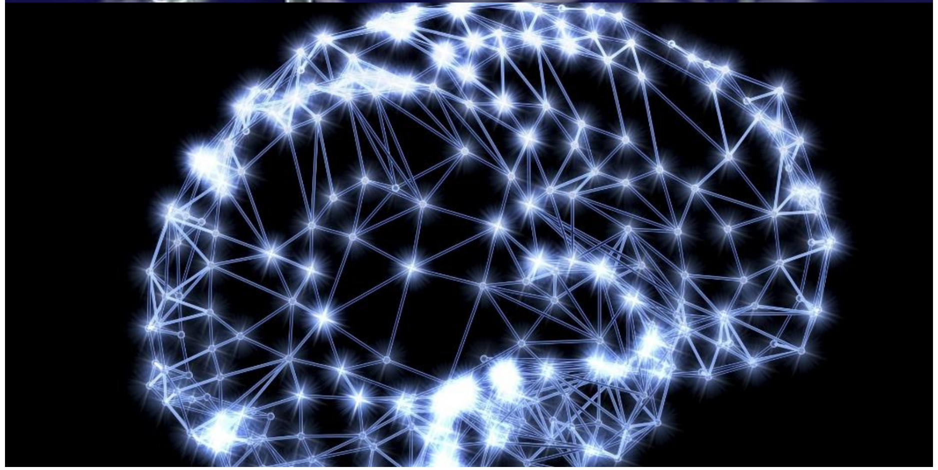
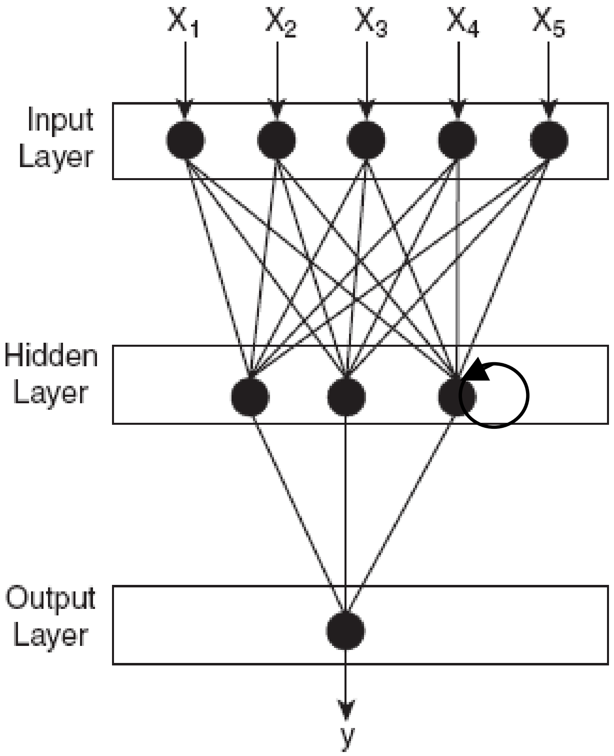
Artificial Neural Networks

Artificial Neural Networks (ANN)

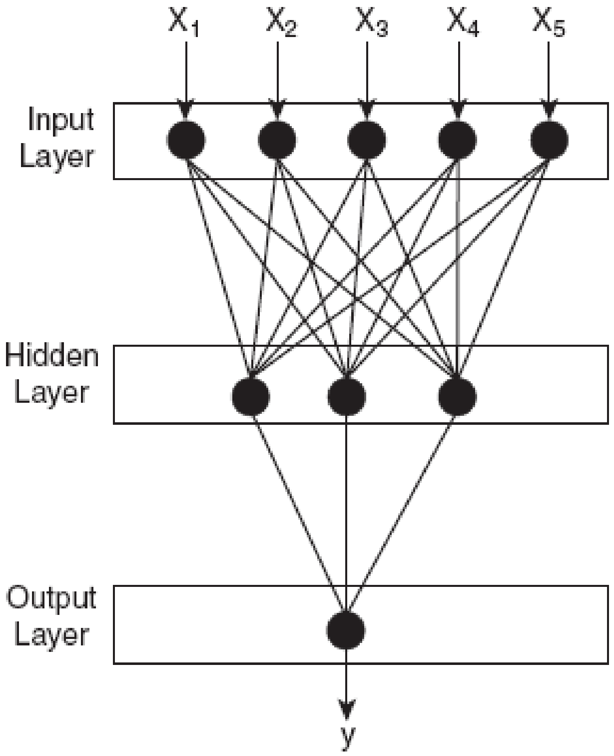
- ANNs inspired by biological neural systems.
- Human brain consists of nerve cells called neurons
- According to neurologists the human brain learns by changing the strength of the synaptic connection between neurons upon *repeated stimulation by the same impulse*.



