ALL IMPORTANT CHEAT SHEETS

NEURAL NETWORKS, ARTIFICIAL INTELLIGENCE & DEEP LEARNING

7/1/2020

COMPILED BY ABHISHEK PRASAD

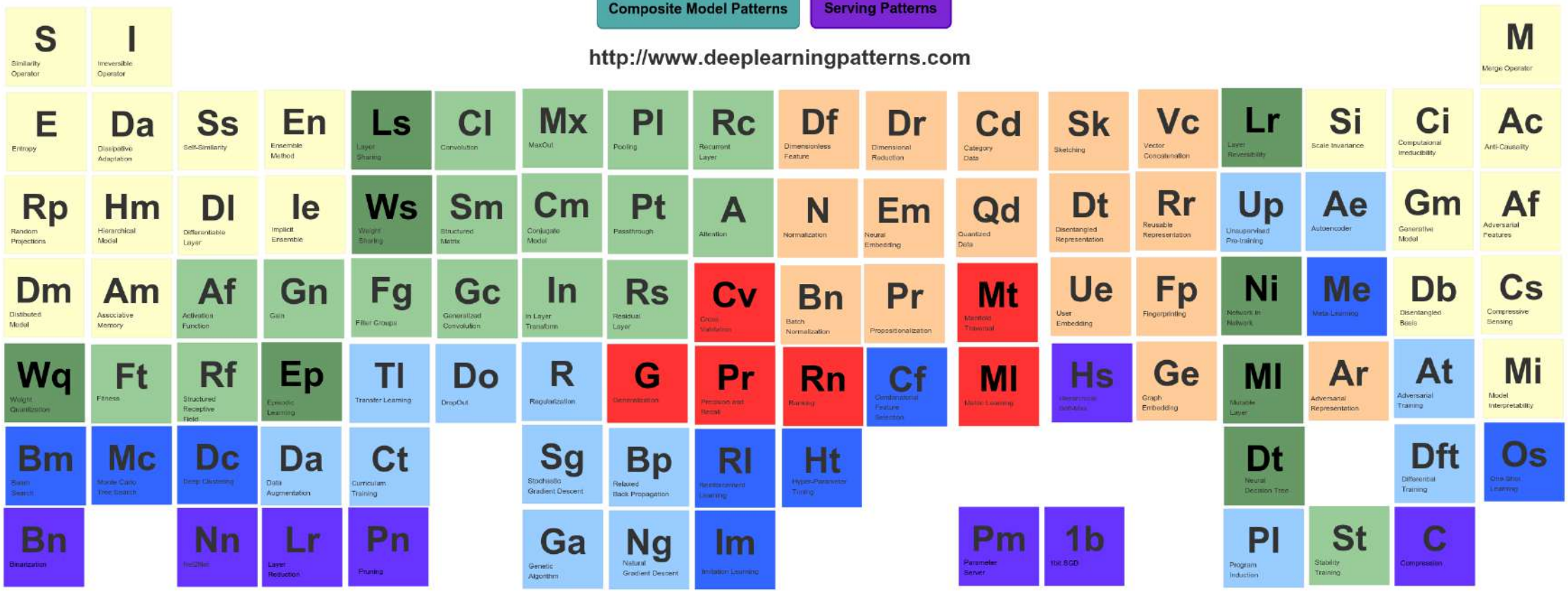
Periodic Table of Deep Learning Patterns

Canonical Patterns | Representation Patterns | Learning Patterns

Model Patterns | Explanation Patterns | Composite Learning Patterns

Composite Model Patterns | Serving Patterns

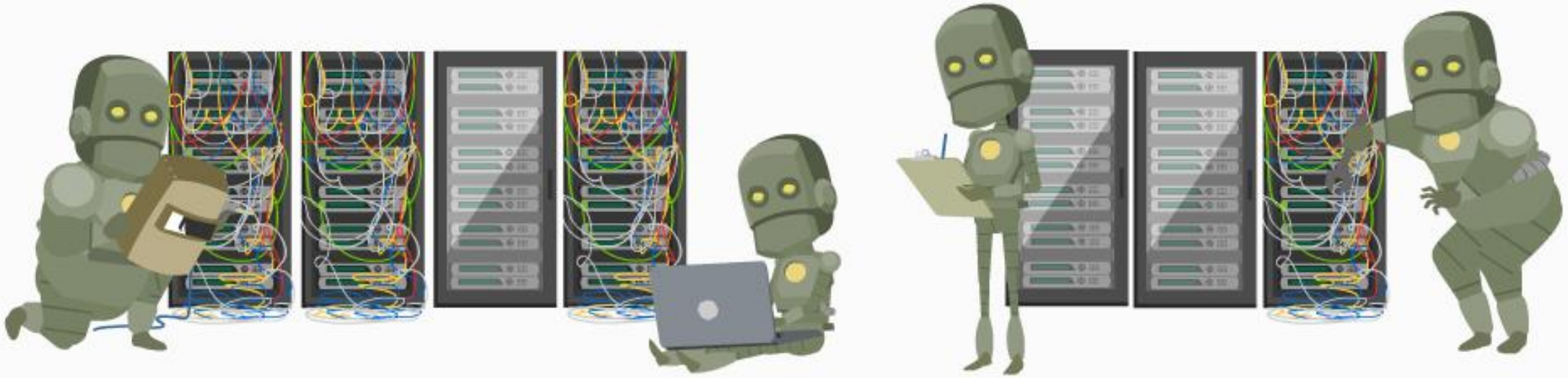
http://www.deeplearningpatterns.com



365 DataScience

NEURAL NETWORKS

CHEAT SHEET



Introduction

Most generally, a machine learning algorithm can be though of as a black box. It takes inputs and gives outputs.

The purpose of this course is to show you how to create this 'black box' and tailor it to your needs.

For example, we may create a model that predicts the weather tomorrow, based on meteorological data about the past few days.

The "black box" in fact is a mathematical model. The machine learning algorithm will follow a kind of trial-and-error method to determine the model that estimates the outputs, given inputs.

Once we have a model, we must **train** it. **Training** is the process through which, the model **learns** how to make sense of input data.

Types of machine learning

Supervised

It is called *supervised* as we provide the algorithm not only with the inputs, but also with the targets (desired outputs). This course focuses on supervised machine learning.

Based on that information the algorithm learns how to produce outputs as close to the **targets** as possible.

The objective function in supervised learning is called **loss function** (also cost or error). We are trying to minimize the loss as the lower the loss function, the higher the accuracy of the model.

Common methods:
- Regression
- Classification

Unsupervised

In *unsupervised* machine learning, the researcher feeds the model with inputs, but **not** with targets. Instead she asks it to find some sort of dependence or underlying logic in the data provided.

For example, you may have the financial data for 100 countries. The model manages to divide (cluster) them into 5 groups. You then examine the 5 clusters and reach the conclusion that the groups are: "Developed", "Developing but overachieving", "Developing but underachieving", "Stagnating", and "Worsening".

The algorithm divided them into 5 groups based on **similarities**, but you didn't know what similarities. It could have divided them by location instead.

Common methods:
- Clustering

Reinforcement

In reinforcement ML, the goal of the algorithm is to maximize its reward. It is inspired by human behavior and the way people change their actions according to incentives, such as getting a reward or avoiding punishment.

The objective function is called a **reward function.** We are trying to maximize the reward function.

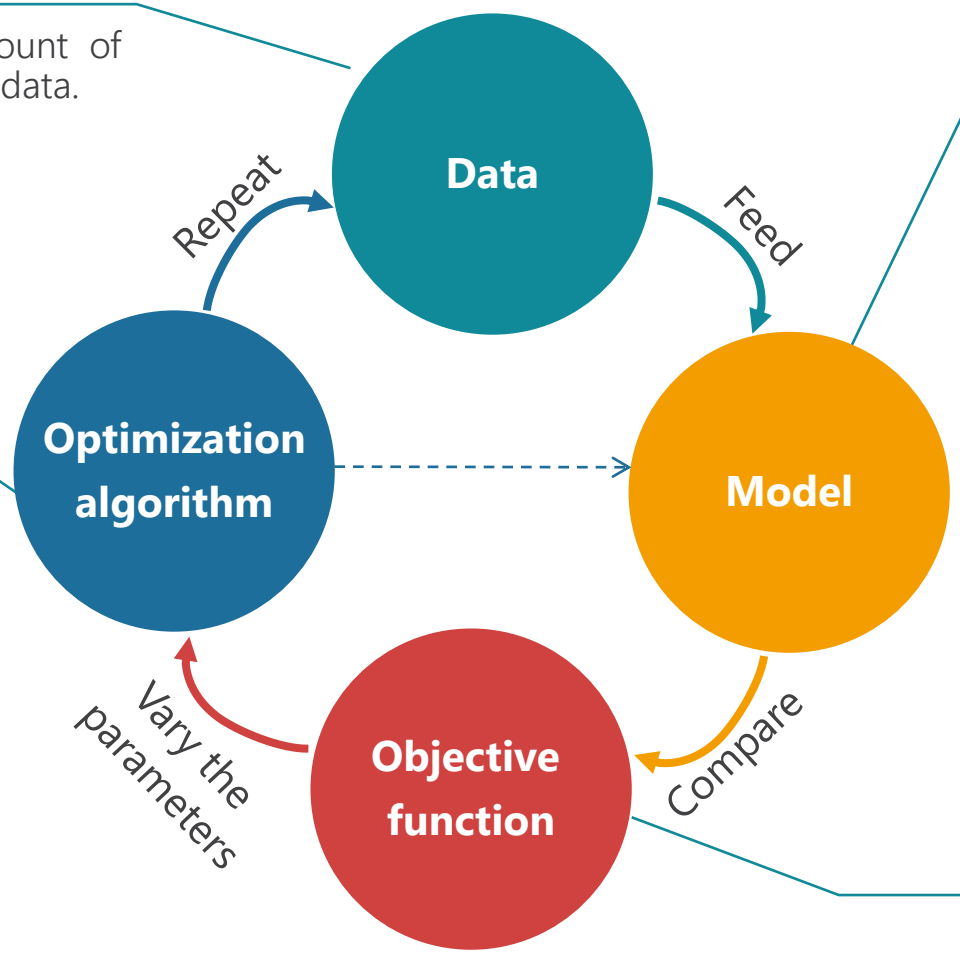
An example is a computer playing Super Mario. The higher the score it achieves, the better it is performing. The score in this case is the objective function.

Common methods:
- Decision process
- Reward system

Building blocks of a machine learning algorithm

The basic logic behind training an algorithm involves four ingredients: data, model, objective function, and optimization algorithm. They are **ingredients**, instead of steps, as the process is iterative.

First, we need to prepare a certain amount of **data** to train on. Usually, we take historical data.



We choose the type of **model**. Roughly speaking, this is some function, which is defined by the *weights* and the *biases*. We feed the input data into the model. Essentially, the idea of the machine learning algorithm is to find the *parameters* for which the model has the highest predictive power.

We achieve the optimization using an **optimization algorithm.** Using the value of the objective function, the optimization algorithm *varies the parameters* of the model. This operation is repeated until we find the values of the parameters, for which the objective function is optimal.

The **objective function** measures the predictive power of our model. Mathematically, the machine learning problem boils down to *optimizing* this function. For example, in the case of loss, we are trying to *minimize* it.

Types of supervised learning

Supervised learning could be split into two subtypes – **regression** and **classification**.

Regression

Regression outputs are continuous numbers.

Examples:

Predicting the EUR/USD exchange rate tomorrow. The output is a number, such as 1.02, 1.53, etc.

Predicting the price of a house, e.g. $234,200 or $512,414.

One of the main properties of the regression outputs is that they are ordered. As 1.02 < 1.53, and 243200 < 512414, we can surely say that one output is bigger than the other.

This distinction proves to be crucial in machine learning as the algorithm somewhat *gets additional information* from the outputs.

Classification

Classification outputs are labels from some sort of class.

Examples:

Classifying photos of animals. The classes are "cats", "dogs", "dolphins", etc.

Predicting conversions in a business context. You can have two classes of customers, e.g."will buy again" and "won't buy again".

In the case of classification, the labels are not ordered and cannot be compared at all. A dog photo is not "more" or "better" than a cat photo (objectively speaking) in the way a house worth $512,414 is "more" (more expensive) than a house worth $234,200.

This distinction proves to be crucial in machine learning, as the different classes are treated on **an equal footing.**

Model

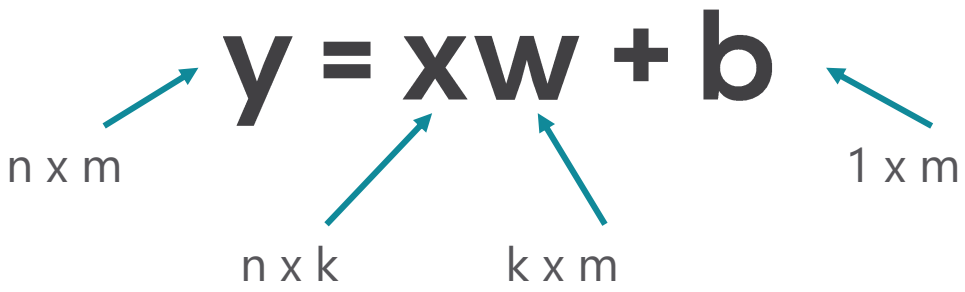
The simplest possible model is **a linear model**. Despite appearing unrealistically simple, in the deep learning context, it is the basis of more complicated models.

There are four elements.

- The input(s), x. That's basically the data that we feed to the model.
- The weight(s), w. Together with the biases, they are called **parameters.** The optimization algorithm will vary the weights and the biases, in order to produce the model that fits the data best.
- The bias(es), b. See weight(s).
- The output(s), y. y is a function of x, determined by w and b.

Each model is determined solely by its parameters (the weights and the biases). That is why we are interested in varying them using the (kind of) trial-and-error method, until we find a model that explains the data sufficiently well.

Model - Continued



Where:

- n is the number of samples (observations)
- m is the number of output variables
- k is the number of input variables

A linear model can represent multidimensional relationships. The shapes of y, x, w, and b are given above (notation is arbitrary).

The simplest linear model, where n = m = k = 1.

y	=	x	w	+	b

The simplest linear model, where m = k = 1, n > 1

y_1	=	$x_1w + b$	=	x_1	w	+	b
y_2		$x_2w + b$		x_2			
...		...		...			
y_n		$x_nw + b$		x_n			

Examples:

27	=	4	6	+	3

Since the weights and biases alone define a model, this example shows **the same model** as above but for many data points.

27	=	4	6	+	3
33		5			
...		...			
-3		-1			

Note that we add the bias to each row, essentially simulating an n x 1 matrix. That's how computers treat such matrices, that's why in machine learning we define the bias in this way.

Model multiple inputs

$$y = xw + b$$

n x m

n x k

k x m

1 x m

Where:

- n is the number of samples (observations)
- m is the number of output variables
- k is the number of input variables

We can extend the model to multiple inputs where n, k > 1, m = 1.

$$\begin{bmatrix} y_1 \\ y_2 \\ \dots \\ \dots \\ y_n \end{bmatrix} = \begin{bmatrix} x_{11}w_1 + x_{12}w_2 + \dots + x_{1k}w_k + b \\ x_{21}w_1 + x_{22}w_2 + \dots + x_{2k}w_k + b \\ \dots \\ \dots \\ x_{n1}w_1 + x_{n2}w_2 + \dots + x_{nk}w_k + b \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1k} \\ x_{21} & x_{22} & \dots & x_{2k} \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ x_{n1} & x_{n2} & \dots & x_{nk} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \dots \\ w_k \end{bmatrix} + \boxed{b}$$

n x 1

n x k

k x 1

1 x 1

Note that we add the bias to each row, essentially simulating an n x 1 matrix. That's how computers treat such matrices, that's why in machine learning we define the bias in this way.

Example on the next page.

Model multiple inputs

$$y = xw + b$$

n x m — n x k — k x m — 1 x m

Where:

- n is the number of samples (observations)
- m is the number of output variables
- k is the number of input variables

We can extend the model to multiple inputs where n, k > 1 m = 1. An example.

y_1	=	9 x (-2) + 12 x 5 +…+ 13 x (-1) + 4	=	9	12	…	13		-2	+	4
y_2		10 x (-2) + 6 x 5 +…+ 2 x (-1) + 4		10	6	…	2		5		
…		…		…	…	…	…		…		
…		…		…	…	…	…		-1		
y_n		7 x (-2) + 7 x 5 +…+ 1 x (-1) + 4		7	7	…	1				
n x 1				n x k					k x 1		1 x 1

Note that we add the bias to each row, essentially simulating an n x 1 matrix. That's how computers treat such matrices, that's why in machine learning we define the bias in this way.

Model multiple inputs and multiple outputs

$$y = xw + b$$

n x m — n x k — k x m — 1 x m

Where:

- n is the number of samples (observations)
- m is the number of output variables
- k is the number of input variables

We can extend the model to multiple inputs where n, k, m > 1.

$$\begin{bmatrix} y_{11} & \cdots & y_{1m} \\ y_{21} & \cdots & y_{2m} \\ \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots \\ y_{n1} & \cdots & y_{nm} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1k} \\ x_{21} & x_{22} & \cdots & x_{2k} \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ x_{n1} & x_{n2} & \cdots & x_{nk} \end{bmatrix} \begin{bmatrix} w_{11} & \cdots & w_{1m} \\ w_{21} & \cdots & w_{2m} \\ \cdots & \cdots & \cdots \\ w_{k1} & \cdots & w_{km} \end{bmatrix} + \begin{bmatrix} b_1 & \cdots & b_m \end{bmatrix}$$

n x m — n x k — k x m — 1 x m

Note that we add the bias to each row, essentially simulating an n x m matrix. That's how computers treat such matrices, that's why in machine learning we define the bias in this way.

Example on the next page.

Model multiple inputs and multiple outputs

$$y = xw + b$$

n x m — n x k — k x m — 1 x m

Where:

- n is the number of samples (observations)
- m is the number of output variables
- k is the number of input variables

We can extend the model to multiple inputs where n, k, m > 1.

$$\begin{bmatrix} y_{11} & \cdots & y_{1m} \\ y_{21} & \cdots & y_{2m} \\ \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots \\ y_{n1} & \cdots & y_{nm} \end{bmatrix} = \begin{bmatrix} x_{11}w_{11} + x_{12}w_{21} + \ldots + x_{1k}w_{k1} + b_1 & \cdots & x_{11}w_{1m} + x_{12}w_{2m} + \ldots + x_{1k}w_{km} + b_m \\ x_{21}w_{11} + x_{22}w_{21} + \ldots + x_{2k}w_{k1} + b_1 & \cdots & x_{21}w_{1m} + x_{22}w_{2m} + \ldots + x_{2k}w_{km} + b_m \\ \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots \\ x_{n1}w_{11} + x_{n2}w_{21} + \ldots + x_{nk}w_{k1} + b_1 & \cdots & x_{n1}w_{1m} + x_{n2}w_{2m} + \ldots + x_{nk}w_{km} + b_m \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1k} \\ x_{21} & x_{22} & \cdots & x_{2k} \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ x_{n1} & x_{n2} & \cdots & x_{nk} \end{bmatrix} \begin{bmatrix} w_{11} & \cdots & w_{1m} \\ w_{21} & \cdots & w_{2m} \\ \cdots & \cdots & \cdots \\ w_{k1} & \cdots & w_{km} \end{bmatrix} + \begin{bmatrix} b_1 & \cdots & b_m \end{bmatrix}$$

n x m — n x k — k x m — 1 x m

Note that we add the bias to each row, essentially simulating an n x m matrix. That's how computers treat such matrices, that's why in machine learning we define the bias in this way.

Objective function

The objective function is a measure of how well our model's outputs match the targets.

The **targets** are the "correct values" which we aim at. In the cats and dogs example, the targets were the "labels" we assigned to each photo (either "cat" or "dog").

Objective functions can be split into two types: **loss** (supervised learning) and **reward** (reinforcement learning). Our focus is supervised learning.

Common loss functions

L2-norm loss
/regression/

$$\sum_i (y_i - t_i)^2$$

The L2-norm of a vector, **a**, (Euclidean length) is given by

$$\|a\| = \sqrt{a^T \cdot a} = \sqrt{a_1^2 + \cdots + a_n^2}$$

The main rationale is that the L2-norm loss is basically the distance from the origin (0). So, the closer to the origin is the difference of the outputs and the targets, the lower the loss, and the better the prediction.

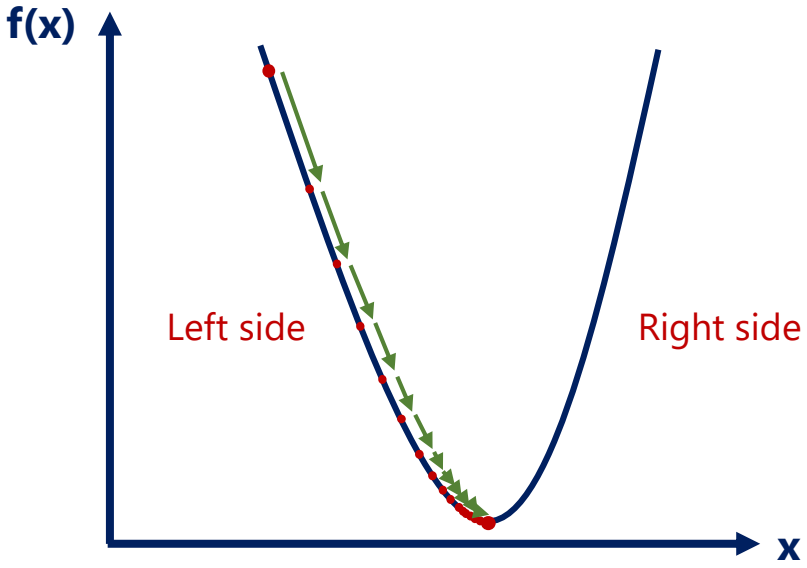
Cross-entropy loss
/classification/

$$-\sum_i \boldsymbol{t}_i \ln \boldsymbol{y}_i$$

The cross-entropy loss is mainly used for classification. Entropy comes from information theory, and measures how much information one is missing to answer a question. Cross-entropy (used in ML) works with probabilities – one is our opinion, the other – the true probability (the probability of a target to be correct is 1 by definition). If the cross-entropy is 0, then we are **not missing any information** and have a perfect model.

Gradient descent

The last ingredient is the optimization algorithm. The most commonly used one is the gradient descent. The main point is that we can find the minimum of a function by applying the rule: $x_{i+1} = x_i - \eta f'(x_i)$, where η is a small enough positive number. In machine learning, η, is called the learning rate. The rationale is that the first derivative at x_i , $f'(x_i)$ shows the slope of the function at x_i.



If the first derivative of f(x) at x_i , $f'(x_i)$, is negative, then we are on the left side of the parabola (as shown in the figure). Subtracting a negative number from x_i (as η is positive), will result in x_{i+1}, that is bigger than x_i. This would cause our next *trial* to be on the right; thus, closer to the sought minimum.

Alternatively, if the first derivative is positive, then we are on the right side of the parabola. Subtracting a positive number from x_i will result in a lower number, so our next trial will be to the left (again closer to the minimum).

So, either way, using this rule, we are approaching the minimum. When the first derivative is 0, we have reached the minimum. Of course, our update rule won't update anymore ($x_{i+1} = x_i - 0$).

The learning rate η, must be low enough so we don't oscillate (bounce around without reaching the minimum) and big enough, so we reach it in rational time.

In machine learning, f(x) is the **loss function**, which we are trying to minimize.

The variables that we are varying until we find the minimum are the **weights and the biases.** The proper update rules are:

$$\boldsymbol{w}_{i+1} = \boldsymbol{w}_i - \eta \nabla_{\mathbf{w}} L(\boldsymbol{w}_i) = \boldsymbol{w}_i - \eta \sum_i \boldsymbol{x}_i \delta_i \quad \text{and} \quad b_{i+1} = b_i - \eta \nabla_b L(b_i) = b_i - \eta \sum_i \delta_i$$

Gradient descent. Multivariate derivation

The multivariate generalization of the gradient descent concept: $x_{i+1} = x_i - \eta f'(x_i)$ is given by: $\boldsymbol{w}_{i+1} = \boldsymbol{w}_i - \eta \nabla_{\mathbf{w}} L(\boldsymbol{w}_i)$ (using this example, as we will need it). In this new equation, w is a matrix and we are interested in the gradient of w, w.r.t. the loss function. As promised in the lecture, we will show the derivation of the gradient descent formulas for the L2-norm loss divided by 2.

Model: $y = xw + b$

Loss: $L = \frac{1}{2}\sum_i (y_i - t_i)^2$

Update rule: $\boldsymbol{w}_{i+1} = \boldsymbol{w}_i - \eta \nabla_{\mathbf{w}} L(\boldsymbol{w}_i)$

(opt. algorithm) $b_{i+1} = b_i - \eta \nabla_b L(b_i)$

Analogically, we find the update rule for the biases.

Please note that the division by 2 that we performed does not change the nature of the loss.

ANY function that holds the basic property of being higher for worse results and lower for better results can be a loss function.

$$\nabla_{\mathbf{w}} L = \nabla_{\mathbf{w}} \frac{1}{2} \sum_i (y_i - t_i)^2 =$$

$$= \nabla_{\mathbf{w}} \frac{1}{2} \sum_i \big((\boldsymbol{x}_i \boldsymbol{w} + b) - t_i\big)^2 =$$

$$= \sum_i \nabla_{\mathbf{w}} \frac{1}{2} (\boldsymbol{x}_i \boldsymbol{w} + b - t_i)^2 =$$

$$= \sum_i \boldsymbol{x}_i (\boldsymbol{x}_i \boldsymbol{w} + b - t_i) =$$

$$= \sum_i \boldsymbol{x}_i (y_i - t_i) \equiv$$

$$\equiv \sum_i \boldsymbol{x}_i \delta_i$$

Linear Vector Spaces:

Definition: A linear vector space, X, is a set of elements (vectors) defined over a scalar field, F, that satisfies the following conditions:

1) if $x \in X$ and $y \in X$ then $x+y \in X$. **2)** $x+y=y+x$ **3)** $(x+y)+z=x+(y+z)$

4) There is a unique vector $0 \in X$, such that $x+0=x$ for all $x \in X$.

5) For each vector $x \in X$ there is a unique vector in X, to be called $(-x)$, such that $x+(-x)=0$. **6)** multiplication, for all scalars $a \in F$, and all vectors $x \in X$,

7) For any $x \in X$, $1x=x$ (for scalar 1).

8) For any two scalars $a \in F$ and $b \in F$ and any $x \in X$, $a(bx)=(ab)x$.

9) $(a+b)x=ax+bx$. **10)** $a(x+y)=ax+ay$.

Linear Independence:

Consider n vectors $\{x_1, x_2, .., x_n\}$. If there exists n scalars $a_1, a_2, \ldots, a_n$, at least one of which is nonzero, such that $a_1x_1 + a_2x_2 + \ldots + a_nx_n = 0$, then the $\{x_i\}$ are linearly dependent.

Spanning a Space:

Let X be a linear vector space and let $\{u_1, u_2, \ldots, u_n\}$ be a subset of vectors in X. This subset spans X if and only if for every vector $x \in X$ there exist scalars $x_1, x_2, \ldots, x_n$ such that $x = x_1u_1+x_2u_2+\ldots+x_nu_n$.

Inner Product:

(x, y) for any scalar function of x and y.

1. $(x,y) = (y,x)$ 2. $(x, ay_1 + by_2) = a(x,y_1) + b(x,y_2)$
3. $(x,x) \geq 0$, where equality holds iff x is the zero vector.

Norm:

Ascalar function $\|x\|$ is called a norm if it satisfies:

1. $\|x\| \geq 0$ 2. $\|x\| = 0$ if and only if $x = 0$.
3. $\|ax\| = |a|\|x\|$ 4. $\|x + y\| \leq \|x\| + \|y\|$

Angle:

The angle θ bet. 2 vectors x and y is defined by $\cos\theta = \frac{(x,y)}{\|x\|\,\|y\|}$

Orthogonality:

2 vectors $x, y \in X$ are said to be orthogonal if $(x,y) = 0$.

Gram Schmidt Orthogonalization:

Assume that we have n independent vectors $y_1, y_2, \ldots, y_n$. From these vectors we will obtain n orthogonal vectors $v_1, v_2, \ldots, v_n$.

$$v_1 = y_1, \quad v_k = y_k - \sum_{i=1}^{k-1} \frac{(v_i, y_k)}{(v_i, v_i)} v_i,$$

$$where\ \frac{(v_i, y_k)}{(v_i, v_i)} v_i\ is\ the\ projection\ of\ y_k\ on\ v_i$$

Vector Expansions:

$$x = \sum_{i=1}^{n} x_i v_i = x_1v_1 + x_2v_2 + \cdots + x_nv_n,$$

$$for\ orthogonal\ vectors, x_j = \frac{(v_j, x)}{(v_j, v_j)}$$

Reciprocal Basis Vectors:

$$(r_i, v_j) = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases}, \quad x_j = (r_j, x)$$

To compute the reciprocal basis vectors: set $\mathbf{B} = [v_1\ v_2\ \ldots\ v_n]$,

$\mathbf{R} = [r_1\ r_2 \ldots\ r_n]$, $\mathbf{R}^T = \mathbf{B}^{-1}$ In matrix form: $\mathbf{x}^v = \mathbf{B}^{-1}\mathbf{x}^s$

Transformations:

A *transformation* consists of three parts:

domain: $X = \{x_i\}$, range: $Y = \{y_i\}$, and a rule relating each $x_i \in X$ to an element $y_i \in Y$.

Linear Transformations:

transformation A is *linear* if:

1. for all $x_1, x_2 \in X$, $A(x_1+x_2) = A(x_1) + A(x_2)$
2. for all $x \in X, a \in R$, $A(ax) = aA(x)$

Matrix Representations:

Let $\{v_1, v_2, \ldots, v_n\}$ be a basis for vector space X, and let $\{u_1, u_2, \ldots, u_n\}$ be a basis for vector space Y. Let A be a linear transformation with domain X and range Y: $A(x) = y$

The coefficients of the matrix representation are obtained from

$$A(v_j) = \sum_{i=1}^{m} a_{ij}u_i$$

Change of Basis: $\mathbf{B}_t = [\mathbf{t}_1\ \mathbf{t}_2\ \ldots \mathbf{t}_n]$, $\mathbf{B}_w = [\mathbf{w}_1\ \mathbf{w}_2\ \ldots \mathbf{w}_n]$

$$\mathbf{A}' = [\mathbf{B}_w^{-1}\mathbf{A}\mathbf{B}_t]$$

Eigenvalues & Eigenvectors: $\mathbf{Az} = \lambda\mathbf{z}$, $|[\mathbf{A} - \lambda\mathbf{I}]| = 0$

Diagonalization: $\mathbf{B} = [\mathbf{z}_1\ \mathbf{z}_2\ \ldots \mathbf{z}_n]$,

where $\{\mathbf{z}, \mathbf{z}_2, \ldots, \mathbf{z}_n\}$ are the eigenvectors of a square matrix A,

$$[\mathbf{B}^{-1}\mathbf{A}\mathbf{B}] = \mathrm{diag}([\lambda_1\ \lambda_2 \ldots \lambda_n])$$

Perceptron Architecture:

$\mathbf{a} = hardlim(\mathbf{Wp} + \mathbf{b})$, $\mathbf{W} = [{}_1\mathbf{w}^T\ {}_2\mathbf{w}^T\ \ldots\ {}_S\mathbf{w}^T]^T$,

$$a_i = hardlim(n_i) = hardlim({}_i\mathbf{w}^T\,\mathrm{p} + b_i)$$

Decision Boundary:

${}_i\mathbf{w}^T\,\mathrm{p} + b_i = 0$

The decision boundary is always orthogonal to the weight vector.

Single-layer perceptrons can only classify linearly separable vectors.

Perceptron Learning Rule

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{ep}^T, \mathbf{b}^{new} = \mathbf{b}^{old} + \mathbf{e},$$

$$where\ \ \mathbf{e} = \mathbf{t} - \mathbf{a}$$

Hebb's Postulate:

"When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased."

Linear Associator:

$\mathbf{a} = purelin(\mathbf{Wp})$

The Hebb Rule:

Supervised Form: $w_{ij}^{new} = w_{ij}^{old} + t_{qi}P_{qi}$

$$\mathbf{W} = \boldsymbol{t}_1\boldsymbol{P}_1^T + \boldsymbol{t}_2\boldsymbol{P}_2^T + \cdots + \boldsymbol{t}_Q\boldsymbol{P}_Q^T$$

$$\mathbf{W} = [\mathbf{t}_1\ \mathbf{t}_2 \ldots \mathbf{t}_Q]\begin{bmatrix}\mathbf{p}_1^T \\ \mathbf{p}_2^T \\ \vdots \\ \mathbf{p}_Q^T\end{bmatrix} = \mathbf{TP}^T$$

Pseudoinverse Rule:

$\mathbf{W} = \mathbf{TP}^+$

When the number, R, of rows of $\mathbf{P}$ is greater than the num ber of columns, Q, of $\mathbf{P}$ and the columns of $\mathbf{P}$ are independent, then the pseudoinverse can be computed by $\mathbf{P}^+ = (\mathbf{P}^T\mathrm{P})^{-1}\mathbf{P}^T$

Variations of Hebbian Learning:

Filtered Learning (Ch.14): $\mathbf{W}^{new} = (1-\gamma)\mathbf{W}^{old} + \alpha\mathbf{t}_q\mathbf{p}_q^T$

Delta Rule (Ch.10): $\mathbf{W}^{new} = \mathbf{W}^{old} + \alpha(\mathbf{t}_q - \mathbf{a}_q)\mathbf{p}_q^T$

Unsupervised Hebb (Ch.13): $\mathbf{W}^{new} = \mathbf{W}^{old} + \alpha\mathbf{a}_q\mathbf{p}_q^T$

Taylor:

$F(\mathbf{x}) = F(\mathbf{x}^*) + \nabla F(\mathbf{x})^T|_{\mathbf{x}=\mathbf{x}^*}(\mathbf{x}-\mathbf{x}^*) + \frac{1}{2}(\mathbf{x}-\mathbf{x}^*)\nabla^2 F(\mathbf{x})^T|_{\mathbf{x}=\mathbf{x}^*}(\mathbf{x}-\mathbf{x}^*) + \cdots$

Grad

$$\nabla F(\mathbf{x}) = \left[\frac{\partial}{\partial x_1}F(\mathbf{x})\ \ \frac{\partial}{\partial x_2}F(\mathbf{x})\ \ \ldots \frac{\partial}{\partial x_n}F(\mathbf{x})\right]^T$$

Hessian:

$$\nabla^2 F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1^2}F(\mathbf{x}) & \frac{\partial}{\partial x_1 \partial x_2}F(\mathbf{x}) \ldots & \frac{\partial}{\partial x_1 \partial x_n}F(\mathbf{x}) \\ \frac{\partial}{\partial x_2 \partial x_1}F(\mathbf{x}) & \frac{\partial}{\partial x_2^2}F(\mathbf{x}) \ldots & \frac{\partial}{\partial x_2 \partial x_n}F(\mathbf{x}) \\ \vdots & \vdots & \vdots \\ \frac{\partial}{\partial x_n \partial x_1}F(\mathbf{x}) & \frac{\partial}{\partial x_n \partial x_2}F(\mathbf{x}) \ldots & \frac{\partial}{\partial x_n^2}F(\mathbf{x}) \end{bmatrix}$$

Directional Derivatives:

1st Dir.Der.: $\frac{\mathbf{p}^T\nabla F(\mathbf{x})}{\|\mathbf{p}\|}$, **2nd Dir.Der.:** $\frac{\mathbf{p}^T\nabla^2 F(\mathbf{x})\mathbf{p}}{\|\mathbf{p}\|^2}$

Minima:

Strong Minimum: if a scalar $\delta > 0$ exists, such that $F(x) < F(x+\Delta x)$ for all Δx such that $\delta > \|\Delta x\| > 0$.

Global Minimum: if $F(x) < F(x+\Delta x)$ for all $\Delta x \neq 0$

Weak Minimum: if it is not a strong minimum, and a scalar $\delta > 0$ exists, such that $F(x) \leq F(x+\Delta x)$ for all Δx such that $\delta > \|\Delta x\| > 0$.

Necessary Conditions for Optimality:

1st-Order Condition: $\nabla F(\mathbf{x})|_{\mathbf{x}=\mathbf{x}^*} = 0$ (Stationary Points)

2nd-Order Condition: $\nabla^2 F(\mathbf{x})|_{\mathbf{x}=\mathbf{x}^*} \geq 0$ (Positive Semi-definite Hessian Matrix).

Quadratic fn.:

$F(x) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\,\mathbf{x} + \mathbf{d}^T\mathbf{x} + c$

$\nabla F(x) = \mathbf{Ax} + \mathbf{d}$, $\nabla^2 F(x) = \mathbf{A}$, $\boldsymbol{\lambda_{min}} \leq \frac{\mathbf{p}^T\mathbf{A}\,\mathbf{p}}{\|\mathbf{p}\|^2} \leq \boldsymbol{\lambda_{max}}$

General Minimization Algorithm:

$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$ *or* $\Delta\mathbf{x}_k = (\mathbf{x}_{k+1} - \mathbf{x}_k) = \alpha_k \mathbf{p}_k$

Steepest Descent Algorithm:

$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{g}_k$ **where,** $\mathbf{g}_k = \nabla F(\mathbf{x})|_{\mathbf{x}=\mathbf{x}_k}$

Stable Learning Rate: ($\alpha_k = \alpha$, **constant**) $\alpha < \frac{2}{\lambda_{max}}$

$\{\lambda_1\ \lambda_2, \ldots, \lambda_n\}$ Eigenvalues of Hessian matrix A

Learning Rate to Minimize Along the Line:

$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \overset{is}{\Rightarrow} \alpha_k = -\frac{\mathbf{g}_k^T \mathbf{p}_k}{\mathbf{p}_k^T \mathbf{A} \mathbf{p}_k}$ (For quadratic fn.)

