Universidad de Alcalá

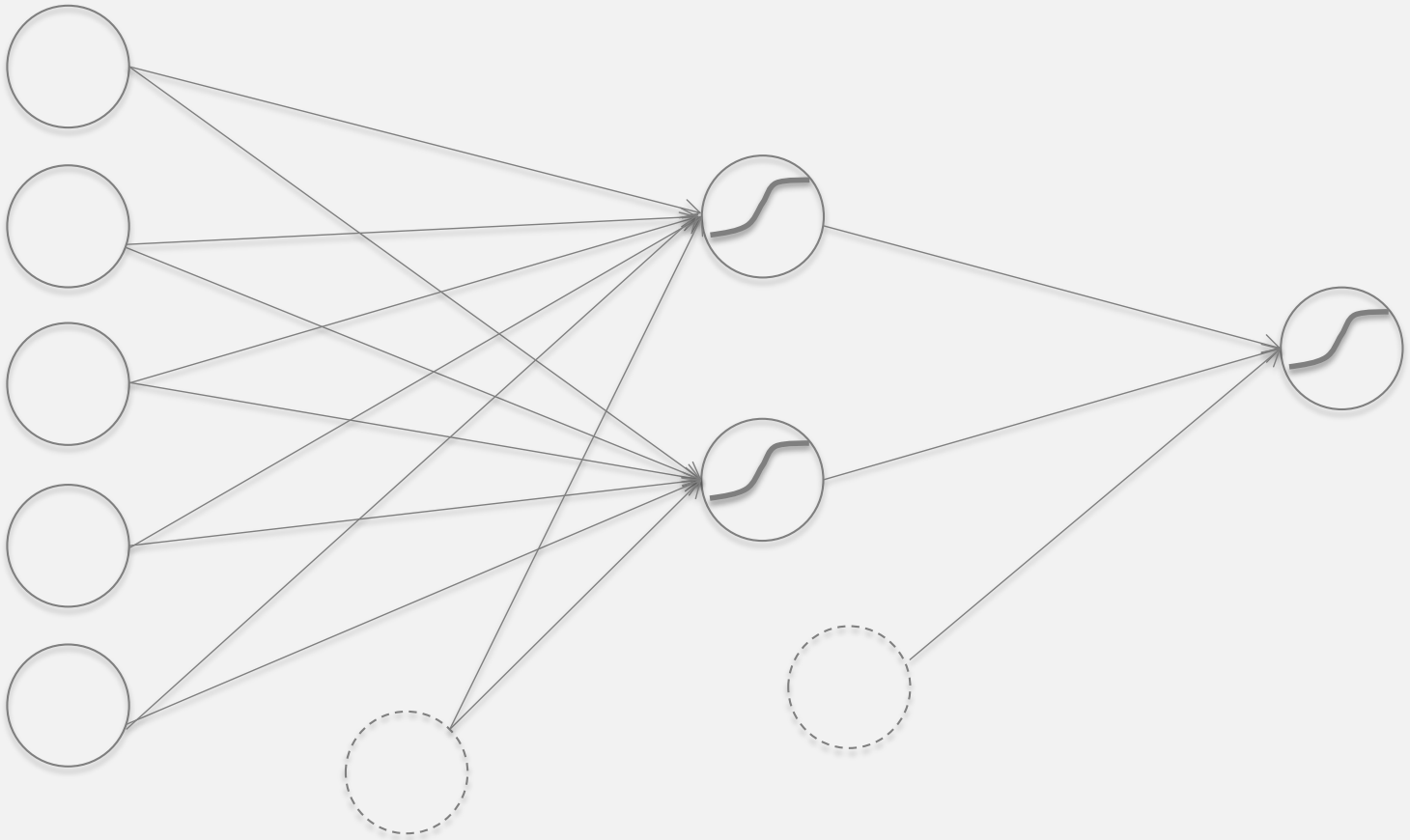
Redes Alimentadas hacia adelante I

Prof. Ignacio Olmeda
AI LAB

FEEDFORWARD NETWORKS

- The *Perceptron* is a special case of a more general model which is called *Feedforward Networks*.

