A Selective Introduction to Deep Learning

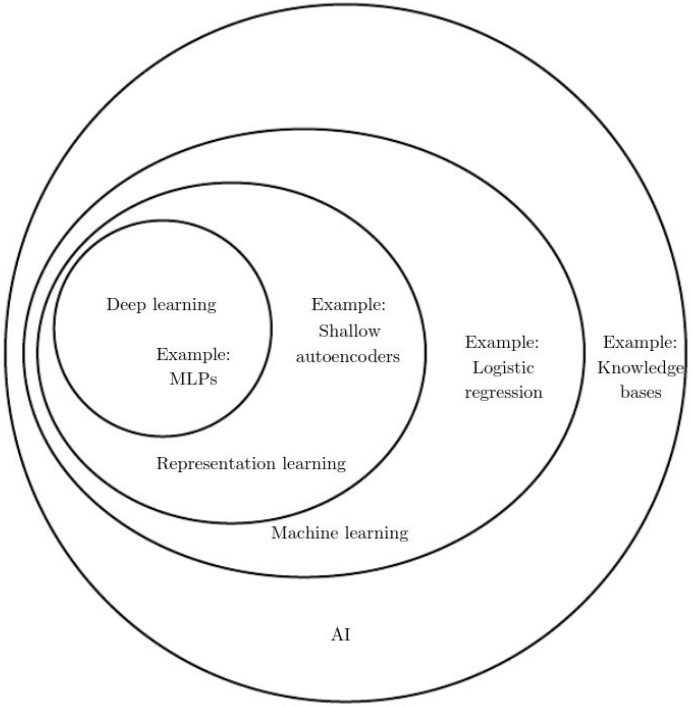
Yassine Ouali

Workshop IA DTY

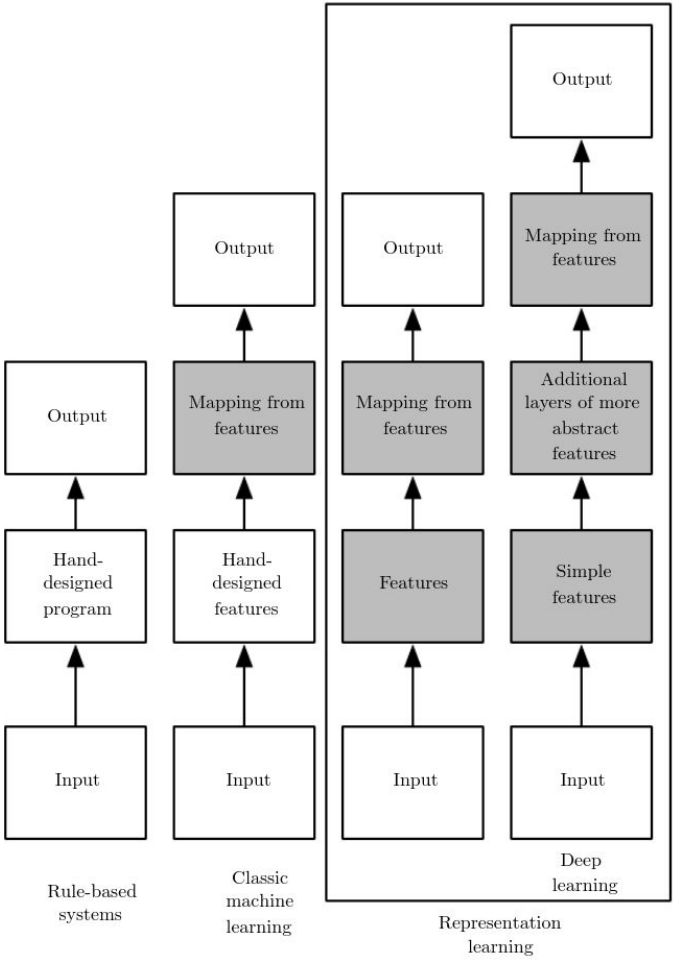
Deep Learning

The goal of Deep Learning is to:

- Allow computers to learn from experience (Algorithms - SGD).
- Understand the world in terms of a hierarchy of concepts, with each concept defined through its relation to simpler concepts (Networks - CNN).
- Gathering knowledge from experience (Data - ImageNet).



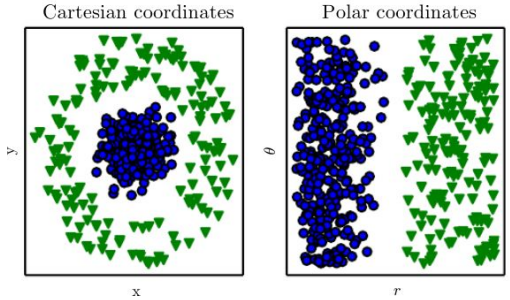
Classic AI and Deep Learning



Systems relying on hard-coded knowledge lack the ability to acquire their own knowledge.

-> Machine learning: enabled computers to tackle problems involving knowledge of the real world and make decisions that appear subjective.

- The performance of ML algorithms depends heavily on the representation of the data they are given.



-> Reuse ML, but this time not only the mapping from representation to output, but also the representation itself.

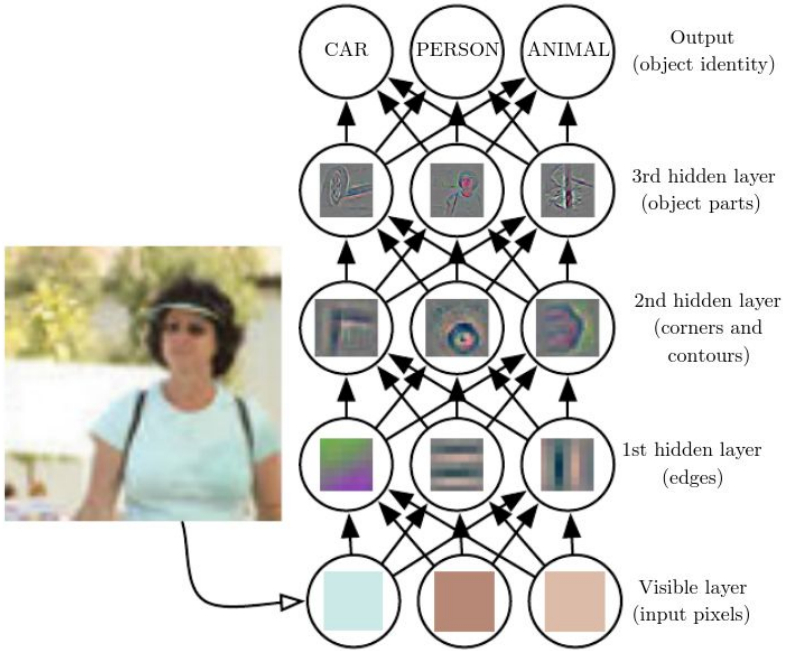
-> Learned representations often result in much better performance than can be obtained with hand-designed representations.

- The factors of variation influence every single piece of data we are able to observe. The individual pixels in an image of a red car might be very close to black at night.

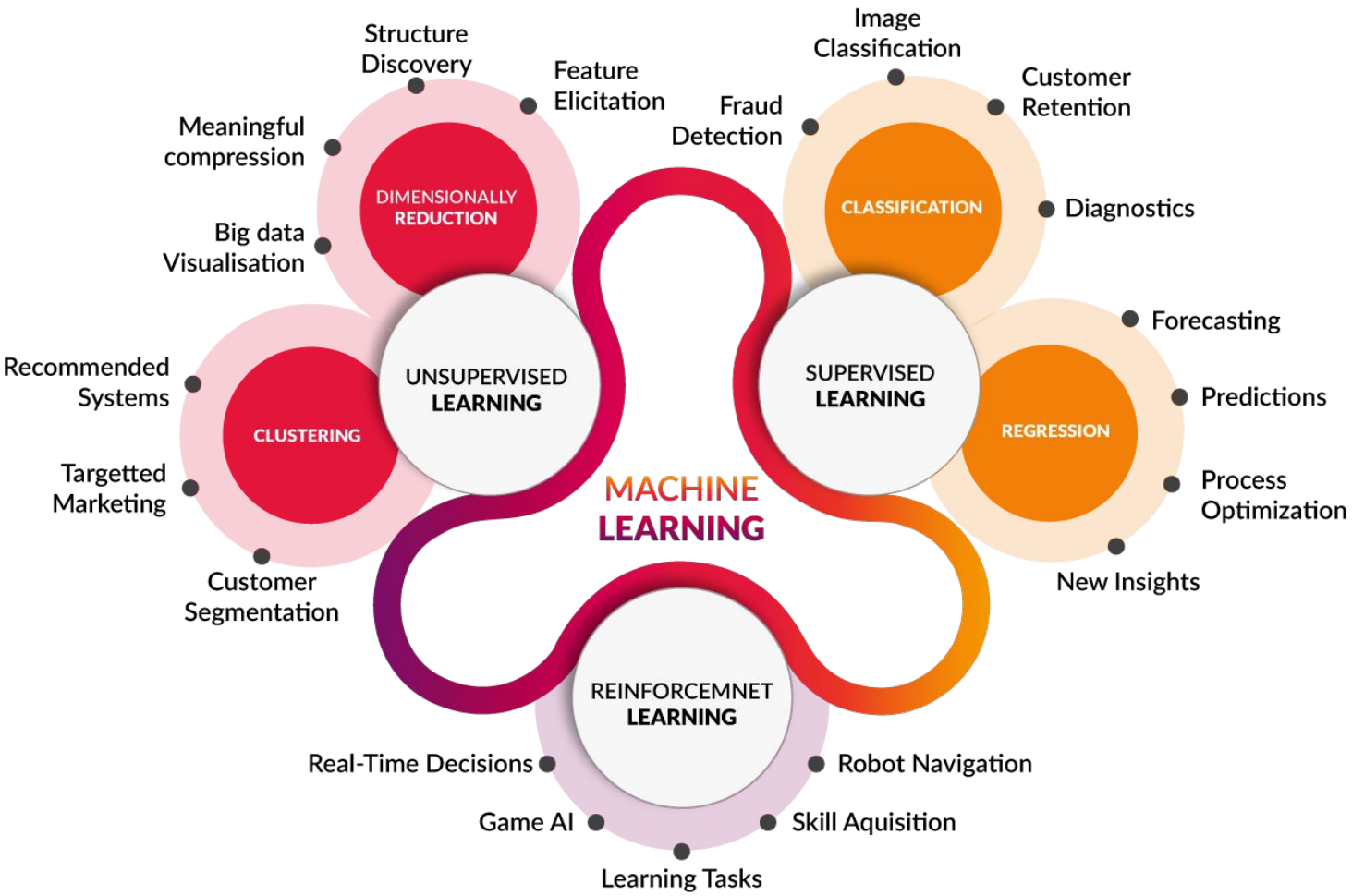
-> **Deep Learning:** disentangles the factors of variation and discard the ones that we do not care about

Deep Learning

Deep learning introduces representations that are expressed in terms of other, simpler representations.
Deep learning enables the computer to build complex concepts out of simpler concepts.



DL: in a nutshell is a composition of many nonlinear functions to model the complex dependency between input features and labels.


MACHINE LEARNING
UNSUPERVISED LEARNING
DIMENSIONALLY REDUCTION
Structure Discovery
Feature Elicitation
Meaningful compression
Big data Visualisation
CLUSTERING
Recommended Systems
Targetted Marketing
Customer Segmentation
SUPERVISED LEARNING
CLASSIFICATION
Image Classification
Customer Retention
Fraud Detection
Diagnostics
REGRESSION
Forecasting
Predictions
Process Optimization
New Insights
REINFORCEMNET LEARNING
Real-Time Decisions
Robot Navigation
Game AI
Skill Aquisition
Learning Tasks

Deep Learning

Modern machine learning deals with the problem of *learning from data*.

Given a training dataset $\{(y_i, \boldsymbol{x}_i)\}_{1 \leq i \leq n}$ Inputs: $\boldsymbol{x}_i \in \mathbb{R}^d$ Outputs: $y_i \in \mathbb{R}$

-> The objective is to learn a function $f : \mathbb{R}^d \mapsto \mathbb{R}$ from a certain function class $\mathcal{F}$ that has good prediction performance on test data.

Images Classification: **x** are raw RGB images and **y** are class labels.

Class of functions $\mathcal{F}$

- Linear classifiers (e.g., linear / logistic regression).
- Kernel methods (e.g., support vector machines).
- Tree-based methods (e.g., decision trees, random forests).
- Nonparametric regression (e.g.,nearest neighbors).

