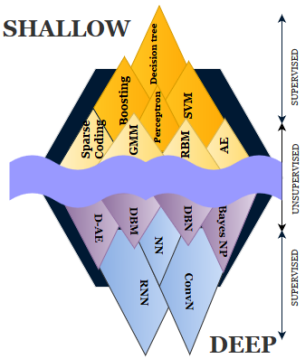
Introduction au Deep Learning

Célian RINGWALD

Université Lyon 2
celian.ringwald@free.fr
May 30, 2016



Abstract

This paper is the result of my research on the Deep Learning, a very hype subject of the Machine Learning nowadays, the occasion to find easily a lot very interesting ressources. I will try to summarising that in a smart way. We will introduce the subject by speaking about the history of this Machine Learning field, and, after that we will present the principal theory bases, on which are grounded this wide extension of Artifical Inteligence. Different uses cases will be placed in perspective with the most well-known kind of architectures. A quick benchmark of the actuals framework of development are presented. We will focus on Keras in the last part and implenting our first tiny deep network : a stacked LSTM. This one will help us for forecasting the snowing precipitations of tomorow near the Alta Lake in Canada.

Cet article est le résultats de diverses recherches effectuées sur le Deep Learning, un sujet très tendances ces derniers temps, l'occasion d'avoir accès à de nombreuses ressources de qualités. Je vais donc essayer de vous résumer cela dans un format adapté. Nous introduirons le sujet par le dessin de son histoire, et en exposerons les bases théoriques sur lequel cette extension de l'intelligence artificielle repose. Nous mettrons ensuite les principaux cas d'utilisation du Deep Learning en perspectives avec les architectures profondes les plus utilisés. Nous ferrons enfin le tour des différentes librairies nous permettant de les implémenter et nous focaliserons sur Keras par la suite. Cette librairie nous permettra de réaliser des prévisions concernant les chutes de neige près du Lac Alta au Canada via l'utilisation d'un réseaux LSTM (Long short term memory).

Table des matières

I Introduction **3**

II Le pourquoi du comment du Deep Learning **4**

III Principaux fondements théoriques **7**
I Le perceptron 7
II Quelques fonctions d'activations couramment utilisées 8
III L'algorithme de descente du gradient 9
IV Optimisations de la descente de gradient 10
V Techniques d'amélioration de l'apprentissage 11

IV Quatre architectures à retenir **13**
I Machine restreinte de Boltzman superficielle et profonde 14
II Auto-encoder et son extension profonde 17
III Réseaux convolutionnels 19
IV Les réseaux récurrents 22

V Supports de développement **25**

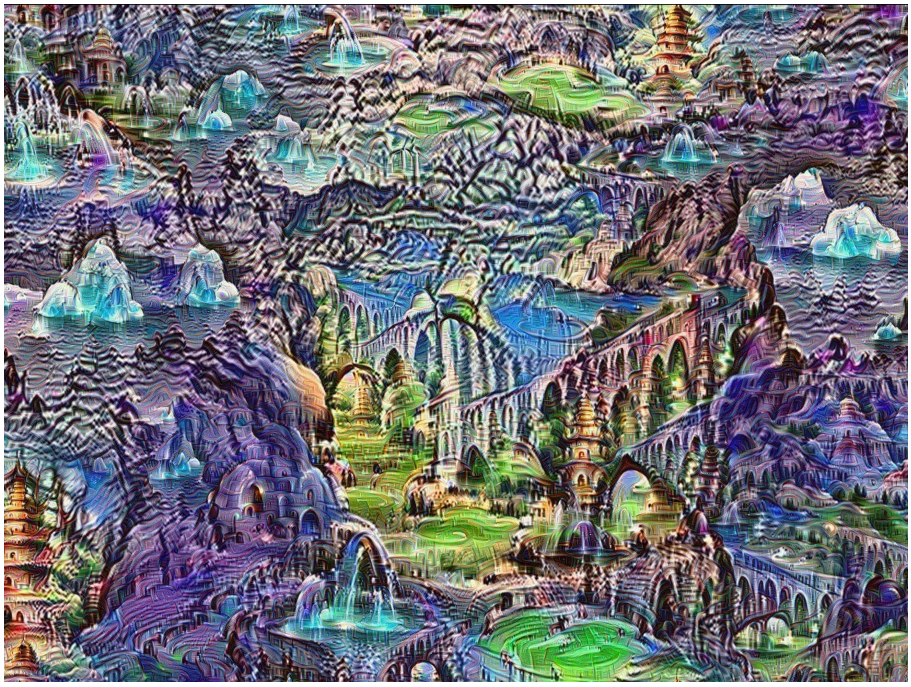
VI Tutoriel : la neige du lac d'Alta **26**
I Import des données 27
II Installation Keras et prise en mains 27
III Préparation des données 28
IV Créations des fonctions liés à l'apprentissage 29
V Visualisation des résultats 30
VI Simple LSTM 31
VII Simple LSTM à 2 couches 33
VIII Pour aller plus loin... 35

VII Conclusion **35**

I. Introduction

Le Deep Learnng occupe aujourd'hui une forte place dans l'actualité : la voiture autonome semble être annoncée (elle est même légiférée depuis mars 2016 par l'Europe des Nations unies, l'intelligence artificielle de Microsoft nommée Tay, chatbot twitter, devenue raciste, la victoire de l'algorithme AlphaGo de DeepMind contre Lee Sedol au jeux de Go, les images psychédéliques de Deep Dream de Google, ou encore la génération automatique de description d'image pour les aveugles chez Facebook. Ces progrès qui semblent à première vue sortie de nulle part, sont en réalité le fruit d'un long travail de recherche entrepris dans les années 80 par une poignée de chercheurs téméraires nous citerons par exemple Hinton, Andrew Ng, Andrej Kurparthy et bien entendu Yann LeCun (ce français de chez facebook qui a la charge actuellement du cours de Deep Learning au collège de France). Après une longue traversée du désert l'intelligence artificielle laisse place aujourd'hui à une véritable euphorie qui se traduit par des levée de fond considérables. Mais d'où viens cette monté en puissance de ce domaine ? En quoi ces nouveaux modèles apportent ils une véritable révolution dans le monde du Machine Learning ?

Figure 1: *Google Dream (@Source : Google)*



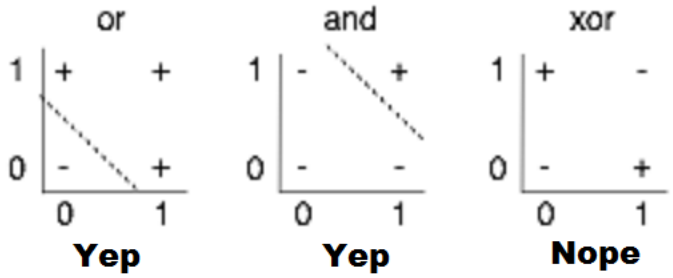
II. Le pourquoi du comment du Deep Learning

Le Deep Learning est avant tout un sous domaine du machine learning, méthodes visant à apprendre à travers des exemples de données afin de prévoir un comportement futur. C'est à travers l'étude des relations entre celles-ci que les systèmes de machine learning peuvent ensuite tenter de prévoir l'issue d'une variable de sortie face de nouveaux exemples observés. Les cas linéaires ont été amplement étudiés, mais les problèmes du monde réels présentent en général des propriétés non linéaires... Des méthodes tels que le SVM (1990) et les arbres de décisions (1960) sont capables de capturer des patterns non linéaires simples, mais demandent d'extraire au préalable des caractéristiques et donc de mener une étape de pré-traitement via à l'appui d'un expert.

Figure 2: *Frank Rosenblatt et sa machine à câblages fou (@Source : The New Yorker)*

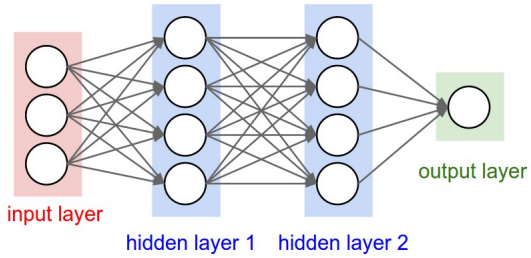
L'histoire du Deep learning commence tout d'abord avec la découverte du Perceptron en 1958 par Rosenblatt et son équipe du Cornell Aeronautical Laboratory. Inspiré par les découvertes du 20^e siècle sur le fonctionnement du cerveaux, Rosenblatt, avant tout psychologue, et le laboratoire de Cornell implémentent le premier réseaux de neurones. La mise en place de ce prototype était alors mécanique, faute de langages évolué. Ces réseaux permettent alors de réaliser des inférences sur des données binaires. Seulement le projet ne convainc pas la communauté scientifique : très long a entraîner le modèle ne résout que des cas linéairement séparables très limités.

