ECS171: Machine Learning

L6
Supervised Classifiers

Instructor: Prof. Maike Sonnewald
TAs: Pu Sun & Devashree Kataria

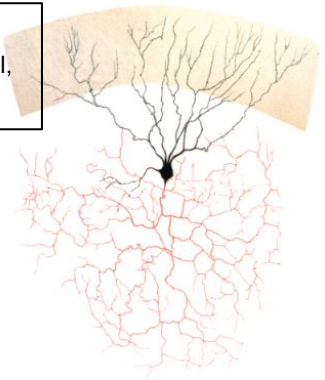

MOCO Amsterdam garden

Intended Learning Outcomes

- Discuss pros/cons of various activation functions and apply them in simple examples
- Discuss aspects of neural networks e.g. layers and nodes
- Describe and apply feature engineering methodologies such as normalisation and label encoding (one-hot and one-output)
- Describe and demonstrate the workings of a feed forward neural network in terms of how a single node is updated (linear combination of inputs and activation function)

The neuron

Golgi's drawing of a cerebellar Type II cell, now called a Golgi cell

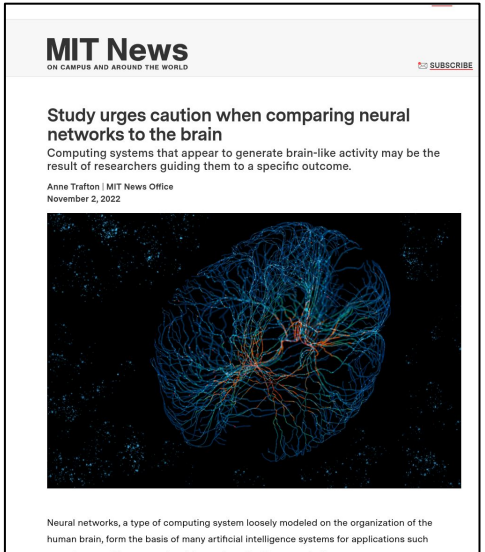


Camillo Golgi (Italy) 1843-1926

Santiago Ramon Cajal (Spain) 1852-1934

- Nobel prize 1906: Golgi (method) & Cajal (insights) for structure of the brain and spinal cord
- Functional localization: Different parts of e.g. the nervous system
- The neuron doctrine: The brain and spinal cord are made up of individual elements, called neurons.
  - Neurons touch one another, but don't fuse

Beware anthropomorphism



MIT News

ON CAMPUS AND AROUND THE WORLD

SUBSCRIBE

Study urges caution when comparing neural networks to the brain

Computing systems that appear to generate brain-like activity may be the result of researchers guiding them to a specific outcome.

Anne Trafton | MIT News Office
November 2, 2022

Neural networks, a type of computing system loosely modeled on the organization of the human brain, form the basis of many artificial intelligence systems for applications such

Anthropomorphism: the attribution of human characteristics or behavior to a god, animal, or object

Perceptron Learning Algorithm recapitulation

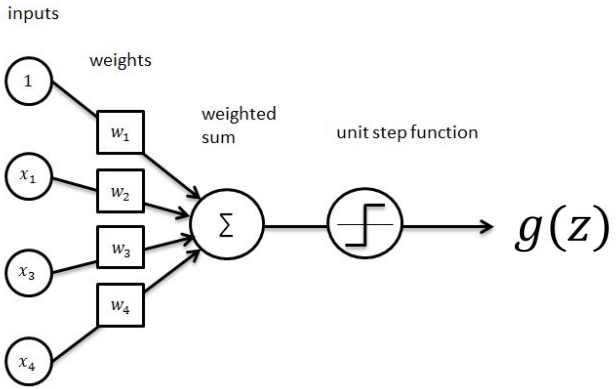
- The Perceptron Learning Algorithm is supervised learning that is good for binary classification
- The Perceptron model takes an input, aggregates it (calculates the weighted sum), and with the step function returns 1 if this is more than a threshold or 0 if it is equal or below

$$w_j = w_j + a\left(y^{(i)} - g\left(x^{(i)}; w\right)\right)x_j^{(i)}$$

$$w_j = w_j + a\left(y^{(i)} - g(z)\right)x_j^{(i)}$$

$$z = w^T x = w_0 + w_1 x_1 + w_2 x_2$$

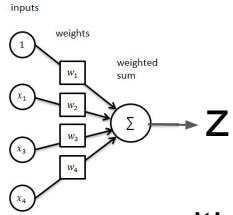
$$g(z) = \begin{cases} 0 \; if \;\; z < threshold \\ 1 \; if \;\; z \geq threshold \end{cases}$$



Example: Perceptron

- Goal: Get perceptron to predict classes from data that we can encode as 1 or 0 (e.g. pass/fail from continuous data of hours studied and slept)
- If we had a **regression problem**, we would have continuous labels (not just 1 and 0), and would build a model etc:

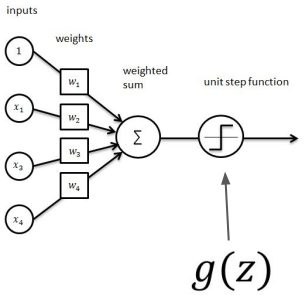
Studied	Slept	Passed
4.85	9.63	1
8.62	3.23	0
5.43	8.23	1
9.21	6.34	0



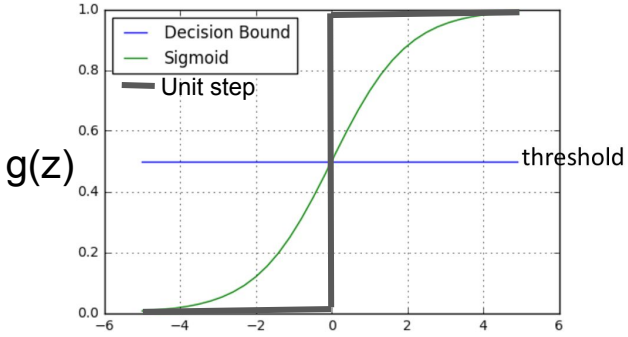
$$z = \mathbf{w}^T\mathbf{x} = w_0 1 + w_1 x_1 + w_2 x_2 + w_3 x + w_4 x_4$$

