Chapter 4: Neural Networks

* Introduction to NN → What & Why ?

Notation ←

* Feed Forward

Vectorization ←

Computation graph ←

* Activation functions & non-linearity of NN → Universality

* Back-propagation

* Optimization technique: From SGD to Adam

* Batch Optimization

* TensorFlow & Pytorch

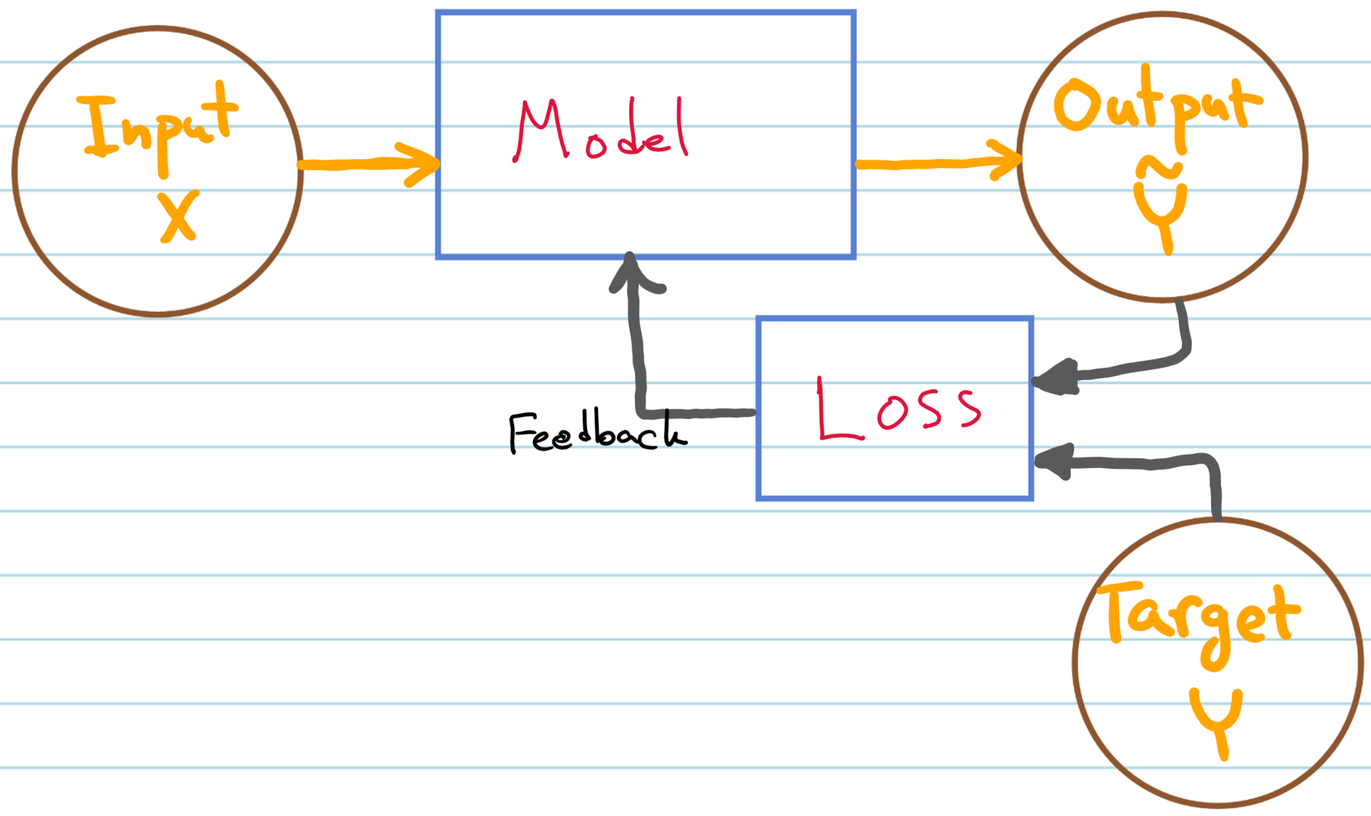
* Overfitting in NN

* Architectures

  - Convolutional NN

  - Recurrent NN

Reminder:

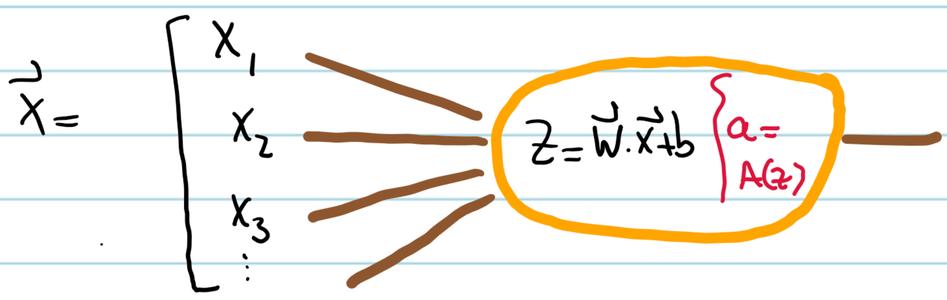


Schematic picture of classification/Regression model training process.

