Deep Neural Nets

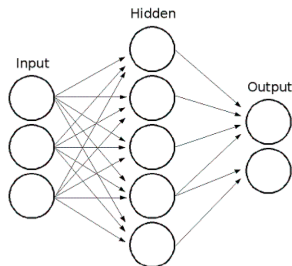
Mladen Kolar and Rob McCulloch

1. Deep Neural Nets
2. XOR
3. Tabloid
4. MNIST Digit Recognition
5. More on Fitting Neural Nets
6. DNN Grid Search for MNIST
7. More on Digit Recognition
8. Back Propagation

1. Deep Neural Nets

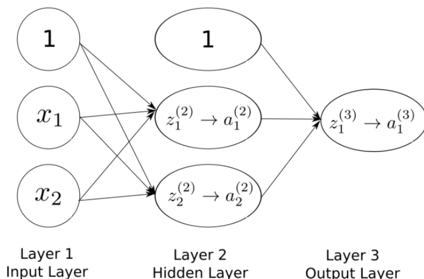
Single Layer Neural Nets

People often depict $x = (x_1, x_2, \dots, x_p)$ as an input layer with a node for each x_i.



Note that this picture is for a three-dimensional x and we now draw a line from each x_i to each hidden unit (or neuron).

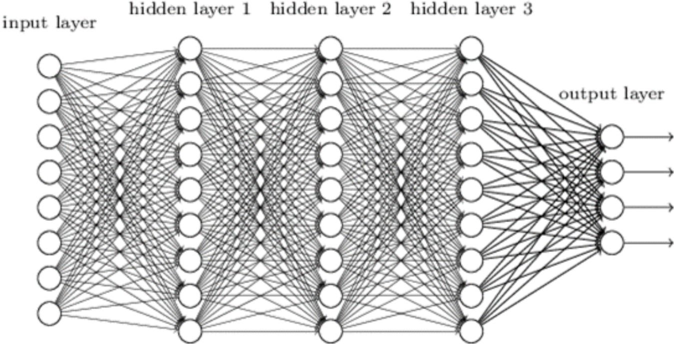
Sometimes they include a node for the intercept in the picture.



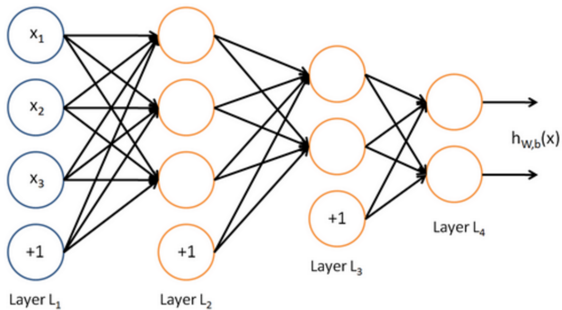
$$
\begin{aligned}
z_1^{(2)} &= b_{10}^{(1)} + b_{11}^{(1)} x_1 + b_{12}^{(1)} x_2 & \longrightarrow \quad & a_1^{(2)} = g(z_1^{(2)}) \\
z_2^{(2)} &= b_{20}^{(1)} + b_{21}^{(1)} x_1 + b_{22}^{(1)} x_2 & \longrightarrow \quad & a_2^{(2)} = g(z_2^{(2)}) \\
z_1^{(3)} &= b_{10}^{(2)} a_0^{(2)} + b_{11}^{(2)} a_1^{(2)} + b_{12}^{(2)} a_2^{(2)} & \longrightarrow \quad & a_1^{(3)} = g(z_1^{(3)})
\end{aligned}
$$

g is the *activation function*.

Deep Neural Network:



A deep neural network is a neural network with more than one hidden layer.



http://ufldl.stanford.edu/wiki/index.php/Neural_Networks

Output Layer:

The first layer is the x values (plus the intercept).

Then there are the "hidden layers" each having a fixed number of units or "neurons".

If we are predicting a single numeric response, then the output layer just has one unit.

If we a predicting a categorical response, then the output layer has C responses where C is the number of categories and these are then softmaxed to give the probabilities.

Note that in fitting deep neural nets it seems to be C outputs softmaxed instead of $C - 1$.

Clearly, there are some interesting issues involved in fitting deep neural nets.

We will use the `R` package H2O to fit deep NNs.

The functions for fitting dNN's have a lot of arguments!!

Let's briefly sketch some of the major issues involved in in fitting dNN's.

Then we will have some understanding of the basic parameters of the fitting function in H2O.

Activation Functions:

Up till now we have used the sigmoid (or logistic) “activation function”

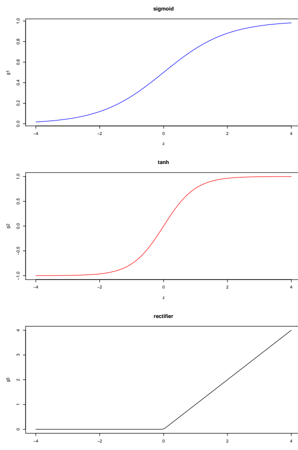
$$g(z) = \frac{1}{(1 + e^{-z})}$$

Other commonly used activation functions are tanh (hyperbolic tangent):

$$g(z) = \frac{e^{z} - e^{-z}}{(e^{z} + e^{-z})}$$

and the rectifier:

$$g(z) = z \text{ for } z > 0, \text{ and 0 else.}$$



Intuitively, it does not seem like there should be much of a difference between sigmoid and tanh, but it turns out tanh works better for gradient computations and seems to be favoured in the deep world.

Stochastic Gradient Descent and Epochs

How do we estimate the parameters ???!!

In NN world the intercepts of the linear combinations are called the "biases" and the slopes are called the weights.

Suppose we have 2 numeric inputs, two hidden layers with 100 units each and 1 numeric output.

Then we have

(2*100) + (100)*(100)+100*1 = 10,300

weights to estimate!!

Gradient Descent

As usual we have training data and a loss function $L(x, y, \theta)$ where θ denotes all the weights and biases.

For example with a numeric outcome we have

$$L(x, y, \theta) = (y - \hat{y}(x, \theta))^2$$

We seek to minimize:

$$\sum_{i=1}^{n} L(x_i, y_i, \theta).$$

where θ is all the biases and weights.

Computing the Hessian matrix is not practical, so the methods are based on the gradient.

Gradient descent just uses the update

$$\theta \to \theta - \epsilon \, \frac{1}{n} \sum_{i=1}^{n} \nabla L(x_i, y_i, \theta).$$

where the gradient is with respect to the elements of θ (all the biases and weights) and ϵ is called the “learning rate”.

Stochastic Gradient Descent

If n is big, each update will take a long time to compute.

Stochastic gradient descent computes the gradient using subsets of the data called *minibatches*.

At iteration k of the algorithm we select a set of minibatch subsets of data $\{x_i^b, y_i^b\}, i = 1, 2, \ldots, m$.

Then we cycle through the minibatches using the update (at each minibatch):

$$\theta \to \theta - \epsilon_k \, \tfrac{1}{m} \sum_{i=1}^{m} \nabla L(x_i^b, y_i^b, \theta).$$

A common practice is to just use a minibatch size of $m = 1$.

In this case we cycle through the observations using the update:

$$\theta \to \theta - \epsilon_k \nabla L(x_i, y_i, \theta).$$

That is, rather than going through the whole data set to compute one gradient and corresponding update, we compute the gradient for each observation and do the update based on that gradient.

We let the ϵ_k decrease over iterations down to some minimum value.

An *epoch* is one pass through the entire data set.

Regularization

We can choose L1 and L2 penalties to regularize the parameter estimation.

Note:

How do we compute the gradient?

It is just the chain rule.

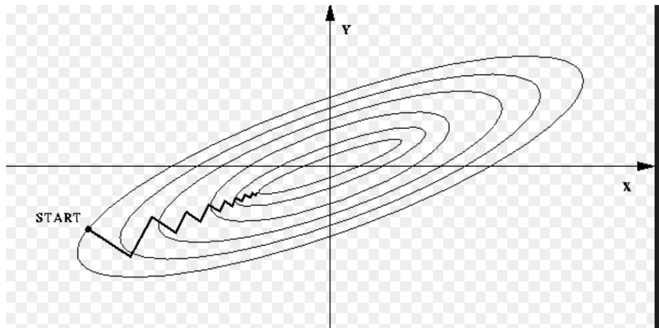
However, a lot of work has gone into organizing the the computations so they can be done efficiently and the method for computing the gradient is called *back propogation*.

To evaluate the model, you move "forward" throught the layers from inputs to output layer.

To evaluate the gradient you move backward from the output layer.

Here is a (stolen) picture showing basic gradient descent.

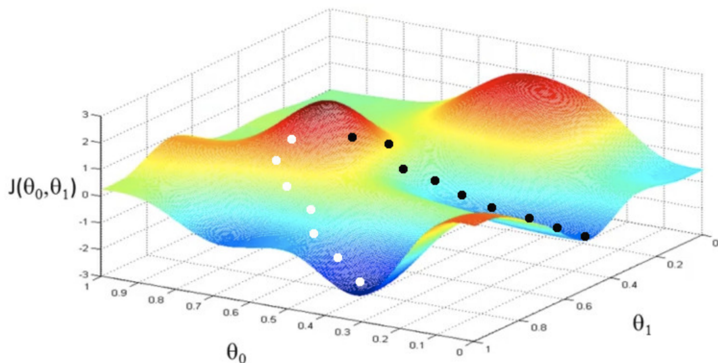
We always move downhill, perpendicular to the contours.



