Optimisation pour le Machine Learning: vers les réseaux de neurones

Rodolphe Le Riche & Didier Rullière

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-aprè

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. BATTISTINI Lisa
#2. GIBOUDEAU Coralie
#3. LEROY Antoine
#4. PAOLO Jules
```

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 :

• 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 RStu- dio, 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) × (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, sigmoid))

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.

```
 # La correction
ninput <- 2 #entres
noutput <- 1 #sortie
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)
```

```
Error in forwardPropagation(dataInputs, Weights, actFuncByLayer):
unused argument (actFuncByLayer)
Traceback:
```

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=actFuncByLayer)
  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=squareLossDeriv) {
  # 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 same format as the weights
  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 biases)
    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 the layers
  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 = actFuncByLayer, actFuncDeriv =
DerivActFunc)
# forwardPropagation(inputs=dataInputs, weightsByLayer=Weights,
activationFunctionsByLayer=actFuncByLayer)
```

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 needed
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
```

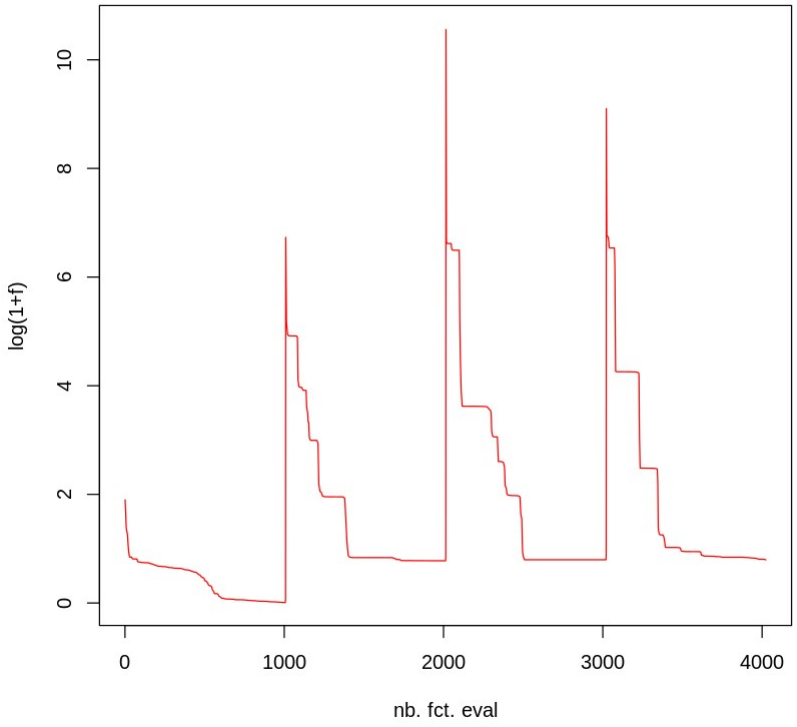
Et maintenant branchons optimiseurs et fonction de perte de réseau de neurone ensemble, pour réaliser un apprentissage:

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

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 comments for more info.

# 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 1.058296 0.171672 0.9623136 -0.5780707
-0.7794103 -0.3249456 0.1753755 -0.9224048 -0.2307617 0.3681053
     converged to x =  -1.819443 -2.689279 -2.846836 -2.003922 -
2.61962 2.98552 -2.049752 -5.241615 -5.774034 2.016533 -0.5841299
0.07811178 2.407452 -2.677676 6.966926 -0.01648342 1.467398
     where f =  0.009391513
**** search no. 2  done:
     started at x =  9.665925 0.8520535 -2.959505 -9.728239 1.324153 -
6.593252 8.904208 -6.940292 1.555679 -2.340857 1.807012 -2.221563 -
7.495461 6.032264 -0.5674507 -5.472412 -4.257868
     converged to x =  6.68499 -9.991948 -8.52543 -8.082223 -0.3752884
-8.269472 9.009985 -3.19708 0.6002852 -0.8790006 -0.1120017 -4.598093
-0.1745857 0.5100216 -0.02406059 -5.208231 -0.436581
     where f =  1.176115
**** search no. 3  done:
     started at x =  -6.044324 3.357041 -6.087453 7.816316 -9.104029
4.383013 4.093345 1.944467 7.157788 3.018581 5.272824 -9.688565 -
7.150293 6.703349 7.622149 -5.227606 8.213823
     converged to x =  -1.372597 2.157235 -9.986543 8.297965 -9.467851
2.388323 -1.260659 2.009346 -10 3.018581 5.272824 -9.688565 -2.761942
-0.3488965 -10 -5.227606 -0.3125134
```

```
     where f =  1.216772
**** search no. 4  done:
     started at x =  4.696986 0.4132267 -1.079167 -4.700774 -8.877291
2.965445 0.5434051 4.014376 -3.441846 -7.288633 8.574096 0.4261565 -
5.997695 4.63702 -6.966185 -8.703822 9.308671
     converged to x =  2.529302 -2.388776 10 -4.230385 -9.558808
2.302771 -0.5527243 1.524186 -5.935997 -3.215991 -2.68987 -10 -
1.361942 -0.3355084 -6.677977 7.184573 1.023231
     where f =  1.214808
**** END RESTARTED DESCENT, overall best:
**** best x: -1.819443 -2.689279 -2.846836 -2.003922 -2.61962 2.98552
-2.049752 -5.241615 -5.774034 2.016533 -0.5841299 0.07811178 2.407452
-2.677676 6.966926 -0.01648342 1.467398
**** best f: 0.009391513
```

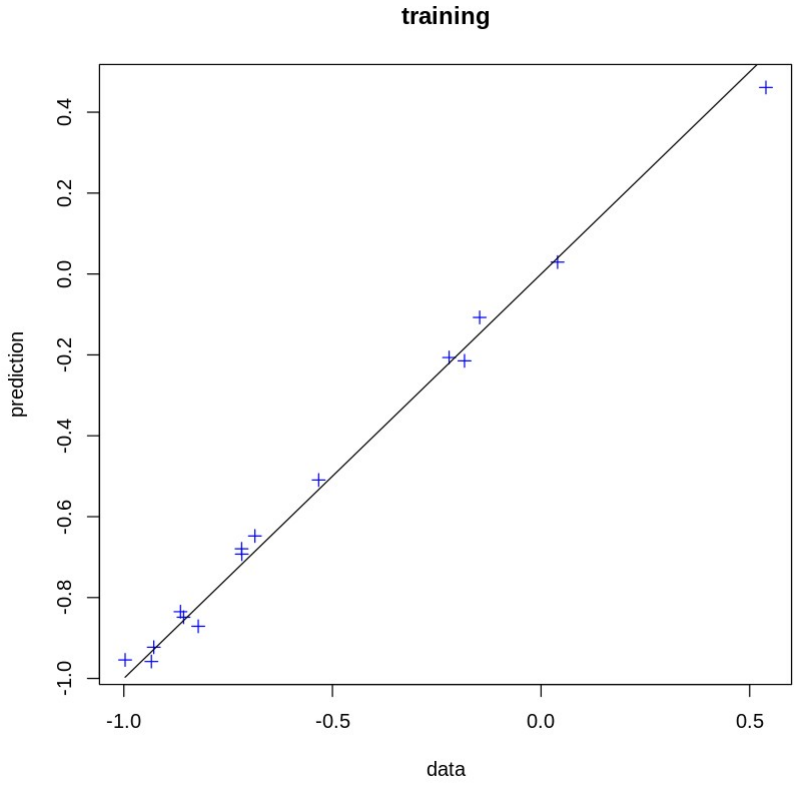



```
# extract the net learned
xBest <- res$xbest
lossBest <- res$fbest
wBest <- xtoWeightsByLayer(xBest,layerSizes)
```

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

```
Training RMSE = 0.03538646
```

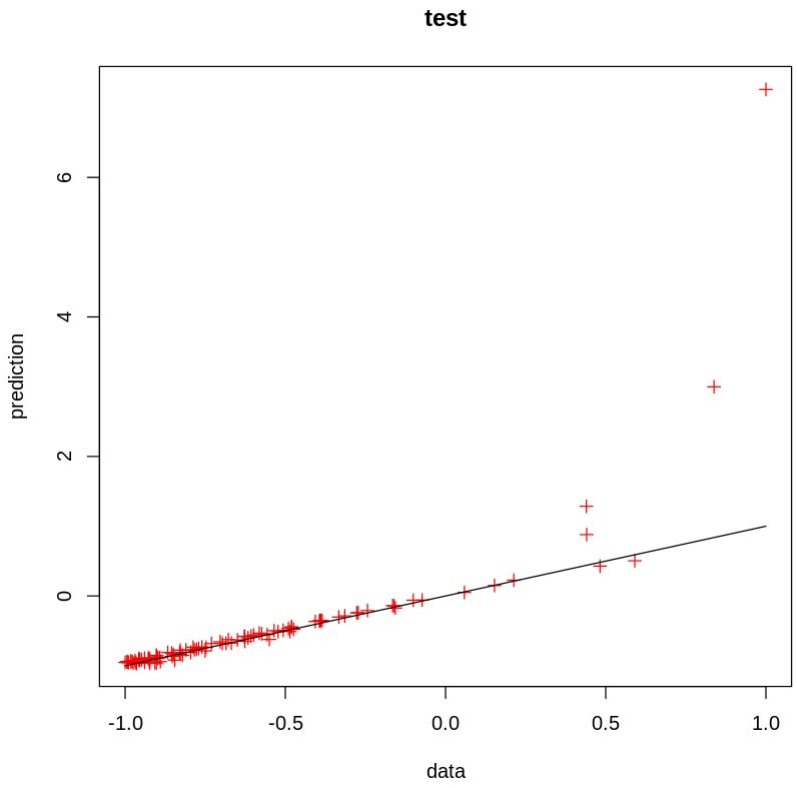



```
# Performance on the testing set
predTest <- forwardPropagation(Xtest, wBest, actFuncByLayer) #
predictions of the NN learned
rmseTest <- sqrt((sum((Ytest-predTest)^2))/ntest)
cat("Test RMSE =",rmseTest,"\n")
```

```
Test RMSE = 0.6700658
```

```
plot(Ytest,predTest,xlab="data",ylab="prediction",main="test",pch=3,co
l="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...

2.a Brancher la backpropagation sur l'apprentissage du réseau

2.a.1. Cas d'étude simple : backpropagation

On se propose dans cette partie de brancher la fonction de backpropagation. L'idée "naïve" ici est simple : puisque la fonction backpropagation nous retourne le gradient de f, on l'exécute un nombre de fois n = epochs, en ajustant les poids avec un learning rate alpha statique.

On décrit tout d'abord notre réseau de neurones simple.