General structure of an ANN



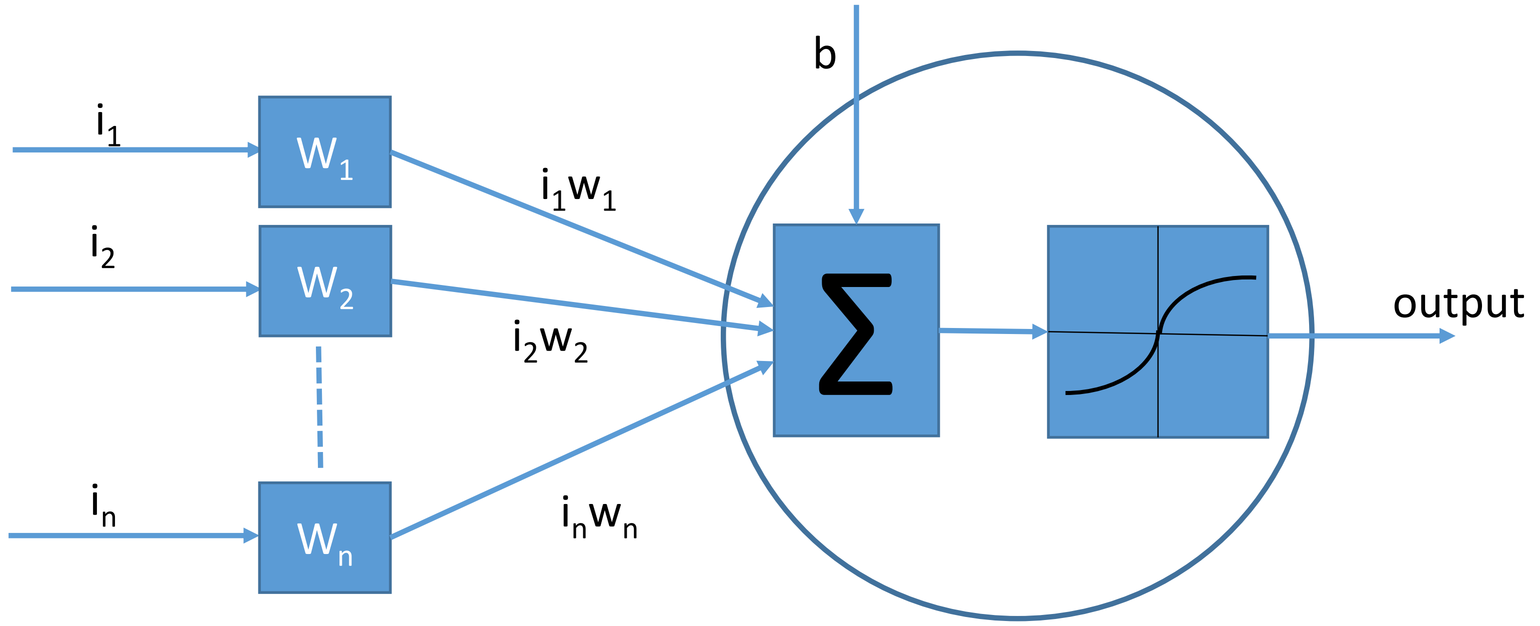
Recurrent NN



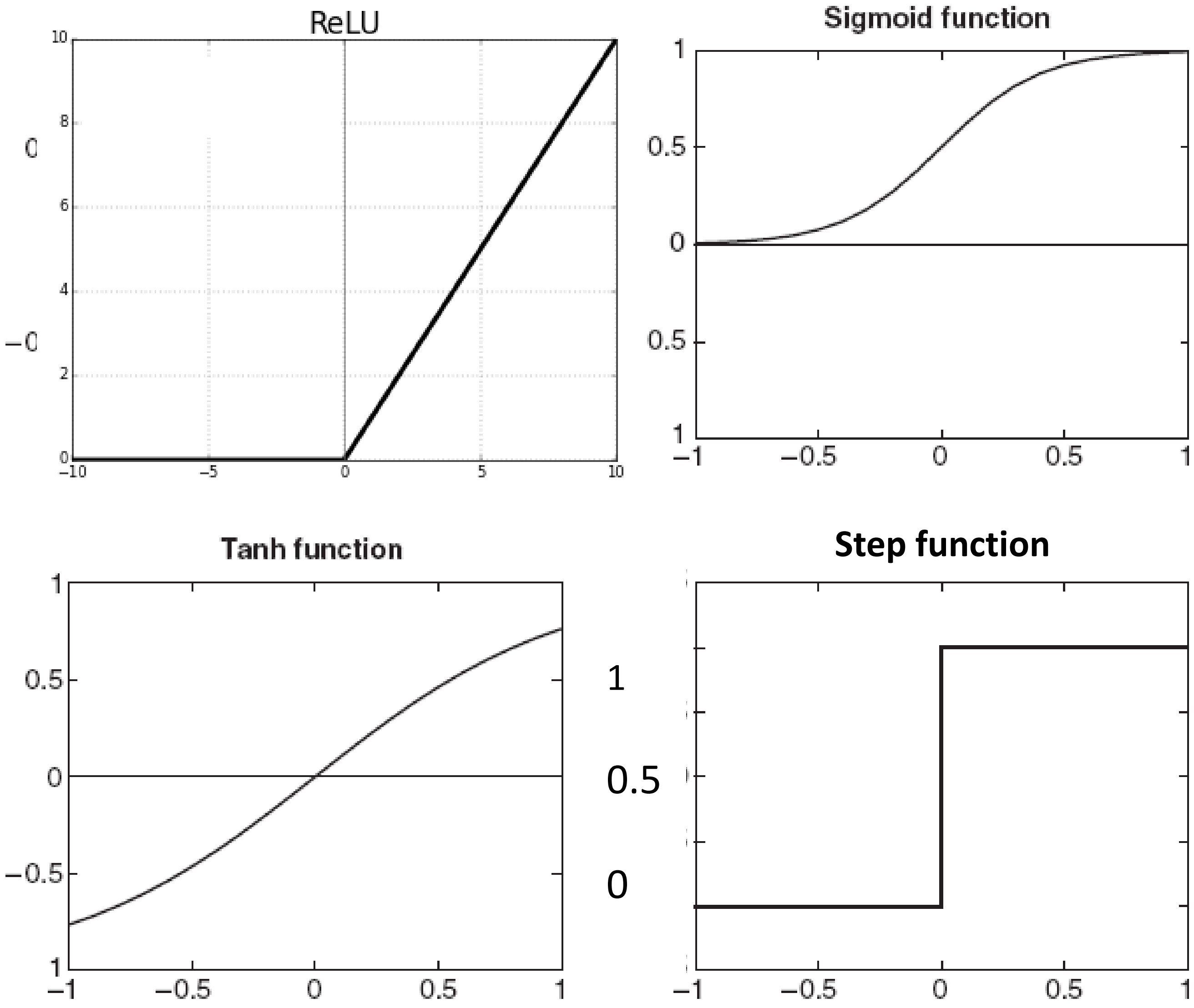
Feed Forward NN

A Look at a Processing Node

- Sums up weighted inputs into it, adds bias and uses activation function to compute result.