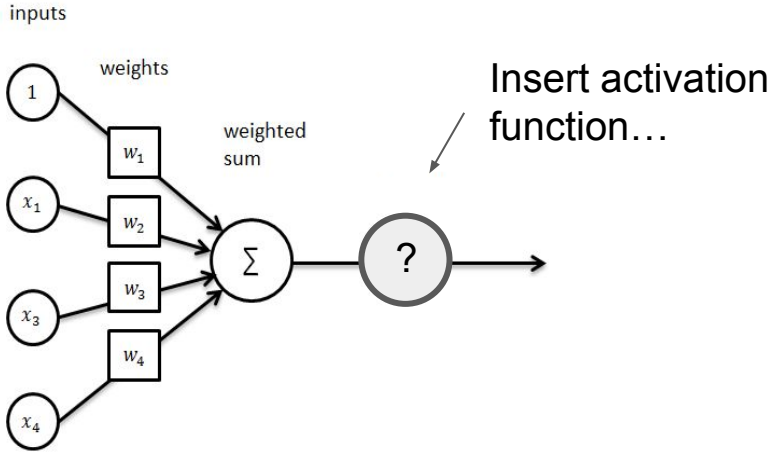
- However, with **classes** as labels we use a **threshold** encoded as an **activation function** g(z) in the perceptron, that takes z (providing continuous output) and returns a class:

$$g(z) = \begin{cases} 0 \; if \;\; z < threshold \\ 1 \; if \;\; z \geq threshold \end{cases}$$



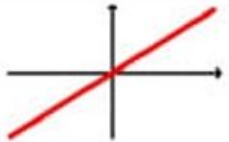
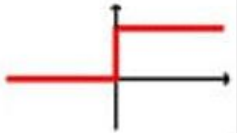
Changes in the activation function





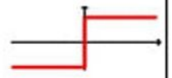
- The ‘activation function’ g(z) is what translates the z into the classes.
- We can change the function used in the activation function

Different activation functions we've seen…

Activation Function	Equation	Example	1D Graph
Linear	$\phi(z) = z$	Adaline, linear regression	
Unit Step (Heaviside Function)	$\phi(z) = \begin{cases} 0 & z < 0 \\ 0.5 & z = 0 \\ 1 & z > 0 \end{cases}$	Perceptron variant	
Sign (signum)	$\phi(z) = \begin{cases} -1 & z < 0 \\ 0 & z = 0 \\ 1 & z > 0 \end{cases}$	Perceptron variant	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multilayer NN	

Activation functions continued.

- **Non-linearity in Neural Networks is introduced using Activation Functions**
- Allows learning complex patterns in data
- Determine the output of a NN
- Determine whether a neuron should “fire”
- Non-Linear activation functions help the model to generalize to a variety of data
- **A NN without an activation function is a linear regressor**

Activation Function	Equation	Example	1D Graph
Linear	$\phi(z) = z$	Adaline, linear regression	
Unit Step (Heaviside Function)	$\phi(z) = \begin{cases} 0 & z < 0 \\ 0.5 & z = 0 \\ 1 & z > 0 \end{cases}$	Perceptron variant	
Sign (signum)	$\phi(z) = \begin{cases} -1 & z < 0 \\ 0 & z = 0 \\ 1 & z > 0 \end{cases}$	Perceptron variant	
Piece-wise Linear	$\phi(z) = \begin{cases} 0 & z \le -½ \\ z + ½ & -½ \le z \le ½ \\ 1 & z \ge ½ \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multilayer NN	
Hyperbolic Tangent (tanh)	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multilayer NN, RNNs	
ReLU	$\phi(z) = \begin{cases} 0 & z < 0 \\ z & z > 0 \end{cases}$	Multilayer NN, CNNs	

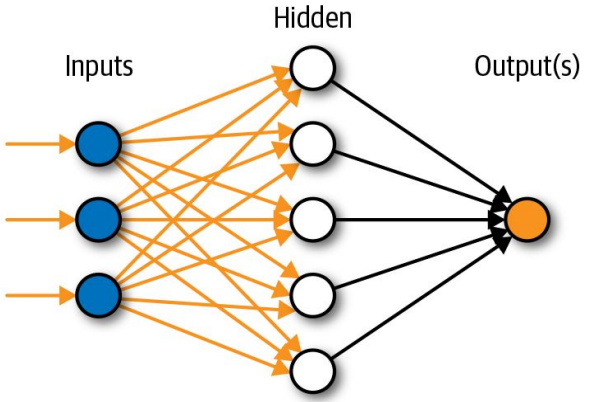
More on activation functions…

Formula	Output Range	Advantages	Disadvantages
Sigmoid $f(x) = 1 / (1 + e^{-x})$	0 to 1	• output a probability • Simple to compute	can cause vanishing gradient problem which can slow down training in deep networks (DNN)
ReLU $f(x) = \max(0, x)$	0 to $+\infty$	• Simple to compute • Non-saturating	can cause dying ReLU problem (neurons that output 0 for any input value < 0)
Leaky ReLU $f(x) = \max(\alpha x, x)$	$-\infty$ to $+\infty$	• Solves the dying ReLU problem • Non-saturating	
Tanh $f(x) = (e^{x} - e^{-x}) / (e^{x} + e^{-x})$	-1 to 1	• Useful when you need to output a value between -1 and 1 • mainly used for binary classification	can cause vanishing gradient problem which can slow down training in DNNs
Softmax $f(x_i) = \exp(x_i) / \Sigma(\exp(x_i))$	0 to 1	Useful to get output probabilities for multiple classes.	Can suffer from numerical instability.
Softplus $f(x) = \log(1 + e^{x})$	0 to $+\infty$	• Similar to ReLU, but smoother and differentiable everywhere, which can make it easier to optimize. • non-saturating • simple to compute.	Can cause vanishing gradient problem for large negative input values, which can slow down training in deep networks.
Swish $f(x) = x/(1 + e^{-\text{beta} * x})$	$-\infty$ to $+\infty$	• Can improve performance compared to ReLU and sigmoid. • Simple to compute.	Requires tuning the beta parameter.

Note: More details than you need in L6. We will return to activation functions when the context arises!

