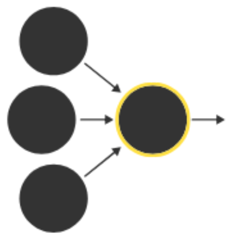
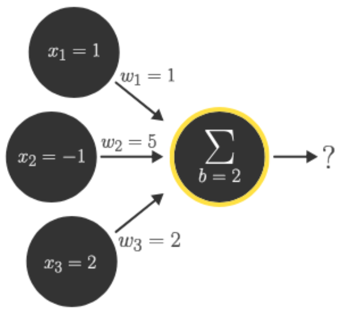


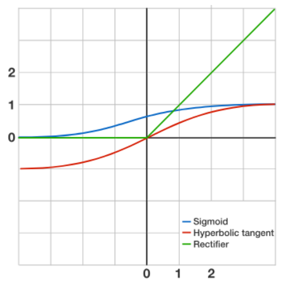
A neuron has many inputs but only one output, so it must "integrate" its inputs into one output (a single number). Recall that the inputs to a neuron are generally outputs from other neurons. What is the most natural way to represent the set of these inputs to a single neuron in an ANN?
(number, *vector, matrix)

In our computational model of a neuron, the inputs defined by the vector $\vec{x}$ are "integrated" by taking the **bias** b plus the dot product of the **inputs** $\vec{x}$ and **weights** $\vec{w}$
$\vec{w} \cdot \vec{x} + b$
The dot product represents a "weighted sum" because it multiplies each input by a weight.
A biological interpretation is that the inputs defining $\vec{x}$ are the outputs of other neurons, the weights defining $\vec{w}$ are the strengths of the connections to those neurons, and the bias b impacts the threshold the computing neuron must surpass in order to fire.

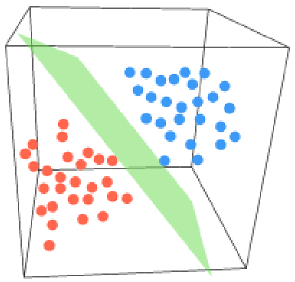


Given the inputs, weights, and bias shown above, what is the integration of these inputs, given by the weighted sum?

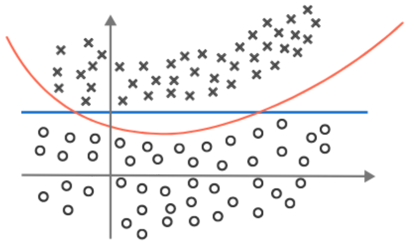
When H(v) is the Heaviside step function, the neuron modeled by $H(\vec{w}\cdot\vec{x}+b)$ fires when $\vec{w}\cdot\vec{x}+b \geq 0$. The hypersurface $\vec{w}\cdot\vec{x}+b=0$ is called the **decision boundary**, since it divides the input vector space into two parts based on whether the input would cause the neuron to fire. This model is known as a **linear classifier** because this boundary is based on a linear combination of the inputs.



Other activation functions: sigmoid, tanh, rectifier (fast to compute!). The power of ANNs is illustrated by the universal approximation theorem, which states that ANNs using activation functions like these can model *any* continuous function. No matter how complicated a situation is, a sufficiently large ANN with the appropriate parameters can model it.
-> ANNs are just function approximators, no magic, just math (like a linear regression)! No consciousness or sth like that!



Data points are classified according to the side of the hyperplane (or decision boundary, in green above) on which they lie.



The figure shows a set of training data in two categories, with two possible decision boundaries forming a binary classification model of this data. Which decision boundary corresponds to a possible model under the perceptron algorithm?

some functions cannot be perfectly modeled by the perceptron: when the two classifications of the function are not linearly separable (that is, it cannot be partitioned by a hyperplane decision boundary). -> XOR
Like in the brain, one neuron is not very powerful. The real strength is revealed when they are combined to form a network.

