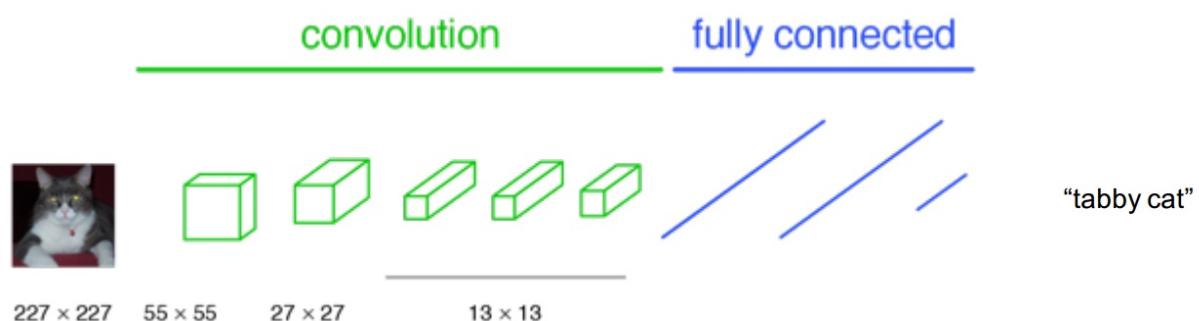
params	AlexNet	FLOPs
4M	FC 1000	4M
16M	FC 4096 / ReLU	16M
37M	FC 4096 / ReLU	37M
	Max Pool 3x3s2	
442K	Conv 3x3s1, 256 / ReLU	74M
1.3M	Conv 3x3s1, 384 / ReLU	112M
884K	Conv 3x3s1, 384 / ReLU	149M
	Max Pool 3x3s2	
	Local Response Norm	
307K	Conv 5x5s1, 256 / ReLU	223M
	Max Pool 3x3s2	
	Local Response Norm	
35K	Conv 11x11s4, 96 / ReLU	105M

On Alexnet the inputs are fixed to be 227x227, so all the pooling effects will scale down the image from 227x227 to 55x55, 27x27, 13x13, then finally a single row vector on the FC layers.

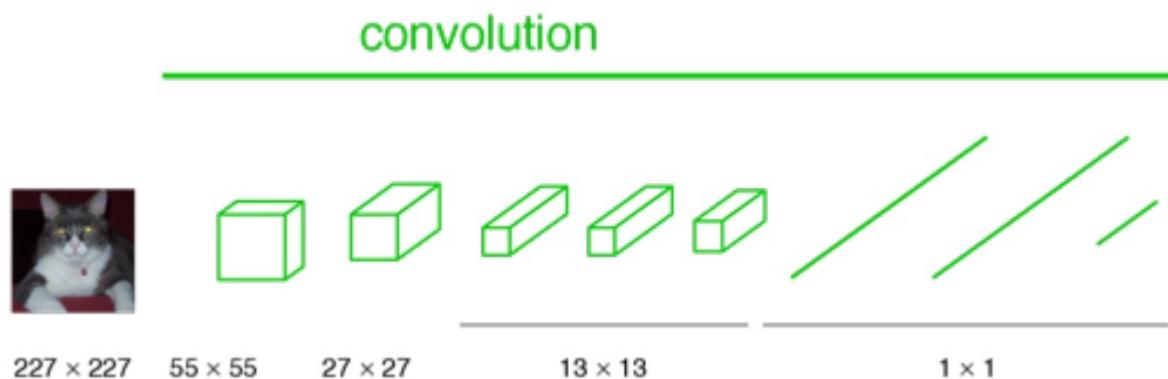


Now let's look on the steps needed to do the conversion.

- 1) We start with a normal CNN for classification with

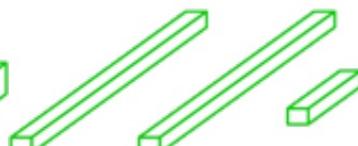


- 2) The second step is to convert all the FC layers to convolution layers 1x1 we don't even need to change the weights at this point. (This is already a fully convolutional neural network). The nice property of FCN networks is that we can now use any image size.



Observe here that with a FCN we can use a different size H x N.

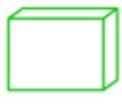
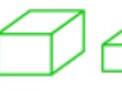
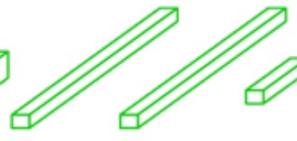
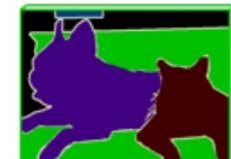
convolution

 $H \times W$  $H/4 \times W/4$  $H/8 \times W/8$  $H/16 \times W/16$  $H/32 \times W/32$

3) The last step is to use a "deconv or transposed convolution" layer to recover the activation positions to something meaningful related to the image size. Imagine that we're just scaling up the activation size to the same image size.

This last "upsampling" layer also have lernable parameters.

convolution

 $H \times W$  $H/4 \times W/4$  $H/8 \times W/8$  $H/16 \times W/16$  $H/32 \times W/32$  $H \times W$

↑
conv, pool,
nonlinearity

↑
upsampling
↑
pixelwise
output + loss

Now with this structure we just need to find some "ground truth" and to end to end learning, starting from e pre-trainned network ie: Imagenet.

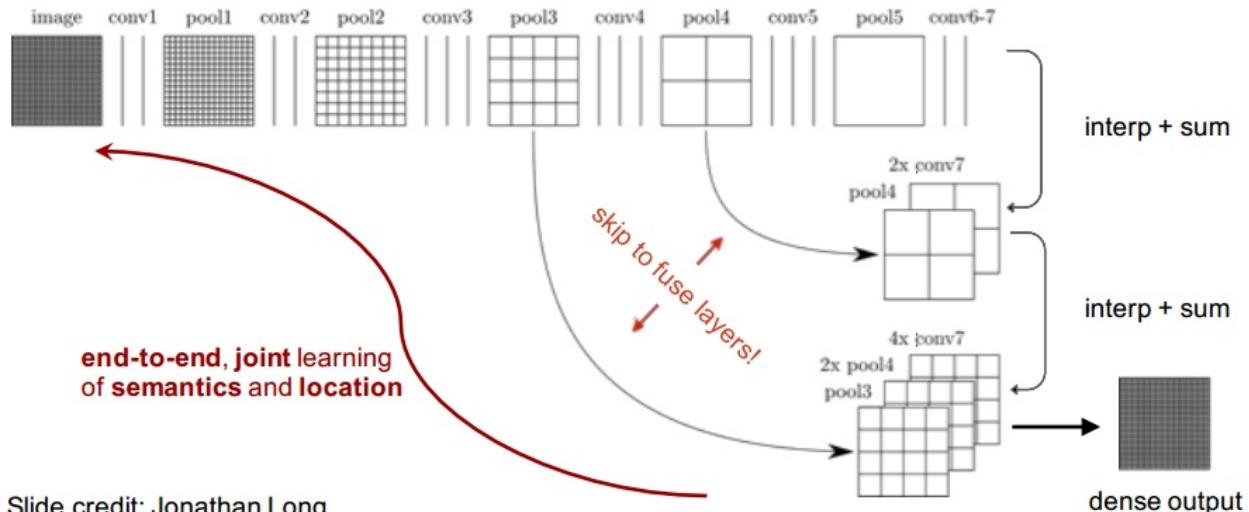
The problem with this approach is that we loose some resolution by just doing this because the activations were downsampled on a lot of steps.



To solve this problem we also get some activation from previous layers and sum them together. This process is called "skip" from the creators of this algorithm.

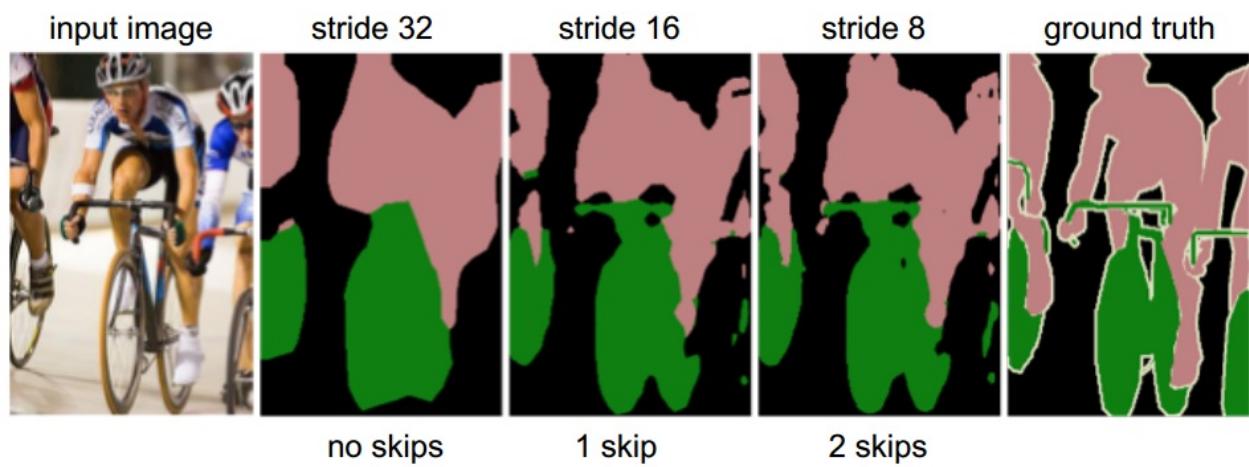
Even today (2016) the winners on Imagenet on the Segmentation category, used an ensemble of FCN to win the competition.

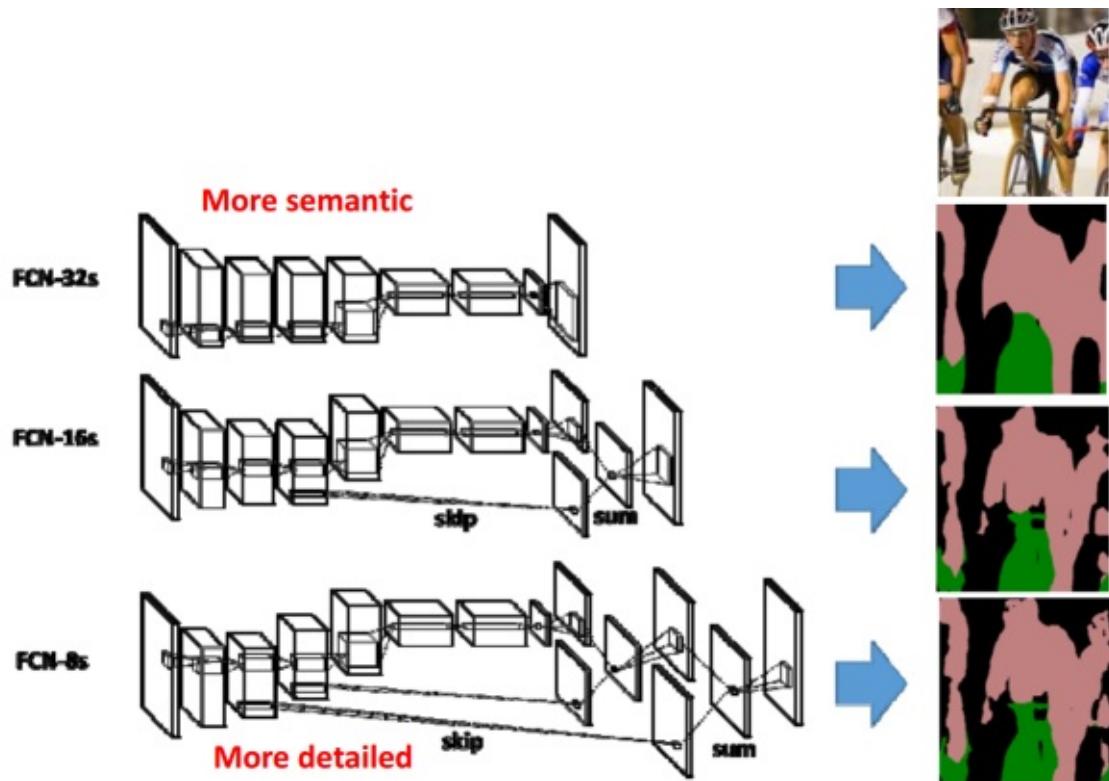
Those up-sampling operations used on skip are also learnable.



Slide credit: Jonathan Long

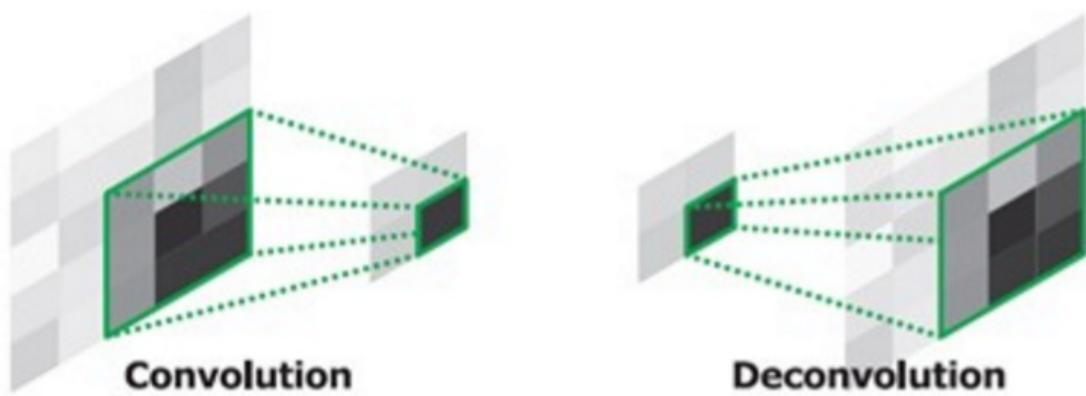
Below we show the effects of this "skip" process notice how the resolution of the segmentation improves after some "skips"

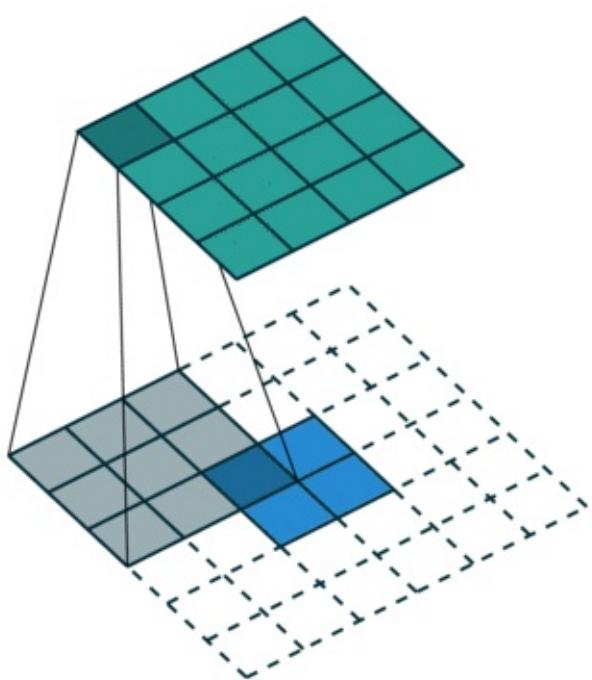
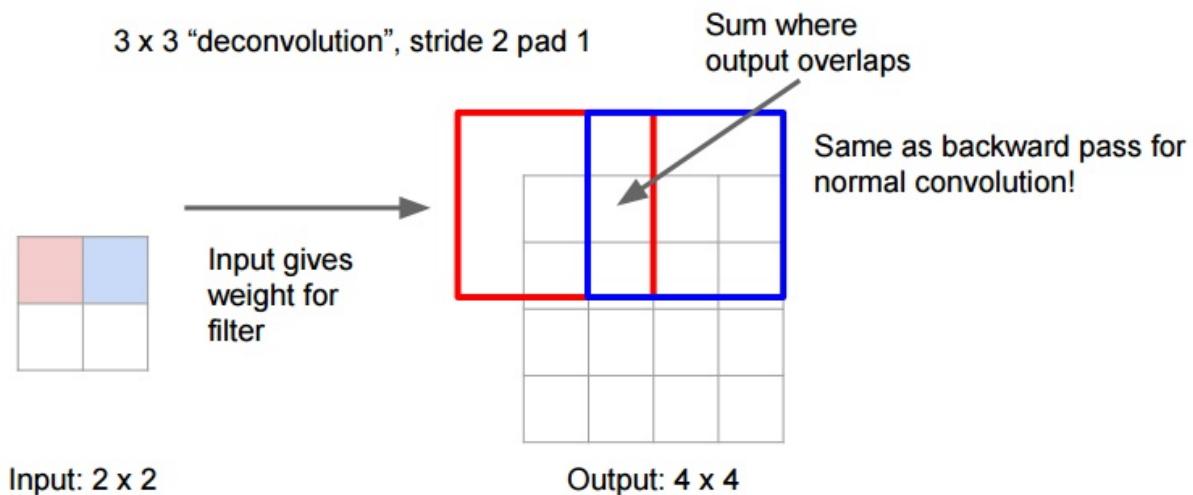




Transposed convolution layer (deconvolution "bad name")

Basically the idea is to scale up, the scale down effect made on all previous layers.

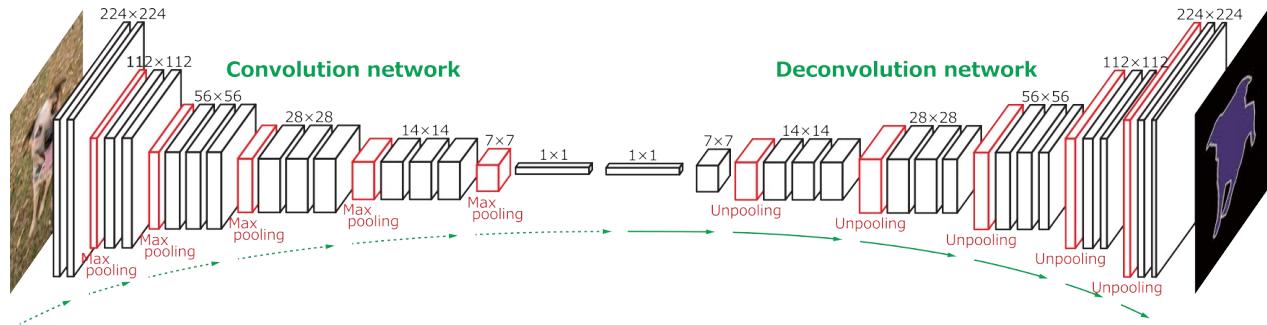




It has this bad name because the upsampling forward propagation is the convolution backpropagation and the upsampling backpropagation is the convolution forward propagation.
Also in caffe source code it is wrong called "deconvolution"

Extreme segmentation

There is another thing that we can do to avoid those "skiping" steps and also give better segmentation.

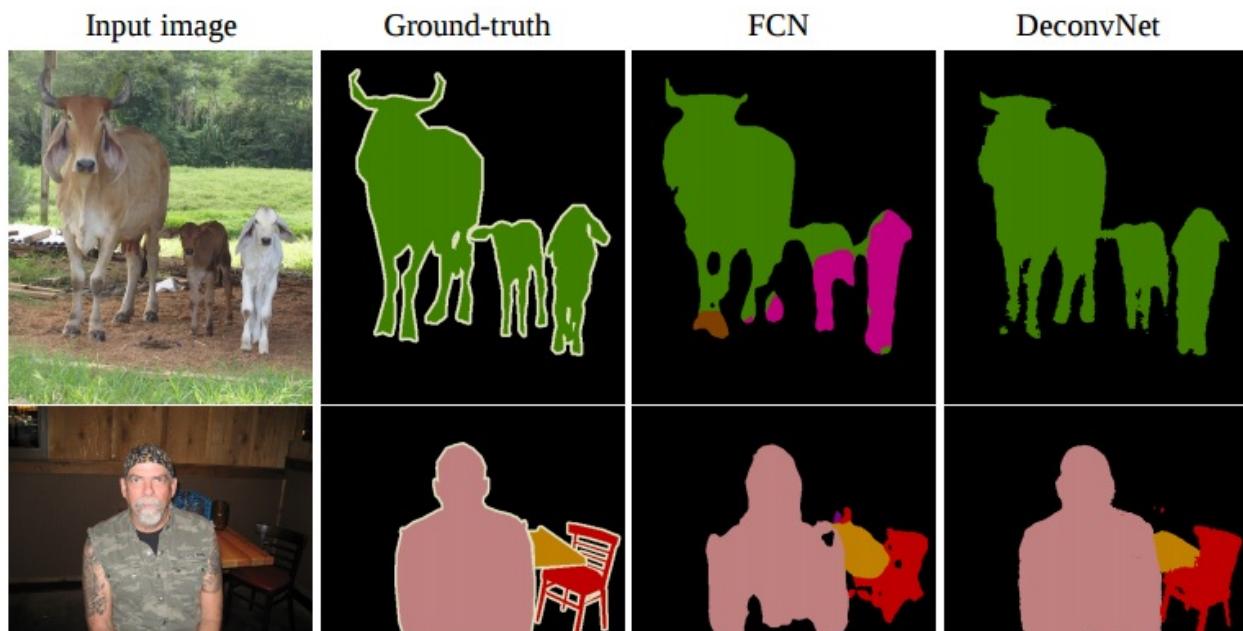


This architecture is called "Deconvnet" which is basically another network but now with all convolution and pooling layers reversed. As you may suspect this is heavy, it takes 6 days to train on a TitanX. But the results are really good.

Another problem is that the training is made in 2 stages.

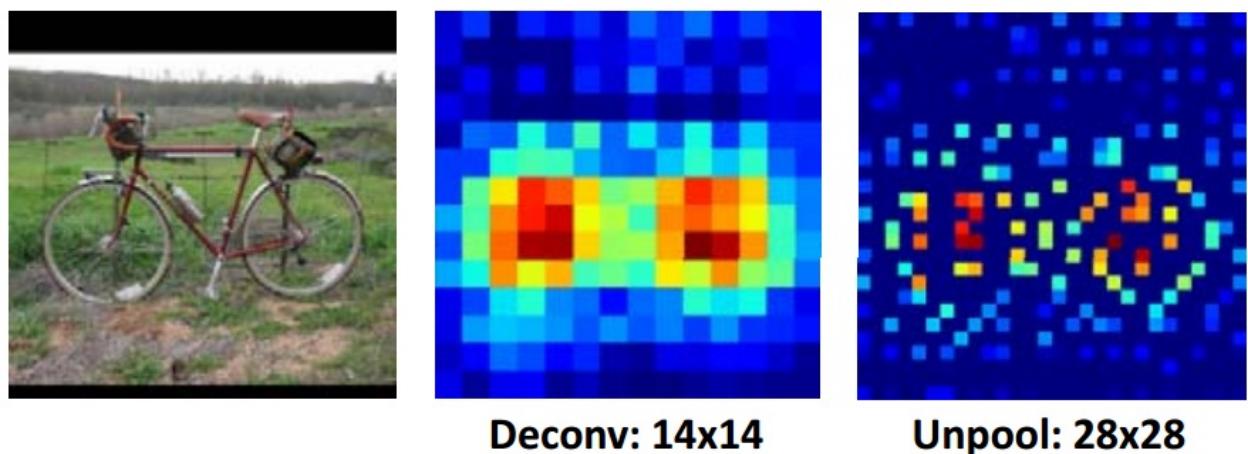
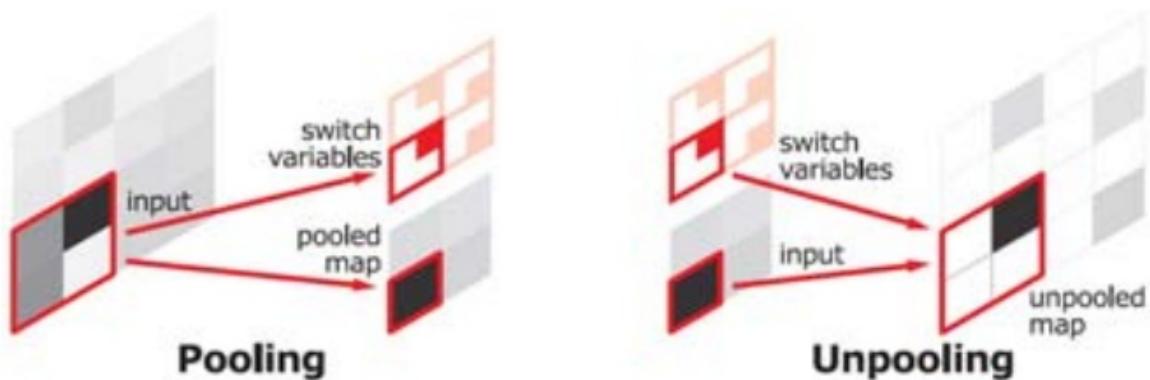
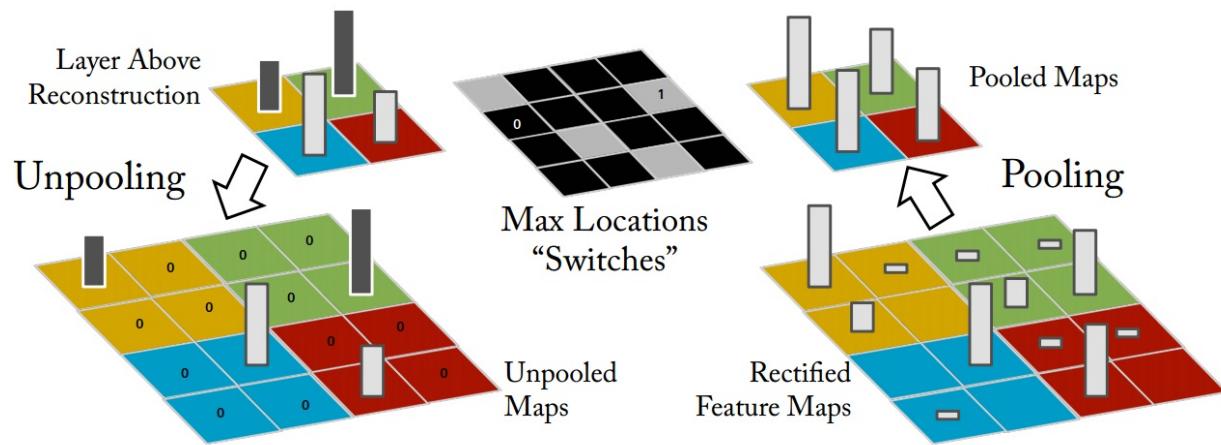
Also Deconvnets suffer less than FCN when there are small objects on the scene.

The deconvolution network output a probability map with the same size as the input.



Unpooling

Besides the deconvolution layer we also need now the unpooling layer. The max-pooling operation is non-invertible, but we can approximate, by recording the positions (Max Location switches) where we located the biggest values (during normal max-pool), then use this positions to reconstruct the data from the layer above (on this case a deconvolution)



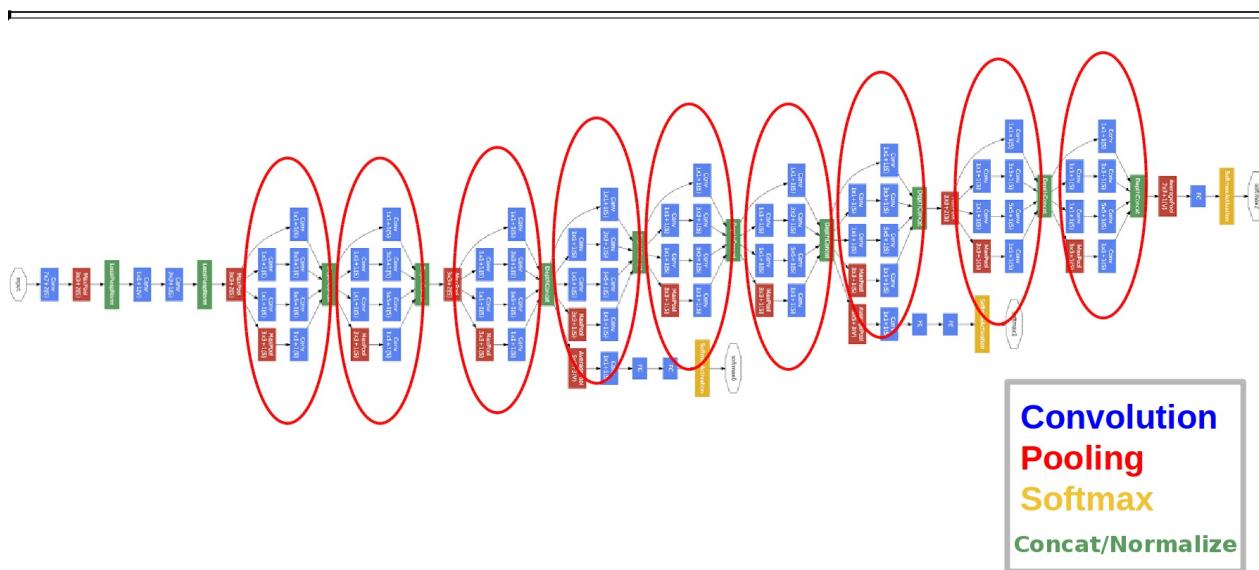
Next Chapter

On the next chapter we will discuss some libraries that support deep learning

GoogleNet

GoogleNet

Introduction



On this chapter you will learn about the googleNet (Winning architecture on ImageNet 2014) and it's inception layers.

2015 ResNet (ILSVRC'15) 3.57

Year	Codename	Error (percent)	99.9% Conf Int
2014	GoogLeNet	6.66	6.40 - 6.92
2014	VGG	7.32	7.05 - 7.60
2014	MSRA	8.06	7.78 - 8.34
2014	AHoward	8.11	7.83 - 8.39
2014	DeeperVision	9.51	9.21 - 9.82
2013	Clarifai [†]	11.20	10.87 - 11.53
2014	CASIAWS [†]	11.36	11.03 - 11.69
2014	Trimp [†]	11.46	11.13 - 11.80
2014	Adobe [†]	11.58	11.25 - 11.91
2013	Clarifai	11.74	11.41 - 12.08
2013	NUS	12.95	12.60 - 13.30
2013	ZF	13.51	13.14 - 13.87
2013	AHoward	13.55	13.20 - 13.91
2013	OverFeat	14.18	13.83 - 14.54
2014	Orange [†]	14.80	14.43 - 15.17
2012	SuperVision [†]	15.32	14.94 - 15.69
2012	SuperVision	16.42	16.04 - 16.80
2012	ISI	26.17	25.71 - 26.65
2012	VGG	26.98	26.53 - 27.43
2012	XRCE	27.06	26.60 - 27.52
2012	UvA	29.58	29.09 - 30.04

human error is around 5.1% on a subset

googleNet has 22 layer, and almost 12x less parameters (So faster and less then Alexnet and much more accurate).