In *Deep learning*, we have a compositional function class:

$$\left\{ f(\boldsymbol{x}; \boldsymbol{\theta}) = \mathbf{W}_L \boldsymbol{\sigma}_L (\mathbf{W}_{L-1} \cdots \boldsymbol{\sigma}_2 (\mathbf{W}_2 \boldsymbol{\sigma}_1 (\mathbf{W}_1 \boldsymbol{x}))) \;\middle|\; \boldsymbol{\theta} = \{\mathbf{W}_1, \ldots, \mathbf{W}_L\} \right\}$$

Deep Learning Example: ImageNet Challenge, aka ILSVRC

A classification task: A dataset of 1.2 million color images with 1000 classes.
The performance of a classifier is then evaluated on a test dataset of 100 thousand images, and in the end the top-5 error 2 is reported.

Model	Year	# Layers	# Params	Top-5 error
Shallow	< 2012	—	—	$> 25\%$
AlexNet	2012	8	61M	16.4%
VGG19	2014	19	144M	7.3%
GoogleNet	2014	22	7M	6.7%
ResNet-152	2015	152	60M	3.6%

DL methods have a clear edge over shallow models that fit linear models / tree-based models on handcrafted features.

But there is a catch. Deep Learning in practice, requires:

- huge datasets that often contain millions of samples.
- immense computing power resulting (GPUs - TPUs).

Can be hard to get right due to:

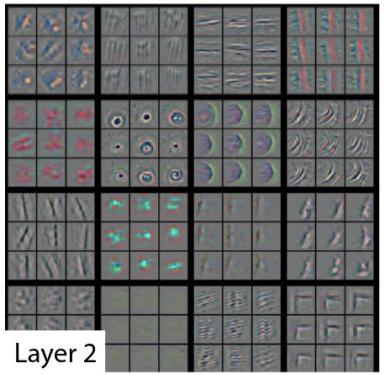
- *over-parameterization:* the number of parameters in state-of-the-art deep learning models is often much larger than the sample size (possible overfitting).
- *nonconvexity*: even with the help of GPUs, training deep learning models is still hard.

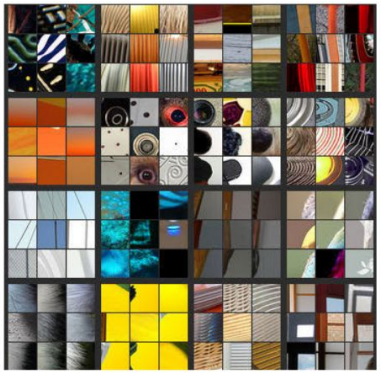
Deep in Deep Learning

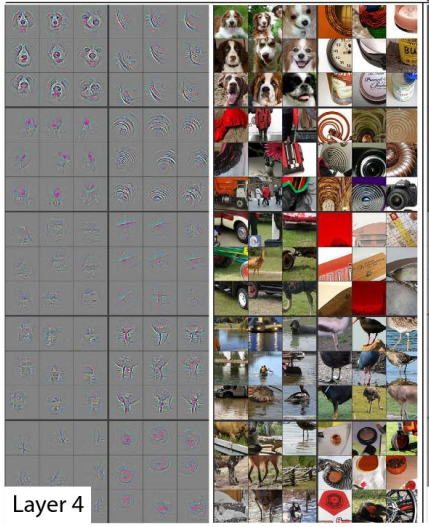
Deep learning expresses complicated nonlinearity through composing many nonlinear functions;
-> Composing lower-level features to create higher-level ones.
-> Represents the data in a different function space suitable for solving difficult tasks.

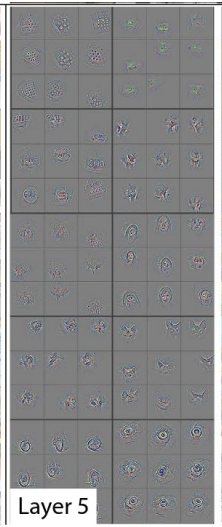
-> With many layers, we'll have a larger class function $\mathcal{F}$ (small approximation error).
-> A biggest generalization power, with deep net & SGD, we'll often have a small training error.

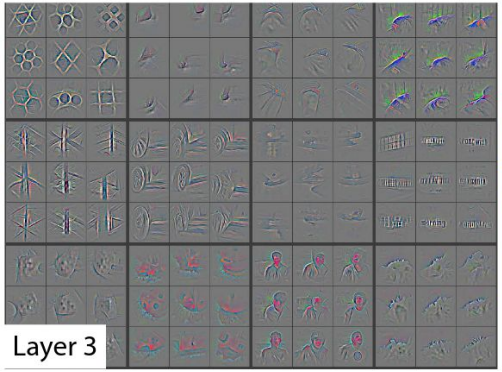

Layer 1

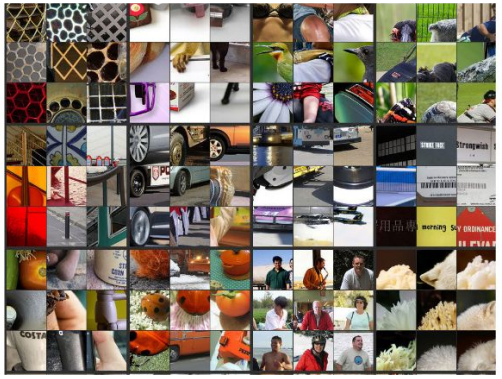

Layer 2

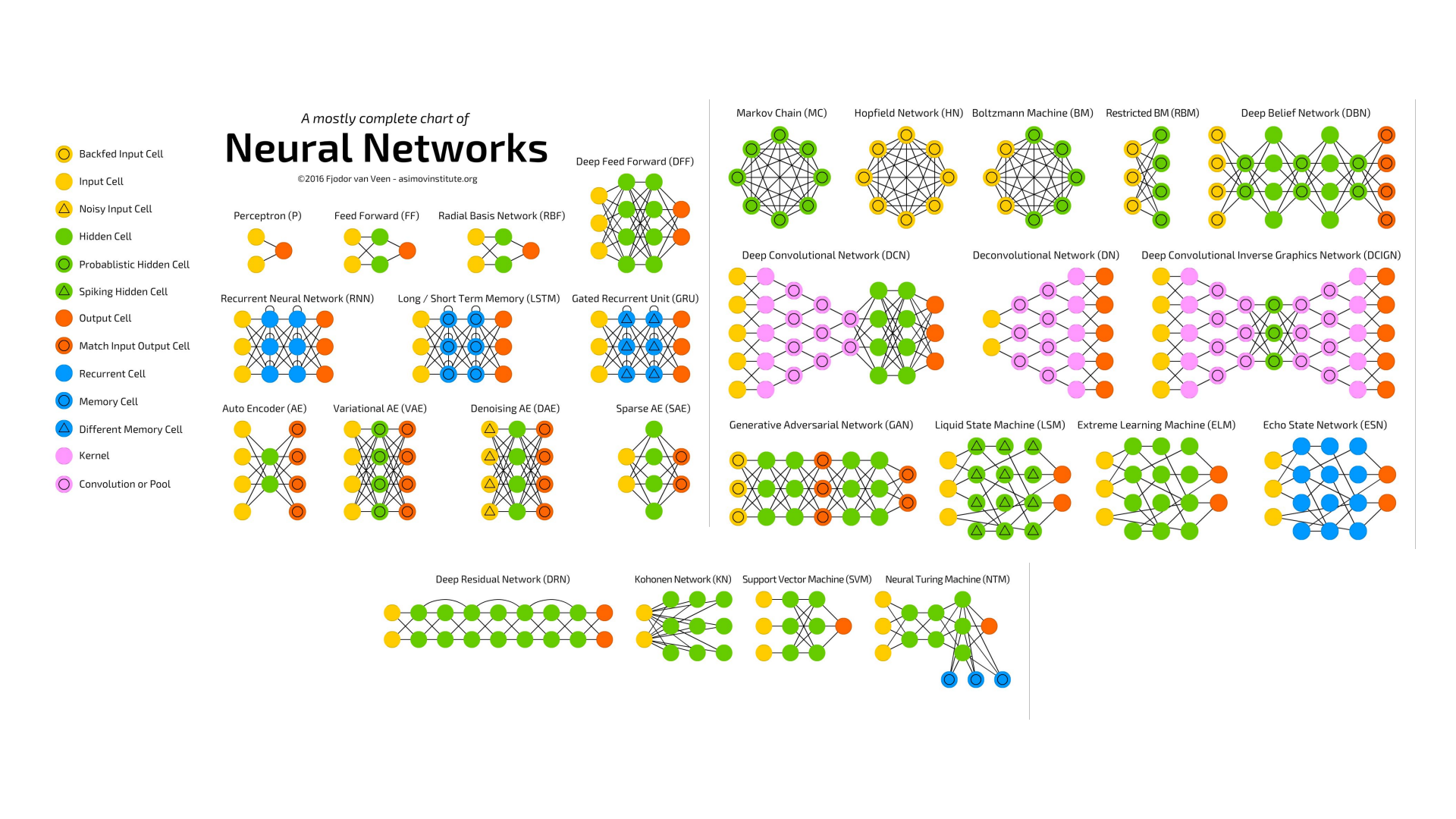



Layer 4


Layer 5


Layer 3




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
A mostly complete chart of
Neural Networks
©2016 Fjodor van Veen - asimovinstitute.org
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)

DL basics

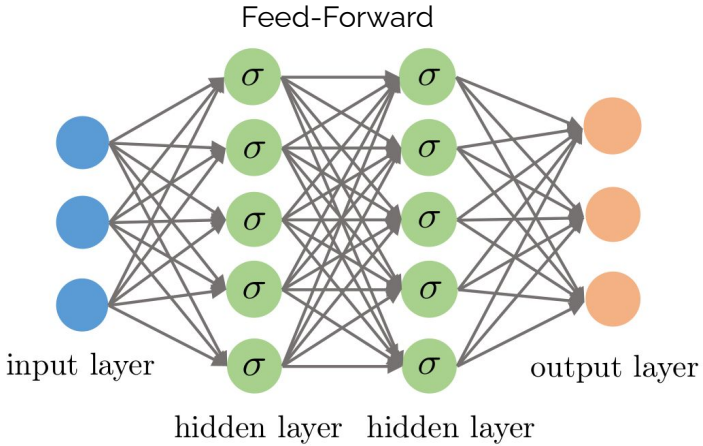
Notations: For a classification problem (regression can be addressed similarly):

The training dataset: $\{(y_i, \boldsymbol{x}_i)\}_{1<i<n}$ where $y_i \in [K] \triangleq \{1, 2, \ldots, K\}$ and $\boldsymbol{x}_i \in \mathbb{R}^d$ (We assume data points are IID)

Goal: find a function $\hat{f}(\boldsymbol{x})$ that predicts the labels ***y*** for a new inputs ***x*** (for the same distribution).

Terminology:

- inputs xi : features;
- outputs yi: labels;
- (xi, yi) example
- $\hat{f}$ Classifier



Deep nets use composition of a series of simple nonlinear functions to model nonlinearity:

$$\boldsymbol{h}^{(L)} = \mathbf{g}^{(L)} \circ \mathbf{g}^{(L-1)} \circ \ldots \circ \mathbf{g}^{(1)}(\boldsymbol{x})$$

Recursively: $\boldsymbol{h}^{(0)} \triangleq \boldsymbol{x} \qquad \boldsymbol{h}^{(l)} = \mathbf{g}^{(l)}(\boldsymbol{h}^{(l-1)})$

Feed-forward nets, aka; *multilayer perceptrons (MLPs)*

$$\boldsymbol{h}^{(\ell)} = \mathbf{g}^{(l)}(\boldsymbol{h}^{(l-1)}) \triangleq \boldsymbol{\sigma}(\mathbf{W}^{(\ell)}\boldsymbol{h}^{(\ell-1)} + \boldsymbol{b}^{(\ell)})$$

Examples of non-linearity:

- ReLU $[\boldsymbol{\sigma}(\boldsymbol{z})]_j = \max\{z_j, 0\}$
- Sigmoid $(1 + e^{-z})^{-1}$

The output:

$$f_k(\boldsymbol{x}; \boldsymbol{\theta}) \triangleq \frac{\exp(z_k)}{\sum_k \exp(z_k)}, \quad \forall\, k \in [K]$$

$$\boldsymbol{z} = \mathbf{W}^{(L+1)}\boldsymbol{h}^{(L)} + \boldsymbol{b}^{(L+1)} \in \mathbb{R}^K$$

DL basics

With the output probabilities $f_k(\boldsymbol{x};\boldsymbol{\theta}) \triangleq \frac{\exp(z_k)}{\sum_k \exp(z_k)}, \quad \forall\, k \in [K]$, we compute the loss (cross-entropy)

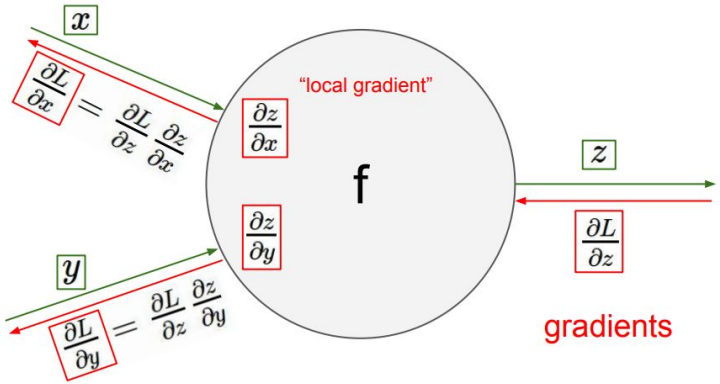
$$\mathcal{L}(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = -\sum_{k=1}^{K} \mathbb{1}\{y = k\} \log p_k, \quad \boldsymbol{\theta} \triangleq \{\mathbf{W}^{(\ell)}, \boldsymbol{b}^{(\ell)} : 1 \le \ell \le L+1\}.$$

Backpropagation:

Using *stochastic gradient descent,* start from random weights $\boldsymbol{\theta}^0$ then iteratively updates the parameters $\boldsymbol{\theta}^t$ we move in the direction of the negative gradient to reduce the loss.

