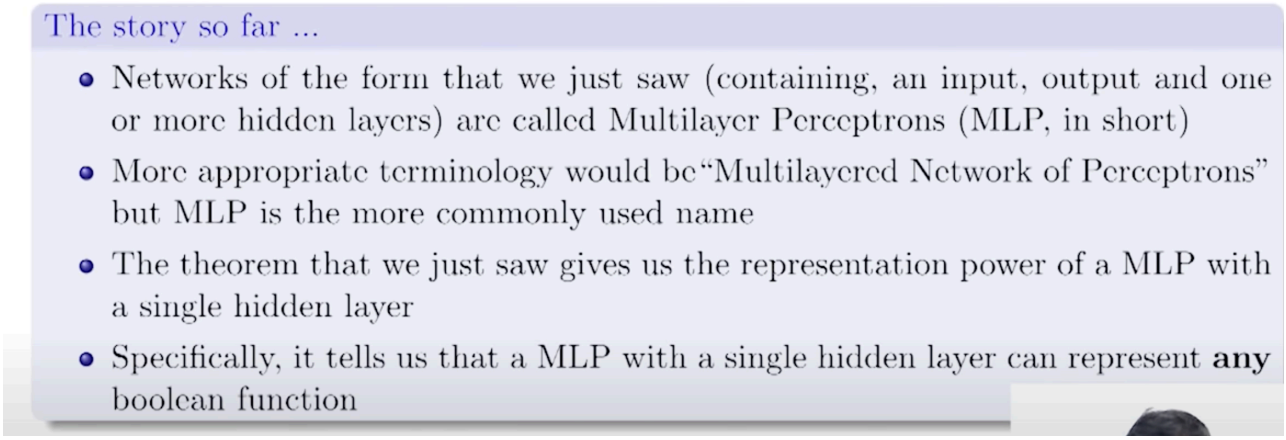
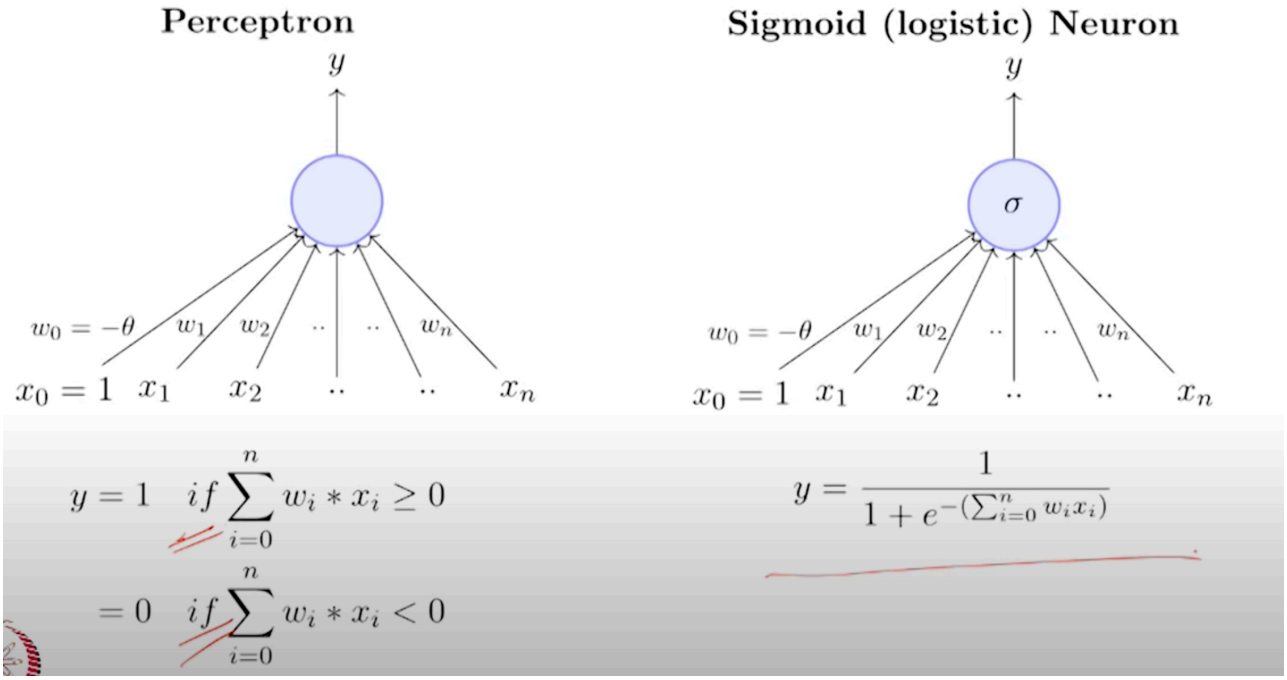
# Dl bASICS0

