Réseaux à composantes continues

Emmanuel Bengio

Dans un contexte de manifold learning, nous cherchons à montrer qu'un type différent de transition, utilisant des composantes qu'on appelle *continues* (au sens topologique) lors de la reconstruction dans un autoencodeur peut aider à conserver une quantité suffisante d'information dans un nombre restreint (vis-à-vis l'entrée) de dimensions.

1 Introduction

1.1 Apprentissage automatique

Ce projet est un projet d'apprentissage automatique, *machine learning*.

L'apprentissage automatique est une terme qui regroupe un ensemble de techniques qui se servent de données du monde réel (images, textes, son, séquences ADN, etc.) pour créer des modèles qui font ressortir des données une information utile. Ces algorithmes s'apparentent ou sont directements issus d'algorithmes statistiques et d'optimisation.

1.2 Deep learning

Un type de modèle que nous utilisons dans ce projet, le réseaux de neurones, est un modèle typiquement utilisé en deep learning. Ce modèle effectue essentiellement une transformation vectorielle non-linéaire de $\mathbb{R}^{d_x}$ vers $\mathbb{R}^{d_h}$.

L'idée principale du deep learning est "d'empiler" ces transformations pour transformer successivement un vecteur d'entrée x vers une représentation utile s. Toute la puissance du deep learning réside dans le fait que ces transformations sont apprises en fonction des données qu'on donne à l'algorithme.

En apprentissage supervisé, cette représentation peut-être une classe, une probabilité, un vecteur de celles-ci, ou tout simplement un vecteur de valeurs désirées dans le cas de la régression. En apprentissage non-supervisé, on cherche typiquement à obtenir un représentation s "utile".

1.3 Représentation

Ce qu'on appelle représentation, c'est simplement la forme concrete numérique que quelque chose va prendre. Par exemple on peut représenter le chiffre "1" par un vecteur dans $\mathbb{R}^1$ contenant l'entier 1; on peut aussi le représenter par une image 32×32 (donc dans $\mathbb{R}^{1024}$) avec un "1" manuscrit. On pourrait aussi bien s'imagine un algorithme qui décompose cette dernière image en un ensemble de composantes géométriques, présentes ou non (on obtient donc un vecteur "binaire" dans $\mathbb{R}^d$).

Qu'est-ce qu'une bonne représentation? Question parfois difficile auquelle on a deux réponses à priori.

Une bonne représentation peut en être une qui démèle les facteurs de variation de l'entrée. Typiquement, une donnée brute va comporter énormément de dimensions redondantes (mais quelle dimension est redondante va dépendre de la donnée, et va donc varier énormément), par exemple pour une image de chiffre manuscrit on peut s'imaginer qu'il n'existe que 4 facteurs de variation principaux: le chiffre en tant que tel, la taille du crayon utilisé, le style d'écriture, la translation/rotation sur l'image. On voit donc qu'une représentation qui démèlerait ces facteurs de variation (e.g. de telles sorte que si le chiffre reste le même mais la translation change, il n'y ait que le facteur de translation qui change) serait non seulement plus abstraite mais aussi sémantiquement plus utile qu'un ensemble à priori disparates de pixels.

Une autre avenue moins spécifique est simplement la préservation de l'information. Si on arrive à mesurer quantitativement qu'une représentation contient autant

(ou presque) d'information que la représentation brute initiale, alors elle risque d'être aussi (mais on espère *plus*) utile que la représentation brute. Par exemple, si un texte qui décrit une image peut être utilisé pour redessiner l'image correctement, alors le texte préserve correctement l'information.

2 Travaux reliés

PCA

La méthode de principal component analysis en est une reliée à la nôtre, mais qui est beaucoup plus naïve. Il s'agit de trouver une transformation qui projette les dimensions correlées vers une nombre moindre de dimensions décorrelées, qu'on appelle composantes principales.

Sparse coding

La méthode de sparse coding en est une autre qui cherche une bonne représenation, mais fait une hypothèse alternative. Elle prédit qu'une représentation à avantage à être en très haute dimension, mais à être très sparse, c'est à dire que le vecteur de représentation sera presque exclusivement constitué de zéros, sauf pour quelques dimensions qui seront actives. Ainsi ce n'est pas tant la valeur de l'activation qui compte mais bien si une dimension est active ou non.