Note, *stochastic* gradient descent will tend to move downhill but not with the full gradient information at each move.

https://www.internalpointers.com/post/gradient-descent-function

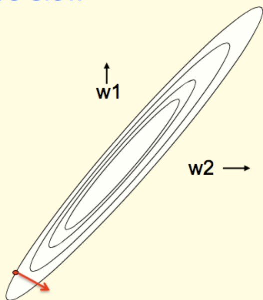
This picture illustrates going to different local minimums depending on the starting value.



This picture gives the basic idea of how gradient descent could be much worse than Newton's method.

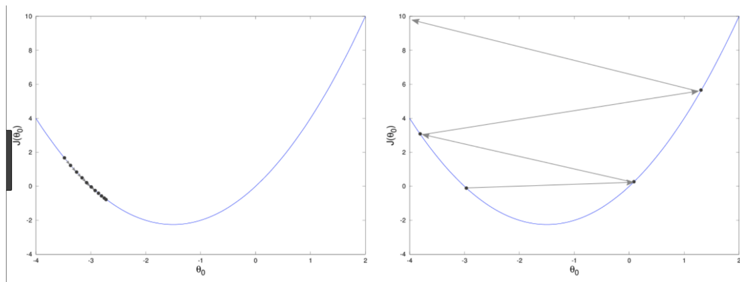
Why learning can be slow

- If the ellipse is very elongated, the direction of steepest descent is almost perpendicular to the direction towards the minimum!
  - The red gradient vector has a large component along the short axis of the ellipse and a small component along the long axis of the ellipse.
  - This is just the opposite of what we want.



This picture shows "gradient" descent in 1-d and illustrates the role of the learning rate.

$$x \to x - \epsilon_k f'(x)$$

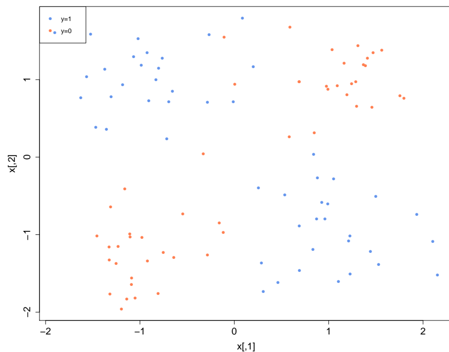


At left we have a small fixed ϵ_k.

At right we have a big fixed ϵ_k.

2. XOR

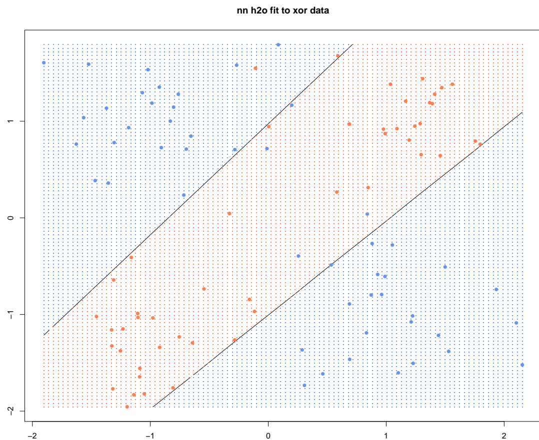
Let's try H2O on the XOR example.



A nn with one hidden layer having 2 neurons.
tanh activation.

```
#  1 hidden 2 neurons
model2 = h2o.deeplearning(x=1:2, y=3,
                          training_frame = dfh2o,
                          hidden = c(2),
                          activation = "Tanh",
                          epochs = 100000,
                          export_weights_and_biases = TRUE,
                          model_id = "xor.model2"
                          #use this if you want to get the same results
                          #seed=99,reproducible=TRUE
                          )

#phat on test
phat = h2o.predict(model2, dftest)
```

Here is a picture of the fit, looks good.



Note: *if I run it again I could get a solution with very little fit !!!!!!!!!!!!*

Let's look at the estimates:

```
> h2o.biases(model2, vector_id = 1)
         C1
1 -11.971459
2   6.656402

> h2o.weights(model2, matrix_id = 1)
         x1         x2
1 13.546680 -14.352386
2  7.834642  -7.051126
```

These are the coefficients going from the input x to the 2 neurons in the hidden layer.

These are the coefficents going from the hidden layer to the to the output.

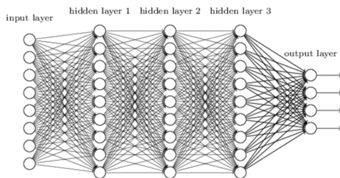
```
> h2o.biases(model2, vector_id = 2)
         C1
1 -3.094724
2  2.822581

[2 rows x 1 column]
> h2o.weights(model2, matrix_id = 2)
         C1        C2
1 -2.588578 -2.885383
2  3.261906 -5.987208
```

So, $O_1 = -3.094724 - 2.588578 g(z_1) - 2.885383 g(z_2)$.

Recall, we have a binary output so we get two outputs which are then softmaxed to get a probability vector.

Deep Features

Remember, in Machine Learning world the x's are called *the features*.



The last layer has the form

$$O_j = \beta_0 + \beta_{j1}\tilde{x}_1 + \beta_{j2}\,\tilde{x}_2 + \ldots + \beta_{jm}\tilde{x}_m$$

where m is the number of units in the final layer.

The $\tilde{x}_i$ are called the *deep features*.
They are nonlinear tranformations of the original x such that y is linearly predicted from them.

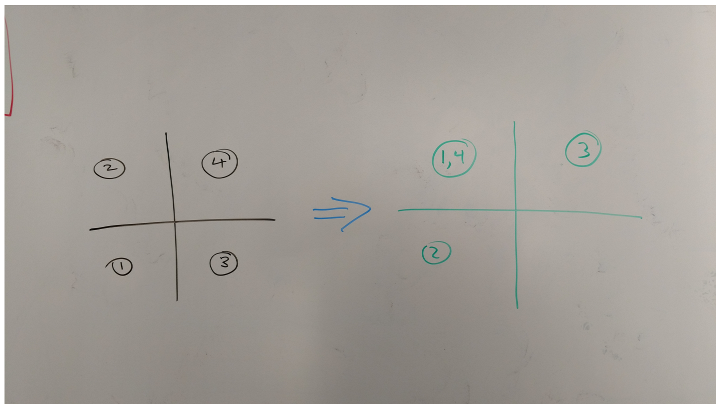
```
> tmp.df = as.h2o(data.frame(x1=c(-1, -1, 1, 1), x2=c(-1, 1, -1, 1)),
+                 destination_frame = "xor.4points")
  |======================================================================| 100%
> trans.features = h2o.deepfeatures(model2, tmp.df, layer = 1)
  |======================================================================| 100%
> as.matrix( h2o.cbind(tmp.df, trans.features) )
     x1 x2 DF.L1.C1   DF.L1.C2
[1,] -1 -1       -1  0.9999284
[2,] -1  1       -1 -0.9999988
[3,]  1 -1        1  1.0000000
[4,]  1  1       -1  0.9999980
```

```
> tanh(-11.971459 + 13.54668 -14.352386)
[1] -1
```

I evaluated the output of the first unit of the hidden layer inputing $(x_1, x_2) = (1, 1)$.

"trans.features" ~ *transformed features*.

With the original features I can't linearly separate the 1's from the 0s'.



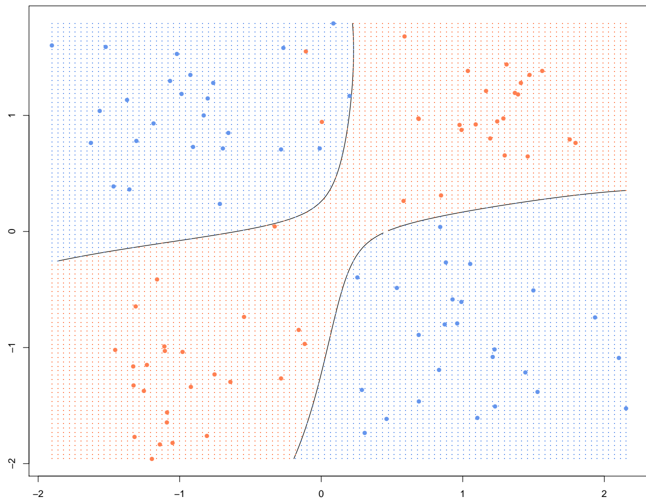
With the transformed features, I can.

Awesome.

This kind of example does make neural nets look magical.

