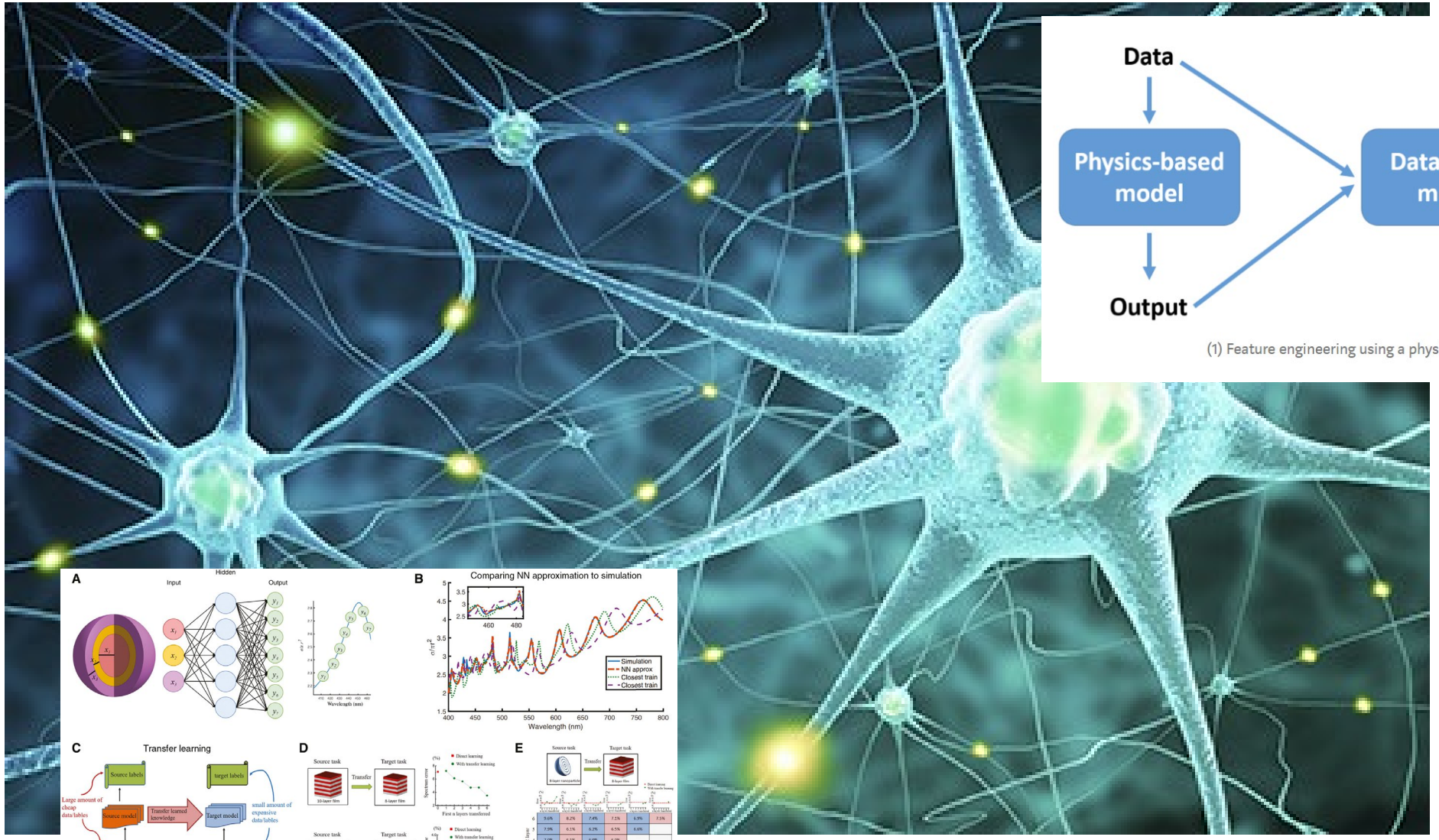
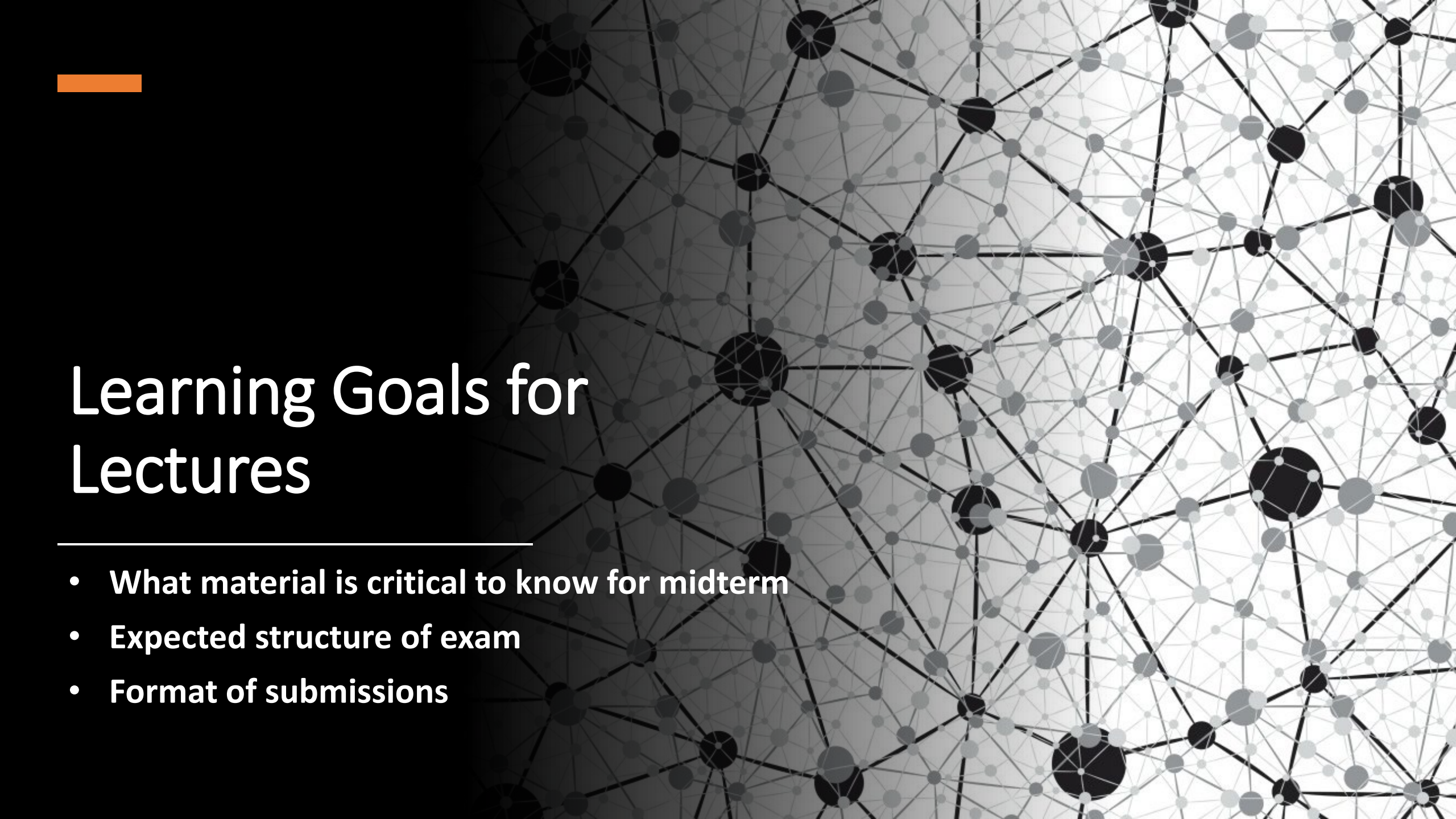
Review for Midterm



https://towardsdatascience.com/physics-guided-neural-networks-pgnns-8fe9dbad9414

https://www.neuraldump.net/2016/03/introduction-to-neural-networks/



Learning Goals for Lectures

- What material is critical to know for midterm
- Expected structure of exam
- Format of submissions

Midterm Exam

- **Format**
  - Take home
  - 5 or 6 problems
  - One problem may be a multi-part question where each part depends upon the previous
  - Any required programming will be similar to what was needed to successfully complete the homework assignments
  - Worth 25% of final grade (each problem worth 4 – 6pts towards final grade)

Midterm Exam

- **Requirements**
  - You must follow directions on the exam to receive full credit
  - If you do not show all of your work, you will lose pts
    - ✓ Example: if you need to rely upon orthogonality and you just state two data sets are orthogonal or not without showing proof, then you will lose credit
  - The text of the exam (solutions) must be neat and thoroughly detailed. If I have to guess at what you are saying or dig through your code to figure out what you are referencing, you will lose credit.
  - You should cut in sections of code into your exam document to add in the flow and presentation of the problem solution

Midterm Exam

- **Example:**

Problem 3 Write a computer program to implement a discrete Hopfield net to …..