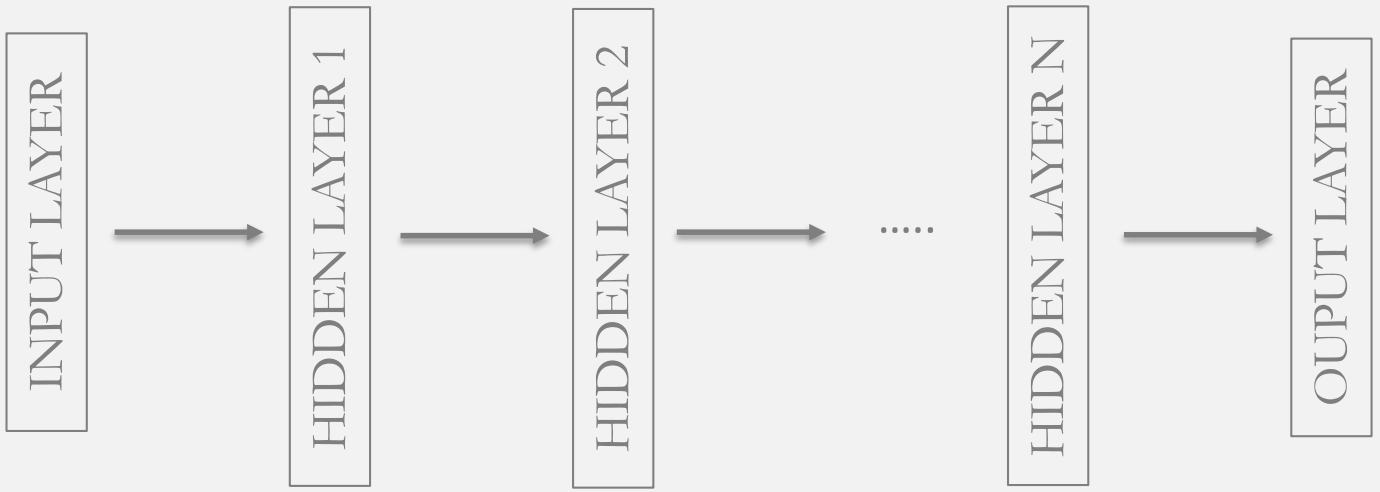
- Schematically, a simple feedforward neural network, with five inputs, two *hidden units* (explained later) and one output, has the following structure:



- First, note that the network has more layers, in the *Perceptron* we only had input and output layers while in our example we have an additional layer between these two.

- This layer is called a *hidden layer*, the name comes from the fact that it has no connection from or to the outside.

- In the preceeding plot we have presented a network with only one hidden layer but there can be more of them:



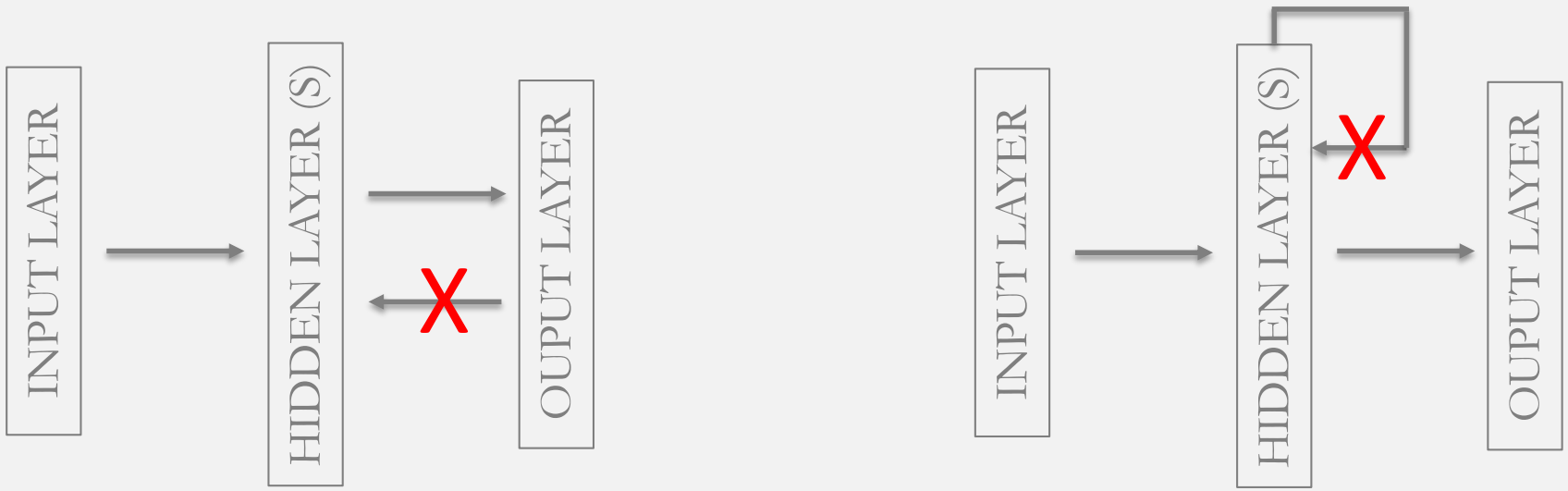
- Hidden layers do not need to have (and generally do not have) the same number of units, the specific number of units in each layer will depend on the particular application.

- Another thing to note is that feedforward networks also make use of the extra artificial inputs that were considered in the *Perceptron*, these inputs are also always equal to +1

- Note that these artificial inputs are added no only in the output unit but also to every unit in the intermediate layer.

- Similarly to the *Perceptron*, the information flows from the input to the output, in the case of feedforward networks, through hidden layers.

- Consequently there are no backward connections or loops as we will find in other architectures that will be studied later (e.g., *sequential networks*).



- In comparison with the *Perceptron*, *Feedforward Networks* extra (*hidden*) layers are not the only difference.