Let's try one hidden layer with 10 units and L1 regularization:

```
model10R = h2o.deeplearning(x=1:2, y=3,
                            dfh2o,
                            hidden = c(10),
                            activation = "Tanh",
                            epochs = 100000,
                            export_weights_and_biases = TRUE,
                            l1 = 1e-2,
                            model_id = "xor.model10R"
)
```


nn h2o fit to xor data
1
0
−1
−2
−2
−1
0
1
2

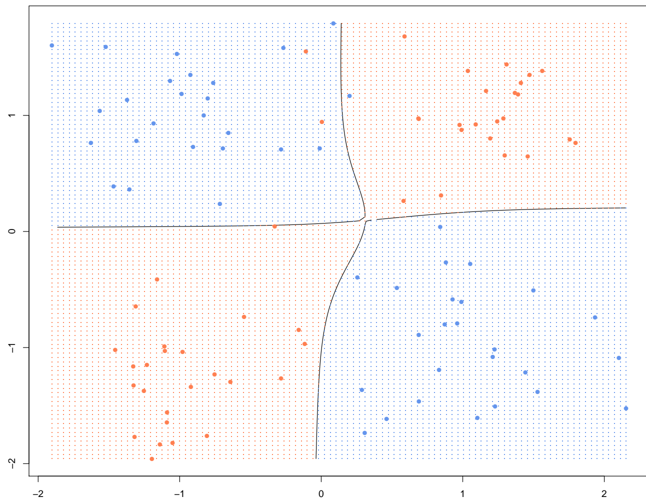
```
> h2o.weights(model10R, matrix_id = 1)
             x1            x2
1  0.0003904525 -0.0004619349
2  0.8172686100  0.9237694740
3 -0.0004394429  0.0000323276
4 -0.0005785795 -0.0005293362
5 -0.0004584224  0.0002672540
6  0.0004523931  0.0003855705

[10 rows x 2 columns]
> h2o.weights(model10R, matrix_id = 2)
            C1        C2            C3            C4            C5
1 0.0003701166 -1.297764 -0.0005743245 -2.533599e-04 -4.597708e-07
2 0.0001853947  1.382249  0.0002720330 -2.158213e-05 -7.892725e-05
             C6         C7         C8         C9           C10
1  0.0003624223  2.1420848  0.1726678 -2.6662343 -0.0001776762
2 -0.0005718899 -0.9607086 -2.9622021  0.5610269 -0.0002848183

[2 rows x 10 columns]
```

Ok, enough fooling around, let's get deep.

```
modelDR = h2o.deeplearning(x=1:2, y=3,
                          dfh2o,
                          hidden = c(3,4),
                          activation = "Tanh",
                          epochs = 100000,
                          export_weights_and_biases = TRUE,
                          l1 = 1e-2,
                          model_id = "xor.modelDR"
)
```

Two hidden layers.
First layer has 3 units, second layer has 4 units.

L1 regularization.
tanh activation.


nn h2o fit to xor data
−2
−1
0
1
2
−2
−1
0
1

$2 \to 3$

$3 \to 4$

$4 \to 2$

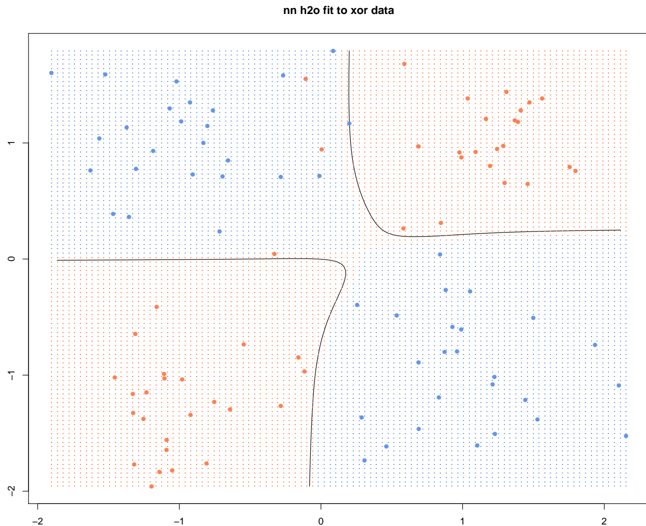
```
> h2o.weights(modelDR, matrix_id = 1)
             x1            x2
1  1.5895231962 -0.0305938125
2 -0.0004432469 -0.0001800873
3  0.0122436108 -2.0536758900

[3 rows x 2 columns]
> h2o.weights(modelDR, matrix_id = 2)
            C1            C2            C3
1 0.0100201899 -4.842104e-04  1.3224930763
2 1.7325805426  4.302524e-04  1.4567693472
3 0.0002096088 -5.461264e-05  0.0001562708
4 1.3153511286 -6.162645e-04 -1.0473971367

[4 rows x 3 columns]
> h2o.weights(modelDR, matrix_id = 3)
          C1        C2            C3         C4
1  0.9344926 -1.814222  0.0004014346  0.2660763
2 -3.0249763  2.523128 -0.0001543377 -3.8591626

[2 rows x 4 columns]
```

```
> h2o.performance(modelDR)
H2OBinomialMetrics: deeplearning
** Reported on training data. **
** Metrics reported on full training frame **

MSE:  0.02630829
RMSE:  0.1621983
LogLoss:  0.1133708
Mean Per-Class Error:  0.03
AUC:  0.9964
Gini:  0.9928

Confusion Matrix for F1-optimal threshold:
        0  1    Error    Rate
0      47  3 0.060000   =3/50
1       0 50 0.000000   =0/50
Totals 47 53 0.030000  =3/100

Maximum Metrics: Maximum metrics at their respective thresholds
                        metric threshold    value idx
1                       max f1  0.394910 0.970874  52
2                       max f2  0.394910 0.988142  52
3                 max f0point5  0.742596 0.975610  48
4                 max accuracy  0.742596 0.970000  48
5                max precision  0.981877 1.000000   0
6                   max recall  0.394910 1.000000  52
7              max specificity  0.981877 1.000000   0
8             max absolute_mcc  0.394910 0.941697  52
```

Let's try the "max f1" threshold.

```
contour(px1, px2, phatg, levels=0.3949, labels="", xlab="", ylab="",
        main= "nn h2o fit to xor data")
points(x, col=ifelse(g==1, "cornflowerblue","coral"),pch=16)
points(gd, pch=".", cex=1.5, col=ifelse(phatv>0.5, "cornflowerblue","coral"))
```

Get's 3 of the 0's (red) wrong.



```
> ## deep features
> tmp.df = as.h2o(data.frame(x1=c(-1, -1, 1, 1), x2=c(-1, 1, -1, 1)),
+                 destination_frame = "xor.4points")
  |======================================================================| 100%
> trans.features = h2o.deepfeatures(modelDR, tmp.df, layer = 2)
  |======================================================================| 100%
> as.matrix( h2o.cbind(tmp.df, trans.features) )
     x1 x2   DF.L2.C1   DF.L2.C2     DF.L2.C3   DF.L2.C4
[1,] -1 -1  0.8399254 -0.9172577 3.285519e-04 -0.9980520
[2,] -1  1 -0.8563508 -0.9996605 3.132807e-05 -0.9047109
[3,]  1 -1  0.8458606  0.9073782 7.016522e-04 -0.8108996
[4,]  1  1 -0.8505694 -0.8524170 4.033999e-04  0.6813285
```

An input (x_1, x_2) is mapped *nonlinearly* to a new x vector with 4 components.

The Deep NN has created nonlinear "deep features" that can be used to predict y more powerfully than the orginal x features.

Better than throwing in x^2 ?? !!

3. Tabloid

Let's try the tabloid with h2o and deep NN.

We already know we can get reasonable results from a single layer nn.

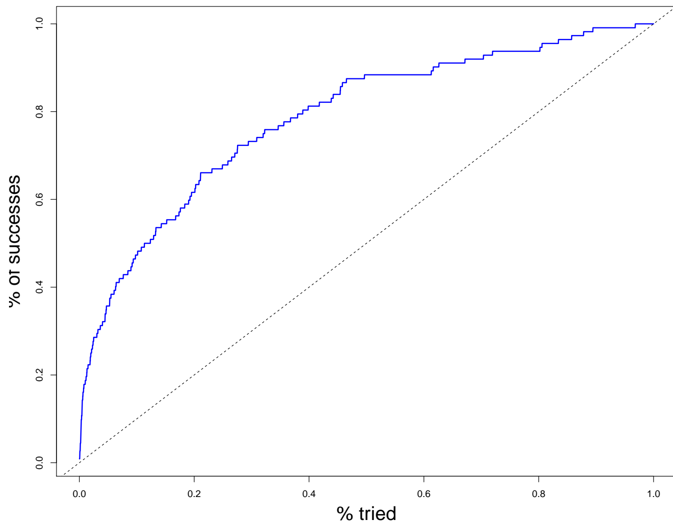
So, this is not an example to highlight the power of nn.
It is just a sanity check to see if we can get reasonable results for a problem we have worked before.

```
## code/libraries
source("lift.R")
library(h2o)

## data: tabloid separated into train and test
trainDf = read.csv("Tabloid_train.csv")
testDf = read.csv("Tabloid_test.csv")

print(names(trainDf))

p=ncol(trainDf)-1
par(mfrow=c(p,2))
for(i in 1:p) {
   plot(trainDf[[i]])
   plot(log(trainDf[[i]]+1))
}

## standardize x , don't need this, h2o will standardize by default
for(i in 1:p) {
   m = mean(trainDf[,i+1]); s = sd(trainDf[,i+1])
   print(c(m,s))
   trainDf[[i+1]] = (trainDf[[i+1]]-m)/s
   testDf[[i+1]] = (testDf[[i+1]]-m)/s
}
```

Let's fit a logit for comparison.

```
##make y=purchase a factor and call it y
trainDf$purchase = as.factor(trainDf$purchase)
testDf$purchase = as.factor(testDf$purchase)
names(trainDf)[1]="y"
names(testDf)[1]="y"

### setup storage for results
phatL = list() #store the test phat for the different methods here

### fit logit
lgfit = glm(y~.,trainDf,family=binomial)
print(summary(lgfit))
phat = predict(lgfit,testDf,type="response")
phatL$logit = matrix(phat,ncol=1) #logit phat

## how is logit
temp = lift.plot(phatL$logit,testDf$y)
```

Lift for logit. Tough to beat.



