Introduction to Deep Learning

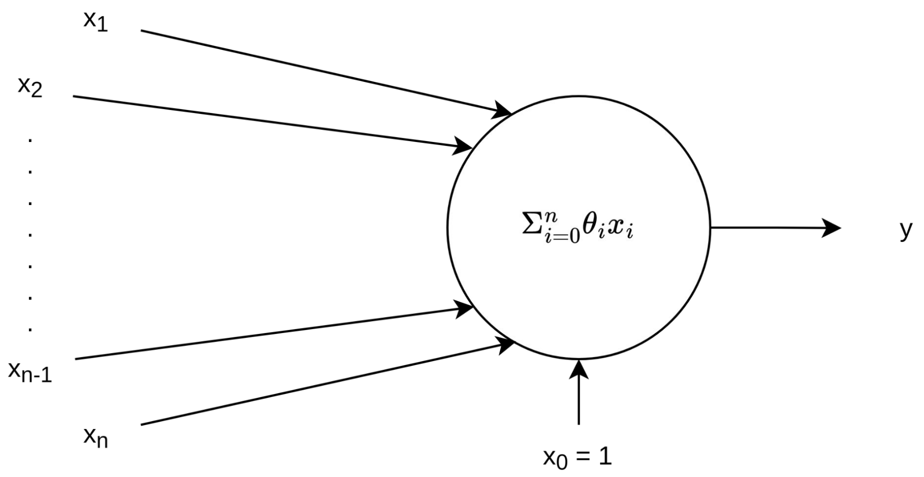
Sarthak Harne

Revisiting Linear Regression

Linear Regression (Multivariate) looks like this:

$$y = \theta^T x \text{ where } \theta = (\theta_n, \dots, \theta_1, \theta_0) \text{ and } x = (x_n, \dots, x_1, x_0)$$

This can also be denoted pictorially by a 'neuron' as:

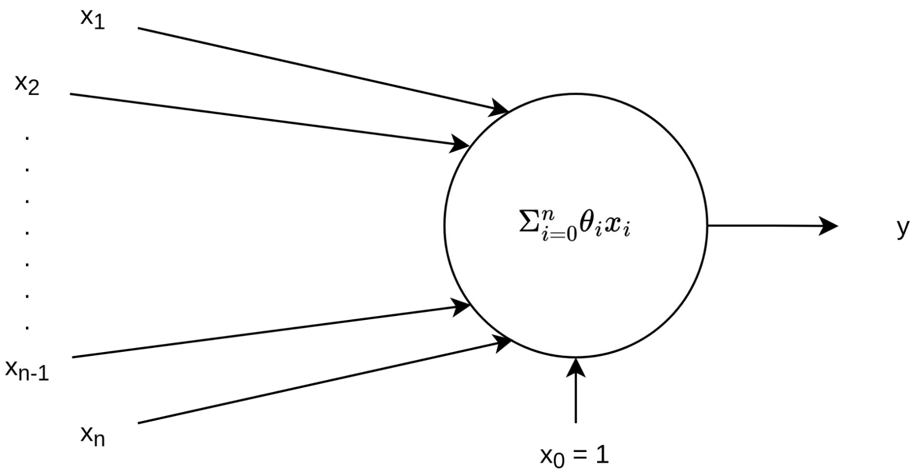


Revisiting Linear Regression - Neuron

The weights are a property of the neuron. For n inputs (including bias), there will be n weights which can be represented as a vector θ

The idea is to add more weights/parameters to learn more complex patterns from the data.

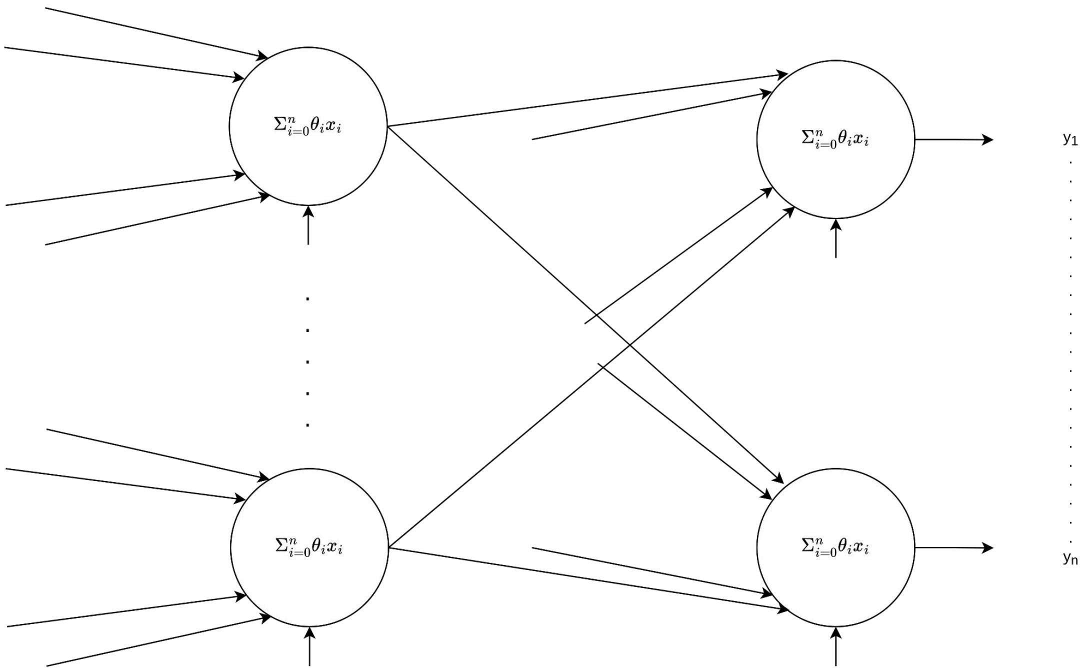
This can be done via adding more layers.