Solution:

A Hopfield network is a fully interconnected iterative autoassociative NN. To solve this problem we must first derive the set of weights to store ….

Weight formation function

```
function [weightMatrix] = hf_training(trainingInput)
    trainingInputVector = reshape(trainingInput.', 1, []) - 1;
    weightMatrix = trainingInputVector' * trainingInputVector;
    weightMatrix = weightMatrix - diag(diag(weightMatrix));
end
```

Testing Code (max # of recognized patterns)

```
w = hf_training(input_1);
w = w + hf_training(input_2);
w = w + hf_training(input_3);
w = w + hf_training(input_4);
% w = w + hf_training(input_5);
% w = w + hf_training(input_6);
% w = w + hf_training(input_7);
% w = w + hf_training(input_8);
% w = w + hf_training(input_9);
% w = w + hf_training(input_0);

temp = reshape(input_1.', 1, []) .* 0.5;

for i = 1:1:1000
    r = randi([1, 35]);
    summation = temp(r);
    for j = 1:1:35
```

• • •

Testing Code (noisy inputs)

```
temp = reshape(dirty_2.', 1, []) .* 0.5;
```

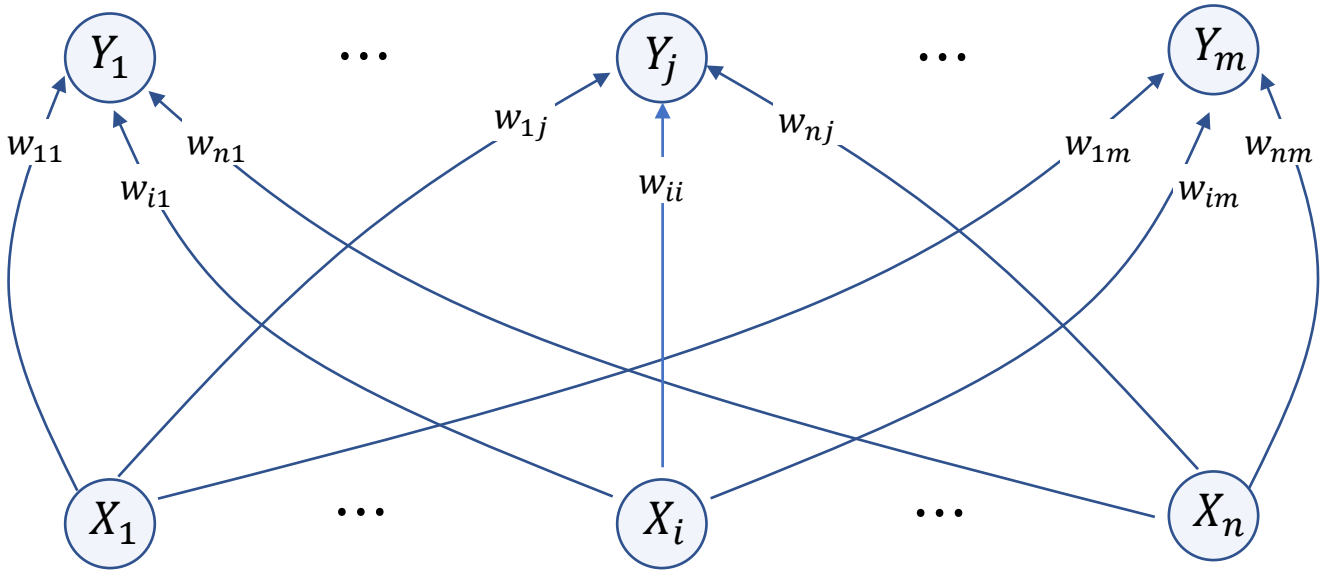
For this, I used a noisy representation of a 2. For this, it failed when trained on the 4 stored numbers that worked when we were testing the maximum recognized patterns. However, when we reduced it to 3 numbers, it was able to successfully recognize the noisy representation of the two and obtain the original input.

```
>> hf_testing

ans =

     0     1     1     1     0
     1     0     0     0     1
     0     0     0     0     1
     0     0     0     1     0
     0     0     1     0     0
     0     1     0     0     0
     1     1     1     1     1
```

For this network the results are:

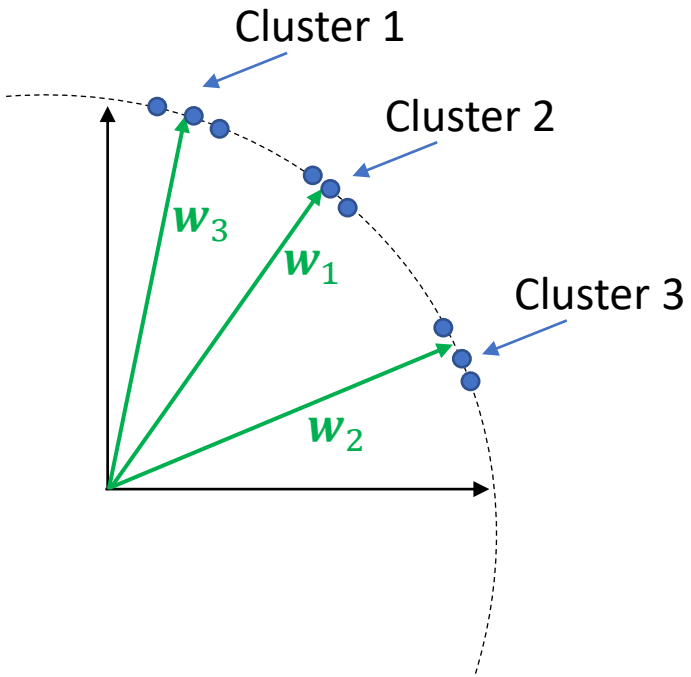
Competitive Networks: Self-Organizing Maps (SOMs)

- **Unsupervised learning**
- **Goal:**
  - Learn to form classes/clusters of exemplars/sample patterns according to similarities
  - Patterns in a cluster would have similar features
  - No prior knowledge as to what features are important for classification or how many classes
- **Architecture:**
  - Output nodes $Y_1,\ldots,Y_m$ can represent m classes
  - Theses output nodes compete with each other (WTA realized either by an external algorithm or by lateral inhibitions as in MAXNET).



Kohonen Self-Organizing Maps (SOMs)

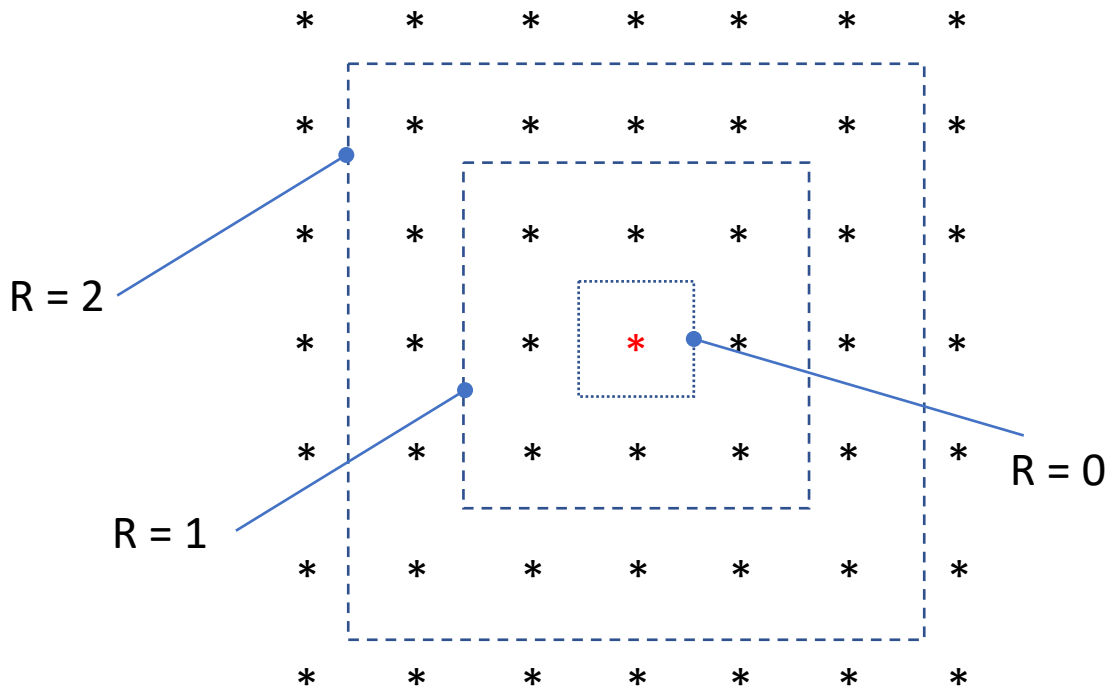
- **Competitive learning (Kohonen 1982) is a special case of SOMs**
- **In competitive learning:**
  - The network is trained to organize the input vector space into subspaces/classes/clusters
  - Each output node corresponds to one class
  - The output nodes are not ordered, i.e. mapping is random

