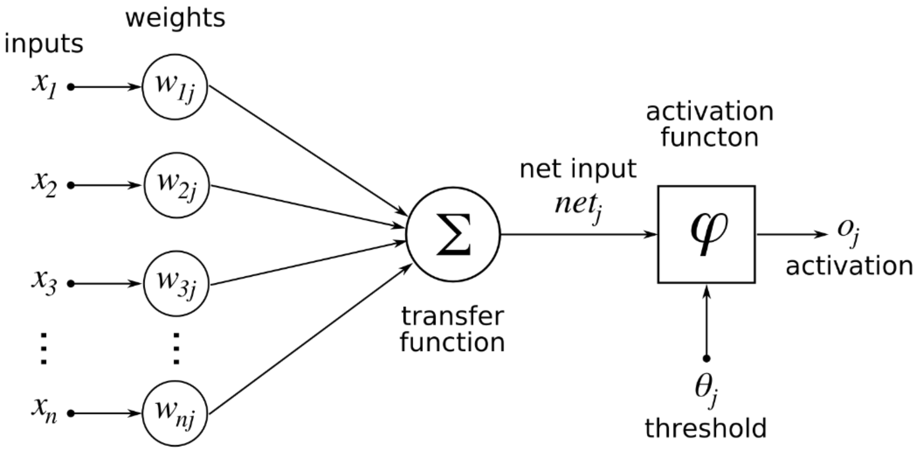
SL03: Neural Networks

Perceptron:

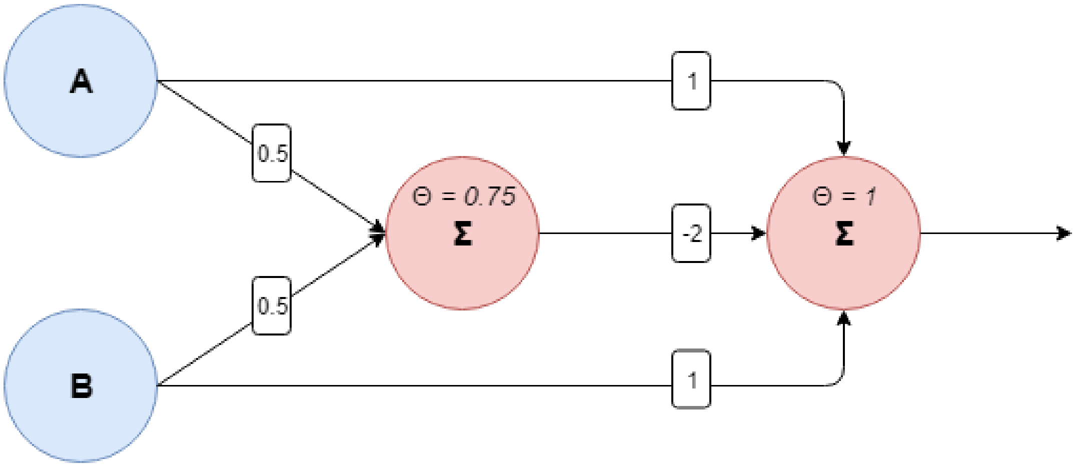
- A perceptron is a simple computational imitation to a brain cell. It allows computations to fire under certain stimuli, allowing for learning.



- The Perceptron calculates the sum of products of the inputs X and their corresponding weights W, and then compare the result with an activation threshold. If the sum is greater than or equal the threshold θ, the Perceptron fires (outputs one).

$$\begin{cases} \sum_{i=1}^{k} X_i W_i \ \geq \theta & y = 1 \\ otherwise & y = 0 \end{cases}$$

- The Perceptron can be used to model Boolean functions:
  - AND: $w_1 = \frac{1}{2}, w_1 = \frac{1}{2}, \theta = \frac{3}{4}$
  - OR: $w_1 = 1, w_1 = 1, \theta = 1$
  - XOR: Requires 2 Perceptrons.