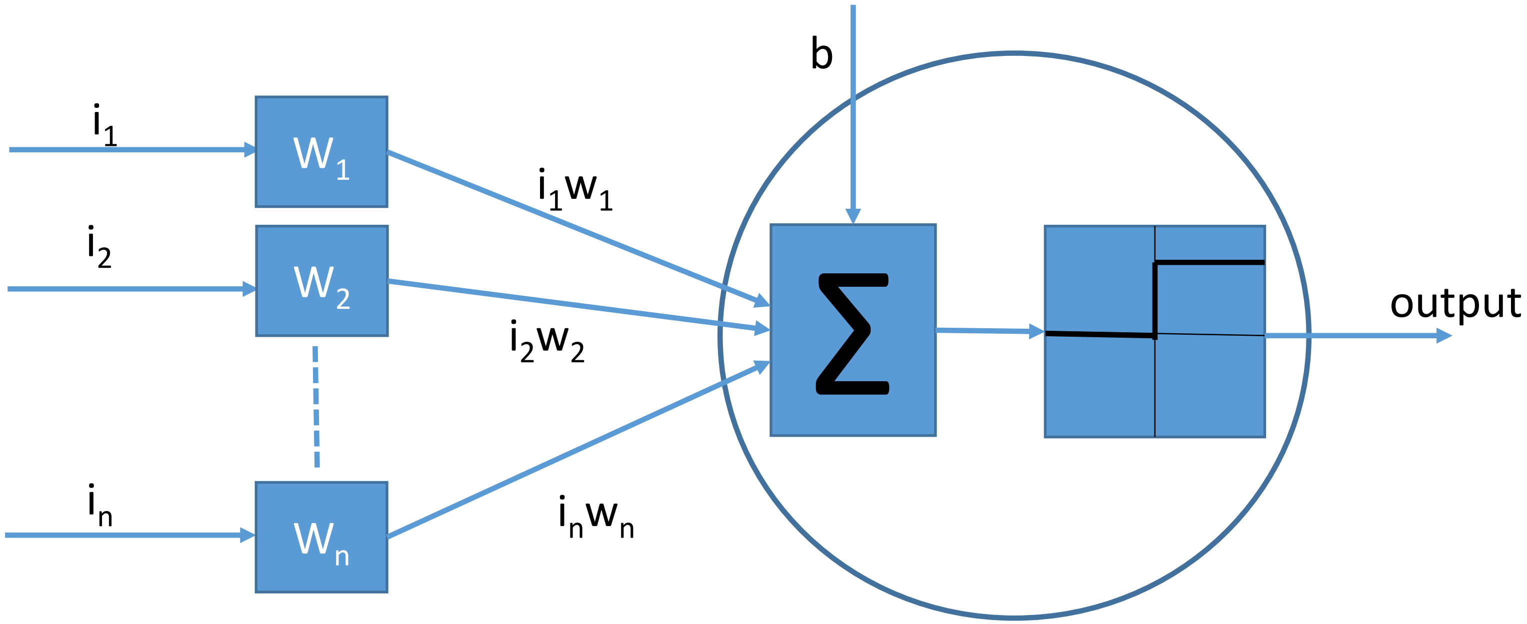
Activation Functions; Examples



Function	Expression
ReLU	R(z)=max(0,z)
Sigmoid	$\sigma(z)=\frac{1}{1+e^{-z}}$
Hyperbolic tan (tanh)	$\sigma(z)=\frac{e^{-z}-e^{z}}{e^{-z}+e^{z}}$
Gaussian/Radial Basis function	$\sigma(z)=e^{-\frac{z^2}{2}}$
Step function	$\sigma(z)=\begin{cases} 0, & z \leq \theta \\ 1, & z > \theta \end{cases}$

Simple Perceptron

- A one-layer feed forward network

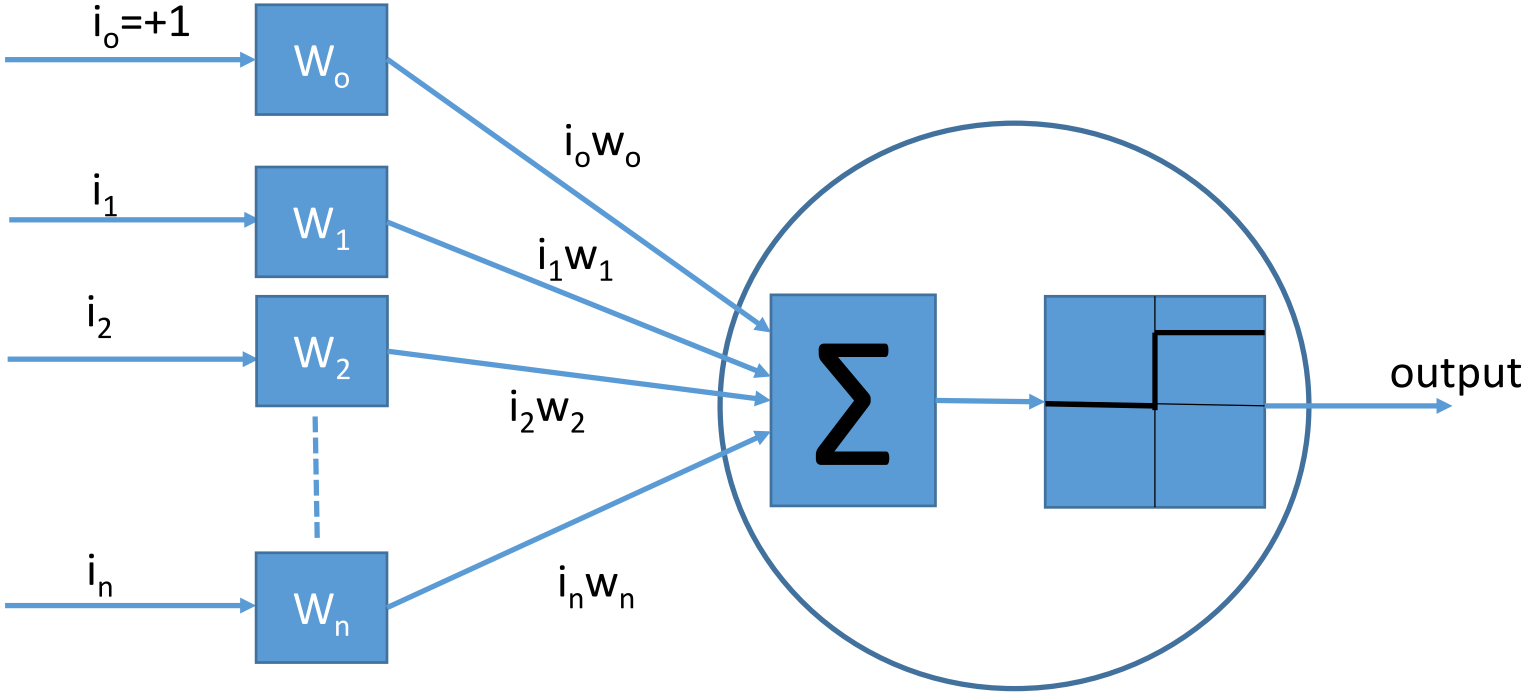


Input to activation function:

$$b + \sum_{i=1}^{n} w_i x_i$$

Simple Perceptron

- Often, bias term represented as a weight with unit input.



Input to activation function:

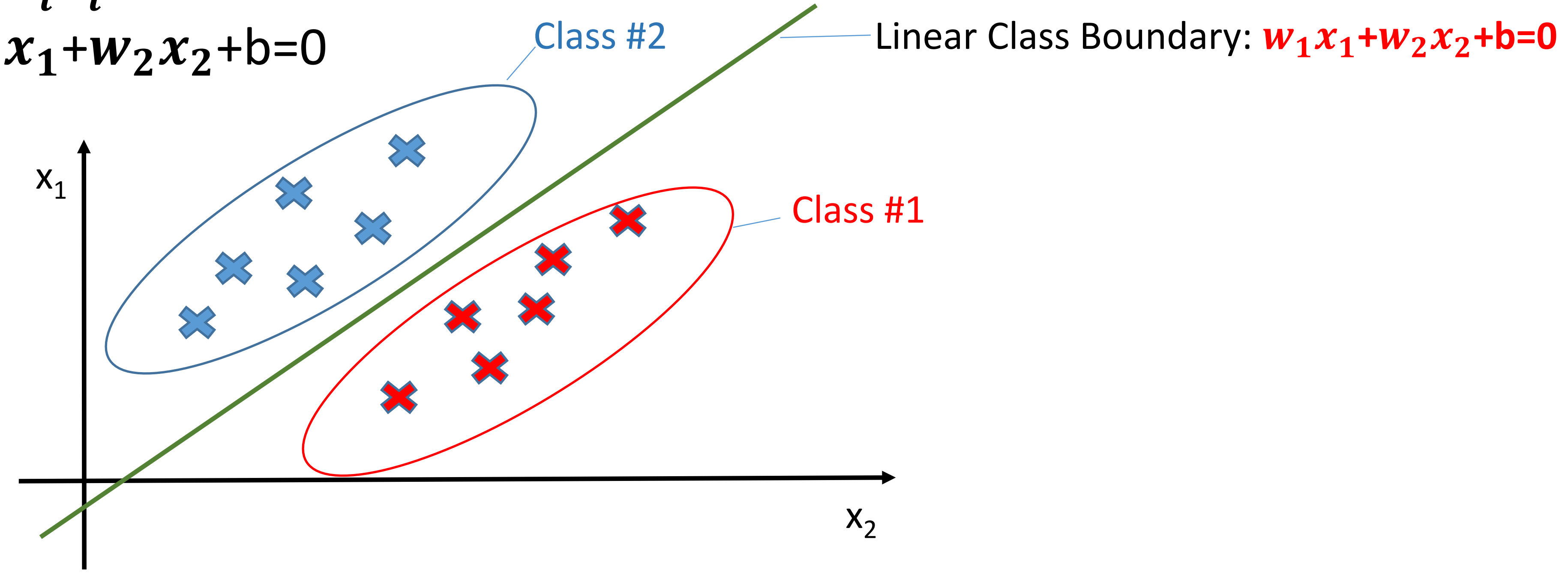
$$\sum_{i=0}^{n} w_i x_i$$

Simple Perceptron

- Used to separate linearly separable patterns.
  - E.g., In 2D, the boundary line is always a straight line…

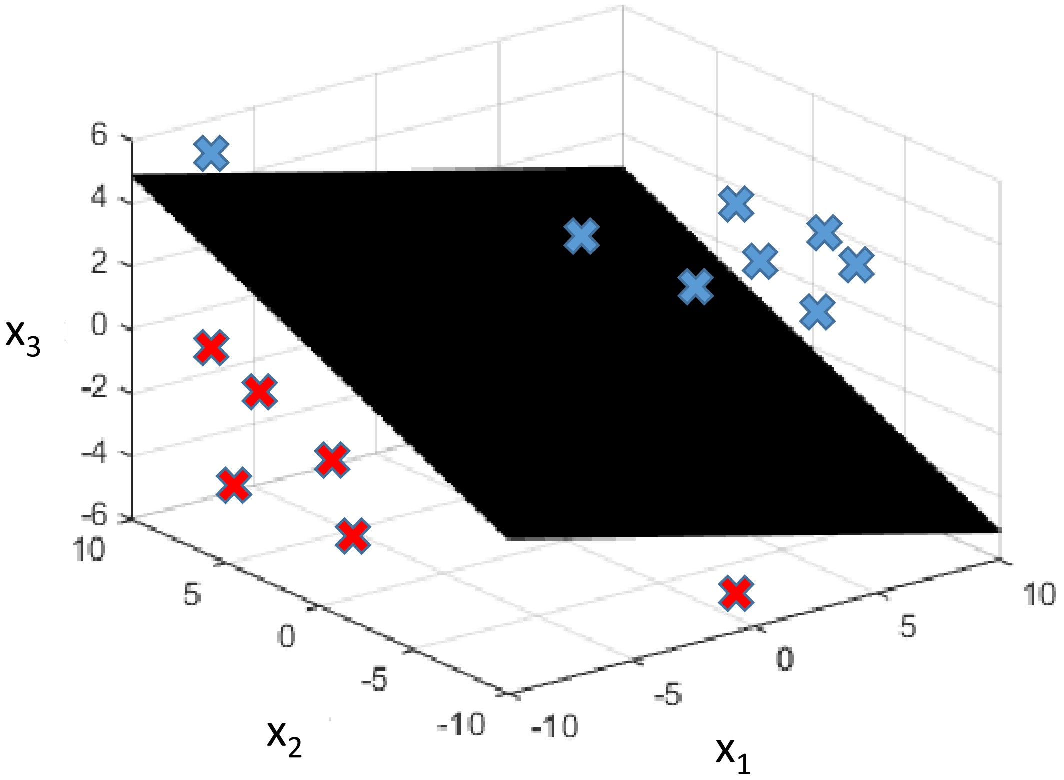
$\sum_{i=0}^{2} w_i x_i$=0

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



  - Weights determine the slope of the line. Bias determines the y-intercept.

