Clustering & Classification using ANNs

Julien Arino
University of Manitoba
julien.arino@umanitoba.ca

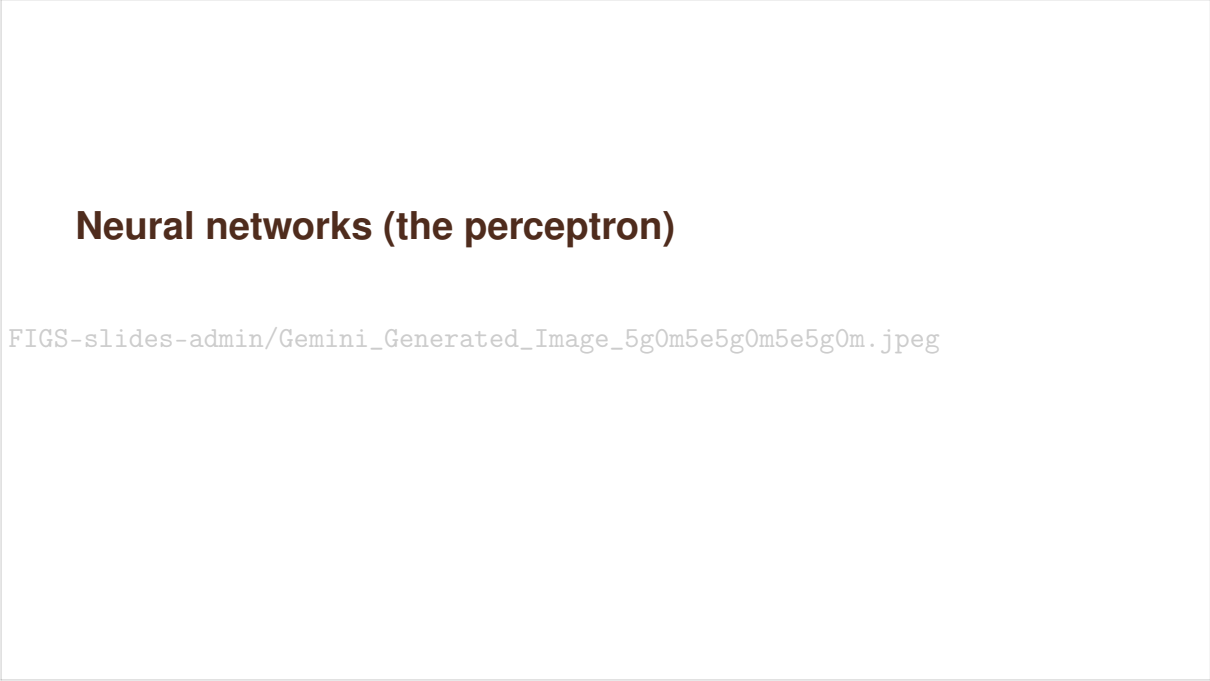
The University of Manitoba campuses are located on original lands of Anishinaabeg, Ininew, Anisininew, Dakota and Dene peoples, and on the National Homeland of the Red River Métis. We respect the Treaties that were made on these territories, we acknowledge the harms and mistakes of the past, and we dedicate ourselves to move forward in partnership with Indigenous communities in a spirit of Reconciliation and collaboration.

Outline

Neural networks (the perceptron)

FIGS-slides-admin/Gemini_Generated_Image_yegaxcyegaxcyega.jpeg

Neural networks (the perceptron)



Artificial neural network (ANN) - from Wikipedia

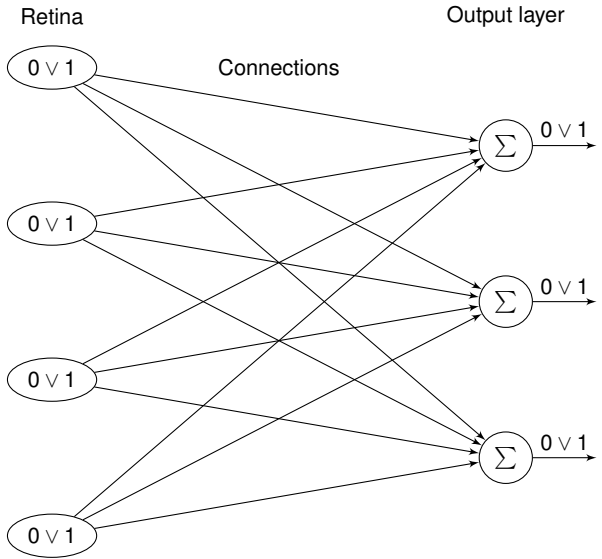
Artificial neural networks *(ANNs) are computing systems inspired by the biological neural networks that constitute animal brains*
An ANN is based on a collection of connected units or nodes called ***artificial neurons****, which loosely model the neurons in a biological brain. Each connection, like the synapses in a biological brain, can transmit a signal to other neurons. An artificial neuron receives signals then processes them and can signal neurons connected to it. The "signal" at a connection is a real number, and the output of each neuron is computed by some non-linear function of the sum of its inputs. The connections are called edges. Neurons and edges typically have a weight that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection. Neurons may have a threshold such that a signal is sent only if the aggregate signal crosses that threshold.*