To use h2o we have to "initialize the server" and put our data into a form h2o can work with.

```
## get setup to run nn in h2o
h2oServer <- h2o.init(ip="localhost", port=54321,
                         max_mem_size="4g", nthreads=-1)
train_h2o = as.h2o(trainDf, destination_frame = "tabloid_train")
test_h2o = as.h2o(testDf, destination_frame = "tabloid_test")
```

Let's fit a nn with one hidden layer having 10 units.

```
if(file.exists(file.path("./", "model1"))) {
  model1 = h2o.loadModel(path = file.path("./", "model1"))
} else {
  model1 = h2o.deeplearning(
          x=2:5, y=1,
          training_frame=train_h2o,
          hidden=10,
          epochs=1000,
          export_weights_and_biases=T,
          l1 = 1e-2,
          model_id = "model1"
          )
  h2o.saveModel(model1, path="./")
}
```

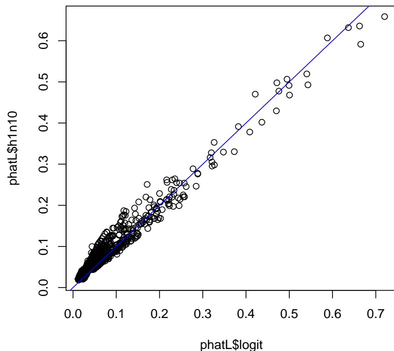
You need to set model_id to controle the file name used by saveModel.
It won't use the "R name" model1 automatically.

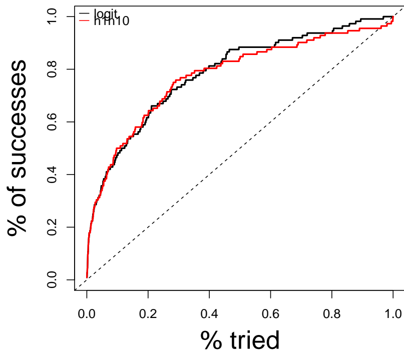
Get $\hat{p}$ on test and compare with logit.

```
phat = predict(model1, test_h2o)
phatL$h1n10 = as.matrix( phat[,3] )

#plot, compare to logit
par(mfrow=c(1,2))
plot(phatL$logit, phatL$h1n10)
abline(0,1,col="blue")
lift.many.plot(phatL, testDf$y)
legend("topleft",legend=names(phatL),col=1:2,lty=rep(1,2),bty="n")
```

Get $\hat{p}$ on test and compare with logit.
Very similar results from logit and single layer nn from h2o.

Good.

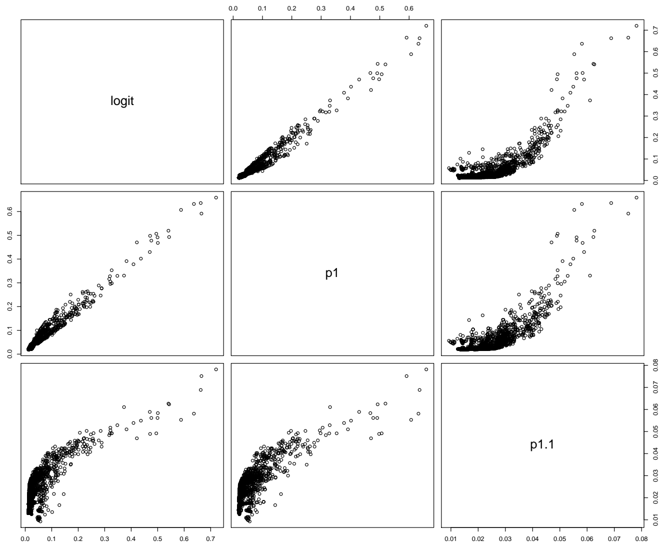


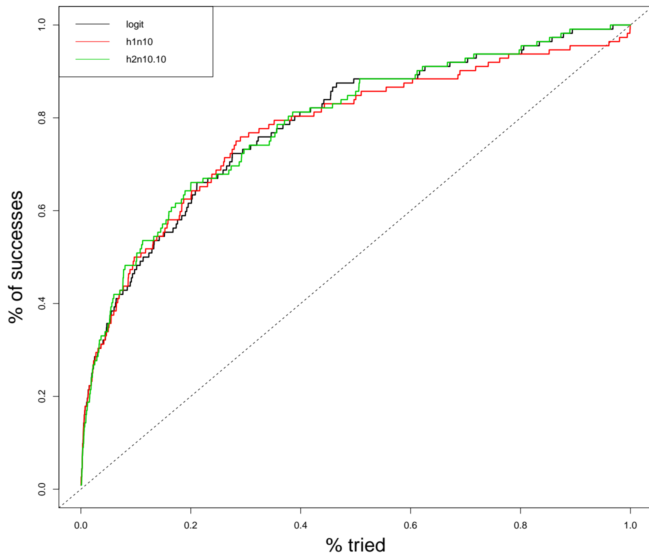


Ok, let's try a deep nn.

```
### fit h2o deep
if (file.exists(file.path("./", "deep.model"))) {
  deep.model = h2o.loadModel(path = file.path("./", "deep.model"))
} else {
  deep.model = h2o.deeplearning(
                x=2:5, y=1,
                training_frame=train_h2o,
                hidden=c(10,10),
                epochs=500,
                activation="RectifierWithDropout",
                l1=1e-3,
                export_weights_and_biases=TRUE,
                model_id = "deep.model"
                )
  h2o.saveModel(deep.model, path="./")
}

phat = predict(deep.model, test_h2o)
phatL$h2n10.10 = as.matrix( phat[,3] )

pairs(phatL)
```

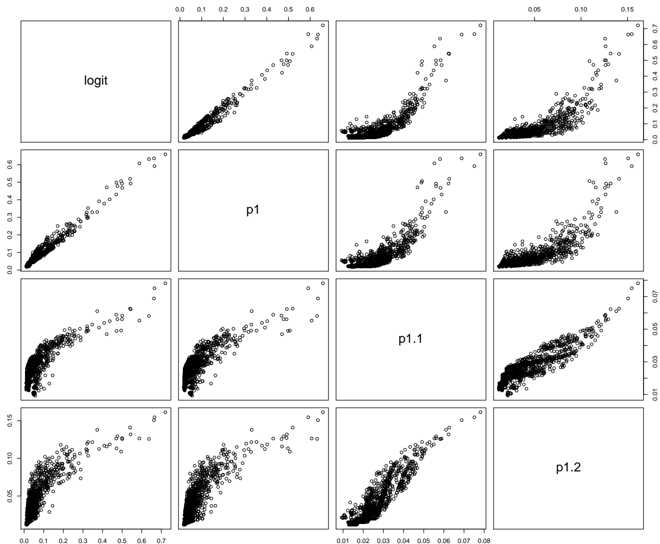
Ok, which is better ??

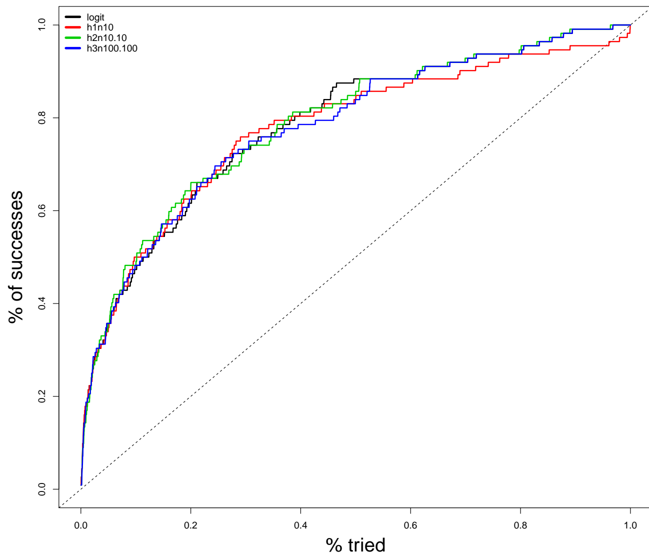



logit
h1n10
h2n10.10
% of successes
0.0
0.2
0.4
0.6
0.8
1.0
% tried
0.0
0.2
0.4
0.6
0.8
1.0

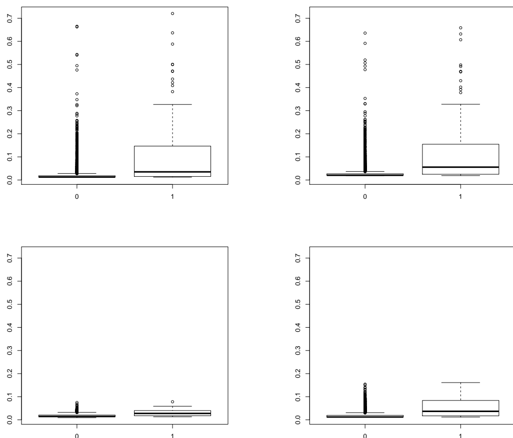
Deeper.

```
if (file.exists(file.path("./", "rdeep.model"))) {
  rdeep.model = h2o.loadModel(path = file.path("./", "rdeep.model"))
} else {
  rdeep.model = h2o.deeplearning(
                  x=2:5, y=1,
                  training_frame=train_h2o,
                  hidden=c(100,100),
                  epochs=500,
                  activation="RectifierWithDropout",
                  l1=1e-3,
                  export_weights_and_biases=TRUE,
                  model_id = "rdeep.model"
                  )
  h2o.saveModel(rdeep.model, path="./")
}

phat = predict(rdeep.model, test_h2o)
phatL$h3n100.100 = as.matrix( phat[,3] )
```


logit
p1
p1.1
p1.2


logit
h1n10
h2n10.10
h3n100.100
% of successes
0.0
0.2
0.4
0.6
0.8
1.0
% tried
0.0
0.2
0.4
0.6
0.8
1.0

