Deep Learning

Frédéric Guyon

Octobre 2018

Apprentissage

Apprentissage=construction d'un modèle de données

- Supervisé: variables en entrée "X", et des variables à prédire "Y"

  classification : Y=classes ou probabilité des classes
  régression : prédiction de valeurs Y
- Non supervisé: variables en entrée "X"
  modélise distribution des X
  partionnement (Kmeans par exemple)
  en particulier modèle génératif

Apprentissage supervisé et non supervisé

- Observations à modéliser $\{x_i\}$ ou variable aléatoire Z
- Construire une fonction de décision f qui renvoie une valeur réelle ou entière
- Fonction d'erreur (Loss) (moindre carrés, vraisemblance)
- f obtenue en minimisant $E[L(f, Z)]$
- Supervisée $Z = (X, Y)$
  - régression: Y valeur réelle $L(f, (X, Y)) = \|f(X) - Y\|^2$ ou $L(f, (X, Y)) = \sum_i (f(x_i) - y_i)^2$
  - classification: Y numéro de classe, $f_i(X) = P(Y = i|X)$ $L(f, (X, Y)) = -\log(P(Y|X))$

Apprentissage non supervisée

- Données non catégorisées
- f estimateur de $P(Z)$=distribution de probabilité des données
- En gros, pour simplifier des données, conserver ce qui est pertinent
- EM=estimation de paramètres d'une densité de probabilité
- ACP = exemple de Compression de données ou filtrage (suppression du bruit)
- Kmeans = exemple de catégorisation/compartimentation des données
- Recherche simplification des données non catégorisées

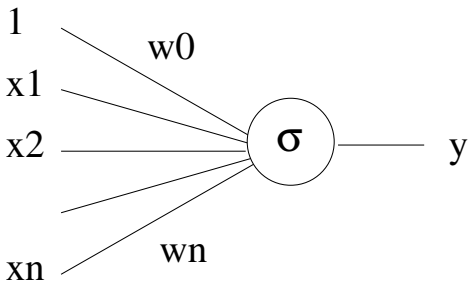
Apprentissage non supervisée

- Problème: généralisation, erreur d'apprentissage/ erreur de test
- Généralisation locale si x_i et x_j proches implique $f(x_i)$ et $f(x_j)$ proche
- Généralisation locale difficile pour données en grandes dimensions (curse of dimensionality)
  nécessité d'apprendre 2^d variations
- Généralisation globale: permet l'extrapolation
- Clustering: représentation locale/ PCA: non locale

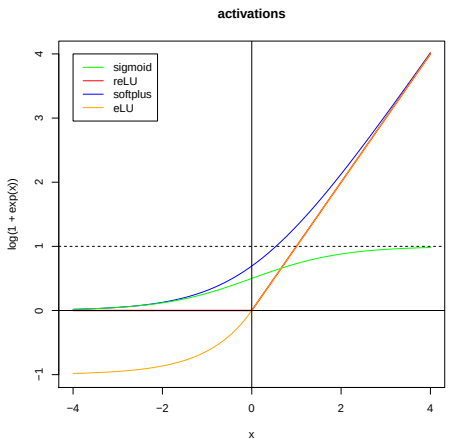
Réseau de neurones: historique

- 1980 : Artificial Neural Network (ANN), K. Fukushima
- 1989: Algorithme de "Back-propagation" du gradient appliqué à NN à plusieurs couches (ZIP codes)
- Problème: apprentissage difficile et long
  vanishing gradient
- 1990-2000 Problèmes de convergence, lenteur ont favorisés l'émergence des SVM
- Toutes les méthodes d'apprentissage, classification et modélisation très dépendantes de la représentation des données
- 2007 : apparition du terme "Deep Learning" (Hinton): RBM+backpropagation
- Actuellement, méthodes les plus performantes sur benchmarks d'évaluation: TIMIT (Reconnaissance de la parole), MNIST (images de chiffres manuscrits)

Réseaux de neurones: le neurone

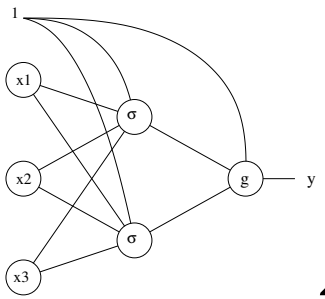


$$y = \sigma\left(w_0 + \sum_{k=1}^{n} w_k x_k\right)$$



sigmoïde	$\sigma(x) = \frac{1}{1+\exp(-x)}$
softplus	$\sigma(x) = \log(1 + \exp(x))$
rectified linear unit (reLU)	$\sigma(x) = \max(0, x)$
exponential linear unit (eLU)	$\sigma(x) = exp(x) - 1$ si $x < 0$ x sinon

Réseaux de neurones



$$z = g(y)$$

- $$y = N(x, w)$$

  avec x: vecteurs représentant object dont on veut prédire la valeur/la classe
  y: valeur/classe prédite
  N: fonction répresentée par le réseaux qui dépend des poids w
- Apprentissage = évaluation des poids w

$$\min_w \|y_{attendu} - N(x, w)\|^2$$

Fonction nnet: 1 couche cachée

- Une seule couche cachée (hidden layer)
- Une couche de sortie avec fonction d'activation selon: apprentissage ou régression
  - Classification: nombre de sorties=nombre de classes
    Fonction softmax:

$$g_k(x) = \frac{\exp(y_k)}{\sum_l \exp(y_l)}$$

  - Régression: approximation de $y = F(x)$ avec $x \in \mathbb{R}^n$ et $y \in \mathbb{R}^m$ nombre d'entrée=n,nombre de sorties=m
    Fonction linéaire:

$$g_k(x) = w_k^0 + w_k^\top z_k$$

Fonction nnet

m: nombres de sorties, N: nombres de vecteurs d'apprentissage

- Critère de minimisation:
  - Pour la régression, moindres carrés

$$J(w) = \sum_{k=1}^{m} \sum_{i=1}^{N} \left(y_k^i - N_k(x^i, w)\right)^2$$

  - Pour la classification, Cross-entropy:

$$J(w) = -\sum_{k=1}^{m} \sum_{i=1}^{N} y_k^i \log(N_k(x^i, w))$$

- Cross-entropy + softmax= régression logistique avec maximum de vraisemblance
- En cas de sur-paramétrisation, régularisation:

$$J(w, \lambda) = J(w) + \lambda \|w\|^2$$

Evaluation du gradient

- Itération de gradient

$$\beta_{km} \leftarrow \beta_{km} - \gamma \sum_{i=1}^{N} \frac{\partial J_i}{\partial \beta_{km}}$$

$$\alpha_{ml} \leftarrow \alpha_{ml} - \gamma \sum_{i=1}^{N} \frac{\partial J_i}{\partial \alpha_{ml}}$$

