Machine Learning TP3: Classification multi-classes

Melvin DUBEE - Tanguy ROUDAUT FIPASE 24

31 octobre 2023

1 Réseau de neurone

1.1 Visualisation des données

Pour ce TP, on se basera sur le même ensemble de données que pour le TP2. Pour débuter, on appliquera la méthode de propagation avant, couramment appelé *feedforward propagation* sur la matrice X. Nous avons 5000 échantillons d'entrainement, où chaque échantillon représente une image d'un chiffre en niveau de gris de $20x20$px. Ce qui nous donne finalement une matrice de dimension 5000 sur 400.
Dans un premier temps, nous allons en afficher une centaine de manière aléatoire à l'aide du script donné. Chaque visualisation est différente.

FIGURE 1 – Visualisation des données

1.2 Représentation du modèle

Dans cette partie, nous allons implémenter l'architecture de réseau de neurone de la figure 2. Les données d'entrainement sont chargées dans les variables X et y, tandis que les paramètres de réseau θ_1 et θ_2 ont déjà été entraînés et sont optimales pour effectuer l'exercice.

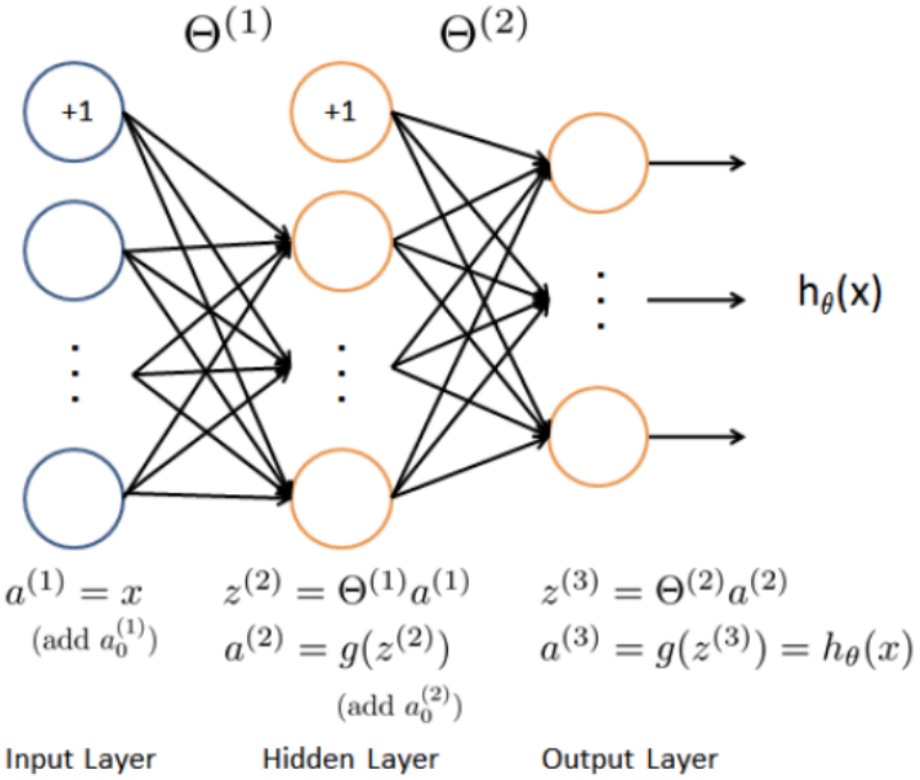

FIGURE 2 – Neural network model

L'objectif est donc d'appliquer la méthode de *feedforward propagation*, cette méthode permettra de renvoyer la prédiction d'un réseau neuronal. Une fois implémenter, lors de l'appelle de la fonction 1 comme dans la stratégie de classification *One-Vs-All*, la prédiction du réseau neuronal sera l'étiquette qui a la plus grande sortie $h_\theta((x))_k$.

