École Centrale Lyon

UE INF
TD6
Rapport

Réseaux de neurones pour la régression et la classification

Élèves :
Hugo PUYBAREAU

Enseignant :
Emmanuel Dellandrea

9 novembre 2023

Table des matières

1 Introduction

Le travail de ce devoir consiste à créer un modèle de réseau de neurones. Il est demandé de créer un modèle répondant à des problèmes de regression et un modèle répondant à des problèmes de classification. Une fois ces modèles créés, il est attendu de les tester avec différents jeux de données ainsi que d'étudier l'influence de leurs paramètres.

2 Modèle pour la régression

J'ai commencé par compléter le programme pour la régression.

2.1 Programmation du modèle

2.1.1 Découpage des données

Le découpage des données est fait à partir de la fonction *decoupage_ donnees* suivante :

```
def decoupage_donnees(x,d,prop_val=0.2, prop_test=0.2):
    """ Découpe les données initiales en trois sous-ensembles distincts d'apprentissage, de validation et de test

    indice_app = int(x.shape[0]*(1-prop_test-prop_val)) #Je crée l'indice d'apprentissage en fonction du ratio
    indice_val = indice_app + int(x.shape[0]*prop_val) #Je crée l'indice de validation en fonction du ratio
    indice_test = indice_val + int(x.shape[0]*prop_test) #Je crée l'indice en fonction du ratio

    x_app, x_val, x_test = x[0:indice_app,:], x[indice_app:indice_val,:], x[indice_val:indice_test+1,:] #Je crée les
    d_app, d_val, d_test = d[0:indice_app,:], d[indice_app:indice_val,:], d[indice_val:indice_test+1,:]

    return x_app, d_app, x_val, d_val, x_test, d_test
```

FIGURE 1 – Code de la fonction *decoupage_ donnees*

Les données sont de base réparties de la manière suivante :
— 60% pour les données d'apprentissage
— 20% pour les données de validation
— 20% pour les données de test

Après recherche, ces proportions sont justifiées : La proportion des données d'apprentissage devrait être la plus élevée par rapport aux autres ensembles, car c'est sur ces données que notre modèle apprendra les modèles et les relations. Une proportion typique se situe entre 60% et 80% de l'ensemble de données (On se rapproche de 80% si l'ensemble est très grand).

Les données de validation sont utilisées pour évaluer les performances du modèle pendant l'entraînement et pour ajuster les hyper-paramètres. Une proportion typique varie généralement entre 10% et 20% de l'ensemble de données.

Les données de test sont utilisées pour évaluer les performances finales du modèle sur des données inconnues, et elles sont généralement utilisées une seule fois après l'entraînement. La proportion typique est généralement d'environ 20% de l'ensemble de données.

2.1.2 Calcul du coût

Le calcul du coût se fait à partir de la fonction *calcul_ cout_ mse* suivante (formule page 16 du cours) :

```python
def calcule_cout_mse(y, d):
    """ Calcule la valeur de la fonction cout MSE (moyenne des différences au carré)
    cout=0

    for i in range(len(y)) :
        cout += (y[i]-d[0][i])**2
    return cout/2
```

FIGURE 2 – Code de la fonction *calcul_ cout_ mse*

Les variables de coûts associées aux différents jeux de données sont initialisées comme des listes possédant autant d'éléments qu'il y a d'itérations d'entraînement.

2.1.3 Passage en avant

Le code de la fonction *passe_ avant* est le suivant :

```python
def passe_avant(x, W, b, activation):
    """ Réalise une passe avant dans le réseau de neurones

    Parametres
    ----------
    x : matrice des entrées, de dimension nb_var x N
    W : liste contenant les matrices des poids du réseau
    b : liste contenant les matrices des biais du réseau
    activation : liste contenant les fonctions d'activation des couches du réseau

    avec N : nombre d'éléments, nb_var : nombre de variables prédictives

    Return
    -------
    a : liste contenant les potentiels d'entrée des couches du réseau
    h : liste contenant les sorties des couches du réseau

    """

    h = [x]
    a = []
    for i in range(len(b)):
        a_i = np.dot(W[i], h[i]) + b[i]
        h_i = activation[i](a_i, False)
        a.append(a_i)
        h.append(h_i)
    return a, h
```

FIGURE 3 – Code de la fonction *passe_ avant*

Cette fonction est utilisée de la manière suivante dans le code pour la régression :

```
y_app = []
y_val = []

for k in range(len(x_app[0])) :
    ##########################################################################
    # Passe avant : calcul de la sortie prédite y sur les données d'apprentissage #
    ##########################################################################
    a, h = passe_avant(x_app[0][k], W, b, activation)
    y_app.append(h[-1]) # Sortie prédite
```

FIGURE 4 – Insertion de la fonction *passe_ avant* dans le code pour la régression

Le code fournit sur Moodle a été un peu modifié car je n'arrivais pas à obtenir des variables de mêmes tailles en sortie de mes fonctions *passe_ avant* et *passe_ arriere*.