- Pas γ appelé taux d'apprentissage
- Calcul du gradient appelé back-propagation du gradient
- Deux procédures d'apprentissage: online ou batch
- Bibliothèque R (nnet) algorithme de minimisation : Quasi-Newton

Evaluation du gradient

- Entrée= N échantillons $x \in \mathbb{R}^p$ x^i $1 \leq i \leq N$
- M neurones dans la couche cachée : h_m $1 \leq m \leq M$
- Sortie = K classes y_k $1 \leq k \leq K$
- Fonctions du réseau

$$h_m = \sigma(\alpha^\top x)$$

$$y_k = g_k(\beta^\top z)$$

$$y = N(x; \alpha; \beta)$$

- Fonctions de moindres carrés

$$J(\alpha, \beta) = \sum_{i=1}^{N} \sum_{k=1}^{K} (y_{ik} - N_k(x^i))^2 = \sum_{i=1}^{N} J_i(\alpha, \beta)$$

Back-propagation du gradient

- Technique pratique de calcul du gradient
- échantillon x à l'entrée du réseau de neurone
- Propagation en avant dans les couches du réseau de neurones : $x_k^{(n-1)} \mapsto x_j^{(n)}$

$$x_j^{(n)} = g(h_j^{(n)}) = g(\sum_k w_{jk}^{(n)} x_k^{(n-1)})$$

- sortie y et erreur $e = y - y_{target}$
- Propagation en arrière de l'erreur $e_i^{(n)} \mapsto e_j^{(n-1)}$:

$$e_j^{(n-1)} = \sigma'(h_j^{(n-1)}) \sum_i w_{ij} e_i^{(n)}$$

- mise à jour les poids :

$$\Delta w_{ij}^{(n)} = \rho e_i^{(n)} x_j^{(n-1)}$$

Deep Learning: Introduction

- Peut être vu comme des réseaux de neurones améliorés avec plusieurs couches
- A la fois non-supervisé et supervisé
- Permettent d'apprendre ce qu'il faut apprendre: apprentissage non supervisé d'une représentation des données (feature learning)
- Représentation des données hiérarchisée: différents niveaux d'abstraction

Difficultés des NN

- Définir les caractérisques à apprendre (features)
- L'efficacité des NN est très dépendantes des caractéristiques
- Méthode de back-propagation du gradient très lentes
- Lorsque plusieurs couches, problème de "gradient vanishing" : corrections du plus en plus faibles en back-propagation
- Problème de généralisation lorsque beaucoup de neurones=sur-apprentissage
- En général, difficultés accrues avec nombre de neurones

Comment ces difficultés ont été surmontées ?

1. Gradient stochastique
2. Régularisations
3. Volume des données disponibles et puissance de calcul

Gradient stochastique

- Fonction = somme des erreurs pour chaque échantillon d'apprentissage

$$L(w) = \sum_{i=1}^{n} L_i(w)$$

- Par exemple: $L_i(w) = \|N(x_i, w) - y_i\|^2$
- Méthode classique: itération de descente de gradient (steppest descent)

$$w \leftarrow w - \rho \nabla L(w)$$

$$w \leftarrow w - \rho \sum_{i=1}^{n} \nabla L_i(w)$$

avec ρ pas de gradient ou taux d'apprentissage

- Sur gros jeu de donnée, gradient coûteux à calculer

Gradient stochastique

- Gradient stochastique: gradient approché par une somme partielle
- tirage d'un échantillon aléatoire des x_i
- si un seul terme x_i : méthode dite en ligne (on line)
- si plusieurs termes x_i: méthode dite par lot (mini-batch)
- Quand tout les x_i sont passés en revue : une époque (epoch)
  - tant que non convergence
  - pour chaque époque
  - mélange aléatoire des échantillons
  - pour i=1 à n : $w \leftarrow w - \rho \nabla L_i(w)$
- La méthode converge presque sûrement vers un minimum local

Optimiseurs: les méthodes de gradients

- On a un coût $L(w; X, y)$ à minimiser en fonction des poids w
- méthode de gradient:

$$w \leftarrow w - \eta \nabla_w L(w)$$

$$w \leftarrow w - \eta \sum_{i \in batch} \nabla_w L_i(w)$$

η= pas de gradient = taux d'apprentissage
batch (lot): ensemble ou sous-ensemble d'apprentissage utilisé dans le calcul du gradient

- Grosses difficultés liées au pas fixe

Optimiseurs: les méthodes de gradients

Stochastic Gradient Descent (SGD)

- Un pas de gradient pour chaque éléments de l'apprentissage
- Epoque: présentation de tout l'ensemble d'apprentissage

$$w \leftarrow w - \eta \nabla_w L_i(w)$$

- En général, l'ensemble est mélangé pour chaque époque
- La descente (minimisation de L) n'est plus assurée: oscillation de L
- Convergence asymptotique assurée si le taux d'apprentissage décroit