$$y = \theta^T x$$

y_i^l is the output of the ith neuron in the lth layer

Adding more layers

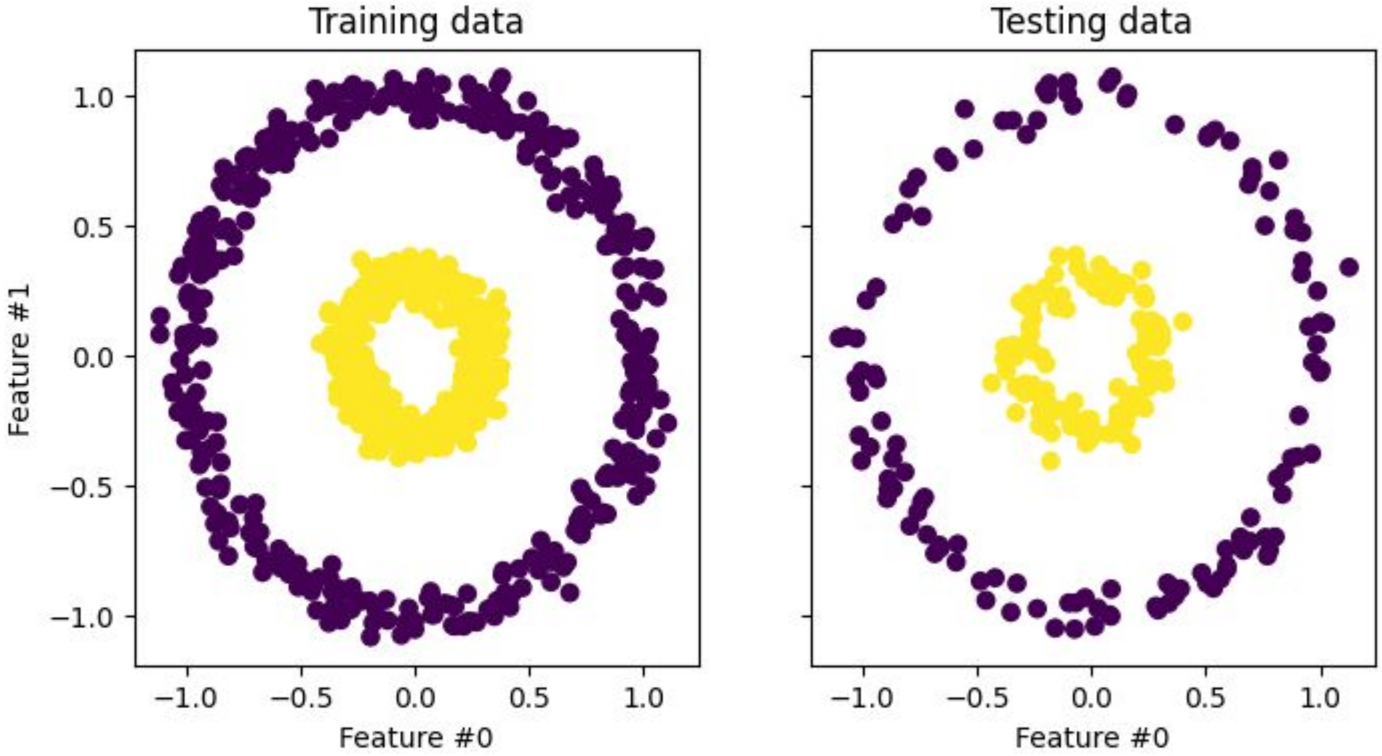


Having a multi-layer setup with no non linearity will just give us another linear output.

This is because adding two linear functions just gives us another linear function

Why do we need Non-Linearity?

