Data Science for Actuaries (ACT6100)

Arthur Charpentier

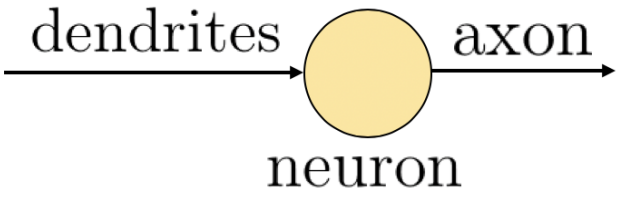
Méthodes supervisées # 6 (Introduction to Neural Nets)

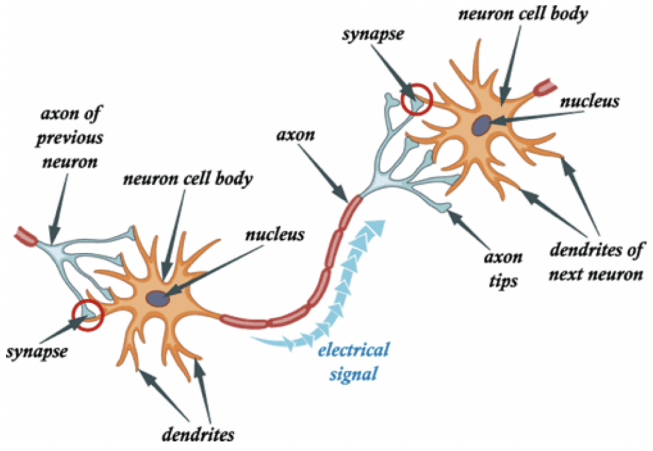
automne 2Q20

https://github.com/freakonometrics/ACT6100/

Neural Nets

A neuron is simply :





Human body has $\sim$ 100 billion neuron cells

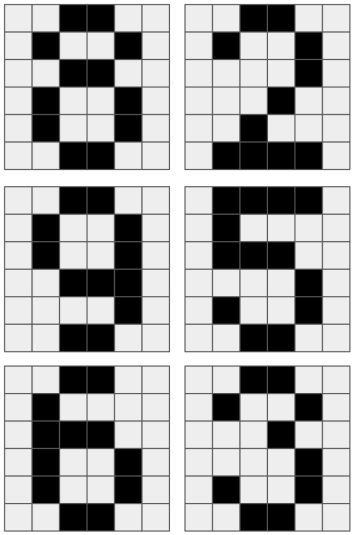
Good at : vision, speech, driving, playing games, etc

Neural Nets & Deep Learning in a Nutshell

Neural Nets are very popular for pictures...
Picture $\boldsymbol{x}_i$ is

- a $n \times n$ matrix in $\{0,1\}^{n^2}$ for black & white
- a $n \times n$ matrix in $[0,1]^{n^2}$ for grey-scale
- a $3 \times n \times n$ array in $([0,1]^3)^{n^2}$ for color
- a $T \times 3 \times n \times n$ tensor in $(([0,1]^3)^T)^{n^2}$ for video

y here is the label ("8", "9", "6", etc)
Suppose we want to recognize a "6" on a picture



$$m(\boldsymbol{x}) = \begin{cases} +1 \text{ if } \boldsymbol{x} \text{ is a "6"} \\ -1 \text{ otherwise} \end{cases}$$

Neural Nets & Deep Learning in a Nutshell

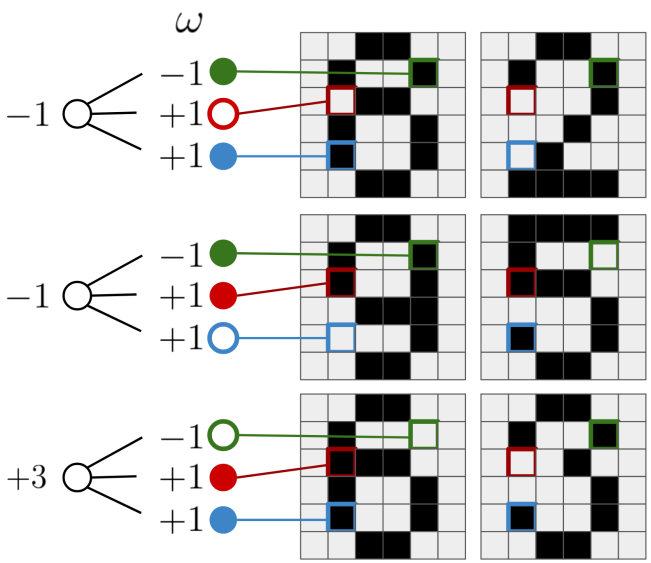
Consider some specifics pixels, and associate weights ω such that

$$\widehat{m}_i(\boldsymbol{x}) = \text{sign}\left(\sum_j \omega_{i,j} x_j\right)$$

where

$$x_j = \begin{cases} +1 \text{ if pixel } x_j \text{ is black} & \blacksquare \\ -1 \text{ if pixel } x_j \text{ is white} & \square \end{cases}$$

for some weights $\omega_{i,j}$ (that can be negative...)



Neural Nets & Deep Learning in a Nutshell

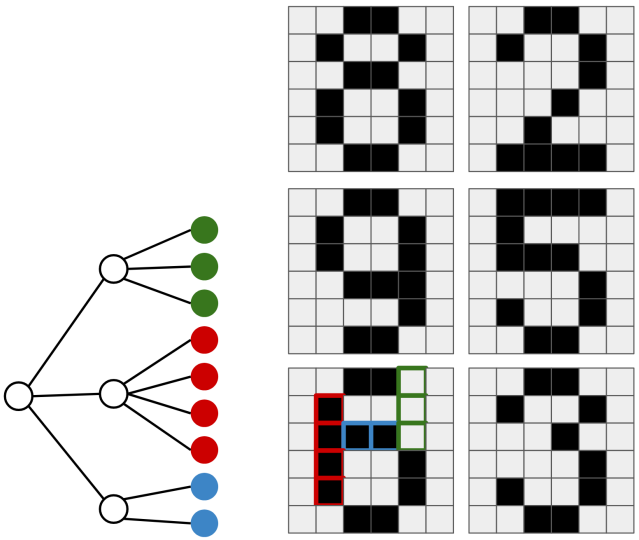
A deep network is a network with a lot of layers

$$\widehat{m}(\boldsymbol{x}) = \text{sign}\left(\sum_i \omega_i \widehat{m}_i(\boldsymbol{x})\right)$$

where $\widehat{m}_i$'s are outputs of previous neural nets

Those layers can capture shapes in some areas

nonlinearities, cross-dependence, etc.



Binary Threshold Neuron & the Perceptron

If $\boldsymbol{x} \in \{0,1\}^p$, McCulloch & Pitts (1943) suggested a simple model, with threshold b

$$y_i = f\left(\sum_{j=1}^{p} x_{j,i}\right) \text{ where } f(x) = \mathbf{1}(x \geq b)$$

where $\mathbf{1}(x \geq b) = +1$ if $x \geq b$ and 0 otherwise, or (equivalently)

$$y_i = f\left(\omega + \sum_{j=1}^{p} x_{j,i}\right) \text{ where } f(x) = \mathbf{1}(x \geq 0)$$

with weight $\omega = -b$. The trick of adding 1 as an input was very important !

$\omega = -1$ is the or logical operator : $y_i = 1$ if $\exists j$ such that $x_{j,i} = 1$

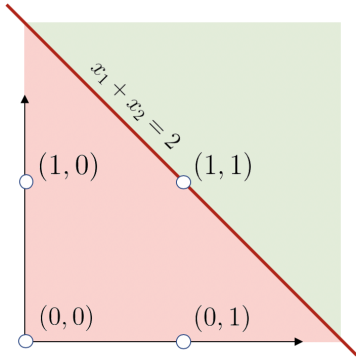
$\omega = -p$, is the and logical operator : $y_i = 1$ if $\forall j$, $x_{j,i} = 1$

Binary Threshold Neuron & the Perceptron

but not possible for the xor logical operator : $y_i = 1$ if $x_{1,i} \neq x_{2,i}$

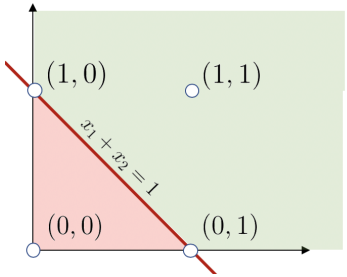
Rosenblatt (1961) considered the extension where x's are real-valued, with weight $\boldsymbol{\omega} \in \mathbb{R}^p$

$$y_i = f\left(\sum_{j=1}^{p} \omega_j x_{j,i}\right) \text{ where } f(x) = \mathbf{1}(x \geq b)$$



where $\mathbf{1}(x \geq b) = +1$ if $x \geq b$ and 0 otherwise, or (equivalently)

$$y_i = f\left(\omega_0 + \sum_{j=1}^{p} \omega_j x_{j,i}\right) \text{ where } f(x) = \mathbf{1}(x \geq 0)$$



with weights $\boldsymbol{\omega} \in \mathbb{R}^{p+1}$

Binary Threshold Neuron & the Perceptron

Minsky & Papert (1969) proved that perceptron were a linear separator, not very powerful

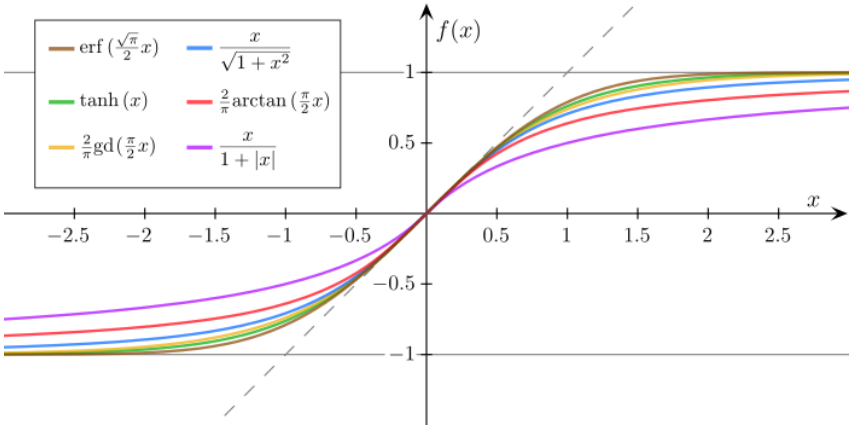
Define the sigmoid function $f(x) = \dfrac{1}{1+e^{-x}} = \dfrac{e^x}{e^x+1}$ (= logistic).

This function f is called the activation function.

If $y \in \{-1, +1\}$, one can consider the hyperbolic tangent

$$f(x)) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$$

or the inverse tangent function (see wikipedia).



Classification : Neural Nets

So here, for a classication problem,

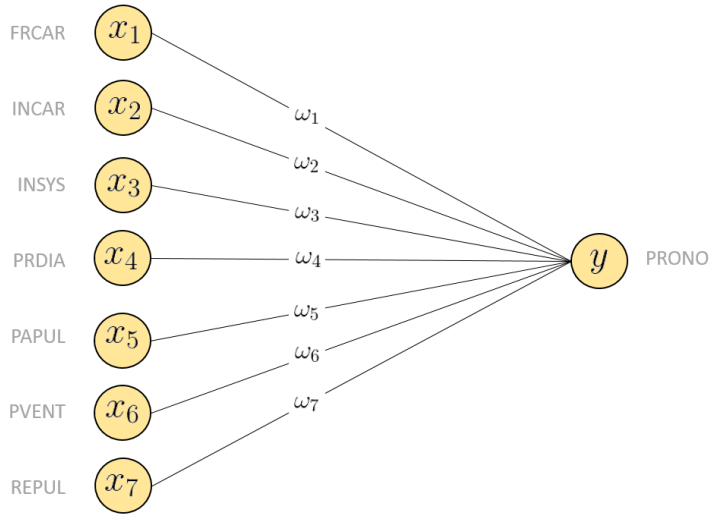
$$y_i = f\left(\omega_0 + \sum_{j=1}^{p} \omega_j x_{j,i}\right) = p(\boldsymbol{x}_i)$$

The error of that model can be computed used the quadratic loss function,

$$\sum_{i=1}^{n} (y_i - p(\boldsymbol{x}_i))^2$$

or cross-entropy

$$\sum_{i=1}^{n} (y_i \log p(\boldsymbol{x}_i) + [1 - y_i] \log[1 - p(\boldsymbol{x}_i)])$$



Classification : Neural Nets

If

$$p(\boldsymbol{x}) = f\left(\omega_0 + \sum_{h=1}^{3} \omega_h f_h\left(\omega_{h,0} + \boldsymbol{x}^\top \boldsymbol{\omega}_h\right)\right)$$

we want to solve, with back propagation,

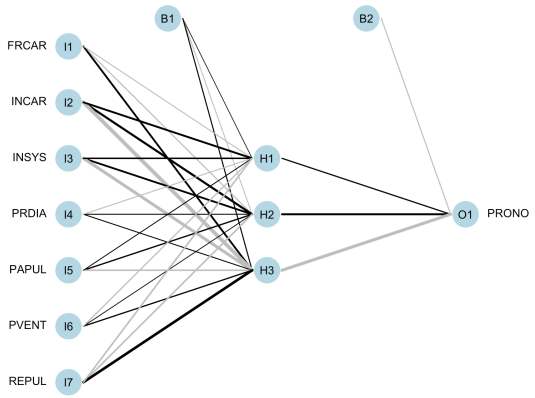
$$\boldsymbol{\omega}^\star = \text{argmin}\left\{\frac{1}{n}\sum_{i=1}^{n} \ell\left(y_i, p(\boldsymbol{x}_i)\right)\right\}$$

To visualize a neural network, use

```
> library(nnet)
> library(NeuralNetTools)
```

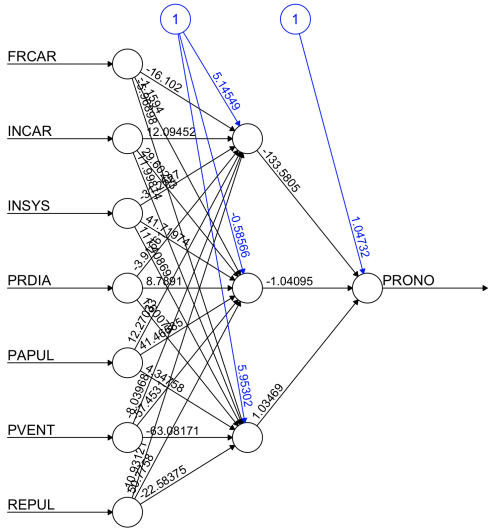
Classification : Neural Nets

```
library(nnet)
mix = function(z) (z-min(z))/(max(z)-min(z))
myocarde = apply(myocarde, 1, mix)
fit_nn = nnet(PRONO~., data=myocarde, size=3)
plotnet(fit_nn)
```



or

```
library(neuralnet)
fit_nn = neuralnet(formula(glm(PRONO~., data = myocarde)), myocarde, hidden = 3, act.fct = sigmoid)
plot(fit_nn)
```



Classification : Neural Nets

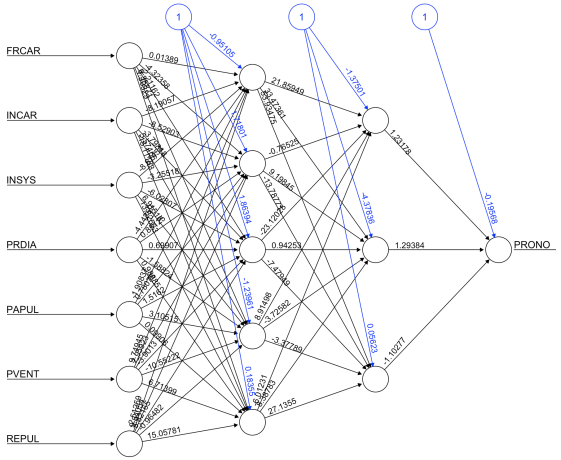
Of course, one can consider a multiple layer network,

$$p(\boldsymbol{x}) = f\left(\omega_0 + \sum_{h=1}^{3} \omega_h f_h\left(\omega_{h,0} + \boldsymbol{z}_h^\top \boldsymbol{\omega}_h\right)\right)$$

where

$$\boldsymbol{z}_h = f\left(\omega_{h,0} + \sum_{j=1}^{k_h} \omega_{h,j} f_{h,j}\left(\omega_{h,j,0} + \boldsymbol{x}^\top \boldsymbol{\omega}_{h,j}\right)\right)$$

```
library(neuralnet)
model_nnet = neuralnet(formula(glm(PRONO
    ~., data = myocarde)), myocarde,
    hidden = 3, act.fct = sigmoid)
plot(model_nnet)
```



