Chemical Engineering 4H03

Artificial Neural Networks (ANNs): Intro and Perceptron Networks

Jake Nease

McMaster University

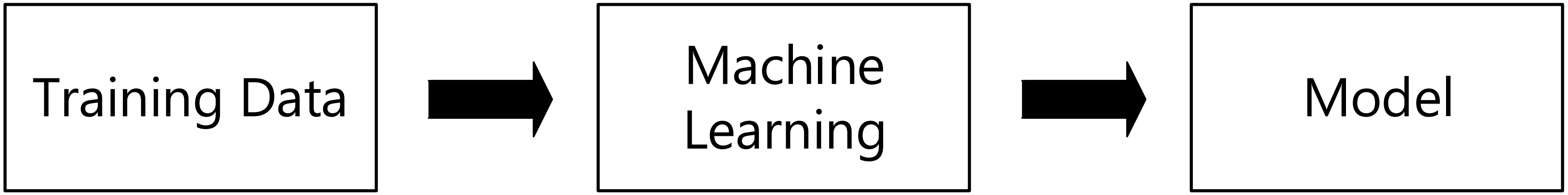
Context: Machine Learning

- Many forms of modeling involve the use of known data (inputs and outputs) with the intent of predicting future outputs
  - For PCA/PLS/PCR/MLR, the known data is in **X** and **Y**
  - We use known outcomes relating **X** and **Y** during model training
  - We then use our model to predict **Y** in the future

- Artificial Neural Networks are a sub-classification of machine learning, which is a mechanism by which a model is trained to reproduce a known result
  - "Mistakes" or "errors" of the model are "corrected" by some sort of optimization or objective, hence the loose use of the term "learning*"

**Reminds me of the purpose of university…*

Context: Machine Learning

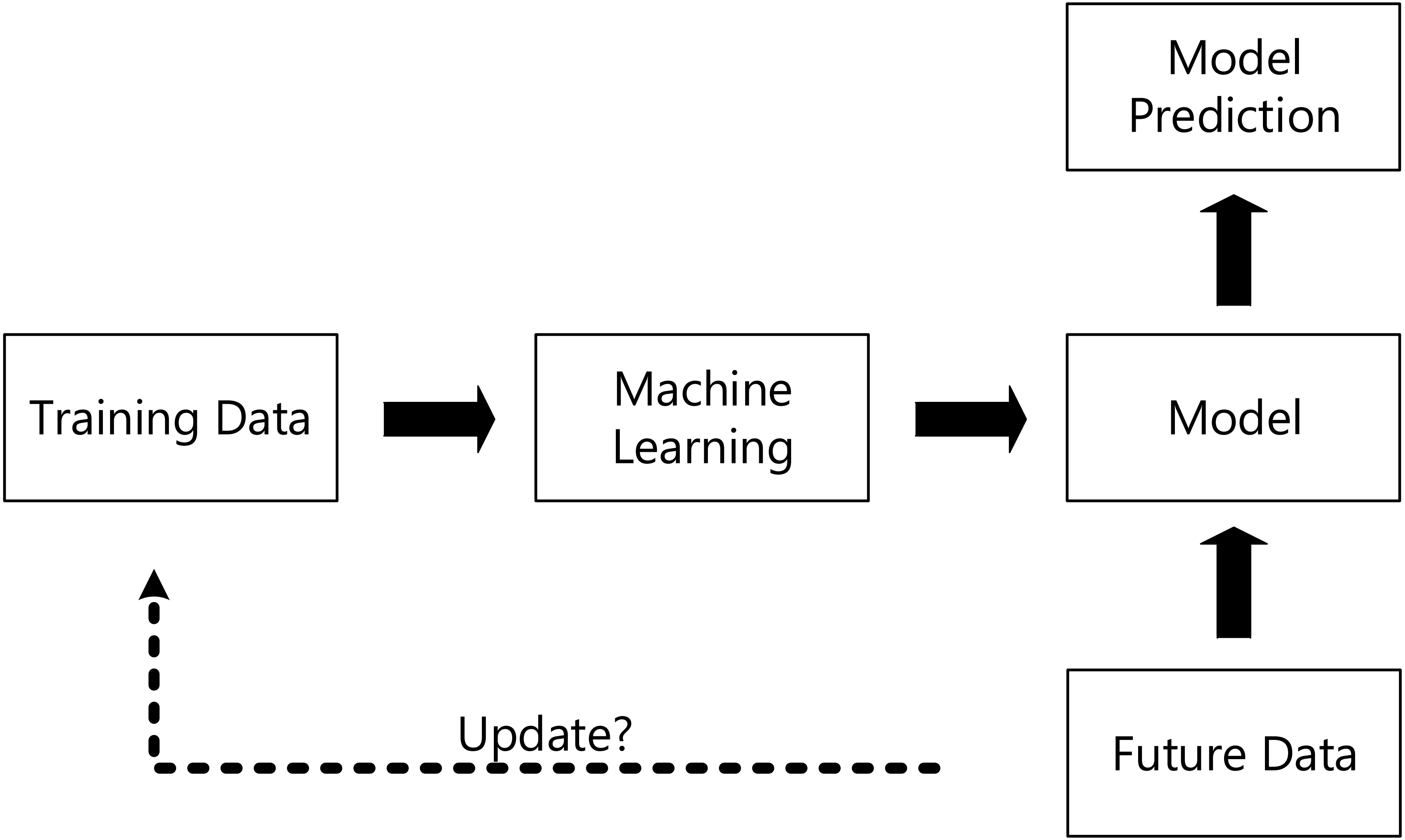


Training Data

Machine Learning

Model

Context: Machine Learning



Model Prediction

Training Data

Machine Learning

Model

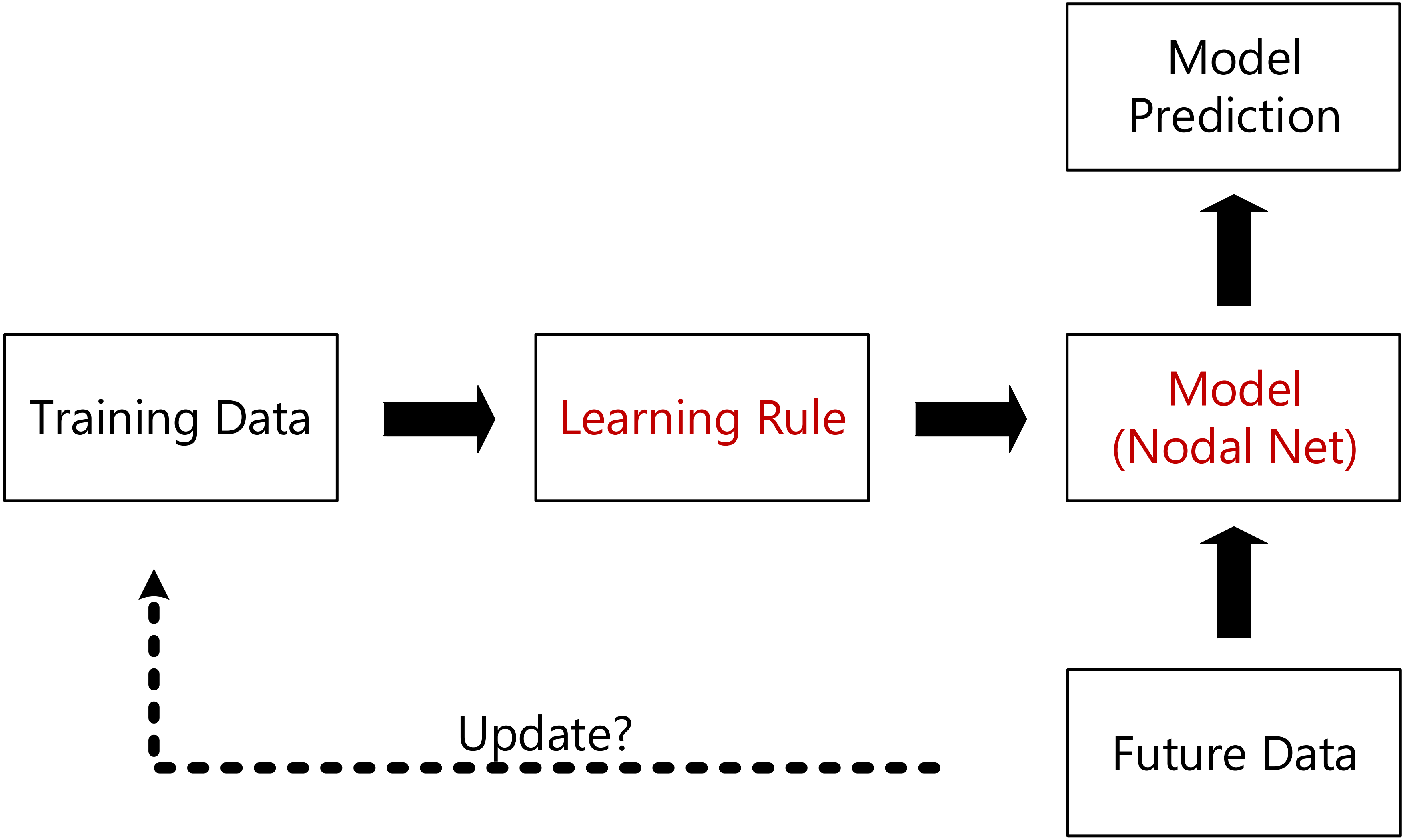
Future Data

Update?