La différence avec une régression logistique multi-classe est que le réseau de neurones sera capable de représenter des modèles complexes qui forment des hypothèses non linéaires, ce qui n'est donc pas le cas pour la régression.

1.3 Implémentation

```python
def predictNeuralNetwork(Theta1, Theta2, X):

    # Useful values
    m, _ = X.shape
    num_labels, _ = Theta2.shape
    # Input Layer
    a1 = X

    # Hidden Layer
    z2 = a1 @ Theta1.T
    a2 = sigmoid(z2)

    # Add column 1's to the matrix X
    a2 = np.hstack((np.ones((X.shape[0], 1)), a2))

    # Output Layer
    z3 = a2 @ Theta2.T
    hypothesis = np.argmax(sigmoid(z3), axis=1) + 1

    return hypothesis

    """return
    Training Set Accuracy: %f 97.52
    Expected training Set Accuracy: 97.5%
    """
```

Listing 1 – predictNeuralNetwork

Ainsi, on obtient la précision et l'affichage de chaque digit de l'ensemble d'entrainement avec leur étiquette associée. Voici quelques exemples visibles ci-dessous.

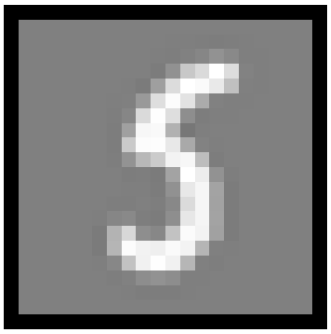

FIGURE 3 – Neural prediction network : digit 5

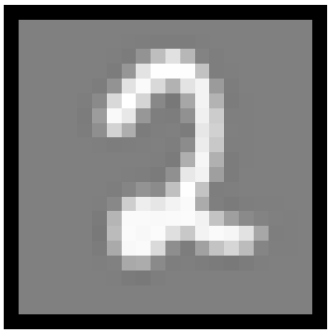

FIGURE 4 – Neural prediction network : digit 2

2 Classificateur pour un réseau de neurone

Cette section a pour but d'implémenter une partie d'un algorithme de rétropropagation dit *Backpropagation*. Cet algorithme permettra d'entrainer notre réseau de neurone afin d'obtenir un coût minimal. Le principe du *Backpropagation* est d'effectuer de manière itérative un calcul entre le gradient de la fonction de coût et les poids du réseau de neurone. Ainsi, à chaque itération, les poids du réseau sont ajustés pour avoir une prédiction plus précise.

2.1 Fonction de coût

Tout d'abord, comme pour les TP passés, nous allons effectuer le calcul du coût $J(\theta)$ qui permet de mesurer la qualité de la prédiction, si le coût est faible alors notre prédiction est proche des valeurs réelles et inversement si le coût est important. La formule utilisée pour calculer le coût change puisque l'on travail sur un réseau de neurone.

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} \left[-y_k^{(i)} \log((h_\theta(x^{(i)}))_k) + (1 - y_k^{(i)}) \log(1 - (h_\theta(x^{(i)}))_k) \right] \tag{1}$$

Pour cette section, il faut bien faire attention à l'implémentation de la matrice Y. En effet, il faut veiller à bien recoder les étiquettes en tant que vecteur composé de 0 et de 1. Pour mieux visualiser, voici la matrice 2.1 correspondant à $y^{(i)}$ si $x^{(i)}$ est une image du digit 2. On a dix chiffres donc $y^{(i)}$ est de dimension 10.

$$y = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Implémentation

```
        y_matrix = np.zeros((num_labels,m))

        for i in range(m):
        for j in range(num_labels):
            y_matrix[j, i] = (j == y[i]-1).astype(int)

        # Compute Cost
        a1 = np.hstack((np.ones((X.shape[0], 1)), X))

        # Hidden Layer
        z2 = a1 @ theta1.T
        a2 = sigmoid(z2)

        # Add column 1's
        a2 = np.hstack((np.ones((a2.shape[0], 1)), a2))

        # Output Layer
        z3 = a2 @ theta2.T

        a3 = sigmoid(z3)

        J = (1/m) * np.sum(-y_matrix.T * np.log(a3) - (1 - y_matrix.T) * np.log(1-a3))

        """return
        Cost at parameters (loaded from ex3weights): 0.287629
        (this value should be about 0.287629)
        """
```