```
# On travaille ici sur un réseau 2, 3, 4, 2

set.seed(1234)

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))

dataInputs <- matrix(runif(layerSizes[1]), ncol=1)
dataOutputs <- matrix(data = 1:layerSizes[length(layerSizes)], ncol =
1)

Weights <- createRandomWeightsByLayer(layerSizes)
print(dataInputs)
print(dataOutputs)
```

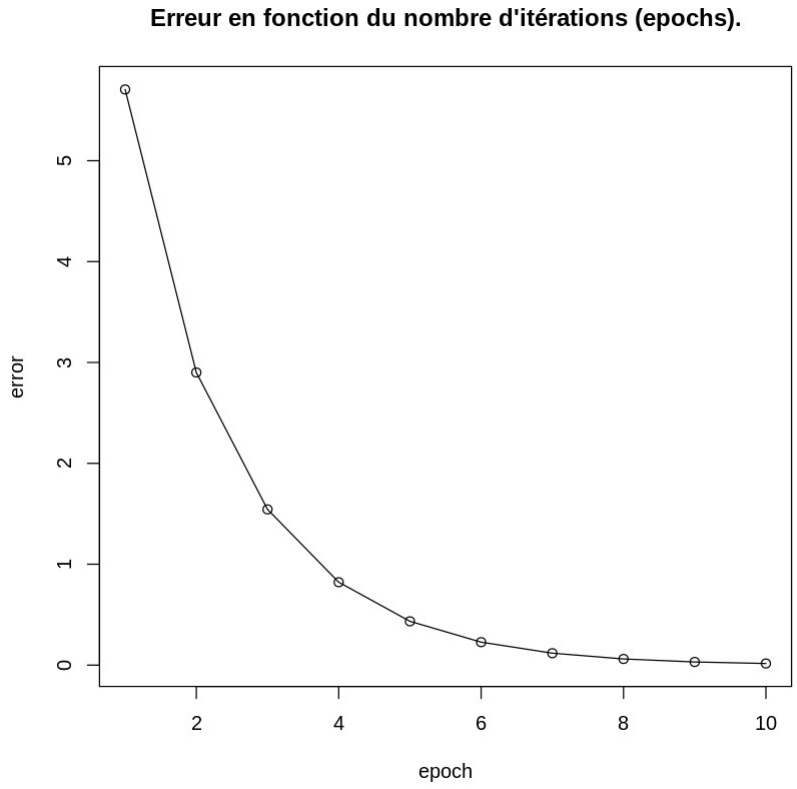
```
          [,1]
[1,] 0.1137034
[2,] 0.6222994
     [,1]
[1,]    1
[2,]    2
```

On implémente ensuite notre algorithme "naïf" :

```
# Dans notre fonction BP_implement, le layerSize est passé en variable
globale

epochs = 10 # nombre d'epochs
alpha = 0.2 # learning rate

BP_implement <- function(epochs, learning_rate, weights, dataIn,
dataOut) {
```

```
  # erreurs
  error <- rep(1, epochs)

  for (i in 1:epochs) {
    # On exécue la backpropagation (la forward propagation est
directement inclue dans la fonction)
    BP <- backPropagation(dataIn=dataIn, dataOut=dataOut,
weightsByLayer=weights, actFunc = actFuncByLayer, actFuncDeriv =
DerivActFunc)

    # On converti nos poids en vecteur et les met à jour
    x <- weightsByLayerToX(weights)
    x <- x - learning_rate * BP$dloss_dtheta

    # On converti notre liste de poids en matrices
    weights <- xtoWeightsByLayer(x, layerSizes)

    # On enregistre l'erreur
    error[i] <- 0.5 * sum(BP$pred-dataOut)^2
  }

  # On met à jour nos poids une dernière fois : ce sont les poids du
modèle entraîné
  x <- weightsByLayerToX(weights)
  x <- x - learning_rate * BP$dloss_dtheta

  return(list(error= error, x= x, pred= BP$pred))
}

print(BP$pred)
print(dataOutputs)

BP <- BP_implement(epochs = epochs, learning_rate = alpha, weights =
Weights, dataIn = dataInputs, dataOut = dataOutputs)
error <- BP$error
plot(error, xlab="epoch", main="Erreur en fonction du nombre
d'itérations (epochs).")
lines(error)
```

```
          [,1]
[1,]  1.688656
[2,] -1.043648
     [,1]
[1,]    1
[2,]    2
```



On remarque bien que, comme escompté, l'erreur diminue en fonction du nombre d'itérations de notre backpropagation. Cela-dit nous sommes ici dans un cas très basique, avec des données d'entrée constituées d'un seul individu. L'idée est dorénavant de pouvoir travailler sur un vrai ensemble de données d'entrée.

Remarquons également que la convergence de notre réseau de neurones dépend aussi grandement du learning rate choisi, mais nous ne détaillerons pas cet aspect dans cette partie.

2.a.2. Adaptation à un ensemble de données d'entrée : BP systématique.

On peut réutiliser notre algorithme de backpropagation afin de traiter un ensemble de données. On décide dans un premier temps d'effectuer une backpropagation pour chaque individu, puisque la fonction codée ne peut traiter qu'un individu. On modifiera dans un second temps cette fonction afin de pouvoir travailler sur un batch d'individus et effectuer une rétro-propagation sur une erreur moyenne, ce qui permettra d'optimiser le temps de calcul.

Dans le cadre d'une régression, la fonction que l'on souhaite apprendre est la suivante :

f(x,y) = (1, 2)

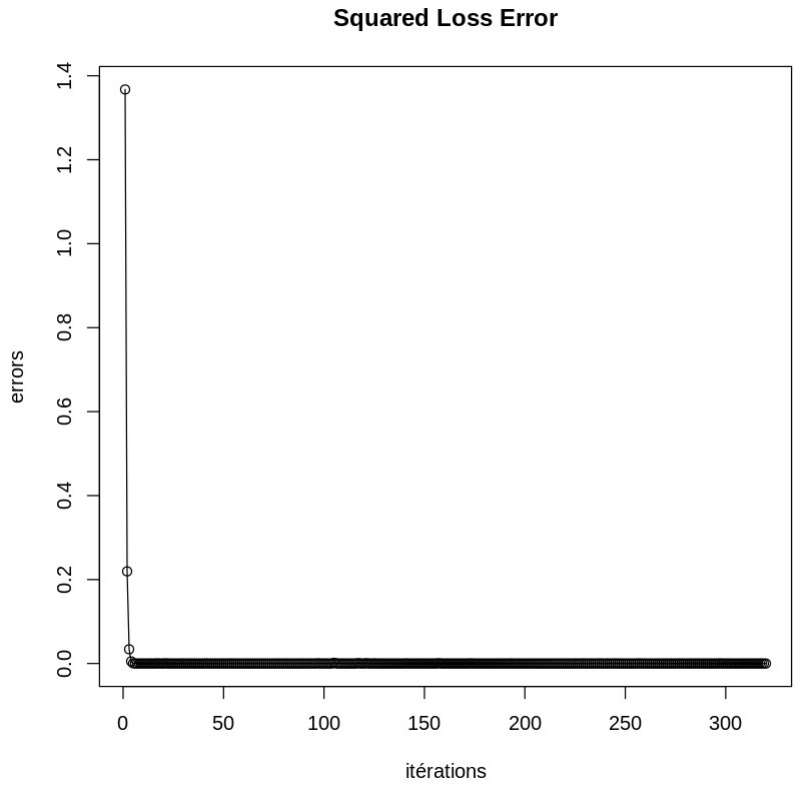
```
# Génération des données d'apprentissage

n_individus <- 100
```

```r
dataInputs <- matrix(c(0,0), 2, n_individus)
dataOutputs <- matrix(c(0,0), 2, n_individus)

for (i in 1:ncol(dataInputs)) {

  dataInputs[,i] <- t(runif(2))
  dataOutputs[1,i] <- 1
  dataOutputs[2,i] <- 2

}


dataInTrain <- dataInputs[,1:(0.8*n_individus)]
dataInTest <- dataInputs[,(0.8*n_individus + 1):n_individus]

dataOutTrain <- dataOutputs[,1:(0.8*n_individus)]
dataOutTest <- dataOutputs[,(0.8*n_individus + 1):n_individus]

print(as.matrix(dataInTrain[,1]))
print(as.matrix(dataOutTrain[,1]))
```

```
            [,1]
[1,] 0.01462726
[2,] 0.78312110
     [,1]
[1,]    1
[2,]    2
```

```r
# Entrainement du modèle sur l'ensemble des individus uns à uns.

epochs = 4
alpha = 0.3
Weights <- createRandomWeightsByLayer(layerSizes)

errors <- c()

for (i in 1:ncol(dataInTrain)) {

  # On exécute la BP pour un seul individu à la fois
  BP <- BP_implement(epochs = epochs, learning_rate = alpha, weights =
Weights, dataIn = as.matrix(dataInTrain[,i]), dataOut =
as.matrix(dataOutTrain[,i]))

  # On met à jour les poids
  Weights <- xtoWeightsByLayer(BP$x, layerSizes)

  # On enregistre l'erreur
  errors <- c(errors, BP$error)

}
```

```
plot(errors, xlab="itérations", main="Squared Loss Error")
lines(errors)

# On récupère le modèle entraîné
x_trained <- BP$x
```




```
# Test du modèle.

trained_weights <- xtoWeightsByLayer(x_trained, layerSizes)

for (i in 1:ncol(dataOutTest)) {

  pred <- forwardPropagation(as.matrix(dataInTest[,i]),
trained_weights, actFuncByLayer)

  # On affiche les 5 premières prédictions
  if (i <= 5) {
    print(pred)
  }

}
```

```
         [,1]
[1,] 1.003615
[2,] 2.005477
         [,1]
[1,] 1.006924
[2,] 2.017361
         [,1]
[1,] 1.004137
[2,] 2.010725
         [,1]
[1,] 1.001339
[2,] 1.998002
         [,1]
[1,] 1.006783
[2,] 2.017151
```

On remarque lors de la phase d'apprentissage que l'erreur semble converger très rapidement vers 0. Le problème reste relativement très simplet. En effectuant une backpropagation pour chaque individu, le modèle essaie de s'adapter itérativement à chacun des individus ce qui rend sa tâche de convergence coûteuse. Effectivement on réalise n_individus * epochs fois notre backpropagation. Toutefois les résultats peuvent fortement varier d'un entraînement à un autre.

Le principal problème de cette modélisation réside dans le temps et le coût de calcul, qui peuvent devenir très important dans un travail avec beaucoup de données d'entrée. Effectivement réaliser une backpropagation sur chaque individu est très coûteux.

Avant de proposer une solution avec des batch, essayons de calculer une fonction de perte moyenne sur l'ensemble des individus, que l'on essaiera de minimiser. On travaille ainsi avec le RMSE : root mean squared error.

2.a.3. Adaptation à un ensemble de données d'entrée : RMSE.

Modifions la fonction backPropagation afin d'effectuer une propagation en avant pour chacun des individus et ensuite calculer une erreur moyenne.