The perceptron

One of the first neural networks (invented 1943, implemented 1957), made for simple classification tasks, for example recognising letters or numbers

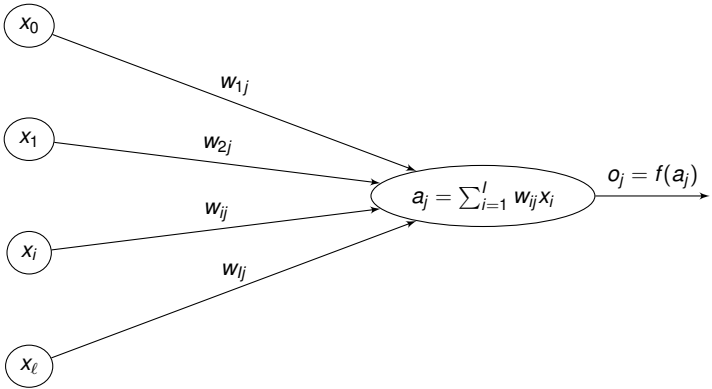
Two layers: the *input* layer (the *retina*) and the *output* layer

Inputs are 0 or 1, so are outputs



The connections into the output layer are called *synapses*, they are modifiable

The activation function



Here,

$$f(a_j) = \begin{cases} 0 & \text{if } a_j \leq 0 \\ 1 & \text{if } a_j > 0 \end{cases}$$

The activation function

We have I input neurons taking values 0 or 1, O output neurons taking values 0 or 1, weights $W = [w_{ij}] \in \mathcal{M}_{IO}$ and a threshold function f

More generally, use a threshold θ_j for each output neuron

$$o_j = \begin{cases} 0 & \text{if } a_j \leq \theta_j \\ 1 & \text{if } a_j > \theta_j \end{cases}$$

The thresholds (or *response bias*) and the weights are modifiable by learning. To do that easily for the threshold, consider an input neuron that is always on, say neuron 0, and set weights $w_{0j} = -\theta_j$, making the weights matrix an $(I + 1) \times O$-matrix

Another way to write the activation is

$$o_j = \begin{cases} 0 & \text{if } a_j + w_{0j} \leq 0 \\ 1 & \text{if } a_j + w_{0j} > 0 \end{cases}$$

where $w_{0j} = -\theta_j$

Learning something simple

The aim is to adjust the synaptic weights so that the proper response is provided to a given stimulus

Let us first do a simple example: the OR truth table

$$\begin{array}{cccc} 0 & 0 & \mapsto & 0 \\ 1 & 0 & \mapsto & 1 \\ 0 & 1 & \mapsto & 1 \\ 1 & 1 & \mapsto & 1 \end{array}$$

So we have two neurons in the retina and a single output neuron

Supervised learning

(From R. Rojas)

Supervised learning*: method in which some input vectors are collected and presented to the network. The output computed by the network is observed and the deviation from the expected answer is measured*

The weights are corrected according to the magnitude of the error in the way defined by the learning algorithm

Also called learning with a teacher, since a control process knows the correct answer for the set of selected input vectors

Further distinctions in supervised learning methods

Methods with *reinforcement* or *error correction*

- **Reinforcement learning**: used when after each presentation of an input-output example, we only know whether the network produces the desired result or not. Weights are updated based on this information (i.e., the Boolean values true or false), so only the input vector can be used for weight correction

- In learning with **error correction**, the magnitude of the error, together with the input vector, determines the magnitude of the corrections to the weights. In many cases, we try to eliminate the error in a single correction step