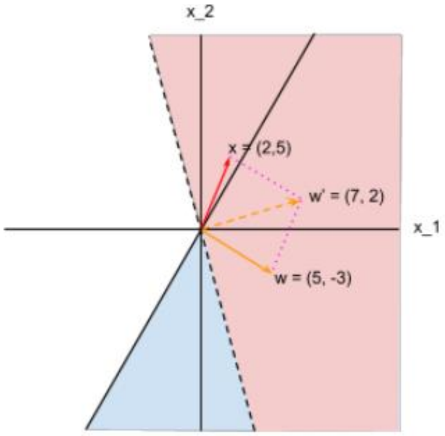


- Perceptrons are always going to compute lines (or hyperplanes in n dimensions). There's a dividing line where we're equal to the threshold.
- Perceptron training:
  There're two ways to learn the weights of a Perceptron:
  - Perceptron Rule (Thresholded):
    → The Perceptron Learning Rule defines how to update the weights in an iterative way.
    → We add one more weight to the equation to correspond to $-\theta$ to simplify the math. The input for this weight is always 1, and it's called the bias unit.
    → It works by "rewarding" the correct weights and "punishing" the incorrect ones. We might think of this as "rotating" the hyperplane to put the training data on the correct side of the boundary.

$$w_i = w_i + \Delta w_i$$

    → One possibility for this would be:

$$\Delta w_i = x_i$$
$$w_i = w_i + x_i$$



    → This rotates the hyperplane towards the point, which will produce the right classification if the point was originally positive and was misclassified as negative. But there're two problems with this rule:
    1. If it was the other way around (negative misclassified as positive) then we should instead rotate the hyperplane away from the point.
    2. The rule doesn't capture the situation where we classify the data correctly, it will update the weights anyway.

    → We need to connect the weights update to the classification correctness. We can do this by using these observations:

$$\hat{y} = (\sum_i w_i x_i \geq 0)$$
$$y - \hat{y} = 0 \quad if\ correct\ classification$$
$$y - \hat{y} = 1 \quad if\ we\ misclassify\ possitive\ as\ negative$$
$$y - \hat{y} = -1 \quad if\ we\ misclassify\ negative\ as\ positive$$

- → Then we can update the weights using this rule:

$$\Delta w_i = (y - \hat{y})x_i$$
$$w_i = w_i + (y - \hat{y})x_i$$

- → The Perceptron Learning Rule works by adjusting the weights based on the neuron's performance.
- → In the best case, the difference between the predicted value $\hat{y}$ and the training example y is zero, which means that the perceptron was able to correctly predict the value of y, so the weight will not be updated. It keeps repeating this process till the error saturates at 0.
- → Otherwise, the weights will be adjusted (positively or negatively) to correct the error in prediction.
- → We'll also update this formula by adding the learning rate η. Which controls the learning process, so that the weights get adjusted by a small value each time to avoid overshooting.

$$\Delta w_i = \eta(y - \hat{y})x_i$$
$$w_i = w_i + \eta(y - \hat{y})x_i$$

- → The Perceptron Learning Rule works only on linearly separable datasets, where we can separate the positive and negative examples using a straight line (or a hyperplane).
- → If the data is linearly separable, the Perceptron will eventually find the suitable line to separate the examples.
- → The problem is, there's no obvious way to determine if the data is linearly separable or not, especially with higher dimensions.
- → One workaround it to run the algorithm on the data and see if it ever stops.

- Gradient Descent (Unthresholded):
  - → Gradient Descent is an algorithm that can work with data that isn't linearly separable.
  - → In one variable, derivative gives us the slope of the tangent line. In several variables, Gradient (the generalization of derivatives in several variables) points towards direction of the fastest increase of the function.
  - → Let D be the training set with ordered pairs (x, y), where y is the target value. Gradient Descent works by taking the dot product a of w and x (which is essentially a simple linear regression), and then minimize the error of this activation using calculus.

$$a = \sum_i x_i w_i$$
$$\hat{y} = \{a \geq 0\}$$
$$E(w) = \frac{1}{2} \sum_{(x,y) \in D} (y - a)^2$$

- → To minimize error $E(w)$, we calculate the partial derivative of $E(w)$ with respect to each of the individual weights (The $\frac{1}{2}$ in the equation was added to simplify the derivative):

$$\frac{dE}{dw_i} = \frac{d}{dw_i}\frac{1}{2}\sum_{(x,y)\in D}(y-a)^2$$

$$= \frac{1}{2} \times 2 \sum_{(x,y)\in D}(y-a)\frac{d}{dw_i} - \sum_{i} x_i w_i$$

$$= \sum_{(x,y)\in D}(y-a)\,(-x_i)$$

- → The error decreases fastest in the direction opposite to the gradient.
- → Robust to data that is not linearly separable.
- → Might converge to a local optima though.