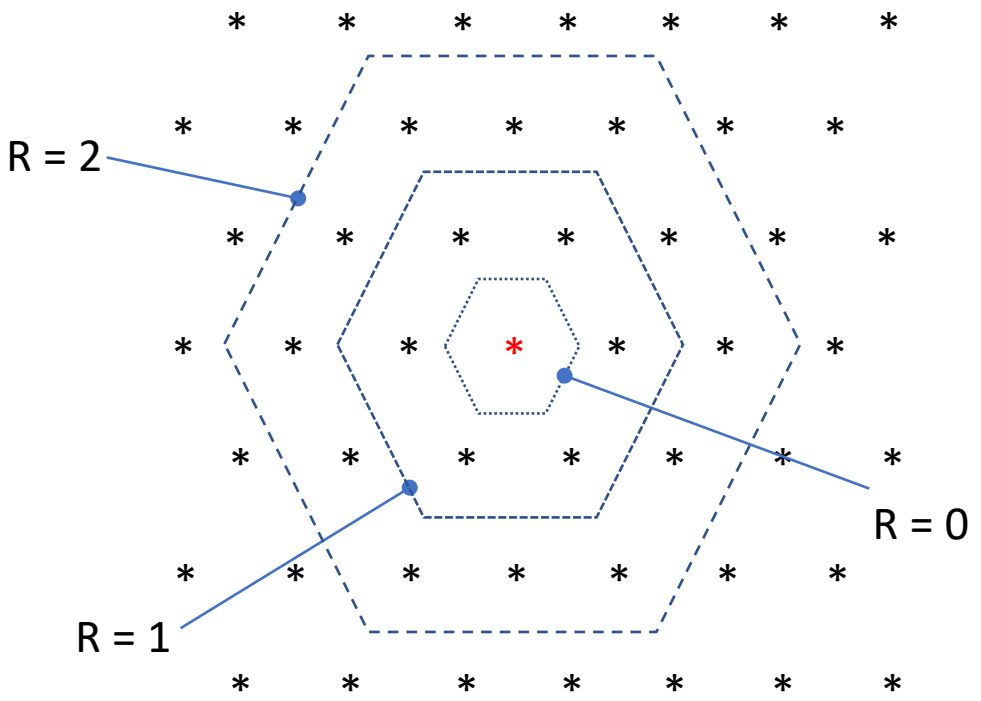


- The topological order of the three clusters is 1, 2, 3
- The order of their maps to the output nodes is: 3, 1, 2
- The mapping does not preserve the topological order of the training vectors

Kohonen Self-Organizing Maps (SOMs)

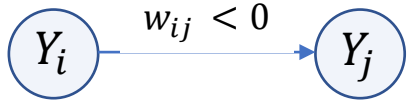
- **Topological mapping**
  - A mapping that preserves the neighborhood relationships between input vectors (topology preserving or feature preserving)
  - If $\mathbf{x}_1$ and $\mathbf{x}_2$ are two neighboring input vectors (by some distance metric), their corresponding winning output nodes (class mappings) i and j must also be close to each other in some fashion
  - One dimensional: line or ring, node i has neighbors i $\pm$ R
  - Two dimensional:
    - Rectangular grid – node (i,j) has neighbors [(i, j $\pm$ R), (i $\pm$ R, j)]
    - Hexagonal grid: 6 neighbors



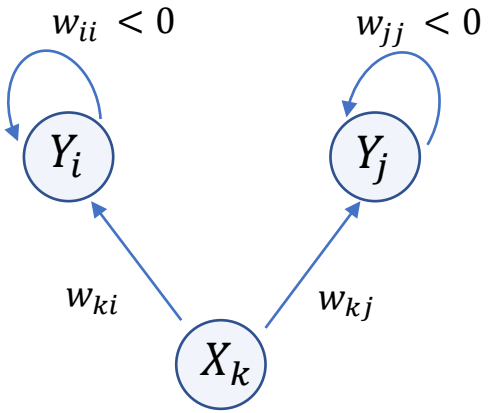


Neural Networks Based on Competition

- **Winner-takes-all (WTA)**
  - Among all competing nodes, only one will win and all others will lose
  - We primarily deal with single winner WTA, but multiple-winners WTA is possible for some particular applications
  - Easiest way to realize WTA – have an external, central arbitrator (algorithm) to decide the winner by comparing the current outputs of the competitors
- **Ways to realize competition in NNs**
  - Lateral inhibition (MAXNET, Mexican hat)
    - Output of each node feeds to the others through inhibitory connections (i.e. negative weights)
  - Resource competition
    - Output of x_k is distributed to y_i and y_j proportional to w_{ki} and w_{kj}, as well as y_i and y_j
    - Self decay

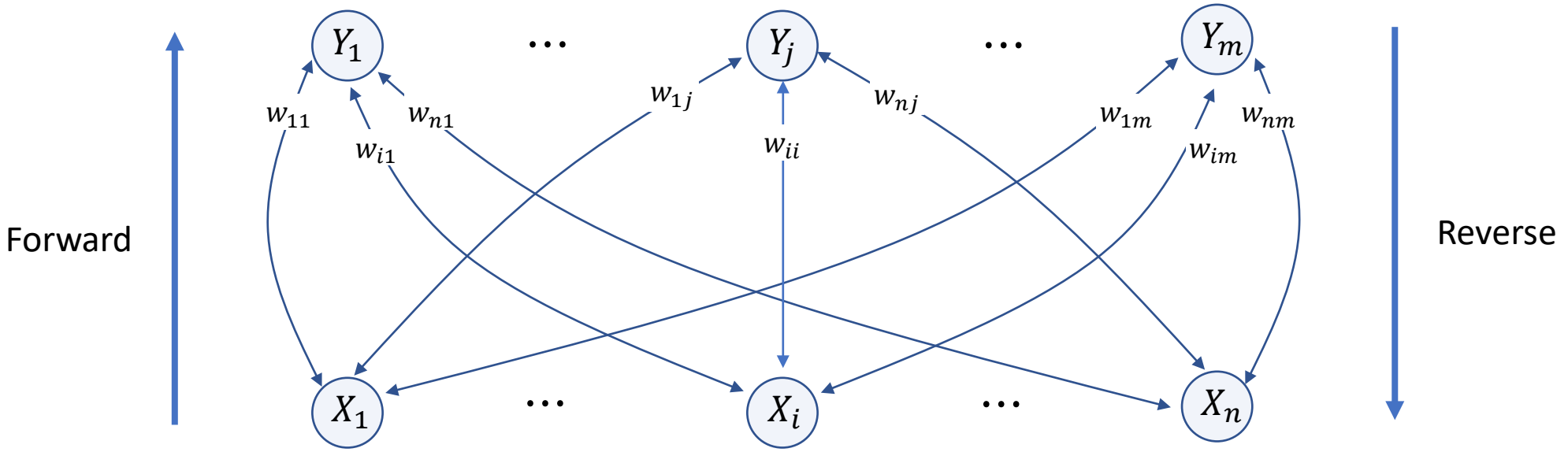


- **Learning methods in competitive networks**
  - Competitive learning
  - Kohonen learning (self-organizing map (SOM))
  - Counter-propagation net
  - Adaptive resonance theory (ART, chapter 5)



