Université de Montréal

PHY 3075 – Modélisation numérique en physique

Projet 6 - Réseaux de neurones artificiels et détection du boson de Higgs

---

par :

Patrice Béchard

20019173

26 avril 2017

1 Introduction

Dans un monde où l'internet des choses gagne de plus en plus de terrain et où une cascade continue de données est produite (données météorologiques, données sur les intérêts d'utilisateurs d'un certain produit, données en bourse, etc.), il devient impossible pour l'humain d'essayer d'analyser et de comprendre celles-ci, vu leur nombre important. Le développement d'algorithmes d'apprentissage permettant à l'ordinateur de classifier des données ou d'en extrapoler des valeurs devient primordial. L'apprentissage machine est un domaine en plein essor permettant à un ordinateur d'*apprendre à apprendre*. L'une des branches de ce domaine est l'apprentissage supervisé, consistant à fournir à l'ordinateur une ensemble de données étiquetés pour lesquelles la réponse est connue. Par des techniques d'optimisation, il sera possible de minimiser la valeur d'une fonction de coût caractérisant l'écart entre la prédiction faite par l'ordinateur et la bonne réponse lui étant fournie. Il est ensuite possible de fournir à l'ordinateur un nouvel ensemble de données, cette fois-ci non étiquetée, et, si l'ordinateur est bien entraîné, pourra généraliser de ses expériences antérieures ainsi prédire la solution.

Les réseaux de neurones sont sont une catégorie d'algorithmes d'apprentissage faisant propager un signal d'entrée de noeud en noeud pour en produire une sortie. Si le résultat de la sortie s'écarte du résultat attendu, le réseau ajustera ses poids internes de sorte à produire une sortie se rapprochant de la solution attendue le plus possible. Une représentation visuelle d'un réseau de neurone artificiel est présenté à la figure 1. Les réseaux de neurones les plus simples sont appelés *feed forward neural networks*, puisque l'information ne s'y propage que dans un sens, soit de l'entrée à la sortie, pour produire un résultat.

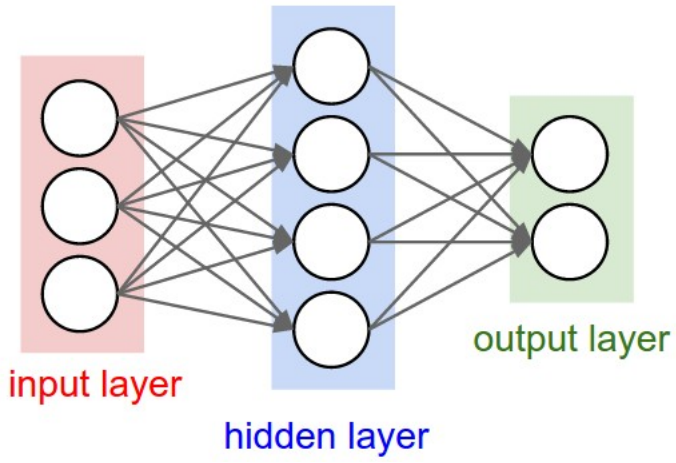

FIGURE 1 – Représentation graphique d'un réseau de neurones *feed forward* possédant une couche interne[6].

Il est possible d'appliquer des algorithmes d'apprentissage à plusieurs problèmes physiques. L'une des applications la plus intéressante est la détection de nouvelles particules élémentaires via les données produites par le CERN. Ce projet consiste à implémenter et entraîner un réseau de neurones artificiels

pouvant détecter le boson de Higgs avec la meilleur efficacité possible. La première section de ce présent rapport présentera une validation du code implémenté en appliquant l'algorithme sur un problème plus simple. Une présentation des résultats obtenus pour la détection du boson de Higgs sera ensuite faite.

2 Validation du code

La validation du code construit a été faite en entrainant le réseau de neurones artificiel créé sur avec un échantillon de 1000 séries de 12 bits consécutifs et en le testant sur un autre échantillon de même taille. Le but était d'entraîner le réseau à pouvoir distinguer une série de 5 bits consécutifs ayant la valeur de 1. Cette même expérience a été réalisée dans les notes de cours (les bits étaient identifiés comme blanc(0) ou noir(1)) [1].