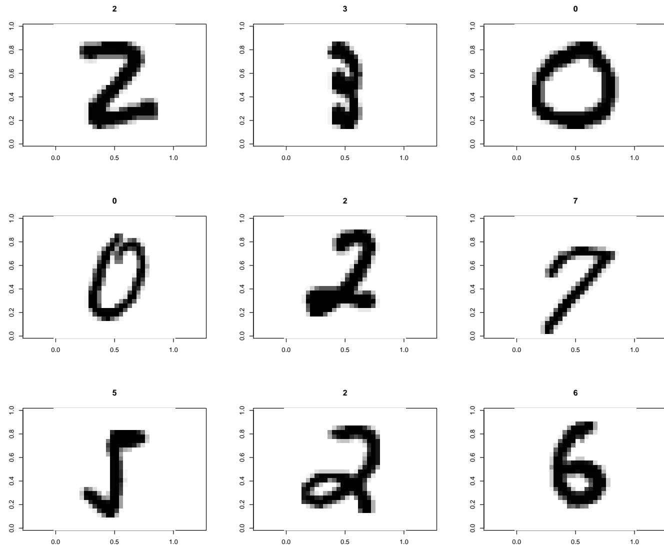
Even though they all give similar lift curves, the fitted probabilities are quite different.



We should find out which one gives the most profit!!!

4. MNIST Digit Recognition

Handwritten digits captured as 0-255 grayscale values on a 28×28 grid.



Digit recognition:

Guess the digit, given the $28^2 = 784$ values:

$$P(y=2 \mid [\text{image of handwritten 2}], b)$$

$$P(y=9 \mid [\text{image of handwritten 9}], b)$$

where "b" is model parameters (e.g. weights).

Easy for a person, hard for a machine !!

Note:

Our black and white images are values in [0,255] on a 2 dimensional grid of pixels.

Color images are (r,g,b) values on a grid of pixels.

(r,g,b): red, green, blue.

For example: the input might be $32 \times 32 \times 3$.

We have 60,000 train observations and 10,000 test.

For each observation $y \in \{0, 1, 2, \ldots, 9\}$.

For each observation, x is $28^2 = 784$ grayscale values in $[0, 255]$.

y counts:

```
   C785 nrow_C785
1     0      5923
2     1      6742
3     2      5958
4     3      6131
5     4      5842
6     5      5421
7     6      5918
8     7      6265
9     8      5851
10    9      5949
```

60,000 train in train60, 10,000 test in test.

```
> range(train[,1:784])
[1]   0 255
> dim(train60)
[1] 60000   785
> dim(test)
[1] 10000   785
```

Split train60 into train and valid:

```
set.seed(99)
parts = h2o.splitFrame(train60,1.0/6.0)
valid = parts[[1]]
train = parts[[2]]
rm(parts)
```

First, we will try the “default” random forests fit using h2o.

```
fp = file.path("./files","mRFdef")
if(file.exists(fp)) {
   mRFdef = h2o.loadModel(fp)
} else {
   mRFdef = h2o.randomForest(x,y,train,
               model_id="mRFdef",
              validation_frame=valid)
   h2o.saveModel(mRFdef,path="./files")
}
cat("is model S4:",isS4(mRFdef),"\n")
cat("model id: ",mRFdef@model_id,"\n")
convRFdef = h2o.confusionMatrix(mRFdef,valid=TRUE)
printfl(convRFdef,dpl,"defRF-conf.rtxt")
```

Here is the confusion matrix:

Confusion Matrix: vertical: actual; across: predicted

	0	1	2	3	4	5	6	7	8	9	Error		Rate
0	935	0	0	0	0	0	3	0	10	0	0.0137	=	13 / 948
1	0	1105	5	4	2	0	1	1	0	0	0.0116	=	13 / 1,118
2	8	3	910	5	4	1	0	9	3	0	0.0350	=	33 / 943
3	4	4	10	983	2	7	2	17	12	3	0.0584	=	61 / 1,044
4	1	3	5	1	924	0	4	2	1	25	0.0435	=	42 / 966
5	11	0	1	6	1	904	14	0	5	3	0.0434	=	41 / 945
6	4	4	0	0	1	7	956	0	7	0	0.0235	=	23 / 979
7	1	4	12	2	4	0	0	1046	0	7	0.0279	=	30 / 1,076
8	4	7	6	8	6	7	7	2	974	13	0.0580	=	60 / 1,034
9	6	1	4	13	13	6	1	12	8	921	0.0650	=	64 / 985
Totals	974	1131	953	1022	957	932	988	1089	1020	972	0.0379	=	380 / 10,038

Really, .038 is amazing.

Now let's try default nn (hidden=c(200,200)) and look at grabbing off some performance metrics on the validation data.

```
fp = file.path("./files","mDNNdef")
if(file.exists(fp)) {
   mDNNdef = h2o.loadModel(fp)
} else {
   mDNNdef = h2o.deeplearning(x,y,train,
             model_id="mDNNdef",
            validation_frame=valid)
   h2o.saveModel(mDNNdef,path="./files")
}
cat("model id: ",mDNNdef@model_id,"\n")
convDNNdef = h2o.confusionMatrix(mDNNdef,valid=TRUE)
printfl(convDNNdef,dpl,"defDNN-conf.rtxt")
perfDNNdef = h2o.performance(mDNNdef,valid=TRUE)
print(perfDNNdef@metrics$hit_ratio_table$hit_ratio)
print(perfDNNdef@metrics$mean_per_class_error)
```

Confusion Matrix: vertical: actual; across: predicted

	0	1	2	3	4	5	6	7	8	9	Error		Rate
0	925	0	2	1	1	1	9	0	8	1	0.0243	=	23 / 948
1	0	1090	11	7	1	0	0	7	2	0	0.0250	=	28 / 1,118
2	2	0	909	16	7	0	1	4	3	1	0.0361	=	34 / 943
3	1	1	8	1015	1	4	1	4	8	1	0.0278	=	29 / 1,044
4	3	0	7	0	937	2	4	2	1	10	0.0300	=	29 / 966
5	10	0	1	32	2	878	7	0	6	9	0.0709	=	67 / 945
6	5	2	2	0	5	3	958	1	3	0	0.0215	=	21 / 979
7	2	2	9	2	5	0	0	1049	1	6	0.0251	=	27 / 1,076
8	2	10	6	9	5	6	4	2	984	6	0.0484	=	50 / 1,034
9	4	1	2	5	16	3	0	10	9	935	0.0508	=	50 / 985
Totals	954	1106	957	1087	980	897	984	1079	1025	969	0.0357	=	358 / 10,038

About the same as random forests.

Now let's try a bunch of random forests to see if we can do better than the default.

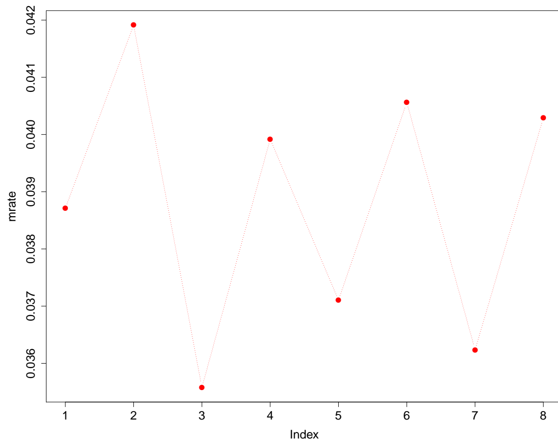
h2o has a "grid" function which supports trying several parameter values.

```
listModels = list()
modelNames = list.files(file.path("./files/"),pattern="Grid_DRF_*")
if(length(modelNames)!=0) {
   numModels = 0
   for (modelName in modelNames) {
      numModels = numModels + 1
      listModels[[numModels]] =
              h2o.loadModel(path = file.path("./files/", modelName))
   }
} else {
   gRF = h2o.grid("randomForest",
      hyper_params=list(
           ntrees=c(100,500),
           mtries=c(28,50),
           min_rows=c(2,5)),
     x=x,y=y,training_frame=train,validation_frame=valid)

   listModels = lapply(gRF@model_ids, function(id) h2o.getModel(id))
   for(m in listModels) h2o.saveModel(m,path="./files")
}
```

Get the missclassification rate for each fitted model, plot the results, and pull off the best one.

```
numModels=length(listModels)
mrate = rep(0,numModels)
for(i in 1:numModels) {
   print(h2o.confusionMatrix(listModels[[i]],valid=TRUE))
   mrate[i] =
     h2o.performance(listModels[[i]],valid=TRUE)@metrics$mean_per_class_error
}

if(dpl) pdf(file="mrate-rftuned.pdf",height=10,width=12)
plot(mrate,pch=16,col="red",cex.axis=1.5,cex.lab=1.5)
if(dpl) dev.off()

bestRF = listModels[[which.min(mrate)]]
cat("bestRF has:\n")
cat("ntrees,mtries,min_rows: ", bestRF@parameters$ntrees,
                              bestRF@parameters$mtries,
                              bestRF@parameters$min_rows,"\n")
```

Missclassification rates over the 8 settings.



