Optimisation pour le Machine Learning: vers les réseaux de neurones

Rodolphe Le Riche* Didier Rullière†

28 novembre 2023

Préliminaires

Une question libre (cf. fin du TP) fera l'objet d'un compte-rendu par groupe de 4 (exceptionnellement 5), sous forme de Rmd + version pdf ou html. Ce compte-rendu servira à l'évaluation. Il est à remettre sur Campus (au lien de rendu TP) avant le 11 décembre 2023 à 23h59. A titre indicatif, le temps à consacrer à ce TP hors cours est d'environ une journée par personne.

Les langages autorisés sont R ou Python (ou autre), à condition de remettre un fichier notebook (.Rmd, .ipnb) et son export html ou pdf, comme détaillé ci-dessous.

ATTENTION, en dehors de la date de remise sur Campus, il y a deux consignes, ci-après.

Consigne 1. Vos noms

Eh oui, pas de noms, pas de note. Insérer ci-après le nom de chaque personne du groupe. Remplacer les noms ci-après, NOM en majuscule (=first family name), Prénom en minuscule.

```
#1. DURAND Victor
#2. DUPONT Annette
#3. DUCHEMIN Pierre
#4. DULAC Isis
#5. DUMMY John
```

L'ordre n'a pas d'importance, mais par convention la personne en position 1 sera celle qui remettra le TP sur Campus.

Merci d'utiliser les noms tels qu'ils apparaissent sur la liste des noms fournie à côté du sujet (directory du jour 4 sur Campus) `Liste Majeure SD 2023-2024.xlsx` (faire des copier-coller), de façon à permettre un appariement automatique de vos notes.

ATTENTION! seul les noms inscrits dans le documents seront utilisés, **aucun ajout a posteriori ne sera accepté**. Vérifiez bien la présence de tous les membres avant la remise.

Consigne 2. Fichiers à rendre

Pour la personne indiquée en position 1 dans la liste de noms, merci de remettre votre TPs sur l'espace dédié du portail emse (Campus). Il est attendu 2 fichiers par groupe :

*CNRS LIMOS, Mines St-Etienne, UCA
†Mines Saint-Etienne, CNRS LIMOS

- un fichier au format R markdown (extension .Rmd, vous pouvez compléter directement directement ce fichier) ou bien au format Jupyter Notebook (extension .ipynb, avec moteur R ou Python). Le fichier doit impérativement contenir vos noms!
- un export au format html ou pdf correspondant exactement au fichier précédent (bouton Knit sur RStudio, ou File>Download as>HTML sur Jupyter Notebook). Il ne doit donc pas contenir d'informations complémentaires par rapport au fichier précédent!

Merci de nommer vos fichiers en faisant apparaître le premier nom, par exemple ici: `TP_RNN_Optim_GroupeDURAND.Rmd` `TP_RNN_Optim_GroupeDURAND.html`

Avant de réaliser votre TP: Vérifiez bien que vous pouvez lancer "Knit to HTML" à l'aide du bouton Knit de RStudio (ou à l'aide de File>Download a>HTML sur Jupyter). Réitérez régulièrement cette opération lors de la réalisation du TP, pour être sûr que la version finale ne génère pas d'erreur! Si un code ne marche pas, vous pouvez le commenter de façon à ce que le html soit bien généré, votre code commenté restera visible.

ATTENTION! des pénalités sont prévues si il manque ce fichier HTML/pdf, ou si il manque le fichier Rmd source.

Partie I: Codes à essayer

Cette première partie est composée de 4 sous-parties:

1. La **propagation en avant**, partie dont le but est de montrer que coder un calcul entrée-sortie de réseau de neurones, c'est simple.
2. La **rétropropagation**
3. La création des données
4. L'optimisation du réseau

1. Propagation en avant

Dans cette partie, qui est donnée, on programme par étape progressives une propagation vers l'avant du signal dans un réseau de neurones.

1a. fonctions d'activation

Exemple de quatre fonctions d'activation et leurs dérivées:

```
linear <- function(x){x}
relu <- function(x){ifelse(x < 0 , 0, x )}
sigmoid <- function(x) { 1 / (1 + exp(-x)) }
#tanh déjà codé dans R base
linearDeriv <- function(x){1}
reluDeriv <- function(x){ifelse(x < 0 , 0, 1 )}
sigmoidDeriv <- function(x) { sigmoid(x)* (1- sigmoid(x))}
tanhDeriv <- function(x){1-tanh(x)^2}
```

1b. Description du réseau

weightsByLayer est une liste contenant les matrices de poids (et de biais). Pour passer de m à n neurones, sa taille est $(m+1) \times (n)$ du fait du biais pris en compte comme une entrée supplémentaire de valeur 1. Ainsi, `W[[k]][i,j]` passe du neurone `i` au neurone `j` au sein de la couche `k`.