Optimiseurs: les méthodes de gradients

Mini-batch gradient descent

- mini-batch=sous-ensemble d'apprentissage (tiré aléatoirement)

$$w \leftarrow w - \eta \sum_{i \in mini\_batch} \nabla_w L_i(w)$$

- converge plus rapidement, et plus stable (moins d'oscillation)
- époque: tout l'ensemble est visité
- Avec très gros jeu de données: mini-batch entre 50 and 256

Optimiseurs: les méthodes de gradients

Gradient avec inertie (momentum)

- d direction de descente est une combinaison de: -∇L et la direction de descente précédente
- but: éviter oscillation et accélérer la convergence

$$d = \gamma d_{prec} - \eta \nabla L(w)$$

$$w \leftarrow w + d$$

Optimiseurs: les méthodes de gradients

Accélération de Nestorov du gradient (NAG)

- d direction de descente est une combinaison de:
  -∇L estimé à la prochaine itération et la direction précédente
- but: éviter oscillation et accélérer la convergence
- plus efficace et stable que le momentum simple.

$$d = \gamma d_{prec} - \eta \nabla L(w + \gamma d_{prec})$$

$$w \leftarrow w + d$$

Optimiseurs: Adagrad

Méthodes de gradient avec pas adaptatif

- le pas est adapté pour chaque paramètre
- le pas dépend de la norme (taille) des gradients

$$w_i \leftarrow w_i - \frac{\eta}{\sqrt{G_i^2 + \epsilon}} \nabla_{w_i} L(w)$$

G_i=somme des carrés des composantes i des gradients précédents

- le pas dépend de tous les gradients précédents
  devient très petit, convergence lente

Optimiseurs: Adadelta et RMSprop

- Adadelta et RMSprop développés indépendamment mais similaires
- Adadelta: dépend seulement implicitement des derniers gradients calculés:

$$E[G^2] = \gamma E[G^2_{prec}] + (1-\gamma) g^2$$

$E[G^2]$ correspond à une moyenne des norme des gradients

$$w_i \leftarrow w_i - \frac{\eta}{\sqrt{E[G^2] + \epsilon}} \nabla_{w_i} L(w)$$

- Adadelta et RMSprop améliorent Adagrad et sont effectivement utilisé
- Paramètres par défaut: $\gamma = 0.9$ et $\eta = 0.001$

Optimiseurs: Adam

Méthode à inertie adaptative (adaptative momentum estimation)
combinaison de RMSprop et méthode avec inertie

- $$m = \beta_1 m_{prec} + (1 - \beta_1) g$$
$$v = \beta_2 v_{prec} + (1 - \beta_2) g^2$$

m: estimation de la moyenne des gradients
v: estimation des normes des gradients $E[g^2]$

$$w \leftarrow w - \frac{\eta}{\sqrt{v} + \epsilon} m$$

Optimiseurs: AdaMax

$$v = \beta_\infty v_{prec} + (1 - \beta_\infty)\|g\|_\infty$$

v estimation des normes infinies des gradients $E[\|g\|_\infty]$

$$\|g\|_\infty = \max |g_i|$$

$$w \leftarrow w - \frac{\eta}{v} m$$

Optimiseurs: Nadam

Combinaison de NAG (Nesterov Accelerated Gradient) et méthode avec inertie (momentum)

$$g = \nabla L(w - \gamma m_{prec})$$

$$v = \beta_2 v_{prec} + (1 - \beta_2) g^2$$

$$m = \gamma m_{prec} + \eta g$$

$$w \leftarrow w - \frac{\eta}{\sqrt{v} + \epsilon}(\beta_1 m + (1 - \beta_1) g)$$

Optimiseurs: AMSGrad

Très utilisé en pratique

$$m = \beta_1 m_{prec} + (1 - \beta_1) g$$

$$v = \beta_2 v_{prec} + (1 - \beta_2) g^2$$

$$\hat{v} = \max(\hat{v}_{prec}, v)$$

$$w \leftarrow w - \frac{\eta}{\sqrt{\hat{v}} + \epsilon} m$$

Régularisations

- Pour les problèmes d'optimisation mal posés: $\min_w L(w)$
  Dans les cas de surparamétrisation: pas assez de données par rapport à la taille du modèle
- Régularisation L2

$$\min_w L(w) + \alpha \sum_k w_k^2$$

  avec $\alpha > 0$ problème toujours bien posé, donne des w_k pas trop grands
- Régularisation L1

$$\min_w L(w) + \alpha \sum_k |w_k|$$

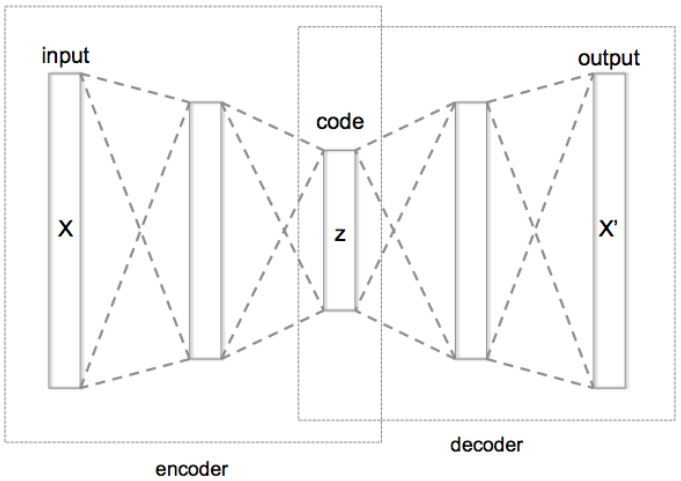
  réduit le nombre de paramètres w_k non nuls
- Dropout: Mise à zéros aléatoires de certaines valeurs de neurones
- Troisième méthode: arrêt prématuré des itérations de gradients

Types de Réseaux

- DNN: (Deep) Neural Network: feedforward NN
- DBN: (Deep) Belief Network : construit sur Restricted Boltzmann Machines
- Auto Encoders
- RNN: Recurrent Neural Networks
- CNN: Convolutional Neural Networks