Figure 3: *Le troisième cas est impossible à résoudre avec un perception(@Source : Andrey Kurenkov)*



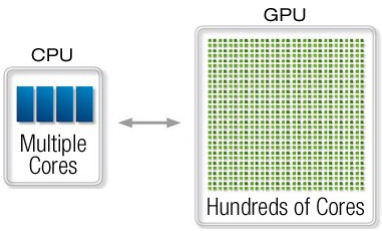
L'idée des couches cachés de neurones est alors vue plus tard comme censés pouvoir résoudre des problèmes plus complexes, la communauté scientifique se repenche alors sur le sujet car celles-ci captures des caractéristiques inhérentes aux données et résout le cas du XOR (Et/Ou booléens).

Figure 4: *Exemple de MLP, multi-layer perceptron (@Source : Andrey Kurenkov)*



L'entraînement de ce modèle s'effectue alors via la notion de rétrogradation du gradient, qui permet aux poids de chaque neurone de s'ajuster de manière à minimiser les erreurs. Minsky et Papert montrent alors qu'il est possible de résoudre des problèmes complexes avec les couches cachées. Cependant il reste encore très long à entraîner et ce type de modèle "'boite noire"' ne séduit que très peu de chercheurs. De plus, de nouveaux problèmes pratiques parviennent aux chercheurs qui se lancent dans la vague : les poids des neurones semblent diminuer étrangement et dans certains cas exploser. Le modèle semble être sujet à l'overfitting, et les calculs visant à optimiser notre réseaux sont très long... Le modèle est alors réduit à une couche cachée dans la pratique afin de ne pas avoir à faire à ce problème.

Figure 5: *Les GPU démultiplient la puissance de calcul @Source : Manuel Alves*



Il fallu attendre dans un premier temps le développement de calculateurs performants offrant la possibilités d'être plus gourmands en calcul, le flot de données du Big Data, ainsi que des chercheurs téméraires afin de résoudre les problèmes précédemment énoncés.
Yann LeCun et le ATT Bell Labs démontrent par la suite que les réseaux de neurones sont capables de reconnaître des lettres écrites à la main en travaillant sur un large jeux de données fournis par l'US Postal service. Ceci en améliorant le modèle de Hubel Wiesel sur le cortex visuel. Celui-ci est composé de couches successives de neurones repérant des caractéristiques locales, regroupées récursivement. Les modèles sur lesquels travaillent LeCun, Yukushima et Riesenhuer sont alors composés de multiples couches de normalisation, de filtrages, de transformations non-linéaires et de regroupement et se terminent alors généralement d'un classifieur.
Le système de LeCun est alors très efficace à son époque, celui-ci alors adopté à grande échelle afin de lire automatiquement les chèques dans de nombreuses infrastructures aux états unis. A la fin des années 1990 celui-ci lisait près de 20 pourcent des chèques américains. Les avancées théoriques récentes relancent donc la machine, permettant d'aller bien au de-las des obstacles d'utilisation des réseaux de neurones. L'usage de réseaux de

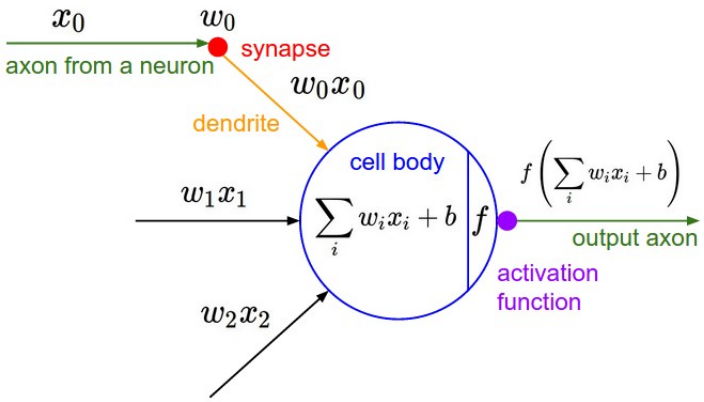
neurones profonds est désormais monnaie courante chez les géants du web. Des processeurs graphiques très puissants permettent alors de manipuler des données de plus en plus complexes : images, vidéos, son... Cependant il existe de nombreuses configurations de réseaux de neurones ayant des portées totalement différentes existent : allant de la brique permettant de traiter des données temporelles à celle permettant de relever les différentes abstractions de celles-ci. De nombreux paramétrages sont donc à effectuer lorsque l'on se lance dans la conception d'un réseau de neurones, et demande en général d'utiliser des astuces permettant d'optimiser le traitements de notre apprentissage, nous en faisons l'état dans la partie théorique de ce papier.

III. Principaux fondements théoriques

I. Le perceptron

Le perceptron comme expliqué précédemment est une simplification mathématique du fonctionnement d'un neurone. C'est en présentant des exemples à celui-ci que ses poids s'ajustent afin d'obtenir au fil de l'entraînement les sorties désirées. Une fonction d'activation est au préalablement définie afin de transformer la somme de nos entrées pondérées par nos poids. Celle-ci est typiquement une fonction sigmoïde.

Figure 6: *Perceptron (@Source : Andrej Karpathy)*



Formellement le processus d'apprentissage d'un neurone s'articule comme suis :

1. Initialiser les poids aléatoirement.
2. pour chaque exemple calculer la sortie ŷ produite par la fonction d'activation
3. Si la sortie obtenue est alors différente la sortie attendue : Ajuster les poids en augmentant leurs valeurs des poids afin de coller à sortie attendue si celui-ci est trop petit, ou le diminuer si la sortie obtenue est trop haute.
4. Répéter l'opération jusqu'à ce que le perceptron ne fasse plus d'erreurs

Mathématiquement la mise à jour des poids s'écrit ainsi :

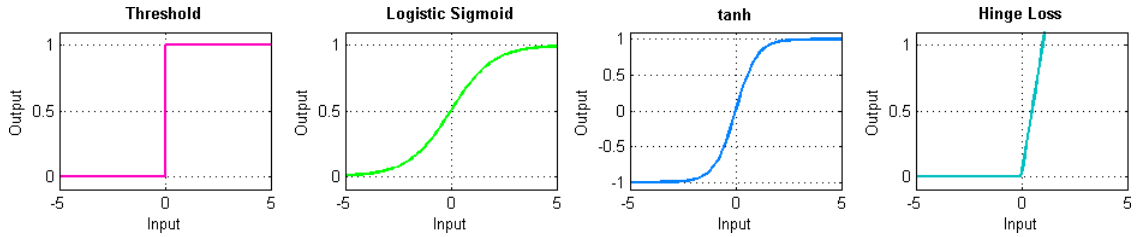
$$w_i = w_i(t) + (y_i^p - f(W'X^p))x_i^p \tag{1}$$

où

$$y = f(\sigma_{i=1}^{N} w_i x_i + w_0) = f(W'X) \tag{2}$$

II. Quelques fonctions d'activations couramment utilisées

Figure 7: *Aperçut de quelques fonctions d'activations (@Source : Adam Harley)*



La fonction d'activation est une fonction mathématique permettant aux neurones de transformer un signal reçut. Celle-ci peut profondément influencer la performance du réseau, il est donc important de bien choisir celle-ci.
La fonction d'activation présente dans les neurones d'entrée et de sortie est généralement la fonction identité. Mais dans le cas de la classification nous utiliserons la fonction softmax en sortie qui est définie entre 0 et 1 :

$$f(x_i) = \frac{exp(x_i)}{\sigma exp(x_i} \tag{3}$$

Le perceptron de Rosenblatt initialement défini contient la fonction sigmoïde :

$$f(x_i) = \frac{1}{1 + e^{-x_i}} \tag{4}$$

C'est une fonction de transfère dite molle dans le sens où seul les grandes valeurs positives de x tendront vers 1, et les forte valeurs négatives tendront vers -1. Celle-ci à la particularité d'avoir tendance à saturer le neurone. Il est donc préférable d'utiliser dans ce cas la fonction tanh(x) définie ainsi :

$$tanh(x_i) = \frac{e^{x_i} - e^{-x_i}}{e^{x_i} + e^{-x_i}} \tag{5}$$

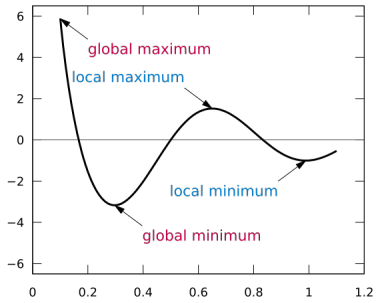
Enfin nous présentons la fonction d'activation appelée ReLU (rectified linear unit), devenue très populaire ces dernières années. Celle-ci ne soufre pas de la saturation, et rend en général plus rapide la convergence de l'algorithme de descente du gradient. Cependant cette fonction peut s'avérer fragile et peux ne plus avoir d'effets au fil des phases d'entraînements. En effet celle-ci peut tendre à bloquer la valeur de sortie à zéro, d'autres dérivées de cette fonction comme le ReLU faible, le ReLU paramétrique et aléatoire permettent de contrer ce problème.
La fonction ReLU :

$$ReLU(x_i) = max(0, x) \tag{6}$$

III. L'algorithme de descente du gradient

Afin d'entraîner un réseaux de neurones nous utilisons l'algorithme du SGD (stochastic gradient descent). Celui-ci est alors considéré comme résolvant un problème d'optimisation, en effet nous tentons ici de minimiser l'erreur produite par notre modèle. Lorsque notre modèle est plus complexe qu'un simple neurones le système à plus de mal à se stabiliser.