Types of Machine Learning

- Supervised Learning
  - The training data contains inputs and **KNOWN outputs**
  - It is the ability of the model-fitting (machine learning) step to know if it is right or wrong
  - Regression, classification, PLS (combined with dimension reduction), ANNs, Decision trees…
- Unsupervised Learning
  - Does not train to predict an outcome, but rather tries to identify **structure** in the data
  - Clustering, support vector machines (SVMs)…
- There are others, but you can Wikipedia them as easily as I can

Context: Artificial Neural Networks



Model Prediction

Training Data

Learning Rule

Model (Nodal Net)

Future Data

Update?

The ADAMS® Outlook*

Many NN researchers are engineers, physicists, neurophysiologists, psychologists, or computer scientists who know little about statistics or nonlinear optimization. NN researchers routinely reinvent methods that have been known in the statistics literature for decades or centuries, but they often fail to understand how these methods work.

Sarle, 1999

- Much like STOCHASTIC OPTIMIZATION (4G plug), ANNs are grounded in solid mathematical theory, are USEFUL, and are "easy to implement"
  - The "easy to implement" bit is because the math is genuinely simple… But we can only use ANNs due to the incredible computing power available on modern computers
- However, like stochastic optimization, there are a lot of "hand-wavy" arguments that lead to modeling **tweaks** good for some problems, but not others
  - This type of thing betrays the ignorance of the researcher
  - ANNs are thus generally not respected in math communities
- But hey, if it works, it works. Who cares. I certainly don't

**For legal reasons it is important for me to mention that I do not represent Thomas Adams or his affiliates*

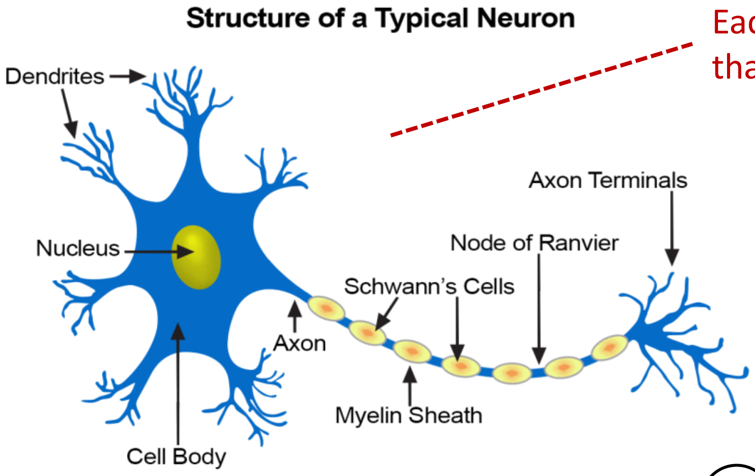
ANN Terminology

Words. Meanings. Lingo.

Organic Information Storage

- Computers and human brains can store information
- Computers store information in dedicated slots, known as memory (there are different kinds)
- Brains store information by altering the association of neurons to store information
  - It is worth noting that a neuron does not actually store information, but it is the *pathway* that helps us remember
- A neural network attempts to mimic the brain's **association of neurons** using a network of **nodes**

Neural Networks as the Brain

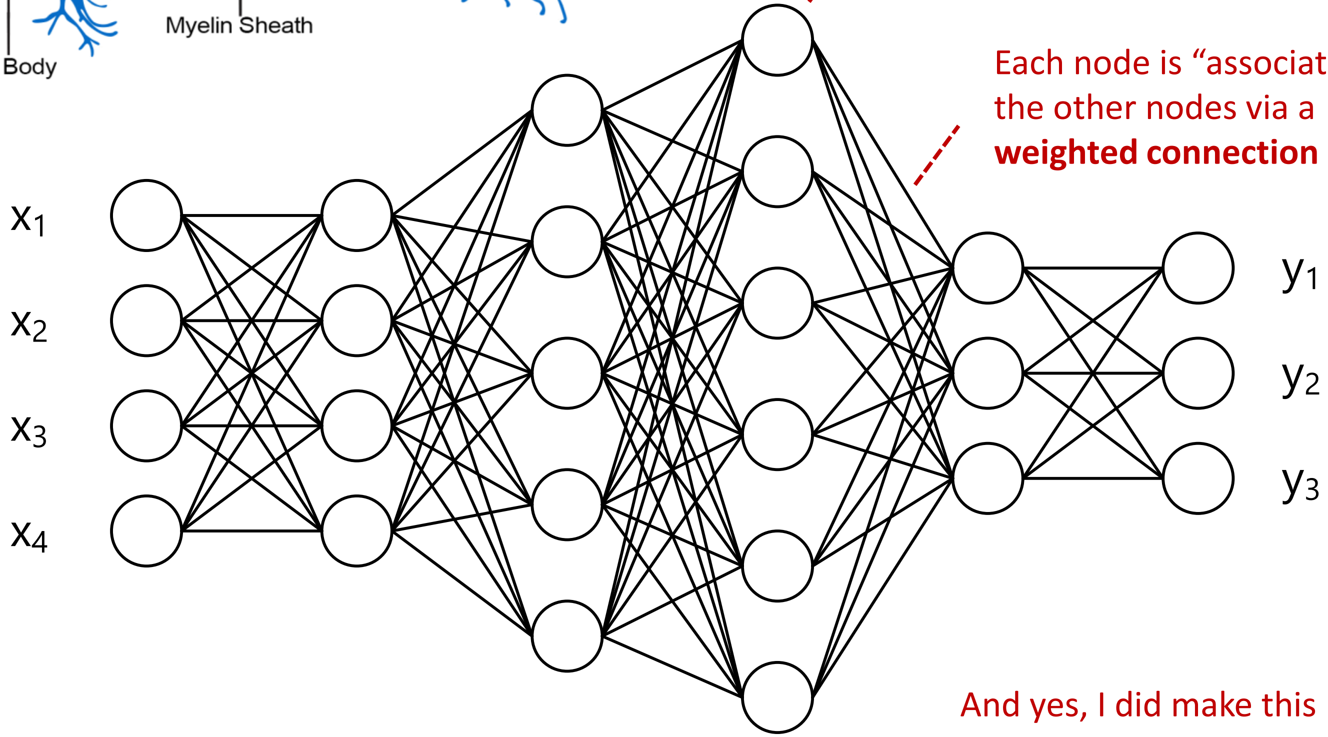


Each neuron can transmit electrical signals to other neurons… And that's about the extent of the biology we'll cover in this course ☺