After Minimization Along the Line:

$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \ \Rightarrow\ \mathbf{g}_{k+1}^T \mathbf{p}_k = 0$

ADALINE: $\mathbf{a} = purelin(\mathbf{W}\mathbf{p} + \mathbf{b})$

Mean Square Error: *(for ADALINE it is a quadratic fn.)*

$F(\mathbf{x}) = E[e^2] = E[(t - a)^2] = E[(t - \mathbf{x}^T\mathbf{z})^2]$

$F(\mathbf{x}) = c - 2\mathbf{x}^T\mathbf{h} + \mathbf{x}^T\mathbf{R}\,\mathbf{x}$,

$c = E[t^2]$, $\mathbf{h} = E[t\mathbf{z}]$ *and* $\mathbf{R} = E[\mathbf{z}\mathbf{z}^T] \Rightarrow \mathbf{A} = 2\mathbf{R}, \mathbf{d} = -2\mathbf{h}$

Unique minimum, if it exists, is $\mathbf{x}^* = \mathbf{R}^{-1}\mathbf{h}$,

where $\mathbf{x} = \begin{bmatrix} {}_1\mathbf{w} \\ b \end{bmatrix}$ and $\mathbf{z} = \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix}$

LMS Algorithm: $\mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha\,\mathbf{e}(k)\,\mathbf{p}^T(k)$

$\mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha\,\mathbf{e}(k)$

Convergence Point: $\mathbf{x}^* = \mathbf{R}^{-1}\mathbf{h}$

Stable Learning Rate: $0 < \alpha < 1/\lambda_{max}$ where λ_{max} is the maximum eigenvalue of R

Adaptive Filter ADALINE:

$$a(k) = purelin(\mathbf{W}\mathbf{p}(k) + b) = \sum_{i=1}^{R} \mathbf{w}_{1,i}\, y(k - i + 1) + b$$

Backpropagation Algorithm:

Performance Index:

Mean Square error: $F(\mathbf{x}) = E[\mathbf{e}^T\mathbf{e}] = E[(\mathbf{t} - \mathbf{a})^T(\mathbf{t} - \mathbf{a})]$

Approximate Performance Index: (single sample)

$\hat{F}(x) = \mathbf{e}^T(k)\mathbf{e}(k) = (\mathbf{t}(k) - \mathbf{a}(k))^T(\mathbf{t}(k) - \mathbf{a}(k))$

Sensitivity: $\mathbf{s}^m = \frac{\partial \hat{F}}{\partial \mathbf{n}^m} = \left[\frac{\partial \hat{F}}{\partial n_1^m} \quad \frac{\partial \hat{F}}{\partial n_2^m} \quad \cdots \quad \frac{\partial \hat{F}}{\partial n_{s^m}^m}\right]^T$

Forward Propagation: $\mathbf{a}^0 = \mathbf{p}$,

$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{m+1}\mathbf{a}^m + \mathbf{b}^{m+1})$ *for* $m = 0, 1, \ldots, M-1$

$\mathbf{a} = \mathbf{a}^M$

Backward Propagation: $\mathbf{s}^M = -2\dot{\mathbf{F}}^M(\mathbf{n}^M)(\mathbf{t} - \mathbf{a})$,

$\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T\mathbf{s}^{m+1}$ *for* $m = M-1, \ldots, 2, 1$, where

$\dot{\mathbf{F}}^m(\mathbf{n}^m) = \mathrm{diag}\left(\left[\dot{f}^m(n_1^m) \quad \dot{f}^m(n_2^m) \quad \ldots \quad \dot{f}^m(n_{s^m}^m)\right]\right)$

$$\dot{f}^m(n_j^m) = \frac{\partial f^m(n_j^m)}{\partial n_j^m}$$

Weight Update (Approximate Steepest Descent):

$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha\mathbf{s}^m(\mathbf{a}^{m-1})^T$

$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha\mathbf{s}^m$

*Heuristic Variations of Backpropagation:

Batching: The parameters are updated only after the entire training set has been presented. The gradients calculated for each training example are averaged together to produce a more accurate estimate of the gradient.(If the training set is complete, i.e., covers all possible input/output pairs, then the gradient estimate will be exact.)

Backpropagation with Momentum (MOBP):

$\Delta\mathbf{W}^m(k) = \gamma\Delta\mathbf{W}^m(k-1) - (1-\gamma)\alpha\,\mathbf{s}^m(\mathbf{a}^{m-1})^T$

$\Delta\mathbf{b}^m(k) = \gamma\Delta\mathbf{b}^m(k-1) - (1-\gamma)\alpha\,\mathbf{s}^m$

Variable Learning Rate Backpropagation (VLBP)

1. If the squared error (over the entire training set) increases by more than some set percentage ζ (typically one to five percent) after a weight update, then the weight update is discarded, the learning rate is multiplied by some factor $\rho < 1$, and the momentum coefficient γ (if it is used) is set to zero.

2. If the squared error decreases after a weight update, then the weight update is accepted and the learning rate is multiplied by some factor $\eta > 1$. If γ has been previously set to zero, it is reset to its original value.

3. If the squared error increases by less than ζ, then the weight update is accepted but the learning rate and the momentum coefficient are unchanged.

Association: $\mathbf{a} = hardlim(\mathrm{W}^0\mathrm{P}^0 + \mathbf{W}\mathbf{p} + b)$

An association is a link between the inputs and outputs of a network so that when a stimulus A is presented to the network, it will output a response **B**.

Associative Learning Rules:

Unsupervised Hebb Rule:

$\mathbf{W}(q) = \mathbf{W}(q-1) + \alpha\,\mathbf{a}(q)\mathbf{p}^T(q)$

Hebb with Decay:

$\mathbf{W}(q) = (1-\gamma)\mathbf{W}(q-1) + \alpha\,\mathbf{a}(q)\mathbf{p}^T(q)$

Instar: $\mathbf{a} = hardlim(\mathbf{W}\mathbf{p} + b)$, $\mathbf{a} = hardlim({}_1\mathbf{w}^T\mathbf{p} + b)$

The instar is activated for ${}_1\mathbf{w}^T\mathbf{p} = \|{}_1\mathbf{w}\|\|\mathbf{p}\| cos\theta \geq -b$

where θ is the angle between $\mathbf{p}$ and ${}_1\mathbf{w}$.

Instar Rule:

${}_i\mathbf{w}(q) = {}_i\mathbf{w}(q-1) + \alpha\,a_i(q)(\mathbf{p}(q) - {}_i\mathbf{w}(q-1))$

${}_i\mathbf{w}(q) = (1-\alpha)\,{}_i\mathbf{w}(q-1) + \alpha\,\mathbf{p}(q)$, *if* $(a_i(q) = 1)$

Kohonen Rule:

${}_i\mathbf{w}(q) = {}_i\mathbf{w}(q-1) + \alpha\,(\mathbf{p}(q) - {}_i\mathbf{w}(q-1))$ *for* $i \in X(q)$

Outstar Rule: $\mathbf{a} = satlins(\mathbf{W}p)$

$\mathbf{w}_j(q) = \mathbf{w}_j(q-1) + \alpha\left(\mathbf{a}(q) - \mathbf{w}_j(q-1)\right)\mathrm{p}_j(q)$

Competitive Layer: $\mathbf{a} = compet(\mathbf{W}\mathbf{p}) = compet(\mathbf{n})$

Competitive Learning with the Kohonen Rule:

${}_{i^*}\mathbf{w}(q) = {}_{i^*}\mathbf{w}(q-1) + \alpha\left(\mathbf{p}(q) - {}_{i^*}\mathbf{w}(q-1)\right)$

$= (1-\alpha)\,{}_{i^*}\mathbf{w}(q-1) + \alpha\,\mathbf{p}(q)$

${}_{i^*}\mathbf{w}(q) = {}_{i^*}\mathbf{w}(q-1)$, $i \neq i^*$ where i^* is the winning neuron.

Self-Organizing with the Kohonen Rule:

${}_i\mathbf{w}(q) = {}_i\mathbf{w}(q-1) + \alpha\left(\mathbf{p}(q) - {}_i\mathbf{w}(q-1)\right)$

$= (1-\alpha)\,{}_i\mathbf{w}(q-1) + \alpha\,\mathbf{p}(q)$, $i \in N_{i^*}(d)$

$N_i(d) = \{j, d_{i,j} \leq d\}$

LVQ Network: $(w_{k,i}^2 = 1) \Rightarrow$ subclass i is a part of class k

$n_i^1 = -\|{}_i\mathbf{w}^1 - \mathbf{p}\|$, $\mathbf{a}^1 = compet(\mathbf{n}^1)$, $\mathbf{a}^2 = \mathbf{W}^2\mathbf{a}^1$

LVQ Network Learning with the Kohonen Rule:

${}_{i^*}\mathbf{w}^1(q) = {}_{i^*}\mathbf{w}^1(q-1) + \alpha\left(\mathbf{p}(q) - {}_{i^*}\mathbf{w}^1(q-1)\right)$,

if $a_{k^*}^2 = t_{k^*} = 1$

${}_{i^*}\mathbf{w}^1(q) = {}_{i^*}\mathbf{w}^1(q-1) - \alpha\left(\mathbf{p}(q) - {}_{i^*}\mathbf{w}^1(q-1)\right)$,

if $a_{k^*}^2 = 1 \neq t_{k^*} = 0$

---

hardlim: $a = \begin{cases} 0 & n < 0 \\ 1 & n \geq 0 \end{cases}$, ***hardlims***: $a = \begin{cases} -1 & n < 0 \\ +1 & n \geq 0 \end{cases}$, ***purelin***: $a = n$, ***Logsig***: $a = \frac{1}{1+e^{-n}}$, ***tansig***: $a = \frac{e^n - e^{-n}}{e^n + e^{-n}}$, ***poslin***: $a = \begin{cases} 0 & n < 0 \\ n & n \geq 0 \end{cases}$,

compet: $a = \begin{cases} 1 & \text{neuron with max } n \\ 0 & \text{all other neurons} \end{cases}$, ***satlin***: $a = \begin{cases} 0 & n < 0 \\ n & -1 \leq n \leq 1 \\ 1 & n > 1 \end{cases}$, ***satlins***: $a = \begin{cases} -1 & n < 0 \\ n & -1 \leq n \leq 1 \\ 1 & n > 1 \end{cases}$,

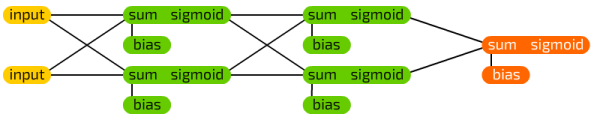
Delay: $a(t) = u(t-1)$, ***Integrator***: $a(t) = \int_0^t u(\tau)d\tau + a(0)$

****HINT:**

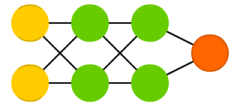
$diag([1\ 2\ 3]) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$

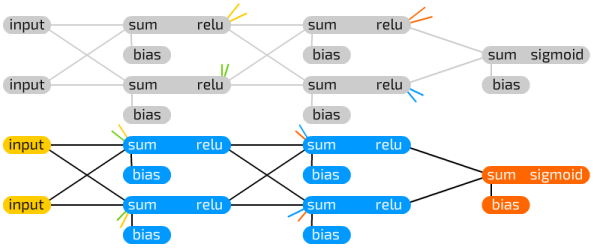
An informative chart to build

Neural Network Graphs

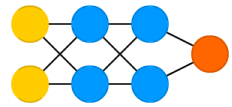


Deep Feed Forward Example

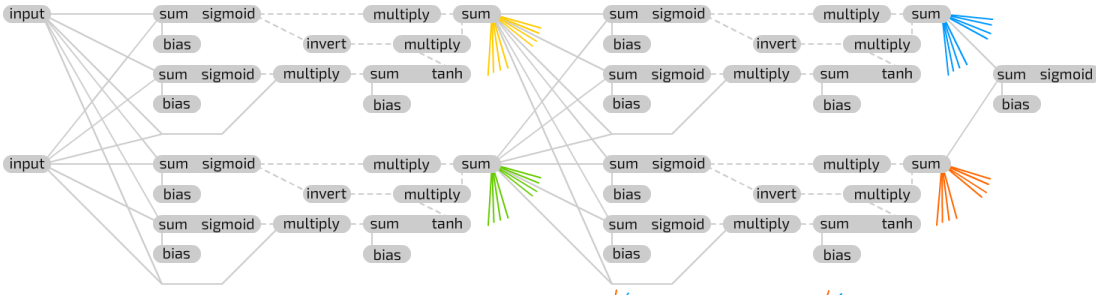




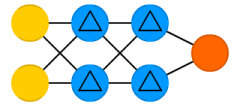
Deep Recurrent Example
(previous iteration)

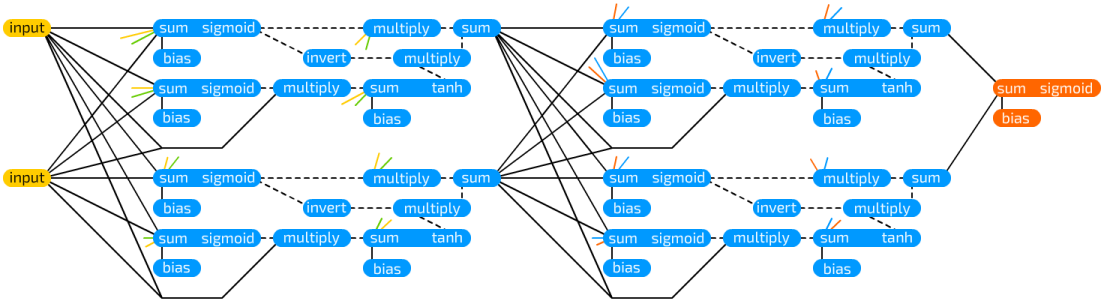


Deep Recurrent Example

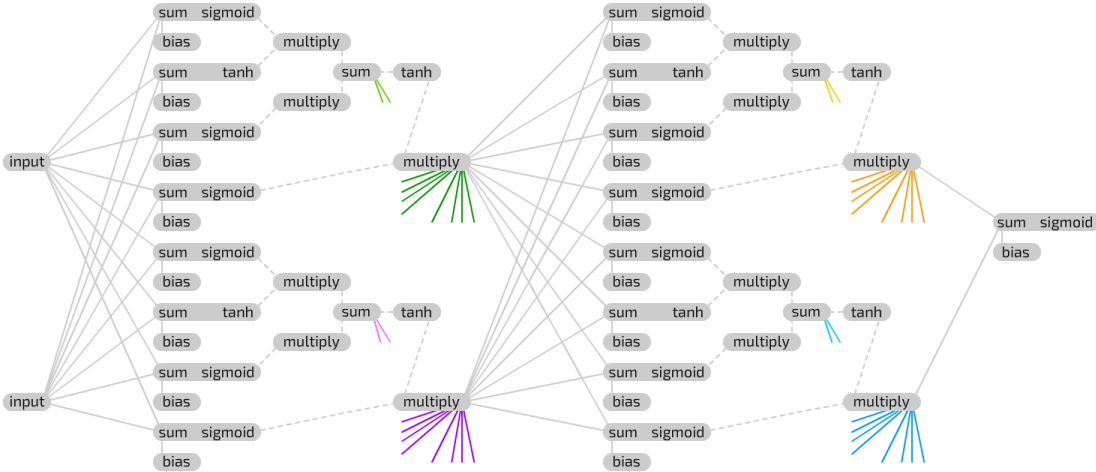


Deep GRU Example
(previous iteration)

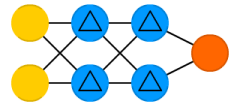


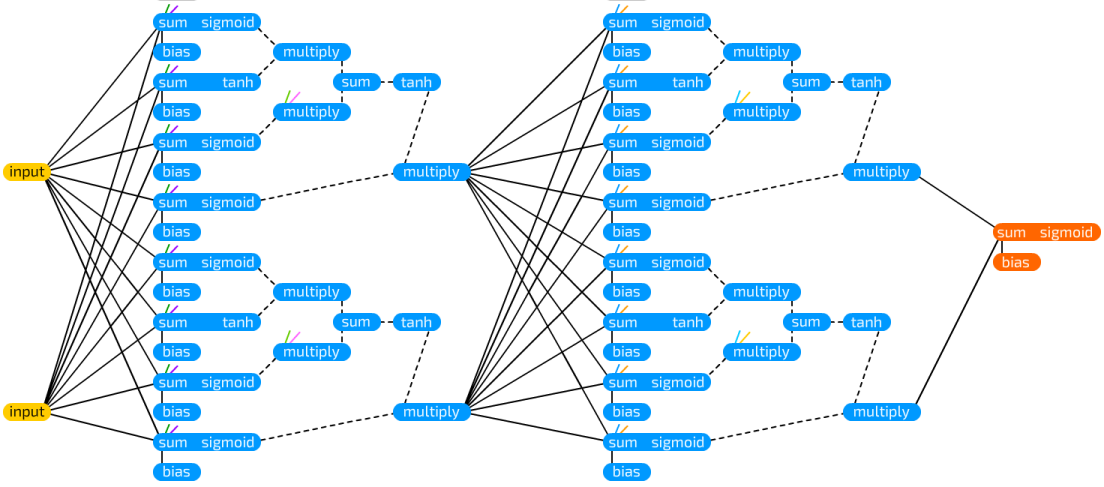


Deep GRU Example



Deep LSTM Example
(previous iteration)





Deep LSTM Example

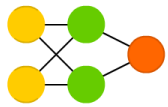
A mostly complete chart of

Neural Networks

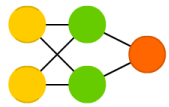
- Backfed Input Cell
- Input Cell
- Noisy Input Cell
- Hidden Cell
- Probablistic Hidden Cell
- Spiking Hidden Cell
- Output Cell
- Match Input Output Cell
- Recurrent Cell
- Memory Cell
- Different Memory Cell
- Kernel
- Convolution or Pool

Perceptron (P)

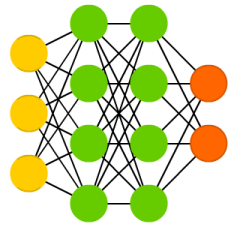
Feed Forward (FF)



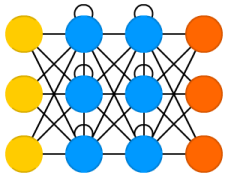
Radial Basis Network (RBF)



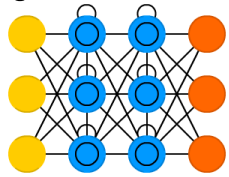
Deep Feed Forward (DFF)



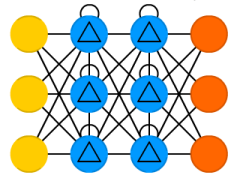
Recurrent Neural Network (RNN)



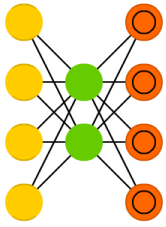
Long / Short Term Memory (LSTM)



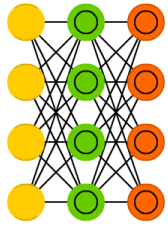
Gated Recurrent Unit (GRU)



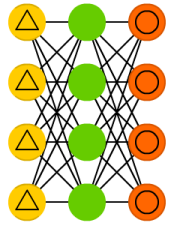
Auto Encoder (AE)



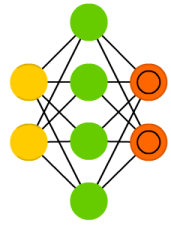
Variational AE (VAE)



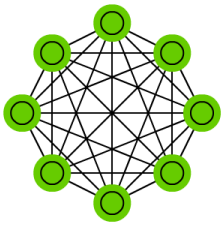
Denoising AE (DAE)



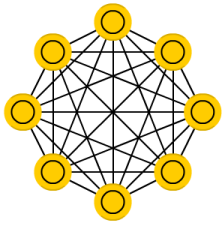
Sparse AE (SAE)



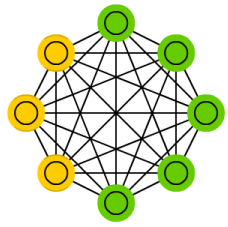
Markov Chain (MC)



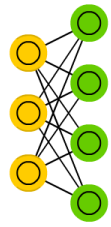
Hopfield Network (HN)



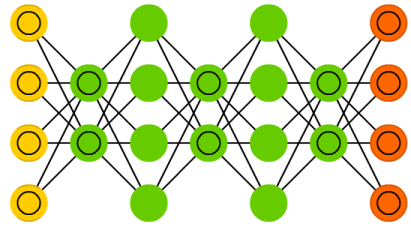
Boltzmann Machine (BM)



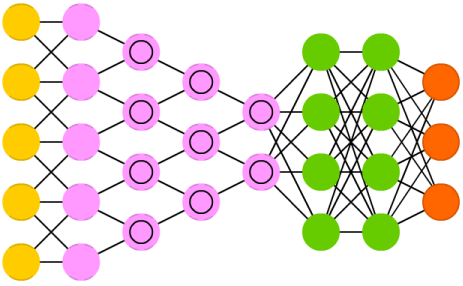
Restricted BM (RBM)



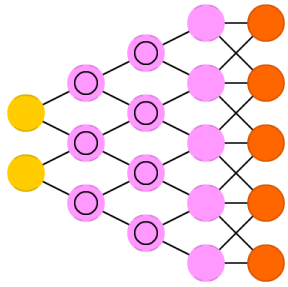
Deep Belief Network (DBN)



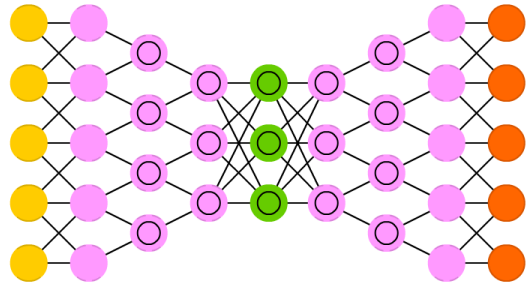
Deep Convolutional Network (DCN)



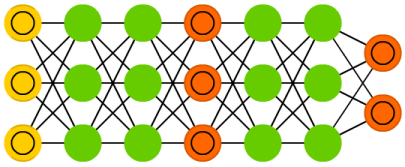
Deconvolutional Network (DN)



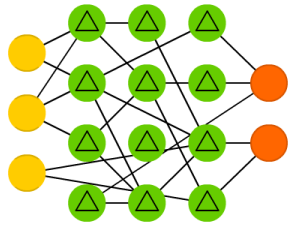
Deep Convolutional Inverse Graphics Network (DCIGN)



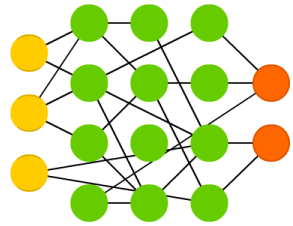
Generative Adversarial Network (GAN)



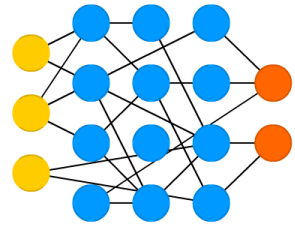
Liquid State Machine (LSM)



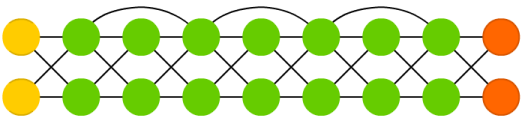
Extreme Learning Machine (ELM)



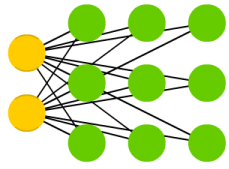
Echo State Network (ESN)



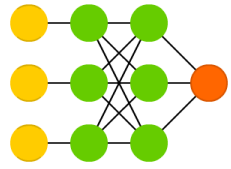
Deep Residual Network (DRN)



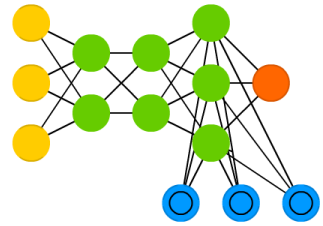
Kohonen Network (KN)



Support Vector Machine (SVM)



Neural Turing Machine (NTM)



Super VIP Cheatsheet: Deep Learning

Afshine Amidi and Shervine Amidi

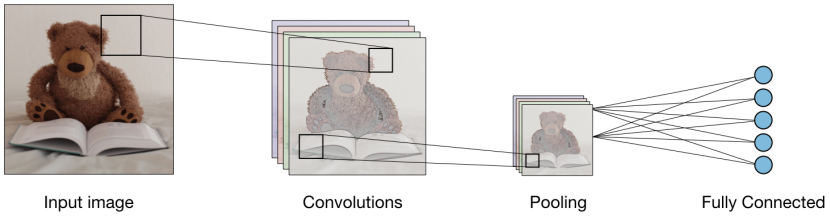
November 25, 2018

Contents

1 Convolutional Neural Networks

1.1 Overview

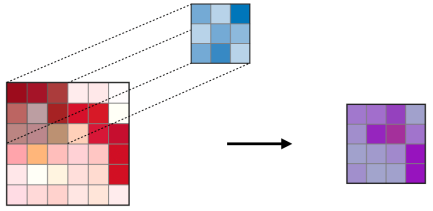
❒ **Architecture of a traditional CNN** – Convolutional neural networks, also known as CNNs, are a specific type of neural networks that are generally composed of the following layers:



The convolution layer and the pooling layer can be fine-tuned with respect to hyperparameters that are described in the next sections.

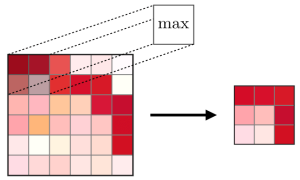
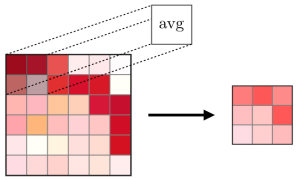
1.2 Types of layer

❒ **Convolutional layer (CONV)** – The convolution layer (CONV) uses filters that perform convolution operations as it is scanning the input I with respect to its dimensions. Its hyperparameters include the filter size F and stride S. The resulting output O is called *feature map* or *activation map*.

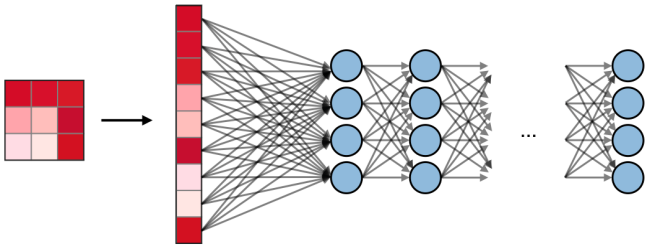


Remark: the convolution step can be generalized to the 1D and 3D cases as well.

❒ **Pooling (POOL)** – The pooling layer (POOL) is a downsampling operation, typically applied after a convolution layer, which does some spatial invariance. In particular, max and average pooling are special kinds of pooling where the maximum and average value is taken, respectively.

	Max pooling	Average pooling
Purpose	Each pooling operation selects the maximum value of the current view	Each pooling operation averages the values of the current view
Illustration		
Comments	- Preserves detected features - Most commonly used	- Downsamples feature map - Used in LeNet

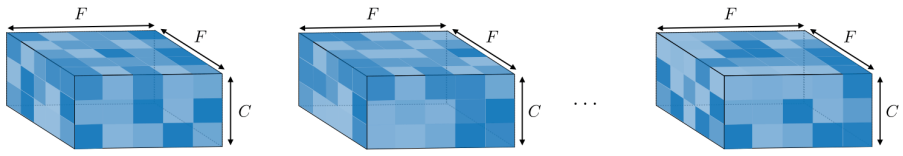
❒ **Fully Connected (FC)** – The fully connected layer (FC) operates on a flattened input where each input is connected to all neurons. If present, FC layers are usually found towards the end of CNN architectures and can be used to optimize objectives such as class scores.



1.3 Filter hyperparameters

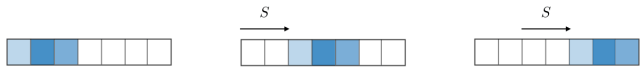
The convolution layer contains filters for which it is important to know the meaning behind its hyperparameters.

❒ **Dimensions of a filter** – A filter of size $F \times F$ applied to an input containing C channels is a $F \times F \times C$ volume that performs convolutions on an input of size $I \times I \times C$ and produces an output feature map (also called activation map) of size $O \times O \times 1$.

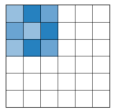
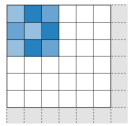
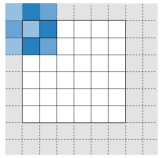


Remark: the application of K filters of size $F \times F$ results in an output feature map of size $O \times O \times K$.

❒ **Stride** – For a convolutional or a pooling operation, the stride S denotes the number of pixels by which the window moves after each operation.



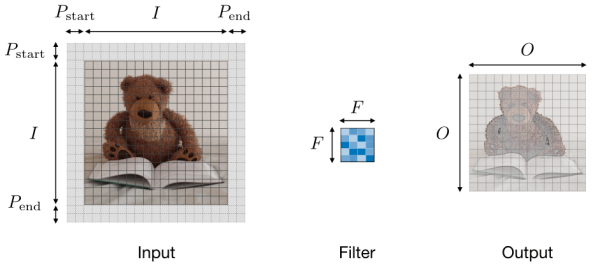
❒ **Zero-padding** – Zero-padding denotes the process of adding P zeroes to each side of the boundaries of the input. This value can either be manually specified or automatically set through one of the three modes detailed below:

	Valid	Same	Full
Value	$P = 0$	$P_{\text{start}} = \left\lfloor \frac{S\lceil \frac{I}{S} \rceil - I + F - S}{2} \right\rfloor$ $P_{\text{end}} = \left\lceil \frac{S\lceil \frac{I}{S} \rceil - I + F - S}{2} \right\rceil$	$P_{\text{start}} \in [\![0, F-1]\!]$ $P_{\text{end}} = F - 1$
Illustration			
Purpose	- No padding - Drops last convolution if dimensions do not match	- Padding such that feature map size has size $\lceil \frac{I}{S} \rceil$ - Output size is mathematically convenient - Also called 'half' padding	- Maximum padding such that end convolutions are applied on the limits of the input - Filter 'sees' the input end-to-end

1.4 Tuning hyperparameters

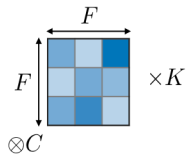
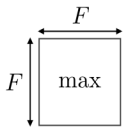
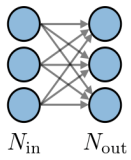
❒ **Parameter compatibility in convolution layer** – By noting I the length of the input volume size, F the length of the filter, P the amount of zero padding, S the stride, then the output size O of the feature map along that dimension is given by:

$$\boxed{O = \frac{I - F + P_{\text{start}} + P_{\text{end}}}{S} + 1}$$



Remark: often times, $P_{start} = P_{end} \triangleq P$, in which case we can replace $P_{start} + P_{end}$ by $2P$ in the formula above.

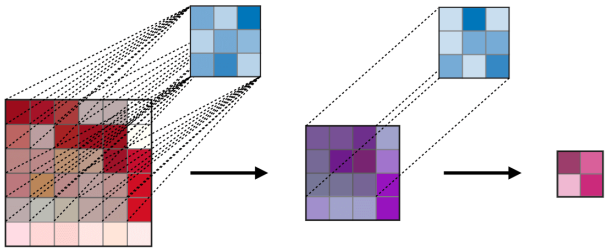
❒ **Understanding the complexity of the model** – In order to assess the complexity of a model, it is often useful to determine the number of parameters that its architecture will have. In a given layer of a convolutional neural network, it is done as follows:

	CONV	POOL	FC
Illustration			
Input size	$I \times I \times C$	$I \times I \times C$	N_{in}
Output size	$O \times O \times K$	$O \times O \times C$	N_{out}
Number of parameters	$(F \times F \times C + 1) \cdot K$	0	$(N_{\text{in}} + 1) \times N_{\text{out}}$
Remarks	- One bias parameter per filter - In most cases, $S < F$ - A common choice for K is $2C$	- Pooling operation done channel-wise - In most cases, $S = F$	- Input is flattened - One bias parameter per neuron - The number of FC neurons is free of structural constraints

❒ **Receptive field** – The receptive field at layer k is the area denoted $R_k \times R_k$ of the input that each pixel of the k-th activation map can 'see'. By calling F_j the filter size of layer j and S_i the stride value of layer i and with the convention $S_0 = 1$, the receptive field at layer k can be computed with the formula:

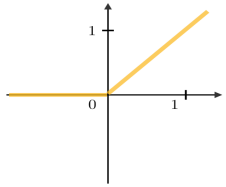
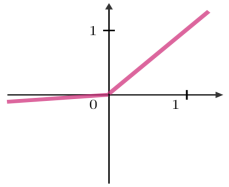
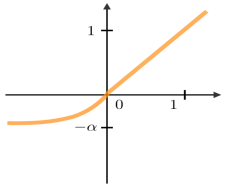
$$R_k = 1 + \sum_{j=1}^{k} (F_j - 1) \prod_{i=0}^{j-1} S_i$$

In the example below, we have $F_1 = F_2 = 3$ and $S_1 = S_2 = 1$, which gives $R_2 = 1+2 \cdot 1+2 \cdot 1 = 5$.



1.5 Commonly used activation functions

❒ **Rectified Linear Unit** – The rectified linear unit layer (ReLU) is an activation function g that is used on all elements of the volume. It aims at introducing non-linearities to the network. Its variants are summarized in the table below:

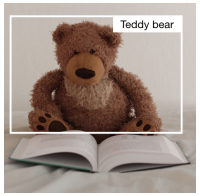
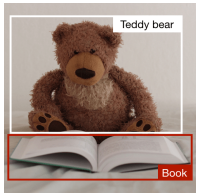
ReLU	Leaky ReLU	ELU
$g(z) = \max(0,z)$	$g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$	$g(z) = \max(\alpha(e^z - 1), z)$ with $\alpha \ll 1$
		
Non-linearity complexities biologically interpretable	Addresses dying ReLU issue for negative values	Differentiable everywhere

❒ **Softmax** – The softmax step can be seen as a generalized logistic function that takes as input a vector of scores $x \in \mathbb{R}^n$ and outputs a vector of output probability $p \in \mathbb{R}^n$ through a softmax function at the end of the architecture. It is defined as follows:

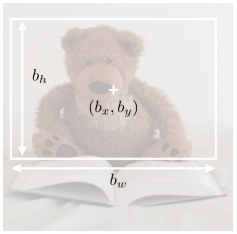
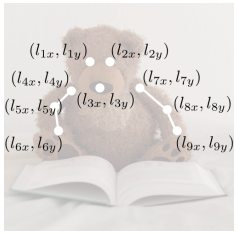
$$p = \begin{pmatrix} p_1 \\ \vdots \\ p_n \end{pmatrix} \quad \text{where} \quad p_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

1.6 Object detection

❒ **Types of models** – There are 3 main types of object recognition algorithms, for which the nature of what is predicted is different. They are described in the table below:

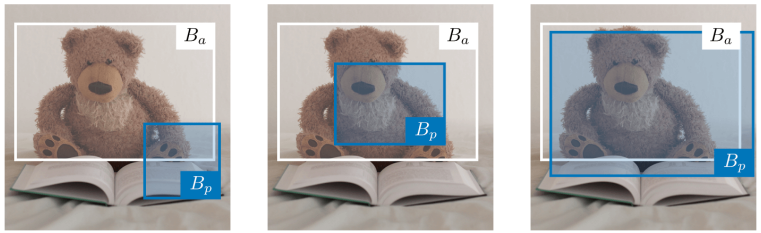
Image classification	Classification w. localization	Detection
		
- Classifies a picture - Predicts probability of object	- Detects object in a picture - Predicts probability of object and where it is located	- Detects up to several objects in a picture - Predicts probabilities of objects and where they are located
Traditional CNN	Simplified YOLO, R-CNN	YOLO, R-CNN

❒ **Detection** – In the context of object detection, different methods are used depending on whether we just want to locate the object or detect a more complex shape in the image. The two main ones are summed up in the table below:

Bounding box detection	Landmark detection
Detects the part of the image where the object is located	- Detects a shape or characteristics of an object (e.g. eyes) - More granular
	
Box of center (b_x,b_y), height b_h and width b_w	Reference points (l_{1x},l_{1y}), ...,(l_{nx},l_{ny})

❒ Intersection over Union – Intersection over Union, also known as IoU, is a function that quantifies how correctly positioned a predicted bounding box B_p is over the actual bounding box B_a. It is defined as:

$$\boxed{\text{IoU}(B_p,B_a) = \frac{B_p \cap B_a}{B_p \cup B_a}}$$



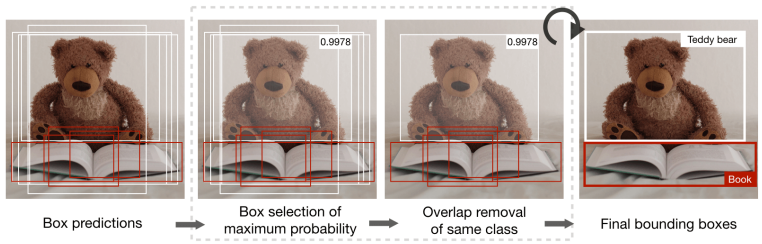
$\text{IoU}(B_p, B_a) = 0.1$ $\text{IoU}(B_p, B_a) = 0.5$ $\text{IoU}(B_p, B_a) = 0.9$

Remark: we always have $IoU \in [0,1]$. By convention, a predicted bounding box B_p is considered as being reasonably good if $IoU(B_p,B_a) \geqslant 0.5$.

❒ Anchor boxes – Anchor boxing is a technique used to predict overlapping bounding boxes. In practice, the network is allowed to predict more than one box simultaneously, where each box prediction is constrained to have a given set of geometrical properties. For instance, the first prediction can potentially be a rectangular box of a given form, while the second will be another rectangular box of a different geometrical form.

❒ Non-max suppression – The non-max suppression technique aims at removing duplicate overlapping bounding boxes of a same object by selecting the most representative ones. After having removed all boxes having a probability prediction lower than 0.6, the following steps are repeated while there are boxes remaining:

- Step 1: Pick the box with the largest prediction probability.
- Step 2: Discard any box having an IoU $\geqslant 0.5$ with the previous box.



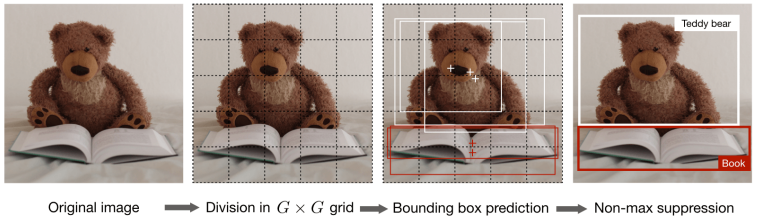
❒ YOLO – You Only Look Once (YOLO) is an object detection algorithm that performs the following steps:

- Step 1: Divide the input image into a $G \times G$ grid.
- Step 2: For each grid cell, run a CNN that predicts y of the following form:

$$\boxed{y = \Big[\underbrace{p_c,b_x,b_y,b_h,b_w,c_1,c_2,...,c_p}_{\text{repeated } k \text{ times}},...\Big]^T \in \mathbb{R}^{G\times G\times k\times(5+p)}}$$

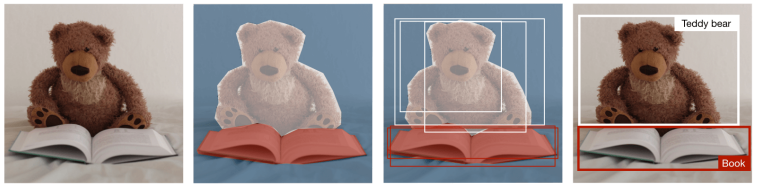
where p_c is the probability of detecting an object, b_x,b_y,b_h,b_w are the properties of the detected bouding box, c_1,...,c_p is a one-hot representation of which of the p classes were detected, and k is the number of anchor boxes.

- Step 3: Run the non-max suppression algorithm to remove any potential duplicate overlapping bounding boxes.



Remark: when $p_c = 0$, then the network does not detect any object. In that case, the corresponding predictions $b_x, ..., c_p$ have to be ignored.

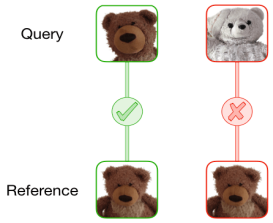
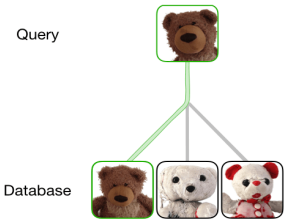
❒ R-CNN – Region with Convolutional Neural Networks (R-CNN) is an object detection algorithm that first segments the image to find potential relevant bounding boxes and then run the detection algorithm to find most probable objects in those bounding boxes.