Les échantillons ont été construits en générant une suite de 12 chiffres possédant une valeur $\in \{0, 1\}$ aléatoirement. Avec un algorithme *force brute*, il était possible de voir si la série de bits possédait bel et bien une série de 5 bits consécutifs possédant une valeur de 1, pour ainsi pouvoir entraîner le réseau de façon supervisée. Trois fonctions d'activation distinctes ont été testées pour voir l'influence sur la sortie du réseau. Celles-ci sont la fonction *logistique*, la fonction *tangente hyperbolique* (tanh) et la fonction *Unité de Rectification Linéaire* (ReLU) et leurs caractéristiques sont présentées au tableau 1.

Nom	Équation	Dérivée	Codomaine
logistique	$f(x) = \frac{1}{1-\mathrm{e}^{-x}}$	$f'(x) = f(x)(1 - f(x))$	$]0, 1[$
tanh	$f(x) = \tanh(x)$	$f'(x) = 1 - f(x)^2$	$]-\frac{\pi}{2}, \frac{\pi}{2}[$
ReLU	$f(x) = \begin{cases} 0 \text{ si } x < 0 \\ x \text{ si } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 \text{ si } x < 0 \\ 1 \text{ si } x \geq 0 \end{cases}$	$[0, \infty[$

TABLE 1 – Caractéristiques principales des fonctions d'activation utilisées pour la confection du réseau de neurones artificiel [5].

Le réseau a été construit de sorte qu'il soit facile de modifier le nombre de noeuds dans chaque couche ainsi que d'ajouter et d'enlever des couches. Pour la vérification du code, la couche d'entrée était composée de 12 noeuds, soit le nombre de bits en entrée. La couche de sortie est composée de 2 noeuds. Pour faire la décision, le réseau prend le noeud de sortie pour lequel la valeur est maximale. Le choix du noeud 0 correspond à l'absence de 5 bits "1" consécutifs, alors que le noeud 1 correspond à la présence de ce regroupement. Le réseau ne comprends qu'une couche interne de 6 noeuds pour l'étape de la validation, ne nécessitant pas plus pour cette tâche simple.

L'évolution de la fonction de coût selon le nombre d'itérations effectuées est présentée à la figure 2

pour chacune des fonctions d'activation testées. Le tout a été effectué avec un paramètre d'entraînement $\alpha = 0.01$ et la méthode d'optimisation par descente de gradient. L'entraînement a été effectuée en *batch* de 20 éléments, voulant dire que la descente de gradient utilise 20 échantillons pour décider de la direction vers laquelle progresser.

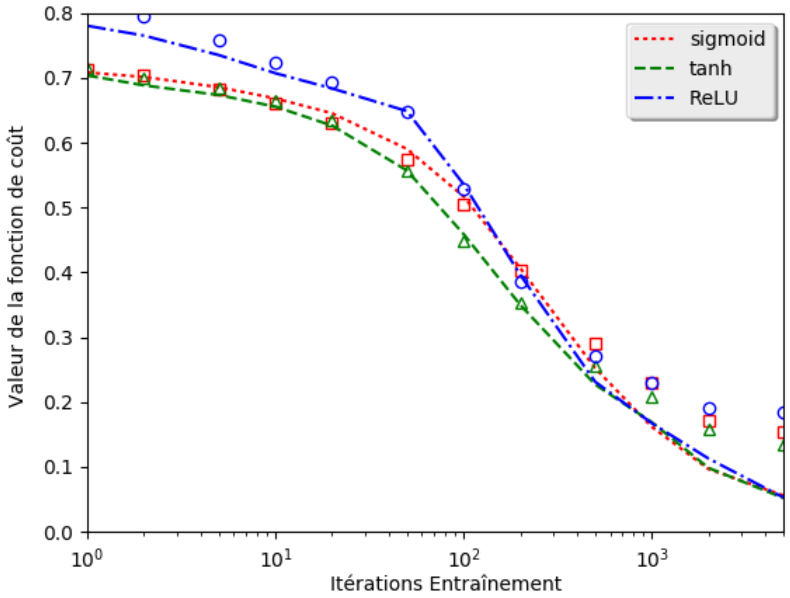

FIGURE 2 – Évolution de la valeur de la fonction de coût selon le nombre d'itérations effectué pour chaque fonction d'activation présentée au tableau 1. Le paramètre d'entraînement a été fixé à 0.01 et la méthode de descente de gradient a été utilisée pour minimiser la fonction de coût. Les courbes présentent les valeurs pour l'échantillon d'entraînement, et les marqueurs représentent les valeurs pour l'échantillon test.

