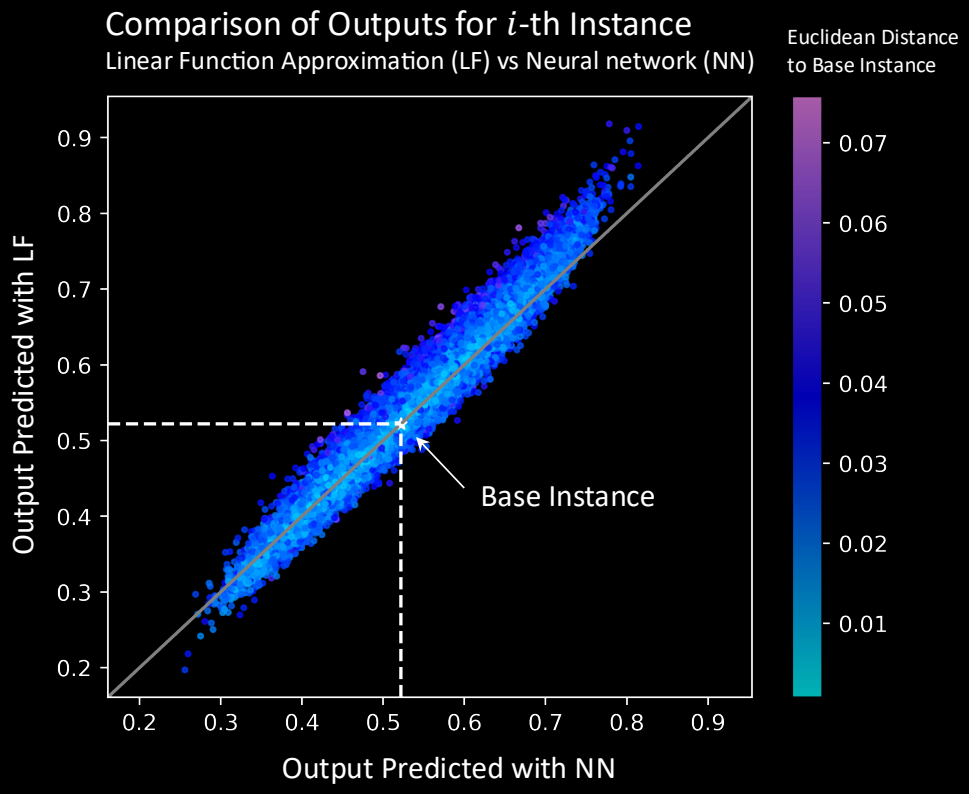
Front-Propagation Algorithm

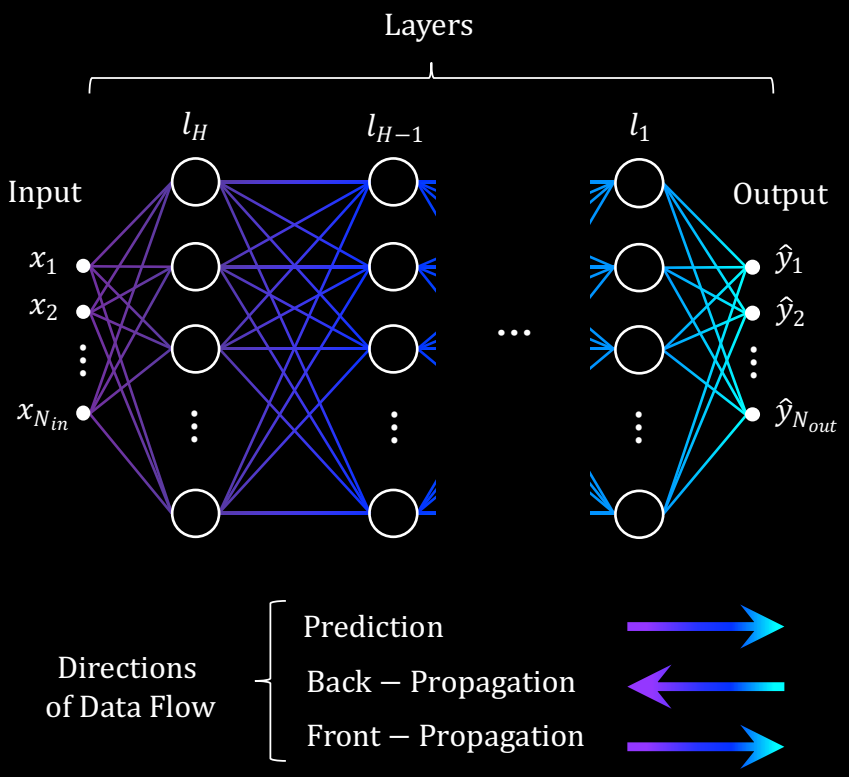
Rezon explains the reasoning of Neural Networks with an in-house developed algorithm that we called Front-Propagation.