Simple Perceptron



- In 3D (i.e., 3 features, x_1, x_2 and x_3), discrimination boundary is a 2D plane:

  $\boldsymbol{w_1 x_1} + \boldsymbol{w_2 x_2} + \boldsymbol{w_3 x_3} + b = 0$
- In general, when we have n features, our decision boundary is n-1 dimensional.
- A **hyperplane** is a subspace of one dimension less than its ambient space
- Learning process of a perceptron classifier can be said to amount to learning a hyperplane classifier

Perceptron Learning Algorithm

- Learning Process is basically to find the vector $\boldsymbol{w}$ of weights $\boldsymbol{w_o}$ through $\boldsymbol{w_n}$ that define a hyperplane separating the classes.
- Algorithm begins with some initial weights vector $\mathbf{w}^{i}$
- Algorithm cycles through the training set, a training sample at a time and makes decisions on adjustment of weights with each input sample.
- It's a mistake-driven algorithm
  - Only updates **w** when it makes a mistake; i.e., when it wrongly predicts the label of the current training example
  - Doesn't update **w** when it correctly predicts the label of the current training example

Perceptron Learning Algorithm

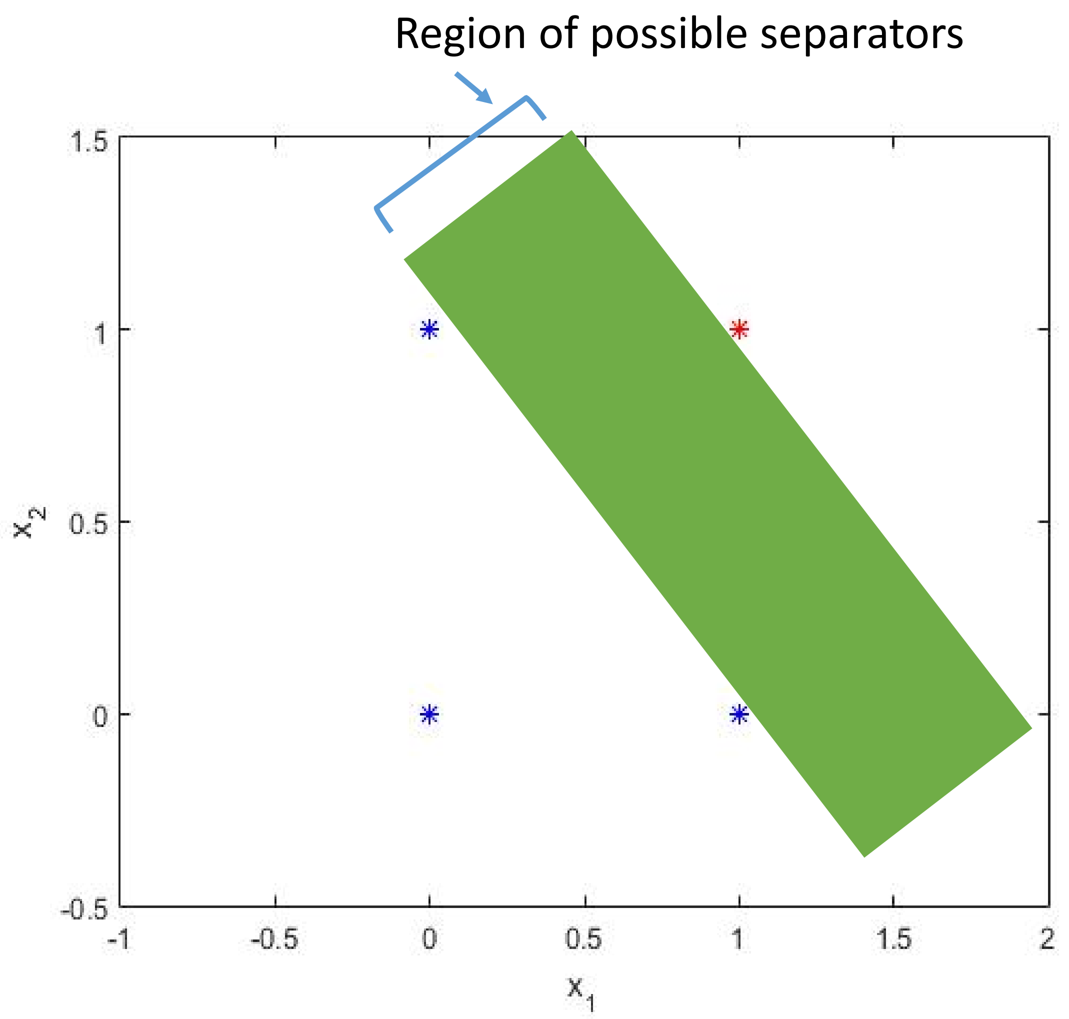
1. Initialize weight vector
2. Select random sample from training set as input
   - ✓Lets denote each sample as (x_{train}, y_{target}) where x is the feature vector and y_{target} is the class label of the sample.
3. Compute the output $y_{classifier}$
4. If $y_{classifier}=y_{target}$, do nothing.
5. Otherwise compute the error $\delta=y_{target} - y_{classifier}$ and the change in weight $\Delta w = \eta \times \delta \times x_{train}$ and update the weight vector using $w_{new}=w_{old}+ \Delta w$
6. Repeat this until the entire training set is classified correctly.