Remark: although the original algorithm is computationally expensive and slow, newer architectures enabled the algorithm to run faster, such as Fast R-CNN and Faster R-CNN.

1.6.1 Face verification and recognition

❒ **Types of models** – Two main types of model are summed up in table below:

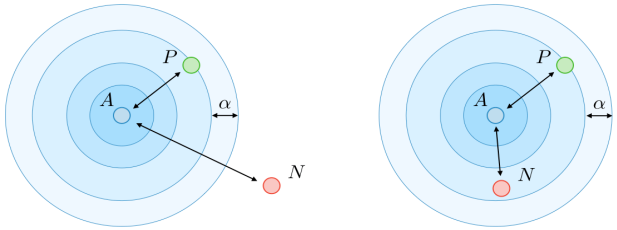
Face verification	Face recognition
- Is this the correct person? - One-to-one lookup	- Is this one of the K persons in the database? - One-to-many lookup
	

❒ **One Shot Learning** – One Shot Learning is a face verification algorithm that uses a limited training set to learn a similarity function that quantifies how different two given images are. The similarity function applied to two images is often noted $d(\text{image 1}, \text{image 2})$.

❒ **Siamese Network** – Siamese Networks aim at learning how to encode images to then quantify how different two images are. For a given input image $x^{(i)}$, the encoded output is often noted as $f(x^{(i)})$.

❒ **Triplet loss** – The triplet loss ℓ is a loss function computed on the embedding representation of a triplet of images A (anchor), P (positive) and N (negative). The anchor and the positive example belong to a same class, while the negative example to another one. By calling $\alpha \in \mathbb{R}^+$ the margin parameter, this loss is defined as follows:

$$\boxed{\ell(A,P,N) = \max\left(d(A,P) - d(A,N) + \alpha, 0\right)}$$



$\ell(A,P,N) = 0$ $\quad$ $\ell(A,P,N) > 0$

1.6.2 Neural style transfer

❒ **Motivation** – The goal of neural style transfer is to generate an image G based on a given content C and a given style S.

Content C $\quad$ Style S $\quad$ Generated image G

❒ **Activation** – In a given layer l, the activation is noted $a^{[l]}$ and is of dimensions $n_H \times n_w \times n_c$

❒ **Content cost function** – The content cost function $J_{\text{content}}(C,G)$ is used to determine how the generated image G differs from the original content image C. It is defined as follows:

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

❒ **Style matrix** – The style matrix $G^{[l]}$ of a given layer l is a Gram matrix where each of its elements $G^{[l]}_{kk'}$ quantifies how correlated the channels k and k' are. It is defined with respect to activations $a^{[l]}$ as follows:

$$\boxed{G^{[l]}_{kk'} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_w^{[l]}} a^{[l]}_{ijk} a^{[l]}_{ijk'}}$$

Remark: the style matrix for the style image and the generated image are noted $G^{[l](S)}$ and $G^{[l](G)}$ respectively.

❒ **Style cost function** – The style cost function $J_{\text{style}}(S,G)$ is used to determine how the generated image G differs from the style S. It is defined as follows:

$$\boxed{J^{[l]}_{\text{style}}(S,G) = \frac{1}{(2n_H n_w n_c)^2}||G^{[l](S)} - G^{[l](G)}||^2_F = \frac{1}{(2n_H n_w n_c)^2} \sum_{k,k'=1}^{n_c} \left(G^{[l](S)}_{kk'} - G^{[l](G)}_{kk'}\right)^2}$$

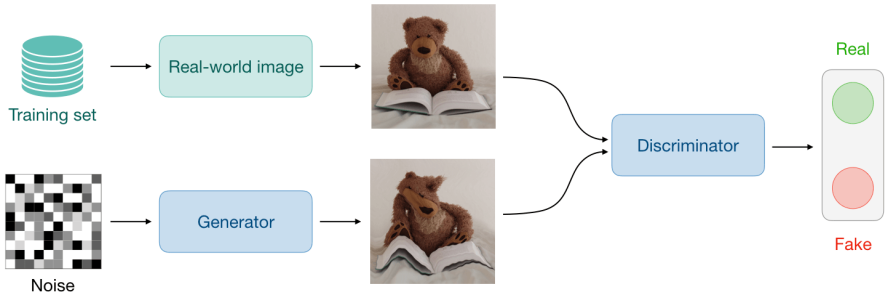
❒ **Overall cost function** – The overall cost function is defined as being a combination of the content and style cost functions, weighted by parameters α,β, as follows:

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

Remark: a higher value of α will make the model care more about the content while a higher value of β will make it care more about the style.

1.6.3 Architectures using computational tricks

❒ **Generative Adversarial Network** – Generative adversarial networks, also known as GANs, are composed of a generative and a discriminative model, where the generative model aims at generating the most truthful output that will be fed into the discriminative which aims at differentiating the generated and true image.



Remark: use cases using variants of GANs include text to image, music generation and synthesis.

❒ ResNet – The Residual Network architecture (also called ResNet) uses residual blocks with a high number of layers meant to decrease the training error. The residual block has the following characterizing equation:

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

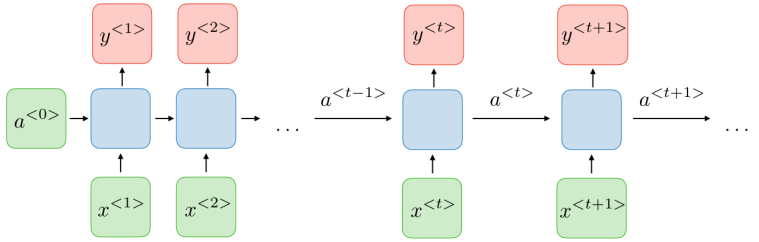
❒ Inception Network – This architecture uses inception modules and aims at giving a try at different convolutions in order to increase its performance. In particular, it uses the 1×1 convolution trick to lower the burden of computation.

⋆ ⋆ ⋆

2 Recurrent Neural Networks

2.1 Overview

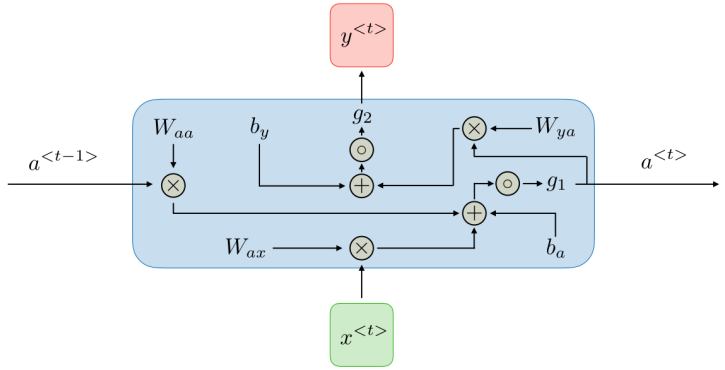
❒ Architecture of a traditional RNN – Recurrent neural networks, also known as RNNs, are a class of neural networks that allow previous outputs to be used as inputs while having hidden states. They are typically as follows:



For each timestep t, the activation $a^{<t>}$ and the output $y^{<t>}$ are expressed as follows:

$$\boxed{a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)} \quad \text{and} \quad \boxed{y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)}$$

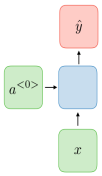
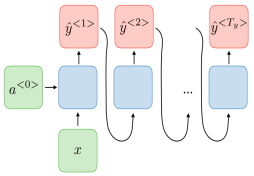
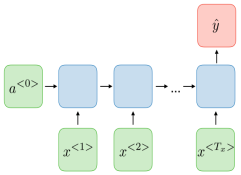
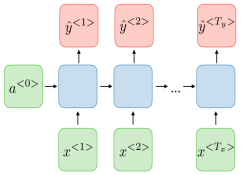
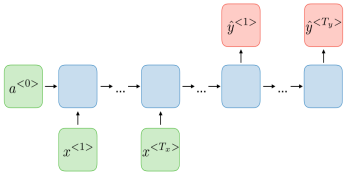
where $W_{ax}, W_{aa}, W_{ya}, b_a, b_y$ are coefficients that are shared temporally and g_1, g_2 activation functions



The pros and cons of a typical RNN architecture are summed up in the table below:

Advantages	Drawbacks
- Possibility of processing input of any length - Model size not increasing with size of input - Computation takes into account historical information - Weights are shared across time	- Computation being slow - Difficulty of accessing information from a long time ago - Cannot consider any future input for the current state

❒ Applications of RNNs – RNN models are mostly used in the fields of natural language processing and speech recognition. The different applications are summed up in the table below:

Type of RNN	Illustration	Example
One-to-one $T_x = T_y = 1$		Traditional neural network
One-to-many $T_x = 1, T_y > 1$		Music generation
Many-to-one $T_x > 1, T_y = 1$		Sentiment classification
Many-to-many $T_x = T_y$		Name entity recognition
Many-to-many $T_x \neq T_y$		Machine translation

❒ **Loss function** – In the case of a recurrent neural network, the loss function $\mathcal{L}$ of all time steps is defined based on the loss at every time step as follows:

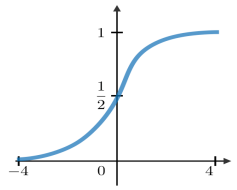
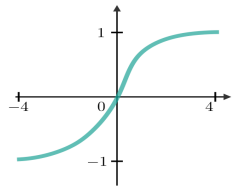
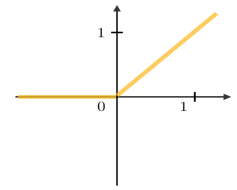
$$\boxed{\mathcal{L}(\widehat{y},y) = \sum_{t=1}^{T_y} \mathcal{L}(\widehat{y}^{<t>},y^{<t>})}$$

❒ **Backpropagation through time** – Backpropagation is done at each point in time. At timestep T, the derivative of the loss $\mathcal{L}$ with respect to weight matrix W is expressed as follows:

$$\boxed{\frac{\partial \mathcal{L}^{(T)}}{\partial W} = \sum_{t=1}^{T} \left.\frac{\partial \mathcal{L}^{(T)}}{\partial W}\right|_{(t)}}$$

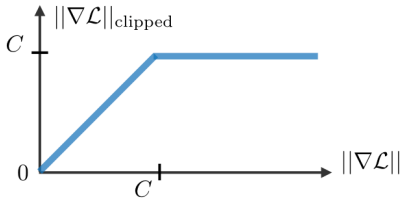
2.2 Handling long term dependencies

❒ **Commonly used activation functions** – The most common activation functions used in RNN modules are described below:

Sigmoid	Tanh	RELU
$g(z) = \frac{1}{1+e^{-z}}$	$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$g(z) = \max(0,z)$
		

❒ **Vanishing/exploding gradient** – The vanishing and exploding gradient phenomena are often encountered in the context of RNNs. The reason why they happen is that it is difficult to capture long term dependencies because of multiplicative gradient that can be exponentially decreasing/increasing with respect to the number of layers.

❒ **Gradient clipping** – It is a technique used to cope with the exploding gradient problem sometimes encountered when performing backpropagation. By capping the maximum value for the gradient, this phenomenon is controlled in practice.



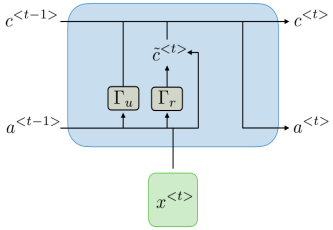
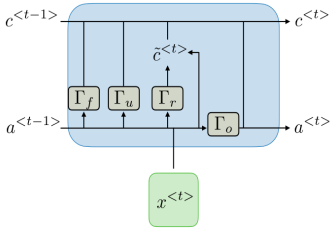
❒ **Types of gates** – In order to remedy the vanishing gradient problem, specific gates are used in some types of RNNs and usually have a well-defined purpose. They are usually noted Γ and are equal to:

$$\boxed{\Gamma = \sigma(Wx^{<t>} + Ua^{<t-1>} + b)}$$

where W, U, b are coefficients specific to the gate and σ is the sigmoid function. The main ones are summed up in the table below:

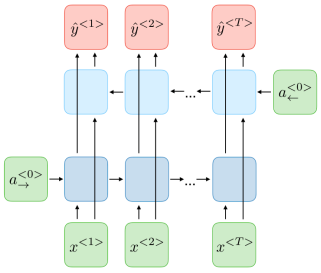
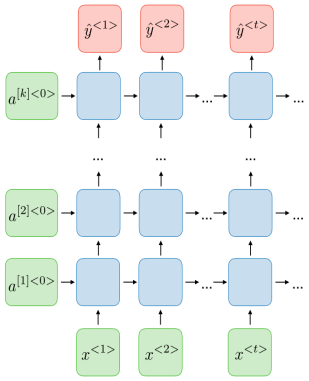
Type of gate	Role	Used in
Update gate Γ_u	How much past should matter now?	GRU, LSTM
Relevance gate Γ_r	Drop previous information?	GRU, LSTM
Forget gate Γ_f	Erase a cell or not?	LSTM
Output gate Γ_o	How much to reveal of a cell?	LSTM

❒ **GRU/LSTM** – Gated Recurrent Unit (GRU) and Long Short-Term Memory units (LSTM) deal with the vanishing gradient problem encountered by traditional RNNs, with LSTM being a generalization of GRU. Below is a table summing up the characterizing equations of each architecture:

	Gated Recurrent Unit (GRU)	**Long Short-Term Memory** (LSTM)
$\tilde{c}^{<t>}$	$\tanh(W_c[\Gamma_r \star a^{<t-1>}, x^{<t>}] + b_c)$	$\tanh(W_c[\Gamma_r \star a^{<t-1>}, x^{<t>}] + b_c)$
$c^{<t>}$	$\Gamma_u \star \tilde{c}^{<t>} + (1-\Gamma_u) \star c^{<t-1>}$	$\Gamma_u \star \tilde{c}^{<t>} + \Gamma_f \star c^{<t-1>}$
$a^{<t>}$	$c^{<t>}$	$\Gamma_o \star c^{<t>}$
Dependencies		

Remark: the sign $\star$ denotes the element-wise multiplication between two vectors.

❒ **Variants of RNNs** – The table below sums up the other commonly used RNN architectures:

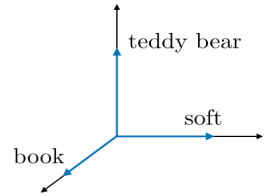
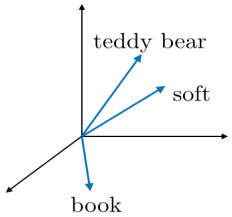
Bidirectional (BRNN)	**Deep** (DRNN)
	

2.3 Learning word representation

In this section, we note V the vocabulary and $|V|$ its size.

2.3.1 Motivation and notations

❒ **Representation techniques** – The two main ways of representing words are summed up in the table below:

1-hot representation	**Word embedding**
	
- Noted o_w - Naive approach, no similarity information	- Noted e_w - Takes into account words similarity

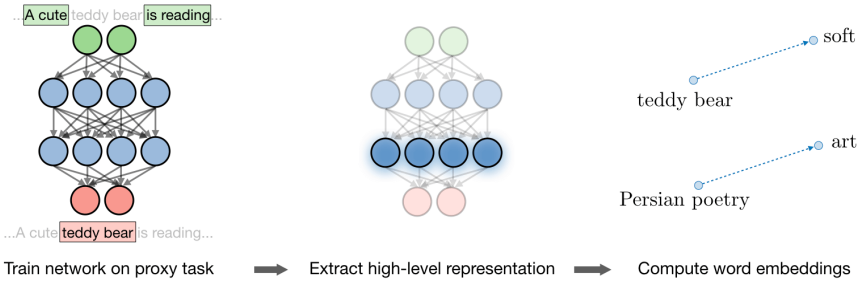
❒ **Embedding matrix** – For a given word w, the embedding matrix E is a matrix that maps its 1-hot representation o_w to its embedding e_w as follows:

$$\boxed{e_w = Eo_w}$$

Remark: learning the embedding matrix can be done using target/context likelihood models.

2.3.2 Word embeddings

❒ **Word2vec** – Word2vec is a framework aimed at learning word embeddings by estimating the likelihood that a given word is surrounded by other words. Popular models include skip-gram, negative sampling and CBOW.



❒ **Skip-gram** – The skip-gram word2vec model is a supervised learning task that learns word embeddings by assessing the likelihood of any given target word t happening with a context word c. By noting θ_t a parameter associated with t, the probability $P(t|c)$ is given by:

$$P(t|c) = \frac{\exp(\theta_t^T e_c)}{\sum_{j=1}^{|V|} \exp(\theta_j^T e_c)}$$

Remark: summing over the whole vocabulary in the denominator of the softmax part makes this model computationally expensive. CBOW is another word2vec model using the surrounding words to predict a given word.

❒ **Negative sampling** – It is a set of binary classifiers using logistic regressions that aim at assessing how a given context and a given target words are likely to appear simultaneously, with the models being trained on sets of k negative examples and 1 positive example. Given a context word c and a target word t, the prediction is expressed by:

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

Remark: this method is less computationally expensive than the skip-gram model.

❒ **GloVe** – The GloVe model, short for global vectors for word representation, is a word embedding technique that uses a co-occurence matrix X where each $X_{i,j}$ denotes the number of times that a target i occurred with a context j. Its cost function J is as follows:

$$J(\theta) = \frac{1}{2} \sum_{i,j=1}^{|V|} f(X_{ij})(\theta_i^T e_j + b_i + b'_j - \log(X_{ij}))^2$$

here f is a weighting function such that $X_{i,j} = 0 \Longrightarrow f(X_{i,j}) = 0$.

Given the symmetry that e and θ play in this model, the final word embedding $e_w^{(\text{final})}$ is given by:

$$e_w^{(\text{final})} = \frac{e_w + \theta_w}{2}$$

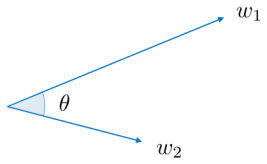
Remark: the individual components of the learned word embeddings are not necessarily interpretable.

2.4 Comparing words

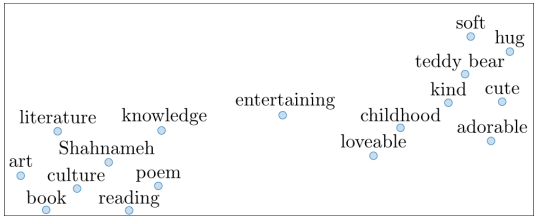
❒ **Cosine similarity** – The cosine similarity between words w_1 and w_2 is expressed as follows:

$$\text{similarity} = \frac{w_1 \cdot w_2}{||w_1||\ ||w_2||} = \cos(\theta)$$

Remark: θ is the angle between words w_1 and w_2.



❒ ***t*-SNE** – t-SNE (t-distributed Stochastic Neighbor Embedding) is a technique aimed at reducing high-dimensional embeddings into a lower dimensional space. In practice, it is commonly used to visualize word vectors in the 2D space.



2.5 Language model

❒ **Overview** – A language model aims at estimating the probability of a sentence $P(y)$.

❒ **n-gram model** – This model is a naive approach aiming at quantifying the probability that an expression appears in a corpus by counting its number of appearance in the training data.

❒ **Perplexity** – Language models are commonly assessed using the perplexity metric, also known as PP, which can be interpreted as the inverse probability of the dataset normalized by the number of words T. The perplexity is such that the lower, the better and is defined as follows:

$$\text{PP} = \prod_{t=1}^{T} \left(\frac{1}{\sum_{j=1}^{|V|} y_j^{(t)} \cdot \hat{y}_j^{(t)}} \right)^{\frac{1}{T}}$$

Remark: PP is commonly used in t-SNE.

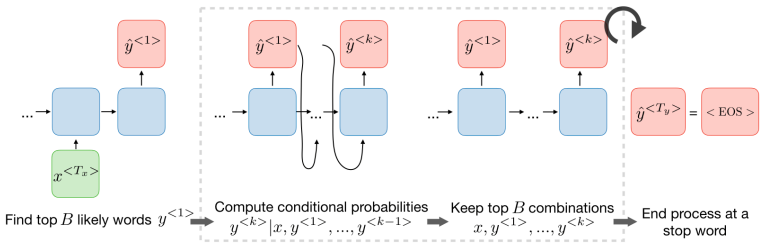
2.6 Machine translation

❒ **Overview** – A machine translation model is similar to a language model except it has an encoder network placed before. For this reason, it is sometimes referred as a conditional language model. The goal is to find a sentence y such that:

$$y = \underset{y^{<1>},\ldots,y^{<T_y>}}{\arg\max}\ P(y^{<1>},\ldots,y^{<T_y>}|x)$$

❒ **Beam search** – It is a heuristic search algorithm used in machine translation and speech recognition to find the likeliest sentence y given an input x.

- Step 1: Find top B likely words $y^{<1>}$
- Step 2: Compute conditional probabilities $y^{<k>}|x,y^{<1>},\ldots,y^{<k-1>}$
- Step 3: Keep top B combinations $x,y^{<1>},\ldots,y^{<k>}$



Remark: if the beam width is set to 1, then this is equivalent to a naive greedy search.

❒ **Beam width** – The beam width B is a parameter for beam search. Large values of B yield to better result but with slower performance and increased memory. Small values of B lead to worse results but is less computationally intensive. A standard value for B is around 10.

❒ **Length normalization** – In order to improve numerical stability, beam search is usually applied on the following normalized objective, often called the normalized log-likelihood objective, defined as:

$$\text{Objective } = \frac{1}{T_y^\alpha}\sum_{t=1}^{T_y}\log\left[p(y^{<t>}|x,y^{<1>},...,y^{<t-1>})\right]$$

Remark: the parameter α can be seen as a softener, and its value is usually between 0.5 and 1.

❒ **Error analysis** – When obtaining a predicted translation $\widehat{y}$ that is bad, one can wonder why we did not get a good translation y^* by performing the following error analysis:

Case	$P(y^*\|x) > P(\widehat{y}\|x)$	$P(y^*\|x) \leqslant P(\widehat{y}\|x)$
Root cause	Beam search faulty	RNN faulty
Remedies	Increase beam width	- Try different architecture - Regularize - Get more data

❒ **Bleu score** – The bilingual evaluation understudy (bleu) score quantifies how good a machine translation is by computing a similarity score based on n-gram precision. It is defined as follows:

$$\text{bleu score} = \exp\left(\frac{1}{n}\sum_{k=1}^{n}p_k\right)$$

where p_n is the bleu score on n-gram only defined as follows:

$$p_n = \frac{\sum_{\text{n-gram}\in\widehat{y}}\text{count}_{\text{clip}}(\text{n-gram})}{\sum_{\text{n-gram}\in\widehat{y}}\text{count}(\text{n-gram})}$$

Remark: a brevity penalty may be applied to short predicted translations to prevent an artificially inflated bleu score.

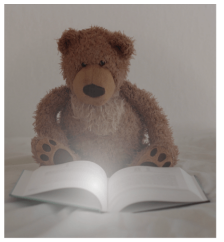
2.7 Attention

❒ **Attention model** – This model allows an RNN to pay attention to specific parts of the input that is considered as being important, which improves the performance of the resulting model in practice. By noting $\alpha^{<t,t'>}$ the amount of attention that the output $y^{<t>}$ should pay to the activation $a^{<t'>}$ and $c^{<t>}$ the context at time t, we have:

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

Remark: the attention scores are commonly used in image captioning and machine translation.

A cute teddy bear is reading Persian literature



A cute teddy bear is reading Persian literature

❒ **Attention weight** – The amount of attention that the output $y^{<t>}$ should pay to the activation $a^{<t'>}$ is given by $\alpha^{<t,t'>}$ computed as follows:

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

Remark: computation complexity is quadratic with respect to T_x.

⋆ ⋆ ⋆

3 Deep Learning Tips and Tricks

3.1 Data processing

❒ **Data augmentation** – Deep learning models usually need a lot of data to be properly trained. It is often useful to get more data from the existing ones using data augmentation techniques. The main ones are summed up in the table below. More precisely, given the following input image, here are the techniques that we can apply:

Original	Flip	Rotation	Random crop
			
- Image without any modification	- Flipped with respect to an axis for which the meaning of the image is preserved	- Rotation with a slight angle - Simulates incorrect horizon calibration	- Random focus on one part of the image - Several random crops can be done in a row

Color shift	Noise addition	Information loss	Contrast change
			
- Nuances of RGB is slightly changed - Captures noise that can occur with light exposure	- Addition of noise - More tolerance to quality variation of inputs	- Parts of image ignored - Mimics potential loss of parts of image	- Luminosity changes - Controls difference in exposition due to time of day

❒ **Batch normalization** – It is a step of hyperparameter γ, β that normalizes the batch $\{x_i\}$. By noting μ_B, σ_B^2 the mean and variance of that we want to correct to the batch, it is done as follows:

$$\boxed{x_i \longleftarrow \gamma \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta}$$

It is usually done after a fully connected/convolutional layer and before a non-linearity layer and aims at allowing higher learning rates and reducing the strong dependence on initialization.

3.2 Training a neural network

3.2.1 Definitions

❒ **Epoch** – In the context of training a model, epoch is a term used to refer to one iteration where the model sees the whole training set to update its weights.

❒ **Mini-batch gradient descent** – During the training phase, updating weights is usually not based on the whole training set at once due to computation complexities or one data point due to noise issues. Instead, the update step is done on mini-batches, where the number of data points in a batch is a hyperparameter that we can tune.

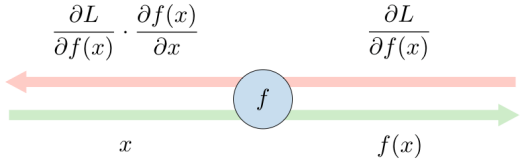
❒ **Loss function** – In order to quantify how a given model performs, the loss function L is usually used to evaluate to what extent the actual outputs y are correctly predicted by the model outputs z.

❒ **Cross-entropy loss** – In the context of binary classification in neural networks, the cross-entropy loss $L(z,y)$ is commonly used and is defined as follows:

$$\boxed{L(z,y) = -\Big[y\log(z) + (1-y)\log(1-z)\Big]}$$

3.2.2 Finding optimal weights

❒ **Backpropagation** – Backpropagation is a method to update the weights in the neural network by taking into account the actual output and the desired output. The derivative with respect to each weight w is computed using the chain rule.

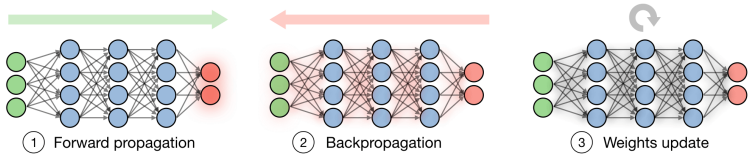


Using this method, each weight is updated with the rule:

$$\boxed{w \longleftarrow w - \alpha \frac{\partial L(z,y)}{\partial w}}$$

❒ **Updating weights** – In a neural network, weights are updated as follows:

- Step 1: Take a batch of training data and perform forward propagation to compute the loss.
- Step 2: Backpropagate the loss to get the gradient of the loss with respect to each weight.
- Step 3: Use the gradients to update the weights of the network.

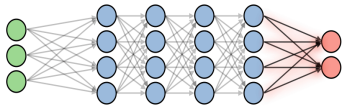
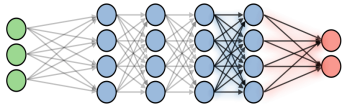
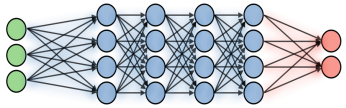


3.3 Parameter tuning

3.3.1 Weights initialization

❒ **Xavier initialization** – Instead of initializing the weights in a purely random manner, Xavier initialization enables to have initial weights that take into account characteristics that are unique to the architecture.

❒ **Transfer learning** – Training a deep learning model requires a lot of data and more importantly a lot of time. It is often useful to take advantage of pre-trained weights on huge datasets that took days/weeks to train, and leverage it towards our use case. Depending on how much data we have at hand, here are the different ways to leverage this:

Training size	Illustration	Explanation
Small		Freezes all layers, trains weights on softmax
Medium		Freezes most layers, trains weights on last layers and softmax
Large		Trains weights on layers and softmax by initializing weights on pre-trained ones

3.3.2 Optimizing convergence

❒ **Learning rate** – The learning rate, often noted α or sometimes η, indicates at which pace the weights get updated. It can be fixed or adaptively changed. The current most popular method is called Adam, which is a method that adapts the learning rate.

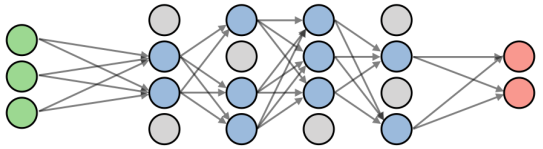
❒ **Adaptive learning rates** – Letting the learning rate vary when training a model can reduce the training time and improve the numerical optimal solution. While Adam optimizer is the most commonly used technique, others can also be useful. They are summed up in the table below:

Method	Explanation	Update of w	Update of b
Momentum	- Dampens oscillations - Improvement to SGD - 2 parameters to tune	$w - \alpha v_{dw}$	$b - \alpha v_{db}$
RMSprop	- Root Mean Square propagation - Speeds up learning algorithm by controlling oscillations	$w - \alpha \frac{dw}{\sqrt{s_{dw}}}$	$b \longleftarrow b - \alpha \frac{db}{\sqrt{s_{db}}}$
Adam	- Adaptive Moment estimation - Most popular method - 4 parameters to tune	$w - \alpha \frac{v_{dw}}{\sqrt{s_{dw}} + \epsilon}$	$b \longleftarrow b - \alpha \frac{v_{db}}{\sqrt{s_{db}} + \epsilon}$

Remark: other methods include Adadelta, Adagrad and SGD.

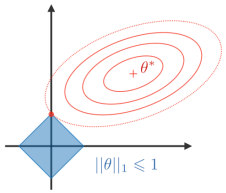
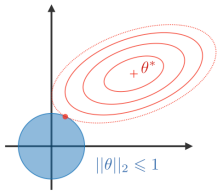
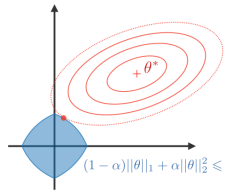
3.4 Regularization

❒ **Dropout** – Dropout is a technique used in neural networks to prevent overfitting the training data by dropping out neurons with probability $p > 0$. It forces the model to avoid relying too much on particular sets of features.

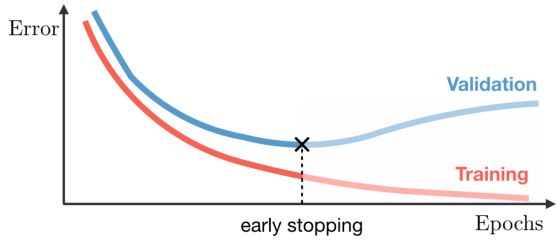


Remark: most deep learning frameworks parametrize dropout through the 'keep' parameter $1-p$.

❒ **Weight regularization** – In order to make sure that the weights are not too large and that the model is not overfitting the training set, regularization techniques are usually performed on the model weights. The main ones are summed up in the table below:

LASSO	Ridge	Elastic Net
- Shrinks coefficients to 0 - Good for variable selection	Makes coefficients smaller	Tradeoff between variable selection and small coefficients
		
$... + \lambda \|\|\theta\|\|_1$ $\lambda \in \mathbb{R}$	$... + \lambda \|\|\theta\|\|_2^2$ $\lambda \in \mathbb{R}$	$... + \lambda \left[(1-\alpha) \|\|\theta\|\|_1 + \alpha \|\|\theta\|\|_2^2 \right]$ $\lambda \in \mathbb{R}, \alpha \in [0,1]$

❒ **Early stopping** – This regularization technique stops the training process as soon as the validation loss reaches a plateau or starts to increase.



3.5 Good practices

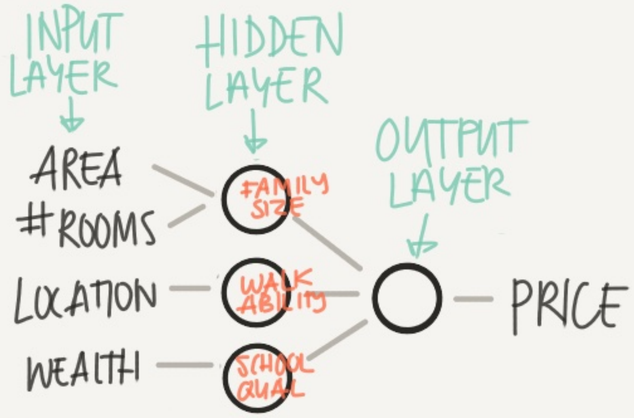
❒ **Overfitting small batch** – When debugging a model, it is often useful to make quick tests to see if there is any major issue with the architecture of the model itself. In particular, in order to make sure that the model can be properly trained, a mini-batch is passed inside the network to see if it can overfit on it. If it cannot, it means that the model is either too complex or not complex enough to even overfit on a small batch, let alone a normal-sized training set.

❒ **Gradient checking** – Gradient checking is a method used during the implementation of the backward pass of a neural network. It compares the value of the analytical gradient to the numerical gradient at given points and plays the role of a sanity-check for correctness.

	Numerical gradient	**Analytical gradient**
Formula	$\frac{df}{dx}(x) \approx \frac{f(x+h)-f(x-h)}{2h}$	$\frac{df}{dx}(x) = f'(x)$
Comments	- Expensive; loss has to be computed two times per dimension - Used to verify correctness of analytical implementation -Trade-off in choosing h not too small (numerical instability) nor too large (poor gradient approx.)	- 'Exact' result - Direct computation - Used in the final implementation

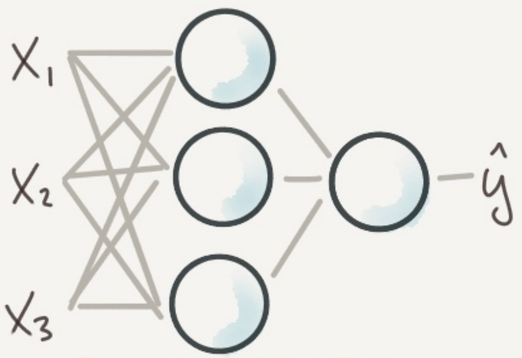
⋆ ⋆ ⋆

INTRO TO DEEP LEARNING

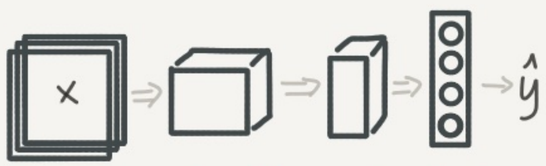


SUPERVISED LEARNING

INPUT: X	OUTPUT: Y	NN TYPE
HOME FEATURES	PRICE	STANDARD NN
AD + USER INFO	WILL CLICK ON AD (0/1)	
IMAGE	OBJECT (1...1000)	CONV. NN (CNN)
AUDIO	TEXT TRANSCRIPT	RECURRENT NN (RNN)
ENGLISH	CHINESE	
IMAGE/RADAR	POS OF OTHER CARS	CUSTOM/HYBRID

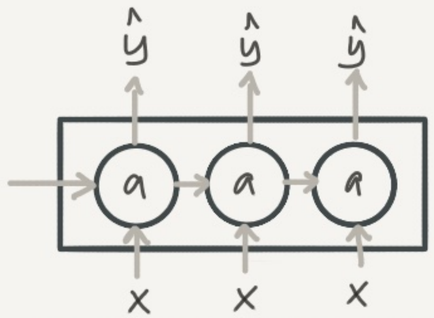


STANDARD NN



CONVOLUTIONAL NN

NETWORK ARCHITECTURES



RECURRENT NN

NNs CAN DEAL WITH BOTH STRUCTURED & UNSTRUCTURED DATA

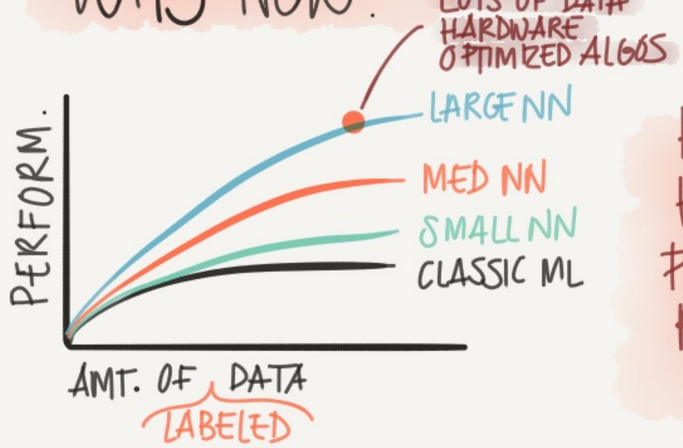
STRUCTURED

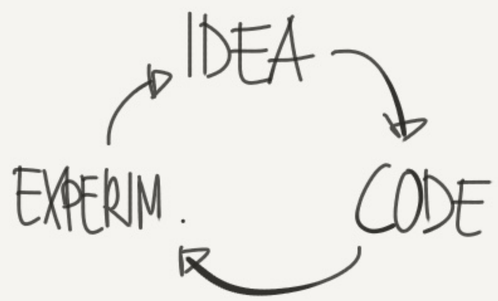
"THE QUICK BROWN FOX"

UNSTRUCTURED

HUMANS ARE GOOD AT THIS

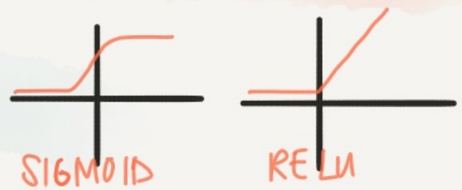
WHY NOW?





FASTER COMPUTATION IS IMPORTANT TO SPEED UP THE ITERATIVE PROCESS

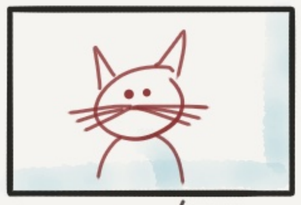
ONE OF THE BIG BREAKTHROUGHS HAS BEEN MOVING FROM SIGMOID TO RELU FOR FASTER GRADIENT DESCENT



LOGISTIC REGRESSION

AS A NEURAL NET

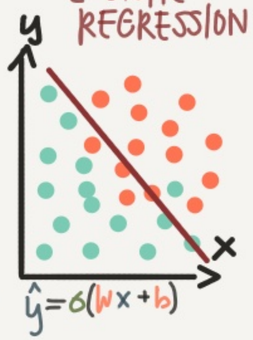
BINARY CLASSIFICATION



1 : CAT
0 : NOT CAT

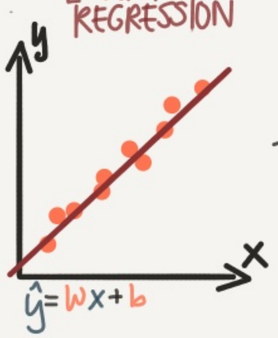
X → y

LOGISTIC REGRESSION



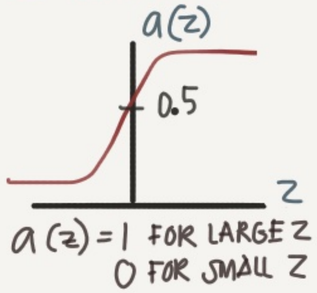
=

LINEAR REGRESSION

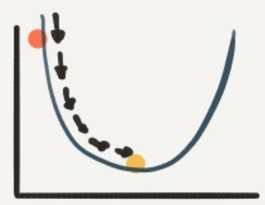


+

σ SIGMOID



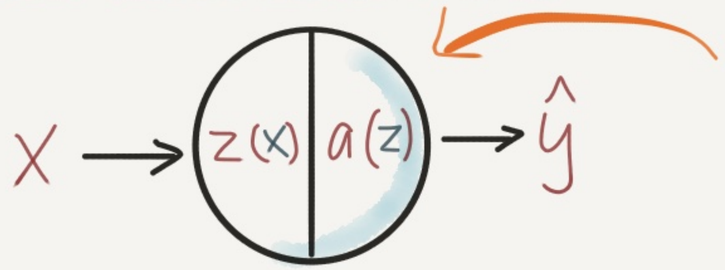
FINDING THE MINIMUM WITH GRADIENT DESCENT



1. FIND THE DOWNHILL DIRECTION (USING DERIVATIVES)
2. WALK (UPDATE W & B) AT A α LEARNING RATE

REPEAT UNTIL YOU REACH BOTTOM (CONVERGE)

PUTTING IT ALL TOGETHER



$$z(x) = wx + b$$

$$\hat{y} = a(z) = \sigma\,\text{SIGMOID}(z)$$

1. FORWARD PROPAGATION • CALCULATE $\hat{y}$
2. BACKWARD PROPAGATION • GRADIENT DESCENT + UPDATE W & b

REPEAT UNTIL IT CONVERGES

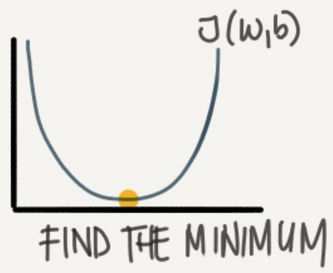
THE TASK IS TO LEARN w & b BUT HOW?