The Front-Propagation algorithm is able to extract the linear function explanation that approximates the behavior of a Neural Network in the vicinity of a given instance (also called the base instance). In other words, the algorithm generates a linear function that can replace the Neural Network and still perform well for other datapoints that are close to the base instance. The plot below shows an example comparison in the outputs predicted with a given Neural Network versus the outputs obtained with the Linear Function approximation of this NN for a cloud of points near the base instance.



The information of the linear function approximation is very relevant in various applications, not only to quantify the contributions of each of the input dimensions, but also to determine if the network is following a biased reasoning, an outlier reasoning, to supervise its behavior, and to extract relevant knowledge of the problem.

To describe the inner workings of the algorithm, we first consider a simple feed-forward neural network with H hidden layers that performs a transformation from a N_{in}-dimensional input space $\mathbf{x} = \{x_1, x_2, \dots, x_{N_{in}}\}$ into a N_{out}-dimensional output space $\hat{\mathbf{y}} = \{\hat{y}_1, \hat{y}_1, \dots, \hat{y}_{N_{out}}\}$.



The objective of the Front-Propagation algorithm is to find the linear function that approximates the behavior of the neural network in the generation of a specific output. To do so, the algorithm traverses all the layers in a forward-pass (from input to output), studying all the parameters of the network.

What is the difference between Front-Propagation and Back-Propagation?

- The Back-Propagation algorithm calculates the gradient of the loss function with respect to the weights and biases, and then corrects those parameters to optimize the network.

- The Front-Propagation algorithm on the other hand calculates the gradient of the outputs with respect to the inputs, which then serve as an explanation of the network's reasoning.

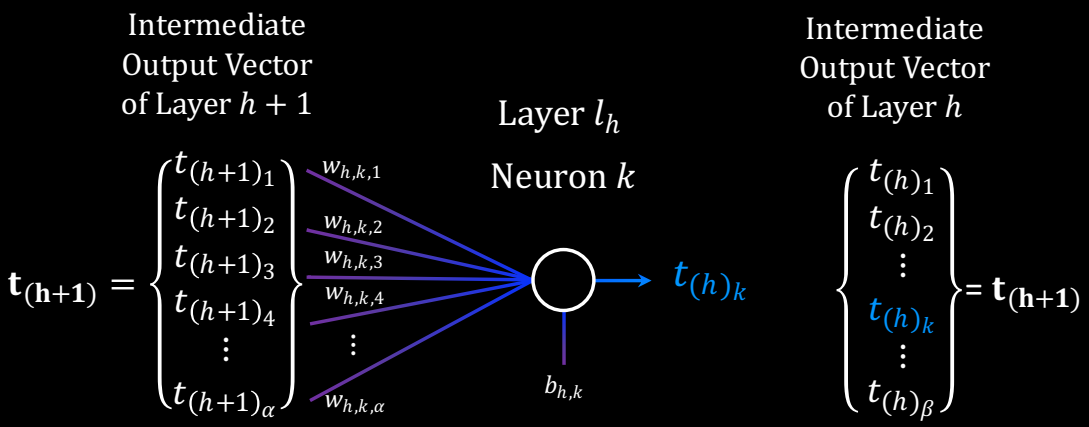
Both the Back and Front propagation algorithms are triggered after the network has made a prediction. In the first case to correct and in the second to understand the network.