Listing 2 – Non Regularized Cost Function

Ici, nous avons donc créé la nouvelle matrice *y_matrix* à partir de l'ensemble d'étiquettes de base, la matrice *y*. Dans ce code, nous parcourons à l'aide de deux boucles *m*, le nombre total d'échantillons d'entraînement, et *num_labels*, le nombre total de classes. Cela nous permet ensuite de comparer la *i*-ème classe de *y* avec la classe *j* courante. En utilisant la commande Python *.astype(int)*, nous convertissons le résultat *True/False* en *1/0*.
Au final, ce code parcourt chaque échantillon et chaque classe, puis remplit la matrice *y_matrix* de telle sorte que la classe de chaque exemple soit indiquée par un 1 dans la ligne correspondante, et toutes les autres classes soient indiquées par des 0.

Le résultat du coût attendu est le bon, on peut considérer que notre fonction non régularisée est correcte.

2.2 Fonction de coût régularisée

Dans cette section, on va régulariser la fonction de coût en s'appuyant sur la formule donnée pour un réseau de neurone.

$$J(\Theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log((h_\Theta(x^{(i)}))_k) - (1-y_k^{(i)})\log(1-(h_\Theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\left[\sum_{j=1}^{25}\sum_{k=1}^{400}(\Theta_{j,k}^{(1)}(1))^2 + \sum_{j=1}^{10}\sum_{k=1}^{25}(\Theta_{j,k}^{(2)}(2))^2\right] \tag{2}$$

Implémentation

```
1
2       # Cost regularisation
3       reg = (Lambda / (2 * m)) * (np.sum(theta1[: , 1:] ** 2) + np.sum(theta2[:, 1:] ** 2))
4       J = J + reg
5
6       """return
7       Checking Cost Function (w/ Regularization) ...
8       ---------------------------
9       Cost at parameters (loaded from ex3weights): 0.383770
10      (this value should be about 0.383770)
11      """
```

Listing 3 – Regularized Cost Function

Le résultat obtenu est correcte, notre fonction de coût est bel et bien régularisée.

3 Backpropagation

Dans cette section, on continue d'implémenter la *(*Backpropagation) avec le calcul du gradient. De plus, on entrainera le réseau de neurone à l'aide de la fonction d'optimisation avancée *fmin_cg* qui nous permet d'obtenir une fonction de coût $J(\theta)$ minimale.

3.1 Sigmoid gradient

La formule du gradient du sigmoïde est obtenue avec la formule suivante.

$$g'(z) = \frac{d}{dz}g(z) = g(z)(1 - g(z)) \qquad \text{avec} \qquad \text{sigmoid}(z) = g(z) = \frac{1}{1 + e^{-z}} \tag{3}$$

Implémentation

```
1   def sigmoidGradient(z):
2
3       g = sigmoid(z) * (1 - sigmoid(z))
4
5       return g
6
7   """return
8   Evaluating sigmoid gradient...
9   Sigmoid gradient evaluated at [1 -0.5 0 0.5 1]:[0.19661193 0.23500371 0.25       0.23500371
    ↪  0.19661193]
10  (this value should be about 0.196612 0.235004 0.250000 0.235004 0.196612)
11  """
```

Listing 4 – sigmoidGradient

3.2 Random initialization

Comme décrit dans le TP, il est important d'initialiser les paramètres de manière aléatoire pour casser la symétrie. On va donc appliquer la formule qui suit, de sorte à garder des paramètres petits pour rendre l'apprentissage plus efficace.