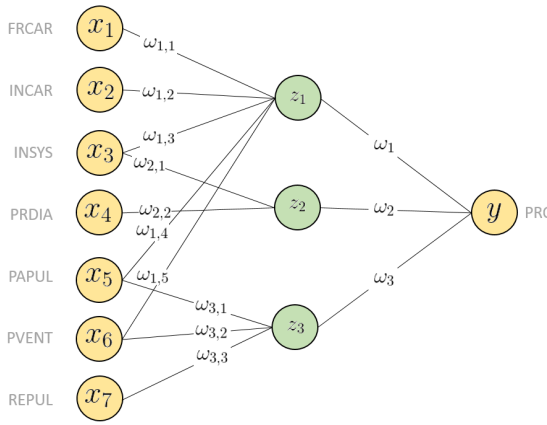
Classification : Neural Nets

Consider a single hidden layer, with 3 different neurons, so that

$$p(\boldsymbol{x}) = f\left(\omega_0 + \sum_{h=1}^{3} \omega_h f_h \left(\omega_{h,0} + \sum_{j=1}^{p} \omega_{h,j} x_j\right)\right)$$

or

$$p(\boldsymbol{x}) = f\left(\omega_0 + \sum_{h=1}^{3} \omega_h f_h \left(\omega_{h,0} + \boldsymbol{x}^\top \boldsymbol{\omega}_h\right)\right)$$



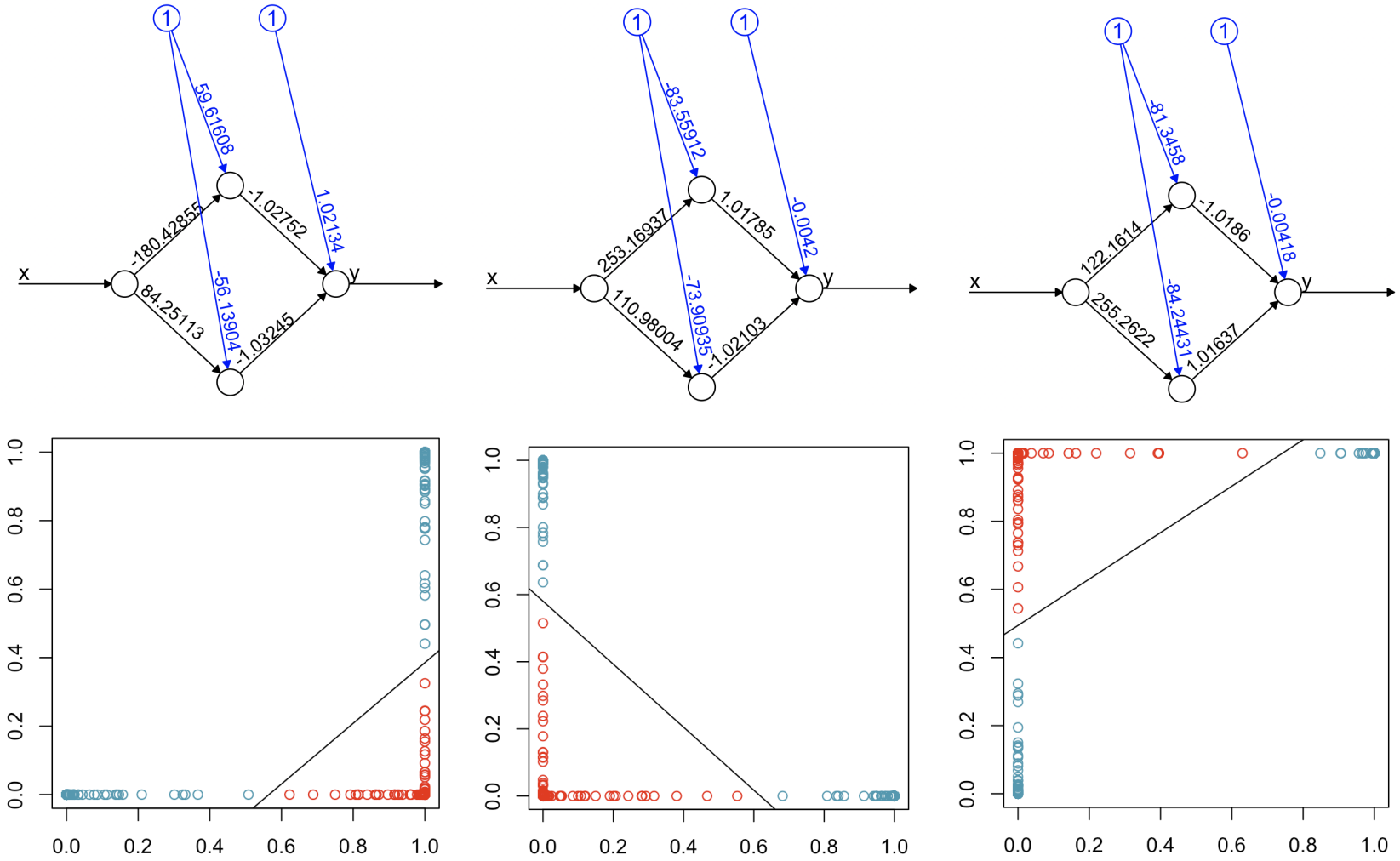
Remark it could be good to have more neurons in the hidden layer than features $x_1, \cdots, x_p$

Interpreting Hidden Layers

```
> n = 100
> x = c(runif(n),1+runif(n),2+runif(n))
> y = rep(c(0,1,0),each=n)
> minmax = function(z) (z-min(z))/(max(z)-min(z))
> xm = minmax(x)
> df = data.frame(x=xm,y=y)
```

```
> sigmoid = function(x) 1 / (1 + exp(-x))
> model_nnet = nnet(y~x,size=2,data=df)
> plotnet(model_nnet)
> library(NeuralNetTools)
> w = neuralweights(model_nnet)
> x1 = cbind(1,df$x)%*%w$wts$"hidden 1 1"
> x2 = cbind(1,df$x)%*%w$wts$"hidden 1 2"
> b = w$wts$`out 1`
> plot(sigmoid(x1),sigmoid(x2))
```

Interpreting Hidden Layers

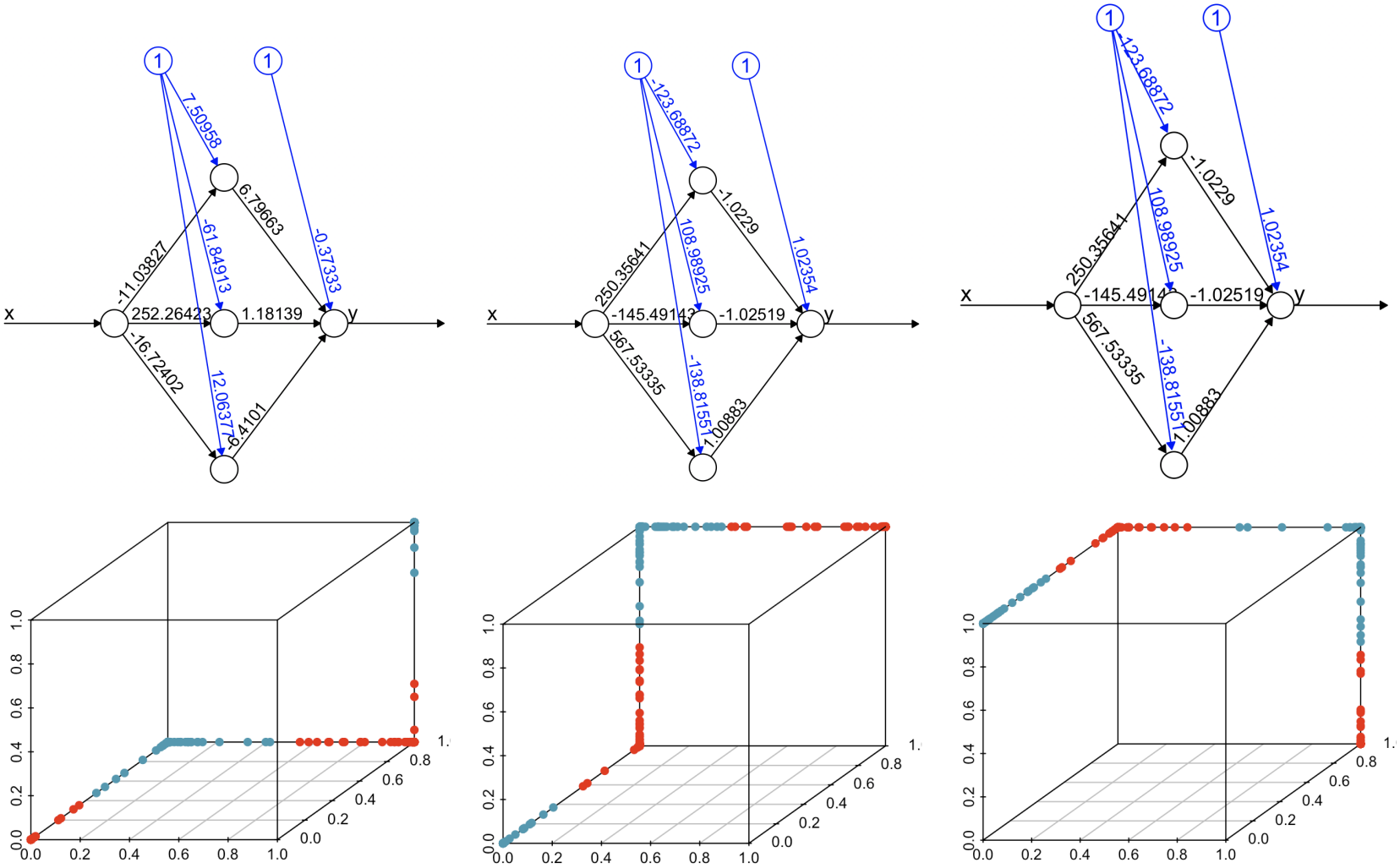


Interpreting Hidden Layers

```
> n=100
> x=c(runif(n),1+runif(n),2+runif(n),3+runif(n))
> y=rep(c(0,1,0,1),each=n)
> xm = minmax(x)
> df = data.frame(x=xm,y=y)
```

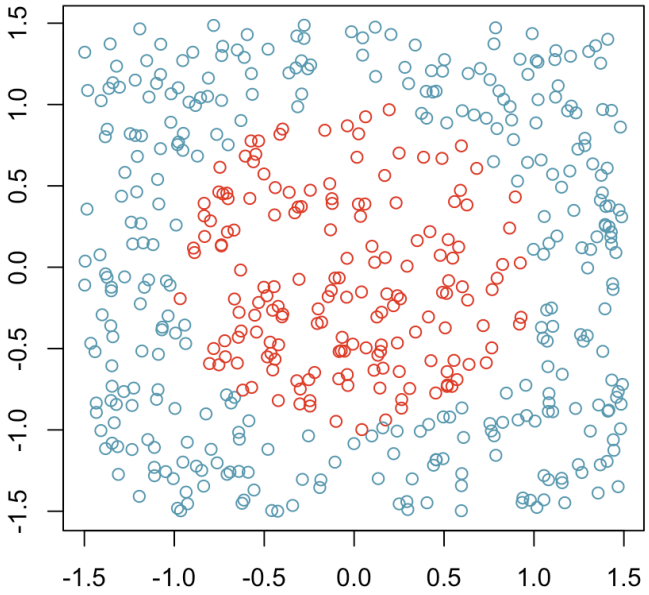
```
> model_nnet = nnet(y~x,size=3,data=df)
> plotnet(model_nnet)
> w = neuralweights(model_nnet)
> x1 = cbind(1,df$x)%*%w$wts$"hidden 1 1"
> x2 = cbind(1,df$x)%*%w$wts$"hidden 1 2"
> x3 = cbind(1,df$x)%*%w$wts$"hidden 1 3"
> b = w$wts$`out 1`
> library(scatterplot3d)
> scatterplot3d(x=sigmoid(x1), y=sigmoid(x2), z=
    sigmoid(x3)
```

Interpreting Hidden Layers



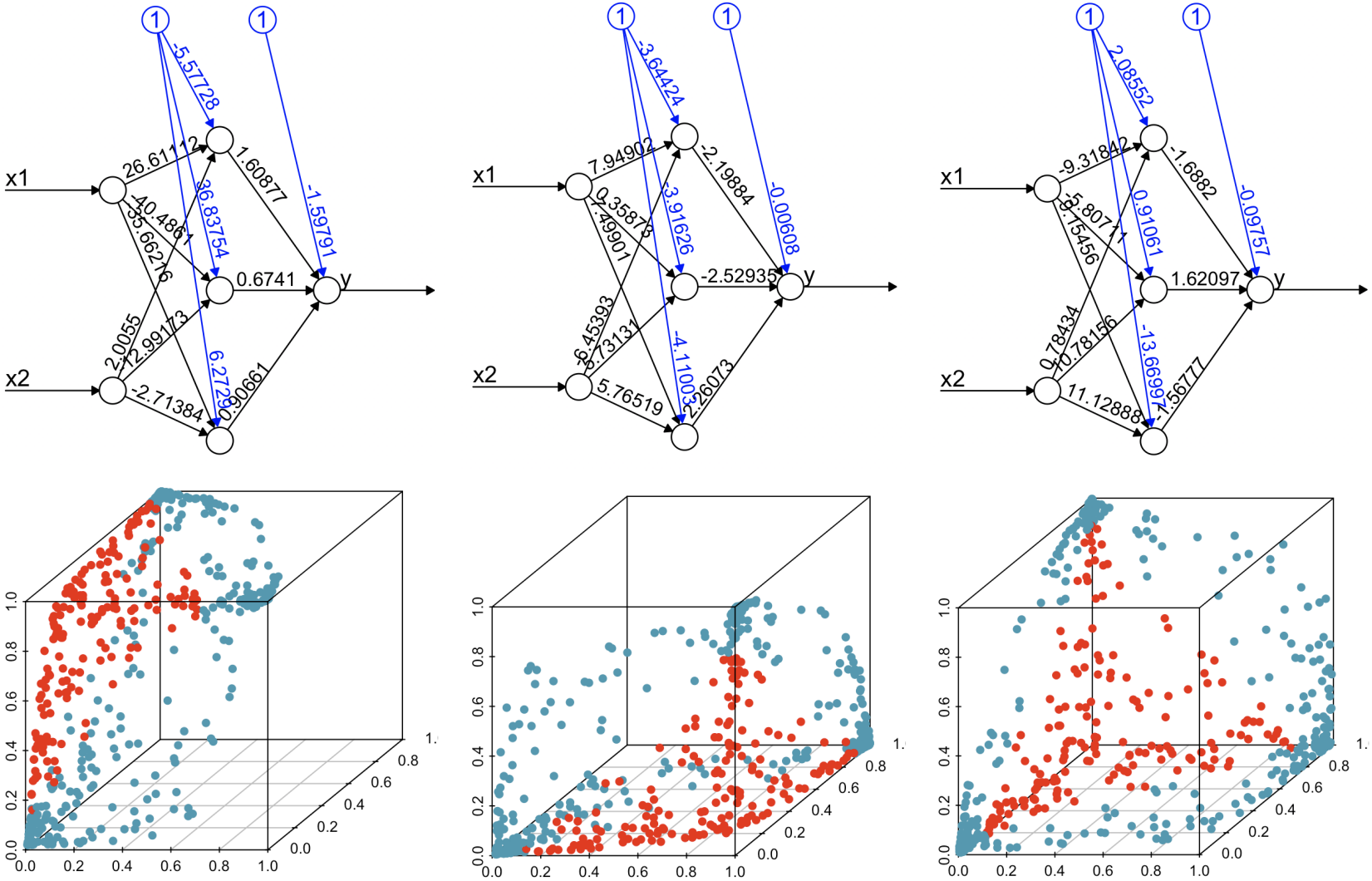
Interpreting Hidden Layers

```
> n=500
> x1=runif(n)*3-1.5
> x2=runif(n)*3-1.5
> y = (x1^2+x2^2)<=1
> x1m = minmax(x1)
> x2m = minmax(x2)
> df = data.frame(x1=x1m,x2=x2m,y=y)
```