Wednesday, 20 March 2024 10:32 AM

References:
[Deep Learning(CS7015): Lec 3.3 Learning Parameters: (Infeasible) guess work]
https://arxiv.org/pdf/2009.05673.pdf
https://rattibha.com/storage/pdf/173716_1.pdf

1)

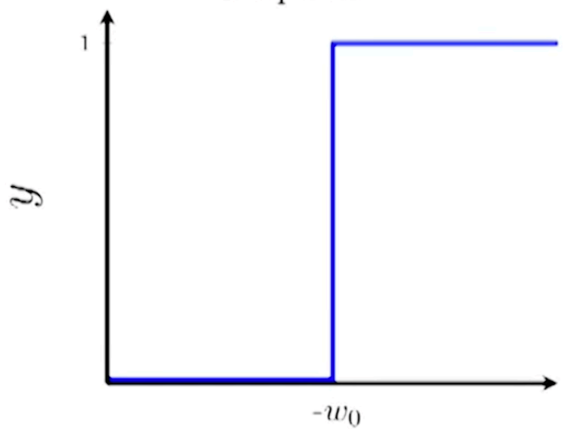



PERCEPTRON(step function) is super harsh function so we are checking Sigmoid. If we increa
get perceptron.

se w in sigmoid then we can

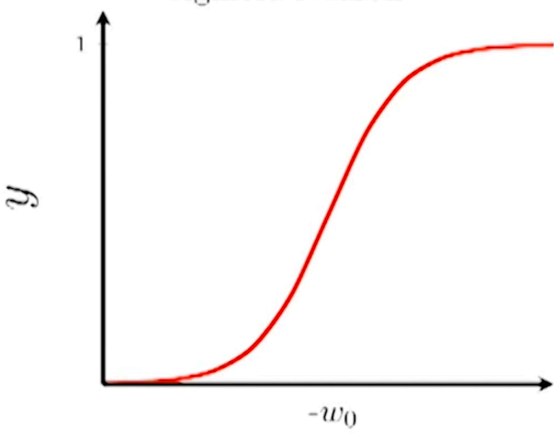
Perceptron




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

Not smooth, not continuous (at $w0$), **not differentiable**

Sigmoid Neuron




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

Smooth, continuous, **differentiable**

As an illustration, consider our movie example

- **Data:** $\{x_i = movie, y_i = like/dislike\}_{i=1}^{n}$
- **Model:** Our approximation of the relation between $\mathbf{x}$ and $y$ (the probability of liking a movie).

$$\hat{y} = \frac{1}{1 + e^{-(\mathbf{w^T x})}}$$

- **Parameter:** $w$
- **Learning algorithm:** Gradient Descent [we will see soon]
- **Objective/Loss/Error function:** One possibility is

$$\mathscr{L}(\mathbf{w}) = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

The learning algorithm should aim to find a $w$ which minimizes the above function (squared error between $y$ and $\hat{y}$)

1) Gradient Descent::

**Gradient Descent Rule**

- The direction $u$ that we intend to move in should be at 180° w.r.t. the gradient
- In other words, move in a direction opposite to the gradient

