Machine Learning

Notes on A. Ng lectures

Jérémy Fix
Jeremy.Fix@Supelec.fr

Résumé *Notes on the Machine Learning lectures of A. Ng.*

Table des matières

1 Linear regression with one or multiple variables

2 Logistic Regression

We focus on classification problems for which the output y is binary (bi-class classification problem, later on, we will speak about multiclass classification) : $y \in \{0, 1\}$. We call class-0 the negative class and class-1 the positive class but these are just arbitrary names. For example, one may seek to learn to classify a tumor to be malignant or not depending on its size, or filter mails as being spam or not depending on some measure.

With linear regression, we can define a boundary condition saying : "if $h_\theta(x)$ is larger or equal than 0.5 then consider it as belonging to class-1, otherwise it belongs to class-0".

Note : il utilise le problème de classer des tumeurs, en ajoutant un outlier tends à déplacer la frontière de décision pour donner un mauvais classifieur. "I would not use linear regression for classification problems".

Il introduit la régression logistique pour contenir l'hypothèse $h_\theta(x)$ dans $[0, 1]$ alors qu'avec la régression linéaire peut donner des valeurs bien au dela de ce domaine alors que les étiquettes valent disont 0 et 1... Mais il y a aussi la sensibilité aux outliers(**à vérifier ! !**)

On introduit le modèle de régression logistique :

$$h_\theta(x) \quad = \quad g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}} \tag{1}$$

On peut voir $h_\theta(x)$ comme la probabilité estimée que $y = 1$ pour l'entrée x. C'est à dire qu'on considère $h_\theta(x)$ comme un modèle paramétré de $P(y = 1|x; \theta)$. Pour trancher si une entrée x appartient à l'une ou l'autre des classes, il faut introduire un seuil sur cette probabilité. Par exemple, si on prédit $y = 1$ quand $h_\theta(x) \geq 0.5$, cela correspond à prédire $y = 1$ quand $\theta^T x \geq 0$. Le plan correspondant à $\theta^T x = 0$ est appelée "decision boundary". Comme le feature vector peut contenir des combinaisons non-linéaires de nos observations, e.g. $x = [1, x_1, x_2, x_1^2, x_2^2]$, la boundary decision peut être non-linéaire quand tracée dans l'espace des observations, e.g. x_1, x_2.

On doit maintenant dériver des algorithmes pour trouver les paramètres θ. Partant d'une base d'apprentissage $(x^{(i)}, y^{(i)})$ avec $\forall i, y^{(i)} \in \{0, 1\}$, on cherche les paramètres θ. Pour la régression linéaire, on a quantifié la classification par une fonction quadratique $J(\theta) = \frac{1}{2m}\sum_i (h_\theta(x^{(i)}) - y^{(i)})^2 = \frac{1}{m}\sum_i \text{cost}(h_\theta(x^{(i)}), y^{(i)})$. La fonction de coût est donc une fonction des coûts de chaque exemple. Pour la régression linéaire, avec un coût quadratique, le coût $J(\theta)$ était convexe et le problème d'optimisation avait donc un seul minimum. Avec une hypothèse h_θ logistique, la fonction de coût n'est plus convexe si on utilise un coût quadratique ce qui a la conséquence fâcheuse d'introduire plusieurs minimums locaux à la fonction de coût (ce qui pose des problèmes pour la minimiser avec, par exemple, une descente de gradient). Pour la régression logisitique, on introduit le coût logarithmique :

$$\text{cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{si } y = 1 \\ -\log(1 - h_\theta(x)) & \text{si } y = 0 \end{cases} \tag{2}$$

On remarque qu'on peut réécrire cette fonction de coût [1] sous la forme $\text{cost}(h_\theta(x), y) = -(1 - y)\log(1 - h_\theta(x)) - y\log(h_\theta(x))$. Cette fonction de coût est convexe et donc sans minimums locaux. On en vient à introduire la fonction de coût $J(\theta)$:

$$J(\theta) \quad = \quad \frac{1}{m}\sum_i \text{cost}(h_\theta(x^{(i)}), y^{(i)}) \tag{3}$$

$$\quad = \quad -\frac{1}{m}\left(\sum_{i=1}^{m} y^{(i)}\log(h_\theta(x^{(i)})) + (1 - y^{(i)})\log(1 - h_\theta(x^{(i)}))\right) \tag{4}$$