Bidirectional Associative Memory (BAM)

- A single-layer nonlinear feedback BAM network with heteroassociative content-addressable memory has n units in the X-layer and m units in the Y-layer
- Three types of BAMs: binary, bipolar, and continuous. Binary and bipolar are very similar with bipolar using providing better performance



Hopfield Application Algorithm (Binary Patterns)

- Step 0: Initialize weights to store patterns
- While activations of the net are not converged, do Steps 1-7:
  - ➢ Step 1: For each input vector, do Steps 2 – 6
    - ✓ Step 2: Set initial activations of the net equal to the external input vector, **x:**

$$y_i = x_i, (i = 1{:}n)$$

    - ✓ Step 3: Do Steps 4 to 6 for each unit Y_i (updated in random order):
      - Step 4: Compute net input:

$$y\_in_i = x_i + \sum_j y_j w_{ji}$$

      - Step 5: Determine activations:

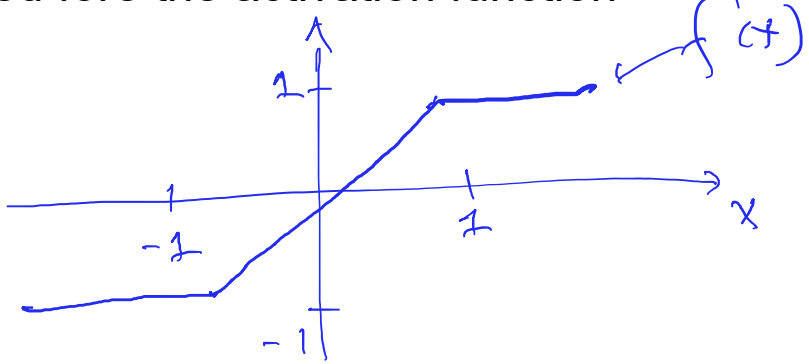
$$y_i = \begin{cases} 1, & if\, y\_in_i > \theta_i \\ y_i, & if\, y\_in_i = \theta_i \\ 0, & if\, y\_in_i < -\theta_i \end{cases}$$

      - Step 6: Broadcast the value of y_i to all other units to update the activation vector
  - ➢ Step 7: test for convergence

- θ_i usually set at zero
- Originally, only external at first time step, later formulated to allow during computation
- Order of update is random, but each unit is updated at the same average rate
- Binary activations are typical, but bipolar formulations are also used
- Convergence test = stable energy?

Brain State in a Box (BSB) Network

- **Another Iterative Associator called BSB was proposed by Anderson, et al., 1977**
- **The output of a linear autoassociator can grow without bound**
- **This can be prevented by modifying the activation function (which is the identify function for the linear associator) to take on values with a cube: that is the activation function produces values that are restricted to between -1 and 1.**
- **This is known as a BSB**
  - In a BSB, like all other similar type nets, there are n units connected to every other unit
  - There is also a trained weight on the self-connection (diagonal terms in the weight matrix), however, Anderson et al., show that setting the diagonal weights to zero (to represent a more biologically plausible model) does not change the performance of the net significantly.
  - A threshold can also be applied fore the activation function



$$f(x) = \begin{cases} 1, & x > 1 \\ x, & -1 \le x \le 1 \\ -1, & x < -1 \end{cases}$$

Recurrent Linear Autoassociator for Pattern Association

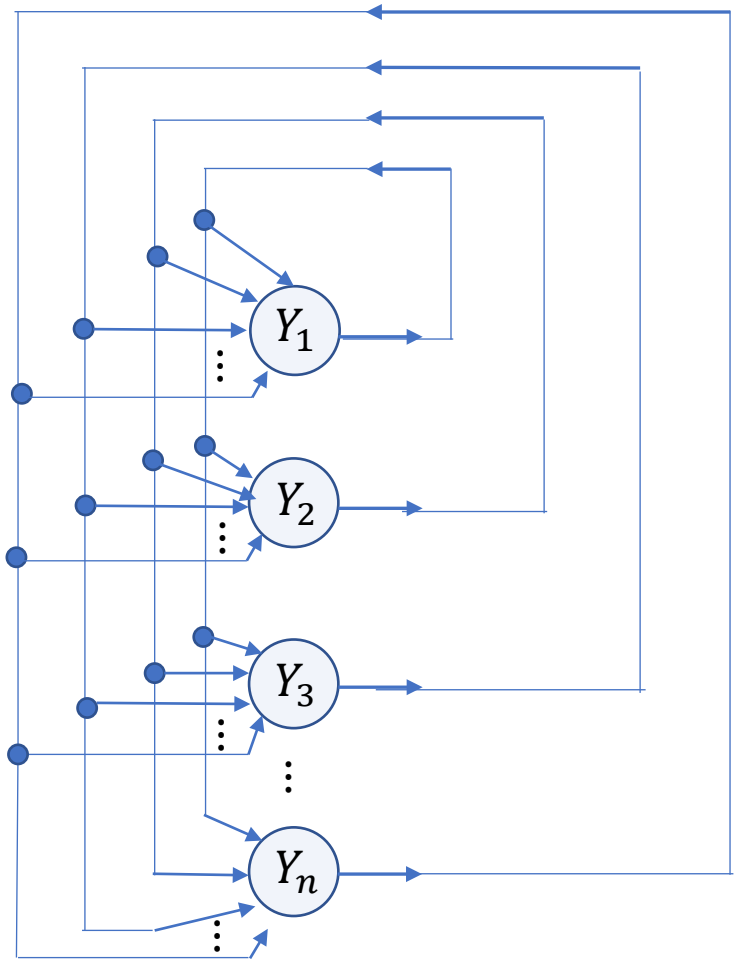
- **Recurrent Linear Autoassociator**
  - Running the recurrent net once results in:

$$\begin{aligned} a(1) &= a(0)W \\ &= (\alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n)W \\ &= \alpha_1 \lambda_1 v_1 + \alpha_2 \lambda_2 v_2 + \ldots + \alpha_n \lambda_n v_n \end{aligned}$$

  - Running the net for t iterations results in:

$$a(t) = \alpha_1 \lambda_1^t v_1 + \alpha_2 \lambda_2^t v_2 + \ldots + \alpha_n \lambda_n^t v_n$$

  - This results in the largest eigenvalue, λ_1, dominating the expression for large values of t.



Pattern Association

- **Associative Memory: brainlike distributed memory that learns by association**
  - Storing and retrieving is based on content rather than storage address
  - In a neural network, information is stored in the weights throughout the system → instantiation
  - Autoassociation (**s = t**) – a neural network stores a set of patterns (vectors) by repeatedly being exposed to them. Unsupervised learning
    - Subsequently presented with partial or noisy versions of an original training pattern and it recalls the output pattern associated with the original training vector
    - If stimulus is similar to the original stimuli used to form an association, it will create the same associated response (e.g. humans recognize various photos of a particular person in different settings).
  - Heteroassociation (**s** ≠ **t**) – an arbitrary set of input patterns is paired with an arbitrary set of output patterns. Supervised learning