The best setting was:

```
ntrees,mtries,min_rows:  100 50 2

where we tried:

ntrees=c(100,500),
           mtries=c(28,50),
           min_rows=c(2,5))
```

5. More on Fitting Neural Nets

Gradient descent + chain rule + lot of tricks

- We will not provide details
- The procedure is called backpropagation

Difficult to train because there are many local minima

- Train multiple nets with different inital weights
- Initialize weights near zero
- Therefore, initial networks near-linear
- Increasingly non-linear functions possible as training progresses

Adaptive Learning Rate

- Automatically set learning rate for each neuron based on its training history
- ADADELTA: http://www.matthewzeiler.com/pubs/googleTR2012/googleTR2012.pdf

Momentum

- $b^{t+1} = b^t - \eta \cdot \nabla J(b) + \alpha(b^t - b^{t-1})$
- α is the momentum parameter
- helps avoiding stuck in a local optimum

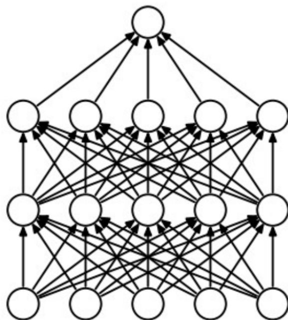
Regularization

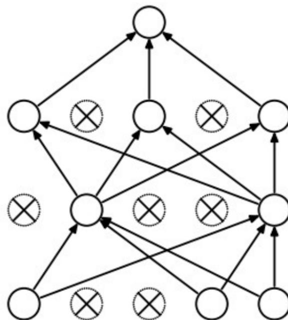
- L1 penalty on the parameters
- L2 penalty on the parameters (weight decay parameter)
- Early stopping

Dropout

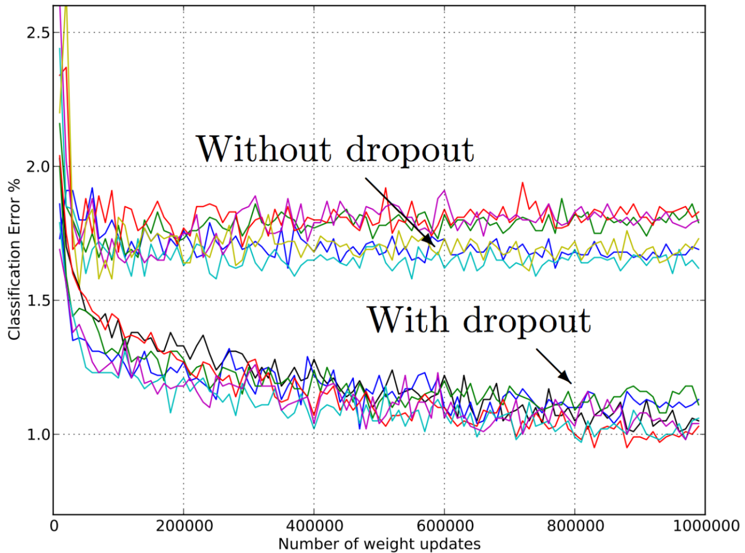
Eliminate some of the connections.

Dropout:


(a) Standard Neural Net


(b) After applying dropout.

https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf


Without dropout
With dropout
Classification Error %
2.5
2.0
1.5
1.0
0
200000
400000
600000
800000
1000000
Number of weight updates

Fitting neural networks: Tips from h2o

- more layers for more complex functions (more nonlinearity).
- more neurons per layer to fit finer structure in data.
- add regularization (max_w2=50 or L1 = 1e-5).
- do a grid search to get a feel for parameters.
- try "Tanh", the "Rectifier".
- try dropout (input 20%, hidden 50%).

Note: max_w2:
An upper limit for the (squared) sum of the incoming weights to a neuron.
h2o default is to have no limit.

Mladen says also see:

http://yyue.blogspot.com/2015/01/a-brief-overview-of-deep-learning.html

Has tips and some general comments of on deep neural nets which capture the spirit.

6. DNN Grid Search for MNIST

Let's try a grid search to see what works with deeplearning.

Loosely following the advice from h2o and the Cook book, but not wanting to run for too long, I tried the following $2^5 = 32$ settings.

We'll try $2^5 = 32$ different deep neural net settings in our grid search.
Several hours on my portable workstation laptop.

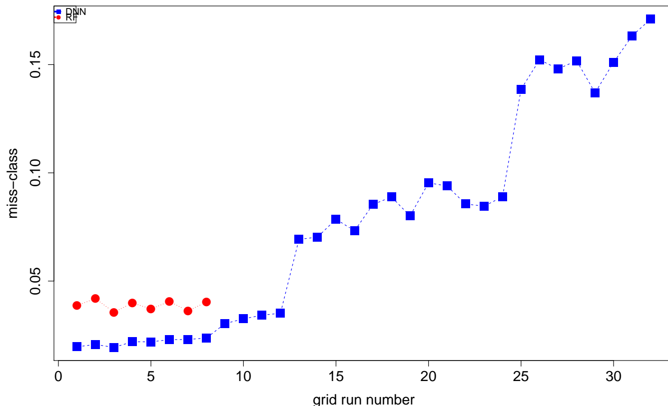
```
> hyper_params
$hidden
$hidden[[1]]
[1] 200 200

$hidden[[2]]
[1] 300 300


$activation
[1] "TanhWithDropout"      "RectifierWithDropout"

$hidden_dropout_ratios
$hidden_dropout_ratios[[1]]
[1] 0.1 0.1

$hidden_dropout_ratios[[2]]
[1] 0.5 0.5


$l1
[1] 1e-04 1e-02

$max_w2
[1] 3.402823e+38 5.000000e+01
```

```
> expand.grid(hyper_params)
      hidden            activation hidden_dropout_ratios    l1       max_w2
1  200, 200       TanhWithDropout              0.1, 0.1 1e-04 3.402823e+38
2  300, 300       TanhWithDropout              0.1, 0.1 1e-04 3.402823e+38
3  200, 200 RectifierWithDropout              0.1, 0.1 1e-04 3.402823e+38
4  300, 300 RectifierWithDropout              0.1, 0.1 1e-04 3.402823e+38
5  200, 200       TanhWithDropout              0.5, 0.5 1e-04 3.402823e+38
6  300, 300       TanhWithDropout              0.5, 0.5 1e-04 3.402823e+38
7  200, 200 RectifierWithDropout              0.5, 0.5 1e-04 3.402823e+38
8  300, 300 RectifierWithDropout              0.5, 0.5 1e-04 3.402823e+38
9  200, 200       TanhWithDropout              0.1, 0.1 1e-02 3.402823e+38
10 300, 300       TanhWithDropout              0.1, 0.1 1e-02 3.402823e+38
11 200, 200 RectifierWithDropout              0.1, 0.1 1e-02 3.402823e+38
12 300, 300 RectifierWithDropout              0.1, 0.1 1e-02 3.402823e+38
13 200, 200       TanhWithDropout              0.5, 0.5 1e-02 3.402823e+38
14 300, 300       TanhWithDropout              0.5, 0.5 1e-02 3.402823e+38
15 200, 200 RectifierWithDropout              0.5, 0.5 1e-02 3.402823e+38
16 300, 300 RectifierWithDropout              0.5, 0.5 1e-02 3.402823e+38
17 200, 200       TanhWithDropout              0.1, 0.1 1e-04 5.000000e+01
18 300, 300       TanhWithDropout              0.1, 0.1 1e-04 5.000000e+01
19 200, 200 RectifierWithDropout              0.1, 0.1 1e-04 5.000000e+01
20 300, 300 RectifierWithDropout              0.1, 0.1 1e-04 5.000000e+01
21 200, 200       TanhWithDropout              0.5, 0.5 1e-04 5.000000e+01
22 300, 300       TanhWithDropout              0.5, 0.5 1e-04 5.000000e+01
23 200, 200 RectifierWithDropout              0.5, 0.5 1e-04 5.000000e+01
24 300, 300 RectifierWithDropout              0.5, 0.5 1e-04 5.000000e+01
25 200, 200       TanhWithDropout              0.1, 0.1 1e-02 5.000000e+01
26 300, 300       TanhWithDropout              0.1, 0.1 1e-02 5.000000e+01
27 200, 200 RectifierWithDropout              0.1, 0.1 1e-02 5.000000e+01
28 300, 300 RectifierWithDropout              0.1, 0.1 1e-02 5.000000e+01
29 200, 200       TanhWithDropout              0.5, 0.5 1e-02 5.000000e+01
30 300, 300       TanhWithDropout              0.5, 0.5 1e-02 5.000000e+01
31 200, 200 RectifierWithDropout              0.5, 0.5 1e-02 5.000000e+01
32 300, 300 RectifierWithDropout              0.5, 0.5 1e-02 5.000000e+01
```

Run the 32 settings in h2o:

```
gDNN = h2o.grid("deeplearning",
                    hyper_params=hyper_params,
                    x=x,y=y,training_frame=train,validation_frame=valid,
                    epochs=200)
```

The best nnet beats the best Random Forest.