- Perceptron Rule vs Gradient Descent:
  - → Perceptron Learning $\Delta w_i = \eta(y - \hat{y})x_i$
  - → Gradient Descent $\Delta w_i = \eta(y - a)x_i$
  - → The reason we can't use $\hat{y}$ in Gradient Descent is that it will make the formula non-differentiable.

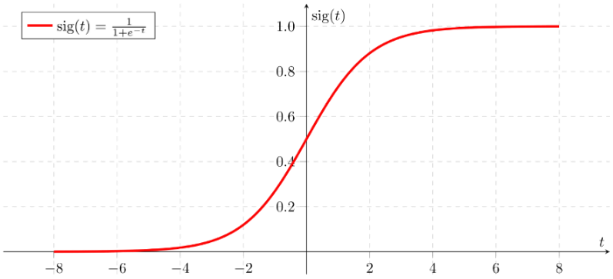
- Sigmoid Function:
  - Motivation: To use the gradient descent above, we had to use a linear unit, which is just a linear function. So we end up with a problem, we want the nonlinearity of the thresholded perceptron, with the robust of the training rule given by gradient descent.
  - The solution would be to use the Sigmoid Function, which is a smoothed function that facilitates a differentiable threshold.

$$\sigma(a) = \frac{1}{1 + e^{-a}}$$

$$a \to -\infty \qquad \sigma(a) \to 0$$

$$a \to +\infty \qquad \sigma(a) \to 1$$

$$\frac{d\sigma(a)}{da} = \sigma(a)(1 - \sigma(a))$$



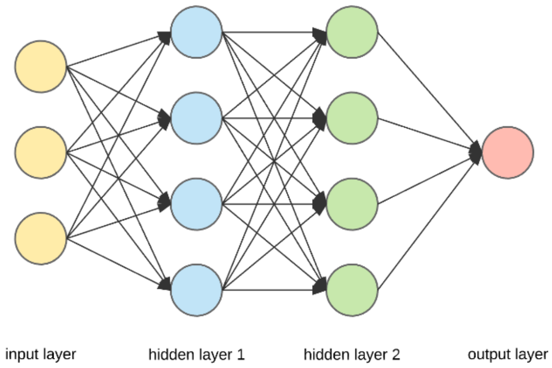
  - We can use perceptrons that operate according to the sigmoid rule:

$$\hat{y} = \sigma(x.w)$$

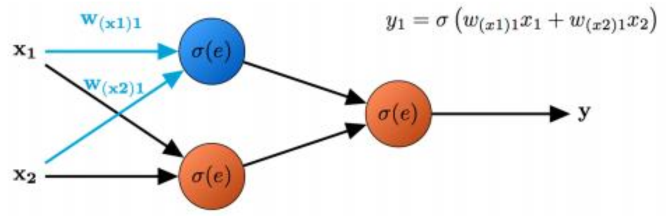
  - And the training rule would be:

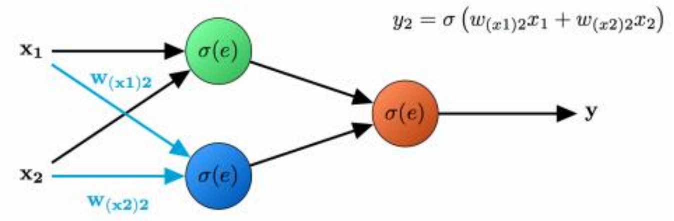
$$\Delta w_i = \eta \sum_{(x,y)\in D}(y - \hat{y}).\hat{y}.(1-\hat{y})x_i$$