Here we want to introduce NN as a new kind of models. These models are fairly similar to some of the models we've already seen. Specifically, one can describe them as an extension of Logistic Reg. (LR) or perceptrons.

Perceptron / LR

We already introduced logistic Reg. There's a similar model that istead of the sigmoid in LR, uses a step function and $f(x) = (\vec{w}.\vec{x} + b) \geq 0$.

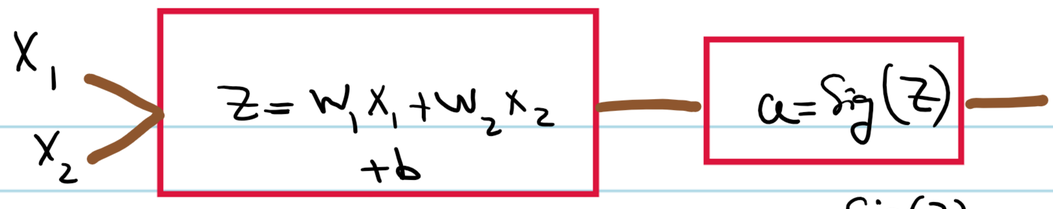


This is similar to some of the models that are used to describe a neuron. Namely, it takes some inputs with different weights and if the result $\vec{w}\vec{x}$ is above some threshold, the neuron would fire, otherwise, it stays silent.

A NN can be described as combination of a bunch of these units (perceptron/LR).

We will briefly revisit these simple models and see why/how we end up with NN. For the rest of this discussion, we'll stick to LR. But almost every thing would be similar for a perceptron.

LR



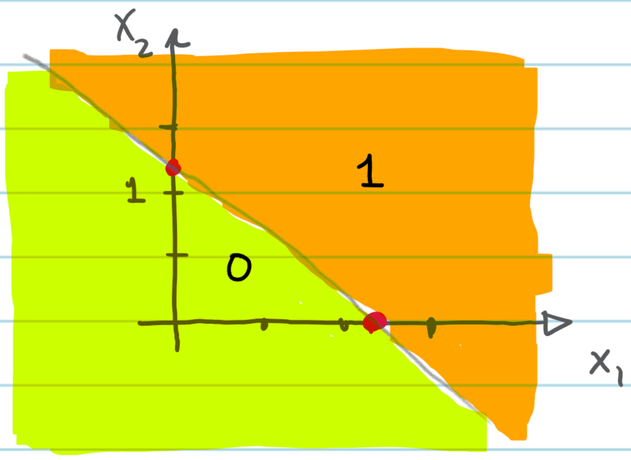
→ This would be a step-function for a perceptron.

For a classification problem

$$\bar{Y} = (a \geq 1/2) \begin{cases} 0 \rightarrow \text{True} \\ 1 \rightarrow \text{False} \end{cases}$$

Example: $w_1 = -1,\ w_2 = -1,\ b = 1.7 \Rightarrow -x_1 - x_2 + 1.7 \geq 0.5$

$$a = 1/2 \Rightarrow \begin{cases} x_1 = 0,\ x_2 = 1.2 \\ x_2 = 0,\ x_1 = 1.2 \end{cases}$$



$$1.2 - x_1 = x_2$$

Consider the example above and restrict inputs to $x_{1,2} \in \{0, 1\}$.

x_1	x_2	Output
0	0	0
0	1	0
1	0	0
1	1	1

$\Rightarrow$ This is the And gate.