```
backPropagationRMSE <- function(dataIn, dataOut, weightsByLayer,
actFunc, actFuncDeriv, lossDeriv=squareLossDeriv) {

  # adding an identity last layer (+row of 0's to implement the bias)
to get simpler recursions
  n_ind <- ncol(dataIn)
  nOutLayer <- nrow(dataOut)

  weightsByLayer[[length(weightsByLayer)+1]] <- rbind(diag(x =
rep(1,nOutLayer)), rep(0,nOutLayer))
  numberOfLayers <- length(weightsByLayer)
  SumPred <- c()
```

```
  # FORWARD PROPAGATION
  for (k in 1:n_ind) {
    A <- vector(mode = "list", numberOfLayers)
    Z <- vector(mode = "list", numberOfLayers)

    for(i in 1:numberOfLayers) {
      if (i==1) {
        A[[i]] <- rbind(as.matrix(dataIn[,k]),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]]) %*% as.matrix(A[[i]])
    }
    # On sommera les valeurs de sortie pour calculer une erreur
moyenne par la suite
    SumPred <- cbind(SumPred, Z[[numberOfLayers]])
  }


  # BACKPROPAGATION
  # on calcule la dérivée partielle de l'erreur moyenne ainsi que
l'erreur moyenne
  delta <- lossDeriv(pred = as.matrix((1/n_ind)*rowSums(SumPred)), y =
as.matrix((1/n_ind)*rowSums(dataOut)))
  error = (1/n_ind)*0.5*sum((as.matrix(rowSums(SumPred)) -
as.matrix(rowSums(dataOut)))^2)
  dloss_dweight <- vector(mode = "list", numberOfLayers-1)

  for (i in numberOfLayers:2) {
    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_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 the layers
  dloss_dtheta <- NULL
  for (i in 1:(numberOfLayers-1)){
    dloss_dtheta <- c(dloss_dtheta,dloss_dweight[[i]])
  }
```

```
  return(list(SumPred = SumPred, dloss_dtheta = dloss_dtheta, error =
error))
}

BP_implementRMSE <- function(epochs, learning_rate, weights, dataIn,
dataOut) {

  # erreurs
  error <- rep(1, epochs)

  for (i in 1:epochs) {
    # On exécue la backpropagation RMSE (la forward propagation est
directement inclue dans la fonction)
    BP <- backPropagationRMSE(dataIn=dataIn, dataOut=dataOut,
weightsByLayer=weights, actFunc = actFuncByLayer, actFuncDeriv =
DerivActFunc)

    # On converti nos poids en vecteur et les met à jour
    x <- weightsByLayerToX(weights)
    x <- x - learning_rate * BP$dloss_dtheta

    # On converti notre liste de poids en matrices
    weights <- xtoWeightsByLayer(x, layerSizes)

    # On enregistre l'erreur
    error[i] <- BP$error
  }

  # On met à jour nos poids une dernière fois : ce sont les poids du
modèle entraîné
  x <- weightsByLayerToX(weights)
  x <- x - learning_rate * BP$dloss_dtheta

  return(list(error= error, x= x, pred= BP$pred))
}
```

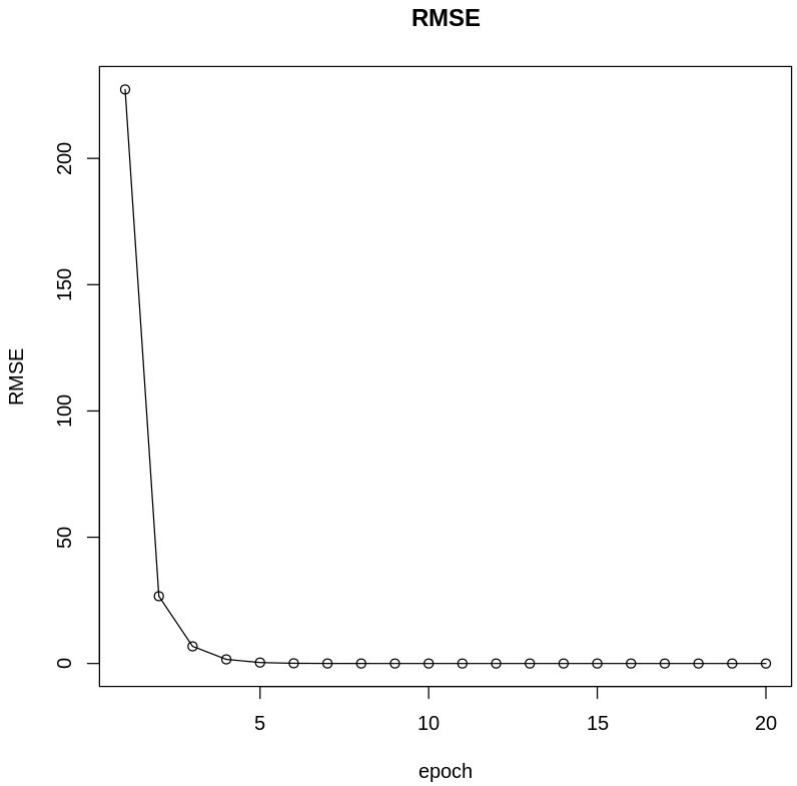
Notre fonction BP_implementRMSE suit donc implicitement l'algorithme suivant :

1. on calcule une forward propagation pour chacun des individus de notre base de données d'entrée,
2. on calcule une erreur moyenne sur ces n individus,
3. on réalise une backpropagation et on update les poids,
4. on répète l'opération un nombre n = epochs de fois.

Testons à présent la convergence de notre modèle ainsi que ses capacités prédictives.

```
# Entraînement du modèle
Weights <- createRandomWeightsByLayer(layerSizes)

BP <- BP_implementRMSE(20, 0.3, Weights, dataInTrain, dataOutTrain)
```

```
RMSE <- BP$error
plot(RMSE, xlab="epoch", main="RMSE")
lines(RMSE)
```




```
# Test du modèle
x <- BP$x
trained_weights <- xtoWeightsByLayer(x, layerSizes)

for (i in 1:ncol(dataOutTest)) {

  pred <- forwardPropagation(as.matrix(dataInTest[,i]),
trained_weights, actFuncByLayer)

  if (i <= 5) {
    print(as.matrix(pred))
  }

}
```

```
          [,1]
[1,] 0.9868361
[2,] 1.9471266
         [,1]
[1,] 1.001034
```

```
[2,] 2.016967
         [,1]
[1,] 1.021278
[2,] 2.069854
          [,1]
[1,] 0.9893514
[2,] 1.9603893
          [,1]
[1,] 0.9846767
[2,] 1.9703664
```

La méthode visant à minimiser la fonction coût RMSE permet d'obtenir les mêmes résultats que la méthode brute de décoffrage en seulement 10 epochs, ce qui réduit grandement le coût de calcul nécessaire.

2.a.4. Méthode entre-deux : utiliser un batch.

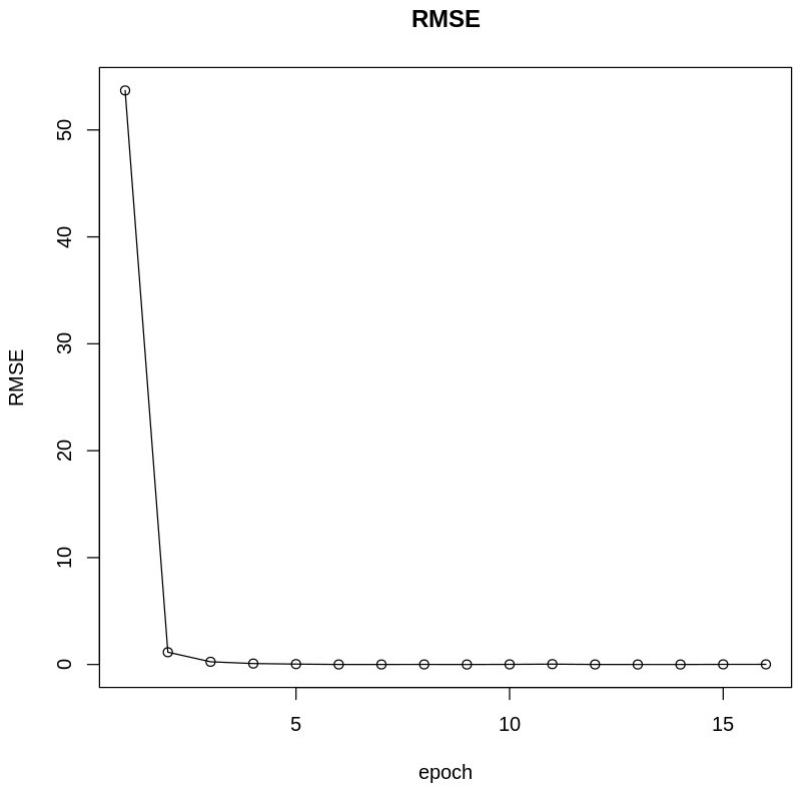
La méthode du batch nous permet de selectionner une portion du nombre d'individus afin de calculer une erreur moyenne sur ces batch_size individus.

On modifie alors légèrement la fonction d'implémentation de la backpropagation pour mettre en place cette méthode.