Linear functions will not give us decision boundaries which are suitable for distributions like these

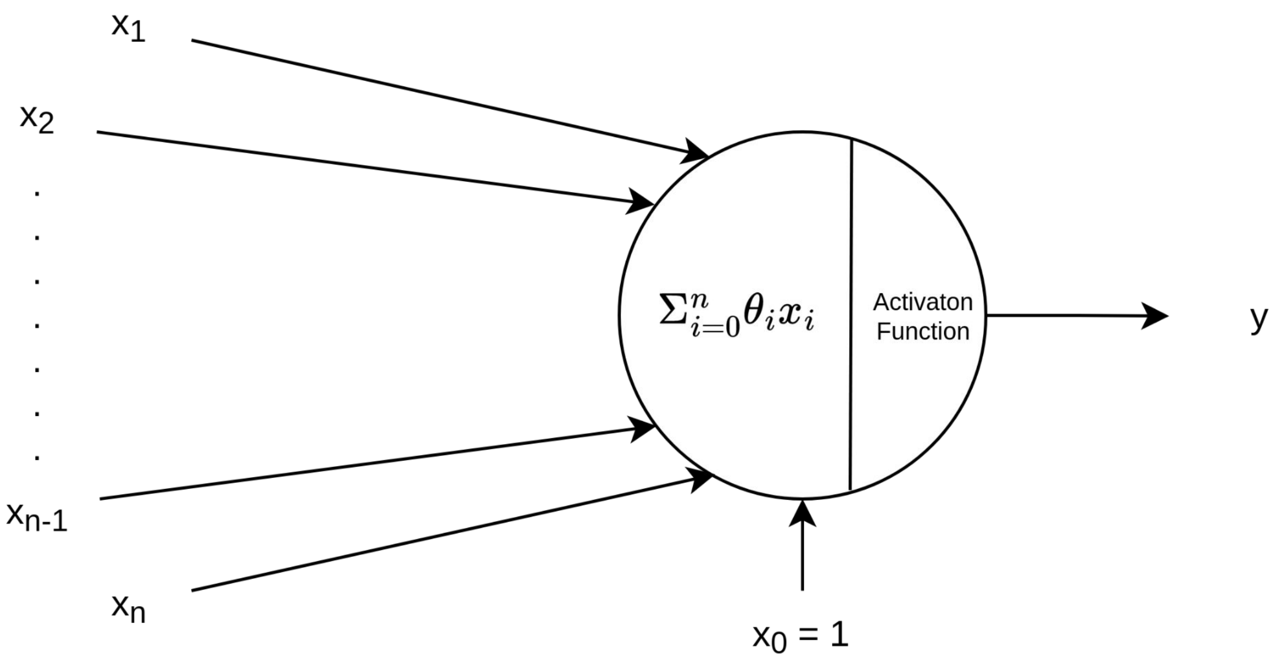


Non-Linearity in a Neuron

We introduce nonlinearity using 'activation functions'.

These are nonlinear functions which are applied on the output of the linear part of the neuron.

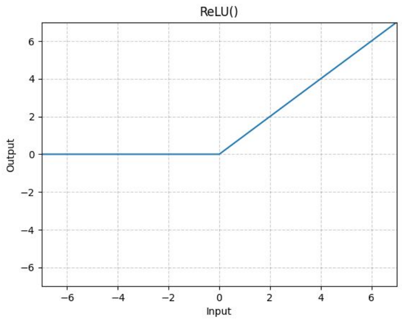
Now the sums across different layers becomes non linear.



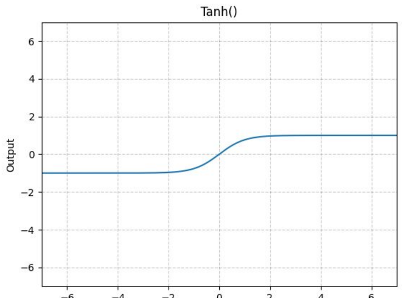
Activation Functions

These are simple nonlinear functions like:

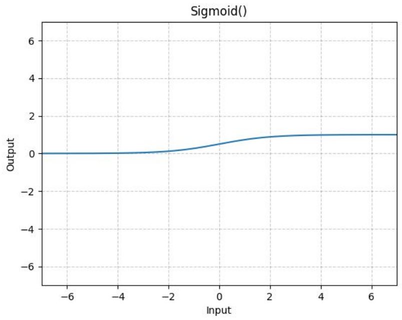
- ReLU
- Tanh
- Sigmoid



$$ReLU(x) = max(0, x)$$



$$tanh(x) = \frac{exp(x) - exp(-x)}{exp(x) + exp(-x)}$$



$$\sigma(x) = \frac{1}{1 + exp(-x)}$$

Images:
https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity

Neural Networks

When we stack these layers we get what is a ‘Feed Forward Network’ (FFN) and a layer with these neurons is known as a ‘Linear Layer’