Each node in a neural network is a replication of a neuron in the brain

Each node is "associated" to the other nodes via a **weighted connection**

These are the **input** "signals" to our electronic brain

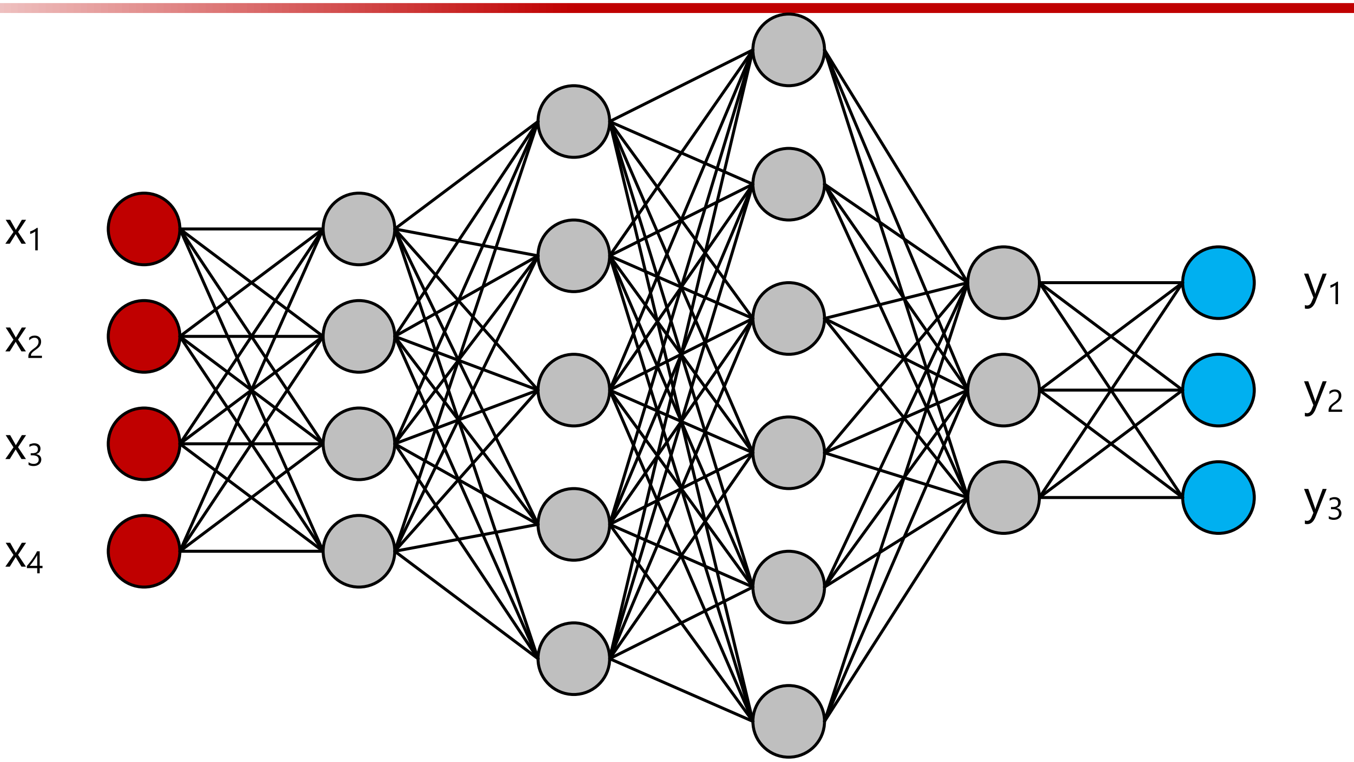


These are the **output** "signals" from our electronic brain

And yes, I did make this picture myself :3

https://training.seer.cancer.gov/images/anatomy/nervous/neuron-structure.jpg

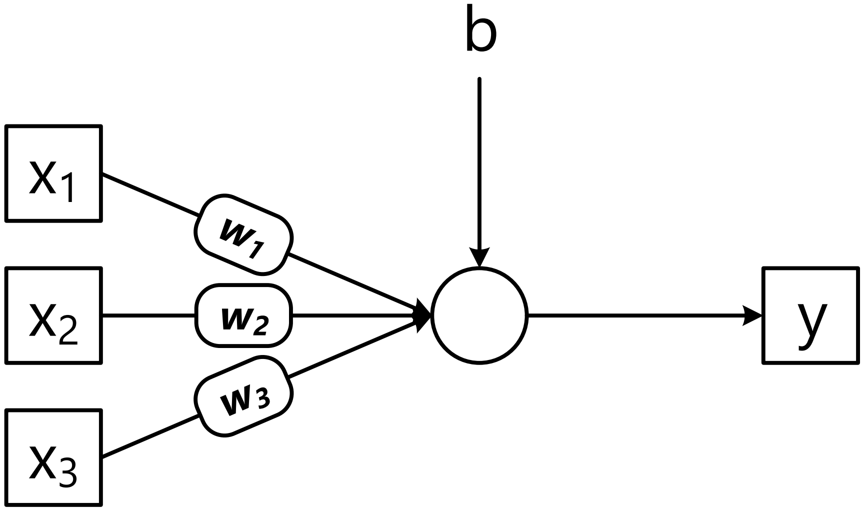
Some Quick Lingo and Terminology



- The input signals are always provided through input nodes
- Inner nodes are called hidden nodes because they are not accessible from outside of the network
- Output nodes are the final “processed” signal from the network
- Each signal between nodes is **weighted**

Node Calculations

- Consider the following node with three inputs $x_1 \cdots x_3$



- Each of the **input signals** x_i are multiplied by **weights** w_i before entering the node
- The node also has a **bias** b added
  - Any guesses as to what b represents in what we have looked at so far?
- The total input to the node is thus:

$$v = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$$

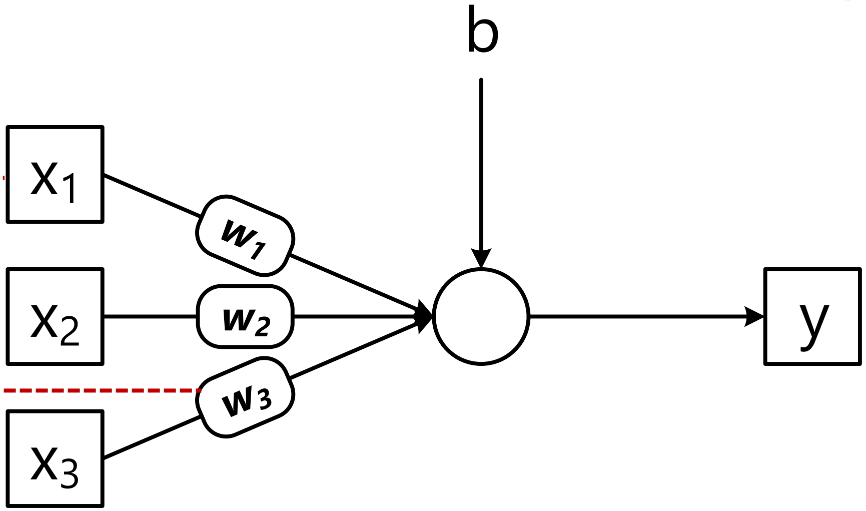
Node Calculations: Input

What does this look like to you?

$v = w_1x_1 + w_2x_2 + w_3x_3 + b$

- Some observations
  - The higher the **weight** w_i, the greater the impact x_i has on v
  - The input to the node is the **weighted sum** of all inputs

- As it turns out, it is the OBJECTIVE of machine learning to select the weights w_i so that y is reproduced from x_i



By the way, input nodes are usually squares in ANN diagrams

If any weight is 0, that input does not affect the network (can be forgotten)

Node Calculations: Output

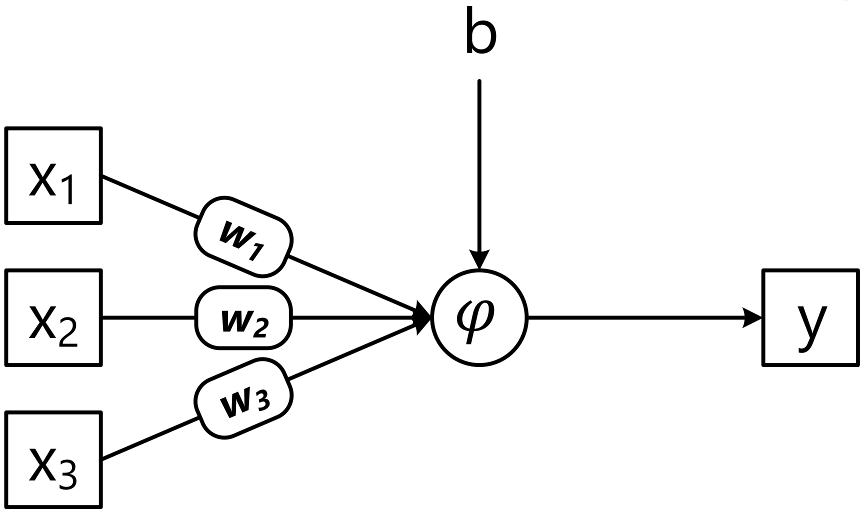
- We can convert the node input to **matrix form** as:
- $v = w_1 x_1 + w_2 x_2 + \cdots + w_K x_K + b$ ------ Note b is scalar (so is v), and there are K instances (columns) of $\boldsymbol{x}$
- $\boldsymbol{w} \triangleq [w_1\ w_2 \cdots w_K]^T$ $\quad\quad$ $\boldsymbol{x} \triangleq [x_1\ x_2 \cdots x_K]^T$

$$v = \boldsymbol{w}^T \boldsymbol{x} + b$$

- The node **output** y is a function of the input v:

$$y = \varphi(v) = \varphi(\boldsymbol{w}^T \boldsymbol{x} + b)$$

- The function $\varphi(\cdot)$ is known as the **node activation function**



Activation Functions

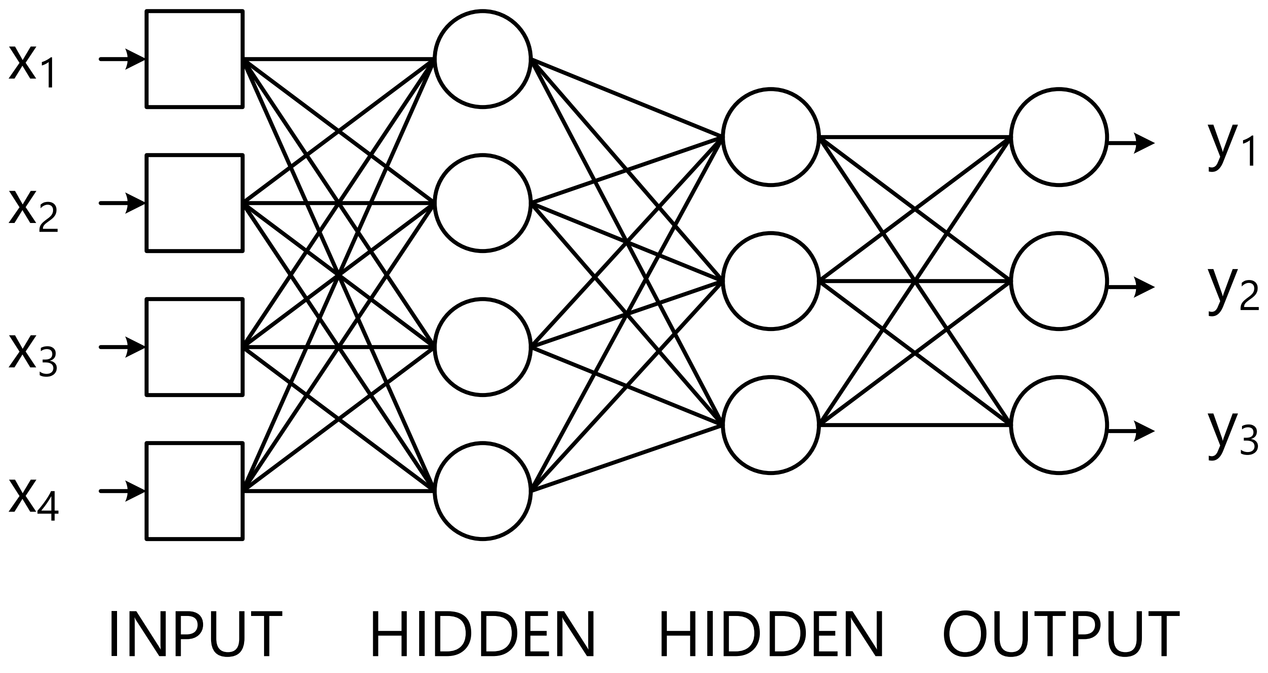
- The activation function $\varphi(v)$ can be selected from almost any of our typical basis functions!
  - Selection depends a little on the application (heuristics!)
  - Most commonly used activation function is the **sigmoid**

