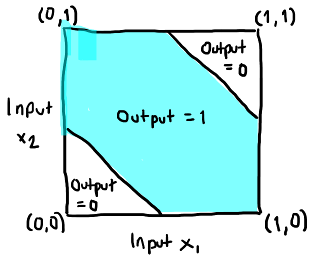
Multilayer Neural Network

Sunday, March 13, 2022 10:48 AM

→ Why multilayer ?

Imagine we have the following case:



- A two variable space given by a data set that has two features (axis) and two classes (output = 0 or 1)

If we build a neural network for the following problem:

Loss function evaluated with test data

with training data

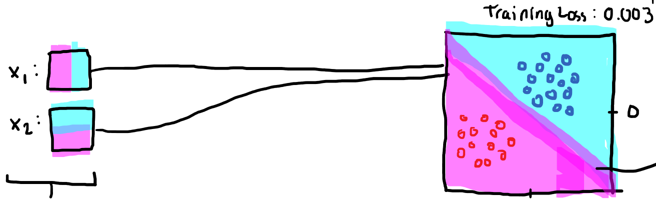
Learning rate: 0.3 Activation: Linear Problem type: classification

Features 0 Hidden Layers Output

Test Loss: 0.003

Training Loss: 0.003

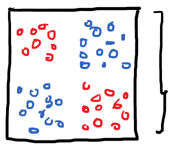
a high one would be ≥ 0.5



this is a hyper plane that divides both classes

1 LAYER: 1 hyper plane

→ But what happens if we need more than one hyper plane?
Here, for example we cannot have one hyper plane that divide all the segments:



the cannot be one line only that separates both groups

↓

actually it might need two planes

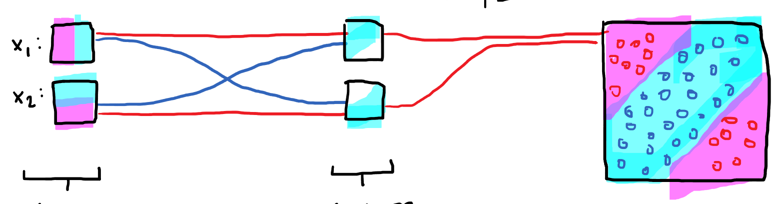
→ Each LAYER adds a hyper plane

→ The activation function curves those hyper planes depending on its function.

↳ the more neurons, the more overtraining might be

Features 1 HIDDEN LAYER, 2 neurons

ReLU activation function



1 LAYER 1 LAYER = 2 hyper planes

HYPERPLANES

A hyperplane is a plane of N dimensions, but we can project a hyperplane in spaces of 2, 3, 4 . . . N dimensions, and thus:

→ 1 hyperplane of 2D: a straight line

→ 1 hyperplane of 3D: a plane

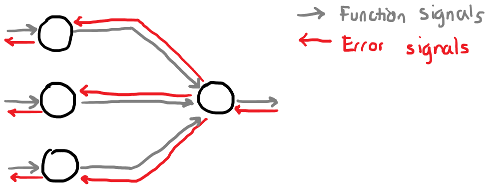
. . .

↳ after the activation function, the plane / line can be curved.

→ Multilayer Neural Network

- Each node/neuron contains a non linear function that can be differentiated.
- The network/architecture contains one or more hidden layers.
- The network has a high connectivity between its nodes which makes it complex.

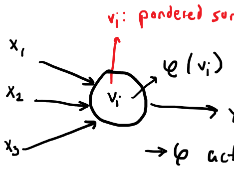
→ Stream of prediction/training



- The information flow goes from the input layer to the output layers, which is the gray line.
- The training flow goes from the output layer to the input layer, which is the red line. This is because we have the answer of which class a data point belongs to, and so the error that drives the training update can be calculated backwards: Back propagation algorithm, for output layer and for the hidden layers.

v_i: pondered sum of inputs $\sum w_i x_i$

softened by an activation function



output

$y(n) = \varphi(v(n))$

→ φ activation function

→ $v(n) = \sum w_i x_i$

→ error

$$e(n) = d(n) - y(n)$$

Average square error $= \mathcal{E}(n) = \frac{1}{2} e^2(n)$

- Output layer case:

  keep in mind that the error is the differences between the desired and obtained result:

$$e_j(n) = d_j(n) - y_j(n)$$

and the quadratic error

$$\mathcal{E}_j(n) = \frac{1}{2} e_j^2(n)$$

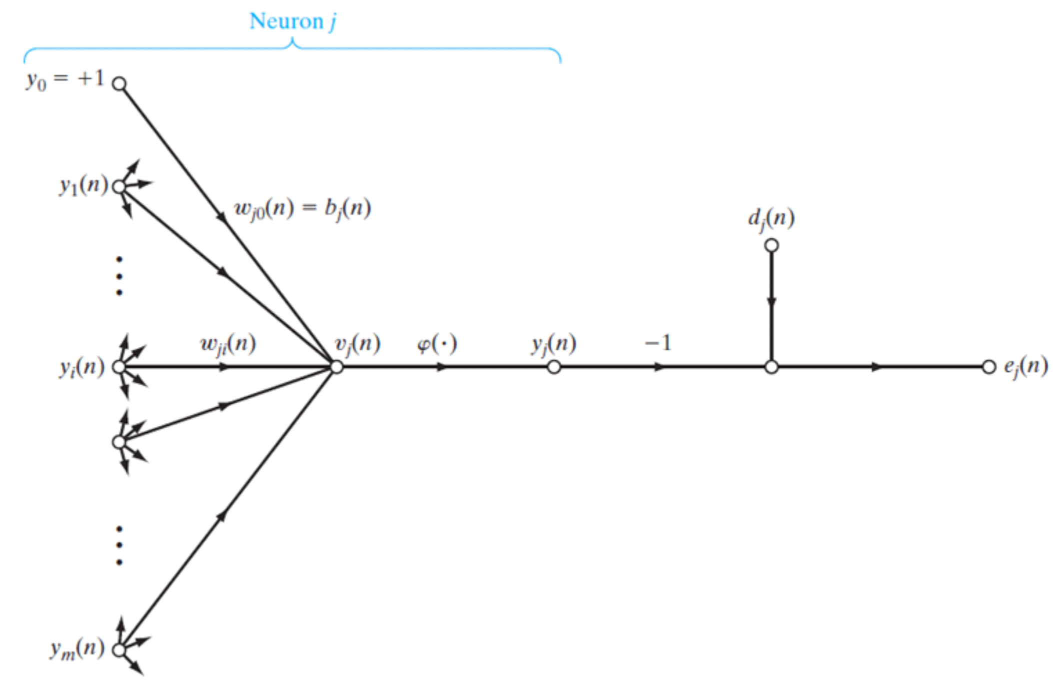
The output after the activation function φ

$$y_j(n) = \varphi_j(v_j(n))$$

And the weighted input

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n)\, y_i(n)$$

→ Back propagation training



Now, we want to calculate the $\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)}$

$$\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} = \overset{(1)}{\frac{\partial \mathcal{E}(n)}{\partial e_j(n)}} \overset{(2)}{\frac{\partial e_j(n)}{\partial y_j(n)}} \overset{(3)}{\frac{\partial y_j(n)}{\partial v_j(n)}} \overset{(4)}{\frac{\partial v_j(n)}{\partial w_{ji}(n)}}$$

(1) $\frac{\partial \mathcal{E}(n)}{\partial e_j(n)} = e_j(n)$ (4) $\frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n)$

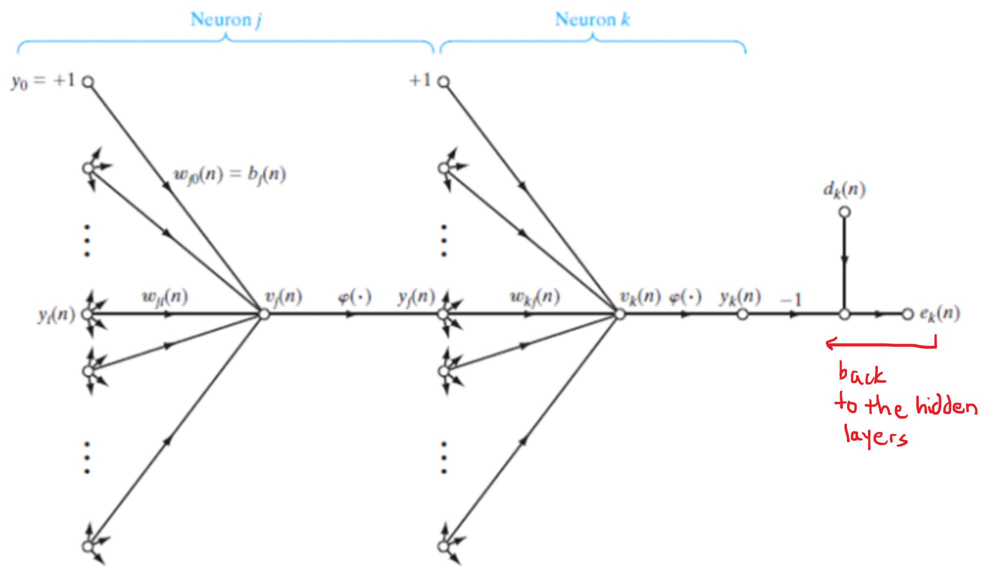
(2) $\frac{\partial e_j(n)}{\partial y_j(n)} = -1$ (3) $\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'(v_j(n))$

Therefore, by multiplying the partial derivatives (1)·(2)·(3)·(4), we obtain, by the use of this rule chain, the descending gradient

$$\Delta w_{ji}(n) = -\eta \frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)}$$

descending so that with iterations, the solution gets approached by this delta that descends through a Cost/Loss function until its critical minimum point.