Neural Networks:

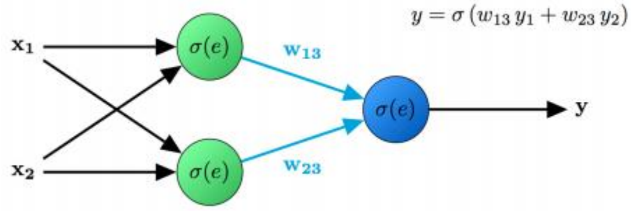


- A Neural Network is a chain of relations between the input and the output, with hidden layers in between.
- Each hidden layer computes the weighted sum (sigmoid) of the layer before it.
- If sigmoid is used in the nodes of the hidden layers, then the mapping from input to output will be differentiable.
  - This means that we can determine how each weight in the network affects the mapping from input to output.
  - This facilitates Backpropagation, which means performing a backward distribution of the error values (computed at the output) back to the network layers to optimize the weights.
- Note that using sigmoid in the Neural Network doesn't always guarantee convergence. Sigmoid (or any other differentiable function we might use in the hidden layers) is not a hard threshold, like the Perceptron. In a multilayer setting, we might end up in a local optima.
- How Backpropagation works?
  1. The neural network is populated with random small weights.
  2. The output of each node will be used to calculate the output of the subsequent node in the next layer.

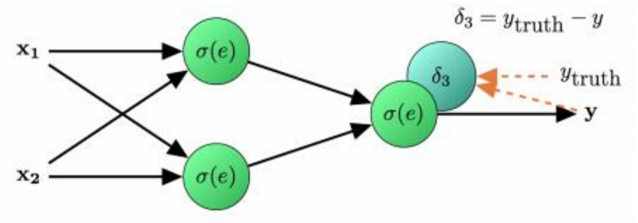


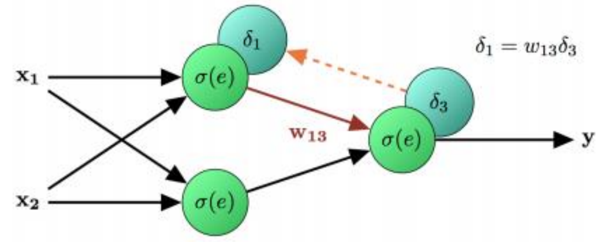


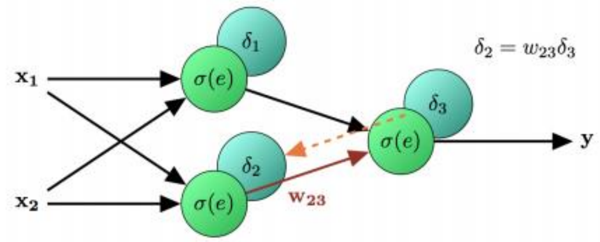
  3. The weights calculated through the network will be used to calculate the final output.



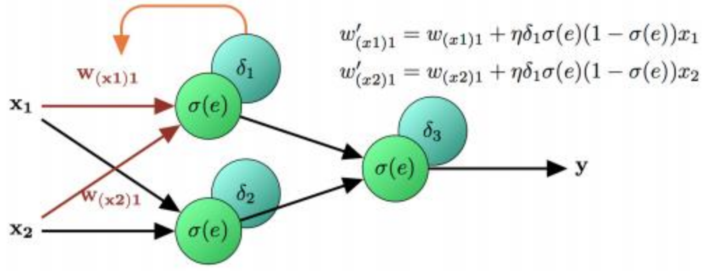
  4. The error between the true value and the approximated value will be calculated.
  5. Using the error calculated at the final node, we can "backpropagate" through the network to calculate the error at each node.