- Examples

– Linear combination of inputs	$\varphi(v) = v$
– Power of inputs (power $\alpha > 1$)	$\varphi(v) = v^{\alpha}$
– Logistical function (s known)	$\varphi(v) = 1 + \tanh(sv)$
– Sigmoid function	$\varphi(v) = \frac{1}{1+e^{-v}}$
– Threshold function (step)	$\varphi(v) = \begin{cases} 0; & v \leq t \\ 1; & v > t \end{cases}$

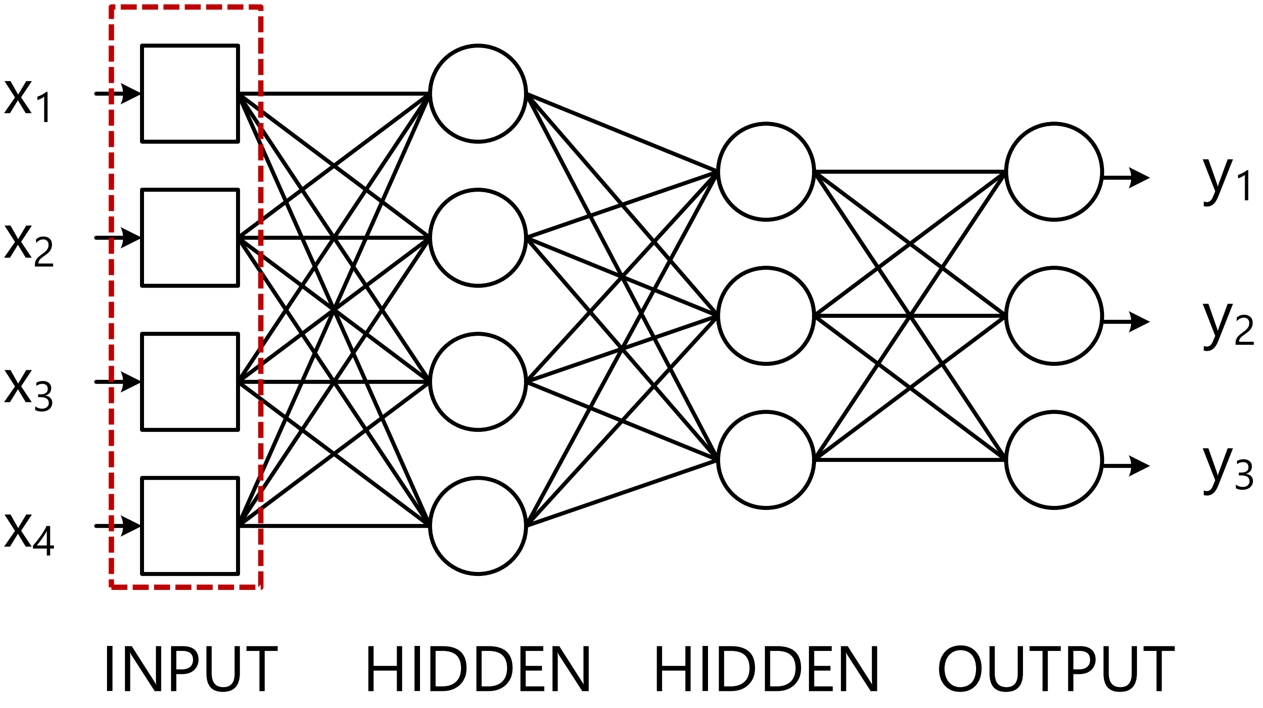
Structure of Neural Networks

- Artificial Neural Networks are formed through the **connections** of nodes with each other
  - Nodes may feed other nodes
  - The most common connection is a **layered connection**



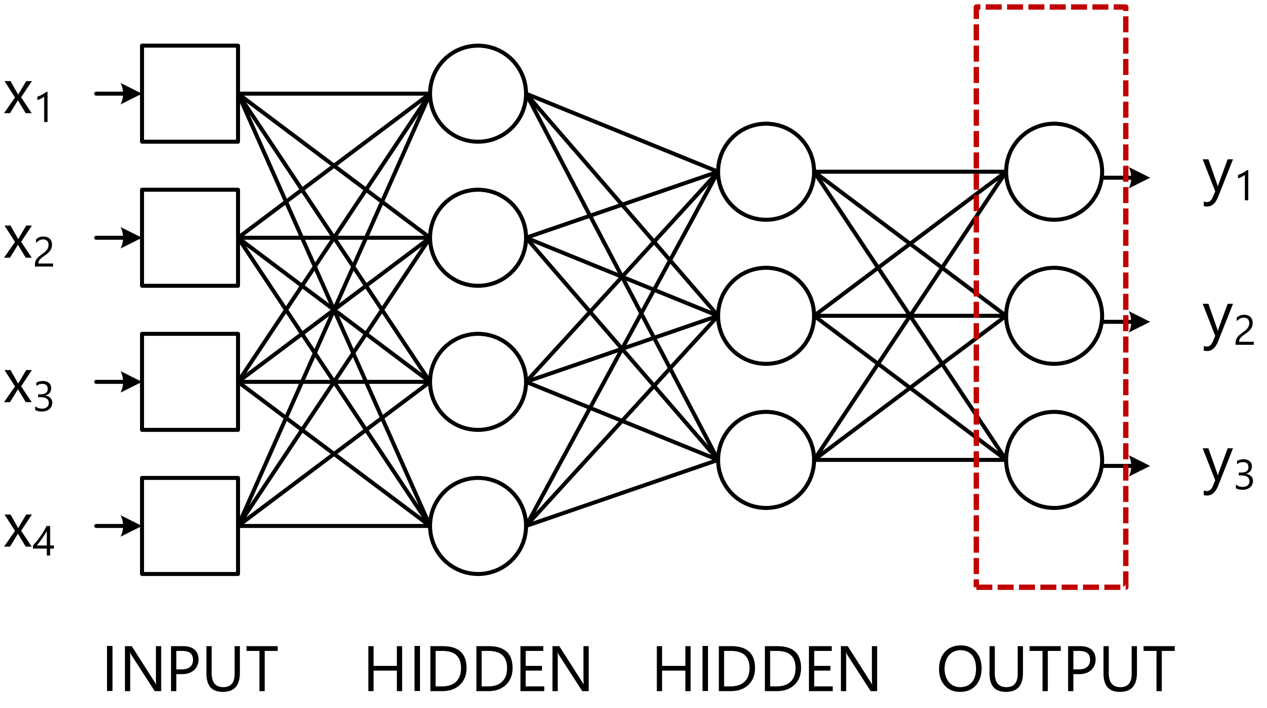
Structure of Neural Networks

- Input Nodes
  - Do not have an activation function or bias ($\varphi(v) = v$)
  - Are denoted as squares
  - Are meant to transmit the data to the ANN



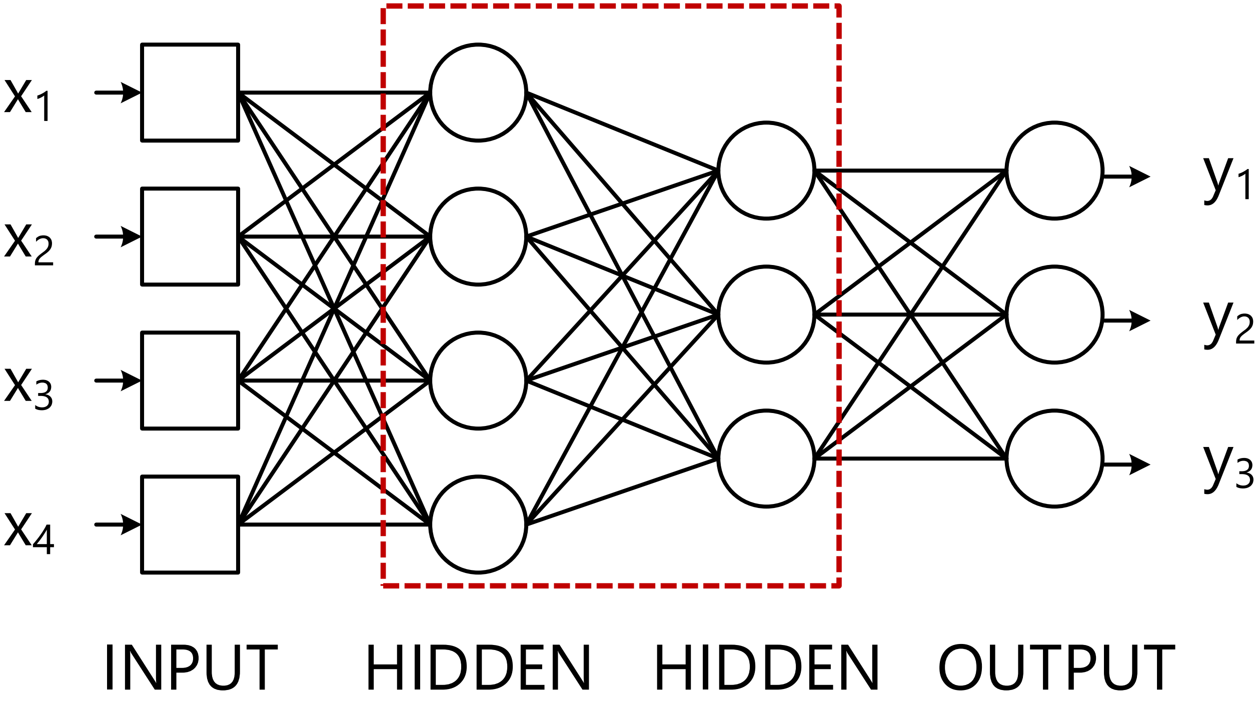
Structure of Neural Networks

- Output Nodes
  - Provide us with the output of the trained ANN model



Structure of Neural Networks

- Hidden Nodes
  - Transform the input signals to the outputs using φ
  - Selection of activation functions can effect model efficacy
  - Activation functions must be known **ahead of time** (BFR?)