L'algorithme de la fonction *passe_ avant* est tiré de cette partie du cours :

Forward pass

Initialization:

$\boldsymbol{h}_0 = \boldsymbol{x}$

for layer $k = 1$ **to** L **do**

  Linear unit:

  $\boldsymbol{a}_k = \boldsymbol{W}_k \boldsymbol{h}_{k-1} + \boldsymbol{b}_k$

  Componentwise non-linear activation:

  $\boldsymbol{h}_k = g_k(\boldsymbol{a}_k)$

end

Output layer:

$\boldsymbol{y} = \boldsymbol{h}_L$

Compute loss:

$E = L(\boldsymbol{y}; \boldsymbol{d})$

FIGURE 5 – Algorithme utilisé dans la fonction *passe_ avant*

2.1.4 Passage en arrière

Le code de la fonction *passage_arriere* est le suivant :

```
def passe_arriere(delta_h, a, h, W, activation):
    """ Réalise une passe arrière dans le réseau de neurones (rétropropagation)

    Parametres
    ----------
    delta_h : matrice contenant les valeurs du gradient du coût par rapport à la sortie du réseau
    a : liste contenant les potentiels d'entrée des couches du réseau
    h : liste contenant les sorties des couches du réseau
    W : liste contenant les matrices des poids du réseau
    activation : liste contenant les fonctions d'activation des couches du réseau

    Return
    -------
    delta_W : liste contenant les matrice des gradients des poids des couches du réseau
    delta_b : liste contenant les matrice des gradients des biais des couches du réseau

    """

    delta_b = [np.zeros(k.shape) for k in b]
    delta_W = [np.zeros(w.shape) for w in W]

    for i in range(len(D_c)-1):
        delta_W.append(2 * np.random.random((D_c[i+1], D_c[i])) - 1)
        delta_b.append(np.zeros((D_c[i+1],1)))

    for i in range(len(W)-1, -1, -1):
        delta = delta_h * activation[i](a[i], True)
        delta_b[i]=delta
        delta_W[i]=np.dot(delta, h[i].T)
        delta_h = np.dot(W[i].T, delta)

    return delta_W, delta_b
```

FIGURE 6 – Code de la fonction *passe_arriere*

Cette fonction n'est utilisé que pour le set d'apprentissage car elle sert uniquement à améliorer le biais b et les matrices de poids W :

```
    for k in range(len(x_app[0])) :
        ##########################################################################################
        # Passe avant : calcul de la sortie prédite y sur les données d'apprentissage #
        ##########################################################################################
        a, h = passe_avant(x_app[0][k], W, b, activation)
        y_app.append(h[-1]) # Sortie prédite

        ########################################
        # Passe arrière : rétropropagation #
        ########################################
        delta_h = (y_app[k]-d_app[0][k]) # Pour la dernière couche
        delta_W, delta_b = passe_arriere(delta_h, a, h, W, activation)

        ################################################
        # Mise à jour des poids et des biais  ########
        ################################################
        for i in range(len(b)-1,-1,-1):
            b[i] -= alpha * delta_b[i]
            W[i] -= alpha * delta_W[i]
```

FIGURE 7 – Insertion de la fonction *passe_arriere* dans le code pour la régression

L'algorithme est tiré de cette partie du cours :

Backward pass

Initialization: Gradient of output layer:

$\nabla_{\boldsymbol{h}_L} E = \nabla L(\boldsymbol{y}; \boldsymbol{d})$

for layer $k = L$ **to** 1 **do**

Componentwise gain of error:

$\boldsymbol{\delta}_k = \nabla_{\boldsymbol{a}_k} E = \nabla_{\boldsymbol{h}_k} E \odot g_k'(\boldsymbol{a}_k)$

Gradient of layer bias:

$\nabla_{\boldsymbol{b}_k} E = \boldsymbol{\delta}_k$

Gradient of weights:

$\nabla_{\boldsymbol{W}_k} E = \boldsymbol{\delta}_k \boldsymbol{h}_{k-1}^T$

Gradient of previous hidden layer:

$\nabla_{\boldsymbol{h}_{k-1}} E = \boldsymbol{W}_k^T \boldsymbol{\delta}_k$

end

FIGURE 8 – Algorithme utilisé dans la fonction *passe_arriere*

2.2 Critique et analyse

J'ai tout d'abord fait l'entraînement du modèle pour la régression avec le jeu de données *food_truck.txt*. J'ai analysé l'influence des hyper-paramètres avec les mêmes fonctions que celles données dans le modèle du cours pour la régression : *relu* pour les couches cachées et *lineaire* pour la couche de sortie.

