uc3m | Universidad Carlos III de Madrid

An introduction to Neural Networks (NNs)

March 27, 2020

Jorge Martín Pérez
jmartinp@it.uc3m.es

When we all think of NNs we have in mind this (maybe with a higher resolution image):

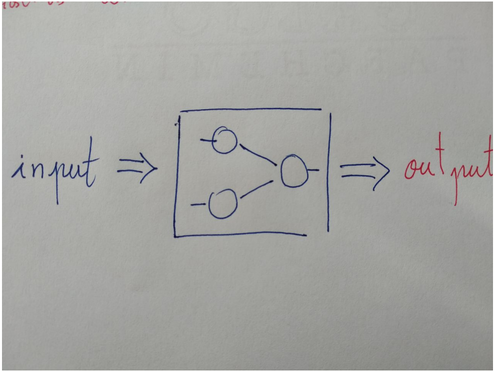

Figure 1:

What people imagine when “data-scientists” say they train a NN:

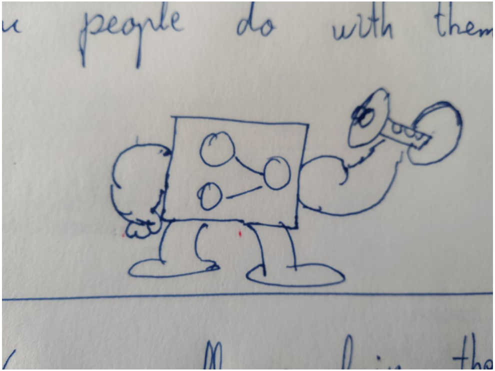

Figure 2: trained-NN.jpg

In these slides we'll try to bring some light to Figures 1,2; i.e.:

- explain **how a NN works**; and
- how to **train** a NN.

So lets start talking about input x, and output y. Since highschool we learn to write it as:

$$y = f(x)$$

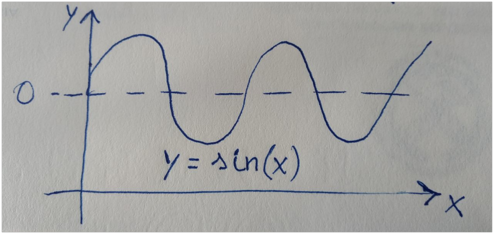

Figure 3:

Traditionally there has been different approaches to try to guess $f(x)$ given the input x:

- splines (numerical calculus);
- regression (statistics); and
- many others[1].

[1] the nicest way to say that no more come to my mind

But when data patterns present non-linearities, things become difficult.

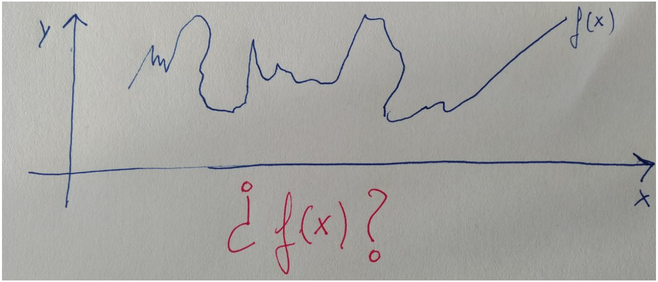

Figure 4: Try to guess this $f(x)$!

So is in situations like Figure 4 where NNs are quite useful.

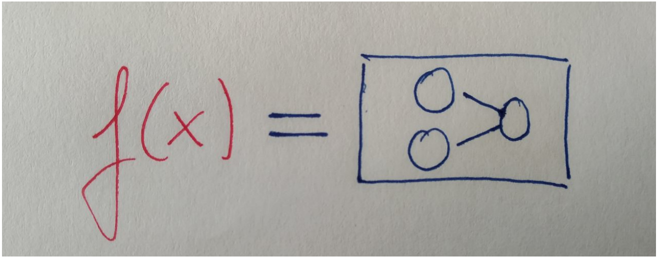

Figure 5: The right box is a NN.

The balls inside the box of Figure 7 are neurons, and probably you already knew that.

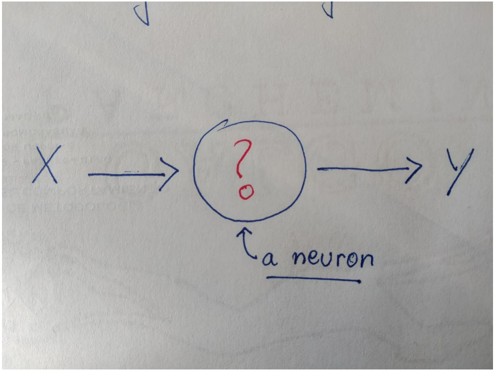

Figure 6: Basically, a neuron takes a number x, and outputs another number y.