La différence entre les résultats produits par les différentes fonctions d'activation sont petites pour cette exemple. La fonction d'activation ReLU possède une plus grande erreur au départ, mais ceci n'est seulement dû aux configurations initiales des matrices de poids, choisis aléatoirement selon une distribution gaussienne. Toutes les fonctions d'activation convergent vers la même valeur finale, soit environ 0.05 à 5000 itérations. Le choix de la fonction d'activation dans ce cas ne semble donc pas changer énormément l'algorithme de minimisation de la fonction de coût. La fonction d'activation sigmoïdale sera utilisée pour le reste de ce projet par choix arbitraire.

La rétropropagation a été faite avec plusieurs fonctions d'optimisation pour voir laquelle était optimale. Celles-ci sont l'algorithme de descente de gradient, l'algorithme RMSProp [4], l'algorithme Adam [7], ainsi que l'algorithme AdaGrad (Gradient adaptif) [2]. Ces fonctions sont directement implantées dans *Tensorflow*, une bibliothèque open-source pour l'apprentissage automatique disponible sur Python. L'évolution de la fonction de coût selon le nombre d'itérations effectuées est présentée à la figure 3 pour chacune des méthodes d'optimisation étudiées. Encore une fois, le paramètre d'entraînement a été choisi

à 0.01 et des *batch* des 20 éléments ont été utilisées pour l'optimisation. La fonction d'activation utilisée pour cette comparaison est la fonction sigmoïdale.

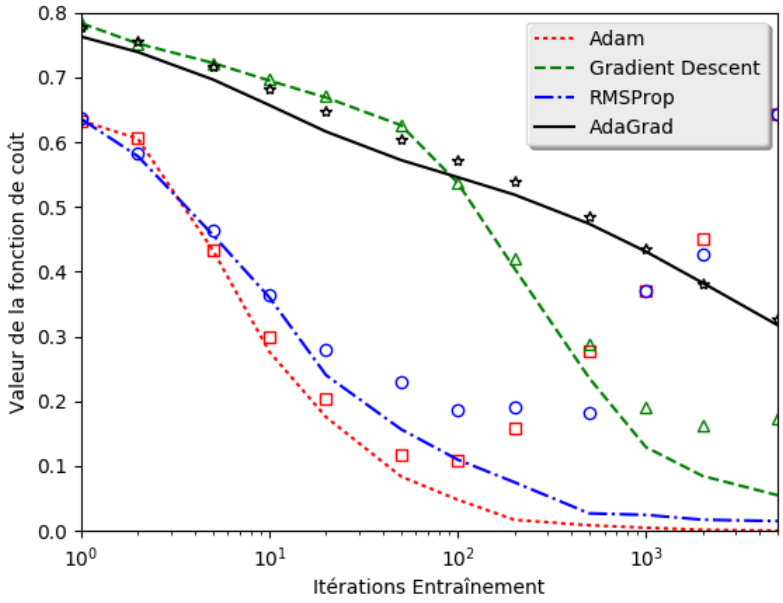

FIGURE 3 – Évolution de la valeur de la fonction de coût selon le nombre d'itérations effectué pour chaque algorithme d'optimisation étudiée. Le paramètre d'entraînement a été fixé à 0.01 et la fonction d'activation sigmoïdale a été utilisée pour cette comparaison. Les courbes présentent les valeurs pour l'échantillon d'entraînement, et les marqueurs représentent les valeurs pour l'échantillon test.