Création de poids aléatoires pour initialiser un réseau de neurones:

```
layerSizes <- c(5,3,2) # 5 inputs, 3 intermediate neurons (=layer1), 2 outputs (=layer2)

createRandomWeightsByLayer <- function(layerSizes) {
  numberOfLayers <- length(layerSizes)-1
  weightsByLayer <- vector("list", numberOfLayers)
  for(i in 1:numberOfLayers) {
    nrows <- layerSizes[i]+1
    ncols <- layerSizes[i+1]
    # random initialization of weights according to LeCun and Bottou
    # E(weight)=0 , V(weight)=1/number_of_inputs
    weightsByLayer[[i]] <- matrix(data = rnorm(n=nrows*ncols,mean=0,sd=sqrt(1/nrows)),nrow = nrows)
  }
  return(weightsByLayer)
}
```

1c. Propagation en avant

Voici une première version de propagation, où toutes les couches on le même type de neurone: c'est restrictif, mais simple (on va bientôt relâcher cette contrainte).

```
forwardPropagation_1af <- function(inputs, weightsByLayer, activationFunction) {
  numberOfLayers <- length(weightsByLayer)
  layer <- inputs
  for(i in 1:numberOfLayers) { #propagation for layer n°i
    # to model the bias parameters, one append 1 to the current layer
    layer <- c(layer, 1)
    layer <- activationFunction(layer %*% weightsByLayer[[i]])
  }
  return(layer)
}
```

Un exemple de propagation vers l'avant dans un réseau:

```
set.seed(1234)
Weights <- createRandomWeightsByLayer(layerSizes)
randomInputs <- runif(layerSizes[1])
forwardPropagation_1af(inputs=randomInputs, Weights, activationFunction = sigmoid)
```

```
##           [,1]      [,2]
## [1,] 0.4839004 0.2955222
```

1d. Une fonction d'activation par neurone

il est possible de modifier le code de `forwardPropagation_1af` de façon à autoriser une fonction d'activation différente par neurone. `activationFunctionsByLayer` est alors une liste (par couche) de liste (par neurone de la couche) de fonctions d'activations. Ainsi, `activationFunctionsByLayer[[i]][[j]]()` correspond à la fonction d'activation du neurone `j` de la couche `i`.

```
forwardPropagation <- function(inputs, weightsByLayer, activationFunctionsByLayer) {
  numberOfLayers <- length(weightsByLayer)
  dimOut <- ncol(weightsByLayer[[numberOfLayers]])
  nbdata <- ncol(inputs)
  y <- matrix(nrow = dimOut, ncol = nbdata)
  for (idata in 1:nbdata) {
    layer <- inputs[,idata]
    for(i in 1:numberOfLayers) { #propagation for layer n°i
```

```
      #to model bias parameters, one append 1 to previousLayerOutput
      layer <- c(layer, 1)
      layer <- as.vector(layer %*% weightsByLayer[[i]])
      numberOfNeurons <- ncol(weightsByLayer[[i]])
      for(j in 1:numberOfNeurons) { layer[j] <- activationFunctionsByLayer[[i]][[j]](layer[j])
      }
    }
    y[,idata]<-layer
  }
  return(y)
}

# and an example of use
layerSizes <- c(5,3,3,2) # 5 inputs, 3 neurons, 3 neurons, 2 outputs
activationFunctionsByLayer <- list(c(sigmoid, relu, sigmoid), c(sigmoid, relu, relu), c(sigmoid, sigmoi

set.seed(1234)
Weights <- createRandomWeightsByLayer(layerSizes)
randomInputs <- matrix(data=runif(layerSizes[1]),ncol=1)
forwardPropagation(randomInputs, Weights, activationFunctionsByLayer)
```

```
##           [,1]
## [1,] 0.4926899
## [2,] 0.2693967
```

1e. Créer des réseaux de neurones et faire quelques exemples de propagations avant.