Pour trouver les paramètres qui minimise cette fonction de coût, on peut considérer une descente de gradient :

$$\theta_j \leftarrow \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta) \tag{5}$$

En calculant le gradient du coût par rapport aux paramètres, on obtient la règle de mise à jour des paramètres :

$$\theta_j \leftarrow \theta_j - \alpha\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \tag{6}$$

1. On aurait pu imaginer une autre formulation de la fonction de coût $\text{cost}(h_\theta(x), y) = -\log((1 - y)(1 - h_\theta(x)) + yh_\theta(x))$ **mais** il y a des raisons théoriques pour préférer la version du texte, elle est convexe et est dérivée de principe de Maximum Likelihood estimation **? ? ? ? ? ?**

Comme pour la régression linéaire, les paramètres sont mis à jour simultanément. On pourra remarquer que cette règle de mise à jour est identique à la règle de mise à jour des paramètres pour la régression linéaire, à ceci près que l'hypothèse h_θ est différente (et la fonction de coût a été particulièrement bien choisie). Comme pour la régression linéaire, on peut ajuster l'échelle des features (feature scaling) pour améliorer la vitesse de convergence de la descente de gradient.

Tracer la fonction de coût quand $y = 1$ pour montrer que si $h_\theta(x)$ tend vers 1, le coût tends vers 0 ; Au contraire si $h_\theta(x)$ tends vers 0 (alors que $y = 1$ puisque que $h_\theta(x) = P(y = 1|x; \theta)$), le coût tends vers l'infini et l'algorithme est donc fortement pénalisé pour cette erreur. On peut mener un raisonnement similaire pour le cas $y = 0$ dans lequel on pénalise fortement lorsque $h_\theta(x) \to 1$.

La descente de gradient est une des manières de minimiser notre fonction de coût. Il en existe d'autres, parfois plus rapide en terme du nombre d'itérations nécessaires pour atteindre le minimum. Une liste d'algorithmes alternatifs est donnée ci-dessous :

- gradient conjugué
- BFGS
- L-BFGS

Ces algorithmes ont un certain nombre d'avantages par rapport à la descente de gradient :

- il n'y a pas besoin de spécifier manuellement un taux d'apprentissage α, celui-ci est ajusté automatiquement
- ils sont en général plus rapide à converger qu'une descente de gradient

Ils sont néanmoins plus compliqués à comprendre et mettre en œuvre. Cela dit, il existe un certain nombre d'implémentations clé-en-main de ces algorithmes.

With Octave, you would make use of the *fminunc* function with specific options setting which algorithm to use.

We now go back to multi-class classification. Therefore, we suppose that the output y can take more than two values. One way to solve this problem is to consider multi-class classification problems as multiple binary classification problems. We then train as many binary classifiers as we have classes. Each of these classifier learns to recognize one class versus all the others. Given all the individual classifiers h_i, we may define label an input x as belonging to the class which maximizes the outputs $h_i(x), \forall i \in [1..k]$.

3 Regularization

3.1 Motivation

Sometimes, when trying to fit an hypothesis to a data set, we may run into a problem called overfitting. Overfitting arises because we have just a partial observation of the true model that generates the data we are trying to fit. We therefore make an hypothesis on the shape of this model. Obviously, when we fit a model on a dataset, we want to interpolate the output for unobserved inputs (inputs that are not in the training set). The hypothesis we consider define how different we can interpolate this output. The simplest example of overfitting can be seen when fitting a polynomial on data set. If we increase the degree of the polynomial, we introduce more flexibility to the model (indeed, with the Lagrange polynomials, we know that we can fit perfectly any data set if we don't constrain the degree of the polynomial). This flexibility can lead to more variability in the model which may impairs the ability of the model to generalize, i.e. to give the right answer on new data unavailable in the training set.

There are two options to adress overfitting. The first one is to reduce the number of features whether by manually selecting the features to keep or to use model selection algorithm. However, throwing away some features may throw away some information contained in the dataset. The second option is called regularization. Regularization is a technique which allows to balance the tendency to overfit the training set by a term which constrains the *complexity* of the model.