Manifold learning

Il existe plusieurs algorithmes de manifold learning, mais il font tous l'hypothèse que l'on a fait plus tôt; on peut projeter les données d'un ensemble vers un espace de dimension beaucoup plus faible qui est suffisant pour expliquer les facteurs de variation des éléments de l'ensemble.

Autoencodeurs

Le modèle d'un autoencodeur force une représentation à l'intérieur du modèle à contenir suffisament d'information pour permettre la reconstruction à partir de cette représentation de l'entrée du modèle.

On observe ainsi que le modèle préserve l'information contenue dans les données, et ce avec quelques avantages. Un extension est le *denoising autoencoder* qui rajoute du bruit aux entrées mais qui force le modèle à débruiter celles-ci. Ainsi on obtient un modèle (et une représentation) plus robuste à ces changements.

Une autre variante est le contractive autoencoder, qui force la représentation à ne varier que dans certaines directions, et de rester invariante aux variations de l'entrée (sauf dans les directions nécessaires à la reconstruction).

3 Réseau de neurones

Les réseaux de neurones artificiels sont ce qu'on utilise dans ce projet. On appelle *feedforward* un réseau où l'information ne se propage que dans un sens, de l'entrée vers la sortie. Les réseaux de neurones sont généralement divisés en couches de neurones où les neurones dans une couche sont reliées à toutes les autres neurones de la couche suivante. On peut voir un réseau de neurone à une seule couche comme une matrice de poids W et un vecteur de biais b: chaque dimension x_i de l'entrée x envoie un signal à toutes les neurones h_j, signal qui est la valeur de x_i pondérée par un poids w_{ij}. Chaque neurone $h_j = \sum_i w_{ij} x_i$ est ensuite "activée" par une fonction dite d'activation f, généralement à allure sigmoidale (comme la tangente hyperbolique tanh ou la fonction logistique $1/(1+e^{-x})$). Chaque activation est centrée autour d'un biais b_j.

On a donc que la sortie du réseau à une couche est:

$$h(\boldsymbol{x}) = f(\boldsymbol{x}W + \boldsymbol{b})$$

Si l'entrée x est un vecteur de dimension d et la sortie $h(x)$ de taille m on a une matrice $W^{d \times m}$ et un vecteur de biais de dimension m. L'idée de l'apprentissage est de trouver les W et b optimaux (et potentiellement les f optimales) à la résolution d'une tâche.

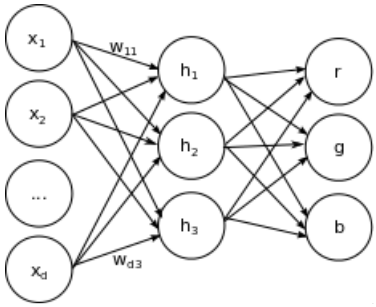

Figure 1, un réseau de neurone à une couche cachée de $m = 3$ neurones.

Un réseau de neurones à plus d'une couche est dit profond (d'où le *deep learning*) et les couches intermédiaires sont dites couches cachées. Pour un réseau de neurones *feedforward* il suffit "d'empiler" les couches en considérant l'entrée d'un couche comme la sortie de la précédente. De plus, chaque couche peut avoir sa fonction d'activation particulière. Voici un exemple avec une couche cachée:

$$h(x) = f_2(f_1(xW^1 + b^1)W^2 + b^2)$$

Modèles autoencodeurs Comme expliqué plus tôt, les modèles autoencodeurs servent à apprendre une représentation h de leur entrée, et le font en minimisant une fonction de perte où le but est d'être capable de reconstruire l'entrée x dans $x^{(r)}$ à partir de h:

$$\mathcal{L}(x) = |x - x^{(r)}|^2$$

Le modèle autoencodeur que l'on utilise comme baseline est le suivant:

$$\begin{aligned} \boldsymbol{h}(\boldsymbol{x}) &= f(f(\boldsymbol{x}W^1 + \boldsymbol{b}^1)W^2 + \boldsymbol{b}^2) \\ \boldsymbol{x}^{(r)}(\boldsymbol{h}) &= f(f(\boldsymbol{h}(W^2)^T + \boldsymbol{b}^{2r})(W^1)^T + \boldsymbol{b}^{1r}) \\ &\text{où } f(x) = \frac{1}{1+e^{-x}} \end{aligned}$$