All this was for the output layer, but for the hidden Layers, since we do not have as reference the desired class, unlike the output layer, the error we get at the end will be back propagated to the hidden layers update.



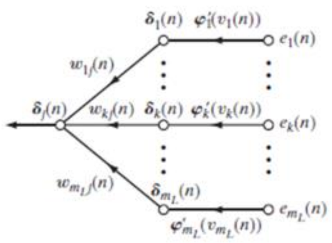
Basically, the error gotten at the output, will be passed through the hidden layers so that their weights can be updated. That's why we say the information stream goes to the right and the error stream to the left.

So, for hidden layers (intermediate layers), the error is considered based on the output called q(n), given that we no longer have d(n) as reference,

$$\delta_j(n) = -\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)}$$

Giving us the gradient as

$$\delta_j(n) = \varphi_j'(v_j(n)) \sum_k \delta_k(n)\, w_{kj}(n), \qquad \text{neuron } j \text{ is hidden}$$

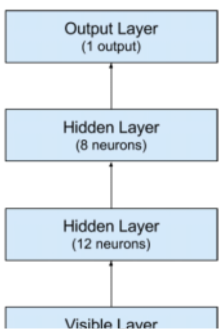


Thus, a summary of the algorithm,

$$\begin{pmatrix}\text{weight}\\ \text{correction}\\ \Delta w_{ji}(n)\end{pmatrix} = \begin{pmatrix}\text{learning rate}\\ \text{parameter}\\ \eta\end{pmatrix} \times \begin{pmatrix}\text{local}\\ \text{gradient}\\ \delta_j(n)\end{pmatrix} \times \begin{pmatrix}\text{input signal}\\ \text{of neuron } j,\\ y_i(n)\end{pmatrix}$$

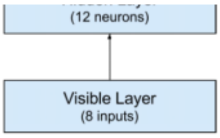
→ Feed forward: feed towards the output (to right).
→ Back propagation: feed towards the left.

i.e. code the following Network:



One neuron since we want 0, 1 as classes (2 outcomes) YES/NO

A	B	C	D	E	F	G	H	I
6	148	72	35	0	33.6	0.627	50	1
1	85	66	29	0	26.6	0.351	31	0
8	183	64	0	0	23.3	0.672	32	1
1	89	66	23	94	28.1	0.167	21	0
0	137	40	35	168	43.1	2.288	33	1
5	116	74	0	0	25.6	0.201	30	0
3	78	50	32	88	31	0.248	26	1
10	115	0	0	0	35.3	0.134	29	0



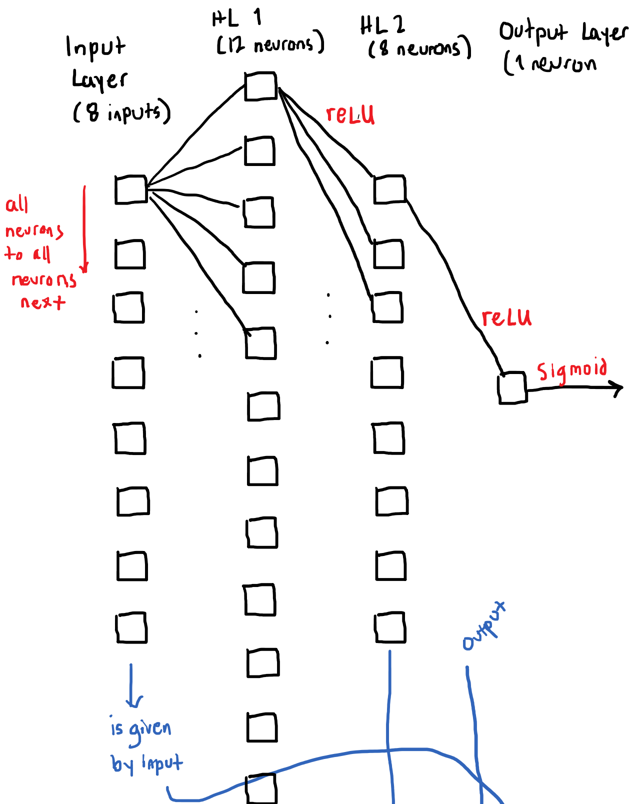
5	116	74	0	0	25.6	0.201	30	0
3	78	50	32	88	31	0.248	26	1
10	115	0	0	0	35.3	0.134	29	0

X matrix (first 8 columns) — Y vector (last column)

Database: Pima Indians Database

Python: keras dense → all neurons are connected to all neurons

keras sequential → you add layers in order: 1st layer, 2nd ...




```
model = Sequential()
model.add(Dense(12, input_dim=8, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

79 Accuracy: Good

↳ HL 1: 24 neurons

↳ Epochs: 500

↳ batch_size: 50 (less frequent)

} converges quickly

loss output in python: 11.31 } 11 registers wrong (average)

→ choose not such a big batch size