**Parameter Update Equations**

$$\begin{bmatrix} w_{t+1} \\ b_{t+1} \end{bmatrix} = \begin{bmatrix} w_t \\ b_t \end{bmatrix} - \eta \begin{bmatrix} \nabla w_t \\ \nabla b_t \end{bmatrix}$$

where, $\nabla w_t = \frac{\partial \mathscr{L}(w,b)}{\partial w}$, $\nabla b = \frac{\partial \mathscr{L}(w,b)}{\partial b}$

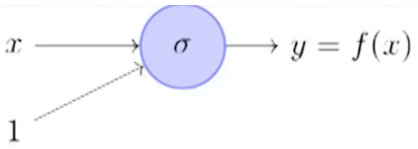
$$\theta_{t+1} = \theta_t + \eta(-\nabla L(\theta))$$

- Let us create an algorithm from this rule ...

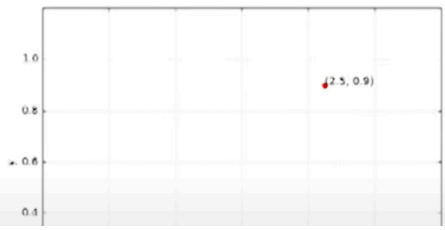
**Algorithm:** gradient_descent()

$t \leftarrow 0$;
$max\_iterations \leftarrow 1000$;
**while** $t < max\_iterations$ **do**
  $w_{t+1} \leftarrow w_t - \eta \nabla w_t$;
  $b_{t+1} \leftarrow b_t - \eta \nabla b_t$;
**end**

- To see this algorithm in practice let us first derive $\nabla w$ and $\nabla b$ for our toy neural network



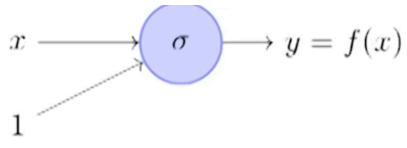

$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$



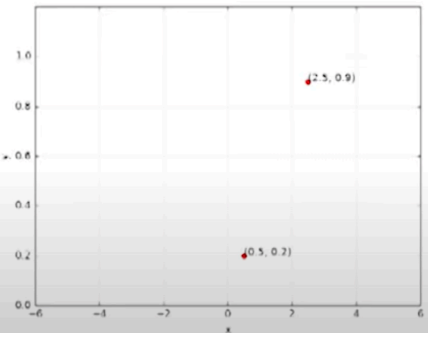

Let's assume there is only 1 point to fit $(x, y)$

$$\mathscr{L}(w, b) = \frac{1}{2} * (f(x) - y)^2$$

$$\nabla w = \frac{\partial \mathscr{L}(w, b)}{\partial w} = \frac{\partial}{\partial w}[\frac{1}{2} * (f(x) - y)^2]$$