Figure 8: *Minimisation de l'erreur via l'algorithme de descente du gradient (@Source : Ivan Vasilev)*



La fonction objectif (typiquement le critère des moindre carré) est alors plus dure à minimiser : comportant un grand nombre de minimum locaux. Critère des moindres carrés :

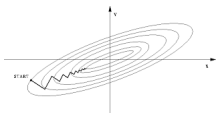
$$E(w) = \frac{1}{2}\sigma_{w\epsilon\omega}(y(w) - \hat{y}(w))^2 \tag{7}$$

La mise à jour des poids est alors définie comme suis :

$$w_i \leftarrow w_i - \eta\frac{(w)}{w_i} \tag{8}$$

Le choix du taux d'apprentissage défini alors la taille du pas effectué à chaque itération. Il doit être ni trop grand, au risque de ne pas trouver le minimum de la fonction et ni trop petit afin de ne pas avoir à faire à un temps de calcul trop long.

Figure 9: *Représentation de l'algorithme de descente du gradient (@Source : Trond Hjorteland)*



Cette technique s'applique alors en présentant un à un de nouveau exemples à notre réseau, ajustant alors pas à pas les poids des neurones. Elle a l'avantage de mettre à jour facilement notre réseau lors de l'ajout d'un nouvel individu et permet alors d'utiliser un modèle pré-entraîné.
Cependant lorsque notre réseau est dit profond le SGD a du mal à converger vers un minimum, les scientifiques travaillant sur le sujet ont donc du mettre en places des techniques améliorant le temps d'apprentissage.

IV. Optimisations de la descente de gradient

1. Momentum : Cette méthode aide à accélérer la phase de descente de gradient, comme son nom l'indique nous rajoutons dans notre équation de mise à jour des poids une "mémoire" : une part de l'update précédent. L'état de l'art fixe ce taux à 0.9. Nous obtenons en utilisant cette technique une diminution des oscillations et une convergence plus rapide.

2. Gradient Accéléré de Nesterov : Cette méthode basée sur le Momentum vise à approximer la position future de nos paramètres. Celle-ci corrige la direction de chaque saut effectué à l'aide du momentum, permettant alors d'éviter d'apprendre trop rapidement et améliore ainsi la réactivité de l'algorithme.

Nous allons désormais présenter quelques algorithmes dynamiques qui adaptent l'update de chaque paramètre afin d'augmenter ou diminuer la mise à jour de ceux-ci en fonction de leur importances.

1. Adagrad : cette méthode adapte le taux d'apprentissage en fonction des autres paramètres en augmentant celui-ci dans le cas où ceux-ci sont peu fréquents et en le diminuant dans le cas inverse. Cette amélioration est bien adaptée aux données éparses, rendant ainsi le SGD plus robuste. Le taux d'apprentissage est fixé à 0.01 à l'initialisation de notre algorithme. Cette méthode élimine le besoin de modifier pas à pas le taux d'apprentissage, mais sa principale faiblesse réside dans l'augmentation de la matrice par laquelle l'on divise notre taux d'apprentissage qui grandit au fil des itérations...

2. Adadelta : Cette algorithme est une amélioration du précédant en réduisant la matrice des gradient à la moyenne des erreurs au carrés des gradients précédents.

3. RMSprop : est une variante du AdaDelta non publiée proposée par Geoff Hinton.

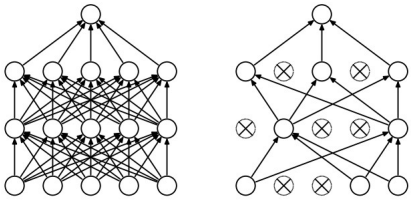
4. Adam : Adaptive Moment Estimation, est une autre méthode visant à estimer le taux d'apprentissage à chaque itération, intégrant cette fois-ci la notion de variance. Celui-ci est alors très performant.

Les méthodes adaptatives sont en soit très efficaces car elles convergent très vite, le Momentum et le NAG quant à eux ont tendances à partir très loin du point optimal avant de réaliser une correction de la direction vers la solution. Si vos données sont assez éparses les méthodes adaptatives sont un choix optimal, d'autant plus qu'elles nous permettent de ne pas avoir à se pencher sur une initialisation du taux d'apprentissage.

V. Techniques d'amélioration de l'apprentissage

1. Parallélisation et distribution du SGD : Lorsque nos devons travailler sur de larges dataset distribués sur des clusters, nous devons aussi penser à paralléliser le calcul du gradient stochastique. Mais celui-ci est par essence séquentiel, se dirigeant pas à pas vers un minimum. Il existe cependant des techniques permettant de lancer un SGD de manière asynchrone, ceci est plus rapide mais tend parfois à une faible convergence. Les algorithmes Hogwild!, Downpour, Elastic Averaging SGD permettent cependant de réaliser ces calculs distribués avec de bonnes performances.

2. Le drop out : le drop-out permet de parer au problème d'over-fitting en "sautant" aléatoirement un neurone selon une probabilité p. Celle-ci est définie par l'état de l'art à 0.2.

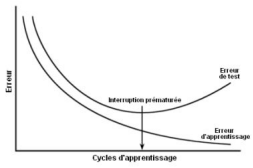


(a) Standard Neural Net (b) After applying dropout.

3. Le mini batch : la technique du mini batch vise à mettre à jour les poids tout les n exemples d'apprentissage. Celle-ci permet de réduire la variabilité de nos paramètres de mise à jour en offrant alors une convergence plus stable. Celle-ci est alors fixée entre 50 et 256, mais cela varie bien entendu du nombre de données disponibles.

4. La normalisation du batch : afin de faciliter la l'apprentissage il est proposé de normaliser nos entrées, ceci nous permet alors de traiter plus facilement les variables ayant différentes échelles, et nous autorise à utiliser un plus haut taux d'apprentissage.

5. Le gradient bruité : l'ajout d'un bruit lors de la mise à jour des gradients, permet de rendre un réseaux plus robuste à l'initialisation de nos paramètres. Cette technique permet alors d'échapper plus facilement à un minimum local.



6. L'arrêt prématuré : Afin de ne pas rester bloqué dans un minimum local, L'arrêt de l'algorithme peut alors être paramétré, diminuant ainsi le temps de calcul. Dans ce cas nous devons définir la patience de l'algorithme : le nombre d'itérations non significatives (pas d'amélioration de la perte) à partir de la quelle nous arrêtons le processus.

7. Initialisation des poids : il existe plusieurs manières d'initialiser nos poids : la plus simple est de les initialiser aléatoirement en divisant par la variance totale de ces aléas. L'état de l'art propose de multiplier les poids aléatoirement définis par la racine carré de 2 divisée par la variance des ces aléas.

8. Régularisations : préviens l'overfitting , typiquement cette technique vise à utiliser une fonction de pénalisation sur nos poids (régularisation L2 est souvent plus performante que la régularisation L1).

9. Augmentation des données : augmenter les données permet aussi d'améliorer la capacité de généralisation de notre modèle, le but est donc ici de passer en entrée de multiples versions de nos données données initiales : pour des images par example nous changerons les couleurs, opérerons des rotations...

10. Méthode graphiques : tracer la courbe de l'évolution de la fonction de perte est aussi un bon indicateur de la performance de votre algorithme. Permettant ainsi de juger de la pertinence de votre taux d'apprentissage et ainsi observer un possible sur-apprentissage, du nombre de couches utilisées...

11. Méthode ensemblistes : combiner des modèles peut être aussi une manière d'améliorer la qualité de prévision. Cela permet en soit de généraliser votre problème.

IV. Quatre architectures à retenir

Il est temps d'entrer dans le vif du sujet, de nombreuses architectures ont dérivées du modèle MLP, n'ayant pas la même fonction. L'intérêt de l'approche des réseaux de neurones est le fait que ces architectures peuvent être empilées l'une à la suite de l'autre. Nous présenterons les quatre architectures les plus utilisées, il en existe encore de nombreuses autres (réseaux siamois, arbres bi-directionnels...)

Il existe trois types d'architectures :

1. Les modèles forward allant de l'entrée vers une sortie (ex: MLP,ConvNet)
2. Les modèles à Feed-back, visant à générer une entrée (ex: Stacked Sparse Coding, Deconvolutional Nets)
3. Et des réseaux birirectionnels (ex : DBM, Stacked Auto-encodeurs)

Ces modèles peuvent alors être entraînés de trois manières différentes :

1. De manière supervisée, nous connaissons les sorties attendues.
2. De manière non-supervisée, nous ne possédons dans ce cas aucune connaissances préalable concernant nos données.
3. Ainsi que l'apprentissage par renforcement.

Nous présenterons ici des modèles supervisés et non supervisés :les machines restreintes de Boltzman, les auto-encoders, les réseaux récurrents et les réseaux convulsifs, ce qui couvre déjà un grand nombre d'applications !