Assignment: Design a LR for the Or gate

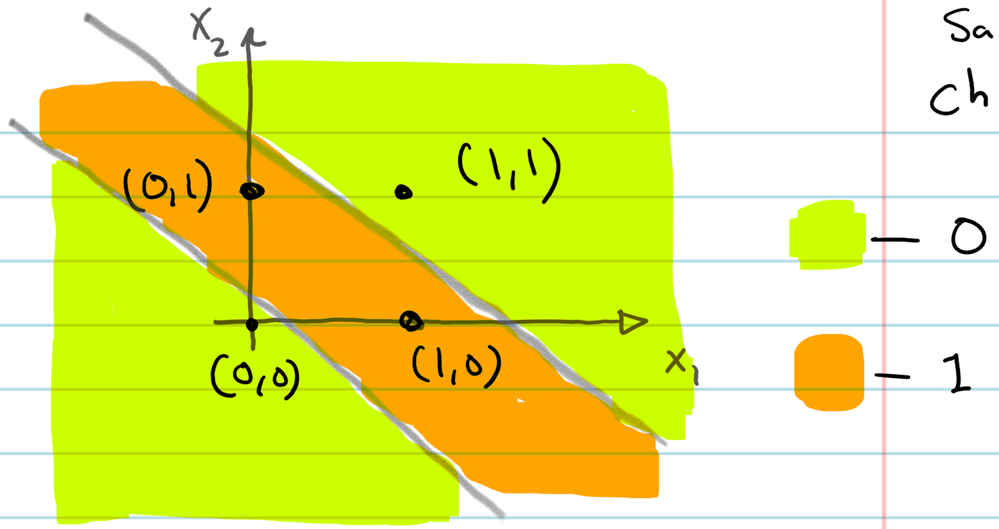
Design a LR for a NOT gate.

Question: Can we do XOR with a single LR unit?

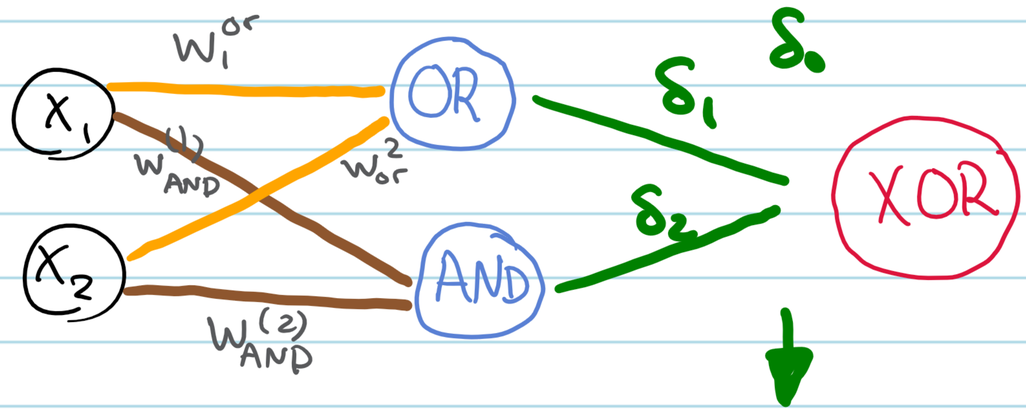
Machine learning
Sadegh Raeisi
Ch4- P5

For the XOR we need:

Is it possible with a single unit?



What's the solution? One idea is to use more than one unit.



x_1	x_2	OR	AND	XOR
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

$$\delta_1(OR) + \delta_2(AND) + \delta_0 \geq 1/2$$

$$1 \qquad -2 \qquad 0.5$$

The idea is that a single unit cannot sometimes directly give some desired outcome, but we can combine multiple units to get the result we want.

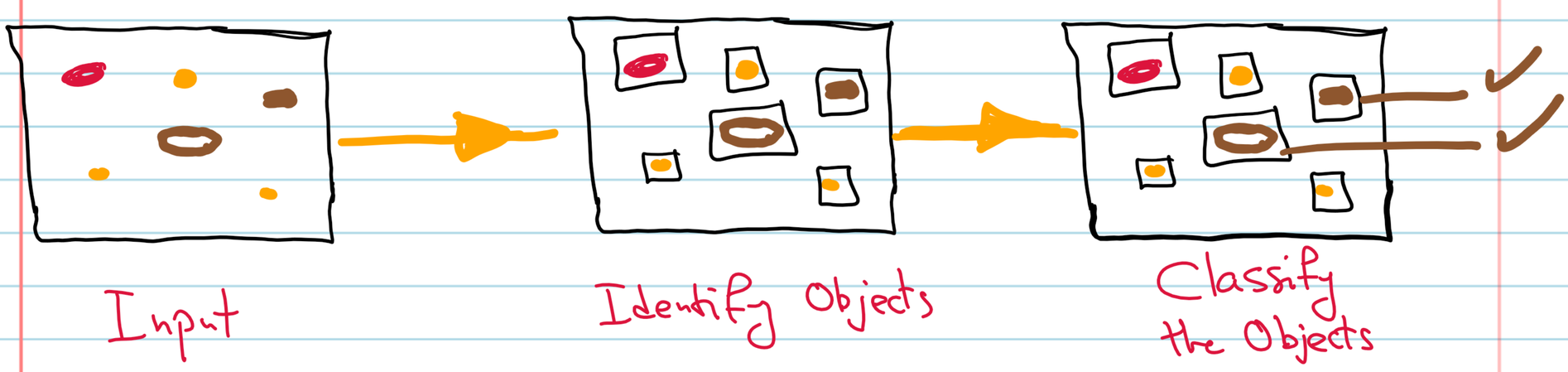
Also not that the gates AND & OR are universal which suggests that any logical gate can be built with combinations of different units in layers.