- Why *stochastic?* each iteration is based only on a subsample of the data. A batch $\mathcal{B} \subset [n]$
  -> Reduces computation, converges to the solutions that can be obtained with GD.

$$\ell_{\mathcal{B}}(\boldsymbol{\theta}) \triangleq |\mathcal{B}|^{-1} \sum_{i \in \mathcal{B}} \mathcal{L}(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$$



Back-prop in computational graphs

$$\frac{\partial \ell_{\mathcal{B}}}{\partial \boldsymbol{h}^{(\ell-1)}} = \frac{\partial \boldsymbol{h}^{(\ell)}}{\partial \boldsymbol{h}^{(\ell-1)}} \cdot \frac{\partial \ell_{\mathcal{B}}}{\partial \boldsymbol{h}^{(\ell)}} ;$$

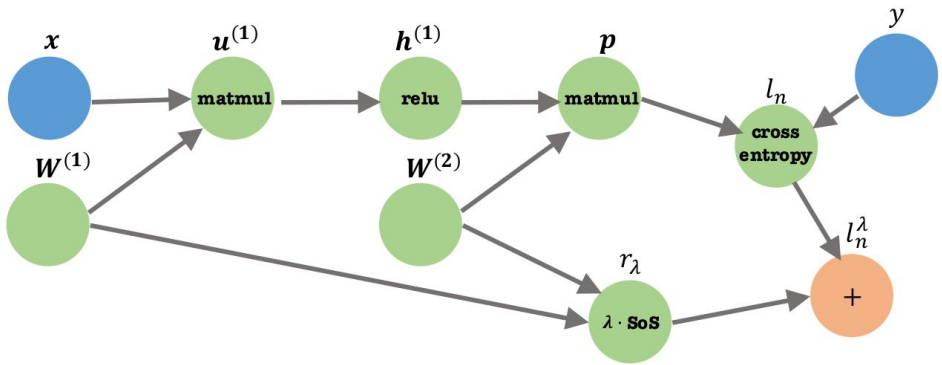
$$\mathbf{W}^{(\ell)} \leftarrow \mathbf{W}^{(\ell)} - \eta \frac{\partial \ell_{\mathcal{B}}}{\partial \mathbf{W}^{(\ell)}}, \quad \text{where} \quad \frac{\partial \ell_{\mathcal{B}}}{\partial W_{jm}^{(\ell)}} = \frac{\partial \ell_{\mathcal{B}}}{\partial h_j^{(\ell)}} \cdot \sigma' \cdot h_m^{(\ell-1)},$$

DL basics

Back-prop in computational graphs: With large models, we can quickly overfit (memorize) the data, without any generalization to the test set -> Penalize large variation in the weights

$$\ell_{\mathcal{B}}^{\lambda}(\boldsymbol{\theta}) = \ell_{\mathcal{B}}(\boldsymbol{\theta}) + r_{\lambda}(\boldsymbol{\theta}) = \ell_{\mathcal{B}}(\boldsymbol{\theta}) + \lambda\Big(\sum_{j,j'}\big(W_{j,j'}^{(1)}\big)^2 + \sum_{j,j'}\big(W_{j,j'}^{(2)}\big)^2\Big)$$

In practice (PyTorch, TensofFlow), the model is represented as a computation graph: Each node represents a function (inside a circle), which is associated with an output of that function (outside a circle)



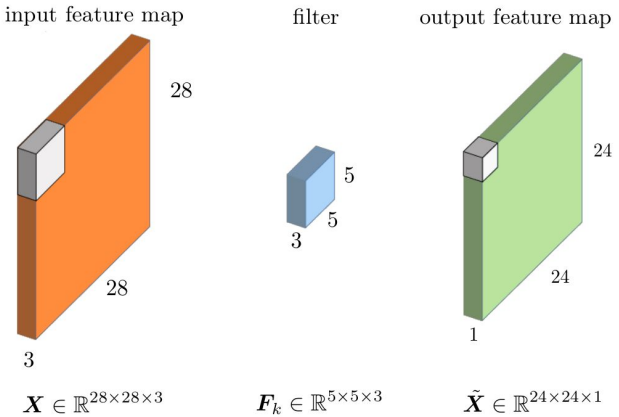
DL Models: Convolutional neural networks

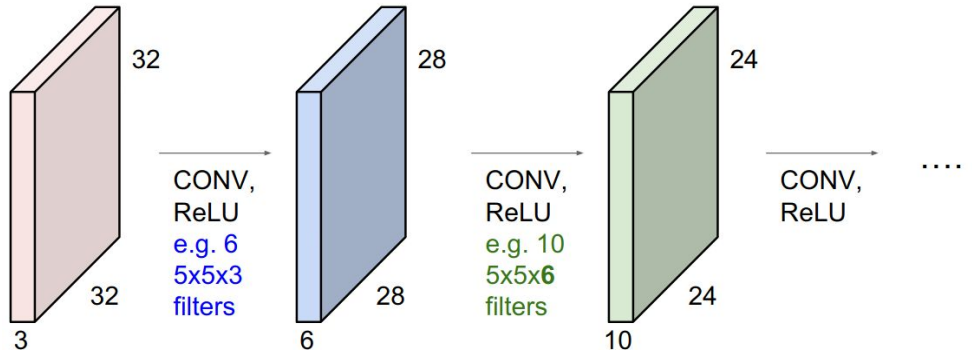
A special type of feed-forward neural networks that is tailored for image processing.

Inputs: 3D tensors; $\boldsymbol{X} \in \mathbb{R}^{d_1 \times d_2 \times d_3}$

Convolutional layer

$$O_{ij}^k = \left\langle [\boldsymbol{X}]_{ij}, \boldsymbol{F}_k \right\rangle = \sum_{i'=1}^{w} \sum_{j'=1}^{w} \sum_{l=1}^{d_3} [\boldsymbol{X}]_{i+i'-1,j+j'-1,l} [\boldsymbol{F}_k]_{i',j',l}.$$





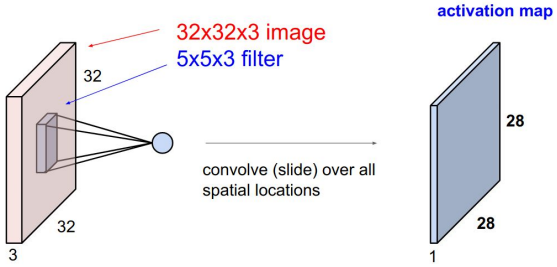
Nonlinearity

$$\tilde{X}_{ijk} = \sigma(O_{ijk}), \qquad \forall\, i \in [d_1 - w + 1], j \in [d_2 - w + 1], k \in [\tilde{d}_3].$$

$$[\boldsymbol{\sigma}(\boldsymbol{z})]_j = \max\{z_j, 0\}$$

DL Models: Convolutional neural networks

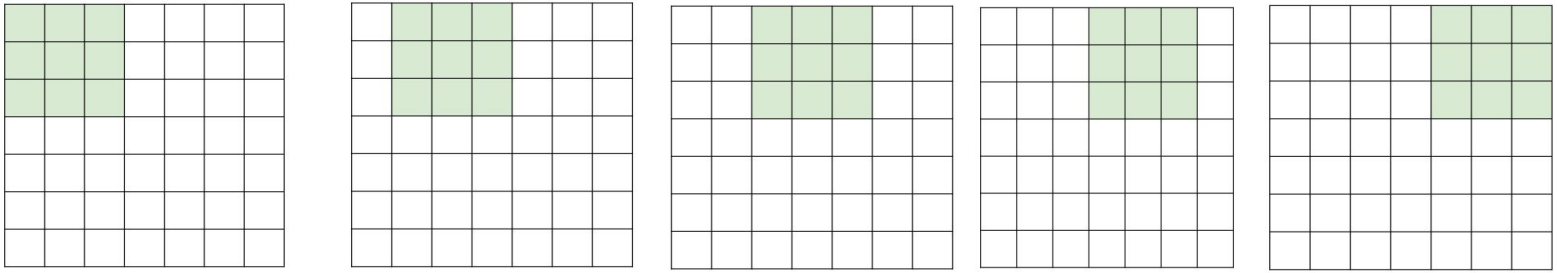
Convolutional layer: A closer look



7x7 input

3 x 3 kernel
stride 1

5x5 output



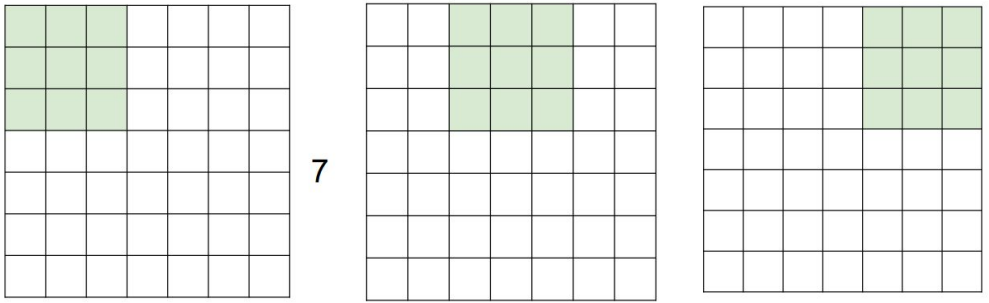
7

7x7 input

3 x 3 kernel
stride 2

3x3 output

7



Output size:
(N - F) / stride + 1

e.g. N = 7, F = 3:
stride 1 => (7 - 3)/1 + 1 = 5
stride 2 => (7 - 3)/2 + 1 = 3
stride 3 => (7 - 3)/3 + 1 = 2.33 :\

DL Models: Convolutional neural networks

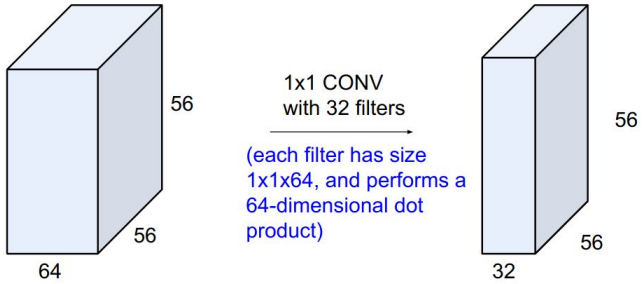
Convolutional layer: Keeping the same spatial dimensions

7x7 input

3 x 3 kernel | stride 1 | padding 1

7x7 output

0	0	0	0	0	0			
0								
0								
0								
0								

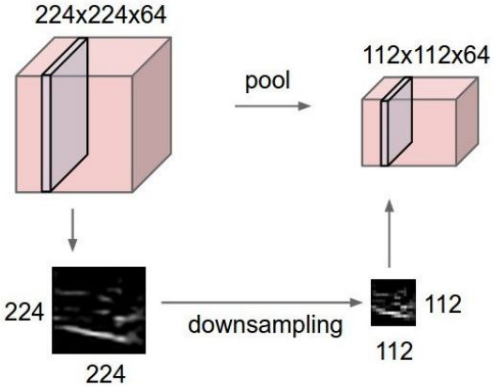


- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

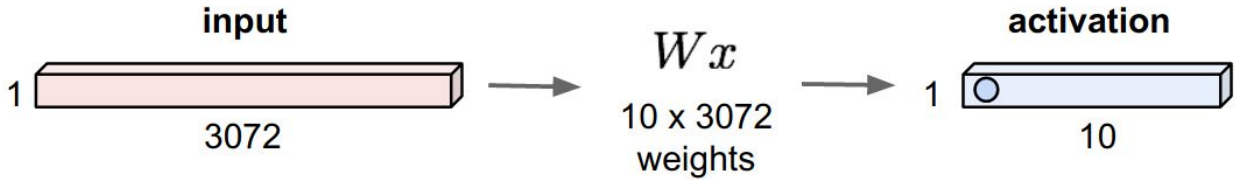
DL Models: Convolutional neural networks