- Feedforward neural networks incorporate these three main differences:

    - Nonlinear functions in the processing units
    - More than one output
    - Extra layers between input and output

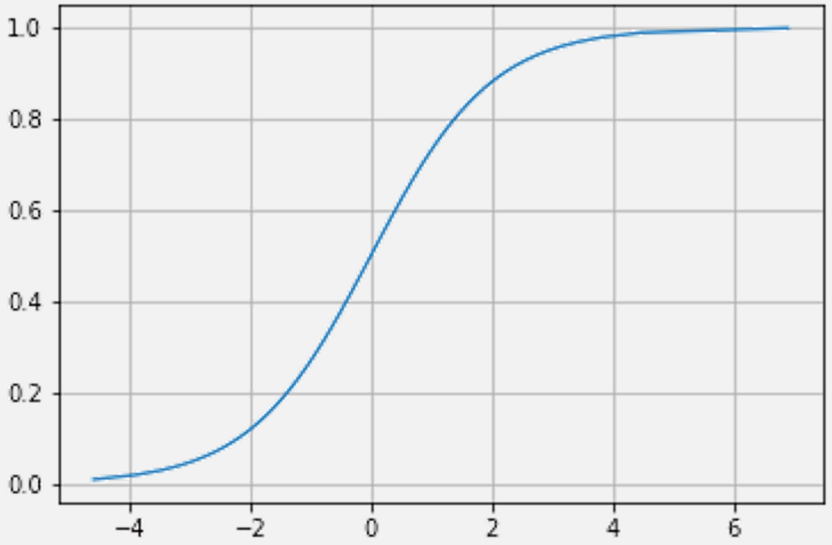
- First let us introduce nonlinearity in the units.
- In the *Perceptron*, units do not perform any computation at all: input units just receive the values of the features that the environment provides and the output unit just receives the sum of weighted inputs of the input units.
- Now, assume that the units may perform some computation, for example, executing some nonlinear function over the inuts that the neuron receives.
- This is called an *activation function*.
- Activation functions try to emulate some of the transformations of incoming signals that real neurons perfom.

- In principle, one might use any nonlinear function (for example the sine function, the square root, etc.) but for mathematical reasons, that we will detail later, it is common to use *sigmoid functions* which have an "s-shape".

- A kind of sigmoid functions extensively is the *logistic function*, it has the following expression:

$$f(x) = \frac{1}{1 + e^{-x}}$$

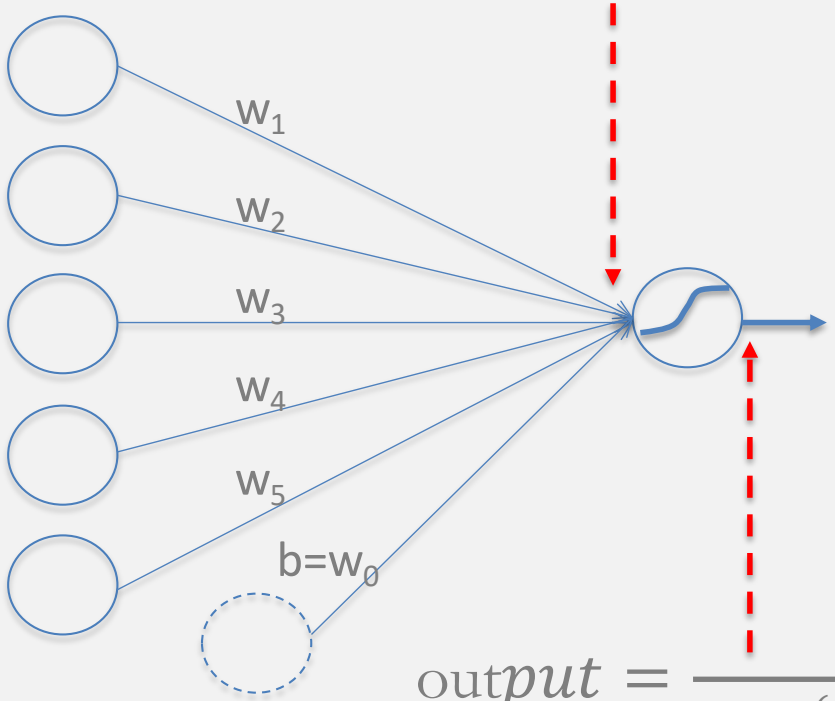
- And shape:



- Note that the logistic function is *bounded*, the minimum value is zero and the maximum value is one, regardless of what the value of x is.