A first learning algorithm

Suppose the training set consists of two sets of points P and N

- **start:** Generate random weight vector w_0; set $t := 0$
- **test:** A vector $x \in P \cup N$ is selected randomly
  - if $x \in P$ and $\langle w_t, x \rangle > 0$ go to **test**
  - if $x \in P$ and $\langle w_t, x \rangle \leq 0$ go to **add**
  - if $x \in N$ and $\langle w_t, x \rangle < 0$ go to **test**
  - if $x \in N$ and $\langle w_t, x \rangle \geq 0$ go to **subtract**
- **add:** set $w_{t+1} = w_t + x$ and $t := t + 1$, goto **test**
- **subtract:** set $w_{t+1} = w_t - x$ and $t := t + 1$, goto **test**

Widrow-Hoff learning rule

Need to provide the correct answer, i.e., this is a supervised learning rule

An output cell only learns if it is mistaken

Present random inputs and apply the rule if the output does not match the known output

Widrow-Hoff learning rule

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} + \eta(t_j - o_j)x_j = w_{ij}^{(t)} + \Delta w_{ij} \quad (1)$$

with

- Δw_{ij} correction to add to the weight w_{ij}
- x_i: value (0 or 1) of the ith retinal cell
- o_j: response of the jth output cell
- t_j target response (correct desired response)
- $w_{ij}^{(t)}$: weight of the synapse between the ith retinal cell and jth output cell at time t. Typically initiated at small random values
- η: small positive constant, the *learning constant*

Learning OR

$$\begin{array}{cccc} 0 & 0 & \mapsto & 0 \\ 1 & 0 & \mapsto & 1 \\ 0 & 1 & \mapsto & 1 \\ 1 & 1 & \mapsto & 1 \end{array}$$

Three cells in the retina (two inputs and the “dummy” cell used for the threshold) and one output cell. So inputs and outputs must be

$$\begin{array}{ccccc} 1 & 0 & 0 & \mapsto & 0 \\ 1 & 1 & 0 & \mapsto & 1 \\ 1 & 0 & 1 & \mapsto & 1 \\ 1 & 1 & 1 & \mapsto & 1 \end{array}$$

Initialise the 3×1 weight matrix W to zero:

$$W = \begin{pmatrix} w_0 = -\theta \\ w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

Procedure

We choose one random association in

$$\begin{array}{cccc} 1 & 0 & 0 & \mapsto & 0 \\ 1 & 1 & 0 & \mapsto & 1 \\ 1 & 0 & 1 & \mapsto & 1 \\ 1 & 1 & 1 & \mapsto & 1 \end{array}$$

say, the fourth one. So we present $[1, 1, 1]$ and expect an output of 1. We have

$$a = \sum_i w_i x_i = (1 \times 0) + (1 \times 0) + (1 \times 0) = 0$$

This being ≤ 0 means that $o = 0$, giving an error of 1

Applying the rule

Suppose the learning constant $\eta = 0.1$. Then applying (1),

$$\Delta w_0 = \eta(t - o)x_0 = 0.1 \times (1 - 0) \times 1 = 0.1$$
$$\Delta w_1 = \eta(t - o)x_0 = 0.1 \times (1 - 0) \times 1 = 0.1$$
$$\Delta w_2 = \eta(t - o)x_0 = 0.1 \times (1 - 0) \times 1 = 0.1$$

Applying the correction, W becomes

$$W = \begin{pmatrix} 0.1 \\ 0.1 \\ 0.1 \end{pmatrix}$$

Trying another input

Suppose we now present the first input $[1, 0, 0]$. This should produce a result of 0. Then

$$a = \sum_i w_i x_i = (1 \times 0.1) + (0 \times 0.1) + (0 \times 0.1) = 0.1$$

which is > 0, so $o = 1$. We compute the correction

$$\begin{aligned}
\Delta w_0 &= \eta(t - o)x_0 = 0.1 \times (0 - 1) \times 1 = -0.1 \\
\Delta w_1 &= \eta(t - o)x_1 = 0.1 \times (0 - 1) \times 0 = 0 \\
\Delta w_1 &= \eta(t - o)x_2 = 0.1 \times (0 - 1) \times 0 = 0
\end{aligned}$$

and adjust the weights, giving

$$W = \begin{pmatrix} 0 \\ 0.1 \\ 0.1 \end{pmatrix}$$

And we are done!