3- Pooling layers: aggregates the information of nearby features into a single one. A 2x2 max-pooling filter computes

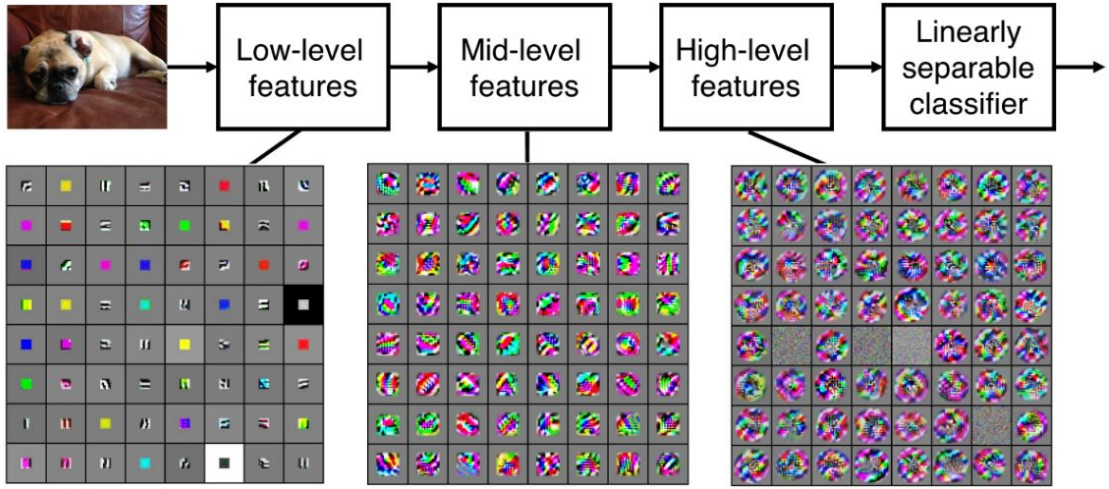
$$\max\{X_{i,j,k}, X_{i+1,j,k}, X_{i,j+1,k}, X_{i+1,j+1,k}\}$$



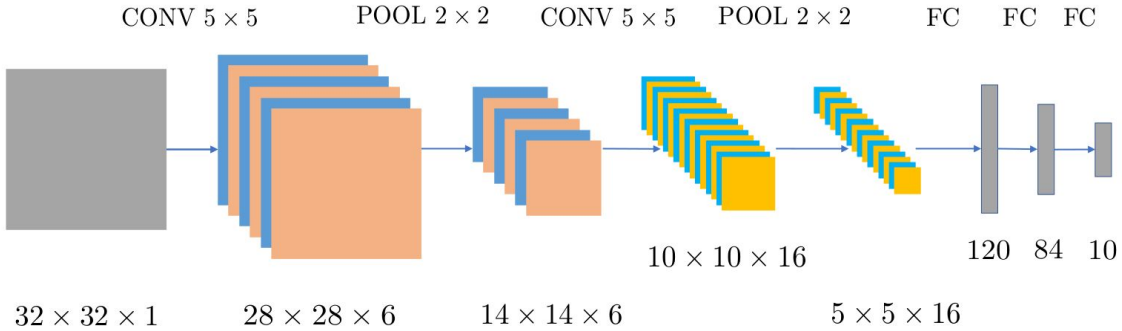
4- Fully connected layers: Treating 3D tensors as 2D vectors to compute the output probabilities $\tilde{\boldsymbol{X}} = \boldsymbol{\sigma}(\mathbf{W}\mathrm{Vec}(\boldsymbol{X}))$



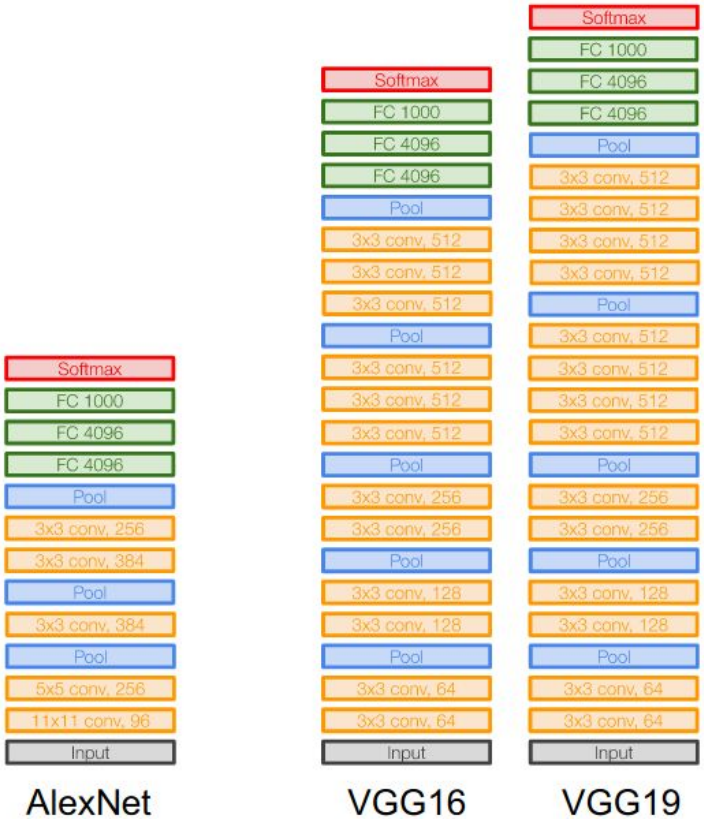
CNN architectures

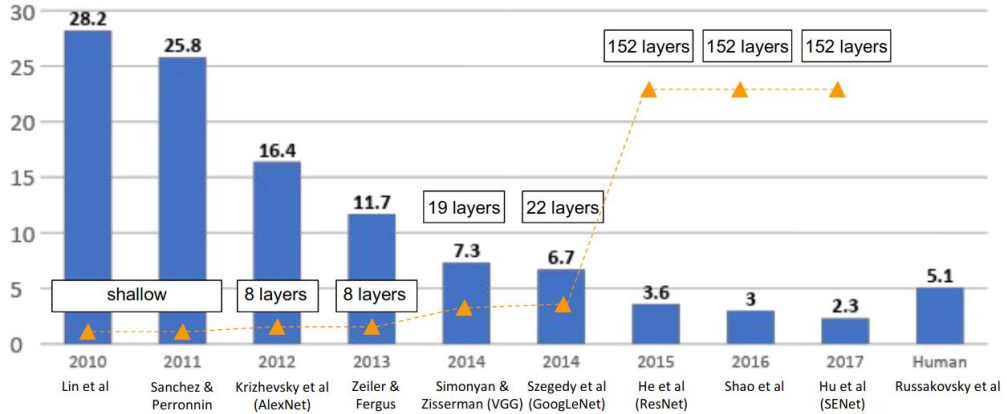


LeNet



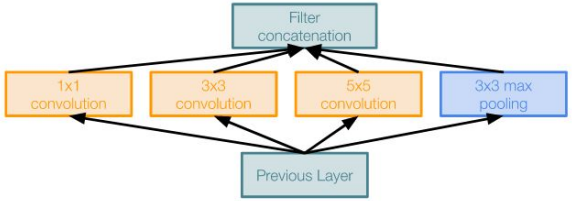
CNN architectures



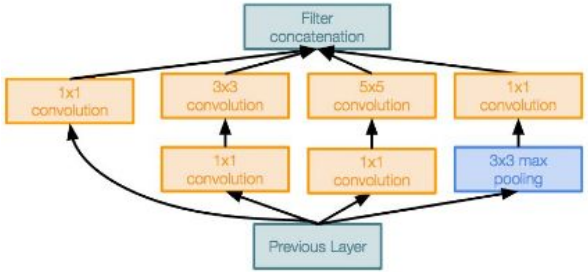


CNN architectures: GoogleNet

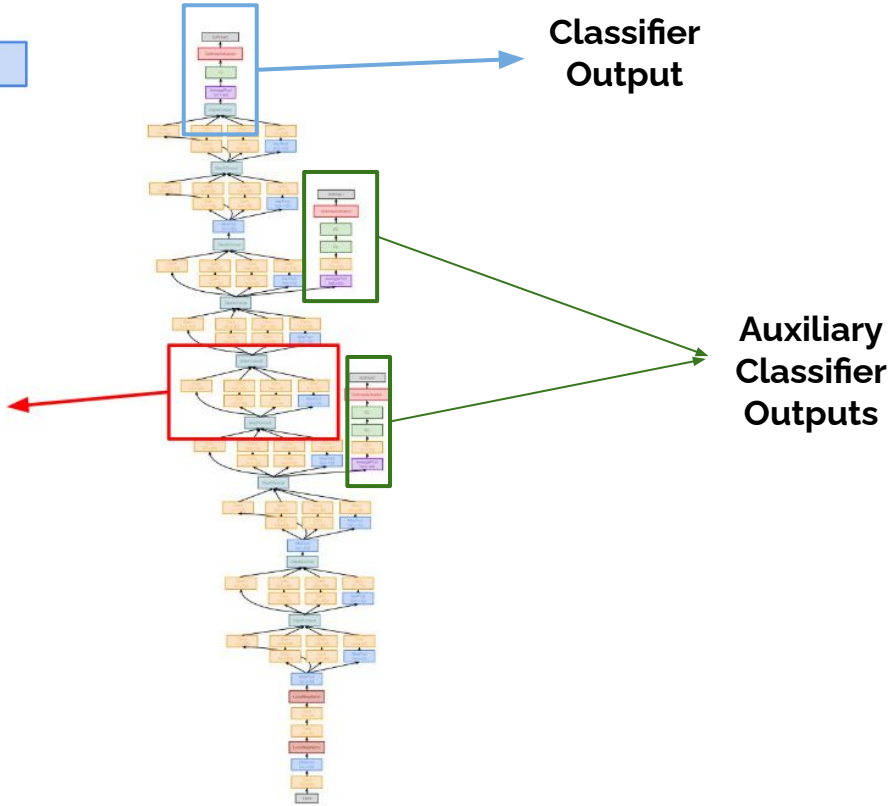
"Inception module": design a good local network topology (network within a network) and then stack these modules on top of each other



Naive Inception module



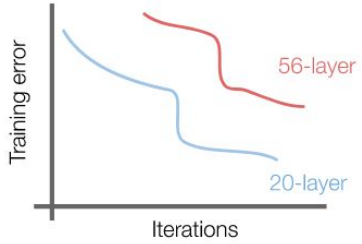
Inception module

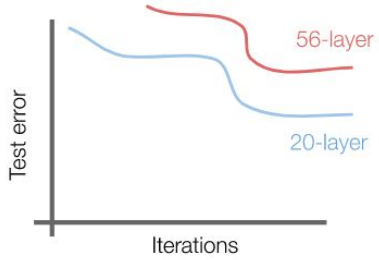


CNN architectures: ResNet

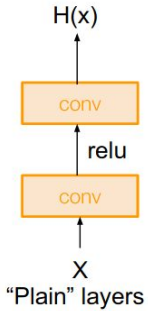
Very deep networks with residual connections: What happens when we continue stacking deeper layers on a “plain” convolutional neural network?

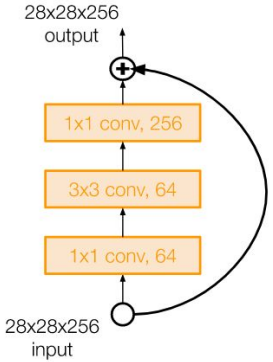
- the problem is an optimization problem, deeper models are harder to optimize

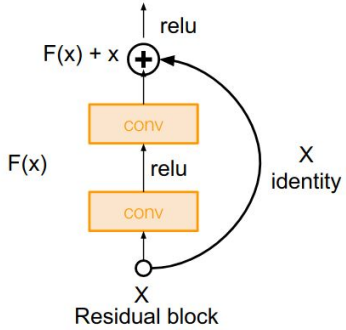


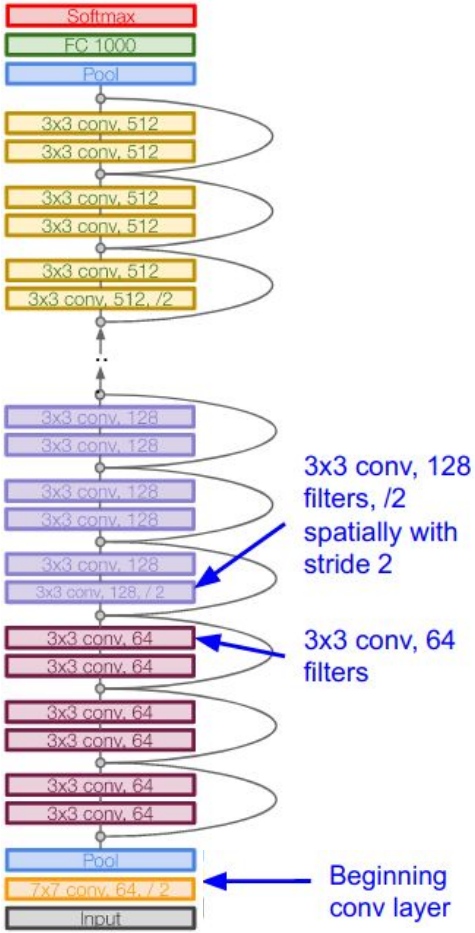


+ ACopying the learned layers from the shallower model and setting additional layers to identity mapping.