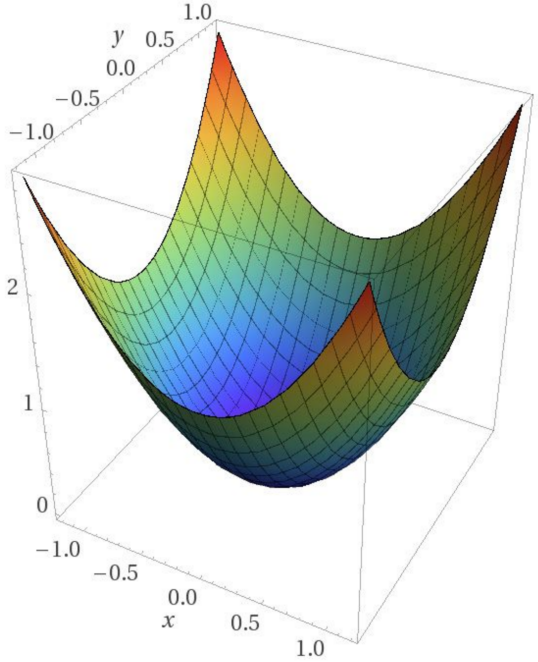

```
> model_nnet = nnet(y~x,size=3,data=df)
> plotnet(model_nnet)
> w = neuralweights(model_nnet)
> x1 = cbind(1,df$x1,df$x2)%*%w$wts$"hidden 1 1"
> x2 = cbind(1,df$x1,df$x2)%*%w$wts$"hidden 1 2"
> x3 = cbind(1,df$x1,df$x2)%*%w$wts$"hidden 1 3"
> b = w$wts$`out 1`
> library(scatterplot3d)
> scatterplot3d(x=sigmoid(x1), y=sigmoid(x2), z=
    sigmoid(x3)
```

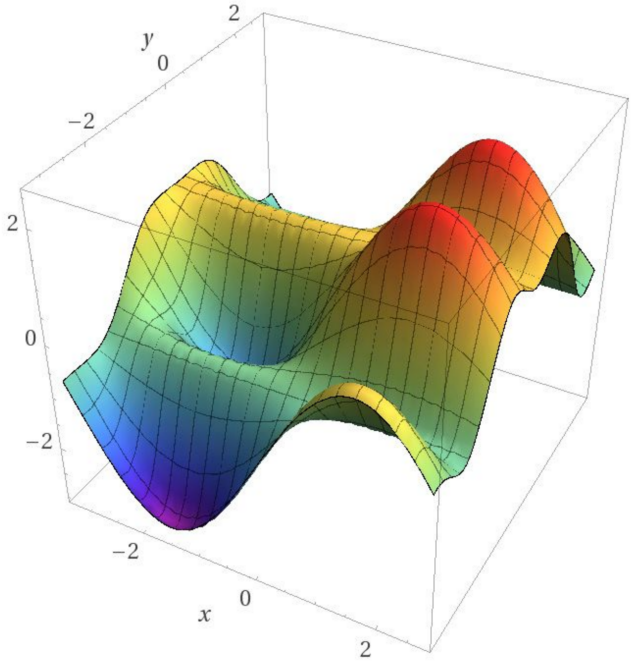
Interpreting Hidden Layers



Optimization Issues

If classical econometrics/regression is based on convex optimization problems (most the time), (deep) Neural Networks are usually not convex problems





Optimization Issues

Consider optimization problem

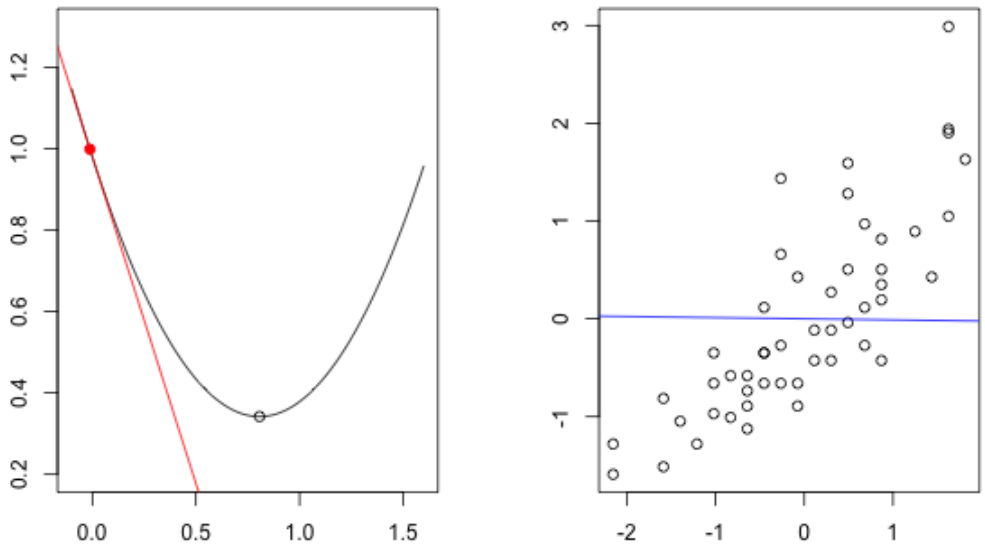
$$\min_{\boldsymbol{x}\in\mathcal{X}} \left\{ f(\boldsymbol{x}) \right\}$$

solved by gradient descent,

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \gamma_n \nabla f(\boldsymbol{x}_n), \;\; n \geq 0$$

with starting point $\boldsymbol{x}_0$

γ_n can be call learning rate



Sometime the dimension of the dataset can be really big: we can't pass all the data to the computer at once to compute $f(\boldsymbol{x})$

we need to divide the data into smaller sizes and give it to our computer one by one

Optimization Issues

Consider optimization problem

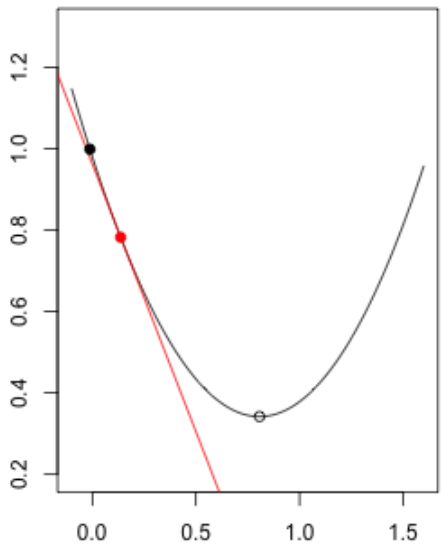
$$\min_{\boldsymbol{x}\in\mathcal{X}} \left\{ f(\boldsymbol{x}) \right\}$$

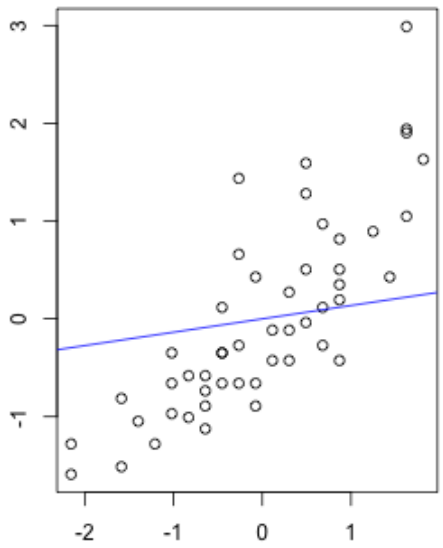
solved by gradient descent,

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \gamma_n \nabla f(\boldsymbol{x}_n), \;\; n \geq 0$$

with starting point $\boldsymbol{x}_0$

γ_n can be call learning rate





Sometime the dimension of the dataset can be really big: we can't pass all the data to the computer at once to compute $f(\boldsymbol{x})$

we need to divide the data into smaller sizes and give it to our computer one by one

Optimization Issues

Consider optimization problem

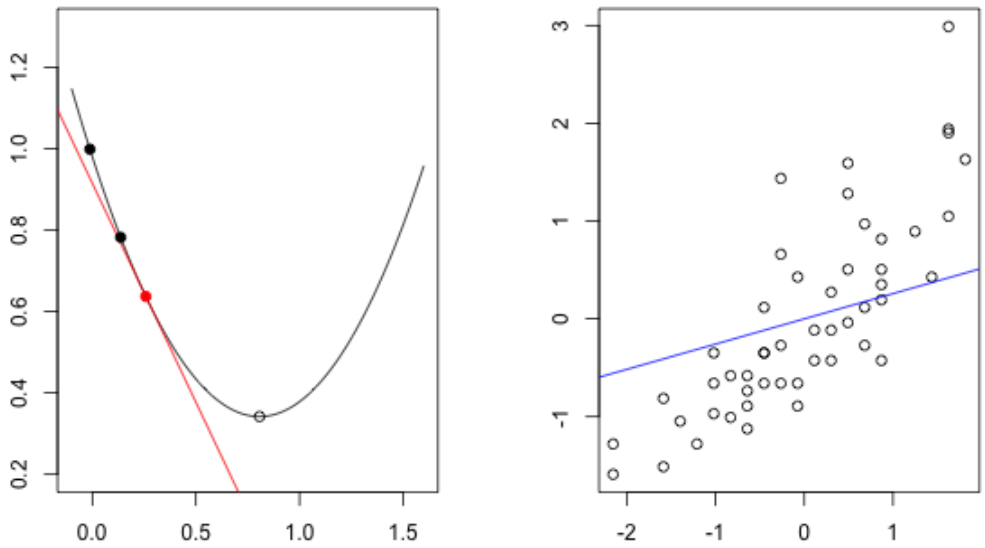
$$\min_{\boldsymbol{x}\in\mathcal{X}} \left\{ f(\boldsymbol{x}) \right\}$$

solved by gradient descent,

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \gamma_n \nabla f(\boldsymbol{x}_n), \;\; n \geq 0$$

with starting point $\boldsymbol{x}_0$

γ_n can be call learning rate



Sometime the dimension of the dataset can be really big: we can't pass all the data to the computer at once to compute $f(\boldsymbol{x})$

we need to divide the data into smaller sizes and give it to our computer one by one

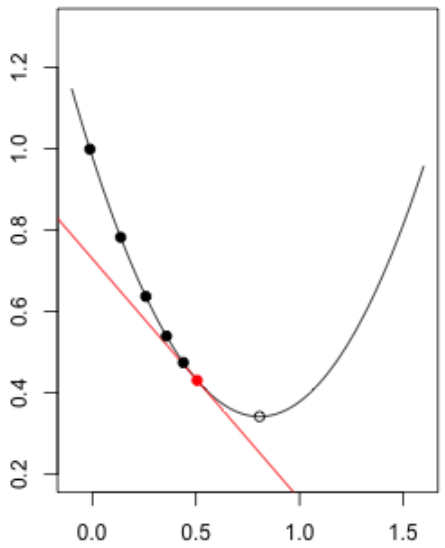
Optimization Issues

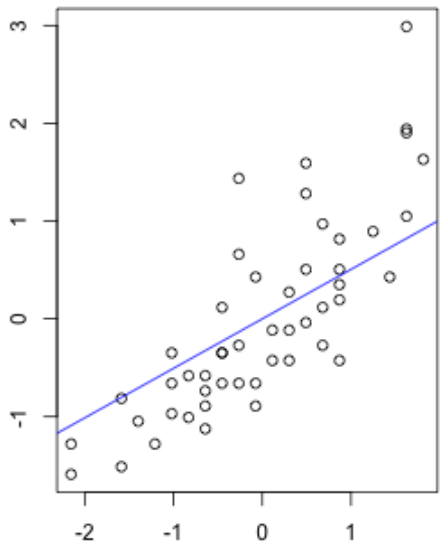
Consider optimization problem

$$\min_{\boldsymbol{x}\in\mathcal{X}} \left\{ f(\boldsymbol{x}) \right\}$$

solved by gradient descent,

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \gamma_n \nabla f(\boldsymbol{x}_n), \;\; n \geq 0$$





with starting point $\boldsymbol{x}_0$
γ_n can be call learning rate
Sometime the dimension of the dataset can be really big: we can't pass all the data to the computer at once to compute $f(\boldsymbol{x})$
we need to divide the data into smaller sizes and give it to our computer one by one

Optimization Issues

Consider optimization problem

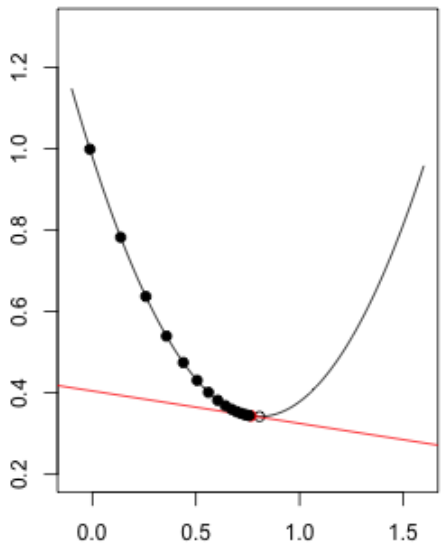
$$\min_{\boldsymbol{x}\in\mathcal{X}} \left\{ f(\boldsymbol{x}) \right\}$$

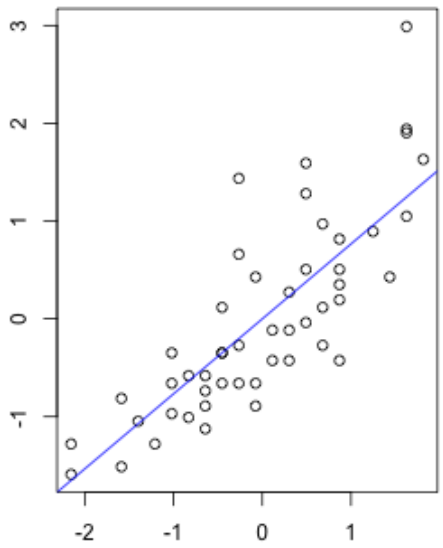
solved by gradient descent,

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \gamma_n \nabla f(\boldsymbol{x}_n), \;\; n \geq 0$$

with starting point $\boldsymbol{x}_0$

γ_n can be call learning rate





Sometime the dimension of the dataset can be really big: we can't pass all the data to the computer at once to compute $f(\boldsymbol{x})$

we need to divide the data into smaller sizes and give it to our computer one by one

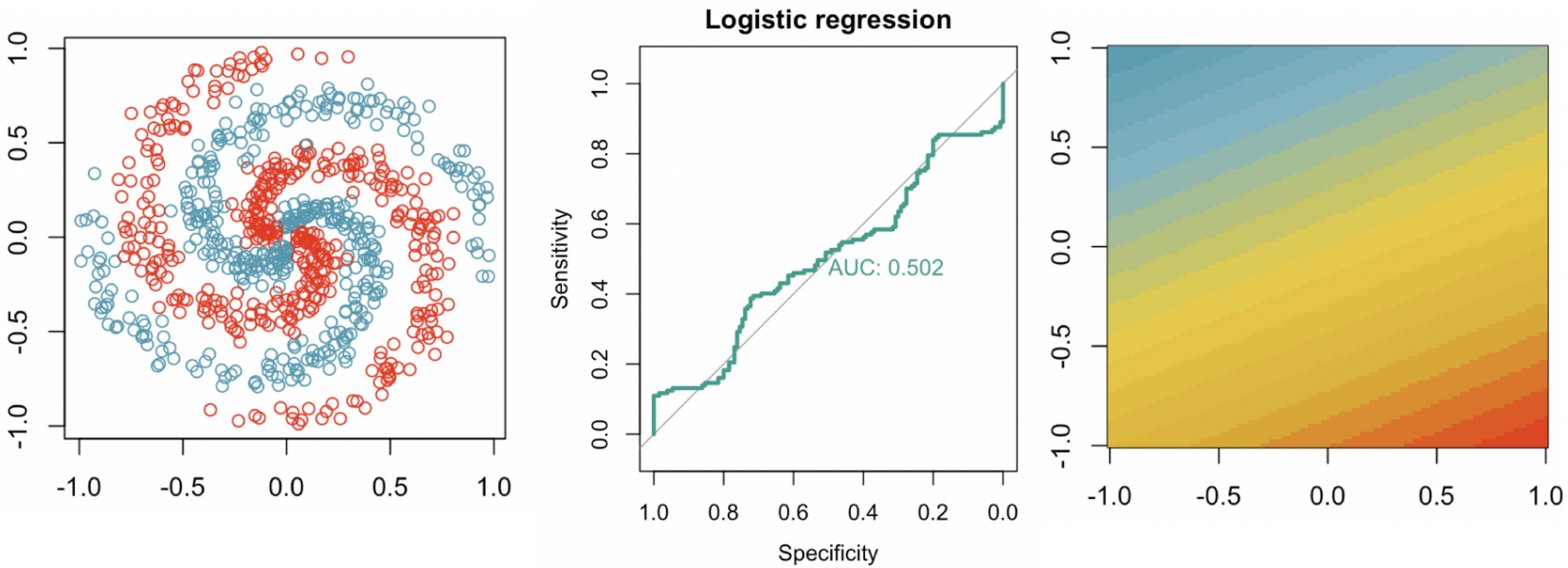
Classification & Non-linearities

Consider some spiral-shaped data

```
n = 200
K = 4
X = data.frame()
for (j in (1:K)){
  r = seq(0.05,1,length = n)
  e = rnorm(n, sd = 0.3)
  t = seq((j-1)*4.7,j*4.7, length = n) + e
  X = rbind(X, data.frame(x =r*sin(t) , y = r*cos(t)))
}
Y = rep(c(1,2,1,2),each=n)
data = cbind(X,y = Y)
data$y = as.factor(data$y)
idx = sample(1:nrow(data),size=nrow(data)*2/3)
df_train = data[idx,]
df_valid = data[-idx,]
```