Pour 5 neurones, une couche et 50 itérations, j'obtiens un coût qui décroit entre les données d'apprentissage, validation et test, ce qui est satisfaisant, cependant j'obtiens ces convergences :

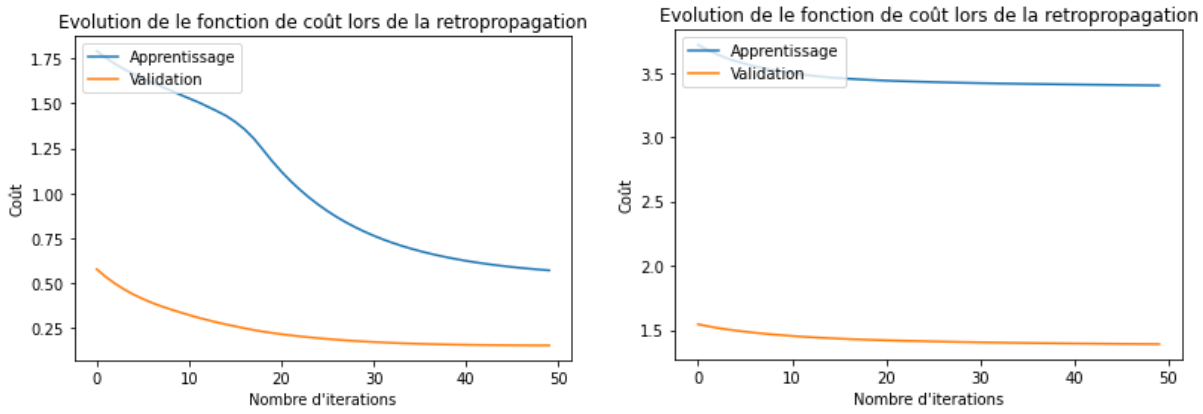

FIGURE 9 – Convergences des coûts pour ces paramètres

Il est donc intéressant de rajouter des itérations, des couches ou des neurones pour obtenir des convergences plus propres :

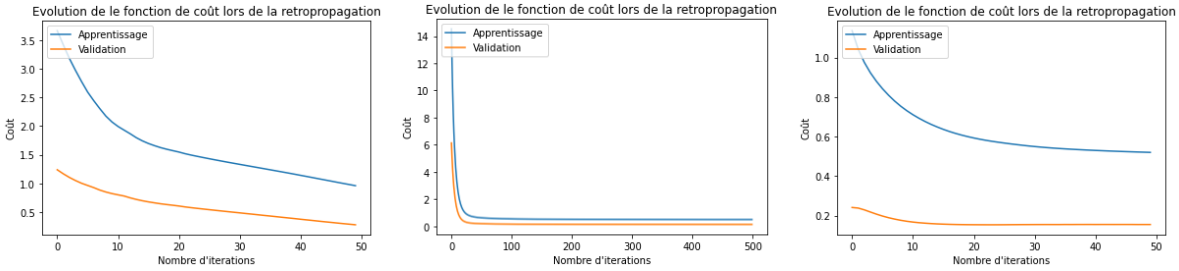

FIGURE 10 – Convergences des coûts pour 10 couches de 5 neurones et 50 itérations, 1 couche de 5 neurones et 500 itérations et pour 1 couche de 50 neurones et 50 itérations

En multipliant chacune des valeurs par 10 je considère pouvoir conclure sur l'affluence de paramètres. Il est évident que c'est le nombre d'itérations du modèle qui affût le plus sur sa convergence et pas forcément le nombre de couches ou de neurones.