$$\epsilon_{\text{init}} = \frac{\sqrt{6}}{\sqrt{L_{\text{in}} + L_{\text{out}}}} \quad \text{avec} \quad L_{\text{in}} = S_l = 10 \quad \text{et} \quad L_{\text{out}} = S_{(l+1)} = 10 + 1 \tag{4}$$

Implémentation

```
1
2   def randInitializeWeights(L_in, L_out):
3
4       # Randomly initialize the weights to small values
5       epsilon_init = 0.12
6       W = np.random.rand(L_out, 1+L_in) * 2 * epsilon_init - epsilon_init
7
8       return W
```

Listing 5 – randInitializeWeights

3.3 Backpropagation algorithm

A partir de cette section, nous n'avons plus à coder puisque l'essentiel du code est déjà implémenté. Nous allons donc synthétiser ce que nous avons compris de cet algorithme et des sections qui suivent cette partie.

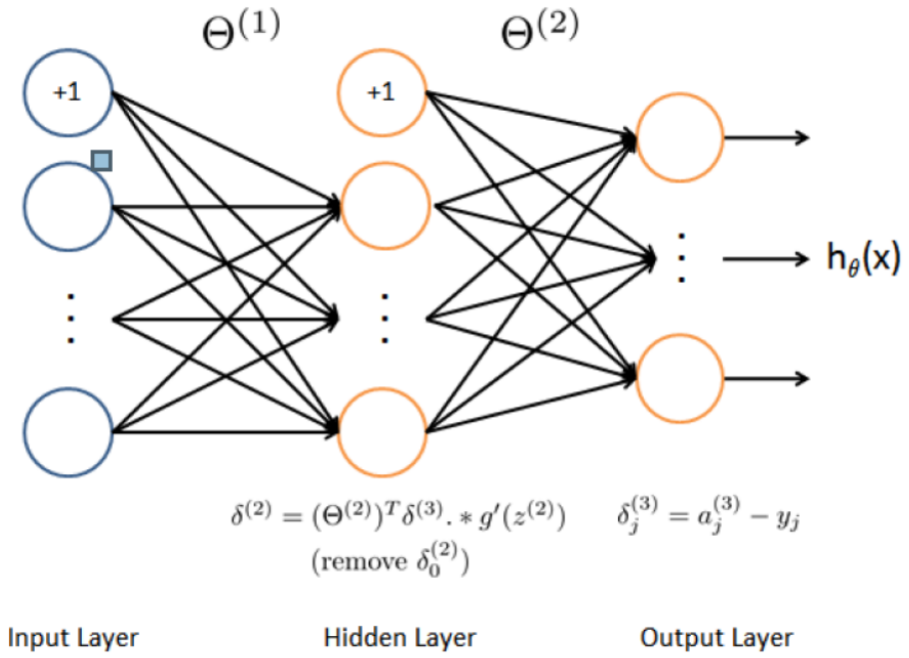

FIGURE 5 – Backpropagagtion updates

Le principe décrit est le suivant, au démarrage, les calculs génèrent une propagation avant jusqu'à la sortie et obtenir l'erreur $h_\theta((x))_k$. Le but principal est de réussir à évaluer quels noeuds va le plus influencer l'erreur de sortie. Pour obtenir l'erreur il suffit de faire la différence entre la sortie du réseau de neurone et les valeurs ciblées de Y. //

Pour arriver à un résultat correcte qui minimise l'erreur, il faut suivre quatre étapes pour chaque échantillons d'entrainement donnés. Ainsi, à chaque itération, en s'entrainant à chaque boucle, le réseau devient plus performant.

1. Propagation avant (Forward Pass)
2. Calcul de l'erreur de la couche de sortie
3. Calcul de l'erreur des couches cachées
4. Addition des gradients pour chaque échantillon
5. Normalisation des gradients

Une fois la dernière étape du dernier échantillon passé, les gradients moyens sont utilisés pour définir et surtout mettre à jour les poids du réseau afin de minimiser l'impacte des noeuds qui augmentent l'erreur.

