Deep learning for predicting cancer

Martin Lumiste

***Abstract*—I build a deep neural network from scratch as an educational exercise. It consists of 2 hidden sigmoid layers, uses backpropagation of errors via batch gradient descent and a fixed learning rate without regularization or parameter tuning. Surprisingly, this simple framework beats both logistic regression and R package *neuralnet* in classification on a breast cancer biopsy dataset where the objective is to predict whether a tumor is malignant.**

***Index Terms*—neural networks, machine learning, deep learning**

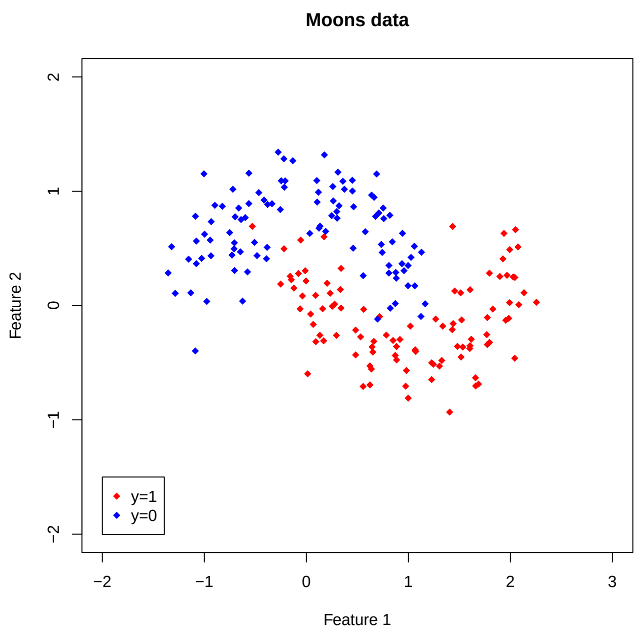
I. Introduction

Deep neural networks are often hailed as omnipotent and mysterious, whereas in reality they are simple sieve estimation methods designed to fit nonlinearities in the data. As all statistical models, they have their pros and cons, one of the former being the ability to sidestep feature engineering and learn nonlinear decision rules straight from the features. A multi-layer feedforward network also has theoretical guarantees of being a *universal approximator*. This means that regardless of the actual functional form of $\mathbb{E}[Y|X]$ (this is what we are trying to find under MSE loss), our model should converge sufficiently close to it. I develop a "deep" neural network with multiple hidden layers to test whether we can improve classification accuracy over conventional models on two standard datasets, one being artifical data and the other real data. My network architecture makes use of the sigmoid function, partly because of its convenient derivative but also because historically it was the first one for which this universal approximation property was shown.

II. Data

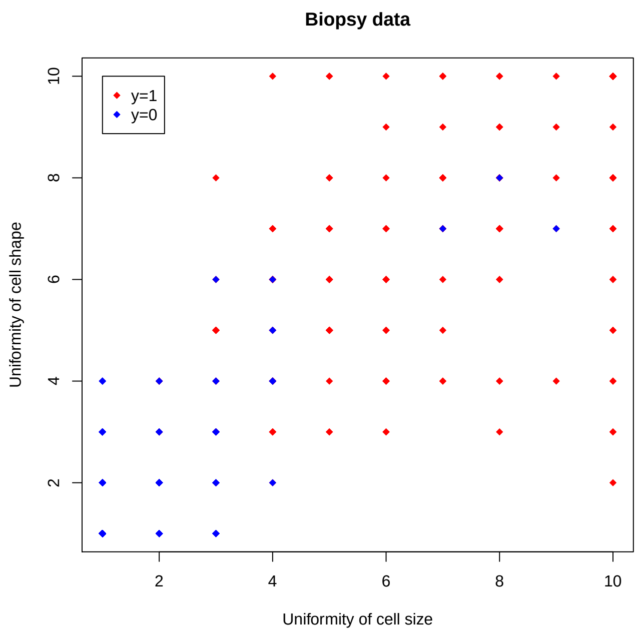
Two datasets were used for prediction. Firstly, I generate pseudodata from the *sci-kit learn* module in Python similarly to this author who implements a NN from scratch, albeit in Python, with less layers and a different architecture. The features are of dimension 200×2 with a binary outcome. The second is a classic R dataset: breast cancer biopsy results from University of Wisconsin, Madison Hospitals study by Dr. Wolberg. $K = 9$ features were measured on $N = 611$ patients on a scale from 1 to 10. These were: clump thickness, uniformity of cell size, uniformity of cell shape, marginal adhesion, single epithelial cell size, bare nuclei, bland chromatin, normal nucleoli, mitoses. Each case was deemed either benign or malignant.