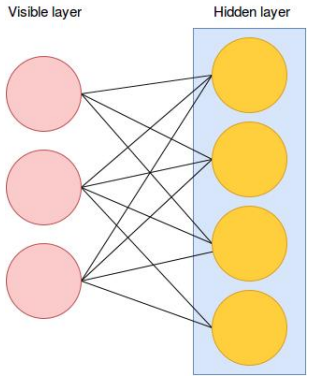
Mais lequel de ces modèles utiliser et dans quel cas ? Lorsque vos données d'entrées ne sont pas labellisées il faudra utiliser dans un premier temps un réseau permettant d'extraire des caractéristiques, et de reconnaître des patterns dans celles-ci. Dans le cas où nos donnés sont linéairement séparables nous nous tournerons vers une ACP, analyse des composantes principales, mais dans le cas de données plus complexes nous devrons faire appel à une machine de Boltzman ou un auto-encodeur. Ceci peux nous permettre de réduire la dimensionnalité de nos données.

Dans le cas où nous souhaitons réaliser un modèle supervisé, nous citerons l'utilisation des réseaux récurrents pour la gestion des données distribuées dans le temps. Nous pourrons ainsi travailler sur des textes : analyse de sentiments, analyse de la parole, réaliser des prévision financières, l'embarquer dans un système anti-collision automobile... Dans le cas d'images nous utiliserons en priorité les réseaux convolutionnels (Conv Net) qui pourront notamment être très utiles lorsque l'on essayera de faire de la reconnaissance d'objets...

I. Machine restreinte de Boltzman superficielle et profonde

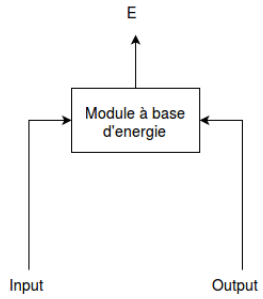
Appelée Harmonium par son inventeur Paul Smolensky, la Machine de Boltzman est une structure à deux couches, dont l'une est cachée, c'est un modèle génératif stochastique qui peut donc apprendre la distribution d'un jeux de données. Les liens entre la couche visible et la couche cachée ne sont pas dirigés, contrairement au MLP, permettant un entraînement dans les deux sens. Nous noterons que lorsque les neurones d'une couche sont interconnectés le modèle n'est plus dit restreint.

Figure 10: *Restrict Boltzman Machine*



Les données d'entrées doivent être binaires, les unités cachées contiennent elles une fonction d'activation renvoyant une valeur comprise entre 0 et 1.

Figure 11: *Energy Based Module*



Ce modèle est basé sur l'énergie, c'est à dire que l'on que essaye de mesurer la compatibilité d'une observation avec une autre qui doit être inférée. Cette valeur scalaire est calculée comme suit :

$$E(v,h) = -\sum_i a_i vi - \sum_j b_j h_j - \sum_i \sum_j v_i w_{i,j} h_j \tag{9}$$

où h est l'unité cachée, v l'unité visible et dont la matrice de poids est définie ainsi : $W = (w_{i,j})$

Cela qui en notation matricielle donne :

$$E(v,h) = -a^T v - b^T h - vTWh \tag{10}$$

L'énergie peux être transformée en probabilité sous forme de distribution de Gibbs:

$$P(v,h) = e^{-E(v,h)}\frac{1}{Z} \tag{11}$$

où Z est une fonction dite de partition, elle a pour but de normaliser nos probabilités afin leur somme soit égale à 1 :

$$Z = \int_{y\epsilon\varepsilon} e^{-E(v,h)} \tag{12}$$

De façon similaire la probabilité marginale P(v) de l'entrée est calculée comme suis :

$$P(v) = \sum_h e^{-E(v,h)}\frac{1}{Z} \tag{13}$$

La fonction d'activation de ces neurones est alors une fonction logistique, celle-ci s'écrit comme suis pour la couche cachée h, et m est le nombre d'unités cachées :

$$P(h_j = 1|v) = \sigma(b_j + \sum_{i=1}^{m} w_{i,j} vi) \tag{14}$$

et comme suis du coté visible :

$$P(v_i = 1|h) = \sigma(a_j + \sum_{j=1}^{m} w_{i,j} hj) \tag{15}$$

L'unité visible peut aussi être multimodale, dans ce cas nous remplaçons la fonction logistique par la fonction softmax :

$$P(v_i^k = 1|h) = \frac{exp(a_i^k + \sum_j W_{i,j}^k h_j)}{\sum_{k=1}^{K} exp(a_i^k + \sum_j W_{i,j}^k h_j)} \tag{16}$$

où k est le nombre de labels possibles.
Ce modèle est utilisé dans des problématiques de topic modeling ainsi que dans les systèmes de recommandations.

L'algorithme d'entraînement de ce modèle est basé sur la divergence contrastive (CD-1):

1. Phase positive :

   l'on fait passer une valeur vi dans couche d'entrée, vi est propagé à la couche cachée, le résultat de la fonction d'activation de h est alors calculé.

2. Phase négative :

   l'on retourne h à la couche visible, et l'on calcule ensuite la valeur de la fonction d'activation de vi', puis l'on retransmet vi' à h.

3. Mise à jour des poids :

$$w(t+1) = w(t) + \eta(vh^t - v'h'^t) \tag{17}$$

   avec η taux d'apprentissage, et v,v',h et h' les résultats des fonctions d'activation et enfin w les poids.

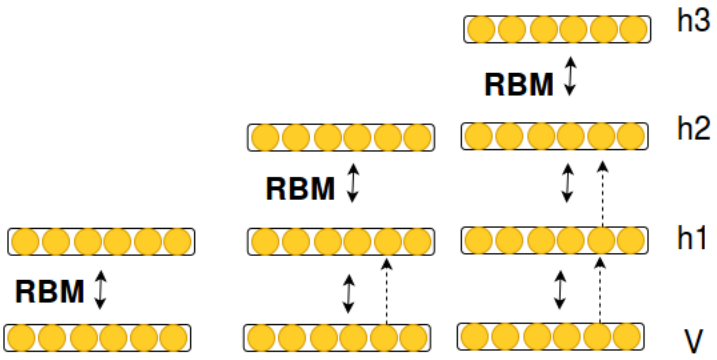
L'idée de cette méthode est de refléter une représentation de v : h permettant de générer un donnée se rapprochant au plus près d'une réalité. Tant que la valeur retournée n'est pas assez proche de la réalité nous avançons d'un pas t.

Le modèle profond de ce modèle est appelé Deep Belief Network (réseaux profonds à croyances). Dans ce cas la couche cachée du RBM t deviens alors la couche visible du RBM t+1. Cette architecture très gourmande s'entraîne à l'aide de l'algorithme de divergence constrastive de cette manière :

1. Entraîner le premier RBM t=1 en utilisant l'algorithme CD-1 présenté précédement
2. Entraîner le second RBM t=2, tant que la valeur visible de t=2 est la valeur de la couche cachée t=1, entraîner la valeur d'entrée par la valeur de la couche visible de t=1. Laquelle est propagée par la suite à la couche cachée t=1. Les données obtenues servent alors à entraîner de nouveau couche cachée de t=2.
3. Répéter la procédure précédente à toute les couches

Tout comme l'auto-encodeur empilé, notre réseaux peut être étendu en connectant le réseaux à un RBM à une ou plusieurs couches interconnectées comme sur ce shéma :
L'on entraîne alors ces couches via back-propagation.

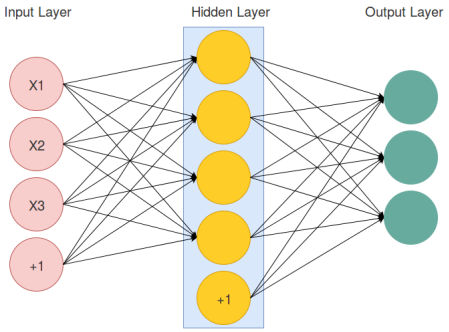
Figure 12: *Deep Belief Network*



II. Auto-encoder et son extension profonde

L'auto-encoder est un réseau feed-forward dont le but est d'apprendre une image abstraite de nos données. Il est entraîné afin de recréer nos données de manière compressée en sortie. Le passage dans notre couche cachée est donc destiné à encoder nos données et à les recomposer en sortie. Contrairement aux MLP Ce réseau possède dans chacune de ces couche d'un biais.