- For example, assume that we have just five input features and that the output unit executes the logistic function (for simplicity we assume no hidden units), we have

$$input = w_0x_0 + w_1x_1 + \ldots + w_5x_5$$



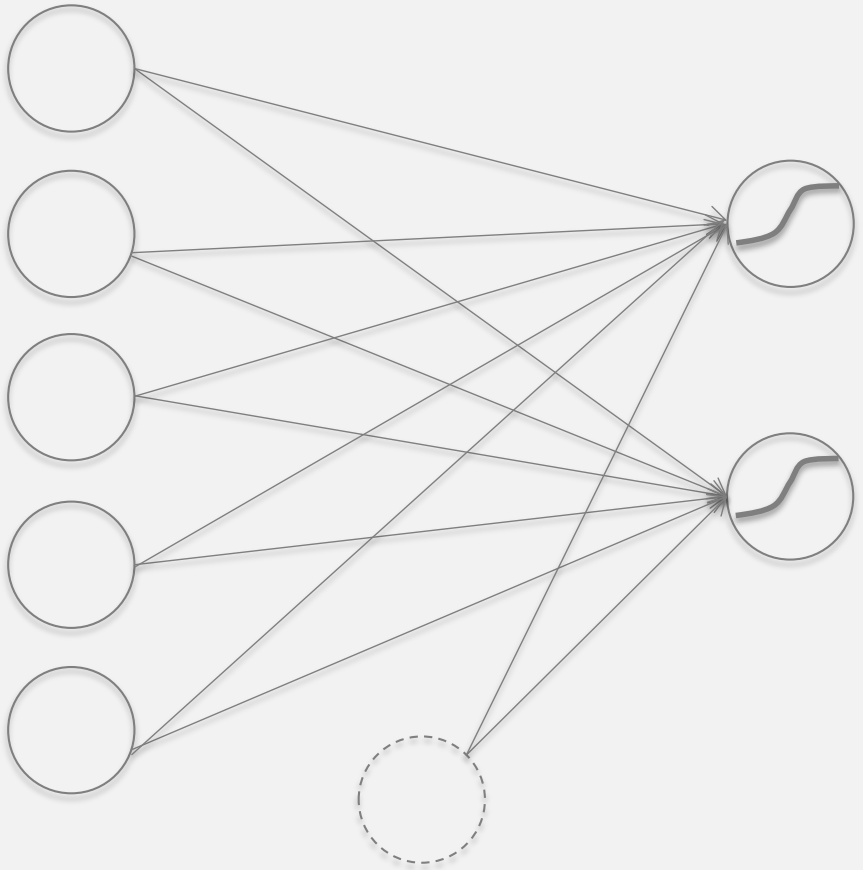
$$output = \frac{1}{1+e^{-(w_0x_0+w_1x_1+\ldots+w_5x_5)}}$$

- Note that the use of a logistic unit at the output adds flexibility to the model since now it is possible to obtain any number between [0,1] instead of only two values {0,1}, as in the *Perceptron*, which used a threshold function.

- For example, a Perceptron can not be used to calculate the degree to which an example belongs to a particular class, examples are just "0s" or "1s".

- But in real applications things are not always this way, we have some mesurement of confidence or probability.

- The second extension is straighforward, multiple outputs can be easily accomodated just replicating the structure.
- For example for two outputs (again, we asume no hidden units):



- Of course, the outputs will provide different values that depend on the same inputs.

- For example, we may want to use some variables such as blood preassure, level of glucosa, etc. to predict the probabilities of two different diseases at the same time, the information can be the same but predictions are of course independent.

- Alternatively, we could hav used two different networks.

- Note that, thought tied together, some of the weights of each of the outputs are independent so that learning is performed for each of the outputs.

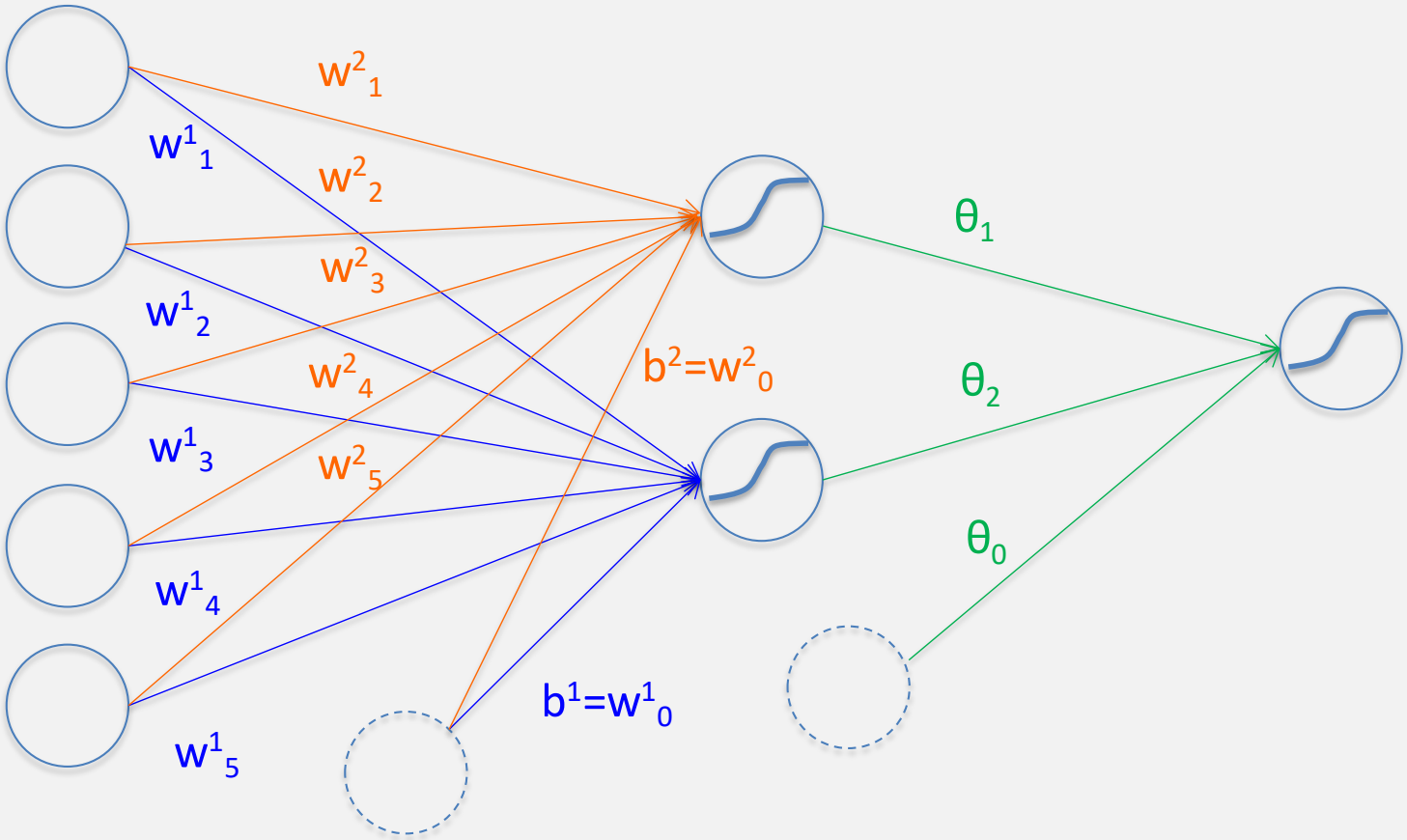
- To do this, we have to compute the errors at the output independently, i.e. we do not compute an overal measure but independent measures of errors for each of the ouputs.