On pourrait par exemple prendre 2 entrées et une sortie pour pouvoir dessiner les fonctions créées. Interpréter les résultats. Cette partie n'est pas évaluée.

```
# # A correction
# ninput <- 2
# noutput <- 1
# layerSizes <- c(2,1,1)
# actFuncByLayer <- list(c(sigmoid), c(linear))
# set.seed(1)
# Weights <- createRandomWeightsByLayer(layerSizes)
# # createRandomWeightByLayer initializes weights of the sigmoid to rather small
# # values so that sigmoids are in their linear regime. To see the plateaus,
# # increase the weights.
# #
# # make inputs as a 2D grid
# no.grid <- 100
# LB <- c(-1,-1)
# UB <- c(1,1)
# x1 <- seq(LB[1], UB[1], length.out=no.grid)
# x2 <- seq(LB[2], UB[2], length.out=no.grid)
# x.grid <- expand.grid(x1, x2)
# dataInputs <- t(x.grid)
# dataOutputs<-forwardPropagation(dataInputs, Weights, actFuncByLayer)
# dataOutputs.grid <- matrix(dataOutputs, no.grid)
#
# ### 2D contour plot
# contour(x1, x2, dataOutputs.grid, nlevels=20, xlab="x1", ylab="x2")
# #### 3D plot
```

```
# mypersp <- persp(x = x1,y=x2,z=dataOutputs.grid,zlab = "NN")
# ## Below is the nicer interactive 3D RGL version
# library("rgl")
# open3d()
# surface3d(x1, x2, dataOutputs.grid, col= "lightblue")
# title3d("NN output", col="blue", font=4)
# decorate3d()
# aspect3d(1, 1, 1)
```

2. Rétro-propagation

2a. Fonction coût et utilitaires associés

On va maintenant programmer la fonction coût, ou "loss function", ou fonction d'apprentissage, i.e., la fonction que l'on minimise pour apprendre le réseau à partir des données. Nous en profitons pour introduire 2 fonctions qui seront utiles:

- `xtoWeightsByLayer` : permet de traduire un vecteur de variables (x en notation optimisation) en une liste de poids
- `weightsByLayerToX` : vice versa, traduit une liste de poids en un vecteur de variables

```
xtoWeightsByLayer <- function(x,layerSizes){
  numberOfLayers <- length(layerSizes)-1
  weightsByLayer <- vector("list", numberOfLayers)
  filledUpTo <- 0
  for(i in 1:numberOfLayers) {
      nrows <- layerSizes[i]+1
      ncols <- layerSizes[i+1]
      istart <- filledUpTo +1
      iend <- filledUpTo + nrows*ncols
      filledUpTo <- iend
      weightsByLayer[[i]] <- matrix(data = x[istart:iend],nrow = nrows)
  }
  return(weightsByLayer)
}

weightsByLayerToX <- function(weightsByLayer){
  x <- NULL
  for (i in 1:length(weightsByLayer)){
    x <- c(x,weightsByLayer[[i]])
  }
  return(x)
}
```

Définition de la fonction perte (ou "loss function").
Attention, pour l'utiliser, les variables

- `layerSizes`
- `actFuncByLayer`
- `dataInputs`, `dataOutputs`

doivent être définies car elles sont passées en variables globales...

```
# Note: layerSizes and the rest of the NN description + dataInputs and dataOutputs
# are passed as global variable, dangerous in general, ok for our small project
squareLoss <- function(x){
    weights <- xtoWeightsByLayer(x,layerSizes=layerSizes)
```

```
  pred <- forwardPropagation(inputs=dataInputs, weightsByLayer=weights, activationFunction=actFuncByLay
  return(0.5*(sum((as.numeric(pred)-as.numeric(dataOutputs))^2)))
}
```

Exemple de calcul de la fonction perte

```
# Example of loss calculation
layerSizes <- c(2,4,2)
actFuncByLayer <- list(c(sigmoid, sigmoid, relu, relu), c(linear, linear))
set.seed(5678)
Weights <- createRandomWeightsByLayer(layerSizes)
dataInputs <- matrix(runif(layerSizes[1]),ncol=1)
dataOutputs <- matrix(data = 1:layerSizes[length(layerSizes)],ncol = 1)
x <- weightsByLayerToX(weightsByLayer = Weights)
squareLoss(x)
```

```
## [1] 2.04404
```