3.4 Gradient checking

Dans cette section, l'objectif est de vérifier chacun de nos gradients préalablement calculés. Le code fourni dans (computeNumericalGradient.py) fonctionne de telle manière à venir tronquer les valeurs de θ. A chaque itération, une composante de θ est perturbée par ϵ puis le coût est réévalué. Pour finir, la formule du gradient numérique est utilisée pour évaluer chaque nouvelles compsantes du gradient numérique. Cette fonction retourne *numgrad* qui servira de comparateur avec les valeurs calculées par

notre algorithme de rétropropagation. Plus les écarts sont minimes, mieux l'algorithme se comporte et est bien entraîné.

Résultat du gradient checking

```
    """return
    Checking Backpropagation...
    [[-9.27825235e-03 -9.27825236e-03]
    [ 8.89911959e-03  8.89911960e-03]
    [-8.36010761e-03 -8.36010762e-03]
    [ 7.62813550e-03  7.62813551e-03]
    [-6.74798369e-03 -6.74798370e-03]
    [-3.04978709e-06 -3.04978914e-06]
    [ 1.42869450e-05  1.42869443e-05]
    [ 4.65597186e-02  4.65597186e-02]]
    The above two columns you get should be very similar.
    (Left-Your Numerical Gradient, Right-Analytical Gradient)

    If your backpropagation implementation is correct, then
    the relative difference will be small (less than 1e-9).

    Relative Difference: 2.4768e-11
    """
```

Listing 6 – Compute Numerical Gradient

Le résultat obtenu est satisfaisant puisque l'écart est inférieur à la différence relative annoncée.

3.5 Regularized Neural Networks

De la même manière que pour les autres TP, on vient régulariser le gradient. La rétropropagation nous a permis de calculer les gradients, le but ici est d'appliquer les formules données dans le sujet pour régulariser le réseau de neurone.

$$\frac{\partial J(\Theta)}{\partial \Theta_{ij}^{(l)}} = \frac{1}{m}\Delta_{ij}^{(l)} + \frac{\lambda}{m}\Theta_{ij}^{(l)} \quad \text{pour} \quad J = 0 \tag{5}$$

$$\frac{\partial J(\Theta)}{\partial \Theta_{ij}^{(l)}} = \frac{1}{m}\Delta_{ij}^{(l)} + \frac{\lambda}{m}\Theta_{ij}^{(l)} \quad \text{pour} \quad J >= 1 \tag{6}$$

Résultat du Regularized gradient

```
"""return
Checking Backpropagation (w/ Regularization) ...
[[-9.27825235e-03 -9.27825236e-03]
 [ 8.89911959e-03  8.89911960e-03]
 [-8.36010761e-03 -8.36010762e-03]
 [ 7.62813550e-03  7.62813551e-03]
 [-6.74798369e-03 -6.74798370e-03]
 [-1.67679797e-02 -1.67679797e-02]
 [ 3.94334829e-02  3.94334829e-02]
 [ 1.50048382e-03  1.50048382e-03]]
The above two columns you get should be very similar.
(Left-Your Numerical Gradient, Right-Analytical Gradient)


If your backpropagation implementation is correct, then
the relative difference will be small (less than 1e-9).

Relative Difference: 2.34399e-11
"""
```

Listing 7 – Regularized Gradient

Le résultat obtenu est satisfaisant puisque l'écart est inférieur à la différence relative annoncée.

3.6 Learning parameters using fmin

Avec la fonction `fmin_cg`, le système apprends un bon ensemble de paramètres. Ce qui permet de vérifier si l'entraînement de notre réseau de neurone est correcte par rapport à l'esemble d'entraînement.