We've talked about how the perceptron works given some weights and a bias (that define a decision boundary), but now we need to determine how to find an appropriate decision boundary given some data. We'll need to define a loss function that measures the extent of the errors from a given setting of weights $\vec{w}$ and bias b. Some intuitive possibilities include the total number of misclassified points and the sum of the distances from misclassified points to the decision boundary.
Loss function: $E(w) = \frac{1}{2}$ SUM $(y - y\_hat)^2$ $y\_hat = \vec{w} \cdot \vec{x} + b$

Now that we have a way of evaluating our perceptron model, how do we determine appropriate weight parameters? We'll consider a mistake driven algorithm. -> start with a simple classifier (e.g., where all weights and the bias are 0), and iteratively adjust the weights and biases based on each incorrectly classified x_i to minimize E.
Example with Perceptron with one input:

x	y
1	1
2	2
3	3

- $w = 1,\ E(\mathbf{w}) = 0$

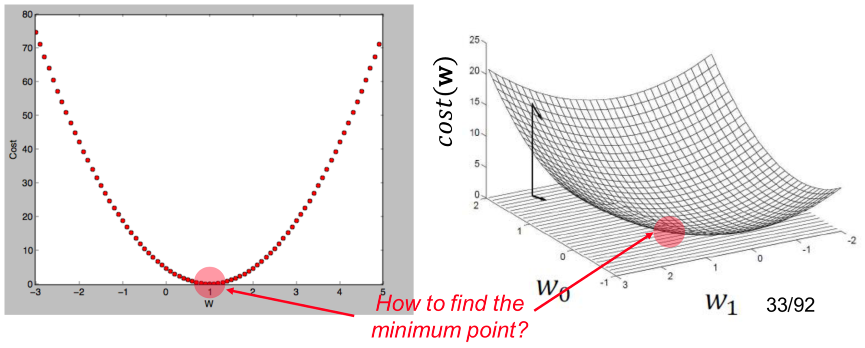
  $\frac{1}{2}\left((1 - 1 \times 1)^2 + (2 - 1 \times 2)^2 + (3 - 1 \times 3)^2\right)$

- $w = 0,\ E(\mathbf{w}) = 4.5$

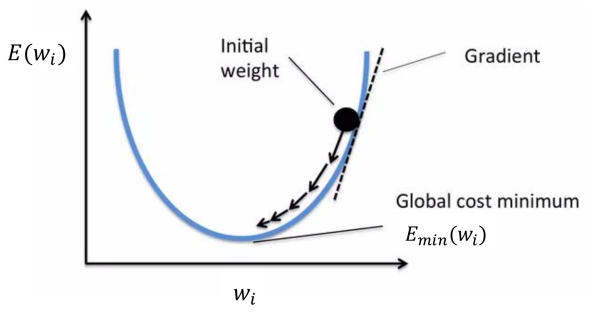
  $\frac{1}{2}\left((1 - 0 \times 1)^2 + (2 - 0 \times 2)^2 + (3 - 0 \times 3)^2\right)$

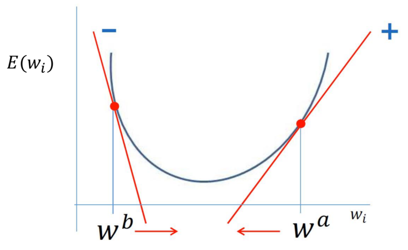
- $w = 2,\ E(\mathbf{w}) = 4.5$

  $\frac{1}{2}\left((1 - 2 \times 1)^2 + (2 - 2 \times 2)^2 + (3 - 2 \times 3)^2\right)$



1. Start with initial guesses
   - Start at random value
2. Each weight is updated by taking a step into the opposite direction of the gradient $\Delta w_i = -\eta \times \frac{\partial E}{\partial w_i}$
   - Compute the partial derivative of the cost function $\frac{\partial E}{\partial w_i}$ for each weight
3. Repeat until you converge to a local minimum





$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

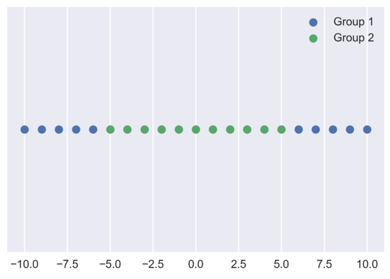
η: learning rate (*e.g.* 0.001)

$\frac{\partial E}{\partial w_i}$: gradient

Getting the gradient by differentiating the loss function:
½ (wx - y)^2
chain rule: g(f(x))' = g'(f(x)) * f'(x)
g(a) = ½ a^2 -> g' = a (a = wx - y)
f(x) = wx - y -> f' = x

E'(x) = a*x = (wx - y) * x
IMPLEMENT! -> perceptron1.py
this was regression, now classification

limits of the Perceptron:
make inseparable groups


● Group 1
● Group 2