LEARNING

Inclusion: Deep Learning ⊂ Representation Learning ⊂ Machine Learning (ML) ⊂ AI

ML definition (T. M. Mitchell): A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

Notations: X: input space, $X \subset \mathbb{R}^d$, Y: output/target space, *d*: dimension of the inputs, $x \epsilon\ \mathbb{R}^d$, $x = (x_1, \dots, x_d)$, x_i : "feature"

Learning: find a mapping function X -- f--> Y, also called "**model**". Usually, a model is a **parametric** function, f_θ, i.e. the goal of learning is to find the best parameters θ

Learning paradigms

Supervised learning: Y is known. Goal: estimate $P(y|x)$. i.e.: predict the label $y = "cat"$ from that image $x =$ 

Unsupervised learning: Y is not known. The goal is to estimate $P(x)$ (density estimation)

Reinforcement learning: learn some actions depending on rewards and environment constraints

Supervised and unsupervised learning = training with examples (i.e. the experience "E" are samples from X: X^{train})

Datasets : X^{train}, X^{val}, X^{test}

X^{train} : at training time, use to compute the best estimator $\hat{f}_\theta$ of f_θ.

$$X^{train} = \begin{bmatrix} x_1^{(1)} & \cdots & x_d^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(m)} & \cdots & x_d^{(m)} \end{bmatrix}, m \text{ training samples}$$

X^{val} : at training time, use to evaluate the pertinence of $\hat{f}_\theta$ with unseen data (evaluate generalization capacity)

X^{test} : at test/inference time, use to evaluate the performance of the final learned model $\hat{f}_\theta$ with fresh unseen data.

Golden rule: X^{val} and X^{test} are never used to train the model, only for evaluation purpose.

K-cross validation: methodology to select a model. Split the training dataset in K folds and iteratively use one fold as the validation set.

OVERFITTING

The model $\hat{f}_\theta$ is learning the training samples $\{X^{train}, Y^{train}\}$ (for supervised learning) by heart (thus, train_error is very good). But the model is not able to generalize, it performs badly with fresh unseen data (test_error is awful).

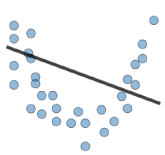
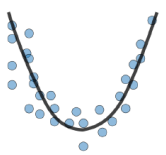
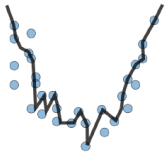
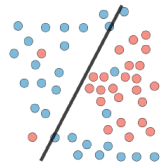
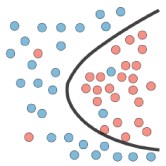
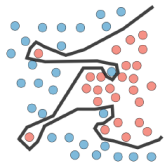
How to detect overfitting ? At learning time: by regularly using unseen data (X^{val}) and comparing the performance with X^{train} and X^{val}. At test time: by checking the performance with a coherent test dataset, X^{test}

Bias / Variance tradeoff

Bias of a model is the difference between the expected prediction and the correct model that we try to predict for given data points. *Variance* of a model is the variability of the model prediction for given data points.

Bias/variance tradeoff: the simpler the model, the higher the bias (underfitting), and the more complex the model, the higher the variance (overfitting).

Why overfitting? Two main reasons: (1) **too much capacity** (add regularization, try simple model at first), (2) **not enough training data** (try data augmentation)

	Underfitting	Just right	Overfitting
Symptoms	- High training error - Training error close to test error - High bias	- Training error slightly lower than test error	- Low training error - Training error much lower than test error - High variance
Regression			
Classification			
Deep learning			
Remedies	- Complexify model - Add more features - Train longer		- Regularize - Get more data

UNSUPERVISED LEARNING

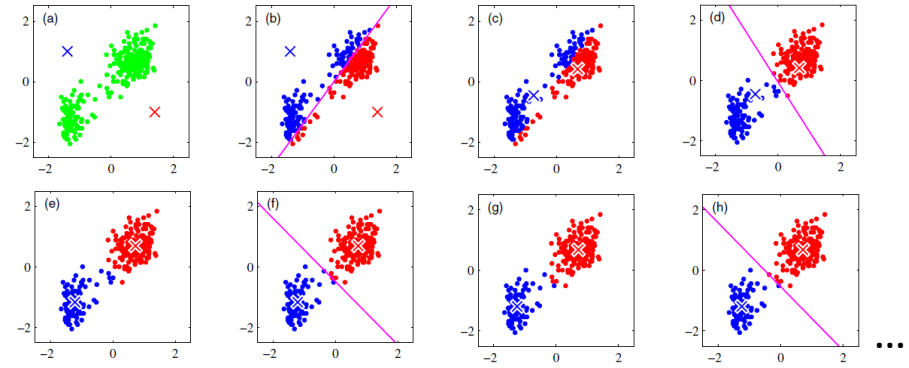
Goal: learn P(x), i.e. find (hidden) structures in data. Major tasks related to unsupervised learning: clustering, outliers detection, dimensionality reduction

DENSITY ESTIMATION. Estimate the underlying properties of the data distribution ($p(x)$)

UNSUPERVISED LEARNING. K-MEANS

Goal: Split the data into K clusters. Each sample belongs to (only) one cluster (partition), the one with the nearest centroid. K-Means minimizes the intra-class variance.