2b. Rétro-propagation

Une routine de rétropropagation, `backPropagation`, est donnée ci-dessous:

```
# derivative of the square loss function, 1/2 ||pred-y||^2 , w.r.t. pred
# i.e.,  partial_Loss / partial_net_output
squareLossDeriv <- function(pred,y){
  return(t(pred-y))
}

# backpropagation function with only 1 data point.
# This routine is NOT optimized for performance (growing lists A and Z are inefficient, ...)
backPropagation <- function(dataIn, dataOut, weightsByLayer, actFunc, actFuncDeriv, lossDeriv=squareLoss
  # adding an identity last layer (+row of 0's to implement the bias) to get simpler recursions
  nOutLayer <- nrow(dataOut)
  weightsByLayer[[length(weightsByLayer)+1]]<-rbind(diag(x = rep(1,nOutLayer)),rep(0,nOutLayer))
  numberOfLayers <- length(weightsByLayer)
  A <- vector(mode = "list", numberOfLayers)
  Z <- vector(mode = "list",numberOfLayers)
  # first, forward propagation to calculate the network state, cf. forwardPropagation function
  for(i in 1:numberOfLayers) {
    if (i==1) {A[[i]]<-rbind(dataIn,1)}
    else {
      for (j in 1:length(actFunc[[i-1]])) { # for each neuron of the layer
        A[[i]][j] <- actFunc[[i-1]][[j]](Z[[i-1]][j])
      }
      A[[i]] <- rbind(as.matrix(A[[i]]), 1) # for the bias
    }
    Z[[i]] <- t(weightsByLayer[[i]]) %*% A[[i]]
  }
  # backward loop
  delta <- lossDeriv(pred=Z[[numberOfLayers]],y=dataOut)
  dloss_dweight <- vector(mode = "list",numberOfLayers-1) # the derivatives are first stored in the sam
  for (i in numberOfLayers:2) {
    # delta(l-1) = delta(l) * dz(l)_da(l) * da(l)_dz(l-1)
    # where delta(l) = dloss_dz(l)
    # da(l)_dz(l-1) is diagonal (neurons within a layer are not connected to each other)
```

```
    #   with an added row of 0's corresponding to the constant a(l)_last=1 (our implementation of the b
    Nbneurons <- length(Z[[i-1]])
      D <- rep(x = NA,Nbneurons)
    for (j in 1:Nbneurons){
      D[j]<-actFuncDeriv[[i-1]][[j]](Z[[i-1]][j])
    }
    delta <- delta %*% t(weightsByLayer[[i]]) %*% rbind(diag(D),rep(0,Nbneurons))
    # dloss_dw(l) = a(l)*delta(l)
    dloss_dweight[[i-1]]<-A[[i-1]] %*% delta
  }
  # derivatives now stored in a vector where each weight matrix is visited by column in the order of th
  dloss_dtheta <- NULL
  for (i in 1:(numberOfLayers-1)){
    dloss_dtheta <- c(dloss_dtheta,dloss_dweight[[i]])
  }
  return(list(pred=Z[[numberOfLayers]],dloss_dtheta=dloss_dtheta))
}
```

Execution de la rétropropagation

```
layerSizes <- c(2,3,4,2)
actFuncByLayer <- list(c(sigmoid, sigmoid, relu),
                       c(sigmoid,sigmoid,relu,relu),
                       c(linear, linear))
DerivActFunc <- list(c(sigmoidDeriv, sigmoidDeriv, reluDeriv),
                     c(sigmoidDeriv,sigmoidDeriv,reluDeriv,reluDeriv),
                     c(linearDeriv, linearDeriv))
set.seed(5678)
Weights <- createRandomWeightsByLayer(layerSizes)
dataInputs <- matrix(runif(layerSizes[1]),ncol=1)
dataOutputs <- matrix(data = 1:layerSizes[length(layerSizes)],ncol = 1)

BP <- backPropagation(dataIn=dataInputs, dataOut=dataOutputs, weightsByLayer=Weights, actFunc = actFuncl
# forwardPropagation(inputs=dataInputs, weightsByLayer=Weights, activationFunctionsByLayer=actFuncByLay
```

On compare le résultat de la rétropropagation à une différence finie calculée directement sur la fonction perte (loss).

```
# function to calculate f and gradient of f by forward finite difference
f.gradf <- function(x,f,h=1.e-8){
  d<-length(x)
  res <- list()
  res$fofx <- f(x)
  res$gradf <- rep(NA,d)
  for (i in 1:d){
    xp <- x
    xp[i] <- x[i]+h
    res$gradf[i] <- (f(xp)-res$fofx)/h
  }
  return(res)
}

x <- weightsByLayerToX(weightsByLayer = Weights)

# do a finite difference at x with squareLoss function
```

```
fdiff <- f.gradf(x=x,f=squareLoss)


# compare with the backpropagation value (should be close to 0)
BP$dloss_dtheta-fdiff$gradf
```

```
##  [1]  6.120490e-08  1.059396e-08  2.680773e-08  5.687843e-08  9.993687e-08
##  [6] -8.188601e-08 -2.566590e-08  5.361274e-08  9.733795e-08 -4.222980e-08
## [11]  5.439764e-08  5.997507e-08  1.266249e-08 -1.204942e-08  2.895221e-08
## [16]  6.990049e-10  7.704855e-09  3.338509e-08 -1.734897e-08 -2.676459e-08
## [21]  7.746944e-08 -4.050435e-08  1.903953e-08  9.144533e-08 -9.564124e-08
## [26]  1.994276e-08  7.121630e-08 -1.106066e-08  1.837879e-08  1.266530e-08
## [31]  2.039633e-09 -3.080408e-08  4.111767e-09 -1.511369e-08  1.672989e-08
```