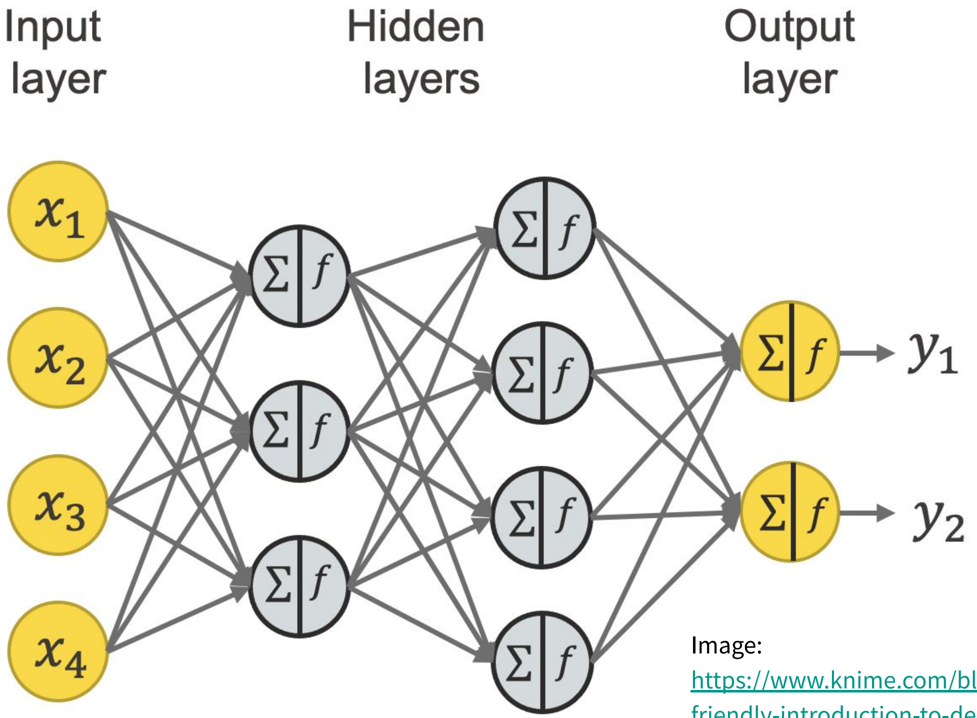


Image: https://www.knime.com/blog/a-friendly-introduction-to-deep-neural-networks

Neural Networks - Notation

For the layer L, neuron i∈[1, …, n] and input k∈[1, …,m], the weights are given as

$$w_i^L = ({}_1w_i^L, \ldots, {}_k w_i^L, \ldots, {}_n w_i^L)$$

This can be combined together as

$$W^L = \begin{bmatrix} \leftarrow (w_1^L)^T \rightarrow \\ \cdot \\ \cdot \\ \cdot \\ \leftarrow (w_n^L)^T \rightarrow \end{bmatrix}$$

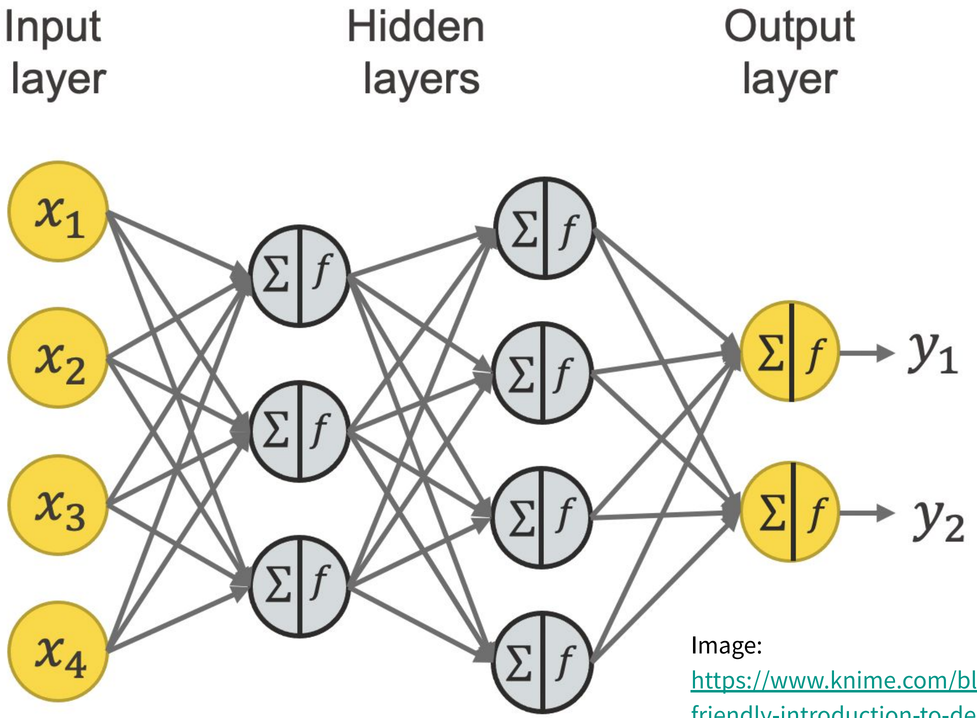


Image: https://www.knime.com/blog/a-friendly-introduction-to-deep-neural-networks