Algorithm. (1) Init K centroids (e.g., randomly or with "K-Means ++") // (2) Assign each sample to the closest centroid. // (3) Update the centroid by computing the mean of the samples belonging in the cluster // (4) Iterate step 2 and 3 until convergence.

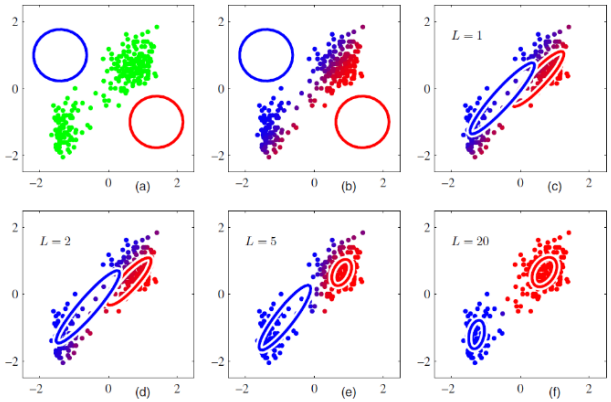


Main issues of (classical) K-Means: (1) How to fix K, the number of clusters ? (2) Stopping criterion : fix the number of iterations, stable cluster assignment, no intra-class variance improvements (3) How to process dynamic dataset (--> online K-Means)

EXPECTATION-MAXIMIZATION WITH GAUSSIAN MIXTURE

Goal: cluster the data with a mixture of K-Gaussian

Algorithm: For $x \epsilon \mathbb{R}^d$, a mixture of K Gaussian is done by $p(x) = \sum_{k=1}^{K} \alpha_k N(x|\mu_k, \Sigma_k)$ with N a multivariate Gaussian with mean $\mu_k \epsilon \mathbb{R}^d$ and covariance matrix $\Sigma_k \epsilon \mathbb{R}^{dxd}$ (a dxd matrix).

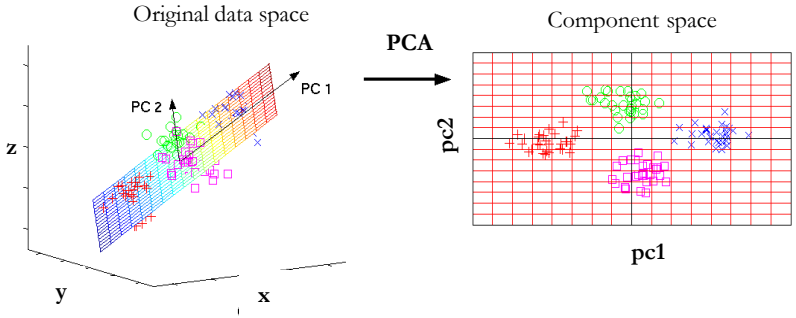


PRINCIPAL COMPONENT ANALYSIS

Goal: Dimensionality reduction / Features extraction (or elimination). Find a new space where data are uncorrelated, i.e. where features are independent of one another (then, more easily interpretable, usable…)

Algorithm: The dataset is composed of N samples $x \epsilon \mathbb{R}^D$, representing in a NxD matrix: X. The covariance matrix is $C=X^TX$.

(1) Decompose the matrix C with eigenvectors decomposition, $C=PDP^T$. D is a diagonal matrix with the eigenvalues. P is the set of the eigenvectors. (2) Keep the $d < D$ first eigenvalues and the associated eigenvectors from P. It gives a projection matrix P'. (3) With P', we can project X (P'X) in a new space in $\mathbb{R}^d$ where data are less correlated.



SUPERVISED LEARNING

Two major supervised tasks: REGRESSION and CLASSIFICATION

	Regression	Classification
Output	Continuous, $y \epsilon \mathbb{R}$	Label (class, category…)
Examples	Linear regression	Logistic regression, Naive Bayes, decision tree, SVM

Type of models

	Discriminative model	Generative model
Goal	Directly estimate $P(y\|x)$	Estimate $P(x\|y)$ to deduce $P(y\|x)$
What is learned ?	Decision boundary	Probability distribution of the data
Illustration		
Examples	Regressions, SVMs	Naive Bayes (exp: GaussianNB)

Illustrations: Pattern Recognition and Machine Learning, C.M. Bishop // http://www.nlpca.org/

SUPERVISED LEARNING

Supervised learning as an optimization problem:

$$\min_{\theta} \frac{1}{m}\sum_{i=0}^{m-1} \mathcal{L}\left(y^{(i)}, f(x^{(i)}, \theta)\right) + \lambda.\Omega(\theta)$$

Regularization

Regularization (avoid overfitting)

L2 (Ridge): … + $\lambda.\|\theta\|_2^2$, makes coefficients θ smaller. *L1 (Lasso):* … + $\lambda.\|\theta\|_1$, shrinks coefficient to 0 (sparcity), parameters selection.