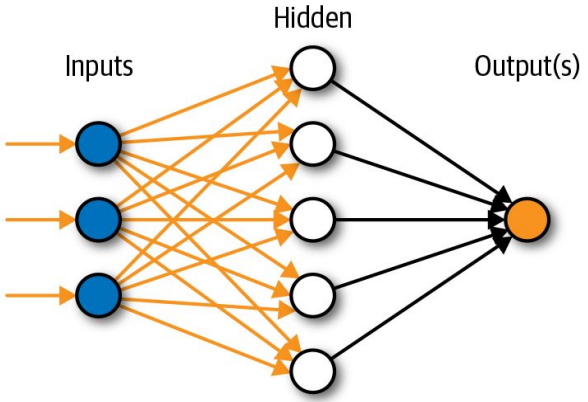
Neural Network: Layer context

- We will cover the neural network concepts:
  - layers, feedforward and ‘fully connected networks’
- Types of layers: input, hidden and output layer
- Feed-Forward NN (FFNN): composed of two or more layers (mostly 3), with activation functions that are usually step or logistic function
- Multi-Layer-Perceptron (MLP): has three or more layers
- Perceptron Learning is a NN without a hidden later, (i.e., two layers that are input and output)
  - Good for emulating the functionality of logical AND and OR, but not good for XOR problem (Boolean functions)



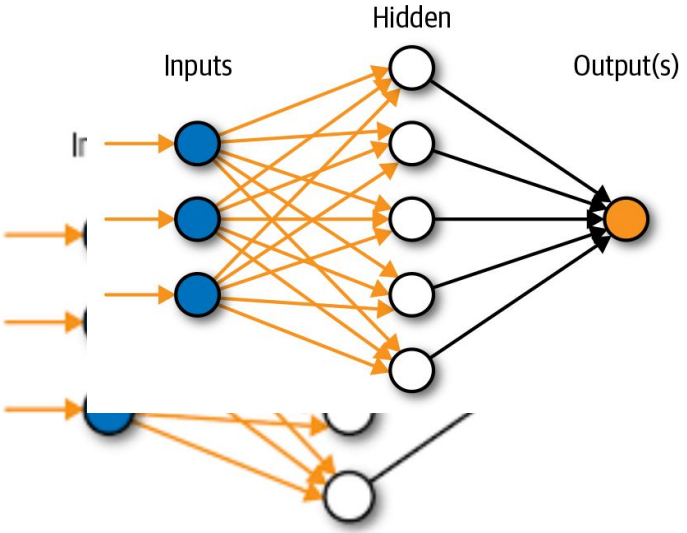
Neural Network: Layer context

- For most tasks 1-2 hidden layers are sufficient
- Too many hidden layers increases the complexity of the model and likely the training time, especially when the “error” is propagated backwards (backpropagation)
- With increasing numbers of hidden layers we have a “Deep” Neural Network (DNN)



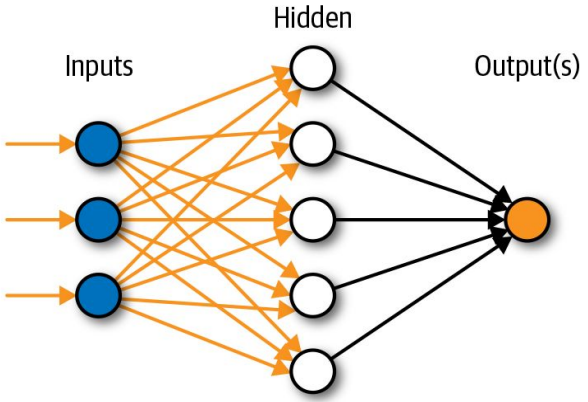
Feed-forward Neural Network (FFNN) : ‘Neurons’ or ‘Nodes’

- Number of nodes in the input layer depends on the number of dataset attributes
- The number of nodes in the output layer may be more than 1 depending on the classification task. How many nodes needed if the class variable has three labels?
- The number of nodes in the hidden layer depends on the complexity of the pattern. An overly large number of nodes can cause overfitting.
  - In case of overfitting, reduce the number of nodes in the hidden layer.
  - In case of low accuracy, increase the number of nodes in the hidden layer.
  - Determined by Trial-and-error

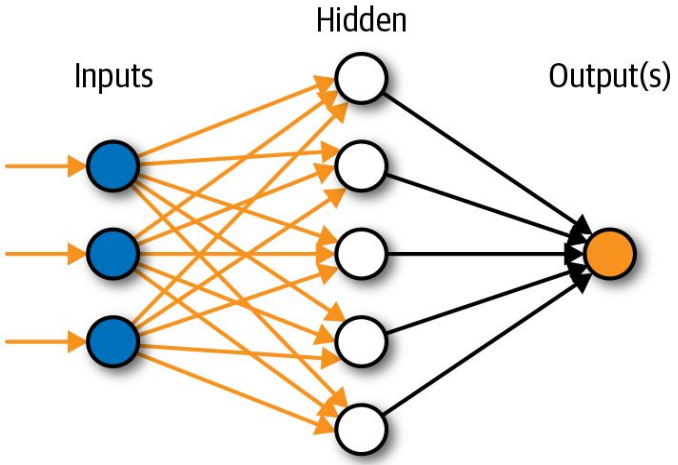


Classification Feed-forward Neural Network (FFNN) : ‘Neurons’ or ‘Nodes’

- Number of nodes in the input layer is given by the number of dataset attributes
- The number of nodes in the output layer may be more than 1 depending on the classification task
- The number of nodes in the hidden layer depends on the classification task (read: the data)
  - An overly large number of nodes can cause overfitting
  - If overfitting: reduce the number of nodes in the hidden layer
  - If low accuracy: increase the number of nodes in
  - the hidden layer
  - Determined by Trial-and-error



```python
import keras
from keras.models import Sequential
from keras.layers import Dense

# define the number of input nodes, hidden nodes, and output
nodes
input_dim = 5
hidden_dim = 10
output_dim = 1

# create a sequential model
model = Sequential()

# add the input layer with ReLU activation function
model.add(Dense(hidden_dim, input_dim=input_dim,
activation='relu'))

# add a hidden layer with ReLU activation function
model.add(Dense(hidden_dim, activation='relu'))

# add the output layer with sigmoid activation function
model.add(Dense(output_dim, activation='sigmoid'))

# compile the model with binary cross-entropy loss function
and Adam optimizer
model.compile(loss='binary_crossentropy', optimizer='adam')

# train the model on the training data
model.fit(X_train, y_train, epochs=100, batch_size=32,
validation_data=(X_test, y_test))
```



- A ‘fully connected’ Feedforward Neural Network (FFNN):
  - Every node is connected to every node in the next layer
  - Weights are valued between 0 and 1

Neural Networks (NN): Pros and Cons

Pros:

- Robust and resilient to noise
- Can model non-linearity

Cons:

- Black-box nature (mostly)
- Prone to overfitting
- Computationally intensive
- Data Requirement
- Hyperparameter tuning
- Standardizing all input attributes

```
>>> from sklearn.preprocessing import MinMaxScaler
>>> data = [[-1, 2], [-0.5, 6], [0, 10], [1, 18]]
>>> scaler = MinMaxScaler()
>>> print(scaler.fit(data))
MinMaxScaler()
>>> print(scaler.data_max_)
[ 1. 18.]
>>> print(scaler.transform(data))
[[0.   0.  ]
 [0.25 0.25]
 [0.5  0.5 ]
 [1.   1.  ]]
>>> print(scaler.transform([[2, 2]]))
[[1.5 0. ]]
```