Neural Networks - Notation

For the layer L, neuron i∈[1, …, n] and input k∈[1, …,m]

The inputs are given as

$$x_i^{L-1} = (x_i^{L-1}, \ldots, x_i^{L-1}, \ldots, x_i^{L-1})$$

The linear output is given as

$$z^L = (z_1^L, \ldots, z_i^L, \ldots, z_n^L)$$

The activation output is given as

$$a^L = f(z^L)$$

Where f is the activation function

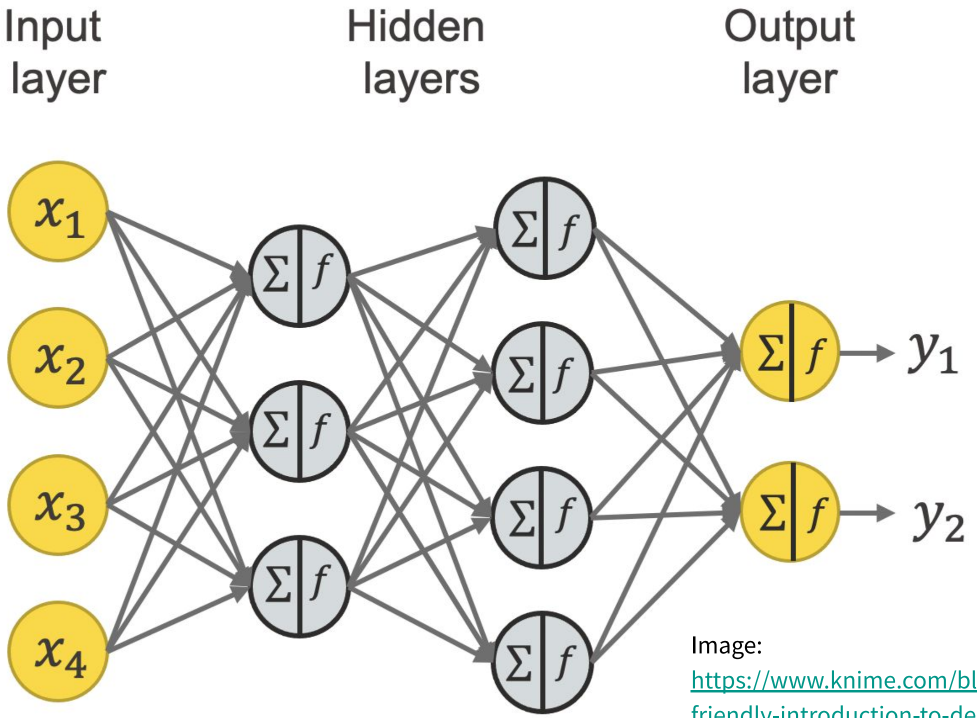


Image: https://www.knime.com/blog/a-friendly-introduction-to-deep-neural-networks

Neural Networks - Notation

For the layer L, neuron i∈[1, …, n] and input k∈[1, …,m]. The final equations are:

$$z^L = W^L x^{L-1}$$

$$a^L = f(z^L)$$

Input for the next layer $\rightarrow$ $$x^L = a^L$$

These are called the forward pass equations

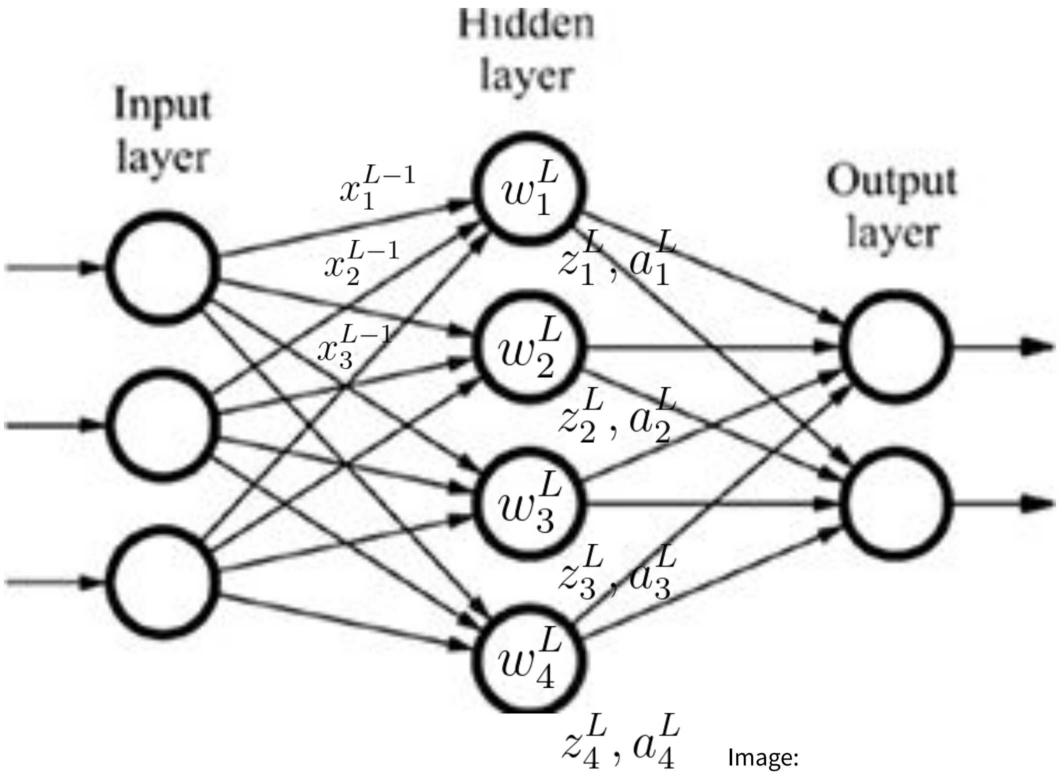


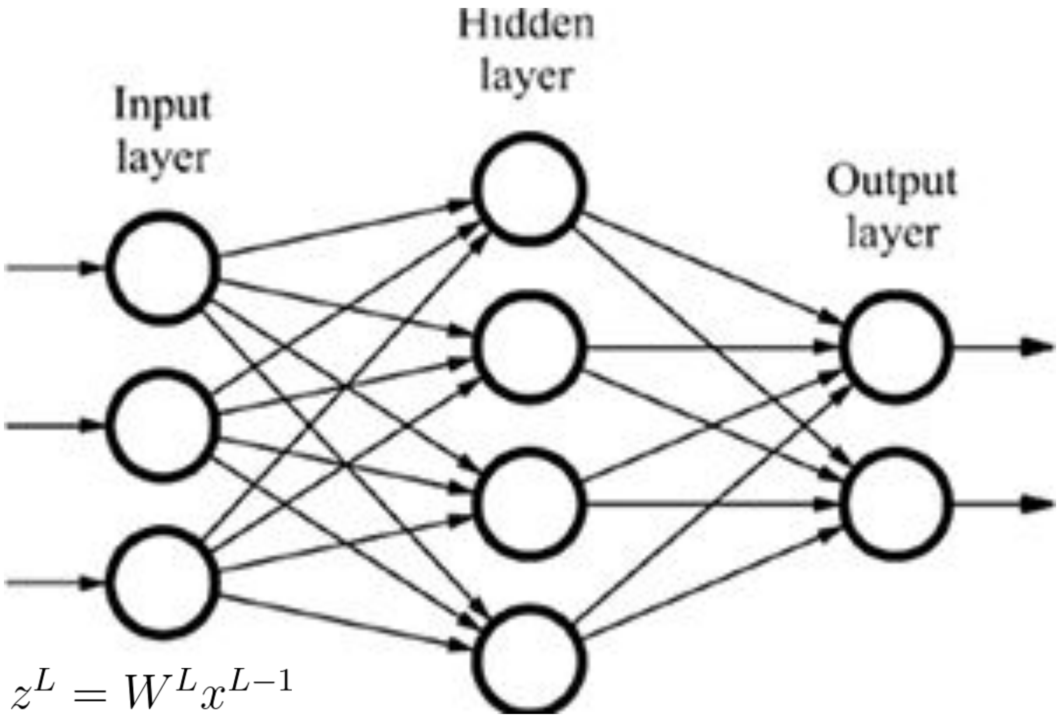
Image: https://www.databricks.com/glossary/neural-network

Neural Networks - Forward Pass

The equations are used to 'propagate' the input to get the output.

Generally, in the final layer, no activation is applied. So, the output becomes (here, L is the final layer number)