Loss function: $\mathcal{L}$ quantifies the difference between the expected output y and the predicted output $\hat{y} = f_\theta(x)$

$\mathcal{L}(y,\hat{y}) = (y-\hat{y})^2$ Least squared loss (Linear Regression)// $\mathcal{L}(y,\hat{y}) = max(0,1-y\hat{y})$ Hinge loss (SVM) //$\mathcal{L}(y,\hat{y}) = log(1+exp(-y\hat{y}))$ Logistic loss (Logistic Regression) // $\mathcal{L}(y,\hat{y}) = -(ylog(\hat{y}) + (1-y)log(1-\hat{y}))$ Cross entropy loss (Neural network)

LINEAR REGRESSION

The prediction is a simple linear combination of the features:$\hat{y} = \theta_1 x_1 + \cdots + \theta_d x_d = \Theta^T x$. Goal: find the best parameters θ_i. One optimal solution minimizes the cost function: $\Theta = (X^T X)^{-1} X^T y$ (aka Normal Equations)

LOGISTIC REGRESSION (LR)

Binary classification: map a linear model with the logistic function, $g(z) = \frac{1}{1+\exp(-z)}$, $g(z)\epsilon]0,1[$

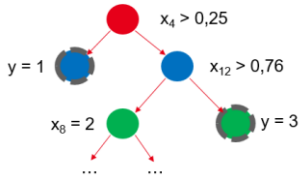
$p(y=1|x) = g(\Theta^T x)$, decision rule: p>0.5 => $\hat{y}$=1, else $\hat{y}$=-1

Multiclass problem (multiclass logistic regression or « softmax regression »), generalization of logistic regression for K classes. K sets of parameters, $\Theta_1, \ldots, \Theta_K$.

$$p(y=i|x) = \frac{\exp(\Theta_i^T x)}{\sum_{j=1}^{K} \Theta_j^T x}$$

DECISION TREE

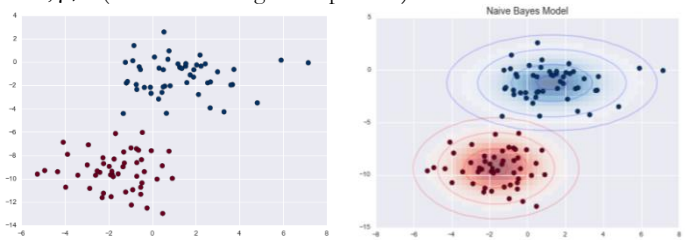
Split the data as a tree using attributes related to the features. Main interest of Decision Tree: Interpretable model. But it is sensitive to overfitting problem (solution: random forest)



Classical metrics to split the tree are entropy-based

NAIVE BAYES (NB) CLASSIFIER

Estimate $P(x|y)$ to deduce what we are looking for: $P(y|x)$. Gaussian NB: $P(x|y)$ follows Gaussian law, μ, σ (which is a strong assumption…)



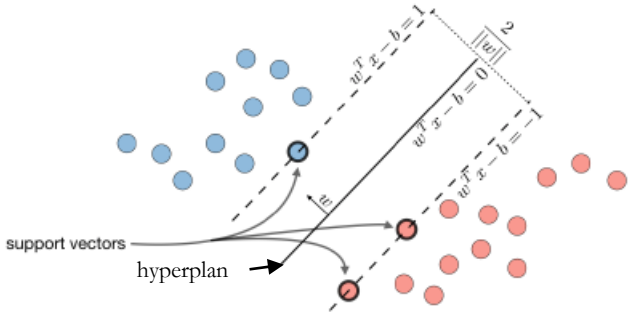
Rely on a strong and sometimes unrealistic assumption but pretty fast, few parameters, easily interpretable, useful for well-separated categories and high-dimensional data.

SUPPORT VECTOR MACHINE (SVM)

Goal: find the best hyperplan H: $\Theta^T x - b = 0$ as an optimal margin classifier, such that the decision is done with $f_\Theta(x) = sign(\Theta^T x - b)$.

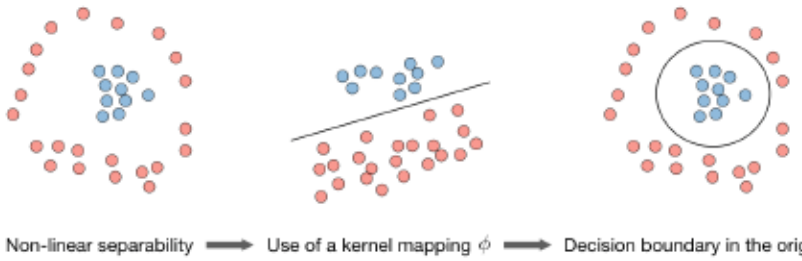
The resulting optimization problem is: $\min\|\Theta\|^2$ such that $y^{(i)}\left(\Theta^T x^{(i)} - b\right) \geq 1$ (Eq.1)

The support vectors are the training samples lying on the margin $\gamma = \frac{1}{\|\Theta\|}$



Hard/Soft margin: by relaxing Eq. 1, we can tolerate some data to lie within the margin. The force of the relaxation is done with an hyperparameter ("C", small C: soft margin) Hard margin: SVM is very sensitive to outliers.

For linearly non separable data: **KERNEL TRICK**. Use a Kernel, expressed as the dot product of a feature mapping function ϕ ($K(x,x') = \phi(x)^T\phi(x')$) to project data in a new space.