Classification & Non-linearities

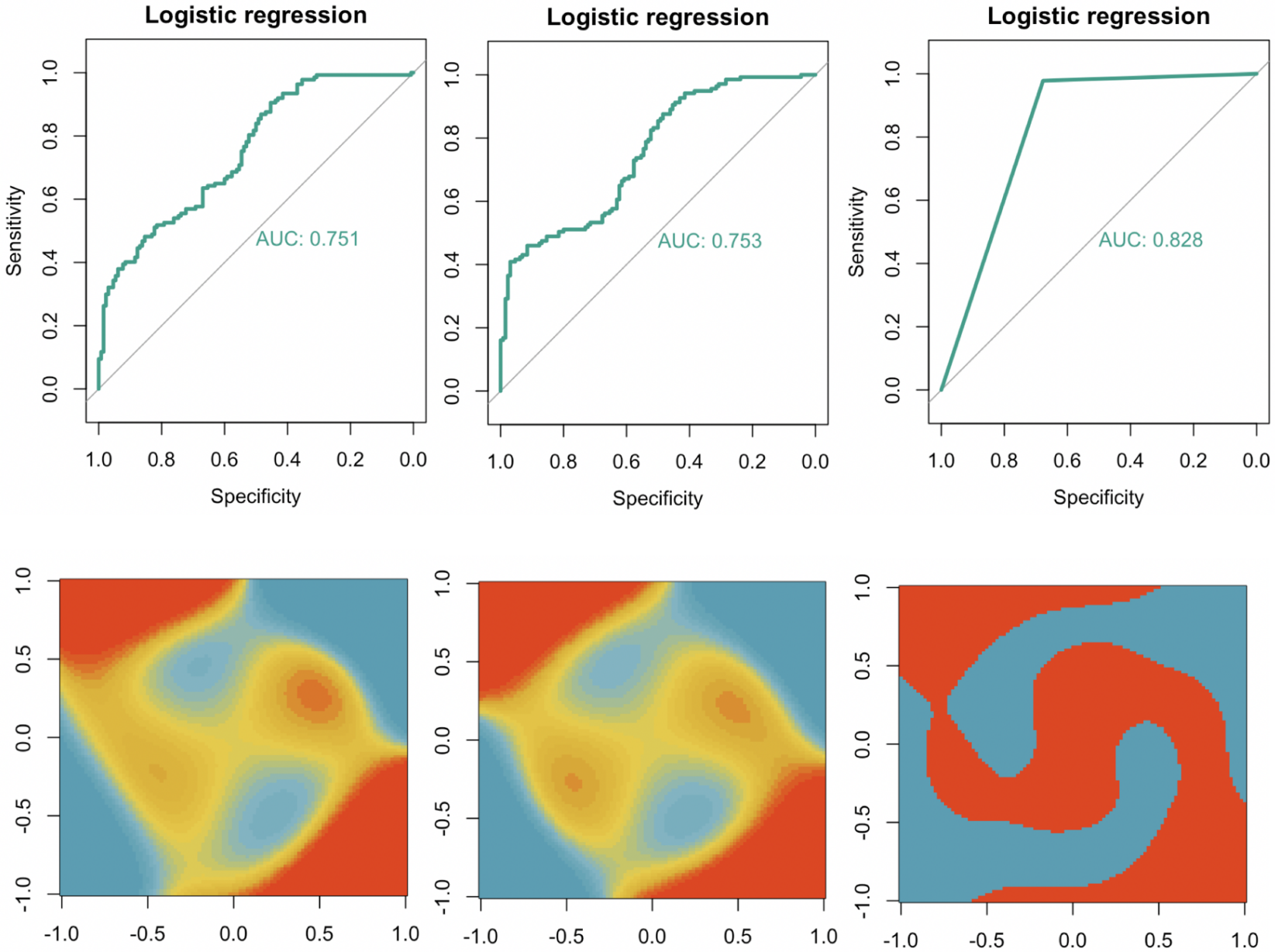
- logistic regression, $\mathbb{P}[Y=1|\boldsymbol{x}] = \dfrac{e^{\boldsymbol{x}^\top\boldsymbol{\beta}}}{1+e^{\boldsymbol{x}^\top\boldsymbol{\beta}}}$



is it really a surprise? $\mathbb{P}[Y=1|\boldsymbol{x}] = \dfrac{e^{\varphi(\boldsymbol{x})}}{1+e^{\varphi(\boldsymbol{x})}}$ for some polynomial $\varphi : \mathbb{R}^2 \to \mathbb{R}$, e.g.

$$\varphi(x_1, x_2) = \beta_0 + \beta_1 x_1^4 + \beta_2 x_1^3 x_2 + \beta_3 x_1^2 x_2^2 + \beta_4 x_1 x_2^3 + \beta_5 x_2^4$$

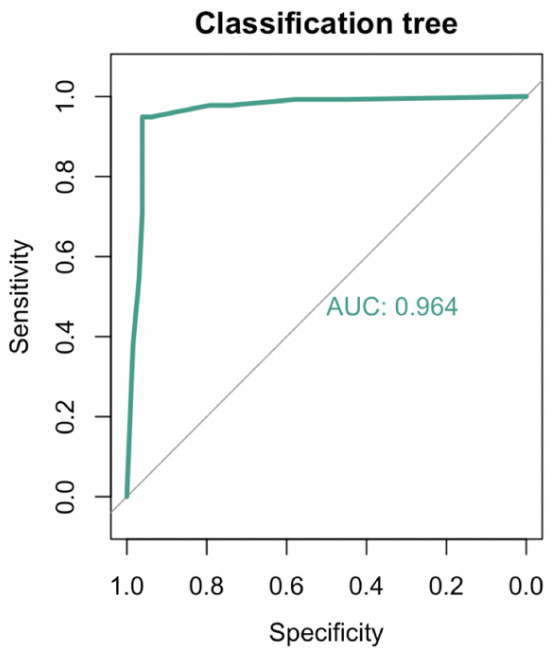
Classification & Non-linearities

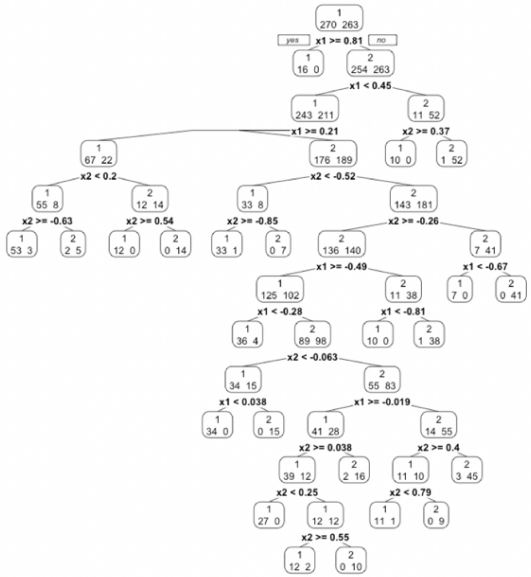


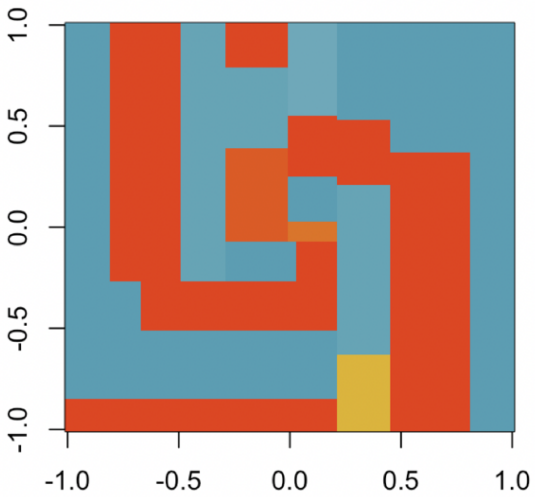
Classification & Non-linearities

- (single) classification tree

```
library(rpart)
cart = rpart(y~., data=df_train)
yhat_cart = predict(cart,newdata=df_valid,type="prob")
    [,2]
```



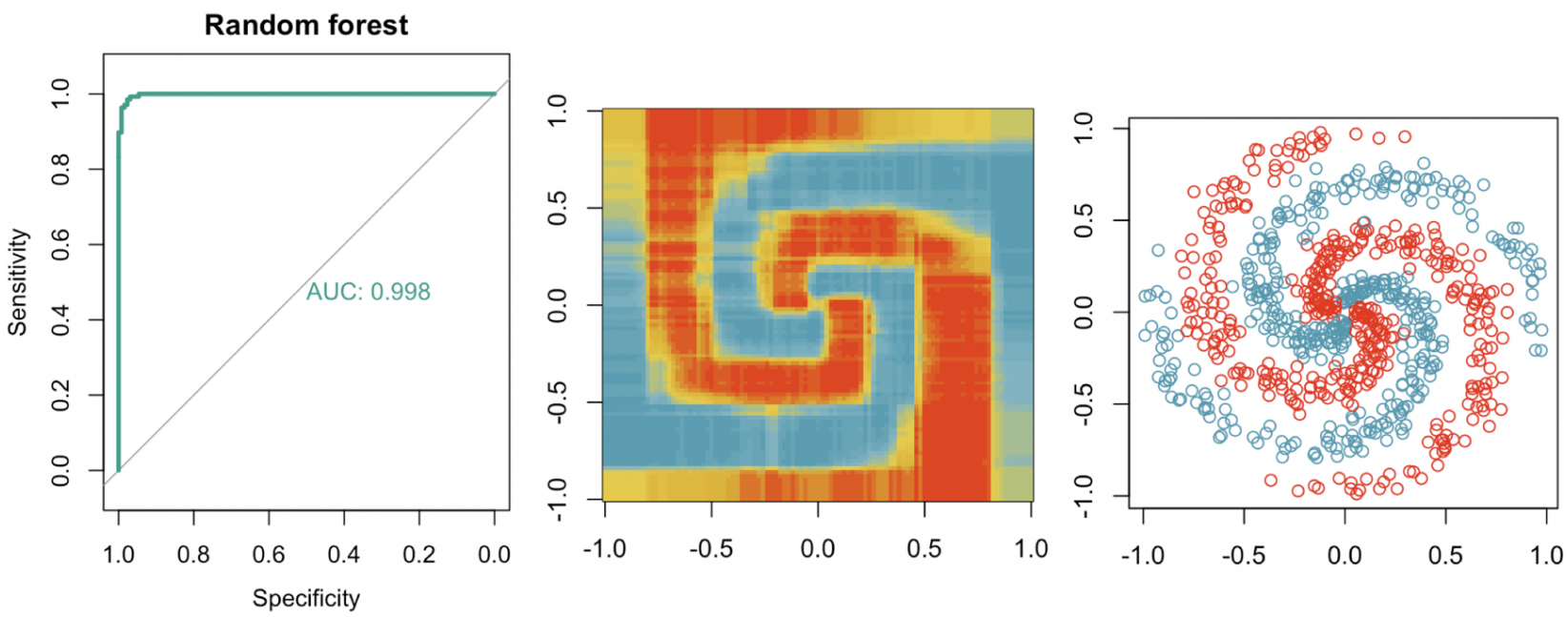




Classification & Non-linearities

- random forest

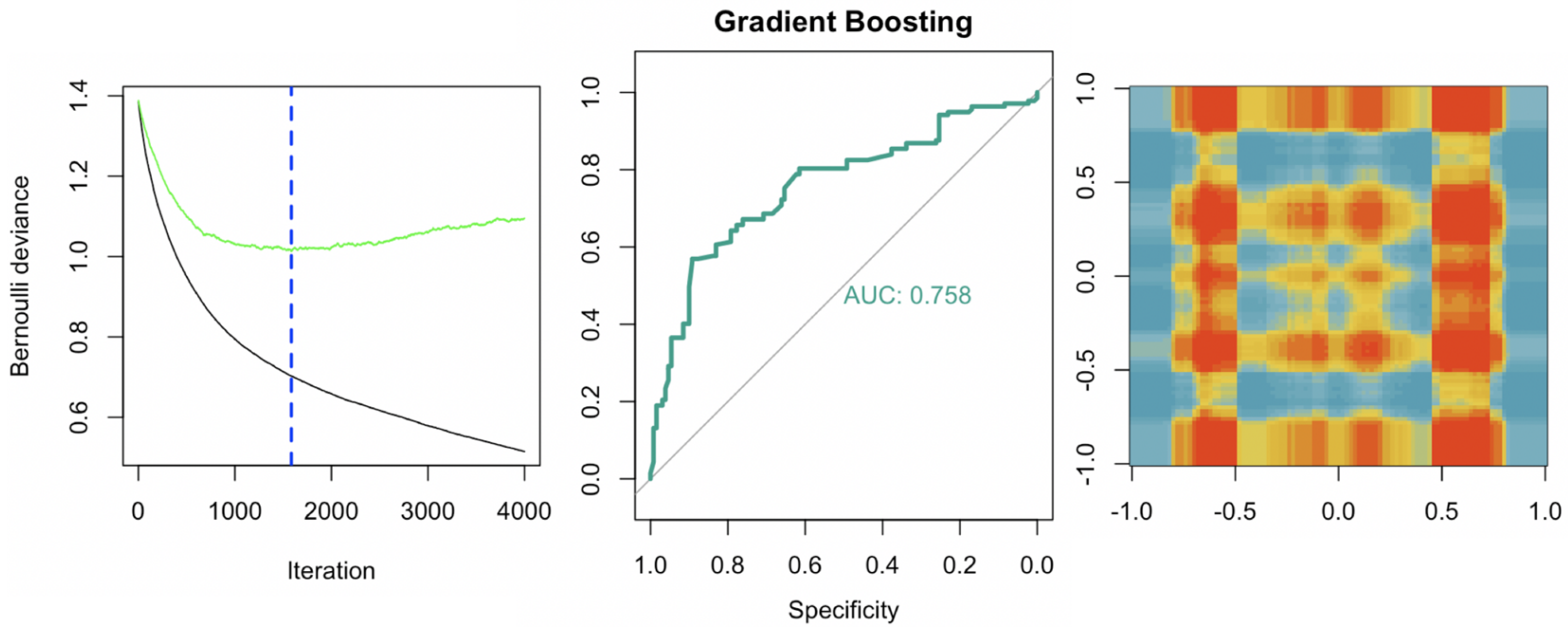
```
library(randomForest)
rf = randomForest(y~., data=df_train,)
yhat_rf = predict(rf, newdata=df_valid,type="prob")[,
    2]
```



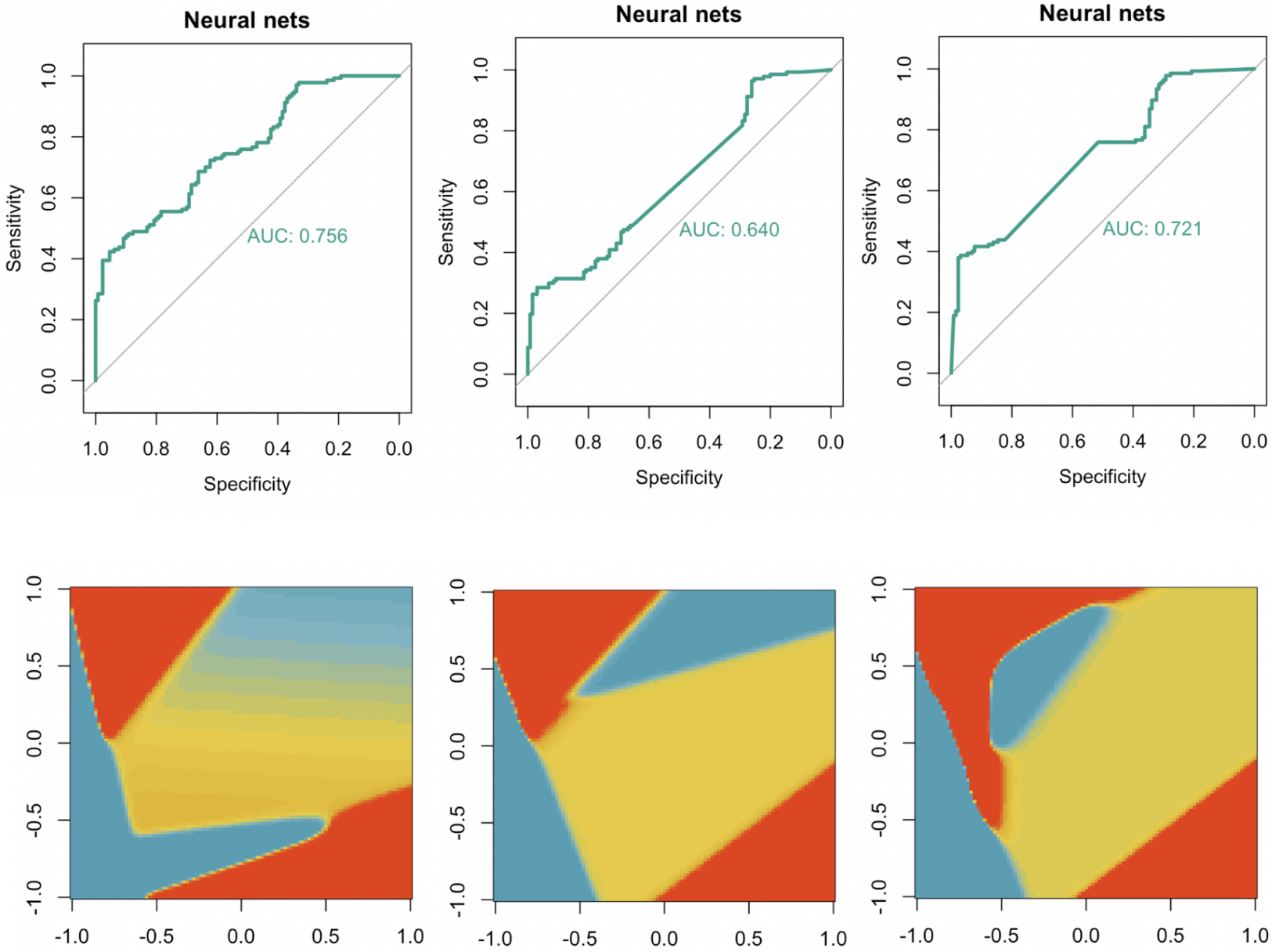
Classification & Non-linearities

- gradient boosting

```
library(gbm)
gb = gbm(as.numeric(y)-1~., data=df_train,
    distribution = "bernoulli", n.trees = 4000,
    shrinkage = .05, cv.folds = 10)
gbm.perf(gb,method="cv")
yhat = predict(gb, newdata=df_valid,type="response")
```