La fonction d'optimisation utilisant l'algorithme d'Adam semble être la meilleure parmi les quatre algorithmes testés pour converger rapidement vers une solution. Cependant, il faut faire attention, puisque le réseau de neurones entre rapidement en régime de surentraînement, alors que la valeur de la fonction de coût appliquée à l'échantillon d'entraînement continue de diminuer, mais que celle reliée à l'échantillon de test commence à augmenter. Le même scénario se produit pour l'algorithme RMSProp, qui converge seulement un peu plus lentement que l'algorithme d'Adam. La descente de gradient diminue selon un profil similaire à celui trouvé à la figure 2, qui est cependant beaucoup plus lent que les deux algorithmes mentionnés plus tôt. Ce n'est donc pas un algorithme intéressant pour la suite des choses. Finalement, l'algorithme AdaGrad converge très lentement vers le minimum global, alors qu'à a fin de la simulation, la valeur de la fonction de coût associée aux exemplaires d'entraînement est encore nettement supérieure à celle trouvée avec les autres algorithmes. L'algorithme ne sera donc pas utilisé dans la suite des choses.

Maintenant que le réseau de neurones fonctionne bel et bien et peut apprendre d'exemples, tout en généralisant, il est possible de l'entraîner pour le problème de détection du boson de Higgs, qui constitue le sujet de ce projet.

3 Résultats et discussion

Après avoir vérifié que le code était bel et bien correct, il nous reste à entraîner et tester le réseau sur l'échantillon d'intérêt, soit les données du CERN pour la détection du boson de Higgs. Cependant, il n'est pas possible de seulement prendre les mêmes paramètres et la même architecture de réseau pour cette étape. La couche d'entrée du réseau possédait cette fois-ci 13 neurones, et la couche de sortie en possédait 2. Deux couches internes ont été utilisées pour l'entraînement de ce réseau. Ceux-ci ont été ajustés à 10 et 8 neurones, respectivement. Ces valeurs ont été trouvées par tâtonnement ainsi qu'en suivant quelques règles générales présentées par Jeff Heaton dans son livre *Introduction to neural networks with Java*[3]. Une autre solution serait d'utiliser des algorithmes de *pruning*, permettant de soustraire les neurones redondants dans les couches externes[9]. Cette méthode n'a cependant pas été implémentée pour le présent projet.

La fonctions d'activation ReLU a été utilisée pour entraîner le réseau. Ce type de fonction d'activation sont souvent utilisés dans la confection de réseaux de neurones profonds [8]. L'algorithme d'optimisation utilisée est l'algorithme d'Adam, qui semblait fonctionner très bien lors des tests à la section précédente.

Pour cette partie, le paramètre d'apprentissage α est fixé à 0.0001 de sorte que l'optimisation se face plus lentement, mais plus précisément. Avec $\alpha = 0.01$ comme à la section précédente, la valeur de la fonction de coût oscille de façon importante et le minimum est donc difficile à trouver. De plus, lorsque $\alpha = 0.001$, l'algorithme a de la difficulté à trouver le minimum, comme dans le cas précédent. La taille des *batch* à passer en même temps dans le réseau est de 20, comme à la section précédente. Le réseau a été entraîné sur 50 000 itérations et a obtenu une moyenne de 75% d'efficacité sur les différents exemplaires du fichier test. Aucun biais n'a été introduit, puisque la répartition entre les vrais positifs, faux négatifs, vrai négatifs et faux positifs était équilibrée.

Dû à la grande non-linéarité de la distribution que l'on souhaite modéliser, l'algorithme d'apprentissage a beaucoup plus de difficulté à minimiser la fonction de coût de sorte à pouvoir prédire efficacement si il y a présence du boson de Higgs sur un exemplaire à tester. La phase d'entraînement a été effectuée sur 1500 exemplaires de la base de donnée, étant interchangés aléatoirement à chacune des 50 000 itérations, de sorte que l'algorithme ne *mémorise* pas les exemplaires. La minimisation de la fonction de coût est beaucoup moins lisse qu'elle ne l'était pour l'exemple de la chaine de 12 bits traitée dans la section précédente. L'évolution de cette minimisation est présentée à la figure 4

On voit sur la figure qu'il aurait été judicieux d'arrêter l'entraînement vers 15000 itérations, puisqu'après, la fonction de coût pour l'échantillon test recommence à augmenter. Des résultats légèrement meilleurs auraient alors été obtenus.