A neuron will produce output y using a function. A commonly used neuron, is the Rectifier Linear Unit (ReLU[2]):

$$y = f(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

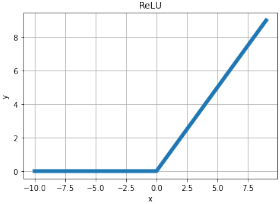

Figure 7: $ReLU(x)$

[2]It has nice properties for learning ▸ Wikipedia

Ok, but a silly ReLU can't recognize my face, as some NNs do

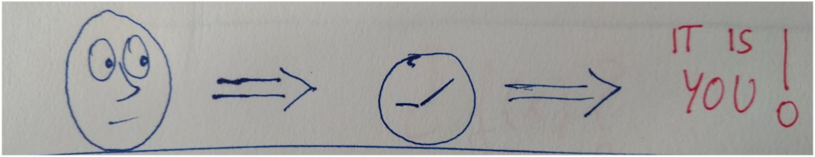

Figure 8:

Right, one neuron cannot. But many of them can, and this is when **weights** come to the game!

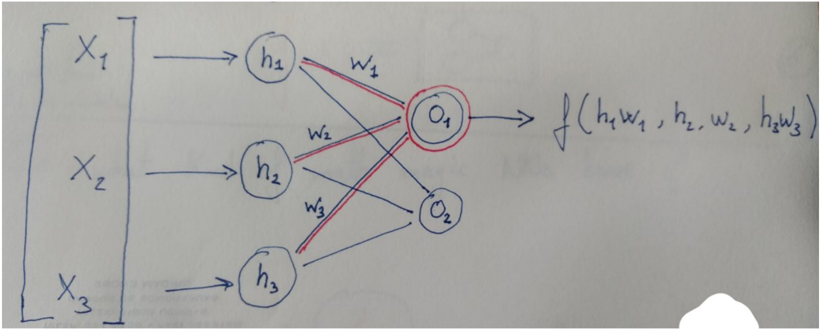

Figure 9: x input vector, h_i hidden neurons, o_i output neurons; and $f(\cdot) = o_i(\cdot)$

An example of a function for a given neuron output would be:

$$f(h_1 w_1, h_2 w_2, h_3 w_3) = ReLU(h_1 w_1 + h_2 w_2 + h_3 w_3)$$

Nice, so basically a NN takes an input vector and yields out an output vector!

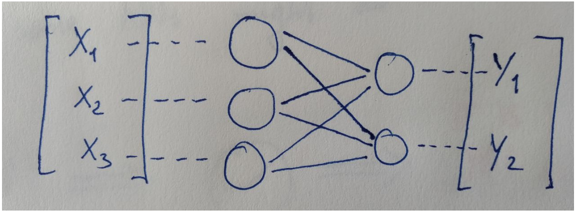

Figure 10:

Yes, plus the NN does quite a **good at capturing non-linearities** (remember Figure 4)?

And how can a NN learn?

And how can a NN learn?

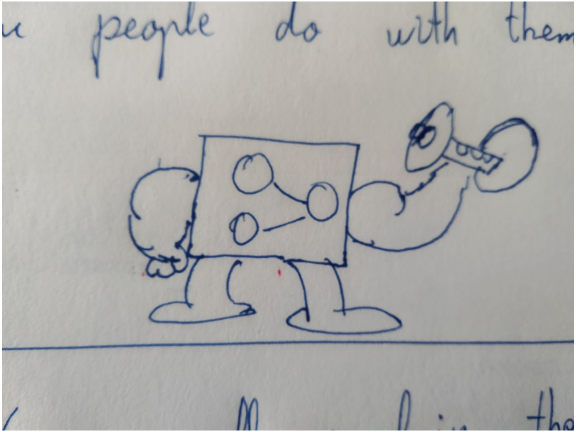

Figure 11: The NN learns its weights with training!

Lets start with an example.
Imagine the zero-defect-manufacturing (ZDM) of 5G-DIVE.
A camera checks a piece, and reports:

- $x_1 \in [0, 11]$ - horizontal-offset;
- $x_2 \in [0, 11]$ - vertical-offset; and
- $x_3 \in [0, 255]$ - black intensity (0-255).

and must decide

- $y_1 \in \{0, 1\}$ - piece needs corrections; and
- $y_2 \in \{0, 1\}$ - throw it to the trash.

To train the network we need and historic of observations and the performed actions:

$$(x_1 = 1, x_2 = 2, x_3 = 220) \qquad (\hat{y}_1 = 0, \hat{y}_2 = 1)$$

we denote with $\hat{y}$ what is commonly known as **labels**.

Imagine we have 200 observations, and not only the real values $(\hat{y}_1, \hat{y}_2)$, but the corresponding predictions (y_1, y_2).