3. Creation des données à apprendre

On génère des données pour la régression avec les fonctions tests de l'optimisation. AN: pour leur utilisation ultérieure dans un réseau de neurone, du fait de la présence de plateau dans les fonctions d'activation telles que tanh et sigmoid, il est important de normaliser les données (a minima faire une transformation linéaire pour les mettre entre -1 et 1). On donne donc les fonctions `normByRow` et `unnormByRow` pour normaliser et dénormaliser.

```
source('test_functions.R')

# normalization routines
normByRow <- function(X){
  nr <- dim(X)[1]
  nc <- dim(X)[2]
  Xnorm <- matrix(nrow = nr,ncol = nc)
  minAndMax <- matrix(nrow = nc,ncol=2)
  for (i in 1:nc){
    zmin<-min(X[,i])
    zmax<-max(X[,i])
    minAndMax[i,]<-c(zmin,zmax)
    Xnorm[,i]<-2*(X[,i]-zmin)/(zmax-zmin)-1
  }
  res<-list()
  res$dat <- Xnorm
  res$minAndMax <- minAndMax
  return(res)
}

unnormByRow <- function(normDat){
  nr <- dim(normDat$dat)[1]
  nc <- dim(normDat$dat)[2]
  X <- matrix(nrow = nr,ncol = nc)
  for (i in 1:nc){
    zmin<-normDat$minAndMax[i,1]
    zmax<-normDat$minAndMax[i,2]
    X[,i]<-(normDat$dat[,i]+1)/2*(zmax-zmin)+zmin
  }
  return(X)
}
```

```
fun<-quadratic
d<-2
LB <- rep(-5,d)
UB <- rep(5,d)
ntrain <- 15
ntest <- 100
ndata <- ntrain + ntest
set.seed(1) # with this seed you can reproduce the data
rawX <-t(replicate(n = ndata,expr = runif(n = d,min = LB,max = UB)))
set.seed(Sys.time()) # unset seed, back to "random"
rawYobs <- apply(X = rawX,MARGIN = 1,FUN = fun)
# normalize the data between -1 and 1
X <- normByRow(rawX)
# you can recover unnormalized data with, for expl : X <- unnormByRow(normIn)
Yobs <- normByRow(as.matrix(rawYobs))
itrain <- 1:ntrain
itest <- (ntrain+1):ndata
# data in stats are stored as 1 row for 1 point.
# In the neural network world, like often in linear algebra, vectors are columns thus a transpose is ne
Xtrain<-t(X$dat[itrain,])
Xtest<-t(X$dat[itest,])
Ytrain<-t(matrix(data=Yobs$dat[itrain,],ncol=1))
Ytest<-t(matrix(data=Yobs$dat[itest,],ncol=1))
```

4. Optimisation des poids et biais du réseau

4a. Optimisation par batch gradient et différences finies

Le gradient batch (cher mais moins bruité) est la somme des gradients pour toutes les données.
Les fonctions `forwardPropagation` et `squareLoss` fonctionnent sur des matrices de données passées à travers les variables globales `dataInputs` ($d \times N$) et `dataOutputs` ($n_K \times N$).

A faire en TP: apprendre un réseau de neurones en optimisant les poids et biais de façon à minimiser la loss totale sur toutes les données. Le code est donné. Il est demandé d'essayer d'autres réseaux : nombre de couches, de neurones, de types de neurones, tailles des ensembles d'apprentissage et de test, ntrain et ntest. Cette partie n'est pas évaluée.

```
# Example of loss calculation
layerSizes <- c(2,4,1)
actFuncByLayer <- list(c(sigmoid, sigmoid, relu, relu), c(linear))
set.seed(5678)
Weights <- createRandomWeightsByLayer(layerSizes)
dataInputs <- Xtrain
dataOutputs <- Ytrain
x <- weightsByLayerToX(weightsByLayer = Weights)
squareLoss(x)
```

```
## [1] 5.685068
```

Et maintenant branchons optimiseurs et fonction de perte de réseau de neurone ensemble, pour réaliser un apprentissage:

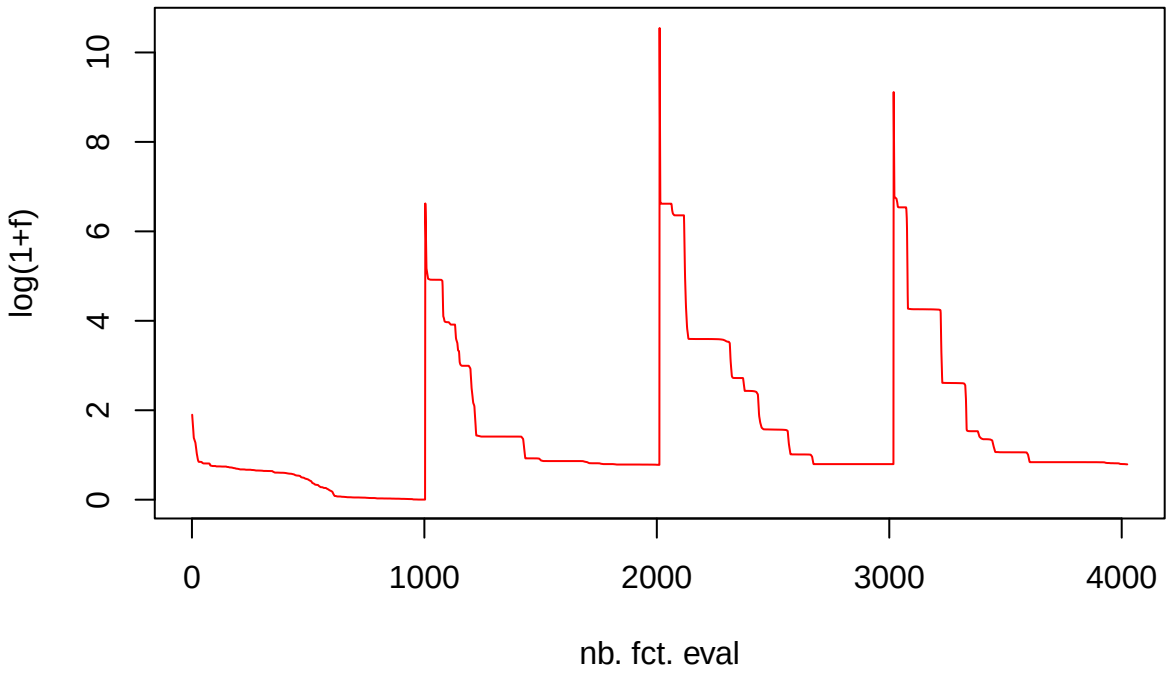
```
source('utilities_optim.R')
source('line_searches.R')
source('gradient_descent.R')
source('restarted_descent.R')
```

```r
pbFormulation <- list()
pbFormulation$fun<-squareLoss #function to minimize
d<-length(x)
pbFormulation$d<-d # dimension
pbFormulation$LB<-rep(-10,d) #lower bounds
pbFormulation$UB<-rep(10,d) #upper bounds

### algorithm settings
optAlgoParam <- list()
set.seed(112233) # repeatable run despite the use of pseudo-random number generators in runif
#
optAlgoParam$budget <- 4000
optAlgoParam$minGradNorm <- 1.e-6
optAlgoParam$minStepSize <- 1.e-11
optAlgoParam$nb_restarts <- 4
#
optAlgoParam$direction_type <- "momentum"
optAlgoParam$linesearch_type <- "armijo"
optAlgoParam$stepFactor <- 0.1 # step factor when there is no line search,
optAlgoParam$beta <- 0.9 # momentum term for direction_type == "momentum" or "NAG"
optAlgoParam$xinit <- weightsByLayerToX(weightsByLayer = Weights)
#
printlevel <- 2 # controls how much is stored and printed, choices: 0 to 4, cf. gradient_descent.R top

# a restarted descent
res <- restarted_descent(pbFormulation=pbFormulation,algoParam = optAlgoParam,printlevel=printlevel)
```

```
## **** START RESTARTED DESCENT
## **** search no. 1  done:
##      started at x =  -0.1698145 -0.140318 -0.1196843 -0.5456268 -0.6203712 -0.826807 -0.06530438 1.05
##      converged to x =  -2.110662 -2.939298 -3.116041 -1.730082 -2.790772 3.04062 -2.053296 -5.252755
##      where f =  0.002872187
## **** search no. 2  done:
##      started at x =  9.665925 0.8520535 -2.959505 -9.728239 1.324153 -6.593252 8.904208 -6.940292 1.5
##      converged to x =  6.661099 -9.994075 -8.559852 -8.164325 -0.3813499 -8.169417 9.021019 -3.215264
##      where f =  1.178867
## **** search no. 3  done:
##      started at x =  -6.044324 3.357041 -6.087453 7.816316 -9.104029 4.383013 4.093345 1.944467 7.157
##      converged to x =  -1.850047 2.271654 -9.990087 8.252023 -9.370944 2.612752 -0.6517697 2.001967 -
##      where f =  1.215648
## **** search no. 4  done:
##      started at x =  4.696986 0.4132267 -1.079167 -4.700774 -8.877291 2.965445 0.5434051 4.014376 -3
##      converged to x =  2.555132 -2.332464 10 -4.025237 -9.618255 2.656293 -0.5023199 1.638693 -5.8213
##      where f =  1.20351
## **** END RESTARTED DESCENT, overall best:
## **** best x: -2.110662 -2.939298 -3.116041 -1.730082 -2.790772 3.04062 -2.053296 -5.252755 -5.784695
## **** best f: 0.002872187
```