À la fin de la simulation, la précision obtenue sur l'échantillon d'entraînement était d'approxima-

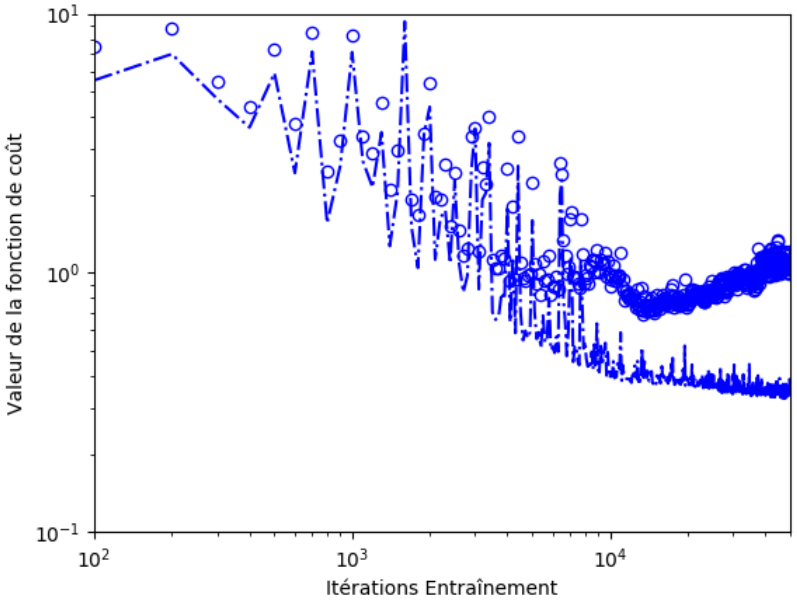

FIGURE 4 – Évolution de la valeur de la fonction de coût selon le nombre d'itérations effectué pour la détection du boson de Higgs. La fonction d'activation ReLu et l'algorithme d'Adam ont été utilisés. La courbe présente les valeurs pour l'échantillon d'entraînement, et les marqueurs représentent les valeurs pour l'échantillon test.

tivement 80% et celle sur l'échantillon test, d'environ 75%. Cette configuration a été préférée à celle où l'optimisation se faisait avec des *batch* volumineuses (500 éléments), retournant une précision bien meilleure sur l'échantillon d'entraînement (environ 90%), mais étant difficilement capable de généraliser, avec une précision sur l'échantillon test se trouvant entre 65% et 70%.

4 Programme

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL']='2'      #no warning messages

import tensorflow as tf                     # ML library
import numpy as np
import copy
import sys
import matplotlib.pyplot as plt

#------------------------------Global variables--------------------------------

higgs = True                            #True : Higgs detection, False : 12 bits string
exam = True                             #True : applying on exam set
actv = 2                                #0: logistic, 1: tanh, 2: ReLU
optim_function = 0      #0: AdamOptimizer, 1: Gradient descent, 2: RMS Prop, 3: AdaGrad
nIter = 50000
batch_size = 50

RANDOM_SEED = 42
tf.set_random_seed(RANDOM_SEED)

if higgs:                               #we train the network on the higgs data
    filetraining = 'trainingset.txt' #training set
    nTraining = 1500
    nTest = 2000 - nTraining
    nNeurons = [13,10,8,2]              #architecture of network
    learningrate = 0.00001
else:                                   #we train the network on strings of bits
    filetraining = 'bitstrings_train.txt' #training set
    nTraining = 1000
    filetest = 'bitstrings_test.txt'  #test set
    nTest = 1000

    nNeurons = [12,6,2]                 #architecture of network
    learningrate = 0.01
```

```
nLayers = len(nNeurons)

#--------------------------------Functions-------------------------------------

def init_weights(shape):
    """ Weight initialization """
    return tf.Variable(tf.random_normal(shape))

def model(x,wMatrix,p_keep_input,p_keep_hidden):
    """Propagation of input through the neural network"""
    x = tf.nn.dropout(x,p_keep_input) #we keep only with some probability
    h = [[] for i in range(nLayers-1)]
    h[0] = copy.copy(x)              #sentinel
    for i in range(1,nLayers-1):
        if actv == 0:                #logistic
            h[i] = tf.sigmoid(tf.matmul(h[i-1],wMatrix[i-1]))
        elif actv == 1:              #tanh
            h[i] = tf.tanh(tf.matmul(h[i-1],wMatrix[i-1]))
        elif actv == 2:              #relu
            h[i] = tf.nn.relu(tf.matmul(h[i-1],wMatrix[i-1]))
        h[i] = tf.nn.dropout(h[i],p_keep_hidden) #probability of keeping change