```
BP_implementRMSE_batch <- function(epochs, learning_rate, weights,
dataIn, dataOut, batch_size) {


  # détermine le nombre de batch dans notre ensemble de départ.
  n_ind <- ncol(dataIn)
  n_batch <- n_ind / batch_size
  indices <- seq(1, n_ind, by = batch_size)

  if (n_ind %% batch_size != 0) {
    print("LE BATCH N'EST PAS UN MUTLIPLE DES ENTREES !!")
    stop()
  }

  # erreurs
  error <- rep(1, n_batch * epochs)
  e <- 1

  # Pour chaque epoch
  for (i in 1:epochs) {

    # On exécue la backpropagation_RMSE pour chaque batch
    for (j in 1:n_batch) {

      # Permet de sélectionner correctement les bonnes plages de
données
      k <- indices[j]
```

```
      BP <- backPropagationRMSE(dataIn=dataIn[,k:(k+batch_size-1)],
dataOut=dataOut[,k:(k+batch_size-1)], weightsByLayer=weights, actFunc
= actFuncByLayer, actFuncDeriv = DerivActFunc)

      # On converti nos poids en vecteur et on les met à jour
      x <- weightsByLayerToX(weights)
      x <- x - learning_rate * BP$dloss_dtheta

      # On converti notre liste de poids en matrices
      weights <- xtoWeightsByLayer(x, layerSizes)

      # On enregistre l'erreur
      error[e] <- BP$error
      e <- e + 1
    }

  }

  # On met à jour nos poids une dernière fois : ce sont les poids du
modèle entraîné
  x <- weightsByLayerToX(weights)
  x <- x - learning_rate * BP$dloss_dtheta

  return(list(error= error, x= x, pred= BP$pred))
}

# Entraînement du modèle
Weights <- createRandomWeightsByLayer(layerSizes)

BP <- BP_implementRMSE_batch(2, 0.2, Weights, dataInTrain,
dataOutTrain, 10)
RMSE <- BP$error
plot(RMSE, xlab="epoch", main="RMSE")
lines(RMSE)

# Test du modèle
x <- BP$x
trained_weights <- xtoWeightsByLayer(x, layerSizes)

for (i in 1:ncol(dataOutTest)) {

  pred <- forwardPropagation(as.matrix(dataInTest[,i]),
trained_weights, actFuncByLayer)

  if (i <= 5) {
    print(as.matrix(pred))
  }

}
```

```
          [,1]
[1,] 1.057380
[2,] 2.082817
          [,1]
[1,] 1.035189
[2,] 2.053074
          [,1]
[1,] 1.037320
[2,] 2.063434
          [,1]
[1,] 1.095143
[2,] 2.130895
          [,1]
[1,] 1.048653
[2,] 2.063725
```



De la même manière que précédemment, le modèle utilisant un batch permet d'arriver au même résultat de converge. Il utilise cela dit légèrement plus de puissance puisque qu'il exécute la backpropagation un nombre batch_size * epochs de fois.

Remarquons également que passer un nombre batch_size = n_ind en paramètre revient à faire une backpropagation RMSE sur l'ensemble de nos données.

Jouer sur les learning rates et leur décroissance.

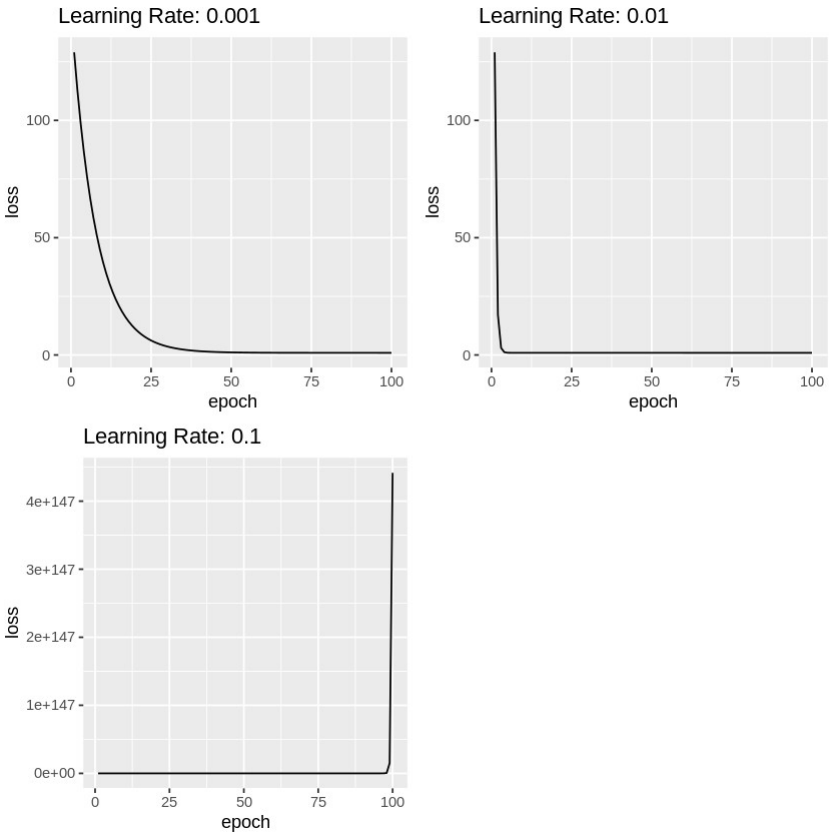
```r
if (!require(ggplot2)) install.packages("ggplot2")
library(ggplot2)
if (!require(gridExtra)) install.packages("gridExtra")
library(gridExtra)

# Générer des données
set.seed(123)
x <- runif(100, min = -10, max = 10)
y <- 2 * x + rnorm(100)

# Fonction pour calculer la perte quadratique moyenne
mse <- function(y, y_pred) {
  mean((y - y_pred) ^ 2)
}

# Fonction pour la descente de gradient
gradient_descent <- function(x, y, lr = 0.01, epochs = 100) {
  m <- 0 # Pente initiale
  b <- 0 # Intercept initial
  n <- length(y) # Nombre de données
  history <- data.frame(epoch = integer(0), loss = numeric(0))

  for (i in 1:epochs) {
    y_pred <- m * x + b
    loss <- mse(y, y_pred)
    history <- rbind(history, data.frame(epoch = i, loss = loss))

    # Calculer les gradients
    grad_m <- (-2/n) * sum(x * (y - y_pred))
    grad_b <- (-2/n) * sum(y - y_pred)

    # Mise à jour des paramètres
    m <- m - lr * grad_m
    b <- b - lr * grad_b
  }

  list(coefficients = c(m, b), history = history)
}

# Tester différents learning rates
lr_values <- c(0.001, 0.01, 0.1)
results <- list()

for (lr in lr_values) {
  results[[as.character(lr)]] <- gradient_descent(x, y, lr = lr,
epochs = 100)
```

```
}

# Visualisation
plot_list <- list()
for (i in 1:length(lr_values)) {
  lr <- lr_values[i]
  plot_list[[i]] <- ggplot(results[[as.character(lr)]]$history, aes(x
= epoch, y = loss)) +
    geom_line() +
    ggtitle(paste("Learning Rate:", lr))
}

do.call(grid.arrange, c(plot_list, ncol = 2))
```

```
Loading required package: ggplot2
```

```
Loading required package: gridExtra
```

```
Warning message in library(package, lib.loc = lib.loc, character.only
= TRUE, logical.return = TRUE, :
“there is no package called ‘gridExtra’”
Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)
```



- Learning Rate 0.001 : Ce taux d'apprentissage est probablement trop faible, car la perte décroît très lentement. Cela signifie que le modèle nécessite plus d'époques pour converger vers une solution optimale.
- Learning Rate 0.01 : Ce taux semble plus approprié. La perte diminue rapidement et semble se stabiliser, ce qui indique que le modèle est en train de converger.
- Learning Rate 0.1 : Ce taux d'apprentissage est probablement trop élevé. La perte a augmenté à un nombre extrêmement grand, ce qui suggère que le modèle diverge. C'est typique d'un learning rate qui est si élevé que les mises à jour de poids "sautent" au-delà du minimum de la fonction de perte, ce qui conduit à une perte de plus en plus grande.

L'expérience montrent l'importance de choisir un learning rate approprié. Un taux trop faible entraînera une convergence très lente, nécessitant un grand nombre d'époques pour atteindre l'optimum. Un taux trop élevé peut causer la divergence et empêcher le modèle de trouver une solution adéquate.

Influence des fonctions d'activation

On s'interesse ici à un réseau de neurones (2,4,1) défini précédemment, on commence par tester le réseau avec des fonctions d'activation identique en tout neurone pour comparer leur efficacité dans notre exemple.

Pour déterminer dans un premier temps quelle fonction d'activation est meilleure quand aplliquée à tous les neurones, on effectue les propagations vers l'avant pour les fonctions sigmoid, linear, relu et tanh. Le code si dessous stocke les valeurs des RMSE sur les données trains et les données tests pour chaque fonction.

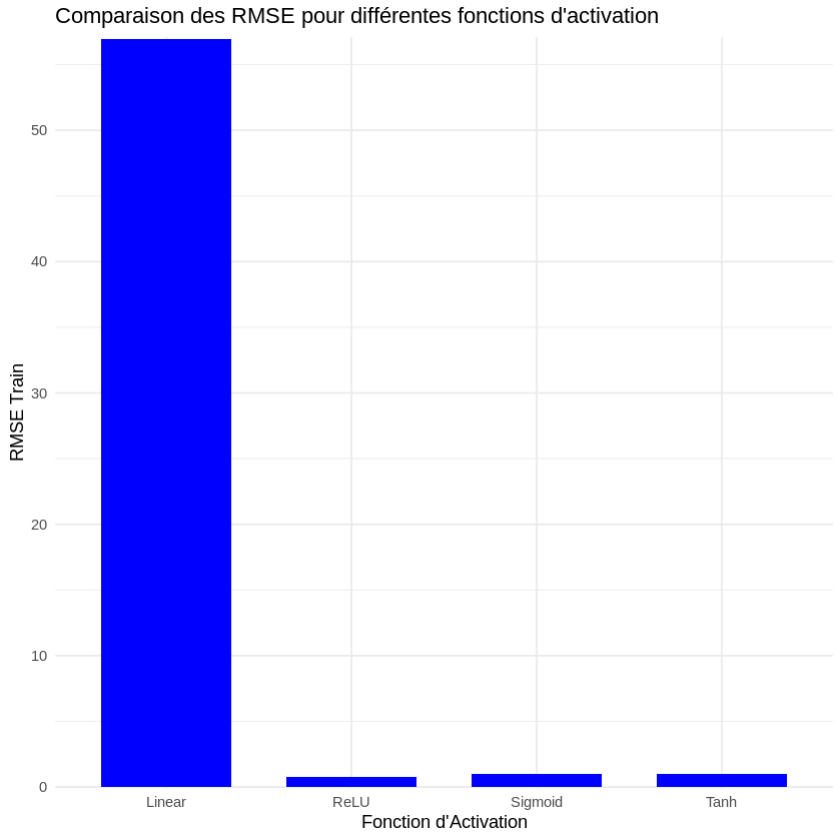
```
actFuncByLayer_sigmoid <- list(c(sigmoid, sigmoid, sigmoid, sigmoid),
c(sigmoid))

# Performance on the training set
predTrain <- forwardPropagation(Xtrain, wBest, actFuncByLayer_sigmoid)
# predictions of the NN learned
rmse_train_sigmoid <- sqrt((sum((Ytrain-predTrain)^2))/ntrain)

# Performance on the testing set
predTest <- forwardPropagation(Xtest, wBest, actFuncByLayer_sigmoid) #
predictions of the NN learned
rmse_test_sigmoid <- sqrt((sum((Ytest-predTest)^2))/ntest)

actFuncByLayer_relu <- list(c(relu, relu, relu, relu), c(relu))