Deep Belief Network Auto Encoder

- au minimum 2 couches :
  1 couche qui encode :$y = \sigma(Wx + b)$
  1 couche qui décode : $x' = \sigma(W'y + b')$
- 1 couche qui transforme les données initiales
  + 1 couche qui reconstruit
- avec plusieurs couches (réseau diabolo)



from Wikipedia::Autoencoder

- Généralisation de l'ACP avec réduction de dimension:
  ACP à k composantes= couche cachée linéaire avec k neurones + moindre carrés

Réseaux Auto Encoder

- Apprentissage de $X \leftrightarrow X$: entrées X et sorties X
- Fonction d'erreur à minimiser pour trouver poids:
  - moindres carrés
    $$L(x, z) = \|x - z\|^2$$
  - cross-entropy
    $$L(x, z) = -\sum_k x_k \log(z_k) + (1 - x_k) \log(1 - z_k)$$
- Minimisation avec gradient stochastique et/ou régularisation
- Evite d'apprendre le réseaux "identité"

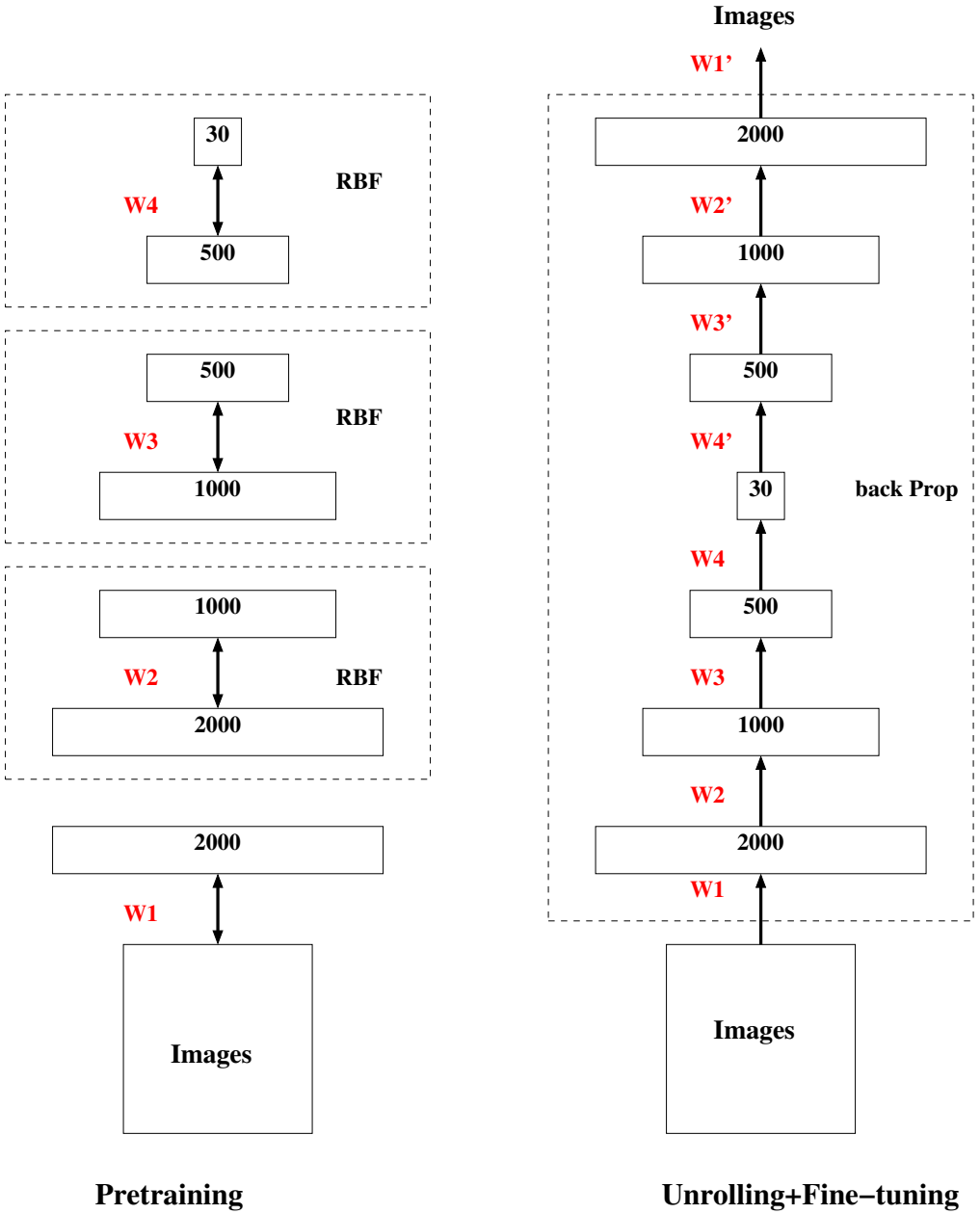
Réseaux Denoising Auto Encoder

- Apprentissage de X bruité $\leftrightarrow$ X non bruité
- Bruit : mise à zéro d'une proportion donnée d'entrées (Vincent, 2008)
- Permet de reconstruire une version débruité à partir d'une entrée dégradée
- Permet de reconstruire une version complète ou complété d'une entrée incomplète avec valeurs manquantes
- Plus robuste