Feature engineering for NNs: Normalising

- **Feature engineering is another name for pre-processing**
- Recall:
  - **Standardizing** with **z-score** (standard normal distribution with a mean (μ) of 0 and a standard deviation (σ) of 1), and
  - **Normalising aka min-max scaling** (X_{norm} rescales the feature to a fixed range, usually 0 to 1)
- NNs usually perform better with min-max scaling
  - Normalization is useful if know the distribution is not Gaussian or when you need to bound values
  - Note: normalisation is **sensitive to outliers** since the minimum and maximum values are used for scaling

$$z = \frac{(x - \mu)}{\sigma}$$

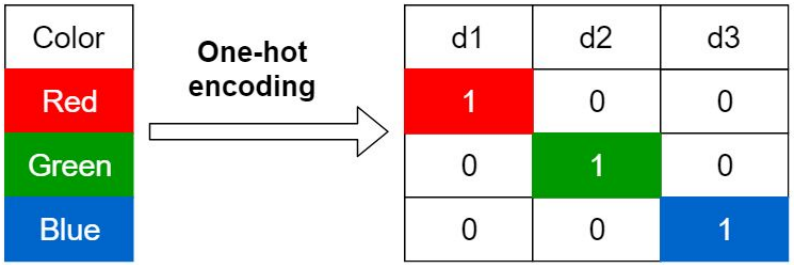
$$X_{\text{norm}} = \frac{(X - X_{\text{min}})}{(X_{\text{max}} - X_{\text{min}})}$$

Feature engineering for NNs: Categorical data

- Categorical variables allow a neural network to understand and represent discrete features of the data:
  - E.g.: Day of the week, color, country
- Learning the representations of categories is largely more generalisable with NNs
- Label Encoding assigns a unique numerical value to categorical data.
  - It assumes an ordered relationship between the categories, so it is not good for encoding nominal attributes
  - Example: Feedback variable taking labels of ‘bad’, ‘good’, and ‘excellent’ taking values of ‘0’,’1’, ‘2’
  - Not recommended for NN due to introducing bias; one-hot encoding is more preferred.
  - Drawback: Arbitrary Ordering can lead to incorrect interpretations

Feature engineering for NNs: Categorical data

- One-hot Encoding is a technique used to represent categorical data as numerical input (i.e., binary vectors) in machine learning.
  - Each label/category in a categorical variable (such as ‘color’) is represented as a binary vector with one element set to 1 and all other elements set to 0
- Example one-hot encoding:
  - Attribute ‘color’ can be ‘red’,’green’, and ‘blue’. The number of labels is 3. You need (k-1 = 2) ‘flag variables’
  - Using one-hot encoding, you can remove ‘color’ column from the dataset and add three columns of ‘red’ , ‘green’ and ‘blue’
- Drawback: it can lead to high-dimensional sparse data, where most of the elements in the encoded vector are zero



One-Hot vs. label encoder

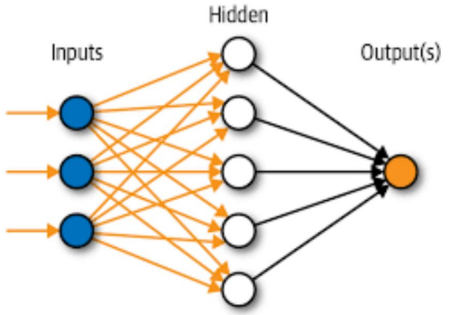
```
from sklearn.preprocessing import
OneHotEncoder
import numpy as np

# create a sample categorical data
data = np.array(['red', 'green', 'blue',
'blue', 'red']).reshape(-1, 1)

# create an instance of the encoder
encoder = OneHotEncoder()

# fit the encoder to the data and transform
the data
data_encoded = encoder.fit_transform(data)

# convert the encoded data to a numpy array
data_encoded = data_encoded.toarray()

# print the encoded data
print(data_encoded)
```

```
[[0. 0. 1.]
[0. 1. 0.]
[1. 0. 0.]
[1. 0. 0.]
[0. 0. 1.]]
```

```
from sklearn.preprocessing import
LabelEncoder
import numpy as np

# create a sample categorical data
data = np.array(['red', 'green', 'blue',
'blue', 'red']).reshape(-1, 1)

# create an instance of the encoder
encoder = LabelEncoder()

# fit the encoder to the data and transform
the data
data_encoded =
encoder.fit_transform(data.ravel())

# print the encoded data
print(data_encoded)
```

```
[2 1 0 0 2]
```

One-output Node Application

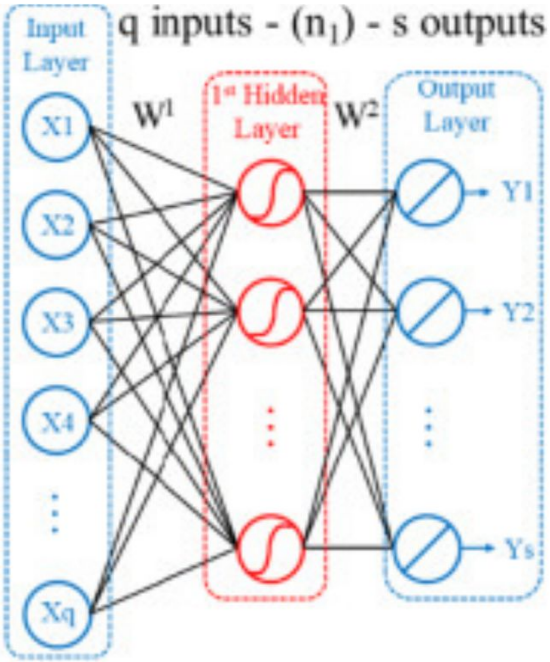


- Good for binary classification (0/1 or win/lose)
- One output node is also good when the output classes are ordered
  - In this case, works for multi-class classification
  - Example: 1, 2, or 3rd place
  - The ‘order’ is used as a continuous or discrete numeric value

$$
\begin{aligned}
&if\ 0 \le output < threshold1 && : classify\ 1st\ place \\
&if\ threshold1 \le output < threshold2 && : classify\ 2nd\ place \\
&if\ threshold2 \le output && : classify\ 3rd\ place
\end{aligned}
$$