# Performance on the training set
predTrain <- forwardPropagation(Xtrain, wBest, actFuncByLayer_relu) #
predictions of the NN learned
rmse_train_relu <- sqrt((sum((Ytrain-predTrain)^2))/ntrain)
```

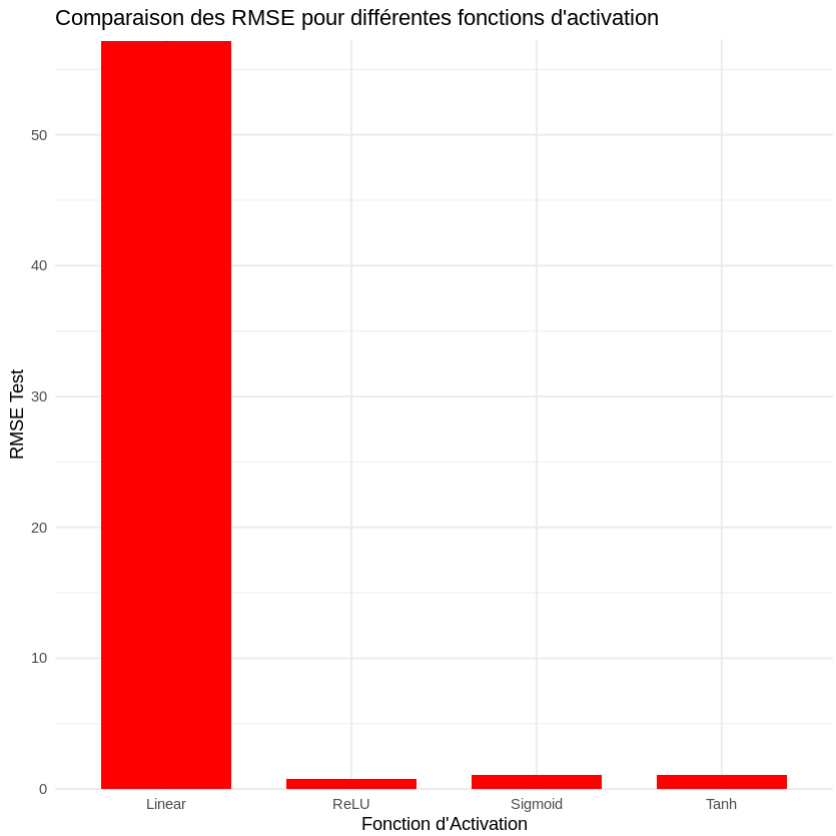
```
# Performance on the testing set
predTest <- forwardPropagation(Xtest, wBest, actFuncByLayer_relu) #
predictions of the NN learned
rmse_test_relu <- sqrt((sum((Ytest-predTest)^2))/ntest)

actFuncByLayer_linear <- list(c(linear, linear, linear, linear),
c(linear))

# Performance on the training set
predTrain <- forwardPropagation(Xtrain, wBest, actFuncByLayer_linear)
# predictions of the NN learned
rmse_train_linear <- sqrt((sum((Ytrain-predTrain)^2))/ntrain)

# Performance on the testing set
predTest <- forwardPropagation(Xtest, wBest, actFuncByLayer_linear) #
predictions of the NN learned
rmse_test_linear <- sqrt((sum((Ytest-predTest)^2))/ntest)

actFuncByLayer_tanh <- list(c(tanh, tanh, tanh, tanh), c(tanh))

# Performance on the training set
predTrain <- forwardPropagation(Xtrain, wBest, actFuncByLayer_sigmoid)
# predictions of the NN learned
rmse_train_tanh <- sqrt((sum((Ytrain-predTrain)^2))/ntrain)

# Performance on the testing set
predTest <- forwardPropagation(Xtest, wBest, actFuncByLayer_sigmoid) #
predictions of the NN learned
rmse_test_tanh <- sqrt((sum((Ytest-predTest)^2))/ntest)
```

On affiche les diagramme barre pour comparer les RMSE des fonctions

```
library(ggplot2)

data <- data.frame(
  Function = c("Sigmoid", "ReLU", "Linear", "Tanh"),
  RMSE_Train = c(rmse_train_sigmoid, rmse_train_relu,
rmse_train_linear,rmse_train_tanh),
  RMSE_Test = c(rmse_test_sigmoid, rmse_test_relu, rmse_test_linear,
rmse_test_tanh)
)

ggplot(data, aes(x = Function)) +
  geom_bar(aes(y = RMSE_Train), position = "dodge", stat = "identity",
fill = "blue", width = 0.7) +
  labs(title = "Comparaison des RMSE pour différentes fonctions
d'activation",
       y = "RMSE Train",
       x = "Fonction d'Activation") +
```

```
  scale_y_continuous(expand = c(0, 0.1)) +
  theme_minimal()

ggplot(data, aes(x = Function)) +
  geom_bar(aes(y = RMSE_Test), position = "dodge", stat =
"identity",fill="red", width = 0.7) +
  labs(title = "Comparaison des RMSE pour différentes fonctions
d'activation",
       y = "RMSE Test",
       x = "Fonction d'Activation") +
  scale_y_continuous(expand = c(0, 0.1)) +
  theme_minimal()
```

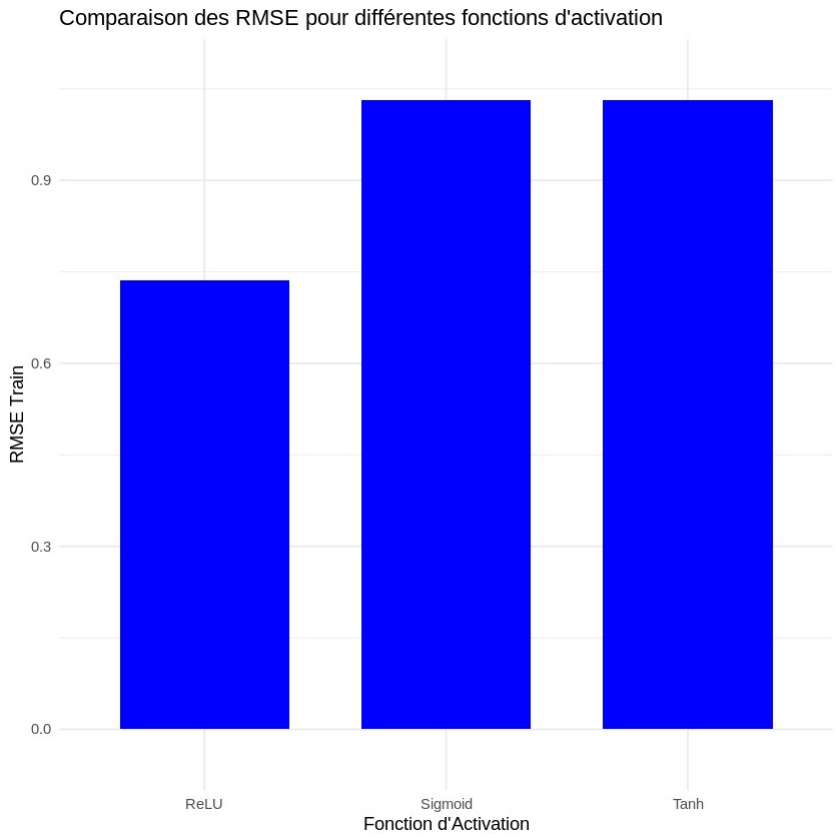


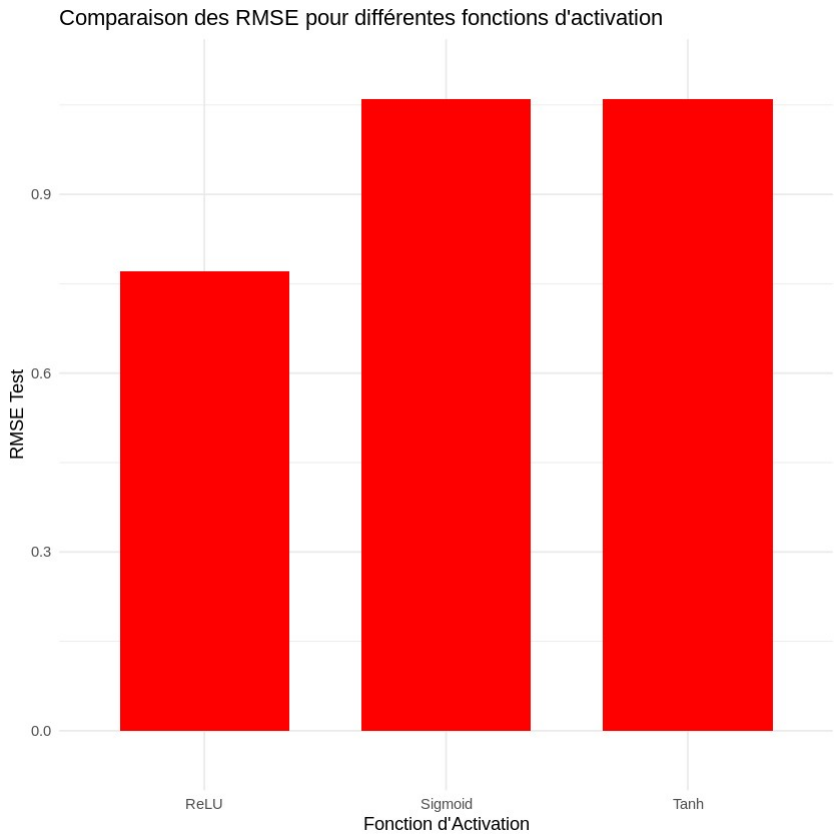


On peut observer ci dessus que le RMSE de la fonction d'activation lineaire seule est très mauvais, ainsi allons nous afficher les même résultats en retirant cette dernière :

```
data <- data.frame(
  Function = c("Sigmoid", "ReLU", "Tanh"),
  RMSE_Train = c(rmse_train_sigmoid, rmse_train_relu,rmse_train_tanh),
  RMSE_Test = c(rmse_test_sigmoid, rmse_test_relu,  rmse_test_tanh)
)

ggplot(data, aes(x = Function)) +
  geom_bar(aes(y = RMSE_Train), position = "dodge", stat = "identity",
fill = "blue", width = 0.7) +
  labs(title = "Comparaison des RMSE pour différentes fonctions
d'activation",
       y = "RMSE Train",
       x = "Fonction d'Activation") +
  scale_y_continuous(expand = c(0, 0.1)) +
  theme_minimal()

ggplot(data, aes(x = Function)) +
  geom_bar(aes(y = RMSE_Test), position = "dodge", stat = "identity",
fill="red",  width = 0.7) +
  labs(title = "Comparaison des RMSE pour différentes fonctions
d'activation",
```

```
      y = "RMSE Test",
      x = "Fonction d'Activation") +
scale_y_continuous(expand = c(0, 0.1)) +
theme_minimal()
```





La fonction d'activation relu est donc la fonction d'activation qui, prise seule permet une erreur RMSE plus faible en test et en training dans notre cas. Mais les fonctions d'activations sigmoid et tanh ont un RMSE très proche.

Il convient donc maintenant de tester des fonctions d'activation différentes selon le neurone considéré. Par exemple on a pu voir préalablement que le choix des fonctions d'activation suivantes : c(sigmoid, sigmoid, relu, relu), c(linear) permet un RMSE test de 0.11 et RMSE train de 0.07. Nous allons donc faire des tests pour tenter d'en trouver le meilleur pour les données train et le meilleur pour les données test sur un réseau de neurone (1,4,2).