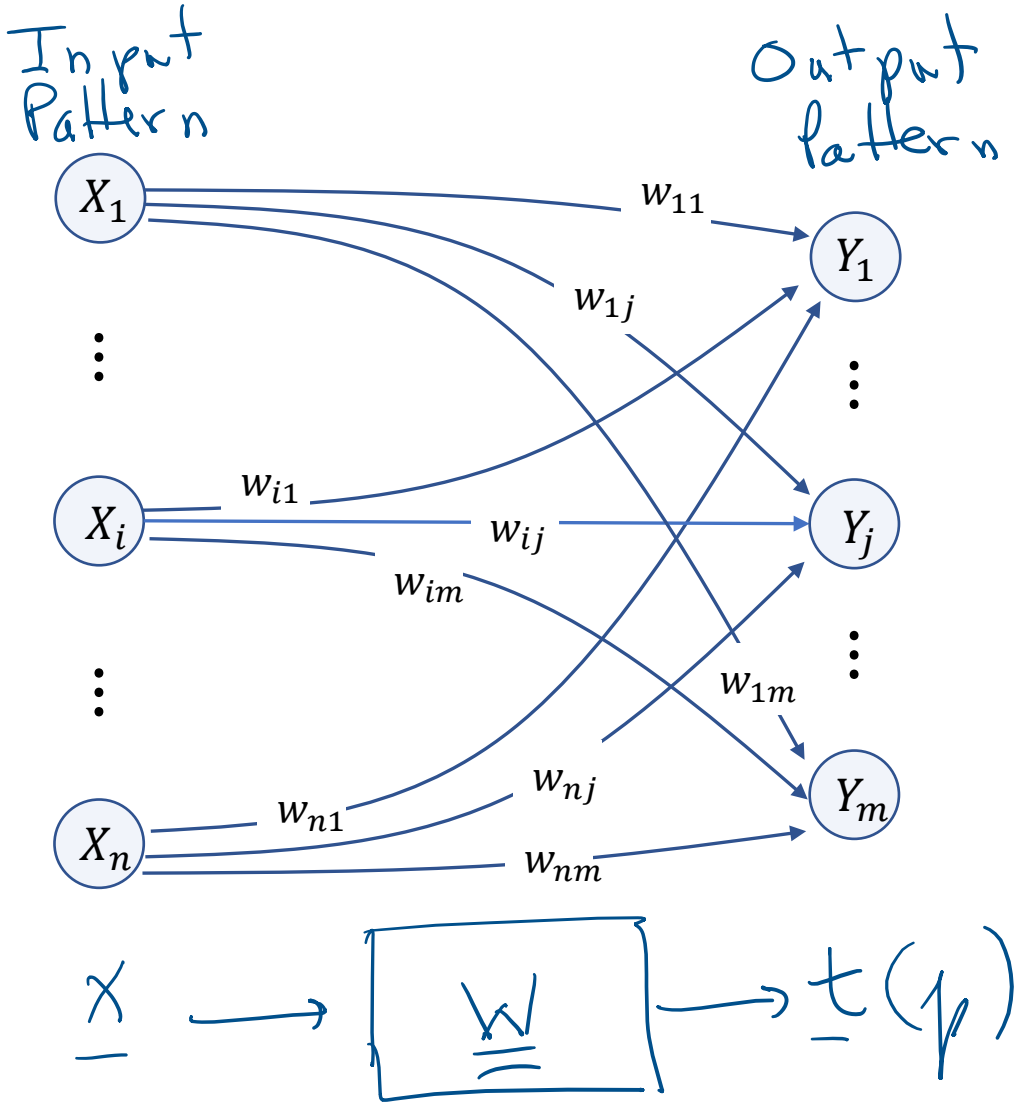
Architectures of an Associative Memory

1. **Feedforward –** information flows from input neurons to output neurons (3.2, 3.3)
2. **Recurrent (iterative) –** connections in the net form closed loops (3.4, 3.5)

* weights are determined to permit the storage of P pattern associations

Pattern Association (Heteroassociative)

- **Heteroassociative Memory Neural Networks**



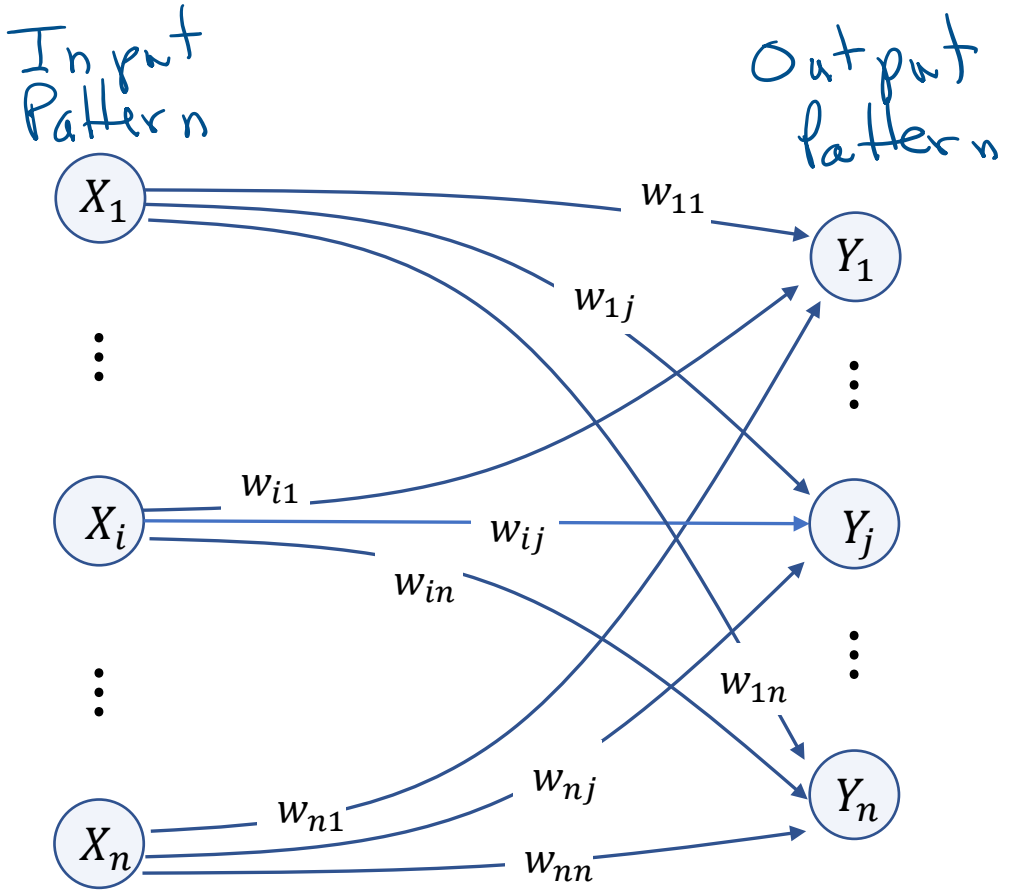
For a given input vector $\underline{x}$ the net as determined by the weights will select one stored pattern $\underline{t}(p)$ or a new pattern

* Note: for heteroassoc. memory

$$\underline{x} \in \mathbb{R}^{n\times 1} \Rightarrow \underline{t} \in \mathbb{R}^{m\times 1}$$

Pattern Association (Autoassociative)

- **Autoassociative Memory Neural Networks (s = t)**



$\underline{x} \longrightarrow \boxed{\underline{\underline{W}}} \longrightarrow \underline{t}(p)$

For a given input vector $\underline{x}$ the net as determined by the weights will select one stored pattern $\underline{t}(p)$ or a new pattern

* Note: for autoassociative memory

$\underline{x} \in \mathbb{R}^{n\times 1} \Rightarrow \underline{t} \in \mathbb{R}^{n\times 1}$

Pattern Association (single-layer nets)

- **Hebb rule for Pattern Association**
  - Binary or bipolar patterns
  - Recall that the Hebb rule adjusts weights and adjustable bias based on the target value and the input data

$$\Delta w_{kj}(n) = \eta y_k(n) x_j(n)$$

$$w_i(new) = w_i(old) + y x_i$$

  - For implementation, outer products ($\otimes$) are used. All weights for the Hebb rule may be found by using the outer product.

Outer Prdt. of $\underline{s}$ & $\underline{t}$

$\boldsymbol{s} \in \mathbb{R}^{nx1}$

$\boldsymbol{t} \in \mathbb{R}^{mx1}$

$$\boldsymbol{s} = \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_i \\ \vdots \\ s_{n-1} \\ s_n \end{bmatrix} \quad \mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_j \\ \vdots \\ t_{m-1} \\ t_m \end{bmatrix}$$

$$\underline{s} \otimes \underline{t} = \underline{s}\,\underline{t}^T = \begin{bmatrix} s_1t_1 & s_1t_2 & \cdots & s_1t_j & \cdots & s_1t_m \\ s_2t_1 & s_2t_2 & \cdots & s_2t_j & \cdots & s_2t_m \\ \vdots & & & & & \\ s_it_1 & s_it_2 & \cdots & s_it_j & \cdots & s_it_m \\ \vdots & & & & & \\ s_nt_1 & s_nt_2 & \cdots & s_nt_j & \cdots & s_nt_m \end{bmatrix}$$

Perfect Recall vs. Cross-Talk

The suitability of the Hebb rule is dependent upon the correlation among the input training vectors

- **If the input vectors are orthogonal (uncorrelated) the Hebb rule will produce the correct weights and the correct output for a given training input = perfect recall**
- **If the input vectors are not orthogonal, the response includes contributions from each of the correlated inputs' target values = cross-talk**

Two vectors are said to be orthogonal if their dot product = 0

$$\langle \underline{s}(k), \underline{s}(p) \rangle = \underline{s}^T(k)\,\underline{s}(p) = 0 \Rightarrow \text{orthogonal}$$

$$= \sum_{i=1}^{n} s_i(k)\, s_i(p) = 0$$

Delta Rule (Independent but not orthogonal)

Original Delta Rule:

- Activation fnc. $\Rightarrow f(x) = x$

$\therefore y_j = \sum x_i w_{ij}$ (i.e. the input to node j)

and

$$w_{ij}(new) = w_{ij}(old) + \alpha (t_j - y_j) x_i$$

$$\Delta w = \alpha (t_j - y_j) x_i$$

This is the delta rule we previously developed based on the method of Steepest descent.