Hidden layers:

The units in the middle, between the input & output are called Hidden layers. These serve as middle calculations that help get to the final output.

A more intuitive example:

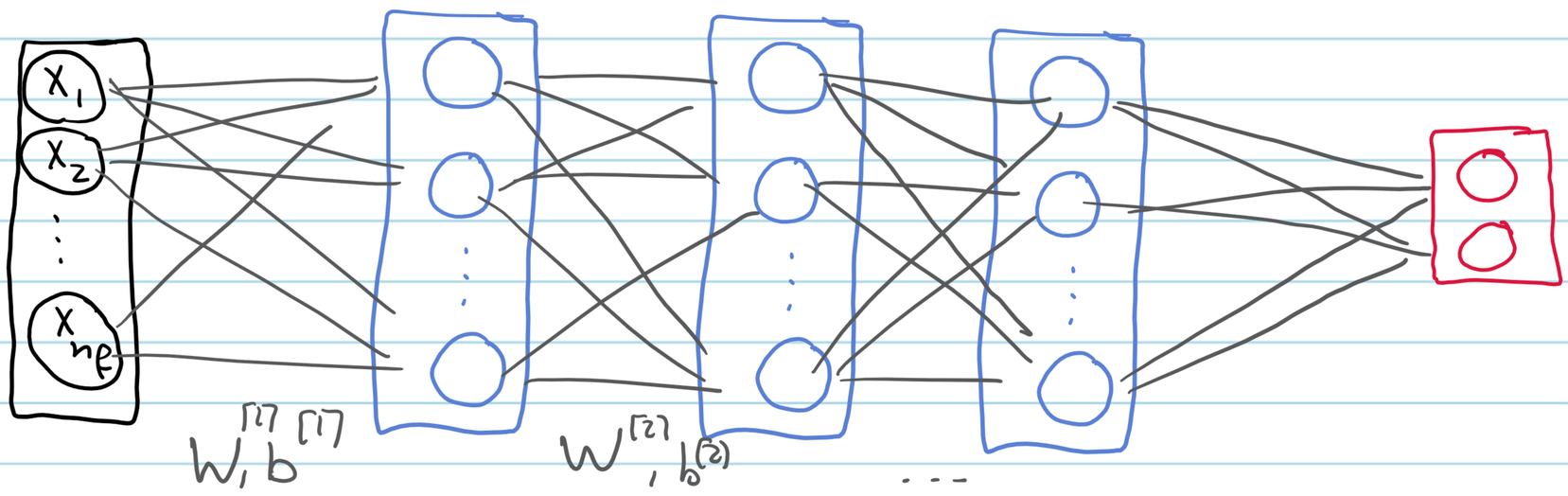
Imagine you want to find a specific type of galaxy in an image



↑
This is a hidden layer.
We're not interested in the results of this layer, but we use it to get to our final output.

Machine learning
Sadegh Raeisi
Ch4- P7

NN: Definition



Each circle is a node and is similar to a LR with respect to the proceeding layer.

Notation

$\vec{X}^{(i)}$: Input $\rightarrow$ i^{th} sample. $\vec{X}^{(i)} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n_f} \end{pmatrix}$ X : shape: (n_f, n_s)

Layer 1

$W^{[1]}$ $\rightarrow$ Weights for connecting inputs to the 1st layer. : shape: (n_f, n_{h1})

$b^{[1]}$ $\rightarrow$ Biases for the first Hidden layer. : Shape $(n_{h_1}, 1)$

$Z^{[1]}$ $\rightarrow$ Linear part of the 1st layer: $Z^{[1]} = \left(W^{[1]}\right)^T . X + b^{[1]}$

$A^{[1]}$ = Activation function for the 1st layer. e.g. Sigmoid.

$a^{[1]} = A^{[1]}\left(Z^{[1]}\right)$ $\rightarrow$ The outputs of the 1st layer.

* Broad Casting

Machine learning
Sadegh Raeisi
Ch4- P8

Notation

		Shape
* Superscript $\bullet^{[l]}$ index of the layer		
* Inputs X		(n_f, n_s)
Outputs $\tilde{Y}$ or Y		$(1, n_s)$
Hidden layers output a		(n_l, n_s)
$W^{[l]}$: Weight connecting the outcomes of layer $l-1$ to l.		(n_{l-1}, n_l)
$b^{[l]}$: The Bias for the layer l.		(n_l, n_s)
$Z^{[l]}$: linear outcome of layer l		(n_l, n_s)

$$Z^{[l]} = \left(W^{[l]}\right)^T \cdot a^{[l-1]} + b^{[l]}$$

$A^{[l]}$: Activation function for layer l.