```
Function = c(sigmoid, relu, linear, tanh)
RMSE_Train_best <- 100
RMSE_Test_best <- 100

for (i in 1:length(Function)){
  for (j in 1:length(Function)){
    for (k in 1:length(Function)){
      for (l in 1:length(Function)){
        for (m in 1:length(Function)){
          actFuncByLayer_best <- list(c(Function[i], Function[j],
Function[k], Function[l]), c(Function[m]))

          # Performance on the training set
          predTrain <- forwardPropagation(Xtrain, wBest,
actFuncByLayer_best) # predictions of the NN learned
```

```
          rmse_train_best <- sqrt((sum((Ytrain-predTrain)^2))/ntrain)

          # Performance on the testing set
          predTest <- forwardPropagation(Xtest, wBest,
actFuncByLayer_best) # predictions of the NN learned
          rmse_test_best <- sqrt((sum((Ytest-predTest)^2))/ntest)

          if (RMSE_Train_best > rmse_train_best){
            RMSE_Train_best <- rmse_train_best
            index_1 <- c(i,j,k,l,m)
          }
          if (RMSE_Test_best > rmse_test_best){
            RMSE_Test_best <- rmse_test_best
            index_2<-c(i,j,k,l,m)
          }
        }
      }
    }
  }
}

# print des indices des fonctions : 1 -> sigmoid, 2 -> relu, 3 ->
linear, 4 -> tanh
cat("indices des fonctions d'activation pour le minimum de RMSE
Train :", index_1, "\n")
cat("indices des fonctions d'activation pour le minimum de RMSE
Test:", index_2, "\n")
```

```
indices des fonctions d'activation pour le minimum de RMSE Train : 1 1
2 3 3
indices des fonctions d'activation pour le minimum de RMSE Test: 1 1 2
2 4
```

Ainsi, pour obtenir le meilleur RMSE sur les données d'apprentissage sur notre réseau de neurone (2,4,1) on utilise les fonctions d'activation : (sigmoid, sigmoid, relu, linear)(linear)

Et pour obtenir le meilleur RMSE sur les données de tests, on utilise les fonctions d'activation : (sigmoid, sigmoid, relu, relu)(tanh)

Attention : ces valeurs sont prises pour un exemple de réseau de neurones, pour un autre réseau il ne faudra pas prendre ces fonctions d'activation mais relancer le code précédent.

Influence de la taille des échantillons de données d'apprentissage et des données test

Nous avons décider de tester l'influence de la taille des échantillons sur le réseau de neurone (1,4,2) avec les fonction d'activation trouvé précédemment (sigmoid, sigmoid, relu, relu)(tanh)

On va considérer 5 tailles d'échantillons, un où 13% des données sont dans le train et 87% dans le test, un où 25% des données sont dans le train et 75% dans le test, un où 50% des données sont dans le train et 50% dans le test, un où 75% des données sont dans le train et 25% dans le test et enfin un où 87% des données sont dans le train et 13% dans le test.

```
actFuncByLayer_te <- list(c(sigmoid, sigmoid, relu, relu), c(tanh)) #
actFuncByLayer_te : te : fonction utilisée pour la taille des
échantillon

# 1er set d'échantillon

ntrain <- 15
ntest <- 100
ndata <- ntrain + ntest
itrain <- 1:ntrain
itest <- (ntrain+1):ndata
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))

# Performance on the training set
predTrain <- forwardPropagation(Xtrain, wBest, actFuncByLayer_te) #
predictions of the NN learned
rmse_train_set_ech_1 <- sqrt((sum((Ytrain-predTrain)^2))/ntrain)

# Performance on the testing set
predTest <- forwardPropagation(Xtest, wBest, actFuncByLayer_te) #
predictions of the NN learned
rmse_test_set_ech_1 <- sqrt((sum((Ytest-predTest)^2))/ntest)

cat("RMSE Train 13/87",rmse_train_set_ech_1, "\n")
cat("RMSE Test 13/87", rmse_test_set_ech_1, "\n")

# 2ème  set d'échantillon

ntrain <- 29
ntest <- 86
ndata <- ntrain + ntest
itrain <- 1:ntrain
itest <- (ntrain+1):ndata
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))

# Performance on the training set
predTrain <- forwardPropagation(Xtrain, wBest, actFuncByLayer_te) #
predictions of the NN learned
rmse_train_set_ech_2 <- sqrt((sum((Ytrain-predTrain)^2))/ntrain)
```

```r
# Performance on the testing set
predTest <- forwardPropagation(Xtest, wBest, actFuncByLayer_te) #
predictions of the NN learned
rmse_test_set_ech_2 <- sqrt((sum((Ytest-predTest)^2))/ntest)

cat("RMSE Train 25/75",rmse_train_set_ech_2, "\n")
cat("RMSE Test 25/75 ", rmse_test_set_ech_2, "\n")

# 3ème set d'échantillon

ntrain <- 57
ntest <- 58
ndata <- ntrain + ntest
itrain <- 1:ntrain
itest <- (ntrain+1):ndata
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))

# Performance on the training set
predTrain <- forwardPropagation(Xtrain, wBest, actFuncByLayer_te) #
predictions of the NN learned
rmse_train_set_ech_3 <- sqrt((sum((Ytrain-predTrain)^2))/ntrain)

# Performance on the testing set
predTest <- forwardPropagation(Xtest, wBest, actFuncByLayer_te) #
predictions of the NN learned
rmse_test_set_ech_3 <- sqrt((sum((Ytest-predTest)^2))/ntest)

cat("RMSE Train 50/50",rmse_train_set_ech_3, "\n")
cat("RMSE Test 50/50", rmse_test_set_ech_3, "\n")

# 4ème set d'échantillon

ntrain <- 86
ntest <- 29
ndata <- ntrain + ntest
itrain <- 1:ntrain
itest <- (ntrain+1):ndata
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))

# Performance on the training set
predTrain <- forwardPropagation(Xtrain, wBest, actFuncByLayer_te) #
predictions of the NN learned
rmse_train_set_ech_4 <- sqrt((sum((Ytrain-predTrain)^2))/ntrain)
```

```r
# Performance on the testing set
predTest <- forwardPropagation(Xtest, wBest, actFuncByLayer_te) #
predictions of the NN learned
rmse_test_set_ech_4 <- sqrt((sum((Ytest-predTest)^2))/ntest)

cat("RMSE Train 75/25",rmse_train_set_ech_4, "\n")
cat("RMSE Test 75/25", rmse_test_set_ech_4, "\n")

# 5ème set d'échantillon

ntrain <- 100
ntest <- 15
ndata <- ntrain + ntest
itrain <- 1:ntrain
itest <- (ntrain+1):ndata
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))

# Performance on the training set
predTrain <- forwardPropagation(Xtrain, wBest, actFuncByLayer_te) #
predictions of the NN learned
rmse_train_set_ech_5 <- sqrt((sum((Ytrain-predTrain)^2))/ntrain)

# Performance on the testing set
predTest <- forwardPropagation(Xtest, wBest, actFuncByLayer_te) #
predictions of the NN learned
rmse_test_set_ech_5 <- sqrt((sum((Ytest-predTest)^2))/ntest)

cat("RMSE Train 87/13",rmse_train_set_ech_5, "\n")
cat("RMSE Test 87/13", rmse_test_set_ech_5, "\n")
```

```
RMSE Train 13/87 0.136235
RMSE Test 13/87 0.1570829
RMSE Train 25/75 0.1402741
RMSE Test 25/75  0.1590405
RMSE Train 50/50 0.1513829
RMSE Test 50/50 0.1575483
RMSE Train 75/25 0.1514956
RMSE Test 75/25 0.1631715
RMSE Train 87/13 0.1509434
RMSE Test 87/13 0.1765432
```

On remarque avec ces résultats que la taille de l'ensemble d'apprentissage par rapport à l'ensembre test n'a pas beaucoup d'incidence sur les résultats. Le seul élément remarquable est qu'avec très peu de données d'apprentissage, on obtient un RMSE Train très faible et un RMSE Test très élevé et inversement pour un très grands nombre de données d'apprentissages

Etudier les capacités de régression de réseaux sur plusieurs fonctions

```
install.packages("neuralnet", dependencies=TRUE)
library(neuralnet)

# Générer des données synthétiques pour une fonction, par exemple,
f(x) = x^2
set.seed(123)
n <- 100
x <- seq(-2, 2, length.out = n)
y <- x^2 + rnorm(n, mean = 0, sd = 0.1)  # Ajouter un peu de bruit à
la fonction

# Créer un dataframe avec les données
data <- data.frame(x = x, y = y)

# Définir l'architecture du réseau de neurones
architecture <- c(1, 5, 1)  # 1 entrée, 5 neurones cachés, 1 sortie

# Définir la fonction de perte (loss) et l'algorithme d'optimisation
(SGD)
loss_function <- "sse"  # Sum of Squared Errors
algorithm <- "backprop" # Backpropagation (qui est la méthode utilisée
par défaut dans neuralnet)

# Créer et entraîner le modèle de réseau de neurones
model <- neuralnet(y ~ x, data = data, hidden = architecture[2],
linear.output = TRUE, algorithm = algorithm, err.fct = loss_function)

# Visualiser les prédictions par rapport aux données réelles
plot(x, y, main = "Régression avec un réseau de neurones", xlab = "x",
ylab = "y", col = "blue", pch = 16)
lines(x, predict(model, data), col = "red", lwd = 2)
legend("topright", legend = c("Données réelles", "Prédictions"), col =
c("blue", "red"), lty = 1:1, cex = 0.8)
```

```
Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)


Error: Argument 'learningrate' must be a numeric value, if the
backpropagation algorithm is used.
Traceback:

1. neuralnet(y ~ x, data = data, hidden = architecture[2],
linear.output = TRUE,
 .     algorithm = algorithm, err.fct = loss_function)
2. stop("Argument 'learningrate' must be a numeric value, if the
```