martin.lumiste@gmail.com, +372 5646 3207



When viewing our Python pseudodata outcomes in the two-dimensional space generated by the features, we see where the *make moons* method in Python gets its name: the clusters of data form two moons. If we were tasked with fitting a line which separates blue and red points with maximum accuracy (but this, in its essence, is the machine learning task of classification), this would not be a straight line. In technical terms, this means that the optimal line (decision boundary) is nonlinear. We know that logistic regression on the raw features can only fit linear decision boundaries, therefore we will consider it as a benchmark to beat with neural network. We also know intuitively that we do not want the decision line to be too complex. Even though with a model that is complicated enough, we could fit this sort of line perfectly to our training sample data, this would likely not *generalize* well for predicting on other data sets. The machine learning task of hyperparameter tuning deals with trying to find this optimal trade-off between complexity and out of sample fit. I omit parameter tuning for now.

A similar image can be drawn from the biopsy data, although the feature space is 9-dimensional so we need to pick 2 plotting features. We see that despite being a real-life "noisy" dataset, there is a clear tendency for malignant ($y = 1$) tumors in the higher value regions of the features. Both datasets are randomly partitioned into 60/40 shares of training/test sets.



III. Data Analysis

Firstly, I run a logistic regression to assess its out of sample prediction fit as a benchmark for more complex models. As this model outputs predicted probabilities, $\hat{y}$, let us define a cut-off rule for the predicted outcome $\bar{y}$:

$$\bar{y} = \mathbb{1}_{\hat{y}>0.5}$$

Also define the accuracy measure as

$$A_N = \frac{1}{N}\sum_{n=1}^{N} \mathbb{1}_{\bar{y}=y}$$

where y is the actual outcome.

```
log.fitm = glm(y ~ .,
               family=binomial(link='logit'),
               data=moons[sampm,])
log.predm = predict(log.fitm,
                    newdata=moons[-sampm,], type='response')
log.predm = ifelse(log.predm > 0.5,1,0)

mean(moons[-sampm,]$y == log.predm)

## [1] 0.88

log.fitb = glm(bin ~ .,
               family = binomial(link='logit')
               ,data=biopsy[sampb,])
log.predb = predict(log.fitb,
                    newdata=biopsy[-sampb,], type='response')
log.predb = ifelse(log.predb > 0.5,1,0)

mean(biopsy[-sampb,]$bin == log.predb)

## [1] 0.95
```

In practice, we can easily evaluate this in R. We see that our vanilla logistic regression model has 87.5 % accuracy on the moons dataset and 94.89 % on biopsy. Let us see if we can improve on this using a neural network model. I will design a network with the following structure:

Rplot01.pdf

It therefore has 2 hidden layers of size 12, with bias included. I use the sigmoid activation function in all layers, including the output. To this end, let us define the learning rate (which sets the speed of parameter changing with each iteration) and helper functions for sigmoid and its derivative (which will be needed for backpropagation).

```
eta = 0.05
sigmoid = function(x) 1/(1 + exp(-x))
der = function(x) sigmoid(x) * (1 - sigmoid(x))
```