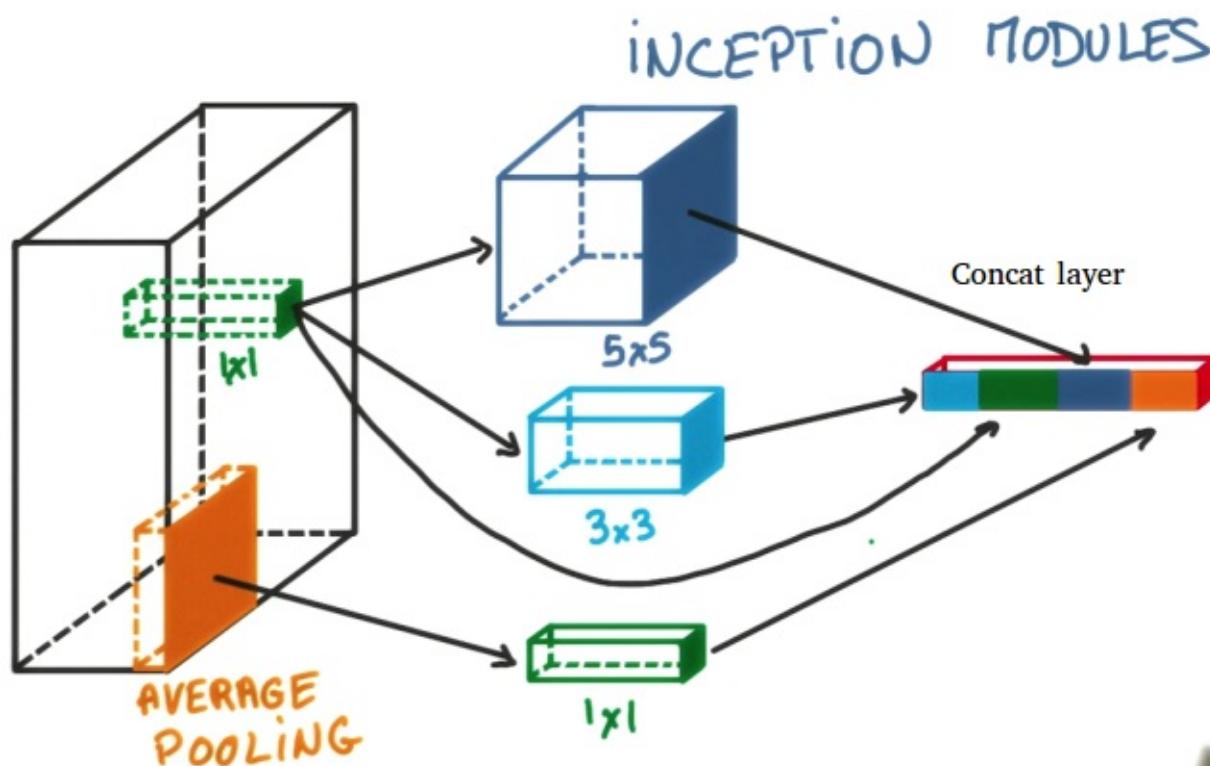
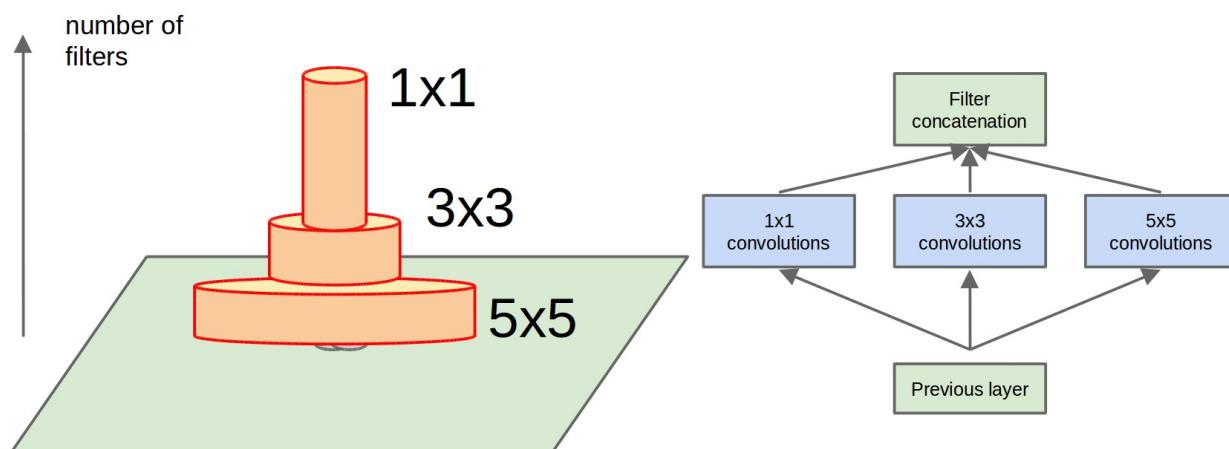
Their idea was to make a model that also could be used on a smart-phone (Keep calculation budget around 1.5 billion multiply-adds on prediction).

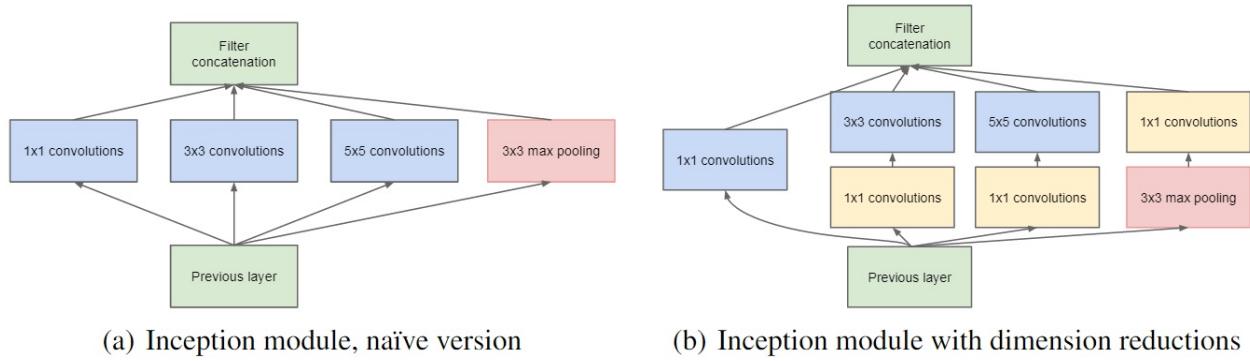
Inception Layer

The idea of the inception layer is to cover a bigger area, but also keep a fine resolution for small information on the images. So the idea is to convolve in parallel different sizes from the most accurate detailing (1x1) to a bigger one (5x5).

The idea is that a series of gabor filters with different sizes, will handle better multiple objects scales. With the advantage that all filters on the inception layer are learnable.

The most straightforward way to improve performance on deep learning is to use more layers and more data, googleNet use 9 inception modules. The problem is that more parameters also means that your model is more prone to overfit. So to avoid a parameter explosion on the inception layers, all bottleneck techniques are exploited.

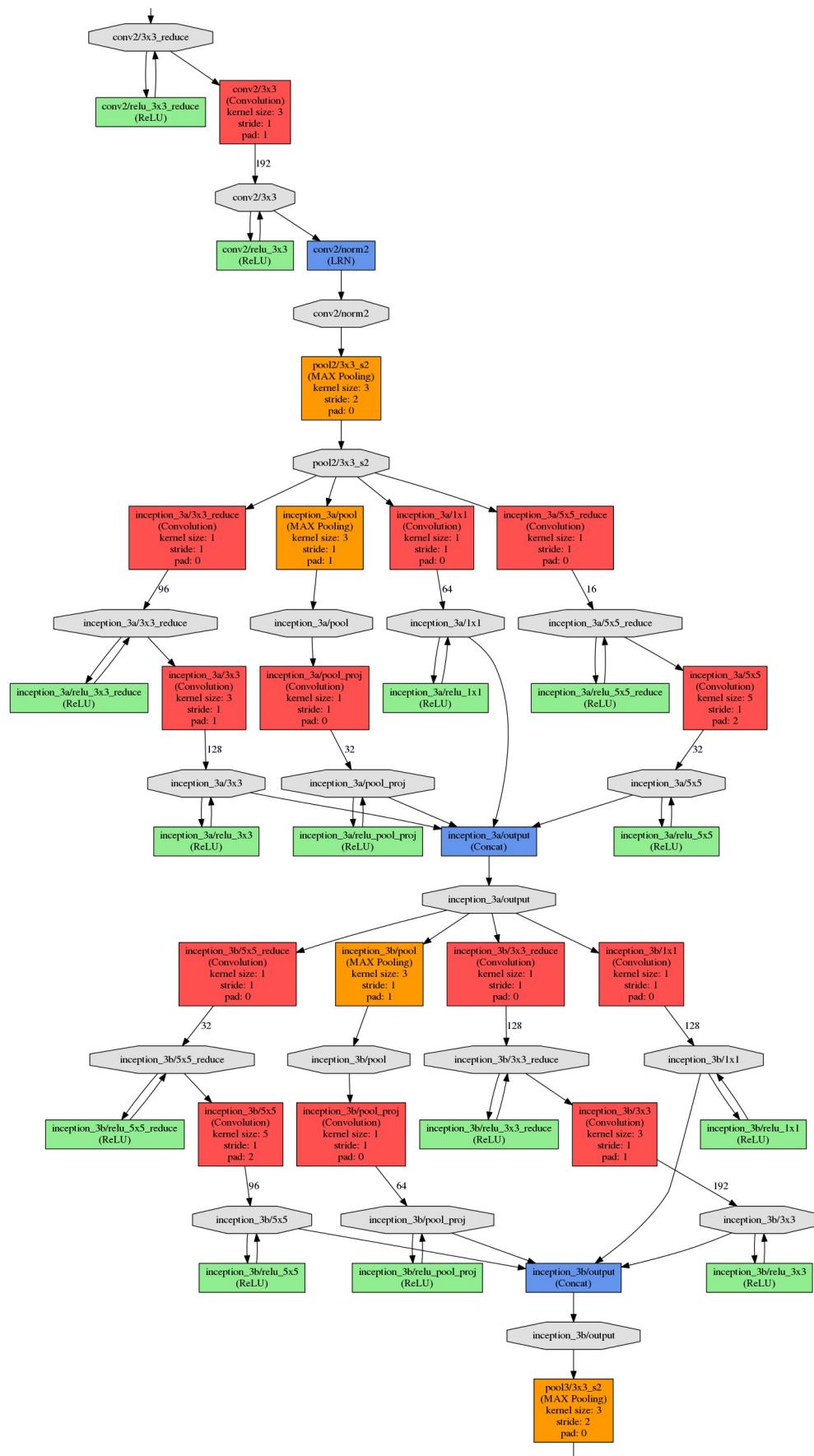




Using the bottleneck approaches we can rebuild the inception module with more non-linearities and less parameters. Also a max pooling layer is added to summarize the content of the previous layer. All the results are concatenated one after the other, and given to the next layer.

Caffe Example

Bellow we present 2 inception layers on cascade from the original googleNet.



Residual Net

Residual Net

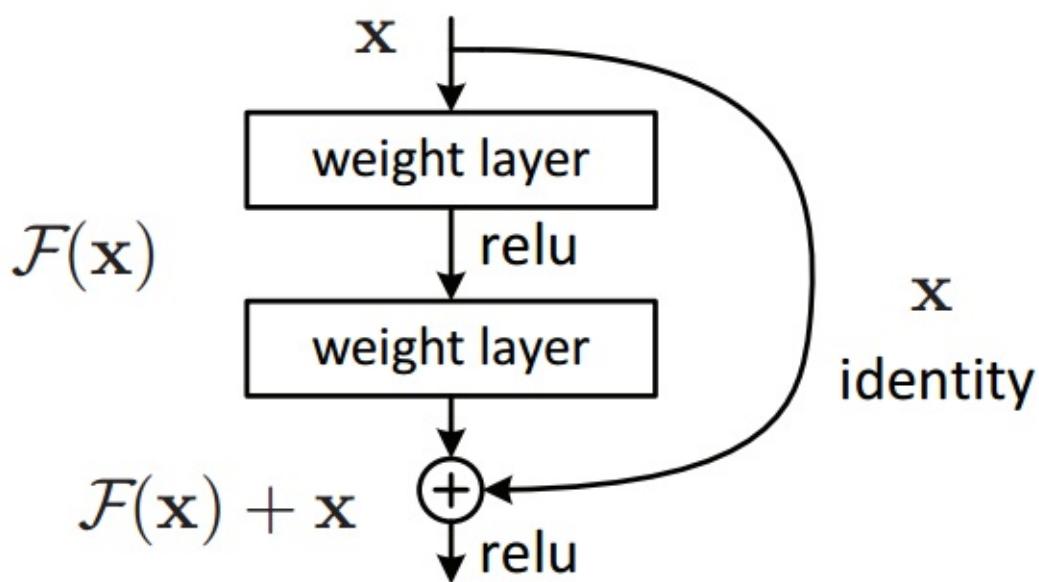
Introduction

This chapter will present the 2016 state of the art on object classification. The ResidualNet it's basically a 150 deep convolution neural network made by equal "residual" blocks.

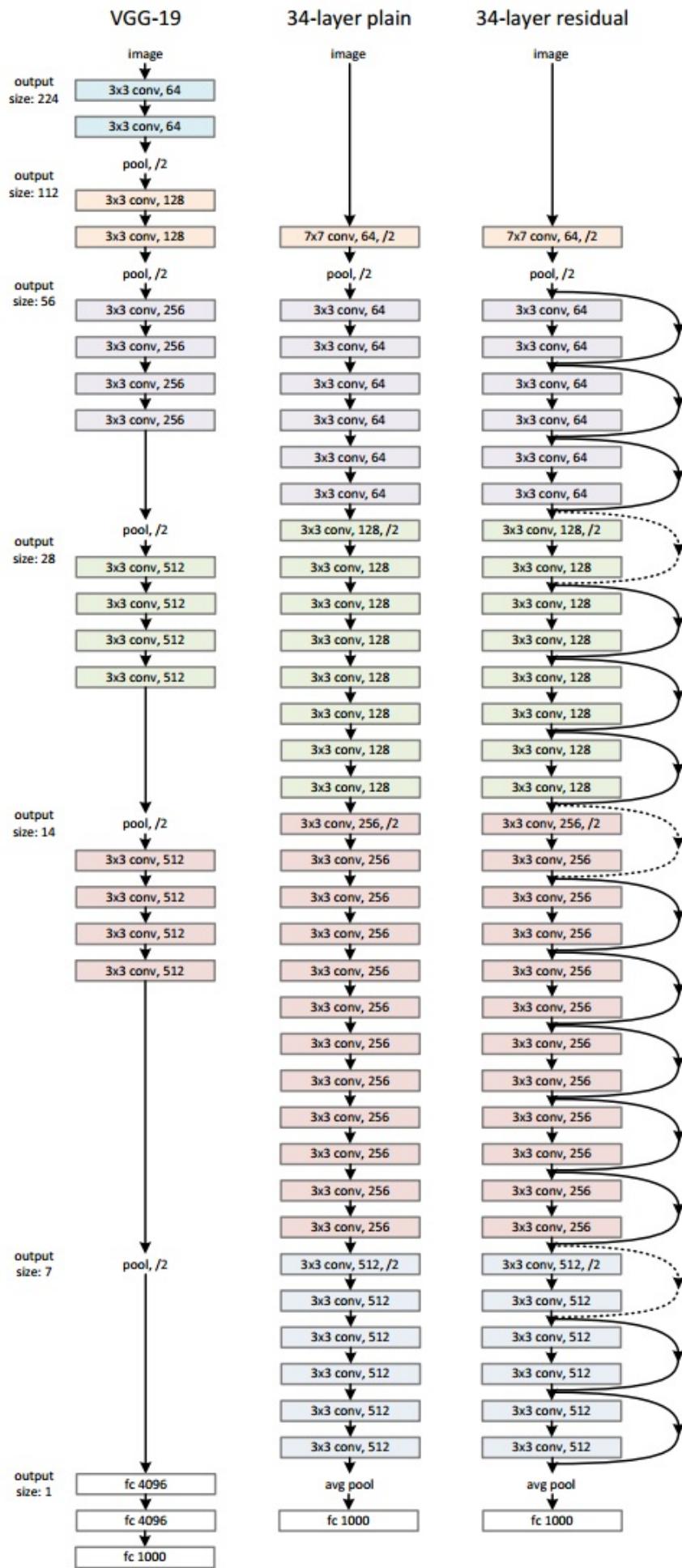
The problem is for real deep networks (more than 30 layers), all the known techniques (Relu, dropout, batch-norm, etc...) are not enough to do a good end-to-end training. This contrast with the common "empirical proven knowledge" that deeper is better.

The idea of the residual network is use blocks that re-route the input, and add to the concept learned from the previous layer. The idea is that during learning the next layer will learn the concepts of the previous layer plus the input of that previous layer. This would work better than just learn a concept without a reference that was used to learn that concept.

Another way to visualize their solution is remember that the back-propagation of a sum node will replicate the input gradient with no degradation.



Bellow we show an example of a 34-deep residual net.

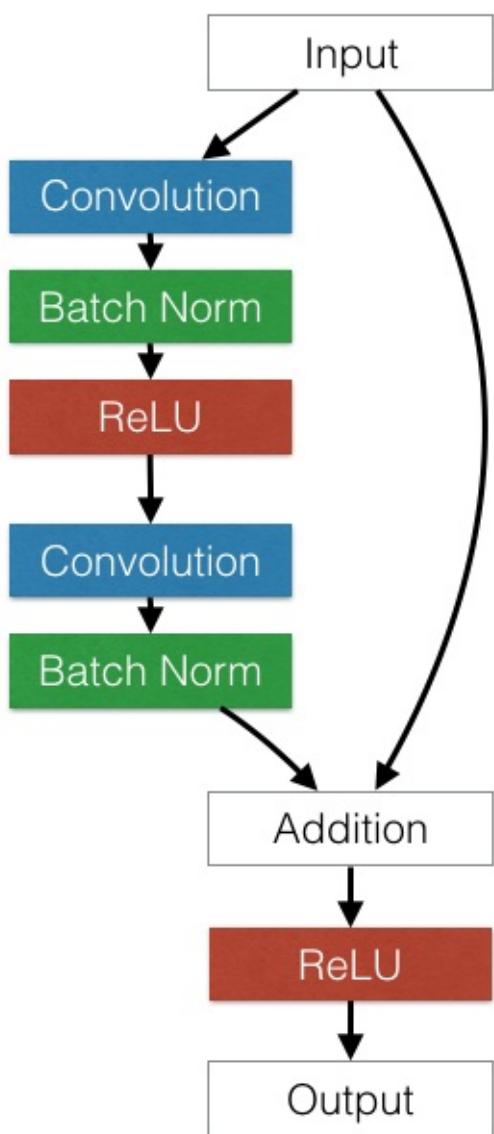


The ResidualNet creators proved empirically that it's easier to train a 34-layer residual compared to a 34-layer cascaded (Like VGG).

Observe that on the end of the residual net there is only one fully connected layer followed by a previous average pool.

Residual Block

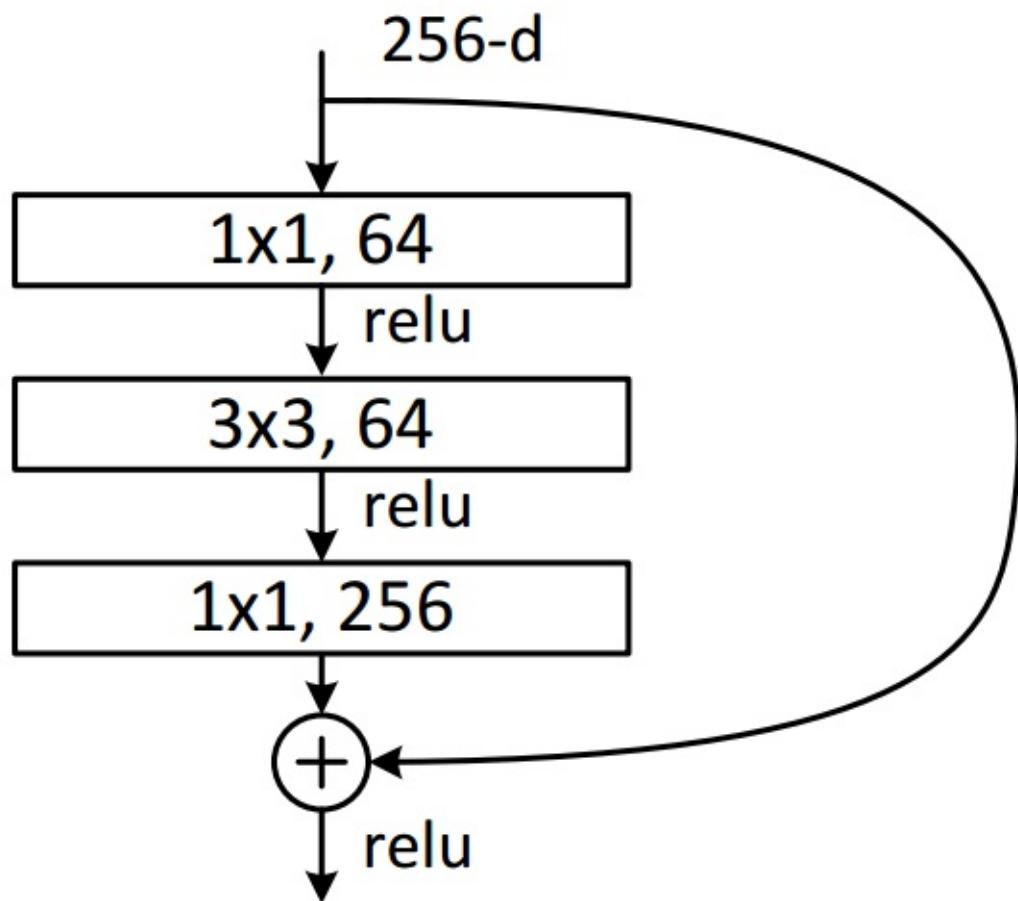
At its core the residual net is formed by the following structure.



Basically this jump and adder creates a path for back-propagation, allowing even really deep models to be trained.

As mentioned before the Batch-Norm block alleviates the network initialization, but it can be omitted for not so deep models (less than 50 layers).

Again like googlenet we must use bottlenecks to avoid a parameter explosion.

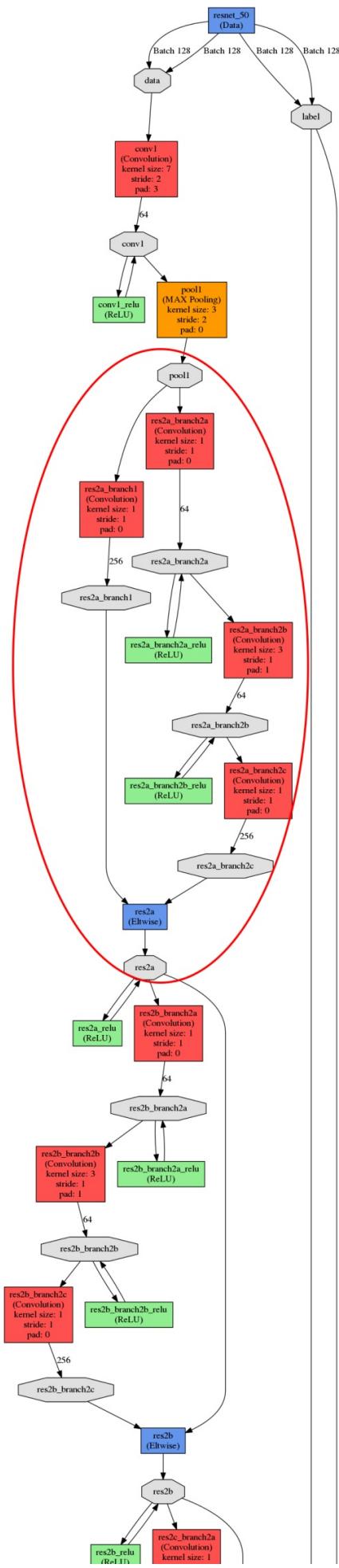


Just to remember for the bottleneck to work the previous layer must have same depth.

Caffe Example

Here we show 2 cascaded residual blocks form residual net, due to difficulties with batch-norm layers, they were omitted but still residual net gives good results.

Residual Net



Deep Learning Libraries

Deep Learning Libraries

Introduction

Discussion, and some examples on the most common deep learning libraries:

- Caffe
- Torch
- TensorFlow
- Theano
- CNTK

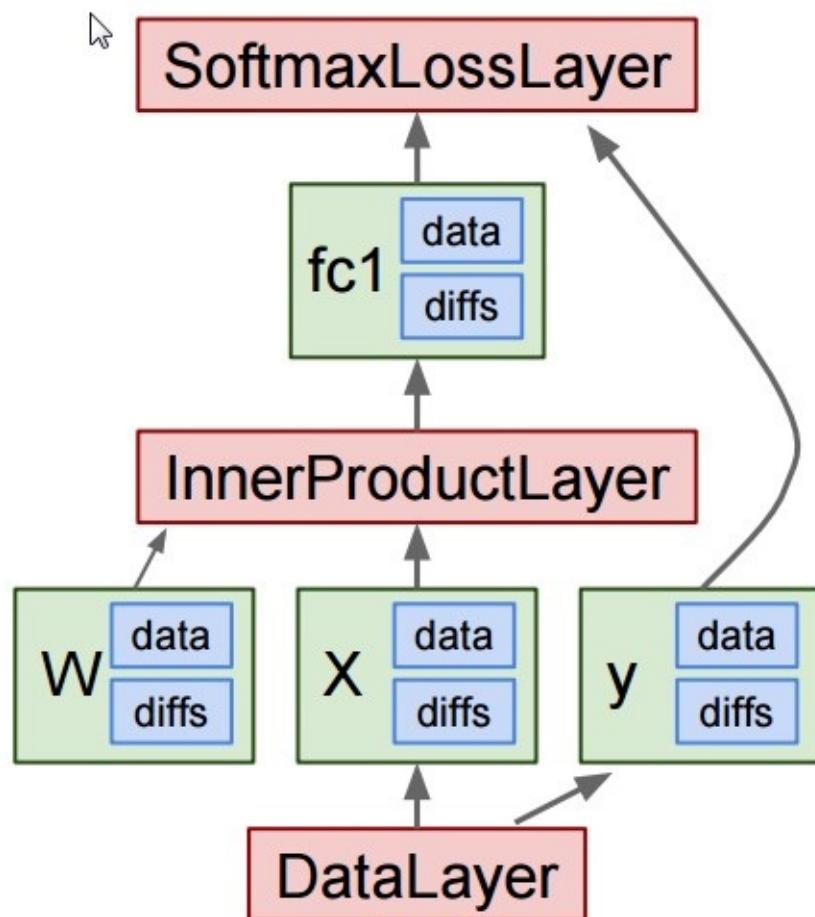
Caffe

One of the most basic characteristic of caffe is that is easy to train simple non recurrent models.

Most cool features:

- Good Performance, allows training with multiple GPUs
- Implementation for CPU and GPU
- Source code is easy to read
- Allow layer definition in Python
- Has bindings for Python and Matlab
- Allows network definition with text language (No need to write code)
- Fast dataset access through LMDB
- Allows network visualization
- Has web interface (Digits)

Caffe Main classes:



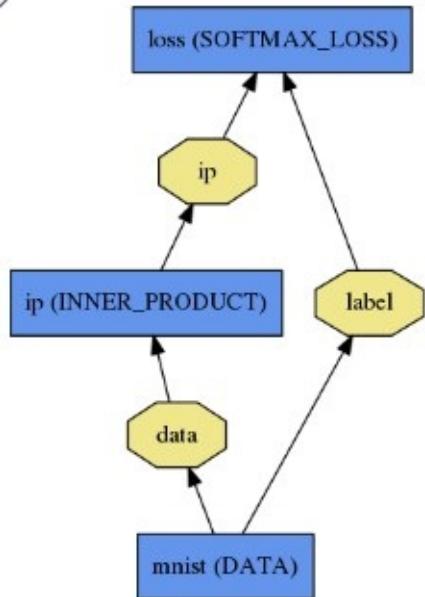
- Blob: Used to store data and diffs(Derivatives)
- Layer: Some operation that transform a bottom blob(input) to top blobs(outputs)
- Net: Set of connected layers
- Solver: Call Net forward and backward propagation, update weights using gradient methods (Gradient descent, SGD, adagrad, etc...)

Caffe training/validation files

path/to/image/1.jpg [label]

Simple example

Here a logistic regression classifier. Imagine as a neural network with one layer and a sigmoid (cross-entropy softmax) non-linearity.



```

name: "LogisticRegressionNet"
layers {
    top: "data" ← Layers and Blobs
    top: "label" ← often have same
    name: "data" ← name!
    type: HDF5_DATA
    hdf5_data_param {
        source: "examples/hdf5_classification/data/train.txt"
        batch_size: 10
    }
    include {
        phase: TRAIN
    }
}
layers {
    bottom: "data"
    top: "fc1"
    name: "fc1"
    type: INNER_PRODUCT
    blobs_lr: 1
    blobs_lr: 2
    weight_decay: 1
    weight_decay: 0
}
  
```

← Layers and Blobs
← often have same
← name!

```

inner_product_param {
    num_output: 2
    weight_filler {
        type: "gaussian"
        std: 0.01
    }
    bias_filler {
        type: "constant"
        value: 0
    }
}
layers {
    bottom: "fc1"
    top: "loss"
    name: "loss"
    type: SOFTMAX_LOSS
}
  
```

Caffe Cons

- Need to write C++ / Cuda code for new layers
- Bad to write protos for big networks (Resnet, googlenet)
- Bad to experience new architectures (Mainstream version does not support Fast RCNN)

Torch

Really good for research, the problem is that use a new language called Lua.

Torch Pros

- Flexible
- Very easy source code
- Easy bidding with C/C++
- Web interface (Digits)

Torch Cons

- New language Lua
- Difficult to load data from directories
- No Matlab bindings
- Less Plug and play than caffe
- Not easy for RNN

Theano

Theano Cons

- More manual
- No matlab bidding
- Slower than other frameworks
- No much pre-trained models

Tensorflow

Tensorflow Pros

- Flexible
- Good for RNN
- Allow distributed training
- Tensorboard for signal visualization
- Python Numpy

Tensorflow Cons

- Not much pre-trained models
- No Support for new object detection features (Ex Roi pooling)
- No support for datasets like Caffe
- Slower than Caffe for single GPU training

CNTK

CNTK Pros

- Flexible
- Good for RNN
- Allows distributed training

CNTK Cons

- No visualization
- Any error CNTK crash
- No simple source code to read
- New language (ndl) to describe networks
- No current matlab or python bindings

Summary

- For research use Torch or Tensorflow (Last option Theano)
- For training convnets or use pre-trained models use Caffe

CS231n Deep learning course summary

	Caffe	Torch	Theano	TensorFlow
Language	C++, Python	Lua	Python	Python
Pretrained	Yes ++	Yes ++	Yes (Lasagne)	Inception
Multi-GPU: Data parallel	Yes	Yes <small>cunn. DataParallelTable</small>	Yes <small>platoon</small>	Yes
Multi-GPU: Model parallel	No	Yes <small>fbcunn.ModelParallel</small>	Experimental	Yes (best)
Readable source code	Yes (C++)	Yes (Lua)	No	No
Good at RNN	No	Mediocre	Yes	Yes (best)

- Get features from known model (Alexnet, Googlenet, Vgg): Use caffe
- Fine tune known models (Alexnet, Googlenet, Vgg): Use Caffe
- Image Captioning: Torch or Tensorflow
- Segmentation: Caffe, Torch
- Object Detection: Caffe with python layers, Torch (More work)
- Language Modelling: Torch, Theano
- Implement Bath Norm: Torch, Theano or Tensorflow

Normally Tensorflow can be used in all cases that torch can, but if you need to understand what a specific layer does, or if you need to create a new layer, use torch instead of tensorflow. Torch is preferable on those cases, because the layer source code is more easy to read in torch.

Next Chapter

On the next chapter we will discuss Distributed Learning

Unsupervised Learning

Introduction

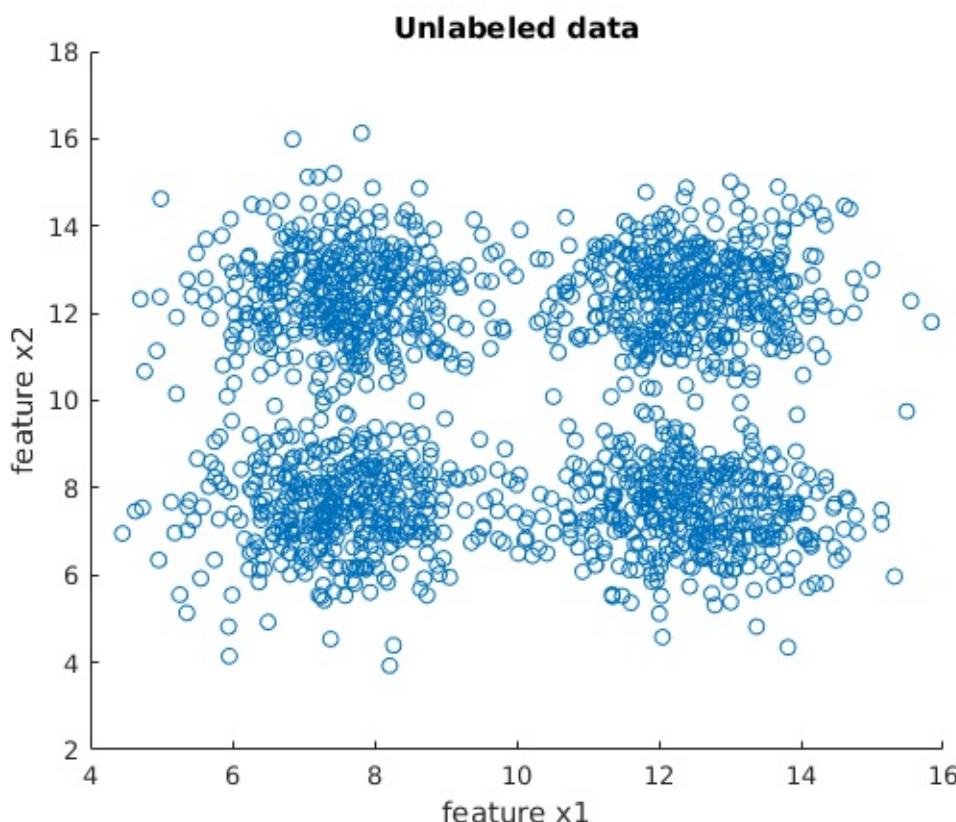
As mentioned on previous chapters, unsupervised learning is about learning information without the label information.

Here the term information means, "structure" for instance you would like to know how many groups exist in your dataset, even if you don't know what those groups mean.