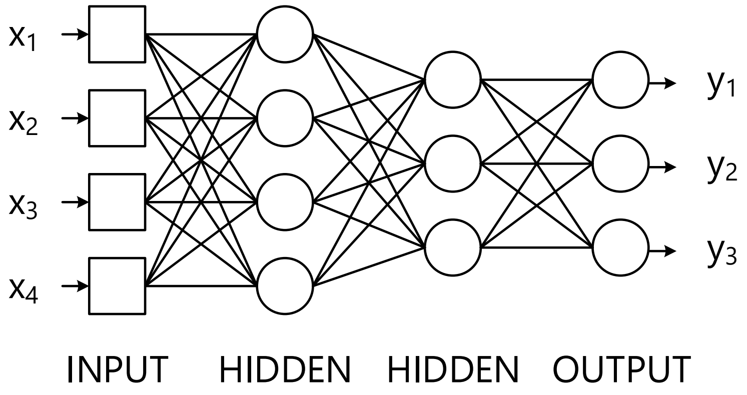


A Little History

- Originally, ANNs were developed as **single-layer networks**
  - Input x_k was transmitted directly to the output weighted by w_k
  - AKA… Linear regression (if >1 input and >1 output, just MLR) if $\varphi(v) = v$ (linear activation function)

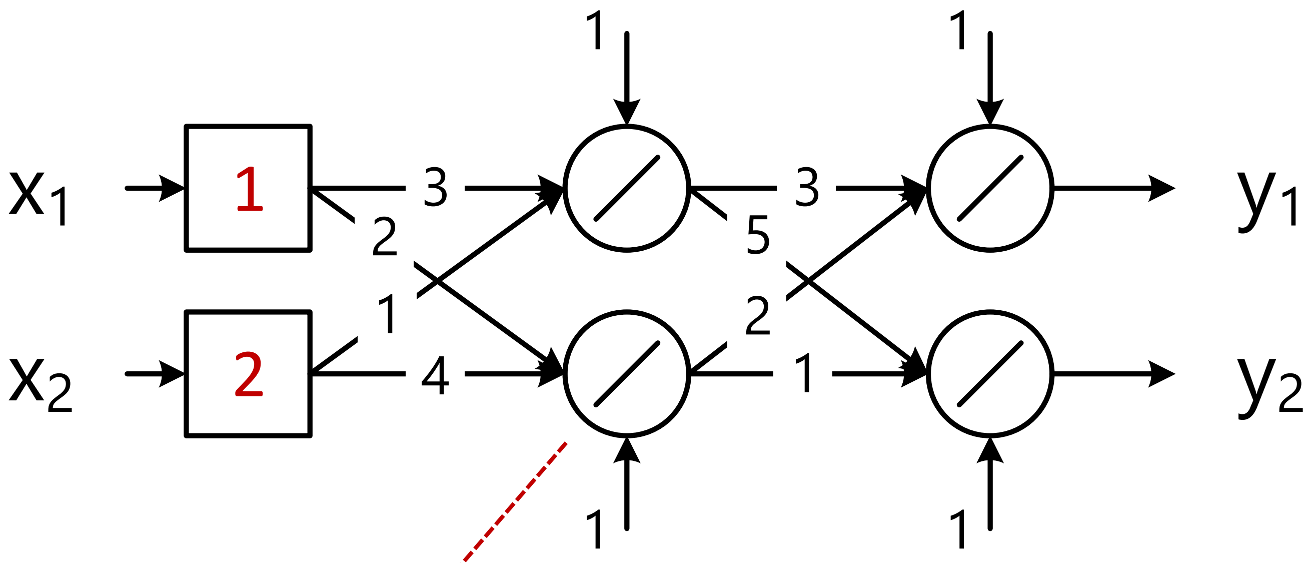
- Later, **hidden layers** were added to allow for each x_k to influence others
  - This is called a **shallow network**
  - Can still be quite big if a lot of inputs/outputs are used

- When more than one hidden layer is present, you arrive at a **deep network** like to right
  - Really just a lot more of the same, but far more weights to fit



Example

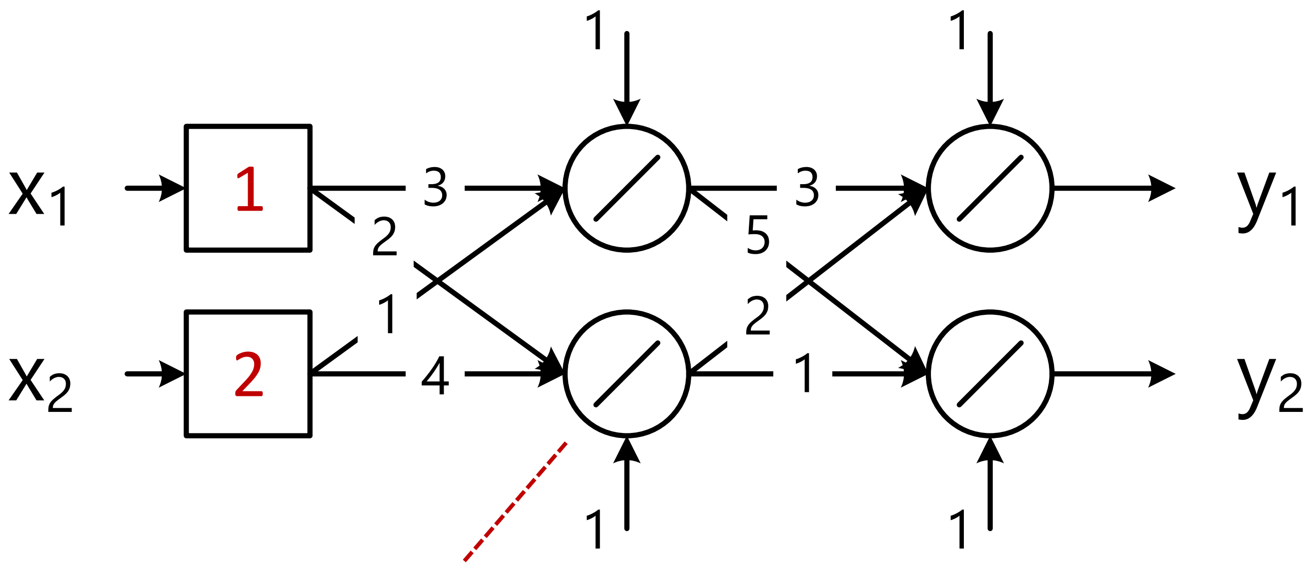
- Compute the outputs for the following ANN, assuming the inputs are $\boldsymbol{x} = [1 \quad 2]^T$
  - Assume all activation functions are linear: $\varphi(v) = v$
  - Try to represent the calculations as **weight matrices**



Note the symbol for a linear activation function is a slash (actually, it is a mini plot of $f(x) = x$)

Discussion: Weighting Matrices

- For this example, re-draw the ANN as an equivalent single-layer ANN
  - What does this mean regarding hidden layers using linear activation functions?



Note the symbol for a linear activation function is a slash (actually, it is a mini plot of $f(x) = x$)

Disadvantages: Linear Activations

- Using only linear activation functions is **equivalent** to using a single-layer ANN with different weights
  - This leads to convergence issues (why)?
  - Also REALLY limits our modeling capabilities (MLR only!)