The first 8 runs are the ones with l1 shrinkage $= 1e - 04$, and $max_w2 = infinity$.

Let's pull off the best nn model.

```
numModels=length(listModels)
mratednn = rep(0,numModels)
for(i in 1:numModels) {
   print(h2o.confusionMatrix(listModels[[i]],valid=TRUE))
   mratednn[i] =
     h2o.performance(listModels[[i]],valid=TRUE)@metrics$mean_per_class_error
}

bestDNN = listModels[[which.min(mratednn)]]
print(h2o.confusionMatrix(bestDNN,valid=TRUE))
```

```
> print(h2o.confusionMatrix(bestDNN,valid=TRUE))
Confusion Matrix: vertical: actual; across: predicted
         0    1   2    3   4   5   6    7    8   9  Error              Rate
0      936    0   2    1   1   0   2    0    6   0 0.0127 =     12 / 948
1        0 1111   1    4   0   0   0    1    1   0 0.0063 =    7 / 1,118
2        1    2 927    6   1   1   0    4    1   0 0.0170 =     16 / 943
3        0    0   8 1025   0   4   0    2    3   2 0.0182 =   19 / 1,044
4        1    2   4    0 935   0   3    1    1  19 0.0321 =     31 / 966
5        5    0   0    2   0 932   4    0    0   2 0.0138 =     13 / 945
6        5    2   1    0   1   6 962    0    2   0 0.0174 =     17 / 979
7        1    1   5    5   0   0   0 1062    1   1 0.0130 =   14 / 1,076
8        0    3   2    9   3   7   3    3 1000   4 0.0329 =   34 / 1,034
9        3    1   1    2   4   9   1    4    5 955 0.0305 =     30 / 985
Totals 952 1122 951 1054 945 959 975 1077 1020 983 0.0192 = 193 / 10,038
```

Error rate down to 2%.

Interesting to see which digits are confused with which digits !!

Let's try fitting our best setting on (train,validation) and predict on test.

But, we may hit a bad local min!!!

```
trainval = h2o.rbind(train,valid)

fp = file.path("./files","mDNNfinal")
if(file.exists(fp)) {
   mDNNfinal = h2o.loadModel(fp)
} else {
   mDNNfinal = h2o.deeplearning(x,y,trainval,
            hidden=c(200,200),
            activation="TanhWithDropout",
            hidden_dropout_ratios=c(.1,.1),
            l1=1e-4,
            epochs=200,
            model_id="mDNNfinal",
           validation_frame=test)

   h2o.saveModel(mDNNfinal,path="./files")
}

print(h2o.confusionMatrix(mDNNfinal,valid=TRUE))
```

```
> print(h2o.confusionMatrix(mDNNfinal,valid=TRUE))
Confusion Matrix: vertical: actual; across: predicted
          0    1    2    3   4   5   6    7   8   9  Error          Rate
0       973    0    1    1   0   1   1    1   2   0 0.0071 =      7 / 980
1         0 1120    1    6   0   1   2    1   4   0 0.0132 =   15 / 1,135
2         3    0 1011    1   4   0   1    8   4   0 0.0203 =   21 / 1,032
3         1    0    4  987   0   7   1    6   3   1 0.0228 =   23 / 1,010
4         2    0    2    1 959   1   6    0   1  10 0.0234 =     23 / 982
5         4    1    0   15   0 858   6    1   4   3 0.0381 =     34 / 892
6        10    2    2    2   1   2 938    0   1   0 0.0209 =     20 / 958
7         1    6   14    6   2   0   0  984   2  13 0.0428 =   44 / 1,028
8         7    1    4    6   5   3   4    5 937   2 0.0380 =     37 / 974
9         6    3    1   10  11   2   1    4   6 965 0.0436 =   44 / 1,009
Totals 1007 1133 1040 1035 982 875 960 1010 964 994 0.0268 = 268 / 10,000
```

Hmm, not as good as on val.
But still not bad.

7. More on Digit Recognition

The digit recognition problem is a famous problem of basic importance in Machine Learning/Statistics.

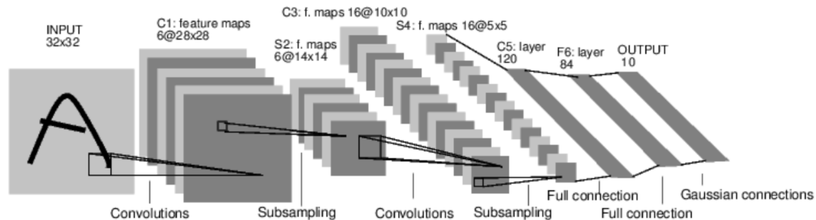
Deep neural nets have been very successful
with some special twists !!!

The pixel layout is a very special structure and some approaches have been developed to take advantage of it.

These approaches coupled with deep learning are the "state of the art".

Let's just get a rough idea of what is involved.

Besides the usual hidden layers we have looked at, different kinds of layers are used to take advantage of the pixel structure:



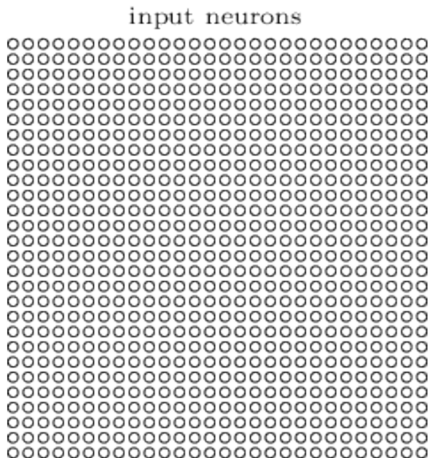
Convolution layers replace a pixel value with the average of nearby pixels.

Pooling layers replace of rectangular set of pixels with the maximum value.

See http://yann.lecun.com/exdb/lenet/

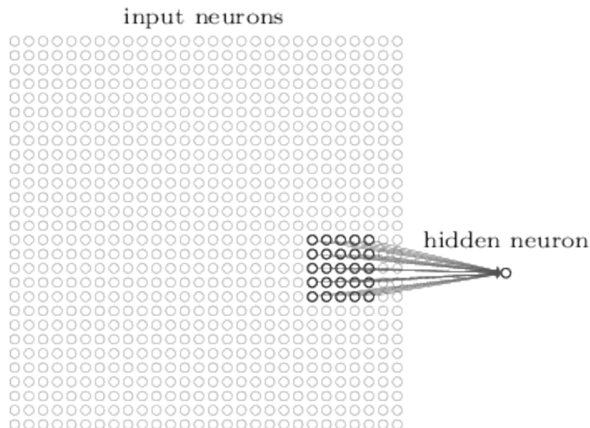
Convolution Layers:

Here is our 28^2 input layer:

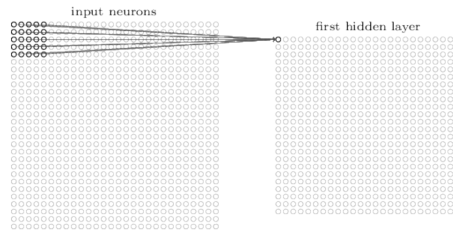


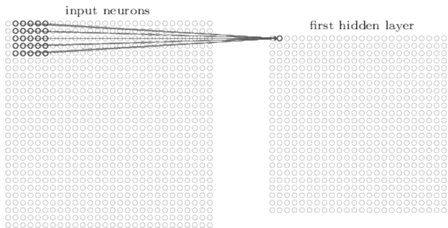
From: http://neuralnetworksanddeeplearning.com/chap6.html

To get single neuron for the next layer, take a weighted average of neurons in a box where the neuron is at the top left corner.
(in images you often make the origin the top left).



You have to pick the weights and number of neighbors.

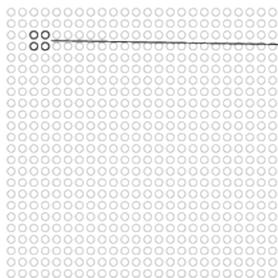


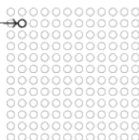


This will give an ouput layer a little smaller or about the same size depending on how you do it.

Pooling Layer:

A pooling layer replaces the pixel values in non-overlapping regions with the maxiumum value.





This will typically reduce the number or neurons in the next layer. The pooling layer “introduces an elmement of local translation invariance” (Efron and Hastie).

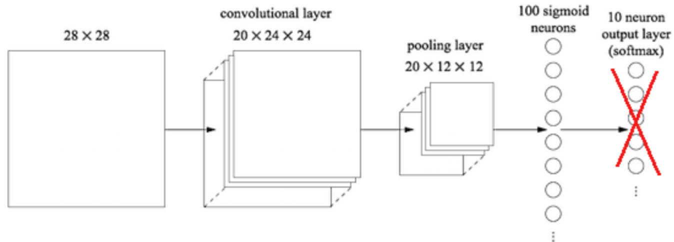
Another cool idea:

Expand the set of examples.

For each (x, y) pair produce a set a pairs
(x_s, y) where x_s is obtained from x by small distortions:
scaling, rotation, . . .

Then add all the generated (x_s, y) to your training data!!!

Another cool idea:



Use the output of the last layer as a representation of your data.

Fit a model with this representation.