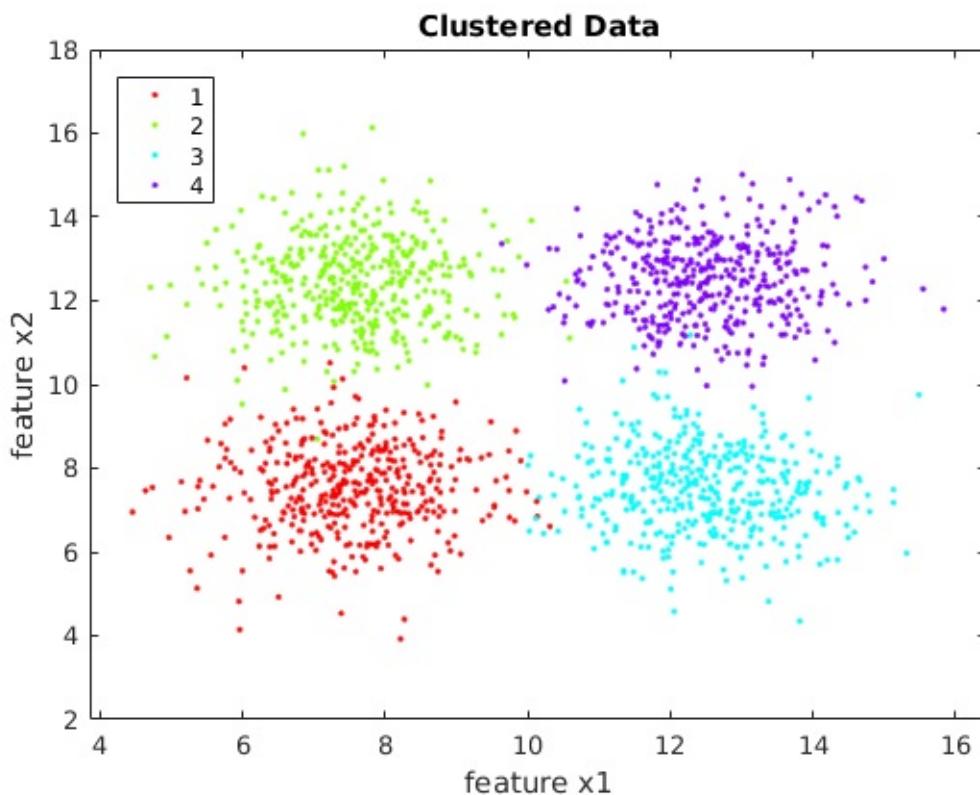
Also we use unsupervised learning to visualize your dataset, in order to try to learn some insight from the data.

Unlabeled data example

Consider the following dataset $X \in R^2$ (X has 2 features)



One type of unsupervised learning algorithm called "clustering" is used to infer how many distinct groups exist on your dataset.

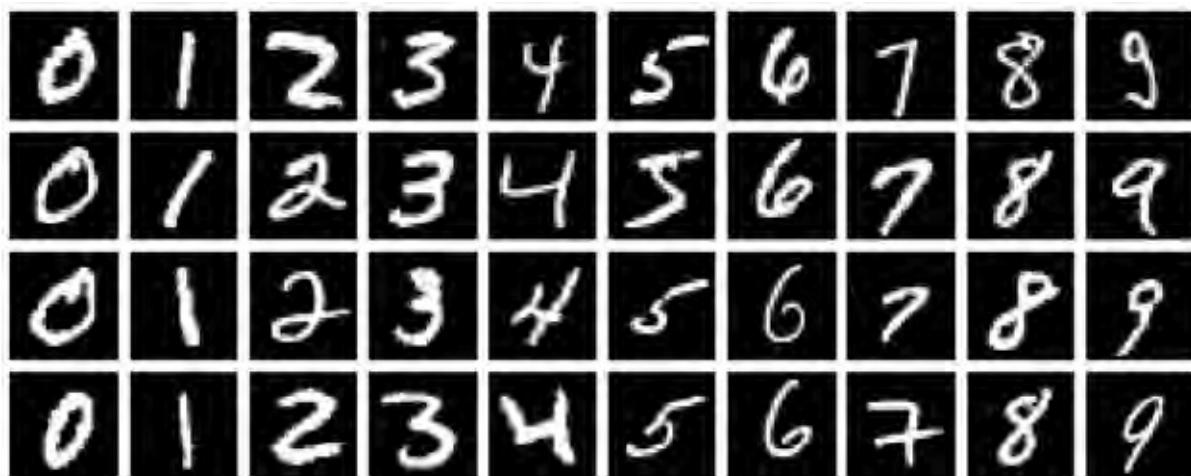


Here we still don't know what those groups means, but we know that there are 4 groups that seems very distinct. On this case we choose a low dimensional dataset R^2 but on real life it could be thousands of dimensions, ie R^{784} for a grayscale 28x28 image.

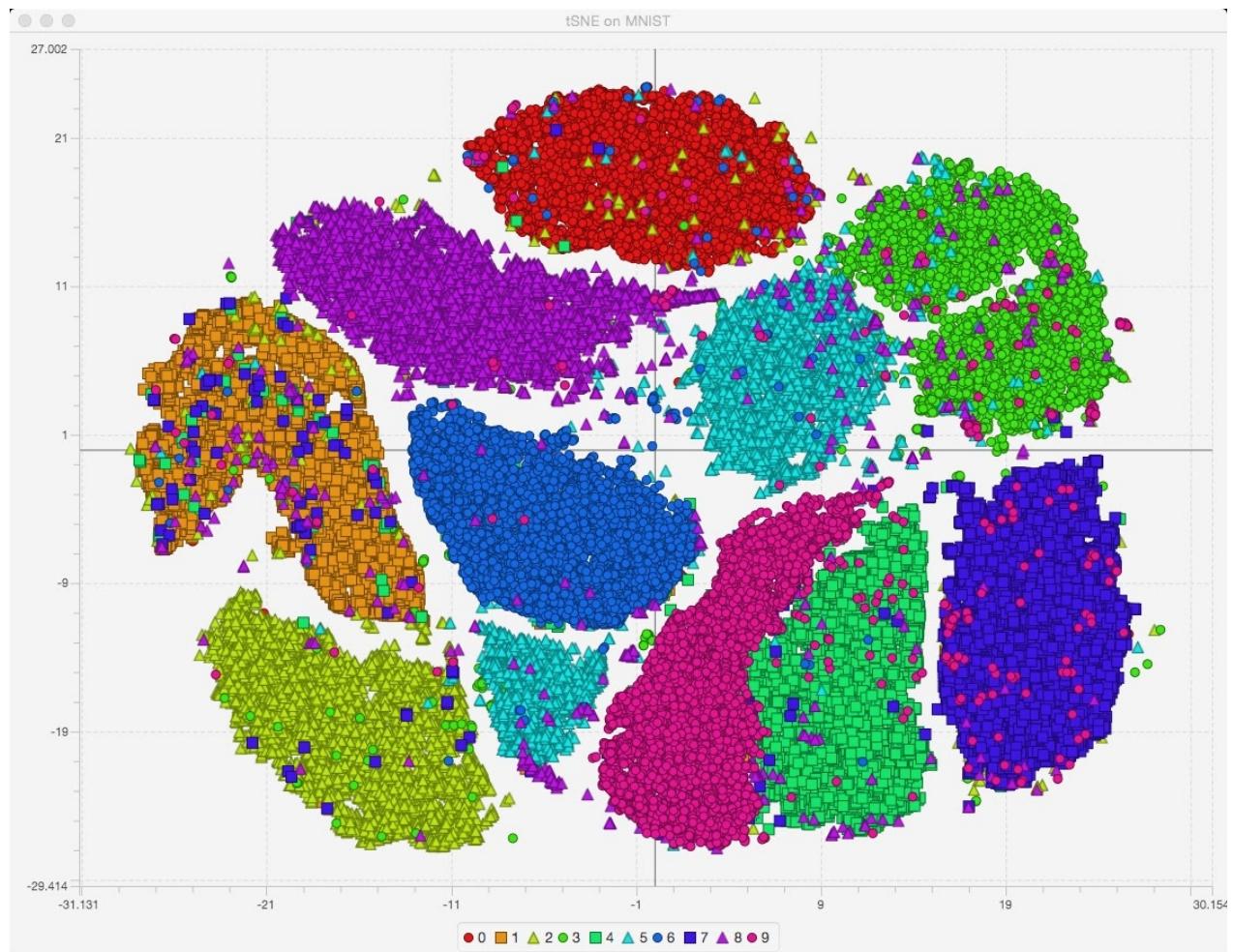
Dimensionality Reduction

In order to improve classification response time (not prediction performance) and sometimes for visualizing your high dimension dataset (2D, 3D), we use dimesionality reduction techniques (ie: PCA, T-Sne).

For example the [MNIST dataset](#) is composed with 60,000 training examples of (0..9) digits, each one with 784 dimensions. This high dimensionality is due to the fact that each digit is a 28x28 grayscale image.



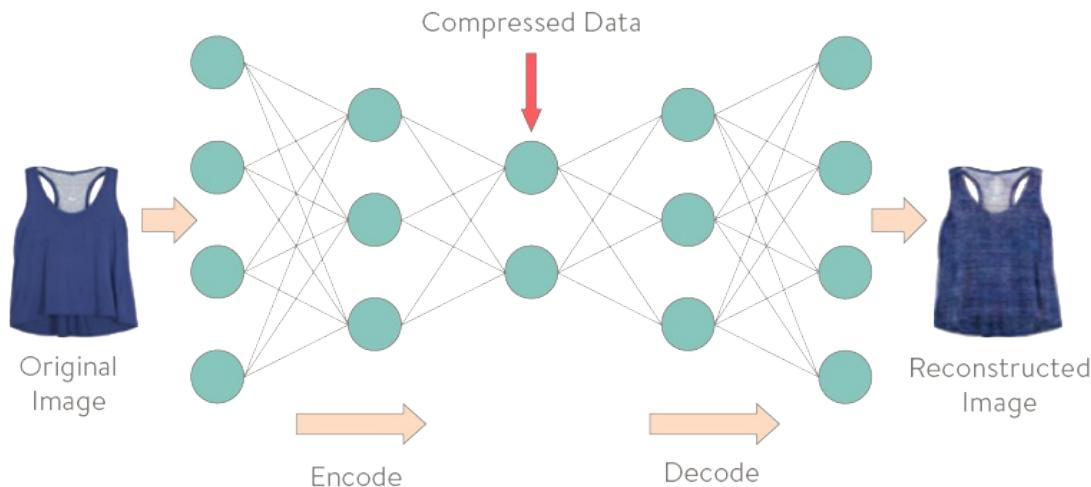
It would be difficult to visualize this dataset, so one option is to reduce its dimensions to something visible on the monitor (2D,3D).



Here is easy to observe that a classifier could have problems to differentiate the digit 1 and 7. Another advantage is that this gives us some hint on how good is our current set of features.

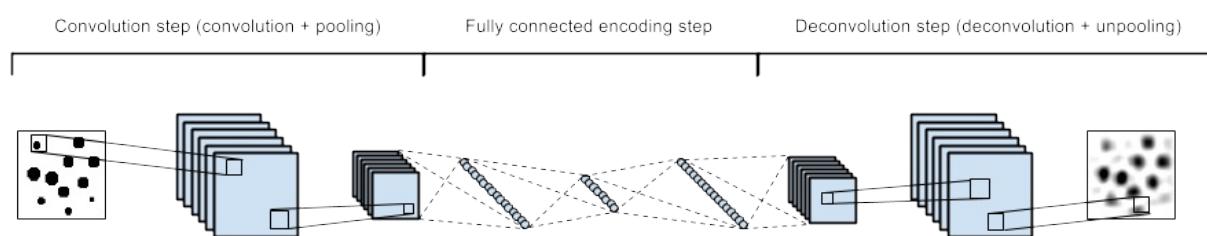
Autoencoders

We can also use neural networks to do dimensionality reduction the idea is that we have a neural network topology that approximate the input on the output layer. On the middle the autoencoder has smaller layer. After training the middle layer has a compressed version (lossy) of the input.



Convolution Neural network pre-train

As we don't need the label information to train autoencoders, we can use them as a pre-trainer to our convolution neural network. So in the future we can start your training with the weights initialized from unsupervised training.



Some examples of this technique can be found here:

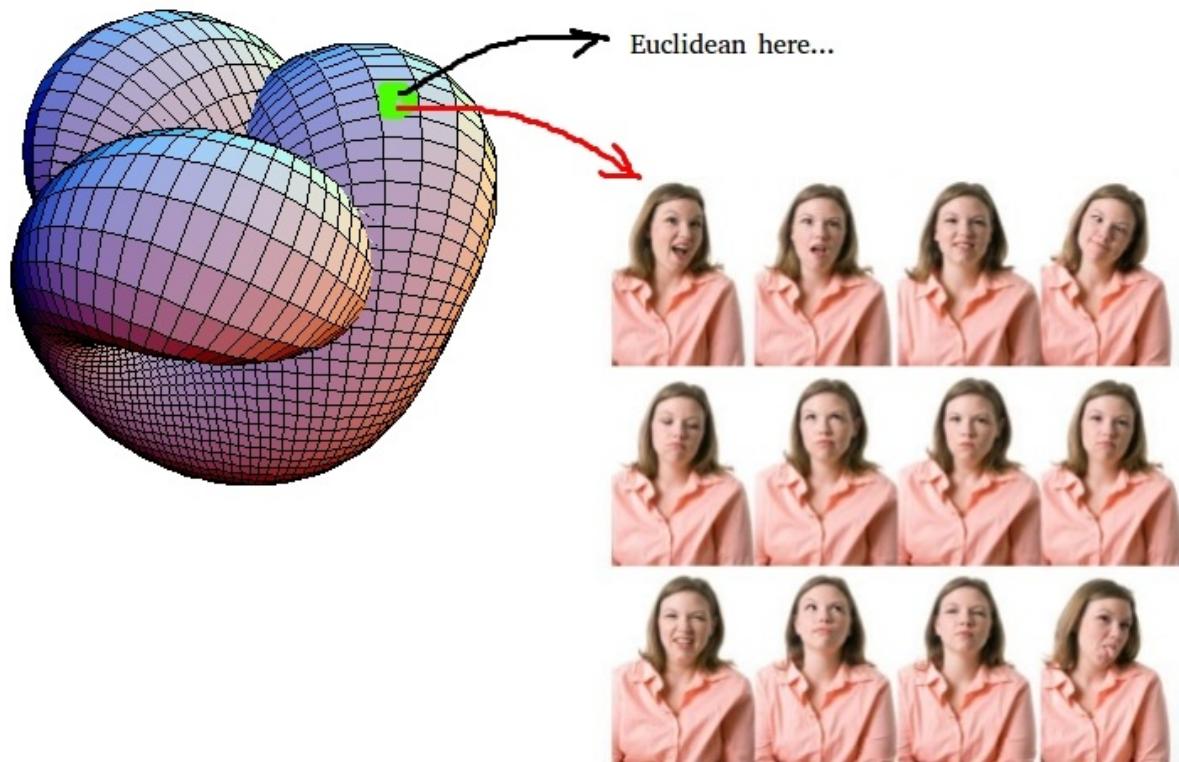
- [With Python](#)
- [With Torch](#)

Data Manifold

Manifold Learning pursues the goal to embed data that originally lies in a high dimensional space in a lower dimensional space, while preserving characteristic properties. This is possible because for any high dimensional data to be interesting, it must be intrinsically low dimensional.

For example, images of faces might be represented as points in a high dimensional space (let's say your camera has 5MP -- so your images, considering each pixel consists of three values [r,g,b], lie in a 15M dimensional space), but not every 5MP image is a face. Faces lie on a sub-manifold in this high dimensional space.

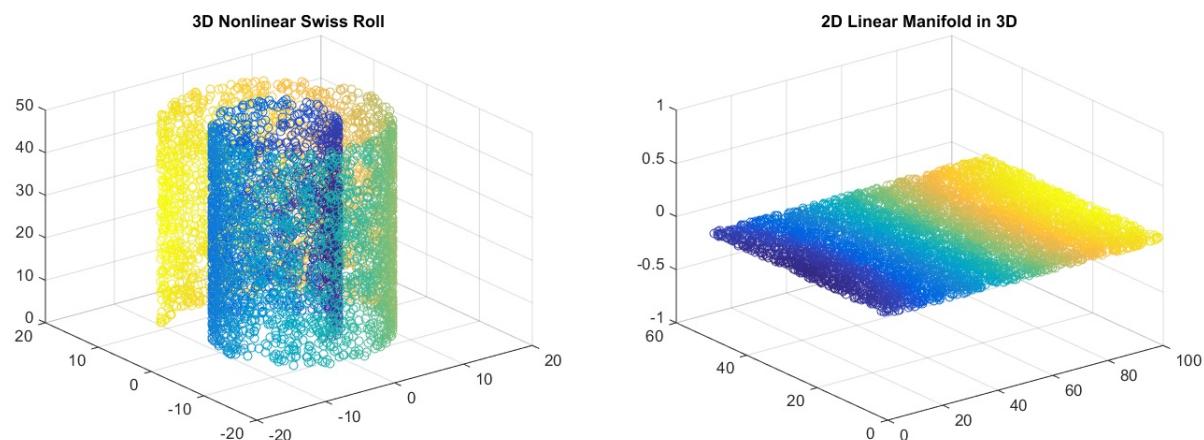
A sub-manifold is locally Euclidean, i.e. if you take two very similar points, for example two images of identical twins they will be close on the euclidian space



For example on the dataset above we have a high dimension manifold, but the faces sit's on a much lower dimension space (almost euclidian). So on this subspace things like distance has a meaning.

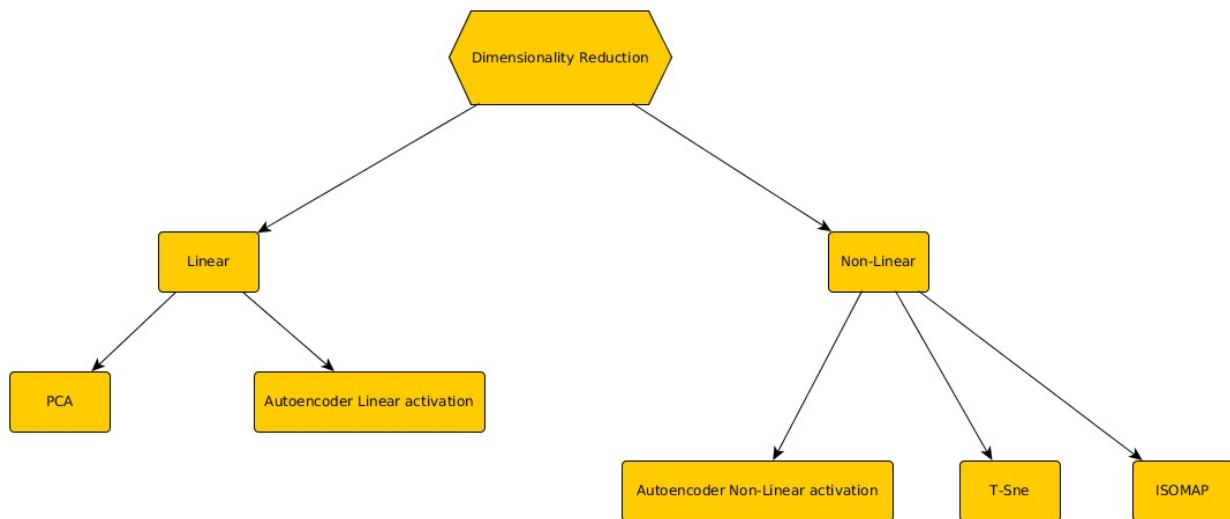
With the increase of more features, the data distribution will not be linear, so simpler linear techniques (ex: PCA) will not be useful for dimensionality reduction. On those cases we need other stuff like T-Sne, Autoencoders, etc..

By the way dimensionality reduction on non-linear manifolds is sometimes called manifold learning.

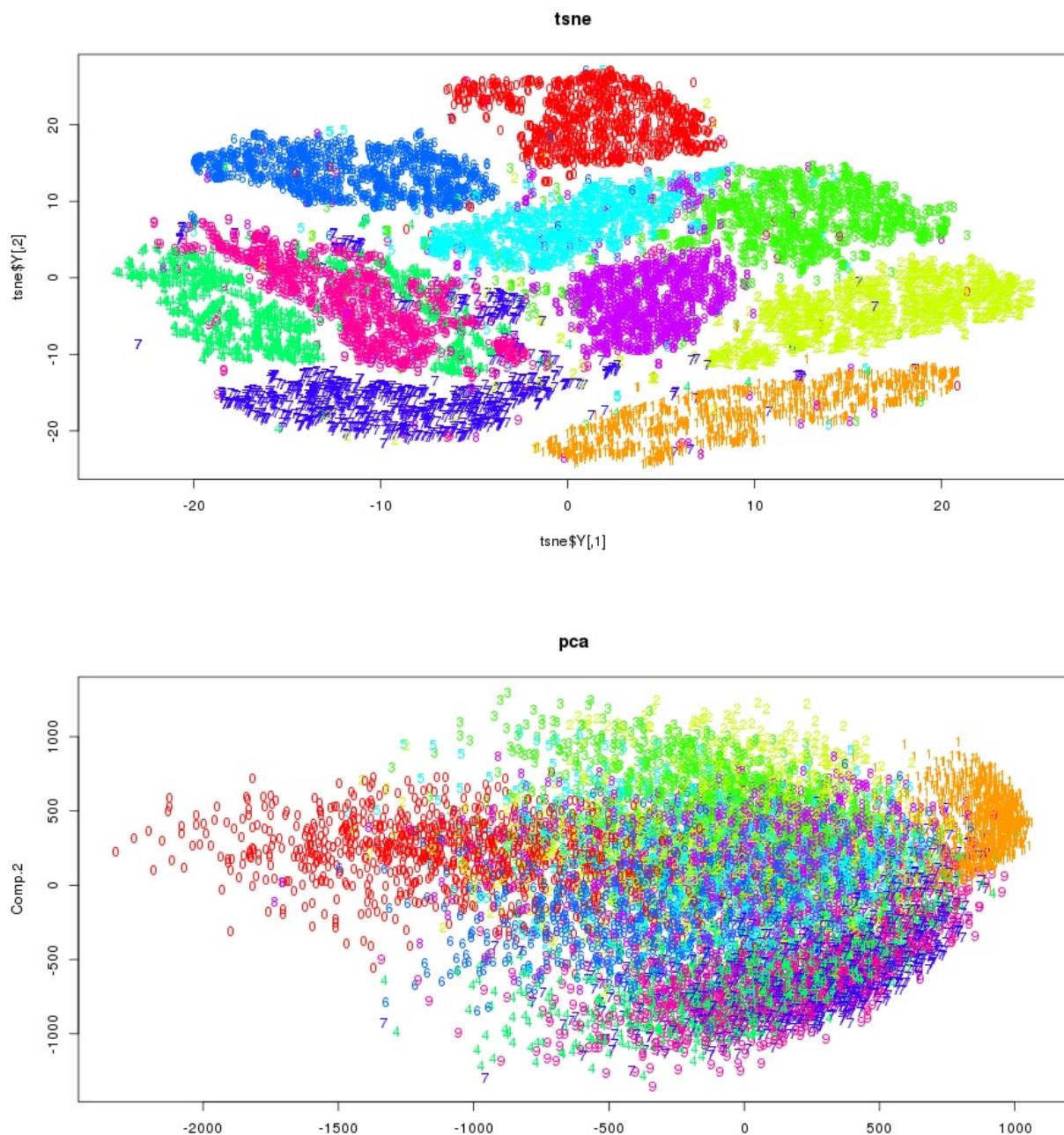


Command Window		Workspace													
New to MATLAB? See resources for Getting Started .															
<pre>>> load swissRoll_dataset.mat >> cmap = jet(numel(swissRoll_labels)); >> subplot(1,2,1); >> gscatter(simple_data(:,1),simple_data(:,2),simple_label); >> subplot(1,2,2); >> scatter3(swissRoll_data(:,1),swissRoll_data(:,2),swissRoll_data(:,3),10,cmap); f1 >> </pre>		<table border="1"> <thead> <tr> <th>Name ↴</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>cmap</td> <td>2048x3 double</td> </tr> <tr> <td>simple_data</td> <td>1600x2 double</td> </tr> <tr> <td>simple_label</td> <td>1600x1 double</td> </tr> <tr> <td>swissRoll_data</td> <td>2048x3 double</td> </tr> <tr> <td>swissRoll_labels</td> <td>2048x1 double</td> </tr> </tbody> </table>		Name ↴	Value	cmap	2048x3 double	simple_data	1600x2 double	simple_label	1600x1 double	swissRoll_data	2048x3 double	swissRoll_labels	2048x1 double
Name ↴	Value														
cmap	2048x3 double														
simple_data	1600x2 double														
simple_label	1600x1 double														
swissRoll_data	2048x3 double														
swissRoll_labels	2048x1 double														

Bellow we have a diagram that guide you depending on the type of problem:



Here is a comparison between the T-SNE method against PCA on MNIST dataset



Principal Component Analysis

Principal Component Analysis

Introduction

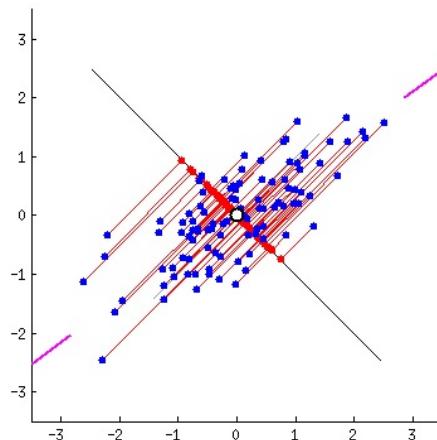
On this chapter we're going to learn about Principal Component Analysis (PCA) which is a tool used to make dimensionality reduction. This is useful because it makes the job of classifiers easier in terms of speed, or to aid data visualization.

So what are principal components then? They're the underlying structure in the data. They are the directions where there is the most variance on your data, the directions where the data is most spread out.

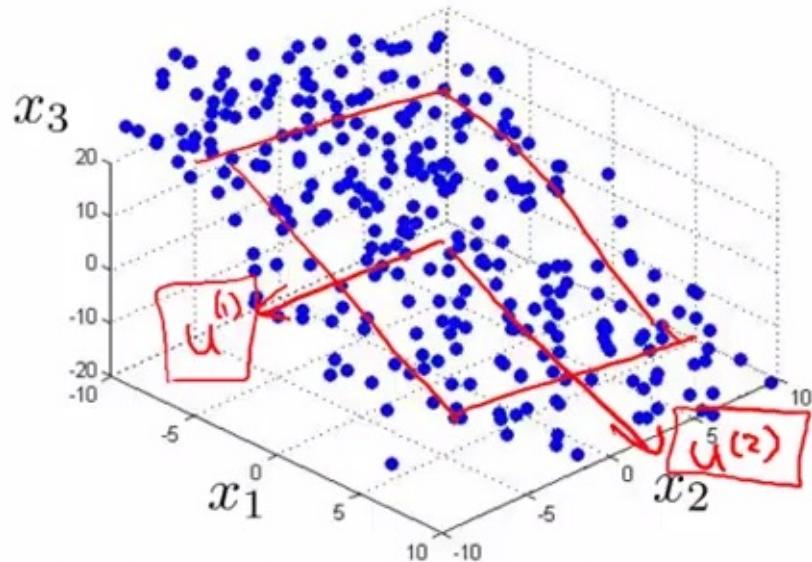
The only limitation of this algorithm is that it works better only when we have a linear manifold.

The PCA algorithm will try to fit a plane that minimizes a projection error (sum of all red-line sizes)

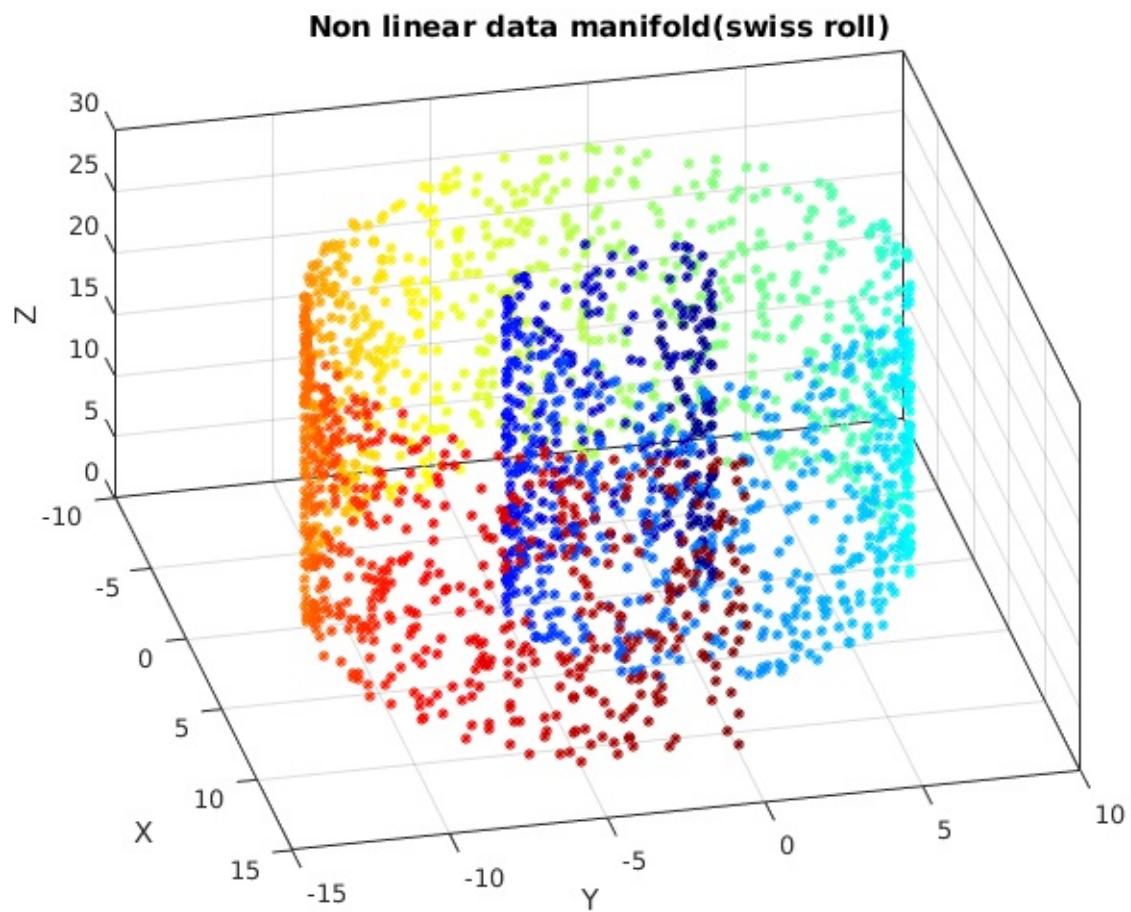
Imagine that the PCA will try to rotate your data looking for an angle where it sees more variances.

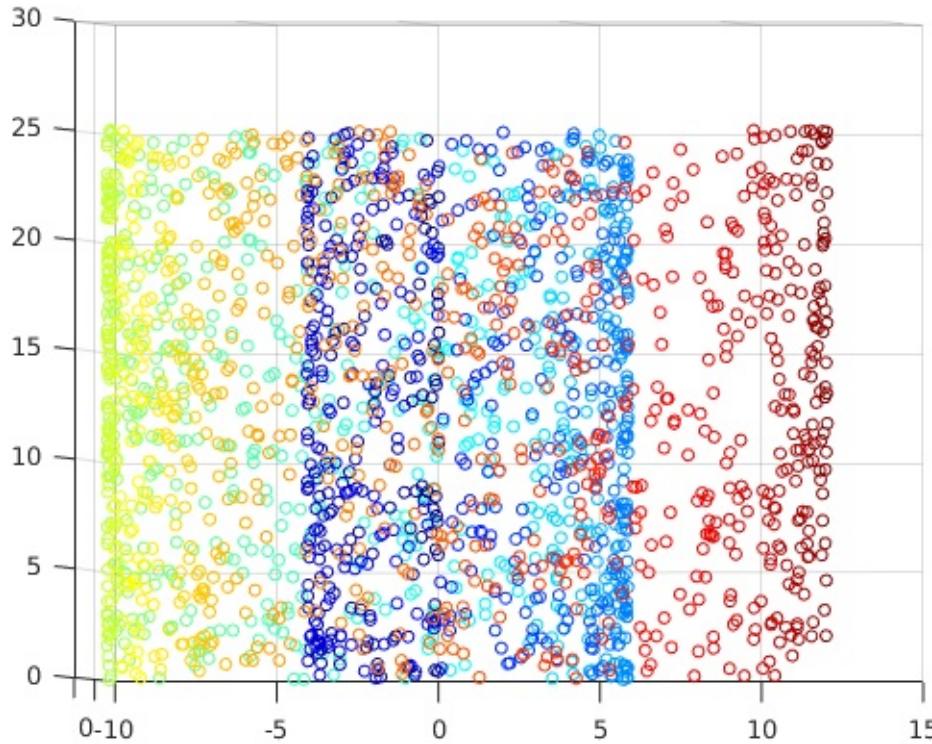


As mentioned before you can use PCA when your data has a linear data manifold.