A : OPTIMIZE HOW GOOD THE GUESS IS BY MINIMIZING THE DIFF BETWEEN GUESS ($\hat{y}$) AND TRUTH (y)

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

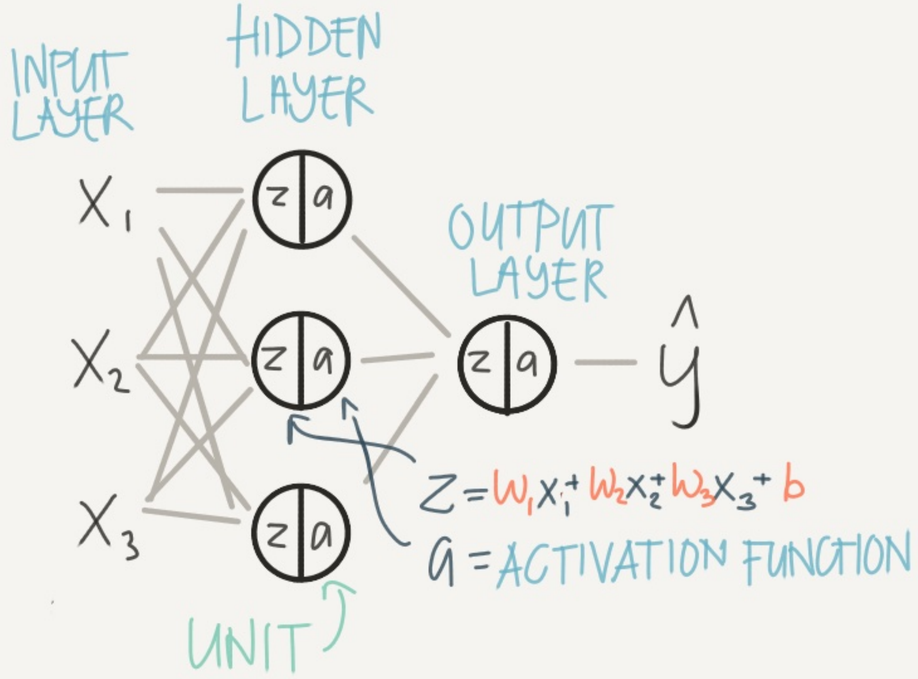
$$\text{COST} = J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

COST = LOSS FOR THE ENTIRE DATASET



SHALLOW NEURAL NETS

2 LAYER NEURAL NET



WHY ACTIVATION FUNCTIONS?

EX. WITH NO ACTIVATION – $a = z$

$$a^{[1]} = z^{[1]} = w^{[1]}x + b^{[1]} \quad \text{LAYER 1}$$
$$a^{[2]} = z^{[2]} = w^{[2]}a^{[1]} + b^{[2]} \quad \text{LAYER 2}$$

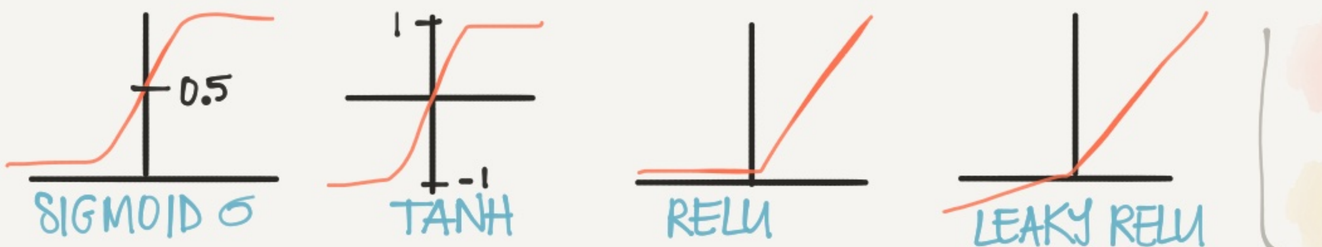
PLUG IN $a^{[1]}$

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

← LINEAR FUNCTION

WE COULD JUST AS WELL HAVE SKIPPED THE WHOLE NEURAL NET & USED LIN. REGR.

ACTIVATION FUNCTIONS



SIGMOID σ
- BINARY CLASSIFIER - ONLY USED FOR OUTPUT LAYER
- SLOW GRAD DESCENT SINCE SLOPE IS SMALL FOR LARGE/SMALL VAL

TANH
- NORMALIZED ⇒ GRADIENT DESCENT IS FASTER

RELU
- DEFAULT CHOICE FOR ACTIVATION
- SLOPE = 1/0

LEAKY RELU
- AVOIDS UNDEF SLOPE AT 0 BUT RARELY USED IN PRACTICE

INITIALIZING w+b

WHAT IF: INIT TO 0

THIS WILL CAUSE ALL THE UNITS TO BE THE SAME AND LEARN EXACTLY THE SAME FEATURES

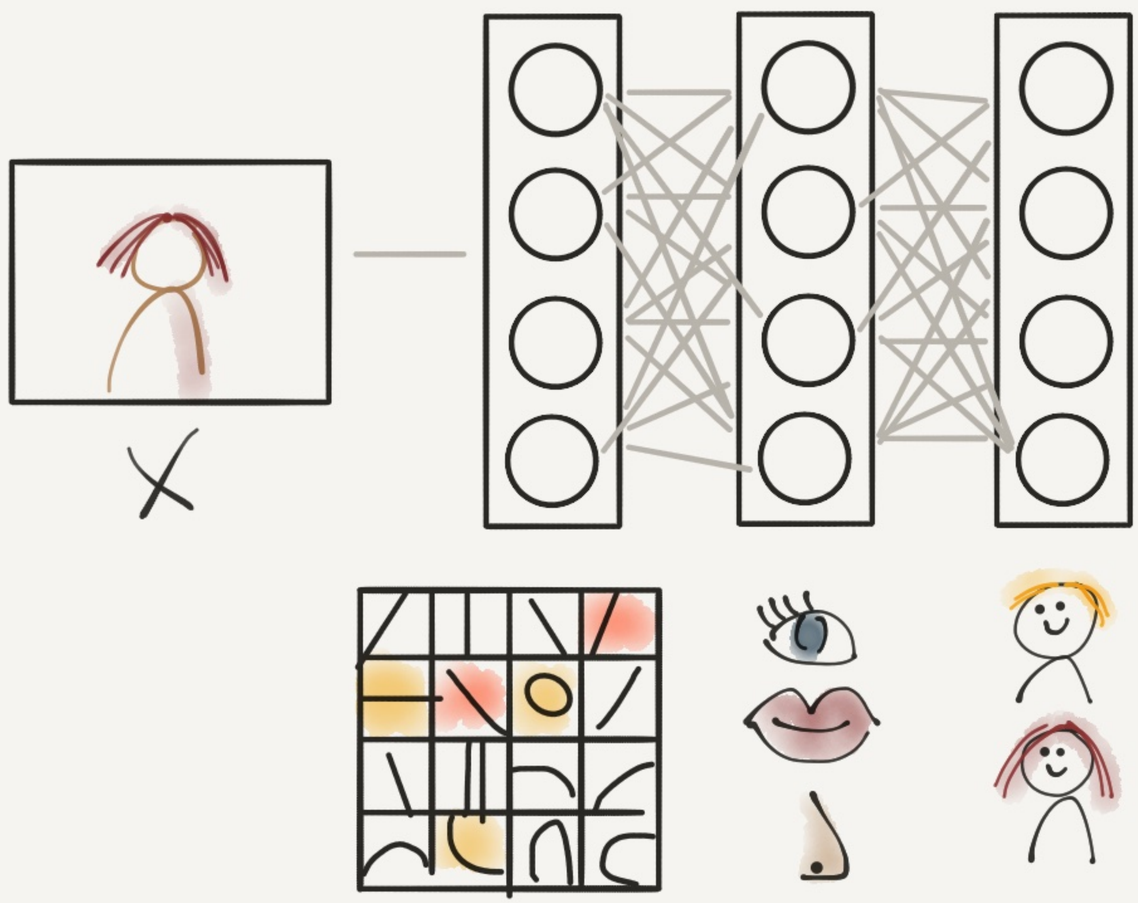
SOLUTION: RANDOM INIT

BUT ALSO WANT THEM SMALL SO RAND * 0.01 ← HYPERPARAM

DEEP NEURAL NETS

THERE ARE FUNCTIONS A SMALL DEEP NET CAN COMPUTE THAT SHALLOW NETS NEED EXP. MORE UNITS TO COMP.

WHY DEEP NEURAL NETS?



LOW LEVEL AUDIO WAVE FEATURES ↗↘ PITCH — PHONEMES — C A T — WORDS — SENTENCES

VERY DATA HUNGRY

NEED LOTS OF COMPUTER POWER

ALWAYS VECTORIZE

VECTOR MULT · CHEAPER THAN FOR LOOPS

COMPUTE ON GPUs

LOTS OF HYPERPARAMS

- LEARNING RATE α
- # ITERATIONS
- # HIDDEN LAYERS
- # HIDDEN UNITS
- CHOICE OF ACTIVATION
- MOMENTUM
- MINI-BATCH SIZE
- REGULARIZATION

SETTING UP
YOUR ML APP

CLASSIC ML

100 - 10000 SAMPLES

DEEP LEARNING

1M SAMPLES

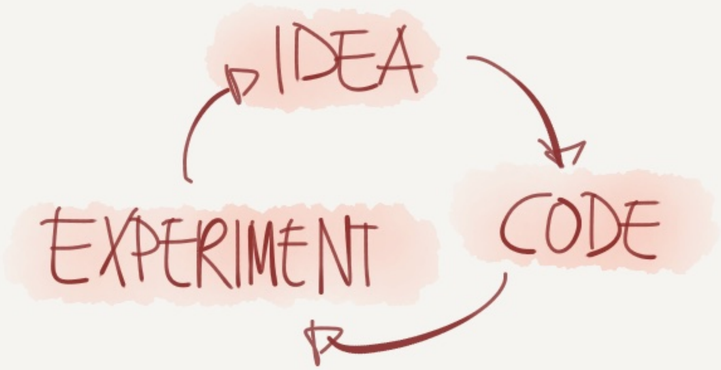
ALL FROM SAME PLACE → DISTRIBUTION

EX: TRAIN  PRO CAT PICS FROM INTERNET

DEV/TEST  BLURRY CAT PICS FROM APP

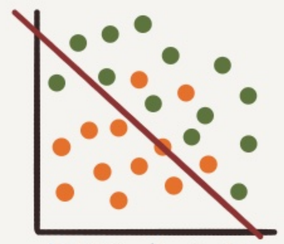
TIP

DEV & TEST SHOULD COME FROM SAME DISTRIBUTION

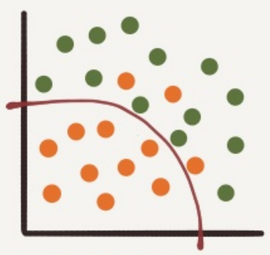


IMPROVING DEEP NEURAL NETS · COURSERA

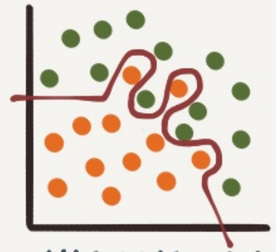
BIAS/VARIANCE



HIGH BIAS "UNDERFIT"



JUST RIGHT

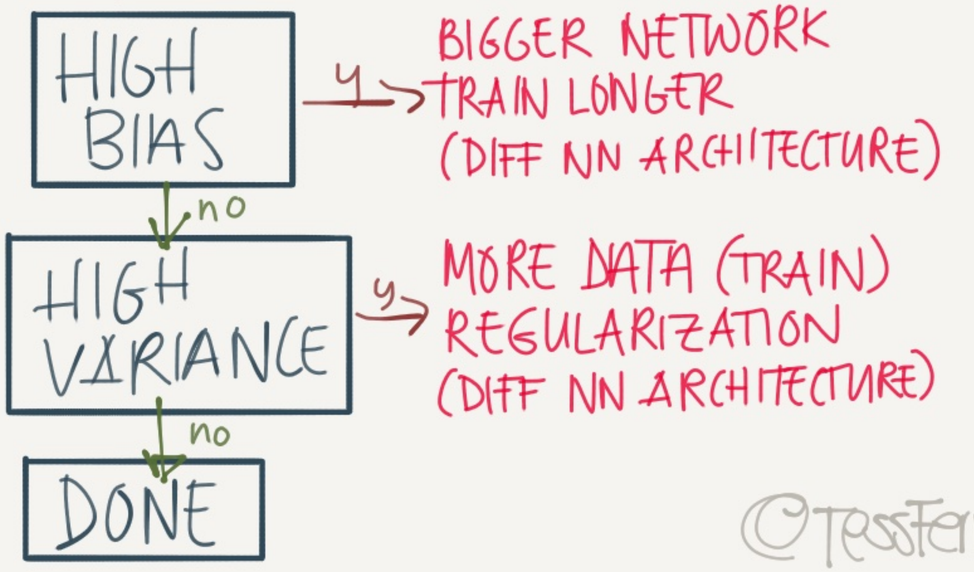


HIGH VARIANCE "OVERFIT"

	ERROR			
TRAIN	1%	15%	15%	0.5%
TEST	11%	16%	30%	1%
	HIGH VARIANCE	HIGH BIAS	HIGH BIAS & VARIANCE	LOW BIAS & VARIANCE

ASSUMING HUMANS GET 0% ERROR

THE ML RECIPE



REGULARIZATION
PREVENTING OVERFITTING

L2 REGULARIZATION

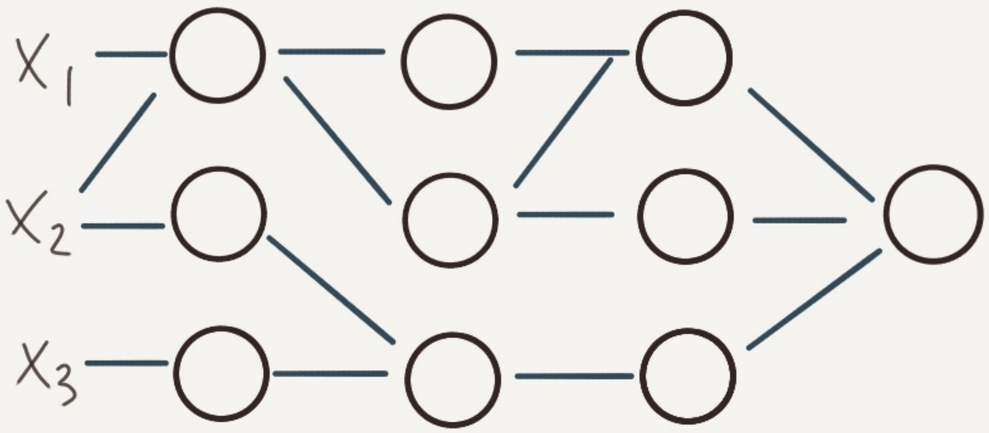
$$\text{COST}: J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y},y) + \frac{\lambda}{2m}\|w\|_2^2$$

← EUCLIDEAN NORM

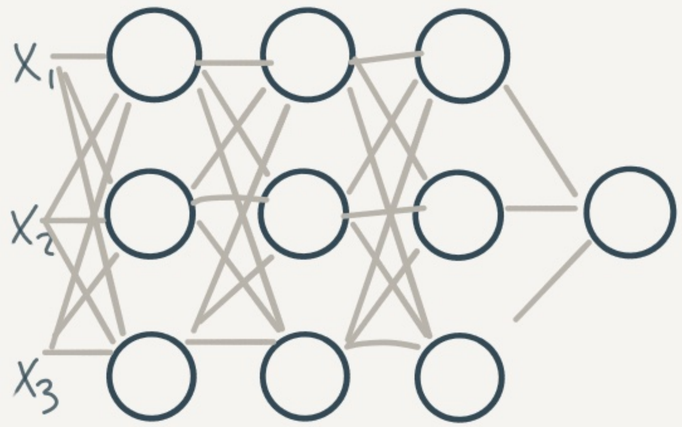
L1 REGULARIZATION

$$\text{COST}: J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y},y) + \frac{\lambda}{m}\|w\|_1$$

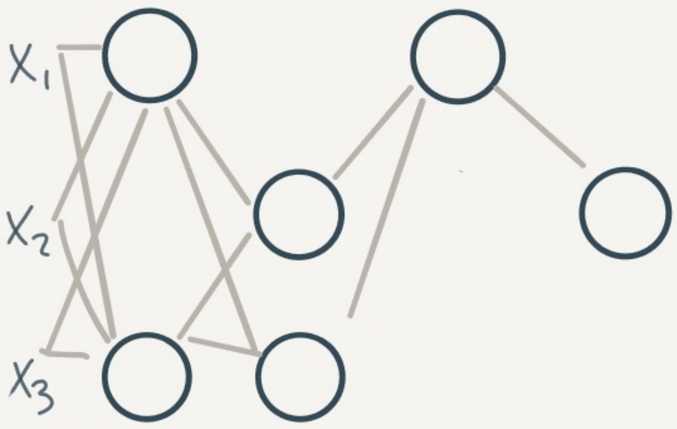
BOTH PENALIZE LARGE WEIGHTS ⇒
SOME WILL BE CLOSE TO Ø ⇒
SIMPLER NETWORKS



DROPOUT



FOR EACH ITERATION & SAMPLE SOME NODES ARE RANDOMLY DROPPED (BASED ON KEEP-PROB)

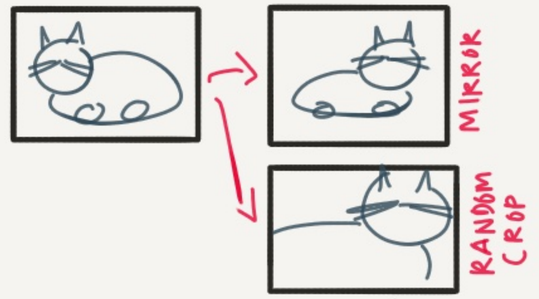


WE GET SIMPLER NWs & LESS CHANCE TO RELY ON SINGLE FEATURES

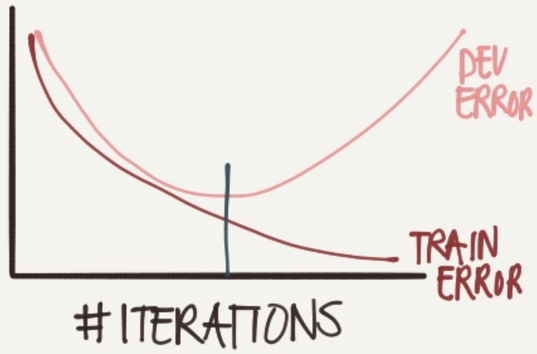
OTHER REGULARIZATION TECHNIQUES

DATA AUGMENTATION

GENERATE NEW PICS FROM EXISTING



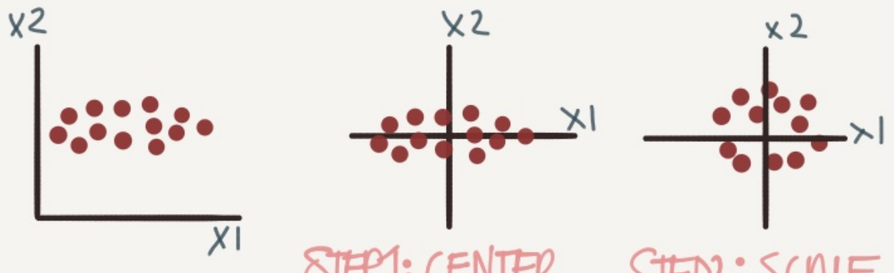
EARLY STOPPING



PROBLEM: AFFECTS BOTH BIAS & VARIANCE

OPTIMIZING TRAINING

NORMALIZING INPUTS



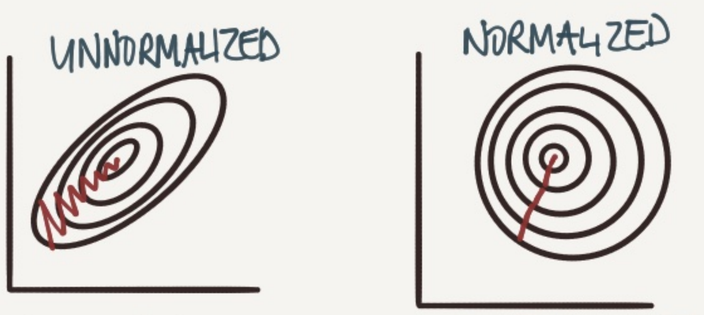
STEP1: CENTER AROUND 0,0

STEP2: SCALE SO VARIANCE IS SAME CX $-1 \rightarrow 1$

TIP

USE SAME AVG/VAR TO NORMALIZE DEV/TEST

WHY DO WE DO THIS?



IF WE NORMALIZE, WE CAN USE A MUCH LARGER LEARNING RATE α

DEALING WITH VANISHING/EXPLODING GRADIENTS

EX: DEEP NW (L LAYERS)

$\hat{y} = W^{[L-1]} W^{[L-2]} \cdots W^{[1]} X + b$

IF $W = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \Rightarrow 0.5^{L-1} \Rightarrow$ VANISHING

OR $W = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix} \Rightarrow 1.5^{L-1} \Rightarrow$ EXPLODING

IN BOTH CASES GRADIENT DESCENT TAKES A VERY LONG TIME

PARTIAL SOLUTION: CHOOSE INITIAL VALUES CAREFULLY

$W^{[l]} = rand * \sqrt{\frac{2}{n^{[l-1]}}}$ (FOR RELU)

$n^{[l-1]}$ = #inputs

XAVIER $\sqrt{\frac{1}{n^{[l-1]}}}$ (FOR TANH)

SETS THE VARIANCE

GRADIENT CHECKING

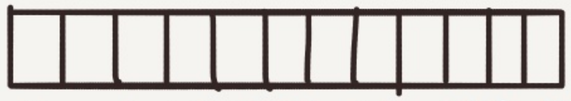
IF YOUR COST DOES NOT DECREASE ON EACH ITER YOU MAY HAVE A BACKPROP BUG.

GRADIENT CHECKING APPROXIMATES THE GRADIENTS SO YOU CAN VERIFY CALC.

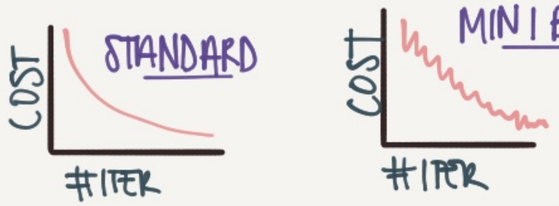
NOTE ONLY USE WHEN DEBUGGING SINCE IT'S SLOW

OPTIMIZATION ALGORITHMS

MINI-BATCH GRAD. DESCENT

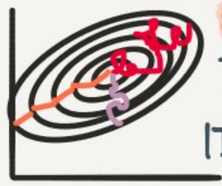


SPLIT YOUR DATA INTO MINI-BATCHES & DO GRAD DESCENT AFTER EACH BATCH
THIS WAY YOU CAN PROGRESS AFTER JUST A SHORT WHILE



CHOOSING THE MINIBATCH SIZE

SIZE = m → BATCH GRAD DESC.
SIZE = 1 → STOCHASTIC GRAD DESC

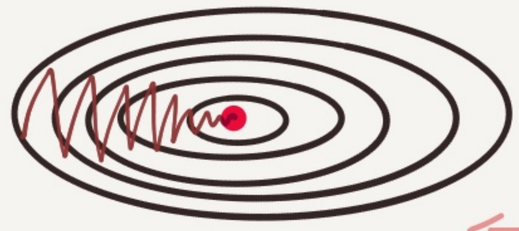


BATCH — TOO LONG PER ITERATION
MIDDLE
STOCHASTIC — LOOSE ALMOST ALL SPEED FROM VECTORIZATION

- SHORT ITER.
- USES VECT.

TIP
IF YOU HAVE < 2000 SAMPLES
USE SIZE = 2000
OTHERWISE, USE 64, 128, 256...
SO X+Y FITS IN CPU/GPU CACHE

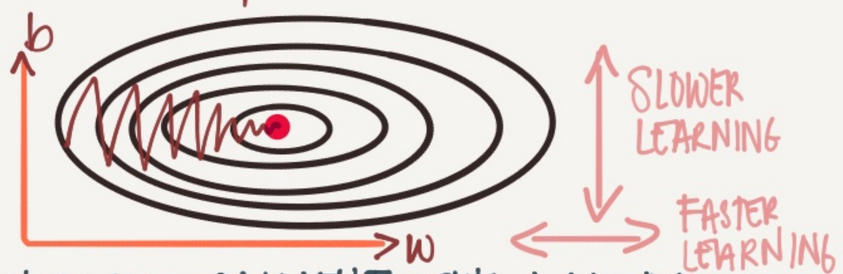
GRADIENT DESCENT W. MOMENTUM



WE WANT TO REDUCE OSCILLATION ↕ SO WE GET TO THE GOAL FASTER

SOLUTION: SMOOTH OUT THE CURVE BY TAKING AN EXPONENTIALLY WEIGHTED AVERAGE OF THE DERIVATIVES (i.e. LAST ONE HAS MORE IMPORTANCE)

RMSProp - ROOT MEAN SQUARED



NORMALIZE GRADIENT USING A MOVING AVG.

$$S_{dw} = \beta S_{dw} + (1-\beta)\, dw^2$$
$$S_{db} = \beta S_{db} + (1-\beta)\, db^2$$
$$w = w - \alpha \frac{dw}{\sqrt{S_{dw}}} \qquad b = b - \alpha \frac{db}{\sqrt{S_{db}}}$$

ADAM OPTIMIZATION

COMBO OF GD W MOMENTUM & RMSProp

LEARNING RATE DECAY

IDEA: USE A LARGE α IN THE BEGINNING. THEN DECREASE AS WE GET CLOSER TO GOAL

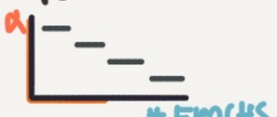
OPTION 1: $\alpha = \frac{1}{1 + \text{DECAYRATE} \cdot \text{EPOCH}} \alpha_0$

EXPONENTIAL: $\alpha = 0.95^{\text{EPOCH}} \alpha_0$

OPTION 3: $\alpha = \frac{k}{\sqrt{\text{EPOCH}}} \alpha_0$

OPTION 4: $\alpha = \frac{k}{\sqrt{t}} \alpha_0$

OPTION 5: DISCRETE STAIRCASE



OPTION 6: MANUAL

EPOCH = 1 PASS THROUGH THE DATA

IMPROVING DEEP NEURAL NETS · COURSERA

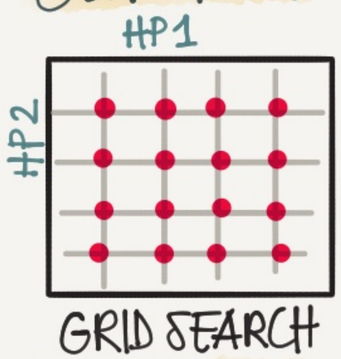
HYPERPARAM TUNING

WHICH HYPERPARAMS ARE MOST IMPORTANT?

- α LEARNING RATE
- # HIDDEN UNITS
- MINIBATCH SIZE
- β MOMENTUM TERM = 0.9
- # LAYERS
- LEARNING RATE DECAY
- $\beta_1 = 0.9$ $\beta_2 = 0.999$ $\varepsilon = 10^{-8}$ (ADAM)

TESTING VALUES

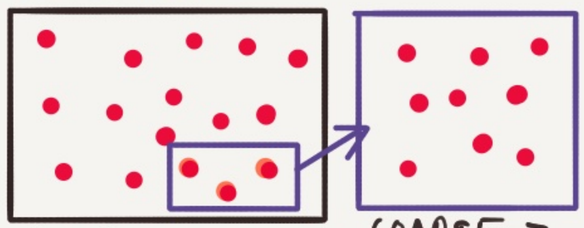
CLASSIC ML



GRID SEARCH

PROBLEM: ONE ITERATION TAKES A LONG TIME & IN 16 GO'S WE HAVE ONLY TRIED 4 α - BUT 4 DIFF ε ← NOT AS IMPORTANT

SOLUTION



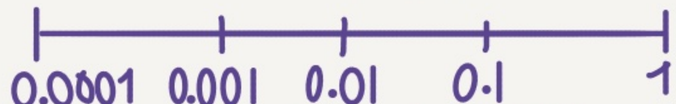
RANDOM SEARCH + COARSE → DENSE

USE AN APPROPRIATE SCALE

HIDDEN UNITS

UNIFORMLY RANDOM

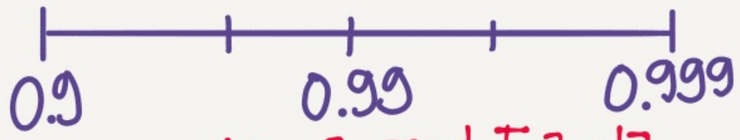
α LEARNING RATE



$r = -4 \cdot \text{rand}\ [-4, 0]$

$\alpha = 10^r$

β EXP WEIGHT AVE



$r = -3 \cdot \text{rand}\ [-3, -1]$

$\beta = (1 - 10^r)$

TIP

RE-EVALUATE YOUR HYP-PARAMS EVERY FEW MONTHS

PANDA VS CAVIAR

MY PANDA IS ACTUALLY A MISCLASSIFIED CAT BECAUSE I CAN'T DRAW PANDAS

BABYSIT ONE MODEL & TUNE

SPAWN LOTS OF MODELS W DIFF HP

GOOD IF YOU HAVE LOTS OF SPARE COMP POWER

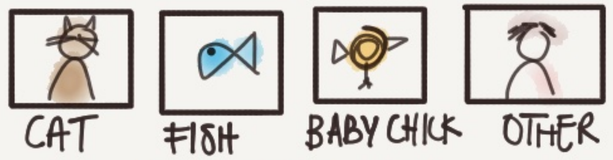
MISC. EXTRAS

BATCH NORMALIZATION

NORMALIZE LAYER OUTPUT

- SPEEDS UP TRAINING
- MAKES WEIGHTS DEEPER IN NW MORE ROBUST (COVARIATE SHIFT)
- SLIGHT REGULARIZING EFFECT

MULTICLASS CLASSIFIC.

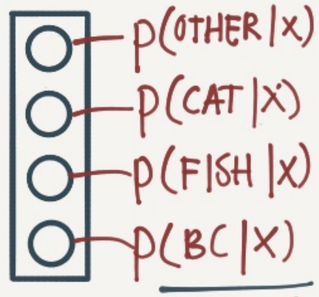


C = # CLASSES = 4

SOFTMAX ACTIVATION

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

$$a^{[L]} = \frac{t}{\sum t_i}$$



SUM: 1

EX: $z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$ $t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}$ 176.3

$\Rightarrow a^{[L]} = \frac{t}{176.3} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.02 \\ 0.114 \end{bmatrix}$ 11.4% PROB IT'S A BABY CHICK

STRUCTURING YOUR ML PROJECTS

SETTING YOUR GOAL

★ GOAL SHOULD BE A SINGLE #

	PRECISION	RECALL
A	95%	90%
B	98%	85%

↳ IS A OR B BEST?

	PRECISION	RECALL	F1
A	95%	90%	92.4%
B	98%	85%	91%

↳ A IS BEST

F1 = HARMONIC MEAN BETW. RECALL & PRECISION

★ DEFINE OPTIMIZING VS SATISFICING METRICS

	ACCURACY	RUNTIME
A	90%	80ms
B	92%	95ms
C	~~95%~~	~~1500ms~~

MAXIMIZE ACC.
GIVEN TIME ≤ 100ms

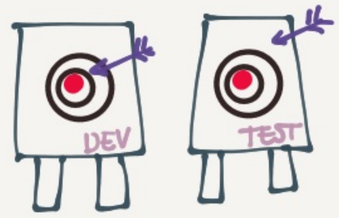
ACCURACY = OPTIMIZING
RUNTIME = SATISFICING

SELECTING YOUR DEV/TEST SETS

DATA

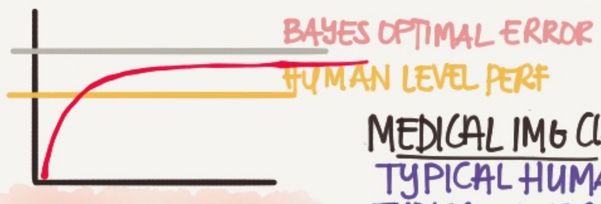
US
UK
~~EUROPE~~
S.AM
INDIA
CHINA
AUST.

OPTION 1:
DEV = UK, US, EUR
TEST = REST



IF DEV & TEST ARE DIFF & WE OPTIMIZE FOR DEV WE WILL MISS THE TEST-TARGET

HUMAN LEVEL PERF



WHY DOES ACC SLOW DOWN WHEN WE SURPASS HUMAN LEVEL PERF?

MEDICAL IMG CLASS
TYPICAL HUMAN 3%
TYPICAL DOCTOR 1%
EXPERIENCED DR. 0.7%
TEAM OF EXP DRs. 0.5%
↑ HUMAN LEV PERF (PROXY FOR BAYES)

1. OFTEN CLOSE TO BAYES
2. A HUMAN CAN NO LONGER HELP IMPROVE (INSIGHTS)
3. DIFFICULT TO ANALYSE BIAS/VARIANCE

CAT CLASSIFICATION

BLURRY ↓

	A	B	
HUMAN	1%	7.5%	← AVOIDABLE BIAS
TRAIN ERR	8%	8%	← VARIANCE
DEV ERR	10%	10%	
	FOCUS ON BIAS	FOCUS ON VARIANCE	

HUMAN
| AVOIDABLE BIAS
TRAIN
| VARIANCE
DEV

- TRAIN BIGGER NETW.
- TRAIN LONGER/BETTER OPT. ALGOS (RMSPROP & ADAM)
- CHANGE NN ARCH OR HYPERPARAMS
- MORE DATA (TRAIN)
- REGULARIZATION
- NN ARCHITECTURE

	A	B	
HUMAN	0.5	0.5	← AVOIDABLE BIAS
TRAIN ERR	0.6	0.3	← VARIANCE
DEV ERR	0.8	0.4	
AVOID. BIAS	0.1	?	↳ DON'T KNOW IF WE OVERFIT OR IF WE'RE CLOSE TO BAYES

OPTIONS TO PROCEED ARE UNCLEAR

STRUCTURING ML PROJECTS • COURSERA

ERROR ANALYSIS

YOU HAVE 10% ERRORS, SOME ARE DOGS MIS-CLASSIFIED AS CATS. SHOULD YOU TRAIN ON MORE DOG PICS?

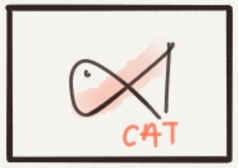
1. PICK 100 MIS-LABLED
2. COUNT ERROR REASONS

	DOG	BLURRY	INSTA FILTER	BIG CAT	...
1	1		1		
2				1	
3		1			
...					
100			1		
	5	...			

5% OF ALL ERRORS

FOCUSING ON DOGS. THE BEST WE CAN HOPE FOR IS 9.5% ERROR

YOU FIND SOME INCORR. LABLED DATA IN THE DEV SET. SHOULD YOU FIX IT?



DL ALGORITHMS ARE PRETTY ROBUST TO RANDOM ERRORS.

BUT NOT TO SYSTEMATIC ERR.

(EX. ALL WHITE CATS INCORR LABLED AS MICE)

ADD EXTRA COL. IN ERROR ANALYSIS AND USE SAME CRITERIA

NOTE IF YOU FIX DEV YOU SHOULD FIX TEST AS WELL.

FOR NEW PROJ. BUILD 1st SYSTEM QUICK & ITERATE

EX: SPEECH RECOGNITION

WHAT SHOULD YOU FOCUS ON?

NOISE
ACCENTS
FAR FROM MIKE

1. START QUICKLY
   DEV/TEST METRICS
2. GET TRAIN-SET
3. TRAIN
4. BIAS/VARIANCE ANAL
5. ERROR ANALYSIS
6. PRIORITIZE NEXT STEP

TRAIN vs DEV/TEST MISMATCH

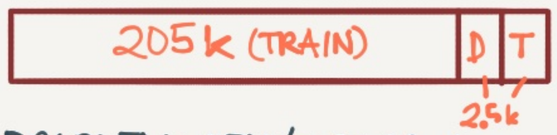
AVAILABLE DATA

200k PRO CAT PICS FROM INTERNET

10k BLURRY CAT PICS FROM APP — WHAT WE CARE ABT

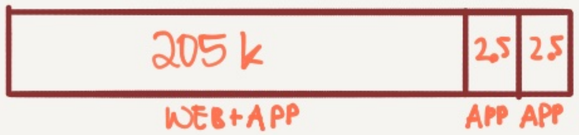
HOW DO WE SPLIT → TRAIN/DEV/TEST?