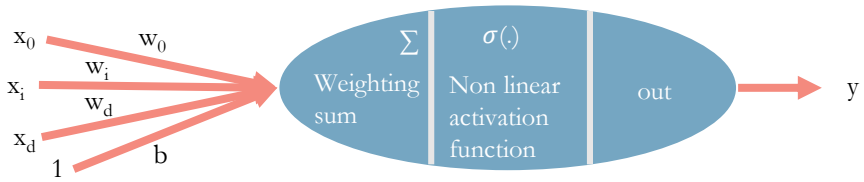
References: Afshine Amidi: https://github.com/afshinea

REPRESENTATION LEARNING

Traditional ("old school") Machine Learning algorithms need to manually extract features from raw data (e.g., color histograms for image classification before using SVM). **Representation Learning** algorithms take raw data as inputs, the features extractions (representation) is a part of the learning process.

NEURON & PERCEPTRON

A (formal) neuron:



The historic Perceptron is a one neuron model with Heaviside function as activation. Thus, the prediction of the Perceptron is very close to logistic regression:

$$\hat{y} = g(w_1 x_1 + \cdots + w_d x_d) = g(W^T x)$$

With the Heaviside function: $g(z) = 1 \; if \; z > 0, \mathrm{g}(z) = 0 \; otherwise$.

SINGLE LAYER NEURAL NETWORK & UNIVERSAL APPROXIMATION THEOREM

We can stack several neurons within a « layer » and combine the different outputs: single layer Neural Network.

The universal approximation theorem states that every continuous function, f, can be approximated, according to an error level ($\boldsymbol{\varepsilon}$), with a single-layer neural network: $\exists n$ and parameters $a, b \in \mathbb{R}^n$ and $W \in \mathbb{R}^{nxd}$

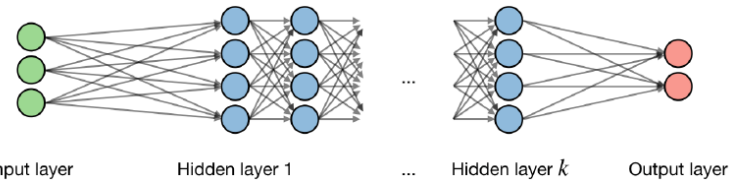
$$\left| \sum_{i=1}^{n} a_i \sigma(w_i^T x + b_i) - f(x) \right| < \varepsilon$$

(DEEP) NEURAL NETWORKS

Neurons are also called "units". We can stack several layers of units. Intermediate layers are called « hidden layers »:

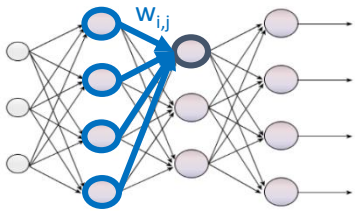
All neurons of an hidden layer (i) share the same activation: g_i. Then the unit (j) has the output: $o_j^{(i)} = g_i\left(w_j^{(i)T} x + b_j^{(i)}\right)$

The "input layer" simply collects the (raw) input data. For example, for 1000 pixels images, the input layer will have 1000 inputs: first input will be the first pixel, second input the second pixel, etc.



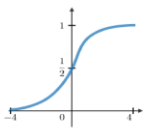
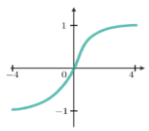
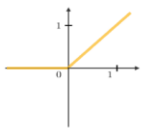
A Multi-Layer Perceptron (MLP) is a Fully Connected (FC) neural network, because a neuron receive information from ALL the neurons of the previous layer.
Each connection is a weight $w_{i,j}$ to learn.

A Deep Neural Network is powerful because it manages to extract low-level features at the first layers then, high-level features at the last layers.



ACTIVATION FUNCTIONS

Activation functions are fundamental in neural networks since they inject nonlinearity. Traditional activations for deep learning are:

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

SOFTMAX : $S(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$

➔ $S(x_i) \in [0,1]$
➔ $\sum_i S(x_i) = 1$

Additionally, **SOFTMAX** function enables to map a set of outputs to a set of probabilities.

References: Afshine Amidi: https://github.com/afshinea

DEEP NEURAL NETWORKS ARCHITECTURES

Two major neural networks: **FEEDFORWARD** (MLP, Convolutional neural networks) and **RECURRENT**.

CROSS-ENTROPY LOSS

For neural networks (NN), the cross-entropy loss is commonly used:

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

With y the expected output (Y^{train}) and $\hat{y}$ the prediction produces by the neural network ($\hat{y} = NN_W(x)$).

TRAINING NEURAL NETWORKS WITH BACKPROPAGATION

Each weight is iteratively updated with a portion of the gradient of the loss between the expected output y and the prediction $\hat{y}$. The "portion" is defined by the **learning rate** parameter (λ) : $\boldsymbol{w_{t+1} = w_t - \lambda \nabla_{w_t} L(\hat{y}, y)}$

We use the chain rule for the gradient : $\frac{\partial L(\hat{y},y)}{\partial w} = \frac{\partial L(\hat{y},y)}{\partial o} \cdot \frac{\partial o}{\partial z} \cdot \frac{\partial z}{\partial w}$ ($\frac{\partial o}{\partial z}$ is activation first derivate)

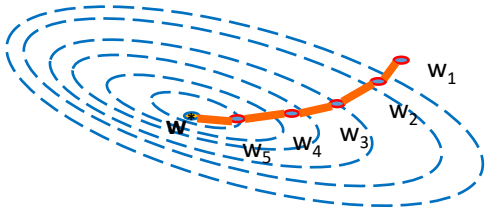
Then, training (update the weights) is done in 4 steps repeated iteratively:

(1) take a batch of training data.