- The third extension, the addition of hidden layers, mentioned previously, it is enormously important and together with the use of nonlinear activation functions is what it gives feedforward neural networks their power.

- Hidden units also perform nonlinear transformations of the inputs received from the preceeding layer.

- These transformations are sent to the next layer but, once again, weighted by some parameters.

- For example, consider a feedforward network with a single "hidden" layer with two units and, for simplicity, only one output, the structure is:



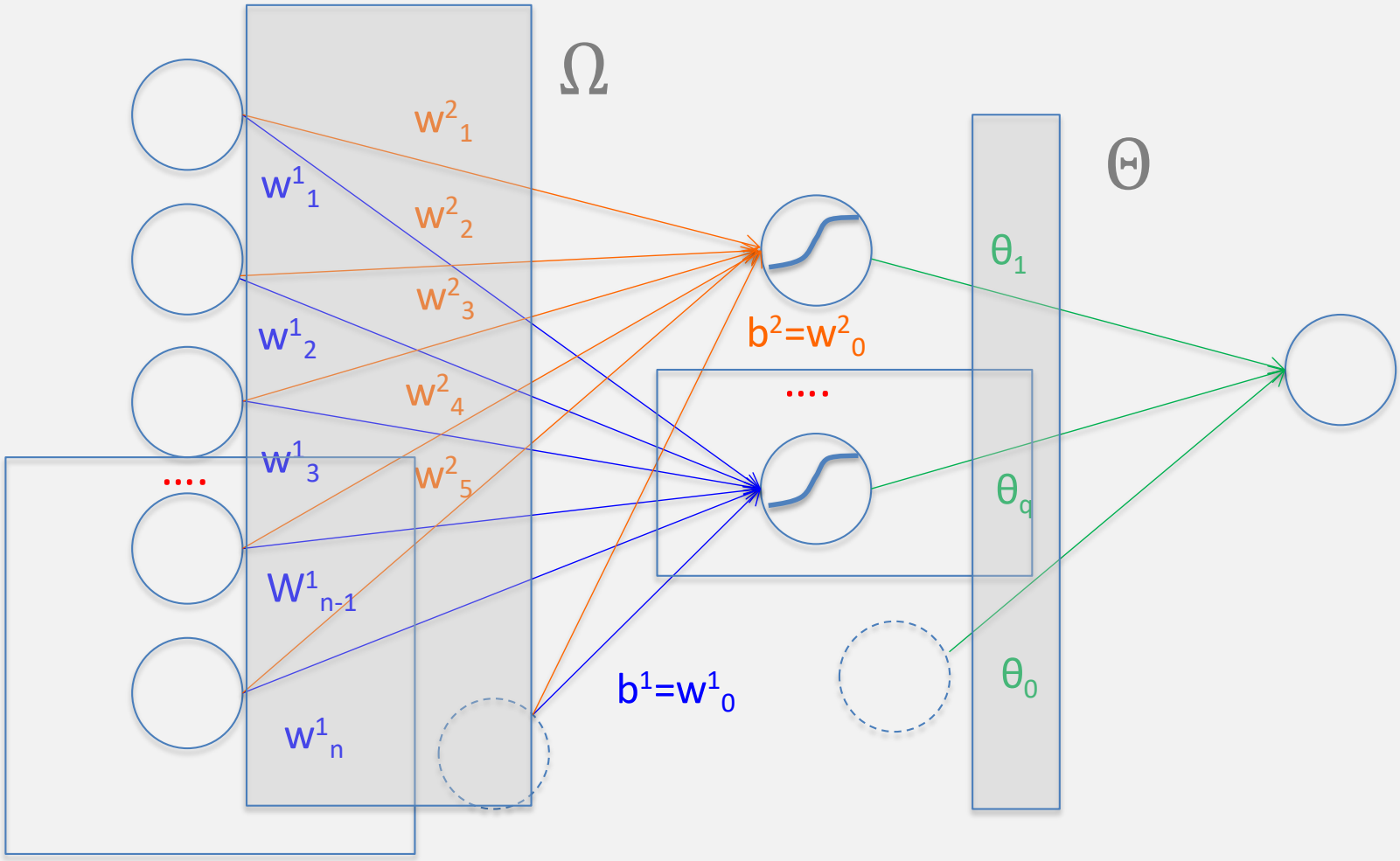
UNIVERSAL APPROXIMATION PROPERTY

- As we have seen feedforward networks are more complex than the perceptron, one may ask why we need to complicate the structure, which is the good of doing this.

- The reason is very simple: while perceptrons are limited in their capabilities (they can only learn linearly separable functions) feedforward neural networks can learn any *arbitrary* function.

- This was demonstrated by Hornik et al. (1989): under very general assumptions, if there exists a relationship between inputs and outputs a single hidden layer neural net could approximate such relationship to any degree of accuracy given enough hidden units.

- To understand this, assume a single hidden layer feedforward network (for simplicity we asume no activation at the output and one single output):



- Notice that the output of the network depends on the inputs (X) as well as on the values of the weights that link the input layer and the hidden layer (Ω) and the weights that link the hidden layer to the ouput layer (Θ) :

$$\text{Output} = \Phi(X, \Omega, \Theta)$$

- Now, according to the mentioned result if we want to find an unknown relationship between X and y:

$$y = f(X)$$

- It is always posible to find a set of weights (Ω, Θ) so that

$$y = f(X) \approx \Phi(X, \Omega, \Theta)$$

To any degree, provided that the number of hidden units is enough.

- Notice how powerful this result is: a feedforward network is capable of Learning *any* function, i.e., if there exists a relationship between the input and the output the network will find it.

- This is what we called the universal approximation property in the context of non parametric models and in practical temrs it means that feedforward ANNs are general problem solvers.

- This explains the success of Deep Learning: ANNs not only work, they *should* work.

- Of course, one has to determine how many hidden units are enough.

- And also we have to be sure that such relatioship exists.

- Of course this does not mean that such models are the only possibility, for example we could use networks with more hidden layers that conserve such property.