1-of-n output Encoding

- There is more than one output node in the output layer (Y)
- Output classes are not ordered (i.e., nominal) e.g., gender: {male, female, unknown}
- Each output node corresponds to one class label, quantified with a probability
- Benefit: it provides probabilities used as a measure of **confidence** in the classification



Feed-Forward Neural Network: Combination Function

- The ‘feedforward’ combination function in a neural network is the operation used to compute the input to a neuron in a given layer from the outputs of the neurons in the previous layer
- The equation describes how data flows through the network during the forward pass
- First: Linear Combination of Inputs: $z = w_1x_1 + w_2x_2 + \ldots + w_nx_n + b$
  - Here, z values are also known as the neuron's ‘pre-activation’ values, and referred to as a ‘net’, sum, or Σ
- Second: Introduce the activation function to z
  - Pass the z, or ‘pre-activation’ values to the activation function (e.g. sigmoid, unit step)
- Third: Propagate to the next layer

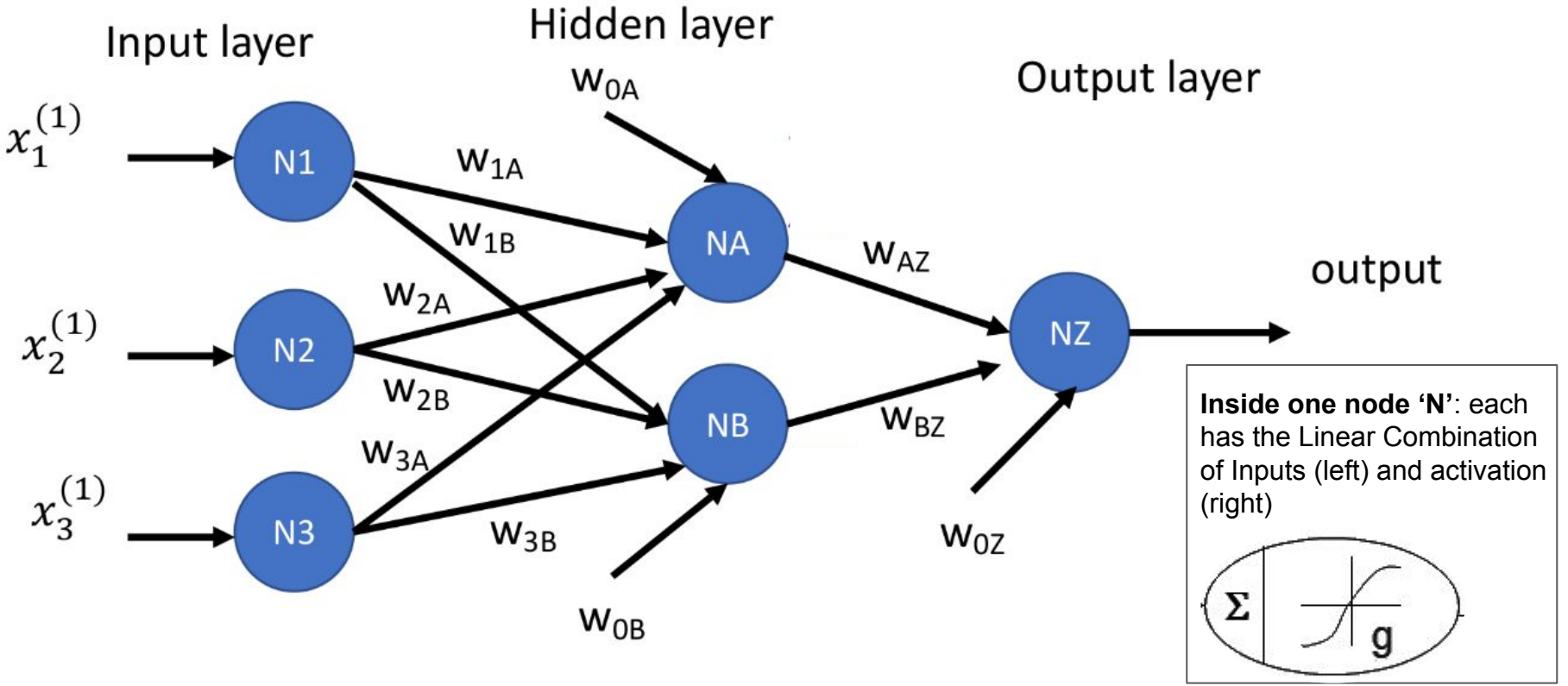
Feed-Forward Neural Network: Combination Function

Nx4 input matrix
N: number of samples
4: number of attributes including x_0
$x_0 \; is \; 1 \; by \; convention$

$$\begin{bmatrix} x_0^{(1)} & x_1^{(1)} & x_2^{(1)} & x_3^{(1)} \\ \vdots & & \ddots & \vdots \\ x_0^{(n)} & & \cdots & x_3^{(n)} \end{bmatrix} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & x_3^{(1)} \\ \vdots & & \ddots & \vdots \\ 1 & & \cdots & x_3^{(n)} \end{bmatrix}$$

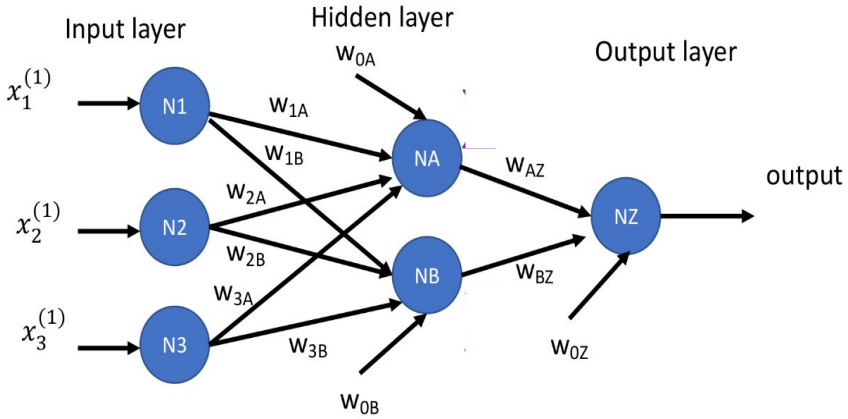
$$y^{(i)} = \sum_{j=0}^{M} w_j x_j^{(i)} = \omega_0 + \omega_1 x_1^{(i)} + \omega_2 x_2^{(i)} + \omega_3 x_3^{(i)}$$

Feed-Forward Neural Network: Combination Function



Feed-Forward Neural Network: Combination Function

$$\begin{bmatrix} x_0^{(1)} & x_1^{(1)} & x_2^{(1)} & x_3^{(1)} \\ \vdots & & \ddots & \vdots \\ x_0^{(n)} & & \cdots & x_3^{(n)} \end{bmatrix} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & x_3^{(1)} \\ \vdots & & \ddots & \vdots \\ 1 & & \cdots & x_3^{(n)} \end{bmatrix}$$