Algorithme DBN

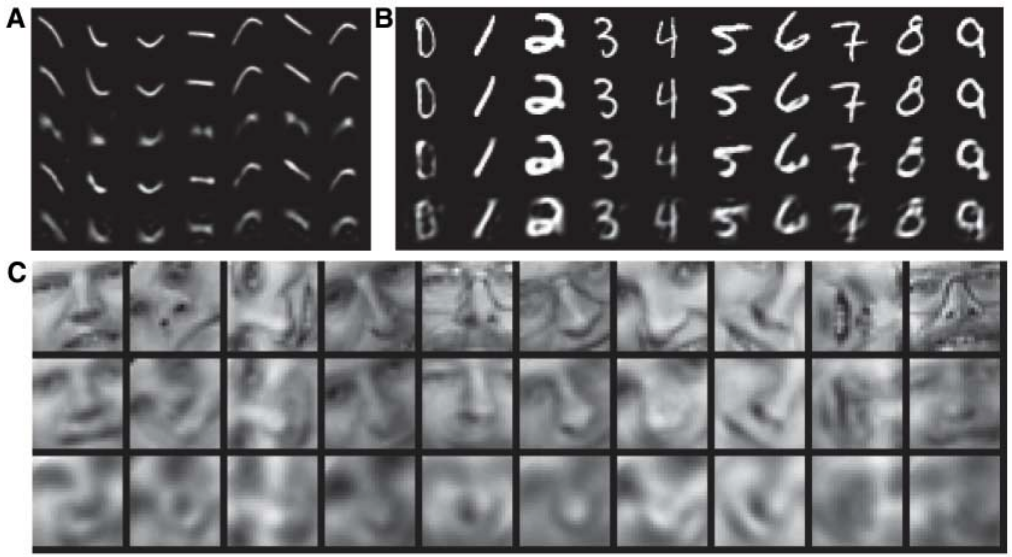
- Pre-training: Apprentissage non supervisé de RBM séparés
- Unrolling: Apprentissage supervisé des RBM assemblés
- Fine-tuning: Amélioration des poids par réseaux de neurones classiques

Deep Architecture: Autoencoder



from Hinton et Salakhutdinov, Science, 313, 2006

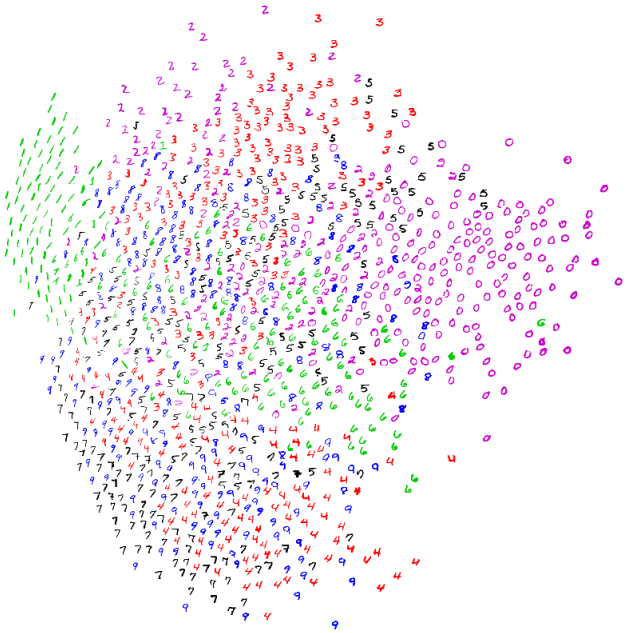
Deep Architecture: Autoencoder



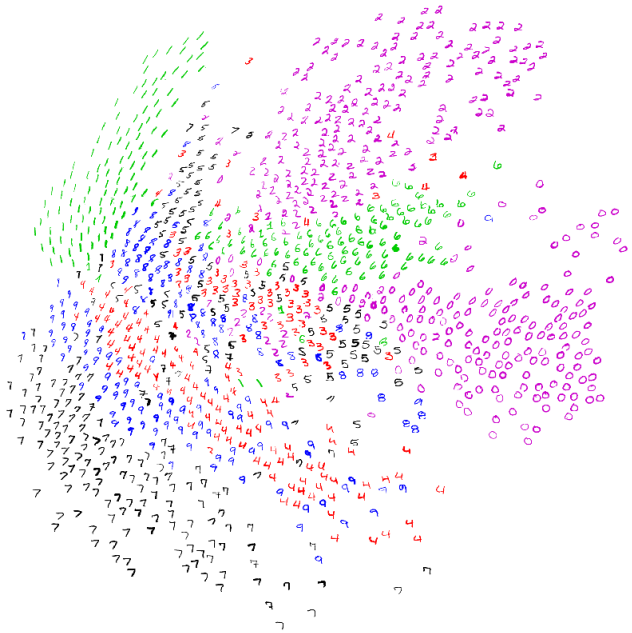
from Hinton et Salakhutdinov, Science, 313, 2006

Réseaux Auto Encoder: example MNIST

ACP



Deep Belief Network



from Hinton et Salakhutdinov, Science, 313, 2006

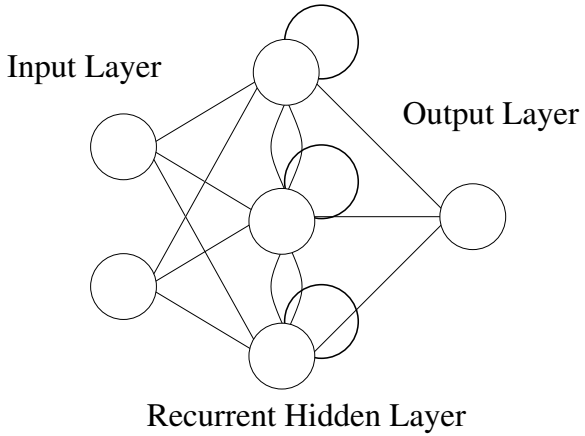
Réseaux Denoising Auto Encoder

Résultats sur données MNIST

Dataset	SVMrbf	SVMpoly	DBN-1	SAA-3	DBN-3	SdA-3 (ν)
basic	3.03±0.15	3.69±0.17	3.94±0.17	3.46±0.16	3.11±0.15	2.80±0.14 (10%)
rot	11.11±0.28	15.42±0.32	14.69±0.31	10.30±0.27	10.30±0.27	10.29±0.27 (10%)
bg-rand	14.58±0.31	16.62±0.33	9.80±0.26	11.28±0.28	6.73±0.22	10.38±0.27 (40%)
bg-img	22.61±0.37	24.01±0.37	16.15±0.32	23.00±0.37	16.31±0.32	16.68±0.33 (25%)
rot-bg-img	55.18±0.44	56.41±0.43	52.21±0.44	51.93±0.44	47.39±0.44	44.49±0.44 (25%)
rect	2.15±0.13	2.15±0.13	4.71±0.19	2.41±0.13	2.60±0.14	1.99±0.12 (10%)
rect-img	24.04±0.37	24.05±0.37	23.69±0.37	24.05±0.37	22.50±0.37	21.59±0.36 (25%)
convex	19.13±0.34	19.82±0.35	19.92±0.35	18.41±0.34	18.63±0.34	19.06±0.34 (10%)