Figure 13: *Stacked Auto-encoder*



L'on peut donc voir cette transformation comme une projection dans l'espace X à l'espace F projeté de nouveau dans X :

$$\phi : X \to F$$
$$\psi : F \to X$$
$$\arg\min \|X - ()\psi \circ \phi)\|^2$$

basiquement l'on note x l'input et son image z :

$$\text{x } \epsilon R^d$$
$$\text{z } \epsilon R^p$$

avec :

$$\text{z}{=}\sigma_1(Wx + b)$$
$$\text{x}{=}\sigma_2(W'x + b')$$

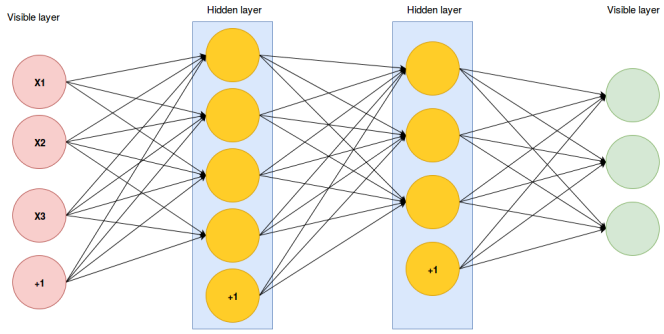
La fonction objectif de ce modèle est alors définie généralement comme une minimisation

de l'erreur quadratique.
Nous noterons que si la couche cachée est plus large que la couche d'entrée ceci aura pour effet d'apprendre la fonction identité...
De nombreuses variations existent ayant pour but de prévenir ce problème : l'auto-encoder avec débruitage (Denoising Auto-encoder), partant du principe qu'un modèle robuste est capable de prendre en entré des données corrompues. Nous projetons donc x_1 dans un nouvel espace stochastique, les données en résultant seront alors nommées x_2. Nous calculerons la fonction objectif de cette manière : $\mathcal{L}(x_1, x_2')$ à la place de $\mathcal{L}(x_1, x_1')$.
D'autres modèles comme l'auto-encodeur éparse, variationnel et la troncature de la décomposition de ma valeur singulière permettent aussi de répondre à cette problématique.
L'entraînement de ce modèle se déroule comme ceci pour chaque input x :

1. propager la valeur de la fonction d'activation de x à la couche cachée, nous obtenons alors la valeur x'.
2. Mesurer la déviation de x' comparée à x.
3. Rétropropagager l'erreur et mettre à jour les poids

Nous pouvons aussi utiliser des variantes de la rétro propagation comme la méthode du gradient conjugué.
Ce modèle peut être aussi dans sa version profonde, appelée pile d'auto-encodeurs.

Figure 14: *Stacked Auto-encoder*

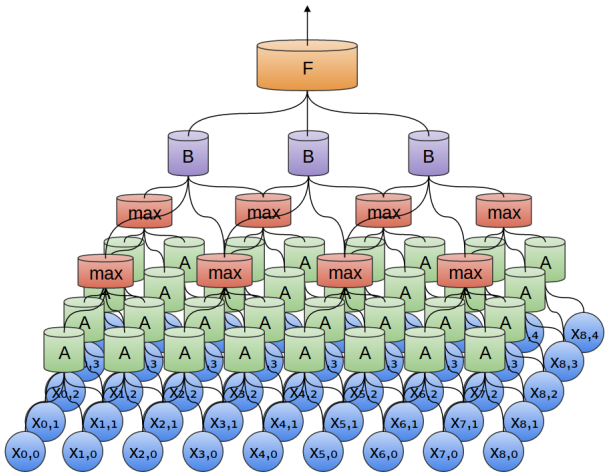


Pour entraîner une pile d'auto-encodeurs nous utiliserons l'algorithme CD-1 de Hinton, en utilisant la micropropagation du gradient (mini-batch).

III. Réseaux convolutionnels

Les réseaux convolutionnels sont des réseaux feed-forward. Inspiré du cortex visuel animal ce modèle est utilisé pour ne pas avoir à faire appel à des phases de pré-processing. Vu comme une suite d'étapes de normalisation, filtrage, d'activation non linéaire, et d'agrégation, celui-ci permet de capturer une représentation abstraire et hiérarchique de nos données. Ayant de nombreuses applications en traitement de l'image, celui-ci peut aussi être utilisé dans des techniques de reconnaissance vocale et dans de nombreux autres domaines.

Figure 15: *Exemple simple de Réseaux Convultionnel 3D @Source : Christopher Olah*



Les réseaux convolutionnels peuvent être utilisés sur des données à dimension différentes :

1. 1 dimension : classification de texte, reconnaissance vocale, prédiction de séries temporelles
2. 2 dimensions : détection d'objets, reconnaissance faciale, modèle spatial...
3. 3 dimensions : reconnaissance vidéo, contextualisation de scène, prédiction de mouvements, analyse biomédicale.

Le modèle apprend alors à travers de nombreuses répliques de chaque neurones, à travers de nombreuses étapes de transformations l'importance du GPU prends donc ici tout son sens. Nous noterons que nous pouvons voir ce modèle comme un module d'une architecture plus complexe tout comme les Deep Belief Network et Stacked Auto-encoders.
Le modèle est alors composé de plusieurs couches :

1. Couche de convolution (A): cette couche effectue un filtrage. Ces neurones cachés sont organisés en trois dimensions, chaque neurone de la couche cachée est lors connecté localement à une petite région de l'entrée.

$$y_{n,m} = \sigma(W_x + b) \tag{18}$$

Figure 16: *ConvNet 2D @Source : Christopher Olah*

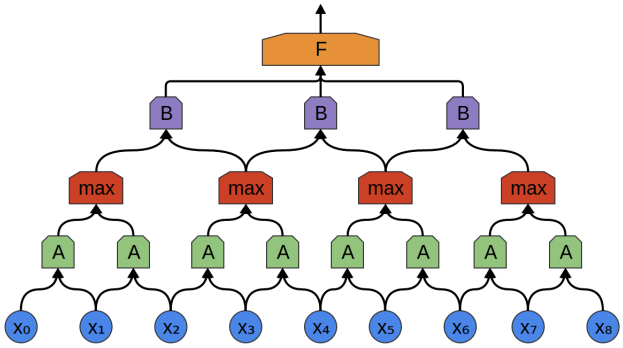
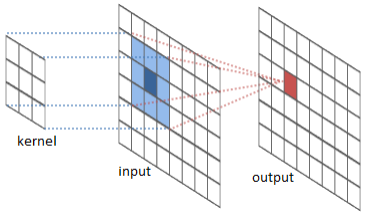
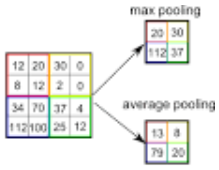


Figure 17: *Phase de convolution @Source : Wikipédia*



2. Couche d'agrégation (max) : cette couche permet alors de descendre en dimension via une fonction d'activation, la plus utilisée est le max pooling, qui récupère le maximum d'une zone définie.

Figure 18: *Max pooling @Source : Wikipédia*



3. Couche rectification linéaire (B): régie par la fonction d'activation ReLu, augmentant ainsi les propriétés non linéaire des données d'entrées : f(x)=max(0,x)

4. Couche entièrement connectée (F): placé en sortie de plusieurs couches précédemment annoncées, cette couche récupère les données de sortie de la dernière couche.

5. Couche de perte : cette couche pénalise alors les résultats obtenus en fonction des résultats attendus. La fonction soft-max est utilisée en classification, la fonction sigmoïde est utilisée dans le cas de prédiction de probabilités indépendantes et la fonction de perte euclidienne dans les cas voués à une régression.

Comment définir les paramètres du modèle ?

1. Le nombre de filtres : Dépends du nombre d'exemples fournis en entrée ainsi que de la complexité des données.

2. La taille du filtre : cela dépend encore de nos entrées, mais dans le cas d'images naturelles, elle est fixée entre 10x10 et 20x20. Le challenge est ici de définir le niveau d'abstraction que l'on souhaite capturer.

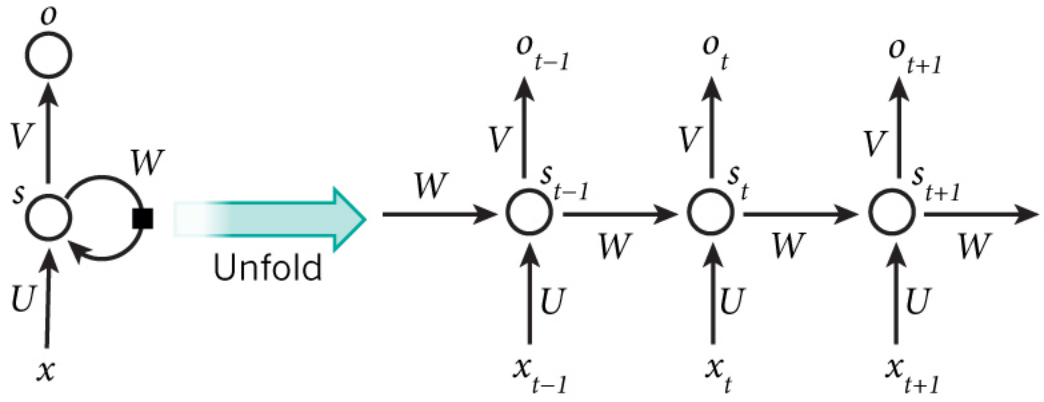
3. La forme du la sortie de l'agrégat : typiquement fixée à 2x2 elle peut être amenée à 4x4 dans les cas où les données possèdent de forte dimensionnalité, plus l'on augmente ses dimensions plus l'on réduit alors la dimension de notre signal.