But for non linear manifolds we're going to have a lot of projection errors.





Calculating PCA

$$X_{prep} = \frac{X - \text{mean}(X)}{\text{std}(X)}$$

1. Preprocess the data:
2. Calculate the covariance matrix: $\sigma = \frac{1}{m} \cdot (X^T \cdot X)$, m is the number of elements, X is a matrix $n \times p$ where n is experiment number and p the features
3. Get the eigenvectors of the covariance matrix $[U, S, V] = \text{svd}(\sigma)$, here the U matrix will be a $n \times n$ matrix where every column of U will be the principal components, if we want to reduce our data from n dimensions to k, we choose k columns from U.

The preprocessing part sometimes includes a division by the standard deviation of each column, but there are cases that this is not needed. (The mean subtraction is more important)

Reducing input data

Now that we calculate our principal components, which are stored on the matrix U, we will reduce our input data $X \in R^n$ from n dimensions to k dimensions $Z \in R^k$. Here k is the number of columns of U. Depending on how you organized the data we can have 2 different formats for Z

$$U_{reduce} = U(:, 1 : k)$$

$$Z = U_{reduce}^T \cdot X_{prep}$$

$$Z = X_{prep} \cdot U_{reduce}$$

Get the data back

To reverse the transformation we do the following:

$$X = [((X_{prep} \cdot U) \cdot U^T) \cdot std(X)] + mean(X)$$

Example in Matlab

To illustrate the whole process we're going to calculate the PCA from an image, and then restore it with less dimensions.

Get some data example

Here our data is a matrix with 15 samples of 3 measurements [15x3]

```
>> X  
  
X =  
  
269.8000    38.9000    50.5000  
272.4000    39.5000    50.0000  
270.0000    38.9000    50.5000  
272.0000    39.3000    50.2000  
269.8000    38.9000    50.5000  
269.8000    38.9000    50.5000  
268.2000    38.6000    50.2000  
268.2000    38.6000    50.8000  
267.0000    38.2000    51.1000  
267.8000    38.4000    51.0000  
273.6000    39.6000    50.0000  
271.2000    39.1000    50.4000  
269.8000    38.9000    50.5000  
270.0000    38.9000    50.5000  
270.0000    38.9000    50.5000  
  
>> [n m] = size(A)  
  
n =  
15  
  
m =  
3
```

Data pre-processing

Now we're going to subtract the mean of each experiment from every column, then divide also each element by the standard deviation of each column.

```
>> X_prep = (X - repmat(mean(X),[n 1])) ./ repmat(std(X),[n 1])  
X_prep =  
-0.0971 -0.0178 0.0636  
1.3591 1.5820 -1.5266  
0.0149 -0.0178 0.0636  
1.1351 1.0487 -0.8905  
-0.0971 -0.0178 0.0636  
-0.0971 -0.0178 0.0636  
-0.9932 -0.8177 -0.8905  
-0.9932 -0.8177 1.0178  
-1.6653 -1.8842 1.9719  
-1.2173 -1.3509 1.6539  
2.0312 1.8486 -1.5266  
0.6870 0.5155 -0.2544  
-0.0971 -0.0178 0.0636  
0.0149 -0.0178 0.0636  
0.0149 -0.0178 0.0636
```

mean and std will work on all columns of X

Calculate the covariance matrix

```
>> sigma = (1/m) * X_prep' * X_prep  
sigma =  
4.6667 4.6207 -3.9356  
4.6207 4.6667 -4.1213  
-3.9356 -4.1213 4.6667
```

Get the principal components

Now we use "svd" to get the principal components, which are the eigen-vectors and eigen-values of the covariance matrix

Principal Component Analysis

```
>> [U,S,~] = svd(sigma)

U =
    -0.5825   -0.4874    0.6505
    -0.5904   -0.2963   -0.7507
     0.5587   -0.8213   -0.1152

S =
    13.1252      0      0
        0    0.8441      0
        0      0    0.0307

>> diag(S)'

ans =
    13.1252    0.8441    0.0307
```

There are different ways to calculate the PCA, for instance matlab gives already a function pca or princomp, which could give different signs on the eigenvectors (U) but they all represent the same components.

```

>> [coeff,score,latent] = pca(X_prep)

coeff =
    0.5825    0.4874   -0.6505
    0.5904    0.2963    0.7507
   -0.5587    0.8213    0.1152

score =
    -0.1026   -0.0003    0.0571
     2.5786   -0.1226    0.1277
    -0.0373    0.0543   -0.0157
     1.7779    0.1326   -0.0536
    -0.1026   -0.0003    0.0571
    -0.1026   -0.0003    0.0571
    -0.5637   -1.4579   -0.0704
    -1.6299    0.1095    0.1495
    -3.1841    0.2496   -0.1041
    -2.4306    0.3647   -0.0319
     3.1275    0.2840   -0.1093
     0.8467    0.2787   -0.0892
    -0.1026   -0.0003    0.0571
    -0.0373    0.0543   -0.0157
    -0.0373    0.0543   -0.0157

latent =
    2.8125
    0.1809
    0.0066

>> [V D] = eig(cov(X_prep))

V =
    0.6505    0.4874   -0.5825
   -0.7507    0.2963   -0.5904
   -0.1152    0.8213    0.5587

D =
    0.0066         0         0
        0    0.1809         0
        0         0    2.8125

```

The one thing that you should pay attention is the order of the input matrix, because some methods to find the PCA, expect that your samples and measurements, are in some pre-defined order.

Recover original data

Now to recover the original data we use all the components, and also reverse the preprocessing.

```
>> X_rec = (((X_prep * U)*U') .* repmat(std(X),[n 1])) + repmat(mean(X),[n 1])
X_rec =
269.8000 38.9000 50.5000
272.4000 39.5000 50.0000
270.0000 38.9000 50.5000
272.0000 39.3000 50.2000
269.8000 38.9000 50.5000
269.8000 38.9000 50.5000
268.2000 38.6000 50.2000
268.2000 38.6000 50.8000
267.0000 38.2000 51.1000
267.8000 38.4000 51.0000
273.6000 39.6000 50.0000
271.2000 39.1000 50.4000
269.8000 38.9000 50.5000
270.0000 38.9000 50.5000
270.0000 38.9000 50.5000
```

Reducing our data

Actually normally we do something before we Now that we have our principal components let's apply for instance k=2

```
>> k=2; U_reduce = U(:,1:k)
```

```
U_reduce =
```

```
-0.5825 -0.4874
-0.5904 -0.2963
0.5587 -0.8213
```

```
>> Z = X_prep * U_reduce
```

```
Z =
```

```
0.1026 0.0003
-2.5786 0.1226
0.0373 -0.0543
-1.7779 -0.1326
0.1026 0.0003
0.1026 0.0003
0.5637 1.4579
1.6299 -0.1095
3.1841 -0.2496
2.4306 -0.3647
-3.1275 -0.2840
-0.8467 -0.2787
0.1026 0.0003
0.0373 -0.0543
0.0373 -0.0543
```

We can use the principal components Z to recreate the data X, but with some loss. The idea is that the data in Z is smaller than X, but with similar variance. On this case we have $X \in R^3$ awe could

reproduce the data X_loss with $Z \in R^{k=2}$, so one dimension less.

```
>> X_loss = ((Z*U_reduce') .* repmat(std(X), [n 1])) + repmat(mean(X), [n 1])

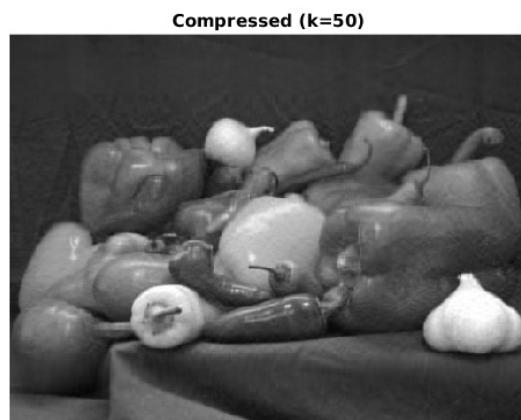
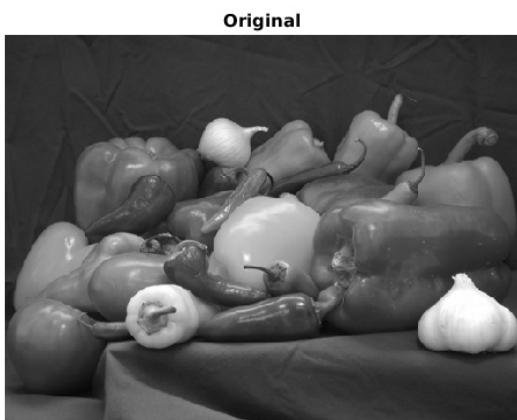
X_loss =

269.8664 38.8839 50.4979
272.5483 39.4640 49.9954
269.9817 38.9044 50.5006
271.9377 39.3151 50.2019
269.8664 38.8839 50.4979
269.8664 38.8839 50.4979
268.1183 38.6198 50.2025
268.3736 38.5579 50.7946
266.8791 38.2293 51.1038
267.7630 38.4090 51.0012
273.4731 39.6308 50.0040
271.0964 39.1251 50.4032
269.8664 38.8839 50.4979
269.9817 38.9044 50.5006
269.9817 38.9044 50.5006
```

Using PCA on images

Before finish the chapter we're going to use PCA on images.

Command Window		Workspace	
New to MATLAB? See resources for Getting Started .	x	Name ↴	Value
>> img = imread('peppers.png');		img	384x512 single
img = rgb2gray(img);		img_loss	384x512 single
img = single(img);		img_prep	384x512 single
img_prep = img - mean2(img);		k	50
[m,n] = size(img_prep);		m	384
sigma = (1/m) * img_prep' * img_prep;		n	512
[U,S,~] = svd(sigma);		S	512x512 single
k=50; U_reduce = U(:,1:k);		sigma	512x512 single
Z = img_prep * U_reduce; % Get 80 principal components		U	512x512 single
img_loss = (Z*U_reduce') + mean2(img);		U_reduce	512x50 single
figure; subplot(1,2,1);		Z	384x50 single
imshow(uint8(img));			
subplot(1,2,2);			
imshow(uint8(img_loss));			



Generative Models

Generative Models

The idea of generative models, is to be able to learn the probability distribution of the training set. By doing this the generative model can create more data from the original data. Imagine as been the perfect dataset augmentation system. So basically it can be used as a unsupervised way to generate samples to train other networks better.

Basically this done by having 2 neural networks playing against each other.

Distributed Learning

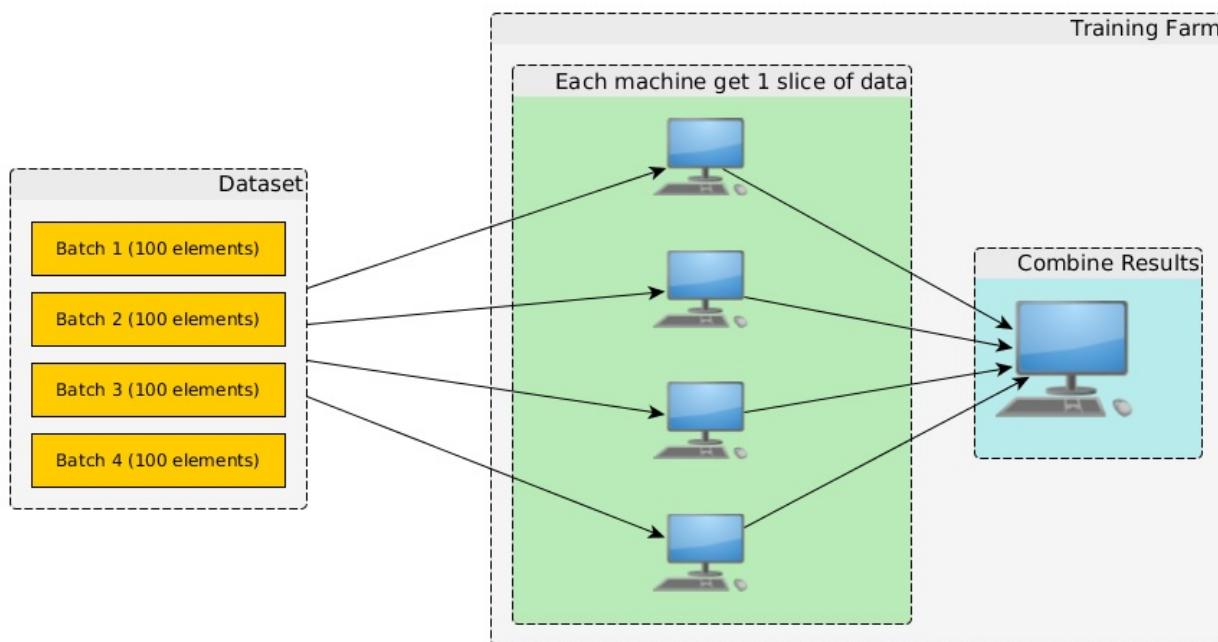
Introduction

Learn how the training of deep models can be distributed across multiple machines.

Map Reduce

Map Reduce can be described on the following steps:

1. Split your training set, in batches (ex: divide by the number of workers on your farm: 4)
2. Give each machine of your farm 1/4th of the data
3. Perform Forward/Backward propagation, on each computer node (All nodes share the same model)
4. Combine results of each machine and perform gradient descent
5. Update model version on all nodes.



Example Linear Regression model

Consider the batch gradient descent formula, which is the gradient descent applied on all training set:

$$\theta_j := \theta_j - \alpha \frac{1}{400} \sum_{i=1}^{400} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

Each machine will deal with 100 elements (After splitting the dataset), calculating $temp_j^{1..4}$, then:

$$temp_j^1 = \sum_{i=1}^{100} (h_\theta(x^{(i)} - y^{(i)}) x_j^{(i)}$$

$$temp_j^2 = \sum_{i=101}^{200} (h_\theta(x^{(i)} - y^{(i)}) x_j^{(i)}$$

$$temp_j^3 = \sum_{i=201}^{300} (h_\theta(x^{(i)} - y^{(i)}) x_j^{(i)}$$

$$temp_j^4 = \sum_{i=301}^{400} (h_\theta(x^{(i)} - y^{(i)}) x_j^{(i)}$$

Each machine is calculating the back-propagation and error for its own split of data. Remember that all machines have the same copy of the model.

After each machine calculated their respective $temp_j^{\text{machine}}$. Another machine will combine those gradients, calculate the new weights and update the model in all machines.

$$\theta_j := \theta_j - \alpha \frac{1}{400} (temp_j^1 + temp_j^2 + temp_j^3 + temp_j^4)$$

The whole point of this procedure is to check if we can combine the calculations of all nodes and still make sense, in terms of the final calculation.

Who use this approach

- Caffe
- Torch (Parallel layer)

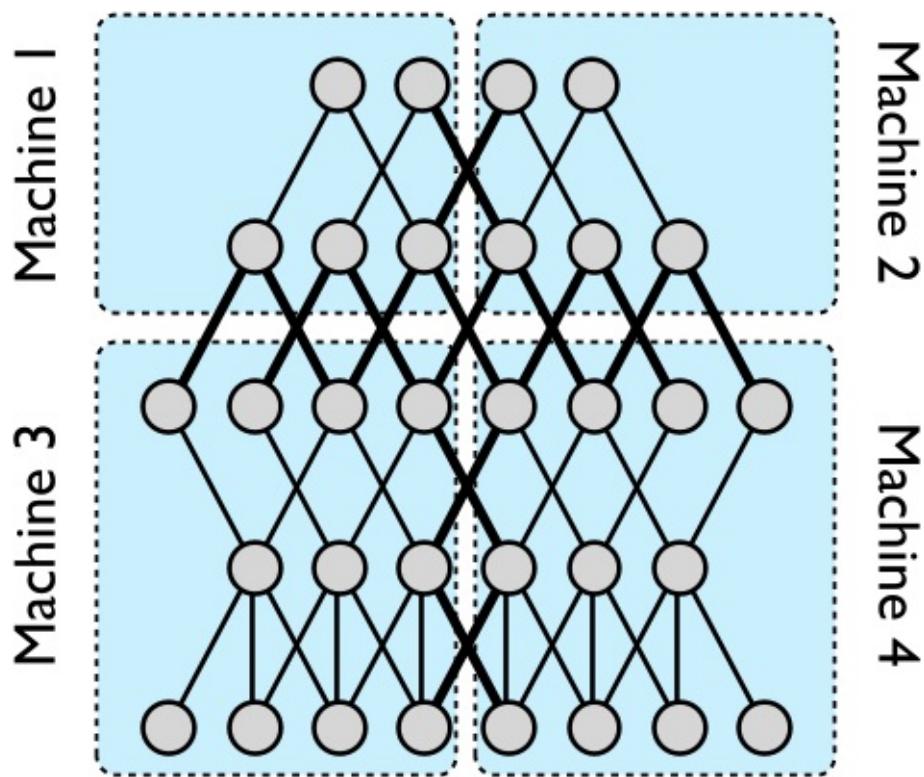
Problems

This approach has some problems:

- The complete model must fit on every machine
- If the model is too big it will take time to update all machines with the same model

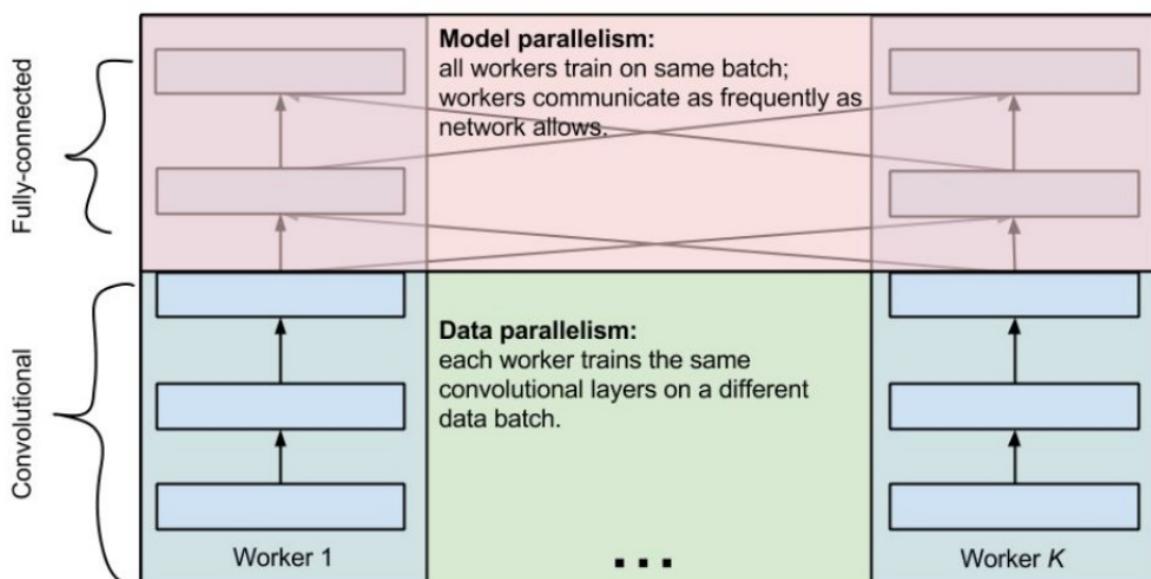
Split weights

Another approach was used on Google DistBelief project where they use a normal neural network model with weights separated between multiple machines.

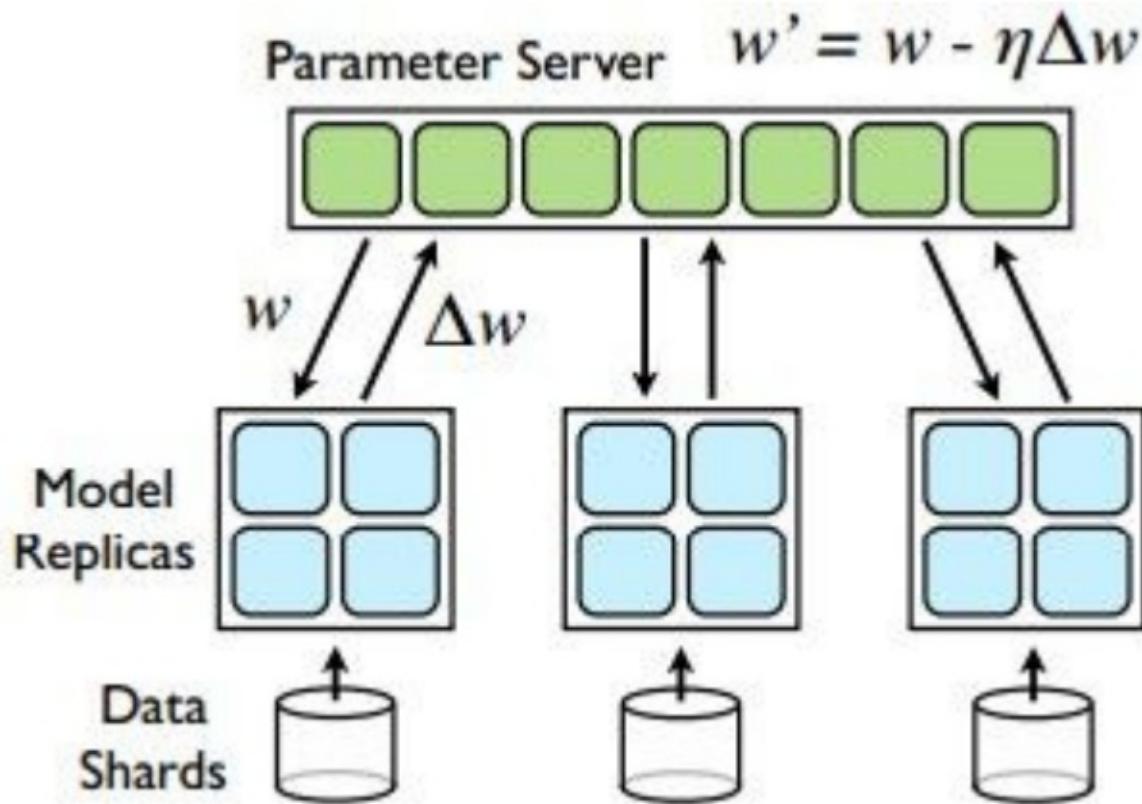


On this approach only the weights (thick edges) that cross machines need to be synchronized between the workers. This technique could only be used on fully connected layers.

If you mix both techniques (reference on Alexnet paper), you do this share fully connected processing (Just a matrix multiplication), then when you need to the the convolution part, each convolution layer get one part of the batch.



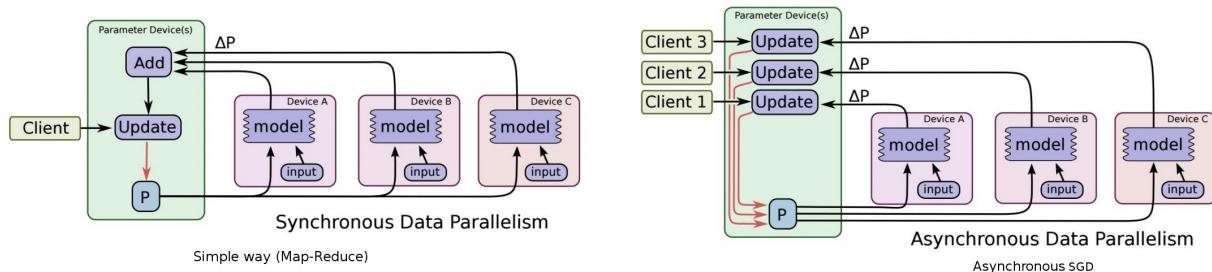
Google approach (old)



Here each model replica is trained independently with pieces of data and a parameter server that synchronize the parameters between the workers.

Google new approach

Now google offer on Tensorflow some automation on choosing which strategy to follow depending on your work.



Asynchronous Stochastic Gradient Descent

Methodology for usage

Introduction

This chapter will give a [recipe](#) on how to tackle Machine learning problems

3 Step process

1. Define what you need to do and how to measure(metrics) how well/bad you are going.
2. Start with a very simple model (Few layers)
3. Add more complexity when needed.

Define goals

Normally by being slightly better than human performance(in terms of accuracy or prediction time), is already enough for a good product.

Metrics

- Accuracy: % of correct examples
- Coverage: Number of processed examples per unit of time
- Precision: Number of detections that are correct
- Error: Amount of error

Start with simplest model

Look if available for the state-of-the-art method for solving that problem. If not available use the following recipe.

1. Lots of noise and now much structure(ex: House price from features like number of rooms,kitchen size, etc...): Don't use deep learning
2. Few noise but lot's of structure (ex: Images, video, text): Use deep learning

Examples for Non-deep methods:

- Logistic Regression
- SVM
- Boosted decision trees (Previous favorite "default" algorithm), used a lot in robotics

What kind of deep

- Few structure: Use only Fully-Connected layers 2-3 layers (Relu, Dropout, SGD+Momentum). Need at least few thousand examples per class.
- Spatial structure: Use CONV layers (Inception/Residual, Relu, Dropout, Batch-Norm, SGD+Momentum).
- Sequencial structure (text, market movement): Use Recurrent networks (LSTM, SGD, Gradient clipping).

When using Residual/Inception networks start with the shallowest example possible.

Solving High train errors

Do the following actions on this order:

1. Inspect for defects on the data. (Need human intervention)
2. Check for software bugs on your library. (Use gradient check, it's probably a backprop error)
3. Tune learning rate (Make it smaller)
4. Make network deeper. (You should start with a shallow network on the beginning)

Solving High test errors

Do the following actions on this order:

1. Do more data augmentation (Also try generative models to create more data)
2. Add dropout and batch-norm
3. Get more data (More data has more influence on Accuracy than anything else)

Some trends

Following the late 2016 Andrew Ng [lecture](#) these are the topics that we need to pay attention.

1. Scalability

Have a computing system that scale well for more data and more model complexity.

2. Team

Have on your team divided with AI people and HPC (Cuda, OpenCl, etc...).

3. Data first

Data is more important than your model, always try to get more quality data before trying to change your model.

4. Data Augmentation

Use normal data augmentation techniques plus Generative models(Unsupervised).

5. Make sure that Validation Set and Test set come from same distribution

This will avoid having a test or validation set that does not tell the reality. Also helps to check if your training is valid.

6. Have Human level performance metric

Have a team of experts to compare with your current system performance. Also it drives decisions between getting more data, or making model more complex.

7. Data server

Have a unified data-warehouse. All team must have access to data, with SSD quality access speed.

8. Using Games

Using games are cool to help augment datasets, but attention because games does not have the same variants of the same class as real life. For example GTA does not have enough car models compared to real life.

9. Ensembles always help

Training separately different networks and averaging their end results always gives some extra 2% accuracy. (Imagenet 2016 best results were simple ensambles)

10. What to do if you have more than 1000 classes

Use hierarchical Softmax to increase performance.

11. How many samples per class do we need to have good result

If training from scratch, use the same number of parameters. For example Model has 1000 parameters, so use 1000 samples per class.

If doing transfer learning is much less (Much is not defined yet, more is better).

Artificial Intelligence

Introduction

In computer science, an ideal "intelligent" machine is a flexible rational agent that perceives its environment and takes actions that maximize its chance of success at some goal.

What means rational

Here we will try to explain what means acting rational:

- Maximally achieve pre-defined goals. (Expected utility)
- Goals expressed in terms of utility (non dimensional scalar value that represent "happiness")
- Be rational means maximize your expected utility

Have a rational (for us intelligent) decision, is only related to the quality of the decision(utility), not the process that lead to that decision, for instance:

- You gave the right answer because you brute-force all possible outcomes
- You gave the right answer because you used a fancy state of the art method

Basically you don't care about the method just the decision.

As the AI field evolve what was considered intelligent, is not considered anymore after someone discover how to do it.

A system is rational if he does the right thing with the information available.

Why we only care to be rational

Basically we don't understand how our brain work, even with the current advances on neuro-science, and the advances on computation power. We don't know how the brain works and take decisions. The only cool thing that we know about the brain, is that to do good decisions, **we need memory and simulation**.

Central AI problems:

1. Reasoning
2. Knowledge
3. Planning: Make prediction about their actions, and choose the one that
4. Learning
5. Perception
6. Natural Language Processing (NLP)

Agents

It's a system (ie: software program) which observes the world through sensors and acts upon an environment using actuators. It directs its activity towards achieving goals. Intelligent agents may

also learn or use knowledge to achieve their goals.

Type of agents

There are a lot of agents types, but here we're going to separate them in 2 classes

- Reflex Agents: Don't care about future effects of its actions (Just use a if-table)
- Planning Agents: Simulate actions consequences before committing to them, using a model of the world.

Both reflex and planning agents can be rational, again we just care about the result action, if the action maximize its expected utility, then it is rational.

we need to identify which type of agent is needed to have a rational behavior.

A reflex or planning agent can be sub-optimal, but normally planning is a good idea to follow.

On this book we're going to see a lot of tools used to address planning. For instance searching is a kind of tool for planning.

Search problem

A search problem finds a solution which is a sequence of actions (a plan) that transform the start state to the goal state.

Types of search:

- Uninformed Search: Keep searching everywhere until a solution is found
- Informed Search: Has kind of "information" saying if we're close or not to the solution (Heuristic)

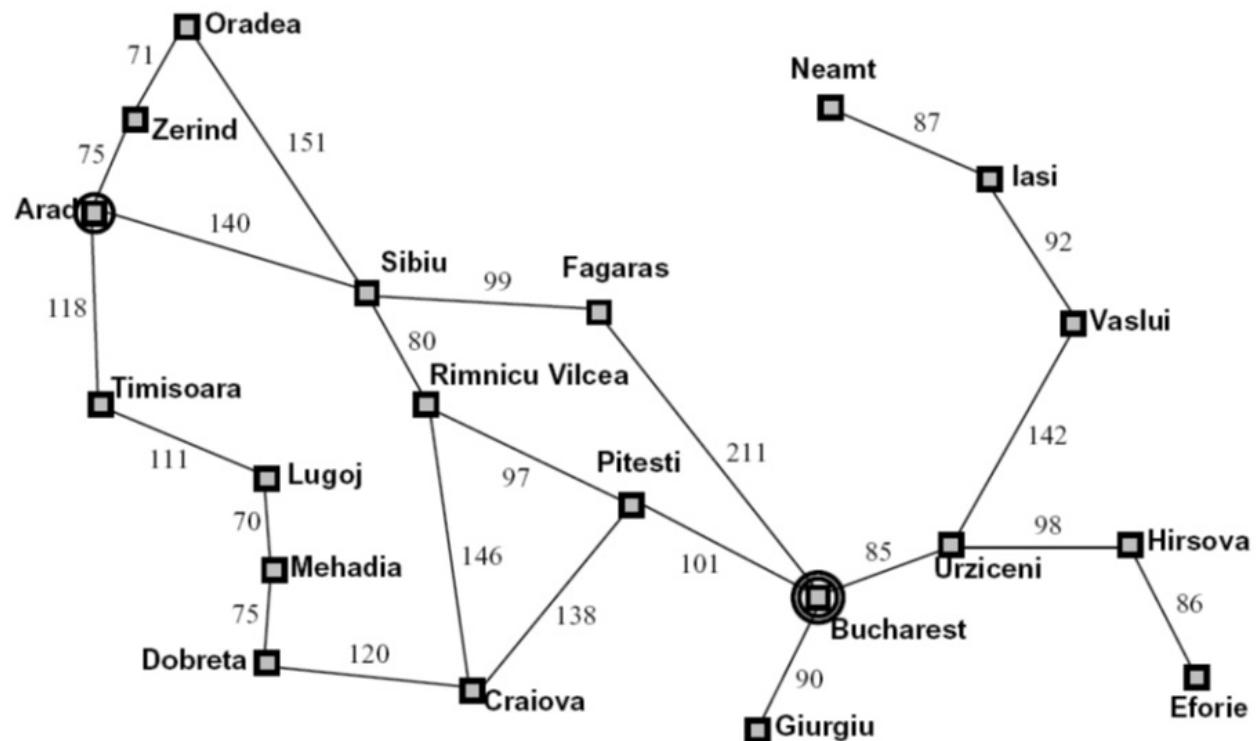
A search problem consist on the following things:

1. A State space: Has the states needed to do planning
2. A successor function: For any state x , return a set of states reachable from x with one action
3. A start state and goal test: Gives initial point and how to check when planning is over

Example our objective is to travel from Arad, to Bucharest



Above you have the world state, we don't need so many details, we only need the cities, how they connect and the distances.