- We had better get used to using different types of activation functions…

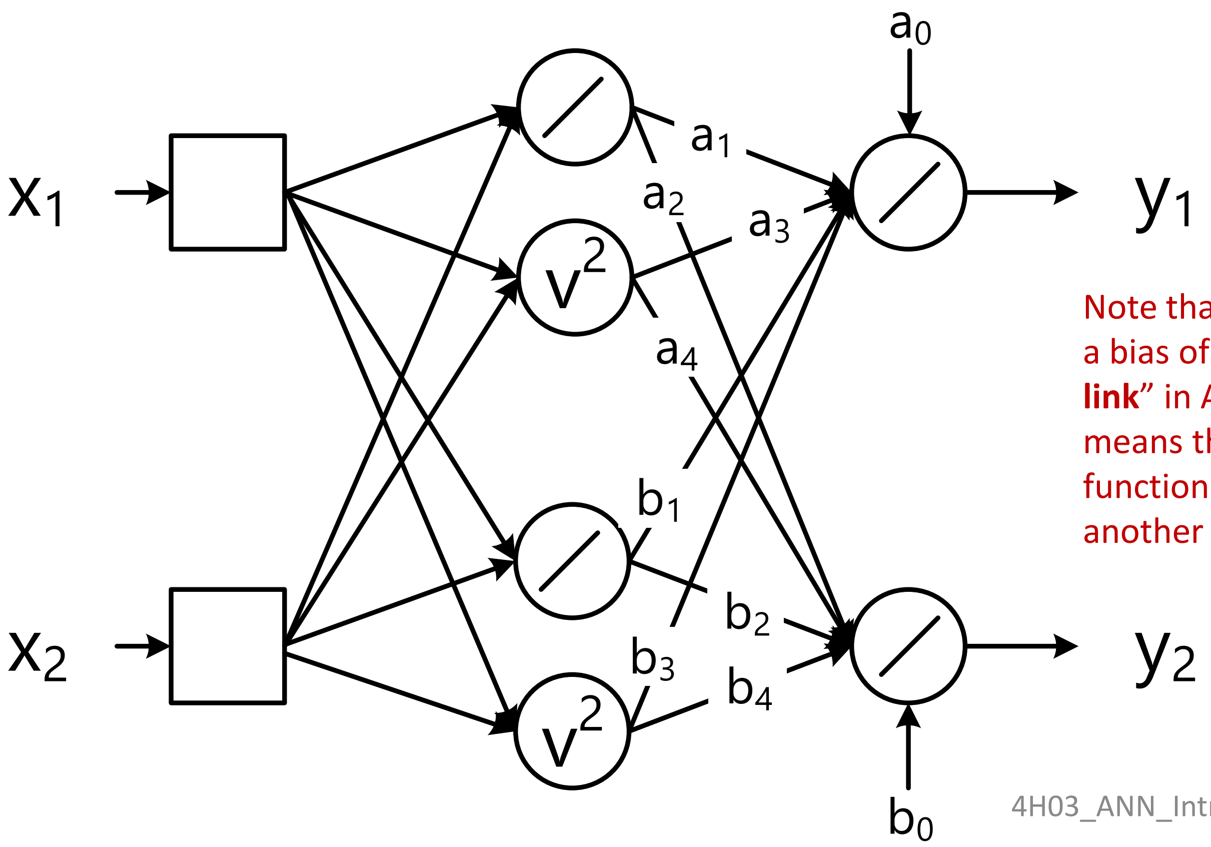
Examples: Different Activations

- PROBLEM 1
  - **DRAW** an ANN that performs simple linear regression for ONE variable exactly like we have covered : $y = a_0 + a_1 x$
  - What (is) are the activation function(s)? What are the weights?

- PROBLEM 2
  - **DRAW** an ANN that performs polynomial regression of order 4 (quartic regression): $y = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4$
  - Do the placements of w_k matter?

Examples: Different Activations

- PROBLEM 3
  - What are the regression equations for the ANN below?
  - Assume all biases are zero if not shown
  - Assume all weights are one if not shown



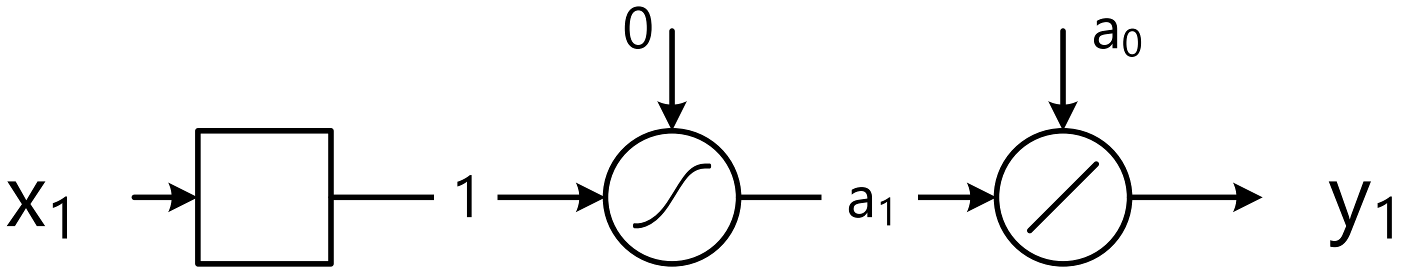
Note that assigning a weight of 1 and a bias of zero is called a "**functional link**" in ANN lingo, and basically means that we just want to use the function to transform the output of another node directly

Training your ANN

Practice Makes Perfect

Training: **Not** Linear in Parameters

- Up to this point, training a single-layer ANN with functional links between the inputs and hidden layer results in a **linear-in-the-parameters regression**:



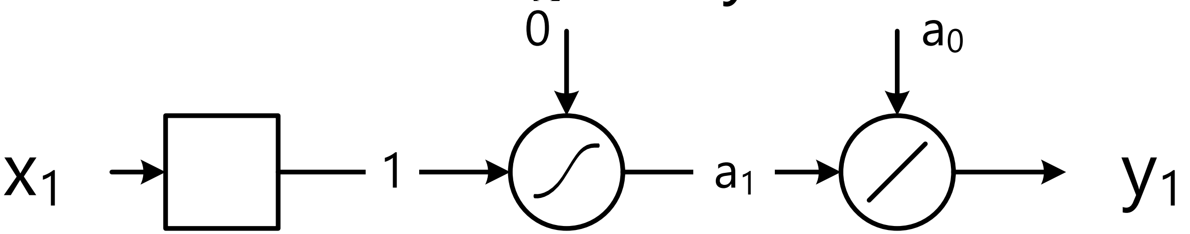
This is the symbol for the sigmoid activation function

$$\hat{y}_1 = a_1 \frac{1}{1+e^{-x_1}} + a_0$$

- We can solve the above using basis function regression!

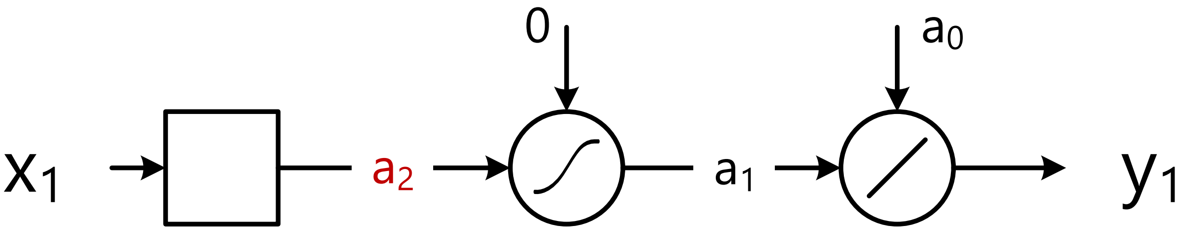
Training: **Not** Linear in Parameters

- HOWEVER, using many functional links is not ideal:



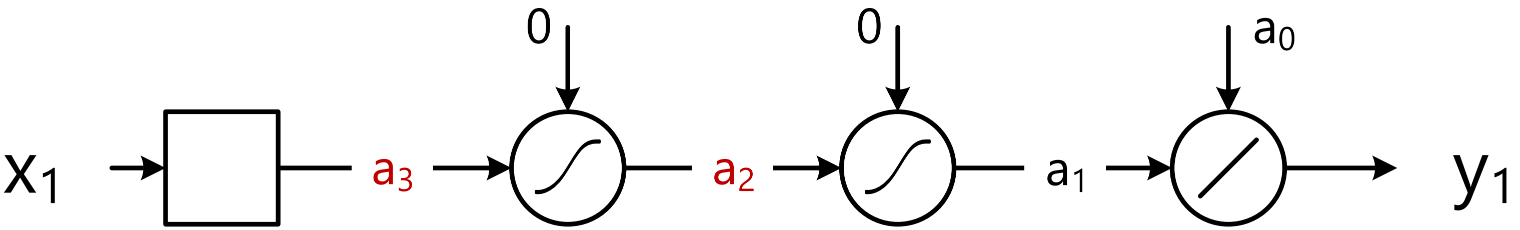
$$\hat{y}_1 = a_1 \frac{1}{1+e^{-x_1}} + a_0$$

This is OK! :)



$$\hat{y}_1 = a_1 \frac{1}{1+e^{-a_2 x_1}} + a_0$$

This is NOT OK! :S



$$\hat{y}_1 = a_1 \frac{1}{1+e^{-a_2\left(\frac{1}{1+e^{-(a_3 x_1)}}\right)}} + a_0$$

This is absolutely horrible *_*

So What Now?

- The bad news is that we can't analytically solve for the $\boldsymbol{w}$
  - Well, at least, we can't take the derivative of SSE and set it to 0
- The good news is that we clearly know the objective: we want $\hat{y}$ to be as close to y (training data) as possible
  - We can therefore do what we always do and quantify the ERROR
- Again, this leads to the nice lingo "supervised machine learning"
  - Which, as we now know, is just using known correct values to quantify an error so we can minimize it!
- At the end of the day, it all leads back to OPTIMIZATION