(2) forward propagation through the network to get $L(\hat{y}, y)$

(3) backpropagate the loss to compute the gradients, $\nabla_w L(\hat{y}, y)$

(4) Use $\nabla_w L(\hat{y}, y)$ to update weights (logically, the last weights will be the first updated)



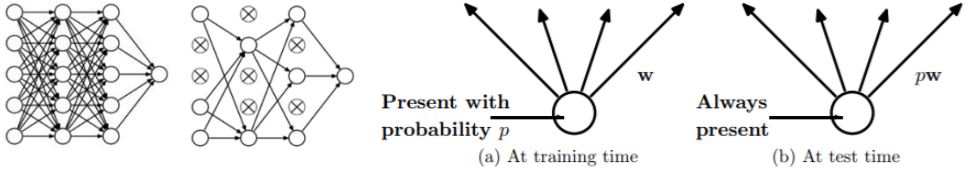
An **EPOCH** is when every training data has been used. Training a deep neural network can use several epochs.

Different strategies ("**optimizer**") can be used to intelligently adapt the learning rate. Classical optimizers are Adam, Adadelta or RMSProp.

Weights can be updated after each training sample (Stochastic Gradient Descent) but usually we use a mini-batch of training data ("**mini-batch SGD**")

REGULARIZATION & DROPOUT

To avoid overfitting, Deep Neural Networks usually need regularization like classical L1 or L2 penalty (see course 1). Dropout is a simple and efficient regularization technique to prevent overfitting by simply dropping out units in a neural network with the probability p (fixed by the user).



CONVOLUTIONAL NEURAL NETWORKS

This type of feedforward neural networks is highly efficient for "spatial" data such as images. The basic idea is to train Convolutional filters (or Kernels). Convolution is a major tool in signal processing to extract "useful" features.

Using a deep CNN on images, first convolutional layers learn to extract simple features (e.g., edges) and the last layers learn to extract high-level features (sometimes clearly identifiable patterns).

CNN is traditionally composed of several layers composed of **Convolution** / **Activation** / **Pooling**

A **CONVOLUTIONAL layer** is defined by:

- The **number of filters** and the **size of the filters**: exp, 64 filters of size K=3 (i.e., 3x3)
- The **Stride** controls how the filter convolves around the input. With stride=1, the filter convolves around the input by shifting one unit at a time.
- **Padding**: how to handle the border? « *same* » means that the output has the same size than the input (usually use zero outside the input). « *valid* » means that the output has size N-K+1.
- If W is the input size, K the size of the filter, P the amount of zero padding, S the stride, then the size of output is $N = \frac{W-K+2P}{S} + 1$

Convolutions have strong properties: sparse connectivity, parameter sharing and equivariant representation. Because CNN learn the weights of the convolutional filters that are shared (i.e. used by every unit of the layer), a CNN has less parameters (weights) to learn than a traditional Fully Connected network (MLP).

ACTIVATION is exactly the same process as a classical multi-layer perceptron. ReLu is a classical choice. After convolution and activation, a **POOLING** processing is performed. Pooling enables translation invariance and is a kind of summary statistic of the outputs. Classical pooling are « Max pooling » or « Mean pooling ». We find the same « stride » parameter as for convolution.

VANISHING GRADIENT

A major issue that makes DNNs hard to train is the Vanishing Gradient problem: during the backpropagation process, we use several products of gradient values and weights that are small. With several hidden layers, it's worse and worse. This is typically true for the sigmoid activation function and explains why ReLU is used for training deep neural networks.

Batch Normalization is a powerful way to fight vanishing gradient and train deeper models. At training time, for each batch, we normalize the activations with the mean and standard deviation.

EXAMPLE OF A CNN ARCHITECTURE (WITH TENSOR SIZE)

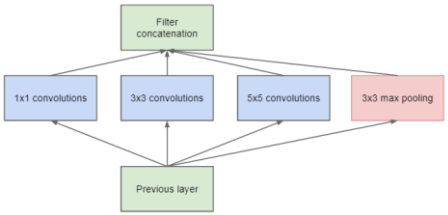
We classify (into **10** classes) color images of size **32x32** pixels with **3** channels (R, G, B) with a CNN composed of 3 conv layers (**padding=valid**, **filter_size=(3,3)**, **stride=(1,1)**) with **32**, **64** and **64** filters respectively. Each convolution is followed by a pooling (Max) with size=(2,2).

Name // Size // Nb params	Comment
Input // (X, 32,32,3) // 0	1st dimension (X) is assigned to the size of the mini-batch.
Conv2D // (X, 30, 30, 32) // 896	30 not 32 because of padding=valid. The 1st conv layer has 32 filters. Params = 3*3*3*32 + 32(bias)
MaxPooling2D // (X, 15, 15, 32) // 0	Classical pooling halve the tensor size (not the channel dimension)
Conv2D // (X, 13,13,64) // 18496	2nd and 3rd conv layers have 64 filters
MaxPooling2D // (X, 6,6,64) // 0	
Conv2D // (X, 4,4,64) // 36928	
Flatten // (X, 1024) // 0	1024 = 4*4*64
Dense // (X, 64) // 65600	A 64 neurons fully connected layer
Dense // (X, 10) // 650	Last layer with Softmax for prediction

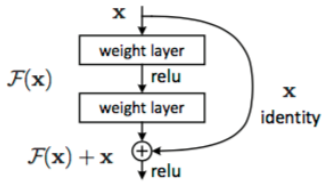
References: Afshine Amidi: https://github.com/afshinea

ADVANCED CNN ARCHITECTURES

Inception block: learn several filter sizes (wider rather than deeper network).