- The property that feedforward networks with only one hidden layer were "enough" to solve arbitrary problems together with the fact that networks with more hidden layer were computationally intractable led to the ubiquity of such models in the last two decades.

- Algorithmic improvements together with an increase in the computational power made that actual implementations generally include more layers, giving birth to what it is now called *deep learning*. .

- A more practical problem is that the result is not *constructive* in the sense that it tells you that such weights exists but does not tell you how to find them.

- The problem of finding the weights of a feedforward neural network fall in the ML domain and is precisely what before we defined as *Learning*.

- In feedforward neural networks is not straighforward to derive a Learning algorithm that delivers the optimal weights.

- In fact, until the Discovery of the *backpropagation algorithm*, (Rumelhart et al. 1986), explained later, no efficient way of finding such wights was available.

- To understand how learning is performed in feedforward neural networks one needs to go deeper in the details f the functioning of such networks, particuarly on how weights affect the output of the network and, consequently, the errors it makes.

- In this sense, the understanding of how loss functions are calculated is essential.

LEARNING AND LOSS FUNCTIONS

- As we have seen there are major differences between feedforward networks and the *Perceptron*.

- Nevertheless, the problem of learning is exactly the same: one needs to find the weights that are optimal for a particular loss function, i.e. which allow to solve a particular problem.

- The flexibility of feedforward networks against perceptrons come to a cost: as we will see the dimensionality of the problem is much higher and this complicates enormously the Learning process.

- As said the dimensionality of feedforward networks is much higher, for example asume a simple network which has *r* inputs, *q* hidden units and *s* outputs, the number of weights is:

$$r \times q$$

(the weights that connect inputs with hidden units)

$$+$$

$$q$$

(the biases for the hidden units)

$$+$$

$$q \times s$$

(the weights that connect the hidden units to the output units)

$$+$$

$$q$$

(the biases for the output units)

$$=$$

$$q(r+s+2)$$

- For a simple network to perform multiclass (10 clases) digit recognition (28x28=784 pixels) and with 20 hidden units we have

$$20(784+20+2) = 16.120 \text{ weights}$$

- As you see, even for problems that can be considered very simple, the number of calculations is huge.

- There are very efficient algorithms that can handle this and much higher number of calculations.