Extended Delta Rule:

- Activation function here is an arbitrary but differentiable fnc.

$$\Delta w_{IJ} = \alpha (t_J - y_J) x_I f'(y_{in_J})$$

* Change the wts. based on the Δ between the computed output (y_J) and target **value** (t_J) rather than the input, (y_j) and target (t_j).

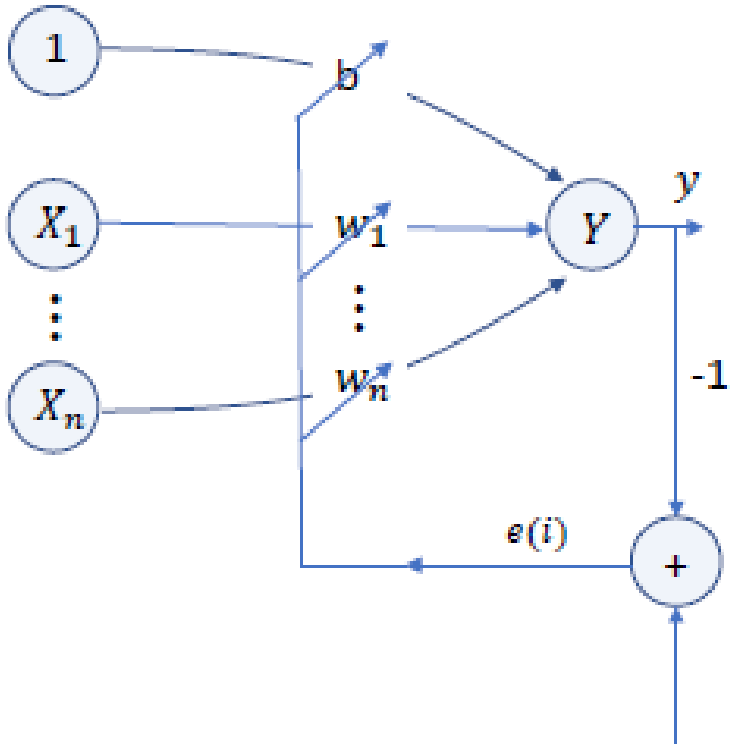
* This changes resulting updat **expression.**

ADALINE (Adaptive linear neuron)

Training Algorithm for ADALINE net:

- Step 0:
  - Initialize weights:
    - ✓ w_i = small random values, i = 1:n, b = small random value
- Step 1: While delta in weights is above a threshold:
  - Step 2: For each training vector (s) and target output (t)
    - Step 3: Set activations for input units
      - ✓ $x_i = s_i$, i = 1:n
    - Step 4: Compute net input to neuron:
      - ✓ $v(n) = \sum_{i=0}^{m} w_i(n)x_i(n) = \boldsymbol{w}^T(n)\boldsymbol{x}(n)$
    - Step 5: Update bias and weights
      - ✓ $w_i(new) = w_i(old) + \alpha(t - v(n))x_i$, i = 1: n
      - ✓ $b(new) = b(old) + \alpha(t - v(n))$
  - Step 6: if largest Δw_n is less than stop, else continue

ADALINE Architecture



MADALINE (MRI)

Training Algorithm for MADALINE Rule 1 (MRI) net:

- Only the HIDDEN weights are adjusted; output weights are fixed
- Activation function for Z_1, Z_2, Y:

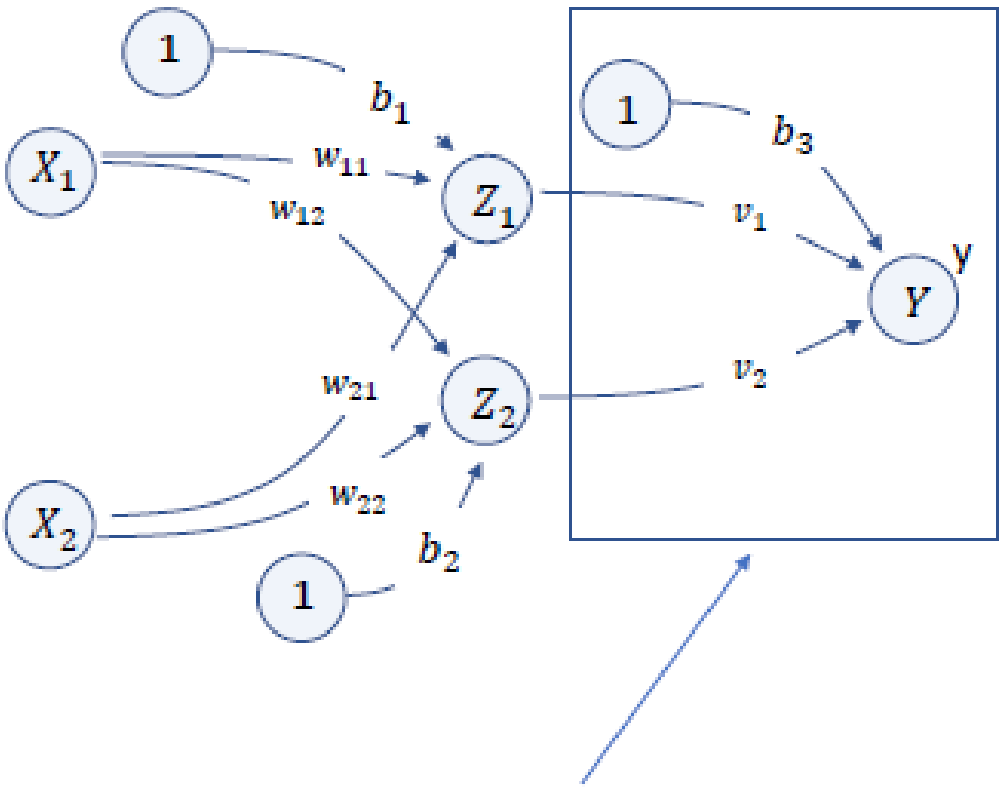
$$f(v) = \begin{cases} 1 & if\ v \geq 0 \\ -1 & if\ v < 0 \end{cases}$$

- Step 0:
  - Initialize weights and learning rate
    - ✓ Adaline w_{ki} = small random values, i = 1:n, b_k = small random value
- Step 1: If weight changes have not stopped or if a max number of iterations not reached:
  - Step 2: For each training vector (**s**) and target output (t)
    - Step 3: Set activations for input units
      - ✓ $x_i = s_i$, i = 1:n
    - Step 4: Compute net input to hidden Adaline neurons:

$$z\_in_1(n) = b_1 + x_1 w_{11} + x_2 w_{21}$$

$$z\_in_2(n) = b_2 + x_1 w_{21} + x_2 w_{21}$$

MADALINE Architecture



Weights and bias set so this performs an OR function:

$$v_1 = v_2 = b_3 = \frac{1}{2}$$

MADALINE (MRII)