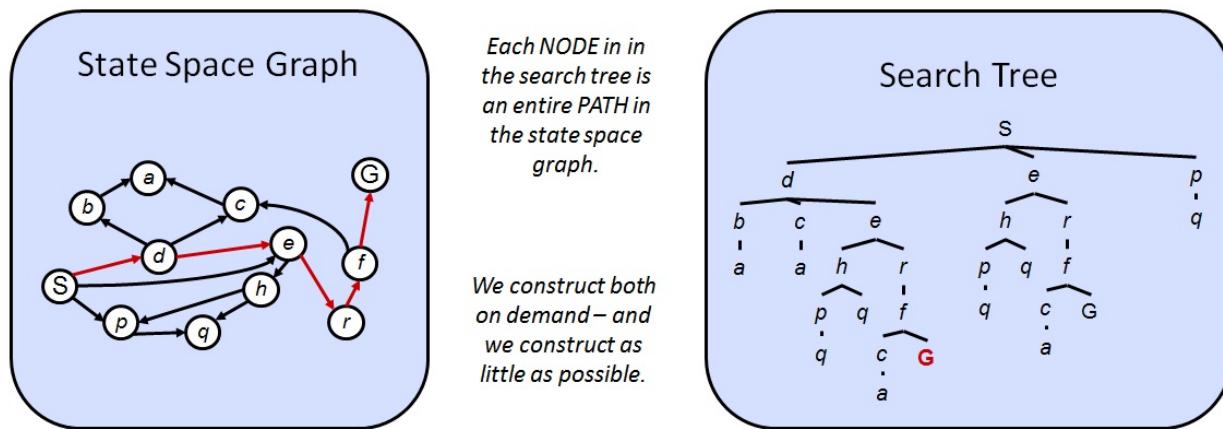


On this search problem we detect the following properties:

- State space: The cities (The only variable pertinent to this search problem)
- Start state: Arad

- Successor Function: Go to adjacent city with cost as distance.

Consider the map above as a graph, it contains nodes, that don't repeat and how they connect along with the costs of its connection.

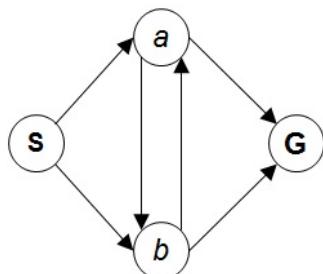


One way to do planning is convert the state space graph to a search tree, then use some algorithms that search for a goal state on the tree. Here we observe the following things:

- The start state will be the tree root node
- Node children represent the possible outcomes for each state.

The problem is that both Search Trees, or State space graphs can be too big to fit inside the computer, for instance the following state space graph has an infinite search tree.

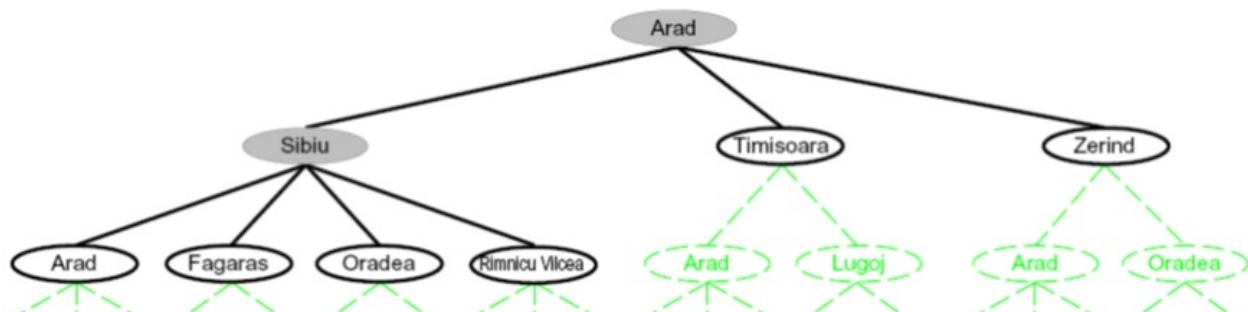
Consider this 4-state graph:



How big is its search tree (from S)?



What to do on those cases, basically you don't keep in memory all the possible solutions of the tree or graph, you navigate the tree for a finite amount of depth.



For instance, look to the state graph of the Romania, we start on Arad (Start state)

- Arad has 3 possible child nodes, Sibiu, Timisoara and Zerind
- We choose the leftmost child node Sibiu
- Then we choose the leftmost child node Arad, which is bad, so we try Fagaras, then Oradea, etc...

The problem is that if one of the tree branches is infinite, or is too big and has no solution we will keep looking on its branch until the end.

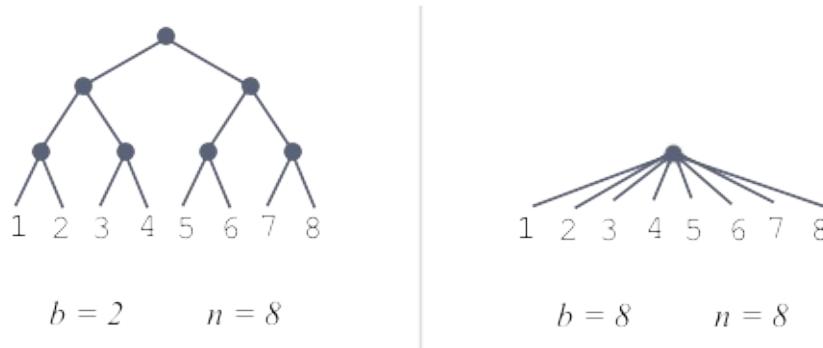
At the point that we choose Sibiu on the first step, we need to keep the other possible nodes (Timisoara, and Zerind) this is called the tree fringe.

Important ideas to keep in mind on tree search:

- First of all tree search is a mechanism used for planning
- Planning means that you have a model of the world, if the model is bad your solution will also be bad
- Fringe: Or cache of other possible solutions
- How to explore the current branch

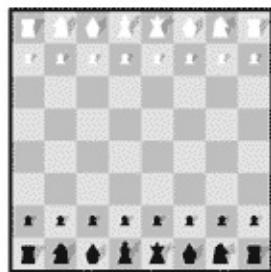
Branching Factor

Branching factor is the number children on each node on a tree.



Old problems like tic-tac-toe or other simple problems can be solved with a search tree or some sort of optimized tree algorithm. But games like chess or go has a huge branching factor, so you could not process them in real-time.

On the animation below we compare the chess and go branching factor.



Next Chapter

On the next chapter we will explore more about trees.

OpenAI Gym

OpenAI Gym

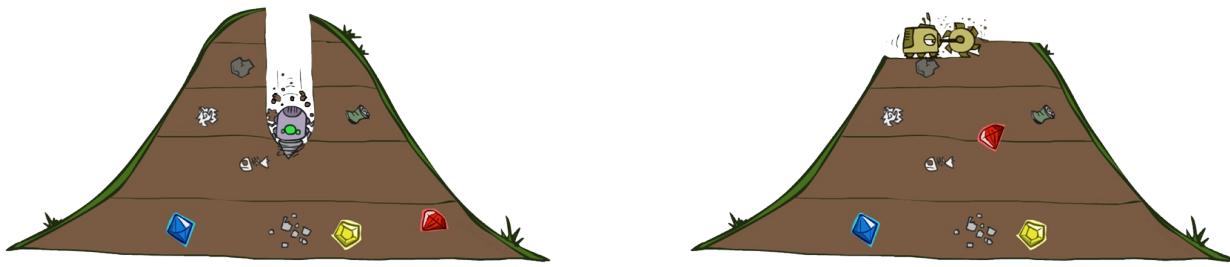
Introduction

Tree Search

Tree Search

Introduction

On this chapter we will learn about some ways to do tree search. Just to remember from the introduction tree search is one of the mechanisms to do planning. Planning means that the agent will simulate possible actions on a model of the world, and choose the one that will maximize it's utility.



On this chapter we will learn the following techniques to tree searching

- Depth first search
- Breadth-Fist search
- Uniform Cost search
- Greedy search
- A-star search A*

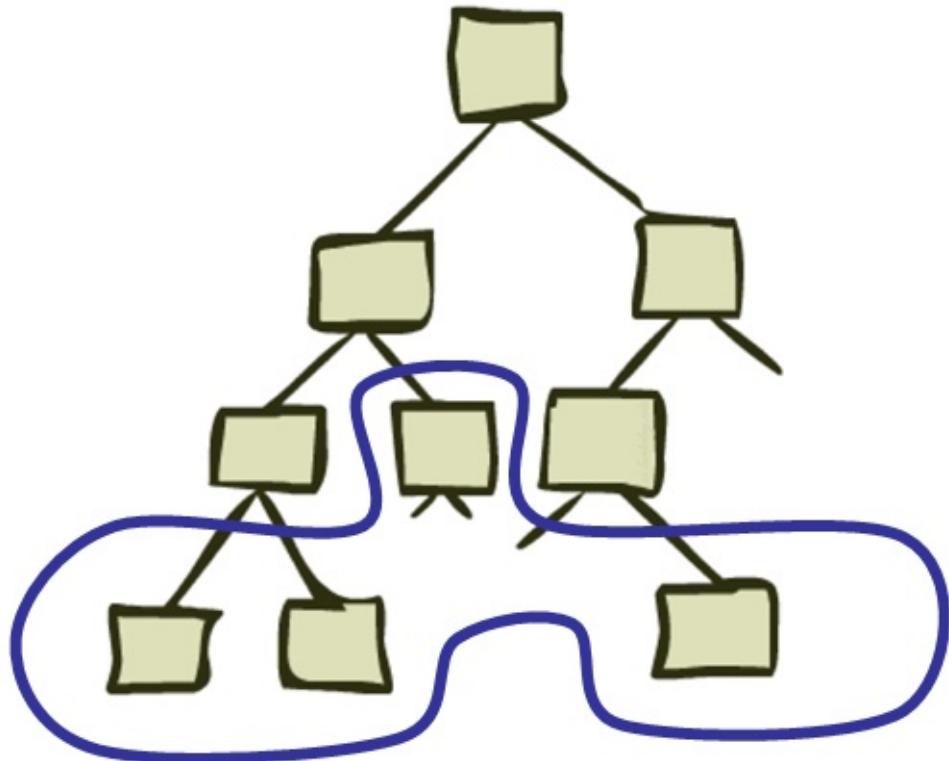
As mentioned before we cannot hold the whole tree on memory, so what we do is to expand the tree only when you needed it and you keep track of the other options that you did not explored yet.

```

function TREE-SEARCH(problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
  end

```

To those parts that are still on memory but not expanded yet we call fringe.



Depth first search

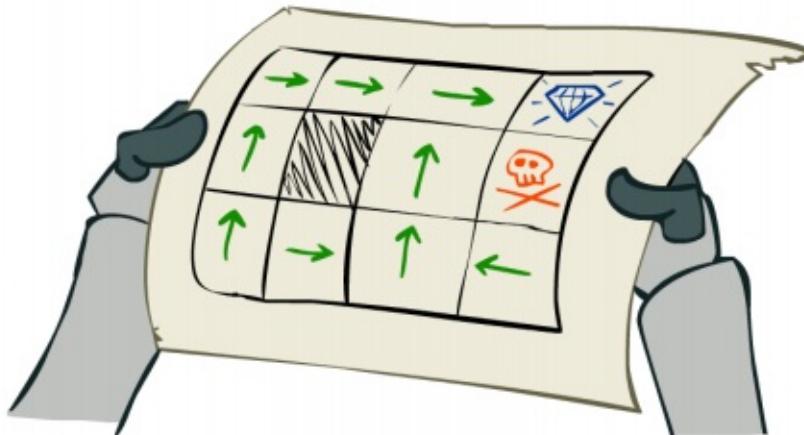
Markov Decision process

Markov Decision process

Introduction

Markov Decision process(MDP) is a framework used to help to make decisions on a stochastic environment. Our goal is to find a policy, which is a map that gives us all optimal actions on each state on our environment.

MDP is somehow more powerful than simple planning, because your policy will allow you to do optimal actions even if something went wrong along the way. Simple planning just follow the plan after you find the best strategy.



What is a State

Consider state as a summary (then called state-space) of all information needed to determine what happens next. There are 2 types of state space:

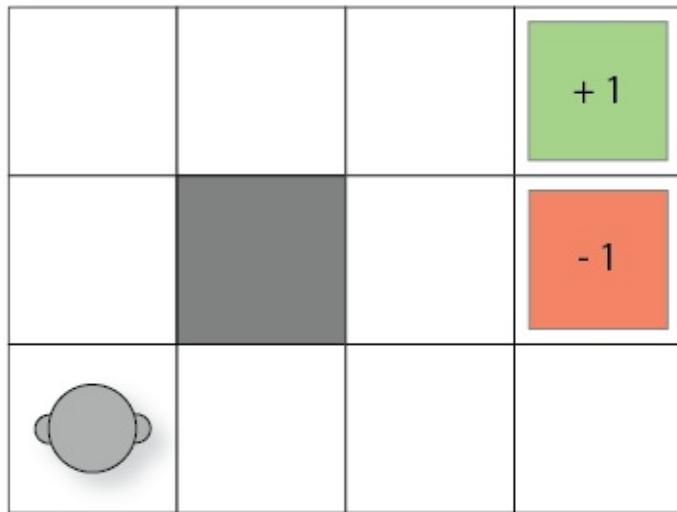
- World-State: Normally huge, and not available to the agent.
- Agent-State: Smaller, have all variables needed to make a decision related to the agent expected utility.

Markovian Property

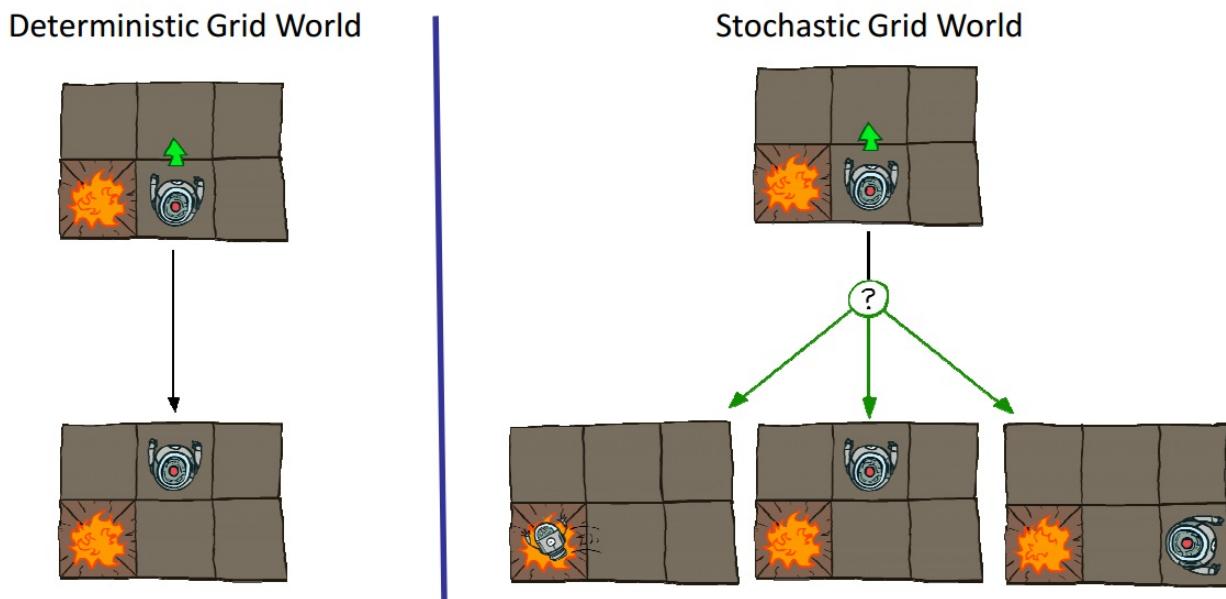
Basically you don't need past states to do a optimal decision, all you need is the current state s . This is because you could encode on your current state everything you need from the past to do a good decision. Still history matters...

Environment

To simplify our universe imagine the grid world, here your agent objective is to arrive on the green block, and avoid the red block. Your available actions are: $\uparrow, \downarrow, \leftarrow, \rightarrow$



The problem is that we don't live on a perfect deterministic world, so our actions could have different outcomes:



For instance when we choose the up action we have 80% probability of actually going up, and 10% of going left or right. Also if you choose to go left or right you have 80% chance of going left and 10% going up or down.

Here are the most important parts:

- States: A set of possible states S
- Model: $T(s, a, s') \therefore P(s'|s, a)$ Probability to go to state s' when you do the action a while you were on state s , is also called transition model.
- Action: $A(s)$, things that you can do on a particular state s
- Reward: $R(s)$, scalar value that you get for been on a state.

- Policy: $\Pi(s) \rightarrow a$, our goal, is a map that tells the optimal action for every state
- Optimal policy: $\Pi^*(s) \rightarrow a$, is a policy that maximize your expected reward $R(s)$

In reinforcement learning we're going to learn a optimal policy by trial and error.

Reinforcement Learning

Reinforcement Learning

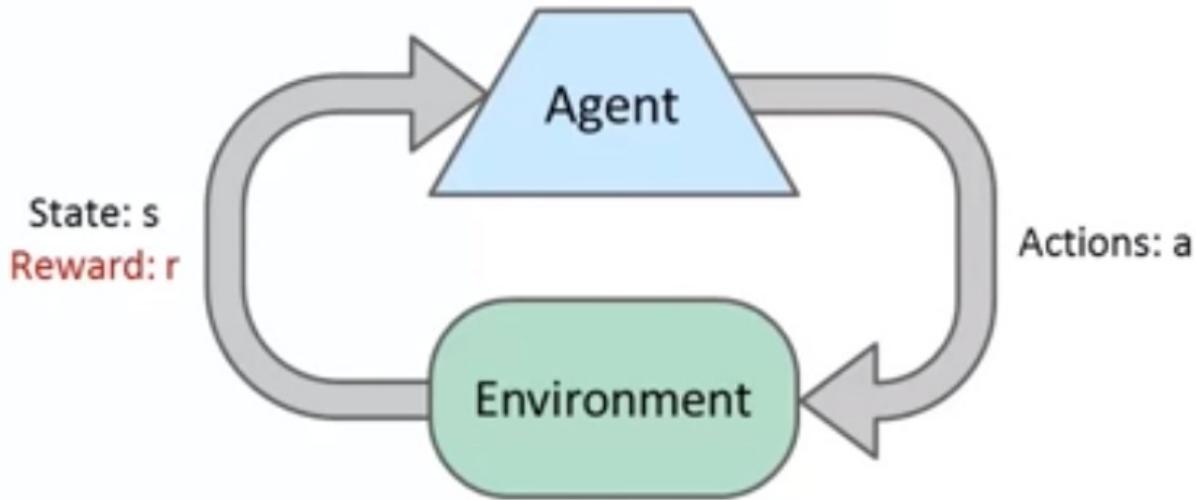
Introduction

On this chapter we will learn the basics for Reinforcement learning (RL). Basically an RL agent differently of solving a MDP where a graph is given, does not know anything about the environment, it learns what to do by exploring the environment. It uses actions, and receive states and rewards. You can only change your environment through actions.

One of the big difficulties of RL is that some actions take time to create a reward, and learning this dynamics can be challenging. Also the reward received by the environment is not related to the last action, but some action on the past.

Some concepts:

- Agents take actions in an environment and receive states and rewards
- Goal is to find a policy $\Pi(s) = \text{action}$ that maximize its utility function $U(s)$
- Inspired by research on psychology and animal learning



Here we don't know which actions will produce rewards, also we don't know when an action will produce rewards, sometimes you do an action that will take time to produce rewards.

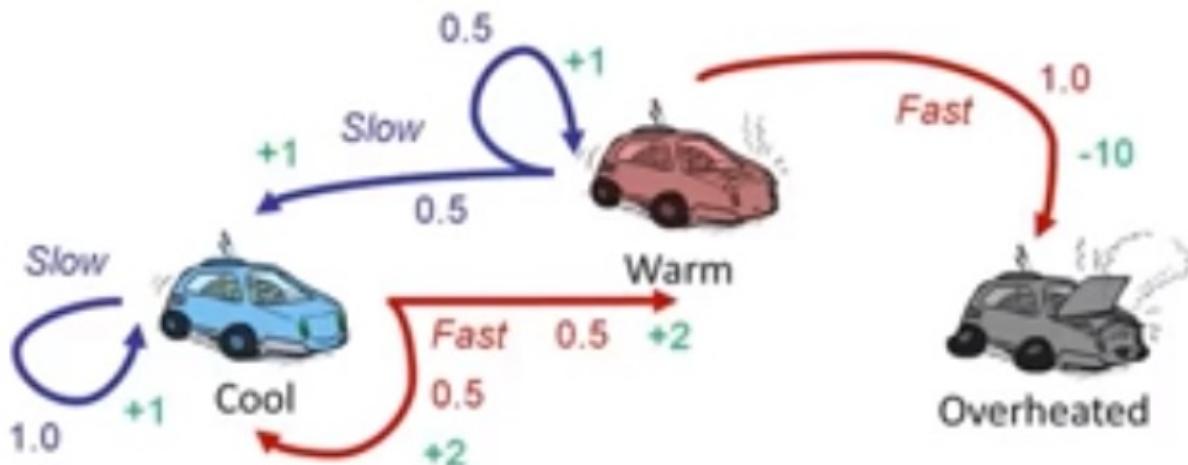
Basically all is learned with interactions with the environment.

Reinforcement learning components:

- Agent: Our robot
- Environment: The game, or where the agent lives.
- A set of states $s \in S$
- Policy: Map between state to actions

- Reward Function $R(s, a, s')$: Gives immediate reward for each state
- Value Function: Gives the total amount of reward the agent can expect from a particular state to all possible states from that state. With the value function you can find a policy.
- Model $T(s, a, s')$ (Optional): Used to do planning, instead of simple trial-and-error approach common to Reinforcement learning. Here s' means the possible state after we do an action a on the state s

In our minds we will still think that there is a Markov decision process (MDP), which have:



We're looking for a policy $\pi(s)$, which means a map that give us optimal actions for every state

The only problem is that we don't have now explicitly $T(s, a, s')$ or $R(s, a, s')$, so we don't know which states are good or what the actions do. The only way to learn those things is to try them out and learn from our samples.

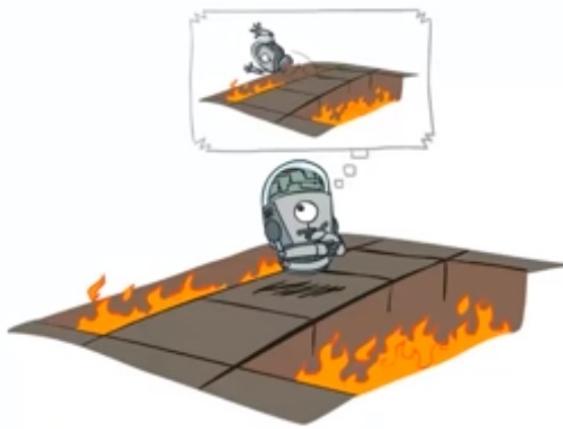
On Reinforcement learning we know that we can move fast or slow (Actions) and if we're cool, warm or overheated (states). But we don't know what our actions do in terms of how they change states.



Offline (MDPs) vs Online (RL)

Another difference is that while a normal MDP planning agent, find the optimal solution, by means of searching and simulation (Planning). A RL agent learns from trial and error, so it will do something

bad before knowing that it should not do. Also to learn that something is real bad or good, the agent will repeat that a lot of times.



Offline Solution



Online Learning

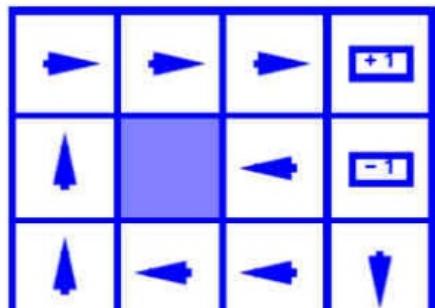
How it works

We're going to learn an approximation of what the actions do and what rewards we get by experience. For instance we could randomly do actions

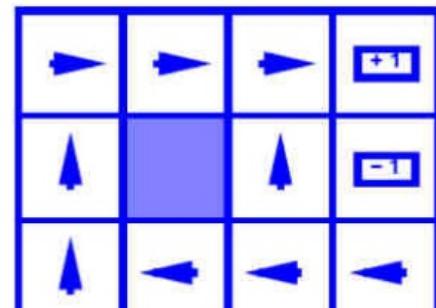
Late Reward

We force the MDP to have good rewards as soon as possible by giving some discount γ reward over time. Basically you modulate how in a rush your agent by giving more negative values to $R(s)$ over time.

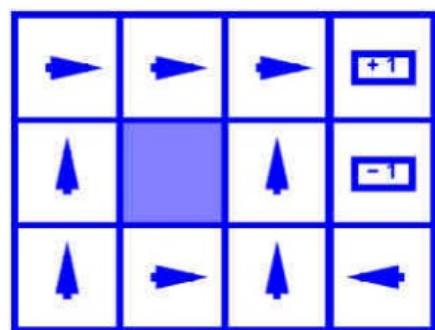
Also you can change the behavior of your agent by giving the amount of time that your agent have.



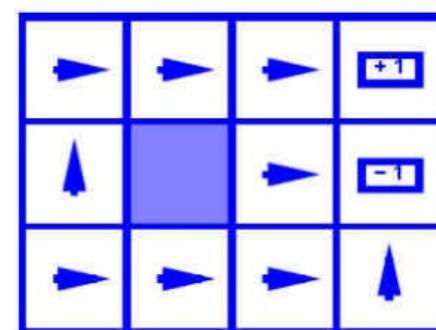
$$R(s) = -0.01$$



$$R(s) = -0.03$$



$$R(s) = -0.4$$



$$R(s) = -2.0$$

Exploration and Exploitation

One of the problems of Reinforcement learning is the exploration vs exploitation dilemma.

- Exploitation: Make the best decision with the knowledge that we already know (ex: Choose the action with biggest Q-value)
- Exploration: Gather more information by doing different (stochastic) actions from known states.

Example: Restaurant

- Exploitation: Go to favorite restaurant, when you are hungry (gives known reward)
- Exploration: Try new restaurant (Could give more reward)

One technique to keep always exploring a bit is the usage of $\epsilon - \text{greedy}$ exploration where before we take an action we add some random factor.

Q_Learning_Simple

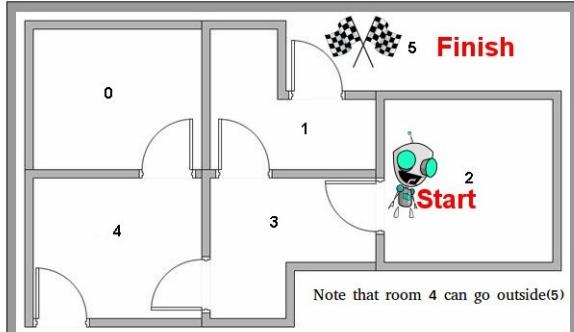
Q_Learning_Simple

Introduction

Q_Learning is a model free reinforcement learning technique. Here we are interested on finding through experiences with the environment the action-value function Q. When the Q function is found we can achieve optimal policy just by selecting the action that gives the biggest expected utility(reward).

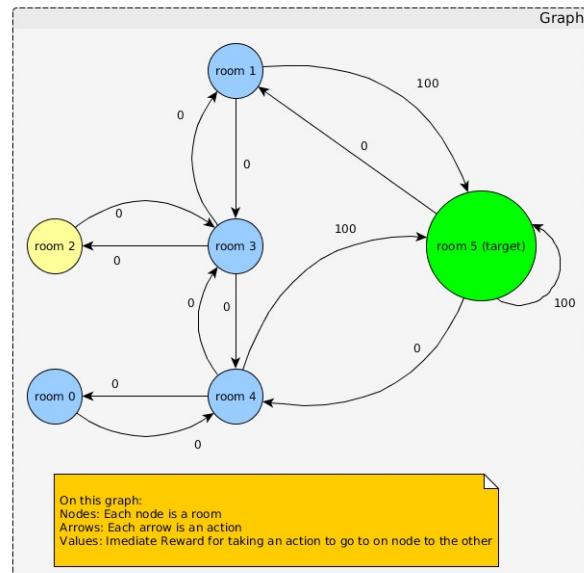
Simple Example

Consider a robot that need to learn how to leave a house with the best path possible, on this example we have a house with 5 rooms, and one "exit" room.



Our objective:
We want an agent that is able to find the goal state(5) from an initial state (ie: 2)

Each time that the robot go to a room that is not the goal state it receives a 0 reward, if we go to the goal state the reward will be 100



On this graph:
Nodes: Each node is a room
Arrows: Each arrow is an action
Values: Immediate Reward for taking an action to go to node to the other

Above we show the house, plant and also a graph representing it. On this graph all rooms are nodes, and the arrows the actions that can be taken on each node. The arrow values, are the immediate rewards that the agent receive by taking some action on a specific room. We choose our reinforcement learning environment to give 0 reward for all rooms that are not the exit room. On our target room we give a 100 reward.

To summarize

- Actions: 0,1,2,3,4,5
- States: 0,1,2,3,4,5
- Rewards: 0,100
- Goal state: 5

We can represent all this mechanics on a reward table.

State	Action					
	0	1	2	3	4	5
0	-1	-1	-1	-1	0	-1
1	-1	-1	-1	0	-1	100
2	-1	-1	-1	0	-1	-1
3	-1	0	0	-1	0	-1
4	0	-1	-1	0	-1	100
5	-1	0	-1	-1	0	100

On this table the rows represent the rooms, and the columns the actions. The values on this matrix represent the rewards, the value (-1) indicate that some specific action is not available.

For example looking the row 4 on this matrix gives the following information:

- Available actions: Go to room 0,3,5
- Possible rewards from room 4: 0 (room 4 to 0),0 (room 4 to 3),100 (room 4 to 5)

The whole point of Q learning is that the matrix R is available only to the environment, the agent need to learn R by himself through experience.

What the agent will have is a Q matrix that encodes, the state,action,rewards, but is initialized with zero, and through experience becomes like the matrix R.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0

As seen on previous chapters, after we have found the Q matrix, we have an optimum policy, and we're done. Basically we just need to use the Q table to choose the action that gives best expected reward. You can also imagine the Q table as the memory of what the agent learned so far through it's experiences.

Algorithm explanation

Just as a quick reminder let's describe the steps on the Q-Learning algorithm

1. Initialize the Q matrix with zeros
2. Select a random initial state
3. For each episode (set of actions that starts on the initial state and ends on the goal state)
 - While state is not goal state
 1. Select a random possible action for the current state
 2. Using this possible action consider going to this next state
 3. Get maximum Q value for this next state (All actions from this next state)
 4. $Q(state, action) = R(state, action) + \text{Gamma} * \text{Max}[Q(nextState, allActions_{nextState})]$

After we have a good Q table we just need to follow it:

1. Set current state = initial state.
2. From current state, find the action with the highest Q value
3. Set current state = next state(state from action chosen on 2).
4. Repeat Steps 2 and 3 until current state = goal state.

Q-Learning on manual

Let's exercise what we learned so far by doing some episodes by hand. Consider $\gamma = 0.8$ and our initial node(state) to be "room 1"

As we don't have any prior experience we start our Q matrix with zeros

$$Q = \begin{matrix} & \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \left[\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{matrix} \right] \end{matrix}$$

Now take a look on our reward table (Part of the environment)

$$R = \begin{matrix} & \begin{matrix} & \text{Action} \end{matrix} \\ \begin{matrix} \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \left[\begin{matrix} & 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & -1 & -1 & -1 & -1 & 0 & -1 \\ 1 & -1 & -1 & -1 & 0 & -1 & 100 \\ 2 & -1 & -1 & -1 & 0 & -1 & -1 \\ 3 & -1 & 0 & 0 & -1 & 0 & -1 \\ 4 & 0 & -1 & -1 & 0 & -1 & 100 \\ 5 & -1 & 0 & -1 & -1 & 0 & 100 \end{matrix} \right] \end{matrix}$$

Episode 1

As we start from state "room 1" (second row) there are only the actions 3(reward 0) or 5(reward 100) do be done, imagine that we choose randomly the action 5.

State	Action					
	0	1	2	3	4	5
0	-1	-1	-1	-1	0	-1
1	-1	-1	-1	0	-1	100
2	-1	-1	-1	0	-1	-1
3	-1	0	0	-1	0	-1
4	0	-1	-1	0	-1	100
5	-1	0	-1	-1	0	100

On this new (next state 5) there are 3 possible actions: 1 , 4 or 5, with their rewards 0,0,100. Basically is all positive values from row "5", and we're just interested on the one with biggest value. We need to select the biggest Q value with those possible actions by selecting $Q(5,1)$, $Q(5,4)$, $Q(5,5)$, then using a "max" function. But remember that at this state the Q table is still filled with zeros.

$$Q(1,5) = R(1,5) + 0.8 \cdot \max([Q(5,1), Q(5,4), Q(5,5)])$$

$$\therefore Q(1,5) = 100 + 0.8(0)$$

As the new state is 5 and this state is the goal state, we finish our episode. Now at the end of this episode the Q matrix will be:

State	Action					
	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	0	100
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0

Episode 2

On this new state we randomly selected the state "room 3", by looking the R matrix we have 3 possible actions on this state, also now by chance we chose the action 1

State	Action					
	0	1	2	3	4	5
0	-1	-1	-1	-1	0	-1
1	-1	-1	-1	0	-1	100
2	-1	-1	-1	0	-1	-1
3	-1	0	0	-1	0	-1
4	0	-1	-1	0	-1	100
5	-1	0	-1	-1	0	100

By selecting the action "1" as our next state will have now the following possible actions

State	Action					
	0	1	2	3	4	5
0	-1	-1	-1	-1	0	-1
1	-1	-1	-1	0	-1	100
2	-1	-1	-1	0	-1	-1
3	-1	0	0	-1	0	-1
4	0	-1	-1	0	-1	100
5	-1	0	-1	-1	0	100

Now let's update our Q table:

$$Q(3, 1) = R(3, 1) + 0.8 \cdot \max([Q(1, 3), Q(1, 5)])$$

$$\therefore Q(3, 1) = 0 + 0.8(100) = 80$$

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	0	100
2	0	0	0	0	0	0
3	0	80	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0

As our new current state 1 is not the goal state, we continue the process. Now by chance from the possible actions of state 1, we choose the action 5. From the action 5 we have the possible actions: 1,4,5 [Q(5,1), Q(5,4), Q(5,5)] unfortunately we did not computed yet this values and our Q matrix remain unchanged.