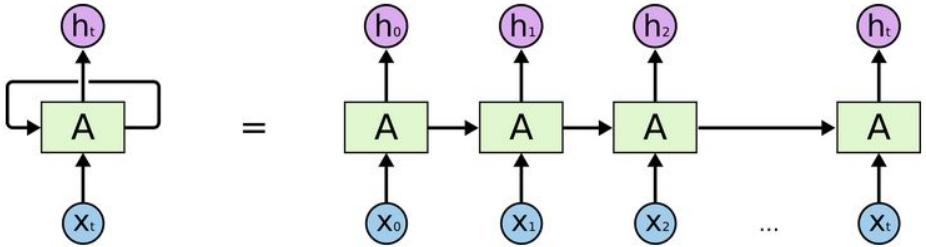




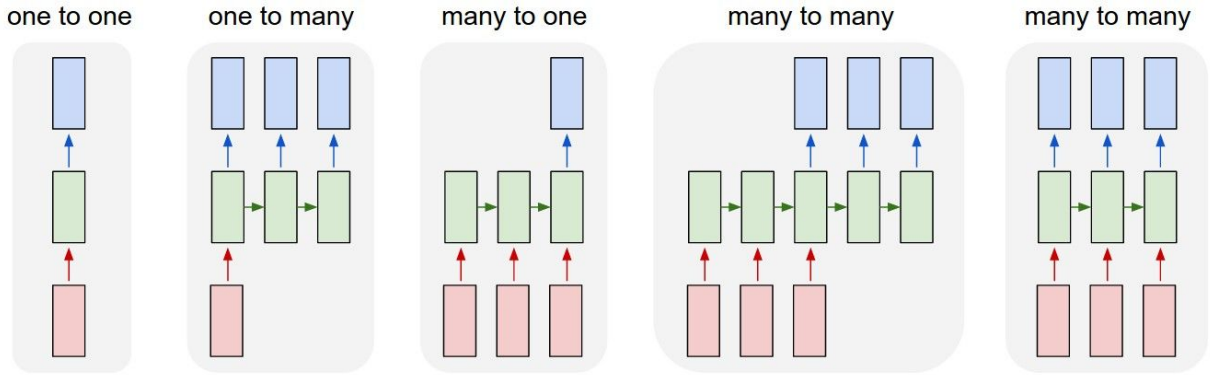


DL Models: Recurrent neural networks

Recurrent nets:
A way to process sequences



Different use cases

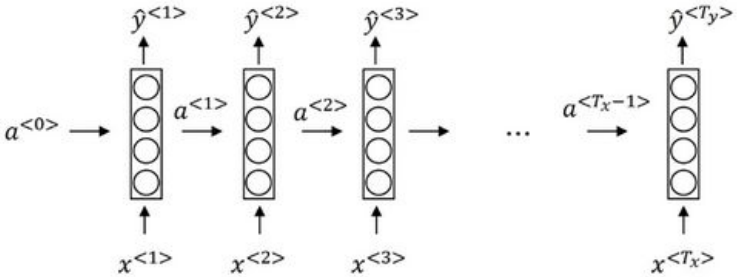


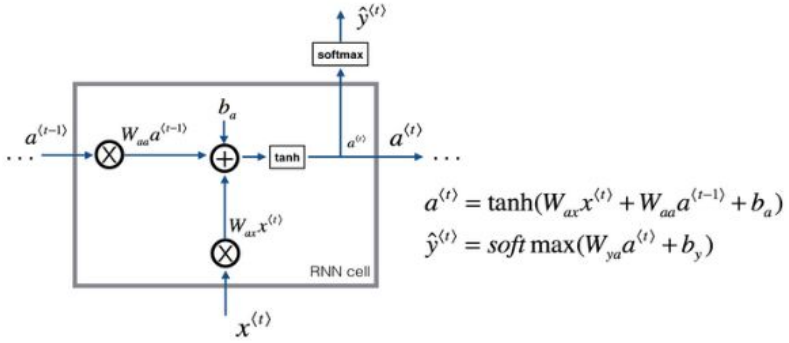
(1) Normal feedforward nets (e.g. image classification).
(2) Sequence output (e.g. image captioning - music generation).
(3) Séquence input (e.g. sentiment analysis).
(4) Sequence input & output (e.g. machine translation).
(5) Synced sequence input and output (e.g. video classification).

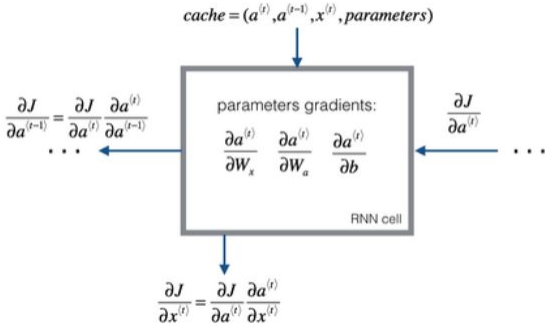
Can be also used with inputs/outputs are fixed vectors

RNNs ~ a repetition of a single rnn cell

Machine translation	Voulez-vous chanter avec moi?	→	Do you want to sing with me?
Video activity recognition		→	Running
Name entity recognition	Yesterday, Harry Potter met Hermione Granger.	→	Yesterday, Harry Potter met Hermione Granger.

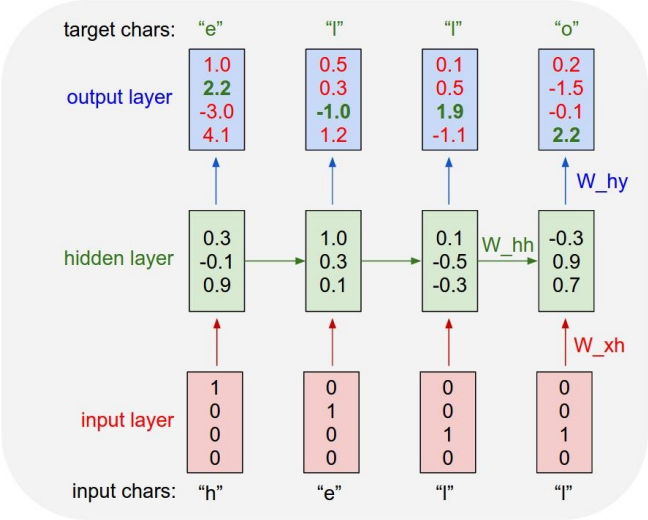


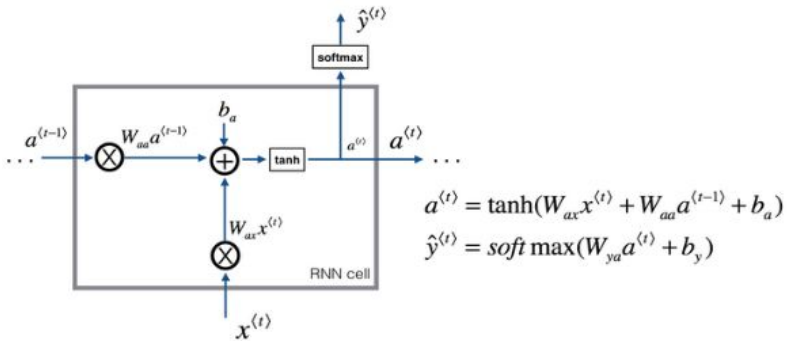




- Propagate the input through the network to get the output.
- Calculate the error using the labels.
- Calculate the derivatives of the error with respect to the network weights.
- Adjust to minimise the error

Use case: Character-Level RNN





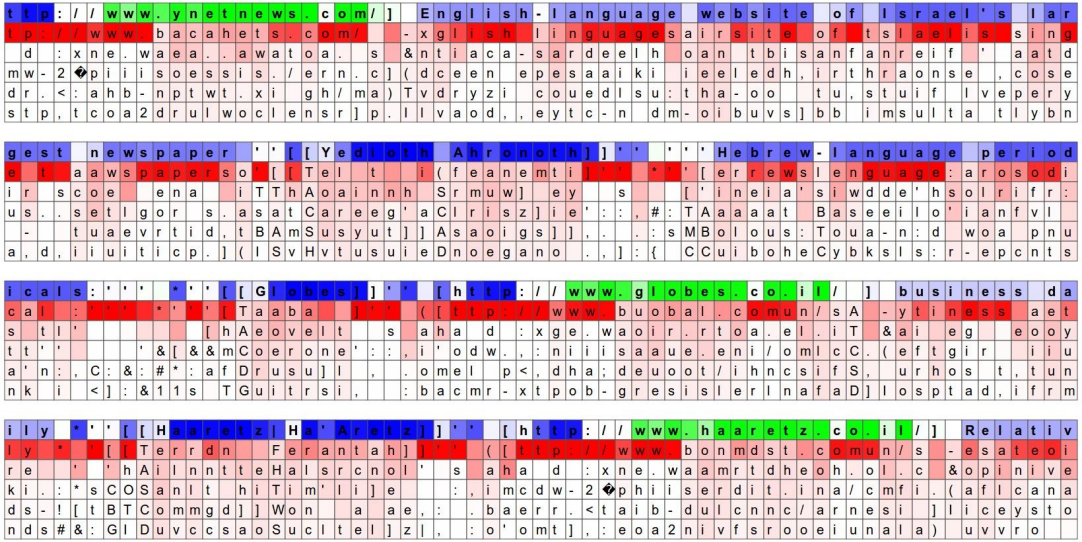
$$a^{\langle t \rangle} = \tanh(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b_a)$$

$$\hat{y}^{\langle t \rangle} = soft\max(W_{ya}a^{\langle t \rangle} + b_y)$$

Pytorch example

```
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()

        self.hidden_size = hidden_size

        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2o = nn.Linear(input_size + hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = self.i2h(combined)
        output = self.i2o(combined)
        output = self.softmax(output)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, self.hidden_size)

n_hidden = 128
rnn = RNN(n_letters, n_hidden, n_categories)
```

Visualizing the predictions



Cell sensitive to position in line:

```
The sole importance of the crossing of the Berezina lies in the fact
that it plainly and indubitably proved the fallacy of all the plans for
cutting off the enemy's retreat and the soundness of the only possible
line of action--the one Kutuzov and the general mass of the army
demanded--namely, simply to follow the enemy up. The French crowd fled
at a continually increasing speed and all its energy was directed to
reaching its goal. It fled like a wounded animal and it was impossible
to block its path. This was shown not so much by the arrangements it
made for crossing as by what took place at the bridges. When the bridges
broke down, unarmed soldiers, people from Moscow and women with children
who were with the French transport, all--carried on by vis inertiae--
pressed forward into boats and into the ice-covered water and did not,
surrender.
```

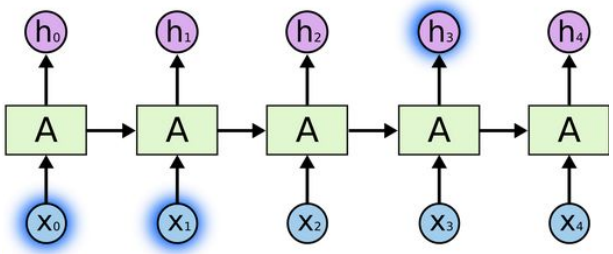
Cell that turns on inside quotes:

```
"You mean to imply that I have nothing to eat out of.... On the
contrary, I can supply you with everything even if you want to give
dinner parties," warmly replied Chichagov, who tried by every word he
spoke to prove his own rectitude and therefore imagined Kutuzov to be
animated by the same desire.

Kutuzov, shrugging his shoulders, replied with his subtle penetrating
smile: "I meant merely to say what I said."
```

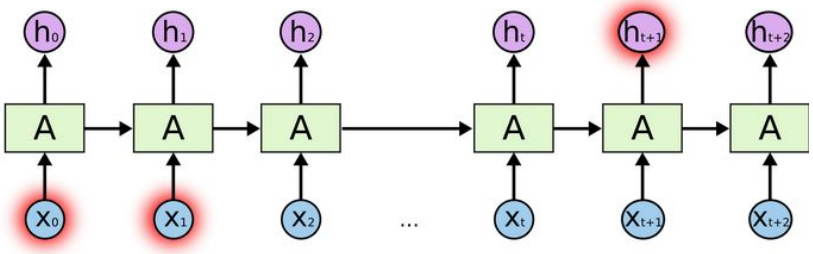
A large portion of cells are not easily interpretable. Here is a typical example:

```
/* Unpack a filter field's string representation from user-space
 * buffer. */
char *audit_unpack_string(void **bufp, size_t *remain, size_t len)
{
 char *str;
 if (!*bufp || (len == 0) || (len > *remain))
  return ERR_PTR(-EINVAL);
 /* Of the currently implemented string fields, PATH_MAX
  * defines the longest valid length.
  */
```