from Vincent,2008

Recurrent Neural Network

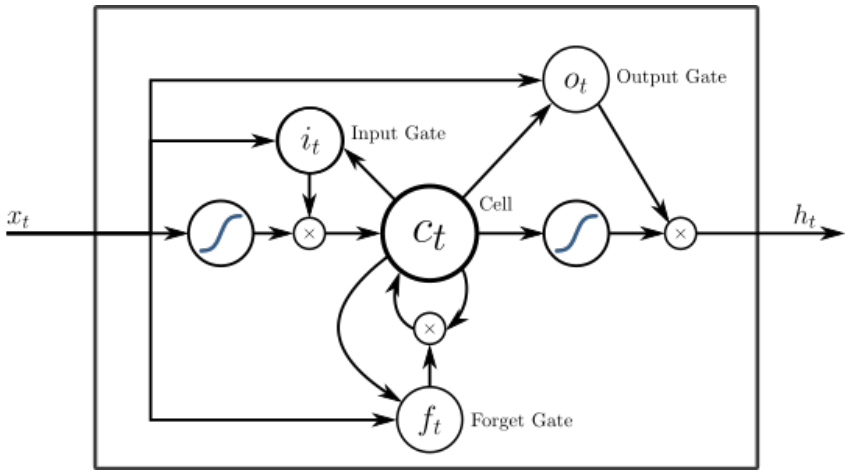


- Boucles: noeuds dont la valeur dépend des valeurs précédentes
- Pour signaux temporels ou successifs: reconnaissance de l'écriture, reconnaissance vocale, films
- par exemple LSTM
- Poids dépendant du temps

LSTM: Long Short Term Memory

- RNN difficile de connecter informations à long terme
- LSTM sont des RNN avec des neurones plus complexes
- neurones=machines effectuant plusieurs opérations: mise en mémoire, effacement de la mémoire, copie des valeurs, fonctions d'activations

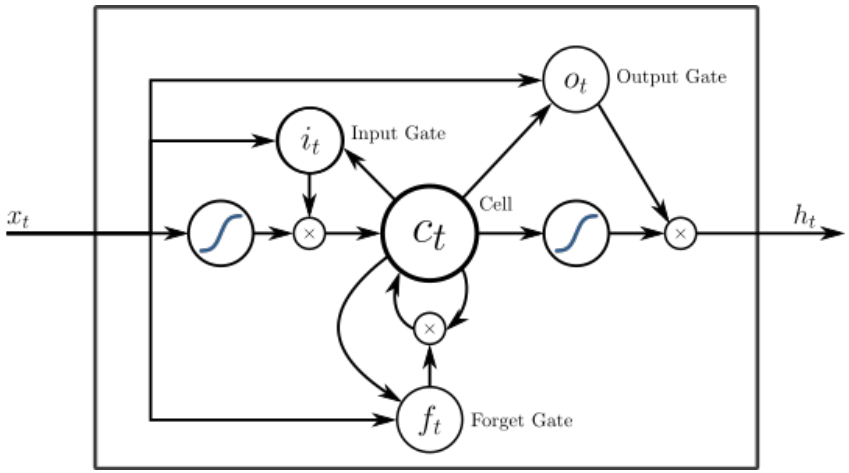
Long Short Term Memory



Long short-term memory, Wikipedia

- Recurrent NN: Google Translate, Amazon Alexa, Microsoft atteint 95.1% de bonne reconnaissance vocale
- les RNN simples souffrent du "gradient vanishing": difficulté d'apprentissage
- noeuds: blocks de neurones: mémoire+read+write+reset
  input=1: on mémorise l'entrée x dans la mémoire (cell)
  output=1: on sort la mémoire
  reset=1: on conserve la précédente entrée

Long Short Term Memory



Long short-term memory, Wikipedia

$$f_t = \sigma_g(W_f x_t + U_f c_{t-1} + b_f)$$
$$i_t = \sigma_g(W_i x_t + U_i c_{t-1} + b_i)$$
$$o_t = \sigma_g(W_i x_t + U_i c_{t-1} + b_i)$$
$$c_t = f_t * c_{t-1} + i_t * \sigma_g(W_c x_t + b_c)$$
$$h_t = o_t * \sigma_h ct$$

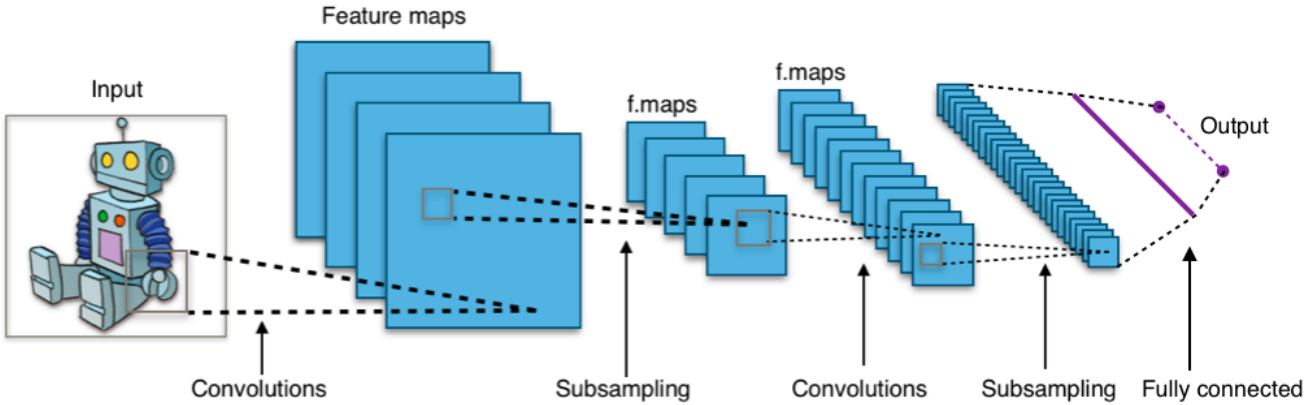
Long Short Term Memory

Théoriquement

- Un NN est un approximateur universel (universal approximation theorems):
  toutes les fonctions (assez régulières) peuvent être approchées par un NN
- Toutes les programmes peuvent être approchés par un LSTM ("RNNs are Turing-Complete")