Perceptron Learning Example

- Perceptron that implements
a logical AND

Input #1	Input #2	Output
0	0	0
0	1	0
1	0	0
1	1	1

- Initial weight vector: [-2 3 1];
  - First component of vector is bias term
- Learning rate: 0.4
- Activation function: σ(z)=$\begin{cases} 0, & z \le 0 \\ 1, & z > 0 \end{cases}$



Perceptron Learning Example

- Notation: $[w_o\ w_1\ w_2]$= [-2 3 1] where w_o is weight of bias term; Features represented as $[x_o\ x_1\ x_2]$, where x_o is the bias term (recall this equals 1), and $x_1\ x_2$ are the input features for our training example. Denote, y_t=y_{target}; y_c=$y_{classifier}$
- Epoch #1

$w=[w_o\ \ w_1\ \ w_2]$			$x=[x_o\ x_1\ \ x_2\ ]$			y_t	Input, z to Activation function	$\sigma(z)=y_c$	$\delta= y_t\text{-}y_c$	$\Delta w=[\Delta w_o\ \ \Delta w_1\ \Delta w_2]$		
-2	3	1	1	0	0	0	**-2**	**0**	**0**	**0**	**0**	**0**
-2	**3**	**1**	1	0	1	0	**-1**	**0**	**0**	**0**	**0**	**0**
-2	**3**	**1**	1	1	0	0	**1**	**1**	**-1**	**-0.4**	**-0.4**	**0**
-2.4	**2.6**	**1**	1	1	1	1	**1.2**	**1**	**0**	**0**	**0**	**0**
-2.4	**2.6**	**1**										

Perceptron Learning Example

Epoch #2

$w=[w_o \ \ w_1 \ \ w_2]$			$x=[x_o \ x_1 \ \ x_2 \ ]$			y_t	z	$\sigma(z)=y_c$	$\delta= y_t-y_c$	$\Delta w=[\Delta w_o \ \ \Delta w_1 \ \Delta w_2]$		
-2.4	**2.6**	**1**	1	0	0	0	**-2.4**	**0**	**0**	**0**	**0**	**0**
-2.4	**2.6**	**1**	1	0	1	0	**-1.4**	**0**	**0**	**0**	**0**	**0**
-2.4	**2.6**	**1**	1	1	0	0	**0.2**	**1**	**-1**	**-0.4**	**-0.4**	**0**
-2.8	**2.2**	**1**	1	1	1	1	**0.4**	**1**	**0**	**0**	**0**	**0**
-2.8	**2.2**	**1**										

Perceptron Learning Example

Epoch #3

$w=[w_o \;\; w_1 \;\; w_2]$			$x=[x_o \; x_1 \;\; x_2 \,]$			y_t	z	$\sigma(z)=y_c$	$\delta= y_t\text{-}y_c$	$\Delta w=[\Delta w_o \;\; \Delta w_1 \; \Delta w_2]$		
-2.8	2.2	1	1	0	0	0	-2.8	0	0	0	0	0
-2.8	2.2	1	1	0	1	0	-1.8	0	0	0	0	0
-2.8	2.2	1	1	1	0	0	-0.6	0	0	0	0	0
-2.8	2.2	1	1	1	1	1	0.4	1	0	0	0	0
-2.8	2.2	1										

Final Weight Vector, w =[-2.8 2.2 1]

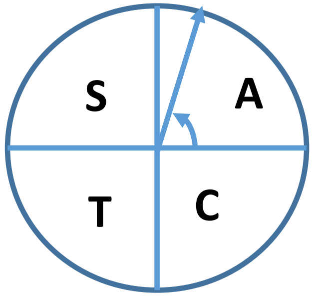
Perceptron Convergence Theorem

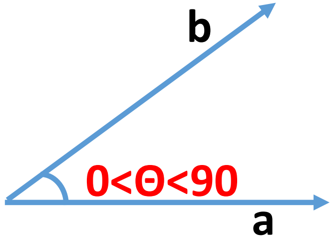
- Given data with linearly separable classes, the perceptron learning rule is guaranteed to find the separating hyperplane in a finite number of iterations
  - Requirement: Learning rate sufficiently small

Under the hood – Why the perceptron rule works

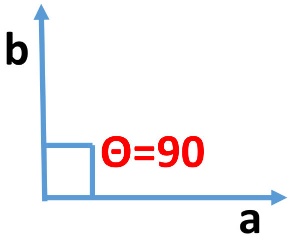
- Preliminaries:
  - Suppose two vectors ***a*** and ***b*** are separated by an angle **θ.** The inner product **a.b** is defined as **$a.b = ||a||\,||b|| \cos \theta$.**
    - L2 norm is always non-negative; -- it can either be positive or zero. This implies the sign of a.b can be inferred from the sign of $\cos \theta$. If $\cos \theta$ is negative, a.b will be negative; if $\cos \theta$ is positive, a.b will be positive.

All **S**tudents **T**ake **C**hemistry !!