The Back-Propagation traverses the network from the output to the input, but the Front-Propagation traverses the network from the input to the output.

The name Front-Propagation was inspired on these similarities with the Back-Propagation and the fact that studies the network in the opposite direction.

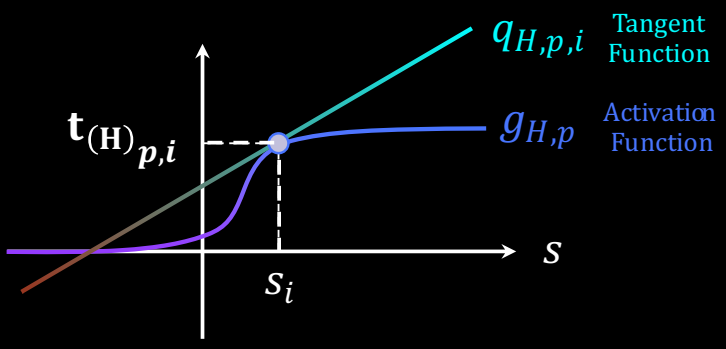
How does it work?

Let 's consider a generic k-th neuron of an arbitrary intermediate layer l_h. The input sent to this neuron is the sum of the product between $\mathbf{t_{(h+1)}}$ and $\mathbf{w_{h,k}}$ plus the bias. Where $\mathbf{t_{(h+1)}}$ represents the intermediate output vector of the previous layer (l_{h+1}), and $\mathbf{w_{h,k}}$ the weight vector of the neuron of interest. The output of the neuron, denoted by $\mathbf{t_{(h)_k}}$, is obtained by mapping this value with the activation function $g_{h,k}$. In formulation, $\mathbf{t_{(h)_k}} = g_{h,k}\left(b_{h,k} + \sum_{j=1}^{\alpha} w_{h,k,j} \cdot t_{(h+1)_j}\right)$.



If the neuron chosen is the p-th neuron of the layer l_H, then $\mathbf{t_{(H)_p}} = g_{H,p}\left(b_{H,p} + \sum_{j=1}^{N_{in}} w_{H,p,j} \cdot x_j\right)$. For the Front-Propagation, instead of substituting x_j with the corresponding values of the instance studied, we preserve x_j as a placeholder variable that identifies the j-th input dimension. The goal is to find the linear dependencies that relate the input and the output spaces. That is why we do not want to substitute $\mathbf{x}$ until we have finished traversing the entire network.

At this point, the argument $s(\mathbf{x})$ of $g_{H,p}(s(\mathbf{x}))$ is in fact a linear combination of the input dimensions, but the activation function introduces a non-linearity. Thus, to find the linear dependencies between $\mathbf{t_{(H)_p}}$ and the input dimensions x_j, we find the tangent linear function that approximates the activation function for the i-th instance studied.



We then plug the linear function $s(\mathbf{x})$ inside the tangent linear function of $g_{H,p}$, which we denote as $q_{H,p,i}(s)$. Note that $q_{H,p,i}(s)$ depends on the i-th instance chosen. The result is $r_{H,p}(\mathbf{x})$, a new linear function that relates the output of this neuron with the input dimensions,

$$r_{H,p}(\mathbf{x}) = q_{H,p,i}(s_{H,p}(\mathbf{x}))$$

This process can be done for all the neurons of the layer. Once its finished, we then aggregate all the output linear functions $r_{H,p}(\mathbf{x})$ with the corresponding weights of the layer in order to generate the next input linear functions $s_{H-1,k}(\mathbf{x})$ for every k-th neuron of the next layer,

$$s_{H-1,k}(\mathbf{x}) = b_{H-1,k} + \sum_{j=1}^{\alpha} w_{H-1,k,j} \cdot r_{H,j}$$

Following this same strategy in every neuron of the network and regrouping the parameters we can obtain the linear output function of each neuron. Once we arrive at the output layer of the Neural Network we obtain the linear function that we were searching for.