OPTION 1: SHUFFLE ALL



PROBLEM: DEV/TEST IS NOW MOSTLY WEB IMG (NOT REPRES. OF END SCENARIO)

SOLUTION: LET DEV/TEST COME FROM APP · THEN SHUFFLE 5k OF APP PICS W WEB FOR TRAIN



BIAS & VARIANCE W MISMATCHED TRAIN/DEV

HUMANS	~0%
TRAIN	1%
DEV ERR	10%

IS THIS DIFF DUE TO THE MODEL NOT GENERALIZING OR IS DEV DATA MUCH HARDER

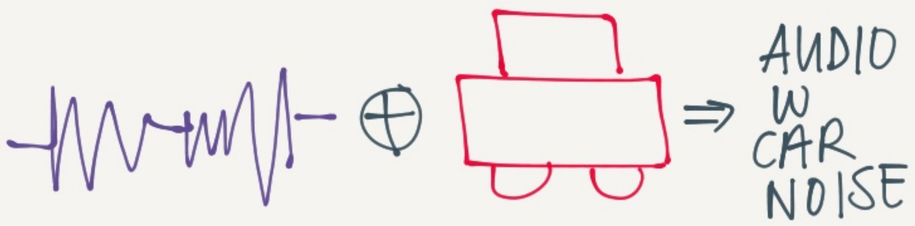
A: CREATE A TRAIN-DEV SET THAT WE DON'T TRAIN ON

	A	B	C	D
TRAIN	1%	1%	10%	10%
TRAIN-DEV	9%	15%	11%	11%
DEV	10%	10%	12%	20%
	VARIANCE	TRAIN/DEV MISMATCH	BIAS	BIAS + DATA MISMATCH

ADDRESSING DATA MISMATCH

EX. CAR GPS · TRAINING DATA IS 10,000H OF GENERAL SPEECH DATA

1. CARRY OUT MANUAL ERROR ANALYSIS TO UNDERSTAND THE DIFFERENCE (EX NOISE, STREET NUMBERS)
2. TRY TO MAKE TRAIN MORE SIMILAR TO DEV OR GATHER MORE DEV-LIKE TRAIN-DATA



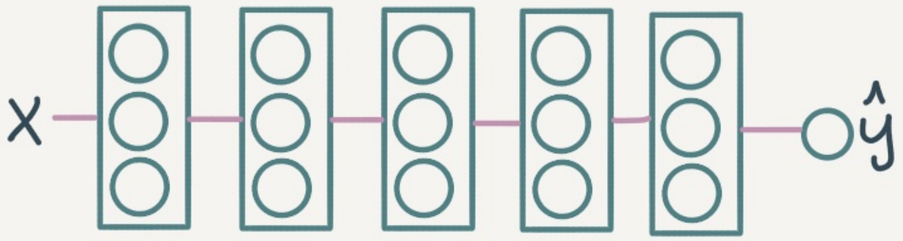
NOTE BE CAREFUL · IF YOU ONLY HAVE 1 HR OF CAR NOISE & APPLY IT TO 10K HR SPEECH YOU MAY OVERFIT TO THE CAR NOISE

STRUCTURING ML PROJECTS · COURSERA

EXTENDED LEARNING

TRANSFER LEARNING

PROBLEM: YOU WANT TO CLASSIFY SOME MEDICAL IMG. YOU HAVE AN NN THAT CLASSIFIES CATS



NN TO CLASSIFY CATS/DOGS/CARS

OPTION 1: YOU ONLY HAVE A FEW RADIOLOGY IMAGES

SOLUTION: INIT W. WEIGHTS FROM CAT NN ONLY RETRAIN LAST LAYER(S) ON RADIOLOGY IMAGES

OPTION 2 YOU HAVE LOTS OF RADIOLOGY IMG.

SOLUTION: INIT WITH WEIGHTS FROM CAT NN RETRAIN ALL LAYERS

TESS NOTES

THIS IS MICROSOFT CUSTOM VISION

MULTI TASK LEARNING

TRAINING ON MULT. TASKS AT ONCE

DETECT
CAR
STOPSIGN
PEDESTR.
TRAFFIC LIGHT

$$\hat{y} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

UNLIKE SOFMAX · MANY THINGS CAN BE TRUE

$$\text{COST: } J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{4} \mathcal{L}(\hat{y}_j^{(i)}, y_j^{(i)})$$

SUMMING OVER ALL OUTP OPTIONS

WE COULD HAVE JUST TRAINED 4 NNS INSTEAD BUT.. MT LEARNING MAKES SENSE WHEN

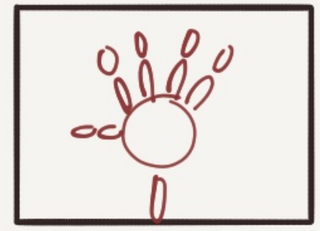
A. THE LEARNING DATA YOU HAVE FOR THE DIFF TASKS IS QUITE SIMILAR – & THE AMOUNTS (EG. 1K CARS, 1K STOP SIGNS)

B. THE SUM OF THE DATA ALLOWS YOU TO TRAIN A BIG ENOUGH NN TO DO WELL ON ALL TASKS

IN REALITY TRANSFER LEARNING IS USED MORE OFTEN

END-TO-END LEARNING

FROM X-RAY OF CHILDS HAND TELL ME THE AGE OF THE CHILD



TYPICAL SOLN:

1. LOCATE BONES TO FIND LENGTHS USING ML
2. TRAIN MODEL TO PREDICT AGE BASED ON BONELENGTH

END-TO-END

RADIOLOGY IMG → CHILD AGE

PROS:

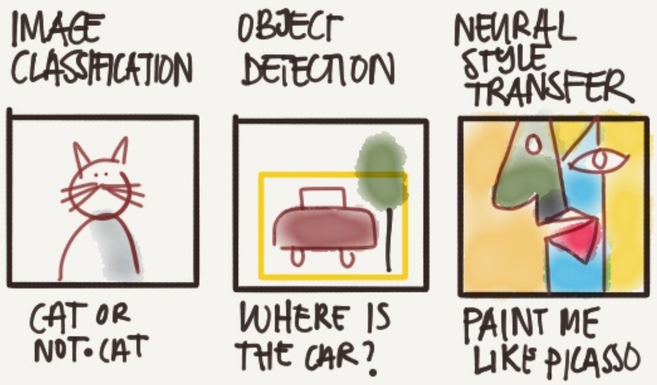
- LET'S THE DATA SPEAK (MAYBE IT FINDS RELATIONS WE'RE UNAWARE OF)
- LESS HAND-DESIGNING OF COMPONENTS NEEDED

CONS:

- NEEDS LARGE AMTS OF LABLED DATA (X→Y)
- EXCLUDES POTENTIALLY USEFUL HAND-MADE COMPONENTS

CONVOLUTION
FUNDAMENTALS

COMPUTER VISION



PROBLEM: IMAGES CAN BE BIG

1000 × 1000 × 3 (RGB) = 3M

WITH 1000 HIDDEN UNITS WE NEED 3M * 1000 = 3B PARAMS

SOLUTION: USE CONVOLUTIONS

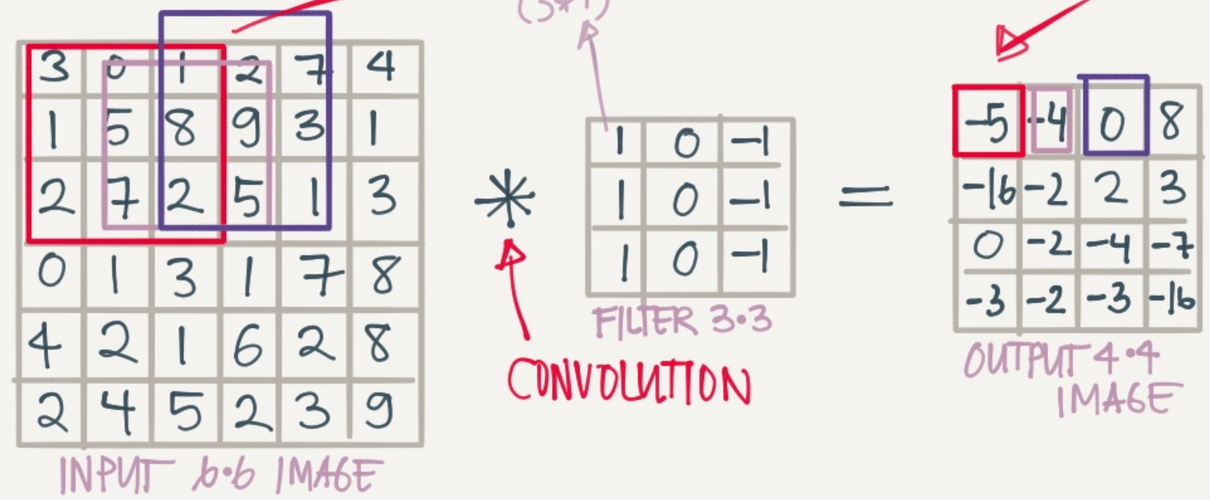
IT'S LIKE SCANNING OVER YOUR IMG WITH A MAGNIFYING GLASS OR FILTER

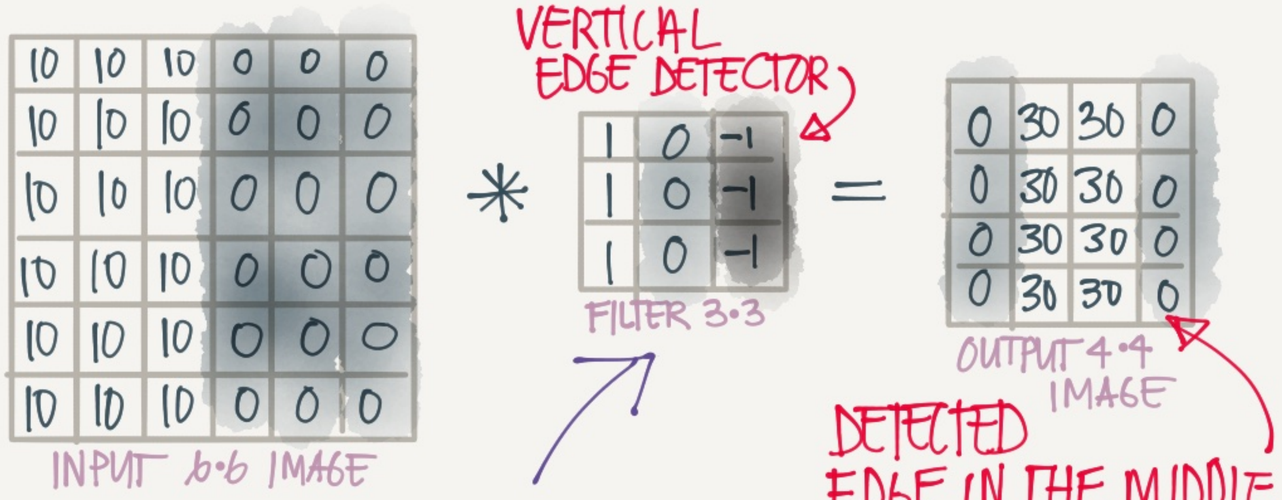
TESS NOTE

ALSO SOLVES THE PROBLEM THAT THE CAT IS NOT ALWAYS IN THE SAME LOCATION IN THE IMG

CONVOLUTIONAL NEURAL NETS · COURSERA

CONVOLUTION





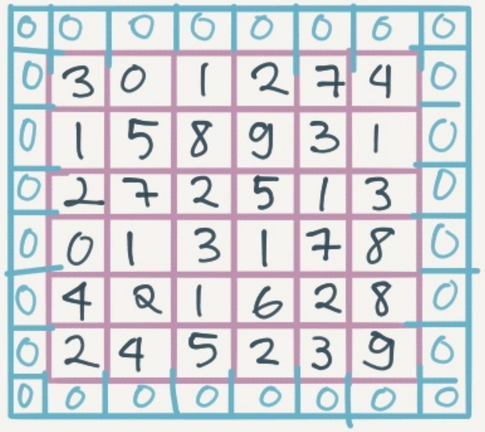
WE COULD HARD-CODE FILTERS · JUST LIKE WE CAN HARD-CODE HEURISTIC RULES.... BUT.... A MUCH BETTER WAY IS TO TREAT THE FILTER # AS PARAMS TO BE LEARNED

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

PADDING

PROBLEM: IMAGES SHRINK
6x6 → 3x3 → 4x4
PROBLEM: EDGES GET LESS 'LOVE'

SOLUTION: PAD W. A BORDER OF 0s BEFORE CONVOLVING



TWO COMMONLY USED PADDING OPTIONS
(HOW MUCH TO PAD)

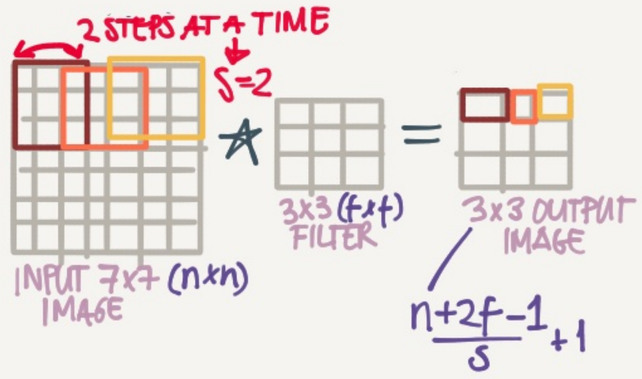
'VALID' ⇒ P = 0 NO PADDING

'SAME' ⇒ P = $\frac{f-1}{2}$ OUTPUT SIZE = INPUT SIZE

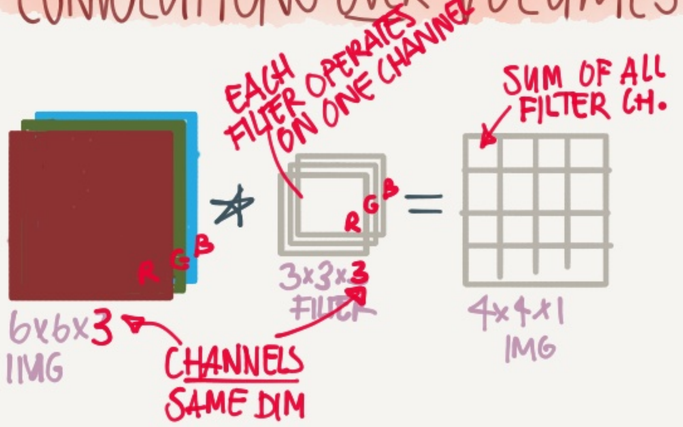
FILTER SIZE

STRIDE

WHAT PACE YOU SCAN WITH



CONVOLUTIONS OVER VOLUMES



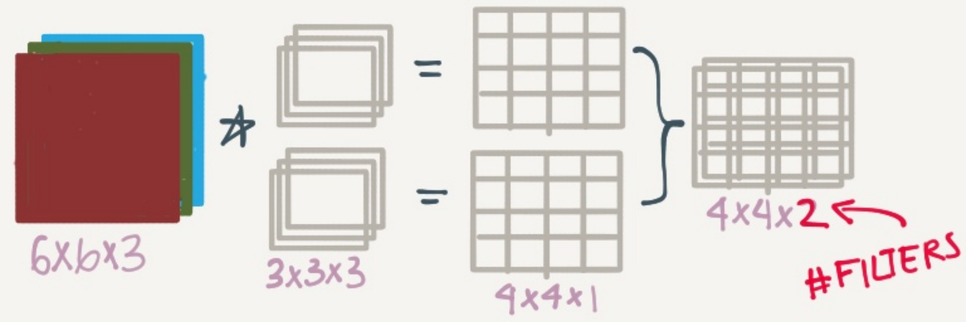
THIS ALLOWS US TO DETECT FEATURES IN COLOR IMAGES FOR EXAMPLE

MAYBE WE WANT TO FIND ALL EDGES OR MAYBE ORANGE BLOBS

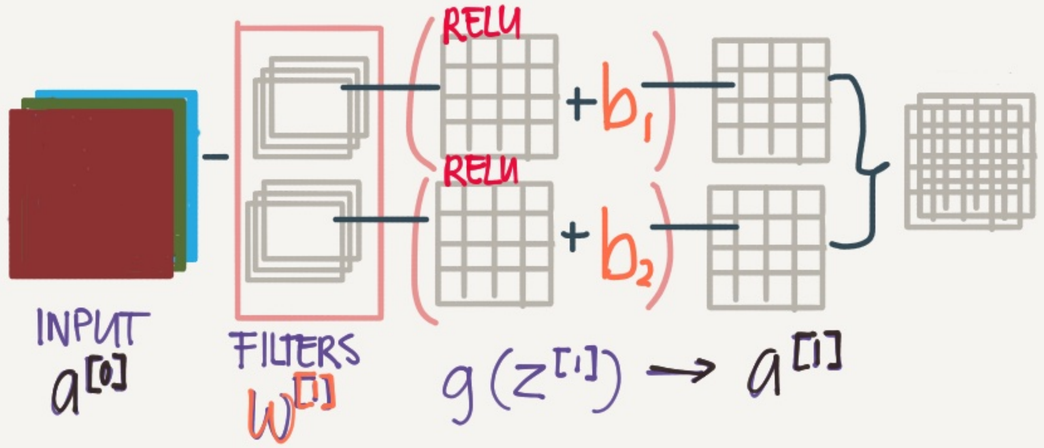
NOTE: ALL CONVOLUTION IDEAS CAN BE APPLIED TO 1D AS WELL LIKE EKG SIGNALS · AND 3D LIKE CT-SCANS

MULTIPLE FILTERS

DETECTING MULTIPLE FEATURES AT A TIME



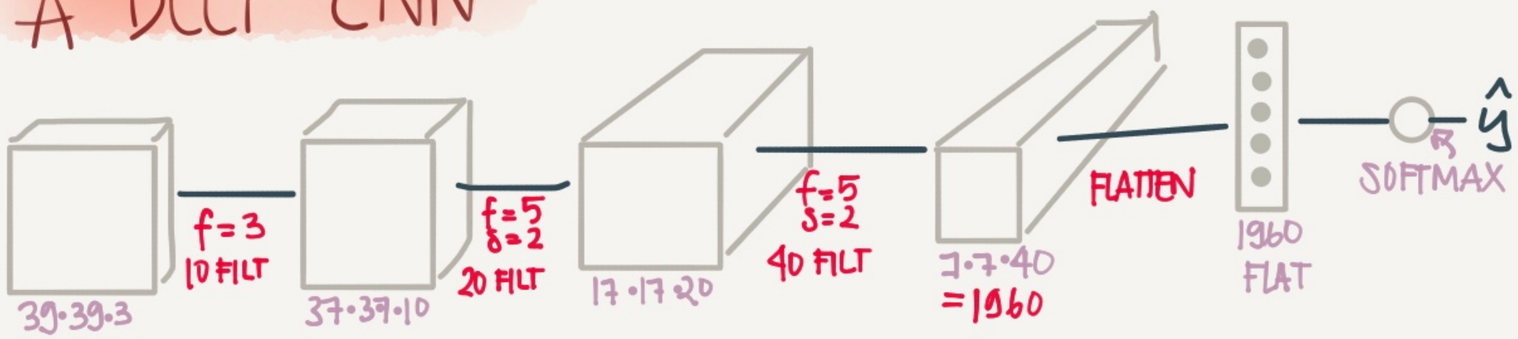
ONE CONV. NET LAYER



NOTE IT DOESN'T MATTER HOW BIG THE INPUT IS - THE LEARNABLE PARAMS W & b ONLY DEPEND ON THE # OF FILTERS AND THEIR SIZES.

$W = 3 \cdot 3 \cdot 3 \cdot 2 = 54$
$b = 2$
} 56 PARAMS TO LEARN

A DEEP CNN



A LOT OF THE WORK IS FIGURING OUT HYPERPARAMS
= #FILTERS, STRIDE, PADDING ETC

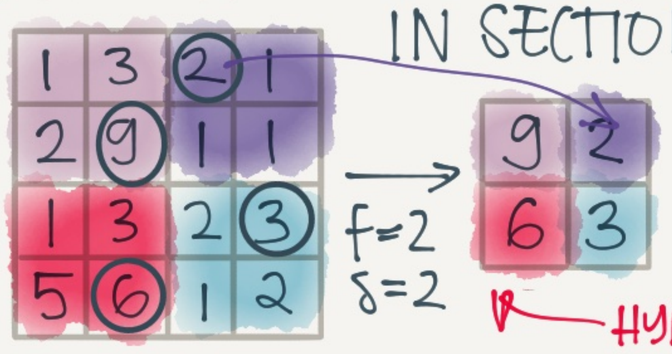
TYPICALLY SIZE → TREND DOWN
FILTERS → TREND UP

TYPICAL CONV.NET LAYERS

CONVOLUTION
POOLING
FULLY CONNECTED

POOLING (MAX)

FIND MAX VAL IN SECTION

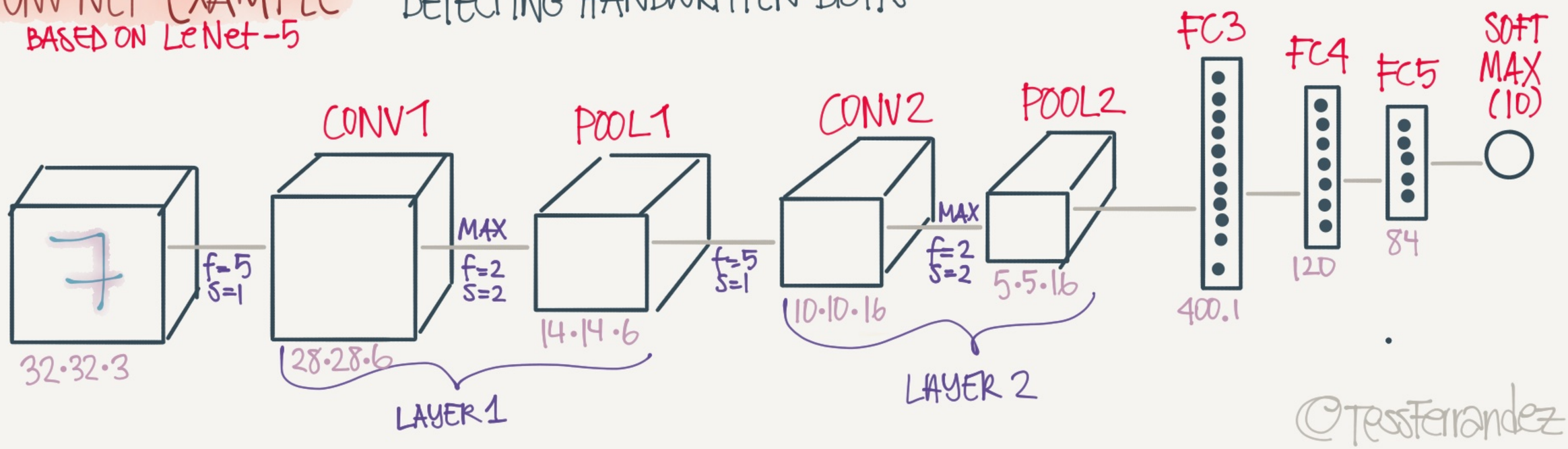


- REDUCES SIZE OF REPRES.
- SPEEDS UP COMPUTATION
- MAKES SOME OF THE DETECTED FEAT. MORE ROBUST

CONV NET EXAMPLE

BASED ON LeNet-5

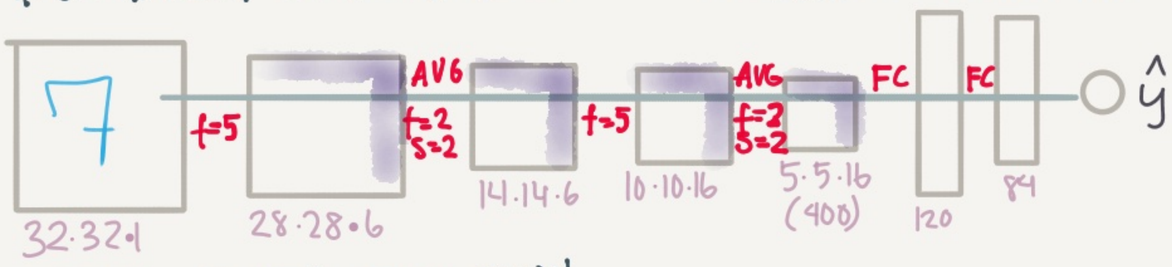
DETECTING HANDWRITTEN DIGITS



CLASSIC CONV. NETS

LeNet-5

DOCUMENT CLASSIFICATION ≈60k PARAMETERS



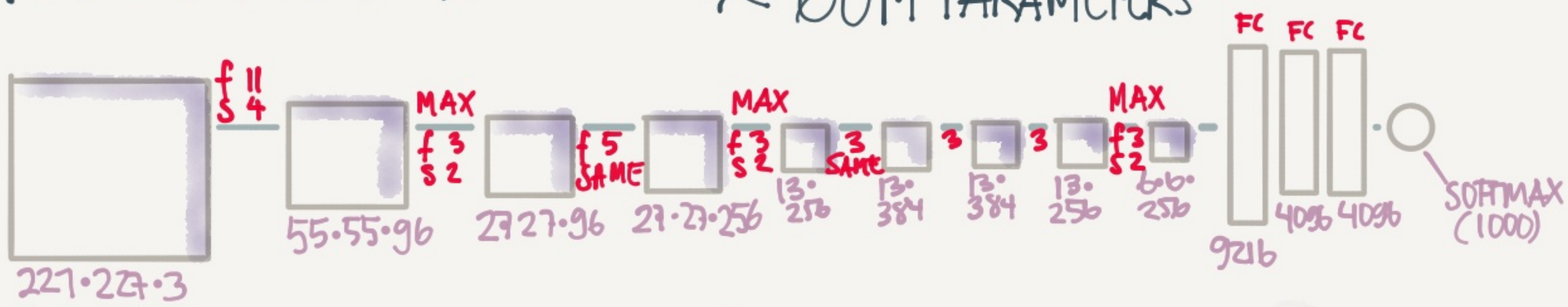
TRENDS: HEIGHT/WIDTH GO DOWN
CHANNELS GO UP

COMMON PATTERN: A COUPLE OF CONV(1+)/POOL LAYERS FOLLOWED BY A FEW FC

OLD STUFF: USED AVG POOLING INST. OF MAX
PADDING WAS NOT VERY COMMON
IT USED SIGMOID/TANH INST OF RELU

AlexNet

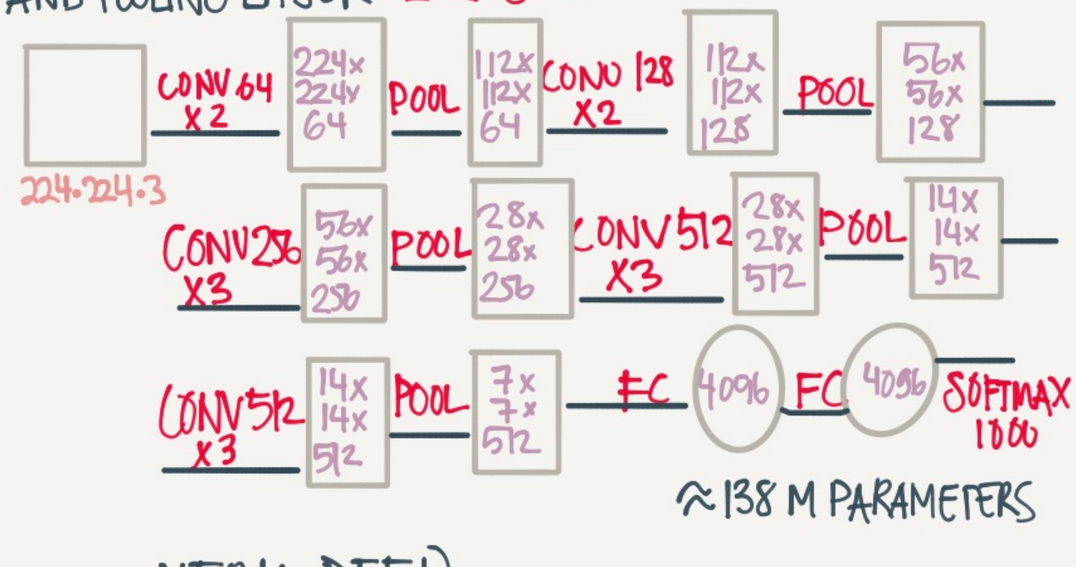
IMAGE CLASSIFICATION ≈60M PARAMETERS



- SIMILAR TO LeNet BUT MUCH BIGGER
- USES RELU
- THE NN THAT GOT RESEARCHERS INTERESTED IN VISION AGAIN

VGG-16

ALL CONV. LAYERS HAVE SAME #PARAMS
f=3x3 S=1 P=SAME
AND POOLING LAYER 2x2 S=2



≈138 M PARAMETERS

- VERY DEEP
- EASY ARCHITECTURE
- #FILTERS DOUBLE 64,128,256,512

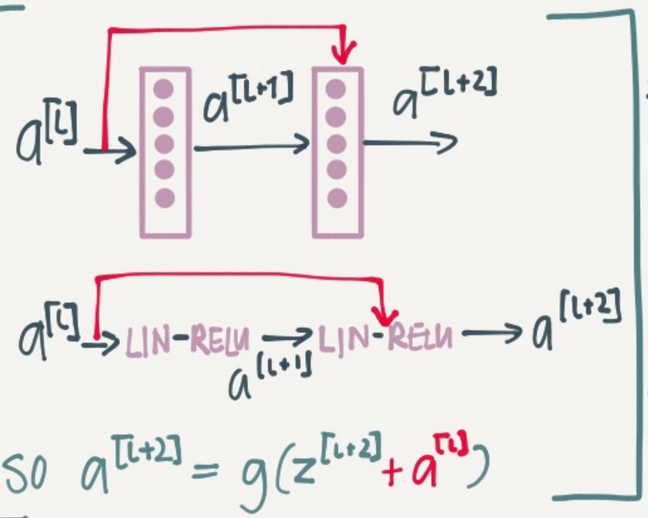
CONVOLUTIONAL NEURAL NETS · COURSERA

SPECIAL NETWORKS

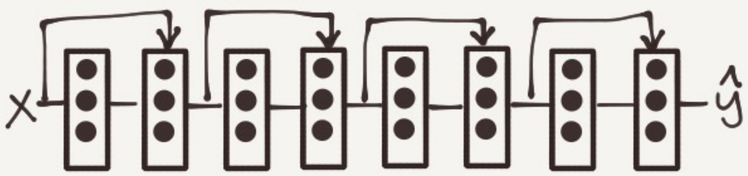
ResNets

PROBLEM: DEEP NN OFTEN SUFFER PROBLEMS W VANISHING OR EXPLODING GRADIENTS

SOLUTION RESIDUAL NETS



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

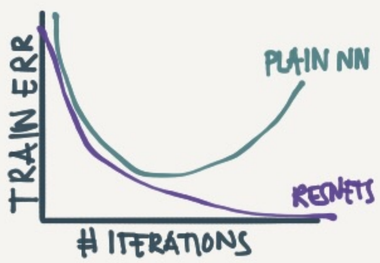


VERY DEEP NN

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

WORST CASE: W & b HAVE VANISHED TO 0 ⇒
$a^{[l+2]} = g(a^{[l]}) = a^{[l]}$
SO EXTRA LAYERS LEARNED NOTHING BUT DIDN'T HURT

BEST CASE: LAYERS LEARNED SOMETHING



CONVOLUTIONAL NEURAL NETS · COURSERA

NETWORK IN NETWORK (1×1 CONVOLUTION)

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

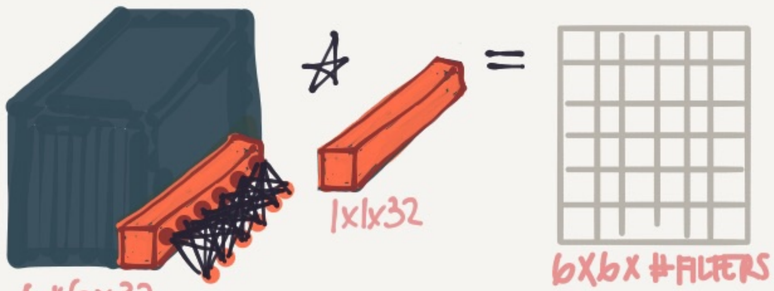
* [2] =

12	10	6	4
8	2	18	10
10	16	4	8
0	6	12	2

1×1 CONVOLUTION

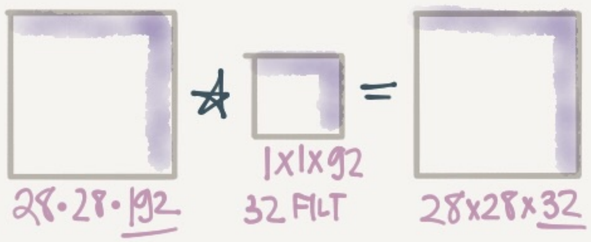
IT SEEMS PRETTY USELESS, BUT IT ACTUALLY SERVES 2 PURPOSES

1. NETWORK IN A NETWORK



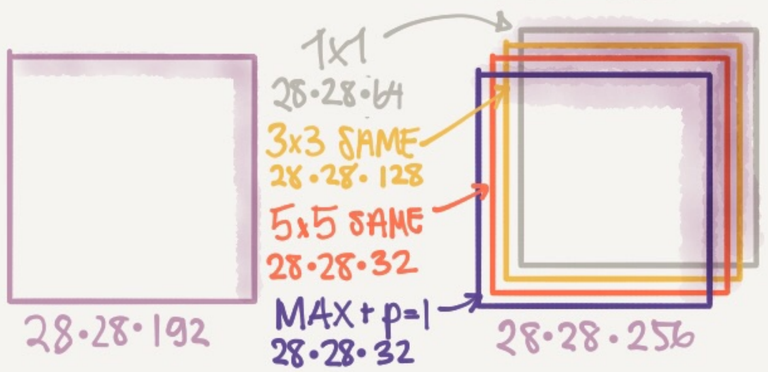
LEARNS COMPLEX, NON-LINEAR RELATIONSHIPS ABOUT A SLICE OF A VOLUME

2. REDUCING # CHANNELS



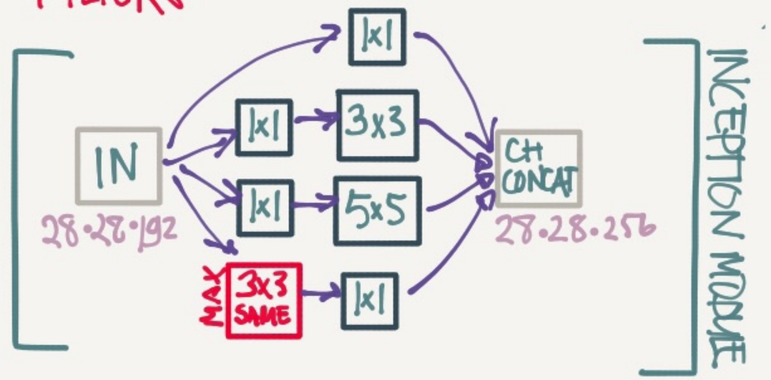
INCEPTION NETWORKS

INSTEAD OF CHOOSING A 1×1, 3×3, 5×5 OR A POOLING LAYER - CHOOSE ALL



PROBLEM: VERY EXPENSIVE TO COMPUTE

SOLUTION: SHRINK THE # CHANNELS W A 1×1 CONV BEFORE APPLYING ALL THE FILTERS



TO BUILD AN INCEPTION NETWORK YOU MAINLY STACK A BUNCH OF INCEPTION MODULES

INCEPTION THE MOVIE

PRACTICAL ADVICE

USE OPEN SOURCE IMPLEMENTATIONS

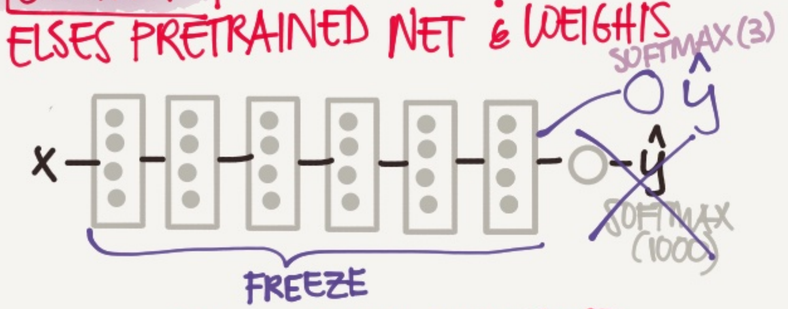
SOME OF THE PAPERS ARE HARD TO IMPLEMENT FROM SCRATCH - USING OS YOU CAN REUSE OTHER PPLS WORK

DON'T FORGET TO CONTRIBUTE

TRANSFER LEARNING

WANT TO TRAIN A CLASSIFIER FOR YOUR CATS BUT DON'T HAVE ENOUGH PICTURES

SOLUTION DOWNLOAD SOMEONE ELSES PRETRAINED NET & WEIGHTS

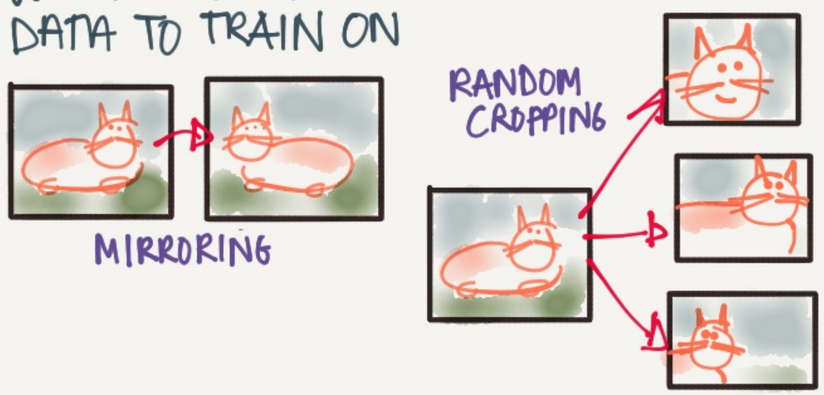


FREEZE THE PARAMS, AND JUST REPLACE THE SOFTMAX LAYER WITH YOUR OWN & TRAIN

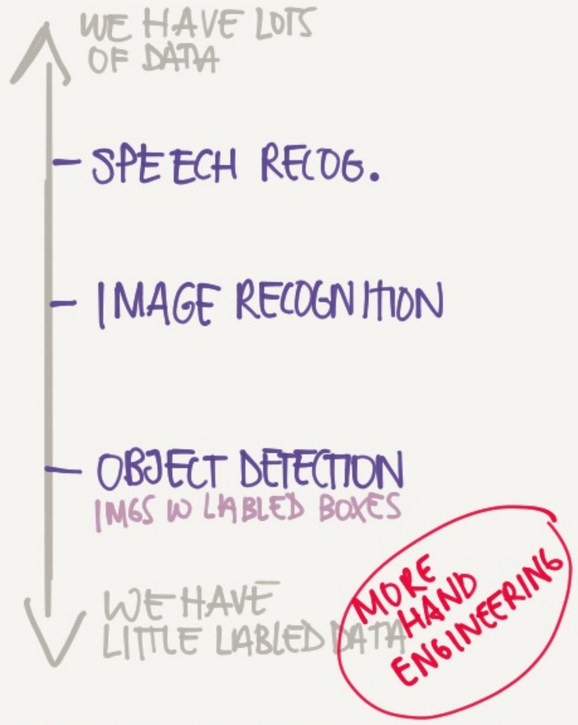
IF YOU HAVE MORE PICS • RETRAIN A MORE OF THE LATER LAYERS (MAYBE INITIALIZING WITH THE PRETRAINED WEIGHTS)

DATA AUGMENTATION

WE ALMOST ALWAYS NEED MORE DATA TO TRAIN ON



STATE OF COMPUTER VISION



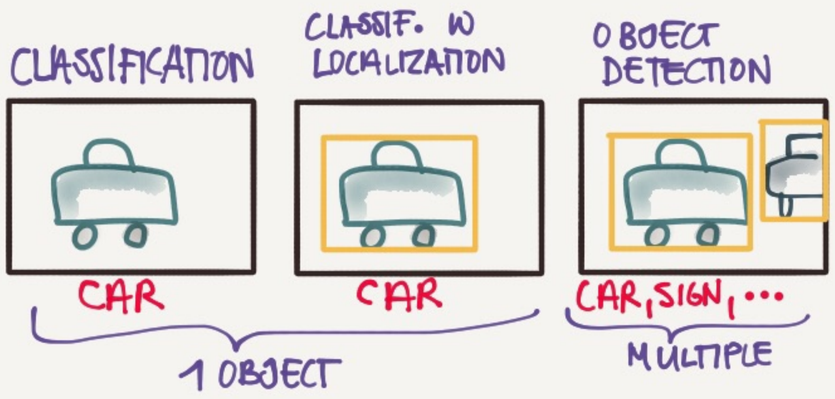
TIPS FOR DOING WELL ON BENCHMARKS/COMPETITIONS

- ENSEMBLING
  - AVG OUTPUTS FROM MULT NN
- MULTI-CROP AT TEST TIME
  - AVG OUTPUTS FROM MULTIPLE CROPS OF THE IMAGE

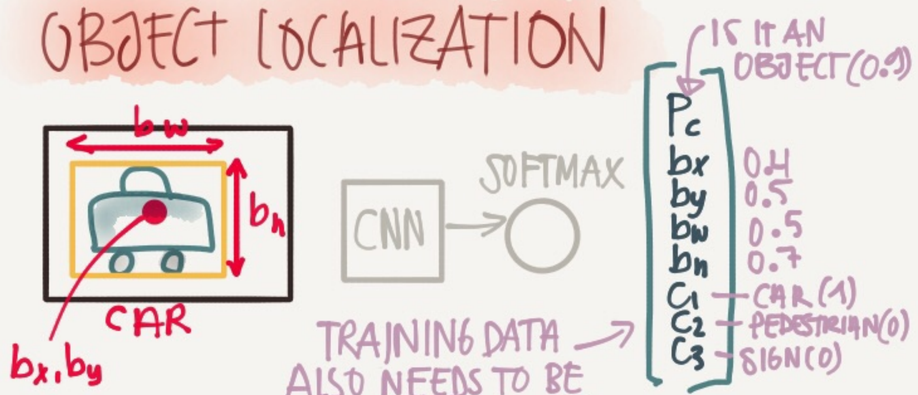
IN PRACTICE THEY ARE NOT USED IN PRODUCTION BECAUSE THEY ARE COMPUTE & MEM EXPENSIVE

CONVOLUTIONAL NEURAL NETS · COURSERA

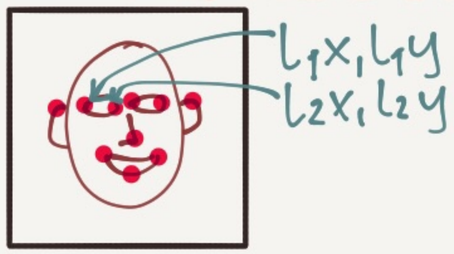
DETECTION ALGORITHMS



OBJECT LOCALIZATION



LANDMARK DETECTION



$$\hat{y} = \begin{bmatrix} \text{FACE} \\ l_{1x} \\ l_{1y} \\ l_{2x} \\ l_{2y} \\ \vdots \end{bmatrix}$$

TO DETECT LANDMARKS IN THE FACE (CORNER OF MOUTH ETC) LABEL THE X,Y COORDS OF THE LANDMARK

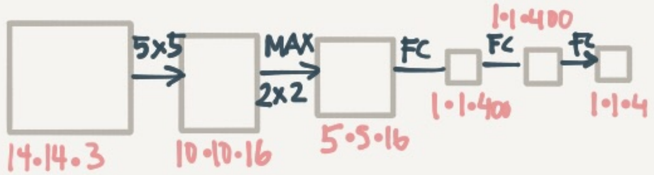
USED FOR SENTIMENT ANALYSIS & FOR EFFECTS LIKE PLACING CROWN ON HEAD ETC.