Initiate weights randomly

w_{0A} =0.5	w_{0B} =0.7	w_{0Z}=0.5
w_{1A} = 0.6	w_{1B} = 0.9	w_{AZ}=0.9
w_{2A}=0.8	w_{2B}=0.8	w_{BZ}=0.9
w_{3A}=0.6	w_{3B}=0.4	

w_{ij}: the weight associated with the i^{th} input to node j

x_{ij}: i^{th} input to node j

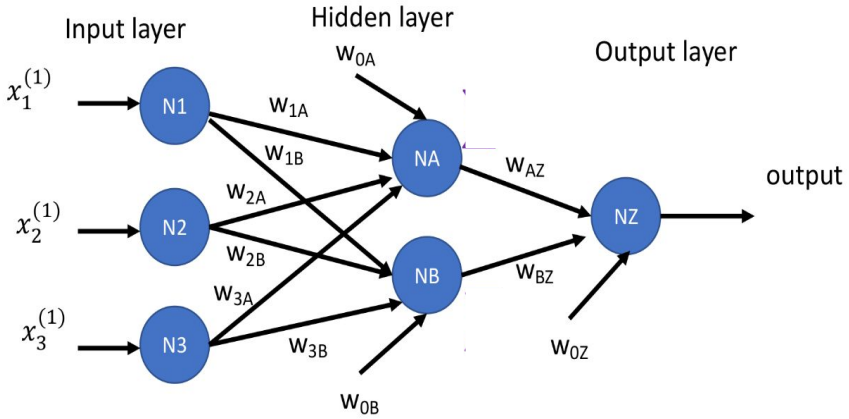
Feed-Forward Neural Network: Combination Function

$$\begin{bmatrix} x_0^{(1)} & x_1^{(1)} & x_2^{(1)} & x_3^{(1)} \\ \vdots & & \ddots & \vdots \\ x_0^{(n)} & & \cdots & x_3^{(n)} \end{bmatrix} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & x_3^{(1)} \\ \vdots & & \ddots & \vdots \\ 1 & & \cdots & x_3^{(n)} \end{bmatrix}$$

Initiate weights randomly

w_{0A} =0.5	w_{0B} =0.7	w_{0Z}=0.5
w_{1A} = 0.6	w_{1B} = 0.9	w_{AZ}=0.9
w_{2A}=0.8	w_{2B}=0.8	w_{BZ}=0.9
w_{3A}=0.6	w_{3B}=0.4	

w_{ij}: the weight associated with the i^{th} input to node j
x_{ij}: i^{th} input to node j



Calculate the scalar value passed to a node j through Linear Combination of Inputs:

$$net^{(i)}{}_j = \sum_k w_{kj} x_{kj}^{(i)}$$

For node NA:

$$net^{(1)}{}_A = \omega_{0j} + \omega_{1j} x_{1j}^{(1)} + \omega_{2j} x_{2j}^{(1)} + \omega_{3j} x_{3j}^{(1)}$$

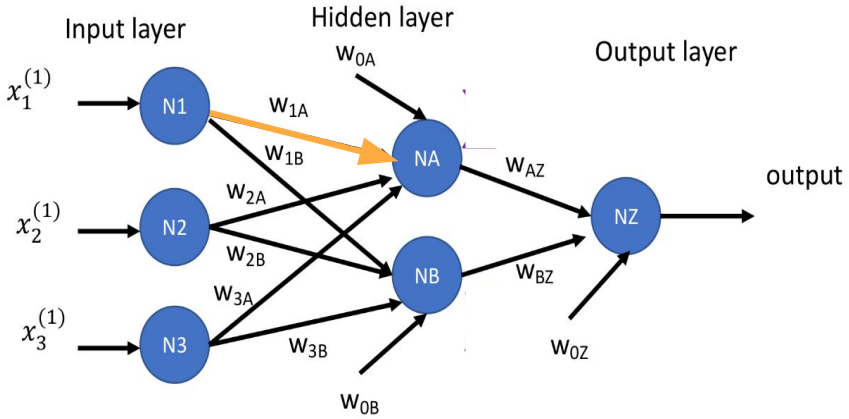
Feed-Forward Neural Network: Combination Function

$$\begin{bmatrix} x_0^{(1)} & x_1^{(1)} & x_2^{(1)} & x_3^{(1)} \\ \vdots & & \ddots & \vdots \\ x_0^{(n)} & & \cdots & x_3^{(n)} \end{bmatrix} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & x_3^{(1)} \\ \vdots & & \ddots & \vdots \\ 1 & & \cdots & x_3^{(n)} \end{bmatrix}$$

Initiate weights randomly

w_{0A} =0.5	w_{0B} =0.7	w_{0Z}=0.5
w_{1A} = 0.6	w_{1B} = 0.9	w_{AZ}=0.9
w_{2A}=0.8	w_{2B}=0.8	w_{BZ}=0.9
w_{3A}=0.6	w_{3B}=0.4	

w_{ij}: the weight associated with the i^{th} input to node j
x_{ij}: i^{th} input to node j



Calculate the scalar value passed to a node j through Linear Combination of Inputs:

$$net^{(i)}{}_j = \sum_k w_{kj} x_{kj}^{(i)}$$

For node NA:

$$net^{(1)}{}_A = \omega_{0j} + \omega_{1j} x_{1j}^{(1)} + \omega_{2j} x_{2j}^{(1)} + \omega_{3j} x_{3j}^{(1)}$$

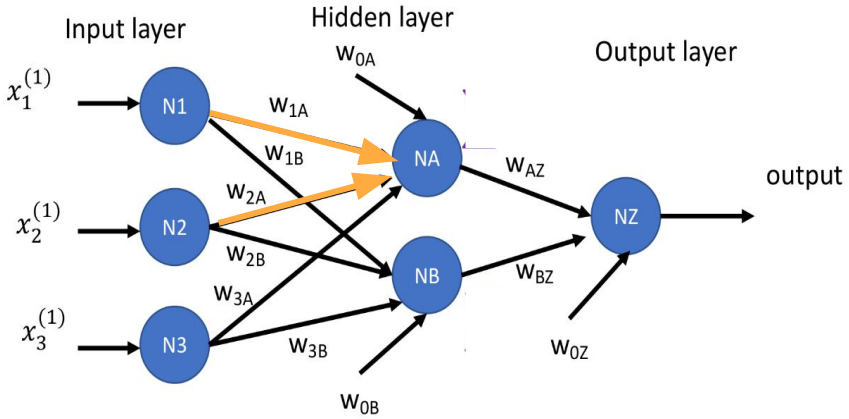
Feed-Forward Neural Network: Combination Function

$$\begin{bmatrix} x_0^{(1)} & x_1^{(1)} & x_2^{(1)} & x_3^{(1)} \\ \vdots & & \ddots & \vdots \\ x_0^{(n)} & & \cdots & x_3^{(n)} \end{bmatrix} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & x_3^{(1)} \\ \vdots & & \ddots & \vdots \\ 1 & & \cdots & x_3^{(n)} \end{bmatrix}$$