During the training, we'd like to minimize:

$$\frac{1}{200}\sum_{i}^{200} ||(\hat{y}_1^i, \hat{y}_2^i) - (y_1^i, y_2^i)||_2^2 \tag{1}$$

this is nothing but the Mean Square Error (MSE)

Ok, but what does this have to do with the trainning and the weights?

Ok, but what does this have to do with the trainning and the weights?
If we check Figure 9, we can see that $o_1 = y_1$ is written as:

$$y_1 = o_1 = f(h_1 w_1, h_2 w_2, h_3 w_3)$$

Ok, but what does this have to do with the trainning and the weights?
If we check Figure 9, we can see that $o_1 = y_1$ is written as:

$$y_1 = o_1 = f(h_1 w_1, h_2 w_2, h_3 w_3)$$

This means that the selection of w_i affect how big is the prediction error ()1)

One can draw the prediction error (MSE in our case) as below.

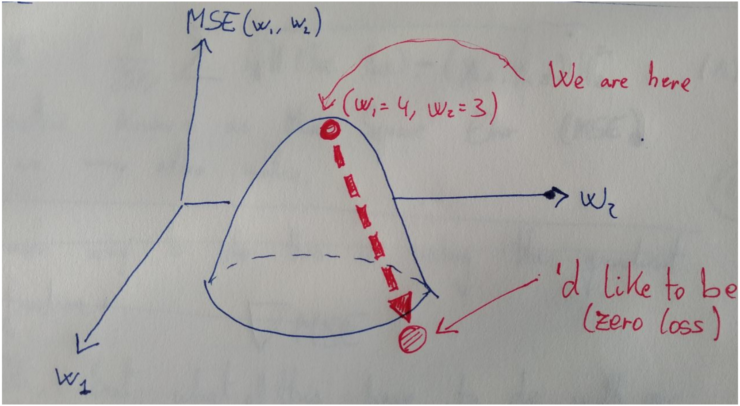

Figure 12: MSE as function of w_1, w_2

Most of you have guessed it right, this is **gradient descend**, it is ML core idea, and we "move" the weights in the direction that minimizes the MSE:

$$(w_1', w_2', w_3') = (w_1, w_2, w_3) - \nabla_w MSE \tag{2}$$

A quite known technique is Stochastic Gradient Descend (**SGD**), and existing methods are inspired in that one.

Back to the game: how do I use this?

Well, in your set of observations you have **batches**: lists of features with its respective labels:

$$batch = \begin{cases} \text{features:} & [(x_1, x_2, x_3), (x_1, x_2, x_3), \ldots], \\ \text{labels:} & [(y_1, y_2), (y_1, y_2), \ldots] \end{cases}$$

You do a gradient-descend step “a batch at a time”.
Lets say we have our batch with 2 observations:

$$batch = \begin{cases} [(x_1 = 1, x_2 = 3, x_3 = 220), (x_1 = 3, x_2 = 1, x_3 = 20), \ldots] \\ [(y_1 = 0, y_2 = 0), (y_1, y_2), \ldots] \end{cases}$$

We plug these values in our network in next slide.

uc3m

$$\begin{cases} y_1 = & ReLU(x_1 = 1)w_1 + ReLU(x_2 = 2)w_2 + ReLU(x_3 = 220)w_3 = \\ & = w_1 + 3w_2 + 220w_3 \\ y_2 = & \dots \\ y_1 = & \dots = 3w_1 + w_2 + 20w_3 \\ y_2 = & \dots \end{cases}$$

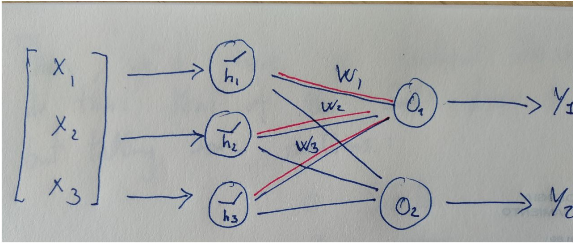

Figure 13:

If we plug above expressions in the MSE expression (1), we have:

$$
\begin{aligned}
MSE(\cdot,\cdot) =& \frac{1}{2}\left(||(\hat{y}_1,\hat{y}_2)-(y_1,y_2)||_2^2 + ||(\hat{y}_1,\hat{y}_2)-(y_1,y_2)||_2^2\right) = \\
&= \frac{1}{2}\big(||(0,1)-(w_1+3w_2+220w_3,\ldots)||_2^2 \\
&\qquad + ||(0,0)-(3w_1+w_2+20w_3,\ldots)||_2^2\big) = \\
&\ldots \\
&=
\end{aligned}
$$

If we plug above expressions in the MSE expression (1), we have:

$$
\begin{aligned}
MSE(\cdot,\cdot) =& \frac{1}{2}\left(||(\hat{y}_1,\hat{y}_2)-(y_1,y_2)||_2^2 + ||(\hat{y}_1,\hat{y}_2)-(y_1,y_2)||_2^2\right) = \\
&= \frac{1}{2}\big(||(0,1)-(w_1+3w_2+220w_3,\ldots)||_2^2 \\
&\quad + ||(0,0)-(3w_1+w_2+20w_3,\ldots)||_2^2\big) = \\
&\ldots \\
&= \text{A MESS!}
\end{aligned}
$$

Happily, NNs atomatically do the job using **back-propagation**[3].