Entrainement des réseaux de neurones On entraine nos modèles en utilisant une descente de gradient. C'est à dire qu'on va minimiser une perte $\mathcal{L}$ en suivant pas-à-pas le gradient des paramètres θ par rapport à la perte.

$$\theta_{t+1} \leftarrow \theta_t + \lambda \frac{\partial \mathcal{L}}{\partial \theta_t}$$

Ici λ est le taux d'apprentissage. C'est la vitesse à laquelle on suit le gradient pour optimiser les paramètres. Comme typiquement le paysage du gradient est loin d'être une belle pente lisse et est plus proche d'un paysage montagneux et accidenté, si λ est trop grand on va passer d'un pic à un autre sans jamais faire baisser l'erreur. Si λ est trop petit, on va rester coincé dans une vallée et on ne pourra plus en ressortir. Il y a donc un compromis important à faire, et un juste millieu à trouver (ce qui peut s'avérer ardu).

4 Modèle à composantes continues

4.1 Motivation

On remarque empiriquement que l'activation des unités dans les réseaux de neurones artificiels se fait de manière plutôt binaire. C'est à dire que pour une entrée donnée, une unique unité va avoir tendance à être soit allumée (plus proche de 1) ou éteinte (proche de 0).

Or, il pourrait être intéressant d'avoir dans un réseaux des neurones qui s'activent de manière continue (au sens topologique, puisque nos unités sont déjà des nombres continus, à point flottant) vis à vis de leur entrée, exprimant ainsi des phénomènes comme la translation.

De plus, dans un contexte de manifold learning, de telles composantes vont probablement aider à démêler et mieux exprimer "sémantiquement" les facteurs de variation continus dans un nombre restreint de dimensions.

4.2 Architecture

On va donc modifier légèrement un autoencodeur pour parvenir à obtenir une telle représenation.

La passe *feedforward* est assez typique, à part qu'on utilise une fonction d'activation linéaire sans biais pour la représentation finale.

Ce qui est particulier c'est la reconstruction $h^{(r)}$.

$$\begin{aligned} \boldsymbol{h} &= f(\boldsymbol{x}W^1 + \boldsymbol{b}^1) \\ \boldsymbol{s} &= \boldsymbol{h}W^2 \\ h_i^{(r)} &= \exp\left(-(\boldsymbol{s} - \mu_i)^T D_i (\boldsymbol{s} - \mu_i)\right) \\ \boldsymbol{x}^{(r)} &= f(\boldsymbol{h}^{(r)}(W^1)^T + \boldsymbol{b}^{1r}) \\ &\text{où } f(x) = \frac{1}{1+e^{-x}} \end{aligned}$$

L'idée principale est qu'à chaque s_j va être associé un ensemble de μ_i. Quand s_j est proche de μ_i, h_i est activé. Si on considère l'axe d'une unité s_j, elle va être parsemée de μ_i, et donc selon l'activation *continue* de s_j sur son axe, différent h_i vont être activés. On a donc bien le transfert d'unités "binaires", les h, vers des unités plus continues, les s. Tout du moins le modèle en est capable.

À date, et pour pouvoir être capable d'entraîner le réseau correctement, on limite D_i à n'associer qu'une seule dimension de s à chaque h_i. Cela veut dire que D_i

est une matrice diagonale, dont la diagonale elle même est à zéro sauf en un point où elle est à 1.

5 Projet

On fait notre projet en Python, et on utilise pour ce projet la librairie theano pour construire notre modèle, ainsi que le baseline.

Le code est donc composé de trois modules. `model.py` contient les classes qui décrivent nos modèles, notemment `HiddenLayer` et `CCLayer` qui sont les composantes (en termes de couches) du modèle. Ce module contient aussi du code pour charger les données nécessaires à l'entraînement.

`train_model.py` permet d'entrainer un modèle, de visualiser ce qui s'y passe lors de l'entraînement et de récolter les résultats.