    return tf.matmul(h[-1],wMatrix[-1]) #return output

def init_bits(file,nSample):
    """Initialization of 12 bits data from txt file"""
    f = open(file)
    f.readline()
    tset = np.zeros([nSample,nNeurons[0]])
    oset = np.zeros([nSample,nNeurons[-1]])
    for i in range(nSample):
        temp = f.readline().strip().split()
        tset[i] = np.array(list(temp[0]),dtype=int)
        if int(temp[-1]):           # consecutive bit strings
            oset[i] = [0,1]
        else:
            oset[i] = [1,0]
    return tset,oset
```

```
def init_set(f,nSample,exam=False):
    """Initialize␣training␣set␣and␣answers␣for␣supervized␣learning"""
    tset = np.zeros([nSample,nNeurons[0]])       #training set empty array
    oset = np.zeros([nSample,nNeurons[-1]])      #answers for training set
    for i in range(nSample):
        temp = f.readline().strip().split()
        if exam:
            tset[i] = [float(elem) for elem in temp]
        else:
            tset[i] = [float(elem) for elem in temp[:-1]]
            temp[-1]=int(temp[-1])
            if temp[-1]:
              oset[i] = [1,0]
            else:
              oset[i] = [0,1]
    #normalisation of values
    #for i in range(13):
    #    tset[:,i] /= abs(max(max(tset[:,i]), min(tset[:,i]), key=abs))
    return tset,oset

def main(optim_function,learningrate):
    if higgs:
        """Higgs␣detection"""
        f = open(filetraining)
        f.readline()
        train_x, train_y = init_set(f,nTraining) #initializing data
        test_x, test_y = init_set(f,nTest)
        if exam:
            fileexam = 'examset.txt'             #exam set
            g = open(fileexam)
            g.readline()
            nExam = 1000
            exam_x, exam_y = init_set(g,nExam,exam=True) #init exam data
    else:
        """5␣straight␣in␣12␣bits"""
        train_x, train_y = init_bits(filetraining,nTraining)
        test_x, test_y = init_bits(filetest,nTest)
```

```
costtrain, costtest = [], []                #empty arrays to plot later

# Symbols
x = tf.placeholder("float", [None,nNeurons[0]])
y = tf.placeholder("float", [None, nNeurons[-1]])

# Weight initializations
wMatrix = [init_weights([nNeurons[i],nNeurons[i+1]])
                                for i in range(nLayers-1)]

# Forward propagation
p_keep_input = tf.placeholder("float")
p_keep_hidden = tf.placeholder("float")
py_x = model(x,wMatrix,p_keep_input,p_keep_hidden)

# Backward propagation
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits
                                        (labels=y, logits=py_x))
#cost = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits
#                                        (labels=y, logits=py_x))

if optim_function == 0:  #adam optimizer
    updates = tf.train.AdamOptimizer(learningrate).minimize(cost)
elif optim_function == 1: #gradient descent optimizer
    updates = tf.train.GradientDescentOptimizer(learningrate).minimize(cost)
elif optim_function == 2: #RMSProp optimizer
    updates = tf.train.RMSPropOptimizer(learningrate).minimize(cost)
elif optim_function == 3: #AdaGrad optimizer
    updates = tf.train.AdagradOptimizer(learningrate).minimize(cost)
predict = tf.argmax(py_x,1)

with tf.Session() as sess:              #launch session
    sess.run(tf.global_variables_initializer()) #init all variables
    #saver = tf.train.Saver()

    for epoch in range(nIter+1):
        # Train with each example
```

```
order = np.random.permutation(np.arange(nTraining)) #shuffling data
i = 0
while i < len(order):        #updating neural net
    start = i
    end = i + batch_size
    X = np.array(train_x[order[start:end]])
    Y = np.array(train_y[order[start:end]])
    sess.run(updates, feed_dict={x: X, y: Y,
                  p_keep_input: 1, p_keep_hidden: 1})
    i += batch_size