ANN Training Procedure

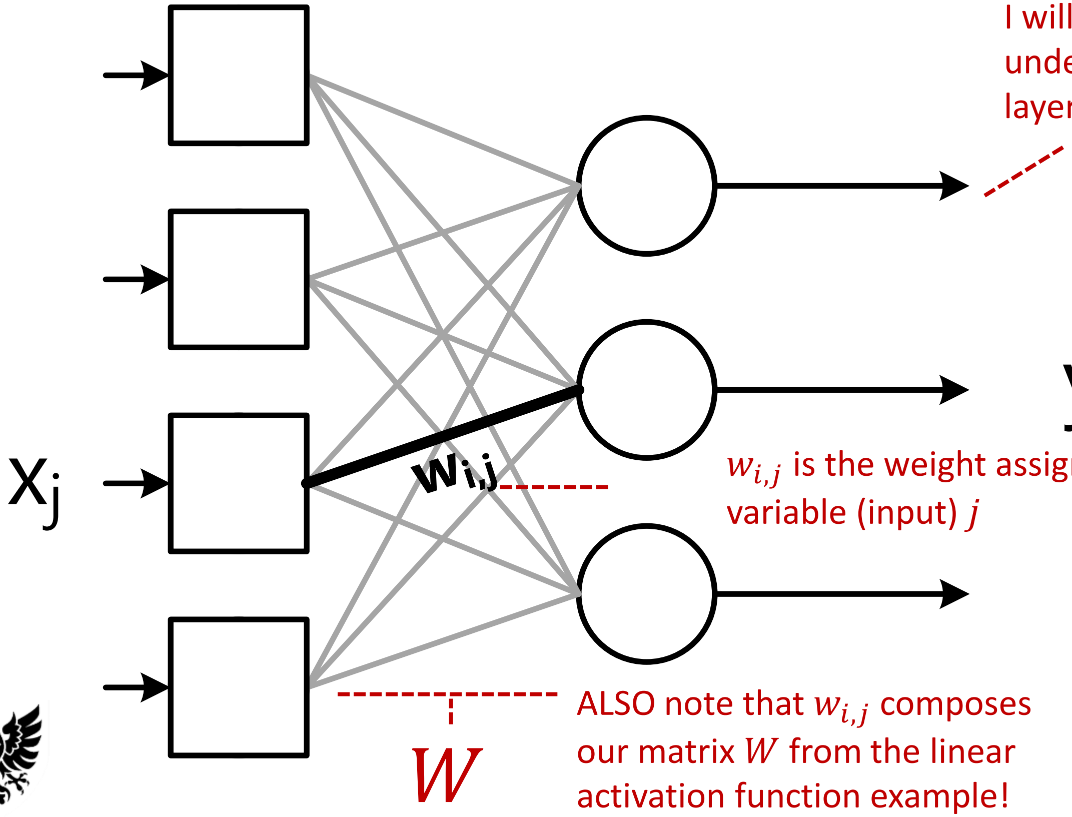
- The ANN training procedure is similar to NIPALS or any other training algorithm:

```
1. Initialize the weights $\boldsymbol{w}$
2. In a loop, until convergence
    1. FOR all data points $\boldsymbol{x}$
        1. Compute the ANN output $\hat{\boldsymbol{y}}$
        2. Compute the error $\boldsymbol{\epsilon}$
        3. Use the error $\boldsymbol{\epsilon}$ to (somehow) adjust the weights
    2. Check for convergence of $\boldsymbol{w}$
    3. Return to (2.1)
```

- **One iteration through step 2.1 is called a "generation" or EPOCH... It's a fancy ANN word for "iteration"**
- The weights may be adjusted once or several times throughout a single epoch

Step 1: Initialization

- Consider a single-layer neural network with J inputs and I outputs ($J = 4$ and $I = 3$ in this case):
  - We can choose anything we want as the initial weights $w_{i,j}$
  - It is our objective to update $w_{i,j}$ based on our friend ϵ_i



I will point out here that as long as we understand this for one layer, subsequent layers just use previous outputs as inputs (etc.)

$w_{i,j}$ is the weight assigned to output i from variable (input) j

ALSO note that $w_{i,j}$ composes our matrix W from the linear activation function example!

Step 2.1.1: Compute ANN Output

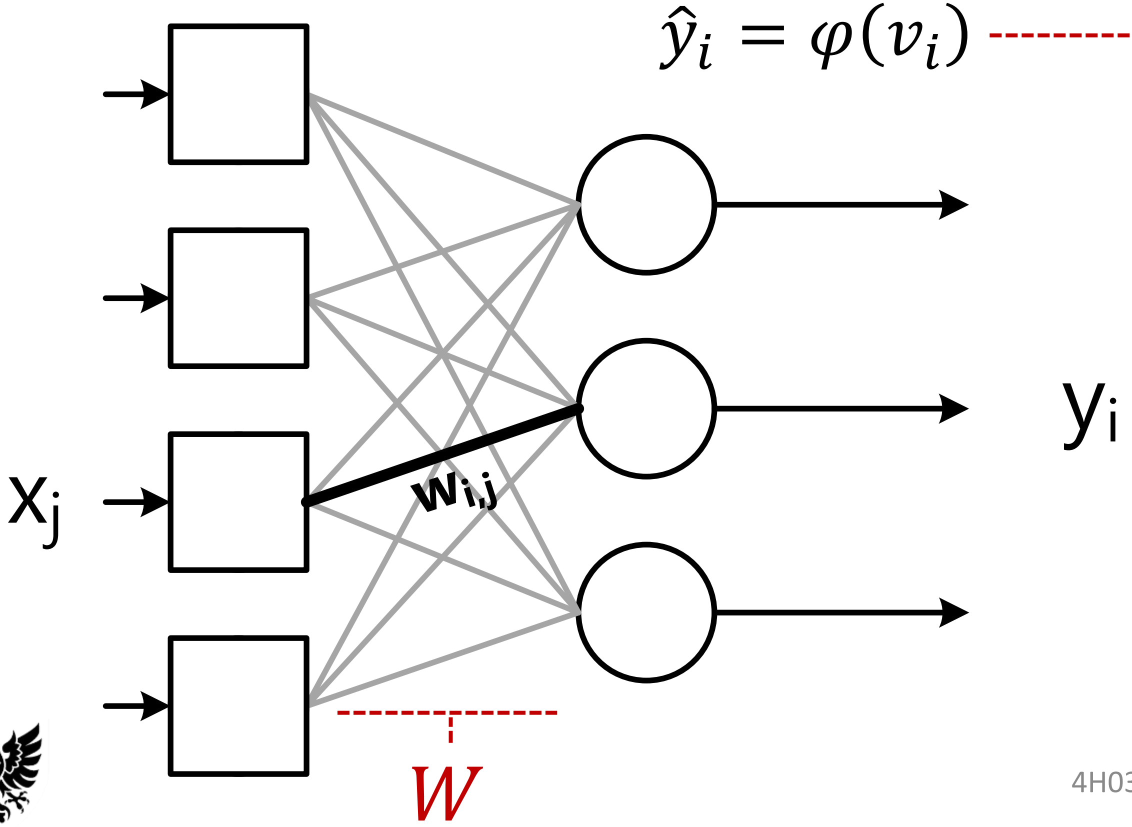
- The ANN output is calculated for each node to find each expected outcome $\hat{y}_i$

$$v_i = \sum_j w_{i,j} x_j + b_i$$

This is nothing more than the weighted sum plus bias for the input to node i

$$\hat{y}_i = \varphi(v_i)$$

This is the output from node i. For now, we'll assume that it is the real output

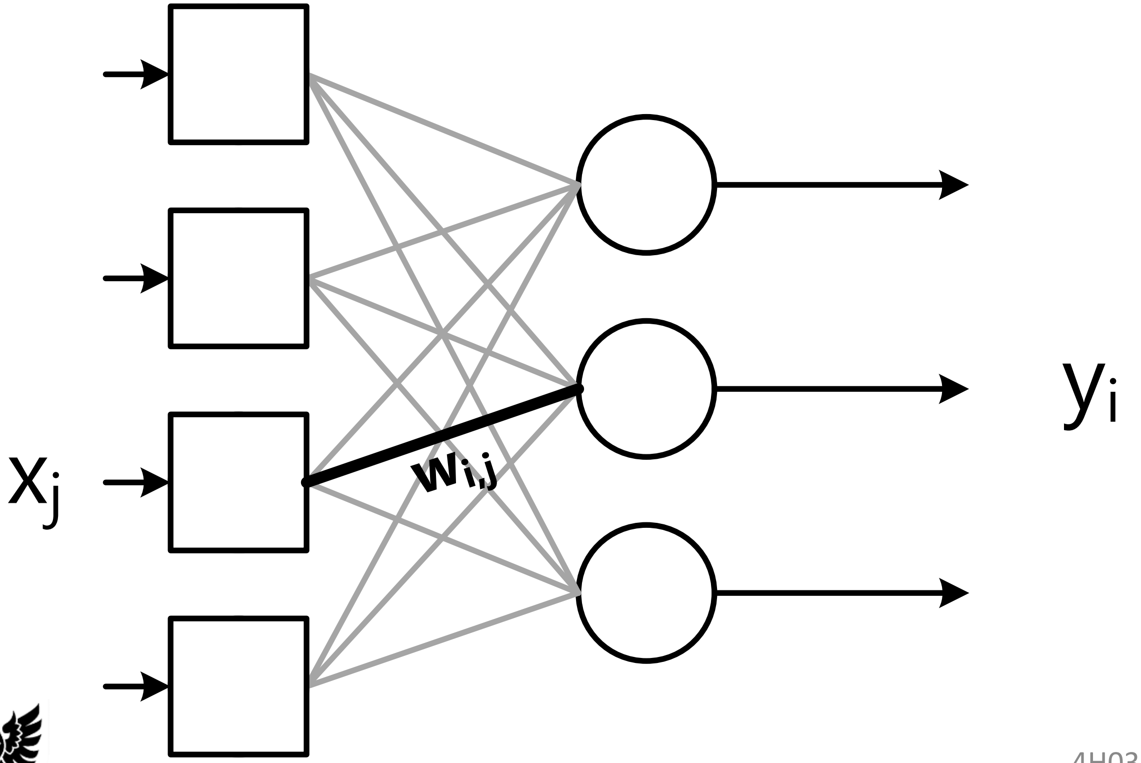


Shockingly, we'll compare this REAL y_i to the calculated $\hat{y}_i$

Step 2.1.2: Compute ANN Error

- Here's a shocker:

$$\epsilon_i = y_i - \hat{y}_i$$



Step 2.1.3: The General Idea

- Regardless of the method used, the overarching objective for all ANN model training is the same:
  - Quantify the model error, and then minimize it