Residual block: fight the vanishing gradient issue by offering a skiping path.



1x1 convolution: a simple dimension reduction trick: squeeze the tensor among the depth, i.e. the number of filters. Can be seen as a feature pooling. Heavily used with inception and residual-based networks.

NATURAL LANGUAGE PROCESSING

Neural networks are relevant for text processing. Usually a word is represented with sparse vectors (high dimensional) thanks to a vocabulary. Word context is simply the window of size *w* nearby the word.

With Machine Learning, one can learn *word context* and *word embedding*. Word embedding is the representation of a word in a small, dense vector. Similar words will have similar representations. Best example: **Word2Vec** (2 layers neural network trained to learn the linguistic context of a word).

LANGUAGE MODEL

Language model = predict what word comes next. $P(w^{t+1}=w_j | x^1,\ldots, x^t)$ where w_j is a word from a vocabulary V. Traditional neural networks (MLP) are not suitable for processing sliding window over text. Recurrent neural networks are the solution.

A traditional performance metric to evaluate a language model is the **Perplexity**. The lower the better:

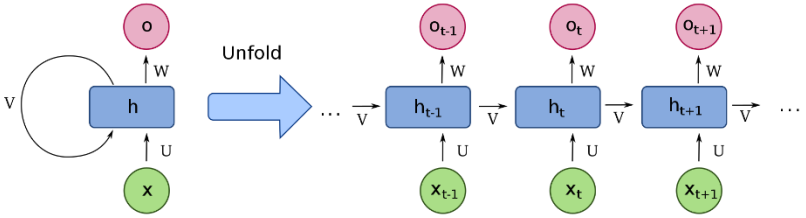
$$PP = \prod_{t=1}^{T} \left(\frac{1}{\sum_{j=1}^{|V|} y_j^{(t)} \cdot \hat{y}_j^{(t)}} \right)^{1/T}$$

Inverse probability of dataset

Normalized by number of words

RECURRENT NEURAL NETWORKS

Recurrent Neural Networks (RNN) deal with temporal / sequential data.

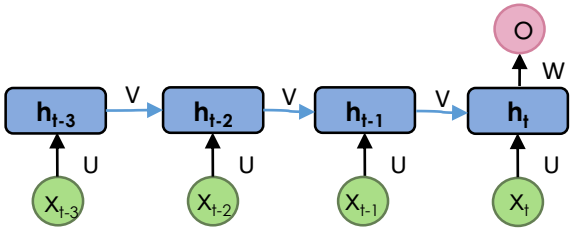


The basic idea is to learn a set of weights (V) associated to the previous state (t-1) of the hidden layer *h*.

$h_t = \emptyset(Ux_t + Vh_{t-1})$ and $o_t = Wh_t$.

Important: the weight matrix U, V and W are shared across time.

Different typologies of connections can be proposed (hidden-to-hidden, output-to-hidden, sequence hidden-to-hidden). Classical RNN suffer from Vanishing Gradient.



A sequence hidden-to-hidden with a time delay of 4 (4 words to predict the next one)

Train a RNN. A classical loss is a cross entropy loss function. The backpropagation is simply the sum of the gradients over time (remember that the weight matrix is shared over time).

Problem of RNN: A RNN has a short memory because of the composition of tanh and the shared weights that squeeze the influence of the first inputs. To fix that issue other architectures have been proposed such as **Long-Short-Term Memory** neural networks (**LSTM**) that aims at accumulating information and forgetting useless information.

ATTENTION MECHANISM

Attention mechanism aims at providing for each word of a sentence the importance of each word according to the other. It is a powerful mechanism to have accurate context information.

Self-attention mechanism are based on 3 vectors : **Q**UERY, **K**EY, **V**ALUE. The attention representation **A** is given by:

$$\boldsymbol{Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V}$$

With d_k the dimension of q and k vectors. This formula is known as the "***scaled dot-product Attention***".

Example:

$x^{(1)}$	$x^{(2)}$	$x^{(3)}$	$x^{(4)}$	$x^{(5)}$
Amélie	**visite**	**Rome**	**en**	**juillet**

NB: $x^{(i)}$ is the representation of the i^{th} word after classical word embedding.