Ce modèle est alors entraîné en utilisant l'algorithme feed-forward en prenant soin d'utiliser des techniques permettant d'optimiser le traitements :la normalisation des variables d'entrées,le drop-out afin de parer au problème d'overfitting, la régularisation des poids d'entrée afin d'améliorer la performance de l'apprentissage.

IV. Les réseaux récurrents

Les réseaux récurrents sont une famille de modèles permettant de prévoir des séquences temporelles. Ce sont des réseaux feed-forward, dans laquelle passent un à un des données triés par ordre d'apparition. Ce type de réseaux est sujet au problème de disparition de gradient au fil du temps lorsque son architecture est profonde. La longueur de la fenêtre d'entrée peut alors être vu comme une couche, la succession de couches de RNN conduit imparablement à ce problème et ne permet pas de capturer les dépendances à long termes. Il existe de nombreux modèles permettant de parer cette embûche dont le LSTM et GRU, aujourd'hui couramment utilisés par la communauté de chercheurs. D'autres techniques comme le Gradient clipping, et le Steeper Gates peuvent en augmenter la performance. Le domaine est prometteur notamment dans des taches de traitement du langage naturel.
Nous comprenons lorsque nous déplions la récurrence l'analogie de la taille des séquences d'entrée avec les couches cachés. Chacune des entrées de la séquence est dépendante de la précédente, offre une sorte de mémoire, comme nous pouvons le voir sur ce schéma.

Figure 19: *Exemple de réseaux récurrent @Source : Denny Britz*



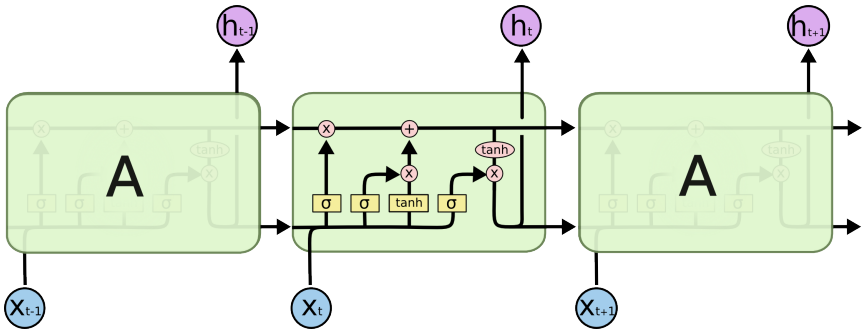
x_t est alors l'entrée au temps t.
s_t est l'état de la couche cachée à l'instant t.
C'est cette fameuse "mémoire" que nous attribuons au réseau, celle-ci est calculée récursivement comme suis :

$$s_t = f(Ux_t + W_{s_{t-1}}) \quad (19)$$

où f(x) est typiquement une fonction d'activation tel que le tanh ou le ReLU. L'état du premier $s_{t=0}$ est alors généralement initialisé à 0.
Contrairement aux modèles traditionnels celui-ci partage ces paramètres (U,V et W) à travers les différents étapes d'apprentissages. Nous réalisons donc la même transformation à travers le réseaux mais avec des entrées différentes.
Le LSTM (Long Short Term Memory) est modèle décrit en 1997 par Sepp Hochreiter et Jurgen Schmidhuber. Ce module n'est pas très éloigné des RNN classiques, il utilise seulement une autre procedure afin de calculer l'état de la couche cachée. Ce système est

conçu comme une boite noire dont la structure interne choisi ou non de garder en mémoire les valeurs d'entrées, nous pouvons en apercevoir le contenu sur ce schéma:

Figure 20: *Schéma du LSTM @Source : Christopher Olah*



La clef du LSTM est le flux horizontale le traversant, ce flux à la possibilité d'ajouter ou supprimer des informations via des "ponts" ou gates. Cette quantité d'information à retenir est alors définie par les fonction sigmoïdes présentes sur les trois ponts du LSTM, lorsque la valeur de sortie de cette fonction est égale à 0, aucune information n'est prise en compte, lorsqu'elle est égale à 1 toute l'information est prise en compte.
Lors du passage dans le premier pont du LSTM de l'entrée xt et de l'output de la couche précédentes ht-1, notre module va donc choisir de retenir où non ces informations cette couche est appelée "forget gate layer". La sortie de celle-ci s'écrit comme suis :

$$f(t) = \sigma(W_f.[h_{t-1}, x_t] + b_f) \tag{20}$$

Une fois ceci effectué le pont suivant appelé "input gate layer" va définir quelle nouvelle information va être retenue et créer un nouveau vecteur de candidat possible pour l'état de la cellule Ct. La sortie du sigmoïde est notée ainsi :

$$i_t = \sigma(W_i.[] + b_i) \tag{21}$$

et celle de la fonction tanh comme cela :

$$\hat{C}_t = tanh(W_C.[h_{t-1,x_t}] + b_C) \tag{22}$$

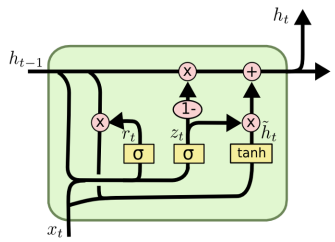
Après cela il est ensuite temps de décider quel état de Ct nous allons sélectionner. Nous repassons donc dans le flux horizontal en utilisant les valeurs précédemment calculées. Nous multiplions ainsi la valeur de l'état Ct-1 par f(t) (permettant donc d'oublier ou pas un certain taux d'informations), puis nous ajoutons le produit de it et $\hat{C}$t permettant ainsi d'obtenir la nouvelle valeur de Ct.
Enfin nous devons bien entendu renvoyer notre prédiction ht au prochain pas de temps. Nous repassons donc l'output ht-1 dans une sigmoïde, passons l'état Ct calculée précédemment et multiplions ces deux valeurs afin d'en extraire notre sortie.

$$h_t = \sigma(W_0[h_{t-1}, x_t] + b_0) * tanh(C_t) \tag{23}$$

Nous venons ainsi de présenter le processus entraîné à travers un LSTM "classique", il en existe d'autres se basant sur le même modèle en y apportant quelques modifications

mineurs, nous en présentons ainsi le GRU (Gated Recurrent Unit) introduit par Cho et al en 2014.

Figure 21: *Schéma du GRU @Source : Christopher Olah*



Le GRU, peut être vu comme une sorte de simplification du LSTM. Ce modèle possède alors deux ponts un "forget gate" et un "reset gate". Intuitivement le reset get détermine comment combiner la nouvelle entrée avec la dernière valeur en mémoire. Et "l'update gate" définit à quelle hauteur la valeur en mémoire doit être conservée. Celui-ci ne garde pas en mémoire l'état de la cellule précédente. Nous pouvons définir ce modèle via ces 4 équations :

$$\begin{aligned} \mathrm{z}&=\sigma(x_t U^z + s_{t-1} W^z) \\ \mathrm{r}&=\sigma(x_t U^r + s_{t-1} W^r) \\ \mathrm{h}&=\tanh(\mathrm{x}_t U^h + (s_{t-1} \circ r) W^h) \\ \mathrm{s}_t &= (1-z) \circ h + z \circ s_{t-1} \end{aligned} \tag{24}$$

Quel modèle choisir ? L'état de l'art ne définit pas clairement de réel vainqueur. Le modèle GRU plus récent semble cependant un peu plus rapide du fait de son moins grand nombre de paramètres. Et semble avoir besoin de moins de données afin d'avoir de bonnes capacités de généralisation. De son coté le LSTM semble plus performant lorsque l'on possède un grand set de données.
L'entraînement de ce type de structure ce fait alors de manière similaire aux autre réseaux de neurones via backpropagation, cependant comme chacune des entrées est dépendantes de la précédente une légère modification est apportée (Backpropagation Through Time).
Afin d'utiliser notre LSTM nous procédons à une modification de notre jeux de données. Nous associons à chaque valeur de x_t les n valeurs passées de x. De cette manière nous faisons entrer pour chaque x, un vecteur composé de n+1 valeurs afin de prédire yt. Puisque nous utilisons une fenêtre de longueur n contenant les n valeurs passées de x_t : x_{t-1}, x_{t-2}... x_{t-n}, nous réalisons une propagation du gradient à travers notre le vecteur d'entré. La somme de ces n propagations est alors moyennée, comme si chacune d'elles étaient égales. Nous transmettons alors cette valeur à la prochaine séquence attribuée à x_{t+1}.

V. Supports de développement

De nombreux framework existent actuellement sur le marché, nombre d'entre eux sont open-source. La plupart ont l'avantage d'être à jour au niveau de l'état de l'art. Nous citerons principalement Theano une librairie python permettant de définir tout le cycle de d'apprentissage optimisation de nos modèles, ce framework sert alors de bases à de nombreuses autres librairies se voulant plus simple à utiliser et optant pour une programmation plus modulaire. Nous citerons Lasagne, Blocks et Keras (dont nous ferons usage dans la section suivante). D'autres frameworks sont alors disponibles dans plusieurs langages comme Torch (ayant tout comme Keras une approche modulaire) tout comme DeepLearning4j. Enfin en tant qu'utilisateur de R nous noterons l'apparition d'un petit nouveau nommé H2O.