Convolutional Neural Network

- Adapté aux signaux images et au son (voix)
- Opération de convolution: même filtre appliqué à tous les neurones
- Partage des poids et connectivité locale=moins de paramètres
- Invariance en translation pour les images



(Source: Wikipedia article: Convolutional Neural Network)

Deep Architecture

- De nombreuses couches (Deep): préférable d'avoir multiples couches qu'une seule grosse couche
- Meilleure généralisation
- Modèles plus compact (moins de neurones)
- Par analogie avec calcul algébrique: factorisation des données (Bengio, 2009)
- Optimisation couche par couche indépendamment (Hinton, 2006)
- Régularisation: arrêt prématuré des itérations de gradients
- Dernière étape de "fine-tuning" avec rétro-propagation du gradient sur toutes les couches

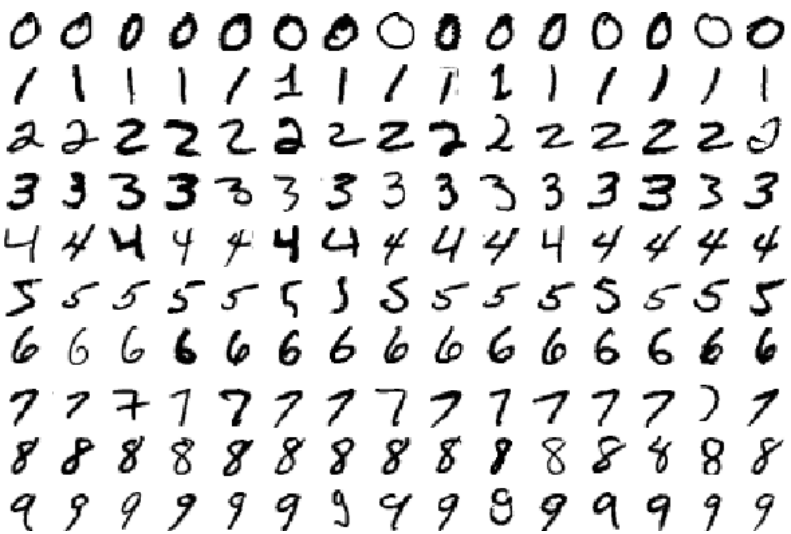
Applications

- Reconnaissance de la parole
- Reconnaissance d'image
- Analyse du langage
- Drug discovery
- Bioinformatique: prédiction des annotations de gènes
  prédiction des fonctions des gènes

MNIST data

`http://yann.lecun.com/exdb/mnist/`

- MNIST=Mixed National Institute of Standards and Technology database
- Banque d'images de chiffres manuscrits 28×28
- 60,000 images pour l'apprentissage et 10,000 images de test



Données MNIST

Linear classifier	1998	7.6
Non-Linear Classifier 40 PCA + quadratic classifier	1998	3.3
Support vector machine	2002	0.56
K-Nearest Neighbors	2007	0.52
Neural network (convolution, 2 couches)	2003	1.6
Deep neural network	2010	0.35
Convolutional Deep neural network	2012	0.23

Performances $>$ humain

Données MNIST

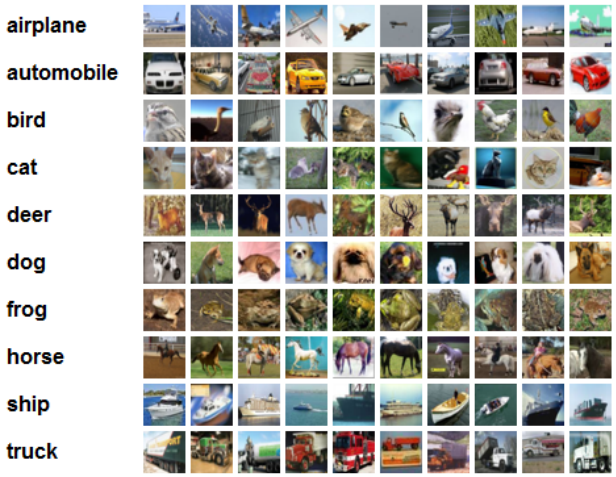
```
mnist <- dataset_mnist()
rotate <- function(x) t(apply(x, 2, rev))
image(1:28,1:28,rotate(mnist$train$x[1,,]),col=grey.colors(12))
mnist$train$y[1]
```

Données TIMIT

- Texas Instrument and M.I.T.
- Reconnaissance de la parole (téléphone)
- 630 orateurs appartenant à 10 “dialectes” anglais $\times$ 10 phrases