Classification & Non-linearities

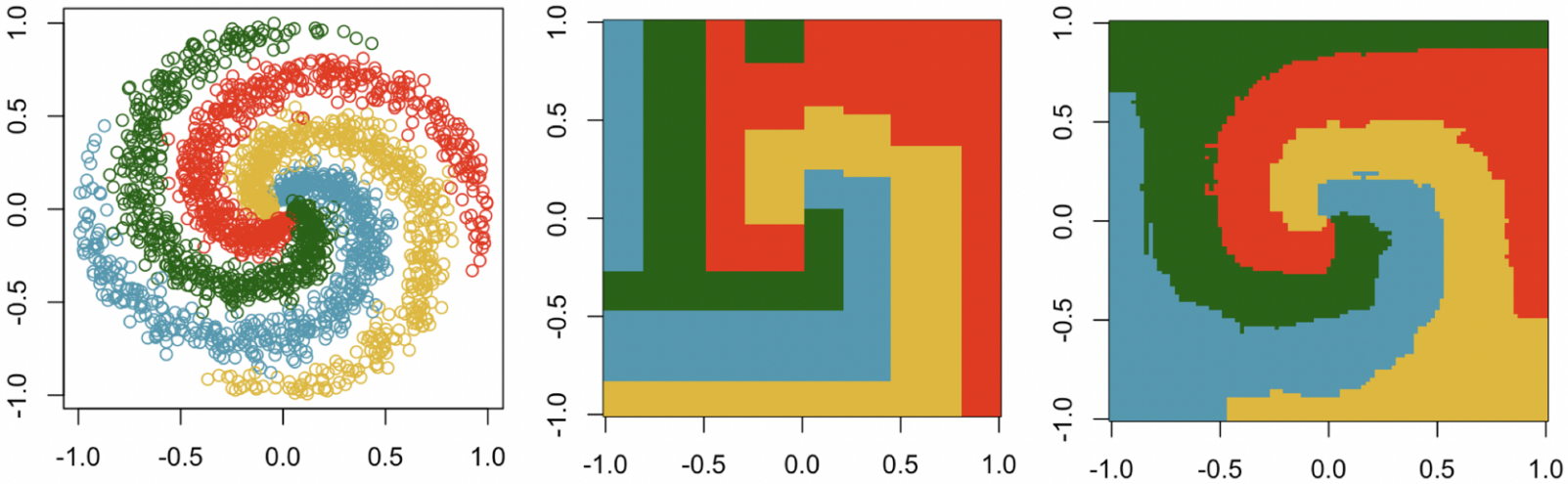


Classification & Non-linearities

One can consider a multiple-class, problem, e.g. 4 classes

```
Y = rep(c(1,2,3,4), each=n)
data = cbind(X, y=Y)
data$y = as.factor(data$y)
idx = sample(1:nrow(data),size=nrow(data)*2/3)
df_train = data[idx,]
df_valid = data[-idx,]
```

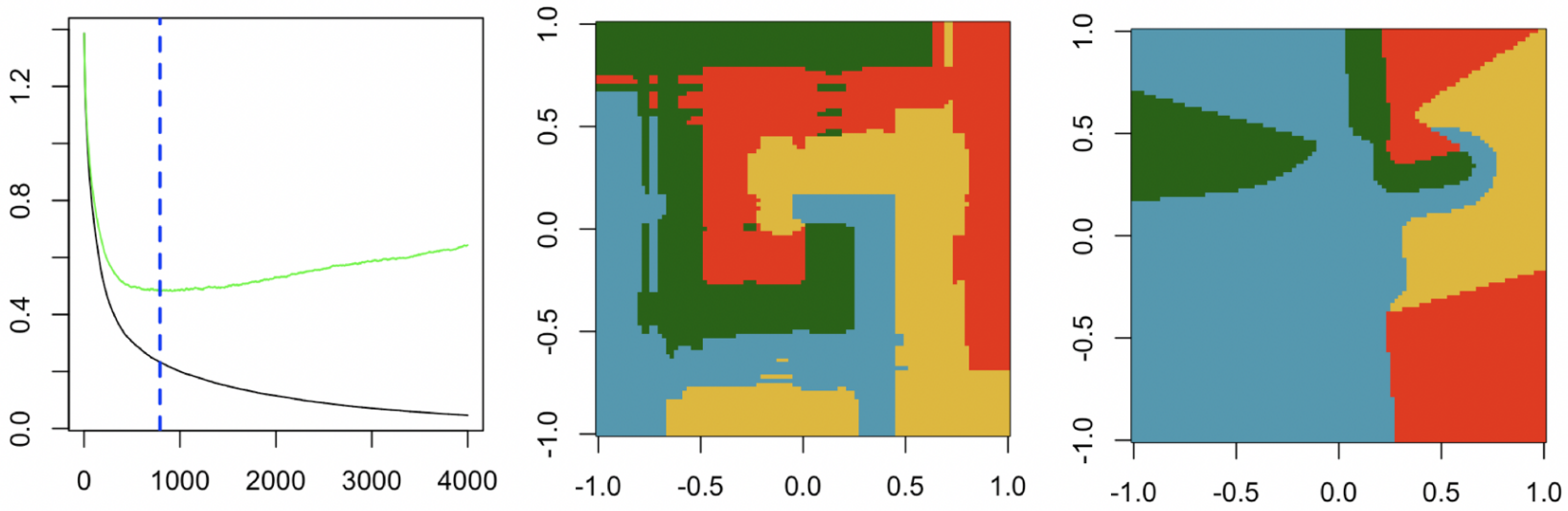
- single tree & random forest



Classification & Non-linearities

- boosting & neural nets

```
df_train_01 = df_train
df_train_01[1:2] = (df_train_01[1:2]+1)/2
require(caret)
nn = train(as.factor(y)~., data=df_train_01, method='nnet)
yhat_nn = predict(nn, (df_valid[,1:2]+1)/2, type='prob')[,2]
```



again, neural nets are not performing well...

Let use try some home-made function... (inspired by Jun Ma)

Classification & Non-linearities

Consider some neural net with 2 layers, and h=10 nodes
from matrix $\boldsymbol{X}$ and vector $\boldsymbol{y}$ (that will be treated as a 0/1 matrix),
let $\boldsymbol{W}_1 \in \mathcal{M}_{2\times h}$, $\boldsymbol{b}_1 \in \mathbb{R}^h$, $\boldsymbol{W}_2 \in \mathcal{M}_{h\times 4}$ and $\boldsymbol{b}_2 \in \mathbb{R}^4$ denote the weights for each layer,
we could have some learning rate and regularization parameters
Consider some ReLU activation function ($x \mapsto x_+$), and a softmax loss function (logistic regression)

$$z_{i,j}^{(1)} = (\boldsymbol{x}_i^\top \boldsymbol{W}_{1j} + \boldsymbol{b}_{1j})_+ \text{ and } z_{i,j}^{(2)} = \boldsymbol{z}_i^\top \boldsymbol{W}_{2j} + \boldsymbol{b}_{2j}$$

then

$$p_{ij} = \frac{\exp[z_{i,j}^{(2)}]}{\exp[z_{i,1}^{(2)}] + \cdots + \exp[z_{i,4}^{(2)}]} \text{ and } \ell_i = -log(\boldsymbol{p}_i)^\top \boldsymbol{y}$$

then use

$$\text{loss } = \frac{1}{n}\sum_{i=1}^{n} \ell_i + \frac{\alpha}{2}(\boldsymbol{W}_1^\top \boldsymbol{W}_1 + \boldsymbol{W}_2^\top \boldsymbol{W}_2)$$

Classification & Non-linearities

Let $d\boldsymbol{s}_i = \dfrac{\boldsymbol{p}_i - \boldsymbol{y}_i}{n}$ so that

$$d\boldsymbol{W}_{2i} = \boldsymbol{z}_i^{(2)\top} d\boldsymbol{s},\ d\boldsymbol{b}_2 = d\boldsymbol{s}^\top \mathbf{1}$$

$$d\boldsymbol{z}_i = (d\boldsymbol{s}\boldsymbol{W}_2^\top)_+,\ d\boldsymbol{W}_1 = \boldsymbol{X}^\top d\boldsymbol{z} \text{ and } d\boldsymbol{b}_1 = d\boldsymbol{z}^\top \mathbf{1}$$

with $d\boldsymbol{W} \leftarrow d\boldsymbol{W} + \alpha \boldsymbol{W}$, finally we can update

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \epsilon d\boldsymbol{W} \text{ and } \boldsymbol{b} \leftarrow \boldsymbol{b} - \epsilon d\boldsymbol{b}$$

We just need to iterate thousand of time, to reach convergence...

Remark we can then get a classification, using

$$z_{i,j}^{(1)} = (\boldsymbol{x}_i^\top \boldsymbol{W}_{1j} + \boldsymbol{b}_{1j})_+ \text{ and } z_{i,j}^{(2)} = \boldsymbol{z}_i^\top \boldsymbol{W}_{2j} + \boldsymbol{b}_{2j}$$

and

$$p_{ij} = \frac{\exp[z_{i,j}^{(2)}]}{\exp[z_{i,1}^{(2)}] + \cdots + \exp[z_{i,4}^{(2)}]} \text{ and } y_i^\star = \underset{i \in \{1,2,3,4\}}{\text{argmax}} \left\{ p_{ij} \right\}$$

Classification & Non-linearities

```
nnet = function(X_train, X_valid, step_size = 0.5, reg
    = 0.001, h = 10, niteration){
  N_train = nrow(X_train)
  K = length(levels(Y_train))
  D = ncol(X_train)
  X_train = as.matrix(X_train)
  Y_train = as.matrix(model.matrix(lm(rep(1,N_train)
    ~0+Y_train)))
  W1 = 0.01 * matrix(rnorm(D*h), nrow = D)
  b1 = matrix(0, nrow = 1, ncol = h)
  W2 = 0.01 * matrix(rnorm(h*K), nrow = h)
  b2= matrix(0, nrow = 1, ncol = K)
  L_train = c()
    for (i in 0:niteration){ ..... }
  return(list(W1, b1, W2, b2, loss = cbind(0:
    niteration,L_train,L_valid)))
}
```

Classification & Non-linearities

```
hidden_layer_train = pmax(0, X_train%*% W1 + matrix(
   rep(b1,N_train), nrow = N_train, byrow = TRUE))
hidden_layer_train = matrix(hidden_layer_train, nrow
   = N_train)
scores_train = hidden_layer_train%*%W2 + matrix(rep(
   b2,N_train), nrow = N_train, byrow = TRUE)
exp_scores = exp(scores_train)
 probs_train = exp_scores / rowSums(exp_scores)
 correct_logprobs = -log(probs_train)
 data_loss = sum(correct_logprobs*Y_train)/N_train
 reg_loss = 0.5*reg*sum(W1*W1) + 0.5*reg*sum(W2*W2)
 loss_train = data_loss + reg_loss
 L_train = c(L_train, loss_train)
 dscores_train = (probs_train-Y_train)/N_train
```

Classification & Non-linearities

```
  dW2 = t(hidden_layer_train)%*%dscores_train
  db2 = colSums(dscores_train)
  dhidden =dscores_train%*%t(W2)
  dhidden[hidden_layer_train <= 0] <- 0
  dW1 = t(X_train)%*%dhidden
  db1 = colSums(dhidden)
  dW2 = dW2 + reg *W2
  dW1 = dW + reg *W1
  W1 = W1-step_size*dW1
  b1 = b1-step_size*db1
  W2 = W2-step_size*dW2
  b2 = b2-step_size*db2
```

We can now run that function

```
nnet.model = nnet(X_train = df_train[,c("x1","x2")],
    Y_train = df_train[,c("y")],step_size = 0.4,reg =
    0.002, h=50, niteration = 3000)
```

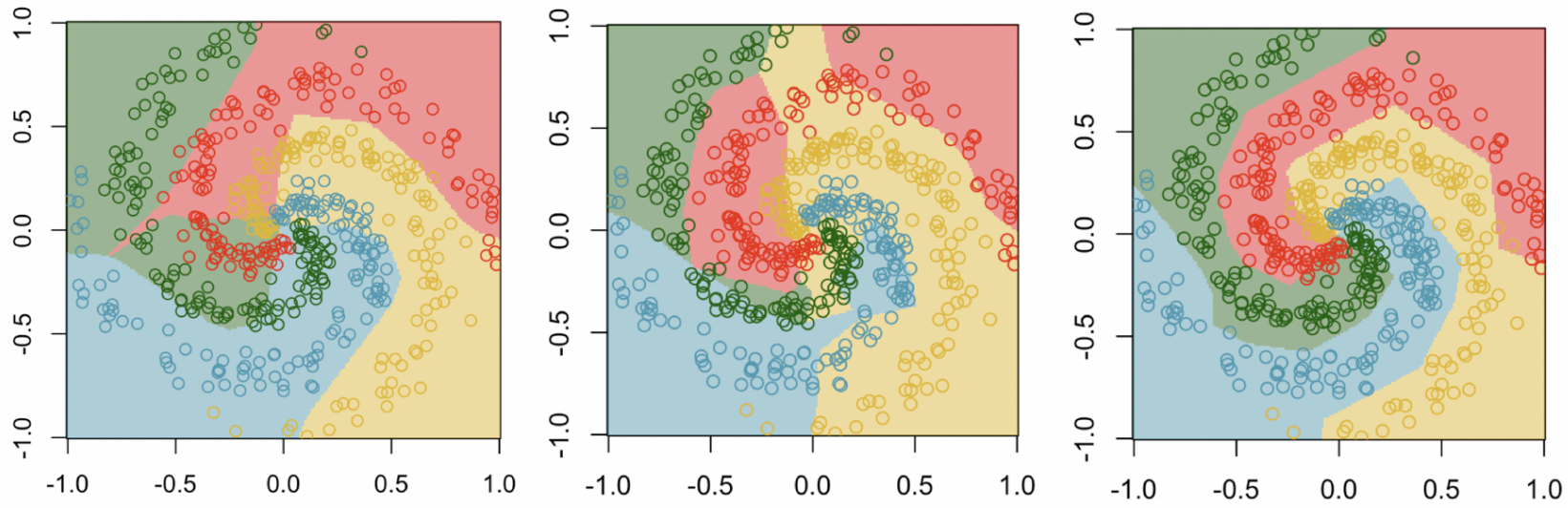
for predictions, use

Classification & Non-linearities

```
nnetPred = function(X, param = list()){
  W = param[[1]]
  b = param[[2]]
  W2 = param[[3]]
  b2 = param[[4]]
  N = nrow(X)
  hidden_layer = pmax(0, X%*% W + matrix(rep(b,N),
    nrow = N, byrow = T))
  hidden_layer = matrix(hidden_layer, nrow = N)
  scores = hidden_layer%*%W2 + matrix(rep(b2,N), nrow
    = N, byrow = T)
  predicted_class = apply(scores, 1, which.max)
  return(predicted_class)
}
```