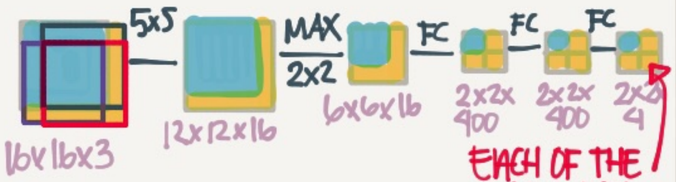
SLIDING WINDOWS DETECTION

1. CREATE TIGHTLY CROPPED IMG OF CARS (LOTS)
2. SLIDE A WINDOW OVER THE IMG. & CLASSIFY THIS WINDOW CAR(1/0) AGAINST YOUR OTHER CARS
3. REPEAT WITH SLIGHTLY LARGER WINDOW SIZE

PROBLEM: VERY EXPENSIVE (TO COMPUTE)

SINCE ADJ WINDOWS SHARE A LOT OF THE COMPUTATIONS WE CAN DO THIS MUCH CHEAPER W CONVOLUTIONS

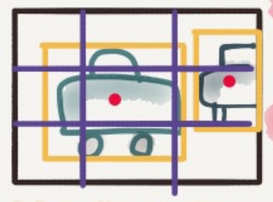




EACH OF THE 4 VALS ARE RESULTS FOR EACH OF THE 4 WINDOWS

NOW WE JUST PASS THROUGH ONCE AND CALC ALL AT THE SAME TIME

YOLO: You Only Look Once

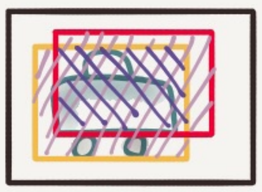


1. SPLIT IMG INTO X(9) GRID CELLS
2. FOR EACH CELL, SAY IF IT CONTAINS CAR+ BOUNDING BOX (IF CELL CONTAINS THE MID POINT) SAME AS OBJECT LOCALIZATION

IN PRACTICE WE MIGHT HAVE A 19x19 GRID

$3 \times 3 = 3 \times 3 \times 8$

HOW DO YOU KNOW HOW GOOD IT IS?



HOW GOOD IS THE RED SQUARE?

$$IOU = \frac{\text{SIZE OF}}{\text{SIZE OF}}$$

INTERSECTION OVER UNION

GENERALLY - IF $IOU \geq 0.5$ IT IS REGARDED AS CORRECT

WHAT IF MULTIPLE SQUARES CLAIM THE SAME CAR?

NON-MAX SUPRESSION

IF TWO BOUNDING BOXES HAVE A HIGH IOU - PICK THE ONE W HIGHEST P_c - GET RID OF THE REST.

ANCHOR BOXES

ANCHOR BOXES LET YOU ENCODE MULTIPLE OBJECTS IN THE SAME SQUARE

FACE RECOGNITION

FACE VERIFICATION

IS THIS PETE?

99% ACC ⇒ PRETTY GOOD

FACE RECOGNITION

WHO IS THIS? (OUT OF K PERSONS)

IF K = 100 NEED MUCH HIGHER THAN 99%

ONE-SHOT LEARNING

NEED TO BE ABLE TO RECOGNIZE A PERSON EVEN THOUGH YOU ONLY HAVE ONE SAMPLE IN YOUR DB.

YOU CAN'T TRAIN A CNN WITH A SOFTMAX (EACH PERSON) BECAUSE

(A) YOU DON'T HAVE ENOUGH SAMPLES

(B) IF A NEW PERSON JOINS YOU NEED TO RETRAIN THE NETWORK

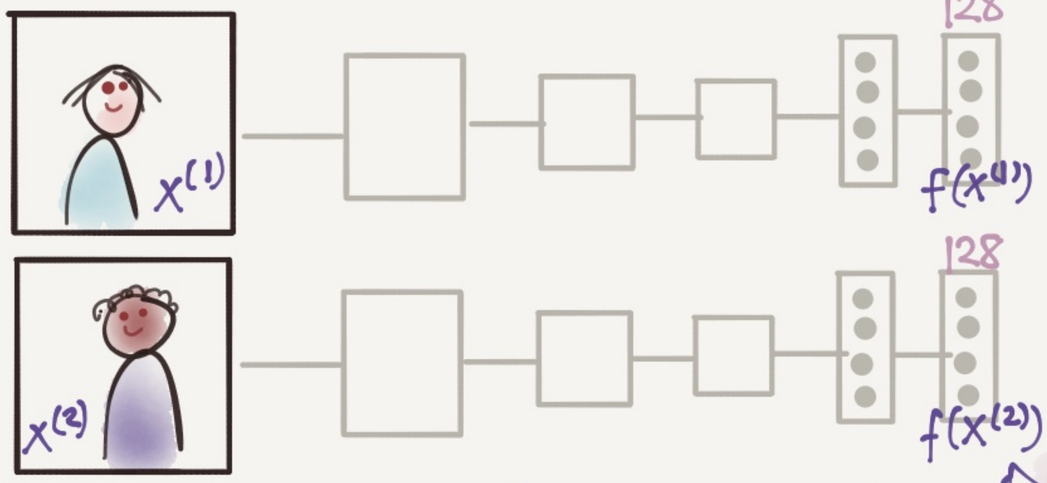
SOLUTION LEARN A SIMILARITY FUNCTION

$d(\text{img1}, \text{img2}) = \text{degree of difference}$

BUT HOW DO YOU LEARN THIS?

SIAMESE NETWORK

DeepFace



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

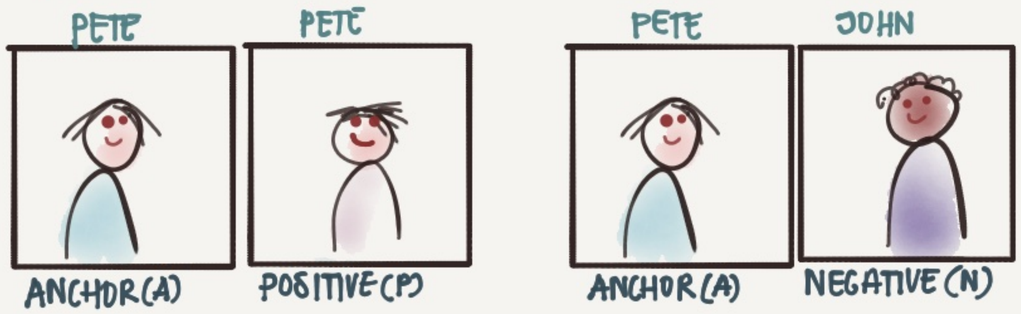
LEARN THE PARAMS OF THE NN SUCH THAT

- IF $x^{(i)}, x^{(j)}$ ARE THE SAME PERSON · $d(x^i, x^j) \Rightarrow$ SMALL
- IF $x^{(i)}, x^{(j)}$ ARE DIFFERENT PEOPLE · $d(x^i, x^j) \Rightarrow$ LARGE

WE CAN ACCOMPLISH THIS WITH THE TRIPLET LOSS FUNCTION

TRIPLET LOSS

FaceNet



WANT $\underbrace{\| f(A) - f(P) \|^2}_{d(A,P)} \leq \underbrace{\| f(A) - f(N) \|^2}_{d(A,N)} \Rightarrow d(A,P) - d(A,N) \leq 0$

BUT WE WANT A GOOD MARGIN, SO...

$d(A,P) - d(A,N) + \alpha \leq 0$

TIP

PRECOMPUTE ENCODINGS FOR PPL IN YOUR DB, SO YOU DONT HAVE TO SAVE IMAGES & COMPUTE ENCODINGS AT RUN. TIME

HOW DO WE CHOOSE TRIPLETS TO TRAIN ON?

- IF A/P ARE VERY SIMILAR, & A/N ARE VERY DIFFERENT TRAINING IS VERY EASY.

SELECT A/N THAT ARE PRETTY SIMILAR TO TRAIN A GOOD NET

SOME BIG COMPANIES HAVE ALREADY TRAINED NETWORKS ON LARGE AMTS OF PHOTOS SO YOU MAY JUST WANT TO REUSE THEIR WEIGHTS

CONVOLUTIONAL NEURAL NETS · COURSERA

NEURAL STYLE TRANSFER

WE CAN VISUALIZE WHAT A NETWORK LEARNS BY LOOKING AT WHAT IMAGES (PARTS) ACTIVATED EACH UNIT MOST

BUT HOW DOES THIS HELP US GENERATE AN IMAGE IN THE STYLE OF ANOTHER?

IDEA:

1. GENERATE A RANDOM IMG
2. OPTIMIZE THE COST FUNCTION

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

$J_{CONTENT}(C,G)$: HOW SIMILAR ARE C & G

$J_{STYLE}(S,G)$: HOW SIMILAR ARE S & G

3. UPDATE EACH PIXEL

CONTENT COST FUNCTION

- USE A PRE-TRAINED CONVNET (EX VGG)
- SELECT A HIDDEN LAYER SOMEWHERE IN THE MIDDLE
  LATER ⇒ COPIES LARGER FEATURES
- LET $a^{[l](C)}$ & $a^{[l](G)}$ BE THE ACTIVATIONS
- IF $a^{[l](C)}$ & $a^{[l](G)}$ ARE SIMILAR THEY HAVE SIMILAR CONTENT
  BECAUSE THEY BOTH TRIGGER THE SAME HIDDEN UNITS

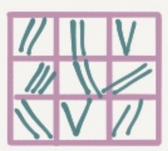
HOW DO WE TELL IF THEY ARE SIMILAR?

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

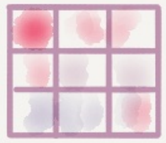
CAPTURING THE STYLE

USING THE STYLE IMG AND THE ACTIVATIONS IN A LAYER.
LOOK THROUGH THE ACTIVATIONS IN THE DIFFERENT CHANNELS TO SEE HOW CORRELATED THEY ARE

WHEN WE SEE PATTERNS LIKE THIS



DO WE USUALLY SEE IT WITH PATCHES LIKE THESE?



STYLE MATRIX

CREATE A MATRIX OF HOW CORRELATED THE ACTIVATIONS ARE, FOR EACH POS (x,y) & CHANNEL PAIR (K, K') FOR THE STYLE IMG & GENERATED

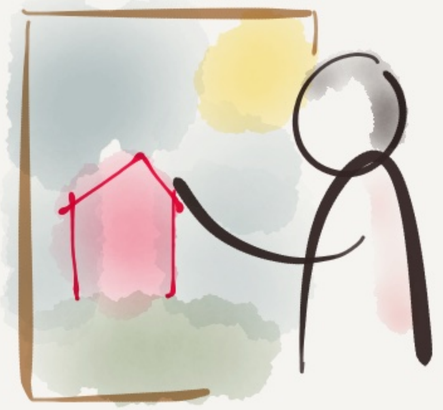
$$G_{KK'} = \sum_{i=1}^{n_H} \sum_{j=1}^{n_W} a_{ijk} \cdot a_{ijk'}$$

THE STYLE COST FUNCTION

$$J(S,G) = \| G^{(S)} - G^{(G)} \|_F^2$$

FROBENIUS NORM

TO GET MORE VISUALLY PLEASING IMAGES IF YOU CALC $J(S,G)$ OVER MULTIPLE LAYERS



RECURRENT
NEURAL NETWORKS

SEQUENCE MODELS · COURSERA

SEQUENCE PROBLEMS

IN	OUT	PURPOSE
	THE QUICK BROWN FOX JUMPED...	SPEECH RECOGNITION
∅		MUSIC GENERATION
THERE IS NOTHING TO LIKE IN THIS MOVIE		SENTIMENT CLASSIFICATION
AGCCCCTGTG AGGAACTAG	AGCCCCTGTG AGGAACTAG	DNA SEQUENCE ANALYSIS
Voulez-vous chanter avec moi?	Do you want to sing with me?	MACHINE TRANSLATION
	RUNNING	VIDEO ACTIVITY RECOGNITION
Yesterday Harry Potter met Hermoine Granger	Yesterday Harry Potter met Hermoine Granger	NAME ENTITY RECOGNITION

NAME ENTITY RECOGNITION

X = HARRY POTTER AND HERMOINE $T_x = 9$ (9 WORDS)
$x^{<1>}$ $x^{<2>}$ ...

GRANGER INVENTED A NEW SPELL

y = 1 1 0 1 $T_y = T_x$
$y^{<1>}$ $y^{<2>}$ ...
1 0 0 0 0

EXAMPLE OF A PROBLEM WHERE EVERY $x^{<i>}$ HAS AN OUTPUT $y^{<i>}$

HOW DO WE REPRESENT WORDS?

CREATE A VOCABULARY (EG 10K MOST COMMON WORDS IN YOUR TEXTS · OR DOWNLOAD EXISTING)

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

EACH WORD IS A ~~ONE-HOT~~ VECTOR

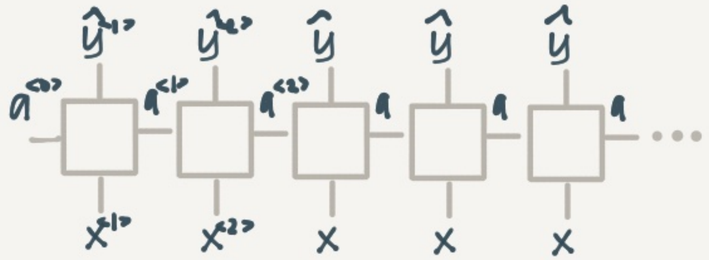
HARRY = [0 0 ... 1 (4075) ... 0]

WE COULD USE A STANDARD NETWORK BUT...

(A) INPUT & OUTPUTS CAN HAVE DIFFERENT LENGTHS IN DIFF EXAMPLES

(B) WE DON'T SHARE FEATURES LEARNED ACROSS DIFFERENT POSITIONS

RECURRENT NEURAL NET (RNN)



PREVIOUS RESULTS ARE PASSED IN AS INPUTS SO WE GET CONTEXT.

$a^{<1>} = g_1(W_a[a^{<0>}, x^{<1>}] + b_a)$ TANH/RELU

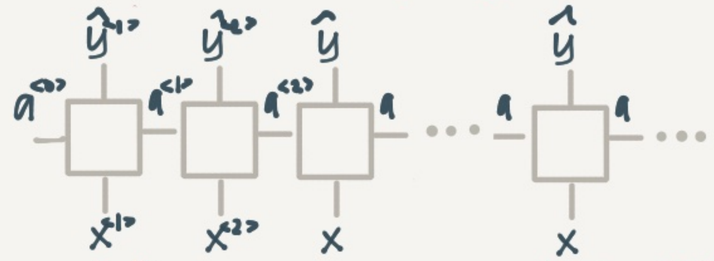
$\hat{y}^{<1>} = g_2(W_{ya}\, a^{<1>} + b_y)$ SIGMOID

THE SAME W & b ARE USED IN ALL TIME STEPS

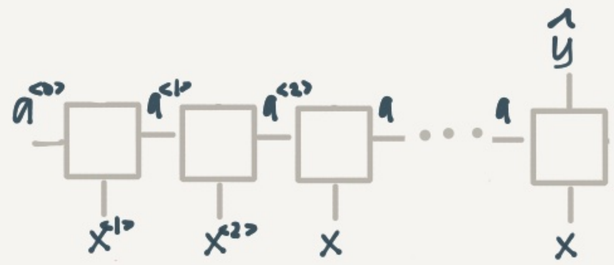
THE LOSS WE OPTIMIZE IS THE SUM OF $\mathcal{L}(\hat{y}, y)$ FROM 1–T

DIFFERENT TYPES OF RNN

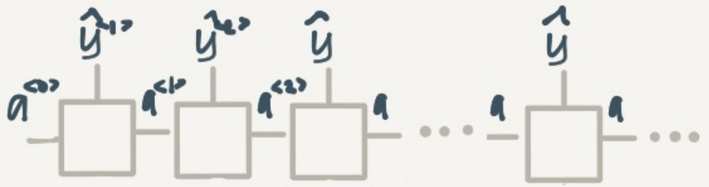
MANY-TO-MANY $T_x = T_y$



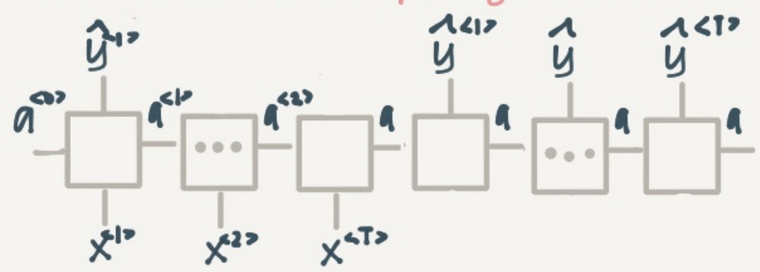
MANY-TO-ONE EX. SENTIMENT ANALYSIS



ONE-TO-MANY · MUSIC GENERATION



MANY-TO-MANY $T_x \neq T_y$ TRANSLATION



SEQUENCE MODELS · COURSERA

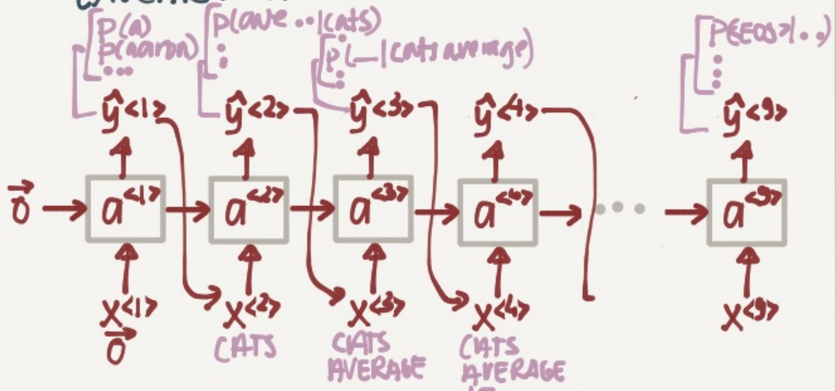
MORE ON RNNs

LANGUAGE MODELLING

HOW DO YOU KNOW IF SOMEONE SAID
THE APPLE AND PAIR SALAD OR
THE APPLE AND PEAR SALAD?

THE PURPOSE OF A LANG. MODEL IS TO CALCULATE THE PROBABILITIES

EX. CATS AVERAGE 15 HOURS OF SLEEP A DAY



SO GIVEN: CATS AVERAGE 15 WHAT IS THE PROB. THE NEXT WORD IS HOURS?

SAMPLING SENTENCES

1. TRAIN ON ALL HARRY POTTER BOOKS.
2. RANDOMLY SELECT A WORD (ONE OF THE TOP WORDS) (EX. THE)
3. PASS THIS INTO THE NEXT TIMESTAMP AND SAMPLE A NEW WORD
4. REPEAT UNTIL X WORDS OR YOU REACHED <EOS>

CAN DO AT CHARACTERLEVEL AS WELL

YAY! YOU ARE NOW YOUR OWN J.K. ROWLING

VANISHING GRADIENTS

THE CAT, WHO ALREADY ATE APPLES AND ORANGES AND A FEW MORE THINGS BLA BLA WAS FULL
THE CATS, WHO ALREADY ATE ... WERE FULL

NEED TO REMEMBER SING/PLURAL FOR A LONG TIME

SINCE LONG SENTENCE ⇒ DEEP RNN
WE GET THE VANISHING GRADIENTS PROB WE HAVE IN STANDARD NNs – I.E THE GRADIENTS FOR CAT/CATS HAVE LITTLE OR NO EFFECT ON WAS/WERE.

NOTE SOMETIMES YOU SEE EXPLODING GRAD (AS OVERFLOW NaN) BUT THIS IS EASILY FIXED WITH GRADIENT CLIPPING

GATED RECURRENT UNIT GRU

HELPS RECALL IF CAT WAS SING. OR PLURAL

GRU

THE GRU ACTS AS A MEMORY — AT EVERY TIMESTEP IT CALCULATES A NEW $\tilde{c}$ TO STORE AND A GATE Γ_u DECIDES TO UPDATE c TO $\tilde{c}$ OR NOT

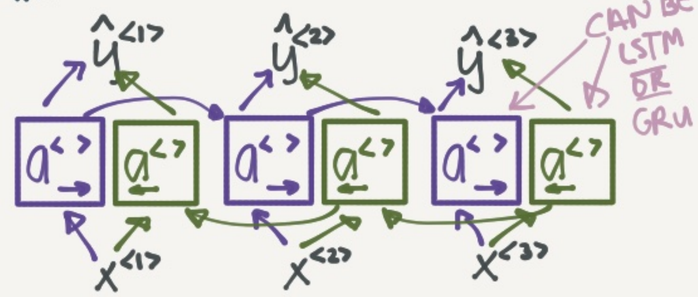
LONG SHORT TERM MEMORY (LSTM)

THE LSTM IS A VARIATION ON THE SAME THEME AS GRU BUT WITH AN ADDITIONAL Γ_F FORGET GATE

BI-DIRECTIONAL RNNs (BRNN)

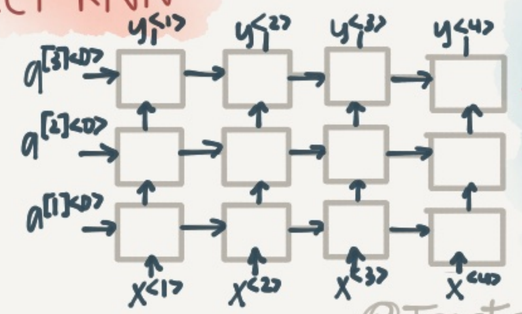
HE SAID, 'TEDDY BEARS ARE ON SALE'
HE SAID, 'TEDDY ROOSEVELT WAS A GREAT PRESIDENT'

PROBLEM: WITHOUT LOOKING FORWARD WE CANT SAY IF TEDDY IS A TOY OR A NAME



ONE DISADVANTAGE IS THAT YOU NEED THE FULL SENTENCE BEFORE YOU BEGIN - SO NOT SUITABLE FOR LIVE SPEECH RECO

DEEP RNN



SINCE THEY ARE ALREADY TEMORALLY DEEP THEY USUALLY DON'T HAVE A LOT OF LAYERS

SEQUENCE MODELS · COURSERA

NLP & WORD EMBEDDINGS

MAN IS TO WOMAN AS KING IS TO QUEEN

PROBLEM: THE ONE-HOT REPR O_{4914} OF APPLE HAS NO INFO ABOUT ITS RELATIONSHIP TO O_{6257} ORANGE

I WANT A GLASS OF ORANGE ____
I WANT A GLASS OF APPLE ____

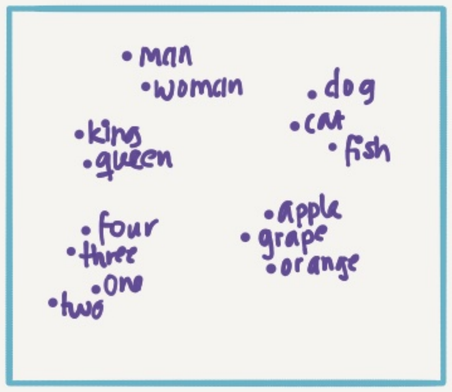
SOLUTION: CREATE A MATRIX OF FEATURES TO DESCRIBE THE WORDS

WORD EMBEDDINGS

	MAN 5391	WOMAN 9853	KING 4914	QUEEN 7157	APPLE 456	ORANGE 6257
GENDER	-1	1	-0.95	0.97	0.00	0.01
ROYAL	0.01	0.02	0.93	0.95	-0.01	0.00
AGE	0.03	0.02	0.7	0.69	0.03	-0.02
FOOD	0.04	0.01	0.02	0.01	0.95	0.97
⋮						

e_{5391}

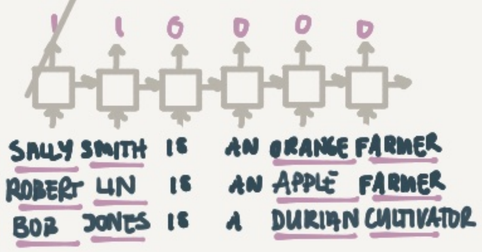
IN REALITY · THE FEATURES ARE LEARNED & NOT AS STRAIGHTFWD AS GENDER/AGE



t-SNE
VISUAL REPRESENT OF 300D WORD EMBEDDINGS

USING WORD EMBEDDINGS

EX. NAME/ENTITY RECOGN



WITH WORD EMBEDDINGS WE UNDERSTAND THAT AN ORANGE FARMER IS A PERSON ⇒ SALLY SMITH = NAME

- APPLE ~ ORANGE ⇒ APPLE FARMER = PERSON
- USING WORD EMBEDDINGS TRAINED ON LOTS OF TEXT WE ALSO GET EMB. FOR MORE UNCOMMON WORDS (DURIAN, CULTIVATOR)

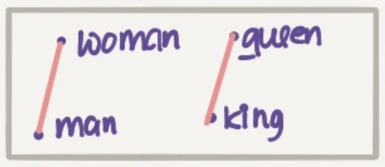
EX. MAN IS TO WOMAN AS KING IS TO ?

E = EMBEDDING MATRIX

	MAN 5391	WOMAN 9853	KING 4914	QUEEN 7157
GENDER	-1	1	-0.95	0.97
ROYAL	0.01	0.02	0.93	0.95
AGE	0.03	0.02	0.7	0.69
FOOD	0.04	0.01	0.02	0.01

e_{man}

$e_{man} - e_{woman} \approx \begin{bmatrix}-2\\0\\0\\0\end{bmatrix}$ VERY SIMILAR $\approx e_{king} - e_{queen}$



FIND MAN→WOMAN
KING→ ?

FIND(w):
$ARG.MAX\ SIM(e_w, e_{KING} - e_{MAN} + e_{WOMAN})$

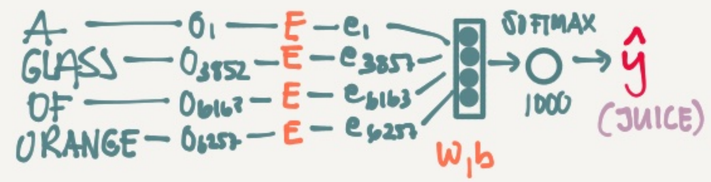
$SIM(u,v) = \frac{u^T v}{\|u\|_2 \|v\|_2}$ COSINE SIMILARITY

LEARNING WORD EMBEDDINGS

HOW DO WE LEARN THE EMBEDDING MATRIX E?

IDEA 1: USING A NEURAL LANG MODEL

I WANT A GLASS OF ORANGE $\hat{y}$



WE CAN USE DIFFERENT CONTEXTS THAN THE LAST 4 WORDS

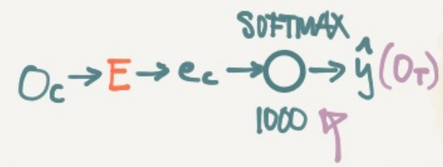
- LAST 4 WORDS
- 4 WORDS LEFT+RIGHT
- LAST 1 WORD
- NEARBY 1 WORD ← SKIPGRAM RANDOM WITHIN EX 5 WORDS

IDEA 2: SKIP-GRAMS WORD2VEC

I WANT A GLASS OF ORANGE JUICE TO GO ALONG WITH MY CEREAL

PICK RANDOM CONTEXT/TARGET PAIRS (WITHIN EX 5 WORDS)

CONTEXT	TARGET
ORANGE	JUICE
ORANGE	GLASS
ORANGE	MY
...	...



NOTE: WHILE THIS SIMPLE NN PREDICTS O_T OUR REAL GOAL IS TO LEARN E

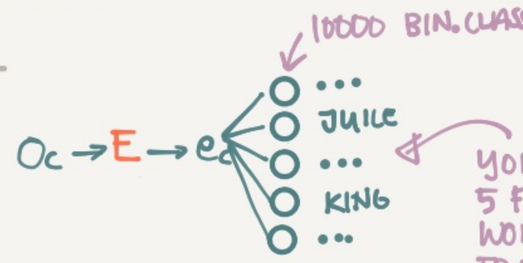
THIS IS VERY COMPUTATIONALLY EXPENSIVE BUT WE CAN OPTIMIZE BY USING A HIERARCHICAL SOFTMAX CLASSIFIER

IDEA: NEGATIVE SAMPLING

1. PICK A CONTEXT/TARGET PAIR AS A POSITIVE EXAMPLE
2. PICK A FEW NEG EXAMPLES CONTEXT + RANDOM

CONTEXT	WORD	TARGET
ORANGE	JUICE	1
ORANGE	KING	0
ORANGE	BOOK	0
ORANGE	THE	0
ORANGE	OF	0

NOTE: SOMETIMES BY CHANCE YOU PICK A POS PAIR · BUT IT DOESN'T MATTER



WORD EMBEDDINGS CONTINUED...

GloVe WORD VECTORS

X_{ij} = #TIMES WORD i (TARGET) APPEARS IN THE CONTEXT OF j (CONTEXT)
(HOW RELATED THEY ARE)

$$\text{MINIMIZE} \sum_{i=1}^{10k} \sum_{j=1}^{10k} f(x_{ij})(\theta_i^T e_j + b_i + b_j' - \log x_{ij})^2$$

$f(x_{ij})$: 0 IF NO CONTEXT
(ALSO HELPS WEIGHING VERY FREQ WORDS (THE, OF...) & VERY INFREQUENT (DURIAN))

EVERYTHING LED UP TO THIS VERY SIMPLE ALGORITHM

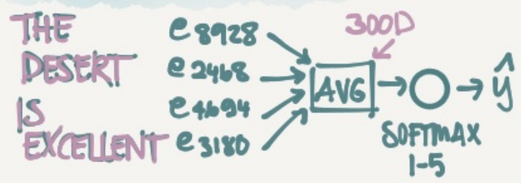
SENTIMENT CLASSIFICATION

X	y
THE DESSERT IS EXCELLENT	★★★★
SERVICE WAS QUITE SLOW	★★
GOOD FOR A QUICK MEAL BUT NOTHING SPECIAL	★★★
COMPLETELY LACKING IN GOOD TASTE GOOD SERVICE AND GOOD AMBIENCE	★

PROBLEM: YOU MAY NOT HAVE A LARGE DATASET

BUT YOU CAN USE AN EMBEDDING MATRIX E THAT IS ALREADY PRE-TRAINED

IDEA: SIMPLE CLASSIFICATION

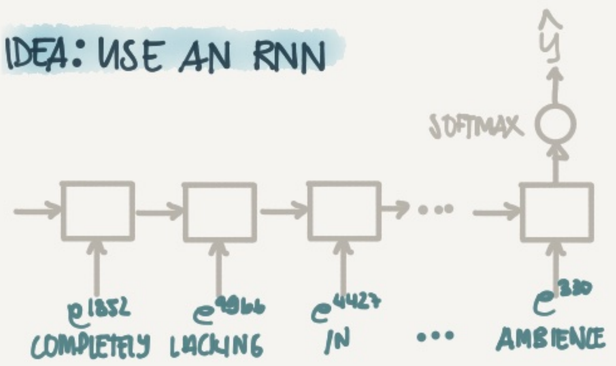


WORKS WELL FOR SHORT SENTENCES BUT DOESN'T TAKE ORDER INTO ACCOUNT

"COMPLETELY LACKING IN GOOD TASTE, GOOD SERVICE AND GOOD AMBIENCE"

THIS MAY BE SEEN AS A +++ REVIEW

IDEA: USE AN RNN



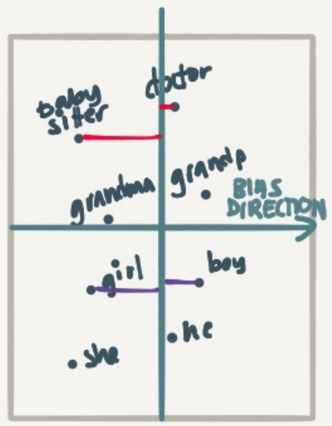
THIS CAN NOW TAKE INTO ACCOUNT THAT COMPLETELY LACKING NEGATES THE WORD GOOD

ELIMINATING BIAS IN WORD EMBEDDINGS

MAN IS TO COMPUTER PROGRAMMER AS WOMAN IS TO HOME MAKER

SOMETIMES THE TEXT CONTAINS & ALGOS LEARN A GENDER, RACE, AGE... BIAS WE DON'T WANT OUR MODELS TO HAVE · EX. HIRING BASED ON GENDER, SENTENCING BASED ON RACE ETC.

ADDRESSING BIAS



1. IDENTIFY BIAS DIRECTION
   $\{e_{he} \rightarrow e_{she}$
   $e_{male} \rightarrow e_{female}$
2. NEUTRALIZE
   FOR EVERY WORD THAT IS NOT DEFINITIONAL (GIRL, BOY, HE, SHE...) PROJECT TO GET RID OF BIAS
3. EQUALIZE PAIRS
   THE ONLY DIFF BETWEEN EX GIRL/BOY SHOULD BE GENDER

HOW DO YOU KNOW WHICH WORDS TO NEUTRALIZE?

DOCTOR, BEARD, SEWING MACHINE?

A: BY TRAINING A CLASSIFIER TO FIND OUT IF A WORD IS DEFINITIONAL

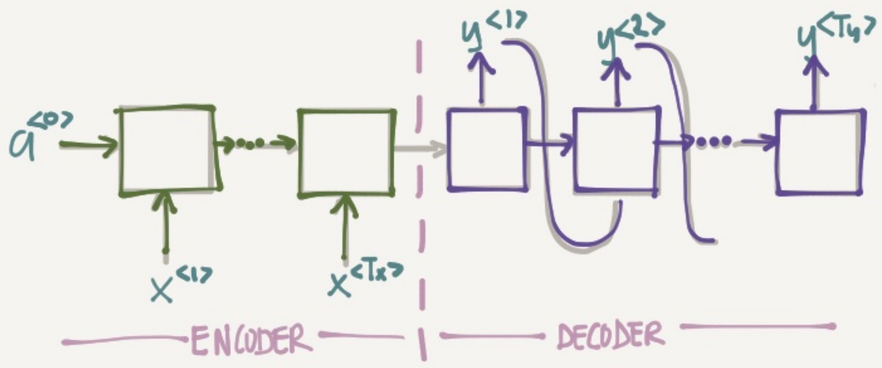
TURNS OUT THE # OF PAIRS IS FAIRLY SMALL SO YOU CAN EVE HAND PICK THEM

SEQUENCE TO SEQUENCE

SEQUENCE MODELS · COURSERA

BASIC MODELS

JANE VISITE L'AFRICE EN SEPTEMBRE → JANE IS VISITING AFRICA IN SEPTEMBER



 → THIS IS A CAT ON A CHAIR

CNN → RNN

HOW DO YOU PICK THE MOST LIKELY SENTENCE?