The Problem of Long-Term Dependencies

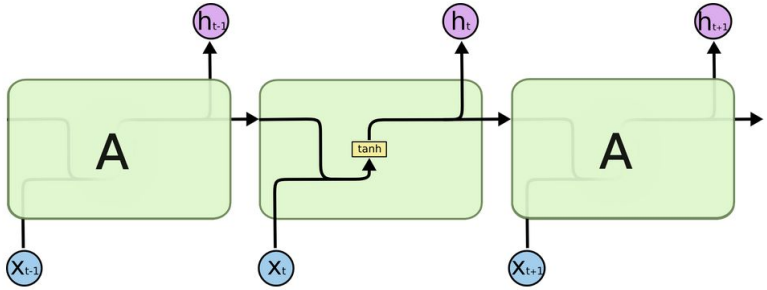


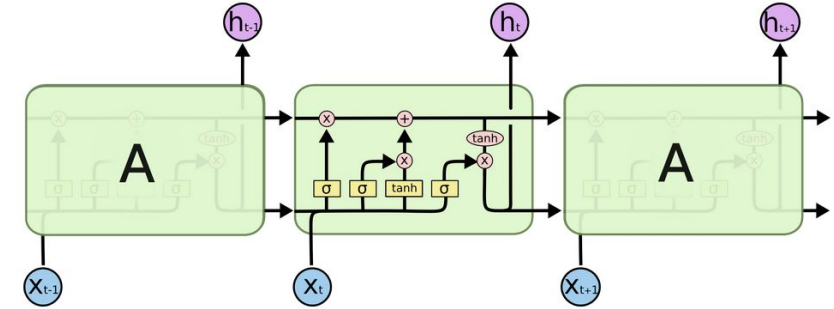
The **clouds** are in the **sky**



I grew up in **France**… I speak fluent **French**

LSTM to the rescue

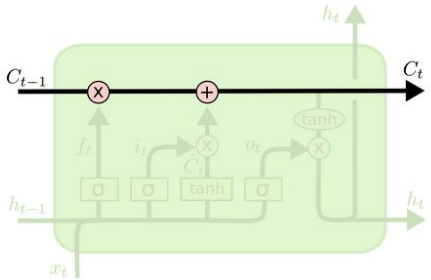




https://imgur.com/gallery/vaNahKE

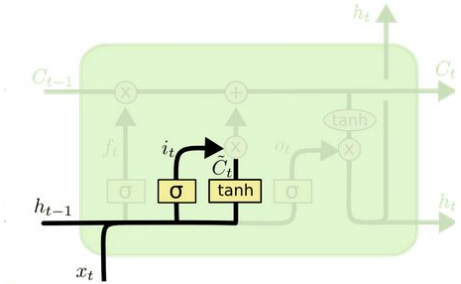
http://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTMs Step-by-Step



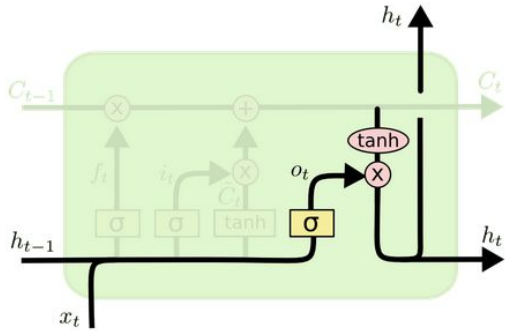
The ability to remove or add information to the main cell state

2)



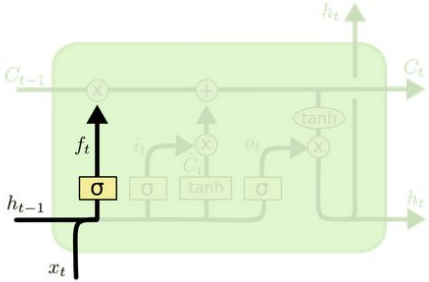
Update gate : 1st what to update, and then compute the new candidate values

4)



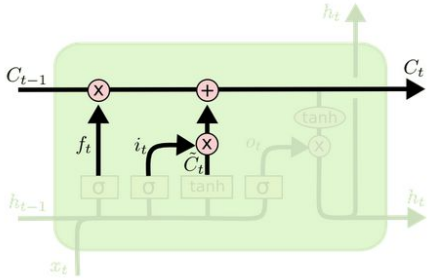
Output gate: From the new state of the main cell, output the relevant informations

1)



Forget gate : what information we're going to forget

3)



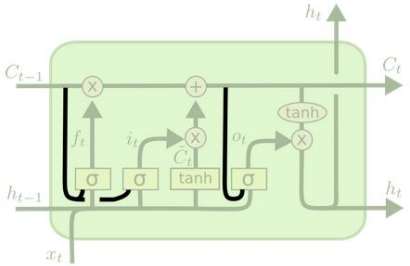
Forget & Update

the *cat/cats* with .. ate ..... *was/were*

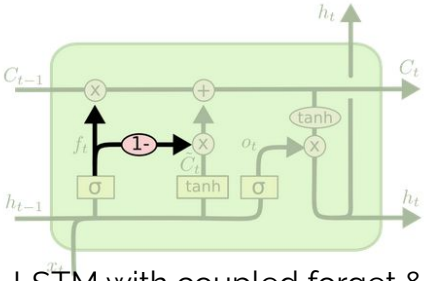
http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Recurrent nets

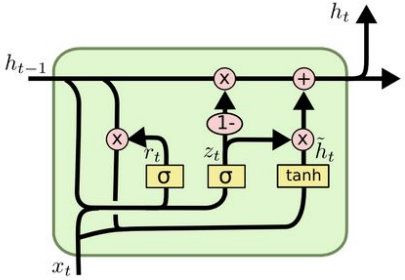
LSTMs variants



Peephole LSTM

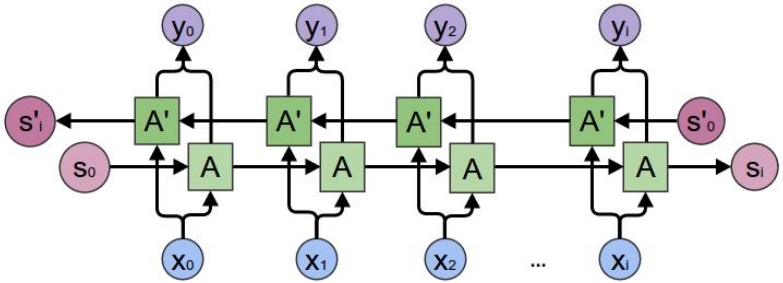


LSTM with coupled forget & update

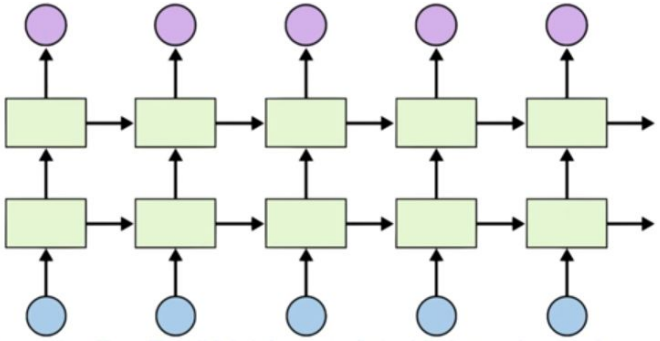


GRU: Gated recurrent unit

Bi-LSTMs



Stacked layers of LSTM cells

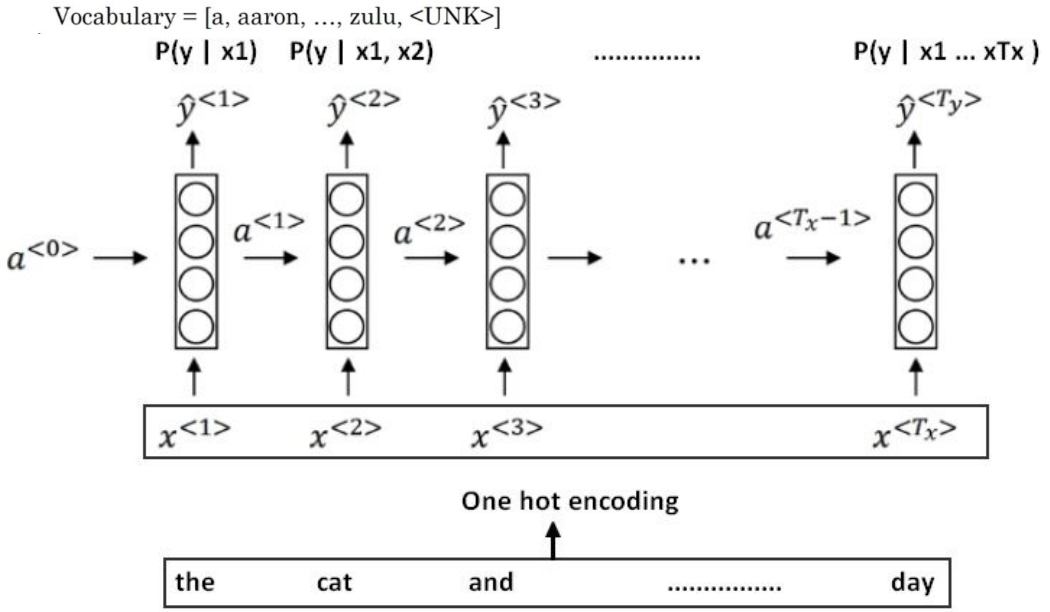


RNNs for Language Modeling

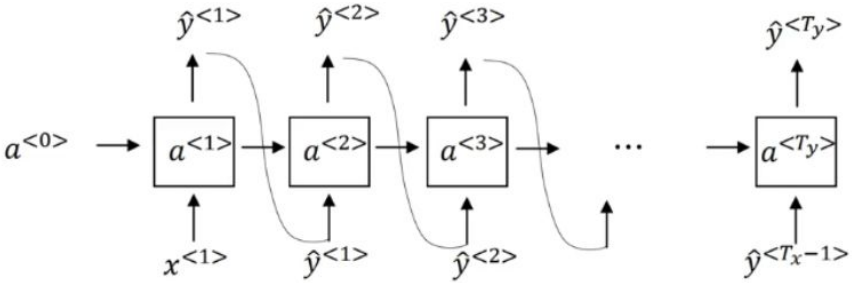
- Preprocess the inputs.
- One hot encoding.
- Batch the dataset.

For each example / batch:

- Calculate the CE loss at each time step.
- Back propagate the loss.



Use the trained RNN for language generation



Unsupervised learning

In supervised learning, we have a labeled dataset $\{(y_i, \mathbf{x}_i)\}$ and we focus on discriminative model, representing $\mathbb{P}(y \mid \boldsymbol{x})$ by a neural network $f(\boldsymbol{x}; \boldsymbol{\theta})$ with parameters $\boldsymbol{\theta}$.

Supervised Learning

Data: (x, y)
x is data, y is label

Goal: Learn a *function* to map x -> y

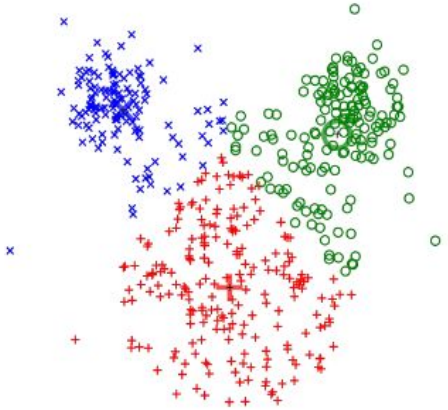
Examples: Classification, regression, object detection, semantic segmentation, image captioning, etc.

Unsupervised Learning

Data: x
Just data, no labels!

Goal: Learn some underlying hidden *structure* of the data

Examples: Clustering, dimensionality reduction, feature learning, density estimation, etc.