Episode 100000

After a lot of episodes our Q matrix can be considered to have convergence, on this case Q will be like this:

	0	1	2	3	4	5
0	0	0	0	400	0	0
1	0	0	0	320	0	500
2	0	0	0	320	0	0
3	0	400	256	0	400	0
4	320	0	0	320	0	500
5	0	400	0	0	400	500

if we divide all of the nonzero elements by it's greatest value (on this case 500) we normalize the Q table (Optional):

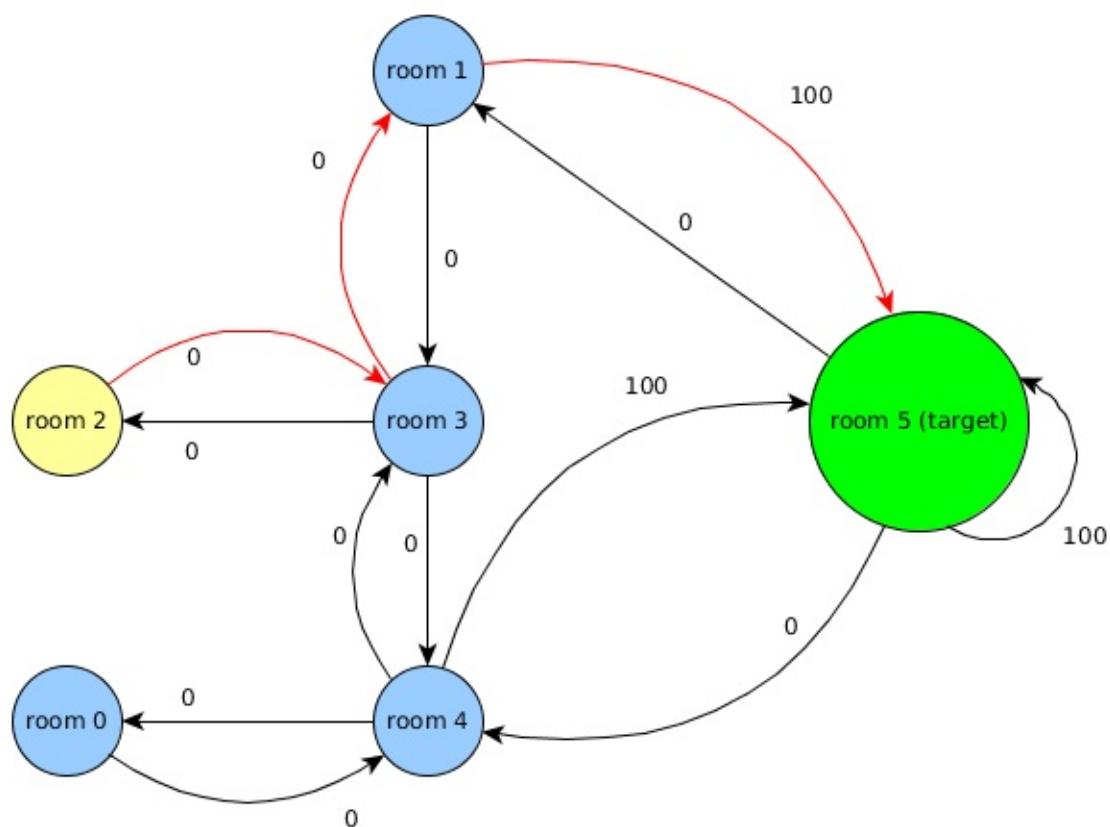
$$Q = \begin{bmatrix} & \textcolor{red}{0} & \textcolor{red}{1} & \textcolor{red}{2} & \textcolor{red}{3} & \textcolor{red}{4} & \textcolor{red}{5} \\ \textcolor{red}{0} & 0 & 0 & 0 & 0 & 80 & 0 \\ \textcolor{red}{1} & 0 & 0 & 0 & 64 & 0 & 100 \\ \textcolor{red}{2} & 0 & 0 & 0 & 64 & 0 & 0 \\ \textcolor{red}{3} & 0 & 80 & 51 & 0 & 80 & 0 \\ \textcolor{red}{4} & 64 & 0 & 0 & 64 & 0 & 100 \\ \textcolor{red}{5} & 0 & 80 & 0 & 0 & 80 & 100 \end{bmatrix}$$

What to do with converged Q table.

Now with the good Q table, imagine that we start from the state "room 2". If we keep choosing the action that gives maximum Q value, we're going from state 2 to 3, then from 3 to 1, then from 1 to 5, and keeps at 5. In other words we choose the actions [2,3,1,5].

Just one point to pay attention. On state 3 we have the options to choose action 1 or 4, because both have the same max value, we choose the action 1. But the other action would also give the same cumulative reward. [0+0+100]

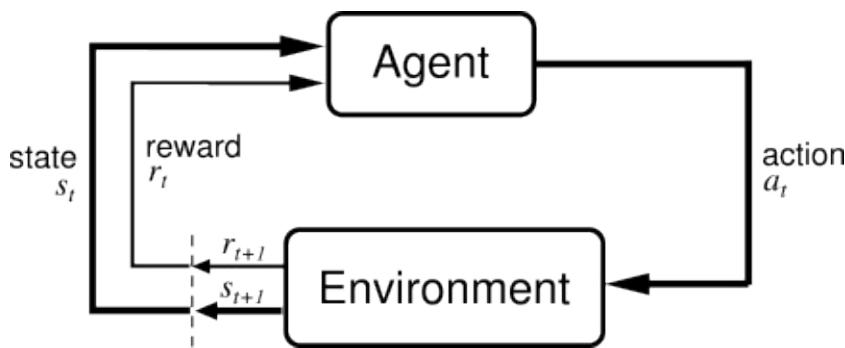
$$Q = \begin{bmatrix} & \textcolor{red}{0} & \textcolor{red}{1} & \textcolor{red}{2} & \textcolor{red}{3} & \textcolor{red}{4} & \textcolor{red}{5} \\ \textcolor{red}{0} & 0 & 0 & 0 & 0 & 80 & 0 \\ \textcolor{red}{1} & 0 & 0 & 0 & 64 & 0 & \textcolor{yellow}{100} \\ \textcolor{red}{2} & 0 & 0 & 0 & \textcolor{green}{64} & 0 & 0 \\ \textcolor{red}{3} & 0 & \textcolor{yellow}{80} & 51 & 0 & 80 & 0 \\ \textcolor{red}{4} & 64 & 0 & 0 & 64 & 0 & 100 \\ \textcolor{red}{5} & 0 & 80 & 0 & 0 & 80 & \textcolor{red}{100} \end{bmatrix}$$



Following converged Q table gives optimum policy

Working out this example in Matlab

Now we're ready to mix those things on the computer, we're going to develop 2 functions, one for representing our agent, and the other for the environment.



Environment

We start by modeling the environment. Which is the part that receives an action from the agent and give as feedback, a immediate reward and state information. This state is actually the state of the environment after some action is made. Just to remember, if our system is markovian all information necessary for choosing the best future action, is encoded on this state.

Observe that the environment has the matrix R and all models needed to completely implement the universe. For example the environment could be a game, our real world (Grid World) etc...

```

function [ reward, state ] = simple_RL_enviroment( action, restart )
% Simple enviroment of reinforcement learning example
%   http://mnemstudio.org/path-finding-q-learning-tutorial.htm
persistent current_state
if isempty(current_state)
    % Initial random state (excluding goal state)
    current_state = randi([1,5]);
end

% The rows of R encode the states, while the columns encode the actions
R = [ -1 -1 -1 -1  0  -1; ...
       -1 -1 -1  0 -1 100; ...
       -1 -1 -1  0 -1  -1; ...
       -1  0  0 -1  0  -1; ...
       0 -1 -1  0 -1 100; ...
       -1  0 -1 -1  0 100 ];

```

```

% Sample our R matrix (model)
reward = R(current_state,action);

% Good action taken
if reward ~= -1
    % Returns next state (st+1)
    current_state = action;
end

% Game should be reseted
if restart == true
    % Choose another initial state
    current_state = randi([1,5]);
    reward = -1;
    % We decrement 1 because matlab starts the arrays at 1, so just take
    % the messages with the same value as the figures on the tutorial
    % take 1....
    fprintf('Enviroment initial state is %d\n',current_state-1);
end

state = current_state;
end

```

Agent

Now on the agent side, we must train to gather experience from the environment. After this we use our learned Q table to act optimally, which means just follow the Q table looking for the actions that gives maximum expected reward. As you may expect the agent interface with the external world through the "simple_RL_enviroment" function.

```

function [ Q ] = simple_RL_agent( )
% Simple agent of reinforcement learning example

```

```
% http://mnemstudio.org/path-finding-q-learning-tutorial.htm
% Train, then normalize Q (divide Q by it's biggest value)
Q = train(); Q = Q / max(Q(:));
% Get the best actions for each possible initial state (1,2,3,4,5)
test(Q);
end

function Q = train()
% Initial training parameters
gamma = 0.8;
goalState=6;
numTrainingEpisodes = 20;
% Set Q initial value
Q = zeros(6,6);

% Learn from environment interaction
for idxEpisode=1:numTrainingEpisodes
    validActionOnState = -1;
    % Reset environment
    [~,currentState] = simple_RL_environment(1, true);

    % Episode (initial state to goal state)
    % Break only when we reach the goal state
    while true
        % Choose a random action possible for the current state
        while validActionOnState == -1
            % Select a random possible action
            possibleAction = randi([1,6]);

            % Interact with environment and get the immediate reward
            [ reward, ~ ] = simple_RL_environment(possibleAction, false);
            validActionOnState = reward;
        end
        validActionOnState = -1;

        % Update Q
        % Get the biggest value from each row of Q, this will create
        % qMax for each state
        next_state = possibleAction;
        qMax = max(Q,[],2);
        Q(currentState,possibleAction) = reward + ...
            (gamma*(qMax(next_state)));

        if currentState == goalState
            break;
        end

        % Non this simple example the next state will be the action
        currentState = possibleAction;
    end
    fprintf('Finished episode %d restart environment\n',idxEpisode);
end
end
```

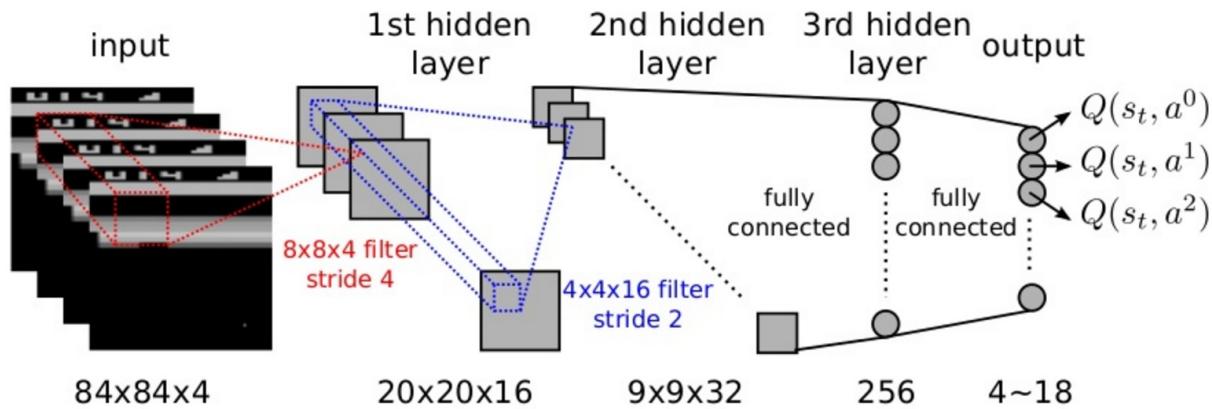
```
function test(Q)
    % Possible permuted initial states, observe that you don't include
    % goal state 6 (room5)
    possible_initial_states = randperm(5);
    goalState=6;

    % Get the biggest action for every state
    [~, action_max] = max(Q,[],2);

    for idxStates=1:numel(possible_initial_states)
        curr_state = possible_initial_states(idxStates);
        fprintf('initial state room_%d actions=[ ', curr_state-1);
        % Follow optimal policy from intial state to goal state
        while true
            next_state = action_max(curr_state);
            fprintf(' %d,', next_state-1);
            curr_state = next_state;
            if curr_state == goalState
                fprintf(']');
                break
            end
        end
        fprintf('\n');
    end
end
```

Deep Q Learning

Deep Q Learning



Introduction

On the previous chapter we learned about the "old school" Q learning, we used matrices to represent our Q tables. This somehow implies that you at least know how many states (rows) you have on your environment, the problem is that sometimes this is not true.

$$Q = \begin{matrix} & \begin{matrix} \textcolor{red}{0} & \textcolor{red}{1} & \textcolor{red}{2} & \textcolor{red}{3} & \textcolor{red}{4} & \textcolor{red}{5} \end{matrix} \\ \begin{matrix} \textcolor{red}{0} \\ \textcolor{red}{1} \\ \textcolor{red}{2} \\ \textcolor{red}{3} \\ \textcolor{red}{4} \\ \textcolor{red}{5} \end{matrix} & \left[\begin{matrix} 0 & 0 & 0 & 0 & 80 & 0 \\ 0 & 0 & 0 & 64 & 0 & \textcolor{yellow}{100} \\ 0 & 0 & 0 & \textcolor{green}{64} & 0 & 0 \\ 0 & \textcolor{yellow}{80} & 51 & 0 & 80 & 0 \\ 64 & 0 & 0 & 64 & 0 & 100 \\ 0 & 80 & 0 & 0 & 80 & \textcolor{red}{100} \end{matrix} \right] \end{matrix}$$

On this Q table we can say the expectation of winning the game by taking the action 1 at state 3 is 80. Bigger Q values means that we expect to win.

Also we learned that reinforcement learning is about learning how to behave on some environment where our only feedback is some sparse and time delayed "labels" called rewards. They are time delayed because there are cases where the environment will only tell if your action was good or bad some time after you actually moved.

Some Formalization before continue

Episode:

Considering our environment "Markovian" which means that our state encode everything needed to take a decision. We define an episode (game) as finite set of states, actions and rewards.

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_{n-1}, a_{n-1}, r_n, r_{terminal}$$

For instance here " r_1 " means the reward that we take by taking the action a_0 on state t_0 . An episode always finish on an end state (Game over).

Policy

Policy is defined as a "map" from states to actions, is the reinforcement learning objective to find the optimal policy. An optimal policy can be derived from a Q function.

Return or Future Total Reward

We define return as the sum of all immediate rewards on an episode. Which means the sum of rewards until the episode finish.

$$R_t = r_1 + r_2 + r_3 + \dots + r_n$$

$$\therefore R_t = \sum_{t=0}^{N-1} r_{t+1}$$

Future Discounted Reward

To give some sort of flexibility we add to our total reward a parameter called gamma(γ). If gamma=0 all future rewards will be discarded, if gamma=1 all future rewards will be considered.

$$R_t = \sum_{t=0}^{N-1} \gamma^t \cdot r_{t+1}$$

Q function

We define the function $Q(s,a)$ as maximum expected "Future Discounted Reward" that we can get if we take an action "a" on state "s", and continue optimally from that point until the end of the episode. When we say "continue optimally" means that we continue choosing actions from the policy derived from Q.

$$Q(s_t, a_t) = \max(R_{t+1})$$

Bellow we show how to derive a policy from the Q function, basically we want the action that gives the biggest Q value on state "s":

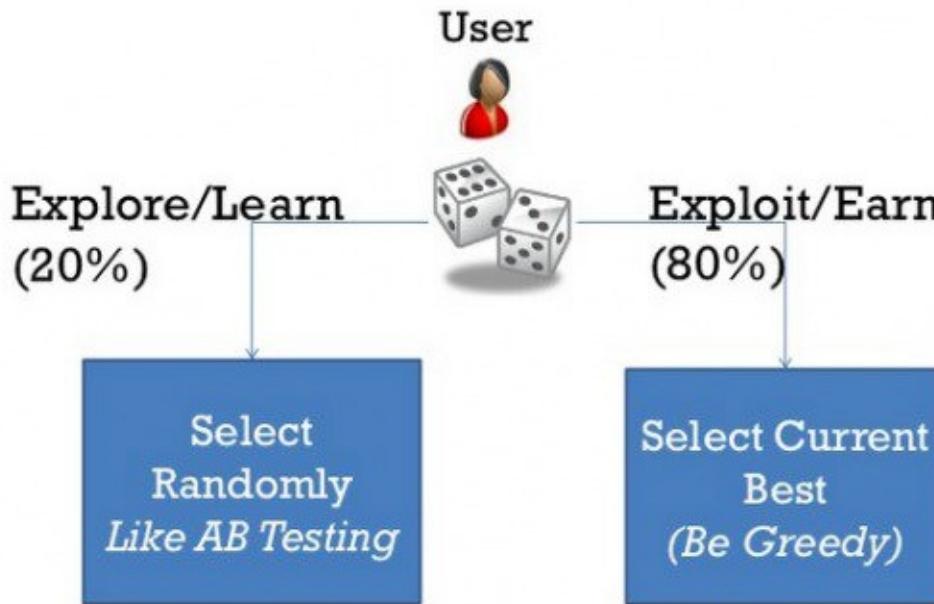
$$\pi(s) = \max_a [Q(s, a)]$$

The function " \max_a " will search for the action "a" that maximizes $Q(s,a)$ You can think that the Q function is "the best possible score at the end of the game after performing action a in state s". So if you have the Q function you have everything needed to win all games!.

Greedy policy

If we choose an action that "ALWAYS" maximize the "Discounted Future Reward", you are acting greedy. This means that you are not exploring and you could miss some better actions. This is called

exploration-exploitation problem. To solve this we use an algorithm called $\epsilon - greedy$, where a small probability ϵ will choose a completely random action from time to time.



How to get the Q function

Basically we get the Q function by experience, using an iterative process called bellman equation.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot [r + \gamma \cdot \max_{a_{t+1}} (Q(s_{t+1}, a_{t+1})) - Q(s_t, a_t)]$$

In terms of algorithm.

```

initialize Q[num_states, num_actions] arbitrarily
observe initial state s
repeat
    select and carry out an action a
    observe reward r and new state s'
    Q[s, a] = Q[s, a] + α(r + γ maxa' Q[s', a'] - Q[s, a])
    s = s'
until terminated
  
```

On the beginning the estimated $Q(s,a)$ will be wrong but with time, the experiences with the environment, will give a true "r" and this reward will slowly shape our estimation to the true $Q(s,a)$.

Deep Q Network

The Deep Q learning is about using deep learning techniques to represent the Q table. Is also a kind of recipe to use Q learning on games.

Input

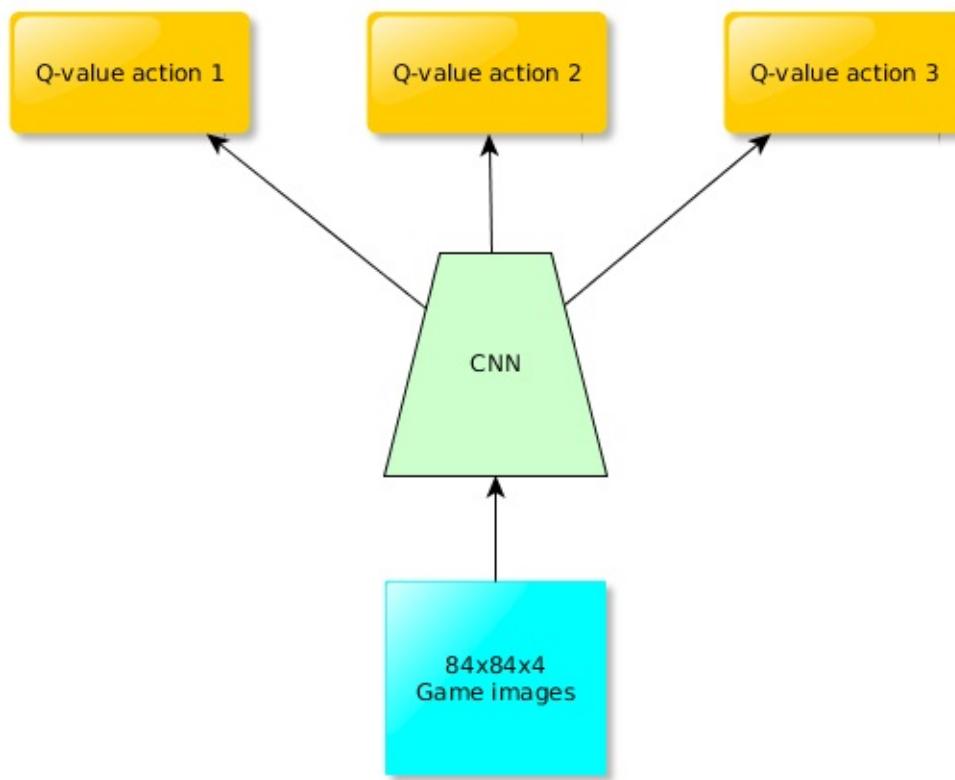
The first thing on this recipe is to get our input, as we may imagine we take information directly form the screen. To add some notion of time we actually get 4 consecutive screens.

1. Get the images
2. Rescale to 84x84
3. Convert to 8 bit grayscale.

Now we have 84x84x256x4 "stuff" as our environment state, this means 7 million states. To find structure on this data we will need a convolution neural network!.

Adding a Convolution Neural Network

Now we will give those 84x84x4 tensor to an CNN, this model will have one output for each actions which will represent a Q-value for each possible action. So for each forward propagation we will have all possible Q values for a particular state encoded on the screen. It's valid to point out that this is not a classification problem but a regression, our Q-value output is a scalar number not a class.



On the Atari paper this CNN had the following structure:

Layer	Input	Filter size	Stride	Num filters	Activation	Output
conv1	84x84x4	8x8	4	32	ReLU	20x20x32
conv2	20x20x32	4x4	2	64	ReLU	9x9x64
conv3	9x9x64	3x3	1	64	ReLU	7x7x64
fc4	7x7x64			512	ReLU	512
fc5	512			18	Linear	18

Notice that there is not pooling layer, this is done because want to retain the spatial information.

Loss function

As our CNN is a regression model our loss function will be a "squared error loss"

$$\text{Loss} = \frac{1}{2} \cdot [\underbrace{r + \max_{a_{t+1}}(Q(s_{t+1}, a_{t+1}; \theta_{t-1}))}_{\text{target}} - \underbrace{Q(s, a; \theta)}_{\text{prediction}}]^2$$

How to get the Deep Q function

Now to iterate and find the real Q value using deep learning we must follow:

1. Do a forward pass for the current state " s " (screens) to get all possible Q values
2. Do a forward pass on the new state s_{t+1} and find the action $\max_a t + 1$ (Action with biggest Q value)
3. Set Q-value target for action to $r + \gamma \max a' Q(s', a')$ (use the max calculated in step 2). For all other actions, set the Q-value target to the same as originally returned from step 1, making the error 0 for those outputs.
4. Use back-propagation and mini-batches stochastic gradient descent to update the network.

Problems with this approach

Unfortunately, we need to solve some problems with this approach. But before talking about the possible solutions let's check which are the problems.

- Exploration-Exploitation issue: This is easy, we just add some random action along the way. (just use ϵ - greedy)
- Local-Minima: During the training we're going to have a lot of screens that are highly correlated, this may guide your network to learn just a replay of the episode. To solve this we need somehow to shuffle the input mini-batch with some other playing data. (Of course of the same game but at different time)

Experience Replay

As mentioned we need to break the similarities of continuous frames during our update step. To do this we will store all game experiences during the episode inside a "replay memory" then during training we will take random mini-batches of this memory. Also you can add some human-experience by adding on the replay memory some human episodes.

Complete Recipe

Algorithm 1 Deep Q-learning with Experience Replay

```

Initialize replay memory  $\mathcal{D}$  to capacity  $N$ 
Initialize action-value function  $Q$  with random weights
for episode = 1,  $M$  do
    Initialise sequence  $s_1 = \{x_1\}$  and preprocessed sequenced  $\phi_1 = \phi(s_1)$ 
    for  $t = 1, T$  do
        With probability  $\epsilon$  select a random action  $a_t$ 
        otherwise select  $a_t = \max_a Q^*(\phi(s_t), a; \theta)$ 
        Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$ 
        Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$ 
        Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $\mathcal{D}$ 
        Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $\mathcal{D}$ 
        Set  $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$ 
        Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  according to equation 3
    end for
end for

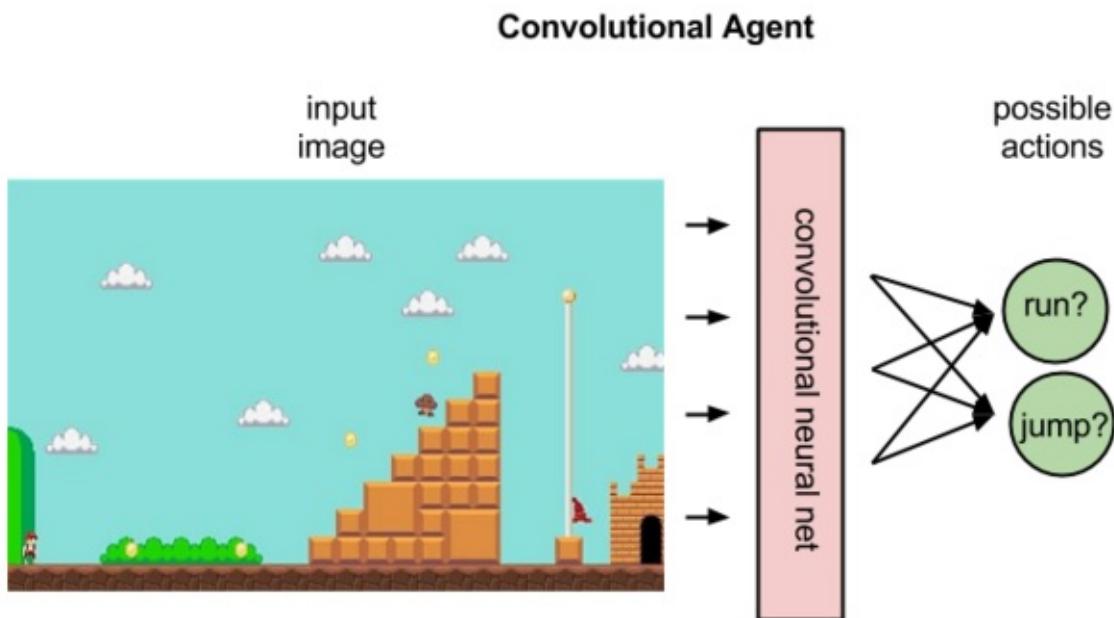
```

A complete tensorflow example can be found [here](#).

Deep Reinforcement Learning

Deep Reinforcement Learning

Introduction



On this chapter we will learn the effects of merging Deep Neural Networks with Reinforcement learning. If you follow AI news you may have heard about some stuff that AI is not capable to do without any specific programming:

- Learn how to play atari from raw image pixels
- Learn how to beat Go champions (Huge branching factor)
- Robots learning how to walk

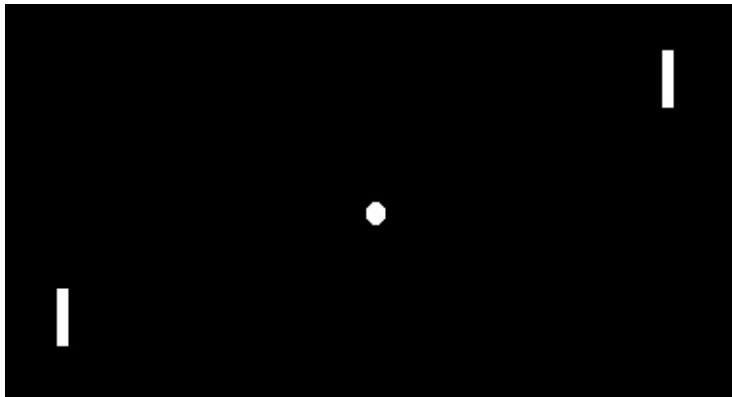
All those achievements fall on the Reinforcement Learning umbrella, more specific Deep Reinforcement Learning. Basically all those achievements arrived not due to new algorithms, but due to more Data and more powerful resources (GPUs, FPGAs, ASICs). Examples:

- Atari: Old Q-Learning algorithm but with a CNN as function approximator (Since 1988 people talk about standard RL with function approximators)
- AlphaGo: Policy gradients that use Monte Carlo Tree Search (MCTS), which is pretty standard.

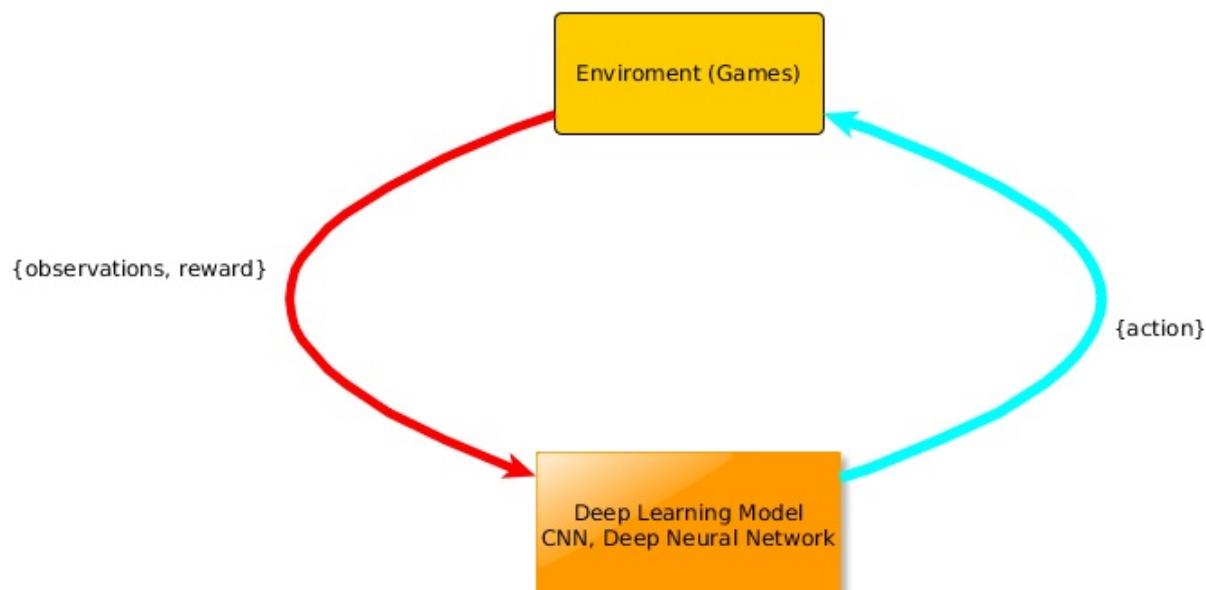
Nowadays Policy Gradients it's the favorite choice for attacking Reinforcement learning(RL) problems. Previously it was DQN (Deep Q learning). One advantage of Policy Gradients is because it can be learned end-to-end.

Policy Gradient

We will start our study with policy gradient using the pong game:



Here we have 2 possible actions (UP/DOWN) and our objective is make the ball pass the opponent paddle. Our inputs are going to be a 80x80x1 image pre-processed from a 210x160x3 image. Our game environment (openAI gym) will give a reward of +1 if you win the opponent, -1 if you lose or 0 otherwise.



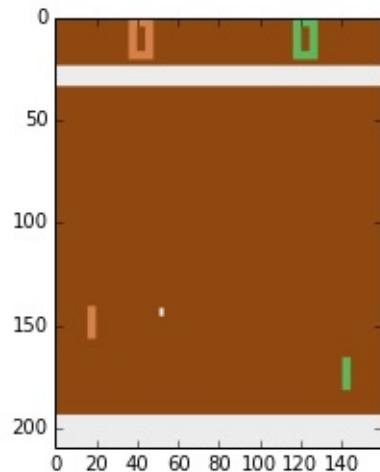
The model will do actions that maximize the rewards.
The problems that are actions that takes a lot of time to give rewards (late rewards)

Here each node is a particular game state, and each edge is a possible transition, also each edge can give a reward. Our goal is to find a policy (map from states to actions) that will give us the best action to do on each state. The whole point is that we don't know this graph, otherwise the whole thing would be just a reflex agent, and also sometimes we cannot fit all this on memory.