Des données à fiter avec un polynôme

Suppose we consider linear regression with some polynomials of the inputs :$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$. If we penalize the two parameters θ_3 and θ_4, the polynomial h_θ will be simpler. To penalize these two parameters we can modify the quadratic cost function by adding terms that increases the cost depending on the amplitude of these parameters, e.g. $J(\theta) = \frac{1}{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + 1000\theta_3^2 + 1000\theta_4^2$. Minimizing this modified cost function will constrain the parameters θ_3 and θ_4 to be small. More generally, as we may not know which parameters influence overfitting, we introduce the following cost function for linear regression :

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda\sum_{i=1}^{n}\theta_j^2 \tag{7}$$

By convention, θ_0 is not regularized. The parameter λ balances the trade off between fitting the dataset and keeping the parameters θ_i small. If λ is set too large, minimization of this cost function may unfortunately lead to underfitting, i.e. we even do not fit the training set. Fortunately, one may define algorithms automatically adjusting the penalization coefficient λ.

Question : est ce qu'introduire le terme de régularization ne rends pas la fonction de coût non convexe ? NON d'après une question du quizz, la fonction de coût reste convexe.

Question : pourquoi est ce qu'on s'en fout de ne pas régularizer le terme θ_0.

Question : pourquoi ne pas pénaliser plus les termes de grand degré ?

3.2 Regularized linear regression

We remind the cost function for linear regression with L2 regularization :

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda\sum_{i=1}^{n}\theta_j^2 \tag{8}$$

Considering this regularized cost function, one must modify the gradient descent algorithm. The update of the parameters now reads :

TODO, calculer le gradient, d'après Ng : $\theta_0 \leftarrow \theta_0 - \alpha/m\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)}; \forall j \neq 0, \theta_j \leftarrow \theta_j - \alpha(1/m\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} + \lambda/m\theta_j) = (1 - \alpha\lambda/m)\theta_j - \alpha/m....$

By isolating the terms depending on θ_j, one gets an update rule of the form $\theta_j \leftarrow \beta\theta_j - \alpha/m..$, we see that we shrink the parameter and move in a direction minimizing the cost function (when considered without the regularization term).

We also saw that we can use normal equations for solving linear regression. Without regularization, the optimal parameters using normal equation was :$\theta = (X^TX)^{-1}X^Ty$. We regularization inside, the normal equation now reads $\theta = (X^TX + \lambda R)^{-1}X^Ty$ where R is a diagnoal matrix filled in with 1, except for the first element which is set to 0. For e.g., with $n = 2$, $R = [000; 010; 001]$.

If you have less examples than features, the matrix X^TX is non invertible. If you use regularization, it is possible to prove that the matrix $X^TX + \lambda R$ cannot be singular and that you can always invert it.**AH OUAIS ? ? ?**.

3.3 Regularized logistic regression

For logistic regression, the cost function reads :

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}\log(h_\theta(x^{(i)})) + (1 - y^{(i)})\log(1 - h_\theta(x^{(i)}))) \tag{9}$$

We can also add L2 regularization to this cost function to get :

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}\log(h_\theta(x^{(i)})) + (1 - y^{(i)})\log(1 - h_\theta(x^{(i)}))) + \lambda/(2m)\sum_j\theta_j^2 \tag{10}$$

The gradient descent update is then modified into $\theta_0 \leftarrow \theta_0 - \alpha/m\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)}; \forall j \neq 0, \theta_j \leftarrow \theta_j - \alpha(1/m\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} + \lambda/m\theta_j) = (1 - \alpha\lambda/m)\theta_j - \alpha/m....$

The update is very similar to regularized linear regression but remind that the hypothesis h_θ is different for logistic regression.

4 Idées de démos

Pour la régularisation, une appli dans laquelle on fit un polynôme sur des données et on règle dynamiquement la contribution λ de la régularisation pour voir la forme du modèle appris.