`finetune.py` permet d'entraîner un modèle de manière supervisée à partir des poids appris par l'entrainement supervisé (que le module précédent effectue), sur une tâche de classification.

Le projet à évolué de mai à juillet, au début on effectuait une approche assez naïve où tous les paramètres du modèles étaient appris et où les poids de reconstruction étaient non liés. J'ai aussi essayé de rajouter une couche cachée, de rajouter des fonctions de régularisation de la variation des s et des h. Il s'est avéré que soit le modèle n'apprenait rien d'utile, soit il n'apprenait pas du tout. En particulier en modifiant les initialisations de D on a pu se rendre à des résultats bien plus intéressants à la fois visuellement et en termes d'erreur numérique.

On s'est donc limité finalement à initialiser les D de la manière décrite à la section précédente, en associant à chaque s_j un nombre fixe de h_i.

6 Résultats

6.1 Poids

En observant qualitativement les poids des premières couches des réseaux, on voit qu'on a obtenu ce qu'on désirait (jusqu'à un certain point). On voit des groupes d'unités h_i correspondre à des uniques unités s_j qui varient en fonction de quel h_i est le plus activé, donc sur leur axe comme on s'y attend.

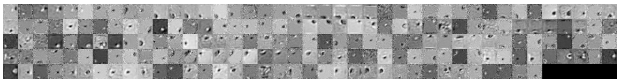

Figure 2, poids de la première couche cachée pour un modèle entrainé sur MNIST

Ces résultats sont encourageants, puisqu'ils montrenent bien que l'on a réussi à capturer ce que l'on cherchait.

6.2 Reconstruction

Les résultats que l'on obtient avec notre modèle sont parfois loin d'être une reconstruction exacte de leur entrée. Par contre, on fait ces expériences dans l'optique de pouvoir radicalement réduire la dimensionnalité des données, et en se comparant à d'autres méthodes (notemment au modèle le plus proche, un autoencodeur) on obtient que notre modèle fait presque aussi bien (erreur entre 2-10% plus élevée).

Notemment, notre modèle rapproche beaucoup l'erreur de test et d'entraînement. Donc malgré qu'on obtient des erreurs légèrement plus élevées, il est possible que notre modèle soit mieux régularisé, qu'il généralise mieux.

6.3 Classification

On obtient pour le *finetuning* des résultats intéressants. Sur une tâche plus complexe, les visages, on obtient une meilleure performance de la part de notre méthode que de la part d'un autoencodeur *à capacité égale* (en termes de nombre de paramètres et d'unités dans la couche finale). Ce point est important ici puisqu'on ne cherche pas à savoir (pour le moment) si notre modèle est meilleur dans l'absolu mais bien s'il est capable de faire mieux, de générer une meilleure représentation, avec peu d'unités dans la couche finale.

	MNIST	*Faces*
Notre méthode	12.96%	17.60%
Autoencodeur	6.52%	19.50%
Réseau à convolution	0.9%	14.2%

7 Conclusion

Il reste donc à explorer plusieurs avenues pour cette méthode, notemment l'initialisation des paramètres et l'architecture en tant que telle, mais aussi la quantification de notre objectif. En effet, nos seules mesures en ce moment sont l'erreur de reconstruction, l'erreur de classification quand on *finetune*, et qualitativement la fidélité visuelle de la reconstruction faite par le modèle.

Il serait fort intéressant de pouvoir mesurer quantitativement une partie de ce qu'on essaie de faire, notemment si des effets locaux comme par exemple la translations d'images se reflètent par une variation continue et consistante de la représenation s. Ce genre de mesures pourraient soit mieux aider à quantifier la performance des modèles, soit carrément servir d'objectifs d'optimisation, de régularisation, lors de l'apprentissage.

Il reste aussi à se comparer plus en profondeur avec d'autres techniques existantes qui essaient d'obtenir les mêmes résultats. Cela nous permetterait peut-être de mieux comprendre certaines faiblesses, ou points forts, de notre modèle.

Liens

Ce rapport, ainsi que le code et les compte rendus hebdomadaires se retrouvent à l'URL suivant: http://bengioe.github.io/ccae