$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$




So if there is only 1 point $(x, y)$, we have,

$$\nabla w = (f(x) - y) * f(x) * (1 - f(x)) * x$$

For two points,

$$\nabla w = \sum_{i=1}^{2} (f(x_i) - y_i) * f(x_i) * (1 - f(x_i)) * x_i$$

$$\nabla b = \sum_{i=1}^{2} (f(x_i) - y_i) * f(x_i) * (1 - f(x_i))$$

```
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w,b,x) : #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x + b)))

def error (w, b) :
    err = 0.0
    for x,y in zip(X,Y) :
        fx = f(w,b,x)
        err += 0.5 * (fx - y) ** 2
    return err

def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

def grad_w(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx) * x

def do_gradient_descent() :
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs) :
        dw, db = 0, 0
        for x,y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

3)

Universal Approximation Theorem:

**Representation power of a multilayer network of perceptrons**

A multilayer network of perceptrons with a single hidden layer can be used to represent any boolean function precisely (no errors)

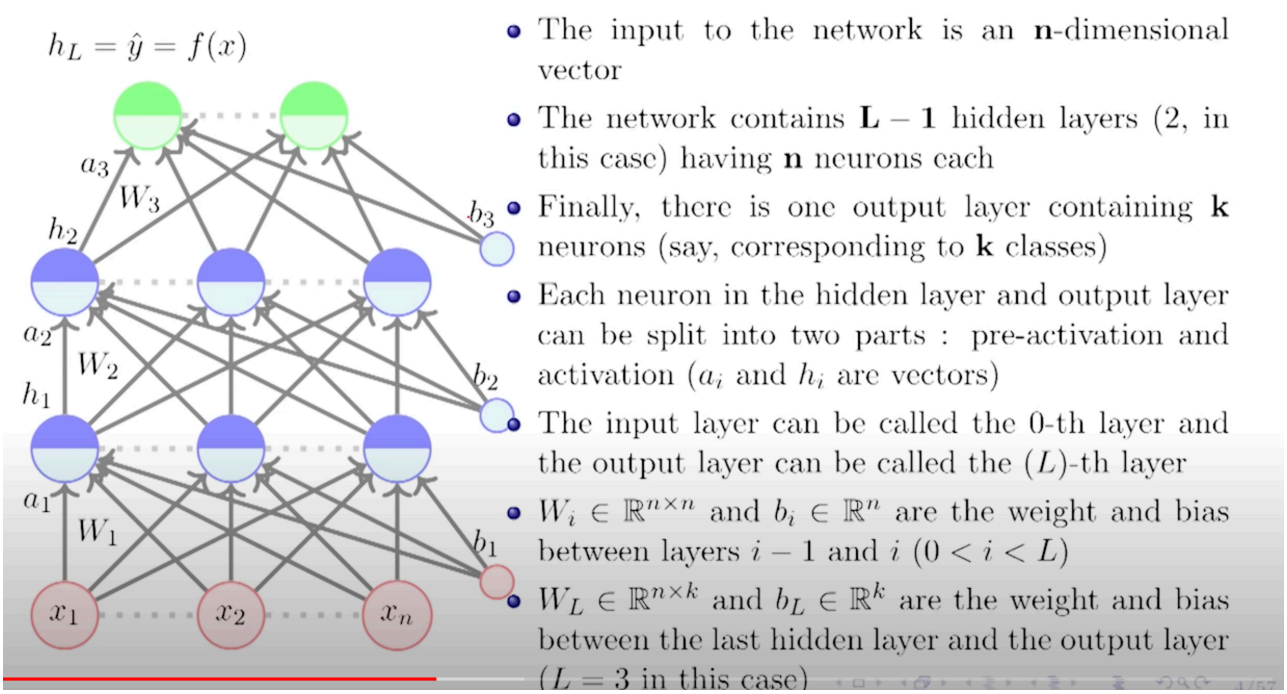
**Representation power of a multilayer network of sigmoid neurons**

A multilayer network of neurons with a single hidden layer can be used to approximate any continuous function to any desired precision

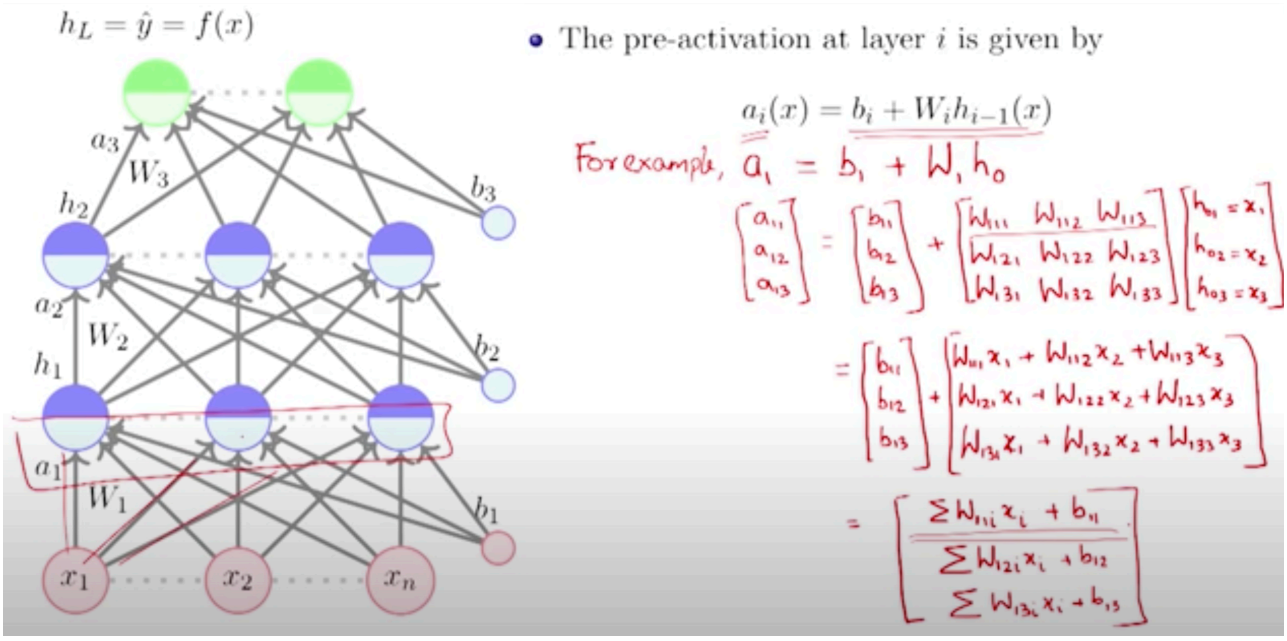