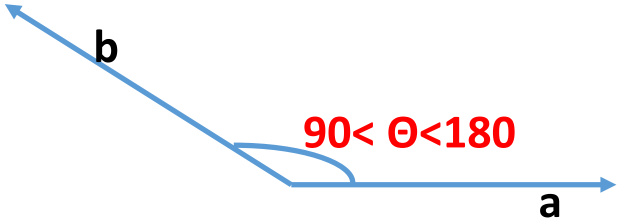




$\cos\Theta > 0 \Rightarrow a.b > 0$



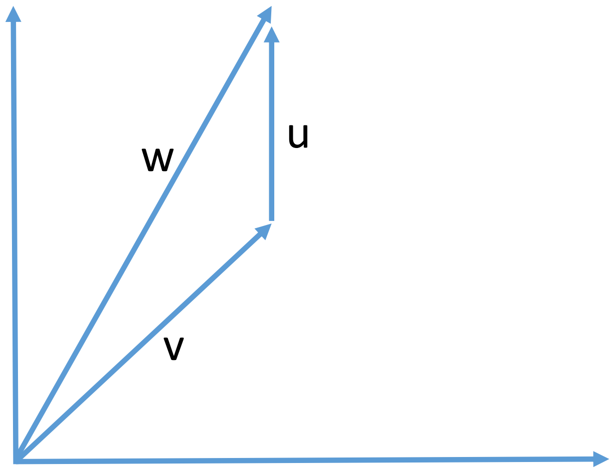
$\cos\Theta = 0 \Rightarrow a.b = 0$



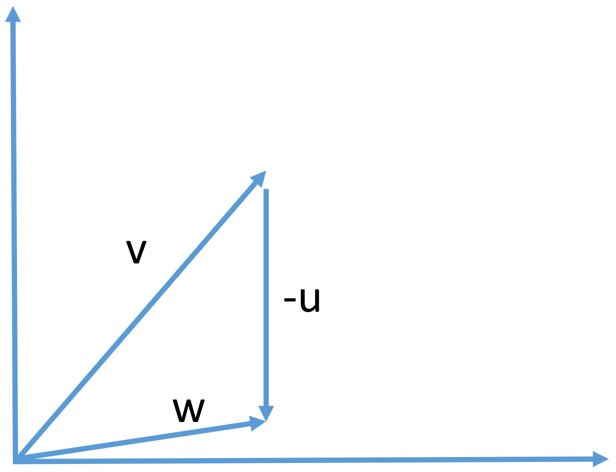
$\cos\Theta < 0 \Rightarrow a.b < 0$

Under the hood – Why the perceptron rule works

- Some more preliminaries …
- Consider two 2D vectors for simplicity. Geometrically, adding two vectors is equivalent to appending one vector to the end of the other.



w=v + u

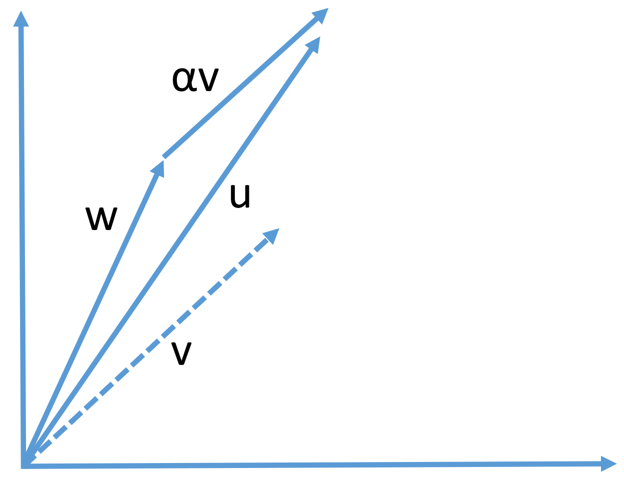


w=v + -u

Under the hood – Why the perceptron rule works

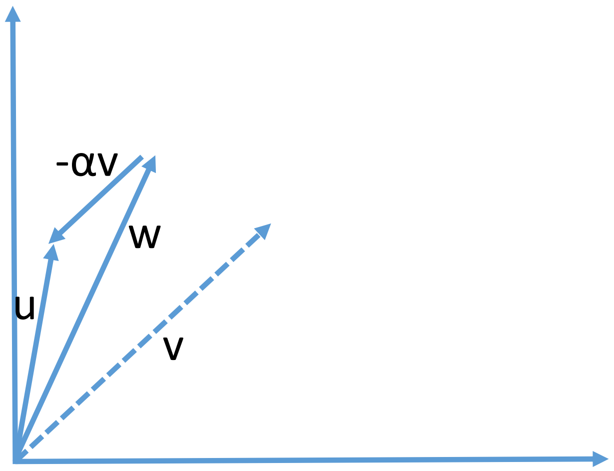
- Some more preliminaries …
- What if we want some guarantees on the result of the addition, e.g., if we want to add some vector to *w* and have the resultant vector (i) pointing more towards the direction of some vector *v*, (ii) pointing further away from the direction of some vector *v*

(i)



$u=w+\alpha v$

(ii)



$u=w+(-\alpha v)$

Under the hood – Why the perceptron rule works

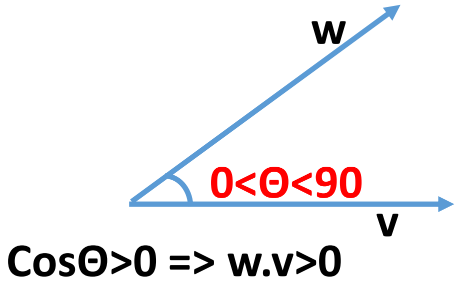
- Recall the input to the activation function was:

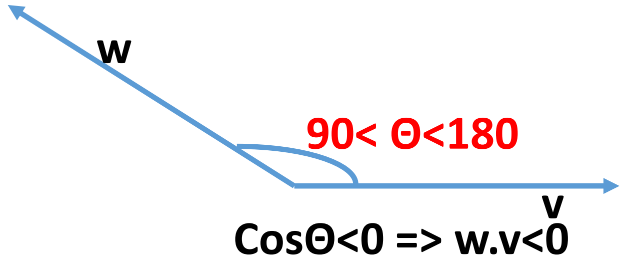
$$\sum_{i=0}^{n} \boldsymbol{w_i x_i}$$

  which for the 3D case, for example, would be equal to:
  $\boldsymbol{w_0 x_0 + w_1 x_1 + w_2 x_2 + w_3 x_3}$
  - But this is the dot product between two vectors w=[$\boldsymbol{w_0\ w_1 w_2\ w_3}$] and x=[$\boldsymbol{x_0\ \ x_1\ x_2\ x_3}$]
  - So, we basically have $\sum_{i=0}^{n} \boldsymbol{w_i x_i}$=$w.v$
  - **So, during the perceptron learning process, all that is happening is a computation of a dot product between the weight vector and each input vector**