VI. Tutoriel : la neige du lac d'Alta

Figure 22: *Lac Alta au Canada @Source : Alexis Chavetnoir*

Ce tutoriel a pour but de vous introduire à l'écriture de programme de Deep Learning. Nous nous pencherons ici sur le cas des réseaux récurrents. Ce choix est justifié dans un premier temps par le fait que ce modèle est très présent dans des traitements de données textuelles, données que j'aime manipuler. Ceux-ci semblent aussi être une alternatives aux méthodes économétriques dans le cas de l'étude de séries temporelles lorsque les données semblent posséder propriétés non stationnaires ayant des relations de dépendances à long termes, comme celles que je manipule actuellement en stage. Ce modèle demande de plus moins de puissance pour être entraîné, ce qui vous permettra de réaliser vos premières inférences de manière confortable sur votre PC personnel. Afin de mettre en place ce tutoriel il nous fallait dans un premier temps disposer de données sur lesquels nous pencher. J'ai donc été récupérer sur le site de l'institut canadien du climat (équivalent du météo france canadien) des données de mesures quotidienne de précipitation de neige entre 2007 et 2016 à proximité du Lac Alta. Ces données relativement simples nous permettrons de comprendre le fonctionnement des LSTM précédemment introduits. Nous réaliserons plusieurs modèles que nous tenterons d'améliorer.

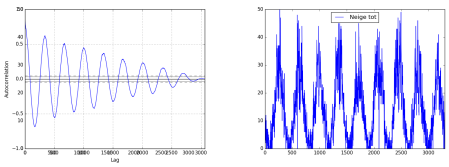
I. Import des données

Commençons par importer nos données

```
import numpy as np
import pandas as pd
from pandas.tools.plotting import autocorrelation_plot
import matplotlib.pyplot as plt

#Load the data
df=pd.DataFrame(pd.read_csv('/home/USER/.../neige.csv', sep=','))
#Plot the time series
df.dropna().plot()
#Delete NA
df2=df.dropna()
#Visualise the correlaction trough time
autocorrelation_plot(df2)
```

Nous obtenons ainsi deux graphiques : le graphe d'autocorrelation largement utilisé en économétrie qui nous permet d'entrevoir les dépendances temporelles de nos données, ainsi que l'évolution de notre série à travers le temps :



II. Installation Keras et prise en mains

Afin d'installer Keras il vous suffira de posséder la dernière version de python et de pip. Deux commandes sont alors requises depuis votre terminal :

1. pip install theano
2. pip install keras

Vous disposerez de cette manière de Théano ainsi que de Keras qui se veux être une surcouche de la première. Comme introduit précédemment cette librairie se veut modulaire. Ainsi en utilisant la simplicité de codage du paradigme de la programmation orientée composants nous pouvons construire de manière très intuitive un réseau profond.

III. Préparation des données

Définition de la fenêtre de nos données et split training/test

```
def _load_data(data, n_prev =7):
    """
    data should be pd.DataFrame()
    """

    docX, docY = [], []
    for i in range(len(data)-n_prev):
        docX.append(data.iloc[i:i+n_prev].as_matrix())
        docY.append(data.iloc[i+n_prev].as_matrix())
    alsX = np.array(docX)
    alsY = np.array(docY)

    return alsX, alsY

def train_test_split(df, test_size=0.3):
    """
    This just splits data to training and testing parts
    """
    ntrn = int(len(df) * (1 - test_size))

    X_train, y_train = _load_data(df.iloc[0:ntrn])
    X_test, y_test = _load_data(df.iloc[ntrn:])

    return (X_train, y_train), (X_test, y_test)
```

IV. Créations des fonctions liés à l'apprentissage

Nous devons ensuite définir notre procédure d'entraînement, phase délicate puisque celle-ci aura un gros impact sur la durée de l'entraînement et sa performance prédictive. Nous entrerons donc dans cette fonction le modèle préalablement construit, ainsi que nos jeux de données d'entraînement. Nous utiliserons le critère adam d'optimisation, présenté précédemment, forçons le module à enregistrer la valeur de la perte de chaque itérations. Nous avons donc ici trois paramètres à définir : la taille du batch, le nombre d'itérations, ainsi que le fait ou non d'arrêter prématurément notre apprentissage via un early stopping.

```
def train_model(model, X_train, Y_train, batch_size, nb_epoch,early_stop):

    #DEFINING OPTIMISER CRITERAS
    model.compile(loss="mean_squared_error", optimizer="adam")

    #FORCE THE MODULE TO GET LOSS HISTORY
    class TrainingHistory(Callback):
        def on_train_begin(self, logs={}):
            self.losses = []
            self.predictions = []
            self.i = 0
            self.save_every = 50

        def on_batch_end(self, batch, logs={}):
            self.losses.append(logs.get('loss'))
            self.i += 1
            if self.i % self.save_every == 0:
                pred = model.predict(X_train)
                self.predictions.append(pred)
    #Create a new instance of history
    history = TrainingHistory()

    #IF we want to early stop ...
    if early_stop==True:
        earlyStopping=EarlyStopping(monitor='val_loss',
        patience=50, verbose=0, mode='auto')
        model.fit(X_train, y_train, batch_size=batch_size,
         nb_epoch=nb_epoch,validation_split=0.05,
         callbacks=[history,earlyStopping])
    # or not ...
    else:
        model.fit(X_train, y_train, batch_size=batch_size,
        nb_epoch=nb_epoch,validation_split=0.05,callbacks=[history])

    return (history)
```

V. Visualisation des résultats

Afin de visualiser les résultats créons deux fonctions permettant de tracer premièrement les prévisions obtenues et celles désirées.

```
def draw_results(predicted,y):
   pd.concat([pd.DataFrame(predicted),pd.DataFrame(y)],
    axis=1).plot()
```

Nous codons ensuite une fonction nous permettant de visualiser le comportement de la fonction de coût au fil du temps. Cette fonction prendra en paramètre l'historique définis à la sortie de l'entraînement :

```
def draw_loss(history):
    plt.figure(figsize=(6, 3))
    plt.plot(history.losses)
    plt.ylabel('error')
    plt.xlabel('iteration')
    plt.title('training error')
    plt.show()
```

VI. Simple LSTM

Commençons d'abord par construire un LSTM très simple, composé simplement d'un module LSTM ainsi que d'une couche de sortie neutre.

```
def make_LSTM_network1(in_out_neurons,hidden_neurons):
    model = Sequential()
    model.add(LSTM(hidden_neurons, input_dim=in_out_neurons,
     forget_bias_init='one', activation='tanh',
     inner_activation='sigmoid', return_sequences=False) )
    model.add(Dense(in_out_neurons,
    input_dim=hidden_neurons))
    return model
```

Nous paramétrons naïvement à titre d'essai le nombre de couches cachés à 30, la taille du batch à 100 et réalisons notre entraînement sur 200 itérations.

```
#Define parameters
in_out_neurons=1
hidden_neurons = 30
batch_size=100
nb_epoch=200
model0= make_LSTM_network1(in_out_neurons,hidden_neurons)

#Split data-set
(X_train, y_train), (X_test, y_test) = train_test_split(df2)

######## TRAINING STEP
t0 = time.clock()
history=train_model(model0, X_train, y_train,
 batch_size, nb_epoch,early_stop=False)
print ('TIME :'+str(time.clock()-t0))
########

##### PREDICT VALUES
predicted = model0.predict(X_test)

#####  COMPUTE RSME
rmse = np.sqrt(((predicted - y_test) ** 2).mean(axis=0))
print('RSME :' +str(rmse))

### PLOTING RESULTS
pd.DataFrame(y_test).plot()
draw_results(predicted,y_test)
draw_loss(history)
```

Combien de temps met ce réseau très simple à s'entraîner sur votre PC ? Avec mon ordina-

teur à 4 coeurs de 1.8 GHz, j'obtiens un temps d'exécution de 180 secondes. Cela n'est pas rien ! A combien est votre RMSE ? Le modèle semble-t-il bien apprendre de nos données ?

Recommencez l'entraînement en modifiant le nombre d'itérations en fonction du nombre d'époques optimales selon votre sortie graphique. Que constatez-vous ?

Que se passent-il lorsque vous augmentez la taille de votre couche cachée ?