- Also, as we will see, some DL architectures are better suited to particular problems reducing the computational load, for example, Convolutional Neural Networks in image recognition.

- In any case, learning works exactly the same as with the *Perceptron* algorithm:

  i) some input is presented to the network,
  ii) a forward pass is performed to calculate the output
  iii) the error of output is computed by comparing the actual and desired outputs and
  iv) such error is used to modify the correponding weights.

- Also, all these operations are performed iteratively, i.e. several passes along the training set are needed.

- As we have seen, one crucial issue in feedforward networks (as well as in any DL or ML model) is the choice of the loss function.

- In the case of the *Perceptron* the loss function was very intuitive since we had only two possible classes, even though we did not have to explicit it, the perceptron algorithm minimizes the number of misclassifications.

- In general applications the situation changes: multiclass, regression problems, asymmetric costs…

- One common error is not to be aware of the underlying loss function that the algorithm is using, for example, in the case of cancer detection that we saw, costs are asymetric.

- In feedforward network we have to define the specific loss function for the specific problem we are solving and, according to that loss function, we have to write the corresponding algorithm to optimize it.

- In particular, the *Perceptron* could be used only for binary classification tasks, but feedforward are general problem solvers which can be used in a variety of problems: multiclass classification, regression, …

- Writing the specific algorithm for a particular loss function can be cumbersome, for this reason most of the practitioners just use the algorithms which are pre-encoded and asume they will work on their particular problem.

- While this is true many times, you have always to be sure which loss function you would like to optimize and also be sure that the algorithm you use is consistent.

- For example, asume the simple case of binary classification bu asume that each of the errors (false positives and false negatives) have distinct consequences, for exaple is not the same to predict that a patient may have a disease when she does not have it that to predict that she has not such disease when, in fact, she has it.

- In such a case, the number of misclassifications is not an adequate loss function since it does not consider the asymetric cost of misclassification.

- Patients will not accept excuses on medical errors just because “this is what the program said”, remember that ML (and DL) algorithms will do what you have trained them to do.

- The problem of building loss functions and algorithms to optimize them in specific settings is quite complex, we will just introduce some concepts that help to understand how Learning is performed in feedforward network with no hidden units.

LEARNING IN A SIMPLE FEEDFORWARD NETWORK

- As we have seen, feedforward networks include three main modifications: i) nonlinear activation functions, ii) arbitrary loss functions and iii) –multiple- hidden layers, all have consequences for learning.

- First, we will analyze how the introduction of nonlinearity changes the learning algorithm.

- In general, when we change a weight the output will change and consequently the error will change, becoming higher or lower depending on the introduced change.

- Of ourse we want to change weights iteratively so that the error decreases, i.e. we want to change only the weights that produce such decrease in error and also change them depending on how big the error was.

- For example, if we notice that changing one weight does not have a big impact on the output then the change should be minor while if it is big the change must be also big.

- We ca understand this better analyzing the perceptron algorithm:

$$\Delta w_i = (a^k - \Phi_w(\boldsymbol{x}^k))\, x^k{}_i$$

- The change in the corresponding weight depends on two things, first, the overall "participation" of the corresponding input unit in the output,

$$x^k{}_i$$

- Intuitively: if there was no participation ($x^k{}_i = 0$) then there is no need to change $\Delta w_i = 0$.

- Note that the participation of the unit in the ouput is proportional, i.e. if we increase the input one unit the

- Second, if the error was big we would need big changes while if it was small we would need just minor changes

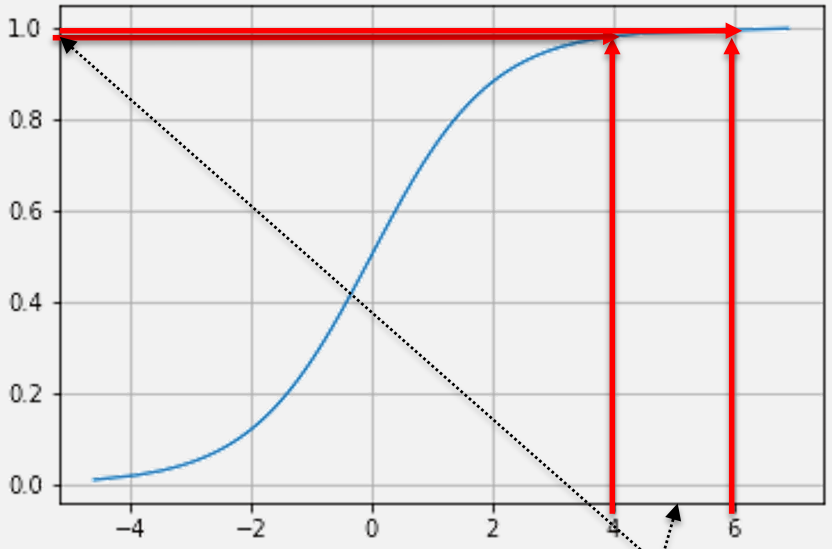
$$(a^k - \Phi_w(\boldsymbol{x}^k)) = \textit{output error}$$

- So, simplyfiying, the perceptron algorithm follows the heuristic:

$$\textit{weight change} = \textit{output error} \times \textit{participation of the input unit}$$

- Now, in feedforward neural networks, nonlinearity makes that a change in the weight does not necessarily produce a change in the ouput of the same magnitude.