- At the end of the day, you are solving

$$\min_{W,\boldsymbol{b}} \phi = \sum_{n=1}^{N} \sum_{i=1}^{I} (\hat{y}_i(\boldsymbol{x}_n) - y_i)^2 \triangleq \sum_{n=1}^{N} \sum_{j=1}^{J} (\epsilon_i)^2$$

You can have different objectives… This is just one

Recall W are all of our weights, $\boldsymbol{b}$ are biases

You can also impose CONSTRAINTS

Step 2.1.3: General Delta Rule

- Once the errors have been computed, update $w_{i,j}$ as:

$$w_{i,j} \leftarrow w_{i,j} + \alpha\delta_i x_j \quad \forall \quad i,j$$

This means “replace” This means “for all”

- We can compute the error function δ_i for some y_i as:

$$\delta_i = \varphi'(v_i)e_i$$

- Basically, this works out to be a **gradient descent**
  - α is the “learning rate” (AKA the aggressiveness of the algorithm)
  - x_j is the input node (or output of previous layer, thus the current layer’s input)

Step 2.1.3: General Delta SIGMOID

- If we are using a sigmoid activation function (common):

$$\varphi'(\gamma) = \frac{d}{d\gamma}\left(\frac{1}{1+e^{-\gamma}}\right) = \frac{1}{1+e^{-\gamma}}\left(1 - \frac{1}{1+e^{-\gamma}}\right) = \varphi(\gamma)[1-\varphi(\gamma)]$$

- And so:

$$\delta_i = \varphi'(v_i)\epsilon_i = \varphi(v_i)\big(1-\varphi(v_i)\big)\epsilon_i$$

- We can therefore use this value of δ_i in:

$$w_{i,j} \leftarrow w_{i,j} + \alpha\delta_i x_j \quad \forall \quad i,j$$

- The next step is to ask: **WHEN** do we update the weights?
  - This is all for a SINGLE input vector $\boldsymbol{x}$

Step 2.2: Updating the Weights

- There are THREE well-known options for this
- In each **epoch**, you can update the weights using
  - SGD (**S**tochastic **G**radient **D**escent)
  - Batch (just SGD but a chunky version)
  - Mini-Batch (LRRH's perfect porridge of the two)

- **SGD**
  - The values of $w_{i,j}$ are updated for **every data point**
  - Thus, if there are N_P training points each with J inputs and I outputs, one epoch updates all weights N_P times, or $N_P \times J \times I$ update calculations are performed
  - So, 100 training points each with 20 inputs and 10 outputs changes the weights 20,000 times in **ONE EPOCH**
  - Performance is usually over-aggressive and all over the place

Step 2.2: Updating the Weights

- **BATCH**
  - The values of $w_{i,j}$ are updated **once per epoch**
  - ALL weight updates are calculated for each point $\boldsymbol{x}_n$ ($\Delta w_{i,j,n}$), but only the average updated weight is used in each epoch
  - So, 100 training points each with 20 inputs and 10 outputs must compute 20,000 new weights in one epoch, but actually only changes them once (so 200 weight updates)

$$\Delta w_{i,j} = \sum_{n=1}^{N_P} \frac{\Delta w_{i,j,n}}{N_P}$$

Recall that $\Delta w_{i,j}$ $is\ \alpha\delta_i x_j$

  - Performance is slow but reliable

Step 2.2: Updating the Weights

- **MINI-BATCH**
  - The values of $w_{i,j}$ are updated $\boldsymbol{G}$ **times per epoch**
  - Training data split into G subgroups
  - Batch training is performed on each subgroup
  - Weight updates are calculated for each point $\boldsymbol{x}_n$ ($\Delta w_{i,j,n}$), but only the average updated weight is used in each subgroup g
  - So, 100 training points each with 20 inputs and 10 outputs must compute 20,000 new weights in one epoch, but actually only changes them $I \times J \times G$ times (so 200G weight updates)

$$\Delta w_{i,j} = \sum_{n \in g} \frac{\Delta w_{i,j,n}}{\frac{N_P}{G}}$$

Recall that $\Delta w_{i,j}$ is $\alpha \delta_i x_j$
This is applied for every subgroup

  - Performance is a decent mix of SGD and Batch

Step 2.2: Check for Convergence

- Convergence can be checked in any way you choose
- Each epoch (k) can be assessed using:
  - $\left\|W^{(k)} - W^{(k-1)}\right\| \leq \varepsilon$
  - $\frac{\left\|W^{(k)} - W^{(k-1)}\right\|}{\left\|W^{(k)}\right\|} \leq \varepsilon$
- Note that we have only looked at a single layer
  - Multi-layer networks behave similarly
  - However, we must look at each layer separately
  - Built-in packages have all kinds of fancy tools
- But we can do ONE simple example manually…

Example in MATLAB

- Let’s train a simple continuous classifier ANN in MATLAB
  - We’ll do it using **SGD** but can diverge to batch if desired

Review of "Clean Notation"

- It is often easiest to think of each layer's output as the "input" to the next layer:

$$\boldsymbol{y}^{(l)} = \varphi\left(\boldsymbol{v}^{(l)}\right)$$

The l means "layer l"

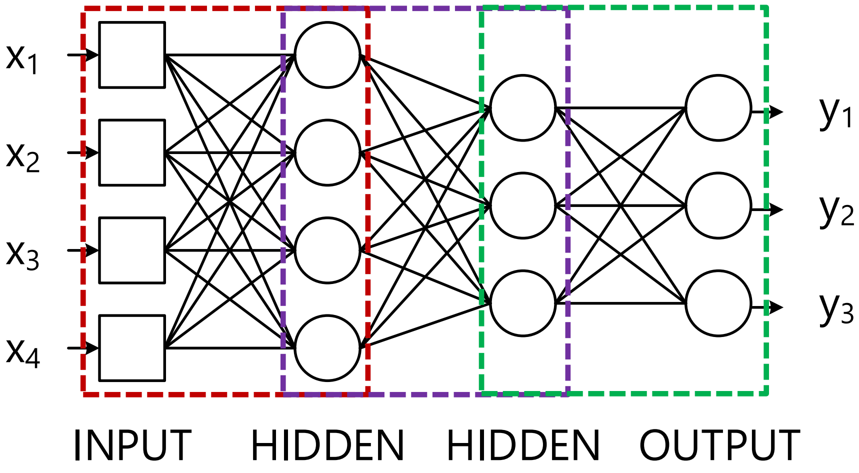
Note that $y^{(0)}$ is the OUTPUT of the input layer (layer 0)

$$\boldsymbol{y}^{(1)} = \varphi\left(W^{(1)}\boldsymbol{y}^{(0)} + \boldsymbol{b}^{(1)}\right)$$

Note that $b^{(l)}$ is the BIAS of layer l

$$\boldsymbol{y}^{(2)} = \varphi\left(W^{(2)}\boldsymbol{y}^{(1)} + \boldsymbol{b}^{(2)}\right)$$

$$\boldsymbol{y}^{(3)} = \varphi\left(W^{(3)}\boldsymbol{y}^{(2)} + \boldsymbol{b}^{(3)}\right)$$



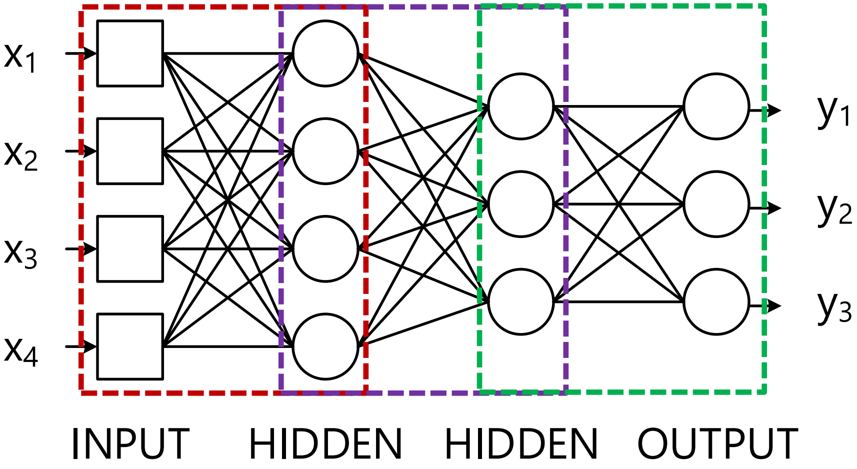
Review of "Clean Notation"

- So what do we really want? Even in deep learning? SGD is doing nothing more than taking the **gradient** of the error function and applying a modification to **all** weights W according to the objective:

$$\mathcal{E} = f\left(W^{(1)}, W^{(2)}, \cdots, W^{(l)}, \boldsymbol{b}^{(1)}, \boldsymbol{b}^{(2)}, \cdots, \boldsymbol{b}^{(l)}\right)$$

$$\boldsymbol{W} \leftarrow \boldsymbol{W} + \boldsymbol{\alpha} \boldsymbol{\nabla}_{\boldsymbol{W},\boldsymbol{b}} \boldsymbol{\mathcal{E}}$$

Are we actually going to do this? Well, no. But it helps to know what is going on and it makes DEEP NETWORKS much easier to understand!



Final Remarks

- We have examined the properties of a simple ANN
  - Activation functions
  - Weights
  - Objectives of machine learning
  - Relation to human brains
  - Numerical examples

- We have developed a method to train a single-layer ANN
  - Computing errors
  - Stochastic Gradient Descent (AKA just optimization)
  - Decision variables: Weights and biases
  - Matrix notation
  - MATLAB example
  - Connection to deep learning (coming up!)