VII. Simple LSTM à 2 couches

Tentons maintenant de réaliser un modèle à deux couches.

```
def make_LSTM_network2(in_out_neurons,hidden_neurons1,hidden_neurons2):
    model = Sequential()
    model.add(LSTM(hidden_neurons1, input_dim=in_out_neurons,
    forget_bias_init='one', activation='tanh',
    inner_activation='sigmoid', return_sequences=True) )
    model.add(LSTM(hidden_neurons2, input_dim=hidden_neurons1,
    forget_bias_init='one', activation='tanh',
    inner_activation='sigmoid', return_sequences=False) )
    model.add(Dense(in_out_neurons, input_dim=hidden_neurons2))
    return model
```

De la même manière qu'avec l'exemple précédant entraînez ce modèle, et tracez les graphiques en résultant. Que constatez vous concernant le temps de calcul d'entraînement ? Que pensez-vous des nouvelles performances de ce modèle ? Ré-entraînez votre modèle en réalisant un early stopping, que constatez vous concernant le temps d'entraînement ? Que se passe-t-il au niveaux du graphique traçant prédictions et valeurs attendues ? Nous sommes d'accord, nous nous rendons compte à ce point qu'une seule couche est nécessaire pour bien capturer les propriétés de nos données. Nous vous invitons cependant à empiler plus de couches, observez l'effet du mini-batch et du drop out dans le modèle 3, ainsi que l'effet de la normalisation du batch dans le modèle 4 :

```
def make_LSTM_network3(in_out_neurons,hidden_neurons,batch1,batch2):
    model = Sequential()
    model.add(LSTM(hidden_neurons, input_shape=(batch1, in_out_neurons),
    forget_bias_init='one', activation='tanh',
    inner_activation='sigmoid', return_sequences=True) )
    model.add(Dropout(0.25))
    model.add(LSTM(hidden_neurons, input_shape=(batch2, hidden_neurons),
     forget_bias_init='one', activation='tanh',
     inner_activation='sigmoid', return_sequences=False) )
    model.add(Dense(in_out_neurons, input_dim=hidden_neurons))
    return model


def make_LSTM_network4(in_out_neurons,hidden_neurons1,
hidden_neurons2,hidden_neurons3):
    model = Sequential()
    model.add(BatchNormalization())
    model.add(LSTM(hidden_neurons1, input_dim=in_out_neurons,
    forget_bias_init='one', activation='tanh',
    inner_activation='sigmoid', return_sequences=True) )
    model.add(Dropout(0.25))
    model.add(BatchNormalization())
    model.add(LSTM(hidden_neurons2, input_dim=hidden_neurons1,
    forget_bias_init='one', activation='tanh',
    inner_activation='sigmoid', return_sequences=False) )
    model.add(Dropout(0.25))
    model.add(BatchNormalization())
    model.add(LSTM(hidden_neurons2, input_dim=hidden_neurons1,
    forget_bias_init='one', activation='tanh',
    inner_activation='sigmoid', return_sequences=False) )
    model.add(Dense(in_out_neurons, input_dim=hidden_neurons2))
    return model
```

A l'aide de la fonction suivante visualiser les poids avant et après votre entraînement, que constatez vous lorsque vous n'utilisez pas les techniques de dropout ?

```
weights_1 = modelX.layers[0].get_weights()
w0_1 = weights_1[0][0][0]
w1_1 = weights_1[1][0]
```

VIII. Pour aller plus loin...

Ce tutoriel bien que très limités par la simplicité des données utilisées vous aura permis de manipuler vos premier réseaux de neuronnes profonds et de visualiser les différents paramétrages possibles et dans cas ceux-ci peuvent vous être utile. Nous avons pu utiliser notamment plusieurs techniques tels que le mini-batch, le dropout, la normalisation du batch...

Les données de neiges sont en réalités facile à prévoir à cette échelle, puisque les précipitations ont lieu en général en hiver. Vous pouvez cependant utiliser ce même algorithme sur les données de pluies présentes dans le fichier $autre_var.csv$ fourni. Où encore en utilisant d'autres variables descriptives (utilisez dans ce cas la fonction merge de Keras). Il est aussi possible de remplacer le bloc LSTM par un bloc GRU, je vous invite dans ce cas à consulter le manuel d'utilisation en ligne de Keras : www.keras.io.

VII. Conclusion

Ce travail de recherche fut pour moi une véritable expérience, cela m'a permis de me plonger dans un univers nouveau afin d'en entrevoir les bases. Le Deep Learning est une facette du machine learning qui était à ce jour pour moi inconnu. Ce fut une véritable chance pour moi de pouvoir travailler dessus, notamment parce que cela m'a permis de me retrouver face à une phase de recherche intense. La discipline semble de plus être très sexy autant du coté de la recherche en totale ébullition ces derniers temps que du coté des entreprises puisque les applications qu'offrent ces modèles sont énormes. Bien entendu l'ambition du plan de cet article vu le temps de travail qui m'était disponible ne m'a permis que de balayer en surface le sujet, notamment dans la phase applicative de cet article. Les sources et références fournies plus loin vous permettrons de vous plonger réellement dans celui-ci. J'espère cependant avoir pu vous fournir quelques clefs vous permettant de mieux entrevoir ce qu'était le Deep Learning.

References

[S. HOCHREITER J. SCHMIDHUBER] Long short term memory (1997) *Deutsche Universität München IDSIA*

[R. PASCANU, T. MIKOLOV, Y. BENGIO] On the difficulty of training recurrent neural networks (2012) *Université de Montréal, Brno University of Technology*

[F. ROSENBLATT] The perceptron, a perceiving and recognizing automaton Project Para (1957) *Cornell Aeronautical Laboratory*

[B. WIDROW et al.] Adaptive "Adaline" neuron using chemical "memistors".(1960) *Stanford Electron. Labs.*

[Y. LECUN, B. BOSER, J. DENKER, D. HENDERSON, R. HOWARD, W. HUBBARD, L. JACKEL] Backpropagation Applied to Handwritten Zip Code Recognition (1989) *Neural Computation , vol.1*

[K. FUKUSHIMA] Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position (1980) *NBiological Cybernetics 36*

[G. HINTON, R.S. ZEMEL] Autoencoders, Minimum Description Length and Helmholtz Free Energy. (1993) *in Jack D. Cowan; Gerald Tesauro Joshua Alspector, ed., 'NIPS' , Morgan Kaufmann*

[D.H. ACKLEY, G. HINTON T.J. SEJNOWSKI] A learning algorithm for boltzmann machines(1985) *Cognitive science, 9(1)*

[LeCun, Y., Chopra, S., Hadsell, R., Ranzato, M., Huang, F.] A tutorial on energy-based learning.(2006) *Predicting structured data, 1, 0*

[Neal, R. M.] Connectionist learning of belief networks.(1992) *Artificial intelligence, 56(1)*

[Hinton, G. E., Dayan, P., Frey, B. J., Neal, R. M.] The" wake-sleep" algorithm for unsupervised neural networks (1995) *Science, 268(5214)*

[Yann LeCun and Yoshua Bengio.] Convolutional networks for images, speech, and time series.(1998) *The handbook of brain theory and neural networks, Michael A. Arbib (E()d.). MIT Press*

[Hinton, G. E., Osindero, S., Teh, Y. W.] A fast learning algorithm for deep belief nets. (2006) *Neural computation, 18(7)*

[Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., Salakhutdinov, R. R.] Improving neural networks by preventing co-adaptation of feature detectors (2012) *arXiv preprint arXiv*

[Bengio, Y., LeCun, Y.] Scaling learning algorithms towards (2007) *AI. Large-scale kernel machines, 34(5)*

[air, V., Hinton, G. E.] Rectified linear units improve restricted boltzmann machines. (2010) *27th International Conference on Machine Learning (ICML-10)*

[K.CHO, B. VAN MERRIENBOER, Caglar GULCEHRE, D. BAHDANAU, Y. BENGIO,F. BOUGARES,H. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation (2014) *Université de Montréal, Jocobs University, Université du Maine*

[T. Ziemke] Recurrent Neural Networks: Design and Applications (1999) *Department of Computer Science*

SOURCES WEB :

1. http://colah.github.io/
2. https://www.toptal.com/machine-learning/an-introduction-to-deep-learning-from-perceptrons-to-deep-networks
3. https://karpathy.github.io/
4. http://www.technologies-ebusiness.com/enjeux-et-tendances/le-deep-learning-pas-a-pas
5. http://earthpy.org/pandas-basics.html
6. http://sebastianruder.com/optimizing-gradient-descent/
7. http://www.andreykurenkov.com/writing/a-brief-history-of-neural-nets-and-deep-learning-part-1/
8. http://blog.terminal.com/demistifying-long-short-term-memory-lstm-recurrent-neural-networks/
9. http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/
10. http://imonad.com/rbm/restricted-boltzmann-machine/ https://devblogs.nvidia.com/parallelforall/und natural-language-deep-neural-networks-using-torch/
11. http://blog.christianperone.com/2015/08/convolutional-neural-networks-and-feature-extraction-with-python/
12. http://blog.christianperone.com/2015/08/convolutional-neural-networks-and-feature-extraction-with-python/
13. https://martin-thoma.com/lasagne-for-python-newbies/
14. http://danielnouri.org/notes/2014/12/17/using-convolutional-neural-nets-to-detect-facial-keypoints-tutorial/
15. https://swarbrickjones.wordpress.com/2015/04/29/convolutional-autoencoders-in-pythontheanolasagne/
16. http://mourafiq.com/2016/05/15/predicting-sequences-using-rnn-in-tensorflow.html
17. http://swanintelligence.com/watch-tiny-neural-nets-learn.html