$a^{[l]}$: Outcomes of layer l : (n_l, n_s)

$$a^{[l]} = A^{[l]}\left(Z^{[l]}\right) = A^{[l]}\left(\left(W^{[l]}\right)^T \cdot a^{[l-1]} + b^{[l]}\right).$$

Question: Implement this function, such that given $\left(W^{[l]}, b^{[l]}, A^{[l]}\right)$ it outputs the function above.

Activation Functions and non-linearity

Question: What would happen if we $A^{[l]}(x) = x \quad \forall l$?

$$a^{[l]} = Z^{[l]} = \left(W^{[l]}\right)^T \cdot a^{[l-1]} + b^{[l]}$$

$$= \left(W^{[l]}\right)^T \cdot \left(W^{[l-1]^T} \cdot a^{[l-2]} + b^{[l-1]} \right) + b^{[l]}$$

$$= \dots$$

$$= W^{[l]^T} \cdot W^{[l-1]^T} \cdots X + \left(b^{[l]} + W^{[l]} b^{[l-1]} + \dots \right)$$

$$= \Omega^T \cdot X + B$$

The whole network reduces to a linear reg. model.

So w/o the activation functions, it would not be the powerful model we need and the depth does not help.

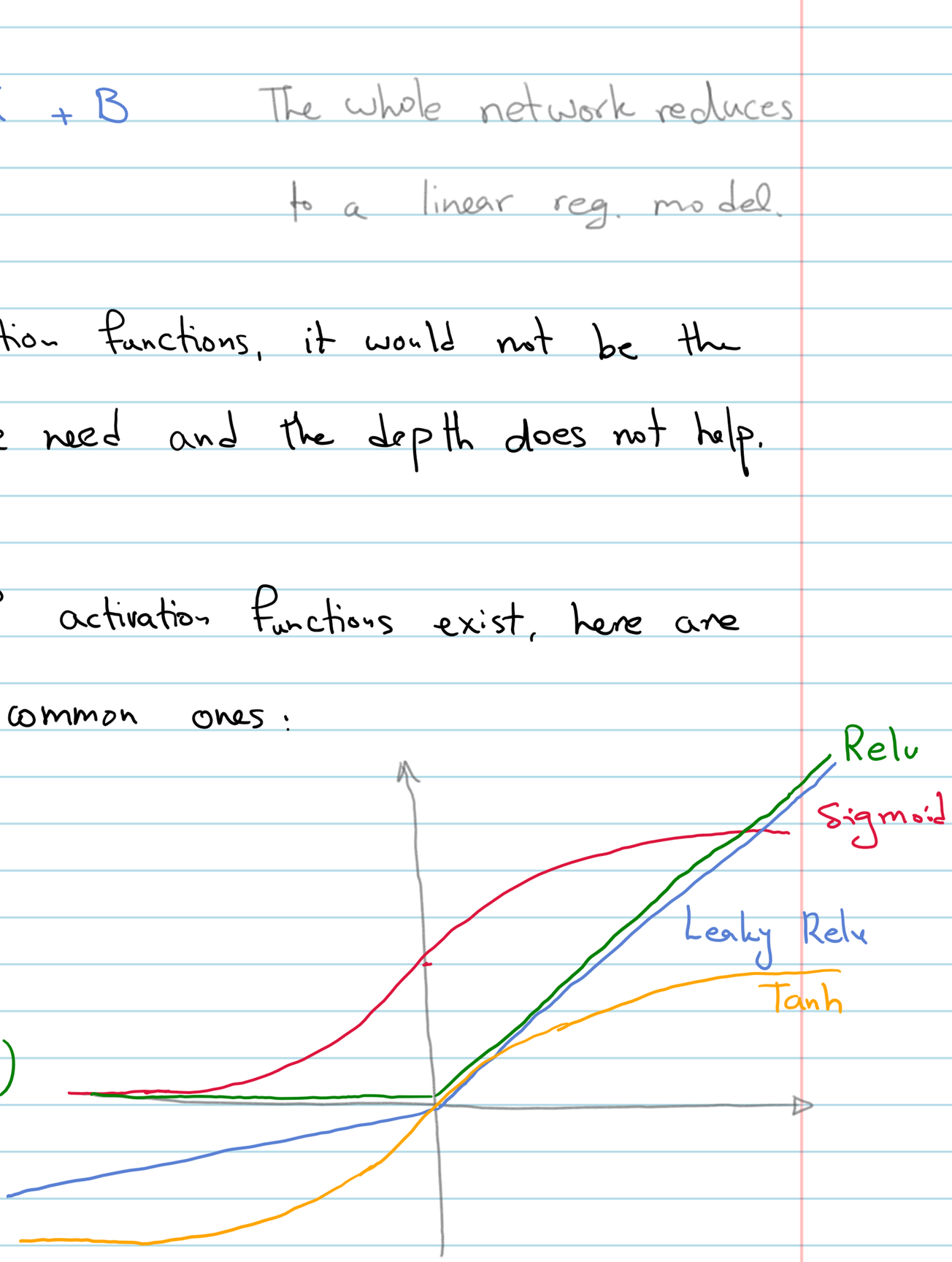
Different choices of activation functions exist, here are some of the more common ones:

Sigmoid: $A(x) = \frac{1}{1+e^{-x}}$

Tanh: $A(x) = \text{Tanh}(x)$

ReLU: $A(x) = \max(0, x)$

Leaky-ReLU: $A(x) = \max(0.1x, x)$



The choice of activation function is one of the hyperparameters that depending on application, could change.

An obvious example is that for a classification problem, i.e. $y \in \{0, 1\}$, the activation for the output layer cannot be ReLU.

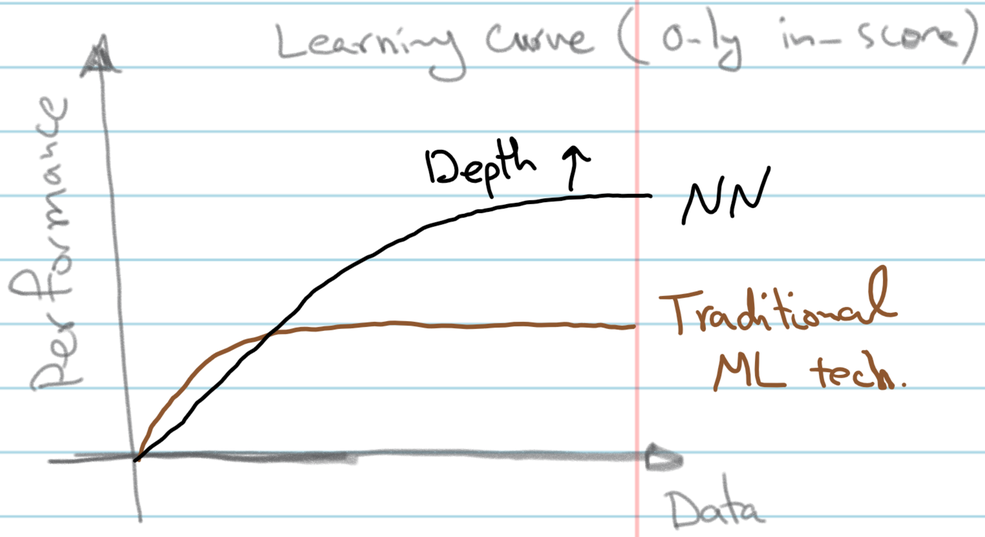
But, there's more to it. Often, the activation function can significantly affect the training time.

What's so good about NN?

- Probably the main advantage is the tunable complexity of NN.

For large enough data, NN could achieve higher performances.

That is to say the bias of the models reduces with more layers.



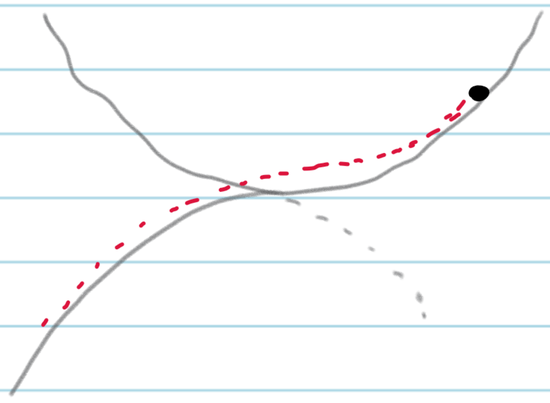
What should scare us

- Too large, to many paran

- Local minima problem

As there are too many param in the model ($\sim 10^4 - 10^7$), it is unlikely to find local minimum.

There are often saddle points and the optimization does not get stuck.



* ...

Disadvantage:

Require too much data (often) and there are too many params to optimize over.

Universality :

Universal approx. Theorem.

* With enough width($O(\exp n_p)$), single layer NN can approximate continous functions on compact subsets of $\mathbb{R}^n$.

* With enough depth with width as few as $n+c$ it would be possible to approximate wide classes of functions. (See relevant refs for more details).

→ It is interesting that $O(\exp n)$ width reduces to linear by allowing more depth.

There are also some results on the role of the activation functions. For instance, it is known that NN are only universal if the activation function is not a polynomial function [1].