Résultat

```
"""return
Visualizing Neural Network...
Training Set Accuracy: 96.000000
"""
```

Listing 8 – `fmin_cg` function

4 Visualising the hidden layer

L'objectif dans cette section est de visualiser les représentations capturées par les unités cachées. On voit que l'image génerée disposes de 25 images ressemblant aux chiffres et donc à des détecteurs qui recherchent des traits et d'autres motifs dans l'entrée.

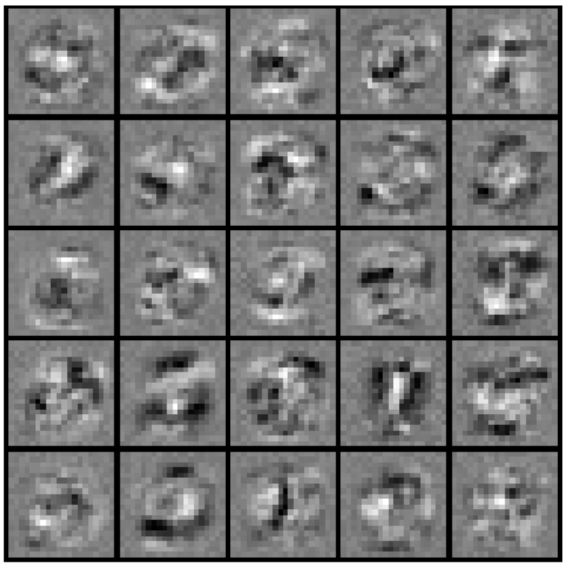

FIGURE 6 – Visualization of Hidden Units

5 Final analysis

Dans cette section, l'objectif est d'augmenter la valeur de *lambda* et de *MaxIter* pour essayer différentes configurations d'apprentissage du réseau neuronal et d'en évaluer la différence de performance.

Malheureusement pour cette partie, nos ordinateurs portable n'ont pas la capacité d'effectuer les calculs lorsque l'on incrémente MaxIter. On restera donc avec les résultats calculés préalablement.

6 Questions

1. **Décrivez l'approche basée sur les réseaux de neurones. Quelle modélisation est retenue ? Quels sont les paramètres à apprendre ?**

   Les réseaux de neurones fonctionnent avec plusieurs couches, la couche d'entrée, les couches cachées et la couche de sortie. Le but est toujours le même, favorisé l'apprentissage automatique de la machine. Les couches sont connectées entre elles. A chaque couche, des calculs sont effectués et les informations sont transmises aux couches suivantes. Elles sont composées de neurones qui correspondent en entrée aux caractéristiques des données que l'on inserent au réseau. Dans les couches cachées, les neurones apprennent des informations de plus en plus compliquées. La couche de sortie renvoit le résultat du réseau de neurone.

   Les paramètres en jeu dans cette approche sont les poids de connexions entre les couches et les biais. L'objectif dans un réseau de neurone est d'ajuster les poids et les biais pour minimiser le coût. Les poids correspondent à la connexion entre les neurones, dans le sens où l'information transmise entre deux neurones sera plus ou moins importante ou non pour le prochain neurone. Les biais quant à eux, influencent l'activation d'un neurone et permettent d'apprendre plus facilement des modèles complexes.

2. **Quelle est la fonction de coût ? Quelles modifications sont effectuées par rapport à celle utilisée en TP2 ?**

   La fonction de coût pour le réseau de neurone nous sert à évaluer la qualité de la prédiction. La différence ici, est qu'à l'aide de la rétropropagation, les paramètres du réseau vont être ajustés à chaque échantillon en fonction du poids calculé en sortie.

   Les différences entre les deux fonctions de coût des deux TP2 sont principalement dans la compléxité de la formule dû aux couches multiples du réseau de neurone. En effet, pour ce TP3, le réseau de neurone s'appuie sur deux θ_1 contre un seul dans le TP2. A la finale, l'objectif reste toujours le même, minimiser l'erreur de prédiction par rapport aux valeurs réelles attendues.