The commented code for training the deep neural network is then as follows:

```
reg.dnn = function(x,y,traindata,iter,hidden,eta){
  #Set seed for reproducibility
  set.seed(11615)
  X = unname(data.matrix(traindata[,x]))
  #Scale features as the sigmoid function does not love
  #extreme inputs
  X = scale(X)
  Y = as.integer(traindata[,y])
  #Initialize random weights and biases
  W1 = matrix(rnorm(ncol(X)*hidden[1]),ncol=hidden[1])
  W2 = matrix(rnorm(hidden[1]*hidden[2]),ncol=hidden[2])
  W3 = matrix(rnorm(hidden[2]),nrow=hidden[2])
  b1 = matrix(0, nrow=1, ncol=hidden[1])
  b2 = matrix(0, nrow=1, ncol=hidden[2])
  b3 = matrix(0, nrow=1, ncol=1)
  #Rather than convergence of the loss function,
  #we simply run the algorithm many times
  for (i in 1:iter) {
    #Feedforward to calculate the outcomes
    S2 = sweep(X %*% W1 ,2, b1, '+')
    Z2 = sigmoid(S2)
    S3 = sweep(Z2 %*% W2 ,2, b2, '+')
    Z3 = sigmoid(S3)
    S4 = sweep(Z3 %*% W3 ,2, b3, '+')
    Z4 = sigmoid(S4)
    if (i %% 100 == 0){
    cat("Iteration",i,"out of",iter,"\n")
    }
    #Backpropagation of errors
    d4 = Z4 - Y
    d3 = d4 * der(Z4)
    e2 = d3 %*% t(W3)
    d2 = e2 * der(Z3)
    e1 = d2 %*% t(W2)
    d1 = e1 * der(Z2)
    #Calculate changes
    dW3 = t(Z3) %*% d3
    dW2 = t(Z2) %*% d2
    dW1 = t(X) %*% d1

    db3 = colSums(d3)
    db2 = colSums(d2)
    db1 = colSums(d1)
    #Update
    W3 = W3 - eta * dW3
    W2 = W2 - eta * dW2
    W1 = W1 - eta * dW1
    b3 = b3 - eta * db3
    b2 = b2 - eta * db2
    b1 = b1 - eta * db1
    #Save model
    model  = list(W1=W1,W2=W2,W3=W3,b1=b1,b2=b2,b3=b3)
  }
  return(model)
}
```

The backpropagation is arguably the most difficult part of the network, but it derives from simple chain rule. We want to minimize the implicit loss function $L = \mathbb{1}\frac{1}{2}(y-\bar{y})^2$ where y has dimension $N \times 1$ and $\mathbb{1}$ is a $1 \times N$ vector of ones. Minimization theory tells us that to iteratively find θ such that it minimizes some function $L(\theta)$ we need to update it as follows:

$$\theta_{t+1} = \theta_t - \alpha\frac{\partial L}{\partial \theta_t}(\theta_t) \quad (1)$$

Let us write

$$L = \mathbb{1}\frac{1}{2}\Big(y - \sigma(\sigma(\sigma(xW_1 + b_1)W_2 + b_2)W_3 + b_3)\Big)^2$$

where W_1, W_2, W_3 are the weighting matrices and b_1, b_2, b_3 the bias vectors and σ the sigmoid function. The task is to find the set of these so as to minimize L. We have:

$$\frac{\partial L}{\partial W_3} = (y - \hat{y}) \cdot \sigma'(Z_4) Z_3^T$$

$$\frac{\partial L}{\partial W_2} = (y - \hat{y}) \cdot \sigma'(Z_4) W_3 \sigma'(Z_3) Z_2^T$$

$$\frac{\partial L}{\partial W_1} = (y - \hat{y}) \cdot \sigma'(Z_4) W_3 \sigma'(Z_3) W_2 \sigma'(Z_2) Z_1^T$$

$$\frac{\partial L}{\partial b_3} = \mathbb{1}(y - \hat{y}) \cdot \sigma'(Z_4)$$

$$\frac{\partial L}{\partial b_2} = \mathbb{1}(y - \hat{y}) \cdot \sigma'(Z_4) W_3 \sigma'(Z_3)$$

$$\frac{\partial L}{\partial b_1} = \mathbb{1}(y - \hat{y}) \cdot \sigma'(Z_4) W_3 \sigma'(Z_3) W_2 \sigma'(Z_1)$$

where $Z_1 = x$, $Z_2 = \sigma(Z_1 W_1 + b_1)$, $Z_3 = \sigma(Z_2 W_2 + b_2)$, $Z_4 = \sigma(Z_3 W_3 + b_3)$ and $\cdot$ denotes elementwise multiplication. Notice how the bias derivatives are multiplied by a vector of ones, so they effectively become sums. After plugging these derivatives into the updating rule (1), we get the same formulas as in the code. The prediction function is just more of the same.

```r
predict.dnn <- function(model, newdata = testdata) {
  X = scale(data.matrix(newdata))
  S2 =  sweep(X %*% model$W1 ,2, model$b1, '+')
  Z2 = sigmoid(S2)
  S3 = sweep(Z2 %*% model$W2, 2, model$b2, '+')
  Z3 = sigmoid(S3)
  S4 = sweep(Z3 %*% model$W3 ,2, model$b3, '+')
  Z4 = sigmoid(S4)

  result = Z4
  result = ifelse(result > 0.5,1,0)
  return(result)
}
```