```
backpropagation algorithm is used.",
 .     call. = FALSE)
```

#Etudier la flexibilité des réseaux de neurones

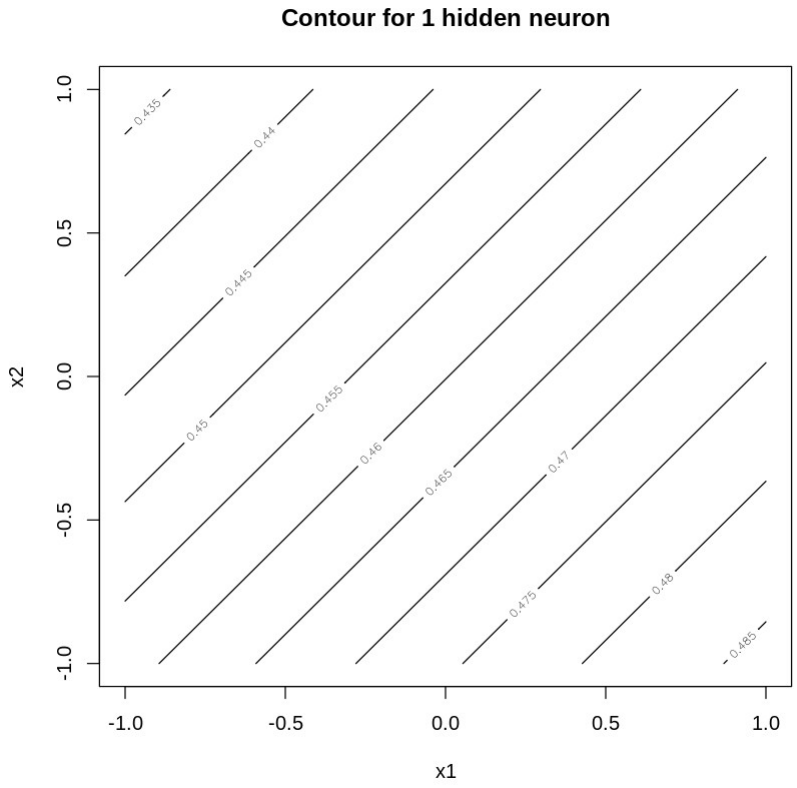
– on pourra reprendre les dessins faits en Question 1.e. Répéter l'opération pour 1, 2, et 4 neurones cachés. Eventuellement ajouter une couche. Commenter.

L'objectif est d'évaluer la flexibilité des réseaux de neurones artificiels à travers différentes configurations architecturales. La flexibilité est évaluée en termes de la capacité du réseau à modéliser des relations complexes entre les entrées et les sorties. Les configurations étudiées varient en nombre de neurones cachés et en nombre de couches cachées.

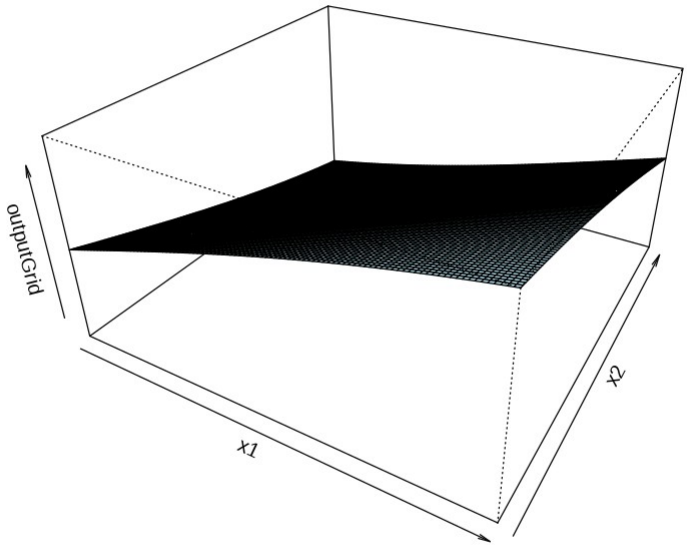
```
library(neuralnet)
library(rgl)

# Fonction d'activation sigmoid
sigmoid <- function(x) {
  1 / (1 + exp(-x))
}

# Fonction pour générer des poids aléatoires pour les couches du
réseau
createRandomWeightsByLayer <- function(layerSizes) {
  weights <- list()
  for (i in 1:(length(layerSizes) - 1)) {
    weights[[i]] <- matrix(runif(layerSizes[i] * layerSizes[i + 1], -
1, 1),
                           nrow = layerSizes[i], ncol = layerSizes[i +
1])
  }
  return(weights)
}

# Fonction de propagation avant
forwardPropagation <- function(inputs, weights, actFunc) {
  for (i in 1:length(weights)) {
    inputs <- actFunc(inputs %*% weights[[i]])
  }
  return(inputs)
}
# Fonction pour générer et afficher les graphiques
generateAndPlot <- function(layerSizes) {
  Weights <- createRandomWeightsByLayer(layerSizes)

  # Création d'une grille 2D d'entrées
  gridLength <- 100
  x1 <- seq(-1, 1, length.out = gridLength)
  x2 <- seq(-1, 1, length.out = gridLength)
  grid <- expand.grid(x1, x2)
```

```
  inputs <- as.matrix(grid)

  # Propagation avant pour obtenir les sorties
  outputs <- forwardPropagation(inputs, Weights, sigmoid)
  outputGrid <- matrix(outputs, nrow = gridLength, ncol = gridLength)

  # Titre pour les graphiques en fonction du nombre de neurones cachés
  title_suffix <- ifelse(length(layerSizes) - 2 == 1, "neuron",
"neurons")
  plot_title <- paste("Contour for", length(layerSizes) - 2,
title_suffix)

  # Graphique 2D (contour)
  contour(x1, x2, outputGrid, xlab = "x1", ylab = "x2", main =
plot_title)

  # Graphique 3D
  plot_title <- paste("3D Surface for", length(layerSizes) - 2,
title_suffix)
  persp(x1, x2, outputGrid, theta = 30, phi = 30, expand = 0.5, col =
'lightblue', main = plot_title)
}


# Définition de la fonction generateAndPlot
generateAndPlot <- function(layerSizes, title) {
  Weights <- createRandomWeightsByLayer(layerSizes)

  # Création d'une grille 2D d'entrées
  gridLength <- 100
  x1 <- seq(-1, 1, length.out = gridLength)
  x2 <- seq(-1, 1, length.out = gridLength)
  grid <- expand.grid(x1, x2)
  inputs <- as.matrix(grid)

  # Propagation avant pour obtenir les sorties
  outputs <- forwardPropagation(inputs, Weights, sigmoid)
  outputGrid <- matrix(outputs, nrow = gridLength, ncol = gridLength)

  # Graphique 2D (contour)
  contour(x1, x2, outputGrid, xlab = "x1", ylab = "x2", main =
paste("Contour for", title))

  # Graphique 3D
  persp(x1, x2, outputGrid, theta = 30, phi = 30, expand = 0.5, col =
'lightblue', main = paste("3D Surface for", title))
}