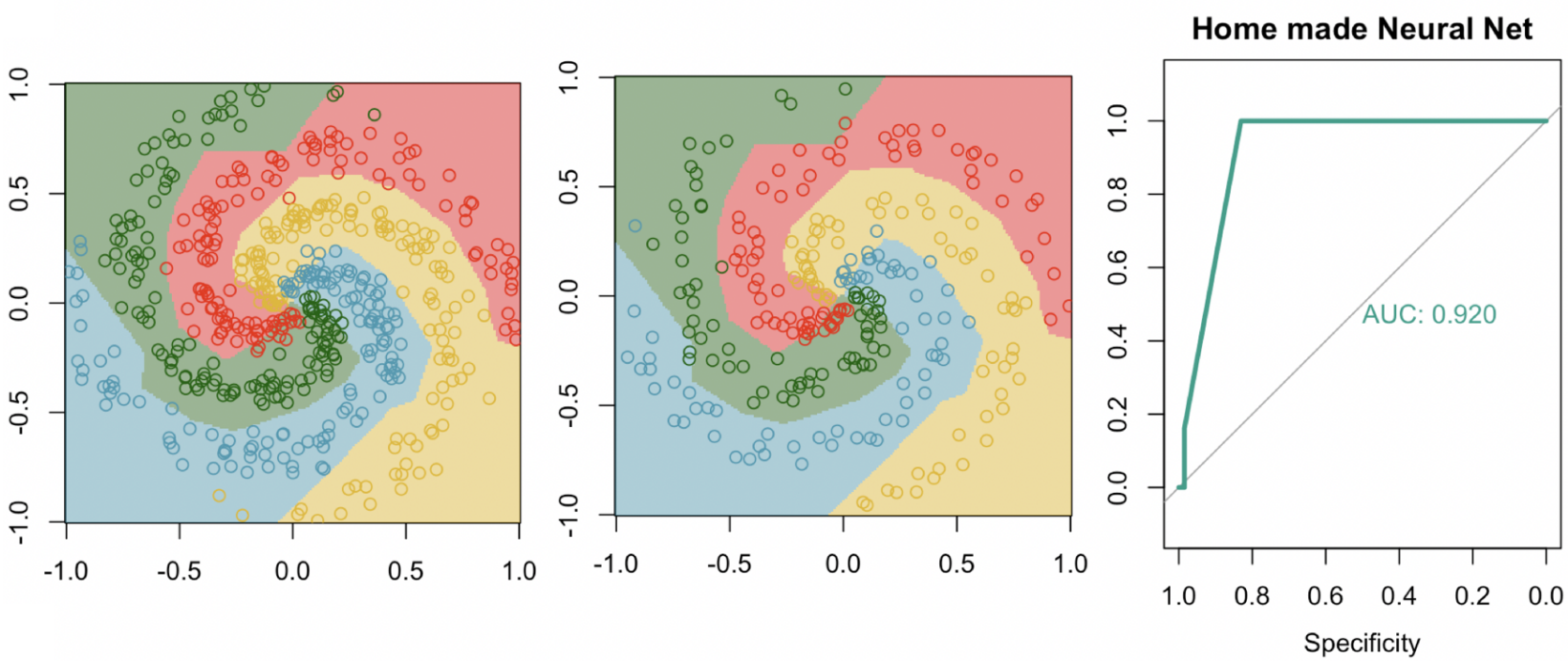
Classification & Non-linearities

Prediction on 1000, 2000 and 3000 iterations



Classification & Non-linearities

We can also compare the training dataset (left) and the validation dataset (center), and compute the multi-label ROC curve



Alzeihmer Detection

See Ewen Gallic's tutorial

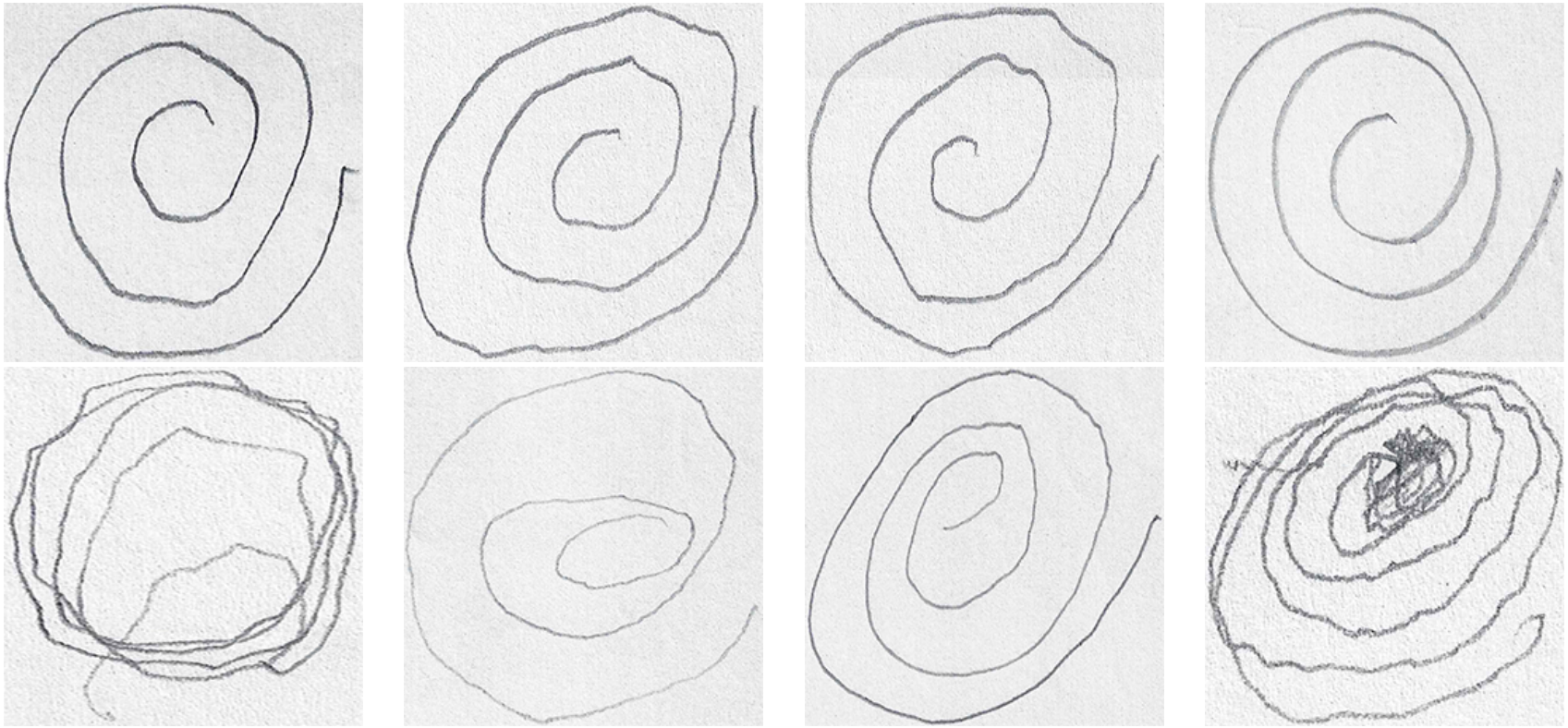
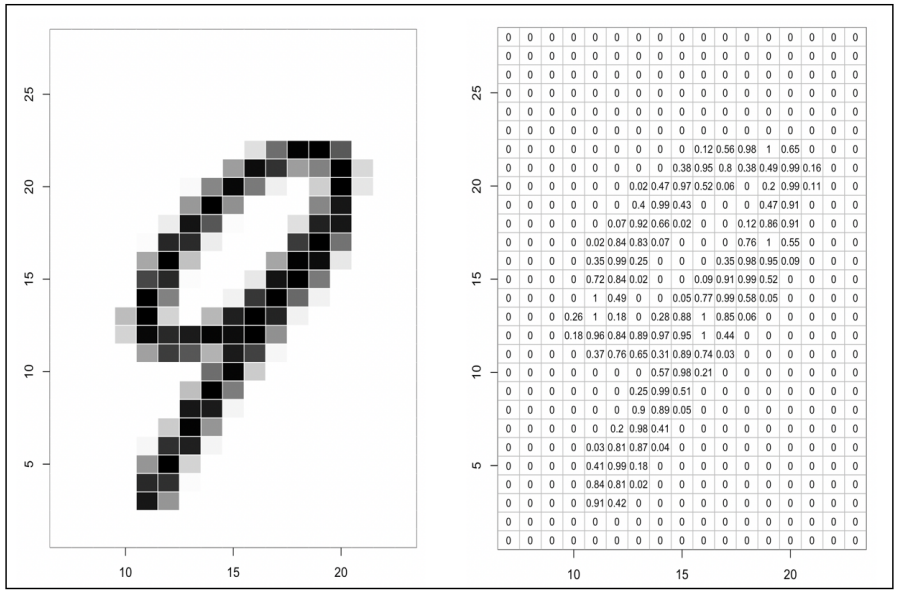


Image 'Recognition' with Neural Networks

Classical $\{0, 1, \cdots, 9\}$ handwritten digit recognition



$$\boxed{9}$$

$\boldsymbol{x}_i \in \mathcal{M}_{28,28}$ $\qquad$ $y_i \in \{0, 1, \cdots, 9\}$

see tensorflow (Google) and keras.

Reading handwritten digits

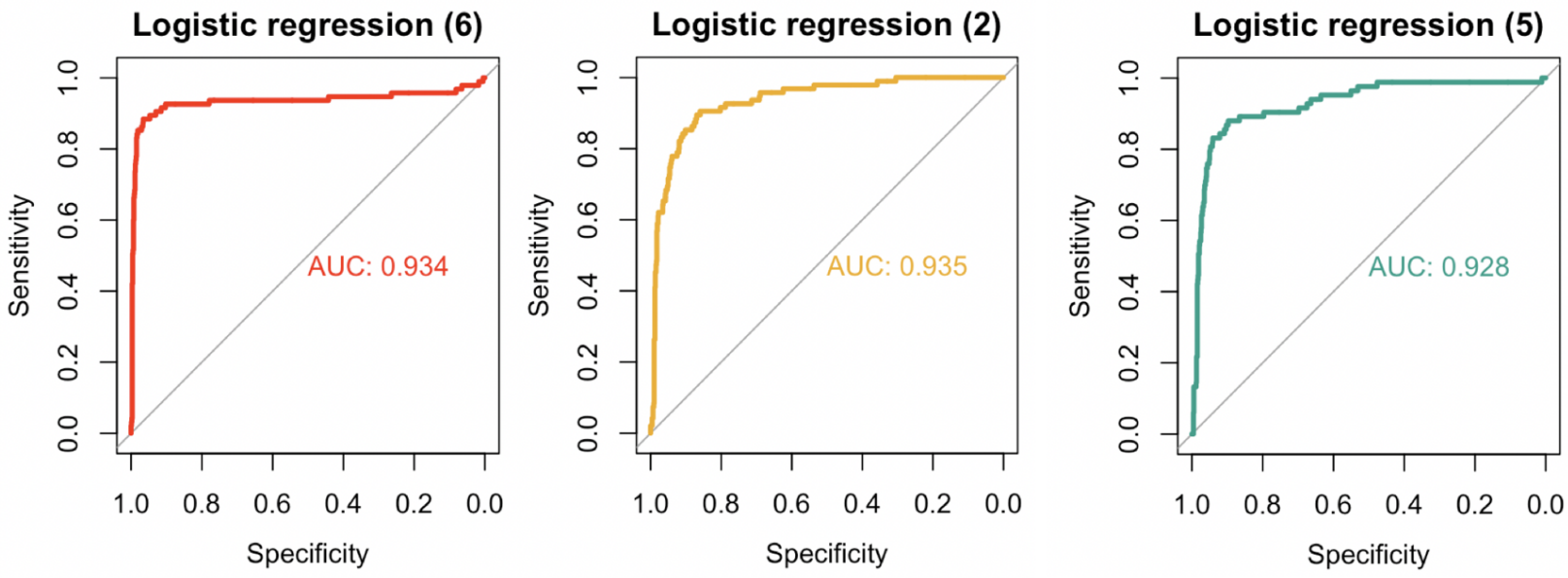
See https://www.tensorflow.org/install

```
library(keras)
mnist <- dataset_mnist()
V = mnist$train$x[1:2500,,]
Y_train=mnist$train$y[1:2500]
MV = NULL
for(i in 1:2500) MV=cbind(MV,as.vector(V[i,,]))
X_train=t(MV)
V = mnist$train$x[2550+1:1000,,]
Y_valid=mnist$train$y[2550+1:1000]
MV = NULL
for(i in 1:1000) MV=cbind(MV,as.vector(V[i,,]))
X_valid=t(MV)
df_train=data.frame(Y=Y_train,X_train)
df_valid=data.frame(Y=Y_valid,X_valid)
```

Reading handwritten digits

- logistic regression

```
reg = glm((Y==6)~., data=df_train)
yhat_logit = predict(reg,newdata=df_valid,type="
    response")
library(pROC)
roc = roc( (df_valid$Y==6), yhat_logit, plot=TRUE,
    print.auc=TRUE,)
```

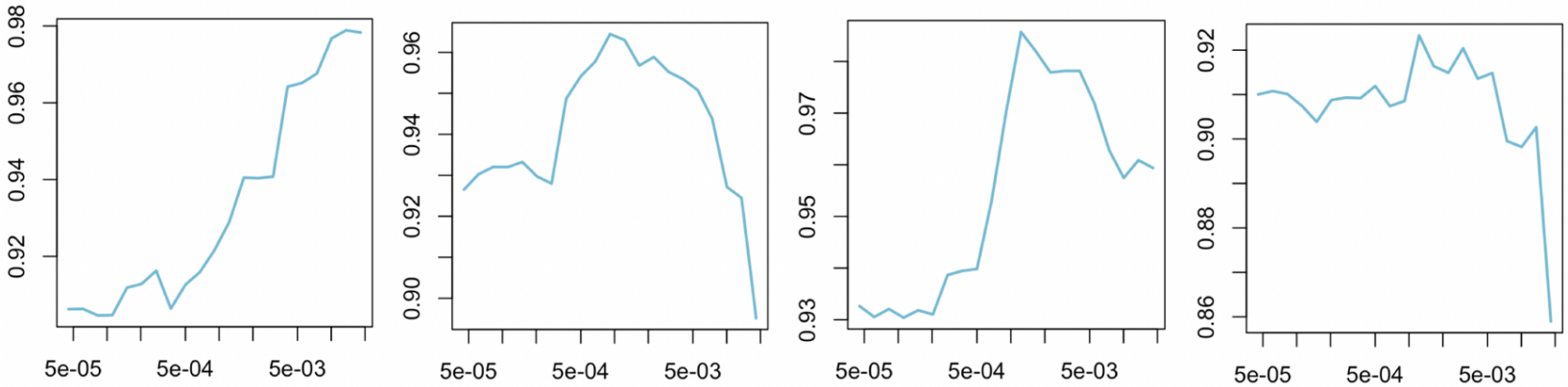


Reading handwritten digits

- logistic regression (with variable selection)