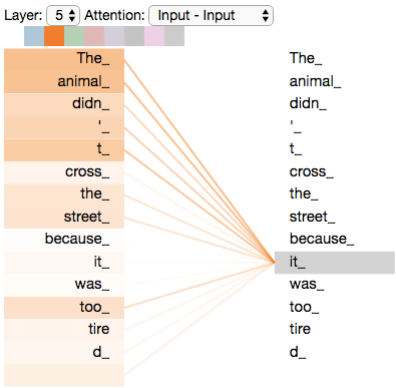
The Attention vector of "Rome" is:

$$A^{(3)} = A(q^{(3)},k,v) = \sum_i \frac{exp(<q^{(3)},k^{(1)}>)}{\sum_j exp(<q^{(3)},k^{(j)}>)} v^{(i)}$$

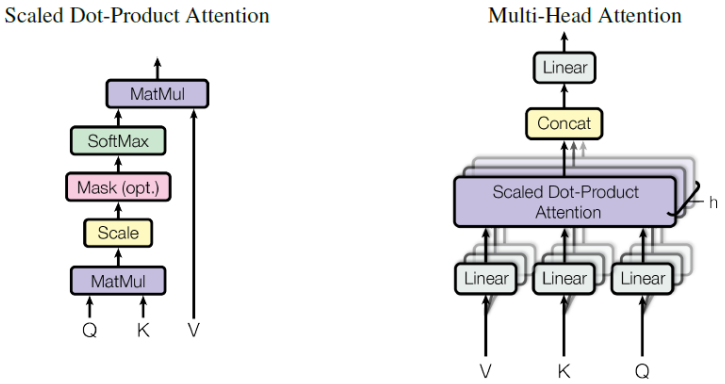
We can process several attention mechanisms in parallel, so that we can capture several contexts ➔ Multi-Head Attention

The attention vector of "Rome" is likely to highlight the importance of "visite". An illustration with another sentence:

Attention of "it" highlights "The animal"

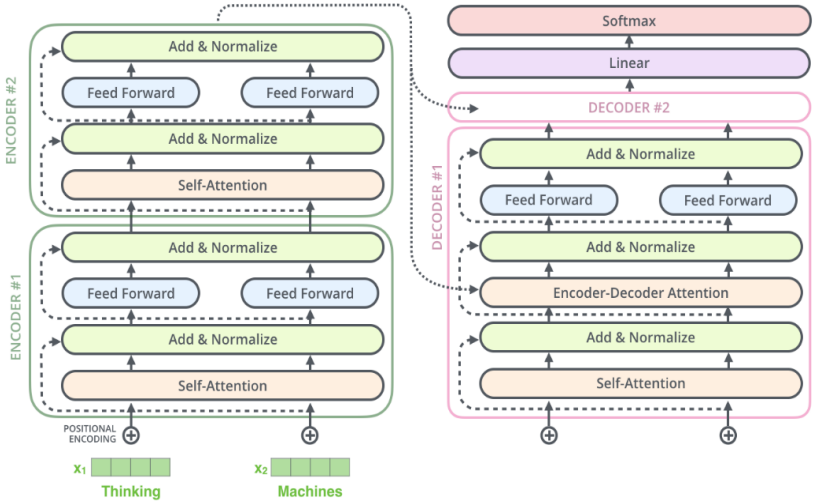


We can process several attention mechanisms in parallel, so that we can capture several contexts ➔ Multi-Head Attention



TRANSFORMER MODELS

Goal: Transformer models are only based on MLP and Attention blocks. No recurrent units ! State-of-the-art performances but need a very large amount of data.



Transformers also work well for vision tasks. The trick is to split an image into patches (like words in a sentence…) and feed to a transformer architecture…

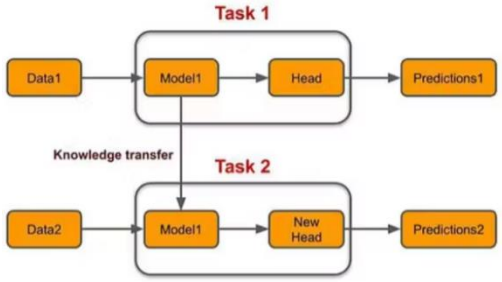
Transformers are huge models that we usually train with **SELF SUPERVISED LEARNING (SSL)**. SSL exploits huge amount of unlabeled data by learning **pretext tasks** (that use the data itself to create « labels »). For example, we mask words in sentences and learn to recover the missing words.

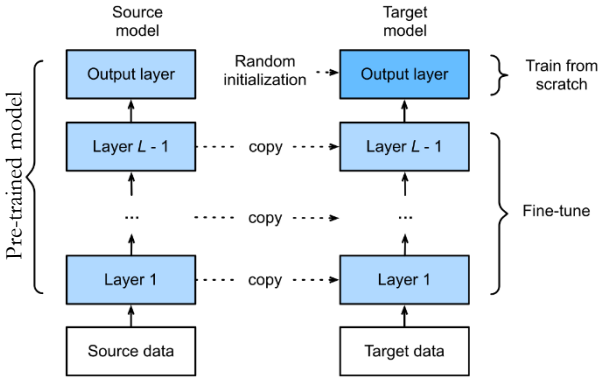
Another example if for computer vision: split a picture into patches and learn to place the patches in the correct order (like a jigsaw…)

TRANSFER LEARNING & FINE-TUNING

Basic idea: use the knowledge of a **pre-trained model**. We define a *source* and a *target tasks* and a *source* and a *target domains*. For different domains, we talk about **Domain Adaptation**. Transfer learning is possible because of Representation Learning since a model learn to extract different level of features.