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

(ENGLISH, FRENCH)

WE DON'T WANT A RANDOMLY GENERATED SENTENCE
(WE WOULD SOMETIMES GET A GOOD, SOMETIMES BAD)
INSTEAD WE WANT TO MAXIMIZE

$$\text{ARG MAX } P(y^{<1>}, \dots, y^{<T_y>} | x)$$

IDEA: USE GREEDY SEARCH

1. PICK THE WORD WITH THE BEST PROBABILITY
2. REPEAT UNTIL <EOS>

WITH THIS WE COULD GET
- JANE IS GOING TO BE VISITING AFRICA THIS SEPTEMBER

INSTEAD OF
- JANE IS VISITING AFRICA THIS SEPTEMBER

SOLUTION: OPTIMISE THE PROB OF THE WHOLE SENTENCE INSTEAD

BEAM SEARCH

1. PICK THE FIRST WORD
   PICK THE B (EX 3) BEST ALTERNATIVES
   (IN, JANE, SEPTEMBER)
2. FOR EACH B WORDS PICK THE NEXT WORD AND EVALUATE THE PAIRS TO END UP W B PAIRS
   $P(y^{<1>}, y^{<2>} | x) = P(y^{<1>} | x)\, P(y^{<2>} | x, y^{<1>})$

(IN SEPTEMBER, JANE IS, JANE VISITS)

NOTE: KEEP TRACK OF THE P FOR THE SENTENCES OF EACH LENGTH - AFTER X ITER (X MAX WORDS) PICK THE BEST

3. REPEAT TIL DONE

$$\text{ARG MAX} \prod_{t=1}^{T_y} P(y^{<t>} | x, y^{<1>}, \dots y^{<t-1>})$$

PROBLEM: MULTIPLYING PROBABILITIES (0<P<1) RESULTS IN A VERY SMALL NUMBER → OVERFLOWS

PROBLEM II: IF WE OPTIMIZE FOR THE MULT WE WILL PREFER SHORT SENTENCES. SINCE EACH WORD WILL REDUCE PROB

INSTEAD WE CAN OPTIMIZE FOR THIS

$$\frac{1}{T_y^{\alpha}} \sum_{t=1}^{T_y} \log(P(y^{<t>} | x, y^{<1>}, \dots, y^{<t-1>}))$$

HOW DO WE PICK B?

LARGE B: BETTER RESULT, SLOWER
SMALL B: WORSE RESULT, BETTER

IN PROD YOU MIGHT SE B=10.
100 IS PROBABLY A BIT TOO HIGH - BUT ITS DOMAIN DEPENDENT

ERROR ANALYSIS IN BEAM S.

HUMAN: JANE VISITS AFRICA IN SEPT... y^*
ALGO: JANE VISITED AFRICA LAST SEPTEMBER $\hat{y}$

HOW DO WE KNOW IF ITS OUR RNN OR OUR BEAM SEARCH WE SHOULD WORK ON?

LET THE RNN GIVE $Py^* = P(y^*|x)$ & $P\hat{y} = P(\hat{y}|x)$

IF $Py^* > P\hat{y}$:
BEAM PICKED THE WRONG ONE
TRY A HIGHER B

ELSE:
THE RNN PICKED THE WRONG PROBS · SO FOCUS ON THE RNN

SEQUENCE TO SEQUENCE

FRENCH: LE CHAT EST SUR LE TAPIS
HUMAN1: THE CAT IS ON THE MAT
HUMAN2: THERE IS A CAT ON THE MAT

HOW DO YOU EVALUATE THE MACHINE TRANSLATION WHEN MULTIPLE ARE RIGHT?

BLEU SCORE

IDEA: CHECK IF THE WORDS (MT) APPEAR IN THE REAL TRANSLATION

THE THE THE THE THE THE THE
SCORE: 7/7

IDEA: ONLY GIVE CREDIT FOR A WORD THE MAX # TIMES IT APPEARS IN A TARGET SENTENCE
SCORE: 2/7 ← COUNT CLIP

THE CAT THE CAT ON THE MAT

BIGRAMS	COUNT	COUNT CLIP
THE CAT	2	1
CAT THE	1	0
CAT ON	1	1
ON THE	1	1
THE MAT	1	1
	6	4/6

BI-GRAM SCORE: 4/6

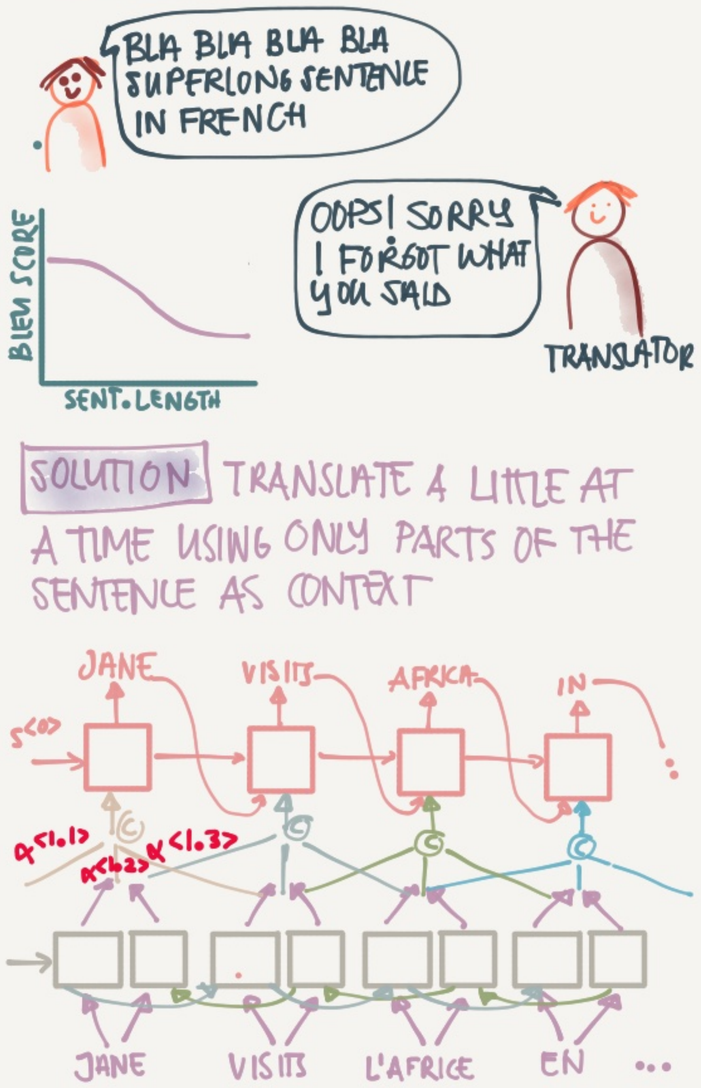
COMBINED BLEU SCORE

$BP \cdot \exp\left(\frac{1}{4}\sum_{n=1}^{4} P_n\right)$

P_1 = SCORE SINGLE WORD
P_2 = SORE BIGRAMS
...
BP = BREVITY PENALTY ← PENALIZES SENTENCES SHORTER THAN THE TARGET

↑ A USEFUL SINGLE NUMBER EVAL METRIC

ATTENTION MODEL



SOLUTION TRANSLATE A LITTLE AT A TIME USING ONLY PARTS OF THE SENTENCE AS CONTEXT

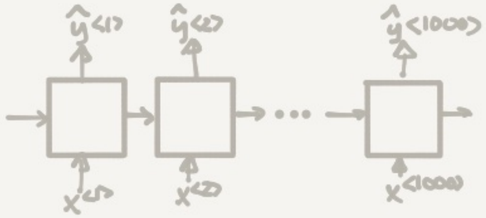
$\alpha^{<t,t'>}$ = HOW MUCH ATTENTION $y^{<t>}$ SHOULD PAY TO $a^{<t'>}$

$$c^{<2>} = \sum_{t'} \alpha^{<2,t'>} \cdot a^{<t'>} \quad \Big| \quad \sum_{t'} \alpha^{<1,t'>} = 1$$

α IS CALCULATED USING A SMALL NEURAL NETWORK

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

SPEECH RECOGNITION



PROBLEM: 10s CLIP AT 100Hz = 1000 INPUTS BUT ONLY ≈ 20 OUTPUTS

SOLUTION USE CTC COST (CONNECTION TEMPORAL CLASSIFICATION)

ttt_h_eee___u___ggg__ → COLLAPSE REPEATED CHARS NOT SEP BY BLANK

TRIGGER WORD DETECTION

SEQUENCE MODELS · COURSERA

Super VIP Cheatsheet: Artificial Intelligence

Afshine Amidi and Shervine Amidi

May 26, 2019

Contents

1 Reflex-based models

1.1 Linear predictors

In this section, we will go through reflex-based models that can improve with experience, by going through samples that have input-output pairs.

❒ **Feature vector** – The feature vector of an input x is noted $\phi(x)$ and is such that:

$$\boxed{\phi(x) = \begin{bmatrix} \phi_1(x) \\ \vdots \\ \phi_d(x) \end{bmatrix} \in \mathbb{R}^d}$$

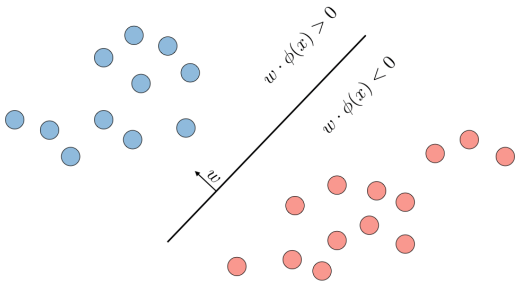
❒ **Score** – The score $s(x,w)$ of an example $(\phi(x),y) \in \mathbb{R}^d \times \mathbb{R}$ associated to a linear model of weights $w \in \mathbb{R}^d$ is given by the inner product:

$$\boxed{s(x,w) = w \cdot \phi(x)}$$

1.1.1 Classification

❒ **Linear classifier** – Given a weight vector $w \in \mathbb{R}^d$ and a feature vector $\phi(x) \in \mathbb{R}^d$, the binary linear classifier f_w is given by:

$$\boxed{f_w(x) = \text{sign}(s(x,w)) = \begin{cases} +1 & \text{if } w \cdot \phi(x) > 0 \\ -1 & \text{if } w \cdot \phi(x) < 0 \\ ? & \text{if } w \cdot \phi(x) = 0 \end{cases}}$$



❒ **Margin** – The margin $m(x,y,w) \in \mathbb{R}$ of an example $(\phi(x),y) \in \mathbb{R}^d \times \{-1,+1\}$ associated to a linear model of weights $w \in \mathbb{R}^d$ quantifies the confidence of the prediction: larger values are better. It is given by:

$$\boxed{m(x,y,w) = s(x,w) \times y}$$

1.1.2 Regression

❒ **Linear regression** – Given a weight vector $w \in \mathbb{R}^d$ and a feature vector $\phi(x) \in \mathbb{R}^d$, the output of a linear regression of weights w denoted as f_w is given by:

$$\boxed{f_w(x) = s(x,w)}$$

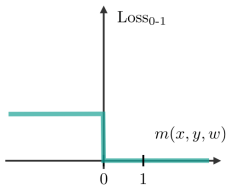
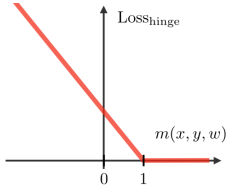
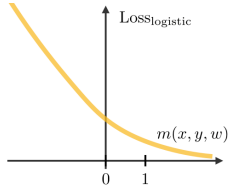
❒ **Residual** – The residual $\text{res}(x,y,w) \in \mathbb{R}$ is defined as being the amount by which the prediction $f_w(x)$ overshoots the target y:

$$\boxed{\text{res}(x,y,w) = f_w(x) - y}$$

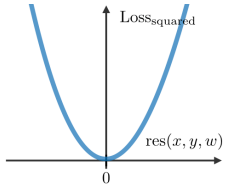
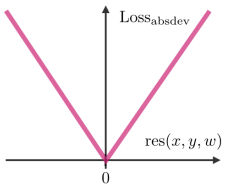
1.2 Loss minimization

❒ **Loss function** – A loss function $\text{Loss}(x,y,w)$ quantifies how unhappy we are with the weights w of the model in the prediction task of output y from input x. It is a quantity we want to minimize during the training process.

❒ **Classification case** – The classification of a sample x of true label $y \in \{-1,+1\}$ with a linear model of weights w can be done with the predictor $f_w(x) \triangleq \text{sign}(s(x,w))$. In this situation, a metric of interest quantifying the quality of the classification is given by the margin $m(x,y,w)$, and can be used with the following loss functions:

Name	Zero-one loss	Hinge loss	Logistic loss
$\text{Loss}(x,y,w)$	$1_{\{m(x,y,w) \leqslant 0\}}$	$\max(1 - m(x,y,w), 0)$	$\log(1 + e^{-m(x,y,w)})$
Illustration			

❒ **Regression case** – The prediction of a sample x of true label $y \in \mathbb{R}$ with a linear model of weights w can be done with the predictor $f_w(x) \triangleq s(x,w)$. In this situation, a metric of interest quantifying the quality of the regression is given by the margin $\text{res}(x,y,w)$ and can be used with the following loss functions:

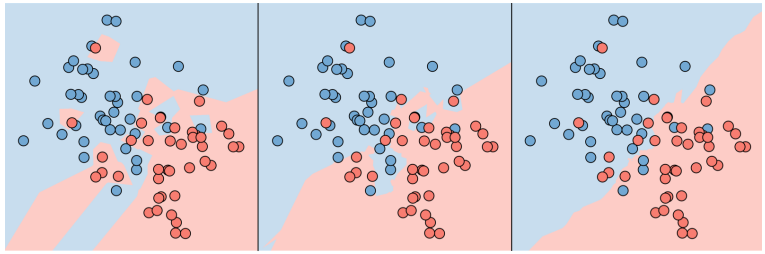
Name	Squared loss	Absolute deviation loss
$\text{Loss}(x,y,w)$	$(\text{res}(x,y,w))^2$	$\lvert\text{res}(x,y,w)\rvert$
Illustration		

❒ **Loss minimization framework** – In order to train a model, we want to minimize the training loss is defined as follows:

$$\boxed{\text{TrainLoss}(w) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y)\in\mathcal{D}_{\text{train}}} \text{Loss}(x,y,w)}$$

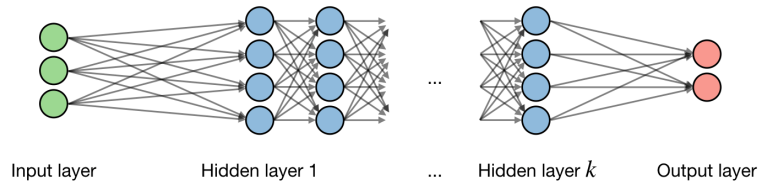
1.3 Non-linear predictors

❒ **k-nearest neighbors** – The k-nearest neighbors algorithm, commonly known as k-NN, is a non-parametric approach where the response of a data point is determined by the nature of its k neighbors from the training set. It can be used in both classification and regression settings.



Remark: the higher the parameter k, the higher the bias, and the lower the parameter k, the higher the variance.

❒ **Neural networks** – Neural networks are a class of models that are built with layers. Commonly used types of neural networks include convolutional and recurrent neural networks. The vocabulary around neural networks architectures is described in the figure below:



By noting i the i^{th} layer of the network and j the j^{th} hidden unit of the layer, we have:

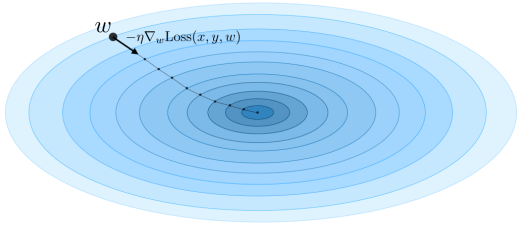
$$\boxed{z_j^{[i]} = {w_j^{[i]}}^T x + b_j^{[i]}}$$

where we note w, b, x, z the weight, bias, input and non-activated output of the neuron respectively.

1.4 Stochastic gradient descent

❒ **Gradient descent** – By noting $\eta \in \mathbb{R}$ the learning rate (also called step size), the update rule for gradient descent is expressed with the learning rate and the loss function $\text{Loss}(x,y,w)$ as follows:

$$\boxed{w \longleftarrow w - \eta\nabla_w \text{Loss}(x,y,w)}$$



❒ **Stochastic updates** – Stochastic gradient descent (SGD) updates the parameters of the model one training example $(\phi(x),y) \in \mathcal{D}_{\text{train}}$ at a time. This method leads to sometimes noisy, but fast updates.

❒ **Batch updates** – Batch gradient descent (BGD) updates the parameters of the model one batch of examples (e.g. the entire training set) at a time. This method computes stable update directions, at a greater computational cost.

1.5 Fine-tuning models

❒ **Hypothesis class** – A hypothesis class $\mathcal{F}$ is the set of possible predictors with a fixed $\phi(x)$ and varying w:

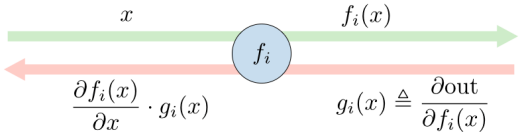
$$\boxed{\mathcal{F} = \left\{f_w : w \in \mathbb{R}^d\right\}}$$

❒ **Logistic function** – The logistic function σ, also called the sigmoid function, is defined as:

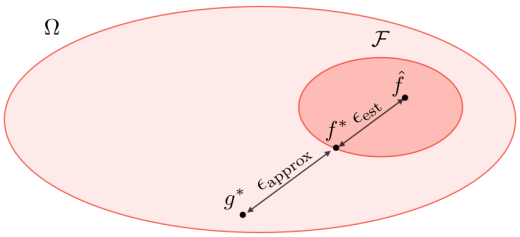
$$\boxed{\forall z \in]-\infty,+\infty[, \quad \sigma(z) = \frac{1}{1+e^{-z}}}$$

Remark: we have $\sigma'(z) = \sigma(z)(1-\sigma(z))$.

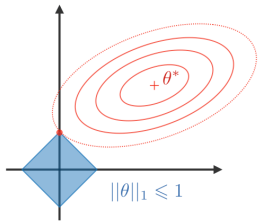
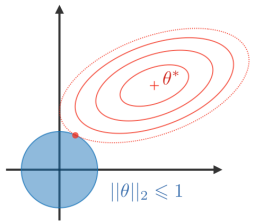
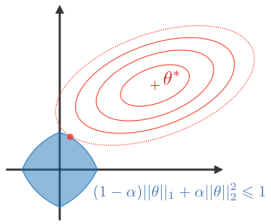
❒ **Backpropagation** – The forward pass is done through f_i, which is the value for the subexpression rooted at i, while the backward pass is done through $g_i = \frac{\partial \text{out}}{\partial f_i}$ and represents how f_i influences the output.



❒ **Approximation and estimation error** – The approximation error ϵ_{approx} represents how far the entire hypothesis class $\mathcal{F}$ is from the target predictor g^*, while the estimation error ϵ_{est} quantifies how good the predictor $\hat{f}$ is with respect to the best predictor f^* of the hypothesis class $\mathcal{F}$.



❒ **Regularization** – The regularization procedure aims at avoiding the model to overfit the data and thus deals with high variance issues. The following table sums up the different types of commonly used regularization techniques:

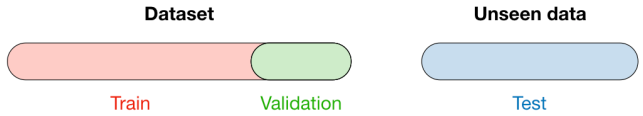
LASSO	Ridge	Elastic Net
- Shrinks coefficients to 0 - Good for variable selection	Makes coefficients smaller	Tradeoff between variable selection and small coefficients
		
$\ldots + \lambda\|\|\theta\|\|_1$ $\lambda \in \mathbb{R}$	$\ldots + \lambda\|\|\theta\|\|_2^2$ $\lambda \in \mathbb{R}$	$\ldots + \lambda\left[(1-\alpha)\|\|\theta\|\|_1 + \alpha\|\|\theta\|\|_2^2\right]$ $\lambda \in \mathbb{R}$, $\alpha \in [0,1]$

❒ **Hyperparameters** – Hyperparameters are the properties of the learning algorithm, and include features, regularization parameter λ, number of iterations T, step size η, etc.

❒ **Sets vocabulary** – When selecting a model, we distinguish 3 different parts of the data that we have as follows:

Training set	Validation set	Testing set
- Model is trained - Usually 80 of the dataset	- Model is assessed - Usually 20 of the dataset - Also called hold-out	- Model gives predictions - Unseen data or development set

Once the model has been chosen, it is trained on the entire dataset and tested on the unseen test set. These are represented in the figure below:



1.6 Unsupervised Learning

The class of unsupervised learning methods aims at discovering the structure of the data, which may have of rich latent structures.

1.6.1 k-means

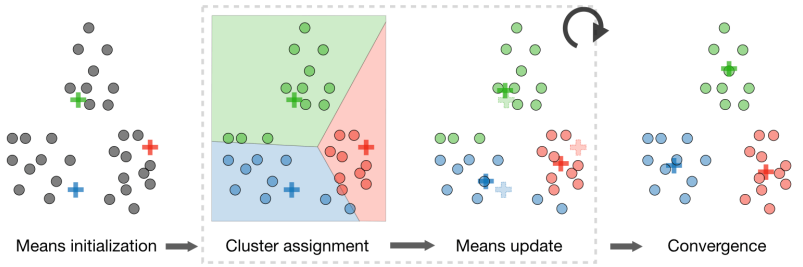
❒ **Clustering** – Given a training set of input points $\mathcal{D}_{\text{train}}$, the goal of a clustering algorithm is to assign each point $\phi(x_i)$ to a cluster $z_i \in \{1,\ldots,k\}$.

❒ **Objective function** – The loss function for one of the main clustering algorithms, k-means, is given by:

$$\boxed{\text{Loss}_{\text{k-means}}(x,\mu) = \sum_{i=1}^{n} ||\phi(x_i) - \mu_{z_i}||^2}$$

❒ **Algorithm** – After randomly initializing the cluster centroids $\mu_1,\mu_2,\ldots,\mu_k \in \mathbb{R}^n$, the k-means algorithm repeats the following step until convergence:

$$\boxed{z_i = \arg\min_j ||\phi(x_i) - \mu_j||^2} \quad \text{and} \quad \boxed{\mu_j = \frac{\sum_{i=1}^{m} 1_{\{z_i=j\}}\phi(x_i)}{\sum_{i=1}^{m} 1_{\{z_i=j\}}}}$$



1.6.2 Principal Component Analysis

❒ **Eigenvalue, eigenvector** – Given a matrix $A \in \mathbb{R}^{n\times n}$, λ is said to be an eigenvalue of A if there exists a vector $z \in \mathbb{R}^n \backslash \{0\}$, called eigenvector, such that we have:

$$\boxed{Az = \lambda z}$$

❒ **Spectral theorem** – Let $A \in \mathbb{R}^{n\times n}$. If A is symmetric, then A is diagonalizable by a real orthogonal matrix $U \in \mathbb{R}^{n\times n}$. By noting $\Lambda = \text{diag}(\lambda_1,...,\lambda_n)$, we have:

$$\boxed{\exists \Lambda \text{ diagonal}, \quad A = U\Lambda U^T}$$

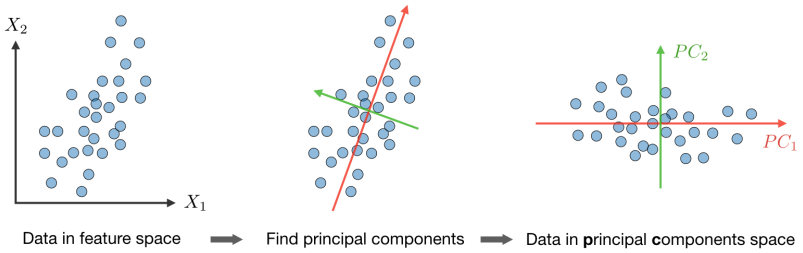
Remark: the eigenvector associated with the largest eigenvalue is called principal eigenvector of matrix A.

❒ **Algorithm** – The Principal Component Analysis (PCA) procedure is a dimension reduction technique that projects the data on k dimensions by maximizing the variance of the data as follows:

- Step 1: Normalize the data to have a mean of 0 and standard deviation of 1.

$$\boxed{x_j^{(i)} \leftarrow \frac{x_j^{(i)} - \mu_j}{\sigma_j}} \quad \text{where} \quad \boxed{\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}} \quad \text{and} \quad \boxed{\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_j^{(i)} - \mu_j)^2}$$

- Step 2: Compute $\Sigma = \dfrac{1}{m}\displaystyle\sum_{i=1}^{m} x^{(i)}x^{(i)T} \in \mathbb{R}^{n\times n}$, which is symmetric with real eigenvalues.
- Step 3: Compute $u_1,...,u_k \in \mathbb{R}^n$ the k orthogonal principal eigenvectors of Σ, i.e. the orthogonal eigenvectors of the k largest eigenvalues.
- Step 4: Project the data on $\text{span}_{\mathbb{R}}(u_1,...,u_k)$. This procedure maximizes the variance among all k-dimensional spaces.



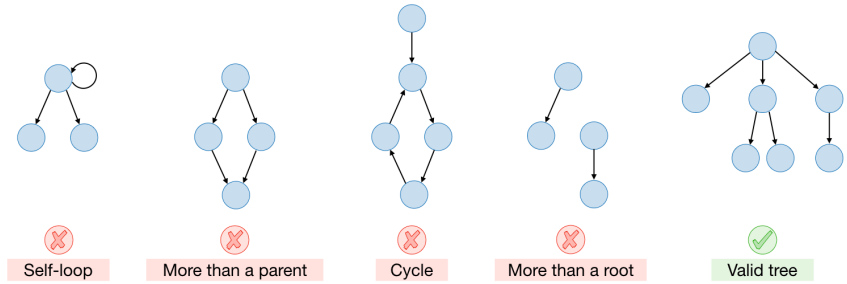
2 States-based models

2.1 Search optimization

In this section, we assume that by accomplishing action a from state s, we deterministically arrive in state Succ(s,a). The goal here is to determine a sequence of actions (a_1,a_2,a_3,a_4,...) that starts from an initial state and leads to an end state. In order to solve this kind of problem, our objective will be to find the minimum cost path by using states-based models.

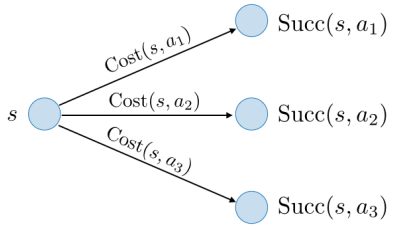
2.1.1 Tree search

This category of states-based algorithms explores all possible states and actions. It is quite memory efficient, and is suitable for huge state spaces but the runtime can become exponential in the worst cases.



❒ **Search problem** – A search problem is defined with:

- a starting state s_{start}
- possible actions Actions(s) from state s
- action cost Cost(s,a) from state s with action a
- successor Succ(s,a) of state s after action a
- whether an end state was reached IsEnd(s)

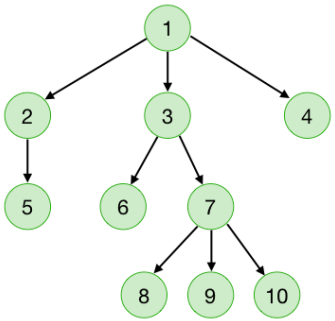


The objective is to find a path that minimizes the cost.

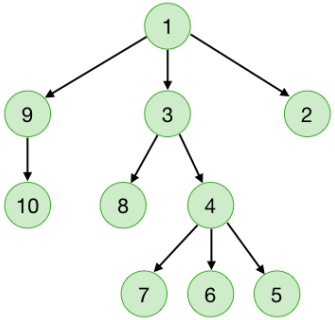
❒ **Backtracking search** – Backtracking search is a naive recursive algorithm that tries all possibilities to find the minimum cost path. Here, action costs can be either positive or negative.

❒ **Breadth-first search (BFS)** – Breadth-first search is a graph search algorithm that does a level-by-level traversal. We can implement it iteratively with the help of a queue that stores at

each step future nodes to be visited. For this algorithm, we can assume action costs to be equal to a constant $c \geqslant 0$.



❒ **Depth-first search (DFS)** – Depth-first search is a search algorithm that traverses a graph by following each path as deep as it can. We can implement it recursively, or iteratively with the help of a stack that stores at each step future nodes to be visited. For this algorithm, action costs are assumed to be equal to 0.



❒ **Iterative deepening** – The iterative deepening trick is a modification of the depth-first search algorithm so that it stops after reaching a certain depth, which guarantees optimality when all action costs are equal. Here, we assume that action costs are equal to a constant $c \geqslant 0$.

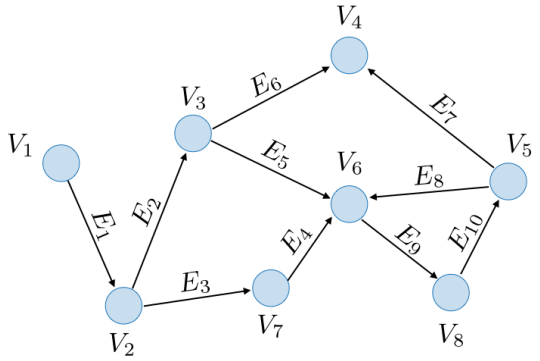
❒ **Tree search algorithms summary** – By noting b the number of actions per state, d the solution depth, and D the maximum depth, we have:

Algorithm	Action costs	Space	Time
Backtracking search	any	$\mathcal{O}(D)$	$\mathcal{O}(b^D)$
Breadth-first search	$c \geqslant 0$	$\mathcal{O}(b^d)$	$\mathcal{O}(b^d)$
Depth-first search	0	$\mathcal{O}(D)$	$\mathcal{O}(b^D)$
DFS-Iterative deepening	$c \geqslant 0$	$\mathcal{O}(d)$	$\mathcal{O}(b^d)$

2.1.2 Graph search

This category of states-based algorithms aims at constructing optimal paths, enabling exponential savings. In this section, we will focus on dynamic programming and uniform cost search.

❒ **Graph** – A graph is comprised of a set of vertices V (also called nodes) as well as a set of edges E (also called links).

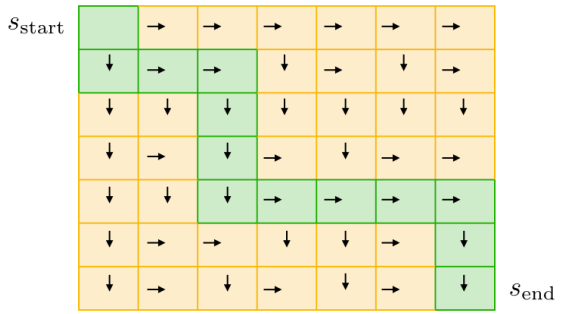


Remark: a graph is said to be acylic when there is no cycle.

❒ **State** – A state is a summary of all past actions sufficient to choose future actions optimally.

❒ **Dynamic programming** – Dynamic programming (DP) is a backtracking search algorithm with memoization (i.e. partial results are saved) whose goal is to find a minimum cost path from state s to an end state s_{end}. It can potentially have exponential savings compared to traditional graph search algorithms, and has the property to only work for acyclic graphs. For any given state s, the future cost is computed as follows:

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ \min\limits_{a \in \text{Actions}(s)} \big[\text{Cost}(s,a) + \text{FutureCost}(\text{Succ}(s,a))\big] & \text{otherwise} \end{cases}$$

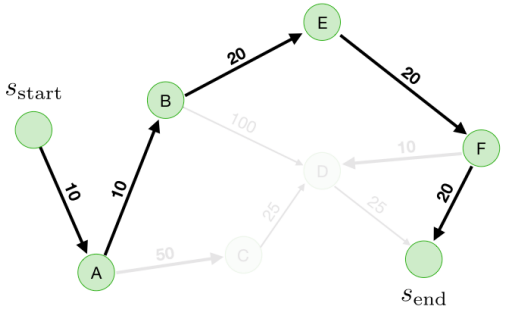


Remark: the figure above illustrates a bottom-to-top approach whereas the formula provides the intuition of a top-to-bottom problem resolution.

❒ **Types of states** – The table below presents the terminology when it comes to states in the context of uniform cost search:

State	Explanation
Explored $\mathcal{E}$	States for which the optimal path has already been found
Frontier $\mathcal{F}$	States seen for which we are still figuring out how to get there with the cheapest cost
Unexplored $\mathcal{U}$	States not seen yet

❒ **Uniform cost search** – Uniform cost search (UCS) is a search algorithm that aims at finding the shortest path from a state s_{start} to an end state s_{end}. It explores states s in increasing order of PastCost(s) and relies on the fact that all action costs are non-negative.



Remark 1: the UCS algorithm is logically equivalent to Djikstra's algorithm.
Remark 2: the algorithm would not work for a problem with negative action costs, and adding a positive constant to make them non-negative would not solve the problem since this would end up being a different problem.

❒ **Correctness theorem** – When a state s is popped from the frontier $\mathcal{F}$ and moved to explored set $\mathcal{E}$, its priority is equal to PastCost(s) which is the minimum cost path from s_{start} to s.

❒ **Graph search algorithms summary** – By noting N the number of total states, n of which are explored before the end state s_{end}, we have:

Algorithm	Acyclicity	Costs	Time/space
Dynamic programming	yes	any	$\mathcal{O}(N)$
Uniform cost search	no	$c \geqslant 0$	$\mathcal{O}(n\log(n))$

Remark: the complexity countdown supposes the number of possible actions per state to be constant.

2.1.3 Learning costs

Suppose we are not given the values of Cost(s,a), we want to estimate these quantities from a training set of minimizing-cost-path sequence of actions $(a_1, a_2, ..., a_k)$.

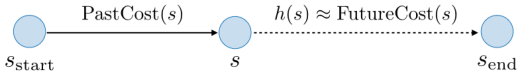
❒ **Structured perceptron** – The structured perceptron is an algorithm aiming at iteratively learning the cost of each state-action pair. At each step, it:

- decreases the estimated cost of each state-action of the true minimizing path y given by the training data,
- increases the estimated cost of each state-action of the current predicted path y' inferred from the learned weights.

Remark: there are several versions of the algorithm, one of which simplifies the problem to only learning the cost of each action a, and the other parametrizes Cost(s,a) to a feature vector of learnable weights.

2.1.4 A* search

❒ **Heuristic function** – A heuristic is a function h over states s, where each $h(s)$ aims at estimating FutureCost(s), the cost of the path from s to s_{end}.



❒ **Algorithm** – A^* is a search algorithm that aims at finding the shortest path from a state s to an end state s_{end}. It explores states s in increasing order of PastCost$(s) + h(s)$. It is equivalent to a uniform cost search with edge costs Cost$'(s,a)$ given by:

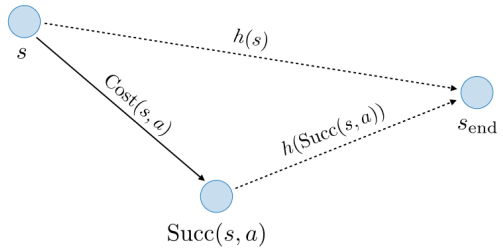
$$\boxed{\text{Cost}'(s,a) = \text{Cost}(s,a) + h(\text{Succ}(s,a)) - h(s)}$$

Remark: this algorithm can be seen as a biased version of UCS exploring states estimated to be closer to the end state.

❒ **Consistency** – A heuristic h is said to be consistent if it satisfies the two following properties:

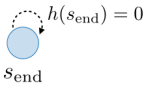
- For all states s and actions a,

$$\boxed{h(s) \leqslant \text{Cost}(s,a) + h(\text{Succ}(s,a))}$$



- The end state verifies the following:

$$\boxed{h(s_{\text{end}}) = 0}$$



❒ **Correctness** – If h is consistent, then A^* returns the minimum cost path.

❒ **Admissibility** – A heuristic h is said to be admissible if we have:

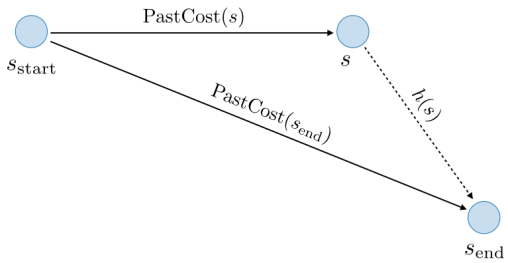
$$\boxed{h(s) \leqslant \text{FutureCost}(s)}$$

❒ **Theorem** – Let $h(s)$ be a given heuristic. We have:

$$\boxed{h(s) \text{ consistent} \implies h(s) \text{ admissible}}$$

❒ **Efficiency** – A^* explores all states s satisfying the following equation:

$$\boxed{\text{PastCost}(s) \leqslant \text{PastCost}(s_{\text{end}}) - h(s)}$$



Remark: larger values of $h(s)$ is better as this equation shows it will restrict the set of states s going to be explored.

2.1.5 Relaxation

It is a framework for producing consistent heuristics. The idea is to find closed-form reduced costs by removing constraints and use them as heuristics.

❒ **Relaxed search problem** – The relaxation of search problem P with costs Cost is noted P_{rel} with costs Cost_{rel}, and satisfies the identity:

$$\boxed{\text{Cost}_{\text{rel}}(s,a) \leqslant \text{Cost}(s,a)}$$

❒ **Relaxed heuristic** – Given a relaxed search problem P_{rel}, we define the relaxed heuristic $h(s) = \text{FutureCost}_{\text{rel}}(s)$ as the minimum cost path from s to an end state in the graph of costs $\text{Cost}_{\text{rel}}(s,a)$.

❒ **Consistency of relaxed heuristics** – Let P_{rel} be a given relaxed problem. By theorem, we have:

$$\boxed{h(s) = \text{FutureCost}_{\text{rel}}(s) \implies h(s) \text{ consistent}}$$

❒ **Tradeoff when choosing heuristic** – We have to balance two aspects in choosing a heuristic:

- Computational efficiency: $h(s) = \text{FutureCost}_{\text{rel}}(s)$ must be easy to compute. It has to produce a closed form, easier search and independent subproblems.
- Good enough approximation: the heuristic $h(s)$ should be close to $\text{FutureCost}(s)$ and we have thus to not remove too many constraints.

❒ **Max heuristic** – Let $h_1(s)$, $h_2(s)$ be two heuristics. We have the following property:

$$\boxed{h_1(s),\, h_2(s) \text{ consistent} \implies h(s) = \max\{h_1(s),\, h_2(s)\} \text{ consistent}}$$

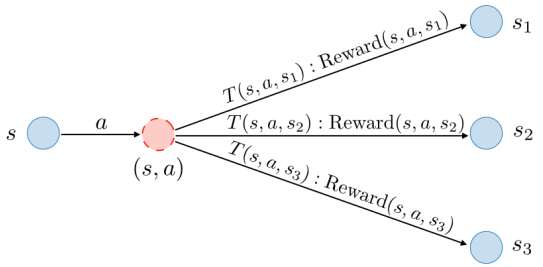
2.2 Markov decision processes

In this section, we assume that performing action a from state s can lead to several states s'_1,s'_2,... in a probabilistic manner. In order to find our way between an initial state and an end state, our objective will be to find the maximum value policy by using Markov decision processes that help us cope with randomness and uncertainty.

2.2.1 Notations

❒ **Definition** – The objective of a Markov decision process is to maximize rewards. It is defined with:

- a starting state s_{start}
- possible actions $\text{Actions}(s)$ from state s
- transition probabilities $T(s,a,s')$ from s to s' with action a
- rewards $\text{Reward}(s,a,s')$ from s to s' with action a
- whether an end state was reached $\text{IsEnd}(s)$
- a discount factor $0 \leqslant \gamma \leqslant 1$



❒ **Transition probabilities** – The transition probability $T(s,a,s')$ specifies the probability of going to state s' after action a is taken in state s. Each $s' \mapsto T(s,a,s')$ is a probability distribution, which means that:

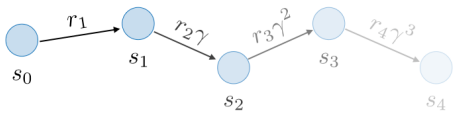
$$\forall s,a, \quad \boxed{\sum_{s' \in \text{ States}} T(s,a,s') = 1}$$

❒ **Policy** – A policy π is a function that maps each state s to an action a, i.e.

$$\boxed{\pi : s \mapsto a}$$

❒ **Utility** – The utility of a path $(s_0, ..., s_k)$ is the discounted sum of the rewards on that path. In other words,

$$\boxed{u(s_0,...,s_k) = \sum_{i=1}^{k} r_i \gamma^{i-1}}$$



Remark: the figure above is an illustration of the case $k = 4$.