With the weights

$$W = \begin{pmatrix} 0 \\ 0.1 \\ 0.1 \end{pmatrix}$$

we are done. Indeed

Input 0	Input 1	Input 2	a	o	Should be
1	0	0	0	0	0
1	1	0	0+0.1+0	1	1
1	0	1	0+0+0.1	1	1
1	1	1	0+0.1+0.1	1	1

Learning XOR

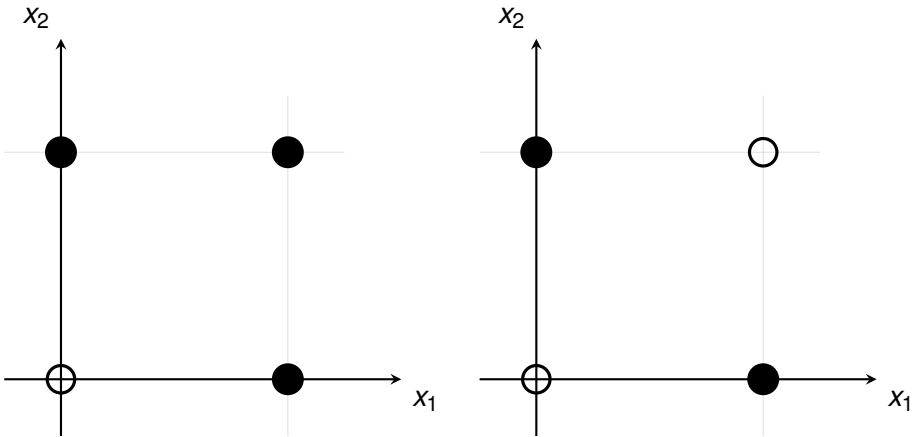
Let us now look at the XOR truth table

$$\begin{array}{cccc} 0 & 0 & \mapsto & 0 \\ 1 & 0 & \mapsto & 1 \\ 0 & 1 & \mapsto & 1 \\ 1 & 1 & \mapsto & 0 \end{array}$$

This problem is not solvable with a simple perceptron of the type we just used, as truth table is not *linearly separable*

Indeed, we would get weights $w_1 > 0$, $w_2 > 0$ to activate when presenting $[1, 0]$ and $[0, 1]$, but would require that the sum of the weights when applied to the input $[1, 1]$, give a negative value.