$$y = W^L x^{L-1}$$



$$z^L = W^L x^{L-1}$$
$$a^L = f(z^L)$$
$$x^L = a^L$$

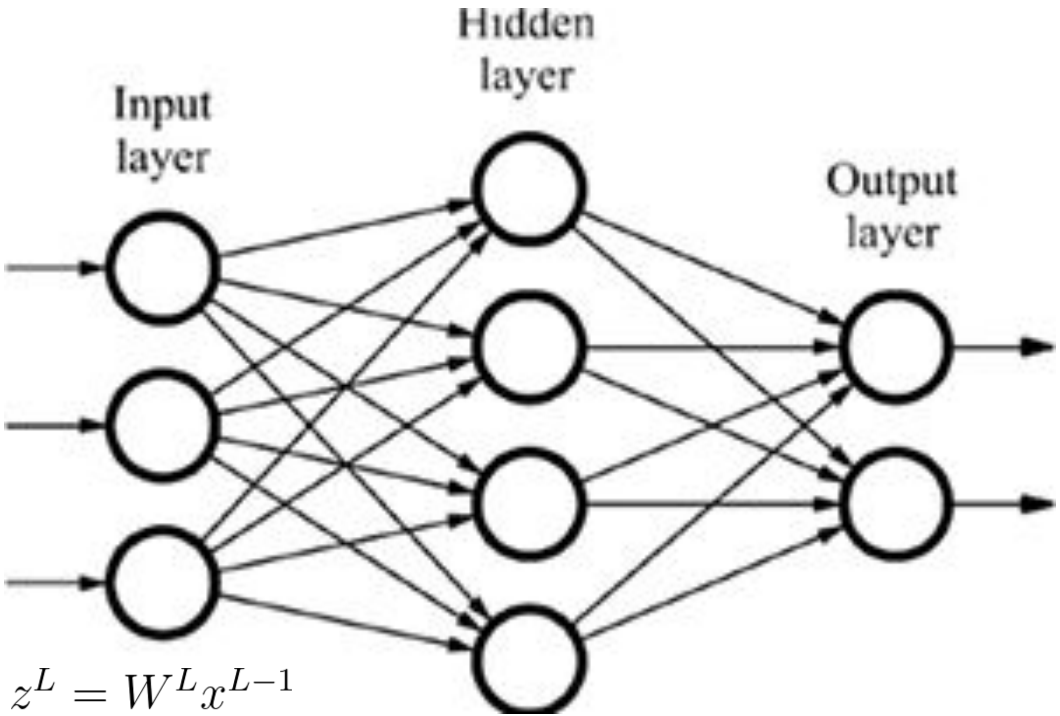
Image: https://www.databricks.com/glossary/neural-network

Neural Networks - Forward Pass

The equations are used to ‘propagate’ the input to get the output.

Generally, in the final layer, no activation is applied. So, the output becomes (here, L is the final layer number)

$$y = W^L x^{L-1}$$



$$z^L = W^L x^{L-1}$$
$$a^L = f(z^L)$$
$$x^L = a^L$$

Image: https://www.databricks.com/glossary/neural-network

Neural Networks - Backpropagation

The basic weight update equation is: $W = W - \alpha \frac{\partial L}{\partial W}$

The main issue with neural networks is calculating the derivative $\frac{\partial L}{\partial W}$

$$\frac{\partial l}{\partial W^L} = \frac{\partial l}{\partial a^L}\frac{\partial a^L}{\partial W^L}$$
$$= \frac{\partial l}{\partial a^L}\frac{\partial a^L}{\partial f}\frac{\partial f}{\partial W^L}$$
$$= \frac{\partial l}{\partial a^L}\frac{\partial a^L}{\partial f}\frac{\partial f}{\partial z^l}\frac{z^L}{\partial W^L}$$

$$L = l(y_{true}, y_{pred})$$
$$z^L = W^L x^{L-1}$$
$$a^L = f(z^L)$$
$$x^L = a^L$$

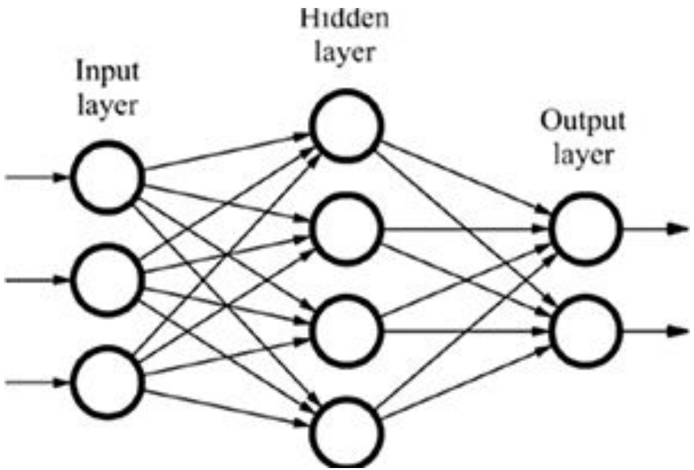


Image: https://www.databricks.com/glossary/neural-network

Neural Networks - Backpropagation

$$\frac{\partial l}{\partial W^L} = \frac{\partial l}{\partial a^L}\frac{\partial a^L}{\partial W^L}$$
$$= \frac{\partial l}{\partial a^L}\frac{\partial a^L}{\partial f}\frac{\partial f}{\partial W^L}$$
$$= \frac{\partial l}{\partial a^L}\frac{\partial a^L}{\partial f}\frac{\partial f}{\partial z^l}\frac{z^L}{\partial W^L}$$

$$L = l(y_{true}, y_{pred})$$
$$z^L = W^L x^{L-1}$$
$$a^L = f(z^L)$$
$$x^L = a^L$$

Parts of this function can be computed during the forward pass, which can make the backpropagation more efficient