In other words, there is a guarantee that for any function $f(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m$, we can always find a neural network (with 1 hidden layer containing enough neurons) whose output g(x) satisfies $|g(x) - f(x)| <$

4)FFN:

Pre-activation ==> summation of all ips and giving it to activation function




- The input to the network is an **n**-dimensional vector
- The network contains $\mathbf{L-1}$ hidden layers (2, in this case) having **n** neurons each
- Finally, there is one output layer containing **k** neurons (say, corresponding to **k** classes)
- Each neuron in the hidden layer and output layer can be split into two parts : pre-activation and activation ($a_i$ and $h_i$ are vectors)
- The input layer can be called the 0-th layer and the output layer can be called the $(L)$-th layer
- $W_i \in \mathbb{R}^{n \times n}$ and $b_i \in \mathbb{R}^n$ are the weight and bias between layers $i-1$ and $i$ $(0 < i < L)$
- $W_L \in \mathbb{R}^{n \times k}$ and $b_L \in \mathbb{R}^k$ are the weight and bias between the last hidden layer and the output layer $(L = 3$ in this case)




- The pre-activation at layer $i$ is given by

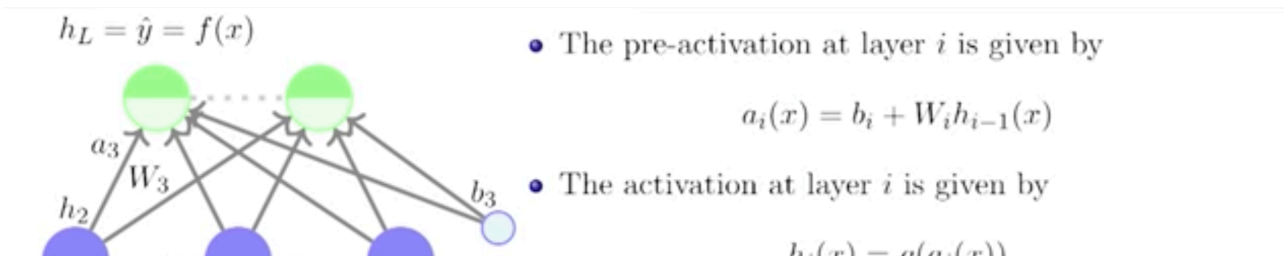
$$a_i(x) = b_i + W_i h_{i-1}(x)$$

For example, $a_1 = b_1 + W_1 h_0$

$$\begin{bmatrix} a_{11} \\ a_{12} \\ a_{13} \end{bmatrix} = \begin{bmatrix} b_{11} \\ b_{12} \\ b_{13} \end{bmatrix} + \begin{bmatrix} W_{111} & W_{112} & W_{113} \\ W_{121} & W_{122} & W_{123} \\ W_{131} & W_{132} & W_{133} \end{bmatrix} \begin{bmatrix} h_{01} = x_1 \\ h_{02} = x_2 \\ h_{03} = x_3 \end{bmatrix}$$