Under the hood – Why the perceptron rule works

- Lets analyze the cases where the algorithm gave us a wrong result
  - Case 1: Learning algorithm assigned a sample to the class label of 1 (i.e., *w.v* was >0) yet it should have belonged to class label 0 (i.e., *w.v* should have been <0).
  - In this case, it means that we had the weight and input vectors aligned as the figure to the left, yet correct classification would have required them to be aligned as the figure to the right



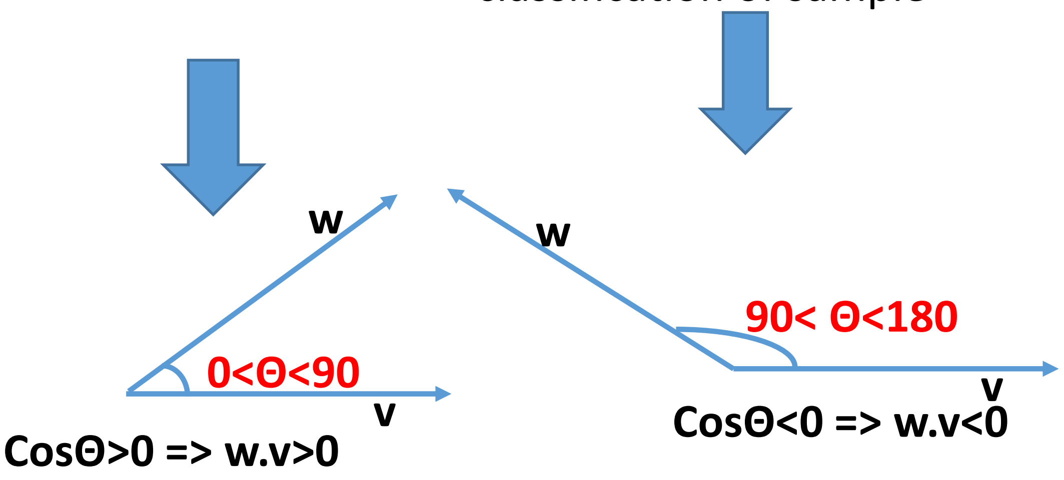


Under the hood – Why perceptron rule works

- Case 1 cont'd

What we have:

What we need for correct classification of sample



How do we fix Case 1?

✓ Rotate vector w away from vector v.

✓ That is, find a new value, w_{new} of w, such that: $w_{new}=w+(-\alpha v)$

Check: How did the perceptron algorithm fix Case 1?

✓ In our Case 1, $y_{classifier}=1$ (because the classifier finds that w.v>0). Since we know that the classifier fails to get the correct decision, this means $y_{target}=0$.
Recall that the error is $\delta=y_{target} - y_{classifier}$
So, that particular iteration would have $\delta= -1$.

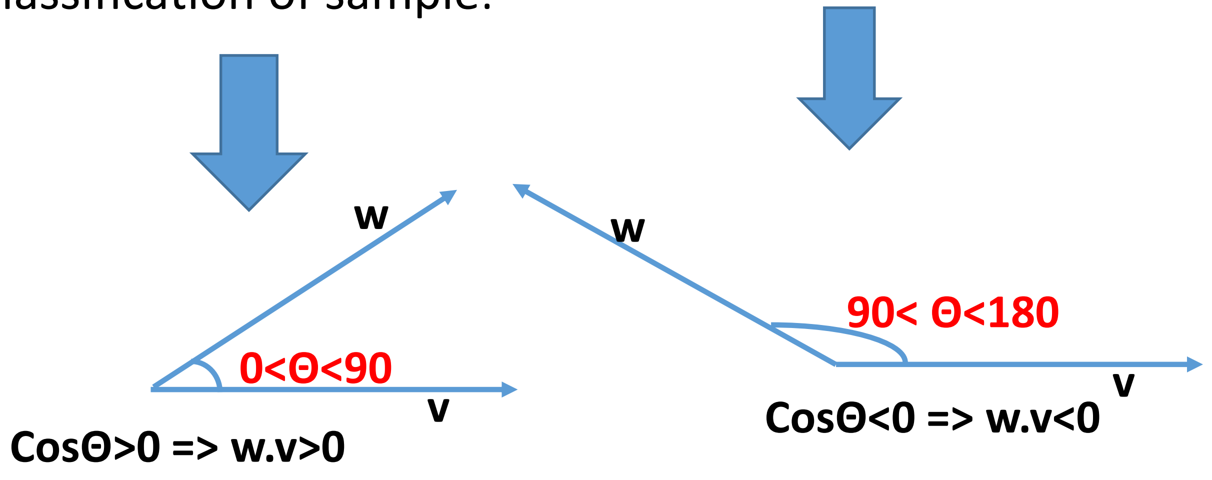
✓ And the weight update would be $\Delta w = \eta \times \delta \times x_{train}$ which is the same as $\Delta w = (-1)*\eta \times x_{train}$
The new weight is hence $w_{new}=w+(-\eta \times x_{train})$ which is in agreement with our strategy above for fixing Case1, since η is a positive constant between 0 and 1 and our v is x_{train}

Under the hood – Why perceptron rule works

- Lets analyze the cases where the algorithm gave us a wrong result
  - Case 2: Learning algorithm assigned a sample to the class label of 0 (i.e., *w.v* was <0) yet it should have belonged to class label 1 (i.e., *w.v* should have been >0).

What we need for correct classification of sample:

What we have:



This time we want to align w more towards the direction of v so as to work towards attaining the condition **w.v>0**

Using an idea similar to what we used in the previous case, we need to get $w_{new}=w+(\alpha v)$.
Easy to check that the perceptron algorithm did just this !

Simple Perceptron: some issues

- On test data, the perceptron will use the final weight vector obtained during training.
- But this may not necessarily be a good idea ! One of the “wrong” weight vectors produced before the final “correct” vector could actually perform better on test data
  - Averaged perceptron (average some of the well performing weights and predict)
  - Voted perceptron (vote on predictions of intermediate vectors)

Simple Perceptron: some issues

- Perceptron rule wont converge if classes not linearly separable.
- E.g., XOR problem (OR without AND)

x1	x2	y
0	0	0
1	0	1
0	1	1
1	1	0