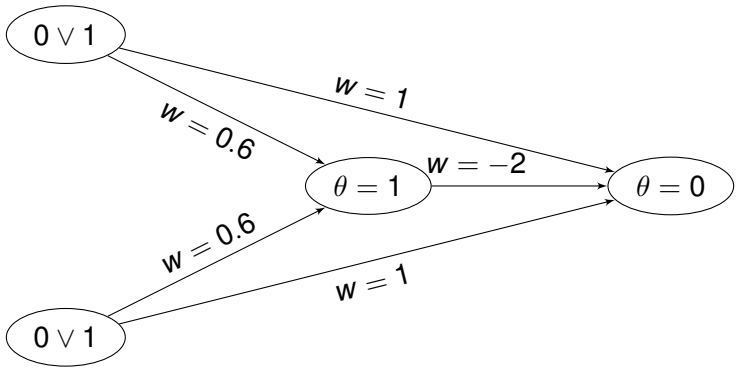
Linear separability and OR and XOR



A single-layer perceptron can only learn linearly separable problems

Adding a hidden layer

It is possible to do XOR, but we need to add a **hidden layer**



Using `neuralnet` to learn OR

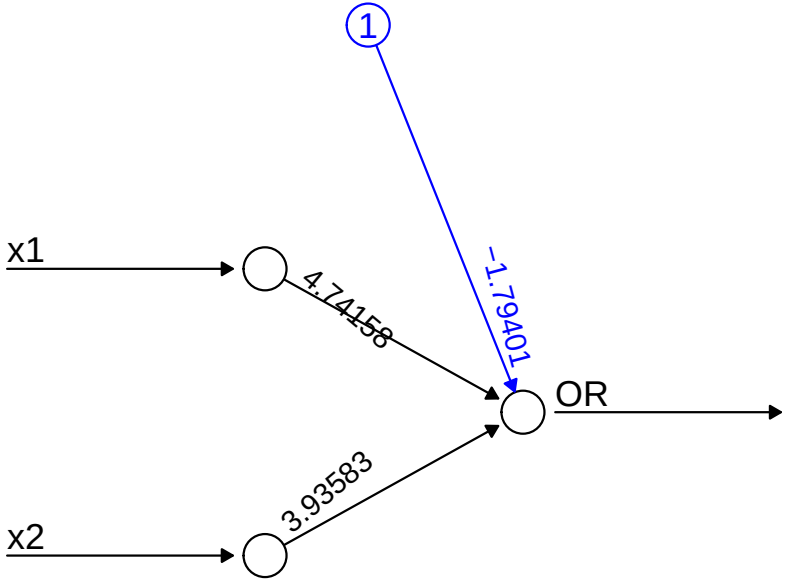
First, create the truth table

```
OR_table = matrix(c(0, 0, 0,
                    1, 0, 1,
                    0, 1, 1,
                    1, 1, 1),
                  nc = 3, byrow = TRUE)
OR_table = as.data.frame(OR_table)
colnames(OR_table) = c("x1", "x2", "OR")
```

Now create and train the NN

The “formula” is to find the OR column using the x1 and x2 columns. We use no hidden layer

```
nn_OR = neuralnet(OR ~ x1 + x2,
                  data = OR_table,
                  act.fct = "logistic",
                  hidden = 0,
                  linear.output = FALSE)
# Plot the result
plot(nn_OR, rep = "best")
```



Error: 0.016931 Steps: 56

Testing the result

```
pred = predict(nn_OR, OR_table)
OR_table$result = pred > 0.5
kable(OR_table, "latex", booktabs = TRUE)
```

x1	x2	OR	result
0	0	0	FALSE
1	0	1	TRUE
0	1	1	TRUE
1	1	1	TRUE

Now the XOR truth table

```
XOR_table = matrix(c(0, 0, 0,
                     1, 0, 1,
                     0, 1, 1,
                     1, 1, 0),
                   nc = 3, byrow = TRUE)
XOR_table = as.data.frame(XOR_table)
colnames(XOR_table) = c("x1", "x2", "XOR")
```

Try to learn it without a hidden layer

```
nn_XOR = neuralnet(XOR ~ x1 + x2,
                   data = XOR_table,
                   act.fct = "logistic",
                   hidden = 0,
                   linear.output = FALSE)
pred = predict(nn_XOR, XOR_table)
XOR_table$result = pred > 0.5
kable(XOR_table, "latex", booktabs = TRUE)
```

x1	x2	XOR	result
0	0	0	FALSE
1	0	1	FALSE
0	1	1	TRUE
1	1	0	TRUE

Now with a hidden layer

```
nn_XOR = neuralnet(XOR ~ x1 + x2,
                   data = XOR_table,
                   act.fct = "tanh",
                   hidden = 1)
pred = predict(nn_XOR, XOR_table)
XOR_table$result = pred > 0.5
kable(XOR_table, "latex", booktabs = TRUE)
```

x1	x2	XOR	result
0	0	0	FALSE
1	0	1	TRUE
0	1	1	TRUE
1	1	0	TRUE

Should look into options.. :)

An example from the `neuralnet` manual – Training vs testing sets

`iris` is a built-in `R` dataset detailing physical characteristics of 150 flowers from 3 iris species

```
train_idx <- sample(nrow(iris), 2/3 * nrow(iris))
iris_train <- iris[train_idx, ]
iris_test <- iris[-train_idx, ]
```

Thus we pick at random 2/3 of the data for training and 1/3 for testing. See some considerations on training, validation and testing on this Wikipedia page

An example from the `neuralnet` manual

```
nn <- neuralnet(Species == "setosa" ~ Petal.Length + Petal.Width,
                iris_train, linear.output = FALSE)
pred <- predict(nn, iris_test)
table(iris_test$Species == "setosa", pred[, 1] > 0.5)

##
##         FALSE TRUE
##   FALSE    32    0
##   TRUE      0   18
```

Another example – multiclass classification

```
nn <- neuralnet((Species == "setosa") +
                  (Species == "versicolor") +
                  (Species == "virginica")
                ~ Petal.Length + Petal.Width,
               iris_train, linear.output = FALSE)
pred <- predict(nn, iris_test)
table(iris_test$Species, apply(pred, 1, which.max))

##
##                1  2  3
##   setosa      18  0  0
##   versicolor   0 13  1
##   virginica    0  1 17
```