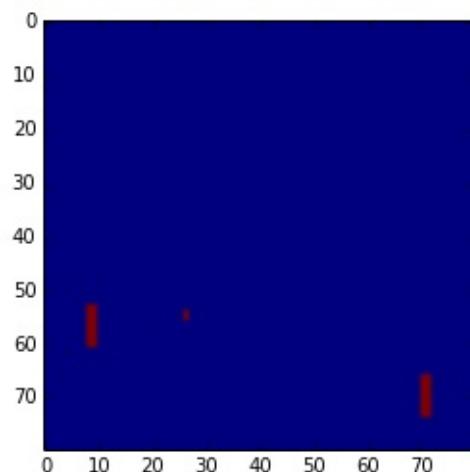
Preprocessing

On this pong example we do some operations on the original 210x160x3 (RGB) image to a simpler grayscale 80x80. This is done to simplify the neural network size.

Before



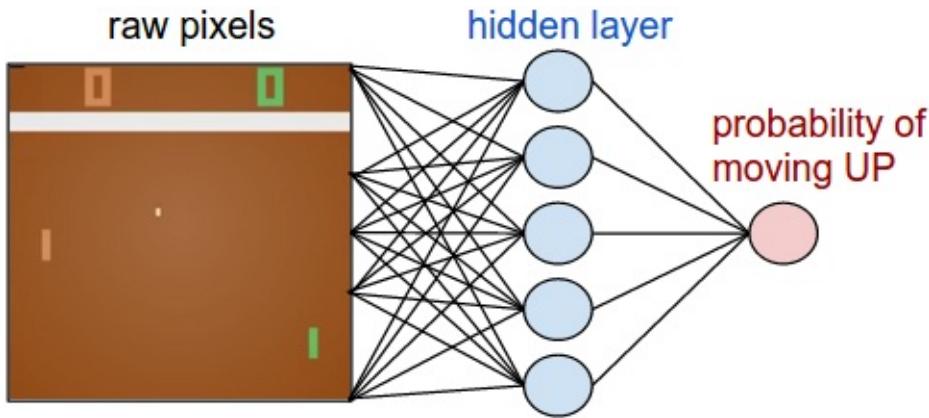
After



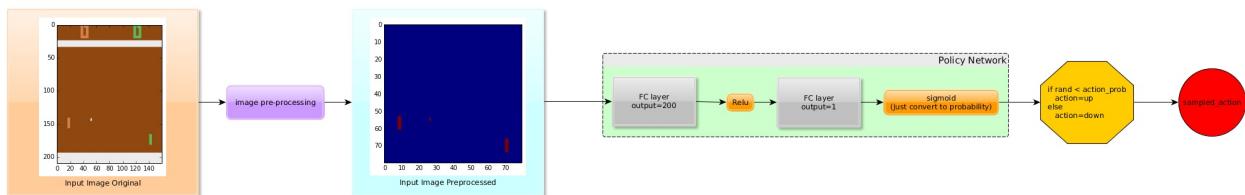
Also to give some notion of time to the network a difference between the current image and the previous one is calculated. (On the DQN paper it was used a convolution neural network with 6 image samples)

Policy Network

The basic idea is to use a machine learning model that will learn a good policy from playing the game, and receiving rewards. So adding the machine learning part. We're going to define a policy network (ex: 2 layer neural network). This simple neural network will receive the entire image and output the probability of going up.

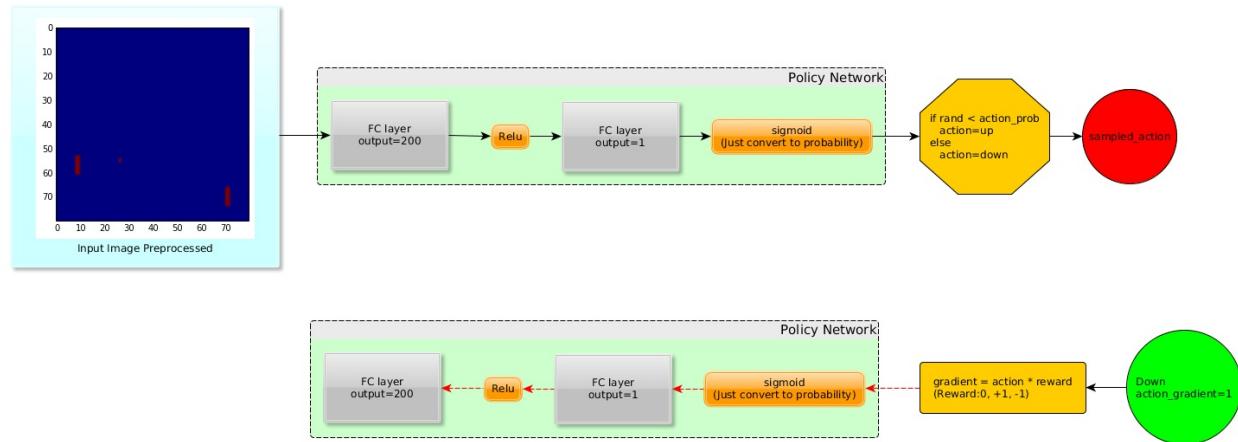


After the probability of going up is calculated we sample an action (UP/DOWN) from a uniform distribution.



Policy Gradient

One difference between supervised learning and reinforcement learning is that we don't have the correct labels to calculate the error to then back-propagate. Here is where the policy gradient idea comes. Imagine the following example:



During the training our policy network gave a "UP" probability of 30% (0.3), therefore our "DOWN" probability will be $100-30=70\%$ (0.7). During our action sampling from this distribution the "DOWN" action was selected. Now what we do is choose a unit gradient "1" and wait for the possible rewards (0,+1,-1). This will modulate our gradient to 3 possible values:

- Zero: Our weights will remain the same
- Positive: Make the network more likely to repeat this action on the future
- Negative: Make the network less likely to repeat this action on the future

So this is the magic of policy gradient, we choose a unit gradient and modulate with our rewards. After a lot of good/bad actions been selected and properly rewarded the policy network map a

"rational/optimum" policy.

Implementation notes

Here we show how to initialize our policy network as a 2 layer neural network with the first layer having 200 neurons and the second output layer 1 neuron.

```

45  # model initialization
46  # The inputs will be a 80x80 difference image (current frame - last frame)
47  # W1 will be 200,(80x80=640), so we will be able to learn 200 different
48  # "states"
49  D = 80 * 80 # input dimensionality: 80x80 grid
50  if resume:
51      # Load saved model
52      print('Loading pre-trained model')
53      model = pickle.load(open('save.p', 'rb'))
54
55      if showWeights == True:
56          print('Converting gradients to 80x80')
57          display_W1(model,H)
58  else:
59      model = {}
60      # Do "Xavier" initialization W1=[200x640] W2=[200]
61      # Our network have 200(H) neurons on the input layer and 1 output neuron
62      model['W1'] = np.random.randn(H,D) / np.sqrt(D)
63      model['W2'] = np.random.randn(H) / np.sqrt(H)

```

On the code above, D is our input difference image after pre-processing and H is the number of neurons on the hidden layer.

Bellow we have the python code for this network forward propagation (policy_forward), on this neural network we didn't use bias, but you could.

```

4  ## Forward propagation of simple policy network, consider X your image
5
6  # Do dot-product then Relu
7  h = np.dot(W1, x)
8  h[h<0] = 0
9
10 # Get "action-up" probability
11 logp = np.dot(W2, h) # compute log probability of going up
12 p = 1.0 / (1.0 + np.exp(-logp)) # sigmoid (gives probability of going up)

```

From the code above we have 2 set of weights W1,W2. Where W1 or the weights of the hidden layer can detect states from the images (Ball is above/below paddle), and W2 can decide what to do (action up/down) on those states. So the only problem now is to find W1 and W2 that lead to expert (rational) policy.

Actually we do a forward propagation to get the score for the "UP" action given an input image "x". Then from this "score" we "toss a biased coin" to actually do our atari joystick action (UP-2, Down-5).

By doing this we hope that some times we're going to do a good action and if we do this a lot of times, our policy network will learn.

```

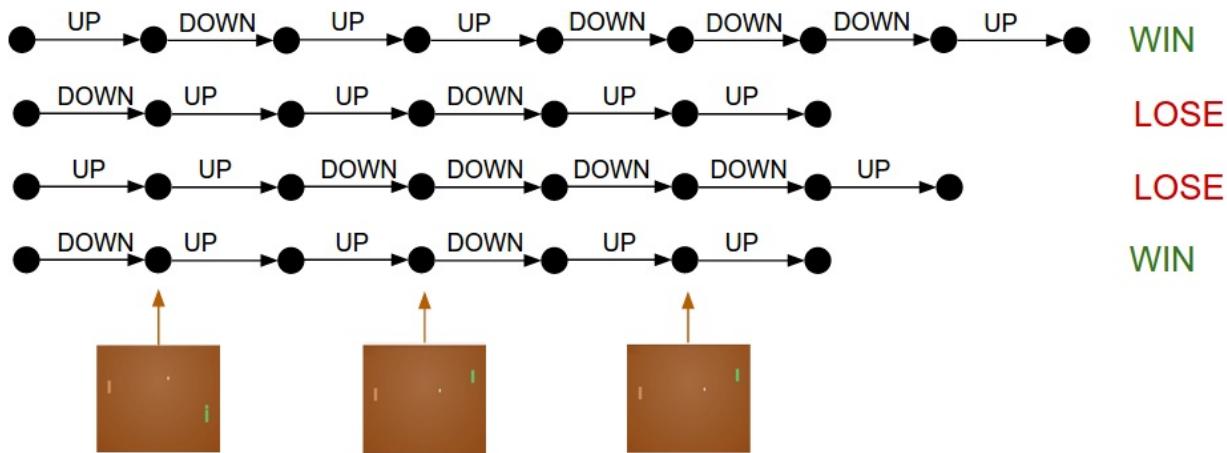
160      # forward the policy network with input image
161      # Return action probability and hidden state activations
162      aprob, h = policy_forward(x)
163
164      # Action will be the atari joystick command check the available commands
165      # here:
166      # https://github.com/openai/gym/blob/master/gym/envs/atari/atari\_env.py
167      # np.random.uniform() return a non-gaussian number from 0..1
168      action = 2 if np.random.uniform() < aprob else 5 # roll the dice!

```

Training the policy network

1. First we will initialize randomly W1,W2.
2. Then we will play 20 games (one episode).
3. Keep track of all games and their result (win/lose)
4. After a configured number of episodes we update our policy network

Assuming that each game has 200 frames, and each episode has 20 games, we have to make 4000 decisions (up/down) per episode. Suppose that we run 100 games 12 we win (+1) and 88 we loose(-1). We need to take all the $12 \times 200 = 2400$ decisions and do a positive(+1) update. (This will encourage do those actions in the future for the same detected states). Now all the $88 \times 200 = 17600$ decisions that make us loose the game we do a negative(-1) update.



The network will now become slightly more likely to repeat actions that worked, and slightly less likely to repeat actions that didn't work.

Some words about Policy Gradient

The policy gradient, are the current (2016) state of the art, but is still very far from human reasoning. For example, you don't need to crash your car hundred of times to start avoiding it. Policy gradient need a lot of samples to start to internalize correct actions, and it must have constant rewards of it. You can imagine that you are doing a kind of "brute-force" search where on the beginning we jitter some actions, and accidentally we do a good one. Also this good actions should be repeated hundred of times before the policy network start to repeat good actions. That's why the agent had to train for 3

days before start to play really good.

When policy gradient will shine:

- When the problem does not rely on very long-term planning
- When it receive frequent reward signals for good/bad actions.

Also if the action space start to be to complex (a lot of actions commands) there are other variants, like "deterministic policy gradients". This variant uses another network called "critic" that learns the score function.

Appendix

Introduction

On this chapter we will learn the basics topics to better understand the book. The following topics will be presented:

- Lua, Torch
- Tensorflow
- Python, numpy
- Matlab

Lua and Torch

Introduction

On this book I stressed out the importance of knowing how to write your own deep learning/artificial intelligence library. But is also very important specially while researching some topic, to understand the most common libraries. This chapter will teach the basics on Torch, but before that we're going also to learn Lua.

Lua language

Lua was first created to be used on embedded systems, the idea was to have a simple cross-platform and fast language. One the main features of Lua is it's easy integration with C/C++.

Lua was originally designed in 1993 as a language for extending software applications to meet the increasing demand for customization at the time.

This extension means that you could have a large C/C++ program and, some parts in Lua where you could easily change without the need to recompile everything.

Torch

Torch is a scientific computing framework based on Lua with CPU and GPU backends. You can imagine like a Numpy but with CPU and GPU implementation. Some nice features:

- Efficient linear algebra functions with GPU support
- Neural Network package, with automatic differentiation (No need to backpropagate manually)
- Multi-GPU support

First contact with Lua

Bellow we have some simple examples on Lua just to have some contact with the language.

```
print("Hello World") -- First thing, note that there is no main...
--[[[
This is how we do a multi-line comment
on lua, to execute a lua program just use...
lua someFile.lua
]]]

someVar = "Leonardo"
io.write("Size of variable is ", #someVar, " and it's value is: \\"", 
-- Variables on lua are dynamically typed...
someVar = 10; -- You can use ";" to end a statement
io.write("Now it's value is:\\"", someVar, "\\"")
```

Lua datatypes

The language offer those basic types:

- Numbers(Float)
- string
- boolean
- table

```
print(type(someVar))
someVar = 'Leonardo' -- Strings can use simple quotes
print(type(someVar))
someVar = true
print(type(someVar))
someVar = {1,2,"leo",true}
print(type(someVar))
```

Doing some math

Normally we will rely on Torch, but Lua has some math support as well.

```
io.write("5 + 3 = ", 5+3, "\n")
io.write("5 - 3 = ", 5-3, "\n")
io.write("5 * 3 = ", 5*3, "\n")
io.write("5 / 3 = ", 5/3, "\n")
io.write("5.2 % 3 = ", 5%3, "\n")
-- Generate random number between 0 and 1
io.write("math.random() : ", math.random(0,3), "\n")
-- Print float to 10 decimals
print(string.format("Pi = %.10f", math.pi))

5 + 3 = 8
5 - 3 = 2
5 * 3 = 15
5 / 3 = 1.66666666666667
5.2 % 3 = 2
math.random() : 2
Pi = 3.1415926536
```

Lua include (require)

The lua statement to include other lua files is the "require", as usual it is used to add some library

```
require 'image'
pedestrian = image.load('./pedestrianSign.png')
itorch.image(pedestrian)
```



Conditionals

Just the simple if-then-else. Lua does not have switch statement.

```
age = 17
if age < 16 then
    print(string.format("You are still a kid with %d years old\n", age))
elseif (age == 34) or (age==35) then
    print("Getting old leo...")
else
    print("Hi young man")
end

-- Lua does not have ternary operators
canVote = age > 18 and true or false -- canVote=true if age>18 else false
io.write("Can I vote: ", tostring(canVote))
```

Loops

Lua have while, repeat and for loops. For loops has also a "for-each" extension to iterate on tables.

```
i = 1
while (i <= 10) do
    io.write(i, "\n")
    i = i+1
    if i==4 then break end
end

-- Initial value, end value, increment at each loop...
for i=1,3,1 do
    io.write(i, "\n")
end

-- Create a table which is a list of items like an array
someTable = {"January", "February", "March", "April", 10}

-- Iterate on table months
for keyVar, valueVar in pairs(someTable) do
    io.write(valueVar, "(key=", keyVar, "), ")
end

January(key=1), February(key=2), March(key=3), April(key=4), 10(key=10)
```

Functions

Defining functions in Lua is quite easy, it's syntax reminds matlab.

```
-- Function definition
function getSum(a,b)
    return a+b
end

-- Call function
print(string.format("5 + 2 = %d", getSum(5,2)))
```

Tables

On Lua we use tables for everything else (ie: Lists, Dictionaries, Classes, etc...)

```
-- tables
dict = {a = 1, b = 2, c = 3}
list = {10,20,30}

-- two prints that display the same value
print(dict.a)
print(dict["a"])
-- Tables start with 1 (Like matlab)
print(list[1])

-- You can also add functions on tables
tab = {1,3,4}
-- Add function sum to table tab
function tab.sum ()
    c = 0
    for i=1,#tab do
        c = c + tab[i]
    end
    return c
end

print(tab:sum()) -- displays 8 (the colon is used for calling method:
-- tab:sum() means tab.sum(tab)
print(tab.sum()) -- On this case it will also work
print(tab)

1
1
10
8
8
{
    1 : 1
    2 : 3
    3 : 4
    sum : function: 0x4035ede8
```

```
}
```

Object oriented programming

Lua does not support directly OOP, but you can emulate all it's main functionalities (Inheritance, Encapsulation) with tables and metatables

Metatable tutorial: Used to override operations (metamethods) on tables.

```
-- [[

Create a class "Animal" with properties:height,weight,name,sound
and methods: new,getInfo,saySomething

]]

-- Define the defaults for our table
Animal = {height = 0, weight = 0, name = "No Name", sound = "No Sound"}

-- Constructor
function Animal:new (height, weight, name, sound)
    -- Set a empty metatable to the table Animal (Crazy whay to create
    setmetatable({}, Animal)
    -- Self is a reference to this table instance
    self.height = height
    self.weight = weight
    self.name = name
    self.sound = sound
    return self
end

-- Some method
function Animal:getInfo()
    animalStr = string.format("%s weighs %.1f kg, is %.1fm in tall", self.name, self.weight, self.height)
    return animalStr
end

function Animal:saySomething()
    print(self.sound)
end

-- Create an Animal
flop = Animal:new(1,10.5,"Flop","Auau")
print(flop.name) -- same as flop["name"]
print(flop:getInfo()) -- same as flop.getInfo(flop)
print(flop:saySomething())

-- Other way to say the samething
print(flop["name"])
print(flop.getInfo(flop))

-- Type of our object
```

```

print(type(flop))

Flop
Flop weighs 10.5 kg, is 1.0m in tall
Auau

Flop
Flop weighs 10.5 kg, is 1.0m in tall
table

```

File I/O

```

-- Open a file to write
file = io.open("./output.txt", "w")

-- Copy the content of the file input.txt to test.txt
for line in io.lines("./input.txt") do
    print(line)
    file:write(string.format("%s from input (At output)\n", line)) -- \
end

file:close()

Line 1 at input
Line 2 at input

```

Run console commands

```

local t = os.execute("ls")
print(t)
local catResult = os.execute("cat output.txt")
print(catResult)

FirstContactTorch.ipynb
input.txt
LuaLanguage.ipynb
output.txt
pedestrianSign.png
plot.png
true

Line 1 at input from input (At output)
Line 2 at input from input (At output)
true

```

First contact with Torch

On this section we're going to see how to do simple operations with Torch, more complex stuff will be dealt latter.

One of the torch objectives is to give some matlab functionality, an usefull cheatsheet can be found

here:

```
-- Include torch library
require 'torch'; -- Like matlab the ";" also avoid echo the output

-- Create a 2x4 matrix
m = torch.Tensor({{1, 2, 3, 4}, {5, 6, 7, 8}})
print(m)

-- Get element at second row and third collumn
print(m[{2,3}])

1 2 3 4
5 6 7 8
[torch.DoubleTensor of size 2x4]

7
```

Some Matrix operations

```
-- Define some Matrix/Tensors
a = torch.Tensor(5,3) -- construct a 5x3 matrix/tensor, uninitialized
a = torch.rand(5,3) -- Create a 5x3 matrix/tensor with random values
b=torch.rand(3,4) -- Create a 3x4 matrix/tensor with random values

-- You can also fill a matrix with values (On this case with zeros)
allZeros = torch.Tensor(2,2):fill(0)
print(allZeros)

-- Matrix multiplication and it's syntax variants
c = a*b
c = torch.mm(a,b)
print(c)
d=torch.Tensor(5,4)
d:mm(a,b) -- store the result of a*b in c

-- Transpose a matrix
m_trans = m:t()
print(m_trans)

0 0
0 0
[torch.DoubleTensor of size 2x2]

0.8259  0.6816  0.3766  0.7048
1.3681  0.9438  0.7739  1.1653
1.2885  0.9653  0.5573  0.9808
1.2556  0.8850  0.5501  0.9142
1.8468  1.3579  0.7680  1.3500
[torch.DoubleTensor of size 5x4]

1 5
2 6
```

```

3 7
4 8
[torch.DoubleTensor of size 4x2]

```

Doing operations on GPU

```

-- Include torch (cuda) library
require 'cutorch'

-- Move arrays to GPU (and convert it's types to cuda types)
a = a:cuda()
b = b:cuda()
d = d:cuda()

-- Same multiplication just different syntax
c = a*b
d:mm(a, b)

print(c)

1.1058  0.6183  1.0518  0.7451
0.5932  0.8015  0.9441  0.5477
0.4915  0.8143  0.9613  0.4345
0.1699  0.6697  0.6656  0.2500
0.6525  0.6174  0.8894  0.4446
[torch.CudaTensor of size 5x4]

```

Plotting

```

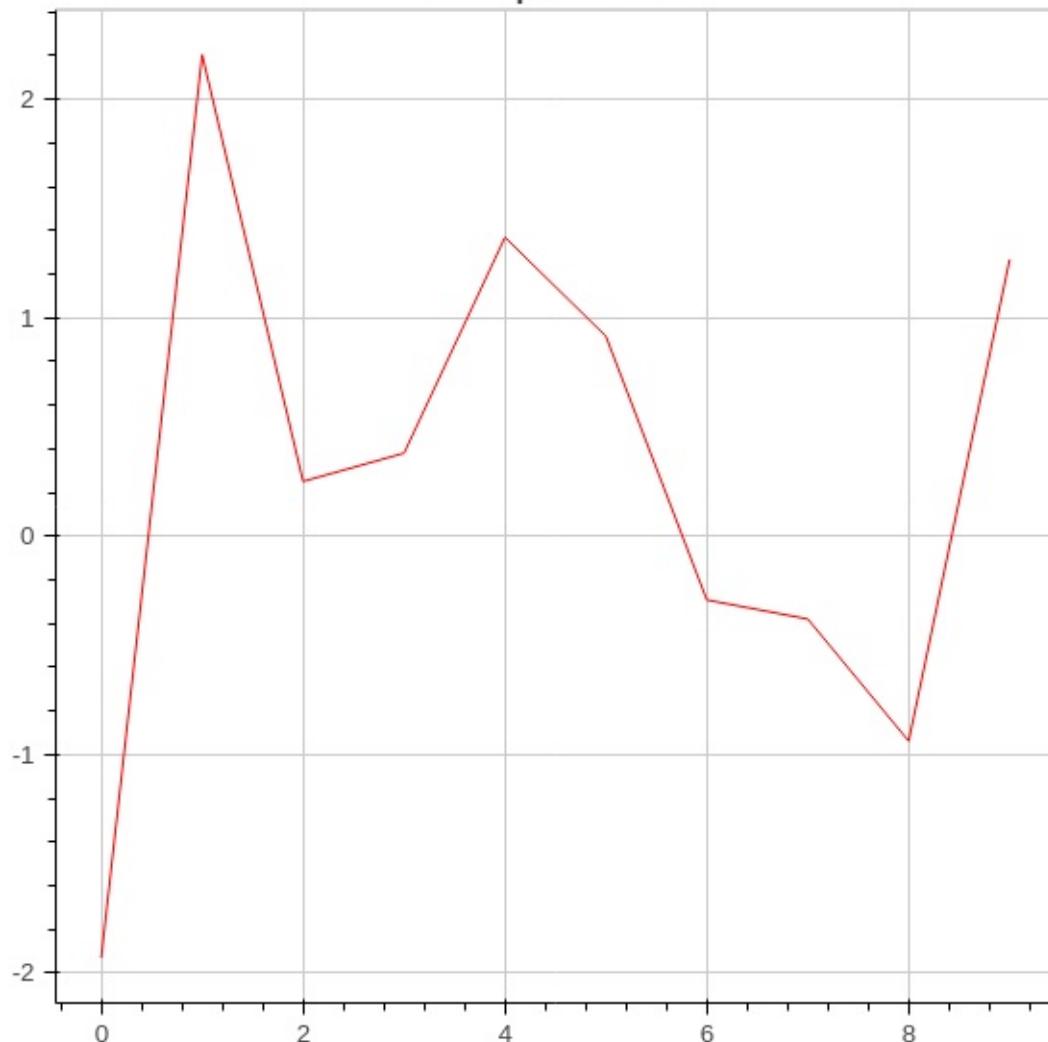
Plot = require 'itorch.Plot'

-- Give me 10 random numbers
local y = torch.randn(10)

-- Get 1d tensor from 0 to 9 (10 elements)
local x = torch.range(0, 9)
Plot():line(x, y, 'red', 'Sinus Wave'):title('Simple Plot'):draw()

```

Simple Plot



Starting with nn (XOR problem)

```

require "nn"

-- make a multi-layer perceptron
mlp = nn.Sequential();
-- 2 inputs, one output 1 hidden layer with 20 neurons
inputs = 2; outputs = 1; hiddenUnits = 20;

-- Mount the model
mlp:add(nn.Linear(inputs, hiddenUnits))
mlp:add(nn.Tanh())
mlp:add(nn.Linear(hiddenUnits, outputs))

```

Define the loss function

On torch the loss function is called criterion, as on this case we're dealing with a binary classification, we will choose the Mean Squared Error criterion

```
criterion_MSE = nn.MSECriterion()
```

Training Manually

Here we're going to back-propagate our model to get the output related to the loss gradient $dout$ then use gradient descent to update the parameters.

```
for i = 1,2500 do
    -- random sample
    local input= torch.randn(2);      -- normally distributed example in
    local output= torch.Tensor(1);
    -- Create XOR lables on the fly....
    if input[1] * input[2] > 0 then
        output[1] = -1
    else
        output[1] = 1
    end

    -- Feed to the model (with current set of weights), then calculate
    criterion_MSE:forward(mlp:forward(input), output)

    -- Reset the current gradients before backpropagate (Always do)
    mlp:zeroGradParameters()
    -- Backpropagate the loss to the hidden layer
    mlp:backward(input, criterion_MSE:backward(mlp.output, output))
    -- Update parameters(Gradient descent) with alpha=0.01
    mlp:updateParameters(0.01)
end
```

Test the network

```
x = torch.Tensor(2)
x[1] =  0.5; x[2] =  0.5; print(mlp:forward(x)) -- 0 XOR 0 = 0 (negative)
x[1] =  0.5; x[2] = -0.5; print(mlp:forward(x)) -- 0 XOR 1 = 1 (positive)
x[1] = -0.5; x[2] =  0.5; print(mlp:forward(x)) -- 1 XOR 0 = 1 (positive)
x[1] = -0.5; x[2] = -0.5; print(mlp:forward(x)) -- 1 XOR 1 = 0 (negative)

-0.8257
[torch.DoubleTensor of size 1]

0.6519
[torch.DoubleTensor of size 1]

0.4468
[torch.DoubleTensor of size 1]

-0.7814
[torch.DoubleTensor of size 1]
```

Trainning with optimim

Torch provides a standard way to optimize any function with respect to some parameters. In our case, our function will be the loss of our network, given an input, and a set of weights. The goal of training a neural net is to optimize the weights to give the lowest loss over our training set of input data. So, we are going to use optim to minimize the loss with respect to the weights, over our training set.

```
-- Create a dataset (128 elements)
batchSize = 128
batchInputs = torch.Tensor(batchSize, inputs)
batchLabels = torch.DoubleTensor(batchSize)

for i=1,batchSize do
    local input = torch.randn(2)           -- normally distributed example i
    local label = 1
    if input[1]*input[2]>0 then          -- calculate label for XOR function
        label = -1;
    end
    batchInputs[i]:copy(input)
    batchLabels[i] = label
end

-- Get flatten parameters (Needed to use optim)
params, gradParams = mlp:getParameters()
-- Learning parameters
optimState = {learningRate=0.01}

require 'optim'

for epoch=1,200 do
    -- local function we give to optim
    -- it takes current weights as input, and outputs the loss
    -- and the gradient of the loss with respect to the weights
    -- gradParams is calculated implicitly by calling 'backward',
    -- because the model's weight and bias gradient tensors
    -- are simply views onto gradParams
    local function feval(params)
        gradParams:zero()

        local outputs = mlp:forward(batchInputs)
        local loss = criterion_MSE:forward(outputs, batchLabels)
        local dloss_doutput = criterion_MSE:backward(outputs, batchLabels)
        mlp:backward(batchInputs, dloss_doutput)
        return loss,gradParams
    end
    optim.sgd(feval, params, optimState)
end
```

Test the network

```
x = torch.Tensor(2)
x[1] = 0.5; x[2] = 0.5; print(mlp:forward(x)) -- 0 XOR 0 = 0 (negative)
x[1] = 0.5; x[2] = -0.5; print(mlp:forward(x)) -- 0 XOR 1 = 1 (positive)
x[1] = -0.5; x[2] = 0.5; print(mlp:forward(x)) -- 1 XOR 0 = 1 (positive)
x[1] = -0.5; x[2] = -0.5; print(mlp:forward(x)) -- 1 XOR 1 = 0 (negative)
```

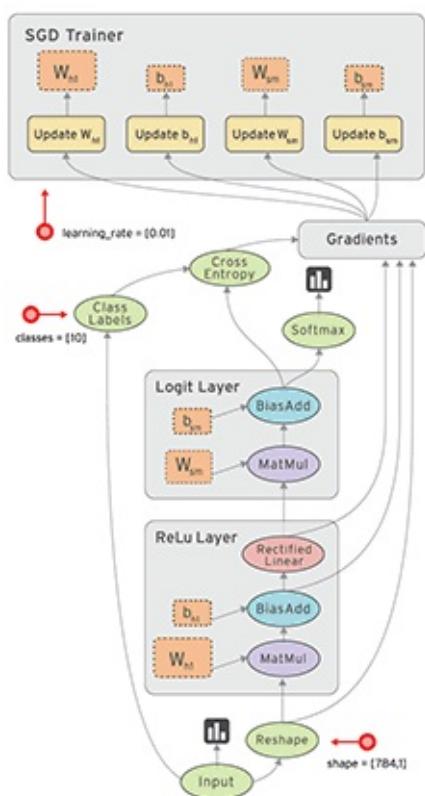
```
-0.6607  
[torch.DoubleTensor of size 1]  
  
0.5321  
[torch.DoubleTensor of size 1]  
  
0.8285  
[torch.DoubleTensor of size 1]  
  
-0.7458  
[torch.DoubleTensor of size 1]
```

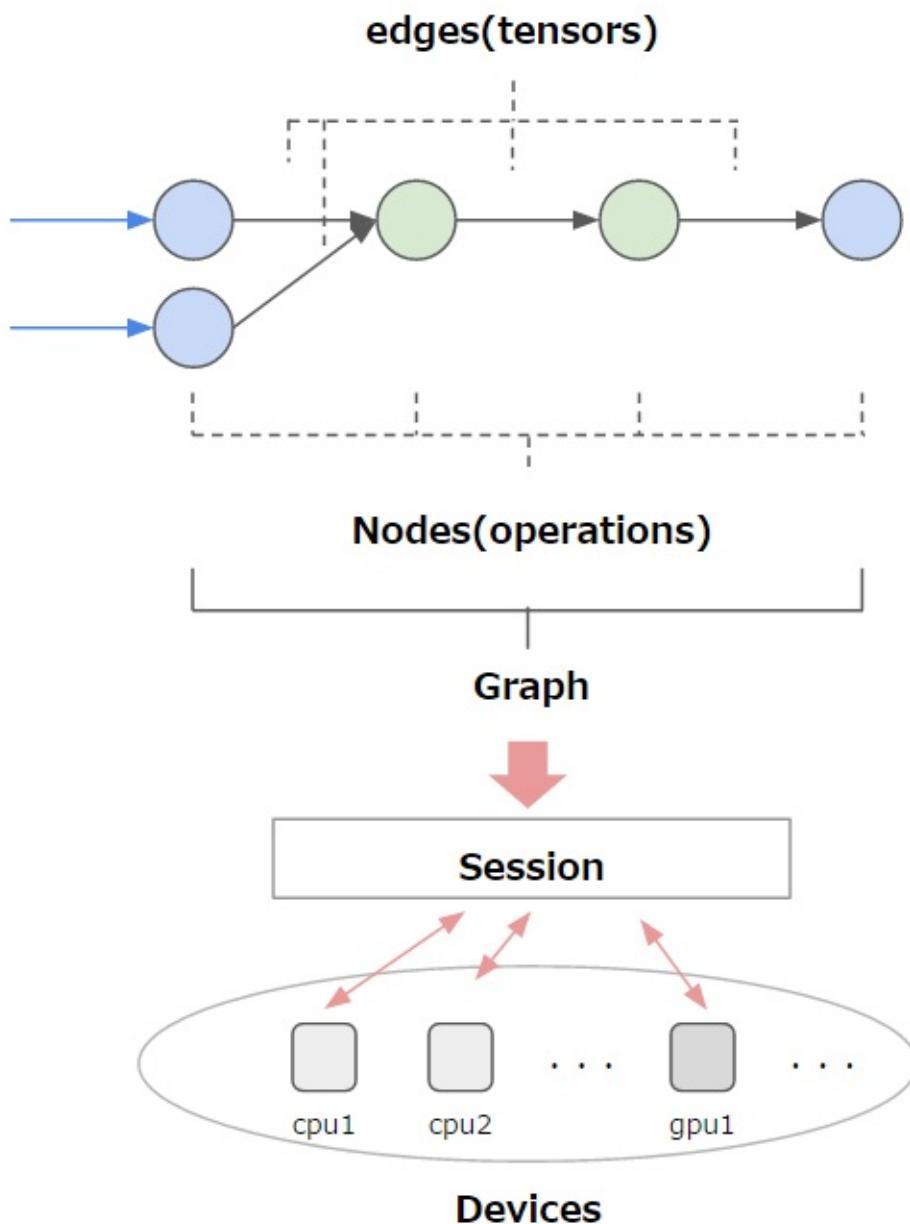
Tensorflow

Tensorflow

Introduction

On this chapter we're going to learn about tensorflow, which is the goolge library for machine learning. In simple words it's a library for numerical computation that uses graphs, on this graph the nodes are the operations, while the edges of this graph are tensors. Just to remember tensors, are multidimensional matrices, that will flow on the tensorflow graphs.