Initiate weights randomly

w_{0A} =0.5	w_{0B} =0.7	w_{0Z}=0.5
w_{1A} = 0.6	w_{1B} = 0.9	w_{AZ}=0.9
w_{2A}=0.8	w_{2B}=0.8	w_{BZ}=0.9
w_{3A}=0.6	w_{3B}=0.4	

w_{ij}: the weight associated with the i^{th} input to node j
x_{ij}: i^{th} input to node j



Calculate the scalar value passed to a node j through Linear Combination of Inputs:

$$net^{(i)}{}_j = \sum_k w_{kj} x_{kj}^{(i)}$$

For node NA:

$$net^{(1)}{}_A = \omega_{0j} + \omega_{1j} x_{1j}^{(1)} + \omega_{2j} x_{2j}^{(1)} + \omega_{3j} x_{3j}^{(1)}$$

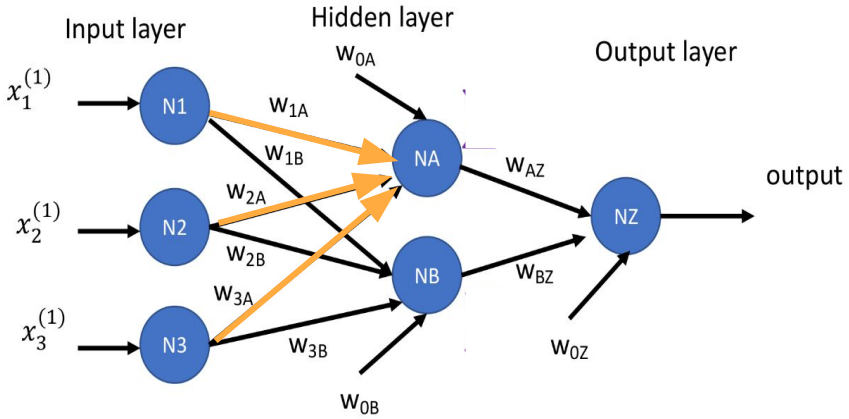
Feed-Forward Neural Network: Combination Function

$$\begin{bmatrix} x_0^{(1)} & x_1^{(1)} & x_2^{(1)} & x_3^{(1)} \\ \vdots & \ddots & & \vdots \\ x_0^{(n)} & \cdots & & x_3^{(n)} \end{bmatrix} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & x_3^{(1)} \\ \vdots & \ddots & & \vdots \\ 1 & \cdots & & x_3^{(n)} \end{bmatrix}$$

Initiate weights randomly

w_{0A} =0.5	w_{0B} =0.7	w_{0Z}=0.5
w_{1A} = 0.6	w_{1B} = 0.9	w_{AZ}=0.9
w_{2A}=0.8	w_{2B}=0.8	w_{BZ}=0.9
w_{3A}=0.6	w_{3B}=0.4	

w_{ij}: the weight associated with the i^{th} input to node j
x_{ij}: i^{th} input to node j



Calculate the scalar value passed to a node j through Linear Combination of Inputs:

$$net^{(i)}{}_j = \sum_k w_{kj} x_{kj}^{(i)}$$

For node NA:

$$net^{(1)}{}_A = \omega_{0j} + \omega_{1j} x_{1j}^{(1)} + \omega_{2j} x_{2j}^{(1)} + \omega_{3j} x_{3j}^{(1)}$$

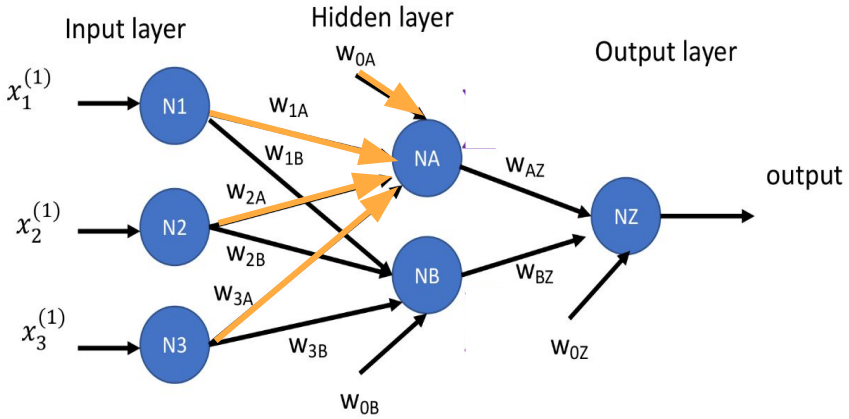
Feed-Forward Neural Network: Combination Function

$$\begin{bmatrix} x_0^{(1)} & x_1^{(1)} & x_2^{(1)} & x_3^{(1)} \\ \vdots & & \ddots & \vdots \\ x_0^{(n)} & & \cdots & x_3^{(n)} \end{bmatrix} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & x_3^{(1)} \\ \vdots & & \ddots & \vdots \\ 1 & & \cdots & x_3^{(n)} \end{bmatrix}$$

Initiate weights randomly

w_{0A} =0.5	w_{0B} =0.7	w_{0Z}=0.5
w_{1A} = 0.6	w_{1B} = 0.9	w_{AZ}=0.9
w_{2A}=0.8	w_{2B}=0.8	w_{BZ}=0.9
w_{3A}=0.6	w_{3B}=0.4	

w_{ij}: the weight associated with the i^{th} input to node j
x_{ij}: i^{th} input to node j



Calculate the scalar value passed to a node j through Linear Combination of Inputs:

$$net^{(i)}{}_j = \sum_k w_{kj} x_{kj}^{(i)}$$

For node NA:

$$net^{(1)}{}_A = \omega_{0j} + \omega_{1j} x_{1j}^{(1)} + \omega_{2j} x_{2j}^{(1)} + \omega_{3j} x_{3j}^{(1)}$$