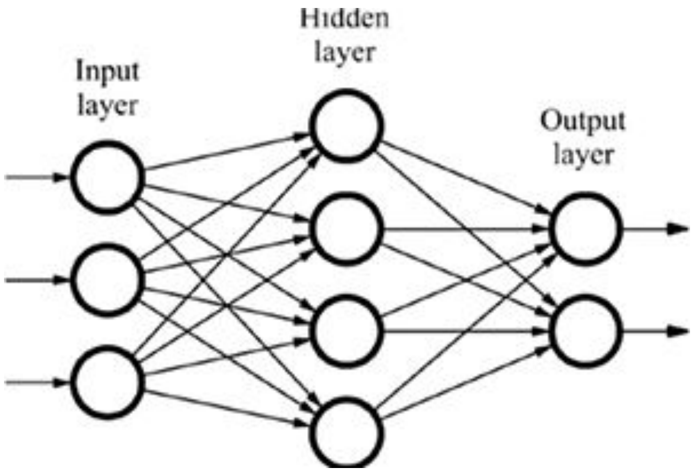
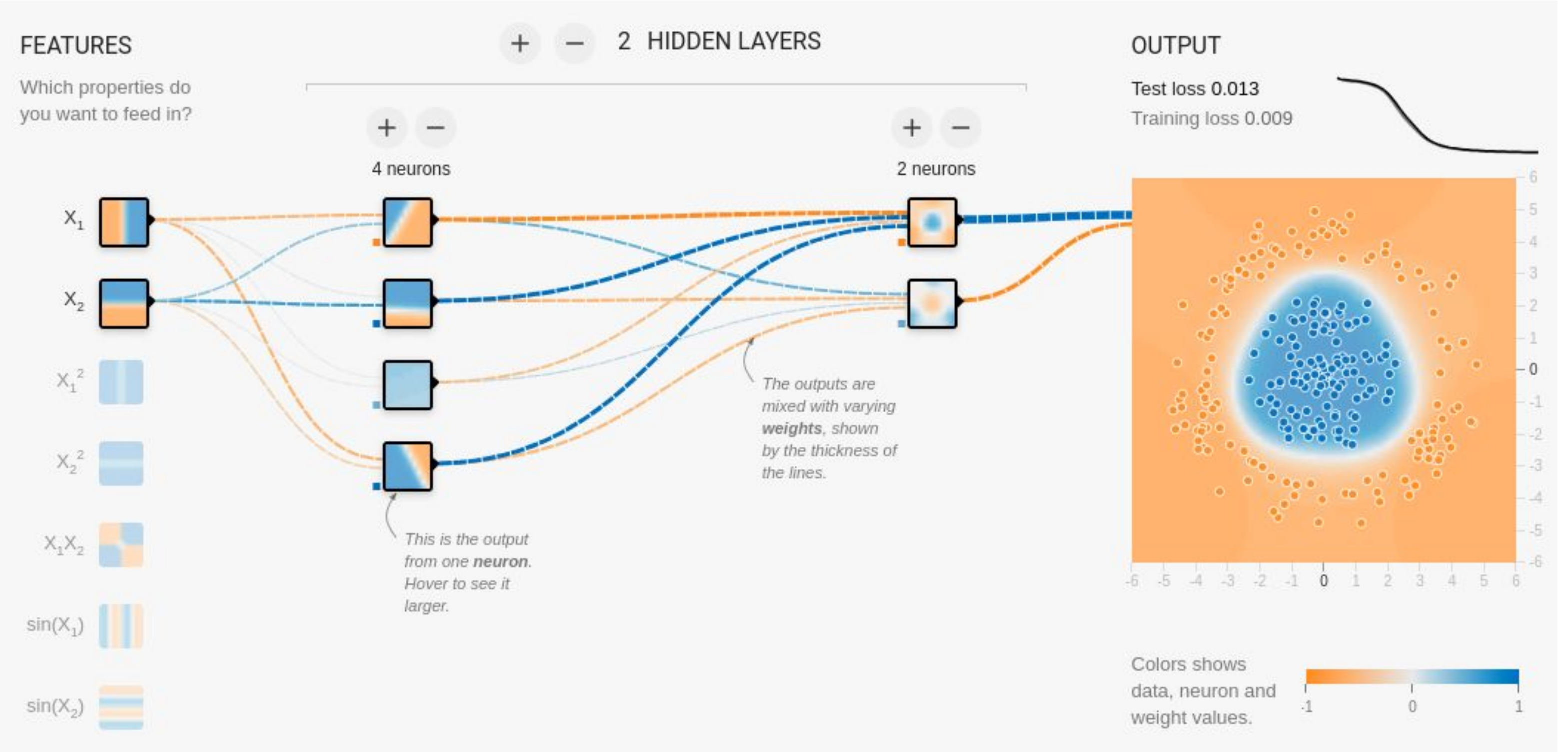


Image: https://www.databricks.com/glossary/neural-network

Neural Networks - Visualization



Source: https://playground.tensorflow.org/

PyTorch Tutorial

Kaggle Notebook: https://www.kaggle.com/code/sarthakharne/pytorch-mnist-example