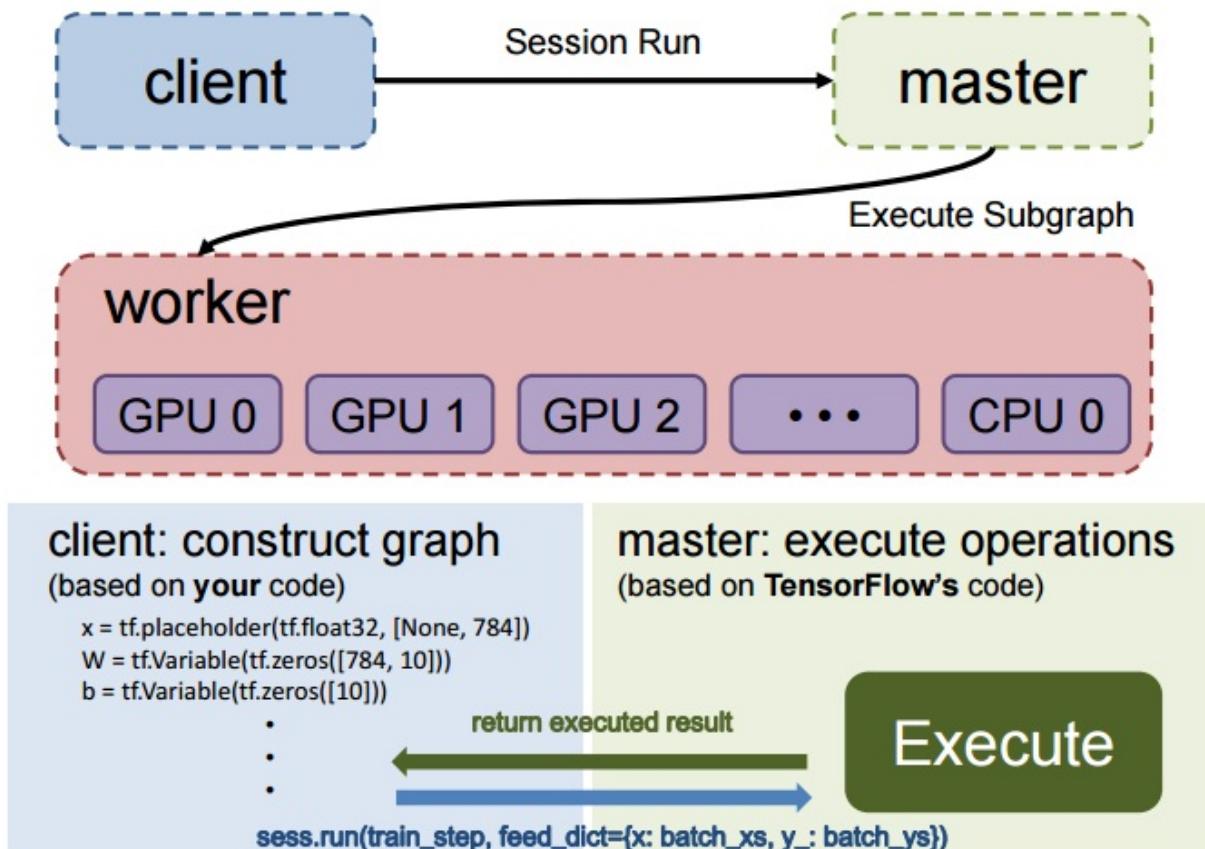
After this computational graph is created it will create a session that can be executed by multiple CPUs, GPUs distributed or not. Here are the main components of tensorflow:

1. Variables: Retain values between sessions, use for weights/bias
2. Nodes: The operations
3. Tensors: Signals that pass from/to nodes
4. Placeholders: Used to send data between your program and the tensorflow graph
5. Session: Place when graph is executed.

The TensorFlow implementation translates the graph definition into executable operations distributed across available compute resources, such as the CPU or one of your computer's GPU cards. In general you do not have to specify CPUs or GPUs explicitly. TensorFlow uses your first GPU, if you have one, for as many operations as possible.

Your job as the "client" is to create symbolically this graph using code (C/C++ or python), and ask tensorflow to execute this graph. As you may imagine the tensorflow code for those "execution nodes" is some C/C++, CUDA high performance code. (Also difficult to understand).

For example, it is common to create a graph to represent and train a neural network in the construction phase, and then repeatedly execute a set of training ops in the graph in the execution phase.



Installing

If you have already a machine with python (anaconda 3.5) and the nvidia cuda drivers installed (7.5) install tensorflow is simple

```
export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux
sudo pip3 install --ignore-installed --upgrade $TF_BINARY_URL
```

If you still need to install some cuda drivers refer [here](#) for instructions

Simple example

Just as a hello world let's build a graph that just multiply 2 numbers. Here notice some sections of the code.

- Import tensorflow library
- Build the graph
- Create a session
- Run the session

Also notice that on this example we're passing to our model some constant values so it's not so useful in real life.

source code

```

import tensorflow as tf

x = tf.constant(8)
y = tf.constant(9)
z = tf.mul(x,y)

sess = tf.Session()

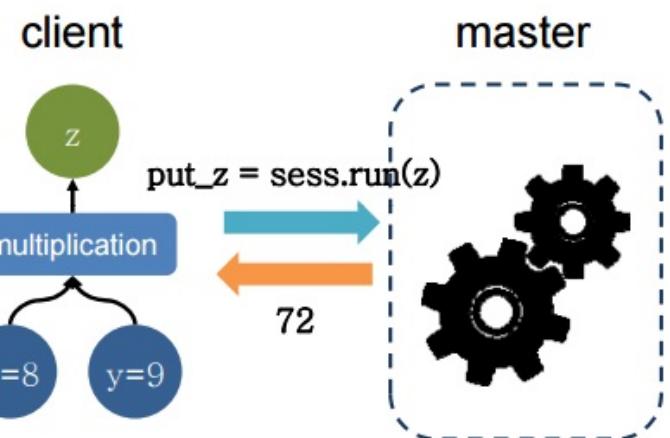
out_z = sess.run(z)

print('out_z: %d' % out_z)

```

output

```
out_z: 72
```

**Exchanging data**

Tensorflow allows exchanging data with your graph variables through "placeholders". Those placeholders can be assigned when we ask the session to run. Imagine placeholders as a way to send data to your graph when you run a session "session.run"

```

# Import tensorflow
import tensorflow as tf

# Build graph
a = tf.placeholder('float')
b = tf.placeholder('float')

# Graph
y = tf.mul(a,b)

# Create session passing the graph
session = tf.Session()
# Put the values 3,4 on the placeholders a,b
print session.run(y,feed_dict={a: 3, b:4})

```

Linear Regression on tensorflow

Now we're going to see how to create a linear regression system on tensorflow

```
# Import libraries (Numpy, Tensorflow, matplotlib)
import numpy as np
```

```

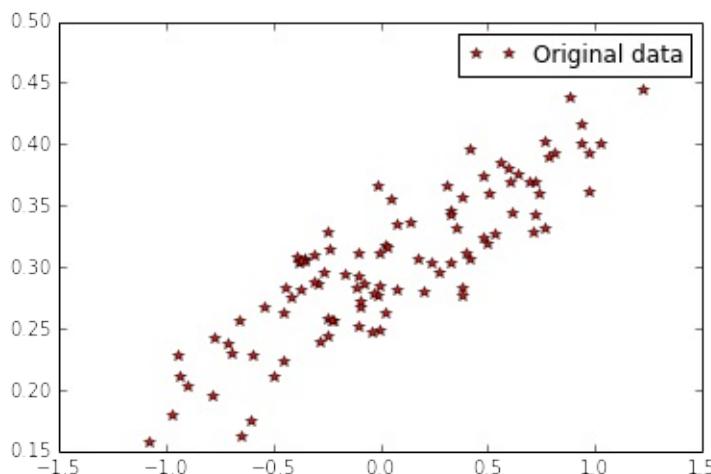
import tensorflow as tf
import matplotlib.pyplot as plt
get_ipython().magic(u'matplotlib inline')

# Create 100 points following a function y=0.1 * x + 0.3 with some noise
num_points = 100
vectors_set = []
for i in xrange(num_points):
    x1 = np.random.normal(0.0, 0.55)
    y1 = x1 * 0.1 + 0.3 + np.random.normal(0.0, 0.03)
    vectors_set.append([x1, y1])

x_data = [v[0] for v in vectors_set]
y_data = [v[1] for v in vectors_set]

# Plot data
plt.plot(x_data, y_data, 'r*', label='Original data')
plt.legend()
plt.show()

```



Now we're going to implement a graph with a function $y = W * x_{data} + b$, a loss function $loss = \text{mean}[(y - y_{data})^2]$. The loss function will return a scalar value with the mean of all distances between our data, and the model prediction.

```

# Create our linear regression model
# Variables resides internally inside the graph memory
W = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
b = tf.Variable(tf.zeros([1]))
y = W * x_data + b

# Define a loss function that take into account the distance between
# the prediction and our dataset
loss = tf.reduce_mean(tf.square(y-y_data))

# Create an optimizer for our loss function (With gradient descent)
optimizer = tf.train.GradientDescentOptimizer(0.5)
train = optimizer.minimize(loss)

```

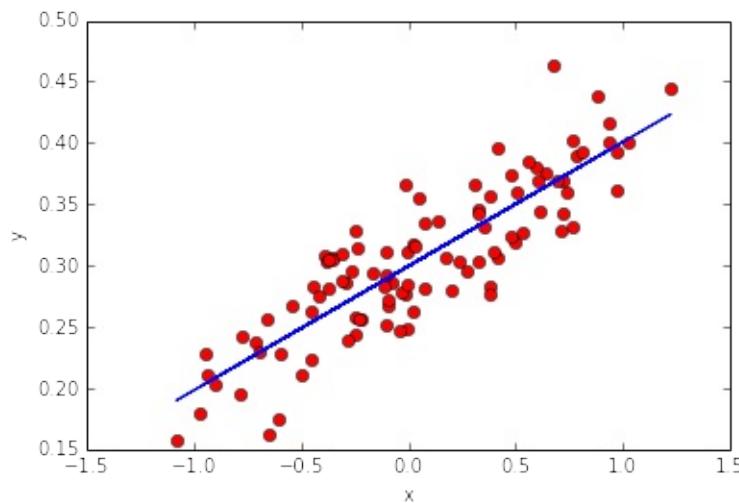
With the graph built, our job is create a session to initialize all our graph variables, which in this case is our model parameters. Then we also need to call a session x times to train our model.

```
# Run session
# Initialize all graph variables
init = tf.initialize_all_variables()
# Create a session and initialize the graph variables (Will acutally
session = tf.Session()
session.run(init)

# Train on 8 steps
for step in xrange(8):
    # Optimize one step
    session.run(train)
    # Get access to graph variables(just read) with session.run(varName)
    print("Step=%d, loss=%f, [W=%f b=%f]" % (step,session.run(loss),
                                               session.run(W), session.run(b)))

# Just plot the set of weights and bias with less loss (last)
plt.plot(x_data, y_data, 'ro')
plt.plot(x_data, session.run(W) * x_data + session.run(b))
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()

# Close the Session when we're done.
session.close()
```



Loading data

Is almost entirely up to you to load data on tensorflow, which means you need to parse the data yourself. For example one option for image classification could be to have text files with all the images filenames, followed by it's class. For example:

trainingFile.txt

```
image1.png 0
image2.png 0
image3.png 1
image4.png 1
image5.png 2
image6.png 2
```

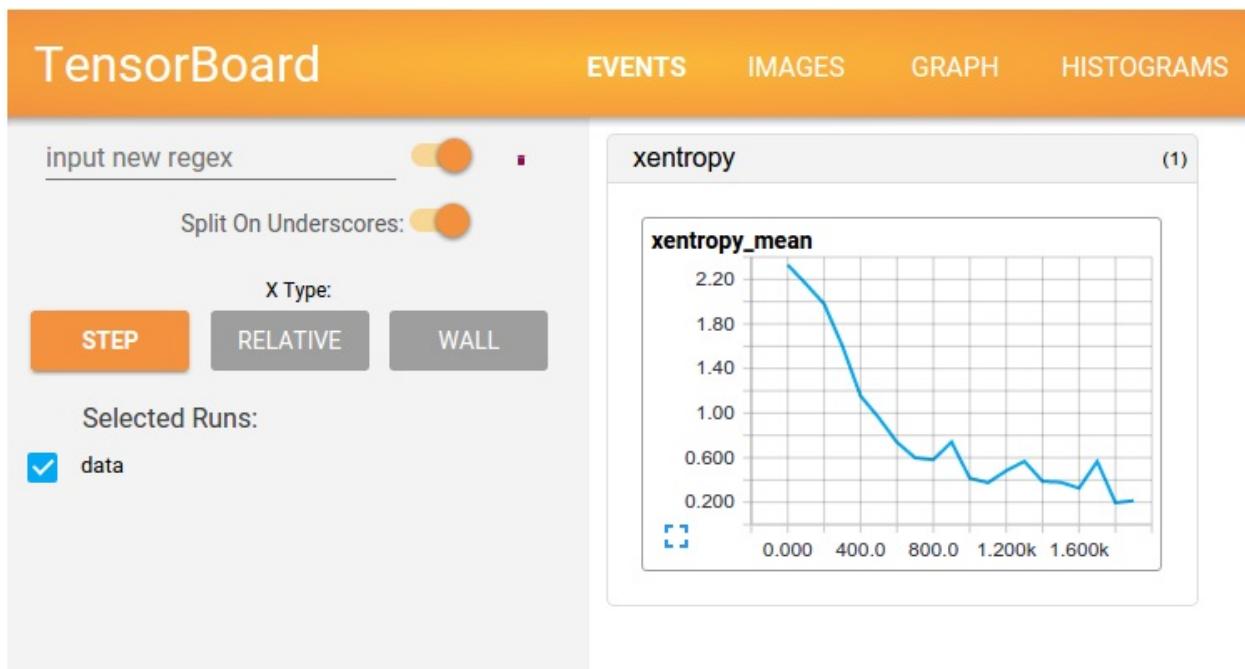
A common API to load the data would be something like this.

```
train_data, train_label = getDataFromFile('trainingFile.txt')
val_data, val_label = getDataFromFile('validationFile.txt')

## Give to your graph through placeholders...
```

Tensorboard

Tensorflow offers a solution to help visualize what is happening on your graph. This tool is called Tensorboard, basically is a webpage where you can debug your graph, by inspecting its variables, node connections etc...



In order to use tensorboard you need to annotate on your graph, with the variables that you want to inspect, ie: the loss value. Then you need to generate all the summaries, using the function `tf.merge_all_summaries()`.

Optionally you can also use the function "`tf.name_scope`" to group nodes on the graph.

After all variables are annotated and you configure your summary, you can go to the console and call:

```
tensorboard --logdir=/home/leo/test
```

Considering the previous example here are the changes needed to add information to tensorboard.

- 1) First we annotate the information on the graph that you are interested to inspect building phase.

Then call `merge_all_summaries()`. On our case we annotated loss (scalar) and W,b(histogram)

```
# Create our linear regression model
# Variables resides internally inside the graph memory

#tf.name_scope organize things on the tensorboard graphview
with tf.name_scope("LinearReg") as scope:
    w = tf.Variable(tf.random_uniform([1], -1.0, 1.0), name="Weights")
    b = tf.Variable(tf.zeros([1.0]), name="Bias")
    y = w * x_data + b

# Define a loss function that take into account the distance between
# the prediction and our dataset
with tf.name_scope("LossFunc") as scope:
    loss = tf.reduce_mean(tf.square(y-y_data))

# Create an optimizer for our loss function
optimizer = tf.train.GradientDescentOptimizer(0.5)
train = optimizer.minimize(loss)

##### Tensorboard stuff
# Annotate loss, weights and bias (Needed for tensorboard)
loss_summary = tf.scalar_summary("loss", loss)
w_h = tf.histogram_summary("W", w)
b_h = tf.histogram_summary("b", b)

# Merge all the summaries
merged_op = tf.merge_all_summaries()
```

2) During our session creation we need to add a call to "tf.train.SummaryWriter" to create a writer. You need to pass a directory where tensorflow will save the summaries.

```
# Initialize all graph variables
init = tf.initialize_all_variables()

# Create a session and initialize the graph variables (Will acutally
session = tf.Session()
session.run(init)

# Writer for tensorboard (Directory)
writer_tensorboard = tf.train.SummaryWriter('/home/leo/test', session)
```

3) Then when we execute our graph, for example during training we can ask tensorflow to generate a summary. Of course calling this every time will impact performance. To manage this you could call this at the end of every epoch.

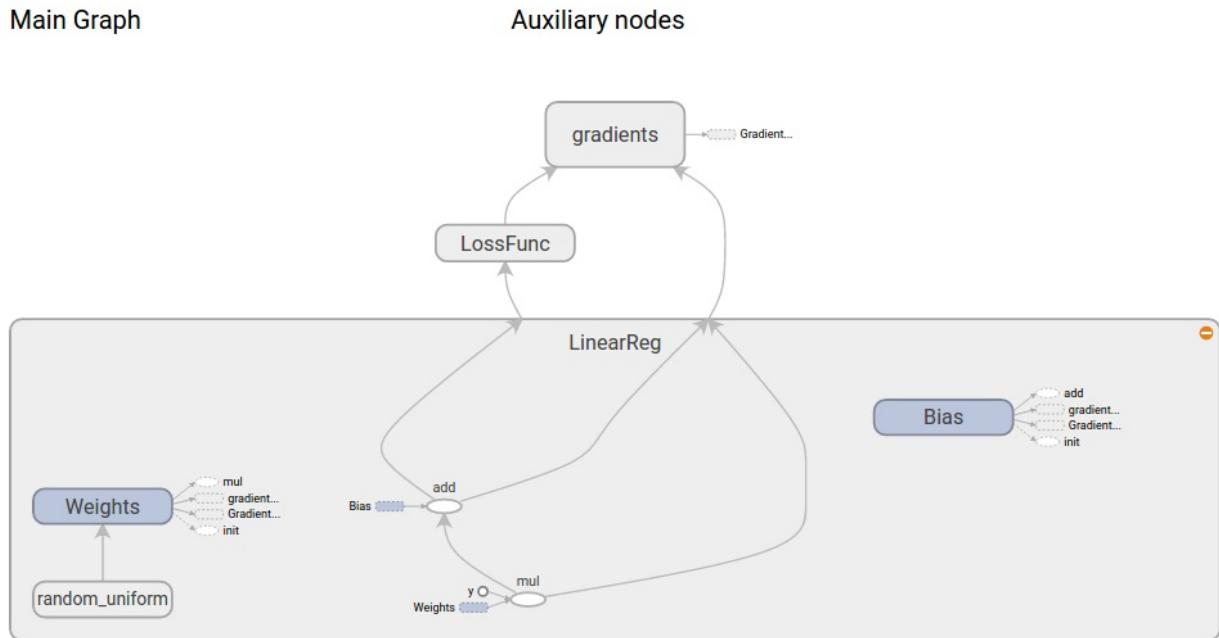
```
for step in xrange(1000):
    # Optimize one step
    session.run(train)

    # Add summary (Everytime could be to much....)
```

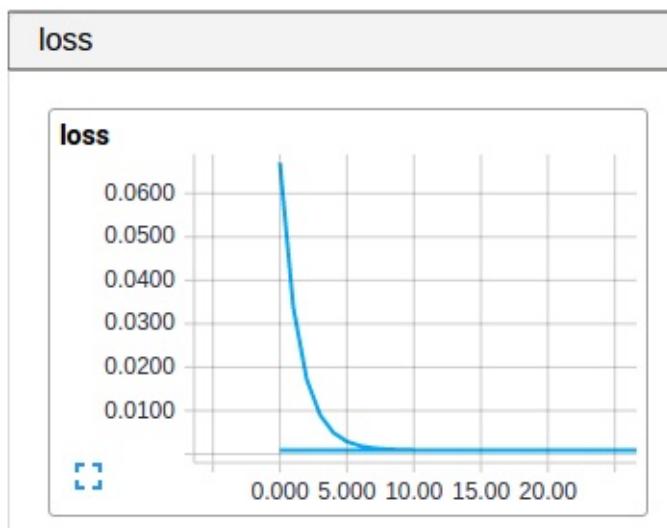
```
result_summary = session.run(merged_op)
writer_tensorboard.add_summary(result_summary, step)
```

Results on tensorboard

Here we can see our linear regression model as a computing graph.



Below we can see how the loss evolved on each iteration.



Sometimes ipython hold versions of your graph that create problems when using tensorboard, one option is to restart the kernel, if you have problems.

Using GPUs

Tensorflow also allows you to use GPUs to execute graphs or particular sections of your graph.

On common machine learning system you would have one multi-core CPU, with one or more GPUs,
320

tensorflow represent them as follows

- "/cpu:0": Multicore CPU
- "/gpu0": First GPU
- "/gpu1": Second GPU

Unfortunately tensorflow does not have an official function to list the devices available on your system, but there is an unofficial way.

```
from tensorflow.python.client import device_lib
def get_devices_available():
    local_device_protos = device_lib.list_local_devices()
    return [x.name for x in local_device_protos]

print(get_devices_available())
['/cpu:0', '/gpu:0', '/gpu:1']
```

Fix graph to a device

Use the "with tf.device('/gpu:0')" statement on python to lock all nodes on this graph block to a particular gpu.

```
import tensorflow as tf

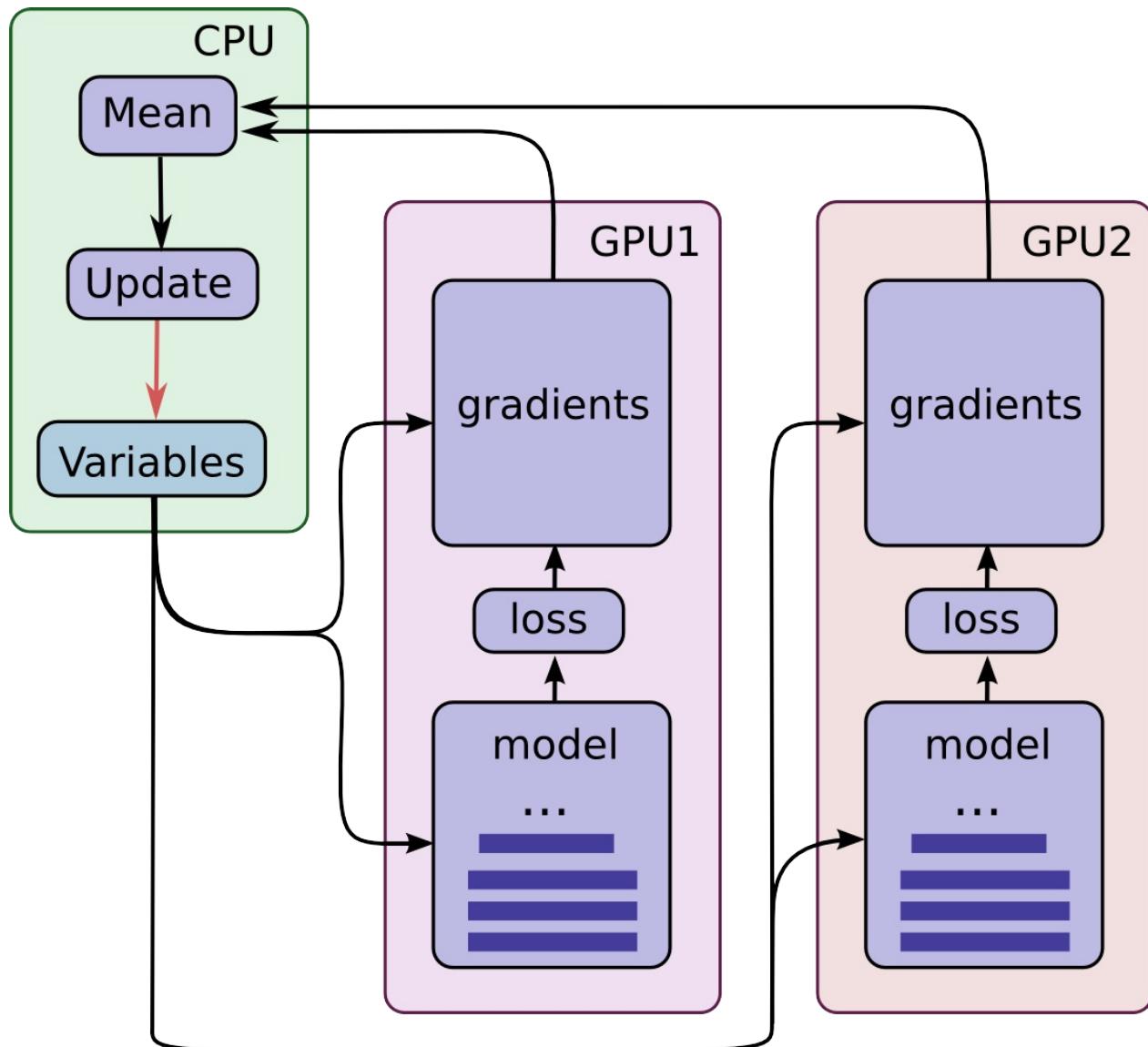
# Creates a graph.
with tf.device('/gpu:0'):
    a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a')
    b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b')
    c = tf.matmul(a, b)

# Creates a session with log_device_placement set to True, this will
# on the log how tensorflow is mapprint the operations on devices
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
# Runs the op.
print(sess.run(c))
sess.close()

[[ 22.  28.]
 [ 49.  64.]]
```

Multiple Gpus and training

Now we will explain how training is one on a multiple GPU system.



Baiscally the steps for multiple gpu training is this:

1. Separate your training data in batches as usual
2. Create a copy of the model in each gpu
3. Distribute different batches for each gpu
4. Each gpu will forward the batch and calculate it's gradients
5. Each gpu will send the gradients to the cpu
6. The cpu will average each gradient, and do a gradient descent. The model parameters are updated with the gradients averaged across all model replicas.
7. The cpu will distribute the new model to all gpus
8. the process loop again until all training is done

Multi Layer Perceptron MNIST

Multi Layer Perceptron MNIST

Load tensorflow library and MNIST data

```
import tensorflow as tf

# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)

print('Test shape:', mnist.test.images.shape)
print('Train shape:', mnist.train.images.shape)

Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
Test shape: (10000, 784)
Train shape: (55000, 784)
```

Neural network parameters

```
# Parameters
learning_rate = 0.001
training_epochs = 15
batch_size = 100
display_step = 1

# Network Parameters
n_hidden_1 = 256 # 1st layer number of features
n_hidden_2 = 256 # 2nd layer number of features
n_input = 784 # MNIST data input (img shape: 28*28)
n_classes = 10 # MNIST total classes (0-9 digits)
```

Build graph

```
x = tf.placeholder("float", [None, n_input])
y = tf.placeholder("float", [None, n_classes])

# Create model
def multilayer_perceptron(x, weights, biases):
    # Use tf.matmul instead of "*" because tf.matmul can change it's
    print('x:', x.get_shape(), 'W1:', weights['h1'].get_shape(), 'b1:',
    # Hidden layer with RELU activation
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1']) # (x*100, 256)
    layer_1 = tf.nn.relu(layer_1)
```

```

# Hidden layer with RELU activation
print( 'layer_1:', layer_1.get_shape(), 'W2:', weights['h2'].get_
layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
layer_2 = tf.nn.relu(layer_2)

# Output layer with linear activation
print( 'layer_2:', layer_2.get_shape(), 'W3:', weights['out'].get_
out_layer = tf.matmul(layer_2, weights['out']) + biases['out'] #
print('out_layer:', out_layer.get_shape())

return out_layer

```

Initialize weights and construct the model

```

# Store layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])), #
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])), #
    'out': tf.Variable(tf.random_normal([n_hidden_2, n_classes])) #
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])), #
    'b2': tf.Variable(tf.random_normal([n_hidden_2])), #
    'out': tf.Variable(tf.random_normal([n_classes])) #
}

# Construct model
pred = multilayer_perceptron(x, weights, biases)

x: (?, 784) W1: (784, 256) b1: (256, )
layer_1: (?, 256) W2: (256, 256) b2: (256, )
layer_2: (?, 256) W3: (256, 10) b3: (10, )
out_layer: (?, 10)

```

Define Loss function, and Optimizer

```

# Cross entropy loss function
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(pred, )

# On this case we choose the AdamOptimizer
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).mini

```

Launch graph

```

# Initializing the variables
init = tf.initialize_all_variables()

# Launch the graph
with tf.Session() as sess:
    sess.run(init)

    # Training cycle
    for epoch in range(training_epochs):

```

```

avg_cost = 0.
total_batch = int(mnist.train.num_examples/batch_size)
# Loop over all batches
for i in range(total_batch):
    batch_x, batch_y = mnist.train.next_batch(batch_size)
    # Run optimization op (backprop) and cost op (to get loss)
    _, c = sess.run([optimizer, cost], feed_dict={x: batch_x,
                                                y: batch_y})
    # Compute average loss
    avg_cost += c / total_batch
# Display logs per epoch step
if epoch % display_step == 0:
    print ("Epoch:", '%04d' % (epoch+1), "cost=", \
        "{:.9f}".format(avg_cost))
print("Optimization Finished!")

# Test model
correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
# Calculate accuracy
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
# To keep sizes compatible with model
print ("Accuracy:", accuracy.eval({x: mnist.test.images, y: mnist.test.labels}))

```

Epoch: 0001 cost= 152.289635962
Epoch: 0002 cost= 39.134648348
...
Epoch: 0015 cost= 0.850344581
Optimization Finished!
Accuracy: 0.9464

Convolution Neural Network MNIST

Convolution Neural Network MNIST

SkFlow

SkFlow

Introduction

In order to make the use of tensorflow simpler to experiment machine learning, google offered a library that stays on top of tensorflow. Skflow make life easier.

Import library

```
import tensorflow.contrib.learn as skflow
from sklearn import datasets, metrics
from sklearn import cross_validation
```

Load dataset

```
iris = datasets.load_iris()
x_train, x_test, y_train, y_test = cross_validation.train_test_split(
    iris.data, iris.target, test_size=0.2, random_state=42)

# Feature columns is required for new versions
feature_columns = skflow.infer_real_valued_columns_from_input(x_train)
```

Linear classifier

```
classifier = skflow.LinearClassifier(feature_columns=feature_columns,
classifier.fit(x_train, y_train, steps=200, batch_size=32)
score = metrics.accuracy_score(y_test, classifier.predict(x_test))
print("Accuracy: %f" % score)
```

Accuracy: 0.966667

Multi layer perceptron

```
classifier = skflow.DNNClassifier(feature_columns=feature_columns, h:
    n_classes=3, model_dir='/tmp/tf/mlp',
classifier.fit(x_train, y_train, steps=200)

score = metrics.accuracy_score(y_test, classifier.predict(x_test))
print("Accuracy: %f" % score)
```

Accuracy: 1.000000

Using Tensorboard

It's much easier to monitor your model with tensorboard through skflow. Just add the parameter "model_dir" to the classifier constructor.

After running this code, type on your server console:

```
tensorboard --logdir=/tmp/tf_examples/test/  
  
classifier = skflow.DNNClassifier(feature_columns=feature_columns, h:  
classifier.fit(x_train, y_train, steps=200)  
score = metrics.accuracy_score(y_test, classifier.predict(x_test))  
print("Accuracy: %f" % score)  
  
Accuracy: 1.000000
```