Let us check the performance of this model on our two toy datasets.

```r
#This code chunk will load a while without cache
modelm = reg.dnn(1:2,3,moons[sampm,],10000,c(12,12),eta)
resultm = predict.dnn(modelm,moons[-sampm,1:2])
mean(moons[-sampm,]$y == resultm)

#Biopsy data
modelb = reg.dnn(1:9,10,biopsy[sampb,],10000,c(12,12),eta)
resultb = predict.dnn(modelb,biopsy[-sampb,1:9])
mean(biopsy[-sampb,]$bin == resultb)
```

The accuracies are 86.25 % and 97.08 % respectively. Now we compare with the R package *neuralnet* which allows for multiple hidden layers in the model. It should be kept in mind that although I enforce the same seed and layer size, the activation functions in *neuralnet* are different.

```r
set.seed(11615)
nn.fitm = neuralnet(y ~ x1 + x2, data=moons[sampm,],
                    hidden = c(12,12),
                    stepmax=10000,linear.output = F)
#nn.fitm = nnet(y~.,data=moons[sampm,],size=12,maxit=10000)
nn.predm = compute(nn.fitm,moons[-sampm,1:2])$net.result
nn.predm = ifelse(nn.predm > 0.5,1,0)
mean(moons[-sampm,]$y == nn.predm)
```

```
## [1] 0.9625
```

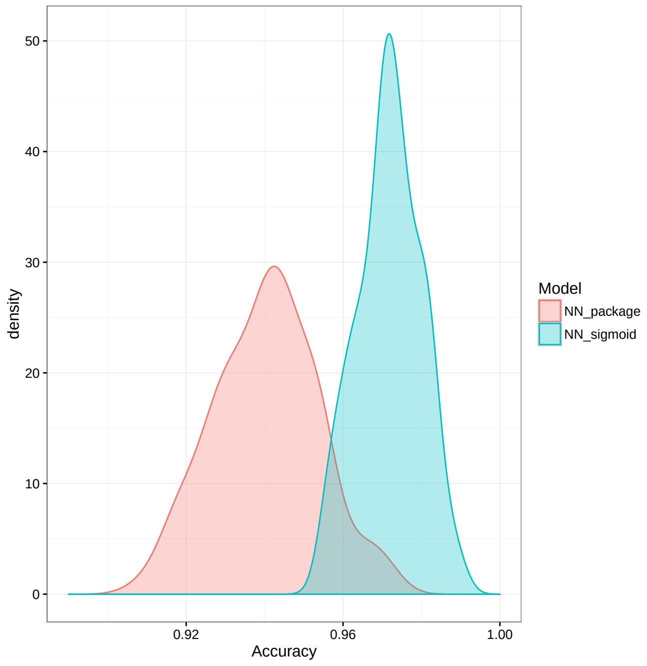
```r
set.seed(11615)
formula = bin ~ V1+V2+V3+V4+V5+V6+V7+V8+V9
nn.fitb = neuralnet(formula, data=biopsy[sampb,],
                    hidden=c(12,12),stepmax=10000)
nn.predb = compute(nn.fitb,biopsy[-sampb,1:9])$net.result
nn.predb = ifelse(nn.predb > 0.5,1,0)
mean(biopsy[-sampb,]$bin == nn.predb)
```

```
## [1] 0.9452554745
```