$$= \begin{bmatrix} b_{11} \\ b_{12} \\ b_{13} \end{bmatrix} + \begin{bmatrix} W_{111}x_1 + W_{112}x_2 + W_{113}x_3 \\ W_{121}x_1 + W_{122}x_2 + W_{123}x_3 \\ W_{131}x_1 + W_{132}x_2 + W_{133}x_3 \end{bmatrix}$$

$$= \begin{bmatrix} \sum W_{11i}x_i + b_{11} \\ \sum W_{12i}x_i + b_{12} \\ \sum W_{13i}x_i + b_{13} \end{bmatrix}$$

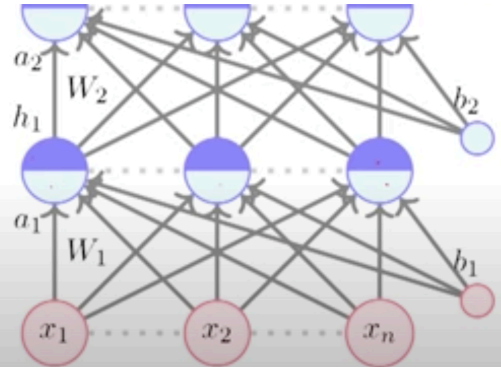



- The pre-activation at layer $i$ is given by

$$a_i(x) = b_i + W_i h_{i-1}(x)$$

- The activation at layer $i$ is given by

$$h_i(x) = g(a_i(x))$$

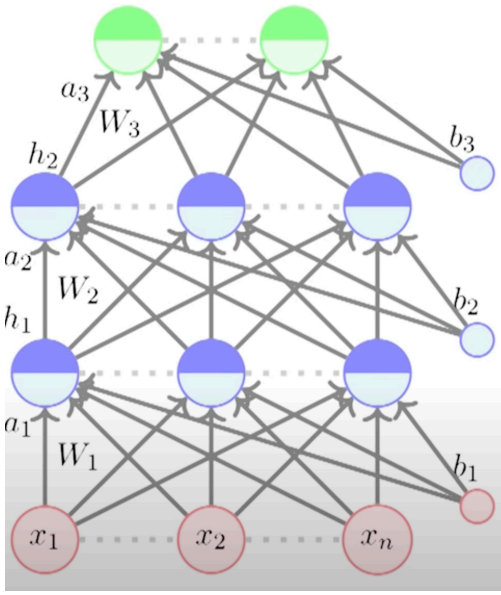



$h_i(x) = g(a_i(x))$

$$\begin{bmatrix} h_{i1} \\ h_{i2} \\ h_{i3} \end{bmatrix} = g\left(\begin{bmatrix} a_{i1} \\ a_{i2} \\ a_{i3} \end{bmatrix}\right) = \begin{bmatrix} g(a_{i1}) \\ g(a_{i2}) \\ g(a_{i3}) \end{bmatrix}$$

eg. $g(a_{i3}) = \sigma(a_{i3}) = \frac{1}{1+e^{-a_{i3}}}$

$h_L = \hat{y} = f(x)$




- The pre-activation at layer $i$ is given by

$$a_i = b_i + W_i h_{i-1}$$

- The activation at layer $i$ is given by

$$h_i = g(a_i)$$

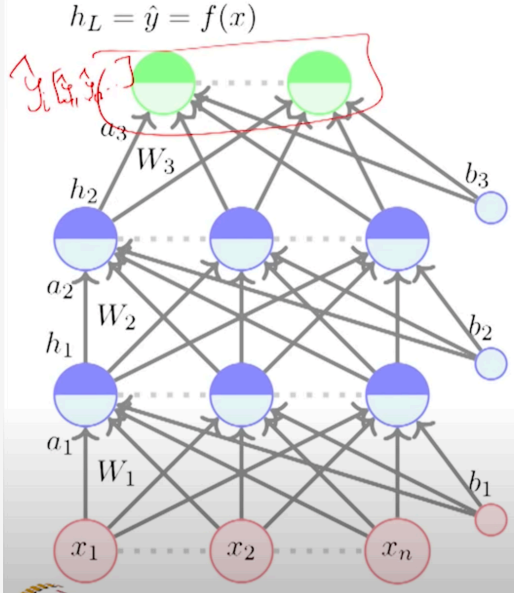
where $g$ is called the activation function (for example, logistic, tanh, linear, *etc.*)

- The activation at layer $i$ is given by

$$f(x) = h_L = O(a_L)$$

where $O$ is the output activation function (for example, softmax, linear, *etc.*)

$h_L = \hat{y} = f(x)$




- **Data:** $\{x_i, y_i\}_{i=1}^N$
- **Model:**

$$\hat{y}_i = f(x_i) = O(W^3 g(W^2 g(W^1 x + b_1) + b_2) + b_3)$$