Some Success Stores:

- Google voice transcription

  http://googleresearch.blogspot.com/2015/08/the-neural-networks-behind-google-voice.html

- Google voice search

  http://googleresearch.blogspot.com/2015/09/google-voice-search-faster-and-more.html

- Google translate app

  http://googleresearch.blogspot.com/2015/07/how-google-translate-squeezes-deep.html

- Facebook face recognition

  http://www.technologyreview.com/news/525586/facebook-creates-software-that-matches-faces-almost-as-well-as-you-do

- Paypal fraud detection

  http://www.slideshare.net/0xdata/paypal-fraud-detection-with-deep-learning-in-h2o-presentationh2oworld2014

8. Back Propagation

We will need a general notation for the neural net model.

Let's start by letting ℓ index the layers.

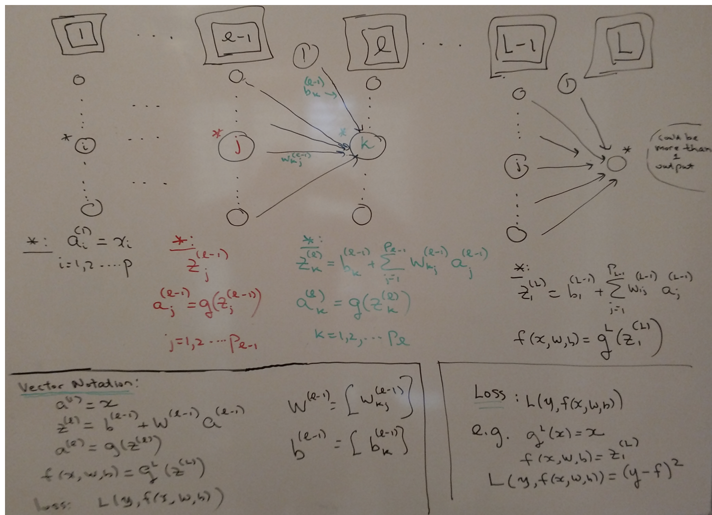
ℓ goes from 1 to L where $\ell = 1$ is the input layer (x) and L is the final output layer.

To keep things simpler, we will have just one outcome with associated activation function g^L. For a single numeric outcome, g^L would typically be the identity function $I(x) = x$.

We will use the some activation function g at all the interior units (neurons).

Let p_ℓ be the number of neurons at layer ℓ.
Note that $p_1 = p$ where p is the dimension of x since that is the input layer.

Here is the general model:



Simplest interesting case.
One x, one hidden layer with 2 neurons, one output.

$w_{ij}^{(\ell)}$: weight from neuron j of layer ℓ to neuron i of layer $\ell+1$

1 2 3

$(p_1, p_2, p_3) = (1, 2, 1)$

$a_1^{(1)} = x$

(*) $z_1^{(2)} = b_1^{(1)} + w_{11}^{(1)} a_1^{(1)}$

$a_1^{(2)} = g(z_1^{(2)})$

(*) $z_2^{(2)} = b_2^{(1)} + w_{21}^{(1)} a_1^{(1)}$

$a_2^{(2)} = g(z_2^{(2)})$

$z_1^{(3)} = b_1^{(2)} + w_{11}^{(2)} a_1^{(2)} + w_{12}^{(2)} a_2^{(2)}$ (++ $g(z_1^{(2)})$)

$f(x,b,w) = g^L(z_1^{(3)})$ $(L=3)$

$L(y,f) = (y-f)^2$

Chain Rule: $L \leftarrow f \leftarrow z_1^{(3)} \leftarrow w_{11}^{(2)}$

Layer 2

$\frac{\partial L}{\partial w_{11}^{(2)}} = -2(y-f)(g^L)'(z_1^{(3)})\, a_1^{(2)} \equiv \delta_1^{(L)} a_1^{(2)}$

$\frac{\partial L}{\partial w_{12}^{(2)}} = -2(y-f)(g^L)'(z_1^{(3)})\, a_2^{(2)} \equiv \delta_1^{(L)} a_2^{(2)}$

$\frac{\partial L}{\partial b_1^{(2)}} = -2(y-f)(g^L)'(z_1^{(3)}) \equiv \delta_1^{(L)} = \delta_1^{(3)}$

$\delta_1^{(3)} = \frac{\partial L}{\partial z_1^{(3)}}$

Layer 1

$\frac{\partial L}{\partial w_{11}^{(1)}} = \frac{\partial L}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial w_{11}^{(1)}} = \delta_1^{(2)} a_1^{(1)}$

$\delta_1^{(2)} = \frac{\partial L}{\partial z_1^{(2)}} = \frac{\partial L}{\partial z_1^{(3)}} \frac{\partial z_1^{(3)}}{\partial z_1^{(2)}} = \delta_1^{(3)} w_{11}^{(2)} g'(z_1^{(2)})$ see ++

$\frac{\partial L}{\partial b_1^{(1)}} = \frac{\partial L}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial b_1^{(1)}} = \delta_1^{(2)}$

$\frac{\partial L}{\partial w_{21}^{(1)}} = \delta_2^{(2)} a_1^{(1)}$

$\delta_2^{(2)} = \delta_1^{(3)} w_{12}^{(2)} g'(z_2^{(2)})$

$\frac{\partial L}{\partial b_2^{(1)}} = \delta_2^{(2)}$

$\delta_i^{(\ell)} = \frac{\partial L}{\partial z_i^{(\ell)}}$

How it Works

Key Quantities:

$\delta_i^{(\ell)}$: effect on loss of a change in $z_i^{(\ell)}$

Iterate

1. initialize by computing $\delta_i^{(L)}$
2. iterate $(\ell+1) \rightarrow (\ell)$ getting $\delta_j^{(\ell)}$ from $\delta_i^{(\ell+1)}$ — "backprop"
3. Get partials for layer ℓ parameters $b^{(\ell)}, W^{(\ell)}$ from $\delta_i^{(\ell+1)}$

Here are the partial derivatives associated with the parameters at layer $L-1$.

This will also initialize the back-progagation algorithm for computing the partials for parameters associated with the other layers.

$$z_1^{(L)} = b_1^{(L-1)} + \sum_{j=1}^{p_{L-1}} w_{1j}^{(L-1)} a_j^{(L-1)}$$

$$f = g^L(z_1^{(L)})$$

$$L = (y-f)^2$$

$$\delta_1^{(L)} \equiv \frac{\partial L}{\partial z_1^{(L)}}$$

$$\frac{\partial L}{\partial w_{1j}^{(L-1)}} = -2(y-f)(g^L)'(z_1^{(L)})\, a_j^{(L-1)} \equiv \delta_1^{(L)} a_j^{(L-1)}$$

$$\frac{\partial L}{\partial b_1^{(L-1)}} = \delta_1^{(L)} = \frac{\partial L}{\partial z_1^{(L)}}$$

$$\frac{\partial L}{\partial w^{(L-1)}} = \delta_1^{(L)} \odot a^{(L-1)} \; ; \; \frac{\partial L}{\partial b_1^{(L-1)}} = \delta_1^{(L)}$$

Multivariate version of chain rule.

$$f(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_p(x) \end{bmatrix} \qquad f : \mathbb{R} \to \mathbb{R}^p \qquad g : \mathbb{R}^p \to \mathbb{R}$$

$$h(x) = g(f_1(x), f_2(x) \cdots f_p(x))$$

$$x \in \mathbb{R} \to \begin{bmatrix} f_1(x) = y_1 \\ \vdots \\ f_p(x) = y_p \end{bmatrix} \in \mathbb{R}^p \to z \in \mathbb{R} = g(y)$$

$$h = g \circ f$$

$$h' = \nabla g \cdot f' = \sum \frac{\partial g}{\partial y_i} \frac{\partial y_i}{\partial x}.$$

Here is the iteration for computing the key $\delta_j^{(\ell)}$ quantities.

Here are the partial derivative in terms of the $\delta_j^{(\ell)}$.

Neural Nets in a Nutshell

Model and Loss

$a^{(1)} = x$; $z^{(\ell)} = b^{(\ell-1)} + W^{(\ell-1)} a^{(\ell-1)}$; $a^{(\ell)} = g^{(\ell)}(z^{(\ell)})$

$f(x, b, w) = a^{(L)}$; $\min_{b,w} \frac{1}{n} \sum_{i=1}^{n} L(y_i, f(x_i, b, w))$

Gradient Computation (Backprop)

- $\delta_i^{(L)} = \frac{\partial L}{\partial f} (g^L)'(z_i^{(L)})$
- $\delta^{(\ell)} = (g^{(\ell)})'(z^{(\ell)}) \odot [W^{(\ell)}]^T \delta^{(\ell+1)}$
- $\frac{\partial L}{\partial W^{(\ell)}} = [\delta^{(\ell+1)}][a^{(\ell)}]^T$ $\quad \frac{\partial L}{\partial b^{(\ell)}} = \delta^{(\ell+1)}$

*
- ε_k schedule
- Nesterov Momentum
- L^1, L^2 regularization
- Dropout

⋮

SGD: Stochastic Gradient Descent

Epochs : $k = 1, 2, .. K$ (pass through data)

Minibatches : $\{x_i^b, y_i^b\}$ $i = 1, 2, \cdots m$, $b = 1, 2, \cdots B$

ε_k: learing rate

$\Theta = (b, w)$

for $k = 1, 2, \cdots K$

for $b = 1, 2, \ldots B$

$\Theta \rightarrow \Theta - \varepsilon_k \frac{1}{m} \sum_{i=1}^{m} \nabla L(x_i^b, y_i^b, \Theta)$ *