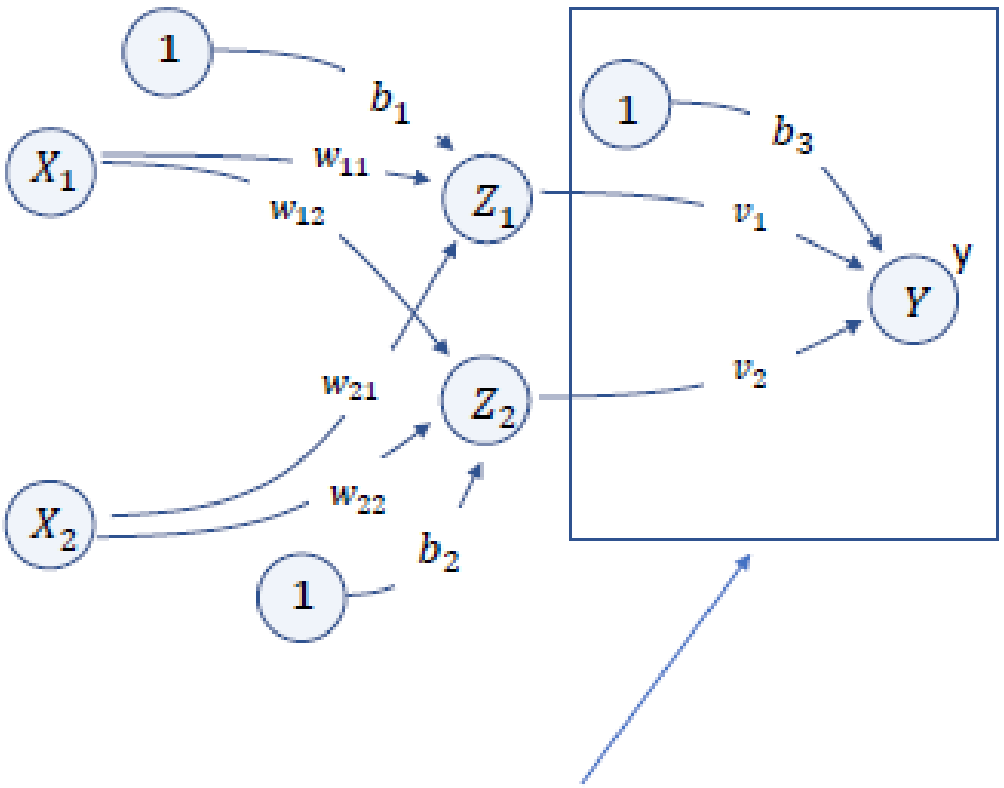
Training Algorithm for MADALINE Rule 2 (MRII) net:

- Slightly different approach to minimal necessary updating of net
- Activation function for Z_1, Z_2, Y:

$$f(v) = \begin{cases} 1 & if\ v \geq 0 \\ -1 & if\ v < 0 \end{cases}$$

- Step 0 – 6 same as MRI:
  - Step 7: Determine the error and update the weights and bias:
    - If t = y, No weight updates
    - else if $t \neq y$, then for each hidden node within a delta of 0 (e.g. $-0.25 \leq \Delta \leq 0.25$) update weights starting with the node with y_in closest to 0 (input is closest to 0)
      - Change that nodes output from +1 to -1 (or vice versa)
      - Recompute the new output of the net
      - If the new output results in a reduced error, adjust that node's weights using its changed output as the target and apply Delta Rule
  - Step 8: If no weight changes or minimal change or if reached a max number of iterations, stop, otherwise, continue

MADALINE Architecture



Weights and bias set so this performs an OR function:

$v_1 = v_2 = b_3 = \frac{1}{2}$

Training Single-Layer NN for Pattern Classification

Three Rules:

1. Hebb (train weight by neuron behavior)
2. Perceptron learning (iterative training)
3. Delta (minimize error over all training patterns)

We're not ready to discuss these yet….

Perceptrons for Pattern Classification

- **Perceptron learning rule is based on iterative adjustments to the weights associated with neurons that sent non-zero signals if there were errors. Uses bias and a modified threshold.**
- **Define the input vector (binary or bipolar) as:**

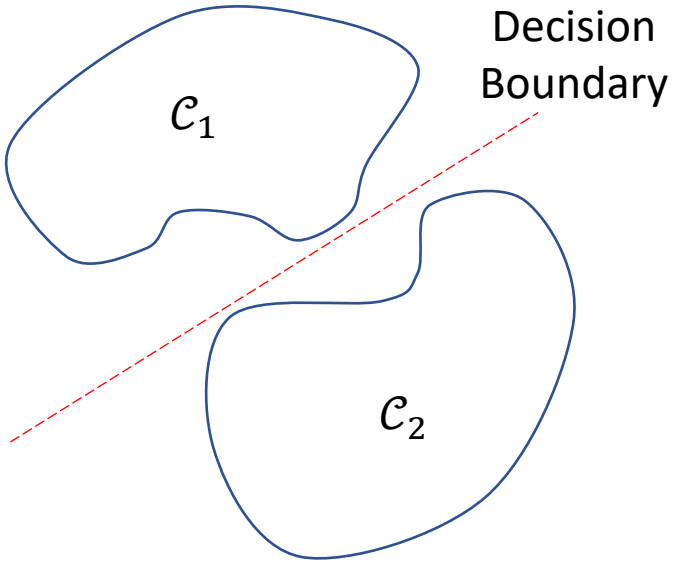
$$\boldsymbol{x}(n) = [+1, x_1(n), x_2(n), \ldots, x_m(n)]^T$$

- **the weight vector as:**

$$\mathbf{w}(n) = [b, w_1(n), w_2(n), \ldots, w_m(n)]^T$$

- **and the linear combiner output as:**

$$\mathrm{v}(n) = \sum_{i=0}^{m} w_i(n)x_i(n) = \boldsymbol{w}^T(n)\boldsymbol{x}(n)$$



- **The equation $w^T(n)x(n)$ = 0 is a hyperplane in m-dimensional space = *decision surface***

Perceptrons for Pattern Classification

- **The activation function for the output neuron is (Block '62):**

$$\mathrm{f}(v(n)) = \begin{cases} 1, & if\ v(n) > \theta \\ 0, & if\ -\theta \leq v(n) \leq \theta \\ -1, & if\ v(n) < -\theta \end{cases}$$

- **Learning rule:** for each input, calculate an output and form an error value. The weights should then be adjusted in the direction indicated by the 'target' value t = (1, -1) or desired value. **Only** the weights associated with a positive signal are adjusted since they contributed to the error (α is the learning rate):

$$w_i(new) = w_i(old) + \alpha t x_i$$

- Training continues until no error is generated

Hebb Net

Hebb Rule

- *Hebb's postulate of learning* is the most famous and oldest of learning rules named after a neuropsychologist Hebb (1949):
  - *When an axon of Cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic changes take place in one or both cells such that A's efficiency as one of the cells firing B is increased.*
- Extended Hebb rule:
  1. If two neurons on either side of a synapse (connection) are activated simultaneously (i.e. synchronously), then the strength of that synapse is selectively increased.
  2. If two neurons on either side of a synapse are activated asynchronously, then that synapse is selectively weakened or eliminated.
- Given a synaptic weight w_{kj} of neuron k with presynaptic and postsynaptic signals denoted x_k and y_k , then in general ***Hebbian learning*** may be expressed as:

$$\Delta w_{kj}(n) = f(y_k(n), x_j(n))$$

- The simplest form of this is:

$$\Delta w_{kj}(n) = \eta y_k(n) x_j(n)$$ Activity product rule

- η is a positive constant that determines the ***rate of learning.***

Hebb Net

- If η is taken as = 1, then the adaptive weight updating algorithm for a single-layer feedforward network is further simplified as:

$$w_i(new) = w_i(old) + yx_i$$

- Algorithm for Hebb net:

1. Initialize weights: $w_i = 0$, i = 1:n
2. For each input training vector (**s**) and target output (t) pair:
   i. Set activations for input units:

      $x_i = s_i$, i = 1:n

   ii. Set activation for output unit:

      y = t

   iii. Adjust the weights

      $$w_i(new) = w_i(old) + yx_i, \ \mathrm{i} = 1{:}\,\mathrm{n}$$

      Adjust the bias

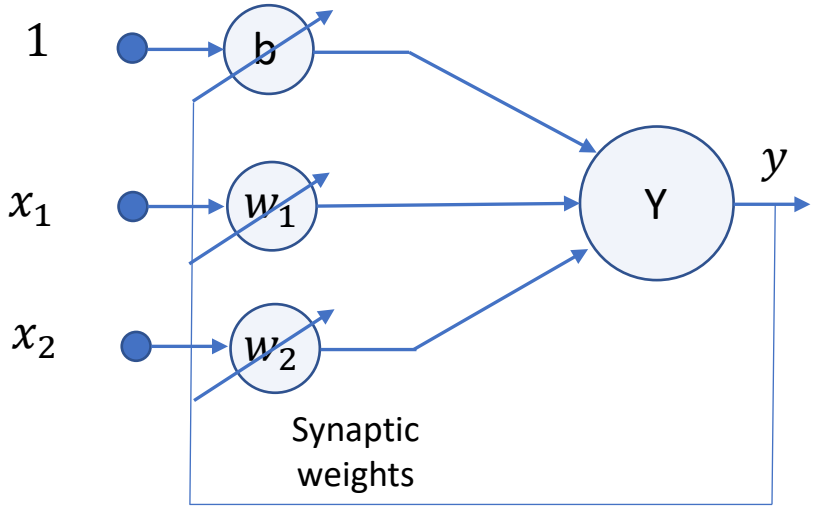
      b(new) = b(old) + y (recall $x_0 = 1$ for the bias path)