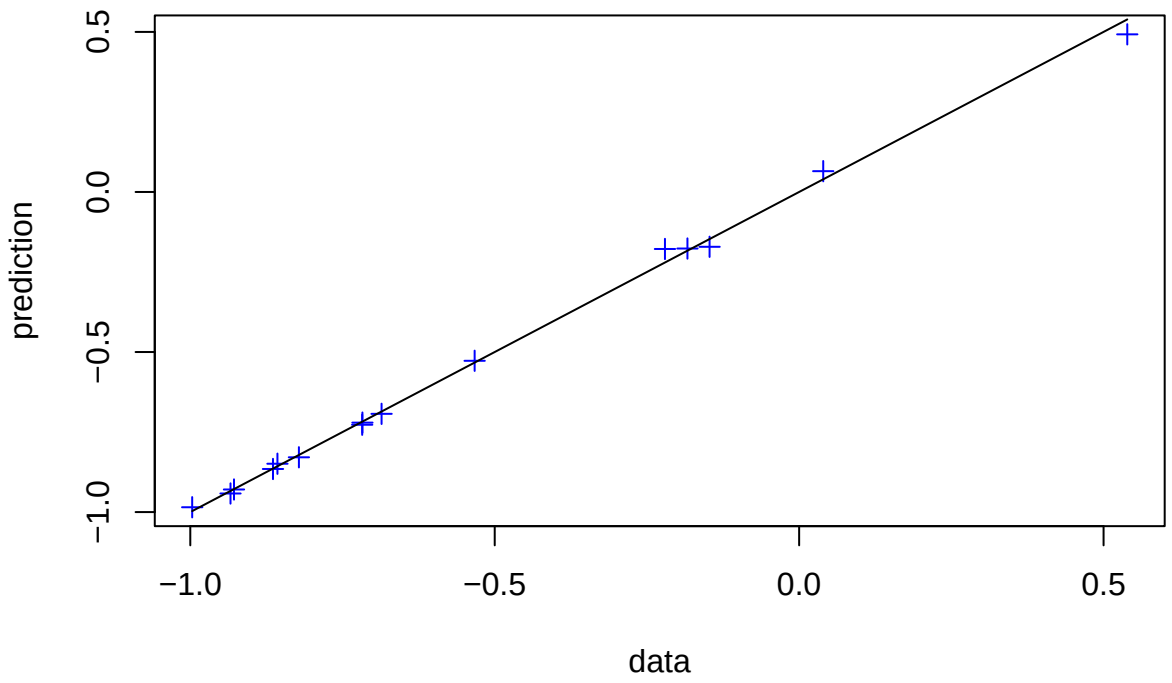

```
# extract the net learned
xBest <- res$xbest
lossBest <- res$fbest
wBest <- xtoWeightsByLayer(xBest,layerSizes)
```

Plot the results, check generalization with test data:

```
# Performance on the training set
predTrain <- forwardPropagation(Xtrain, wBest, actFuncByLayer) # predictions of the NN learned
rmseTrain <- sqrt((sum((Ytrain-predTrain)^2))/ntrain)
cat("Training RMSE =",rmseTrain,"\n")
```

```
## Training RMSE = 0.01956932
```