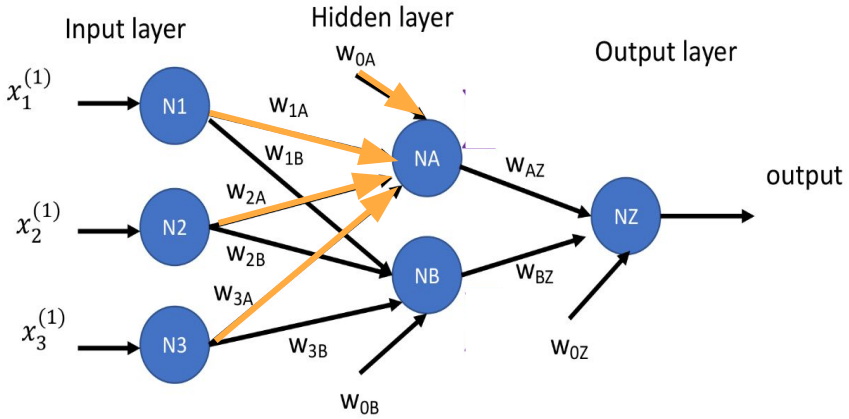
Feed-Forward Neural Network: Combination Function

$$\begin{bmatrix} x_0^{(1)} & x_1^{(1)} & x_2^{(1)} & x_3^{(1)} \\ \vdots & \ddots & & \vdots \\ x_0^{(n)} & \cdots & & x_3^{(n)} \end{bmatrix} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & x_3^{(1)} \\ \vdots & \ddots & & \vdots \\ 1 & \cdots & & x_3^{(n)} \end{bmatrix}$$

Initiate weights randomly

w_{0A} =0.5	w_{0B} =0.7	w_{0Z}=0.5
w_{1A} = 0.6	w_{1B} = 0.9	w_{AZ}=0.9
w_{2A}=0.8	w_{2B}=0.8	w_{BZ}=0.9
w_{3A}=0.6	w_{3B}=0.4	

w_{ij}: the weight associated with the i^{th} input to node j
x_{ij}: i^{th} input to node j



Calculate the scalar value passed to a node j through Linear Combination of Inputs:

$$net^{(i)}{}_j = \sum_k w_{kj} x_{kj}^{(i)}$$

For node NA:

$$net^{(1)}{}_A = \omega_{0j} + \omega_{1j} x_{1j}^{(1)} + \omega_{2j} x_{2j}^{(1)} + \omega_{3j} x_{3j}^{(1)} = 1.32$$

Feed-Forward Neural Network: Combination Function

Our scalar value passed to a node NA through Linear Combination of Inputs:

$$net^{(1)}{}_A = \omega_{0j} + \omega_{1j}x_{1j}^{(1)} + \omega_{2j}x_{2j}^{(1)} + \omega_{3j}x_{3j}^{(1)} = 1.32$$

The input (1.32) is now given to the activation function g e.g. a sigmoid such that in general:

$$y = \frac{1}{1+e^{-z}} = g(net_j)$$

For NA, the result now becomes:

$$g(net_A) = \frac{1}{1+e^{-1.38}} = 0.7892$$

Similarly for NB, going through the Linear Combination of Inputs for node NB:

$$g(net_B) = \frac{1}{1+e^{-net_B}} = 0.8176$$

We then for the *next* layer would repeat the process until we reach the output layer…

Feed-Forward Neural Network: Combination Function

w_{0A} =0.5	w_{0B} =0.7	w_{0Z}=0.5
w_{1A} = 0.6	w_{1B} = 0.9	w_{AZ}=0.9
w_{2A}=0.8	w_{2B}=0.8	w_{BZ}=0.9
w_{3A}=0.6	w_{3B}=0.4	

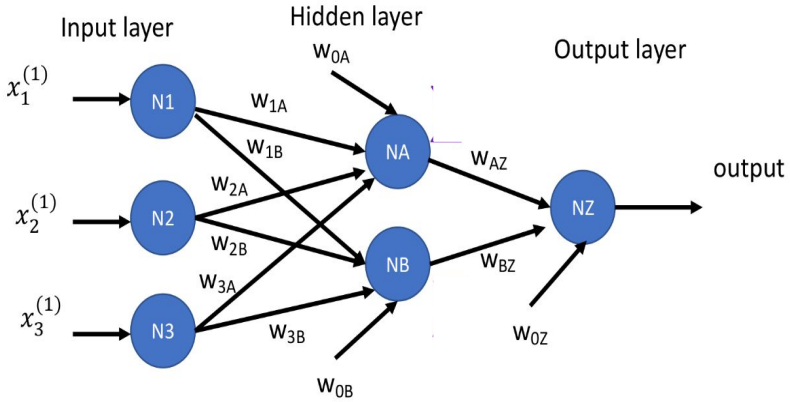
From NA and NB we have 0.7892 and 0.8176 (effective our x_{AZ} and x_{BZ}) and weights w_{0Z} , w_{AZ} and w_{BZ} are in the table

We calculate the Linear Combination of Inputs in NZ as:

$$net^{(1)}{}_Z = \sum_k w_{kZ} x_{kZ}^{(1)} = \omega_{0Z} + \omega_{AZ} x_{AZ}^{(1)} + \omega_{BZ} x_{BZ}^{(1)} = 0.5 + 0.9(0.7892) + 0.9(0.8176) = 1.9461$$

This again is passed to the activation function g:

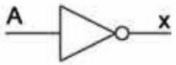
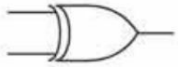
$$g(net_Z) = \frac{1}{1 + e^{-1.9461}} = 0.8750$$



Output from the NN for pass 1 through the network, and it is the predicted value for the first observation in the dataset D.

Emulating Boolean Functions with a NN

Logical Gates

Name	NOT	AND	NAND	OR	NOR	XOR	XNOR
Alg. Expr.	$\overline{A}$	AB	$\overline{AB}$	$A+B$	$\overline{A+B}$	$A \oplus B$	$\overline{A \oplus B}$
Symbol							
Truth Table	A X 0 1 1 0	B A X 0 0 0 0 1 0 1 0 0 1 1 1	B A X 0 0 1 0 1 1 1 0 1 1 1 0	B A X 0 0 0 0 1 1 1 0 1 1 1 1	B A X 0 0 1 0 1 0 1 0 0 1 1 0	B A X 0 0 0 0 1 1 1 0 1 1 1 0	B A X 0 0 1 0 1 0 1 0 0 1 1 1

Source: https://medium.com/autonomous-agents/how-to-teach-logic-to-your-neuralnetworks-116215c71a49