[1] Leshno, Moshe; Lin, Vladimir Ya.; Pinkus, Allan; Schocken, Shimon (January 1993). "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function". Neural Networks. 6 (6): 861–867. doi:10.1016/S0893-6080(05)80131-5

Here we give an intuition to why and how the feedforward function can universally approximate continoues functions.

But first, we need to make some remarks.

Remarks

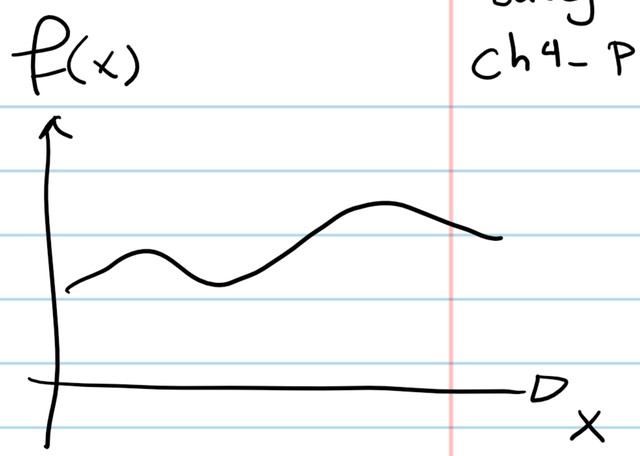
* The universality means that the feedword function, with enough width & depth is capable to approximate a function, i.e. it can get arbitrarily close. But it does not have to reconstruct the exact function.

* Often, the universality theorems specify classes of functions that the feedforward function can approximate, e.g. Continous functions. So they cannot approximate functions outside the specified sets good enough.

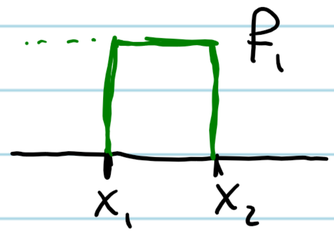
Intuition of universality

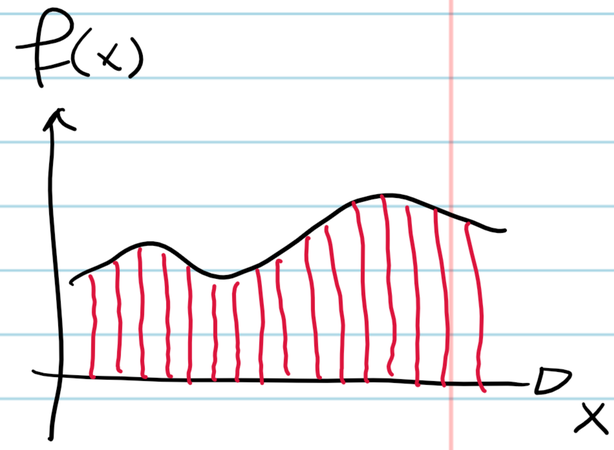
Let's say we want to approximate $f(x)$.

We can break this into small intervals.