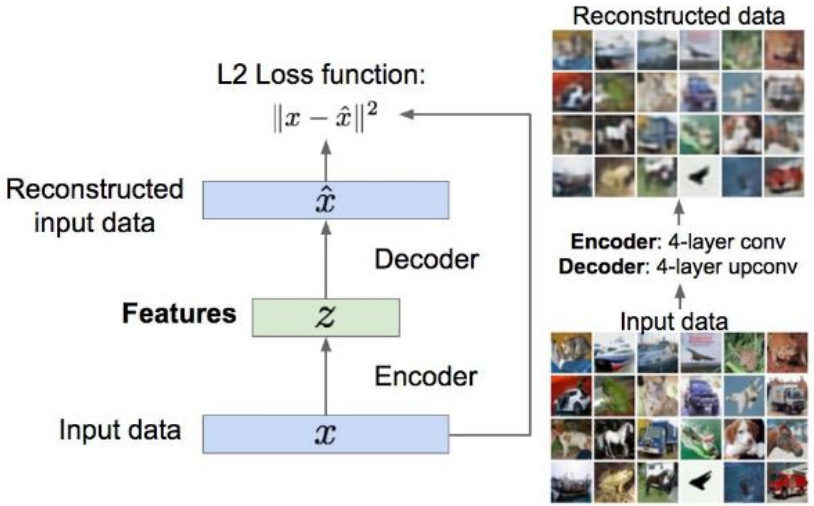


K-means clustering

Deep unsupervised learning

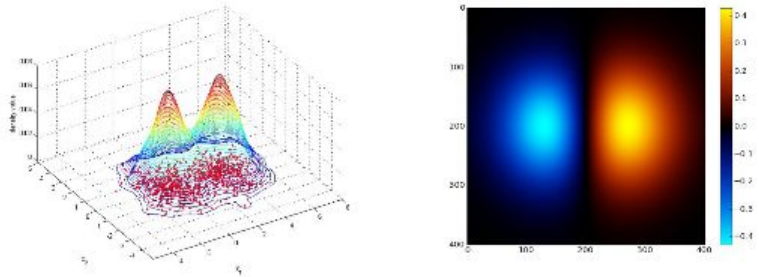
In unsupervised learning: We are interested in extracting information from the unlabeled data ***xi*** (without using labels ***yi***).

- Can be low-dimensional embeddings of the data. -> **Autoencoders**
- A generative model capable of approximating the data distribution $\mathbb{P}_{\mathbf{X}}(\boldsymbol{x})$. -> **GaNs**



Autoencoders
(Feature learning)

1-d density estimation



2-d density estimation

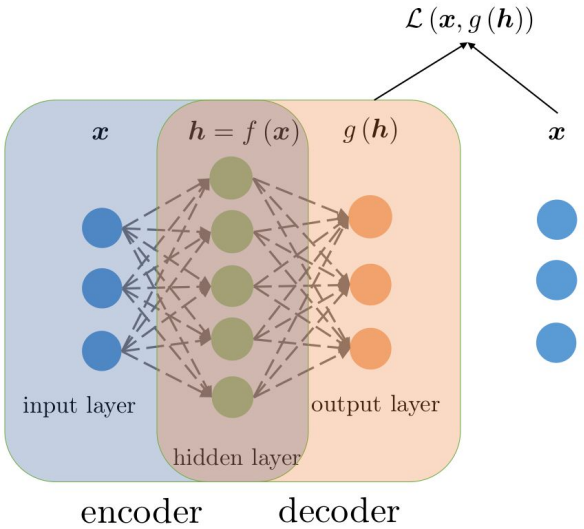
Autoencoders

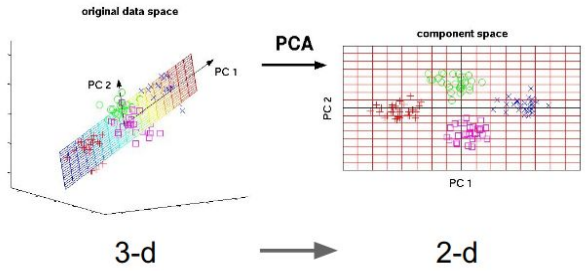
in dimension reduction, the goal is to reduce the dimensionality of the data and at the same time preserve its salient features:

- PCA: embedding the data **xi** into a low-dimensional space using a linear function **f** such that maximum variance can be explained (PCA is a special case of autoencoders).
- Equivalently; with autoencoders with want to learn two functions **f** and **g** so that the difference between **g(f(xi))** and **xi** is minimal.

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{W}_f \boldsymbol{x} \triangleq \boldsymbol{h} \quad \text{and} \quad \boldsymbol{g}(\boldsymbol{h}) = \boldsymbol{W}_g \boldsymbol{h}, \quad \text{where} \quad \boldsymbol{W}_f \in \mathbb{R}^{k \times d} \text{ and } \boldsymbol{W}_g \in \mathbb{R}^{d \times k}.$$

$$\text{minimize}_{\boldsymbol{W}_f, \boldsymbol{W}_g} \quad \frac{1}{n} \sum_{i=1}^{n} \|\boldsymbol{x}_i - \boldsymbol{W}_f \boldsymbol{W}_g \boldsymbol{x}_i\|_2^2. \qquad \text{minimize}_{\boldsymbol{f}, \boldsymbol{g}} \quad \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(\boldsymbol{x}_i, \boldsymbol{g}(\boldsymbol{h}_i)) \quad \text{with} \quad \boldsymbol{h}_i = \boldsymbol{f}(\boldsymbol{x}_i), \quad \text{for all } i \in [n].$$





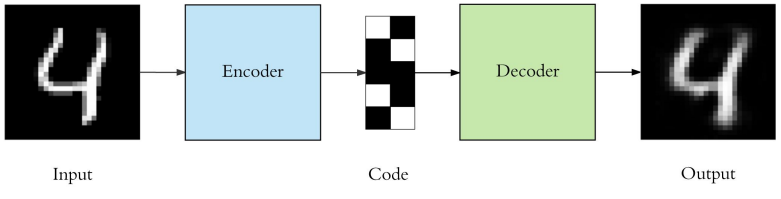
Principal Component Analysis
(Dimensionality reduction)

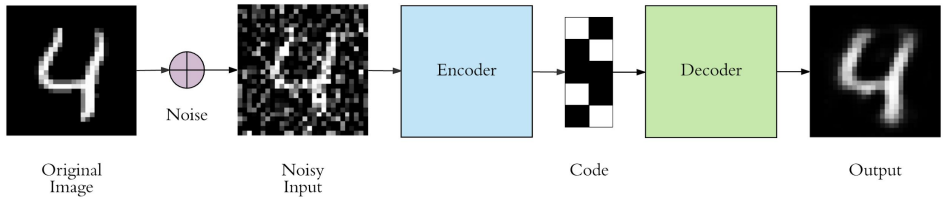
Autoencoders

Sparse autoencoders: Find a sparse representation of the input data.

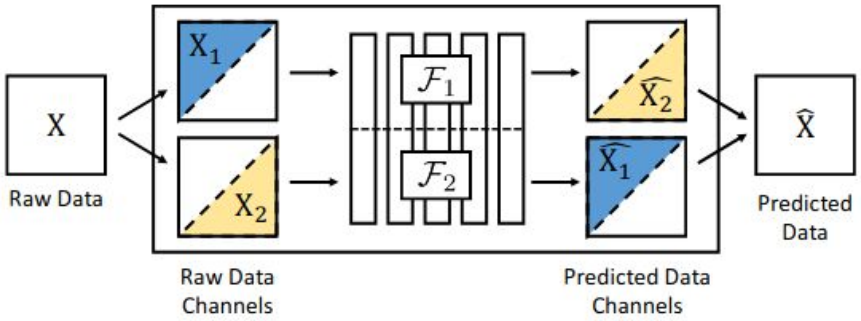
$$\min_{\boldsymbol{f},\boldsymbol{g}} \underbrace{\frac{1}{n}\sum_{i=1}^{n}\mathcal{L}\left(\boldsymbol{x}_i, \boldsymbol{g}\left(\boldsymbol{h}_i\right)\right)}_{\text{loss}} + \underbrace{\lambda\left\|\boldsymbol{h}_i\right\|_1}_{\text{regularizer}} \quad \text{with} \quad \boldsymbol{h}_i = \boldsymbol{f}\left(\boldsymbol{x}_i\right), \text{ for all } i \in [n].$$

Denoising autoencoders: The model must be robust to noise

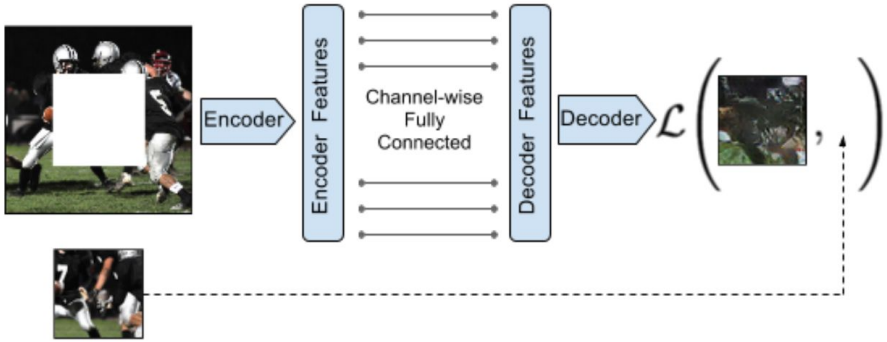




Split-Brain autoencoder



Context autoencoder



Generative Models

Training data ~ $p_{data}(x)$

Generated samples ~ $p_{model}(x)$

Want to learn $p_{model}(x)$ similar to $p_{data}(x)$

Taxonomy of Generative Models

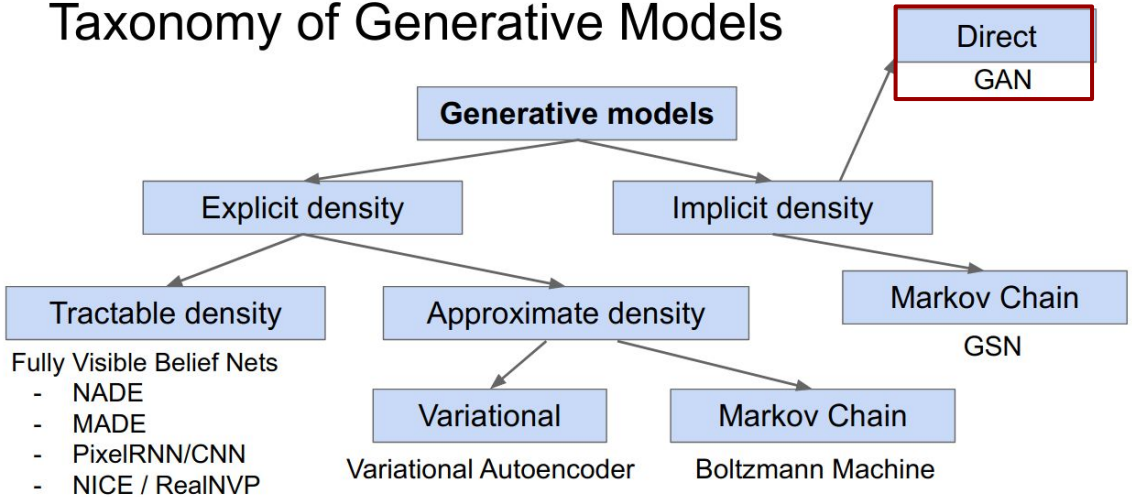


Figure copyright and adapted from Ian Goodfellow, Tutorial on Generative Adversarial Networks, 2017.

Generative Adversarial Networks

PixelCNNs define tractable density function, optimize likelihood of training data:

$$p_\theta(x) = \prod_{i=1}^{n} p_\theta(x_i | x_1, ..., x_{i-1})$$

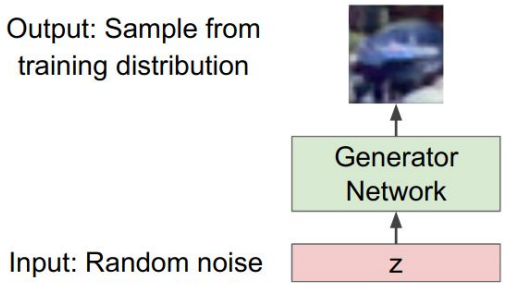
VAEs define intractable density function with latent **z**:

$$p_\theta(x) = \int p_\theta(z) p_\theta(x|z) dz$$

Cannot optimize directly, derive and optimize lower bound on likelihood instead

What if we give up on explicitly modeling density, and just want ability to sample?