- **Parameters:** $\theta = W_1, .., W_L, b_1, b_2, ..., b_L (L = 3)$
- **Algorithm:** Gradient Descent with Backpropagation (we will see soon)
- **Objective/Loss/Error function:** Say,

$$min \frac{1}{N} \sum_{i=}^{N}$$

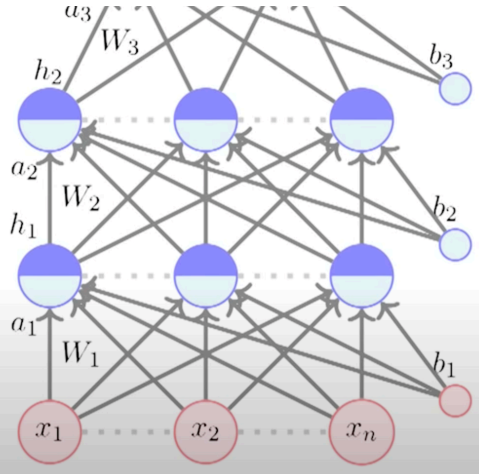
*In general, min* $\mathscr{L}(\theta$

where $\mathscr{L}(\theta)$ is some function

$h_L = \hat{y} = f(x)$

- Recall our gradient descent algorithm
- We can write it more concisely as

**Algorithm:** gradient_descent()




**Algorithm:** gradient_descent()

$t \leftarrow 0$;
$max\_iterations \leftarrow 1000$;
$Initialize \quad \theta_0 = [W_1^0, ..., W_L^0, b_1^0, ..., b_L^0]$;
**while** $t{+}{+} < max\_iterations$ **do**
  $\theta_{t+1} \leftarrow \theta_t - \eta \nabla \theta_t$;
**end**

- where $\nabla \theta_t = \left[\frac{\partial \mathscr{L}(\theta)}{\partial w_t}, \frac{\partial \mathscr{L}(\theta)}{\partial b_t}\right]^T$
- Now, in this feedforward neural network, instead of $\theta = [w, b]$ we have $\theta = W_1, W_2, .., W_L, b_1, b_2, .., b_L$
- We can still use the same algorithm for learning the parameters of our model

**We need to answer two questions**

- How to choose the loss function $\mathscr{L}(\theta)$?
- How to compute $\nabla\theta$ which is composed of $\nabla W_1, \nabla W_2, ..., \nabla W_{L-1} \in \mathbb{R}^{n \times n}, \nabla W_L \in \mathbb{R}^{n \times k}$,
  $\nabla b_1, \nabla b_2, ..., \nabla b_{L-1} \in \mathbb{R}^n$ and $\nabla b_L \in \mathbb{R}^k$ ?