Which ones are the least well predicted

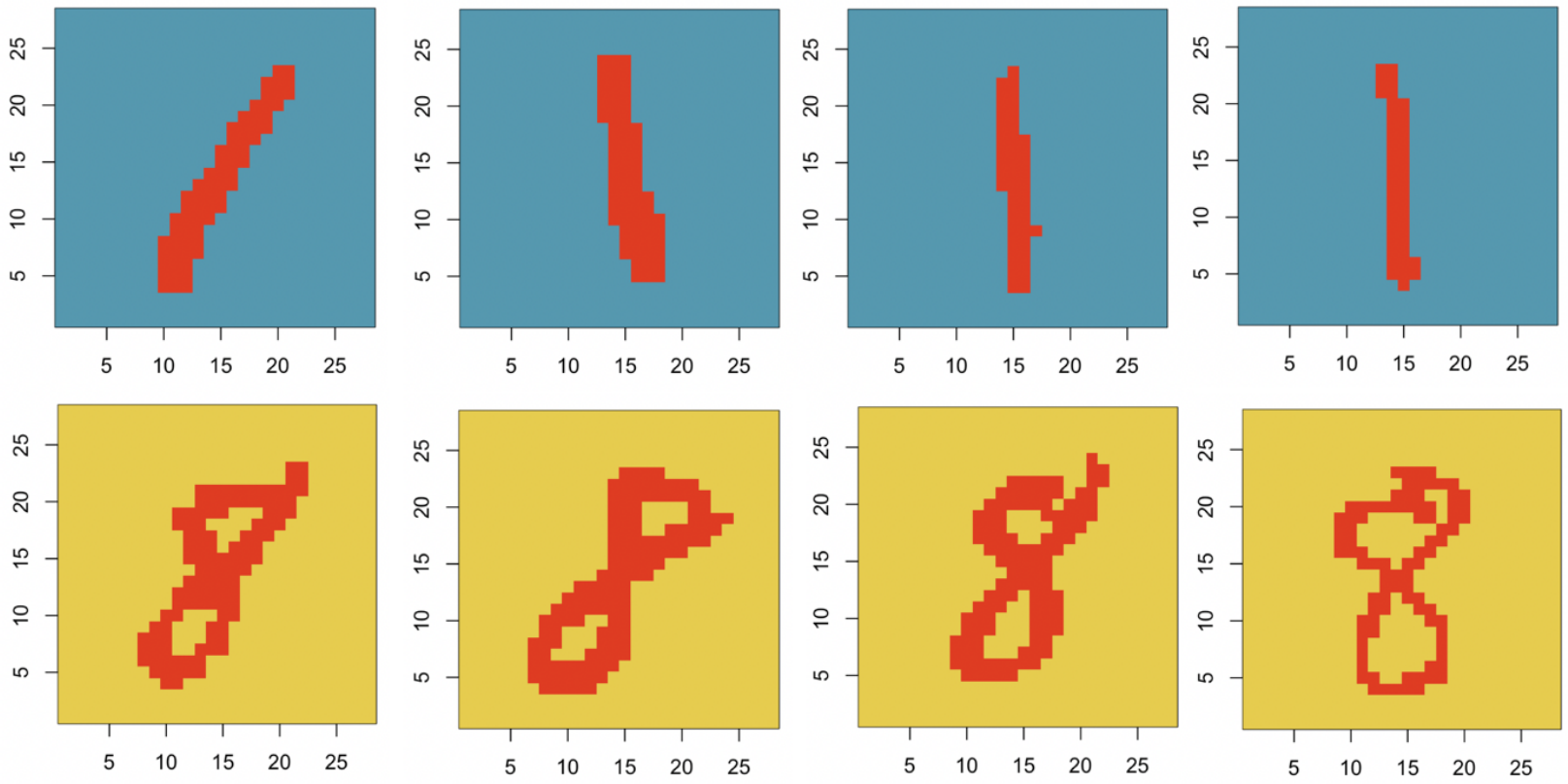
Here we consider a logistic regression on all 28×28 variables, including non-significant ones,
we can use LASSO for variable selection (here impact of λ on AUC,



Reading handwritten digits

- logistic regression (with variable selection)

Examples of 1's and 8's in the training dataset

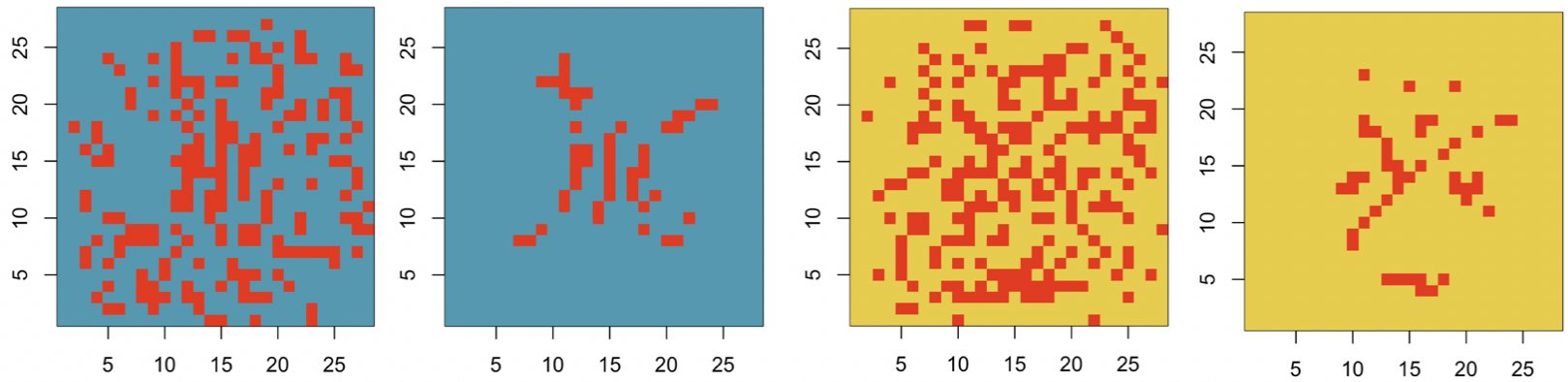


using LASSO with various λ's which $\widehat{\beta}_j$'s are non-null ?

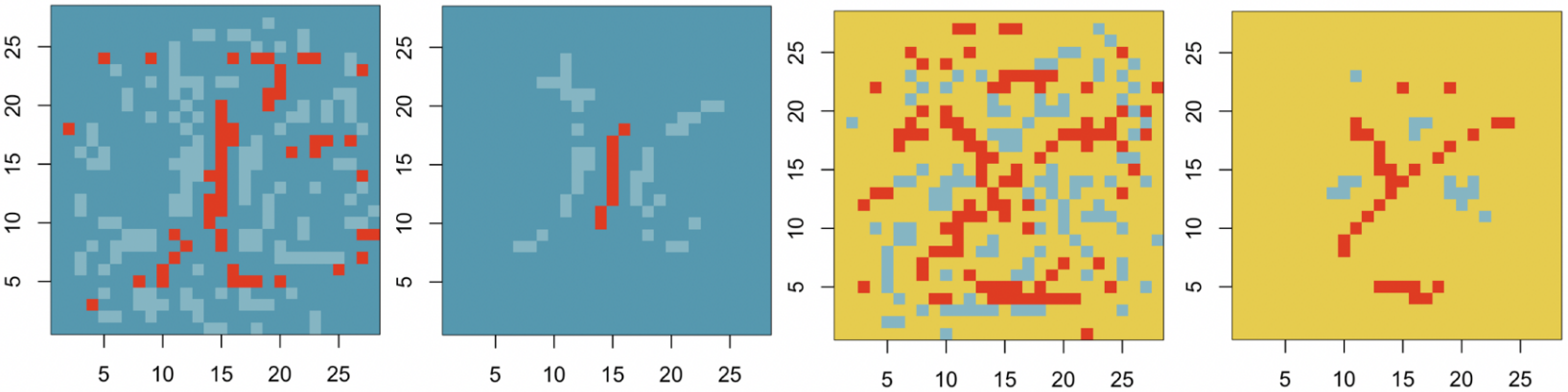
Reading handwritten digits

- logistic regression (with variable selection)

(red $= \widehat{\beta}_j \neq 0$)



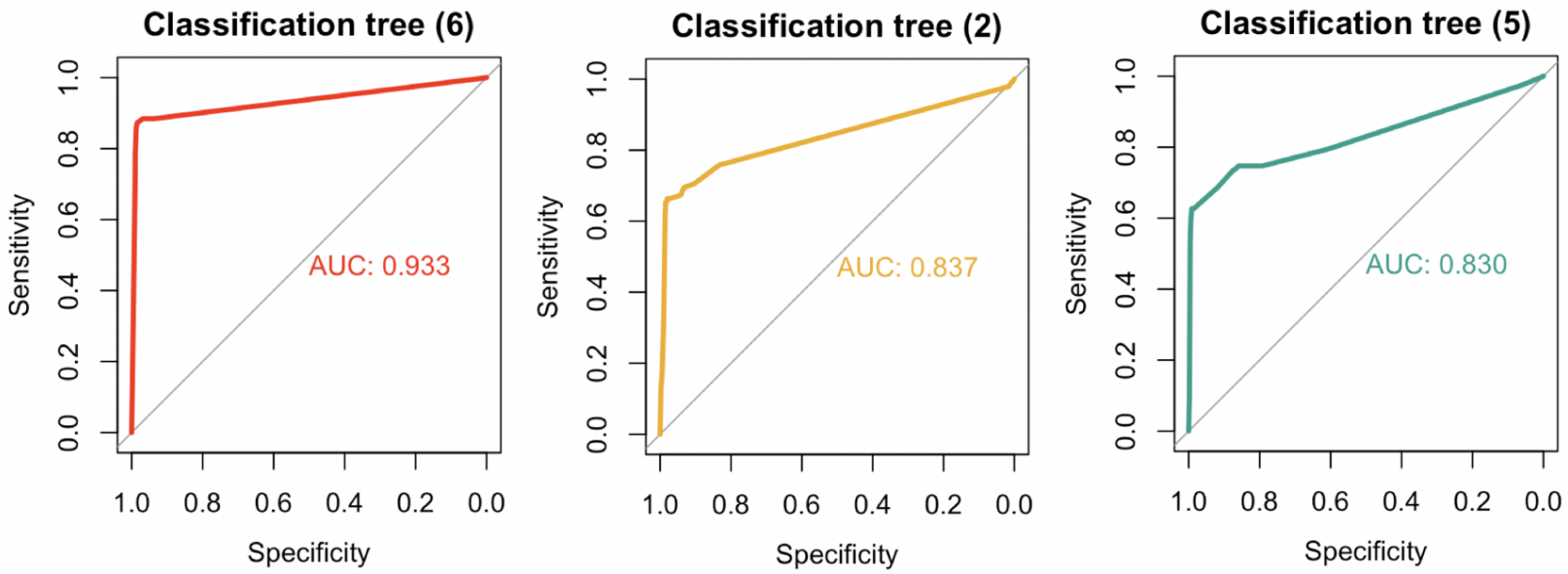
(red $= \widehat{\beta}_j > 0$ vs red $= \widehat{\beta}_j < 0$)



Reading handwritten digits

- classification tree

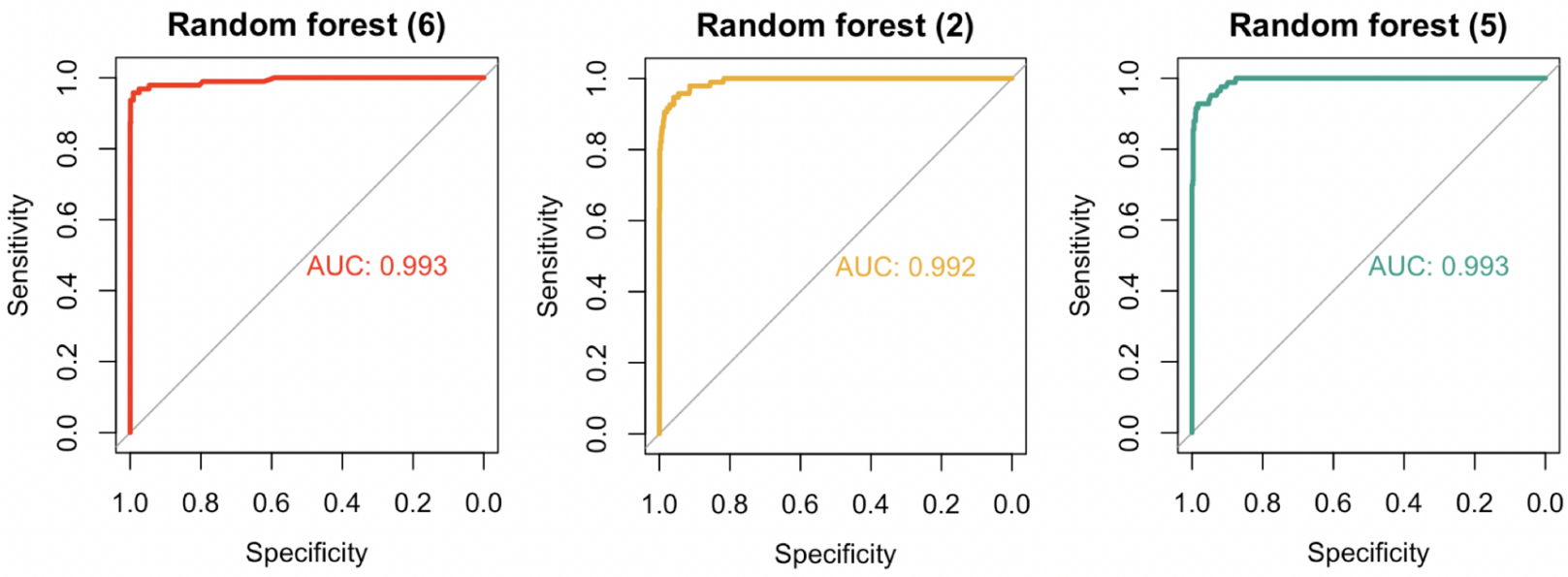
```
library(rpart)
cart = rpart(as.factor(Y==6)~., data=df_train)
yhat_cart = predict(cart,newdata=df_valid,type="prob")
    [,2]
roc = roc( (df_valid$Y==6), yhat_cart, plot=TRUE,
    print.auc=TRUE,)
```



Reading handwritten digits

- random forest

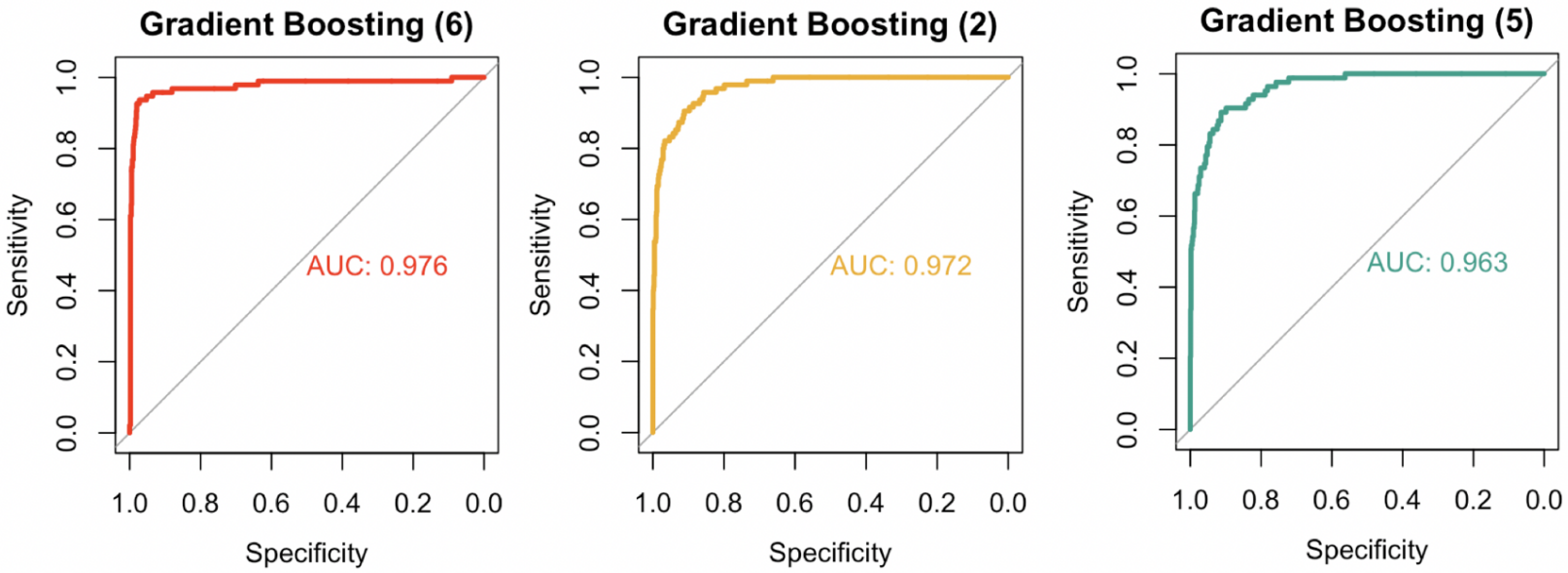
```
library(randomForest)
rf = randomForest(as.factor(Y==6)~., data=df_train,)
yhat_rf = predict(rf, newdata=df_valid,type="prob")[,
    2]
roc = roc( (df_valid$Y==6), yhat_rf, plot=TRUE,print.
    auc=TRUE,)
```