GANs: don't work with any explicit density function!
Instead, take game-theoretic approach: learn to generate from training distributior through 2-player game

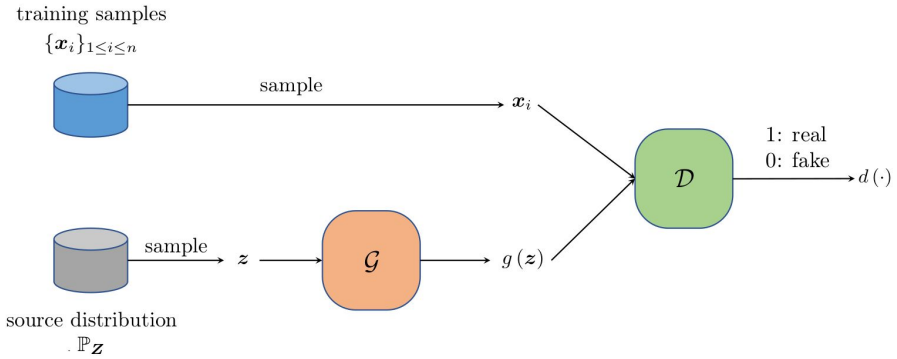


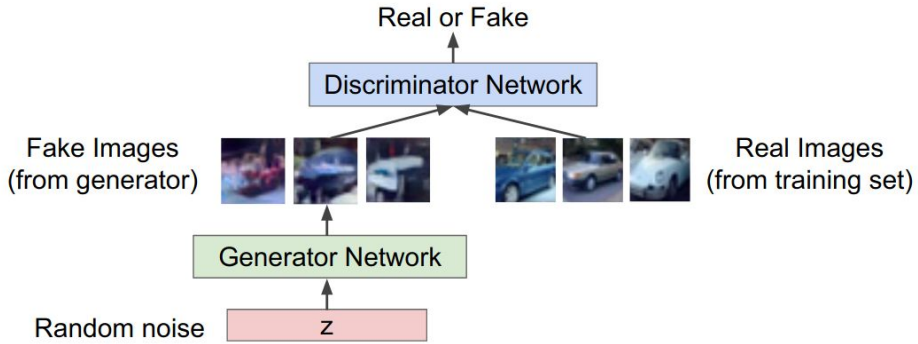
Generative Adversarial Networks

Aimed at density estimation: estimating the underlying probability density function $\mathbb{P}_X$ from which the data is generated.

GaNs are capable of estimation the density in high dimensional spaces (Images):

- GANs put more emphasis on sampling from the distribution **Px** than estimation.
- GANs define the density estimation implicitly through a source distribution **Pz** and a generator function **g(·).**





Discriminator outputs likelihood in (0,1) of real image

Minimax objective function:

$$\min_{\theta_g}\max_{\theta_d}\left[\mathbb{E}_{x\sim p_{data}}\log \underbrace{D_{\theta_d}(x)}_{\text{Discriminator output for real data x}} + \mathbb{E}_{z\sim p(z)}\log(1-\underbrace{D_{\theta_d}(G_{\theta_g}(z))}_{\text{Discriminator output for generated fake data G(z)}})\right]$$

Generative Adversarial Networks

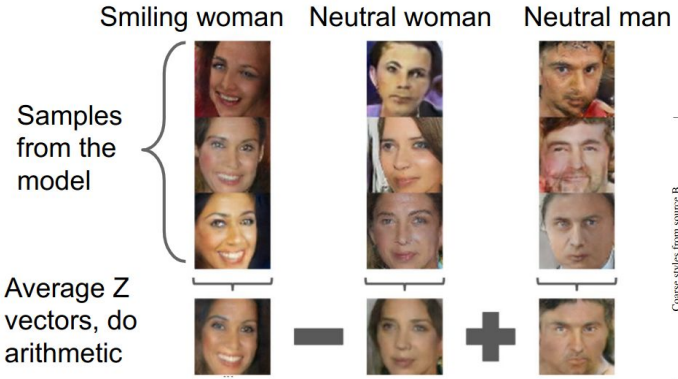
for number of training iterations **do**
  for k steps **do**
    - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    - Update the discriminator by ascending its stochastic gradient:

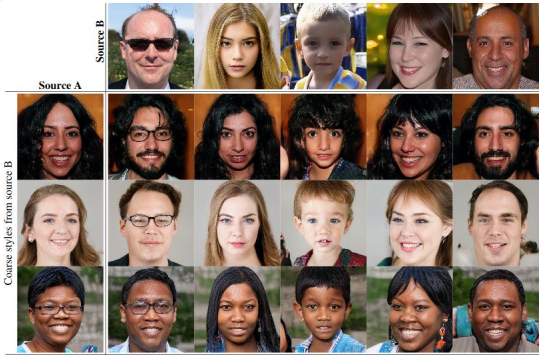
$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D_{\theta_d}(x^{(i)}) + \log(1 - D_{\theta_d}(G_{\theta_g}(z^{(i)}))) \right]$$

  end for
  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by ascending its stochastic gradient (improved objective):

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log(D_{\theta_d}(G_{\theta_g}(z^{(i)})))$$

end for

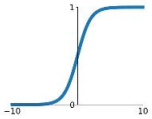




Training Deep Nets

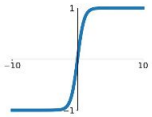
Sigmoid

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



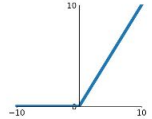
tanh

$\tanh(x)$



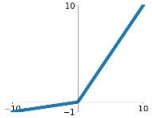
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

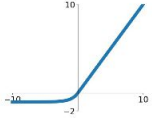


Maxout

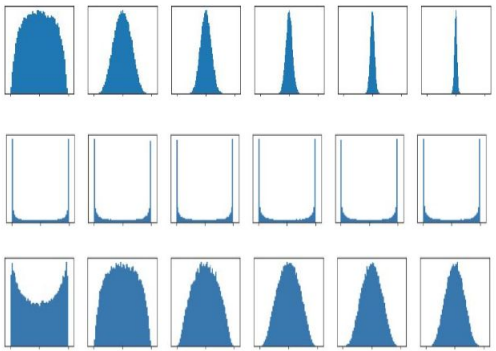
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Activations

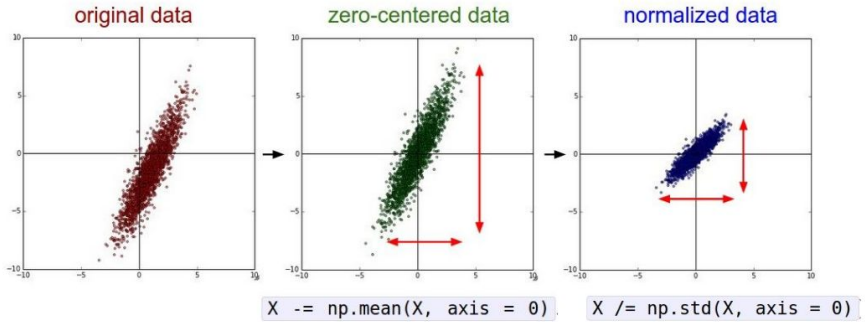


Initialization too small:
Activations go to zero, gradients also zero,
No learning =(

Initialization too big:
Activations saturate (for tanh),
Gradients zero, no learning =(

Initialization just right:
Nice distribution of activations at all layers,
Learning proceeds nicely

Initializing Weights



Data preprocessing

Input: $x : N \times D$

Learnable scale and shift parameters:

$\gamma, \beta : D$

Learning $\gamma = \sigma$, $\beta = \mu$ will recover the identity function!

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$ Per-channel mean, shape is D

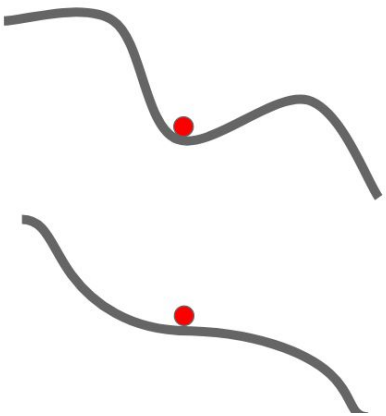
$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$ Per-channel var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$ Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$ Output, Shape is N x D

Batch Normalization

Training Deep Nets: Optimization



SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

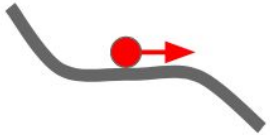
SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

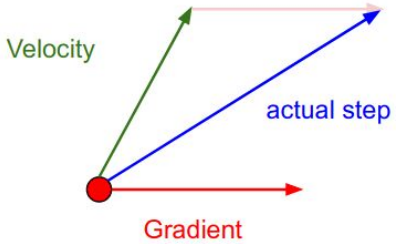
Nesterov Momentum

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

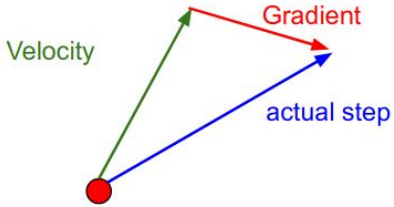
Saddle points



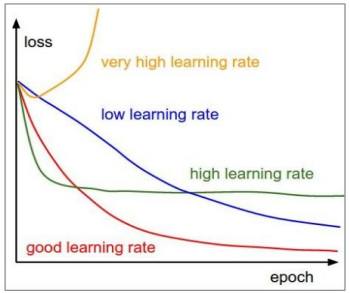
Momentum update:



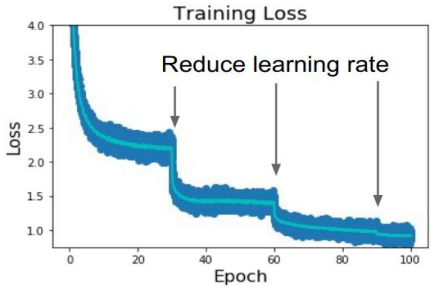
Nesterov Momentum

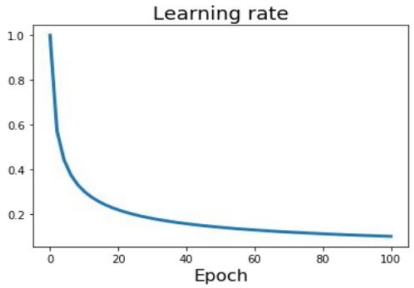


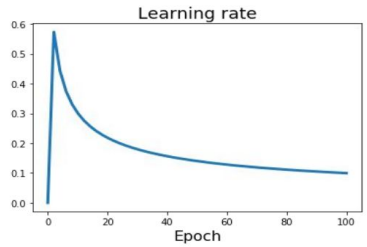
Learning rate Decay

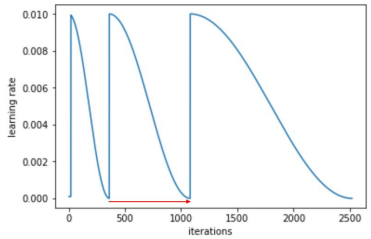


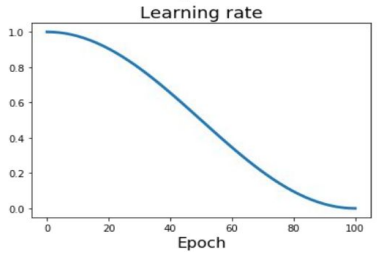
How to find the correct learning rate ?

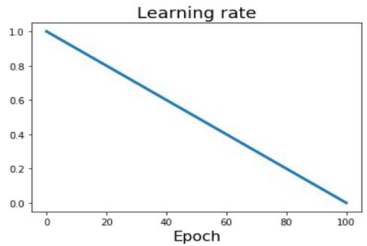




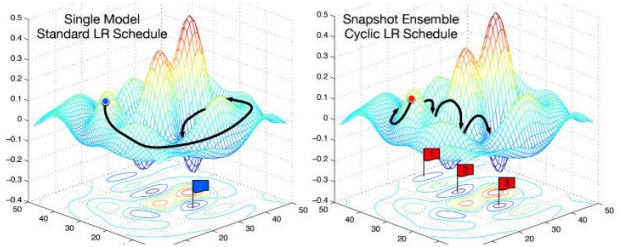


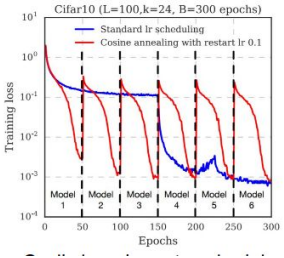




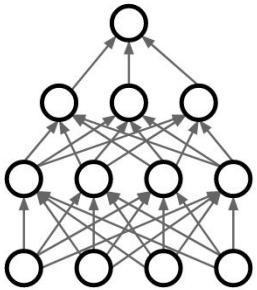


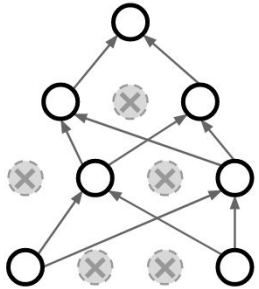
Model Ensembles





SGDR





Dropout

A there is a lot more .....

neural networks practitioner
music = loss function

Acknowledgements

- Stanford's CS231n
- The Unreasonable Effectiveness of Recurrent Neural Networks By Andrej Karpathy
- Understanding LSTM Networks By Chris Olah
- A Selective Overview of Deep Learning By Jianqing Fan, Cong Ma, Yiqiao Zhong
- Deep Learning Book by Ian Goodfellow and Yoshua Bengio and Aaron Courville
- Visualizing and Understanding Convolutional Networks By Matthew D. Zeiler and Rob Fergus
- Some Figures from the following papers: GoogleNet, ResNet, AlexNet, Split-brain autoencoders, Context autoencoder, SGDC, DC-GaN, Style-GaN.

The End.