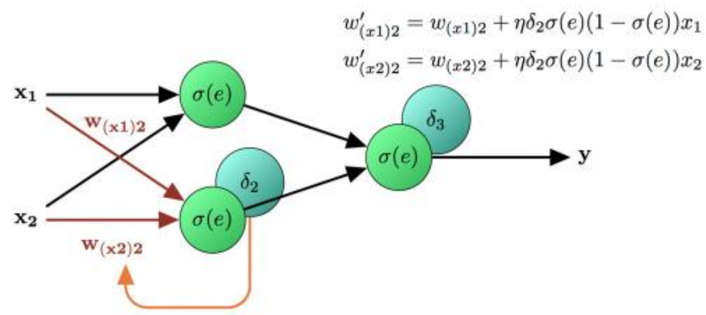


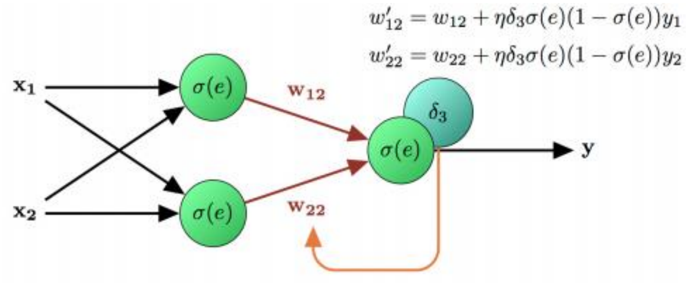




6. Then we adjust the weights using the calculated errors.


$w'_{(x1)1} = w_{(x1)1} + \eta\delta_1\sigma(e)(1-\sigma(e))x_1$
$w'_{(x2)1} = w_{(x2)1} + \eta\delta_1\sigma(e)(1-\sigma(e))x_2$
$w_{(x1)1}$
$w_{(x2)1}$
x_1
x_2
δ_1
$\sigma(e)$
δ_2
$\sigma(e)$
δ_3
$\sigma(e)$
y


$w'_{(x1)2} = w_{(x1)2} + \eta\delta_2\sigma(e)(1-\sigma(e))x_1$
$w'_{(x2)2} = w_{(x2)2} + \eta\delta_2\sigma(e)(1-\sigma(e))x_2$
x_1
x_2
$w_{(x1)2}$
$w_{(x2)2}$
$\sigma(e)$
δ_2
$\sigma(e)$
δ_3
$\sigma(e)$
y


$w'_{12} = w_{12} + \eta\delta_3\sigma(e)(1-\sigma(e))y_1$
$w'_{22} = w_{22} + \eta\delta_3\sigma(e)(1-\sigma(e))y_2$
x_1
x_2
$\sigma(e)$
$\sigma(e)$
w_{12}
w_{22}
δ_3
$\sigma(e)$
y

Optimizing Weights:

- There’re many optimization methods to find a global minima:
  - Momentum: An extra factor to push past the local minima.
  - Higher order derivatives: Use a combination of weights.
  - Randomized optimization.
  - Penalizing complexity: Penalizing more nodes, more layers or larger weights.

Restriction Bias:

- The Restriction Bias determines what is the set of hypotheses we will consider, which in turn highlights the representation power of the learning algorithm.
- As we discussed, Perceptions can only work with linear data (half spaces). While sigmoid allow for representing more complex data, and using hidden layer increase the representation power even more.
- All these components of Neural Networks allow for modeling many types of functions:
  - Boolean: Network of threshold-like units.
  - Continuous: A single hidden layer with enough hidden nodes.
  - Arbitrary: Multiple hidden layers.
- We can conclude that Neural Networks have a low Restriction Bias because they have the ability to model a wide variety of functions. This, however, increase the possibility of overfitting.
  - To avoid overfitting, when designing neural networks, we restrict ourselves to a bounded number of hidden layers with a bounded number of units. We can use cross validation to decide these numbers.
- A Neural Network will not only overfit because of excessive complexity but can also overfit because of excessive training (higher magnitude of the weights). We can use cross validation to decide when to stop training.

Preference Bias:

- The Preference Bias determines which representation is preferred.
- One preference in Neural Networks is to choose the initial weights to be small random values:
    - Random values provide variability that helps in avoiding local minima.
    - Small values have low complexity, which is preferred because larger weights can lead to overfitting (satisfies the Restriction Bias).
- In summary, Neural Networks prefer simpler and generalizable representations (In accordance with Occam's Razor).