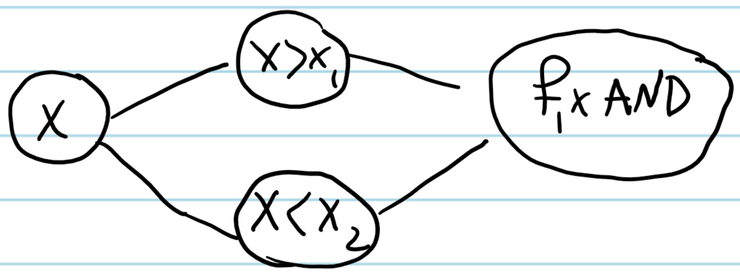


Then for each interval, we have





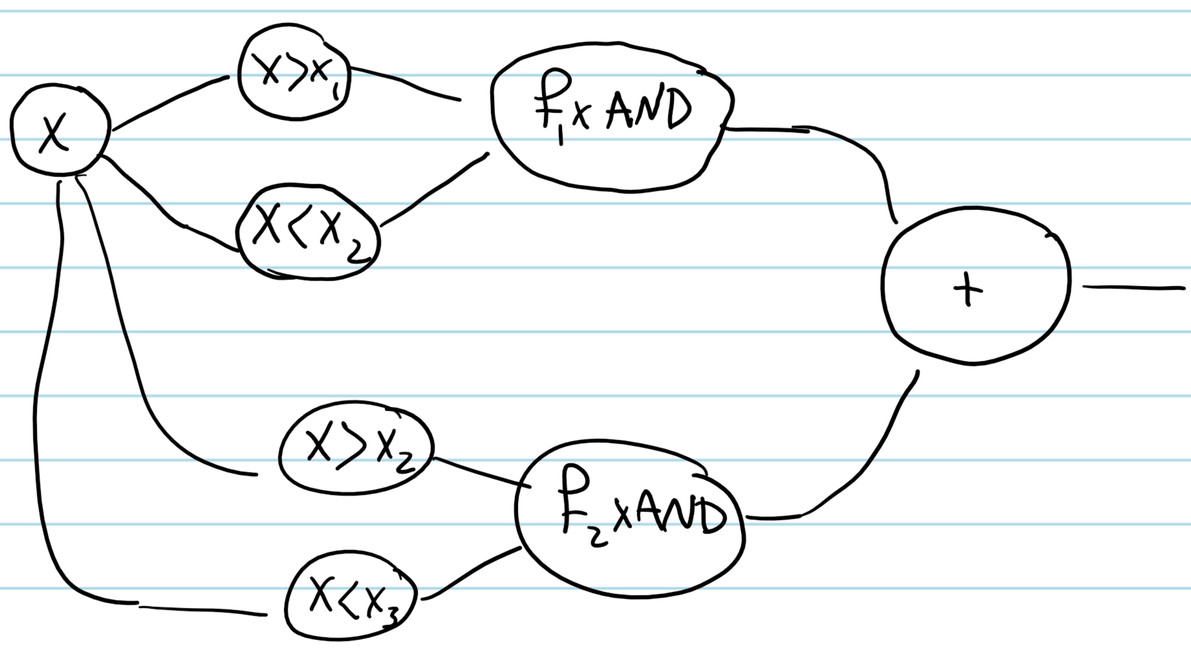
We can break the whole function to a combination of these pieces. For each part, we have:

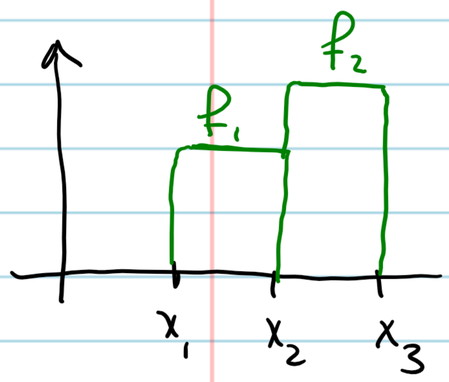


Outside $[X_1, X_2]$ → AND=0 → 0

Inside $[X_1, X_2]$ → AND=1 → f_1

This makes a single unit. To make the full function, we need to sum these units. for that, we can use just add them up:





Similarly with more nodes, we can make the rest of the function.

Also, to improve the approx. we would need to reduce the bin sizes.

This construction uses only one hidden layer, but clearly one increase the depth to create a tree that specifies each interval and this would reduces the required nodes exponentially.

See Nielson's book (ch 4) for more details.

Training : Gradient Descent

Let's say that we want to train an LR to fit $\{X, Y\}$.

We need a loss function. We take Cross-entropy :

$$L_{ce}(Y, \tilde{Y}) = -\sum_i Y_i \log \tilde{Y}_i + (1-Y_i)\log(1-\tilde{Y}_i)$$

For LR, we have $\tilde{Y} = \sigma(W \cdot X + b)$ $\quad W \to W^T$ (for simplicity)

Each step of the GD:

$$W \to W - \alpha \frac{\partial L_{ce}}{\partial W}$$

$$b \to W - \alpha \frac{\partial L_{ce}}{\partial W}$$

$$\frac{\partial L}{\partial W_i} = \frac{\partial L}{\partial \tilde{Y}} \frac{\partial \tilde{Y}}{\partial W_i} = \dots \text{(Chain rule)} = \left(\frac{-Y}{\tilde{Y}} + \frac{1-Y}{1-\tilde{Y}}\right) \frac{\partial \tilde{Y}}{\partial W_i}$$

$$\frac{\partial \tilde{Y}}{\partial W_i} = \frac{\partial \tilde{Y}}{\partial z} \frac{\partial z}{\partial W}$$

$$\frac{\partial \tilde{Y}}{\partial z} = \frac{\partial \sigma(z)}{\partial z} = \sigma(z)(1-\sigma(z)) = \tilde{Y}(1-\tilde{Y}) \qquad (n_\ell, 1)$$

$$\frac{\partial L}{\partial W_i} = \underbrace{\left(\frac{-Y}{\tilde{Y}} + \frac{1-Y}{1-\tilde{Y}}\right) \tilde{Y}(1-\tilde{Y})}_{\left(-Y(1-\tilde{Y}) + \tilde{Y}(1-Y)\right) = (\tilde{Y} - Y)} (X_i) = (\tilde{Y} - Y) X_i \qquad (1, n_s) \quad (n_\ell, n_{\ell-1})$$

$$\frac{\partial L}{\partial b} = (\tilde{Y} - Y)$$