- To ilustrate consider the output of a logistic function:



$$f(x) = \frac{1}{1 + e^{-x}}$$

			increase
x =	4	6	2
f(x) =	0,98201	0,99753	0,01551 ≈ 0

- When we have a function which is *linear* on the weights:

$$f(w_0, w_1, \ldots, w_n) = w_0 x_0 + w_1 x_1 + \ldots + w_n x_n$$

If we change one weight, i.e. w_j to $w_j + \Delta w_j$, the function changes to:

$$f(w_0, w_1, \ldots, w_j + \Delta w_j, \ldots, w_n) =$$
$$= w_0 x_0 + w_1 x_1 + \ldots + (w_j + \Delta w_j) x_j + \ldots w_n x_n =$$
$$f(w_0, w_1, \ldots, w_n) + \Delta w_j \; x_j$$

- But when it is *nonlinear* such reaction is not proportional, as we have seen, for the logistic (asume, for simplicity, x_j =1):

$$f(w_j = 4) = 0.982$$
$$f(w_j + \Delta w_j = 4 + 2) = f\left(w_j = 6\right) = 0.997$$
$$\Delta w_j x_j = 2$$

$$f(w_0, w_1, \dots, w_j + \Delta w_j, \dots, w_n) = 0.997 \neq 0.982 + 2 = 2.982$$

- So, it is less straighforward to calculate the "reaction" of the network to changes in weights, it will depend on the particular function used.

- When weights are changed, the value of the loss function changes because the ouput would also change.

- So, if the weight changes, the output would change and the loss (error) would also change.

- Consequently, we need to evaluate the change in the value of the error a function (*loss*) against one of its inputs:

$$\frac{change\ in\ error}{change\ in\ one\ weight}$$

- In calculus, the change in the value of a function against one of its inputs is called the partial derivative of the function, with our notation it is represented as:

$$\frac{\partial L}{\partial w_j}$$

- Where L is the loss function and w_j is any of the weights of the network.

- In feedforward neural networks the loss function is arbitrary, i.e. depending on the problem we would use one or another.

- For example, if we are trying to classify some example into a binary class, the loss function employed could be the *negative entropy*:

$$L(Y,\phi_W(X)) = \sum_{i=1}^{m} -y^i \log(\phi_w(x^i)) - (1-y^i)\log(1-\phi_w(x^i))$$

- If w changes, then Φ changes and consequently L changes.

- If we want to produce successful changes in the weights, i.e. those which make the error smaller, then we have to predict the change in the loss function according to such errors.

- We would need to calculate the partial derivative of the loss against each of the weights:

- And then to move accordingly, i.e.
  - If the loss *increases* when we increase the weight then we would like to *decrease* the weight.
  - If the loss *decreases* when we increase the weight then we would like to *increase* the weight.

- So, weights must be changed moving in the oposite direction to the partial derivative:

$$w'_i = w_i + \Delta wi = w_i - \frac{\partial L}{\partial w_i}$$

LEARNING ALGORITHM FOR A FEEDWORWARD NETWORK WITH NO HIDDEN UNITS

- As said, to make the network learn we can calculate the gradient of the loss function along each of the weights and move in the opposite direction.

- We have

$$\frac{\partial g(y,\phi_W(X))}{\partial w_j} =$$

$$\frac{\partial\left(-\frac{1}{m}\sum_{i=1}^{m} y^i \log\phi(w_0x_0^i+\ldots+w_jx_j^i+\ldots+w_nx_n^i)-(1-y^i)\log(1-\phi(w_0x_0^i+\ldots+w_jx_j^i+\ldots+w_nx_n^i))\right)}{\partial w_j}$$

- Note that, again, the terms that appear in the above equation cancel when taking the derivarive against a particular w_j because they are constants

- Also, since

$$\frac{\partial \log \phi_w(x)}{\partial w_j} = \frac{\partial \phi_W(X)}{\partial w_j} \frac{1}{\phi_W(X)} = -\phi_W(X)(1-\phi_W(X)) \frac{1}{\phi_W(X)} x_j =$$
$$= -(1-\phi_W(W))x_j$$

- We have

$$\frac{\partial g(y, \phi_W(X))}{\partial w_j} = \frac{1}{m} \sum_{i=1}^{m} (\phi_W(X^i) - y^i) x_j^i$$

- Notice that this expression is very similar to the one we found when we calculated the parameters of a linear model recursively, the only difference is that now instead of having

$$\sum_{i=1}^{m}(\Omega X^i - y^i)x_j^i$$

- We have

$$\sum_{i=1}^{m}(\phi_W(X^i) - y^i)x_j^i$$

- Summarizing, the learning algorithm is:

$$w_j = w_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(\phi_W(X^i) - y^i)x_j^i, \quad \forall j = 1,2,...,n$$