Reading handwritten digits

- gradient boosting

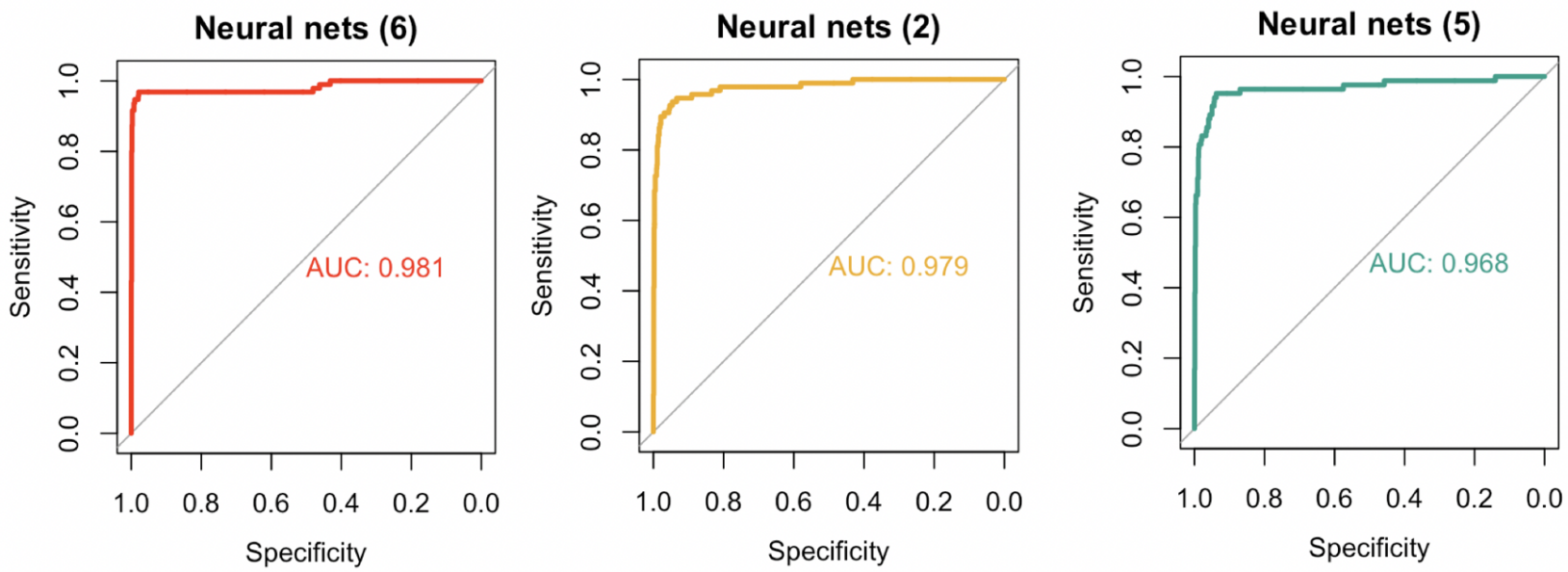
```
library(gbm)
gb = gbm((Y==6)~., data=df_train, distribution = "bernoulli")
yhat_gb = predict(gb, newdata=df_valid,type="response")
roc = roc( (df_valid$Y==6), yhat_gb, plot=TRUE, print.auc=TRUE,)
```



Reading handwritten digits

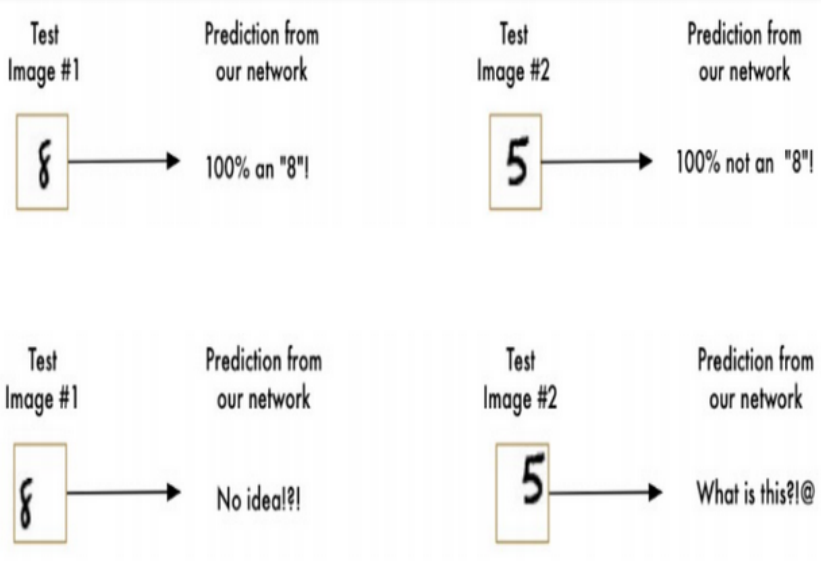
- neural nets

```
df_train_01 = df_train
df_train_01[,-1] = df_train_01[,-1]/255
require(caret)
nn = train(as.factor(Y==6)~., data=df_train_01, method
    ='nnet')
yhat_nn = predict(nn, df_valid/255, type='prob')[,2]
roc = roc( (df_valid$Y==6), yhat_logit, plot=TRUE)
```



Convolutional Network

FIGURE 11.9. *Examples of training cases from ZIP code data. Each image is a 16 × 16 8-bit grayscale representation of a handwritten digit.*

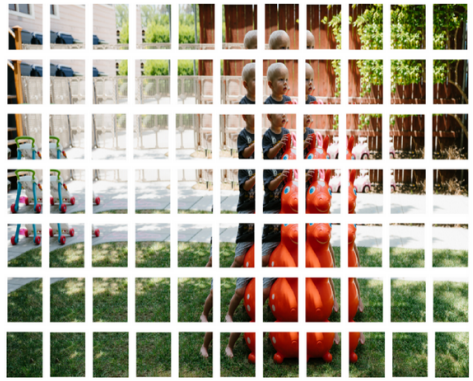


source: Hastie *et al.* (2009))

Neural Nets fails to recognize '8' when the digit is not centered
translation, scale and (small) rotation invariances are needed

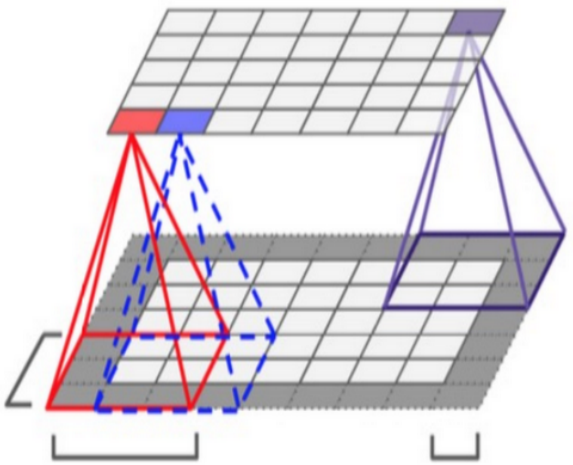
Convolutional Network

Break the image into overlapping image tiles and, feed each image tile into a small neural network with the same weights (and the same activation function)



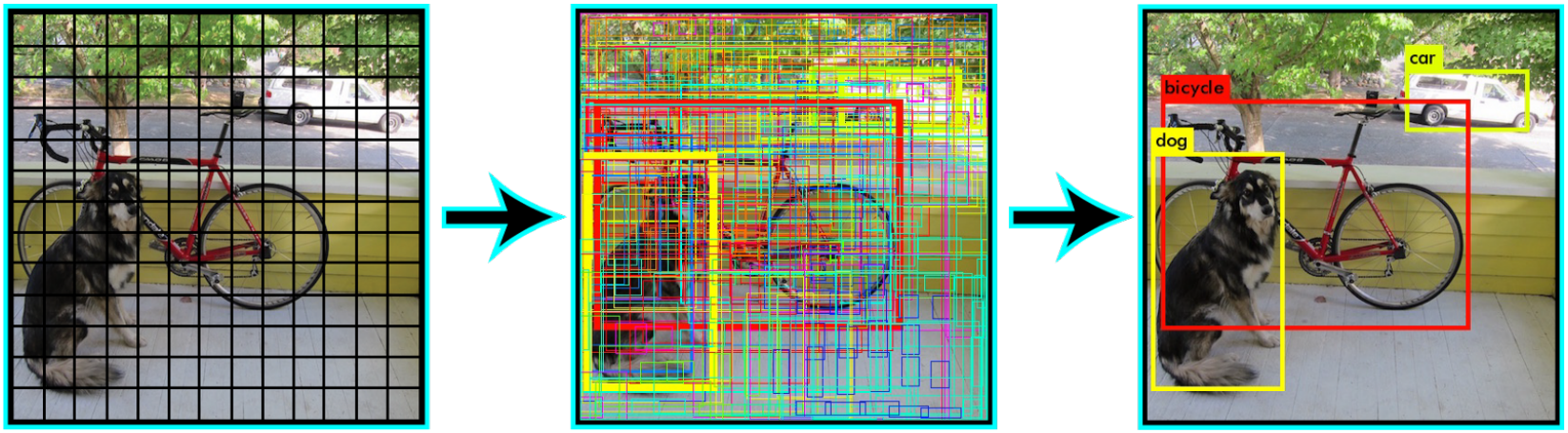
1. ConvNets exploit spatially local correlation: each neuron is locally- connected (to only a small region of the input volume)
2. Reduce the size of the array, using a pooling algorithm.

(pooling step reduces the dimensionality of each feature)



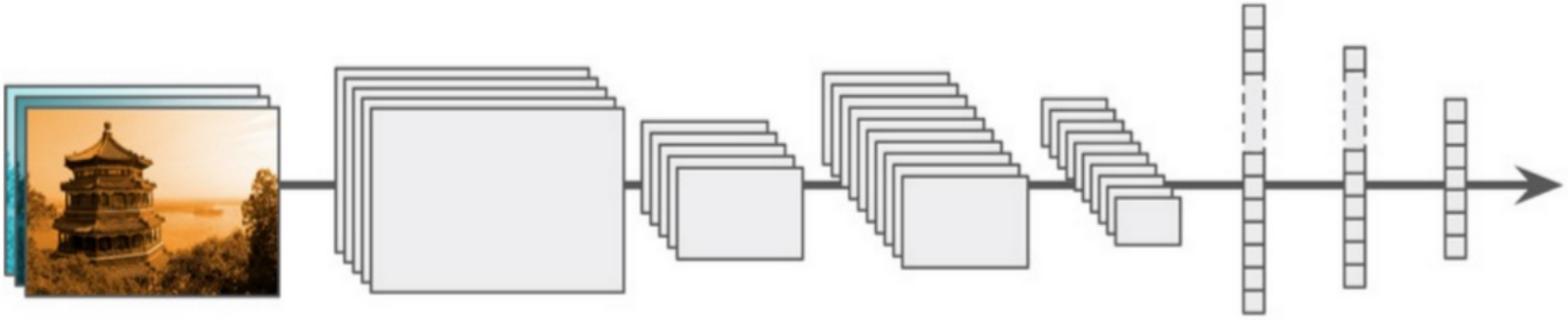
Convolutional Network

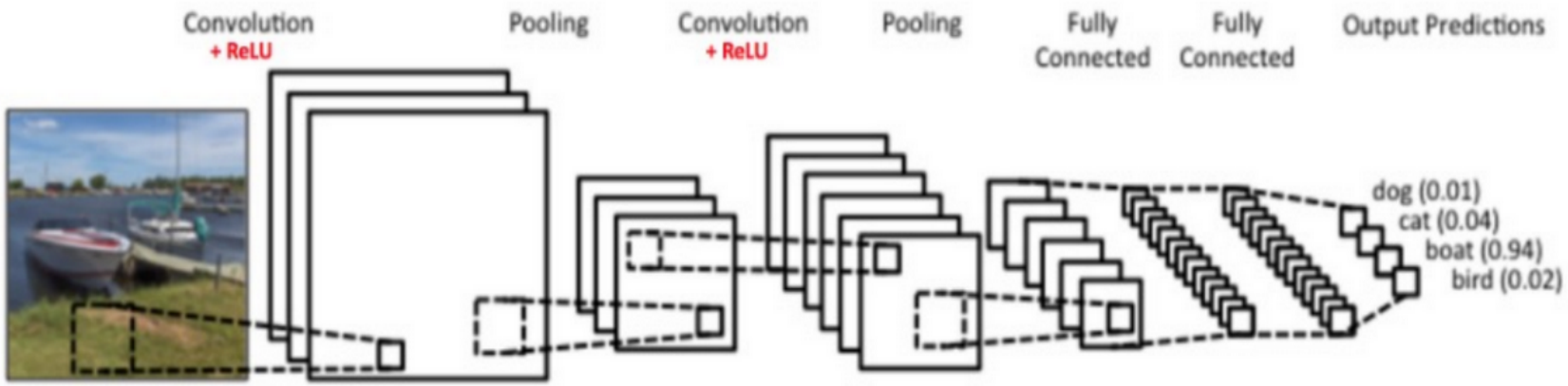
pooling:
- makes the input representations smaller and more manageable
- reduces the number of weights and links in the network, therefore, controlling overfitting
- makes the network invariant to small transformations, distortions and translations in the input image
- helps us arrive at an almost scale invariant representation of our image



(via yolo for objects detection)

Convolutional Network





Recurrent Networks

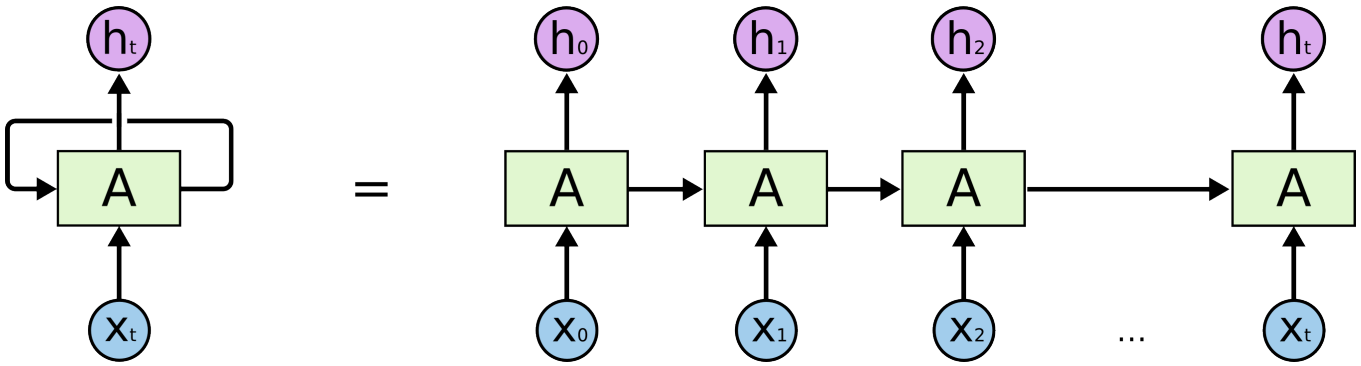
Recurrent neural networks are networks with loops, allowing information to persist. With a classical neural net

$$y_i = m(\boldsymbol{x}_i)$$

while, with a recurrent neural net

$$y_t = m(\boldsymbol{x}_t, y_{t-1}) = m(\boldsymbol{x}_t, m(\boldsymbol{x}_{t-1}, y_{t-2})) = \cdots$$

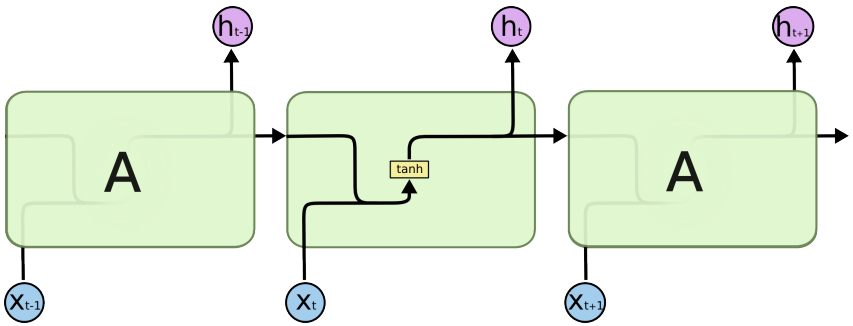
A is the neural net, h is the output (y) and x some covariates.



(source https://colah.github.io/)
See Sutskever (2017, Training Reccurent Neural Networks)

Recurrent Networks

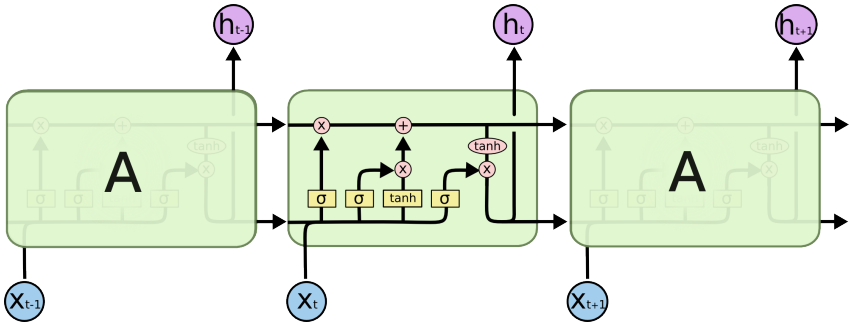
A classical RNN (with a single layer) would be



(source https://colah.github.io/)

"*In theory, RNNs are absolutely capable of handling such 'long-term dependencies'. A human could carefully pick parameters for them to solve toy problems of this form. Sadly, in practice, RNNs don't seem to be able to learn them*" see Benghio *et al.* (1994))

Recurrent Networks



"*RNNs can keep track of arbitrary long-term dependencies in the input sequences. The problem of "vanilla RNNs" is computational (or practical) in nature: when training a vanilla RNN using back-propagation, the gradients which are back-propagated can "vanish" (that is, they can tend to zero) "explode" (that is, they can tend to infinity), because of the computations involved in the process*" (from wikipedia)