A classical method for **transfer learning** is to freeze the first layers of a pre-trained model (provide features) and train only a classifier on top:



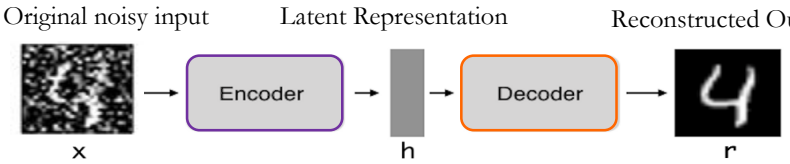


In a **fine-tuning** scenario, we only train specific layers of the pre-trained model on target data.

AUTOENCODER

Autoencoder are neural networks for unsupervised learning. The goal is to learn to encode and decode the input. I.e. for autoencoder y=x. The intermediate representation is called the **latent representation, h** : AE(x) = Decode(Encode(x)) = x with Encode(x)=h.

Denoising autoencoders learn to reconstruct noisy inputs



Variational autoencoder (VAE). By sampling the latent space with the decoder part, we can **generate** new samples from X. For now, we saw supervised **discriminative model** (i.e. classification or regression), here is a first step to **generative model**. Problem: the latent space is not well-distributed. VAE adds a constraint on the distribution of the latent space (for example a Gaussian distribution). That means a VAE has two loss: (1) one for the reconstruction error – classical loss such as MSE; (2) one – Kullback Leibler divergence – for the latent distribution constraint.

GENERATIVE AI

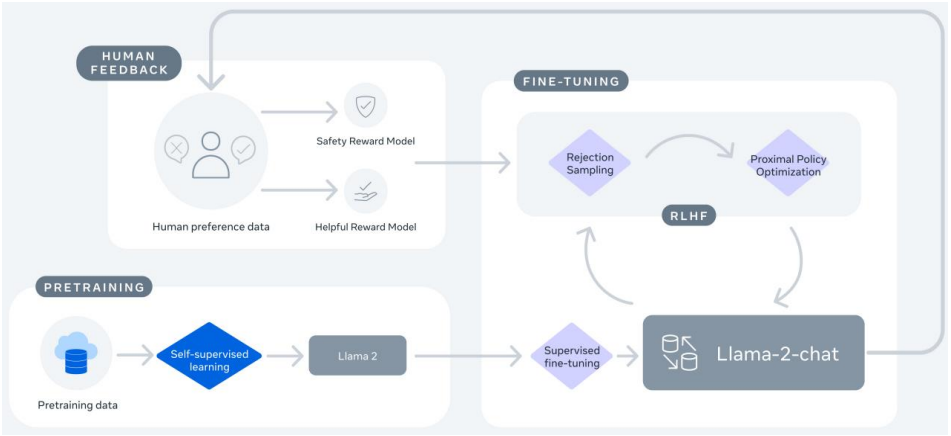
With a predictive model, you assume there is one correct output for your input (e.g., this image is a cat). With a **generative model** you assume there may be multiple correct outputs (e.g., if you ask the same question several times, chatGPT will output different answers). The goal of a generative model is classically : $P(x|c)$ where c is a condition such as a prompt. A « prompt » (c) could be a text condition, or an image, or audio…

With a generative model, you can perform **density estimation** and **sampling**, **imputation** (filling missing values), **latent space arithmetic** (remember word2vec)…

VAE are generative models. Generative Adversarial Network (**GAN**) are also famous ones. In a GAN, you jointly train a GENERATOR model (the « artist ») and a DISCRIMINATOR (the « art critic »). The goal is to improve the generator so that it fools the discriminator.

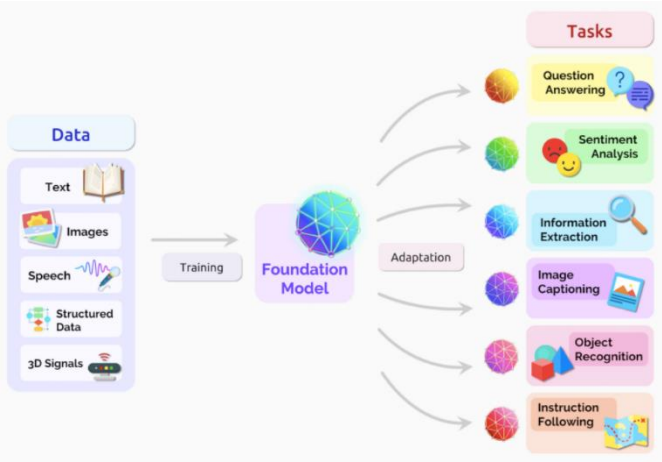
Modern AI systems are based on transformer-based generative models. Typical examples are chat-bot systems such as **chatGPT**. For NLP, such systems are based on:

(1) A pre-trained language model trained on massive datasets (e.g., GPT-3, LLAMA 2)
(2) Supervised Fine-Tuning (**SFT**) for chat task (with annotated resources)
(3) Reinforcement Learning with Human Feedback (**RLHF**) for **Model Alignment**: align the model with human requirements (mainly safety & ethics)



FOUNDATION MODELS

A key concept and a new paradigm is **Foundation Models** that are very big models trained on broad data, usually thanks to self supervised learning that can be **adapted** (e.g., fine-tuned) to a wide range of downstream tasks.



A milestone example is **CLIP** for Contrastive Language-Image Pre-training that aims at performing **zero-shot learning** :