if epoch % 100 == 0:         #showing relevant info
    results_train = sess.run(predict, feed_dict={x: train_x,
                      p_keep_input: 1., p_keep_hidden: 1.})
    results_test = sess.run(predict, feed_dict={x: test_x,
                      p_keep_input: 1., p_keep_hidden: 1.})
    train_accuracy = np.mean(np.argmax(train_y, axis=1) == results_train)
    test_accuracy = np.mean(np.argmax(test_y, axis=1) == results_test)
    results_matrix = np.zeros([2,2])
    for k in range(nTraining):
        if np.argmax(train_y[k]):
            if results_train[k]: #vrai positif
                results_matrix[1][1] += 1
            else:                #faux negatif
                results_matrix[1][0] += 1
        else:
            if results_train[k]: #faux positif
                results_matrix[0][1] += 1
            else:                #faux negatif
                results_matrix[0][0] += 1

    print("Epoch = %d, train accuracy = %.2f%%, test accuracy = %.2f%%"
         % (epoch , 100. * train_accuracy, 100. * test_accuracy))
    costing = sess.run(tf.reduce_mean(cost),feed_dict=
        {x: train_x, y: train_y,p_keep_input: 1., p_keep_hidden: 1.})
    print("cost function value :",costing)
    print(results_matrix)
```

```
            costtrain.append(costing)        #to plot later
            costtest.append(sess.run(tf.reduce_mean(cost),feed_dict={x: test_x,
                y: test_y, p_keep_input: 1., p_keep_hidden: 1.}))

        if epoch == nIter and exam:      #evaluating exam set
            results_train = sess.run(predict, feed_dict={x: train_x,
                                p_keep_input: 1., p_keep_hidden: 1.})
            results_test = sess.run(predict, feed_dict={x: exam_x,
                                        p_keep_input: 1., p_keep_hidden: 1.})
            train_accuracy = np.mean(np.argmax(train_y, axis=1) == results_train)
            print("Epoch = %d, train accuracy = %.2f%%"
                 % (1, 100. * train_accuracy))
            costing = sess.run(tf.reduce_mean(cost),feed_dict=
                {x: train_x, y: train_y,p_keep_input: 1., p_keep_hidden: 1.})
            print("cost function value : ",costing)
            print(results_matrix)
            h = open('results.txt','w')  #saving results
            for i in range(nExam):
                h.write(str(results_test[i])+'\n')

    #saver.save(sess,"/tmp/higgs_ffnn.ckpt")
    return costtrain,costtest

#-------------------------MAIN-------------------------------------------------

ls = ['r:','g--','b-.','k-']     # line styles
ms = ['rs','g^','bo','k*']       # marker styles
#plotlabels = ['Adam','Gradient Descent','RMSProp','AdaGrad']
plotlabels = ['sigmoid','tanh','ReLU']
for i in [2]:
    costtrain,costtest = main(i,0.001)
    plt.loglog(np.arange((nIter-1)//100 + 1)*100,costtrain,ls[i],label=plotlabels[i])
    plt.loglog(np.arange((nIter-1)//100 + 1)*100,costtest,ms[i],mfc='none')
plt.legend(fancybox=True,shadow=True)
plt.xlabel("ItÃ©rations EntraÃ®nement")
plt.ylabel("Valeur de la fonction de coÃ»t")
plt.axis([100,nIter,0.1,10])
plt.savefig("cost_optimizer.png")
```

Références

[1] Paul Charbonneau. *Modélisation Numérique en Physique, Notes de cours*. Département de Physique, Université de Montréal, 2016.

[2] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul) :2121–2159, 2011.

[3] Jeff Heaton. *Introduction to neural networks with Java*. Heaton Research, Inc., 2008.

[4] Geoffrey Hinton, Nirsh Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a overview of mini–batch gradient descent. 2012.

[5] Bekir Karlik and A Vehbi Olgac. Performance analysis of various activation functions in generalized mlp architectures of neural networks. *International Journal of Artificial Intelligence and Expert Systems*, 1(4) :111–122, 2011.

[6] Andrej Karpathy. Neural net, 2016. [Online ; accessed April 26, 2017].

[7] Diederik Kingma and Jimmy Ba. Adam : A method for stochastic optimization. *arXiv preprint arXiv :1412.6980*, 2014.

[8] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553) :436–444, 2015.

[9] Georg Thimm and Emile Fiesler. Pruning of neural networks. Technical report, IDIAP, 1997.