Je conserverai à partir de maintenant le nombre d'itérations à 500, le nombre de couches à 5 et le nombre de neurones à 50. En faisant évoluer le taux d'apprentissage α, j'obtiens les convergences suivantes :

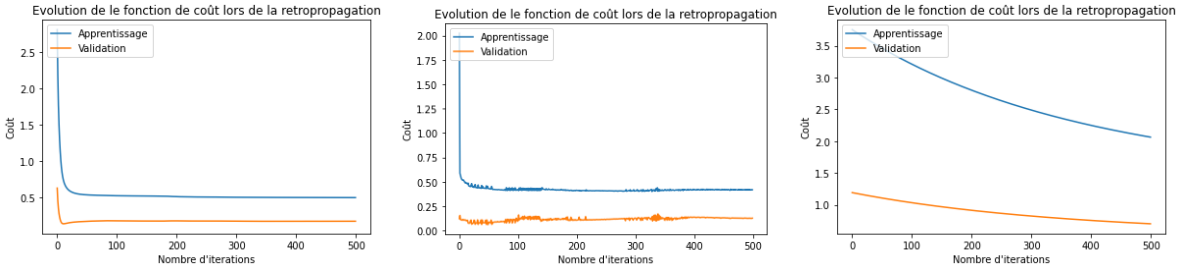

FIGURE 11 – Convergences des coûts pour les taux d'apprentissages 0.001, 0.1 et 0.00001

Je remarque ainsi que le taux d'apprentissage influe très fortement la convergence du modèle. En effet, Un taux d'apprentissage trop faible peut ralentir la convergence du modèle, ce qui signifie que le modèle prendra plus de temps pour atteindre un état où il produit des prédictions satisfaisantes. D'autre part, un taux d'apprentissage trop élevé peut entraîner des oscillations et une instabilité dans la convergence.

Enfin j'obtiens cette courbe là pour le set de validation, il est compliqué de conclure quoi que ce soit dessus car les valeurs ont été normalisées.

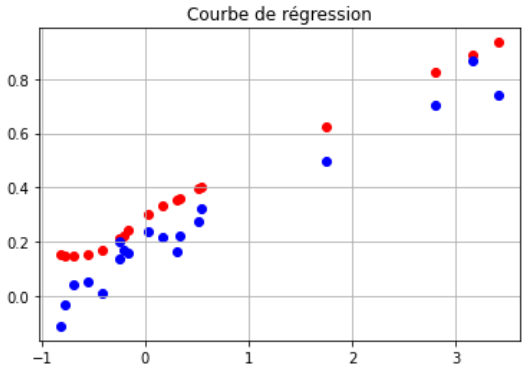

FIGURE 12 – Tracé de la régression pour les données de validation

3 Modèle pour la classification

Je crée maintenant le deuxième modèle en modifiant le code de celui pour la régression.

3.1 Programmation du modèle

Seule la fonction de calcul de coût va changer. Les autres fonctions sont identiques. Il faut également ajouter le code de la fonction *softmax* car elle sera utilisée pour l'activation de la sortie.

3.1.1 Fonction *softmax*

Le code rédigé est le suivant :

```python
def softmax(x, deriv=False) :
    """

    Calcule la valeur de la fonction softmax ou de sa dérivée appliquée à z.

    Parametres
    ----------
    z : peut être un scalaire ou une matrice
    k : l'indice de la probabilité calculé
    deriv : booléen. Si False renvoie la valeur de la fonction linéire, si True renvoie sa dérivée

    Return
    -------
    s : valeur de la fonction softmax appliquée à z ou de sa dérivée. Même dimension que z

    """

    if deriv :
        s = softmax(x)
        return s * (1 - s)
    else :
        exp_x = np.exp(x-max(x)) #je mets ca sinon ca fait que diverger
        # Calcul de la somme des exponentielles pour chaque ligne
        sum_exp_x = np.sum(exp_x)
        # Calcul du softmax pour chaque ligne
        return exp_x / sum_exp_x
```

FIGURE 13 – Code de la fonction *softmax*

3.1.2 Calcul du coût

Le code de la fonction de calcul de coût est le suivant :

```python
def calcule_cout_mse(y, d):
    """ Calcule la valeur de la fonction cout MSE (moyenne des différences au carré)

    Parametres
    ----------
    y : matrice des données prédites
    d : matrice des données réelles

    Return
    ------->
    cout : nombre correspondant à la valeur de la fonction cout (moyenne des différences au carré)

    """
    cout = 0

    for i in range (len(y)):
        cout = -np.sum(d[i][0] * np.log(y[i]))
    return cout
```

FIGURE 14 – Calcul de coût avec la formule de l'entropie croisée

La formule est tirée de celle donnée dans le cours :

$$E(\boldsymbol{W}) = -\sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i)} \sum_{k=1}^{K} d_k^i \log y_k^i$$

FIGURE 15 – Formule pour l'entropie croisée dans un problème multi-classes

Je suis incapable de faire converger le coût dans mon code. Je ne comprends pas pourquoi, cela fait plus d'une semaine que je n'arrive pas à faire marcher la classification.