❒ **Q-value** – The Q-value of a policy π by taking action a from state s, also noted $Q_\pi(s,a)$, is the expected utility of taking action a from state s and then following policy π. It is defined as follows:

$$\boxed{Q_\pi(s,a) = \sum_{s' \in \text{ States}} T(s,a,s') \left[\text{Reward}(s,a,s') + \gamma V_\pi(s')\right]}$$

❒ **Value of a policy** – The value of a policy π from state s, also noted $V_\pi(s)$, is the expected utility by following policy π from state s over random paths. It is defined as follows:

$$\boxed{V_\pi(s) = Q_\pi(s,\pi(s))}$$

Remark: $V_\pi(s)$ is equal to 0 if s is an end state.

2.2.2 Applications

❒ **Policy evaluation** – Given a policy π, policy evaluation is an iterative algorithm that computes V_π. It is done as follows:

- Initialization: for all states s, we have

$$\boxed{V_\pi^{(0)}(s) \longleftarrow 0}$$

- Iteration: for t from 1 to T_{PE}, we have

$$\forall s, \quad \boxed{V_\pi^{(t)}(s) \longleftarrow Q_\pi^{(t-1)}(s,\pi(s))}$$

with

$$\boxed{Q_\pi^{(t-1)}(s,\pi(s)) = \sum_{s' \in \text{ States}} T(s,\pi(s),s') \left[\text{Reward}(s,\pi(s),s') + \gamma V_\pi^{(t-1)}(s')\right]}$$

Remark: by noting S the number of states, A the number of actions per state, S' the number of successors and T the number of iterations, then the time complexity is of $\mathcal{O}(T_{PE}SS')$.

❒ **Optimal Q-value** – The optimal Q-value $Q_{\text{opt}}(s,a)$ of state s with action a is defined to be the maximum Q-value attained by any policy starting. It is computed as follows:

$$\boxed{Q_{\text{opt}}(s,a) = \sum_{s' \in \text{ States}} T(s,a,s') \left[\text{Reward}(s,a,s') + \gamma V_{\text{opt}}(s')\right]}$$

❒ **Optimal value** – The optimal value $V_{\text{opt}}(s)$ of state s is defined as being the maximum value attained by any policy. It is computed as follows:

$$\boxed{V_{\text{opt}}(s) = \max_{a \in \text{ Actions}(s)} Q_{\text{opt}}(s,a)}$$

❒ **Optimal policy** – The optimal policy π_{opt} is defined as being the policy that leads to the optimal values. It is defined by:

$$\forall s, \quad \boxed{\pi_{\text{opt}}(s) = \operatorname*{argmax}_{a \in \text{ Actions}(s)} Q_{\text{opt}}(s,a)}$$

❒ **Value iteration** – Value iteration is an algorithm that finds the optimal value V_{opt} as well as the optimal policy π_{opt}. It is done as follows:

- Initialization: for all states s, we have

$$\boxed{V_{\text{opt}}^{(0)}(s) \longleftarrow 0}$$

- Iteration: for t from 1 to T_{VI}, we have

$$\forall s, \quad \boxed{V_{\text{opt}}^{(t)}(s) \longleftarrow \max_{a \in \text{ Actions}(s)} Q_{\text{opt}}^{(t-1)}(s,a)}$$

with

$$\boxed{Q_{\text{opt}}^{(t-1)}(s,a) = \sum_{s' \in \text{ States}} T(s,a,s') \left[\text{Reward}(s,a,s') + \gamma V_{\text{opt}}^{(t-1)}(s')\right]}$$

Remark: if we have either $\gamma < 1$ or the MDP graph being acyclic, then the value iteration algorithm is guaranteed to converge to the correct answer.

2.2.3 When unknown transitions and rewards

Now, let's assume that the transition probabilities and the rewards are unknown.

❒ **Model-based Monte Carlo** – The model-based Monte Carlo method aims at estimating $T(s,a,s')$ and $\text{Reward}(s,a,s')$ using Monte Carlo simulation with:

$$\boxed{\widehat{T}(s,a,s') = \frac{\#\text{ times } (s,a,s') \text{ occurs}}{\#\text{ times } (s,a) \text{ occurs}}}$$

and

$$\boxed{\widehat{\text{Reward}}(s,a,s') = r \text{ in } (s,a,r,s')}$$

These estimations will be then used to deduce Q-values, including Q_π and Q_{opt}.

Remark: model-based Monte Carlo is said to be off-policy, because the estimation does not depend on the exact policy.

❒ **Model-free Monte Carlo** – The model-free Monte Carlo method aims at directly estimating Q_π, as follows:

$$\boxed{\widehat{Q}_\pi(s,a) = \text{average of } u_t \text{ where } s_{t-1} = s, a_t = a}$$

where u_t denotes the utility starting at step t of a given episode.

Remark: model-free Monte Carlo is said to be on-policy, because the estimated value is dependent on the policy π used to generate the data.

❒ **Equivalent formulation** – By introducing the constant $\eta = \frac{1}{1+(\#\text{updates to } (s,a))}$ and for each (s,a,u) of the training set, the update rule of model-free Monte Carlo has a convex combination formulation:

$$\boxed{\widehat{Q}_\pi(s,a) \leftarrow (1-\eta)\widehat{Q}_\pi(s,a) + \eta u}$$

as well as a stochastic gradient formulation:

$$\boxed{\widehat{Q}_\pi(s,a) \leftarrow \widehat{Q}_\pi(s,a) - \eta(\widehat{Q}_\pi(s,a) - u)}$$

❒ **SARSA** – State-action-reward-state-action (SARSA) is a boostrapping method estimating Q_π by using both raw data and estimates as part of the update rule. For each (s,a,r,s',a'), we have:

$$\boxed{\widehat{Q}_\pi(s,a) \longleftarrow (1-\eta)\widehat{Q}_\pi(s,a) + \eta\Big[r + \gamma\widehat{Q}_\pi(s',a')\Big]}$$

Remark: the SARSA estimate is updated on the fly as opposed to the model-free Monte Carlo one where the estimate can only be updated at the end of the episode.

❒ **Q-learning** – Q-learning is an off-policy algorithm that produces an estimate for Q_{opt}. On each (s,a,r,s',a'), we have:

$$\boxed{\widehat{Q}_{\text{opt}}(s,a) \leftarrow (1-\eta)\widehat{Q}_{\text{opt}}(s,a) + \eta\Big[r + \gamma \max_{a' \in \text{ Actions}(s')} \widehat{Q}_{\text{opt}}(s',a')\Big]}$$

❒ **Epsilon-greedy** – The epsilon-greedy policy is an algorithm that balances exploration with probability ϵ and exploitation with probability $1-\epsilon$. For a given state s, the policy π_{act} is computed as follows:

$$\boxed{\pi_{\text{act}}(s) = \begin{cases} \underset{a \in \text{ Actions}}{\text{argmax}}\ \widehat{Q}_{\text{opt}}(s,a) & \text{with proba } 1-\epsilon \\ \text{random from Actions}(s) & \text{with proba } \epsilon \end{cases}}$$

2.3 Game playing

In games (e.g. chess, backgammon, Go), other agents are present and need to be taken into account when constructing our policy.

❒ **Game tree** – A game tree is a tree that describes the possibilities of a game. In particular, each node is a decision point for a player and each root-to-leaf path is a possible outcome of the game.

❒ **Two-player zero-sum game** – It is a game where each state is fully observed and such that players take turns. It is defined with:

- a starting state s_{start}
- possible actions $\text{Actions}(s)$ from state s
- successors $\text{Succ}(s,a)$ from states s with actions a
- whether an end state was reached $\text{IsEnd}(s)$
- the agent's utility $\text{Utility}(s)$ at end state s
- the player $\text{Player}(s)$ who controls state s

Remark: we will assume that the utility of the agent has the opposite sign of the one of the opponent.

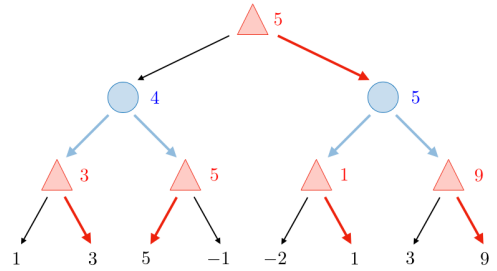
❒ **Types of policies** – There are two types of policies:

- Deterministic policies, noted $\pi_p(s)$, which are actions that player p takes in state s.
- Stochastic policies, noted $\pi_p(s,a) \in [0,1]$, which are probabilities that player p takes action a in state s.

❒ **Expectimax** – For a given state s, the expectimax value $V_{\text{exptmax}}(s)$ is the maximum expected utility of any agent policy when playing with respect to a fixed and known opponent policy π_{opp}. It is computed as follows:

$$\boxed{V_{\text{exptmax}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max\limits_{a \in \text{Actions}(s)} V_{\text{exptmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{agent} \\ \sum\limits_{a \in \text{Actions}(s)} \pi_{\text{opp}}(s,a) V_{\text{exptmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{opp} \end{cases}}$$

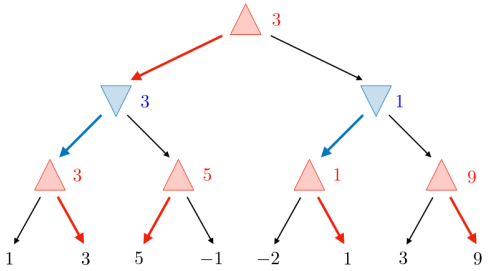
Remark: expectimax is the analog of value iteration for MDPs.



❒ **Minimax** – The goal of minimax policies is to find an optimal policy against an adversary by assuming the worst case, i.e. that the opponent is doing everything to minimize the agent's utility. It is done as follows:

$$\boxed{V_{\text{minimax}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max\limits_{a \in \text{Actions}(s)} V_{\text{minimax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{agent} \\ \min\limits_{a \in \text{Actions}(s)} V_{\text{minimax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{opp} \end{cases}}$$

Remark: we can extract π_{max} and π_{min} from the minimax value $V_{minimax}$.



❒ **Minimax properties** – By noting V the value function, there are 3 properties around minimax to have in mind:

- *Property 1*: if the agent were to change its policy to any π_{agent}, then the agent would be no better off.

$$\boxed{\forall \pi_{\text{agent}}, \quad V(\pi_{\max},\pi_{\min}) \geqslant V(\pi_{\text{agent}},\pi_{\min})}$$

- *Property 2*: if the opponent changes its policy from $\pi_{\min}$ to π_{opp}, then he will be no better off.

$$\boxed{\forall \pi_{\text{opp}}, \quad V(\pi_{\max},\pi_{\min}) \leqslant V(\pi_{\max},\pi_{\text{opp}})}$$

- *Property 3*: if the opponent is known to be not playing the adversarial policy, then the minimax policy might not be optimal for the agent.

$$\boxed{\forall \pi, \quad V(\pi_{\max},\pi) \leqslant V(\pi_{\text{exptmax}},\pi)}$$

In the end, we have the following relationship:

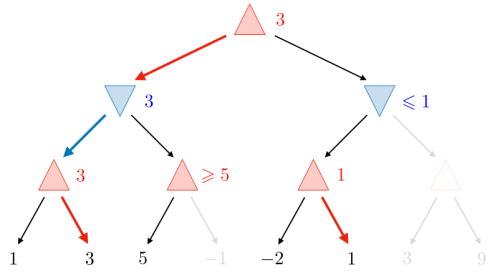
$$\boxed{V(\pi_{\text{exptmax}},\pi_{\min}) \leqslant V(\pi_{\max},\pi_{\min}) \leqslant V(\pi_{\max},\pi) \leqslant V(\pi_{\text{exptmax}},\pi)}$$

2.3.1 Speeding up minimax

❒ **Evaluation function** – An evaluation function is a domain-specific and approximate estimate of the value $V_{\text{minimax}}(s)$. It is noted $\text{Eval}(s)$.

Remark: $\text{FutureCost}(s)$ is an analogy for search problems.

❒ **Alpha-beta pruning** – Alpha-beta pruning is a domain-general exact method optimizing the minimax algorithm by avoiding the unnecessary exploration of parts of the game tree. To do so, each player keeps track of the best value they can hope for (stored in α for the maximizing player and in β for the minimizing player). At a given step, the condition $\beta < \alpha$ means that the optimal path is not going to be in the current branch as the earlier player had a better option at their disposal.



❒ **TD learning** – Temporal difference (TD) learning is used when we don't know the transitions/rewards. The value is based on exploration policy. To be able to use it, we need to know rules of the game $\text{Succ}(s,a)$. For each (s,a,r,s'), the update is done as follows:

$$\boxed{w \longleftarrow w - \eta\Big[V(s,w) - (r + \gamma V(s',w))\Big]\nabla_w V(s,w)}$$

2.3.2 Simultaneous games

This is the contrary of turn-based games, where there is no ordering on the player's moves.

❒ **Single-move simultaneous game** – Let there be two players A and B, with given possible actions. We note $V(a,b)$ to be A's utility if A chooses action a, B chooses action b. V is called the payoff matrix.

❒ **Strategies** – There are two main types of strategies:

- A pure strategy is a single action:

$$\boxed{a \in \text{Actions}}$$

- A mixed strategy is a probability distribution over actions:

$$\forall a \in \text{Actions}, \quad \boxed{0 \leqslant \pi(a) \leqslant 1}$$

❒ **Game evaluation** – The value of the game $V(\pi_A,\pi_B)$ when player A follows π_A and player B follows π_B is such that:

$$\boxed{V(\pi_A,\pi_B) = \sum_{a,b} \pi_A(a)\pi_B(b)V(a,b)}$$

❒ **Minimax theorem** – By noting π_A,π_B ranging over mixed strategies, for every simultaneous two-player zero-sum game with a finite number of actions, we have:

$$\boxed{\max_{\pi_A}\min_{\pi_B} V(\pi_A,\pi_B) = \min_{\pi_B}\max_{\pi_A} V(\pi_A,\pi_B)}$$

2.3.3 Non-zero-sum games

❒ **Payoff matrix** – We define $V_p(\pi_A,\pi_B)$ to be the utility for player p.

❒ **Nash equilibrium** – A Nash equilibrium is (π_A^*,π_B^*) such that no player has an incentive to change its strategy. We have:

$$\boxed{\forall \pi_A, V_A(\pi_A^*,\pi_B^*) \geqslant V_A(\pi_A,\pi_B^*)} \quad \text{and} \quad \boxed{\forall \pi_B, V_B(\pi_A^*,\pi_B^*) \geqslant V_B(\pi_A^*,\pi_B)}$$

Remark: in any finite-player game with finite number of actions, there exists at least one Nash equilibrium.

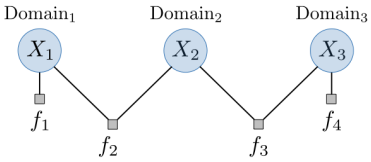
3 Variables-based models

3.1 Constraint satisfaction problems

In this section, our objective is to find maximum weight assignments of variable-based models. One advantage compared to states-based models is that these algorithms are more convenient to encode problem-specific constraints.

3.1.1 Factor graphs

❒ **Definition** – A factor graph, also referred to as a Markov random field, is a set of variables $X = (X_1,...,X_n)$ where $X_i \in \text{Domain}_i$ and m factors $f_1,...,f_m$ with each $f_j(X) \geqslant 0$.



❒ **Scope and arity** – The scope of a factor f_j is the set of variables it depends on. The size of this set is called the arity.

Remark: factors of arity 1 and 2 are called unary and binary respectively.

❒ **Assignment weight** – Each assignment $x = (x_1,...,x_n)$ yields a weight $\text{Weight}(x)$ defined as being the product of all factors f_j applied to that assignment. Its expression is given by:

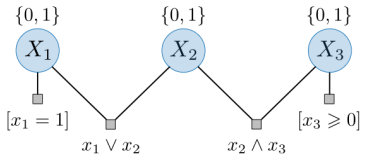
$$\boxed{\text{Weight}(x) = \prod_{j=1}^{m} f_j(x)}$$

❒ **Constraint satisfaction problem** – A constraint satisfaction problem (CSP) is a factor graph where all factors are binary; we call them to be constraints:

$$\boxed{\forall j \in [\![1,m]\!], \quad f_j(x) \in \{0,1\}}$$

Here, the constraint j with assignment x is said to be satisfied if and only if $f_j(x) = 1$.

❒ **Consistent assignment** – An assignment x of a CSP is said to be consistent if and only if $\text{Weight}(x) = 1$, i.e. all constraints are satisfied.



3.1.2 Dynamic ordering

❒ **Dependent factors** – The set of dependent factors of variable X_i with partial assignment x is called $D(x,X_i)$, and denotes the set of factors that link X_i to already assigned variables.

❒ **Backtracking search** – Backtracking search is an algorithm used to find maximum weight assignments of a factor graph. At each step, it chooses an unassigned variable and explores its values by recursion. Dynamic ordering (*i.e.* choice of variables and values) and lookahead (*i.e.* early elimination of inconsistent options) can be used to explore the graph more efficiently, although the worst-case runtime stays exponential: $O(|\text{Domain}|^n)$.

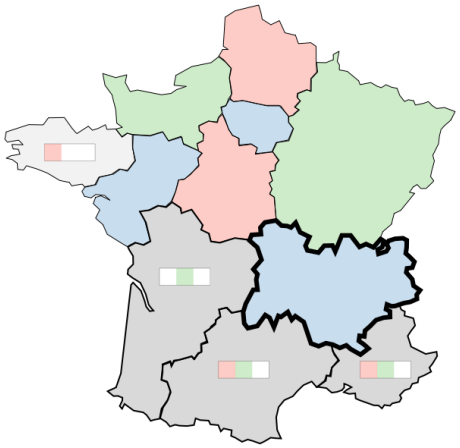
❒ **Forward checking** – It is a one-step lookahead heuristic that preemptively removes inconsistent values from the domains of neighboring variables. It has the following characteristics:

- After assigning a variable X_i, it eliminates inconsistent values from the domains of all its neighbors.
- If any of these domains becomes empty, we stop the local backtracking search.
- If we un-assign a variable X_i, we have to restore the domain of its neighbors.

❒ **Most constrained variable** – It is a variable-level ordering heuristic that selects the next unassigned variable that has the fewest consistent values. This has the effect of making inconsistent assignments to fail earlier in the search, which enables more efficient pruning.

❒ **Least constrained value** – It is a value-level ordering heuristic that assigns the next value that yields the highest number of consistent values of neighboring variables. Intuitively, this procedure chooses first the values that are most likely to work.

Remark: in practice, this heuristic is useful when all factors are constraints.



The example above is an illustration of the 3-color problem with backtracking search coupled with most constrained variable exploration and least constrained value heuristic, as well as forward checking at each step.

❒ **Arc consistency** – We say that arc consistency of variable X_l with respect to X_k is enforced when for each $x_l \in \text{Domain}_l$:

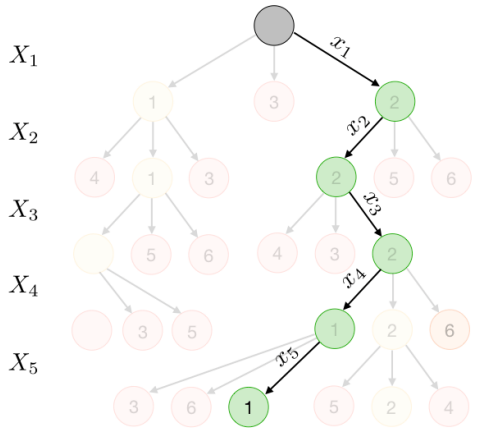
- unary factors of X_l are non-zero,
- there exists at least one $x_k \in \text{Domain}_k$ such that any factor between X_l and X_k is non-zero.

❒ **AC-3** – The AC-3 algorithm is a multi-step lookahead heuristic that applies forward checking to all relevant variables. After a given assignment, it performs forward checking and then successively enforces arc consistency with respect to the neighbors of variables for which the domain change during the process.

Remark: AC-3 can be implemented both iteratively and recursively.

3.1.3 Approximate methods

❒ **Beam search** – Beam search is an approximate algorithm that extends partial assignments of n variables of branching factor $b = |\text{Domain}|$ by exploring the K top paths at each step. The beam size $K \in \{1,...,b^n\}$ controls the tradeoff between efficiency and accuracy. This algorithm has a time complexity of $O(n \cdot Kb\log(Kb))$.
The example below illustrates a possible beam search of parameters $K = 2$, $b = 3$ and $n = 5$.



Remark: $K = 1$ corresponds to greedy search whereas $K \to +\infty$ is equivalent to BFS tree search.

❒ **Iterated conditional modes** – Iterated conditional modes (ICM) is an iterative approximate algorithm that modifies the assignment of a factor graph one variable at a time until convergence. At step i, we assign to X_i the value v that maximizes the product of all factors connected to that variable.

Remark: ICM may get stuck in local minima.

❒ **Gibbs sampling** – Gibbs sampling is an iterative approximate method that modifies the assignment of a factor graph one variable at a time until convergence. At step i:

- we assign to each element $u \in \text{Domain}_i$ a weight $w(u)$ that is the product of all factors connected to that variable,
- we sample v from the probability distribution induced by w and assign it to X_i.

Remark: Gibbs sampling can be seen as the probabilistic counterpart of ICM. It has the advantage to be able to escape local minima in most cases.

3.1.4 Factor graph transformations

❒ **Independence** – Let A,B be a partitioning of the variables X. We say that A and B are independent if there are no edges between A and B and we write:

$$\boxed{A,B \text{ independent} \iff A \perp\!\!\!\perp B}$$

Remark: independence is the key property that allows us to solve subproblems in parallel.

❒ **Conditional independence** – We say that A and B are conditionally independent given C if conditioning on C produces a graph in which A and B are independent. In this case, it is written:

$$\boxed{A \text{ and } B \text{ cond. indep. given } C \iff A \perp\!\!\!\perp B|C}$$

❒ **Conditioning** – Conditioning is a transformation aiming at making variables independent that breaks up a factor graph into smaller pieces that can be solved in parallel and can use backtracking. In order to condition on a variable $X_i = v$, we do as follows:

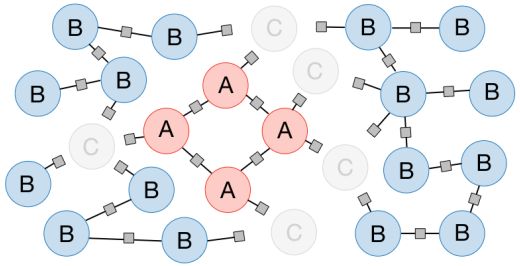
- Consider all factors $f_1,...,f_k$ that depend on X_i
- Remove X_i and $f_1,...,f_k$
- Add $g_j(x)$ for $j \in \{1,...,k\}$ defined as:

$$\boxed{g_j(x) = f_j(x \cup \{X_i : v\})}$$

❒ **Markov blanket** – Let $A \subseteq X$ be a subset of variables. We define MarkovBlanket(A) to be the neighbors of A that are not in A.

❒ **Proposition** – Let $C = \text{MarkovBlanket}(A)$ and $B = X \backslash (A \cup C)$. Then we have:

$$\boxed{A \perp\!\!\!\perp B|C}$$



❒ **Elimination** – Elimination is a factor graph transformation that removes X_i from the graph and solves a small subproblem conditioned on its Markov blanket as follows:

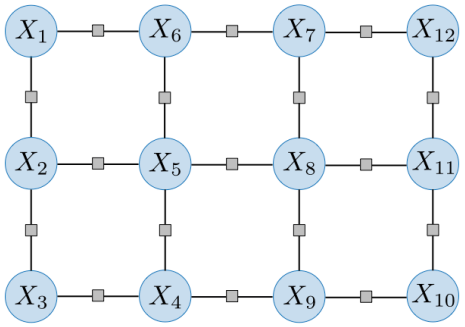
- Consider all factors $f_{i,1},...,f_{i,k}$ that depend on X_i
- Remove X_i and $f_{i,1},...,f_{i,k}$
- Add $f_{\text{new},i}(x)$ defined as:

$$\boxed{f_{\text{new},i}(x) = \max_{x_i} \prod_{l=1}^{k} f_{i,l}(x)}$$

❒ **Treewidth** – The treewidth of a factor graph is the maximum arity of any factor created by variable elimination with the best variable ordering. In other words,

$$\boxed{\text{Treewidth} = \min_{\text{orderings}} \max_{i \in \{1,...,n\}} \text{arity}(f_{\text{new},i})}$$

The example below illustrates the case of a factor graph of treewidth 3.



Remark: finding the best variable ordering is a NP-hard problem.

3.2 Bayesian networks

In this section, our goal will be to compute conditional probabilities. What is the probability of a query given evidence?

3.2.1 Introduction

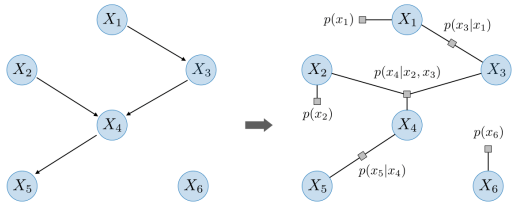
❒ **Explaining away** – Suppose causes C_1 and C_2 influence an effect E. Conditioning on the effect E and on one of the causes (say C_1) changes the probability of the other cause (say C_2). In this case, we say that C_1 has explained away C_2.

❒ **Directed acyclic graph** – A directed acyclic graph (DAG) is a finite directed graph with no directed cycles.

❒ **Bayesian network** – A Bayesian network is a directed acyclic graph (DAG) that specifies a joint distribution over random variables $X = (X_1,...,X_n)$ as a product of local conditional distributions, one for each node:

$$\boxed{P(X_1 = x_1,...,X_n = x_n) \triangleq \prod_{i=1}^{n} p(x_i|x_{\text{Parents}(i)})}$$

Remark: Bayesian networks are factor graphs imbued with the language of probability.



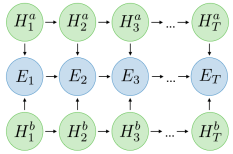
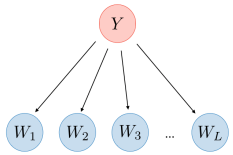
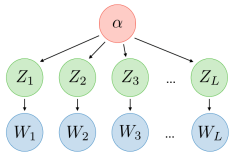
❒ **Locally normalized** – For each $x_{\text{Parents}(i)}$, all factors are local conditional distributions. Hence they have to satisfy:

$$\boxed{\sum_{x_i} p(x_i|x_{\text{Parents}(i)}) = 1}$$

As a result, sub-Bayesian networks and conditional distributions are consistent.

Remark: local conditional distributions are the true conditional distributions.

❒ **Marginalization** – The marginalization of a leaf node yields a Bayesian network without that node.

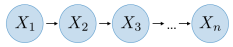
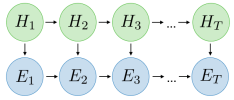
Factorial HMM	$H_t^o \underset{o\in\{a,b\}}{\sim} p(H_t^o\|H_{t-1}^o)$ $E_t \sim p(E_t\|H_t^a, H_t^b)$		Multiple object tracking
Naive Bayes	$Y \sim p(Y)$ $W_i \sim p(W_i\|Y)$		Document classification
Latent Dirichlet Allocation (LDA)	$\alpha \in \mathbb{R}^K$ distribution $Z_i \sim p(Z_i\|\alpha)$ $W_i \sim p(W_i\|Z_i)$		Topic modeling

3.2.2 Probabilistic programs

❒ **Concept** – A probabilistic program randomizes variables assignment. That way, we can write down complex Bayesian networks that generate assignments without us having to explicitly specify associated probabilities.

Remark: examples of probabilistic programs include Hidden Markov model (HMM), factorial HMM, naive Bayes, latent Dirichlet allocation, diseases and symptoms and stochastic block models.

❒ **Summary** – The table below summarizes the common probabilistic programs as well as their applications:

Program	Algorithm	Illustration	Example
Markov Model	$X_i \sim p(X_i\|X_{i-1})$		Language modeling
Hidden Markov Model (HMM)	$H_t \sim p(H_t\|H_{t-1})$ $E_t \sim p(E_t\|H_t)$		Object tracking

3.2.3 Inference

❒ **General probabilistic inference strategy** – The strategy to compute the probability $P(Q|E=e)$ of query Q given evidence $E=e$ is as follows:

- Step 1: Remove variables that are not ancestors of the query Q or the evidence E by marginalization
- Step 2: Convert Bayesian network to factor graph
- Step 3: Condition on the evidence $E=e$
- Step 4: Remove nodes disconnected from the query Q by marginalization
- Step 5: Run probabilistic inference algorithm (manual, variable elimination, Gibbs sampling, particle filtering)

❒ **Forward-backward algorithm** – This algorithm computes the exact value of $P(H=h_k|E=e)$ (smoothing query) for any $k \in \{1,...,L\}$ in the case of an HMM of size L. To do so, we proceed in 3 steps:

- Step 1: for $i \in \{1,...,L\}$, compute $F_i(h_i) = \sum_{h_{i-1}} F_{i-1}(h_{i-1})p(h_i|h_{i-1})p(e_i|h_i)$
- Step 2: for $i \in \{L,...,1\}$, compute $B_i(h_i) = \sum_{h_{i+1}} B_{i+1}(h_{i+1})p(h_{i+1}|h_i)p(e_{i+1}|h_{i+1})$
- Step 3: for $i \in \{1,...,L\}$, compute $S_i(h_i) = \frac{F_i(h_i)B_i(h_i)}{\sum_{h_i} F_i(h_i)B_i(h_i)}$

with the convention $F_0 = B_{L+1} = 1$. From this procedure and these notations, we get that

$$\boxed{P(H = h_k|E = e) = S_k(h_k)}$$

Remark: this algorithm interprets each assignment to be a path where each edge $h_{i-1} \to h_i$ is of weight $p(h_i|h_{i-1})p(e_i|h_i)$.

❒ **Gibbs sampling** – This algorithm is an iterative approximate method that uses a small set of assignments (particles) to represent a large probability distribution. From a random assignment x, Gibbs sampling performs the following steps for $i \in \{1,...,n\}$ until convergence:

- For all $u \in \text{Domain}_i$, compute the weight $w(u)$ of assignment x where $X_i = u$
- Sample v from the probability distribution induced by w: $v \sim P(X_i = v|X_{-i} = x_{-i})$
- Set $X_i = v$

Remark: X_{-i} denotes $X\backslash\{X_i\}$ and x_{-i} represents the corresponding assignment.

❒ **Particle filtering** – This algorithm approximates the posterior density of state variables given the evidence of observation variables by keeping track of K particles at a time. Starting from a set of particles C of size K, we run the following 3 steps iteratively:

- Step 1: proposal - For each old particle $x_{t-1} \in C$, sample x from the transition probability distribution $p(x|x_{t-1})$ and add x to a set C'.
- Step 2: weighting - Weigh each x of the set C' by $w(x) = p(e_t|x)$, where e_t is the evidence observed at time t.
- Step 3: resampling - Sample K elements from the set C' using the probability distribution induced by w and store them in C: these are the current particles x_t.

Remark: a more expensive version of this algorithm also keeps track of past particles in the proposal step.

❒ **Maximum likelihood** – If we don't know the local conditional distributions, we can learn them using maximum likelihood.

$$\boxed{\max_\theta \prod_{x \in \mathcal{D}_{\text{train}}} p(X = x; \theta)}$$

❒ **Laplace smoothing** – For each distribution d and partial assignment $(x_{\text{Parents}(i)}, x_i)$, add λ to $\text{count}_d(x_{\text{Parents}(i)}, x_i)$, then normalize to get probability estimates.

❒ **Algorithm** – The Expectation-Maximization (EM) algorithm gives an efficient method at estimating the parameter θ through maximum likelihood estimation by repeatedly constructing a lower-bound on the likelihood (E-step) and optimizing that lower bound (M-step) as follows:

- E-step: Evaluate the posterior probability $q(h)$ that each data point e came from a particular cluster h as follows:

$$\boxed{q(h) = P(H = h|E = e; \theta)}$$

- M-step: Use the posterior probabilities $q(h)$ as cluster specific weights on data points e to determine θ through maximum likelihood.

4 Logic-based models

4.1 Concepts

In this section, we will go through logic-based models that use logical formulas and inference rules. The idea here is to balance expressivity and computational efficiency.

❒ **Syntax of propositional logic** – By noting f,g formulas, and $\neg, \wedge, \vee, \to, \leftrightarrow$ connectives, here are the following possible logical expressions that we can write:

Name	Symbol	Translation
Negation	$\neg f$	not f
Conjunction	$f \wedge g$	f and g
Disjunction	$f \vee g$	f or g
Implication	$f \to g$	if f then g
Biconditional	$f \leftrightarrow g$	f, that is to say g

Remark: formulas can be built up recursively.

❒ **Model** – A model w is an assignment of truth values to propositional symbols.

❒ **Interpretation function** – Let f be a formula, w be a model, then the interpretation function $\mathcal{I}(f,w)$ is such that:

$$\boxed{\mathcal{I}(f,w) \in \{0,1\}}$$

❒ **Set of models** – We note $\mathcal{M}(f)$ the set of models w for which we have:

$$\boxed{\forall w \in \mathcal{M}(f), \quad \mathcal{I}(f,w) = 1}$$

❒ **Knowledge base** – A knowledge base KB is defined to be a set of formulas representing their conjunction, as follows:

$$\boxed{\mathcal{M}(\text{KB}) = \bigcap_{f \in \text{KB}} \mathcal{M}(f)}$$

❒ **Entailment** – A knowledge base KB that is said to entail f is noted $\text{KB} \models f$. We have:

$$\boxed{\text{KB} \models f \iff \mathcal{M}(\text{KB}) \subseteq \mathcal{M}(f)}$$

❒ **Contradiction** – A knowledge base KB contradicts f if and only if we have the following:

$$\boxed{\mathcal{M}(\text{KB}) \cap \mathcal{M}(f) = \varnothing}$$

Remark: KB contradicts f if and only if KB entails $\neg f$.

❒ **Contingency** – f is said to be contingent when there is a non-trivial overlap between the models of KB and f, i.e. when we have:

$$\boxed{\varnothing \subsetneq \mathcal{M}(\text{KB}) \cap \mathcal{M}(f) \subsetneq \mathcal{M}(\text{KB})}$$

Remark: we can quantify the uncertainty of the overlap of the two by computing the following quantity:

$$P(f|\text{KB}) = \frac{\sum_{w \in \mathcal{M}(\text{KB} \cup \{f\})} P(W=w)}{\sum_{w \in \mathcal{M}(\text{KB})} P(W=w)}$$

❒ **Satisfiability** – A knowledge base KB is said to be satisfiable if we have:

$$\boxed{\mathcal{M}(\text{KB}) \neq \varnothing}$$

❒ **Model checking** – Model checking is an algorithm that takes as input a knowledge base KB and checks whether we have $\mathcal{M}(\text{KB}) \neq \varnothing$.

Remark: popular model checking algorithms include DPLL and WalkSat.

❒ **Modus ponens inference rule** – For any propositional symbols $p_1,...,p_k,q$, we have:

$$\boxed{\frac{p_1,...,p_k, \quad (p_1 \wedge ... \wedge p_k) \longrightarrow q}{q}}$$

Remark: this can take linear time.

❒ **Inference rule** – If $f_1,...,f_k,g$ are formulas, then the following is an inference rule:

$$\boxed{\frac{f_1,...,f_k}{g}}$$

❒ **Derivation** – We say that KB derives f, and we note $\text{KB} \vdash f$, if and only if f eventually gets added to KB.

❒ **Soundness/completeness** – A set of inference rules can have the following properties:

Property	Meaning
Soundness	$\{f : \text{KB} \vdash f\} \subseteq \{f : \text{KB} \models f\}$
Completeness	$\{f : \text{KB} \vdash f\} \supseteq \{f : \text{KB} \models f\}$

4.2 Propositional logic

❒ **Definite clause** – By noting $p_1,...,p_k,q$ propositional symbols, we define a definite clause as having the following form:

$$\boxed{(p_1 \wedge ... \wedge p_k) \longrightarrow q}$$

Remark: the case when $q = false$ is called a goal clause.

❒ **Horn clause** – A Horn clause is defined to be either a definite clause or a goal clause.

❒ **Modus ponens on Horn clauses** – Modus ponens is complete with respect to Horn clauses if we suppose that KB contains only Horn clauses and p is an entailed propositional symbol. Applying modus ponens will then derive p.

❒ **Resolution inference rule** – The resolution inference rule is a generalized inference rule and is written as follows:

$$\boxed{\frac{f_1 \vee ... \vee f_n \vee p, \quad \neg p \vee g_1 \vee ... \vee g_m}{f_1 \vee ... \vee f_n \vee g_1 \vee ... \vee g_m}}$$

Remark: this can take exponential time.

❒ **Conjunctive normal form** – A conjunctive normal form (CNF) formula is a conjunction of clauses.

❒ **Conversion to CNF** – Every formula f in propositional logic can be converted into an equivalent CNF formula f':

$$\boxed{\mathcal{M}(f) = \mathcal{M}(f')}$$

❒ **Resolution-based inference** – The resolution-based inference algorithm follows the following steps:

- Step 1: Convert all formulas into CNF
- Step 2: Repeatedly apply resolution rule
- Step 3: Return unsatisfiable if and only if derive false

4.3 First-order logic

The idea here is that variables yield compact knowledge representations.

❒ **Model** – A model w in first-order logic maps:

- constant symbols to objects
- predicate symbols to tuple of objects

❒ **Definite clause** – By noting $x_1,...,x_n$ variables and $a_1,...,a_k,b$ atomic formulas, a definite clause has the following form:

$$\boxed{\forall x_1,...,\forall x_n, \quad (a_1 \wedge ... \wedge a_k) \rightarrow b}$$

❒ **Modus ponens** – By noting $x_1,...,x_n$ variables and $a_1,...,a_k,b$ atomic formulas, a modus ponens has the following form:

$$\boxed{\frac{a_1,...,a_k \quad \forall x_1,...,\forall x_n (a_1 \wedge ... \wedge a_k) \rightarrow b}{b}}$$

❒ **Substitution** – A substitution θ is a mapping from variables to terms. For instance, $\text{Subst}(\theta,f)$ returns the result of performing substitution θ on f.

❒ **Unification** – Unification takes two formulas f and g and returns a substitution θ which is the most general unifier:

$$\boxed{\text{Unify}[f,g] = \theta} \quad \text{s.t.} \quad \boxed{\text{Subst}[\theta,f] = \text{Subst}[\theta,g]}$$

or fail if no such θ exists.

❒ **Completeness** – Modus ponens is complete for first-order logic with only Horn clauses.

❒ **Semi-decidability** – First-order logic, even restricted to only Horn clauses, is semi-decidable.

- if KB $\models f$, forward inference on complete inference rules will prove f in finite time
- if KB $\not\models f$, no algorithm can show this in finite time

❒ **Resolution rule** – By noting $\theta = \text{Unify}(p,q)$, we have:

$$\boxed{\frac{f_1 \vee ... \vee f_n \vee p, \quad \neg q \vee g_1 \vee ... \vee g_m}{\text{Subst}[\theta, f_1 \vee ... \vee f_n \vee g_1 \vee ... \vee g_m]}}$$