[3] Wikipedia

Happily, NNs atomatically do the job using **back-propagation**[3].

In any case, imagine previous slide gave us:

$$MSE(\cdot, \cdot) = w_2 + 4w_3$$

then, our batch would give as a gradient of

$$\nabla_w MSE(\cdot, \cdot) = (0, 1, 4)$$

and we would update our weights (init as zero):

$$(w_1', w_2', w_3') = (w_1 = 0, w_2 = 0, w_3 = 0) - \alpha(0, 1, 4)$$

[3] Wikipedia

Wait, what was that α?

Wait, what was that α?

it is the **learning rate**, i.e., how fast you want the NN to learn upon each batch update.
Usually you decrease α as you advance in the training

So lets recap what we've learned:

1. Our NN has weights w_i;

So lets recap what we've learned:

1. Our NN has weights w_i;
2. Each training step uses a batch:
   - inputs: $[(x_1, x_2, x_3), (x_1, x_2, x_3)]$
   - labels: $[(y_1, y_2), (y_1, y_2)]$

So lets recap what we've learned:

1. Our NN has weights w_i;
2. Each training step uses a batch:
   - inputs: $[(x_1, x_2, x_3), (x_1, x_2, x_3)]$
   - labels: $[(y_1, y_2), (y_1, y_2)]$
3. For each batch we see how good is the prediction in that batch:

$$MSE(\cdot, \cdot) = \frac{1}{2}\left(||(\hat{y}_1, \hat{y}_2) - (y_1, y_2)||_2^2 + ||(\hat{y}_1, \hat{y}_2) - (y_1, y_2)||_2^2\right)$$

So lets recap what we've learned:

1. Our NN has weights w_i;
2. Each training step uses a batch:
   - inputs: $[(x_1, x_2, x_3), (x_1, x_2, x_3)]$
   - labels: $[(y_1, y_2), (y_1, y_2)]$
3. For each batch we see how good is the prediction in that batch:

$$MSE(\cdot,\cdot) = \frac{1}{2}\left(||(\hat{y}_1, \hat{y}_2) - (y_1, y_2)||_2^2 + ||(\hat{y}_1, \hat{y}_2) - (y_1, y_2)||_2^2\right)$$

4. We update w_i using gradient descend based on MSE:

$$(w_1', w_2', w_3') = (w_1, w_2, w_3) - \alpha \nabla_w MSE(\cdot,\cdot)$$

Talk is cheap, show me the code

Linus Torvalds

Talk is cheap, show me the code

Linus Torvalds
Luca Cominardi

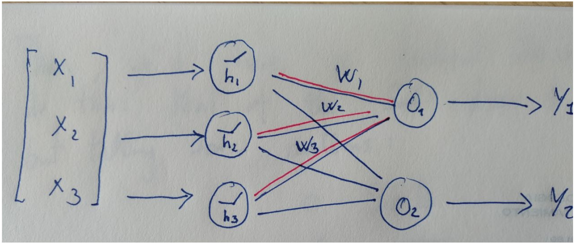

Figure 14:

```
NN_model = tf.keras.Sequential([
  # (h1,h2,h3)
  tf.keras.layers.Activation('relu', input_shape=[3]),
  # (y1, y2)
  tf.keras.layers.Dense(2, use_bias=False),
])
```

You can find a complete implementation of the ZDM example in

▸ GitHub

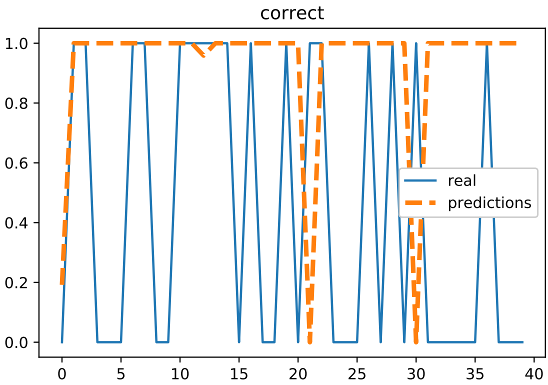

Figure 15: Predicted y_1 output in ZDM