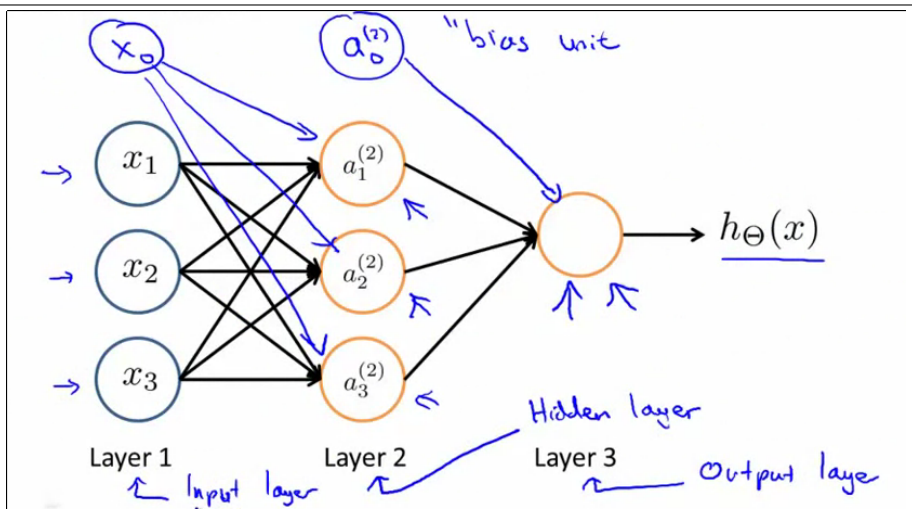
5 Neural Networks : Representation

A vérifier : overfitting if we increase the number of features with a constant number of data ?

Neural networks are the state of the art technique for several machine learning problems. To motivate neural networks, let us consider a non-linear classification problem. As we saw, we can use logistic regression with polynomial features (say $x_1x_2, x_2^3x_3, ..$). However, given the size of our inputs, the size of the feature vector may be a polynomial of it which will certainly be quite big and increase the tendency to overfitting (if we consider too many features relative to the number of the data we have, we can easily encounter overfitting). To give on example, if one considers object recognition from images of size 50×50, one obtains images with $n = 2500$ pixels. If we just consider only quadratic terms x_ix_j, one gets around 3 million features, the number of features is indeed $n^2/2$. One may limit the number of features, but this selection is, there, somehow arbitrary. Neural networks are a much better way to define non-linear hypotheses than logistic regression with polynomial features.

Neural networks, while originally derived from the desire to mimick the brain, is used from an abstract viewpoint in Machine Learning where we consider simple massively interconnected entities. Each entity receives signal from others, process it locally and transmit a signal to other entities. The simple model we consider is a logistic unit, which receives inputs denotes x_i, computes a weighted sum of it and apply a sigmoid transfer function on it. This is nothing more than what we saw in the previous section $y = h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$. We also add a bias unit to the inputs $x_0 = 1$ in addition to the signals received from the other neurons $x_i, i \geq 1$. In the neural network literature, θ are called the weights. We can build layers of such units and interconnect them. A multilayer neural network is shown on fig.1.

Figure 1 – ..



We will denote $a_i^{(j)}$ the activation of the i-th unit of j-th layer. We denote $\theta^{(j)}$ the matrix of weights from layer j to layer $j+1$ [2]. The activation is computed according to :

$$\begin{cases} \forall i \geq 1, \forall j > 1, a_i^{(j)} & = g(\sum_k \theta_k^{(j-1)} a_k^{(j-1)}) \\ \forall i \geq 1, a_i^{(1)} & = x_i \\ \forall j, a_0^{(i)} & = 1 \end{cases} \tag{11}$$

These equations tell nothing more than that the first neuron is the bias for the next layer, the activations of the first layer equal our inputs and each neuron in a subsequent layer computes its activation as a sigmoid of a weighted sum of the activations of the previous layer. A vectorized notation for computing the activations of layer $j+1$ is simply $a^{(j+1)} = g(\theta^{(j)}.a^{(j)})$ where $\theta^{(j)}$ is a matrix of size $|L_{j+1}| \times |L_j|$ with $|L_j|$ the number of neurons in layer j (the bias being included).

2. We here consider only feedforward neural networks were the connections always go from layer j to layer $j+1$