Random NN	1989	26.1
DNN	2014	18.2
DNN-HMM	2014	12.3
Human		2-4

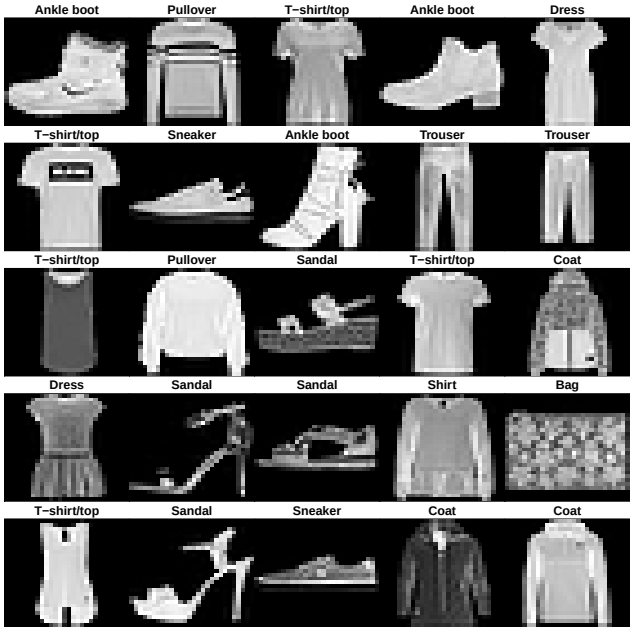
Données CIFAR



- CIFAR=Canadian Institute for Advanced Research
- CIFAR-10: 60000 images couleur 32x32 10 classes(6000 images par classe)
- CIFAR-100: 100 classes $\times$ 600 images

Meilleurs résultats pour CIFAR-10 en 2015 : 3.5 % erreurs
Meilleurs résultats pour CIFAR-10 en 2015: 25 % erreurs

Données MNIST-Fashion



- 60000 28x28 images en niveaux de gris
- 10 types de vêtements
- ensemble de validation de 10000 images.

Keras

- Interface de programmation avec TensorFlow
- TensorFlow: bibliothèque de fonctions pour l'apprentissage développé par Google
- Keras interface écrite en Python
- Keras en R: appelle fonction python

Keras Layer

Tous les calculs organisées en couches: activation mais aussi activation, output, dropout, etc...

- Core Layers: couches indispensables input, dense, activation, dropout, reshape, flatten
- Recurrent Layers
- Convolutional Layers
- Pooling Layers

Fonctions coûts

Fonctions “Loss”

- loss_mean_squared_error
- loss_mean_absolute_error
- loss_binary_crossentropy (2 classes),
  loss_categorical_crossentropy (¿2 classes)
- loss_kullback_leibler_divergence

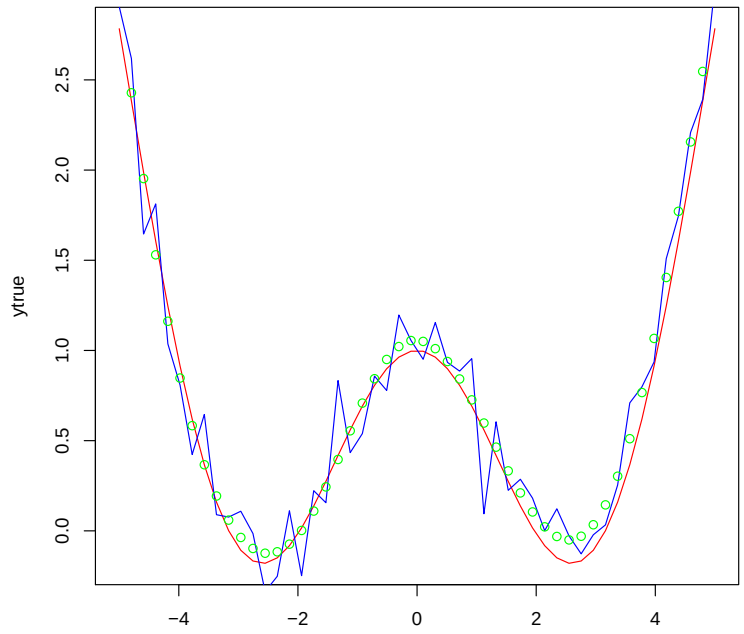
Optimiseurs: les méthodes de gradients

Fonctions keras pour l'optimisation:
toutes construites sur des pas de gradients

```
optimizer_adadelta  optimizer_adamax    optimizer_sgd
optimizer_adagrad   optimizer_nadam
optimizer_adam      optimizer_rmsprop
```

Exemples: Régression

```
library(keras)
model = keras\_model\_sequential()
layer\_dense(model, units=20, activation="relu", input_shap
layer\_dense(model, units=1, activation="linear", input_sha
compile(model,loss = "mean\_squared\_error", optimizer = op
fit(model, x, yobs, epochs = 1500, batch\_size = 10)
ypred=predict(model,x)
```



Exemples: Classification

```
X=as.matrix(iris[,1:4])
y=as.integer(iris[,5])-1
Y=to_categorical(y)
```

Exemples: Classification

```
Y=to\_categorical(y-1)
model = keras\_model\_sequential()
layer\_dense(model, units=20, activation="sigmoid", input_shape=2)
layer\_dense(model, units=2, activation="softmax", input_shape=20)
compile(model, loss = "categorical_crossentropy", optimizer = optimizer\_rmspro
fit(model, X,Y, epochs = 50, batch_size = 30)
classes=predict\_classes(model, X)
table(classes, y)
```

Exemples: Classification

```
train=sample(1:150,100)
x_train = as.matrix(iris[train, 1:4])
y_train = iris[train, 5]
x_test = as.matrix(iris[-train, 1:4])
y_test = iris[-train,5]
y_train = to_categorical(as.integer(y_train)-1)
y_test = to_categorical(as.integer(y_test)-1)
```

Exemples: Classification des iris

```
model = keras_model_sequential()
layer_dense(model, units = 5, activation = 'relu', input_shape = 4)
layer_dense(model,units = 3, activation = 'softmax')
model
compile(model,loss = 'categorical_crossentropy',optimizer = optimizer_rmsprop()
  metrics = 'accuracy')
fit(model, x_train, y_train, epochs = 500, batch_size = 50, validation_split =
evaluate(model, x_test, y_test)
classes=predict_classes(model, x_train)
table(classes, iris[train,5])
classes=predict_classes(model, x_test)
table(classes, iris[-train,5])
```