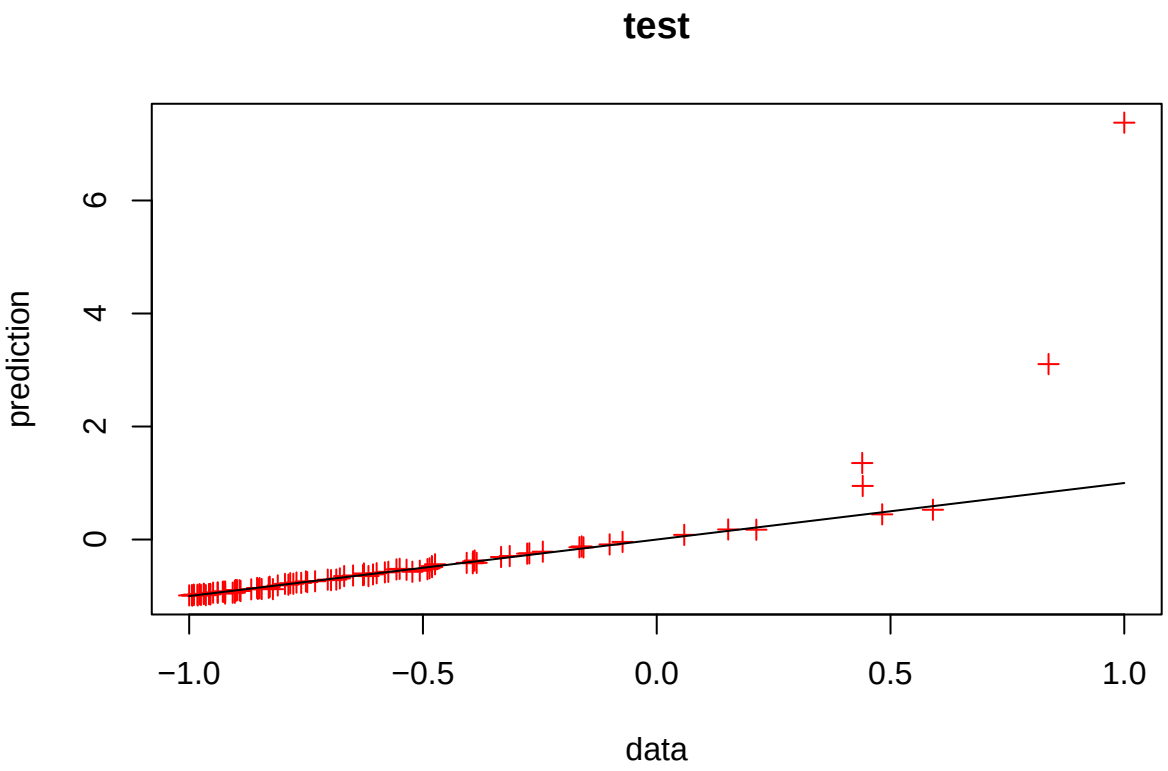
```
plot(Ytrain,predTrain,xlab="data",ylab="prediction",main="training",pch=3,col="blue")
ymin <- min(Ytrain)
ymax <- max(Ytrain)
lines(x = c(ymin,ymax), y=c(ymin,ymax))
```




```
# Performance on the testing set
predTest <- forwardPropagation(Xtest, wBest, actFuncByLayer) # predictions of the NN learned
rmseTest <- sqrt((sum((Ytest-predTest)^2))/ntest)
cat("Test RMSE =",rmseTest,"\n")
```

```
## Test RMSE = 0.6850739
```

```
plot(Ytest,predTest,xlab="data",ylab="prediction",main="test",pch=3,col="red")
ymin <- min(Ytest)
ymax <- max(Ytest)
lines(x = c(ymin,ymax), y=c(ymin,ymax))
```



Partie 2: extensions libres

C'est sur cette partie que vous serez évalués. Voici quelques idées, vous pouvez piocher une idée (voire plusieurs si vous le souhaitez), ou développer vos propres idées et tests. Il n'est pas nécessaire de présenter énormément de résultats, juste quelques essais qui montrent que vous avez compris des choses :-) Faites vos propres essais, ne recyclez pas les TPs des années précédentes.

- brancher la backpropagation sur l'apprentissage du réseau (l'exemple donné utilisait les différences finies)
- améliorer l'efficacité numérique ou la lisibilité de la fonction `backPropagation` donnée
- jouer sur les learning rates, la décroissance des learning rates
- coder un optimiseur déterministe avec les équations d'AdaGrad ou RMSProp ou Adam, qui permettent des "learning rates" individualisés par variable d'optimisation. On peut par exemple trouver ces équations en https://en.wikipedia.org/wiki/Stochastic_gradient_descent). Tester sur des fonctions analytiques.
- retrouver expérimentalement sur des fonctions quadratiques les taux de convergence théoriques
- jouer sur les mini-batches, i.e., étudier l'effet de la taille des batches
- utiliser plusieurs fonctions d'activation
- jouer sur des échantillons tests et apprentissage, notamment sur la taille des échantillons
- travailler sur la visualisation (cf. par exemple https://playground.tensorflow.org/)
- étudier les capacités de régression de réseaux sur plusieurs fonctions
- adapter à la classification
- comparer avec d'autres méthodes
- utiliser des régularisations
- travailler sur l'impact du bruit, bruiter les outputs, ...
- essayer une structure d'autoencodeur (contraction d'une des couches)
- Etudier la flexibilité des réseaux de neurones
  - on pourra reprendre les dessins faits en Question 1.e. Répéter l'opération pour 1, 2, et 4 neurones cachés. Eventuellement ajouter une couche. Commenter.

  - Proposer une formule pour calculer la souplesse de la fonction résultant d'un réseau de neurone (= une mesure de "capacité" ou "complexité"): cette souplesse est à estimer à structure de réseau donnée (nombre de couches, nombre et types de neurones), mais pour des poids et biais variables. De nombreuses réponses sont possibles à cette question, le plus important étant d'expliquer.
- optimisation des poids par gradient stochastique: Lorsque le gradient est calculé avec une seule donnée prise au hasard (un point de `Xtrain` et la sortie `Ytrain` associée), la fonction de perte devient aléatoire. Dans ce cas, on dispose bien d'une version bruitée du gradient de la loss par rétropropagation ou différences finies. Essayer d'optimiser un réseau par descente de gradient stochastique.
- l'idée qui vous rendra célèbre...