Hebb Net

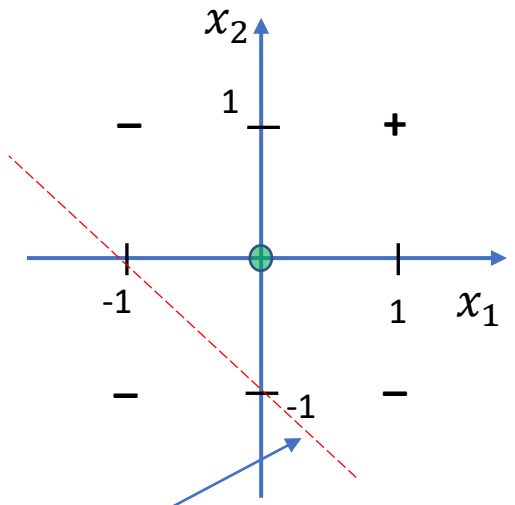
- Example of Hebb learning algorithm applied to SLNN (logical AND function):

$w_1(0) = 0 \;\; w_2(0) = 0 \;\; b(0) = 0$

Initial: (0, 0, 0)



x_1	x_2	b	t
1	1	1	1
1	-1	1	-1
-1	1	1	-1
-1	-1	1	-1



$$x_2 = -\frac{w_1}{w_2}x_1 - \frac{b}{w_2}$$

$x_2 = -x_1 - 1$

(w_1, w_2, b)

Time step 1: (1, 1, 1)

$\Delta w_i(n) = x_i(n)y(n)$

$\Delta w_1(1) = x_1(1)y(1) = 1 * 1 = 1$ $\quad$ $\Delta w_2(1) = x_2(1)y(1) = 1 * 1 = 1$ $\quad$ $\Delta b(1) = y(1) = 1$

$w_1(1) = w_1(0) + \Delta w_1(1) = 0 + 1 = 1$ $\quad$ $w_2(1) = w_2(0) + \Delta w_2(1) = 0 + 1 = 1$ $\quad$ $b(1) = b(0) + \Delta b(1) = 0 + 1 = 1$

M&P: Biases and Thresholds

If a bias is included, the activation function is typically written as:

$$f(v) = \begin{cases} 1 & if\ v \geq 0 \\ -1 & if\ v < 0 \end{cases} \qquad v = \sum_{j=1}^{n} w_j x_j + b$$

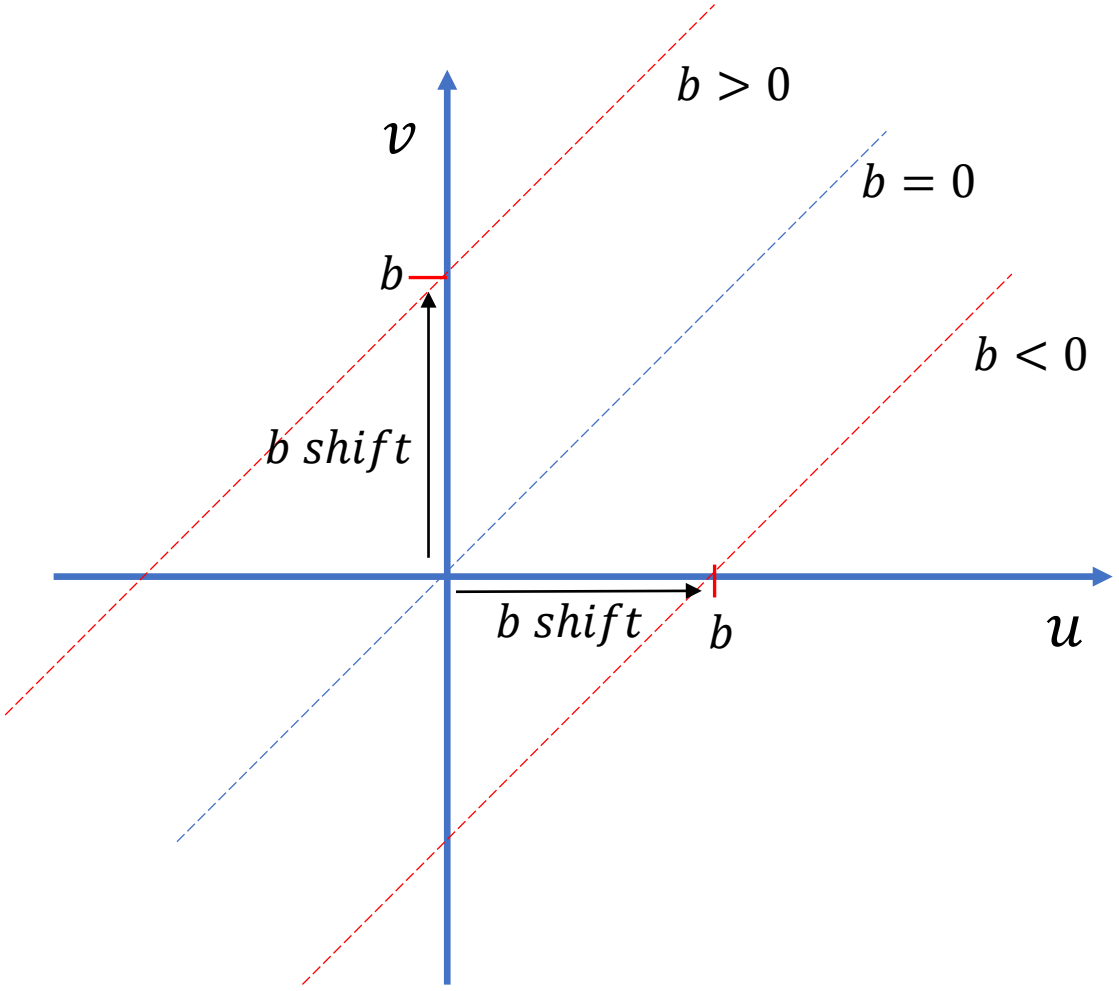
If a bias is NOT used, a threshold is used:

$$f(v) = \begin{cases} 1 & if\ v \geq \theta \\ -1 & if\ v < \theta \end{cases} \qquad v = \sum_{j=1}^{n} w_j x_j$$

- **It turns out, a threshold is equivalent to an adjustable bias…**

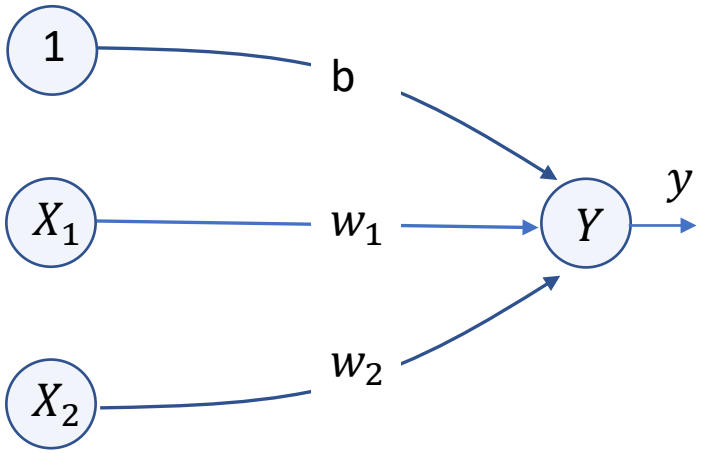
M&P: Biases and Thresholds

Recall, we wrote the input to the neuron as: $v = u + b$



Affine Transform (geometric transformation that preserves parallelism) produced by the presence of a bias

M&P: Bias



$$u = x_1 w_1 + x_2 w_2$$

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

The line in data space that separates the values of x_1 and x_2 for which the net with a bias gives a positive response and where it gives a negative response is:

$$w_1 x_1 + w_2 x_2 + b = 0$$

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{b}{w_2}$$