# N'oubliez pas de redéfinir les fonctions sigmoid et
createRandomWeightsByLayer si nécessaire.
```

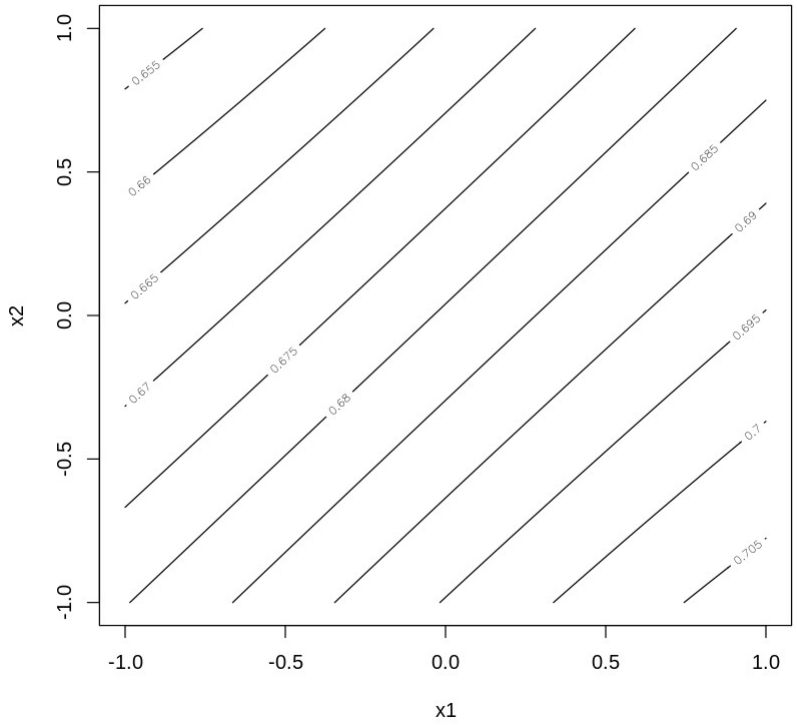
```r
# Ensuite, utilisez la fonction generateAndPlot avec les nouveaux
titres comme suit:
generateAndPlot(c(2, 1, 1), "1 hidden neuron")
generateAndPlot(c(2, 2, 1), "2 hidden neurons")
generateAndPlot(c(2, 4, 1), "4 hidden neurons")
generateAndPlot(c(2, 2, 2, 1), "1 additional layer")
```



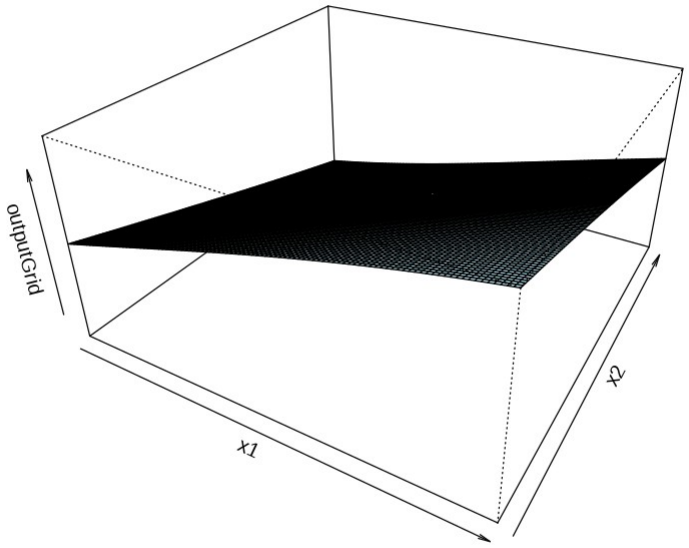
3D Surface for 1 hidden neuron



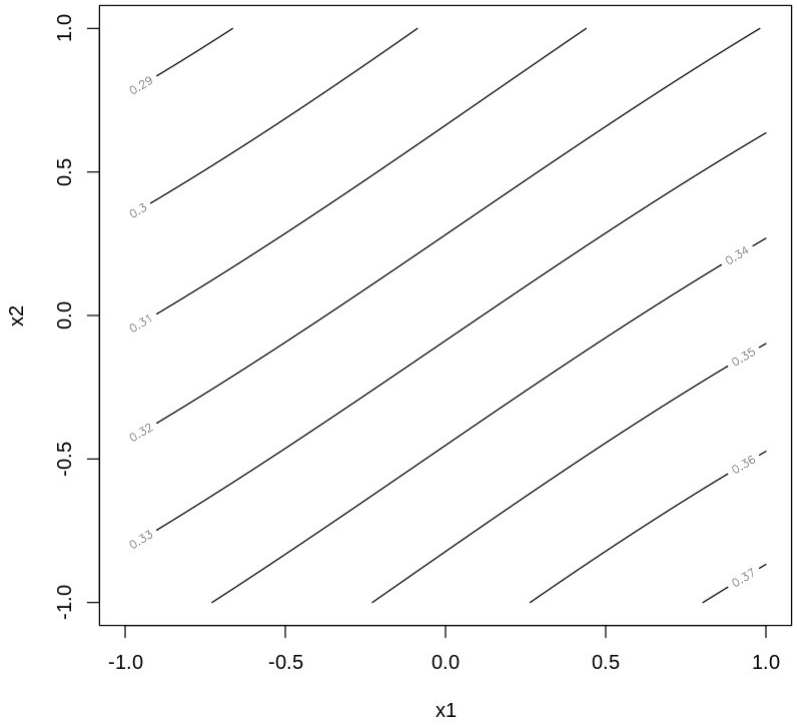
Contour for 2 hidden neurons



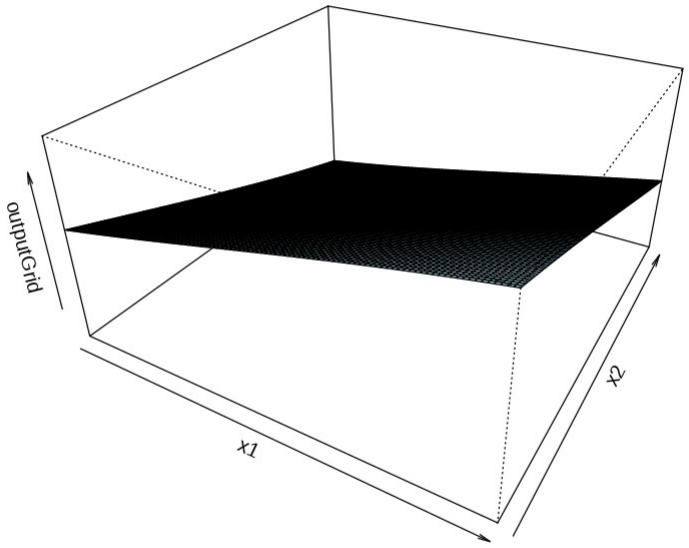
3D Surface for 2 hidden neurons



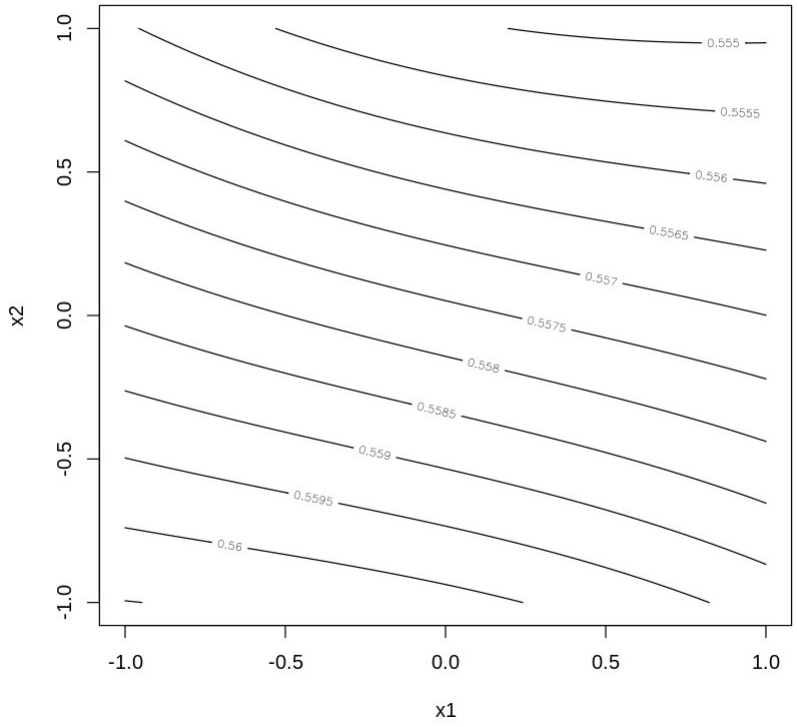
Contour for 4 hidden neurons

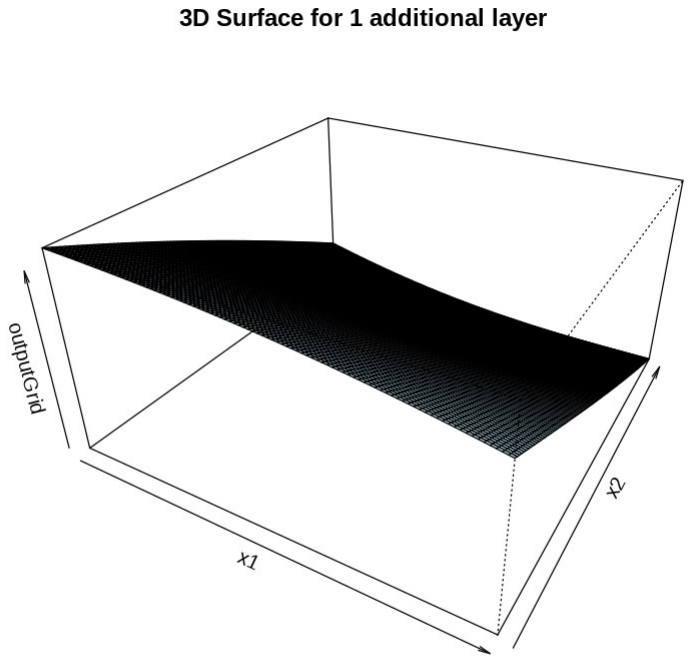


3D Surface for 4 hidden neurons



Contour for 1 additional layer





Analyse des résultats obtenus :

Configuration avec 1 neurone caché : Les résultats montrent une fonction de décision simple, comme le démontre la surface 3D avec une inclinaison uniforme. Le réseau avec un seul neurone caché présente une flexibilité très limitée, capable uniquement de capturer des comportements linéaires ou quasi-linéaires entre les variables x1 et x2.

Configuration avec 2 neurones cachés : Une légère augmentation de la complexité est observée. La surface 3D montre une pente similaire à celle avec un seul neurone caché, mais avec une courbure minime suggérant une capacité accrue à modéliser des non-linéarités discrètes. Cette configuration indique une flexibilité modeste, permettant au réseau de saisir des dynamiques légèrement plus complexes.

Configuration avec 4 neurones cachés : Avec quatre neurones cachés, il est attendu que le réseau puisse modéliser des relations plus sophistiquées. Les graphiques révèlent une complexité marginale additionnelle. Cependant, si les graphiques ne montrent pas de changements significatifs par rapport à la configuration à deux neurones cachés, cela pourrait suggérer des contraintes dans la capacité d'apprentissage du réseau ou la nécessité d'ajuster d'autres paramètres du modèle.

Configuration avec une couche supplémentaire : Normalement, l'ajout d'une couche supplémentaire devrait introduire une augmentation notable de la flexibilité du réseau. Cela n'est cependant pas évident dans les résultats obtenus, suggérant que l'ajout d'une couche n'a pas eu l'effet escompté sur la capacité du réseau à modéliser des fonctions complexes. Cela

pourrait être attribué à des facteurs tels que l'initialisation des poids ou la sélection des fonctions d'activation.

En conclusion:

Les résultats expérimentaux indiquent que la complexité et la flexibilité du réseau de neurones s'accroissent avec le nombre de neurones cachés, bien que cette augmentation ne soit pas aussi marquée que théoriquement attendue.

L'introduction d'une couche supplémentaire n'a pas non plus produit une amélioration significative de la modélisation des relations complexes.

Ces observations suggèrent qu'il pourrait être nécessaire de revoir les stratégies d'initialisation, les fonctions d'activation ou les protocoles d'entraînement pour exploiter pleinement la capacité des réseaux de neurones à apprendre des représentations de données hautement non linéaires. Pour une évaluation complète, une analyse des performances sur un ensemble de test séparé serait également cruciale pour juger de la capacité du réseau à généraliser à partir de ses apprentissages.

Formulation de la Souplesse des Réseaux de Neurones

La souplesse d'un réseau de neurones, souvent qualifiée de capacité ou complexité du modèle, se rapporte à la variété des fonctions que le réseau peut apprendre. Pour quantifier cette souplesse dans le contexte d'une architecture de réseau donnée, nous proposons une formule qui considère à la fois la structure et la topologie du réseau tout en restant indépendante des valeurs spécifiques des poids et biais.

Une approche pour formuler cette souplesse est de comptabiliser le nombre total de paramètres ajustables, y compris les poids et les biais. Cependant, ce nombre doit être pondéré par la capacité de chaque paramètre à contribuer à la complexité du modèle. La formule proposée est la suivante :

La souplesse d'un réseau de neurones peut être formulée comme suit :

$ \text{Souplesse}(N, T, L) = \sum_{i=1}^{L} T_i (N_{i-1} \times N_i + N_i) $

où :

- L est le nombre total de couches dans le réseau,
- N_i représente le nombre de neurones dans la couche i,
- T_i est un terme de pondération correspondant au type de neurone ou de fonction d'activation dans la couche i,
- $N_{i-1} \times N_i$ calcule le nombre de poids entre la couche $i-1$ et la couche i,
- N_i additionnel représente le nombre de biais dans la couche i.

Le terme de pondération T_i doit être choisi de manière à refléter la capacité de la fonction d'activation utilisée dans la couche $_i$ à modéliser des non-linéarités. Par exemple, pour une

fonction d'activation ReLU, on pourrait choisir un T_i plus grand que pour une fonction sigmoïde en raison de sa capacité à activer des neurones de manière non saturante, ce qui augmente la souplesse du réseau.

Il est important de noter que cette formule est une approximation et que d'autres facteurs tels que l'architecture globale du réseau, les mécanismes de régularisation, et l'initialisation des poids jouent également un rôle crucial dans la détermination de la souplesse réelle du réseau. De plus, des concepts théoriques tels que la dimension de Vapnik-Chervonenkis (VC) ou la complexité de Rademacher peuvent offrir des perspectives plus formelles, bien que leur calcul soit souvent non trivial.

En conclusion, la souplesse estimée par cette formule offre un aperçu de la capacité intrinsèque du réseau à apprendre des fonctions variées. Néanmoins, l'évaluation empirique de la performance du modèle sur des données de validation et de test reste essentielle pour comprendre pleinement sa capacité à généraliser.