We achieve 96.25 % and 94.53 % accuracies. Not surprisingly, the optimized and tested R library outperforms both my model and logistic regression by a wide margin on the moons data set which is built for classification. However, the four-layer sigmoid model remains best of the three in terms of predictive accuracy on the biopsy data set. Obviously, the small size of both datasets (200 and 611) means that results are sensitive to sampling of the train/test sets. Let us run 100 Monte Carlo repetitions to gauge the ”true” accuracy on this sample.

```r
MC = 100
MCmat = matrix(NA,nrow=MC,ncol=6)
colnames(MCmat) = c("Logisticm","Logisticb",
                    "NNSm","NNSb","NNm","NNb")
for (i in 1:MC){
  tryCatch({
  set.seed(i)
  print(i)
  sampm = sample(1:dim(moons)[1],0.6*dim(moons)[1])
  sampb = sample(1:dim(biopsy)[1],0.6*dim(biopsy)[1])
  log.fitm = glm(y ~ .,
                 family=binomial(link='logit'),
                 data=moons[sampm,])
  log.predm = predict(log.fitm,
                      newdata=moons[-sampm,], type='response')
  log.predm = ifelse(log.predm > 0.5,1,0)
  acclm = mean(moons[-sampm,]$y == log.predm)

  log.fitb = glm(bin ~ .,
                 family = binomial(link='logit'),
                 data=biopsy[sampb,])
  log.predb = predict(log.fitb,
                      newdata=biopsy[-sampb,], type='response')
  log.predb = ifelse(log.predb > 0.5,1,0)
  acclb = mean(biopsy[-sampb,]$bin == log.predb)

  modelm = reg.dnn(1:2,3,moons[sampm,],10000,c(12,12),eta)
  resultm = predict.dnn(modelm,moons[-sampm,1:2])
  accnm = mean(moons[-sampm,]$y == resultm)

  modelb = reg.dnn(1:9,10,biopsy[sampb,],10000,c(12,12),eta)
  resultb = predict.dnn(modelb,biopsy[-sampb,1:9])
  accnb = mean(biopsy[-sampb,]$bin == resultb)

  nn.fitm = neuralnet(y ~ x1 + x2,
                      data=moons[sampm,], hidden = c(12,12),
                      stepmax=10000,linear.output = F)
  nn.predm = compute(nn.fitm,moons[-sampm,1:2])$net.result
  nn.predm = ifelse(nn.predm > 0.5,1,0)
  accnnm = mean(moons[-sampm,]$y == nn.predm)

  formula = bin ~ V1+V2+V3+V4+V5+V6+V7+V8+V9
  nn.fitb = neuralnet(formula, data=biopsy[sampb,],
                      hidden=c(12,12),stepmax=10000)
  nn.predb = compute(nn.fitb,biopsy[-sampb,1:9])$net.result
  nn.predb = ifelse(nn.predb > 0.5,1,0)
  accnnb = mean(biopsy[-sampb,]$bin == nn.predb)

  MCmat[i,1] = acclm
  MCmat[i,2] = acclb
  MCmat[i,3] = accnm
  MCmat[i,4] = accnb
  MCmat[i,5] = accnnm
  MCmat[i,6] = accnnb
  }, error=function(e){})
}
```

The average accuracies for the three models over MC repetitions are: 84.58 % and 96.67 % for logistic regression, 84.58 % and 97.18 % for the sigmoid neural network, 93.94 % and 94.04 % for the packaged network. The results are in line with our initial findings. Finally, let us plot the nonparametric distributions of accuracies for the two network models on the biopsy data set: we confirm that our model beats the *neuralnet* one.



This small experiment is enough to show that some elementary calculus and a few lines of code can provide a powerful classifier which holds its own against logistic regression and even more sophisticated neural network models. Finally, the author has reconfirmed for himself that NN remains a volatile method, as small changes to the sampling, seeds or architecture of the network can affect results significantly. For this reason, more stable ensemble models such as random forest should be preferred for preliminary results as NNs need a lot of data-dependent tuning.

Appendix A
Python code for generating the pseudodata

```
# Generate a dataset and plot it
import numpy as np
import sklearn.datasets
import matplotlib.pyplot as plt

np.random.seed(0)
X, y = sklearn.datasets.make_moons(200, noise=0.20)
plt.scatter(X[:,0], X[:,1], s=40, c=y, cmap=plt.cm.Spectral)
```

References

[1] G. Cybenko, *Approximations by superpositions of sigmoidal functions*, Mathematics of Control, Signals and Systems, 2(4), 303-314, 1989
[2] T. Hastie, R. Tibshirani and J. Friedman, *The Elements of Statistical Learning*, 2nd ed, Springer Series in Statistics, 2009
[3] K. Hornik, *Approximation Capabilities of Multilayer Feedforward Networks*, Neural Networks, 4(2), 251-257, 1991